TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms Basic Lab

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Basic data types, I/O with files

Chapter 2. Recursion

Chapter 3. Lists

Chapter 4. Stack and Queue

Chapter 5. Trees

Chapter 6. Sorting

Chapter 7. Searching

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 5. Trees

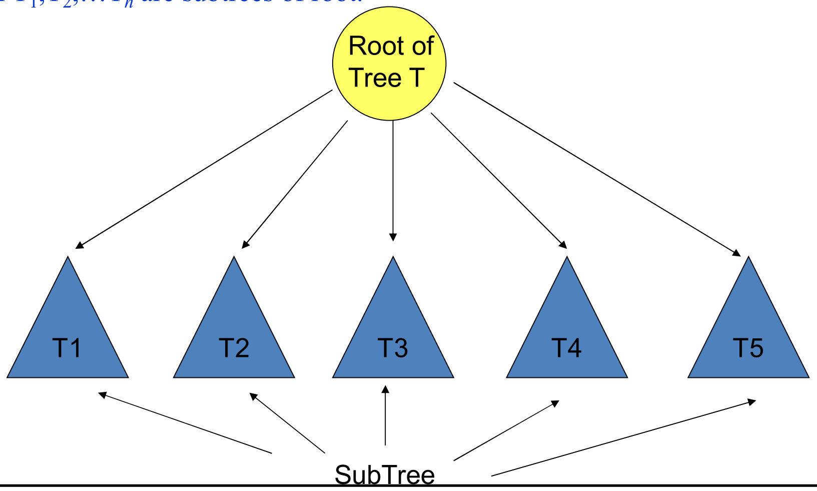
Nguyễn Khánh Phương

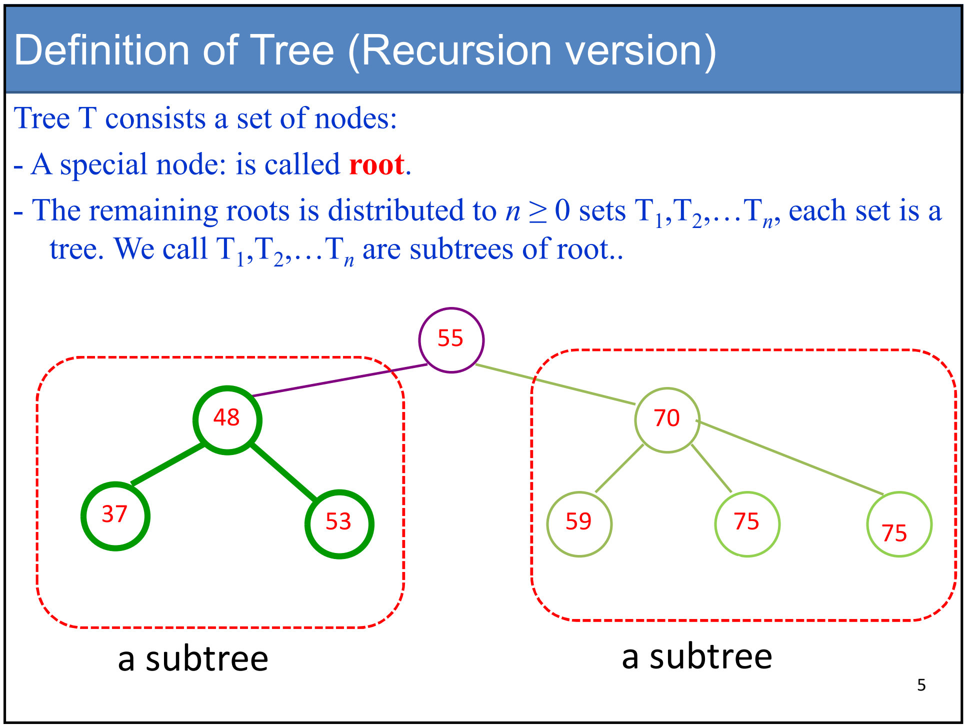
Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Definition of Tree (Recursion version)

Tree T consists a set of nodes:

- A special node: is called **root**.
- The remaining roots is distributed to $n \geq 0$ sets $T_1, T_2, \ldots T_n$, each set is a tree. We call $T_1, T_2, \ldots T_n$ are subtrees of root.



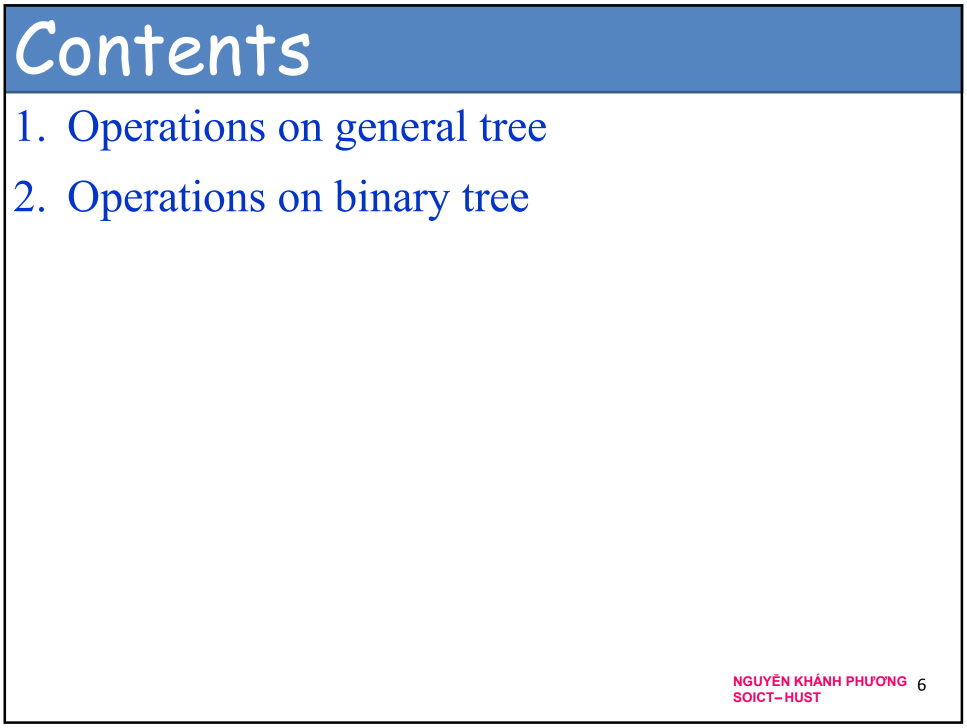


Definition of Tree (Recursion version)

Tree T consists a set of nodes:

- A special node: is called **root**.

- The remaining roots is distributed to $n \geq 0$ sets $T_1, T_2, \ldots T_n$, each set is a tree. We call $T_1, T_2, \ldots T_n$ are subtrees of root..



Contents

1. Operations on general tree
2. Operations on binary tree

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

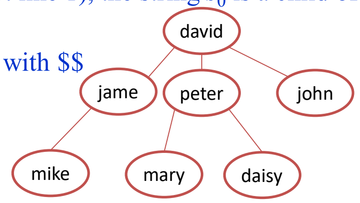
Contents

1. **Operations on general tree**
2. Operations on binary tree

Declaration general tree

- The data of a tree is stored in an external text file with the format:
  - Each line contains a sequence of strings s_0, s_1, …, s_k terminated by $, and s_1, s_2, …, s_k are children of s_0 from left to right (s_1 is the left-most child).

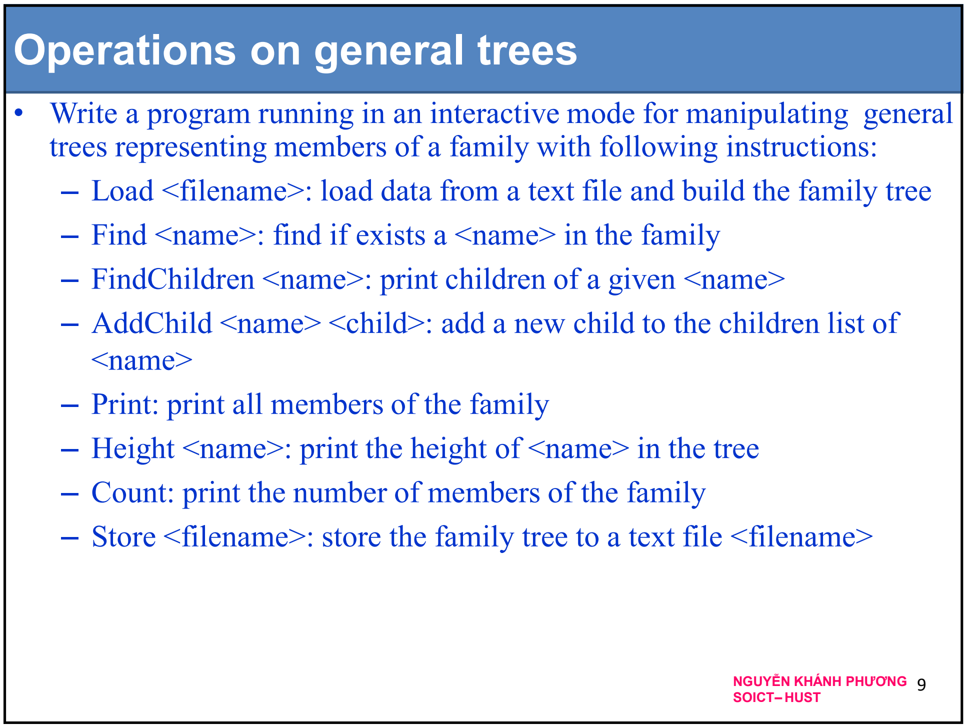
  Note: in each line (except line 1), the string s_0 is a child of some node appearing in previous lines).
  - The file is terminated with $$




```
david jame peter john $
peter mary daisy $
jame mike $
$$
```

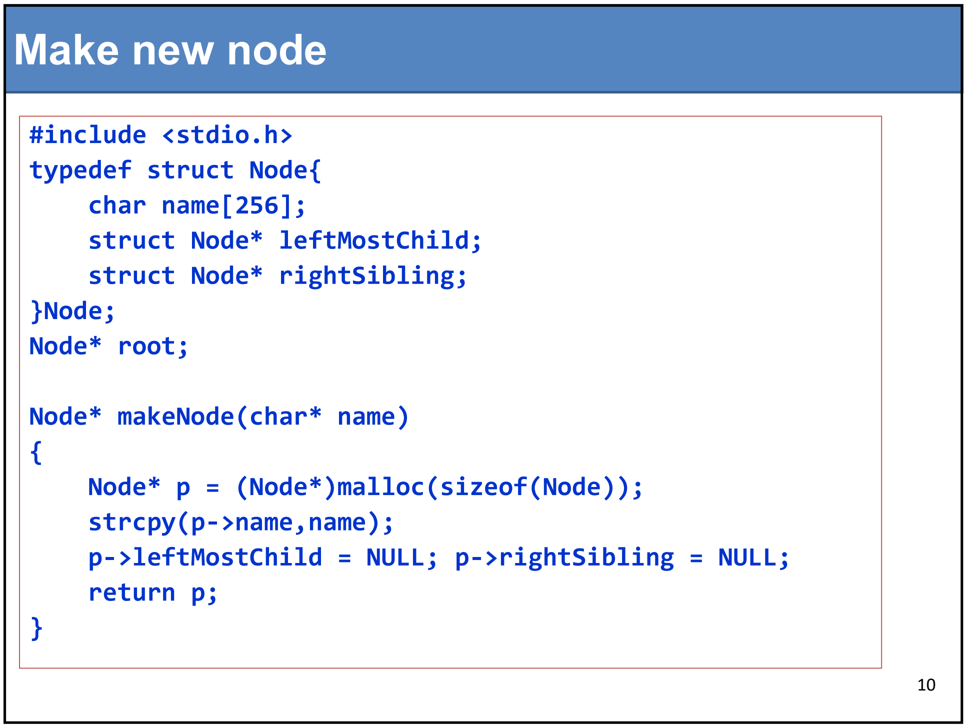
- Each node of a tree has the following structure:

```
typedef struct Node{
    char name[256];
    struct Node* leftMostChild; // pointer to the left-most child
    struct Node* rightSibling;// pointer to the right sibling
}Node;
```



Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print children of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>



Make new node

```
#include <stdio.h>
typedef struct Node{
    char name[256];
    struct Node* leftMostChild;
    struct Node* rightSibling;
}Node;
Node* root;

Node* makeNode(char* name)
{
    Node* p = (Node*)malloc(sizeof(Node));
    strcpy(p->name,name);
    p->leftMostChild = NULL; p->rightSibling = NULL;
    return p;
}
```

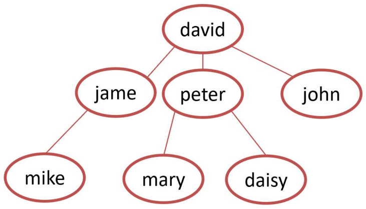
Operations on general trees

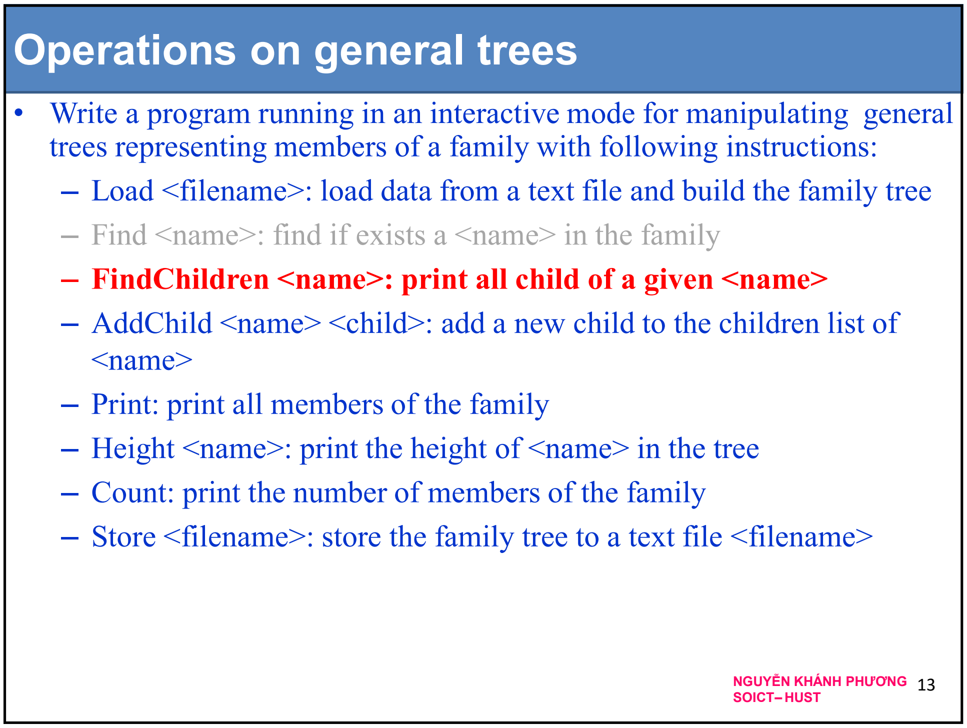
- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - **Find <name>: if there exists a <name> in the family**
  - FindChildren <name>: print children of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>

Find <name>: if there exists <name> in the tree

Find a node with name on the tree having root node pointed by pointer `root`:

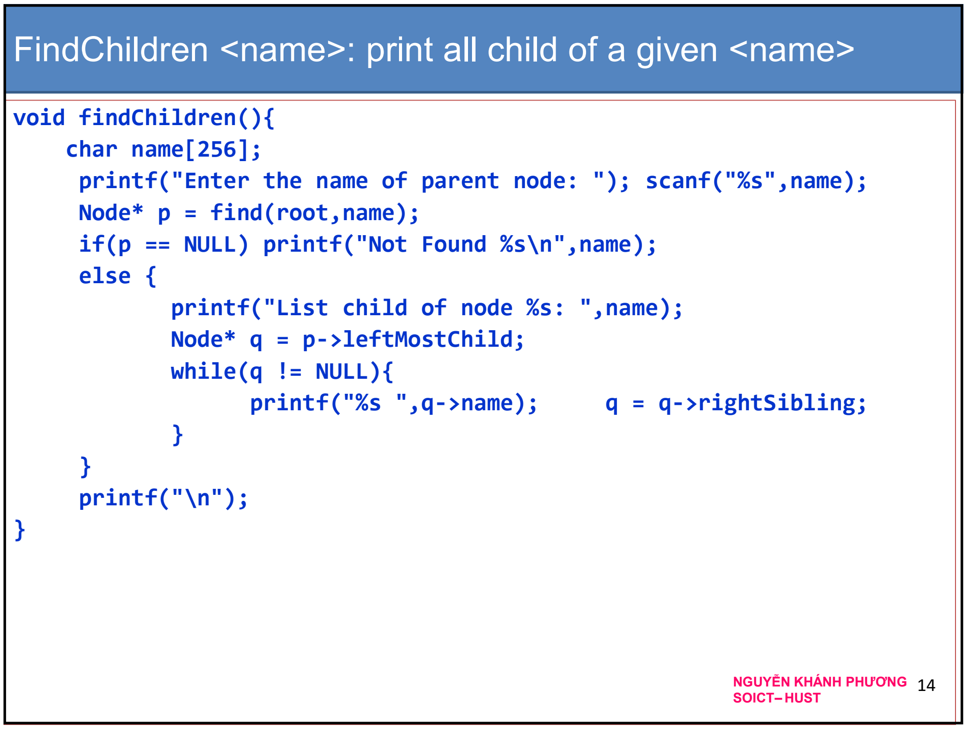
```
Node* find(Node* root, char* name)
{
    if (root == NULL) return NULL;
    if (strcmp(root->name,name) == 0) return root;
    Node* p = root->leftMostChild;
    while (p != NULL){
        Node* q = find(p,name);
        if(q != NULL) return q;
        p = p->rightSibling;
    }
}
void processFind()
{
        printf("Enter the name you want to find: ");
        scanf("%s",name);
        Node* p = find(root,name);
        if(p == NULL) printf("Not Found %s\n",name);
        else printf("Found %s\n",name);
}
```





Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - **FindChildren <name>: print all child of a given <name>**
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>



FindChildren <name>: print all child of a given <name>

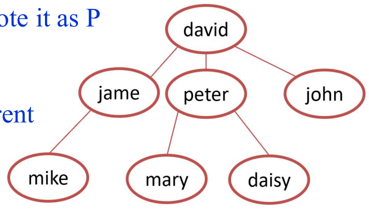
```
void findChildren(){
    char name[256];
     printf("Enter the name of parent node: "); scanf("%s",name);
     Node* p = find(root,name);
     if(p == NULL) printf("Not Found %s\n",name);
     else {
            printf("List child of node %s: ",name);
            Node* q = p->leftMostChild;
            while(q != NULL){
                  printf("%s ",q->name);     q = q->rightSibling;
            }
     }
     printf("\n");
}
```

Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print all child of a given <name>
  - **AddChild <name> <child>: add a new child to the children list of <name>**
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>

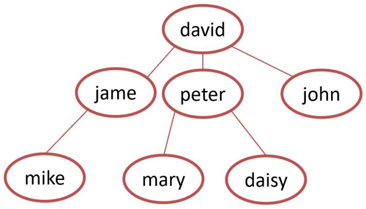
AddChild <nameP> <nameC>: add a new child with name = nameC to the children list of node with name = <nameP>

- Find the node with the given name = nameP. Denote it as P
- Find the rightmost child of the node P.
- Make the new node with name = nameC
- Make the new node as the right sibling of the current rightmost child of the node P.



```
void addChild(char *nameP, char* nameC)
{
    Node* p = find(root,nameP);
    if(p == NULL) return;
    Node *childp = p->leftMostChild;
    while (cp->rightSibling != NULL)
           childp = childp->rightSibling;
     childp->rightSibling = makeNode(nameC);
}
```

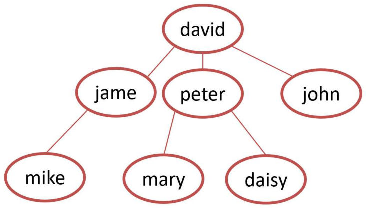
AddChild <nameP> <nameC>: add a new child with name = nameC to the children list of node with name = <nameP>



Recursive version

```
Node* addLast(Node* p, char*name){
    if(p == NULL) return makeNode(name);
    p->rightSibling = addLast(p->rightSibling, name);
    return p;
}
void addChild(char *nameP, char* nameC){
    Node* p = find(root,nameP);
    if(p == NULL) return;
    p->leftMostChild = addLast(p->leftMostChild,nameC);
}
```

AddChild <nameP> <nameC>: add a new child with name = nameC to the children list of node with name = <nameP>



```
void processAddChild()
{
   char nameP[256], nameC[256];
   printf("Enter the name of parent node: ");
   scanf("%s",nameP);
   printf("Enter the name of child node: ");
   scanf("%s",nameC);
   addChild(nameP,nameC);
}
```

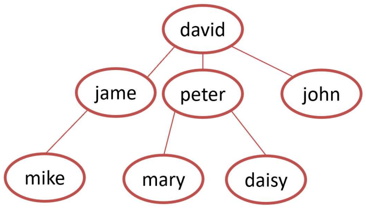
Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print all child of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - **Print: print all members of the family**
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>

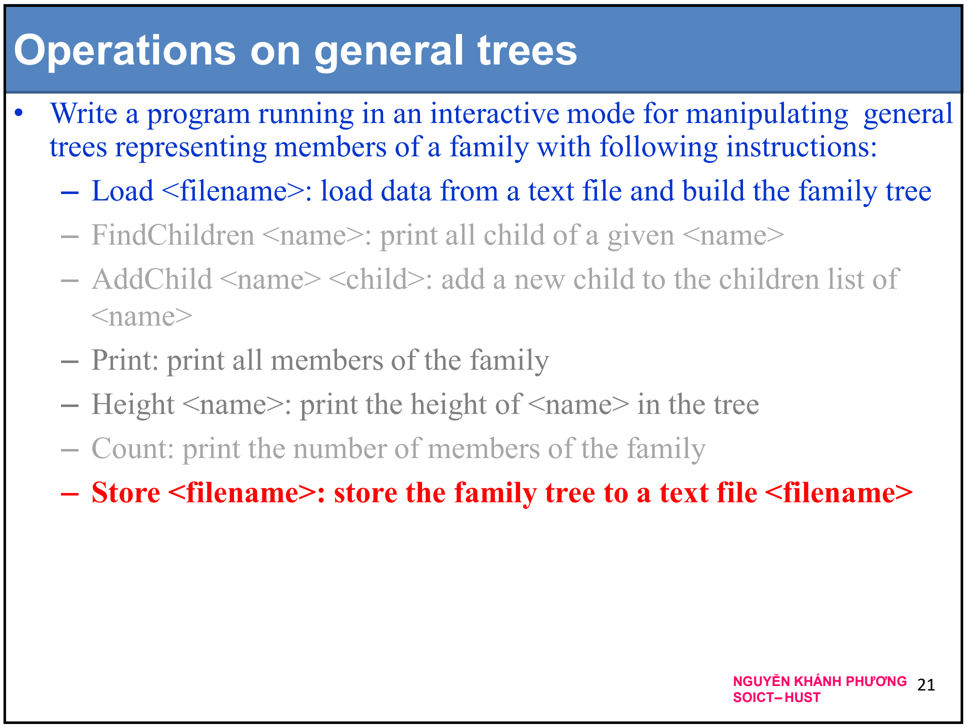
NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Print all elements in the tree on the screen using the same format as in the read file

```
void printTree(Node* root)
{
    if (root == NULL) return;
    printf("%s: ",root->name);
    Node* p = root->leftMostChild;
    while (p != NULL){
        printf("%s ",p->name);
        p = p->rightSibling;
    }
    printf("\n");
    p = root->leftMostChild;
    while (p != NULL){
        printTree(p);
        p = p->rightSibling;
    }
}
```

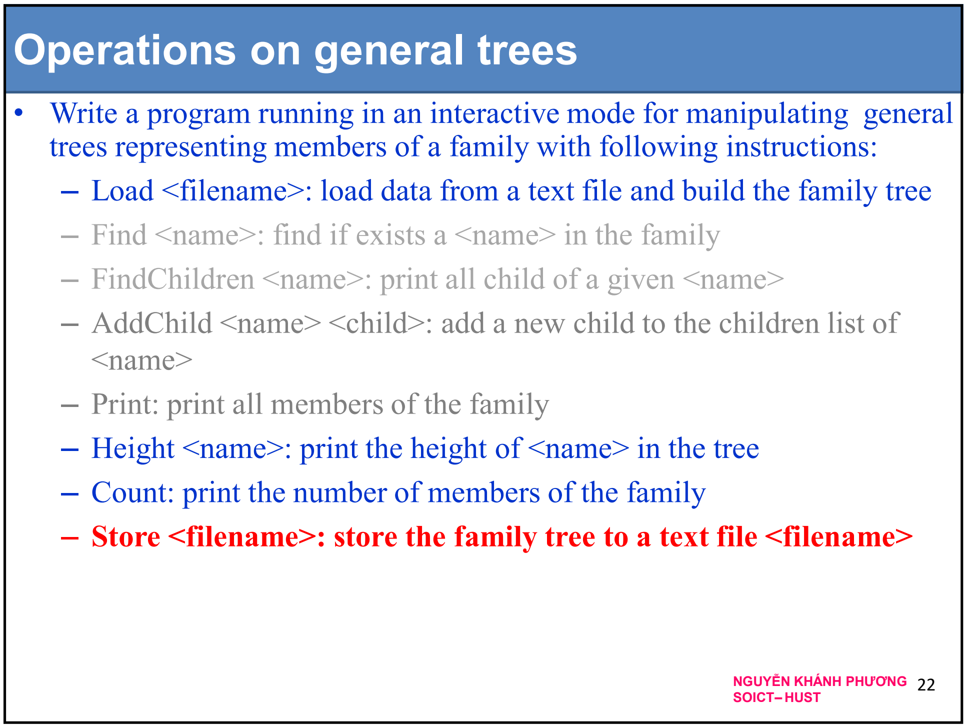


```
david jame peter john $
peter mary daisy $
jame mike $
$$
```



Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - FindChildren <name>: print all child of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - **Store <filename>: store the family tree to a text file <filename>**

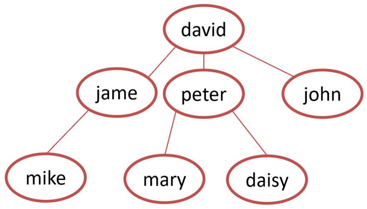


Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print all child of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - Count: print the number of members of the family
  - **Store <filename>: store the family tree to a text file <filename>**

Print all elements in the tree to a file as the same format of the read file

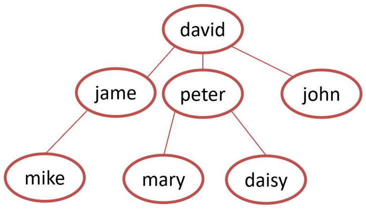
```
void printTreeF(Node* root, FILE* f)
{
    if (root == NULL) return;
    fprintf(f,"%s ",root->name);
    Node* p = root->leftMostChild;
    while(p != NULL){
        fprintf(f,"%s ",p->name);
        p = p->rightSibling;
    }
    fprintf(f," $\n");
    p = root->leftMostChild;
    while (p != NULL){
        printTreeF(p,f);
        p = p->rightSibling;
    }
}
```



```
david jame peter john $
peter mary daisy $
jame mike $
$$
```

Print all elements in the tree to a file as the same format of the read file

```
void processStore()
{
      char filename[256];
      printf("Enter the name of file: "); scanf("%s",filename);
      FILE* f = fopen(filename,"w");
      printTreeF(root,f);
      fprintf(f,"$$");
      fclose(f);
}
```



```
david jame peter john $
peter mary daisy $
jame mike $
$$
```

Operations on general trees

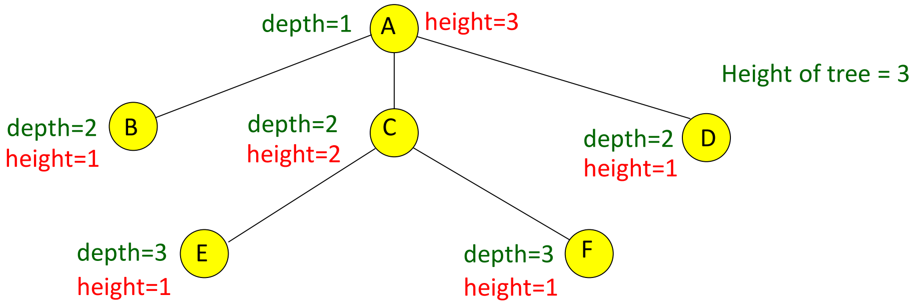
- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print all child of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - **Height <name>: print the height of <name> in the tree**
  - Count: print the number of members of the family
  - Store <filename>: store the family tree to a text file <filename>

Tree terminology

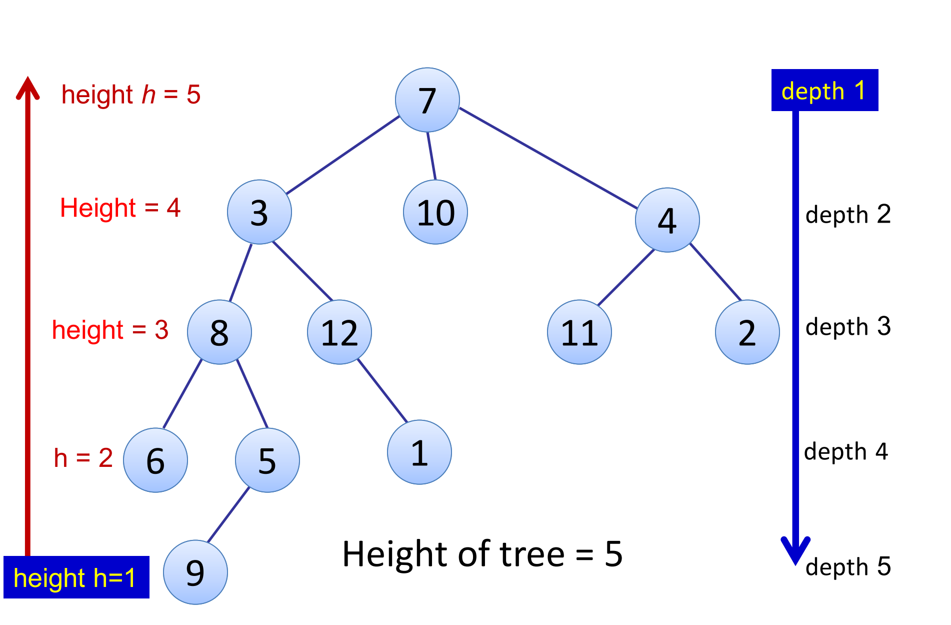
- Path: a sequence of nodes and edges connecting a node with a descendant
- Length of a path = number of edges = number of nodes - 1

(e.g.: Length of path (A → C → E) = 2; length of path (C → F) = 1)

- Depth/level of a node N = 1 + length of path from root to N
- Height of node N = 1 + length of longest path from N to a leaf
- Height (depth) of tree = height of root (= maximum depth of any node on the tree)

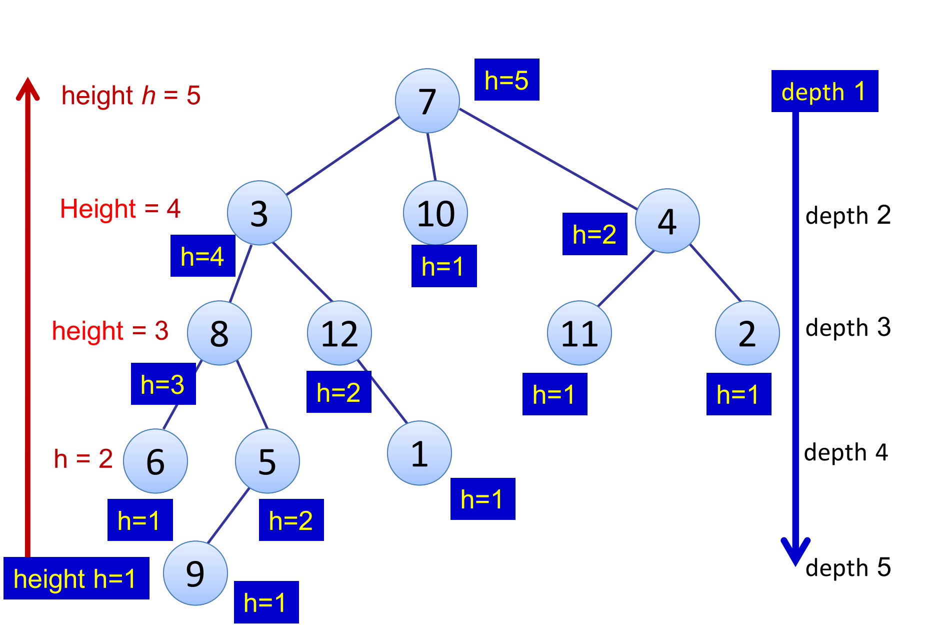


Height and depth/level



Height of tree = 5

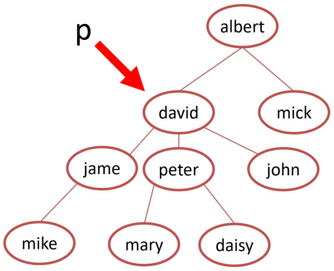
Height of node N = 1 + length of longest path from N to a leaf



Find height of the node pointed by pointer p

Height of p = 1 + max height of all child nodes of p

```
int height(Node* p) //return height of the node pointed by p on the tree
{
    if (p == NULL) return 0;
    int maxH = 0;
    Node* q = p->leftMostChild;
    while (q != NULL)
    {
        int h = height(q);
        maxH = maxH < h ? h : maxH;
        q = q->rightSibling;
    }
    return maxH + 1;
}
void processHeight()
{
     printf("Enter the name of node you want to know the height: ");
     scanf("%s",name);
     Node* p = find(root,name);
     if (p == NULL) printf("Not Found %s\n",name);
     else    printf("Found node %s having height = %d\n",name,height(p));
}
```



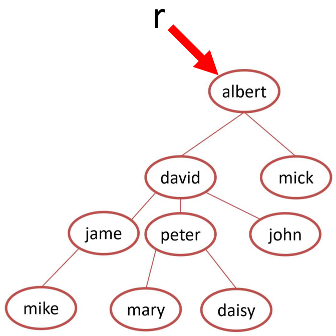
Operations on general trees

- Write a program running in an interactive mode for manipulating general trees representing members of a family with following instructions:
  - Load <filename>: load data from a text file and build the family tree
  - Find <name>: find if exists a <name> in the family
  - FindChildren <name>: print all child of a given <name>
  - AddChild <name> <child>: add a new child to the children list of <name>
  - Print: print all members of the family
  - Height <name>: print the height of <name> in the tree
  - **Count: print the number of members of the family**
  - Store <filename>: store the family tree to a text file <filename>

Count the number of nodes in the tree

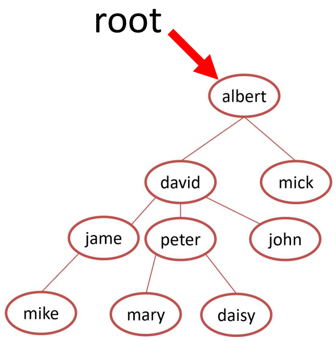
Number of nodes in the tree with root node pointed by r
= 1 + number of nodes in each subtrees

```
int count(Node* r)
//return the number of nodes in the tree with root node pointed by the pointer r
{
    if (r == NULL) return 0;
    int cnt = 1; //is root
    Node* q = r->leftMostChild;
    while(q != NULL)
    {
        cnt += count(q);
        q = q->rightSibling;
    }
    return cnt;
}
```



Free memory for all allocated memory

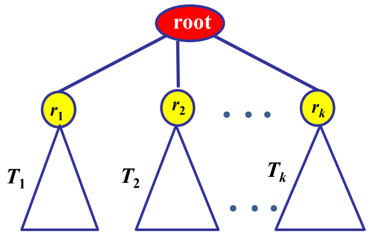
```
void freeTree(Node* root)
{
    if (root == NULL) return;
    Node* p = root->leftMostChild;
    while (p != NULL){
        Node* sp = p->rightSibling;
        freeTree(p);
        p = sp;
    }
    printf("free node %s\n",root->name);
    free(root);
    root = NULL;
}
```



Free memory for all nodes in the tree

- To free, we traverse tree in postorder: a node is visited after its descendants.

Example: Postorder traversal on tree T:

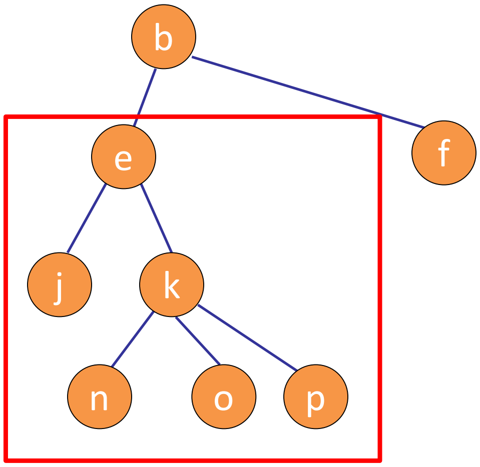


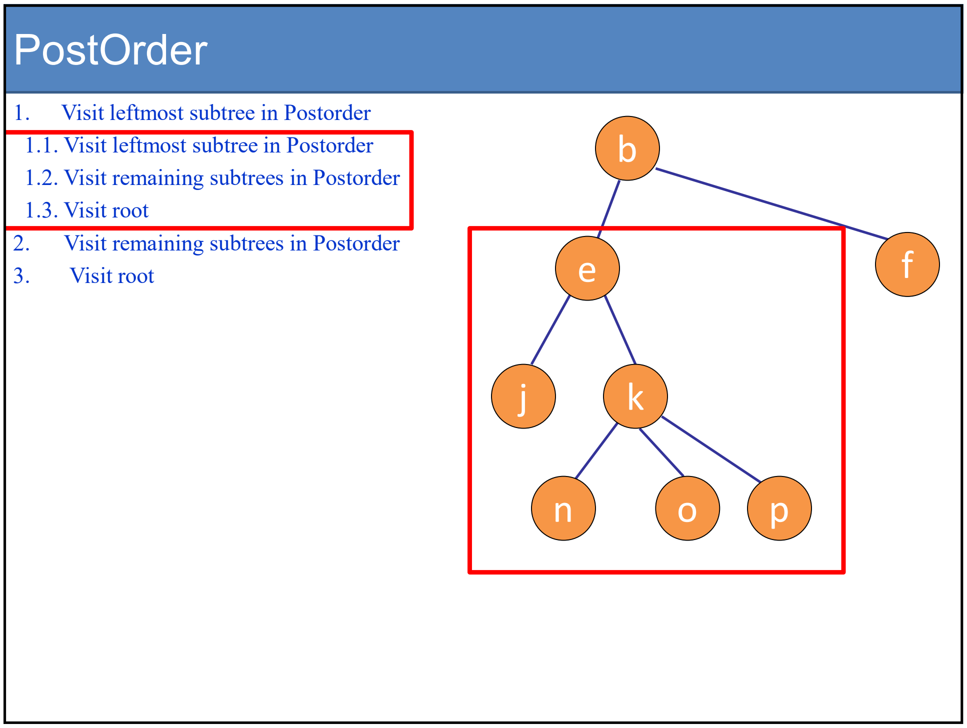
```
procedure postorder(root)
  for each child c of root from left to right
       postorder(c)
   end
   visit r
```

- Step 1: visit T_1 in postorder,
- Step 2: visit T_2 in postorder,
- ……..
- Step k: visit T_k in postorder,
- Step k+1: visit root r

PostOrder

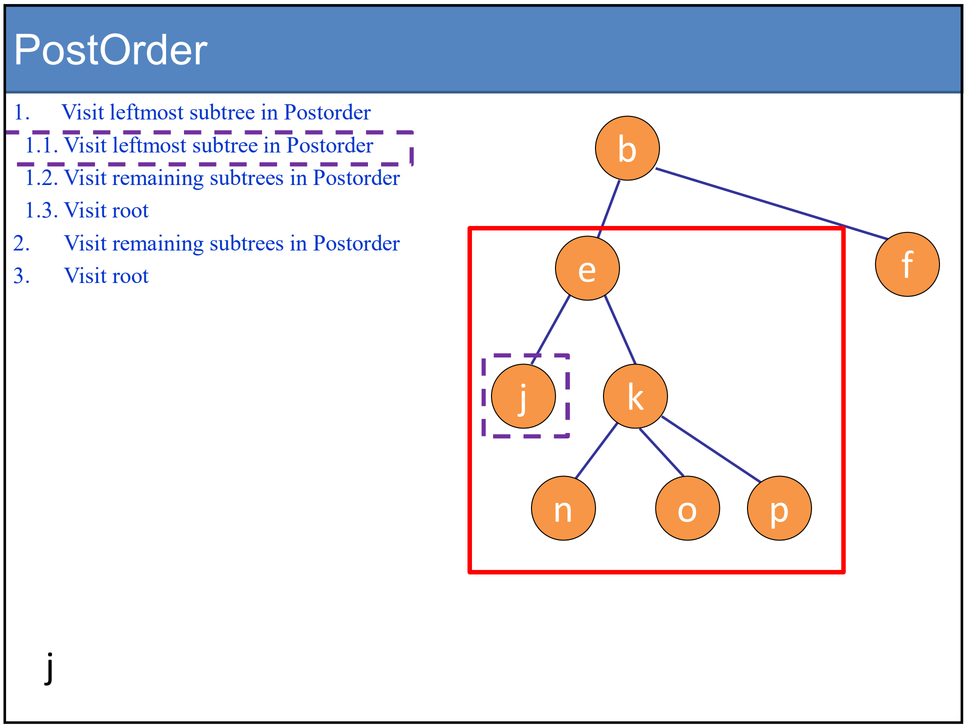
1. Visit leftmost subtree in Postorder
2. Visit remaining subtrees in Postorder
3. Visit root





PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



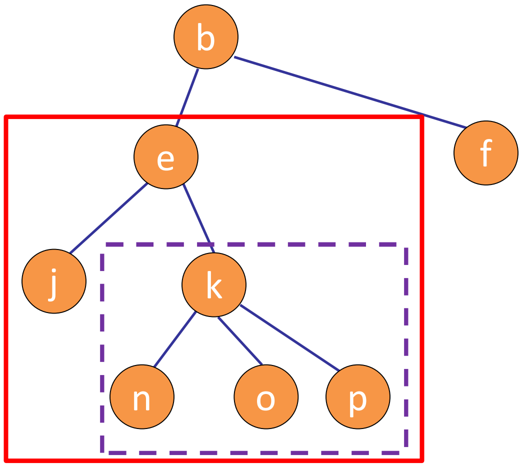
PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root

j

PostOrder

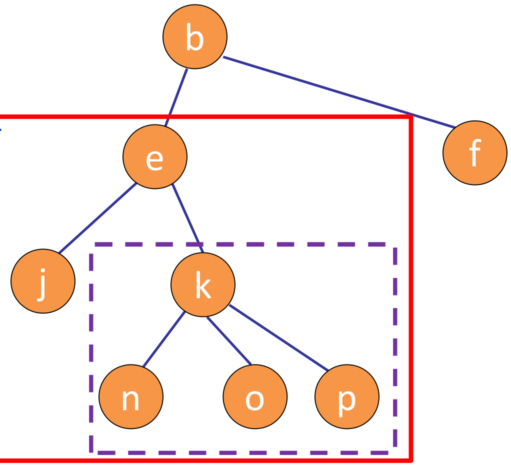
1. Visit leftmost subtree in Postorder
  1.1. Visit leftmost subtree in Postorder
  1.2. Visit remaining subtrees in Postorder
  1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



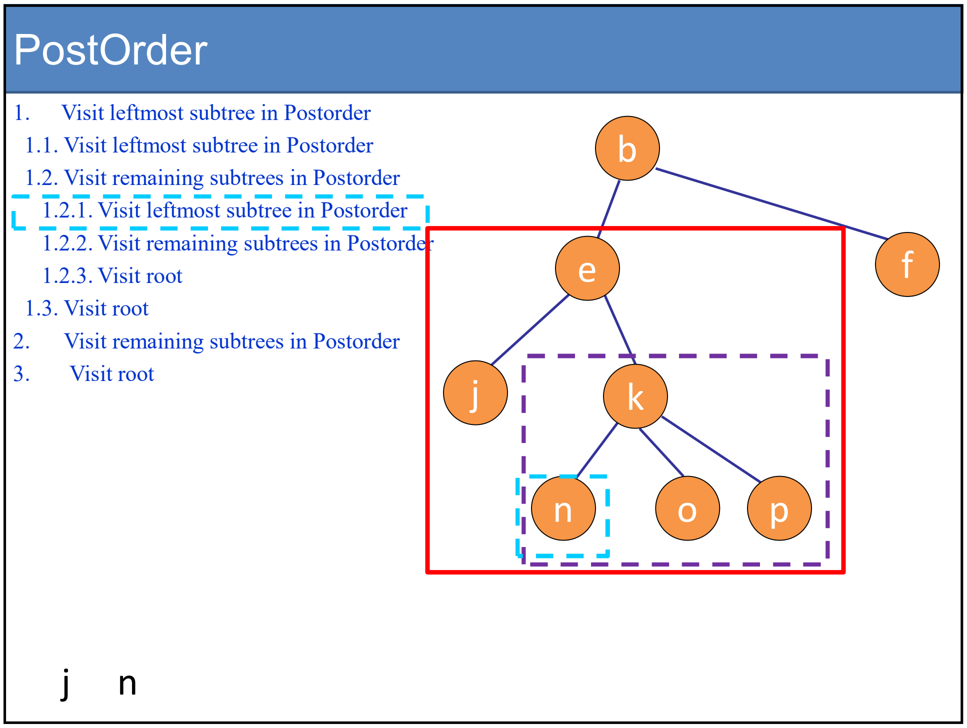
j

PostOrder

1. Visit leftmost subtree in Postorder
  1.1. Visit leftmost subtree in Postorder
  1.2. Visit remaining subtrees in Postorder
    1.2.1. Visit leftmost subtree in Postorder
    1.2.2. Visit remaining subtrees in Postorder
    1.2.3. Visit root
  1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



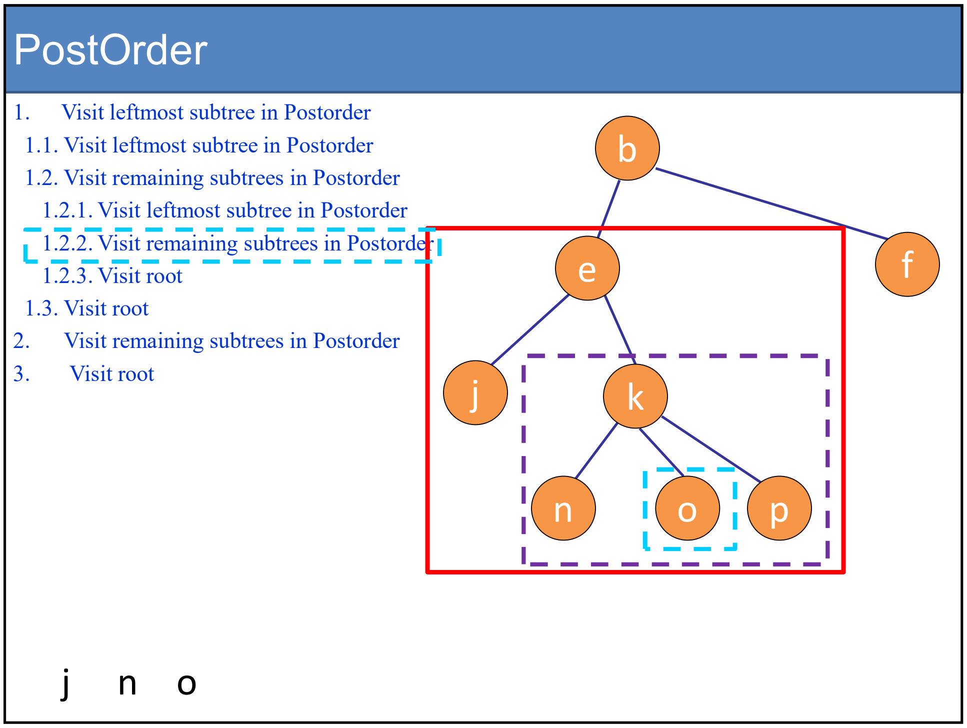
j



PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root

j n



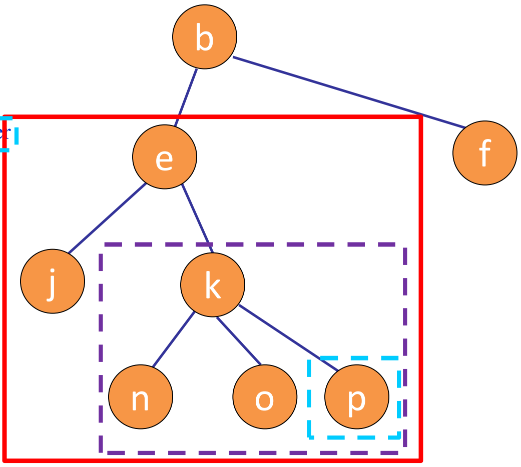
PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root

j n o

PostOrder

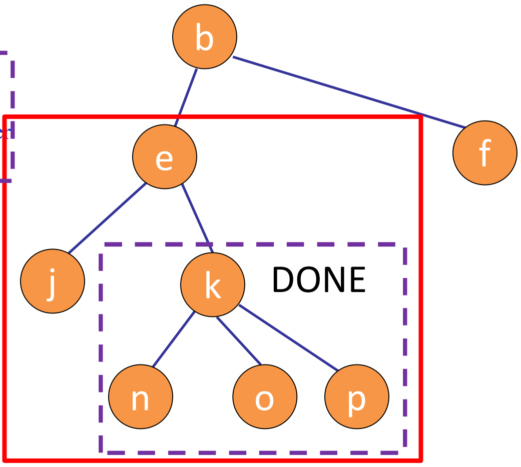
1. Visit leftmost subtree in Postorder
1.1. Visit leftmost subtree in Postorder
1.2. Visit remaining subtrees in Postorder
1.2.1. Visit leftmost subtree in Postorder
1.2.2. Visit remaining subtrees in Postorder
1.2.3. Visit root
1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



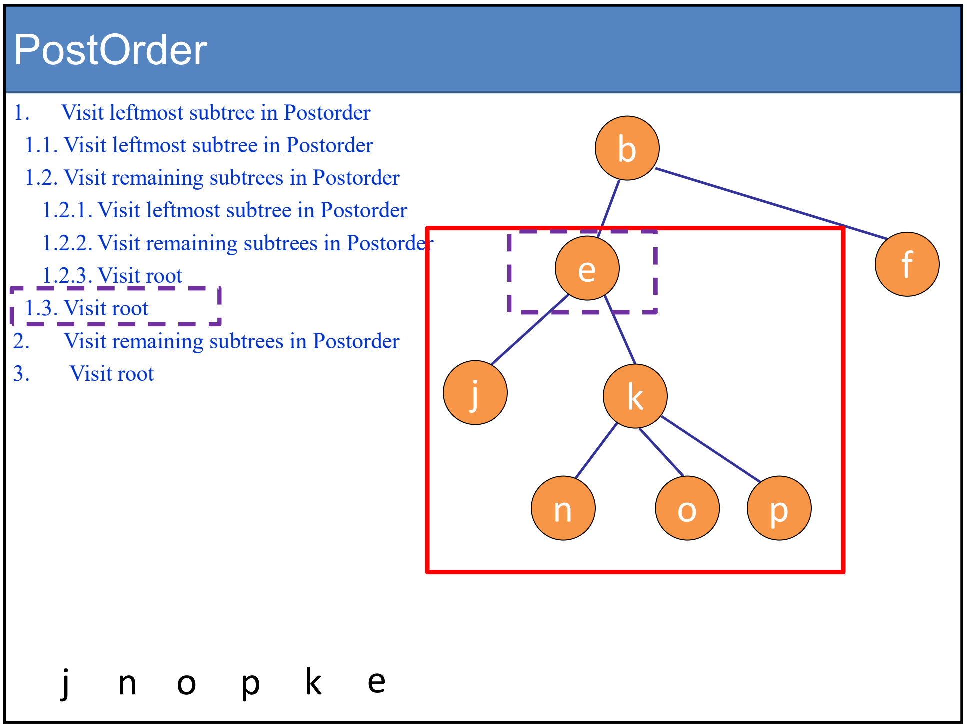
j n o p

PostOrder

1. Visit leftmost subtree in Postorder
1.1. Visit leftmost subtree in Postorder
1.2. Visit remaining subtrees in Postorder
1.2.1. Visit leftmost subtree in Postorder
1.2.2. Visit remaining subtrees in Postorder
1.2.3. Visit root
1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



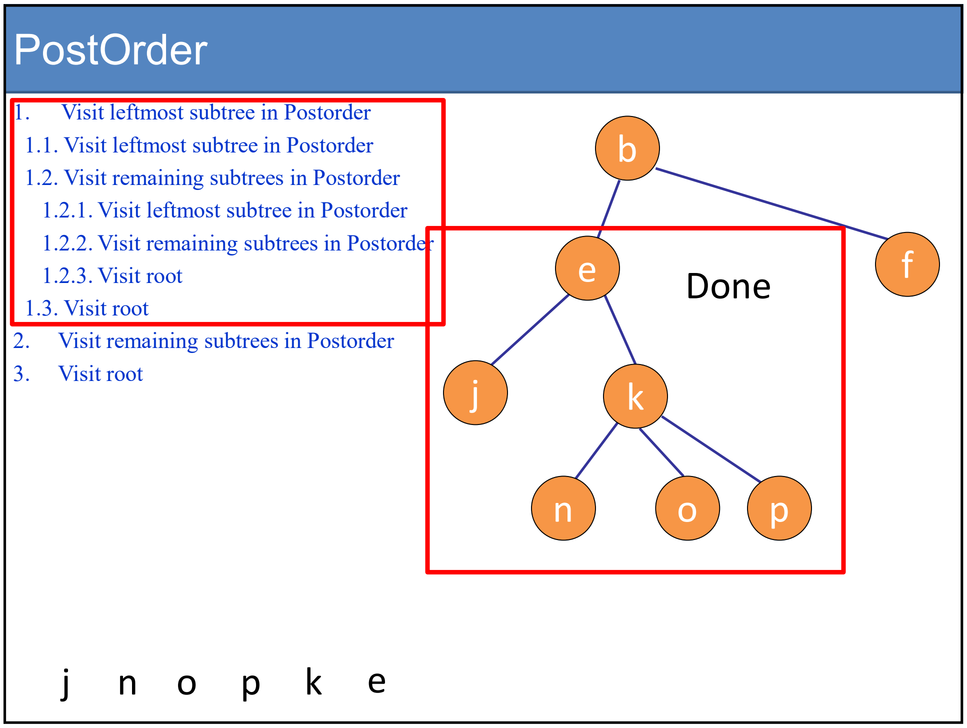
j n o p k



PostOrder

```
1.   Visit leftmost subtree in Postorder
 1.1. Visit leftmost subtree in Postorder
 1.2. Visit remaining subtrees in Postorder
   1.2.1. Visit leftmost subtree in Postorder
   1.2.2. Visit remaining subtrees in Postorder
   1.2.3. Visit root
 1.3. Visit root
2.   Visit remaining subtrees in Postorder
3.   Visit root
```

j n o p k e



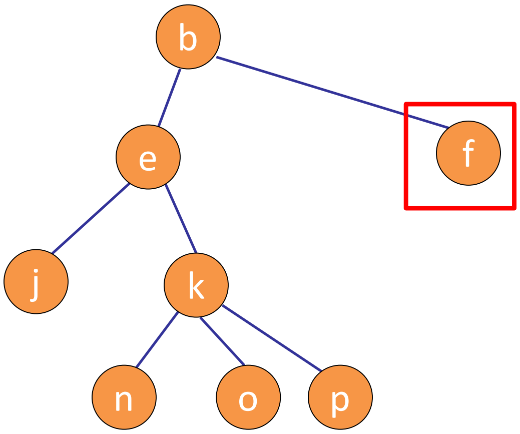
PostOrder

```
1.   Visit leftmost subtree in Postorder
 1.1. Visit leftmost subtree in Postorder
 1.2. Visit remaining subtrees in Postorder
   1.2.1. Visit leftmost subtree in Postorder
   1.2.2. Visit remaining subtrees in Postorder
   1.2.3. Visit root
 1.3. Visit root
2.   Visit remaining subtrees in Postorder
3.   Visit root
```

j n o p k e

PostOrder

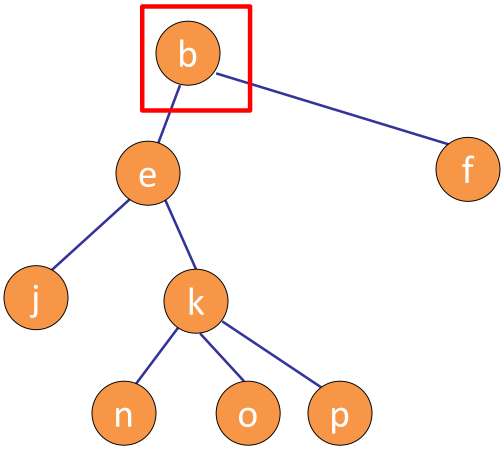
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k e f

PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root

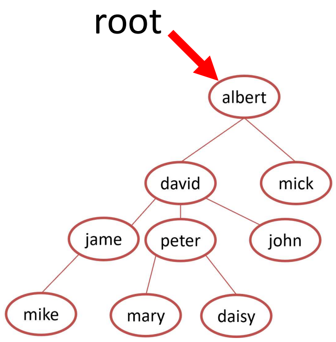


j n o p k e f b

Free memory for all allocated memory

```
procedure postorder(root)
  for each child c of root from left to right
      postorder(c)
   end
   visit r
```

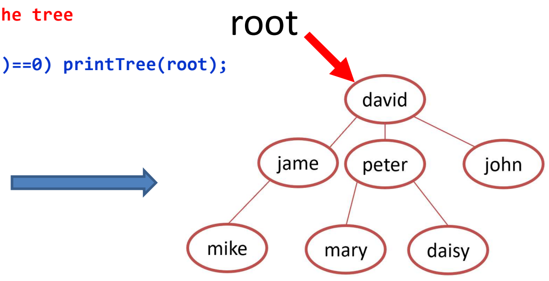
```
void freeTree(Node* root)
{
    if (root == NULL) return;
    Node* p = root->leftMostChild;
    while (p != NULL){
        Node* sp = p->rightSibling;
        freeTree(p);
        p = sp;
    }
    printf("free node %s\n",root->name);
    free(root);
    root = NULL;
}
```

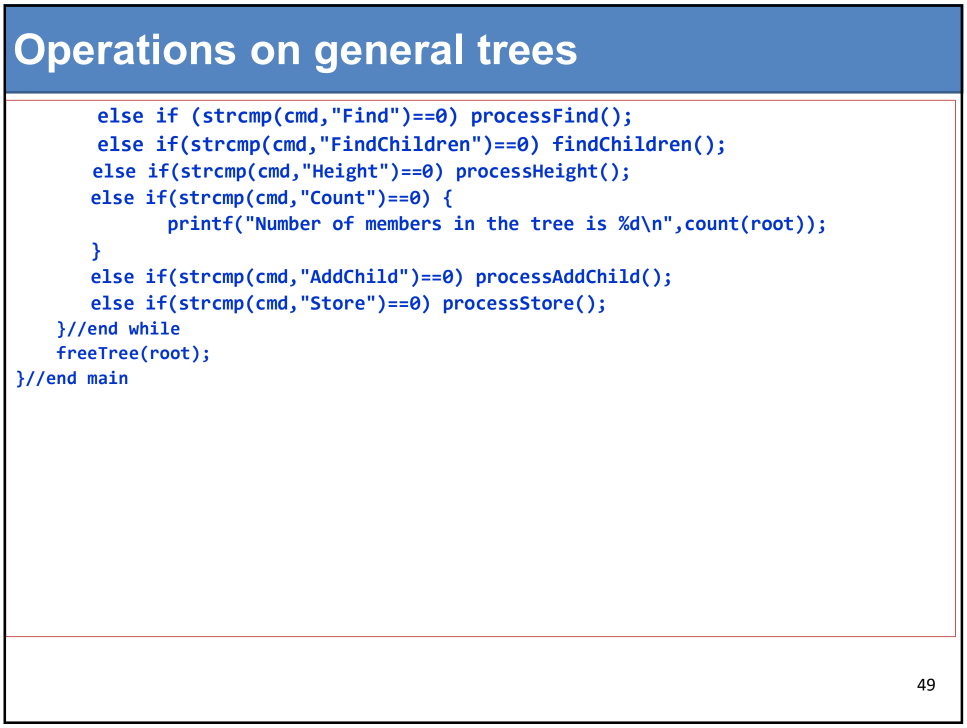


Operations on general trees

```
void main(){
    while (1){
        char cmd[256], name[256];
        printf("Enter command: ");  scanf("%s",cmd);
        if (strcmp(cmd,"Quit") == 0) break;
        else if (strcmp(cmd,"Load")==0)
        {
            char filename[256];
            printf("Enter the name of file: "); scanf("%s",filename);
            FILE* f = fopen(filename,"r");
            Read file and create the tree
        }
        else if (strcmp(cmd,"Print")==0) printTree(root);
```

```
david jame peter john $
peter mary daisy $
jame mike $
$$
```

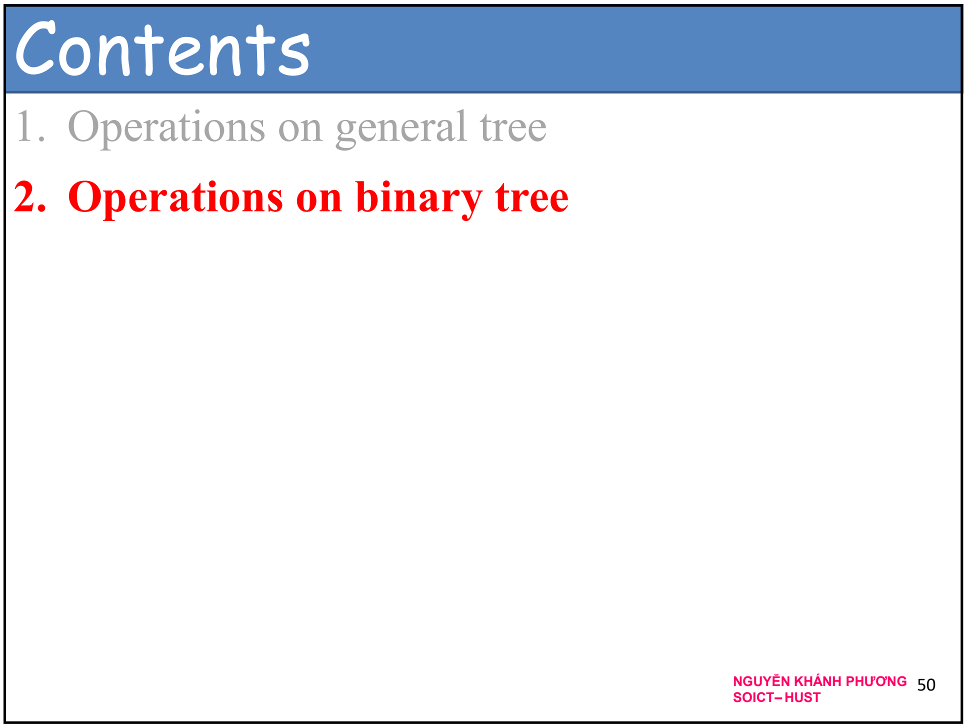




Operations on general trees

```
        else if (strcmp(cmd,"Find")==0) processFind();
        else if(strcmp(cmd,"FindChildren")==0) findChildren();
        else if(strcmp(cmd,"Height")==0) processHeight();
        else if(strcmp(cmd,"Count")==0) {
               printf("Number of members in the tree is %d\n",count(root));
        }
        else if(strcmp(cmd,"AddChild")==0) processAddChild();
        else if(strcmp(cmd,"Store")==0) processStore();
    }//end while
    freeTree(root);
}//end main
```

49

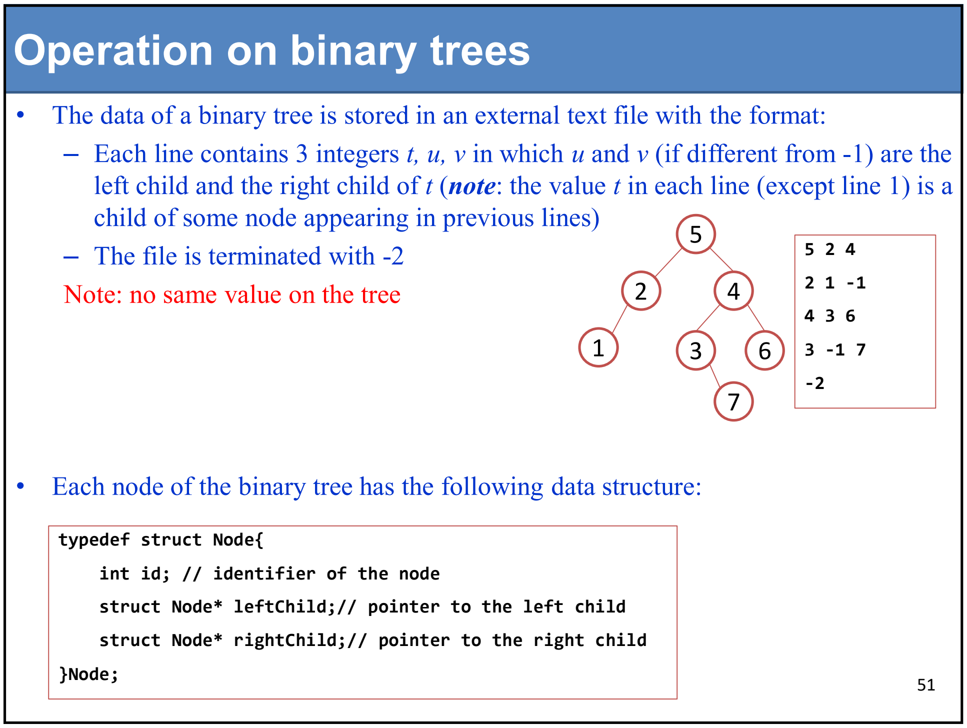


Contents

1. Operations on general tree
2. **Operations on binary tree**

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

50



Operation on binary trees

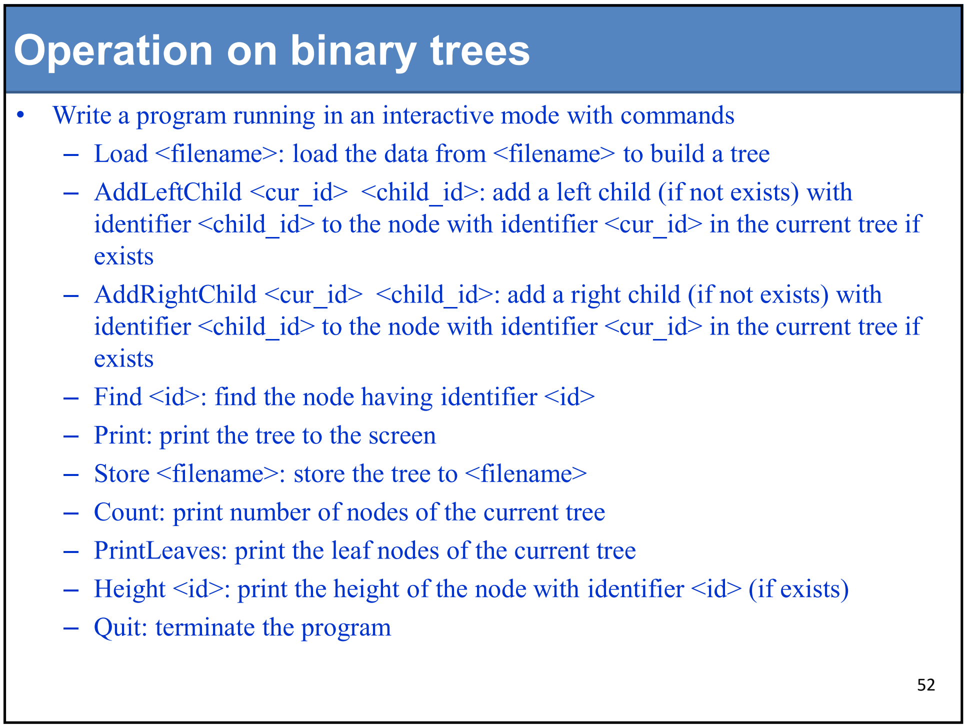
- The data of a binary tree is stored in an external text file with the format:
  - Each line contains 3 integers *t, u, v* in which *u* and *v* (if different from -1) are the left child and the right child of *t* (***note***: the value *t* in each line (except line 1) is a child of some node appearing in previous lines)
  - The file is terminated with -2

  Note: no same value on the tree

```
5 2 4
2 1 -1
4 3 6
3 -1 7
-2
```

- Each node of the binary tree has the following data structure:

```
typedef struct Node{
    int id; // identifier of the node
    struct Node* leftChild;// pointer to the left child
    struct Node* rightChild;// pointer to the right child
}Node;
```



Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

Make new node

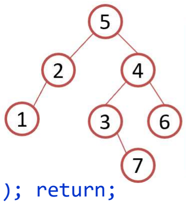
```
#include <stdio.h>
typedef struct Node{
    int id;
    struct Node* leftChild;
    struct Node* rightChild;
}Node;
Node* root;

Node* makeNode(int id){
    Node* p = (Node*)malloc(sizeof(Node));
    p->id = id;
    p->leftChild = NULL; p->rightChild = NULL;
    return p;
}
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

addLeftChild and addRightChild



```
5 2 4
2 1 -1
4 3 6
3 -1 7
-2
```

```
void addLeftChild(int u, int left){
    Node* pu = find(root,u);
    if(pu == NULL){
        printf("Not found %d\n",u); return;
    }
    if(pu->leftChild != NULL){
        printf("Node %d has already leftChild\n",u); return;
    }
    pu->leftChild = makeNode(left);
}
void addRightChild(int u, int right){
    Node* pu = find(root,u);
    if(pu == NULL){
        printf("Not found %d\n",u); return;
    }
    if(pu->rightChild != NULL){
        printf("Node %d has already rightChild\n",u); return;
    }
    pu->rightChild = makeNode(right);
}
```

addLeftChild and addRightChild

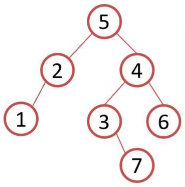
```
void processAddLeftChild(){
    int id,idC;
    printf("Enter values of node and its child node: ");
    scanf("%d%d",&id,&idC);
    addLeftChild(id,idC);
}
void processAddRightChild(){
    int id,idC;
    printf("Enter values of node and its right node: ");
    scanf("%d%d",&id,&idC);
    addRightChild(id,idC);
}
```

Operation on binary trees

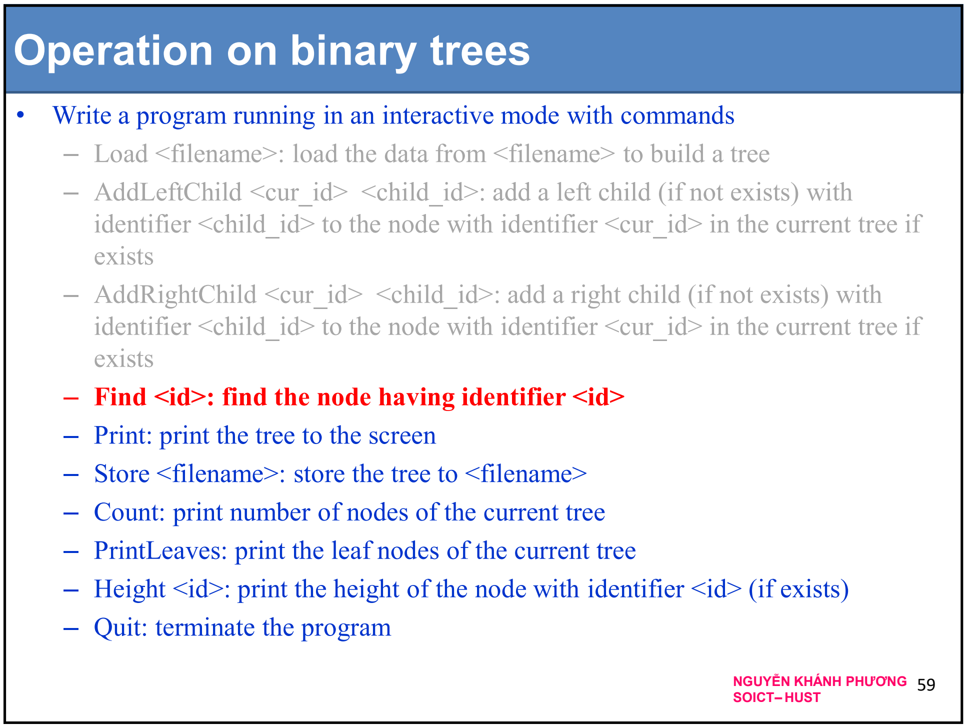
- Write a program running in an interactive mode with commands
  - **Load <filename>: load the data from <filename> to build a tree**
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

Load file

```
void load(char* filename){
    FILE* f = fopen(filename,"r");
    root = NULL;
    ........................
    fclose(f);
}
void processLoad(){
    char filename[256];
    printf("Enter the name of file you want to read: ");scanf("%s",filename);
    load(filename);
}
```

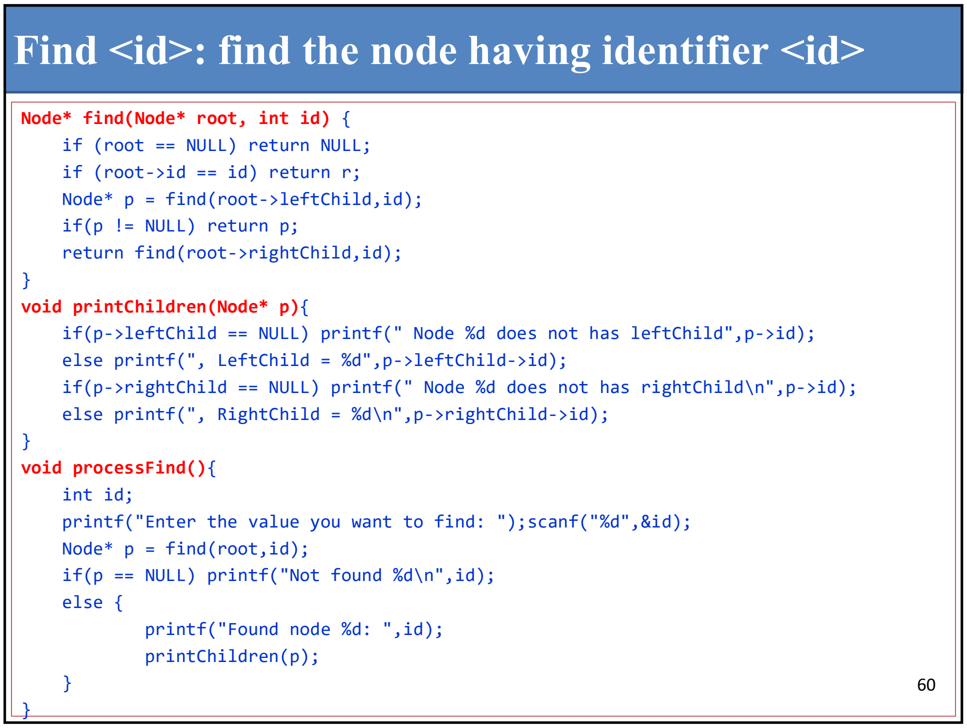


```
5 2 4
2 1 -1
4 3 6
3 -1 7
-2
```



Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - **Find <id>: find the node having identifier <id>**
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program



Find <id>: find the node having identifier <id>

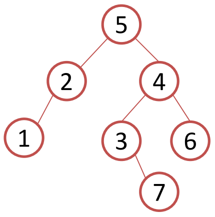
```
Node* find(Node* root, int id) {
    if (root == NULL) return NULL;
    if (root->id == id) return r;
    Node* p = find(root->leftChild,id);
    if(p != NULL) return p;
    return find(root->rightChild,id);
}
void printChildren(Node* p){
    if(p->leftChild == NULL) printf(" Node %d does not has leftChild",p->id);
    else printf(", LeftChild = %d",p->leftChild->id);
    if(p->rightChild == NULL) printf(" Node %d does not has rightChild\n",p->id);
    else printf(", RightChild = %d\n",p->rightChild->id);
}
void processFind(){
    int id;
    printf("Enter the value you want to find: ");scanf("%d",&id);
    Node* p = find(root,id);
    if(p == NULL) printf("Not found %d\n",id);
    else {
            printf("Found node %d: ",id);
            printChildren(p);
    }
}
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - **Print: print the tree to the screen**
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

Print all nodes on the tree

```
void printTree(Node* root){
    if (root == NULL) return;
    printf("%d: ",r->id);
    if(root->leftChild == NULL)  printf("leftChild = NULL");
    else printf("leftChild = %d",root->leftChild->id);
    if(root->rightChild == NULL)  printf(", rightChild = NULL");
    else printf(", rightChild = %d",root->rightChild->id);
    printf("\n");
    printTree(root->leftChild);
    printTree(root->rightChild);
}
```



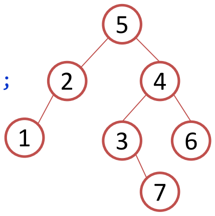
```
5: leftChild = 2, rightChild = 4
2: leftChild = 1, rightChild=NULL
1: leftChild = NULL, rightChild=NULL
4: leftChild = 3, rightChild = 6
3: leftChild = NULL, rightChild = 7
5: leftChild = NULL, rightChild = NULL
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - **Store <filename>: store the tree to <filename>**
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

Print all nodes on the tree to the file

```
void printTreeF(Node* root, FILE* f){
    if(root == NULL) return;
    fprintf(f,"%d ",root->id);
    if(root->leftChild == NULL)  fprintf(f,"-1 ");
    else fprintf(f,"%d ",root->leftChild->id);
    if(root->rightChild == NULL)  fprintf(f,"-1 ");
    else fprintf(f,"%d ",root->rightChild->id);
    fprintf(f,"\n");
    printTreeF(root->leftChild,f);
    printTreeF(root->rightChild,f);
}
void processStore(){
    char filename[256];
    printf("Enter the name of file: "); scanf("%s",filename);
    FILE* f = fopen(filename,"w");
    printTreeF(root,f);
    fprintf(f,"-2");
    fclose(f);
}
```



```
5 2 4
2 1 -1
4 3 6
3 -1 7
-2
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - **Count: print number of nodes of the current tree**
  - PrintLeaves: print the leaf nodes of the current tree
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

Count number of nodes on the tree

```
//return the number of nodes on the tree having root node pointed by pointer root
int count(Node* root){
    if(root == NULL) return 0;
    return 1 + count(root->leftChild) + count(root->rightChild);
}
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - **PrintLeaves: print the leaf nodes of the current tree**
  - Height <id>: print the height of the node with identifier <id> (if exists)
  - Quit: terminate the program

PrintLeaves: print all the leaf nodes of the tree

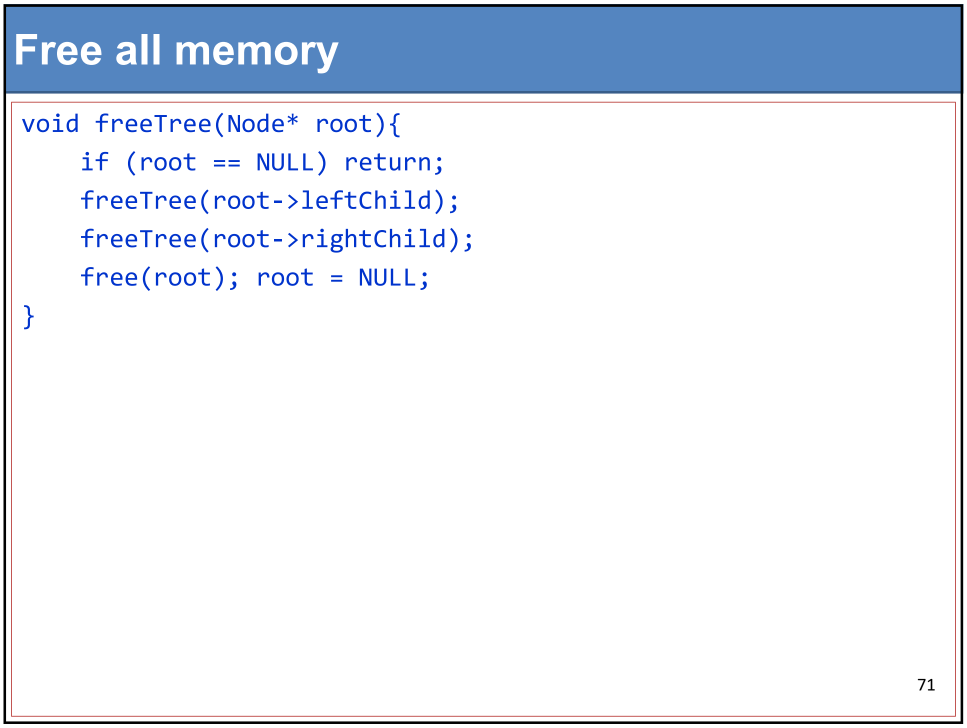
```
//print all the leaf nodes on the tree having root node pointed by pointer root
void printLeaves(Node* root){
    if (root == NULL) return;
    if (root->leftChild == NULL && root->rightChild == NULL)
        printf("%d ",root->id);
    printLeaves(root->leftChild);
    printLeaves(root->rightChild);
}
```

Operation on binary trees

- Write a program running in an interactive mode with commands
  - Load <filename>: load the data from <filename> to build a tree
  - AddLeftChild <cur_id> <child_id>: add a left child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - AddRightChild <cur_id> <child_id>: add a right child (if not exists) with identifier <child_id> to the node with identifier <cur_id> in the current tree if exists
  - Find <id>: find the node having identifier <id>
  - Print: print the tree to the screen
  - Store <filename>: store the tree to <filename>
  - Count: print number of nodes of the current tree
  - PrintLeaves: print the leaf nodes of the current tree
  - **Height <id>: print the height of the node with identifier <id> (if exists)**
  - Quit: terminate the program

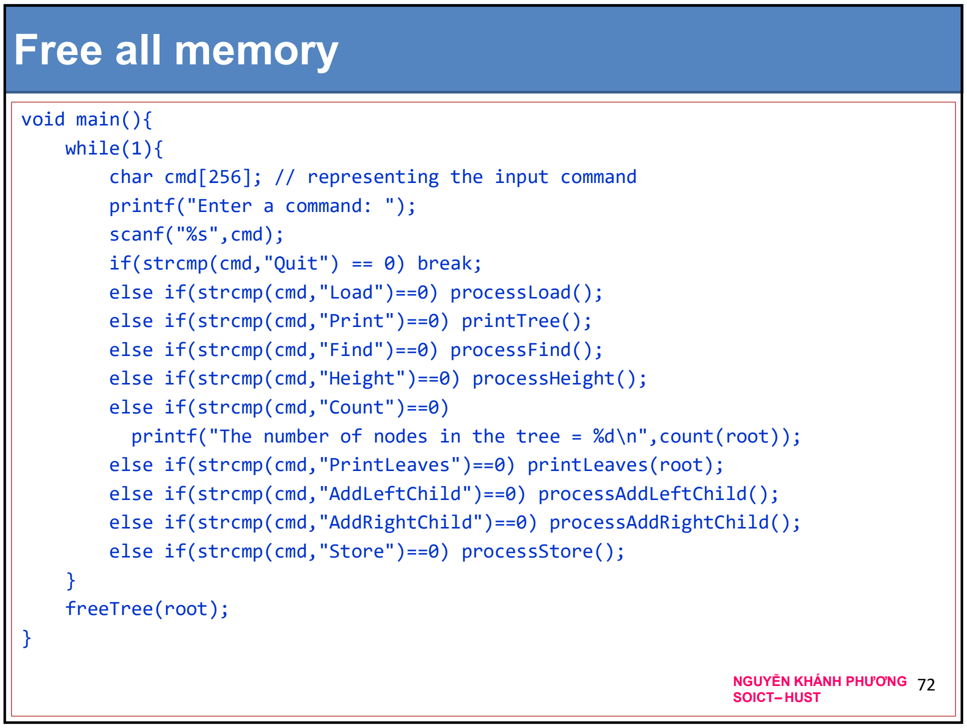
Find height of a node with value = id

```
//return height of the node pointed by pointer p:
int height(Node* p){
    if (p == NULL) return 0;
    int maxH = 0;
    int hL = height(p->leftChild);
    if (maxH < hL) maxH = hL;
    int hR = height(p->rightChild);
    if(maxH < hR) maxH = hR;
    return maxH + 1;
}
void processHeight(){
    int id;
    printf("Enter the value of the node you want to know height: ");
    scanf("%d",&id);
    Node* p = find(root,id);
    if(p == NULL) printf("Not found %d on the tree\n",id);
    else printf("Height of node %d is %d\n",height(p));
}
```



Free all memory

```
void freeTree(Node* root){
    if (root == NULL) return;
    freeTree(root->leftChild);
    freeTree(root->rightChild);
    free(root); root = NULL;
}
```



Free all memory

```
void main(){
    while(1){
        char cmd[256]; // representing the input command
        printf("Enter a command: ");
        scanf("%s",cmd);
        if(strcmp(cmd,"Quit") == 0) break;
        else if(strcmp(cmd,"Load")==0) processLoad();
        else if(strcmp(cmd,"Print")==0) printTree();
        else if(strcmp(cmd,"Find")==0) processFind();
        else if(strcmp(cmd,"Height")==0) processHeight();
        else if(strcmp(cmd,"Count")==0)
          printf("The number of nodes in the tree = %d\n",count(root));
        else if(strcmp(cmd,"PrintLeaves")==0) printLeaves(root);
        else if(strcmp(cmd,"AddLeftChild")==0) processAddLeftChild();
        else if(strcmp(cmd,"AddRightChild")==0) processAddRightChild();
        else if(strcmp(cmd,"Store")==0) processStore();
    }
    freeTree(root);
}
```