Data Structure and Algorithm

Gồm Dynamic Data Structure và Static Data Structure (lưu ý để tìm hiểu sau).

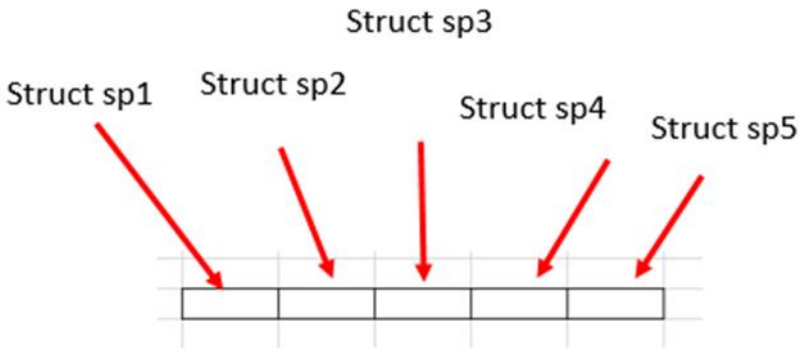
Giới thiệu

Chương trình A **đếm số lượng của 1 sản phẩm**, dữ liệu ở đây là 1 số nguyên, khai báo 1 biến int count để **lưu trữ 1 số lượng sản phẩm**.	Chương trình B yêu cầu **đếm số lượng của nhiều sản phẩm**, tương tự chương trình A ta khai báo int count1 và int count 2 để lưu trữ sản phẩm 1 sản phẩm 2, nhưng như vậy dữ liệu bị phân mảnh và khó xử lý vì vậy ta cần sử dụng mảng các số nguyên, các phần tử mảng sẽ lưu trữ số lượng của 1 sản phẩm	Chương trình C yêu cầu quản lý thông tin của nhiều sản phẩm như ID, số lượng, giá của sản phẩm Tương tự chương trình B ta có thể sử dụng mảng

Mỗi mảng gồm chứa 1 thông tin của các sản phẩm:

Mảng 1 chứa thông tin ID.
Mảng 2 chứa thông tin số lượng.

- Tốt nhất là ta sử dụng 1 mảng duy nhất, mỗi phần tử của mảng là 1 struct chứa thông tin của sản phẩm.
- Struct này sẽ chứa các thông tin lưu trữ ID, số lượng và giá của sản phẩm

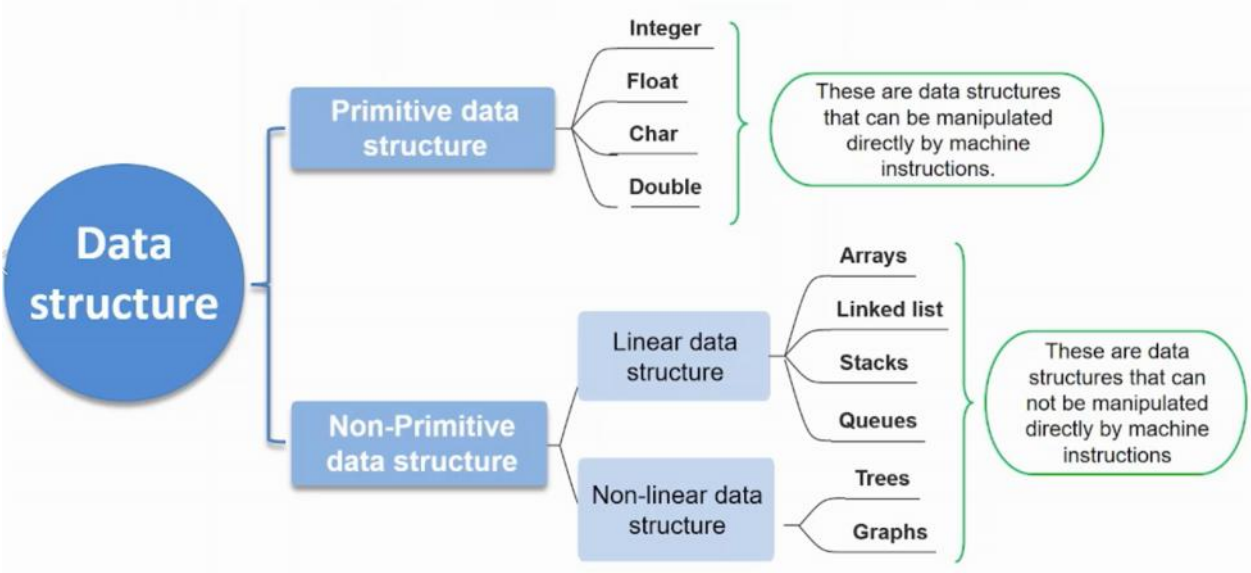


Qua các ví dụ trên ta thấy 1 chương trình có thể có các cách sử dụng dữ liệu tuỳ theo yêu cầu đặt ra.

Mỗi cách có 1 dạng dữ liệu riêng và nó được gọi là 1 **cấu trúc dữ liệu** (data struture).

→**cấu trúc dữ liệu** là 1 định dạng riêng biệt về tổ chức và lưu trữ dữ liệu, nhờ đó ta có thể truy cập và làm việc theo những cách thích hợp để làm cho 1 chương trình hoạt động hiệu quả.

Cấu trúc dữ liệu chia làm 2 loại, **kiểu nguyên thuỷ và ko nguyên thuỷ**



Cấu trúc dữ liệu kiểu nguyên thuỷ bao gồm các kiểu số nguyên, thực, ký tự, các cấu trúc dữ liệu này có thể được thao tác trực tiếp bằng máy tính ví dụ như chúng ta có thẻ cộng trực tiếp 2 số nguyên vào dấu cộng.

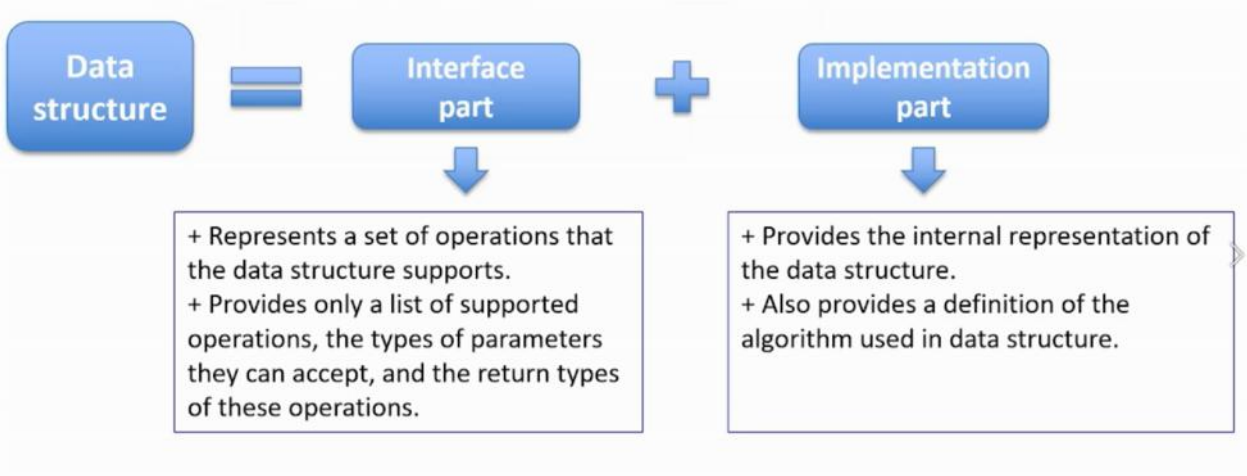
Cấu trúc dữ liệu kiểu ko nguyên thuỷ chia làm 2 loại: tuyến tính và ko tuyến tính.

cấu trúc dữ liệu được coi là **tuyến tính** khi các thành phần dữ liệu được xây dựng thành 1 chuỗi danh sách liên tiếp nhau như mảng (arrays), danh sách liên kết (linked list), ngăn xếp (stack), queues.

Cấu trúc dữ liệu được coi là ko **tuyến tính** khi các thành phần dữ liệu ko được tổ chức thành 1 chuỗi các thứ tự như dạng cây (tree), đồ hoạ (graph).

Những cấu trúc dữ liệu ko nguyên thuỷ này ko thể được thao tác trực tiếp bằng các câu lệnh của máy tính, ví dụ như chúng ta ko thể cộng trực tiếp 2 mảng với nhau bằng dấu cộng.

Thiết kế của cấu trúc dữ liệu



Gồm 2 phần:

Interface part là phần giao diện:

- Biểu diễn tập hợp gồm các thao tác mà cấu trúc dữ liệu đó hỗ trợ.
- Cung cấp 1 danh sách các thao tác các kiểu tham số mà chúng có thể chấp nhận và kiểu trả về của các thao tác này

Implementation part là phần triển khai:

- Cung cấp sự biểu diễn bên trong của cấu trúc dữ liệu.
- Cung cấp định nghĩa các thuật toán được sử dụng trong cấu trúc dữ liệu đó mà nó nên độc lập với interface part.

Ví dụ: các chương trình sử dụng collection,

Là 1 tập hợp gồm danh sách các đối tượng như 1 **chương trình quản lý lớp học**: các tập sử dụng danh sách các học sinh các môn học, danh cách các thao tác cơ bản như tạo mới 1 danh sách, thêm mới 1 đối tượng và danh sách, tìm kiếm 1 đối tượng trong danh sách, xoá bỏ 1 đối tượng khỏi danh sách và xoá bỏ toàn bộ danh sách.

Interface part ở đây ta hiểu là:

Cung cấp 1 header file chứa khai báo tất cả các hàm: create, add, find	⇔	Biểu diễn tập hợp gồm các thao tác mà cấu trúc dữ liệu đó hỗ trợ
Đồng thời cũng định nghĩa các kiểu dữ liệu được sử dụng ví dụ như hàm add, ta cần chỉ ra tham số đầu vào là đối tượng học sinh, kiểu trả về là boolean để thông báo đã thêm vào thành công hay chưa	⇔	Cung cấp 1 danh sách các thao tác các kiểu tham số mà chúng có thể chấp nhận và kiểu trả về của các thao tác này

Implementation part ở đây có thể hiểu là:

Triển khai cho source code. Ví dụ như hàm Find ta có thể sử dụng thuật toán tìm kiếm tuần tự hoặc tìm kiếm nhị phân.	⇔	Cung cấp sự biểu diễn bên trong của cấu trúc dữ liệu.

Phần 2: Commonly used Data Structures:

Mảng là dạng đơn giản nhất của tập hợp, mỗi **đối tượng** trong mảng được gọi là các **phần tử mảng**, mỗi phần tử mảng có cùng 1 kiểu dữ liệu mặc dù chúng có các giá trị khác nhau.

Từng phần tử được **truy cập theo chỉ mục** bằng cách sử dụng 1 loạt các số nguyên liên tiếp nhau gọi là index.

Khai báo mảng 1 chiều A gồm 10 phần tử:

```
Int A[10];
For (I = 0; I < 10; I++)
{
        A[i]= I +1;
}
```

Trong 1 mảng, các phần tử được lưu trữ liên tiếp nhau trong bộ nhớ và kích thước của các phần tử đều bằng nhau.

Phần tử có thể là các kiểu dữ liệu cơ bản như int, short, long hoặc cũng có thể là 1 pointer hoặc struct.

Bởi vì các phần tử có cùng kích thước bộ nhớ nên ta có thể **sử dụng pointer để truy cập mảng.**

Ví dụ ta khai báo 1 mảng số nguyên A có 10 phần tử:

```
int A[10];
int N = 10;
```

1 con trỏ kiểu số nguyên int để trỏ đến A

```
int *p;
p = &A[0];
```

Con trỏ p trỏ đến từng phần tử trong mảng và gán giá trị từ 1 đến 10 cho từng mà con trỏ trỏ đến.

```
For (i=0; i < N; i++){
        *(p+1) = i+ 1;
}
```

Câu hỏi: Nếu con trỏ truy cập vượt quá kích thước của mảng thì chuyện gì sẽ xảy ra?

Trả lời: Khi đó ta có thể đang đọc các giá trị ko biết trước vì đang truy cập đến địa chỉ của vùng nhớ ko xác định và có thể ghi đè lên giá trị của 1 biến khác hoặc gặp lỗi nếu như con trỏ đang trỏ đến 1 vùng nhớ cấm truy cập (0xC00000005).

Mảng đa chiều

Ma trận 3x3 có 3 hàng

A11 a12 a13

A21 a22 a23

A31 a32 a33

Có thể lưu trữ bằng mảng 2 chiều: 3 hàng 3 cột

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

Hoặc có thể khởi tạo với số phần tử bằng tích của kích thước các chiều:

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

Nếu khởi tạo số phần tử ít hơn hoặc nhiều hơn số lượng mà mảng cần có, nếu nhiều hơn thì complier cảnh báo, nếu ít hơn thì ví trí các phần tử còn thiếu sẽ chứa giá trị 0 hoặc giá trị rác.

Cách truy cập phần tử của mảng đa chiều

In ra ma trận 5 hàng 2 cột

```
Int a[5][2] = {{0,0},{1,2},{2,4},{3,6},{4,8}Ư;
Int I, j;
/*Output each array element’s value*/
```

```
For (I = 0; i<5;i++)
{
    For(j=0;j<2;j++)
    {
        Printf(“a[%d][%d] = %d\n”,I,j,a[i][j]);
    }
}
```

Sử dụng mảng thì rất rõ ràng và nhanh chóng nhưng cần xác định kích thước mảng khi khởi tạo, nếu muốn di chuyển 1 phần tử trong danh sách thì cần di chuyển các phần tử trước khi thêm vào, xoá cũng vậy

Như vậy trung bình thì ta cần di chuyển 1 nửa số lượng phần tử. trong trường hợp xấu nhất là thêm 1 phần tử vào vị trí đầu tiên, ta cần di chuyển toàn bộ phần tử hiện tại. nếu như chương trình yêu cầu thêm xoá bỏ phần tử được thường xuyên nó có thể làm tăng thời gian chạy của chương trình, cần cân nhắc khi sử dụng mảng có số lượng lớn

Mảng cũng ko thể mở rộng tự động, vì vậy khi cần mở rộng ta cần cấp phát 1 mảng mới có kích thước lớn hơn, sau đó sao chép các phần tử ở mảng cũ sang mảng mới, điều này làm giảm hiệu năng của chương trình.

Để tránh các hạn chế của mảng ở trên, chúng ta có thể sử dụng linked list, là 1 kiểu dữ liệu linh động rất linh hoạt. các phần tử có thể được thêm vào hoặc xoá đi tuỳ ý.

Với link list, các phần tử được cấp phát động khi cần thiết và số lượng các phần tử thêm vào danh sách chỉ bị giới hạn bởi bộ nhớ còn khả dụng

Linked list được hiểu là các nodes liên kết với nhau, mỗi phần tử là 1 node, mỗi node bao gồm 2 phần: đối tượng dữ liệu (data item) là phần chứa data của phần tử hoặc nó cũng có thể là 1 con trỏ để trỏ đến 1 đối tượng dữ liệu

Và 1 con trỏ next để trỏ đến node tiếp theo trong danh sách, có tác dụng để liên kết node khác trong danh sách.

Node cuối cùng trong danh sách sẽ chứa con trỏ, trỏ đến NULL để chi ra rằng nó là cuối cùng của danh sách hay còn gọi là tail (đuôi).

Linked list có 1 con trỏ trỏ đến list gọi là head và ban đầu được khởi tạo bằng NULL

Để thêm phần tử đầu tiên vào list, ta làm theo các bước:

Bước 1: cấp phát vùng nhớ cho node, trỏ con trỏ data của node đến đối tượng dữ liệu và trỏ con trỏ next của node đến NULL. Trỏ con trỏ head đến node.

Cần nhấn mạnh rằng để đại diện cho 1 danh sách chỉ đơn giản là 1 con trỏ trỏ đến node đầu tiên của list gọi là head

Để thêm 1 phần tử đầu tiên vào list, ta làm theo các bước: cấp phát vùng nhớ cho node, trỏ con trỏ dữ liệu của node cầ thêm vào đối tượng dữ liệu. trỏ con trỏ next của node tới head hiện tại, trỏ con trỏ head đến node. Như vậy node mới thêm vào sẽ liên kết với node đầu của list ban đầu, con trỏ head sẽ trỏ đến node mới thêm vào và lúc này node đó trở thành node đứng đầu của list

Ví dụ: đầu tiên định nghĩa kiểu của node là struct:

```
Struct t_node{
        Void *item;
        Struct t_node *next;
}
```

T_node là 1 struct gồm 1 item dùng để chứa data của node, tuỳ thuộc vào data cần lưu trữ, nó có thể là 1 integer, 1 char, 1 struct.

ở đây ta dùng 1 con trỏ để trỏ vào data, 1 con trỏ next kiểu t_node để linked tới node khác. ở đây để khai báo 1 con trỏ để trỏ đến kiểu struct của chính nó và ngôn ngữ C cho phép điều này. Tiếp theo ta định nghĩa lại 1 kiểu con trỏ struct t_node là Node để cho việc sử dụng thuận tiện:

```
typedef struct t_node *Node;
```

ta định nghĩa 1 kiểu struct collection bao gồm Node head đại diện cho linked list, ngoài ra ta có thể định nghĩa 1 việc khác như kích thước của collection hoặc ID của collection

```
struct collection{
Node head;
….
}
```

Tiếp theo là triển khai hàm addđ

```
Void AddToCollection (Collection c, void *item)
{
```

Đầu tiên ta cấp phát 1 vùng nhớ cho node new

Node new = malloc(sizeof(struct t_node));

Tiếp theo ta gán con trỏ item của node new tới node mà hàm add truyền vào

New->item = item; đây là 1 con trỏ trỏ đến data mà ta muốn add vào list và con trỏ node new trỏ đến head của collection

New->next = c->head

Gán head của collection vào node new

c->head = new

}

Như vậy đã implement hàm add

Sau đây là triển khai hàm find

Tìm kiếm 1 node mà data của nó thoả mãn 1 điều kiện nào đó, ví dụ ta có list chứa thông tin các thông tin sinh viên của 10 lớp học, mỗi lớp có data của 1 sinh viên gồm hộ tên, mã sinh viên của trường, điểm trung bình, ta có thể tìm và lấy ra thông tin của 1 sinh viên dựa vào mã sinh viên.

Đầu tiên ta khởi tạo 1 con trỏ N trỏ đến head của phần tử đầu tiên của danh sách. Như ta đã biết, danh sách sẽ có node cuối cùng trỏ đến NULL hoặc danh sách rỗng cũng có giá trị NULL.

```
Void *FindinC (Collection c, void *key){

Node n = c->head;

While (n != NULL){

If(KeyCmp(ItemKey(n->item),key)==0){

Return n -> item;

}

n=n-> next;

}

Return NULL;

}
```

Cho nên ta dùng vòng lặp while để kiểm tra n, nếu n khác NULL, ta tiếp tục kiểm tra data item của node có thoả mãn điều kiện khi truyền vào hay không, nếu thoả mãn, ta trả về node hiện tại và kết thúc. Nếu không ta tiếp tục gán n bằng next n để tra phần tử tiếp theo trong danh sách cho đến khi n bằng NULL, có nghĩa là không tìm thấy node nào thoả mãn thì hàm sẽ trả về giá trị NULL

Tiếp theo triển khai hàm Delete chỉ đơn giản là sự xoá bỏ sự liên kết giữa 2 node và liên kết lại đến 1 node khác

Ví dụ về xoá 1 node mà data của nó thoả mãn 1 điều kiện key như hàm find phần trước,
Void *DeleteFormC (Collection c, void *key){

ta sử dụng 2 con trỏ n, prev cùng gán đến giá trị head của danh sách.

Node n, prev = c->head;

Trước hết ta kiểm tra node đầu tiên của danh sách node n có khác NULL hay ko

If (n!= NULL){

Nếu khác NULL ta tiếp tục kiểm tra data item của node n có thoả mãn hay ko bằng điều kiện của giá trị key truyền vào hay không

If (KeyCmpItemKey(n->item),key)==0)

{

Nếu thoả mãn, ta trỏ head đến n next, trả về node n và kết thúc

c->head = n->next;

return n;

nghĩa là head sẽ trỏ đến phần tử thứ 2 của list hay nói cách khác node đầu tiên đã bị xoá bỏ khỏi danh sách. Nếu node n đầu tiên ko thoả mãn, ta gán giá trị n = next n

}

N = n->next; lúc này prev trỏ đến phần tử đầu tiên, n trỏ đến phần tử thứ 2 trong danh sách

}

Sau đó ta dùng vòng lặp while để kiểm tra node n,

While(n!=NULL){

Nếu node n khác nULL, ta tiếp tục

If (KeyCmpItemKey(n->item),key)==0){

Kiểm tra data item của node có thoả mãn điều kiện của giá trị key truyền vào hay không

Prev ->next = n -> next;

Return n;

Nếu thoả mãn ta gán next của prev vào next của n sau đó return n là node vừa bị xoá bỏ khỏi list và kết thúc, nếu ko thoả mãn

}

Prev = n;

N = n->next;

Ta gán prev với n và n gán với next n; để kiểm tra đến phần tử tiếp theo trong list

}

Return NULL;

}sau khi duyệt hết danh sách, n đã bằng NULL có nghĩa là ko có phần tử nào thoả mãn được điều kiện để xoá nên nó phải trả về giá trị NULL.

Như vậy node đó bị xoá khỏi danh sách nhưng nó vẫn còn lưu lại trên bộ nhớ. Vậy làm thế nào để giải phóng bộ nhớ cho node đó?

Nếu như phần item của node đó là 1 con trỏ, trỏ đến 1 đối tượng của data khác, ta cần dùng lệnh free để free data đằng trước 1 con trỏ item và sau đó ta sẽ free cho node.

1 số biến thể của linked list.

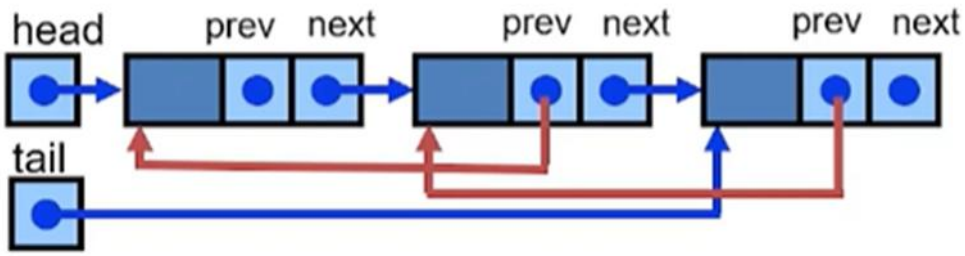
Đơn giản nhất là phần tử cuối cùng của danh sách sẽ được thêm vào đầu và được truy cập đầu tiên. (last in first out)

Chúng ta có thể thay đổi sang dạng first in first out bằng sách sử dụng thêm 1 con trỏ tail để trỏ tới phần tử cuối cùng của danh sách, khi thêm vào, ta sẽ thêm vào tail và khi truy cập sẽ bắt đầu từ head, bằng cách đó, phần tử được thêm vào danh sách sẽ được truy cập trước

Nếu để con trỏ next của tail luôn trỏ đến Head thay vì vị trí NULL, ta đã tạo ra 1 danh sách liên kết vòng. Khi đó head sẽ là tail next và khi đó ta có thể triển khai dạng fifo hay lifo tuỳ ý với chỉ 1 con trỏ.

Danh sách liên kết đôi

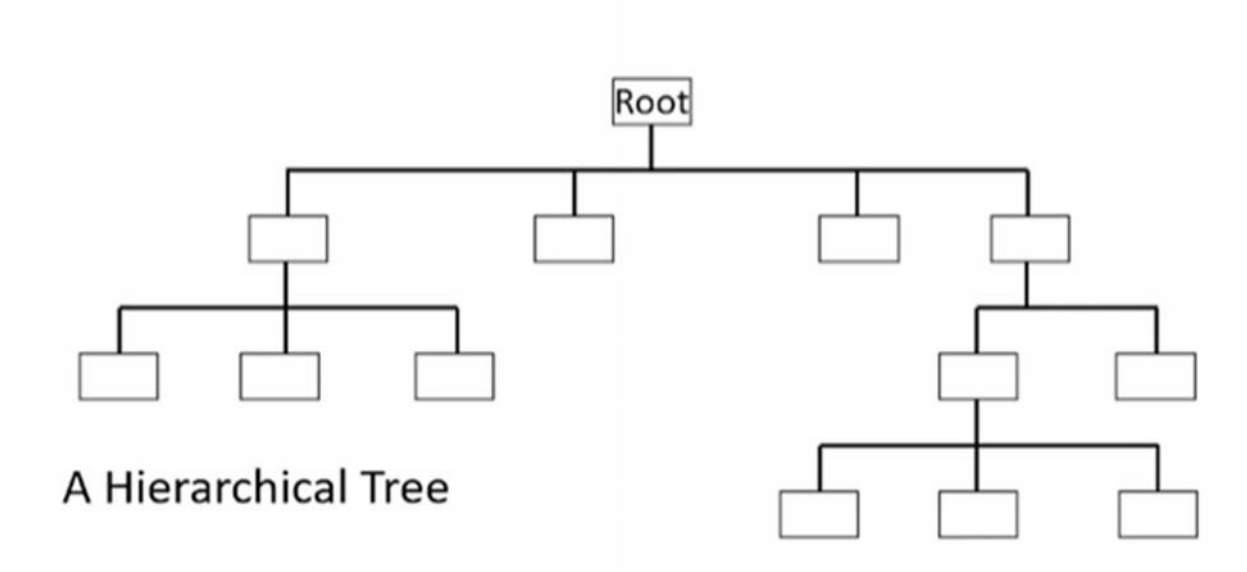
Để tạo danh sách liên kết đôi, mỗi node ta sử dụng thêm 1 con trỏ previous để trỏ đến node phía trước của nó



Vì vậy danh sách này có thể bắt đầu được truy cập theo cả 2 chiều từ head hoặc từ tail. Nó được sử dụng cho các ứng dụng yêu cầu truy cập được list theo cả 2 chiều, ví dụ như tìm kiếm tên có trong danh bạ đã được sắp xếp

Cấu trúc dữ liệu không tuyến tính là tree (dạng cây), tree là 1 tập hợp hữu hạn các phần tử không rỗng, trong đó có 1 phần tử được gọi là ROOT và các phần tử còn lại được phân chia thành m tập con rời rạc, và mỗi tập con đó tự nó là 1 tree

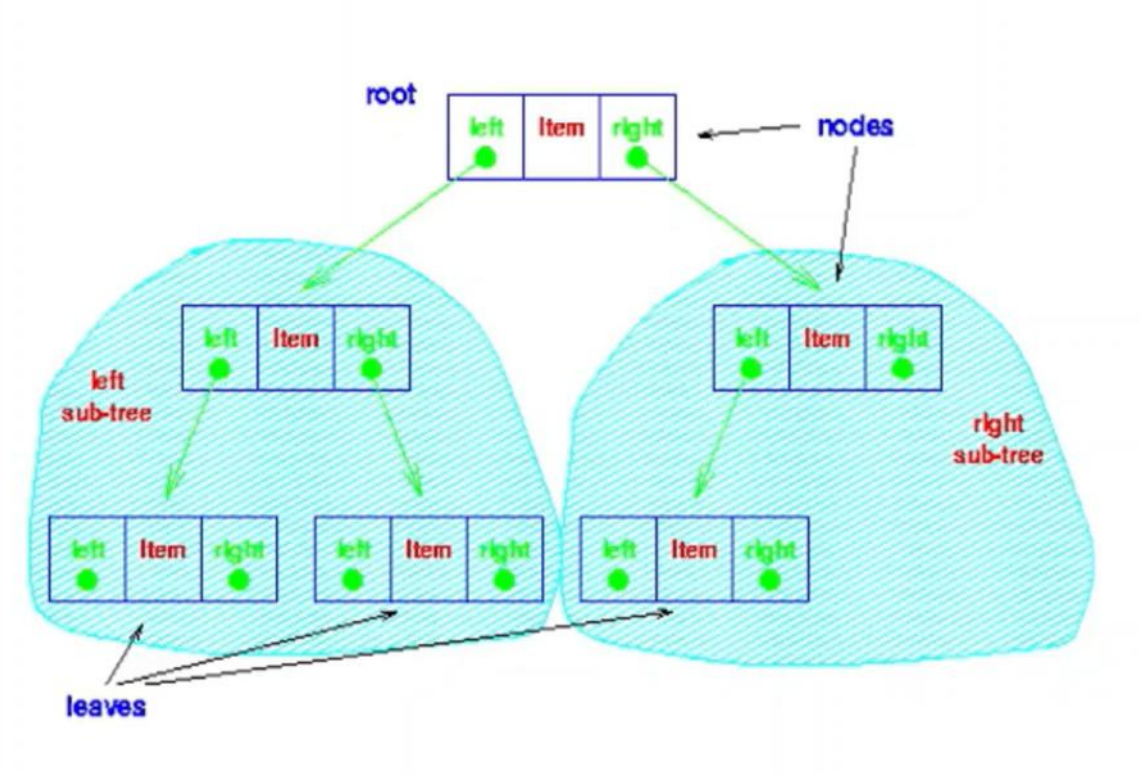
Nhìn vào sơ đồ ta thấy từ 1 ROOT nó sẽ chia thành các nhánh nhỏ



Và các nhánh nhỏ có thể được chia thành các nhánh nhỏ hơn, ta thấy hình dạng tương tự như 1 cái cây.

Kiểu cấu trúc dữ liệu này cũng được chia thành nhiều dạng nhỏ hơn như binary tree, n – array tree, red – black tree, …

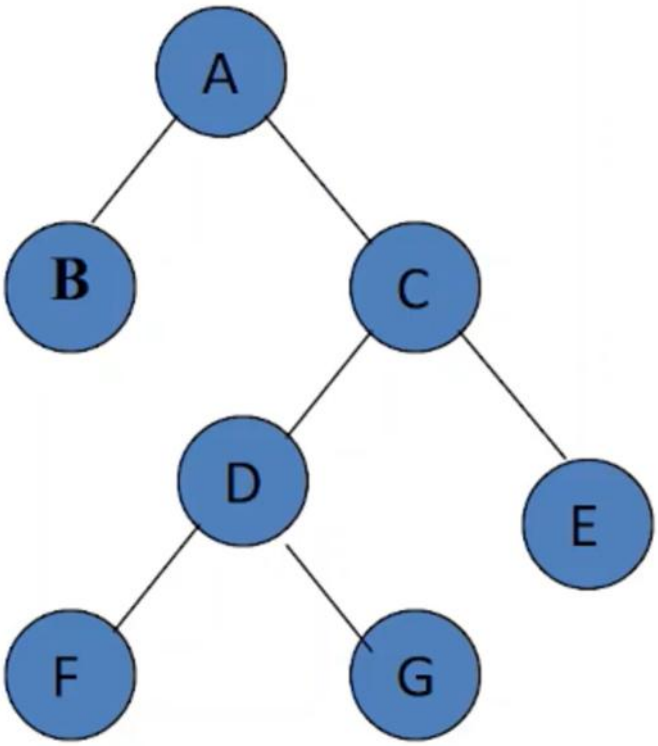
Dạng đơn giản nhất của tree



Đó là binary tree, 1 binary tree bao gồm 1 node gọi là root node, left và right sub tree, và 2 cây con này đều là binary tree. Những node ở vị trí thấp nhất của cây, node mà ko có cây con được gọi là các node lá.

1 cây được gọi là cây nhị phân có thứ tự nếu như key của tất cả các nodes trong cây con bên trái bé hơn node root. Và key của các node bên trong cây con bên phải lớn hơn root node và 2 cây con tự nó đều là cây nhị phân có thứ tự.

Các khái niệm có trong binary tree. 2 node là anh em nếu nó là 1 node trái phải của 1 node cha.



Node n1 là node tổ tiên của node n2 và con cháu của n1. Nếu n1 là node cha của node n2 hoặc là cha của node tổ tiên của n2

Ví dụ A là 1 root của cây nhị phân, B là cây con bên trái của A. như vậy, A là cha của B và B là con của A, E là con của C và C là con của A, như vậy A là tổ tiên của E và E là con cháu của A.

1 cây nhị phân đầy đủ hoặc nghiêm ngặt nếu tất cả các node ko phải node lá sẽ có 2 node con bên trái và bên phải hay nói cách khác, bất kỳ 1 node cho cây đều phải có 2 node con bên trái phải hoặc ko có node con nào, ví dụ hình trên là 1 cây nhị phân đầy đủ

Level: mức của node, node ROOT sẽ có level bằng 0. Level của 1 node bất kỳ sẽ lớn hơn 1 đơn vị so với node của cha nó. Ví dụ node A có level 0 và node B và C sẽ có level 1. Node D và E là 2.

Depth: độ sâu, là level lớn nhất của các node trong tree, cũng có thể gọi là chiều cao của cây.

Hình trên có depth là 3 -> 1 cây nhị phân có thể chứa 2 mũ I node tại level I, ví dụ tại level 1, cây có 2 mũ 1 node chính là bằng 2 node B và C

Tổng tối đa các node có trong cây nhị phân có độ sâu

$d = 2^{d+1} - 1$ (node)

Cây có độ sâu là 2 thì sẽ có 7 node

Ví dụ triển khai cho cây nhị phân

Định nghĩa 1 struct tên node bao gồm

Struct t_node {

1 con trỏ item để trỏ đến phần data của node

Void *item;

2 con trỏ t_node để link node con bên trái và phải của node

Struct t_node *left;

Struct t_node *right;L

};

Định nghĩa lại kiểu con trỏ t_node là Node cho dễ sử dụng

Typedef struct t_node *Node;

Định nghĩa 1 collection bao gồm 1 nút root trỏ đến gốc của tree (root)

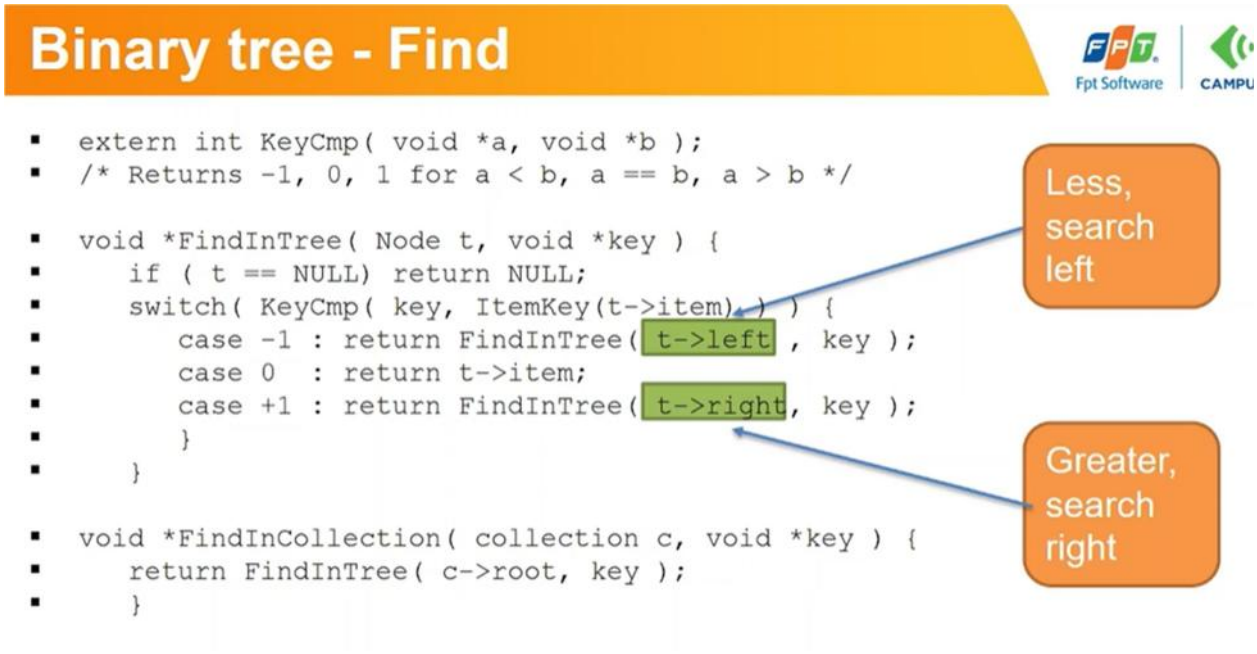
Struct t_collection {

Node root;

Ngoài ra có thể thêm các thông tin khác như số lượng phần tử, độ sâu của tree

…

}



Hàm Tìm kiếm trong 1 dãy nhị phân có thứ tự, ta xây dựng 1 hàm đệ quy find tree có tham số truyền vào là node t, và giá trị key cần tìm kiếm. trong hàm này ta kiểm tra 1 node t là NULL và trả về giá trị NULL và kết thúc.

Có nghĩa là ko tìm thấy được node của giá trị key mong muốn.

Việc node khác nULL, ta so sánh giá trị key mong muốn và giá trị key trong data của node hiện tại.

Hàm so sánh này cần được xây dựng sao cho trả về trừ 1 nếu giá trị key mong muốn nhỏ hơn giá trị trong data của node hiện tại

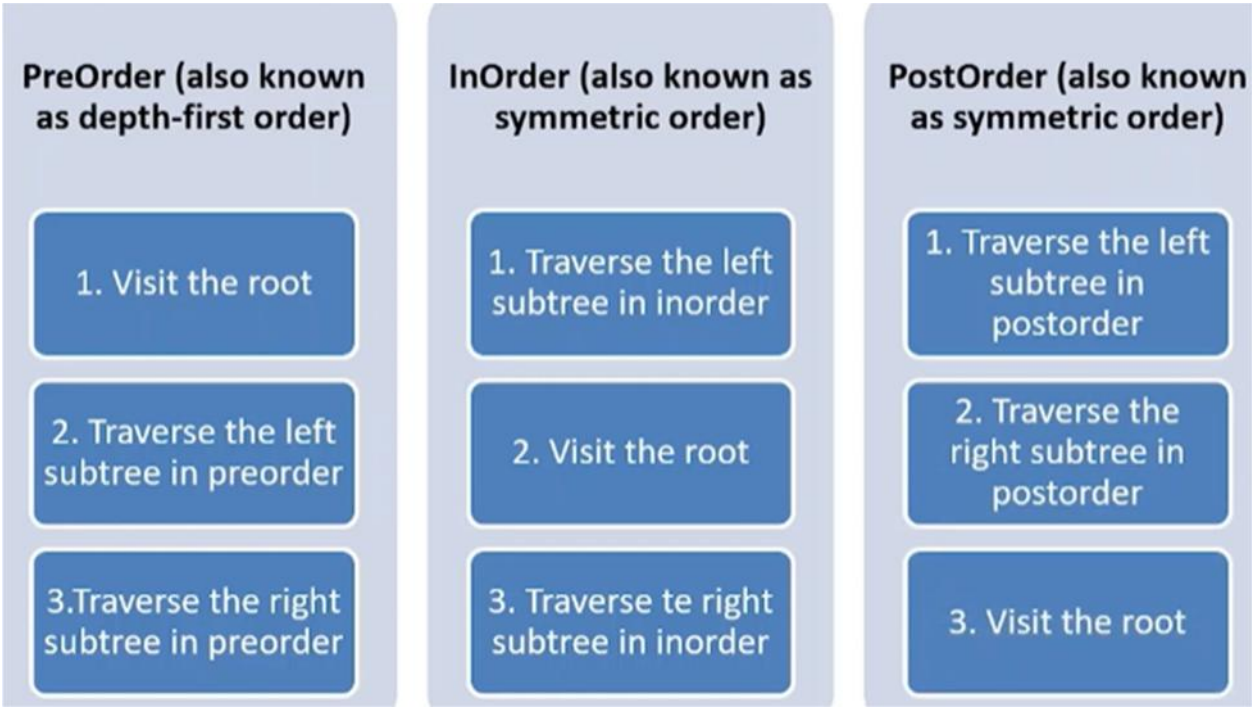
Trả về 1 nếu nó lớn hơn và trả về 0 nếu bằng nhau, nếu trả về 0 thì có nghĩa là đã tìm thấy và trả về data item của node hiện tại

Nếu trả về - 1 thì ta tiếp tục phải gọi hàm đệ quy Find In Tree với node truyền vào là node con bên trái, nếu trả về 1 thì ta truyền vào node con bên phải. trong hàm file in collection ta sẽ gọi hàm find in tree với tham số truyền vào là node root của tree và bắt đầi tìm kiếm từ root. Như vậy ta sẽ dùng đệ quy để tìm từ node root sau đó đi dần đến các node con cháu của nó, sẽ trả về data item của node thoả mãn giá trị key hoặc trả về NULL khi mà ta kiểm tra được node lá mà vẫn chưa thoả mãn giá trị key.

Để duyệt qua tất cả các node có trong 1 tree ta cần có quy tắc theo thứ tự nhất định để tránh bỏ xót bất kỳ node nào

Ta có 3 kiểu duyệt node theo thứ tự như sau

preOrder là kiểu mà ta sẽ duyệt lần lượt theo thứ tự từ node root đầu tiên, tiếp theo là cây con bên trái, cuối cùng là cây con bên phải

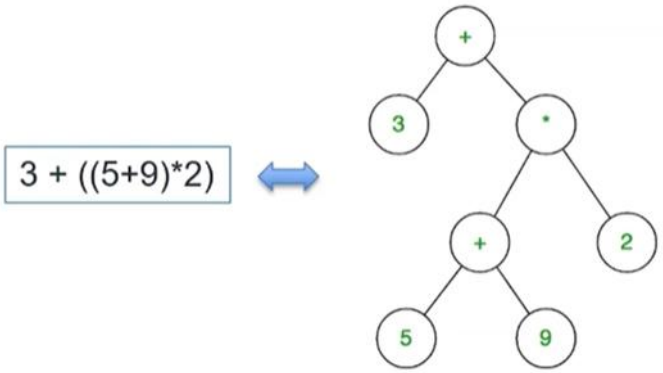


Cần chú ý rằng khi duyệt các cây con trái phải thì ta cũng cần tuân thủ theo preOrder

InOrder với Post Order, chỉ thay tên là được và tương tự như trên.

Để hiểu hơn: Traverse position

Các ứng dụng: cho các chương trình cần lấy 1 trong 2 lựa chọn tại mỗi thời điểm trong 1 quá trình, nó có thể được dùng cho các chương trình tìm kiếm số bị trùng lặp, sử dụng cho thuật toán sắp xếp. dùng để biểu diễn 1 biểu thức có các toán tử và toán hạng



Expression Tree

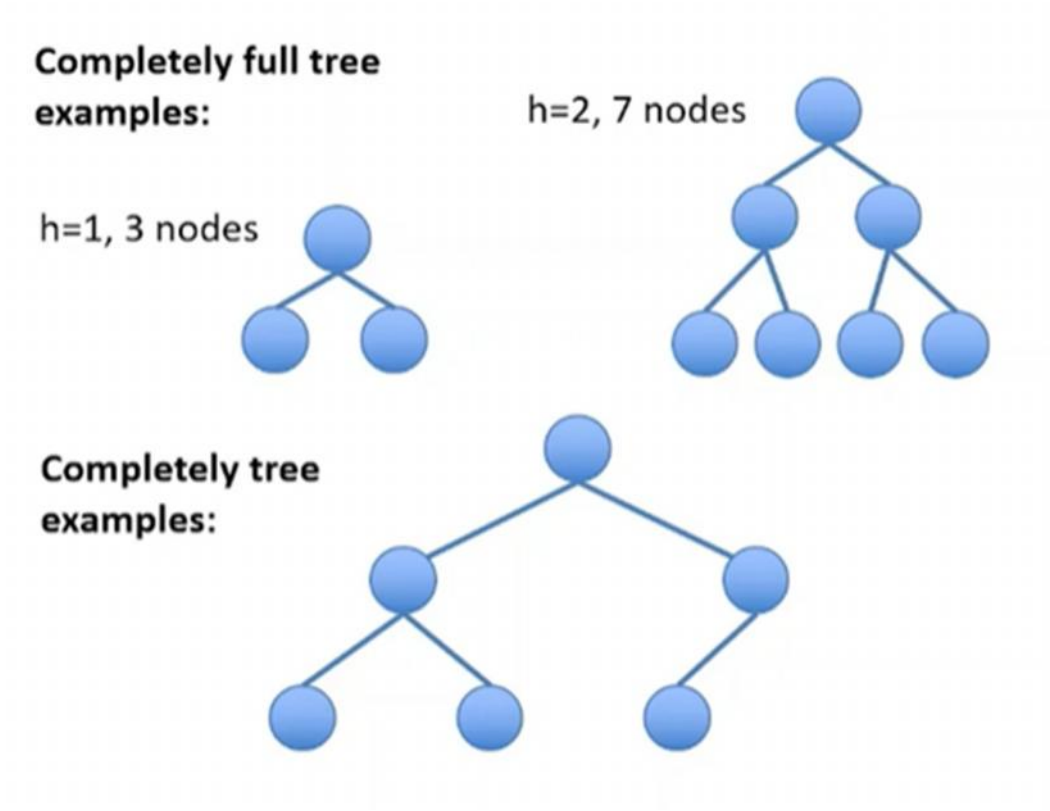
1 cây nhị phân được gọi là 1 cây hoàn chỉnh đầy đủ nó có chiều cao h và có đủ

: 2^(*h*+1)-1 nodes.

1 cây chiều cao h = 1 sẽ có số node = 3

H = 2 thì 7 nodes.

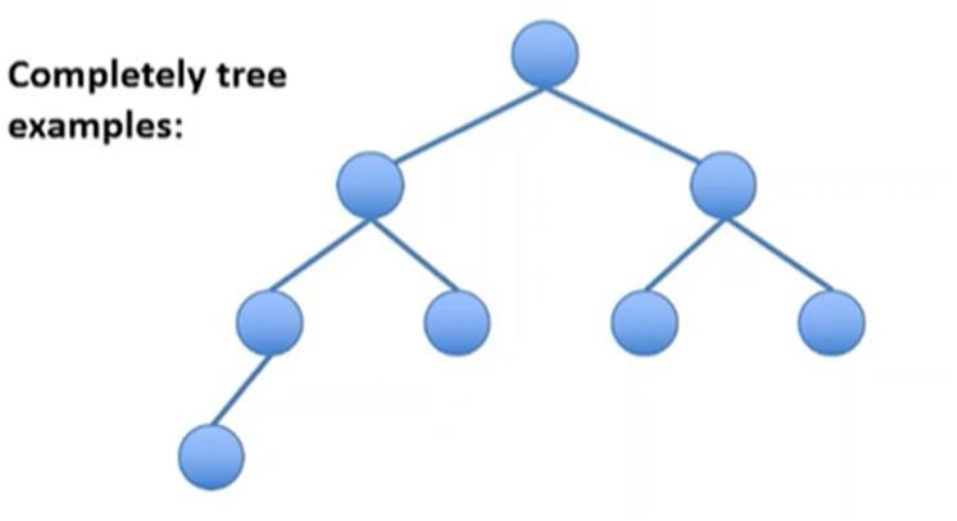
1 cây nhị phân có chiều cao h được gọi là 1 cây hoàn chỉnh nếu nó là rỗng hoặc cây con bên trái của nó là 1 cây con hoàn chỉnh đầy đủ với chiều cao h -1 và cây con bên phải là 1 cây hoàn chỉnh có chiều cao h -1



Cây con bên trái là 1 cây hoàn chỉnh, đầy đủ với chiều cao h -1 = 1 và cây con bên phải là 1 cây được coi là hoàn chỉnh với chiều cao h – 1 = 1.

Hoặc cây con bên trái là 1 cây con hoàn chỉnh với chiều cao h -1 và cây con bên phải là 1 cây con hoàn chỉnh, đầy đủ với chiều cao h – 2

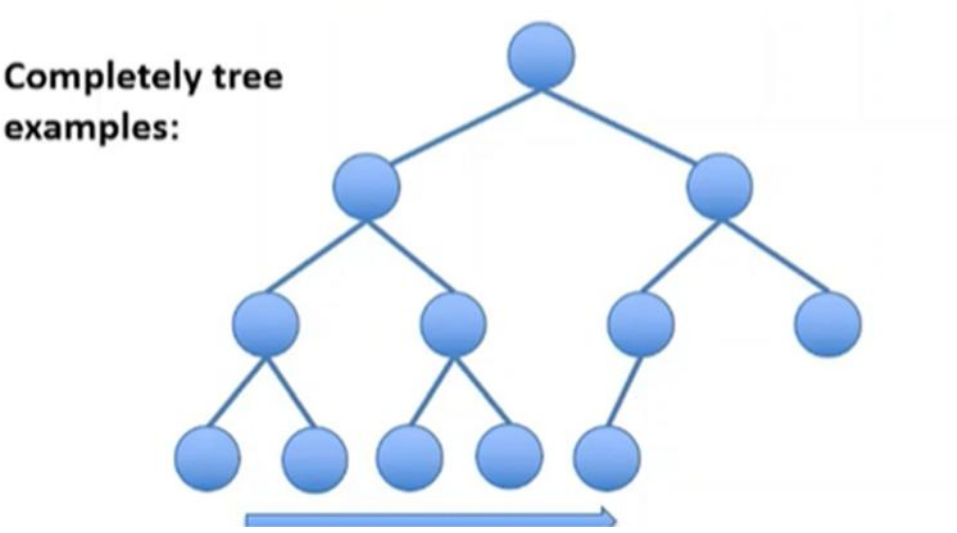
Ví dụ ta có 1 cây có chiều cao h = 3, cây con bên trái là 1 cây con hoàn chỉnh với chiều cao h – 1 = 2



Và cây con bên phải là 1 cây con hoàn chỉnh, đầy đủ với chiều cao là h – 2 = 1

1 cây hoàn chỉnh khi thêm mới phần tử sẽ được điền từ bên trái sang sao cho tất cả các node lá có cùng level hoặc tất cả các lá ở 2 level liền kề và các node lá ở level thấp nhất phải ở bên trái

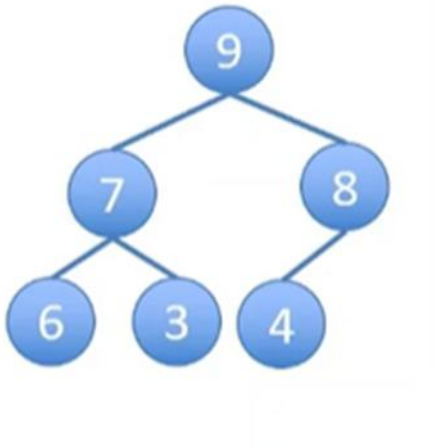
Ví dụ ta thêm các node vào cây hoàn chỉnh như hình vẽ sau:



Hay nói cách khác, cây hoàn chỉnh là cây mà khi nó thêm các node sẽ theo thứ tự điền đầy từng level và thêm từ trái sang phải

1 cây hoàn chỉnh có thuộc tính heap nếu nó là rỗng hoặc key tại root lớn hơn key tại 2 node con của nó và 2 cây con này cũng có thuộc tính heap.

Ví dụ: 1 cây hoàn chỉnh có key là các số nguyên

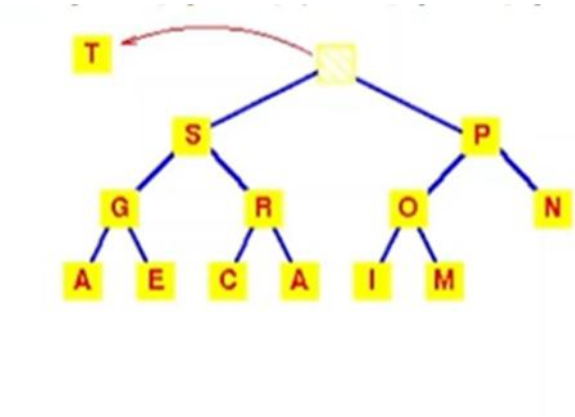


Node root lớn nhất có key là 9 lớn hơn key của 2 node con là 7 và 8; 2 node con này cũng có giá trị key lớn hơn các node con của chúng

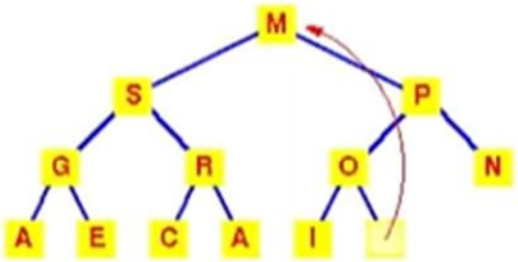
Heap có thể được sử dụng như 1 hàng đợi ưu tiên, phần tử có mức ưu tiên cao nhất là nằm ở root và sẽ được ưu tiên lấy ra đầu tiên. Nhưng nếu node root được lấy ra nó sẽ còn lại 2 cây con và chúng ta cần tạo lại 1 cây duy nhất có thuộc tính heap

Lợi ích khi sử dụng cấu trúc heap là chúng ta có thể lấy ra phần tử có độ ưu tiên cao nhất hoặc thêm vào 1 phần tử mới, đối với độ phức tạp O(logn) (and insert a new one in O(logn) time).

Ví dụ 1 heap có key là các chữ cái mà các chữ cái mà các chữ cái càng nằm ở cuối trong bảng chữ cái thì sẽ càng lớn và ta sẽ thực hiện xoá node T ở root

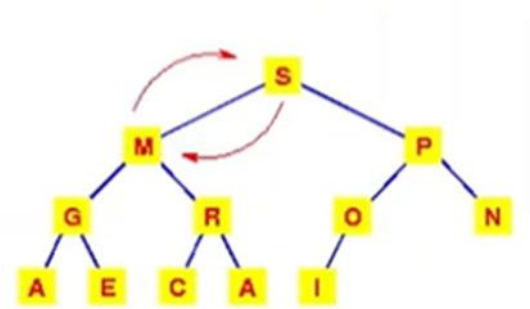


Để tìm ra cách duy trì thuộc tính heap dựa trên thực tế cây hoàn chỉnh được điền đầy từ bên trái



Node dưới cùng bên phải là node thay thế vào root đang trống, ở đây là node M

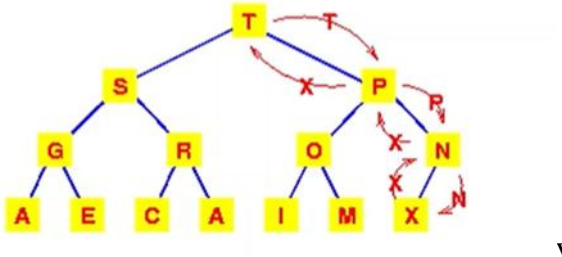
Điều này đã vi phạm điều kiện root phải lớn hơn mỗi con của nó vì M nhỏ hơn S và P, ta thấy S lớn hơn M và P nên ta đổi chỗ M và S



Nhánh con này lại tiếp tục mất thuộc tính heap vì M nhỏ hơn R nên ta tiếp tục đổi chỗ M và R. vì heap có chiều cao h nên ta cần nhiều nhất h lần thay đổi vị trí của root với các node con của nó để khôi phục hoàn toàn thuộc tính heap vì vậy nó có độ phức tạp O(h) hay còn là O(logn), n là số phần tử trong heap.

Để thêm 1 node vào heap, ta thực hiện quá trình ngược lại cách xoá lúc trước

Đầu tiên ta đặt nó vào vị trí trống thấp nhất từ bên trái (and move it up), sau đó ta thực hiện so sánh nó



với node cha của nó và nếu cần thiết ta thực hiện đổi chỗ. Cứ như vậy nó có thể được chuyển dần lên trên cho đến khi đảm bảo thuộc tính heap và nó cũng có độ phức tạp O(h) hay còn là O(logn).

Với ví dụ trên ta thấy phần tử heap được thêm vào sẽ được so sánh và đổi chỗ lần lượt với N và P

Queue là 1 cách xếp động được chia làm 4 loại theo cách sắp xếp thứ tự như

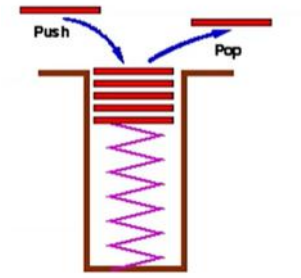
Hàng đợi FIFO là 1 hàng đợi mà 1 phần tử được thêm vào đầu tiên sẽ được lấy ra đầu tiên

Hàng đợi LIFO là hàng đợi mà phần tử thêm vào cuối cùng sẽ được lấy ra đầu tiên

Hàng đợi ưu tiên là hàng đợi mà các phần tử được sắp xếp theo thứ tự: phần tử có độ ưu tiên cao nhất sẽ được lấy ra đầu tiên

Các hàng đợi có thể được triển khai bằng Linked List

Stack là 1 dạng danh sách đặc biệt triển khai theo kiểu LIFO



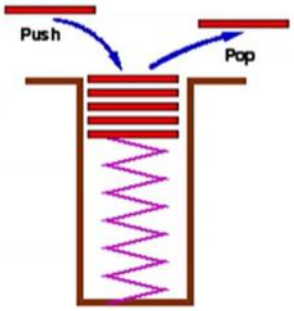
Có 2 phương thức chính là push, thêm 1 phần tử vào đầu danh sách, pop: lấy ra 1 phần tử được thêm vào gần nhất và xoá nó ra khỏi stack

Stack giống như 1 máy xếp đĩa, đĩa được cho vào cuối cùng sẽ được lấy ra đầu tiên

1 số phương thức khác như IsEmpty để kiểm tra stack rỗng hay ko

Top để lấy ra phần tử đầu tiên trong stack mà ko xoá nó khỏi stack

- **Stacks are a special form of collection with LIFO semantics**
- Two methods
  - ✓ int push(Stack s, void *item);
    - Add item to the top of the stack
  - ✓ void *pop(Stack s);
    - Remove most recently pushed item from the top of the stack
- Like a plate stacker
- Other methods
  - ✓ int IsEmpty(Stack s);
    - Determines whether the stack has anything in it
  - ✓ void *Top(Stack s);
    - Return the item at the top without deleting it



Stack có thể được triển khai bằng Array hoặc Linked List

Stack rất hữu dụng cho đệ quy

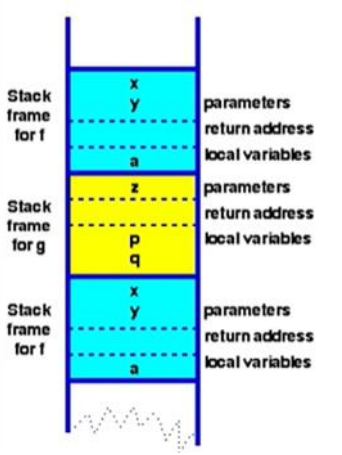
Bộ nhớ stack rất quan trọng trong việc gọi và trả về trong các hàm và thủ tục

```
function f( int x, int y)
{
    int a;
    if ( term_condition ) return …;
    a = ….;
    return g( a );
}
function g( int z )
{
    int p, q;
    p = …. ; q = …. ;
    return f(p,q);
}
```

Ví dụ 1 hàm f có tham số truyền vào là x và y và biến cục bộ a

Hàm g có tham số truyền vào là z, biến cục bộ p, q trong hàm g này nó sẽ gọi đến hàm f và check điều kiện nếu ko thoả mãn nó lại tiếp tục gọi lại hàm g và xảy ra đệ quy giữa hàm g và hàm f.

Mỗi lần trước khi gọi hàm kia, các tham số truyền vào biến cục bộ và địa chỉ trả về của hàm này sẽ được lưu vào stack và 2 hàm sẽ gọi đi gọi lại lẫn nhau



cho đến khi điều kiện trả về trong hàm f thoả mãn.

Mỗi lần trước khi quay lại đã gọi nó, các biến cục bộ, **tham số (?, lưu ở trong thanh ghi R0123 cơ mà)**, địa chỉ trả về của hàm đã gọi nó sẽ được lấy ra khỏi stack

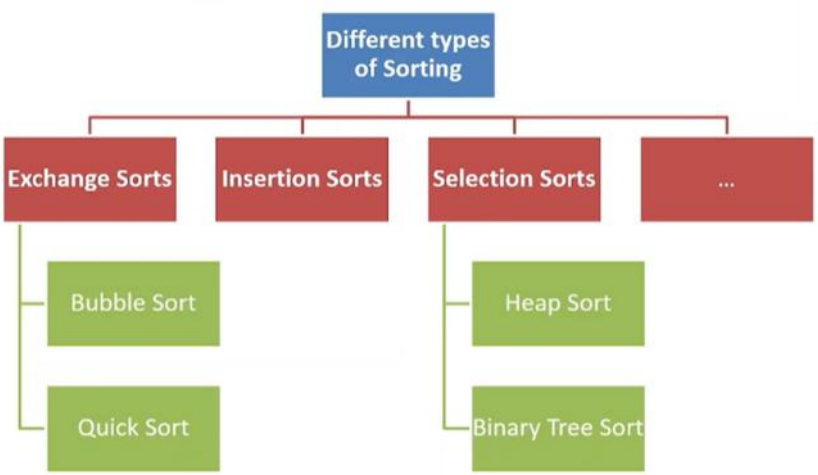
Vì vậy khi sử dụng thuật toán đệ quy, ta cần đảm bảo điều kiện trả về điều kiện hợp lý. Nếu đệ quy quá nhiều lần có thể bộ nhớ stack sẽ bị tràn và xảy ra lỗi chương trình

1 số các thuật toán cơ bản trong lập trình:

1 số thuật toán sắp xếp

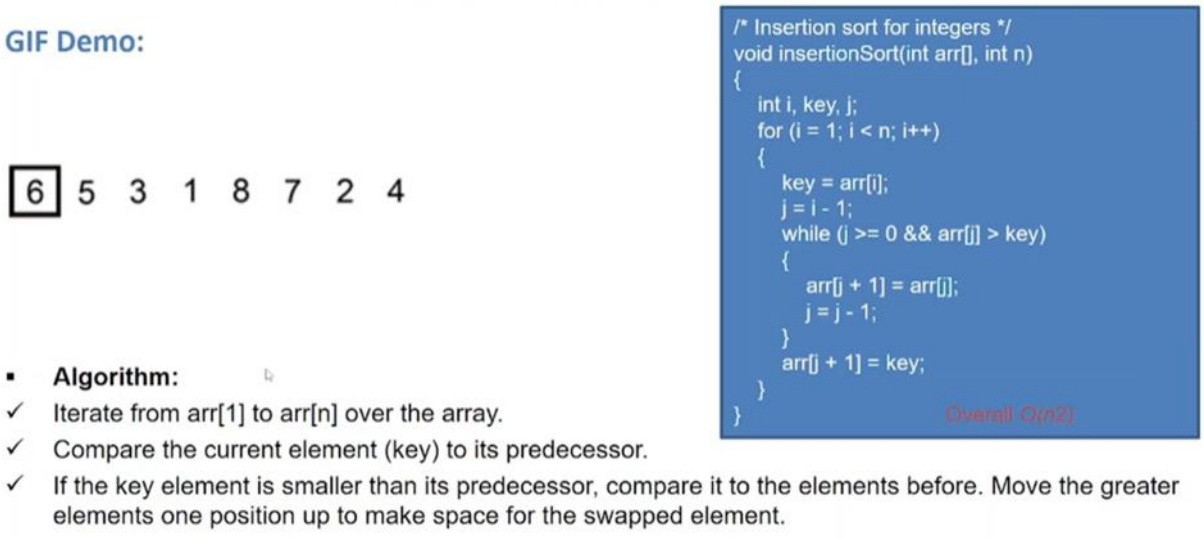
Thuật toán sắp xếp có rất nhiều loại và có 1 số loại cơ bản sau

Sắp xếp theo kiểu đổi chỗ gồm có sắp xếp nổi bọt, sắp xếp nhanh, sắp xếp theo kiểu chèn, sắp xếp theo kiểu lựa chọn, sắp xếp bằng heap hoặc cây nhị phân



sắp xếp 1 mảng n số nguyên theo thứ tự tăng dần từ đầu đến cuối mảng

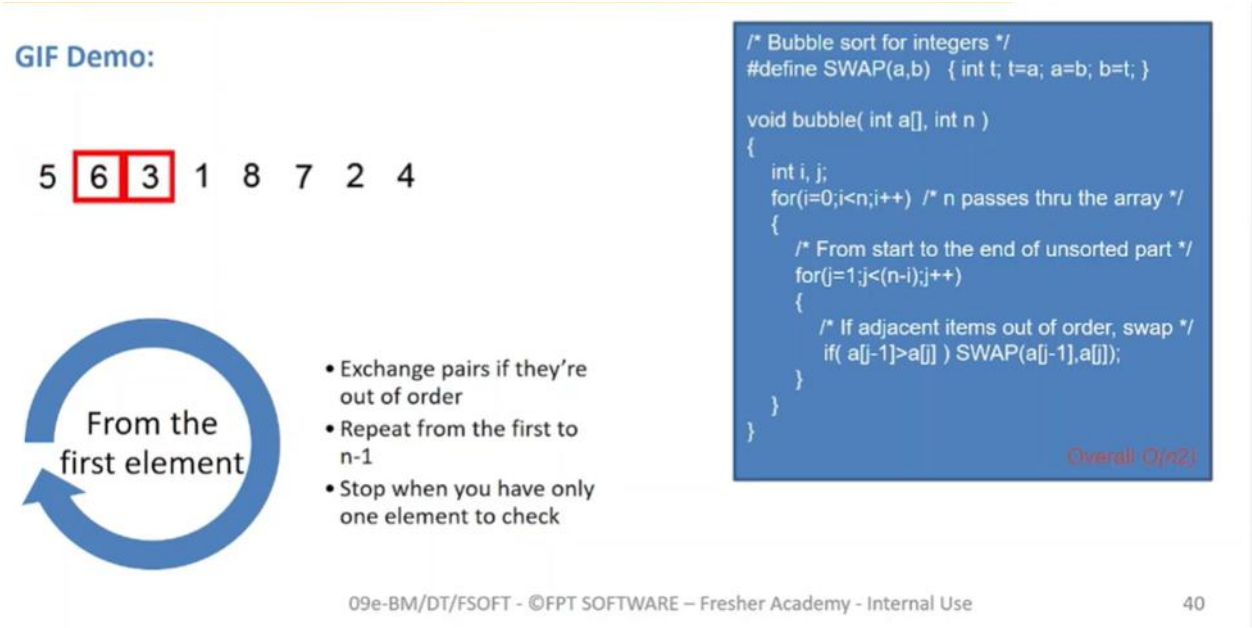
sắp xếp tràn



Từ phần tử thứ nhất đến phần tử cuối cùng thứ n của mảng, so sánh phần tử hiện tại với phần tử liền trước của nó, nếu phần tử hiện tại nhỏ hơn phần tử liền trước, tiếp tục so sánh với phần tử liền trước nữa và chuyển các phần tử lớn hơn lên 1 đơn vị tạo 1 chỗ trống để chèn phần tử hiện tại vào đó. Cứ như vậy, các phần tử càng nhỏ thì càng được chèn vào đầu mảng, các phần tử lớn hơn càng bị dịch chuyển về cuối mảng

Sắp xếp nổi bọt

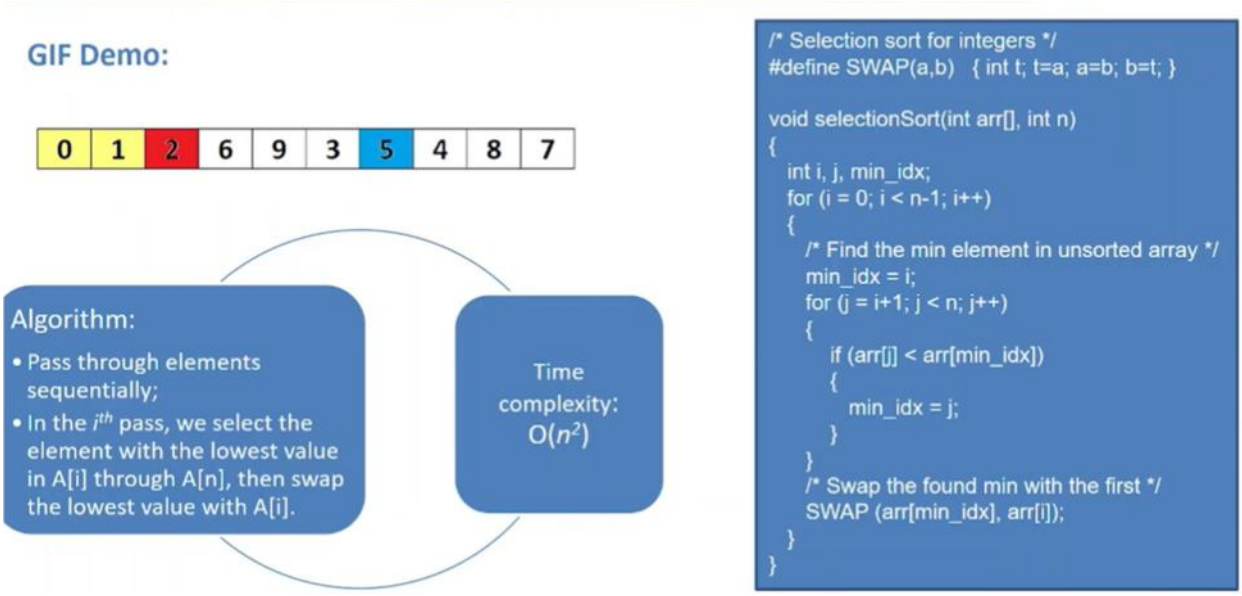
Thuật toán như sau



Ta bắt đầu duyệt từ phần tử 0, so sánh phần tử 0 và 1 với nhau, nếu nó ko đúng thứ tự, ta đổi chỗ 2 phần tử này với nhau, tiếp tục như vậy với phần tử 1 và 2, sau đó tiếp tục cho đến khi phần tử cuối cùng thứ n trong mảng được so sánh. Sau quá trình này phần tử lớn nhất đã được chuyển xuống cuối mảng, ta quay trở lại duyệt từ phần tử 0 cho tới n – 1, sau mỗi lần duyệt ta lại swap thêm được 1 phần tử vào đúng vị trí.

Cứ như vậy cho đến khi chỉ còn 1 phần chỉ cần được kiểm tra thì dừng lại

Sắp xếp lựa chọn



Ta sẽ duyệt từ phần tử 0, ta so sánh từ phần tử 1 đến N với phần tử 0 để tìm ra giá trị nhỏ nhất, sau đó đổi chỗ giá trị nhỏ nhất với phần tử 0. Tiếp tục quay lại duyệt từ phần tử 1 để tìm ra giá trị nhỏ thứ 2. Sau mỗi lần duyệt ta lại xếp thêm được 1 phần tử vào đúng vị trí.

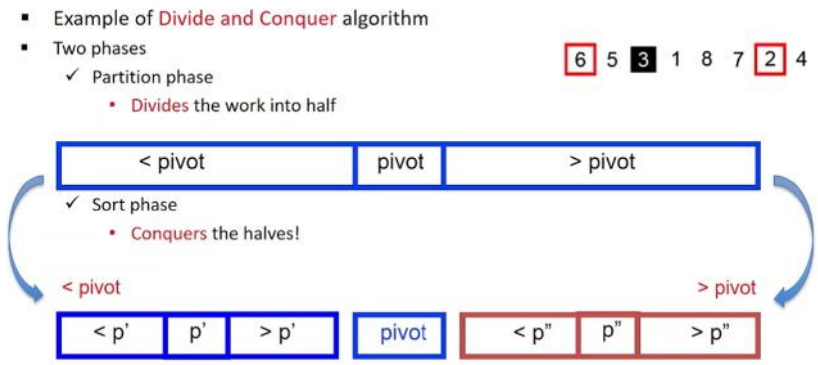
Cứ như vậy duyệt đến phần tử thứ n – 1 thì dừng lại.

Sắp xếp nhanh, hay còn lại là sắp xếp khoanh vùng (chia để trị)

Chọn 1 phần tử trong mảng để làm giá trị pivot, chia các phần tử trong mảng làm 2 mảng, 1 mảng có các giá trị nhỏ hơn giá trị pivot và 1 mảng có giá trị lớn hơn giá trị pivot

Với mỗi mảng trên, ta lại sắp xếp với phương pháp tương tự: tạo 1 giá trị pivot mới và chia thành các mảng mới, cứ như vậy, ta tạo thành 1 thuật toán đệ quy và nó kết thúc khi chia nhỏ ra chỉ còn 1 phần tử hoặc rỗng. mỗi mảng con được sắp xếp thì khi lặp lại ta sẽ thu được 1 mảng cuối cùng đã được sắp xếp

Và thuật toán này có độ phức tạp là O(nlogn) hoặc O(n^2)



Heap sort, như đã tìm hiểu ở binary tree, heap cũng có tính chất của sự sắp xếp nên ta có thể dùng heap để sắp xếp như sau:

tạo 1 heap, thêm các phần tử của mảng vào heap và đồng thời duy trì thuộc tính heap. Khi tất cả các phần tử đã được thêm vào heap, ta sẽ lấy lần lượt từng phần tử ra từ root và đồng thời duy trì thuộc tính heap. Như vậy các phần tử lấy ra đã được sắp xếp theo thứ tự

việc thêm vào và lấy ra khỏi heap tốn O(logn) times và chúng ta cần phải đệ quy n lần nên thuật toán có độ phức tạp là O(nlogn).

Phần 4: thuật toán tìm kiếm, tìm kiếm là 1 hoạt động cơ bản để tìm ra 1 phần tử trong 1 tập hợp các phần tử. mỗi phần tử có 1 giá trị key

Tìm kiếm chính là tìm ra 1 phần tử có giá trị key mong muốn. nếu nhiều phần tử có giá trị key thì ta nên xử lý như thế nào, ta có thể lấy ra phần tử đầu tiên, cuối cùng, bất kỳ hoặc lấy ra toàn bộ các phần tử có giá trị khi đó

Những điểm cần cân nhắc khi triển khai thuật toán sắp xếp như là thời gian thực hiện trung bình, thời gian thực hiện trong trường hợp xấu nhất và thời gian thực hiện tốt nhất có thể.

Tìm kiếm tuần tự, có thể ứng dụng được cho cả danh sách đã sắp xếp hoặc không, ta có 1 ví dụ về 1 mảng A[] có các phần tử lưu trữ từ 1 cho đến N

Chương trình sẽ trả về phần tử đầu tiên có giá trị I là k và trả về NULL nếu ko có phần tử nào có giá trị là k.

Thuật toán tìm kiếm tuần tự như sau.

Các phần tử trong mảng sẽ được so sánh với giá trị key theo thứ tự xuất hiện trong mảng cho đến khi gặp được 1 phần tử trong mảng có giá trị key để trả về dữ liệu của phần tử đó

Sequential Search

```
data find (item A[], int N, int k)
{
  A[0].data = NULL;
  int i = N;
  while( (A[i].key != k) & (i > 0) )
  {
     i--;
  }
  return A[i].data;
}
```

Ta gán data tại vị trí phần tử 0 là NULL, khai báo 1 index I = N

Dùng 1 vòng while cho I chạy từ n về 0 và kiểm tra key của phần tử tại ví trí key có bằng giá trị k hay không, nếu bằng, ta return data của phần tử tại ví trí I, nếu ko bằng ta kiểm tra cho đến khi I = 0 thì ta trả về NULL

Ví dụ về tìm kiếm tuần tự với 1 phần tử chặn

Sequential Search
with a sentinel

```
data find (item A[], int N, int k)
{
  A[0].key = k; /*sentinel*/
  A[0].data = NULL;
  int i = N;
  while( A[i].key !=  k )
  {
     i--;
  }
 return A[i].data;
}
```

ta gán giá trị I của phần tử 0 bằng 1 giá trị k, data của phần tử 0 là NULL, dùng 1 vòng while cho I chạy từ n giảm dần đến khi gặp 1 phần tử có giá trị I cần tìm. Ta chắc chắn được là sẽ thấy 1 Ai có giá trị I = k vì có phần từ chặn là A0 đã được gán giá trị I = k, nếu I khác 0 thì sẽ có data trả về, còn I = 0 thì giá trị Null sẽ được trả về.

Vậy điều gì cải thiện ở đây khi ta sử dụng điều kiện chặn

Như ta đã thấy ở vòng lặp while, thông thường sẽ có 3 thao tác cần thực hiện: so sánh giá trị I, kiểm tra giá trị I lớn hơn 0 và giảm giá trị I hoặc khi có phần tử chặn, ta đảm bảo được sẽ có ít nhất phần tử 0 thoả mãn giá trị I nên ta chỉ cần 2 thao tác thực hiện so sánh giá trị I, giảm giá trị I

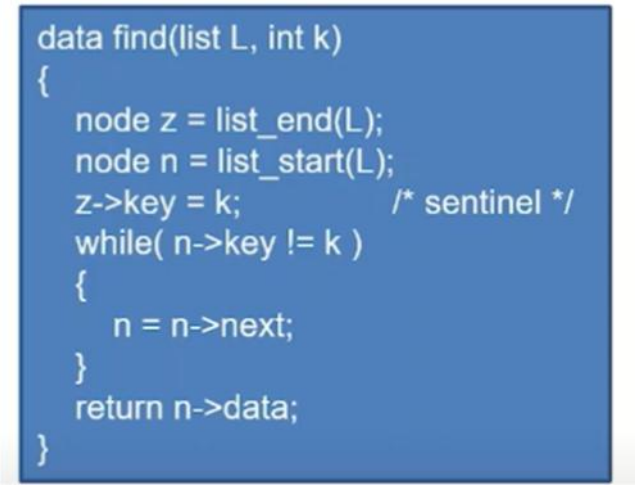
Tìm kiếm tuần tự là 1 thuật toán đơn giản thông thường, độ phức tạp về thời gian của nó tỷ lệ thuận với số phần tử n, O(n), trường hợp xấu nhất là ta cần thực hiện N+1 lần so sánh khi ko có phần tử nào cần tìm thấy và tốt nhất là thực hiện 1 lần so sánh khi phần tử cần tìm đứng ở đầu

Tính trung bình thì sẽ là (N+1)/2.

Phương pháp tìm kiếm này áp dụng cho cả array lẫn linked list.

Ví dụ về tìm kiếm tuần tự trong linked list

Nó sẽ trả về data của phần tử đầu tiên 1 giá trị I = k in list và trả về NULL khi không tìm thấy. thuật toán tìm kiếm trong linked list cũng tương tự với mảng, giá trị của từng node trong list sẽ được so sánh với giá trị k theo thứ tự từ đầu list đến cuối cho đến khi tìm thấy node có giá trị I = k để trả về



```
data find(list L, int k)
{
   node z = list_end(L);
   node n = list_start(L);
   z->key = k;              /* sentinel */
   while( n->key != k )
   {
      n = n->next;
   }
   return n->data;
}
```

ta dùng node z ở cuối list để làm phần tử chặn nên gán key của nó bằng k

Dùng node n làm phần tử đầu tiên của list và bắt đầu tìm kiếm.

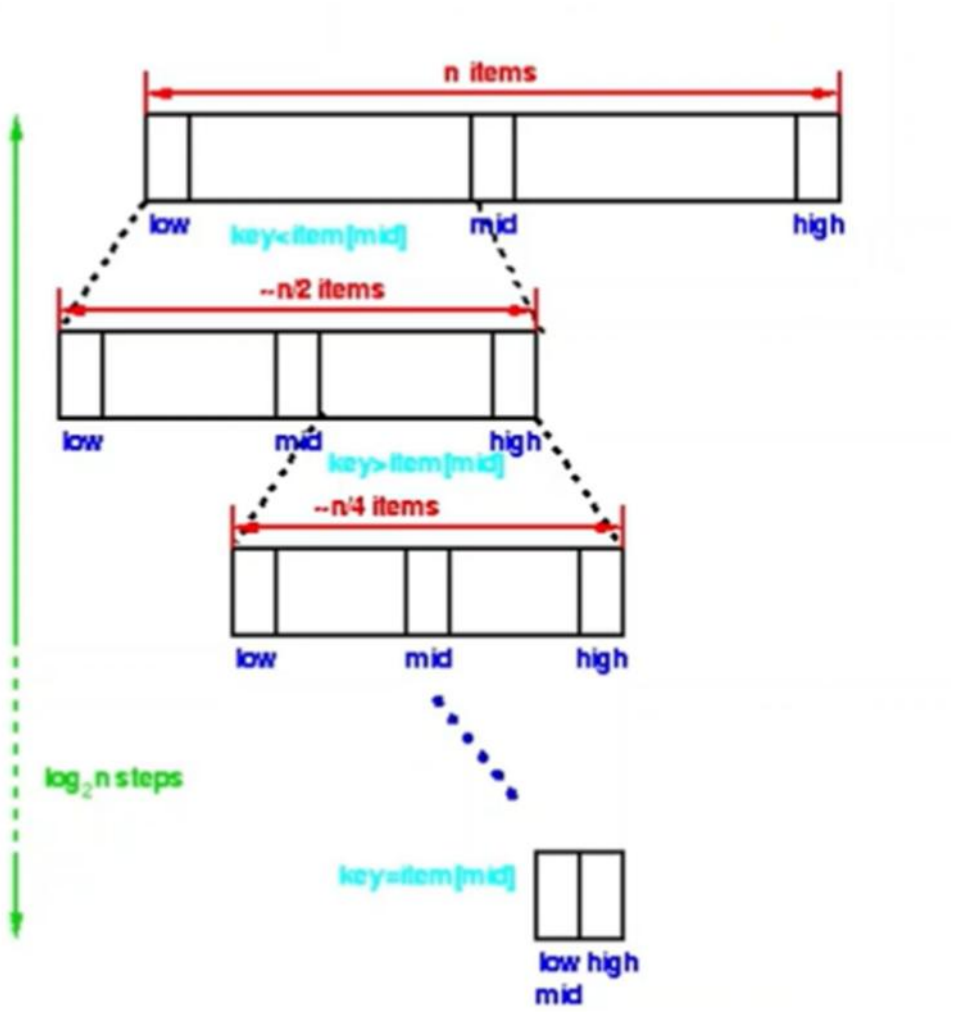
Ta dùng vòng lặp while để kiểm tra giá trị I của node n có bằng k hay ko, nếu ko bằng thì ta tiếp tục trỏ vào node tiếp theo trong danh sách cho đến khi gặp được 1 phần tử có giá trị I thì trả về data của nó hoặc trả về NULL khi đã tìm thấy node cuối cùng trong list

Các phần tử ta đánh giá là thường xuyên được tìm kiếm sẽ ở những vị trí đầu của danh sách. Ví dụ như quản lý 1 list các cuốn sách. Những cuốn sách đang nổi bật sẽ được nằm ở đầu danh sách vì nó có thể được tìm kiếm nhiều hơn.

Cách thứ hai, trong các lần tìm kiếm, phần tử được tìm thấy sẽ được di chuyển lên phía đầu danh sách, ví dụ quản lý list các bài hát. Những bài hát nào được tìm 1 lần thì người dùng sẽ tiếp tục tìm lại lần sau, với cách này thì ta nên sử dụng linked list cho hiệu quả

Cách cải thiện này rất khó để phân tích độ phức tạp trong lý thuyết nhưng nó rất hữu ích trong thực tế.

Thuật toán tìm kiếm nhị phân có thể được áp dụng cho 1 mảng đã được sắp xếp theo key, đầu tiên ta kiểm tra giá trị key của phần tử ở giữa mảng. nếu đúng, ta trả về luôn phần tử đó, nếu key cần tìm nhỏ hơn key ở phần tử giữa, như vậy phần tử cần tìm sẽ nằm ở nửa bé hơn của mảng



Nếu key cần tìm lớn hơn key ở phần tử giữa. như vậy phần tử cần tìm đã nằm ở nửa lớn hơn của mảng. sau khi xác định được key nằm ở nửa nào của mảng, ta tiếp tục coi mảng đó là mảng mới và quay trở lại bước đầu tiên kiểm tra cho đến khi nào tìm thấy phần tử có giá trị i cần tìm hoặc đến khi ko thể chia nửa ra được nữa thì kết thúc hàm và ko thể tìm thấy phần tử mong muốn.

Tìm kiếm 1 phần tử trong danh sách c, c là 1 danh sách đã được sắp xếp tăng dần theo key và key khác NULL

Chương trình sẽ trả về giá trị y cần tìm hoặc trả về null nếu ko tìm thấy

Ta cần xây dựng 1 hàm đệ quy insert

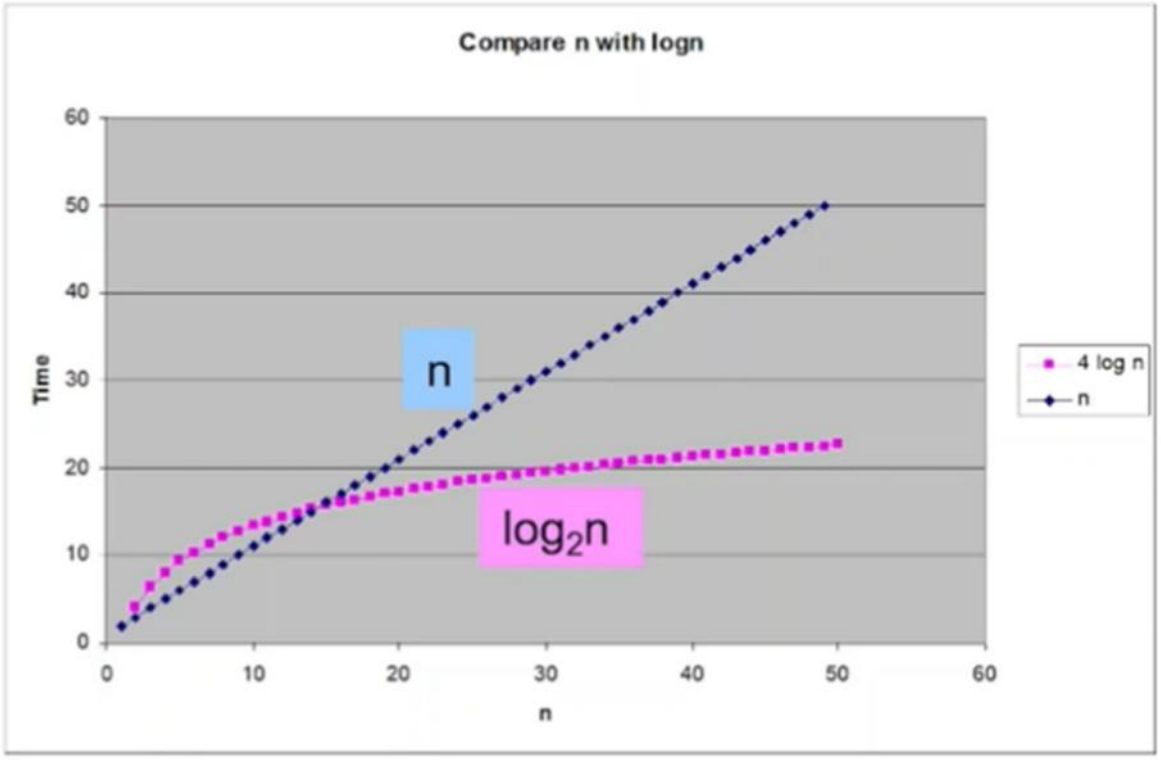
```
static void *bin_search( collection c, int low, int high, void *key ) {
  int mid;
  if ( low > high ) return NULL; /* Termination check */
  mid = ( high + low )/2;
  switch ( memcmp( ItemKey( c->items[mid] ), key, c->size) )
 {
    case 0:  return c->items[mid];              /* Match, return item found */
    case -1: return bin_search( c, low, mid-1, key);   /* search lower half */
    case 1:  return bin_search( c, mid+1, high, key); /* search upper half */
    default : return NULL;
  }
}
void *FindInCollection( collection c, void *key ) {
  int low, high;
  low = 0; high = c->item_cnt-1;
  return bin_search( c, low, high, key );
}
```

Với tham số truyền vào là danh sách C, low high là vị trí đầu và cuối của phần tử mảng cần kiểm tra và giá trị I cần tìm

Trong hàm này ta sẽ so sánh nếu LOW lớn hơn HIGH có nghĩa là đã hết phần tử để kiểm tra và ko tìm thấy và trả về NULL, nếu low vẫn nhỏ hơn high, ta tìm kiếm từ điểm giữa, mid là trung bình của low và high, so sánh giá trị I và phần tử ở vị trí mid này, bằng 0 là có nghĩa là đã tìm thấy, ta trả về phần tử tại ví trị mid này. Nếu là -1, key cần tìm nhỏ hơn, ta gọi lại chính hàm insert để kiểm tra nửa mảng nhỏ hơn, nếu là 1, key cần tìm lớn hơn, ta đi vào kiểm tra ở nửa mảng lớn hơn. Như vậy có 2 điểm kiểm tra để kết thúc quá trình đệ quy, thứ nhất là khi low lớn hơn high nghĩa là ko tìm thấy, thứ 2 là khi so sánh trả về 0 nghĩa là đã tìm thấy thì ta return về phần tử đó

Trong hàm file collection, ta bắt đầu kiểm tra cách gọi hàm insert, với low và high ở vị trí đầu tiên và cuối cùng trong mảng

So sánh tìm kiếm nhị phân và tìm kiến tuần tự



Với tìm kiếm tuần tự, trường hợp xấu nhất là thực hiện n lần so sánh với danh sách có n phần tử và tăng tuyến tính so với chiều tăng của số lượng phần tử, hay nói cách khác n là đường thẳng như trên sơ đồ. Với tìm kiếm nhị phân, trường hợp xấu nhất là thực hiện log n lần so sánh và gia tăng tỷ thuận so với chiều tăng của số lượng phần tử n, đó là 1 đường cong log n. với bài toán tìm kiếm số lượng phần tử ít thì 2 phương pháp trên không có quá nhiều khác biệt nhưng với những bài toán có số lượng phần tử càng lớn thì sự chênh lệch giữa 2 phương pháp trên càng lớn