PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees

Binary Trees: Basic Concept and Definitions

Dr. Shylaja S S
Ms. Kusuma K V

Binary Tree: is a Non Linear Data Structure

Definition: Finite set of elements that is either empty or is partitioned into three subsets

- First subset: is a single element, called the root
- Second subset: is a binary tree, called the left binary tree
- Third subset: is a binary tree, called the right binary tree

Figure 3 shows a Binary Tree with A as its root. Tree with nodes B, D, E, G form the left subtree and the tree with nodes C, F, H, I form the right subtree.

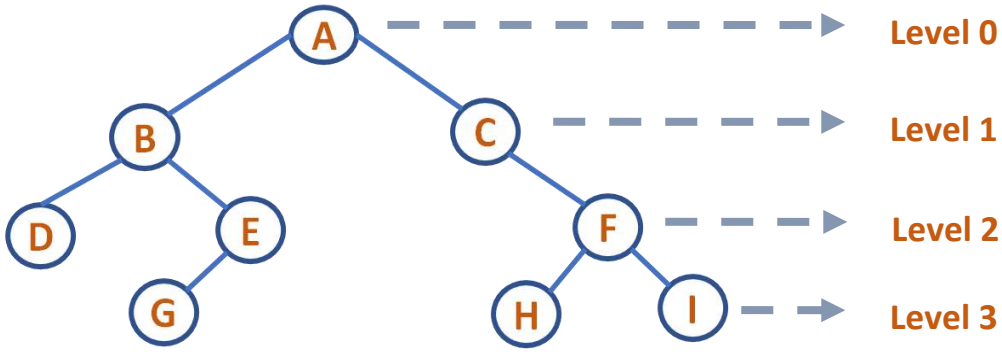


Figure 3: Binary Tree

Each element of a binary tree is called a **node** of the tree.

Left node B of A is called left child of A. Right node C of A is called the right child of A.

Two nodes are called **siblings** if they are left and right children of the same parent. A is called the parent of B and C. B and C are called siblings

A node which has no child is called **leaf node/external node**

A node which has a child is called the non **leaf node/internal node**

A node N1 is called the **ancestor** of a node N2 if N1 is either the parent of N2 or N1 is the parent of some ancestor of N2. A node N2 becomes the **descendent** of node N1. Descendent can be either the left descendent or the right descendent.

Level of a node: Root has Level 0; Level of any other node is one more than its parent.

Depth of a tree: Maximum level of any leaf in the tree (path length from the deepest leaf to the root)

Depth of a node: Path length from the node to the root

Height of a tree: Path length from the root node to the deepest leaf

Height of a node: Path length from the node to the deepest leaf

In Figure 3, D, G, H and I are leaf nodes. A, B, C, E and F are internal nodes. A is the ancestor of all the nodes in the tree. B is the left descendant of A. C is the right descendent of A.

Level of node A: 0, Level of node B, C: 1, Level of node D, E, F: 2, Level of node G, H, I: 3

Depth of tree: 3

Depth of node A: 0, Depth of node B, C: 1, Depth of node D, E, F: 2, Depth of node G, H, I: 3

Height of tree: 3

Height of Node A: 3, Height of Node B, C: 2, Height of Node E, F: 1, Height of Node D,G,H,I: 0

Figure 4 shows some structures that are not binary trees.

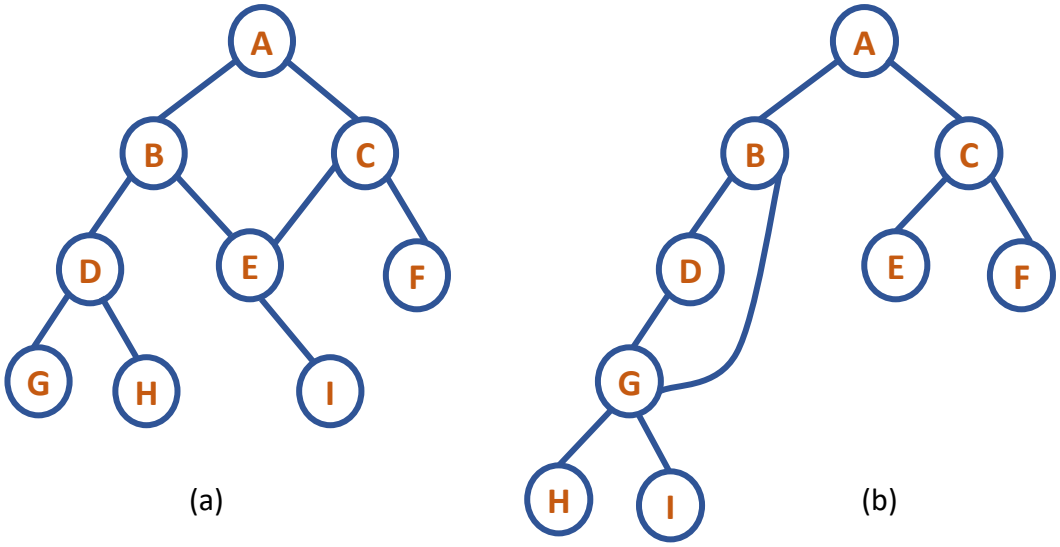


Figure 4: Structures that are not Binary Trees

Note: Different authors use the following tree terminologies in different ways.
Strictly Binary Tree: A Binary tree where every node has either zero/two children.

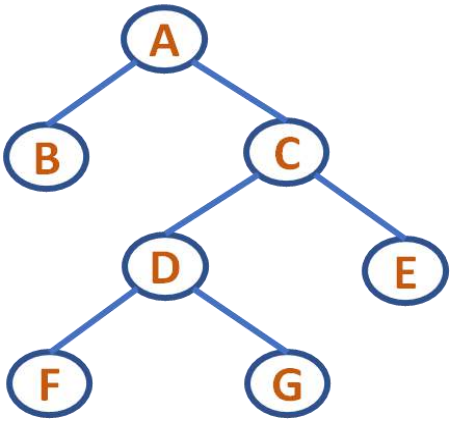


Figure 5: Strictly Binary Tree

Fully Binary Tree: A binary tree with all the leaves at the same level.

- If the binary tree has depth d, then there are 0 to d levels
- Total no. of nodes = $2^0+2^1+\ldots\ldots+2^d = 2^{(d+1)}-1$

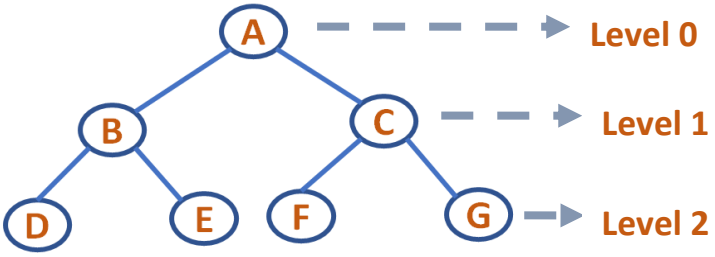


Figure 6: Full Binary Tree of depth 2

Complete Binary Tree: For a Complete Binary Tree with n nodes and depth d:

1. Any node nd at level less than d-1 has two children
2. For any node nd of the tree with a right descendent at level d, nd must have a left child and every left descendent of nd is either a leaf at level d or has two children

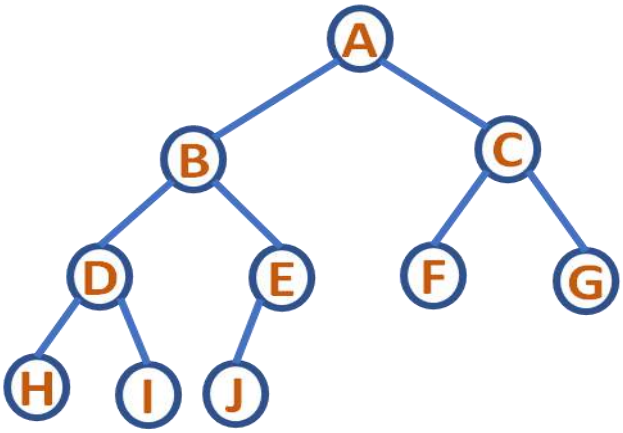
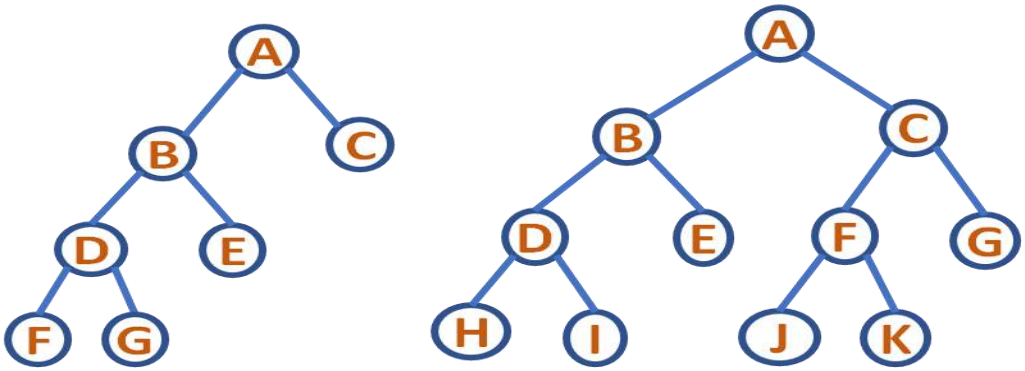


Figure 7: Complete Binary Tree

The binary tree of Figure 8(a) is not complete, since it contains leaves at levels 1, 2, and 3, thereby violating condition 1. The binary tree of Figure 8(b) satisfies condition1, since every leaf is either at level 2 or at level 3. However, condition 2 is violated, since A has a right descendent at level 3 (J) but also has a left descendant that is a leaf at level 2(E).



(a) (b)

Figure 8: Structures that are not Complete Binary Trees

Binary Tree Properties

- Every node except the root has exactly one parent
- A tree with n nodes has n-1 edges (every node except the root has an edge to its parent)
- A tree consisting of only root node has height of zero
- The total number of nodes in a full binary tree of depth d is $2^{(d+1)} - 1$, $d \geq 0$
- For any non-empty binary tree, if n_0 is the number of leaf nodes and n2 the nodes of degree 2, then $n_0 = n_2 + 1$

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Binary Search Tree (BST) and its Implementation using Dynamic Allocation: Insertion

Dr. Shylaja S S
Ms. Kusuma K V

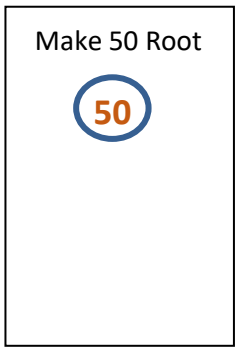
Binary Search Tree: is a Binary tree that is either empty or in which every node contains a key and satisfies the conditions:

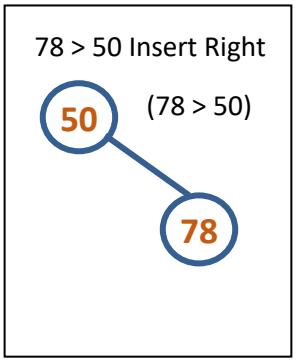
- The key in the left side of a child (if it exists) is less than the key in the parent node
- The key in the right side of a child (if it exists) is greater than the key in the parent node
- The left and right subtrees of the root are again binary search trees

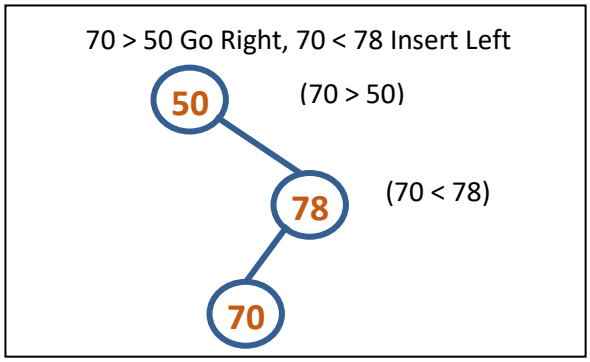
If in case of equal keys we can modify the definition but normally we take distinct keys.

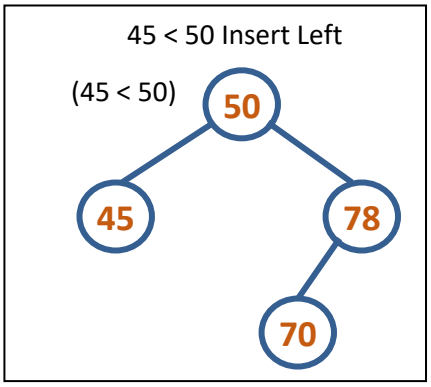
Construct a Binary Search Tree for the elements inserted in the following order:

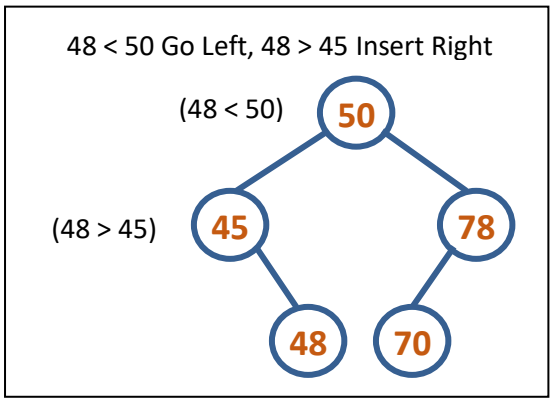
50, 78, 70, 45, 48, 35, 85

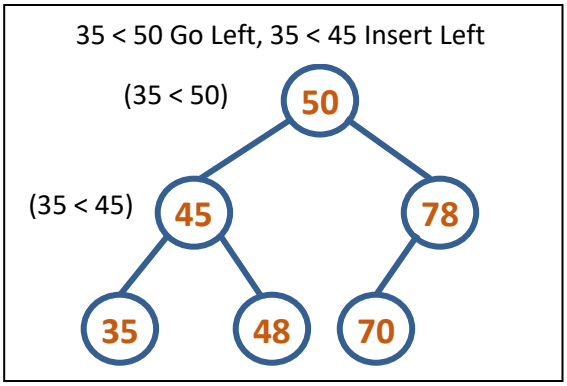


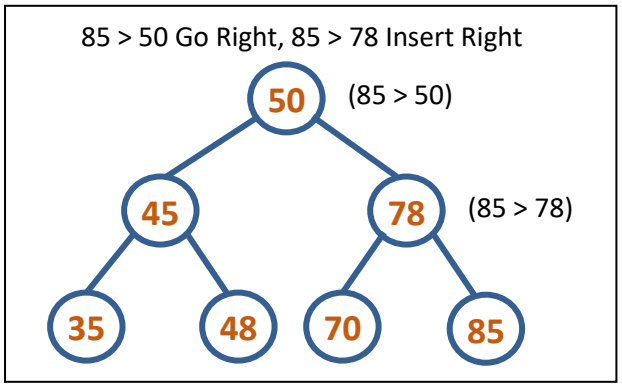












BST: Linked Representation

In linked representation each node in a binary tree has three fields, the left child field, information field and the right child field. Pictorially a node used for linked representation may be as shown in Figure3. If the left subtree is empty then the corresponding node's left child field is set to null. If the right subtree is empty then the corresponding node's right child field is set to null. If the tree itself is empty the root pointer is set to null.

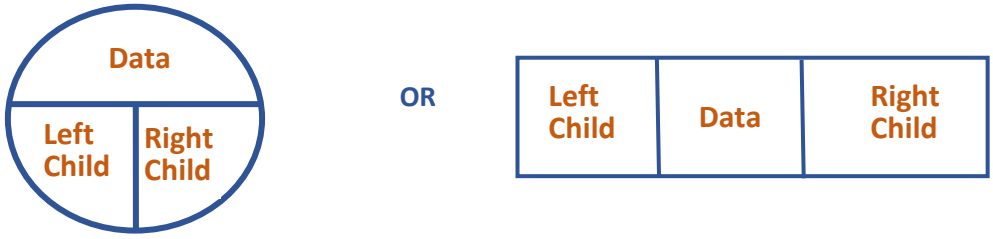


Figure 3: Pictorial representation of a node in Linked representation

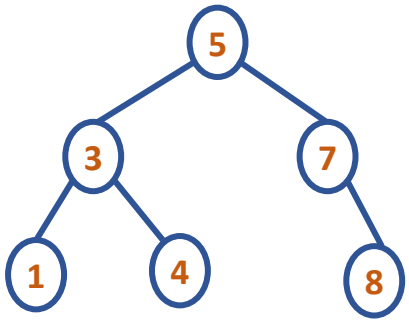


Figure 4: Binary Search Tree

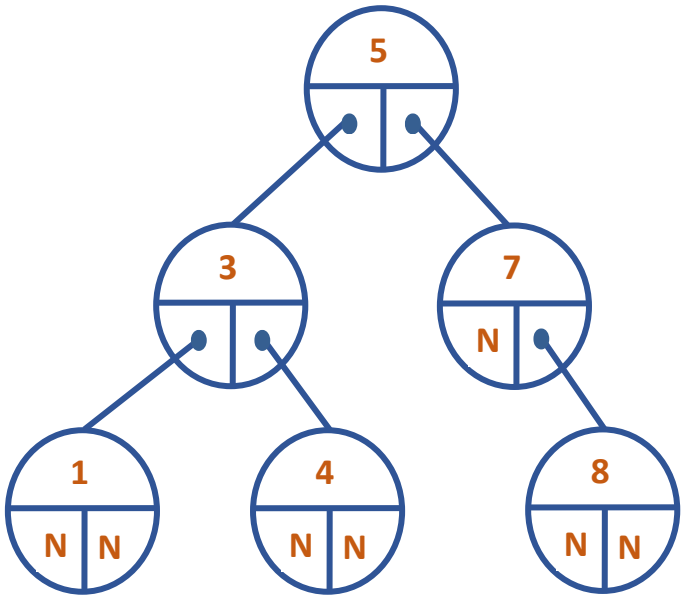


Figure 5: Linked Representation of BST in Figure 4

```
//BST Linked Representation
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
        int info;
        struct node *left,*right;
}NODE;
```

```
typedef struct tree
{
	NODE *root;
}TREE;

void init(TREE *pt)
{
	pt->root=NULL;
}

void creat(TREE *pt)
{
	NODE *temp,*p,*q;
	int wish;

	printf("Enter the root info\n");
	pt->root=(NODE*)malloc(sizeof(NODE));
	scanf("%d",&pt->root->info);

	pt->root->left=NULL;
	pt->root->right=NULL;
	do{
		printf("Enter an element\n");
		temp=(NODE*)malloc(sizeof(NODE));
		scanf("%d",&temp->info);
		temp->left=NULL;
		temp->right=NULL;
		q=NULL;
		p=pt->root;

		while(p!=NULL)
		{
			q=p;
			if(temp->info < p->info)
				p=p->left;
			else
				p=p->right;
		}
		if(temp->info < q->info)
			q->left=temp;
		else
			q->right=temp;
```

```
                printf("Do you wish to add another? 1/0\n");
                scanf("%d",&wish);
        }while(wish);
}

void intr(NODE* p)
{
        if(p!=NULL)
        {
                intr(p->left);
                printf("%d ",p->info);
                intr(p->right);
        }
}

void intrav(TREE *pt)
{
        intr(pt->root);
}

int main()
{
        TREE tobj;

        creat(&tobj);
        intrav(&tobj);

        return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Binary Search Tree (BST): Deletion

Dr. Shylaja S S
Ms. Kusuma K V

Deletion of a Node in Binary Search Tree

Consider the following 3 cases for deletion of a node in Binary Search Tree so that even after the node is deleted the BST property is preserved.

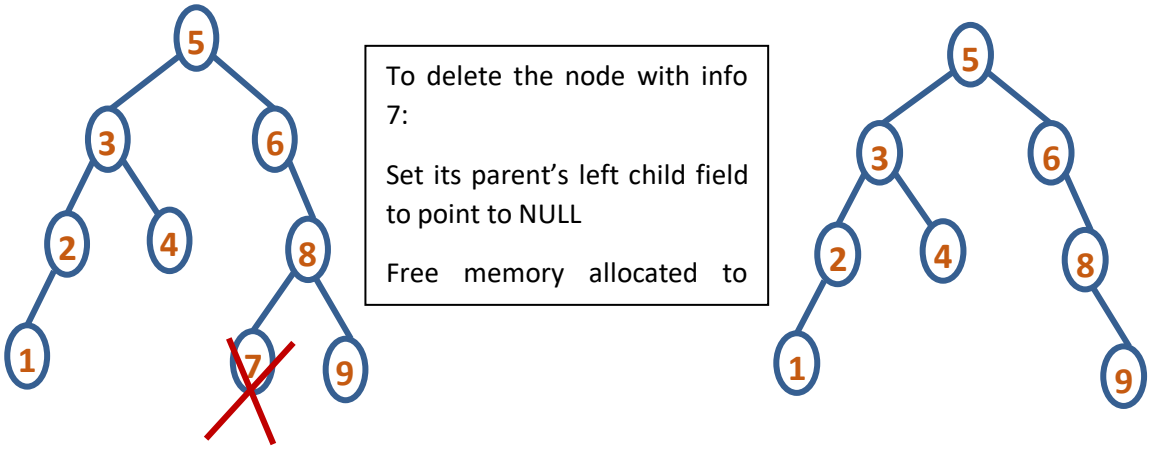
Case 1: Node with no child (leaf node)

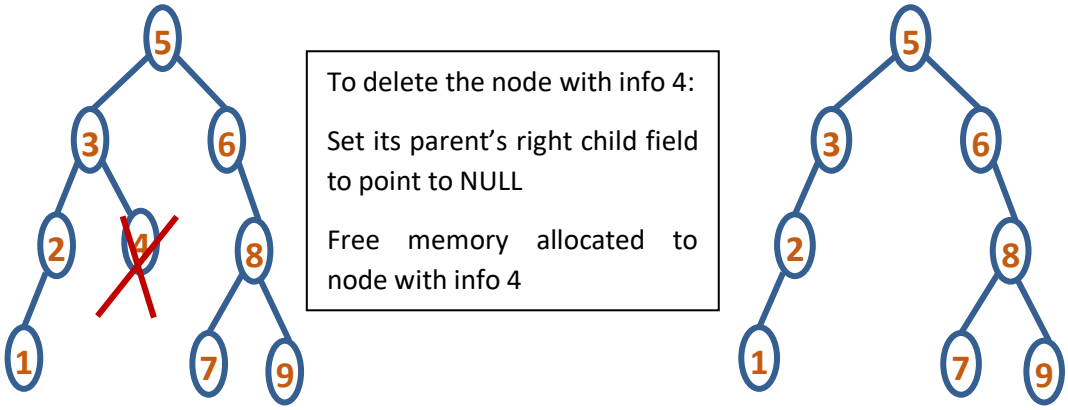
Case 2: Node with 1 child

Case 3: Node with 2 children

Case 1: Deletion of a leaf node

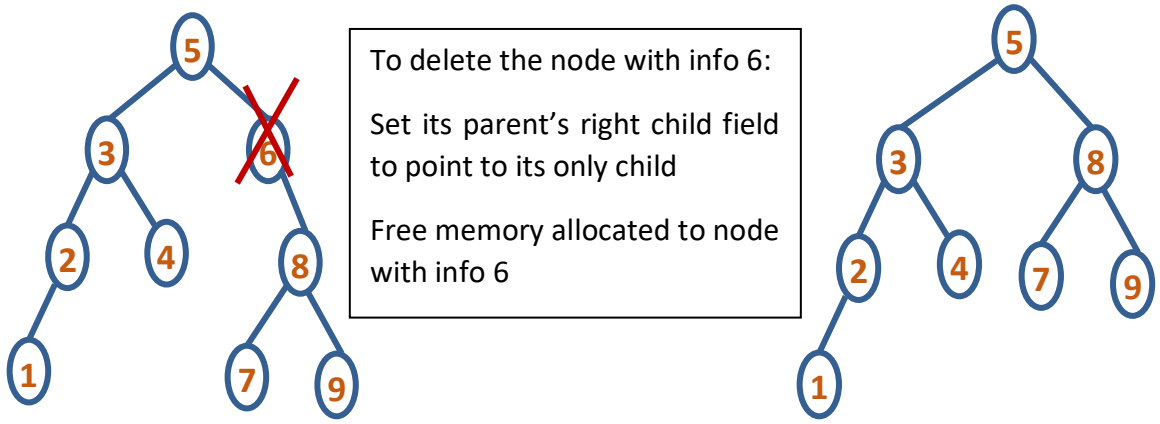
Set its parent's corresponding child field to point to NULL. Free memory allocated to the node.

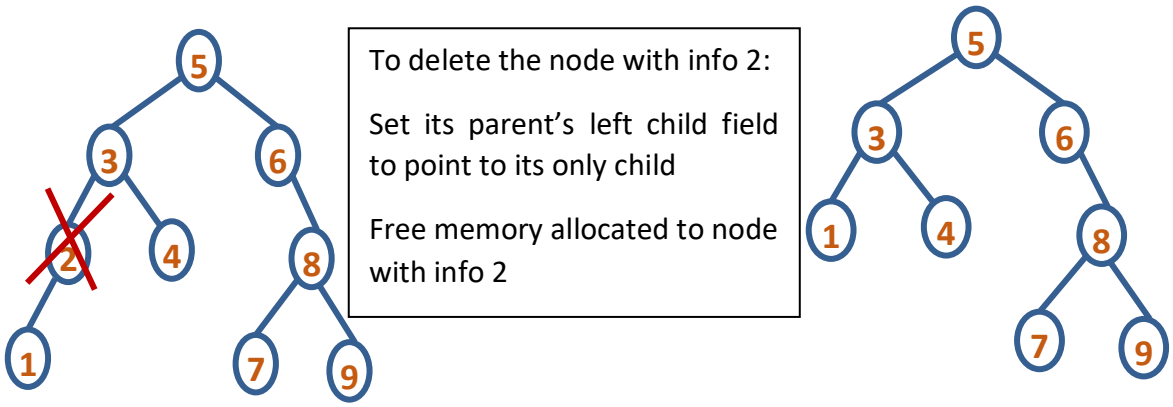




Case 2: Deletion of a node with one child

Connect the node's parent with node's child node





Case 3: Deletion of a node with 2 children

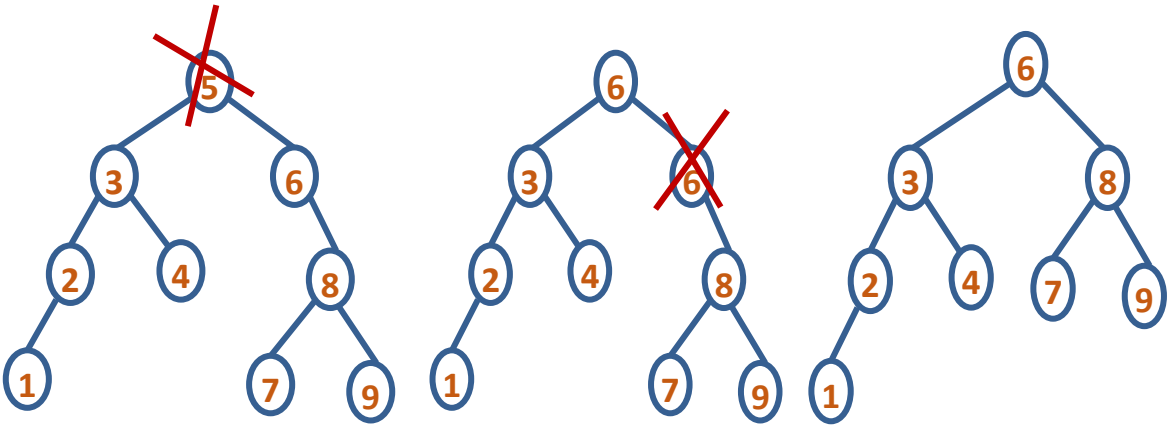
This case can be handled in 2 ways:

Way 1: Replace the node to be deleted with its inorder successor (Smallest in its Right subtree or Leftmost in its Right subtree) and delete that inorder successor.

Way2: Replace the node to be deleted with its inorder predecessor (Largest in its Left subtree) and delete that inorder predecessor.

For the deletion of inorder successor/predecessor, case 3 gets reduced to either case 1 or case 2. We know how to solve case1 and case2.

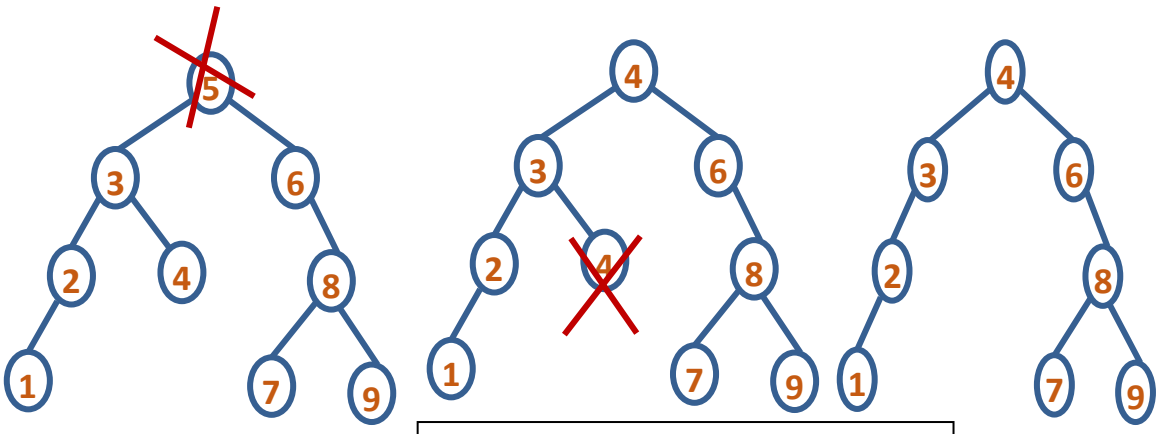
To delete node with info 5 (Way 1: Replace with inorder successor)



Replace 5 with its inorder successor

Now delete that inorder successor
Now case3 has got changed to case2 (In general may change to case2 or case1)

To delete node with info 5 (Way 2: Replace with inorder predecessor)



Replace 5 with its inorder predecessor

Now delete that inorder predecessor
Now case3 has got changed to case1 (In general may change to case2 or case1)

Note: For implementation we shall consider replacing with inorder successor.

```
//BST Deletion
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
        int info;
        struct node *left,*right;
}NODE;
typedef struct tree
{
        NODE *root;
}TREE;

void init(TREE  *pt)
{
        pt->root=NULL;
}

NODE* createNode(int e)
{
        NODE *temp=malloc(sizeof(NODE));
        temp->left=NULL;
        temp->right=NULL;
        temp->info=e;
        return temp;
}

void create(TREE *pt)
{
        NODE *p,*q;
        int e, wish;

        printf("Enter info\n");
        scanf("%d",&e);

        pt->root=createNode(e);

        do{
                printf("Enter info\n");
                scanf("%d",&e);

                q=NULL;
```

```
                p=pt->root;

                while(p!=NULL)
                {
                        q=p;

                        if(e < p->info)
                                p = p->left;
                        else
                                p = p->right;
                }

                if(e < q->info)
                        q->left = createNode(e);
                else
                        q->right = createNode(e);

                printf("Do you wish to continue\n");
                scanf("%d",&wish);
        }while(wish);
}

void io(NODE* r)
{
        if(r!=NULL)
        {
                io(r->left);
                printf("%d ",r->info);
                io(r->right);
        }
}

void inorder(TREE *pt)
{
        io(pt->root);
}
```

```
NODE* delNode(NODE *r,int ele)
{
        NODE *temp,*p;

        if(r==NULL)
                return r;

        if(ele < r->info)
                r->left = delNode(r->left,ele);
        else if(ele > r->info)
                r->right =  delNode(r->right,ele);
        else
        {
                if(r->left == NULL)                     //1 right child or No children
                {
                        temp=r->right;
                        free(r);
                        return temp;
                }
                else if(r->right == NULL)               //1 left  child or No children
                {
                        temp=r->left;
                        free(r);
                        return temp;
                }
                else
                {       //Node to be deleted has both children
                        //Finding p's leftmost child which is the inorder successor
                        p=r->right;
                        while(p->left != NULL)
                                p=p->left;

                        r->info=p->info;
                        r->right=delNode(r->right, p->info);
                }
        }
        return r;
}
void deleteNode(TREE *pt,int e)
{
        pt->root=delNode(pt->root,e);
}
```

```
int main()
{
        int e;
        TREE t;
        init(&t);
        create(&t);
        inorder(&t);
        printf("Enter the element to be deleted\n");
        scanf("%d",&e);
        deleteNode(&t,e);
        printf("After deletion\n");
        inorder(&t);

        return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
BST: Implementation using Arrays

Dr. Shylaja S S
Ms. Kusuma K V

Implicit Array Representation of BST

In the implicit array representation, an array element is allocated whether or not it serves to contain a node of a tree. We must, therefore, flag unused array elements as nonexistent, or null, tree nodes. This may be accomplished by adding a logical flag field, used, to each node. Each node then contains two fields: info and used.

```
typedef struct node
{
        int info;
        int used;
}NODE;
```

NODE[p].used is TRUE if node p is not a null node and FALSE if it is a null node.

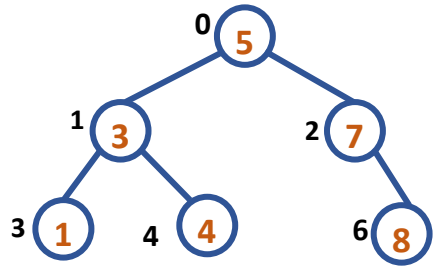


Figure 2: Binary Search Tree

info	5	3	7	1	4		8
used	1	1	1	1	1	0	1
Position: p	0	1	2	3	4	5	6

Array Representation of
Binary Search Tree in Figure 2

//BST Array Implementation

```
#include<stdio.h>

typedef struct tree_node
{
        int info;
        int used;
}TREE;

#define MAXNODES 50

void init(TREE t[MAXNODES])
{
        for(int i=0;i<MAXNODES;i++)
                t[i].used=0;
}
```

```
void create(TREE *bst)
{
	int ele, wish;

	printf("Enter the root element\n");
	scanf("%d",&bst[0].info);
	bst[0].used=1;

	do{
		printf("Enter an element\n");
		scanf("%d",&ele);

		int p=0;

		while(p<MAXNODES && bst[p].used)
		{
			if(ele<bst[p].info)
				p=2*p+1;
			else
				p=2*p+2;
		}

		if(p>=MAXNODES)
			printf("Insertion not possible\n");
		else
		{
			bst[p].info=ele;
			bst[p].used=1;
		}
		printf("Do you wish to add another\n");
		scanf("%d",&wish);
	}while(wish);
}
void inorder(TREE* bst, int r)
{
	if(bst[r].used)
	{
		inorder(bst,2*r+1);
		printf("%d ",bst[r].info);
		inorder(bst,2*r+2);
	}
}
```

```
int main()
{
        TREE bst[MAXNODES];

        init(bst);
        create(bst);
        inorder(bst,0);
        return 0;
}
```

Note that under this representation it is always required to check that the range (NUMNODES) has not been exceeded whenever we move down the tree.

Array Implementation is suitable for a complete binary tree. Otherwise there will be many vacant positions in between. In such cases we may look into an alternate representation of tree nodes, i.e., the Linked representation, which makes use of the left and right pointers along with the info field.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Binary Trees: Traversal

Dr. Shylaja S S
Ms. Kusuma K V

Binary Tree Traversal:

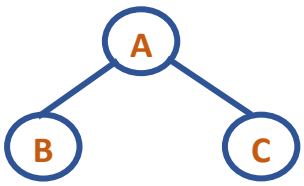
Traversal is the process of moving through all the nodes in a binary tree and visiting each one in turn. The action taken when each node is visited depends on the application; here we shall print the content of each node on visiting.

Tree being a non linear data structure, there are many different ways in which we could traverse all the nodes. When we write an algorithm to traverse a binary tree, we shall almost always wish to proceed so that the same rules are applied at each node, and we thereby adhere to a general pattern.

At a given node, then, there are three tasks we shall wish to do in some order. They are:
1) Visiting a node, let us denote it by V
2) Traversing the left subtree, let us denote it by L
3) Traversing the right subtree, let us denote it by R
The key distinction in traversal orders is to decide if we are to visit the node itself before traversing either subtree, between the subtrees, or after traversing both the subtrees.
The three tasks can be done in six different ways: VLR, LVR, LRV, VRL, RVL, RLV.

By Standard convention, these 6 ways are reduced to three by permitting only the ways in which the left subtree is traversed before the right. The three mirror images are clearly similar. The three ways with left subtree before right subtree are given the following names:
VLR – Preorder, LVR – Inorder, LRV – Postorder
These three names are chosen according to the step at which the given node is visited. With prorder traversal, the node is visited before the subtrees, with inorder traversal, node is visited between the left and right subtree, and with postorder traversal, the node is visited after both the subtrees.

Eg1: Consider a simple example of a Binary Tree shown below:



Preorder Traversal: A B C
Inorder Traversal: B A C
Postorder Traversal: B C A

Binary Tree Recursive Traversal

Preorder Traversal:

- Root Node is visited before the subtrees
- Left subtree is traversed in preorder
- Right subtree is traversed in preorder

```
void preorder(NODE* root)
{
        if(root!=NULL)
        {
                printf("%d ",root->info);
                preorder(root->left);
                preorder(root->right);
        }
}
```

Inorder Traversal:

- Left subtree is traversed in Inorder
- Root Node is visited
- Right subtree is traversed in Inorder

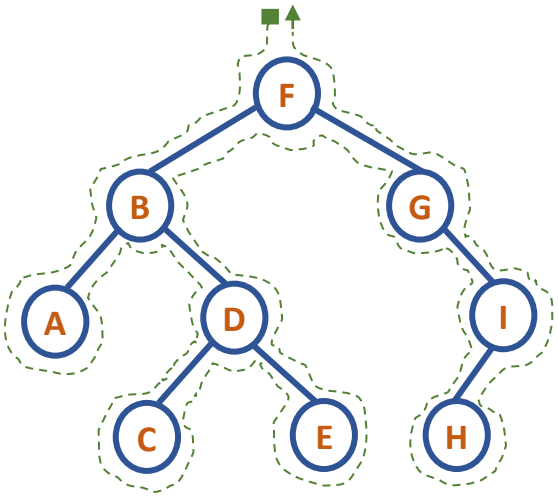
```
void inorder(NODE* root)
{
        if(root!=NULL)
        {
                inorder(root->left);
                printf("%d ",root->info);
                inorder(root->right);
        }
}
```

Postorder Traversal:

- Left subtree is traversed in postorder
- Right subtree is traversed in postorder
- Root Node is visited

```
void postorder(NODE* root)
{
        if(root!=NULL)
        {
                postorder(root->left);
                postorder(root->right);
                printf("%d ",root->info);
        }
}
```

Eg2: Write the Preorder, Inorder and Postorder Traversal for the Binary Tree given below:



Preorder Traversal: F B A D C E G I H
Inorder Traversal: A B C D E F G H I
Postorder Traversal: A C E D B H I G F

Binary Tree Iterative Traversal

1) Inorder Traversal

```
//Iterative Inorder Traversal
iterativeInorder(root)
 s = emptyStack
current = root
do {
                while(current != null)
                {       /* Travel down left branches as far as possible saving pointers to
nodes passed in the stack*/
                        push(s, current)
                        current = current->left
                } //At this point, the left subtree is empty
                poppedNode = pop(s)
                print poppedNode ->info                 //visit the node
                current = poppedNode ->right            //traverse right subtree
} while(!isEmpty(s) or current != null)
```

2) Preorder Traversal

```
//Iterative Preorder Traversal
iterativePreorder(root)
current=root
if (current == null)
   return
s = emptyStack
push(s, current)
```

```
while(!isEmpty(s)) {
        current = pop(s)
        print current->info
        //right child is pushed first so that left is processed first
        if(current->right !=NULL)
                push(s, current->right)
        if(current->left !=NULL)
                push(s, current->left)
}
```

3) Postorder Traversal

```
//Iterative Postorder Traversal
iterativePostorder(root)
s1 = emptyStack
s2 = emptyStack
push(s1, root)

while(!isEmpty(s1)) {
        current = pop(s1)
        push(s2,current)
        if(current->left !=NULL)
                push(s1, current->left)
        if(current->right !=NULL)
                push(s1, current->right)
}
while(!isEmpty(s2)) {          //Print all the elements of stack2
        current = pop(s2)
        print current->info
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Threaded Binary Search Tree

Dr. Shylaja S S
Ms. Kusuma K V

Threaded binary search tree and its implementation

```
//Iterative Inorder Traversal Revisited
iterativeInorder(root)
 s = emptyStack
p = root
do {
                while(p != null)
                {       /* Travel down left branches as far as possible saving pointers to
nodes passed in the stack*/
                        push(s, p)
                        p = p->left
                } //At this point, the left subtree is empty
                p = pop(s)
                print p ->info          //visit the node
                p = p ->right           //traverse right subtree
} while(!isEmpty(s) or p != null)
```

If we examine the iterativeInorder function to find out the reason for the need of stack, we see that the stack is popped when *p* equals NULL. This happens in two cases:
case (i): The **while** loop is exited after having been executed one or more times. This implies that the program has travelled down left branches until it reached a NULL pointer, stacking a pointer to each node as it was passed. Thus, the top element of the stack is the value of *p* before it became NULL. If an auxiliary pointer *q* is kept one step behind *p*, the value of *q* can be used directly and need not be popped.

case (ii): The **while** loop is skipped entirely. This occurs after reaching a node with an empty right subtree, executing the statement *p = p -> right*, and returning to repeat the body of the **do while** loop. At this point, top points to the node whose left subtree was just traversed. If we had not stacked the pointer to each node as it was passed then we would have lost our way in this case. So, a node with an empty right subtree, instead of containing a NULL pointer in its right field, suppose it contained a pointer to the node that would be on top of the stack at that point in the algorithm (that is, a pointer to its inorder successor), then there would no longer be a need for the stack, since the last node visited during traversal of a left subtree points directly to its inorder successor. Such a pointer is called a **thread** and must be differentiable from a tree pointer that is used to link a node to its left or right subtree.

A binary tree in which the right pointer of a node, points to the inorder successor if in case it is not pointing to the child is called **Right In-Threaded Binary Tree**. Figure 1 shows a binary search tree and Figure 2 shows the corresponding right in-threaded binary search tree. The threads are drawn in dotted lines to differentiate them from tree pointers. Note that the rightmost node in the tree still has a NULL right pointer, since it has no inorder successor.

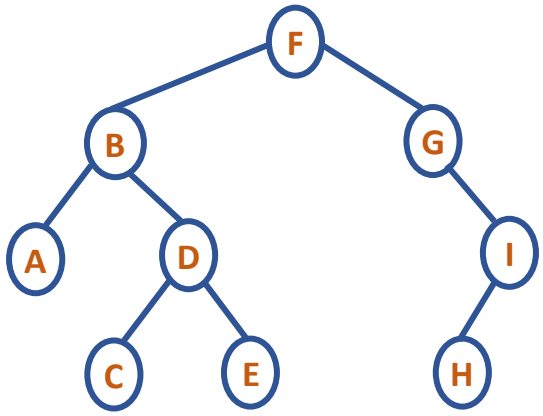


Figure 1: Binary Search Tree

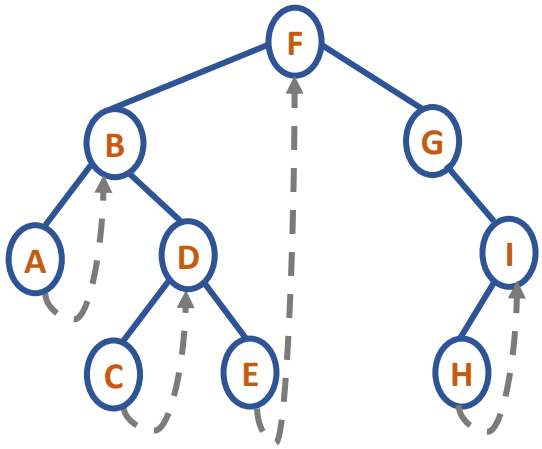


Figure 2: Right In-Threaded Binary Search Tree

Nodes with Right Pointer NULL	A	C	E	H	I
Inorder Successor	B	D	F	I	-

Inorder Traversal:
A B C D E F G H I

A binary tree in which the left pointer of a node, points to the inorder predecessor if in case it is not pointing to the child is called **Left In-Threaded Binary Tree**. A binary tree in which the right and left pointer of a node, points to the inorder successor and inorder predecessor respectively if in case it is not pointing to the right and left child respectively is called **In-Threaded Binary Tree**. Figure 3 and Figure 4 shows the left in-threaded and in-threaded binary search tree respectively, corresponding to the binary search tree in Figure 1.

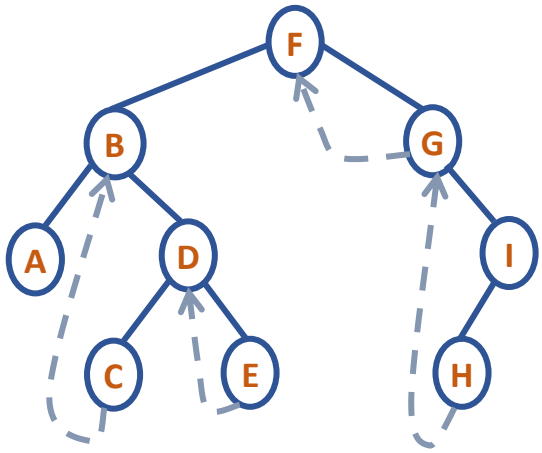


Figure 3: Left In-Threaded Binary Search Tree

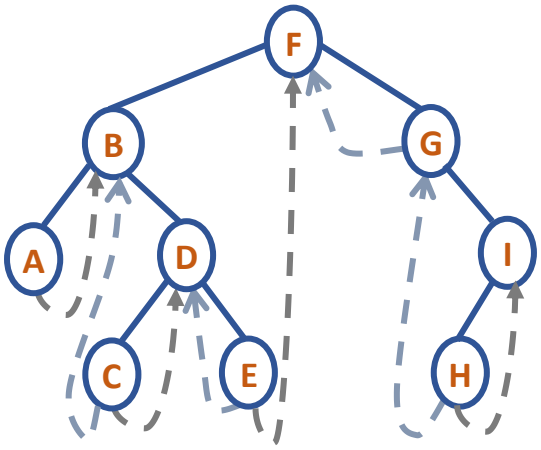


Figure 4: In-Threaded Binary Search Tree

Nodes with Left Pointer NULL	A	C	E	G	H
Inorder Predecessor	-	B	D	F	G

Inorder Traversal:
A B C D E F G H I

To implement a right in-threaded binary search tree under the dynamic node implementation of a binary tree, an extra logical field, rthread is included within each node to indicate whether or not its right pointer is a thread. Note that, for consistency, the rthread field of the rightmost node of a tree (that is, the last node in the tree's inorder traversal) is also set to TRUE, although its right field remains NULL. Such a node is declared as follows:

```
typedef struct node
{
        int info;
        bool rthread;           //TRUE if right is NULL or a non-NULL thread
        struct node *left;      //Pointer to left child
        struct node *right;     //Pointer to right child
}NODE;
```

//C program to demonstrate construction and inorder traversal of right in-threaded binary search tree

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

typedef struct node
{
        int info;
        bool rthread;
        struct node *left;
        struct node *right;
}NODE;

typedef struct tree
{
        NODE *root;
}TREE;

void init(TREE *pt)
{
        pt->root=NULL;
}
```

```
NODE* createNode(int e)
{
        NODE* temp=malloc(sizeof(NODE));
        temp->info=e;
        temp->left=NULL;
        temp->right=NULL;
        temp->rthread=1;
        return temp;
}

void setLeft(NODE* q,int e)             //set node to left of queue
{
        NODE* temp=createNode(e);
        q->left=temp;
        temp->right=q;
}

void setRight(NODE* q,int e)
{
        NODE* temp=createNode(e);
        temp->right=q->right;                   //thread
        q->right=temp;
        q->rthread=0;
}

void inOrder(TREE *t)
{
        NODE *p=t->root;
        NODE *q;

        do{
                q=NULL;
                while(p!=NULL)
                {
                        q=p;
                        p=p->left;
                }
                if(q!=NULL)
                {
                        printf("%d ",q->info);
                        p=q->right;
```

```
                        while(q->rthread && p!=NULL)
                        {
                                printf("%d ",p->info);
                                q=p;
                                p=p->right;
                        }
                }
        }while(q!=NULL);
}

void create(TREE *pt)
{
        NODE *p,*q;
        int e,wish;

        printf("Enter root info\n");
        scanf("%d",&e);

        pt->root=createNode(e);

        do{
                printf("Enter info\n");
                scanf("%d",&e);

                p=pt->root;
                q=NULL;

                while(p!=NULL)
                {
                        q=p;

                        if(e<p->info)
                                p=p->left;
                        else{
                                if(p->rthread)
                                        p=NULL;
                                else
                                        p=p->right;
                        }
                }
                if(e < q->info)
                        setLeft(q,e);
```

```
                else
                        setRight(q,e);

                printf("Do you wish to add another element\n");
                scanf("%d",&wish);
        }while(wish);
}

int main()
{
        TREE t;
        init(&t);
        create(&t);
        inOrder(&t);

        return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Expression Trees

Dr. Shylaja S S
Ms. Kusuma K V

Expression Tree

An expression tree is built up from the simple operands and operators of an (arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree and the operators as internal nodes.

For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator, and the right subtree contains everything in the right operand.

For a unary operator, one of the two subtrees will be empty. We traditionally write some unary operators to the left of their operands, such as '-'(unary negation) or the standard functions like log() and cos(). Other unary operators are written on the right, such as the factorial function ()! or the function that takes the square of a number, () ^{2}. Sometimes either side is permissible, such as the derivative operator, which can be written as d/dx on the left, or as ()' on the right, or the incrementing operator ++ in the C language (where the actions on the left and right are different). If the operator is written on the left, then in the expression tree we take its left subtree as empty, so that the operand appear on the right side of the operator in the tree, just as they do in the expression. If the operator appears on the right, then its right subtree will be empty, and the operands will be in the left subtree of the operator.

The expression trees of few simple expressions are shown in Figure 1.

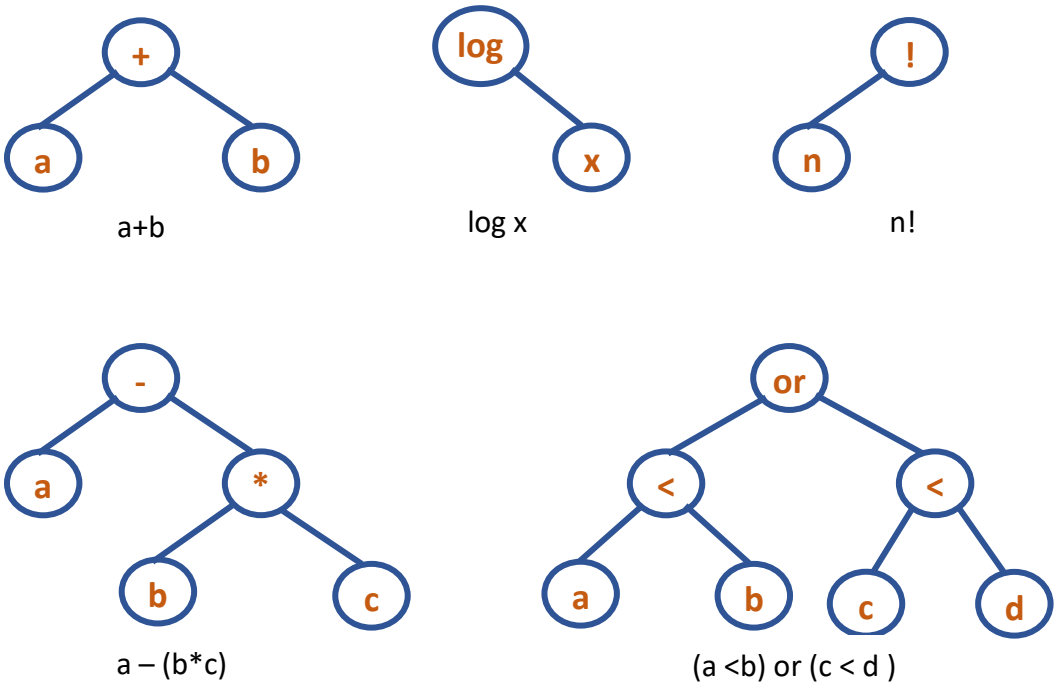


Figure 1: Expression Trees

Construction and Evaluation of an Expression Tree

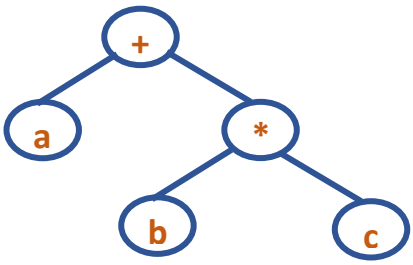
1) Scan the postfix expression till the end, one symbol at a time
    a) Create a new node, with symbol as info and left and right link as NULL
    b) If symbol is an operand, push address of node to stack
    c) If symbol is an operator
        i) Pop address from stack and make it right child of new node
        ii) Pop address from stack and make it left child of new node
        iii) Now push address of new node to stack
2) Finally, stack has only element which is the address of the root of expression tree

//Binary Expression Tree

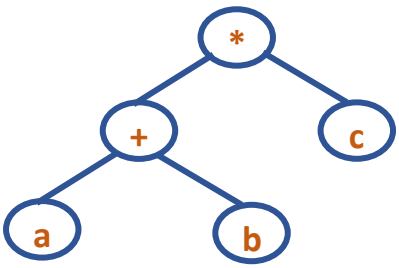
```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#define MAX 50

typedef struct node
{
        char info;
        struct node *left,*right;
}NODE;

typedef struct tree
{
        NODE* root;
}TREE;

typedef struct stack
{
        NODE* s[50];
        int top;
}STACK;

void init_trr(TREE *pt)
{
        pt->root=NULL;
}

void init_stack(STACK *ps)
{
        ps->top=-1;
}
```

```
int push(STACK *ps,NODE* e)
{
	if(ps->top==MAX-1)
		return 0;
	ps->top++;
	ps->s[ps->top]=e;

	return 1;
}

NODE* pop(STACK *ps)
{
	//Assumption: empty condition checked before entering the pop
	NODE *t=ps->s[ps->top];
	ps->top--;

	return t;
}

//Expression Tree Evaluation
float eval(NODE* r)
{
	float res;
	switch(r->info)
	{
		case '+': return (eval(r->left)+eval(r->right));
				break;
		case '-': return (eval(r->left)-eval(r->right));
				break;
		case '*': return (eval(r->left)*eval(r->right));
				break;
		case '/': return (eval(r->left)/eval(r->right));
				break;
		default: return r->info - '0';
	}
	return res;
}

float eval_tree(TREE *pt)
{
	return eval(pt->root);
}
```

```
int main()
{
        char postfix[MAX];
        STACK sobj;
        TREE tobj;
        NODE *temp;
        init_stack(&sobj);
        printf("Enter a valid postfix expression\n");
        scanf("%s",postfix);

        int i=0;
        while(postfix[i] != '\0')
        {
                temp = (NODE*) malloc(sizeof(NODE));

                temp->info = postfix[i];
                temp->left=NULL;
                temp->right=NULL;

                if(isdigit(postfix[i]))
                        push(&sobj,temp);
                else
                {
                        temp->right=pop(&sobj);
                        temp->left=pop(&sobj);
                        push(&sobj,temp);
                }
                i++;
         }
        tobj.root=pop(&sobj);
        printf("%f",eval_tree (&tobj));
        return 0;
}
```



(i) a + (b*c)



(ii) (a + b)*c

Figure 2: Expressions and their binary tree representations

Traversal of the binary expression trees in Figure 2 is as follows:

Figure 2(i): Preorder: +a*bc	Inorder: a+b*c	Postorder: abc*+
Figure 2(ii): Preorder: *+abc	Inorder: a+b*c	Postorder: ab+c*

It can be observed that traversing the binary expression tree in preorder and postoder yields the corresponding prefix and postfix form of the infix expression. But we can see that the inorder traversal of the binary expression trees doesn't always yield the infix form of the expression. This is because the binary expression tree does not contain parentheses, since the ordering of the operations is implied by the structure of the tree.

General Expression Tree

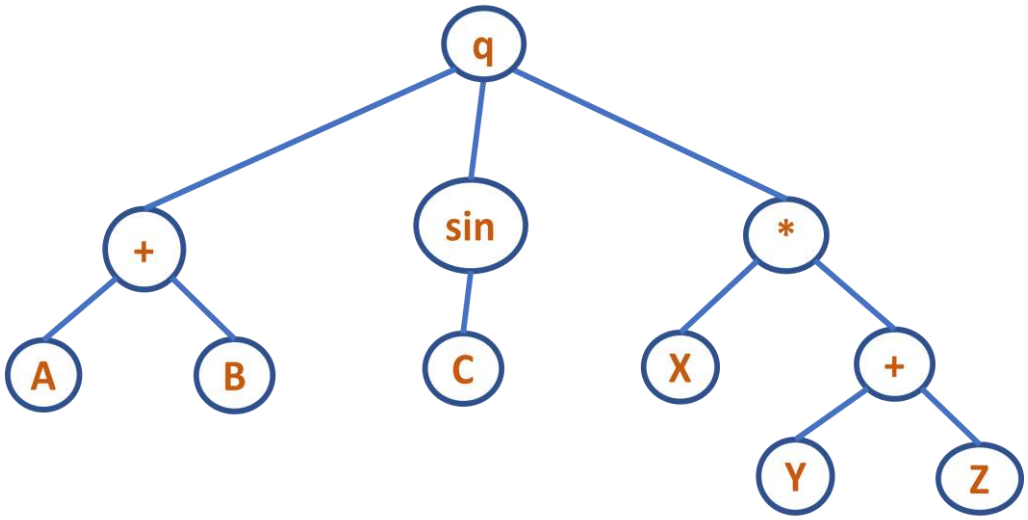


Figure: Tree representation of an arithmetic expression

Here node can be either an operand or an operator

```
struct treenode
{
 short int utype;
  union{
    char operator[MAX];
     float val;
   }info;
  struct treenode *child;
  struct treenode *sibling;
};
typedef struct treenode TREENODE;
```

```
void replace(TREENODE *p)
{
  float val;
  TREENODE *q,*r;
  if(p->utype == operator)
 {
   q = p->child;
   while(q != NULL)
    {
      replace(q);
      q = q->next;
    }
 }
  value = apply(p);
  p->utype = OPERAND;
  p->val = value;
  q = p->child;
  p->child = NULL;
  while(q != NULL)
  {
      r = q;
      q = q->next;
      free(r);
  }
 }
}

float eval(TREENODE *p)
{
 replace(p);
 return(p->val);
 free(p);
 }
```

Constructing a Tree

```
void setchildren(TREENODE *p,TREENODE *list)
{
  if(p == NULL) {
    printf("invalid insertion");
    exit(1);
  }
 if(p->child != NULL) {
   printf("invalid insertion");
   exit(1);
 }
 p->child = list;
}

void addchild(TREENODE *p,int x)
{
 TREENODE *q;
 if(p==NULL)
 {
   printf("void insertion");
   exit(1);
 }
 r = NULL;
 q = p->child;

 while(q != NULL)
 {
   r = q;
   q = q->next;
 }
 q = getnode();
 q->info = x;
 q->next = NULL;

 if(r==NULL)
   p->child=q;
 else
   r->next=q;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

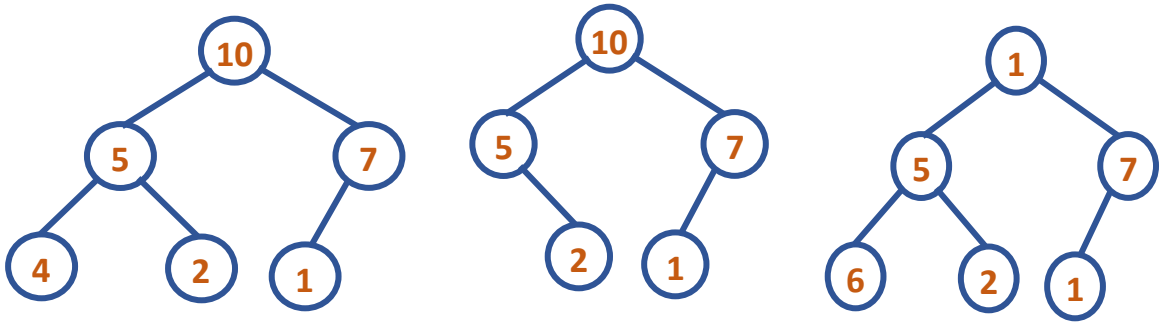
UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Heap: Implementation using arrays

Dr. Shylaja S S
Ms. Kusuma K V

Heap: Concept and Implementation

Definition: A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:
1. The tree's shape requirement - The binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing
2. The parental dominance requirement - The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)



In the above figures, only the left most binary tree is a heap. The binary tree in the middle is not a heap because it doesn't satisfy the shape requirement. The rightmost binary tree is not a heap because it doesn't satisfy the parental dominance requirement.

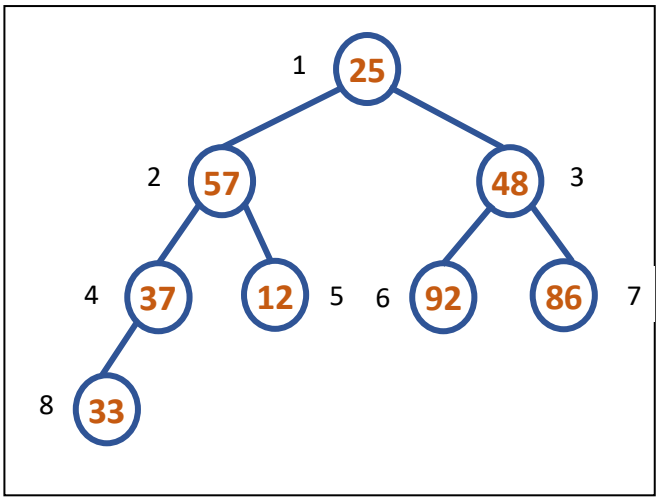
Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$
2. The root of a heap always contains its largest element
3. A node of a heap considered with all its descendants is also a heap
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap.
   In such a representation,
   a) The parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions
   b) The children of a key in the array's parental position i ($1 <= i <= \lfloor n/2 \rfloor$) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i ($2 <= i <= n$) will be in position $\lfloor n/2 \rfloor$

Bottom up Heap Construction for the elements: 25, 57, 48, 37, 12, 92, 86, 33

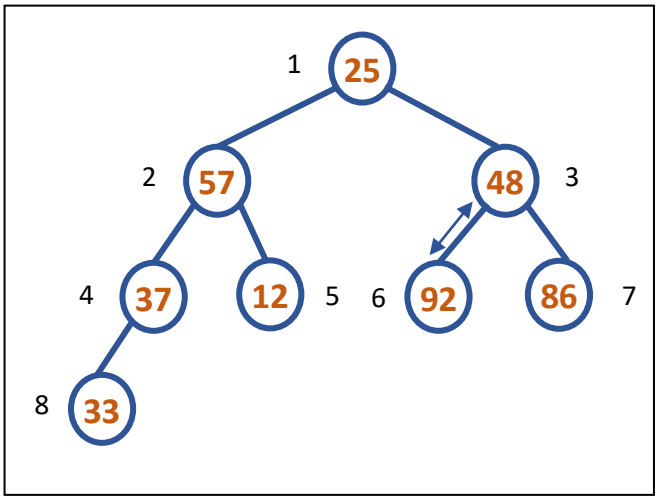
-	25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7	8

At k = 4, v = 37
Compare 37 with its only child 33
37 > 33, it's a heap at k=4



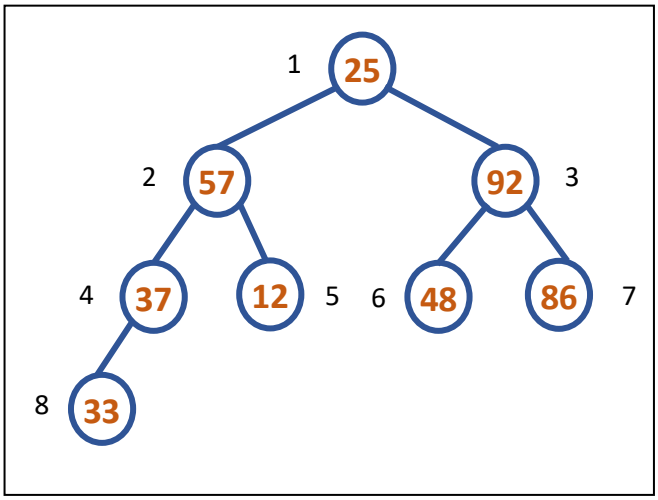
-	25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7	8

At k = 3, v = 48
Largest child: 92
Compare 48 with its largest child
48 < 92, Heapify



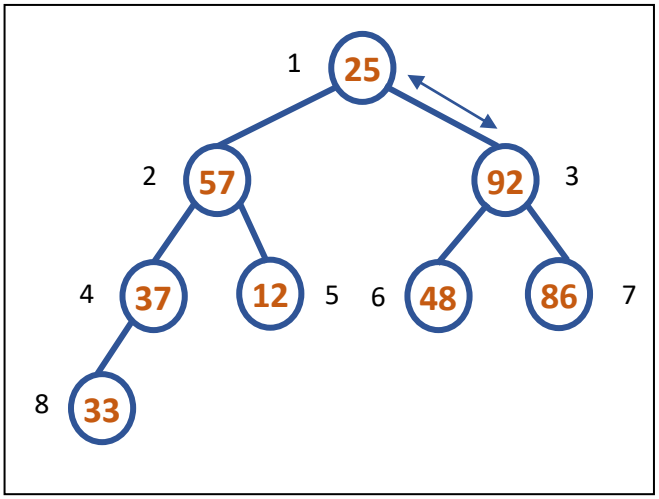
-	25	57	92	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 2, v = 57
Largest child: 37
Compare 57 with its largest child
57 > 37, It's a heap at k=2



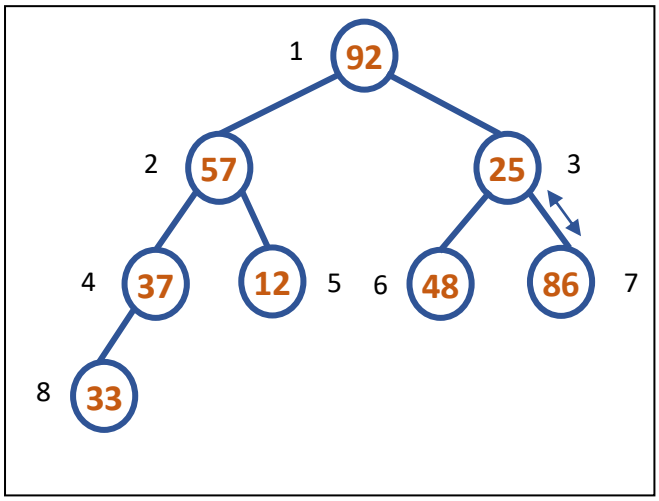
-	25	57	92	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 1, v = 25
Largest child:92
Compare 25 with its largest child
25 < 92, Heapify

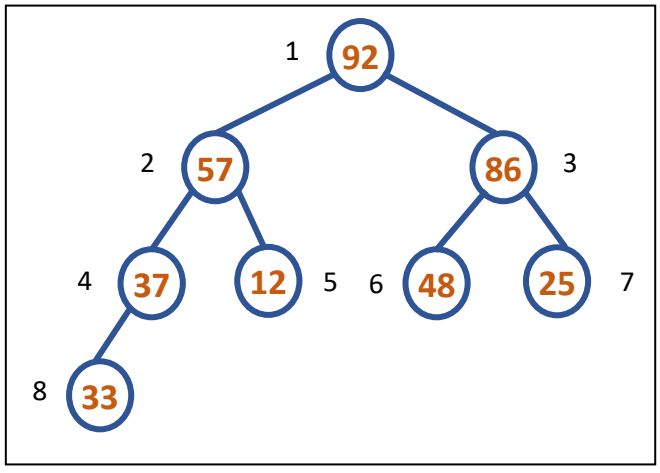


-	92	57	25	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 3, v = 25
Largest child:86
Compare 25 with its largest child
25 < 86, Heapify



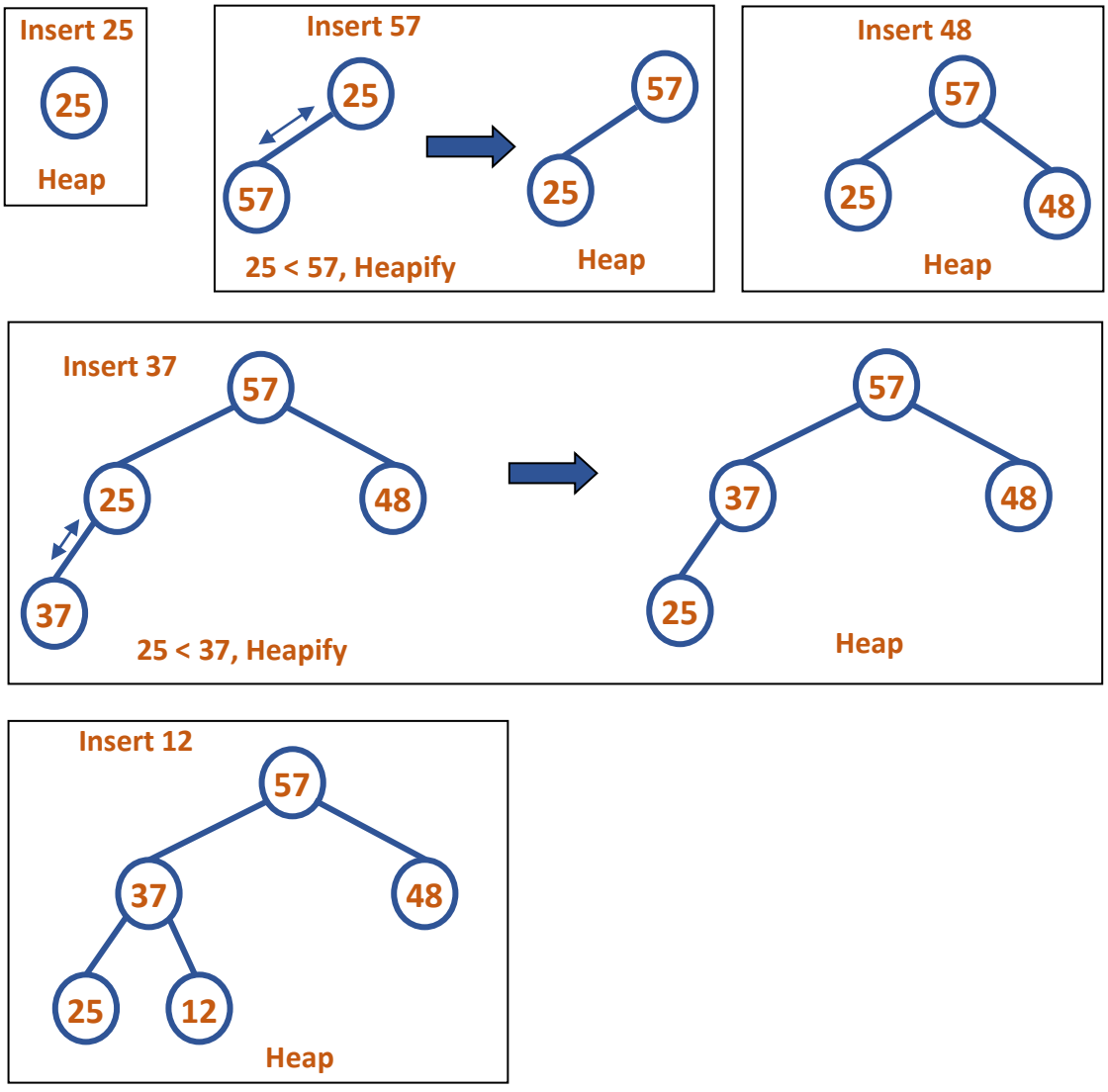
-	92	57	86	37	12	48	25	33
0	1	2	3	4	5	6	7	8

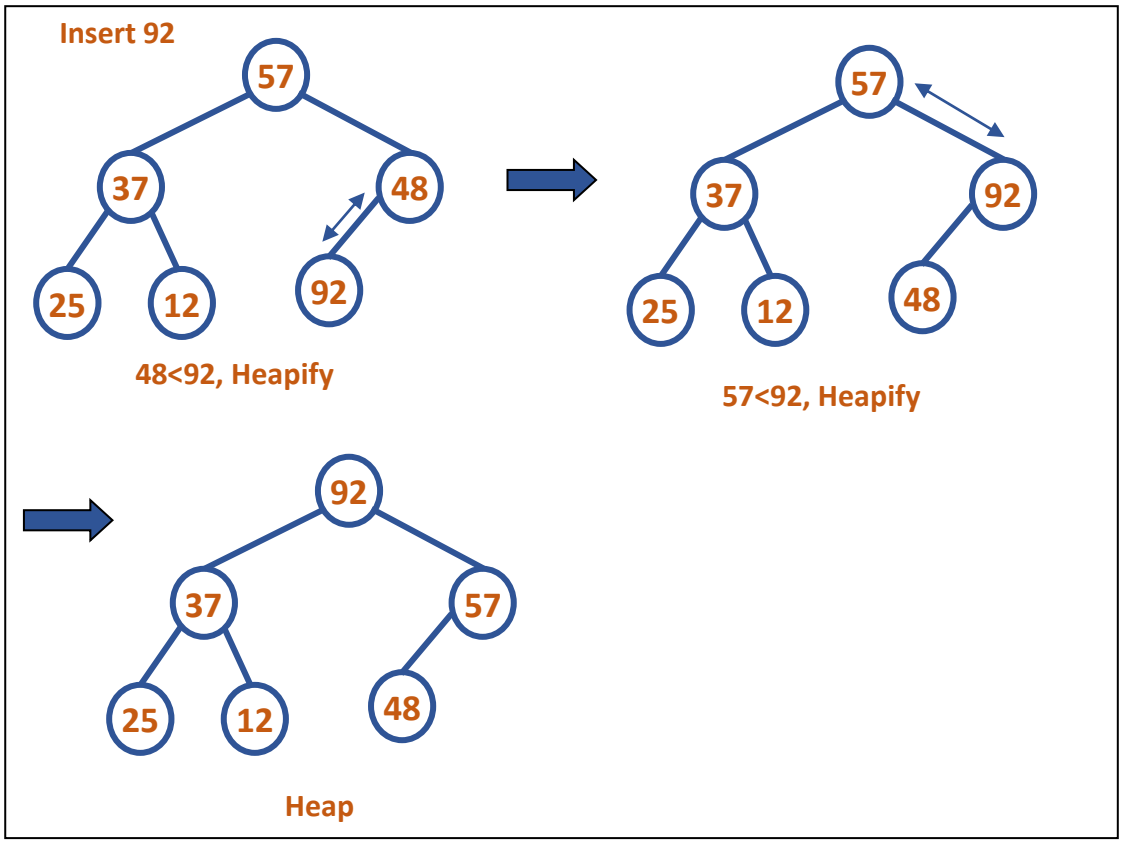


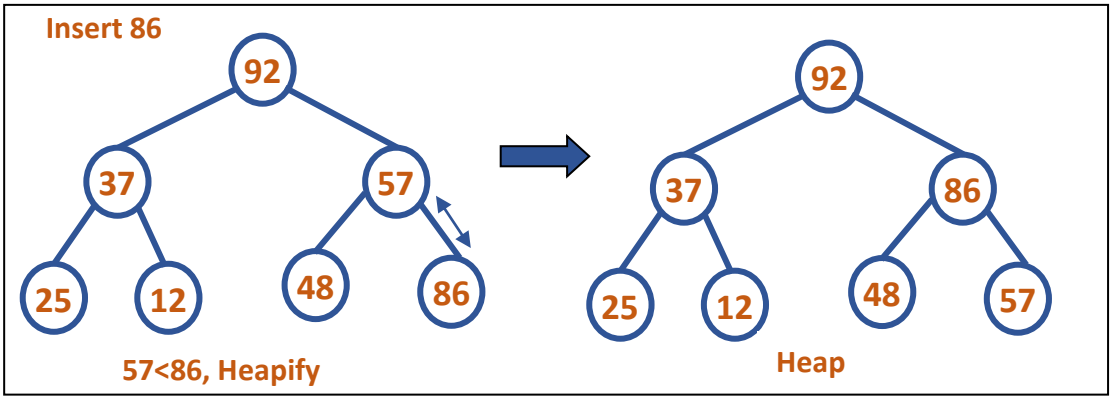
Bottom up Heap Constructed

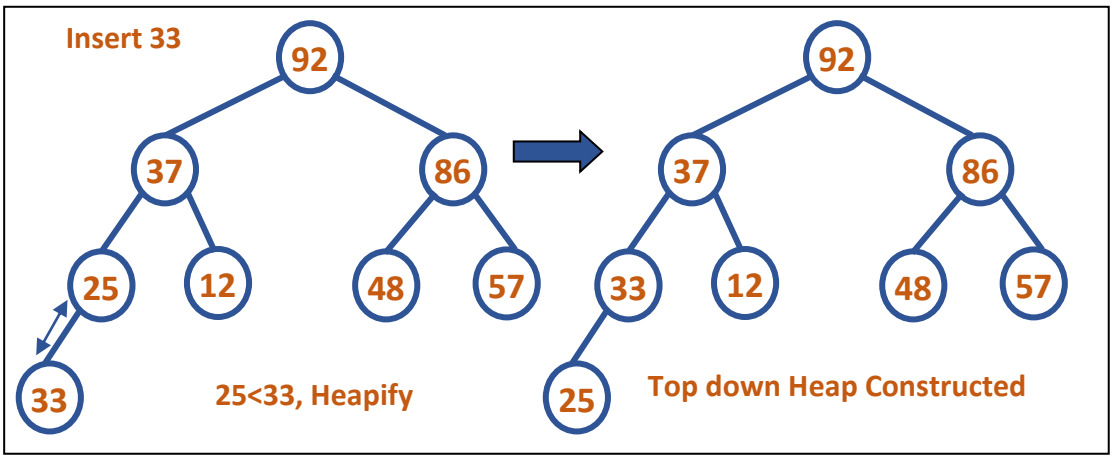
```
ALGORITHM HeapBottomUp(H[1…n])
//Constructs a heap from the elements of a given array by the bottom-up algorithm
//Input: An array H[1…n] of orderable items
//Output: A heap H[1…n]
for i = ⌊n/2⌋ downto 1 do
        k = i; v=H[k]; heap = false
        while not heap and 2*k ≤ n do
                j = 2*k
                if j < n                        //there are two children
                   if H[j] < H[j+1]   j = j+1
                if v ≥ H[j]
                   heap = true
                else H[k] = H[j];   k = j
        H[k] = v
```

Top down Heap Construction for the elements: 25, 57, 48, 37, 12, 92, 86, 33




Insert 92
57
37
48
25
12
92
48<92, Heapify
57
37
92
25
12
48
57<92, Heapify
92
37
57
25
12
48
Heap


Insert 86
92
37
57
25
12
48
86
57<86, Heapify
92
37
86
25
12
48
57
Heap


Insert 33
92
37
86
25
12
48
57
33
25<33, Heapify
92
37
86
33
12
48
57
25
Top down Heap Constructed

Top down Heap Construction

1. First, attach a new node with key K in it after the last leaf of the existing heap
2. Then sift K up to its appropriate place in the new heap as follows
3. Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap);
4. Otherwise, swap these two keys and compare K with its new parent.
5. This swapping continues until K is not greater than its last parent or it reaches the root.
6. In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get K's value.

```
//C program to demonstrate top-down heap construction and heap sort
#include<stdio.h>
#define MAX 50
int main()
{
 int a[MAX];
 int i,c,p,n,elt;

 printf("enter the number of elements\n");
 scanf("%d",&n);

 printf("enter the elements\n");
 for(i=0;i<n;i++)
  scanf("%d",&a[i]);

 //Heapify
 for(i=1;i<n;i++)
  {
    elt=a[i];
    c=i;
    p=(c-1)/2;
   while(c>0 && a[p]<elt)
     {
       a[c]=a[p];
       c=p;
       p=(c-1)/2;
     }
   a[c]=elt;
  }
```

```
//display heapified elements
printf("\nElements of heap:\n");
for(i=0;i<n;i++)
 printf("%d ",a[i]);

//Heap sort
for(i=n-1;i>0;i--)
 {
  elt=a[i];
  a[i]=a[0];
  p=0;
 if(i==1)
  c=-1;
 else
  c=1;
 if(i>2 && a[2]>a[1])
  c=2;
 while(c>=0 && elt<a[c])
  {
   a[p]=a[c];
   p=c;
   c=2*p+1;
   if(c+1<=i-1 && a[c]<a[c+1])
    c=c+1;
   if(c>i-1) c=-1;
   }
  a[p]=elt;
 }

printf("\nSorted elements(Heap sort):\n");
for(i=0;i<n;i++)
 printf("%d ",a[i]);

printf("\n");

return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
N-ary Trees and Forest

Dr. Shylaja S S
Ms. Kusuma K V

Tree: is a Non Linear Data Structure

Definition: Finite nonempty set of elements

- One element is the root
- Remaining elements are partitioned into m≥0 disjoint subsets each of which is itself a tree

Ordered Tree: a tree in which subtrees of each node form an ordered set

- In such a tree we define first, second, ..., last child of a particular node
- First child is called the oldest child and last child the youngest child

Figure 1 shows an ordered tree with A as its root, B is the oldest child and D is the youngest child of A. E is the oldest and F is the youngest child of B.

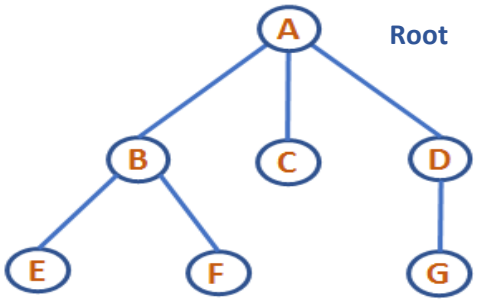


Figure 1: Ordered Tree

Note: Here on we will refer an ordered tree as just a tree

n-ary tree: A rooted tree in which each node has no more than n children.

A binary tree is an n-ary tree with n=2. Figure 2 shows an n-ary tree with n=5.

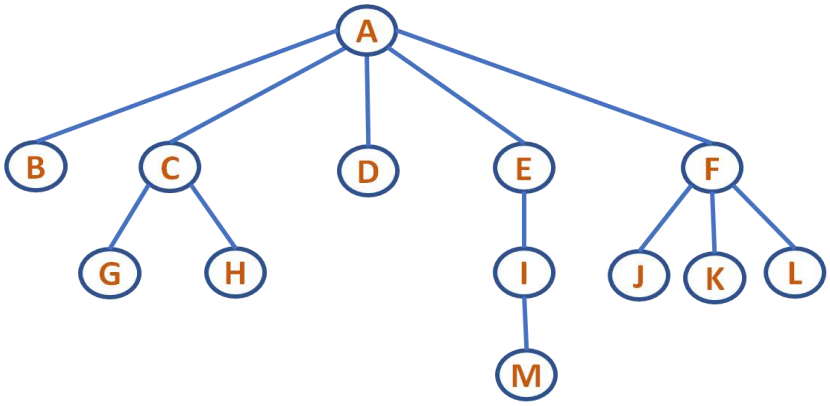


Figure 2: n-ary tree, n=5

Forest: is an ordered set of ordered trees

In the representation of a binary tree, each node contains an information field and two pointers to its two children. Trees may be represented as an array of tree nodes or a dynamic variable may be allocated for each node created. But how many pointers should a tree node contain? The number of children a node can have varies and may be as large or as small as desired.

```
#define MAXCHILD 20
struct treenode{
        int info;
        struct treenode *child[MAX];
};
```

where MAX is a constant

Restriction with the above implementation is that a node cannot have more than MAX children. Therefore the tree cannot be expanded.

Consider an alternative implementation as follows: All the children of a given node are linked and only the oldest child is linked to the parent

A node has link to first child and a link to immediate sibling

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

Conversion of an n-ary tree to a Binary Tree

Using Left Child – Right Sibling Representation an n-ary tree can be converted to a binary tree as follows:

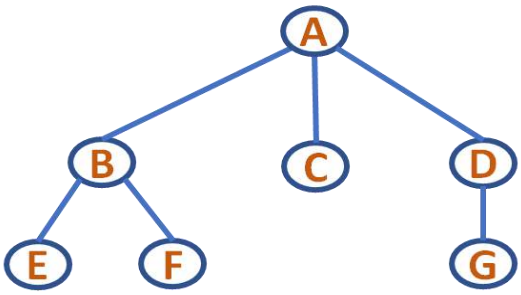
1) Link all the siblings of a node

2) Delete all links from a node to its children except for the link to its leftmost child

The **left child in binary tree** is the node which is the oldest child of the given node in an n-ary tree, and the **right child is the node to the immediate right of the given node** on the same horizontal line. Such a binary tree will not have a right sub tree.

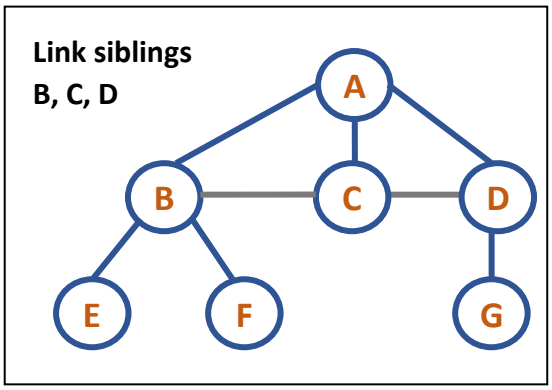
The node structure looks as shown below:

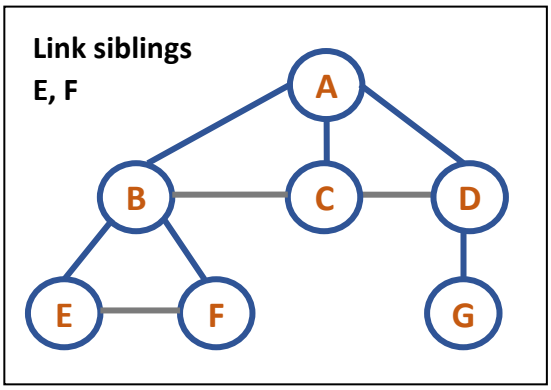
Consider the 3-ary tree shown below:



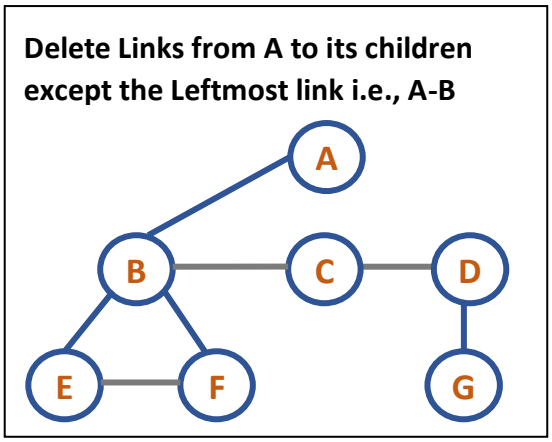
3 - ary tree

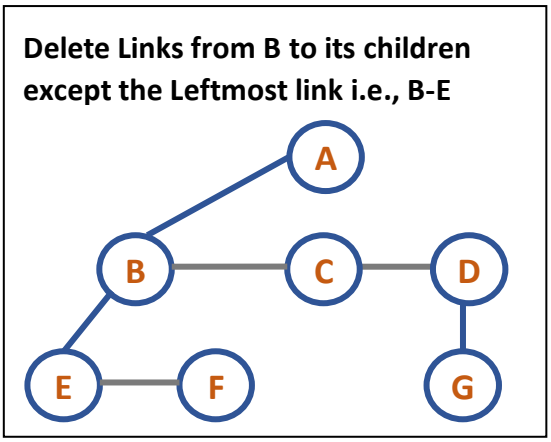
First step in the conversion is to Link all the siblings of a node.





Second step is to delete all the links from a node to its children except for the link to its leftmost child.

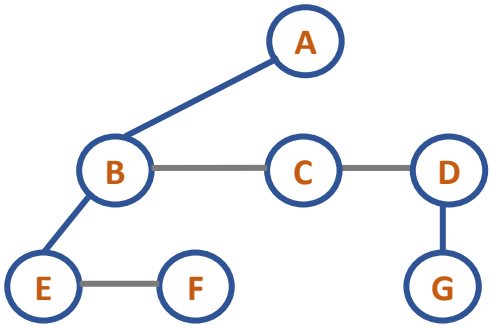




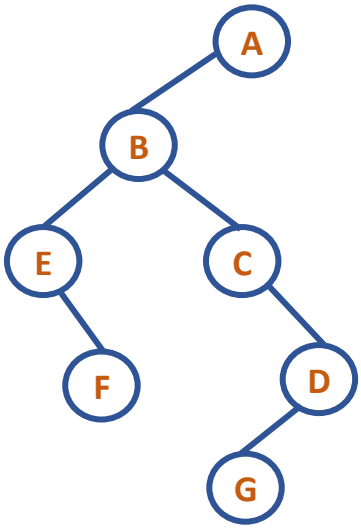
There are no more multiple links from any parent node to its children. The tree so obtained is the binary tree. But it doesn't look like one. Use the left child-right sibling relationship and make the tree look like a binary tree i.e., for any given node, its leftmost child will become the left child and immediate sibling becomes the right child. The binary tree so obtained will always have an empty right subtree for the root node. This is because the root of the tree we are transforming has no siblings.

For node A: Left child is B and has no Right child
For Node B: Left child is E and Right child is C
For Node C: No Left child and Right child is D
For node D: Left child is G and has no Right child
For Node E: No Left child and Right child is F
For Node F and Node G: No children (No Left child: because they do not have a child and no Right child: because they do not have a sibling towards right)

Left Child – Right Sibling representation of the given 3-ary tree



Corresponding Binary Tree representation of the given 3-ary tree



Conversion of a Forest to a Binary Tree

In the n-ary tree to binary tree conversion as stated above we saw that the right subtree for the root node of the binary tree is always empty. This is because the root of the tree we are transforming has no siblings.

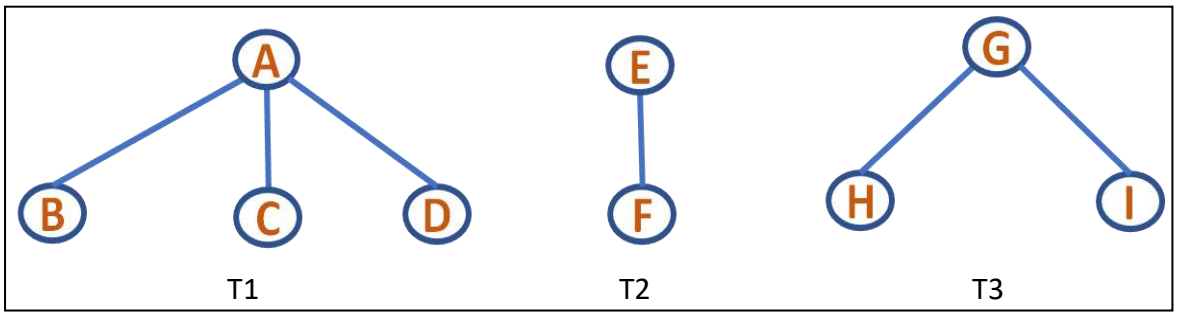
On the other hand, if we have a forest then these can all be transformed into a single binary tree as follows:

1) First obtain the binary tree representation of each of the trees in the forest
2) Link all the binary trees together through the right sibling field of the root nodes

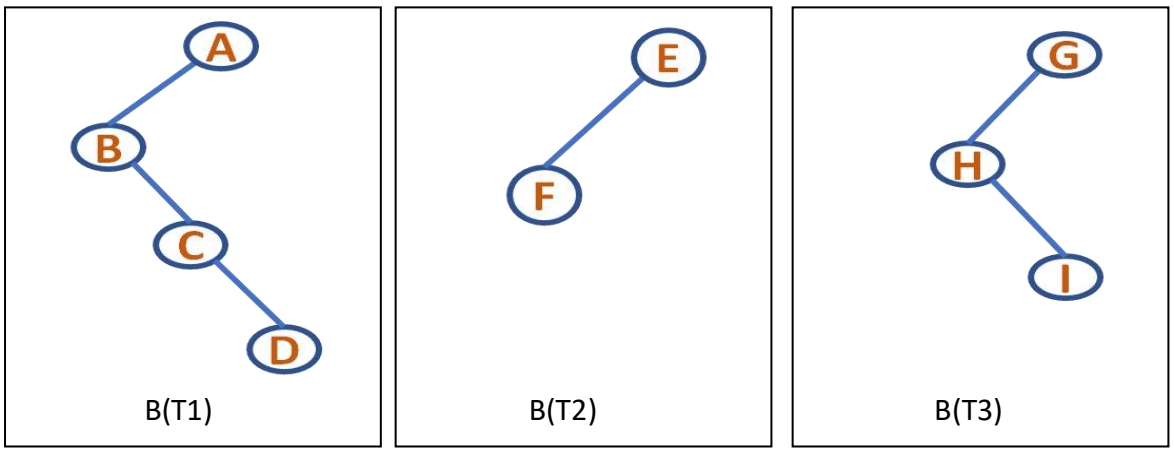
Conversion of a Forest to a Binary Tree can be formally defined as follows:

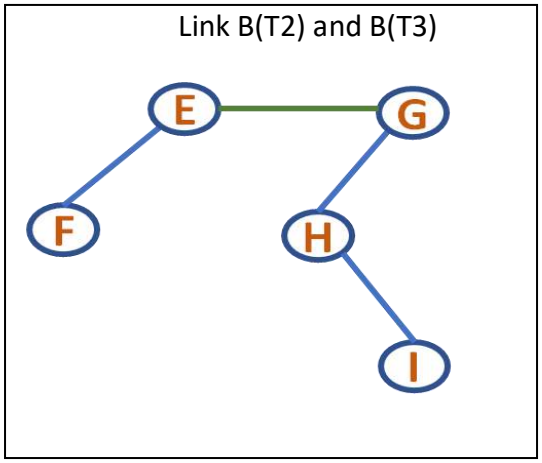
- If $T_1,...,T_n$ is a forest of n trees, then the binary tree corresponding to this forest, denoted by $B(T_1, ..., T_n)$:
  - is empty if $n = 0$
  - has root equal to root (T_1)
  - has left subtree equal to $B(T_{11}, T_{12}, ...,T_{1m})$ where T_{11} ,..., T_{1m} are the subtrees of root(T_1)
  - has right subtree $B(T_2, ..., T_n)$

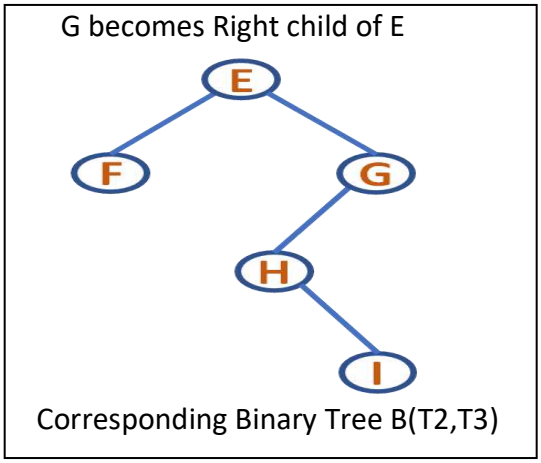
Consider the following Forest with three Trees:

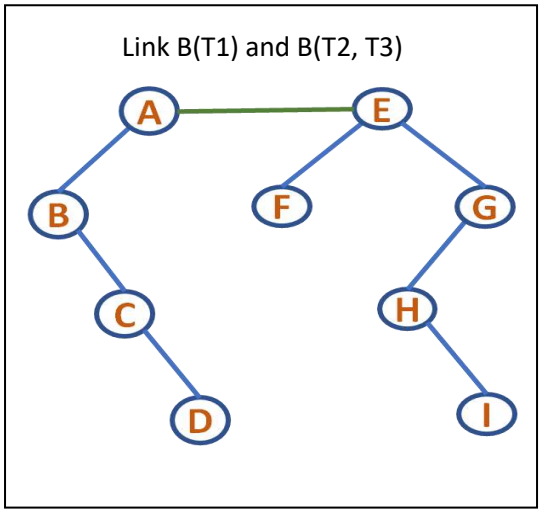


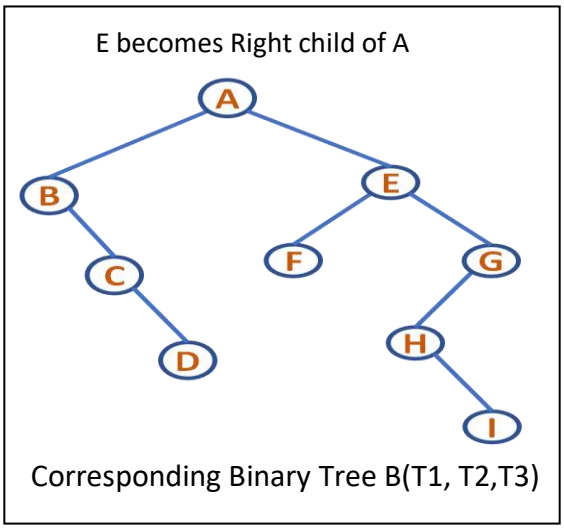
Corresponding Binary Trees:











Tree Traversal

The traversal methods for binary trees induce traversal methods for forests. The preorder, inorder, or postorder traversals of a forest may be defined as the preorder, inorder, or postorder traversals of its corresponding binary tree.

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

With the treenode implemented as having pointers to first child and immediate sibling, the traversal preorder, inorder and postorder for a tree are defined as below:

Preorder:

1. Visit the root of the first tree in the forest
2. Traverse in preorder the forest formed by the subtrees of the first tree, if any
3. Traverse in preorder the forest formed by the remaining trees in the forest, if any

```
void preorder(TREE *root)
{
   if(root!=NULL)
   {
     printf(" %d ",root->info);
     preorder(root->child);
     preorder(root->sibling);
  }
}
```

Inorder:

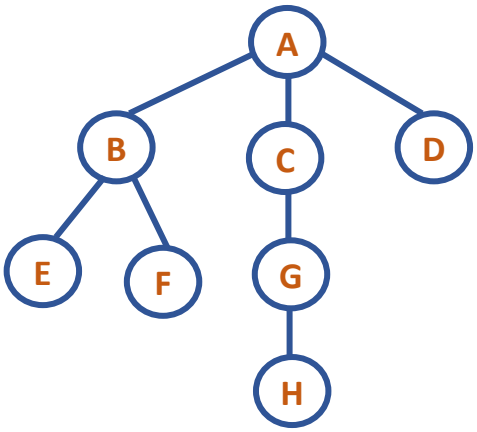
1. Traverse in inorder the forest formed by the subtrees of the first tree, if any
2. Visit the root of the first tree in the forest
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any

```
void inorder(TREE *root)
{
  if(root!=NULL)
    {
      inorder(root->child);
      printf(" %d ",root->info);
      inorder(root->sibling);
    }
}
```

Postorder:

1. Traverse in postorder the forest formed by the subtrees of the first tree, if any
2. Traverse in postorder the forest formed by the remaining trees in the forest, if any
3. Visit the root of the first tree in the forest

```
void postorder(TREE *root)
{
  if(root!=NULL)
    {
      postorder(root->child);
      postorder(root->sibling);
      printf(" %d ", root->info);
    }
}
```



Traversal of the above n-ary Tree:

Preorder: ABEFCGHD

Inorder: EFBHGCDA

Postorder: FEHGDCBA

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Priority Queue using Heap

Dr. Shylaja S S
Ms. Kusuma K V

Priority Queue using Heap

Ascending Heap: Root has the lowest element. Each node's data is greater than or equal to its parent's data. It is also called **min heap**.

Descending Heap: Root has the highest element. Each node's data is lesser than or equal to its parent's data. It is also called **max heap**.

Priority Queue is a Data Structure in which intrinsic ordering of the elements does determine the results of its basic operations.

Ascending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If *apq* is an ascending priority queue, the operation *pqinsert(apq,x)* inserts element x into apq and *pqmindelete(apq)* removes the minimum element from *apq* and returns its value.

Descending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed. If *dpq* is a descending priority queue, the operation *pqinsert(dpq,x)* inserts element x into *dpq* and *pqmaxdelete(dpq)* removes the maximum element from *dpq* and returns its value.

The operation *empty(pq)* applies to both types of priority queue and determines whether a priority queue is empty. *pqmindelete* or *pqmaxdelete* can only be applied to a non empty priority queue.

Once *pqmindelete* has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest element, and so on. Thus the operation successively retrieves elements of a priority queue in ascending order (However, if a small element is inserted after several deletions, the next retrieval will return that smallest element, which may be smaller than a previously retrieved element). Similarly, *pqmaxdelete* retrieves elements of a descending priority queue in descending order. This explains the designation of a priority queue as either ascending or descending.

The elements of a priority queue need not be numbers or characters that can be compared directly. They may be complex structures that are ordered on one or several fields. For example, telephone-book listings consist of names, addresses, and phone numbers and are ordered by name.

Sometimes the field on which the elements of a priority queue are ordered is not even part of the elements themselves; it may be a special, external value used specifically for the purpose of ordering the priority queue. For example, a stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. Te element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion. In both cases the time of insertion is not part of the elements themselves but is used to order the priority queue.

Heap as a Priority queue:

A heap allows a very efficient implementation of a priority queue. Now we will look into implementation of a descending priority queue using a descending heap. Let *dpq* be an array that implicitly represents a descending heap of size k. Because the priority queue is contained in array elements 0 to k-1, we add k as a parameter of insertion and deletion operations. Then the operation *pqinsert(dpq, k, elt)* can be implemented by simply inserting elt into its proper position in the descending list formed by the path from the root of the heap (dpq[0]) to the leaf dpq[k]. Once *pqinsert(dpq, k, elt)* has been executed, dpq becomes a heap of size k+1.

The insertion is done by traversing the path from the empty position k to position 0 (root), seeking the first element greater than or equal to *elt*. When that element is found, elt is inserted immediately preceding it in the path (i.e., *elt* is inserted as its child). As each element less than *elt* is passed during the traversal, it is shifted down one level in the tree to make room for *elt* (This shifting is necessary because we are using the sequential representation rather than a linked representation of the tree. A new element cannot be inserted between two existing elements without shifting some existing elements).

This heap insertion operation is also called the *siftup* operation because *elt* sifts its way up the tree. The following algorithm implements ***pqinsert(dpq, k, elt)***. **Algorithm for siftup**

```
c = k;
p = (c-1)/2;                    //p is parent of c
while(c>0 && dpq[p]<elt) {
  dpq[c]=dpq[p];
  c=p;                          //advance up the tree
  p=(c-1)/2;
}
dpq[c]=elt;
```

To implement *pqmaxdelete(dpq,k),* we note that the maximum element is always at the root of a k-element descending heap. When that element is deleted, the remaining k-1 elements in positions 1 through k-1 must be redistributed into positions 0 through k-2 so that the resulting array segment from dpq[0] through dpq[k-2] remains a descending heap. Let adjustheap(root,k) be the operation of rearranging the elements dpq[root+1] through dpq[k] into dpq[root] through dpq[k-1] so that subtree(root, k-1) forms a descending heap. Then ***pqmaxdelete(dpq,k)*** for a k-element descending heap can be implemented as:

```
p = dpq[0];
adjustheap(0,k-1);
return(p);
```

In a descending heap, not only is the root element the largest element in the tree, but an element in any position p must be the largest in subtree(p,k). Now subtree(p,k) consists of three groups of elements: its root, dpq[p]; its left subtree, subtree(2*p+1, k); and its right subtree, subtree(2*p+2, k). dpq[2*p+1], the left child of the root , is the largest element of the left subtree, and dpq[2*p+2], the right son of the root, is the largest element of the right

subtree. When the root dpq[p] is deleted, the larger of these two children must move up to take its place as te new largest element of subtree(p,k). Then the subtree rooted at position of the larger element moved up must be readjusted in turn.

Algorithm largechild(p,m)

```
c = 2*p+1;
if(c+1 <= m && x[c] < x[c+1])
  c=c+1;
if(c > m)
  return -1;
else
  return (c);
```

Then, *adjustheap(root,k)* may be implemented recursively as:

Algorithm adjustheap(root,k) **//recursive**

```
p = root;
c = largechild(p,k-1);
if(c >= 0 && dpq[k] < dpq[c]){
  dpq[p] = dpq[c];
  adjustheap(c,k);
}
else
  dpq[p] = dpq[k];
```

adjustheap(root,k) may be implemented iteratively as:

```
p = root;
kvalue = dpq[k];
c = largechild(p,k-1);
while(c >= 0 && kvalue < dpq[c]){
    dpq[p] = dpq[c];
    p = c;
    c = largechild(p,k-1);
}
dpq[p] = kvalue;
```

//implementation of the priority queue using heap

```
#include<stdio.h>
#define MAX 50
typedef struct priq
{
 int pq[MAX];
 int n;
}PQ;
```

```
void init(PQ *pt)
{
 pt->n=0;
}
void disp(PQ *pt)
{
 int i;
 for(i=0;i<pt->n;i++)
  printf("%d ",pt->pq[i]);
}
int enqueue(PQ *pt,int e)
{int p,c;
  if(pt->n==MAX-1) return 0;

    c=pt->n;
    p=(c-1)/2;
    while(c>0 && pt->pq[p]<e)
      {
       pt->pq[c]=pt->pq[p];
        c=p;
        p=(c-1)/2;
      }
    pt->pq[c]=e;
    pt->n=pt->n+1;
    return 1;
}
int dequeue(PQ *pt,int *ele)
{
  int p,c;
  *ele=pt->pq[0];
  int elt=pt->pq[pt->n-1];
  p=0;
  if(pt->n==1)
    c=-1;
  else c=1;
   if(pt->n>2 && pt->pq[2]>pt->pq[1])
    c=c+1;
  while(c>=0 && elt<pt->pq[c])
   {
      pt->pq[p]=pt->pq[c];
      p=c;
      c=2*p+1;
```

```
    if(c+1<pt->n-1 &&  pt->pq[c+1]>pt->pq[c])
      c=c+1;
    if(c>=pt->n-1) c=-1;
   }
  pt->pq[p]=elt;
  pt->n=pt->n-1;
  return 1;
}

int main()
{
 PQ pobj;
int k,choice,ele;

 init(&pobj);

 do{
  printf("1. Enqueue 2 Dequeue 3 Display\n");
  printf("Enter the choice");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1: printf("enter the information");
        scanf("%d",&ele);
        enqueue(&pobj,ele);
        break;
  case 2: k=dequeue(&pobj,&ele);
       if(!k) printf("empty");
       else
        printf("%d dequeues element",ele);
       break;
  case 3: disp(&pobj);
         break;
 }
 }while(choice<4);

return 0;
}
```