ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA CÔNG NGHỆ THÔNG TIN

BÀI GIẢNG HỌC PHẦN :

CẤU TRÚC DỮ LIỆU

BIÊN SOẠN : PHAN CHÍ TÙNG

LƯU HÀNH NỘI BỘ

ĐÀ NẴNG 2022

ĐỀ CƯƠNG HỌC PHẦN : CẤU TRÚC DỮ LIỆU

Chương 1. CÁC KHÁI NIỆM CƠ BẢN		(6)
1.1. Thuật toán và cấu trúc dữ liệu	0.5	0.5
1.2. Các kiểu dữ liệu cơ bản trong ngôn ngữ C		
1.2.1. Kiểu dữ liệu đơn giản	1	1.5
1.2.1.1. Kiểu ký tự		
1.2.1.2. Kiểu số nguyên		
1.2.1.3. Kiểu số thực		
1.2.2. Kiểu dữ liệu có cấu trúc	1	2.5
1.2.2.1. Kiểu mảng		
1.2.2.2. Kiểu chuỗi ký tự		
1.2.2.3. Kiểu bản ghi		
1.3. Kiểu con trỏ	1.5	4.0
1.3.1. Định nghĩa		
1.3.2. Khai báo kiểu con trỏ		
1.3.3. Hàm địa chỉ		
1.3.4. Các phép toán trên kiểu con trỏ		
1.4. Kiểu tham chiếu	1+	5.0
1.4.1. Định nghĩa		
1.4.2. Khai báo kiểu tham chiếu		
1.4.3. Ứng dụng kiểu tham chiếu		
1.5. Đệ qui	1-	6.0
1.5.1. Định nghĩa		
1.5.2. Các nguyên lý khi dùng kỹ thuật đệ qui		
Chương 2. DANH SÁCH		(9)
2.1. Khái niệm	0	6.0
2.2. Danh sách đặc	2	8.0
2.2.1. Định nghĩa		
2.2.2. Biểu diễn danh sách đặc		
2.2.3. Các phép toán trên danh sách đặc		
2.2.4. Ưu nhược điểm của danh sách đặc		
2.3. Danh sách liên kết (đơn)	3	11.0
2.3.1. Định nghĩa danh sách liên kết		
2.3.2. Biểu diễn danh sách liên kết		
2.3.3. Các phép toán trên danh sách liên kết		
2.3.4. Ưu nhược điểm của danh sách liên kết		
2.4. Danh sách đa liên kết	2	13.0
2.4.1. Định nghĩa		
2.4.2. Biểu diễn danh sách đa liên kết		
2.4.3. Các phép toán trên danh sách đa liên kết		
2.5. Danh sách liên kết kép	0.5	13.5
2.5.1. Định nghĩa		
2.5.2. Biểu diễn danh sách liên kết kép		
2.5.3. Các phép toán trên danh sách liên kết kép		
2.6. Danh sách liên kết vòng	0.5	14.0
2.7. Danh sách hạn chế		
2.7.1. Khái niệm	0	14.0
2.7.2. Ngăn xếp	0.5	14.5
2.7.2.1. Định nghĩa		
2.7.2.2. Biểu diễn ngăn xếp bằng danh sách liên kết		
2.7.2.3. Các phép toán trên ngăn xếp được biểu diễn bằng danh sách liên kết		

---o-O-o---

Tài liệu tham khảo:

[1] Đỗ Xuân Lôi, Cấu trúc dữ liệu và giải thuât, NXB Khoa học và kĩ thuật, 2003
[2] Nguyễn Hồng Chương, Cấu trúc dữ liệu ứng dụng và cài đặt bằng C, NXB TPHCM, 2003
[3] Lê Xuân Trường, Cấu trúc dữ liệu bằng ngôn ngữ C, NXB Thống kê, 1999
[4] Larry Nyhoff Sanford Leestma, Lập trình nâng cao bằng Pascal với các cấu trúc dữ liệu, 1991
[5] Nguyễn Trung Trực, Cấu trúc dữ liệu, 2000
[6] Đinh Mạnh Tường, Cấu trúc dữ liệu và thuật toán, NXB Khoa học và kĩ thuật, 2000
[7] Yedidyah Langsam, Moshe J.Augenstein, Aaron M.Tenenbaum, Data Structures Using C and C++, Prentice Hall, 1996
[8] Alfred V.Aho, John E.Hopcroft, Jeffrey D. Ullman, Data Structures and Algorithms, Addison Wesley, 1983

Chương 1: CÁC KHÁI NIỆM CƠ BẢN

1.1. Thuật toán và cấu trúc dữ liệu:

- Dữ liệu: Nói chung dữ liệu là tất cả những gì mà máy tính xử lý.
- Kiểu dữ liệu: Mỗi kiểu dữ liệu gồm các giá trị có cùng chung các tính chất nào đó và trên đó xác định các phép toán.
- Cấu trúc dữ liệu: là cách tổ chức và lưu trữ dữ liệu trong bộ nhớ máy tính.
- Thuật toán (hay Giải thuật): là tập hợp các bước theo một trình tự nhất định để giải một bài toán.
- Giữa cấu trúc dữ liệu và thuật toán có quan hệ mật thiết với nhau. Nếu ta biết cách tổ chức cấu trúc dữ liệu hợp lý thì thuật toán sẽ đơn giản hơn. Khi cấu trúc dữ liệu thay đổi thì thuật toán thường sẽ thay đổi theo.

1.2. Các kiểu dữ liệu cơ bản trong ngôn ngữ C:

1.2.1. Kiểu dữ liệu đơn giản: Có giá trị là đơn duy nhất. Gồm các kiểu dữ liệu đơn giản sau:

1.2.1.1. Kiểu ký tự: Có giá trị là một ký tự bất kỳ đặt giữa hai dấu nháy đơn, có kích thước 1 Byte (8 bit) và biểu diễn được một ký tự trong bảng mã ASCII, ví dụ: 'A' , '9' hoặc '+' . Gồm 2 kiểu ký tự chi tiết:

Tên kiểu	Miền giá trị
Char	từ -128 đến 127
unsigned char	từ 0 đến 255

1.2.1.2. Kiểu số nguyên: Có giá trị là một số nguyên, ví dụ số 2001. Gồm các kiểu số nguyên sau:

Tên kiểu	Kích thước	Miền giá trị
Int	2 Byte	từ -32768 đến 32767
unsigned int	2 Byte	từ 0 đến 65535
Long	4 Byte	từ -2147483648 đến 2147483647
unsigned long	4 Byte	từ 0 đến 4294967295

Lưu ý: Các kiểu ký tự cũng được xem là kiểu nguyên 1 Byte.

1.2.1.3. Kiểu số thực: Có giá trị là một số thực, ví dụ số 1.7 . Gồm các kiểu số thực sau:

Tên kiểu	Kích thước	Miền giá trị
Float	4 Byte	từ 3.4E-38 đến 3.4E+38
Double	8 Byte	từ 1.7E-308 đến 1.7E+308
long double	10 Byte	từ 3.4E-4932 đến 1.1E4932

Ví dụ 1: Viết chương trình nhập nhóm máu, chiều cao, năm sinh. Rồi tính và in tuổi.

Có 4 giá trị là: nhóm máu, chiều cao, năm sinh, tuổi.

Dữ liệu vào: nhóm máu, chiều cao, năm sinh. Dữ liệu ra: tuổi.

Dùng 4 biến kiểu đơn giản để chứa 4 giá trị là:

Biến tên nm kiểu char để chứa nhóm máu.

Biến tên cc kiểu float để chứa chiều cao.

Biến tên ns kiểu int để chứa năm sinh.

Biến tên t kiểu int để chứa tuổi.

Công việc:

- Nhập nhóm máu:
  - Máy hỏi.
  - Người trả lời.
- Nhập chiều cao:
  - Máy hỏi.
  - Người trả lời.
- Nhập năm sinh:
  - Máy hỏi.
  - Người trả lời.
- Tính tuổi.
- In tuổi.

```
#include <stdio.h>
#include <conio.h>>
#include <iostream>
#include <iomanip>
#include <string.h>
using namespace std;main()
```

```
{  float  cc;
   int    ns, t;
   char   nm ;
   cout << "\n Nhap nhom mau:";  cin >> nm;
   cout << "\n Nhap chieu cao:"; cin >> cc;
   cout << "\n Nhap nam sinh:";  cin >> ns;
   t=2022-ns;
   cout << "\n Tuoi la:" << t << " nghe." ;
}
```

Ví dụ 2: Nhập chiều dài, chiều rộng hình chữ nhật. Rồi tính và in chu vi.

Có 3 giá trị là: chiều dài, chiều rộng, chu vi hình chữ nhật.

Dữ liệu vào: Chiều dài, chiều rộng hình chữ nhật.

Dữ liệu ra: Chu vi hình chữ nhật.

Dùng 3 biến đơn giản để chứa 3 giá trị là:

Biến tên cd kiểu float để chứa chiều dài.

Biến tên cr kiểu float để chứa chiều rộng.

Biến tên cv kiểu float để chứa chu vi.

Công việc:

- Nhập chiều dài:
  - Máy hỏi.
  - Người trả lời.
- Nhập chiều rộng:
  - Máy hỏi.
  - Người trả lời.
- Tính chu vi.
- In chu vi.

```
#include  <iostream>
using  namespace  std;
main()
{  float   d, r, cv;
   cout << "\n Nhap chieu dai :"; cin >> d;
   cout << "\n Nhap chieu rong:"; cin >> r;
   cv=(d+r)*2;
   cout << "\n Chu vi la:" << cv << " nghe." ;
}
```

1.2.2. <u>Kiểu dữ liệu có cấu trúc:</u> Có giá trị gồm nhiều thành phần. Gồm các kiểu sau:

1.2.2.1. Kiểu mảng: Gồm nhiều thành phần có cùng kiểu dữ liệu, mỗi thành phần gọi là một phần tử, các phần tử được đánh chỉ số từ 0 trở đi. Để viết một phần tử của biến mảng thì ta viết tên biến mảng, tiếp theo là chỉ số của phần tử đặt giữa hai dấu ngoặc vuông.

Ví dụ lệnh: `float   A[3] ;`

Khai báo A là một biến mảng gồm 3 phần tử là A[0] , A[1] , A[2] đều có giá trị thuộc kiểu float.

Ví dụ 2: Nhập chiều dài, chiều rộng hình chữ nhật. Rồi tính và in chu vi.

Dùng 1 biến mảng A kiểu float gồm 3 phần tử:

Phần tử A[0] để chứa chứa chiều dài.

Phần tử A[1] để chứa chứa chiều rộng.

Phần tử A[2] để chứa chứa chu vi.

```
#include  <iostream>
using  namespace  std;
main()
{  float  A[3];
   cout << "\n Nhap chieu dai :"; cin >> A[0];
   cout << "\n Nhap chieu rong:"; cin >> A[1];
   A[2]=(A[0]+A[1])*2;
   cout << "\n Chu vi la:" << A[2];
```

```
}
```

1.2.2.2. Kiểu chuỗi ký tự:

Kiểu chuỗi ký tự có giá trị là một dãy ký tự bất kỳ đặt giữa 2 dấu nháy kép, ví dụ “Le Li”

Ta có thể xem chuỗi ký tự là một mảng mà mỗi phần tử là một ký tự.

Ta có thể khai báo biến chuỗi ký tự ht và gán giá trị “Le Li” bằng lệnh:

```
char  ht[15] = ”Le  Li” ;
```

Ví dụ: Dùng các biến riêng biệt ht kiểu chuỗi ký tự chứa họ tên, biến cc kiểu số thực float chứa chiều cao, biến ns kiểu số nguyên int chứa năm sinh, biến t kiểu số nguyên int chứa tuổi. Chương trình nhập họ tên, chiều cao, năm sinh của một người, rồi tính và in tuổi của người đó.

```
#include  <stdio.h>
#include  <conio.h>
#include  <iostream>
#include  <iomanip>
#include  <string.h>
using  namespace  std;
main()
{  char   ht[15];
   float  cc;
   int    ns, t;
   cout << "\n Nhap ho ten:"; fflush(stdin); gets(ht);
   cout << "\n Nhap chieu cao:"; cin >> cc;
   cout << "\n Nhap nam sinh:"; cin >> ns;
   t=2022-ns;
   cout << "\n Ban:" << ht << "  , Cao: " << cc << " m ," << setw(5) << t << " Tuoi.";
}
```

1.2.2.3. Kiểu bản ghi:

Kiểu bản ghi gồm nhiều thành phần có kiểu dữ liệu giống nhau hoặc khác nhau, mỗi thành phần gọi là một trường. Để viết một trường của biến bản ghi thì ta viết tên biến bản ghi, tiếp theo là dấu chấm, rồi đến tên trường

Ví dụ:

```
struct   SVIEN
         {  char  ht[15];
            float  cc;
            int  ns, t;
         };
         SVIEN   SV;
```

Đầu tiên khai báo kiểu bản ghi SVIEN gồm các trường ht, cc, ns, t lần lượt chứa họ tên, chiều cao, năm sinh, tuổi của một sinh viên. Sau đó khai báo biến SV thuộc kiểu SVIEN, như vậy biến SV là biến bản ghi gồm các trường được viết cụ thể là SV.ht , SV.cc, SV.ns, và SV.t

Ví dụ 1:Chương trình nhập họ tên, chiều cao, năm sinh của một người, rồi tính tuổi của người đó.

```
#include  <stdio.h>
#include  <conio.h>
#include  <iostream>
#include  <iomanip>
#include  <string.h>
using  namespace  std;
main()
{  struct  SVIEN
   {  char   ht[15];
      float  cc;
      int    ns, t;
   };
   SVIEN  SV;
   cout << "\n Nhap ho ten:"; fflush(stdin); gets(SV.ht);
   cout << "\n Nhap chieu cao:"; cin >> SV.cc;
   cout << "\n Nhap nam sinh:"; cin >> SV.ns;
```

```
  SV.t=2022-SV.ns;
  cout << "\n Ban:" << SV.ht << "  , Cao: " << SV.cc << " m ," << setw(5) << SV.t << " Tuoi.";
}
```

Ví dụ 2: Đầu tiên khai báo kiểu bản ghi HCN gồm 3 trường d, r, cv kiểu số thực float lần lượt chứa chiều dài, chiều rộng, chu vi hình chữ nhật. Sau đó khai báo biến B thuộc kiểu HCN, vậy biến B là biến bản ghi gồm 3 trường được viết đầy đủ là B.d, B.r và B.cv. Chương trình nhập chiều dài, chiều rộng, rồi tính chu vi hình chữ nhật.

```
#include  <iostream>
using  namespace  std;
main()
{  struct  HCN
   {  float  d, r, cv;
   };
   HCN  B;
   cout << "\n Nhap chieu dai :"; cin >> B.d;
   cout << "\n Nhap chieu rong:"; cin >> B.r;
   B.cv=(B.d+B.r)*2;
   cout << "\n Chu vi la:" << B.cv;
}
```

1.3. Kiểu con trỏ:

1.3.1. Định nghĩa:

Con trỏ là một biến mà giá trị của nó là địa chỉ của một đối tượng dữ liệu trong bộ nhớ. Đối tượng ở đây có thể là một biến hoặc một hàm.

Địa chỉ của một vùng nhớ trong bộ nhớ là địa chỉ của byte đầu tiên của vùng nhớ đó.

1.3.2. Khai báo biến con trỏ:

Ta có thể khai báo kiểu con trỏ trước, rồi sau đó khai báo biến con trỏ thuộc kiểu con trỏ đó.

```
typedef   kiểudữliệu   *kiểucontrỏ ;
kiểucontrỏ   biếncontrỏ ;
```

hoặc ta có thể khai báo trực tiếp biến con trỏ như sau:

```
kiểudữliệu   *biếncontrỏ ;
```

ví dụ

```
typedef   int   *xxx;
xxx   p;
```

ban đầu khai báo xxx là kiểu con trỏ chỉ đến giá trị kiểu int, sau đó khai báo p là biến thuộc kiểu xxx, như vậy biến p là biến con trỏ chỉ đến giá trị thuộc kiểu int.

Hoặc ta có thể khai báo trực tiếp biến con trỏ p như sau:

```
int   *p;
```

Tương tự ta có các khai báo:

```
int   ns=1993, t, *p, *p2;
float   cc=1.65, *pf;
char   nm='0', *pc;              // pc là biến con trỏ kiểu ký tự char
```

Lưu ý, khi biến p có giá trị là địa chỉ của một vùng nhớ mà trong vùng nhớ đó có chứa dữ liệu D thì ta nói rằng p là biến con trỏ chỉ đến dữ liệu D, vùng nhớ mà biến con trỏ p chỉ đến sẽ có tên gọi là *p hoặc p->

1.3.3. Hàm địa chỉ:

```
&biến
```

Trả về địa chỉ của biến trong bộ nhớ

1.3.4. Các phép toán trên kiểu con trỏ:

- Phép gán: Ta có thể gán địa chỉ của một biến cho biến con trỏ cùng kiểu.

Ví dụ:

```
p = &ns ;
```

Hoặc gán giá trị của hai biến con trỏ cùng kiểu cho nhau.

Ví dụ:

```
p2 = p;
```

Không được dùng các lệnh gán:

```
p=&cc;          hoặc  pf=&ns;          hoặc  pf=p;
```

- Phép cộng thêm vào con trỏ một số nguyên (đối với con trỏ liên quan đến mảng).
- Phép so sánh bằng nhau = = hoặc khác nhau !=

Ví dụ: if (p == p2) . . .
hoặc if (p != p2) . . .
- Hằng con trỏ NULL: cho biết con trỏ không chỉ đến đối tượng nào cả. Giá trị này có thể được gán cho mọi biến con trỏ kiểu bất kỳ.
Ví dụ: p = NULL; hoặc pf = NULL;
- Phép cấp phát vùng nhớ:
Lệnh biếncontrỏ = new kiểudữliệu;
Vd lệnh: p = new int;
Cấp phát vùng nhớ có kích thước 2 Byte (ứng với kiểu dữ liệu int) và gán địa chỉ của vùng nhớ này cho biến con trỏ p, như vậy vùng nhớ đó có tên gọi là *p hoặc p->
Tương tự ta có lệnh: pf=new float;
- Phép thu hồi vùng nhớ:
Lệnh: delete biếncontrỏ;
Vd lệnh: delete p;
thu hồi vùng nhớ mà biến con trỏ p chỉ đến.
Ví dụ:

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <iomanip>
#include <string.h>
using namespace std;
main()
{ struct SVIEN
  { char ht[15];
    float cc;
    int ns, t;
  };
  SVIEN *p;
  p = new SVIEN;
  cout << "\n Nhap ho ten:"; fflush(stdin); gets((*p).ht);
  cout << "\n Nhap chieu cao:"; cin >> (*p).cc;
  cout << "\n Nhap nam sinh:"; cin >> p->ns;
  (*p).t=2022-(*p).ns;
  cout << "\n Ban:" << (*p).ht << " , Cao: " << (*p).cc << " m ," << setw(5) << p->t << " Tuoi.";
  delete p;
}
```

1.4. Kiểu tham chiếu

1.4.1. Định nghĩa:

Ngôn ngữ C có 3 loại biến:

- Biến giá trị: chứa một giá trị dữ liệu thuộc về một kiểu nào đó (kiểu số nguyên, kiểu số thực, kiểu ký tự . . .)

ví dụ int ns=1993;

- Biến con trỏ: chứa địa chỉ của một đối tượng.

Ví dụ: int *p=&ns;

Hai loại biến này đều được cấp bộ nhớ và có địa chỉ riêng.

- Loại thứ ba là biến tham chiếu, là biến không được cấp phát bộ nhớ, không có địa chỉ riêng, được dùng làm bí danh cho một biến khác và dùng chung vùng nhớ của biến đó.

1.4.2. Khai báo biến kiểu tham chiếu:

Cú pháp: kiểudữliệu &biếnthamchiếu = biếnbịthamchiếu ;
Trong đó, biếnthamchiếu sẽ tham chiếu đến biếnbịthamchiếu và dùng chung vùng nhớ của biếnbịthamchiếu này.
Vd float cc=1.7;
float &f = cc;

Ban đầu khai báo biến cc kiểu số thực float. Sau đó khai báo biến tham chiếu f sẽ tham chiếu đến biến bị tham chiếu cc cùng kiểu float. Vậy biến tham chiếu f sẽ dùng chung vùng nhớ của biến cc. Khi đó những thay đổi giá trị của biến cc cũng là những thay đổi của biến f và ngược lại, chẳng hạn nếu tiếp theo có lệnh:

f = 1.8;

thì biến cc sẽ tự động có giá trị mới là 1.8

1.4.3. Ứng dụng kiểu tham chiếu:

```
#include  <stdio.h>
#include  <conio.h>
#include  <iostream>
#include  <iomanip>
#include  <string.h>
using  namespace  std;
void  DOI ( int  x , int  &y , int  *z )       // tham so hinh thuc
{ cout << "\n\n   Con truoc: " << setw(5) << x << setw(5) << y << setw(5) << *z ;
  x=x+1; y=y+2; *z=*z+3;
  cout << "\n\n   Con   sau: " << setw(5) << x << setw(5) << y << setw(5) << *z ;
}
main()
{ int  i=10, j=20, k=30;
  cout << "\n\n Chinh truoc: " << setw(5) << i << setw(5) << j << setw(5) << k ;
  DOI( i , j , &k );
  cout << "\n\n Chinh   sau: " << setw(5) << i << setw(5) << j << setw(5) << k ;
}
```

Kết quả hiện lên màn hình là:

Chính trước:	10	20	30
Con truoc:		10	20	30
Con sau:		11	22	33
Chinh sau:	10	22	33	

1.5. Đệ qui

1.5.1. Định nghĩa:

Trong thân một chương trình mà có lệnh gọi ngay chính nó thực hiện thì gọi là tính đệ qui của chương trình.

1.5.2. Các nguyên lý khi dùng kỹ thuật đệ qui:

- Tham số hóa bài toán: để thể hiện kích cỡ của bài toán.
- Tìm trường hợp dễ nhất: mà ta biết ngay kết quả bài toán.
- Tìm trường hợp tổng quát: để đưa bài toán với kích cỡ lớn về bài toán tương tự có kích cỡ nhỏ hơn.

Ví dụ 1: Viết chương trình tính giai thừa của số nguyên không âm n bằng giải thuật đệ qui.

- Tham số hóa bài toán: Gọi n là số nguyên không âm cần tính giai thừa.
- Trường hợp dễ nhất: Nếu n=0 thì n!=1
- Trường hợp tổng quát: Nếu n>0 thì n!=(n-1)!*n

```
#include  <iostream>
using  namespace  std;
long  fact(int  n)
{  if  (n==0)  return  1;
   else  return  n*fact(n-1);
}
main()
{  int  n; long  kq;
   cout << "\n Nhap so can tinh giai thua n="; cin >> n;
   kq=fact(n);
   cout << "\n Ket qua la:" << kq;
}
```

Ví dụ 2: Viết chương trình tính ước số chung lớn nhất của 2 số nguyên x và y.

- Tham số hóa bài toán: Gọi x và y là 2 số nguyên cần tính ước số chung lớn nhất.
- Trường hợp dễ nhất: Nếu x=y thì ước chung lớn nhất của chúng là x.

- Trường hợp tổng quát: Nếu x< >y thì

USCLN(x, y)=USCLN(x , y-x) nếu x<y

USCLN(x, y)=USCLN(x-y , y) nếu y<x

Chẳng hạn, tìm ước số chung lớn nhất của 30 và 18.

30 có các ước số là: 1 , 2 , 3 , 5 , 6 , 10 , 15 , 30

18 có các ước số là: 1 , 2 , 3 , 6 , 9 , 18

30 và 18 có các ước số chung là: 1 , 2 , 3 , 6

Ước số chung lớn nhất của 30 và 18 là: 6

```
#include  <iostream>
using  namespace  std;
int  uscln(int  x, int  y)
{  if  (x==y)  return  x;
   else  if  (x<y)  return  uscln(x, y-x);
        else  return  uscln(x-y, y);
}
main()
{
   int  a, b, c;
   cout << "\n Nhap so nguyen thu nhat:"; cin >> a;
   cout << "\n Nhap so nguyen thu  hai:"; cin >> b;
   c = uscln(a, b);
   cout << "\n Uoc so chung lon nhat la:" << c;
}
```

--- HẾT CHƯƠNG 1---

Chương 2: DANH SÁCH

2.1. Khái niệm:

- Danh sách: là một dãy các phần tử a_1, a_2, a_3, . . . a_n trong đó nếu biết được phần tử đứng trước thì sẽ biết được phần tử đứng sau.
- n: là số phần tử của danh sách.
- Danh sách rỗng: là danh sách không có phần tử nào cả, tức n=0
- Danh sách là khái niệm thường gặp trong cuộc sống, như danh sách các sinh viên trong một lớp, danh sách các môn học trong một học kỳ . . .
- Có 2 cách cơ bản biểu diễn danh sách:
  + Danh sách đặc: các phần tử được lưu trữ kế tiếp nhau trong bộ nhớ, phần tử thứ i được lưu trữ ngay trước phần tử thứ i+1 dưới hình thức một mảng.
  + Danh sách liên kết: các phần tử được lưu trữ tại những vùng nhớ khác nhau trong bộ nhớ, nhưng chúng được kết nối với nhau nhờ các vùng liên kết.
- Các phép toán thường dùng trên danh sách:
  + Khởi tạo danh sách (tức là làm cho danh sách có, nhưng là danh sách rỗng).
  + Kiểm tra xem danh sách có rỗng không.
  + Liệt kê các phần tử có trong danh sách.
  + Tìm kiếm phần tử trong danh sách.
  + Thêm phần tử vào danh sách.
  + Xóa phần tử ra khỏi danh sách.
  + Sửa các thông tin của phần tử trong danh sách.
  + Thay thế một phần tử trong danh sách bằng một phần tử khác.
  + Sắp xếp thứ tự các phần tử trong danh sách.
  + Ghép một danh sách vào một danh sách khác.
  + Trộn các danh sách đã có thứ tự để được một danh sách mới cũng có thứ tự.
  + Tách một danh sách ra thành nhiều danh sách.
  . . .
- Trong thực tế một bài toán cụ thể chỉ dùng một số phép toán nào đó, nên ta phải biết cách biểu diễn danh sách cho phù hợp với bài toán.

2.2. Danh sách đặc:

2.2.1. Định nghĩa:

Danh sách đặc là danh sách mà các phần tử được lưu trữ kế tiếp nhau trong bộ nhớ dưới hình thức một mảng.

2.2.2. Biểu diễn danh sách đặc:

Xét danh sách có tối đa 100 sinh viên gồm các thông tin: họ tên, chiều cao, cân nặng tiêu chuẩn, như :

1	LÊ LI	1.7	65
2	LÊ BI	1.8	75
3	LÊ VI	1.4	35
4	LÊ NI	1.6	55
5	LÊ HI	1.5	45

Trong đó cân nặng tiêu chuẩn được tính theo công thức:

Cân nặng tiêu chuẩn (kg) = Chiều cao x 100 – 105

Khai báo:

```
const  int  Nmax=100;
typedef  char   infor1[15];
typedef  float  infor2;
typedef  int    infor3;
struct  element
{ infor1  ht;
  infor2  cc;
  infor3  cntc;
};
```

```
typedef  element  DS[Nmax];
DS  A;
int  n;
```
Hằng Nmax kiểu int chứa số phần tử tối đa có thể có của danh sách.
Biến n kiểu int chứa số phần tử thực tế hiện nay của danh sách, ví dụ n=5.
Kiểu bản ghi element gồm các trường ht, cc, cntc lần lượt chứa họ tên, chiều cao, cân nặng tiêu chuẩn của một sinh viên.
infor1, infor2, infor3 lần lượt là các kiểu dữ liệu của các trường ht, cc, cntc.
DS là kiểu mảng gồm Nmax phần tử kiểu element.
Biến A kiểu DS là biến mảng gồm Nmax phần tử kiểu element.

2.2.3. Các phép toán trên danh sách đặc:

- Khởi tạo danh sách: Khi mới khởi tạo danh sách là rỗng, ta cho n nhận giá trị 0.

```
void  Create(DS  A, int  &n)
{ n=0;
}
```

- Liệt kê các phần tử trong danh sách: Ta liệt kê các phần tử từ phần tử đầu tiên trở đi.

```
void  Display(DS  A, int  n)
{ int  i;
  for  (i=1; i<=n; i++)
        printf("\n %15s  %7.2f  %7d" ,A[i].ht, A[i].cc, A[i].cntc);
}
```

- Tìm kiếm một phần tử trong danh sách: Tìm phần tử có họ tên x cho trước. Ta tìm bắt đầu từ phần tử đầu tiên trở đi, cho đến khi tìm được phần tử cần tìm hoặc đã kiểm tra xong phần tử cuối cùng mà không có thì dừng. Hàm Search(A, n, x) tìm và trả về giá trị kiểu int, là số thứ tự của phần tử đầu tiên tìm được hoặc trả về giá trị -1 nếu tìm không có.

```
int  Search(DS  A, int  n, infor1  x)
{ int  i;
  i=1;
  while  ( (i<=n)  &&  (strcmp(A[i].ht,x)!=0) )
        i++;
  if  (i<=n)  return  i;
  else  return  -1;
}
```

- Thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào vị trí thứ t trong danh sách.

Điều kiện: n<Nmax và $1 \le t \le n+1$

Khi đó các phần tử từ thứ t đến thứ n được dời xuống 1 vị trí trong đó phần tử ở dưới thì dời trước, phần tử ở trên dời sau. Sau đó chèn phần tử mới vào vị trí thứ t, cuối cùng tăng giá trị n lên 1 đơn vị.

```
void  InsertElement(DS  A, int &n, int  t, infor1  x, infor2  y, infor3  z).
{ int  i;
  if ( (n<Nmax) && (t>=1) && (t<=n+1) )
        { for (i=n; i>=t; i--)
                    A[i+1]=A[i];
          strcpy(A[t].ht,x); A[t].cc=y; A[t].cntc=z;
          n++;
        }
}
```

- Xóa phần tử thứ t trong danh sách, Điều kiện: $1 \le t \le n$

Khi đó các phần tử từ thứ t+1 đến thứ n được dời lên 1 vị trí, trong đó phần tử ở trên thì dời trước, phần tử ở dưới dời sau, cuối cùng giảm giá trị của n xuống 1 đơn vị.

```
void  DeleteElement(DS  A, int  &n,  int  t)
{ int i;
  if ( (t>=1) && (t<=n) )
        { for (i=t+1; i<=n; i++)
                A[i-1]=A[i];
```

```
            n--;
          }
}
```

2.2.4. Ưu nhược điểm của danh sách đặc:

* Ưu điểm:
  - Dễ viết chương trình.
  - Tiết kiệm bộ nhớ cho mỗi phần tử.
  - Chỉ tiện lợi cho danh sách dùng ít bộ nhớ.

* Khuyết điểm:
  - Không tiện lợi cho danh sách dùng nhiều bộ nhớ.
  - Cần vùng nhớ liên tục.
  - Khai báo trước số lượng cụ thể số phần tử của danh sách.

2.3. Danh sách liên kết (đơn):

2.3.1. Định nghĩa danh sách liên kết:

Danh sách liên kết là danh sách mà các phần tử được kết nối với nhau nhờ các vùng liên kết.

2.3.2. Biểu diễn danh sách liên kết:

Xét danh sách sinh viên gồm các thông tin: họ tên, chiều cao, cân nặng tiêu chuẩn.

1	LÊ LI	1.7	65
2	LÊ BI	1.8	75
3	LÊ VI	1.4	35
4	LÊ NI	1.6	55
5	LÊ HI	1.5	45

```
typedef  char   infor1[15];
typedef  float  infor2;
typedef  int    infor3;
struct   element
{ infor1   ht;
  infor2   cc;
  infor3   cntc;
  element  *next;
};
typedef  element  *List;
List  F;// hoặc element   *F;
```

Hỏi F: 2002 ở đâu? int *F;
Hỏi F: 1.7 ở đâu ? float *F;
Hỏi F: LE LI 1.7 65 ở đâu ? element *F;

Kiểu bản ghi element gồm các trường ht, cc, cntc dùng để chứa các thông tin của một phần tử trong danh sách, ngoài ra còn có thêm trường liên kết next chứa địa chỉ của phần tử tiếp theo trong danh sách.
Kiểu con trỏ List dùng để chỉ đến một phần tử kiểu element.
Biến con trỏ F luôn luôn chỉ đến phần tử đầu tiên trong danh sách liên kết.

3 P5. Bệnh nhân 271 mắc bệnh ngày 2/5 là chuyên gia người Anh. Tên tác giả.
5 P3.Tối 23.1, Bệnh viện Chợ Rẫy (TP.HCM) xác nhận 2 bệnh nhân số 1 và số 2 tại Việt Nam, là 2 cha con người Trung Quốc, người con làm việc tại Long An. Xem tiếp trang 9.
7 P1.BÀI: TÌNH HÌNH COVID 19 VIỆT NAM:
Dịch bệnh Covid-19 bắt đầu bùng phát từ tháng 12.2019 tại thành phố Vũ Hán, Trung Quốc. Đã lan ra 212 quốc gia với 2.645.703 ca nhiễm, và 184.325 người tử vong. Xem tiếp trang 20.
9 P4.Ngày 30.1, có thêm 3 bệnh nhân là số 3, số 4, số 5 thuộc tỉnh Vĩnh Phúc. Các bệnh nhân này ở trong nhóm được một công ty Nhật cử sang Trung Quốc tập huấn. Xem tiếp trang 3.
20 P2.Tại Việt Nam, ca nhiễm Covid-19 đầu tiên được phát hiện vào ngày 23.1, và tới nay đã có 268 trường hợp nhiễm bệnh. Xem tiếp trang 5.

7 P1.BÀI: TÌNH HÌNH COVID 19 VIỆT NAM:

Dịch bệnh Covid-19 bắt đầu bùng phát từ tháng 12.2019 tại thành phố Vũ Hán, Trung Quốc. Đã lan ra 212 quốc gia với 2.645.703 ca nhiễm, và 184.325 người tử vong. Xem tiếp trang 20.
20 P2.Tại Việt Nam, ca nhiễm Covid-19 đầu tiên được phát hiện vào ngày 23.1, và tới nay đã có 271 trường hợp nhiễm bệnh. Xem tiếp trang 5.
5 P3.Tối 23.1, Bệnh viện Chợ Rẫy (TP.HCM) xác nhận 2 bệnh nhân số 1 và số 2 tại Việt Nam, là 2 cha con người Trung Quốc, người con làm việc tại Long An. Xem tiếp trang 9.
9 P4.Ngày 30.1, có thêm 3 bệnh nhân là số 3, số 4, số 5 thuộc tỉnh Vĩnh Phúc. Các bệnh nhân này ở trong nhóm được một công ty Nhật cử sang Trung Quốc tập huấn. Xem tiếp trang 3.
3 P5. Bệnh nhân 271 mắc bệnh ngày 2/5 là chuyên gia người Anh. Tên tác giả.

2.3.3. Các phép toán trên danh sách liên kết:

- Khởi tạo danh sách: Khi mới khởi tạo danh sách là rỗng, ta cho F nhận giá trị NULL.

```
void  Create(List  &F)
{  F=NULL;
}
```

F=7TĐT;
p=F; thì p=7TĐT;
p=(*p).next;
Nếu (*p).next=NULL; thì p đang ở phần tử cuối cùng.
p=NULL; thì p đang ở **dưới** phần tử cuối cùng.

F=	7TĐT	1	LÊ LI	1.7	65
	8NLB	2	LÊ BI	1.8	75
	5TĐT	3	LÊ VI	1.4	35
	6NLB	4	LÊ NI	1.6	55
P=	9TĐT	5	LÊ HI	1.5	45

Thì dữ liệu của các phần tử được lưu trữ trong bộ nhớ RAM là:

	5TĐT	LÊ VI	1.4	35	9TDT
F=	7TĐT	LÊ LI	1.7	65	8NLB
	9TĐT	LÊ HI	1.5	45	NULL
	6NLB	LÊ NI	1.6	55	9TĐT
	8NLB	LÊ BI	1.8	75	5TĐT

F=7TĐT; p=F; *p (*p).ht (*p).cc (*p).cntc (*p).next
Before=after;
after=(*after).next; p=(*p).next;

- Liệt kê các phần tử trong danh sách: Ta liệt kê các phần tử kể từ phần tử đầu tiên được chỉ bởi biến con trỏ F và dựa vào trường liên kết next để lần lượt liệt kê các phần tử tiếp theo.

Biến con trỏ p lần lượt chỉ đến từng phần tử trong danh sách bắt đầu từ phần tử đầu tiên chỉ bởi F trở đi.

```
void  Display(List  F)
{ List p;
  p=F; int  d=0;
  while  (p != NULL)
  { d++;
    cout <<"\n "<<d<<": "<<(*p).ht<<setw(5)<<(*p).cc << setw(5) << (*p).cntc;
    p=(*p).next;
  }
}
```

- Tìm kiếm một phần tử trong danh sách: Tìm phần tử có họ tên x trong danh sách.

Ta tìm bắt đầu từ phần tử đầu tiên được chỉ bởi F trở đi cho đến khi tìm được phần tử cần tìm hoặc đã kiểm tra xong phần tử cuối cùng mà không có thì dừng. Hàm Search(F, x) kiểu List, tìm và trả về địa chỉ của phần tử đầu tiên tìm được hoặc trả về giá trị NULL nếu tìm không có.

```
List  Search(List  F, infor1  x)
{ List  p;  p=F;
  while ( (p!=NULL) && strcmp((*p).ht,x) !=0 )
```

```
    p= (*p).next;
  return  p;
}
```

- Thêm một phần tử vào đầu danh sách:

Thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào đầu danh sách.

Biến con trỏ p chỉ đến phần tử mới cần thêm vào.

```
void  InsertFirst(List  &F, infor1  x, infor2  y, infor3  z)
{ List  p;
  p=new  element;
  strcpy((*p).ht,x); (*p).cc=y; (*p).cntc=z;
  (*p).next = F;                  // 1      "Cho" (*p).next "chỉ đến phần tử giống" F "chỉ"
  F = p;                          // 2
}
```

```
F=NULL;
Nhập LI
                        1       LI
Nhập BI
                        1       BI
                        2       LI
Nhập VI
                        1       VI
                        2       BI
                        3       L1
```

- Thêm một phần tử vào danh sách đã có thứ tự:

Thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào danh sách trước đó đã có thứ tự họ tên tăng dần.

Biến con trỏ p chỉ đến phần tử mới cần thêm vào.

Các biến con trỏ before và after lần lượt chỉ đến phần tử đứng ngay trước và ngay sau phần tử mới. Để tìm after thì ta tìm bắt đầu từ phần tử đầu tiên chỉ bởi F trở đi cho đến khi gặp được phần tử đầu tiên có họ tên lớn hơn x thì dừng, rồi chèn phần tử mới vào giữa.

```
void  InsertSort(List  &F, infor1  x, infor2  y, infor3  z)
{ List  p, before, after;
  p=new  element;
  strcpy((*p).ht,x); (*p).cc=y; (*p).cntc=z;
  after=F;
  while ( (after!=NULL) && ( strcmp((*after).ht , x)<0 ) )
  { before=after;
    after=(*after).next;
  };
  (*p).next=after;                // 1
  if (F==after)  F=p;             // 2'
  else  (*before).next=p;         // 2
}
```

- Xóa phần tử đầu tiên trong danh sách: Biến con trỏ p chỉ đến phần tử cần xóa. Ta cho F chỉ đến phần tử tiếp theo.

```
void  DeleteFirst(List  &F)
{ List  p;
  if (F!=NULL)
     { p=F;
       F=(*F).next;               // 1
       delete  p;
     }
}
```

- Xóa phần tử chỉ bởi biến con trỏ t: Biến con trỏ before chỉ đến phần tử đứng ngay trước phần tử cần xóa, biến con trỏ after chỉ đến phần tử đứng ngay sau phần tử chỉ bởi biến before.

```
void  DeleteElement(List  &F, List  t)
{ List before, after;
  after=F;
  while  ( ( after!=NULL) && (after!=t) )
  { before = after;
    after=(*after).next;
  }
  if (after!=NULL)                          // Tìm được phần tử cần xóa
     { if (F==t)   F=(*t).next;             // 1': t chỉ đến phần tử đầu tiên
       else  (*before).next=(*t).next;     // 1: t chỉ đến phần tử phía sau
       delete  t;
     }
}
```

2.3.4. Ưu nhược điểm của danh sách liên kết:

* Ưu điểm:
  - Dùng được cho các danh sách dùng nhiều bộ nhớ.
  - Dùng được các vùng nhớ rời rạc (không bắt buộc phải liên tục).
  - Không cần phải khai báo trước số lượng phần tử.

* Khuyết điểm:
  - Hơi khó viết chương trình.
  - Tốn kém bộ nhớ cho mỗi phần tử vì phải có thêm trường liên kết.

2.4. Danh sách đa liên kết

2.4.1. Định nghĩa:

Danh sách đa liên kết là danh sách có nhiều mối liên kết.

2.4.2. Biểu diễn danh sách đa liên kết:

Xét danh sách đa liên kết các sinh viên gồm họ tên, chiều cao, cân nặng tiêu chuẩn. Trong danh sách này có khi ta cần danh sách được sắp xếp theo thứ tự họ tên tăng dần, cũng có khi ta cần danh sách được sắp xếp theo thứ tự chiều cao tăng dần. Khi đó mỗi phần tử trong danh sách đa liên kết là một bản ghi ngoài các trường ht, cc, cntc chứa dữ liệu của bản thân nó thì còn có thêm 2 trường liên kết. Trường liên kết thứ nhất ta có thể đặt tên là next1 dùng để chỉ đến phần tử đứng ngay sau nó theo thứ tự họ tên, trường liên kết thứ hai next2 chỉ đến phần tử đứng ngay sau nó theo thứ tự chiều cao.

```
typedef  char   infor1[15];
typedef  float  infor2;
typedef  int    infor3;
struct   element
{ infor1   ht;
  infor2   cc;
  infor3   cntc;
  element  *next1, *next2;
};
typedef  element  *List;
List  F1, F2;
```

Biến con trỏ F1 chỉ đến phần tử đầu tiên trong danh sách được sắp theo thứ tự họ tên tăng dần, biến con trỏ F2 chỉ đến phần tử đầu tiên được sắp theo thứ tự chiều cao tăng dần.

Ví dụ ta có danh sách:

	7TĐT	LÊ LI	1.7	65
	8NLB	LÊ BI	1.8	75
	5TĐT	LÊ VI	1.4	35
	6NLB	LÊ NI	1.6	55
	9TĐT	LÊ HI	1.5	45

Danh sách sắp theo thứ tự họ tên tăng dần:

F1=	8NLB	1	LÊ BI	1.8	75

	9TĐT	2	LÊ HI	1.5	45
	7TĐT	3	LÊ LI	1.7	65
	6NLB	4	LÊ NI	1.6	55
	5TĐT	5	LÊ VI	1.4	35

Danh sách sắp theo thứ tự chiều cao tăng dần:

F2=	5TĐT	1	LÊ VI	1.4	35
	9TĐT	2	LÊ HI	1.5	45
	6NLB	3	LÊ NI	1.6	55
	7TĐT	4	LÊ LI	1.7	65
	8NLB	5	LÊ BI	1.8	75

Thì dữ liệu của các phần tử được lưu trữ trong bộ nhớ RAM là:

F2=	5TĐT	LÊ VI	1.4	35	NULL	9TĐT
	7TĐT	LÊ LI	1.7	65	6NLB	8NLB
	9TĐT	LÊ HI	1.5	45	7TĐT	6NLB
	6NLB	LÊ NI	1.6	55	5TĐT	7TĐT
F1=	8NLB	LÊ BI	1.8	75	9TĐT	NULL

2.4.3. Các phép toán trên danh sách đa liên kết:

- Khởi tạo danh sách: Khi mới khởi tạo danh sách là rỗng, ta cho F1 và F2 nhận giá trị NULL.

```
void  Create(List  &F1, List  &F2)
{  F1=NULL;  F2=NULL;
}
```

- Liệt kê các phần tử trong danh sách theo thứ tự họ tên: Ta liệt kê bắt đầu từ phần tử đầu tiên chỉ bởi biến con trỏ F1, và dựa vào trường liên kết next1 để lần lượt liệt kê các phần tử tiếp theo.

```
void  Display1(List  F1)
{ List  p;
  p=F1;
  while  (p != NULL)
  {       cout << "\n " << d << ": " << (*p).ht << setw(5) << (*p).cc << setw(5) << (*p).cntc;
          p=(*p).next1;
  }
}
```

- Liệt kê các phần tử trong danh sách theo thứ tự chiều cao: Ta liệt kê bắt đầu từ phần tử đầu tiên chỉ bởi biến con trỏ F2, và dựa vào trường liên kết next2 để lần lượt liệt kê các phần tử tiếp theo.

```
void  Display2(List  F2)
{ List  p;
  p=F2;
  while  (p != NULL)
  {       cout << "\n " << d << ": " << (*p).ht << setw(5) << (*p).cc << setw(5) << (*p).cntc;
          p=(*p).next2;
  }
}
```

- Tìm phần tử có họ tên x, chiều cao y: Trong danh sách sắp xếp theo thứ tự họ tên có phần tử đầu tiên được chỉ bởi biến con trỏ F1, ta tìm từ phần tử đầu tiên trở đi cho đến khi tìm được phần tử có họ tên là x, chiều cao là y, hoặc gặp phần tử có họ tên lớn hơn x thì không có và dừng. Hàm Search(F1, x, y) kiểu List, tìm và trả về địa chỉ của phần tử đầu tiên tìm được hoặc trả về giá trị NULL nếu tìm không có.

```
List  Search(List  F1, infor1  x, infor2  y)
{ List  p;
  p=F1;
  while ( (p!=NULL) && ( (strcmp((*p).ht,x)<0) ||
                                 (  (strcmp((*p).ht,x)==0) && ((*p).cc!=y) ) ) )
           p= (*p).next1;
if  ( (p!=NULL) && (strcmp((*p).ht,x)==0) && ((*p).cc==y) )              return  p;
  else  return  NULL;
}
```

- Thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào danh sách:

Biến con trỏ p chỉ đến phần tử mới cần thêm vào. Biến con trỏ before chỉ đến phần tử đứng ngay trước phần tử mới theo thứ tự họ tên và thứ tự chiều cao. Biến con trỏ after chỉ đến phần tử đứng ngay sau phần tử được chỉ bởi before.

```
void  InsertElement(List  &F1, List  &F2,infor1  x,infor2  y,infor3  z)
{ List  p, before, after;
  p=new  element;
  strcpy((*p).ht,x); (*p).cc=y; (*p).cntc=z;
  // Tim before va after theo ho ten
  after=F1;
  while ( (after!=NULL) && (strcmp((*after).ht,x)<0) )
  { before=after;
        after=(*after).next1;
  };
  (*p).next1=after;
  if (F1==after)  F1=p;
  else  (*before).next1=p;
  // Tim before va after theo chieu cao
  after=F2;
  while ( (after!=NULL) && ( (*after).cc<y ) )
  { before=after;
        after=(*after).next2;
  };
  (*p).next2=after;
  if (F2==after)  F2=p;
  else  (*before).next2=p;
}
```

- Xóa một phần tử trong danh sách: Tìm rồi xóa phần tử có họ tên x, chiều cao y.

Biến con trỏ p chỉ đến phần tử cần xóa. Biến con trỏ before lần lượt chỉ đến phần tử đứng ngay trước phần tử cần xóa theo thứ tự họ tên và thứ tự chiều cao.

```
void  DeleteElement(List  &F1, List  &F2, infor1  x, infor2  y)
{ List  p, t, before;
  // Tim p
  // Tim before theo ho ten
  p=F1;
  while ( (p!=NULL) && ( (strcmp((*p).ht,x)<0) ||
        ( (strcmp((*p).ht,x)==0) && ((*p).cc!=y) ) ) )
  { before = p;
        p=(*p).next1;
  }
  if  ( (p!=NULL) && (strcmp((*p).ht,x)==0) && ((*p).cc==y) )   // nếu tìm có
  { if  (F1==p)  F1=(*p).next1;
        else  (*before).next1=(*p).next1;
        // Tim before theo chieu cao
        t=F2;
        while  (t!=p)
        { before = t;
          t = (*t).next2;
        }
        if  (F2==p)  F2=(*p).next2;
        else  (*before).next2 = (*p).next2;
        delete  p;
  }
}
```

2.5. Danh sách liên kết kép

2.5.1. Định nghĩa: Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách kết nối với phần tử đứng ngay trước và kết nối với phần tử đứng ngay sau nó.

Với danh sách các phần tử:

7TĐT	1	LÊ LI	1.7	65
8NLB	2	LÊ BI	1.8	75
5TĐT	3	LÊ VI	1.4	35
6NLB	4	LÊ NI	1.6	55
9TĐT	5	LÊ HI	1.5	45

Thì dữ liệu của các phần tử được lưu trữ trong bộ nhớ RAM là:

	5TĐT	8NLB	LÊ VI	1.4	35	6NLB
F=	7TĐT	NULL	LÊ LI	1.7	65	8NLB
	9TĐT	6NLB	LÊ HI	1.5	45	NULL
	6NLB	5TĐT	LÊ NI	1.6	55	9TĐT
	8NLB	7TĐT	LÊ BI	1.8	75	5TĐT

2.5.2. Biểu diễn danh sách liên kết kép:

Các khai báo sau định nghiã một danh sách liên kết kép đơn giản trong đó ta dùng hai con trỏ: pPrev liên kết với phần tử đứng trước và pNext như thường lệ, liên kết với phần tử đứng sau:

```
typedef         struct  tagDNode
{    Data       Info;
struct tagDNode* pPre;              // trỏ đến phần tử đứng trước
struct tagDNode* pNext;             // trỏ đến phần tử đứng sau
}DNODE;
typedef         struct  tagDList
{       DNODE* pHead;               // trỏ đến phần tử đầu danh sách
DNODE* pTail;                               // trỏ đến phần tử cuối danh sách
}DLIST;
```

khi đó, thủ tục khởi tạo một phần tử cho danh sách liên kết kép được viết lại như sau :

```
DNODE*      GetNode(Data x)
{       DNODE *p;
        // Cấp phát vùng nhớ cho phần tử
        p = new DNODE;
        if ( p==NULL)   {
printf("khong du bo nho");
exit(1);
        }
// Gán thông tin cho phần tử p
p ->Info = x;
        p->pPrev = NULL;
        p->pNext = NULL;
        return p;
}
```

2.5.3. Các phép toán trên danh sách liên kết kép:

Tương tự danh sách liên kết đơn, ta có thể xây dựng các thao tác cơ bản trên danh sách liên kết kép (xâu kép). Một số thao tác không khác gì trên xâu đơn. Dưới đây là một số thao tác đặc trưng của xâu kép:

- Chèn 1 phần tử vào danh sách:

Có 4 loại thao tác chèn new_ele vào danh sách:

- Cách 1: Chèn vào đầu danh sách

Cài đặt :

```
void AddFirst(DLIST &l, DNODE* new_ele)
{           if (l.pHead==NULL)  //Xâu rỗng
{    l.pHead = new_ele; l.pTail = l.pHead;
}
else
```

```
{       new_ele->pNext = l.pHead;   // (1)
l.pHead ->pPrev = new_ele;  // (2)
l.pHead = new_ele;    // (3)
}
}
NODE* InsertHead(DLIST &l, Data x)
{       NODE* new_ele = GetNode(x);
 if (new_ele ==NULL)  return NULL;
if (l.pHead==NULL)
{       l.pHead = new_ele; l.pTail = l.pHead;
}
else
{       new_ele->pNext = l.pHead;   // (1)
l.pHead ->pPrev = new_ele;  // (2)
l.pHead = new_ele;    // (3)
}
return new_ele;
}
```

- Cách2: Chèn vào cuối danh sách

  Cài đặt :

```
void AddTail(DLIST &l, DNODE *new_ele)
{       if (l.pHead==NULL)
{       l.pHead = new_ele; l.pTail = l.pHead;
}
else
{       l.pTail->Next = new_ele;      // (1)
new_ele ->pPrev = l.pTail;    // (2)
l.pTail = new_ele;      // (3)
}
}

NODE* InsertTail(DLIST &l, Data x)
{       NODE* new_ele = GetNode(x);
 if (new_ele ==NULL)  return NULL;
if (l.pHead==NULL)
{       l.pHead = new_ele; l.pTail = l.pHead;
}
else
{       l.pTail->Next = new_ele;      // (1)
new_ele ->pPrev = l.pTail;    // (2)
l.pTail = new_ele;      // (3)
}
return new_ele;
}
```

- Cách 3 : Chèn vào danh sách sau một phần tử q

  Cài đặt :

```
void AddAfter(DLIST &l, DNODE* q,DNODE* new_ele)
{       DNODE* p = q->pNext;
        if ( q!=NULL)
{       new_ele->pNext = p;  //(1)
new_ele->pPrev = q;  //(2)
q->pNext = new_ele;  //(3)
if(p != NULL)
p->pPrev = new_ele;  //(4)
```

```
if        (q == l.pTail)
l.pTail = new_ele;
}
else  //chèn vào đầu danh sách
AddFirst(l, new_ele);
}

void InsertAfter(DLIST &l, DNODE *q, Data x)
{         DNODE* p = q->pNext;
          NODE* new_ele = GetNode(x);
if (new_ele ==NULL)  return NULL;
                        if ( q!=NULL)
{         new_ele->pNext = p;  //(1)
new_ele->pPrev = q;  //(2)
q->pNext = new_ele;  //(3)
if(p != NULL)
p->pPrev = new_ele;  //(4)
if(q == l.pTail)
l.pTail = new_ele;
}
else  //chèn vào đầu danh sách
AddFirst(l, new_ele);
}
```

- Cách 4 : Chèn vào danh sách trước một phần tử q

Cài đặt :

```
void   AddBefore(DLIST &l, DNODE q, DNODE* new_ele)
{         DNODE* p = q->pPrev;
if ( q!=NULL)
{         new_ele->pNext = q;  //(1)
new_ele->pPrev = p;  //(2)
q->pPrev = new_ele;  //(3)
if(p != NULL)
p->pNext = new_ele;  //(4)
if(q == l.pHead)
l.pHead = new_ele;
}
else  //chèn vào đầu danh sách
AddTail(l, new_ele);
}

  void InsertBefore(DLIST &l, DNODE q, Data x)
{         DNODE* p = q->pPrev;
          NODE* new_ele = GetNode(x);
  if (new_ele ==NULL)  return NULL;
  if ( q!=NULL)
{         new_ele->pNext = q;  //(1)
new_ele->pPrev = p;  //(2)
q->pPrev = new_ele;  //(3)
if(p != NULL)
p->pNext = new_ele;  //(4)
if(q == l.pHead)
l.pHead = new_ele;
}
else  //chèn vào đầu danh sách
```

```
AddTail(l, new_ele);
}
```

- Xóa 1 phần tử ra khỏi danh sách:

Có 5 loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng.

- Hủy phần tử đầu xâu:

```
Data   RemoveHead(DLIST &l)
{         DNODE         *p;
Data    x = NULLDATA;
  if ( l.pHead != NULL)
{         p = l.pHead; x = p->Info;
l.pHead = l.pHead->pNext;
l.pHead->pPrev = NULL;
delete p;
if(l.pHead == NULL) l.pTail = NULL;
else l.pHead->pPrev = NULL;
}
return x;
}
```

- Hủy phần tử cuối xâu:

```
Data RemoveTail(DLIST &l)
{   DNODE *p;
Data     x = NULLDATA;
if ( l.pTail != NULL)
{p = l.pTail; x = p->Info;
l.pTail = l.pTail->pPrev;
l.pTail->pNext = NULL;
delete p;
if(l.pHead == NULL)     l.pTail = NULL;
else l.pHead->pPrev = NULL;
}
return x;
}
```

- Hủy một phần tử đứng sau phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{     DNODE *p;
      if ( q != NULL)
{         p = q ->pNext ;
if ( p != NULL)
{         q->pNext = p->pNext;
if(p == l.pTail)          l.pTail = q;
else p->pNext->pPrev = q;
delete p;
}
}
else      RemoveHead(l);
}
```

- Hủy một phần tử đứng trước phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{         DNODE         *p;
  if ( q != NULL)
{         p = q ->pPrev;
if ( p != NULL)
{         q->pPrev = p->pPrev;
if(p == l.pHead)         l.pHead = q;
```

```
else p->pPrev->pNext = q;
delete p;
}
}
else     RemoveTail(l);
}
```

- Hủy 1 phần tử có khoá k

```
int RemoveNode(DLIST &l, Data k)
{        DNODE        *p = l.pHead;
NODE *q;
while( p != NULL)
{        if(p->Info == k) break;
p = p->pNext;
}
If       (p == NULL) return 0; //Không tìm thấy k
q = p->pPrev;
if ( q != NULL)
{        p = q ->pNext ;
if ( p != NULL)
{        q->pNext = p->pNext;
if(p == l.pTail)
l.pTail = q;
else p->pNext->pPrev = q;
}
}
else     //p là phần tử đầu xâu
{        l.pHead = p->pNext;
if(l.pHead == NULL)
         l.pTail = NULL;
else     l.pHead->pPrev = NULL;
}
delete p;
return 1;
}
```

* Nhận xét: Danh sách liên kết kép về mặt cơ bản có tính chất giống như xâu đơn. Tuy nhiên nó có một số tính chất khác xâu đơn như sau:

Xâu kép có mối liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên xâu đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối xâu kép, còn trên xâu đơn thao tác này tồn chi phí O(n).

Bù lại, xâu kép tốn chi phí gấp đôi so với xâu đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.

2.6. Danh sách liên kết vòng

Danh sách liên kết vòng là một danh sách đơn (hoặc kép) mà phần tử đứng cuối cùng chỉ đến phần tử đầu tiên trong danh sách. Để biểu diễn ta có thể sử dụng các kỹ thuật biểu diễn như danh sách đơn (hoặc kép).

7TĐT	1	LÊ LI	1.7	65
8NLB	2	LÊ BI	1.8	75
5TĐT	3	LÊ VI	1.4	35
6NLB	4	LÊ NI	1.6	55
9TĐT	5	LÊ HI	1.5	45

Thì dữ liệu của các phần tử được lưu trữ trong bộ nhớ RAM là:

	5TĐT	LÊ VI	1.4	35	6NLB
F=	7TĐT	LÊ LI	1.7	65	8NLB
	9TĐT	LÊ HI	1.5	45	7TĐT

6NLB LÊ NI 1.6 55 9TĐT
8NLB LÊ BI 1.8 75 5TĐT

Ta có thể khai báo xâu vòng như khai báo xâu đơn (hoặc kép). Trên danh sách vòng ta có các thao tác thường gặp sau:

- Tìm phần tử trên danh sách vòng:

Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như là phần tử đầu tiên để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
NODE*       Search(LIST &l, Data  x)
{           NODE      *p;
p = l.pHead;
do
{       if ( p->Info == x)
return p;
p = p->pNext;
}while  (p != l.pHead);          // chưa đi giáp vòng
return p;
}
```

- Thêm phần tử đầu xâu:

```
void    AddHead(LIST &l, NODE *new_ele)
{       if(l.pHead == NULL)  //Xâu rỗng
{       l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
{       new_ele->pNext = l.pHead;
l.pTail->pNext = new_ele;
l.pHead = new_ele;
}
}
```

- Thêm phần tử vào cuối danh sách:

```
void    AddTail(LIST &l, NODE *new_ele)
{       if(l.pHead == NULL)  //Xâu rỗng
{       l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
{       new_ele->pNext = l.pHead;
l.pTail->pNext = new_ele;
l.pTail = new_ele;
}
}
```

- Thêm phần tử sau nút q:

```
void    AddAfter(LIST &l, NODE *q, NODE *new_ele)
{       if(l.pHead == NULL)  //Xâu rỗng
{       l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
{       new_ele->pNext = q->pNext;
q->pNext = new_ele;
if(q == l.pTail)
l.pTail = new_ele;
}
```

```
}
- Xóa phần tử đầu tiên:
void RemoveHead(LIST &l)
{       NODE *p = l.pHead;
        if(p == NULL) return;
if (l.pHead = l.pTail)  l.pHead = l.pTail = NULL;
else
{       l.pHead = p->Next;
if(p == l.pTail)
l.pTail->pNext = l.pHead;
}
delete p;
}
- Xóa phần tử đứng sau nút q:
void RemoveAfter(LIST &l, NODE *q)
{       NODE *p;
        if(q != NULL)
{       p = q ->Next ;
if ( p == q)  l.pHead = l.pTail = NULL;
else
{       q->Next = p->Next;
if(p == l.pTail)
l.pTail = q;
}
delete p;
}
}
```

Nhận xét: Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách.

2.7. Danh sách hạn chế:

2.7.1. Khái niệm:

Danh sách hạn chế là danh sách mà các phép toán chỉ được thực hiện ở một phạm vi nào đó của danh sách, trong đó thường người ta chỉ xét các phép thêm vào hoặc loại bỏ chỉ được thực hiện ở đầu danh sách. Danh sách hạn chế có thể được biểu diễn bằng danh sách đặc hoặc bằng danh sách liên kết. Có 2 loại danh sách hạn chế phổ biến là ngăn xếp và hàng đợi.

2.7.2. Ngăn xếp (hay chồng hoặc Stack):

2.7.2.1. Định nghĩa:

Ngăn xếp là một danh sách mà các phép toán thêm vào hoặc loại bỏ chỉ được thực hiện ở cùng một đầu của danh sách. Đầu này gọi là đỉnh của ngăn xếp. Như vậy phần tử thêm vào đầu tiên sẽ được lấy ra cuối cùng.

2.7.2.2. Biểu diễn ngăn xếp bằng danh sách liên kết:

Xét ngăn xếp các sinh viên gồm họ tên, chiều cao, cân nặng tiêu chuẩn.

```
struct   element
{ infor1    ht;
  infor2    cc;
  infor3    cntc;
  element    *next;
};
typedef    element   *Stack;
Stack   S;
```

Ta dùng danh sách liên kết để chứa các phần tử trong ngăn xếp. Một phần tử trong danh sách liên kết chứa một phần tử trong ngăn xếp, theo thứ tự là phần tử đầu tiên trong danh sách liên kết chứa phần tử ở đỉnh của ngăn xếp.

Stack là kiểu con trỏ chỉ đến 1 phần tử trong danh sách.

Biến con trỏ S chỉ đến phần tử đầu tiên trong danh sách.

2.7.2.3. Các phép toán trên ngăn xếp được biểu diễn bằng danh sách liên kết:

- Khởi tạo ngăn xếp: Khi mới khởi tạo, ngăn xếp là rỗng ta cho S nhận giá trị NULL.

```
void  Create(Stack   &S)
{  S=NULL;
}
```

- Phép liệt kê các phần tử trong ngăn xếp:

```
void  Display(Stack  S)
{ Stack  p;
  p=S; int  d=0;
  while  (p != NULL)
  {   d++;
      cout <<"\n "<<d<<": "<<(*p).ht<<setw(5)<<(*p).cc << setw(5) << (*p).cntc;
      p=(*p).next;
  }
}
```

- Phép thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào ngăn xếp (thêm vào đầu ngăn xếp):

```
void  InsertStack(Stack  &S, infor1  x, infor2  y, infor3  z)
{ Stack  p;
  p=new  element;
  strcpy((*p).ht,x); (*p).cc=y; (*p).cntc=z;
  (*p).next=S;
  S=p;
}
```

- Phép xóa một phần tử trong ngăn xếp (xóa phần tử đầu tiên).

```
void  DeleteStack(Stack  &S)
{ Stack  p;
  if (S!=NULL)
     { p=S;
       S=(*p).next;                  // 1
       delete  p;
     }
}
```

Bài tập: Dùng ngăn xếp, viết chương trình nhập số nguyên dương n, rồi đổi ra số nhị phân.

2.7.3. Hàng đợi (hay Queue):

2.7.3.1. Định nghĩa:

Hàng đợi là danh sách mà phép thêm vào được thực hiện ở đầu này, còn phép loại bỏ được thực hiện ở đầu kia của danh sách. Như vậy phần tử thêm vào đầu tiên sẽ được lấy ra đầu tiên.

2.7.3.2. Biểu diễn hàng đợi bằng danh sách liên kết:

Xét hàng đợi các sinh viên gồm họ tên, chiều cao, cân nặng tiêu chuẩn.

```
struct   element
{ infor1   ht;
  infor2    cc;
  infor3    cntc;
  element    *next;
};
typedef    element   *Queue;
Queue    Front, Rear;
```

Ta dùng danh sách liên kết để chứa các phần tử trong hàng đợi, một phần tử trong danh sách liên kết chứa một phần tử trong hàng đợi theo thứ tự là phần tử đầu tiên trong danh sách liên kết chứa phần tử đầu tiên trong hàng đợi.

Biến con trỏ Front chỉ đến phần tử đầu tiên của danh sách liên kết, đó chính là phần tử đầu tiên của hàng đợi.

Biến con trỏ Rear chỉ đến phần tử cuối cùng của danh sách liên kết, đó chính là phần tử cuối cùng của hàng đợi.

2.7.3.3. Các phép toán trên hàng đợi được biểu diễn bằng danh sách liên kết:

- Khởi tạo hàng đợi: Khi mới khởi tạo, hàng đợi là rỗng ta cho Front và Rear nhận giá trị NULL.

```
void  Create(Queue   &Front, Queue  &Rear)
{  Front=NULL;  Rear=NULL;
}
```

- Liệt kê các phần tử trong hàng đợi:

```
void  Display(Queue  Front, Queue  Rear)
{ Queue p; int  d=0;
  p=Front;
  while  (p != NULL)
  { d++;
    cout <<"\n "<<d<<": "<<(*p).ht<<setw(5)<<(*p).cc << setw(5) << (*p).cntc;
    p=(*p).next;
  }
}
```

- Thêm một phần tử có họ tên x, chiều cao y, cân nặng tiêu chuẩn z vào hàng đợi (vào cuối danh sách liên kết).

```
void  InsertQueue(Queue  &Front, Queue  &Rear, infor1  x, infor2  y, infor3  z)
{ Queue  p;
  p=new  element;
  strcpy((*p).ht,x); (*p).cc=y; (*p).cntc=z;
  (*p).next=NULL;                       // 1
   if  (Front==NULL)  Front=p;          // 2'
    else  (*Rear).next=p;               // 2
   Rear=p;                              // 3
}
```

- Phép xóa một phần tử trong hàng đợi (xóa phần tử đầu tiên) .

```
void  DeleteQueue(Queue  &Front, Queue  &Rear)
{ Queue   p;
  if  (Front != NULL)
     { p=Front;
          Front=(*Front).next;                          // 1
       if  (Front==NULL)  Rear=NULL;          // 2
       delete  p;
     }
}
```

--- HẾT CHƯƠNG 2 ---

Chương 3: CÂY

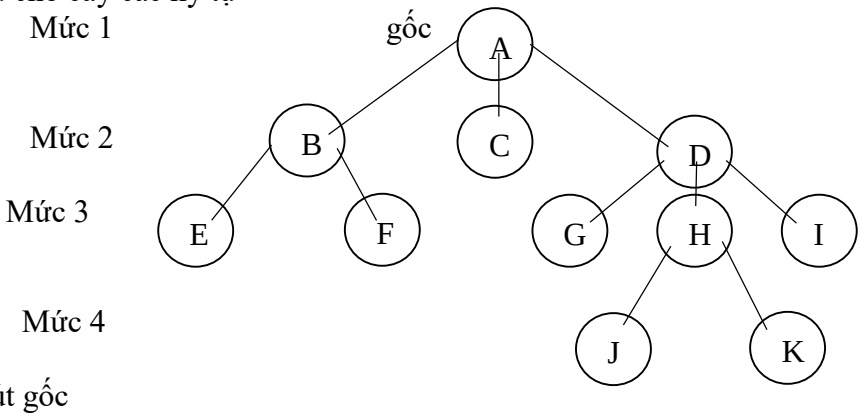
Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu cây. Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn, nó được áp dụng để tổ chức thông tin trong các hệ cơ sở dữ liệu, để mô tả cấu trúc cú pháp của các chương trình nguồn khi xây dựng các chương trình dịch. Rất nhiều các bài toán mà ta gặp trong các lĩnh vực khác nhau được quy về việc thực hiện các phép toán trên cây. Trong chương này chúng ta sẽ trình bày định nghĩa và các khái niệm cơ bản về cây. Chúng ta cũng sẽ xét các phương pháp biểu diễn cây và sự thực hiện các phép toán cơ bản trên cây. Sau đó chúng ta sẽ nghiên cứu kỹ một dạng cây đặc biệt, đó là cây tìm kiếm nhị phân.

3.1. Một số khái niệm

3.1.1. Các định nghĩa:

- Cây: là một tập hợp hữu hạn các phần tử, mỗi phần tử gọi là một nút (Node), trong đó có một nút đặc biệt gọi là gốc (Root), giữa các nút có một quan hệ phân cấp gọi là quan hệ cha con.

Ví dụ cho cây các ký tự

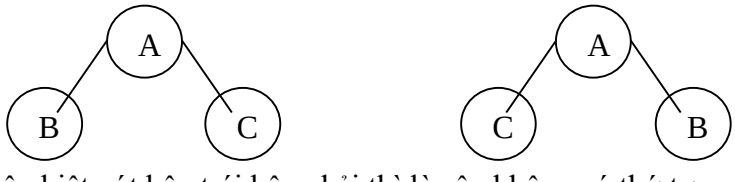


A: nút gốc

A là nút cha của các nút B, C, D.

Các nút B, C, D là các nút con của A.

- Cây rỗng: cây không có nút nào cả.
- Cấp của nút: số nút con của nó, vd nút B có cấp là 2.
- Cấp của cây: cấp lớn nhất của các nút có trên cây. Cây có cấp n gọi là cây n phân, ví dụ cây trên là cây tam phân.
- Lá: nút có cấp là 0, ví dụ các là E , F, C, G, J, K
- Mức: Nút gốc có mức là 1. Nút cha có mức i thì nút con có mức i+1.
- Chiều cao của cây: mức lớn nhất trên cây, ví dụ cây trên có chiều cao 4.
- Nút trước, nút sau: Nút x là nút trước của nút y nếu cây con gốc x có chứa nút y, khi đó y là nút sau của nút x. ví dụ D là nút trước của nút J.
- Đường đi (path): Dãy nút $u_1, u_2, \ldots u_k$ mà nút bất kỳ u_i là cha của nút u_{i+1} thì dãy đó là đường đi từ nút u_1 đến nút u_k
- Độ dài đường đi: là số cạnh có trên đường đi, ví dụ dãy DHJ là đường đi từ nút D đến nút J với độ dài là 2.
- Cây có thứ tự (ordered tree): là cây mà nếu ta thay đổi vị trí của các cây con thì ta có một cây mới. Như vậy nếu ta đổi các nút bên trái và bên phải thì ta được một cây mới, ví dụ sau đây là 2 cây khác nhau:



- Cây mà không phân biệt nút bên trái bên phải thì là cây không có thứ tự.
- Rừng: là tập hợp hữu hạn các cây phân biệt.

3.1.2. Các cách biểu diễn cây:

- Biểu diễn cây bằng đồ thị.
- Biểu diễn cây bằng giản đồ.
- Biểu diễn cây bằng các cặp dấu ngoặc lồng nhau.

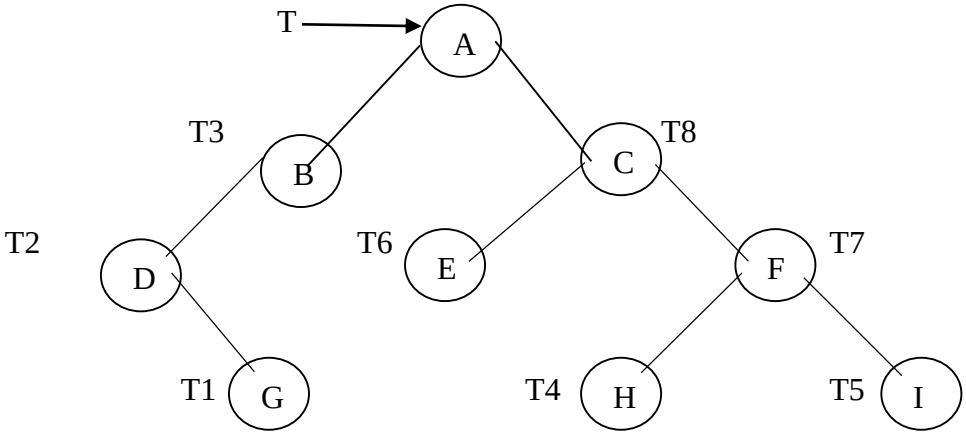
- Biểu diễn cây bằng phương pháp căn lề.
- Biểu diễn cây bằng phương pháp chỉ số.

3.2. Cây nhị phân:

3.2.1. Định nghĩa và tính chất:

3.2.1.1. Định nghĩa:

Cây nhị phân là cây mà mọi nút trên cây có tối đa 2 con. Trong cây nhị phân người ta có phân biệt nút con trái và nút con phải, như vậy cây nhị phân là cây có thứ tự.



3.2.1.2. Các dạng đặc biệt của cây nhị phân:

Cây nhị phân suy biến là cây lệch trái hoặc cây lệch phải.

Cây zic-zắc.

Cây nhị phân hoàn chỉnh: các nút ứng với các mức trừ mức cuối cùng đều có 2 con.

Cây nhị phân đầy đủ: có các nút tối đa ở cả mọi mức.

Cây nhị phân đầy đủ là một trường hợp đặc biệt của cây nhị phân hoàn chỉnh.

3.2.1.3. Các tính chất:

- Mức i của cây nhị phân có tối đa 2^{i-1} nút.
- Cây nhị phân với chiều cao i có tối đa 2^i-1 nút.
- Cây nhị phân với n nút thì chiều cao $h>\log_2(n)$.

3.2.2. Biểu diễn cây nhị phân:

3.2.2.1. Biểu diễn cây nhị phân bằng danh sách đặc (mảng):

Với cây nhị phân hoàn chỉnh, ta có thể đánh số thứ tự cho các nút trên cây từ gốc trở xuống và từ trái sang phải bắt đầu từ 0 trở đi. Khi đó nút thứ i có nút con trái là thứ 2i+1 và có nút con phải thứ 2i+2.

Khi đó ta dùng một mảng 1 chiều để lưu trữ các nút trên cây nhị phân, trong đó phần tử thứ i của mảng chứa nút thứ i của cây nhị phân.

Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường: trường data mô tả thông tin gắn với mỗi đỉnh, trường left chỉ đỉnh con trái, trường right chỉ đỉnh con phải. Giả sử các đỉnh của cây được đánh số từ 0 đến max-1, khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau.

Khai báo:

```
const  int  max= . . . ;
struct   element
{ char    data;          // trường chứa dữ liệu
  int    left;
  int    right;
};
typedef   node  Tree[max];
Tree  V;
```

Ví dụ cây nhị phân đã cho ở trên được biểu diễn như sau:

	data	left	right
0	A	1	2
1	B	3	-1
2	C	5	6
3	D	-1	8
4		-1	-1

5	E	-1	-1
6	F	13	14
7		-1	-1
8	G	-1	-1
9		0	0
10		0	0

3.2.2.2. Biểu diễn cây nhị phân bằng danh sách liên kết:

Chúng ta còn có thể sử dụng con trỏ để cài đặt cây nhị phân. Trong cách này mỗi bản ghi element biểu diễn một nút của cây gồm trường data chứa dữ liệu của bản thân nút, ngoài ra còn có thêm 2 trường liên kết left, right lần lượt chỉ đến nút con trái và nút con phải của nó. Ta có khai báo sau:

```
struct    element
{ char   data;          // trường chứa dữ liệu
  element   *left , *right;
};
typedef   element  *Tree;
Tree   T;                  // hoặc   element   *T;
```

Kiểu con trỏ Tree chứa địa chỉ của 1 nút của cây nhị phân.
Biến con trỏ T kiểu Tree chứa địa chỉ của nút gốc của cây nhị phân .

3.2.3. Các phép toán trên cây nhị phân được biểu diễn bằng danh sách liên kết.

a. Khởi tạo: Khi mới khởi tạo, cây là rỗng ta cho T nhận giá trị NULL

```
void  Create(Tree  &T)
{  T = NULL;
}
```

b. Các phép duyệt cây:

Phép duyệt cây là liệt kê tất cả các nút có trên cây mỗi nút đúng 1 lần theo một thứ tự nào đó. Thường có 3 phép duyệt cây là:

- Duyệt cây theo thứ tự trước (đối với gốc): Kiểu duyệt này trước tiên thăm nút gốc, sau đó thăm các nút của cây con trái rồi đến các nút của cây con phải. . Gốc
. Cây con trái
. Cây con phải

Ví dụ khi duyệt cây nhị phân đã cho theo thứ tự trước ta được dãy nút A B D G C E F H I
Hàm con duyệt trước có thể trình bày bằng giải thuật đệ quy là:

```
void  DuyetTruoc(Tree  T)
{  if  (T != NULL)
        {  cout << (*T).data;
           DuyetTruoc( (*T).left );
           DuyetTruoc( (*T).right );
        }
}
```

- Duyệt cây theo thứ tự giữa: Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm nút gốc, rồi thăm các nút của cây con phải.
. Cây con trái
. Gốc
. Cây con phải

Ví dụ khi duyệt cây nhị phân đã cho theo thứ tự giữa ta được dãy nút D G B A E C H F I
Hàm con duyệt giữa có thể trình bày bằng giải thuật đệ quy là:

```
void  DuyetGiua(Tree  T)
{  if  (T != NULL)
        {  DuyetGiua( (*T).left );
          cout << (*T).data;
           DuyetGiua( (*T).right );
        }
}
```

- Duyệt cây theo thứ tự sau:Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm các nút của cây con phải, cuối cùng thăm nút gốc.

. Cây con trái
. Cây con phải
. Gốc

Ví dụ khi duyệt cây nhị phân đã cho theo thứ tự sau ta được dãy nút G D B E H I F C A
Hàm con duyệt sau có thể trình bày bằng giải thuật đệ quy là:

```
void  DuyetSau(Tree  T)
{  if  (T != NULL)
        {  DuyetSau( (*T).left );
           DuyetSau( (*T).right );
           cout << (*T).data;
        }
}
```

c. Hàm tạo cây nhị phân mới từ 2 cây nhị phân cho trước:
Cho trước 2 cây nhị phân có gốc lần lượt chỉ bởi L và R. Cho trước giá trị x. Ta sẽ tạo ra cây nhị phân mới có gốc có giá trị là x, và có cây con trái là cây nhị phân thứ nhất chỉ bởi L, có cây con phải là cây nhị phân thứ hai chỉ bởi R. Hàm NODE(x, L, R) kiểu Tree, trả về địa chỉ của gốc của cây nhị phân mới được tạo ra.

```
Tree  NODE(char  x, Tree  L, Tree  R)
{  Tree  p;
   p=new  element;
   (*p).data=x; (*p).left=L; (*p).right=R;
   return  p;
}
```

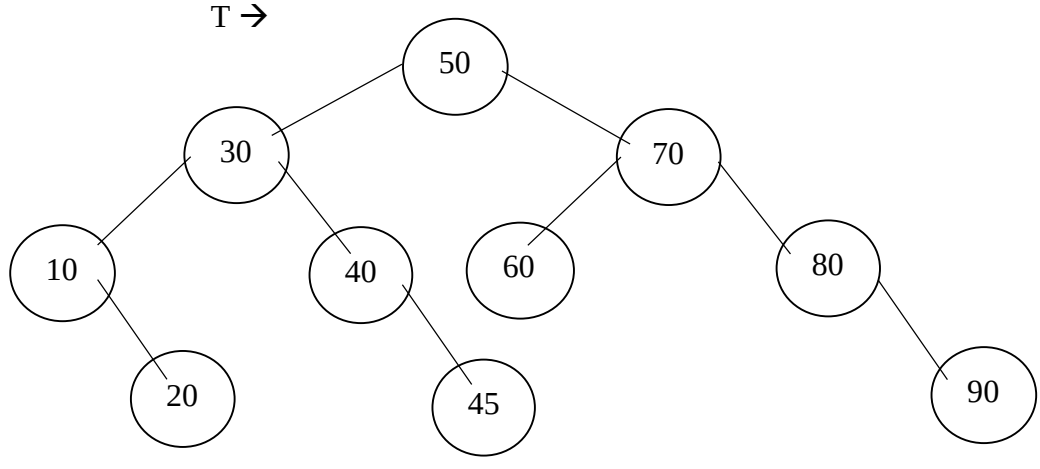
d. Tạo trực tiếp cây nhị phân:

```
T1=NODE('G', NULL, NULL);
T2=NODE('D', NULL, T1);
T3=NODE('B', T2, NULL);
T4=NODE('H', NULL, NULL);
T5=NODE('I', NULL, NULL);
T6=NODE('E', NULL, NULL);
T7=NODE('F', T4, T5);
T8=NODE('C', T6, T7);
T=NODE('A', T3, T8);
```

3.3. Cây nhị phân tìm kiếm

3.3.1. Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân mà tại mỗi nút bất kỳ thì khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.



Nhờ ràng buộc về khóa trên CNPTK, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây việc tìm kiếm trở nên nhanh đáng kể.

3.3.2. Các phép toán trên cây nhị phân tìm kiếm:

- Duyệt cây:

Thao tác duyệt cây trên cây nhị phân tìm kiếm hoàn toàn giống như trên cây nhị phân. Chỉ có một lưu ý nhỏ là khi duyệt theo thứ tự giữa, trình tự các nút duyệt qua sẽ cho ta một dãy các nút theo thứ tự tăng dần của khóa.

- Tìm một phần tử x trên cây:

```
TNODE   *searchNode(TREE T, Data X)
{
if    (T){
if  ( T->.Key = = X)    return T;
if(T->Key > X)
return   searchNode(T->pLeft, X);
else
return   searchNode(T->pRight, X);
}
return NULL;
}
```

Ta có thể xây dựng một hàm tìm kiếm tương đương không đệ qui như sau:

```
Tree   searchNode(Tree   T, infor   x)          Hùng    →    Dũng=*Hùng
{  Tree   p=T;                                              p  →  *p
  while   (p!=NULL)
  {  if   (x==(*p).data)  return   p;
    else   if   (x<(*p).data)   p=(*p).left;
                          else   p=(*p).right;
  }
}
```

Hoặc:

```
TNODE * searchNode(TREE Root,  Data x)
{         NODE *p = Root;
   while (p != NULL)
{ if(x == p->Key) return p;
else
if(x < p->Key)p = p->pLeft;
else      p = p->pRight;
}
return NULL;
}
```

Dễ dàng thấy rằng số lần so sánh tối đa phải thực hiện để tìm phần tử X là h, với h là chiều cao của cây. Như vậy thao tác tìm kiếm trên CNPTK có n nút tốn chi phí trung bình khoảng $O(\log_2 n)$

- Thêm một phần tử x vào cây:

Việc thêm một phần tử X vào cây phải bảo đảm 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 thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.

```
void   insertNode(Tree   &T,  infor   x)
{  if   (T==NULL)
                  {  T=new   element;
                    (*T).data=x; (*T).left=NULL; (*T).right=NULL;
                  }
  else  {  if    (x<(*T).data)   insertNode((*T).left, x);
                        else   if   (x>(*T).data)   insertNode((*T).right, x);
                  }
}
```

Ví dụ, ban đầu cây rỗng, ta lần lượt nhập 50 , 30 , 40, 70, 20 , … thì cây như thế nào?

- Xóa một phần tử trên cây:

Việc xóa một phần tử ra khỏi cây phải đảm bảo điều kiện ràng buộc của cây nhị phân tìm kiếm.
Có 3 trường hợp khi hủy nút X có thể xảy ra:

- . X là nút lá.
- . X chỉ có 1 con (trái hoặc phải).
- . X có đủ cả 2 con .

- **Trường hợp thứ nhất:** chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác.
- **Trường hợp thứ hai:** trước khi hủy X ta móc nối cha của X với con duy nhất của X.
- **Trường hợp cuối cùng:** ta không thể hủy trực tiếp do X có đủ 2 con ⇒ Ta sẽ hủy gián tiếp. Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y để thế cho phần tử X. Phần tử Y này có tối đa một con. Thông tin lưu tại Y sẽ được chuyển lên lưu tại X. Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu.

Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là CNPTK.
Có 2 phần tử thỏa mãn yêu cầu:

- . Phần tử nhỏ nhất (trái nhất) trên cây con phải.
- . Phần tử lớn nhất (phải nhất) trên cây con trái.

Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình. Ở đây, chúng ta sẽ chọn phần tử (phải nhất trên cây con trái làm phần tử thế mạng.

Sau khi hủy phần tử X=18 ra khỏi cây tình trạng của cây sẽ như trong hình dưới đây (phần tử 23 là phần tử thế mạng):
Hàm delNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây:

```
int  delNode(TREE &T, Data X)
{       if  (T==NULL)          return 0;
    if(T->Key > X)
        return   delNode (T->pLeft, X);
    if(T->Key < X)
        return delNode (T->pRight, X);
    else
        { //T->Key == X
            TNode*    p = T;
            if(T->pLeft == NULL)
                T       = T->pRight;
            else   if(T->pRight == NULL)
                            T    = T->pLeft;
                    else {          //T có cả 2 con
                                TNode*    q = T->pRight;
                                searchStandFor(p, q);
                            }
            delete p;
        }
}
```

Trong đó, hàm searchStandFor được viết như sau:

```
//Tìm phần tử thế mạng cho nút p
void searchStandFor(TREE &p, TREE &q)
{       if(q->pLeft)
            searchStandFor(p, q->pLeft);
    else        {       p->Key          = q->Key;
                p       = q;
                q       = q->pRight;
            }
}
```

- Tạo một cây nhị phân tìm kiếm:

Ta có thể tạo một cây nhị phân tìm kiếm bằng cách lặp lại quá trình thêm 1 phần tử vào một cây rỗng.

- Xóa toàn bộ cây:

Việc toàn bộ cây có thể được thực hiện thông qua thao tác duyệt cây theo thứ tự sau. Nghĩa là ta sẽ hủy cây con trái, cây con phải rồi mới hủy nút gốc.

```
void  removeTree(TREE &T)
{       if  (T)
        {       removeTree(T->pLeft);
                removeTree(T->pRight);
                delete(T);
        }
}
```

3.3.3. Đánh giá:

Tất cả các thao tác searchNode, insertNode, delNode trên CNPTK đều có độ phức tạp trung bình O(h), với h là chiều cao của cây.

Trong trong trường hợp tốt nhất, CNPTK có n nút sẽ có độ cao h = $\log_2(n)$. Chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.

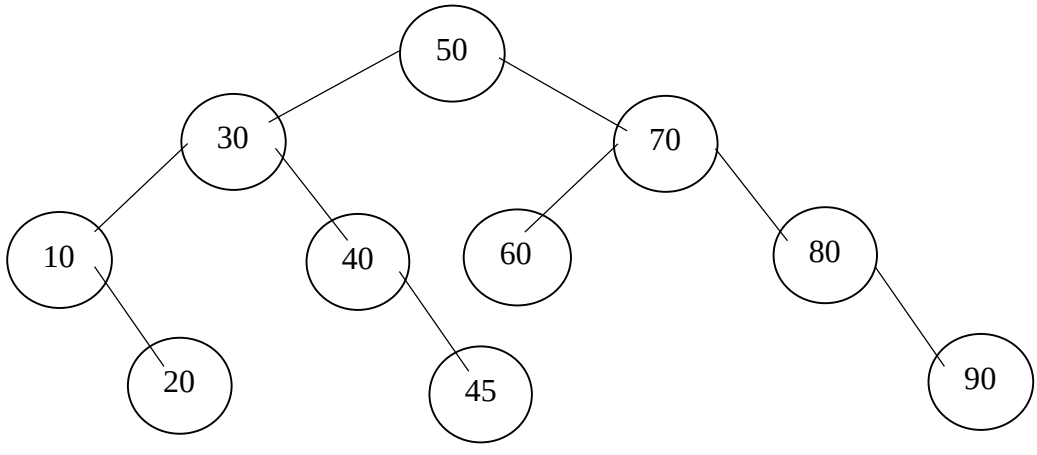
Tuy nhiên, trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 DSLK (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp O(n). Vì vậy cần có cải tiến cấu trúc của CNPTK để đạt được chi phí cho các thao tác là $\log_2(n)$

3.4. Cây nhị phân cân bằng:

3.4.1. Cây cân bằng hoàn toàn (về số nút):

3.4.1.1. Định nghĩa:

Cây cân bằng hoàn toàn là cây nhị phân tìm kiếm mà tại mỗi nút của nó thì số nút của cây con trái và số nút của cây con phải chênh lệch không quá một. Ví dụ cho cây cân bằng hoàn toàn sau:



3.4.1.2. Đánh giá:

Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay hủy các nút trên cây có thể làm cây mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ nhanh. Với cây cân bằng hoàn toàn, trường hợp xấu nhất ta chỉ phải tìm qua $\log_2 n$ phần tử (n là số nút trên cây).

CCBHT có n nút có chiều cao h >= $\log_2 n$. Đây chính là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên CTDL này.

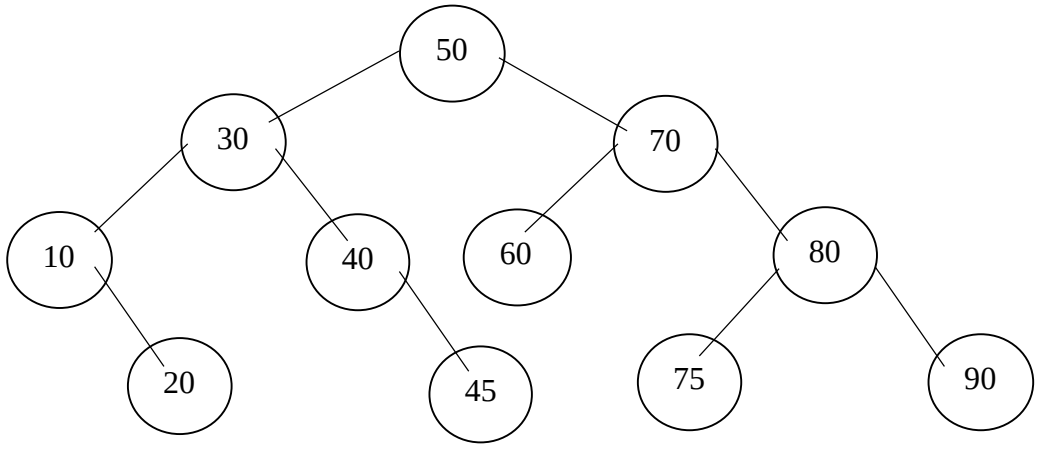
Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một CTDL khác có đặc tính giống CCBHT nhưng ổn định hơn.

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 phải bảo đảm chi phí cho thao tác tìm kiếm đạt ở mức O($\log_2 n$).

3.4.2. Cây cân bằng (về độ cao):

3.4.2.1. Định nghĩa:

Cây cân bằng là cây nhị phân tìm kiếm mà tại mỗi nút của nó thì độ cao của cây con trái và độ cao của cây con phải chênh lệch không quá 1. Ví dụ cho cây cân bằng sau:



Dễ dàng thấy cây cân bằng hoàn toàn là cây cân bằng. Còn cây cân bằng thì chưa chắc là cây cân bằng hoàn toàn. Ví dụ cây đã cho là cây cân bằng, nhưng không phải là cây cân bằng hoàn toàn.

3.4.2.2. Lịch sử cây cân bằng (AVL tree)

AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng Adelson-Velskii và Landis (1962). Vì lý do này, người ta gọi cây nhị phân cân băng là cây AVL. Tù nay về sau, chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Từ khi được giới thiệu, cây AVL đã nhanh chóng tìm thấy 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 CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, …

3.4.2.3. Chiều cao của cây AVL

Một vấn đề quan trọng, như đã đề cập đến ở phần trước, là ta pjải 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 phải có tối thiểu bao nhiêu nút ?

Gọi N(h) là 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 1 cây con AVL tối thiểu chiều cao h-1 và 1 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:

$N(h) > 2N(h-2)$

$N(h) > 2^2N(h-4)$

…

$N(h) > 2^{iN}(h-2i)$

$N(h) > 2^{h/2-1}$

$h < 2\log_2(N(h)) + 2$

Như vậy, cây AVL có chiều cao $O(\log_2(n))$.

3.4.2.4. Cấu trúc dữ liệu cho cây AVL

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 một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:

CSCB(p) = 0 <=> Độ cao cây trái (p) = Độ cao cây phải (p)

CSCB(p) = 1 <=> Độ cao cây trái (p) < Độ cao cây phải (p)

CSCB(p) =-1 <=> Độ cao cây trái (p) > Độ cao cây phải (p)

Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau:

p->balFactor = CSCB(p);

Độ cao cây trái (p) ký hiệu là hL

Độ cao cây phải(p) ký hiệu là hR

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau:

```
typedef struct tagAVLNode   {
        char balFactor; //Chỉ số cân bằng
        Data    key;
        struct tagAVLNode*  pLeft;
        struct tagAVLNode*  pRight;
}AVLNode;
typedef AVLNode     *AVLTree;
```

Để tiện cho việc trình bày, ta định nghĩa một số hăng số sau:

```
#define LH -1 //Cây con trái cao hơn
#define EH -0 //Hai cây con bằng nhau
#define RH 1  //Cây con phải cao hơn
```

3.4.2.5. Đánh giá cây AVL

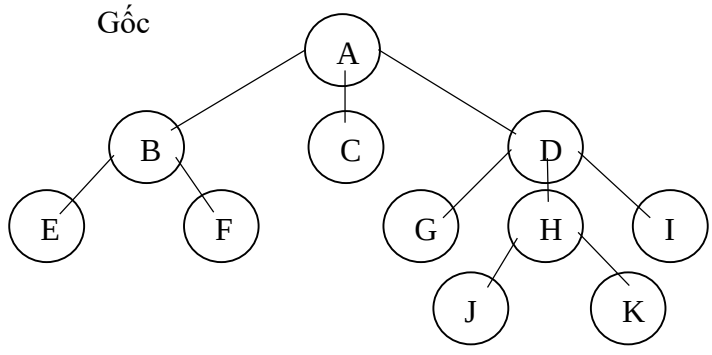
- Cây cân bằng là CTDL ổn định hơn hẳn CCBHT vì chỉ khi thêm hủy làm cây thay đổi chiều cao các trường hợp mất cân bằng mới có khả năng xảy ra.
- Cây AVL với chiều cao được khống chế sẽ cho phép thực thi các thao tác tìm thêm hủy với chi phí O ($\log_2(n)$) và bảo đảm không suy biến thành O(n).

3.5. Cây tổng quát:

3.5.1. Định nghĩa:

Cây tổng quát là cây mà các nút trên cây có số con là bất kỳ.

Ví dụ cho cây tam phân các ký tự:



3.5.2. Biểu diễn cây tổng quát bằng danh sách liên kết:

Mỗi nút của cây là một bản ghi, ngoài các trường chứa dữ liệu của bản thân nó, còn có thêm các trường liên kết khác lưu trữ địa chỉ của các nút con

3.5.3. Các phép duyệt cây tổng quát

Tương tự như cây nhị phân, đối với cây tổng quát cũng có 3 phép duyệt cơ bản là:

- Duyệt cây theo thứ tự trước (đối với gốc): Kiểu duyệt này trước tiên thăm nút gốc, sau đó lần lượt thăm các nút của các cây con.

. Gốc
. Cây con trái nhất
. Các cây con phải

Ví dụ khi duyệt cây trên theo thứ tự trước ta được dãy nút:

A B E F C D G H J K I

- Duyệt cây theo thứ tự giữa: Kiểu duyệt này trước tiên thăm nút các nút của cây con trái nhất, sau đó thăm nút gốc rồi đến các cây con phải.

. Cây con trái nhất
. Gốc
. Cây con phải

Ví dụ khi duyệt cây trên theo thứ tự giữa ta được dãy nút:

E B F A C G D J H K I

- Duyệt cây theo thứ tự sau: Kiểu duyệt này trước tiên thăm các nút của cây con trái nhất, sau đó thăm các nút của các cây con phải, cuối cùng thăm nút gốc.

. Cây con trái nhất
. Các cây con phải
. Gốc

Ví dụ khi duyệt cây trên theo thứ tự sau ta được dãy nút:

E F B C G J K H I D A

3.5.4. Cây nhị phân tương đương:

Nhược điểm của các cấu trúc cây tổng quát là bậc của các nút trên cây có thể dao động trong một biên độ lớn ⇒ việc biểu diễn gặp nhiều khó khăn và lãng phí. Hơn nữa, việc xây dựng các thao tác trên cây tổng quát phức tạp hơn trên cây nhị phân nhiều. Vì vậy, thường nếu không quá cần thiết phải sử dụng cây tổng quát, người ta chuyển cây tổng quát thành cây nhị phân.

Ta có thể biến đổi một cây tổng quát bất kỳ thành một cây nhị phân tương đương theo qui tắc sau:

- Giữ lại nút con trái nhất làm nút con trái.
- Chuyển nút em kề phải thành 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.

Một ví dụ quen thuộc trong tin học về ứng dụng của duyệt theo thứ tự sau là việc xác định tổng kích thước của một thư mục trên đĩa

Một ứng dụng quan trọng khác của phép duyệt cây theo thứ tự sau là việc tính toán giá trị của biểu thức dựa trên cây biểu thức

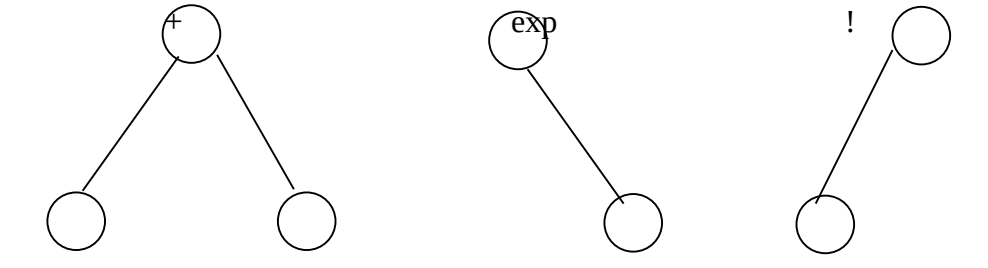
(3 + 1)×3/(9 – 5 + 2) – (3×(7 – 4) + 6) = –13

Một ví dụ hay về cây nhị phân là cây biểu thức. Cây biểu thức là cây nhị phân gắn nhãn, biểu diễn cấu trúc của một biểu thức (số học hoặc logic). Mỗi phép toán hai toán hạng (chẳng hạn, +, -, *, /) được biểu diễn bởi cây nhị phân, gốc của nó chứa ký hiệu phép toán, cây con trái biểu diễn toán hạng bên trái, còn cây con phải biểu diễn toán hạng bên phải. Với các phép toán một hạng như ‘phủ định’ hoặc ‘lấy giá trị đối’ 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 bên minh họa một số cây biểu thức.

Ta có nhận xét rằng, nếu đi qua cây biểu thức theo thứ tự trước ta sẽ được biểu thức Balan dạng prefix (ký hiệu phép toán đứng trước các toán hạng). Nếu đi

qua cây biểu thức theo thứ tự sau, ta có biểu thức Balan dạng postfix (ký hiệu phép toán đứng sau các toán hạng); còn theo thứ tự giữa ta nhận được cách viết thông thường của biểu thức (ký hiệu phép toán đứng giữa hai toán hạng).



--- HẾT CHƯƠNG 3 ---

Chương 4: SẮP XẾP THỨ TỰ DỮ LIỆU

4.1. Bài toán sắp xếp thứ tự:

Sắp xếp là quá trình xử lý một danh sách các phần tử (hoặc các mẫu tin) để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên nội dung thông tin lưu giữ tại mỗi phần tử.

Cho trước một dãy gồm n số thực A[1], A[2], ... , A[n] được lưu trữ trong cấu trúc dữ liệu mảng:

```
float   A[100];
```

Sắp xếp dãy số A[1], A[2], ... , A[n] là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới $A[k_1]$, $A[k_2]$, ... , $A[k_n]$ có thứ tự (giả sử xét thứ tự tăng) nghĩa là $A[k_{i-1}] <= A[k_i]$. Mà để quyết định được những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Chính vì vậy, hai thao tác so sánh và gán là các thao tác cơ bản của hầu hết các thuật toán sắp xếp.

Khi xây dựng một thuật toán sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của thuật toán. Đối với các dãy số được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được đặt nặng, do vậy những thuật toán sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả ngoài vùng nhớ lưu trữ dãy số ban đầu thường ít được quan tâm. Thay vào đó, các thuật toán sắp xếp trực tiếp trên dãy số ban đầu - gọi là các thuật toán sắp xếp tại chỗ - lại được đầu tư phát triển. Phần này giới thiệu một số giải thuật sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp nội.

Trước hết ta có thể xây dựng hàm con hoan_vi(float *x, float *y) dùng để hoán vị 2 phần tử chỉ bởi x và y cho nhau bằng cách sử dụng biến trung gian tg có cùng kiểu và thực hiện 3 phép gán:

```
void   hoan_vi(float   *x, float   *y)
{  float   tg;
   tg=*x;
   *x=*y;
   *y=tg;
}
```

Các chương trình dưới đây đều sắp xếp mảng theo thứ tự tăng dần. Nếu muốn sắp theo thứ tự giảm dần thì chỉ cần đổi chiều dấu bất đẳng thức.

4.2. Sắp thứ tự nội:

4.2.1. Sắp thứ tự bằng phương pháp lựa chọn trực tiếp:

- Giải thuật:

Ta thấy rằng, nếu mảng có thứ tự, phần tử A[i] luôn là min(A[i], A[i+1], ... , A[n]). Ý tưởng của thuật toán: Chọn phần tử nhỏ nhất trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành; sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn n-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho dãy hiện hành ... đến khi dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có n phần tử, vậy tóm tắt ý tưởng thuật toán là thực hiện n-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy. Các bước tiến hành như sau :

- Bước 1: i = 1;
- Bước 2: Tìm phần tử A[vt] nhỏ nhất trong dãy hiện hành từ A[i] đến A[n].
- Bước 3 : Hoán vị hai phần tử A[i] và A[vt].
- Bước 4 : Nếu i <= n-1 thì i++; Lặp lại Bước 2.

  Ngược lại: Dừng. //n-1 phần tử đã nằm đúng vị trí.

Giả sử ta có mảng số thực A gồm n=6 phần tử có giá trị cụ thể là: A[1]=9.9 ; A[2]=8.8 ; A[3]=5.5 ; A[4]=6.6 ; A[5]=4.4 ; A[6]=7.7 ;

Quá trình sắp xếp lựa chọn mảng A qua từng bước được mô tả là:

Bước i	vt	j	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Ban đầu			9.9	8.8	5.5	6.6	4.4	7.7
Sau bước i=1	5	2,3,4,5,6	4.4 /	8.8	5.5	6.6	9.9	7.7
Sau bước i=2	3	3,4,5,6	4.4	5.5 /	8.8	6.6	9.9	7.7
Sau bước i=3	4	4,5,6	4.4	5.5	6.6 /	8.8	9.9	7.7
Sau bước i=4	6	5,6	4.4	5.5	6.6	7.7 /	9.9	8.8
Sau bước i=5	6	6	4.4	5.5	6.6	7.7	8.8 /	9.9

Cài đặt: Cài đặt thuật toán sắp xếp lựa chọn trực tiếp thành hàm sap_xep_chon như sau:

```
void   sap_xep_chon(float   A[], int   n)
```

```
{  int  vt;                  // chỉ số phần tử nhỏ nhất trong dãy hiện hành
   for  (i=1; i<=n-1; i++)
   {    vt=i;                // tạm thời xem phần tử nhỏ nhất tại vị trí thứ i
        for  (j=i+1; j<=n; j++)
             if  (A[vt]>A[j])  vt=j;
        hoan_vi(&A[i], &A[vt]);
   }
}
```

Lưu ý: Để sắp xếp theo thứ tự giảm dần thì trong chương trình trên ta chỉ cần đổi chiều bất đẳng thức trong điều kiện so sánh, thành là:

```
if  (A[vt]<A[j])  vt=j;
```

- Đánh giá giải thuật:

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

Số lần hoán vị (một hoán vị bằng 3 phép gán) lại phụ thuộc vào tình trạng ban đầu của dãy số, ta chỉ có thể ước lược trong từng trường hợp như sau :

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	n(n-1)/2	0
Xấu nhất	n(n-1)/2	3n(n-1)/2

4.2.2. Sắp thứ tự bằng phương pháp chèn:

- Giải thuật:

Cho dãy ban đầu A[1], A[2], ... , A[n] .Ta có thể xem như đoạn gồm một phần tử A[1] đã được sắp thứ tự. Sau đó thêm A[2] vào đoạn A[1] sẽ có đoạn A[1..2] được sắp thứ tự. Tiếp tục thêm A[3] vào đoạn A[1..2] để có đoạn A[1..3] được sắp thứ tự. Tiếp tục cho đến khi thêm xong A[n] vào đoạn A[1..n-1] sẽ có dãy A[1..n] được sắp thứ tự. Các bước tiến hành như sau :

- Bước 1: i = 2; // xem như đoạn A[1] đã được sắp xếp.
- Bước 2: x = A[i]; Tìm vị trí j+1 thích hợp trong đoạn A[1..i-1] để chèn A[i] vào.
- Bước 3: Dời chỗ các phần tử từ A[j+1] đến A[i-1] sang phải 1 vị trí để dành chỗ cho A[i] .
- Bước 4: A[j+1] = x; // vậy có đoạn A[1..i] đã được sắp xếp.
- Bước 5: i = i+1;

Nếu i <= n thì lặp lại Bước 2, ngược lại thì dừng.

Quá trình sắp xếp xen vào mảng A qua từng bước được mô tả là:

Bước i	j	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Ban đầu		9.9 /	8.8	5.5	6.6	4.4	7.7
Sau bước i=2	0	8.8	9.9 /	5.5	6.6	4.4	7.7
Sau bước i=3	0	5.5	8.8	9.9 /	6.6	4.4	7.7
Sau bước i=4	1	5.5	6.6	8.8	9.9 /	4.4	7.7
Sau bước i=5	0	4.4	5.5	6.6	8.8	9.9 /	7.7
Sau bước i=6	3	4.4	5.5	6.6	7.7	8.8	9.9

Cài đặt hàm con sắp xếp chèn như sau:

```
void   sap_xep_chen(float   A[], int   n)
{  for  (i=2; i<=n; i++)
   {     x=A[i]; j=i-1;
         while  ( (j>=1) && (x<A[j]) )
         {    A[j+1]=A[j];
              j--;
         }
         A[j+1]=x;              // chèn phần tử x vào sau phần tử thứ j
   }
}
```

Nhận xét:

Khi tìm vị trí thích hợp để chèn A[i] vào đoạn A[1] đến A[i-1], do đoạn đã được sắp, nên có thể sử dụng giải thuật tìm nhị phân để thực hiện việc tìm vị trí j+1, khi đó có giải thuật sắp xếp chèn nhị phân:

```
void   sap_xep_xen_vao2(float   A[], int   n)
{       int    t, p, m, i;
        float   x;//lưu giá trị A[i] tránh bị ghi đè khi dời chỗ các phần tử.
        for  (int i=1 ; i<n ; i++)
        {       x = A[i]; t = 1; p = i-1;
                while  (t<=p)   // tìm vị trí chèn x
                {       m = (t+p)/2;    // tìm vị trí thích hợp m
                        if   (x < A[m]) p = m-1;
                        else      t = m+1;
                }
                for  (int   j = i-1 ; j >=l ; j--)
                        A[j+1] = A[j];// dời các phần tử sẽ đứng sau x
                A[t] = x;                       // chèn x vào dãy
        }
}
```

- Đánh giá giải thuật:

Đối với giải thuật chèn trực tiếp, các phép so sánh xảy ra trong mỗi vòng lặp while tìm vị trí thích hợp j+1, và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử A[pos] tương ứng. Giải thuật thực hiện tất cả n-1 vòng lặp while, do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lược trong từng trường hợp.

4.2.3. Sắp thứ tự bằng phương pháp nổi bọt:

- Giải thuật:

Ý tưởng chính của giải thuật là xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đúng đầu dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có vị trí đầu dãy là i. Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét. A[j-1] <= A[j]

Các bước tiến hành như sau :

- Bước 1 : i = 1; // lần xử lý đầu tiên
- Bước 2 : j = n; // Duyệt từ cuối dãy ngược về vị trí i
  Trong khi (j >=i+1) thực hiện:
  Nếu A[j-1]>A[j] thì hoán vị A[j-1] với A[j];
  j = j-1; //xét cặp phần tử kế trước
- Bước 3 : i = i+1; // lần xử lý kế tiếp
  Nếu i<=n-1 thì lặp lại bước 2, ngược lại thì dừng

Quá trình sắp xếp nổi bọt mảng A qua từng bước được mô tả là:

Bước i	j	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Ban đầu		9.9	8.8	5.5	6.6	4.4	7.7
Sau bước i=1	6,5,4,3,2	4.4 /	9.9	8.8	5.5	6.6	7.7
Sau bước i=2	6,5,4,3	4.4	5.5 /	9.9	8.8	6.6	7.7
Sau bước i=3	6,5,4	4.4	5.5	6.6 /	9.9	8.8	7.7
Sau bước i=4	6,5	4.4	5.5	6.6	7.7 /	9.9	8.8
Sau bước i=5	6	4.4	5.5	6.6	7.7	8.8 /	9.9

Cài đặt thuật toán sắp xếp theo kiểu nổi bọt thành hàm con:

```
void   sap_xep_noi_bot(float   A[], int   n)
{   int  i, j;
    for  (i=1; i<=n-1; i++)
    {   for  (j=n; j>=i+1; j--)
            if  (A[j-1] > A[j])  hoan_vi(&A[j-1], &A[j]);
    }
}
```

- Đánh giá giải thuật:

Đối với giải thuật nổi bọt, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh.

Nhận xét: BubbleSort có các khuyết điểm sau: không nhận diện được tình trạng dãy đã có thứ tự hay có thứ tự từng phần. Các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.

4.2.4. Sắp thứ tự bằng phương pháp trộn trực tiếp :

Để sắp xếp dãy A[1], A[2], ... , A[n] , giải thuật Merge Sort dựa trên nhận xét sau:

Mỗi dãy A[1], A[2], ... , A[n] bất kỳ đều có thể coi như là một tập hợp các dãy con liên tiếp mà mỗi dãy con đều đã có thứ tự. Ví dụ dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 dãy con không giảm (12); (2, 8); (5); (1, 6); (4, 15).

Dãy đã có thứ tự coi như có 1 dãy con.

Như vậy, một cách tiếp cận để sắp xếp dãy là tìm cách làm giảm số dãy con không giảm của nó. Đây chính là hướng tiếp cận của thuật toán sắp xếp theo phương pháp trộn.

Trong phương pháp Merge sort, mấu chốt của vấn đề là cách phân hoạch dãy ban đầu thành các dãy con. Sau khi phân hoạch xong, dãy ban đầu sẽ được tách ra thành 2 dãy phụ theo nguyên tắc phân phối đều luân phiên. Trộn từng cặp dãy con của hai dãy phụ thành một dãy con của dãy ban đầu, ta sẽ nhân lại dãy ban đầu nhưng với số lượng dãy con ít nhất giảm đi một nửa. Lặp lại qui trình trên sau một số bước, ta sẽ nhận được 1 dãy chỉ gồm 1 dãy con không giảm. Nghĩa là dãy ban đầu đã được sắp xếp.

Giải thuật trộn trực tiếp là phương pháp trộn đơn giản nhất. Việc phân hoạch thành các dãy con đơn giản chỉ là tách dãy gồm n phần tử thành n dãy con. Đòi hỏi của thuật toán về tính có thứ tự của các dãy con luôn được thỏa trong cách phân hoạch này vì dãy gồm một phân tử luôn có thứ tự. Cứ mỗi lần tách rồi trộn, chiều dài của các dãy con sẽ được nhân đôi.

Các bước thực hiện thuật toán như sau:

- Bước 1 : // Chuẩn bị
  k = 1; // k là chiều dài của dãy con trong bước hiện hành
- Bước 2 :

Tách dãy $a_1, a_2, ., a_n$ thành 2 dãy b, c theo nguyên tắc luân phiên từng nhóm k phần tử:

$b = a_1, ., a_k, a_{2k+1}, ., a_{3k}, .$

$c = a_{k+1}, ., a_{2k}, a_{3k+1}, ., a_{4k}, .$

- Bước 3 :
  Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào a.
- Bước 4 :
  k = k*2;
  Nếu k < n thì trở lại bước 2.
  Ngược lại: Dừng

- Ví dụ
  Cho dãy số a:
  12 2 8 5 1 6 4 15
  k = 1:
  k = 2: k = 4:
- **Cài đặt**

```
int     b[MAX], c[MAX]; // hai mảng phụ
void  MergeSort(int  A[], int  n)
{
        int     p, pb, pc;      // các chỉ số trên các mảng a, b, c.
        int     i, k = 1;       // độ dài của dãy con khi phân hoạch.
        do      {
                // tách a thanh b và c;
                p = pb = pc = 0;
                while(p < n)    {
                        for(i = 0; (p < n)&&(i < k); i++)
                                b[pb++] = A[p++];
                        for(i = 0; (p < n)&&(i < k); i++)
                                c[pc++] = A[p++];
                }
                Merge(a, pb, pc, k); //trộn b, c lại thành a
```

```
            k *= 2;
        }while(k < n);
}
```

Trong đó hàm Merge có thể được cài đặt như sau :

```
void  Merge(int  A[], int  nb, int  nc, int  k)
{       int  p, pb, pc, ib, ic, kb, kc;
        p = pb = pc = 0; ib = ic = 0;
        while ((0 < nb)&&(0 < nc))      {
            kb = min(k, nb); kc = min(k, nc);
            if (b[pb+ib] <= c[pc+ic])       {
                A[p++] = b[pb+ib]; ib++;
                if (ib == kb) {
                    for (; ic<kc; ic++) A[p++] = c[pc+ic];
                    pb += kb; pc += kc; ib = ic = 0;
                    nb -= kb; nc -= kc;
                }
            }
            else     {
                A[p++] = c[pc+ic]; ic++;
                if(ic == kc) {
                    for(; ib<kb; ib++) A[p++] = b[pb+ib];
                    pb += kb; pc += kc; ib = ic = 0;
                    nb -= kb; nc -= kc;
                }
            }
        }
}
```

- Đánh giá giải thuật:

Ta thấy rằng số lần lặp của bước 2 và bước 3 trong thuật toán MergeSort bằng $\log_2 n$ do sau mỗi lần lặp giá trị của k tăng lên gấp đôi. Dễ thấy, chi phí thực hiện bước 2 và bước 3 tỉ lệ thuận với n. Như vậy, chi phí thực hiện của giải thuật MergeSort sẽ là $O(n\log_2 n)$. Do không sử dụng thông tin nào về đặc tính của dãy cần sắp xếp, nên trong mọi trường hợp của thuật toán chi phí là không đổi. Đây cũng chính là một trong những nhược điểm lớn của thuật toán.

4.2.5. Sắp thứ tự bằng phương pháp vun đống:

4.2.5.1. Thuật toán sắp xếp cây:

Khi tìm phần tử nhỏ nhất ở bước i, phương pháp sắp xếp chọn trực tiếp không tận dụng được các thông tin đã có được do các phép so sánh ở bước i-1. Vì lý do trên người ta tìm cách xây dựng một thuật toán sắp xếp có thể khắc phục nhược điểm này.

Mấu chốt để giải quyết vấn đề vừa nêu là phải tìm ra được một cấu trúc dữ liệu cho phép tích lũy các thông tin về sự so sánh giá trị các phần tử trong qua trình sắp xếp. Giả sử dữ liệu cần sắp xếp là dãy số : 5 2 6 4 8 1được bố trí theo quan hệ so sánh và tạo thành sơ đồ dạng cây như sau :

Trong đó một phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i+1, do đó phần tử ở mức 0 (nút gốc của cây) luôn là phần tử lớn nhất của dãy. Nếu loại bỏ phần tử gốc ra khỏi cây (nghĩa là đưa phần tử lớn nhất về đúng vị trí), thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác được bảo toàn, nghĩa là bước kế tiếp có thể sử dụng lại các kết quả so sánh ở bước hiện tại. Trong ví dụ trên ta có:

Loại bỏ 8 ra khỏi cây và thế vào các chỗ trống giá trị -? để tiện việc cập nhật lại cây:

Có thể nhận thấy toàn bộ nhánh trái của gốc 8 cũ được bảo toàn, do vậy bước kế tiếp để chọn được phần tử lớn nhất hiện hành là 6, chỉ cần làm thêm một phép so sánh 1 với 6.

Tiến hành nhiều lần việc loại bỏ phần tử gốc của cây cho đến khi tất cả các phần tử của cây đều là -?, khi đó xếp các phần tử theo thứ tự loại bỏ trên cây sẽ có dãy đã sắp xếp. Trên đây là ý tưởng của giải thuật sắp xếp cây.

4.2.5.2. Cấu trúc dữ liệu HeapSort:

Tuy nhiên, để cài đặt thuật toán này một cách hiệu quả, cần phải tổ chức một cấu trúc lưu trữ dữ liệu có khả năng thể hiện được quan hệ của các phần tử trong cây với n ô nhớ thay vì 2n-1 như trong ví dụ.

Khái niệm heap và phương pháp sắp xếp Heapsort do J.Williams đề xuất đã giải quyết được các khó khăn trên.

Định nghĩa Heap:

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử $a_l, a_2, ..., a_r$ thoả các quan hệ sau với mọi i ⎪ [l, r]:

1. $a_i >= a_{2i}$
2. $a_i >= a_{2i+1}$ {(a_i, a_{2i}), (a_i, a_{2i+1}) là các cặp phần tử liên đới}

Heap có các tính chất sau :

- Tính chất 1 : Nếu $a_l, a_2, ..., a_r$ là một heap thì khi cắt bỏ một số phần tử ở hai đầu của heap, dãy con còn lại vẫn là một heap.
- Tính chất 2 : Nếu $a_1, a_2, ..., a_n$ là một heap thì phần tử a_1 (đầu heap) luôn là phần tử lớn nhất trong heap.
- Tính chất 3 : Mọi dãy $a_l, a_2, ..., a_r$ với $2l > r$ là một heap.

Giải thuật Heapsort :

Giải thuật Heapsort trải qua 2 giai đoạn :

- Giai đoạn 1 :Hiệu chỉnh dãy số ban đầu thành heap;
- Giai đoạn 2: Sắp xếp dãy số dựa trên heap:
  - Bước 1: Đưa phần tử nhỏ nhất về vị trí đúng ở cuối dãy:
    r = n; Hoánvị (**a_1 , a_r**);
  - Bước 2: Loại bỏ phần tử nhỏ nhất ra khỏi heap: r = r-1;
    Hiệu chỉnh phần còn lại của dãy từ $a_1, a_2 ... a_r$ thành một heap.
  - Bước 3: Nếu r>1 (heap còn phần tử): Lặp lại Bước 2
    Ngược lại : Dừng

Dựa trên tính chất 3, ta có thể thực hiện giai đoạn 1 bằng cách bắt đầu từ heap mặc nhiên $a_{n/2+1}, a_{n/2+2} ... a_n$, lần lượt thêm vào các phần tử $a_{n/2}, a_{n/2-1}, ., a_1$ ta sẽ nhân được heap theo mong muốn. Như vậy, giai đoạn 1 tương đương với n/2 lần thực hiện bước 2 của giai đoạn 2.

Ví dụ: Cho dãy số a:

12 2 8 5 1 6 4 15

Giai đoạn 1: hiệu chỉnh dãy ban đầu thành heap

Giai đoạn 2: Sắp xếp dãy số dựa trên heap :

thực hiện tương tự cho r=5,4,3,2 ta được:

- Cài đặt:

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

a. Thủ tục hiệu chỉnh dãy $a_l, a_{l+1} ... a_r$ thành heap :

Giả sử có dãy $a_l, a_{l+1} ... a_r$, trong đó đoạn $a_{l+1} ... a_r$, đã là một heap. Ta cần xây dựng hàm hiệu chỉnh $a_l, a_{l+1} ... a_r$ thành heap. Để làm điều này, ta lần lượt xét quan hệ của một phần tử a_i nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1}, nếu vi phạm điều kiện quan hệ của heap, thì đổi chỗ a_i với phần tử liên đới thích hợp của nó. Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
void Shift (int  A[ ], int  l, int  r )
{       int     x,i,j;
        i = l; j =2*i; // (ai , aj ), (ai , aj+1) là các phần tử liên đới
        x = A[i];
        while ((j<=r)&&(cont))
        {
        if (j<r) // nếu có đủ 2 phần tử liên đới
                if (A[j]<A[j+1])// xác định phần tử liên đới lớn nhất                  j = j+1;
                if (A[j]<x)exit();// thoả quan hệ liên đới, dừng.
                else
```

```
            {       A[i] = A[j];
                    i = j;    // xét tiếp khả năng hiệu chỉnh lan truyền
                    j = 2*i;
                    A[i] = x;
            }
        }
    }
```

b. Hiệu chỉnh dãy a_1 , a_2 ...a_n thành heap :

Cho một dãy bất kỳ a_1 , a_2, ..., a_r , theo tính chất 3, ta có dãy $a_{n/2+1}$, $a_{n/2+2}$ ... a_n đã là một heap. Ghép thêm phần tử $a_{n/2}$ vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{n/2}$, $a_{n/2+1}$, ..., a_r thành heap, .:

```
void CreateHeap(int   A[], int   n )
{       int     l;
        l = n/2;// A[l] là phần tử ghép thêm
        while (l > 0) do
        {
                Shift(A,l,n);
                l = l -1;
        }
}
```

Khi đó hàm Heapsort có dạng sau :

```
void    HeapSort (int  A[], int  n)
{       int     r;
        CreateHeap(A, n)
        r = n-1; // r là vị trí đúng cho phần tử nhỏ nhất
        while(r > 0) do
        {       Hoanvi(A[1],A[r]);
                r = r -1;
                Shift(a,1,r);
        }
}
```

- Đánh giá giải thuật:

Việc đánh giá giải thuật Heapsort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp là $O(n\log_2 n)$

4.2.6. Sắp thứ tự bằng phương pháp nhanh (Quick_Sort):

Để sắp xếp dãy A[1], A[2], ... , A[n] giải thuật QuickSort dựa trên việc phân hoạch dãy ban đầu thành hai phần :

- Dãy con 1: Gồm các phần tử A[1] ... A[i-1] có giá trị nhỏ hơn hoặc bằng chốt.
- Dãy con 2: Gồm các phần tử A[i] ... A[n] có giá trị lớn hơn hoặc bằng chốt.

với chốt là giá trị của một phần tử tùy ý trong dãy ban đầu.

(<=chốt) (chốt) (>=chốt)

Nếu các dãy con 1 và dãy con 2 chỉ có 1 phần tử thì dãy con đó đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và con 2 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi dãy con 1 và dãy con 2 được sắp. Để sắp xếp dãy con 1 và dãy con 2, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu đã trình bày ở trên.

Giải thuật phân hoạch dãy A[t] , A[t+1], ... , A[p] thành 2 dãy con:

- Bước 1: Chọn tùy ý một phần tử A[k] trong dãy là giá trị chốt, $t \le k \le p$

chot = A[k]; i = t; j = p;

Nếu ta lấy phần tử chính giữa làm chốt thì: chot=A[(t+p)/2];

- Bước 2: Phát hiện và hiệu chỉnh cặp phần tử A[i], A[j] nằm sai chỗ:
  - . Bước 2a : Trong khi (A[i]<chot) i++;
  - . Bước 2b : Trong khi (A[j]>chot) j--;
  - . Bước 2c : Nếu i<= j thì Hoán vị (&A[i], &A[j]);
- Bước 3 : i → i → . . . ← j ← j t j i p

Nếu i < j: Lặp lại Bước 2// vì chưa xét hết mảng

Nếu i >= j: Dừng

NHẬN XÉT:

- Về nguyên tắc, có thể chọn giá trị chốt là một phần tử tùy ý trong dãy, nhưng để đơn giản, dễ diễn đạt giải thuật, phần tử có vị trí giữa thường được chọn, khi đó k = (t +p)/ 2

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

- Cài đặt:

Thuật toán QuickSort có thể được cài đặt đệ qui như sau :

```
void   quicksort(float  A[], int  t, int  p)
{     if  (t<p)
        {     int  i=t, j=p;
              float          chot;
              chot=A[(t+p)/2];  // chon phan tu o giua lam chot
                 while  (i < j)
                 {     while  (A[i] < chot)   i++;
                       while  (A[j] > chot)   j--;
                       if  (i <= j)
                           {  hoan_vi(&A[i], &A[j]);
                                 i++ ; j--;
                           }
              }
                 quicksort(a,t,j);
                 quicksort(a,i,p);
         }
}
```

- Đánh giá giải thuật:

Hiệu qủa thực hiện của giải thuật QuickSort phụ thuộc vào việc chọn giá trị mốc. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đều chọn được phần tử median (phần tử lớn hơn (hay bằng) nửa số phần tử, và nhỏ hơn (hay bằng) nửa số phần tử còn lại) làm mốc, khi đó dãy được phân chia thành 2 phần bằng nhau và cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhằm phần tử có giá trị cực đại (hay cực tiểu) là mốc, dãy sẽ bị phân chia thành 2 phần không đều: một phần chỉ có 1 phần tử, phần còn lại gồm (n-1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong. Ta có bảng tổng kết:

Trường hợp	Độ phức tạp
Tốt nhất	n*log(n)
Trung bình	n*log(n)
Xấu nhất	n^2

4.3. Sắp thứ tự ngoại:

Sắp thứ tự ngoại là sắp thứ tự trên tập tin. Khác với sắp xếp dãy trên bộ nhớ có số lượng phần tử nhỏ và truy xuất nhanh, tập tin có thể có số lượng phần tử rất lớn và thời gian truy xuất chậm. Do vậy việc sắp xếp trên các cấu trúc dữ liệu loại tập tin đòi hỏi phải áp dụng các phương pháp đặc biệt.

Chương này sẽ giới thiệu một số phương pháp như sau:

- Phương pháp trộn RUN
- Phương pháp trộn tự nhiên

4.3.1. Phương pháp trộn RUN:

- *Khái niệm cơ bản:*

Run là một dãy liên tiếp các phần tử được sắp thứ tự.

Ví dụ: *1 2 3 4 5 là một run gồm có 5 phần tử*

Chiều dài run chính là số phần tử trong Run. Chẳng hạn, run trong ví dụ trên có chiều dài là 5.

Như vậy, mỗi phần tử của dãy có thể xem như là 1 run có chiều dài là1. Hay nói khác đi, mỗi phần tử của dãy chính là một run có chiều dài bằng 1.

Việc tạo ra một run mới từ 2 run ban đầu gọi là trộn run (merge). Hiển nhiên, run được tạo từ hai run ban đầu là một dãy các phần tử đã được sắp thứ tự.

- *Giải thuật:*

Giải thuật sắp xếp tập tin bằng phương pháp trộn run có thể tóm lược như sau:

Input: f0 là tập tin cần sắp thứ tự.

Output: f0 là tập tin đã được sắp thứ tự.

Gọi f1, f2 là 2 tập tin trộn.

Các tập tin f0, f1, f2 có thể là các tập tin tuần tự (text file) hay có thể là các tập tin truy xuất ngẫu nhiên (File of <kiểu>)

Bước 1:

- Giả sử các phần tử trên f0 là:

  24 12 67 33 58 42 11 34 29 31

- f1 ban đầu rỗng, và f2 ban đầu cũng rỗng.
- Thực hiện phân bố m=1 phần tử lần lượt từ f0 vào f1 và f2:

  f1: 24 67 58 11 29
  f0: 24 12 67 33 58 42 11 34 29 31
  f2: 12 33 42 34 31

- Trộn f1, f2 thành f0:

  f0: 12 24 33 67 42 58 11 34 29 31

Bước 2:

-Phân bố m=2 phần tử lần lượt từ f0 vào f1 và f2:

  f1: 12 24 42 58 29 31
  f0: 12 24 33 67 42 58 11 34 29 31
  f2: 33 67 11 34

- Trộn f1, f2 thành f0:

  f1: 12 24 42 58 29 31
  f0: 12 24 33 67 11 34 42 58 29 31
  f2: 33 67 11 34

Bước 3:

- Tương tự bước 2, phân bố m=4 phần tử lần lượt từ f0 vào f1 và f2, kết quả thu được như sau:

  f1: 12 24 33 67 29 31
  f2: 11 34 42 58

- Trộn f1, f2 thành f0:

  f0: 11 12 24 33 34 42 58 67 29 31

Bước 4:

- Phân bố m=8 phần tử lần lượt từ f0 vào f1 và f2:

  f1: 11 12 24 33 34 42 58 67
  f2: 29 31

- Trộn f1, f2 thành f0:

  f0: 11 12 24 29 31 33 34 42 58 67

Bước 5:

Lặp lại tương tự các bước trên, cho đến khi chiều dài m của run cần phân bổ lớn hơn chiều dài n của f0 thì dừng. Lúc này f0 đã được sắp thứ tự xong.

Cài đặt:

```
/*
Sap xep file bang phuong phap tron truc tiep
Cai dat bang Borland C 3.1 for DOS.
*/
#include <conio.h>
#include <stdio.h>
void   tao_file();
void   xuat_file();
void   chia(FILE   *a,FILE   *b,FILE   *c, int p);
void   tron(FILE   *b,FILE   *c,FILE   *a, int p);
```

```
int   p,n;
/**/
main ()
{       FILE   *a,*b,*c;
        tao_file();
        xuat_file();
        p = 1;
        while (p < n)
        {       chia(a,b,c,p);
                tron(b,c,a,p);
                p=2*p;
        }
        printf("\n");
        xuat_file();
        getch();
}
void   tao_file()
/*
Tao file co n phan tu
*/
{       int i,x;
        FILE *fp;
        fp=fopen("d:\\ctdl\\sorfile\bang.int","wb");
        printf("Cho biet so phan tu : ");
        scanf("%d",&n);
        for (i=0;i<n;i++)
        {       scanf("%d",&x);
                fprintf(fp,"%3d",x);
        }
        fclose(fp);
}
void   xuat_file()
/*
Hien thi noi dung cua file len man hinh
*/
{       int x;
        FILE *fp;
        fp=fopen("d:\\ctdl\\sortfile\bang.int","rb");
        i=0;
        while (i<n)
        {       fscanf(fp,"%d",&x);
                printf("%3d",x);
                i++;
        }
        fclose(fp);
}
void   chia(FILE *a,FILE *b,FILE *c,int p)
/*
Chia xoay vong file a cho file b va file c moi lan p phan tu cho den khi het file a.
*/
{       int dem,x;
        a=fopen("d:\ctdl\sortfile\bang.int","rb");
        b=fopen("d:\ctdl\sortfile\bang1.int","wb");
        c=fopen("d:\ctdl\sortfile\bang2","wb");
```

```
        while (!feof(a))
        {       /*Chia p phan tu cho b*/
                dem=0;
                while ((dem<p) && (!feof(a)))
                {       fscanf(a,"%3d",&x);
                        fprintf(b,"%3d",x);
                        dem++;
                }
                /*Chia p phan tu cho c*/
                dem=0;
                while ((dem<p) && (!feof(a)))
                {       fscanf(a,"%3d",&x);
                        fprintf(c,"%3d",x);
                        dem++;
                }
        }
        fclose(a); fclose(b); fclose(c);
}
void   tron(FILE   *b, FILE   *c, FILE   *a, int p)
/*
Tron p phan tu tren b voi p phan tu tren c thanh 2*p phan tu tren a
cho den khi file b hoac c het.
*/
{       int stop,x,y,l,r;
        a=fopen("d:\ctdl\sortfile\bang.int","wb");
        b=fopen("d:\ctdl\sortfile\bang1.int","rb");
        c=fopen("d:\ctdl\sortfile\bang2.int","rb");
        while ((!feof(b)) && (!feof(c)))
        {       l=0;/*so phan tu cua b da ghi len a*/
                r=0;/*so phan tu cua c da ghi len a*/
                fscanf(b,"%3d",&x);
                fscanf(c,"%3d",&y);
                stop=0;
                while ((l!=p) && (r!=p) && (!stop))
                {       if (x<y)
                        {       fprintf(a,"%3d",x);
                                l++;
                                if ((l<p) && (!feof(b)))
                                /*chua du p phan tu va chua het file b*/
                                        fscanf(b,"%3d",&x);
                                else
                                {       fprintf(a,"%3d",y);
                                        r++;
                                        if (feof(b))  stop=1;
                                }
                        }
                        else
                        {       fprintf(a,"%3d",y);
                                r++;
                                if ((r<p) && (!feof(c)))
                                /*chua du p phan tu va chua het file c*/
                                        fscanf(c,"%3d",&y);
                                else
                                {       fprintf(a,"%3d",x);
```

```
                                        l++;
                                        if (feof(c))
                                                stop=1;
                                }
                        }
                }
        }
        /*
        Chep phan con lai cua p phan tu tren b len a
        */
        while ((!feof(b)) && (l<p))
        {       fscanf(b,"%3d",&x);
                fprintf(a,"%3d",x);
                l++;
        }
        /*
        Chep phan con lai cua p phan tu tren c len a
        */
        while ((!feof(c)) && (r<p))
        {       fscanf(c,"%3d",&y);
                fprintf(a,"%3d",y);
                r++;
        }
        if (!feof(b))
        {       /*chep phan con lai cua b len a*/
                while (!feof(b))
                {       fscanf(b,"%3d",&x);
                        fprintf(a,"%3d",x);
                }
        }
        if (!feof(c))
        {       /*chep phan con lai cua c len a*/
                while (!feof(c))
                {       fscanf(c,"%3d",&x);
                        fprintf(a,"%3d",x);
                }
        }
        fclose(a); fclose(b); fclose(c);
}
```

4.3.2. Các phương pháp trộn tự nhiên:

- *Giải thuật:*

Trong phương pháp trộn đã trình bày ở trên, giải thuật không tận dụng được chiều dài cực đại của các run trước khi phân bố; do vậy, việc tối ưu thuật toán chưa được tận dụng.

Đặc điểm cơ bản của phương pháp trộn tự nhiên là tận dụng độ dài "tự nhiên" của các run ban đầu; nghĩa là, thực hiện việc trộn các run có độ dài cực đại vơi nhau cho đến khi dãy chỉ bao gồm một run: dãy đã được sắp thứ tự.

Input: f0 là tập tin cần sắp thứ tự.

Output: f0 là tập tin đã được sắp thứ tự.

Lặp Cho đến khi dãy cần sắp chỉ gồm duy nhất một run.

Phân bố:

- Chép một dây con có thứ tự vào tắp tin phụ fi (i>=1). Khi chấm dứt dây con này, biến eor (end of run) có giá trị True.
- Chép dây con có thứ tự kế tiếp vào tập tin phụ kế tiếp fi+1 (xoay vòng).
- Việc phân bố kết thúc khi kết thúc tập tin cần sắp f0.

Trộn:

- Trộn 1 run trong f1 và1 run trong f2 vào f0.
- Việc trộn kết thúc khi duyệt hết f1 và hết f2 (hay nói cách khác, việc trộn kết thúc khi đã có đủ n phần tử cần chép vào f0).

Cài đặt:

```
/*
Sap xep file bang phuong phap tron tu nhien
*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
void  CreatFile(FILE *Ft,int  N);
void  ListFile(FILE *);
void  Distribute();
void  Copy(FILE *,FILE *);
void  CopyRun(FILE *,FILE *);
void  MergeRun();
void Merge();
//
typedef int DataType;
FILE  *F0,*F1,*F2;
Int  M,N,Eor;
/*
Bien eor dung de kiem tra ket thuc Run hoac File
*/
DataType  X1,X2,X,Y;
//Ham main
main(void)
{       clrscr();
        randomize();
        cout<<" Nhap so phan tu: ";
        cin>>N;
        CreatFile(F0,N);
        ListFile(F0);
        do
        {       F0=fopen("d:\\ctdl\\sortfile\\bang.int","rb");
                F1=fopen("d:\\ctdl\\sortfile\\bang1.int","wb");
                F2=fopen("d:\\ctdl\\sortfile\\bang2.int","wb");
                Distribute();
                F0=fopen("d:\\ctdl\\sortfile\\bang.int","wb");
                F1=fopen("d:\\ctdl\\sortfile\\bang1.int","rb");
                F2=fopen("d:\\ctdl\\sortfile\\bang2.int","rb");
                M=0;
                Merge();
        }while (M != 1);
        ListFile(F0);
        getch();
}
void  CreatFile(FILE  *Ft,int  Num)
/*Tao file co ngau nhien n phan tu* */
{       randomize();
        Ft=fopen("d:\\ctdl\\sortfile\\bang.int","wb");
        for( int i = 0 ; i < Num ; i++)
```

```
        {       X = random(30);
                fprintf(Ft,"%3d",X);
        }
        fclose(Ft);
}
void  ListFile(FILE  *Ft)
/*Hien thi noi dung cua file len man hinh */
{       DataType X,I=0;
        Ft = fopen("d:\\ctdl\\sortfile\\bang.int","rb");
        while ( I < N )
        {       fscanf(Ft,"%3d",&X);
                cout<<" "<<X;
                I++;
        }
        printf("\n\n");
        fclose(Ft);
}
/**/
void  Copy(FILE *Fi,FILE *Fj)
{       //Doc phan tu X tu Tap tin Fi, ghi X vao Fj
        //Eor==1, Neu het Run(tren Fi) hoac het File Fi
        fscanf(Fi,"%3d",&X);
        fprintf(Fj,"%3d",X);
        if  (!feof(Fi) )
        {       fscanf(Fi,"%3d",&Y);
                long curpos = ftell(Fi)-2;
                fseek(Fi, curpos, SEEK_SET);
        }
        if ( feof(Fi) ) Eor = 1;
        else Eor = (X > Y) ? 1 : 0 ;
}

void  Distribute()
/*Phan bo luan phien cac Run tu nhien tu F0 vao F1 va F2*/
{       do
        {       CopyRun(F0,F1);
                if( !feof(F0) ) CopyRun(F0,F2);
        }while( !feof(F0) );
        fclose(F0);
        fclose(F1);
        fclose(F2);
}
void  CopyRun(FILE *Fi,FILE *Fj)
/*Chep 1 Run tu Fi vao Fj */
{       do
                Copy(Fi,Fj);
        while ( !Eor);
}
void  MergeRun()
/*Tron 1 Run cua F1 va F2 vao F0*/
{       do
        {       fscanf(F1,"%3d",&X1);
                long curpos = ftell(F1)-2;
                fseek(F1, curpos, SEEK_SET);
```

```
                fscanf(F2,"%3d",&X2);
                curpos = ftell(F2)-2;
                fseek(F2, curpos, SEEK_SET);
                if  ( X1 <= X2 )
                {       Copy(F1,F0);
                        if  (Eor)  CopyRun(F2,F0);
                }
                else
                {       Copy(F2,F0);
                        if  (Eor)  CopyRun(F1,F0);
                }
        } while ( !Eor );
}
void Merge()
/*Tron cac run tu F1 va F2 vao F0*/
{       while( (!feof(F1)) && (!feof(F2)) )
        {       MergeRun();
                M++;
        }
        while( !feof(F1) )
        {       CopyRun(F1,F0);
                M++;
        }
        while( !feof(F2) )
        {       CopyRun(F2,F0);
                M++;
        }
        fclose(F0);
        fclose(F1);
        fclose(F2);
}
```

--- HẾT CHƯƠNG 4 ---

Chương 5: TÌM KIẾM DỮ LIỆU

5.1. Nhu cầu tìm kiếm dữ liệu:

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

Ví dụ: tra cứu từ điển, tìm sách trong thư viện...

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

Ví dụ: Các từ trong từ điển được sắp xếp theo từng vần, trong mỗi vần lại được sắp xếp theo trình tự alphabet; sách trong thư viện được xếp theo chủ đề ...

Vì thế, khi xây dựng một hệ quản lý thông tin trên máy tính, bên cạnh các thuật toán tìm kiếm, các thuật toán sắp xếp dữ liệu cũng là một trong những chủ đề được quan tâm hàng đầu.

Hiện nay đã có nhiều giải thuật tìm kiếm và sắp xếp được xây dựng, mức độ hiệu quả của từng giải thuật còn phụ thuộc vào tính chất của cấu trúc dữ liệu cụ thể mà nó tác động đến. Dữ liệu được lưu trữ chủ yếu trong bộ nhớ chính và trên bộ nhớ phụ, do đặc điểm khác nhau của thiết bị lưu trữ, các thuật toán tìm kiếm và sắp xếp được xây dựng cho các cấu trúc lưu trữ trên bộ nhớ chính hoặc phụ cũng có những đặc thù khác nhau. Chương này sẽ trình bày các thuật toán tìm kiếm dữ liệu được lưu trữ trên bộ nhớ chính.

- Tập dữ liệu được lưu trữ là dãy số thực A[1], A[2], ... , A[n].

Giả sử chọn cấu trúc dữ liệu mảng để lưu trữ dãy số này trong bộ nhớ chính, có khai báo :

```
float   A[100];
```

Lưu ý các bản cài đặt trong giáo trình sử dụng ngôn ngữ C, do đó chỉ số của mảng mặc định bắt đầu từ 0, nên các giá trị của các chỉ số có chênh lệch so với thuật toán, nhưng ý nghĩa không đổi.

- Khoá cần tìm là x, được khai báo như sau:

```
float   x;
```

5.2. Các thuật toán tìm kiếm:

Cho trước mảng A gồm n phần tử số thực A[1], A[2], ... , A[n] , và cho trước số thực x. Cần tìm phần tử trong mảng A mà có giá trị x. Các hàm tìm kiếm kiểu int dưới đây tìm và trả về vị trí của phần tử đầu tiên trong mảng A mà có giá trị x (nếu có), hoặc trả về giá trị -1 (nếu tìm không có).

5.2.1. Tìm kiếm tuần tự:

Giải thuật:

Tìm tuyến tính là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Thuật toán tiến hành so sánh x lần lượt với phần tử thứ nhất, thứ hai, ... của mảng A cho đến khi gặp được phần tử có khóa cần tìm, hoặc đã tìm hết mảng mà không thấy x. Các bước tiến hành như sau :

- Bước 1:
  i = 1; // bắt đầu từ phần tử đầu tiên của dãy.
- Bước 2: So sánh A[i] với x, có 2 khả năng:
  A[i] = x : Tìm thấy, trả về giá trị i, rồi dừng.
  A[i] != x : Sang Bước 3.
- Bước 3 :
  i = i+1; // xét tiếp phần tử kế trong mảng.
  Nếu i<=n thì lặp lại bước 2; ngược lại hết mảng, tìm không thấy, thì trả về giá trị -1, rồi dừng.

Cài đặt: Từ mô tả trên đây của thuật toán tìm tuyến tính, có thể cài đặt hàm timkiem_tuantu(A, n, x) như sau:

```
int  timkiem_tuantu(float  A[], int  n, float  x)
{  int  i=1;
   while  ( (i<=n) && (A[i]!=x) )  i++;
   if  (i<=n)  return  i;              // A[i] là phần tử có khóa x
   else  return  -1;                   //  khong tim thay x
}
```

Trong cài đặt trên đây, nhận thấy mỗi lần lặp của vòng lặp while phải tiến thành kiểm tra 2 điều kiện (i<=n) - điều kiện biên của mảng - và (A[i]!=x) - điều kiện kiểm tra chính. Nhưng thật sự chỉ cần kiểm tra điều kiện chính (A[i] !=x), để cải tiến cài đặt, có thể dùng phương pháp *"lính canh"* - đặt thêm một phần tử có giá

trị x vào cuối mảng, như vậy bảo đảm luôn tìm thấy x trong mảng, sau đó dựa vào vị trí tìm thấy để kết luận. Cài đặt cải tiến sau đây của hàm timtuantu giúp giảm bớt một phép so sánh trong vòng lặp :

```
int  tim_tuantu(float  A[ ] , int  n , float  x)
{       int  i=1;                       // mảng gồm n phần tử A[1] , A[2] , … , A[n]
        A[n+1] = x;             // thêm phần tử thứ n+1
        while (A[i]!=x )        i++;
        if (i==n+1)     return -1;      // tìm hết mảng nhưng không có x
        else   return  i; // tìm thấy x tại vị trí i
}
```

Đánh giá giải thuật :

Có thể ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Trường hợp giải thuật tìm tuyến tính, có:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị x
Xấu nhất	n+1	Phần tử cuối cùng có giá trị x
Trung bình	(n+1)/2	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

Vậy giải thuật tìm tuyến tính có độ phức tạp tính toán cấp n:

$T(n) = O(n)$

Nhận xét:

- Giải thuật tìm tuyến tính không phụ thuộc vào thứ tự của các phần tử mảng, do vậy đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy số bất kỳ.

- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

5.2.2. Tìm kiếm nhị phân:

Giải thuật:

Đối với những dãy số đã có thứ tự (giả sử thứ tự tăng dần), các phần tử trong dãy có quan hệ $A[i-1] \le A[i] \le A[i+1]$, từ đó kết luận được nếu x<A[i] thì x chỉ có thể xuất hiện trong đoạn trước A[1] ... A[i-1] của dãy, ngược lại nếu x>A[i] thì x chỉ có thể xuất hiện trong đoạn sau A[i+1] … A[n] của dãy. Giải thuật tìm nhị phân áp dụng nhận xét trên đây để tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong dãy. Ý tưởng của giải thuật là tại mỗi bước tiến hành so sánh x với phần tử nằm ở vị trí giữa của dãy tìm kiếm hiện hành, dựa vào kết quả so sánh này để quyết định giới hạn dãy tìm kiếm ở bước kế tiếp là đoạn trước hay đoạn sau của dãy tìm kiếm hiện hành. Giả sử dãy tìm kiếm hiện hành bao gồm các phần tử A[t] … A[p] , các bước tiến hành như sau :

- Bước 1: t=1; p=n; // tìm kiếm trên tất cả các phần tử
- Bước 2: A[t] A[g-1] A[g] A[g+1] A[p]
  - g = (t+p)/2; // lấy mốc ở giữa để so sánh
  - So sánh x với A[g] , có 3 khả năng :
    - x<A[g] : // tìm tiếp x trong đoạn trước A[t] … A[g-1] là:
      p=g-1;
    - x>A[g] : // tìm tiếp x trong đoạn sau A[g+1] … A[p] là:
      t=g+1;
    - x=A[g] : Tìm thấy, trả về giá trị g, rồi dừng
- Bước 3:
  - Nếu t ≤ p // còn phần tử chưa xét, nên tìm tiếp là:
    Lặp lại Bước 2.
  - Ngược lại // đã xét hết tất cả các phần tử, tìm không thấy:
    trả về giá trị -1, rồi dừng

Cài đặt:

Thuật toán tìm nhị phân có thể được cài đặt thành hàm timkiem_nhiphan(a, n, x) như sau:

```
int  timkiem_nhiphan(float  A[], int  n, float  x)
{  int  t, p, g;
   t=1;                 // vi tri dau ben trai
```

```
    p=n;                    // vi tri cuoi ben phai
    while  (t<=p)
    {  g=(t+p)/2;           // lay vi tri so giua
       if  (x<A[g])  p=g-1;
       else  if  (x>A[g])  t=g+1;
             else  return  g;
    }
    return  -1;             // nếu không tìm thấy.
}
```

Đánh giá giải thuật :

Trường hợp giải thuật tìm nhị phân, có bảng phân tích sau:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng có giá trị x
Xấu nhất	$\log_2 n$	Không có x trong mảng
Trung bình	$(\log_2 n)/2$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

Vậy giải thuật tìm nhị phân có độ phức tạp tính toán cấp n: $T(n) = O(\log_2 n)$

Nhận xét:

- Giải thuật tìm nhị phân dựa vào quan hệ giá trị của các phần tử mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.
- Giải thuật tìm nhị phân tiết kiệm thời gian hơn rất nhiều so với giải thuật tìm tuyến tính do

$$T_{\text{nhị phân}}(n) = O(\log_2 n) < T_{\text{tuyếntính}}(n) = O(n)$$

Tuy nhiên khi muốn áp dụng giải thuật tìm nhị phân cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự. Thời gian này không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại . Tất cả các nhu cầu đó tạo ra khuyết điểm chính cho giải thuật tìm nhị phân. Ta cần cân nhắc nhu cầu thực tế để chọn một trong hai giải thuật tìm kiếm trên sao cho có lợi nhất.

--- HẾT CHƯƠNG 5 ---

Chương 6: CẤU TRÚC DỮ LIỆU BẢNG BĂM MỞ

Tổ chức cấu trúc kiểu này gồm từ tập (có thể vô hạn các phần tử) được phân ra B lớp đánh số từ 0 đến B-1 theo một hàm băm h (hash function) nào đó sao cho x trong tập là đối số của h và h(x) xác định một số nguyên từ 0, 1, 2, ..., B-1. Lớp thứ i gồm các phần tử x mà h(x)=i , tức là phần tử x sẽ được phân vào lớp h(x).

Nếu có N phần tử thì trung bình mỗi lớp (thùng) chứa N/B phần tử. Nếu chọn B tương đối lớn thì số phần tử trung bình ở mỗi lớp sẽ càng ít. Khi đó mỗi phép toán trên cấu trúc này trung bình mất một thời gian cố định không lớn lắm, độc lập với n (tức tương đương B).

Vấn đề quan trọng là chọn hàm băm h sao cho số phần tử trong các thùng (lớp) không chênh lệch nhiều.

Ví dụ ta chỉ xét các số nguyên tố. Ta xây dựng bảng băm mở A gồm 6 lớp. Trong đó:

Lớp đầu tiên A[0] chứa các số nguyên tố tận cùng bằng 1.

Lớp A[1] chứa các số nguyên tố tận cùng bằng 2 (chỉ chứa số 2).

Lớp A[2] chứa các số nguyên tố tận cùng bằng 3.

Lớp A[3] chứa các số nguyên tố tận cùng bằng 5 (chỉ chứa số 5).

Lớp A[4] chứa các số nguyên tố tận cùng bằng 7.

Lớp A[5] chứa các số nguyên tố tận cùng bằng 9.

Ta xây dựng bảng băm mở như sau:

```
const   int   B=6;     // co B=6 lop, tu lop 0 den lop B-1=5
struct   element
{        int    key;
         element  *next;
};
typedef   element  *List;
List   p;
List   A[B];
int   b;
```

Ta xây dựng hàm băm mở h với đầu vào số x, thì ta tính lấy chữ số cuối cùng y của số x để xác định lớp sẽ chứa phần tử x.

```
int  H(int  x)
{        int  y, kq;
         y=x % 10;
         switch (y)
         {        case  1: kq=0; break;
                  case  2: kq=1; break;
                  case  3: kq=2; break;
                  case  5: kq=3; break;
                  case  7: kq=4; break;
                  case  9: kq=5; break;
         }
         return  kq;
}
```

Kiểm tra số x có phải là số nguyên tố không. Hàm trả về 1 nếu x đúng là số nguyên tố:

```
int  ktrasonguyento(int  x)
{        int  j;
         j=2;
         while  ( (j<=sqrt(x)) && (x % j !=0) ) j++;
         if  ( j>sqrt(x) )  return  1;
         else  return  0;
}
```

Khởi động bảng băm A:

```
void  MAKENULL()
{        for  (i=0; i<=B-1; i++)
         A[i]=NULL;
}
```

Duyệt toàn bộ bảng băm A:

```
void  Display()
{       for  (i=0; i<=B-1; i++)
        {       printf("\n Lop A[%d]:" , i);
                p=A[i];
                while  (p!=NULL)
                {       printf("%5d" , (*p).key);
                        p=(*p).next;
                }
        }
}
```

Hàm kiểm tra xem lớp i có rỗng không. Hàm trả về 1 nếu đúng, trả về 0 nếu sai:

```
int  emptyHashTable(int   i)
{       return   (A[i]==NULL ? 1:0) ;
}
```

Hàm kiểm tra xem toàn bộ bảng băm có rỗng không. Hàm trả về 1 nếu tất cả các lớp đều rỗng, 0 nếu ngược lại:

```
int  empty()
{       for  (i=0; i<=B-1; i++)
                if   (A[i] != NULL)  return  0;
        return  1;
}
```

Tìm kiếm xem có phần tử x trong bảng băm không, nếu có thì trả về số thứ tự của lớp chứa phần tử x, nếu không có thì trả về -1.

```
int  MEMBER(int  x)
{       i=H(x);
        p=A[i];
        while  ( (p!=NULL) && (x!=(*p).key) )
                p=(*p).next;
        if  ( p!=NULL)  return  i;
        else  return  -1;
}
```

Thêm phần tử x vào bảng băm. Thêm vào đầu lớp:

```
void  insert(int  x)
{       if  (MEMBER(x)==-1)
        {       p = new element;
                (*p).key=x;
                i=H(x);
                (*p).next=A[i];
                A[i]=p;
        }
}
```

Xóa phần tử x ra khỏi bảng băm:

```
void  DELETE(int  x)
{       List   before;
        i=MEMBER(x);
        if  (i!=-1)
                {       p=A[i];
                while  ( (p!=NULL) && ((*p).key!=x) )
                {       before=p;
                        p=(*p).next;
                }
                if  (p==A[i])   A[i]=(*p).next;
                else   (*before).next=(*p).next;
```

```
                delete p;
                }
        }
```

Xóa tất cả các phần tử trong lớp i:

```
        void  clearHashTable(int  i)
        {       while  (A[i] != NULL)
                {       p=A[i];
                        A[i] = (*p).next;
                        delete  p;
                }
        }
```

Xóa tất cả các phần tử trong bảng băm A:

```
        void  clear()
        {       for  (i=0; i<=B-1; i++)
                        clearHashTable(i);
        }
```

--- HẾT CHƯƠNG 6 ---

THI CUỐI KỲ:

1. Thi viết.
2. Đề đóng.
3. Nội dung:
    a. Đệ qui.
    b. Chương 2: Danh sách liên kết, Ngăn xếp, Hàng đợi.
    c. Chương 3: Cây. Cây nhị phân. Cây nhị phân tìm kiếm.
    d. Chương 4: Sắp xếp thứ tự dữ liệu.
    e. Chương 5: Tìm kiếm tuần tự, tìm kiếm nhị phân.
4. Khi thi: Sinh viên có mặt tại phòng thi lúc . . . ngày . . . , chuẩn bị thẻ sinh viên.
5. Ban đầu tất cả tắt Webcam. Khi thầy đọc tên điểm danh em nào thì em đó: Bật Webcam + Nói: Có + Đưa thẻ SV trước webcam. Trong quá trình làm bài thì luôn bật webcam, bật micro và giữ im lặng.
6. Xem đề thi bằng cách: Nháy de_thi.docx
7. Thời gian làm bài: 60 phút.
8. Làm bài:
    a. Viết trên giấy, ở đầu trang đầu tiên ghi họ tên, lớp sinh hoạt, nhóm, thi cuối kỳ, môn …
    b. Cuối giờ: Chụp ảnh, tạo thành file .PDF
    c. Tên file bài làm là STT_Họ Và Tên.PDF , ví dụ: 01_Lê Văn Li.PDF
9. Nộp bài: Tại My work, nháy Attach, upload, chọn file bài làm .PDF để tải lên hệ thống, chờ load file thành công, nháy Turn in.
10. Rồi chờ để thầy kiểm tra lại là file bài làm đã tải lên hệ thống được chưa.
11. Nếu đã tải lên hệ thống rồi thì kết thúc thi, kết thúc buổi, thoát, nghỉ.