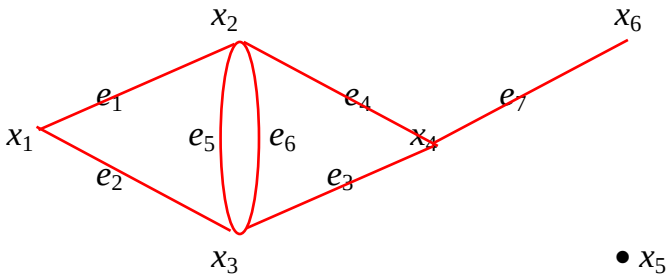
CHƯƠNG IV

CÂY

I. CÁC KHÁI NIỆM CƠ BẢN

1. Đồ thị. *Đồ thị* là tập hợp các đỉnh (nút) và các cạnh nối các đỉnh. Ký hiệu đồ thị $G = (V, E)$, trong đó V là tập hợp các đỉnh, E là tập hợp các cạnh.

◊ *Ví dụ*



Hình trên là đồ thị $G = (V,E)$ trong đó tập các đỉnh là

$$V = \{x_1, x_2, x_3, x_4, x_5, x_6\}$$

và tập các cạnh là

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$$

Đường đi là dãy các đỉnh và các cạnh nối liên tiếp nhau.

Độ dài đường đi là số các cạnh của nó.

Ví dụ, trong đồ thị trên dãy

$$(x_1, e_1, x_2, e_4, x_4, e_7, x_6)$$

là đường đi từ x_1 đến x_6 có độ dài bằng 3.

Chu trình là *đường đi khép kín* (có đỉnh đầu và đỉnh cuối trùng nhau).

Ví dụ, trong đồ thị trên

$$(x_1, e_1, x_2, e_4, x_4, e_3, x_3, e_2, x_1)$$

là chu trình từ x_1 quay về x_1.

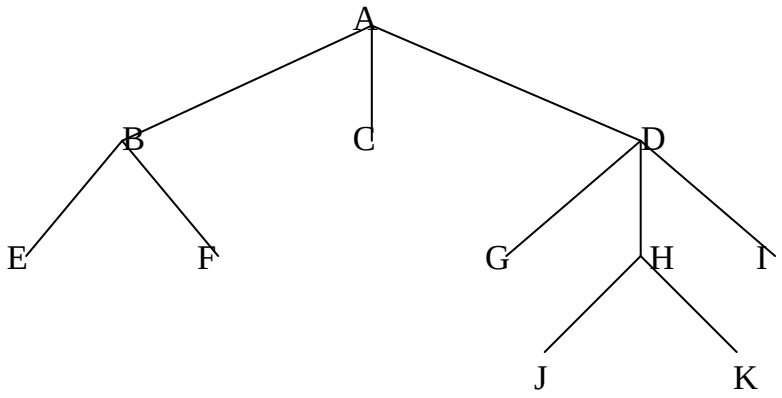
Đồ thị G gọi là *liên thông* nếu giữa hai đỉnh bất kỳ bao giờ cũng tồn tại đường đi nối chúng với nhau.

Đồ thị trên không liên thông vì từ x_5 không có đường đi đến các đỉnh khác. Đỉnh x_5 gọi là *đỉnh cô lập*.

2. Cây. *Cây* là đồ thị liên thông không chứa chu trình (giữa 2 đỉnh bất kỳ của cây chỉ tồn tại một đường đi duy nhất nối chúng với nhau).

◊ *Ví dụ:* Một người tên A có ba con là B, C và D. B có hai con là E và F. C độc thân không có con. D có ba con là G, H và I. H có 2 con là J và K.

Khi đó, sơ đồ gia phả của A có thể biểu diễn như một cây sau



Gốc của cây là một đỉnh đặc biệt do người dùng ấn định, thông thường là đỉnh trên cùng của cây.

Cho T là cây có gốc v_0. Giả sử x, y, z là các đỉnh trong T và $(v_0, v_1, ..., v_n)$ là đường đi trong T. Khi đó ta gọi

v_{k-1} là *cha* của v_k, $\forall k$=1, …, n.

$v_0,...,v_{k-1}$ là *tiền bối* của v_k, $\forall k$=1, …, n.

v_k là *con* của v_{k-1}, $\forall k$=1, …, n.

y là *hậu thế* của x, nếu x là tiền bối của y

y và z là *anh em* nếu chúng đều là con của đỉnh x

Nếu tập các nút rỗng ta có cây *rỗng*.

Một nút (không phải gốc) và các nút hậu thế của nó tạo thành cây gọi là *cây con*.

Bậc của nút là số nút con của nút đó.

Bậc của cây là bậc lớn nhất của các nút. Cây có bậc n gọi là *cây n-phân*.

Nút trong là nút có bậc > 0.

Nút ngoài (*nút lá*) là nút có bậc bằng 0.

Mức của nút gốc là 0. Các con của gốc có mức là 1.

Nếu một nút có mức m, thì các nút con của nó có mức là m+1.

Chiều cao (*sâu*) của cây là mức lớn nhất của các nút.

Quay lại ví dụ trên. Giả sử ta chọn nút A là gốc. Khi đó

Các nút B, C, D là anh em vì cùng là con của A. A là cha của B, C, D.

D là nút trong có bậc là 3 và có mức là 1.

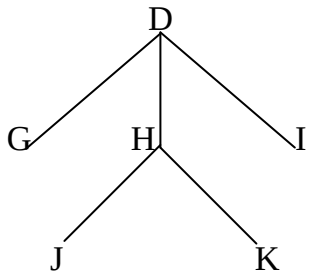
C là nút ngoài có bậc là 0 và có mức là 1.

J, K là các nút ngoài có bậc là 0 và có mức là 3.

Cây có chiều cao là 3 và bậc là 3.

Đây là cây tam phân.

Cây con xuất phát từ nút D là cây sau



II. CÂY NHỊ PHÂN

1. Định nghĩa và tính chất

Cây nhị phân là một dạng cấu trúc cây quan trọng, mỗi nút của nó chỉ có tối đa hai nút con, và tồn tại nút có 2 nút con.

Với mỗi nút trên cây nhị phân, cây con xuất phát từ nút con trái gọi là *cây con trái* và cây con xuất phát từ nút con phải gọi là *cây con phải* của nó.

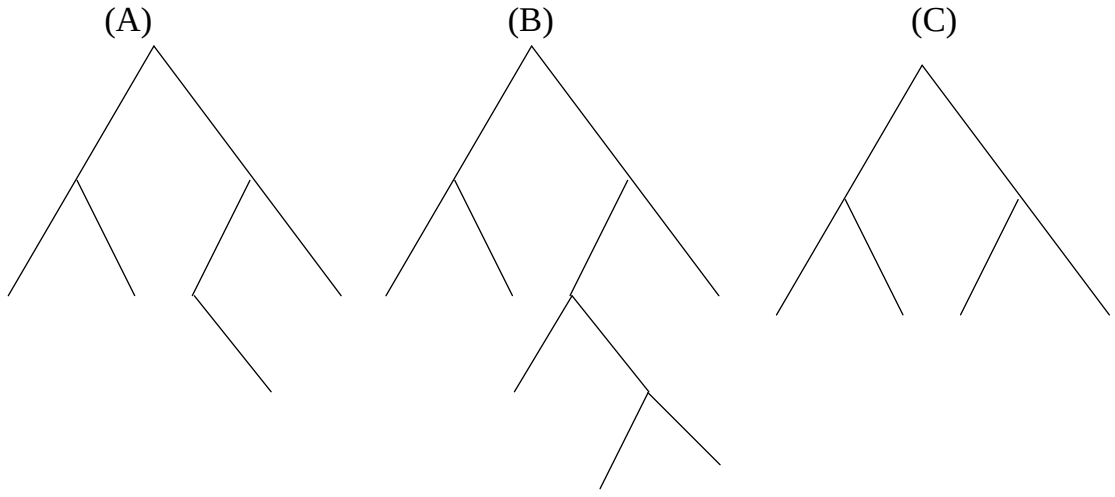
Cây nhị phân đầy đủ là cây thoả mãn mỗi nút trong của nó có bậc 2.

Cây nhị phân cân bằng là cây nhị phân thoả mãn mọi nút lá đều có mức bằng h hoặc $(h-1)$ với h là chiều cao của cây.

Cây nhị phân cân bằng hoàn toàn là cây nhị phân thoả mãn với mọi nút, số nút cây con trái và số nút cây con phải chênh lệch nhau không quá 1.

◊ *Ghi chú*. Cây nhị phân cân bằng hoàn toàn cũng là cây nhị phân cân bằng.

◊ *Ví dụ.* Trong các cây sau, (A) là cây nhị phân cân bằng hoàn toàn, nhưng không đầy đủ, (B) là cây nhị phân đầy đủ, nhưng không cân bằng, (C) là cây nhị phân đầy đủ, cân bằng hoàn toàn.



• ***Định lý***

(a) Số lượng tối đa các nút mức k ($k \geq 0$) trên cây nhị phân là 2^k.

(b) Số lượng tối đa các nút trên cây nhị phân chiều cao h là $2^{h+1} - 1$.

Chứng minh

(a) Chứng minh bằng quy nạp.

◊ Trường hợp $k = 0$: Chỉ có một nút mức 0, đó là gốc. Ta có $2^0 = 1$, vậy mệnh đề đúng.

◊ Giả sử mệnh đề đúng với $(k-1)$, ta chứng minh nó đúng với k. Thật vậy, ký hiệu V_i là số nút mức i. Khi đó theo giả thiết quy nạp ta có $V_{k-1} \le 2^{k-1}$. Vì mỗi nút mức k đều phải là con của nút mức $k-1$, mà mỗi nút chỉ có tối đa 2 con, nên

$$V_k \le 2.V_{k-1} \le 2.2^{k-1} = 2^k$$

(b) Theo mệnh đề (a), số lượng các nút trên cây nhị phân không vượt quá tổng

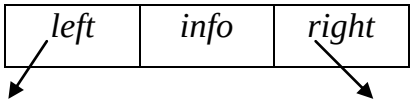
$$2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} - 1$$

2. Biểu diễn cây nhị phân

Để biểu diễn cây nhị phân ta dùng danh sách chứa dữ liệu *info* và có hai vùng liên kết:

Vùng *left* chỉ đến nút con bên trái.

Vùng *right* chỉ đến nút con bên phải.



và một con trỏ *root* chỉ vào nút gốc.

a. Khai báo cây

Trong PASCAL	Trong C
Type InfoType = <kiểu dữ liệu>; *NodePointer = ^Node;* *Node = record* *info : InfoType;* *left : NodePointer;* *right: NodePointer;* *end;* *Var root : NodePointer;*	*typedef <kiểu dữ liệu> InfoType;* *struct Node {* *InfoType info;* *struct Node *left;* *struct Node *right;* *};* *typedef struct Node *NodePointer;* *NodePointer root ;*

b. Khởi tạo cây rỗng

Trong PASCAL	Trong C
Procedure Initialize; *begin* *root := nil;* *end;*	*void Initialize*() { *root = NULL;* }

3. Duyệt cây nhị phân

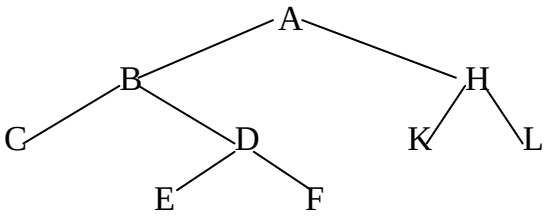
Phép *duyệt cây* là lần lượt đi qua tất cả các nút và mỗi nút chỉ qua 1 lần. Có 6 cách duyệt cây dựa vào thứ tự duyệt của các nút bên trái (L), nút gốc (N) và các nút bên phải (R) là:

Thứ tự trước (*Preorder*) : NLR, NRL (nút gốc trước)

Thứ tự giữa (*Inorder*) : LNR, RNL (nút gốc giữa)

Thứ tự sau (*Postorder*) : LRN, RLN (nút gốc sau)

◊ *Ví dụ.* Duyệt cây nhị phân sau (cây đầy đủ, cân bằng nhưng không cân bằng hoàn toàn):



Giả sử A là nút gốc, khi đó ta có các kết quả duyệt như sau:

NLR : ABCDEFHKL NRL : AHLKBDFEC

LNR : CBEDFAKHL RNL : LHKAFDEBC

LRN : CEFDBKLHA RLN : LKHFEDCBA

Ta nhận thấy thứ tự của NLR ngược với RLN, LNR ngược với RNL, LRN ngược với NRL. Do đó ta chỉ xét ba phép duyệt cơ bản là NLR, LNR, LRN.

a. Duyệt cây theo thứ tự NLR

• *Giải thuật đệ quy*

Trong PASCAL	Trong C
procedure Traverse_NLR(p: NodePointer); *begin* *if p <> nil then* *begin* *write(p^.info);* *Traverse_NLR(p^.left);* *Traverse_NLR(p^.right);* *end;* *end;*	*void Traverse_NLR(NodePointer p)* { *if (p != NULL)* { *printf("%d", p->info);* *Traverse_NLR(p->left);* *Traverse_NLR(p->right);* } }

◊ *Ghi chú:* Thủ tục gọi là *Traverse_NLR(root);*

• *Giải thuật không đệ quy*

Trong giải thuật này ta dùng ngăn xếp để khử đệ quy. Sau đây là một số tham số dùng trong chương trình.

Hằng số *StackNo* chỉ số phần tử của ngăn xếp.

Mảng *stack* lưu trữ ngăn xếp.

Biến *sp* là con trỏ ngăn xếp.

Biến logic *cont* (tiếp tục) có giá trị *false* khi giải thuật kết thúc.

◊ Trong PASCAL:

```
procedure Traverse_NLR;
var  cont : boolean;
     p    : NodePointer;
     stack: array[1..stackno] of  NodePointer; {ngăn xếp}
     sp   : 0..stackno; {con trỏ ngăn xếp}
begin
     p := root;
     sp := 0;
     cont := true;
     while  (p <> nil)  and  cont  do
          begin
               write(p^.info);
               if  p^.right <> nil  then
                    begin
                         {chứa nút con bên phải vào stack}
                         sp := sp + 1;
                         stack[sp] := p^.right
                    end;
               if  p^.left <> nil  then
                    p := p^.left
               else
                    if  sp = 0  then {giải thuật kết thúc}
                         cont := false
                    else
                         begin
                         {lấy ra từ stack}
                              p := stack[sp];
                              sp := sp - 1
                         end;
          end;
end;
```

◊ Trong C:

```
void Traverse_NLR()
{
    int cont ;
    NodePointer p;
    NodePointer stack[stackno+1];// ngăn xếp
    unsigned int sp; //con trỏ ngăn xếp
    p = root;
    sp = 0;
    cont = 1;
    while ((p != NULL) &&  cont)
        {
            printf("%d",p->info);
            if  (p->right != NULL)
                {
                    /*chứa nút con bên phải vào stack*/
                    sp = sp + 1;
                    stack[sp] = p->right;
                }
            if  (p->left != NULL)
                p = p->left;
            else
                if  (sp == 0) //giải thuật kết thúc
                    cont = 0;
                else
                    {  // lấy ra từ stack
                        p = stack[sp];
                        sp = sp – 1;
                    }
        }
}
```

b. Duyệt cây theo thứ tự LNR

- *Giải thuật đệ quy*

Trong PASCAL	Trong C
procedure Traverse_LNR(p:NodePointer); *begin* *if p <> nil then* *begin* *Traverse_LNR(p^.left);* *write(p^.info);* *Traverse_LNR(p^.right);* *end;* *end;*	*void Traverse_LNR(NodePointer p)* { *if (p != NULL)* { *Traverse_LNR(p->left);* *printf("%d", p->info);* *Traverse_LNR(p->right);* } }

◊ *Ghi chú:* Thủ tục gọi là *Traverse_LNR*(*root*);

c. Duyệt cây theo thứ tự LRN

• *Giải thuật đệ quy*

Trong PASCAL

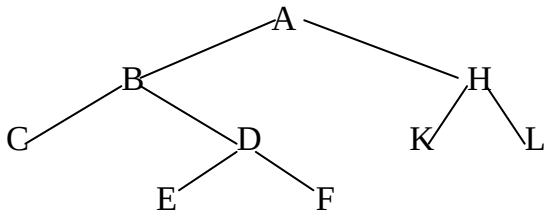
```
procedure Traverse_LRN(p:NodePointer);
begin
    if p <> nil then
        begin
            Traverse_LRN(p^.left);
            Traverse_LRN(p^.right);
            write(p^.info);
        end;
end;
```

Trong C

```
void Traverse_LRN(NodePointer p)
{
    if (p != NULL)
    {
        Traverse_LRN(p->left);
        Traverse_LRN(p->right);
        printf("%d", p->info);
    }
}
```

◊ *Ghi chú :* Thủ tục gọi là *Traverse_LRN*(*root*);

◊ *Ví dụ:* Viết chương trình tạo cây nhị phân và thực hiện các phép duyệt cây ở hình sau (cây đầy đủ, cân bằng nhưng không cân bằng hoàn toàn). Chương trình sử dụng các thủ tục duyệt cây trình bày ở trên.



Trong PASCAL

```
Type InfoType    = char;
     NodePointer   = ^node;
     node = record
               info : InfoType;
               left, right: NodePointer;
            end;
Var Root, AddPtr, CurPtr: NodePointer;
Procedure CreateNode(c: char);
begin
    new(AddPtr);
    with  AddPtr^  do
        begin
            info  := c;
            left  := nil;
            right := nil;
        end;
```

Trong C

```
typedef   char InfoType;
struct  node {
                InfoType info;
                struct node *left;
                struct node *right;
            };
typedef struct node * NodePointer;
NodePointer Root, AddPtr, CurPtr ;
void CreateNode(char c);
main()
{
        Root = NULL;
        //nhập 'A'
        CreateNode('A');
        Root = AddPtr;
        //nhập 'B'
```

```
end;
BEGIN {chương trình chính}
        Root := nil;
        {nhập 'A'}
        CreateNode('A');
        root := AddPtr;
        {nhập 'B'}
        CreateNode('B');
        root^.left := AddPtr;
        CurPtr := AddPtr;
        {nhập 'C'}
        CreateNode('C');
        CurPtr^.left := AddPtr;
        {nhập 'D'}
        CreateNode('D');
        CurPtr^.right := AddPtr;
        CurPtr := AddPtr;
        {nhập 'E'}
        CreateNode('E');
        CurPtr^.left := AddPtr;
        {nhập 'F'}
        CreateNode('F');
        CurPtr^.right := AddPtr;
        {nhập 'H'}
        CreateNode('H');
        root^.right := AddPtr;
        CurPtr := AddPtr;
        {nhập 'K'}
        CreateNode('K');
        CurPtr^.left := AddPtr;
        {nhập 'L'}
        CreateNode('L');
        CurPtr^.right := AddPtr;
        { Duyệt cây theo thứ tự NLR}
        Traverse_NLR(Root);
        { Duyệt cây theo thứ tự LNR}
        Traverse_LNR(Root);
        { Duyệt cây theo thứ tự LRN}
        Traverse_LRN(Root);
END.
```

```
        CreateNode('B');
        Root->left = AddPtr;
        CurPtr = AddPtr;
        //nhập 'C'
        CreateNode('C');
        CurPtr->left = AddPtr;
        //nhập 'D'
        CreateNode('D');
        CurPtr->right = AddPtr;
        CurPtr = AddPtr;
        //nhập 'E'
        CreateNode('E');
        CurPtr->left = AddPtr;
        //nhập 'F'
        CreateNode('F');
        CurPtr->right = AddPtr;
        //nhập 'H'
        CreateNode('H');
        Root->right = AddPtr;
        CurPtr = AddPtr;
        //nhập 'K'
        CreateNode('K');
        CurPtr->left = AddPtr;
        //nhập 'L'
        CreateNode('L');
        CurPtr->right = AddPtr;
        //Duyệt cây theo thứ tự NLR
        Traverse_NLR(Root);
        //Duyệt cây theo thứ tự LNR
        Traverse_LNR(Root);
        // Duyệt cây theo thứ tự LRN
        Traverse_LRN(Root);
}
void CreateNode(char c)
{
    AddPtr =
(NodePointer)malloc(sizeof(struct Node));
    AddPtr->info    = c;
    AddPtr->left    = NULL;
    AddPtr->right   = NULL;
}
```

4. Tính số nút, chiều cao của cây

- *Tính số nút*

Trong PASCAL

```
Function  sonut(t: NodePointer): integer;
begin
     if  (t = nil)   sonut := 0
     else  sonut := 1 + sonut(t^.left) +
          sonut(t^.right) ;
end;
```

Trong C

```
int  sonut(NodePointer  t)
{
     if  (t == NULL)    return 0 ;
     else  return (1 + sonut(t->left) +
          sonut(t->right));
}
```

- *Tính số nút lá*

Trong PASCAL

```
Function  sonutla(t: NodePointer): integer;
begin
     if  (t = nil)   sonutla := 0
     else
          if (t^.left = nil) and (t^.right = nil)
               then sonutla := 1
          else
               sonutla := sonutla(t^.left) +
                          sonutla(t^.right)
end;
```

Trong C

```
int  sonutla(NodePointer t)
{
     if  (t == NULL)   return  0;
     else
          if (( t->left == NULL) &&
            (t->right==NULL)) return 1;
          else
               return  (sonutla(t->left) +
                    sonutla(t->right));
}
```

- *Tính chiều cao*

Sử dụng hàm *max*(*x*,*y*) trả về số lớn hơn trong hai số *x* và *y*.

Trong PASCAL

```
Function chieucao(t: NodePointer): integer;
begin
     if  (t = nil)   chieucao := -1
     else  chieucao := 1+ max
          (chieucao(t^.left), chieucao(t^.right));
end;
```

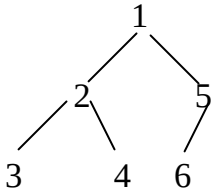
Trong C

```
int  chieucao(NodePointer t)
{
     if  (t == NULL)  return  -1;
     else  return (1+ max
(chieucao(t->left), chieucao(t->right)));
}
```

5. Cây nhị phân cân bằng hoàn toàn

Cây nhị phân cân bằng hoàn toàn là cây nhị phân thoả mãn với mọi nút số nút cây con trái và số nút cây con phải chênh lệch nhau không quá 1.

◊ *Ví dụ*. Cây cân bằng hoàn toàn có 6 nút



- ***Tạo cây cân bằng hoàn toàn có n nút***

Giả sử các giá trị khóa được nhập từ bàn phím và cây được khai báo như ở mục 2.

Trong PASCAL

```
Function CayCBHT(n: integer) NodePointer;
Var t: NodePointer;
    x: integer;
begin
    if (n = 0) CayCBHT := nil
    else
        begin
            new(t) ;
            readln(x);
            t^.info := x ;
            t^.left := CayCBHT(n div 2) ;
            t^.right :=CayCBHT(n –1
                                – n div 2);
            CayCBHT := t ;
        end;
end;
```

Trong C

```
NodePointer CayCBHT(int n)
{
    NodePointer t ;
    if (n == 0) return NULL;
    else
    {
        t = (NodePointer)
            malloc(sizeof(struct node));
            scanf("%d", &x);
            t->info = x;
            t->left =CayCBHT(n /2);
            t->right =CayCBHT(n –1
                              – n/2);
            return t ;
    }
}
```

III. CÂY NHỊ PHÂN TÌM KIẾM

1. Cây nhị phân tìm kiếm (Binary Search Tree - BST)

Cây nhị phân tìm kiếm là cây nhị phân mà tại mỗi nút của cây thoả mãn điều kiện: khoá của nó lớn hơn khoá của các nút cây con bên trái và nhỏ hơn hoặc bằng khoá của các nút cây con bên phải.

Ký hiệu

K_n là khoá nút xét;

K_l là khoá nút bất kỳ thuộc cây con bên trái;

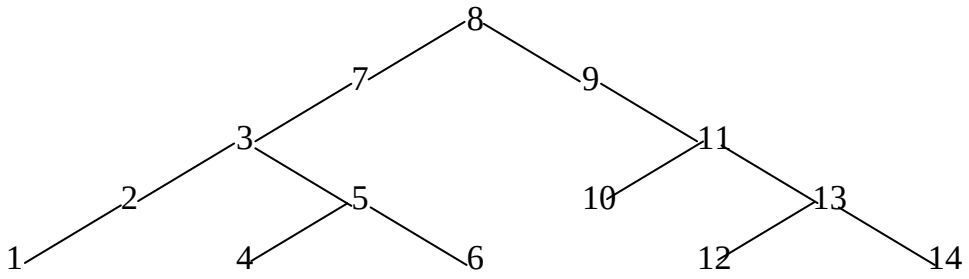
K_r là khoá nút bất kỳ thuộc cây con bên phải.

Khi đó

$$K_l < K_n \le K_r$$

Như vậy, theo định nghĩa thì khi duyệt cây nhị phân tìm kiếm theo thứ tự giữa (*LNR*) ta được dãy có thứ tự khóa tăng dần.

◊ *Ví dụ*. Cho cây nhị phân tìm kiếm



Khi duyệt theo thứ tự giữa LNR ta có dãy

1 2 3 4 5 6 7 8 9 10 11 12 13 14

Trước hết ta nhắc lại ***khai báo cây nhị phân***:

Trong PASCAL	Trong C
Type InfoType = <kiểu dữ liệu>; *NodePointer = ^Node;* *Node = record* *info : InfoType;* *left, right: NodePointer;* *end;* *Var root : NodePointer;*	*typedef <kiểu dữ liệu> InfoType;* *struct Node {* *InfoType info;* *struct Node *left;* *struct Node *right;* }; *typedef struct Node *NodePointer;* *NodePointer root ;*

2. Tìm kiếm

Hàm sau tìm trong cây nhị phân tìm kiếm gốc *root* nút *LocPtr* chứa *SearchInfo* và nút cha *parent* của nó. Nếu tìm thấy *found* trả giá trị *true* (hoặc 1), ngược lại *found* trả giá trị *false* (hoặc 0).

◊ Trong PASCAL:

```
Procedure BSTSearch(root : NodePointer; SearchInfo : InfoType;
        var LocPtr, parent : NodePointer; var found : boolean);
begin
    LocPtr := root;
    parent := nil;
    found := false;
    while not found and (LocPtr <> nil) do
        begin
            parent := LocPtr;
            if  SearchInfo < LocPtr^.info  then
                LocPtr := LocPtr^.left
            else
                if  SearchInfo > LocPtr^.info  then
                    LocPtr := LocPtr^.right
                else { SearchInfo = LocPtr^.info }
                    found := true
        end; {while}
end;
```

◊ Trong C:

```
void  BSTSearch(NodePointer root ; InfoType SearchInfo ;
                    NodePointer *LocPtr, *parent ; int *found)
{
    *LocPtr = root;
    *parent = NULL;
    *found = 0;
    while ((!found) && (LocPtr != NULLl))
        {
            *parent = *LocPtr;
            if  (SearchInfo < (*LocPtr)->info)
                *LocPtr = (*LocPtr)->left;
            else
                if  (SearchInfo > (*LocPtr)->info)
                    *LocPtr = (*LocPtr)->right;
                else
                    *found = 1;
        }
}
```

Gọi thủ tục: *BSTSearch*(*root , SearchInfo , &LocPtr, &parent, &found*);

Độ phức tạp trung bình là : 1.386**log*$_2$*n* = O(*log*$_2$*n*)

Độ phức tạp xấu nhất là : *n*

3. Thêm nút mới

Thủ tục sau thêm nút mới với trường *info* có giá trị *NewInfo* vào cây nhị phân tìm kiếm gốc *root*.

a. Thủ tục không đệ quy

◊ Trong PASCAL:

```
Procedure  BSTInsert(var  root : NodePointer; NewInfo : InfoType);
var
    LocPtr,       {con trỏ tìm}
    Parent : NodePointer; {con trỏ nút cha}
    found : boolean;
begin
    BSTSearch(root, NewInfo, LocPtr, parent, found);
    if found then
        writeln('đã có mục này trong cây')
    else
        begin
            new(LocPtr);
            LocPtr^.info := NewInfo;
            LocPtr^.right := nil;
            LocPtr^.left := nil;
            if parent = nil  then  {cây rỗng}
                root := LocPtr
            else
                if  NewInfo < parent^.info then {chèn trái}
                    parent^.left := LocPtr
                else
                    parent^.right := LocPtr {chèn phải}
        end;
end;
```

◊ Trong C:

```
void  BSTInsert(NodePointer  *root; InfoType NewInfo)
{
    NodePointer LocPtr, parent; //con trỏ nút tìm, nút cha
    int found ;
    BSTSearch(root , NewInfo, &LocPtr, &parent, &found);
```

```
        if  (found)
            printf("đã có mục này trong cây");
        else
            {
                LocPtr = (NodePointer)malloc(sizeof(struct Node));
                LocPtr->info = NewInfo;
                LocPtr->right = NULL;
                LocPtr->left = NULL;
                if (parent == NULL) //cây rỗng
                    *root = LocPtr;
                else
                    if  (NewInfo < parent->info)// chèn trái
                        parent->left = LocPtr;
                    else //  chèn phải
                        parent->right = LocPtr;
            }
    }
```

Gọi thủ tục: *BSTInsert*(*&root, NewInfo*).

b. Thủ tục đệ quy

◊ Trong PASCAL:

```
    procedure  BSTInsert(NewInfo : InfoType; var root : NodePointer);
    begin
        if  root = nil then
            begin
                new(root);
                with root^ do
                    begin
                        info := NewInfo;
                        left := nil;
                        right:= nil;
                    end;
            end
        else
            if NewInfo < root^.info then
                BSTInsert(NewInfo, root^.left)
            else
                BSTInsert(NewInfo, root^.right);
    end;
```

Gọi thủ tục : *BSTInsert*(*x, root*);

◊ Trong C:

```
void  BSTInsert(InfoType NewInfo ; NodePointer *root)
{
    if  (*root == NULL)
        {
            *root =  (NodePointer)malloc(sizeof(struct Node));
            *root->info = NewInfo;
            *root->left = NULL;
            *root->right = NULL;
        }
    else
        if (NewInfo < *root->info)
            BSTInsert(NewInfo, *root->left);
        else
            BSTInsert(NewInfo, *root->right);
}
```

Gọi thủ tục : *BSTInsert*(*x, &root*);

4. Tìm và loại bỏ

Giả sử ta cần loại bỏ nút *x* sao cho vẫn giữ tính chất cây nhị phân tìm kiếm. Gọi *parent* là nút cha của *x*. Có ba khả năng sau:

a. Nút loại bỏ là nút lá

- Nếu *x* là nút con trái của *parent* thì gán

Trong Pascal: *parent^.left := nil;*

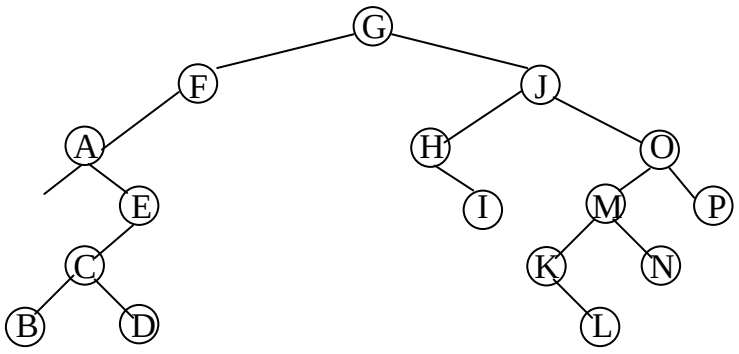
Trong C: *parent->left = NULL;*

- Nếu *x* là nút con phải của *parent* thì gán

Trong Pascal: *parent^.right := nil;*

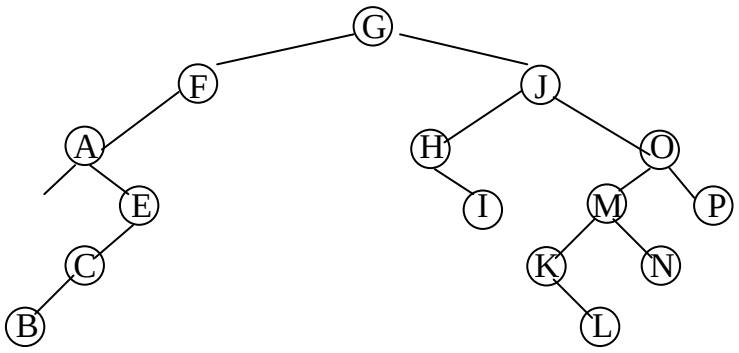
Trong C: *parent->right = NULL;*

◊ *Ví dụ:* Cho cây T_0 với gốc Ⓖ sau:



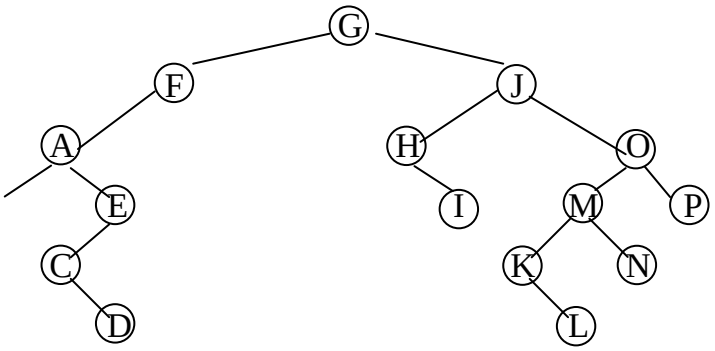
Cây T_0

Giả sử ta cần bỏ nút D. Vì D là nút lá và là nút con phải của nút C, nên ta chỉ cần trỏ con trỏ phải (*right*) của C đến *nil* hoặc NULL và sẽ nhận được cây T_1 sau



Cây T_1

Giả sử ta cần bỏ nút B khỏi cây T_0. Vì B là nút lá và là nút con trái của nút C, nên ta chỉ cần trỏ con trỏ trái (*left*) của C đến *nil* và sẽ nhận được cây T_2 sau



Cây T_2

b. Nút loại bỏ chỉ có một cây con child

- Nếu *x* là nút con trái của *parent* thì gán

Trong Pascal: *parent^.left := child;*

Trong C: *parent->left = child;*

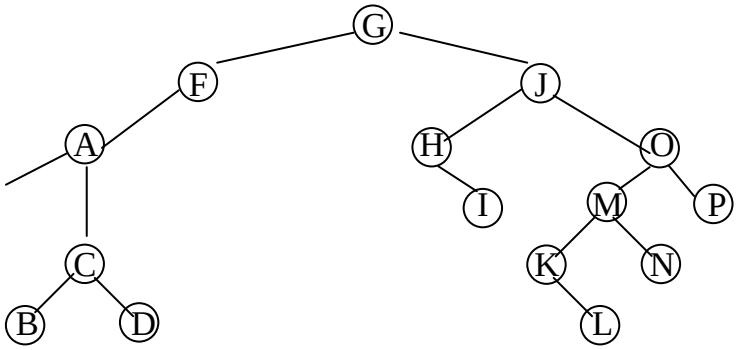
- Nếu *x* là nút con phải của *parent* thì gán

Trong Pascal: *parent^.right := child;*

Trong C: *parent->right = child;*

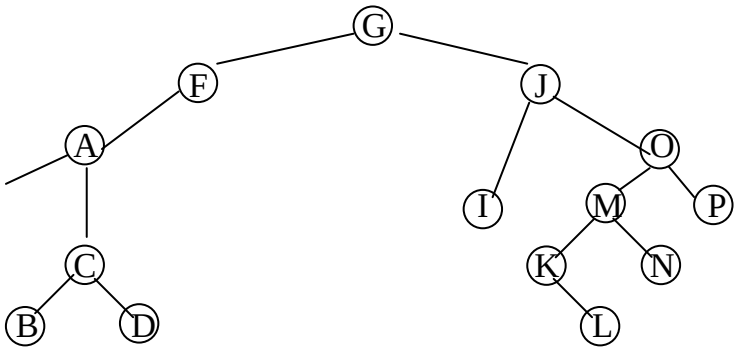
◊ *Ví dụ:* Xét cây T_0 cho ở trên.

Giả sử ta cần loại bỏ nút E, là nút chỉ có một nút con trái là C. Vì E là nút con phải của A, nên ta chỉ cần trỏ con trỏ phải (*right*) của A đến C và nhận được cây T_3 sau:



Cây T_3

Bây giờ giả sử ta cần loại bỏ khỏi cây T_3 nút H, là nút chỉ có một nút con phải là I. Vì H là nút con trái của J, nên ta chỉ cần trỏ con trỏ trái (*left*) của J đến I và nhận được cây T_4 sau:



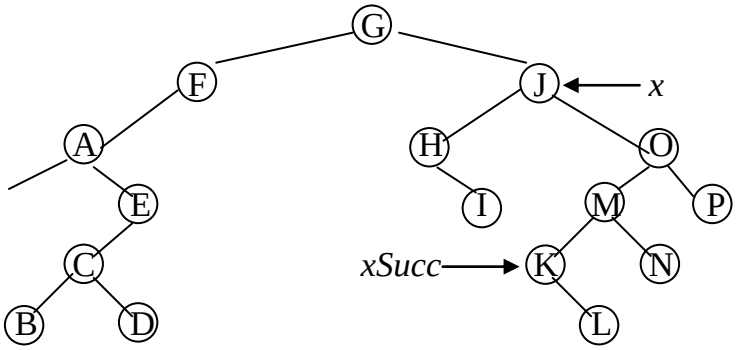
Cây T_4

c. Nút loại bỏ có cây con trái và phải

Có thể loại nút *x* bằng một trong hai cách đối xứng sau.

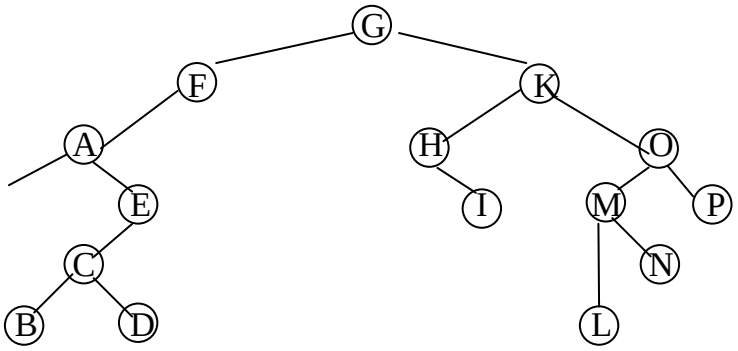
(*i*) Thay dữ liệu của nút *x* bằng dữ liệu của nút cực trái của cây con bên phải nút *x*. Sau đó xóa nút cực trái này. Nút cực trái này có thể là nút lá hoặc chỉ có cây con bên phải.

◊ *Ví dụ:* Giả sử ta cần loại khỏi cây T_0 nút J. Cây con bên phải của nút J là cây gốc O. Xuất phát từ nút O đi sang trái ta nhận được nút cực trái là K. Nút K chỉ có một nút con phải. Hình dưới minh hoạ quá trình trên. Ở đây *x* chỉ nút loại bỏ, và *xSucc* chỉ nút cực trái của cây con phải của *x*.



Cây T_0

Bây giờ thay dữ liệu của nút K (*xSucc*) vào nút J (*x*) và loại bỏ nút K theo mục b, ta nhận được cây T_5 sau

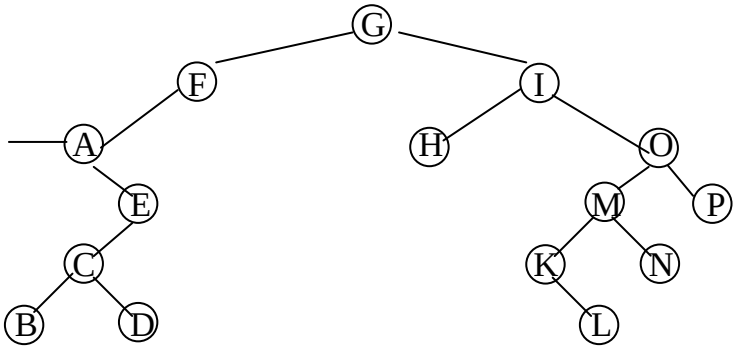


Cây T_5

(*ii*) Thay dữ liệu của nút *x* bằng dữ liệu của nút cực phải của cây con bên trái nút *x*. Sau đó xóa nút cực phải này. Nút cực phải này có thể là nút lá hoặc chỉ có cây con bên trái.

◊ *Ví dụ:* Giả sử ta cần loại khỏi cây T_0 nút J. Cây con bên trái của nút J là cây gốc H. Xuất phát từ nút H đi sang phải ta nhận được nút cực phải là I. Nút I là nút lá.

Bây giờ thay dữ liệu của nút I vào nút J và loại bỏ nút I theo mục a, ta nhận được cây T_6 sau



Cây T_6

♦ ***Giải thuật***

Tìm nút có *info = DelInfo* của cây gốc *root* và loại khỏi cây.

◊ Trong PASCAL:

```
Procedure BSTDelete(DelInfo : InfoType; var root : NodePointer);
var x,        {chỉ nút chứa DelInfo}
    parent,   {Cha của x hay xSucc}
    xSucc,    {phần tử sau của x}
    SubTree : NodePointer;      {cây con của x}
    found   : boolean;   {đúng , nếu tìm thấy DelInfo trong cây}
Begin
    BSTSearch(root, DelInfo, x, parent, found); {tìm nút x chứa DelInfo}
    if not found then
        writeln('mục không có trong cây')
    else
        begin
            if (x^.left <> nil) and (x^.right <> nil) then
                begin
                {tìm phần tử cực trái của cây con bên phải và cha của nó}
                    xSucc := x^.right;
                    parent := x;
                    while (xSucc^.left <> nil) do
                        begin    {đi xuống bên trái}
                            parent := xSucc;
                            xSucc := xSucc^.left;
                        end
                {Chuyển nội dung của xSucc vào x,và chỉ x đến xSucc sẽ xoá x}
                    x^.info := xSucc^.info;
                    x := xSucc;
                end;
            SubTree := x^.left;
            if SubTree = nil then
                SubTree := x^.right;
            if parent = nil then   {xoá nút gốc}
                root := SubTree
            else
                if parent^.left = x then
                    parent^.left := SubTree
                else
                    parent^.right := SubTree
            dispose(x);
        end
End;
```

Gọi thủ tục : *BSTDelete*(*DelInfo, root*);

◊ Trong C:

```
void BSTDelete(InfoType DelInfo ; NodePointer  *root )
```

```
{
    NodePointer x,        //chỉ nút chứa DelInfo
    parent,  //Cha của  x  hay  xSucc
    xSucc,   //phần tử sau của x
    SubTree ;    //cây con của x
    int found ;   //đúng , nếu tìm thấy DelInfo  trong cây
    BSTSearch(root, DelInfo, &x, &parent, &found);//tìm nút x chứa DelInfo
    if (!found)
        printf("mục không có trong cây");
    else
        {
            if ((x->left !=NULL) && (x->right != NULL))
                {
                //tìm phần tử cực trái của cây con bên phải và cha của nó
                    xSucc = x->right;
                    parent = x;
                    while (xSucc->left != NULL)
                        {    //đi xuống bên trái
                            parent = xSucc;
                            xSucc = xSucc->left;
                        }
            //Chuyển nội dung của xSucc vào x,và chỉ x đến xSucc sẽ xoá x
                    x->info = xSucc->info;
                    x = xSucc;
                }
            SubTree = x->left;
            if (SubTree == NULL)
                SubTree = x->right;
            if (parent == NULL) //   xoá nút gốc
                root = SubTree;
            else
                if (parent->left == x)
                    parent->left = SubTree;
                else
                    parent->right = SubTree;
            free(x);
        }
}
```

Gọi thủ tục : *BSTDelete*(*DelInfo, &root*);

5. Sắp xếp bằng cây nhị phân tìm kiếm

Sử dụng tính chất của cây nhị phân tìm kiếm ta có phương pháp sắp xếp bằng cây nhị phân như sau:

Bước 1: Khởi tạo cây nhị phân.

Bước 2: Lần lượt nhập dữ liệu và thêm vào cây.

Bước 3: Duyệt cây theo thứ tự giữa (Inorder) LNR sẽ nhận được danh sách sắp xếp tăng dần.

Bước 3': Duyệt cây theo thứ tự giữa (Inorder) RNL sẽ nhận được danh sách sắp xếp giảm dần.

◊ *Ví dụ:* Tạo cây nhị phân tìm kiếm có 10 nút (dữ liệu là số nguyên nhập từ bàn phím).

Trong PASCAL

```
BEGIN {chương trình chính}
    {khởi tạo cây}
    root := nil;
    {bước 2: Nhập và thêm vào cây}
    for i := 1 to 10 do
        begin
            readln(x);
            BSTInsert(x, root);
        end;
    {Duyệt cây thứ tự giữa LNR nhận được
    danh sách tăng dần}
    Traverse_LNR(root);
    {Duyệt cây thứ tự giữa RNL nhận được
    danh sách giảm dần}
    Traverse_RNL(root);
    {đọc phần tử xoá}
    readln(x);
    {xoá phần tử x}
    BSTDelete(x, root);
    {duyệt lại cây}
    Traverse_LNR(root);
END.
```

Trong C

```
main()
{
    //khởi tạo cây
    root = NULL;
    //bước 2: Nhập và thêm vào cây
    for (i = 1; i <= 10; i++)
        {
            scanf("%d",&x);
            BSTInsert(x, &root);
        }
    /*Duyệt cây LNR nhận được danh
    sách tăng dần*/
    Traverse_LNR(root);
    /*Duyệt cây RNL nhận được danh
    sách giảm dần*/
    Traverse_RNL(root);
    //đọc phần tử xoá
    scanf("%d",&x);
    //xoá phần tử x
    BSTDelete(x, &root);
    //duyệt lại cây
    Traverse_LNR(root);
}
```

BÀI TẬP

1. Viết chương trình tạo cây nhị phân tìm kiếm chứa các số nguyên (ít nhất 10 số) nhập từ bàn phím, rồi hiển thị chúng theo thứ tự giữa.

2. Viết chương trình nhập các số nguyên n, a, b ($n > 0, a < b, n << b-a$) và thực hiện các công việc sau:

a) Tạo cây nhị phân tìm kiếm chứa n số nguyên ngẫu nhiên trong khoảng $[a;b]$.

b) Duyệt cây theo thứ tự giữa LNR (tăng dần) và RNL (giảm dần).

c) Đếm số nút, số nút lá của cây.

d) Tính chiều cao của cây.

e) Xóa các nút chứa số lẻ.

3. Cho danh sách từ khóa Pascal

program, const, type, function, procedure, begin, end, div, mod, array, for, to, do, repeat, until, with, while, label, goto

Viết chương trình

a) Tạo cây nhị phân tìm kiếm lưu các từ trong danh sách trên.

b) Duyệt cây theo thứ tự giữa LNR và RNL.

c) Đếm số nút, số nút lá của cây.

d) Tính chiều cao của cây.

e) Xóa những từ chứa ký tự '*o*'.