CHƯƠNG 8

CÂY

Các CTDL được nghiên cứu trong các chương trước (danh sách, ngăn xếp, hàng đợi) là các CTDL tuyến tính (các thành phần dữ liệu được sắp xếp tuyến tính). Trong chương này chúng ta sẽ nghiên cứu một loại CTDL không tuyến tính: CTDL cây. Cây là một trong các CTDL quan trọng nhất trong khoa học máy tính. Hầu hết các hệ điều hành đều tổ chức các file dưới dạng cây. Cây được sử dụng trong thiết kế chương trình dịch, xử lý ngôn ngữ tự nhiên, đặc biệt trong các vấn đề tổ chức dữ liệu để tìm kiếm và cập nhật dữ liệu hiệu quả. Trong chương này, chúng ta sẽ trình bày các vấn đề sau đây:

- Các khái niệm cơ bản về cây và các CTDL biểu diễn cây.
- Nghiên cứu một lớp cây đặc biệt: Cây nhị phân.

- Nghiên cứu CTDL cây tìm kiếm nhị phân và cài đặt KDLTT tập động bởi cây tìm kiếm nhị phân.

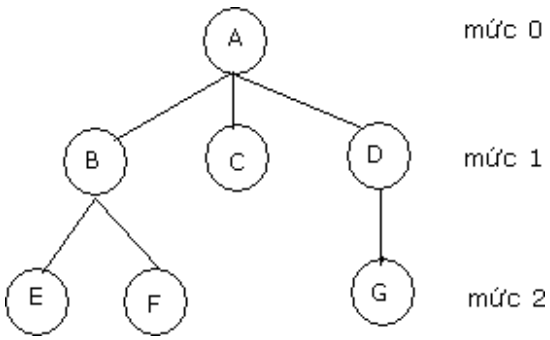
8.1 CÁC KHÁI NIỆM CƠ BẢN

Chúng ta có thể xác định khái niệm cây bằng hai cách: đệ quy và không đệ quy. Trước hết chúng ta đưa ra định nghĩa cây thông qua các khái niệm trong đồ thị định hướng. Một ví dụ điển hình về cây là tập hợp các thành viên trong một dòng họ với quan hệ cha – con. Trừ ông tổ của dòng họ này, mỗi một người trong dòng họ là con của một người cha nào đó trong dòng họ. Biểu diễn dòng họ dưới dạng đồ thị hướng: quan hệ cha – con được biểu diễn bởi các cung của đồ thị, nếu A là cha của B, thì trong đồ thị có cung đi từ đỉnh A tới đỉnh B. Xem xét các đặc điểm của đồ thị định hướng này, chúng ta đưa ra định nghĩa cây như sau:

Cây là một đồ thị định hướng thỏa mãn các tính chất sau:

- Có một đỉnh đặc biệt được gọi là **gốc** cây
- Mỗi đỉnh C bất kỳ không phải là gốc, tồn tại duy nhất một đỉnh P có cung đi từ P đến C. Đỉnh P được gọi là **cha** của đỉnh C, và C là con của P
- Có đường đi duy nhất từ gốc tới mỗi đỉnh của cây.

Người ta quy ước biểu diễn cây như trong hình 8.1: gốc ở trên cùng, hàng dưới là các đỉnh con của gốc, dưới một hàng là các đỉnh con của các đỉnh trong hàng đó, có đoạn nối từ đỉnh cha tới đỉnh con (cần lưu ý cung luôn luôn đi từ trên xuống dưới)



Hình 8.1. Biểu diễn hình học một cây

Sau đây chúng ta sẽ đưa ra một số thuật ngữ hay được dùng đến sau này.

Mở rộng của quan hệ cha – con, là quan hệ tổ tiên – con cháu. Trong cây nếu có đường đi từ đỉnh A tới đỉnh B thì A được gọi là **tổ tiên** của B, hay B là **con cháu** của A. Chẳng hạn, gốc cây là tổ tiên của các đỉnh còn lại trong cây.

Các đỉnh cùng cha được xem là **anh em**. Chẳng hạn, trong cây ở hình 8.1 các đỉnh B, C, D là anh em.

Các đỉnh không có con được gọi là **lá**. Trong hình 8.1, các đỉnh lá là E, F, C, G. Một đỉnh không phải là lá được gọi là **đỉnh trong**.

Một đỉnh bất kỳ A cùng với tất cả các con cháu của nó lập thành một cây gốc là A. Cây này được gọi là **cây con** của cây đã cho. Nếu đỉnh A là con của gốc, thì cây con gốc A được gọi là **cây con của gốc.**

Độ cao của cây là số đỉnh nằm trên đường đi dài nhất từ gốc tới một lá. Chẳng hạn, cây trong hình 8.1 có độ cao là 3. Dễ dàng thấy rằng, độ cao của cây là độ cao lớn nhất của cây con của gốc cộng thêm 1.

Độ sâu của một đỉnh là độ dài đường đi từ gốc tới đỉnh đó. Chẳng hạn, trong hình 8.1, đỉnh G có độ sâu là 2.

Cây là một CTDL **phân cấp**: Các đỉnh của cây được phân thành các mức. Mức của mỗi đỉnh được xác định đệ quy như sau:

- Gốc ở mức 1
- Mức của một đỉnh = mức của đỉnh cha + 1

Như vậy, các đỉnh trong cùng một mức là đỉnh con của một đỉnh nào đó ở mức trên. Độ cao của cây chính là mức lớn nhất của cây. Ví dụ, cây trong hình 8.1 được phân thành 3 mức: mức 1 chỉ gồm có gốc, mức 2 gồm các đỉnh A, B, C, D, mức 3 gồm các đỉnh E, F, G.

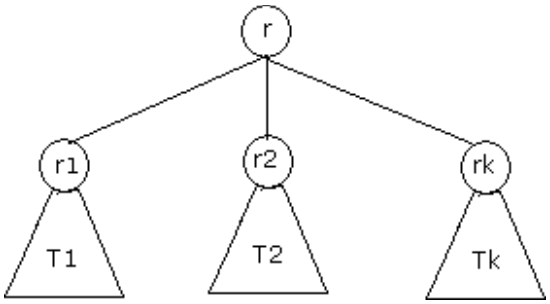
Sau này chúng ta chỉ quan tâm đến các cây được sắp. **Cây được sắp** là cây mà các đỉnh con của mỗi đỉnh được sắp sếp theo một thứ thứ tự xác định. Giả sử a là một đỉnh và các con của nó được sắp xếp theo thứ tự b_1, b_2, ..., b_k ($k \geq 1$), khi đó đỉnh b_1 được gọi là **con cả** của a, còn đỉnh b_{i+1} (i=1, 2, ...k-1) được gọi là **em liền kề** của đỉnh b_i.

Trong biểu diễn hình học, đỉnh con cả là đỉnh ngoài cùng bên trái, đỉnh b_k là đỉnh ngoài cùng bên phải.

Trên đây chúng ta đã định nghĩa cây như một đồ thị định hướng có một số tính chất đặc biệt. Khái niệm cây còn có thể định nghĩa một cách khác: định nghĩa đệ quy.

Định nghĩa đệ quy. Cây là một tập hợp không rỗng T các phần tử (được gọi là các **đỉnh**) được xác định đệ quy như sau:

- Tập chỉ có một đỉnh a là cây, cây này có gốc là a.
- Giả sử T_1, T_2, ..., T_k ($k \geq 1$) là các cây có gốc tương ứng là r_1, r_2, , r_k , trong đó hai cây bất kỳ không có đỉnh chung. Giả sử r là một đỉnh mới không có trong các cây đó. Khi đó tập T gồm đỉnh r và tất cả các đỉnh trong các cây T_i (i=1, ..., k) lập thành một cây có gốc là đỉnh r, và r là đỉnh cha của đỉnh r_i hay r_i là đỉnh con của r (i=1, ..., k). Các cây T_i (i=1, ..., k) được gọi là các cây con của gốc r. Cây T được biểu diễn hình học như sau:



Sử dụng định nghĩa cây đệ quy, chúng ta có thể dễ dàng đưa ra các thuật toán đệ quy cho các nhiệm vụ xử lý trên cây.

Sau đây chúng ta xét xem có thể biểu diễn cây bởi các CTDL như thế nào.

Cài đặt cây

Cây có thể cài đặt bởi các CTDL khác nhau. Chúng ta có thể sử dụng mảng để cài đặt cây. Song cách này không thuận tiện, ít được sử dụng. Sau đây, chúng ta trình bày hai phương pháp cài đặt cây thông dụng nhất.

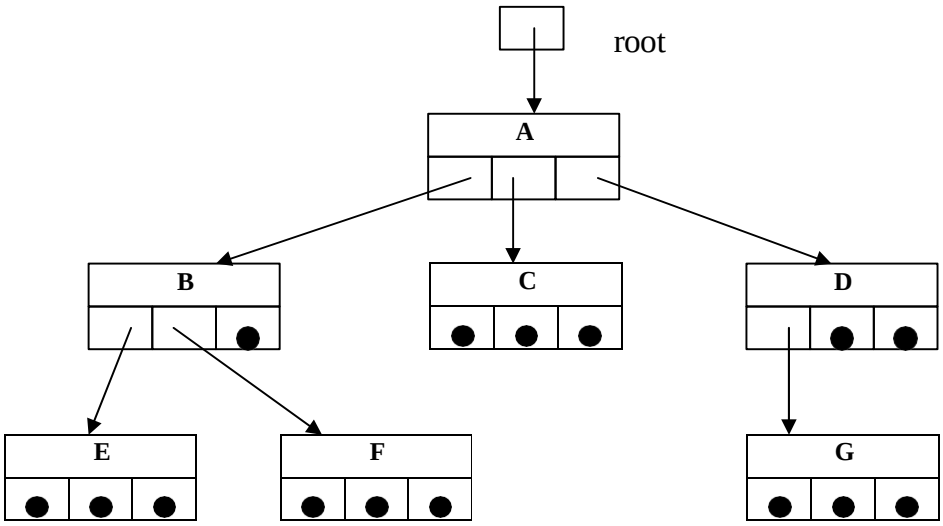
Phương pháp 1 (chỉ ra danh sách các đỉnh con của mỗi đỉnh). Với mỗi đỉnh của cây, ta sử dụng một con trỏ trỏ tới một đỉnh con của nó. Và như vậy, mỗi đỉnh của cây được biểu diễn bởi một cấu trúc gồm hai thành phần: một biến **data** lưu dữ liệu chứa trong đỉnh đó và một mảng **child** các con trỏ trỏ tới các đỉnh con. Giả sử, mỗi đỉnh chỉ có nhiều nhất K đỉnh con, khi đó ta có thể mô tả mỗi đỉnh bởi cấu trúc sau:

```
const int K = 10;
template <class Item>
{
  Item   data;
  Node*   child [K];
};
```

Chúng ta có thể truy cập tới một đỉnh bất kỳ trong cây bằng cách đi theo các con trỏ bắt đầu từ gốc cây. Vì vậy, ta cần có một con trỏ ngoài trỏ tới gốc cây, con trỏ root:

Node <Item>* root;

với cách cài đặt này, cây trong hình 8.1 được cài đặt bởi CTDT được biểu diễn hình học trong hình 8.2.



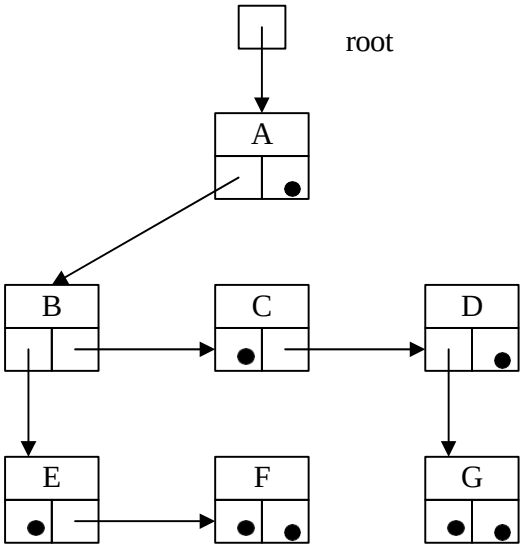
Hình 8.2. Cài đặt cây bởi mảng con trỏ.

Phương pháp 2 (chỉ ra con cả và em liền kề của mỗi đỉnh). Trong một cây, số đỉnh con của các đỉnh có thể rất khác nhau. Trong trường hợp đó, nếu sử dụng mảng con trỏ, sẽ lãng phí bộ nhớ. Thay vì sử dụng mảng con trỏ, ta chỉ sử dụng hai con trỏ: con trỏ firstChild trỏ tới đỉnh con cả và con trỏ nextSibling trỏ tới em liền kề. Mỗi đỉnh của cây được biểu diễn bởi cấu trúc sau:

```
template <class Item>
struct Node
{
```

```
    Item    data;
    Node*;  firstChild;
    Node*   nextSibling;
};
```

Chúng ta cũng cần có một con trỏ ngoài root trỏ tới gốc cây như trong phương pháp 1. Với cách này, cây trong hình 8.1 được cài đặt bởi CTDL như trong hình 8.3. Dễ dàng thấy rằng, xuất phát từ gốc đi theo con trỏ firstChild hoặc con trỏ nextSibling, ta có thể truy cập tới đỉnh bất kỳ trong cây. Ta có nhận xét rằng, các con trỏ nextSibling liên kết các đỉnh tạo thành một danh sách liên kết biểu diễn danh sách các đỉnh con của mỗi đỉnh.



Hình 8.3. Cài đặt cây sử dụng hai con trỏ.

Cần chú ý rằng, trong một số trường hợp, để thuận tiện cho các xử lý, ta có thể đưa thêm vào cấu trúc Node một con trỏ **parent** trỏ tới đỉnh cha.

8.2 DUYỆT CÂY

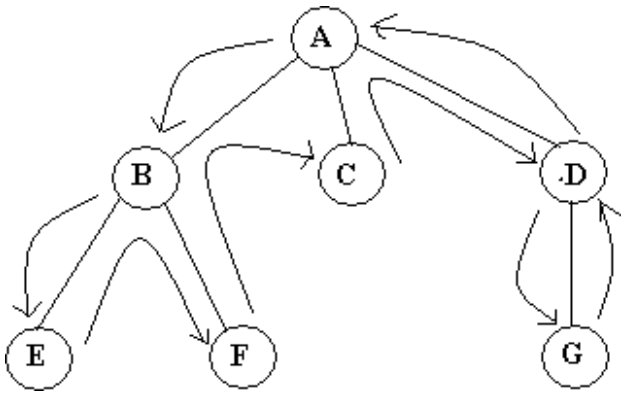
Người ta thường sử dụng cây để tổ chức dữ liệu. Khi dữ liệu được tổ chức dưới dạng cây, thì hành động hay được sử dụng là duyệt cây. **Duyệt**

cây có nghĩa là lần lượt thăm các đỉnh của cây theo một trật tự nào đó và tiến hành các xử lý cần thiết với các dữ liệu trong mỗi đỉnh của cây, chẳng hạn như in ra các dữ liệu đó. Có ba phương pháp duyệt cây hay được sử dụng nhất trong các ứng dụng là: duyệt cây theo **thứ tự trước** (preorder), theo **thứ tự trong** (inorder) và theo **thứ tự sau** (postorder). Chúng ta xác định các phương pháp duyệt cây này. Các phương pháp duyệt cây được mô tả rất đơn giản bằng đệ quy. Giả sử T là cây có gốc r và các cây con của gốc là T_1, T_2, ..., T_k (k>=0).

Duyệt cây T theo thứ tự trước có nghĩa là:

- Thăm gốc r.
- Duyệt lần lượt các cây con T_1,..., T_k theo thứ tự trước.

Chẳng hạn, xét cây trong hình 8.1. Thứ tự các đỉnh được thăm theo phương pháp này là A, B, E, F, C, D, G. Phân tích kỹ thuật duyệt cây theo thứ tự trước, ta rút ra quy luật đi thăm các đỉnh của cây như sau: đầu tiên thăm gốc r , sau đó đi xuống thăm con cả r_1 của gốc (r_1 là gốc của cây con T_1), rồi lại tiếp tục đi xuống thăm con cả của r_1... Khi không đi sâu xuống được, tức là đạt tới một đỉnh không có con (lá), ta quay lên cha của nó và đi xuống thăm một đỉnh con tiếp theo (nếu có) của đỉnh cha đó, rồi lại tiếp tục đi xuống... Quá trình đi thăm các đỉnh của cây trong hình 8.1 được biểu diễn bởi hình 8.4. Như vậy, duyệt cây theo thứ tự trước có nghĩa là xuất phát thăm từ gốc, luôn luôn đi sâu xuống thăm các đỉnh con, chỉ trừ khi nào không xuống dưới được nữa mới quay lên đỉnh cha để rồi lại tiếp tục đi xuống. Do đó, kỹ thuật duyệt cây theo thứ tự trước còn được gọi là kỹ thuật **tìm kiếm theo độ sâu**.



Hình 8.4. Thăm các đỉnh của cây theo thứ tự trước

Duyệt cây T theo thứ tự trong được quy định như sau:

- Duyệt cây con T_1 theo thứ tự trong
- Thăm gốc r
- Duyệt lần lượt các cây con T_2, ..., T_k theo thứ tự trong.

Ví dụ, thứ tự các đỉnh của cây trong hình 8.1 được thăm theo thứ tự là E, B, F, A, C, G, D.

Duyệt cây T theo thứ tự sau được tiến hành như sau:

- Duyệt lần lượt các cây con T_1, ...T_k theo thứ tự sau
- Thăm gốc r

Chẳng hạn, cũng với cây trong hình 8.1, các đỉnh được thăm theo phương pháp này lần lượt là E, F, B, C, G, D, A.

Bây giờ chúng ta cài đặt các hàm duyệt cây. Các kỹ thuật duyệt cây được xác định đệ quy, vì vậy dễ dàng cài đặt các kỹ thuật duyệt cây bởi các hàm đệ quy. Sau đây ta viết hàm đệ quy duyệt cây theo thứ tự trước: hàm Preorder. Hàm này chứa một tham biến là con trỏ root trỏ tới gốc cây. Lưu ý

rằng, C++ cho phép tham biến của một hàm có thể là hàm. Chúng ta đưa vào hàm Preorder một tham biến khác, đó là hàm với khai báo sau:

```
void   f(Item&);
```

Hàm f là hàm bất kỳ thực hiện các xử lý nào đó với dữ liệu có kiểu Item, trong đó Item là kiểu của dữ liệu chứa trong đỉnh của cây. Giả sử cây được cài đặt bằng phương pháp sử dụng hai con trỏ: firstChild và nextSibling. Khi đó hàm đệ quy Preorder được cài đặt như sau:

```
template <class Item>
void   Preorder (Node<Item>*   root, void   f(Item&))
{
       if (root ! = NULL)
       {
            f (root → data);
            node <Item>*    P = root → firstChild;
            while (P ! = NULL)
            {
                 Preorder (P, f);
                 P = P → nextSibling;
            }
}
```

Chúng ta cũng có thể cài đặt hàm Preorder không đệ quy. Muốn vậy chúng ta cần sử dụng một ngăn xếp để lưu các đỉnh nằm trên đường đi từ gốc tới đỉnh đang được thăm để khi tới một đỉnh mà không đi sâu xuống được thì biết được đỉnh cha của nó mà quay lên. Thay vì lưu các đỉnh, ngăn xếp sẽ lưu các con trỏ trỏ tới các đỉnh.

Hàm Preorder không đệ quy:

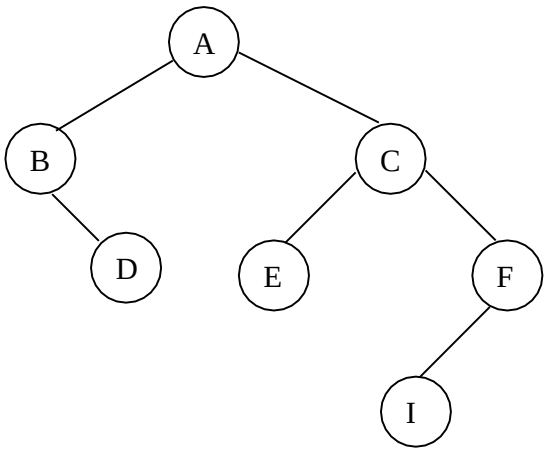
```
template <class Item>;
void  Preorder (Node <Item>*   root, void  f (Item))
{
        Stack <Node<Item>*> S; // Khởi tạo ngăn xếp rỗng S lưu
                               // các con trỏ.
        Node <Item>*  P = root
        while (P ! = NULL)
```

```
        {
                f (P → data);
                S. Push (P); // Đẩy con trỏ P vào ngăn xếp S
                P = P → firstChild;
        }
        while (! S. Empty ())
        {
                P = S. Pop (); // Loại con trỏ P ở đỉnh ngăn xếp.
                P = P → nextSibling;
                while (P ! = NULL)
                {
                        f (P → data);
                        S. Push (P);
                        P = P → firstChild;
                }
        }
}
```

Một cách tương tự, bạn đọc có thể viết ra các hàm đệ quy và không đệ quy thực hiện duyệt cây theo thứ tự trong: hàm Inorder, và duyệt cây theo thứ tự sau: hàm Postorder.

8.3 CÂY NHỊ PHÂN

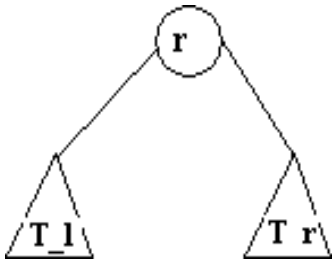
Trong mục này, chúng ta xét một lớp cây đặc biệt: **Cây nhị phân (Binary tree)**. Cây nhị phân có thể rỗng (không có đỉnh nào), nếu không rỗng thì mỗi đỉnh của cây nhị phân chỉ có nhiều nhất hai con được phân biệt là con trái và con phải. Điều đó có nghĩa rằng, trong cây nhị phân không rỗng, một đỉnh có thể không có con, có thể có đầy đủ cả con trái và con phải, có thể có con trái không có con phải hoặc có con phải nhưng không có con trái. Hình 8.5 biểu diễn một cây nhị phân: các đỉnh A, C có hai con, đỉnh B có con phải, đỉnh F có con trái.



Hình 8.5. Một cây nhị phân

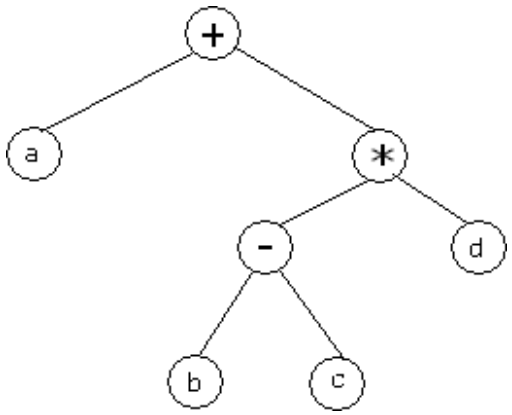
Chúng ta có thể định nghĩa cây nhị phân một cách đệ quy như sau:

- Một tập rỗng là cây nhị phân. Ta gọi là cây nhị phân rỗng
- Giả sử T_L và T_R là hai cây nhị phân không có đỉnh chung và r là một đỉnh không có trong các cây T_L và T_R. Khi đó một cây nhị phân T được tạo thành với gốc là r, có T_L là cây con trái của gốc và T_R là cây con phải của gốc. Cây nhị phân T được biểu diễn hình học như sau:



Trong định nghĩa trên, nếu T_L không rỗng thì gốc của nó được gọi là đỉnh con trái của r. Tương tự, nếu T_R không rỗng thì gốc của nó được gọi là đỉnh con phải của r.

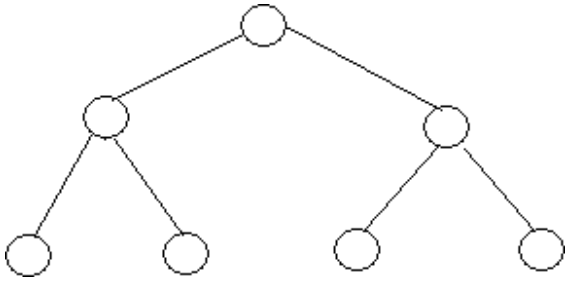
Một ví dụ về cây nhị phân là cây biến thức, nó biểu diễn các biểu thức số học. Trong cây biểu thức, các đỉnh trong biểu diễn các phép toán + , - , *, / ; cây con trái (cây con phải) của một đỉnh trong biểu diễn biểu thức con là toán hạng bên trái (toán hạng bên phải) của phép toán chứa ở đỉnh trong đó; các lá của cây biểu diễn các toán hạng có trong biểu thức. Chẳng hạn, biểu thức a + (b – c) * d được biểu diễn bởi cây nhị phân như trong hình 8.6. Chú ý rằng, nếu viết ra các đỉnh của cây biểu thức theo thứ tự sau, chúng ta nhận được biểu thức số học dạng postfix. Chẳng hạn với cây trong hình 8.6, ta có biểu thức dạng postfix: a b c – d * +.



Hình 8.6. Cây nhị phân biểu diễn biểu thức a + (b – c) * d

Sau đây chúng ta xác định một số dạng cây nhị phân đặc biệt sẽ được sử dụng đến sau này.

Cây nhị phân đầy đủ (full binary tree). Cây nhị phân rỗng (có độ cao 0), hoặc cây chỉ có đỉnh gốc (độ cao 1) được xem là cây nhị phân đầy đủ. Cây nhị phân có độ cao h >= 2 được xem là đầy đủ, nếu tất cả các đỉnh ở các mức trên (mức 1, 2, ..., h-1) đều có đầy đủ cả hai con. Hình 8.7 minh họa một cây nhị phân đầy đủ.

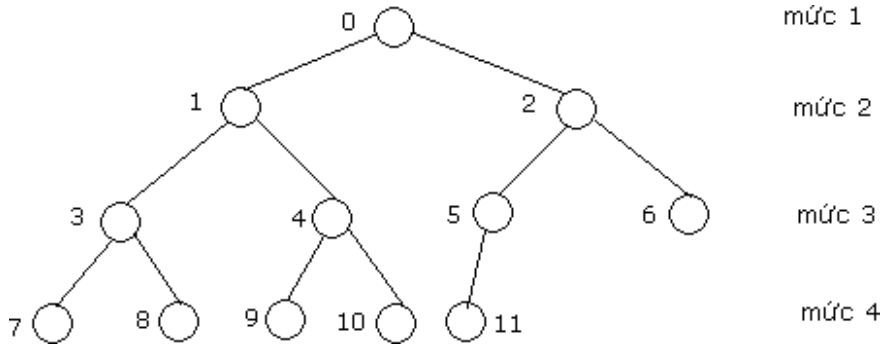


Hình 8.7. Cây nhị phân đầy đủ

Cây nhị phân hoàn toàn (complete binary tree). Các cây nhị phân có độ cao h<=1 (tức là cây rỗng hoặc chỉ có một đỉnh) là cây nhị phân hoàn toàn. Cây nhị phân có độ cao h>= 2 được xem là hoàn toàn, nếu

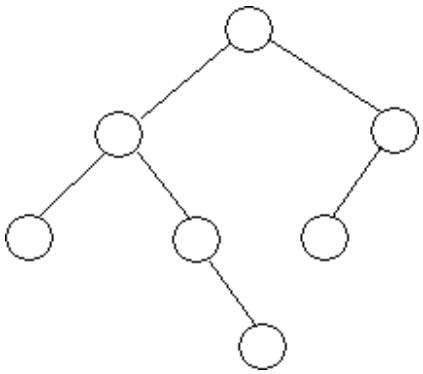
- Cây kể từ mức h – 1 trở lên là cây nhị phân đầy đủ.
- Ở mức h - 1, nếu một đỉnh chỉ có một con thì các đỉnh đứng trước nó (nếu có) có đầy đủ hai con, nếu một đỉnh chỉ có một con thì nó phải là đỉnh con trái.

Ví dụ, cây trong hình 8.8 là cây nhị phân hoàn toàn.



Hình 8.8. Cây nhị phân hoàn toàn

Cây nhị phân cân bằng (balanced binary tree). Trong một cây nhị phân, nếu độ cao của cây con trái của một đỉnh bất kỳ và độ cao của cây con phải của đỉnh đó khác nhau không quá 1 thì cây được gọi là cây nhị phân cân bằng. Hình 8.9 minh họa một cây nhị phân cân bằng.



Hình 8.9. Cây nhị phân cân bằng

Cấu trúc dữ liệu biểu diễn cây nhị phân

Do mỗi đỉnh của cây nhị phân chỉ có hai cây con: cây con trái và cây con phải, nên cách biểu diễn thông dụng nhất là sử dụng hai con trỏ: con trỏ **left** trỏ tới gốc của cây con trái, con trỏ **right** trỏ tới gốc của cây con phải, khi mà cây con trái (cây con phải) rỗng thì con trỏ left (right) có giá trị là NULL. Bằng cách này, mỗi đỉnh của cây nhị phân có cấu trúc sau:

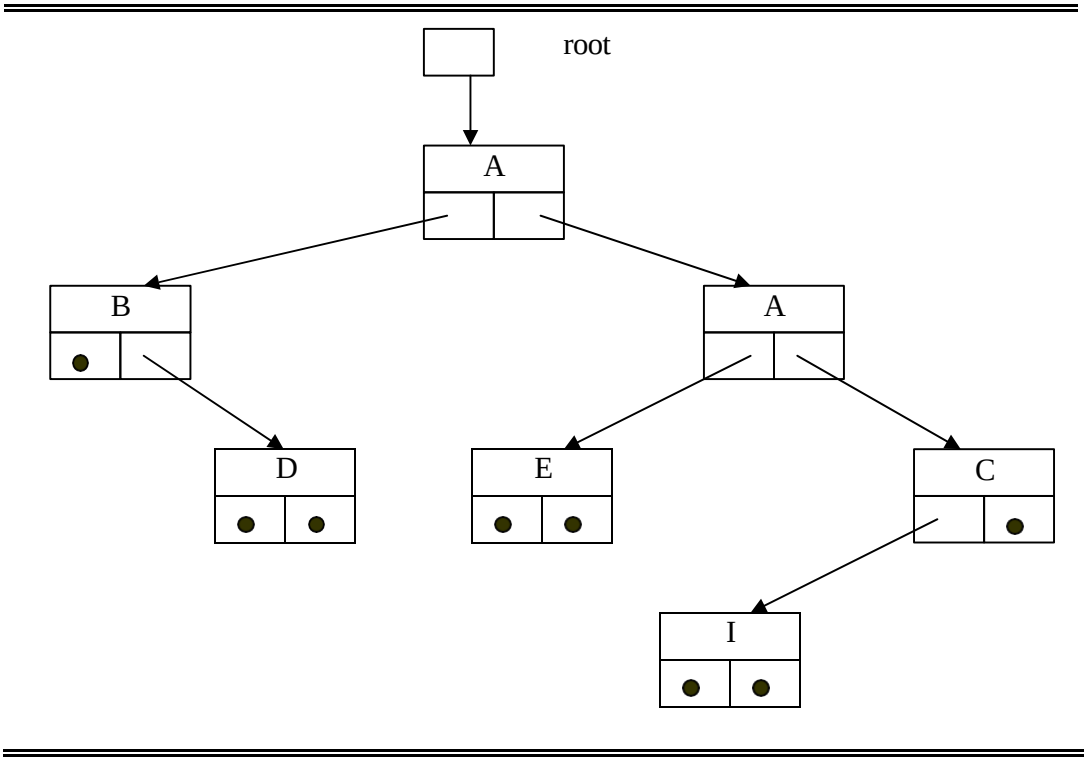
```
template <class Item>
struct     Node
{
     Item     data; // Dữ liệu chứa trong mỗi đỉnh
     Node*    left;
     Node*    right;
};
```

Mỗi cây nhị phân được biểu diễn bởi một con trỏ ngoài trỏ tới đỉnh gốc của cây: con trỏ root.

Node* root;

Khi mà cây nhị phân rỗng, thì con trỏ root có giá trị là NULL. Chẳng hạn, cây nhị phân trong hình 8.5 được biểu diễn bởi CTDL được minh họa trong hình 8.10. Có thể cài đặt cây nhị phân bởi mảng, bạn đọc hãy đưa ra cách cài đặt này (bài tập).

Biểu diễn cây nhị phân hoàn toàn bởi mảng. Từ các tính chất đặc biệt của cây nhị phân hoàn toàn, chúng ta có thể đưa ra cách cài đặt rất đơn giản và hiệu quả. Chúng ta đánh số các đỉnh cây nhị phân hoàn toàn theo thứ tự các mức từ trên xuống dưới, trong cùng một mức thì theo thứ tự từ trái qua phải, bắt đầu từ gốc được đánh số là 0, như trong hình 8.8. Với cách đánh số này, ta có nhận xét rằng, nếu một đỉnh được đánh số là i thì đỉnh con trái (nếu có) là 2*i + 1, đỉnh con phải (nếu có) là 2*i + 2, còn cha của i là đỉnh (i - 1)/2. Do đó chúng ta có thể sử dụng một mảng T để lưu các đỉnh của cây nhị phân hoàn toàn, đỉnh i (i = 0, 1, 2,...) được lưu trong thành phần T[i] của mảng.



Hình 8.10. Cài đặt cây nhị phân sử dụng con trỏ.

Duyệt cây nhị phân. Cũng như đối với cây tổng quát, người ta thường tiến hành duyệt cây nhị phân theo thứ tự trước, trong và sau. Duyệt cây nhị phân theo thứ tự trước (Preorder) có nghĩa là: Nếu cây không rỗng thì thăm gốc trước, sau đó duyệt cây con trái của gốc theo thứ tự trước, rồi duyệt cây con phải của gốc theo thứ tự trước. Chúng ta có thể dễ dàng cài đặt phương pháp duyệt cây nhị phân theo thứ tự trước bởi hàm đệ quy như sau:

```
template <class Item>
void    Preorder (Node <Item> *  root, void  f (Item &))
// Thăm các đỉnh của cây nhị phân theo thứ tự trước
// và tiến hành các xử lý (được mô tả bởi hàm f) với dữ
// liệu chứa trong mỗi đỉnh của cây.
{if (root! = NULL)
        {
         f (root → data);
         Preorder (root → left, f);
```

```
        Preorder (root → right, f);
      }
}
```

Bạn đọc hãy tự mình đưa ra định nghĩa duyệt cây nhị phân theo thứ tự trong và theo thứ tự sau, và viết các hàm đệ quy cài đặt các phương pháp duyệt cây đó (bài tập).

8.4 CÂY TÌM KIẾM NHỊ PHÂN

Một trong các ứng dụng quan trọng nhất của cây nhị phân là sử dụng cây nhị phân để tổ chức dữ liệu. Trong các chương trình, thông thường chúng ta cần phải lưu một tập các dữ liệu, rồi thường xuyên phải thực hiện cá phép toán: tìm kiếm dữ liệu, cập nhật dữ liệu... Trong các chương 4 và 5, chúng ta đã nghiên cứu sự cài đặt **KDLTT** tập động (một tập dữ liệu với các phép toán tìm kiếm, xen, loại...) bởi danh sách. Nếu tập dữ liệu được lưu trong **DSLK** thì các phép toán tìm kiếm, xen, loại, ... đòi hỏi thời gian O(n), trong đó n là số dữ liệu. Nếu tập dữ liệu được sắp xếp thành một danh sách theo thứ tự tăng (giảm) theo khóa tìm kiếm, và danh sách này được lưu trong mảng, thì phép toán tìm kiếm chỉ đòi hỏi thời gian O(logn) nếu sử dụng kỹ thuật tìm kiếm nhị phân (mục 4.4), nhưng các phép toán xen, loại vẫn cần thời gian O(n). Trong mục này, chúng ta sẽ nghiên cứu cách tổ chức một tập dữ liệu dưới dạng cây nhị phân, các dữ liệu được chứa trong các đỉnh của cây nhị phân theo một trật tự xác định, cấu trúc dữ liệu này cho phép ta cài đặt các phép toán tìm kiếm, xen, loại,... chỉ trong thời gian O(h), trong đó h là độ cao của cây nhị phân.

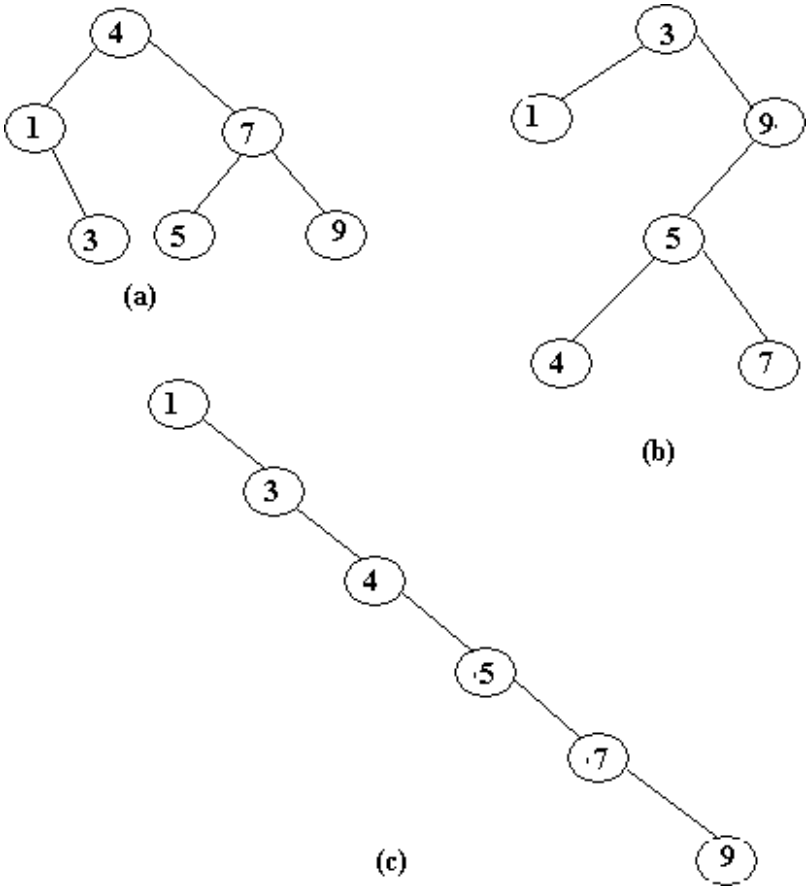
8.4.1 Cây tìm kiếm nhị phân

Giả sử chúng ta có một tập dữ liệu, các dữ liệu có kiểu Item nào đó chứa một thành phần được lấy làm khóa tìm kiếm. Chúng ta giả thiết rằng các giá trị khóa có thể sắp thứ tự tuyến tính, thông thường các giá trị khóa là các số nguyên, các số thực, các ký tự hoặc xâu ký tự. Chúng ta sẽ lưu tập dữ liệu đó trong một cây nhị phân (khóa của dữ liệu được nói tới như là khóa

của một đỉnh) theo trật tự như sau: giá trị khóa của một đỉnh bất kỳ lớn hơn các giá trị khóa của tất cả các đỉnh ở cây con trái của đỉnh đó và nhỏ hơn các giá trị khóa của tất cả các đỉnh ở cây con phải của đỉnh đó. Do đó, chúng ta có định nghĩa sau:

Cây tìm kiếm nhị phân (binary search tree) là cây nhị phân thỏa mãn tính chất sau: đối với mỗi đỉnh x trong cây, nếu y là đỉnh bất kỳ ở cây con trái của x thì khóa của x lớn hơn khóa của y, còn nếu y là đỉnh bất kỳ ở cây con phải của x thì khóa của x nhỏ hơn khóa của y.

Ví dụ. Chúng ta xét các cây nhị phân với các giá trị khoá của các đỉnh là các số nguyên. Các cây nhị phân trong hình 8.11 là các cây tìm kiếm nhị phân. Chúng ta có nhận xét rằng, các cây tìm kiếm nhị phân trong hình 8.11 biểu diễn cùng một tập hợp dữ liệu, nhưng cây trong hình 8.11a có độ cao là 3, cây trong hình 8.11b có độ cao là 4, còn cây trong hình 8.11c có tất cả các cây con trái của các đỉnh đều rỗng và nó có độ cao là 6. Một nhận xét quan trọng khác là, nếu chúng ta duyệt cây tìm kiếm nhị phân theo thứ tự trong, chúng ta sẽ nhận được một dãy dữ liệu được sắp xếp theo thứ tự khóa tăng dần. Chẳng hạn, với các cây tìm kiếm nhị phân trong hình 8.11, ta có dãy các giá trị khóa là 1, 3, 4, 5,7, 9.



Hình 11. Các cây tìm kiếm nhị phân

Chúng ta cũng có thể định nghĩa cây tìm kiếm nhị phân bởi đệ quy như sau:

- Cây nhị phân rỗng là cây tìm kiếm nhị phân
- Cây nhị phân không rỗng T là cây tìm kiếm nhị phân nếu:
    1. Khóa của gốc lớn hơn khóa của tất cả các đỉnh ở cây con trái T_L và nhỏ hơn khóa của tất cả các đỉnh ở cây con phải T_R.
    2. Cây con trái T_L và cây con phải T_R là các cây tìm kiếm nhị phân.

Sử dụng định nghĩa đệ quy này, chúng ta dễ dàng đưa ra các thuật toán đệ quy thực hiện các phép toán trên cây tìm kiếm nhị phân, như chúng ta sẽ thấy trong mục sau đây.

8.4.2 Các phép toán tập động trên cây tìm kiếm nhị phân

Bây giờ chúng ta xét xem các phép toán tập động (tìm kiếm, xen, loại, ...) sẽ được thực hiện như thế nào khi mà tập dữ liệu được cài đặt bởi cây tìm kiếm nhị phân. Chúng ta sẽ chỉ ra rằng, các phép toán tập động trên cây tìm kiếm nhị phân chỉ đòi hỏi thời gian O(h), trong đó h là độ cao của cây. Như độc giả đã thấy trong hình 8.12, một tập dữ liệu có thể lưu trong các cây tìm kiếm nhị phân có độ cao của cây có thể là n, trong đó n là số dữ liệu. Như vậy, trong trường hợp xấu nhất thời gian thực hiện các phép toán tập động trên cây tìm kiếm nhị phân là O(n). Tuy nhiên, trong mục 8.5 chúng ta sẽ chứng tỏ rằng, nếu cây tìm kiếm nhị phân được tạo thành bằng cách xen vào các dữ liệu được lấy ra từ tập dữ liệu một cách ngẫu nhiên, thì thời gian trung bình của các phép toán tập động là O(logn). Hơn nữa, bằng cách áp dụng các kỹ thuật hạn chế độ cao của cây, chúng ta có thể đảm bảo thời gian logarit cho các phép toán tập động trên cây tìm kiếm nhị phân (xem chương 11)

Dưới đây chúng ta sẽ đưa ra các thuật toán thực hiện các phép toán tập động trên cây tìm kiếm nhị phân. Chúng ta sẽ mô tả các thuật toán bởi các hàm dưới dạng giả mã . Trong các thuật toán, chúng ta sẽ sử dụng các ký hiệu sau đây: T là cây tìm kiếm nhị phân có gốc là root, dữ liệu chứa ở gốc được ký hiệu là rootData, cây con trái của gốc là T_L, cây con phải của gốc là T_R; v là một đỉnh, dữ liệu chứa trong đỉnh v được ký hiệu là data(v), đỉnh con trái của v là leftChild(v), đỉnh con phải là rightChild(v); dữ liệu trong các đỉnh có kiểu Item, khóa của dữ liệu d được ký hiệu là d.key.

Phép toán tìm kiếm. Cho cây tìm kiếm nhị phân T, để tìm xem cây T có chứa dữ liệu với khóa k cho trước hay không, chúng ta kiểm tra xem gốc có chứa dữ liệu với khóa k hay không. Nếu không, giả sử k < rootData.key, khi đó do tính chất của cây tìm kiếm nhị phân, dữ liệu với khóa k chỉ có thể

chứa trong cây con trái của gốc, và do đó, ta chỉ cần tiếp tục tìm kiếm trong cây con trái của gốc. Tương tự, nếu k > rootData.key, sự tìm kiếm được hạn chế trong phạm vi cây con phải của gốc. Từ đó, ta có thuật toán tìm kiếm đệ quy sau:

```
bool    Search (T, k)
// Tìm dữ liệu với khóa k trong cây tìm kiếm nhị phân
// Hàm trả về true (false) nếu tìm thấy (không tìm thấy)
{
if (T rỗng)
        return   false;
else if (rootData.key = = k)
        return   true;
      else if (k <rootData.key)
                        Search (T_L, k);
           else         Search (T_R, k)'
}
```

Phân tích thuật toán đệ quy trên, chúng ta dễ dàng đưa ra thuật toán tìm kiếm không đệ quy. Sử dụng biến v chạy trên các đỉnh của cây T bắt đầu từ gốc. Khi v là một đỉnh nào đó của cây T, chúng ta kiểm tra xem đỉnh v có chứa dữ liệu với khóa k hay không. Nếu không, tùy theo khóa k nhỏ hơn (lớn hơn) khóa của dữ liệu trong đỉnh v mà chúng ta đi xuống đỉnh con trái (con phải) của v. Thuật toán tìm kiếm không đệ quy là như sau:

```
bool        Search (T, k)
{
        if (T rỗng)
                return false;
        else {
                v = root;
                do {
                        if (data (v).key = = k)
                                return true;
                        else    if (k < data (v).key)
                                        if (v có con trái)
                                                v = leftchild (v);
                                        else    return false;
```

```
                else    if (v có con phải)
                                v = rightchild (v);
                        else    return false;
                }
        while (1);
}
```

Các đỉnh mà biến v chạy qua tạo thành một đường đi từ gốc hướng tới một lá của cây. Trong trường hợp xấu nhất, biến v sẽ dừng lại ở một đỉnh lá. Bởi vì độ cao của cây là độ dài của đường đi dài nhất từ gốc tới lá, do đó thời gian của phép toán Search là O(h). Chúng ta nhận thấy có sự tương tự giữa kỹ thuật tìm kiếm nhị phân (xem 4.4) và kỹ thuật tìm kiếm trên cây tìm kiếm nhị phân. Trong quá trình tìm kiếm trên cây tìm kiếm nhị phân , tại mỗi thời điểm chúng ta hạn chế tìm kiếm ở cây con trái hoặc ở cây con phải; còn trong tìm kiếm nhị phân chúng ta tiếp tục tìm kiếm ở nửa bên trái hay nửa bên phải của mảng. Tuy nhiên trong tìm kiếm nhị phân, tại mỗi thời điểm không gian tìm kiếm (mảng) được chia đôi, nửa bên trái và nửa bên phải bằng nhau; điều đó đảm bảo thời gian trong tìm kiếm nhị phân là O(logn). Nhưng trong cây tìm kiếm nhị phân, cây con trái và cây con phải có thể có số đỉnh rất khác nhau, do đó nói chung thời gian tìm kiếm trên cây tìm kiếm nhị phân không phải là O(logn), chỉ có thời gian này khi mà cây tìm kiếm nhị phân được xây dựng "cân bằng" tại mọi đỉnh.

Các phép toán tìm dữ liệu có khóa nhỏ nhất (phép toán Min) và tìm dữ liệu có khóa lớn nhất (phép toán Max)

Do tính chất của cây tìm kiếm nhị phân, nếu cây con phải không rỗng thì dữ liệu có khóa lớn nhất phải nằm ở cây con phải; tương tự, nếu cây con trái không rỗng thì dữ liệu có khóa nhỏ nhất phải nằm ở cây con trái. Từ đó, chúng ta dễ dàng đưa ra thuật toán đệ quy tìm dữ liệu có khóa lớn nhất (nhỏ nhất). Sau đây là hàm Min đệ quy:

```
Item    Min (T)
// Cây T không rỗng
// Hàm trả về dữ liệu có khóa nhỏ nhất.
```

```
{
        if (cây con trái T_L rỗng)
                        return    rootData;
         else           return    Min (T_L);
}
```

Chúng ta có nhận xét rằng, đỉnh chứa dữ liệu có khóa lớn nhất (nhỏ nhất) là đỉnh ngoài cùng bên phải (ngoài cùng bên trái). Chúng ta có thể đạt tớ đỉnh ngoài cùng bên phải (ngoài cùng bên trái) bằng cách xuất phát từ gốc và luôn luôn đi xuống nhánh bên phải (luôn luôn đi xuống nhánh bên trái). Do đó, chúng ta có thể đưa ra thuật toán không đệ quy cho phép toán Min như sau:

```
Item    Min (T)
{
  v = root;
  while (v có đỉnh con trái)
        v = leftChild (v);
  return        data (v);
}
```

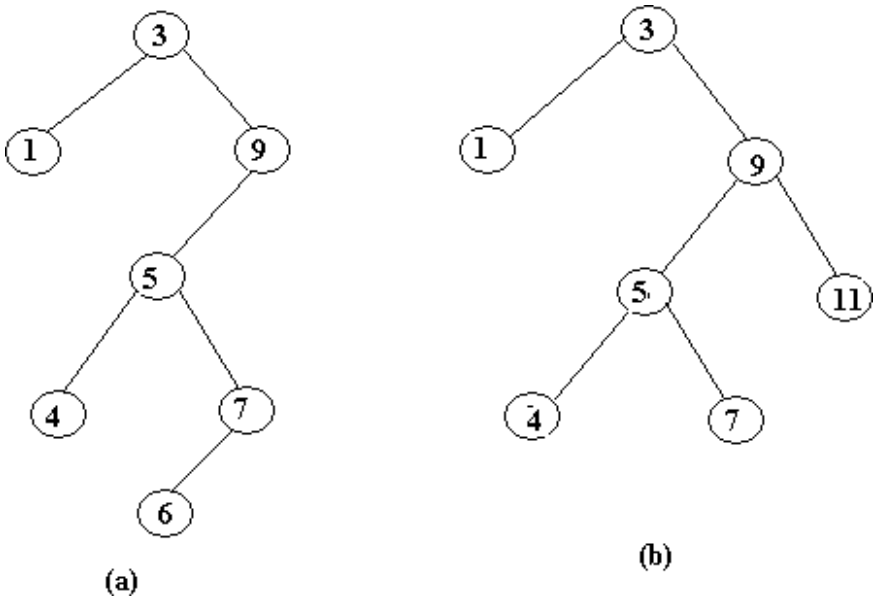
Phép toán xen (Insert). Chúng ta cần xen vào cây tìm kiếm nhị phân T một đỉnh mới chứa dữ liệu d, nhưng cần phải đảm bảo cây nhận được sau khi xen vẫn còn là cây tìm kiếm nhị phân. Nếu cây T rỗng thì ta chỉ cần tạo ra cây T chỉ có một đỉnh gốc chứa dữ liệu d. Trong trường hợp cây T không rỗng, nếu d.key < rootData.key (d.key > rootData.key) thì để đảm bảo tính chất của cây tìm kiếm nhị phân, chúng ta cần phải xen đỉnh mới chứa dữ liệu d vào cây con trái của gốc (cây con phải của gốc).

Hàm Insert đệ quy như sau:

```
void    Insert (T, d)
// xen vào cây T một đỉnh mới chứa dữ liệu d
{
   if (T rỗng)
        Tạo ra cây T chỉ có gốc chứa dữ liệu d;
   else  if (d.key < rootData.key)
```

```
                    insert (T_L, d);
            else if (d.key > rootData.key)
                    insert (T_R, d);
    }
```

Thuật toán Insert không đệ quy cũng tương tự như thuật toán Search không đệ quy. Nếu cây T không rỗng, ta xuất phát từ gốc và khi ở một đỉnh thì ta đi tiếp xuống đỉnh con trái, hoặc đỉnh con phải hoàn toàn giống như khi tìm kiếm, và dừng lại tại đỉnh v khi mà d.key < data (v).key nhưng đỉnh v không có con trái, hoặc khi mà d.key > data (v).key nhưng v không có con phải. Sau đó, ta gắn đỉnh mới chứa dữ liệu d làm đỉnh con trái của đỉnh v khi mà d.key < data (v).key, hoặc gắn đỉnh mới làm đỉnh con phải của v trong trường hợp kia. Chẳng hạn, nếu chúng ta xen vào cây trong hình 8.11b đỉnh mới chứa dữ liệu với khóa 6, ta nhận được cây trong hình 8.12a, còn nếu xen vào đỉnh có khóa 11, ta nhận được cây trong hình 8.12b.

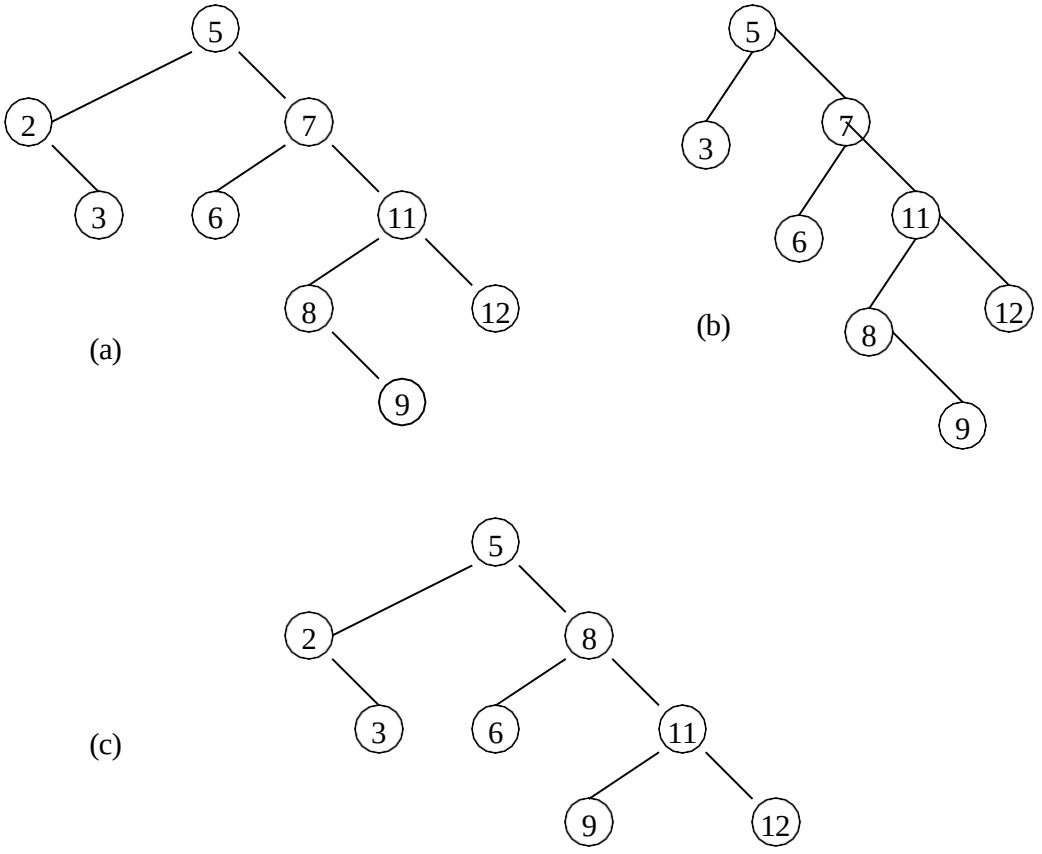


Hình 8.12. (a) Xen vào cây 8.11b đỉnh mới có khóa 6
(b) Xen vào cây 8.11b đỉnh mới có khóa 11

Phép toán loại (Delete). Chúng ta cần phải loại khỏi cây T một đỉnh chứa dữ liệu với khóa là k. Phép toán Delete là phép toán phức tạp nhất trong các phép toán trên cây tìm kiếm nhị phân. Trước hết, chúng ta cần phải áp dụng thủ tục tìm kiếm để định vị đỉnh cần loại trong cây. Giả sử đỉnh cần loại là đỉnh v. Các bước cần tiến hành tiếp theo phụ thuộc vào đỉnh v chỉ có nhiều nhất là một con hay có đầy đủ cả hai con. Chúng ta xét từng trường hợp:

1. Đỉnh v chỉ có nhiều nhất một con (tức là v chỉ có con phải, hoặc chỉ có con trái, hoặc v là lá). Trường hợp này rất đơn giản, chẳng hạn giả sử đỉnh v chỉ có con phải, khi đó nếu cha của v là đỉnh p và v là con trái của p thì ta chỉ cần đặt con trái của p là con phải của v. Chẳng hạn, xét cây trong hình 8.13a, đỉnh cần loại là đỉnh 2, đỉnh này chỉ có con phải là đỉnh 3, đỉnh 2 là con trái của đỉnh 5. Để loại đỉnh 2, ta chỉ cần đặt con trái của đỉnh 5 là đỉnh 3, ta nhận được cây trong hình 8.13b.

2. Chúng ta có nhận xét rằng, phép toán loại khỏi cây đỉnh có khóa nhỏ nhất (Delete Min) là trường hợp riêng của trường hợp này, bởi vì đỉnh có khóa nhỏ nhất là đỉnh ngoài cùng bên trái của cây, nó là đỉnh không có con trái.



Hình 8.13. (a) Một cây nhị phân
(b) Cây nhị phân (a) sau khi loại đỉnh 2
(c) Cây nhị phân (a) sau khi loại đỉnh 7

3. Đỉnh v có đầy đủ cả hai con, chẳng hạn đỉnh 7 trong cây hình 8.13. Vấn đề đặt ra là sau khi bỏ đi đỉnh v, chúng ta cần phải bố trí các đỉnh còn lại của cây như thế nào để nó vẫn còn là cây tìm kiếm nhị phân. Cách giải quyết vấn đề là quy về trường hợp đã xét, tức là chúng ta cần biến đổi cây thế nào để dẫn đến chỉ cần loại đỉnh có nhiều nhất một con. Trong cây con bên phải của đỉnh v, đỉnh có khóa nhỏ nhất là đỉnh ngoài cùng bên trái,

giả sử đó là đỉnh u. Nếu chúng ta thay dữ liệu chứa trong v bởi dữ liệu chứa trong đỉnh u thì thay vì loại đỉnh v ta chỉ cần loại đỉnh u, mà đỉnh u thì không có con trái. Chẳng hạn, trong cây hình 8.13a, giả sử đỉnh v là đỉnh chứa khóa 7, khi đó đỉnh u là đỉnh chứa khóa 8. Sau khi sao chép dữ liệu từ đỉnh u sang đỉnh v và loại đỉnh u ta nhận được cây trong hình 8.13c. Bạn đọc dễ dàng chứng minh được rằng các hành động như trên sẽ đảm bảo cây kết quả vẫn còn là cây tìm kiếm nhị phân (bài tập).

Tổng kết các bước đã trình bày trên, chúng ta đưa ra thuật toán loại đệ quy như sau:

```
void    Delete (T, k)
// Cây T không rỗng
// Loại khỏi cây T đỉnh chứa dữ liệu với khóa k
{
    if (k < rootData.key)
                Delete (TL, k); //Loại ở cây con trái
    else if (k > rootData.key)
                Delete (TR, k); // Loại ở cây con phải
          else if (cả hai TL và TR không rỗng)
                {
                    rootData = Min (TR);
                    DeleteMin (TR);
                }
               else if ( TL rỗng )
                                T = TR;
                    else        T = TL;
}
```

Chúng ta cũng có thể đưa ra thuật toán loại không đệ quy. Để xác định đỉnh cần loại, ta chỉ cần một biến v chạy trên các đỉnh của cây bắt đầu từ gốc và dừng lại tại đỉnh cần loại (như trong thuật toán tìm kiếm không đệ quy). Tiếp theo để xác định đỉnh ngoài cùng bên trái trong cây con phải của đỉnh v, chúng ta cần sử dụng hai biến: biên u ghi lại đỉnh ngoài cùng bên trái trong cây con phải của đỉnh v và biến p ghi lại cha của đỉnh u. Sao chép dữ

liệu từ đỉnh u lên đỉnh v. Rồi sau đó gắn con mới cho đỉnh p khi bỏ đi đỉnh u. Độc giả hãy viết ra thuật toán loại không đệ quy này (bài tập).

Chúng ta có nhận xét rằng, các phép toán Insert, Delete cũng chỉ đòi hỏi thời gian O(h), bởi vì quá trình xác định vị trí để xen hoặc vị trí để loại là quá trình đi từ gốc xuống lá và xem xét dữ liệu chứa trong các đỉnh trên đường đi đó.

8.5 CÀI ĐẶT TẬP ĐỘNG BỞI CÂY TÌM KIẾM NHỊ PHÂN

Trong mục trước chúng ta đã nghiên cứu các thuật toán thực hiện các phép toán tập động (Search, Insert, Delete, Min, Max, DeleteMin) khi tập dữ liệu được cài đặt bởi CTDL cây tìm kiếm nhị phân. Trong mục này, chúng ta sẽ nghiên cứu sự cài đặt KDLTT tập động. Trước hết, chúng ta sẽ giả thiết rằng, dữ liệu được lưu trong đỉnh của cây tìm kiếm nhị phân có kiểu Item, Item là kiểu bất kỳ chứa một trường **key** (khóa tìm kiếm) có kiểu là **keyType**, keyType là kiểu có thứ tự bất kỳ (chẳng hạn int, double, char, string). Để đơn giản cho cách viết, ta giả thiết rằng, chúng ta có thể truy cập trực tiếp tới trường key, chẳng hạn khi Item là một struct.

KDLTT tập động sẽ được cài đặt bởi lớp khuôn phụ thuộc tham biến kiểu Item, lớp này được đặt tên là lớp BSTree (lớp cây tìm kiếm nhị phân). Trước hết, chúng ta xây dựng lớp BSNode (lớp đỉnh cây tìm kiếm nhị phân). Lớp BSNode cùng là lớp khuôn phụ thuộc tham biến kiểu Item và được mô tả trong hình 8.14. Cần lưu ý rằng, tất cả các thành phần của lớp BSNode đều là private, nhưng khai báo lớp BSTree là bạn để các hàm thành phần của lớp BSTree có thể truy cập trực tiếp đến các thành phần của lớp BSNode, lớp BSNode chỉ chứa một hàm kiến tạo.

```
template <class Item>
class    BSTree; //khai thác trước
template <class Item>
class     BSNode
{
```

```
        Item   data;
        BSNode*   left;
        BSNode*    right;
        Node (const Item & element)
        // Khởi tạo một đỉnh chứa dữ liệu là element
        : data (element), left (NULL), right (NULL) {}
       friend    class   BSTree <Item>;
};
```

Hình 8.14. Lớp đỉnh cây tìm kiếm nhị phân

Bây giờ chúng ta thiết kế lớp BSTree. Lớp này chỉ chứa một thành phần dữ liệu là con trỏ root trỏ tới gốc cây. Chúng ta sẽ cài đặt các hàm thực hiện các phép toán tập động (các hàm Search, Insert, Delete...) theo các thuật toán đệ quy đã trình bày trong mục 8.4.2. Nhưng các hàm này nằm trong giao diện của lớp BSTree, chúng không chứa tham biến root. Vậy làm thế nào để có thể viết các lời gọi đệ quy? Kỹ thuật được sử dụng ở đây là, chúng ta đưa vào các hàm ẩn tương ứng, các hàm này chứa một tham biến là con trỏ trỏ tới gốc các cây con. Sử dụng các hàm ẩn, việc cài đặt các hàm trong giao diện của lớp sẽ rất đơn giản, chỉ cần gọi các hàm ẩn tương ứng với tham số là con trỏ root. Lớp BSTree được khai báo trong hình 8.15.

```
template <class  Item>
class      BSTree
{
 public:
       BSTree () // Khởi tạo cây rỗng
           {root = NULL;}
       BSTree (const BSTree & T); // Hàm kiến tạo copy
       BSTree & Operator = (const BSTree & T); //Toán tử gán.
       virtual ~ BSTree () // Hàm hủy
           { MakeEmpty (root); }
        bool    Empty () const
           {return root = = NULL;}
```

```
        void     Insert (const Item & element);
        // Xen vào dữ liệu mới element
        void     Delete (keyType k);
        // Loại dữ liệu có khóa là k
        Item &   DeleteMin ()
        // Loại dữ liệu có khóa nhỏ nhất; hàm trả về
        // dữ liệu này, nếu cây không rỗng
        bool     Search (keyType k, Item & I)   const;
        // Tìm dữ liệu có khóa k, biến I lưu lại dữ liệu đó.
        // Hàm trả về true nếu tìm thấy, false nếu không.
        Item &    Min ()   const;
        // Hàm trả về dữ liệu có khóa nhỏ nhất, nếu cây không rỗng.
        Item &    Max ()   const;
        // Hàm trả về dữ liệu có khóa lớn nhất, nếu cây không rỗng.
        typedef    BSNode <Item>   Node;
private:
        Node *   root;
        // Các hàm ẩn phục vụ cho cài đặt các hàm public:
        void    MakeEmpty (Node * & P);
        // Làm cho cây gốc trỏ bởi P trở thành rỗng, thu hồi các
        // tế bào nhớ đã cấp phát cho các đỉnh của cây.
        void     Insert (const Item & element, Node* & P);
        // Xen dữ liệu mới element vòa cây gốc trỏ bởi P.
        void    Delete (keyType k, Node* & P);
        // Loại dữ liệu có khóa k khỏi cây gốc trỏ bởi P
        Item &   DeleteMin (Node* & P);
        // Loại dữ liệu có khóa nhỏ nhất khỏi cây gốc trỏ bởi P
        Item &    Min(Node * P) const;
        // Hàm trả về dữ liệu có khóa nhỏ nhất
        // trên cây gốc trỏ bởi P.
        Item &    Max( Node * P)  const;
        // Hàm trả về dư liệu có khoá lớn nhất.
        Item & CopyTree (Node* Q, Node* & P);
        // Copy cây gốc trỏ bởi Q thành cây gốc trỏ bởi P
};
```

Hình 8.15. Lớp cây tìm kiếm nhị phân

Sau đây chúng ta cài đặt các hàm thành phần của lớp BSTree. Các hàm trong mục public được cài đặt rất đơn giản, chỉ cần gọi các hàm ẩn tương ứng và thay tham biến P bởi con trỏ root. Chúng ta viết ra một số hàm

```
template <class Item>
BSTree <Item> : : BSTree (const BSTree <Item> & T)
{
     root = NULL;
     *this = T;
}

template <class Item>
BSTree<Item> & BSTree <Item>: : Operator = (const BSTree <Item>
& T)
{
    if (this ! = T)
    {
          MakeEmpty (root);
          if (T.root ! = NULL)
          CopyTree (&T, root);
     }
    return    * this;
}

template <clas Item>
void BSTree <Item> : : Insert (const Item & element)
{
    Insert (element, root);
}
```

Các hàm thành phần public còn lại được cài đặt tương tự, chỉ chứa một lời gọi hàm ẩn tương ứng, bạn đọc hãy tự viết ra các hàm đó.

Vấn đề còn lại của chúng ta là cài đặt các hàm ẩn. Tất cả các hàm ẩn này đều có một điểm chung là các hàm đệ quy, chứa một tham biến để gọi đệ quy, đó là biến con trỏ P trỏ tới gốc cây. Chúng ta sẽ cài đặt các hàm đệ quy này theo các thuật toán đệ quy đã được đưa ra trong mục 8.4.2.

```
template <class Item>
void    BSTree <Item> : : MakeEmpty (Node* & P)
```

```
{
    if (P ! = NULL)
     {
        MakeEmpty (P → left);
       MakeEmpty (P → right);
       delete P;
       P = NULL;
     }
}

template <class Item>
void   BSTree <Item> : : Insert (const Item & element, Node * & P)
{
    if (P = = NULL)
          P = new  Node<Item> (element);
    else if (element.key < P → data.key)
                Insert (element, P → left);
          else if (element.key > P → data.key)
                Insert (element, P → right);
}

template <class Item>
Item &   BSTree <Item> : : DeleteMin (Node* & P)
{
    assert (P ! = NULL); // Kiểm tra cây không rỗng
    if (P → left ! = NULL)
            return    DeleteMin (P → left);
    else {
            Item   element = P → data;
             Node <Item>*   Ptr = P;
             P = P → right;
            delete    Ptr;
            return    element;
          }
}


template <class Item>
void   BSTree <Item> : : Delete (keyType k, Node* & P)
{
   if (P! = NULL)
```

```
        {
          if (k < P → data.key)
                Delete (k, P → left);
          else if (k> P → data.key)
                Delete (k, P → right);
          else
          if ((P → left != NULL) & & (P → right != NULL))
          // Đỉnh P có đầy đủ 2 con.
                P → data = DeleteMin (P → right);
          else {
                Node <Item> *    Ptr = P;
                P = (P → left ! = NULL)? P → left : P → right;
                delete    Ptr;
              }

          }
}

template <class Item>
bool BSTree <Item>: : Search (keyType k, Item & I, Node* P)
{
    if (P ! = NULL)
        if (k = = P → data.key
           {
                I = P → data;
                return    true;
           }
         else if (k < P → data.key)
                    return    Search (k, I, P → left);
              else      return    Search (k, I, P → right);
   else {
           I = *(new Item);
           return    false;
         }
}
```

Khi sử dụng hàm Search, cần chú ý rằng nếu không tìm thấy (hàm trả về false) thì giá trị lưu trong biến I chỉ là giá trị giả tạo!

```
template <class Item>
```

```
Item & BSTree <Item> : : Min(Node*  P)
{
    assert (P! = NULL);
    if (P → left ! = NULL)
            return    Min(P → left);
   else     return    P → data;
}
template <class Item>
void BSTree <Item> : : CopyTree (Node*Q, Node* & P)
{
   if (Q = = NULL)
        P = NULL;
   else {
            P = new Node <Item> (Q → data);
            CopyTree (Q → left, P → left);
            CopyTree (Q → right, P → right);
         }
}
```

Trên đây chúng ta đã trình bày một cách thiết kế lớp BSTree, trong đó chúng ta đã cài đặt các phép toán tập động (Search, Insert, Delete, Min, Max, DeleteMin) sử dụng các thuật toán đệ quy. Đương nhiên, bạn cũng có thể cài đặt các phép toán tập động không đệ quy, và do đó bạn không cần đưa vào các hàm ẩn. Bạn hãy thiết kế và cài đặt lớp BSTree theo cách đó (bài tập).

8.6 THỜI GIAN THỰC HIỆN CÁC PHÉP TOÁN TẬP ĐỘNG TRÊN CÂY TÌM KIẾM NHỊ PHÂN

Trong mục 8.4.2 chúng ta đã chỉ ra rằng, thời gian thực hiện các phép toán Search, Insert và Delete trên cây tìm kiếm nhị phân là O(h), trong đó h là độ cao của cây. Tuy nhiên, cùng một tập dữ liệu chúng ta có thể lưu trong các cây tìm kiếm nhị phân có độ cao rất khác nhau. Chúng ta đã chỉ ra điều đó trong hình 8.11, ở đó chúng ta đã đưa ra ba cây tìm kiếm nhị phân cùng biểu diễn một tập gồm 6 dữ liệu với các giá trị khóa là 4, 1, 3, 7, 5, 9. Xuất phát từ cây tìm kiếm nhị phân rỗng, bằng cách sử dụng phép toán xen vào

cây một dãy các dữ liệu, ta sẽ nhận được một cây tìm kiếm nhị phân biểu diễn tập dữ liệu đó. Chẳng hạn, chúng ta có cây trong hình 8.11a, khi chúng ta xen vào cây rỗng lần lượt các dữ liệu với các khóa 4, 7, 5, 1, 3, 9; cây này có độ cao là 3. Nhưng nếu chúng ta xen vào cây rỗng lần lượt các dữ liệu với các giá trị khóa được sắp xếp theo thứ tự tăng dần, chúng ta sẽ nhận được cây trong hình 8.11c. Cây 8.11c có đặc điểm là tất cả các cây con trái của mỗi đỉnh đều rỗng, nó có độ cao là 6, cây này thực sự là một DSLK, việc tìm kiếm, xen, loại trên cây này là tìm kiếm, xen, loại trên DSLK. Như vậy, cây tìm kiếm nhị phân biểu diễn tập N dữ liệu trong trường hợp xấu nhất (khi cây suy biến thành DSLK), thời gian thực hiện các phép toán Search, Insert, Delete trên cây tìm kiếm nhị phân là O(N).

Một câu hỏi được đặt ra là, có thể xây dựng được cây tìm kiếm nhị phân biểu diễn tập N dữ liệu với độ cao nhỏ nhất có thể được bằng bao nhiêu? Có thể chứng minh được khẳng định sau đây:

Độ cao nhỏ nhất của cây nhị phân N đỉnh là

$h = [\log_2(N + 1)]$

trong đó, [x] ký hiệu số nguyên nhỏ nhất $\geq$ x, chẳng hạn [3, 41] = 4.

Cây nhị phân hoàn toàn, hoặc cây nhị phân cân bằng sẽ là các cây có độ cao ngắn nhất như trên. Như vậy trong trường hợp tốt nhất thì thời gian thực hiện các phép toán Search, Insert, Delete là O(logN).

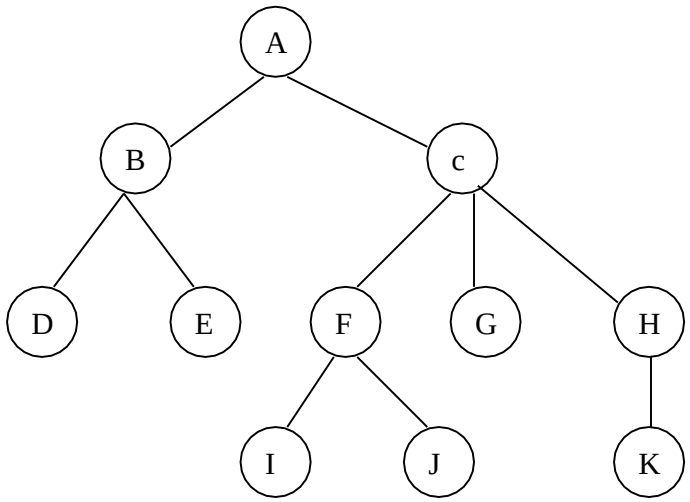
Cây tìm kiếm nhị phân ngẫu nhiên. Từ cây tìm kiếm nhị phân ban đầu rỗng, chúng ta xen vào cây N dữ liệu theo một thứ tự ngẫu nhiên; chúng ta sẽ gọi cây tìm kiếm nhị phân nhận được bằng cách đó là **cây tìm kiếm nhị phân được xây dựng một cách ngẫu nhiên**, hay gọn hơn: **cây tìm kiếm nhị phân ngẫu nhiên**.

Người ta đã chứng minh được rằng, độ cao trung bình của các cây tìm kiếm nhị phân ngẫu nhiêu với N đỉnh là O(logN). Từ kết quả này chúng ta suy ra rằng, thời gian trung bình của các phép toán Search, Insert, Delete, Min, Max,… trên cây tìm kiếm nhị phân ngẫu nhiên là O(logN).

BAI TẬP.

1. Hãy đưa ra cách biểu diễn cây bởi mảng. Mô tả CTDL biểu diễn cây theo cách đó bằng các khai báo trong C + +.

2. Cho cây:



   Hãy viết ra danh sách các đỉnh khi duyệt cây theo các thứ tự preorder, inorder và postorder.

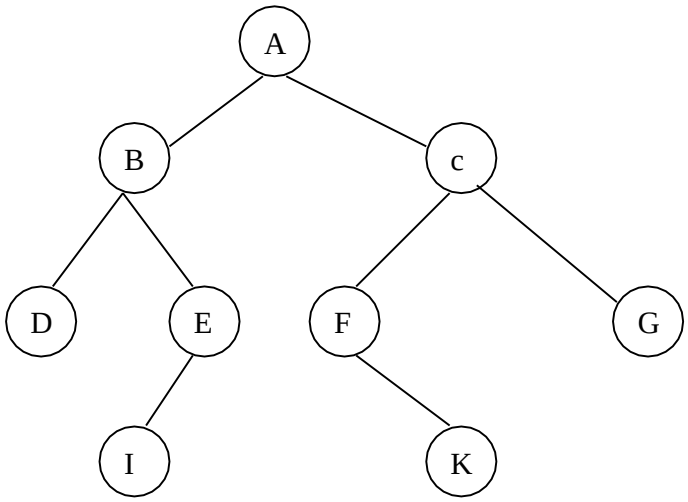
3. Giả sử cây được biểu diễn bằng cách sử dụng hai con trỏ firstChild và nextSibling. Bằng cách sử dụng ngăn xếp, hãy viết các hàm không đệ quy duyệt cây theo thứ tự inorder và postorder.

4. Mô tả và cài đặt lớp đỉnh cây nhị phân (class BinaryNode). Lớp này cần thoả mãn các đòi hỏi sau:
   - Chứa ba biến: data (lưu dữ liệu chứa trong đỉnh), hai con trỏ left và right.
   - Hàm kiến tạo để tạo ra một đỉnh chứa dữ liệu d cho trước, hai con trỏ left và right đều là NULL.
   - Chứa các hàm printPreorder( ),printInorder( ) và printPostorder( ) để in ra các dữ liệu lưu trong các đỉnh của cây con với gốc tại đỉnh đang xét theo các thứ tự trước, trong và sau.

5. Sử dụng lớp đỉnh cây nhị phân (bài tập 4), hãy đặc tả và cài đặt lớp cây nhị phân (class BinaryTree) theo các đòi hỏi và chỉ dẫn sau đây:

- Mục private chứa con trỏ root trỏ tới đỉnh gốc cây và các hàm ẩn cần thiết cho sự cài đặt các hàm trong mục public.
- Mục public chứa các hàm sau:
  - Hàm kiến tạo mặc định tạo ra cây rỗng.
  - Hàm kiến tạo ra cây chỉ có một đỉnh gốc chứa dữ liệu d cho trước.
  - Hàm kiến tạo copy.
  - Hàm huỷ.
  - Toán tử gán.
  - Hàm kiểm tra cây có rỗng không.
  - Hàm cho biết độ cao của cây.
  - Hàm cho biết số đỉnh trong cây.
  - Các hàm in ra các dữ liệu chứa trong các đỉnh của cây theo thứ tự preorder, inorder và postorder.

6. Hãy đặc tả và cài đặt ba lớp công cụ lặp trên cây nhị phân: PreIterator, InIterator và PostIterator, chứa các hàm công cụ giúp ta duyệt cây nhị phân theo các thứ tự trước, trong và sau. Mỗi lớp cần chứa hàm kiến tạo và bốn hàm sau:
   1) Hàm start( ) xác định đỉnh bắt đầu duyệt. Chẳng hạn, với cây nhị phân sau đây, nếu duyệt theo thứ tự trước thì bắt đầu là đỉnh A, còn nếu duyệt theo thứ tự trong và sau thì bắt đầu là đỉnh D.



   2) Hàm Valid( ) tương tự như trong lớp công cụ lặp trên danh sách liên kết.
   3) Hàm Current( ) trả về dữ liệu chứa trong đỉnh hiện thời.

4) Hàm Advance( ). Chẳng hạn, nếu đỉnh hiện thời là B, thì trong lớp PreIterator hàm Advance( ) cho ra đỉnh tiếp theo là D, còn trong lớp InIterator nó cho ra đỉnh tiếp theo là I, trong lớp PostIterator nó lại cho ra đỉnh tiếp theo là K.

Tương tự như trong lớp công cụ lặp trên danh sách liên kết, mỗi một trong ba lớp công cụ lặp trên cây nhị phân, cần chứa một con trỏ hằng trỏ tới gốc cây nhị phân, một con trỏ trỏ tới đỉnh hiện thời. Ngoài ra nó cần chứa một ngăn xếp để lưu các đỉnh trong quá trình duyệt. Chú ý rằng, cả ba lớp công cụ lặp trên cây nhị phân đều có giao diện và các biến thành phần giống nhau, chỉ có các hàm Start( ) và Advance( ) được cài đặt khác nhau để đảm bảo các đỉnh cây được thăm theo đúng thứ tự trước, trong và sau tương ứng.

7. Hãy vẽ ra cây tìm kiếm nhị phân được tạo thành bằng cách xen lần lượt các dữ liệu với các giá trị khoá là 5, 9, 2, 4, 1, 6, 10, 8, 3, 7, xuất phát từ cây rỗng. Sau đó hãy đưa ra cây kết quả khi loại gốc cây.

8. Trong lớp BSTree chúng ta đã cài đặt các hàm thực hiện các phép toán tập động bằng cách sử dụng các hàm ẩn đệ quy tương ứng. Hãy cài đặt các hàm đó (Search(k, I), Min( ), Max( ), Insert(element), Delete(k), DeleteMin( )) không đệ quy, và do đó không cần đưa vào các hàm ẩn.

9. Sử dụng cây tìm kiếm nhị phân, hãy đưa ra thuật toán sắp xếp mảng theo thứ tự khoá tăng dần, bằng cách sử dụng các phép toán Insert và DeleteMin.

10. Giả sử chúng ta có một tập dữ liệu, trong đó các dữ liệu khác nhau có thể có khoá bằng nhau. Hãy đưa ra cách cài đặt tập dữ liệu đó bởi cây tìm kiếm nhị phân. (Gợi ý: mỗi đỉnh cây chứa một danh sách các dữ liệu cùng khoá). Với CTDL cài đặt tập dữ liệu đã đưa ra đó, hãy cài đặt các hàm Search(d) (tìm dữ liệu d), Insert(d) (xen vào dữ liệu d) và Delete(d) (loại dữ liệu d).

11. Giả sử tập dữ liệu được lưu giữ dưới dạng cây tìm kiếm nhị phân. Bài toán tìm kiếm phạm vị được xác định như sau: Cho hai giá trị khoá $k_1 < k_2$, ta cần tìm tất cả các dữ liệu d mà $k_1 \le d.key \le k_2$. Hãy thiết kế và cài đặt thuật toán cho bài toán tìm kiếm phạm vi.