

Chương 1: CÁC KHÁI NIỆM CƠ BẢN

- 1.1. Thuật toán và cấu trúc dữ liệu
- 1.2. Các kiểu dữ liệu cơ bản trong ngôn ngữ C
 - 1.2.1. Kiểu dữ liệu đơn giản
 - 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
 - 1.2.2.1. Kiểu mảng
 - 1.2.2.2. Kiểu chuỗi ký tự
 - 1.2.2.3. Kiểu bản ghi
- 1.3. Kiểu con trỏ
 - 1.3.1. 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
 - 1.4.1. Định nghĩa
 - 1.4.2. Khai báo kiểu tham chiếu
 - 1.4.3. Ứng dụng kiểu tham chiếu
- 1.5. Đề qui
 - 1.5.1. Định nghĩa
 - 1.5.2. Các nguyên lý khi dùng kỹ thuật đệ qui

Chương 2: DANH SÁCH

- 2.1. Khái niệm
- 2.2. Danh sách đặc
 - 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
 - 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
 - 2.4.1. 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
 - 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
- 2.7. Danh sách hạn chế
 - 2.7.1. Khái niệm
 - 2.7.2. Ngăn xếp
 - 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
 - 2.7.3. Hàng đợi
 - 2.7.3.1. Định nghĩa

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

Chương 3: CÂY

- 3.1. Một số khái niệm
 - 3.1.1. Các định nghĩa
 - 3.1.2. Các cách biểu diễn cây
- 3.2. Cây nhị phân
 - 3.2.1. Định nghĩa và tính chất
 - 3.2.1.1. Định nghĩa
 - 3.2.1.2. Các dạng đặc biệt của cây nhị phân
 - 3.2.1.3. Các tính chất của cây nhị phân
 - 3.2.2. Biểu diễn cây nhị phân
 - 3.2.2.1. Biểu diễn cây nhị phân bằng danh sách đặc
 - 3.2.2.2. Biểu diễn cây nhị phân bằng danh sách liên kết
 - 3.2.3. Các phép toán trên cây nhị phân được biểu diễn bằng danh sách liên kết
- 3.3. Cây nhị phân tìm kiếm
 - 3.3.1. Định nghĩa
 - 3.3.2. Các phép toán trên cây nhị phân tìm kiếm
 - 3.3.3. Đánh giá
- 3.4. Cây nhị phân cân bằng
 - 3.4.1. Cây cân bằng hoàn toàn
 - 3.4.1.1. Định nghĩa
 - 3.4.1.2. Đánh giá
 - 3.4.2. Cây cân bằng
 - 3.4.2.1. Định nghĩa
 - 3.4.2.2. Lịch sử cây cân bằng (AVL)
 - 3.4.2.3. Chiều cao của cây AVL
 - 3.4.2.4. Cấu trúc dữ liệu cho cây AVL
 - 3.4.2.5. Đánh giá cây AVL
- 3.5. Cây tổng quát
 - 3.5.1. Định nghĩa
 - 3.5.2. Biểu diễn cây tổng quát bằng danh sách liên kết
 - 3.5.3. Các phép duyệt cây tổng quát
 - 3.5.4. Cây nhị phân tương đương

---o-O-o---

Tài liệu tham khảo:

- [1] Đỗ Xuân Lôi, Cấu trúc dữ liệu và giải thuật, NXB Khoa học và kỹ 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 kỹ thuật, 2000
- [7] Yedidiah 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 bất kỳ 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 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. 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 sẽ thay đổi theo

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

1.2.1. Kiểu dữ liệu đơn giản: Kiểu dữ liệu đơn giản có giá trị đơn duy nhất, gồm các kiểu:

1.2.1.1. Kiểu ký tự:

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

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

1.2.1.2. Kiểu nguyên:

Kiểu nguyên có giá trị là một số nguyên, ví dụ số 1991, gồm các kiểu nguyên sau:

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

Lưu ý: Các kiểu ký tự cũng có thể xem là kiểu nguyên 1 Byte

1.2.1.3. Kiểu thực:

Kiểu thực có giá trị là một số thực, ví dụ số 1.65, gồm các kiểu thực sau:

Kiểu	Miền giá trị	Kích thước
float	từ 3.4E-38 đến 3.4E+38	4 Byte
double	từ 1.7E-308 đến 1.7E+308	8 Byte
long double	từ 3.4E-4932 đến 1.1E4932	10 Byte

1.2.2. Kiểu dữ liệu có cấu trúc:

Kiểu dữ liệu có cấu trúc có giá trị gồm nhiều thành phần, gồm các kiểu sau:

1.2.2.1. Kiểu mảng:

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 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

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 và gán giá trị cho biến chuỗi ký tự như sau:

```
char ht[15] = "Le Li" ;
```

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 ta viết tên biến bản ghi, tiếp theo là dấu chấm, rồi đến tên trường

Ví dụ:

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

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

Ví dụ

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main()
{ struct SVIEN
  { char ht[15];
    int ns, t;
    float cc;
  };
  SVIEN SV;
  printf("\n Nhap ho ten:"); fflush(stdin); gets(SV.ht);
  printf("\n Nhap nam sinh:"); scanf("%d", &SV.ns);
  SV.t=2011-SV.ns;
  printf("\n Nhap chieu cao:"); scanf("%f", &SV.cc);
  printf("\n Ban %s , cao %7.2f m , %7d tuoi", SV.ht, SV.cc, SV.t);
  getch();
}
```

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 kiểudữ liệu *kiểucontrỏ ;
kiểucontrỏ biếncontrỏ ;
```

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

```
kiểodữliệu *biếncontrỏ ;
```

ví dụ

```
typedef float *xyz;
xyz pf;
```

hoặc:

```
float *pf;
```

khai báo pf là biến con trỏ chỉ đến một giá trị kiểu float

Tương tự ta có các khai báo:

```
float cc=1.65;
char nm='0', *pc; // pc là con trỏ kiểu ký tự char
int ns=1991, t, *p, *p2;
```

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

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`

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 <string.h>
main()
{ struct SVIEN
  { char ht[15];
    int ns, t;
    float cc;
  };
  SVIEN *p;
  p=new SVIEN;
  printf("\n Nhập họ ten:"); fflush(stdin); gets((*p).ht);
  printf("\n Nhập nam sinh:"); scanf("%d", &(*p).ns);
  (*p).t=2011-(*p).ns;
  printf("\n Nhập chiều cao:"); scanf("%f", &(*p).cc);
  printf("\n Ban %s , cao %7.2f m , %7d tuoi", (*p).ht, (*p).cc, (*p).t);
  delete p;
  getch();
}
```

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 (nguyên, thực, ký tự . . .),

ví dụ `int ns=1991;`

- 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 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=1.65;`

`float &f = cc;`

Khai báo 2 biến thực `cc` và `f`. Biến tham chiếu `f` sẽ tham chiếu đến biến `cc` cùng kiểu `float`, 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 = -7.6;
thì biến cc sẽ có giá trị mới là -7.6

1.4.3. Ứng dụng kiểu tham chiếu

```
#include <stdio.h>
#include <conio.h>
void DOI(int x, int &y, int *z)
{ printf("\n Con truoc: %d %d %d", x, y, *z);
  x = x+1; y = y+2; *z = *z+3;
  printf("\n Con sau: %d %d %d", x, y, *z);
}
void main()
{ int i=10, j=20, k=30;
  DOI(i, j, &k);
  printf("\n Chinh sau: %d %d %d", i, j, k);
  getch();
}
```

1.5. Đệ qui

1.5.1. Định nghĩa

Một chương trình gọi ngay chính nó thực hiện 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 có kích cỡ

nhỏ hơn

Ví dụ: Bài toán Tháp Hà Nội: Cần chuyển n đĩa từ cọc A (trong đó đĩa lớn ở dưới, đĩa nhỏ ở trên) sang cọc B với các điều kiện:

- . Mỗi lần chỉ được chuyển một đĩa
- . Trên các cọc, luôn luôn đĩa lớn ở dưới, đĩa nhỏ ở trên
- . Được dùng cọc trung gian thứ ba C

Giải: - Tham số hóa bài toán:

Gọi n: là số lượng đĩa cần chuyển
x: cọc xuất phát
y: cọc đích
z: cọc trung gian

Hàm con CHUYEN(n, x, y, z) dùng để chuyển n đĩa từ cọc xuất phát x sang cọc đích y với cọc trung gian z

- Tìm trường hợp dễ nhất: $n = 1$, khi đó ta chuyển đĩa từ cọc x sang cọc y
- Tìm trường hợp tổng quát:

B1: Chuyển n-1 đĩa từ cọc xuất phát x sang cọc trung gian z

B2: Chuyển 1 đĩa từ cọc xuất phát x sang cọc đích y

B3: Chuyển n-1 đĩa từ cọc trung gian z sang cọc đích y

```
#include <stdio.h>
#include <conio.h>
int i;
void CHUYEN(int &n, char x, char y, char z)
{ if (n==1)
  { i++;
    printf("\n %d : %c --> %c", i, x, y);
  }
  else
  { CHUYEN(n-1, x, z, y);
    CHUYEN(1, x, y, z);
    CHUYEN(n-1, z, y, x);
  }
}
void main()
{ int n;
  printf("\n Nhap so dia can chuyen:"); scanf("%d", &n);
  CHUYEN(n, 'A', 'B', 'C');
  getch();
}
```

Chương 2: DANH SÁCH

2.1. Khái niệm

- Danh sách: là một dãy các phần tử $a_0, a_1, a_2, \dots, a_{n-1}$ 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 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 $i-1$ được lưu trữ ngay trước phần tử thứ i giống như 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

Các phần tử của danh sách 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 :

Lê Li	1.7	65
Lê Bi	1.8	75
Lê Vi	1.4	35
Lê Ni	1.6	55
Lê Hi	1.5	45

Khai báo:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
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=0; i<=n-1; 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=0;
  while ( (i<=n-1) && (strcmp(A[i].ht,x)!=0) )
    i++;
  if (i<=n-1) 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 0 ≤ t ≤ n

Khi đó các phần tử từ thứ t đến thứ n-1 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>=0) && (t<=n) )
    { for (i=n-1; 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: 0 ≤ t ≤ n-1

Khi đó các phần tử từ thứ t+1 đến thứ n-1 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>=0) && (t<=n-1) )
    { for (i=t+1; i<=n-1; i++)
        A[i-1]=A[i];
      n--;
    }
}
```

2.2.4. Ưu nhược điểm của danh sách đặc

2.3. Danh sách liên kết

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

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
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;
```

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 dùng để chỉ đến phần tử đầu tiên trong danh sách liên kết

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;
}
```

- 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;
  while (p != NULL)
  { printf("\n %15s  %7.2f  %7d", (*p).ht , (*p).cc , (*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
  F=p;              // 2
}
```

- Thêm một phần tử vào danh sách 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

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

```
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=(*p).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)
  { if (F==t) F=(*t).next;          // 1'
    else (*before).next=(*t).next; // 1
    delete t;
  }
}
```

2.3.4. Ưu nhược điểm của danh sách liên kết

Danh sách có số phần tử là bất kỳ miễn là bộ nhớ đủ lớn, thực hiện nhanh các phép thêm vào và loại bỏ. Tuy nhiên chiếm dụng bộ nhớ nhiều

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 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

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)
    {   printf("\n %15s  %7.2f  %7d", (*p).ht , (*p).cc , (*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)
    {   printf("\n %15s  %7.2f  %7d", (*p).ht , (*p).cc , (*p).cntc);
        p=(*p).next2;
    }
}

```

- 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;
    // Tìm before va after theo họ 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;
    // Tìm 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 có kết nối với phần tử đứng ngay trước và phần tử đứng ngay sau nó

2.5.2. Biểu diễn danh sách liên kết kép

Các khai báo sau định nghĩa 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 một 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 tửq

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);
}

```

- Hủy một phần tử 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 mỗi 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 tốn chi phí $O(n)$.

Bù lại, xâu kép tốn chi phí gấp đôi so với xâu đơn cho việc lưu trữ các mỗi 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)

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 phần tử đầu xâu 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 cuối xâu:


```
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;
    }
}
```

- Hủy phần tử đầu xâu:

```
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;
}
```

- Hủy 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

```
typedef char infor1[15];
typedef float infor2;
typedef int infor3;
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;
    while (p != NULL)
    {
        printf("\n %15s %7.2f %7d", (*p).ht , (*p).cc , (*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;
        delete p;
    }
}
```

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;
  p=Front;
  while (p != NULL)
  { printf("\n %20s  %7.2f  %7d" ,  (*p).ht ,  (*p).cc ,  (*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;
  }
}

```

BÀI TẬP LÝ THUYẾT

BÀI 1: Phân tích ưu, khuyết điểm của xâu liên kết so với mảng. Tổng quát hóa các trường hợp nên dùng xâu liên kết.

BÀI 2: Xây dựng một cấu trúc dữ liệu thích hợp để biểu diễn đa thức $P(x)$ có dạng :

$$P(x) = c_1x^n + c_2x^{n-1} + \dots + c_kx^k$$

Biết rằng:

- Các thao tác xử lý trên đa thức bao gồm :
 - + Thêm một phần tử vào cuối đa thức
 - + In danh sách các phần tử trong đa thức theo :
 - . thứ tự nhập vào
 - . ngược với thứ tự nhập vào
 - + Hủy một phần tử bất kỳ trong danh sách
- Số lượng các phần tử không hạn chế
- Chỉ có nhu cầu xử lý đa thức trong bộ nhớ chính.

a)Giải thích lý do chọn CTDL đã định nghĩa.

b)Viết chương trình con ước lượng giá trị của đa thức $P(x)$ khi biết x .

c)Viết chương trình con rút gọn biểu thức (gộp các phần tử cùng số mũ).

Bài 3: Xét đoạn chương trình tạo một xâu đơn gồm 4 phần tử (không quan tâm dữ liệu) sau đây:

```
Dx = NULL; p=Dx;
Dx = new (NODE);
for(i=0; i < 4; i++)
{
    p = new (NODE);
    p->next = NULL;
}
```

Đoạn chương trình có thực hiện được thao tác tạo nêu trên không ? Tại sao ? Nếu không thì có thể sửa lại như thế nào cho đúng ?

Bài 4: Một ma trận chỉ chứa rất ít phần tử với giá trị có nghĩa (ví dụ: phần tử khác không) được gọi là ma trận thưa.

Dùng cấu trúc xâu liên kết để tổ chức biểu diễn một ma trận thưa sao cho tiết kiệm nhất (chỉ lưu trữ các phần tử có nghĩa).

a)Viết chương trình cho phép nhập, xuất ma trận.

b)Viết chương trình con cho phép cộng hai ma trận.

Bài 5: Bài toán Josephus : có N người đã quyết định tự sát tập thể bằng cách đứng trong vòng tròn và giết người thứ M quanh vòng tròn, thu hẹp hàng ngũ lại khi từng người lần lượt ngã khỏi vòng tròn. Vấn đề là tìm ra thứ tự từng người bị giết.

Ví dụ : $N = 9, M = 5$ thì thứ tự là 5, 1, 7, 4, 3, 6, 9, 2, 8

Hãy viết chương trình giải quyết bài toán Josephus, sử dụng cấu trúc xâu liên kết.

Bài 6: Hãy cho biết nội dung của stack sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong stack in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

Bài 7: Hãy cho biết nội dung của hàng đợi sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào hàng đợi, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong hàng đợi in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

Bài 8: Giả sử phải xây dựng một chương trình soạn thảo văn bản, hãy chọn cấu trúc dữ liệu thích hợp để lưu trữ văn bản trong quá trình soạn thảo. Biết rằng :

- Số dòng văn bản không hạn chế.
- Mỗi dòng văn bản có chiều dài tối đa 80 ký tự.
- Các thao tác yêu cầu gồm :
 - + Di chuyển trong văn bản (lên, xuống, qua trái, qua phải)
 - + Thêm, xóa sửa ký tự trong một dòng

- + Thêm, xoá một dòng trong văn bản
- + Đánh dấu, sao chép khối

Giải thích lý do chọn cấu trúc dữ liệu đó.

Bài 9: Viết hàm ghép 2 xâu vòng L_1 , L_2 thành một xâu vòng L với phần tử đầu xâu là phần tử đầu xâu của L_1 .

BÀI TẬP THỰC HÀNH

Bài 10: Cài đặt thuật toán sắp xếp Chèn trực tiếp trên xâu kép. Có phát huy ưu thế của thuật toán hơn trên mảng hay không ?

Bài 11: Cài đặt thuật toán QuickSort theo kiểu không đệ qui.

Bài 12: Cài đặt thuật toán MergeSort trên xâu kép.

Bài 13: Cài đặt lại chương trình quản lý nhân viên theo bài tập 6 chương 1, nhưng sử dụng cấu trúc dữ liệu xâu liên kết. Biết rằng số nhân viên không hạn chế.

Bài 14: Cài đặt một chương trình soạn thảo văn bản theo mô tả trong bài tập 8.

Bài 15: Cài đặt chương trình tạo một bảng tính cho phép thực hiện các phép tính $+$, $-$, $*$, $/$, \div trên các số có tối đa 30 chữ số, có chức năng nhớ ($M+$, $M-$, MC , MR).

Bài 16: Cài đặt chương trình cho phép nhận vào một biểu thức gồm các số, các toán tử $+$, $-$, $*$, $/$, $\%$, các hàm toán học \sin , \cos , \tan , \ln , \exp , dấu mở, đóng ngoặc $($, $)$ và tính toán giá trị của biểu thức này.

Chương 3: CÂY

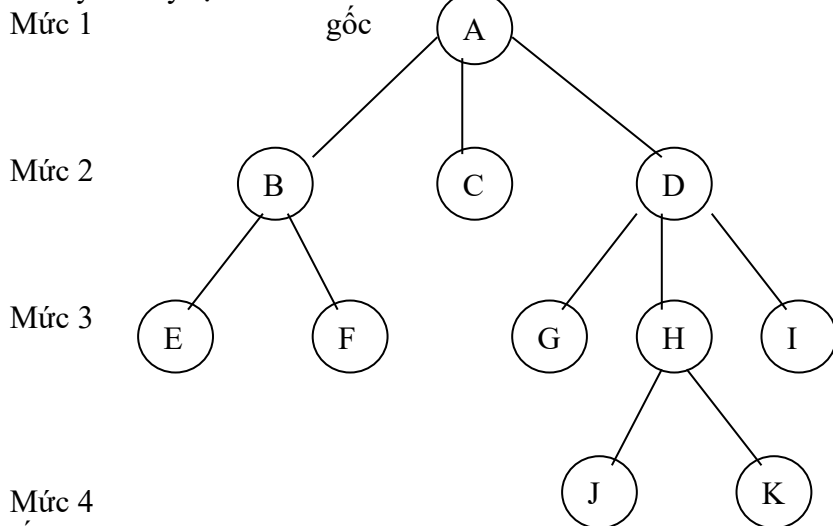
Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu cây. Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn, nó được áp dụng để tổ chức thông tin trong các hệ cơ 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



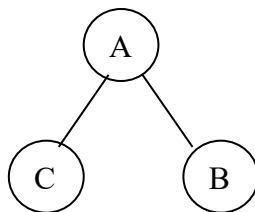
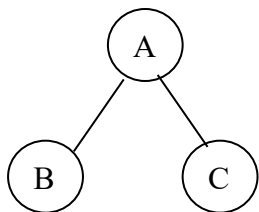
A: nút gốc

A là nút cha của B, C, D

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 F, C, G, J
- 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, \dots, 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: 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:



- Rừng: là tập hợp hữu hạn các cây phân biệt

3.1.2. Các cách biểu diễn cây

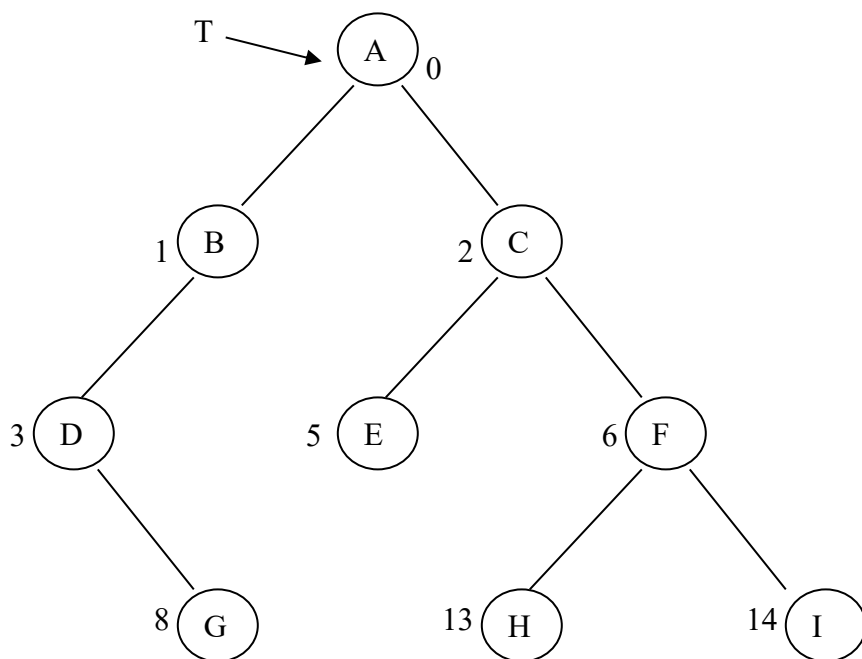
- Biểu diễn cây bằng đồ thị
- Biểu diễn cây bằng giản đồ
- Biểu diễn cây bằng các cặp dấu ngoặc lồng nhau
- Biểu diễn cây bằng phương pháp căn lề
- Biểu diễn cây bằng phương pháp chỉ số

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 zig-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 $h \geq \log_2(n)$

3.2.2. Biểu diễn cây nhị phân

3.2.2.1. Biểu diễn cây nhị phân bằng danh sách đặc

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 con phải của nó. Ta có khai báo sau:

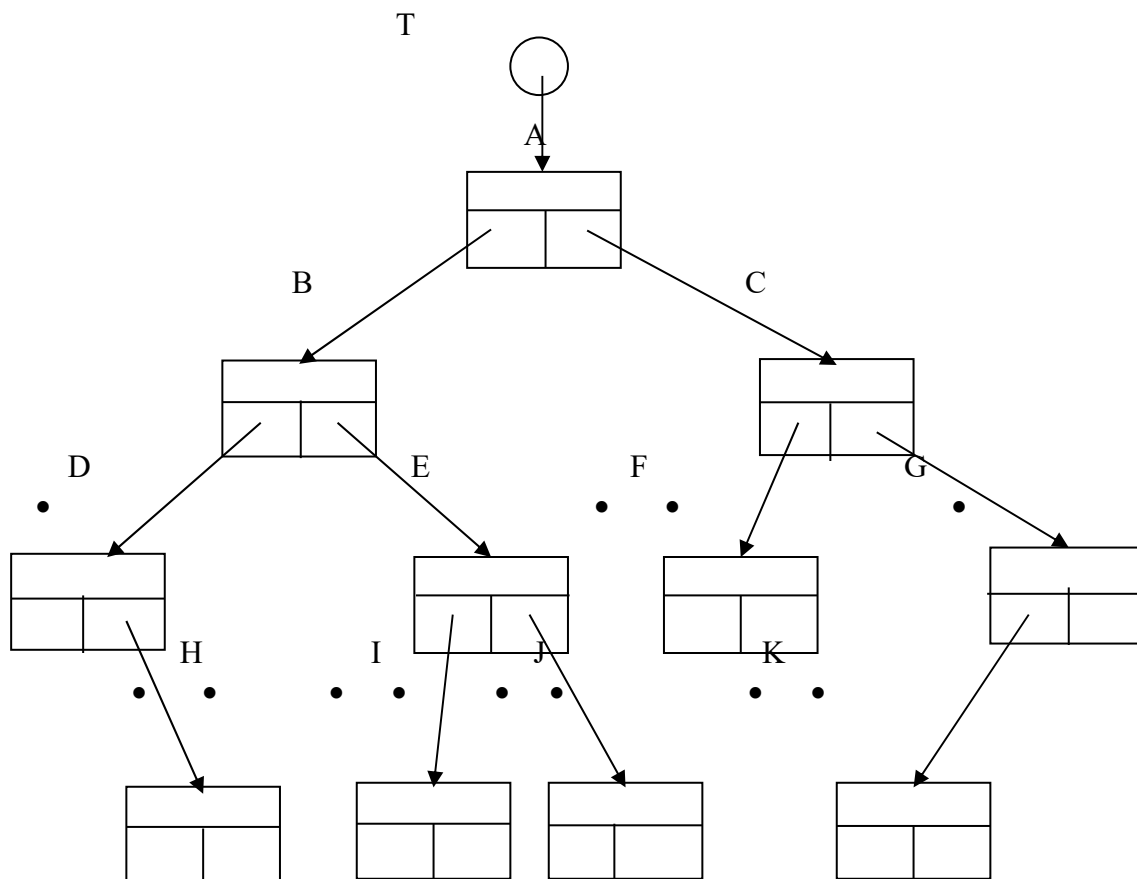
```
struct element
{ char  data;          // trường chứa dữ liệu
  element *left , *right;
};
typedef element *Tree;
Tree  T;
```

Kiểu bản ghi element chứa một nút của cây nhị phân, gồm trường data chứa dữ liệu của nút, hai trường con trỏ left, right lần lượt chứa địa chỉ của nút con trái và địa chỉ của nút con phải của nó

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

Với cách cài đặt này, cấu trúc dữ liệu biểu diễn cây nhị phân trên được minh họa bởi hình sau:



Từ nay về sau chúng ta sẽ chỉ sử dụng cách biểu diễn cây nhị phân bằng con trỏ. Các phép toán đối với cây nhị phân sau này đều được thể hiện trong cách biểu diễn bằng con trỏ.

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 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 ABDGCEFHI

Hàm duyệt có thể trình bày đơn giản như sau:

```
void DuyệtTruoc(Tree T)
{ if (T != NULL)
  { printf("(%T).data);
    DuyệtTruoc( (*T).left );
    DuyệtTruoc( (*T).right );
  }
}
```

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

Hàm duyệt có thể trình bày đơn giản như sau:

```
void DuyệtGiua(Tree T)
{ if (T != NULL)
  { DuyệtGiua( (*T).left );
    printf(( *T).data);
    DuyệtGiua( (*T).right );
  }
}
```

- Duyệt cây theo thứ tự sau: Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm các 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 GDBEHIFCA

Hàm duyệt có thể trình bày đơn giản như sau:

```
void DuyệtSau(Tree T)
{ if (T != NULL)
  { DuyệtSau( (*T).left );
    DuyệtSau( (*T).right );
    printf(( *T).data);
  }
}
```

c. Hàm tạo cây nhị phân mới từ 2 cây nhị phân cho trước

d. Tạo trực tiếp cây nhị phân

3.3. Cây nhị phân tìm kiếm

3.3.1. Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân trong đó 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:

```
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.

Hàm insert trả về giá trị -1, 0, 1 khi không đủ bộ nhớ, gập nút cũ hay thành công:

```
int insertNode(TREE &T, Data X)
{
    if(T) {
        if(T->Key == X) return 0; //đã có
        if(T->Key > X)
            return insertNode(T->pLeft, X);
        else
            return insertNode(T->pRight, X);
    }
    T = new Tnode;
    if(T == NULL) return -1; //thiếu bộ nhớ
    T->Key = X;
    T->pLeft = T->pRight = NULL;
    return 1; //thêm vào thành công
}
```

- Xóa một phần tử trên cây


Việc hủy 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 nó.



Trường hợp cuối cùng: ta không thể hủy trực tiếp do X có đủ 2 con \Rightarrow Ta sẽ hủy gián tiếp. Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y. Phần tử 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 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.

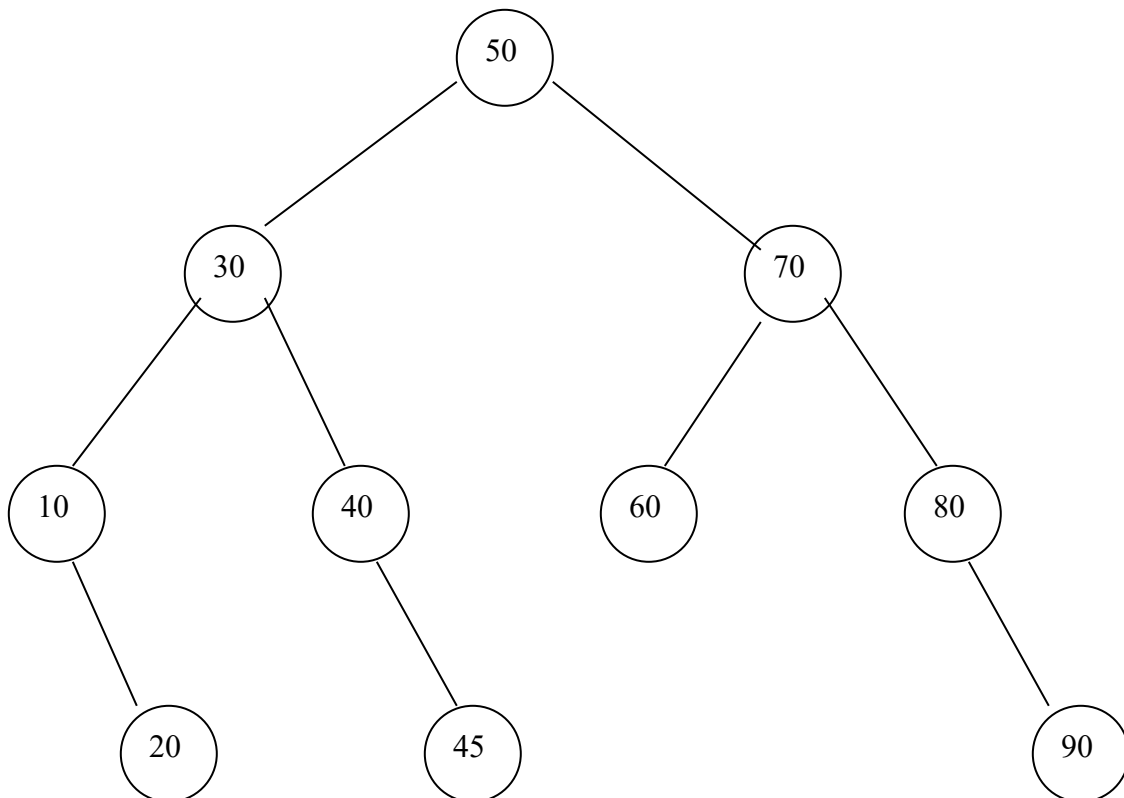
Tuy nhiên, trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 DSLK (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp $O(n)$. Vì vậy cần có cải tiến cấu trúc của CNPTK để đạt được chi phí cho các thao tác là $\log_2(n)$

3.4. Cây nhị phân cân bằng

3.4.1. Cây cân bằng hoàn toàn

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 chênh lệch không quá một so với số nút của cây con phải. 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. Đối với cây cân bằng hoàn toàn, trong 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 $h \approx \log_2 n$. Đây chính là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên CTDL này.

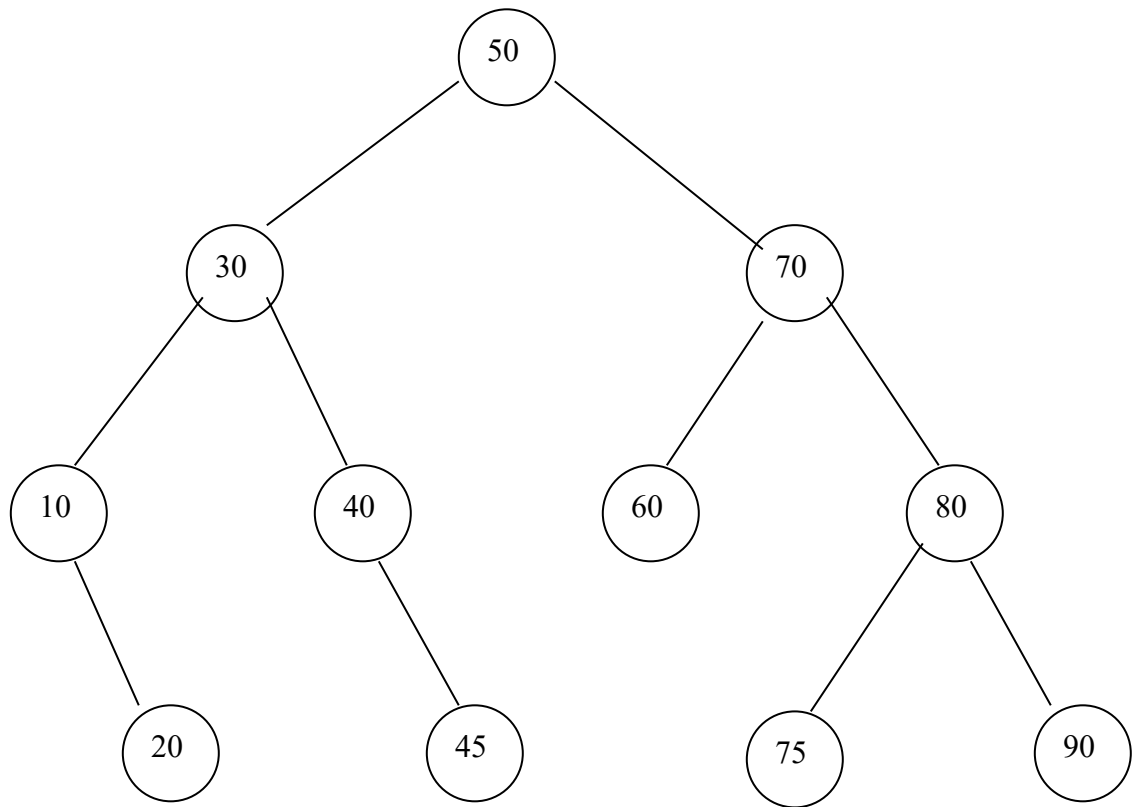
Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một CTDL khác có đặc tính giống CCBHT nhưng ổn định hơn.

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

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 chênh lệch không quá một so với độ cao của cây con phải, ví dụ cho cây cân bằng sau:



Dễ dàng thấy CCBHT là cây cân bằng. Điều ngược lại không đúng, ví dụ cây cân bằng đã cho là không 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 bằng là cây AVL. Từ 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 phải khẳng định cây AVL n nút phải có chiều cao khoảng $\log_2(n)$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ 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:

$$N(h) = 1 + N(h-1) + N(h-2) \quad (1)$$

Ta lại có: $N(h-1) > N(h-2)$

Nên từ (1) suy ra:

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

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

...

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

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

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

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:

$CSCB(p) = 0 \Leftrightarrow \text{Độ cao cây trái}(p) = \text{Độ cao cây phải}(p)$

$CSCB(p) = 1 \Leftrightarrow \text{Độ cao cây trái}(p) < \text{Độ cao cây phải}(p)$

$CSCB(p) = -1 \Leftrightarrow \text{Độ cao cây trái}(p) > \text{Độ cao cây phải}(p)$

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

$p \rightarrow \text{balFactor} = CSCB(p);$

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 hằng 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.

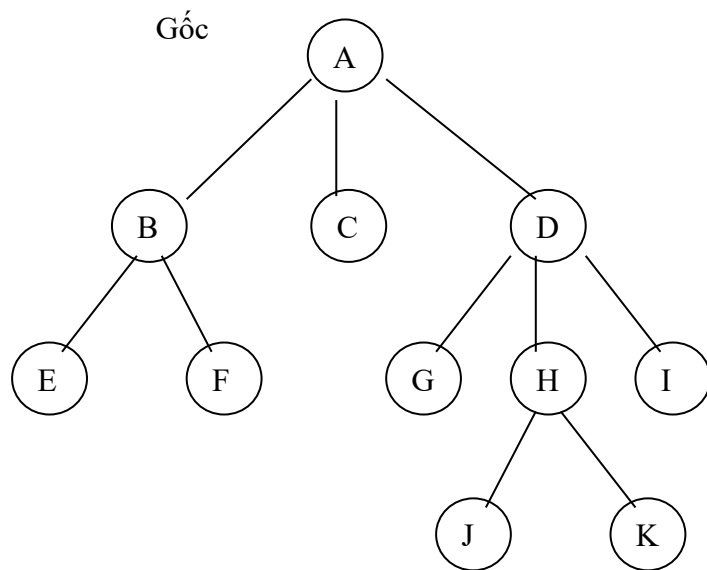
☐ Cây AVL với chiều cao được khống chế sẽ cho phép thực thi các thao tác tìm thêm hủy với chi phí $O(\log_2(n))$ và bảo đảm không suy biến thành $O(n)$.

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

- 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

- 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

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 \Rightarrow 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 bất kỳ thành một cây nhị phân theo qui tắc sau:

- Giữ lại nút con trái nhất làm nút con trái.

- Các nút con còn lại chuyển thành 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.

Ta có thể xem ví dụ dưới đây để thấy rõ hơn qui trình.

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

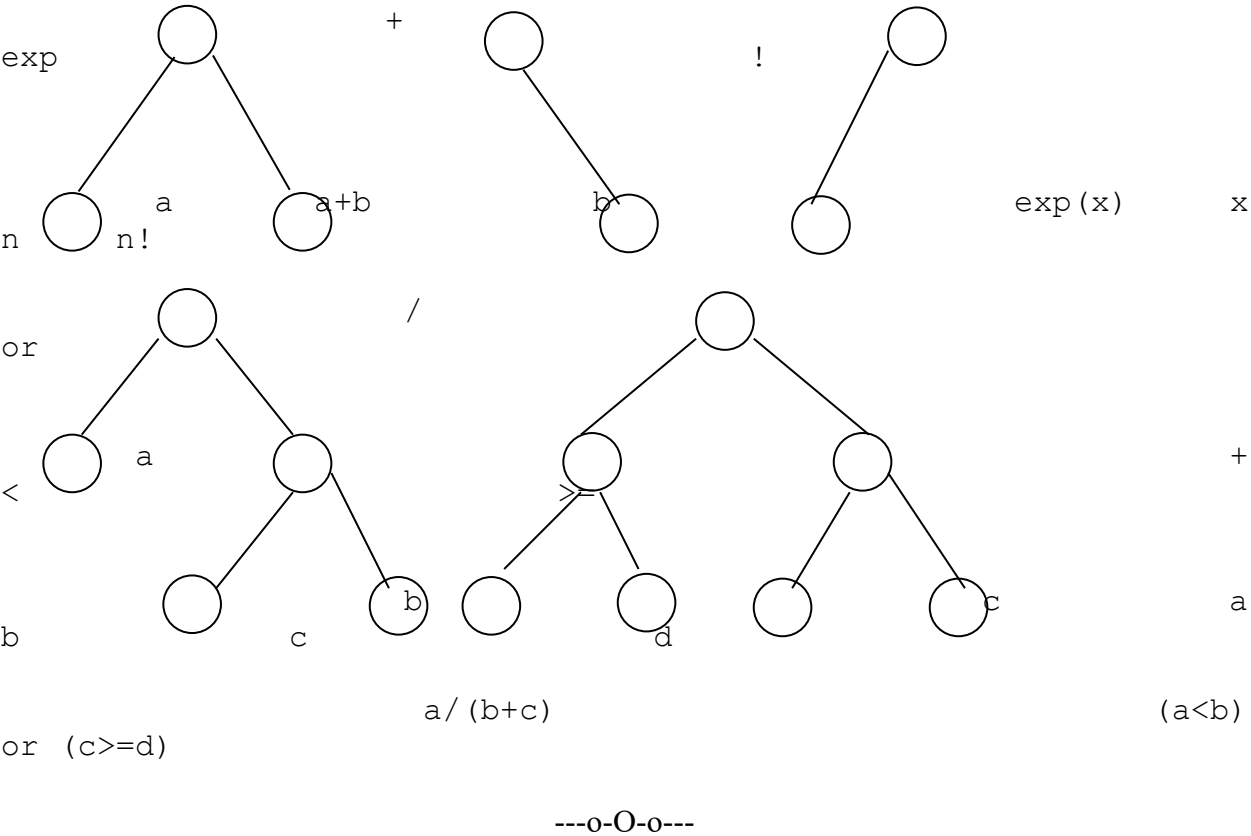
Một ứng dụng quan trọng khác của phép duyệt cây theo thứ tự sau là việc tính toán giá trị của biểu thức dựa trên cây biểu thức

$$(3 + 1) \times 3 / (9 - 5 + 2) - (3 \times (7 - 4) + 6) = -13$$

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).



BÀI TẬP LÝ THUYẾT

Bài 1. Hãy trình bày các vấn đề sau đây:

- Định nghĩa và đặc điểm của cây nhị phân tìm kiếm.
- Thao tác nào thực hiện tốt trong kiểu này.
- Hạn chế của kiểu này là gì ?

Bài 2. Xét thuật giải tạo cây nhị phân tìm kiếm. Nếu thứ tự các khóa nhập vào là như sau:

8 3 5 2 20 11 30 9 18 4

thì hình ảnh cây tạo được như thế nào ?

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau :

15, 20

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ (nêu rõ phương pháp hủy khi nút có cả 2 cây con trái và phải)

Bài 3. Áp dụng thuật giải tạo cây nhị phân tìm kiếm cân bằng để tạo cây với thứ tự các khóa nhập vào là như sau :

5 7 2 1 3 6 10

thì hình ảnh cây tạo được như thế nào ? Giải thích rõ từng tình huống xảy ra khi thêm từng khóa vào cây và vẽ hình minh họa.

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau :

5, 6, 7, 10

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ và giải thích

Bài 4. Viết các hàm xác định các thông tin của cây nhị phân T:

- Số nút lá
- Số nút có đúng 1 cây con
- Số nút có đúng 2 cây con
- Số nút có khóa nhỏ hơn x (giả sử T là CNPTK)
- Số nút có khóa lớn hơn x (giả sử T là CNPTK)
- Số nút có khóa lớn hơn x và nhỏ hơn y (T là CNPTK)
- Chiều cao của cây

- h. In ra tất cả các nút ở tầng (mức) thứ k của cây T
- i. In ra tất cả các nút theo thứ tự từ tầng 0 đến tầng thứ h-1 của cây T (h là chiều cao của T).
- j. Kiểm tra xem T có phải là cây cân bằng hoàn toàn không.
- k. Độ lệch lớn nhất trên cây. (Độ lệch của một nút là độ lệch giữa chiều cao của cây con trái và cây con phải của nó. Độ lệch lớn nhất trên cây là độ lệch của nút có độ lệch lớn nhất).

Bài 5. Xây dựng cấu trúc dữ liệu biểu diễn cây N-phân ($2 < N \leq 20$).

- a. Viết chương trình con duyệt cây N-phân và tạo sinh cây nhị phân tương ứng với các khoá của cây N-phân.
- b. Giả sử khóa được lưu trữ chiếm k byte, mỗi con trỏ chiếm 4 byte, vậy dùng cây nhị phân thay cây N-phân thì có lợi gì trong việc lưu trữ các khoá ?

Bài 6. Viết hàm chuyển một cây N-phân thành cây nhị phân.

Bài 7.Viết hàm chuyển một cây nhị phân tìm kiếm thành xâu kép có thứ tự tăng dần.

Bài 8.Giả sử A là một mảng các số thực đã có thứ tự tăng. Hãy viết hàm tạo một cây nhị phân tìm kiếm có chiều cao thấp nhất từ các phần tử của A.

Bài 9.Viết chương trình con đảo nhánh (nhánh trái của một nút trên cây trở thành nhánh phải của nút đó và ngược lại) một cây nhị phân .

Bài 10.Hãy vẽ cây AVL với 12 nút có chiều cao cực đại trong tất cả các cây AVL 12 nút.

Bài 11.Tìm một dãy N khóa sao cho khi lần lượt dùng thuật toán thêm vào cây AVL để xen các khóa này vào cây sẽ phải thực hiện mỗi thao tác cân bằng lại(LL,LR, RL,RR) ít nhất một lần.

Bài 12.Hãy tìm một ví dụ về một cây AVL có chiều cao là 6 và khi hủy một nút lá (chỉ ra cụ thể) việc cân bằng lại lan truyền lên tận gốc của cây. Vẽ ra từng bước của quá trình hủy và cân bằng lại này.

BÀI TẬP THỰC HÀNH:

Bài 13.Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

Bài 14.Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL.

Bài 15.Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh- Việt.

Bài 16.Viết chương trình khảo sát tần xuất xảy ra việc cân bằng lại của các thao tác thêm và hủy một phần tử trên cây AVL bằng thực nghiệm. Chương trình này phải cho phép tạo lập ngẫu nhiên các cây AVL và xóa ngẫu nhiên cho đến khi cây rỗng. Qua đó cho biết số lần xảy ra cân bằng lại trung bình của từng thao tác.

---o-O-o---