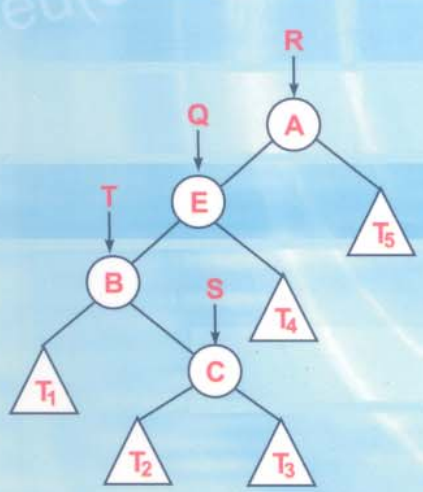
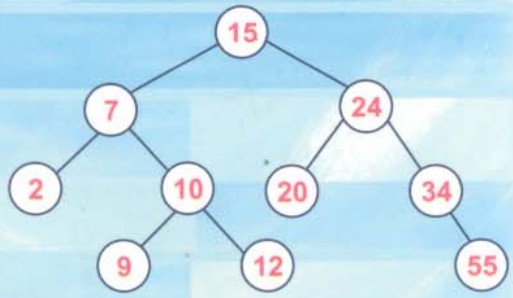
TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
ThS. **An Văn Minh** - ThS. **Trần Hùng Cường**

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT





NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
ThS. AN VĂN MINH - ThS. TRẦN HÙNG CƯỜNG

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG
Hà Nội - 2009

DANH SÁCH HỘI ĐỒNG THẨM ĐỊNH

PGS. TS. Đoàn Văn Ban:	Chủ tịch
ThS. Ngô Đức Vĩnh:	Thư ký
PGS. TS. Ngô Quốc Tạo:	Ủy viên
ThS. Lê Anh Thắng:	Ủy viên
ThS. Nguyễn Mạnh Cường:	Ủy viên

Mã số: GD 14 HM 09

LỜI NÓI ĐẦU

Công nghệ thông tin ngày càng được ứng dụng rộng rãi và hiệu quả trong mọi lĩnh vực khoa học tự nhiên và xã hội. Để thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết được trên máy tính thì vấn đề thiết kế, lựa chọn cấu trúc dữ liệu và giải thuật là một giai đoạn quan trọng trong qui trình thiết kế và xây dựng phần mềm.

Nhằm giới thiệu những kiến thức cơ bản về cấu trúc dữ liệu và giải thuật, Khoa Công nghệ Thông tin - Trường Đại học Công nghiệp Hà Nội đã phối hợp với Nhà xuất bản Thông tin và Truyền thông xuất bản cuốn sách ***"Cấu trúc dữ liệu và giải thuật"***. Cuốn sách do ThS. An Văn Minh và ThS. Trần Hùng Cường biên soạn dựa theo đề cương chi tiết qui định của Trường Đại học Công nghiệp Hà Nội và đã được Hội đồng khoa học của Trường thẩm định. Đây là môn học cơ sở cùng tên trong chương trình đào tạo kỹ sư công nghệ thông tin. Bài giảng gồm 5 chương với nội dung như sau:

Chương 1: Tổng quan về cấu trúc dữ liệu và giải thuật, bao gồm các khái niệm về cấu trúc dữ liệu và giải thuật, mối quan hệ giữa chúng, vấn đề thiết kế cấu trúc dữ liệu, thiết kế và phân tích giải thuật, đánh giá độ phức tạp của giải thuật.

Chương 2: Đệ quy và giải thuật đệ quy, một phương pháp thiết kế giải thuật khá quan trọng, nhất là với các giải thuật biểu diễn các thao tác xử lý cấu trúc dữ liệu dạng cây.

Chương 3: Sắp xếp và tìm kiếm, tập trung vào vấn đề mô tả, thiết kế và đánh giá các giải thuật sắp xếp và tìm kiếm thông dụng, cũng như vấn đề cài đặt các giải thuật này trong bài toán ứng dụng.

Chương 4: Danh sách tuyến tính, một loại cấu trúc dữ liệu rất phổ biến trong các bài toán tin học. Trong chương này trình bày các phương pháp lưu trữ danh sách và các thao tác xử lý tương ứng với mỗi loại danh sách.

Chương 5: Cây, một dạng cấu trúc dữ liệu phi tuyến tính, chương này chủ yếu nói về cây nhị phân và các ứng dụng của chúng.

Bài tập sau mỗi chương đã được chọn lọc ở mức độ phù hợp với sinh viên, qua đó giúp cho sinh viên hiểu sâu sắc thêm về bài giảng, củng cố thêm về kỹ thuật cài đặt chương trình và nắm bắt được một số kiến thức không được trực tiếp giới thiệu trong bài giảng.

Để học tốt môn học này đòi hỏi sinh viên phải thành thạo ít nhất một ngôn ngữ lập trình cơ bản như Pascal, C hay C++, v.v..., thành thạo các kỹ thuật lập trình như: cấu trúc rẽ nhánh, cấu trúc lặp, kỹ thuật lập trình đơn thể (sử dụng hàm, thủ tục).

Mặc dù nhóm tác giả có nhiều cố gắng trong công tác biên soạn song sẽ khó tránh khỏi thiếu sót, chúng tôi rất mong nhận được ý kiến đóng góp của các bạn đồng nghiệp và bạn đọc để lần xuất bản sau được hoàn thiện hơn.

Mọi ý kiến đóng góp xin gửi về Khoa Công nghệ Thông tin – Trường Đại học Công nghiệp Hà Nội.

Xin trân trọng giới thiệu./.

Hà Nội, tháng 9 năm 2009

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Mỗi bài toán bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên các đối tượng đó. Vì thế, 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 thành phần 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 biểu diễn chúng một cách thích hợp nhất, để vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này 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 của bài toán, cần tìm ra các giải pháp tương ứng để giải quyết, mỗi giải pháp cần phải xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn. Đây là bước xây dựng giải thuật cho bài toán. Có thể sử dụng các giải thuật có sẵn, hoặc tự xây dựng.

Tuy nhiên, khi giải quyết bài toán, ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng các giải thuật mà không chú trọng tới tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn 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. Để xác định được giải thuật phù hợp ta cần phải biết nó tác động đến loại dữ liệu nào và khi lựa chọn cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào tác động lên dữ liệu đó. Như vậy trong một bài toán, cấu trúc dữ liệu và giải thuật có mối quan hệ chặt chẽ với nhau, được thể hiện qua "công thức":

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi, thường giải thuật cũng phải thay đổi theo để tránh việc xử lý "gượng ép", thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa nhanh vừa tiết kiệm, giải thuật cũng đơn giản và dễ hiểu hơn.

Ví dụ 1: Một chương trình quản lý điểm thi của sinh viên cần lưu các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số tương ứng với 4 môn học khác nhau nên dữ liệu có dạng như sau:

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

Xét thao tác xử lý là xuất điểm số các môn của từng sinh viên.

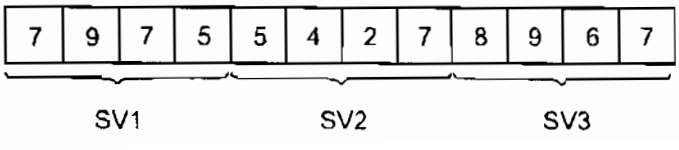
Giả sử có các phương án tổ chức lưu trữ như sau:

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

Có tất cả 3(SV) * 4(Môn) = 12 điểm số cần lưu trữ, do đó ta khai báo mảng như sau:

```
int a[12];
```

Khi đó mảng a các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i là phần tử tại dòng i cột j trong bảng. Để truy xuất đến phần tử này ta phải sử dụng công thức xác định chỉ số tương ứng trong mảng a:

Bảng điểm (dòng i, cột j) ⇒ a[(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 sau:

a[i] ⇒ bảng điểm (dòng(i/cột) + 1), cột (i % số cột))

Với phương án này, giải thuật xử lý được viết như sau:

```
void xuat(int a[ ])
{ int i, mon, so_mon;
  so_mon = 4;
  for (i = 0; i < 12; i++)
  {       sv = i/so_mon;
          mon = i % so_mon;
          cout<<"\nĐiểm môn:"<<mon;
          cout<<" của sinh viên "<<sv;
          cout<<" là:"<< a[i]<<endl;
  }
}
```

Phương án 2: *Sử dụng mảng hai chiều*

Khai báo mảng hai chiều a có kích thước 3 dòng * 4 cột như sau:

int *a[3][4];*

	Cột 1	**Cột 2**	**Cột 3**	**Cột 4**
Dòng 1	a[1][1] = 7	a[1][2] = 9	a[1][3] = 7	a[1][4] = 5
Dòng 2	a[2][1] = 5	a[2][2] = 4	a[2][3] = 2	a[2][4] = 7
Dòng 3	a[3][1] = 8	a[3][2] = 9	a[3][3] = 6	a[3][4] = 7

Và 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 ở dòng i cột j trong mảng.

Bảngđiểm (dòng i, cột j) ⇒ a[i,j]

Với phương án này, giải thuật xử lý được viết như sau:

```
void xuat( int a[3][4])
{
  int i, j, so_sv, so_mon;
  so_mon = 4; so_sv = 3;
  for (i = 0; i < so_sv; i++)
        {
              for (j = 0; j < so_mon; j++)
              {cout<<"\nĐiểm môn:"<<mon;
               cout<<"của sinh viên"<<sv;
               cout<<"là:"<< a[i][j]<<endl;
              }
        }
}
```

Nhận xét:

Có thể thấy rõ phương án 2 cung cấp một cấu trúc dữ liệu lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản hơn, tự nhiên hơn.

2. CÁC TIÊU CHUẨN ĐÁNH GIÁ CẤU TRÚC DỮ LIỆU

Do tầm quan trọng của cấu trúc dữ liệu đã được trình bày trong phần trên, nên nhất thiết phải chú trọng đến việc lựa chọn một phương án tổ chức dữ liệu thích hợp cho bài toán cụ thể. Một cấu trúc dữ liệu tốt phải thoả mãn các tiêu chuẩn sau:

**** 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 lựa chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ: Một số tình huống sau chọn cấu trúc lưu trữ sai

- Chọn biến số nguyên "***int***" để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng *5%), do vậy khi làm tròn mọi giá trị tiền thưởng sẽ gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường Trung học, mỗi lớp có thể nhận tối đa 25 học sinh. Lớp hiện có 20 học sinh, mỗi tháng, mỗi học sinh đóng học phí 15.000 đồng. Chọn một biến số nguyên (khả năng trong phạm vi (-32768 ÷ 32767)) để lưu trữ tổng số học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 5 học sinh nữa vào lớp thì giá trị tổng học phí thu được là 375000 đồng, vượt khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn số và sai lệch.

* ***Phù hợp với các giải thuật xử lý trên đó***: Tiêu chuẩn này giúp tăng hiệu quả khi giải quyết bài toán, việc phát triển các giải thuật đơn giản, tự nhiên hơn và chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

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

Ví dụ: Một số tình huống chọn cấu trúc lưu trữ lãng phí

- Sử dụng biến "***int***" (2 byte) để lưu trữ một giá trị cho biết tháng hiện hành. Trong tình huống này ta chỉ cần sử dụng biến kiểu "***char***" là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 60 phần tử (giới hạn số học viên trong lớp tối đa là 60). Nếu số lượng học

viên thật sự ít hơn 60, thì gây lãng phí bộ nhớ. Hơn nữa, số học viên có thể thay đổi theo từng kỳ, từng năm. Trong trường hợp này ta cần có một cấu trúc dữ liệu linh động hơn mảng, chẳng hạn danh sách móc nối.

3. CÁC CẤU TRÚC DỮ LIỆU CƠ SỞ

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thô sơ. Nếu muốn phản ánh được dữ liệu thực tế vốn rất đa dạng và phong phú, cần phải xây dựng những phép ánh xạ, những qui 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 phần đầu, giữa hình thức lưu trữ 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.

3.1. Định nghĩa kiểu dữ liệu

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

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

Ví dụ:

- Kiểu dữ liệu **Ký tự alphabet** = <V_c, O_c> với:

  V_c = {a - z, A - Z}

  O_c = {lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa,...}

- Kiểu dữ liệu **Số nguyên** = <V_i, O_i> với:

  V_i = {-32768 ÷32767}

  O_i = {+, -, *, /, %, các phép so sánh, các phép toán logic nhị phân}

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.

3.2. Các thuộc tính của một kiểu dữ liệu

Một kiểu dữ liệu bao gồm các thuộc tính sau:

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

3.3. Các kiểu dữ liệu cơ bản

Thông thường trong một hệ kiểu của ngôn ngữ lập trình sẽ có một số kiểu dữ liệu được gọi là *kiểu dữ liệu đơn* hay *kiểu dữ liệu nguyên tử* (atomic).

Thông thường, các kiểu dữ liệu cơ bản bao gồm:

- Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic, liệt kê, miền con
- Kiểu không rời rạc: số thực.

Tuỳ từng ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn này có thể khác nhau đôi chút. Chẳng hạn, với ngôn ngữ lập trình C/C++, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và trong ngôn ngữ lập trình C/C++, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic đúng (TRUE) và giá trị logic sai (FALSE) được biểu diễn trong ngôn ngữ lập trình C/C++ như là các giá trị nguyên khác 0 và bằng 0. Trong khi đó ngôn ngữ lập trình Pascal định nghĩa tất cả các kiểu dữ liệu đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ.

Các kiểu dữ liệu của C/C++ được cho trong bảng sau:

Tên kiểu	Phạm vi	Kích thước	Giải thích
int	-32768 ÷ +32767	2 byte	Số nguyên có dấu
char	-128 ÷ +127	1 byte	
long	-2147483648 ÷ +2147483647	4 byte	
unsigned char	0 ÷ 255	1 byte	Số nguyên không dấu
unsigned int	0 ÷ 65535	2 byte	
unsigned long	0 ÷ 4294967259	4 byte	

Tên kiểu	Phạm vi	Kích thước	Giải thích
float	$1.2*10^{-38} \div 3.4*10^{38}$	4 byte	Số thực (dấu chấm động)
double	$2.2*10^{-308} \div 1.8*10^{308}$	8 byte	
long double	$3.5*10^{-3942} \div 3.4*10^{4932}$	10 byte	

3.4. Các kiểu dữ liệu có cấu trúc

Khi giải quyết các bài toán phức tạp, nếu chỉ sử dụng các dữ liệu các dữ liệu đơn là không đủ, ta phải cần đến các *cấu trúc dữ liệu*. Một cấu trúc dữ liệu bao gồm một tập hợp các *dữ liệu nguyên tử*, các thành phần này kết hợp với nhau theo một phương thức được qui định bởi ngôn ngữ lập trình. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, cấu trúc, v.v... và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

3.4.1. Mảng một chiều

Trong ngôn ngữ lập trình C/C++ và trong nhiều ngôn ngữ thông dụng khác có một cách đơn giản nhất để tổ chức lưu trữ các đối tượng trong một tập hợp, đó là cách sắp xếp các đối tượng đó thành một dãy. Để lưu trữ dãy đối tượng trong máy tính người ta sử dụng mảng một chiều. Khi đó ta có một cấu trúc dữ liệu được gọi là mảng (array). Như vậy, có thể nói một mảng là một cấu trúc dữ liệu gồm một dãy xác định các dữ liệu thành phần cùng một kiểu (mảng số nguyên, mang số thực, mảng các cấu trúc, v.v...).

Trong C/C++ việc khai báo một mảng là khá đơn giản, cần chỉ ra kiểu dữ liệu của phần tử, tên mảng, kích thước mảng, mẫu như sau:

<Kiểu phần tử> <tên mảng> <[kích thước]>;

Ví dụ: ***int*** A[10]; //khai báo mảng A chứa 10 số nguyên (chỉ số từ 0 đến 9).

3.4.2. Chuỗi

Trong C/C++, chuỗi thực chất là một mảng các ký tự, tuy nhiên có khác là trong chuỗi có chứa ký tự kết thúc '\0'. Việc nhập và hiển

thị chuỗi cũng đơn giản hơn mảng, ta có thể sử dụng các hàm nhập xuất chuẩn trong thư viện "stdio.h" như *scanf(), gets(), printf(), puts()*.

C/C++ cũng định nghĩa một số hàm xử lý chuỗi (thư viện string.h) như: *strcpy(), strlen(), strcmp(), strchr(), strcat(), strstr(),...*

3.4.3. Mảng nhiều chiều

Mảng nhiều chiều được sử dụng nhiều nhất là mảng 2 chiều (mảng của mảng), có thể hình dung mảng 2 chiều giống như một bảng gồm các dòng và các cột, chẳng hạn, bảng ghi nhiệt độ trung bình trong 5 năm ở năm thành phố.

	2003	2004	2005	2006	2007
Hà Nội	27	27,5	28,5	30	27
TP. Hồ Chí Minh	32,5	32	30,5	31	30
Huế	30	31	32	28	29
Hải Phòng	27,5	26,5	26	27,5	27
Hạ Long	25	27	26	28	27

Cấu trúc khai báo của mảng nhiều chiều được viết như sau:

<Kiểu phần tử> <tên mảng ><[kích thước chiều 1]...[kích thước chiều n]>;

Sau tên mảng, mỗi cặp ngoặc vuông [ ] được tính là một chiều. Chữ số ghi trong cặp ngoặc [ ] là số phần tử của chiều đó.

Ví dụ: ***float*** temp [5][5]; // mảng 2 chiều temp, kích thước 5x5, mảng các số thực.

3.4.4. Cấu trúc

Cấu trúc là tập hợp các mẫu dữ liệu khác nhau của một đối tượng (các mẫu dữ liệu có thể có kiểu khác nhau). Các mẫu dữ liệu đó được gọi là thành phần dữ liệu của cấu trúc. Các cấu trúc có thể được sử dụng để tạo nên các kiểu dữ liệu khác, chẳng hạn như mảng cấu trúc.

Giả sử T_1, T_2,...,T_n là các kiểu đã cho, và F_1, F_2,..., F_n là các tên thành phần. Khi đó ta có thể thành lập kiểu cấu trúc ST với n thành phần dữ liệu, thành phần thứ i có tên là F_i và các giá trị của nó có kiểu T_i với i = 1, 2,..., n.

```
struct ST
{
        T1      F1;
        T2      F2;
        ...
        Tn      Fn;
};
struct ST s1, s2, d[20];
```

trong khai báo này s1, s2 là các biến cấu trúc, d là một mảng cấu trúc.

Để truy nhập vào một thành phần dữ liệu của một cấu trúc ta viết theo mẫu:

<tên biến cấu trúc>.<tên thành phần>

Ví dụ: s1.F1, d[2].Fn

Mỗi giá trị của kiểu cấu trúc ST là một bộ k giá trị (t_1, t_2,..., t_k), trong đó $t_i \in T_i$ (i = 1, 2, ..., k).

Ví dụ: Khai báo cấu trúc lưu trữ phân số gồm tử số và mẫu số là các số nguyên.

```
struct phan_so
{
        int tu_so;
        int mau_so;
};
//khai báo các biến lưu trữ phân số
struct phan_so p1, p2, ps[100];
```

Ở đây, p1, p2 là hai biến cấu trúc lưu trữ phân số, còn ps là mảng lưu trữ một dãy nhiều nhất là 100 phân số, ps được gọi là mảng cấu trúc.

3.4.5. Kiểu con trỏ

Một phương pháp quan trọng nữa để kiến tạo các cấu trúc dữ liệu, đó là sử dụng con trỏ.

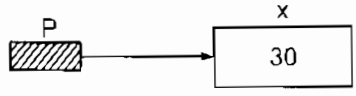
Con trỏ là biến được sử dụng để lưu địa chỉ của một biến khác.

Con trỏ được khai báo theo mẫu:

<kiểu dữ liệu> *<tên con trỏ>;

Ví dụ: ***int*** *P, x;

P=&x; *//P chứa địa chỉ của biến x, hay P trỏ vào x*



Hình 1.1: Biểu diễn con trỏ

Sau này con trỏ được dùng để tạo ra kiểu danh sách móc nối, hoặc cây là các cấu trúc dữ liệu rất quan trọng, ta sẽ có dịp tìm hiểu kỹ hơn cách sử dụng con trỏ trong chương 3.

3.4.6. Kiểu file (tệp tin)

Khác với các kiểu dữ liệu trước đây, số nguyên, số thực, mảng, chuỗi, v.v... dữ liệu được lưu ở bộ nhớ trong. Vì thế, khi chương trình kết thúc, dữ liệu cũng bị xóa. Để khắc phục trường hợp này, dữ liệu cần được lưu ở bộ nhớ ngoài, đó là các tệp tin.

- Theo cách lưu trữ này, khi thao tác cần một tên tệp tin (gồm cả đường dẫn), một con trỏ tệp (FILE * tên_con_trỏ).

- Khi thao tác với tệp tin cũng cần các hàm xử lý, bạn đọc có thể xem trong [3].

3.5. Các phép toán trong các kiểu dữ liệu của C/C++

Như đã nói ở trên, với mỗi kiểu dữ liệu ta chỉ có thể thực hiện một số phép toán nhất định trên các dữ liệu của kiểu. Ta không thể áp dụng một số phép toán trên các dữ liệu thuộc kiểu này cho các dữ liệu thuộc kiểu khác. Các phép toán rất quan trọng, nó là công cụ để thao

tác dữ liệu. Ta có thể chia tập hợp các phép toán trên các kiểu dữ liệu của C/C++ thành hai lớp sau:

3.5.1. Các phép toán truy nhập

Phép toán này dùng để truy nhập đến các thành phần của một đối tượng dữ liệu, chẳng hạn truy nhập đến các phần tử của một mảng, đến các thành phần dữ liệu của cấu trúc.

Ví dụ:

- Giả sử A là một mảng một chiều với n phần tử, (i = 0, 1,..., n-1) khi đó A[i] cho phép ta truy nhập đến thành phần thứ i+1 của mảng.

- Nếu X là một biến cấu trúc thì việc truy nhập đến trường F của nó được thực hiện bởi phép toán X.F.

3.5.2. Các phép toán kết hợp dữ liệu

Ngôn ngữ lập trình C/C++ có một tập hợp phong phú các phép toán kết hợp một hoặc nhiều dữ liệu đã cho thành dữ liệu mới. Sau đây là một số nhóm các phép toán chính.

*** Các phép toán số học**: Đó là các phép toán +, - * , / trên tập số thực; các phép toán +, -, * , /, %, trên tập số nguyên.

*** Các phép toán so sánh**: Trên các đối tượng thuộc các kiểu có thứ tự, ta có thể thực hiện các phép toán so sánh = = (bằng), != (khác), < (nhỏ hơn), > (lớn hơn), <= (nhỏ hơn hoặc bằng), >= (lớn hơn hoặc bằng). Cần chú ý rằng, kết quả của các phép toán này là một giá trị logic (true/false).

*** Các phép toán logic**: Đó là các phép toán &&, ||, !, được thực hiện trên hai giá trị *false* và *true*. Trong C/C++ không có kiểu logic, mà false là giá trị bằng không, và true là giá trị khác không.

4. GIẢI THUẬT - PHÂN TÍCH VÀ ĐÁNH GIÁ GIẢI THUẬT

4.1. Giải thuật

Giải thuật (algorithm) là một trong những khái niệm quan trọng nhất trong tin học. Thuật ngữ giải thuật xuất phát từ nhà toán học

A-rập Abu Ja'far Mohammed ibn Musa al Khowarizmi (khoảng năm 825). Tuy nhiên, trong lúc bấy giờ và trong nhiều thế kỷ sau, nó không mang nội dung như ngày nay chúng ta quan niệm. Giải thuật nổi tiếng nhất, có từ thời cổ Hy Lạp là giải thuật Euclid, giải thuật tìm ước chung lớn nhất của hai số nguyên. Có thể mô tả giải thuật này như sau:

**** Giải thuật Euclid***

Vào: m, n nguyên dương

Ra: d, ước số chung lớn nhất của m và n.

Phương pháp

Bước 1: Tìm r, phần dư của phép chia m cho n

Bước 2: Nếu r = 0, thì d ← n (gán giá trị của n cho d) và dừng lại

Ngược lại, thì m ← n, n ← r và quay lại bước 1

4.1.1. Khái niệm

Giải thuật là một dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện để giải quyết vấn đề đặt ra.

4.1.2. Đặc trưng của giải thuật

Định nghĩa trên, còn chứa đựng nhiều điều chưa rõ ràng. Để hiểu đầy đủ ý nghĩa của giải thuật, chúng ta nêu ra 5 đặc trưng của nó:

* ***Bộ dữ liệu vào:*** Mỗi giải thuật cần có một số lượng (có thể bằng 0) dữ liệu vào (input). Đó là các giá trị cần đưa vào khi giải thuật bắt đầu làm việc. Các dữ liệu này cần được lấy từ các tập hợp giá trị cụ thể nào đó. Chẳng hạn, trong giải thuật Euclid trên, m và n là các dữ liệu vào lấy từ tập các số nguyên dương.

* ***Dữ liệu ra***: Mỗi giải thuật cần có một hoặc nhiều dữ liệu ra. Đó là các giá trị có quan hệ hoàn toàn xác định với các dữ liệu vào và là kết quả của sự thực hiện giải thuật. Trong giải thuật Euclid có một dữ liệu ra, đó là d, khi thực hiện đến bước 2 và phải dừng lại (trường hợp r = 0), giá trị của d là ước số chung lớn nhất của m và n.

* ***Tính xác định***: Mỗi bước của giải thuật cần phải được mô tả một các chính xác, chỉ có một cách hiểu duy nhất. Hiển nhiên đây là một đòi hỏi rất quan trọng. Bởi vì, nếu một bước có thể hiểu theo nhiều cách khác nhau, thì cùng một dữ liệu vào, những người thực hiện giải thuật khác nhau có thể dẫn đến các kết quả khác nhau. Để đảm bảo được tính xác định giải thuật cần phải được mô tả trong các ngôn ngữ lập trình. Trong các ngôn ngữ này, các mệnh đề được tạo thành theo qui tắc, cú pháp nghiêm ngặt và chỉ có một ý nghĩa duy nhất.

* ***Tính khả thi***: Tất cả các phép toán có mặt trong các bước của giải thuật phải đủ đơn giản. Điều này có nghĩa là, người lập trình có thể thực hiện chỉ bằng giấy trắng và bút trong khoảng thời gian hữu hạn. Chẳng hạn, với giải thuật Euclid, ta chỉ cần thực hiện các phép chia số nguyên, các phép gán và phép so sánh để biết được $r = 0$ hay $r \neq 0$.

* ***Tính dừng***: Với mọi bộ dữ liệu vào thoả mãn các điều kiện của dữ liệu vào, giải thuật phải dừng lại sau một số hữu hạn các bước thực hiện. Chẳng hạn, giải thuật Euclid thoả mãn điều kiện này. Bởi vì, khi thực hiện bước 1 thì giá trị của r nhỏ hơn n, nếu $r \neq 0$ thì giá trị của n ở bước 2 là giá trị của r ở bước trước, ta có $n > r = n_1 > r_1 = n_2 > r_2$,... Dãy số nguyên dương giảm dần cần phải kết thúc ở 0, do đó sau một số hữu hạn bước giá trị của r phải bằng 0, giải thuật dừng.

4.2. Biểu diễn giải thuật

Có nhiều phương pháp biểu diễn giải thuật. Có thể biểu diễn giải thuật bằng danh sách các bước, các bước được diễn đạt bằng ngôn ngữ tự nhiên và các ký hiệu toán học. Có thể biểu diễn bằng sơ đồ khối. Tuy nhiên, như đã trình bày, để đảm bảo tính xác định của giải thuật thì nên biểu diễn nó bằng ngôn ngữ lập trình.

4.3. Phân tích giải thuật

Giả sử đối với một bài toán nào đó chúng ta có một số giải thuật để giải. Một câu hỏi đặt ra là, chúng ta cần chọn giải thuật nào trong

số các giải thuật đã có để giải bài toán một cách hiệu quả nhất. Sau đây ta phân tích giải thuật và đánh giá độ phức tạp tính toán của nó.

4.3.1. Tính hiệu quả của giải thuật

Khi giải quyết một vấn đề, chúng ta cần chọn trong số các giải thuật, một giải thuật mà chúng ta cho là tốt nhất. Vậy ta cần lựa chọn giải thuật dựa trên cơ sở nào? Thông thường ta dựa trên hai tiêu chuẩn sau đây:

1. Giải thuật đơn giản, dễ hiểu, dễ cài đặt (để viết chương trình)

2. Giải thuật sử dụng tiết kiệm nhất nguồn tài nguyên của máy tính và đặc biệt, chạy nhanh nhất có thể được.

Khi ta viết một chương trình chỉ để sử dụng một số ít lần và cái giá của thời gian viết chương trình vượt xa cái giá của chạy chương trình thì tiêu chuẩn (1) là quan trọng nhất. Nhưng có trường hợp ta cần viết các chương trình (thủ tục hoặc hàm) để sử dụng nhiều lần, cho nhiều người sử dụng, khi đó giá của thời gian chạy chương trình sẽ vượt xa giá viết nó. Chẳng hạn, các thủ tục sắp xếp, tìm kiếm được sử dụng rất nhiều lần, cho rất nhiều người trong các bài toán khác nhau. Trong trường hợp này ta cần dựa trên tiêu chuẩn (2). Ta sẽ cài đặt giải thuật có thể rất phức tạp, miễn là chương trình nhận được chạy nhanh hơn các giải thuật khác.

Tiêu chuẩn (2) được xem là tính hiệu quả của giải thuật. Tính hiệu quả của giải thuật bao gồm hai nhân tố cơ bản

- Dung lượng nhớ cần thiết để lưu giữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả của giải thuật.

- Thời gian cần thiết để thực hiện giải thuật (ta gọi là thời gian chạy chương trình, thời gian này không phụ thuộc vào các yếu tố vật lý của máy tính như tốc độ xử lý của máy tính, ngôn ngữ viết chương trình, v.v...).

Chúng ta sẽ chỉ quan tâm đến thời gian thực hiện giải thuật. Vì vậy khi nói đến đánh giá độ phức tạp của giải thuật, có nghĩa là ta nói

đến đánh giá thời gian thực hiện. Một giải thuật có hiệu quả được xem là giải thuật có thời gian chạy ít hơn các giải thuật khác.

4.3.2. Đánh giá thời gian thực hiện của giải thuật

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một giải thuật.

a. Phương pháp thử nghiệm:

Chương trình được viết và cho chạy với các dữ liệu vào khác nhau trên một máy tính nào đó. Thời gian chạy chương trình phụ thuộc vào các yếu tố sau đây:

1. Các dữ liệu vào

2. Chương trình dịch, để chuyển chương trình mã nguồn thành chương trình mã máy.

3. Tốc độ thực hiện các phép toán của máy tính được sử dụng để chạy chương trình.

Vì thời gian chạy chương trình phụ thuộc vào nhiều yếu tố, nên ta không thể biểu diễn chính xác thời gian chạy là bao nhiêu đơn vị thời gian chuẩn, chẳng hạn nó là bao nhiêu giây nếu không nêu các cấu hình hệ máy thực hiện.

b. Phương pháp lý thuyết:

Ta sẽ coi thời gian thực hiện của giải thuật như là một hàm số của kích thước dữ liệu vào. Kích thước của dữ liệu vào là một tham số đặc trưng cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Đơn vị tính kích thước của dữ liệu vào phụ thuộc vào các giải thuật cụ thể. Chẳng hạn, đối với các giải thuật sắp xếp mảng, thì kích thước của dữ liệu vào là số thành phần (phần tử) của mảng, đối với giải thuật giải hệ n phương trình tuyến tính với n ẩn, ta chọn n là kích thước. Thông thường dữ liệu vào là một số nguyên dương n. Ta sẽ sử dụng hàm số T(n), trong đó n là kích thước dữ liệu vào, để biểu diễn thời gian thực hiện của một giải thuật.

Có thể xác định thời gian thực hiện T(n) là số phép toán sơ cấp cần phải tiến hành khi thực hiện giải thuật. Các phép toán sơ cấp là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số chỉ phụ thuộc vào cách cài đặt được sử dụng. Chẳng hạn các phép toán số học +, -, *, /, các phép toán so sánh =, !=,... là các phép toán sơ cấp.

4.3.3. Đánh giá độ phức tạp tính toán của giải thuật

Khi đánh giá thời gian thực hiện bằng phương pháp toán học, chúng ta sẽ bỏ qua yếu tố phụ thuộc vào cách cài đặt, chỉ tập trung vào xác định độ lớn của thời gian thực hiện T(n). Ký hiệu toán học O (đọc là ô lớn) được sử dụng để mô tả độ lớn của hàm T(n).

Giả sử n là số nguyên không âm, T(n) và f(n) là các hàm thực không âm. Ta viết T(n) = O(f(n)) (đọc: T(n) là ô lớn của f(n)), nếu và chỉ nếu tồn tại các hằng số dương c và n_0 sao cho $T(n) \le c.f(n)$, với $\forall n > n_0$.

Nếu một giải thuật có thời gian thực hiện T(n) = O(f(n)), chúng ta sẽ nói rằng giải thuật có thời gian thực hiện cấp f(n).

Ví dụ: Giả sử $T(n) = 10n^2 + 4n + 4$

Ta có: $T(n) \le 10n^2 + 4n^2 + 4n^2 = 18n^2, \forall n \ge 1$

Vậy $T(n) = O(n^2)$. Trong trường hợp này ta nói giải thuật có độ phức tạp (có thời gian thực hiện) cấp n^2.

Bảng sau đây cho ta các cấp thời gian thực hiện giải thuật được sử dụng rộng rãi nhất và tên gọi thông thường của chúng.

Ký hiệu ô lớn (O)	Tên gọi thông thường
O(1)	Hằng
$O(\log_2 n)$	logarit
O(n)	Tuyến tính
$O(n\log_2 n)$	$n\log_2 n$
$O(n^2)$	Bình phương
$O(n^3)$	Lập phương
$O(2^n)$	Mũ

Danh sách trên sắp xếp theo thứ tự tăng dần của cấp thời gian thực hiện.

Các hàm như $\log_2 n$, n, $n\log_2 n$, n^2, n^3 được gọi là các hàm đa thức. Giải thuật với thời gian thực hiện có cấp hàm đa thức thì thường chấp nhận được.

Các hàm như 2^n, n!, n^n được gọi là hàm loại mũ. Một giải thuật mà thời gian thực hiện của nó là các hàm loại mũ thì tốc độ rất chậm.

4.3.4. Xác định độ phức tạp tính toán

Xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể dẫn đến những bài toán phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta cũng có thể phân tích được bằng một số qui tắc đơn giản.

a. Qui tắc tổng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai giai đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$ thì thời gian thực hiện đoạn P_1 rồi P_2 tiếp theo sẽ là $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

Ví dụ: Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n\log_2 n)$ thì thời gian thực hiện 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Khi đó thời gian thực hiện chương trình sẽ là $O(\max(n^3, n\log_2 n)) = O(n^3)$.

b. Qui tắc nhân:

Nếu tương ứng với P_1 và P_2 là $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện P_1 và P_2 lồng nhau sẽ là: $T_1(n).T_2(n) = O(f(n).g(n))$

Trong tài liệu quốc tế các giải thuật thường được trình bày dưới dạng các thủ tục hoặc hàm trong ngôn ngữ tựa Pascal/C. Để đánh giá thời gian thực hiện giải thuật, ta cần biết cách đánh giá thời gian thực

hiện các câu lệnh của Pascal/C. Các câu lệnh trong Pascal/C được định nghĩa đệ qui như sau:

1. Các phép gán, đọc, viết, **goto** là các câu lệnh. Các lệnh này gọi là lệnh đơn

2. Nếu S_1, S_2,..., S_n là các câu lệnh thì:

$\{S_1; S_2;...; S_n\}$

là câu lệnh và được gọi là lệnh hợp thành (hoặc khối lệnh).

3. Nếu S_1 và S_2 là các câu lệnh và E là biểu thức logic thì:

if (E) S_1; ***else*** S_2;

và ***if*** (E) S_1;

là câu lệnh và được gọi là lệnh ***if* - lệnh rẽ nhánh điều kiện.**

4. Nếu S_1, S_2,..., S_{n+1} là các câu lệnh, E là biểu thức có kiểu thứ tự đếm được, và v_1, v_2, ..., v_n là các giá trị cùng kiểu với E thì:

```
switch (E)
{
        v1 : S1 ; break;
        v2 : S2 ; break;
        ...
        vn : Sn ; break;
        [default: Sn+1;]
}
```

là câu lệnh và được gọi là lệnh ***switch* - lệnh rẽ nhánh lựa chọn.**

5. Nếu S là câu lệnh và E là biểu thức logic thì:

while (E) S;

là câu lệnh và được gọi là lệnh ***while* - vòng lặp.**

6. Nếu S_1, S_2,..., S_n là các câu lệnh, E là biểu thức logic thì:

do $\{S_1, S_2,..., S_n\}$ ***while*** (E);

là câu lệnh và được gọi là lệnh ***do ... while* - vòng lặp**

7. Với S là câu lệnh, E_1 và E_2 là các biểu thức có cùng một kiểu thứ tự đếm được, thì:

for (i= E_1; i<=E_2; i++) S;

là câu lệnh, và

for (i= E_2; i>=E_1; i--) S;

là câu lệnh. Lệnh này được gọi là lệnh ***for* - vòng lặp.**

Giả sử rằng, các lệnh gán không chứa các lời gọi hàm. Khi đó để đánh giá thời gian thực hiện một chương trình, ta có thể áp dụng phương pháp đệ qui sau:

1. Thời gian thực hiện các lệnh đơn: gán, đọc, viết là O(1).

2. Lệnh hợp thành: thời gian thực hiện lệnh hợp thành được xác định bởi luật tổng.

3. Lệnh ***if***: Giả sử thời gian thực hiện các lệnh S_1, S_2 là O(f(n)) và O(g(n)) tương ứng. Khi đó thời gian thực hiện lệnh ***if*** là O(max (f(n), g(n))).

4. Lệnh ***switch***: Lệnh này được đánh giá như lệnh ***if***.

5. Lệnh ***while***: Giả sử thời gian thực hiện lệnh S (thân của ***while***) là O(f(n)) và g(n) là số tối đa các lần thực hiện lệnh S, khi đó thời gian thực hiện lệnh ***while*** là O(f(n).g(n)).

6. Lệnh ***do...while***: Giả sử thời gian thực hiện khối {S_1, S_2, ..., S_n} là O(f(n)) và g(n) là số lần lặp tối đa. Khi đó thời gian thực hiện lệnh **do...*while*** là O(f(n).g(n)).

7. Lệnh ***for***: Lệnh này được đánh giá tương tự như lệnh ***do...while*** và ***while***.

Đánh giá thủ tục (hoặc hàm) đệ qui:

Trước hết chúng ta xét một ví dụ cụ thể, ta sẽ đánh giá thời gian thực hiện của hàm đệ qui sau:

```
int fact (int n)
{
        if (n <= 1) return 1;
        else return n* fact (n - 1);
}
```

Trong hàm này kích thước của dữ liệu vào là n, giả sử thời gian thực hiện hàm là T(n).

- Với n = 1, chỉ cần thực hiện lệnh gán fact = 1, do đó T(1) = O(1).

- Với n > 1, cần thực hiện lệnh gán fact = n*fact(n - 1). Do đó thời gian T(n) là O(1) (để thực hiện phép nhân và phép gán) cộng với T(n -1) (để thực hiện lời gọi đệ qui fact(n - 1)).

Tóm lại, ta có quan hệ đệ qui sau:

$$T(1) = O(1)$$

$$T(n) = O(1) + T(n - 1)$$

Thay các O(1) bởi các hằng nào đó, ta nhận được quan hệ đệ quy sau:

$$T(1) = C_1$$

$$T(n) = C_2 + T(n - 1)$$

Để giải phương trình đệ qui, tìm T(n), chúng ta áp dụng phương pháp thế lặp. Ta có phương trình đệ qui:

$$T(m) = C_2 + T(m - 1), \text{ với } m > 1$$

Thay m lần lượt bởi 2, 3,..., n - 1, n, ta nhận được các quan hệ sau:

$$T(2) = C_2 + T(1)$$

$$T(3) = C_2 + T(2)$$

...

$$T(n -1) = C_2 + T(n -2)$$

$$T(n) = C_2 + T(n - 1)$$

Bằng các phép thế liên tiếp, ta nhận được:

$$T(n) = (n - 1) C_2 + T(1)$$

hay $T(n) = (n - 1) C_2 + C_1$, trong đó C_1 và C_2 là các hằng nào đó.

Do đó, $T(n) = O(n)$.

Từ ví dụ trên, suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện thủ tục (hàm) đệ qui. Để đơn giản, ta giả thiết rằng các thủ tục (hàm) là đệ qui trực tiếp. Điều đó có nghĩa là các thủ tục (hàm) chỉ chứa các lời gọi đệ qui đến chính nó. Giả sử thời gian thực hiện thủ tục là $T(n)$, với n là kích thước dữ liệu vào. Khi đó thời gian thực hiện các lời gọi đệ qui được đánh giá thông qua các bước sau:

- Đánh giá thời gian thực hiện $T(n_0)$, với n_0 là cỡ dữ liệu vào nhỏ nhất có thể được (trong ví dụ trên, đó là $T(1)$).

- Đánh giá thân của thủ tục theo qui tắc 1-7 (qui tắc đánh giá thời gian thực hiện các câu lệnh) ta sẽ nhận được quan hệ đệ qui sau:

$$T(n) = F(T(m_1), T(m_2), ..., T(m_k))$$

Trong đó $m_1, m_2, ..., m_k < n$. Giải phương trình đệ qui này, ta sẽ nhận được sự đánh giá của $T(n)$.

4.4. Phân tích một số giải thuật

Ví dụ 1: Phân tích giải thuật Euclid

```
int Euclid (int m, int n)
{
  int r ;
  r = m % n;                          //(1)
  while (r!= 0)                       //(2)
        {
              m = n;                  //(3)
              n = r;                  //(4)
              r = m % n;              //(5)
        }
  return n;                           //(6)
}
```

Thời gian thực hiện giải thuật phụ thuộc vào số nhỏ nhất trong hai số m và n. Giả sử $m \geq n > 0$, khi đó cỡ của dữ liệu vào là n. Các lệnh (1) và (6) có thời gian thực hiện là O(1) vì chúng là các câu lệnh gán. Do đó thời gian thực hiện giải thuật là thời gian thực hiện lệnh ***while***, ta đánh giá thời gian thực hiện câu lệnh (2). Thân của lệnh này là khối gồm ba lệnh (3), (4) và (5). Mỗi lệnh có thời gian thực hiện là O(1). Do đó khối có thời gian thực hiện là O(1). Ta còn phải đánh giá số lớn nhất các lần thực hiện lặp khối.

Ta có: $m = n.q_1 + r_1$, $0 \leq r_1 < n$

$n = r_1.q_2 + r_2$, $0 \leq r_2 < r_1$

Nếu $r_1 \leq n/2$ thì $r_2 < r_1 \leq n/2$, do đó $r_2 < n/2$

Nếu $r_1 > n/2$ thì $q_2 = 1$, tức là $n = r_1 + r_2$, do đó $r_2 < n/2$.

Tóm lại, ta luôn có $r_2 < n/2$.

Như vậy cứ hai lần thực hiện khối lệnh thì phần dư r giảm đi còn một nửa của n. Gọi k là số nguyên lớn nhất sao cho $2^k \leq n$. Suy ra số lần lặp tối đa là $2k + 1 \leq 2\log_2 n + 1$. Do đó thời gian thực hiện lệnh ***while*** là $O(\log_2 n)$. Đó cũng là thời gian thực hiện của giải thuật.

Ví dụ 2: Giải thuật tính giá trị của e^x tính theo công thức gần đúng

$e^x = 1 + x/1! + x^2/2! + ... + x^n/n!$, với x và n cho trước

```
float Exp1 (int n, float x)
{
        //Tính từng số hạng sau đó cộng dồn lại
        float s, p;
        int i, j;
        s = 1;                                  //(1)
        for (i =1; i<=n; i++)                   //(2)
        {       p = 1;                          //(3)
                for (j =1; j<=i; j++)           //(4)
                    p = p*x/j;                  //(5)
                    s = s + p;                  //(6)
        }
  return s;                                     //(7)
}
```

Ta thấy câu lệnh (1) và (7) là các câu lệnh gán nên chúng có thời gian thực hiện là O(1). Do đó, thời gian thực hiện của giải thuật phụ thuộc vào câu lệnh (2). Ta đánh giá thời gian thực hiện câu lệnh này. Trong thân của câu lệnh này bao gồm các lệnh (3), (4), (5) và (6). Hai câu lệnh (3) và (7) có thời gian thực hiện là O(n) vì mỗi câu lệnh được thực hiện n lần. Riêng câu lệnh (5) thì thời gian thực hiện nó còn phụ thuộc vào câu lệnh (4) nên ta còn phải đánh giá thời gian thực hiện câu lệnh (4).

Với i = 1 thì câu lệnh (5) được thực hiện 1 lần

Với i = 2 thì câu lệnh này được thực hiện 2 lần

...

Với i = n thì câu lệnh này được thực hiện n lần

Suy ra tổng số lần thực hiện câu lệnh (5) là:

$$1 + 2 + ... + n = n(n + 1)/2 \text{ lần}$$

Do đó thời gian thực hiện câu lệnh này là $O(n^2)$ và đây cũng là thời gian thực hiện của giải thuật.

Ta có thể viết giải thuật này theo cách khác: Dựa vào số hạng trước để tính số hạng sau:

$$\frac{x^2}{2!} = \frac{x}{1!}.\frac{x}{2} \ldots\ldots \frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!}.\frac{x}{n}$$

Giải thuật được viết dưới dạng hàm như sau:

```
float Exp2 (int n, float x)
{ float s, p;
  int i;
  s = 1;                                //(1)
  p = 1;                                //(2)
  for (i =1; i<= n; i++)                //(3)
  {       p = p*x/i;                    //(4)
          s = s + p;                    //(5)
  }
  return s;                             //(6)
}
```

Tương tự như giải thuật trước, các câu lệnh (1), (2), (6) có thời gian thực hiện là O(1). Do đó thời gian thực hiện giải thuật phụ thuộc vào câu lệnh (3). Vì hai câu lệnh (4) và (5) đều có thời gian thực hiện là O(n) nên thời gian thực hiện của giải thuật là O(n).

Như vậy từ hai giải thuật trên ta có thể nói rằng giải thuật thứ hai có nhiều ưu điểm hơn giải thuật thứ nhất với n đủ lớn (với n nhỏ thì thời gian thực hiện hai giải thuật này tương đương nhau).

Ví dụ 3: Tìm trong dãy số $s_1, s_2, ..., s_n$ một phần tử có giá trị bằng x cho trước

Vào: Dãy $s_1, s_2, ..., s_n$ và khoá cần tìm x

Ra: Vị trí phần tử có khoá x hoặc là n + 1 nếu không tìm thấy.

```
int linear_search(int s[ ], int n, int x)
{ int i;
  i = 0;
  do
        {
              i = i + 1;
        }
  while (i<= n || s[i] != x);
  return i;
}
```

Ví dụ này ta không thể đánh giá như hai ví dụ trên. Do quá trình tìm kiếm không những phụ thuộc vào kích thước của dữ liệu vào, mà còn phụ thuộc vào tình trạng của dữ liệu. Tức là thời gian thực hiện giải thuật còn phụ thuộc vào vị trí của phần tử trong dãy bằng x. Quá trình tìm kiếm chỉ dừng khi tìm thấy phần tử bằng x, hoặc duyệt hết dãy mà không tìm thấy. Vì vậy, trong những trường hợp như trên ta cần phải đánh giá thời gian tính tốt nhất, tồi nhất và trung bình của giải thuật với kích thước đầu vào n. Rõ ràng thời gian tính của giải thuật có thể được đánh giá bởi số lần thực hiện câu lệnh i = i + 1 (gọi là phép toán tích cực).

Nếu s[1] = x thì câu lệnh i = i + 1 trong thân vòng lặp ***do...while*** thực hiện 1 lần. Do đó thời gian tính tốt nhất của giải thuật là O(1).

Nếu x không xuất hiện trong dãy khoá đã cho, thì câu lệnh i = i + 1 được thực hiện n lần. Vì thế thời gian tính tồi nhất là O(n).

Cuối cùng ta tính thời gian tính trung bình của giải thuật. Nếu x được tìm thấy ở vị trí thứ i của dãy thì câu lệnh i = i + 1 phải thực hiện i lần (i = 1, 2,..., n), còn nếu x không xuất hiện trong dãy thì câu lệnh i = i + 1 phải thực hiện n lần.

Từ đó suy ra số lần trung bình phải thực hiện câu lệnh i = i + 1 là:

$$[(1 + 2 + ... + n) + n] /(n + 1)$$

Ta có:

$$[(1 + 2 + ... + n) + n] /(n + 1) \leq (n^2 + n)/(n + 1) = n$$

Vậy thời gian tính trung bình của giải thuật là O(n).

Nhận xét:

Việc xác định T(n) trong trường hợp trung bình thường gặp nhiều khó khăn vì sẽ phải dùng tới công cụ toán đặc biệt, hơn nữa tính trung bình có nhiều cách quan niệm. Trong các trường hợp mà T(n) trung bình thường khó xác định, người ta thường đánh giá giải thuật qua giá trị xấu nhất của T(n). Hơn nữa, trong một số lớp giải thuật, việc xác định trường hợp xấu nhất là rất quan trọng.

Qua chương này ta thấy rằng giai đoạn thiết kế hoặc lựa chọn các cấu trúc dữ liệu và các giải thuật khi phát triển dự án phần mềm là hết sức quan trọng, nó quyết định một phần sự thành bại của một dự án tin học. Chất lượng của dự án cũng phụ thuộc vào việc đánh giá các cấu trúc dữ liệu và các giải thuật khi chúng được thiết kế và lựa chọn.

Trong các chương sau, cuốn sách sẽ trình bày một số các giải thuật và cấu trúc dữ liệu cơ bản, được áp dụng chủ yếu trong các bài toán quản lý như các giải thuật tìm kiếm, sắp xếp, cấu trúc danh sách tuyến tính, cây, v.v... cùng với cách thiết kế, đánh giá và áp dụng chúng trong bài toán thực tế.

BÀI TẬP CHƯƠNG 1

1. Nêu khái niệm thuật toán, các đặc trưng và các phương pháp biểu diễn thuật toán. Cho ví dụ minh họa.
2. Phân biệt khái niệm thuật toán và thuật giải. Cho ví dụ minh họa.
3. Tìm thêm các ví dụ minh họa mối quan hệ giữa cấu trúc dữ liệu và giải thuật (kiểu số nguyên, kiểu số thực,...).
4. Phân biệt cấu trúc dữ liệu và cấu trúc lưu trữ. Cho ví dụ minh họa.
5. Nêu nguyên tắc của phương pháp thiết kế top-down. Cho ví dụ minh họa.
6. Chứng minh rằng nếu $T(n) = O(n)$ thì $1(n) = O(n^2)$.
7. Cho $f(n) = 10n^2 + n - 5$. Chứng minh rằng $f(n) = O(n^2)$.
8. Chứng minh rằng $\lg n! = O(n \lg n)$.
9. Với các đoạn chương trình dưới đây, hãy xác định về thời gian của giải thuật bằng ký pháp chữ O lớn.

a.

```
for (i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
                x = x + 1 ;
```

b.

```
j = n;
while (j >= 1)
        {
                for (i = 1; i <= j; i++)
                        x = x + 1;
                        j = j/2;
        }
```

10. Cho $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ ($a_n \neq 0$). Chứng minh rằng $f(x) = O(x^n)$.

Chương 2

ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

1. KHÁI NIỆM VỀ ĐỆ QUY

Ta nói một đối tượng là đệ quy nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Ví dụ: Trong toán học ta gặp các định nghĩa đệ quy sau:

+ *Số tự nhiên:*

- 1 là số tự nhiên.

- n là số tự nhiên nếu n-1 là số tự nhiên.

+ *Hàm n giai thừa: n!*

- 0! = 1

- Nếu n > 0 thì n! = n(n-1)!

2. GIẢI THUẬT ĐỆ QUY VÀ THỦ TỤC ĐỆ QUY

2.1. Giải thuật đệ quy

Nếu lời giải của của một bài toán T được giải bằng lời giải của một bài toán T1, có dạng giống như T, thì lời giải đó được gọi là lời giải đệ quy. Giải thuật tương ứng với lời giải đệ quy gọi là giải thuật đệ quy.

Ở đây T1 có dạng giống T nhưng theo một nghĩa nào đó T1 phải "nhỏ" hơn T.

Chẳng hạn với bài toán tính n!, thì tính n! là bài toán T còn tính (n-1)! là bài toán T1 ta thấy T1 cùng dạng với T nhưng nhỏ hơn (n-1 < n).

Hay với bài toán tìm một từ trong quyển từ điển. Có thể nêu giải thuật như sau:

if (từ điển là một trang)

tìm từ trong trang này

else

{ *Mở từ điển vào trang "giữa"*

Xác định xem nửa nào của từ điển chứa từ cần tìm;

if *(từ đó nằm ở nửa trước) tìm từ đó ở nửa trước*

else *tìm từ đó ở nửa sau.*

}

Giải thuật này được gọi là giải thuật đệ quy. Việc tìm từ trong quyển từ điển được được giải quyết bằng bài toán nhỏ hơn đó là việc tìm từ trong một nửa thích hợp của quyển từ điển.

Ta thấy có hai điểm chính cần lưu ý:

1. Sau mỗi lần từ điển được tách làm đôi thì một nửa thích hợp sẽ lại được tìm bằng một chiến thuật như đã dùng trước đó (nửa này lại được tách đôi).

2. Có một trường hợp đặc biệt, đó là sau nhiều lần tách đôi từ điển chỉ còn một trang. Khi đó việc tách đôi ngừng lại và bài toán trở thành đủ nhỏ để ta có thể tìm từ mong muốn bằng cách tìm tuần tự. Trường hợp này gọi là ***trường hợp suy biến***.

2.2. Thủ tục đệ quy

Với giải thuật tìm kiếm như trên ta viết một thủ tục tương ứng như sau:

SEARCH(dict, word) //Tìm từ word trong từ điển dict

{

if (Từ điển chỉ còn là một trang)

tìm từ word trong trang này

else

{ mở từ điển vào trang giữa

```
            xác định xem nửa nào của từ điển chứa từ word
            if (từ word nằm ở nửa trước của từ điển)
                    return SEARCH(dict\{nửa sau}, word);
            else    return SEARCH(dict\{nửa trước}, word);
    }
}
```

Thủ tục trên được gọi là thủ thục đệ quy. Nó có những đặc điểm cơ bản sau:

1. Trong thủ tục đệ quy có lời gọi đến chính thủ tục đó. Ở đây trong thủ tục SEARCH có lời gọi call SEARCH (lời gọi này được gọi là lời gọi đệ quy).

2. Sau mỗi lần có lời gọi đệ quy thì kích thước của bài toán được thu nhỏ hơn trước. Ở đây khi có lời gọi call SEARCH thì kích thước từ điển chỉ còn bằng một nửa so với trước đó.

3. Có một trường hợp đặc biệt, trường hợp suy biến là khi lời gọi call SEARCH với từ điển dict chỉ còn là một trang. Khi trường hợp này xảy ra thì bài toán còn lại sẽ được giải quyết theo một cách khác hẳn (tìm từ word trong trang đó bằng cách tìm kiếm tuần tự) và việc gọi đệ quy cũng kết thúc. Chính tình trạng kích thước bài toán giảm dần sau mỗi lần gọi đệ quy, dẫn tới trường hợp suy biến.

Một số ngôn ngữ cấp cao như: Pascal, C, Algol, v.v... cho phép viết các thủ tục đệ quy. Nếu thủ tục đệ quy chứa lời gọi đến chính nó thì gọi là đệ quy trực tiếp. Cũng có trường hợp thủ tục chứa lời gọi đến thủ tục khác mà ở thủ tục này lại chứa lời gọi đến nó. Trường hợp này gọi là đệ quy gián tiếp.

3. THIẾT KẾ GIẢI THUẬT ĐỆ QUY

Khi bài toán đang xét hoặc dữ liệu đang xử lý được định nghĩa dưới dạng đệ quy thì việc thiết kế các giải thuật đệ quy tỏ ra rất thuận lợi. Hầu như nó phản ánh rất sát nội dung của định nghĩa đó.

Không có giải thuật đệ quy vạn năng cho tất cả các bài toán đệ quy, nghĩa là mỗi bài toán cần thiết kế một giải thuật đệ quy riêng.

Ta xét một số bài toán sau:

3.1. Hàm n!

Hàm này được định nghĩa như sau:

$$Factorial(n) = \begin{cases} 1 & \text{nếu } n = 0 \\ n * \text{Factorial}(n - 1) & \text{nếu } n > 0 \end{cases}$$

Giải thuật đệ quy được viết dưới dạng hàm dưới đây:

```
Factorial (n)
{
        if (n==0) return 1;
        else    return n*Factorial(n-1);
}
```

Trong hàm trên lời gọi đến nó nằm ở câu lệnh gán sau ***else***.

Mỗi lần gọi đệ quy đến Factorial, thì giá trị của n giảm đi 1. Ví dụ, Factorial(4) gọi đến Factorial(3), gọi đến Factorial(2), gọi đến Factorial(1), gọi đến Factorial(0) đây là trường hợp suy biến, nó được tính theo cách đặc biệt Factorial(0) = 1.

3.2. Bài toán dãy số FIBONACCI

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán được đặt ra như sau:

- Các con thỏ không bao giờ chết.

- Hai tháng sau khi ra đời một cặp thỏ mới sẽ sinh ra một cặp thỏ con.

- Khi đã sinh, thì cứ sau mỗi tháng chúng lại sinh được một cặp con mới.

Giả sử bắt đầu từ một cặp thỏ mới sinh, hỏi đến tháng thứ n sẽ có bao nhiêu cặp?

Ví dụ: Với n = 6, ta thấy:

Tháng thứ 1: 1 cặp (cặp ban đầu)

Tháng thứ 2: 1 cặp (cặp ban đầu vẫn chưa sinh con)

Tháng thứ 3: 2 cặp (đã có thêm 1 cặp con do cặp ban đầu sinh ra)

Tháng thứ 4: 3 cặp (cặp ban đầu vẫn sinh thêm)

Tháng thứ 5: 5 cặp (cặp con bắt đầu sinh)

Tháng thứ 6: 8 cặp (cặp con vẫn sinh tiếp)

Đặt F(n) là số cặp thỏ ở tháng thứ n. Ta thấy chỉ những cặp thỏ đã có ở tháng thứ n-2 mới sinh con ở tháng thứ n do đó số cặp thỏ ở tháng thứ n là:

F(n) = F(n-2) + F(n-1), vì vậy F(n) có thể được tính như sau:

$$F(n) = \begin{cases} 1 & \text{nếu } n \le 2 \\ F(n-2) + F(n-1) & \text{nếu } n > 2 \end{cases}$$

Dãy số thể hiện F(n) ứng với các giá trị của n = 1, 2, 3, 4..., có dạng

1 1 2 3 5 8 13 21 34 55....

nó được gọi là dãy số Fibonacci. Nó là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học.

Sau đây là giải thuật đệ quy dạng hàm thể hiện việc tính F(n).

```
Fibonacci (n)
{       if (n<=2)       return 1;
        else            return Fibonacci(n-2) + Fibonacci(n-1);
}
```

Ở đây trường hợp suy biến ứng với 2 giá trị F(1) = 1 và F(2) = 1.

Chú ý:

Đối với hai bài toán nêu trên thì việc thiết kế các giải thuật đệ quy tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa của nó xác định được một cách dễ dàng.

Nhưng không phải lúc nào tính đệ quy trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy. Mà việc thiết kế một giải thuật đệ quy đòi hỏi phải giải đáp được các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng nhỏ hơn như thế nào?

- Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ quy?

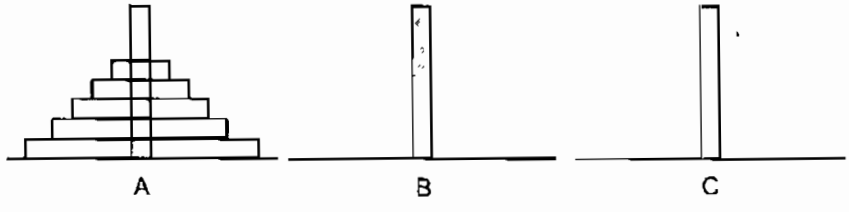
- Trường hợp đặc biệt nào của bài toán được gọi là trường hợp suy biến?

Sau đây ta xét thêm bài toán phức tạp hơn.

3.4. Bài toán "Tháp Hà Nội"

Bài toán này mang tính chất là một trò chơi, nội dung như sau:

Có n đĩa, kích thước nhỏ dần, mỗi đĩa có lỗ ở giữa. Có thể xếp chồng chúng lên nhau xuyên qua một cọc, đĩa to ở dưới, đĩa nhỏ ở trên để cuối cùng có một chồng đĩa dạng như hình tháp như hình 2.1.



Hình 2.1: Chồng đĩa trước khi chuyển

**** Yêu cầu đặt ra là:***

Chuyển chồng đĩa từ cọc A sang cọc khác, chẳng hạn cọc C, theo những điều kiện:

- Mỗi lần chỉ được chuyển một đĩa.

- Không khi nào có tình huống đĩa to ở trên đĩa nhỏ (dù là tạm thời).

- Được phép sử dụng một cọc trung gian, chẳng hạn cọc B để đặt tạm đĩa.

Để đi tới cách giải tổng quát, trước hết ta xét vài trường hợp đơn giản.

** Trường hợp có 1 đĩa:*

- Chuyển đĩa từ cọc A sang cọc C.

** Trường hợp 2 đĩa:*

- Chuyển đĩa thứ nhất từ cọc A sang cọc B.

- Chuyển đĩa thứ hai từ cọc A sang cọc C.

- Chuyển đĩa thứ nhất từ cọc B sang cọc C.

Ta thấy với trường hợp n đĩa (n>2) nếu coi n-1 đĩa ở trên, đóng vai trò như đĩa thứ nhất thì có thể xử lý giống như trường hợp 2 đĩa được, nghĩa là:

- Chuyển n-1 đĩa trên từ A sang B.

- Chuyển đĩa thứ n từ A sang C.

- Chuyển n-1 đĩa từ B sang C.

Lược đồ thể hiện 3 bước này như hình 5.1.

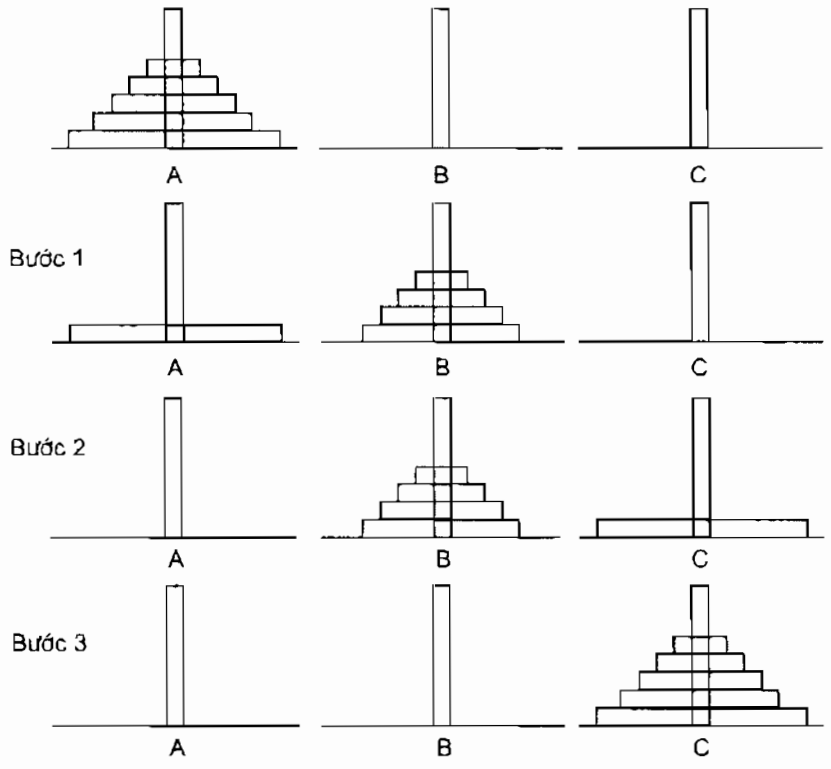
Như vậy, bài toán ***"Tháp Hà Nội"*** tổng quát với n đĩa đã được dẫn đến bài toán tương tự với kích thước nhỏ hơn, chẳng hạn từ chỗ chuyển n đĩa từ cọc A sang cọc C nay là chuyển n-1 đĩa từ cọc A sang cọc B và ở mức này thì giải thuật lại là:

- Chuyển n-2 đĩa từ cọc A sang cọc C.

- Chuyển 1 đĩa từ cọc A sang cọc B.

- Chuyển n-2 đĩa từ cọc B sang cọc C.

và cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chuyển 1 đĩa.



Hình 2.2: Các bước chuyển đĩa

Vậy thì các đặc điểm của đệ quy trong giải thuật đã được xác định và ta có thể viết giải thuật đệ quy của bài toán "Tháp Hà Nội" như sau:

```
Chuyen(n, A, B, C)
{
        if (n= =1) chuyển đĩa từ A sang C;
        else
        {       call Chuyen(n-1, A, C, B);
                call Chuyen(1, A, B, C);
                call Chuyen(n-1, B, A, C);
        }
}
```

Bạn có thể tự cài đặt giải thuật này với cấu trúc dữ liệu biểu diễn bằng mảng một chiều, nếu tốt hơn có thể biểu diễn giải thuật bằng đồ họa.

4. HIỆU LỰC CỦA ĐỆ QUY

Qua các ví dụ trên ta có thể thấy: đệ quy là một công cụ để giải quyết các bài toán. Có những bài toán, bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn giải thuật lặp tính n! có thể viết:

```
Factorial(n)
{
        if (n==0 || n==1) return 1;
  else
  {     gt=1;
        for (i=2; i<=n; i++)
                gt = gt*i;
        return gt;
  }
}
```

Hoặc ta xét giải thuật lặp tính số Fibonacci thứ n:

```
Fibonacci(n)
{
  if (n<=2)         return 1;
  else
  {     Fib1 = 1; Fib2 = 1;
        for (i=3; i<=n; i++)
        {
                Fibn = Fib1 + Fib2;
                Fib1= Fib2;
                Fib2 = Fibn;
        }
        return Fibn;
  }
}
```

Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó. Có những bài toán việc nghĩ ra giải thuật đệ quy thuận lợi hơn nhiều so với giải thuật lặp và có những giải thuật đệ quy thực sự có hiệu lực cao, chẳng hạn giải thuật sắp xếp kiểu phân đoạn (Quick Sort) hoặc các giải thuật duyệt cây nhị phân mà ta đã có dịp xét trong các chương trước của môn học này.

Một điều nữa cần nói thêm là: về mặt định nghĩa, công cụ đệ quy đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Ta sẽ thấy vai trò của công cụ này trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu v.v...

Chú thích: khi thay các giải thuật đệ quy bằng các giải thuật lặp tương ứng ta gọi là khử đệ quy. Tuy nhiên, có những bài toán việc khử đệ quy tương đối đơn giản (ví dụ: giải thuật tính n!, tính số fibonaci...), nhưng có những bài toán việc khử đệ quy là rất phức tạp (ví dụ: bài toán Tháp Hà Nội, giải thuật sắp xếp phân đoạn...).

BÀI TẬP CHƯƠNG 2

Bài 1: Xét định nghĩa đệ quy:

$$\text{Acker}(m, n) = \begin{cases} n + 1 \text{ nếu } m = 0 \\ \text{Acker}(m - 1, 1) \text{ nếu } n = 0 \\ \text{Acker}(m - 1, \text{Acker}(m, n - 1)) \text{ với các trường hợp khác} \end{cases}$$

- Hãy xác định Acker(1, 2)
- Viết hàm đệ quy thực hiện tính giá trị của hàm này.

Bài 2: Cho dãy số A = {4, 3, -2, -6, -5, 0, 4, ...}

- Xây dựng định nghĩa đệ quy cho việc tính A_n ($n \geq 0$)
- Xây dựng giải thuật đệ quy tính A_n
- Tính A_{10}.

Bài 3: Cho hàm số:

$$F(x) = \begin{cases} \cos(x) & \text{nếu } x = 0 \\ x & \text{nếu } x < 0 \\ F(x - \pi) + F(x - \pi/2) & \text{trong các trường hợp còn lại} \end{cases}$$

Yêu cầu:

- Tính $F(5\pi/2)$ và giải thích cách tính
- Thiết kế giải thuật đệ quy và viết hàm đệ quy để tính giá trị của hàm F.

Bài 4: Giải thuật tính ước số chung lớn nhất của hai số nguyên dương p, q (p > q) được cho như sau:

Gọi r là số dư trong phép chia p cho q.

Nếu r = 0 thì ước số chung lớn nhất là q.

Nếu r ≠ 0 thì gán cho p giá trị q, gán cho q giá trị của r và lặp lại quá trình.

a. Hãy xây dựng định nghĩa đệ quy cho hàm USCLN(p, q).

b. Viết một giải thuật đệ quy và một giải thuật lặp thể hiện hàm đó.

c. Hãy nêu rõ các đặc điểm của một giải thuật đệ quy được thể hiện trong trường hợp này.

d. Xử lý trường hợp p < q.

Bài 5: Nêu rõ các bước thực hiện khi có lời gọi hàm chuyen(3, A, B, C) trong bài toán Tháp Hà Nội.

Bài 6: Viết một hàm lặp và một hàm đệ quy thực hiện việc in ngược một chuỗi ký tự. Ví dụ "PASCAL" thì in ra là "LACSAP".

Bài 7: Viết một hàm lặp và một hàm đệ quy thực hiện việc đếm số chữ số của một số nguyên dương.

Bài 8: Cài đặt chương trình cho bài toán Tháp Hà Nội với cấu trúc dữ liệu là mảng một chiều (nếu xây dựng được chương trình biểu diễn bài toán bằng đồ họa thì càng tốt).

Chương 3 SẮP XẾP VÀ TÌM KIẾM

Sắp xếp và tìm kiếm là những vấn đề hết sức quen thuộc với chúng ta những người học và làm việc trong ngành công nghệ thông tin. Hơn nữa, với những người làm công tác quản lý hồ sơ, đây là công việc mà họ thường xuyên phải thực hiện, nó tiêu tốn khá nhiều thời gian. Nhưng với sự hỗ trợ của phần mềm tin học, công việc này trở nên rất đơn giản và rất nhanh, chỉ bằng những thao tác lựa chọn trên màn hình máy tính. Tuy nhiên, với chúng ta, những người làm tin học cần phải tìm hiểu gốc rễ của vấn đề, đó là các giải thuật được sử dụng để cài đặt cho máy tính, chúng sẽ được trình bày cụ thể trong chương này.

1. CÁC PHƯƠNG PHÁP SẮP XẾP

1.1. Khái niệm sắp xếp

Yêu cầu sắp xếp thường xuyên xuất hiện trong các ứng dụng tin học với các mục đích khác nhau, sắp xếp dữ liệu lưu trữ trong máy tính để thuận lợi cho việc tìm kiếm, sắp xếp các kết quả để in ra trên bảng biểu, v.v...

1.1.1. Khái niệm

Sắp xếp được hiểu là một quá trình bố trí lại vị trí các phần tử của một tập đối tượng nào đó theo một thứ tự xác định, chẳng hạn như thứ tự tăng dần của một dãy số, thứ tự từ điển đối với các chữ cái,...

Nhìn chung, dữ liệu được sắp xếp có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước tập đối tượng sắp xếp là tập các bản ghi, mỗi bản ghi gồm một số trường, dữ liệu tương ứng với

những thuộc tính khác nhau. Tuy nhiên, không phải toàn bộ các trường dữ liệu của bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ dựa vào một hoặc một vài trường nào đó, trường như vậy được gọi là khoá. Quá trình sắp xếp sẽ được tiến hành dựa vào giá trị của khoá, vì vậy bài toán sắp xếp được phát biểu như sau:

1.1.2. Phát biểu bài toán sắp xếp

Vào: Dãy n đối tượng, mỗi đối tượng có một khóa sắp xếp

Ra: Dãy đối tượng với trật tự mới

Khóa sắp xếp ở đây phụ thuộc vào mục đích sắp xếp và dữ liệu cần sắp xếp. Chẳng hạn, sắp xếp dãy số thì giá trị của các số là khóa, còn sắp xếp danh sách sinh viên theo tên, thì tên của sinh viên là khóa sắp xếp, v.v...

Dưới đây ta sẽ tìm hiểu một số phương pháp sắp xếp hay dùng.

1.2. Ba phương pháp sắp xếp đơn giản

Để đơn giản, nhưng không mất tính tổng quát, chúng tôi mô tả các phương pháp sắp xếp thông qua bài toán sắp xếp dãy số nguyên, thực chất là sắp xếp mảng.

Bài toán sắp xếp:

Vào: Dãy **X** có **n** số nguyên: $X_1, X_2, \ldots, X_n$

Ra: Dãy **X** với các phần tử theo thứ tự tăng dần

1.2.1. Phương pháp lựa chọn (Selection Sort)

Một trong những phương pháp đơn giản nhất để thực hiện sắp xếp một mảng là sắp xếp dựa trên phép lựa chọn.

a. Nguyên tắc sắp xếp

Để sắp xếp dãy X theo chiều tăng dần người ta làm như sau:

- Thực hiện n-1 lần duyệt dãy từ trái sang phải.

- Mỗi lần duyệt, chọn phần tử nhỏ nhất, đổi chỗ cho phần tử đầu dãy được xét.

Cụ thể như sau:

Ở lần duyệt thứ i (i = 1, 2,..., n-1), ta duyệt dãy từ X_i đến X_n và chọn khoá có giá nhỏ nhất, giả sử là X_m (m = i, i+1,..., n) và đổi chỗ X_m với X_i.

Như vậy: sau i lần duyệt, i khoá nhỏ hơn đã lần lượt ở các vị trí thứ nhất, thứ hai,..., thứ i theo đúng thứ tự sắp xếp.

Phương pháp sắp xếp lựa chọn được mô tả qua ví dụ sau:

Ví dụ:

Bảng 3.1: Mô tả phương pháp sắp xếp lựa chọn

Lần duyệt	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	**11**	65	58	34	
i = 1	11	**23**	74	42	65	58	34	Duyệt từ X_1 đến X_7, X_4 nhỏ nhất đổi chỗ cho X_1
i = 2	11	23	74	42	65	58	**34**	Duyệt từ X_2 đến X_7, X_2 nhỏ nhất. Đây là trường hợp đặc biệt, phần tử nhỏ nhất là phần tử đầu dãy. Vì thế thực tế không cần đổi chỗ
i = 3	11	23	34	**42**	65	58	74	Duyệt từ X_3 đến X_7, X_7 nhỏ nhất đổi chỗ cho X_3
i = 4	11	23	34	42	65	**58**	74	Giống trường hợp i = 2
i = 5	11	23	34	42	58	**65**	74	Duyệt từ X_5 đến X_7, X_6 nhỏ nhất đổi chỗ cho X_5
i = 6	11	23	34	42	58	65	74	Giống trường hợp i = 2

Trong mô tả trên, vị trí gạch chân là phần tử đầu dãy được xét, vị trí in đậm là phần tử có giá trị nhỏ nhất trong dãy được xét.

Bạn đọc có thể tự mình lấy ví dụ minh họa trong trường hợp dãy được sắp xếp theo chiều giảm dần.

b. Thiết kế giải thuật

Với nguyên tắc sắp xếp như trên, việc thiết kế giải thuật khá đơn giản với kỹ thuật lặp gồm các bước như sau:

1. Đếm số lần duyệt

for (i=1; i<=n-1; i++)

2. Duyệt dãy và chọn phần tử nhỏ nhất (*tương tự như giải thuật tìm max*).

Đặt m = i; //Giả định X_i là phần tử nhỏ nhất

for (j=i+1; j<=n; j++)

if ($X_j < X_m$) m = j;

3. Đổi chỗ X_m và X_i

Dưới đây là giải thuật sắp xếp bằng phương pháp lựa chọn, trong trường hợp dãy được sắp theo chiều tăng dần:

```
void SELECTION_SORT (int X[ ], int n)
//Sắp xếp dãy khóa X có n phần tử
{
  for (i = 0; i < n-1; i++)
  {
        m = i;
        for (j = i+1; j < n; j++)
              if (X[j] < X[m] )
                    m = j;
        if (m != i )
        {
              tg = X[i];
              X[i] = X[m];
              X[m] = tg;
        }
  }
}
```

1.2.2. Sắp xếp kiểu thêm dần (Insertion Sort)

a. Nguyên tắc sắp xếp:

Nguyên tắc sắp xếp của phương pháp này dựa theo kinh nghiệm của những người chơi bài. Khi có i-1 lá bài đã được sắp xếp ở trên tay, nếu rút thêm lá bài thứ i nữa thì sắp xếp thế nào? Câu trả lời là có thể so sánh lá bài mới lần lượt với lá bài thứ i-1, thứ i-2,... để tìm ra vị trí thích hợp của lá bài mới và chèn vào vị trí đó.

Dựa trên nguyên tắc này, có thể triển khai một cách sắp xếp như sau:

- Đầu tiên, dãy được coi chỉ gồm một khóa X_1 đã được sắp xếp.

- Xét thêm X_2, so sánh X_2 với X_1 để xác định vị trí chèn X_2, ta được một dãy gồm 2 khóa đã được sắp xếp.

- Xét thêm X_3, lại so sánh X_3 với X_2 và X_1 để xác định vị trí chèn X_3, tương tự như vậy đối với các khoá X_4, X_5,..., và cuối cùng là X_n, ta được bảng khoá đã sắp xếp hoàn toàn.

Có thể mô tả ngắn gọn phương pháp này như sau:

Để sắp xếp dãy X, có n phần tử X_1, X_2,..., X_n ta thực hiện n-1 lần chia dãy thành dãy đích và dãy nguồn:

- Dãy đích gồm các phần tử từ X_1 đến X_i (với i = 1, 2,..., n-1)

- Dãy nguồn gồm các phần tử từ X_{i+1} đến X_n

Mỗi lần chia lấy phần tử đầu dãy nguồn (là X_{i+1}) chèn vào vị trí thích hợp trong dãy đích.

Ví dụ:

Bảng 3.2: Mô tả phương pháp sắp xếp thêm dần

Lần chia	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	11	65	58	34	
i = 1	42 →	**23**	74	11	65	58	34	Lấy t = X_2 = 23 chèn vào dãy đích
	42	42	74	11	65	58	34	
i = 2	23 →	42	**74**	11	65	58	34	Lấy t = X_3 = 74 chèn vào dãy đích
i = 3	23 →	42 →	74 →	**11**	65	58	34	Lấy t = X_4 = 11 chèn vào dãy đích
	23	23	42	74	65	58	34	
i = 4	11	23	42	74 →	**65**	58	34	Lấy t = X_5 = 65 chèn vào dãy đích
	11	23	42	***74***	74	58	34	
i = 5	11	23	42	65 →	74 →	**58**	34	Lấy t = X_6 = 58 chèn vào dãy đích
	11	23	42	***65***	65	74	34	
i = 6	11	23	42 →	58 →	65 →	74 →	**34**	Lấy t = X_7 = 34 chèn vào dãy đích
	11	23	***42***	42	58	65	74	
	11	23	34	42	58	65	74	Dãy được sắp

Trong mô tả này, các phần tử in đậm là dãy đích, phần tử gạch chân-in đậm là phần tử đầu dãy nguồn, các mũi tên chỉ việc dịch các phần tử sang phải, phần tử nghiêng-đậm là vị trí thích hợp trong dãy đích, để chèn phần tử đầu dãy nguồn.

Tương tự, bạn đọc có thể tự mô tả trường hợp dãy được sắp xếp theo chiều giảm dần.

b. Giải thuật:

Qua mô tả trên ta xây dựng giải thuật sắp xếp bằng phương pháp thêm dần dưới dạng một hàm như sau:

```
void INSERTION_SORT (int X[ ], int n)
{
  for (i = 0; i < n; i++)
  {
        t = X[i+1];
        j = i;
        while (X[j] > 0 && j > -1)
        {
              X[j+1] = X[j];
              j = j-1;
        }
        X[j+1] = t;
  }
}
```

1.2.3. Sắp xếp kiểu nổi bọt (Bubble Sort)

Trong hai phương pháp sắp xếp nêu trên, kỹ thuật đổi chỗ đều đã được sử dụng nhưng chưa trở thành điểm nổi bật. Ở phương pháp này, việc đổi chỗ các cặp khoá kế cận khi chúng ngược thứ tự sẽ được thực hiện thường xuyên cho tới khi toàn bộ dãy khoá được sắp xếp.

a. Nguyên tắc sắp xếp:

Với dãy X đã cho gồm n phần tử, để sắp xếp dãy theo chiều tăng dần người ta làm như sau:

- Thực hiện n-1 lần duyệt dãy từ trái sang phải.

- Mỗi lần duyệt, lần lượt so sánh các cặp phần tử liên tiếp, giả sử là X_j và X_{j+1} nếu chúng ngược thứ tự thì đổi chỗ.

Nguyên tắc nêu trên được mô tả trong bảng sau:

Bảng 3.3: Mô tả phương pháp sắp xếp nổi bọt

Lần duyệt	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	11	65	58	34	
i = 1	**42**	**23**	74	11	65	58	34	So sánh cặp X_1, X_2 và đổi chỗ
	23	**42**	**74**	11	65	58	34	So sánh cặp X_2, X_3 và đổi chỗ
	23	42	**74**	**11**	65	58	34	So sánh cặp X_3, X_4 và đổi chỗ
	23	42	11	**74**	**65**	58	34	So sánh cặp X_4, X_5 và đổi chỗ
	23	42	11	65	**74**	**58**	34	So sánh cặp X_5, X_6 và đổi chỗ
	23	42	11	65	58	**74**	**34**	So sánh cặp X_6, X_7 và đổi chỗ
	23	42	11	65	58	34	**74**	Phần tử lớn nhất ở cuối dãy
i = 2	23	11	42	58	34	**65**	74	Tương tự với lần duyệt i = 1
i = 3	11	23	42	24	**58**	65	74	
i = 4	11	23	24	**42**	58	65	74	
i = 5	11	23	**24**	42	58	65	74	
i = 6	11	**23**	24	42	58	65	74	Dãy được sắp

Qua mô tả trên ta thấy:

- Sau mỗi lần duyệt ta được 1 phần tử đứng đúng vị trí.

- Cụ thể: sau lần duyệt thứ nhất (i = 1) ta có phần tử lớn nhất đứng cuối dãy.

- Ở lần duyệt thứ 2 (i = 2) ta chỉ cần xét các phần tử từ X_1 đến X_{n-1} (làm tương tự lần duyệt 1 ta có phần tử lớn nhất trong số n-1 phần tử còn lại đứng ở vị trí n-1)

- Vậy ở lần duyệt thứ i (i = 1, 2,..., n-1) ta chỉ xét n-i+1 phần tử đầu dãy.

- Vì so sánh các cặp liên tiếp nên trong lần duyệt thứ i (i = 1, 2,..., n-1) ta chỉ duyệt từ X_1 đến X_{n-i}.

Ta có giải thuật như sau:

b. Giải thuật:

```
void BUBBLE_SORT (int X[ ], int n)
{
  for (i = 1; i <= n-1; i++)
        for (j = 0; j < n-i; j++)
              if (X[j] > X[j + 1])
              {
                    tg = X[j]; X[j] = X[j + 1]; X[j + 1] = tg;
              }
}
```

Trên đây ta vừa tìm hiểu ba phương pháp sắp xếp đơn giản, ta sẽ xem xét tính hiệu quả của nó trong phần dưới đây.

1.2.4. Đánh giá ba phương pháp sắp xếp đơn giản

Đối với một số phương pháp sắp xếp, khi xét tới hiệu lực của nó, ngoài những đánh giá về mặt không gian nhớ cần thiết, người ta thường lưu ý đặc biệt tới chi phí về thời gian. Mà thời gian thì chủ yếu phụ thuộc vào việc thực hiện các phép so sánh giá trị khoá và các phép chuyển chỗ bản ghi khi sắp xếp. Vì vậy thông thường người ta lấy số lượng trung bình các phép so sánh và các phép chuyển chỗ làm đại lượng đặc trưng cho chi phí về thời gian thực hiện của từng phương pháp. Tuy nhiên, ở đây ta chỉ xét tới phép so sánh và coi nó như một "phép toán tích cực" của các giải thuật sắp xếp.

Đối với phương pháp sắp xếp kiểu lựa chọn, ta thấy: ở lượt thứ i (i = 1, 2,..., n-1) để tìm khoá nhỏ nhất bao giờ cũng cần $C_i = (n-i)$ phép so sánh. Số lượng phép so sánh này không phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra:

$$C_{min} = C_{max} = C_{tb} = \sum_{i=1}^{n-1}(n-1) = \frac{n(n-1)}{2}$$

Còn với phương pháp sắp xếp kiểu thêm dần (giải thuật INSERTION_SORT) thì có hơi khác. Rõ ràng số lượng phép so sánh phụ thuộc vào dãy khoá ban đầu. Trường hợp thuận lợi nhất ứng với dãy khoá đã được sắp xếp rồi. Như vậy ở mỗi lượt chỉ cần 1 phép so sánh. Do đó

$$C_{min} = \sum_{i=2}^{n} 1 = n-1$$

Nhưng nếu dãy khoá ban đầu có thứ tự ngược với thứ tự sắp xếp thì ở lượt thứ i phải cần có: $C_i = (i-1)$ phép so sánh. Vì vậy:

$$C_{max} = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

Nếu giả sử mọi giá trị khoá đều xuất hiện đồng khả năng thì trung bình ở lượt thứ i có thể coi như $C_i = i/2$ phép so sánh. Suy ra:

$$C_{tb} = \sum_{i=2}^{n}\frac{i}{2} = \frac{n(n-1)}{2}$$

Nhìn vào các kết quả đánh giá ở trên ta thấy INSERTION_SORT tỏ ra có "tốt hơn" so với hai phương pháp kia. Tuy nhiên, với n khá lớn, chi phí về thời gian thực hiện được đánh giá qua cấp độ lớn, thì cả ba phương pháp đều có cấp $O(n^2)$ và đây vẫn là một chi phí cao so với một số phương pháp mà ta sẽ xét thêm sau đây.

1.3. Sắp xếp phân đoạn

1.3.1. Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn (PARTITION_SORT) là một cải tiến của phương pháp sắp xếp đổi chỗ và là một phương pháp sắp xếp khá tốt. Chính vì thế, C.A.R. Hoare người tạo ra phương pháp này đã đặt tên cho nó là sắp xếp nhanh (QUICK_SORT).

Nguyên tắc của phương pháp này có thể được mô tả như sau:

- Chọn một khoá làm "*chốt*", thông thường để thuận lợi cho việc cài đặt, người ta chọn khóa trung tâm của "*đoạn được xét*" làm chốt.

- Xếp các khoá nhỏ hơn chốt về bên trái chốt (phía đầu dãy), các khoá lớn hơn chốt về bên phải chốt (phía cuối dãy) bằng cách:

Các phần tử trong dãy được so sánh với khoá chốt và sẽ đổi vị trí cho nhau hoặc cho chốt nếu chúng nằm trước chốt mà lại lớn hơn chốt hoặc nằm sau chốt mà lại nhỏ hơn chốt.

- Kết thúc một lượt đổi chỗ, dãy khoá được chia thành hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt và (có thể) khoá chốt nằm ở giữa hai đoạn, cũng chính là vị trí đúng của nó trong dãy.

Lặp lại quá trình trên đối với từng phân đoạn cho đến khi toàn bộ dãy khoá được sắp xếp. Ở các lượt xử lý tiếp theo, chỉ có một phân đoạn được xử lý, còn phân đoạn còn lại phải được ghi nhớ và xử lý sau.

1.3.2. Minh hoạ phương pháp phân đoạn

Sắp xếp dãy khoá X:

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	65	58	94	36	99	87

Với quy ước chọn khoá chốt là khóa trung tâm trong dãy đang xét, như vậy với dãy khoá ban đầu thì khoá chốt là $t = X_5 = 65$.

Giả sử cần sắp xếp một đoạn từ X_{left} đến X_{right} trên dãy. Để xác định vị trí các khoá cần đổi chỗ cho nhau, ta dùng hai biến i và j để duyệt từ hai đầu của dãy khóa như sau:

(1) Ban đầu, i = left; j = right;

(lượt sắp đầu tiên left = 1, right = 10, và $t = X_5 = 65$ với dãy trên)

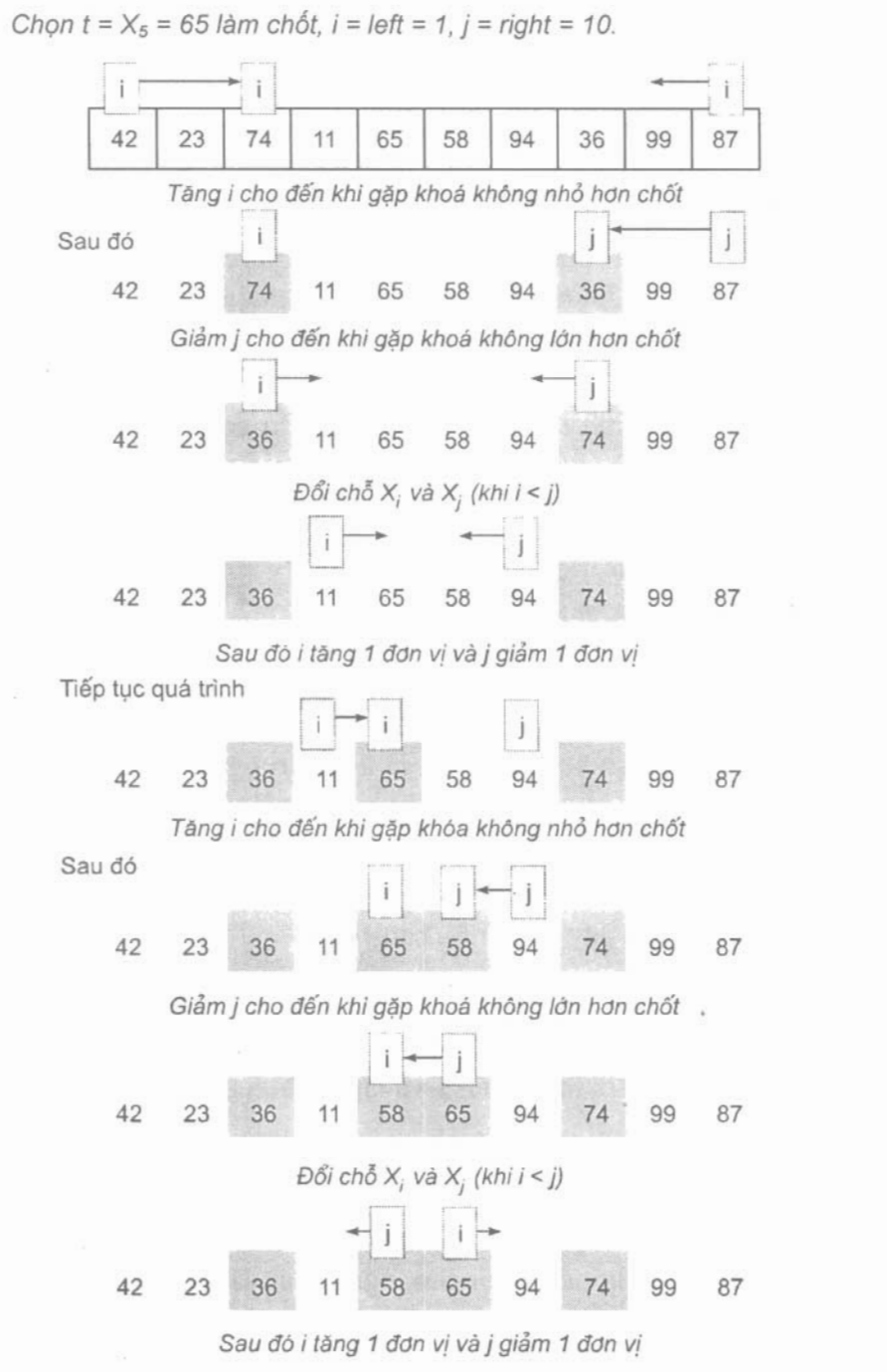
(2) Nếu $X_i < t$ thì tăng i lên 1 và lặp lại quá trình đó

Nếu $X_j > t$ thì j được giảm đi 1 và lặp lại quá trình đó

(3) Đổi chỗ X_i và X_j cho nhau (nếu $i < j$)

Quay lại bước (2) cho đến khi $i > j$ thì dãy X được chia thành 3 dãy nếu vị trí chốt không bị đổi chỗ, ngược lại dãy X được chia thành 2 dãy và kết thúc một lượt sắp xếp. Từ X_l đến X_j là dãy gồm các phần tử nhỏ hơn chốt, và từ X_i đến X_r là các phần tử lớn hơn chốt.

Diễn biến của lượt đầu được mô tả như sau:



Hình 3.1: Mô tả phương pháp sắp xếp phân đoạn

Đến đây kết thúc sắp xếp lượt một vì j < i.

Như vậy, sau một lượt duyệt, dãy khoá được chia thành hai phân đoạn, phân đoạn 1 từ X_{left} đến X_j gồm các phần tử nhỏ hơn chốt và phân đoạn 2 từ X_i đến X_{right} gồm các phần tử lớn hơn chốt.

Tiếp tục sắp xếp phân đoạn 1 và phân đoạn 2 cũng theo kỹ thuật như ở lượt đầu. Quá trình phân đoạn kết thúc khi mỗi phân đoạn chỉ còn một phần tử. Đến đây ta hoàn thành việc sắp xếp dãy.

Các phân đoạn được sắp với cùng một kỹ thuật, vì vậy khi cài đặt giải thuật sắp xếp người ta sử dụng kỹ thuật đệ quy.

1.3.3. Giải thuật

Dưới đây là giải thuật sắp xếp bằng phương pháp phân đoạn được viết tựa ngôn ngữ C, đây là một giải thuật được thiết kế theo kỹ thuật đệ quy.

```
void Quick_Sort (int X[ ], int left, int right)
/*Sắp xếp dãy khoá X với phân đoạn từ Xleft đến Xright */
{
  if ( left < right )
  //Việc phân đoạn chỉ thực hiện với phân đoạn có 2 phần tử trở lên
  {
        int i = left;
        int j = right;
        int k = (left+right)/2;
        int t = X[k];
        while (i <= j)
        {
              while (X[i] < t) i = i + 1;
              while (X[j] > t) j = j - 1;
              if (i <= j)
              {
                    int tg = X[i]; X[i] = X[j]; X[j] = tg;
```

```
                    i++; j--;
                }
            }
            Quick_Sort (X, left, j);
            Quick_Sort (X, i, right);
        }
    }
```

1.3.4. Đánh giá phương pháp

Trước hết ta hãy để ý đến một vài chi tiết có ảnh hưởng tới hiệu lực của phương pháp, đồng thời cũng thể hiện rõ đặc điểm của phương pháp này.

a. Vấn đề chọn "chốt":

Trong phần minh họa trên ta chọn chốt là phần tử trung tâm của phân đoạn cần sắp. Tuy nhiên, bạn có thể chọn phần tử bất kỳ trong phân đoạn để làm chốt. Nhưng rõ ràng khi thể hiện giải thuật ta phải định ra một cách thức chọn chốt cụ thể.

Vấn đề là chốt mà ta chọn được có tốt không? Nếu chốt ta chọn rơi vào đúng khoá nhỏ nhất (hoặc lớn nhất) của phân đoạn cần xử lý thì sau mỗi lượt ta chỉ tách ra được một phân đoạn con có kích thước nhỏ hơn trước là 1 phần tử (vì đã bớt đi một phần tử là "chốt") và phân đoạn này tiếp tục được xử lý. Như vậy ta đã quay trở lại phương pháp sắp xếp kiểu nổi bọt đơn giản. Việc chọn chốt như thế này đã dẫn đến tình huống xấu nhất của phương pháp.

Nếu gọi "*trung vị*" (median) của một dãy khoá là khoá sẽ đứng ở giữa dãy đó sau khi dãy đã được sắp xếp, nghĩa là nó lớn hơn một nửa số khoá của dãy và nhỏ hơn số còn lại, thì tốt nhất vẫn là chọn được đúng trung vị làm "chốt". Lúc đó sau mỗi lượt ta sẽ tách ra được hai phân đoạn con có độ dài gần như nhau và phân đoạn xử lý tiếp theo có kích thước chỉ bằng "nửa" phân đoạn đã chứa nó.

Nhưng làm sao có thể chọn được đúng trung vị? Nếu giả thiết: sự xuất hiện của các khoá trong dãy là đồng khả năng thì trung vị có

thể là bất kỳ một khoá nào trong dãy. Trong giải thuật trên, ta chọn khoá đứng đầu làm chốt là dựa trên cơ sở này. Nhưng với cách chọn này, nếu dãy khoá có khuynh hướng đã theo thứ tự sắp xếp thì khả năng xấu nhất lại xuất hiện. Tuy nhiên nếu khuynh hướng này hay xuất hiện thì việc chọn khoá đang được đứng ở giữa dãy lại gặp thuận lợi.

Để dung hoà với cách chọn như trên, đồng thời cũng để kết hợp với một đề nghị sau này của Hoare là: "Chọn trung vị của một dãy khoá nhỏ hơn, thuộc dãy khoá cho, làm chốt", R.C.Singleton đã đưa ra một cách chọn là:

Chọn X[q] là "chốt" với X[q] là trung vị của ba khoá X[1eft], X[(left+right)/2] và X[right] trong đó left và right là chỉ số của khoá đầu và khoá cuối của phân đoạn. Các kiểu chọn khoá chốt còn được nhiều tác giả khác nữa đưa ra và cũng có nhiều kết quả đáng chú ý.

b. Vấn đề phối hợp với cách sắp xếp khác

Khi kích thước của các phân đoạn đã khá nhỏ, việc tiếp tục phân đoạn nữa theo phương pháp QUICK_SORT thực ra sẽ không có lợi. Lúc đó sử dụng một số phương pháp sắp xếp đơn giản lại tiện hơn. Vì vậy, sắp xếp NHANH thường không tiến hành triệt để mà dừng lại ở lúc cần thiết để gọi tới một phương pháp sắp xếp đơn giản, giao cho nó tiếp tục thực hiện sắp xếp với các phân đoạn nhỏ còn lại. Kunth (1974) có nêu: 9 có thể coi là kích thước giới hạn của phân đoạn để sau đó QUICK_SORT gọi tới phương pháp sắp xếp đơn giản.

Ngoài ra việc chọn phân đoạn nào để xử lý tiếp theo cũng là một vấn đề cần được xem xét tới.

Bây giờ ta sẽ đánh giá giải thuật QUICK_SORT.

Gọi T(n) là thời gian thực hiện giải thuật ứng với một bảng n khoá, P(n) là thời gian để phân đoạn một bảng n khoá thành hai bảng con. Ta có thể viết:

$$T(n) = P(n) + T(j\text{-left}) + T(right\text{-}j)$$

Chú ý rằng $P(n) = Cn$ với C là hằng số.

Trường hợp xấu nhất xảy ra khi bảng khoá vốn đã có thứ tự sắp xếp: sau khi phân đoạn một trong hai bảng con là rỗng (j = left hoặc j = right).

Giả sử j = left, ta có:

$$\begin{aligned} T_x(n) &= P(n) + T_x(0) + T_x(n-1) \\ &= Cn + T_x(n-1) \\ &= Cn + C(n-1) + T_x(n-2) \\ &\cdots \\ &= \sum_{k=1}^{n} C_k + T_x(0) \\ &= C.\frac{n(n+1)}{2} = O(n^2) \end{aligned}$$

Trường hợp này QUICK_SORT không hơn gì các phương pháp đã nêu trước đây.

Trường hợp tốt nhất xảy ra khi bảng luôn luôn được chia đôi, nghĩa là j = (left+right)/2. Lúc đó:

$$\begin{aligned} T_1(n) &= P(n)b + 2T_1(n/2) \\ &= Cn + 2T_1(n/2) \\ &= Cn + 2C(n/2) + 4T_1(n/4) = 2Cn_n + 2^2T_1(n/4) \\ &= Cn + 2C(n/2) + 4C(n/4) + 8T_1(n/8) = 3C_n + 2^3T_1(n/8) \\ &\cdots \\ &= (\log_2 n)\, Cn + 2^{long}2^mT_1(1) \\ &= O(n\log_2 n) \end{aligned}$$

Việc xác định giá trị trung bình $T_{tb}(n)$ không còn đơn giản như hai trường hợp trên, nên ta sẽ không xét chi tiết. Kết quả mà ta cần ghi nhận là: người ta đã chứng minh được:

$$T_{tb}(n) = O(n\log_2 n)$$

Như vậy rõ ràng là khi n khá lớn, QUICK_SORT đã tỏ ra có hiệu lực hơn hẳn ba phương pháp đã nêu. Tuy nhiên, việc sử dụng kỹ thuật đệ qui sẽ tiêu tốn rất nhiều bộ nhớ khi thực hiện giải thuật trên máy tính.

Sau đây ta xem xét một phương pháp sắp xếp với thời gian cũng rất tốt đó là phương pháp vun đống.

1.4. Sắp xếp vun đống

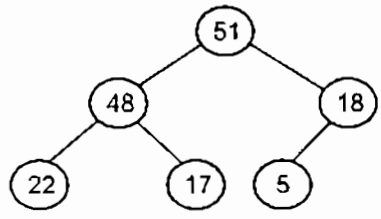
1.4.1. Giới thiệu phương pháp

Trước tiên, ta xem xét khái niệm đống (HEAP).

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khoá sao cho khoá ở nút "cha" bao giờ cũng lớn hơn khóa ở các nút "con".

Đống được lưu trữ kế tiếp trên máy (bởi mảng một chiều).

Như vậy đối với đống, nếu nút cha ở vị trí thứ i thì 2 nút con (nếu có) sẽ ở vị trí thứ 2*i và 2*i+1.



Hình 3.2: Đống

Ví dụ: đống là cây T có dạng như hình 2.2.

1.4.2. Nguyên tắc sắp xếp

Sắp xếp kiểu vun đống (HEAP_SORT) được chia thành hai giai đoạn:

- Giai đoạn tạo đống: Cây nhị phân biểu diễn dãy khoá ban đầu được biến đổi để tạo thành đống.

- Giai đoạn sắp xếp: ở giai đoạn này, ta tiến hành nhiều lượt hai phép xử lý sau:

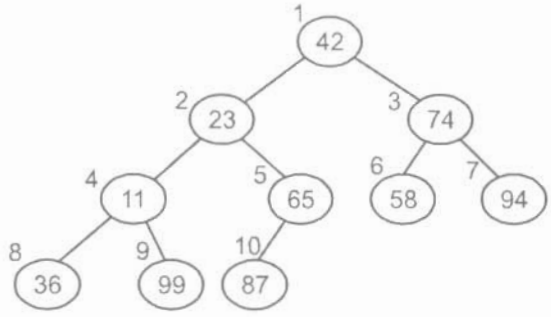
  + Đưa khóa trội về đúng vị trí, bằng cách thực hiện hoán vị các khoá.

  + Vun lại cây gồm các khoá còn lại (sau khi đã loại khoá trội) thành đống.

Ví dụ mô tả: Giả sử với dãy khoá X

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	65	58	94	36	99	87

Cây nhị phân biểu diễn dãy khoá ban đầu như hình 3.3(a).

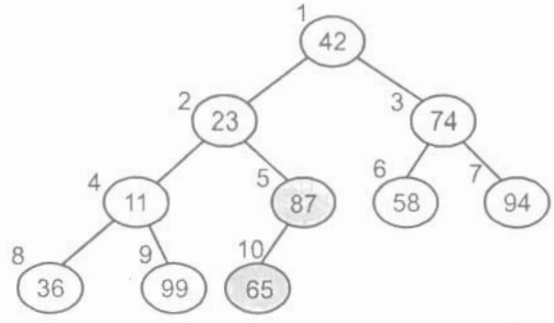


Hình 3.3(a): Cây nhị phân biểu diễn dãy khóa

Lưu ý, cây nhị phân hình 3.3(a) chỉ là mô hình giúp ta hình dung được quá trình sắp xếp theo phương pháp vun đống. Còn quá trình sắp xếp thực sự vẫn là việc hoán đổi vị trí trực tiếp các phần tử mảng.

Với cây này để nó trở thành đống ta làm như sau:

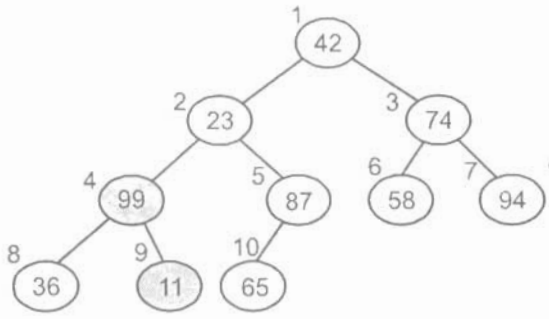
+ Cây con có nút gốc là 65 (vị trí thứ 5 trở thành đống, hình 3.3(b)).



Hình 3.3(b): Xử lý nút khóa X_5

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	87	58	94	36	99	65

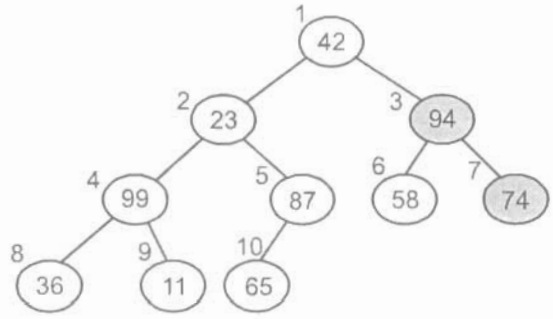
+ Cây con có nút gốc 11 (nút thứ 4) trở thành đống (hình 3.3(c)).



Hình 3.3(c): Xử lý nút khóa X_4

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	99	87	58	94	36	11	65

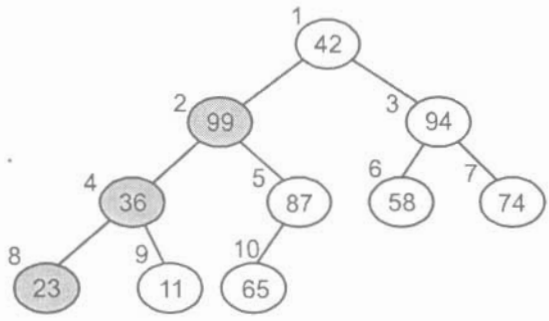
+ Cây con có nút gốc 74 (nút thứ 3) trở thành đống (hình 3.3(d)).



Hình 3.3(d): Xử lý nút khóa X_3

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	94	99	87	58	74	36	11	65

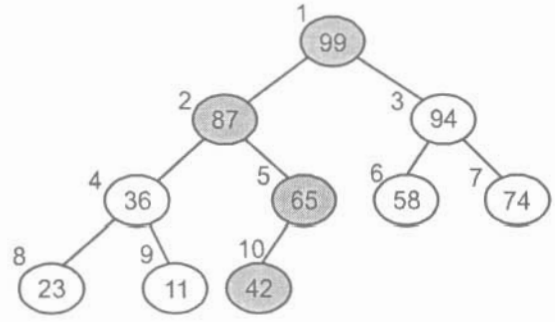
+ Cây con có nút gốc 23 trở thành đống (nút thứ 2) - hình 3.3(e).



Hình 3.3(e): Xử lý nút khóa X_2

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	99	94	36	87	58	74	23	11	65

+ Cuối cùng, cây con có nút gốc 42 trở thành đống (nút thứ 1) - hình 3.3(f).

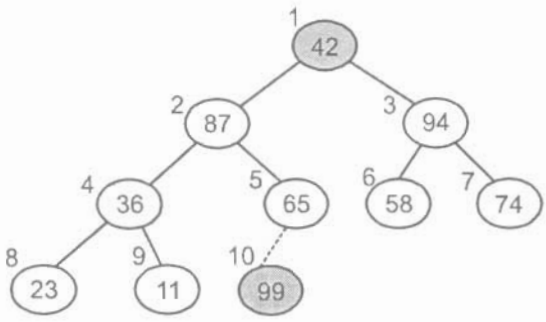


Hình 3.3(f): Xử lý nút khóa X_1 - Đống đầu tiên

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
99	87	94	36	65	58	74	23	11	42

Đến đây quá trình tạo đống đầu tiên kết thúc, ta thấy khóa trội được chuyển về về vị trí đầu tiên của mảng (nút gốc của cây-đống).

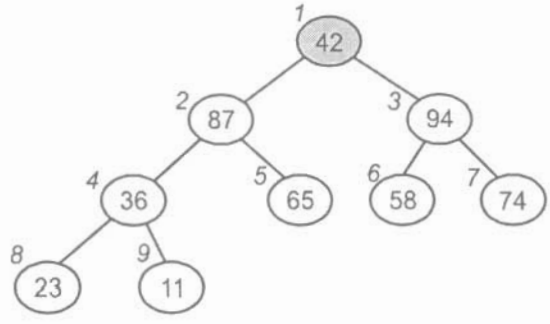
Sang giai đoạn 2, khoá trội 99 được chuyển đến vị trí cuối cùng bằng cách hoán vị cho khoá 42, sau khi hoán vị, nút ứng với khóa trội 99 coi như được loại khỏi cây - hình 3.4.



Hình 3.4: Hoán đổi nút đầu với nút cuối

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	87	94	36	65	58	74	23	11	99

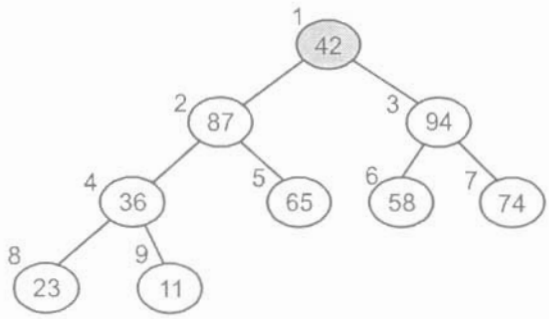
Cây còn lại được vun lại thành đống. Nó có dạng như hình 3.4(a).



Hình 3.4(a): Cây sau khi loại nút cuối

Nhận thấy rằng, cây này chưa phải là một đống, nhưng tất cả các cây con của cây này đều là đống. Vì vậy, ở bước này và tất cả các bước còn lại ta chỉ cần xét nút gốc (nút thứ 1) của cây.

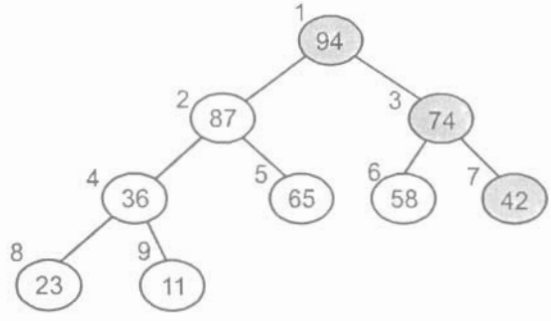
Ta có đống thứ 2 như hình 3.4(b).



Hình 3.4(b): Đống thứ 2

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
94	87	74	36	65	58	42	23	11	**99**

Lại đổi vị trí nút đầu tiên và nút cuối cùng - hình 3.4(c).



Hình 3.4(c): Đổi vị trí nút đầu và nút cuối

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
11	87	74	36	65	58	42	23	**94**	**99**

Sau đó một lượt xử lý tiếp theo được thực hiện tương tự và cứ như thế cho đến khi toàn bộ dãy khóa được sắp xếp. Bạn đọc có thể tự mình mô tả quá trình sắp xếp trong các bước tiếp theo và mô tả việc sắp xếp dãy khóa theo thứ tự giảm dần.

1.4.3. Giải thuật

Nhận thấy, có thể coi một nút lá là một cây con đã thoả mãn tính chất của đống, có nghĩa một nút lá được coi là một đống. Như vậy,

bước tạo đống hay vun đống được quy về một phép xử lý chung là: chuyển một cây thành đống mà cây con trái và cây con phải của gốc đã là đống rồi. Ta xây dựng riêng một thủ tục vun đống là thủ tục Hoan_vi.

Mặt khác, đối với cây nhị phân hoàn chỉnh có n nút, chỉ các nút ứng với các vị trí từ 1 đến (n/2) mới có thể là nút cha của các nút khác. Nên khi tạo đống thủ tục Hoan_vi chỉ cần áp dụng với các cây con có gốc ở vị trí (n/2), (n/2) - 1,..., 1. Còn khi vun đống thì luôn áp dụng với cây có gốc ở vị trí 1.

Giải thuật sắp xếp dãy n khoá X_1, X_2,..., X_n được biểu diễn bởi 3 thủ tục tựa ngôn ngữ lập trình C sau đây.

```
void Hoan_vi(X[ ],k,r)
{
  if (X[k] khác lá và có giá trị nhỏ hơn 2 con)
  {       + Chọn con có giá trị lớn hơn, giả sử là X[j];
          + Đổi chỗ X[k], và X[j] ;
          + call Hoan_vi(X,j,r);
  }
}
```

Thủ tục này thực hiện biến đổi cây có gốc là X[k] thành đống, với r là vị trí của khoá cuối cùng trong dãy được xét. Thủ tục này cũng có thể viết dưới dạng một giải thuật lặp để cải thiện thời gian sắp xếp. Bạn đọc dựa vào minh hoạ trên để "làm mịn" thủ tục này.

```
void Tao_dong_dau_tien(X[], n)
{
  for (i = n/2; i >= 1; i--)
          Hoan_vi(X, i, n);
}
```

Thủ tục này thực hiện biến đổi dãy ban đầu (n khoá) thành dãy biểu diễn đống đầu tiên. Quá trình biến đổi đống đầu tiên được thực

hiện từ dưới lên, các đống tiếp theo sẽ được thực hiện từ trên xuống (bạn đọc có thể xem lại phần minh họa).

```
void HEAP_SORT(X[ ], n)
{
Tao_dong_dau_tien(X,n);
  for (i = n; i >= 2; i--)
  {
        Đổi chỗ (X[1], X[i]);
        Hoan_vi(X, 1, i-1);
  }
}
```

Thủ tục này thực hiện việc sắp dãy khoá theo chiều tăng dần, sử dụng hai thủ tục bên trên. Tuy nhiên, thủ tục Hoan_vi được viết dưới dạng đệ qui, vì thế làm chậm quá trình sắp xếp và tốn bộ nhớ. Trường hợp này ta nên khử đệ quy, nghĩa là sử dụng giải thuật lặp, việc này khá đơn giản, bạn đọc tự nghiên cứu, xem như một bài tập.

1.4.4. Phân tích đánh giá

Đối với HEAP_SORT ta chú ý tới việc đánh giá trong trường hợp xấu nhất. Như ta đã biết: một cây nhị phân hoàn chỉnh có n nút thì chiều cao của cây đó là $[\log_2(n+1)]$. Khi tạo đống cũng như khi vun lại đống trong giai đoạn sắp xếp, trường hợp xấu nhất thì số lượng phép so sánh cũng chỉ tỷ lệ với chiều cao của cây. Do đó có thể suy ra, trong trường hợp xấu nhất, cấp độ lớn của thời gian thực hiện HEAP_SORT chỉ là $O(n\log_2 n)$. Việc đánh giá thời gian thực hiện trung bình phức tạp hơn, ta không xét tới mà chỉ ghi nhận một kết quả đã được chứng minh là: cấp độ lớn của thời gian thực hiện trung bình giải thuật HEAP_SORT cũng là $O(n\log_2 n)$.

Có thể nhận xét thêm là: QUICK_SORT còn phải dùng thêm không gian nhớ cho ngăm xếp (stack), để bảo lưu thông tin về các phân đoạn sẽ được xử lý tiếp theo (vì thực hiện đệ qui). Còn

HEAP_SORT thì ngoài một nút nhớ phụ, để thực hiện đổi chỗ, nó không cần thêm gì nữa.

1.5. Sắp xếp kiểu trộn

Bây giờ ta xét tới một phương pháp sắp xếp mới, khá đặc biệt ở chỗ nó dựa trên một phép toán rất đơn giản là phép trộn.

1.5.1. Phép trộn hai đường

Một tư tưởng hết sức đơn giản là trộn hai dãy đã được sắp xếp thành một dãy được sắp xếp.

Giả sử cho hai dãy được sắp xếp:

X: 12 25 28

và Y: 3 9 15 32 39

Khi đó ta sẽ trộn hai dãy X và Y thành dãy Z cũng được sắp xếp tăng như sau:

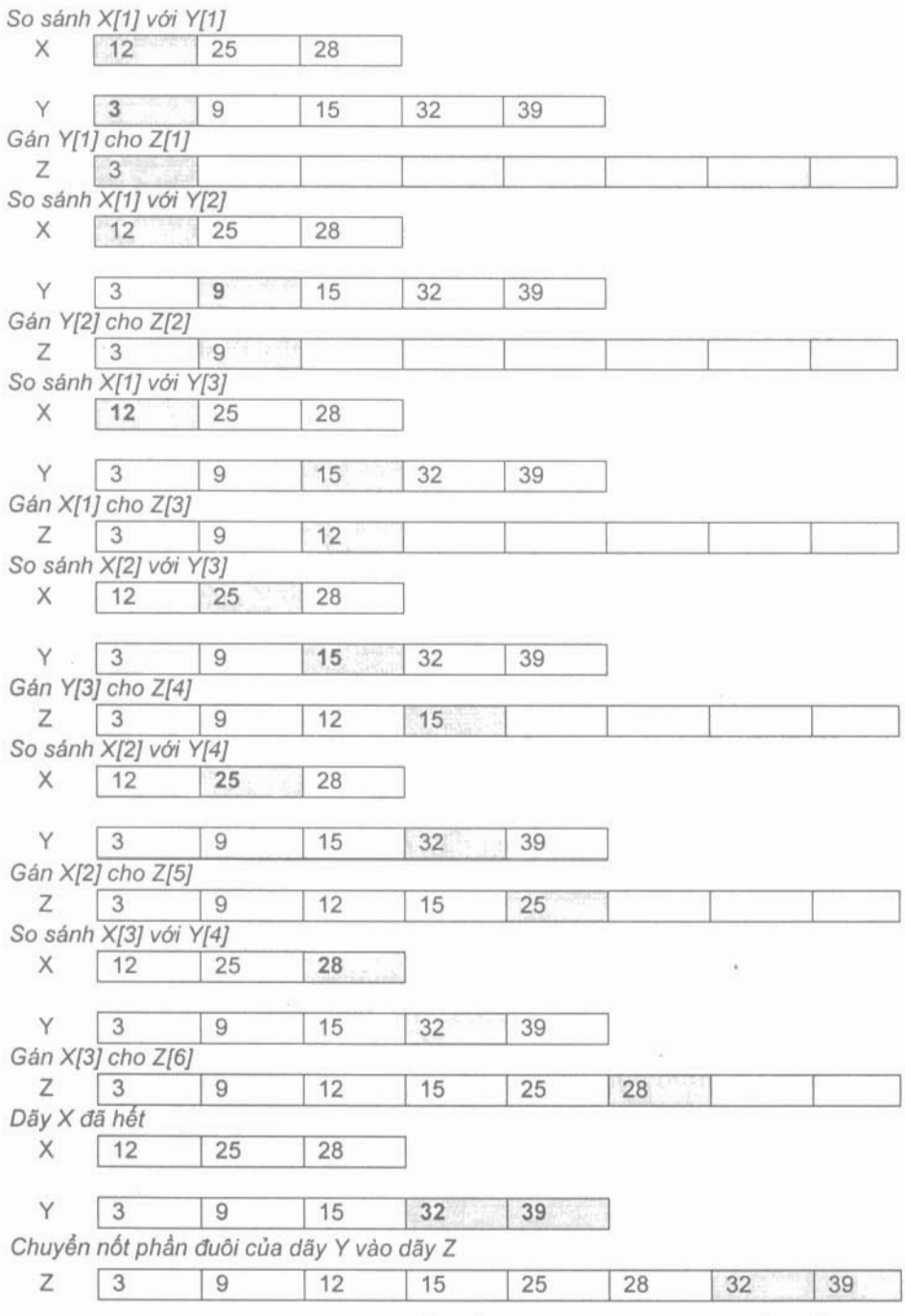
Z: 3 9 12 15 25 28 32 39

Có thể mô tả cách trộn đơn giản như sau:

+ So sánh hai khóa nhỏ nhất của hai dãy X và Y, chọn khóa nhỏ hơn đặt vào vị trí thích hợp trong dãy Z, rồi loại khóa được chọn khỏi dãy chứa nó.

+ Quá trình như vậy cứ tiếp tục cho đến khi một trong hai dãy X hoặc Y đã hết, khi đó chuyển nốt phần đuôi của dãy còn lại vào dãy Z là xong.

Hình 3.5 minh họa cách làm trên.



Hình 3.5: Trộn dãy X và dãy Y sắp xếp tăng, thành dãy Z sắp xếp tăng

Bây giờ ta xét tới giải thuật MERGING, các dãy X, Y, Z được lưu trữ bởi các mảng, với m, n lần lượt là độ dài của X và Y. Việc so sánh X[i] và Y[j] sẽ xác định phần tử của dãy nào được gán cho Z[k].

```
void MERGING(X[ ], m, Y[ ], n, Z)
{
  //1. Khởi tạo các chỉ số
        i = 1; j = 1; k = 1;
  //2. Chuyển các phần phần tử từ dãy X, Y vào dãy Z
        while (i <= m && j <= n)
        {
              if (X[i] < Y[j])
              {
                    Z[k] = X[i]; i++; k++;
              }
              else
              {
                    Z[k] = Y[j]; j++; k++;
              }
        }
  //3. Một trong hai mạch đã hết, chuyển đuôi của dãy còn lại vào Z
        while (i <= m)
        {
              Z[k] = X[i]; i++; k++;
        }
        while (j <= n)
        {
              Z[k] = Y[j]; j++; k++;
        }
}
```

1.5.2. Sắp xếp kiểu trộn hai đường trực tiếp

a. Giới thiệu phương pháp

Với ý tưởng trộn hai dãy đã sắp xếp thành một dãy đã sắp xếp người ta đã phát triển thành một phương pháp sắp xếp mới mà ta quen gọi là phương pháp trộn.

Giả sử cần sắp xếp dãy khóa X với độ dài n: X_1, X_2, ..., X_n. Có thể thấy rằng mỗi khóa X_i trong dãy có thể xem như một dãy con được sắp với độ dài là 1 (sau đây ta gọi là một vệt).

Nếu trộn hai vệt như vậy ta sẽ được một vệt mới với độ dài là 2. Lại trộn hai vệt độ dài 2 ta được một vệt độ dài 4, tiếp tục trộn như thế, cuối cùng ta sẽ được một vệt có chiều dài n, như thế dãy ban đầu được sắp xếp hoàn toàn.

**** Ta mô tả phương pháp vừa nêu trong ví dụ sau:***

Cho dãy khóa: 42 23 74 11 65 58 94 36 99 87

Coi dãy khóa gồm 10 mạch có độ dài 1

[42] [23] [74] [11] [65] [58] [94] [36] [99] [87]

Trộn các cặp vệt độ dài 1 kề nhau ta được:

[23 42] [11 74] [58 65] [36 94] [87 99]

Trộn các cặp vệt độ dài 2 kề nhau ta được:

[11 23 42 74] [36 58 65 94] [87 99]

Ta giữ nguyên vệt lẻ cặp: [87 99]

Trộn các cặp vệt độ dài 4 kề nhau ta được:

[11 23 36 42 58 65 74 94] [87 99]

Ta giữ nguyên vệt lẻ cặp: [87 99]

Cuối cùng, trộn hai vệt ta có dãy được sắp xếp:

[11 23 36 42 58 65 74 87 94 99]

Qua ví dụ trên bạn đọc đã hiểu được phương pháp sắp trộn, sau đây ta đi thiết kế giải thuật cho phương pháp này.

b. Giải thuật

Giải thuật gồm ba công đoạn:

+ Thủ tục trộn hai vệt thành một vệt

+ Thủ tục trộn một lượt các cặp vệt độ dài *l* (có thể một vệt có chiều dài nhỏ hơn *l*).

+ Thủ tục sắp xếp trộn

Ta cải tiến thủ tục MERGING thành thủ tục trộn hai vệt kề nhau trên dãy khóa X, vệt được trộn nằm trên dãy Z.

```
void MERGE (X[ ], bt1, w1, bt2, w2, Z[ ])
{
//bt1, bt2: là vị trí biên trái của hai vệt, w1, w2 là độ dài của hai vệt
      i = bt1; j = bt2; bp1 = bt1+w1-1; bp2 = bt2+w2-1; k = bt1;
//bp1, bp2 là biên phải của hai vệt, k là biên trái của vệt mới trên dãy Z
  while (i <= bp1 && j <= bp2)
  {
        if (X[i] < X[j])
        {
              Z[k] = X[i]; i++; k++;
        }
        else
        {
              Z[k] = X[j]; j++; k++;
        }
  }
  while (i <= bp1)
  {
        Z[k] = X[i]; i++; k++;
  }
  while (j <= bp2)
  {
        Z[k] = X[j]; j++; k++;
  }
}
```

**** Thủ tục trộn một lần các cặp vệt***

Với dãy khóa X độ dài n: X_1, X_2,..., X_n, các dãy con trong X là các vệt có độ dài *l*, trừ dãy con cuối cùng có thể có độ dài nhỏ hơn *l*.

Nếu *cv* là số cặp vệt có độ dài *l* thì *cv* = *n*/(2**l*)

Đặt s = 2**l***cv* thì s là số các phần tử thuộc các cặp vệt, và *r* = *n*-*s* là số các phần tử còn lại.

```
void MERGE_PASS (X[ ], n, l, Z[ ])
{
  //Z là dãy có độ dài n, chứa các phần tử của X sau khi trộn
  //1. Khởi tạo các giá trị ban đầu
        cv = n/(2*l);
        s = 2*l*cv;
        r = n – s;
  //2. Trộn từng cặp mạch
        for (j = 1; j <= cv; j++)
        {
              b1 = l + (2*j -2)*l ; //biên trái của mạch thứ nhất
              MERGE(X, b1, l, b1+l, Z);
        }
  //3. Chỉ còn một vệt
        if (r<=l)
        for (j=1; j<=r; j++)
              Z[s+j] = X[s+j];
//4. Còn hai vệt nhưng một vệt có độ dài nhỏ hơn l
        else MERGE(X, s+1, l, s+l+1, r-l, Z);
}
```

Thủ tục MERGE_PASS được gọi trong thủ tục thực hiện sắp xếp kiểu trộn hai đường trực tiếp sau đây:

```
void MERGE_SORT (X[ ], n)
{
  //1. Khởi tạo số phần từ trong một vệt
        l =1;
  //2. Sắp xếp trộn
```

```
        while (l<n)
            {
            //trộn và chuyển các phần tử vào dãy Z
                MERGE_PASS(X, n, l, Z);
            //trộn và chuyển các phần tử trở lại dãy X
                MERGE_PASS(X, n, 2*l, Z);
                l = l*4;
            }
}
```

Sau khi sắp xếp xong các phần tử của dãy khóa X vẫn ở chỗ cũ.

c. Phân tích đánh giá

Trong phương pháp này ta thấy, số lượng phép toán chuyển chỗ thường nhiều hơn số lượng phép toán so sánh. Chẳng hạn, với thủ tục *MERGE*, trong câu lệnh *while*, ứng với một phép so sánh có một phép đổi chỗ. Nhưng nếu một vệt nào đó hết trước, thì phần đuôi của vệt còn lại được chuyển chỗ mà không tương ứng với một phép so sánh nào. Vì vậy, với phương pháp này ta chọn phép toán tích cực là phép toán chuyển chỗ để đánh giá thời gian thực hiện của giải thuật.

Nhận thấy rằng, ở bất kỳ lượt trộn nào (MERGE_PASS) thì toàn bộ các khóa cũng được chuyển sang dãy mới (từ X sang Z, hoặc từ Z sang X). Như vậy chi phí thời gian cho một lượt trộn là O(n). Ngoài ra số lượt gọi thủ tục MERGE_PASS trong thủ tục MERGE_SORT là $[\log_2 n]$, vì ở lượt 1 kích thước của vệt là $l=1=2^0$. Ở lượt i kích thước của vệt sẽ là 2^{i-1}, mà sau lượt cuối cùng thì vệt đã có kích thước là n.

Vậy sắp xếp kiểu trộn hai đường trực tiếp có thời gian thực hiện là $O(n\log_2 n)$. Tuy nhiên, chi phí về không gian nhớ khá lớn, nó đòi hỏi 2n phần tử nhớ, gấp đôi so với các phương pháp khác. Do đó người ta thường sử dụng phương pháp này khi sắp xếp ngoài, sắp xếp dữ liệu trong tệp tin.

Kết luận chung:

Ta nhận thấy, với cùng một mục đích sắp xếp mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Cấu trúc dữ liệu được

lựa chọn để mô tả đối tượng sắp xếp đã ảnh hưởng rất lớn tới việc lựa chọn giải thuật sắp xếp thích hợp. Các giải thuật sắp xếp đơn giản thể hiện ba kỹ thuật cơ sở của sắp xếp (dựa vào phép so sánh các giá trị khoá). Tuy nhiên, cấp độ lớn của thời gian thực hiện chúng là (n^2), do đó chỉ nên sử dụng chúng khi n nhỏ. Các giải thuật cải tiến như QUICK_SORT, HEAP_SORT đã đạt được thời gian thực hiện tốt hơn là $O(n\log_2 n)$, thường được sử dụng khi n lớn. Nếu dãy khoá cần sắp xếp vốn có khuynh hướng hầu như đã được sắp xếp sẵn rồi thì QUICK_SORT lại không nên dùng. Nhưng nếu ban đầu dãy có khuynh hướng ít nhiều có thứ tự ngược với thứ tự sắp xếp thì HEAP_SORT lại tỏ ra thuận lợi.

Việc khẳng định phương pháp sắp xếp này tốt hơn các phương pháp khác chỉ là tương đối, vì thế việc chọn một phương pháp sắp xếp thích hợp thường tuỳ thuộc vào từng yêu cầu, từng điều kiện cụ thể của bài toán.

2. TÌM KIẾM

2.1. Bài toán tìm kiếm

Tìm kiếm là một đòi hỏi thường xuyên trong việc xử lý các bài toán tin học, nhất là đối với các bài toán quản lý. Tuy nhiên, ở đây ta chỉ xét các giải thuật tìm kiếm một cách tổng quát, không liên quan đến mục đích xử lý cụ thể nào.

Ta mô tả bài toán tìm kiếm như sau:

Cho một bảng gồm n bản ghi R_1, R_2,..., R_n. Mỗi bản ghi R_i có một khoá tìm kiếm là R_i.key ($1 \le i \le n$). Hãy tìm bản ghi có giá trị khoá tương ứng là KH cho trước.

KH được gọi là khoá tìm kiếm.

Việc tìm kiếm sẽ hoàn thành khi có một trong hai tình huống sau đây xảy ra:

1. Tìm được bản ghi có giá trị khoá tương ứng bằng KH, lúc đó việc tìm kiếm là thành công.

2. Không tìm được bản ghi nào có khoá tương ứng bằng KH, lúc đó việc tìm kiếm là không thành công.

Cũng giống như việc sắp xếp, khoá của mỗi bản ghi chính là đặc điểm nhận biết của bản ghi đó trong tìm kiếm. Để đơn giản, trong các giải thuật ta coi nó là đại diện cho bản ghi đó và trong các ví dụ ta cũng chỉ nói tới khoá. Ta coi các khoá R_i.key ($1 \le i \le n$) là các số khác nhau. Trong chương này ta cũng chỉ xét các phương pháp tìm kiếm cơ bản, phổ dụng, đối với dữ liệu được lưu trữ ở bộ nhớ trong và được gọi là tìm kiếm trong. Có hai phương pháp tìm kiếm trong là tìm kiếm tuần tự và tìm kiếm nhị phân mà ta sẽ xét trong các phần sau đây.

2.2. Tìm kiếm tuần tự

2.2.1. Nguyên tắc tìm kiếm

Tìm kiếm tuần tự là phương pháp tìm kiếm rất đơn giản. Nguyên tắc tìm kiếm theo phương pháp này có thể tóm tắt như sau:

Bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khoá tìm kiếm KH với các khoá tương ứng của các bản ghi trong bảng, cho đến khi tìm được một bản ghi mong muốn trong bảng (trường hợp tìm kiếm thành công, trả về vị trí của bản ghi tìm được) hoặc đã hết bảng mà không thấy (trường hợp tìm kiếm không thành công).

Ví dụ minh hoạ

Giả sử các khoá tương ứng của các bản ghi trong bảng có 6 bản ghi là dãy số:

38 12 -56 95 23 11 và khoá cần tìm là KH = 95, khi đó việc tìm kiếm được thực hiện như mô tả dưới đây:

Với i = 1 < n	R_1.key = 38 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 2 < n	R_2.key = 12 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 3 < n	R_3.key = -56 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 4 < n	R_4.key = 95 = KH, kết thúc tìm kiếm và trả về vị trí tìm được là i = 4

Ví dụ trên đây minh hoạ cho một phép tìm kiếm thành công, bạn đọc tự lấy ví dụ minh hoạ cho phép tìm kiếm không thành công.

2.2.2. Giải thuật

Thủ tục dưới đây thể hiện giải thuật tìm kiếm tuần tự, tìm kiếm khoá KH trên một dãy khoá X, có n phần tử. Nếu có nó sẽ đưa ra chỉ số của khoá ấy, nếu không có nó đưa ra giá trị 0.

```
int Sequence_Search(X[ ], n, KH)
{
  //1. Khởi tạo
        i=1;
  //2  Tìm khoá trong dãy
        while (X[i] != KH && i <= n) i = i+1;
  //3. Kiểm tra và trả về kết quả tìm kiếm
        if (i <= n) return (i);
        else return (0);
}
```

Để "tiết kiệm" thời gian ta có thể cải tiến giải thuật trên bằng cách thêm vào một khoá phụ X_{n+1}, có giá trị bằng KH như sau:

```
int Improvement_Sequence_Search(X[ ], n, KH)
{
  //1. Khởi tạo
        i = 1; X[n+1] = KH;
  //2. Tìm khoá trong dãy
        while (X[i] != KH)
              i = i+1;
  //3. Kiểm tra và trả về kết quả tìm kiếm
        if (i == n+1) return (0);
        else return (i);
}
```

Với giải thuật cải tiến, ta giảm được một nửa số phép toán kiểm tra trong vòng lặp *while*, vì thế tiết kiệm được khá nhiều thời gian khi số lượng bản ghi trong dãy X thật sự lớn.

2.2.3. Đánh giá hiệu lực của giải thuật

Để đánh giá hiệu lực của phép tìm kiếm ta có thể dựa vào số lượng các phép so sánh. Ta thấy với giải thuật trên (cải tiến). trong trường hợp tốt nhất, chỉ cần 1 phép so sánh, $C_{min} = 1$, còn trong trường hợp xấu nhất số phép toán so sánh là $C_{max} = n+1$. Nếu hiện tượng khoá tìm kiếm trùng với một khoá nào đó của bảng là đồng khả năng thì $C_{tb} = (n+1)/2$. Tóm lại, cả hai trường hợp xấu nhất cũng như trung bình, thời gian thực hiện tìm kiếm theo giải thuật trên là O(n).

Trong trường hợp dãy khoá đã được sắp xếp theo chiều tăng dần (hoặc giảm dần), thời gian trung bình thực hiện giải thuật sẽ nhỏ hơn. Tuy nhiên, trong trường hợp này ta có thể áp dụng phương pháp tìm kiếm khác, với thời gian nhanh hơn nhiều, đó là phương pháp tìm kiếm nhị phân.

2.3. Phương pháp tìm kiếm nhị phân

2.3.1. Nguyên tắc

Phương pháp tìm kiếm nhị phân là phương pháp tìm kiếm khá thông dụng. Trong phương pháp này ta áp dụng chiến lược "chia để trị" để giải quyết bài toán tìm kiếm. Việc này cũng giống như việc ta tìm một từ trong quyển từ điển. Ta có thể hiểu nôm na thế này: để tìm từ, ta mở từ điển vào trang "giữa", tìm từ trong trang này, nếu có, việc tìm kiếm là thành công, còn nếu không có ta tìm ở "nửa trước" hoặc nửa sau của từ điển. Việc tìm ở "nửa trước" hay "nửa sau" cũng được thực hiện giống như trên (nghĩa là ta cũng mở vào trang giữa...). Việc tìm kiếm dừng lại khi tìm được từ trong một trang "giữa" nào đó (tìm kiếm thành công), hoặc khi từ điển chỉ còn một trang, mà không có từ cần tìm (tìm kiếm không thành công).

Như vậy, trong phương pháp tìm kiếm nhị phân, giả sử với dãy khoá là $X_l, X_{l+1},..., X_r$ nó luôn chọn khoá ở giữa là X_j, với $j = [(l+r)/2]$ để so sánh với khoá tìm kiếm KH. Tìm kiếm sẽ kết thúc nếu $KH = X_j$. Nếu $KH < X_j$ tìm kiếm được thực hiện tiếp với $X_{l+1}, X_{l+2},..., X_{j-1}$, còn $KH > X_j$, tìm kiếm được thực hiện tiếp với $X_{j+1}, X_{j+2},..., X_r$. Với dãy

khoá sau, một kỹ thuật tương tự lại được sử dụng. Quá trình tìm kiếm được tiếp tục khi tìm thấy khoá mong muốn hoặc dãy khoá xét đó là rỗng (không thấy).

Ví dụ minh hoạ

Giả sử các khoá tương ứng của các bản ghi trong bảng có 6 bản ghi là dãy số tăng dần:

-56 11 12 23 38 95 và khoá cần tìm là KH = 95, khi đó việc tìm kiếm được thực hiện như mô tả dưới đây.

X_1 X_2 X_3 X_4 X_5 X_6

[-56 11 **12** 23 38 95]

Bước 1: j = 3, khoá ở giữa là $X_3 = 12$, $KH > X_3$

Tìm kiếm tiếp tục với dãy khoá

[23 **38** 95]

Bước 2: j = 5, khoá ở giữa là $X_5 = 38$, $KH > X_5$

Tìm kiếm tiếp tục với dãy khoá

[95]

Bước 3: j = 6, khoá ở giữa là $X_6 = 95$, $KH = X_5 \Rightarrow$ Tìm kiếm thành công!

Ví dụ minh hoạ tìm kiếm không thành công, bạn đọc xem như là một bài tập.

Sau đây là giải thuật tìm kiếm nhị phân.

2.3.2. Giải thuật

Giải thuật được viết dưới dạng một thủ tục dạng đệ quy, tựa ngôn ngữ C. Nếu tìm thấy, giải thuật trả về vị trí của khoá được tìm thấy (giá trị j). Nếu không tìm thấy, giải thuật trả về giá trị 0.

```
int BINARY_SEARCH(X[ ], l, r, KH)
{
//1. Trường hợp dãy được xét rỗng, tìm kiếm không thành công
  if (l > r)
        return 0;
```

```
//2. Tìm kiếm trong dãy được xét
  else {
        j = (l+r) % 2;
  //2.1. Tìm kiếm thành công
        if (KH == X[j])  return j;
  //2.2. Tìm ở nửa dãy trái
        else
              if (KH < X[j]) return BINARY_SEARCH(X,l,j-1,KH);
  //2.3. Tìm ở nửa dãy phải
              else    return BINARY_SEARCH(X,j+1,r,KH);
  }
}
```

Trong giải thuật trên *l*, r tương ứng là chỉ số của khoá đầu tiên và chỉ số của khoá cuối cùng của dãy đang xét.

Giải thuật tìm kiếm nhị phân cũng có thể được viết dưới dạng lặp (khử đệ quy), việc này bạn đọc có thể tự làm.

2.3.3. Đánh giá giải thuật

Trong giải thuật trên ta thấy số lượng phép so sánh phụ thuộc vào giá trị khoá KH. Trường hợp thuận lợi nhất đối với dãy khoá X_1, X_2,..., X_n, mà lời gọi sẽ là BINARY_SEARCH(X,1,n,KH) là KH = X[(n+1)/2], nghĩa là chỉ cần một phép so sánh, lúc đó $T_1 = O(1)$. Trường hợp xấu nhất có phức tạp hơn. Ta gọi w(r-*l*+1) là hàm biểu thị số lượng phép so sánh trong trường hợp xấu nhất ứng với một lời gọi BINARY_SEARCH(X,*l*,r,KH) và đặt n = r-*l*+1 (ứng với dãy khoá mà *l* = 1, r = n) thì trong trường hợp xấu nhất ta sẽ phải gọi đệ quy, vậy ta có:

$$w(n) = 1+w(n \text{ div } 2)$$

Với phương pháp truy hồi có thể viết:

$$w(n) = 1+1+w(n \text{ div } 2^2) \Rightarrow w(n) = 1+1+1+w(n \text{ div } 2^3)$$

Như vậy w(n) có dạng $w(n) = k+w(n \text{ div } 2^k)$

Khi $(n \text{ div } 2^k) = 1$ ta có $w(n \text{ div } 2^k) = w(1)$ và khi đó tìm kiếm phải kết thúc. Song $(n \text{ div } 2^k) = 1$ thì suy ra $2^k \le n \le 2^{k+1}$, do đó

$k \le \log_2 n < k+1$, nghĩa là có thể viết $k = \log_2 n$. Vì vậy, cuối cùng ta có $w(n) = \log_2 n + 1$ hay $T_X(n) = O(\log_2 n)$.

Người ta cũng chứng minh được $T_{tb}(n) = O(\log_2 n)$.

Rõ ràng so với tìm kiếm tuần tự (có thời gian trung bình là O(n)), chi phí tìm kiếm nhị phân ít hơn nhiều. Hiện tại không có phương pháp tìm kiếm nào, dựa trên việc so sánh giá trị khoá lại có thể đạt được kết quả tốt hơn.

Tuy nhiên, nên nhớ rằng tìm kiếm nhị phân chỉ thực hiện trên dãy khoá đã được sắp xếp, nên trong tìm kiếm cũng phải tính đến chi phí cho việc sắp xếp. Nếu dãy khoá luôn biến động, thì chi phí cho việc sắp xếp sẽ lớn, và đó là một trở ngại lớn với phương pháp tìm kiếm này.

Chương này đã giới thiệu với bạn đọc một số phương pháp sắp xếp và tìm kiếm khá thông dụng, cùng với ưu và nhược điểm của mỗi phương pháp. Mong rằng bạn đọc sẽ tìm hiểu thật kỹ và thành công trong việc ứng dụng chúng trong các bài toán tin học mà mình sẽ triển khai.

BÀI TẬP CHƯƠNG 3

1. Cho dãy khóa X: 50 8 34 6 98 17 83 25 66 42 21 59 63 71 85

   a. Hãy minh họa ba phương pháp sắp xếp đơn giản đã nêu, qua dãy khóa X theo thứ tự tăng dần, giảm dần.

   b. Tính số lượng phép toán đổi chỗ trong mỗi trường hợp, đưa ra nhận xét.

   c. Minh họa các phương pháp sắp xếp phân đoạn, vun đống và trộn qua dãy khóa X theo thứ tự tăng dần, giảm dần.

d. Minh họa phương pháp tìm kiếm tuần tự khóa k1 = 98, k2 = 63, k3 = 44 trên dãy khóa X và tính số phép toán so sánh trong mỗi trường hợp.

e. Với dãy khóa X được sắp xếp theo chiều tăng dần, hãy minh họa việc tìm kiếm các khóa k1, k2, k3 trên dãy X theo phương pháp tìm kiếm nhị phân.

f. Với dãy khóa X được sắp xếp theo chiều giảm dần, hãy minh họa việc tìm kiếm các khóa k1, k2, k3 trên dãy X theo phương pháp tìm kiếm nhị phân.

2. Hãy viết lại giải thuật của ba phương pháp sắp xếp đơn giản và "chạy chậm" các giải thuật này khi chúng được sử dụng để sắp xếp dãy khóa X (bạn đọc cũng nên tự lấy thêm các dãy khóa khác để minh họa).

3. Viết lại các giải thuật của các phương pháp sắp xếp phân đoạn, vun đống và trộn. "Chạy chậm" các giải thuật này trên dãy khóa X.

4. Viết chương trình ứng dụng thực hiện các yêu cầu sau:

- Nhập một dãy X, với n số từng đôi một khác nhau.
- Nhập số k, bằng phương pháp tìm kiếm tuần tự, hãy cho biết số k có trong dãy X hay không, nếu có cho biết vị trí của nó.
- Sắp xếp dãy theo chiều tăng dần bằng phương pháp lựa chọn/nổi bọt/thêm dần/phân đoạn/vun đống/trộn.
- Hiển thị dãy vừa sắp xếp ra màn hình.
- Nhập số m, bằng phương pháp tìm kiếm nhị phân, hãy cho biết số m có trong dãy X (vừa sắp) hay không, nếu có cho biết vị trí của nó.
- Sắp xếp dãy theo chiều giảm dần bằng phương pháp lựa chọn/nổi bọt/thêm dần/phân đoạn/vun đống/trộn.
- Hiển thị dãy vừa sắp xếp ra màn hình.

- Nhập số t, bằng phương pháp tìm kiếm nhị phân, hãy cho biết số t có trong dãy X (vừa sắp) hay không, nếu có cho biết vị trí của nó.

5. Viết chương trình

- Tạo một danh sách sinh viên, mỗi sinh viên gồm các thông tin: Mã sinh viên, họ và tên, năm sinh, giới tính và điểm tổng kết.
- Hiển thị danh sách ra màn hình sao cho điểm trung bình của sinh viên theo thứ tự giảm dần (sử dụng một trong các phương pháp sắp xếp phân đoạn/vun đống/trộn).
- Hiển thị danh sách ra màn hình sao cho tên của sinh viên theo thứ tự trong bảng chữ cái (sử dụng một trong ba phương pháp sắp xếp đơn giản).
- Nhập vào mã của một sinh viên, bằng phương pháp tìm kiếm tuần tự/nhị phân hãy cho biết sinh viên có mã vừa nhập có trong danh sách không, nếu có hãy cho biết vị trí của nó trong danh sách.

Chương 4 DANH SÁCH TUYẾN TÍNH

Trong chương này, chúng ta sẽ nghiên cứu danh sách tuyến tính, một trong các mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong việc cài đặt các bài toán ứng dụng. Các phương pháp cài đặt danh sách khác nhau sẽ được xem xét. Hai kiểu dữ liệu trừu tượng đặc biệt quan trọng là ngăn xếp (Stack) và hàng đợi (Queue) sẽ được nghiên cứu. Chương này cũng sẽ trình bày một số ứng dụng phổ biến của danh sách.

1. KHÁI NIỆM DANH SÁCH TUYẾN TÍNH

1.1. Khái niệm danh sách

Về mặt toán học, danh sách là một dãy hữu hạn các phần tử thuộc cùng một lớp đối tượng nào đó. Chẳng hạn, danh sách sinh viên của một lớp, danh sách các số nguyên, danh sách các báo xuất bán hàng ngày ở thủ đô, v.v...

Giả sử L là một danh sách có n phần tử (n >= 0).

L = (a1, a2,..., an)

Ta gọi n là độ dài của danh sách. Nếu n >= 1 thì a1 được gọi là phần tử đầu tiên, an được gọi là phần tử cuối cùng của danh sách L. Nếu n = 0 thì danh sách L được gọi là danh sách rỗng.

Một tính chất quan trọng của danh sách là các phần tử của nó được sắp tuyến tính: nếu n > 1 thì phần tử a_i "đi trước" phần tử a_{i+1}. Ta gọi a_i (i = 1, 2,..., n) là phần tử ở vị trí thứ i của danh sách. Nghĩa là, một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì danh sách đó được gọi là danh sách tuyến tính.

1.2. Các phép toán trên danh sách

Khi mô tả một mô hình dữ liệu, chúng ta cần xác định các phép toán có thể thực hiện trên mô hình toán học được dùng làm cơ sở cho mô hình dữ liệu. Có rất nhiều phép toán trên danh sách. Trong các ứng dụng, thông thường chúng ta chỉ sử dụng một nhóm các phép toán nào đó. Sau đây là một số phép toán cơ bản trên danh sách tuyến tính.

Giả sử L là một danh sách, các phần tử của nó có kiểu ***Item***, k là vị trí của một phần tử trong danh sách. Các phép toán sẽ được mô tả bởi các hàm sau đây:

1. Khởi tạo danh sách rỗng

void Initialize(***List*** *L);

2. Xác định độ dài của danh sách

int Length(***List*** *L);

3. Loại phần tử ở vị trí thứ k của danh sách

void Delete(***int*** k, ***List*** *L);

4. Xen phần tử X vào danh sách sau vị trí thứ k

void Insert_After(***Item*** X, ***int*** k, ***List*** *L);

5. Xen phần tử X vào danh sách trước vị trí thứ k

void Insert_Before(***Item*** X, ***int*** k, ***List*** *L);

6. Tìm phần tử X trong danh danh sách

int Search(***Item*** X, ***List*** *L);

Hàm Search trả về 1 nếu X có trong L, ngược lại trả về 0

7. Kiểm tra xem danh sách có rỗng không?

int Empty(***List*** *L); *//Hàm Empty trả về 1 nếu L rỗng, ngược lại trả về 0*

8. Kiểm tra xem danh sách có đầy không?

int Full(***List*** *L); *//Hàm Full trả về 1 nếu L đầy, ngược lại trả về 0*

9. Duyệt danh sách

Trong nhiều ứng dụng chúng ta phải đi qua danh sách, từ đầu đến cuối danh sách và thực hiện một nhóm các thao tác nào đó đối với mỗi phần tử của danh sách.

void Traverse(***List*** *L);

10. Các phép toán khác

Còn có thể kể ra nhiều phép toán khác. Chẳng hạn truy nhập đến phần tử thứ i của danh sách (để tham khảo hoặc thay thế), kết hợp hai danh sách thành một danh sách, tách một danh sách thành nhiều danh sách, v.v...

Ví dụ: Giả sử có danh sách L = (3, 2, 1, 5). Khi đó, thực hiện Delete(3, L) ta được danh sách (3, 2, 5). Kết quả của Insert_Before(1, 6, L) ta được danh sách (6, 3, 2, 1, 5).

Sau đây ta sẽ xét một số loại danh sách và ứng dụng của chúng.

2. LƯU TRỮ KẾ TIẾP CỦA DANH SÁCH TUYẾN TÍNH

Ta biết rằng danh sách tuyến tính là một danh sách hoặc rỗng, hoặc có dạng L = (a1, a2,..., an). Trong danh sách tuyến tính luôn tồn tại một phần tử đầu là **a1** và một phần tử cuối là **an** ($n \geq 1$).

Để lưu trữ danh sách tuyến tính trong bộ nhớ máy tính, một phương pháp rất tự nhiên là sử dụng mảng một chiều, trong đó mỗi thành phần của mảng lưu trữ một phần tử tương ứng của danh sách, các phần tử kế nhau của danh sách được lưu trữ trong các thành phần kế nhau của mảng. Lưu trữ danh sách theo cách này gọi là lưu trữ kế tiếp.

Tuy nhiên, việc sử dụng mảng một chiều cũng có những ưu điểm và nhược điểm nhất định của nó:

+ Vì mảng được lưu trữ kế tiếp nên việc truy nhập vào một thành phần nào đó được thực hiện trực tiếp dựa vào địa chỉ tính được (chỉ số), nên tốc độ nhanh và đồng đều đối với mọi phần tử.

+ Khi khai báo một mảng ta phải xác định số lượng phần tử của mảng, điều này sẽ tuỳ thuộc vào số lượng phần tử của danh sách mà mảng sẽ lưu trữ, nhưng điều này rất khó thực hiện vì số lượng phần tư của danh sách luôn luôn biến động. Do đó, có thể dẫn đến lãng phí bộ nhớ (có những phần tử mảng không được sử dụng) hoặc thiếu bộ nhớ (do tất cả các phần tử mảng đã được sử dụng trong khi ta cần thêm vào danh sách một số phần tử nào đó).

Sau đây ta trình bày cách cài đặt danh sách tuyến tính bởi mảng một chiều:

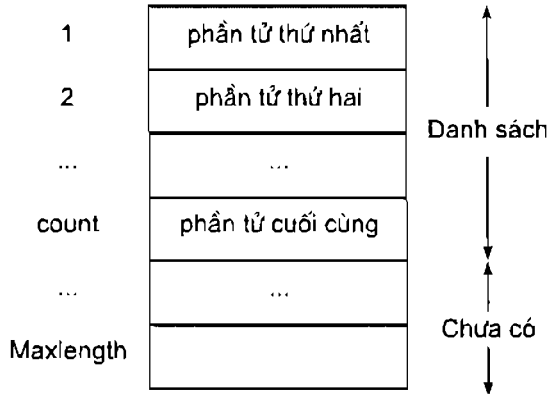
Giả sử độ dài tối đa của danh sách là một số nguyên dương N nào đó, các phần tử trong danh sách có kiểu dữ liệu là ***Item***. ***Item*** có thể là các kiểu dữ liệu đơn (số nguyên, số thực, ký tự), hoặc các kiểu dữ liệu có cấu trúc (chuỗi, cấu trúc). Danh sách được biểu diễn bởi một cấu trúc gồm hai thành phần dữ liệu.

+ Thành phần thứ nhất là mảng các ***Item***, phần tử thứ i của danh sách được lưu trữ bởi phần tử thứ i của mảng.

+ Thành phần thứ hai ghi chỉ số của phần tử mảng lưu trữ phần tử cuối cùng của danh sách.

Ta có khai báo cấu trúc dữ liệu của danh sách như sau:

```
#define Maxlength = N
// Khai báo kiểu Item (nếu cần)
struct List
{
        Item E[Maxlength];
        int count;
};
struct List L; //Khai báo danh sách L
```



Hình 4.1: Mảng biểu diễn danh sách

Trong cách cài đặt danh sách bởi mảng, các phép toán trên danh sách được thực hiện rất dễ dàng. Để khởi tạo danh sách rỗng chỉ cần một lệnh gán:

L.count = 0;

Độ dài thực của danh sách là L.count, danh sách đầy nếu L.count=Maxlength.

Ví dụ: Khai báo danh sách lưu trữ thông tin về sinh viên

```
#define Maxlength 100
struct student                  //Item ở đây là student
{
  char std_no[10];              //Mã sinh viên
  char std_name[30];            //Họ tên
  int age;                      //Tuổi
  float avg_point;              //Điểm trung bình
};
struct List
{
  struct student E[Maxlength];
  int count;
};
struct List L;
```

Dưới đây ta cài đặt hai phép toán trên danh sách: phép toán bổ sung một phần tử mới vào danh sách và phép toán loại bỏ một phần tử khỏi danh sách.

1. Loại bỏ một phần tử ở vị trí k trong danh sách

```
int DeleteL(int k, struct List *L)
{
  int i;
  if (k>=1 && k <= L->count)
  {
        i = k;
        while (i < L->count)
        {
              L->E[i] = L->E[i+1] ;
              i = i + 1;
        }
        L->count = L->count - 1;
        return 1;
  }
  else return 0;
}
```

Hàm DeleteL thực hiện phép loại một phần tử ở vị trí k trong danh sách. Phép toán được thực hiện khi danh sách không rỗng và k chỉ vào một phần tử trong danh sách. Giá trị trả về của hàm cho biết phép toán có được thực hiện thành công hay không (trả về 1 nếu thành công, trả về 0 nếu không thành công). Khi loại bỏ, ta phải dồn các phần tử ở các vị trí k+1, k+2,..., L.count lên trên một vị trí và giảm số lượng phần tử của danh sách đi một đơn vị (L.count = L.count – 1).

2. Bổ sung một phần tử vào trước phần tử ở vị trí k trong danh sách (dữ liệu của phần tử này được lưu trong biến X).

```
int InsertL(int k, Item X, struct List *L)
{
  int i;
```

```
if (L->count < Maxlength && k <= L->count && k>=1)
{
        i = L->count + 1;
        while (i > k)
        {
                L->E[i] = L->E[i-1] ;
                i = i - 1;
        }
        L->count = L->count + 1;
        L->E[k] = X;
        return 1;
}
else return 0;
}
```

Hàm InsertL thực hiện phép bổ sung một phần tử vào trước phần tử ở vị trí k trong danh sách. Phép toán được thực hiện khi danh sách chưa đầy và k chỉ vào một phần tử trong danh sách. Giá trị trả về của hàm cho biết phép toán có được thực hiện thành công hay không (trả về 1: thành công, trả về 0: không thành công). Khi bổ sung, ta phải dãn các phần tử ở các vị trí L.count,..., k+1, k xuống dưới một vị trí và tăng số lượng phần tử của danh sách lên một đơn vị (L.count = L.count + 1).

**** Nhận xét về phương pháp cài đặt danh sách bởi mảng:***

Việc cài đặt danh sách bởi mảng có một số ưu điểm và nhược điểm sau:

Ưu điểm: Do tính chất của mảng, nên việc cài đặt danh sách bởi mảng cho phép ta truy nhập trực tiếp vào bất kỳ phần tử nào trong danh sách nên tốc độ truy nhập nhanh và đồng đều đối với mọi phần tử. Các phép toán cũng đều được thực hiện một cách dễ dàng.

Nhược điểm: Khi thực hiện các phép toán bổ sung một phần tử vào danh sách hoặc loại bỏ một phần tử ra khỏi danh sách ở vị trí trí k

nào đó, ta phải đẩy tất cả các phần tử sau k xuống dưới hoặc lên trên một vị trí, nên tốn nhiều thời gian. Tuy nhiên, nhược điểm chủ yếu của phương pháp cài đặt này là không gian nhớ cố định dành để lưu trữ các phần tử của danh sách. Không gian nhớ này bị quy định bởi kích thước của mảng (kích thước của mảng được xác định khi khai báo và nó không thể thay đổi trong khi thực hiện chương trình). Do đó có thể dẫn đến trường hợp lãng phí bộ nhớ (do khai báo kích thước mảng quá lớn so với số lượng các phần tử của danh sách) hoặc thiếu bộ nhớ (mảng đã đầy trong khi ta muốn bổ sung thêm một số phần tử nào đó vào danh sách).

Để khắc phục các nhược điểm trên đây người ta sử dụng một phương pháp khác để cài đặt danh sách tuyến tính đó là danh sách móc nối.

3. DANH SÁCH MÓC NỐI

Như đã nêu ở phần trên, lưu trữ kế tiếp đối với danh sách tuyến tính đã bộc lộ rõ nhược điểm trong trường hợp thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử, trường hợp xử lý đồng thời nhiều danh sách, v.v...

Việc sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính, mà ta gọi là danh sách móc nối (hay còn gọi là danh sách liên kết), chính là một giải pháp nhằm khắc phục nhược điểm trên. Tuy nhiên, trước khi tìm hiểu về danh sách móc nối ta nhắc lại một số khái niệm về con trỏ, phương tiện được sử dụng để cài đặt danh sách móc nối.

3.1. Kiểu con trỏ và các khái niệm liên quan

Tất cả các biến có kiểu dữ liệu mà ta đã nghiên cứu như số, ký tự, mảng, cấu trúc..., được gọi là biến tĩnh vì chúng được xác định một cách rõ ràng khi khai báo, sau đó chúng được dùng thông qua tên. Thời gian tồn tại của biến tĩnh cũng là thời gian tồn tại của khối chương trình có chứa khai báo biến này. Chẳng hạn, các biến tĩnh được khai báo trong chương trình (biến toàn cục) sẽ tồn tại từ khi

chương trình được thực hiện cho đến khi kết thúc chương trình, còn các biến tĩnh được khai báo trong một hàm (biến cục bộ) sẽ tồn tại từ khi hàm được triệu gọi cho đến khi kết thúc.

Ngoài các biến tĩnh được xác định trước, người ta còn có thể tạo ra các biến trong lúc chạy chương trình, tuỳ theo nhu cầu. Việc tạo ra các biến theo kiểu này được gọi là cấp phát bộ nhớ động, các biến được tạo ra được gọi là biến động.

Các biến động không có tên. Trong C/C++, để tạo ra biến động, người ta sử dụng một kiểu biến đặc biệt, gọi là con trỏ và các hàm/toán tử cấp phát bộ nhớ động (*malloc(), calloc(), realloc() trong thư viện malloc.h, toán tử new*) thông qua con trỏ. Khi không sử dụng biến động nữa, người ta có thể xoá nó khỏi bộ nhớ, việc này gọi là thu hồi bộ nhớ động. Để thu hồi bộ nhớ dành cho biến động, người ta dùng hàm *free()*/toán tử *delete* và thông qua con trỏ đã sử dụng để tạo ra biến động.

So với biến tĩnh, việc sử dụng biến động có ưu điểm là tiết kiệm được bộ nhớ. Bởi vì, khi cần dùng biến động thì người ta sẽ tạo ra nó và khi không cần nữa người ta lại có thể xoá nó khỏi bộ nhớ. Còn đối với các biến tĩnh, chúng được xác định và cấp phát bộ nhớ khi biên dịch, chúng sẽ chiếm giữ bộ nhớ trong suốt thời gian chương trình làm việc. Chẳng hạn, nếu cần sử dụng một mảng ta phải khai báo ngay ở phần đầu chương trình, ngay lúc này ta đã phải xác định kích thước của mảng và thường khai báo dôi ra, gây lãng phí bộ nhớ.

3.1.1. Con trỏ

Con trỏ là một biến dùng để chứa địa chỗ nhớ chỉ của một biến khác.

Cách khai báo con trỏ

<Kiểu dữ liệu> <*tên con trỏ>;

Ví dụ:

int *p, *q; //Khai báo p và q là hai con trỏ kiểu nguyên

Nghĩa là p, q được dùng để lưu địa chỉ của các biến nguyên.

3.1.2. Các phép toán con trỏ

Giả sử có khai báo ***int*** *p, *q, x;

Khi đó ta có thể thực hiện các phép toán

+ Gán địa chỉ cho con trỏ

Ví dụ: p = &x; //Gán địa chỉ của biến x cho p, hay p trỏ vào x.

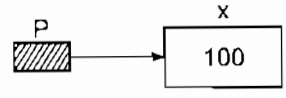
+ Phép gán hai con trỏ cùng kiểu. *Ví dụ*: q = p; //q và p cùng trỏ vào x.

+ Phép so sánh hai con trỏ cùng kiểu gồm: so sánh == (bằng nhau) và phép sánh != (khác nhau).

Phép toán một ngôi * được sử dụng với con trỏ để trả về giá trị của chỗ nhớ do con trỏ trỏ đến, hoặc làm thay đổi giá trị của chỗ nhớ đó.

Cách viết <***tên_con_trỏ***>

Ví dụ:




```
int *p, x, y;
x = 100;
p = &x;          //p trỏ vào x
y = *p;          //khi đó ta có giá trị của y = 100
*p = 500;        //Khi đó ta cũng có x = 500
```

3.1.3. Giá trị NULL

NULL là một giá trị con trỏ đặc biệt dành cho các biến con trỏ, nó được dùng để báo rằng con trỏ không lưu địa chỉ của biến nào. Giá trị NULL có thể được đem gán cho bất kỳ biến con trỏ nào. Đương nhiên khi đó việc thâm nhập vào biến động thông qua con trỏ có giá trị NULL là vô nghĩa.

3.1.4. Con trỏ cấu trúc

Con trỏ chứa địa chỉ của một biến cấu trúc được gọi là con trỏ cấu trúc, khi đó ta có thể thao tác với cấu trúc thông qua con trỏ. Việc truy nhập vào các thành phần của cấu trúc bằng con trỏ được viết theo cách sau:

<tên_con_trỏ> —> <tên thành phần>

Ví dụ:

```
struct   Hoc_sinh
{
         char   Ho_ten[25];
         int    tuoi;
         float  diem;
};
struct Hoc_sinh *p, h;
p = &h;
```

Khi đó việc truy xuất vào các thành phần của cấu trúc h thông qua con trỏ p được viết như sau:

```
strcpy(p->Ho_ten, "Nguyen Trong Huan");
p->diem = 7.4;
cin>>p->tuoi;
```

3.1.6. Cấp phát và thu hồi bộ nhớ động

Trong ngôn ngữ lập trình C, có thể sử dụng các hàm cấp phát bộ nhớ động gồm: malloc(), calloc(), các hàm này được định nghĩa trong thư viện malloc.h

+ Hàm malloc(), cấp phát cho con trỏ một vùng nhớ liên tiếp, kích thước vùng nhớ được chỉ ra bởi tham số size.

Cú pháp: ***void *malloc(size_t size)***

Trong đó size là kích thước vùng nhớ được cấp phát cho con trỏ được tính bằng byte.

Ví dụ:

```
int *p;
p=(int*) malloc (sizeof(int));
```

Câu lệnh trên cấp phát cho con trỏ p một chỗ nhớ có kích thước bằng một dữ liệu kiểu ***int***.

sizeof là toán tử trả về kích thước chỗ nhớ của một dữ liệu thuộc một kiểu dữ liệu nào đó.

+ Hàm calloc(), cấp phát một vùng nhớ gồm n chỗ nhớ.

Cú pháp: ***void** *calloc(**int** n, **size_t** size)*

Ví dụ:

```
int *p;
p=(int*)calloc(1, sizeof(int);
```

Dưới đây ta xét một chương trình được cài đặt với con trỏ cấu trúc:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct   Hoc_sinh
{
        char   ht[25];
        int    tuoi;
        char   qq[40];
};
void Nhap_du_lieu(struct Hoc_sinh *p)
{
        cout<< "\tHo ten: " ; fflush(stdin); gets(p->ht);
        cout<< "\tTuoi: "; cin>>p->tuoi;
        cout<< "\tQue quan: "; fflush(stdin) ; gets(p->qq);
}
void Hien_thi(struct Hoc_sinh p)
{       cout<< "\tHo ten: "<<p->ht<<endl;
        cout<< "\tTuoi: "<<p->tuoi<<endl;
        cout<< "\tQue quan: "<<p->qq<<endl;
}
void main()
{
        struct Hoc_sinh *p;
        p=(struct Hoc_sinh*) malloc(sizeof(struct Hoc_sinh));
```

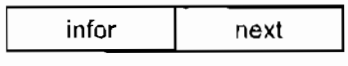
```
        cout<< "Nhap thong tin ve hoc sinh"<<endl;
        Nhap_du_lieu(p);
        cout<< "Thong tin hoc sinh vua nhap"<<endl;
        Hien_thi(*p);
        free(p);
        getch();
}
```

Qua ví dụ trên ta thấy rằng, để đưa dữ liệu vào bộ nhớ thông qua biến con trỏ ta cần phải cấp phát bộ nhớ cho nó và sau khi không sử dụng nữa ta có thể xoá nó khỏi bộ nhớ. Tuy nhiên, nếu chỉ lưu trữ dữ liệu đơn giản như vậy thì ta không cần đến biến con trỏ, nó được sử dụng trong một ứng dụng quan trọng hơn đó là việc cài đặt danh sách liên kết. Sau đây ta xét tới một số dạng danh sách móc nối.

3.2. Danh sách móc nối đơn

3.2.1. Nguyên tắc

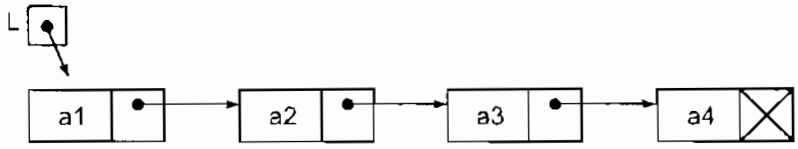
Trong cách cài đặt này, danh sách móc nối được tạo nên từ các phần tử nhỏ mà ta gọi là nút (Node). Các nút này có thể nằm bất kỳ đâu trong bộ nhớ máy tính. Mỗi nút là một cấu trúc gồm hai thành phần, *infor* chứa thông tin của phần tử trong danh sách, *next* là một con trỏ, nó trỏ vào nút đứng sau. Qui cách của mỗi nút có thể hình dung như sau:



Riêng nút cuối cùng thì không có nút đứng sau nó nên thành phần *next* của nút này có giá trị NULL để báo kết thúc danh sách.

Để có thể truy nhập vào mọi nút trong danh sách, ta phải truy nhập từ nút đầu tiên, nghĩa là cần có một con trỏ L trỏ tới nút đầu tiên này.

Nếu dùng mũi tên để chỉ mối nối, ta sẽ có hình ảnh của một danh sách móc nối đơn như hình 4.2:



Hình 4.2: Biểu diễn danh sách móc nối đơn

Dấu ⊠ chỉ giá trị trường *next* của phần tử này bằng NULL.

Giả sử các phần tử trong danh sách có kiểu dữ liệu là ***Item***. Dưới đây là khai báo cấu trúc dữ liệu biểu diễn danh sách móc nối đơn.

```
//định nghĩa kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item infor;
        struct Node *next;
};
struct Node *L;  //Khai báo con trỏ L trỏ vào đầu danh sách
```

L = NULL nếu danh sách rỗng.

Ví dụ: Khai báo danh sách lưu trữ thông tin về sinh viên:

```
struct student                  //Item ở đây là student
{
        char std_no[10];        //Mã sinh viên
        char std_name[30];      //Họ tên
        int age;                //Tuổi
        float avg_point;        //Điểm trung bình
};
struct Node
{
        struct student infor;
        struct Node *next;
};
struct Node *L;  //Khai báo biến con trỏ L trỏ vào đầu danh sách
```

3.2.2. Các phép toán trên danh sách móc nối đơn

Bây giờ chúng ta sẽ xem xét một số phép toán tác động trên danh sách nối đơn.

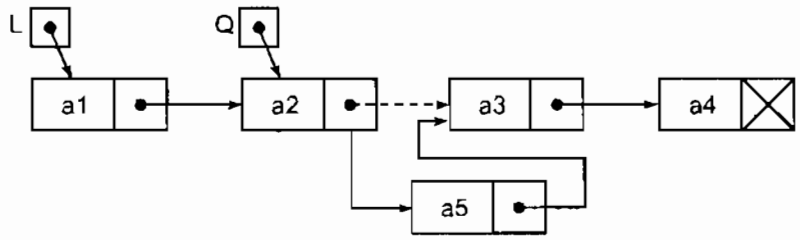
Điều kiện để danh sách móc nối đơn rỗng là L = NULL. Do đó, để khởi tạo danh sách rỗng ta chỉ cần lệnh gán: L = NULL;

Danh sách móc nối chỉ đầy khi không còn không gian nhớ để cấp phát cho các phần tử mới của danh sách. Chúng ta giả thiết điều này không xảy ra, nghĩa là danh sách móc nối không bao giờ đầy. Do đó, phép toán bổ sung một phần tử vào danh sách luôn luôn được thực hiện.

a. Bổ sung một nút mới vào danh sách móc nối đơn

Giả sử Q là một con trỏ, trỏ vào một nút trong danh sách, ta cần bổ sung một phần tử mới với thông tin lưu trong biến X vào sau nút được trỏ bởi Q. Phép toán này được thực hiện bởi thủ tục sau:

```
void InsertAfter(struct Node **L, struct Node *Q , Item X)
{
  struct Node *p;
//1. Tạo một nút mới
        p=(struct Node *)malloc(sizeof(struct Node));
        p->infor = X;
//2. Thực hiện bổ sung, nếu danh sách rỗng thì bổ sung nút mới vào thành nút đầu tiên, ngược lại bổ sung nút mới vào sau nút được trỏ bởi Q.
        if (*L == NULL)
        {
                p->next = NULL;
                *L = p;
        }
        else {
                p->next = Q->next;
                Q->next = p;
        }
}
```



Hình 4.3: Mô tả phép bổ sung một phần tử vào sau nút trỏ bởi Q trong danh sách

Giả sử bây giờ ta cần bổ sung nút mới vào trước nút được trỏ bởi Q. Phép toán này phức tạp hơn. Khó khăn là ở chỗ, nếu Q không phải là nút đầu tiên của danh sách (Q≠L) thì ta không thể xác định được nút đứng trước Q để kết nối nó với nút mới. Có thể giải quyết khó khăn bằng cách, đầu tiên ta vẫn bổ sung nút mới vào sau Q, sau đó trao đổi giá trị chứa trong phần infor giữa nút mới và nút được trỏ bởi Q. Thủ tục thực hiện phép toán này xin dành cho bạn đọc.

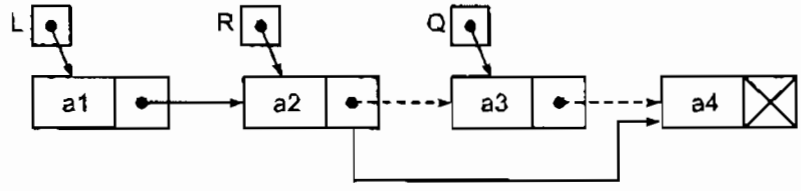
b. Loại bỏ một nút ra khỏi danh sách móc nối đơn

Cho danh sách móc nối đơn được trỏ bởi L. Q là một con trỏ, trỏ vào một nút trong danh sách. Giả sử ta cần loại bỏ nút được trỏ bởi Q. Ở đây ta cũng gặp khó khăn là nếu Q không phải là nút đầu tiên thì không xác định được nút đứng trước Q. Trong trường hợp này ta phải tìm đến nút đứng trước Q và cho con trỏ R trỏ vào nút đó, tức là Q = R->next. Sau đó ta mới thực hiện loại bỏ nút Q. Ta có thủ tục sau:

```
int DeleteL(struct Node **L, struct Node *Q, Item &X)
{
  struct Node *R;
  //1. Trường hợp danh sách rỗng
  if (*L = = NULL)
  {
        return 0;
  }
  X = Q->infor;   //lưu thông tin của nút cần loại bỏ vào biến X
```

```
    //2. Trường hợp nút trỏ bởi Q là nút đầu tiên
        if (Q == *L)
        {
            *L = Q->next; free(Q);
            return 1;
        }
    //3. Tìm đến nút đứng trước nút trỏ bởi Q
        R = *L;
        while (R->next != Q)
            R = R->next;
    //4. Loại bỏ nút trỏ bởi Q
        R->next = Q->next;     free(Q);
}
```

Phép toán loại bỏ được mô tả bởi hình 4.4.



Hình 4.4: Mô tả phép loại bỏ một phần tử ra khỏi danh sách

c. Ghép hai danh sách móc nối đơn

Giả sử có hai danh sách móc nối đơn lần lượt được trỏ bởi L1 và L2. Thủ tục sau thực hiện việc ghép hai danh sách đó thành một danh sách mới được trỏ bởi L1.

```
void COMBINE(struct Node **L1, struct Node *L2)
{
  struct Node *R;
  //1. Trường hợp danh sách trỏ bởi L2 rỗng
  if (L2 == NULL) return;
  //2. Trường hợp danh sách trỏ bởi P rỗng
  if (*L1 == NULL)
```

```
{
        *L1 = L2; return;
}
//3. Tìm đến nút cuối danh sách trỏ bởi P
R = *L1;
while (R->next != NULL)
        R = R->next;
//4. Ghép
R->next = L2;
}
```

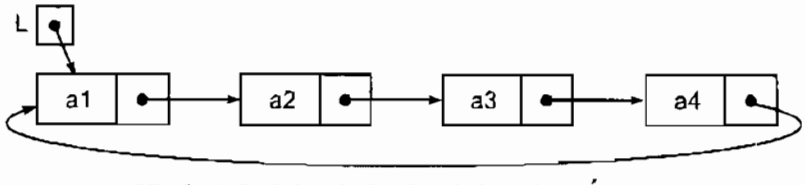
Nhận xét: Rõ ràng với các danh sách tuyến tính mà kích thước luôn biến động trong quá trình xử lý hay thường xuyên có các phép bổ sung và loại bỏ tác động, thì việc lưu trữ bằng danh sách móc nối như trên tỏ ra thích hợp. Tuy nhiên, cách cài đặt này cũng có những nhược điểm nhất định:

Chỉ có phần tử đầu tiên trong danh sách được truy nhập trực tiếp, các phần tử khác chỉ được truy nhập sau khi đã đi qua các phần tử đứng trước nó.

Ở mỗi nút trong danh sách phải có thêm trường *next* để lưu trữ địa chỉ của nút tiếp theo, do đó với cùng một danh sách thì việc cài đặt bởi danh sách móc nối sẽ tốn bộ nhớ hơn so với cài đặt bằng mảng.

3.3. Danh sách nối vòng

Một cải tiến của danh sách móc nối đơn là kiểu danh sách móc nối vòng. Nó khác với danh sách móc nối đơn ở chỗ: trường *next* của nút cuối cùng trong danh sách không phải bằng NULL, mà nó trỏ đến nút đầu tiên trong danh sách, tạo thành một vòng tròn. Hình ảnh của nó như hình 4.5.

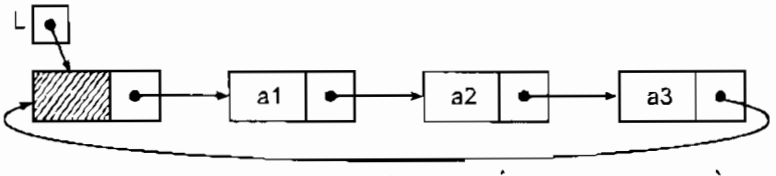


Hình 4.5: Mô tả danh sách móc nối vòng

Cải tiến này làm cho việc truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng được. Như vậy, đối với danh sách móc nối vòng chỉ cần cho biết con trỏ trỏ tới nút muốn loại bỏ ta sẽ thực hiện được vì luôn tìm được đến nút đứng trước đó. Với phép ghép, phép tách cũng có những thuận lợi nhất định.

Tuy nhiên, danh sách nối vòng có một nhược điểm rất rõ là trong khi xử lý, nếu không cẩn thận sẽ dẫn tới một chu trình không kết thúc, bởi vì không biết được vị trí kết thúc danh sách.

Để khắc phục nhược điểm này, người ta đưa thêm vào danh sách một nút đặc biệt gọi là "nút đầu danh sách". Trường infor của nút này không chứa dữ liệu của phần tử nào và con trỏ L bây giờ trỏ tới nút đầu danh sách này. Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức không bao giờ rỗng. Hình ảnh của nó minh họa như hình 4.6.



Hình 4.6: Mô tả danh sách móc nối vòng có nút đầu

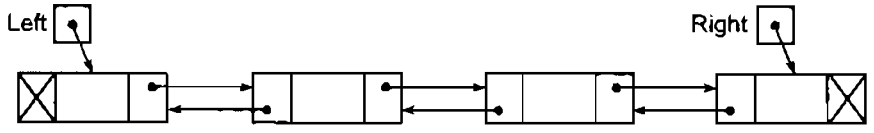
Sau đây là đoạn giải thuật bổ sung một nút vào thành nút đầu tiên trong danh sách có "nút đầu danh sách" trỏ bởi L.

```
P=(struct Node*) malloc(sizeof(struct Node));
P->infor = X;
P->next = L->next;
L->next = P;
```

3.4. Danh sách móc nối hai chiều

Khi làm việc với danh sách, có những xử lý trên mỗi nút của danh sách lại liên quan đến cả nút đứng trước và nút đứng sau. Trong

những trường hợp như thế, để thuận tiện, người ta đưa vào mỗi nút của danh sách hai con trỏ: *Next_Left* trỏ đến nút đứng trước và *Next_Right* trỏ đến nút đứng sau nó. Để truy nhập vào danh sách ta dùng hai con trỏ: con trỏ *Left* trỏ vào nút đầu tiên và con trỏ *Right* trỏ vào nút cuối cùng của danh sách. Hình ảnh của danh sách móc nối hai chiều được minh họa trên hình 4.7.



Hình 4.7: Mô tả danh sách móc nối đôi

Ta có thể khai báo cấu trúc dữ liệu danh sách móc nối hai chiều như sau:

```
//Khai báo kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item infor;
        struct Node *Next_Left, *Next_Right;
};
struct Node *Left, *Right;
```

Việc cài đặt danh sách móc nối hai chiều sẽ tiêu tốn nhiều bộ nhớ hơn so với danh sách móc nối đơn. Song bù lại, danh sách móc nối đôi có những ưu điểm mà danh sách móc nối đơn không thể có được, chẳng hạn: khi xem xét danh sách móc nối đôi ta có thể lùi lại sau, hoặc tiến lên trước.

Các phép toán trên danh sách móc nối hai chiều được thực hiện dễ dàng hơn. Chẳng hạn, khi thực hiện phép toán loại bỏ, với danh sách móc nối đơn, ta không thể thực hiện được nếu không biết nút đứng trước nút cần loại bỏ. Trong khi đó, ta có thể tiến hành dễ dàng trên danh sách móc nối hai chiều.

Dưới đây là một số giải thuật tác động lên danh sách móc nối hai chiều nói trên.

3.4.1. Phép bổ sung một nút mới

Cho hai con trỏ Left và Right lần lượt trỏ tới nút đầu và nút cuối của một danh sách móc nối hai chiều, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện bổ sung một nút mới, mà dữ liệu chứa ở biến X, vào trước nút trỏ bởi con trỏ M.

```
void Bo_sung(struct Node **Left, struct Node **Right, struct Node
                *M, Item X)
{
  struct Node *P;
  //1. Tạo nút mới
  P = (Node*)malloc(sizeof(Node));
  P->infor = X;
  //Trường hợp danh sách rỗng
  if (*Right == NULL)
  {
        P->Next_Left = NULL; P->Next_Right = NULL;
        *Left = P; *Right = P;
  }
  else
  //Trường hợp M trỏ tới nút đầu tiên
  if (M == *Left)
  {     P->Next_Left = NULL;
        P->Next_Right = M
        M->Next_Left = P;
        *Left = P;
  }
  else //Bổ sung vào giữa
  {     P->Next_Left = M->Next_Left;
        P->Next_Right = M;
        M->Next_Left = P;
        P->Next_Left->Next_Right = P;
        }
}
```

3.4.2. Loại bỏ một nút trên danh sách

Cho hai con trỏ Left và Right lần lượt trỏ tới nút đầu và nút cuối của một danh sách móc nối hai chiều, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện loại bỏ nút trỏ bởi M ra khỏi danh sách.

```
int Loai_bo(struct Node **Left, struct Node **Right, struct Node *M)
{
  if (*Left = = NULL)
        return 0;
  else
        if (*Left = = *Right)
        {       *Left = NULL; *Right = NULL;
        }
        else
              if (M = = *Left)
              {       *Left = (*Left)->Next_Right;
                      (*Left)->Next_Left = NULL;
              }
              else if (M == Right)
              {       Right = Right^.Next_Left;
                      Right^.Next_Right = NULL;
              }
              else    {
                      M->Next_Left->Next_Right = M->Next_Right;
                      M->Next_Right->Next_Left = M->Next_Left;
                      }
}
```

Chú ý: Trong các ứng dụng, người ta cũng thường sử dụng các danh sách móc nối hai chiều vòng tròn, có nút đầu danh sách. Với loại danh sách này, ta có tất cả các ưu điểm của danh sách móc nối hai chiều và danh sách vòng tròn.

3.5. Ứng dụng danh sách móc nối: các phép tính số học trên đa thức

Trong mục này ta sẽ xét các phép tính số học cơ bản (cộng, trừ, nhân, chia) đối với đa thức một ẩn có dạng:

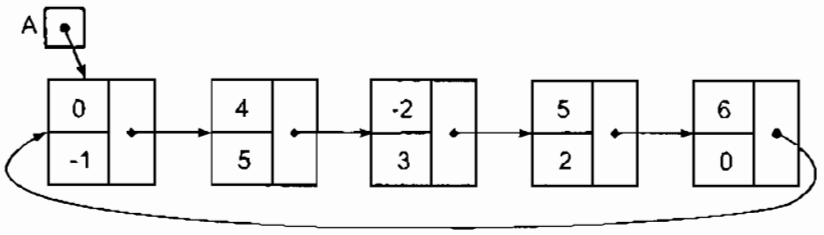
$$A(x) = a_n x^n + a_{n-1}x^{n-1} + ...+ a_1 x + a_0 \qquad (1)$$

Mỗi hạng thức của đa thức được đặc trưng bởi hệ số và số mũ của x. Giả sử các hạng thức trong đa thức được sắp xếp theo thứ tự giảm dần của số mũ, như trong đa thức (1). Ta thấy đa thức như một danh sách tuyến tính với các phần tử của danh sách là các hạng thức của đa thức. Khi ta thực hiện các phép toán trên đa thức ta sẽ nhận được đa thức có bậc không thể đoán trước được. Ngay cả những đa thức có bậc xác định thì số các hạng thức của nó cũng biến đổi rất nhiều từ một đa thức này đến một đa thức khác. Do đó, phương pháp tốt nhất là biểu diễn đa thức dưới dạng một danh sách móc nối. Mỗi nút của danh sách là một bản ghi gồm ba trường: coef chỉ hệ số, exp chỉ số mũ của x và con trỏ Next để trỏ tới nút tiếp theo. Cấu trúc dữ liệu mô tả một hạng thức (một nút) như sau:

```
struct Node
{
        float coef;
        int exp;
        struct Node *Next;
};
```

Vì những ưu điểm của danh sách vòng tròn có nút đầu danh sách (không cần kiểm tra danh sách rỗng, mọi thành phần đều có thành phần đi sau), ta chọn danh sách móc nối vòng tròn để biểu diễn đa thức. Với cách chọn này việc thực hiện các phép toán đa thức sẽ rất gọn. Nút đầu danh sách là nút đặc biệt, có exp = -1.

Như vậy với đa thức: $A(x) = 4x^5 - 2x^3 + 5x^2 + 6$ sẽ được biểu diễn như hình 4.8.



Hình 4.8: Danh sách móc nối đôi biểu diễn đa thức

Sau đây chúng ta sẽ xét phép cộng hai đa thức A(x) và B(x). Con trỏ A trỏ tới đầu danh sách biểu diễn đa thức A(x), con trỏ B trỏ tới đầu danh sách biểu diễn đa thức B(x). Sau khi thực hiện phép cộng hai đa thức trên ta được đa thức C(x) và con trỏ C trỏ tới đầu danh sách biểu diễn C(x).

**** Giải thuật***

Trước hết cần phải thấy rằng để thực hiện phép cộng đa thức A(x) với đa thức B(x) ta phải tìm đến từng hạng thức của các đa thức đó, nghĩa là phải dùng hai biến con trỏ P và Q để duyệt qua hai danh sách tương ứng với hai đa thức A(x) và B(x) trong quá trình tìm này.

Ta thấy có những trường hợp sau:

1. EXP(P) = EXP(Q), ta sẽ phải thực hiện cộng giá trị coef ở hai nút đó, nếu giá trị tổng khác không thì phải tạo ra nút mới thể hiện hạng thức tổng đó và gắn vào danh sách ứng với C(x).

2. EXP(P) > EXP(Q) (hoặc ngược lại cũng tương tự): phải sao chép nút P và gắn vào danh sách của C(x).

3. Nếu một danh sách kết thúc trước: phần còn lại của danh sách kia sẽ được sao chép và gắn vào danh sách của C(x).

Mỗi lần một nút mới được tạo ra đều phải gắn vào cuối danh sách C(x). Do đó, cần một con trỏ R trỏ vào nút cuối của danh sách C(x).

Công việc này được thực hiện nhiều lần, vì vậy cần được thể hiện bằng một thủ tục gọi là Attack(h, m, R). Nó thực hiện: lấy một nút mới, đưa vào trường coef của nút này giá trị h (hệ số), đưa vào

trường exp giá trị m (số mũ) và gắn nút mới đó vào sau nút trỏ bởi con trỏ R.

```
void Attack(float h, int m, struct Node *R)
{       struct Node *N;
        N=(struct Node*)malloc(sizeof(struct Node));
        N->coef = h;
        N->exp = m;
        R->Next = N;
        R = N;
}
```

Sau đây là thủ tục cộng hai đa thức:

```
void Add(struct Node *A, struct Node *B, struct Node **C)
{
        struct Node *P, *Q, *R;
        float x;
        P = A; Q = B;
        *C = (struct Node*)malloc(sizeof(struct Node));
        *C->exp = -1; R = *C;
        while (P->exp != -1 && Q->exp != -1)
        if (P->exp = Q->exp)
        {       x = P->coef + Q->coef;
                if (x != 0)
                        Attack(x, P->exp, R);
                P = P->Next; Q = Q->Next;
        }
        else    if (P->exp < Q->exp)
                {       Attack(Q->coef, Q->exp, R);
                        Q = Q->Next;
                }
                else {  Attack(P->coef, P->exp, R);
                }
  while (P->exp != -1)    //Danh sách ứng với B(x) đã hết
```

```
    {
            Attack(P->coef, P->exp, R);
            P = P->Next;
    }
    while (Q->exp != -1)    //Danh sách ứng với A(x) đã hết
    {
            Attack(Q->coef, Q->exp, R);
            Q = Q->Next;
    }
    R->Next = *C;
}
```

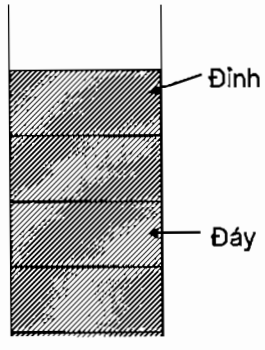
4. STACK VÀ QUEUE

4.1. Stack (Ngăn xếp)

4.1.1. Khái niệm

Ngăn xếp (Stack) là một danh sách tuyến tính, trong đó phép bổ sung một phần tử vào ngăn xếp và phép loại bỏ một phần tử khỏi ngăn xếp luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

Có thể hình dung Ngăn xếp như cơ cấu của một hộp tiếp đạn. Việc đưa đạn vào hộp đạn hay lấy đạn ra khỏi hộp chỉ được thực hiện ở đầu hộp. Viên đạn mới nạp nằm ở đỉnh còn viên đạn nạp đầu tiên nằm ở đáy hộp.



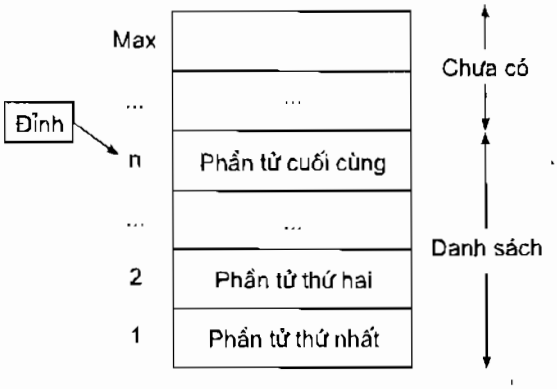
Hình 4.9: Ngăn xếp

4.1.2. Cài đặt ngăn xếp bởi mảng

Giả sử danh sách được biểu diễn là một ngăn xếp, có độ dài tối đa là một số nguyên dương N nào đó, các phần tử của ngăn xếp có kiểu dữ liệu là ***Item***. ***Item*** có thể là các kiểu dữ liệu đơn, hoặc các kiểu dữ liệu có cấu trúc. Chúng ta biểu diễn ngăn xếp bởi một bản ghi gồm 2 trường. Trường thứ nhất là mảng các ***Item***, trường thứ 2 ghi chỉ số của thành phần mảng lưu trữ phần tử ở đỉnh của ngăn xếp. Cấu trúc dữ liệu biểu diễn ngăn xếp được khai báo theo mẫu sau:

```
#define Max N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct   Stack
{
        Item E[Max];
        unsigned int top;
};
struct Stack S; //Khai báo ngăn xếp S
```

Với cách cài đặt này, nếu S.top = 0 thì S là ngăn xếp rỗng, S.top = Max thì S là ngăn xếp đầy.



Hình 4.10: Mảng biểu diễn ngăn xếp

Ví dụ: Đoạn chương trình:

```
#define Max 100
struct   Hoc_sinh
```

```
{       char ho_ten[25];
        int tuoi;
};
struct Stack
{
        struct Hoc_sinh E[max];
        unsigned int top;
};
struct Stack S;
```

Khai báo ngăn xếp S có thể chứa tối đa 100 phần tử, mỗi phần tử (***Item***) là một cấu trúc Hoc_sinh gồm 2 thành phần ho_ten và tuoi.

Các phép toán trên ngăn xếp

Giả sử S là ngăn xếp, các phần tử của nó có kiểu ***Item*** và X là một phần tử có cùng kiểu với các phần tử của ngăn xếp. Ta có các phép toán sau với ngăn xếp S.

a. Khởi tạo ngăn xếp rỗng (ngăn xếp không chứa phần tử nào)

```
void Initialize (struct Stack *S)
{
        S->top = 0;
}
```

b. Kiểm tra ngăn xếp rỗng

```
int Empty (struct Stack S)
{
        return (S.top = = 0);
}
```

Hàm Empty nhận giá trị true nếu S rỗng và false nếu S không rỗng.

c. Kiểm tra ngăn xếp đầy

```
int Full (struct Stack S)
{
  return (S.top == Max);
}
```

Hàm *Full()* nhận giá trị true nếu S đầy và false nếu không.

d. Thêm một phần tử mới vào đỉnh ngăn xếp

Để bổ sung phần tử X vào đỉnh của ngăn xếp S, trước hết kiểm tra xem S có đầy không. Nếu S đầy thì bổ sung không thực hiện được, ngược lại X được bổ sung vào đỉnh của S. Hàm PUSH trả về 1 nếu bổ sung thành công, ngược lại trả về 0.

```
int PUSH (struct Stack *S, Item X)
{
        if (Full(S)) return 0;
        else
        {
                S->top = S->top + 1;
                S->E[S->top] = X;
                return 1;
        }
}
```

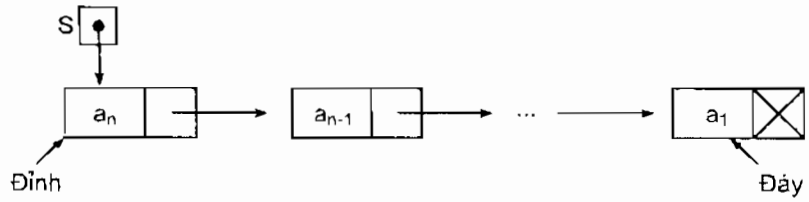
e. Loại bỏ phần tử ở đỉnh của ngăn xếp

Việc loại bỏ được thực hiện nếu S không rỗng, giá trị của phần tử bị loại bỏ được gán cho biến X. Hàm *POP*() trả về 1 nếu loại bỏ thành công, ngược lại trả về 0.

```
int POP (struct Stack *S; Item *X)
{
        if (Empty(*S)) return 0;
        else
        {       *X = S->E[S->top];
                S->top = S->top – 1;
                return 1;
        }
}
```

4.1.3. Cài đặt ngăn xếp bởi danh sách móc nối đơn

Để cài đặt ngăn xếp bởi danh sách móc nối đơn, ta sử dụng con trỏ S trỏ vào phần tử ở đỉnh của ngăn xếp (hình 4.11).



Hình 4.11: Danh sách móc nối đơn biểu diễn ngăn xếp

Cấu trúc dữ liệu của ngăn xếp được khai báo như sau:

```
struct Node
{
        Item Infor;
        Node *Next;
};
struct Node *S;
```

Trong cách cài đặt này, ngăn xếp rỗng khi S = NULL. Ta giả sử việc cấp phát bộ nhớ động cho các phần tử mới luôn thực hiện. Do đó, ngăn xếp không bao giờ đầy và phép toán PUSH luôn thực hiện thành công.

) Các hàm và thủ tục thực hiện các phép toán trên ngăn xếp:

a. Khởi tạo ngăn xếp rỗng:

```
void Create(struct Node **S)
{       *S = NULL;
}
```

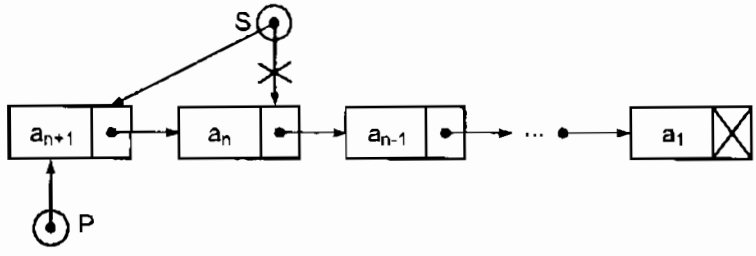
b. Kiểm tra ngăn xếp rỗng:

```
int Empty(struct Node *S)
{       return (S == NULL);
}
```

c. Bổ sung một phần tử vào đỉnh ngăn xếp:

```
void PUSH(struct Node **S, Item X)
{
        struct Node *P;
```

```
        P = new Node;
        P->Infor = X;
        P->Next = NULL;
        if (*S== NULL)   *S = P;
        else
        {       P->Next = *S; *S = P;
        }
}
```

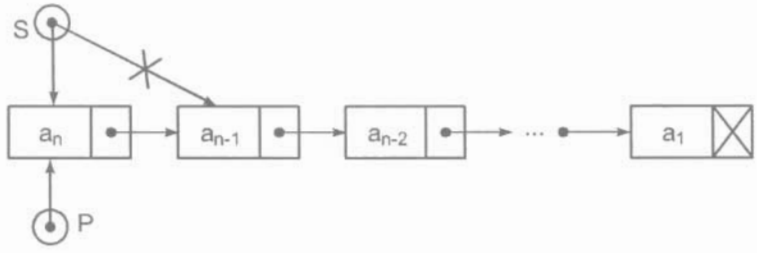


Hình 4.12: Minh hoạ thao tác PUSH

d. Lấy ra một phần tử ở đỉnh ngăn xếp:

```
int POP(struct Node **S, Item *X)
{
        struct Node *P;
        if (Empty(*S)) return 0;
        else
        {
                P = *S;
                X = (*S)->Infor;
                *S = (*S)->Next;
                delete P;
                return 1;
        }
}
```

Hàm POP trả về 1 nếu việc loại bỏ thành công, ngược lại trả về 0.

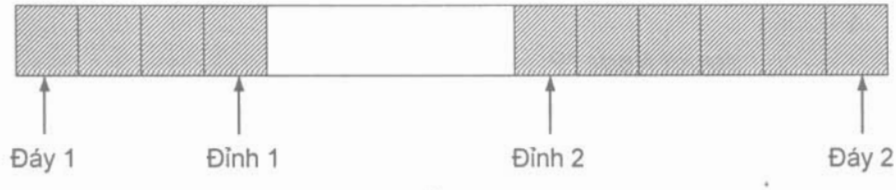


Hình 4.13: Minh hoạ thao tác POP

4.1.4. Xử lý với nhiều ngăn xếp

Có những trường hợp cùng một lúc ta phải xử lý nhiều ngăn xếp trên cùng một không gian nhớ. Như vậy, có thể xảy ra tình trạng một ngăn xếp này đã bị tràn trong khi không gian dự trữ cho ngăn xếp khác vẫn còn chỗ trống (tràn cục bộ). Làm thế nào để khắc phục được tình trạng này?

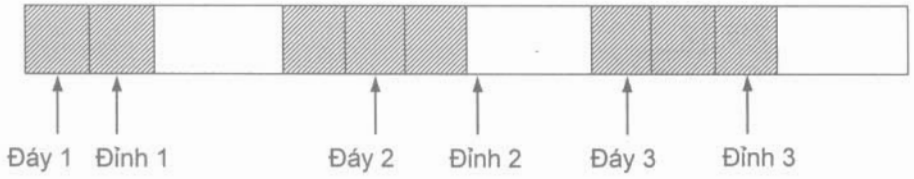
Nếu là hai ngăn xếp thì có thể giải quyết dễ dàng. Ta không qui định kích thước tối đa cho từng ngăn xếp nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta đặt hai ngăn xếp ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như hình 4.14.



Hình 4.14: Hai ngăn xếp trên một không gian nhớ

Như vậy, có thể một ngăn xếp này dùng lấn sang quá nửa không gian dự trữ nếu như ngăn xếp kia chưa dùng đến. Do đó hiện tượng tràn chỉ xảy ra khi toàn bộ không gian nhớ dành cho chúng đã được dùng hết.

Nhưng nếu số lượng ngăn xếp từ 3 trở lên thì không thể làm theo kiểu như vậy được, mà phải có giải pháp linh hoạt hơn nữa. Chẳng hạn có 3 ngăn xếp, lúc đầu không gian nhớ có thể chia đều cho cả 3, như hình 4.15.



Hình 4.15: Ba ngăn xếp trên một không gian nhớ

Nhưng nếu có một ngăn xếp nào phát triển nhanh bị tràn trước mà ngăn xếp khác vẫn còn chỗ thì phải dồn chỗ cho nó bằng cách hoặc đẩy ngăn xếp đứng sau nó sang bên phải hoặc lùi chính ngăn xếp đó sang trái trong trường hợp có thể. Như vậy thì đáy của các ngăn xếp phải được phép di động và dĩ nhiên các giải thuật bổ sung hoặc loại bỏ phần tử đối với các ngăn xếp hoạt động theo kiểu này cũng phải thay đổi.

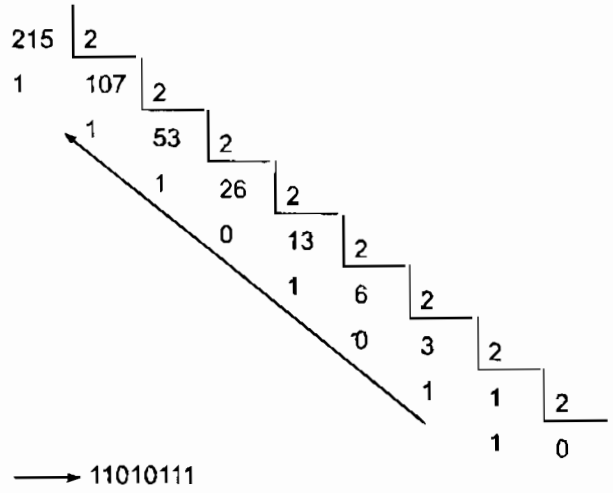
4.1.5. Một số ứng dụng của ngăn xếp

a. Ứng dụng đổi cơ số

Ta biết rằng dữ liệu lưu trữ trong bộ nhớ của máy tính đều được biểu diện dưới dạng mã nhị phân. Như vậy các số xuất hiện trong chương trình đều phải chuyển đổi từ hệ thập phân sang hệ nhị phân trước khi thực hiện các phép xử lý.

Khi đổi một số nguyên từ hệ thập phân sang hệ nhị phân người ta dùng phép chia liên tiếp cho 2 và lấy các số dư (là các chữ số nhị phân) theo chiều ngược lại.

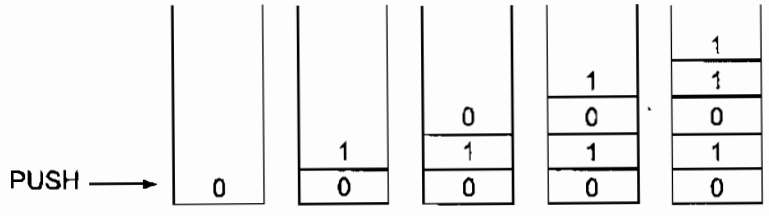
Ví dụ: Đổi số 215 sang hệ nhị phân:



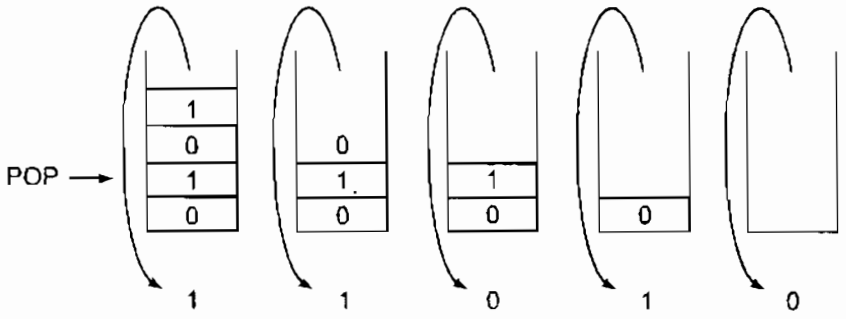
Ta thấy trong cách biến đổi này các số được tạo ra sau lại được hiển thị trước. Cơ chế sắp xếp này chính là cơ chế hoạt động của ngăn xếp. Để thực hiện biến đổi ta sẽ dùng một ngăn xếp để lưu trữ các số dư qua từng phép chia: Khi thực hiện phép chia thì nạp số dư vào ngăn xếp, sau đó lấy chúng lần lượt từ ngăn xếp ra.

Ví dụ:

Số: $(26)_{10} = (11010)_2$ trong quá trình biến đổi các số dư lần lượt sẽ là: (0 1 0 1 1).



Hình 4.16(a): Mô tả hoạt động lưu trữ dữ liệu vào ngăn xếp



Hình 4.16.(b): Mô tả hoạt động lấy dữ liệu từ ngăn xếp

Ta khai báo cấu trúc dữ liệu cho bài toán này như sau:

```
#define Max 16  //thực hiện đổi số nguyên có kích thước 2 byte
typedef int Item; //Item là chữ số nhị phân
struct    Stack
{
        Item E[Max];
        int top;
};
struct Stack S;
/*S là ngăn chứa các số dư qua các phép chia trong quá trình chuyển đổi*/
```

Giải thuật sử dụng ngăn xếp thực hiện chuyển đổi số nguyên dương N từ hệ cơ số 10 sang hệ cơ số 2. Trong giải thuật này có sử dụng các phép toán trên ngăn xếp.

```
void Chuyen_doi(N)
{
  1 - while (N != 0)
        R = N % 2;      //tính số dư trong phép chia N cho 2
        call PUSH(S, R);
        N = N/2;        //thay N bằng thương của phép chia N cho 2
  2 - cout<<"\nma nhi phan: ";
        while (!Empty(S))
```

```
        {
            call POP(S, R);
            cout<<R;
        }
}
```

b. Ứng dụng định giá biểu thức số học theo ký pháp nghịch đảo

Nhiệm vụ của bộ dịch là tạo ra các chỉ thị máy cần thiết để thực hiện các lệnh của chương trình nguồn. Một phần trong nhiệm vụ này là tạo ra các chỉ thị định giá các biểu thức số học. Chẳng hạn câu lệnh gán X = A*B + C.

Bộ dịch phải tạo ra các chỉ thị máy tương ứng như sau:

1 - LOA A: Tìm giá trị của A lưu trữ trong bộ nhớ và tải nó vào thanh ghi.

2 - MUL B: Tìm giá trị của B và nhân nó với giá trị đang ở thanh ghi.

3 - ADD C: Tìm giá trị của C và cộng nó với giá trị trong thanh ghi.

4 - STO X: Đưa giá trị trong thanh ghi vào lưu trữ ở vị trí tương ứng của X, trong bộ nhớ.

Trong các ngôn ngữ lập trình, biểu thức số học được viết như dạng thông thường của toán học nghĩa là theo kí pháp trung tố (infix notation) mỗi kí hiệu của phép toán hai ngôi được đặt giữa hai toán hạng, có thể thêm dấu ngoặc.

Chẳng hạn: 5*(7+3)

Dấu ngoặc là cần thiết vì nếu viết 5*7 + 3 thì theo qui ước về thứ tự ưu tiên của phép toán (mà các ngôn ngữ lập trình đều chấp nhận) thì biểu thức trên nghĩa là lấy 5 nhân 7 được kết quả cộng với 3.

Nhà logic học người Ba Lan Lukasiewicz đã đưa ra dạng biểu thức số học theo ký pháp hậu tố (postfix notation) và tiền tố (prefix notation) mà được gọi là dạng ký pháp Ba Lan.

Ở dạng hậu tố các toán tử đi sau các toán hạng. Như biểu thức 5*(7+3) sẽ có dạng: 5 7 3 - *

Còn ở dạng tiền tố thì các toán tử sẽ đi trước các toán hạng. Khi đó biểu thức 5*(7+3) có dạng: * 5 - 7 3

Ông cũng khẳng định rằng đối với các dạng ký pháp này dấu ngoặc là không cần thiết.

Nhiều bộ dịch khi định giá biểu thức số học thường thực hiện: trước hết chuyển các biểu thức dạng trung tố có dấu ngoặc sang dạng hậu tố, sau đó mới tạo các chỉ thị máy để định giá biểu thức ở dạng hậu tố. Việc biến đổi từ dạng trung tố sang dạng dạng hậu tố không khó khăn gì, còn việc định giá theo dạng hậu tố thì dễ dàng hơn, "máy móc" hơn so với dạng trung tố.

Để minh hoạ ta xét định giá của biểu thức sau:

1 5 + 8 4 1 - - *

tương ứng với biểu thức thông thường: (1 + 5) * (8 - (4 - 1))

Biểu thức này được đọc từ trái sang phải cho tới khi tìm ra một toán tử. Hai toán hạng được đọc cuối cùng, trước toán tử này, sẽ được kết hợp với nó. Trong ví dụ của chúng ta thì toán tử đầu tiên được đọc là + và hai toán hạng tương ứng với nó là 1 và 5, sau khi kết hợp biểu thực con này có giá trị là 6, thay vào ta có biểu thức rút gọn:

6 8 4 1 - - *

Lại đọc từ trái sang phải, toán tử tiếp theo là - và ta xác định được 2 toán hạng của nó là 4 và 1. Thực hiện phép toán ta có dạng rút gọn:

6 8 3 - *

Lại tiếp tục ta đi tới:

6 5 *

và cuối cùng thực hiện phép toán * ta có kết quả là 30.

Phương pháp định giá biểu thức hậu tố như trên đòi hỏi phải lưu trữ các toán hạng cho tới khi một toán tử được đọc, tại thời điểm này

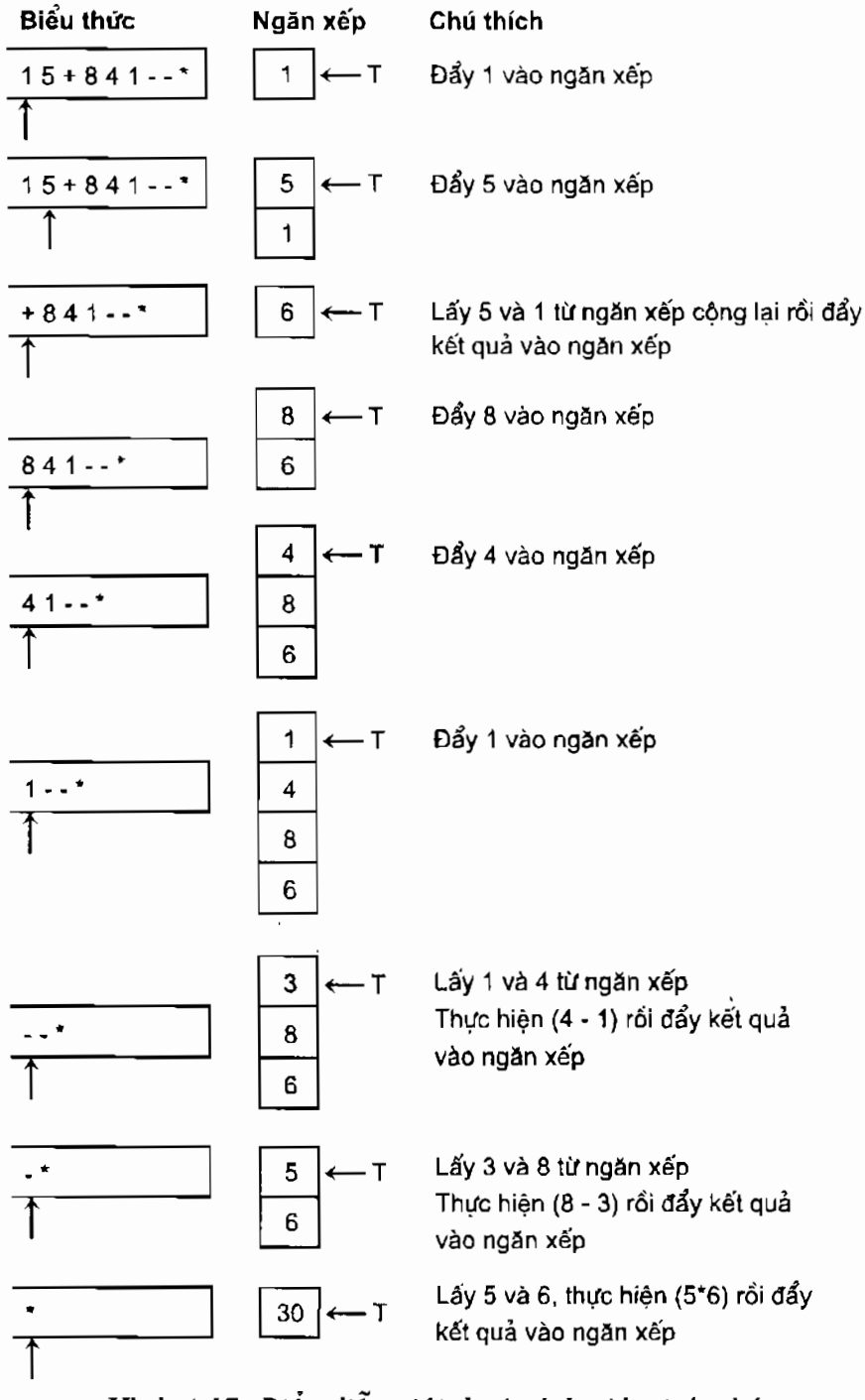
hai toán hạng cuối cùng phải được tìm ra và kết hợp với toán tử này. Như vậy ở đây đã xuất hiện cơ chế hoạt động "vào sau ra trước" nghĩa là ta sẽ phải sử dụng tới ngăn xếp để lưu trữ các toán hạng. Cứ mỗi lần đọc được một toán tử thì hai giá trị sẽ được lấy ra từ ngăn xếp để áp đặt toán tử đó lên chúng và kết quả lại được đẩy vào ngăn xếp.

Giải thuật sau đây thể hiện các ý trên.

```
void Dinh_Gia()
{
/*giải thuật này sử dụng ngăn xếp S để lưu trữ các toán hạng đọc từ biểu thức*/
  do
  {
        //Đọc phần tử X tiếp theo trong biểu thức;
        if (X là toán hạng)
              call PUSH(S, X);
              else
              {
                    call POP(S, Y);
                    call POP(S, Z);
//tác động toán tử X lên hai toán hạng Y và Z rồi gán cho biến W
              W = Y (X) Z; call PUSH(S, W);
        }
  }
  while (gặp dấu kết thúc biểu thức);
  POP(S, R);
  cout<<R;
}
```

Dưới đây là hình ảnh minh họa việc thực hiện giải thuật này với biểu thức:

1 5 + 8 4 1 - - *



Hình 4.17: Biểu diễn giải thuật tính giá trị đa thức

4.2. Queue (Hàng đợi)

4.2.1. Khái niệm

Một kiểu dữ liệu trừu tượng quan trọng khác được xây dựng trên cơ sở mô hình dữ liệu danh sách tuyến tính là hàng đợi. Hàng đợi là kiểu danh sách tuyến tính trong đó, phép bổ sung một phần tử vào hàng đợi được thực hiện ở một đầu, gọi là lối sau (rear) và phép loại bỏ một phần tử được thực hiện ở đầu kia, gọi là lối trước (front).

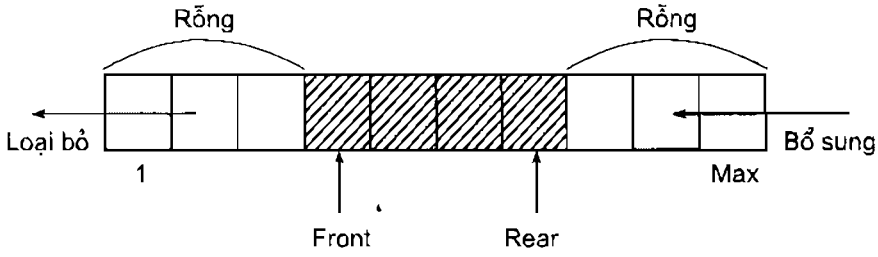
Như vậy, cơ cấu của hàng đợi là vào ở một đầu, ra ở đầu khác, phần tử vào trước thì ra trước, phần tử vào sau thì ra sau. Do đó, hàng đợi còn được gọi là danh sách kiểu ***FIFO*** (*First In First Out*). Trong thực tế ta cũng thấy có những hình ảnh giống hàng đợi, chẳng hạn, hàng người chờ mua vé tàu, học sinh xếp hàng đi vào lớp, v.v...

4.2.2. Cài đặt hàng đợi bởi mảng

Ta có thể biểu diễn hàng đợi bởi mảng với việc sử dụng hai chỉ số front để chỉ vị trí đầu hàng đợi (lối trước) và rear để chỉ vị trí cuối hàng đợi (lối sau). Cấu trúc dữ liệu hàng đợi được biểu diễn như sau:

```
#define   max N
//Khai báo kiểu dữ liệu Item nếu cần
struct Queue
{
        unsigned int front, rear;
        Item E[max];
};
struct Queue Q;
//Khai báo hàng đợi Q lưu trữ các phần tử của danh sách
```

Trong cách cài đặt như trên, hàng đợi Q là rỗng nếu Q.rear = 0 và hàng đầy nếu Q.rear = Max.



Hình 4.18: Mảng biểu diễn hàng đợi

**** Các phép toán trên hàng đợi***

Giống như ngăn xếp, khi thực hiện các thao tác với hàng đợi ta không được phép truy nhập tùy tiện vào các phần tử của hàng đợi, mà phải sử dụng các phép toán đặc biệt được định nghĩa trên hàng đợi. Đó là các phép toán sau:

a. Khởi tạo hàng đợi rỗng

```
void Initialize(struct Queue *Q)
{       Q->front = 1; Q->rear = 0;
}
```

b. Kiểm tra hàng đợi rỗng

```
int Empty(struct Queue Q)
{
        return (Q.rear ==0);
}
```

c. Kiểm tra hàng đợi đầy

```
int Full(struct Queue Q)
{
        return (Q.rear == max);
}
```

d. Bổ sung một phần tử mới vào đầu hàng đợi

Khi bổ sung một phần tử mới (với thông tin lưu trong biến X) vào hàng đợi cần kiểm tra xem hàng có đầy không, nếu hàng chưa đầy

thì bổ sung phần tử mới vào hàng, ngược lại việc bổ sung không được thực hiện.

```
int AddQ(struct Queue *Q, Item X)
{
        if (Full(*Q)) return 0;
        else
        {
                Q->rear = Q->rear + 1;
                Q->E[Q->rear] = X;
                return 1;
        }
}
```

Hàm AddQ thực hiện bổ sung phần tử mới vào cuối hàng đợi, hàm trả về 1 nếu bổ sung thành công, và trả về 0 nếu ngược lại.

e. Loại bỏ một phần tử ra khỏi hàng đợi

Khi loại bỏ một phần tử cần phải kiểm tra xem hàng đợi có rỗng không, nếu hàng đợi rỗng thì không thể thực hiện việc loại bỏ, ngược lại thì loại bỏ phần tử ở đầu hàng, nội dung của phần tử này được lưu trong biến X. Thêm nữa, khi loại bỏ, nếu hàng chỉ có một phần tử (nghĩa là hàng sẽ rỗng sau khi loại bỏ) thì cần khởi tạo lại hàng.

Sau đây là thủ tục thực hiện việc loại bỏ.

```
int DeleteQ(struct Queue *Q, Item X)
{
  if (Empty(*Q))
        return 0;
  else
  {
        X = Q->E[Q->front];
        if (Q->front == Q->rear)
        {
                Q->front = 1; Q->rear = 0;      //khởi tạo lại hàng đợi
        }
```

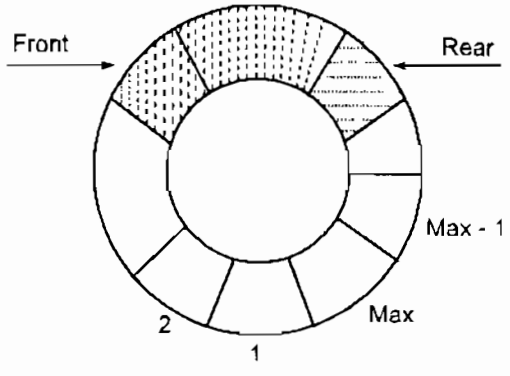
```
            else    Q->front = Q->front + 1;
            return 1;
    }
}
```

Hàm DeleteQ thực hiện lấy ra một phần tử ở đầu hàng đợi, hàm trả về 1 nếu phép lấy ra thành công, ngược lại trả về 0.

Nhận xét: Phương pháp cài đặt hàng đợi bởi mảng với hai chỉ số như trên có nhược điểm lớn. Nếu phép loại bỏ không thường xuyên làm cho hàng rỗng, thì các chỉ số front và rear sẽ tăng liên tục và sẽ vượt quá kích thước của mảng. Hàng sẽ trở thành đầy, mặc dù các vị trí trống trong mảng có thể vẫn còn nhiều (do việc loại bỏ các phần tử ở đầu hàng). Để tránh tình trạng này, mỗi khi hàng đợi đầy ta lại kiểm tra không gian nhớ phía trước hàng đợi, nếu còn trống thì đẩy hàng đợi về phía trước để tạo ra không gian nhớ trống ở phía sau. Tuy nhiên việc này tiêu tốn rất nhiều thời gian.

4.2.3. Cài đặt hàng đợi bởi mảng vòng tròn

Để khắc phục nhược điểm nêu trên, người ta đưa ra phương pháp cài đặt hàng đợi bởi mảng vòng tròn. Đó là một mảng với chỉ số chạy trong miền 1..max, với mọi i = 1, 2,..., max – 1, phần tử thứ i của mảng đi trước phần tử thứ i + 1, còn phần tử thứ max đi trước phần tử đầu tiên, tức là các phần tử của mảng được xếp thành vòng tròn (xem hình 4.20).



Hình 4.20: Hàng đợi vòng tròn

Khi biểu diễn hàng bởi mảng vòng tròn, để biết khi nào hàng đầy, khi nào hàng rỗng ta cần đưa thêm vào biến count để đếm số phần tử trong hàng. Chúng ta có khai báo cấu trúc dữ liệu sau:

```
#define   max N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct    Queue
{
        int count;
        int front, rear;
        Item E[max];
};
struct Queue Q;
```

Với cách cài đặt này, hàng rỗng khi Q.count = 0, hàng đầy khi Q.count = max.

Khi làm việc với mảng vòng tròn, cần lưu ý rằng, phần tử đầu tiên của mảng đi sau phần tử thứ max. Sau đây chúng ta sẽ cài đặt thao tác bổ sung một phần tử vào hàng đợi, và loại bỏ một phần tử khỏi hàng đợi, các thao tác khác dành cho bạn đọc.

```
void AddQ(struct Queue *Q, Item X)
{
        if (Q->count == max) cout<<"\nHang day"<<endl;
        else
        {
                if (Q->rear == max)
                        Q->rear = 1;
                else
                        Q->rear = Q->rear + 1;
                        Q->E[Q->rear] = X;
                        Q->count = Q->count + 1;
        }
}
```

```
void DeleteQ(struct Queue *Q, Item X)
{
        if (Q->count = = 0) cout<<"\nhang rong"<<endl;
        else {
                X = Q->E[Q->front];
                if (Q->front = = Q->rear)
                {       Q->front = 1; Q->rear = 0;
                }
                else
                        if (Q->front = = max) Q->front = 1;
                        else Q->front = Q->front + 1;
                Q->count = Q->count – 1;
        }
}
```

Hàng đợi thường dùng để thực hiện các "tuyến chờ" (waiting lines) trong xử lý động, đặc biệt trong các hệ mô phỏng (simulation), đó là các hệ mô hình hoá các quá trình động và người ta dùng mô hình này để nghiên cứu hoạt động của các quá trình ấy.

4.2.4. Cài đặt hàng đợi bởi danh sách móc nối đơn

Như ta đã biết, đối với ngăn xếp việc truy nhập chỉ được thực hiện ở một đầu (đỉnh). Vì vậy, việc cài đặt ngăn xếp bằng danh sách móc nối là khá tự nhiên. Chẳng hạn, với danh sách móc nối đơn trỏ bởi L thì có thể coi L như con trỏ trỏ tới đỉnh của ngăn xếp. Bổ sung một nút vào ngăn xếp chính là việc bổ sung một nút vào thành nút đầu tiên của danh sách, còn loại bỏ một nút ra khỏi ngăn xếp chính là loại bỏ nút đầu tiên của danh sách đang trỏ bởi L. Trong việc bổ sung với ngăn xếp dạng này không cần kiểm tra hiện tượng tràn như với ngăn xếp lưu trữ kế tiếp.

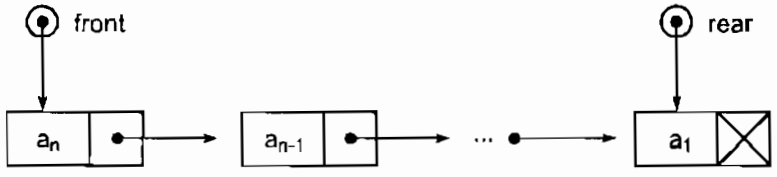
Đối với hàng đợi thì loại bỏ ở một đầu, còn bổ sung thì ở đầu kia. Nếu coi danh sách móc nối đơn như một hàng đợi thì việc loại bỏ một nút cũng giống như với ngăn xếp, nhưng bổ sung một nút thì phải thực gắn nút mới vào cuối hàng đợi, nghĩa là phải tìm đến nút cuối

cùng. Trong trường hợp này, để lưu trữ danh sách người ta dùng hai con trỏ, một con trỏ trỏ vào nút đầu danh sách và một con trỏ trỏ vào nút cuối danh sách.

Cấu trúc dữ liệu biểu diễn hàng đợi như sau:

```
//Khai báo kiểu dữ liệu Item (nếu cần)
struct   Node
{        Item Infor;
         struct Node *Next;
};
struct   Queue
{
         struct Node *front;
         struct Node *rear;
};
struct Queue Q;
```

Trong cách biểu diễn trên, front là con trỏ trỏ đến nút đầu hàng đợi, rear là con trỏ trỏ đến nút cuối hàng đợi.



Hình 4.21: Danh sách móc nối biểu diễn hàng đợi

Với cách cài đặt này, hàng đợi được xem là không khi nào đầy. Hàng đợi rỗng khi Q.front = NULL.

Sau đây là các hàm và thủ tục thực hiện các phép toán trên hàng đợi.

a. Khởi tạo hàng đợi rỗng:

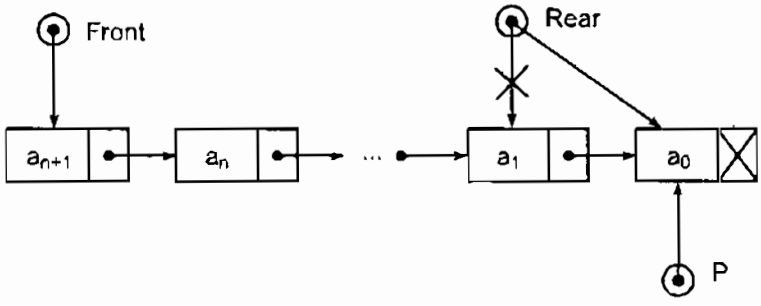
```
void Create(struct Queue *Q)
{
         Q->front = NULL;
         Q->rear = NULL;
}
```

b. Kiểm tra hàng đợi rỗng:

```
int Empty(struct Queue Q)
{
        return (Q.front = = NULL);
}
```

c. Bổ sung một phần tử vào cuối hàng đợi:

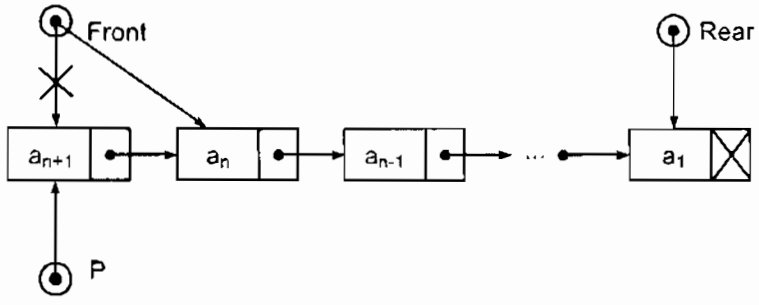
```
void ADD(struct Queue *Q, Item X)
{
  struct Node *P;
        P = (struct Node*)malloc(sizeof(struct Node));
        P->Infor = X;
        P->Next = NULL;
        if (Empty(*Q))
        {       Q->front = P;
                Q->rear = P;
        }
        else    {
                Q->rear->Next = P;
                Q->rear = P;
                }
}
```



Hình 4.22: Minh họa thao tác bổ sung

d. Lấy ra một phần tử ở đầu hàng đợi

```
int Del(struct Queue *Q, Item X)
{
  struct Node P;
  if (Empty(*Q)) return 0;
  else    {
                P = Q->front;
                X = Q->front->Infor;
                Q->front = Q->front->Next;
                free(P);
                return 1;
          }
}
```

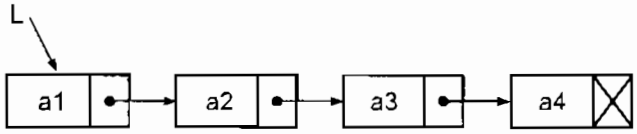


Hình 4.23: Minh họa thao tác lấy ra một phần tử

Trên đây chúng ta đã xem xét một loại cấu trúc dữ liệu, được sử dụng rất phổ biến trong các ứng dụng, đó là danh sách tuyến tính với các dạng khác nhau được cài đặt theo hai cách: bằng mảng (lưu trữ kế tiếp) và bằng con trỏ (lưu trữ móc nối), cùng với các phép toán xử lý tương ứng trên mỗi loại. Bạn đọc cũng được tìm hiểu hai cách thức lưu trữ đặc biệt đó là danh sách kiểu ngăn xếp và kiểu hàng đợi. Mỗi loại danh sách và cách cài đặt chúng có những ưu và nhược điểm khac nhau. Tuy nhiên, để hiểu rõ hơn về nó, ta cần cài đặt một số ứng dụng nhỏ trên máy đối với mỗi loại. Khi cài đặt hãy lựa chọn cấu trúc lưu trữ phù hợp với dữ liệu của bài toán.

BÀI TẬP CHƯƠNG 4

Bài 1: Cho danh sách nối đơn với nút đầu tiên được trỏ bởi con trỏ L như hình vẽ:



Yêu cầu:

a. Vẽ hình mô tả trạng thái của danh sách trong quá trình tạo mới danh sách gồm 4 phần tử như trên, xuất phát từ một danh sách rỗng (yêu cầu mô tả từng bước trong quá trình).

b. Vẽ hình mô thao tác loại bỏ phần tử đầu tiên (a1) và phần tử thứ 3 (a3) trong danh sách (cần chú thích rõ ràng).

c. Vẽ hình mô tả thao tác bổ sung phần tử 'a5' vào đầu danh sách và vào sau phần tử thứ 3 (a3) trong danh sách.

d. Giả sử a3 là nhỏ nhất, vẽ hình mô tả tình trạng của danh sách trong thao tác sắp xếp tăng dần bằng phương pháp lựa chọn ở lần duyệt đầu tiên (chỉ vẽ trạng thái lúc bắt đầu duyệt, lúc chuẩn bị đổi chỗ và sau khi đổi chỗ).

e. Giả sử danh sách lưu trữ thông tin về các sinh viên, mỗi sinh viên gồm: Mã sinh viên, họ và tên, năm sinh, điểm tổng kết. Cài đặt chương trình thực hiện các yêu cầu sau:

- Khai báo cấu trúc dữ liệu của danh sách.
- Tạo mới danh sách gồm 10 phần tử (các thông tin được nhập từ bàn phím). Hoặc tạo mới danh sách với các thông tin phần tử nhập từ bàn phím, việc nhập kết thúc khi tên của sinh viên nhập vào là xâu rỗng.
- Hiển thị danh sách lên màn hình.
- Xóa phần tử đầu tiên trong danh sách, hiển thị lại danh sách
- Xác định tỷ lệ sinh viên giỏi (dtk >= 8), khá (dtk >= 6.5), trung bình (dtk >= 5), yếu (còn lại).

- Xóa phần tử thứ 4 trong danh sách, hiển thị lại danh sách.
- Thêm một phần tử vào đầu danh sách, hiển thị lại danh sách
- Thêm một phần tử vào sau phần tử thứ 3 trong danh sách, hiển thị lại danh sách.
- Tìm sinh viên có tên "Doanh" trong danh sách, hiển thị kết quả tìm kiếm (nếu có hiển thị thông tin đầy đủ của sinh viên này).
- Sắp xếp danh sách theo chiều giảm dần của điểm tổng kết, hiển thị lại danh sách.
- Thêm thông tin một học sinh mới vào danh sách sao cho trật tự vừa sắp không bị thay đổi.

Bài 2: Thực hiện lại các yêu cầu của bài 1 với danh sách nối đôi.

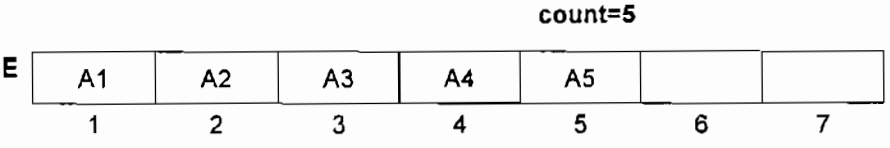
Bài 3: Cài đặt chương trình thực hiện các yêu cầu:

- Tạo một danh sách nối đơn/nối đôi lưu trữ các số nguyên, dữ liệu được nhập từ bàn phím, việc nhập kết thúc khi số nguyên nhập vào là -1 (lưu ý -1 không phải là một phần tử của danh sách) hoặc cho phép nhập n số với n nhập từ bàn phím.
- Hiển thị danh sách lên màn hình.
- Bổ sung số nguyên X vào vị trí K trong danh sách (X, K nhập từ bàn phím), hiển thị lại danh sách.
- Xóa số thứ k trong danh sách (k nhập từ bàn phím), hiển thị lại danh sách.
- Xóa các số âm trong danh sách, hiển thị lại danh sách.
- Sắp xếp danh sách theo chiều tăng dần/giảm dần, hiển thị lại danh sách.
- Nhập một số nguyên, bổ sung nó vào danh sách sao cho danh sách vẫn có thứ tự, hiển thị lại danh sách.
- Cho biết chiều dài của danh sách, số lượng số âm, số lượng số dương.

Bài 4: Cài đặt chương trình thực hiện các yêu cầu sau:

- Tạo một danh sách nối đơn/nối đôi lưu trữ các phân số có tử số và mẫu số là các số nguyên khác không, dữ liệu được nhập từ bàn phím, việc nhập kết thúc nếu gặp phân số mà tử số hoặc mẫu số của nó được nhập là số không.
- Hiển thị danh sách lên màn hình (ví dụ: 3/4, -2/5, 1/2,...)
- Cho biết trong danh sách trên có bao nhiêu phân số chưa được tối giản, hãy tối giản các phân số đó, hiển thị lại danh sách.
- Bổ sung một phân số mới vào vị trí thứ k trong danh sách (phân số mới và k được nhập từ bàn phím), hiển thị lại danh sách.
- Tính tổng các phân số có cùng mẫu số là 9 trong danh sách, hiển thị kết quả sau khi đã tối giản.
- Xóa các phân số có tử số là số âm, hiển thị lại danh sách.

Bài 5: Danh sách L được lưu trữ kế tiếp biểu diễn bởi hình vẽ (E: mảng lưu các phần tử của danh sách, count biến nguyên lưu độ dài thực của danh sách.



Yêu cầu:

a. Vẽ hình mô tả trạng thái của danh sách qua các bước trong quá trình tạo mới danh sách từ một danh sách rỗng.

b. Vẽ hình mô tả trạng thái của danh sách trong thao tác loại bỏ phần tử đầu tiên (A1) và phần tử thứ 3 (A3) trong danh sách (cần chú thích rõ ràng).

c. Vẽ hình mô tả quá trình bổ sung phần tử A6 vào đầu danh sách, vào sau phần tử thứ 3 (A3) trong danh sách.

d. Giả sử danh sách lưu trữ thông tin về các sinh viên, mỗi sinh viên gồm: Mã sinh viên, họ và tên, năm sinh, điểm tổng kết. Hãy cài đặt chương trình thực hiện các yêu cầu sau:
   - Khai báo cấu trúc dữ liệu của danh sách
   - Nhập mới 10 phần tử cho danh sách
   - Hiển thị danh sách lên màn hình
   - Xóa phần tử đầu tiên trong danh sách, hiển thị lại danh sách
   - Xóa phần tử thứ 4 trong danh sách, hiển thị lại danh sách
   - Thêm một phần tử vào đầu danh sách, hiển thị lại danh sách
   - Thêm một phần tử vào sau phần tử thứ 3 trong danh sách, hiển thị danh sách
   - Tìm sinh viên có tên "Doanh" trong danh sách, hiển thị kết quả tìm kiếm.
   - Sắp xếp danh sách theo chiều giảm dần của điểm tổng kết, hiển thị lại danh sách.

Bài 6: Cài đặt chương trình thực hiện các yêu cầu:

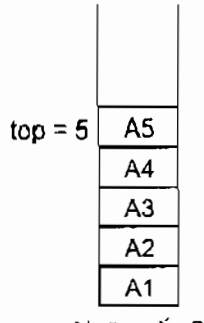
- Tạo một danh sách lưu trữ kế tiếp các số nguyên, dữ liệu được nhập từ bàn phím, việc nhập kết thúc khi số nguyên nhập vào là -1.
- Hiển thị danh sách lên màn hình.
- Bổ sung số nguyên X vào vị trí K trong danh sách (X, K nhập từ bàn phím), hiển thị lại danh sách.
- Xóa số thứ I trong danh sách (I nhập từ bàn phím), hiển thị lại danh sách.
- Xóa các số âm trong danh sách, hiển thị lại danh sách.
- Sắp xếp danh sách theo chiều tăng dần/giảm dần, hiển thị lại danh sách.
- Nhập một số nguyên, bổ sung nó vào danh sách sao cho danh sách vẫn có thứ tự, hiển thị lại danh sách.

Bài 7: Cài đặt chương trình thực hiện các yêu cầu sau:

- Tạo một danh sách lưu trữ kế tiếp các phân số có tử số và mẫu số là các số nguyên khác không, dữ liệu được nhập từ bàn phím, việc nhập kết thúc nếu gặp phân số mà tử số hoặc mẫu số của nó được nhập là số không.
- Hiển thị danh sách lên màn hình (ví dụ: 3/4, -2/5, 1/2,...).
- Cho biết trong danh sách trên có bao nhiêu phân số chưa được tối giản, hãy tối giản các phân số đó, hiển thị lại danh sách.
- Bổ sung một phân số mới vào vị trí thứ k trong danh sách (phân số mới và k được nhập từ bàn phím), hiển thị lại danh sách.
- Tính tổng các phân số có cùng mẫu số là 9 trong danh sách, hiển thị kết quả sau khi đã tối giản.
- Xóa các phân số có tử số là số âm, hiển thị lại danh sách

Bài 8: Cho ngăn xếp S được biểu diễn bởi hình vẽ.

- Vẽ hình mô tả trạng thái của ngăn xếp qua các bước, khi thực hiện thao tác loại bỏ phần tử A2 trong ngăn xếp.
- Vẽ hình mô tả trạng thái của ngăn xếp qua các bước khi thực hiện thao tác bổ sung phần tử A6 vào đáy ngăn xếp.
- Giả sử ngăn xếp được lưu trữ bởi một danh sách nối đơn/lưu trữ kế tiếp với mỗi phần tử là một số nguyên.



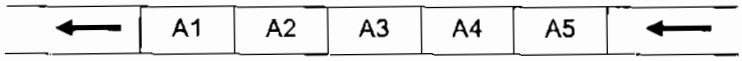
Ngăn xếp S

Yêu cầu:

- Cài đặt cấu trúc dữ liệu của ngăn xếp cùng với các phép toán cơ sở của nó.
- Cài đặt phép toán loại bỏ phần tử thứ 2 trong ngăn xếp tính từ đáy.

- Cài đặt phép toán bổ sung 1 phần tử vào đáy ngăn xếp
- Cài đặt phép toán loại bỏ phần tử thứ k trong ngăn xếp.

Bài 9: Cho hàng đợi Q được biểu diễn bởi hình vẽ sau:



- Vẽ hình mô tả trạng thái của hàng đợi khi thực hiện thao tác bổ sung phần tử A6 vào đầu hàng đợi, và bổ sung phần tử A7 vào sau phần tử A3
- Vẽ hình mô tả trạng thái của hàng đợi khi thực hiện thao tác loại bỏ phần tử A3 và phần tử A5 trong hàng đợi.
- Giả sử hàng đợi được lưu trữ kế tiếp/lưu trữ bởi một danh sách nối đơn, mỗi phần tử chứa một số nguyên.

Yêu cầu:

- Cài đặt cấu trúc dữ liệu của hàng đợi cùng với các phép toán cơ sở của nó
- Cài đặt phép toán bổ sung một phần tử vào đầu hàng
- Cài đặt phép toán bổ sung một phần tử vào sau phần tử thứ 3 tính từ đầu hàng
- Cài đặt phép toán loại bỏ phần tử thứ k trong hàng.

Chương 5

CÂY

Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu cây (tree). Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục hoặc mục lục của cuốn sách cũng là một cây. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn nó được áp dụng để tổ chức thông tin trong các hệ cơ sở dữ liệu, để mô tả cấu trúc cú pháp của các chương trình nguồn khi xây dựng các chương trình dịch. Rất nhiều bài toán mà ta gặp trong các lĩnh vực khác nhau được quy về việc thực hiện các phép toán trên cây. Trong chương 4 chúng ta sẽ trình bày định nghĩa, các khái niệm cơ bản về cây. Chúng ta cũng sẽ xét các phương pháp cài đặt cây và thực hiện các phép toán cơ bản trên cây. Sau đó ta nghiên cứu kỹ một số dạng cây đặc biệt đó là cây nhị phân tìm kiếm và cây cân bằng.

1. CÂY VÀ CÁC KHÁI NIỆM LIÊN QUAN

1.1. Định nghĩa

Định nghĩa 1: Một cây là tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (root). Giữa các nút có mối quan hệ phân cấp gọi là quan hệ "cha-con".

Định nghĩa 2: Cây được định nghĩa đệ quy như sau:

- Một nút là một cây và nút này cũng là gốc của cây.

- Giả sử T_1, T_2,..., T_n ($n \geq 1$) là các cây có gốc tương ứng r_1, r_2,..., r_n. Khi đó cây T với gốc r được hình thành bằng cách cho r trở thành nút cha của các nút r_1, r_2,..., r_n

1.2. Một số khái niệm cơ bản

- *Bậc của một nút*: là số con của nút đó

- *Bậc của một cây*: là bậc của nút có bậc lớn nhất trên cây đó (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n - phân

- *Nút gốc*: là nút có không có nút cha

- *Nút lá*: là nút có bậc bằng 0

- *Nút nhánh*: là nút có bậc khác 0 và không phải là nút gốc

- *Mức của một nút*

Mức (gốc (T_0)) =1

Gọi T_1, T_2,..., T_n là các cây con của T_0.

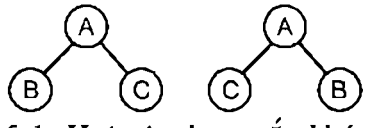
Khi đó Mức (T1) = Mức (T2) = ... = Mức (Tn) = Mức (T0) +1

- *Chiều cao của cây*: là mức của nút có mức lớn nhất có trên cây đó

- *Đường đi*: Dãy các đỉnh n_1, n_2,..., n_k được gọi là đường đi nếu n_i là cha của n_{i+1} ($1 \le i \le k-1$)

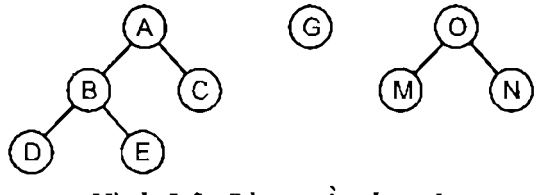
- *Độ dài của đường đi*: là số nút trên đường đi -1

- *Cây được sắp thứ tự*: Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ tự nhất định, thì cây được gọi là cây được sắp (cây có thứ tự). Chẳng hạn, hình 5.1 minh họa hai cây được sắp khác nhau.



Hình 5.1: Hai cây được sắp khác nhau

- *Rừng*: là tập hợp hữu hạn các cây phân biệt.



Hình 5.2: Rừng gồm ba cây

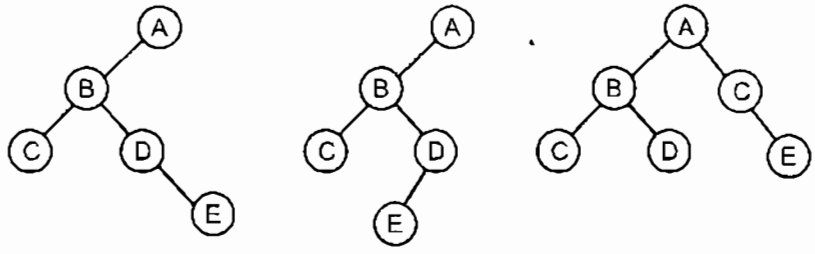
Sau đây ta sẽ tìm hiểu một loại cây đặc biệt được gọi là cây nhị phân.

2. CÂY NHỊ PHÂN

2.1. Định nghĩa

Cây nhị phân là cây mà mỗi nút có tối đa hai cây con. Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải.

Như vậy cây nhị phân là cây có thứ tự.



Hình 5.3: Một số dạng cây nhị phân

2.2. Tính chất

Đối với cây nhị phân cần chú ý tới một số tính chất sau:

1. Số lượng tối đa các nút có ở mức i trên cây nhị phân là 2^{i-1} ($i \geq 1$)

2. Số lượng nút tối đa trên một cây nhị phân có chiều cao h là 2^h-1 ($h \geq 1$).

**** Chứng minh***

+ Tính chất 1 sẽ được chứng minh bằng qui nạp

Bước cơ sở: với i = 1, cây nhị phân có $1 = 2^0$ nút. Vậy mệnh đề đúng với i = 1.

Bước qui nạp: Giả sử kết quả đúng với mức i, nghĩa là ở mức này cây nhị phân có tối đa 2^{i-1} nút, ta chứng minh mệnh đề đúng với mức i + 1.

Theo định nghĩa cây nhị phân thì tại mỗi nút có tối đa hai cây con nên mỗi nút ở mức i có tối đa hai con. Do đó theo giả thiết qui nạp ta suy ra tại mức i + 1 ta có:

$$2^{i-1} \times 2 = 2^i \text{ nút.}$$

+ Tính chất 2 được chứng minh như sau:

Ta đã biết rằng chiều cao của cây là số mức lớn nhất có trên cây đó. Theo tính chất 1 ta suy ra số nút tối đa có trên cây nhị phân với chiều cao h là:

$$2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1.$$

Từ kết quả này có thể suy ra:

Nếu cây nhị phân có n nút thì chiều cao của nó là $h = \lceil \log_2(n+1) \rceil$

(Ta qui ước: $\lceil x \rceil$ là số nguyên trên của x

$\lfloor x \rfloor$ là số nguyên dưới của x).

2.3. Biểu diễn cây nhị phân

2.3.1. Lưu trữ kế tiếp

Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

Ta có thể sử dụng một mảng để lưu trữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường:

Infor: mô tả thông tin gắn với mỗi đỉnh

Left: chỉ đỉnh con trái

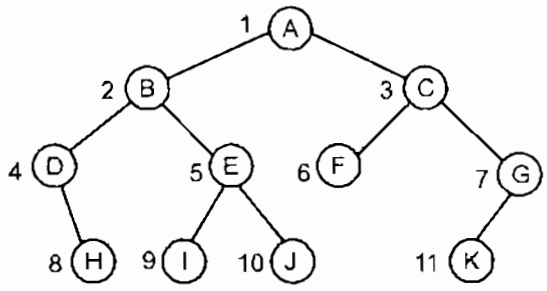
Right: chỉ đỉnh con phải.

Giả sử các đỉnh của cây được được đánh số từ 1 đến max, dữ liệu của các đỉnh trên cây có kiểu là ***Item***. Khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau:

```
#define max = N;          //Số thứ tự lớn nhất của nút trên cây
//Khai báo kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item Infor;
        int Letf;
```

```
        int Right;
};
struct Node T[max];      //T là mảng lưu trữ các nút của cây.
```

Ví dụ:



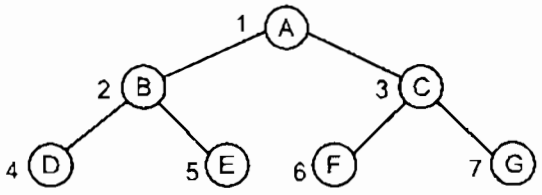
Hình 5.4: Một cây nhị phân

Hình 5.5 minh hoạ cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 5.4.

	infor	Left	Right
1	A	2	3
2	B	4	5
3	C	6	7
4	D	0	8
5	E	9	10
6	F	0	0
7	G	11	9
8	H	0	0
9	I	0	0
10	J	0	0
11	K	0	0

Hình 5.5: Cấu trúc dữ liệu biểu diễn cây

Nếu có một cây nhị phân hoàn chỉnh đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và từ trái qua phải đối với các nút ở mỗi mức. Ví dụ hình 5.6 minh họa cây nhị phân được đánh số.



Hình 5.6: Cây nhị phân được đánh số

Ta có nhận xét sau: con của nút thứ i là các nút thứ 2i và 2i + 1 hoặc cha của nút thứ j là $\lfloor j/2 \rfloor$.

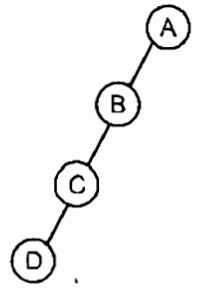
Như vậy, ta có thể lưu trữ cây này bằng một vectơ V, theo nguyên tắc: nút thứ i của cây được lưu trữ ở V[i]. Đó chính là cách lưu trữ kế tiếp đối với cây nhị phân. Với cách lưu trữ này nếu biết được địa chỉ của nút con sẽ tính được địa chỉ nút cha và ngược lại.

Với cây đầy đủ nêu trên thì hình ảnh lưu trữ sẽ như sau:

A	B	C	D	E	F	G
v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]

Nhận xét:

Nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp vì sẽ gây ra lãng phí bộ nhớ do có nhiều phần tử bỏ trống (ứng với cây con rỗng). Ta hãy xét cây như hình 5.6. Để lưu trữ cây này ta phải dùng mảng gồm 15 phần tử mà chỉ có 5 phần tử khác rỗng, hình ảnh lưu trữ miền nhớ của cây này (hình 5.7).



Hình 5.7: Cây nhị phân đặc biệt

	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	...

(∅: chỉ chỗ trống)

Nếu cây nhị phân luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động thì cách lưu trữ này gặp phải một số nhược điểm như tốn thời gian khi phải thực hiện các thao tác này, độ cao của cây phụ thuộc vào kích thước của mảng,...

2.3.2. Lưu trữ móc nối

Cách lưu trữ móc nối khắc phục được những nhược điểm của cách lưu trữ kế tiếp đồng thời phản ánh được dạng tự nhiên của cây.

Trong cách lưu trữ móc nối, mỗi nút tương ứng với một phần tử nhớ có qui cách như sau:

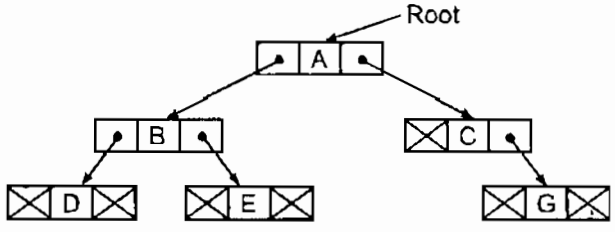
Letf	Infor	Right

- Trường Infor ứng với thông tin (dữ liệu) của nút
- Trường Left ứng với con trỏ, trỏ tới cây con trái của nút đó
- Trường Right ứng với con trỏ, trỏ tới cây con phải của nút đó

Ta có thể khai báo cấu trúc dữ liệu như sau:

```
struct Node
 {      Item infor;
        struct Node *Left, *Right;
 };
struct Node *Root; //Con trỏ Root trỏ vào nút gốc cây
```

Ví dụ: cây nhị phân hình 5.6 có dạng lưu trữ móc nối như hình 5.8.



Hình 5.8: Cấu trúc dữ liệu biểu diễn cây

Để lưu trữ và thao tác với cây, cần một con trỏ Root, trỏ tới nút gốc của cây.

2.4. Phép duyệt cây nhị phân

Phép xử lý các nút trên cây - mà ta gọi chung là phép "thăm" các nút một cách hệ thống, sao cho mỗi nút được thăm đúng một lần theo một thứ tự xác định, gọi là phép duyệt cây. Có thể duyệt cây nhị phân

theo một trong ba thứ tự: duyệt trước, duyệt giữa và duyệt sau, các phép duyệt này được định nghĩa đệ quy như sau:

2.4.1. Duyệt theo thứ tự trước (preorder traversal)

Nếu cây khác rỗng

- Thăm gốc
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước.

2.4.2. Duyệt theo thứ tự giữa (inorder traversal)

Nếu cây khác rỗng

- Duyệt cây con trái theo thứ tự giữa
- Thăm gốc
- Duyệt cây con phải theo thư tự giữa.

2.4.3. Duyệt theo thứ tự sau (postorder traversal)

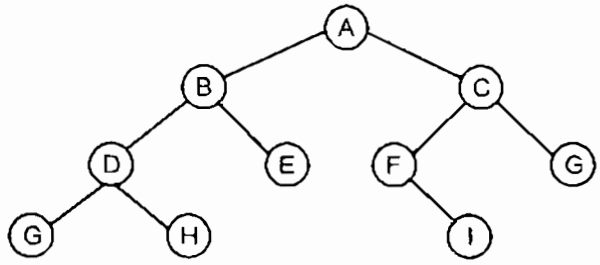
Nếu cây khác rỗng

- Duyệt cây con trái theo thứ tự sau
- Duyệt cây con phải theo thư tự sau
- Thăm gốc.

Tương ứng với ba phép duyệt ta có ba thủ tục duyệt cây nhị phân. Sau đây là thủ tục đệ quy duyệt cây theo thứ tự trước:

```
void PreOrder (struct Node *Root)
{
        if (Root != NULL)
        {
                visit(Root);
                PreOrder(Root->Left);
                PreOrder(Root->Right);
        }
}
```

Một cách tương tự, ta có thể viết được các thủ tục đệ quy đi qua cây theo thứ tự giữa và theo thứ tự sau.



Hình 5.9: Duyệt cây nhị phân

Với cây nhị phân hình 5.9, dãy các nút được thăm trong các phép duyệt là:

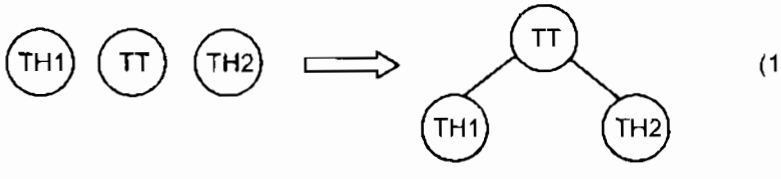
- *Duyệt theo thứ tự trước:* A B D G H E C F I G
- *Duyệt theo thứ giữa:* G D H B E A F I C G
- *Duyệt theo thứ tự sau:* G H D E B I F G C A

2.5. Cây nhị phân biểu diễn biểu thức

Cây biểu thức là cây nhị phân mà nút gốc và các nút nhánh chứa các toán tử (phép toán) còn các nút lá thì chứa các toán hạng.

2.5.1. Cách dựng cây biểu thức

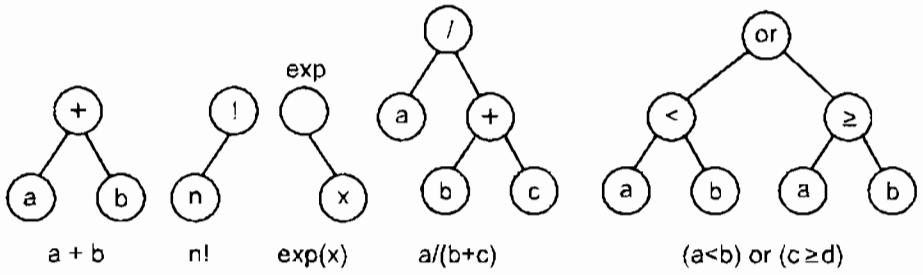
Đối với phép toán hai ngôi (chẳng hạn +, -, *, /) được biểu diễn bởi cây nhị phân mà gốc của nó chứa toán tử, cây con trái biểu diễn toán hạng bên trái, còn cây con bên phải biểu diễn toán hạng bên phải.



Hình 5.10: Biểu diễn phép toán hai ngôi

Đối với phép toán một ngôi như "phủ định" hoặc "phép toán đổi dấu", hoặc các hàm chuẩn như exp() hoặc cos()... thì cây con bên trái

rỗng. Còn với các phép toán một toán hạng như phép "lấy đạo hàm" ()' hoặc hàm "giai thừa" ()! thì cây con bên phải rỗng.



Hình 5.11: Một số cây biểu thức

Nhận xét:

Nếu ta đuyệt cây biểu thức theo thứ tự trước thì ta được biểu thức Ba Lan dạng tiền tố (pre-fix). Nếu duyệt cây nhị phân theo thứ tự sau thì ta có biểu thức Ba Lan dạng hậu tố (post-fix); còn theo thứ giữa thì ta nhận được cách viết thông thường của biểu thức (dạng trung tố).

2.5.2. Ví dụ

Để minh cho nhận xét này ta lấy ví dụ sau:

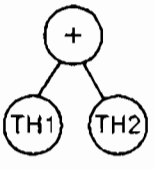
Cho biểu thức P = a*(b - c) + d/e

1. Hãy dựng cây biểu thức biểu diễn biểu thức trên

2. Đưa ra biểu thức ở dạng tiền tố và hậu tố

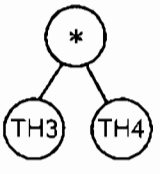
Giải:

1. Ta có $\left.\begin{matrix} TH1 = a*(b - c) \\ TH1 = d/e \end{matrix}\right\}$ suy ra cây biểu thức có dạng:

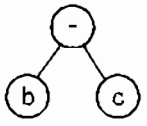


Xét TH1 = a*(b - c), toán hạng chưa ở dạng cơ bản ta phải phân tích để được như ở (1)

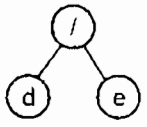
$\left.\begin{matrix} TH3 = a \\ TH4 = b - c \end{matrix}\right\}$ cây biểu thức của toán hạng này là:



Toán hạng TH4 đã ở dạng cơ bản nên ta có ngay cây biểu thức:



Cũng tương tự như vậy đối với toán hạng TH2, cây biểu thức tương ứng với toán hạng này là:



Tổng hợp cây biểu thức của các toán hạng ta được cây biểu thức (hình 5.12).

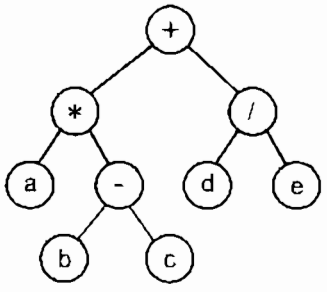
Bây giờ ta duyệt cây biểu thức ở hình 5.12.

Duyệt theo thứ tự trước:

+ * a - b c / d e

Duyệt theo thứ sau:

a b c - * d e / +



Hình 5.12: Cây biểu thức

3. CÂY TỔNG QUÁT

3.1. Biểu diễn cây tổng quát

Đối với cây tổng quát cấp m nào đó có thể sử dụng cách biểu diễn móc nối tương tự như đối với cây nhị phân. Như vậy ứng với mỗi nút ta phải dành ra m trường mối nối để trỏ tới các con của nút đó và như vậy số mối nối không sẽ rất nhiều: nếu cây có n nút sẽ có tới n(m-1) + 1 "mối nối không" trong số m×n mối nối.

Nếu tuỳ theo số con của từng nút mà định ra mối nối, nghĩa là dùng nút có kích thước biến đổi thì sự tiết kiệm không gian nhớ này sẽ phải trả giá bằng những phức tạp của quá trình xử lý trên cây.

Để khắc phục các nhược điểm trên, ta dùng cách biểu diễn cây nhị phân để biểu diễn cây tổng quát.

Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:

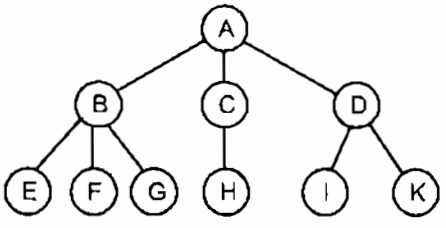
- Giữ lại nút con trái nhất làm nút con trái

- Các nút con còn lại chuyển thành các nút con phải

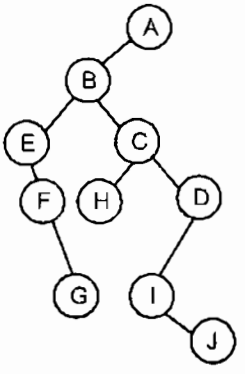
- Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu. Khi đó cây nhị phân này được gọi là cây nhị phân tương đương.

Ta có thể xem ví dụ dưới đây để thấy rõ qui trình. Giả sử có cây tổng quát như hình 5.13.

Cây nhị phân tương đương sẽ như hình 5.14.



Hình 5.13: Cây tổng quát



Hình 5.14: Cây nhị phân tương đương

3.2. Phép duyệt cây tổng quát

Phép duyệt cây tổng quát cũng được đặt ra tương tự như đối với cây nhị phân. Tuy nhiên, có một điều cần phải xem xét thêm, khi định nghĩa phép duyệt, đó là:

1. Sự nhất quán về thứ tự các nút được thăm giữa phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó.

2. Sự nhất quán giữa định nghĩa phép duyệt cây tổng quát với định nghĩa phép duyệt cây nhị phân. Vì cây nhị phân cũng có thể coi là cây tổng quát và ta có thể áp dụng định nghĩa phép duyệt cây tổng quát cho cây nhị phân.

Ta có thể xây dựng được định nghĩa của phép duyệt cây tổng quát T như sau:

Duyệt theo thứ tự trước

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Thăm gốc của T

   Duyệt các cây con thứ nhất T_1 của gốc của cây T theo thứ tự trước

   Duyệt các cây con còn lại T_2, T_3,..., T_n của gốc T theo thứ tự trước.

Duyệt theo thứ tự giữa

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Duyệt cây con thứ nhất T_1 của gốc của cây T theo thứ tự giữa

   Thăm gốc của cây T

   Duyệt các cây con còn lại T_2, T_3,..., T_n của gốc của cây T theo thứ tự giữa.

Duyệt theo thứ tự sau

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Duyệt cây con thứ nhất T_1 của gốc của cây T theo thứ tự sau

   Duyệt các cây con còn lại T_2, T_3,...,T_n của gốc của cây T theo thứ tự sau

   Thăm gốc của cây T.

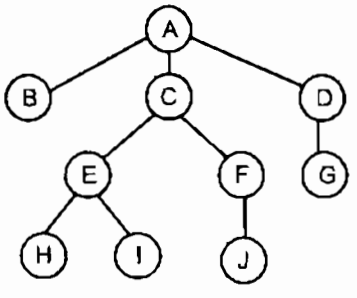
Ví dụ với cây ở hình 5.15 thì dãy tên các nút được thăm sẽ là:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B A H E I C J F G D

Thứ tự sau: B H I E J F C G D A

Bây giờ ta dựng cây nhị phân tương đương với cây tổng quát ở hình 5.15.



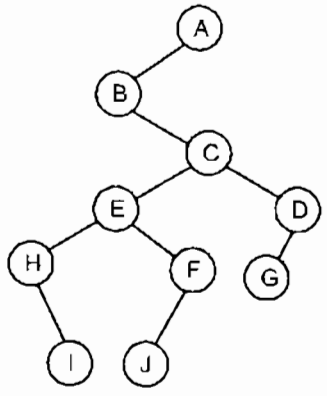
Hình 5.15: Cây tổng quát thể hiện quá trình duyệt

Dãy các nút được thăm khi duyệt nó theo phép duyệt của cây nhị phân:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B H I E J F C G D A

Thứ tự sau: I H JF E G D C B A



Hình 5.16: Cây nhị phân tương đương

Nhận xét: Với thứ tự trước, phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó đều cho một dãy tên như nhau. Phép duyệt cây tổng quát theo thứ tự sao cho dãy tên giống như dãy tên các nút được duyệt theo thứ tự giữa trong phép duyệt cây nhị phân. Còn phép duyệt cây tổng quát theo thứ tự giữa thì cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tương đương. Do đó đối với cây tổng quát, nếu định nghĩa phép duyệt như trên người ta thường chỉ nêu hai phép duyệt theo thứ tự trước và phép duyệt theo thứ tự sau.

4. TÌM KIẾM TRÊN CÂY NHỊ PHÂN

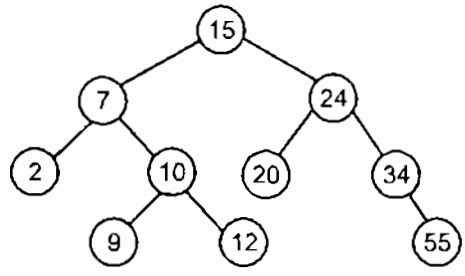
Cây nhị phân được sử dụng vào nhiều mục đích khác nhau. Tuy nhiên việc sử dụng cây nhị phân để lưu giữ và tìm kiếm thông tin vẫn là một trong những áp dụng quan trọng nhất của cây nhị phân. Trong phần này chúng ta sẽ nghiên cứu một lớp cây nhị phân đặc biệt phục vụ cho việc tìm kiếm thông tin, đó là cây nhị phân tìm kiếm.

Trong thực tế, một lớp đối tượng nào đó có thể được mô tả bởi một kiểu bản ghi, các trường của bản ghi biểu diễn các thuộc tính của đối tượng. Trong bài toán tìm kiếm thông tin, ta thường quan tâm đến một nhóm các thuộc tính nào đó của đối tượng mà thuộc tính này hoàn toàn xác định được đối tượng. Chúng ta gọi các thuộc tính này là khoá. Như vậy, khoá là một nhóm các thuộc tính của một lớp đối tượng sao cho hai đối tượng khác nhau phải có giá trị khác nhau trên nhóm thuộc tính đó.

4.1. Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân hoặc rỗng hoặc không rỗng thì phải thoả mãn đồng thời các điều kiện sau:

- Khoá của các nút thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các nút thuộc cây con phải của nút gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm.



Hình 5.17: Cây nhị phân tìm kiếm

Hình 5.17 biểu diễn một cây nhị phân tìm kiếm, trong đó khoá của các đỉnh là các số nguyên.

4.2. Cài đặt cây nhị phân tìm kiếm

Mỗi nút trên cây tìm kiếm nhị phân có dạng

Left	infor	Right

Trong đó trường Left: con trỏ chỉ tới cây con trái

Right: con trỏ chỉ tới cây con phải

infor: chứa thông tin của nút

Giả sử dữ liệu trên mỗi nút của cây có kiểu dữ liệu là ***Item***, khi đó cấu trúc dữ liệu của CNPTK được định nghĩa như sau:

```
struct Node
{
  Item infor;
  struct Node *Left, *Right;
};
struct Node *Root;
```

Tiếp theo ta nghiên cứu các phép toán trên cây nhị phân tìm kiếm.

4.3. Các thao tác cơ bản trên cây nhị phân tìm kiếm

4.3.1. Tìm kiếm

Tìm kiếm trên cây là một trong các phép toán quan trọng nhất đối với cây nhị phân tìm kiếm. Ta xét bài toán sau

Bài toán: Giả sử mỗi nút trên cây nhị phân tìm kiếm là một bản ghi, biến con trỏ Root chỉ tới gốc của cây và X là khoá cho trước. Vấn đề đặt ra là, tìm xem trên cây có chứa nút với khoá X hay không.

**** Giải thuật đệ qui***

```
struct Node *search (struct Node *Root; Item X)
/* Hàm search trả về con trỏ trỏ tới nút có khoá bằng X, ngược lại trả về NULL*/
{
  if (Root = = NULL) return NULL;
  else
        if (X < Root->infor) return search(Root->Left, X);
        else
              if (X > Root->infor) return search(Root->Right, X);
              else return Root;
}
```

**** Giải thuật lặp***

Trong thủ tục này, ta sẽ sử dụng biến địa phương *found* có kiểu int để điều khiển vòng lặp, nó có giá trị ban đầu là 0. Nếu tìm kiếm thành công thì *found* nhận giá trị 1, vòng lặp kết thúc và hàm *search*() trả về con trỏ trỏ đến nút có trường khoá bằng X. Còn nếu không tìm thấy thì giá trị của *found* vẫn là 0 và hàm *search*() trả về NULL.

```
struct Node *search (struct Node *Root, Item X)
{
        int found=0;
        struct Node *p;
        p = Root;
        while (p != NULL && found==0)
```

```
        {
                if (X > p->infor)
                p = p->Right;
                else
                        if (X < p->infor)
                        p = p->Left;
                        else     found = 1;
        }
        return p;
}
```

4.3.2. Đi qua CNPTK

Như ta đã biết CNPTK cũng là cây nhị phân nên các phép duyệt trên cây nhị phân cũng vẫn đúng trên CNPTK. Chỉ có lưu ý nhỏ là khi duyệt theo thứ tự giữa thì được dãy khoá theo thứ tự tăng dần.

4.3.3. Chèn một nút vào CNPTK

Việc thêm một nút có trường khoá bằng X vào cây phải đảm bảo điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất, do ta có thể thực hiện quá trình tương tự như thao tác tìm kiếm. Khi kết thúc việc tìm kiếm cũng chính là lúc tìm được chỗ cần chèn.

**** Giải thuật lặp***

Trong thủ tục này ta sử dụng biến con trỏ địa phương Q chạy trên các nút của cây bắt đầu từ gốc. Khi đang ở một nút nào đó, Q sẽ xuống nút con trái (hay phải) tuỳ theo khoá ở nút gốc lớn hơn (hay nhỏ hơn) khoá X.

Ở tại một nút nào đó khi P muốn xuống nút con trái (phải) thì phải kiểm tra xem nút này có nút con trái (phải) không. Nếu có thì tiếp tục xuống, ngược lại thì treo nút mới vào bên trái (phải) nút đó. Điều kiện Q = NULL sẽ kết thúc vòng lặp. Quá trình này được lại lặp khi có đỉnh mới được chèn vào.

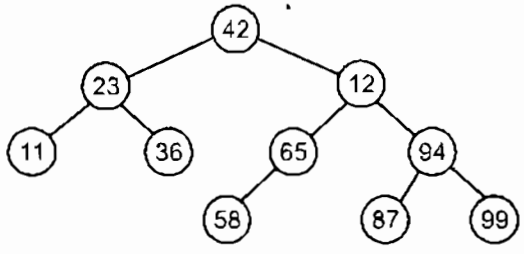
```
/* Hàm insert() bổ sung thêm nút có khóa X vào cây, hàm trả về 1 nếu
bổ sung thành công, ngược lại hàm trả về 0 - trường hợp nút có khóa
X đã có trên cây*/
int Insert (struct Node **Root, Item X)
{
        struct Node *P, *Q;
        P = new Node;
        P->infor = X;
        P->Left = NULL,
        P->Right = NULL;
        if (*Root = = NULL)
        {
                *Root = P;
                return 1;
        }
        else
        {       Q = *Root;
                while (Q != NULL)
                if (X < Q->infor)
                {
                        if (Q->Left != NULL) Q = Q->Left;
                        else    { Q->Left = P;
                                  return 1;
                                }
                }

        else
                if (X > Q->infor)
                {
                        if (Q->Right != NULL) Q = Q->Right;
                        else {
                                Q->Right = P;
                                return 1;
                              }
```

```
                }
                else return 0;
            }
    }
```

Nhận xét:

Để dựng được CNPTK ứng với một dãy khoá đưa vào bằng cách liên tục bổ các nút ứng với từng khoá, bắt đầu từ cây rỗng. Ban đầu phải dựng lên cây với nút gốc là khoá đầu tiên sau đó đối với các khoá tiếp theo, tìm trên cây không có thì bổ sung vào.

Ví dụ với dãy khoá: 42 23 74 11 65 58 94 36 99 87 thì cây nhị phân tìm kiếm dựng được sẽ có dạng ở hình 5.18.



Hình 5.18: Một cây nhị phân tìm kiếm

4.3.4. Loại bỏ nút trên cây nhị phân tìm kiếm

Đối lập với phép toán chèn vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi CNPTK một đỉnh có khoá X (ta gọi tắt là nút X) cho trước, sao cho việc huỷ một nút ra khỏi cây cũng phải bảo đảm điều kiện ràng buộc của CNPTK.

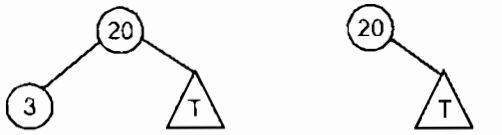
Có ba trường hợp khi huỷ một nút X có thể xảy ra:

- X là nút lá

- X là nút nửa lá (chỉ có một con trái hoặc con phải)

- X có đủ hai con (trường hợp tổng quát)

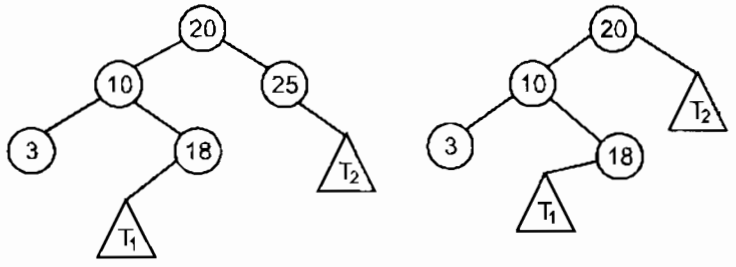
Trường hợp thứ nhất: chỉ đơn giản huỷ nút X vì nó không liên quan đến phần tử nào khác.



Cây trước khi xóa Cây sau khi xóa

Hình 5.19: Xóa nút lá trên cây

Trường hợp thứ hai: Trước khi xoá nút X cần móc nối cha của X với nút con (nút con trái hoặc nút con phải) của nó.

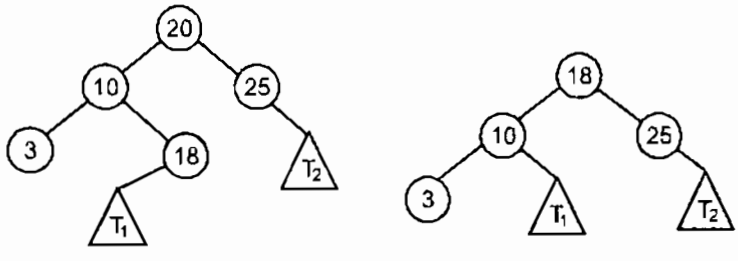


a. Cây trước khi xóa b. Cây sau khi xóa đỉnh (25)

Hình 5.20: Xóa nút nửa lá

Trường hợp tổng quát: khi nút bị loại bỏ có cả cây con trái và cây con phải, thì nút thay thế nó hoặc là nút ứng với khoá nhỏ hơn ngay sát trước nó (nút cực phải của cây con trái nó) hoặc nút ứng với khoá lớn hơn ngay sát sau nó (nút cực trái của cây con phải nó). Như vậy ta sẽ phải thay đổi một số mối nối ở các nút:

- Nút cha của nút bị loại bỏ
- Nút được chọn làm nút thay thế
- Nút cha của nút được chọn làm nút thay thế.

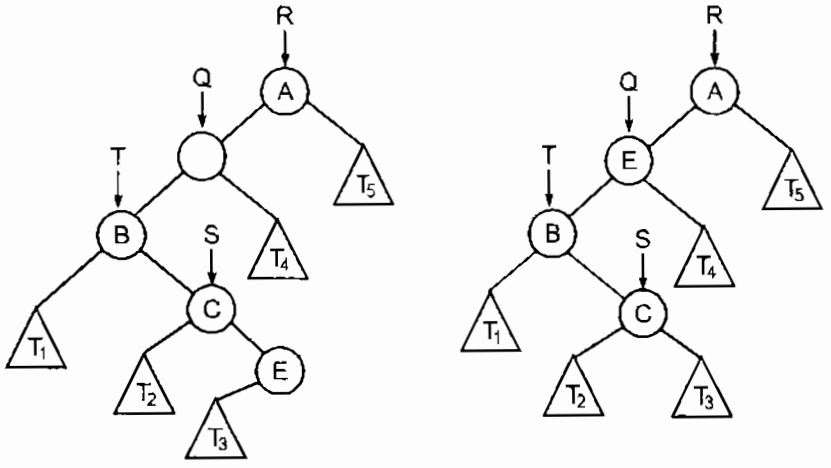


1. Cây trước khi xóa đỉnh 20 2. Cây sau khi xóa đỉnh 20

Hình 5.21: Xóa nút đầy đủ

Ở đây ta chọn nút thay thế nút bị xoá là nút cực phải của cây con trái (nút 18).

Sau đây là giải thuật thực hiện việc loại bỏ một nút trỏ bởi Q. Ban đầu Q chính là nối trái hoặc nối phải của một nút R trên cây nhị phân tìm kiếm, mà ta giả sử đã biết rồi.



1. Cây trước khi xóa nút trỏ bởi Q 2. Cây sau khi xóa nút trỏ bởi Q

Hình 5.22: Xóa nút trỏ bởi con trỏ Q

```
void Del (struct Node *Q)        //xoá nút trỏ bởi Q
{
        struct Node *T,*S, *P;
        P = Q;  //Xử lý trường hợp nút lá và nút nửa lá
        if (P->Left == NULL)
        {
                Q=P->Right; //R^.Left = P^.Right
                delete P;
        }
        else
                if (P->Right == NULL)
                {       Q = P->Left; delete P;
                }
                else    //Xử lý trường hợp tổng quát
```

```
        {
            T = P->Left;
            if (T->Right == NULL)
            {
                T->Right = P->Right;
                Q = T;
                delete P;
            }
            else
            {
                S = T->Right;
                //Tìm nút thay thế, là nút cực phải của cây
                while (S->Right != NULL)
                {
                    T = S;
                    S = T->Right;
                }
                S->Right = P->Right;
                T->Right = S->Left;
                S->Left = P->Left;
                Q = S;
                delete P;
        }
    }
}
```

Thủ tục xoá trường dữ liệu bằng X

- Tìm đến nút có trường dữ liệu bằng X
- Áp dụng thủ tục Delete để xoá

Sau đây chúng ta sẽ viết thủ tục loại khỏi cây gốc Root đỉnh có khoá X cho trước. Đó là thủ tục đệ quy, nó tìm ra đỉnh có khoá X, sau đó áp dụng thủ tục Del để loại đỉnh đó ra khỏi cây.

```
void Delete (Item X)
{
  if (Root != NULL)
```

```
            if (x < Root->infor) Delete(Root->Left, X)
            else if (X > Root->infor) Delete(Root->Right, X)
            else Dellete (Root);
}
```

Nhận xét:

Việc huỷ toàn bộ cây có thể thực hiện thông qua thao tác duyệt cây theo thứ sau. Nghĩa là ta sẽ huỷ cây con trái, cây con phải rồi mới huỷ nút gốc.

```
void RemoveTree ()
{
  if (Root != NULL)
  {
            RemoveTree(Root->Left);
            RemoveTree(Root->Right);
            delete Root;
  }
}
```

4.4. Thời gian thực hiện các phép toán trên CNPTK

Trong mục này ta sẽ đánh giá thời gian để thực hiện các phép toán trên CNPTK. Ta có nhận xét rằng, thời gian thực các phép tìm kiếm là số phép so sánh giá trị khoá X cho trước với khoá của các nút nằm trên đường đi từ gốc tới nút nào đó trên cây. Do đó, thời gian thực hiện các phép tìm kiếm, bổ sung và loại bỏ là độ dài đường đi từ gốc tới một nút nào đó trên cây.

Với giải thuật tìm kiếm nêu trên, ta thấy dạng cây nhị phân tìm kiếm dựng được hoàn toàn phụ thuộc vào dãy khoá đưa vào. Như vậy nghĩa là trong quá trình xử lý động ta không thể biết trước được cây sẽ phát triển ra sao, hình dạng của nó sẽ như thế nào.

Trong trường hợp nó là một cây nhị phân hoàn chỉnh (ta gọi là cân đối ngay cả khi nó chưa đầy đủ) thì chiều cao của nó là $\lceil \log_2(n+1) \rceil$, nên chi phí tìm kiếm có cấp độ là $O(\log_2 n)$.

Trong trường hợp nó là một cây nhị phân suy biến thành một danh sách tuyến tính (khi mà mỗi nút chỉ có một con trừ nút lá). Lúc đó các thao tác trên cây sẽ có độ phức tạp là O(n).

Người ta chứng minh được rằng số lượng trung bình các phép so sánh trong tìm kiếm trên cây nhị phân tìm kiếm là:

$$C_{tb} \approx 1{,}386 \log_2 n$$

Do đó cấp độ lớn của thời gian thực hiện trung bình các phép toán cũng chỉ là $O(\log_2 n)$.

5. CÂY CÂN BẰNG (AVL TREE)

Giả sử ta có một tập hợp dữ liệu nào đó. Vấn đề đặt ra là, ta phải tổ chức các dữ liệu đó như thế nào sao cho việc cập nhật thông tin (tìm kiếm, bổ sung và loại bỏ) đạt hiệu quả nhất. Với các kiểu dữ liệu này người ta có tổ chức trên cây nhị phân tìm kiếm. Khi đó thời gian trung bình thực hiện các phép toán là $O(\log_2 n)$. Trong nhiều áp dụng chúng ta cần thường xuyên thực hiện các phép toán bổ sung và loại bỏ khỏi CNPTK. Điều này có thể dẫn đến trường hợp cây suy biến thành danh sách tuyến tính. Đối với những cây này, việc thực hiện các phép toán kém hiệu quả do chi phí thời gian thực hiện là O(n). Để khắc phục nhược điểm này ta tổ chức dữ liệu trên một lớp cây đặc biệt sao cho thời gian thực hiện các phép toán luôn là $O(\log_2 n)$.

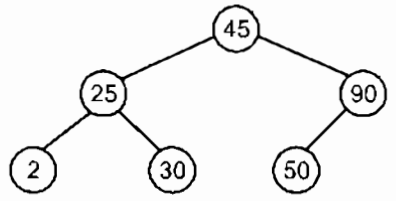
5.1. Cây cân bằng hoàn toàn

Định nghĩa: Cây cân bằng hoàn toàn (CCBHT) là cây nhị phân tìm kiếm mà tại mỗi nút của nó, số nút của cây con trái và cây con phải không chênh lệch nhau quá một.

Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay huỷ các nút trên cây có thể làm mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây là rất lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ rất nhanh. Đối với CCBHT, trong trường hợp xấu nhất ta cũng chỉ phải tìm qua log2n phần tử (n là số nút trên cây).

Hình 5.23 là ví dụ về một CCBHT.



Hình 5.23: Cây cân bằng hoàn toàn

CCBHT có n nút, có chiều cao h = $\log_2 n$. Đây chính là lý do cho phép đảm bảo khả năng tìm kiếm nhanh trên cấu trúc dữ liệu này.

Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một cấu trúc dữ liệu khác có đặc tính giống CCBHT nhưng ổn định hơn.

Như vậy, cần tìm cách tổ chức một cây đạt trạng thái cân bằng yếu hơn và việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ nhưng vẫn đảm bảo chi phí cho thao tác tìm kiếm đạt ở mức $O(\log_2 n)$.

5.2. Cây cân bằng

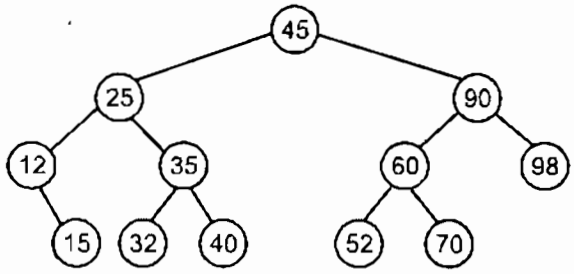
Năm 1962, P.M.ADELSON - VELSKI và E.LANDIS đã mở đầu cách giải quyết này bằng cách đưa ra dạng cây cân đối mới mà sau này mang tên của họ, đó là cây nhị phân cân đối AVL. Chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Từ khi được giới thiệu, cây AVL đã nhanh chóng được ứng dụng trong nhiều bài toán khác nhau. Vì vậy, nó mau chóng trở nên thịnh hành và thu hút nhiều nghiên cứu. Từ cây AVL, người ta đã phát triển thêm nhiều loại cấu trúc dữ liệu hữu dụng khác như cây đỏ - đen, B - Tree, v.v...

5.2.1. Định nghĩa

Cây AVL là cây nhị phân tìm kiếm mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch nhau không quá một.

Hình 5.24 là ví dụ cây cân bằng.



Hình 5.24: Cây AVL

Dễ dàng thấy CCBHT là cây cân bằng. Điều này ngược lại không đúng. Tuy nhiên, cây AVL là cấu trúc dữ liệu ổn định hơn hẳn CCBHT.

5.2.2. Chiều cao của cây AVL

Một vấn đề quan trọng, như đã đề cập trong phần trước, là ta phải khẳng định cây AVL n nút, phải có chiều cao khoảng $\log_2 n$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ có tối thiểu bao nhiêu nút?

Gọi N(h) số nút tối thiểu của cây AVL có chiều cao h.

Ta có N(0) = 0, N(1) = 1 và N(2) = 2.

Cây AVL tối thiểu có chiều cao h sẽ có cây con AVL tối thiểu chiều cao h-1 và cây con AVL tối thiểu chiều cao h-2. Như vậy:

$N(h) = 1 + N(h - 1) + N(h - 2)$ (1)

Ta lại có: $N(h - 1) > N(h - 2)$

Nên từ (1) suy ra:

$N(h) > 2N(h - 2)$

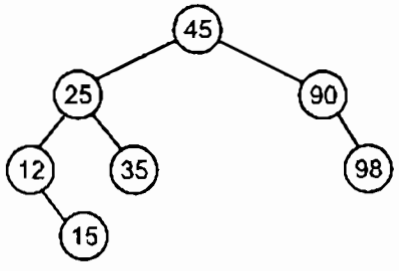
$N(h) > 2^2 N(h - 4)$

...

$N(h) > 2^i N(h - 2i)$

$\Rightarrow N(h) > 2^{h/2-1}$

$\Rightarrow h < 2\log_2(N(h)) + 2$



Hình 5.25: Cây AVL tối thiểu

Như vậy, cây AVL có chiều cao $O(\log_2 n)$.

Ví dụ: cây AVL tối thiểu có chiều cao h = 4 (hình 5.25).

5.2.3. Chỉ số cân bằng của một nút

Định nghĩa: chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.

Đối với cây AVL chỉ số cân bằng của mỗi nút có thể nhận một trong ba giá trị sau đây:

0: độ cao cây con trái bằng độ cao cây con phải

1: cây con phải có độ cao lớn hơn độ cao cây con trái là 1 (lệch phải)

-1: cây con trái có độ cao lớn hơn độ cao cây con phải là 1 (lệch trái).

5.2.4. Biểu diễn cây AVL

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Khi đó cấu trúc dữ liệu của cây cân bằng được khai báo như sau:

```
struct Node
{
        Item infor;
        struct Node *Left, *Right;
        int bal;
};
struct Node *Root;
```

Trong khai báo trên ta bổ sung thêm trường bal (balance: cân bằng), *bal=0* hai cây con cao bằng nhau, *bal=1* cao bên phải, *bal=-1* cao bên trái.

Sau đây chúng ta sẽ xét các phép toán bổ sung và loại bỏ trên cây AVL.

5.2.5. Bổ sung trên cây AVL

Khi bổ sung nút mới với khoá X cho trước được thực hiện bằng cách sau:

- Áp dụng giải thuật bổ sung vào cây nhị phân tìm kiếm

- Cân bằng lại các đỉnh mà tại đó tính cân bằng bị phá vỡ (độ cao của hai cây con khác nhau 2).

Xét các trường hợp sau

Trường hợp 1: Cây lệch trái (lệch phải) sau khi bổ sung vào cây con phải (trái) cây cân bằng.

Trường hợp 2: Hai cây con có độ cao bằng nhau sau khi bổ sung thì cây lệch trái hoặc lệch phải.

Trường hợp 3: Cây con trái (phải) cao hơn cây con phải (trái) 1, sau khi bổ sung vào cây con trái (phải) thì hai cây này có độ cao chênh nhau là 2. Như vậy, tính cân bằng bị phá vỡ.

Đối với các trường hợp 1 (TH1) và trường hợp 2 (TH2) khi bổ sung nút mới thì tính cân bằng không bị phá vỡ nên ta chỉ cần chỉnh lại các hệ số cân bằng ở nút đang xét và ở các nút tiền bối của nó.

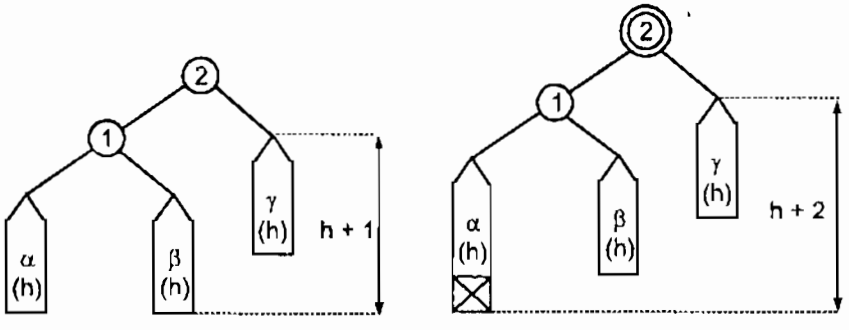
Đối với trường hợp 3 (TH3) ta phải sửa lại cây con mà ta đang xét là nút gốc (ta sẽ gọi là nút bất thường).

Qui ước:

(2) chỉ nút bất thường ứng với khoá bằng 2

⊠ chỉ nút mới bổ sung

α(h) chỉ cây con α có chiều cao h

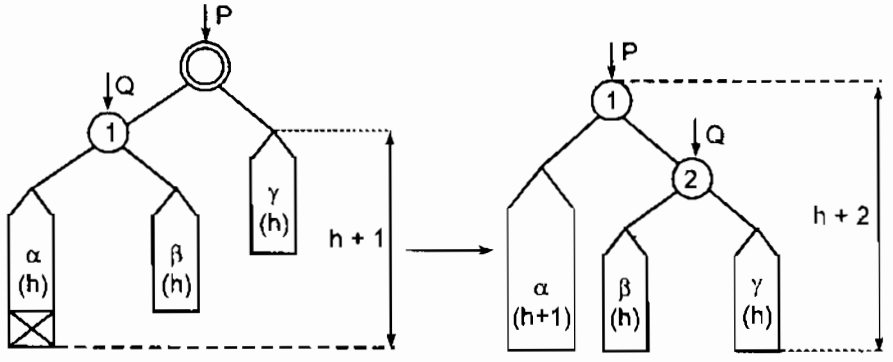


a) Trước khi bổ sung b) Sau khi bổ sung cây mất cân đối

Hình 5.26: Bổ sung vào cây AVL

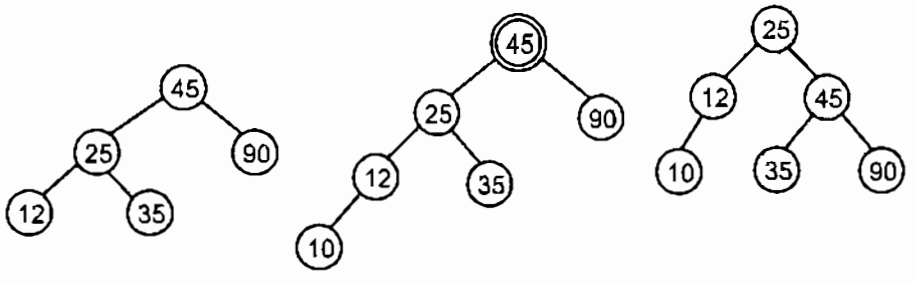
- *Trường hợp 3.1*: Nút bổ sung làm tăng chiều cao cây con trái của nút con trái nút bất thường

Đối với trường hợp này để tái cân đối ta phải thực hiện phép quay từ trái sang phải để đưa nút (1) lên vị trí gốc cây con, nút (2) sẽ trở thành con phải của nó và β được gắn vào thành con trái của (2). Người ta gọi phép quay này là phép quay đơn (single rotation) hay ta còn gọi là phép quay phải.



Hình 5.27: Quay phải cây P

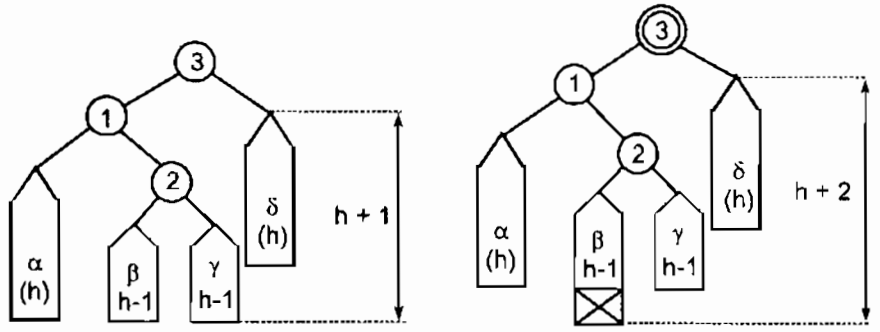
Ví dụ: Cho CNPTK hình 5.28.



a) Cây ban đầu b) Bổ sung thêm nút (10) c) Trái cân đối bằng phép quay đơn

Hình 5.28: Phép quay đơn

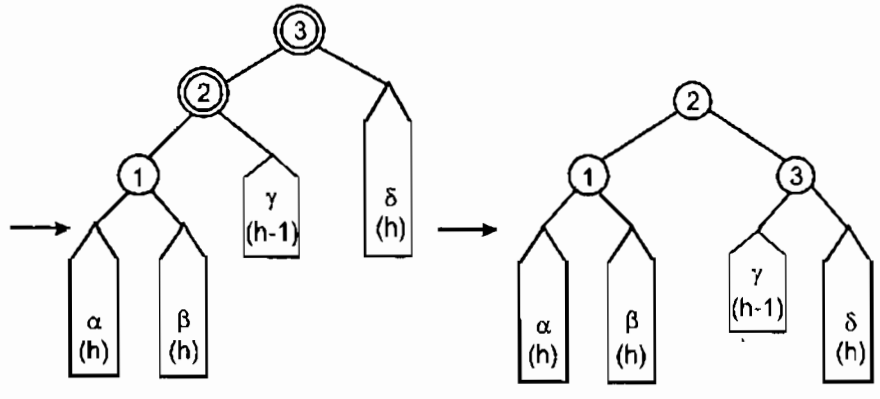
- *Trường hợp 3.2*: Nút mới bổ sung làm tăng chiều cao cây con phải của nút con trái nút bất thường.



a) Trước khi bổ sung b) Sau khi bổ sung cây mất cân đối

Hình 5.29: Bổ sung vào cây AVL

Đối với trường hợp này để tái cân đối ta sử dụng phép quay kép (double rotation), đó là việc phối hợp hai phép quay đơn: quay trái đối với cây con trái ((1), (2)) và quay phải đối với cây ((3), (2)) như hình 5.30.

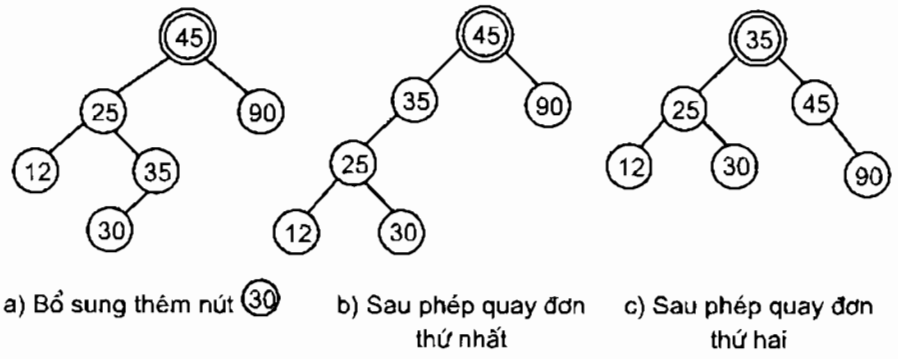


a) Sau phép quay đơn thứ nhất quay trái

b) Sau phép quay đơn thứ hai quay phải

Hình 5.30: Phép quay kép

Ví dụ: Cho cây như hình 5.31.



a) Bổ sung thêm nút 30 b) Sau phép quay đơn thứ nhất c) Sau phép quay đơn thứ hai

Hình 5.31: Minh họa phép quay kép

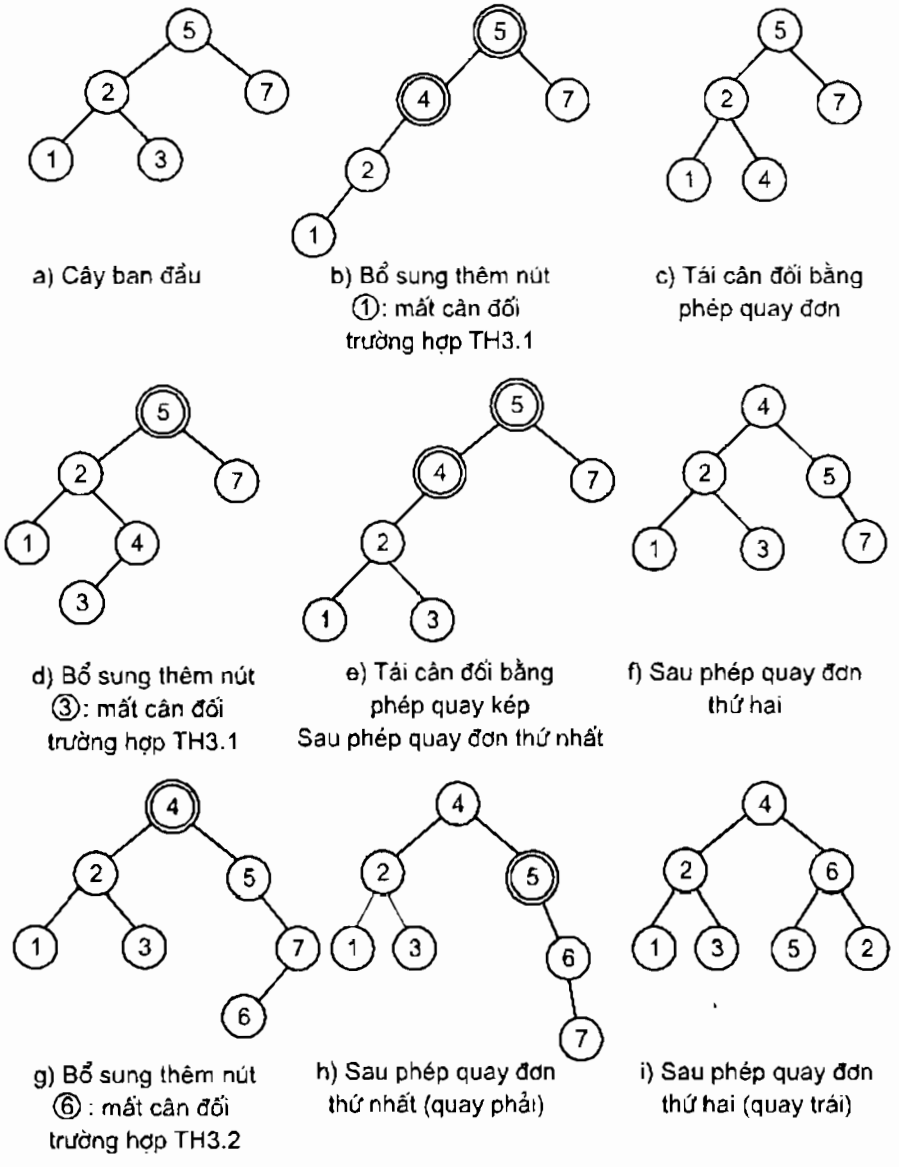
Nhận xét:

Sau khi thực hiện phép quay để tái cân đối cây con mà nút gốc là nút bất thường, chiều cao của các cây con đó vẫn giữ nguyên như trước lúc bổ sung, nghĩa là phép quay không làm ảnh hưởng tới chiều cao các cây có liên quan tới cây con này.

Trong quá trình thực hiện các phép quay, tính chất của cây nhị phân tìm kiếm luôn luôn được đảm bảo.

Đối với các trường hợp khi bổ sung thêm nút mới làm tăng chiều cao cây con phải của nút con phải và nút mới bổ sung làm chiều cao cây con trái của nút con phải nút bất thường, thì ta lần lượt thực hiện theo thứ tự ngược lại tương ứng với trường hợp TH3.1 và TH3.2.

Ví dụ sau đây minh hoạ cụ thể và các phép xử lý tương ứng.



Hình 5.32 : Minh họa các phép xử lý trên cây AVL

5.2.6. Huỷ một nút trên cây AVL

Cũng giống như thao tác thêm một nút, việc huỷ nút X ra khỏi cây AVL thực hiện giống như trên CNPTK. Chỉ sau khi huỷ, nếu tính cân bằng bị phá vỡ thì ta sẽ thực hiện việc cân bằng lại.

Tuy nhiên, việc cân bằng lại trong thao tác huỷ sẽ phức tạp hơn nhiều so với việc cân bằng khi thêm một nút do có thể xảy ra phản ứng dây chuyền.

Nhận xét:

- Thao tác thêm một nút có độ phức tạp O(1)

- Thao tác huỷ một nút có độ phức tạp O(h)

- Với cây cân bằng trung bình hai lần thêm vào cây thì cần một lần cân bằng lại; 5 lần huỷ thì cần một lần cân bằng lại.

- Việc huỷ một nút có thể phải cân bằng dây chuyền các nút từ gốc cho đến phần tử bị huỷ trong khi thêm vào chỉ cần một lần cân bằng cục bộ.

- Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn, nhưng việc cân bằng lại đơn giản hơn nhiều

- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.

Trong chương này chúng tôi đã trình bày các kiến thức căn bản về một dạng cấu trúc dữ liệu khá phức tạp, cấu trúc cây. Cây có nhiều dạng như cây tổng quát, cây nhị phân. Cấu trúc lưu trữ cũng khác nhau có thể lưu trữ cây bởi mảng hoặc con trỏ. Mong rằng qua đây bạn đọc nắm bắt được cách thức xây dựng cấu trúc dữ liệu dạng cây cùng với các phép xử lý và các thao tác trên cây, từ đó định ra phương pháp xây dựng phần mềm của mình một cách hiệu quả.

BÀI TẬP CHƯƠNG 5

1. Dựng cây nhị phân biết thứ tự các đỉnh khi duyệt theo:

a. Thứ tự trước: A D F G H K L P Q R W Z

Thứ tự giữa: G F H K D L A W R Q P Z

b. Theo thứ tự sau: F G H D A L P Q R Z W K

Thứ tự giữa: G F H K D L A W R Q P Z

2. Với mỗi biểu thức số học dưới đây, hãy vẽ cây nhị phân biểu diễn biểu thức ấy, rồi dùng các kiểu duyệt để tìm biểu thức tiền tố và hậu tố tương đương.

   a. a/(b - (c - (d - (e - f))))

   b. ((a * (b + c))/(d - (e + f))) * (g/(h/(i * j)))

3. Hãy trình bày các vấn đề sau:

   a. Định nghĩa và đặc điểm của cây nhị phân tìm kiếm

   b. Thao tác thực hiện tốt nhất trong kiểu này

   c. Hạn chế của kiểu này là gì?

4. Xét giải thuật tạo cây nhị phân tìm kiếm. Nếu thứ tự các khoá nhập vào là như sau:

   8 3 5 2 20 11 30 9 18 4

   thì hình ảnh cây được tạo như thế nào?

   Sau đó, nếu huỷ lần lượt các nút theo thứ tự như sau:

   11 20 8

   thì cây sẽ thay đổi như thế nào trong từng bước huỷ, vẽ cây minh hoạ qua từng bước.

5. Áp dụng thuật giải tạo cây AVL để dựng cây với thứ tự các khoá nhập vào như sau: 5 7 2 1 3 6 10 thì hình ảnh cây tạo được như thế nào? Giải thích rõ từng tình huống xảy ra khi thêm từng khoá vào cây và vẽ hình minh hoạ.

   Sau đó, nếu huỷ lần lượt các nút theo thứ tự như sau:

   5 6 7

   thì cây sẽ thay đổi như thế nào trong từng bước huỷ, vẽ sơ đồ và giải thích.

6. Viết các hàm xác định các thông tin của cây nhị phân T

   - Số nút lá

   - Số nút có đúng một cây con

- Số nút có đúng hai cây con
- Số nút có khoá nhỏ hơn X (giả sử T là CNPTK)
- Số nút có khoá lớn hơn X (giả sử T là CNPTK)
- Chiều cao của cây
- In ra tất cả các nút ở mức thứ k của cây
- Kiểm tra xem T có phải là cây cân bằng hoàn toàn không.

7. Xây dựng cấu trúc dữ liệu biểu diễn cây n- phân và tạo sinh cây nhị phân tương ứng với các khoá của cây n - phân.

   Giả sử khoá được lưu trữ chiếm k byte, mỗi con trỏ chiếm 4 byte, vậy dùng cây nhị phân thay cho cây n- phân thì có lợi gì trong việc lưu trữ các khoá.

8. Viết hàm chuyển một cây n- phân thành cây nhị phân.

9. Hãy tìm một ví dụ về một cây AVL có chiều cao là 6 và khi huỷ một nút lá (chỉ cụ thể) việc cân bằng lại lan truyền lên tận gốc của cây. Vẽ ra từng bước của quá trình huỷ và cân bằng lại này.

10. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

11. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL

    Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh - Việt.

12. Thông tin của mỗi nhân viên bao gồm mã nhân viên (kiểu số); tên nhân viên (kiểu chuỗi). Danh sách các nhân viên như sau: (30, Tùng), (35, Lan), (10, Sơn), (25, Nam), (20, Lý), (12, Lan), (40, Tuyết), (50, Cường), (60, Hảo), (55, Ánh), (48, Hương)

    a. Khai báo cấu trúc dữ liệu bằng CNPTK để lưu trữ cac nhân viên trên.

    b. Dựng CNPTK lần lượt các nhân viên này.

    c. Viết hàm cho biết thông tin của các nhân viên có tên là "Lan".

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Thống kê, 2000

[2] Đinh Mạnh Tường, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học và Kỹ thuật, 2001.

[3] Phạm Văn Ất, *Kỹ thuật lập trình C cơ sở và nâng cao*, NXB Khoa học và Kỹ thuật, 1999.

[4] Robert Kruse, C.L.Tondo, Bruce Leung, *Data structures & program design in C.*

[5] J.Courtin, I.Kowarski, *Nhập môn thuật toán và cấu trúc dữ liệu* (Nguyễn Ngọc Kỳ, Lương Chi Mai, Nguyễn Thanh Tùng dịch).

MỤC LỤC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Chịu trách nhiệm xuất bản

NGUYỄN THỊ THU HÀ

Biên tập: NGÔ MỸ HẠNH

BÙI NGỌC KHOA

Trình bày mỹ thuật: NGUYỄN MẠNH HOÀNG

Sửa bản in: BÙI NGỌC KHOA

Thiết kế bìa: TRẦN HỒNG MINH

In 2000 bản, khổ 16 x 24 cm, tại Công ty Cổ phần In Hà Nội

Số đăng ký kế hoạch xuất bản: 365-2009/CXB/6 - 139/TTTT

Số quyết định xuất bản: 182/QĐ-NXB TT&TT ngày 02 tháng 10 năm 2009

In xong và nộp lưu chiểu tháng 10 năm 2009.

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

Trụ sở chính: 18 Nguyễn Du, Hà Nội

ĐT Biên tập: 04.35772143, 04.35772145 ĐT Phát hành: 04.35772138

E-mail: nxb.tttt@mic.gov.vn Fax: 04.35772037

Website: www.nxbthongtintruyenthong.vn

Chi nhánh TP. Hồ Chí Minh: 8A đường D2, Phường 25, Quận Bình Thạnh, TP. Hồ Chí Minh

Điện thoại: 08.35127750, 35127751 Fax: 08.35127751

E-mail: cnsg.nxbtttt@mic.gov.vn

Chi nhánh TP. Đà Nẵng: 42 Trần Quốc Toản, quận Hải Châu, TP. Đà Nẵng

Điện thoại: 0511.3897467 Fax: 0511.3843359

E-mail: cndn.nxbtttt@mic.gov.vn

MỜI CÁC BẠN ĐÓN ĐỌC

1. NHẬP MÔN TIN HỌC
2. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
3. CƠ SỞ DỮ LIỆU
4. PHÂN TÍCH THIẾT KẾ HỆ THỐNG
5. THIẾT KẾ WEB
6. MẠNG CĂN BẢN

SÁCH CỦA NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG CÓ BÁN TẠI:

1. **Nhà sách Tiền phong**
   175 Nguyễn Thái Học, Hà Nội
2. **Nhà sách Nguyễn Văn Cừ**
   36 Xuân Thủy, Cầu Giấy, Hà Nội
3. **Nhà sách Minh Châu**
   Số 10 và 14/40 Tạ Quang Bửu, Hai Bà Trưng, Hà Nội
4. **Nhà sách Bách Khoa**
   Số 1, Đường Giải Phóng, Hà Nội
   86 - 107 Tô Hiến Thành, Quận 10, TP. HCM
5. **Nhà sách Thăng Long**
   2 Bis Nguyễn Thị Minh Khai, Quận 1, TP. HCM

Giá: 30.000đ