LỜI NÓI ĐẦU

"Cấu trúc dữ liệu và giải thuật" là môn học chính yếu và là kiến thức nền tảng của chuyên ngành Công nghệ Thông tin. Nhằm xây dựng một giáo trình vừa đảm bảo tính chuẩn mực của sách giáo khoa, vừa đáp ứng nhu cầu thực hành của sinh viên. Chúng tôi đã tham khảo nhiều tài liệu có giá trị của nhiều tác giả trong và ngoài nước nhằm cung cấp kiến thức về môn học "Cấu trúc dữ liệu và giải thuật" một cách có hệ thống, nhiều vấn đề được minh họa trực quan và hướng dẫn theo từng bước lập trình cụ thể cho sinh viên.

Giáo trình giới thiệu các cấu trúc dữ liệu (từ cơ bản đến phức tạp) và các giải thuật trên các cấu trúc này. Các giải thuật được trình bày theo các cấu trúc điều khiển chuẩn và được minh họa bằng Ngôn ngữ Lập trình C.

Giáo trình bao gồm 6 chương và 1 phụ lục:

- **Chương 1** trình bày về tầm quan trọng của cấu trúc dữ liệu và giải thuật trong một đề án tin học; Đánh giá cấu trúc dữ liệu và giải thuật cũng như tổng quan về đánh giá độ phức tạp giải thuật. Bên cạnh đó trình bày các kiểu dữ liệu thường gặp trong lập trình.
- **Chương 2** trình bày một số giải thuật tìm kiếm và sắp xếp nội (bộ nhớ trong). Đánh giá độ phức tạp của các giải thuật tìm kiếm và sắp xếp nội.
- **Chương 3** trình bày khái niệm về các loại danh sách liên kết (cấu trúc dữ liệu động) cùng với cách tổ chức, các thao tác và giải thuật, ứng dụng của nó.
- **Chương 4** trình bày các khái niệm về cấu trúc danh sách hạn chế. Ngăn xếp, hàng đợi cùng với cách tổ chức, các thao tác và giải thuật, ứng dụng của nó.
- **Chương 5** trình bày các khái niệm về cấu trúc cây. Cây nhị phân, cây nhị phân tìm kiếm cùng với cách tổ chức, các thao tác và giải thuật, ứng dụng. Bên cạnh đó cũng trình bày cấu trúc dữ liệu cây nhị phân tìm kiếm cân bằng, các thao tác cân bằng lại khi cây bị mất cân bằng.
- **Chương 6** trình bày các khái niệm về Bảng băm (hash Table), cùng với cách tổ chức, các thao tác và giải thuật của một số phương pháp băm cơ bản.
- **Phần phụ lục** trình bày một số kiến thức cơ bản nhằm giúp sinh viên hệ thống lại những nội dung đã học như: Cách phát sinh dữ liệu ngẫu nhiên, các bước để giải một bài toán tin học, đệ quy, khử đệ quy, …

Mặc dù rất nhiều cố gắng nhưng chắc chắn không thể tránh khỏi những thiếu sót. Tác giả rất mong nhận được những ý kiến đóng góp và phê bình quí báu của các bạn đọc để cuốn giáo trình này ngày càng được hoàn thiện hơn.

Tác giả

MỤC LỤC

Chương 1. TỔNG QUAN

1.1. Tầm quan trọng của cấu trúc dữ liệu và giải thuật trong một đề án tin học

1.1.1. Vai trò của cấu trúc dữ liệu

Có thể nói rằng không có một chương trình máy tính nào mà không có dữ liệu để xử lý. Dữ liệu có thể là dữ liệu đưa vào (input data), dữ liệu trung gian hoặc dữ liệu đưa ra (output data). Việc tổ chức để lưu trữ dữ liệu phục vụ cho chương trình có ý nghĩa rất quan trọng trong toàn bộ hệ thống chương trình. Việc xây dựng cấu trúc dữ liệu quyết định rất lớn đến chất lượng cũng như công sức của người lập trình trong việc thiết kế, cài đặt chương trình. Như vậy, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề:

– **Tổ chức biểu diễn các đối tượng thực tế:**

Các đối tượng dữ liệu thực tế rất đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức, xây dựng các cấu trúc thích hợp sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế đó, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng *cấu trúc dữ liệu* cho bài toán.

– **Xây dựng các thao tác xử lý dữ liệu:**

Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải tác động lên dữ liệu để cho ra kết quả mong muốn, đây là bước xây dựng *giải thuật* cho bài toán.

Ví dụ: Một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 4 sinh viên. Do mỗi sinh viên có ba điểm số ứng với ba môn học khác nhau nên dữ liệu có dạng bảng như sau:

Bảng 1-1: Bảng điểm sinh viên

Sinh viên	Môn 1	Môn 2	Môn 3
SV1	8	6	4
SV2	9	5	3
SV3	6	7	2
SV4	5	6	5

Chỉ xét thao tác xử lý là xuất điểm số các môn học của từng sinh viên. Giả sử có các phương án tổ chức lưu trữ sau:

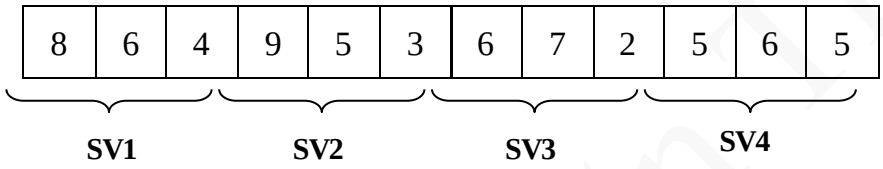
– **Phương án 1:** Sử dụng mảng một chiều

Có tất cả 4(sv) x 3(môn) = 12 điểm số cần lưu trữ, do đó khai báo mảng **diem** như sau:

```
int diem [12] = { 8   6   4
                  9   5   3
                  6   7   2
                  5   6   5 };
```

Khi đó trong mảng **diem** các phần tử sẽ được lưu trữ như sau:

Bảng 1-2: Mảng một chiều chứa điểm của sinh viên



Và truy xuất điểm số môn j của sinh viên i (là phần tử tại dòng i, cột j trong bảng) - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng **diem**:

BảngĐiểm(dòng i, cột j) → **diem**[((i - 1) * số cột) + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định như sau:

diem[i] → **diem**[((i - 1) * số cột) + j]

Ở phương án này, thao tác xử lý được cài đặt như sau:

```
1.  void inDiem()
2.  {
3.      int somon = 3;
4.      int sv, mon;
5.      for(int i = 0; i < 12; i++)
6.      {
7.          sv = i / somon;
8.          mon = i % somon + 1;
9.          if(mon == 1)
10.             printf("\nĐiểm của SV thứ %d: \n", sv);
11.         printf("\tĐiểm môn %d là: %d \n", mon, diem[i]);
12.     }
13. }
```

– **Phương án 2:** Sử dụng mảng hai chiều:

Khai báo mảng hai chiều **diem** có kích thước (**3 cột × 4 dòng)** như sau:

```
int diem [12] = { {8    6    4},
```

{9 1 3},
{6 7 2},
{5 6 5} };

Khi đó trong mảng **diem** các phần tử sẽ được lưu trữ như sau:

Bảng 1-3: Mảng hai chiều chứa điểm của sinh viên

	cột 0	**cột 1**	**cột 2**
Dòng 0	diem[0][0] = 8	diem[0][1] = 6	diem[0][2] = 4
Dòng 1	diem[1][0] = 9	diem[1][1] = 1	diem[1][2] = 3
Dòng 2	diem[2][0] = 6	diem[2][1] = 7	diem[2][2] = 2
Dòng 3	diem[3][0] = 5	diem[3][1] = 6	diem[3][2] = 5

Như vậy truy xuất điểm số môn j của sinh viên i là phần tử tại dòng i cột j trong bảng – cũng chính là phần tử nằm ở vị trí dòng i cột j trong mảng:

BảngĐiểm(dòng i, cột j) → **diem**[i][j]

Ở phương án này, thao tác xử lý được cài đặt như sau:

```
1.  void inDiem()
2.  {
3.      int somon = 3, sosv = 4;
4.      for(int i = 0; i < sosv; i++)
5.      {
6.          printf("\nĐiểm của SV thứ %d: \n", i);
7.          for(int j = 0; j < somon; j++)
8.              printf("\tĐiểm môn %d là: %d \n", j, diem[i][j]);
9.      }
10. }
```

Nhận xét: Ta có thể thấy rằng phương án 2 cung cấp một cấu trúc dữ liệu phù hợp với dữ liệu thực tế hơn phương án 1, do đó giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản và tự nhiên hơn.

Tuy nhiên, khi giải quyết một bài toán trên máy tính, chúng ta thường chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn các đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật.

➔Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau.

1.1.2. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật

Trong một đề án tin học, cấu trúc dữ liệu và giải thuật mối quan hệ với nhau, chúng được thể hiện qua công thức nổi tiếng của nhà toán học người Thụy sĩ **Niklaus Wirth** như sau:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Như vậy, khi đã có cấu trúc dữ liệu tốt, nắm vững giải thuật thực hiện thì việc thể hiện chương trình bằng một ngôn ngữ cụ thể chỉ là vấn đề thời gian. Khi có cấu trúc dữ liệu mà chưa tìm ra giải thuật thì không thể có chương trình và ngược lại không thể có giải thuật khi chưa có cấu trúc dữ liệu. Một chương trình máy tính chỉ có thể được hoàn thiện khi có đầy đủ cả cấu trúc dữ liệu để lưu trữ dữ liệu và giải thuật xử lý dữ liệu theo yêu cầu của bài toán đặt ra.

Lưu ý: Giải thuật có khi còn được gọi là thuật giải hay thuật toán.

1.1.3. Xây dựng giải thuật

1.1.3.1. Khái niệm giải thuật

Giải thuật là một chương trình được thiết kế để giải quyết yêu cầu bài toán đặt ra.

Giải thuật hay thuật toán dùng để chỉ phương pháp hay cách thức (method) để giải quyết vấn đề. Giải thuật có thể được minh họa bằng ngôn ngữ tự nhiên (natural language), bằng sơ đồ (flow chart) hoặc bằng mã giả (pseudo code). Trong thực tế, giải thuật thường được minh họa hay thể hiện bằng mã giả dựa trên một hay một số ngôn ngữ lập trình nào đó (thường là ngôn ngữ mà người lập trình chọn để cài đặt giải thuật), chẳng hạn như C/C++, Pascal, …

Khi đã xác định được cấu trúc dữ liệu thích hợp, người lập trình sẽ bắt đầu tiến hành xây dựng giải thuật tương ứng theo yêu cầu của bài toán đặt ra trên cơ sở của cấu trúc dữ liệu đã được chọn. Để giải quyết một vấn đề có thể có nhiều phương pháp, do vậy sự lựa chọn phương pháp phù hợp là một việc mà người lập trình phải cân nhắc và tính toán. Sự lựa chọn này cũng có thể góp phần đáng kể trong việc giảm bớt công việc của người lập trình trong phần cài đặt giải thuật trên một ngôn ngữ cụ thể.

1.1.3.2. Tính chất giải thuật

Giải thuật có 5 tính chất quan trọng sau: Dữ liệu, Tính chính xác, Tính dừng, Tính đúng đắn và Tính khả thi.

1.1.3.2.1. Dữ liệu

Ví dụ: Cho một mảng a có n phần tử kiểu nguyên. Hãy sắp xếp mảng a tăng dần.

- Dữ liệu gồm: Dữ liệu vào mảng a chưa sắp xếp thứ tự, có n phần tử.
- Dữ liệu ra: Mảng a đã có thứ tự.

1.1.3.2.2. Tính xác định

Ví dụ: Với *bài toán* trên.

Tính xác định được thể hiện ở việc xác định được giải thuật nào áp dụng vào bài toán trên.

1.1.3.2.3. Tính dừng

Ví dụ: Với *bài toán* trên.

```
1.  void sapXep_TangDan(ItemType a[ ], int n)
2.  {
3.      for(int i = 0; i < n - 1; i++)
4.          for(int j = i + 1; j < n;)
5.          {
6.              if(a[j] < a[i])
7.                  hoanVi(a[i], a[j]);
8.          }
9.  }
```

Giải thuật trên lặp vô tận; không cho ra kết quả. Do không tăng biến j.

1.1.3.2.4. Tính đúng đắn

Ví dụ: Với *bài toán* trên.

```
1.  void sapXep_TangDan(ItemType a[ ], int n)
2.  {
3.      for(int i = 0; i < n - 1; i++)
4.      {
5.          for(int j = i + 2; j < n; j++)
6.          {
7.              if(a[i] > a[j])
8.                  hoanVi(a[i], a[j]);
9.          }
10.     }
11. }
```

Bài toán này không đúng đắn thể hiện ở việc gắn j = i + 2.

1.1.3.2.5. Tính khả thi

Ví dụ: Với *bài toán* trên.

```
1.  void sapXep_TangDan(ItemType a[ ], int n)
2.  {
3.      ItemType min;
```

```
4.      for(int i = 0; i < n - 1; i++)
5.      {
6.          min = a[i];
7.          for(int j = i + 1; j < n; j++)
8.              if(min > a[j])
9.                  min = a[j];
10.         printf("%4d", min);
11.     }
12. }
```

Giải thuật trên không khả thi, tuy viết giải thuật có in ra kết quả sắp xếp tăng dần nhưng không trả về mảng a sắp xếp tăng dần.

1.2. Đánh giá cấu trúc dữ liệu và giải thuật

1.2.1. Các tiêu chuẩn đánh giá cấu trúc dữ liệu

Qua phần trên ta đã thấy được vai trò và tầm quan trọng của việc lựa chọn một phương án tổ chức dữ liệu thích hợp trong một chương trình hay một đề án tin học. Một cấu trúc dữ liệu tốt phải thoả mãn các tiêu chuẩn sau:

- Phản ánh đúng thực tế của bài toán đặt ra.
- Phù hợp với các thao tác xử lý (*dễ dàng trong việc thao tác dữ liệu*).
- Tiết kiệm tài nguyên (*CPU, bộ nhớ trong, bộ nhớ ngoài, thời gian,...*).

1.2.1.1. Phản ánh đúng thực tế

Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ: Một số trường hợp chọn cấu trúc dữ liệu sai:

- Chọn một số nguyên **int** để lưu trữ điểm trung bình của sinh viên (được tính theo công thức trung bình cộng của các môn học có hệ số), như vậy sẽ làm tròn mọi điểm số của sinh viên gây ra việc đánh giá sinh viên không chính xác qua điểm số. Trong trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế cũng như phản ánh chính xác kết quả học tập của sinh viên.
- Trong trường phổ thông, một lớp có 50 học sinh, mỗi tháng đóng quỹ lớp 1000 đồng. Nếu chọn một biến số kiểu **unsigned int** (khả năng lưu trữ 0 – 65535) để lưu trữ tổng tiền quỹ của lớp học trong tháng, nếu xảy ra trường hợp trong hai tháng liên tiếp không có chi hoặc tăng tiền đóng quỹ của mỗi học sinh lên 2000 đồng thì tổng qũy lớp thu được là 100000 đồng, vượt khỏi khả năng lưu trữ của biến đã chọn, gây nên tình trạng tràn, sai lệnh. Như vậy khi chọn biến dữ liệu ta phải tính đến các trường hợp phát triển của đại lượng chứa trong biến để chọn

liệu dữ liệu thích hợp. Trong trường hợp trên ta có thể chọn kiểu dữ liệu **long** (có kích thước 4 bytes, khả năng lưu trữ là -2147483648 → 2147483647) hoặc chọn kiểu dữ liệu **unsigned long** (có kích thước 4 bytes, khả năng lưu trữ là 0 → 4294967293) để lưu trữ tổng tiền quỹ lớp.

1.2.1.2. Phù hợp với các thao tác xử lý

Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: Phát triển các giải thuật đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ: Một số trường hợp chọn cấu trúc dữ liệu không phù hợp.

Khi cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xóa, sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

1.2.1.3. Tiết kiệm tài nguyên hệ thống

Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có hai loại tài nguyên cần lưu tâm nhất là CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tuỳ vào tình huống cụ thể khi thực hiện đề án. Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì chọn cấu trúc dữ liệu có yếu tố tiết kiệm thời gian xử lý ưu tiên hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ: Một số trường hợp chọn cấu trúc dữ liệu gây lãng phí.

- Sử dụng biến kiểu **int (**2 bytes) để lưu trữ một giá trị thông tin về ngày trong tháng. Nhưng vì số ngày của một tháng chỉ có thể nhận các giá trị từ 1 đến 31 nên chỉ cần sử dụng biến kiểu **char** (1 byte) là đủ.
- Để lưu trữ danh sách nhân viên trong công ty mà sử dụng mảng 1000 phần tử. Nếu số lượng nhân viên thật sự ít hơn 1000 (bị giảm hoặc biên chế không đủ) thì sẽ gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng – ví dụ danh sách liên kết.

1.2.2. Đánh giá độ phức tạp của giải thuật

Việc đánh giá độ phức tạp của một giải thuật quả không dễ dàng chút nào. Như vậy, làm thế nào để chọn được giải thuật tốt? Một giải thuật được gọi là tốt khi nói đến tính hiệu quả của nó. Người ta thường ước lượng thời gian thực hiện giải thuật T(n) để có thể có sự so sánh tương đối giữa các giải thuật với nhau. Trong thực tế, thời gian thực hiện một giải thuật còn phụ thuộc rất nhiều vào các điều kiện khác như cấu tạo của máy

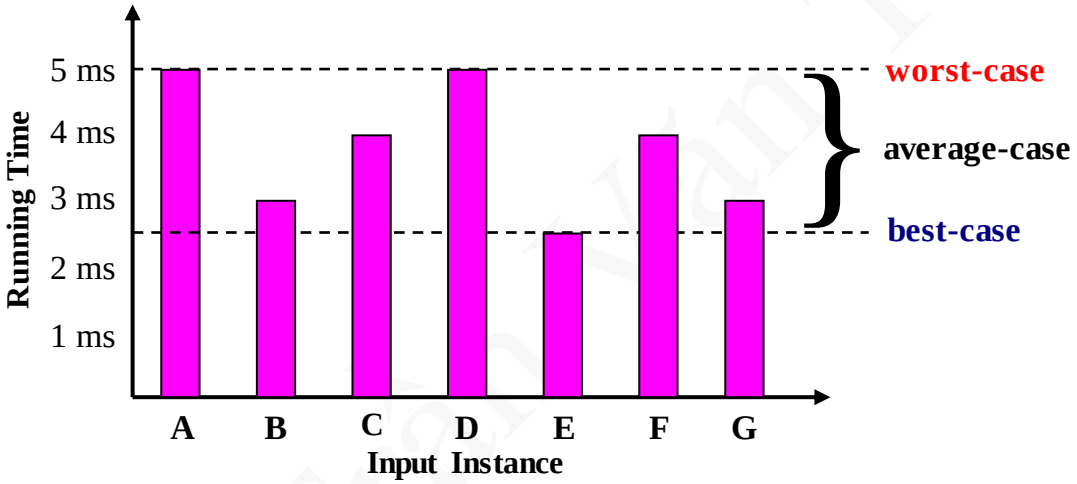
tính, dữ liệu đưa vào, …, ở đây chúng ta chỉ xem xét trên mức độ của lượng dữ liệu đưa vào ban đầu cho giải thuật thực hiện.

Để ước lượng thời gian thực hiện giải thuật chúng ta có thể xem xét trong hai trường hợp:

- Trong trường hợp tốt nhất: **Tmin**
- Trong trường hợp xấu nhất: **Tmax**

➔Từ đó chúng ta có thể ước lượng thời gian thực hiện trung bình của giải thuật: **Tavg**

Ta có thể đánh giá giải thuật bằng phương pháp thực nghiệm thông qua việc cài đặt giải thuật rồi chọn các bộ dữ liệu thử nghiệm. Thống kê các thông số nhận được khi chạy các dữ liệu này ta sẽ có một đánh giá về giải thuật.



Hình 1.1: Ước lượng thời gian thực hiện giải thuật

Tuy nhiên, phương pháp thực nghiệm có một số nhược điểm sau khiến nó khó có khả năng áp dụng trên thực tế:

- Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên giải thuật sẽ chịu sự hạn chế của ngôn ngữ lập trình này.
- Hiệu quả của giải thuật sẽ bị ảnh hưởng bởi trình độ của người cài đặt.
- Việc chọn được các bộ dữ liệu thử nghiệm đặc trưng cho tất cả tập các dữ liệu vào của giải thuật là rất khó khăn và tốn nhiều chi phí.
- Các số liệu thu nhận được phụ thuộc nhiều vào phần cứng mà giải thuật được thử nghiệm trên đó, điều này khiến cho việc so sánh các giải thuật khó khăn nếu chúng được thử nghiệm ở những nơi khác nhau.

Vì những lý do trên, người ta đã tìm kiếm những phương pháp đánh giá giải thuật hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như

vậy là phương pháp đánh giá giải thuật theo hướng xấp xỉ tiệm cận qua các khái niệm toán học O lớn - O(), o nhỏ - o(), Ω(), ≡().

Thông thường các vấn đề cần giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) sẽ được gọi là N. Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết để giải quyết vấn đề) như một hàm số theo N. Người ta quan tâm đến **trường hợp trung bình**, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, **trường hợp tốt nhất** tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp tốt nhất và **trường hợp xấu nhất** tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể.

Việc xác định chi phí trong trường hợp trung bình thường được quan tâm nhiều nhất vì nó đại diện cho đa số trường hợp sử dụng giải thuật. Tuy nhiên, việc xác định chi phí trung bình này lại gặp nhiều khó khăn. Vì vậy, trong nhiều trường hợp, người ta xác định chi phí trong trường hợp xấu nhất (chặn trên) thay cho việc xác định chi phí trong trường hợp trung bình. Hơn nữa, trong một số bài toán, việc xác định chi phí trong trường hợp xấu nhất là rất quan trọng. Ví dụ: Các bài toán trong ngành hàng không, phẫu thuật, …

1.2.2.1. Các bước phân tích bài toán

- **Bước 1:** Trong việc phân tích một giải thuật là xác định đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của giải thuật và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của giải thuật. Chúng ta không thể đạt tới điều lý tưởng này cho bất kỳ một giải thuật không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến bao đóng của thống kê về tính năng của giải thuật bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chặn trên" bất chấp dữ liệu nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".
- **Bước 2:** Trong phân tích một giải thuật là nhận ra các thao tác trừu tượng của giải thuật để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một giải thuật sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của giải thuật, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt này cho phép chúng ta so sánh các giải thuật một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.
- **Bước 3:** Trong quá trình phân tích giải thuật là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một chặn trên cho thời gian chạy chương trình, vấn đề là phải tìm ra một chặn trên tốt nhất, tức là thời gian chạy chương trình khi gặp dữ liệu nhập của trường hợp xấu nhất. Trường

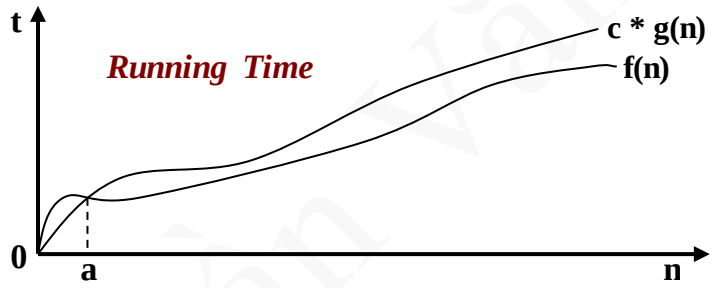
hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích giải thuật dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một giải thuật thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng toán học trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thýờng thích ýớc lýợng ðể tránh sa vào chi tiết.

1.2.2.2. Đánh giá độ phức tạp

1.2.2.2.1. Khái niệm "O"

Cho f(n) và g(n) ta nói: $f(n) \in O(g(n))$ khi và chỉ khi $f(n) \leq c * g(n)$, $\forall n \geq a$ với c và a là hằng số, và f(n), g(n) là các hàm trên miền số tự nhiên.



Hình 1.2: Biểu đồ thể hiện 2 hàm f và g

Chú ý: "O" có nghĩa là **Big - O** notation.

Ví dụ:

- $1.000.001 \leq c * 1.000.000$, với $c \in \mathbf{R}$ do đó ta chọn được một giá trị cho c = 2 sao cho biểu thức trên đúng.
- $3n^2 \leq c * n^2$, với $c \in \mathbf{R}$ do đó ta chọn được một giá trị cho c = 4 sao cho biểu thức trên đúng.
- $7n + 5 \leq c*n$, với $c \in \mathbf{R}$ do đó ta chọn được một giá trị cho c = 8 sao cho biểu thức trên đúng với n là những giá trị đủ lớn.

1.2.2.2.2. Quy tắc xác định O(?)

Xét thành phần có bậc cao nhất của f với f là đa thức tuyến tính.

Ví dụ:

- $(4n^2 + 8n - 4) \in O(n^2)$.

– $(4n^3\log(n) + 4n^3 - 1) \in O(n^3\log(n))$.

1.2.2.2.3. Ví dụ về cách đánh giá giải thuật

Cho một mảng a gồm n phần tử.

```
1. void temp(ItemType a[ ], int n, ItemType x)
2. {
3.       for(int i = 0; i < n; i++)
4.            if(a[i] == x)
5.                 a[i] = 1;
6.            else
7.                 a[i] = 0;
8. }
```

Mỗi lần i lặp:

– Số phép so sánh của khối lệnh con trong vòng lặp for là 1.

– Số phép gán của khối lệnh con trong vòng lặp for là 1.

– Do $i = \overline{0..(n-1)}$ nên số phép so sánh của hàm Temp là: $\sum_{i=0}^{n-1}(1+1)+1 = 2n+1$ và số phép gán của hàm Temp là: $\sum_{i=0}^{n-1}(1+1)+1 = 2n+1$.

– Vậy số lần gán và so sánh của giải thuật Temp là: c(2n + 1) + d(2n + 1) = 2(c + d)n + (c + d).

Vậy độ phức tạp của giải thuật là: **O(n)**.

1.2.2.3. Sự phân lớp các giải thuật

Như đã được chú ý ở trên, hầu hết các giải thuật đều có một tham số chính là **n**, thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số **n** có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị, v.v… Hầu hết tất cả các giải thuật trong giáo trình này có thời gian chạy tiệm cận tới một trong các hàm sau:

1) ***Hằng số:*** Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất này thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số.

2) ***log(n):*** Khi thời gian chạy của chương trình là **logarit** tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại này xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn.

   Ví dụ: Khi n = 10 thì $\log_{10}(n)$ là 1; khi n = 100 thì $\log_{10}(n)$ là 2.

   Bất cứ khi nào n được nhân đôi, $\log_{10}(n)$ tăng lên thêm một hằng số.

3) ***n:*** Khi thời gian chạy của một chương trình là **tuyến tính**.

Ví dụ: Khi n là một triệu thì thời gian chạy cũng cỡ một triệu. Khi n được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi.

Đây là tình huống tối ưu cho một giải thuật mà phải xử lý n dữ liệu nhập.

4) ***nlog(n):*** Đây là thời gian chạy tăng dần lên cho các giải thuật mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Chúng ta nói rằng thời gian chạy của giải thuật như thế là nlog(n).

Ví dụ: Khi n là một triệu, $n\log_2(n)$ có lẽ khoảng hai mươi triệu. Khi n được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi.

5) n^2**:** Khi thời gian chạy của một giải thuật là **bậc hai**. Thời gian bình phương thường tăng dần lên trong các giải thuật mà xử lý tất cả các cặp phần tử dữ liệu (có thể là hai vòng lặp lồng nhau).

Ví dụ: Khi n là một ngàn thì thời gian chạy là một triệu. Khi n được nhân đôi thì thời gian chạy tăng lên gấp bốn lần.

6) 2^n**:** Một số ít giải thuật có thời gian chạy lũy thừa, nhưng lại thích hợp trong một số trường hợp thực tế, mặc dù các giải thuật như thế là "không tưởng" để giải các bài toán.

Ví dụ: Khi n là hai mươi thì thời gian chạy là một triệu.

Khi n gấp đôi thì thời gian chạy được nâng lên lũy thừa hai.

1.2.2.4. Phân tích trường hợp trung bình

Một tiếp cận trong việc nghiên cứu tính năng của giải thuật là khảo sát trường hợp trung bình. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của giải thuật: Ví dụ một giải thuật sắp xếp có thể thao tác trên một mảng N số nguyên ngẫu nhiên, hay một giải thuật hình học có thể xử lý n điểm ngẫu nhiên trên

mặt phẳng với các toạ độ nằm giữa 0 và 1. Kế đến là tính toán thời gian thực hiện trung bình của mội chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ thị đó, sau cùng cộng tất cả chúng lại với nhau. Có ít nhất ba khó khăn trong cách tiếp cận này như thảo luận dưới đây:

- **Trước tiên**, là trên một số máy tính rất khó xác định chính xác số lượng thời gian đòi hỏi cho mỗi chỉ thị. Trường hợp xấu nhất thì đại lượng này bị thay đổi và một số lượng lớn các phân tích chi tiết cho một máy tính có thể không thích hợp đối với một máy tính khác. Đây chính là vấn đề mà các nghiên cứu về độ phức tạp tính toán cũng cần phải né tránh.
- **Thứ hai**, chính việc phân tích trường hợp trung bình lại thường là đòi hỏi toán học quá khó. Do tính chất tự nhiên của toán học thì việc chứng minh các chặn trên thì thường ít phức tạp hơn bởi vì không cần sự chính xác. Hiện nay chúng ta chưa biết được tính năng trong trường hợp trung bình của rất nhiều giải thuật.
- **Thứ ba** (và chính là điều quan trọng nhất) trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập thường gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bản tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho giải thuật sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho giải thuật hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các ứng dụng thông thường.

1.3. Kiểu dữ liệu

Máy tính chỉ có thể lưu trữ dữ liệu ở dạng nhị phân sơ cấp. Để phản ánh được dữ liệu thực tế đa dạng và phong phú, cần phải xây dựng các phép ánh xạ, những quy tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là kiểu dữ liệu. Như đã phân tích ở mục trước, giữa hình thức lưu trữ dữ liệu và các thao tác xử lý trên đó có quan hệ mật thiết với nhau, từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau:

1.3.1. Khái niệm về kiểu dữ liệu

Kiểu dữ liệu **T** được xác định bởi một bộ **<V, O>**, với:

- **V**: Tập các giá trị hợp lệ mà một đối tượng kiểu T có thể lưu trữ.
- **O**: Tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.

Mỗi kiểu dữ liệu thường được đại diện bởi một tên (định danh). Mỗi phần tử dữ liệu có kiểu T sẽ có giá trị trong miền V và có thể được thực hiện các phép toán thuộc tập hợp các phép toán trong O.

Để lưu trữ các phần tử dữ liệu này thường phải tốn một số byte(s) trong bộ nhớ, số byte(s) này gọi là kích thước của kiểu dữ liệu.

Ví dụ:

- Giả sử có kiểu dữ liệu **mẫu tự alpha** = <$\mathbf{V_c}$, $\mathbf{O_c}$> với

  $\mathbf{V_c}$ = {a .. z, A .. Z}

  $\mathbf{O_c}$ = {lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa…}

- Giả sử có kiểu dữ liệu **số nguyên** = <$\mathbf{V_i}$, $\mathbf{O_i}$> với

  $\mathbf{V_i}$ = {–32768 .. 32767}

  $\mathbf{O_i}$ = { + , –, *, /, %}

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội dung dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

Các thuộc tính của một kiểu dữ liệu bao gồm:

- Tên kiểu dữ liệu.
- Miền giá trị.
- Kích thước lưu trữ.
- Tập các toán tử tác động lên kiểu dữ liệu.

1.3.2. Các kiểu dữ liệu cơ sở

Hầu hết các ngôn ngữ lập trình đều có cung cấp các kiểu dữ liệu cơ sở. Tùy vào mỗi ngôn ngữ mà các kiểu dữ liệu cơ sở có thể có các tên gọi khác nhau song chung quy lại có những loại kiểu dữ liệu cơ sở như sau:

- **Kiểu số nguyên:** Có thể có dấu hoặc không có dấu và thường có các kích thước sau:
  + Kiểu số nguyên 1 byte.
  + Kiểu số nguyên 2 bytes.
  + Kiểu số nguyên 4 bytes.

➔Kiểu số nguyên thường được thực hiện với các phép toán: **O** = { + , -, *, /, DIV, MOD, <, >, <= , >= , = , …}.

- **Kiểu số thực:** Thường có các kích thước sau:
  + Kiểu số thực 4 bytes.

+ Kiểu số thực 6 bytes.

+ Kiểu số thực 8 bytes.

+ Kiểu số thực 10 bytes.

➔Kiểu số thực thường được thực hiện với các phép toán: **O** = { + , -, *, /, <, >, <= , >= , = , …}.

– **Kiểu ký tự:** Có thể có các kích thước sau:

+ Kiểu ký tự byte.

+ Kiểu ký tự 2 bytes.

➔Kiểu ký tự thường được thực hiện với các phép toán: **O** = { + , -, <, >, <= , >= , = , ORD, CHR, …}.

– **Kiểu chuỗi ký tự:** Có kích thước tùy thuộc vào từng ngôn ngữ lập trình.

➔Kiểu chuỗi ký tự thường được thực hiện với các phép toán: **O** = { + , &, <, >, <= , >= , = , Length, Trunc, …}.

– **Kiểu luận lý:** Thường có kích thước 1 byte.

➔Kiểu luận lý thường được thực hiện với các phép toán: **O** = {NOT, AND, OR, XOR, <, >, <= , >= , = , …}.

1.3.3. Các kiểu dữ liệu có cấu trúc

Kiểu dữ liệu có cấu trúc là các kiểu dữ liệu được xây dựng trên cơ sở các kiểu dữ liệu đã có (cũng có thể là một kiểu dữ liệu có cấu trúc khác). Tùy vào từng ngôn ngữ lập trình song thường có các loại sau:

– **Kiểu mảng (*hay còn gọi là dãy*):** Kích thước bằng tổng kích thước của các phần tử.

– **Kiểu bản ghi (*hay cấu trúc*):** Kích thước bằng tổng kích thước các thành phần (Field).

1.3.3.1. Kiểu mảng

Mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc, thường được lưu trữ liên tiếp nhau trong bộ nhớ. Mảng có thể một chiều hay nhiều chiều. Một dãy số chính là hình tượng của mảng một chiều, ma trận là hình tượng của mảng hai chiều.

Một điều đáng lưu ý là mảng hai chiều có thể coi là mảng một chiều trong đó mỗi phần tử của nó là một mảng một chiều. Tương tự như vậy, một mảng n chiều có thể coi là mảng một chiều trong đó mỗi phần tử là một mảng n - 1 chiều.

Trong ngôn ngữ lập trình C, mảng một chiều được khai báo như sau:

<Kiểu dữ liệu> <Tên biến> [<Số phần tử>];

Ví dụ: Để khai báo một mảng a chứa tối đa 50 số nguyên ta khai báo như sau:

```
int a[50];
```

Có thể vừa khai báo vừa gán giá trị khởi động cho một mảng như sau:

```
int a[5] = {3, 5, 7, -9, 16};
```

Trong trường hợp trên ngôn ngữ C cho phép ta khai báo một cách tiện lợi hơn như sau:

```
int a[ ] = {3, 5, 7, -9, 16};
```

Điều này cho thấy không cần chỉ ra số lượng phần tử cụ thể trong khai báo. Trình biên dịch sẽ tự động là việc này.

Tương tự, có thể khai báo một mảng hai chiều theo cú pháp sau:

<Kiểu dữ liệu> <Tên biến> [<Số dòng>][<Số cột>];

Ví dụ: Có thể khai báo mảng hai chiều như sau:

```
int a[50][40];
```

hay

```
int a[ ][ ] = { {  1,  5,  6,   -7,   4},
                {  -8,   9,  -2,    5,   3},
                {   4,  -6,   7,    3,   1} };
```

(mảng a sẽ có kích thước là 3 × 5)

1.3.3.2. Kiểu chuỗi ký tự

Chuỗi ký tự là một trong các kiểu dữ liệu có cấu trúc đơn giản nhất và thường các ngôn ngữ lập trình đều định nghĩa nó như một kiểu cơ bản. Do tính thông dụng của kiểu chuỗi ký tự các ngôn ngữ lập trình luôn cung cấp sẵn một bộ các hàm thư viện xử lý trên

kiểu dữ liệu này. Trong ngôn ngữ lập trình C, thư viện *string.h* chứa các hàm xử lý chuỗi ký tự rất đa dạng và phong phú.

Chuỗi ký tự trong ngôn ngữ lập trình C được cấu trúc như một chuỗi liên tiếp các ký tự kết thúc bằng ký tự có mã ASCII bằng 0 (NULL character). Như vậy, giới hạn chiều dài của một chuỗi ký tự trong ngôn ngữ lập trình C là một segment (chứa tối đa 65535) ký tự, ký tự đầu tiên được đánh số thứ tự là ký tự thứ 0.

Có thể khai báo một chuỗi ký tự theo một số cách sau đây:

- char s[10]; *// khai báo một chuỗi ký tự có chiều dài tối đa là 10 (kể cả ký tự kết thúc)*
- char s[ ] = "ABC"; *//khai báo một chuỗi ký tự s có chiều dài bằng chiều dài của chuỗi "ABC" và giá trị khởi đầu của S là "ABC"*
- char *s = "ABC"; *//giống cách khai báo trên*

Ở ví dụ trên có thể thấy rằng một hằng chuỗi ký tự được thể hiện bằng một chuỗi ký tự đặt trong cặp dấu nháy kép.

Các thao tác trên chuỗi ký tự rất đa dạng. Sau đây là một số thao tác thông dụng:

1. So sánh hai chuỗi	strcmp
2. Sao chép hai chuỗi	strcpy
3. Kiểm tra một chuỗi nằm trong chuỗi kia	strstr
4. Cắt một từ ra khỏi một chuỗi	strok
5. Đổi một số ra một chuỗi	itoa
6. Đổi một chuỗi ra số nguyên, số thực	atoi, atof
7. Đổi một hay một số giá trị ra chuỗi	sprintf
8. Nhập một chuỗi ký tự	gets
9. Xuất một chuỗi ký tự	puts

1.3.3.3. Kiểu dữ liệu có cấu trúc do người dùng định nghĩa

Trong thực tế các kiểu dữ liệu cơ bản không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc.

Khai báo tổng quát của kiểu struct như sau:

```
struct <tên kiểu struct>
{
    <kiểu dữ liệu 1> <tên trường 1>;
    <kiểu dữ liệu 2> <tên trường 2>;
    ...
```

```
};
```

Ví dụ: Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

- Mã sinh viên: *Chuỗi ký tự.*
- Tên sinh viên: *Chuỗi ký tự.*
- Ngày sinh: *Kiểu ngày tháng*.
- Nơi sinh: *Chuỗi ký tự.*
- Điểm thi: *Số thực.*

Để thể hiện các thông tin về ngày tháng năm sinh cần phải xây dựng một cấu trúc dữ liệu như sau:

```
1. struct Date
2. {
3.       char Ngay[2];
4.       char Thang[2];
5.       char Nam[4];
6. };
```

➔Từ đó có thể xây dựng kiểu dữ liệu thể hiện thông tin mỗi sinh viên:

```
1. struct SinhVien
2. {
3.       char Masv[20];
4.       char Tensv[20];
5.       char Noisinh[20];
6.       Date Ngaysinh;
7.       float Diemthi;
8. };
```

Kiểu cấu trúc bổ sung nhiều thiếu sót của kiểu mảng, giúp cho khả năng thể hiện các đối tượng đa dạng của thế giới hiện thực vào trong máy tính một cách dễ dàng, và chính xác hơn.

1.3.3.4. Kiểu **union**

Kiểu "union" là một dạng cấu trúc dữ liệu đặc biệt của ngôn ngữ lập trình C. Nó rất giống với kiểu cấu trúc. Chỉ khác một điều, trong kiểu union, các trường được phép dùng chung một vùng nhớ, có thể truy xuất dưới các dạng khác nhau.

Khai báo tổng quát của kiểu union như sau:

```
union <tên kiểu union>
{
     <kiểu dữ liệu 1> <tên trường 1>;
     <kiểu dữ liệu 2> <tên trường 2>;
```

```
    <kiểu dữ liệu 3> <tên trường 3>;
    ...
};
```

Ví dụ: Có thể định nghĩa kiểu số như sau:

```
1.  union Number
2.  {
3.      int i;
4.      long l;
5.  };
```

Việc truy xuất đến một trường trong union được thực hiện hoàn toàn giống như trong struct. Giả sử có biến n kiểu Number. Khi đó, n.i là một số kiểu int còn n.l là một số kiểu long, nhưng cả hai đều dùng chung một vùng nhớ, Vì vậy, khi thực hiện lệnh gán: n.i = 0xfd03; thì giá trị của n.i cũng bị thay đổi (n.i sẽ bằng 3).

Việc dùng kiểu union rất có lợi khi cần khai báo các cấu trúc dữ liệu mà nội dung của nó thay đổi tùy trạng thái. Ví dụ để mô tả các thông tin về một nhân viên, có thể khai báo một kiểu dữ liệu như sau:

```
1.  struct NhanVien
2.  {
3.      char    HoTen[30];
4.      int     NamSinh;
5.      char    NoiSinh[50];
6.      char    GioiTinh; //0: Nữ, 1: Nam
7.      char    DiaChi[50];
8.      char    Ttgd; //0: Chưa có gia đình, 1: Có gia đình
9.      union
10.     {
11.         char TenVo[30];
12.         char TenChong[30];
13.     }
14. };
```

1.3.4. Kiểu dữ liệu con trỏ

Các ngôn ngữ lập trình thường cung cấp cho chúng ta một kiểu dữ liệu đặc biệt để lưu trữ các địa chỉ của bộ nhớ, đó là con trỏ (Pointer). Tùy vào loại con trỏ gần (near pointer) hay con trỏ xa (far pointer) mà kiểu dữ liệu con trỏ có các kích thước khác nhau:

- Con trỏ gần: 2 bytes
- Con trỏ xa: 4 bytes

Cho trước kiểu T = <V, O>. Kiểu con trỏ, ký hiệu "T_P", chỉ đến các phần tử có kiểu "T" được định nghĩa:

T_P = <V_P, O_P>

Trong đó:

- **V_P** = {{ các địa chỉ có thể lưu trữ những đối tượng có kiểu T}, NULL} (với NULL là một giá trị đặc biệt tượng trưng cho một giá trị không biết hoặc không quan tâm).
- **O_P** = {các thao tác định địa chỉ của một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó}.

  (*Thường gồm các thao tác tạo một con trỏ chỉ đến một đối tượng thuộc kiểu T, hủy một đối tượng dữ liệu thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó*).
- Nói một cách dễ hiểu, kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng khác.
- Biến thuộc kiểu con trỏ T_P là biến mà giá trị của nó là địa chỉ của một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

Lưu ý: Kích thước của biến con trỏ tuỳ thuộc vào quy ước số byte địa chỉ trong từng mô hình bộ nhớ của từng ngôn ngữ lập trình cụ thể.

1.3.5. Kiểu dữ liệu tập tin

Tập tin (File) có thể xem là một kiểu dữ liệu đặc biệt, kích thước tối đa của tập tin tùy thuộc vào không gian đĩa nơi lưu trữ tập tin. Việc đọc và ghi dữ liệu trực tiếp trên tập tin rất mất thời gian và không bảo đảm an toàn cho dữ liệu trên tập tin đó. Do vậy, trong thực tế, chúng ta không thao tác trực tiếp dữ liệu trên tập tin mà chỉ cần chuyển từng phần hoặc toàn bộ nội dung của tập tin vào trong bộ nhớ trong để xử lý.

TỔNG KẾT CHƯƠNG

Trong chương này, chúng ta đã xem xét các khái niệm về cấu trúc dữ liệu, kiểu dữ liệu. Thông thường, các ngôn ngữ lập trình luôn định nghĩa sẵn một số kiểu dữ liệu cơ bản. Các kiểu dữ liệu này thường có cấu trúc đơn giản. Để thể hiện được các đối tượng đa dạng trong thế giới thực, chỉ dùng các kiểu dữ liệu này là không đủ nên cần xây dựng các kiểu dữ liệu mới phù hợp với đối tượng mà nó biểu diễn. Thành phần dữ liệu luôn là một vế quan trọng trong mọi chương trình. Vì vậy, việc thiết kế cấu trúc dữ liệu tốt là một vấn đề đáng quan tâm.

Vế thứ hai trong chương trình là các giải thuật (thuật toán). Một chương trình tốt phải có các cấu trúc dữ liệu phù hợp với các giải thuật hiệu quả. Khi khảo sát các giải thuật, người ta thường quan tâm đến chi phí thực hiện giải thuật. Chí phí này bao gồm chi phí về tài nguyên và thời gian cần để thực hiện giải thuật. Nếu như những đòi hỏi về tài nguyên có thể dễ dàng xác định thì việc xác định thời gian thực hiện nó không đơn giản. Có một số cách khác nhau để ước lượng khoảng thời gian này. Tuy nhiên, cách tiếp cận hợp lý nhất là hướng xấp xỉ tiệm cận. Hướng tiếp cận này không phụ thuộc ngôn ngữ, môi trường cài đặt cũng như trình độ của lập trình viên. Nó cho phép so sánh các giải thuật được khảo sát ở những nơi có vị trí địa lý rất xa nhau. Tuy nhiên, khi đánh giá ta cần chú ý thêm đến hệ số vô hướng trong kết quả đánh giá. Có khi hệ số này ảnh hưởng đáng kể đến chi phí thực của của giải thuật.

Do việc đánh giá chi phí thực hiện trung bình của giải thuật thường phức tạp nên người ta thường đánh giá chi phí thực hiện giải thuật trong trường hợp xấu nhất. Hơn nữa, trong một số giải thuật, việc xác định trường hợp xấu nhất là rất quan trọng.

1.4. Câu hỏi và bài tập

1.4.1. Câu hỏi

1. Trình bày tầm quan trọng của Cấu trúc dữ liệu và Giải thuật đối với người lập trình?
2. Các tiêu chuẩn để đánh giá cấu trúc dữ liệu và giải thuật?
3. Khi xây dựng giải thuật có cần thiết phải quan tâm tới cấu trúc dữ liệu hay không? Tại sao?
4. Tìm thêm một số ví dụ minh họa mối quan hệ giữa cấu trúc dữ liệu và giải thuật.
5. Cho biết một số kiểu dữ liệu được định nghĩa sẵn trong một ngôn ngữ lập trình mà bạn thường sử dụng. Cho biết một số kiểu dữ liệu xây dựng trước này có đủ để đáp ứng mọi yêu cầu về tổ chức dữ liệu không?
6. Một ngôn ngữ lập trình có nên cho phép người sử dụng tự định nghĩa thêm các kiểu dữ liệu có cấu trúc? Giải thích và cho ví dụ.

7. Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau những điểm nào? Một cấu trúc dữ liệu có thể có nhiều cấu trúc lưu trữ được không? Ngược lại một cấu trúc lưu trữ có thể tương ứng với nhiều cấu trúc dữ liệu được không? Cho ví dụ minh họa.
8. Các tiêu chuẩn để đánh giá một giải thuật? Và O( ) là gì?
9. Độ phức tạp có phải là tiêu chuẩn để đánh giá một giải thuật nhanh hay chậm? Cái gì là thước đo đánh giá độ phức tạp của bài toán.

1.4.2. Bài tập

10. Cho mảng một chiều các số nguyên, viết chương trình tìm một phần tử x trong mảng (nếu có trả về vị trí đầu tiên x xuất hiện trong mảng, nếu không trả về -1) bằng hai cách: Không sử dụng phần tử cầm canh và sử dụng phần tử cầm canh.
11. Tìm kiếm một giá trị x trên mảng đã sắp xếp.
12. Viết chương trình thống kê số phép gán và số phép so sánh trong bài 10 và 11.
13. Khai báo CTDL cho một sinh viên (SV) gồm các thông tin:
    - MaSV : *Số nguyên 4 bytes*
    - HoLot : *Chuỗi tối đa 20 kí tự*
    - Ten : *Chuỗi tối đa 8 kí tự*
    - NamSinh : *Số nguyên 2 bytes*
    - DTB : *Điểm trung bình số thực*

    Cài đặt các hàm sau:
    a. Nhập vào mảng một chiều a gồm n phần tử, mỗi phần tử là thông tin một SV (với $0 < n < 100$).
    b. Tìm sinh viên có MaSV = x trong mảng.
    c. Tìm sinh viên có MaSV = x (giả sử đã sắp tăng theo MaSV).
    d. Tìm SV có tên là x (nếu có nhiều SV thì chỉ cần tìm SV đầu tiên).
    e. In ra màn hình tất cả các SV có năm sinh = 1993.
14. Sử dụng các kiểu dữ liệu cơ bản trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính đa thức có bậc tự nhiên n ($0 \le n \le 100$) trên trường số thực (a_i, $x \in R$):

$$f_n(x) = \sum_{i=0}^{n} a_i x^i$$

Với cấu trúc dữ liệu được xây dựng, hãy trình bày giải thuật và cài đặt chương trình để thực hiện các công việc sau:

- Nhập, xuất các đa thức.
- Tính giá trị của đa thức tại giá trị x_0 nào đó.
- Tính tổng, tích của hai đa thức.

15. Tương tự như bài tập 14 nhưng đa thức trong trường số hữu tỷ Q (các hệ số a_i và x là các phân số có tử số và mẫu số là các số nguyên).
16. Cho bảng giờ tàu đi từ ga Sài Gòn đến các ga khác như sau (ga cuối là ga Hà Nội):

Bảng 1-4: Mảng hai chiều chứa bảng giờ tàu

TÀU ĐI	S2	S4	S6	S8	S10	S12	S14	S16	S18	LH2	SN2
HÀNH TRÌNH	**32 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**41 giờ**	**27giờ**	**10g30**
SÀI GÒN ĐI	*21g00*	*21g50*	*11g10*	*15g40*	*10g00*	*12g30*	*17g00*	*20g00*	*22g20*	*13g20*	*18g40*
MƯỜNG MÁN		2g10	15g21	19g53	14g07	16g41	21g04	1g15	3g16	17g35	22g58
THÁP CHÀM		5g01	18g06	22g47	16g43	19g19	0g08	4g05	6g03	20g19	2g15
NHA TRANG	*4g10*	*6g47*	*20g00*	*0g47*	*18g50*	*21g10*	*1g57*	*5g42*	*8g06*	*22g46*	*5g15*
TUY HÒA		9g43	23g09	3g39	21g53	0g19	5g11	8g36	10g50	2g10	
DIÊU TRÌ	*8g12*	*11g49*	*1g20*	*5g46*	*0g00*	*2g30*	*7g09*	*10g42*	*13g00*	*4g15*	
QUẢNG NGÃI		15g41	4g55	9g24	3g24	5g55	11g21	14g35	17g04	7g34	
TAM KỲ			6g11	10g39	4g38	7g10	12g40	16g08	18g21	9g03	
ĐÀ NẴNG	*13g27*	*19g04*	*8g29*	*12g20*	*6g19*	*9g26*	*14g41*	*17g43*	*20g17*	*10g53*	
HUẾ	*16g21*	*22g42*	*12g29*	*15g47*	*11g12*	*14g32*	*18g13*	*21g14*	*23g50*	*15g10*	
ĐÔNG HÀ		0g14	13g52	17g12	12g42	16g05	19g38	22g39	1g25		
ĐỒNG HỚI	19g15	2g27	15g52	19g46	14g41	17g59	21g38	0g52	3g28		
VINH	*23g21*	*7g45*	*21g00*	*1g08*	*20g12*	*23g50*	*2g59*	*7g07*	*9g20*		
THANH HÓA		10g44	0g01	4g33	23g09	3g33	6g39	9g59	12g20		
NINH BÌNH		12g04	1g28	5g54	0g31	4g50	7g57	11g12	13g51		
NAM ĐỊNH		12g37	2g01	6g26	1g24	5g22	8g29	11g44	14g25		
PHỦ LÝ		13g23	2g42	7g08	2g02	6g00	9g09	12g23	15g06		
ĐẾN HÀ NỘI	*5g00*	*14g40*	*4g00*	*8g30*	*3g15*	*7g10*	*10g25*	*13g45*	*16g20*		

Sử dụng các kiểu dữ liệu cơ bản, hãy xây dựng cấu trúc dữ liệu thích hợp để lưu trữ bảng giờ tàu trên vào bộ nhớ trong và bộ nhớ ngoài (disk) của máy tính.

Với cấu trúc dữ liệu đã được xây dựng ở trên, hãy trình bày giải thuật và cài đặt chương trình để thực hiện các công việc sau:

- Xuất ra giờ đến của một tàu T_0 nào đó tại một ga G_0 nào đó.
- Xuất ra giờ đến các ga của một tàu T_0 nào đó.
- Xuất ra giờ các tàu đến một ga G_0 nào đó.
- Xuất ra bảng giờ tàu theo mẫu ở trên.

Lưu ý:

- Các ô trống ghi nhận tại các ga đó, tàu này không đi đến hoặc chỉ đi qua mà không dừng lại.
- Dòng "HÀNH TRÌNH" ghi nhận tổng số giờ tàu chạy từ ga **Sài Gòn** đến ga **Hà Nội**.

17. Tương tự như bài tập 16, nhưng chương trình cần ghi nhận thêm thông tin về đoàn tàu khi dừng tại các ga chỉ để tránh tàu hay để cho khách lên/xuống (các dòng in nghiêng tương ứng với các ga có khách lên/xuống, các dòng khác chỉ dừng để tránh tàu).
18. Sử dụng kiểu dữ liệu cấu trúc trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính trạng thái của các cột đèn giao thông (*có ba đèn: Xanh, Đỏ, Vàng*). Với cấu trúc dữ liệu đã được xây dựng, hãy trình bày giải thuật và cài đặt chương trình để mô phỏng (minh họa) cho hoạt động của hai cột đèn trên hai tuyến đường giao nhau tại một ngã tư.
19. Sử dụng các kiểu dữ liệu cơ bản trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính trạng thái của một bàn cờ CARO có kích thước $M \times N$ ($0 \leq M, N \leq 20$). Với cấu trúc dữ liệu được xây dựng, hãy trình bày giải thuật và cài đặt chương trình để thực hiện các công việc sau:
    - In ra màn hình bàn cờ CARO trong trạng thái hiện hành.
    - Kiểm tra xem có ai thắng hay không? Nếu có thì thông báo "Kết thúc", nếu không có thì thông báo "Tiếp tục".
20. Giả sử quy tắc tổ chức quản lý nhân viên của một công ty như sau:
    - Thông tin về một nhân viên bao gồm lý lịch và bảng chấm công:
        + Lý lịch nhân viên:
            - Mã nhân viên: *Chuỗi 8 ký tự*.
            - Họ và tên nhân viên: *Chuỗi 30 ký tự (có thể tách ra làm 3 phần*).
            - Tình trạng gia đình: *1 ký tự* (M = Married, S = Single).

- Số con: *Số nguyên ≤20.*
- Trình độ văn hoá: *Chuỗi 2 ký tự.*

  (C1 = cấp 1; C2 = cấp 2; C3 = cấp 3; ĐH = đại học, CH = cao học)
- Lương căn bản: *Số nguyên ≤1.000.000*

+ Chấm công nhân viên:
  - Số ngày nghỉ có phép trong tháng: *Số nguyên ≤28*
  - Số ngày nghỉ không phép trong tháng: *Số nguyên ≤28*
  - Số ngày làm thêm trong tháng: *Số nguyên ≤28*
  - Kết quả công việc: *Chuỗi 2 ký tự* (TO = Tốt; BT = đạt ; KE = Kém)
  - Lương thực lĩnh trong tháng: *Số nguyên ≤2.000.000*

– Quy tắc lĩnh lương:

+ Lương thực lĩnh = Lương căn bản + Phụ trội

+ Trong đó nếu:
  - Số con > 2: Phụ trội = 5% Lương căn bản
  - Trình độ văn hoá = CH: Phụ trội = 10% lương căn bản
  - Làm thêm: Phụ trội = 4% lương căn bản/ngày
  - Nghĩ không phép: Phụ trội = -5% lương căn bản/ngày

– Chức năng yêu cầu:

+ Cập nhật lý lịch, bảng chấm công cho nhân viên (*thêm, xóa, sửa*).

+ Xem bảng lương hàng tháng.

+ Tìm thông tin của một nhân viên.

Tổ chức cấu trúc dữ liệu thích hợp để biểu diễn các thông tin trên, và cài đặt chương trình theo các chức năng đã mô tả.

Lưu ý:

– Nên phân biệt thông tin mang tính chất tĩnh (lý lịch) và động (chấm công hàng tháng).

– Số lượng nhân viên tối đa là 50 người.

Chương 2. TÌM KIẾM VÀ SẮP XẾP

2.1. Giới thiệu

Trong chương này sẽ xem xét các giải thuật tìm kiếm và sắp xếp thông dụng. Cấu trúc dữ liệu chính để minh họa các thao tác này chủ yếu là mảng một chiều. Đây cũng là một trong những cấu trúc dữ liệu thông dụng nhất.

Khi khảo sát giải thuật tìm kiếm, chúng ta sẽ làm quen với hai giải thuật. Giải thuật thứ nhất là giải thuật tìm kiếm tuần tự, giải thuật này có độ phức tạp tuyến tính O(n). Giải thuật thứ hai là giải thuật tìm kiếm nhị phân, giải thuật này có độ phức tạp $O(\log_2(n))$. Tuy giải thuật tìm kiếm nhị phân có ưu điểm tìm kiếm nhanh hơn giải thuật tìm kiếm tuyến tính nhưng giải thuật tìm kiếm nhị phân phải sắp xếp trước khi tìm kiếm. Vì thế cần phải dựa vào điều kiện thực tế để quyết định chọn giải thuật nào.

Phần tiếp theo của chương sẽ trình bày các giải thuật sắp xếp thông dụng từ đơn giản đến phức tạp (tức là từ chi phí cao đến chi phí thấp).

Các giải thuật đơn giản đó là các giải thuật *đổi chỗ trực tiếp, chèn trực tiếp, chọn trực tiếp, nổi bọt*; các giải thuật này đều có chi phí về thời gian thực hiện khoảng $O(n^2)$. Các giải thuật *Shell Sort, Heap Sort* đều là những giải thuật cải tiến từ những giải thuật đơn giản trên, chúng có chi phí về thời gian thực hiện nhỏ hơn $O(n^2)$. Còn lại giải thuật *Merge Sort* và *Quick Sort* là những giải thuật thực hiện theo chiến lược chia để trị; tuy việc cài đặt chúng phức tạp nhưng chi phí thực hiện của chúng lại thấp khoảng $O(n\log_2(n))$.

2.2. Nhu cầu tìm kiếm và sắp xếp dữ liệu trong một hệ thống thông tin

Trong hầu hết các hệ lưu trữ, quản lý dữ liệu, thao tác tìm kiếm thường được thực hiện nhiều để khai thác thông tin.

Ví dụ: Tra cứu từ điển, tìm sách trong thư viện, …

Do các hệ thống thông tin thường phải lưu trữ một khối lượng dữ liệu đáng kể, nên việc xây dựng các giải thuật cho phép tìm kiếm nhanh sẽ có ý nghĩa rất lớn. Cho nên nếu dữ liệu trong hệ thống được tổ chức theo một trật tự nào đó, thì việc tìm kiếm sẽ tiến hành nhanh chóng và hiệu quả hơn.

Ví dụ: Sách trong thư viện được sắp xếp theo từng chủ đề.

Hiện nay có nhiều giải thuật tìm kiếm và sắp xếp được xây dựng, mức độ hiệu quả của từng giải thuật còn phụ thuộc vào tính chất của cấu trúc dữ liệu cụ thể mà nó tác động đến.

2.3. Các giải thuật tìm kiếm

2.3.1. Khái quát về tìm kiếm

Trong thực tế, khi thao tác, khai thác dữ liệu chúng ta hầu như lúc nào cũng phải thực hiện thao tác tìm kiếm. Việc tìm kiếm nhanh hay chậm tùy thuộc vào trạng thái và trật tự của dữ liệu trên đó. Kết quả của việc tìm kiếm có thể là không có (*không tìm thấy*) hoặc có (*tìm thấy*). Nếu kết quả tìm kiếm là có tìm thấy thì nhiều khi chúng ta còn phải xác định xem vị trí của phần tử dữ liệu tìm thấy là ở đâu? Trong phạm vi của phần này chúng ta tìm cách giải quyết các câu hỏi này.

Trước khi đi vào nghiên cứu chi tiết, chúng ta giả sử rằng mỗi phần tử dữ liệu được xem xét có một thành phần khóa (*Key*) để nhận diện, có kiểu dữ liệu là T nào đó; các thành phần còn lại là thông tin (*Info*) liên quan đến phần tử dữ liệu đó. Như vậy mỗi phần tử dữ liệu có cấu trúc dữ liệu như sau:

```
1. typedef DataType1 KeyType; //Định nghĩa kiểu dữ liệu của Khóa
2. typedef DataType2 InfoType; //Định nghĩa kiểu dữ liệu của thông tin
3. struct NameItem
4. { //Tên kiểu dữ liệu mới do người dùng đặt tùy ý gợi nhớ
5.       KeyType Key;
6.       InfoType Info;
7. };
```

Trong giáo trình này, để đơn giản thì khi nói tới giá trị của một phần tử dữ liệu cũng đồng nghĩa với việc nói tới giá trị khóa (Key) của phần tử dữ liệu đó.

Việc tìm kiếm một phần tử có thể diễn ra trên một dãy/mảng (*tìm kiếm nội*) hoặc diễn ra trên một tập tin/file (*tìm kiếm ngoại*). Phần tử cần tìm là phần tử cần thỏa mãn điều kiện tìm kiếm (*thường có giá trị khóa Key bằng giá trị tìm kiếm x cho trước*). Tùy thuộc vào từng bài toán cụ thể mà điều kiện tìm kiếm có thể khác nhau.

Công việc tìm kiếm sẽ hoàn thành nếu có một trong hai tình huống sau xảy ra:

- Tìm được phần tử có khóa Key bằng với giá trị cần tìm x, lúc đó phép tìm kiếm thành công → Cho biết vị trí tìm thấy.
- Không tìm được phần tử có khóa Key nào bằng với giá trị cần tìm x, lúc đó phép tìm kiếm thất bại.

Tuy nhiên, do hạn chế về thời gian và để đơn giản hóa việc trình bày các giải thuật; nên trong phần này chúng tôi xin được phép dùng mảng một chiều để lưu danh sách có n phần tử $\mathbf{a_0, a_1, a_2, \ldots, a_{n-1}}$ trong bộ nhớ chính.

Lưu ý: Khi trình bày các thuật giải trong giáo trình này, tác giả cố gắng minh họa theo ngôn ngữ lập trình C.

2.3.2. Đặt vấn đề

Bài toán tìm kiếm được phát biểu như sau: Giả sử có một mảng **a** gồm **n** phần tử: a_0, a_1, a_2…, a_{n-1}. Vấn đề đặt ra là có hay không phần tử có giá trị bằng **x** trong mảng a? Nếu có thì phần tử có giá trị bằng x là phần tử thứ mấy trong mảng a?

2.3.3. Giải thuật tìm kiếm tuyến tính (Linear Search)

2.3.3.1. Giải thuật

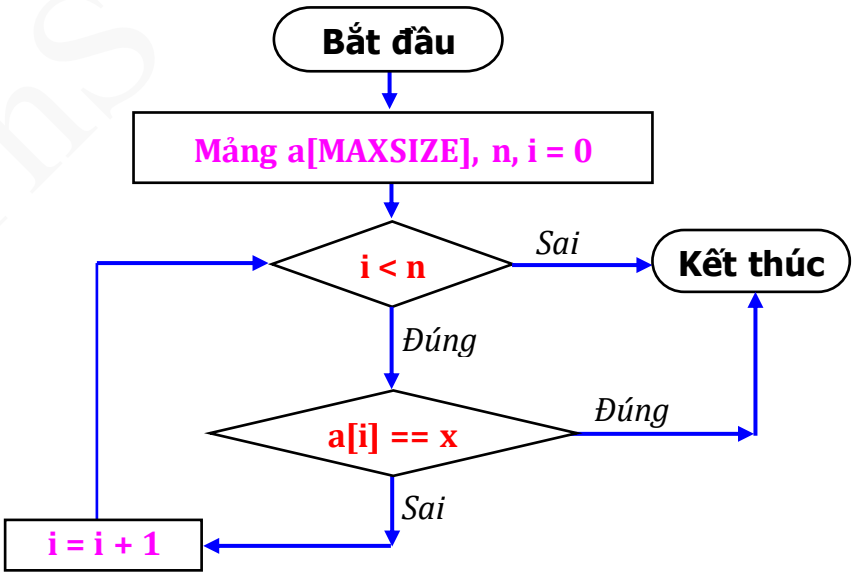
Gọi **x** là giá trị cần tìm và **a** là mảng chứa dữ liệu có **n** phần tử.

- ***Ý tưởng giải thuật:*** Tiến hành so sánh x với từng phần tử của mảng a, lần lượt so sánh với phần tử thứ 1, phần tử thứ 2, …. Quá trình so sánh được thực hiện tuần tự cho đến khi gặp phần tử có khóa cần tìm x, hoặc đã tìm hết mảng mà không thấy x.

- ***Các bước tiến hành giải thuật:***

- **Bước 1:** Khởi gán giá trị biến i: i = 0;
- **Bước 2:** So sánh a[i] với giá trị x cần tìm, có hai khả năng:
  + Nếu (a[i] == x) thì: Tìm thấy x tại vị trí thứ i. Dừng.
  + Ngược lại (a[i] != x) thì sang Bước 3.
- **Bước 3:** Tăng biến i lên 1: i = i + 1; *//xét phần tử kế trong mảng*
  + Nếu (i == n) thì: Hết mảng, không tìm thấy. Dừng
  + Ngược lại: Quay lại Bước 2.

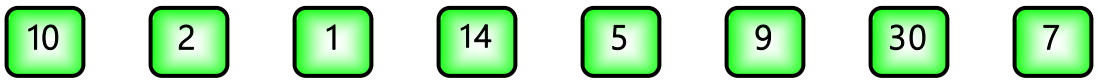
2.3.3.2. Lưu đồ



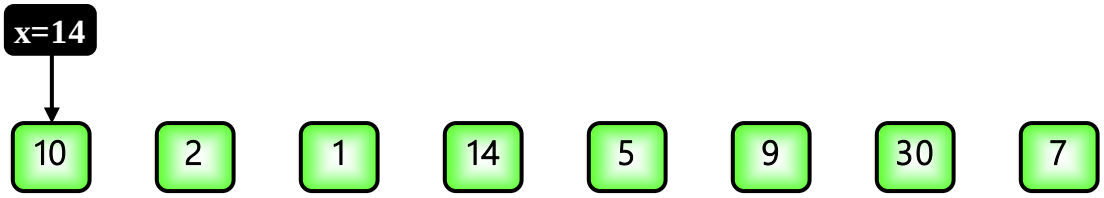
Hình 2.1: Lưu đồ giải thuật tìm kiếm tuyến tính

2.3.3.3. Ví dụ minh họa giải thuật

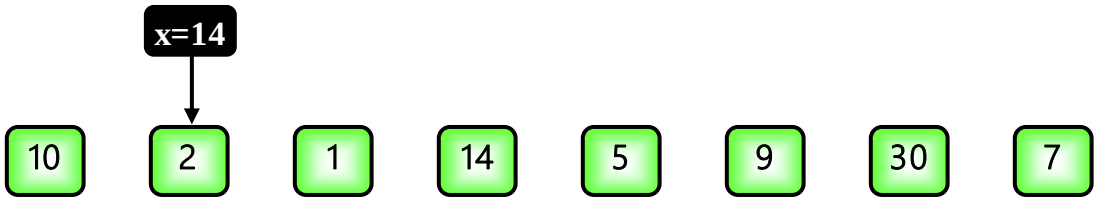
Cho dãy số a gồm **8** phần tử như sau:



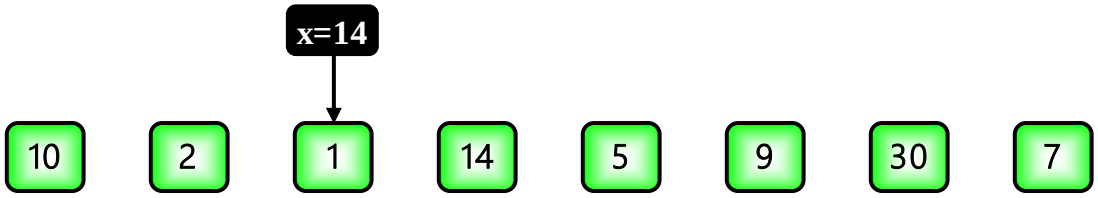
Lần 1: i=0.



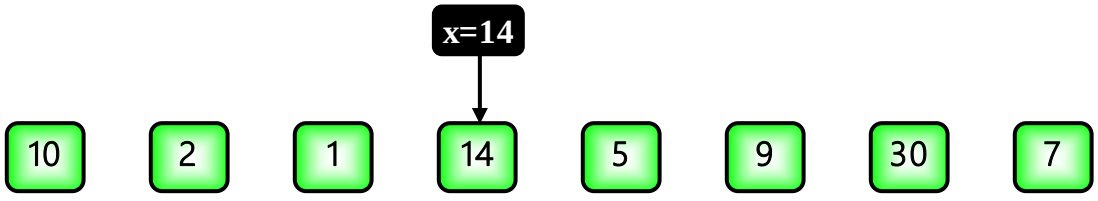
Lần 2: i=1.



Lần 3: i=2.



Lần 4: i=3.



Lần 5: Dừng.

Hình 2.2: Ví dụ minh họa giải thuật tìm kiếm tuyến tính

2.3.3.4. Cài đặt giải thuật

Hàm LinearSearch được cài đặt bên dưới sẽ nhận vào một mảng chứa các số nguyên a gồm n phần tử và một giá trị cần tìm x. Sau khi thực hiện xong hàm sẽ trả về vị trí i đầu tiên nếu tìm thấy giá trị x trong mảng a, và trả về giá trị -1 nếu không tìm thấy x trong mảng.

```
1.  int linearSearch_While(ItemType a[ ], int n, ItemType x)
2.  {
3.      int i = 0;
4.      while( (i < n) && (a[i] != x) )
5.          i++;
6.      if(i == n)
7.          return -1; //Không tìm thấy x
8.      else
9.          return i; //Tìm thấy x tại vị trí i
10. }
```

Hoặc viết bằng vòng lặp for:

```
1.  int linearSearch_For(ItemType a[ ], int n, ItemType x)
2.  {
3.      for(int i = 0; i < n; i++)
4.          if(a[i] == x)
5.              return i;
6.      return –1;
7.  }
```

Trong phần cài đặt trên, chúng ta thấy rằng công việc tìm kiếm chỉ đơn giản thực hiện bằng một vòng lặp để duyệt qua tất cả các phần tử trong mảng. Vòng lặp chỉ kết thúc khi tìm thấy giá trị x trong mảng hoặc đã duyệt hết các phần tử có trong mảng. Như vậy khi kết thúc vòng lặp chỉ số i có hai khả năng xảy ra:

- Nếu i < n: phần tử thứ i có giá trị bằng với giá trị x cần tìm. Do đó, hàm trả về giá trị i.
- Nếu i = n: đã duyệt qua khỏi phần tử cuối cùng của mảng nhưng vẫn không tìm thấy phần tử nào có giá trị bằng với x. Do đó, hàm trả về giá trị -1.

2.3.3.5. Đánh giá giải thuật

Dễ dàng ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Bảng sau sẽ cho biết độ phức tạp trong từng trường hợp cụ thể:

Bảng 2-1: Đánh giá giải thuật tìm kiếm tuyến tính

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1 + 1	Phần tử đầu tiên có giá trị là x
Xấu nhất	2n + 1	Phần tử cuối cùng có giá trị là x
Trung bình	$\frac{2n+3}{2}$	Giả sử xác xuất các phần tử trong mảng nhận giá trị x là như nhau

Vậy giải thuật có độ phức tạp **O(n)**.

2.3.3.6. Nhận xét.

- Giải thuật tìm kiếm tuyến tính không phụ thuộc vào thứ tự tăng dần hay giảm dần của các phần tử trong mảng.
- Số phép so sánh của giải thuật trong trường hợp xấu nhất là 2n + 1.
- Để giảm thiểu số phép so sánh trong vòng lặp cho giải thuật chỉ còn n + 1, có thể thêm phần tử "lính canh" vào cuối dãy như sau:

```
1.  int improveLinearSearch(ItemType a[ ], int n, ItemType x)
2.  {
3.      int i = 0;
4.      a[n] = x; //a[n] là phần tử "lính canh"
5.      while(a[i] != x)
6.          i++;
7.      if(i < n)
8.          return i; //Tìm thấy x tại vị trí i
9.      else
10.         return -1; //Không tìm thấy x
11. }
```

2.3.4. Giải thuật tìm kiếm nhị phân (Binary Search)

2.3.4.1. Giải thuật

Giả sử ta có dãy chứa các giá trị đã được sắp xếp tăng (có thứ tự tăng theo một tiêu chuẩn nào đó), các phần tử trong dãy có quan hệ $\mathbf{a_{i-1} \le a_i \le a_{i+1}}$, từ đó kết luận được nếu $\mathbf{x > a_i}$ thì x chỉ có thể xuất hiện trong đoạn $[\mathbf{a_{i+1}, a_n}]$ của dãy, ngược lại nếu $\mathbf{x < a_i}$ thì x chỉ có thể xuất hiện trong đoạn $[\mathbf{a_0, a_{i-1}}]$ của dãy. Giải thuật tìm nhị phân áp dụng nhận xét trên đây để tìm các giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong dãy.

Ý tưởng giải thuật: xét một dãy khóa đã sắp xếp tăng dần từ a[left] ≤ ... ≤ a[mid] ≤ … ≤ a[right]. Tại mỗi bước tiến hành so sánh khóa cần tìm x với phần tử nằm ở giữa a[mid] của dãy hiện hành, dựa vào kết quả so sánh này để quyết định giới

hạn dãy tìm kiếm ở bước kế tiếp là nửa bên trái từ a[left] ≤ ... ≤ a[mid-1] hay nửa bên phải từ a[mid + 1] ≤ ... ≤ a[right]. Quá trình tìm kiếm sẽ kết thúc khi left > right: không tìm thấy phần tử nào có khóa x, hoặc tìm được một phần tử a[mid] = x.

Các bước tiến hành giải thuật:

Gọi x là khóa cần tìm, a là mảng chứa các giá trị dữ liệu gồm n phần tử đã sắp xếp, Left và Right là chỉ số đầu và cuối của đoạn cần tìm, Mid là chỉ số của phần tử nằm giữa của đoạn cần tìm. Công việc tìm kiếm được tiến hành như sau:

- **Bước 1:** Khởi gán giá trị ban đầu cho các biến (*tìm kiếm trên toàn bộ dãy*)

  Left = 0; Right = n - 1;

- **Bước 2:** Tính giá trị biến Mid: Mid = (Left + Right)/2;

  So sánh a[Mid] với giá trị cần tìm x, có ba khả năng có thể xảy ra:

  + Nếu (a[Mid] == x): Tìm thấy. Dừng

  + Nếu (a[Mid] > x): Chuẩn bị tìm x trong dãy con $a_{Left}..a_{Mid-1}$: Right = Mid – 1;

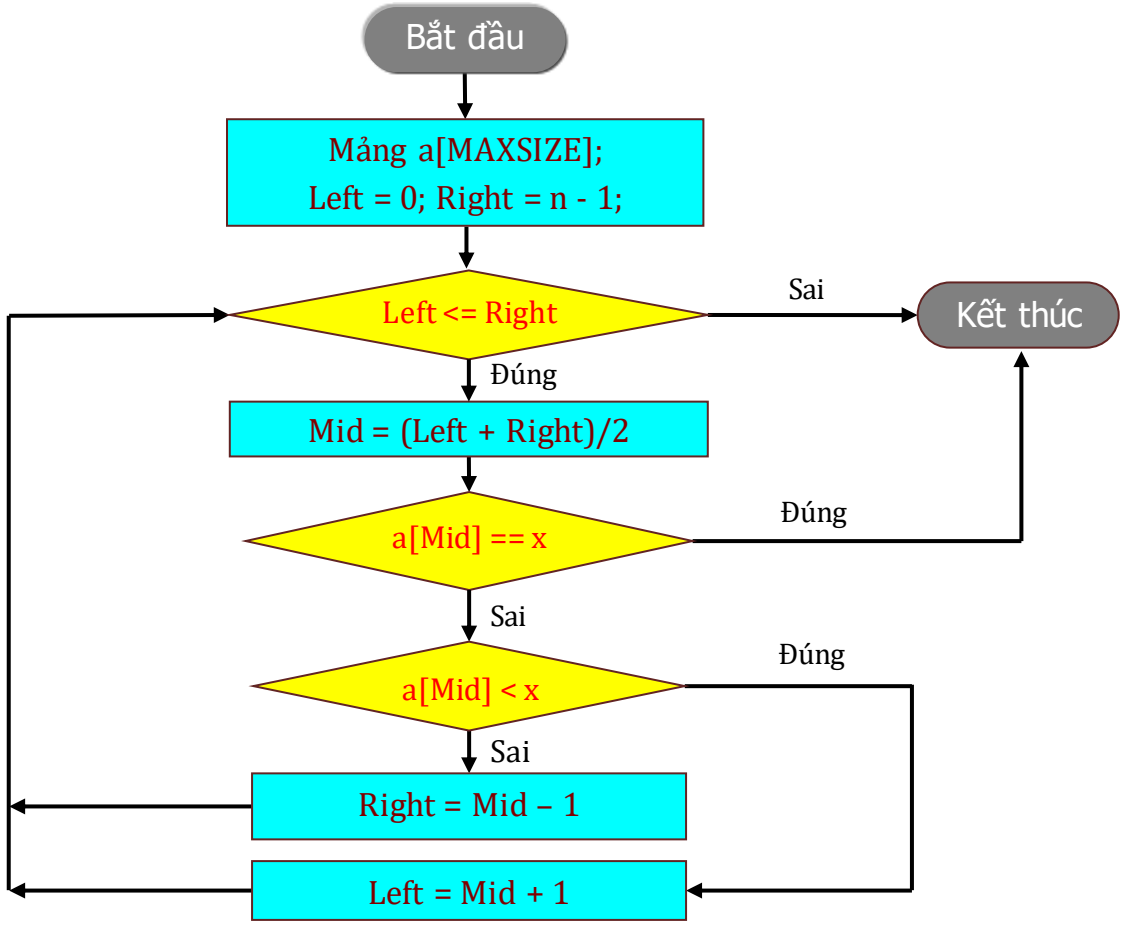
  + Nếu (a[Mid] < x): Chuẩn bị tìm x trong dãy con $a_{Mid+1}..a_{Right}$: Left = Mid + 1;

- **Bước 3:**

  + Nếu (Left ≤ Right): Dãy hiện hành vẫn còn phần tử. Quay lại Bước 2.

  + Ngược lại: Dãy hiện hành hết phần tử. Dừng.

2.3.4.2. Lưu đồ

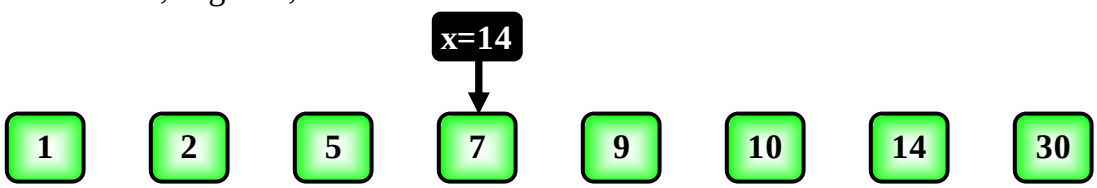


Hình 2.3: Lưu đồ giải thuật tìm kiếm nhị phân

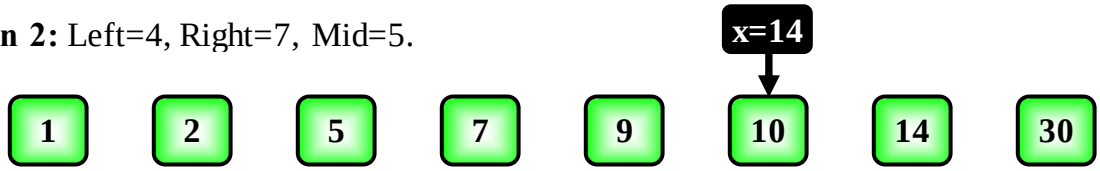
2.3.4.3. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử có thứ tự như sau:

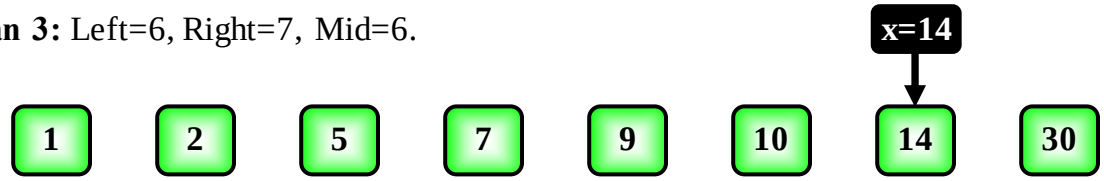
Lần 1: Left=0, Right=7, Mid=3.



Lần 2: Left=4, Right=7, Mid=5.



Lần 3: Left=6, Right=7, Mid=6.



Lần 4: Dừng.

Hình 2.4: Ví dụ minh họa giải thuật tìm kiếm nhị phân

2.3.4.4. Cài đặt giải thuật

2.3.4.4.1. Cài đặt hàm không đệ quy

```
1.  int binarySearch(ItemType a[ ], int n, ItemType x)
2.  {
3.      int Left = 0;
4.      int Right = n - 1;
5.      while(Left <= Right)
6.      {
7.          int Mid = (Left + Right) / 2;
8.          if(x == a[Mid])
9.              return Mid;
10.         else if(x < a[Mid])
11.             Right = Mid – 1;
12.         else
13.             Left = Mid + 1;
14.     }
15.     return –1;
16. }
```

Hàm binarySearch cũng chỉ đơn giản sử dụng một vòng lặp để duyệt qua các phần tử trong mảng. Tuy nhiên, vòng lặp này sẽ không duyệt qua hết tất cả các phần tử như đối với tìm kiếm tuyến tính. Tư tưởng cài đặt của hàm này cũng giống như đối với linearSearch, nghĩa là khi kết thúc vòng lặp cũng có hai khả năng xảy ra:

- Chỉ số Left vẫn còn bé hơn hoặc bằng chỉ số Right. Điều này có nghĩa là đã có một phần tử nào đó bằng với giá trị x cần tìm, cụ thể là giá trị của phân tử Mid. Do đó, hàm sẽ trả về giá trị nằm trong biến Mid.
- Chỉ số Left đã vượt qua chỉ số Right, nghĩa là đã duyệt qua hết các phần tử trong mảng mà không tìm thấy phần tử nào có giá trị bằng x. Do đó, hàm trả về giá trị -1.

2.3.4.4.2. Cài đặt hàm đệ quy

```
1.  int binarySearch(ItemType a[ ], int Left, int Right, ItemType x)
2.  {
3.      if(Left > Right)
4.          return (-1);
5.      int Mid = (Left + Right)/2;
6.      if(x == a[Mid])
7.          return Mid;
8.      else if(x < a[Mid])
9.          return binarySearch(a, Left, Mid - 1, x);
10.     else
11.         return binarySearch(a, Mid + 1, Right, x);
12. }
```

2.3.4.5. Đánh giá giải thuật

Người ta đã chứng minh được độ phức tạp tính toán của giải thuật tìm nhị phân như bảng sau:

Bảng 2-2: Đánh giá giải thuật tìm kiếm nhị phân

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng ban đầu có giá trị x.
Xấu nhất	$\log_2(n)$	Phần tử cần tìm nằm ở cuối mảng.
Trung bình	$\log_2(\frac{n}{2})$	Giả sử xác xuất các phần tử trong mảng nhận giá trị x là như nhau.

Vậy giải thuật có độ phức tạp $\mathbf{O(\log_2(n))}$.

2.3.4.6. Nhận xét

Giải thuật tìm nhị phân phụ thuộc vào thứ tự của các phần tử trong mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.

Giải thuật *tìm kiếm nhị phân* tiết kiệm thời gian hơn rất nhiều so với giải thuật *tìm kiếm tuyến tính* do $\mathbf{O_{Nhị\ phân}}(\log_2 n) < \mathbf{O_{Tuyến\ tính}}(n)$. Tuy nhiên khi muốn áp dụng giải thuật tìm kiếm nhị phân cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự, thời gian này là không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại, … tất cả các nhu cầu đó tạo ra khuyết điểm chính cho giải thuật tìm kiếm nhị phân.

2.4. Các giải thuật sắp xếp

2.4.1. Định nghĩa sắp xếp

Sắp xếp danh sách (dãy) có n phần tử $a_0, a_1, ..., a_{n-1}$ là quá trình xử lý các phần tử trong danh sách để đặt chúng theo một thứ tự thỏa mãn một số tiêu chuẩn nào đó dựa trên thông tin lưu tại mỗi phần tử. Để quyết định những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Như vậy, hai thao tác cơ bản của giải thuật sắp xếp là so sánh và đổi chỗ, khi xây dựng một giải thuật sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của giải thuật.

Đối với các danh sách được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được xem xét ưu tiên hàng đầu, do vậy những giải thuật sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả, ngoài vùng nhớ lưu trữ danh sách ban đầu, thường ít được quan tâm. Thay vào đó, các giải thuật sắp xếp trực tiếp trên danh sách ban đầu - gọi là các giải thuật sắp xếp tại chỗ - lại được đầu tư phát triển. Phần này giới thiệu một số giải thuật sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp nội.

Để sắp xếp một danh sách chưa có thứ tự thành danh sách có thứ tự (giả sử có thứ tự tăng), ta phải tiến hành triệt tiêu tất cả các cặp nghịch thế trong danh sách, hay nói cách khác là tìm cách sắp xếp lại danh sách để sao cho tất cả các cặp đều là thuận thế.

Khái niệm nghịch thế:

Xét một danh sách (mảng một chiều) có n phần tử $a_0, a_1 \ldots, a_{n-1}$.

Nếu có $\mathbf{i < j}$ và $\mathbf{a_i > a_j}$ thì ta gọi đó là 1 cặp nghịch thế.

Ví dụ: Cho mảng a như sau: 14 5 7 8 3

→Vậy dãy trên ta có các cặp nghịch thế sau: (14, 5); (7, 3); (8, 3); ….

2.4.2. Giải thuật sắp xếp đổi chỗ trực tiếp (Interchange Sort)

2.4.2.1. Giải thuật

- ***Ý tưởng giải thuật***: Xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, lần lượt làm triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế, cuối cùng đưa được phần tử nhỏ nhất (*lớn nhất*) về vị trí đúng là đầu dãy hiện hành, sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn (n - 1) phần tử của dãy ban đầu, và bắt đầu từ phần tử ở vị trí thứ 2 trong lần sắp xếp tiếp theo. Lặp lại quá trình xử lý như trên với các phần tử còn lại trong dãy, cho đến khi dãy hiện hành không còn phần tử nào thì dừng.

- ***Các bước tiến hành giải thuật:***

Giả sử a là mảng một chiều có n phần tử chưa được sắp xếp thứ tự. Ta áp dụng ý tưởng giải thuật để tiến hành sắp xếp mảng a tăng dần như sau:

- **Bước 1:** Khởi gán giá trị biến i: i = 0; *//bắt đầu từ đầu dãy*
- **Bước 2:** Khởi gán giá trị biến j: j = i + 1; *//bắt đầu từ phần tử kế sau phần tử thứ i.*
- **Bước 3:**

  Lặp lại trong khi (**j < n**) thì thực hiện:

  + **Bước 3.1:** Nếu (**a[j] < a[i]**) thì *//xét cặp a[j] và a[i]*

    Hoán vị a[j] với a[i] *// do a[j] tạo nghịch thế với a[i]*

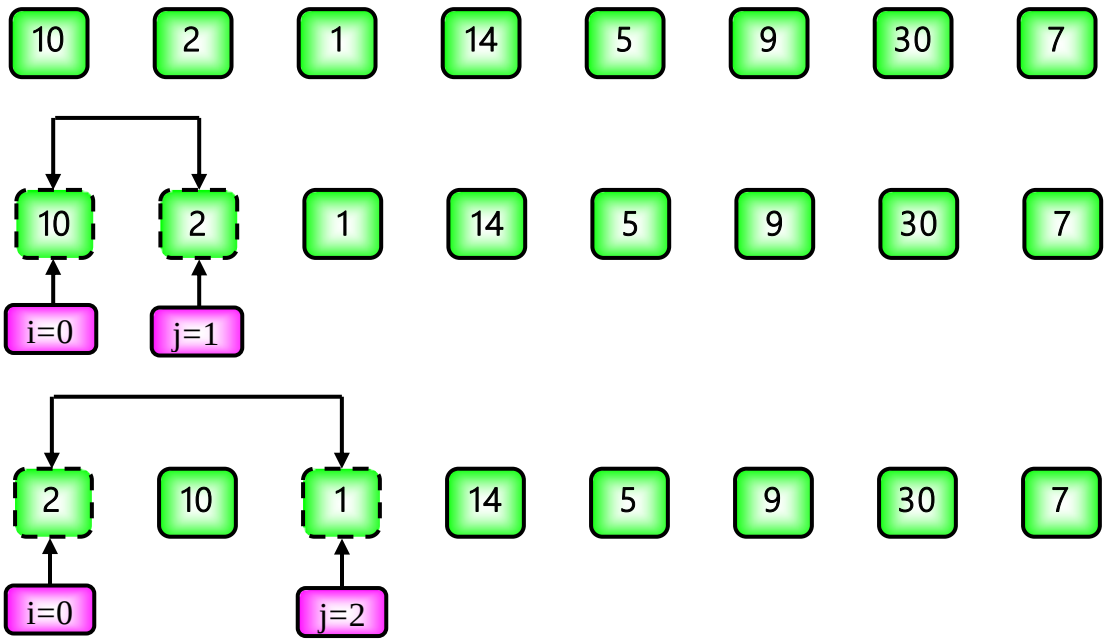
  + **Bước 3.2:** Tăng biến j: j = j + 1; *//tăng j lên 1 để tìm nghịch thế mới*
- **Bước 4:** Tăng biến i: i = i + 1; *//tăng i lên 1 để xét phần tử tiếp theo*

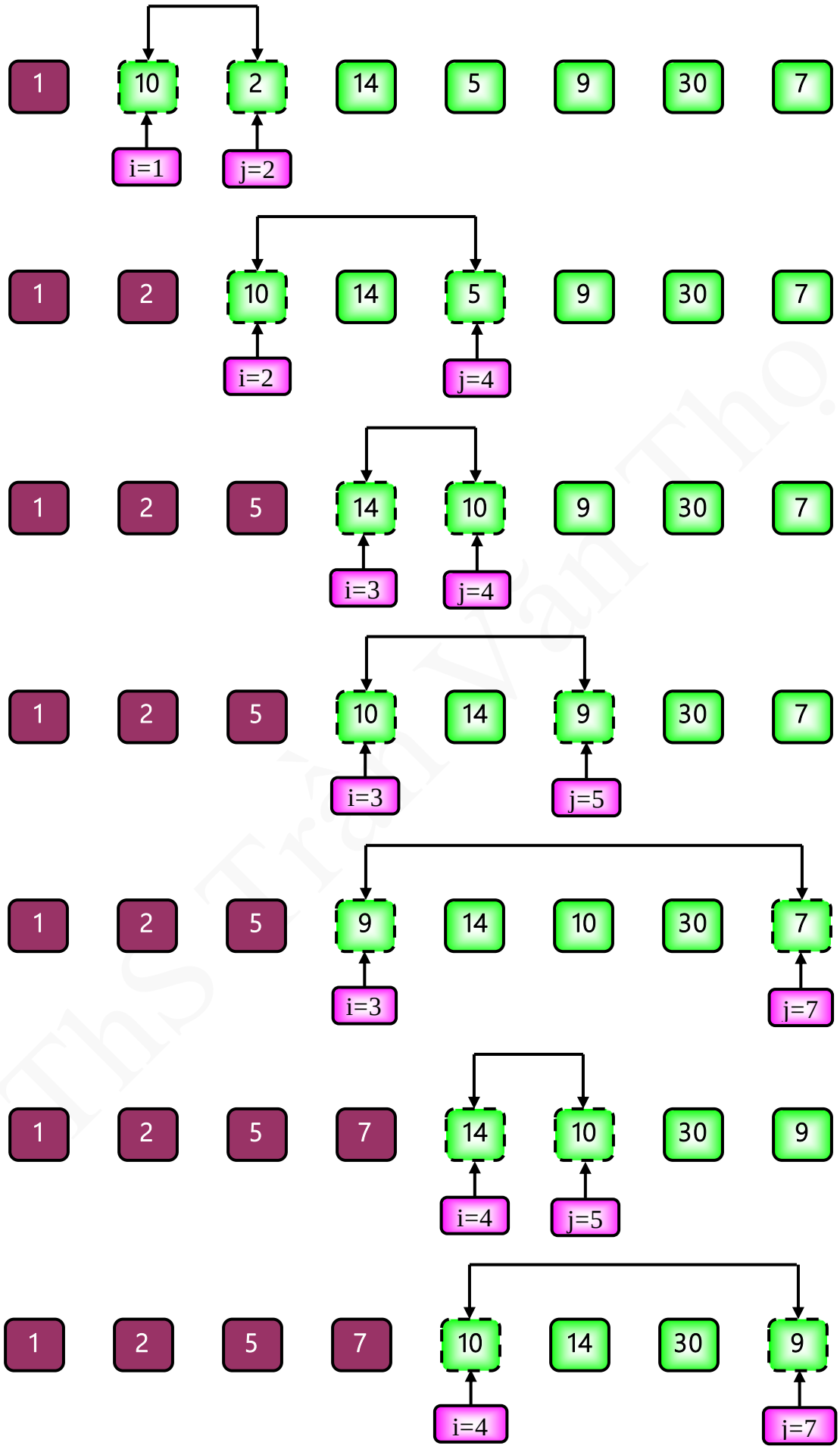
  Nếu (**i < n - 1**) thì: Quay lại Bước 2.

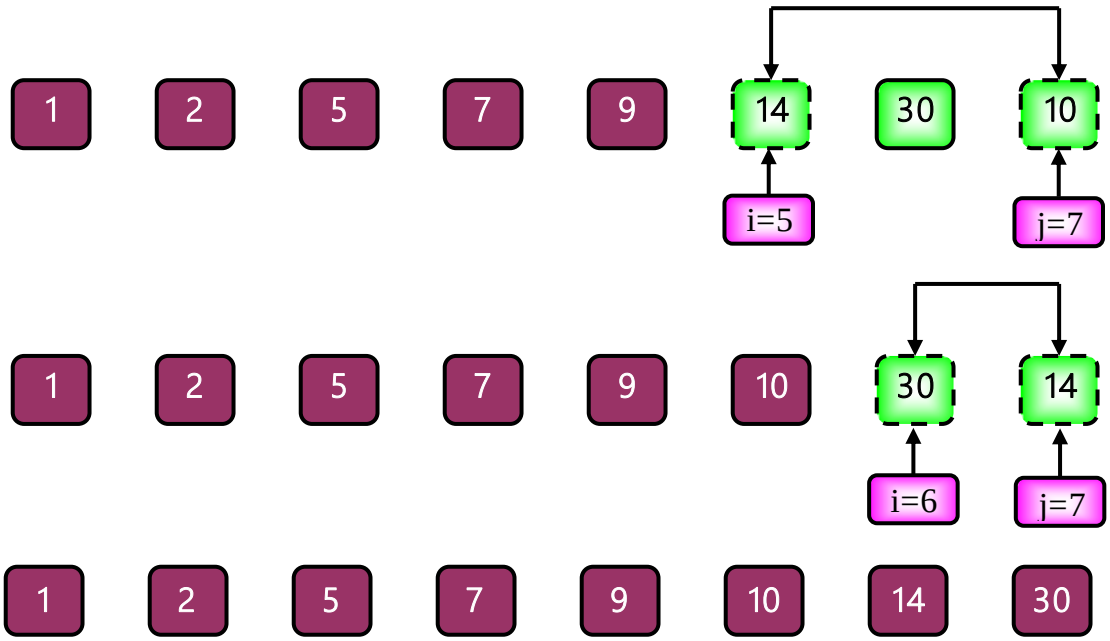
  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

2.4.2.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:







Dừng.

Hình 2.5: Ví dụ minh họa giải thuật sắp xếp đổi chỗ trực tiếp

Vậy dãy đã được sắp xếp tăng dần.

2.4.2.3. Cài đặt giải thuật

Hàm interchangeSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

```
1. void interchangeSort_Ascending(ItemType a[ ], int n)
2. {
3.     for(int i = 0; i < n - 1; i++)
4.         for(int j = i + 1; j < n; j++)
5.             if(a[i] > a[j])
6.                 swap(a[i], a[j]);
7. }
```

Trong đó hàm **swap** là hàm hoán vị hai phần tử (*có kiểu dữ liệu là ItemType*) cho nhau, như sau:

```
1. void swap(ItemType &x, ItemType &y)
2. {
3.     ItemType tam;
4.     tam = x;
5.     x = y;
6.     y = tam;
7. }
```

2.4.2.4. Đánh giá giải thuật

Chúng ta có thể lập bảng đánh giá giải thuật như sau:

Bảng 2-3: Đánh giá giải thuật sắp xếp đổi chỗ trực tiếp

Trường hợp	Số lần so sánh	Số lần gán
Tốt nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	$3\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=3\frac{n(n-1)}{2}$

Vậy độ phức tạp của giải thuật là **O(n²)**.

2.4.3. Giải thuật sắp xếp nổi bọt (Bubble Sort)

2.4.3.1. Giải thuật

- ***Ý tưởng giải thuật***: Xuất phát từ cuối dãy (*đầu dãy*) và tiến hành đổi chỗ các cặp phần tử kế cận nhau để đưa phần tử nhỏ hơn (*lớn hơn*) về vị trí đầu (*cuối*) của dãy hiện hành. Sau khi đã được chuyển về đúng vị trí thì phần tử này sẽ không cần xét đến ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có phần tử đầu dãy ở vị trí là i. Lặp lại quá trình xử lý như trên với các phần tử còn lại trong dãy, cho đến khi dãy hiện hành không còn phần tử nào thì dừng.

- ***Các bước tiến hành giải thuật:***

- **Bước 1:** Khởi gán giá trị biến i: i = 0; *//lần xử lý đầu tiên*
- **Bước 2:** Khởi gán giá trị biến j: j = n - 1; *//Duyệt từ cuối dãy đến phần tử thứ i*

  Lặp lại trong khi (**j > i**) thì thực hiện:

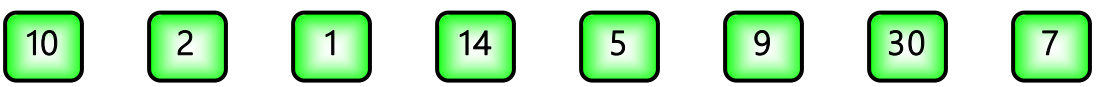
  + **Bước 2.1:** Nếu (a[j] < a[j - 1]) thì hoán đổi 2 phần tử a[j] và a[j - 1] với nhau.

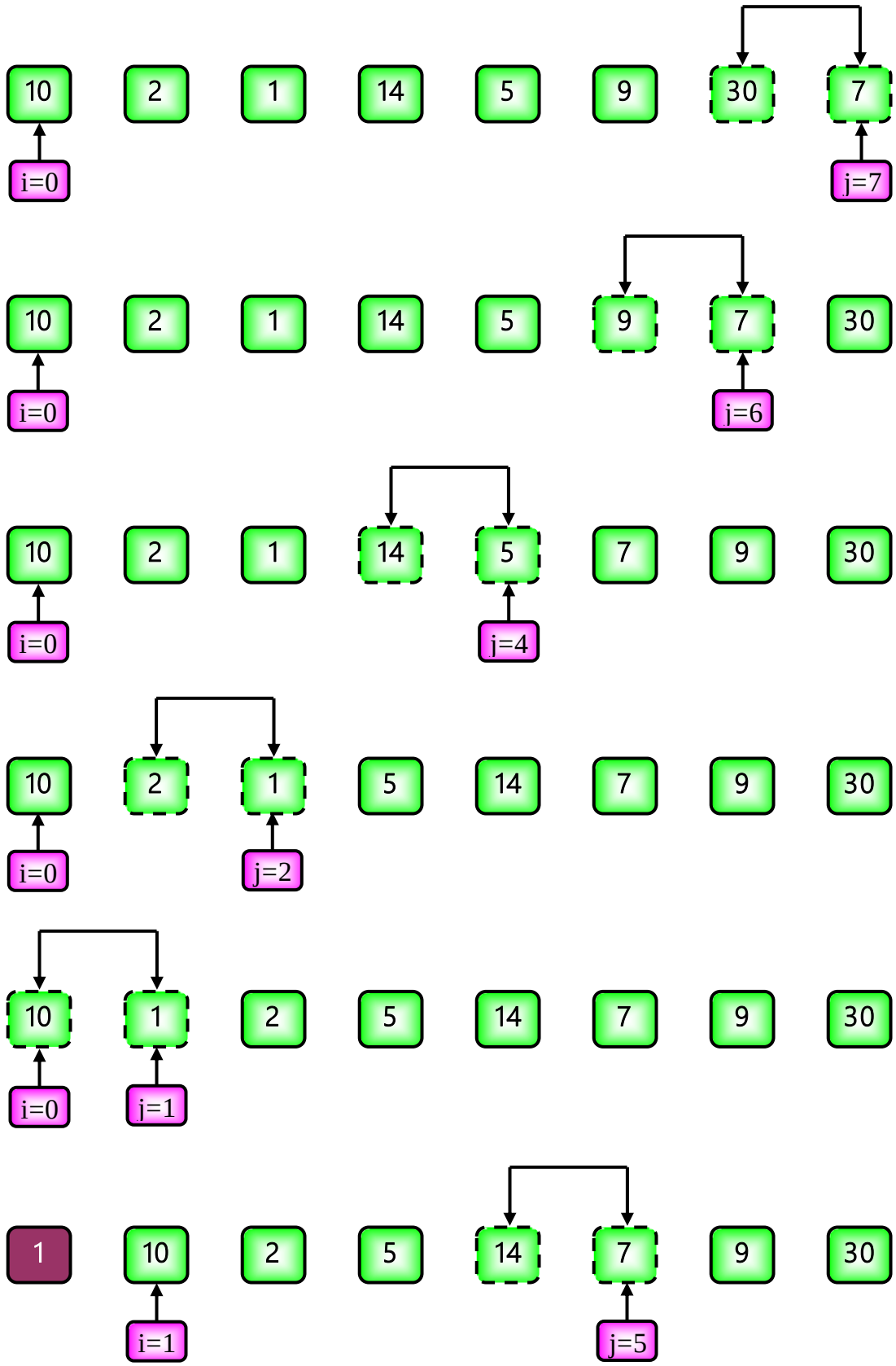
  + **Bước 2.2:** Giảm giá trị biến j: j = j - 1;
- **Bước 3:** Tăng giá trị biến i: i = i + 1; *//lần xử lý kế tiếp, dãy chỉ còn n-i phần tử*
- **Bước 4:** Nếu (**i < n**) thì: Quay lại Bước 2.

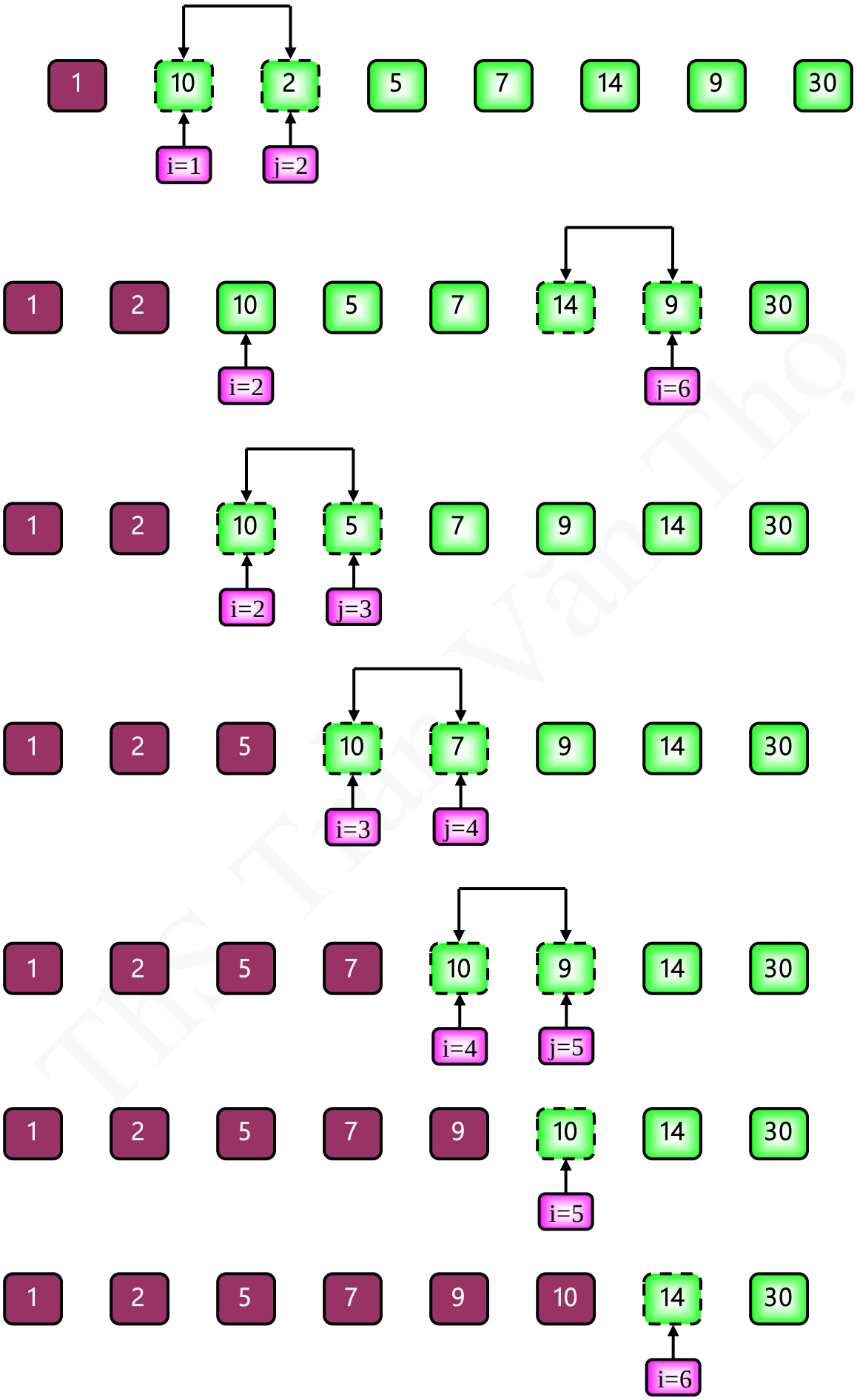
  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

2.4.3.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:




10 2 1 14 5 9 30 7
i=0
j=7
10 2 1 14 5 9 7 30
i=0
j=6
10 2 1 14 5 7 9 30
i=0
j=4
10 2 1 5 14 7 9 30
i=0
j=2
10 1 2 5 14 7 9 30
i=0
j=1
1 10 2 5 14 7 9 30
i=1
j=5



Dừng.

Hình 2.8: Ví dụ minh họa giải thuật sắp xếp nổi bọt

Vậy dãy đã được sắp xếp tăng dần.

2.4.3.3. Cài đặt giải thuật

Hàm bubbleSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

```
1. void bubbleSort_Ascending(ItemType a[ ], int n)
2. {
3.     for(int i = 0; i < n - 1; i++)
4.         for(int j = n - 1; j > i; j--) //tìm phần tử min từ vị trí i+1,..,n-1 đưa về vị trí i
5.             if(a[j] < a[j - 1])
6.                 swap(a[j], a[j - 1]);
7. }
```

2.4.3.4. Đánh giá giải thuật

Chúng ta có thể lập bảng đánh giá giải thuật như sau:

Bảng 2-6: Đánh giá giải thuật sắp xếp nổi bọt

Trường hợp	Số lần so sánh	Số lần gán
Tốt nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	$3\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=3\frac{n(n-1)}{2}$

Vậy độ phức tạp của giải thuật là **O(n^2)**.

2.4.3.5. Nhận xét

- Khi sắp xếp bằng giải thuật Bubble Sort thì các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.
- Bubble Sort có khuyết điểm là không nhận diện được tình trạng dãy đã có thứ tự từng đoạn. Để khắc phục được nhược điểm này của Bubble Sort người ta có giải thuật cải tiến Shaker Sort (*xem mục 2.4.6*).

2.4.4. Giải thuật sắp xếp chọn trực tiếp (Selection Sort)

2.4.4.1. Giải thuật

Ý tưởng giải thuật: Chọn phần tử nhỏ nhất (*lớn nhất*) trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành, sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn (n - 1) phần tử của dãy ban đầu, và bắt đầu từ phần tử ở vị trí thứ 2 trong lần sắp xếp tiếp theo. Lặp lại quá trình xử lý như trên với các phần tử còn lại trong dãy, cho đến khi dãy hiện hành chỉ còn đúng một phần tử thì dừng.

Các bước tiến hành giải thuật:

Dãy ban đầu a có n phần tử chưa được sắp xếp, sẽ thực hiện (n - 1) lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy. Các bước tiến hành như sau:

- **Bước 1:** Khởi gán giá trị biến i: i = 0;
- **Bước 2:** Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[n - 1].
  - + **Bước 2.1:** Gán biến min = i; *//giả sử phần tử tại vị trí i có giá trị nhỏ nhất*
  - + **Bước 2.2:** Tìm giá trị nhỏ nhất trong đoạn [i + 1, n - 1].
    - Khởi gán giá trị biến j: j = i + 1;
    - Lặp lại trong khi (**j < n - 1**) thì thực hiện:

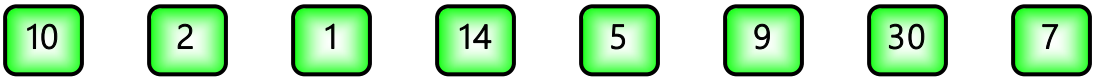
      Nếu (a[j] < a[min]) thì Gán min = j; *//vị trí min mới.*
- **Bước 3:** Hoán vị hai phần tử: a[min] và a[i]
- **Bước 4:** Nếu (**i < n - 1**) thì

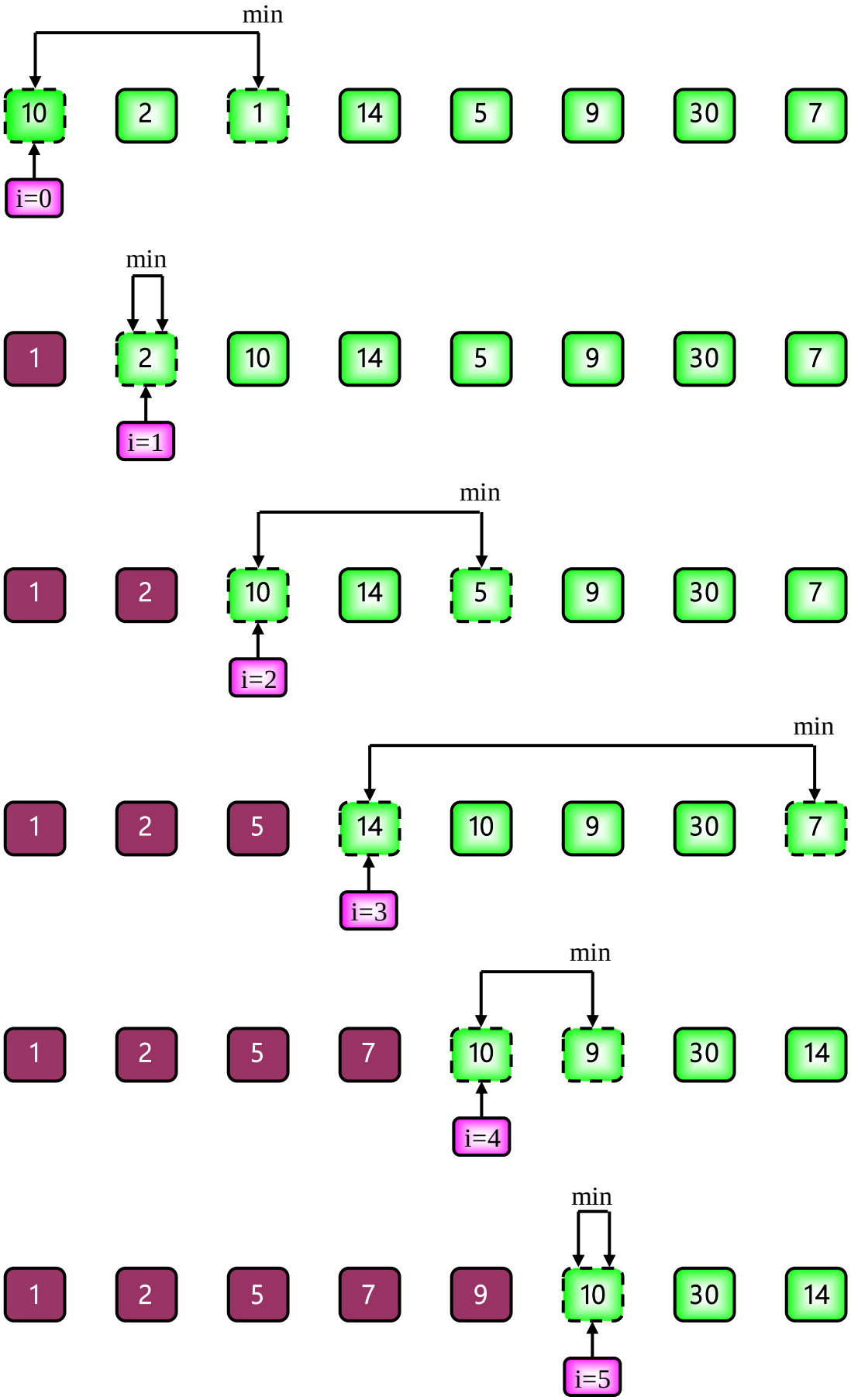
  Tăng giá trị biến i: i = i + 1; Quay lại Bước 2.

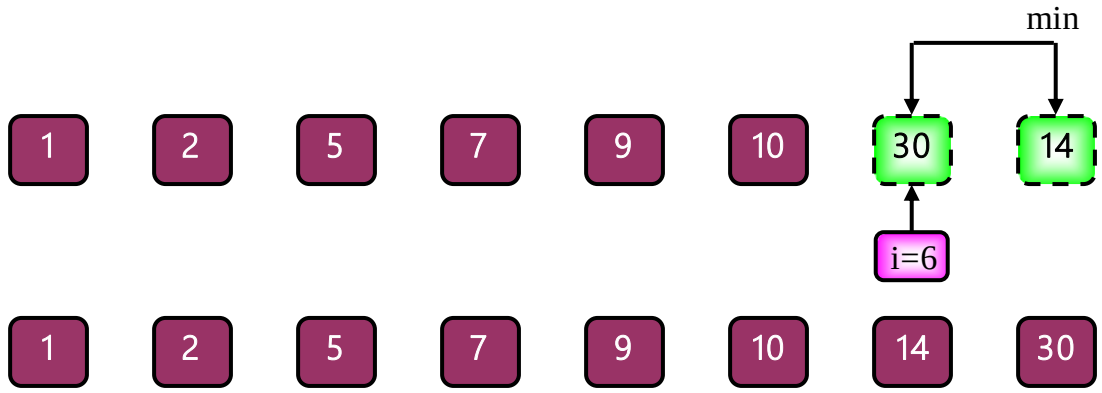
  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

2.4.4.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:




min
10
2
1
14
5
9
30
7
i=0
min
1
2
10
14
5
9
30
7
i=1
min
1
2
10
14
5
9
30
7
i=2
min
1
2
5
14
10
9
30
7
i=3
min
1
2
5
7
10
9
30
14
i=4
min
1
2
5
7
9
10
30
14
i=5



Dừng.

Hình 2.7: Ví dụ minh họa giải thuật sắp xếp chọn trực tiếp

Vậy dãy đã được sắp xếp tăng dần.

2.4.4.3. Cài đặt giải thuật

Hàm selectionSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

```
1.  void selectionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 0; i < n - 1; i++)
4.      {
5.          int min = i;
6.          for(int j = i + 1; j < n; j++)
7.              if(a[j] < a[min])
8.                  min = j;
9.          swap(a[i], a[min]);
10.     }
11. }
```

Cải tiến (*tối ưu*):

```
1.  void selectionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 0; i < n - 1; i++)
4.      {
5.          int min = i;
6.          for(int j = i + 1; j < n; j++)
7.              if(a[j] < a[min])
8.                  min = j; // có phần tử j nhỏ hơn phần tử tại vị trí min
9.          if(min != i)     // nếu có phần tử khác nhỏ hơn phần tử tại vị trí i
10.             swap(a[min], a[i]);
11.     }
12. }
```

2.4.4.4. Đánh giá giải thuật

Chúng ta có thể lập bảng đánh giá giải thuật như sau:

Bảng 2-5: Đánh giá giải thuật sắp xếp chọn trực tiếp

Trường hợp	Số lần so sánh	Số lần gán
Tốt nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	$\sum_{i=0}^{n-2}(4)=4(n-1)$
Xấu nhất	$\sum_{i=0}^{n-2}((n-1)-(i+1)+1)=\frac{n(n-1)}{2}$	$\sum_{i=0}^{n-2}(4+n-i-1)=\frac{(n-1)(n+8)}{2}$

Vậy độ phức tạp của giải thuật là **O(n^2)**.

2.4.4.5. Nhận xét

Đối với giải thuật chọn trực tiếp, có thể thấy rằng ở lượt thứ i, bao giờ cũng cần (n - 1) lần so sánh để xác định phần tử nhỏ nhất hiện hành. Số lượng phép so sánh này không phụ thuộc vào tình trạng của dãy số ban đầu, do vậy trong mọi trường hợp có thể kết luận: Số lần so sánh = $\frac{n(n-1)}{2}$.

2.4.5. Giải thuật sắp xếp chèn trực tiếp (Insertion Sort)

2.4.5.1. Giải thuật

Ý tưởng giải thuật: là một giải thuật sắp xếp bắt chước cách sắp xếp quân bài của những người chơi bài. Muốn sắp một bộ bài theo trật tự người chơi bài rút lần lượt từ quân thứ 2, so với các quân đứng trước nó để chèn vào vị trí thích hợp.

Cơ sở lập luận của giải thuật sắp xếp chèn có thể mô tả như sau: Giả sử có một dãy ban đầu có n phần tử là a_0, a_1, ..., a_{n-1} chưa được sắp xếp. Xét dãy con gồm i phần tử đầu a_0, a_1, ..., a_{i-1}. Với i=1, dãy gồm một phần tử đã được sắp xếp. Giả sử trong dãy có i-1 phần tử đầu a_0, a_1, ..., a_{i-2} đã được sắp xếp, tìm cách chèn phần tử có giá trị bằng x vào dãy có i-1 phần tử đầu thì phải tìm vị trí thích hợp của nó trong dãy a_0, a_1, ..., a_{i-2}, vị trí thích hợp đó là vị trí đứng trước phần tử lớn hơn nó và sau phần tử nhỏ hơn hoặc bằng nó, sau khi chèn phần tử có giá trị x vào dãy có i-1 phần tử đầu a_0, a_1, ..., a_{i-2} sẽ thu được dãy có i phần tử đầu a_0, a_1, ... , a_{i-1} được sắp xếp.

Các phần tử ≤ x	**Các phần tử > x**	**Các phần tử chưa sắp**
a_0, a_1, ..., a_{k-1}, x	, a_k, a_{k+1}, ..., a_{i-1}	, a_i, a_{i+1}, ..., a_{n-1}

Lặp lại quá trình xử lý như trên với các phần tử còn lại trong dãy, cho đến khi dãy hiện hành không còn phần tử nào thì dừng.

Các bước tiến hành giải thuật:

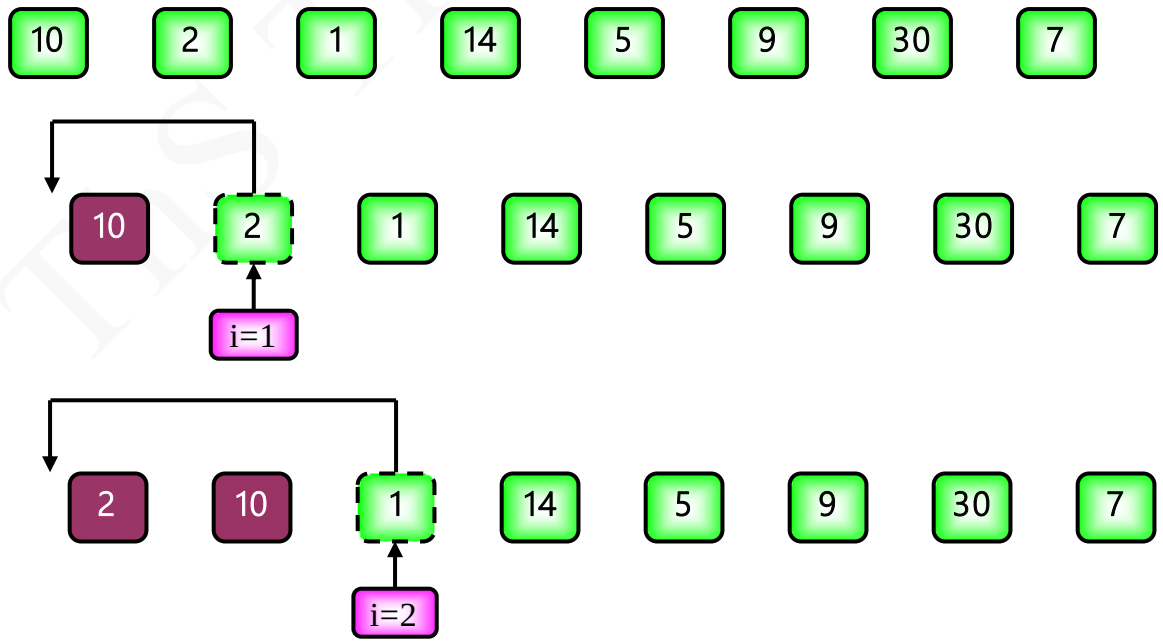
- **Bước 1:** Khởi gán giá trị biến i: i = 1; *//giả sử dãy ban đầu có 1 phần tử là a[0] đã được sắp xếp.*
- **Bước 2:**
  - + **Bước 2.1:** Gán biến x = a[i]; *//giữ lại giá trị cũ của phần tử tại vị trí i*
  - + **Bước 2.2:** Tìm vị trí j thích hợp trong đoạn [0, i - 1] để chèn a[i] vào.
    - Khởi gán giá trị biến j: j = i - 1;
    - Lặp lại trong khi (**j ≥ 0** và **a[j] > x**) thì thực hiện:

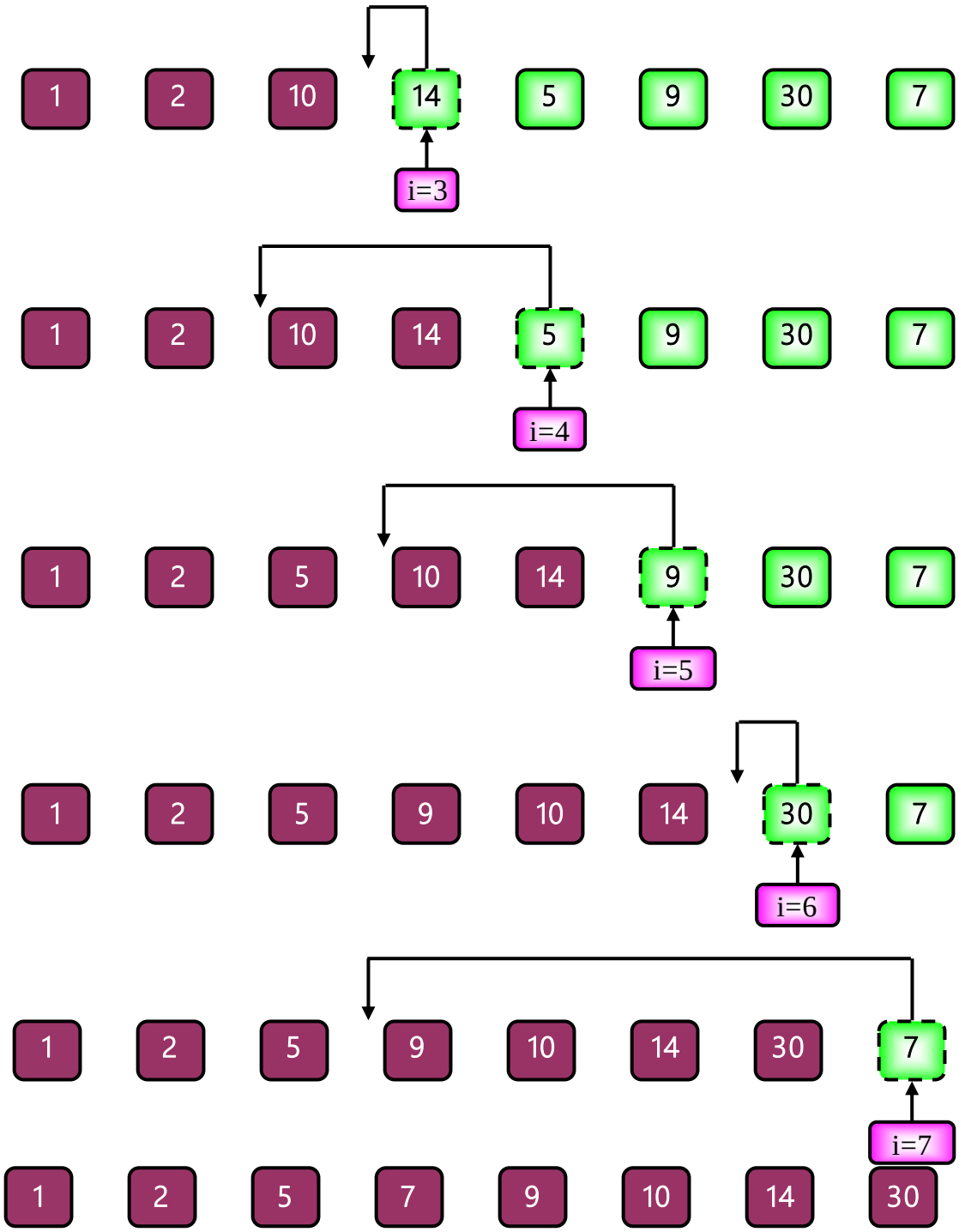
      Gán a[j + 1] = a[j]; *//Dịch chuyển giá trị sang phải 1 vị trí.*
- **Bước 3:** Gán a[j + 1] = x; *//có đoạn a[0]...a[i] đã được sắp xếp.*
- **Bước 4:** Tăng giá trị biến i: i = i + 1;
- **Bước 5:** Nếu (**i < n**) thì: Quay lại Bước 2.

  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

2.4.5.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:





Dừng.

Hình 2.6: Ví dụ minh họa giải thuật sắp xếp chèn trực tiếp

Vậy dãy đã được sắp xếp tăng dần.

2.4.5.3. Cài đặt giải thuật

Hàm insertionSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

1. void **insertionSort_Ascending**(ItemType a[ ], int n)

```
2.  {
3.      for(int i = 1; i < n; i++)
4.      {
5.          ItemType x = a[i]; // Lưu giữ lại giá trị cũ của a[i] để tránh bị ghi đè
                               khi dời chỗ các phần tử.
6.          int j; // Vị trí (position) thích hợp để chèn x vào.
7.          for(j = i - 1; j >= 0; j--)
8.          {
9.              if(a[j] > x)
10.                 a[j + 1] = a[j];
11.             else
12.                 break;
13.         }
14.         a[j + 1] = x;
15.     }
16. }
```

Cách cài đặt khác 1:

```
1.  void insertionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 1; i < n; i++)
4.      {
5.          ItemType x = a[i]; // Lưu giữ lại giá trị cũ của a[i] để tránh bị ghi đè khi
                               dời chỗ các phần tử.
6.          int j = i - 1; // Vị trí (position) thích hợp để chèn x vào.
7.          while(j >= 0)
8.          {
9.              if(a[j] > x)
10.                 a[j + 1] = a[j];
11.             else
12.                 break;
13.             j--;
14.         }
15.         a[j + 1] = x;
16.     }
17. }
```

Cách cài đặt khác 2:

```
1.  void insertionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 1; i < n; i++)
4.      {
5.          ItemType x = a[i]; // Lưu giữ lại giá trị cũ của a[i] để tránh bị ghi đè khi
                               dời chỗ các phần tử.
6.          int j = i - 1;     // Vị trí (position) thích hợp để chèn x vào.
7.          while( (j >= 0) && (a[j] > x) )
8.          {
9.              a[j + 1] = a[j];
```

```
10.             j--;
11.         }
12.         a[j + 1] = x;
13.     }
14. }
```

Cách cài đặt khác 3:

```
1.  void insertionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 1; i < n; i++)
4.      {
5.          ItemType x = a[i]; // Lưu giữ lại giá trị cũ của a[i] để tránh bị ghi đè khi
                               dời chỗ các phần tử.
6.          int j;  // Vị trí (position) thích hợp để chèn x vào.
7.          for(j = i - 1; (j >= 0 && a[j] > x); j--)
8.              a[j + 1] = a[j];
9.          a[j + 1] = x;
10.     }
11. }
```

2.4.5.4. Đánh giá giải thuật

Ta thấy các phép so sánh xảy ra trong vòng lặp nhằm tìm vị trí thích hợp j để chèn giá trị x. Mỗi lần so sánh mà thấy vị trí đang xét không thích hợp sẽ dời phần tử a[j] sang phải.

Ta cũng thấy số phép gán và số phép so sánh của giải thuật phụ thuộc vào tình trạng của dãy ban đầu. Do đó ta chỉ có thể ước lượng như sau:

a. **Trường hợp tốt nhất:** Dãy ban đầu đã có thứ tự. Ta tìm được ngay vị trí thích hợp để chèn ngay lần so sánh đầu tiên mà không cần phải vào thực hiện vòng lặp bên trong. Như vậy, với i chạy từ 1 đến n - 1 thì số phép so sánh tổng cộng sẽ là n - 1. Còn với số phép gán, do giải thuật không chạy vào vòng lặp bên trong nên xét i bất kỳ, ta luôn chỉ phải tốn 2 phép gán (x = a[i] và a[j + 1] = x). Từ đây, ta tính được số phép gán tổng cộng bằng 2(n - 1).

b. **Trường hợp xấu nhất:** Dãy ban đầu có thứ tự ngược. Ta thấy ngay vị trí thích hợp j luôn là vị trí đầu tiên của dãy đã có thứ tự, và do đó, để tìm ra vị trí này ta phải duyệt hết dãy đã có thứ tự. Xét i bất kỳ, ta có số phép so sánh là i - 1, số phép gán là (i - 1) + 2 = i + 1. Với i chạy từ 1 đến n - 1, ta tính được số phép so sánh tổng cộng bằng 1 + 2 + … + (n - 1) = n(n - 1)/2 và số phép gán bằng 3 + 4 + .. + (n + 1) = (n + 4)(n - 1)/2

Từ những phân tích như trên, chúng ta có thể lập bảng đánh giá độ phức tạp của giải thuật như sau:

Bảng 2-4: Đánh giá giải thuật sắp xếp chèn trực tiếp

Trường hợp	Số lần so sánh	Số lần gán
Tốt nhất	$\sum_{i=1}^{n-1}(i) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} 2 = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1}(i) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(i+2) = \frac{n(n-1)}{2} + 2(n-1)$

Vậy độ phức tạp của giải thuật là **O(n^2)**.

2.4.5.5. Nhận xét

- **Ưu điểm:** Số lần gán trong giải thuật chèn trực tiếp giảm ba lần so với giải thuật đổi chỗ trực tiếp.
- **Nhược điểm:** Tìm vị trí chèn với độ phức tạp là **O(n)**.

Khi tìm vị trí thích hợp để chèn a[i] vào đoạn a[0] đến a[i - 1], mà đoạn này đã được sắp xếp, nên có thể sử dụng giải thuật tìm kiếm nhị phân để thực hiện việc tìm kiếm vị trí cần chèn. Vì vậy ta có giải thuật chèn nhị phân như sau:

Giải thuật chèn nhị phân (Binary Insertion Sort):

```
1.  void binaryInsertionSort_Ascending(ItemType a[ ], int n)
2.  {
3.      for(int i = 1; i < n; i++)
4.      {
5.          ItemType x = a[i]; //lưu giá trị a[i] tránh bị ghi đè khi dời chỗ các phần tử.
6.          int Left = 0;
7.          int Right = i – 1;
8.          while(Left <= Right) //tìm vị trí cần chèn x
9.          {
10.             int Mid = (Left + Right) / 2; //tìm vị trí thích hợp Mid
11.             if(x < a[Mid])
12.                 Right = Mid – 1;
13.             else
14.                 Left = Mid + 1;
15.         }
16.         //Dời các phần tử lớn hơn phần tử cần chèn qua phải 1 vị trí.
17.         for(int j = i – 1; j >= Left; j--)
18.             a[j + 1] = a[j]; //dời các phần tử sẽ đứng sau x
19.         a[Left] = x; //chèn x vào dãy
20.     }
21. }
```

2.4.6. Giải thuật sắp xếp Shaker Sort

2.4.6.1. Giải thuật

Ý tưởng giải thuật: Đây là giải thuật cải tiến của giải thuật Bubble Sort nhưng được cải tiến từ sắp xếp 1 chiều thành 2 chiều. Trong mỗi lần sắp xếp, duyệt mảng theo hai lượt từ hai phía khác nhau:

- Lượt đi: Đẩy phần tử nhỏ về đầu mảng.
- Lượt về: Đẩy phần tử lớn về cuối mảng.

Ghi nhận lại những đoạn đã sắp xếp nhằm tiết kiệm các phép so sánh thừa.

Các bước tiến hành giải thuật:

- **Bước 1:** *//Đoạn Left→Right là đoạn cần được sắp xếp*

  + Khởi gán giá trị biến Left: Left = 0;

  + Khởi gán giá trị biến Right: Right = n - 1;

  + Khởi gán giá trị biến k: k = n - 1; *// Vị trí k xảy ra hoán vị sau cùng để làm cơ sở thu hẹp đoạn Left→Right*

- **Bước 2:**

  + **Bước 2.1:**

    Khởi gán giá trị biến j: j = Right; *//đẩy phần tử nhỏ về đầu mảng*

    Lặp lại trong khi (**j > Left**) thì thực hiện:

      Nếu (a[j] < a[j - 1]) thì thực hiện đồng thời 2 công việc sau:

        Hoán vị a[j] với a[j - 1];

        k = j;

      Giảm giá trị biến j: j = j - 1;

    Gán lại giá trị biến Left: Left = k; *//loại phần tử đã có thứ tự ở đầu dãy*

  + **Bước 2.2:**

    i = Left;

    Lặp lại trong khi (**i < Right**) thì thực hiện:

      Nếu (a[i] > a[i + 1]) thì thực hiện đồng thời 2 công việc sau:

        Hoán vị a[i] với a[i + 1];

k = i;

Tăng giá trị biến i: i = i + 1;

Gán lại giá trị biến Right: Right = k; *//loại phần tử đã có thứ tự ở cuối dãy*

- **Bước 3:**

Nếu (**Left < Right**) thì Quay lại Bước 2.

Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

2.4.6.2. Cài đặt giải thuật

Hàm shakerSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhận.

```
1.  void shakerSort_Ascending(ItemType a[ ], int n)
2.  {
3.      int Left = 0;
4.      int Right = n – 1;
5.      int k = n – 1;
6.      while(Left < Right)
7.      {
8.          for(int j = Right; j > Left; j--)  //Tìm phần tử min trong đoạn
                                                 đưa về vị trí Left
9.          {
10.             if(a[j] < a[j - 1])
11.             {
12.                 swap(a[j], a[j – 1]);
13.                 k = j;
14.             }
15.         }
16.         Left = k;
17.         for(int i = Left; i < Right; i++) //Tìm phần tử max trong đoạn
                                                đưa về vị trí Right
18.         {
19.             if(a[i] > a[i + 1])
20.             {
21.                 swap(a[i], a[i + 1]);
22.                 k = i;
23.             }
24.         }
25.         Right = k;
26.     }
27. }
```

2.4.7. Giải thuật sắp xếp vun đống (Heap Sort)

2.4.7.1. Định nghĩa cấu trúc dữ liệu Heap

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử $a_{Left}, a_{Left+1}, ..., a_{Right}$ thỏa các quan hệ với mọi $i \in [Left, Right/2]$:

- $\mathbf{a_i \geq a_{2i}}$
- $\mathbf{a_i \geq a_{2i+1}}$, {*trong đó* (a_i, a_{2i}), (a_i, a_{2i+1}) *là các cặp phần tử liên đới*}

Và có hai tính chất sau:

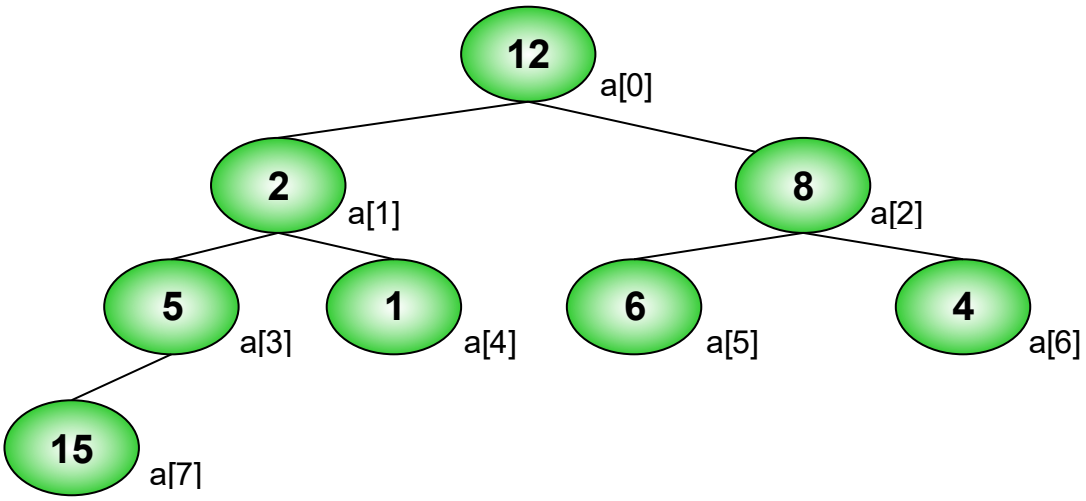
- **Tính chất 1:** Phần tử a_0 (đầu Heap) luôn là phần tử lớn nhất trong Heap.
- **Tính chất 2:** Cắt bỏ một số phần tử về phía phải của Heap (*cuối Heap*) thì dãy con còn lại vẫn là một Heap.

2.4.7.2. Giải thuật

Ý tưởng giải thuật:

- Heap Sort tận dụng được các phép so sánh ở bước i - 1 mà giải thuật sắp xếp chọn trực tiếp không tận dụng được.
- Để làm được điều này Heap Sort thao tác dựa trên cây.

Cho dãy số:	**12**	**2**	**8**	**5**	**1**	**6**	**4**	**15**
Vị trí:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*



- Ở cây trên, phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i + 1, do đó phần tử ở nút gốc là phần tử lớn nhất.

- Nếu loại bỏ gốc ra khỏi cây, thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác thì bảo toàn.
- Bước kế tiếp có thể sử dụng lại kết quả so sánh của bước hiện tại.

Các bước tiến hành giải thuật:

Giải thuật Heap Sort trải qua hai giai đoạn:

- **Giai đoạn 1:** Hiệu chỉnh dãy số ban đầu thành Heap.
- **Giai đoạn 2:** Sắp xếp dãy số dựa trên Heap.
  - + **Bước 1:** Đưa phần tử lớn nhất về vị trí đúng ở cuối dãy.

    Gán giá trị biến Right = n - 1;

    Hoán vị (a_0, a_{Right});
  - + **Bước 2:** Loại bỏ phần tử lớn nhất ra khỏi Heap:

    Gán giá trị biến Right = Right - 1;

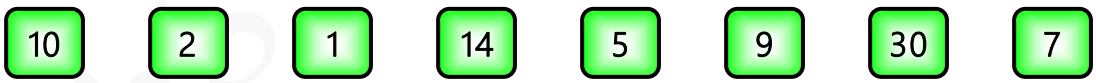
    Hiệu chỉnh phần còn lại của dãy từ a_0, a_1, …, a_{Right} thành một Heap.
  - + **Bước 3:**

    Nếu (**Right > 1**) (*Nghĩa là Heap còn phần tử*) thì Quay lại Bước 2.

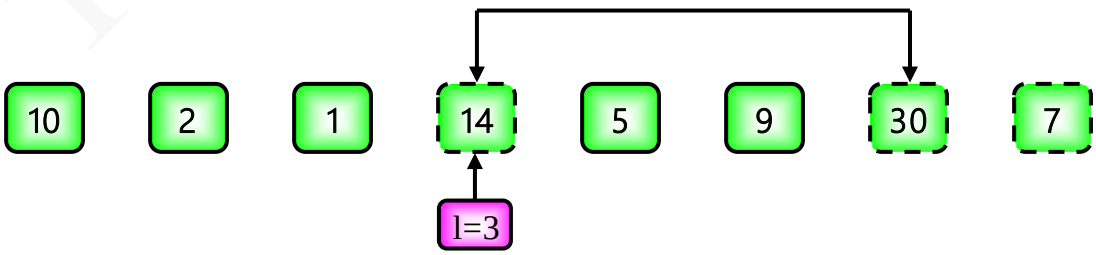
    Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

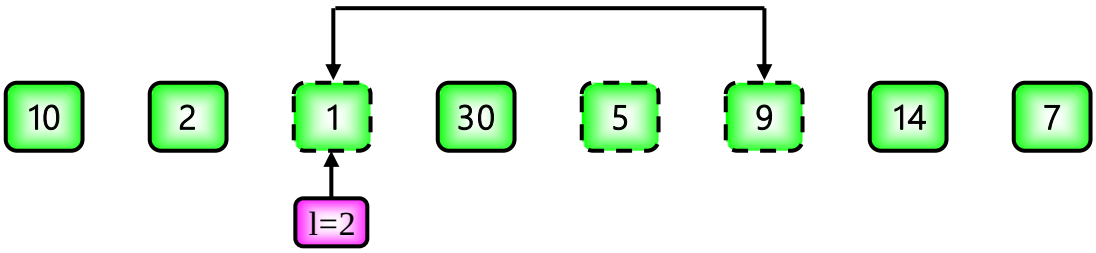
2.4.7.3. Ví dụ minh họa giải thuật

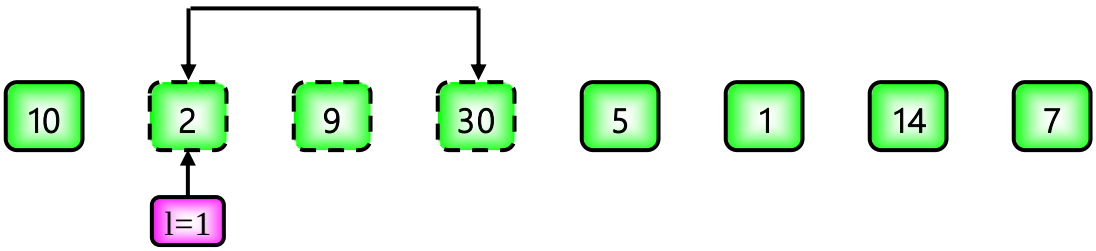
Cho dãy số a gồm **8** phần tử như sau:

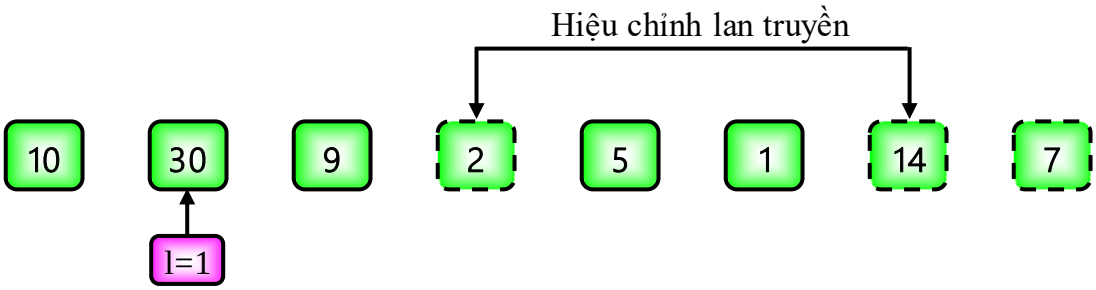


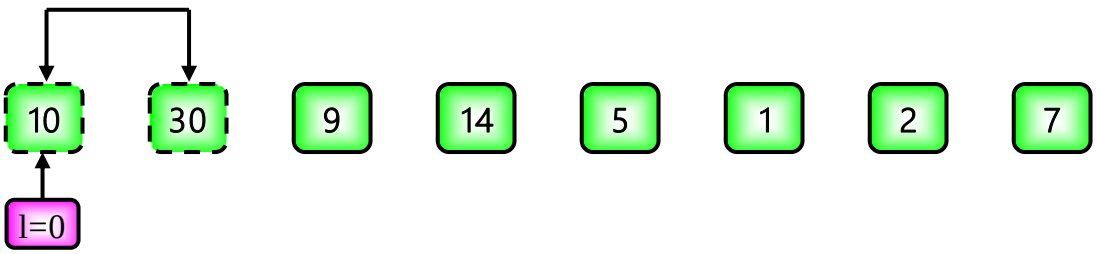
Giai đoạn 1:

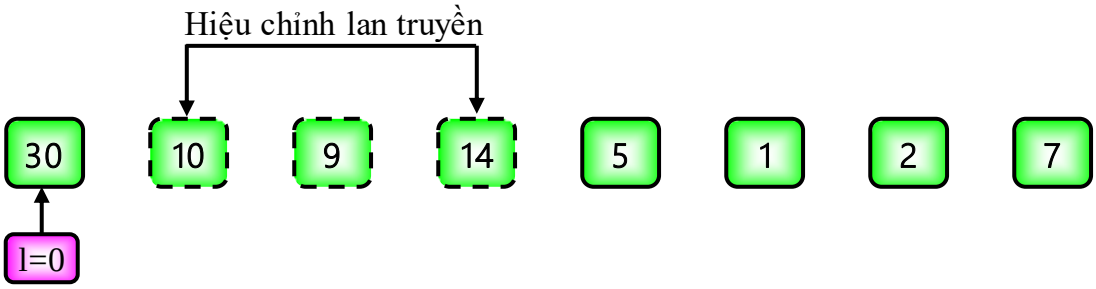



10
2
1
30
5
9
14
7
l=2

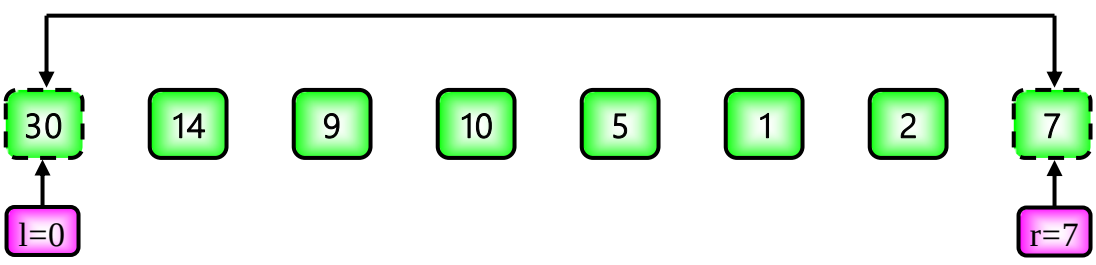

10
2
9
30
5
1
14
7
l=1

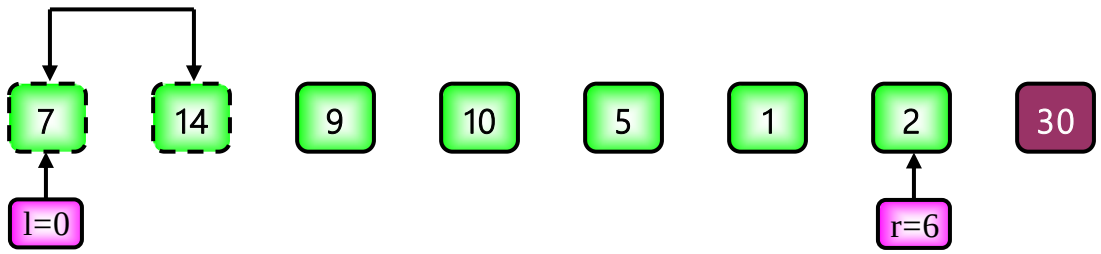

Hiệu chỉnh lan truyền
10
30
9
2
5
1
14
7
l=1

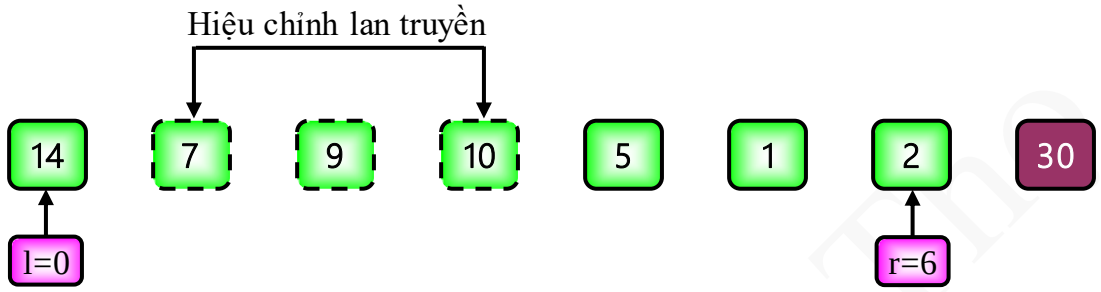

10
30
9
14
5
1
2
7
l=0

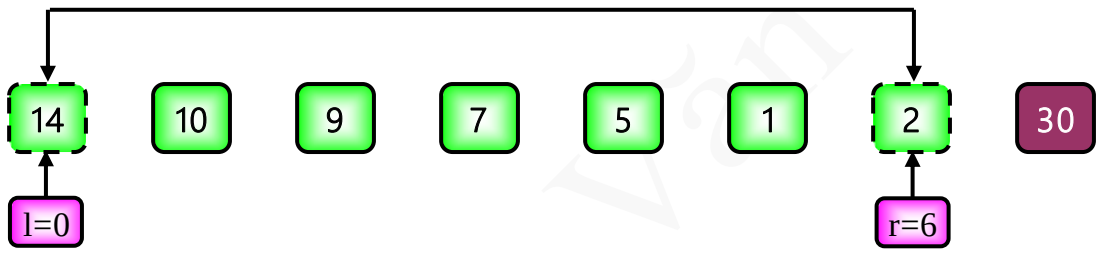

Hiệu chỉnh lan truyền
30
10
9
14
5
1
2
7
l=0

Giai đoạn 2:

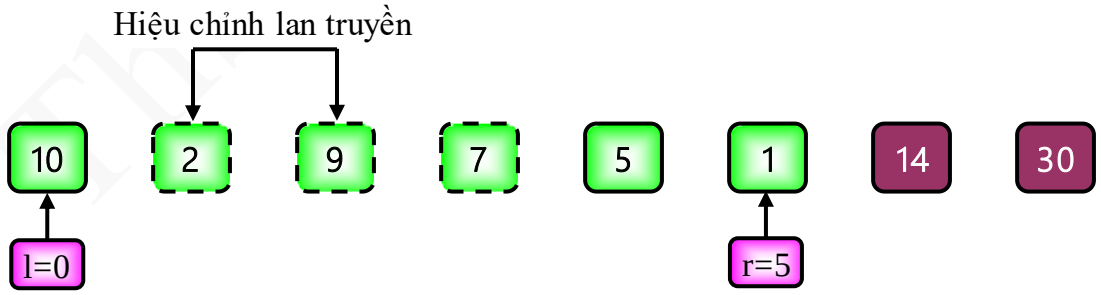

30
14
9
10
5
1
2
7
l=0
r=7

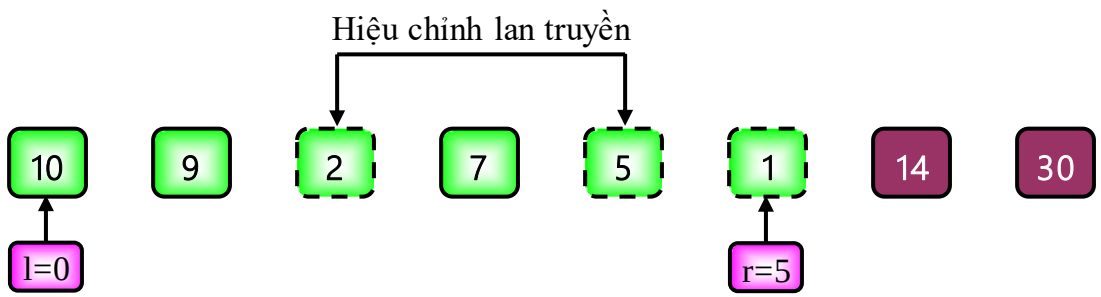

7
14
9
10
5
1
2
30
l=0
r=6

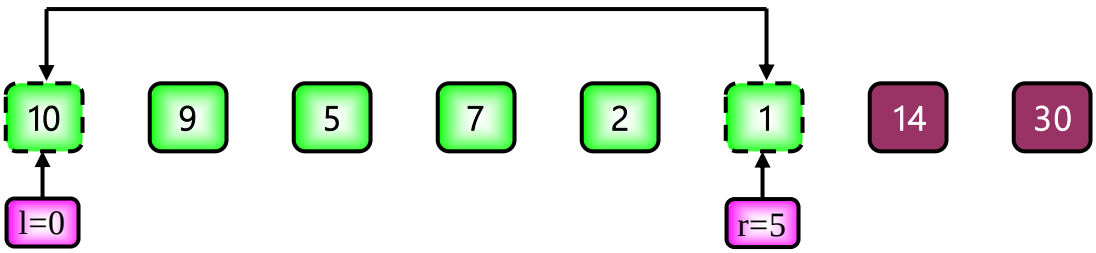

Hiệu chỉnh lan truyền
14
7
9
10
5
1
2
30
l=0
r=6

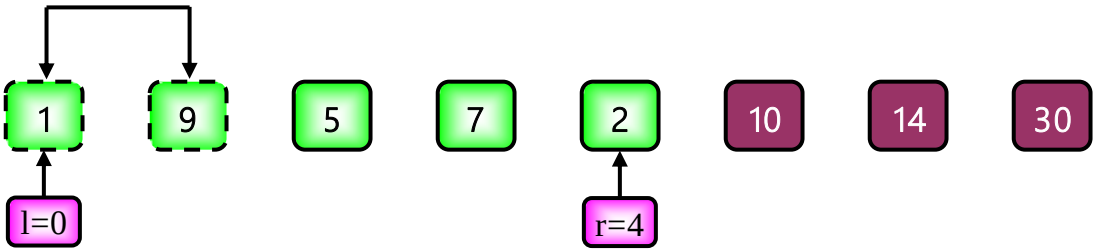

14
10
9
7
5
1
2
30
l=0
r=6


2
10
9
7
5
1
14
30
l=0
r=5

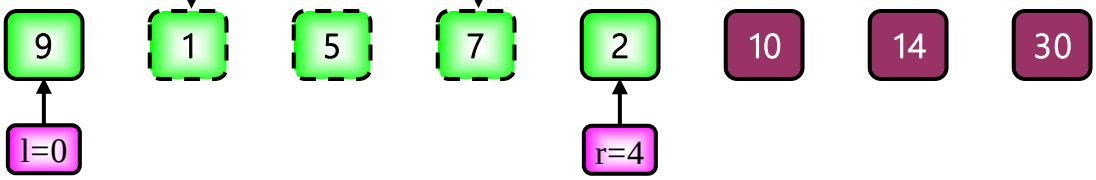

Hiệu chỉnh lan truyền
10
2
9
7
5
1
14
30
l=0
r=5

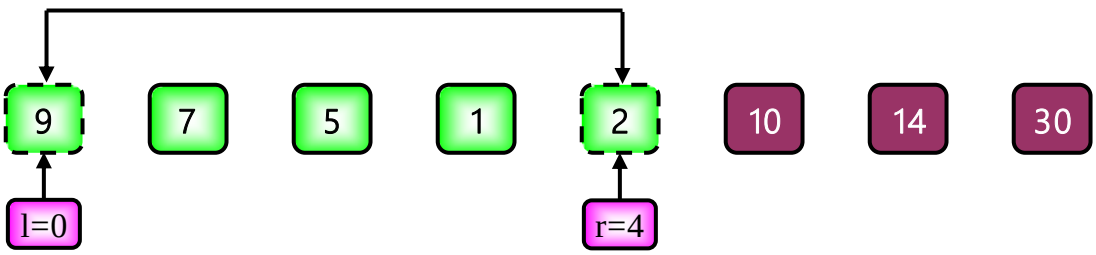

Hiệu chỉnh lan truyền
10
9
2
7
5
1
14
30
l=0
r=5

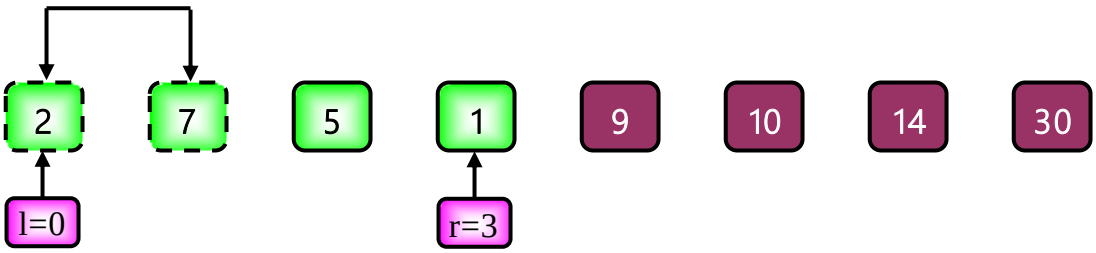




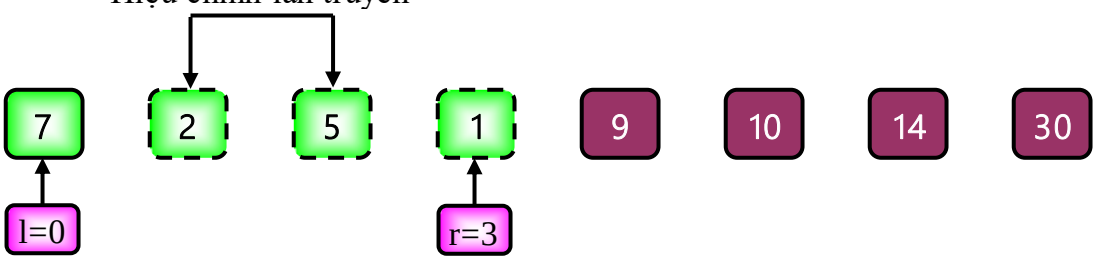
Hiệu chỉnh lan truyền

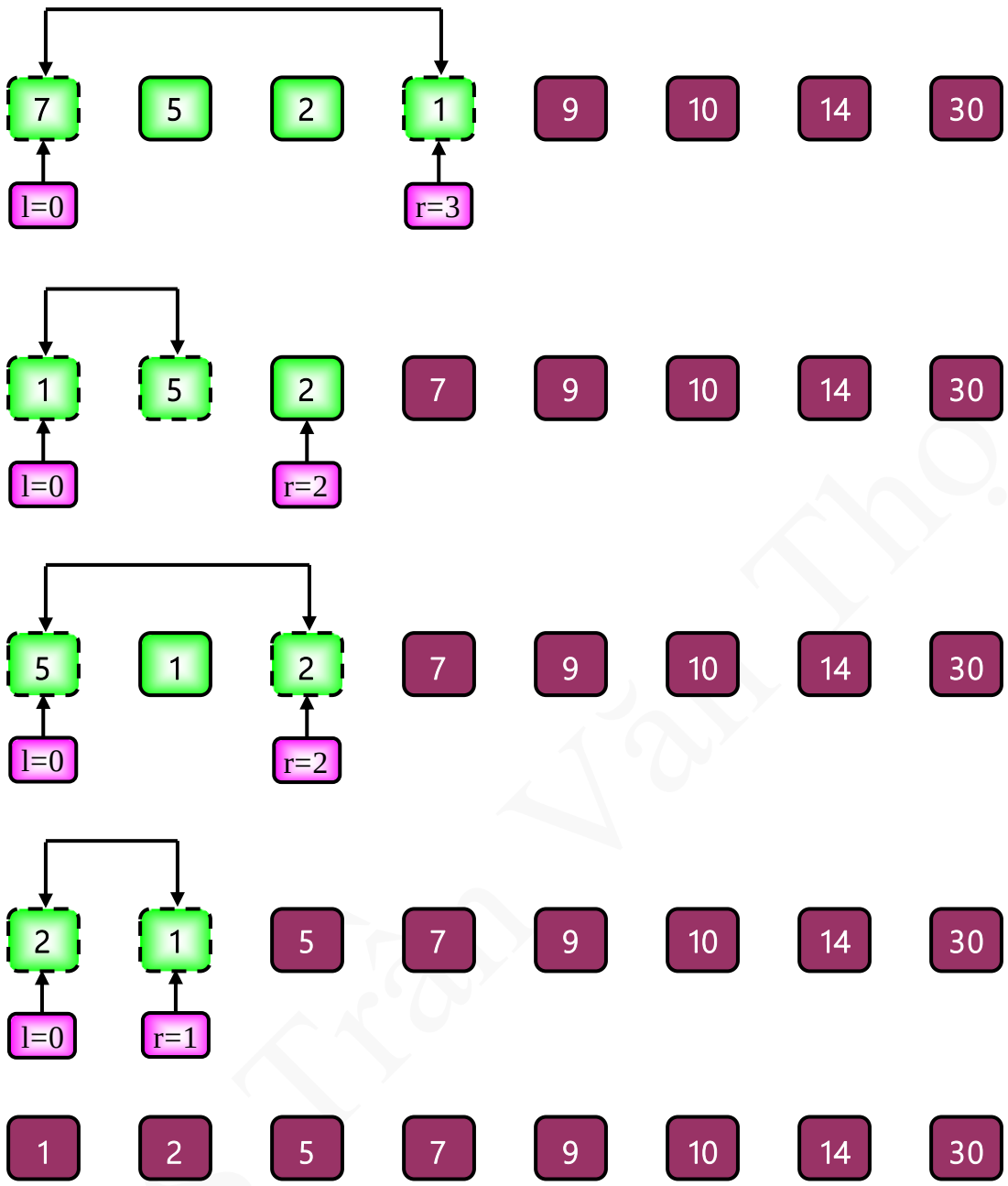






Hiệu chỉnh lan truyền





Dừng.

Hình 2.9: Ví dụ minh họa giải thuật sắp xếp Heap Sort

Vậy dãy đã được sắp xếp tăng dần.

2.4.7.4. Cài đặt giải thuật

Để cài đặt giải thuật Heap Sort cần xây dựng các thủ tục phụ trợ:

2.4.7.4.1. Thủ tục hiệu chỉnh dãy $\mathbf{a_{Left}, ..., a_{Right}}$ thành Heap

Lần lượt xét các quan hệ của một phần tử liên đới của nó trong dãy là a_i nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1}, nếu vi phạm điều kiện quan hệ của

Heap, thì đổi chỗ a_i với phần tử liên đới thích hợp của nó. Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
//Hiệu chỉnh a[Left], a[Left+1], ..., a[Right] thành Heap
1.  void shift(ItemType a[ ], int Left, int Right)
2.  {
3.      if(Left >= Right) return;
4.      int i = Left;
5.      int j = 2 * i; //(ai, aj), (ai, aj+1) là các phần tử liên đới
6.      ItemType x = a[i];
7.      while(j <= Right)
8.      {//thực hiện hiệu chỉnh giữa a[i] với a[j] và a[j+1]
9.          if( (j < Right) && (a[j] < a[j + 1]) )
10.             j = j + 1;
11.         if(a[j] <= x)
12.             return; //dừng không hiệu chỉnh nữa
13.         swap(a[i], a[j]);
14.         i = j;
15.         j = 2 * i;
16.     }
17. }
```

2.4.7.4.2. Hiệu chỉnh dãy $a_0, ..., a_{n-1}$ thành Heap

Cho một dãy bất kỳ $a_{Left}, ..., a_{Right}$, theo tính chất của Heap thì $a_{n/2+1}, a_{n/2+2}, ..., a_{n-1}$ đã là một Heap. Ghép thêm phần tử $a_{n/2}$ vào bên trái Heap hiện hành và hiệu chỉnh lại dãy $a_{n/2}, ..., a_{Right}$ thành Heap.

```
//hiệu chỉnh a[0], a[1], .... , a[n-1] thành heap
1.  void createHeap(int a[ ], int n)
2.  {
3.      int Left = n/2;
4.      while(Left >= 0)
5.      {
6.          Shift(a, Left, n - 1);
7.          Left--;
8.      }
9.  }
```

Khi đó hàm heapSort_Ascending có dạng như sau:

```
1.  void heapSort_Ascending(ItemType a[ ], int n)
2.  {
3.      createHeap(a, n);
4.      int Right = n - 1;
5.      while(Right > 0)
6.      {
7.          swap(a[0], a[Right]);
8.          Right--;
9.          shift(a, 0, Right);
10.     }
```

11. }

2.4.7.5. Đánh giá giải thuật.

Việc đánh giá giải thuật Heap Sort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp ≈**O(nlog$_2$(n))**.

2.4.8. Giải thuật sắp xếp Shell Sort

2.4.8.1. Giải thuật

Ý tưởng giải thuật: Đây là giải thuật cải tiến của giải thuật Insertion Sort. Phân hoạch dãy ban đầu thành các dãy con. Tiến hành sắp xếp các dãy con theo phương pháp chèn trực tiếp. Dùng phương pháp chèn trực tiếp sắp xếp lại cả dãy.

Giải thuật Shell Sort dựa trên ý tưởng sắp xếp các phần tử theo giải thuật chèn trực tiếp nhưng với độ dài bước giảm dần. Ý tưởng của giải thuật sắp xếp này là phân hoạch (*phân chia*) dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau h vị trí:

Dãy ban đầu: a_0 a_2 … a_{n-1} được xem như sự xen kẽ các dãy con sau:

- Dãy con thứ nhất: a_0 a_{h+0} a_{2h+0}
- Dãy con thứ hai: a_1 a_{h+1} a_{2h+1}
- ….
- Dãy con thứ h: a_{h-1} a_{2h-1} a_{3h-1} …

Tiến hành sắp xếp các phần tử trong cùng dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối (chỉ đúng trong dãy con, so với toàn bộ các phần tử trong dãy ban đầu có thể chưa đúng) một cách nhanh chóng, sau đó giảm khoảng cách h để tạo thành các dãy con mới (tạo điều kiện để so sánh một phần tử với nhiều phần tử khác trước đó không ở cùng dãy con với nó) và lại tiếp tục sắp xếp… Giải thuật dừng khi h = 1, lúc này bảo đảm tất cả các phần tử trong dãy ban đầu sẽ được so sánh với nhau để xác định trật tự đúng cuối cùng.

Yếu tố quyết định tính hiệu quả của một giải thuật là cách chọn khoảng cách h trong từng bước sắp xếp. Giả sử quyết định sắp xếp k bước, các khoảng cách chọn phải thỏa điều kiện: $h_i > h_{i+1}$ và $h_k = 1$.

Tuy nhiên, đến nay vẫn chưa có tiêu chuẩn rõ ràng trong việc lựa chọn dãy giá trị khoảng cách tốt nhất, một số dãy được Knuth đề nghị:

- $h_i = (h_{i-1} - 1)/3$ và $h_k = 1$, $k = \log_3 n - 1$

  Ví dụ: 127, 40, 13, 4, 1

- $h_i = (h_{i-1} - 1)/2$ và $h_k = 1$, $k = \log_2 - 1$

Ví dụ: 15, 7, 3, 1

– h có dạng 3i + 1: 364, 121, 40, 13, 4, 1

– Dãy fibonaci: 34, 21, 13, 8, 5, 3, 2, 1

– h là dãy các số nguyên tố giảm dần đến 1: 13, 11, 7, 5, 3, 1.

Các bước tiến hành giải thuật:

- **Bước 1:** Chọn k khoảng cách h[1], h[2], …, h[k] và i = 0.
- **Bước 2:**
  + **Bước 2.1:** Phân hoạch dãy ban đầu thành các dãy con cách nhau h[i] khoảng cách.
  + **Bước 2.2:** Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp.
- **Bước 3:** Tăng giá trị biến i: i = i + 1;

  Nếu (**i ≤ k**) thì: Quay lại Bước 2.

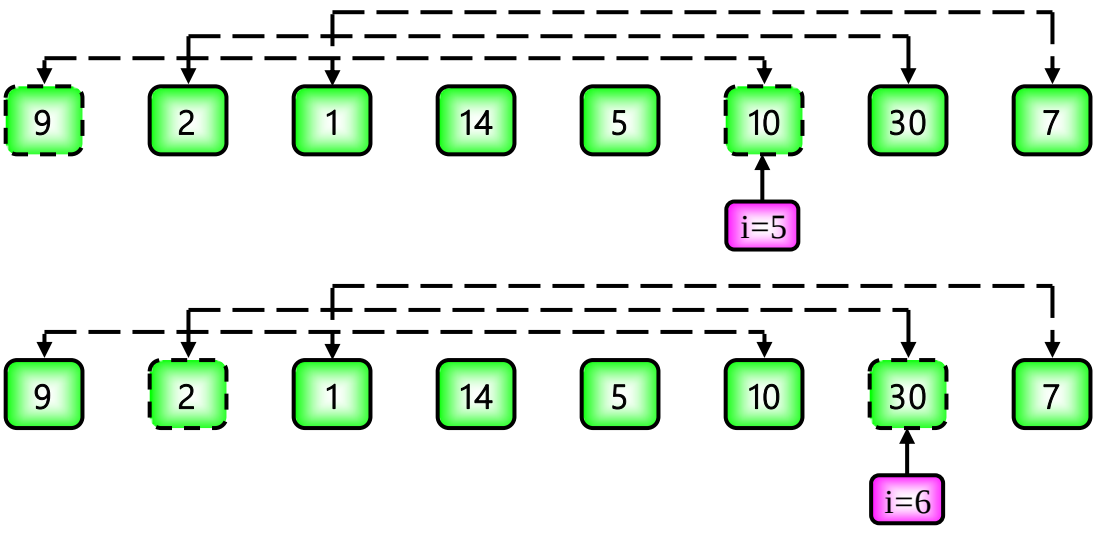
  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

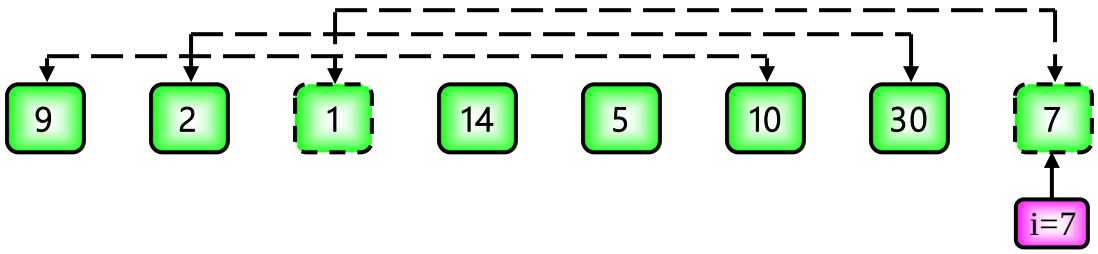
2.4.8.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử sau:

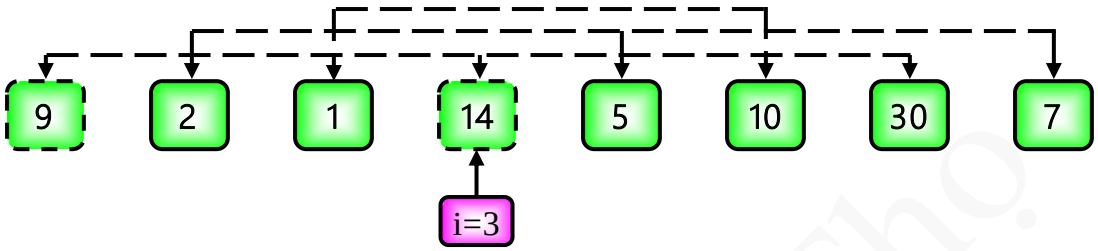
Giả sử chọn các khoảng cách h là: 5, 3, 1.

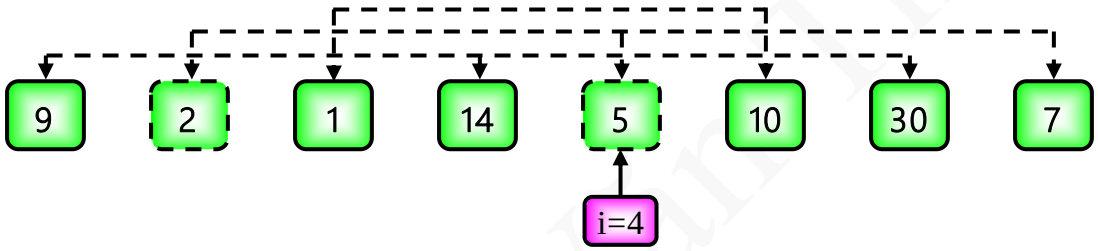
h = 5:

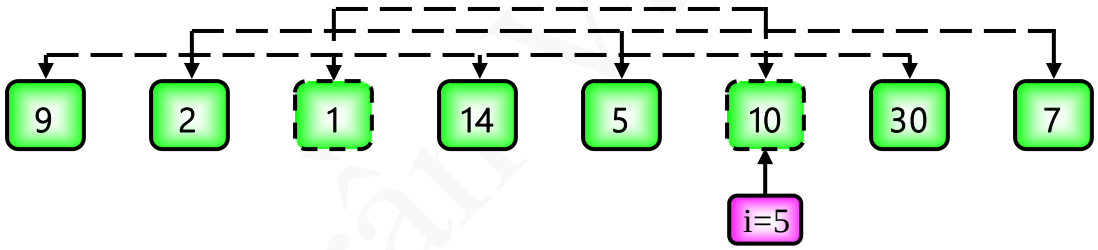


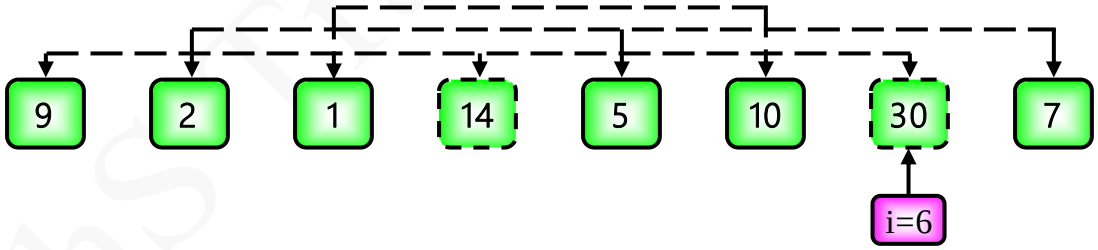

9
2
1
14
5
10
30
7
i=7

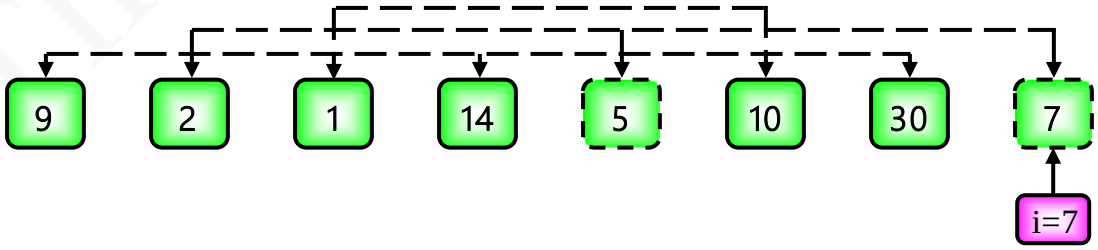
h = 3:


9
2
1
14
5
10
30
7
i=3

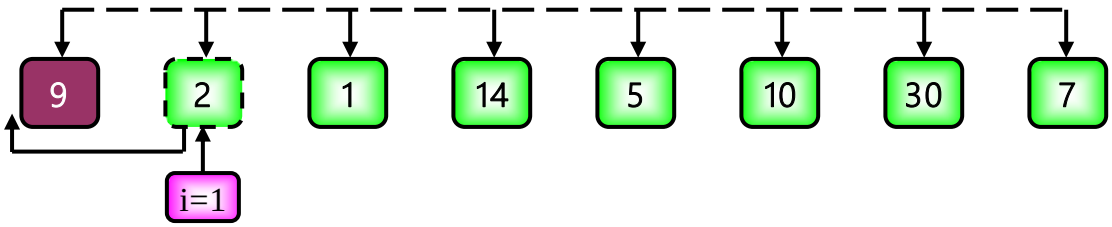

9
2
1
14
5
10
30
7
i=4

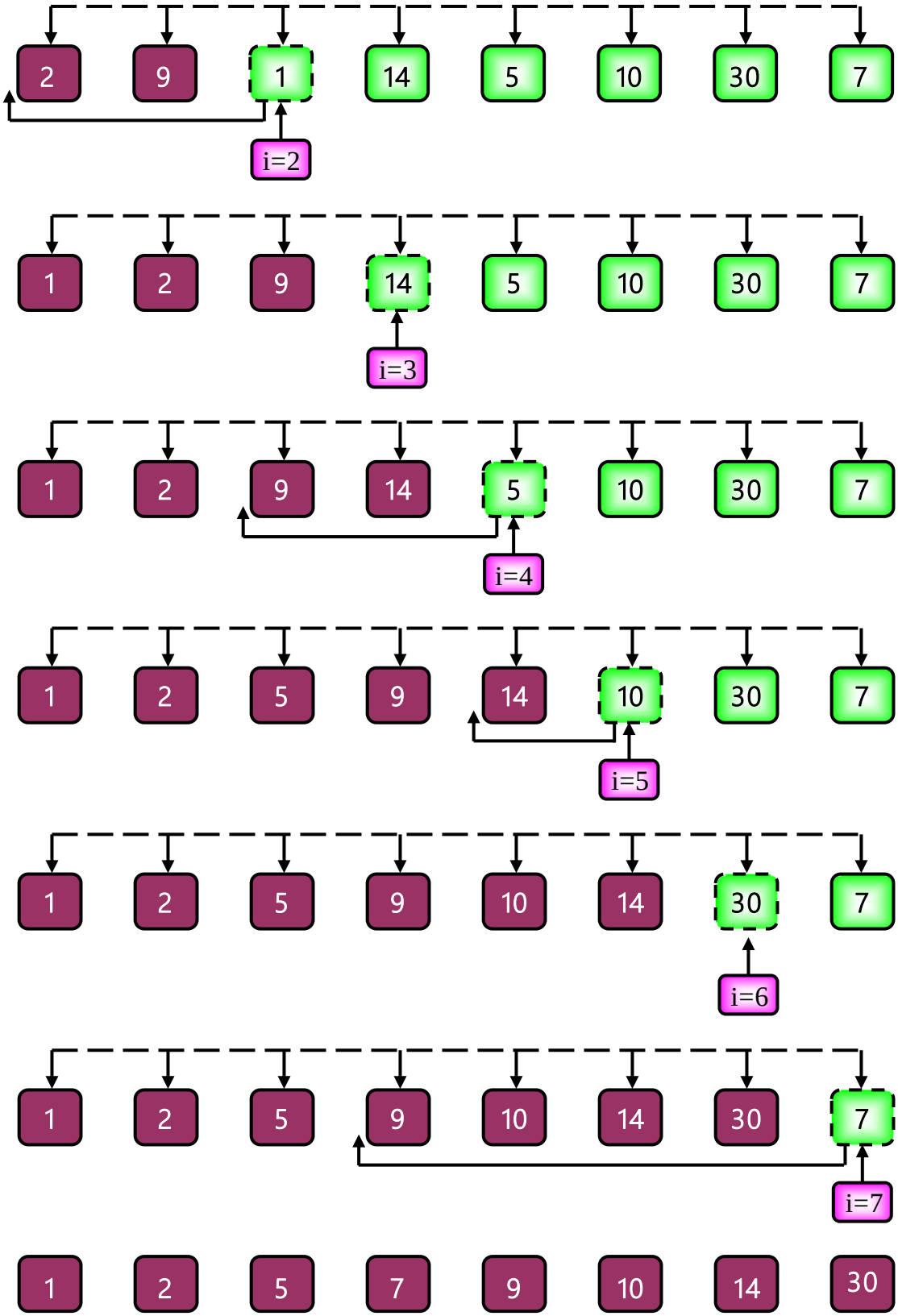

9
2
1
14
5
10
30
7
i=5


9
2
1
14
5
10
30
7
i=6


9
2
1
14
5
10
30
7
i=7

h = 1:


9
2
1
14
5
10
30
7
i=1



Dừng.

Hình 2.10: Ví dụ minh họa giải thuật sắp xếp Shell Sort

Vậy dãy đã được sắp xếp tăng dần.

2.4.8.3. Cài đặt giải thuật

Hàm shellSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập. Giả sử chọn được dãy độ dài h[1], h[2], …, h[k], giải thuật Shell Sort có thể được cài đặt như sau:

```
1.  void shellSort_Ascending(ItemType a[ ], int n, int h)
2.  {
3.      while(h > 0)
4.      {
5.          for(int i = h; i < n; i++)
6.          {
7.              ItemType x = a[i];
8.              int j = i – h;
9.              while( (j >= 0) && (x < a[j]) )
10.             {
11.                 a[j + h] = a[j];
12.                 j = j – h;
13.             }
14.             a[j + h] = x;
15.         }
16.         h = h - 2;
17.     }
18. }
```

2.4.8.4. Đánh giá giải thuật

Việc đánh giá giải thuật Shell Sort rất phức tạp. Đây là giải thuật nhanh nhất và hiệu quả nhất trong các giải thuật sắp xếp lớp $O(n^2)$. Tuy nhiên hiệu quả của giải thuật còn phụ thuộc vào dãy các độ dài được chọn. Trong trường hợp chọn dãy có độ dài theo công thức $h_i = h_{i-1} + 2$ và $h_1 = 1$ và $i \in [1, k]$, với $k = \log_2(n) - 1$ thì giải thuật có độ phức tạp $\approx \mathbf{n^{1,2}}$ ($<< \mathbf{n^2}$).

2.4.8.5. Nhận xét

Yếu tố quyết định tính hiệu quả của giải thuật là cách chọn khoảng cách h. Tuy nhiên, đến nay vẫn chưa có tiêu chuẩn rõ ràng trong việc lựa chọn dãy giá trị khoảng cách tốt nhất.

2.4.9. Giải thuật sắp xếp nhanh (Quick Sort)

2.4.9.1. Giải thuật

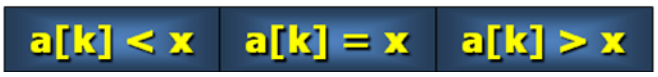
Ý tưởng giải thuật:

Giải thuật Quick Sort sắp xếp dãy $a_0, a_1, \ldots, a_{n-1}$ dựa trên việc phân hoạch dãy ban đầu thành ba phần:

- **Phần 1:** Gồm các phần tử $a_0 \ldots a_{j-1}$ có giá trị không lớn hơn x.
- **Phần 2:** Gồm các phần tử $a_j \ldots a_i$ có giá trị bằng với x.
- **Phần 3:** Gồm các phần tử $a_{i+1} \ldots a_{n-1}$ giá trị không nhỏ hơn x.

Với x là giá trị của một phần tử gọi là phần tử chốt trong dãy ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được phân hoạch làm ba dãy con (*ba đoạn*) như sau:

- **Dãy con thứ 1:** Gồm những $\mathbf{a_k < x}$, với k = 0..j-1
- **Dãy con thứ 2:** Gồm những $\mathbf{a_k = x}$, với k = j..i
- **Dãy con thứ 3:** Gồm những $\mathbf{a_k > x}$, với k = i+1..n-1.



Trong đó dãy con thứ 2 đã có thứ tự, nếu các dãy con 1 và 3 chỉ có một phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp xếp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn một phần tử thì dãy ban đầu chỉ được sắp xếp khi các dãy con 1 và 3 có thứ tự. Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành phân hoạch từng dãy con theo cùng phương pháp phân hoạch giống như dãy ban đầu vừa trình bày.

Các bước tiến hành giải thuật:

- **Bước 1:**

Trong dãy từ $\mathbf{a_{Left}, \ldots, a_{Right}}$ Chọn một phần tử làm giá trị chốt là **a**[Mid], với Mid = (Left + Right) / 2;

x = a[Mid]; i = Left; j = Right;

- **Bước 2:**

Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ:

+ **Bước 2.1:** Lặp lại trong khi (a[i] < x) thì thực hiện tăng i: i++;

+ **Bước 2.2:** Lặp lại trong khi (a[j] > x) thì thực hiện giảm j: j--;

+ **Bước 2.3:** Nếu ($i \le j$) thì thực hiện đồng thời 3 nhiệm vụ sau:

Hoán vị a[i] với a[j]

Thực hiện tăng i: i++;

Thực hiện giảm j: j--;

- **Bước 3:**

+ Nếu (**i < j**) thì: Quay lại Bước 2.

+ Ngược lại: Sang Bước 4.

- **Bước 4:**

+ Quay lại Bước 1 và thực hiện đệ quy sắp xếp cho nữa trái, từ **Left** đến **j**.

+ Quay lại Bước 1 và thực hiện đệ quy sắp xếp cho nữa phải, từ **i** đến **Right**.

Nhận xét:

– Phần tử chốt x được chọn sẽ tác động đến hiệu quả thực hiện giải thuật vì nó quyết định số lần phân hoạch. Số lần phân hoạch sẽ ít nhất nếu ta chọn được x là phần tử có giá trị trung bình (median) của dãy. Tuy nhiên, do chi phí xác định phần tử median quá cao nên trên thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa của dãy làm phần tử chốt với hy vọng nó có thể gần với giá trị median.

– Trong phạm vi giáo trình này để đơn giản và dễ diễn đạt giải thuật nên tác giả chọn phần tử có vị trí chính giữa làm phần tử chốt.

Các bước tiến hành giải thuật theo đệ quy:

Có thể phát biểu giải thuật sắp xếp Quick Sort một cách đệ quy trên dãy $a_{Left} \dots a_{Right}$ như sau:

- **Bước 1:** Nếu (Left ≥ Right) thì Kết thúc. *//dãy có ít hơn 2 phần tử, xem như đã được sắp xếp rồi.*

- **Bước 2:** Phân hoạch dãy $\mathbf{a_{Left} \dots a_{Right}}$ thành các dãy con:

+ Dãy con 1 gồm những phần tử: $a_{Left} \dots a_j < x$

+ Dãy con 2 gồm những phần tử: $a_{j+1} \dots a_{i-1} = x$

+ Dãy con 3 gồm những phần tử: $a_i \dots a_{Right} > x$

- **Bước 3:**

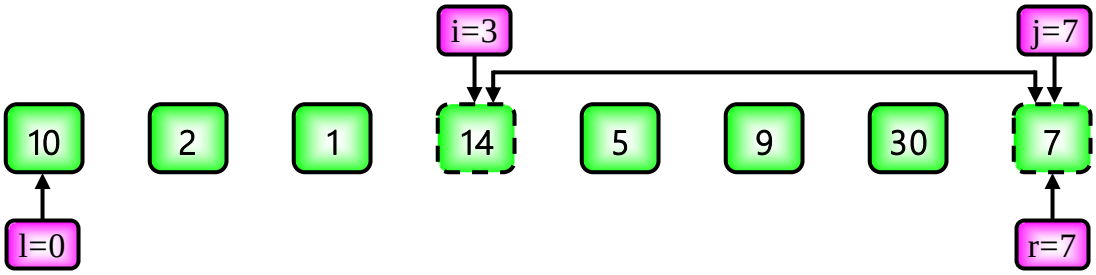
Nếu (**Left < j**) thì Phân hoạch và tiếp tục sắp xếp trên **dãy 1**: $\mathbf{a_{Left} \dots a_j}$

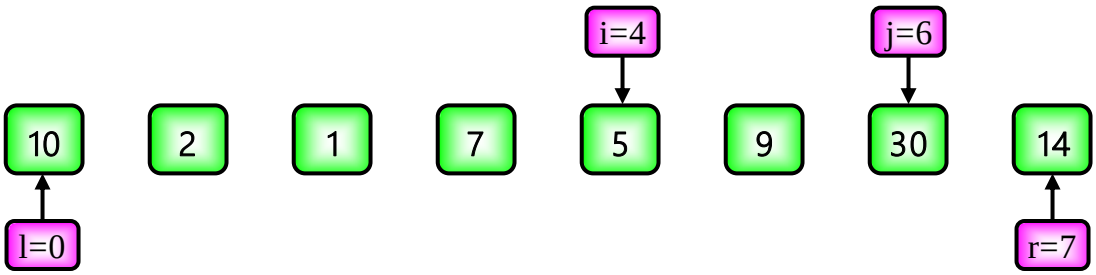
Nếu (**i < Right**) thì Phân hoạch và tiếp tục sắp xếp trên **dãy 3**: $\mathbf{a_i \dots a_{Right}}$

2.4.9.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử sau:

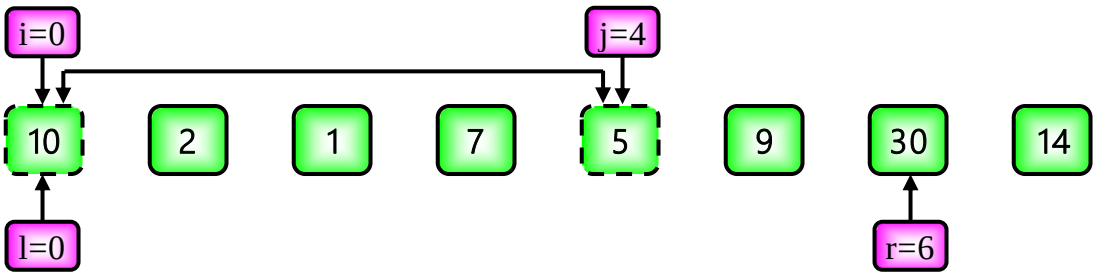
Phân hoạch đoạn Left = 0, Right = 7, Mid = 3, x = 14.

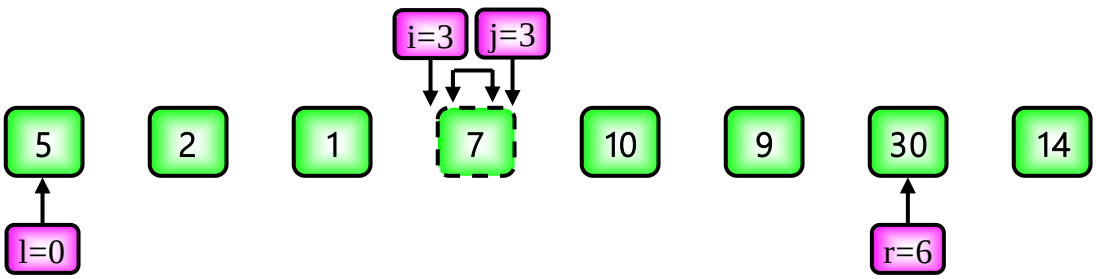


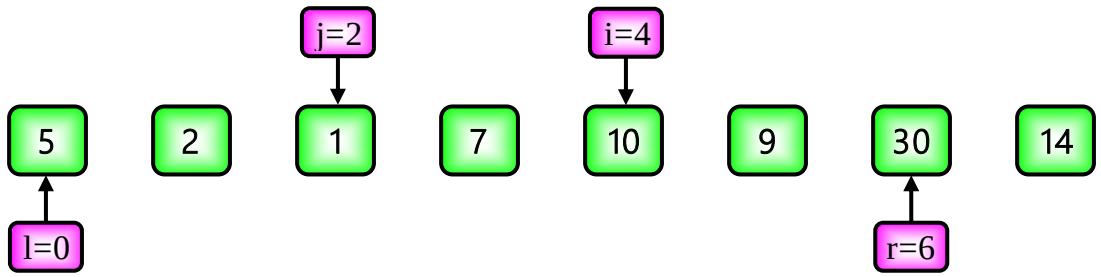


Dừng lặp.

Phân hoạch đoạn Left = 0, Right = j = 6, Mid = 3, x = 7. (chưa làm phân hoạch đoạn Left = i = 6, Right = 7).

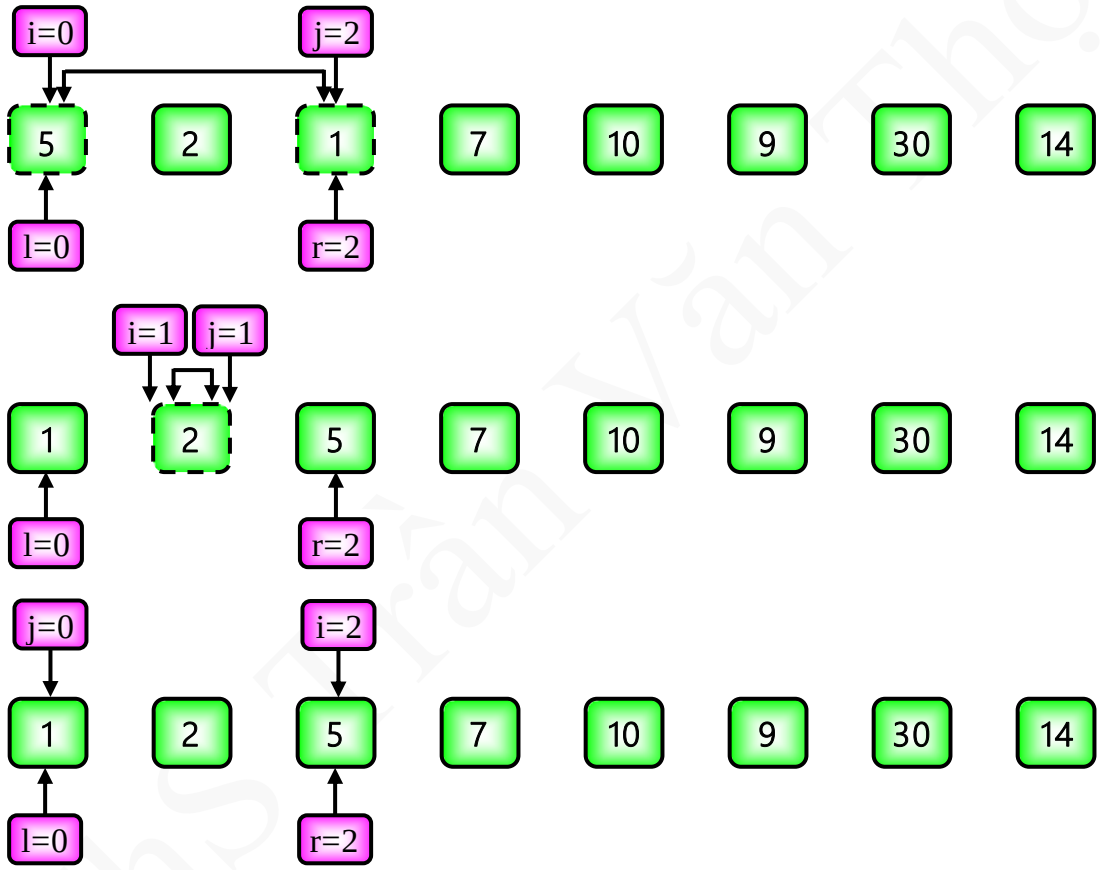






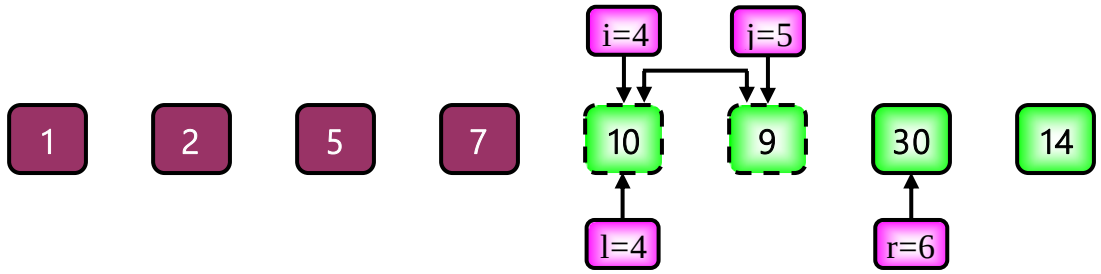
Dừng lặp.

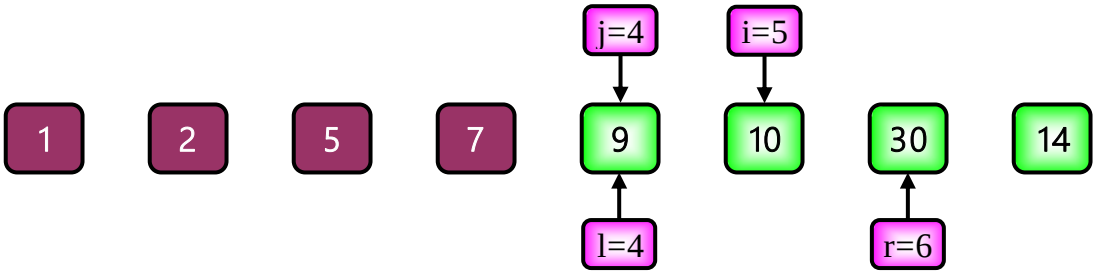
Phân hoạch đoạn Left = 0, Right = j = 2, Mid = 1, x = 2. (chưa làm phân hoạch đoạn Left = i = 4, Right = 6).



Dừng lặp.

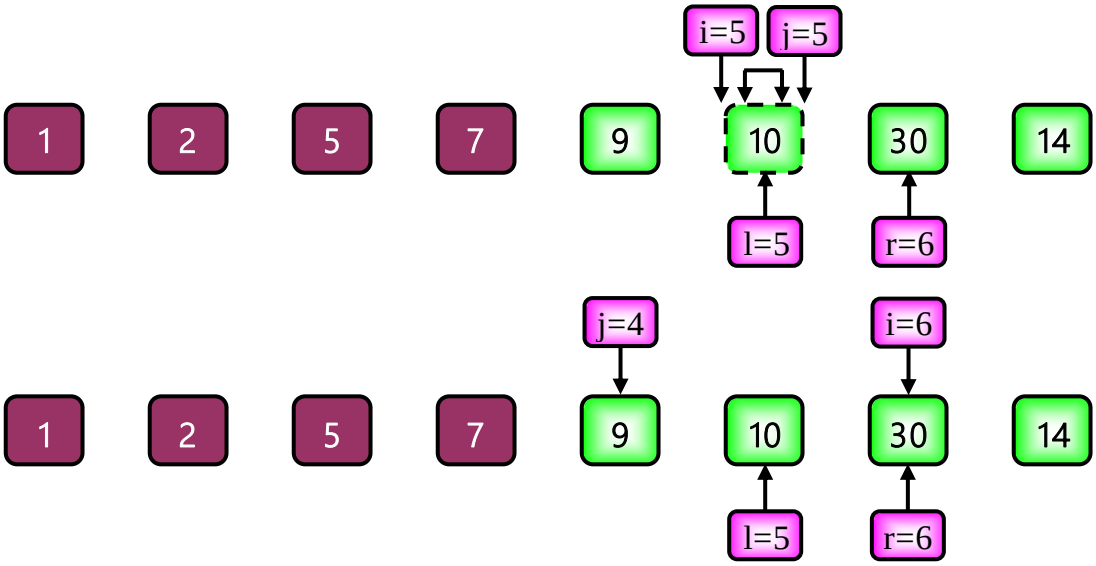
Gọi lại phân hoạch đoạn Left = i = 4, Right = 6, Mid = 5, x = 9.





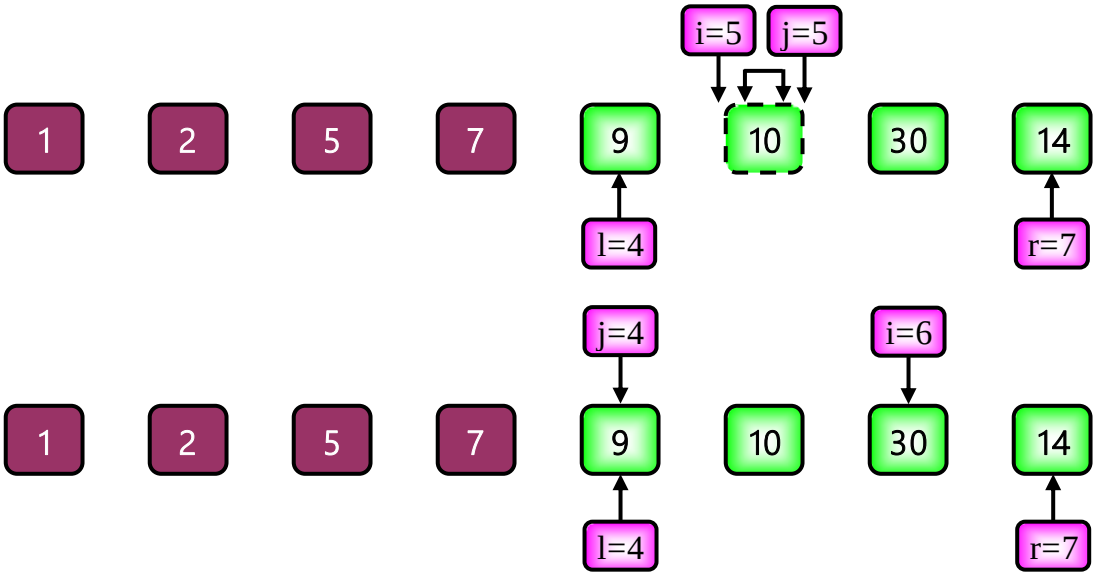
Dừng lặp.

Phân hoạch đoạn Left = i = 5, Right = 6, Mid = 5, x = 10.



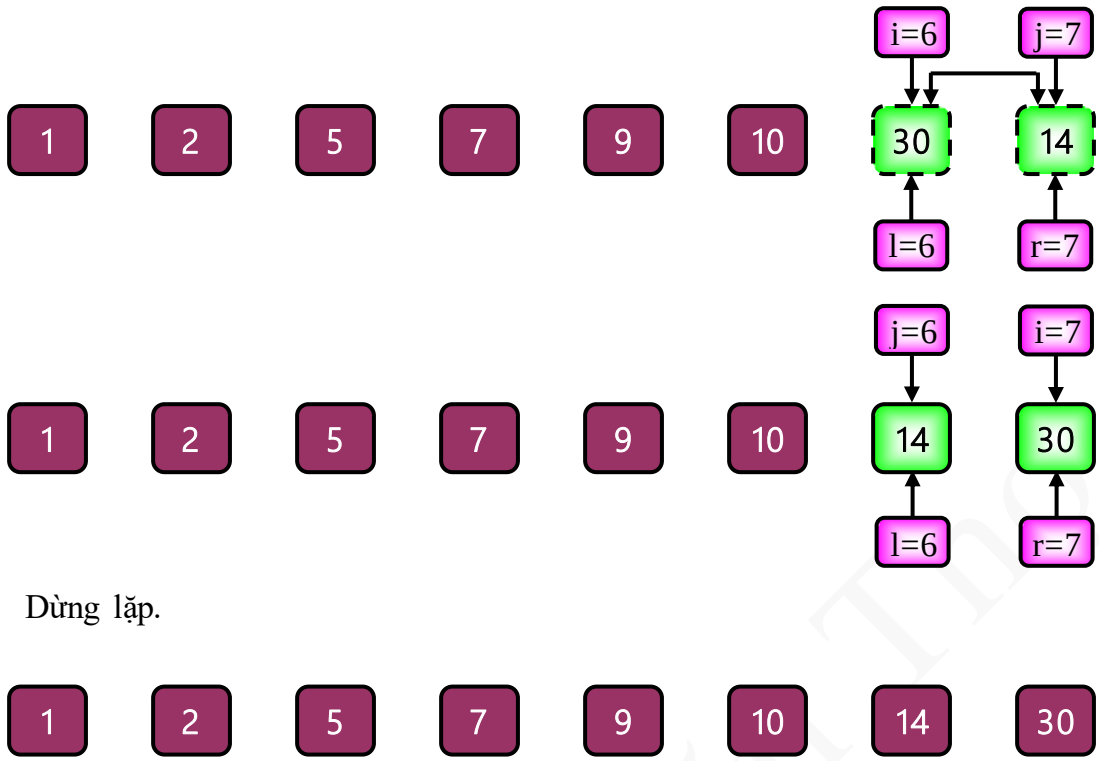
Dừng lặp.

Gọi lại phân hoạch đoạn Left = i = 4, Right = 7, Mid = 5, x = 10.



Dừng lặp.

Phân hoạch đoạn Left = i = 6, Right = 7, Mid = 6, x = 30.



Dừng lặp.

Dừng.

Hình 2.11: Ví dụ minh họa giải thuật sắp xếp Quick Sort

Vậy dãy đã được sắp xếp tăng dần.

2.4.9.3. Cài đặt giải thuật

Hàm quickSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

```
1.  void quickSort_Ascending(ItemType a[ ], int Left, int Right)
2.  {
3.      if(Left >= Right) return; //Điều kiện dừng của đệ quy
4.      int Mid = (Left + Right) / 2;
5.      int i = Left;
6.      int j = Right;
7.      ItemType x = a[Mid];
8.      do
9.      {
10.         while(a[i] < x)
11.             i++;
12.         while(a[j] > x)
13.             j--;
14.         if(i <= j)
15.         {
```

```
16.             swap(a[i], a[j]);
17.             i++;
18.             j--;
19.         }
20.     }while(i < j);
21.     quickSort_Ascending(a, Left, j);      //Đệ quy sắp xếp nữa trái
22.     quickSort_Ascending(a, i, Right);     //Đệ quy sắp xếp nữa phải
23. }
```

2.4.9.4. Đánh giá giải thuật

Ta nhận thấy hiệu quả của giải thuật phụ thuộc vào việc chọn phần tử chốt (*hay giá trị mốc*).

a. **Trường hợp tốt nhất:** mỗi lần phân hoạch ta đều chọn được phần tử median (phần tử lớn hơn hay bằng nửa số phần tử và nhỏ hơn hay bằng nửa số phần tử còn lại) làm phần tử chốt. Khi đó dãy được phân hoạch thành hai phần bằng nhau, và ta cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong. Ta cũng dễ nhận thấy trong mỗi lần phân hoạch ta cần duyệt qua n phần tử. Vậy độ phức tạp trong trường hợp tốt nhất thuộc $\mathbf{O(n\log_2(n))}$.

b. **Trường hợp xấu nhất:** mỗi lần phân hoạch ta chọn phải phần tử có giá trị cực đại hoặc cực tiểu làm phần tử chốt. Khi đó dãy bị phân hoạch thành hai phần không đều: một phần chỉ có một phần tử, phần còn lại có n - 1 phần tử. Do đó, ta cần tới n lần phân hoạch mới sắp xếp xong. Vậy độ phức tạp trong trường hợp xấu nhất thuộc $\mathbf{O(n^2)}$.

Từ những phân tích như trên, chúng ta có thể lập bảng đánh giá giải thuật như sau:

Bảng 2-7: Đánh giá giải thuật sắp xếp Quick Sort

Trường hợp	Độ phức tạp
Tốt nhất	$O(n*\log_2(n))$
Trung bình	$O(n*\log_2(n))$
Xấu nhất	$O(n^2)$

2.4.9.5. Nhận xét

Hiệu quả của giải thuật Quick Sort phụ thuộc vào việc chọn giá trị phần tử chốt. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đều chọn được phần tử median làm mốc, khi đó dãy được phân chia thành hai phần bằng nhau và chỉ cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhằm phần tử có giá trị cực đại hay cực tiểu làm mốc, dãy sẽ bị phân chia thành hai phần không đều: Một phần

chỉ có một phần tử, phần còn lại có (n - 1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong.

2.4.10. Giải thuật sắp xếp trộn trực tiếp (Merge Sort)

2.4.10.1. Giải thuật

- ***Ý tưởng giải thuật:*** Phân hoạch dãy ban đầu $a_0, a_1, \ldots, a_{n-1}$ thành *các dãy con*. Sau khi *phân hoạch xong*, dãy ban đầu sẽ được tách thành hai dãy con (dãy phụ) theo nguyên tắc phân phối đều luân phiên. Rồi trộn *từng cặp dãy con* của hai dãy phụ thành một dãy với nguyên tắc thứ tự tăng dần. Lặp lại quy trình trên sau một số bước, ta sẽ nhận được một dãy chỉ gồm một dãy con không giảm.

- ***Các bước tiến hành giải thuật:***

- **Bước 1:** *//Chuẩn bị*

  k = 1; *// k là chiều dài của dãy con trong bước hiện hành*

- **Bước 2:**

  + **Bước 2.1:** Phân rã (*tách*) dãy $a_0, a_1, ., a_{n-1}$ thành 2 dãy b, c theo nguyên tắc phân phối luân phiên từng nhóm k phần tử:

    b = a_0, ..., a_{2k}, …, a_{4k}, …, a_{6k}, …

    c = a_k, ... , a_{3k}, …, a_{5k}, …, a_{7k}, …

  + **Bước 2.2:** Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào a.

- **Bước 3:**

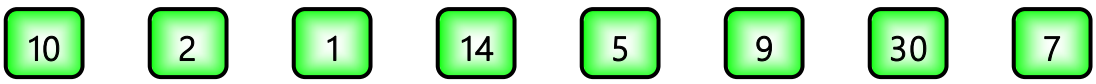
  Tăng giá trị biến k: k = k * 2;

  Nếu (**k < n**) thì: Quay lại Bước 2.

  Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

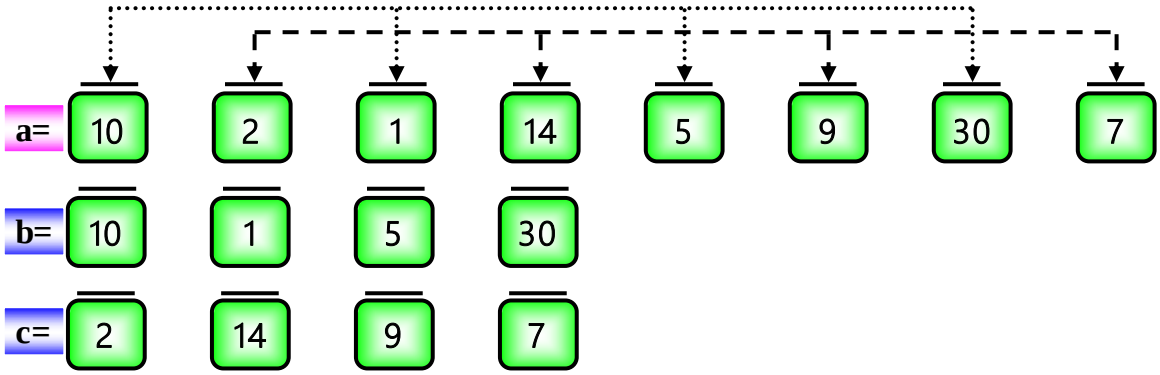
2.4.10.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:



Lần 1 ($k = 2^0 = 1$):

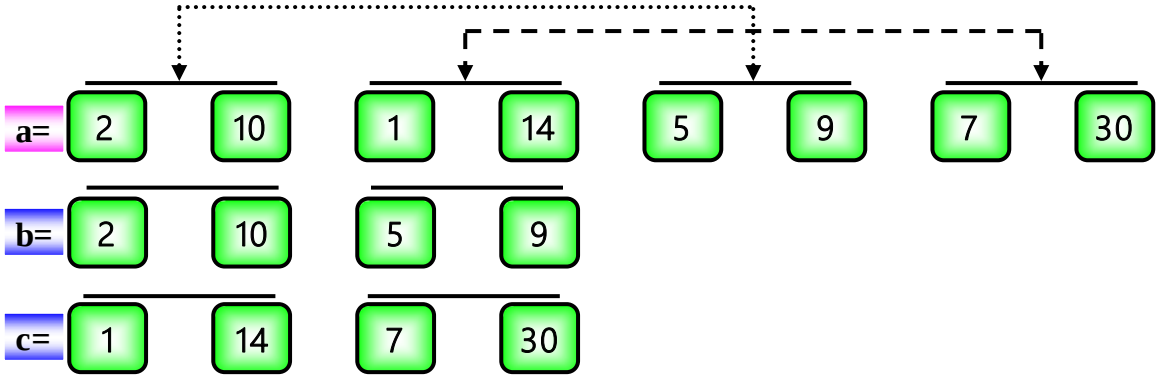
– ***Phân rã dãy a ra hai dãy con b và c lần lượt k phần tử***



– ***Trộn từng bộ k phần tử từ hai dãy con b và c vào lại dãy a***

Lần 2 ($k = k*2 = 1*2 = 2^1 = 2$):

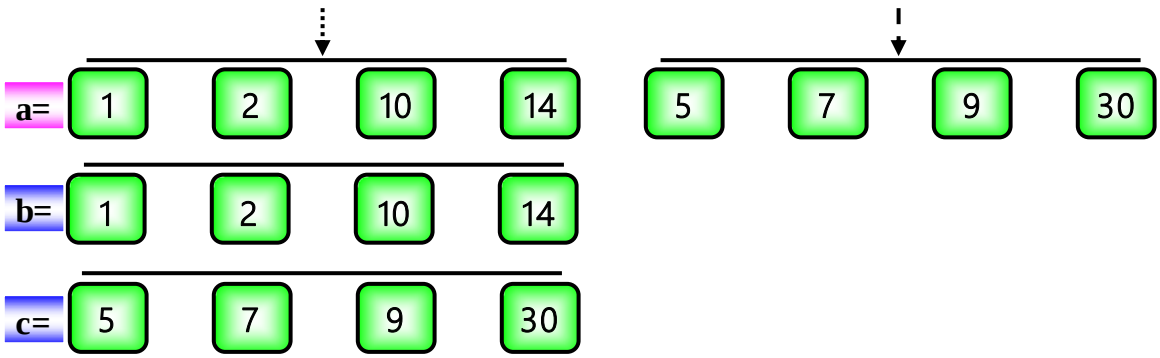
– ***Phân rã dãy a ra hai dãy con b và c lần lượt k phần tử***



– ***Trộn từng bộ k phần tử từ hai dãy con b và c vào lại dãy a***

Lần 3 ($k = k*2 = 2*2 = 2^2 = 4$):

– ***Phân rã dãy a ra hai dãy con b và c lần lượt k phần tử***



– ***Trộn từng bộ k phần tử từ hai dãy con b và c vào lại dãy a***

Lần 4 ($k = k*2 = 4*2 = 2^3 = 8 = n$): Dừng

Hình 2.12: Ví dụ minh họa giải thuật sắp xếp Merge Sort

Vậy dãy đã được sắp xếp tăng dần.

2.4.10.3. Cài đặt giải thuật

Hàm mergeSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung tăng dần và tiến hành sắp xếp ngay trên mảng đã nhập.

Định nghĩa thêm hàm **min** được sử dụng trong giải thuật trên như sau:

```
#define min(a, b) (a > b) ? b : a
```

Hàm trộn lần lượt hai dãy con tăng dần với nhau, và gán lại vào mảng a:

```
1.  void merge(ItemType a[ ], int n, ItemType b[ ], int nb, ItemType c[ ], int nc, int
    k)
2.  {
3.      int ia, ib, ic, jb, jc, kb, kc;
4.      ia = ib = ic = 0;
5.      jb = jc = 0;
6.      while( (nb > 0) && (nc > 0) )
7.      {
8.          kb = Min(k, nb);
9.          kc = Min(k, nc);
10.         if (b[ib + jb] <= c[ic + jc])
11.         {
12.             a[ia++] = b[ib + jb++];
13.             if(jb == kb)
14.                 while(jc < kc)
15.                     a[ia++] = c[ic + jc++];
16.         }
17.         else
18.         {
19.             a[ia++] = c[ic + jc++];
20.             if(jc == kc)
21.                 while(jb < kb)
22.                     a[ia++] = b[ib + jb++];
23.         }
24.         if( (jb == kb) && (jc == kc) )
25.         {
26.             ib += kb;
```

```
            ic += kc;
            nb -= kb;
            nc -= kc;
            jb = jc = 0;
        }
    }
}
```

```
void mergeSort_Ascending(ItemType a[ ], int n)
{
    ItemType b[MAXSIZE], c[MAXSIZE];
    int ia, ib, ic, jc, jb; //các chỉ số trên mảng a, b, c
    int k = 1; //độ dài của dãy con khi phân hoạch
    do
    {
        //Phân bổ lần lượt k phần tử từ mảng a vào 2 mảng b và c
        ia = ib = ic = 0;
        while(ia < n)
        {
            for(jb = 0; (ia < n) && (jb < k); jb++)
                b[ib++] = a[ia++]; //phân bổ k phần tử vào mảng b
            for(jc = 0; (ia < n) && (jc < k); jc++)
                c[ic++] = a[ia++]; //phân bổ k phần tử vào mảng c
        }
        merge(a, n, b, ib, c, ic, k);
        k *= 2;
    }while(k < n);
}
```

2.4.10.4. Đánh giá giải thuật

Do **k = k*2** (*hay* ***k *= 2***) nên số lần lặp của vòng lặp do..while là $\log_2(n)$, và vòng lặp thứ hai lặp n lần. Nên độ phức tạp của giải thuật là **$O(n*\log_2(n))$**.

2.4.10.5. Nhận xét

Nhược điểm lớn nhất của giải thuật là không tận dụng những thông tin đặc tính của dãy sắp xếp. Ví dụ dãy đã có thứ tự sẵn từng đoạn. Vì vậy người ta cần phải cải tiến giải thuật như phần sau.

2.4.11. Giải thuật sắp xếp trộn tự nhiên (Natural Merge Sort)

2.4.11.1. Giải thuật

- ***Ý tưởng giải thuật:*** Phân hoạch *dãy ban đầu* thành *các dãy con*. Sau khi phân hoạch xong, dãy ban đầu sẽ được tách thành hai dãy phụ theo nguyên tắc *phân phối các run có chiều dài cực đại*. Sau đó *trộn từng cặp run có chiều dài cực đại*

của hai dãy phụ thành một dãy với nguyên tắc thứ tự tăng dần. Lặp lại quy trình trên sau một số bước, sẽ nhận được một dãy chỉ gồm một dãy con không giảm.

Khái niệm đường chạy (run):

Một đường chạy của dãy số a là một dãy con không giảm của cực đại của a. Nghĩa là, đường chạy r = (a_i, a_{i+1},…, a_j) phải thỏa mãn hai điều kiện:

+ $0 \le i \le j < n$, với n là số phần tử của mảng a
+ $a_k \le a_{k+1}$, $\forall k, i \le k \le j$

Ví dụ: dãy 12, 2, 8, 15, 1, 6, 10, 17, 4, 11 có thể xem như gồm 4 đường chạy (12); (2, 8, 15); (1, 6, 10, 17); và (4, 11).

Các bước tiến hành giải thuật:

- **Bước 1:** *//Chuẩn bị*

  run = 0; *// run dùng để đếm số đường chạy (số run)*

- **Bước 2:**

  + **Bước 2.1:** Phân rã (*tách*) dãy a_0, a_1, ., a_{n-1} thành 2 dãy b, c theo nguyên tắc phân phối luân phiên từng run:

    Phân phối cho b một đường chạy; run = run + 1;

    Nếu (a còn phần tử chưa phân phối) thì

      Phân phối cho c một đường chạy; run = run + 1;

  + **Bước 2.2:**

    Nếu (a còn phần tử) thì: Quay lại Bước 2.1;

– **Bước 3:**

Trộn từng cặp đường chạy của 2 dãy b, c vào a.

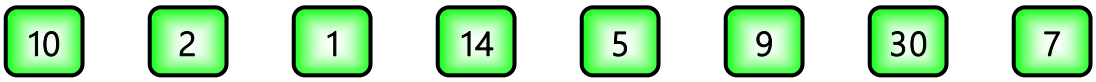
– **Bước 4:**

Nếu run ≥ 2 thì: Quay trở lại Bước 1;

Ngược lại: Hết dãy, Dừng. *//Dãy đã cho đã sắp xếp đúng vị trí.*

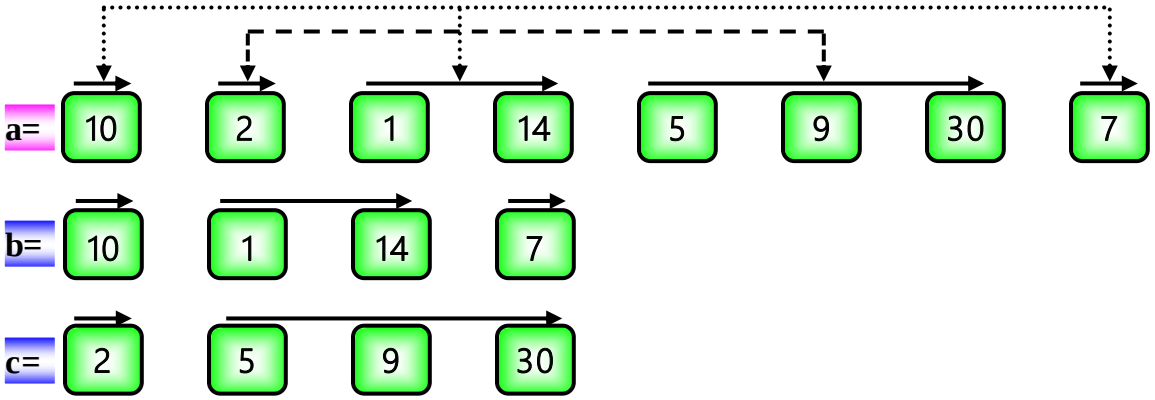
2.4.11.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:



Lần 1:

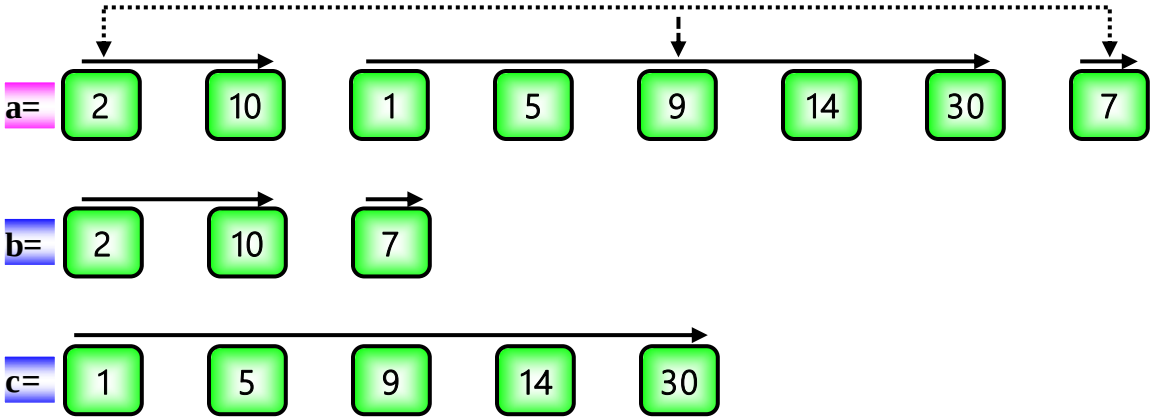
– ***Phân rã dãy a ra hai dãy con b và c lần lượt từng run một***



– ***Trộn lần lượt từng run từ hai dãy con b và c vào lại dãy a***

Lần 2:

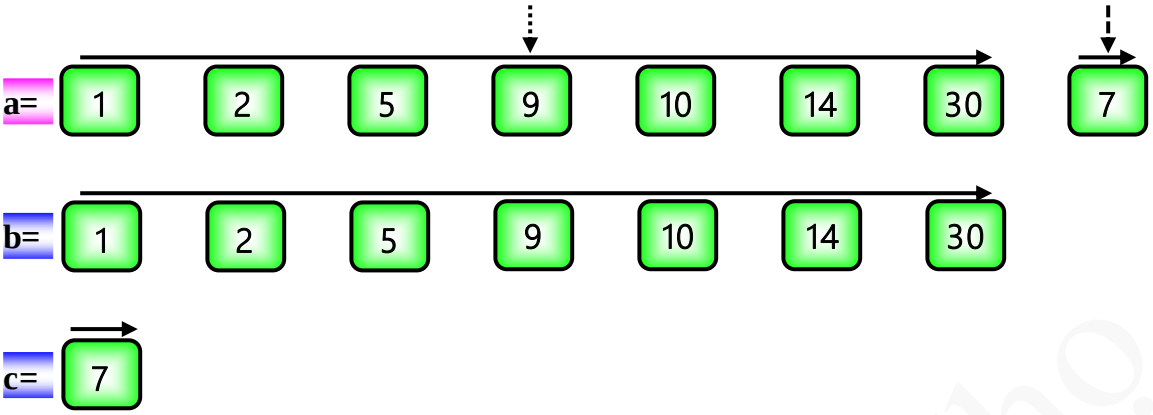
– ***Phân rã dãy a ra hai dãy con b và c lần lượt từng run một***



– ***Trộn lần lượt từng run từ hai dãy con b và c vào lại dãy a***

Lần 3:

– ***Phân rã dãy a ra hai dãy con b và c lần lượt từng run một***



– ***Trộn lần lượt từng run từ hai dãy con b và c vào lại dãy a***

Dừng.

Hình 2.13: Ví dụ minh họa giải thuật sắp xếp trộn tự nhiên

Vậy dãy đã được sắp xếp tăng dần.

2.4.11.3. Cài đặt giải thuật

Hàm naturalMergeSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung và tiến hành sắp xếp ngay trên mảng đã nhập.

```
1.  int lengthOfRun(ItemType a[ ], int pos)
2.  {
3.      int len = 1;
4.      while(a[pos] <= a[pos + 1])
5.      {
6.          len++;
7.          pos++;
8.      }
9.      return len;
10. }
```

```
1.  void naturalMerge(ItemType a[ ], ItemType b[ ], int nb, ItemType c[ ], int nc)
2.  {
3.      int ia, ib, ic, jb, jc, kb, kc;
```

```
4.      ia = ib = ic = 0;
5.      jb = jc = 0;
6.      while( (nb > 0) && (nc > 0) )
7.      {
8.          kb = Min(lengthOfRun(b, ib), nb);
9.          kc = Min(lengthOfRun(c, ic), nc);
10.         if(b[ib + jb] <= c[ic + jc])
11.         {
12.             a[ia++] = b[ib + jb++];
13.             if(jb == kb)
14.                 while(jc < kc)
15.                     a[ia++] = c[ic + jc++];
16.         }
17.         else
18.         {
19.             a[ia++] = c[ic + jc++];
20.             if(jc == kc)
21.                 while(jb < kb)
22.                     a[ia++] = b[ib + jb++];
23.         }
24.         if( (jb == kb) && (jc == kc) )
25.         {
26.             ib += kb;
27.             ic += kc;
28.             nb -= kb;
29.             nc -= kc;
30.             jb = jc = 0;
31.         }
32.     }
33. }
```

```
1.  void naturalMergeSort_Ascending(ItemType a[ ],int n)
2.  {
3.      int k, nb, nc, run;
4.      ItemType b[MAXSIZE], c[MAXSIZE];
5.      do
6.      {
7.          k = nb = nc = 0;
8.          run = 0;
9.          do
10.         {
11.             run++;
12.             do
13.             {
14.                 b[nb++] = a[k++]; //đổ run từ mảng a vào mảng b
15.             }while( (a[k-1] <= a[k]) && (k < n) );
16.             if(k < n)
17.             {
```

```
18.                 run++;
19.                 do
20.                 {
21.                     c[nc++] = a[k++];//đổ run từ mảng a vào mảng c
22.                 }while( (a[k-1] <= a[k]) && (k < n) );
23.             }
24.         }while(k < n);
25.         naturalMerge(a, b, nb, c, nc);
26.     }while(run >= 2);
27. }
```

2.4.11.4. Đánh giá giải thuật

Do số lần lặp của vòng lặp **do..while** nhỏ hơn hoặc bằng $\log_2(n)$, và vòng lặp thứ hai lặp n lần. Nên độ phức tạp của giải thuật là **$O(n*\log_2(n))$**.

2.4.11.5. Nhận xét

Giải thuật trộn tự nhiên khác giải thuật trộn trực tiếp ở chỗ thay vì luôn cứng nhắc phân hoạch theo dãy con có chiều dài k, việc phân hoạch sẽ theo đơn vị là đường chạy. ta chỉ cần biết số đường chạy của a sau lần phân hoạch cuối cùng là có thể biết thời điểm dừng của giải thuật vì dãy đã có thứ tự là dãy chỉ có một đường chạy.

Một nhược điểm lớn nữa của giải thuật trộn là khi cài đặt giải thuật đòi hỏi thêm không gian bộ nhớ để lưu các dãy phụ b và c. Hạn chế này khó chấp nhận trong thực tế vì các dãy cần sắp xếp thường có kích thước lớn. Vì vậy giải thuật trộn thường được dùng để sắp xếp các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hoặc file.

2.4.12. Giải thuật sắp xếp trộn cải tiến khác (Improve Merge Sort)

2.4.12.1. Giải thuật

- ***Ý tưởng giải thuật:*** Merge Sort là phương pháp sắp xếp bằng cách trộn hai danh sách (*dãy con)* đã có thứ tự thành một danh sách có thứ tự. Quá trình này được thực hiện qua nhiều bước trộn liên tục để cuối cùng ta được một danh sách tăng dần.

- ***Các bước tiến hành giải thuật:***

- Bước 1:

+ Xem danh sách cần sắp xếp có n phần tử tương ứng như là n danh sách con đã có thứ tự, mỗi danh sách chỉ có duy nhất một phần tử.

+ Lần lượt trộn từng cặp phần tử trên 2 danh sách con kế cận nhau thành từng cặp phần tử tăng dần. Sau khi trộn xong chúng ta sẽ có n/2 danh sách con có thứ tự tăng dần, mỗi danh sách có 2 phần tử tăng dần.

- **Bước 2:**

+ Xem danh sách cần sắp xếp có n phần tử tương ứng như là n/2 danh sách con đã có thứ tự, mỗi danh sách có 2 phần tử tăng dần.

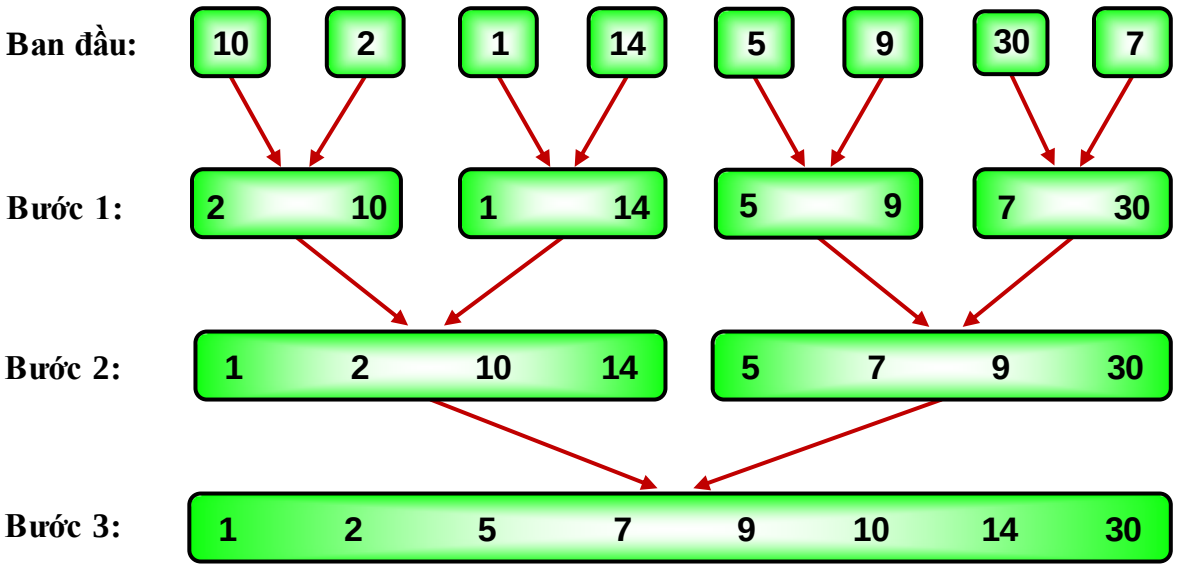
+ Lần lượt trộn từng cặp 2 danh sách con kế cận nhau thành từng dãy con có 4 phần tử tăng dần. Sau khi trộn xong chúng ta sẽ có n/4 danh sách con có thứ tự tăng dần, mỗi danh sách có 4 phần tử tăng dần.

- ...

- Quá trình trên tiếp tục được thực hiện cho đến khi chúng ta có được 1 danh sách có n phần tử tăng dần.

2.4.12.2. Ví dụ minh họa giải thuật

Cho dãy số a gồm **8** phần tử như sau:



Hình 2.14: Ví dụ minh họa giải thuật sắp xếp Merge Sort cải tiến

Vậy dãy đã được sắp xếp tăng dần.

2.4.12.3. Cài đặt giải thuật

Hàm improveMergeSort_Ascending nhận vào một mảng a chứa dãy số gồm n phần tử cần sắp xếp nội dung tăng dần theo khóa chỉ được định (nếu có).

```
1. void improveMergeSort_Ascending(ItemType a[], int n)
2. {
```

```
3.      int i, j, k, size, low1, up1, low2, up2;
4.      ItemType aTemp[MAXSIZE];
5.      size = 1;
6.      while(size < n)
7.      {
8.          low1 = 0;
9.          k = 0;
10.         while(low1 + size < n)
11.         {
12.             low2 = low1 + size;
13.             up1 = low2 - 1;
14.             if(low2 + size - 1 < n)
15.                 up2 = low2 + size - 1;
16.             else
17.                 up2 = n - 1;
18.             i = low1;
19.             j = low2;
20.             while(i <= up1 && j <= up2)
21.                 if(a[i] < a[j])
22.                     aTemp[k++] = a[i++];
23.                 else
24.                     aTemp[k++] = a[j++];
25.             while(i <= up1)
26.                 aTemp[k++] = a[i++];
27.             while(j <= up2)
28.                 aTemp[k++] = a[j++];
29.             low1 = up2 + 1;
30.         }
31.         for(i = low1; k < n; i++)
32.             aTemp[k++] = a[i];
33.         for(i = 0; i < n; i++)
34.             a[i] = aTemp[i];
35.         size *= 2;
36.     }
37. }
```

2.4.12.4. Đánh giá giải thuật

Nếu danh sách có n phần tử thì ta phải tiến hành **$\log_2(n)$** bước trộn.

2.4.13. Giải thuật sắp xếp phân lô (Radix Sort)

2.4.13.1. Giải thuật

Ý tưởng giải thuật: Radix Sort là một giải thuật tiếp cận theo một hướng hoàn toàn khác. Nếu như trong các giải thuật khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì Radix Sort lại dựa trên nguyên tắc phân loại thư của

bưu điện. Vì lý do đó Radix Sort còn có tên là Postman's Sort. Radix Sort không hề quan tâm đến việc so sánh giá trị của phần tử mà bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưu điện thường tổ chức một hệ thống phân loại thư phân cấp. Trước tiên, các thư đến cùng một tỉnh, thành phố sẽ được sắp chung vào một lô để gửi đến tỉnh thành tương ứng. Bưu điện các tỉnh thành này lại thực hiện công việc tương tự. Các thư đến cùng một quận, huyện sẽ được xếp vào chung một lô và gửi đến quận, huyện tương ứng. Cứ như vậy, các bức thư sẽ được trao đến tay người nhận một cách có hệ thống mà công việc sắp xếp thư không quá nặng nhọc.

Mô tả giải thuật:

- Xem mỗi phần tử A[i] trong dãy A[0], ..., A[n-1] là một số nguyên có tối đa m chữ số.
- Lần lượt phân loại các chữ số theo hàng đơn vị, hàng chục, hàng trăm, ... tương tự như việc phân loại thư theo tỉnh thành, quận huyện, phường xã, …
- Tại mỗi bước phân loại tiến hành nối các dãy con từ danh sách đã phân loại theo thứ tự từ 0 đến 9.
- Sau khi phân loại xong ở hàng thứ m cao nhất, chúng ta sẽ thu được danh sách các phần tử được sắp xếp thứ tự yêu cầu.

Các bước tiến hành giải thuật:

- Bước 1:

+ Khởi tạo k = 0 *// k thể hiện chữ số phân loại, k = 0: tương ứng với hàng đơn vị, k = 1: tương ứng với hàng trăm, ...*

- Bước 2: ***//Tạo các lô chứa các phần tử được phân loại B[0], ..., B[9]***

+ Khởi tạo 10 lô: B[0] = B[1] = ... = B[9] = 0 *// khởi tạo rỗng*

+ B[j] sẽ chứa các phần tử có chữ số thứ k là j.

- Bước 3:

+ Khởi tạo i = 0

+ Trong khi i < n thì

Đặt A[i] vào dãy B[j] với j là chữ số thứ k của A[i]

- Bước 4:

+ Nối 10 lô: B[0], B[1], ..., B[9] lại theo đúng trình tự thành dãy A.

- **Bước 5:**

+ Gán k = k + 1 *//Tăng k lên 1 đơn vị để sang hàng mới*

+ Nếu k < m thì quay lại Bước 2 *//m là số lượng chữ số tối đa của các số*

Ngược lại: Hết dãy, Dừng.

2.4.13.2. Ví dụ minh họa giải thuật

Áp dụng giải thuật Radix Sort để sắp xếp tăng dần dãy số a gồm 10 phần tử như sau: 375, 216, 875, 419, 915, 127, 833, 190, 563, 410. Số lượng chữ số tối đa của các phần tử là 3.

- **Phân lô dãy theo hàng đơn vị (k = 0):**

Số hàng đơn vị	Các dãy con
0	19**0**, 41**0**
1	
2	
3	83**3**, 56**3**
4	
5	37**5**, 87**5**, 91**5**
6	21**6**
7	12**7**
8	
9	41**9**

➔ Dãy có được sau khi phân lô theo hàng đơn vị là: **190, 410, 833, 563, 375, 875, 915, 216, 127, 419**

- **Phân lô dãy theo hàng chục (k = 1):**

Số hàng đơn vị	Các dãy con
0	
1	4**1**0, 9**1**5, 2**1**6, 4**1**9
2	1**2**7
3	8**3**3
4	
5	
6	5**6**3
7	3**7**5, 8**7**5
8	

9	190

➔ Dãy có được sau khi phân lô theo hàng đơn vị là: **410, 915, 216, 419, 127, 833, 563, 375, 875, 190**

- **Phân lô dãy theo hàng trăm (k = 2):**

Số hàng đơn vị	Các dãy con
0	
1	127, 190
2	216
3	375
4	410, 419
5	563
6	
7	
8	833, 875
9	915

➔ Dãy có được sau khi phân lô theo hàng đơn vị là: **127, 190, 216, 375, 410, 419, 563, 833, 875, 915**

Vậy dãy đã được sắp xếp tăng dần.

2.4.13.3. Cài đặt giải thuật

```
1.  void radixSort_Ascending(ItemType a[ ], int n, int MaxDigit)
2.  { //Chỉ đúng với mảng dương (giá trị các phần tử >0)
3.      int i, j, k, digit, num;
4.      int h = 10; //biến để lấy các con số, bắt đầu từ hàng đơn vị
5.      long B[10][MAXSIZE]; //mảng 2 chiều chứa các phần tử phân lô
6.      int Len[10]; //kính thước của từng mảng B[i]
7.      for(k = 0; k < MaxDigit; k++)
8.      { //xét lần lượt từng hàng, MaxDigit: là số chữ số tối đa
9.          for(j = 0; j <= 9; j++)
10.             Len[j] = 0; //khởi tạo kích thước các dãy B[j] là 0
11.         for(i = 0; i < n; i++)
12.         { //duyệt qua tất cả các phần tử của mảng
13.             digit = (a[i]%h)/(h/10); //lấy chữ số theo hàng h
14.             B[digit][Len[digit]++] = a[i];
15.         }
16.         num = 0; //chỉ số bắt đầu cho mảng a
17.         for(i = 0; i < 10; i++) //duyệt qua các dãy từ B[0]→B[9]
18.             for(j = 0; j < Len[i]; j++)
19.                 a[num++] = B[i][j];
20.         h *= 10; //qua hàng kế tiếp
```

```
21.     }
22. } //end of RadixSort
```

2.4.13.4. Đánh giá giải thuật

Với một dãy n số, mỗi số có tối đa m chữ số, giải thuật thực hiện m lần các thao tác phân lô và ghép lô. Trong thao tác phân lô, mỗi phần tử chỉ được xét đúng một lần, khi ghép cũng vậy. Như vậy, chi phí cho việc thực hiện giải thuật hiển nhiên là O(2mn) = O(mn) ≈ O(n) vì m << n (m nhỏ hơn n rất nhiều).

2.4.13.5. Nhận xét giải thuật

- Sau lần phân phối thứ k các phần tử của A vào các lô B[0], B[1], …, B[9] và lấy ngược trở ra, nếu chỉ xét đến k + 1 chữ số của các phần tử trong A, ta sẽ có một mảng tăng dần nhờ trình tự lấy ra từ 0 → 9. Nhận xét này bảo đảm tính đúng đắn của giải thuật.
- Giải thuật có độ phức tạp tuyến tính nên hiệu quả khi sắp dãy có rất nhiều phần tử, nhất là khi khóa sắp xếp không quá dài so với số lượng phần tử (điều này thường gặp trong thực tế).
- Giải thuật không có trường hợp xấu nhất và tốt nhất. Mọi dãy số đều được sắp với chi phí như nhau nếu chúng có cùng số phần tử và các khóa có cùng chiều dài.
- Giải thuật cài đặt thuận tiện với các mảng với khóa sắp xếp là chuỗi (ký tự hay số) hơn là khóa số như trong ví dụ do tránh được chi phí lấy các chữ số của từng số.
- Số lượng lô lớn (10 khi dùng số thập phân, 26 khi dùng chuỗi ký tự tiếng Anh, …) nhưng tổng kích thước của tất cả các lô chỉ bằng dãy ban đầu nên ta không thể dùng mảng để biểu diễn B. Như vậy, phải dùng cấu trúc dữ liệu động để biểu diễn B. Nên Radix Sort rất thích hợp cho sắp xếp trên danh sách liên kết.
- Người ta cũng dùng phương pháp phân lô theo biểu diễn nhị phân của khóa sắp xếp. Khi đó ta có thể dùng hoàn toàn cấu trúc dữ liệu mảng để biểu diễn B và chỉ cần dùng hai lô B[0] và B[1]. Tuy nhiên, khi đó chiều dài khóa sẽ lớn. Khi sắp các dãy không nhiều phần tử, giải thuật Radix Sort sẽ mất ưu thế so với các giải thuật khác.

TỔNG KẾT CHƯƠNG

Chương này đã xem xét các giải thuật tìm kiếm và sắp xếp thông dụng. Cấu trúc dữ liệu chính để minh họa các thao tác này chủ yếu là mảng một chiều. Đây cũng là một trong những cấu trúc dữ liệu thông dụng nhất.

Khi khảo sát các giải thuật tìm kiếm, chúng ta đã làm quen với hai giải thuật. Giải thuật thứ nhất là giải thuật tìm kiếm tuần tự, giải thuật này có độ phức tạp tuyến tính O(n). Ưu điểm của nó là tổng quát và có thể mở rộng để thực hiện các bài toán tìm kiếm đa dạng. Tuy nhiên, chi phí giải thuật khá cao nên ít khi được sử dụng. Giải thuật thứ hai là giải thuật tìm kiếm nhị phân, giải thuật này có ưu điểm là tìm kiếm rất nhanh và có độ phức tạp là $\log_2(n)$, nhưng chỉ có thể áp dụng đối với dữ liệu đã có thứ tự theo khóa tìm kiếm. Do đòi hỏi của thực tế, thao tác tìm kiếm phải nhanh vì đây là thao tác có tần suất sử dụng rất cao nên giải thuật tìm kiếm nhị phân thường được dùng nhiều hơn giải thuật tìm kiếm tuần tự. Chính vì vậy xuất hiện nhu cầu phát triển các giải thuật sắp xếp hiệu quả.

Phần tiếp theo của chương trình bày các giải thuật sắp xếp thông dụng theo thứ tự từ đơn giản đến phức tạp (từ chi phí cao đến chi phí thấp).

Phần lớn các giải thuật sắp xếp cơ bản dựa trên sự so sánh giá trị giữa các phần tử. Bắt đầu từ nhóm các giải thuật cơ bản, đơn giản nhất. Đó là các giải thuật đổi chỗ trực tiếp, chèn trực tiếp, chọn trực tiếp, nổi bọt. Các giải thuật này đều có một điểm chung là chi phí thực hiện tỷ lệ với $\mathbf{n^2}$.

Tiếp theo, chúng ta khảo sát một số cải tiến của các giải thuật trên. Nếu như các giải thuật chèn nhị phân (cải tiến của chèn trực tiếp), Shaker Sort (cải tiến của nổi bọt); tuy chi phí có ít hơn các giải thuật gốc nhưng chúng vẫn chỉ là các giải thuật thuộc nhóm có độ phức tạp $O(n^2)$, thì các giải thuật Shell Sort (cải tiến của chèn trực tiếp), Heap Sort (cải tiến của chọn trực tiếp) lại có độ phức tạp nhỏ hơn hẳn các giải thuật gốc. Giải thuật Shell Sort có độ phức tạp $O(n^x)$ với $1 < x < 2$ và giải thuật Heap Sort có độ phức tạp $O(n\log_2 n)$.

Các giải thuật Quick Sort và Merge Sort là những giải thuật thực hiện theo chiến lược chia để trị. Cài đặt chúng tuy phức tạp hơn các giải thuật khác nhưng chi phí thực hiện lại thấp, cả hai giải thuật đều có độ phức tạp $O(n\log_2 n)$. Giải thuật Merge Sort có nhược điểm là cần dùng thêm bộ nhớ đệm, giải thuật này sẽ phát huy tốt ưu điểm của mình hơn khi cài đặt trên các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hay file.

Giải thuật Quick Sort, như tên gọi của mình được đánh giá là giải thuật sắp xếp nhanh nhất trong số các giải thuật sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Tuy có chi phí trong trường hợp xấu nhất là $O(n^2)$ nhưng trong kiểm nghiệm thực tế, giải thuật Quick Sort chạy nhanh hơn hai giải thuật cùng nhóm $O(n\log_2 n)$ là Merge Sort

và Heap Sort. Từ giải thuật Quick Sort, ta cũng có thể xây dựng được một giải thuật hiệu quả tìm phần tử trung vị (median) của một dãy số.

Người ta cũng chứng minh được rằng $O(n\log_2 n)$ là ngưỡng chặn dưới của các giải thuật sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Để vượt qua ngưỡng này, cần phải phát triển giải thuật mới theo hướng khác các giải thuật trên. Giải thuật Radix Sort là một giải thuật như vậy. Nó được phát triển dựa trên sự mô phỏng quy trình phân phối thư của những người đưa thư. Giải thuật này đại diện cho nhóm các giải thuật sắp xếp có độ phức tạp tuyến tính. Tuy nhiên, thường thì các giải thuật này không thích hợp cho việc cài đặt trên cấu trúc dữ liệu mảng một chiều.

Trên thực tế dữ liệu cần thao tác có thể rất lớn do vậy thông thường thì các dữ liệu được lưu trên bộ nhớ thứ cấp, tức trên các đĩa từ. Việc thực hiện các thao tác sắp xếp trên các dữ liệu này đòi hỏi phải có các phương pháp khác thích hợp. Tuy nhiên trong khuôn khổ giáo trình này, các giải thuật trên là tương đối khó, tôi sẽ trình bày trong một cuốn giáo trình khác dành cho trình độ cao hơn.

2.5. Câu hỏi và bài tập

2.5.1. Câu hỏi

1. Xét mảng các số nguyên có nội dung như sau:

   -7 -2 -2 0 5 6 10 16 24.

   a. Tính số lần so sánh để tìm ra phần tử x = -2. Áp dụng cho giải thuật:

      - Tìm kiếm tuyến tính.
      - Tìm kiếm nhị phân.

   Nhận xét và so sánh hai giải thuật tìm kiếm nêu trên trong trường hợp này và trong trường hợp tổng quát.

   b. Trường hợp tìm kiếm nhị phân, phần tử nào sẽ được tìm thấy (thứ 1 hay 2).

2. Xây dựng giải thuật tìm kiếm phần tử nhỏ nhất (lớn nhất) trong một mảng các số nguyên.
3. Cho biết ý tưởng của từng giải thuật sắp xếp trên. Với ý tưởng các giải thuật trên thực hành với ví dụ sau:

   9 4 2 -1 7 14 2 3.

4. Trong ba giải thuật sắp xếp cơ bản (chọn trực tiếp, chèn trực tiếp, nổi bọt) giải thuật nào thực hiện sắp xếp nhanh nhất với một dãy có thứ tự. Giải thích.
5. Cho một ví dụ minh họa ưu điểm của giải thuật Shaker Sort đối với Bubble Sort khi sắp xếp một dãy số.
6. Xét đoạn code cài đặt thao tác phân hoạch trong giải thuật Quick Sort sau đây:

```
int i = 0; j = n - 1; x = a[n/2];
do
{
    while(a[i] < x) i++;
    while(a[j] > x) j--;
    swap(a[i], a[j]);
}while(i <= j);
```

Có dãy a[0], a[1], ..., a[n - 1] nào làm đoạn chương trình trên sai hay không? Cho ví dụ minh họa.

7. Hãy xây dựng giải thuật tìm phần tử trung vị (median) của một dãy số a_0, a_1, ..., a_{n-1} dựa trên giải thuật Quick Sort. Cho biết độ phức tạp của giải thuật này?

2.5.2. Bài tập

8. Cài đặt mười hai giải thuật sắp xếp tăng (và giảm) và hai giải thuật tìm kiếm đã trình bày. Thể hiện trực quan các thao tác giải thuật đó. Tính thời gian thực hiện của mỗi giải thuật.
9. Cài đặt giải thuật sắp xếp tăng các số nguyên trong một mảng bằng giải thuật chọn trực tiếp.
10. Cài đặt hàm tìm dãy con tăng dài nhất của mảng một chiều a có n phần tử.
11. Cài đặt hàm trộn hai mảng một chiều có thứ tự tăng b và c có m và n phần tử thành một mảng một chiều a có m + n phần tử và cũng có thứ tự tăng.
12. Tiếp theo bài tập 13 – chương 1. Hãy cài đặt các giải thuật như sau:
    a. Sắp xếp mảng tăng/giảm theo MaSV (bằng các giải thuật đã học).
    b. Sắp xếp mảng tăng/giảm theo DTB (bằng các giải thuật đã học).
    c. Sắp xếp mảng tăng/giảm theo Ten (bằng các giải thuật đã học).
13. Bằng việc tạo ngẫu nhiên một mảng a có n phần tử nguyên. Thử viết chương trình lập bảng so sánh thời gian thực hiện của các giải thuật sắp xếp.
14. Hãy viết hàm tìm tất cả các số nguyên tố nằm trong mảng một chiều a có n phần tử.
15. Hãy viết hàm tìm dãy con tăng dài nhất của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần tử của a).
16. Hãy viết hàm đếm số đường chạy của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần tử của a).
17. Hãy viết chương trình minh họa trực quan các giải thuật tìm kiếm và sắp xếp để hỗ trợ cho những người học môn cấu trúc dữ liệu.
18. Cài đặt giải thuật tìm phần tử trung vị (median) của một dãy số bạn đã xây dựng trong bài tập 7.

Chương 3. DANH SÁCH LIÊN KẾT

3.1. Giới thiệu

Với các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: Kiểu số thực, Kiểu kí tự, ... hoặc từ các cấu trúc đơn giản khác như: Mẫu tin, Tập hợp, Mảng, … lập trình viên có thể giải quyết hầu hết các bài toán đặt ra. Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này có đặc điểm chung là không thay đổi được kích thước, cấu trúc trong quá trình sống, do đó thường cứng nhắc, gò bó khiến đôi khi khó diễn tả được thực tế vốn sinh động, phong phú. Các kiểu dữ liệu kể trên được gọi là các kiểu dữ liệu tĩnh.

Nhằm đáp ứng nhu cầu thể hiện sát thực tế, bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, cần phải tìm cách tổ chức kết hợp dữ liệu với những kích thước linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống. Các hình thức tổ chức dữ liệu như vậy được gọi là *cấu trúc dữ liệu động*.

Dạng đơn giản nhất của các cấu trúc dữ liệu động là danh sách liên kết. Đây là các cấu trúc dữ liệu tuyến tính. Điểm đặc trưng của cấu trúc dữ liệu này là khả năng truy xuất tuần tự. Tuy nhiên điều này làm cho việc truy vấn thông tin lưu trên cấu trúc dữ liệu này thường chậm; nhưng chúng cho phép sử dụng bộ nhớ hiệu quả hơn, thể hiện rõ qua các thao tác trên danh sách liên kết đơn, danh sách liên kết đôi và danh sách liên kết vòng sẽ được trình bày trong giáo trình này.

3.2. Định nghĩa danh sách

Cho T là một kiểu được định nghĩa trước, kiểu danh sách T_x gồm các phần tử thuộc kiểu T được định nghĩa là: $\mathbf{T_x = <V_x, O_x>}$

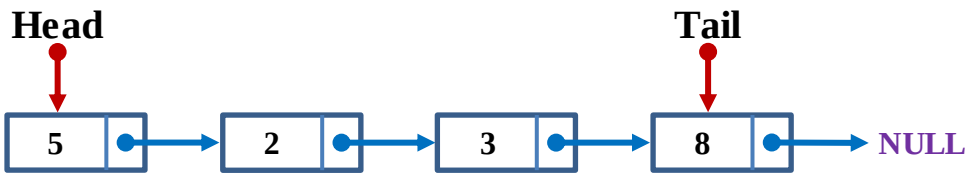
Trong đó:

- $\mathbf{V_x}$ = {Tập hợp có thứ tự các phần tử kiểu T được móc nối với nhau theo trình tự tuyến tính}.
- $\mathbf{O_x}$ = {Các phương thức thiết lập trên danh sách T_x (như: tạo danh sách; thêm phần tử, tìm phần tử trong danh sách; hủy phần tử; hủy danh sách, …)}

Lưu ý: Mỗi phần tử của danh sách ta có thể gọi ngắn gọn là nút.

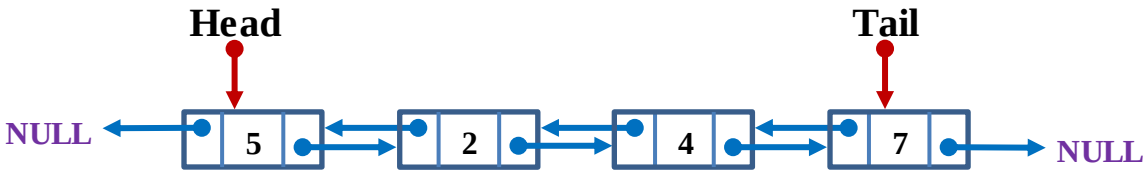
3.3. Các loại danh sách liên kết

- **Danh sách liên kết đơn:** Mỗi phần tử liên kết với một phần tử đứng liền kế sau nó trong danh sách.



Hình 3.1: Hình vẽ minh họa danh sách liên kết đơn

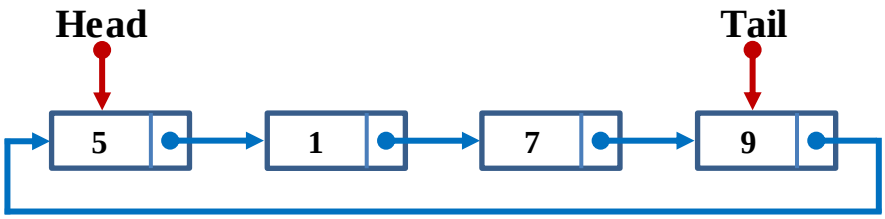
– **Danh sách liên kết đôi:** Mỗi phần tử liên kết với một phần tử đứng liền kế trước và một phần tử đứng liền kế sau nó trong danh sách.



Hình 3.2: Hình vẽ minh họa danh sách liên kết đôi

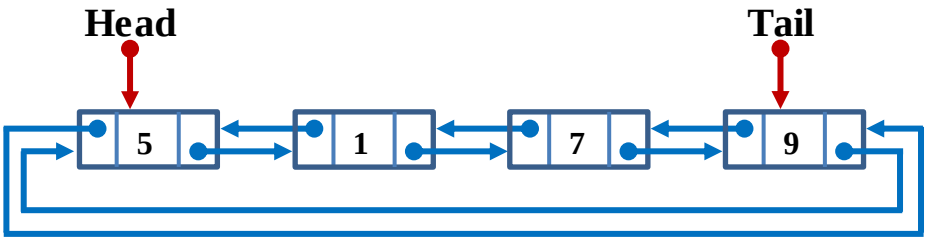
– **Danh sách liên vòng:** Phần tử cuối danh sách liên kết với phần tử đầu danh sách.

+ Danh sách liên kết vòng đơn.



Hình 3.3: Hình vẽ minh họa danh sách liên kết vòng đơn

+ Danh sách liên kết vòng đôi.



Hình 3.4: Hình vẽ minh họa danh sách liên kết vòng đôi

3.4. Danh sách liên kết đơn

3.4.1. Tổ chức danh sách

Mỗi phần tử liên kết với phần tử đứng liền kế sau nó trong danh sách.

Mỗi phần tử của danh sách đơn là một cấu trúc chứa hai thành phần chính:

– **Thành phần dữ liệu (Info):** Lưu trữ các thông tin về bản thân phần tử.

– **Thành phần liên kết (Link):** Lưu trữ địa chỉ của phần tử kế sau trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Để đơn giản và tiện theo dõi nên trong giáo trình này tác giả quy ước thành phần dữ liệu (Info) có kiểu **ItemType** là số nguyên.

➔Ta có cấu trúc dữ liệu của một nút trong danh sách liên kết đơn như sau:

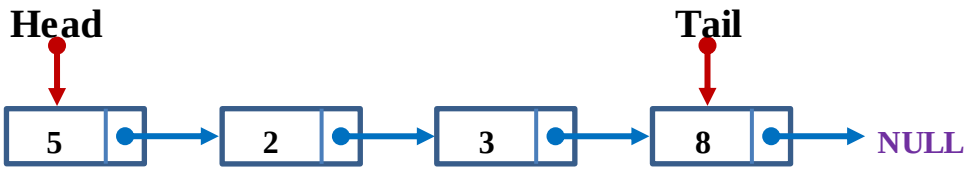
```
1.  typedef int ItemType; //ItemType cũng có thể là kiểu float, SINHVIEN, PHANSO,
                          ...
2.  struct SNode
3.  {
4.      ItemType Info;    //Lưu thông tin (dữ liệu) của bản thân.
5.      SNode* Next;      //Con trỏ trỏ đến SNode kế sau.
6.  };
```

Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đơn chính là sự liên kết các biến động này với nhau, nhờ vậy đạt được sự linh động khi thay đổi số lượng các phần tử.

Nếu biết được địa chỉ của phần tử đầu tiên trong danh sách đơn thì có thể dựa vào thông tin Next của nó để truy xuất đến phần tử thứ hai của danh sách, và lại dựa vào thông tin Next của phần tử thứ hai để truy xuất phần tử thứ ba,... nghĩa là để quản lý một danh sách đơn chỉ cần biết địa chỉ phần tử đầu của danh sách. Thường một con trỏ **Head** sẽ được dùng để lưu trữ địa chỉ phần tử đầu danh sách, nhưng thực tế có nhiều trường hợp cần làm việc với phần tử cuối cùng, khi đó mỗi lần muốn xác định phần tử cuối cùng lại phải duyệt từ đầu danh sách. Để tiện lợi, có thể sử dụng thêm một con trỏ **Tail** giữ địa chỉ phần tử cuối danh sách. Nên ta có cấu trúc của một danh sách liên kết đơn như sau:

```
1.  struct SList
2.  { //Kiểu danh sách liên kết đơn
3.      SNode* Head;   //Lưu địa chỉ của SNode đầu tiên trong SList
4.      SNode* Tail;   //Lưu địa chỉ của SNode cuối cùng trong SList
5.  };
```

Lúc này ta có danh sách liên kết đơn như sau:

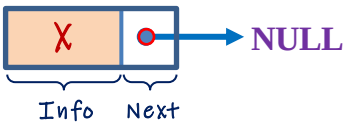


Hình 3.5: Hình vẽ minh họa cấu trúc danh sách liên kết đơn

3.4.2. Các thao tác cơ bản trên danh sách liên kết đơn

- Tạo một nút có trường Info bằng x.
- Khởi tạo một danh sách liên kết đơn rỗng.
- Kiểm tra danh sách liên kết đơn rỗng.
- Thêm một phần tử có khóa x vào danh sách.
- Tìm một phần tử có Info bằng x.
- Hủy một phần tử trong danh sách.
- Duyệt danh sách.
- Sắp xếp danh sách.

3.4.2.1. Tạo node



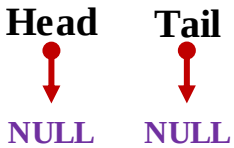
Hình 3.6: Hình vẽ minh họa tạo một nút mới cho danh sách liên kết đơn

Hàm trả về địa chỉ phần tử mới tạo như sau:

```
1.  SNode* createSNode(ItemType x)
2.  {
3.      SNode* p = new SNode;
4.      if(p == NULL)
5.      {
6.          printf("Không đủ bộ nhớ để cấp phát nút mới!");
7.          getch();
8.          return NULL;
9.      }
10.     p→Info = x;
11.     p→Next = NULL;
12.     return p;
13. }
```

3.4.2.2. Khởi tạo danh sách rỗng

Do danh sách chưa có phần tử nào, nên con trỏ đầu Head (*lưu địa chỉ của nút đầu tiên*) và con trỏ cuối Tail (*lưu địa chỉ của nút cuối cùng*) đều bằng NULL.




```
1.  void initSList(SList &sl)
```

```
2.  {
3.          sl.Head = NULL;
4.          sl.Tail = NULL;
5.  }
```

3.4.2.3. Kiểm tra danh sách liên kết đơn rỗng

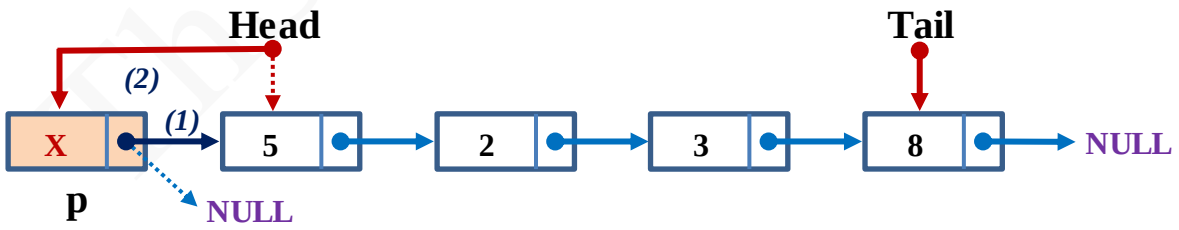
Kiểm tra danh sách liên kết đơn đã có phần tử nào hay chưa? Nghĩa là kiểm tra con trỏ đầu Head (*hoặc con trỏ cuối Tail*) có bằng NULL hay không? Hàm trả sẽ về 1 nếu danh sách liên kết đơn chưa có phần tử nào (*rỗng*), ngược lại (*không rỗng*) thì hàm sẽ trả về 0.

```
1.  int isEmpty(SList sl)
2.  {
3.          if(sl.Head == NULL)
4.                  return 1; //Nếu danh sách rỗng
5.          else
6.                  return 0; //Nếu danh sách không rỗng
7.  }
```

3.4.2.4. Thêm một phần tử vào danh sách

- Nguyên tắc thêm: Khi thêm một phần tử vào SList thì có làm cho Head, Tail thay đổi?
- Các vị trí cần thêm một phần tử vào SList:
  - + Thêm vào đầu SList.
  - + Thêm vào cuối SList.
  - + Thêm vào sau một phần tử q trong SList.

3.4.2.4.1. Thêm một phần tử vào đầu danh sách



Hình 3.7: Hình vẽ minh họa thêm một nút vào đầu danh sách liên kết đơn

Giải thuật:

- **Bước 1:** Nếu phần tử muốn thêm p không tồn tại thì không thực hiện.
- **Bước 2:** Nếu SList rỗng thì
  - + Head = p;

+ Tail = Head;

- **Bước 3:** Ngược lại

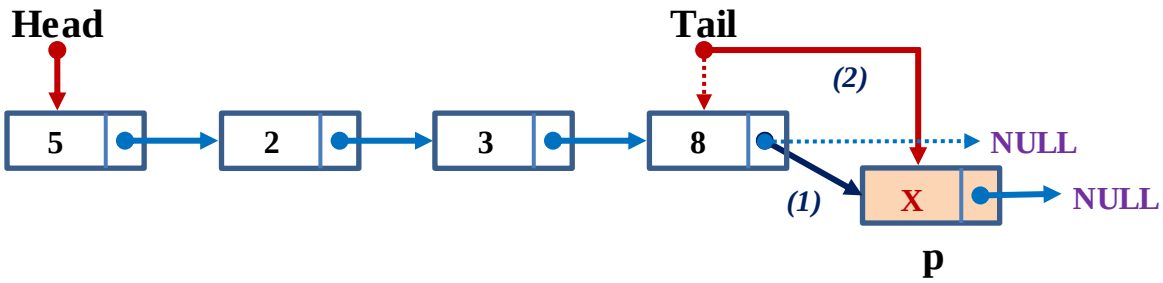
+ p→Next = Head;

+ Head = p;

Cài đặt:

```
1.  int insertHead(SList &sl, SNode* p)
2.  {
3.      if(p == NULL)
4.          return 0; //Thực hiện không thành công
5.      if(isEmpty(sl) == 1)
6.      {
7.          sl.Head = p;
8.          sl.Tail = p;
9.      }
10.     else
11.     {
12.         p→Next = sl.Head;
13.         sl.Head = p;
14.     }
15.     return 1; //Thực hiện thành công
16. }
```

3.4.2.4.2. Thêm một phần tử vào cuối danh sách



Hình 3.8: Hình vẽ minh họa thêm một nút vào cuối danh sách liên kết đơn

Giải thuật:

- **Bước 1:** Nếu phần tử muốn thêm p không tồn tại thì không thực hiện.

- **Bước 2:** Nếu SList rỗng thì:

+ Head = p;

+ Tail = p;

- **Bước 3:** Ngược lại

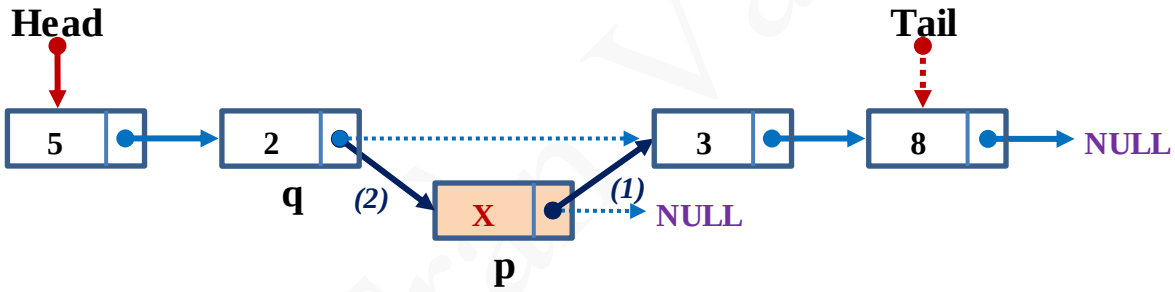
+ Tail→Next = p;

+ Tail = p;

Cài đặt:

```
1.  int insertTail(SList &sl, SNode* p)
2.  {
3.      if(p == NULL)
4.          return 0; //Thực hiện không thành công
5.      if(isEmpty(sl) == 1)
6.      {
7.          sl.Head = p;
8.          sl.Tail = p;
9.      }
10.     else
11.     {
12.         sl.Tail→Next = p;
13.         sl.Tail = p;
14.     }
15.     return 1; //Thực hiện thành công
16. }
```

3.4.2.4.3. Thêm một phần tử vào sau phần tử q



Hình 3.9: Hình vẽ minh họa thêm vào sau nút q của danh sách liên kết đơn

Giải thuật:

- **Bước 1:** Nếu phần tử q hoặc phần tử muốn thêm p không tồn tại thì không thực hiện.
- **Bước 2:**
  + p→Next = q→Next;
  + q→Next = p;
  + Nếu q là phần tử cuối thì: Tail = p;

Cài đặt:

```
1.  int insertAfter(SList &sl, SNode* q, SNode* p)
2.  {
3.      if(q == NULL || p == NULL)
4.          return 0; //Thực hiện không thành công
```

```
5.      p→Next = q→Next;
6.      q→Next = p;
7.      if(sl.Tail == q)
8.          sl.Tail = p;
9.      return 1; //Thực hiện thành công
10. }
```

3.4.2.5. Tìm kiếm phần tử trong danh sách

Danh sách liên kết đơn đòi hỏi truy xuất tuần tự, do đó áp dụng giải thuật tìm tuyến tính để xác định phần tử trong danh sách có khóa bằng với x. Sử dụng một con trỏ phụ trợ p để lần lượt trỏ đến các phần tử trong danh sách. Nếu tìm thấy có 1 nút có khóa bằng với x thì trả về con trỏ chứa nút đó, ngược lại nếu duyệt hết danh sách mà vẫn không tìm thấy nút nào có khóa bằng với x thì trả về NULL.

Các bước của giải thuật tìm nút có Info bằng x trong list đơn:

- **Bước 1:** Nếu danh sách rỗng thì: trả về NULL;
- **Bước 2:** Gán p = Head; *//địa chỉ của phần tử đầu tiên trong DSLK đơn*
- **Bước 3:**

  Lặp lại trong khi (p != NULL và p→Info != x) thì thực hiện:

  p = p→Next; *//xét phần tử kế sau*
- **Bước 4:** Trả về p; *//nếu (p != NULL) thì p lưu địa chỉ của phần tử có khóa bằng x, hoặc NULL là không có phần tử cần tìm.*

Cài đặt:

```
1.  SNode* findSNode(SList sl, ItemType x)
2.  {
3.      if(isEmpty(sl) == 1)
4.          return NULL;
5.      SNode* p = sl.Head;
6.      while( (p != NULL) && (p→Info != x) )
7.          p = p→Next;
8.      return p;  //có thể NULL: không tìm thấy, hoặc khác NULL: tìm thấy
9.  }
```

Có thể viết bằng lệnh for như sau:

```
1.  SNode* findSNode(SList sl, ItemType x)
2.  {
3.      for(SNode* p = sl.Head; p != NULL; p = p→Next)
4.          if(p→Info == x)
```

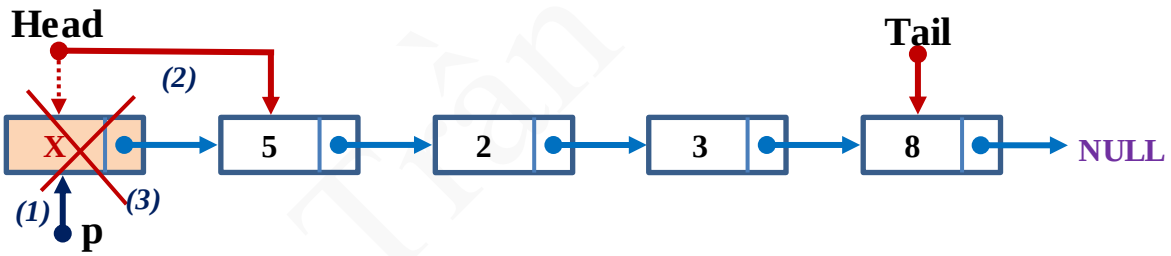
```
5.                  return p; //Trả về p khác NULL: tìm thấy
6.          return NULL; //Trả về NULL: không tìm thấy
7.  }
```

3.4.2.6. Xóa phần tử của danh sách

- Nguyên tắc: Phải cô lập phần tử cần xóa trước khi xóa (hủy và thu hồi vùng nhớ đã được cấp phát cho nó trước đó).
- Các vị trí cần xóa:
  + Xóa phần tử đứng đầu SList.
  + Xóa phần tử đứng cuối SList.
  + Xóa phần tử đứng sau q trong danh sách SList.
  + Xóa phần tử có khóa bằng x của SList.

Ở phần trên, các phần tử trong danh sách liên kết đơn được cấp phát vùng nhớ động bằng hàm **new**, thì sẽ được giải phóng vùng nhớ bằng hàm **delete**.

3.4.2.6.1. Xóa phần tử đầu danh sách



Hình 3.10: Hình vẽ minh họa xóa một nút đầu của danh sách liên kết đơn

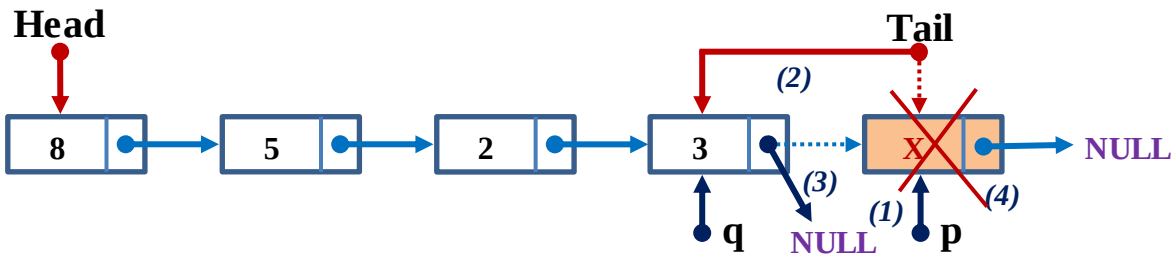
Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Khi (Head != NULL) thì: thực hiện các công việc sau:
  + **Bước 2.1:** p = Head;
  + **Bước 2.2:** Head = Head→Next;
  + **Bước 2.3:** Nếu (Head == NULL) thì: Tail = NULL;
  + **Bước 2.4:** Lưu lại thông tin nút bị xóa;
  + **Bước 2.5:** delete(p); *//Hủy nút do p trỏ đến (con trỏ p)*

Cài đặt:

```
1.  int deleteHead(SList &sl, ItemType &x)
2.  {
3.      if(isEmpty(sl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      SNode* p = sl.Head;
6.      sl.Head = sl.Head→Next;
7.      if(sl.Head == NULL)
8.          sl.Tail = NULL;
9.      x = p→Info;      //Lấy Thông tin của nút bị hủy
10.     delete p;  //Hủy nút do p trỏ đến
11.     return 1;  //Thực hiện thành công
12. }
```

3.4.2.6.2. Xóa phần tử cuối danh sách



Hình 3.11: Hình vẽ minh họa xóa một nút cuối của danh sách liên kết đơn

Giải thuật:

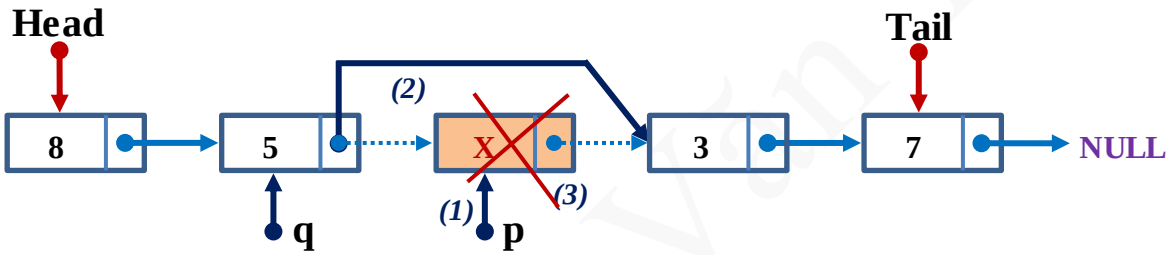
- **Bước 1:** Nếu (Tail == NULL) thì: không thực hiện gì cả.
- **Bước 2:** Khi (Tail != NULL) thì: thực hiện các công việc sau:
  - + **Bước 2.1:** Gán q = Head; và p = Tail;
  - + **Bước 2.2:** Nếu (p == q) thì: Head = Tail = NULL;
  - + **Bước 2.3:** Ngược lại: Tìm đến nút kế trước nút Tail.
  - + **Bước 2.4:** Tail = q;
  - + **Bước 2.5:** Tail→Next = NULL;
  - + **Bước 2.6:** Lưu lại thông tin nút bị xóa;
  - + **Bước 2.7:** delete(p); *//Hủy nút do p trỏ đến*

Cài đặt:

```
1.  int deleteTail(SList &sl, ItemType &x)
2.  {
3.      if(isEmpty(sl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      SNode* p = sl.Tail;
```

```
6.      SNode* q = sl.Head;
7.      if(p == q)
8.          sl.Head = sl.Tail = NULL;
9.      else
10.     {
11.         while(q→Next != p)    //Tìm nút kế trước nút Tail
12.             q = q→Next;
13.         sl.Tail = q;
14.         sl.Tail→Next = NULL;
15.     }
16.     x = p→Info;     //Lấy thông tin của nút bị hủy
17.     delete p;   //Hủy nút do p trỏ đến
18.     return 1;  //Thực hiện thành công
19. }
```

3.4.2.6.3. Xóa phần tử sau phần tử q của danh sách



Hình 3.12: Hình vẽ minh họa xóa sau nút q của danh sách liên kết đơn

Giải thuật:

- **Bước 1:** Nếu (q == NULL hoặc q→Next == NULL) thì: không thực hiện.
- **Bước 2:** Khi (q != NULL và q→Next != NULL) thì: thực hiện các công việc sau:
  - + **Bước 2.1:** p = q→Next;
  - + **Bước 2.2:** q→Next = p→Next;
  - + **Bước 2.3:** Nếu (p == Tail) thì: Tail = q;
  - + **Bước 2.4:** Lưu lại thông tin nút bị xóa;
  - + **Bước 2.5:** delete(p); *//Hủy nút do p trỏ đến*

Cài đặt:

```
1.  int deleteAfter(SList &sl, SNode* q, ItemType &x)
2.  {
3.      if(q == NULL || q→Next == NULL)
4.          return 0;  //Thực hiện không thành công
5.      SNode* p = q→Next;
```

```
6.      q→Next = p→Next;
7.      if(sl.Tail == p)
8.          sl.Tail = q;
9.      x = p→Info;     //Lấy thông tin của nút bị hủy
10.     delete p;  //Hủy nút do p trỏ đến
11.     return 1;  //Thực hiện thành công
12. }
```

3.4.2.6.4. Xóa một phần tử có khóa x

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Tìm phần tử p có khóa bằng x, và q là phần tử đứng kế trước p.
- **Bước 3:** Nếu (p == NULL) thì: Không có nút chứa x nên không thực hiện.
- **Bước 4:** *//Ngược lại (p != NULL) → nghĩa là tồn tại nút p chứa khóa x*

  + Nếu (p == Head) thì: *//p là nút đầu danh sách → xóa đầu*

    deleteHead(**sl**, x);

  + Ngược lại:

    deleteAfter(**sl**, q, x); *//Xóa nút p chứa x kế sau nút q*

Cài đặt:

```
1.  int deleteSNodeX(SList &sl, ItemType x)
2.  {
3.      if(isEmpty(sl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      SNode* p = sl.Head;
6.      SNode* q = NULL;    //sẽ trỏ đến nút kế trước p
7.      while( (p != NULL) && (p→Info != x) )
8.      {//vòng lặp tìm nút p chứa x, q là nút kế trước p
9.          q = p;
10.         p = p→Next;
11.     }
12.     if(p == NULL)   //không tìm thấy phần tử có khóa bằng x
13.         return 0;  //Thực hiện không thành công
14.     if(p == sl.Head)//p có khóa bằng x là nút đầu danh sách
15.         deleteHead(sl, x);
16.     else  //xóa nút p có khóa x nằm kế sau nút q
17.         deleteAfter(sl, q, x);
18.     return 1;  //Thực hiện thành công
19. }
```

3.4.2.7. Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách. Như là:

- Đếm các phần tử của danh sách.
- Tìm tất cả các phần tử thỏa mãn điều kiện nào đó.
- Hủy toàn bộ danh sách (và giải phóng bộ nhớ).

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Thông báo danh sách rỗng và không thực hiện.
- **Bước 2:** Gán p = Head; *//p lưu địa chỉ của phần tử đầu trong SList*
- **Bước 3:**

  Lặp lại trong khi (danh sách chưa hết) thì thực hiện:

  + Xử lý phần tử p.

  + p = p→Next; *//Xét phần tử kế*

Cài đặt (*tổng quát*):

```
1.  void processSList(SList sl)
2.  {
3.      if(isEmpty(sl) == 1)
4.      {
5.          printf("\nDanh sach rong!");
6.          return;
7.      }
8.      SNode* p = sl.Head;
9.      while(p != NULL)
10.     {
11.         <Xử lý nút p tuỳ trường hợp cụ thể>
12.         p = p→Next;
13.     }
14. }
```

Ví dụ: Hàm in nội dung danh sách ra màn hình sẽ được viết cụ thể như sau:

```
1.  void showSList(SList sl)
2.  {
3.      if(isEmpty(sl) == 1)
4.      {
5.          printf("\nDanh sach rong!");
6.          return;
7.      }
```

```
8.      printf("\nNoi dung danh sach la: ");
9.      SNode* p = sl.Head;
10.     while(p != NULL)
11.     {
12.         printf("%4d", p→Info); //xuất nội dung của nút p
13.         p = p→Next;
14.     }
15. }
```

3.4.2.8. Hủy toàn bộ danh sách

Hủy tất cả các phần tử của danh sách đồng nghĩa với việc thu hồi lại những vùng nhớ đã cấp phát cho các nút của danh sách. Duyệt danh sách một cách tuần tự từ đầu đến cuối để hủy lần lượt từng nút một.

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Lặp lại trong khi (danh sách còn phần tử) thì thực hiện lần lượt những việc sau:
  - + **Bước 2.1:** Gán p = Head;
  - + **Bước 2.2:** Gán Head = Head→Next; *//Cập nhật con trỏ Head*
  - + **Bước 2.3:** Hủy con trỏ p;
- **Bước 3:** Gán Tail = NULL; *//bảo toàn tính nhất quán khi danh sách rỗng*

Cài đặt:

```
1.  int deleteSList(SList &sl)
2.  {
3.      if(isEmpty(sl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      while(sl.Head != NULL)
6.      {
7.          SNode* p = sl.Head;
8.          sl.Head = sl.Head→Next;
9.          delete p; //Hoặc gọi hàm đã viết deleteSNode(p);
10.     }
11.     sl.Tail = NULL;
12.     return 1;  //Thực hiện thành công
13. }
```

3.4.2.9. Sắp xếp danh sách

Một danh sách có thứ tự (danh sách được sắp xếp) là một danh sách mà các phần tử của nó được sắp xếp theo một thứ tự nào đó dựa trên một trường khóa. Để sắp xếp một danh sách, ta có thể thực hiện một trong hai phương án sau:

3.4.2.9.1. **Phương án 1:** Hoán vị nội dung các phần tử trong danh sách (*vùng Info*)

Với phương án này, có thể chọn một trong những giải thuật sắp xếp đã biết để cài đặt lại trên danh sách như thực hiện trên mảng, điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên danh sách thông qua liên kết thay vì chỉ số như trên mảng. Do dựa trên việc hoán vị nội dung của các phần tử, phương án này đòi hỏi sử dụng thêm vùng nhớ trung gian nên chỉ thích hợp với các danh sách mà thành phần Info của các phần tử có kích thước nhỏ. Hơn nữa, số lần hoán vị có thể lên đến bậc n^2 với danh sách có n phần tử, không tận dụng được các ưu điểm của danh sách.

Ví dụ: Cài đặt giải thuật sắp xếp **Chọn lựa trực tiếp** trên danh sách:

```
1.  void selectionSort_SList_Ascending (SList &sl)
2.  {
3.      if(isEmpty(sl) == 1 || sl.Head == sl.Tail) return;
4.      SNode* min;
5.      for(SNode* p = sl.Head; p != sl.Tail; p = p→Next)
6.      {
7.          min = p;
8.          for(SNode* q = p→Next; q != NULL; q = q→Next)
9.              if(min→Info > q→Info)
10.                 min = q;
11.         if(min != p)
12.             swap(min→Info, p→Info); //Hoán vị
13.     }
14. }
```

3.4.2.9.2. **Phương án 2:** Thay đổi các mối liên kết (thao tác trên vùng Next)

Tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ (đồng thời hủy danh sách cũ). Giả sử danh sách mới sẽ được quản lý bằng danh sách Result, ta có giải thuật như sau:

- **Bước 1:** Khởi tạo danh sách mới *Result* là rỗng
- **Bước 2:** Tìm trong danh sách cũ *SListOld* thì phần tử min là phần tử nhỏ nhất
- **Bước 3:** Tách min khỏi danh sách *SListOld*
- **Bước 4:** Chèn min vào cuối danh sách *Result*
- **Bước 5:** Quay lại Bước 2 khi chưa hết danh sách *SListOld*

Cài đặt: (*Sinh viên tự thực hiện xem như 1 bài tập*)

➔**Ưu, nhược điểm của hai phương án trên là:**

– **Thay đổi thành phần Info (dữ liệu):**

+ ***Ưu điểm:*** Cài đặt đơn giản, tương tự như sắp xếp mảng.

+ ***Nhược điểm:***

- Đòi hỏi thêm vùng nhớ khi hoán vị nội dung của hai phần tử → chỉ phù hợp với những danh sách có kích thước Info nhỏ.
- Khi kích thước Info (dữ liệu) lớn chi phí cho việc hoán vị thành phần Info lớn.
- Làm cho thao tác sắp xếp chậm.

– **Thay đổi thành phần Next:**

+ ***Ưu điểm:***

- Kích thước của trường này không thay đổi, do đó không phụ thuộc vào kích thước bản chất dữ liệu lưu tại mỗi nút.
- Thao tác sắp xếp nhanh.

+ ***Nhược điểm:*** Cài đặt phức tạp.

3.5. Danh sách liên kết đôi

3.5.1. Tổ chức danh sách

Mỗi phần tử liên kết với một phần tử đứng kế sau và một phần tử đứng kế trước trong danh sách.

Mỗi phần tử của danh sách đôi là một cấu trúc chứa hai thành phần chính:

– **Thành phần dữ liệu (Info):** Lưu trữ các thông tin về bản thân phần tử.

– **Thành phần liên kết (Link):** Lưu trữ địa chỉ của phần tử kế trước và địa chỉ của phần tử kế sau trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử đầu hay phần tử cuối.

Để đơn giản và tiện theo dõi nên trong giáo trình này tác giả quy ước thành phần dữ liệu (Info) có kiểu **ItemType** là số nguyên.

➔Ta có cấu trúc dữ liệu của một nút trong danh sách liên kết đôi như sau:

```
1.  typedef int ItemType; //ItemType cũng có thể là kiểu float, SINHVIEN, PHANSO
2.  struct DNode
3.  {
4.      ItemType Info;      //Lưu thông tin (dữ liệu) của bản thân.
```

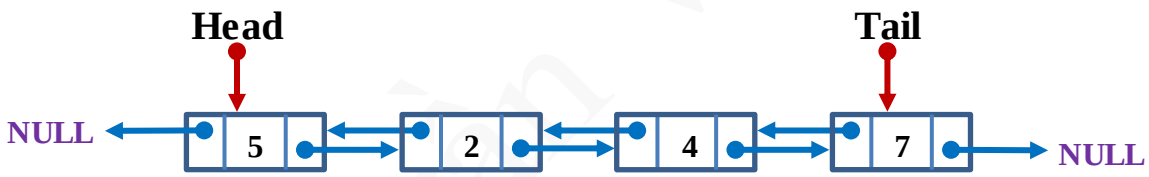
```
5.      DNode* Prev;      //Con trỏ trỏ đến DNode kế trước.
6.      DNode* Next;      //Con trỏ trỏ đến DNode kế sau.
7.  };
```

Một phần tử trong danh sách đôi là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đôi chính là sự liên kết các biến động này với nhau, nhờ vậy đạt được sự linh động khi thay đổi số lượng các phần tử.

Nếu biết được địa chỉ của phần tử đầu tiên hay địa chỉ của phần tử cuối cùng trong danh sách đôi thì có thể dựa vào thông tin Next hay Prev của nó để truy xuất đến các phần tử còn lại. Nghĩa là để quản lý một danh sách đôi chỉ cần biết địa chỉ phần tử đầu hay địa chỉ phần tử cuối danh sách. Một con trỏ Head sẽ được dùng để lưu trữ địa chỉ phần tử đầu danh sách, một con trỏ Tail sẽ được dùng để lưu trữ địa chỉ phần tử cuối danh sách. Vì vậy cấu trúc của một danh sách liên kết đôi được khai báo như sau:

```
1.  struct DList
2.  {//Kiểu danh sách liên kết đôi
3.      DNode* Head;  //Lưu địa chỉ Node đầu tiên trong DList
4.      DNode* Tail;  //Lưu địa chỉ của Node cuối cùng trong DList
5.  };
```

Lúc này ta có danh sách liên kết đôi như sau:

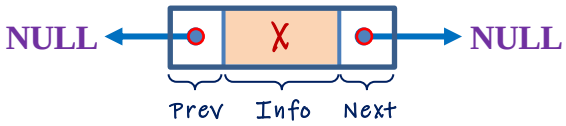


Hình 3.29: Hình vẽ minh họa cấu trúc danh sách liên kết đôi

3.5.2. Các thao tác cơ bản trên danh sách liên kết đôi

- Tạo một nút có trường Info bằng x.
- Khởi tạo một danh sách liên kết đôi rỗng.
- Kiểm tra danh sách liên kết đôi rỗng.
- Thêm một phần tử có khóa x vào danh sách.
- Tìm một phần tử có Info bằng x.
- Hủy một phần tử trong danh sách.
- Duyệt danh sách.
- Sắp xếp danh sách.

3.5.2.1. Tạo một node mới



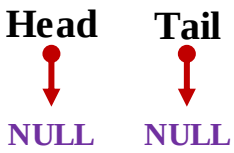
Hình 3.30: Hình vẽ minh họa tạo một nút mới cho danh sách liên kết đôi

Hàm trả về địa chỉ phần tử mới tạo như sau:

```
1.  DNode* createDNode(ItemType x)
2.  {
3.      DNode* p = new DNode;
4.      if(p == NULL)
5.      {
6.          printf("Không đủ bộ nhớ để cấp phát nút mới!");
7.          getch();
8.          return NULL;
9.      }
10.     p→Info = x;
11.     p→Prev = NULL;
12.     p→Next = NULL;
13.     return p;
14. }
```

3.5.2.2. Khởi tạo danh sách rỗng

Địa chỉ của nút đầu tiên, địa chỉ của nút cuối cùng đều không có.




```
1.  void initDList(DList &dl)
2.  {
3.      dl.Head = NULL;
4.      dl.Tail = NULL;
5.  }
```

3.5.2.3. Kiểm tra danh sách liên kết đôi rỗng

Kiểm tra danh sách liên kết đôi đã có phần tử nào hay chưa? Nghĩa là kiểm tra con trỏ đầu Head (*hoặc con trỏ cuối Tail*) có bằng NULL hay không? Hàm trả sẽ về 1 nếu danh sách liên kết đôi chưa có phần tử nào (*rỗng*), ngược lại (*không rỗng*) thì hàm sẽ trả về 0.

```
1.  int isEmpty(DList dl)
2.  {
3.      if(dl.Head == NULL)
4.          return 1; //Nếu danh sách rỗng
```

```
5.      else
6.          return 0; //Nếu danh sách không rỗng
7.  }
```

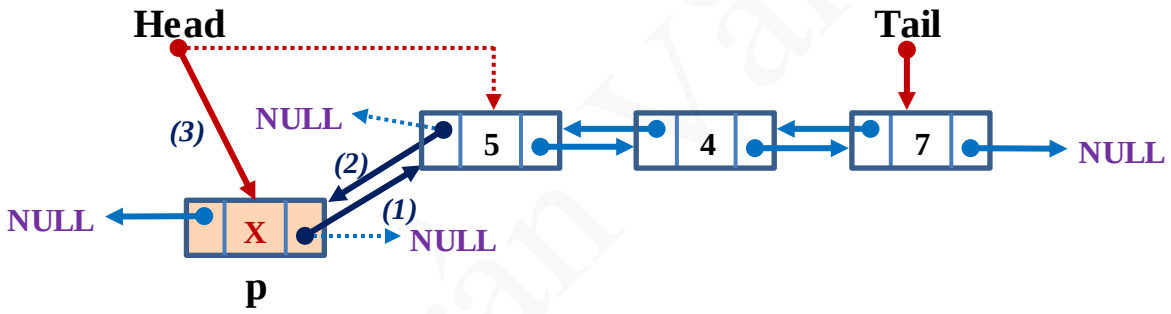
3.5.2.4. Thêm một phần tử vào danh sách

– Nguyên tắc thêm: Khi thêm một phần tử vào DList thì có làm cho Head, Tail thay đổi?

– Các vị trí cần thêm một phần tử vào DList:

+ Thêm vào đầu DList.
+ Thêm vào cuối DList.
+ Thêm vào sau một phần tử q trong DList.
+ Thêm vào trước một phần tử q trong DList.

3.5.2.4.1. Thêm một phần tử p chứa giá trị x vào đầu danh sách



Hình 3.31: Hình vẽ minh họa thêm một nút vào đầu danh sách liên kết đôi

Giải thuật:

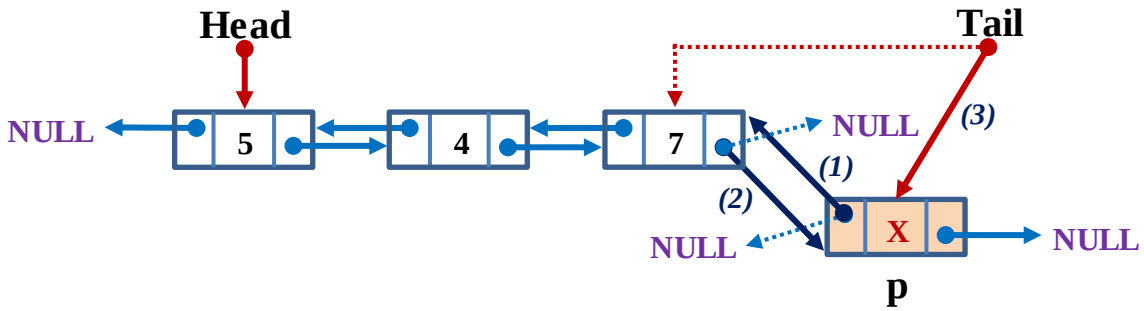
- **Bước 1:** Nếu phần tử muốn thêm p không tồn tại thì: không thực hiện.
- **Bước 2:** Nếu DList rỗng thì:
  + Head = p;
  + Tail = p;
- **Bước 3:** Ngược lại:
  + p→Next = Head;
  + Head→Prev = p;
  + Head = p;

Cài đặt:

```
1.  int insertHead(DList &dl, DNode* p)
2.  {
```

```
3.      if(p == NULL)
4.          return 0; //Thực hiện không thành công
5.      if(isEmpty(dl) == 1)
6.      {
7.          dl.Head = p;
8.          dl.Tail = p;
9.      }
10.     else
11.     {
12.         p→Next = dl.Head;
13.         dl.Head→Prev = p;
14.         dl.Head = p;
15.     }
16.     return 1; //Thực hiện thành công
17. }
```

3.5.2.4.2. Thêm một phần tử p chứa giá trị x vào cuối danh sách



Hình 3.32: Hình vẽ minh họa thêm một nút vào cuối danh sách liên kết đôi

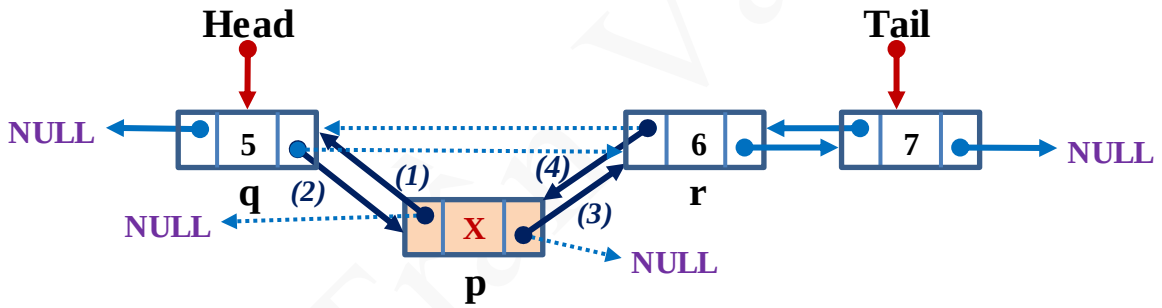
Giải thuật:

- **Bước 1:** Nếu phần tử muốn thêm p không tồn tại thì: không thực hiện.
- **Bước 2:** Nếu DList rỗng thì:
  - + Head = p;
  - + Tail = p;
- **Bước 3:** Ngược lại:
  - + p→Prev = Tail;
  - + Tail→Next = p;
  - + Tail = p;

Cài đặt:

```
1.  int insertTail(DList &dl, DNode* p)
2.  {
3.      if(p == NULL)
4.          return 0; //Thực hiện không thành công
5.      if(isEmpty(dl) == 1)
6.      {
7.          dl.Head = p;
8.          dl.Tail = p;
9.      }
10.     else
11.     {
12.         dl.Tail→Next = p;
13.         p→Prev = dl.Tail;
14.         dl.Tail = p;
15.     }
16.     return 1; //Thực hiện thành công
17. }
```

3.5.2.4.3. Thêm một phần tử p có giá trị x vào sau phần tử q có giá trị y



Hình 3.33: Hình vẽ minh họa thêm vào sau nút q của danh sách liên kết đôi

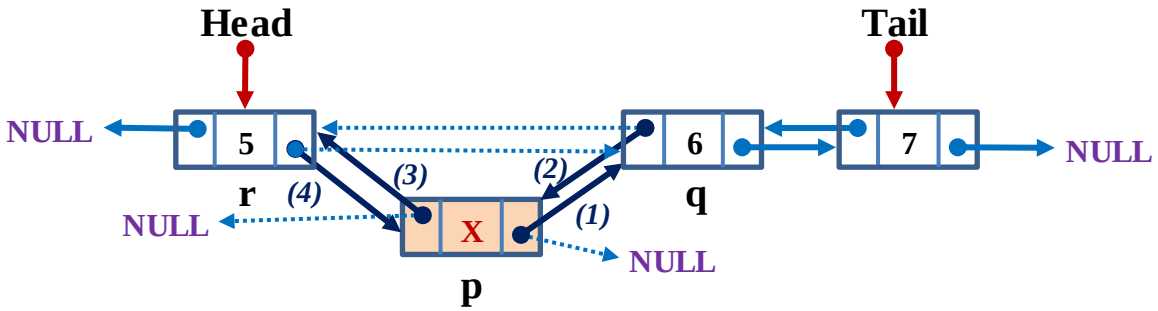
Giải thuật:

- **Bước 1:** Nếu phần tử q hoặc phần tử muốn thêm p không tồn tại thì: không thực hiện.
- **Bước 2:**
  - + r = q→Next;
  - + p→Prev = q;
  - + q→Next = p;
  - + p→Next = r;
- **Bước 3:**
  - + Nếu (r != NULL) thì: r→Prev = p;
  - + Ngược lại (q là phần tử cuối) thì: Tail = p;

Cài đặt:

```
1.  int insertAfter(DList &dl, DNode* q, DNode* p)
2.  {//Chèn nút p vào sau nút q
3.      if(q == NULL || p == NULL)
4.          return 0;  //Thực hiện không thành công
5.      DNode* r = q→Next;
6.      p→Next = r;
7.      p→Prev = q;
8.      q→Next = p;
9.      if(r != NULL)
10.         r→Prev = p;
11.     else  //if(q == dl.Tail)
12.         dl.Tail = p;
13.     return 1;  //Thực hiện thành công
14. }
```

3.5.2.4.4. Thêm một phần tử p có giá trị x vào trước phần tử q có giá trị y



Hình 3.34: Hình vẽ minh họa thêm vào trước nút q của danh sách liên kết đôi

Giải thuật:

- **Bước 1:** Nếu phần tử q hoặc phần tử muốn thêm p không tồn tại thì: không thực hiện.
- **Bước 2:**
  - + r = q→Prev;
  - + p→Next = q;
  - + q→Prev = p;
  - + p→Prev = r;
- **Bước 3:**
  - + Nếu r != NULL thì: r→Next = p;
  - + Ngược lại (q là phần tử đầu) thì: Head = p;

Cài đặt:

```
1.  int insertBefore(DList &dl, DNode* q, DNode* p)
2.  {//Chèn nút p vào trước nút q
3.      if(q == NULL || p == NULL)
4.          return 0;   //Thực hiện không thành công
5.      DNode* r = q→Prev;
6.      p→Next = q;
7.      p→Prev = r;
8.      q→Prev = p;
9.      if(r != NULL)
10.         r→Next = p;
11.     else  //if(q == dl.Head)
12.         dl.Head = p;
13.     return 1;  //Thực hiện thành công
14. }
```

3.5.2.5. Tìm kiếm phần tử trong danh sách

Danh sách liên kết đôi đòi hỏi truy xuất tuần tự, do đó áp dụng giải thuật tìm tuyến tính để xác định phần tử trong danh sách có khóa bằng với x. Sử dụng một con trỏ phụ trợ p để lần lượt trỏ đến các phần tử trong danh sách. Nếu tìm thấy có một nút có khóa bằng với x thì trả về con trỏ chứa nút đó, ngược lại nếu duyệt hết danh sách mà vẫn không tìm thấy nút nào có khóa bằng với x thì trả về NULL.

Các bước của giải thuật tìm nút có Info bằng x trong danh sách liên kết đôi (*duyệt xuôi*):

- **Bước 1:** Nếu danh sách rỗng thì: trả về NULL;
- **Bước 2:** Gán p = Head; *//địa chỉ của phần tử đầu trong DSLK đôi*
- **Bước 3:**

  Lặp lại trong khi (p != NULL và p→Info != x) thì thực hiện:

  p = p→Next; *//xét phần tử kế sau*
- **Bước 4:** Trả về p; *//nếu (p != NULL) thì p lưu địa chỉ của phần tử có khóa bằng x, hoặc NULL là không có phần tử cần tìm.*

Cài đặt:

```
1.  DNode* findDNode(DList dl, ItemType x)
2.  {
3.      if(isEmpty(dl) == 1)
4.          return NULL; //Không tìm thấy
5.      DNode* p = dl.Head;
6.      while( (p != NULL) && (p→Info != x) )
```

```
7.              p = p→Next;
8.          return p; //Không tìm thấy: NULL, ngược lại: khác NULL
9.  }
```

3.5.2.6. Xóa phần tử của danh sách

– Nguyên tắc: Phải cô lập phần tử cần xóa trước khi hủy.

– Các vị trí cần xóa:

+ Xóa phần tử đứng đầu DList.

+ Xóa phần tử đứng cuối DList.

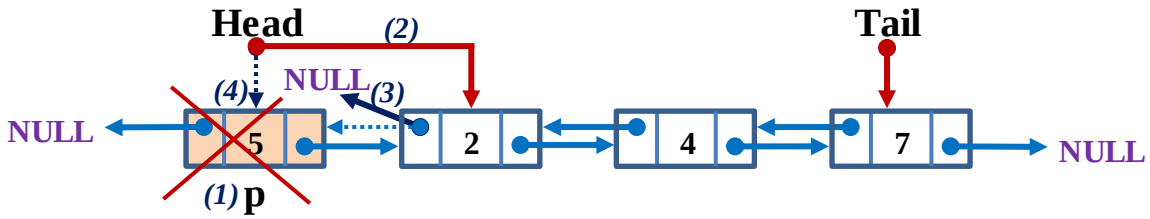
+ Xóa phần tử đứng sau q trong danh sách DList.

+ Xóa phần tử đứng trước q trong danh sách DList.

+ Xóa phần tử có khóa bằng x.

Ở phần trên, các phần tử trong danh sách liên kết đôi được cấp phát vùng nhớ động bằng hàm **new**, thì sẽ được giải phóng vùng nhớ bằng hàm **delete**.

3.5.2.6.1. Xóa phần tử đầu danh sách



Hình 3.35: Hình vẽ minh họa xóa một nút ở đầu danh sách liên kết đôi

Giải thuật:

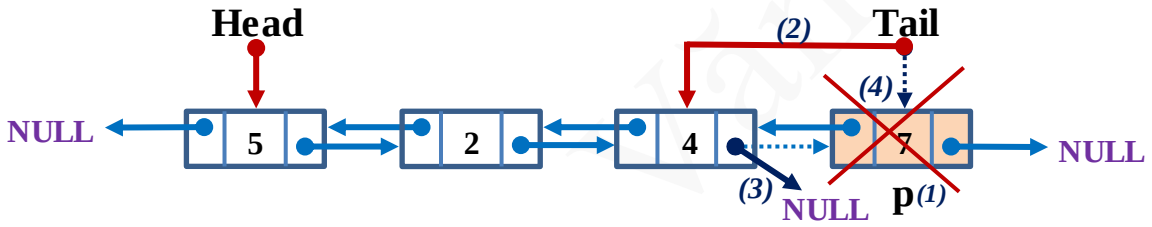
- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Thực hiện việc xóa nút đầu danh sách như sau:
  + **Bước 2.1:** Gán p = Head;
  + **Bước 2.2:** Head = Head→Next;
  + **Bước 2.3:** Kiểm tra nếu (Head == NULL) thì: Tail = NULL;

    Ngược lại: Head→Prev = NULL;
  + **Bước 2.4:** Lưu giữ lại giá trị nút bị xóa
  + **Bước 2.5:** delete(p); *//Hủy nút do p trỏ đến*

Cài đặt:

```
1.  int deleteHead(DList &dl, ItemType &x)
2.  {
3.      if(isEmpty(dl) == 1)
4.          return 0;   //Thực hiện không thành công
5.      DNode* p = dl.Head;
6.      dl.Head = dl.Head→Next;
7.      if(dl.Head == NULL)
8.          dl.Tail = NULL;
9.      else
10.         dl.Head→Prev = NULL;
11.     x = p→Info;     //Lấy thông tin của nút bị hủy
12.     delete p;   //Hủy nút do p trỏ đến
13.     return 1;   //Thực hiện thành công
14. }
```

3.5.2.6.2. Xóa phần tử cuối danh sách



Hình 3.36: Hình vẽ minh họa xóa ở cuối danh sách liên kết đôi

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Thực hiện việc xóa nút cuối danh sách như sau:
  - + **Bước 2.1:** Gán p = Tail;
  - + **Bước 2.2:** Gán Tail = Tail→Prev;
  - + **Bước 2.3:** Kiểm tra nếu (Tail == NULL) thì: Head = NULL;

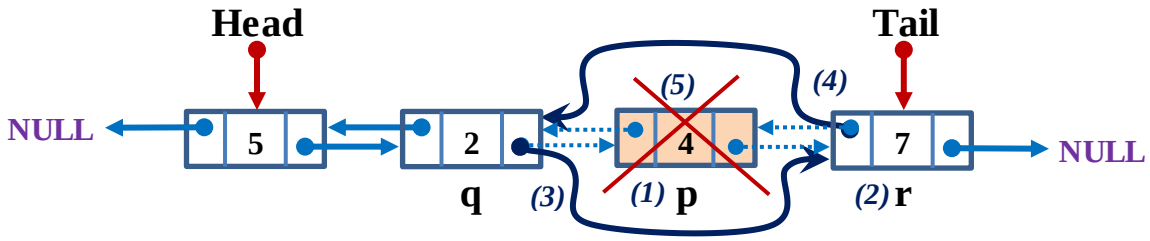
    Ngược lại: Tail→Next = NULL;
  - + **Bước 2.4:** Lưu giữ lại giá trị nút bị xóa
  - + **Bước 2.5:** delete(p); *//Hủy nút do con trỏ p trỏ đến*

Cài đặt:

```
1.  int deleteTail(DList &dl, ItemType &x)
2.  {
3.      if(isEmpty(dl) == 1)
4.          return 0;   //Thực hiện không thành công
5.      DNode* p = dl.Tail;
```

```
6.      dl.Tail = dl.Tail→Prev;
7.      if(dl.Tail == NULL)
8.          dl.Head = NULL;
9.      else
10.         dl.Tail→Next = NULL;
11.     x = p→Info;     //Lấy thông tin của nút bị hủy
12.     delete p;   //Hủy nút do p trỏ đến
13.     return 1;   //Thực hiện thành công
14. }
```

3.5.2.6.3. Xóa phần tử sau phần tử q của danh sách



Hình 3.37: Hình vẽ minh họa xóa sau nút q của danh sách liên kết đôi

Giải thuật:

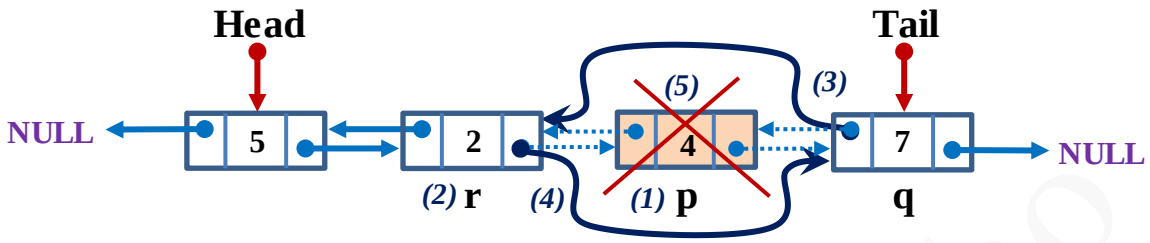
- **Bước 1:** Nếu (q == NULL hoặc q == Tail) thì: Không thực hiện gì cả.
- **Bước 2:** Thực hiện việc xóa nút p kế sau nút q như sau:
  - + **Bước 2.1:** Gán p = q→Next;
  - + **Bước 2.2:** Gán r = p→Next; q→Next = r;
  - + **Bước 2.3:** Nếu (r == NULL) thì: Tail = q;
  - + **Bước 2.4:** Ngược lại: r→Prev = q;
  - + **Bước 2.5:** Lưu giữ lại giá trị nút bị xóa
  - + **Bước 2.6:** delete(p); *//Hủy nút do con trỏ p trỏ đến*

Cài đặt:

```
1.  int deleteAfter(DList &dl, DNode* q, ItemType &x)
2.  {//Hủy bỏ nút kế sau nút q do con trỏ p trỏ đến
3.      if(q == NULL || q == dl.Tail)
4.          return 0;   //Thực hiện không thành công
5.      DNode* p = q→Next;
6.      DNode* r = p→Next;
7.      q→Next = r;
8.      if(r == NULL)
9.          dl.Tail = q;
10.     else
11.         r→Prev = q;
```

```
12.     x = p→Info;     //Lấy thông tin của nút bị hủy
13.     delete p;   //Hủy nút do p trỏ đến
14.     return 1;   //Thực hiện thành công
15. }
```

3.5.2.6.4. Xóa phần tử trước phần tử q của danh sách



Hình 3.38: Hình vẽ minh họa xóa trước nút q của danh sách liên kết đôi

Giải thuật:

- **Bước 1:** Nếu (q == NULL hoặc q == Head) thì: Không thực hiện gì cả.
- **Bước 2:** Thực hiện việc xóa nút kế trước nút q như sau:
  - + **Bước 2.1:** Gán p = q→Prev;
  - + **Bước 2.2:** Gán r = p→Prev; q→Prev = r;
  - + **Bước 2.3:** Nếu (r == NULL) thì: Head = q;
  - + **Bước 2.4:** Ngược lại: r→Next = q;
  - + **Bước 2.5:** Lưu giữ lại giá trị nút bị xóa
  - + **Bước 2.6:** delete(p); *//Hủy nút do p trỏ đến*

Cài đặt:

```
1.  int deleteBefore(DList &dl, DNode* q, ItemType &x)
2.  { //Hủy bỏ nút kế trước nút q do con trỏ p trỏ đến
3.      if(q == NULL || q == dl.Head)
4.          return 0;   //Thực hiện không thành công
5.      DNode* p = q→Prev;
6.      DNode* r = p→Prev;
7.      q→Prev = r;
8.      if(r == NULL)
9.          dl.Head = q;
10.     else
11.         r→Next = q;
12.     x = p→Info;     //Lấy thông tin của nút bị hủy
13.     delete p;   //Hủy nút do p trỏ đến
14.     return 1;   //Thực hiện thành công
15. }
```

3.5.2.6.5. Xóa một phần tử có khóa x

Giải thuật:

- **Bước 1:** Gán p = findDNode(**dl**, x); *//p là phần tử có khóa x cần hủy.*
- **Bước 2:** Nếu (p == NULL) thì: Không thực hiện gì cả.
- **Bước 3:** Khi (p != NULL) thì thực hiện các công việc sau:
  - + **Bước 3.1:** Nếu (p == Head) thì: Xóa nút đầu deleteHead(**dl**, x);
  - + **Bước 3.2:** Ngược lại, Nếu (p == Tail) thì: Xóa cuối deleteTail(**dl**, x);
  - + **Bước 3.3:** Ngược lại:

    q = p→Prev;

    r = p→Next;

    q→Next = r;

    r→Prev = q;
  - + **Bước 3.4:** delete(p); *//Hủy nút do p trỏ đến*

Cài đặt:

```
1.  int deleteDNodeX(DList &dl, ItemType x)
2.  {
3.      DNode* p = findDNode(dl, x);
4.      if(p == NULL)
5.          return 0;   //Thực hiện không thành công
6.      if(p == dl.Head)
7.          deleteHead(dl, x);
8.      else if(p == dl.Tail)
9.          deleteTail(dl, x);
10.     else
11.     {
12.         DNode* q = p→Prev;
13.         DNode* r = p→Next;
14.         q→Next = r;
15.         r→Prev = q;
16.         delete p;   //Hủy nút do p trỏ đến
17.         return 1;   //Thực hiện thành công
18.     }
19. }
```

Hoặc có thể sử dụng lại hàm **deleteAfter** như sau:

```
1.  int deleteDNodeX(DList &dl, ItemType x)
2.  {
3.      DNode* p = findDNode(dl, x);
4.      if(p == NULL)
5.          return 0;   //Thực hiện không thành công
6.      if(p == dl.Head)
7.          deleteHead(dl, x);
```

```
8.      else if(p == dl.Tail)
9.              deleteTail(dl, x);
10.     else
11.     {
12.             DNode* q = p→Prev;    //q là nút kết trước p
13.             deleleAfter(dl, q, x);    //Xóa nút kế sau q →xóa p
14.             return 1;  //Thực hiện thành công
15.     }
16. }
```

3.5.2.7. Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách. Như là:

- Đếm các phần tử của danh sách.
- Tìm tất cả các phần tử thoả điều kiện.
- Hủy toàn bộ danh sách (và giải phóng bộ nhớ).

3.5.2.7.1. Giải thuật duyệt xuôi

- **Bước 1:** Nếu danh sách rỗng thì: Thông báo và không thực hiện
- **Bước 2:** Gán p = Head; *//p lưu địa chỉ của phần tử đầu trong DList*
- **Bước 3:**

  Lặp lại trong khi (danh sách chưa hết) thì thực hiện:

  + Xử lý phần tử p

  + p = p→Next; *//Xét phần tử kế sau*

Cài đặt (*tổng quát*):

```
1.  void processDList(DList dl)
2.  {
3.      if(isEmpty(dl) == 1)
4.      {
5.              printf("\nDanh sách rỗng!");
6.              return;
7.      }
8.      DNode* p = dl.Head;
9.      while(p != NULL)
10.     {
11.             processDNode(p);     //Xử lý nút p tuỳ trường hợp cụ thể
12.             p = p→Next;
```

```
13.     }
14. }
```

Ví dụ: Hàm in nội dung danh sách ra màn hình theo chiều xuôi sẽ được viết cụ thể như sau:

```
1.  void showDList(DList dl)
2.  {
3.      if(isEmpty(dl) == 1)
4.      {
5.          printf("\nDanh sách rỗng!");
6.          return;
7.      }
8.      printf("\nNội dung danh sách: ");
9.      DNode* p = dl.Head;
10.     while(p != NULL)
11.     {
12.         printf("%4d", p→Info);
13.         p = p→Next;
14.     }
15. }
```

3.5.2.7.2. Giải thuật duyệt ngược

- **Bước 1:** Nếu (Tail == NULL) thì: Thông báo danh sách rỗng, không thực hiện
- **Bước 2:** Gán p = Tail; *//p lưu địa chỉ của phần tử đầu trong DList*
- **Bước 3:**

  Lặp lại trong khi (danh sách chưa hết) thì thực hiện:

  + Xử lý phần tử p

  + p = p→Prev; *//Xét phần tử kế trước*

Cài đặt (*tổng quát*):

```
1.  void processDList(DList dl)
2.  {
3.      if(isEmpty(dl) == 1)
4.      {
5.          printf("\nDanh sách rỗng!");
6.          return;
7.      }
8.      DNode* p = dl.Tail;
9.      while(p != NULL)
10.     {
11.         processDNode(p);     //Xử lý p tuỳ trường hợp cụ thể
12.         p = p→Prev;
13.     }
14. }
```

Ví dụ: Hàm in nội dung danh sách ra màn hình theo chiều ngược sẽ được viết cụ thể như sau:

```
1.  void showDList(DList dl)
2.  {
3.      if(isEmpty(dl) == 1)
4.      {
5.          printf("\nDanh sách rỗng!");
6.          return;
7.      }
8.      printf("\nNội dung danh sách: ");
9.      DNode* p = dl.Tail;
10.     while(p != NULL)
11.     {
12.         printf("%4d", p→Info);
13.         p = p→Prev;
14.     }
15. }
```

3.5.2.8. Hủy toàn bộ danh sách

Hủy tất cả các phần tử của danh sách đồng nghĩa với việc thu hồi lại những vùng nhớ đã cấp phát cho các nút của danh sách. Duyệt danh sách một cách tuần tự từ đầu đến cuối (hoặc từ cuối về đầu) để hủy lần lượt từng nút một.

Giải thuật duyệt xuôi:

- **Bước 1:** Lặp lại trong khi (danh sách còn phần tử) thì thực hiện:
  - + **Bước 1.1:** Gán p = Head;
  - + **Bước 1.2:** Gán Head = Head→Next; *//Cập nhật lại Head*
  - + **Bước 1.3:** Hủy con trỏ p;
- **Bước 2:** Gán Tail = NULL; *//bảo toàn tính nhất quán khi danh sách rỗng*

Cài đặt:

```
1.  int deleteDList(DList &dl)
2.  {
3.      if(isEmpty(dl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      while(dl.Head != NULL)
6.      {
7.          DNode* p = dl.Head;
8.          dl.Head = dl.Head→Next;
16.         delete p;  //Hủy nút do p trỏ đến
9.      }
10.     dl.Tail = NULL;
```

```
11.     return 1;  //Thực hiện thành công
12. }
```

3.5.2.9. Sắp xếp danh sách

Một danh sách có thứ tự (danh sách được sắp xếp) là một danh sách mà các phần tử của nó được sắp xếp theo một thứ tự nào đó dựa trên một trường khóa. Để sắp xếp một danh sách, ta có thể thực hiện một trong hai phương án sau:

3.5.2.9.1. **Phương án 1:** Hoán vị nội dung các phần tử trong danh sách (*vùng Info*)

Với phương án này, có thể chọn một trong những giải thuật sắp xếp đã biết để cài đặt lại trên danh sách như thực hiện trên mảng, điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên danh sách thông qua liên kết thay vì chỉ số như trên mảng. Do dựa trên việc hoán vị nội dung của các phần tử, phương án này đòi hỏi sử dụng thêm vùng nhớ trung gian nên chỉ thích hợp với các danh sách mà thành phần Info của các phần tử có kích thước nhỏ. Hơn nữa, số lần hoán vị có thể lên đến bậc $\mathbf{n^2}$ với danh sách có n phần tử, không tận dụng được các ưu điểm của danh sách.

Ví dụ: Cài đặt giải thuật sắp xếp nổi bọt trên danh sách như sau:

```
1.  void bubbleSort_DList_Ascending (DList &dl)
2.  {
3.      if(isEmpty(dl) == 1 || dl.Head == dl.Tail) return;
4.      for(DNode* p = dl.Head; p != dl.Tail; p = p→Next)
5.      {
6.          for(DNode* q = dl.Tail; q != p; q = q→Prev)
7.              if(q→Info < q→Prev→Info)
8.                  swap(q→Info, q→Prev→Info);
9.      }
10. }
```

3.5.2.9.2. **Phương án 2:** Thay đổi các mối kiên kết (*hai con trỏ Next và Prev*)

Tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ (đồng thời hủy danh sách cũ). Giả sử danh sách mới sẽ được quản lý bằng danh sách *Result*, và giải thuật như sau:

- **Bước 1:** Khởi tạo danh sách mới *Result* là rỗng.
- **Bước 2:** Tìm trong danh sách *DListOld* thì phần tử min là phần tử nhỏ nhất.
- **Bước 3:** Tách min khỏi danh sách *DListOld*.
- **Bước 4:** Chèn min vào cuối danh sách *Result*.

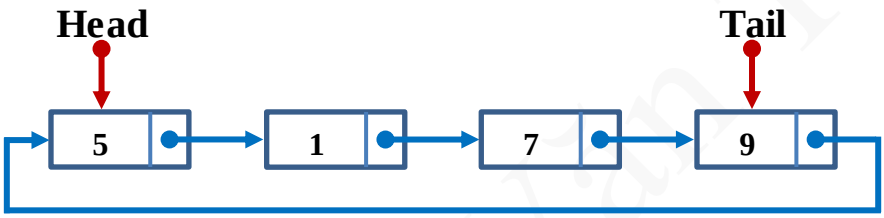
- **Bước 5:** Quay lại Bước 2 khi chưa hết danh sách *DListOld*.

Cài đặt: (*Sinh viên tự thực hiện xem như 1 bài tập*)

3.6. Danh sách liên kết vòng

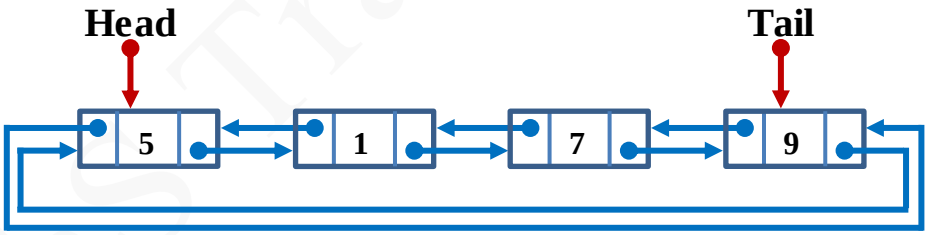
Danh sách liên kết vòng về cơ bản được phát triển mở rộng từ danh sách liên kết đơn và danh sách liên kết đôi. Vì vậy danh sách liên kết vòng cũng có hai loại là danh sách liên kết vòng đơn và danh sách liên kết vòng đôi.

Danh sách liên kết vòng đơn có phần tử cuối liên kết với phần tử đầu của danh sách.



Hình 3.39: Hình vẽ minh họa danh sách liên kết vòng đơn

Danh sách liên kết vòng đôi có phần tử cuối liên kết với phần tử đầu và phần tử đầu liên kết với phần tử cuối danh sách.



Hình 3.40: Hình vẽ minh họa danh sách liên kết vòng đôi

Tuy nhiên, giáo trình này chỉ trình bày danh sách liên kết vòng đơn. Vì vậy tác giả quy ước dùng thuật ngữ danh sách liên kết vòng thay cho danh sách liên kết vòng đơn.

3.6.1. Tổ chức danh sách

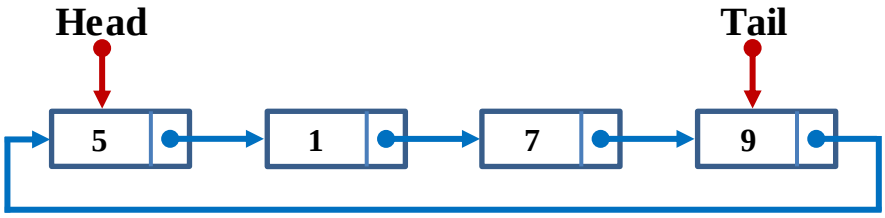
Cấu trúc của một danh sách liên kết vòng như sau:

➔Ta có cấu trúc dữ liệu của một nút trong danh sách liên kết đơn như sau:

```
1. typedef int ItemType; //ItemType cũng có thể là kiểu float, SINHVIEN, PHANSO, ...
2. struct CNode
3. {
4.         ItemType Info;     //Lưu thông tin (dữ liệu) của bản thân.
```

```
5.      CNode* Next;    //Con trỏ trỏ đến CNode kế sau.
6.  };
7.  struct CList
8.  {//Kiểu danh sách liên kết vòng
9.      CNode* Head;    //Lưu địa chỉ Node đầu tiên trong CList
10.     CNode* Tail;    //Lưu địa chỉ của Node cuối cùng trong CList
11. };
```

Lúc này ta có danh sách liên kết vòng như sau:

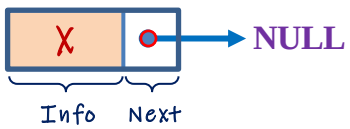


Hình 3.41: Hình vẽ minh họa cấu trúc danh sách liên kết vòng đơn

3.6.2. Các thao tác cơ bản trên danh sách liên kết vòng

- Tạo một nút có trường Info bằng x.
- Khởi tạo một danh sách liên kết vòng rỗng.
- Kiểm tra danh sách liên kết vòng rỗng.
- Thêm một phần tử có khóa x vào danh sách.
- Tìm một phần tử có Info bằng x.
- Hủy một phần tử trong danh sách.
- Duyệt danh sách.
- Sắp xếp danh sách.

3.6.2.1. Tạo một node mới



Hình 3.42: Hình vẽ minh họa tạo một nút mới cho danh sách vòng đơn

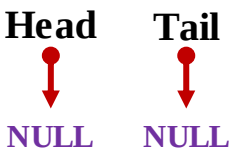
Hàm trả về địa chỉ phần tử mới tạo như sau:

```
1.  CNode* createCNode(ItemType x)
2.  {
3.      CNode* p = new CNode;
4.      if(p == NULL)
5.      {
6.          printf("Không đủ bộ nhớ để cấp phát nút mới!");
7.          getch();
8.          return NULL;
```

```
9.      }
10.     p→Info = x;
11.     p→Next = NULL;
12.     return p;
13. }
```

3.6.2.2. Khởi tạo danh sách rỗng

Địa chỉ của nút đầu tiên, địa chỉ của nút cuối cùng đều không có.




```
1. void initCList(CList &cl)
2. {
3.      cl.Head = NULL;
4.      cl.Tail = NULL;
5. }
```

3.6.2.3. Kiểm tra danh sách liên kết vòng rỗng

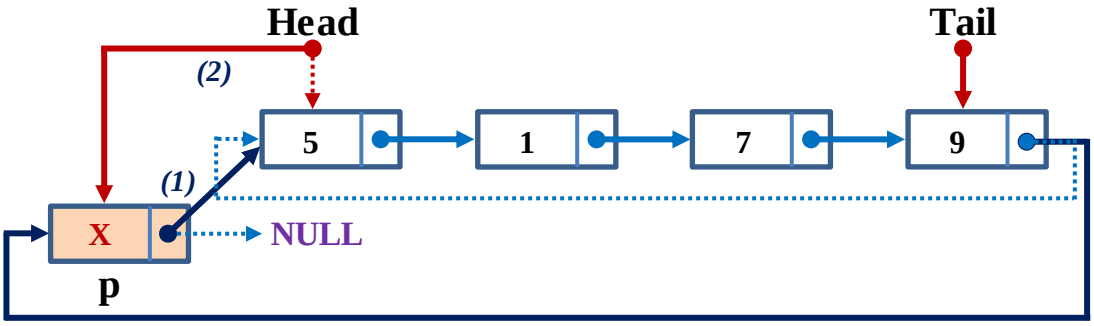
Kiểm tra danh sách liên kết vòng đã có phần tử nào hay chưa? Nghĩa là kiểm tra con trỏ đầu Head (*hoặc con trỏ cuối Tail*) có bằng NULL hay không? Hàm trả sẽ về 1 nếu danh sách liên kết vòng chưa có phần tử nào (*rỗng*), ngược lại (*không rỗng*) thì hàm sẽ trả về 0.

```
1. int isEmpty(CList cl)
2. {
3.      if(cl.Head == NULL)
4.          return 1;  //Nếu danh sách rỗng
5.      else
6.          return 0;  //Nếu danh sách không rỗng
7. }
```

3.6.2.4. Thêm một phần tử vào danh sách

- Nguyên tắc thêm: Khi thêm một phần tử vào CList thì sẽ làm cho Head và Tail thay đổi, nên cần phải cập nhật lại chúng.
- Các vị trí cần thêm:
  + Thêm phần tử vào đầu CList.
  + Thêm phần tử vào cuối CList.
  + Thêm phần tử vào sau phần tử q.
  + Thêm phần tử vào trước phần tử q.

Giáo trình này chỉ trình bày trường hợp thêm vào đầu danh sách. Các trường hợp còn lại dành cho các bạn sinh viên (*xem như bài tập*).



Hình 3.43: Hình vẽ minh họa thêm một nút vào đầu danh sách vòng đơn

Giải thuật:

- **Bước 1:** Nếu phần tử muốn thêm p không tồn tại thì: không thực hiện.
- **Bước 2:** Nếu CList rỗng thì:
  + Head = Tail = p;
- **Bước 3:** Ngược lại:
  + p→Next = Head;
  + Head = p;
- **Bước 4:** Cho phần tử cuối trỏ về phần tử đầu mới của danh sách
  + Tail→Next = Head;

Cài đặt:

```
1.  int insertHead(CList &cl, CNode* p)
2.  {
3.      if(p == NULL)
4.          return 0;  //Thực hiện không thành công
5.      if(isEmpty(cl) == 1)
6.      {
7.          cl.Head = p;
8.          cl.Tail = p;
9.      }
10.     else
11.     {
12.         p→Next = cl.Head;
13.         cl.Head = p;
14.     }
15.     cl.Tail→Next = cl.Head;
16.     return 1;  //Thực hiện thành công
17. }
```

3.6.2.5. Tìm kiếm phần tử trong danh sách

Danh sách liên kết vòng đòi hỏi truy xuất tuần tự, do đó áp dụng giải thuật tìm tuyến tính để xác định phần tử trong danh sách có khóa bằng với x. Sử dụng một con trỏ phụ trợ p để lần lượt trỏ đến các phần tử trong danh sách. Nếu tìm thấy có một nút có khóa bằng với x thì trả về con trỏ chứa nút đó, ngược lại nếu duyệt hết danh sách mà vẫn không tìm thấy nút nào có khóa bằng với x thì trả về NULL.

Các bước của giải thuật tìm nút có Info bằng x trong list vòng:

- **Bước 1:** Nếu danh sách rỗng thì: Trả về NULL
- **Bước 2:** Nếu (Head→Info == x) thì: Trả về Head *//Có phần tử cần tìm.*
- **Bước 3:**

  p = Head→Next; *// phần tử kế sau phần tử đầu danh sách*

  Lặp lại trong khi (p→Info != x và p != Head) thì thực hiện:

  p = p→Next; *//xét phần tử kế*
- **Bước 4:**

  Nếu (p→Info == x) thì: Trả về p *//Có phần tử cần tìm.*

  Ngược lại: Trả về NULL *//Không có phần tử cần tìm.*

Cài đặt:

```
1.  CNode* findCNode(CList cl, ItemType x)
2.  {
3.      if(isEmpty(cl) == 1)
4.          return NULL;    //Danh sách rỗng
5.      if(cl.Head→Info == x)
6.          return cl.Head; //Tìm thấy
7.      CNode *p = cl.Head→Next;
8.      while( (p != cl.Head) && (p→Info != x) )
9.          p = p→Next;
10.     if(p→Info == x)
11.         return p;   //Tìm thấy
12.     return NULL;    //Không tìm thấy
13. }
```

3.6.2.6. Xóa phần tử của danh sách

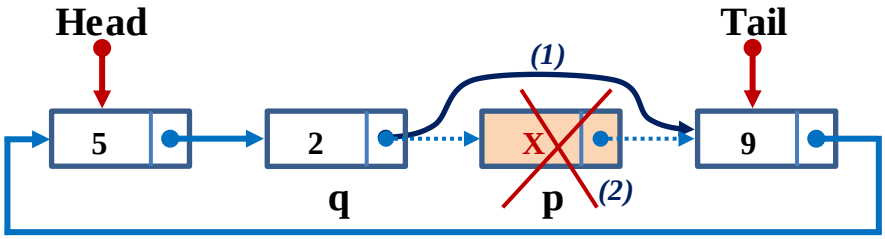
- Nguyên tắc: Phải cô lập phần tử cần xóa trước khi hủy.
- Các vị trí cần xóa:
  + Xóa phần tử đứng đầu CList.
  + Xóa phần tử đứng cuối CList.

+ Xóa phần tử đứng sau q trong danh sách liên kết vòng.

+ Xóa phần tử có khóa bằng x.

Giáo trình này chỉ trình bày trường hợp xóa phần tử có khóa bằng x trong danh sách. Các trường hợp còn lại dành cho các bạn sinh viên (*xem như bài tập*).

Ở phần trên, các phần tử trong danh sách liên kết vòng được cấp phát vùng nhớ động bằng hàm **new**, thì sẽ được giải phóng vùng nhớ bằng hàm **delete**.



Hình 3.44: Hình vẽ minh họa xóa nút có khóa x trong danh sách vòng đơn

Giải thuật:

- **Bước 1:** p = findCNode(**cl**, x); *//p là phần tử có khóa x cần hủy.*
- **Bước 2:** Nếu (p == NULL) thì: Không có nút nào có giá trị bằng x nên không thực hiện.
- **Bước 3:** Khi (p != NULL) thì:
  + **Bước 3.1:** Nếu (p == Head) thì:
    Nếu (Head == Tail) thì: *//Danh sách chỉ có 1 nút*
      Gán lại giá trị cho Head và Tail: Head = Tail = NULL;
    Ngược lại:
      Head = Head→Next;
      Tail→Next = Head;
  + **Bước 3.2:** Ngược lại:
    q = findCNodePrevious(**cl**, p); *//q là nút kế trước p*
    q→Next = p→Next;
    Nếu (Tail == p) thì:
      Gán lại con trỏ Tail là: Tail = q;
  + **Bước 3.3:** delete(p); *//Hủy nút do con trỏ p trỏ đến*

Cài đặt:

```
1.  CNode* findCNodePrevious(CList cl, CNode* p)
2.  { //Tìm nút q là nút kế trước p
3.      CNode* q = cl.Head;
4.      if(q == p)
5.          return p;  //danh sách chỉ có 1 nút
6.      else
7.      {
8.          while(q→Next != p)
9.              q = q→Next;
10.         return q;  //q là nút đứng kế trước nút p
11.     }
12. }
```

```
1.  int deleteCNodeX(CList &cl, ItemType x)
2.  {
3.      CNode *q, *p = findCNode(cl, x);
4.      if(p == NULL)
5.          return 0;  //Thực hiện không thành công
6.      if(p == cl.Head)
7.      {
8.          if(cl.Head == cl.Tail)   //Danh sách chỉ có 1 nút
9.              cl.Head = cl.Tail = NULL;
10.         else
11.         {
12.             cl.Head = cl.Head→Next;
13.             cl.Tail→Next = cl.Head;
14.         }
15.     }
16.     else
17.     {
18.         q = findCNodePrevious(cl, p);
19.         q→Next = p→Next;
20.         if(cl.Tail == p)
21.             cl.Tail = q;
22.     }
23.     delete p;   //Hủy nút do p trỏ đến
24.     return 1;   //Thực hiện thành công
25. }
```

3.6.2.7. Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách. Như là:

- Đếm các phần tử của danh sách.
- Tìm tất cả các phần tử thoả điều kiện.

– Hủy toàn bộ danh sách (và giải phóng bộ nhớ).

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:** Gán p = Head; *//p lưu địa chỉ của phần tử đầu trong Clist*
- **Bước 3:**

Lần lượt thực hiện các công việc sau:

- Xử lý phần tử p;
- p = p→Next; *//Xét phần tử kế*

Lặp lại trong khi (p != **cl**.Head):

Cài đặt (*tổng quát*):

```
1.  void processCList(CList &cl)
2.  {
3.      if(isEmpty(cl) == 1)
4.      {
5.          printf("\nDanh sach rong!");
6.          return;
7.      }
8.      CNode *p = cl.Head;
9.      do
10.     {
11.         ProcesCNode(p);      //xử lý node p theo yêu cầu cụ thể
12.         p = p→Next;
13.     }while(p != cl.Head);
14. }
```

Ví dụ: Hàm in nội dung danh sách ra màn hình sẽ được viết cụ thể như sau:

```
1.  void showCList(CList cl)
2.  {
3.      if(isEmpty(cl) == 1)
4.      {
5.          printf("\nDanh sach rong!");
6.          return;
7.      }
8.      printf("\nNoi dung cua danh sach la: ");
9.      CNode *p = cl.Head;
10.     do
11.     {
12.         printf("%4d", p→Info);
13.         p = p→Next;
14.     }while(p != cl.Head);
15. }
```

3.6.2.8. Hủy toàn bộ danh sách

Hủy tất cả các phần tử của danh sách đồng nghĩa với việc thu hồi lại những vùng nhớ đã cấp phát cho các nút của danh sách. Duyệt danh sách một cách tuần tự từ đầu đến cuối để hủy lần lượt từng nút một.

Giải thuật:

- **Bước 1:** Nếu danh sách rỗng thì: Không thực hiện gì cả.
- **Bước 2:**
  - + **Bước 2.1:**

    p = Head→Next;

    Lặp lại trong khi (p != Head) thì thực hiện:

    - q = p;
    - p = p→Next;
    - Hủy q;
  - + **Bước 2.2:** Hủy p; *//Vì lúc này con trỏ p đang trỏ đến Head*

    Gán Head = Tail = NULL; *//Gán lại 2 con trỏ Head và Tail trỏ NULL*

Cài đặt:

```
1.  int deleteCList(CList &cl)
2.  {
3.      if(isEmpty(cl) == 1)
4.          return 0;  //Thực hiện không thành công
5.      CNode *p, *q;
6.      p = cl.Head→Next;
7.      while(p != cl.Head)
8.      {
9.          q = p;
10.         p = p→Next;
11.         delete q;
12.     }
13.     cl.Head = cl.Tail = NULL;
14.     delete p;   //Lúc này p đang trỏ đến vị trí cl.Head cũ;
15.     return 1;   //Thực hiện thành công
16. }
```

3.7. Ưu nhược điểm của danh sách liên kết

Do các phần tử (nút) được lưu trữ không liên tiếp nhau trong bộ nhớ, do vậy danh sách liên kết có các ưu nhược điểm sau đây:

– Mật độ sử dụng bộ nhớ của danh sách liên kết không tối ưu tuyệt đối (<100%).

- Việc truy xuất và tìm kiếm các phần tử của danh sách liên kết mất nhiều thời gian bởi luôn luôn phải duyệt tuần tự qua các phần tử trong danh sách.
- Tận dụng được những không gian bộ nhớ nhỏ để lưu trữ từng nút, tuy nhiên bộ nhớ lưu trữ thông tin mỗi nút lại tốn nhiều hơn do còn phải lưu thêm thông tin về vùng liên kết. Như vậy nếu vùng dữ liệu của mỗi nút là lớn hơn thì tỷ lệ mức tiêu tốn bộ nhớ này là không đáng kể, ngược lại thì nó lại gây lãng phí bộ nhớ.
- Việc thêm, bớt các phần tử trong danh sách, tách/ghép các danh sách khá dễ dàng do chúng ta chỉ cần thay đổi mối liên kết giữa các phần tử với nhau.

3.8. Câu hỏi và bài tập

3.8.1. Câu hỏi

1. Phân tích ưu điểm và khuyết điểm của danh sách liên kết với mảng. Tổng quát hoá các trường hợp nên dùng danh sách liên kết.
2. Hiểu được các thao tác thêm, xóa, sửa, tìm kiếm nút. Nắm được các điểm mạnh của cấu trúc dữ liệu dạng danh sách liên kết.
3. Xây dựng một cấu trúc dữ liệu sinh viên thích hợp. Với các thao tác sau:
   - Thêm một sinh viên.
   - Xóa một sinh viên.
   - Sửa một sinh viên.
   - Sắp xếp tăng dần danh sách sinh viên đó theo MaSV hoặc tên.
   - Tìm kiếm sinh viên theo MaSV hoặc tên.
4. Nắm được cơ chế hoạt động của Stack, Queue. Và các thao tác thêm một phần tử vào Stack, Queue và lấy một phần tử ra khỏi Stack, Queue.

   Ứng dụng vào việc lưu các kí tự của một chuỗi bằng Queue/Stack. Hãy cho biết nội dung của Queue và Stack sau mỗi thao tác trong dãy: "hell*o***w*orld**".

   Biết rằng: Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào Queue/Stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong Stack/Queue in lên màn hình.

5. Một ma trận chỉ chứa rất ít phần tử với giá trị có nghĩa (ví dụ: phần tử $\neq 0$) được gọi là ma trận thưa.

   Ví dụ:

$$\begin{pmatrix} 0 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \end{pmatrix}$$

Dùng cấu trúc xâu liên kết để tổ chức biểu diễn một ma trận thưa sao cho tiết kiệm nhất (chỉ lưu trữ các phần tử có nghĩa).

- Viết chương trình cho phép nhập, xuất ma trận.
- Viết chương trình con cho phép cộng hai ma trận.

6. Giả sử xây dựng một chương trình soạn thảo văn bản, hãy chọn cấu trúc dữ liệu thích hợp để lưu trữ văn bản trong quá trình soạn thảo. Biết rằng:
   - Số dòng văn bản không hạn chế.
   - Mỗi dòng văn bản có chiều dài tối đa 80 tý tự.
   - Các thao tác gồm:
     + Di chuyển trong văn bản.
     + Thêm, xóa sửa ký tự trong một dòng.
     + Thêm xóa sửa một dòng trong văn bản.

3.8.2. Bài tập

1. Cài đặt tất cả các thao tác cơ bản trên danh sách liên kết đơn, danh sách kiên kết đôi, danh sách kiên kết vòng.
2. Cài đặt giải thuật sắp xếp chèn trực tiếp trên danh sách đôi. Nhận xét xem nó có phát huy ưu thế của giải thuật hơn trên mảng hay không?
3. Cài đặt giải thuật Quick Sort theo kiểu không đệ quy.
4. Cài đặt lại chương trình quản lý sinh viên theo như bài tập 3.
5. Cài đặt lại chương trình soạn thao văn bản theo mô tả trong bài tập 4.
6. Cài đặt chương trình tạo một bảng tính cho phép thực hiện các phép tính + , -, *, /, % trên các số tối đa 30 chữ số.
7. Cài đặt chương trình cho phép nhập vào một biểu thức gồm các số, các toán tử + , -, *, /, %, các hàm toán học sin, cos, tan, ln, ex, dấu mở đóng ngoặc “(”, ”)” và tính toán giá trị của biểu thức.
8. Cài đặt chương trình cho phép nhập vào hai đa thức gồm các toán hạng là các số, các toán tử + , -, *, và biến x, và thực hiện việc + , -, *,/hai đa thức.

Chương 4. NGĂN XẾP VÀ HÀNG ĐỢI

Trong các thao tác trên danh sách không phải lúc nào cũng có thể thực hiện được tất cả mà nhiều khi các thao tác này bị hạn chế trong một số loại danh sách, đó là danh sách hạn chế.

Như vậy, danh sách hạn chế là danh sách mà các thao tác trên đó bị hạn chế trong một chừng mực nào đó tùy thuộc vào danh sách. Trong phần này chúng ta xem xét hai loại danh sách hạn chế chủ yếu đó là: Ngăn xếp (Stack), Hàng đợi (Queue).

4.1. Ngăn xếp - Stack

4.1.1. Định nghĩa

Ngăn xếp là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh sách và thao tác lấy ra một phần tử từ trong danh sách được thực hiện ở cùng một đầu (*hay còn gọi là đỉnh*).

Như vậy, các phần tử được đưa vào trong ngăn xếp sau cùng sẽ được lấy ra trước tiên, phần tử đưa vào trong ngăn xếp trước tiên sẽ được lấy ra sau cùng. Do đó mà ngăn xếp còn được gọi là danh sách vào sau ra trước (LIFO List) và cấu trúc dữ liệu này còn được gọi là cấu trúc **LIFO** (**Last In – First Out**).

Có nhiều cách để biểu diễn và tổ chức các ngăn xếp:

- Sử dụng danh sách đặc (*mảng 1 chiều*);
- Sử dụng danh sách liên kết.

Do ở đây cả hai thao tác thêm vào và lấy ra đều được thực hiện ở một đầu nên chúng ta chỉ cần quản lý vị trí đầu của danh sách dùng làm mặt (đỉnh) cho ngăn xếp thông qua biến chỉ số đỉnh Top. Chỉ số này có thể là cùng chiều (đầu) hoặc ngược chiều (cuối) với thứ tự các phần tử trong mảng và trong danh sách liên kết. Điều này có nghĩa là đỉnh ngăn xếp có thể là đầu mảng, đầu danh sách liên kết mà cũng có thể là cuối mảng, cuối danh sách liên kết. Để thuận tiện, ở đây chúng ta giả sử đỉnh của ngăn xếp là đầu mảng, đầu danh sách liên kết. Trường hợp ngược lại, sinh viên tự áp dụng tương tự.

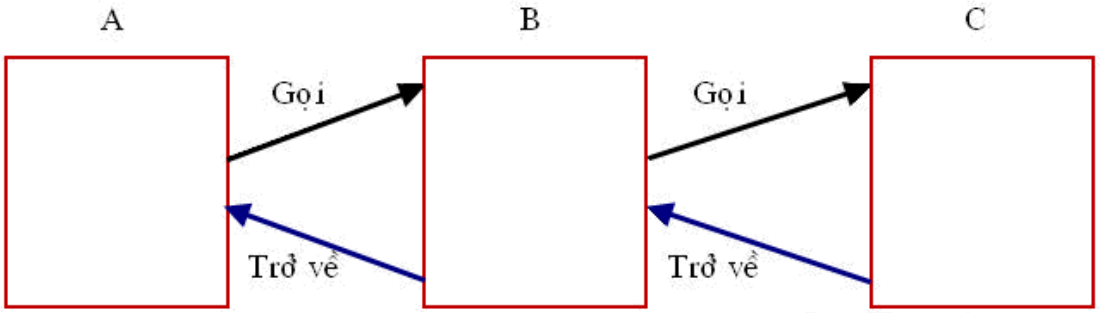
Ở đây chúng ta cũng sẽ biểu diễn và tổ chức ngăn xếp bằng danh sách đặc và bằng danh sách liên kết đơn được quản lý bởi con trỏ đầu danh sách. Do vậy cấu trúc dữ liệu của ngăn xếp và các thao tác trên đó sẽ được trình bày thành hai trường hợp khác nhau.

4.1.2. Ứng dụng của Stack

Stack có nhiều ứng dụng, chẳng hạn như:

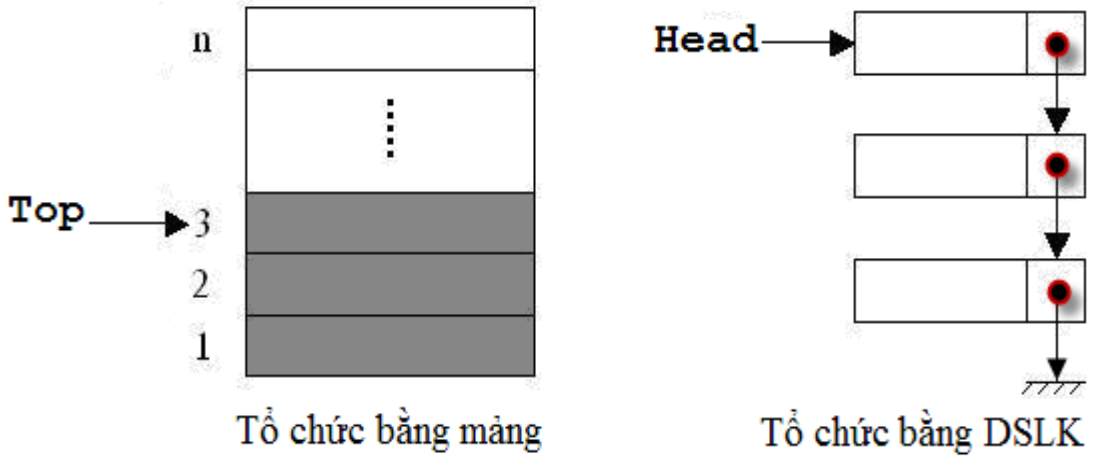
- Trình biên dịch.
- Khử đệ quy đuôi.
- Lưu vết các quá trình quay lui, vét cạn.

Ví dụ: Chương trình A gọi chương trình B, chương trình B gọi chương trình C. Khi chương trình C được thực hiện xong thì sự điều khiển chương trình sẽ trở về thực hiện chương trình B, rồi khi chương trình B được thực hiện xong thì sự điều khiển chương trình sẽ trở về thực hiện chương trình A. Như vậy chương trình B được gọi sau sẽ được trở về thực hiện trước chương trình A. Đó là nhờ điểm nhập (entry point) trở về của các chương trình được chứa trong Stack.



Hình 4.1: Hình vẽ minh họa cơ chế Stack

Stack có thể được tổ chức theo dạng mảng hoặc theo danh sách liên kết. Vì phép thêm vào và phép loại bỏ chỉ được thực hiện ở cùng một đầu nên ta chỉ cần một chỉ điểm gọi là con trỏ của Stack (Stack pointer).



Hình 4.2: Hình vẽ minh họa cấu trúc tổ chức Stack

4.1.3. Các thao tác trên Stack

Stack chủ yếu dùng các thao tác sau:

- initStack(&s): Khởi tạo Stack mới s.
- isEmpty(s): Kiểm tra Stack s có rỗng hay không.
- push(s, o): Thêm đối tượng o vào đỉnh của Stack s.
- pop(s, x): Lấy đối tượng từ đỉnh của Stack s và gán giá trị của nó cho x.
- getTop(s, x): Lấy giá trị của đối tượng ở đỉnh của Stack s và gán giá trị đó cho x mà không hủy đối tượng này ra khỏi Stack s.

4.1.4. Biểu diễn Stack bằng mảng

4.1.4.1. Tổ chức dữ liệu

Ta định nghĩa Stack s là một mảng chứa các phần tử.

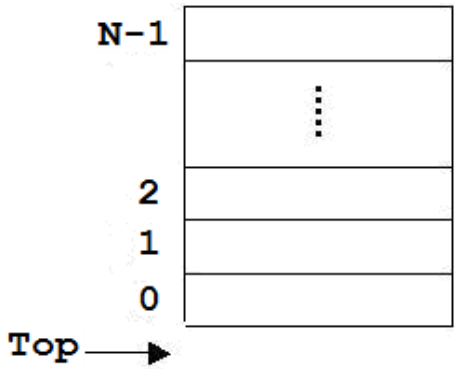
Hình 4.3: Hình vẽ minh họa tổ chức Stack bằng mảng một chiều

```
1. #define MAXSIZE 100
2. typedef int ItemType;
3. struct Stack
4. {
5.       ItemType Info[MAXSIZE];
6.       int Top;
7. };
```

4.1.4.2. Các thao tác trên Stack

4.1.4.2.1. Khởi tạo Stack

Sau khi khởi tạo thì Stack phải rỗng, nên giá trị của Top sẽ bằng **-1**.



Hình 4.4: Hình vẽ minh họa khởi tạo Stack rỗng

```
1. void initStack(Stack &s)
2. {
3.       s.Top = -1;
4. }
```

4.1.4.2.2. Kiểm tra Stack rỗng

```
1. int isEmpty(Stack s)
2. {//Hàm kiểm tra Stack có rỗng hay không
3.       if(s.Top == -1)
4.             return 1;  //Nếu Stack rỗng
5.       else
```

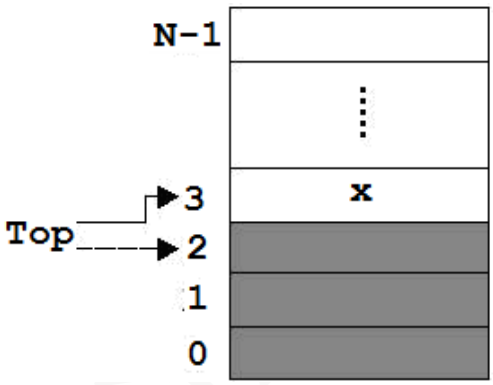
```
6.          return 0;  //Nếu Stack không rỗng
7.  }
```

4.1.4.2.3. Kiểm tra Stack đầy

```
1.  int isFull(Stack s)
2.  {//Kiểm tra Stack có đầy hay không
3.      if(s.Top >= MAXSIZE)
4.          return 1;
5.      else
6.          return 0;
7.  }
```

4.1.4.2.4. Thêm một phần tử mới vào Stack

Giả sử ta cần thêm giá trị x vào Stack.



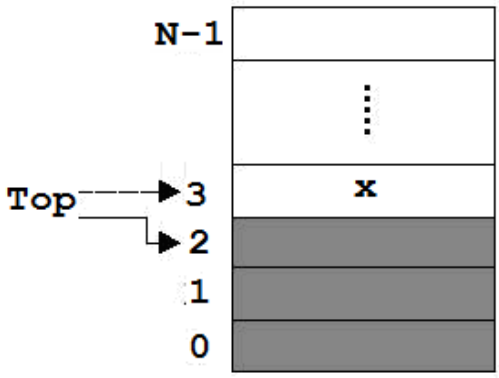
Hình 4.5: Hình vẽ minh họa thêm phần tử vào Stack

Hàm **push** trả về giá trị **0** nếu Stack bị đầy, hoặc trả về **1** nếu thêm phần tử mới vào Stack thành công.

```
1.  int push(Stack &s, ItemType x)
2.  {
3.      if(isFull(s) == 1)
4.          return 0; //Thực hiện không thành công
5.      s.Top++;
6.      s.Info[s.Top] = x;
7.      return 1; //Thực hiện thành công
8.  }
```

4.1.4.2.5. Lấy một phần tử ra khỏi Stack

Giả sử cần lấy giá trị x ở đỉnh ra khỏi Stack.



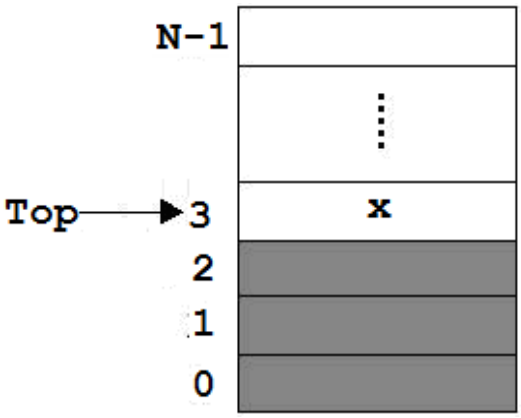
Hình 4.6: Hình vẽ minh họa lấy phần tử ở đỉnh Stack

Hàm **pop** trả về giá trị **0** nếu Stack rỗng, hoặc trả về giá trị **1** nếu lấy thành công phần tử ở đỉnh Stack.

```
1. int pop(Stack &s, ItemType &x)
2. {
3.       if(isEmpty(s) == 1)
4.             return 0; //Thực hiện không thành công
5.       x = s.Info[s.Top]; //Lấy thông tin của phần tử ở đỉnh Stack
6.       s.Top--;
7.       return 1; //Thực hiện thành công
8. }
```

4.1.4.2.6. Lấy giá trị của phần tử ở đỉnh Stack

Giả sử cần lấy giá trị x ở đỉnh của Stack mà không hủy nó.



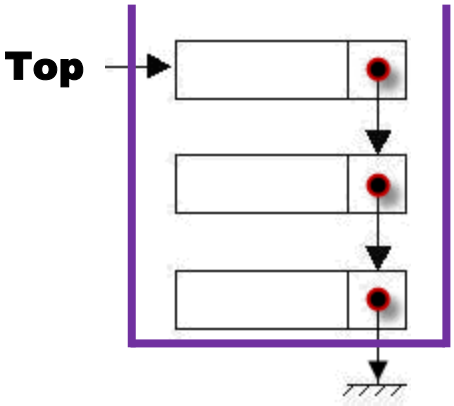
Hình 4.7: Hình vẽ minh họa lấy giá trị phần tử ở đỉnh Stack

Hàm **getTop** trả về giá trị **0** nếu Stack rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử ở đỉnh Stack.

```
1. int getTop(Stack s, ItemType &x)
2. {
3.       if(isEmpty(s) == 1)
4.             return 0; //Thực hiện không thành công
5.       x = s.Info[s.Top]; //Lấy thông tin của phần tử ở đỉnh Stack
6.       return 1; //Thực hiện thành công
7. }
```

4.1.5. Biểu diễn Stack bằng danh sách liên kết

4.1.5.1. Tổ chức dữ liệu



Hình 4.8: Hình vẽ minh họa tổ chức Stack bằng DSLK đơn

Stack là một danh sách liên kết được khai báo như sau:

```
1.  typedef int ItemType;
2.  struct StackNode
3.  {
4.      ItemType Info;
5.      StackNode* Next;
6.  };
7.  struct Stack
8.  {
9.      StackNode* Top;
10. };
```

4.1.5.2. Các thao tác trên Stack

4.1.5.2.1. Khởi tạo Stack

Sau khi khởi tạo thì Stack phải rỗng, nên con trỏ Top sẽ trỏ NULL.

Hình 4.9: Hình vẽ minh họa khởi tạo Stack rỗng

```
1.  void initStack(Stack &s)
2.  {
3.      s.Top = NULL;
4.  }
```

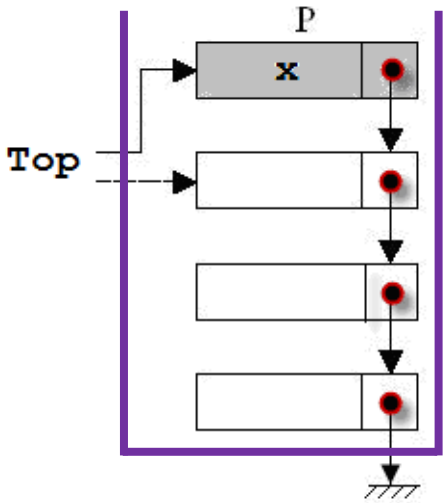
4.1.5.2.2. Kiểm tra tính rỗng của Stack

```
1.  int isEmpty(Stack s)
```

```
2.  {//Hàm kiểm tra Stack có rỗng hay không
3.      if(s.Top == NULL)
4.          return 1;  //Nếu Stack rỗng
5.      else
6.          return 0;  //Nếu Stack không rỗng
7.  }
```

4.1.5.2.3. Thêm một phần tử mới vào Stack

Giả sử ta cần thêm giá trị x vào Stack.



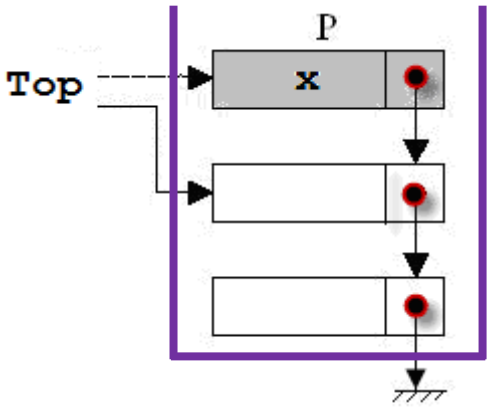
Hình 4.10: Hình vẽ minh họa thêm phần tử vào Stack

Hàm **push** tiến hành thêm phần tử mới p vào đầu Stack, nếu thành công hàm trả về **1** ngược lại hàm trả về **0**.

```
1.  int push(Stack &s, StackNode* p)
2.  {
3.      if(p == NULL)
4.          return 0;  //Thực hiện không thành công
5.      if(isEmpty(s) == 1)
6.          s.Top = p;
7.      else
8.      {
9.          p→Next = s.Top;
10.         s.Top = p;
11.     }
12.     return 1;  //Thực hiện thành công
13. }
```

4.1.5.2.4. Lấy một phần tử ra khỏi Stack

Giả sử cần lấy phần tử p có giá trị x ở đỉnh ra khỏi Stack.



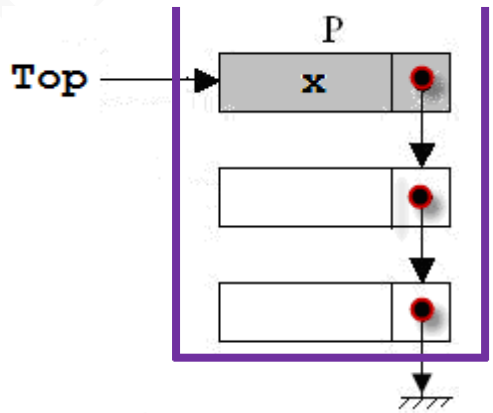
Hình 4.11: Hình vẽ minh họa lấy phần tử ở đầu Stack

Hàm **pop** trả về giá trị **0** nếu Stack rỗng, hoặc trả về giá trị **1** nếu lấy thành công phần tử p có giá trị x ở đỉnh ra khỏi Stack.

```
1.  int pop(Stack &s, ItemType &x)
2.  {
3.      if(isEmpty(s) == 1)   //Stack rỗng
4.          return 0;  //Thực hiện không thành công
5.      StackNode* p = s.Top;
6.      s.Top = s.Top→Next;
7.      x = p→Info;      //Lấy thông tin của nút ở đỉnh Stack
8.      delete p;   //Hủy nút do p trỏ đến
9.      return 1;   //Thực hiện thành công
10. }
```

4.1.5.2.5. Lấy giá trị của phần tử ở đỉnh của Stack

Giả sử cần lấy giá trị x của phần tử p ở đỉnh của Stack.



Hình 4.12: Hình vẽ minh họa lấy giá trị phần tử ở đầu Stack

Hàm **getTop** trả về giá trị **0** nếu Stack rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử p ở đỉnh Stack.

```
1.  int getTop(Stack s, ItemType &x)
2.  {
3.      if(isEmpty(s) == 1)   //Stack rỗng
```

```
4.              return 0;  //Thực hiện không thành công
5.          x = s.Top→Info;       //Lấy thông tin của nút ở đỉnh Stack
6.          return 1;  //Thực hiện thành công
7.  }
```

4.2. Hàng đợi - Queue

4.2.1. Định nghĩa

Hàng đợi là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh sách được thực hiện ở một đầu này và thao tác lấy ra một phần tử từ trong danh sách lại được thực hiện ở đầu kia.

Như vậy, các phần tử được đưa vào trong hàng đợi trước sẽ được lấy ra trước, phần tử đưa vào trong hàng đợi sau sẽ được lấy ra sau. Do đó mà hàng đợi còn được gọi là danh sách vào trước ra trước (FIFO List) và cấu trúc dữ liệu này còn được gọi là cấu trúc FIFO (First In – First Out).

Tương tự như ngăn xếp, có nhiều cách để biểu diễn và tổ chức các hàng đợi:

- Sử dụng danh sách đặc,
- Sử dụng danh sách liên kết.

Tuy nhiên, điều quan trọng và cần thiết là chúng ta phải quản lý vị trí hai đầu của hàng đợi thông qua hai biến: Biến trước (Head) và Biến sau (Tail). Hai biến này có thể cùng chiều hoặc ngược chiều với thứ tự các phần tử trong mảng và trong danh sách liên kết. Điều này có nghĩa là đầu hàng đợi có thể là đầu mảng, đầu danh sách liên kết mà cũng có thể là cuối mảng, cuối danh sách liên kết. Để thuận tiện, ở đây chúng ta giả sử đầu hàng đợi cũng là đầu mảng, đầu danh sách liên kết. Trường hợp ngược lại, sinh viên tự áp dụng tương tự.

Ở đây chúng ta sẽ biểu diễn và tổ chức hàng đợi bằng danh sách đặc và bằng danh sách liên kết đơn được quản lý bởi hai con trỏ đầu và cuối danh sách. Do vậy cấu trúc dữ liệu của hàng đợi cũng như các thao tác trên hàng đợi sẽ được trình bày thành hai trường hợp khác nhau.

4.2.2. Ứng dụng của Queue

Hàng đợi có nhiều ứng dụng:

- Khử đệ quy.
- Tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng.
- Tổ chức quản lý và phân phối tiến trình trong các hệ điều hành.
- Tổ chức bộ đệm bàn phím.

4.2.3. Các thao tác trên Queue

Queue chủ yếu dùng các thao tác sau:

- initQueue(&q): Khởi tạo Queue mới q.
- isEmpty(q): Kiểm tra Queue q có rỗng hay không.
- insertQueue(q, o): Thêm đối tượng o vào đuôi của Queue q.
- deleteQueue(q, x): Lấy đối tượng từ đầu của Queue q và gán giá trị của nó cho x.
- getHead(q, x): Lấy giá trị của đối tượng ở đầu của Queue q và gán giá trị đó cho x mà không hủy đối tượng này ra khỏi Queue q.
- getTail(q, x): Lấy giá trị của đối tượng ở cuối của Queue q và gán giá trị đó cho x mà không hủy đối tượng này ra khỏi Queue q.

4.2.4. Biểu diễn Queue dùng mảng

4.2.4.1. Tổ chức dữ liệu

Ta định nghĩa Queue q là một mảng chứa các phần tử.

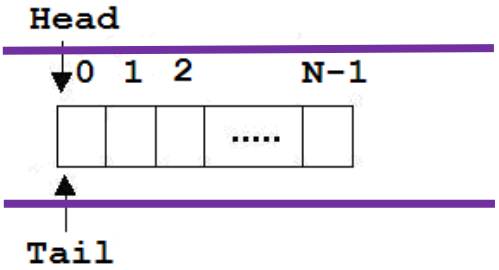
Hình 4.13: Hình vẽ minh họa tổ chức Queue bằng mảng một chiều

```
1. #define MAXSIZE 100
2. typedef int ItemType;
3. struct Queue
4. {
5.       ItemType Info[MAXSIZE];
6.       int n;      //Số phần tử hiện hành có trong Queue
7.       int Head;   //Chỉ số của phần tử đầu trong Queue
8.       int Tail;   //Chỉ số của phần tử cuối trong Queue
9. };
```

4.2.4.2. Các thao tác trên Queue

4.2.4.2.1. Khởi tạo Queue

Khi khởi tạo, Queue là rỗng, ta cho Head và Tail bằng **-1**.



Hình 4.14: Hình vẽ minh họa khởi tạo Queue rỗng

```
1. void initQueue(Queue &q)
2. {
3.       q.n = 0;
4.       q.Head = -1;
5.       q.Tail = -1;
6. }
```

4.2.4.2.2. Kiểm tra Queue rỗng

```
1. int isEmpty(Queue q) //Kiểm tra Queue có rỗng hay không?
2. { //Hàm trả về 1: nếu Queue rỗng, và 0: nếu Queue khác rỗng
3.       if(q.n == 0)
4.             return 1;  //Nếu Queue rỗng
5.       else
6.             return 0;  //Nếu Queue không rỗng
7. }
```

4.2.4.2.3. Kiểm tra Queue đầy

```
1. int isFull(Queue q) //Kiểm tra Queue có đầy thật hay không?
2. { //Hàm trả về 1: nếu Queue đầy, và 0: nếu Queue chưa đầy
3.       if(q.n == MAXSIZE)
4.             return 1;
5.       else
6.             return 0;
7. }
```

4.2.4.2.4. Thêm một phần tử mới vào cuối Queue

Giả sử ta cần thêm giá trị x vào Queue.

Hàm **insertQueue** trả về giá trị **0** nếu Queue bị đầy, hoặc trả về **1** nếu thêm phần tử mới vào Queue thành công.

```
1. int insertQueue (Queue &q, ItemType x)
2. {
3.       if(isFull(q) == 1)
4.             return 0; //Thực hiện không thành công
5.       q.Tail = (q.Tail + 1) % MAXSIZE;
6.       q.Info[q.Tail] = x;
7.       q.n++;
```

```
8.      return 1; //Thực hiện thành công
9.  }
```

Cách khác:

```
1.  int insertQueue (Queue &q, ItemType x)
2.  {
3.      int i, h, t;
4.      if(q.Tail-q.Head+1==MAXSIZE)      //Queue đầy nên không thể thêm
5.          return 0;  //Thực hiện không thành công
6.      if(q.Head == -1)
7.      {
8.          q.Head = 0;
9.          q.Tail = -1;
10.     }
11.     if(q.Tail == MAXSIZE - 1)     //Queue đầy ảo
12.     {
13.         h = q.Head;
14.         t = q.Tail;
15.         for(i = h; i <= t; i++)
16.             q.a[i - h] = q.a[i];
17.         q.Head = 0;
18.         q.Tail = t - h;
19.     }
20.     q.Tail++;
21.     q.a[q.Tail] = x;
22.     return 1;  //Thực hiện thành công
23. }
```

4.2.4.2.5. Lấy một phần tử ở đầu ra khỏi Queue

Giả sử cần lấy giá trị x ở đỉnh ra khỏi Stack.

Hàm deleteQueue trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công phần tử ở đầu Queue.

```
1.  int deleteQueue(Queue &q, ItemType &x)
2.  {
3.      if(isEmpty(q) == 1)
4.          return 0;  //Thực hiện không thành công
5.      q.Head = (q.Head + 1) % MAXSIZE;
6.      x = q.Info[q.Head];
7.      q.n--;
8.      return 1;  //Thực hiện thành công
9.  }
```

Cách khác:

```
1.  int deleteQueue(Queue &q, ItemType &x)
2.  {
3.      if(q.Head == -1)      //Queue rỗng
4.      {
```

```
5.              printf("Hàng đợi rỗng!");
6.              return 0;  //Thực hiện không thành công
7.      }
8.      x = q.Info[q.Head];    //Lấy thông tin của phần tử ở đầu Queue
9.      q.Head++;
10.     if(q.Head > q.Tail)     //Trường hợp có 1 phần tử
11.     {
12.             q.Head = -1;
13.             q.tail = -1;
14.     }
15.     return 1;  //Thực hiện thành công
16. }
```

4.2.4.2.6. Lấy giá trị phần tử ở đầu Queue

Giả sử cần lấy giá trị x ở đầu của Queue mà không hủy nó.

Hàm **getHead** trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử ở đầu Queue.

```
1.  int getHead(Queue q, ItemType &x)
2.  {
3.      if(isEmpty(q) == 1)  //Queue rỗng
4.      {
5.              printf("Queue rong!");
6.              return 0;  //Thực hiện không thành công
7.      }
8.      x = q.Info[q.Head];    //Lấy thông tin của phần tử ở đầu Queue
9.      return 1;  //Thực hiện thành công
10. }
```

4.2.4.2.7. Lấy giá trị phần tử ở cuối Queue

Giả sử cần lấy giá trị x ở cuối của Queue mà không hủy nó.

Hàm **getTail** trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử ở cuối Queue.

```
11. int getTail(Queue q, ItemType &x)
12. {
13.     if(isEmpty(q) == 1)  //Queue rỗng
14.     {
15.             printf("Queue rong!");
16.             return 0;  //Thực hiện không thành công
17.     }
18.     x = q.Info[q.Tail];     //Lấy thông tin của phần tử ở cuối Queue
19.     return 1;  //Thực hiện thành công
20. }
```

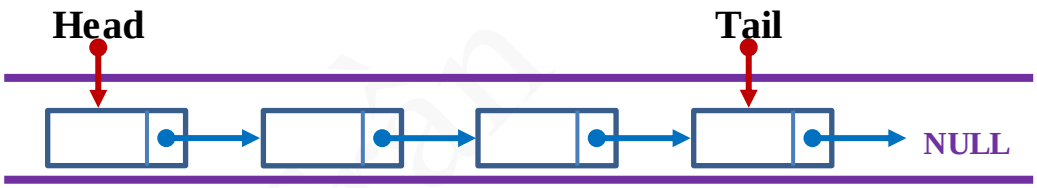
4.2.4.2.8. Xem nội dung của Queue

Xuất ra màn hình nội dung của hàng đợi để xem chứ không xóa bất kỳ phần tử nào của hàng đợi.

```
1.  void showQueue(Queue q)
2.  {
3.      if(isEmpty(q) == 1)
4.      {
5.          printf("\nhang doi rong!");
6.          return;
7.      }
8.      printf("\nNoi dung cua hang doi: ");
9.      for(int i = q.Head + 1; i != (q.Tail); i = (i+1) % MAXSIZE)
10.     {
11.         ItemType x = q.Info[i];
12.         printf("%4d", x);
13.     }
14.     printf("%4d", q.Info[q.Tail]);
15. }
```

4.2.5. Biểu diễn Queue bằng danh sách liên kết

4.2.5.1. Tổ chức dữ liệu



Hình 4.15: Hình vẽ minh họa tổ chức Queue bằng DSLK đơn

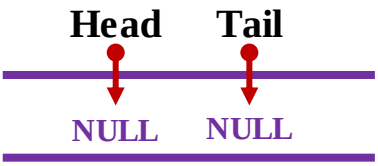
Queue là một danh sách liên kết được khai báo như sau:

```
1.  typedef int ItemType;
2.  struct QueueNode
3.  {
4.      ItemType Info;
5.      QueueNode* Next;
6.  };
7.  struct Queue
8.  {
9.      QueueNode* Head;
10.     QueueNode* Tail;
11. };
```

4.2.5.2. Các thao tác trên Queue

4.2.5.2.1. Khởi tạo Queue

Khi khởi tạo, Queue là rỗng, ta cho Head và Tail bằng NULL.



Hình 4.16: Hình vẽ minh họa khởi tạo Queue rỗng

```
1.  void initQueue(Queue &q)
2.  {
3.      q.Head = NULL;
4.      q.Tail = NULL;
5.  }
```

4.2.5.2.2. Kiểm tra tính rỗng của Queue

```
1.  int isEmpty(Queue q)      //Queue có rỗng hay không
2.  { //Hàm trả về 1: nếu Queue rỗng, và 0: nếu Queue khác rỗng
3.      if(q.Head == NULL)
4.          return 1;  //Nếu Queue rỗng
5.      else
6.          return 0;  //Nếu Queue không rỗng
7.  }
```

4.2.5.2.3. Thêm một phần tử mới vào Queue

Giả sử cần thêm giá trị x vào Queue.

Hàm **insertQueue** tiến hành thêm phần tử mới p vào cuối Queue, nếu thành công hàm trả về **1** ngược lại hàm trả về **0**.

```
1.  int insertQueue(Queue &q, QueueNode* p)
2.  {
3.      if(p == NULL)
4.          return 0;  //Thực hiện không thành công
5.      if(isEmpty(q) == 1)
6.      {
7.          q.Head = p;
8.          q.Tail = p;
9.      }
10.     else
11.     {
12.         q.Tail→Next = p;
13.         q.Tail = p;
14.     }
15.     return 1;  //Thực hiện thành công
16. }
```

4.2.5.2.4. Lấy một phần tử ở đầu ra khỏi Queue

Giả sử cần lấy phần tử p có giá trị x ở đầu ra khỏi Queue.

Hàm **deleteQueue** trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công phần tử p có giá trị x ở đỉnh Queue.

```
1.  int deleteQueue(Queue &q, ItemType &x)
2.  {
3.      if(isEmpty(q) == 1)
4.          return 0;  //Thực hiện không thành công
5.      QueueNode* p = q.Head;
6.      q.Head = q.Head→Next;
7.      if(q.Head == NULL)
8.          q.Tail = NULL;
9.      x = p→Info;      //Lấy thông tin của nút bị hủy
10.     delete p;  //Hủy nút do p trỏ đến
11.     return 1;  //Thực hiện thành công
12. }
```

4.2.5.2.5. Lấy giá trị của phần tử ở đầu của Queue

Giả sử cần lấy giá trị x của phần tử p ở đầu của Queue.

Hàm **getHead** trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử p ở đầu Queue.

```
1.  int getHead(Queue q, ItemType &x)
2.  {
3.      if(isEmpty(q) == 1)
4.          return 0;  //Thực hiện không thành công
5.      x = q.Head→Info;      //Lấy thông tin của nút ở đầu Queue
6.      return 1;  //Thực hiện thành công
7.  }
```

4.2.5.2.6. Lấy giá trị của phần tử ở cuối của Queue

Giả sử cần lấy giá trị x của phần tử p ở cuối của Queue.

Hàm **getTail** trả về giá trị **0** nếu Queue rỗng, hoặc trả về giá trị **1** nếu lấy thành công giá trị phần tử p ở cuối của Queue.

```
1.  int getTail(Queue q, ItemType &x)
2.  {
3.      if(isEmpty(q) == 1)
4.          return 0;  //Thực hiện không thành công
5.      x = q.Tail→Info;      //Lấy thông tin của nút ở cuối Queue
6.      return 1;  //Thực hiện thành công
7.  }
```

4.2.5.2.7. Xem nội dung của Queue

Xuất ra màn hình nội dung của hàng đợi để xem chứ không xóa bất kỳ phần tử nào của hàng đợi.

```
1.  void showQueue(Queue q)
2.  {
3.      if(isEmpty(q) == 1)
4.      {
5.          printf("\nHàng đợi rỗng!");
6.          return;     //Thực hiện không thành công
7.      }
8.      printf("\nNội dung hàng đợi là: ");
9.      for(QueueNode* p = q.Head; p != NULL; p = p→Next)
10.         printf("%4d", p→Info);
11. }
```

4.3. Câu hỏi và bài tập

4.3.1. Câu hỏi

1. Nắm được cơ chế hoạt động của Stack. Và các thao tác thêm một phần tử vào Stack và lấy một phần tử ra khỏi Stack.

   Ứng dụng vào việc lưu các kí tự của một chuỗi bằng Stack. Hãy cho biết nội dung của Stack sau mỗi thao tác trong dãy: "hell*o***w*orld**".

   Biết rằng: Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào Stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong Stack in lên màn hình.

2. Nắm được cơ chế hoạt động của Queue. Và các thao tác thêm một phần tử vào Queue và lấy một phần tử ra khỏi Queue.

   Ứng dụng vào việc lưu các kí tự của một chuỗi bằng Queue. Hãy cho biết nội dung của Queue sau mỗi thao tác trong dãy: "hell*o***w*orld**".

   Biết rằng: Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào Queue, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong Queue in lên màn hình.

4.3.2. Bài tập

3. Ứng dụng stack để viết các chương trình cơ bản sau:

   a. Đổi cơ số (từ hệ 10 sang hệ a, với $a \geq 2$).

   b. Đảo ngược 1 chuỗi ký tự bất kỳ.

   c. Khử đệ qui cho bài toán tính giai thừa.

   d. Khử đệ qui cho bài toán Tháp Hà Nội.

   e. Khử đệ qui giải thuật Quick Sort.

4. Cài đặt chương trình cho phép nhập vào một biểu thức gồm các số, các toán tử + , -, *, / và tính toán giá trị của biểu thức.
5. Mở rộng chương trình trên khi cho phép nhập vào một biểu thức gồm các số, các toán tử + , -, *, /, %, các hàm toán học sin, cos, tan, ln, ex, dấu mở đóng ngoặc "(", ")" và tính toán giá trị của biểu thức.
6. Kiểm tra cặp ngoặc:

   Mỗi dấu "(", "{", or "[" đều phải có một dấu đóng tương ứng ")", "}", or "]"

   a. Đúng: ( )(( )){([( )])}

   b. Sai:)(( ))

   c. Sai: ({[ ])}

   Viết giải thuật nhận một xâu (*danh sách*) đầu vào gồm các ký tự mở, đóng ngoặc. Kiểm tra xâu có hợp lệ không?
7. Dựa trên bài toán người sản xuất (NSX)- người tiêu dùng (NTD). NSX sản xuất ra hàng hóa và chất vào kho thứ tự, NTD lấy hàng hóa ra theo thứ tự của queue, cái gì chất vào trước thì tiêu thụ trước (nếu không nó hỏng). Hãy mô phỏng bài toán với các thao tác sau:
   a. Nhập một danh sách n (n>0) mặt hàng vào kho.
   b. Xem thông tin tất cả hàng hóa trong kho.
   c. Xem thông tin mặt hàng chuẩn bị được xuất kho.
   d. Xuất khỏi kho một mặt hàng và cho xem thông tin của mặt hàng đó.
   e. Xem thông tin mặt hàng mới vừa nhập vào kho.
   f. Tìm và xem thông tin của một mặt hàng bất kỳ trong kho.
   g. Xuất toàn bộ hàng hóa trong kho.
8. Mô phỏng việc xếp hàng mua vé tàu (xe lửa).

Chương 5. CÂY

5.1. Giới thiệu

Do nhược điểm về tốc độ tìm kiếm của danh sách liên kết nên người ta đã tìm kiếm các cấu trúc dữ liệu khác hiệu quả hơn. Cây là một trong những phương án cho bài toán này. Do đó trong chương này chúng ta sẽ xem xét cấu trúc dữ liệu cây.

Phần đầu của chương trình bày các khái niệm liên quan đến cây nhị phân, các thao tác trên cấu trúc dữ liệu này. Phần tiếp theo, cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân. Đây là một cấu trúc dữ liệu cho phép tìm kiếm hiệu quả hơn hẳn danh sách liên kết mà các thao tác không phức tạp nhiều nên cây nhị phân tìm kiếm là một cấu trúc dữ liệu thông dụng.

Trong một số trường hợp cây nhị phân tìm kiếm bị suy biến thành danh sách liên kết. Điều này làm cho cây nhị phân cân bằng (AVL) ra đời. Cấu trúc cây nhị phân cân bằng đảm bảo thao tác tìm kiếm chỉ tốn chi phí $\log_2(n)$ với n là số nút trên cây. Nhưng nhược điểm là thao tác cân bằng lại mỗi khi cây mất cân bằng. Việc thêm hay hủy phần tử đều có thể làm cây mất cân bằng, việc cân bằng lại có thể lan truyền lên tận gốc.

5.2. Giới thiệu về cây

Cây là một cấu trúc rất quan trọng và được dùng rất nhiều trong các giải thuật. Trong chương này ta sẽ tìm hiểu các khái niệm cơ bản về cây, các phép toán quan trọng trên cây, biểu diễn cây trên máy tính. Cây có nhiều ứng dụng trong đời sống hàng ngày, chẳng hạn như tổ chức các quan hệ họ hàng trong gia phả, mục lục của một sách, …

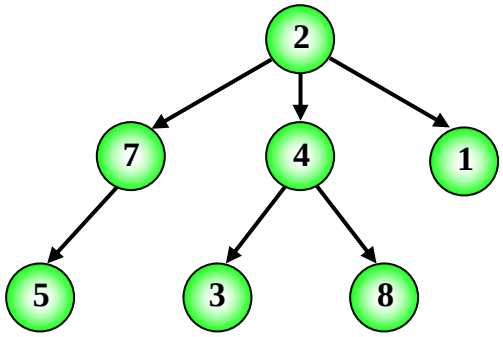
Chúng ta đã tìm hiểu về giải thuật tìm kiếm tuyến tính và tìm kiếm nhị phân. Đối với giải thuật tìm kiếm tuyến tính ta có thể đưa vào danh sách liên kết, nhưng giải thuật tìm kiếm nhị phân thì không thể. Trong phần này, chúng ta xét đến cây nhị phân tìm kiếm, mà việc tìm kiếm nhị phân có thể thực hiện trên đó, là một dạng của danh sách liên kết.

5.3. Cấu trúc cây

5.3.1. Định nghĩa cây

Cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có một nút đặc biệt được gọi là nút gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2, …, T_n theo quan hệ phân cấp trong đó T_i cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i + 1. Quan hệ này được gọi là quan hệ cha - con và có duy nhất một đường đi từ gốc đến nút con.

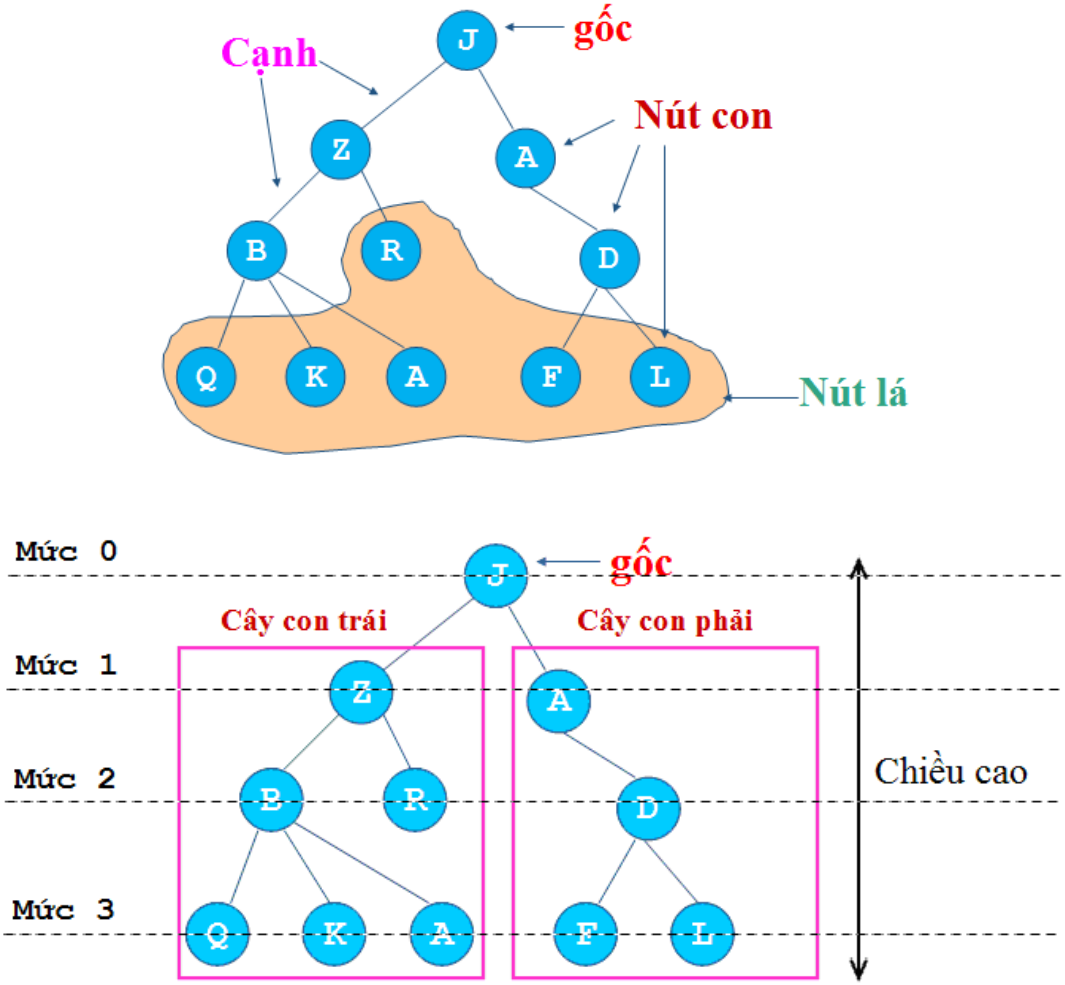
Ví dụ: Giả sử có một cây như sau:



Hình 5.1: Ví dụ minh họa cây

5.3.2. Một số khái niệm cơ bản

- *Bậc của một nút:* Là số cây con của nút đó (nút lá có bậc bằng 0).
- *Bậc của một cây:* Là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút trong cây). Cây có bậc n thì gọi là cây n - phân.
- *Nút gốc:* Không có nút cha.
- *Nút lá:* Không có nút con hay nút có bậc bằng 0.
- *Nút trong (nút nhánh):* Không phải nút lá và nút gốc hay nút có bậc khác 0.
- *Nút trước và nút sau của một nút:* Nút T được gọi là nút trước của nút S nếu cây con có gốc là T chứa cây con có gốc là S. Khi đó, nút S được gọi là nút sau của nút T.
- *Nút cha và nút con của một nút:* Nút B được gọi là nút cha của nút C nếu nút B là nút trước của nút C và mức của nút C lớn hơn mức của nút B là 1 mức. Khi đó, nút C được gọi là nút con của nút B.
- *Nút anh em (nút đồng cấp):* Các nút có cùng một nút cha.
- *Chiều dài đường đi của một nút:* Là số đỉnh (số nút) tính từ nút gốc để đi đến nút đó. Như vậy, chiều dài đường đi của nút gốc luôn luôn bằng 1, chiều dài đường đi tới một nút bằng chiều dài đường đi tới nút cha nó cộng thêm 1.
- *Độ dài đường đi của một cây:* Được định nghĩa là tổng các độ dài đường đi của tất cả các nút của cây.
- *Mức của một nút:* Là độ dài đường đi từ gốc đến nút đó.
- *Chiều cao của một nút:* Là mức của nút đó cộng thêm 1.
- *Chiều cao của một cây:* Là chiều cao lớn nhất của các nút trong cây.
- *Rừng:* Là tập hợp các cây. Như vậy, nếu một cây bị loại bỏ nút gốc có thể cho ta một rừng.



Hình 5.2: Hình vẽ minh họa các khái niệm liên quan đến cây

5.3.3. Nhận xét

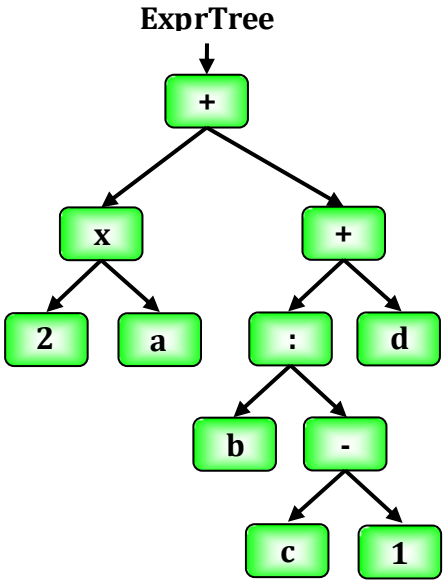
- Trong cấu trúc cây không tồn tại chu trình.
- Tổ chức một cấu trúc cây cho phép truy cập nhanh đến các phần tử của nó.

5.4. Cây nhị phân

5.4.1. Định nghĩa

Cây nhị phân là cây có bậc bằng hai (bậc của mỗi nút tối đa bằng 2).

Ví dụ: Cây nhị phân biểu diễn biểu thức (2 × a) + [b: (c – 1) + d] như sau:



Hình 5.3: Ví dụ cây nhị phân biểu diễn biểu thức

5.4.2. Biểu diễn cây nhị phân

Để biểu diễn cây nhị phân trong bộ nhớ máy tính chúng ta có thể sử dụng danh sách có hai mối liên kết để quản lý địa chỉ của hai nút gốc cây con (cây con trái và cây con phải). Như vậy cấu trúc dữ liệu của cây nhị phân tương tự như cấu trúc dữ liệu của danh sách liên kết đôi nhưng về cách thức liên kết thì khác nhau.

Mỗi nút (phần tử) của cây nhị phân ứng với một biến động gồm ba thành phần:

- Thông tin (dữ liệu) lưu trữ tại nút: **Info**.
- Địa chỉ nút gốc của cây con trái trong bộ nhớ: **Left**.
- Địa chỉ nút gốc của cây con phải trong bộ nhớ: **Right**.


```
1.  typedef <Kiểu dữ liệu cơ bản hoặc có cấu trúc> ItemType;
2.  struct TNode
3.  {
4.      ItemType Info; //Thông tin (dữ liệu) của nút
5.      TNode* Left;   //Con trỏ đến nút con bên trái
6.      TNode* Right;  //Con trỏ đến nút con bên phải
7.  };
8.  struct BTree
9.  { //Định nghĩa kiểu dữ liệu của cây nhị phân
10.     TNode* Root;
11. };
```

Do tính chất mềm dẻo của cách biểu diễn bằng cấp phát liên kết động, phương pháp này được dùng chủ yếu trong cây nhị phân. Từ phần này trở đi, khi nói đến cây nhị phân, chúng ta sẽ dùng phương pháp này.

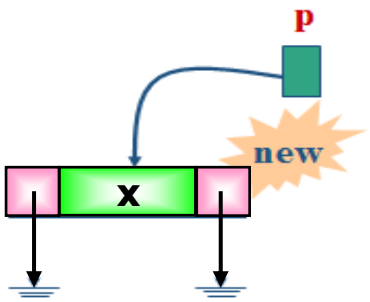
5.4.3. Các thao tác trên cây nhị phân

5.4.3.1. Khởi tạo cây nhị phân rỗng

Việc khởi tạo cây nhị phân chỉ đơn giản là khởi gán cho con trỏ quản lý địa chỉ nút gốc một giá trị NULL. Hàm khởi tạo cây nhị phân như sau:

```
1. void initBTree(BTree &bt)
2. {
3.       bt.Root = NULL;
4. }
```

5.4.3.2. Tạo nút chứa giá trị x



Hình 5.4: Hình vẽ minh họa tạo một nút mới cho cây

Hàm tạo mới một nút có thành phần dữ liệu là x, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo thì hàm sẽ trả về con trỏ NULL.

```
1. TNode* createTNode(ItemType x)
2. {
3.       TNode* p = new TNode;
4.       if(p == NULL)
5.       {
6.             printf("Không đủ bộ nhớ để cấp phát nút mới!");
7.             getch();
8.             return NULL;
9.       }
10.      p→Info = x;
11.      p→Left = NULL;
12.      p→Right = NULL;
13.      return p;
14. }
```

5.4.3.3. Kiểm tra cây nhị phân rỗng

Kiểm tra cây nhị phân đã có phần tử nào hay chưa? Nghĩa là kiểm tra con trỏ Root có bằng NULL hay không? Hàm trả sẽ về 1 nếu cây nhị phân chưa có phần tử nào (*rỗng*), ngược lại (*không rỗng*) thì hàm sẽ trả về 0.

```
1. int isEmpty(BTree bt)
2. {
3.       if(bt.Root == NULL)
4.             return 1;  //Nếu cây rỗng
5.       else
6.             return 0;  //Nếu cây không rỗng
7. }
```

5.4.3.4. Duyệt qua các nút trên cây nhị phân

Trong thao tác này nhằm tìm cách duyệt qua (ghé thăm) tất cả các nút trong cây nhị phân để thực hiện một thao tác xử lý nào đó đối với mỗi nút (*như xem nội dung thành phần dữ liệu chẳng hạn*). Căn cứ vào thứ tự duyệt nút gốc so với hai nút gốc của hai cây con, thao tác duyệt có thể thực hiện theo một trong ba thứ tự sau:

- Duyệt theo thứ tự nút gốc trước (*Preorder*): Theo cách duyệt này thì nút gốc sẽ được duyệt trước sau đó mới duyệt đến hai cây con. Căn cứ vào thứ tự duyệt hai cây con mà chúng ta có hai cách duyệt theo thứ tự nút gốc trước:
  + Duyệt nút gốc, duyệt cây con trái, duyệt cây con phải (*Root - Left - Right*).
  + Duyệt nút gốc, duyệt cây con phải, duyệt cây con trái (*Root - Right - Left*).

- Duyệt theo thứ tự nút gốc giữa (*Inorder*): Theo cách duyệt này thì chúng ta duyệt một trong hai cây con trước rồi đến duyệt nút gốc và sau đó mới duyệt cây con còn lại. Căn cứ vào thứ tự duyệt hai cây con chúng ta cũng sẽ có hai cách duyệt theo thứ tự nút gốc giữa:
  + Duyệt cây con trái, duyệt nút gốc, duyệt cây con phải (*Left - Root - Right*).
  + Duyệt cây con phải, duyệt nút gốc, duyệt cây con trái (*Right - Root - Left*).

- Duyệt theo thứ tự nút gốc sau (*Postorder*): Tương tự như duyệt theo nút gốc trước, trong cách duyệt này thì nút gốc sẽ được duyệt sau cùng so với duyệt hai nút gốc của hai cây con. Do vậy, căn cứ vào thứ tự duyệt hai cây con mà chúng ta cũng có hai cách duyệt theo thứ tự nút gốc sau:
  + Duyệt cây con trái, duyệt cây con phải, duyệt nút gốc (*Left - Right - Root*).
  + Duyệt cây con phải, duyệt cây con trái, duyệt nút gốc (*Right - Left - Root*).

Giáo trình chỉ trình bày ba cách duyệt sau: Node - Left - Right, Left - Node - Right, Left - Right - Node và sử dụng giải thuật đệ quy. Các cách duyệt khác (*bằng giải thuật đệ quy hay không đệ quy*) sinh viên tự vận dụng tương tự.

5.4.3.4.1. Duyệt theo thứ tự nút gốc trước (Node - Left - Right)

Kiểu duyệt này trước tiên thăm nút gốc, sau đó thăm các nút của cây con trái, rồi cuối cùng thăm các nút của cây con phải. Hàm duyệt có thể trình bày đơn giản như sau:

```
1. void traverseNLR(TNode* root)
2. {
3.     if(root == NULL) return;
4.     printf("%4d", root→Info);   //Xử lý thông tin của nút gốc
5.     traverseNLR(root→Left);
6.     traverseNLR(root→Right);
7. }
```

5.4.3.4.2. Duyệt theo thứ tự nút gốc giữa (Left - Node - Right)

Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm nút gốc, rồi cuối cùng thăm các nút của cây con phải. Hàm duyệt có thể trình bày đơn giản như sau:

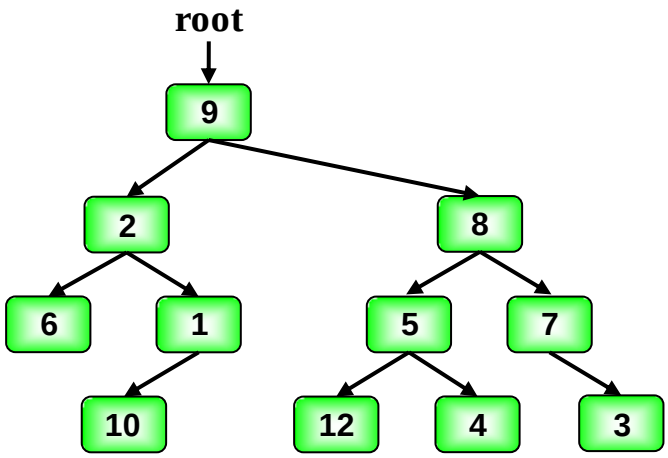
```
1. void traverseLNR(TNode* root)
2. {
3.     if(root == NULL) return;
4.     traverseLNR(root→Left);
5.     printf("%4d", root→Info);   //Xử lý thông tin của nút gốc
6.     traverseLNR(root→Right);
7. }
```

5.4.3.4.3. Duyệt theo thứ tự nút gốc sau (Left - Right - Node)

Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm các nút của cây con phải, rồi cuối cùng thăm nút gốc. Hàm duyệt có thể trình bày đơn giản như sau:

```
1. void traverseLRN(TNode* root)
2. {
3.     if(root == NULL) return;
4.     traverseLRN(root→Left);
5.     traverseLRN(root→Right);
6.     printf("%4d", root→Info);   //Xử lý thông tin của nút gốc
7. }
```

Ví dụ: Hãy cho biết kết quả của các phép duyệt của cây nhị phân sau:



Hình 5.5: Ví dụ về các phép duyệt cây

Kết quả các phép duyệt cây là:

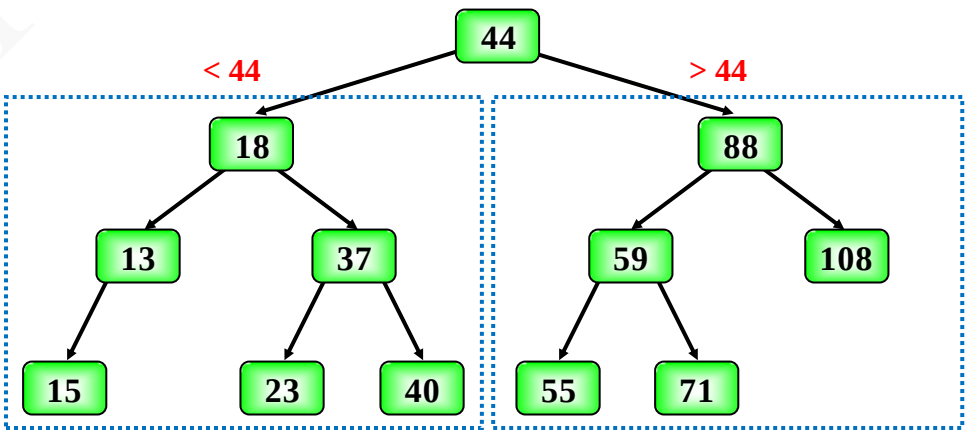
- traverseNLR: 9, 2, 6, 1, 10, 8, 5, 12, 4, 7, 3.
- traverseLNR: 6, 2, 10, 1, 9, 12, 5, 4, 8, 7, 3.
- traverseLRN: 6, 10, 1, 2, 12, 4, 5, 3, 7, 8, 9.
- traverseNRL, traverseRNL, traverseRLN? (*Sinh viên tự làm xem như bài tập*)

5.5. Cây nhị phân tìm kiếm

5.5.1. Định nghĩa

Cây nhị phân tìm kiếm là cây nhị phân có thành phần khóa của mỗi nút lớn hơn thành phần khóa của tất cả các nút trong cây con trái và nhỏ hơn thành phần khóa của tất cả các nút trong cây con phải của nó.

Ví dụ: Hình ảnh sau minh họa một cây nhị phân tìm kiếm.



Hình 5.6: Ví dụ về cây nhị phân tìm kiếm

Khóa nhận diện (*Info*) của các nút trong cây nhị phân tìm kiếm đôi một khác nhau (*không có hiện tượng trùng khóa*).

Nút ở bên trái nhất là nút có giá trị khóa nhận diện nhỏ nhất và nút ở bên phải nhất là nút có giá trị khóa nhận diện lớn nhất trong cây nhị phân tìm kiếm.

Trong một cây nhị phân tìm kiếm thứ tự duyệt cây Left - Node - Right là thứ tự duyệt theo sự tăng dần các giá trị của Info trong các nút và thứ tự duyệt cây Right - Node - Left là thứ tự duyệt theo sự giảm dần các giá trị của Key trong các nút.

Nhờ ràng buộc về khóa trên Cây nhị phân tìm kiếm, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây, việc tìm kiếm trở nên nhanh đáng kể. Nếu số nút trên cây là **n** thì chi phí tìm kiếm trung bình chỉ khoảng $\mathbf{log_2 n}$.

Trong thực tế, khi xét đến cây nhị phân chủ yếu ta xét Cây nhị phân tìm kiếm.

5.5.2. Các thao tác trên cây nhị phân tìm kiếm

Do cây nhị phân tìm kiếm là cây nhị phân nên những thao tác vừa trình bày ở trên được hiển nhiên áp dụng mà không cần phải trình bày lại. Trong phần này tác giả chỉ trình bày thêm một số thao tác như sau:

5.5.2.1. Tìm kiếm một phần tử x trong cây

Giả sử cần tìm trên cây nhị phân tìm kiếm xem có tồn tại nút có khóa Info bằng với x hay không?

Để thực hiện thao tác này chúng ta sẽ vận dụng giải thuật tìm kiếm nhị phân: Do đặc điểm của cây nhị phân tìm kiếm thì tại một nút, nếu Info của nút này khác với x thì x chỉ có thể tìm thấy như sau:

- Hoặc trên cây con trái của nút này nếu x nhỏ hơn Info của nó.
- Hoặc trên cây con phải của nút này nếu x lớn hơn Info của nó.

Cách 1: Viết hàm dạng đệ quy

```
1.  TNode* findTNode(TNode* root, ItemType x)
2.  {
3.      if(root == NULL)
4.          return NULL;
5.      if(root→Info == x)
6.          return root;      //tìm được khóa thì dừng
7.      else if(root→Info > x)
8.          return findTNode(root→Left, x);
9.      else
10.         return findTNode(root→Right, x);
11. }
```

Cách 2: Viết hàm dạng không đệ quy

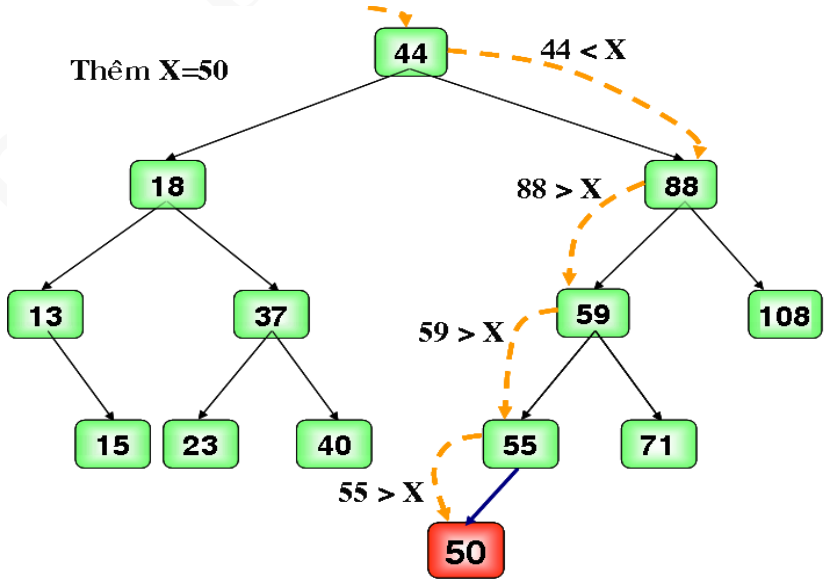
```
1.  TNode* findTNode(TNode* root, ItemType x)
2.  {
3.      TNode* p = root;
4.      while(p != NULL)
5.      {
6.          if(p→Info == x)
7.              return p;
8.          else if(p→Info > x)
9.              p = p→Left;
10.         else
11.             p = p→Right;
12.     }
13.     return NULL;
14. }
```

5.5.2.2. Thêm một phần tử x vào cây

Giả sử cần thêm một nút có thành phần dữ liệu (**Info**) là x vào trong cây nhị phân tìm kiếm sao cho sau khi thêm cây vẫn là một cây nhị phân tìm kiếm. Trong thao tác này trước hết phải tìm kiếm vị trí cho phép thêm, sau đó mới tiến hành thêm nút mới vào cây (do vậy giải thuật còn được gọi là giải thuật tìm kiếm và thêm vào cây). Quá trình tìm kiếm tuân thủ các bước trong giải thuật tìm kiếm đã trình bày ở trên.

Giải thuật này chỉ trình bày thao tác thêm vào cây nhị phân tìm kiếm trong trường hợp không có hiện tượng trùng lặp khóa. Vì vậy, nếu x bị trùng với Info của một trong các nút đã có trong cây nhị phân tìm kiếm thì sẽ không thực hiện thao tác thêm nữa.

Ví dụ sau minh họa khi thêm nút có giá trị X = 50 vào cây nhị phân tìm kiếm:



Hình 5.7: Ví dụ thêm một nút vào cây nhị phân tìm kiếm

Cách 1: Viết hàm đệ quy

```
1.  int insertTNode(TNode* &root, ItemType x)
2.  {
3.      if(root == NULL)      //Cây rỗng
4.      {
5.          root = createTNode(x);
6.          return 0;  //Thực hiện không thành công
7.      }
8.      if(root→Info == x)
9.          return 0;  //Bị trùng giá trị nút
10.     if(x < root→Info)
11.         insertTNode(root→Left, x);    //Thêm bên trái
12.     else
13.         insertTNode(root→Right, x);  //Thêm bên phải
14.     return 1;  //Thực hiện thành công
15. }
```

Hoặc nếu truyền vào hàm là con trỏ nút, thì hàm được định nghĩa lại như sau:

```
1.  int insertTNode(TNode* &root, TNode* p)
2.  {
3.      if(p == NULL)
4.          return 0;  //Thực hiện không thành công
5.      if(root == NULL)      //Cây rỗng, nên thêm vào gốc
6.      {
7.          root = p;
8.          return 1;  //Thực hiện thành công
9.      }
10.     if(root→Info == p→Info)
11.         return 0;  //Bị trùng giá trị nút
12.     if(p→Info < root→Info)
13.         insertTNode(root→Left, p);    //Thêm bên trái
14.     else
15.         insertTNode(root→Right, p);  //Thêm bên phải
16.     return 1;  //Thực hiện thành công
17. }
```

Cách 2: Viết hàm không đệ quy

```
1.  int insertTNodeNoRecursion(TNode* &root, ItemType x)
2.  {
3.      TNode *p, *q, *r;
4.      p = createTNode(x);
5.      if(root == NULL)      //Cây rỗng, nên thêm vào gốc
6.      {
7.          root = p;
8.          return 1;  //Thực hiện thành công
9.      }
10.     q = root;
11.     r = NULL;
```

```
12.     while(q != NULL)
13.     {
14.         if(q→Info == x) //nút cần thêm đã có trong cây
15.             return 0;   //Thực hiện không thành công
16.         r = q;
17.         if(q→Info > x)
18.             q = q→Left;
19.         else
20.             q = q→Right;
21.     }
22.     if(r→Info > x)
23.         r→Left = p;
24.     else
25.         r→Right = p;
26.     return 1;  //Thực hiện thành công
27. }
```

5.5.2.3. Tạo cây từ mảng a có sẵn dữ liệu

Giả sử ta đã có một mảng a chứa n phần tử, ta có thể tạo một cây nhị phân tìm kiếm từ mảng này.

```
1.  void createBTreeFromArray(BTree  &bt, ItemType a[ ], int n)
2.  {//Ham tao cay NPTK tu mang a
3.      initBTree(bt);
4.      for(int i = 0; i < n; i++)
5.      {
6.          TNode* p = createTNode(a[i]);
7.          insertTNode(bt.Root, p);
8.      }
9.  }
```

5.5.2.4. Tạo cây tự động chứa n số nguyên

Tạo một cây nhị phân tìm kiếm bằng cách phát sinh ngẫu nhiên n số nguyên và lần lượt thêm chúng vào cây.

```
1.  void createAutomaticBTree(BTree &bt)
2.  {
3.      int n;
4.      ItemType x;
5.      printf("Cho biet so nut cua cay: ");
6.      scanf("%d", &n);
7.      initBTree(bt);
8.      srand((unsigned)time(NULL));      //Tạo số mới sau mỗi lần thực hiện
9.      for(int i = 1; i <= n; i++)
10.     {
11.         x = (rand() % 199) - 99;      //Tạo một số ngẫu nhiêu từ -99→99
12.         TNode* p = createTNode(x);
13.         insertTNode(bt.Root, p);
```

```
14.     }
15.     printf("\nDa tao cay thanh cong!");
16. }
```

5.5.2.5. Xóa một phần tử x trong cây

Cũng như thao tác thêm một nút vào trong cây nhị phân tìm kiếm, thao tác hủy một nút trên cây nhị phân tìm kiếm cũng phải bảo đảm cho cây sau khi hủy nút đó thì cây vẫn là một cây nhị phân tìm kiếm. Đây là một thao tác không đơn giản bởi nếu không cẩn thận thì sau khi hủy nút, cây có thể sẽ biến cây thành một rừng.

Giả sử cần hủy nút có thành phần dữ liệu (**Info**) là x ra khỏi cây nhị phân tìm kiếm. Điều đầu tiên trong thao tác này là phải tìm kiếm địa chỉ của nút cần hủy là DelTNode, sau đó mới tiến hành hủy nút có địa chỉ là DelTNode này nếu tìm thấy (*do vậy giải thuật này còn được gọi là giải thuật tìm kiếm và loại bỏ trên cây*). Quá trình tìm kiếm đã trình bày ở trên, phần này chỉ trình bày thao tác hủy khi tìm thấy nút có địa chỉ DelTNode (DelTNode→Info == x) và trong quá trình tìm kiếm sẽ phải giữ địa chỉ nút cha của nút cần hủy là FatherDelTNode.

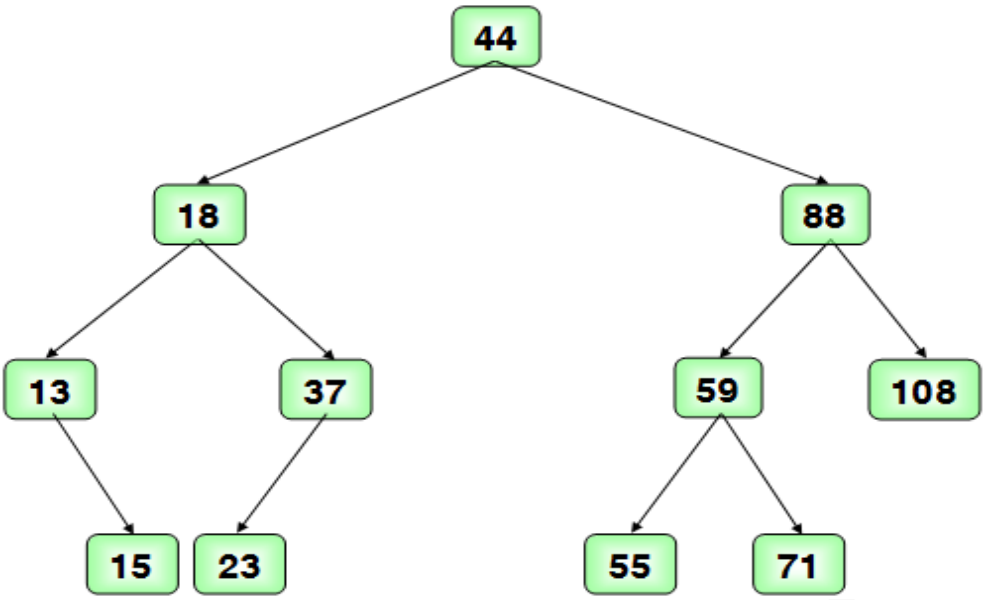
Việc hủy nút có địa chỉ DelTNode có thể xảy ra một trong ba trường hợp sau:

- ***DelTNode là nút lá:*** Trong trường hợp này đơn giản chỉ cần cắt bỏ mối quan hệ cha - con giữa FatherDelTNode và DelTNode bằng cách cho con trỏ FatherDelTNode→Left (nếu DelTNode là nút con bên trái của FatherDelTNode) hoặc cho con trỏ FatherDelTNode→Right (nếu DelTNode là nút con bên phải của FatherDelTNode) về con trỏ NULL và tiến hành hủy (delete) nút có địa chỉ DelTNode này.

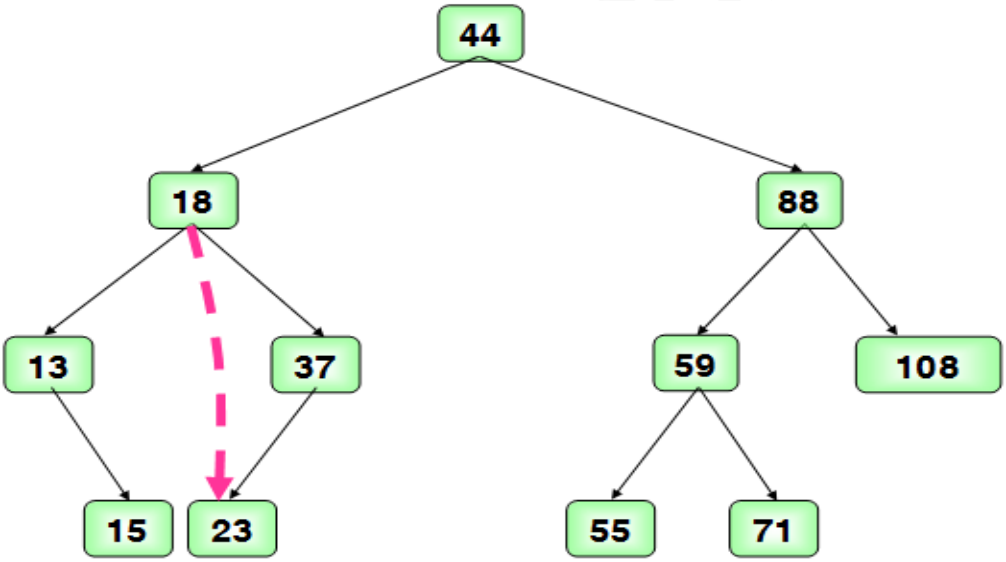
- ***DelTNode chỉ có một nút con:*** Trong trường hợp này cũng khá đơn giản chỉ cần chuyển mối quan hệ cha - con giữa FatherDelTNode và DelTNode thành mối quan hệ cha - con giữa FatherDelTNode và nút gốc cây con của DelTNode rồi tiến hành cắt bỏ mối quan hệ cha - con giữa DelTNode và một nút gốc cây con của nó và tiến hành hủy nút có địa chỉ DelTNode này.

- ***DelTNode có đủ hai nút con:*** Trong trường hợp này sẽ không hủy nút có địa chỉ DelTNode mà hủy nút có địa chỉ của phần tử thế mạng là nút phải nhất trong cây con trái của DelTNode, hoặc là nút trái nhất trong cây con phải của DelTNode. Sau khi chuyển toàn bộ nội dung dữ liệu của nút thế mạng cho DelTNode thì mới tiến hành hủy nút thế mạng.

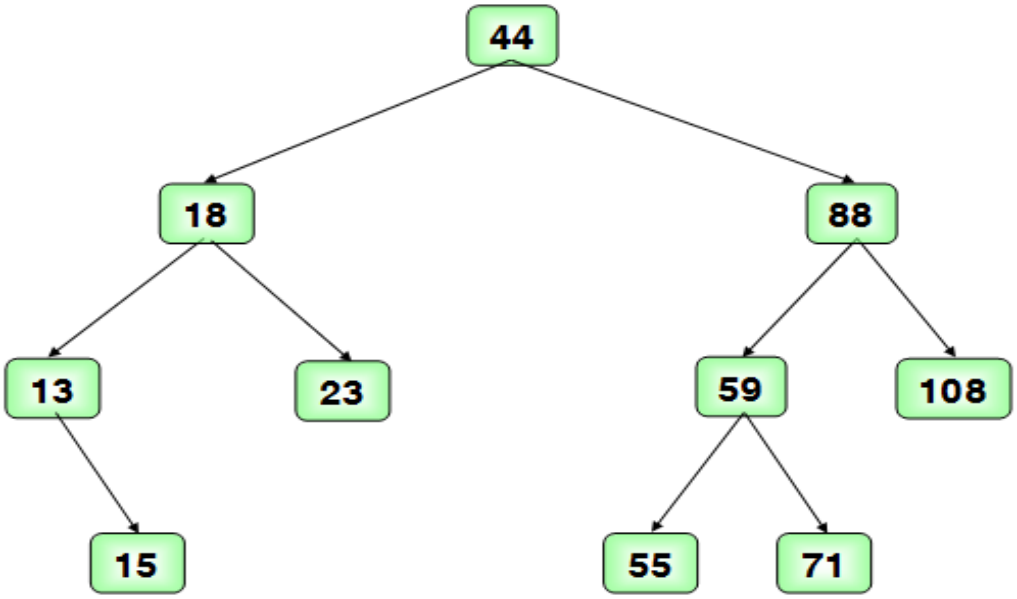
Ví dụ 1: Minh họa cho trường hợp phần tử bị xóa có một cây con



Xóa nút có giá trị x = 37 trong cây.

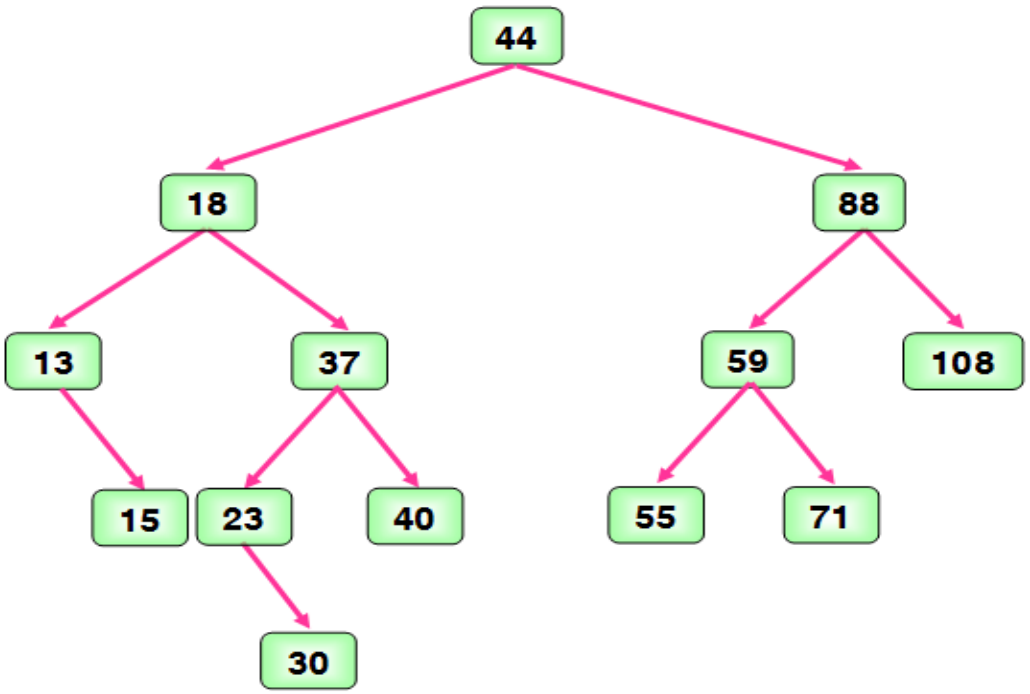


Sau khi xóa nút có giá trị x thì cây sẽ là:

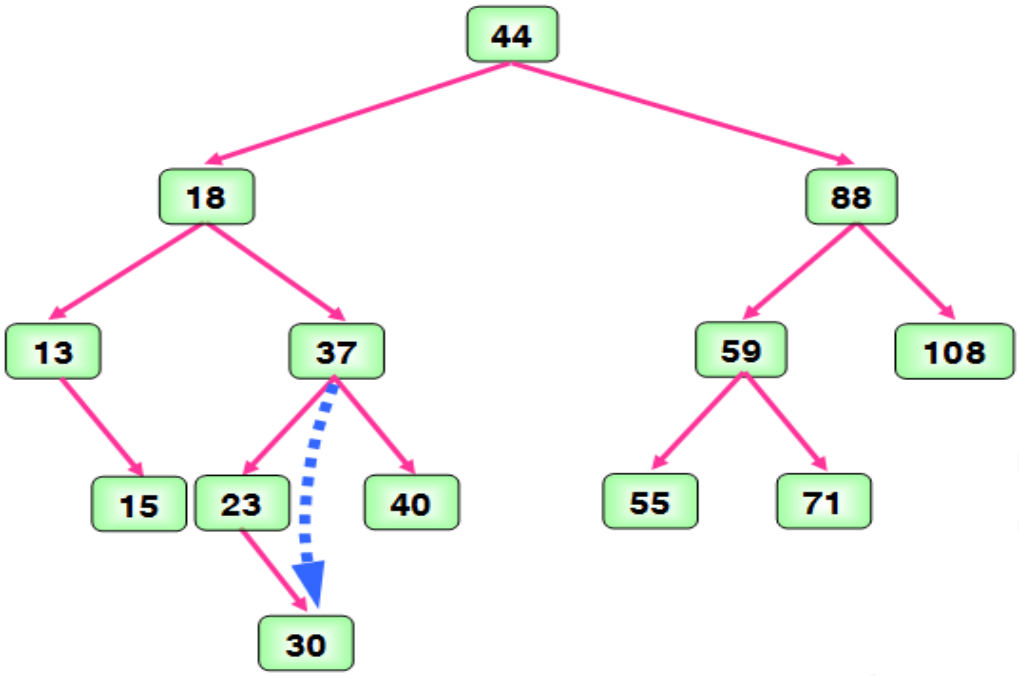


Hình 5.8: Ví dụ xóa nút chỉ có một cây con từ cây nhị phân tìm kiếm

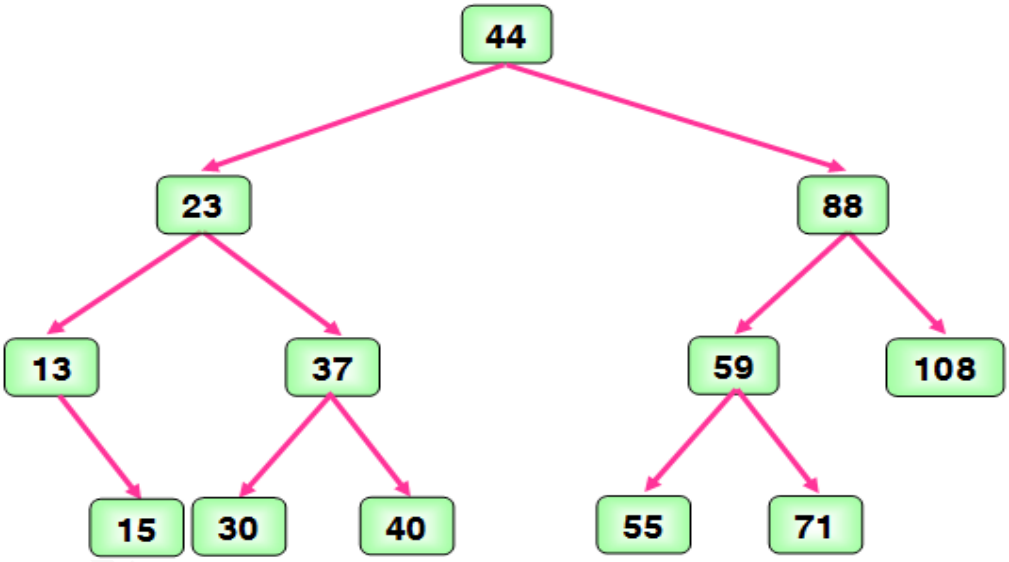
Ví dụ 2: Minh họa cho trường hợp phần tử bị xóa có hai cây con



Xóa nút có giá trị x = 18, lúc đó nút có giá trị (khóa) bằng 23 sẽ là nút thế mạng (là *nút nhỏ nhất của cây con phải*).



Sau khi xóa nút có giá trị x thì cây sẽ là:



Hình 5.9: Ví dụ xóa nút có hai cây con từ cây nhị phân tìm kiếm

Giải thuật:

- **Bước 1:** Tìm phần tử p có khóa x.
- **Bước 2:** Có 3 trường hợp:
  + p là nút lá: Hủy p.
  + p có 1 con: Tạo liên kết từ phần tử cha của p đến con của p, rồi hủy p.
  + p có 2 con: Tìm phần tử thế mạng cho p là 1 trong 2 phần tử sau:

- "Phần tử thế mạng là phần tử phải nhất của cây con trái của p (*phần tử lớn nhất trong các phần tử nhỏ hơn p*)".
- "Phần tử thế mạng là phần tử trái nhất của cây con phải của p (*phần tử nhỏ nhất trong các phần tử lớn hơn p*)."

+ Sau đó chép thông tin của phần tử thế mạng vào p và hủy phần tử thế mạng (*do phần tử thế mạng có tối đa một con*).

Cài đặt:

```
1.  int deleteTNodeX(TNode* &root, ItemType x)
2.  {
3.      if(root == NULL)      //khi cây rỗng
4.          return 0;  //Thực hiện không thành công
5.      if(root→Info > x)
6.          return deleteTNodeX(root→Left, x);
7.      else if(root→Info < x)
8.          return deleteTNodeX(root→Right, x);
9.      else
10.     { //root→Info = x, tìm nút thế mạng cho root
11.         TNode* p = root;
12.         if(root→Left == NULL)  //khi cây con không có nhánh trái
13.         {
14.             root = root→Right;
15.             delete p;
16.         }
17.         else if(root→Right == NULL)    //khi cây con không có nhánh phải
18.         {
19.             root = root→Left;
20.             delete p;
21.         }
22.         else      //khi cây con có cả 2 nhánh, chọn min của nhánh phải để thế mạng
23.         {
24.             TNode* q = findTNodeReplace(p);
25.             delete q;
26.         }
27.     }
28.     return 1;     //Thực hiện thành công
29. }
```

Hàm tìm nút con thế mạng (*phải viết trước hàm* deleteTNodeX):

```
1.  TNode* findTNodeReplace(TNode* &p)
2.  {//Hàm tìm nút q thế mạng cho nút p, f là nút cha của nút q.
3.      TNode* f = p;
4.      TNode* q = p→Right;
5.      while(q→Left != NULL)
6.      {
```

```
7.              f = q;     //Luu cha cua q
8.              q = q→Left;       //q qua ben trai
9.          }
10.         p→Info = q→Info;    //tim duuc phan tu the mang cho p la q
11.         if(f == p)     //neu cha cua q la p
12.             f→Right = q→Right;
13.         else
14.             f→Left = q→Right;
15.         return q;     //tra ve con tro q the mang cho p
16. }
```

5.5.2.6. Xóa toàn bộ cây

Thao tác chỉ đơn giản là việc thực hiện nhiều lần thao tác hủy một nút trên cây nhị phân tìm kiếm cho đến khi cây trở thành cây rỗng.

```
1.  int deleteBTree(TNode* &root)
2.  {
3.      if(root == NULL)
4.          return 0;  //Thực hiện không thành công
5.      deleteTree(root→Left);      //Xóa nút con bên nhánh trái
6.      deleteTree(root→Right);     //Xóa nút con bên nhánh phải
7.      deleteTNode(root);  //Xóa nút gốc
8.      return 1;      //Thực hiện thành công
9.  }
```

5.5.3. Nhận xét

Tất cả các thao tác tìm kiếm, xóa, thêm trên cây nhị phân tìm kiếm đều có độ phức tạp trung bình $\mathbf{O(\log_2(n))}$.

Cây nhị phân tìm kiếm là một cấu trúc dữ liệu cho phép tìm kiếm hiệu quả hơn hẳn danh sách liên kết, mà các thao tác không quá phức tạp. Cây nhị phân tìm kiếm là một trong những cấu trúc dữ liệu động thông dụng nhất. Tuy nhiên trong một số trường hợp nó suy biến thành danh sách liên kết.

5.6. Cây nhị phân tìm kiếm cân bằng

5.6.1. Định nghĩa

– **Cây cân bằng tương đối:**

Theo Adelson - Velskii và Landis đưa ra định nghĩa về cây cân bằng tương đối như sau: Cây cân bằng tương đối là một cây nhị phân thỏa mãn điều kiện là đối với mỗi nút của cây thì chiều cao của cây con trái và chiều cao của cây con phải của nút đó hơn kém nhau không quá một.

Cây cân bằng tương đối còn được gọi là cây AVL (AVL Tree).

– **Cây cân bằng hoàn toàn:**

Cây cân bằng hoàn toàn là một cây nhị phân thỏa mãn điều kiện là đối với mỗi nút của cây thì số nút ở cây con trái và số nút ở cây con phải của nút đó hơn kém nhau không quá một.

Như vậy, một cây cân bằng hoàn toàn chắc chắn là một cây cân bằng tương đối.

Cân cân bằng hoàn toàn có n nút có chiều cao h = $\log_2(n)$. Đây là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên cấu trúc dữ liệu này.

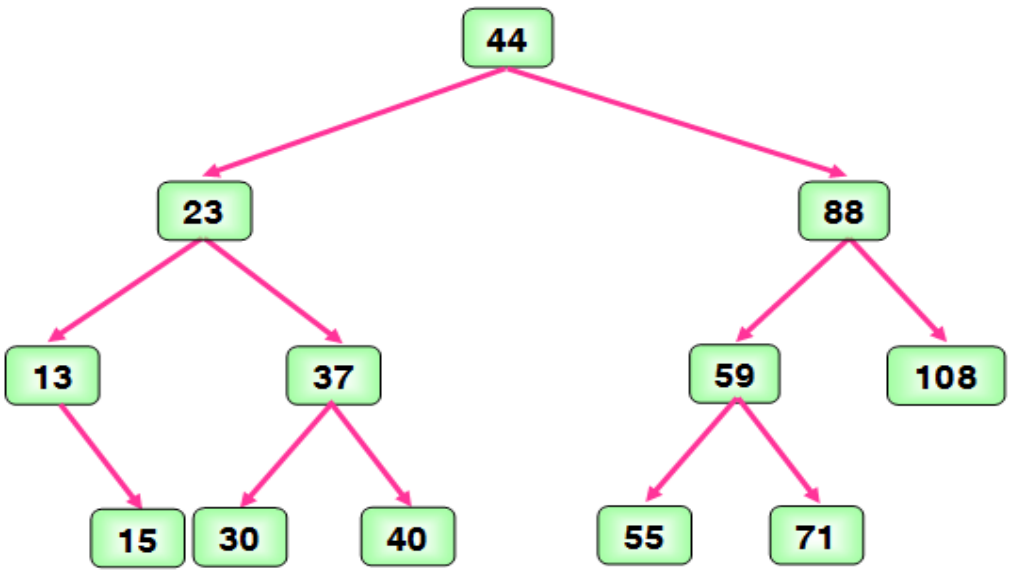
Nhược điểm cây cân bằng hoàn toàn là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng.

5.6.2. Cây AVL (AVL Tree)

5.6.2.1. Định nghĩa

Cây nhị phân tìm kiếm cân bằng (AVL) là cây nhị phân tìm kiếm mà tại mỗi nút của nó chiều cao của cây con bên trái và chiều cao của cây con bên phải chênh lệch nhau không quá một.

Ví dụ: Hình vẽ sau là một cây AVL.



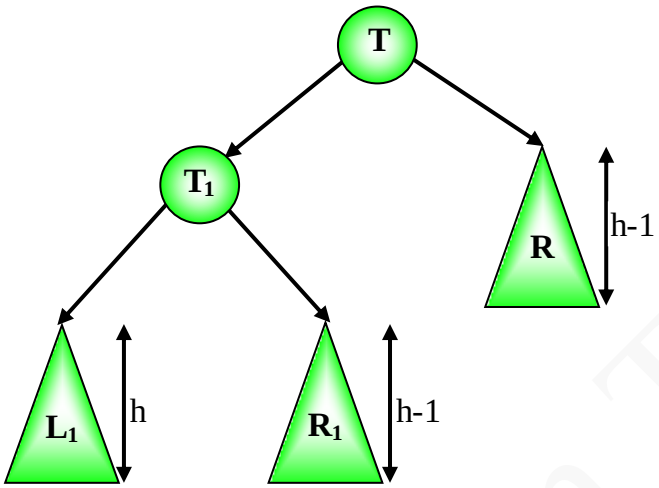
Hình 5.10: Ví dụ minh họa cây AVL

5.6.2.2. Các trường hợp cây bị mất cân bằng

5.6.2.2.1. **Trường hợp 1:** Cây T lệch về bên trái (có 3 khả năng).

a. Cây T lệch về bên trái của cây con trái (LL: Left - Left):

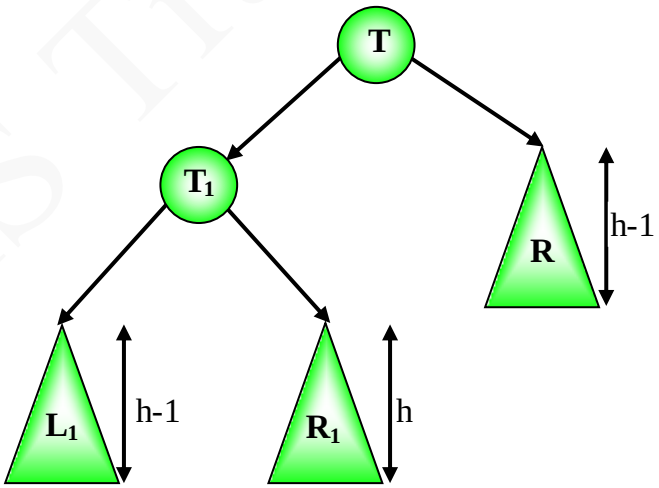
- Chiều cao cây con phải R của cây T là (h - 1).
- Chiều cao cây con trái T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 của cây T_1 là h và chiều cao của cây con phải R_1 của cây T_1 là (h - 1).



Hình 5.11: Hình vẽ minh họa cây AVL lệch trái LL

b. Cây T lệch về bên phải của cây con trái (LR: Left - Right):

- Chiều cao cây con phải R của cây T là (h - 1).
- Chiều cao cây con trái T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 của cây T_1 là (h - 1) và chiều cao của cây con phải R_1 của cây T_1 là h.

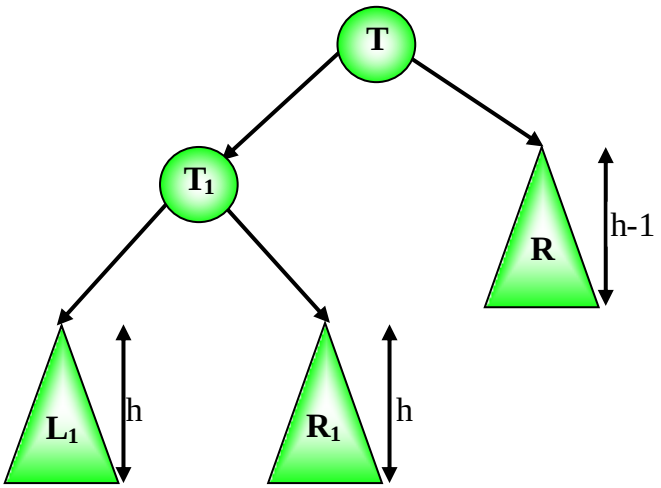


Hình 5.12: Hình vẽ minh họa cây AVL lệch trái LR

c. Cây con trái của T cân bằng (LB: Left - Balance):

- Chiều cao cây con phải R của cây T là (h - 1).

– Chiều cao cây con trái T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 và cây con phải R_1 của cây T_1 đều là h.



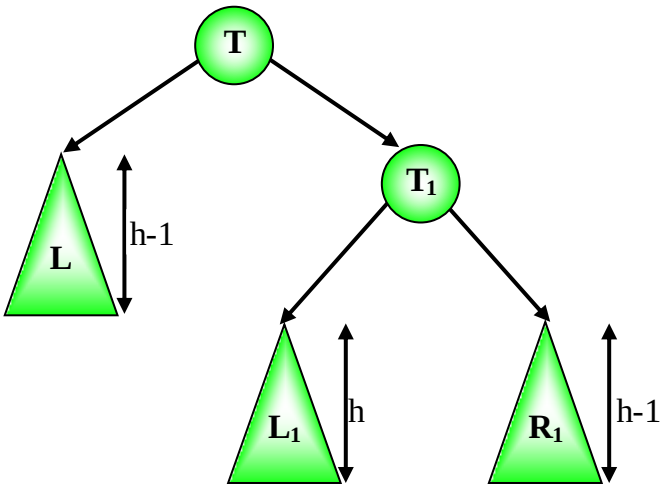
Hình 5.13: Hình vẽ minh họa cây AVL lệch trái LB

5.6.2.2.2. **Trường hợp 2:** Cây T lệch về bên phải (có 3 khả năng).

a. Cây T lệch về bên trái của cây con phải (RL: Right - Left):

– Chiều cao cây con trái L của cây T là (h - 1).

– Chiều cao cây con phải T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 của cây T_1 là h và chiều cao của cây con phải R_1 của cây T_1 là (h - 1).

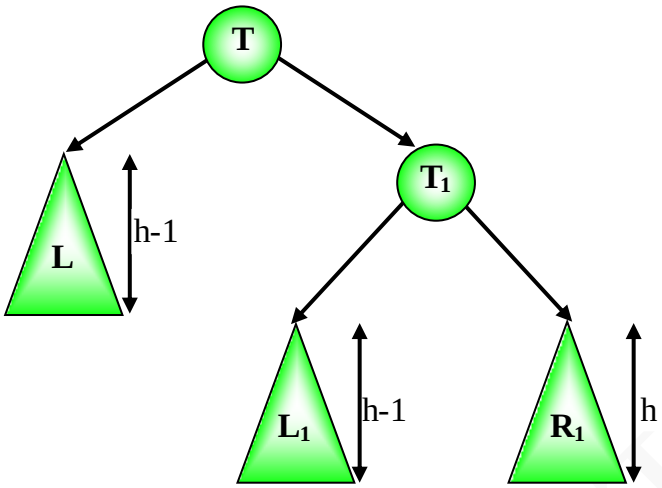


Hình 5.14: Hình vẽ minh họa cây AVL lệch phải RL

b. Cây T lệch về bên phải của cây con phải (RR: Right - Right):

– Chiều cao cây con trái L của cây T là (h - 1).

– Chiều cao cây con phải T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 của cây T_1 là (h - 1) và chiều cao của cây con phải R_1 của cây T_1 là h.

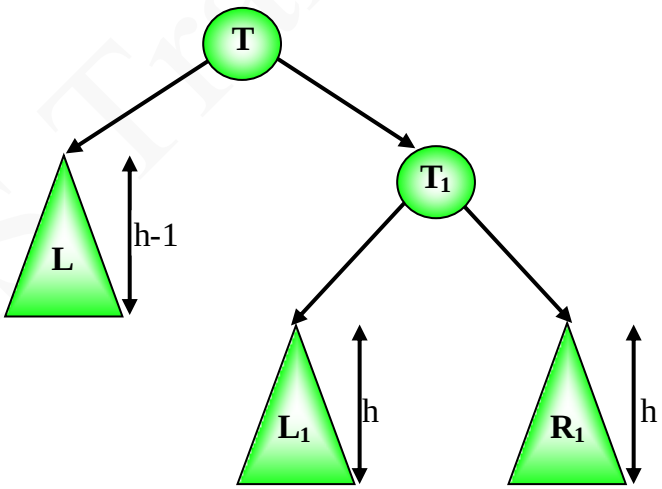


Hình 5.15: Hình vẽ minh họa cây AVL lệch phải RR

c. Cây con phải của T cân bằng (RB: Right - Balance):

– Chiều cao cây con trái L của cây T là (h - 1).

– Chiều cao cây con phải T_1 của cây T là (h + 1), trong đó chiều cao của cây con trái L_1 và cây con phải R_1 của cây T_1 đều là h.



Hình 5.16: Hình vẽ minh họa cây AVL lệch phải RB

5.6.2.3. Chỉ số cân bằng của một nút

5.6.2.3.1. Định nghĩa:

Chỉ số cân bằng của một nút là hiệu của chiều cao cây con bên phải và cây con bên trái của nó.

Đối với cây cân bằng thì chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau:

- **CSCB(p) = 0** ⇔ Độ cao cây con trái (p) = Độ cao cây con phải (p).
- **CSCB(p) = -1** ⇔ Độ cao cây con trái (p) > Độ cao cây con phải (p).
- **CSCB(p) = 1** ⇔ Độ cao cây con trái (p) < Độ cao cây con phải (p).

Chúng ta ký hiệu như sau:

- p→balFactor = CSCB(p).
- $\mathbf{h_L}$: Độ cao cây con trái.
- $\mathbf{h_R}$: Độ cao cây con phải.

Để khảo sát cây cân bằng thì cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau:

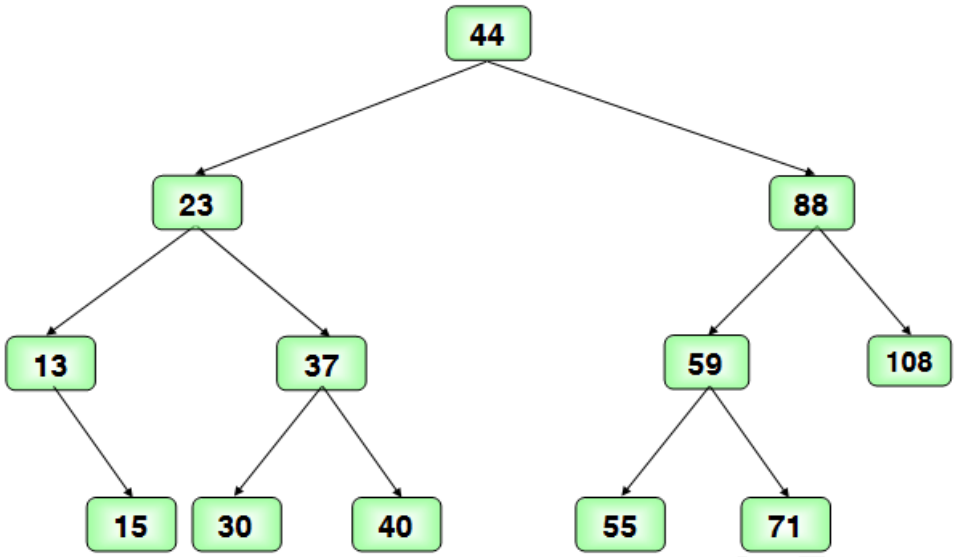
```
1.  #define LH -1       //Cây con trái cao hơn (lệch trái)
2.  #define EH 0        //Cây con trái bằng cây con phải (cân bằng)
3.  #define RH 1        //Cây con phải cao hơn (lệch phải)
4.  typedef <Kiểu dữ liệu của một nút> ItemType; //có thể là int

5.  struct AVLNode
6.  {
7.      int balFactor;
8.      ItemType Info;
9.      AVLNode* Left;
10.     AVLNode* Right;
11. };

12. struct AVLTree
13. {
14.     AVLNode* Root;
15. };
```

Ta nhận thấy trường hợp thêm hay hủy một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây. Việc cân bằng lại một cây sẽ phải thực hiện sao cho ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng. Như đã nói ở trên, cây cân bằng cho phép việc cân bằng lại chỉ xảy ra trong giới hạn cục bộ nên chúng ta có thể thực hiện được mục tiêu vừa nêu.

5.6.2.3.2. Ví dụ: Cho cây sau:



Hình 5.17: Ví dụ minh họa cây AVL có nút mất cân bằng lệch trái

Xét nút p có giá trị bằng 88, thì hệ số cân bằng của nút p: CSCB(p) = -1.

5.6.2.4. Các thao tác trên cây AVL

Khi thêm hay xóa một nút trên cây, có thể làm cho cây mất tính cân bằng, khi ấy sẽ phải tiến hành cân bằng lại cây.

Có khả năng cây bị mất cân bằng khi thay đổi chiều cao:

- Thêm bên trái → Lệch sang nhánh trái.
- Thêm bên phải → Lệch sang nhánh phải.
- Hủy bên phải → Lệch sang nhánh trái.
- Hủy bên trái → Lệch sang nhánh phải.

Vì vậy việc cân bằng lại cây là tìm cách bố trí lại cây sao cho chiều cao của hai cây con được cân đối với nhau:

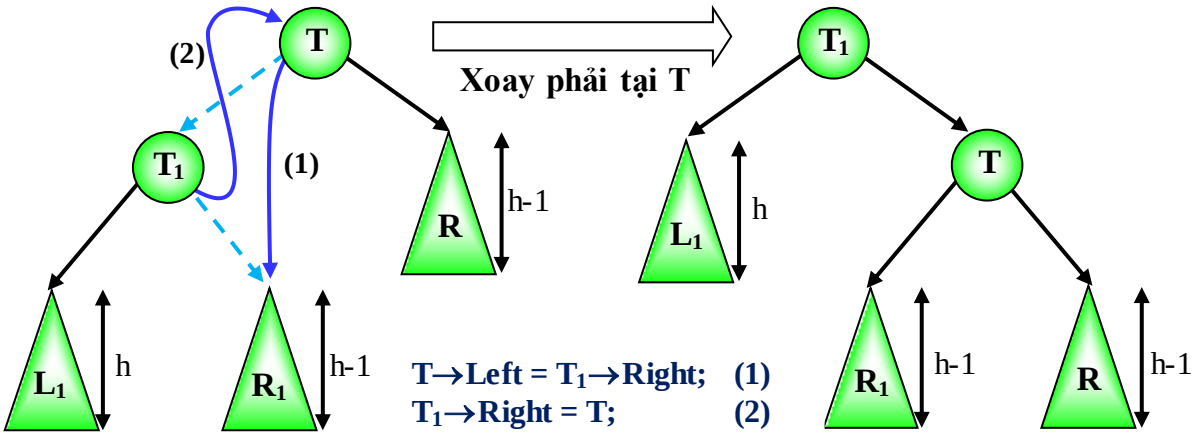
- Kéo nhánh cao bù cho nhánh thấp.
- Phải bảo đảm cây vẫn là nhị phân tìm kiếm.

5.6.2.4.1. Cân bằng lại cây

Tại nút T không cân bằng. Ta phải giải quyết hai trường hợp sau:

a. Trường hợp 1:

– **Cân bằng lại LL (Left - Left):**

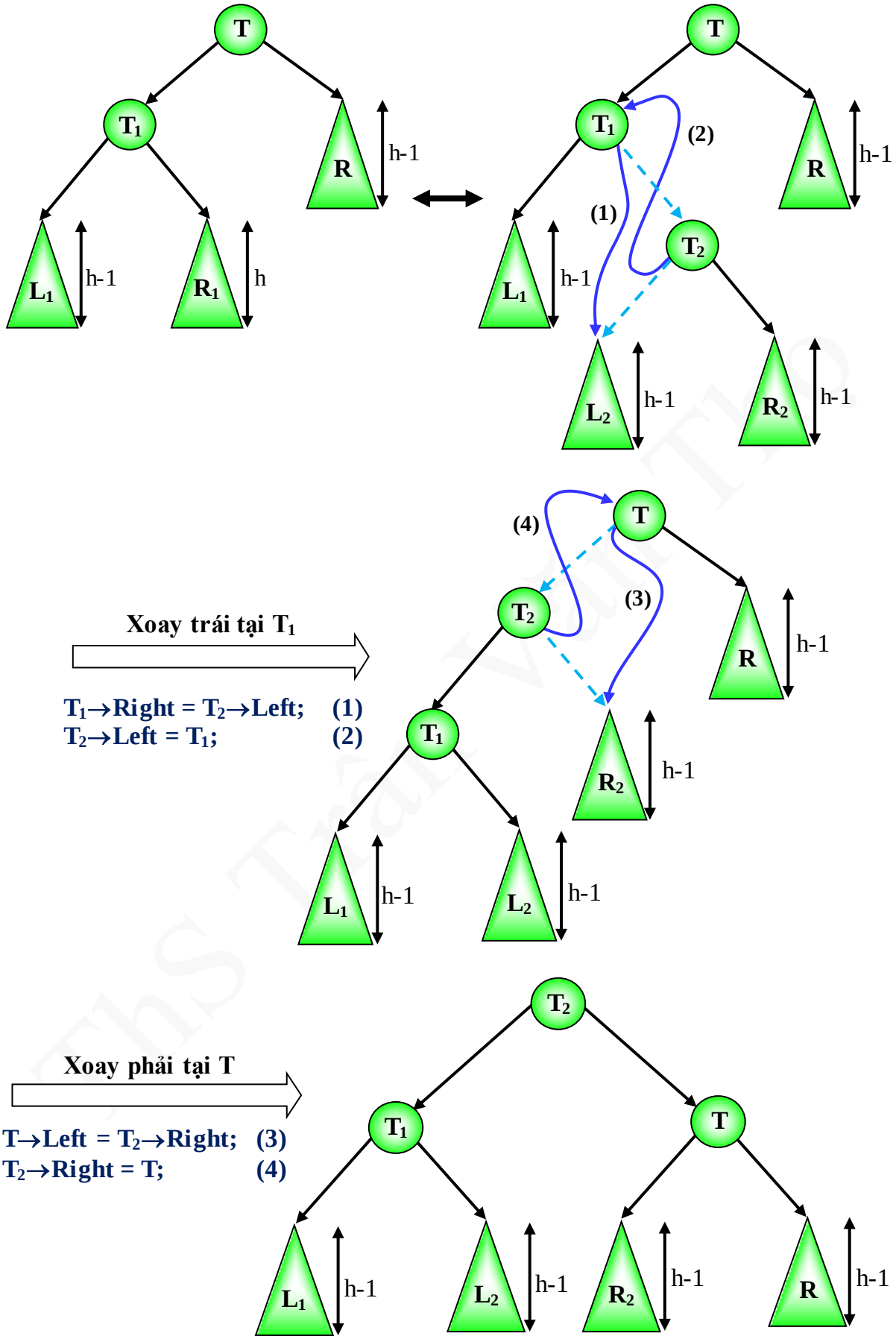


Hình 5.18: Hình vẽ minh họa cân bằng lại LL cho cây AVL

Cài đặt:

```
1.  void rotateLL(AVLNode* &T)
2.  {
3.      AVLNode* T1 = T→Left;
4.      T→Left = T1→Right;
5.      T1→Right = T;
6.      switch(T1→balFactor)
7.      {
8.          case LH:   //cây T1 lệch trái
9.              T→balFactor = EH;
10.             T1→balFactor = EH;
11.             break;
12.         case EH:   //cây T1 cân bằng
13.             T→balFactor = LH;
14.             T1→balFactor = RH;
15.             break;
16.     }
17.     T = T1;
18. }
```

– **Cân bằng lại LR (Left - Right):**

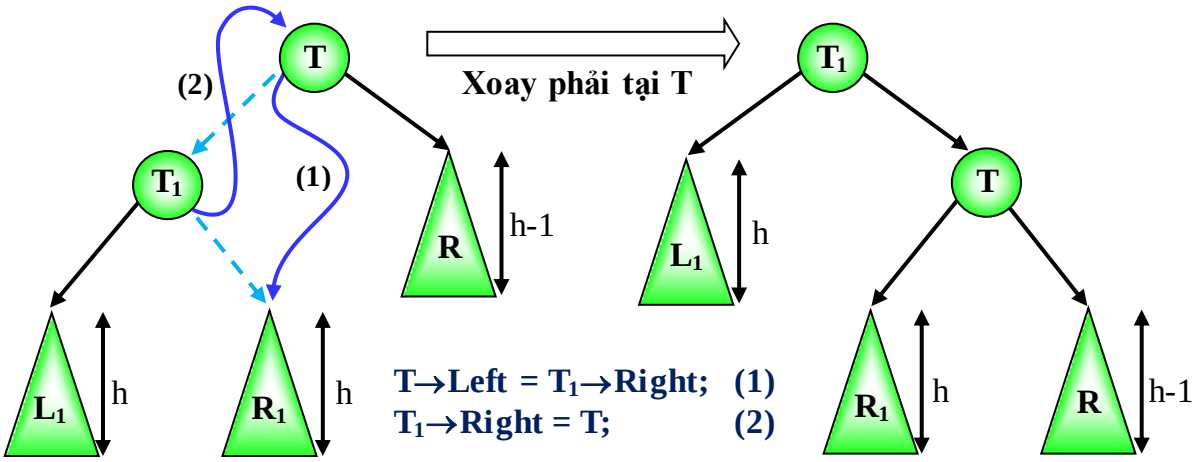


Hình 5.19: Hình vẽ minh họa cân bằng lại LR cho cây AVL

Cài đặt:

```
1.  void rotateLR(AVLNode* &T)
2.  {
3.      AVLNode* T1 = T→Left;
4.      AVLNode* T2 = T1→Right;
5.      T1→Right = T2→Left;
6.      T2→Left = T1;
7.      T→Left = T2→Right;
8.      T2→Right = T;
9.      switch(T2→balFactor)
10.     {
11.         case LH:
12.             T→balFactor = RH;
13.             T1→balFactor = EH;
14.             break;
15.         case EH:
16.             T→balFactor = EH;
17.             T1→balFactor = EH;
18.             break;
19.         case RH:
20.             T→balFactor = EH;
21.             T1→balFactor = LH;
22.             break;
23.     }
24.     T2→balFactor = EH;
25.     T = T2;
26. }
```

– **Cân bằng lại LB (Left - Balance):**

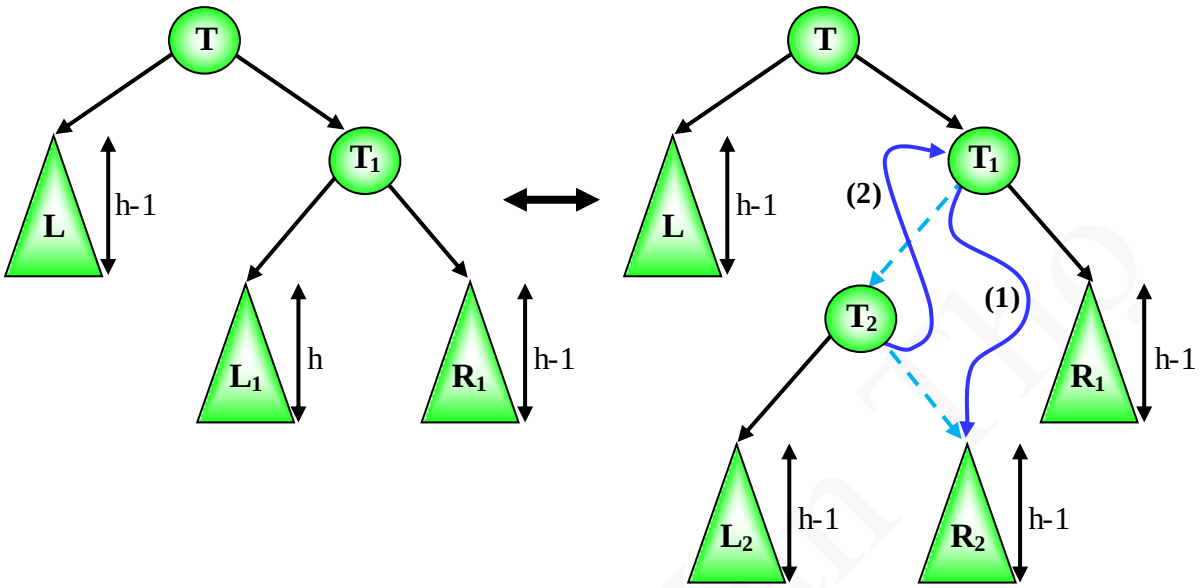


Hình 5.20: Hình vẽ minh họa cân bằng lại LB cho cây AVL

Cài đặt: *Việc cân bằng lại cây trong trường hợp **LB** hoàn toàn giống như **LL**.*

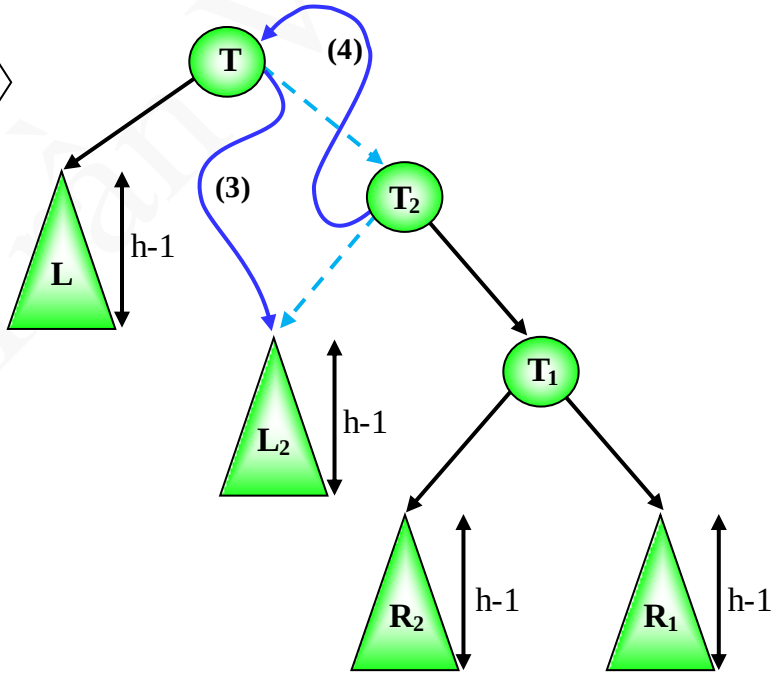
b. Trường hợp 2:

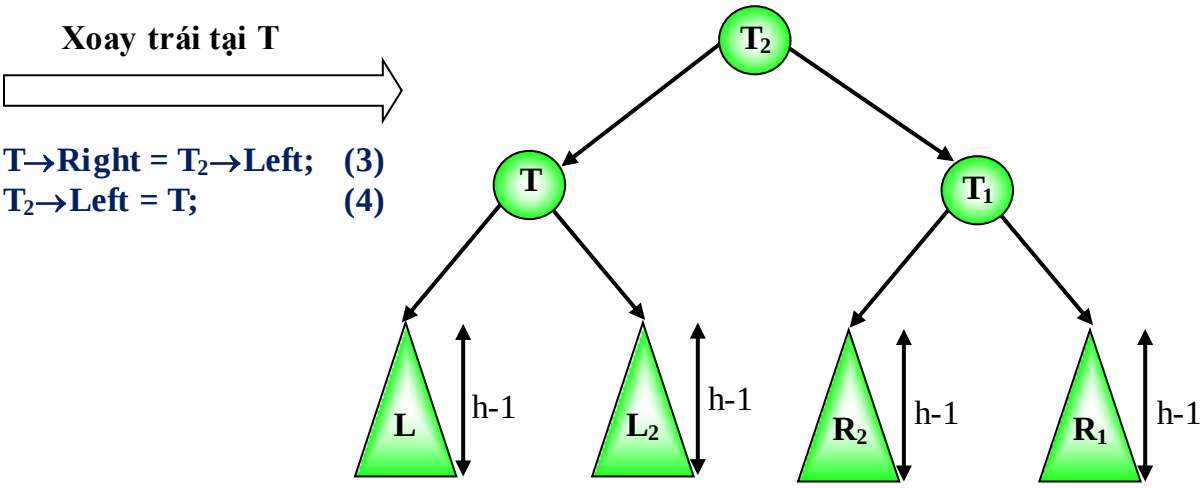
– **Cân bằng lại RL (Right - Left):**



Xoay phải tại T_1

$T_1 \rightarrow Left = T_2 \rightarrow Right$; (1)
$T_2 \rightarrow Right = T_1$; (2)



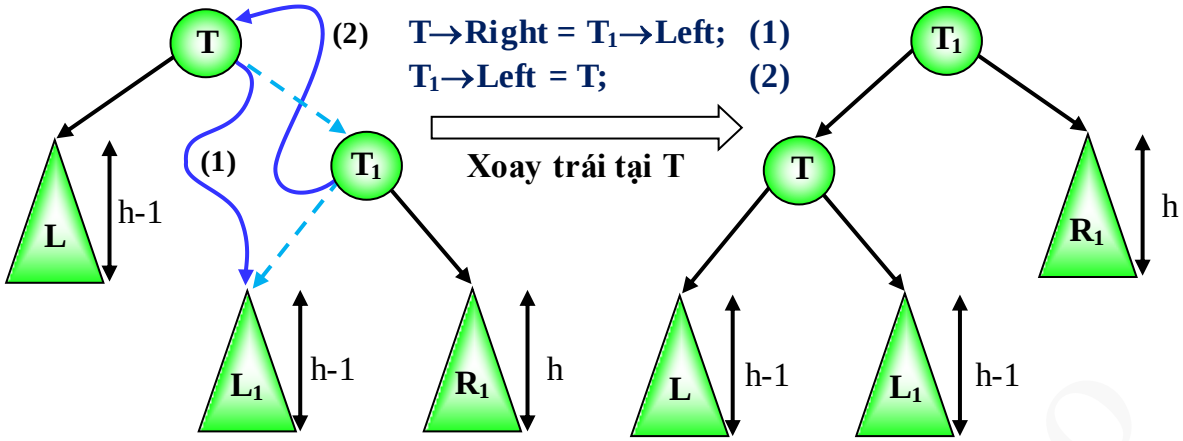


Hình 5.21: Hình vẽ minh họa cân bằng lại RL cho cây AVL

Cài đặt:

```
1.  void rotateRL(AVLNode* &T)
2.  {
3.      AVLNode* T1 = T→Right;
4.      AVLNode* T2 = T1→Left;
5.      T1→Left = T2→Right;
6.      T2→Right = T1;
7.      T→Right = T2→Left;
8.      T2→Left = T;
9.      switch(T2→balFactor)
10.     {
11.         case RH:
12.             T→balFactor = LH;
13.             T1→balFactor = EH;
14.             break;
15.         case EH:
16.             T→balFactor = EH;
17.             T1→balFactor = EH;
18.             break;
19.         case LH:
20.             T→balFactor = EH;
21.             T1→balFactor = RH;
22.             break;
23.     }
24.     T2→balFactor = EH;
25.     T = T2;
26. }
```

– **Cân bằng lại RR (Right - Right):**

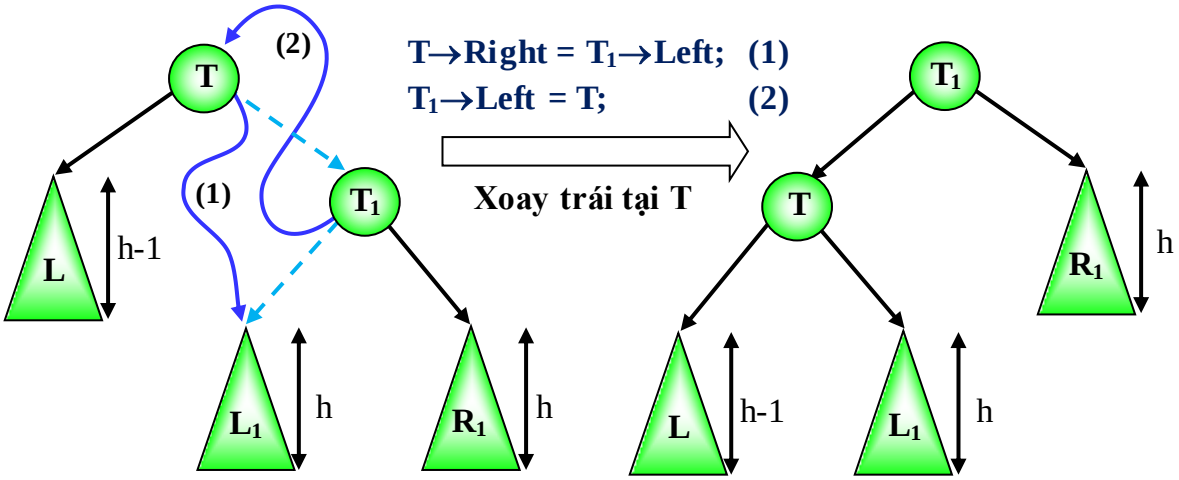


Hình 5.22: Hình vẽ minh họa cân bằng lại RR cho cây AVL

Cài đặt:

```
1.  void rotateRR(AVLNode* &T)
2.  {
3.      AVLNode* T1 = T→Right;
4.      T→Right = T1→Left;
5.      T1→Left = T;
6.      switch(T1→balFactor)
7.      {
8.          case RH:  //cây T1 lệch phải
9.              T→balFactor = EH;
10.             T1→balFactor = EH;
11.             break;
12.         case EH:  //cây T1 cân bằng
13.             T→balFactor = RH;
14.             T1→balFactor = LH;
15.             break;
16.     }
17.     T = T1;
18. }
```

– **Cân bằng lại RB (Right - Balance):**



Hình 5.23: Hình vẽ minh họa cân bằng lại RB cho cây AVL

Cài đặt: *Việc cân bằng lại cây trong trường hợp **RB** hoàn toàn giống như **RR**.*

c. Cân bằng lại cây AVL khi bị mất cân bằng

```
1.  int balanceLeft(AVLNode* &T)
2.  {//Khi cây T lệch bên trái cần cân bằng lại
3.      AVLNode* T1 = T→Left;
4.      switch(T1→balFactor)
5.      {
6.          case LH:
7.              rotateLL(T);
8.              return 2;
9.          case EH:
10.             rotateLL(T);
11.             return 1;
12.         case RH:
13.             rotateLR(T);
14.             return 2;
15.     }
16.     return 0;     //Thực hiện không thành công
17. }
```

```
1.  int balanceRight(AVLNode* &T)
2.  {//Khi cây T lệch bên phải cần cân bằng lại
3.      AVLNode* T1 = T→Right;
4.      switch(T1→balFactor)
5.      {
6.          case LH:
7.              rotateRL(T);
8.              return 2;
9.          case EH:
```

```
10.                 rotateRR(T);
11.                 return 1;
12.             case RH:
13.                 rotateRR(T);
14.                 return 2;
15.         }
16.         return 0;     //Thực hiện không thành công
17. }
```

5.6.2.4.2. Thêm một phần tử x vào cây

- Thêm bình thường như trường hợp cây nhị phân tìm kiếm.
- Nếu cây tăng trưởng chiều cao:
  + Lần ngược về gốc để phát hiện nút bị mất cân bằng.
  + Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp.
- Việc cân bằng lại chỉ cần thực hiện một lần tại nơi mất cân bằng.

Trước tiên, ta xét các tình huống có thể xảy ra khi ta thêm x vào cây con trái của nút P. Ta thấy có ba tình huống có thể xảy ra:

- p→balFactor = 0 ($h_L = h_R$): Thêm x vào thì p→balFactor = -1
- p→balFactor = 1 ($h_L < h_R$): Thêm x vào thì p→balFactor = 0
- p→balFactor = -1 ($h_L > h_R$): Thêm x vào thì phải cân bằng lại cây.

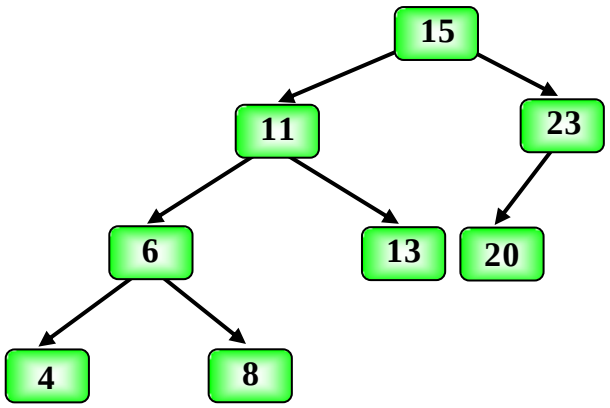
➔ Trong 3 trường hợp trên thì chỉ có trường hợp 3 là cần cân bằng lại cây sau khi thêm, do lúc đầu $h_L = h_R + 1$, nếu thêm một phần tử có khả năng làm tăng h_L ➔ $h_L = h_R + 2$.

Tương tự, nếu thêm một phần tử x vào nhánh phải của p ta cũng có ba tình huống:

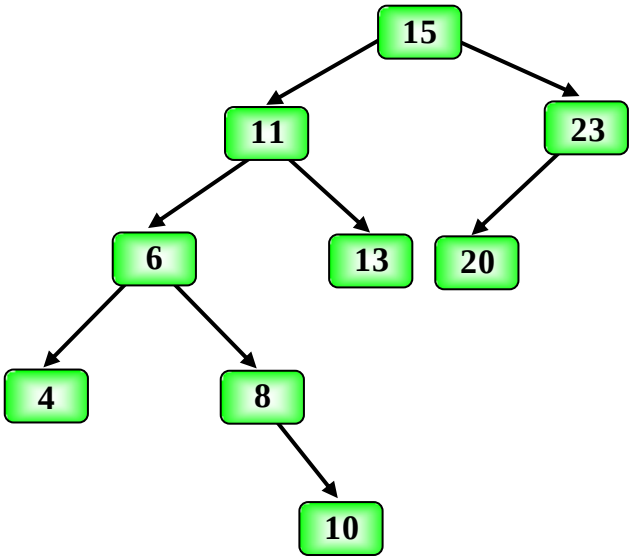
- p→balFactor = 0 ($h_L = h_R$): Thêm x vào thì p→balFactor = 1
- p→balFactor = -1 ($h_L > h_R$): Thêm x vào thì p→balFactor = 0
- p→balFactor = 1 ($h_L < h_R$): Têm x vào thì phải cân bằng lại cây.

➔ Trong ba trường hợp trên thì chỉ có trường hợp 3 là cần cân bằng lại cây sau khi thêm, do lúc đầu $h_R = h_L + 1$, nếu thêm một phần tử có khả năng làm tăng h_R ➔ $h_R = h_L + 2$.

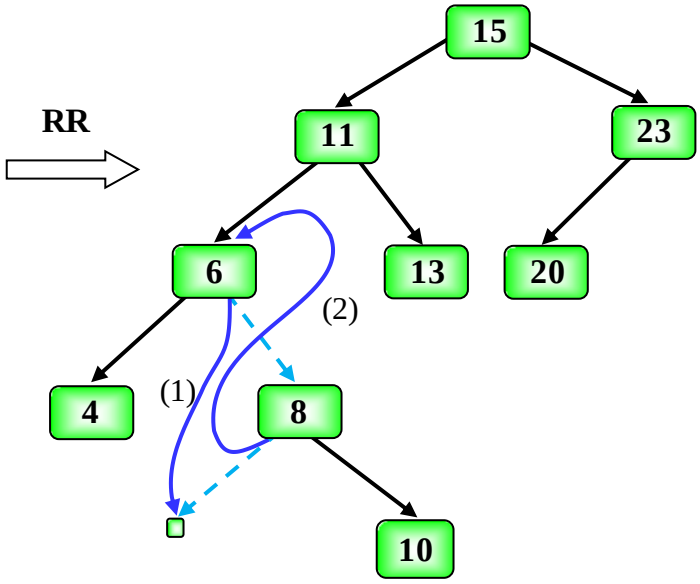
Ví dụ: Cho cây sau:

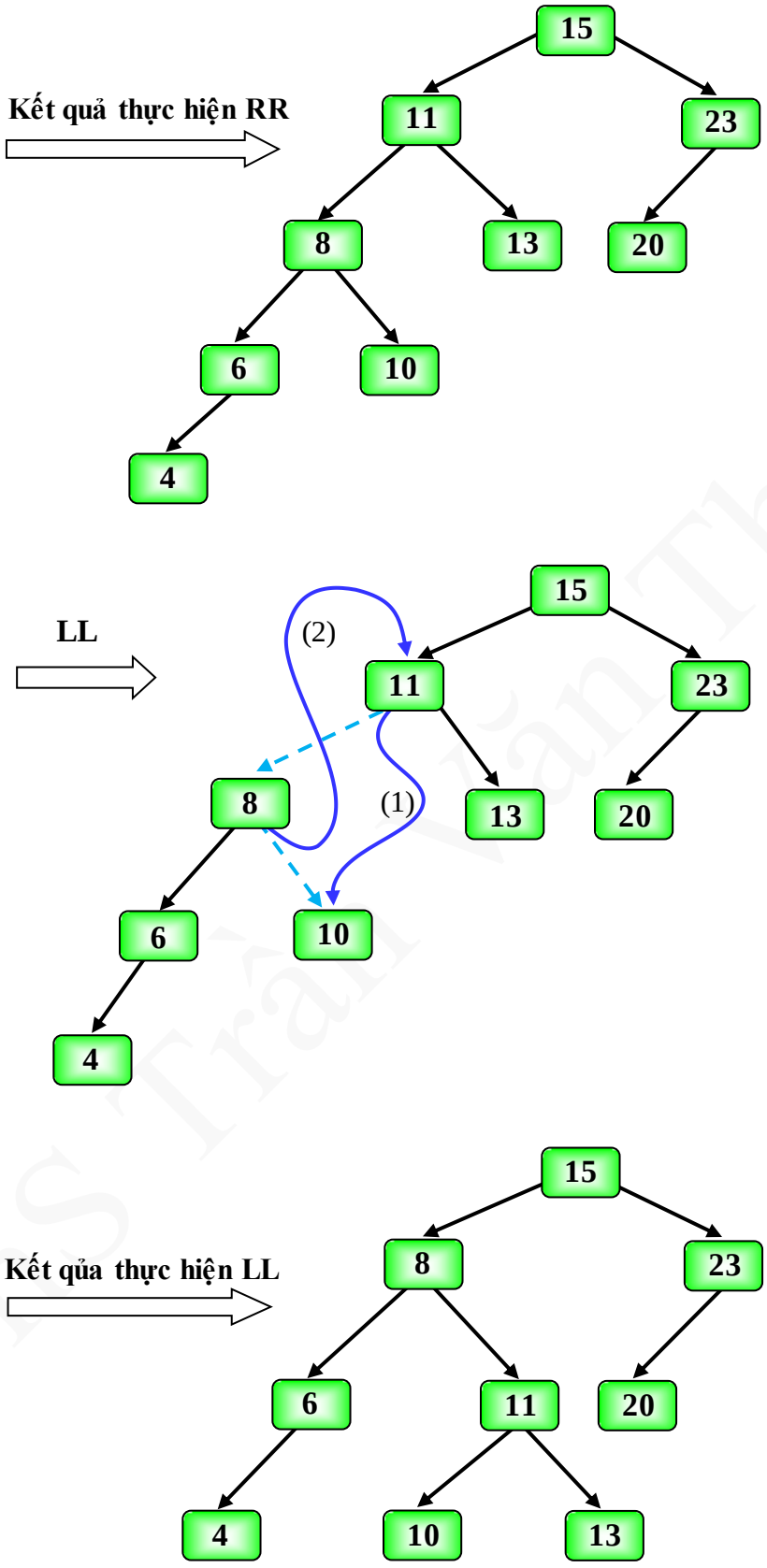


Thêm nút p có giá trị 10 vào cây. Thực hiện như sau:



Cây bị mất cân bằng LR (Left - Right) tại nút có giá trị 11 cần phải cân bằng lại:





Hình 5.24: Ví dụ minh họa thêm một nút vào cây AVL

Vậy cây đã được cân bằng sau khi sắp xếp.

Giải thuật:

- **Bước 1:** Đi theo đường tìm kiếm để xác định phần tử cần thêm vào cây.
- **Bước 2:** Nếu phần tử cần thêm chưa tồn tại trên cây thì:
  + Thêm phần tử đó vào cây.
  + Xác định lại hệ số cân bằng của phần tử mới thêm.
- **Bước 3:** Lần ngược theo đường tìm kiếm và kiểm tra hệ số cân bằng tại mỗi nút, nếu thấy mất cân bằng thì phải cân bằng lại.

Cài đặt:

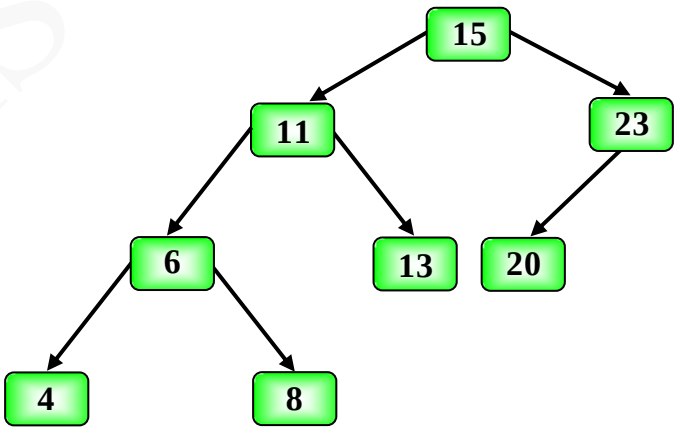
```
1.  int insertAVLNode(AVLNode* &T, TNode* p)
2.  {
3.      if(p == NULL)
4.          return -1;    //Nút mới p muốn thêm không tồn tại
5.      if(T == NULL)
6.      {//Cây rỗng nên thêm nút mới vào gốc
7.          T = p;
8.          return 2;     //Thực hiện thành công nút mới có giá trị x
9.      }
10.     //Cây không rỗng
11.     int Result;
12.     if(T→Info == p→Info)
13.         return 0; //Không thêm được vì tồn tại nút có giá trị bằng x
14.     if(T→Info > p→Info)
15.     {//Thêm nút mới vào nhánh trái
16.         Result = insertAVLNode(T→Left, p);
17.         if(Result < 2)
18.             return Result;
19.         switch(T→balFactor)
20.         {
21.             case RH:
22.                 T→balFactor = EH;
23.                 return 1;
24.             case EH:
25.                 T→balFactor = LH;
26.                 return 2;
27.             case LH:
28.                 balanceLeft(T);
29.                 return 1;
30.         }
31.     }
32.     else
33.     {//Thêm nút mới vào nhánh phải
34.         Result = insertAVLNode(T→Right, p);
35.         if(Result < 2)
36.             return Result;
```

```
37.             switch(T→balFactor)
38.             {
39.                 case LH:
40.                     T→balFactor = EH;
41.                     return 1;
42.                 case EH:
43.                     T→balFactor = RH;
44.                     return 2;
45.                 case RH:
46.                     balanceRight(T);
47.                     return 1;
48.             }
49.         }
50. }
```

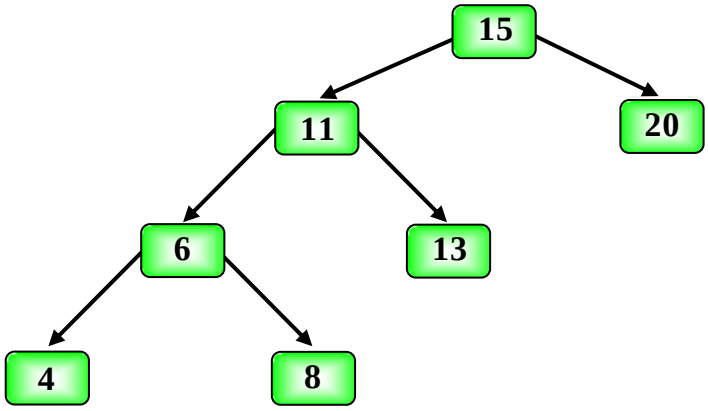
5.6.2.4.3. Xóa một phần tử x trong cây

- Các trường hợp loại bỏ một phần tử trên cây cân bằng vẫn được giải quyết như khi loại bỏ trên cây nhị phân tìm kiếm.
- Nếu cây giảm chiều cao:
  + Lần ngược về gốc để phát hiện nút bị mất cân bằng.
  + Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp.
  + Tiếp tục lần ngược lên nút cha.
- Việc cân bằng lại có thể lan truyền lên tận gốc.
- Các trường hợp cân bằng lại giải quyết giống như thêm vào cây.

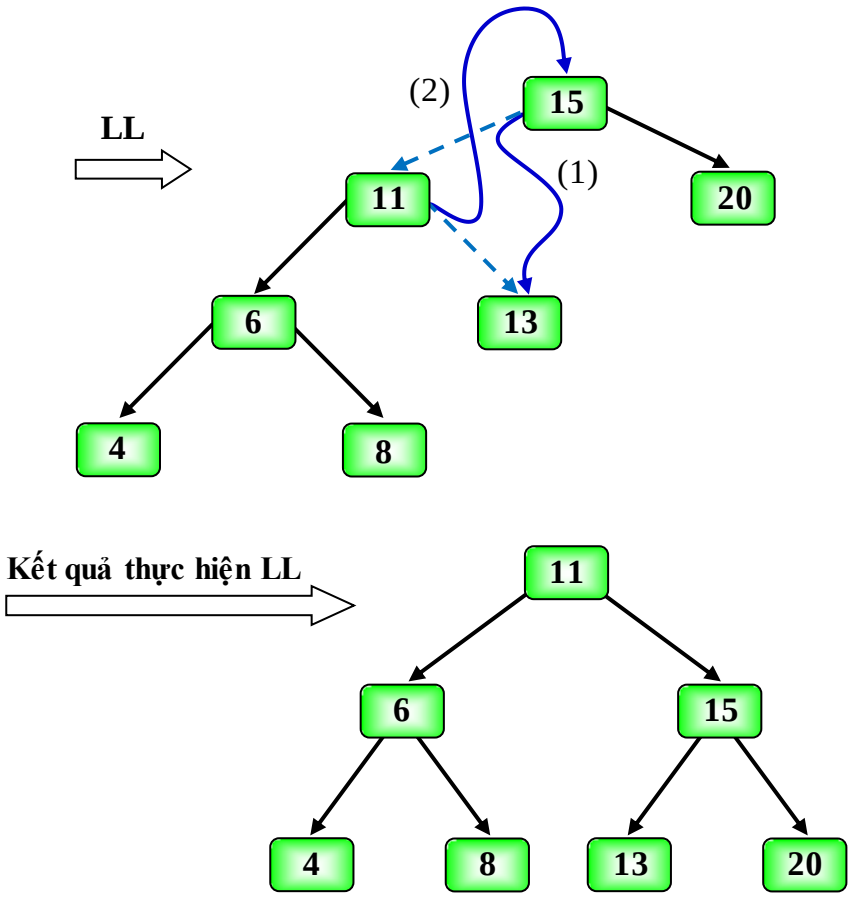
Ví dụ: Cho cây cân bằng sau:



Hủy phần tử có giá trị 23. Ta thực hiện như sau:



Cây bị mất cân bằng LL (Left - Left) tại nút có giá trị 15 cần phải cân bằng lại:



Hình 5.25: Ví dụ minh họa xóa một nút từ cây AVL

Vậy cây đã cân bằng lại sau khi xóa nút có giá trị 23.

Giải thuật:

- Nếu (p == NULL) thì
  + "Không có phần tử để hủy"
- Ngược lại:
  + Nếu (x < p→Info) thì

- Thực hiện hủy trên cây con trái.
- Cân bằng lại cho p nếu cây con trái giảm độ cao (Balance - Left)

+ Ngược lại:

- Nếu (x > p→Info) thì

  Thực hiện hủy trên cây con phải.

  Cân bằng lại cho p nếu cây con phải giảm độ cao (Balance - Right)

- Ngược lại: //*x = p→Info*

  Hủy p giống giải thuật trên cây nhị phân tìm kiếm.

Cài đặt:

```
1.  int searchStandFor(AVLNode* &p, AVLNode* &q)
2.  {
3.      int Result;
4.      if(q→Left != NULL)
5.      {
6.          Result = searchStandFor(p, q→Left);
7.          if(Result < 2)
8.              return Result;
9.          switch(q→balFactor)
10.         {
11.             case LH:
12.                 q→balFactor = EH;
13.                 return 2;
14.             case EH:
15.                 q→balFactor = RH;
16.                 return 1;
17.             case RH:
18.                 return balanceRight(q);
19.         }
20.     }
21.     else
22.     {
23.         p→Info = q→Info;
24.         p = q;
25.         q = q→Right;
26.     }
27.     return 2;
28. }
```

```
1.  int deleteAVLNode(AVLNode* &T, ItemType x)
2.  { //Xóa nút có Info bằng với x
3.      int Result;
4.      if(T == NULL)
5.          return 0;  //Thực hiện không thành công
```

```
6.      if(T→Info > x)
7.      {
8.          Result = deleteAVLNode(T→Left, x);
9.          if(Result < 2)
10.             return Result;
11.         switch(T→balFactor)
12.         {
13.             case LH:
14.                 T→balFactor = EH;
15.                 return 2;
16.             case EH:
17.                 T→balFactor = RH;
18.                 return 1;
19.             case RH:
20.                 return balanceRight(T);
21.         }
22.     }
23.     else if(T→Info < x)
24.     {
25.         Result = deleteAVLNode(T→Right, x);
26.         if(Result < 2)
27.             return Result;
28.         switch(T→balFactor)
29.         {
30.             case RH:
31.                 T→balFactor = EH;
32.                 return 2;
33.             case EH:
34.                 T→balFactor = LH;
35.                 return 1;
36.             case LH:
37.                 return balanceLeft(T);
38.         }
39.     }
40.     else
41.     {
42.         AVLNode* p = T;
43.         if(T→Left == NULL)
44.         {
45.             T = T→Right;
46.             Result = 2;
47.         }
48.         else
49.         {
50.             if(T→Right == NULL)
51.             {
52.                 T = T→Left;
53.                 Result = 2;
54.             }
55.             else
56.             {
```

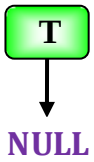
```
57.                     Result = searchStandFor(p, T→Right);
58.                     if(Result < 2)
59.                       return Result;
60.                     switch(T→balFactor)
61.                     {
62.                       case RH:
63.                             T→balFactor = EH;
64.                             return 2;
65.                       case EH:
66.                             T→balFactor = LH;
67.                             return 1;
68.                       case LH:
69.                             return balanceLeft(T);
70.                     }
71.                 }
72.             }
73.             delete p;
74.             return Result;
75.         }
76.         return Result;
77. }
```

5.6.2.4.4. Tạo cây nhị phân cân bằng AVL

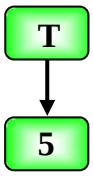
Việc tạo cây nhị phân cân bằng AVL là quá trình thêm lần lượt các phần tử (nút) vào cây, quá trình này sẽ dẫn đến việc mất cân bằng và phải cân bằng lại cây.

a. Ví dụ: Tạo cây AVL từ danh sách sau:

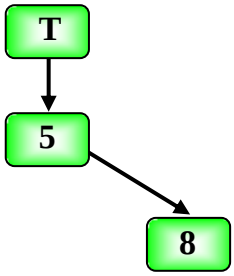
Tạo cây rỗng:



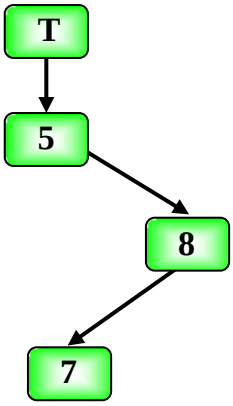
Thêm nút có giá trị 5 vào cây:



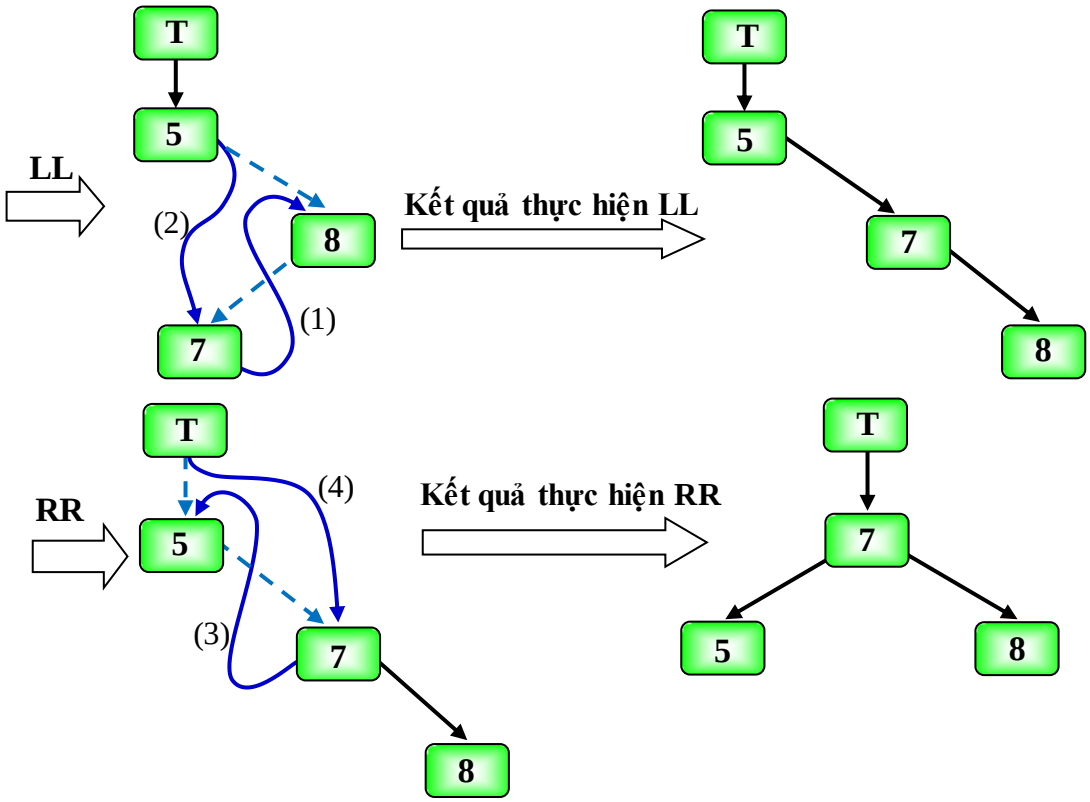
Thêm nút có giá trị 8 vào cây:



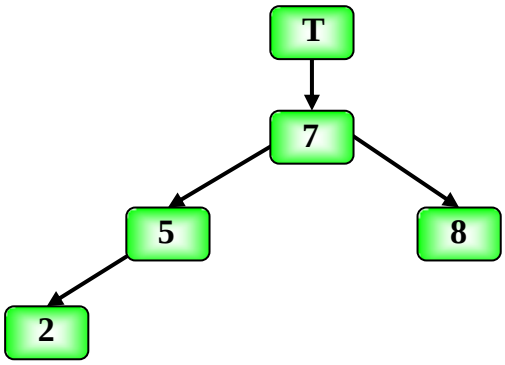
Thêm nút có giá trị 7 vào cây:



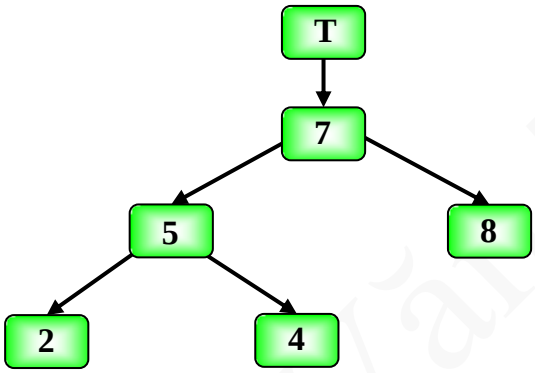
Cây bị mất cân bằng RL tại nút có giá trị 5 cần phải cân bằng lại:



Thêm nút có giá trị 2 vào cây:



Thêm nút có giá trị 4 vào cây:



Hình 5.26: Ví dụ minh họa về cách tạo cây AVL

Vậy cây nhị phân cân bằng AVL đã hoàn tất.

b. Cài đặt:

```
1.  void createAVLTree(AVLNode* &T, int n)
2.  {
3.      ItemType x;
4.      int Result, i = 1;
5.      do
6.      {
7.          printf("Tạo nút thứ %d có giá trị: ", i);
8.          scanf("%d", &x);
9.          Result = insertAVLNode(T, x);
10.         if(Result != 0)
11.             i++;
12.     }while(i <= n);
13. }
```

5.7. Câu hỏi và bài tập

5.7.1. Câu hỏi

1. Hiểu được các thao tác duyệt trên cây. Ứng dụng xóa cây được thực hiện theo thứ tự nào.

2. Đặc điểm của cây nhị phân tìm kiếm. Thao tác nào thực hiện tốt trong cây nhị phân tìm kiếm. Hạn chế của cây nhị phân tìm kiếm là gì?
3. Xét giải thuật tạo cây nhị phân tìm kiếm. Nếu thứ tự các khóa nhập vào là như sau:

   7 1 13 11 14 3 -4 8 5 9.

   thì hình ảnh cây tạo được như thế nào?

   Sau đó hủy lần lượt các nút theo thứ tự: 3, 7 thì cây sẽ thay đổi như thế nào trong từng bước hủy.
4. Áp dụng giải thuật tạo cây nhị phân tìm kiếm cân bằng để tạo cây với khóa nhập vào là như sau:

   7 1 13 10 14 17 11 8 12 16.

   thì hình ảnh cây tạo được như thế nào? Lưu ý vẽ hình minh họa khi thêm từng khóa vào cây.

   Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau: 1, 8, 12 thì cây thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ và giải thích.
5. Xây dựng cấu trúc dữ liệu biểu diễn cây n - phân ($2 < n < 10$). Viết chương trình con duyệt cây n - phân.

5.7.2. Bài tập

6. Cho cây nhị phân tìm kiếm mà mỗi nút là một số nguyên. Hãy viết chương trình để thực hiện những chức năng sau:
   a. Tạo cây nhị phân tìm kiếm bằng ba cách (*Cách 1:* Cho trước một mảng a có n phần tử, hãy tạo một cây nhị phân tìm kiếm có n node, mỗi nút lưu một phần tử của mảng. *Cách 2:* Nhập liệu từ bàn phím. *Cách 3:* Tạo ngẫu nhiên tự động).
   b. Duyệt cây nhị phân tìm kiếm bằng sáu cách: traverseNLR, traverseLNR, traverseLRN, traverseNRL, traverseRNL, traverseRLN.
   c. Thêm một nút có giá trị x làm con trái của nút T.
   d. Thêm một nút có giá trị x làm con phải của nút T.
   e. Thêm một nút có giá trị x vào cây.
   f. Đếm số nút trên cây.
   g. Cho biết các phần tử trên cây có lớn hơn x hay không?
7. Cho cây nhị phân tìm kiếm mà mỗi nút là một phân số. Hãy viết chương trình để thực hiện những chức năng sau:

a. Tạo cây nhị phân tìm kiếm bằng hai cách (nhập liệu từ bàn phím, tạo ngẫu nhiên tự động).

b. Duyệt cây nhị phân tìm kiếm bằng sáu cách: traverseNLR, traverseLNR, traverseLRN, traverseNRL, traverseRNL, traverseRLN.

c. Thêm một nút là phân số x làm con trái của nút T.

d. Thêm một nút là phân số x làm con phải của nút T.

e. Thêm một nút là phân số x vào cây.

f. Đếm số phân số lớn hơn 1.

g. Tối giản tất cả các nút của cây.

8. Ứng dụng cây nhị phân (*cây biểu thức*) để viết chương trình thực hiện việc biểu diễn và tính giá trị của biểu thức.

9. Cho cây nhị phân tìm kiếm như ở bài tập 6. Hãy bổ sung những chức năng sau:

a. Tìm kiếm một nút có giá trị x trên cây hay không?

b. Xóa nút con trái của một nút T.

c. Xóa nút con phải của một nút T.

d. Xóa nút có giá trị x trên cây.

e. Xuất các phần tử theo chiều giảm dần.

f. Tính tổng các nút trong cây (*không đệ quy/đệ quy*).

g. Tính tổng các giá trị dương trên cây.

h. Đếm số giá trị lớn hơn x, nhỏ hơn x, có giá trị trong đoạn [x, y].

i. Xóa toàn bộ cây.

10. Cho cây nhị phân tìm kiếm như ở bài tập 7. Hãy bổ sung những chức năng sau:

a. Tìm kiếm một phân số x trên cây hay không?

b. Xóa một phân số x trên cây.

c. Xóa những phân số lớn hơn 2.

d. Xóa những phân số có mẫu số là số nguyên tố.

e. Xóa toàn bộ danh sách.

11. Tiếp theo bài tập 6. Hãy viết các hàm thực hiện các chức năng sau:

a. Viết hàm xuất các số hoàn thiện trong cây.

b. Viết hàm xuất tất cả các nút trên tầng thứ k của cây. (*)

c. Viết hàm xuất tất cả các nút trên cây theo thứ tự từ tầng 0 đến tầng h - 1 của cây (với h là chiều cao của cây). (*)

d. Đếm số lượng nút lá mà thông tin tại nút đó là giá trị chẵn.

e. Đếm số lượng nút có đúng một con mà thông tin tại nút đó là số nguyên tố.

f. Đếm số lượng nút có đúng hai con mà thông tin tại nút đó là số chính phương.

g. Đếm số lượng nút trên tầng thứ k của cây.

h. Đếm số lượng nút nằm ở tầng thấp hơn tầng thứ k của cây.

i. Đếm số lượng nút nằm ở tầng cao hơn tầng thứ k của cây.

j. Tính tổng các nút trong cây.

k. Tính tổng các nút lá trong cây.

l. Tính tổng các nút có đúng một con.

m. Tính tổng các nút có đúng hai con.

n. Tính tổng các nút lẻ.

o. Tính tổng các nút lá mà thông tin tại nút đó là giá trị chẵn.

p. Tính tổng các nút có đúng một con mà thông tin tại nút đó là số nguyên tố.

q. Tính tổng các nút có đúng hai con mà thông tin tại nút đó là số chính phương.

12. Cho cây nhị phân tìm kiếm như ở bài tập 6. Hãy bổ sung những chức năng sau:

a. Đếm số nút của cây.

b. Đếm số nút lá của cây.

c. Đếm số nút có đúng hai nút con của cây.

d. Đếm số nút chỉ có một nút con của cây.

e. Đếm số nút có con là nút lá của cây.

f. Tính chiều cao của cây.

g. Tìm nút có giá trị lớn nhất của cây.

h. Tính tổng giá trị các nút của cây.

i. Tính tổng giá trị các nút là số nguyên tố của cây.

j. Tính tổng giá trị các nút là số chính phương của cây.

k. Tính tổng giá trị các nút là số hoàn thiện của cây.

l. Tính tổng giá trị các nút là số thịnh vượng của cây.

m. Tính tổng giá trị các nút là số yếu của cây.

13. Tiếp theo bài tập 6. Hãy:

a. Kiểm tra cây nhị phân T có phải là "cây nhị phân tìm kiếm" hay không?

b. Kiểm tra cây nhị phân T có phải là "cây nhị phân cân bằng" hay không?

c. Kiểm tra cây nhị phân T có phải là "cây nhị phân cân bằng hoàn toàn" hay không?

14. Giả sử a là một mảng các số thực đã có thứ tự tăng. Hãy viết hàm tạo một cây nhị phân tìm kiếm có chiều cao thấp nhất từ các phần tử của a.
15. Viết chương trình con đảo nhánh (nút trái và nút phải của một nút cha bất kỳ trên cây được đảo cho nhau) một cây nhị phân.
16. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.
17. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL.
18. Viết chương trình cho phép tạo, tra cứu và sữa chữa từ điển Anh - Việt.

Chương 6. BẢNG BĂM

6.1. Mô tả bảng băm

Trong khoa học máy tính, bảng băm là một cấu trúc dữ liệu sử dụng hàm băm để ánh xạ từ giá trị xác định, được gọi là khóa (ví dụ như tên của một người), đến giá trị tương ứng (ví dụ như số điện thoại của họ). Tư tưởng của bảng băm là dựa vào giá trị các khóa k[1..N], chia các khóa đó ra thành các nhóm. Những khóa thuộc cùng một nhóm có một đặc điểm chung và đặc điểm này không có trong các nhóm khác. Khi có một khóa tìm kiếm X, trước hết ta xác định xem nếu X thuộc vào dãy khóa đã cho thì nó phải thuộc nhóm nào và tiến hành tìm kiếm trên nhóm đó.

Một ví dụ là trong cuốn từ điển Anh – Việt, các bạn sinh viên thường dán 26 mảnh giấy nhỏ, ghi thông tin là các ký tự từ A(a) đến Z(z) vào các trang để đánh dấu trang nào là trang khởi đầu của một đoạn chứa các từ có cùng chữ cái đầu. Để khi tra từ thì chỉ cần tìm trong các trang chứa những từ có cùng chữ cái đầu với từ cần tìm.

Một ví dụ khác, là trên dãy các khóa số tự nhiên, ta có thể chia nó ra làm m nhóm, mỗi nhóm gồm các khóa đồng dư theo môđun m.

Do đó, bảng băm là một mảng kết hợp. Hàm băm được sử dụng để chuyển đổi từ khóa thành chỉ số (giá trị băm) trong mảng lưu trữ các giá trị tìm kiếm.

Trong trường hợp lý tưởng, hàm băm luôn chuyển đổi các khóa khác nhau đến các chỉ số khác nhau. Tuy nhiên trong thực tế, điều này hiếm khi xảy ra (trừ khi các khóa là cố định: không có thêm khóa mới nào được bổ sung vào bảng sau khi tạo bảng). Thay vào đó, hầu hết các thiết kế bảng băm đều giả sử các khóa khác nhau có thể được băm vào cùng một giá trị (gọi là *va chạm* hay *xung đột* băm), và cung cấp cách giải quyết xung đột.

Trong một bảng băm có kích thước đủ lớn, thời gian trung bình cho mỗi lần tra cứu là độc lập với số lượng phần tử trong bảng. Nhiều thiết kế bảng băm cũng cho phép chèn thêm và xóa bỏ tùy ý của cặp khóa - giá trị trong thời gian trung bình (hoặc trừ dần) không đổi cho mỗi thao tác.

Trong nhiều trường hợp, bảng băm có hiệu quả hơn so với cây tìm kiếm hoặc bất kỳ cấu trúc dữ liệu tìm kiếm nào. Vì lý do này, chúng được sử dụng rộng rãi trong nhiều loại phần mềm máy tính, đặc biệt là cho mảng kết hợp, lập chỉ mục cơ sở dữ liệu, tổ chức bộ nhớ đệm, và cấu trúc dữ liệu tập hợp.

Có nhiều cách cài đặt bảng băm:

- **Cách thứ nhất:** Chia dãy khóa thành các đoạn, mỗi đoạn chứa những khóa thuộc cùng một nhóm và ghi nhận lại vị trí các đoạn đó. Để khi có khóa tìm kiếm, có thể xác định được ngay cần phải tìm khóa đó trong đoạn nào.
- **Cách thứ hai:** Chia dãy khóa làm m nhóm, mỗi nhóm là một danh sách nối đơn chứa các giá trị khóa và ghi nhận lại chốt của mỗi danh sách nối đơn. Với một khóa tìm kiếm, ta xác định được phải tìm khóa đó trong danh sách nối đơn nào và tiến hành tìm kiếm tuần tự trên danh sách nối đơn đó. Với cách lưu trữ này, việc bổ sung cũng như loại bỏ một giá trị khỏi tập hợp khóa dễ dàng hơn rất nhiều so với phương pháp trên.
- **Cách thứ ba:** Nếu chia dãy khóa làm m nhóm, mỗi nhóm được lưu trữ dưới dạng cây nhị phân tìm kiếm và ghi nhận lại gốc của các cây nhị phân tìm kiếm đó. Phương pháp này có thể nói là tốt hơn hai phương pháp trên, tuy nhiên dãy khóa phải có mối quan hệ thứ tự toàn phần thì mới làm được.

6.1.1. Mô tả dữ liệu

Bảng băm được mô tả bằng các thành phần sau:

- Có tập khóa (key) của các nút trên bảng băm gọi là tập **K**.
- Có tập các địa chỉ (address) của bảng băm được gọi là tập **A**.
- Có hàm băm (hash function) để ánh xạ một khóa trong tập **K** thành 1 địa chỉ trong tập **A**.

Bảng băm được mô tả bằng hình vẽ như sau:

6.1.2. Các tác vụ trên bảng băm

Bảng băm có thể có các tác vụ sau:

- **Tác vụ khởi động (*Initalize*):** Cấp phát bộ nhớ và khởi động các giá trị ban đầu cho bảng băm.
- **Tác vụ kiểm tra rỗng (*Empty*):** Kiểm tra bảng băm có rỗng hay không.

- **Tác vụ lấy kích thước của bảng băm (*Size*):** Cho biết số phần tử hiện có của bảng băm.
- **Tác vụ tìm kiếm (*Search*):** Đây là một trong những tác vụ thường được sử dụng nhất của bảng băm. Tác vụ này sẽ tìm kiếm các phần tử trong bảng băm dựa vào khóa Key của từng phần tử.
- **Tác vụ thêm một phần tử (*Insert*):** Tác vụ này thêm một phần tử mới vào bảng băm. Sau khi thêm số phần tử hiện có của bảng băm tăng thêm một đơn vị.
- **Tác vụ xoá một phần tử (*Remove*):** Tác vụ này được dùng để xoá một phần tử ra khỏi bảng băm. Sau khi xóa số phần tử hiện có của bảng băm giảm bớt đi một đơn vị.
- **Tác vụ sao chép (*Copy*):** Tác vụ này được dùng để tạo một bảng băm mới từ bảng băm cũ đã có.
- **Tác vụ duyệt bảng băm (*Traverse*):** Tác vụ này dùng để duyệt qua tất cả các phần tử trên bảng băm theo thứ tự địa chỉ từ nhỏ đến lớn.

6.1.3. Các bảng băm thông dụng

Với mỗi loại bảng băm, chúng ta phải xác định tập khóa **K**, xác định tập địa chỉ **A** và xây dựng hàm băm **hashFunction**.

Khi xây dựng hàm băm chúng ta muốn các khóa khác nhau sẽ ánh xạ vào các địa chỉ khác nhau, nhưng thực tế thì thường xảy ra trường hợp các khóa khác nhau lại ánh xạ vào cùng một địa chỉ, chúng ta gọi là xung đột. Do đó khi xây dựng bảng băm chúng ta phải xây dựng phương án giải quyết sự xung đột trên bảng băm.

Các phương pháp xử lý đụng độ:

- Bảng băm đóng, gồm:
  - Phương pháp nối kết trực tiếp (Direct chaining Method).
  - Phương pháp nối kết hợp nhất (Coalesced chaining Method).
- Bảng băm địa chỉ mở (Open - addressing), gồm:
  - Phương pháp dò tuyến tính (Linear probing Method).
  - Phương pháp dò bậc hai (Quadratic probing Method).
  - Phương pháp băm kép (Double hashing Method).

Trong chương này ta sẽ nghiên cứu các bảng băm thông dụng, với mỗi bảng băm có chiến lược giải quyết sự xung đột riêng như sau:

- **Bảng băm với phương pháp nối kết trực tiếp:** Mỗi địa chỉ của bảng băm (gọi là một bucket) tương ứng với một danh sách liên kết. Các phần tử (các nút) bị xung đột được nối kết với nhau trên cùng một danh sách liên kết.
- **Bảng băm với phương pháp nối kết hợp nhất:** Bảng băm loại này được cài đặt bằng danh sách kề, mỗi phần tử (nút) có hai trường: trường Key chứa khóa của nút và trường Next chỉ nút kế bị xung đột. Các nút bị xung đột được nối kết với nhau qua trường liên kết Next.
- **Bảng băm với phương pháp dò tuyến tính:** Ví dụ như khi thêm phần tử vào bảng băm loại này, nếu băm lần đầu bị xung đột thì lần lượt dò địa chỉ kế… cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.
- **Bảng băm với phương pháp dò bậc hai:** Ví dụ như khi thêm nút vào bảng băm loại này, nếu băm lần đầu bị xung đột thì lần lượt dò đến địa chỉ mới, lần dò i ở phần tử cách khoảng i^2 cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.
- **Bảng băm với phương pháp kép:** Bảng băm loại này dùng hai hàm băm khác nhau, băm lần đầu với hàm băm thứ nhất nếu bị xung đột thì xét địa chỉ khác bằng hàm băm thứ hai.

6.1.4. Hàm băm

Hàm băm là hàm biến đổi khóa của phần tử (nút) thành địa chỉ trên bảng băm, là giải thuật nhằm sinh ra các **giá trị băm** tương ứng với mỗi **khối dữ liệu** (có thể là một chuỗi kí tự, một đối tượng trong lập trình hướng đối tượng, v.v...). **Giá trị băm** đóng vai gần như một **khóa** để phân biệt các **khối dữ liệu**, tuy nhiên, người ta chấp **hiện tượng trùng khóa** hay còn gọi là đụng độ và cố gắng cải thiện giải thuật để giảm thiểu sự đụng độ đó.

Hàm băm thường được dùng trong bảng băm nhằm giảm **chi phí tính toán** khi tìm một **khối dữ liệu** trong một tập hợp (nhờ việc so sánh các **giá trị băm** nhanh hơn việc so sánh những **khối dữ liệu** có kích thước lớn).

Vì tính thông dụng của bảng băm mà ngày nay nhiều ngôn ngữ lập trình đều cung cấp thư viện ứng dụng bảng băm, thường gọi là thư viện collection trong đó có các vấn đề như: tập hợp (collection), danh sách (list), bảng (table), ánh xạ (mapping), từ điển (dictionary). Thông thường, các lập trình viên chỉ cần viết hàm băm cho các đối tượng nhằm tích hợp với thư viện bảng băm đã được xây dựng sẵn.

- Khóa của hàm băm có thể là khóa ở dạng số hay dạng chuỗi.
- Địa chỉ được tính ra là các số nguyên trong khoảng 0 đến MAXSIZE – 1, với MAXSIZE là số địa chỉ trên bảng băm.

- Hàm băm thường được dùng ở dạng công thức: Ví dụ như công thức HF(Key) = Key % MAXSIZE, với MAXSIZE là độ lớn của bảng băm.

Một hàm băm tốt phải thoả các yêu cầu sau:

- Tính toán nhanh.
- Giảm thiểu sự xung đột.
- Các khoá được phân bố đều trong bảng (*Phân bố đều các nút trên MAXSIZE địa chỉ khác nhau của bảng băm*).
- Xử lý được các loại khóa có kiểu dữ liệu khác nhau.

6.1.5. Ưu điểm của bảng băm

Bảng băm là một cấu trúc dữ liệu dung hoà tốt giữa thời gian truy xuất và dung lượng bộ nhớ:

- Nếu không có sự giới hạn về bộ nhớ thì chúng ta có thể xây dựng bảng băm với mỗi khóa ứng với một địa chỉ với mong muốn thời gian truy xuất tức thời.
- Nếu dung lượng của bộ nhớ có giới hạn thì tổ chức một số khóa có cùng địa chỉ, lúc này thời gian truy xuất có bị giảm đi.

6.1.6. Ứng dụng của bảng băm

Các hàm băm được ứng dụng trong nhiều lĩnh vực, chúng thường được thiết kế phù hợp với từng ứng dụng. Ví dụ, các hàm băm mật mã học giả thiết sự tồn tại của một đối phương - người có thể cố tình tìm các dữ liệu vào với cùng một giá trị băm. Một hàm băm tốt là một phép biến đổi "một chiều", nghĩa là không có một phương pháp thực tiễn để tính toán được dữ liệu vào nào đó tương ứng với giá trị băm mong muốn, khi đó việc giả mạo sẽ rất khó khăn. Một hàm một chiều mật mã học điển hình không có tính chất hàm đơn ánh và tạo nên một hàm băm hiệu quả; một hàm trapdoor mật mã học điển hình là hàm đơn ánh và tạo nên một hàm ngẫu nhiên hiệu quả.

Bảng băm được ứng dụng nhiều trong thực tế, rất thích hợp khi tổ chức dữ liệu có kích thước lớn và được lưu trữ ở bộ nhớ ngoài.

Bảng băm, một ứng dụng quan trọng của các hàm băm, cho phép tra cứu nhanh một bản ghi dữ liệu nếu cho trước khóa của bản ghi đó (Lưu ý: các khóa này thường không bí mật như trong mật mã học, nhưng cả hai đều được dùng để "mở khóa" hoặc để truy nhập thông tin.) Ví dụ, các khóa trong một từ điển điện tử Anh - Anh có thể là các từ tiếng Anh, các bản ghi tương ứng với chúng chứa các định nghĩa. Trong trường hợp này, hàm băm phải ánh xạ các xâu chữ cái tới các chỉ mục của mảng nội bộ của bảng băm.

Các hàm băm dành cho việc phát hiện và sửa lỗi tập trung phân biệt các trường hợp mà dữ liệu đã bị làm nhiễu bởi các quá trình ngẫu nhiên. Khi các hàm băm được dùng cho các giá trị tổng kiểm, giá trị băm tương đối nhỏ có thể được dùng để kiểm chứng rằng một file dữ liệu có kích thước tùy ý chưa bị sửa đổi. Hàm băm được dùng để phát hiện lỗi truyền dữ liệu. Tại nơi gửi, hàm băm được tính cho dữ liệu được gửi, giá trị băm này được gửi cùng dữ liệu. Tại đầu nhận, hàm băm lại được tính lần nữa, nếu các giá trị băm không trùng nhau thì lỗi đã xảy ra ở đâu đó trong quá trình truyền. Việc này được gọi là kiểm tra dư (*redundancy check*).

Các hàm băm còn được ứng dụng trong việc nhận dạng âm thanh, chẳng hạn như xác định xem một file MP3 có khớp với một file trong danh sách một loại các file khác hay không.

Ví dụ: *Giải thuật tìm kiếm chuỗi ký tự Rabin - Karp* là một giải thuật tìm kiếm chuỗi kí tự tương đối nhanh, với thời gian chạy trung bình **O**(n).

Giải thuật mang tên hai nhà khoa học phát minh ra nó Michael O. Rabin (sinh năm 1931, người Đức) and Richard M. Karp (sinh năm 1931, người Mỹ), đều được giải Turing Award, giải thương uy tín nhất trong ngành khoa học máy tính và công nghệ thông tin mang tên nhà khoa học máy tính lừng danh người Anh Alan Turing. Tư tưởng chính của phương pháp này là sử dụng phương pháp băm (hashing). Tức là mỗi một chuỗi sẽ được gán với một giá trị của hàm băm (hash function), ví dụ chuỗi "hello" được gán với giá trị 5 chẳng hạn, và hai chuỗi được gọi là bằng nhau nếu giá trị băm của nó bằng nhau. Như vậy thay vì việc phải đối sánh các chuỗi con của T với mẫu P, ta chỉ cần so sánh giá trị hàm băm của chúng và đưa ra kết luận.

Đặc tả giải thuật Rabin - Karp như sau:

```
1.  function algorithm_Rabin_Karp(string T[1..n], string P[1..m])
2.      hsub = hashFunction(P[1..m]) //giá trị băm của chuỗi con P
3.      hs = hashFunction(T[1..m])        //giá trị băm của chuỗi mẹ T
4.      for i from 1 to n-m+1
5.          if hs = hsub
6.              if T[i..i+m-1] = P
7.                  return i
8.      hs = hashFunction(T[i+1..i+m]) //giá trị băm của T[i+1..i+m]
9.      return not found
10. end function
```

Vấn đề đặt ra ở đây là khi có quá nhiều chuỗi sẽ tồn tại các trường hợp các chuỗi khác nhau có giá trị băm giống nhau, do đó khi tìm thấy hai chuỗi có giá trị băm giống nhau vẫn phải kiểm tra lại xem chúng có thực sự bằng nhau hay không (dòng số 6), may

mắn là trường hợp này rất ít xảy ra với một hàm băm thiết kế đủ tốt.

Phân tích giải thuật ta thấy: dòng số 2, 3, 6, 8 có độ phức tạp là O(m), nhưng dòng số 2, 3 chỉ thực hiện duy nhất một lần, dòng số 6 chỉ thực hiện khi giá trị băm bằng nhau (rất ít), chủ yếu là dòng số 8 sẽ quyết định độ phức tạp của giải thuật. Bởi khi tính giá trị băm cho T[i+1..i+m] ta mất thời gian là O(m), công việc này được thực hiện trong n-m+1 lần.

Như vậy ta phải tính lại giá trị hs trong thời gian hằng số (constant time), cách giải quyết ở đây là tính giá trị băm của T[i+1..i+m] dựa vào giá trị băm của T[i..i+m-1] bằng cách sử dụng cách băm tròn (rolling hash, là cách băm mà giá trị đầu vào được băm với một kích thước cửa số cổ định trượt trên độ dài của giá trị cần băm). Cụ thể trong bài toán này, ta sử dụng công thức sau để tính giá trị băm tiếp theo trong một khoảng thời gian hằng số: hashFunction(T[i+1…i+m]) = hashFunction(T[i+1…i+m-1]) - ASCII(T[i]) + ASCII (T[i+m]), trong đó ASCII(i) là mã ASCII của ký tự i. Như vậy trong trường hợp này độ phức tạp chỉ còn là O(n).

Đó là một cách băm đơn giản, dưới đây sẽ trình bày một hàm băm phức tạp và tốt hơn cho các trường hợp dữ liệu lớn. Đó là sử dụng các số nguyên tố lớn. Ví dụ như chuỗi "hi" băm bằng số nguyên tố 101 sẽ có giá trị băm là 104 × 1011 + 105 × 1010 = 10609 (ASCII của ký tự 'h' là 104 và của ký tự 'i' là 105).

Thêm nữa, ta có thể tính giá trị băm của một chuỗi con dựa vào các chuỗi con trước nó, ví dụ như ta có chuỗi "abracadabra", ta cần tìm một mẫu tìm kiếm có độ dài là 3. Ta có thể tính giá trị băm của chuỗi "bra" dựa vào giá trị băm của chuỗi "abr" (chuỗi con trước nó) bằng cách lấy giá trị băm của "abr" trừ đi giá trị băm của ký tự 'a' đầu tiên (ví dụ như 97 × 1012 (97 là giá trị ASCII của ký tự 'a' và 101 là số nguyên tố đang sử dụng) và cộng thêm giá trị băm cảu ký tự 'a' cuối cùng trong chuỗi "bra" (ví dụ như 97 × 1010 = 97).

Giải thuật Rabin-Karp sử dụng hàm băm tính toán nhanh, không phụ thuộc vào chiều dài chuỗi cần tìm kiếm Tiết kiệm bộ nhớ vì không lưu trữ lại nhiều kết quả tìm kiếm trước đó. Không sử dụng các phương pháp nhận biết, loại trừ các kết quả xấu cho các bước tìm kiếm sau.

6.2. Bảng băm với phương pháp kết nối trực tiếp

6.2.1. Mô tả

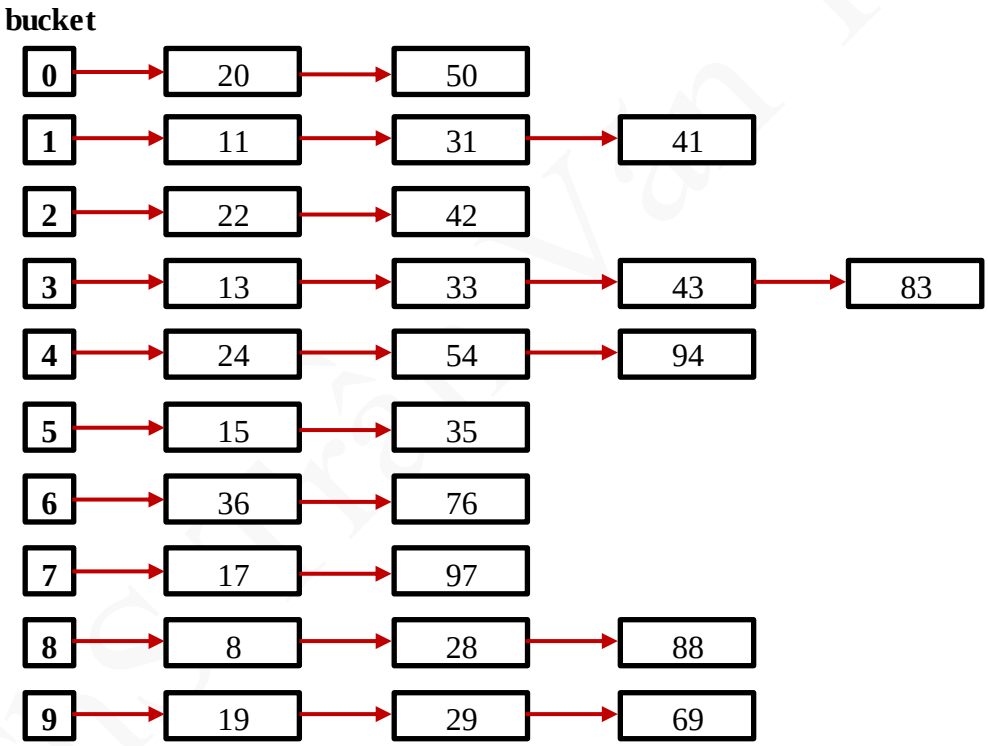
Bảng băm được cài đặt bằng danh sách liên kết, các nút trên bảng băm được băm thành MAXSIZE danh sách liên kết (từ danh sách 0 đến danh sách MAXSIZE - 1). Các nút bị xung đột tại địa chỉ i được nối kết trực tiếp với nhau qua danh sách liên kết thứ i. Chẳng hạn, với MAXSIZE = 10, các phần tử có hàng đơn vị là 5 sẽ được băm vào danh sách liên kết thứ i = 5.

Khi thêm một phần tử (nút) có khóa Key vào bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng 0 đến MAXSIZE - 1 ứng với danh sách liên kết thứ i mà nút này sẽ được thêm vào.

Khi tìm kiếm một phần tử (nút) có khóa Key trên bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng từ 0 đến MAXSIZE - 1 ứng với danh sách liên kết thứ i có thể chứa nút, việc tìm kiếm nút trên bảng băm quy về bài toán tìm kiếm trên danh sách liên kết.

Sau đây để minh họa cho vấn đề vừa nêu, xét bảng băm có cấu trúc như sau:

- Tập khóa K: là tập số tự nhiên
- Tập địa chỉ MAXSIZE: có 10 địa chỉ từ 0 → 9
- Chọn hàm băm là **HF(Key) = Key % 10**.



Bảng băm dùng phương pháp nối kết trực tiếp

Hình trên minh họa bảng băm vừa miêu tả. Theo hình vẽ, bảng băm đã "*băm*" phần tử trong tập khóa K theo 10 danh sách liên kết khác nhau, mỗi danh sách liên kết gọi là một bucket.

- **bucket 0:** gồm những phần tử có khóa tận cùng bằng 0.
- **bucket i (i=1..9):** gồm những phần tử có khóa tận cùng bằng i. Để giúp việc truy xuất bảng băm dễ dàng, các phần tử trên các bucket cần thiết được tổ chức theo một thứ tự, chẳng hạn như từ nhỏ đến lớn theo khóa.
- Khi khởi động bảng băm, con trỏ đầu của các bucket là NULL.

Theo cấu trúc này, với tác vụ insert, hàm băm sẽ được dùng để tính địa chỉ của khóa Key của phần tử cần thêm, tức là xác định được bucket chứa phần tử và đặt phần

tử cần thêm vào bucket này. Với tác vụ search, hàm băm sẽ được dùng để tính địa chỉ của khóa Key của phần tử cần tìm và tìm trên bucket tương ứng.

6.2.2. Cài đặt

6.2.2.1. Khai báo cấu trúc bảng băm

```
#define MAXSIZE 10
struct HashNode
{
    int Key;
    HashNode* Next;
};
HashNode* bucket[MAXSIZE];
```

6.2.2.2. Hàm băm

```
int hashFunction(int Key)
{
    return (Key % MAXSIZE);
}
```

6.2.2.3. Tìm kiếm một phần tử trên bảng băm

```
int search(int k)
{
    int b = hashFunction(k);
    HashNode* p = bucket[b];
    while(k > p→Key && p != NULL)
        p = p→Next;
    if(p == NULL || k != p→Key)
        return -1;
    else
        return b;
}
```

6.2.2.4. Thêm vào một phần tử

```
void insertAfter(HashNode* p, int k)
{
    if(p == NULL)
        printf("Khong them vao Node moi duoc");
    else
    {
        HashNode* q = new HashNode;
        q→Key = k;
        q→Next = p→Next;
        p→Next = q;
    }
}
// tac vu nay chi su dung khi them vao mot bucket co thu tu
void place(int b, int k)
{
```

```
    HashNode *p, *q;
    q = NULL;
    for(p = bucket[b]; p != NULL && k > p→Key; p = p→Next)
        q = p;
    if(q == NULL)
        push(b, k);
    else
        insertAfter(q, k);
}
//them mot Node co khoa la k vao trong bang bam
void insert(int k)
{
    int b = hashFunction(k);
    place(b, k);
}
```

6.2.2.5. Xoá một phần tử

```
int deleteAfter(HashNode* p)
{
    HashNode* q;
    int k;
    if(p == NULL || p→Next == NULL)
    {
        printf("\n Khong xoa Node duoc");
        return 0;
    }
    q = p→Next;
    k = q→Key;
    p→Next = q→Next;
    freeNode(q);
    return k;
}
//Xoa mot phan tu co khoa k ra khoi bang bam
void remove(int k)
{
    int b;
    HashNode* p, *q;
    b = hashFunction(k);
    p = bucket[b];
    q = p;
    while(p != NULL && p→Key != k)
    {
        q = p;
        p = p→Next;
    }
    if(p == NULL)
        printf("\n Khong co Node co khoa la: %d", b);
    else if(p == bucket[b])
        pop(b);
    else
```

```
            deleteAfter(q);
    }
```

6.3. Bảng băm với phương pháp kết nối hợp nhất

6.3.1. Mô tả

Bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có MAXSIZE nút. Các nút bị xung đột địa chỉ được nối kết với nhau qua một danh sách liên kết.

Mỗi nút của bảng băm là một mẫu tin có hai trường:

- Trường **Key**: chứa khóa của nút.
- Trường **Next**: con trỏ chỉ nút kế tiếp nếu có xung đột.

Khi khởi động bảng băm thì tất cả trường Key được gán giá trị là NULLKEY, tất cả các trường Next được gán là -1.

Hình vẽ sau mô tả bảng băm ngay sau khi khởi động:

0	**NULLKEY**	-1
1	**NULLKEY**	-1
2	**NULLKEY**	-1
3	**NULLKEY**	-1
…	**…**	…
MAXSIZE - 1	**NULLKEY**	-1

Khi thêm một nút có khóa Key vào bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng từ 0 đến MAXSIZE - 1.

- Nếu chưa bị xung đột thì thêm nút mới tại địa chỉ i này.
- Nếu bị xung đột thì nút mới được cấp phát là nút trống phía cuối mảng. Cập nhật liên kết Next sao cho các nút bị xung đột hình thành một danh sách liên kết.

Khi tìm một nút có khóa Key trong bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng từ 0 đến MAXSIZE - 1, tìm nút khóa Key trong danh sách liên kết xuất phát từ địa chỉ i.

Minh họa:

Sau đây là minh họa cho bảng băm có tập khóa là số tự nhiên, tập địa chỉ có 10 địa chỉ (MAXSIZE = 10), chọn hàm băm f(Key) = Key % 10. Hình vẽ sau đây minh họa cho tiến trình thêm các nút 10, 15, 26, 30, 25, 35 vào bảng băm.

- Hình (a): Sau khi thêm 3 nút 10, 15, 26 vào bảng băm – lúc này chưa bị xung đột.
- Hình (b): Thêm nút 30 vào bảng băm - bị xung đột tại địa chỉ 0 - nút 30 được cấp phát tại địa chỉ 9, trường Next của nút tại địa chỉ 0 được gán là 9.
- Hình (c): Thêm nút 25 vào bảng băm - bị xung đột tại địa chỉ 5 - nút 25 được cấp phát tại địa chỉ 8, trường Next của nút tại địa chỉ 5 được gán là 8.
- Hình (d): Thêm nút 35 vào bảng băm - bị xung đột tại địa chỉ 5 và địa chỉ 8 - nút 35 được cấp phát tại địa chỉ 7, trường Next của nút tại địa chỉ 8 được gán là 7.

(a)

0	**10**	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	NULLKEY	-1
4	NULLKEY	-1
5	**15**	-1
6	**26**	-1
7	NULLKEY	-1
8	NULLKEY	-1
9	NULLKEY	-1

(b)

0	**10**	9
1	NULLKEY	-1
2	NULLKEY	-1
3	NULLKEY	-1
4	NULLKEY	-1
5	**15**	-1
6	**26**	-1
7	NULLKEY	-1
8	NULLKEY	-1
9	**30**	-1

(c)

0	**10**	9
1	NULLKEY	-1
2	NULLKEY	-1
3	NULLKEY	-1
4	NULLKEY	-1
5	**15**	8
6	**26**	-1
7	NULLKEY	-1
8	**25**	-1
9	**30**	-1

(d)

0	**10**	9
1	NULLKEY	-1
2	NULLKEY	-1
3	NULLKEY	-1
4	NULLKEY	-1
5	**15**	8
6	**26**	-1
7	**35**	-1
8	**25**	7
9	**30**	-1

Hình minh họa việc thêm các khóa vào bảng băm

6.3.2. Cài đặt

6.3.2.1. Khai báo cấu trúc bảng băm

```
#define TRUE 1
#define FALSE 0
#define NULLKEY -1
#define MAXSIZE 100
struct HashNode
{
    int Key;
    int Next;
};

HashNode HashTable[MAXSIZE];
int avail;
```

6.3.2.2. Tác vụ khởi động cho bảng băm

```
void initialize()
{
```

```
    for(int i = 0; i < MAXSIZE; i++)
    {
        HashTable[i].Key = NULLKEY;
        HashTable[i].Next = -1;
    }
    avail = MAXSIZE-1;
}
```

6.3.2.3. *Kiểm tra bảng băm rỗng*

```
// Kiem tra bang bam co rong hay khong?
int isEmpty()
{
    for(int i = 0; i < MAXSIZE; i++)
    {
        if (HashTable[i].Key != NULLKEY)
            return FALSE;
    }
    return TRUE;
}
```

6.3.2.4. *Tác vụ tìm kiếm*

```
// Tim mot khoa co trong bang bam hay khong,
// neu tim thay tra ve dia chi, khong thay tra ve MAXSIZE
int search(int k)
{
    int i = hashFunction(k);
    while(k != HashTable[i].Key && i != -1)
        i = HashTable[i].Next;
    if(k == HashTable[i].Key)
        return i;
    else
        return MAXSIZE;
}
```

6.3.2.5. *Chọn nút con để cập nhật khi xảy ra xung đột*

```
// Chon Node con trong phia duoi bang hash de cap nhat khi xay ra xung dot
int getEmpty()
{
    while(HashTable[avail].Key != NULLKEY)
        avail--;
    return avail;
}
```

6.3.2.6. *Tác vụ thêm một phần tử vào bảng băm*

```
// Them mot Node co khoa k vao bang bam
int insert(int k)
{
    int i, j;
    i = search(k);
    if(i != MAXSIZE)
```

```
    {
        printf("\n khoa %d bi trung, khong them vao duoc", k);
        return i;
    }
    i = hashFunction(k);
    while(HashTable[i].Next >= 0)
        i = HashTable[i].Next;
    if(HashTable[i].Key == NULLKEY)
        j = i;     // lần đầu tiên, khong co su dung do
    else
    {
        j = getEmpty();
        if(j < 0)
        {
            printf("\n Bang bam bi day khong them vao duoc");
            return j;
        }
        else
        {
            HashTable[i].Next = j;
        }
    }
    HashTable[j].Key = k;
    return j;
}
```

6.3.2.7. *Tác vụ duyệt bảng băm*

```
// Xem chi tiet thong tin bang bam
void viewTable()
{
    for(int i = 0; i < MAXSIZE; i++)
    {
        printf("\nTable[%2d]: %4d %4d", i, HashTable[i].Key, HashTable[i].Next);
    }
}
```

6.4. Bảng băm với phương pháp dò tuyến tính

6.4.1. Mô tả

Bảng băm trong trường hợp này được cài đặt bằng một danh sách kề có MAXSIZE nút, mỗi nút của bảng băm là một mẫu tin có một trường Key để chứa khóa của nút.

Khi thêm nút có khóa Key vào bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng từ 0 đến MAXSIZE - 1:

- Nếu chưa bị xung đột thì thêm nút mới tại địa chỉ i này.
- Nếu bị xung đột thì hàm băm lần 1 f1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm lại lần 2 f2 sẽ xét địa chỉ kế tiếp nữa… quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm nút vào địa chỉ này.

Khi tìm một nút có khóa Key trong bảng băm, hàm băm f(Key) sẽ xác định địa chỉ i trong khoảng từ 0 đến MAXSIZE - 1, tìm nút khóa Key trong khối đặc chứa các nút xuất phát từ địa chỉ i.

Hàm băm lại của phương pháp dò tìm tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại được biểu diễn bằng công thức sau: **f(Key) = (f(Key) + i) % MAXSIZE**

Minh họa:

Sau đây là minh họa cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ, chọn hàm băm f(Key) = Key %10.

- Hình vẽ sau miêu tả tiến trình thêm các nút 32, 53, 22, 92, 17, 34 vào bảng băm.
- Hình (a): Sau khi thêm 2 nút 32 và 53 vào bảng băm - lúc này chưa bị xung đột.
- Hình (b): Thêm nút 22 và 92 vào bảng băm - bị xung đột tại địa chỉ 2, nút 22 được cấp phát tại địa chỉ 4, nút 92 được cấp phát tại địa chỉ 5.
- Hình (c): thêm nút 17 và 34 vào bảng băm - nút 17 không bị xung đột cấp phát tại địa chỉ 7, nút 34 bị xung đột tại địa chỉ 4, được cấp tại địa chỉ 6.

(a)

0	NULLKEY
1	NULLKEY
2	**32**
3	**53**
4	NULLKEY
5	NULLKEY
6	NULLKEY
7	NULLKEY
8	NULLKEY
9	NULLKEY

(b)

0	NULLKEY
1	NULLKEY
2	**32**
3	**53**
4	**22**
5	**92**
6	NULLKEY
7	NULLKEY
8	NULLKEY
9	NULLKEY

(c)

0	NULLKEY
1	NULLKEY
2	**32**
3	**53**
4	**22**
5	**92**
6	**34**
7	**17**
8	NULLKEY
9	NULLKEY

6.4.2. Cài đặt

6.4.2.1. Khai báo cấu trúc bảng băm và định nghĩa các hằng số

```
#define TRUE 1
#define FALSE 0
#define NULLKEY -1
#define MAXSIZE 100

struct HashNode
{
    int Key;
};
```

```
HashNode HashTable[MAXSIZE];
int N;
```

6.4.2.2. Hàm băm

```
int hashFunction(int Key)
{
    return (Key % MAXSIZE);
}
```

6.4.2.3. Tác vụ khởi động

```
void initialize()
{
    for(int i = 0; i < MAXSIZE; i++)
    {
        HashTable[i].Key = NULLKEY;
    }
    N = 0;
}
```

6.4.2.4. Kiểm tra bảng băm có rỗng hay không

```
int isEmpty()
{
    if(N == 0)
        return TRUE;    //Nếu bảng băm rỗng
    else
        return FALSE;   //Nếu bảng băm không rỗng
}
```

6.4.2.5. Kiểm tra bảng băm có đầy hay không

```
int isFull()
{
    if(N == MAXSIZE - 1)
        return TRUE;    //Nếu bảng băm đầy
    else
        return FALSE;   //Nếu bảng băm chưa đầy
}
```

6.4.2.6. Tác vụ tìm kiếm

```
//Tac vu tim 1 Key, tim thay tra ve index, khong thay tra ve MAXSIZE
int search(int k)
{
    int i = hashFunction(k);
    while(HashTable[i].Key != k && HashTable[i].Key != NULLKEY)
    {
        i = i + 1;
        if(i >= MAXSIZE)
            i = i - MAXSIZE;
    }
    if(HashTable[i].Key == k)
```

```
        return i;
    else
        return MAXSIZE;
}
```

6.4.2.7. *Tác vụ thêm một phần tử vào bảng băm*

```
// Them khoa k vao bang bam
int insert(int k)
{
    if(isFull())
    {
        printf("\n Bang bam bi day, khong them vao duoc");
        return MAXSIZE;
    }
    int i = hashFunction(k);
    while(HashTable[i].Key != NULLKEY)
    {
        i++;
        if(i >= MAXSIZE)
        i = i - MAXSIZE;
    }
    HashTable[i].Key = k;
    N++;
    return i;
}
```

6.4.2.8. *Tác vụ duyệt bảng băm*

```
// Xem chi tiet thong tin bang bam
void viewTable()
{
    for(int i = 0; i < MAXSIZE; i++)
    {
        printf("\n Table[%2d]: %4d", i, HashTable[i].Key);
    }
}
```

6.5. Bài tập

1. Viết một chương trình hiện thực từ điển Anh - Việt, chương trình có cài đặt bảng băm với phương pháp nối kết trực tiếp. Mỗi nút của bảng băm có khai báo các trường sau:

   – Trường word là khóa chứa một từ tiếng anh.
   – Trường mean là nghĩa tiếng Việt.
   – Trường Next là con trỏ chỉ nút kế bị xung đột cùng địa chỉ.

   Tập khóa là một chuỗi tiếng anh, tập địa chỉ có 26 chữ cái. Chọn hàm băm sau cho khóa bắt đầu bằng ký tự a được băm vào địa chỉ 0, b băm vào địa chỉ 1, …, z băm vào địa chỉ 25. Chương trình có những chức năng như sau:

- Nhập vào một từ.
- Xem từ điển theo ký tự đầu.
- Xem toàn bộ từ điển.
- Tra từ điển.
- Xoá một từ.
- Xóa toàn bộ từ điển.
- Thoát khỏi chương trình.

2. Viết chương trình xem thông tin về một sinh viên qua mã số sinh viên.

Thông tin về tất cả sinh viên chứa trong tập tin văn bản, mỗi sinh viên chiếm một dòng văn bản gồm MSSV, họ, tên và điểm các môn học. Chương trình sẽ đọc tập tin văn bản này để tạo ra bảng băm. Mỗi nút của bảng băm bao gồm trường MSSV dùng làm khóa, trường line cho biết vị trí dòng văn bản của sinh viên. Khi truy xuất thông tin về một sinh viên chúng ta nhập MSSV, qua bảng băm chúng ta xác định được vị trí dòng văn bản và đọc thông tin sinh viên qua dòng văn bản này.

Chương trình có các chức năng như sau:

- Xem tất cả MSSV
- Xem MSSV trong khoảng từ … đến …
- Xem thông tin về sinh viên qua MSSV.

PHỤ LỤC

1. Sinh dữ liệu vào/ra

1.1. Các phương pháp sinh dữ liệu vào/ra

Hầu hết các bài toán trong tin học đều đòi hỏi dữ liệu vào và ra. Người ta thường dùng ba phương pháp sinh và nạp sau đây:

- Nạp dữ liệu trực tiếp từ bàn phím. Phương pháp này được dùng khi có dữ liệu nhập ít.
- Sinh dữ liệu thông qua hàm random. Phương thức này nhanh chóng và tiện lợi, nếu khéo tổ chức có thể sinh ngẫu nhiên được các dữ liệu đáp ứng một số điều kiện định trước.
- Đọc dữ liệu từ một tập tin. Phương thức này khá tiện lợi khi phải chuẩn bị trước những dữ liệu phức tạp. Kết quả thực hiện chương trình cũng thường được thông báo trực tiếp trên màn hình hoặc ghi vào tập tin văn bản.

1.2. Bài tập

1. Sinh ngẫu nhiên một mảng a gồm n số nguyên nằm trong khoảng $\overline{-k,...,k}$, với k > 0.
2. Sinh ngẫu nhiên một mảng a gồm n số nguyên được sắp không giảm.
3. Sinh ngẫu nhiên một mảng a gồm n số nguyên sao cho phần tử sau luôn là bội số của phần tử trước.
4. Sinh ngẫu nhiên một mảng hai chiều a[m][n] gồm các số nguyên sao cho các phần tử đối xứng qua đường chéo chính, tức là a[i][j] = a[j][i].
5. Độ cao của một số tự nhiên là tổng các chữ số của số đó. Phát sinh các số tự nhiên có tối đa năm chữ số và có độ cao h cho trước và ghi kết quả vào một tập tin văn bản có tên cho trước.
6. Một tập tin văn bản fn có ghi sơ đồ của một vùng biển hình chữ nhật chiều ngang 250 kí tự, chiều dài không hạn chế. Trên biển có các con tàu hình chữ nhật chứa các kí tự 1, vùng nước được biểu thị qua các dấu cách. Biết rằng các con tàu không dính nhau, hãy đếm số lượng tàu.

2. Các bước để giải bài toán tin học

- **Bước đầu tiên:** Là bước quan trọng nhất là hiểu rõ nội dung của bài toán. Để hiểu rõ bài toán theo cách tiếp cận tin học ta phải gắng xây dựng một số thí dụ phản ánh đúng các yêu cầu đề ra của đầu bài rồi thử giải các thí dụ đó để hình thành dần những hướng đi của giải thuật.

- **Bước thứ hai:** Là dùng một ngôn ngữ quen thuộc, tốt nhất là ngôn ngữ toán học đặc tả các đối tượng cần xử lý ở mức độ trừu tượng, lập các tương quan, xây dựng các hệ thức thể hiện các qua hệ giữa các đại lượng cần xử lý.
- **Bước thứ ba:** Là xác định cấu trúc dữ liệu để biểu diễn các đối tượng cần xử lý cho phù hợp với các thao tác của giải thuật. Trong những bước tiếp theo ta tiếp tục làm mịn dần các đặc tả theo trình tự từ trên xuống, từ trừu tượng hóa đến cụ thể, từ đại thể đến chi tiết.
- **Bước cuối cùng:** Là sử dụng ngôn ngữ lập trình đã chọn để viết chương trình cho hoàn chỉnh. Ở bước này ta dùng kỹ thuật đi từ dưới lên, từ những thao tác nhỏ đến những thao tác tổ hợp.

Sau khi nhận được chương trình ta cho chương trình chạy thử với các dữ liệu lấy từ các thí dụ đã xây dựng ở bước đầu tiên. Cuối cùng là việc xây dựng thủ tục một cách khoa học và có chủ đích nhằm kiểm tra tính tin cậy của chương trình thu được và thực hiện một số cải tiến.

Ví dụ: Tìm tất cả các cặp số có hai chữ số sao cho chúng là hai số nguyên tố cùng nhau.

Hiểu đầu bài: Ta kí hiệu (a, b) là ước chung lớn nhất (UCLN) của hai số tự nhiên a và b. Khi đó ta có một số ví dụ như sau:

(23, 21) = 1, vậy (23, 21) là cặp số có 2 chữ số cần tìm.
(21, 14) = 7, vậy (21, 14) không là cặp số có 2 chữ số cần tìm.

Đặc tả: Gọi số có hai chữ số lần lượt là a, b. Ta đã có giải thuật Euclid để tìm cặp số nguyên tố cùng nhau.

- **Bước 1:** Ta có a, b, và r = a mod b.
- **Bước 2:** Sau đó: a = b; b = r.
- **Bước 3:**
  + Nếu r != 0 thì: Quay lại Bước 1.
  + Ngược lại: Sang Bước 4.
- **Bước 4:**
  + Nếu b = 1 thì: Cặp số trên là nguyên tố cùng nhau.
  + Ngược lại: Cặp số trên không là nguyên tố cùng nhau.

Biểu diễn dữ liệu ta dùng hai số nguyên để biểu số cặp số trên, và một số nguyên để lưu trữ phần dư khi chia a cho b.

int a, b, r;

Viết chương trình

```
1.  int kiemTraNguyenToCungNhau(int a, int b)
2.  {
3.      int r = 1;
4.      a = abs(a);
5.      b = abs(b);
6.      while(r)
7.      {
8.          r = a % b; //r = a mod b
9.          a = b;
10.         b = r;
11.     }
12.     return b;
13. }
```

Bài tập

1. Tìm các số tự nhiên lẻ có ba chữ số. Ba chữ số này theo trật tự viết từ trái qua phải tạo thành một cấp số cộng.
2. Tìm các số tự nhiên có ba chữ số. Ba chữ số này theo trật tự viết từ trái qua phải tạo thành một cấp số nhân.
3. Phát sinh ngẫu nhiên n số nguyên không âm cho mảng nguyên a.
4. Tìm cách chia mảng a gồm n số nguyên dương cho trước thành hai đoạn có tổng các phần tử trong mỗi đoạn bằng nhau.
5. Tìm cách chia mảng a gồm n số nguyên dương cho trước thành hai đoạn có tổng các phần tử trong đoạn này gấp k lần tổng các phần tử trong đoạn kia, k nguyên dương.

3. Đệ quy

3.1. Khái niệm đệ quy

Một khái niệm X có tính đệ quy nếu trong định nghĩa của X có sử dụng ngay chính khái niệm X.

Ví dụ:

Phép cộng dồn có tính đệ quy

i = 0;

i = i + 1;

Phép tính giai thừa có tính đệ quy

0! = 1! = 1;

n! = n * (n – 1)!;

Hàm (*Function*) có thể có lời gọi của chính nó. Tính chất này được gọi là tính đệ quy.

Ví dụ:

```
1. long int tinhGiaiThua(int n)
2. {
3.       if(n == 0)
4.            return 1;
5.       else
6.            return (n * tinhGiaiThua(n – 1));
7. }
```

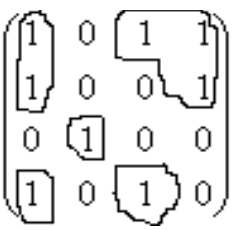
3.2. Nhận xét

- ***Ưu điểm:*** Dùng đệ quy sẽ làm cho chương trình ngắn ngọn, phương pháp giải quyết bài toán trở nên đơn giản.
- ***Nhược điểm:*** Có thể tốn bộ nhớ và thời gian nhiều hơn so với cách không dùng đệ quy.

3.3. Bài tập

1. Lập chương trình tính **100!**
2. Lập chương trình tính $\mathbf{7^{50}}$.
3. Đếm số con tàu trên biển.

Ví dụ: Cho ma trận sau:



những số 1 đứng kề nhau gộp lại là một con tàu. Vậy ma trận trên có tới 5 con tàu (*như hình trên*).

4. Bài toán tháp Hà nội.

Cho **3** cọc và **n** đĩa (*lồng vào cọc*) đặt ở cọc số 1 và theo thứ tự đĩa lớn nằm dưới đĩa nhỏ nằm trên. Hãy chuyển **n** đĩa đó sang cọc số 3 sao cho thứ tự vẫn được bảo toàn. Được sử dụng cọc số 2 để làm trung gian và thỏa mãn yêu cầu khi chuyển đĩa nhỏ được đặt trên đĩa lớn. Mỗi lần chỉ được chuyển một đĩa. Hãy lập chương trình mô tả bài toán đó.

4. Khử đệ quy

Khử đệ quy là việc thay vì viết hàm đệ quy ta sẽ viết hàm đó thành hàm không đệ quy.

4.1. Ví dụ

Tính giai thừa của một số n nguyên bất kỳ.

```
1. int tinhGiaiThua(int n)
2. {
3.     long int s = 1;
4.     for(int i = 1; i <= n; i++)
5.         s = s * i; //s *= i;
6.     return s;
7. }
```

4.2. Bài tập

1. Xem lại phần tìm một nút trên cây nhị phân tìm kiếm có hàm viết khử đệ quy.
2. Khử đệ quy của hàm in các nút trên cây theo thứ tự: traverseNLR, traverseLNR, traverseLRN, traverseNRL, traverseRNL, traverseRLN. (*gợi ý sử dụng Stack và Queue*).

5. Khử đệ quy đuôi

Khử đệ quy đuôi là quá trình *khử lời gọi hàm đệ quy cuối* của hàm đệ quy.

5.1. Ví dụ

Xuất ra màn hình nội dung các nút trên cây theo thứ tự traverseNLR.

Hàm đệ quy:

```
1. void traverseNLR(TNode* root)
2. {
3.     if(root == NULL) return;
4.     printf("%4d", root→Info);
5.     traverseNLR(root→Left);
6.     traverseNLR(root→Right);
7. }
```

Hàm khử đệ quy đuôi:

```
1. void traverseNLR(TNode* root)
2. {
3.       while(root != NULL)
4.       {
5.            printf("%4d", root→Info);
6.            traverseNLR(root→Left);
7.            root = root→Right;
8.       }
9. }
```

5.2. Bài tập

1. Viết hàm khử đệ quy đuôi của việc in ra các nút trên cây theo traverseLNR, traverseLRN.
2. Khử đệ quy đuôi của bài toán tháp Hà Nội.

TÀI LIỆU THAM KHẢO

Tiếng Việt

[1]. Trần Hạnh Nhi và Dương Anh Đức, *Giáo trình cấu trúc dữ liệu và giải thuật,* Đại học Quốc gia TP. HCM, 2001.

[2]. Nguyễn Hồng Chương, *Cấu trúc dữ liệu - Ứng dụng và cài đặt bằng C*, NXB TP. HCM, 2003.

[3]. Nguyễn Quốc Cường - Hoàng Đức Hải, *Cấu trúc dữ liệu + Giải thuật = Chương trình*, NXB Giáo dục, 1996.

[4]. Nguyễn Trung Trực, *Cấu trúc dữ liệu,* Đại học Bách khoa TP. HCM, 1994.

[5]. Nhóm tác giả Khoa CNTT, *Cấu trúc dữ liệu và giải thuật*, Trường Cao đẳng Công nghiệp Thực phẩm TP. HCM, 2007.

[6]. Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học Kỹ thuật, 1996.

[7]. Trần Hùng Cường, *Cấu trúc dữ liệu và giải thuật*, Đại học Công nghiệp Hà Nội, 2008.

[8]. Lê Minh Hoàng, *Giải thuật và giải thuật,* Đại học Sư phạm Hà Nội, 2002.

[9]. Lê Văn Vinh, *Cấu trúc dữ liệu 1,* Đại học Sư phạm Kỹ thuật Tp.HCM, 2008.

[10]. Nhóm tác giả, *Cấu trúc dữ liệu,* Đại học Cần thơ, 2003.

Tiếng Anh

[11]. A.V. Aho, J.E Hopcroft, J.D Ulman, *Data structures and algorithms*, insertison Wesley, 1983.

[12]. MARK ALLEN WEISS, *Data Structures and Algorithm Analysis*, InC. The Benjamin/Cummings Publishing Company, 1993.

[13]. Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

[14]. Robert Sedgewick, *Cẩm nang thuật toán*, Nhà xuất bản Khoa học Kỹ thuật, 1994.

[15]. Th. Cormen, Ch. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1998.