

Unit 11 Trees and Their Applications

Structure:

- 11.1 Introduction
 - Objectives
- 11.2 Tree Terminologies
- 11.3 Binary Tree
- 11.4 Balanced Tree (B Tree)
- 11.5 AVL Tree
- 11.6 Application of Trees
- 11.7 Summary
- 11.8 Terminal Questions
- 11.9 Answers to Self Assessment and Terminal Questions

11.1 Introduction

So far, we have discussed the data structures where linear ordering was maintained using arrays and linked lists. These data structures and their relationships are invariably expressed using single dimension. For some problems, it is not possible to maintain this linear ordering using linked list. Using non-linear data structures such as trees, graphs, multi-linked structures etc., more complex relations can be expressed. In this unit on trees, we begin with some definitions, the various operations that can be performed on trees along with various applications.

A **tree** consists of a finite set of elements, called **nodes** and a finite set of directed lines, called **branches** that connect the nodes. The number of branches associated with a node is the **degree** of the node. When the branch is directed towards the node, it is an **indegree** branch; When the branch is directed away from the node it is an **outdegree** branch, the sum of **outdegree** and indegree branches equals to the degree of the node.

Objectives:

At the end of this unit, you will be able to explain:

- Tree Concepts
- Binary Tree and its type
- Various operations on binary trees
- Binary Search Tree [BST] and its operation
- Tree operation implementation using C
- B Tree and its operation
- AVL Tree and its operation
- Application of Trees

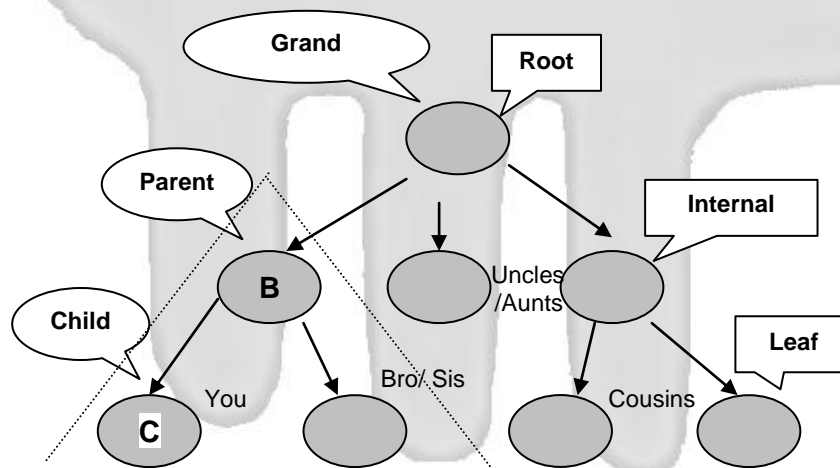
11.2 Tree Terminologies

Fig. 11.1: Tree T

Elements of Tree**Root**

The unique node with no predecessor

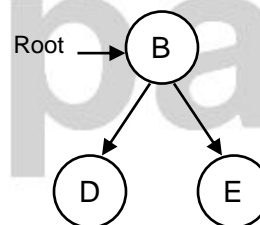


Fig. 11.2

Leaf

A node with no successors / Child

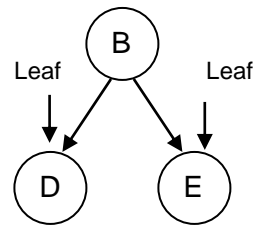


Fig. 11.3

Non Leaf

A node which has both a parent and at least one child

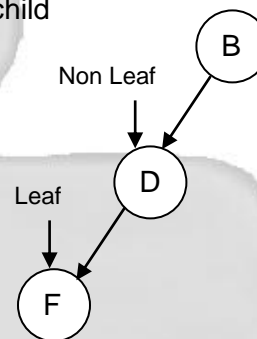


Fig. 11.4

Children /Child

The successors of a node

Parent

The unique predecessor of a node

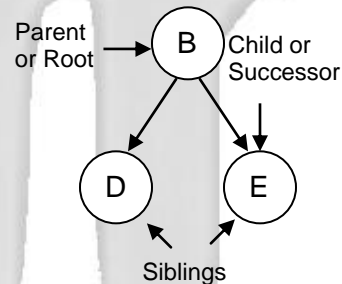


Fig. 11.5

Siblings

Two nodes that have the same parent, in a binary tree the two children are called the left and right. In the figure to the right, D and E are siblings. In this example $m=2$ so the tree is a binary tree

Internal Nodes

Nodes that are not root and not leaf are called as internal nodes.

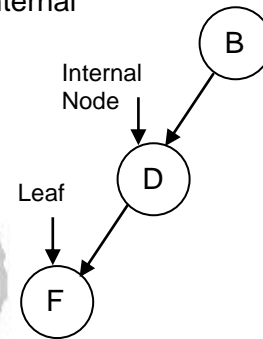


Fig. 11.6

Subtrees

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T , together with all the nodes below it, comprises a subtree of T . The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

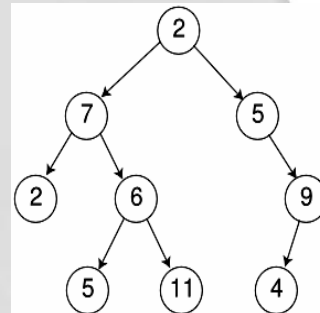


Fig. 11.7

11.3 Binary Tree

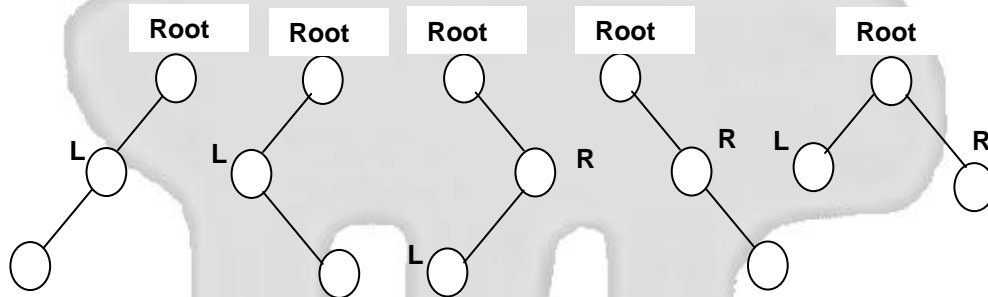
In computer science, a binary tree is a tree data structure in which each node has at most two children. Typically, the child nodes are called left and right. Binary trees are commonly used to implement binary search trees and binary heaps.

A Binary tree T is defined as a finite set of elements, called nodes such that,

- T is empty, called NULL tree or empty tree
- T contains a distinguished node R , called Root of t and the remaining nodes of T form an ordered pair of disjoint binary Tree T_1 and T_2 .
- The node contains max two children.

A **binary tree** is a tree in which no node can have more than two subtrees. In other words, a node can have zero, one, or two subtrees. In other words, a tree in which every parent has one or two children (but not more than that) is called as binary tree.

The “*root*” component of binary tree is the forefather of all children. But it can have only **upto** two children one “*right*” child and “*left*” child. These children can become fathers and each can produce only two children. In fact a child can become a father, grandfather, great grandfather and son. Fig shows five binary trees all with three nodes.



We can have binary trees with any number of nodes.

Fig. 11.8

11.3.1 Storage Representation

The trees can be represented using sequential allocation technique (using arrays) or by allocating the memory for a node dynamically (using linked allocation technique). In a linked allocation technique a node in a tree has three fields:

- info which contains the actual information
- llink which contains address of the left subtree
- rlink - contains address of the right subtree.

So, a node can be represented using structure as shown below:

```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};
typedef struct node* NODE;
```

A pointer variable root can be used to point to the root node always. If the tree is empty, the pointer variable root points to NULL indicating the tree is empty. The pointer variable root can be declared and initialized as `NODE root = NULL;`

Memory can be allocated or de-allocated using the functions **getnode()** and **freemnode()** as we discussed in linked lists in unit 9. A tree can also be represented using an array, which is called sequential representation (array representation). Consider the trees shown in fig. 6.4.a and fig. 6. 4.b.

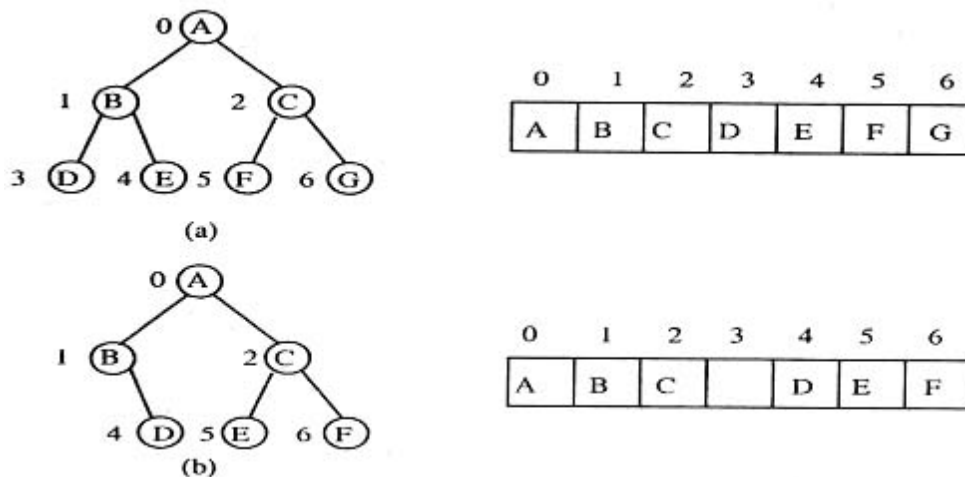


Fig. 11.9: A Sequential Representation of a Tree

The nodes are numbered sequentially from 0. The node with position 0 is considered as the root node. If an index i is 0, it gives the position of the root node. Given the position of any other node i , $2i+1$ gives the position of the left child and $2i+2$ gives the position of the right child. If i is the position of the left child, $i+1$ gives the position of the right child and if i is the position of the right child, $i-1$ gives the position of the left child. Given the position of any node i , the parent position is given by $(i-1)/2$. If i is odd, it points to the left child otherwise, it points to the right child. The different ways in which a tree can be represented using an array is shown below.

- In the first representation shown below, some of the locations may be used and some may not be used. For this, a flag field namely, **used** is used just to indicate whether a memory location is used to represent a node or not. If flag field **used** is 0, the corresponding memory location is not used and indicates the absence of node at that position. So, each node contains two fields:
 - **info** where the information is stored
 - **used** indicates the presence or the absence of a node

The structure declaration for this is:

```
#define MAX_SIZE 200
struct node
{
    int info;
    int used;
};
typedef struct node NODE;
```

An array **A** of type **NODE** can be used to store different items and the declaration for this is shown below:

```
NODE A [MAX_SIZE];
```

- An alternate representation is that, instead of using a separate flag field used to check the presence of a node; one can initialize each location in the array to 0 indicating the node is not used. non-zero value in the location indicates the presence of the node.

11.3.2 Operation on Binary Tree

Example: Function to insert an item into a / Create Tree

```
NODE insert(int item, NODE root)
{
    NODE temp; /* Node to be inserted */
    NODE cur    /* Child node */
    NODE prev;  /* Parent node */
    char direction[10]; /* Directions where the node has to be inserted */
    int i;          /* Maximum depth where a node can be inserted */
    temp = getnode(); /* Obtain a node from the availability list */
    temp->info = item; /* Copy the necessary information */
    temp->llink = temp->rlink = NULL;
    if ( root == NULL ) return temp; /* Node is inserted for the first time */
    printf("Give the directions where you want to insert\n");
    scanf("%s", direction);
    toupper(direction); /* Convert the directions to upper case */
    prev = NULL;
    cur = root;

    /* find the position to insert */
    for ( i = 0; i < strlen(direction) && cur != NULL; i++ )
    {
        prev = cur; /* Parent */
        if ( direction[i] == 'L' ) /* If direction is (L) move left */
            cur = cur->llink;
        else /* Otherwise move towards right */
```



```
        cur = cur->rlink;
    }
    if ( cur != NULL || i!= strlen(direction))
    {
        printf("Insertion not possible\n");
        freenode(temp);
        return root;
    }
    /* insert the node at the leaf level */
    if ( direction[i-1 ] == 'L' )
        prev->llink = temp; /* Attach node to the left of the parent */
    else
        prev->rlink = temp; /*Attach node to the right of the parent*/
    return root;
}
```

Binary Tree Traversals

Traversing is the most common operation that can be performed on trees. In the traversal technique each node in the tree is processed or visited exactly once systematically one after the other. The different traversal techniques are **Inorder**, **Preorder** and **Postorder**.

There are three types of tree traversals

- **Preorder**
- **Inorder**
- **Postorder**

Algorithm for Tree Traversals

A binary tree T is in memory. The algorithm does a Preorder Traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

- The **preorder traversal** of a binary tree can be recursively defined as follows
 1. Process the root Node [N]
 2. Traverse the Left subtree in preorder [L]
 3. Traverse the Right subtree in preorder [R]

Step I	Set Top:=1, STACK[1]=NULL, PTR:=Root
Step II	Repeat Step III to Step V while PTR ≠ NULL
Step III	Apply PROCESS INFO[PTR]
Step IV	If RIGHT[PTR] ≠ NULL then Set Top:=Top+1 STACK[Top]=RIGHT[PTR] [End of if Structure]
Step V	If LEFT[PTR] ≠ NULL then Set PTR =LEFT[PTR] Else PTR=STACK[Top], Top=Top-1 [End of if Structure] [End of Step II Loop]
Step VI	Exit

Pictorial Representation of Preorder Traversal

Manipal

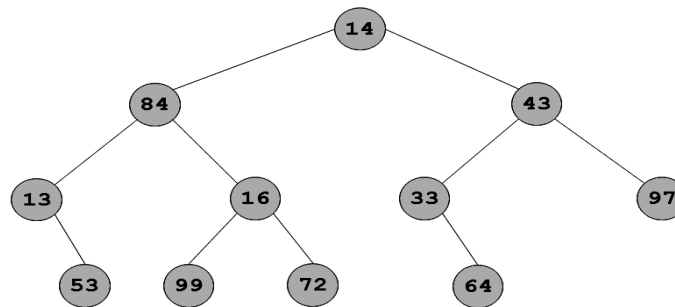
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

**14**

Stack



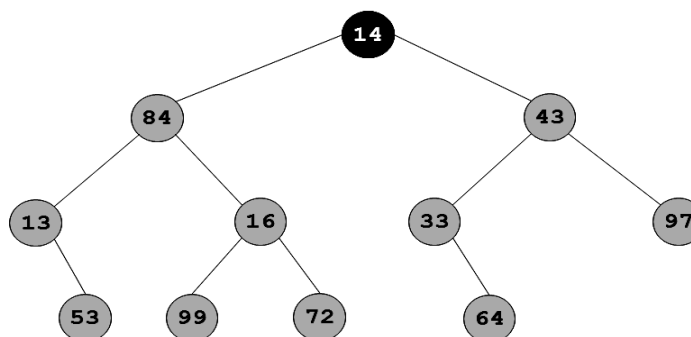
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14**84****43**

Stack



Push the root onto the stack.

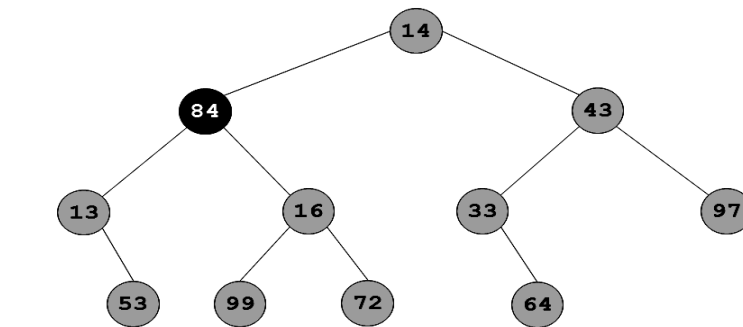
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84

13
16
43

Stack



Push the root onto the stack.

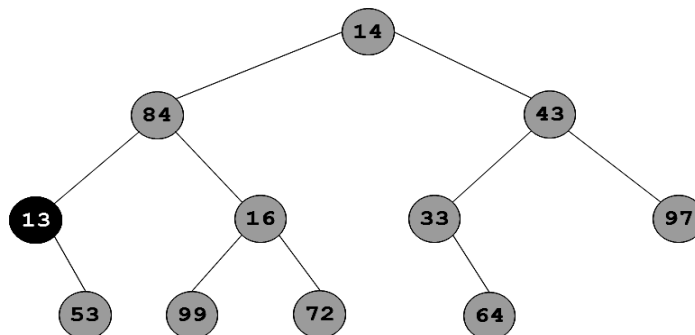
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13

53
16
43

Stack



Push the root onto the stack.

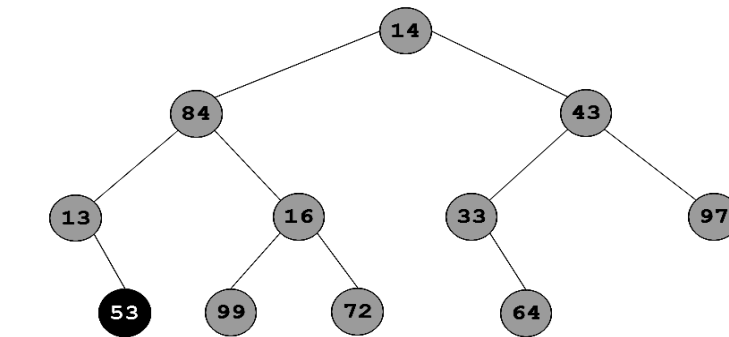
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53

16
43

Stack



Push the root onto the stack.

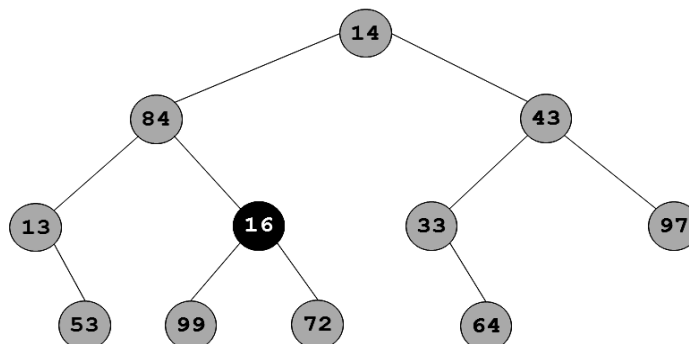
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16

99
72
43

Stack



Push the root onto the stack.

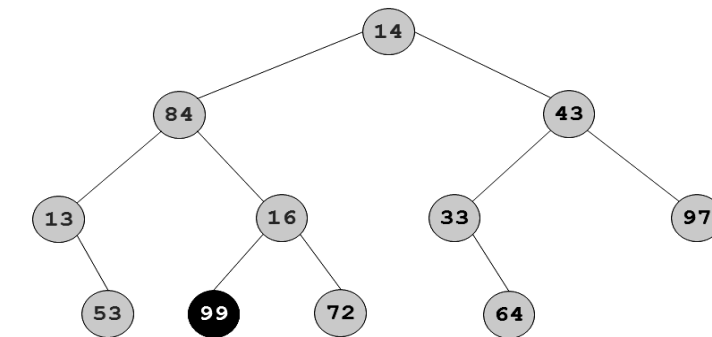
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99

72
43

Stack



Push the root onto the stack.

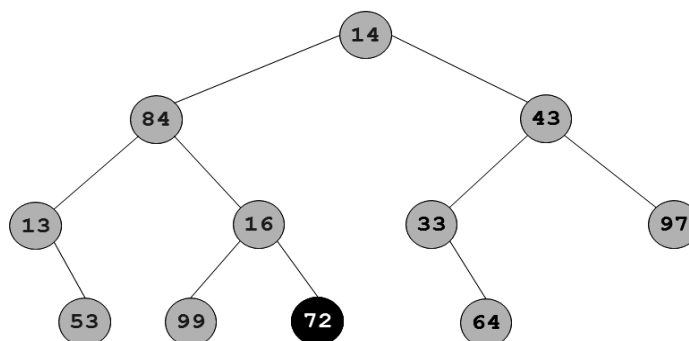
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72

43

Stack



Push the root onto the stack.

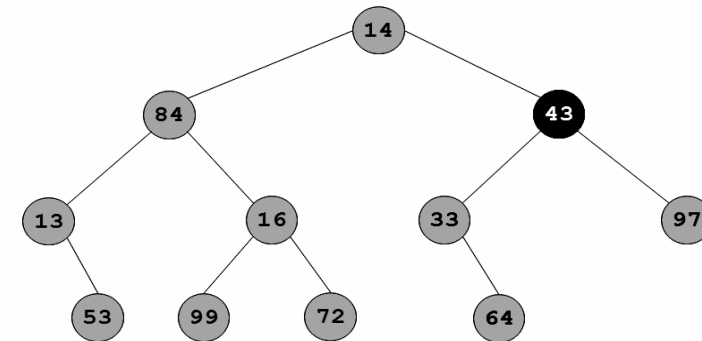
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43

33
97

Stack



Push the root onto the stack.

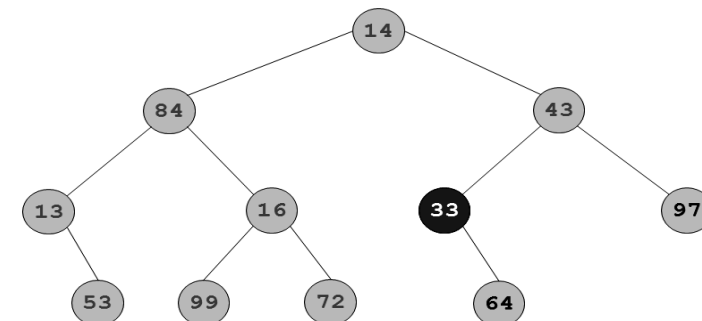
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33

64
97

Stack



Push the root onto the stack.

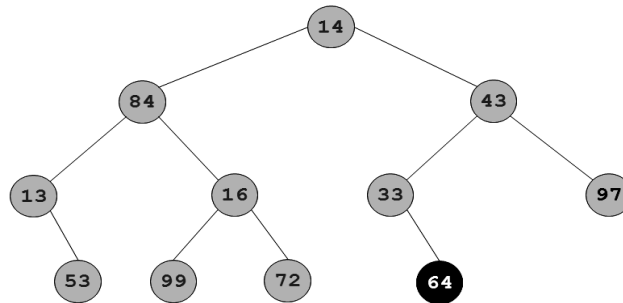
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33 64

97

Stack



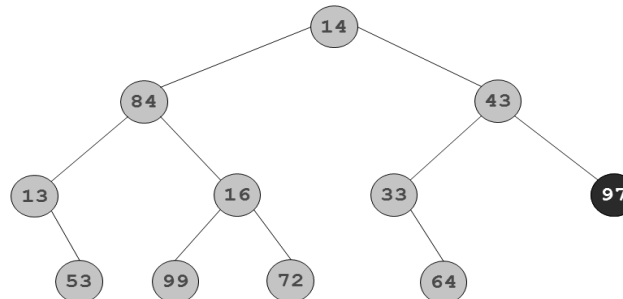
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33 64 97

Stack



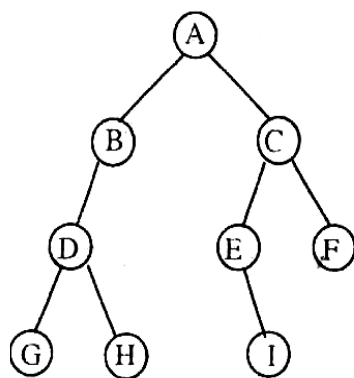
- The **Postorder Traversal** of a binary tree can be recursively defined as follows:
 1. Traverse the Left subtree in postorder[L]
 2. Traverse the Right subtree in postorder [R]
 3. Process the root Node [N]

Step I	Set Top:=1, STACK[1]=NULL, PTR:=Root
Step II	Repeat Step III to Step V while PTR ≠ NULL
Step III	Set Top:=Top+1, Stack[Top]:=PTR
Step IV	If RIGHT[PTR] ≠ NULL then Set Top:=Top+1 and STACK[Top] = - RIGHT[PTR] [End of If Structure]
Step V	Set PTR:= LEFT[PTR] [End of Step II Loop]
Step VI	Set PTR:=STACK[PTR], Top:=Top-1
Step VII	Repeat while PTR>0 then Apply PROCESS to INFO[PTR] Set PTR=STACK[PTR] Top:=Top-1 [End of Loop]
Step VIII	If PTR<0 then Set PTR = - PTR Goto Step II Loop [End of If Structure]
Step IX	Exit

- The **Inorder Traversal** of a binary tree can be recursively defined as follows:
 1. Traverse the Left subtree in postorder[L]
 2. Process the root Node [N]
 3. Traverse the Right subtree in postorder [R]

Step I	Set Top:=1, STACK[1]=NULL, PTR:=Root
Step II	Repeat while PTR ≠ NULL Set Top:= Top+1 and STACK[Top]:=PTR Set PTR:=LEFT[PTR] [End of Loop]
Step III	Set PTR:=STACK[Top] and Top:=Top-1

Step IV	Repeat Step V to Step VII while PTR \neq NULL
Step V	Apply PROCESS to INFO[PTR]
Step VI	If RIGHT[PTR] \neq NULL then Set PTR:=RIGHT[PTR] Goto Step II [End of If Structure]
Step VII	Set PTR:= STACK[Top] and Top:=Top-1 [End of If Step IV Loop]
Step VIII	Exit

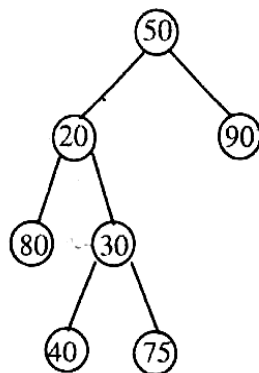
Examples

(a)

Inorder : G D H B A E I C F

Preorder : A B D G H C E I F

Postorder: G H D B I E F C A



(b)

Inorder : 80 20 40 30 75 50 90

Preorder : 50 20 80 30 40 75 90

Postorder: 80 40 75 30 20 90 50

Fig. 11.10


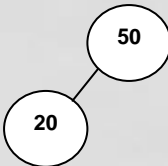
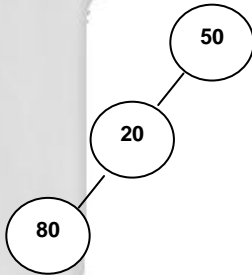
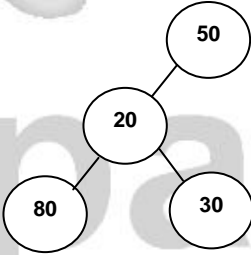
Construct Binary Tree from Inorder and Preorder Sequence:

Insert elements from preorder and check the position from Inorder sequence

Example:

Inorder: 80 20 40 30 75 50 90

Preorder: 50 20 80 30 40 75 90

Insert 50	It is the root	
Insert 20	Check the position of value 20 of 50 in the Inorder sequence, which is left side of 50	
Insert 80	Check the position of value 80 of 50 and then of 20, it is the left most.	
Insert 30	Check the position of value 30 which is left side of the 50 but right side of the 20. So it will be right of 20	

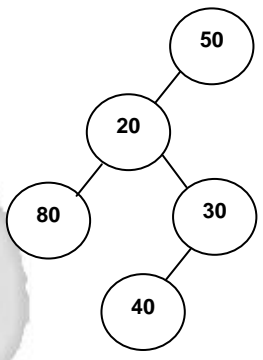
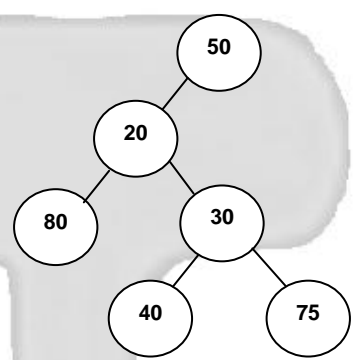
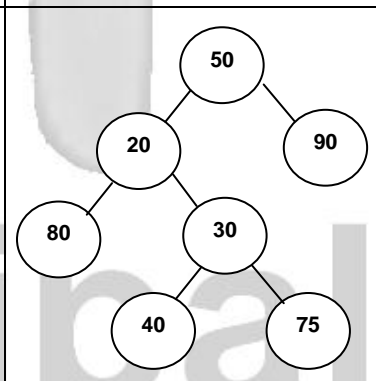
Insert 40	Check the position of value 40 which is left of 50 and right of 20 then left of 30	 <pre>graph TD; 50((50)) --- 20((20)); 50 --- 30((30)); 20 --- 80((80)); 20 --- 40((40)); 30 --- ;</pre>
Insert 75	Check the position of the value 75 which is left of 50 , right of 20 , right of 30	 <pre>graph TD; 50((50)) --- 20((20)); 50 --- 30((30)); 20 --- 80((80)); 20 --- 40((40)); 30 --- 75L((75)); 30 --- 75R((75));</pre>
Insert 90	Check the position of the value 90 which is right side of 50	 <pre>graph TD; 50((50)) --- 20((20)); 50 --- 90((90)); 20 --- 80((80)); 20 --- 30((30)); 30 --- 40((40)); 30 --- 75((75));</pre>

Fig. 11.11

11.3.3 Complete Binary Tree

If the out degree of every node in a tree is either 0 or 2, then the tree is said to be strictly binary tree i.e., each node can have maximum two children or empty. A binary tree is said to be strictly binary if every non-leaf node has non-empty left and right sub trees.

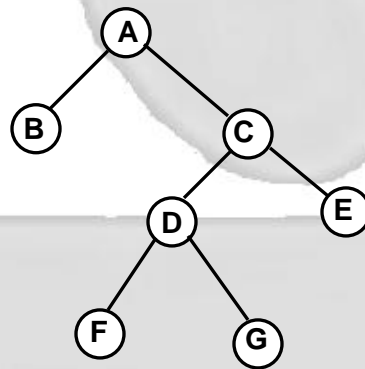


Fig. 11.12 Strictly Binary Tree

11.3.4 Full / Strictly Binary Tree

A strictly binary tree in which the number of nodes at any level l is 2^{l-1} , the tree is said to be a complete binary tree. The tree shown in figure 11.2 is a strictly binary tree and at the same time it is a complete binary tree.

In a complete binary tree, the total number of nodes at level 0 is 1 i.e., 2^0

Number of nodes at level 1 is 2 i.e., 2^1

Number of nodes at level 2 is 4 i.e., 2^2

Number of nodes at level 3 is 8 i.e., 2^3

.....

Number of nodes at the last level d is 2^d .

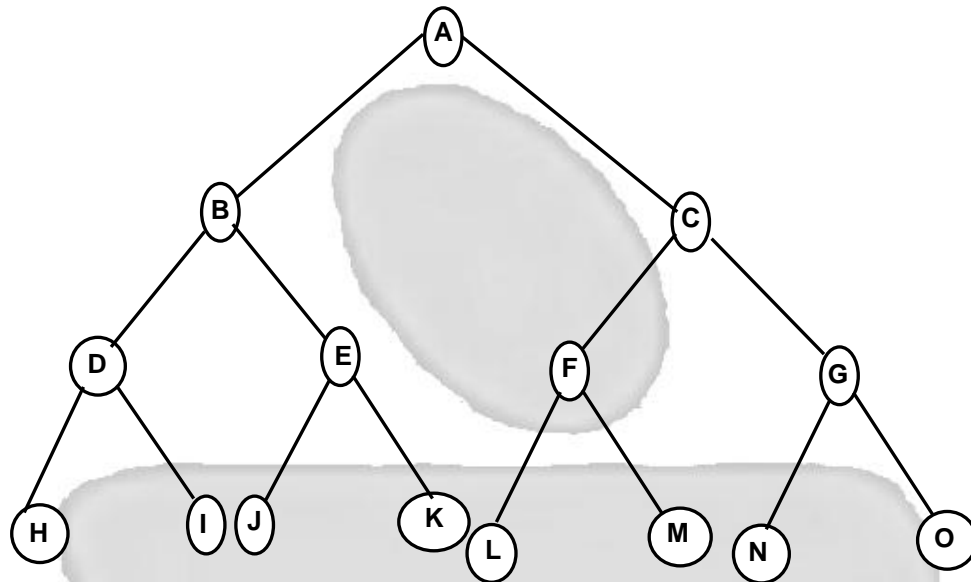


Fig. 11.13: Full Binary Tree

It is clear from the figure 11.13 that all the nodes in the final level d are leaf nodes and the total number of leaf nodes is 2^d . In a complete binary tree, we can easily find the number of non-leaf nodes. Now, let us find the number of non-leaf nodes in a complete binary tree.

Total number of nodes in the complete binary tree

$$= 2^0 + 2^1 + 2^2 + \dots + 2^d$$

Summation of this series is given by

$$S = a(r^n - 1) / (r - 1)$$

where $a = 1$, $n = d + 1$ and $r = 2$

$$\text{So, total number of nodes } n_t = 2^{d+1} - 1$$

Nodes at level d are all leaf nodes.

$$\text{So, number of non-leaf nodes is given by } 2^{d+1} - 1 - 2^d$$

which is equal to $2^d - 1$.

11.3.5 Binary Search Tree (BST)

A binary search tree is a binary tree in which for each node say x in the tree, elements in the left subtree are less than info (x) and elements in the right subtree are greater or equal to info(x). Every node in the tree should satisfy this condition, if left subtree or right subtree exists. It is not difficult to see that this property guarantees that the inorder traversal of T will yield a sorted listing of the elements of Tree T .

Other common operations performed on Binary Search Trees are

- **Searching** – Search for a specific item in the tree
- **Insertion** – An item is inserted
- **Deletion** – Deleting a node from a given tree.

A binary search tree T is in memory and ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM.

There are four cases:

- 1) **LOC = NULL and PAR = NULL will indicate that the tree is empty.**
- 2) **LOC \neq NULL and PAR = NULL will indicate that ITEM is the root of T**
- 3) **LOC = NULL and PAR \neq NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.**
- 4) **LOC \neq NULL and PAR \neq NULL will indicate the ITEM already exists in T .**

- **Searching:** Finding an element from Binary Search Tree

Start searching from the root node and move downward towards left or right based on whether the item lies towards left subtree or right subtree. This is achieved by comparing this item with root node of an appropriate subtree (left subtree or right subtree). If two items are same then search is successful and address of that node is returned. If the item is less than info (root), search in the left subtree, otherwise search in the right subtree. If

item is not found in the tree, finally root points to NULL and return root indicating search is unsuccessful.

Algorithm:

FIND (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

Step I	[Tree Empty Checking] If ROOT = NULL then Set LOC:=NUL, PAR:=NULL Return [End of If Structure]
Step II	[ITEM at Root Checking] If ITEM = INFO[ROOT] then Set LOC:=ROOT, PAR:=NULL Return [End of if Structure]
Step III	[Initialization pointer PTR and SAVE] If ITEM < INFO[ROOT] then Set PTR:=LEFT[ROOT] and SAVE:=ROOT Else Set PTR:=RIGHT[ROOT] and SAVE:=ROOT [End of If Structure]
Step IV	Repeat Step V and Step VI while PTR ≠ NULL
Step V	[ITEM Found Checking] IF ITEM = INFO[PTR] then Set LOC:=PTR and PAR :=SAVE Return [End of If Structure]
Step VI	IF ITEM< INFO[PTR] then Set SAVE:=PTR and PTR:=LEFT[PTR] Else Set SAVE:=PTR and PTR:=RIGHT[PTR] [End of If Structure] [End of Step IV Loop]
Step VII	[Search Unsuccessful] Set LOC:= NULL and PAR:=SAVE
Step VIII	Exit

Function:**a) Function to search for an item in BST using iteration**

```
NODE iterative_search (int item, NODE root)
{
    /* search for the item */
    while ( root != NULL && item != root->info )
    {
        root = ( item < root->info ) ? root->llink : root->rlink;
    }
    return root;
}
```

b) Function to search for an item in BST using recursion

```
NODE recursive_search (int item, NODE root)
{
    if ( root == NULL || item == root->info )
        return root;
    if ( item < root->info )
        return recursive_search(item, root->llink);
    return recursive_search(item, root->rlink);
}
```

- **Insertion** - Inserting an element from Binary Search Tree

Creating a binary search tree is nothing but repeated insertion of an item into the existing tree. So, we concentrate on how to insert an item into the tree. In a BST (Binary Search Tree), items towards left subtree of a node temp will be less than info (temp) and the items in the right subtree are greater or equal to info (temp).

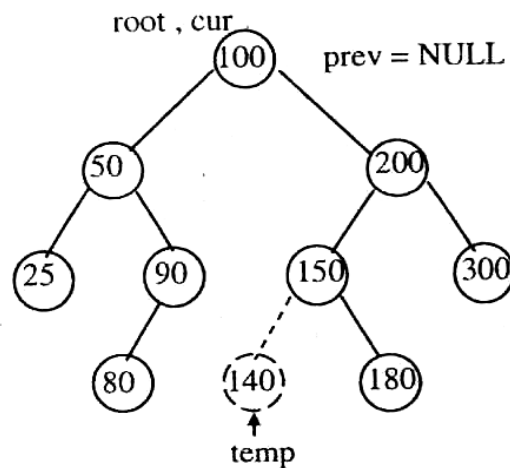


Fig. 11.14

Consider the BST shown in the above fig. Suppose the node pointed to by temp (with item 140) has to be inserted. The item 140 is compared with root node i.e., 100. Since it is greater, the right subtree of root node has to be considered. Now compare 140 with 200 and since it is less, consider the left subtree of the node 200. Now, compare 140 with 150. Since it is less, consider the left subtree of node 150. Since, left subtree of node 150 empty, the node containing the item 140 has to be inserted towards left of a node 150. Thus, to find the appropriate place and insert an item, the search should start from the root node. This is achieved by using two pointer variables prev and cur. The pointer variable prev always points to parent of cur node. Initially cur points to root node and prev points to NULL.

Algorithm:**INSERT (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)**

Step I	Call FIND(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
Step II	If LOC \neq NULL then exit
Step III	[Copy ITEM into new node in AVAIL list] If AVAIL = NULL then

	Display "Overflow " and Exit [End of If Structure] Set NEW:=AVAIL, AVAIL:=LEFT[AVAIL] and INFO[NEW]:=ITEM Set LOC:=NEW, LEFT[NEW]= NULL and RIGHT[NEW]:=NULL
Step IV	[Add ITEM to Tree T] IF PAR = NULL then Set ROOT:= NEW Else IF ITEM <INFO[PAR] then Set LEFT[PAR]:= NEW Else Set RIGHT[PAR]:= NEW [End of If Structure]
Step V	Exit

Example: Function to insert an item into an ordered binary tree

NODE insert (int item, NODE root)

```

{
    NODE temp, cur, prev;
    temp = getnode( );    /* Obtain new node from the availability list */
    temp->info = item;    /* Copy appropriate data */
    temp->llink = NULL;
    temp->rlink = NULL;
    if ( root == NULL) return temp; /* Insert a node for the first time */
    /* find the position to insert */
    prev = NULL;
    cur = root;
    while ( cur != NULL )
    {
        /* Obtain parent */
        prev = cur;
        /* do not insert duplicate item */
    }
}

```

```

        if ( item == cur->info )
        {
            printf("Duplicate items not allowed\n");
            freenode(temp);
            return root;
        }
        /* Find the appropriate left or right child */
        cur = ( item < cur->info ) ? cur->llink : cur->rlink;
    }
    if ( item < prev->info ) /* If node to be inserted is less than parent */
        rev->llink = temp;    /* Insert towards left of the parent */
    else
        prev->rlink = temp; /* otherwise, insert towards right of the parent */
    return root;    /* Return the root of the tree */
}

```

- **Deletion** – Delete an element from Binary Search Tree

The deletion algorithm first find the location of the node N which contains ITEM and also the which contains ITEM and also the location of the parent node P(N). There are three different cases.

- N has No Child:** N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer
- N has exactly one child:** N is deleted from T by simply replacing the location of N in P(N) by the location of only child of N.
- N has two children:** Let S(N) denotes the inorder successor of N. then N is deleted from T by first deleting S(N) from T. and replacing node N in T by node S(N).

CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, when N does not have two children. The pointer PAR gives the location of the parent of N or else PAR=NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N or else CHILD=NULL indicates N has no children.

Step I	[Initializes CHILD] If LEFT[LOC] = NULL and RIGHT[LOC]=NULL then Set CHILD:=NULL Else If LEFT[LOC] ≠ NULL then Set CHILD:=LEFT[LOC] Else Set CHILD:=RIGHT[LOC] [End of If Structure]
Step II	If PAR ≠ NULL then If LOC = LEF[PAR] then Set LEFT[PAR]:=CHILD Else Set RIGHT[PAR]:=CHILD [End of If Structure] Else Set ROOT:= CHILD [End of If Structure]
Step III	Return

CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete the node N at location LOC, when delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N or else PAR=NULL indicates that N is the ROOT node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

Step I	[Find SUC and PARSUC] Set PTR:=RIGHT[LOC] and SAVE:=LOC Repeat while LEFT[PTR] ≠ NULL Set SAVE:=PTR and PTR:=LEFT[PTR] [End of Loop Structure] Set SUC:=PTR and PARSUC:=SAVE
Step II	[Delete Inorder Successor] Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
Step III	[Replace node N by its Inorder Successor] If PAR ≠ NULL then If LOC =LOC[PAR] then Set LEFT[PAR]:=SUC Else Set RIGHT[PAR]:=SUC [End of If Structure] Else Set ROOT:=SUC [End of If Structure] Set LEFT[SUC]:=LEFT[LOC] and RIGHT [SUC]:=RIGHT[LOC]
Step IV	Return

Now we can formally state that deletion algorithm

A binary search tree T is in memory and an ITEM of information is given this algorithm deletes ITEM from the Tree.

DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

Step I	[Find the location of ITEM and its parent] Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC,PAR)
Step II	[Check the ITEM] If LOC = NULL then Display "Item Not Found" Exit [End of if Structure]

Step III	[Delete Node Containing ITEM] If RIGHT[LOC] \neq NULL and LEFT[LOC] \neq NULL then Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR) Else Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR) [End of If Structure]
Step IV	[Return deleted node to the AVAIL List] Set LEFT[LOC]:=AVAIL and AVAIL:=LOC
Step V	Exit

Example 1: Function to delete an item from the tree

NODE delete_ item (int item, NODE root)

```

{
    NODE cur, parent, suc, q;
    if( root == NULL)
    {
        printf("Tree is empty! Item not found\n");
        return root;
    }
    /*obtain the position of the node to be deleted and its parent */
    parent = NULL;
    cur = root;
    while ( cur != NULL && item != cur->info )
    {
        parent = cur;
        cur = ( item < cur->info ) ? cur->llink : cur->rlink;
    }
    if ( cur == NULL)
    {
        printf("Item not found\n");
        return root;
    }
}

```

```

    }

/* Item found and delete it */
*****/

/*          CASE 1          */
*****/

if ( cur->llink == NULL)    /* If left subtree is empty */
q = cur->rlink;  /* obtain the address of non empty right subtree */
    else if ( cur->rlink == NULL) /* If right subtree is empty */
q = cur->llink;  /* obtain the address of non empty left subtree */
    else
    {
        /******/

        /*          CASE 2          */
        /******/

        /* obtain the inorder successor */
        suc = cur->rlink;    /* Inorder successor lies towards right */
        while (suc->llink != NULL) /* and immediately keep traversing left */
        {
            suc = suc->llink;
        }
        suc->llink = cur->llink; /* Attach left of node to be deleted to left */
        /* of successor of the node to be deleted */
        q = cur->rlink;  /* Right subtree is obtained */
    }
    /* If parent does not exist return q itself as the root */
    if (parent == NULL) return q;
    /* connecting parent of the node to be deleted to q */
    if ( cur == parent->llink)
        parent->llink = q;

```



```
        else
            parent->rlink = q;
        freenode(cur);
        return root;
    }
```

11.3.6 Threaded Binary Tree

Binary tree is threaded by making all right child pointers that would normally be null point to the Inorder successor of the node, and all left child pointers that would normally be null point to the Inorder predecessor of the node.

A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal. It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable.

This is possible, because if a node (k) has a right child (m) then m's left pointer must be either a child, or a thread back to k. In the case of a left child, that left child must also have a left child or a thread back to k, and so we can follow m's left children until we find a thread, pointing back to k. The situation is similar for when m is the left child of k

Manipal

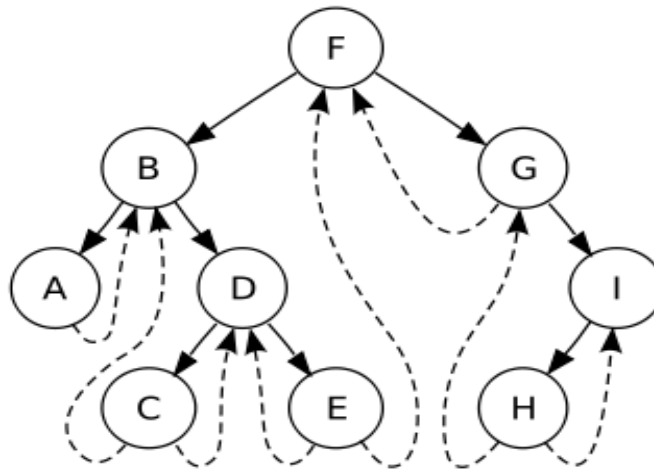


Fig. 11.15: Threaded Binary Tree

11.3.7 Red-Black Tree

A red-black tree is a binary search tree where each node has a *color* attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree apply.

- A node is either **red** or **black**.
- The root is black. (This rule is used in some definitions and not others. Since the root can always be changed from red to black but not necessarily vice-versa this rule has little effect on analysis.)
- All leaves are black, even when the parent is black (The leaves are the *null* children.)
- Both children of every red node are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes, either counting or not counting the null black nodes. (Counting or not counting the null black nodes does not affect the structure as long as the choice is used consistently.)

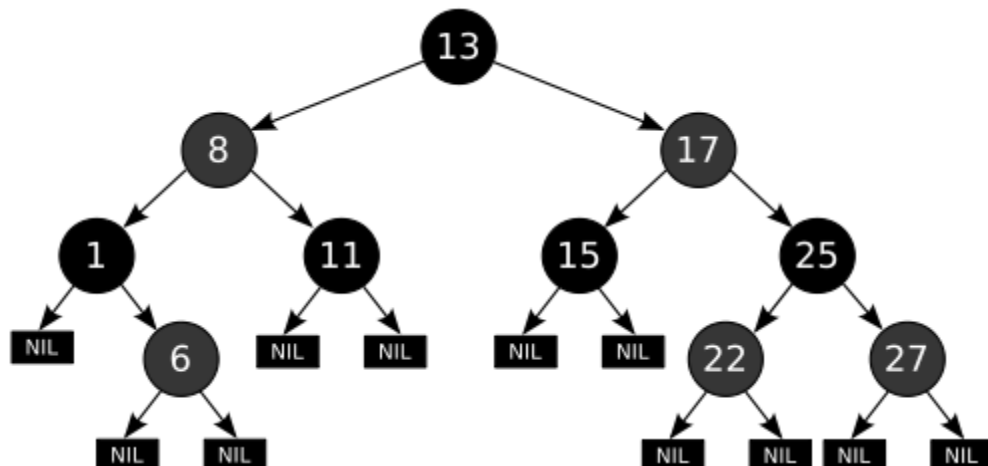


Fig. 11.16: Red Black Tree

Red-black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry can be based on red-black trees.

11.4 Balanced Tree (B Tree)

Unlike a binary-tree, each node of a B-Tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a sub tree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a sub tree containing all keys greater than any keys in the node.

- The order of B Tree implies maximum number of values can be added into a single node.
- If the order is 3 then maximum 3 values can be added and 4 links.

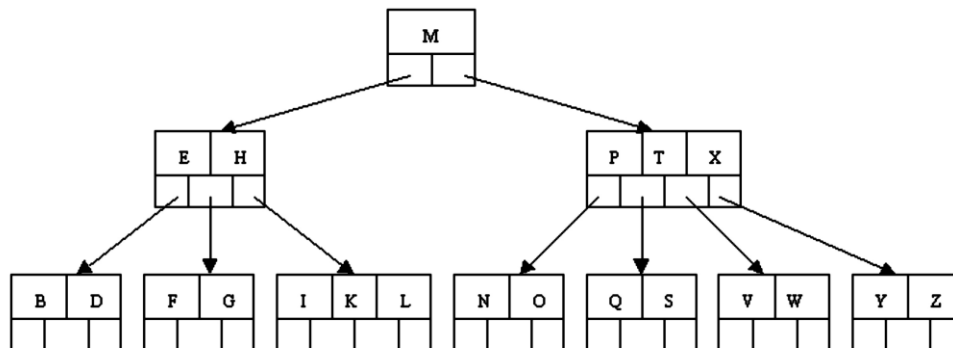


Fig. 11.17: B Tree

A B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has $\leq m$ children.
2. Every node (except root and leaves) has $\geq m/2$ children.
3. The root has at least 2 children if it is not a leaf node.
4. All leaves appear in the same level, and carry no information.
5. A non-leaf node with k children contains $k - 1$ keys

A B-tree is kept balanced by requiring that all external nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

11.5 AVL Tree

In computer science, an **AVL tree** is a self-balancing binary search tree, and it is the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. Lookup, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information". It is a special type of Balanced Binary Search Tree. There is the Balanced Factor (BF) that must be -1, 0 or +1 for each node.

The Balanced Factor will be calculated as follows:

Balanced Factor(BF) = Left Height – Right Height

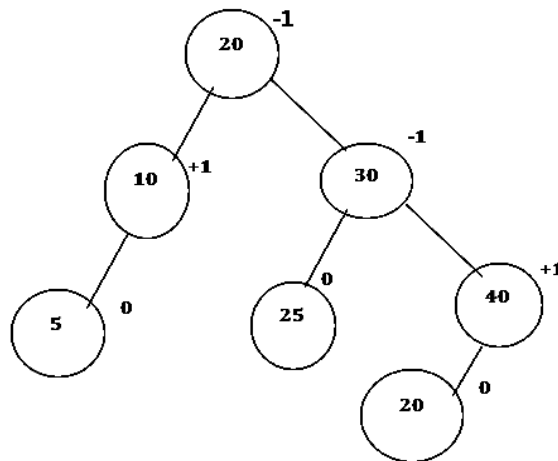


Fig. 11.18: AVL Tree

Operations

The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

- **Insertion**

Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary.

If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation is needed. At most a single or double rotation will be needed to balance the tree.

There are basically four cases which need to be accounted for, of which two are symmetric to the other two. For simplicity, the root of the unbalanced subtree will be called P, the right child of that node will be called R, and the left child will be called L.

If the balance factor of P is 2, it means that the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must then be checked.

If the balance factor of R is 1, it means the insertion occurred on the (external) right side of that node and a left rotation is needed (tree rotation) with P as the root.

If the balance factor of R is -1, this means the insertion happened on the (internal) left side of that node. This requires a double rotation. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

The other two cases are identical to the previous two, but with the original balance factor of -2 and the left subtree outweighing the right subtree.

Only the nodes traversed from the insertion point to the root of the tree need be checked, and rotations are a constant time operation, and because the height is limited to $O(\log(n))$, the execution time for an insertion is $O(\log(n))$.

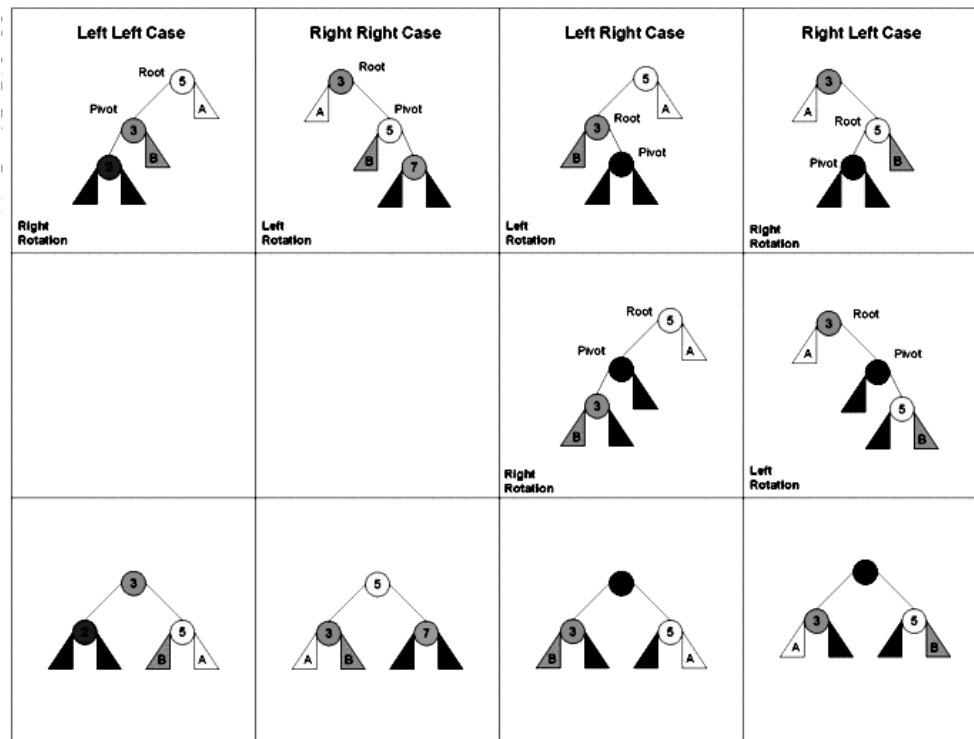


Fig. 11.19: Pictorial description of how rotation causes rebalancing in an AVL Tree

- Deletion**

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves

the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is $O(\log(n))$ for lookup plus maximum $O(\log(n))$ rotations on the way back to the root; so the operation can be completed in $O(\log n)$ time.

- **Lookup**

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes $O(\log n)$ time. No special provisions need to be taken, and the tree's structure is not modified by lookups. (This is in contrast to splay tree lookups, which do modify their tree's structure.)

If each node additionally records the size of its subtree (including itself and its descendants), then the nodes can be retrieved by index in $O(\log n)$ time as well.

Once a node has been found in a balanced tree, the *next* or *previous* node can be obtained in amortized constant time. (In a few cases, about $2 \cdot \log(n)$ links will need to be traversed. In most cases, only a single link needs to be traversed. On the average, about two links need to be traversed.)

11.6 Application of Trees

- Trees are often used in implementing games, particularly board games.
 - Node represents a position in the board.
 - Branches from a node represent all the possible moves from that position.
 - The children represent the new positions.

- Ordered binary trees can be used to represent Expression Evaluation and evaluate arithmetic expressions.
 - if a node is a leaf, the element in it specifies the value.
 - if it is not a leaf, evaluate the children and combine them according to the operation specified by the element.

11.7 Summary

A tree consists of a finite set of elements, called nodes and a finite set of directed lines, called branches that connect the nodes. There are different types of trees such as Binary Tree, Binary Search Tree, Threaded Binary Tree, AVL Tree, Red Black Tree etc. There are number of applications of tree that are also included in this unit.

Self Assessment Questions

1. The unique node with no predecessor called Root. (True / False)
2. Nodes that are not root and not leaf are called as internal node. (True / False)
3. Preorder is a binary traversal. (True / False)
4. Complete Binary Tree and Full Binary Tree are same. (True / False)
5. BST and AVL tree are same. (True / False)
6. AVL Tree is a Balanced Tree. (True / False)

11.8 Terminal Questions

1. Define a binary tree and discuss its properties.
2. Define strictly binary tree.
3. Explain the complete binary tree.
4. Explain tree storage representation using sequential technique with suitable example.
5. List the various operations that can be performed on binary tree.

6. Write a C function to insert an item into a tree.
7. Explain the tree traversal techniques with suitable examples.
8. Write an algorithm for preorder traversal of binary tree.
9. Define BST with its common operations.
10. Explain the steps involved in insertion of item into BST.
11. Define Tree. Explain Binary tree with its properties.
12. Explain the storage representation of Binary tree with sequential representation of a tree.
13. List and describe the various operations on binary tree using linked representation.
14. Explain the types of algorithms for tree traversal.
15. Write a note on Binary Search Tree (BST).
16. Explain the two types of cases to delete node from tree.
17. What is AVL tree? Describe the basic properties of AVL tree.
18. What is red-black tree? List some of its applications.

11.9 Answers to Self Assessment and Terminal Questions

Self Assessment Questions

1. True
2. True
3. True
4. False
5. False
6. False

Terminal Question

1. Section 11.3
2. Section 11.3.4
3. Section 11.3.3
4. Section 11.3.1

5. Section 11.3.2
6. Section 11.3.2
7. Section 11.3.2
8. Section 11.3.2
9. Section 11.3.5
10. Section 11.3.5
11. Sections 11.3 and 11.3.2
12. Section 11.3.2
13. Section 11.3.2
14. Section 11.3.2
15. Section 11.3.5
16. Section 11.3.2
17. Section 11.5
18. Section 11.3.7



Manipal