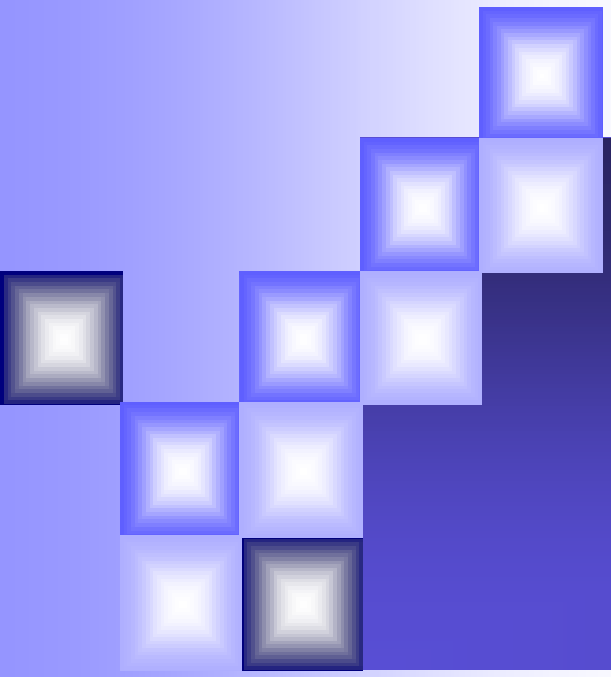


Cấu trúc cây

Mục tiêu

- Giới thiệu khái niệm cấu trúc cây.
- Cấu trúc dữ liệu cây nhị phân tìm kiếm: tổ chức, các thuật toán, ứng dụng.
- Giới thiệu cấu trúc dữ liệu cây nhị phân tìm kiếm

Cấu trúc cây

Cấu trúc cây
Một số định nghĩa

- **Định nghĩa 1:** cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có 1 nút đặc biệt được gọi là gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2, ... , Tn theo quan hệ phân cấp trong đó Ti cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta còn gọi là quan hệ cha-con.
- **Định nghĩa 2:** cấu trúc cây với kiểu cơ sở T là một nút cấu trúc rỗng được gọi là cây rỗng (NULL). Một nút mà thông tin chính của nó có kiểu T, nó liên kết với một số hữu hạn các cấu trúc cây khác cũng có kiểu cơ sở T. Các cấu trúc này được gọi là những cây con của cây đang xét.

Cấu trúc cây

Một số khái niệm cơ bản

- **Bậc của một nút :** là số cây con của nút đó
- **Bậc của một cây :** là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n-phân.
- **Nút gốc :** là nút không có nút cha.
- **Nút lá :** là nút có bậc bằng 0 .
- **Nút nhánh :** là nút có bậc khác 0 và không phải là gốc .
- **Mức của một nút :**
  - Mức (gốc (T)) = 0.
  - Gọi T_1, T_2, T_3, ... , Tn là các cây con của T_0
  - Mức (T_1) = Mức (T_2) = ... = Mức (Tn) = Mức (T_0) + 1.

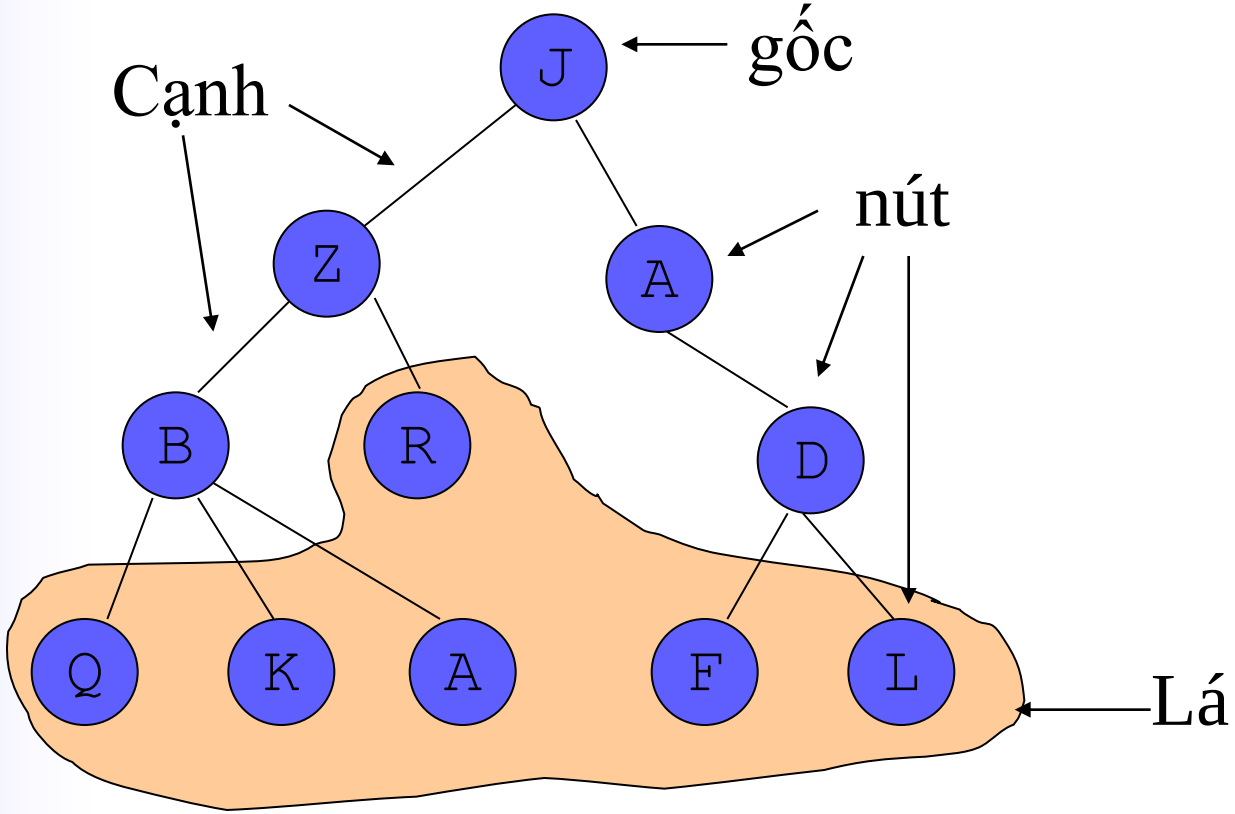
Cấu trúc cây

Một số khái niệm cơ bản

- **Độ dài đường đi từ gốc đến nút x :** là số nhánh cần đi qua kể từ gốc đến x
- **Độ dài đường đi tổng của cây :** $P_T = \sum_{X \in T} P_X$

  trong đó Px là độ dài đường đi từ gốc đến X.
- **Độ dài đường đi trung bình** : PI = PT/n (n là số nút trên cây T).
- **Rừng cây:** là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng.

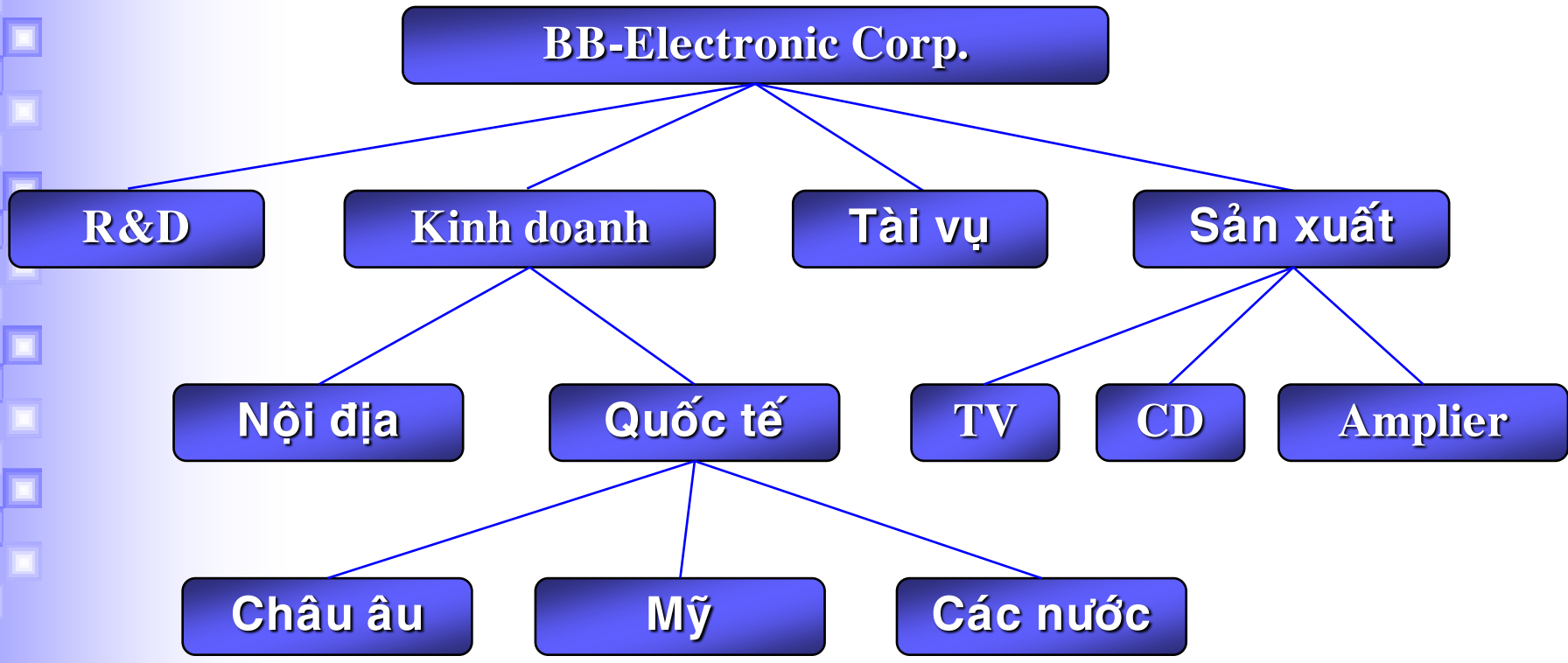
Khái niệm



Cấu trúc cây

Một số ví dụ về đối tượng các cấu trúc dạng cây

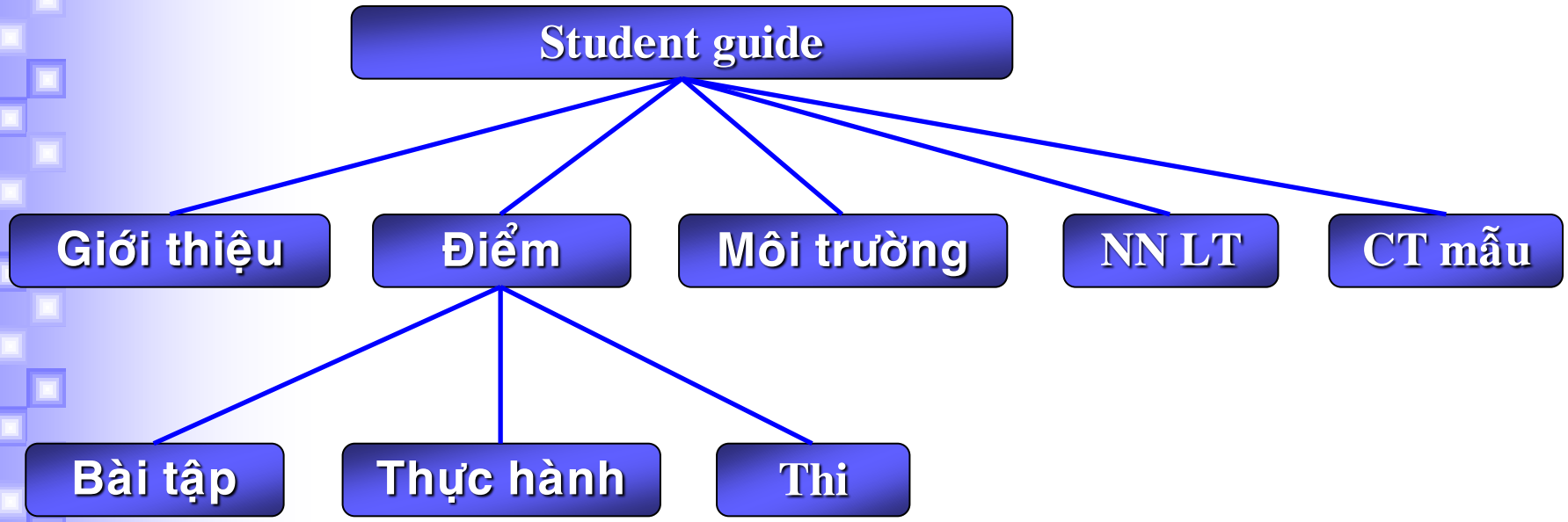
- Sơ đồ tổ chức của một công ty



Cấu trúc cây

Một số ví dụ về đối tượng các cấu trúc dạng cây

- Mục lục một quyển sách



Cấu trúc cây

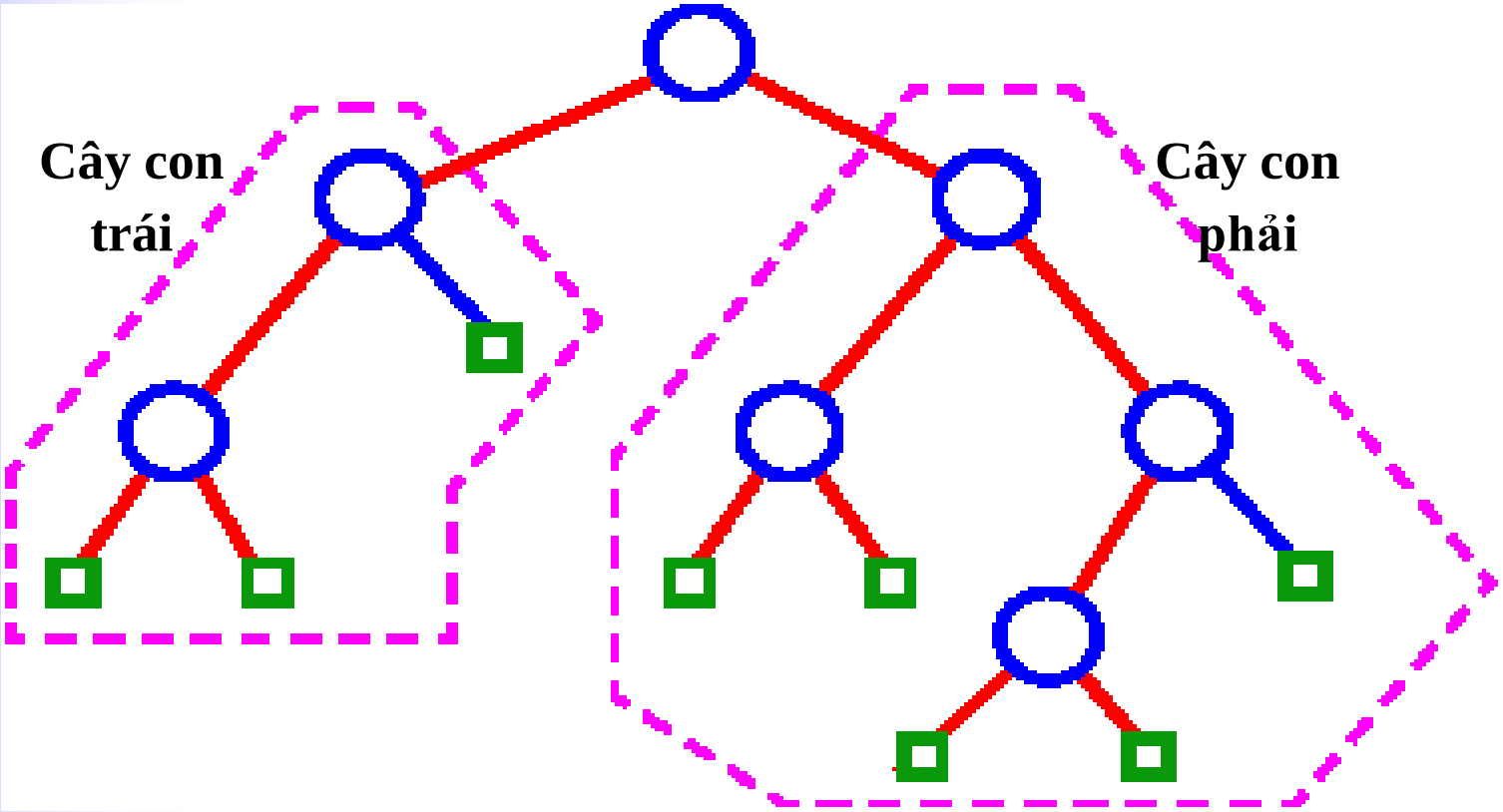
- **Nhận xét:**
  - Trong cấu trúc cây không tồn tại chu trình
  - Tổ chức 1 cấu trúc cây cho phép truy cập nhanh đến các phần tử của nó.

Cây nhị phân

Cây nhị phân

- **Định nghĩa:** Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con
- Trong thực tế thường gặp các cấu trúc có dạng cây nhị phân. Một cây tổng quát có thể biểu diễn thông qua cây nhị phân.

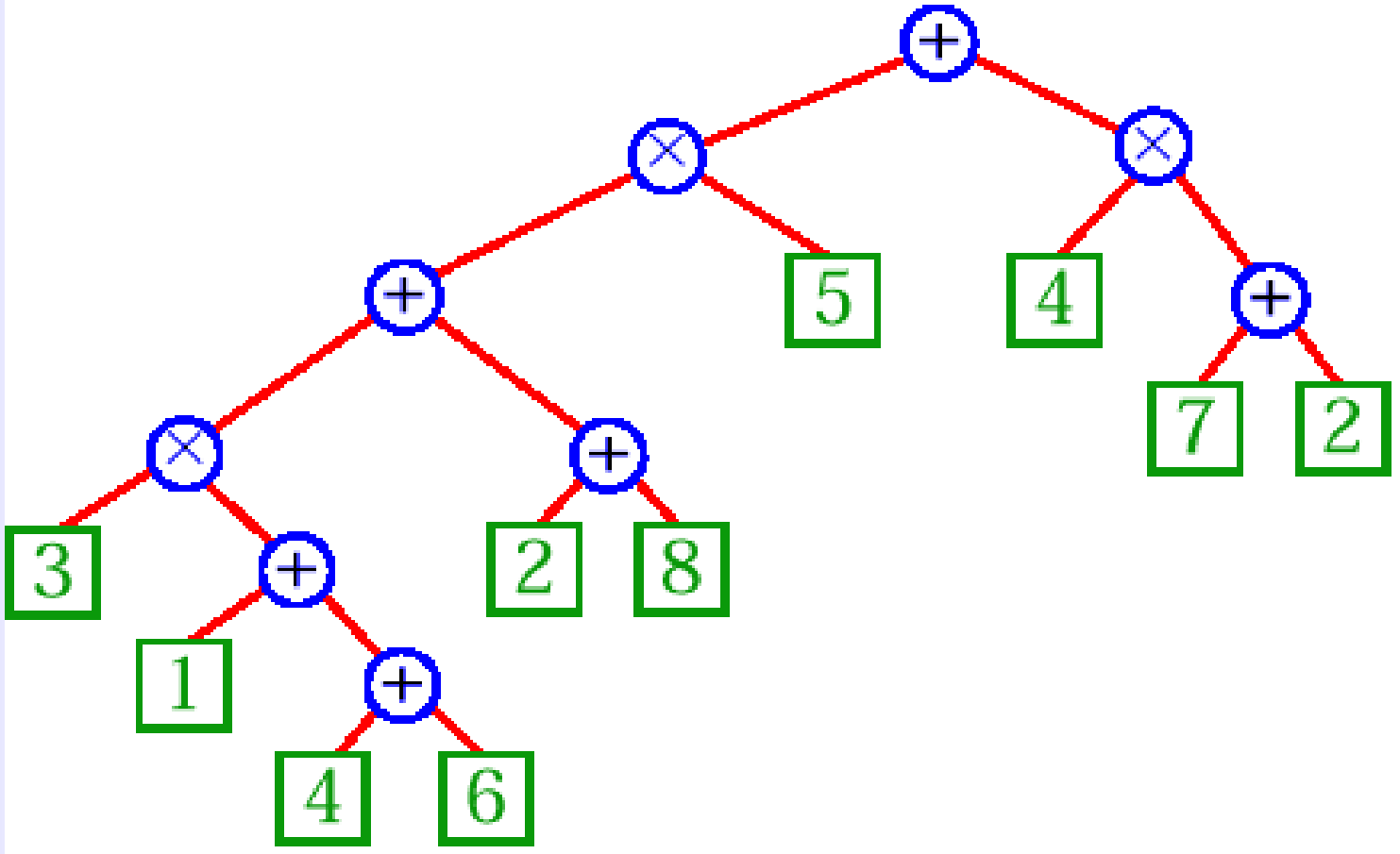
Cây nhị phân



Hình ảnh một cây nhị phân

Cây nhị phân

- Cây nhị phân dùng để biểu diễn một biểu thức toán học:

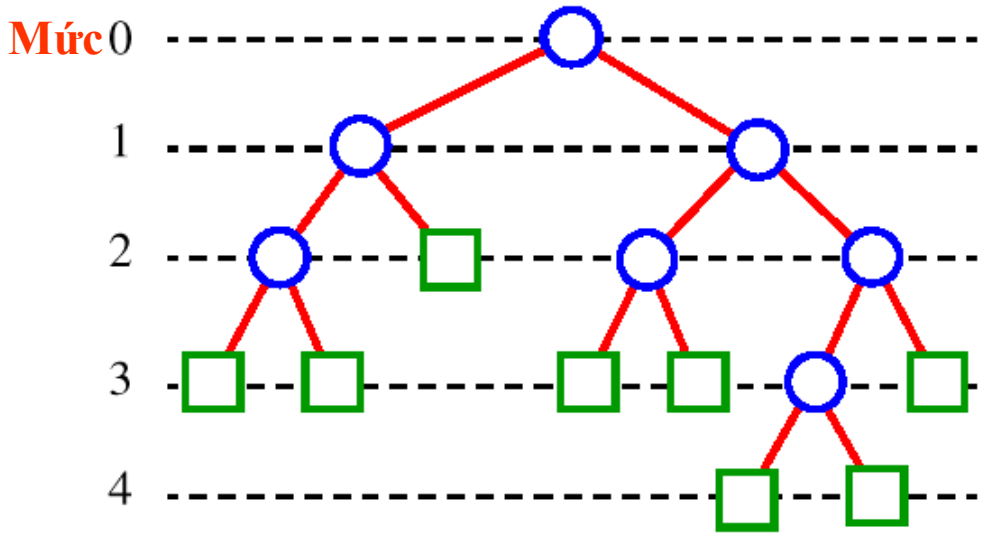


((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))

Cây nhị phân

Một số tính chất của cây nhị phân

- Số nút nằm ở mức i $\leq 2^i$
- Chiều cao cây h là mức cao nhất + 1.
- Số nút lá $\leq 2^{h-1}$, với h là chiều cao của cây.
- Chiều cao của cây h $\geq \log_2$(số nút trong cây).
- Số nút trong cây $\leq 2^h-1$.
- Đường đi (path)
  - Tên các node của quá trình đi từ node gốc theo các cây con đến một node nào đó.



Cây nhị phân

Biểu diễn cây nhị phân T

- Cây nhị phân là một cấu trúc bao gồm các phần tử (nút) được kết nối với nhau theo quan hệ "cha-con" với mỗi cha có tối đa 2 con. Để biểu diễn cây nhị phân ta chọn phương pháp cấp phát liên kết. Ứng với một nút, ta sử dụng một biến động lưu trữ các thông tin sau:
  - Thông tin lưu trữ tại nút.
  - Địa chỉ nút gốc của cây con trái trong bộ nhớ.
  - Địa chỉ nút gốc của cây con phải trong bộ nhớ.

Cây nhị phân

Để đơn giản, ta khai báo cấu trúc dữ liệu như sau :

```
typedef struct NODE
{
        int data;
        NODE* left;
        NODE* right;
};
typedef struct NODE* TREE;
TREE root;
```

Tạo cây nhị phân

```
void CreateTree(TREE &root)
{
        int x;
        printf("\nGia tri node :");
        x=toupper(getch());
        if(isspace(x)==0)
        {
                root=(node*)malloc(sizeof(node));
                root ->data=x;
                printf("\nCon trai cua %c (ENTER NULL)",x);
                CreateTree(root->left);
                printf("\nCon phai cua %c (ENTER NULL)",x);
                CreateTree(root->right);
        }
        else root=NULL;
}
```

Cây nhị phân

Duyệt cây nhị phân

- Có 3 kiểu duyệt chính có thể áp dụng trên cây nhị phân:
  - Duyệt theo thứ tự trước (**NLR**)
  - Duyệt theo thứ tự giữa (**LNR**)
  - Duyệt theo thứ tựi sau (**LRN**).
- Tên của 3 kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm 2 cây con.

Cây nhị phân

Duyệt theo thứ tự trước (Node-Left-Right)

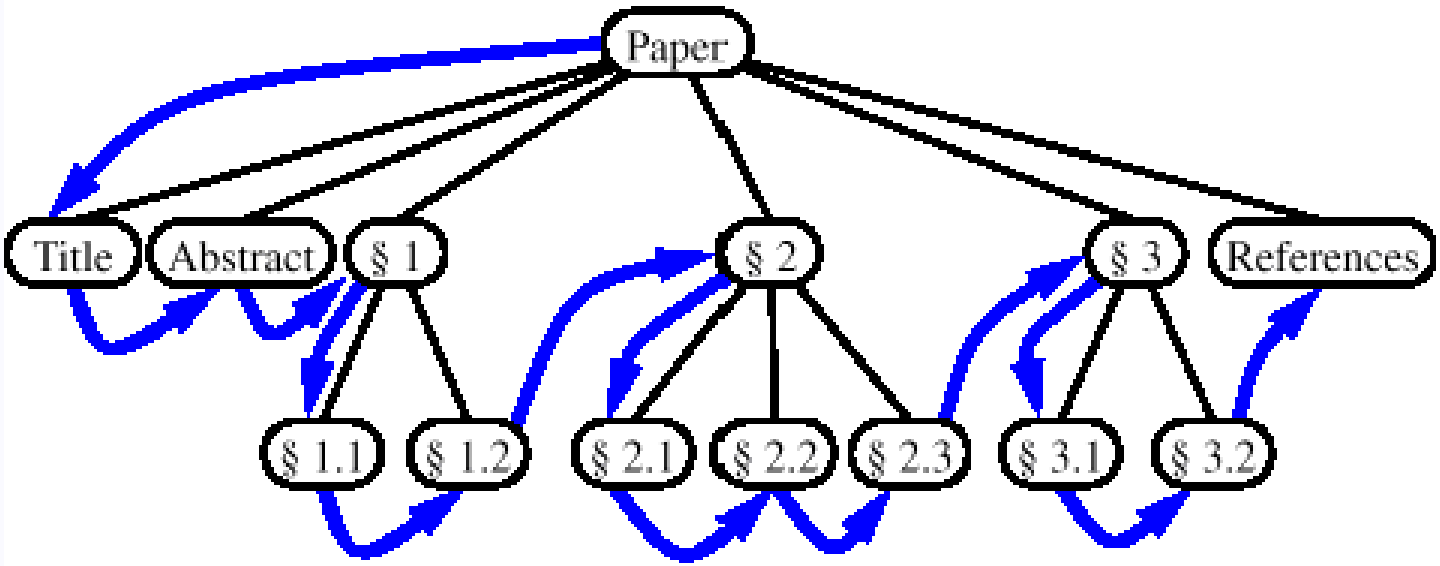
- Kiểu duyệt này trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi đến cây con phải.
- Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void    NLR(TREE root)
{
  if (Root != NULL)
  {
   <Xử lý Root>;//Xử lý tương ứng theo nhu cầu
   NLR(root->left);
   NLR(root->right);
  }
}
```

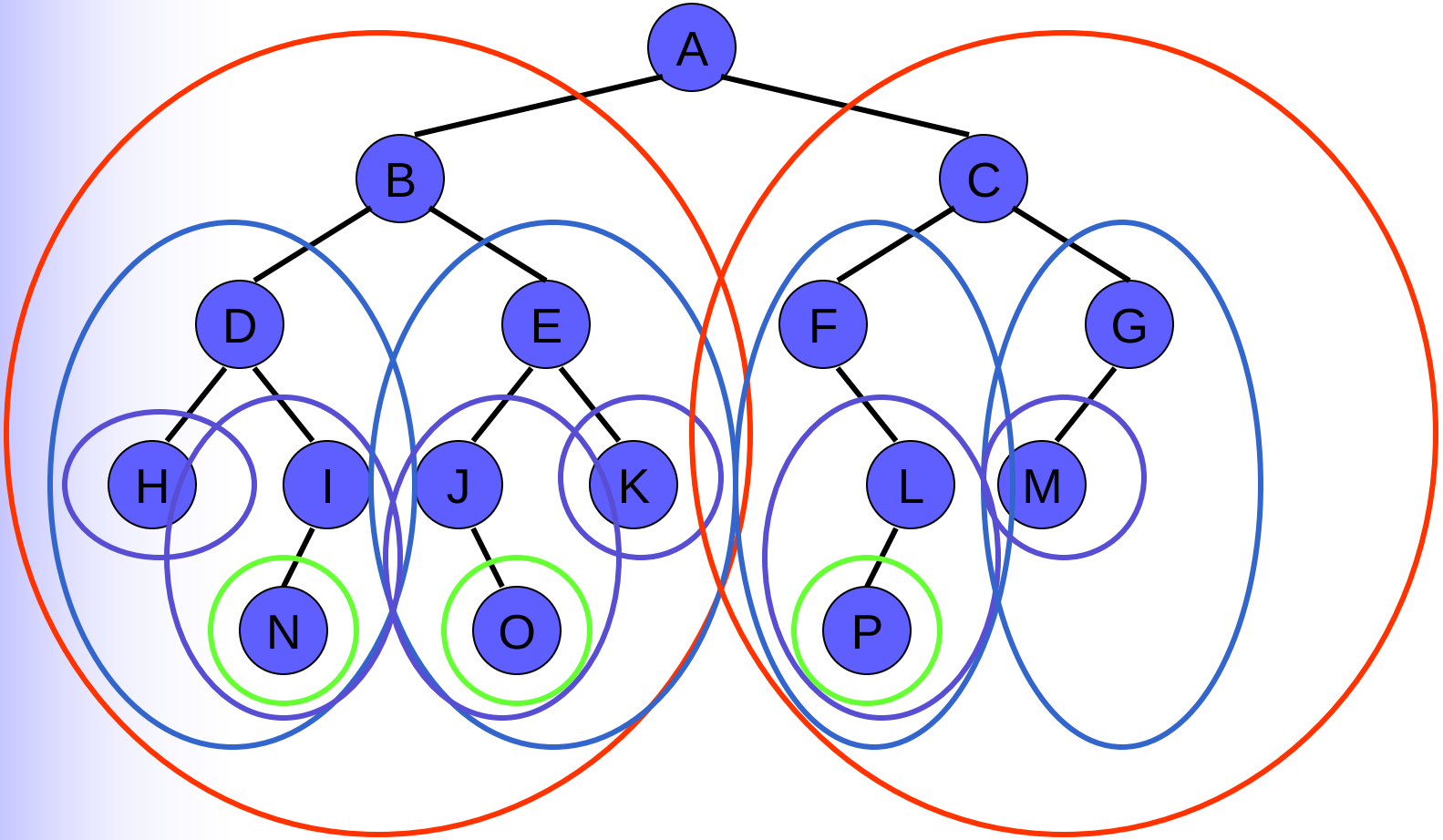
Cây nhị phân

Duyệt theo thứ tự trước (Node-Left-Right)

- Một ví dụ: đọc một quyển sách hay bài báo từ đầu đến cuối như minh họa trong hình bên dưới:



Duyệt theo thứ tự trước (Node-Left-Right)



Kết quả: A B D H I N E J O K C F L P G M

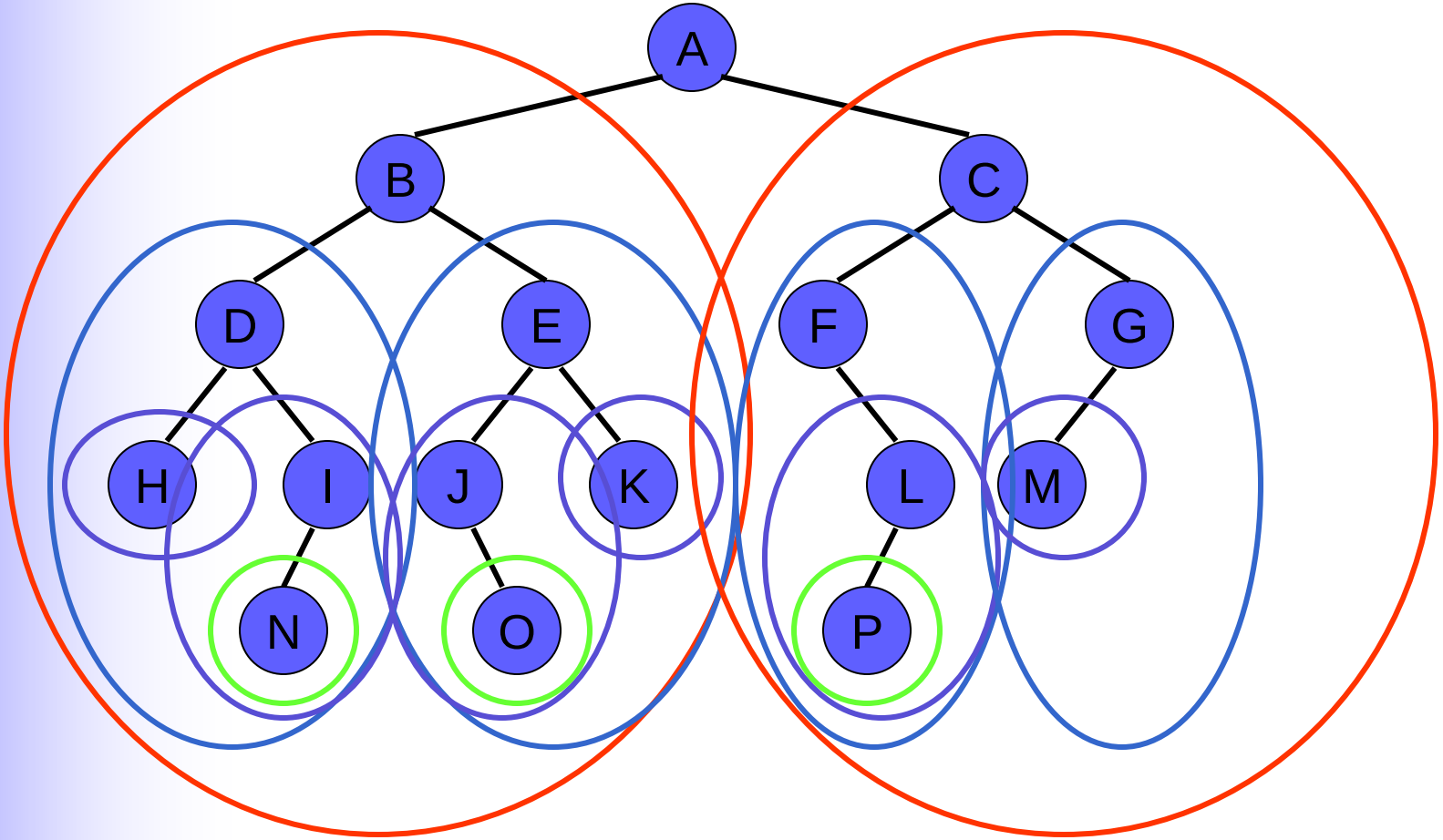
Cây nhị phân

Duyệt theo thứ tự giữa (Left- Node-Right)

- Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm nút gốc rồi đến cây con phải.
- Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void        LNR(TREE root)
{
    if (root != NULL)
    {
            LNR(root->left);
            <Xử lý Root>;     //Xử lý tương ứng theo nhu cầu
            LNR(root->right);
    }
}
```

Duyệt theo thứ tự giữa (Left- Node-Right)



Kết quả: H D N I B J O E K A F P L C M G

Cây nhị phân

Duyệt theo thứ tự sau (Left-Right-Node)

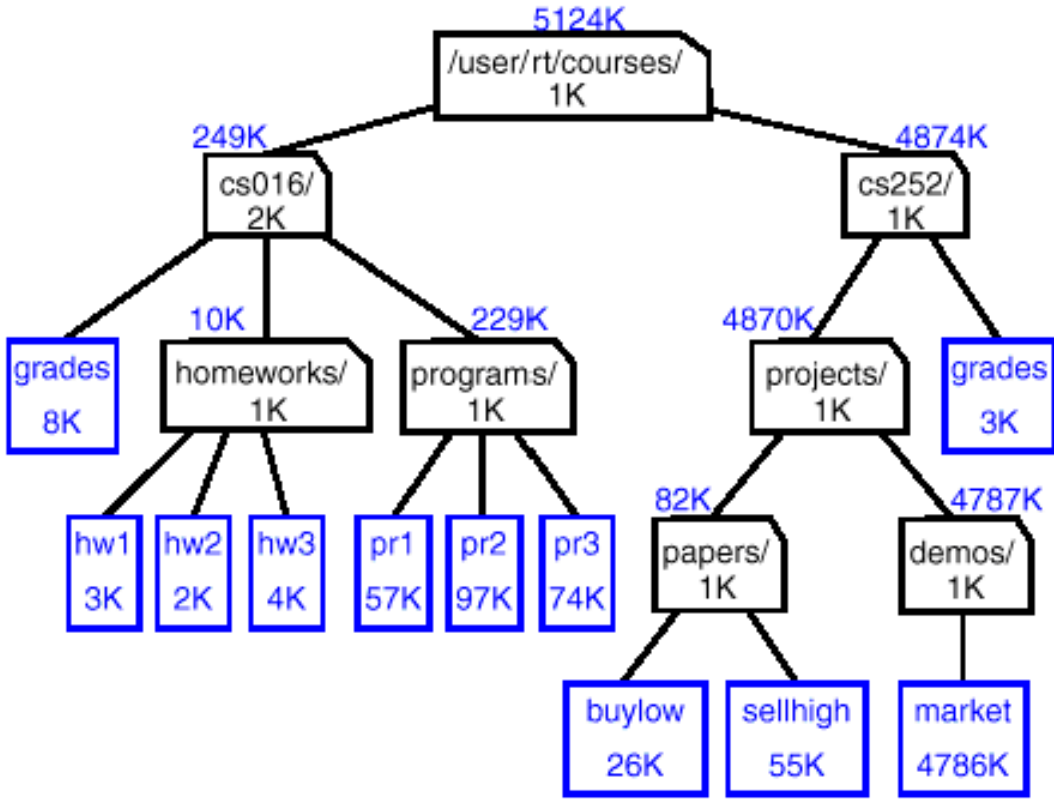
- Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm đến cây con phải rồi cuối cùng mới thăm nút gốc.
- Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void        LRN(TREE root)
{
   if (root != NULL)
   {
     LRN(root->left);
     LRN(root->right);
     <Xử lý Root>;     //Xử lý tương ứng theo nhu cầu
   }
}
```

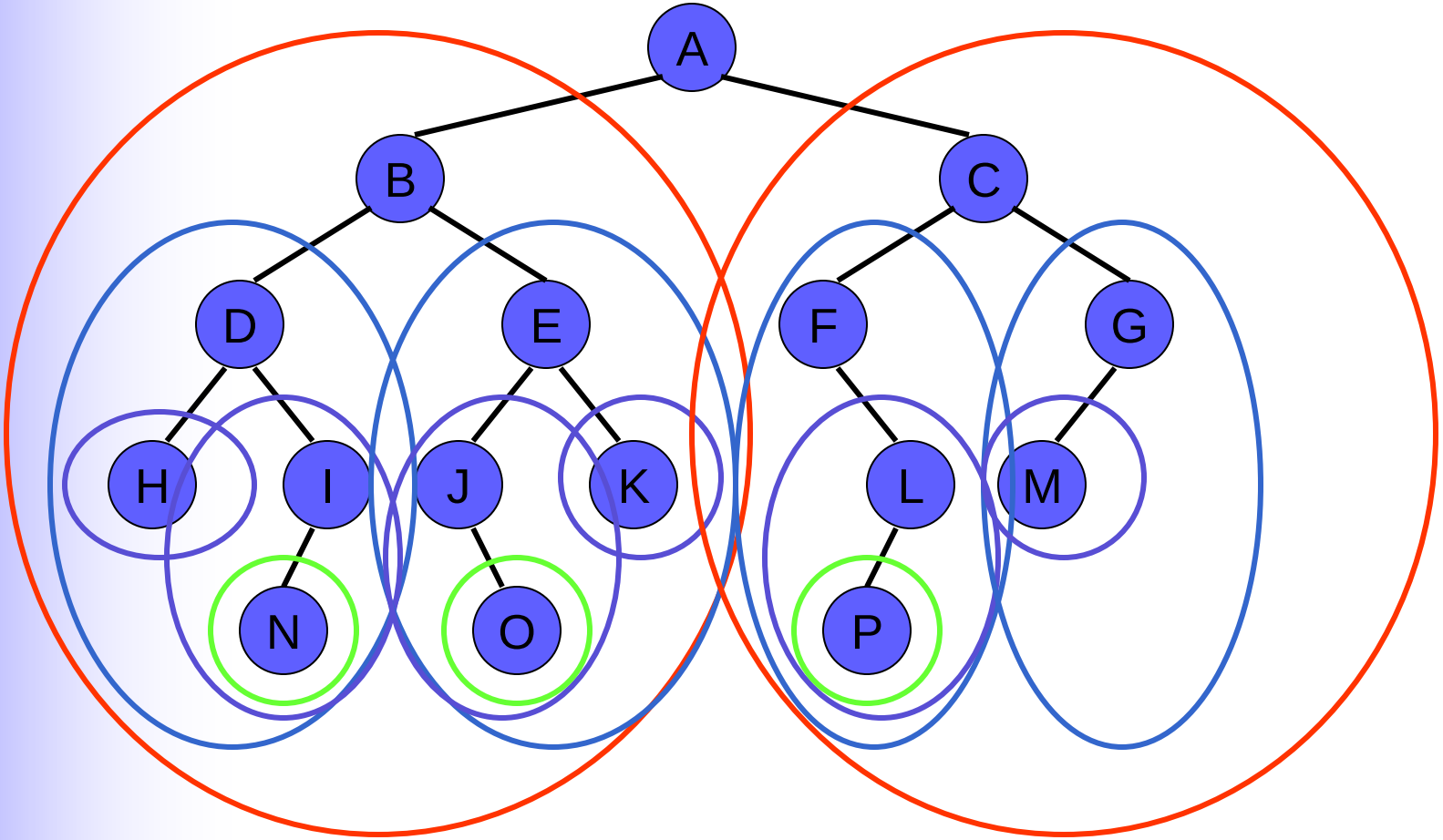
Cây nhị phân

Duyệt theo thứ tự sau (Left-Right-Node)

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



Duyệt theo thứ tự sau (Left-Right-Node)

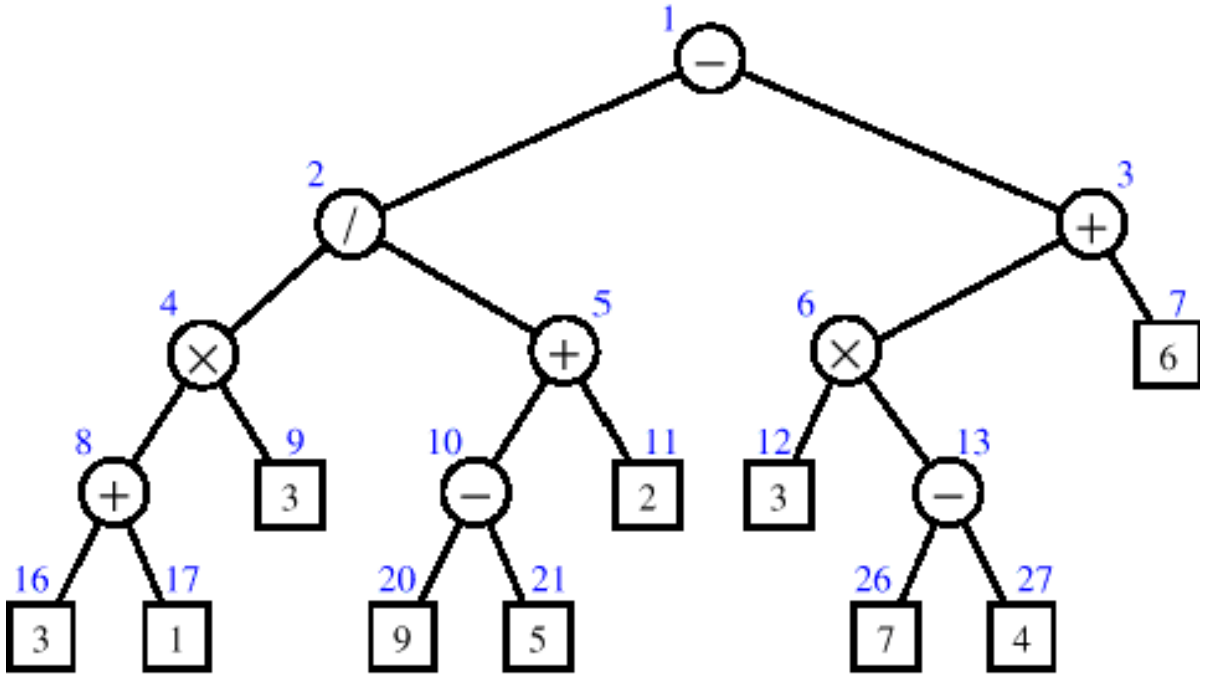


Kết quả: H N I D O J K E B P L F M G C A

Cây nhị phân

Duyệt theo thứ tự sau (Left-Right-Node)

- Tính toán giá trị của biểu thức dựa trên cây biểu thức



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

Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

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

Cây nhị phân

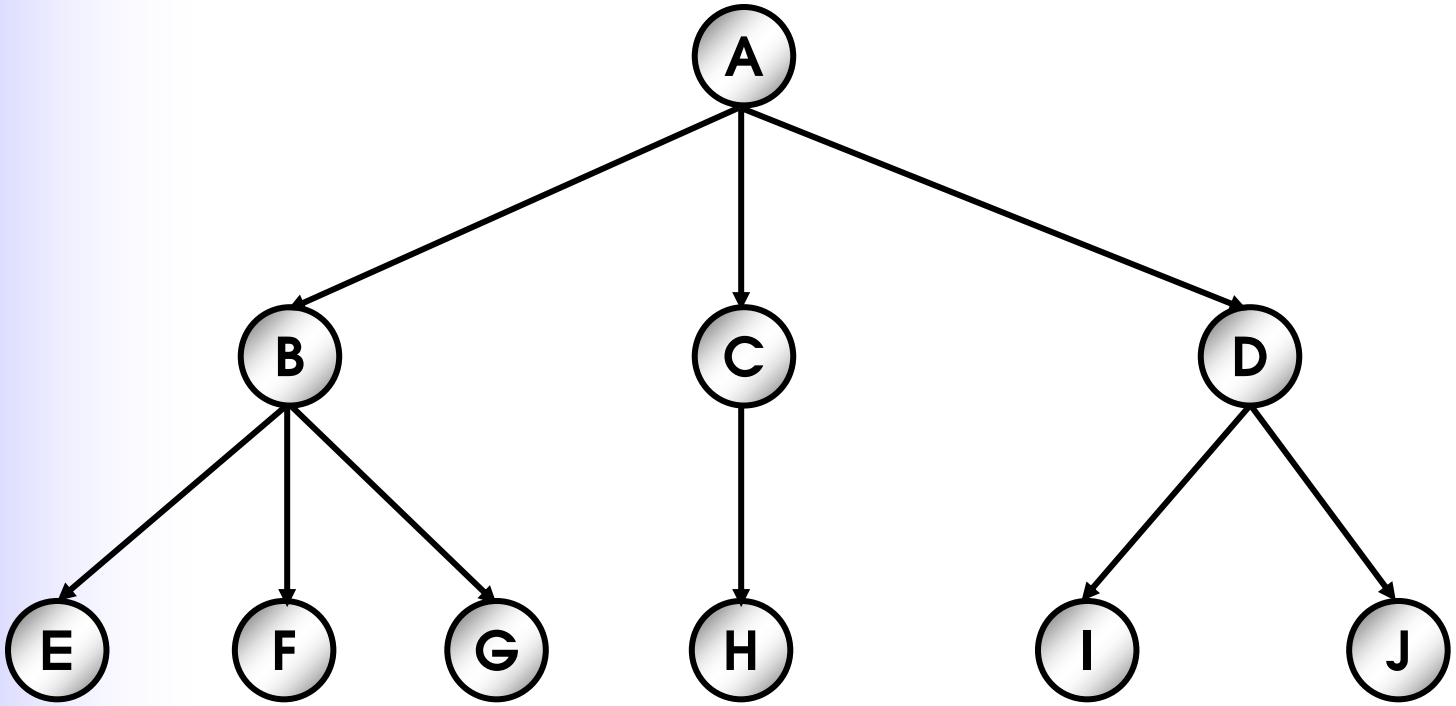
Biểu diễn cây tổng quát bằng cây nhị phân

- Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:
  - Giữ lại nút con trái nhất làm nút con trái.
  - Các nút con còn lại chuyển thành nút con phải.
  - Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.

Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

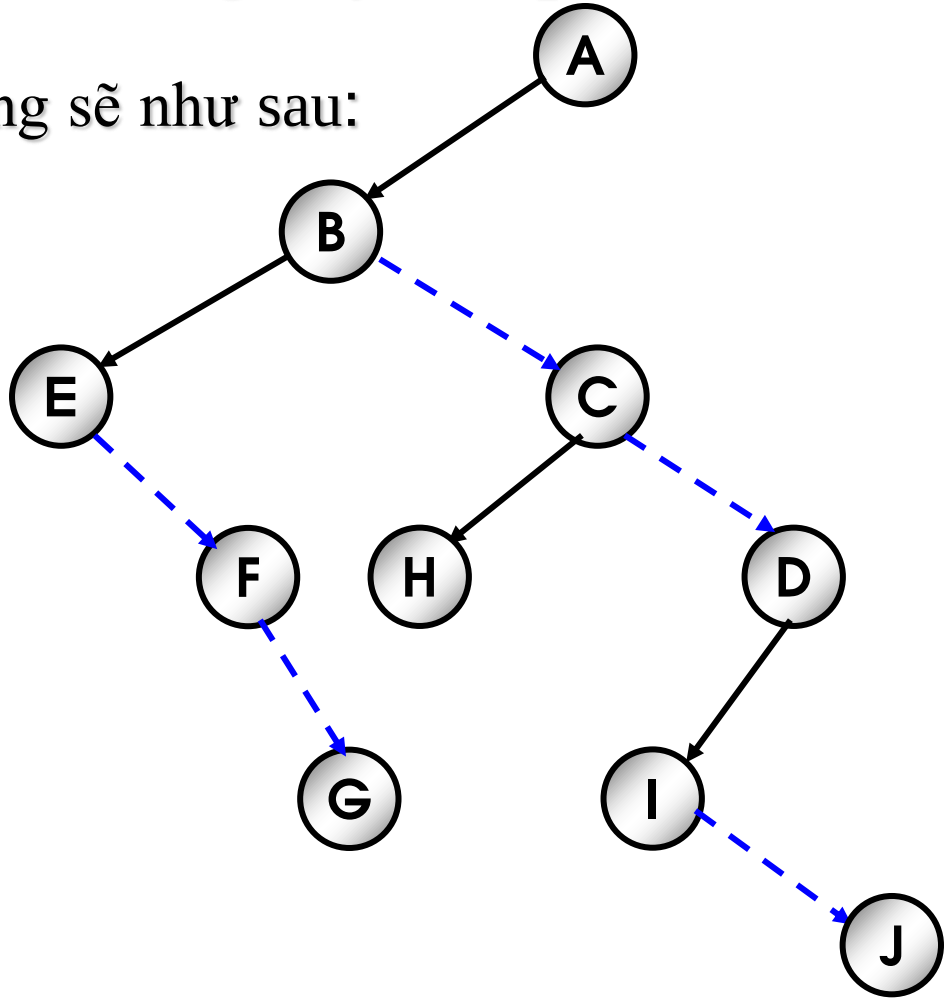
- Giả sử có cây tổng quát như hình bên dưới:



Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

- Cây nhị phân tương ứng sẽ như sau:



Cây nhị phân

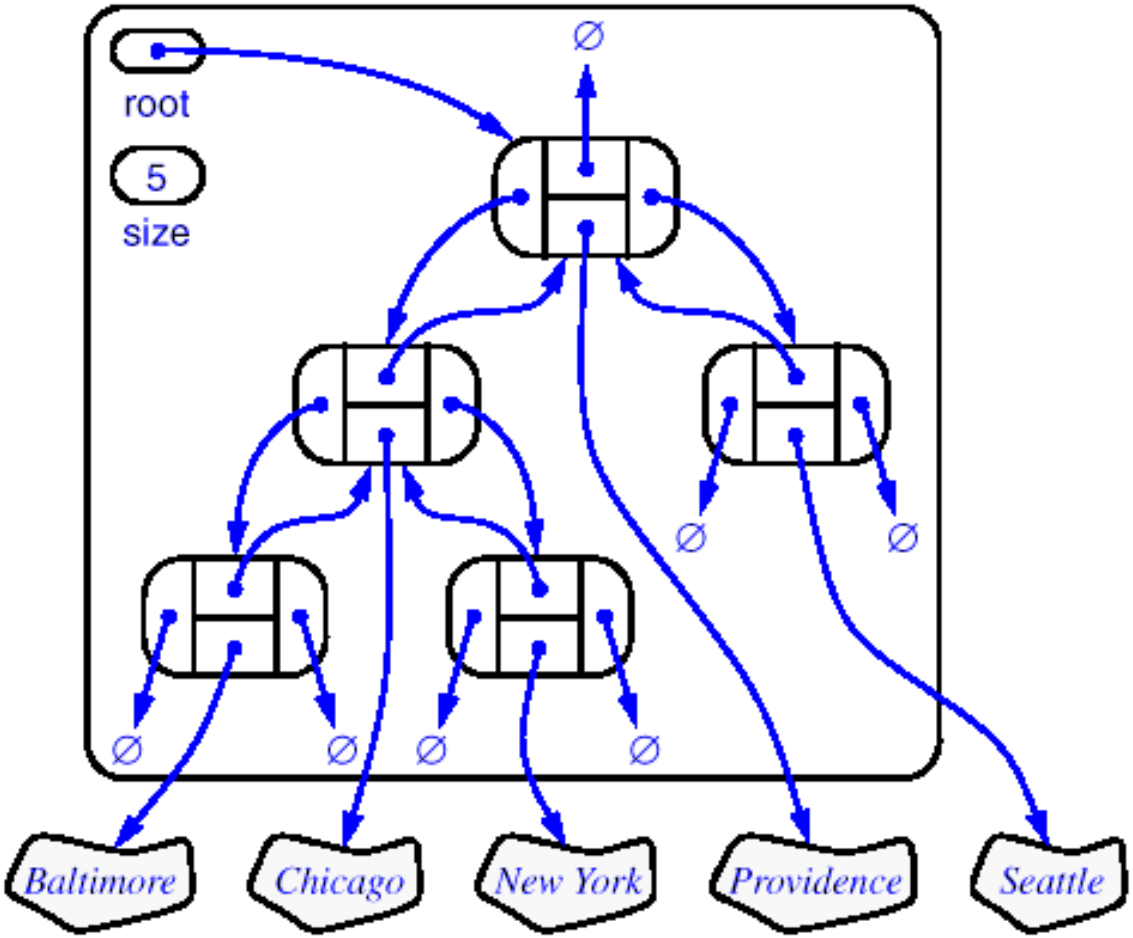
Một cách biểu diễn cây nhị phân khác

- Đôi khi, khi định nghĩa cây nhị phân, người ta quan tâm đến cả quan hệ 2 chiều cha con chứ không chỉ một chiều như định nghĩa ở phần trên.
- Lúc đó, cấu trúc cây nhị phân có thể định nghĩa lại như sau:

```
typedef struct tagTNode
{
  DataType              Key;
  struct tagTNode*      pParent;
  struct tagTNode*      pLeft;
  struct tagTNode*      pRight;
}TNODE;
typedef TNODE    *TREE;
```

Cây nhị phân

Một cách biểu diễn cây nhị phân khác



Cây nhị phân tìm kiếm

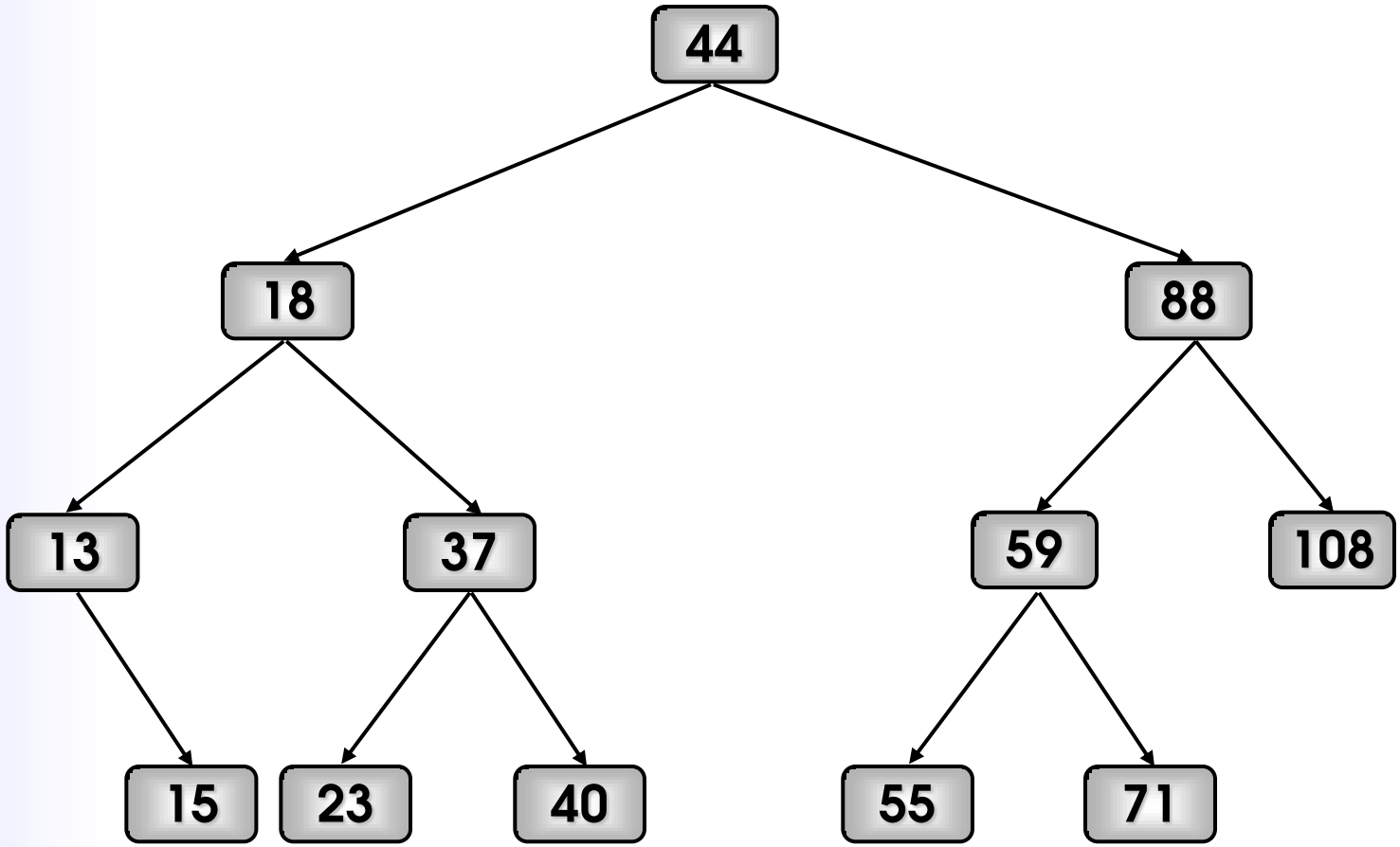
Cây nhị phân tìm kiếm

- Trong chương 3, chúng ta đã làm quen với một số cấu trúc dữ liệu động. Các cấu trúc này có sự mềm dẻo nhưng lại bị hạn chế trong việc tìm kiếm thông tin trên chúng (chỉ có thể tìm kiếm tuần tự).
- Nhu cầu tìm kiếm là rất quan trọng. Vì lý do này, người ta đã đưa ra cấu trúc cây để thỏa mãn nhu cầu trên.
- Tuy nhiên, nếu chỉ với cấu trúc cây nhị phân đã định nghĩa ở trên, việc tìm kiếm còn rất mơ hồ.
- Cần có thêm một số ràng buộc để cấu trúc cây trở nên chặt chẽ, dễ dùng hơn.
- Một cấu trúc như vậy chính là **cây nhị phân tìm kiếm.**

Cây nhị phân tìm kiếm

- Định nghĩa: cây nhị phân tìm kiếm (CNPTK) là cây nhị phân trong đó tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.
- Nếu số nút trên cây là N thì chi phí tìm kiếm trung bình chỉ khoảng $\log_2 N$.

Cây nhị phân tìm kiếm



Thêm một nút vào cây

```
int  InsertTree(tree  &root , int  x)
{
        if(root != NULL)
        {
                if(root->data==x) return 0;
                if(root->data>x) return InsertTree(root->letf,x);
                else  return InsertTree(root->right,x);
        }
        else
        {
                root=(node*)malloc(sizeof(node);
                if(root !=NULL) return -1;
                root->data=x;
                root->left=root->right=NULL;
                return 1;
        }
}
```

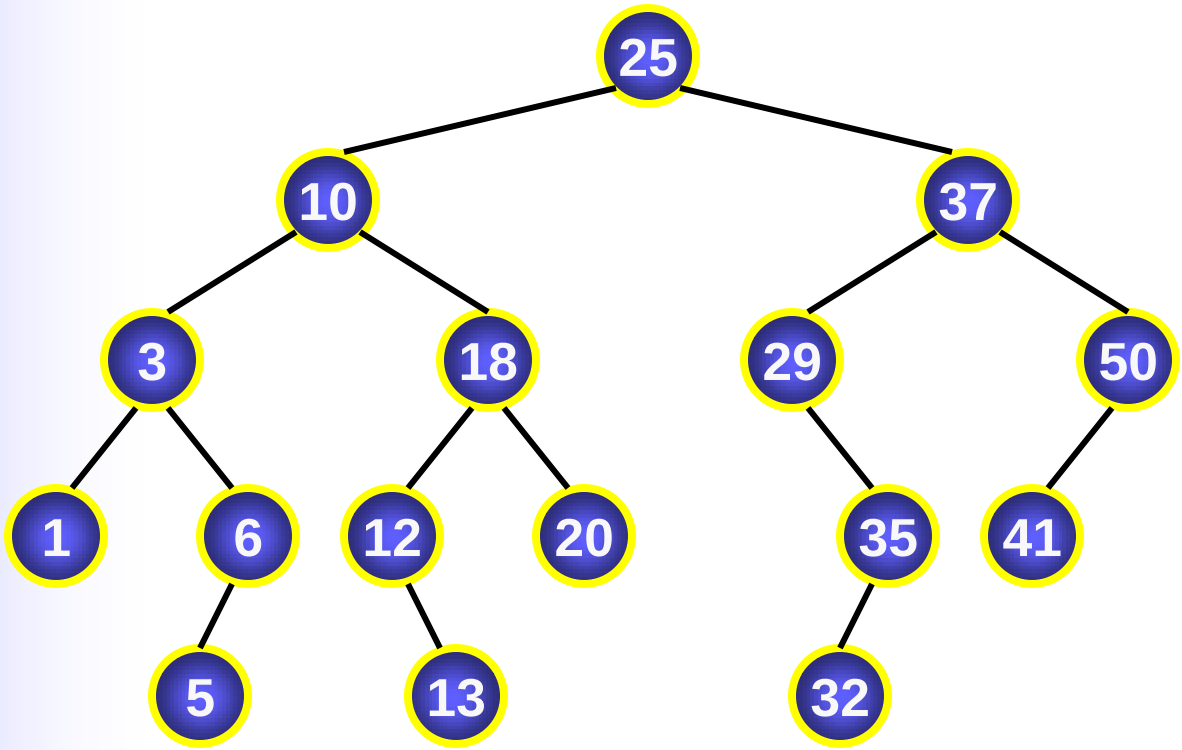
Tạo cây nhị phân tìm kiếm

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

```
void CreateTree(tree &root)
{
        int x,n;
        printf("Nhap n = "); scanf("%d",&n);
        for(int i=1; i<=n;i++)
        {
                scanf("%d",&x);
                InsertTree(root,x);
        }
}
```

Tạo cây nhị phân tìm kiếm

25 37 10 18 29 50 3 1 6 5 12 20 35 13 32 41



25	37	10	18	29	50	3	1	6	5	12	20	35	13	32	41

Duyệt cây nhị phân tìm kiếm

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

Tìm một phần tử x trong cây (đệ quy)

- Tìm một phần tử x trong cây (đệ quy):

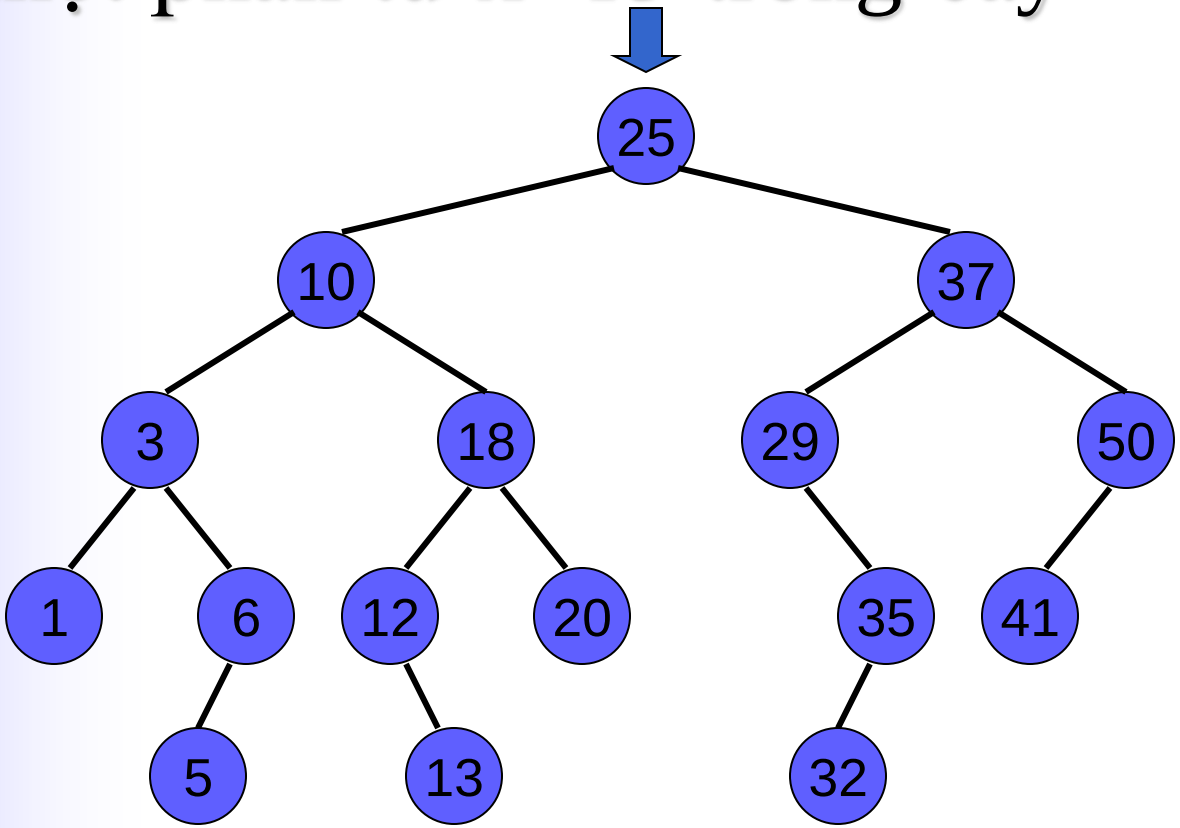
```
TNODE* searchNode(TREE root, Data X)
{
   if(root)
   {
        if(root->data == X)
            return root;
        if(root->data > X)
            return searchNode(root->left, X);
        return searchNode(root->right, X);
   }
   return NULL;
}
```

Tìm một phần tử x trong cây (không đệ quy)

- Tìm một phần tử x trong cây (không đệ quy):

```
TNODE * searchNode(TREE root,  Data x)
{
    TNODE *p = root;
    while (p != NULL)
    {
        if(x == p->data)   return p;
        else
            if(x < p->data) p = p->left;
            else   p = p->right;
    }
    return NULL;
}
```

Tìm một phần tử x=13 trong cây

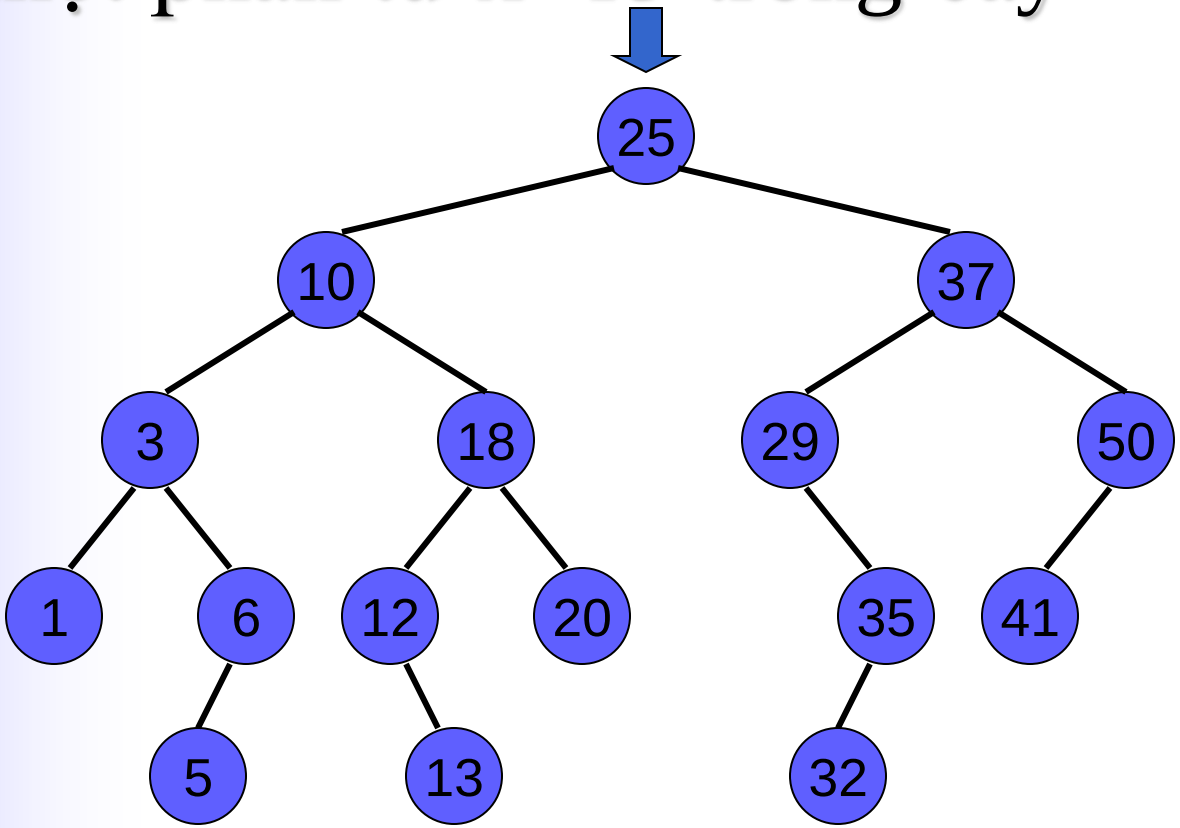


Tìm kiếm 13

Tìm thấy

Số node duyệt: 5
Số lần so sánh: 9

Tìm một phần tử x=13 trong cây



Tìm kiếm 13

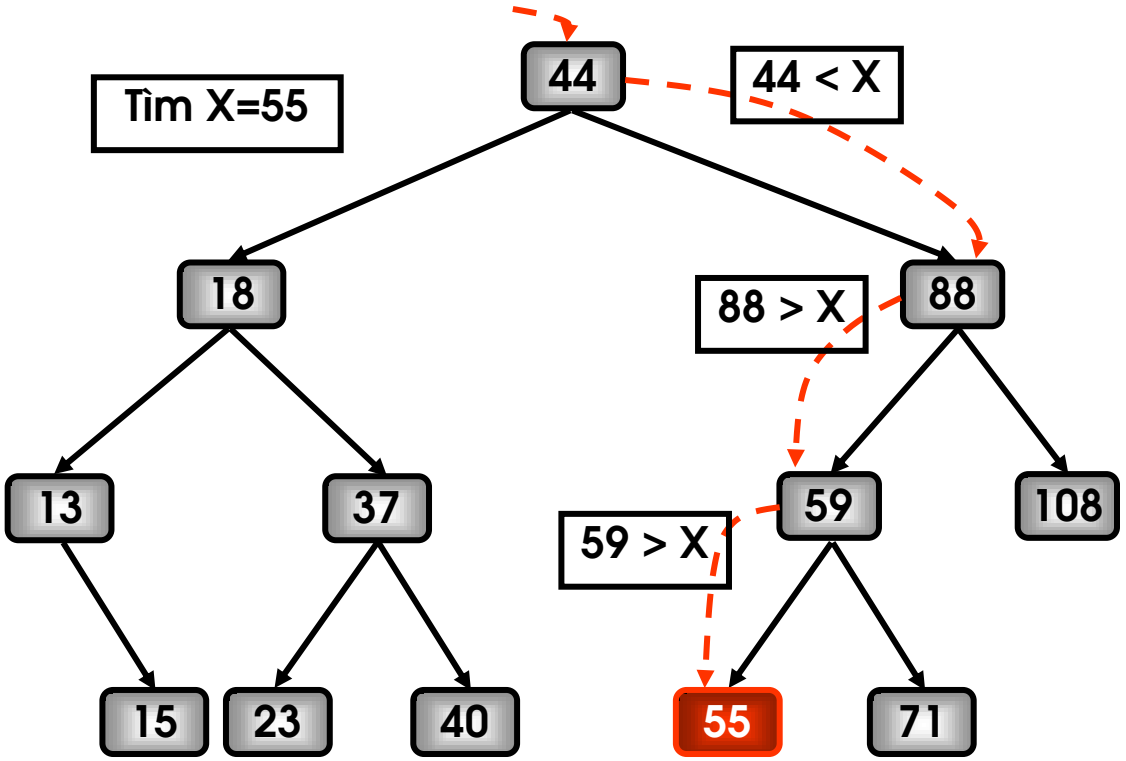
Tìm thấy

Số node duyệt: 5
Số lần so sánh: 9

Tìm một phần tử x trong cây

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

Tìm một phần tử x trong cây



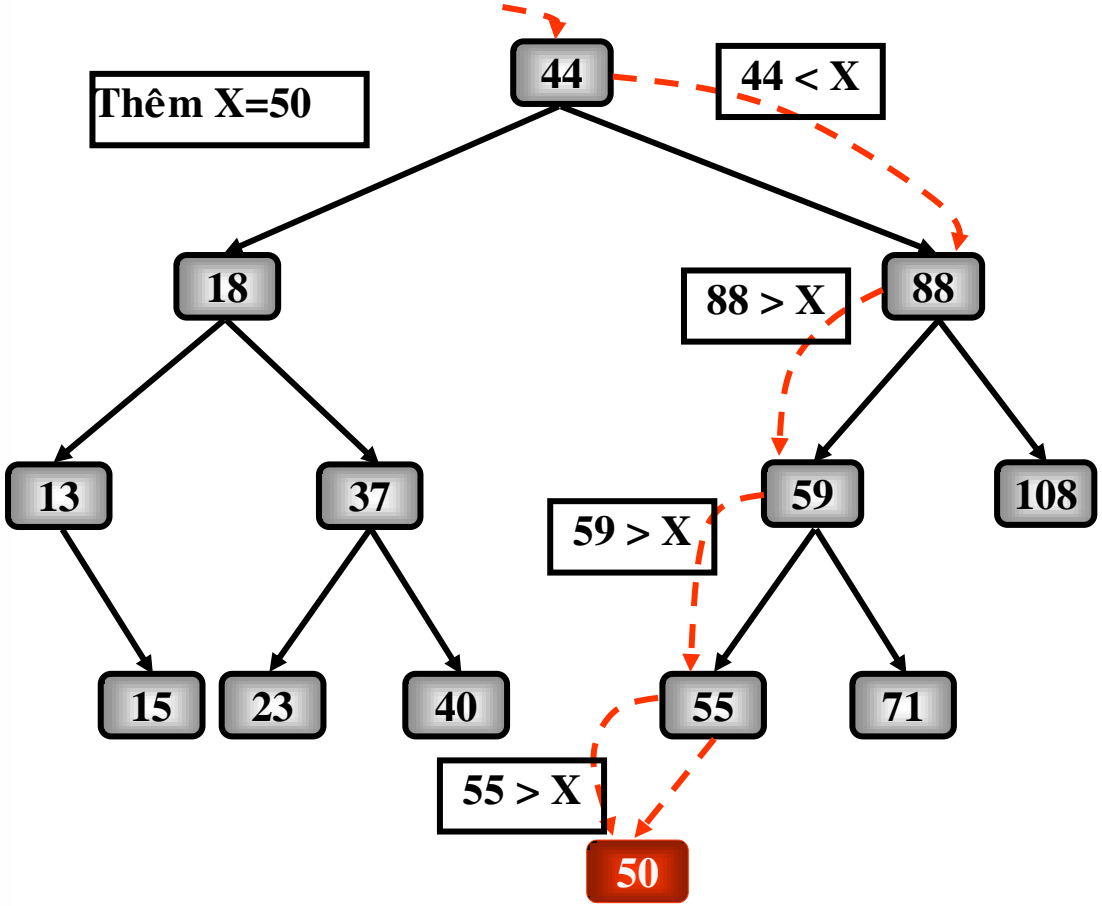
Thêm một phần tử x vào cây

- Thêm một phần tử x vào cây:
  - Việc thêm một phần tử X vào cây phải bảo đảm điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút ngoài sẽ là tiện lợi nhất do ta có thể thực hiên quá trình tương tự thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.
  - Hàm insert trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công:

Thêm một phần tử x vào cây

```
int insertNode(TREE &root, Data X)
{ if (root) {
    if(root->data == X) return 0; // đã có
    if(root->data >  X)
          return insertNode(root->left, X);
    else
          return insertNode(root->right, X);
  }
  root    = new Node;
  if (root == NULL) return -1; // thiếu bộ nhớ
  root->data   = X;
  root->left = root->right = NULL;
  return 1;   // thêm vào thành công
}
```

Thêm một phần tử x vào cây

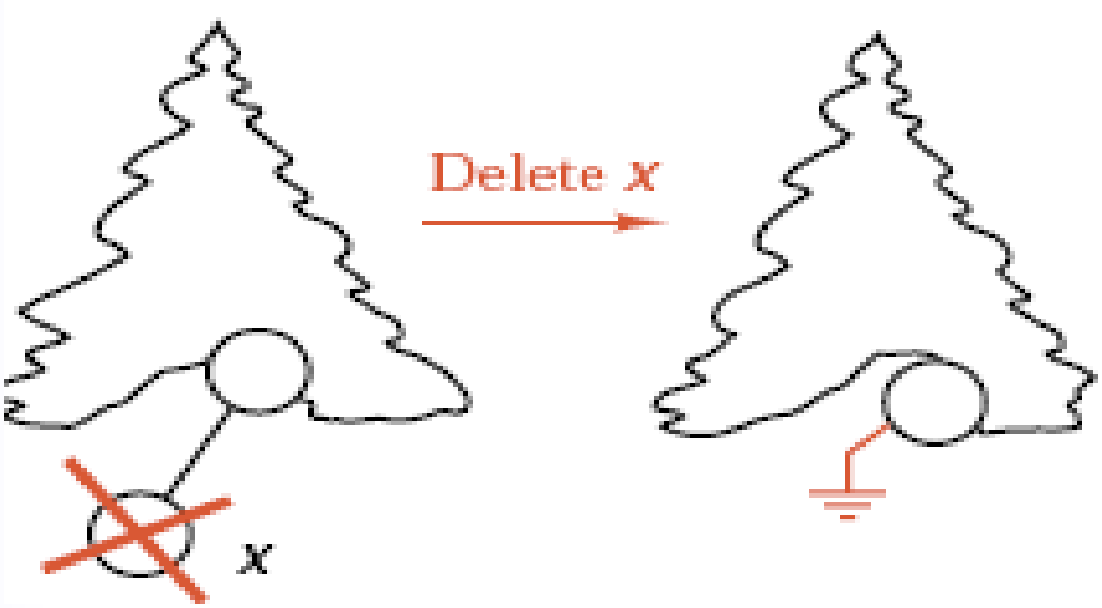


Hủy một phần tử có khóa x

- Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của CNPTK.
- Có 3 trường hợp khi hủy nút X có thể xảy ra:
  - X là nút lá.
  - X chỉ có 1 con (trái hoặc phải).
  - X có đủ cả 2 con

Hủy một phần tử có khóa x

Trường hợp 1: X là nút lá.

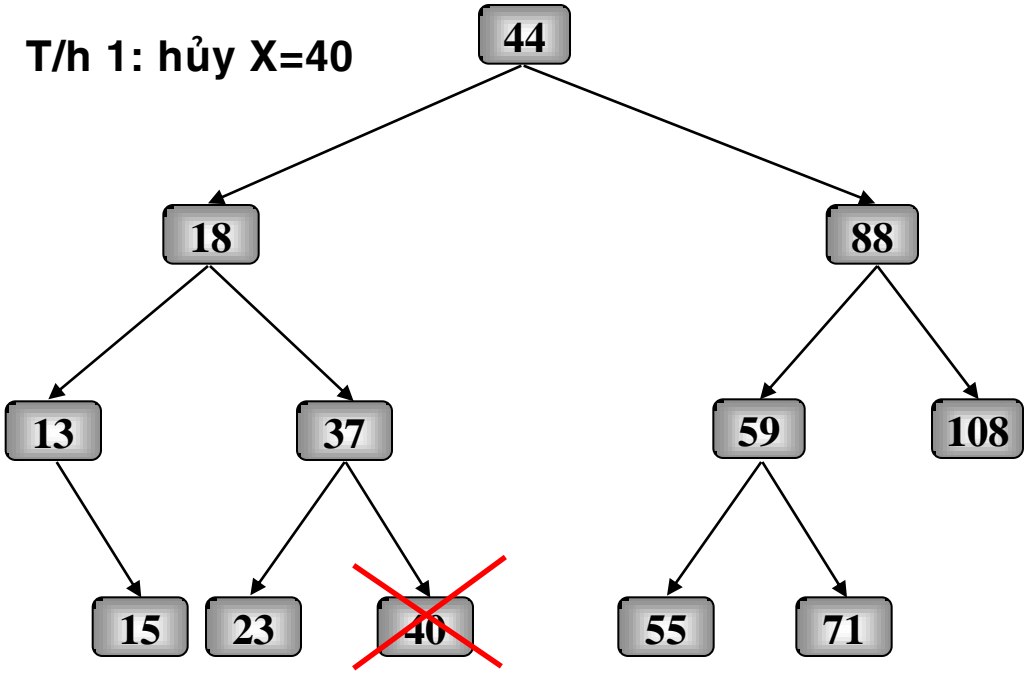


1. Xóa node này
2. Gán liên kết từ cha của nó thành rỗng

Hủy một phần tử có khóa x

Trường hợp 1: X là nút lá.

- **Ví dụ :** chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác.



Hủy một phần tử có khóa x

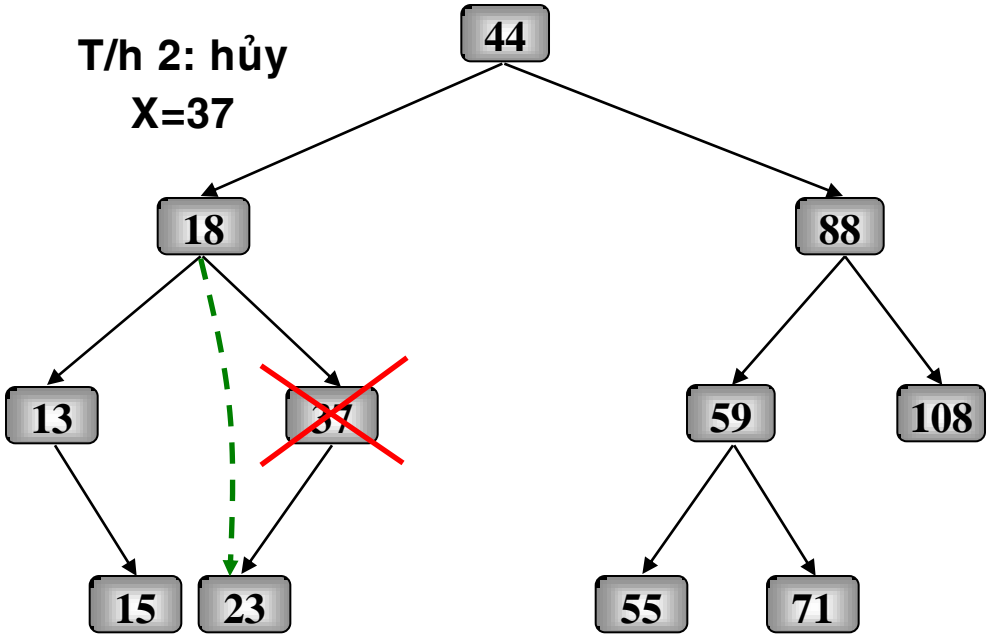
Trường hợp 2: X chỉ có 1 con (trái hoặc phải)

1. **Gán liên kết từ cha của nó xuống con duy nhất của nó**
2. **Xóa node này**

Hủy một phần tử có khóa x

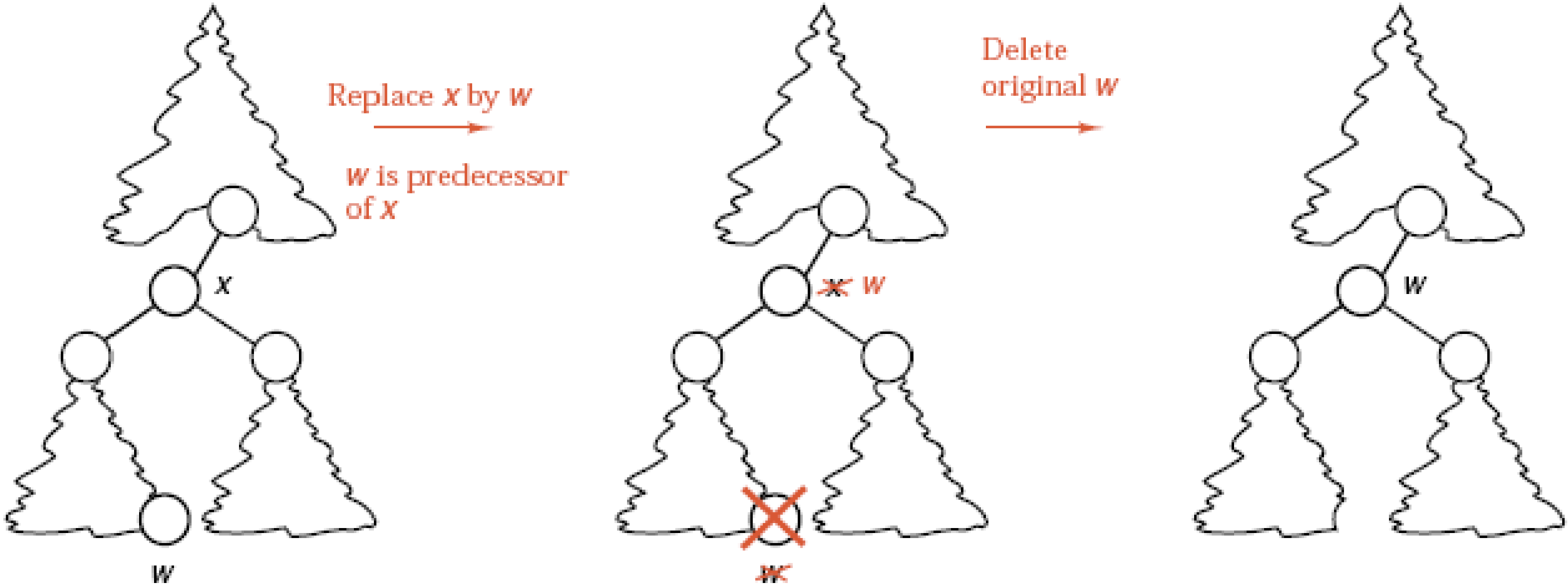
Trường hợp 2: X chỉ có 1 con (trái hoặc phải)

- **Trường hợp thứ hai:** trước khi hủy X ta móc nối cha của X với con duy nhất của nó



Hủy một phần tử có khóa x

Trường hợp 3: X có đủ 2 con



1. **Tìm w là node trước node x trên phép duyệt cây inorder (chính là node cực phải của cây con bên trái của x)**
2. **Thay x bằng w**
3. **Xóa node w cũ (giống trường hợp 1 hoặc 2 đã xét)**

Hủy một phần tử có khóa x

Trường hợp 3: X có đủ 2 con

- Trường hợp cuối cùng:
  - Không thể hủy trực tiếp do X có đủ 2 con
  - Hủy gián tiếp:
    - Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y. Phần tử này có tối đa một con.
    - Thông tin lưu tại Y sẽ được chuyển lên lưu tại X.
    - Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu.
  - Vấn đề: chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là CNPTK.

Hủy một phần tử có khóa x

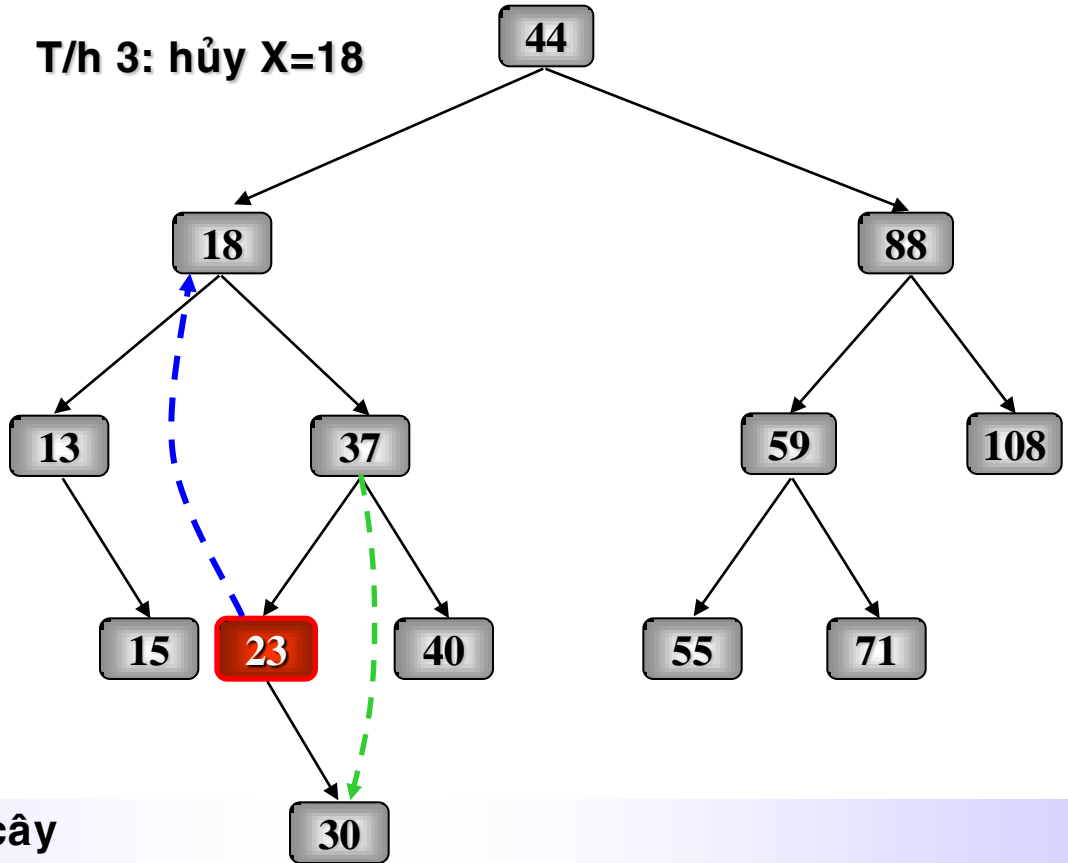
Trường hợp 3: X có đủ 2 con

- Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là CNPTK.
- Có 2 phần tử thỏa mãn yêu cầu:
  - Phần tử nhỏ nhất (trái nhất) trên cây con phải.
  - Phần tử lớn nhất (phải nhất) trên cây con trái.
- Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình.
- Ở đây, ta sẽ chọn phần tử phải nhất trên cây con trái làm phân tử thế mạng.

Hủy một phần tử có khóa x

Trường hợp 3: X có đủ 2 con

- Khi hủy phần tử X=18 ra khỏi cây, phần tử 23 là phần tử thế mạng:



Hủy một phần tử có khóa x

Trường hợp 3: X có đủ 2 con

- Hàm **delNode** trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây:

  int delNode(TREE &root, Data X)

- Hàm **searchStandFor** tìm phần tử thế mạng cho nút p

  void searchStandFor(TREE &p, TREE &q)

Hủy một phần tử có khóa x

```
int delNode(TREE &root, Data X)
{
    if(root== NULL)  return 0;
    if(root->data > X)      return delNode(root->left, X);
    if(root->data < X)      return delNode(root->right, X);
    //T->Key == X
    Node* p = root;
    if(root->left == NULL)
        root = root->right;
    else
          if(root->right == NULL)
              root = root->left;
          else // T cĩ dủ 2 con
              searchStandFor(p, root->right);
    delete p;
}
```

Hủy một phần tử có khóa x

```
void searchStandFor(TREE &p, TREE &q)
{
   if(q->left)
        searchStandFor(p, q->left);
   else
   {
      p->data = q->data;
      p  = q;
      q  = q->right;
   }
}
```

Hủy toàn bộ cây nhị phân tìm kiếm

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

```
void removeTree(TREE &root)
{
  if(root)      {
     removeTree(root->reft);
     removeTree(root->right);
     delete(root);
  }
}
```

Cây nhị phân tìm kiếm

- **Nhận xét:**
  - Tất cả các thao tác searchNode, insertNode, delNode đều có độ phức tạp trung bình O(h), với h là chiều cao của cây
  - Trong trong trường hợp tốt nhất, CNPTK có n nút sẽ có độ cao h = $\log_2(n)$. Chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.
  - Trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 danh sách liên kết (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp O(n).
  - Vì vậy cần có cải tiến cấu trúc của CNPTK để đạt được chi phí cho các thao tác là $\log_2(n)$.