SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I – Advanced Data Structures – SCS1201

I. BASIC TREE CONCEPTS

TREES:

A tree is a non-linear data structure that is used to represents hierarchical relationships between individual data items. A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., Disjoined) subsets each of which is itself a tree are called sub tree.

Ordinary and Binary Trees Terminology:

Tree: A tree is a finite set of one or more nodes such that, there is a specially designated node called root. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2,..Tn, where each of these set is a tree T1,…Tn are called the subtrees of the root.

Branch: Branch is the link between the parent and its child.

Leaf: A node with no children is called a leaf.

Subtree: A Subtree is a subset of a tree that is itself a tree.

Degree: The number of subtrees of a node is called the degree of the node. Hence nodes that have degree zero are called leaf or terminal nodes. The other nodes are referred as non-terminal nodes.

Children: The nodes branching from a particular node X are called children of X and X is called its parent.

Siblings: Children of the same parent are said to be siblings.

Degree of tree: Degree of the tree is the maximum of the degree of the nodes in the tree.

Ancestors: Ancestors of a node are all the nodes along the path from root to that node. Hence root is ancestor of all the nodes in the tree.

Level: Level of a node is defined by letting root at level one. If a node is at level L, then its children are at level L + 1.

Height or depth: The height or depth of a tree is defined to be the maximum level of any node in the tree.

Climbing: The process of traversing the tree from the leaf to the root is called climbing the tree.

Descending: The process of traversing the tree from the root to the leaf is called descending the tree.

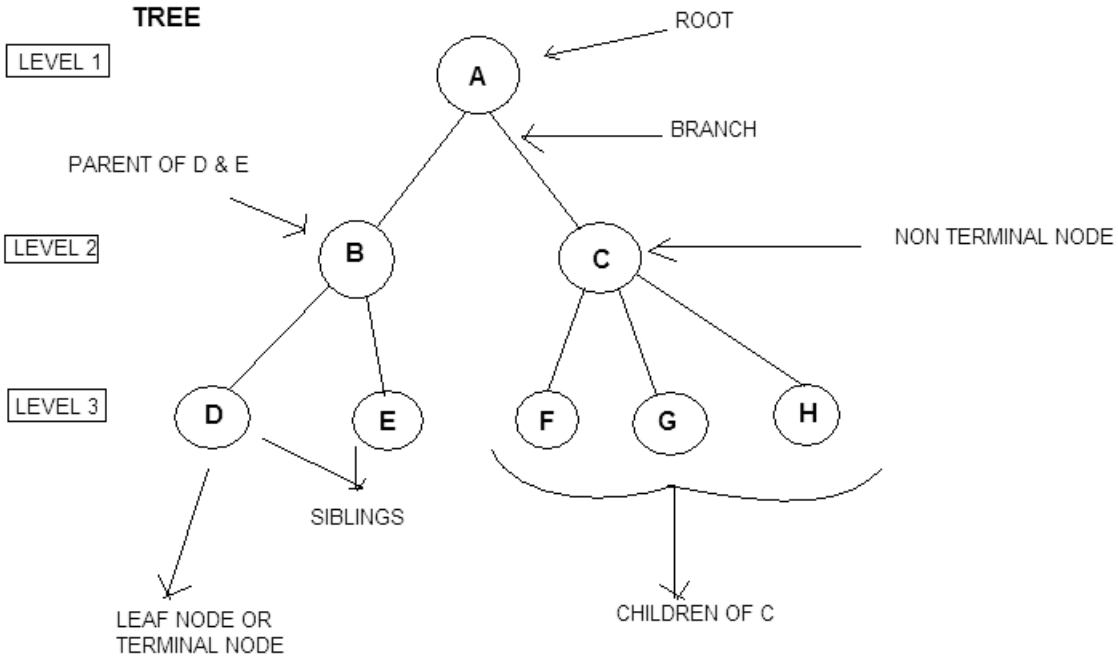


Fig 1.1: Terminologies of Tree

BINARY TREE

Binary tree has nodes each of which has no more than two child nodes.

Binary tree: A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and right subtree.

Left child: The node present to the left of the parent node is called the left child.

Right child: The node present to the right of the parent node is called the right child.

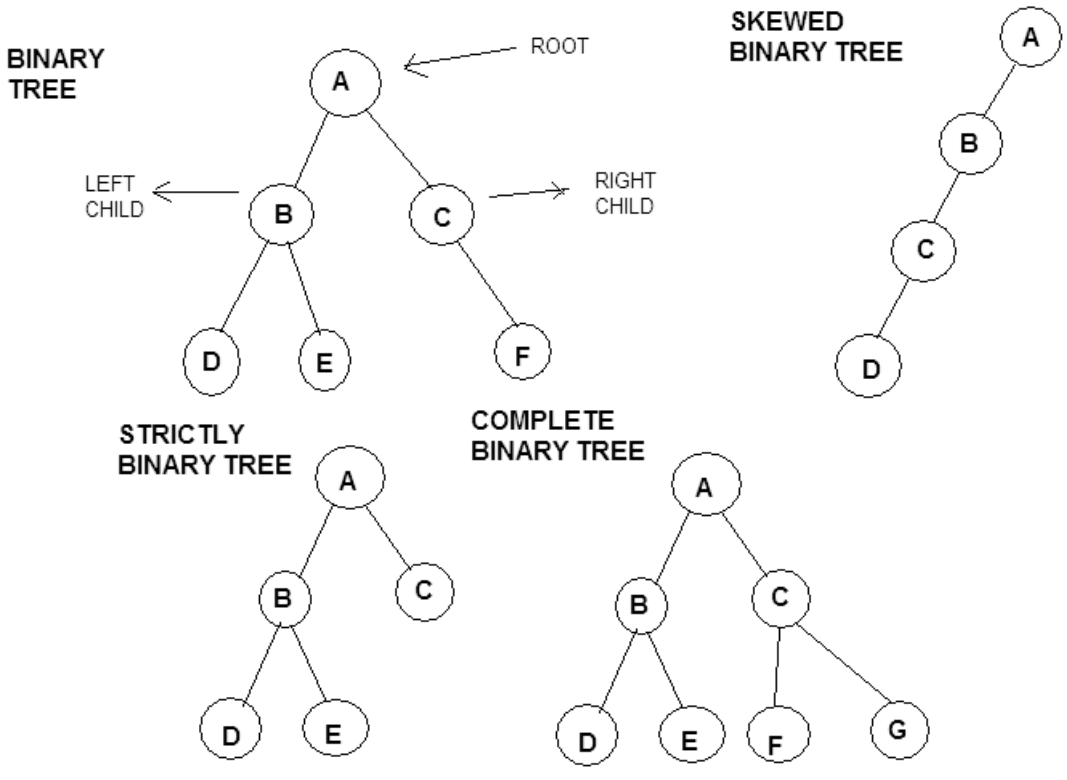


Fig 1.2: Types of Binary Tree

Skewed Binary tree: If the new nodes in the tree are added only to one side of the binary tree then it is a skewed binary tree.
Strictly binary tree: If the binary tree has each node consisting of either two nodes or no nodes at all, then it is called a strictly binary tree.
Complete binary tree: If all the nodes of a binary tree consist of two nodes each and the nodes at the last level does not consist any nodes, then that type of binary tree is called a complete binary tree. It can be observed that the maximum number of nodes on level i of a binary tree is **2i - 1**, where $i \geq 1$. The maximum number of nodes in a binary tree of depth k is **2k – 1**, where $k \geq 1$.

PROPERTIES OF BINARY TREES

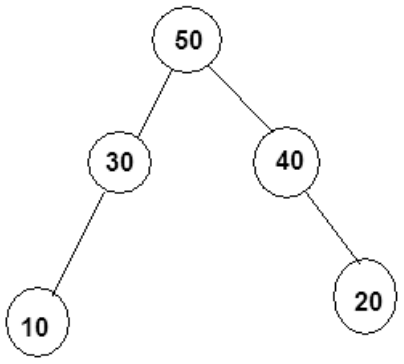
- The number of nodes n in a full binary tree, is at least $n= 2h+1$ and at most $n = 2h+1-1$, where h is the height of the tree.
- A binary tree of n elements has n-1 edges.
- A binary tree of height h has at least h and at most 2h - 1 elements
- A tree consisting of only a root node has a height of 0.
- The number of leaf nodes in a perfect binary tree, is $l= (n+1)/2$. This means that a perfect binary tree with l leaves has $n=2l-1$ nodes.
- The number of internal nodes in a **complete** binary tree of n nodes is n/2.

REPRESENTATION OF BINARY TREES

There are two ways in which a binary tree can be represented. They are:
(i) Array representation of binary trees.
(ii) Linked representation of binary trees.

ARRAY REPRESENTATION OF BINARY TREES

When arrays are used to represent the binary trees, then an array of size 2k is declared where, k is the depth of the tree. For example, if the depth of the binary tree is 3, then maximum 23 - 1 = 7 elements will be present in the node and hence the array size will be 8. This is because the elements are stored from position one leaving the position 0 vacant. But generally, an array of bigger size is declared so that later new nodes can be added to the existing tree. The following binary tree can be represented using arrays as shown.



50	30	40	10	20

Fig 1.3: Array Representation of Binary Tree

The root element is always stored in position 1. The left child of node i is stored in position 2i and right child of node is stored in position 2i + 1. Hence the following formulae can be used to identify the parent, left child and right child of a particular node.
Parent(i) = i / 2, if i ≠ 1. If i = 1 then i is the root node and root does not have parent.
Left child(i) = 2i, if 2i ≤ n, where n is the maximum number of elements in the tree. If 2i > n, then i has no left child.
Right child(i) = 2i + 1, if 2i + 1 ≤ n. If 2i + 1 > n, then i has no right child.

The empty positions in the tree where no node is connected are represented in the array using -1, indicating absence of a node. Using the formula, we can see that for a node 3, the parent is 3/ 2 _1. Referring to the array locations, we find that 50 is the parent of 40. The left child of node 3 is 2*3 _ 6. But the position 6 consists of -1 indicating that the left child does not exist for the node 3. Hence 50 does not have a left child. The right child of node 3 is 2*3 + 1 _ 7. The position 7 in the array consists of 20. Hence, 20 is the right child of 40.

LINKED REPRESENTATION OF BINARY TREES

In linked representation of binary trees, instead of arrays, pointers are used to connect the various nodes of the tree. Hence each node of the binary tree consists of three parts namely, the info, left and right. The info part stores the data, left part stores the address of the left child and the right part stores the address of the right child. Logically the binary tree in linked form can be represented as shown.

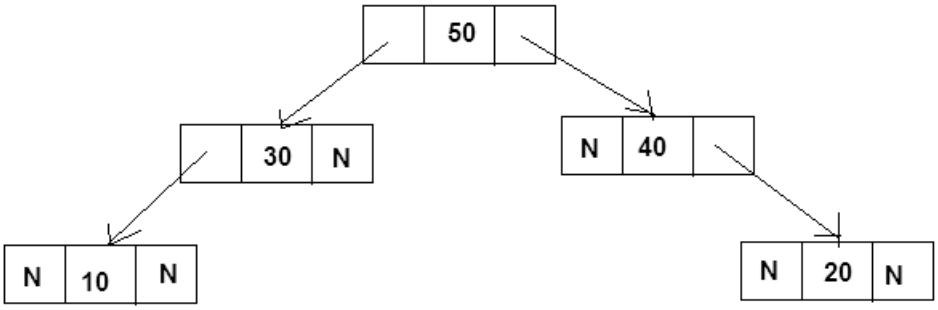


Fig 1.4: Linked Representation

The pointers storing NULL value indicates that there is no node attached to it. Traversing through this type of representation is very easy. The left child of a particular node can be accessed by following the left link of that node and the right child of a particular node can be accessed by following the right link of that node.

BINARY TREE ADT REPRESENTATIONS:

Abstract Data Type (ADT): An Abstract Data Type is a representation in which we provide a specification of the instances as well as of the operations that are to be performed. More precisely, An Abstract Data Type is a data type that is organized in such a way that the specification of the object and the specification of the operation on the object are separated from the representation of the object and the implementation of the operation.
Abstract Datatype BinT(node *root)
{
Instances: Binarytree is a non linear data structure which contains every node except the leaf nodes at most two child nodes.
Operations:

1.Insertion:
This operations is used to insert the nodes in the binary tree.
2.Deletion:
This operations is used to delete any node from the binary tree. Note that if root node is removed the tree becomes empty.
}

BINARY TREE TRAVERSALS
There are three standard ways of traversing a binary tree T with root R. They are:
(i) Preorder Traversal
(ii) Inorder Traversal
(iii) Postorder Traversal

General outline of these three traversal methods can be given as follows:
Preorder Traversal:
(1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.
Inorder Traversal:
(1) Traverse the left subtree of R in inorder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.
Postorder Traversal:
(1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree. The difference between the algorithms is the time at which the root R is processed. The three algorithms are sometimes called, respectively, the node-left-right (NLR) traversal, the left-node-right (LNR) traversal and the left-right-node (LRN) traversal. Now we can present the detailed algorithm for these traversal methods in both recursive method and iterative method.

Traversal algorithms using recursive approach
Preorder Traversal
In the preorder traversal the node element is visited first and then the right subtree of the node and then the right subtree of the node is visited. Consider the following case where we have 6 nodes in the tree A, B, C, D, E, F. The traversal always starts from the root of the tree. The node A is the root and hence it is visited first. The value at this node is processed. The processing can be doing some computation over it or just printing its value. Now we check if there exists any left child for this node if so apply the preorder procedure on the left subtree. Now check if there is any right subtree for the node A, the preorder procedure is applied on the right subtree.
Since there exists a left subtree for node A, B is now considered as the root of the left subtree of A and preorder procedure is applied. Hence, we find that B is processed next and then it is checked if B has a left subtree. This recursive method is continued until all the nodes are visited.

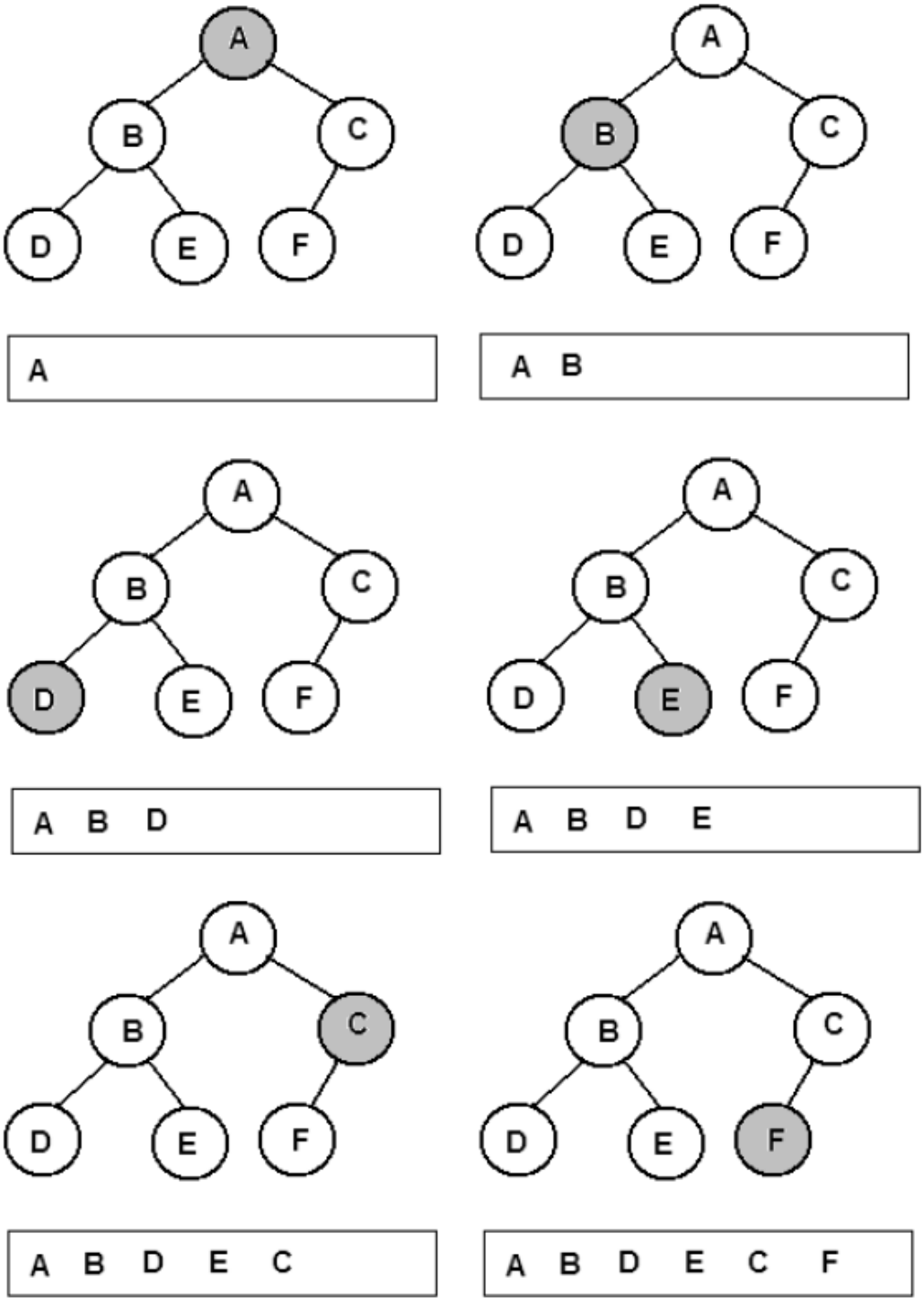


Fig 1.5: Preorder Traversal

The algorithm for the above method is presented in the pseudo-code form below:

Algorithm

PREORDER(ROOT)

```
Temp = ROOT
If temp = NULL
        Return
End if
Print info(temp)
If left(temp) ≠ NULL
        PREORDER( left(temp))
End if
If right(temp) ≠ NULL
        PREORDER(right(temp))
End if
```

End PREORDER

Inorder Traversal

In the Inorder traversal method, the left subtree of the current node is visited first and then the current node is processed and at last the right subtree of the current node is visited. In the following example, the traversal starts with the root of the binary tree. The node A is the root and it is checked if it has the left subtree. Then the inorder traversal procedure is applied on the left subtree of the node A. Now we find that node D does not have left subtree. Hence the node D is processed and then it is checked if there is a right subtree for node D. Since there is no right subtree, the control returns back to the previous function which was applied on B. Since left of B is already visited, now B is processed. It is checked if B has the right subtree. If so apply the inroder traversal method on the right subtree of the node B. This recursive procedure is followed till all the nodes are visited.

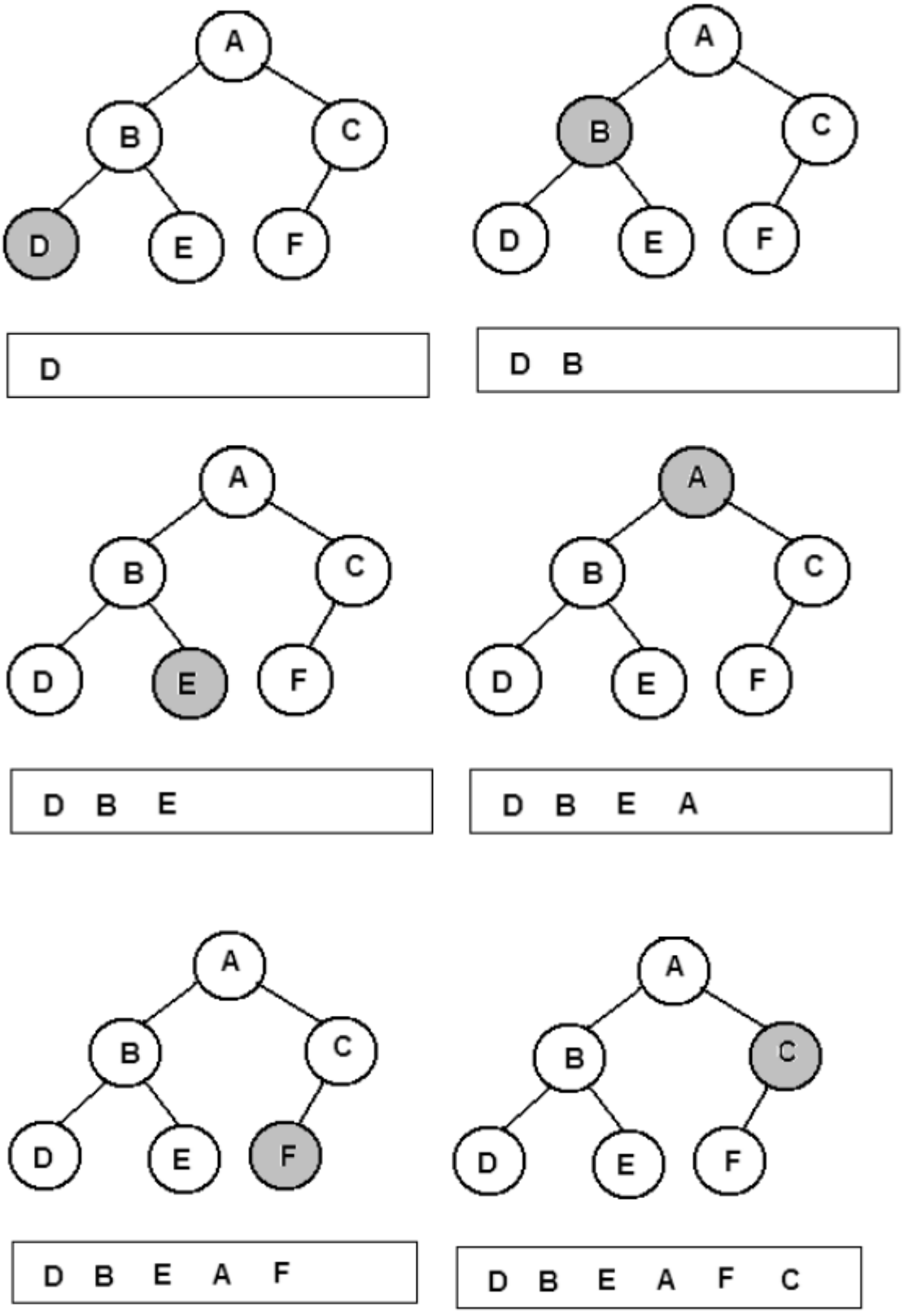


Fig 1.6 : Inorder Traversal

Algorithm
INORDER(ROOT)

```
Temp = ROOT
If temp = NULL
        Return
End if
If left(temp) ≠ NULL
        INORDER(left(temp))
End if
Print info(temp)
If right(temp) ≠ NULL
        INORDER(right(temp))
End if
```

End INORDER

Postorder Traversal

In the postorder traversal method the left subtree is visited first, then the right subtree and at last the current node is processed. In the following example, A is the root node. Since A has the left subtree the postorder traversal method is applied recursively on the left subtree of A. Then when left subtree of A is completely is processed, the postorder traversal method is recursively applied on the right subtree of the node A. If right subtree is completely processed, then the current node A is processed.

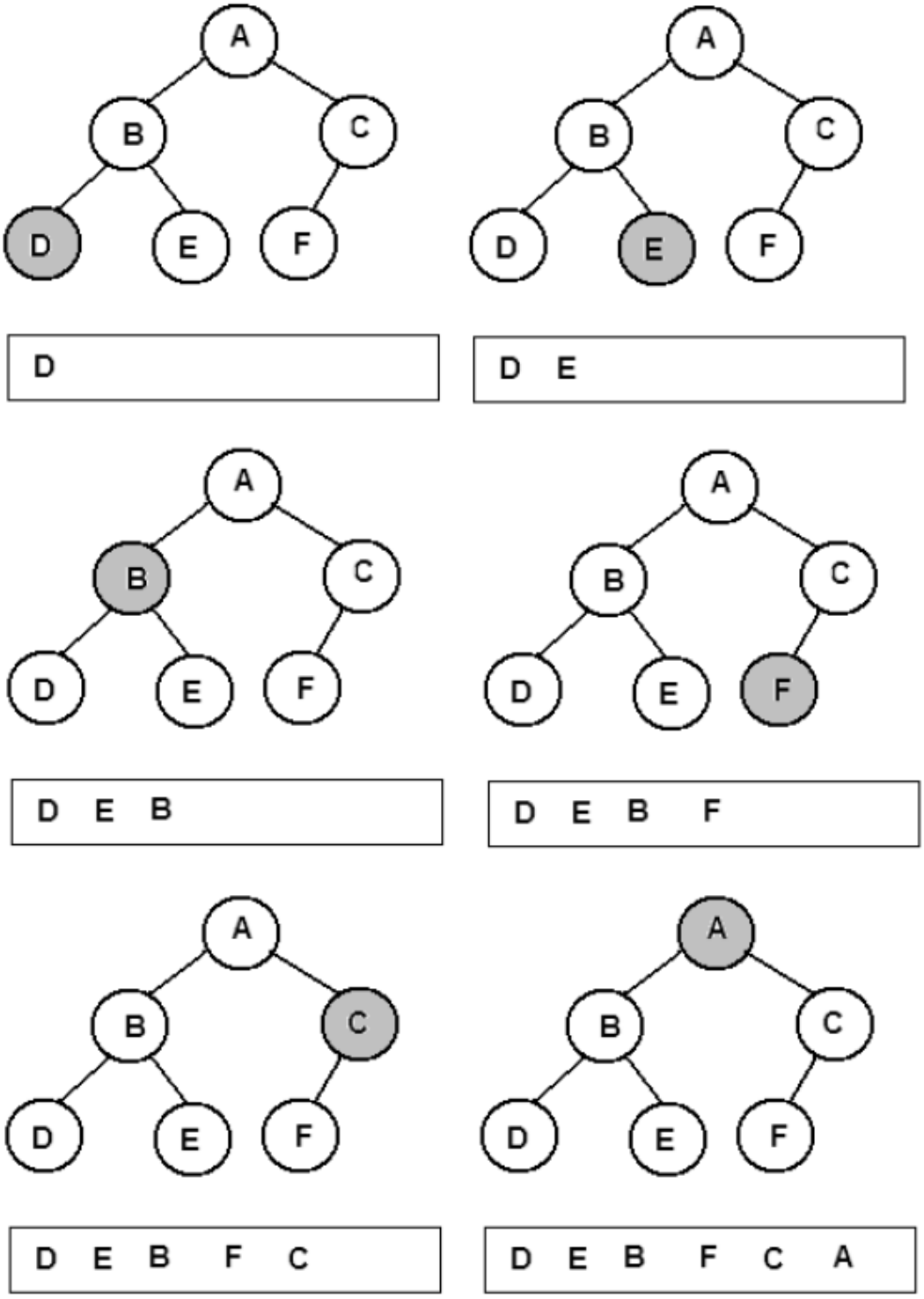


Fig 1.7: Postorder Traversal

Algorithm
POSTORDER(ROOT)

```
Temp = ROOT
If temp = NULL
        Return
End if
If left(temp) ≠ NULL
        POSTORDER(left(temp))
```

End if
If right(temp) ≠ NULL
POSTORDER(right(temp))
End if
Print info(temp)

End POSTORDER

Binary Tree Traversal Using Iterative Approach

Preorder Traversal

In the iterative method a stack is used to implement the traversal methods. Initially the stack is stored with a NULL value. The root node is taken for processing first. A pointer temp is made to point to this root node. If there exists a right node for the current node, then push that node into the stack. If there exists a left subtree for the current node then temp is made to the left child of the current node. If the left child does not exist, then a value is popped from the stack and temp is made to point to that node which is popped and the same process is repeated. This is done till the NULL value is popped from the stack.

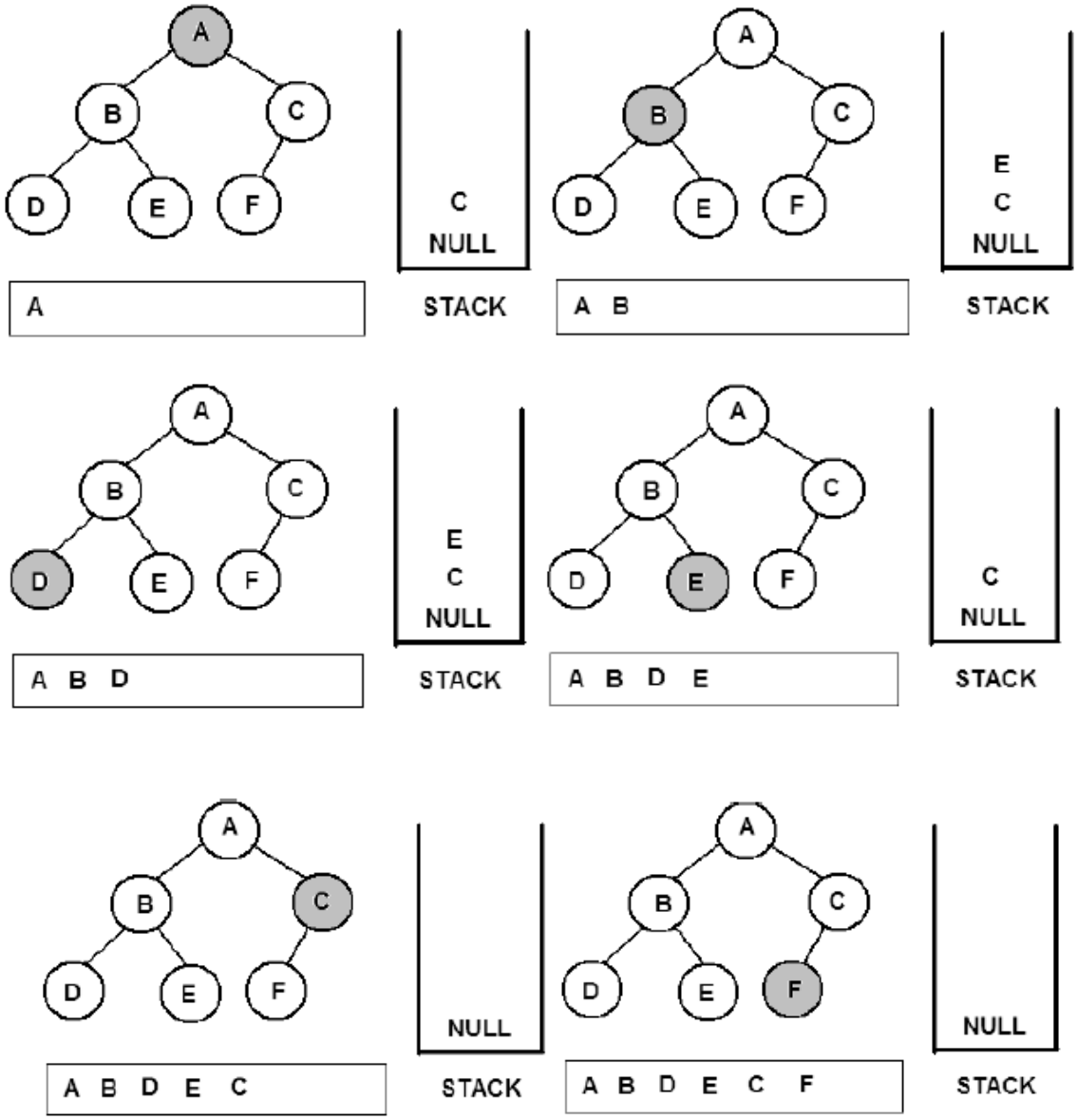


Fig 1.8: Preorder Traversal

Algorithm

PREORDER(ROOT)

```
Temp = ROOT, push(NULL)
While temp ≠ NULL
        Print info(temp)
        If right(temp) ≠ NULL
                Push(right(temp))
        End if
        If left(temp) ≠ NULL
                Temp = left(temp)
        Else
                Temp = pop( )
        End if
End while
```

End PREORDER

Inorder Traversal

In the Inorder traversal method, the traversal starts at the root node. A pointer Temp is made to point to root node. Initially, the stack is stored with a NULL value and a flag RIGHTEXISTS is made equal to 1. Now for the current node, if the flag RIGHTEXISTS = 1, then immediately it is made 0, and the node pointed by temp is pushed to the stack. The temp pointer is moved to the left child of the node if the left child exists. Every time the temp is moved to a new node, the node is pushed into the stack and temp is moved to its left child. This is continued till temp reaches a NULL value.

After this one by one, the nodes in the stack are popped and are pointed by temp. The node is processed and if the node has right child, then the flag RIGHTEXISTS is set to 1 and the process describe above starts from the beginning. Thus, the process stops when the NULL value from the stack is popped.

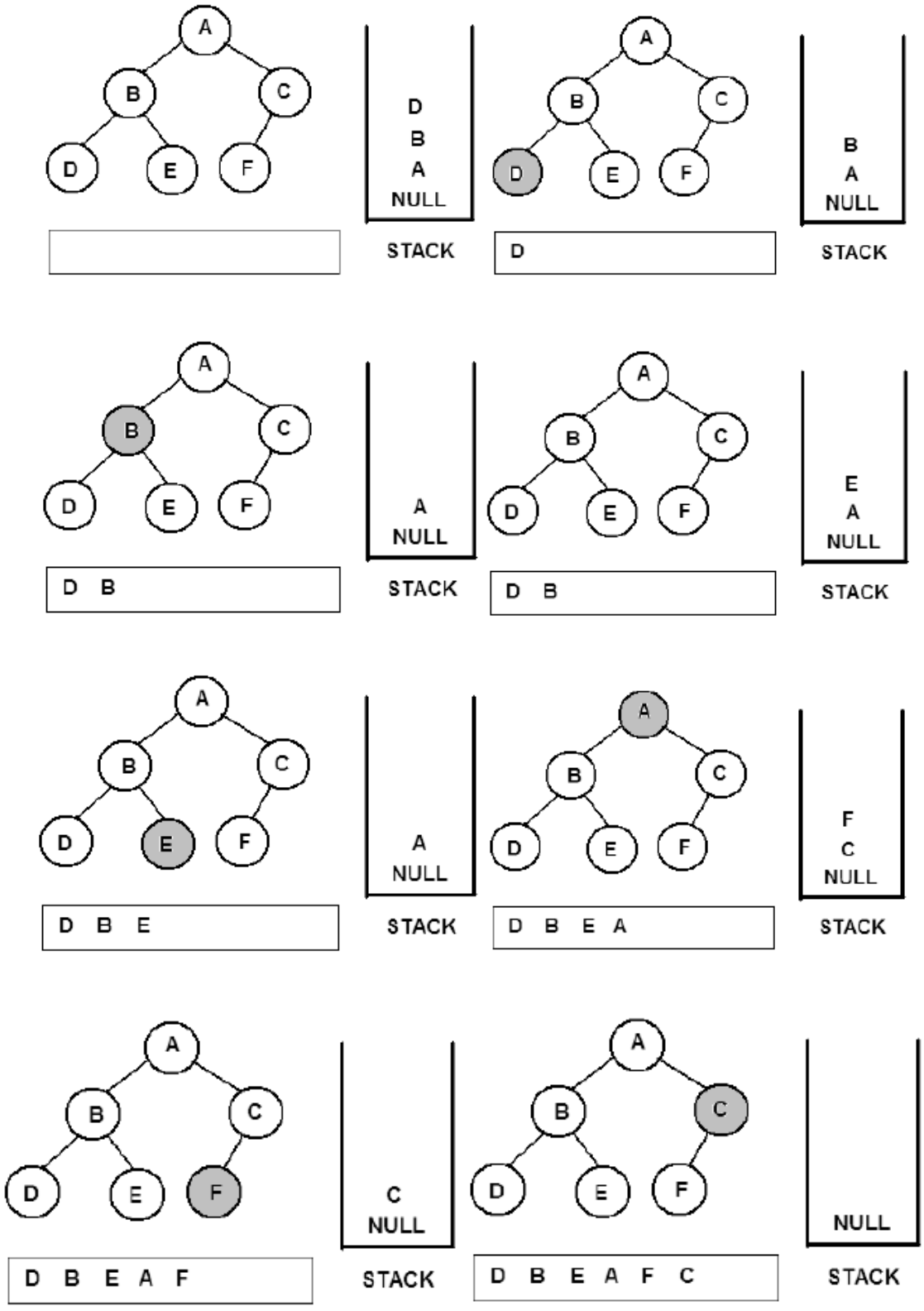


Fig 1.9: Inorder Traversal

Algorithm

INORDER(ROOT)

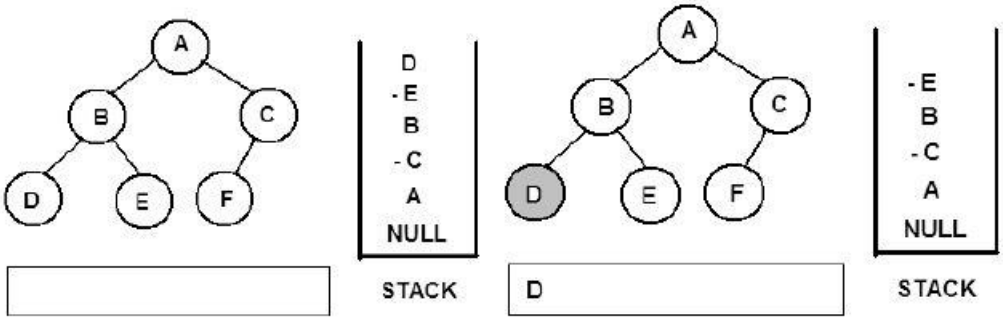
```
Temp = ROOT, push(NULL), RIGHTEXISTS = 1
While RIGHTEXISTS = 1
    RIGHTEXISTS = 0
    While temp ≠ NULL
        Push(temp)
        Temp = left(temp)
    End while
    While (TRUE)
        Temp = pop( )
        If temp = NULL
            Break
        End if
        Print info(temp)
        If right(temp) ≠ NULL
            Temp = right(temp)
            RIGHTEXISTS = 1
            Break
        End if
    End while
End while
```

End INORDER

Postorder Traversal

In the postorder traversal method, a stack is initially stored with a NULL value. A pointer temp is made to point to the root node. A flag RIGHTEXISTS is set to 1. A loop is started and continued until this flag is 1. The current node is pushed into the stack and it is checked if it has a right child. If so, the negative of value of that node is pushed into the stack and the temp is moved to its left child if it exists. This process is repeated till the temp reached a NULL value.

Now the values in the stack are popped one by one and are pointed by temp. If the value popped is positive then that node is processed. If the value popped is negative, then the value is negated and pointed by temp. The flag RIGHTEXISTS is set to 1 and the same above process repeats. This continues till the NULL value from the stack is popped.



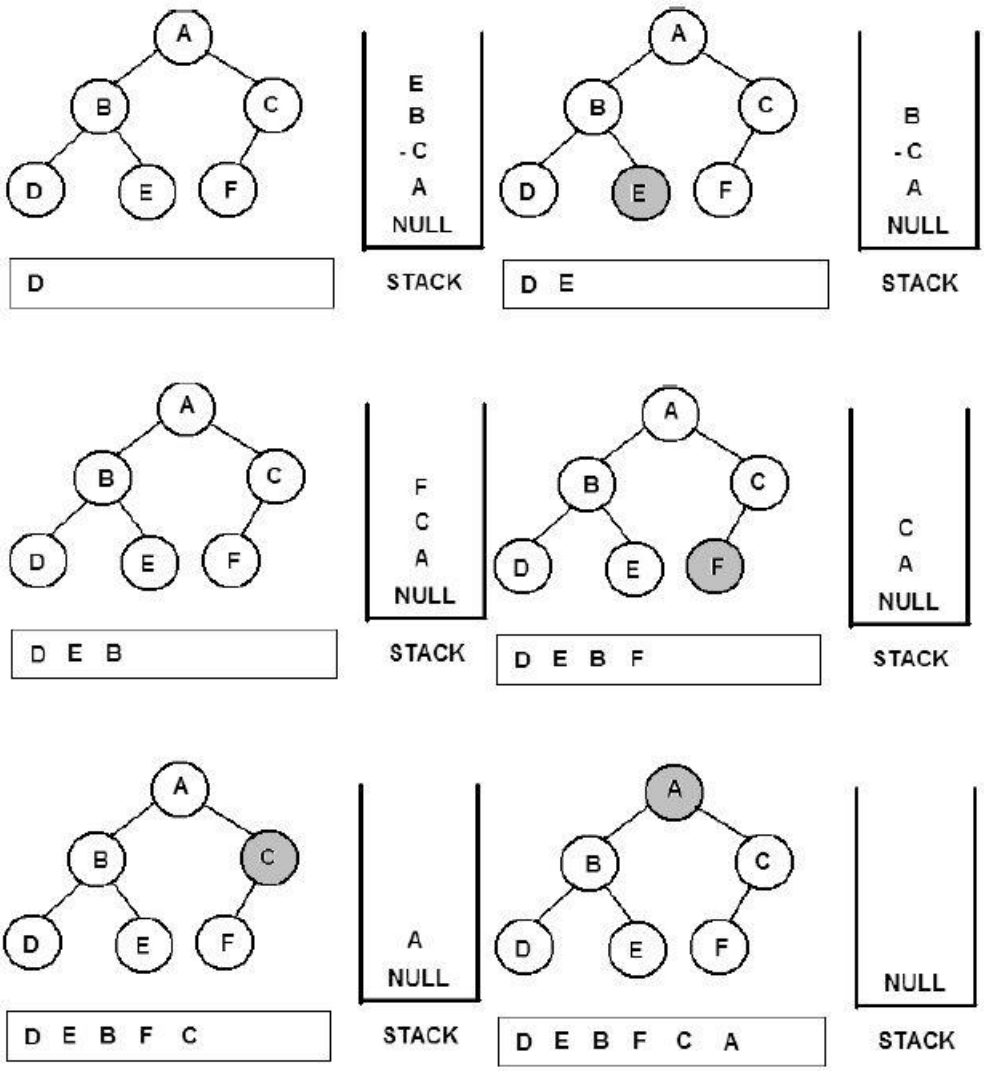


Fig 1.9 : Postorder Traversal

Algorithm

POSTORDER(ROOT)

```
Temp = ROOT, push(NULL), RIGHTEXISTS = 1
While RIGHTEXISTS = 1
        RIGHTEXISTS = 0
        While temp ≠ NULL
                Push (temp)
                If right(temp) ≠ NULL
                        Push( - right(temp))
                End if
                Temp =left(temp)
        End while
        Do
                Temp = pop( )
                If temp > 0
                        Print info(temp)
                End if
```

```
                If temp < 0
                        Temp = -temp
                        RIGHTEXISTS = 1
                        Break
                End if
            While temp ≠ NULL
        End while
End POSTORDER
```

BINARY SEARCH TREES

Binary Search Tree: A Binary tree T is a Binary Search Tree (BST), if each node N of T has the following property : The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

Consider the following tree. The root node 60 is greater than all the elements (54, 23, 58) in its left subtree and is less than all elements in its right subtree (78, 95). Similarly, 54 is greater than its left child 23 and lesser than its right child 58. Hence each and every node in a binary search tree satisfies this property. The reason why we go for a Binary Search tree is to improve the searching efficiency. The average case time complexity of the search operation in a binary search tree is O(log n).

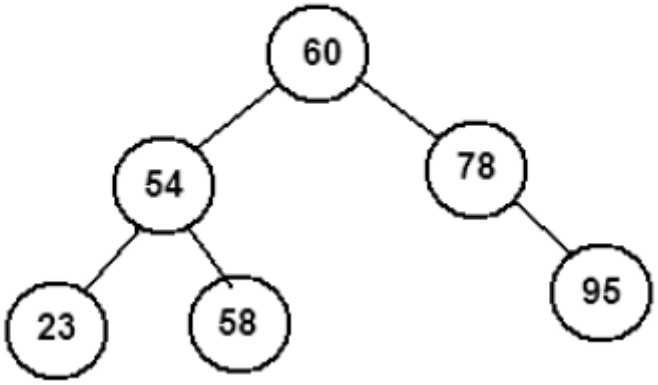


Fig 1.10: Binary Search Tree

Consider the following list of numbers. A binary tree can be constructed using this list of numbers, as shown.

38 14 8 23 18 20 56 45 82 70

Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is compared with 14, and as it is less that 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.

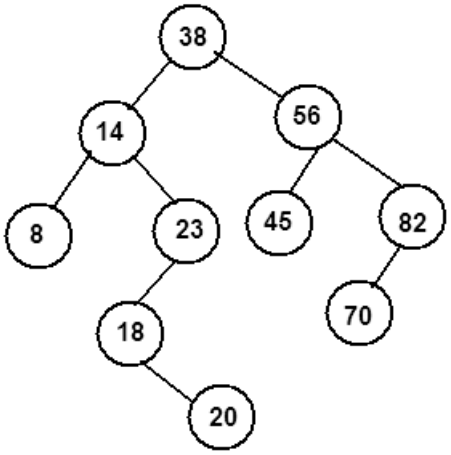


Fig 1.1: Construction of Binary Search Tree

Search Operation in a Binary Search Tree

The search operation on a BST returns the address of the node where the element is found. The pointer LOC is used to store the address of the node where the element is found. The pointer PAR is used to point to the parent of LOC. Initially the pointer TEMP is made to point to the root node. Let us search for a value 70 in the following BST. Let k = 70. The k value is compared with 38. As k is greater than 38, move to the right child of 38, i.e., 56. k is greater than 56 and hence we move to the right child of 56, which is 82. Now since k is lesser than 82, temp is moved to the left child of 82. The k value matches here and hence the address of this node is stored in the pointer LOC.

Every time the temp pointer is moved to the next node, the current node is made pointed by PAR. Hence, we get the address of that node where the k value is found, and also the address of its parent node though PAR.

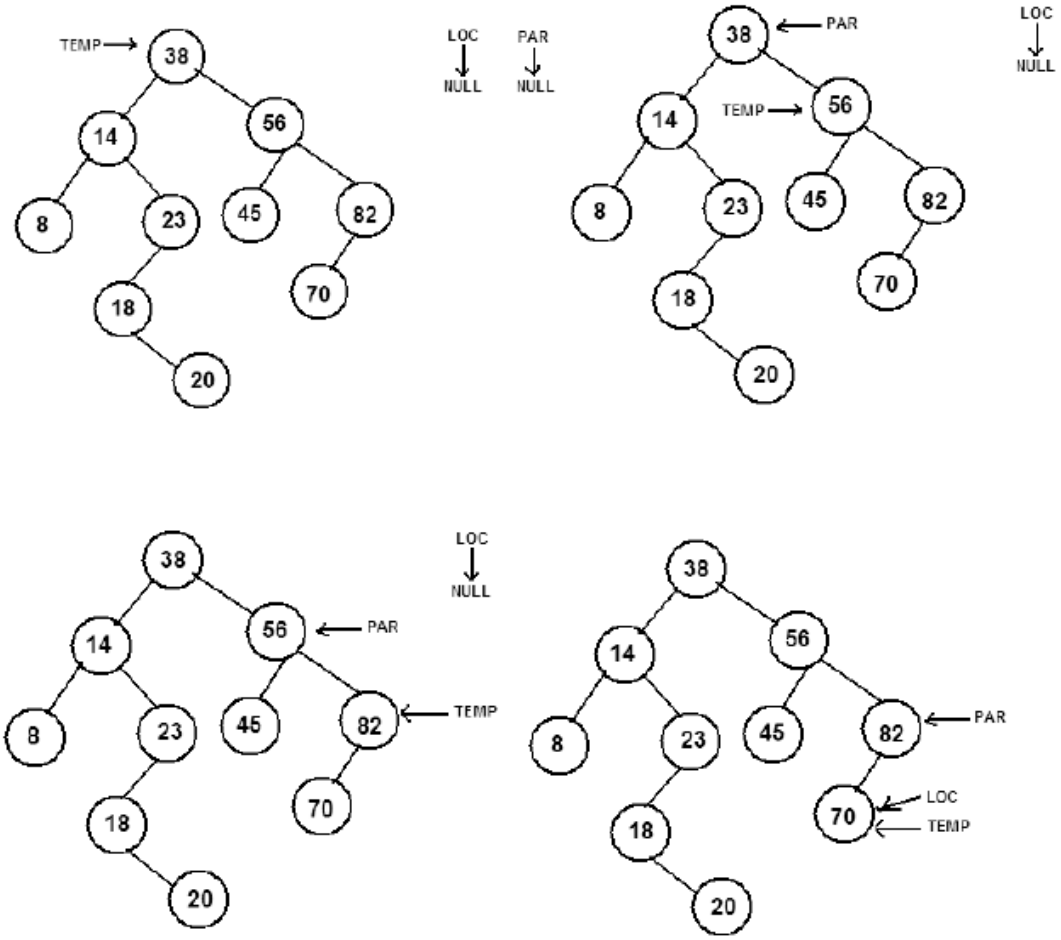


Fig 1.11: Search Operation

Algorithm
SEARCH(ROOT, k)
Temp = ROOT, par = NULL, loc = NULL
While temp ≠ NULL
If k = info(temp)
Loc = temp
Break
End if
If k < info(temp)
Par = temp
Temp = left(temp)
Else
Par = temp
Temp = right(temp)
End if
End while
End SEARCH

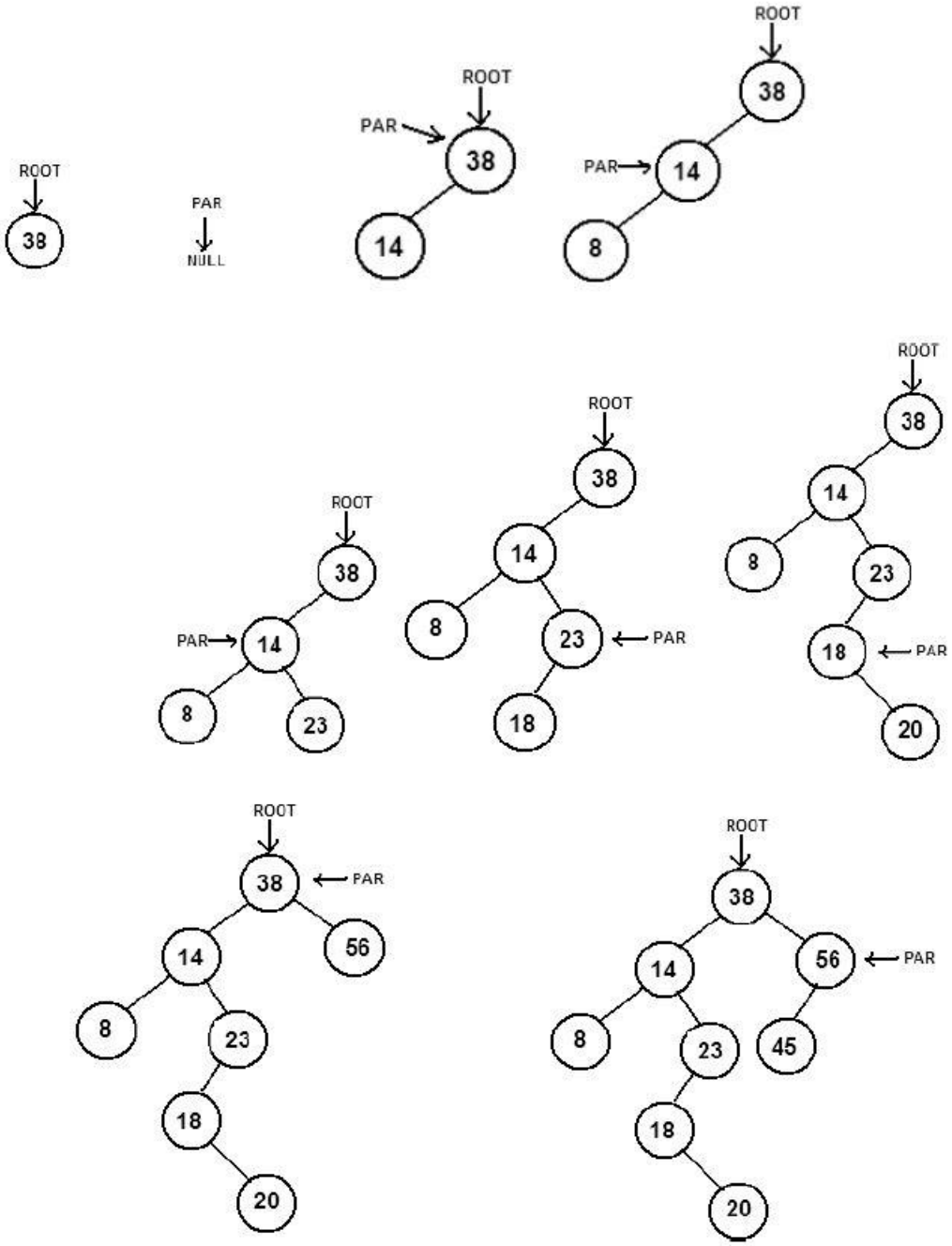
Insert Operation in a Binary Search Tree

The BST itself is constructed using the insert operation described below.

Consider the following list of numbers. A binary tree can be constructed using this list of numbers using the insert operation, as shown.

39 14 8 23 18 20 56 45 82 70

Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is compared with 14, and as it is less than 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.



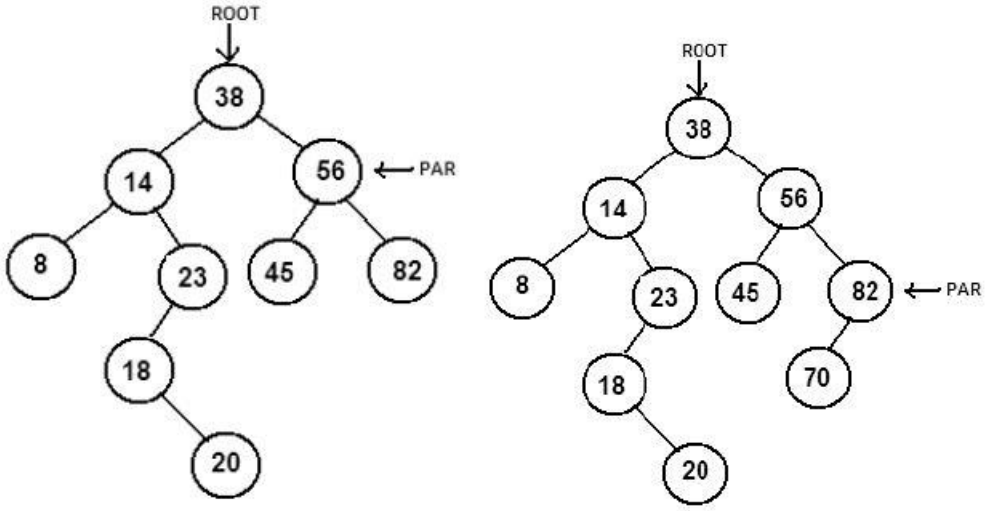


Fig 1.12: Insert Operation

INSERT(ROOT, k)
Temp = ROOT, par = NULL
While temp ≠ NULL
If k = info(temp)
Print "Item already exists!"
Return
End if
If k < info(temp)
Par = temp
Temp = left(temp)
Else
Par = temp
Temp = right(temp)
End if
End while
Info(R) = k, left(R) = NULL, right(R) = NULL
If par = NULL
ROOT = R
End if
If k < info(par)
Left(par) = R
Else
Right(par) = R
End if
End INSERT

Delete Operation in a Binary Search Tree

The delete operation in a Binary search tree follows two cases. In case A, the node to be deleted has no children or it has only one child. In case B, the node to be deleted has both left child and the right child. It is taken care that, even after deletion the binary search tree property holds for all the nodes in the BST.

Algorithm

DELETE(ROOT, k)
SEARCH(ROOT, k)

If Loc = NULL
Print “Item not found”
Return
End if
If right(Loc) ≠ NULL and left(Loc) ≠ NULL
CASEB(Loc, par)
Else
CASEA(Loc, par)
End if
End DELETE

Case A:

The search is operation is performed for the key value that has to be deleted. The search operation, returns the address of the node to be deleted in the pointer LOC and its parents’ address is returned in a pointer PAR. If the node to be deleted has no children then, it is checked whether the node pointed by LOC is left child of PAR or is it the right child of PAR. If it is the left child of PAR, then left of PAR is made NULL else right of PAR is made NULL.

The sequence of steps followed for deleting 20 from the tree is as shown.

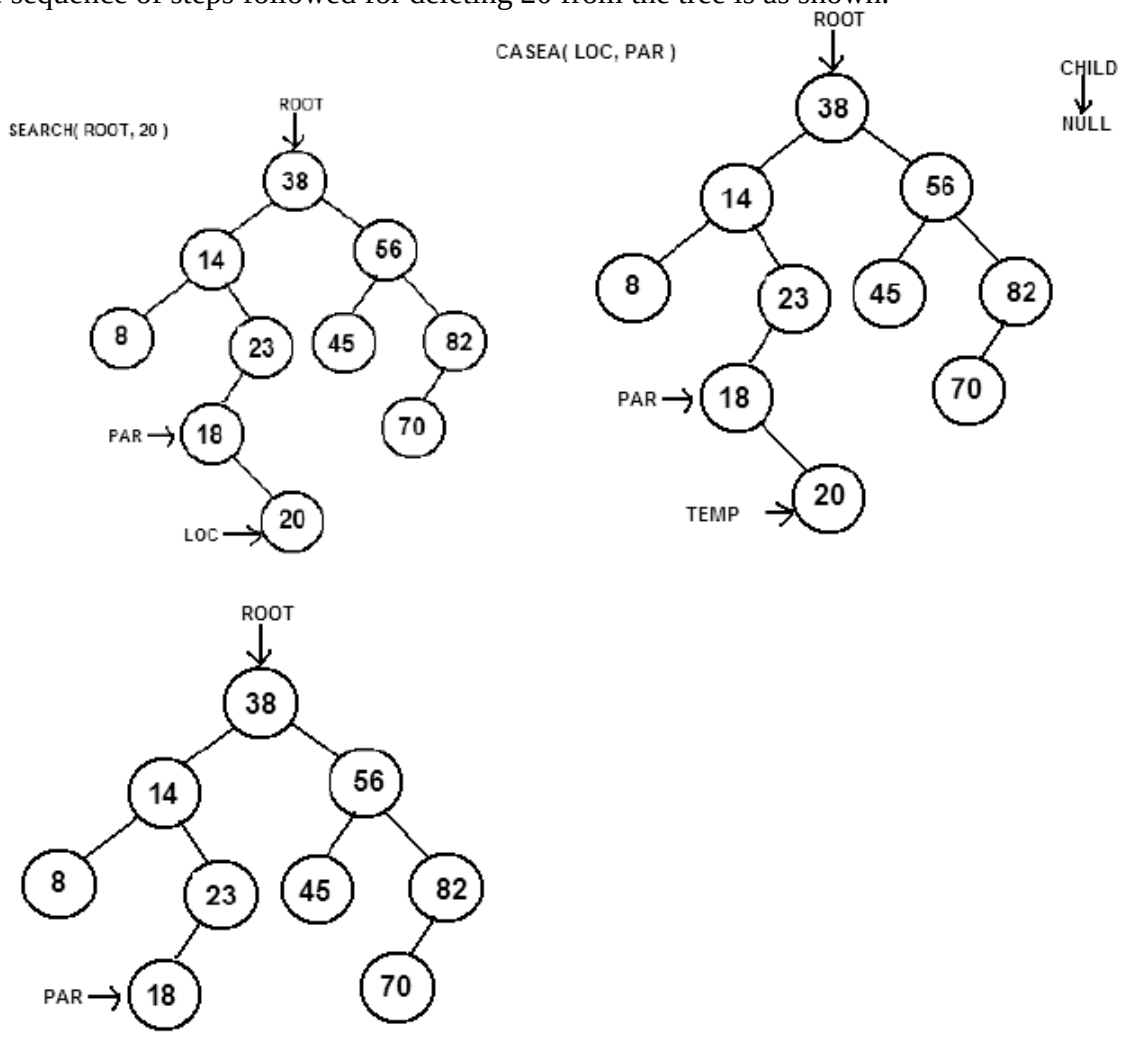


Fig 1.13: Delete Operation

If the node to be deleted has one child, then a new pointer CHILD is made to point to the child of LOC. If LOC is left child of PAR then left of PAR is pointed to CHILD. If LOC is right child of PAR then right of PAR is pointed to CHILD.
The sequence of steps for deleting the node 23 is shown.
FF

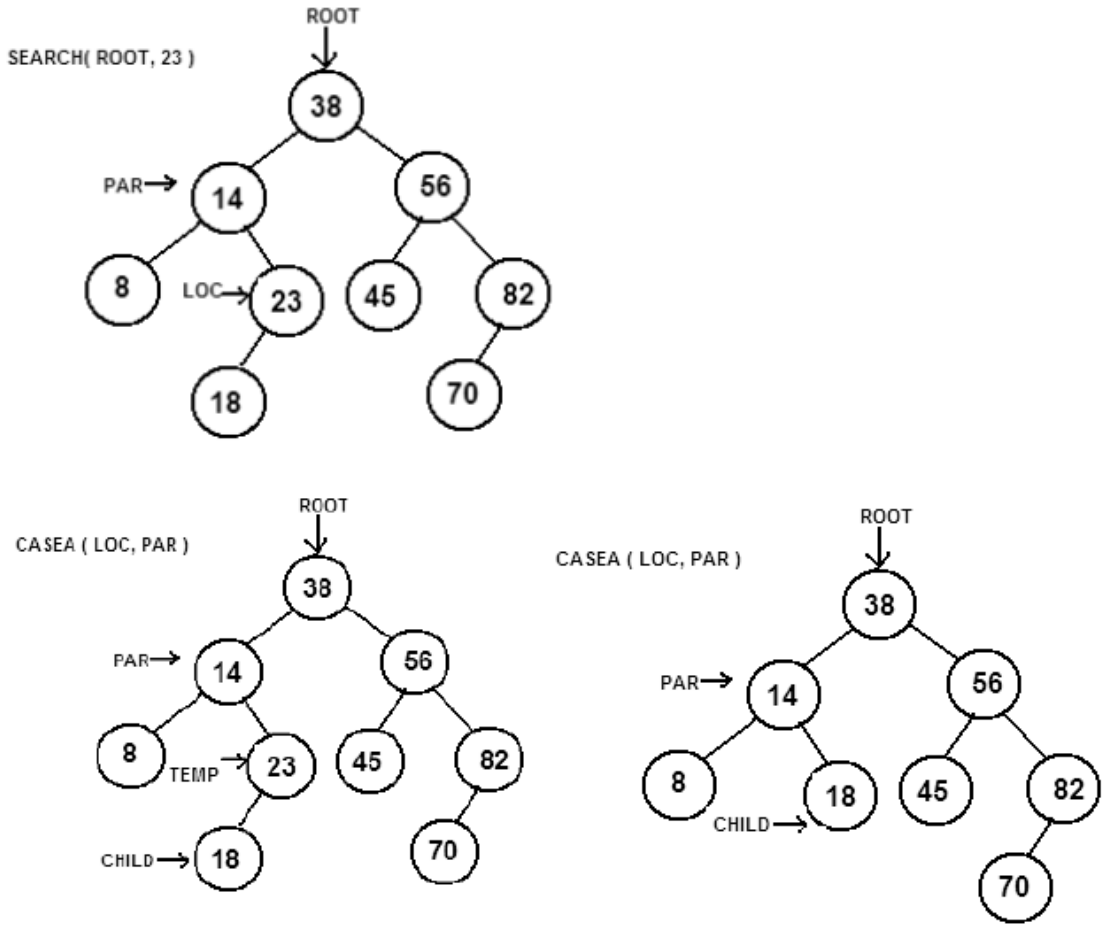
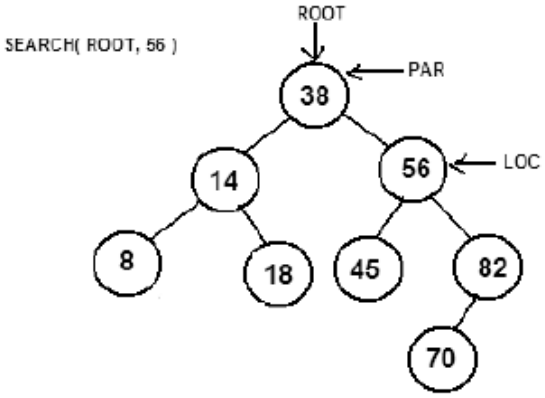
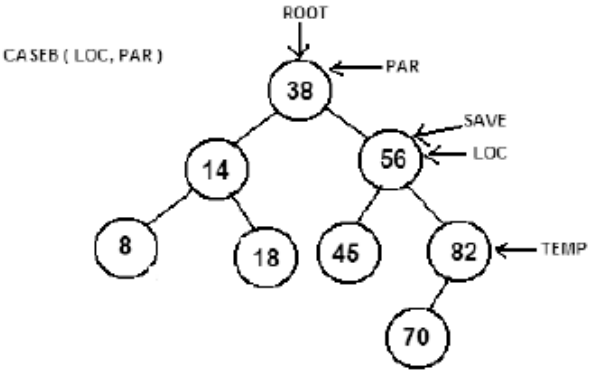


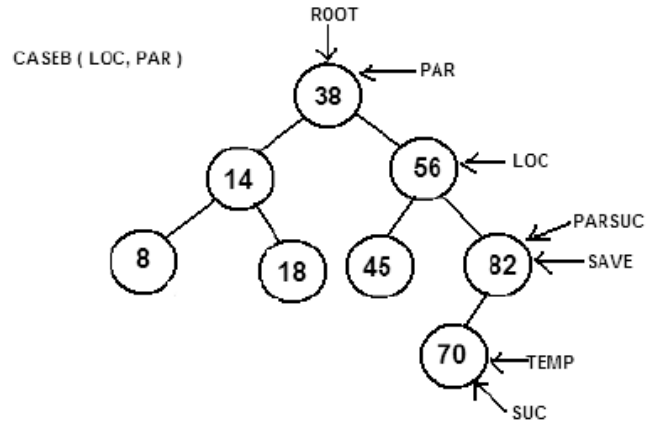
Fig 1.14: Delete Operation

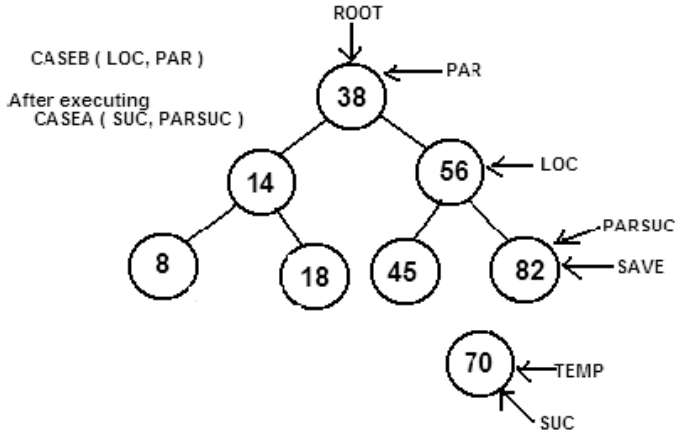
Case B:

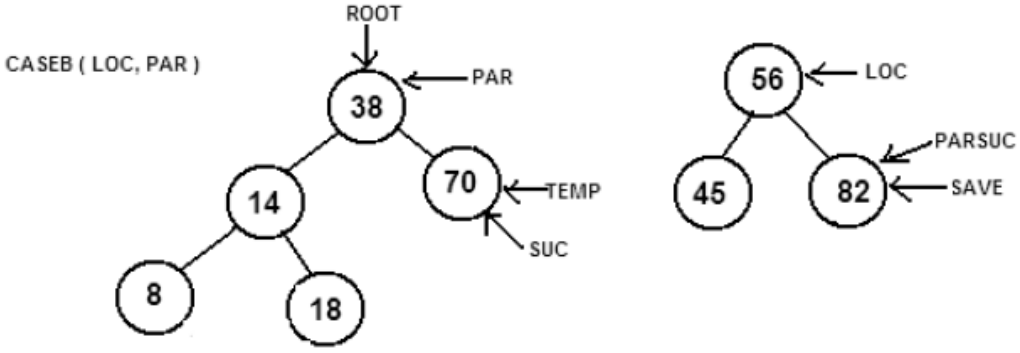
In this case, the node to be deleted has both the left child and the right child. Here we introduce two new pointers SUC and PARSUC. The inorder successor of the node to be deleted is found out and is pointed by SUC and its parent node is pointed by the PARSUC. In the following example the node to be deleted is 56 which has both the left child and the right child. The inorder successor of 56 is 70 and hence it is pointed by SUC. Now the SUC replaces 56 as shown in the following sequence of steps.


SEARCH(ROOT, 56)
ROOT
PAR
LOC
38
14
56
8
18
45
82
70


CASEB (LOC, PAR)
ROOT
PAR
SAVE
LOC
TEMP
38
14
56
8
18
45
82
70


CASEB (LOC, PAR)
ROOT
PAR
38
14
56
LOC
8
18
45
82
PARSUC
SAVE
70
TEMP
SUC


CASEB (LOC, PAR)
After executing
CASEA (SUC, PARSUC)
ROOT
PAR
38
14
56
LOC
8
18
45
82
PARSUC
SAVE
70
TEMP
SUC


CASEB (LOC, PAR)
ROOT
PAR
38
14
70
TEMP
SUC
8
18
56
LOC
PARSUC
45
82
SAVE

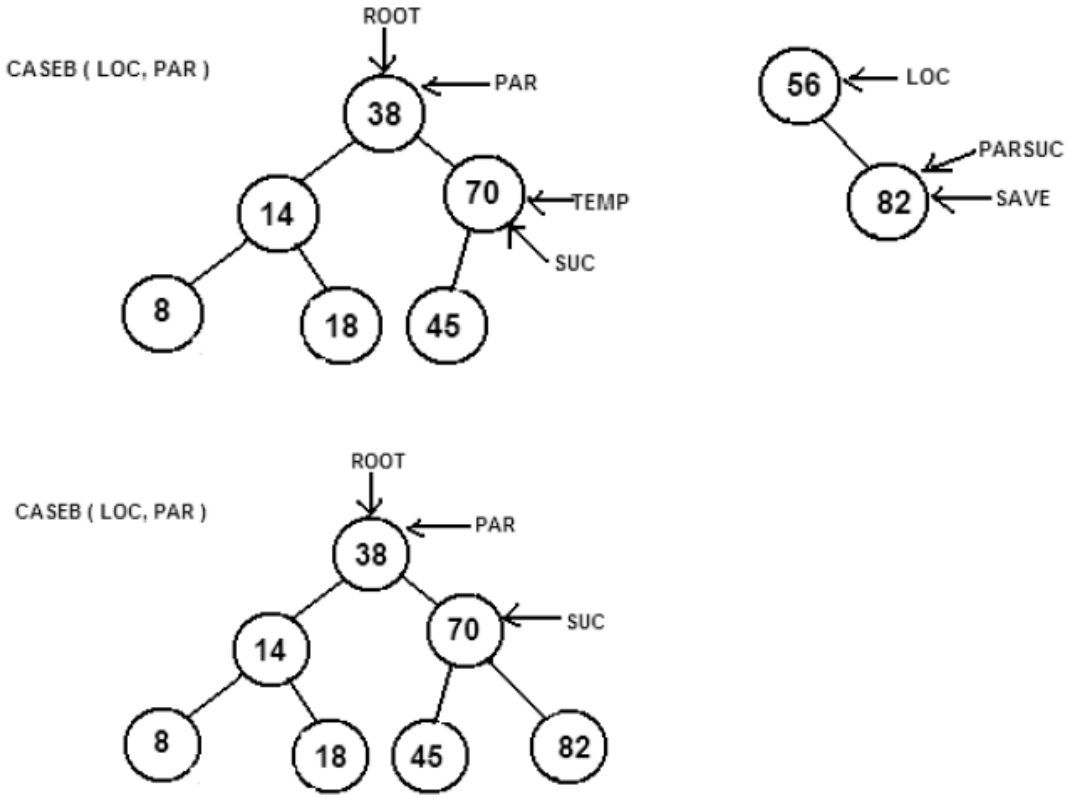


Fig 1.14: All cases of Delete Operation

Algorithm

CASEA(Loc, par)

Temp = Loc
If left(temp) = NULL and right(temp) = NULL
Child = NULL
Else
If left(temp) ≠ NULL
Child = left(temp)
Else
Child = right(temp)
End if
End if
If par ≠ NULL
If temp = left(par)
Left(par) = child
Else
Right(par) = child
End if
Else
ROOT = child
End if
End CASEA

CASEB(Loc, par)

Temp = right(Loc)

```
Save = Loc
While left(temp) ≠ NULL
Save = temp
Temp = left(temp)
End while
Suc = temp
Parsuc = save
CASEA( suc, parsuc)
If par ≠ NULL
If Loc = left(par)
Left(par) = suc
Else
Right(par) = suc
End if
Else
ROOT = suc
End if
Left(suc) = left(Loc)
Right(suc) = right(Loc)
End CASEB
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

Advanced Data Structures – SCS1201

II. ADVANCED TREE CONCEPTS

Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are a greater number of NULL pointer than actual pointers.

Generally, in any binary tree linked list representation, if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL (**N+1 are NULL out of 2N**). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called **threads**.

A threaded binary tree defined as follows:

"A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

Why do we need Threaded Binary Tree?

Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals. Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

Table 2.1: Comparison between a normal binary tree and threaded binary tree

Threaded Binary Tree	Normal Binary Tree
In Threaded Binary Trees, the null pointers are used as threads.	In a normal Binary Tree, the null pointers remain null.
We can use the null pointers which is a efficient way to use memory.	We can't use null pointers so it is a wastage of memory.
Traversal is easy and no need of stack and recursive approach.	Traverse is not easy and not memory efficient.
Structure is complex.	Less complex than threaded binary tree.
Insertion and Deletion takes more time.	It takes minimal time compare to threaded binary tree.

Types of threaded binary trees:

Single Threaded: Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

Double threaded: Each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.

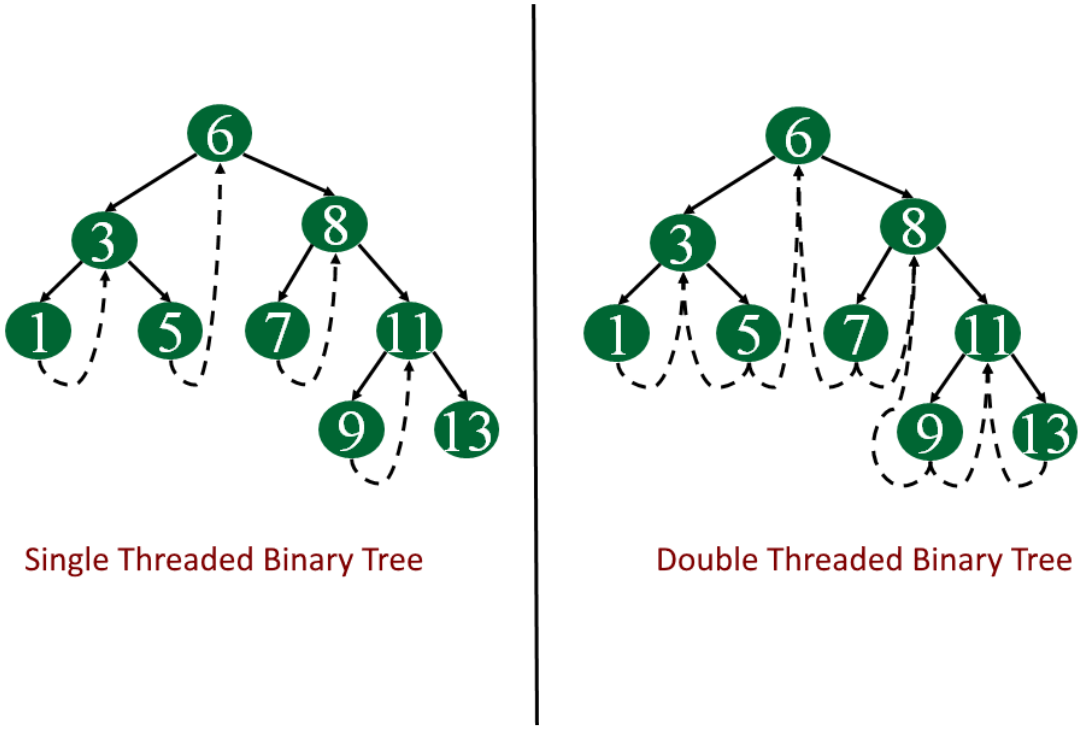
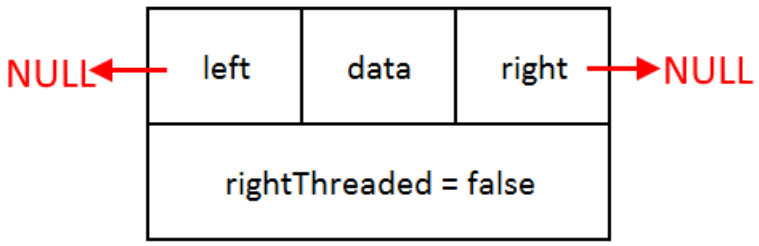


Fig 2.1 : Types of Threaded Binary Tree

Single Threaded: Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

Implementation:

Let's see how the Node structure will look like




```
class Node
{
Node left;
```

```
Node right;
int data;
boolean rightThread;
public Node(int data)
{
this.data = data;
rightThread = false;
}
}
```

In normal BST node we have left and right references and data but in threaded binary tree we have boolean another field called "rightThreaded". This field will tell whether node's right pointer is pointing to its inorder successor, but how, we will see it further.

Operations:

- ✓ Insert node into tree
- ✓ Print or traverse the tree.

Insert():

The insert operation will be quite similar to Insert operation in Binary search tree with few modifications. To insert a node our first task is to find the place to insert the node.

- ✓ Take current = root .
- ✓ Start from the current and compare root.data with n.
- ✓ Always keep track of parent node while moving left or right.
- ✓ if current.data is greater than n that means we go to the left of the root, if after moving to left, the current = null then we have found the place where we will insert the new node. Add the new node to the left of parent node and make the right pointer points to parent node and rightThread = true for new node.

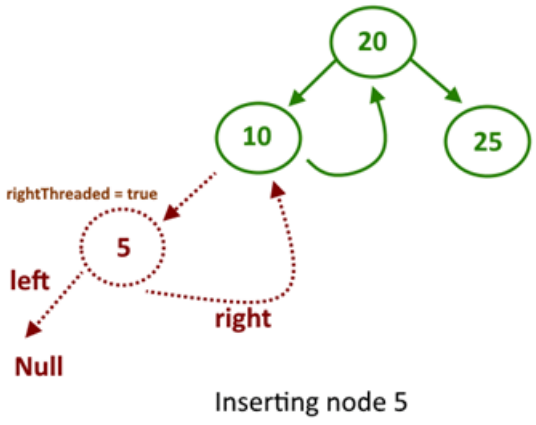


Fig 2.2: Insertion on Single Threaded Binary Tree

- ✓ if current.data is smaller than n that means we need to go to the right of the root, while going into the right sub tree, check rightThread for current node, means right thread is provided and points to the in order successor, if rightThread = false then and current reaches to null, just insert the new node else if rightThread = true then we need to detach the right pointer (store the reference, new node right reference will be point to

it) of current node and make it point to the new node and make the right reference point to stored reference.

Traverse():

- ✓ Traversing the threaded binary tree will be quite easy, no need of any recursion or any
- ✓ stack for storing the node. Just go to the left most node and start traversing the tree using
- ✓ right pointer and whenever rightThread = false again go to the left most node in right subtree.

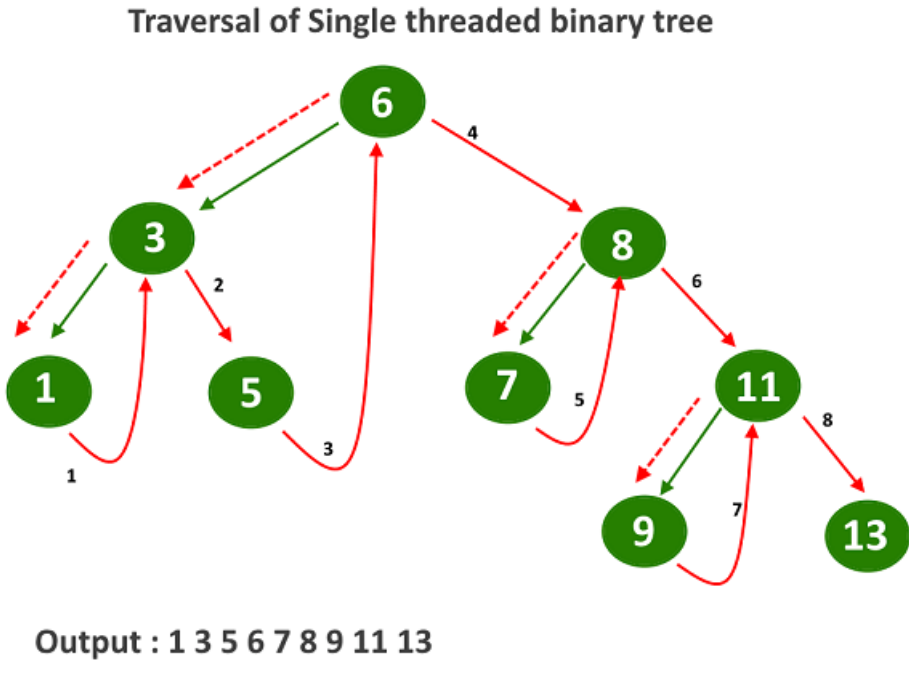


Fig 2.3 : Traversing of Single Threaded Binary Tree

```
Node leftMost(Node n)
{
Node ans = n;
if (ans == null)
{
return null;
}
while (ans.left != null)
 {
ans = ans.left;
}
return ans;
}
void inOrder(Node n)
 {
```

```
Node cur = leftmost(n);
while (cur != null)
 {
print(cur);
if (cur.rightThread)
{
cur = cur.right;
}
 Else
 {
cur = leftmost(cur.right);
}
}
}
```

HEIGHT BALANCED TREES (AVL TREES)

The Height balanced trees were developed by researchers Adelson-Velskii and Landis. Hence these trees are also called AVL trees. Height balancing attempts to maintain the balance factor of the nodes within limit.

Height of the tree: Height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree.

Balance factor: The balance factor of a node is defined to be the difference between the height of the node's left subtree and the height of the node's right subtree.

Consider the following tree. The left height of the tree is 5, because there are 5 nodes (45, 40, 35, 37 and 36) visited in traversing the branch that leads to a leaf node at the deepest level of this tree.

Balance factor = height of left subtree – height of the right subtree

In the following tree the balance factor for each and every node is calculated and shown. For example, the balance factor of node 35 is (0 – 2) = - 2.

The tree which is shown below is a binary search tree. The purpose of going for a binary search tree is to make the searching efficient. But when the elements are added to the binary search tree in such a way that one side of the tree becomes heavier, then the searching becomes inefficient. The very purpose of going for a binary search tree is not served. Hence we try to adjust this unbalanced tree to have nodes equally distributed on both sides. This is achieved by rotating the tree using standard algorithms called the AVL rotations. After applying AVL rotation, the tree becomes balanced and is called the AVL tree or the height balanced tree.

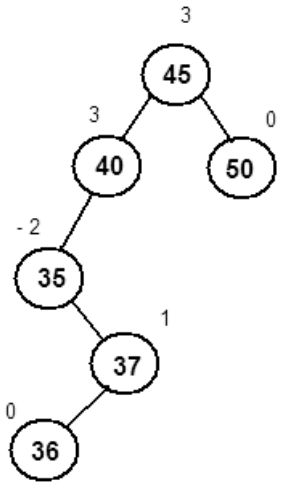


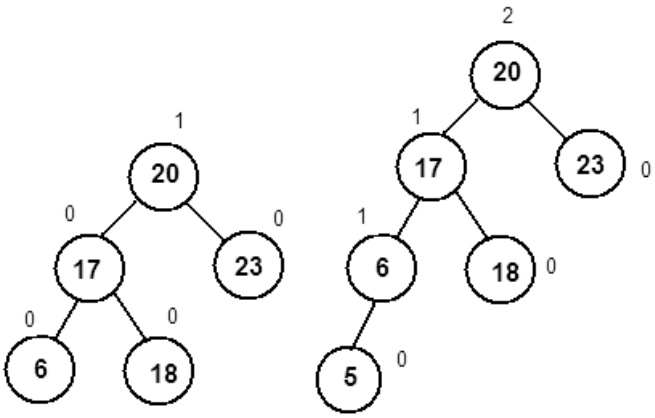
Fig 2.4: Binary Tree

The tree is said to be balanced if each node consists of a balance factor either -1 or 0 or 1. If even one node has a balance factor deviated from these values, then the tree is said to be unbalanced. There are four types of rotations. They are:

1. Left-of-Left rotation.
2. Right-of-Right rotation.
3. Right-of-Left rotation.
4. Left-of-Right rotation.

Left-of-Left Rotation

Consider the following tree. Initially the tree is balanced. Now a new node 5 is added. This addition of the new node makes the tree unbalanced as the root node has a balance factor 2. Since this is the node which is disturbing the balance, it is called the pivot node for our rotation. It is observed that the new node was added as the left child to the left subtree of the pivot node. The pointers P and Q are created and made to point to the proper nodes as described by the algorithm. Then the next two steps rotate the tree. The last two steps in the algorithm calculates the new balance factors for the nodes and is seen that the tree has become a balanced tree.



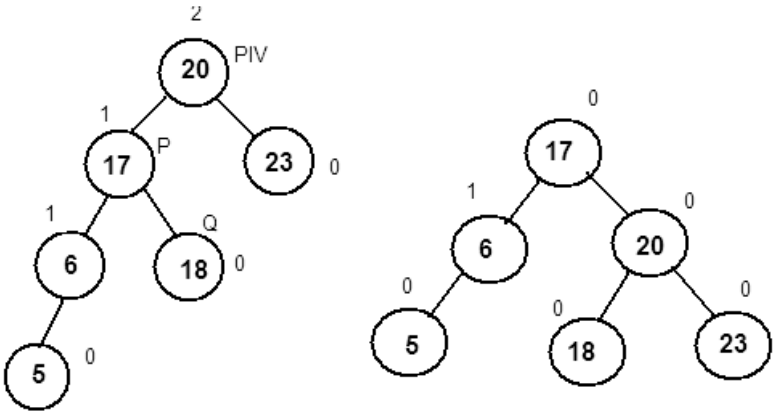


Fig 2.5: Left-of-Left Rotation

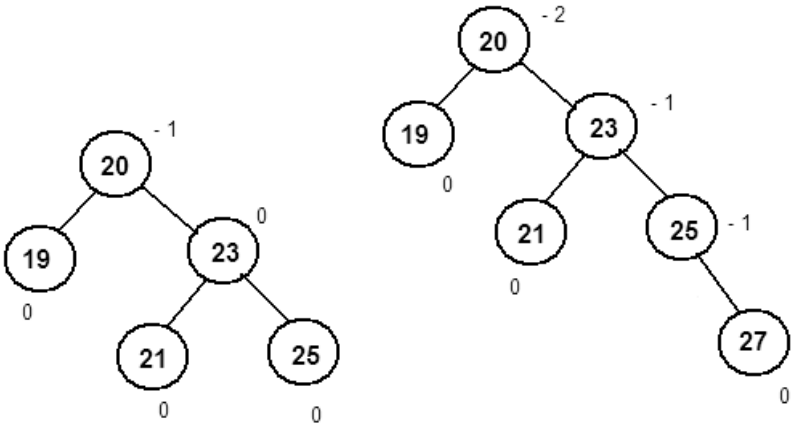
LEFT-OF-LEFT(pivot)

P = left(pivot)
Q = right(P)
Root = P
Right(P) = pivot
Left(pivot) = Q

End LEFT-OF-LEFT

Right-of- Right Rotation

In this case, the pivot element is fixed as before. The new node is found to be added as the right child to the right subtree of the pivot element. The first two steps in the algorithm sets the pointer P and Q to the correct positions. The next two steps rotate the tree to balance it. The last two steps calculate the new balance factor of the nodes.



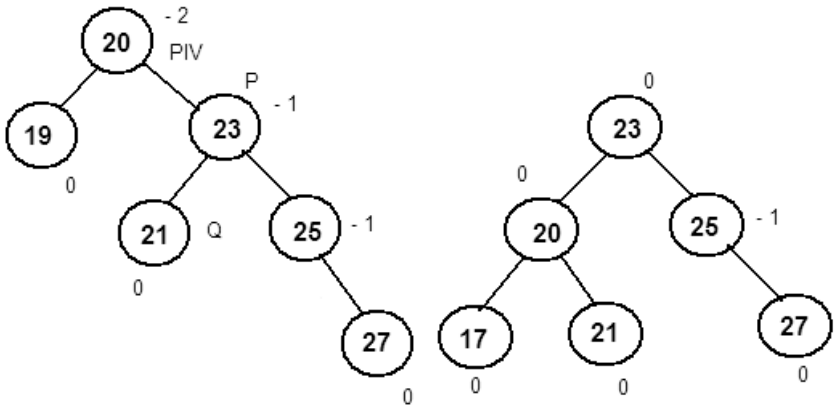


Fig 2.6: Right-of-Right Rotation

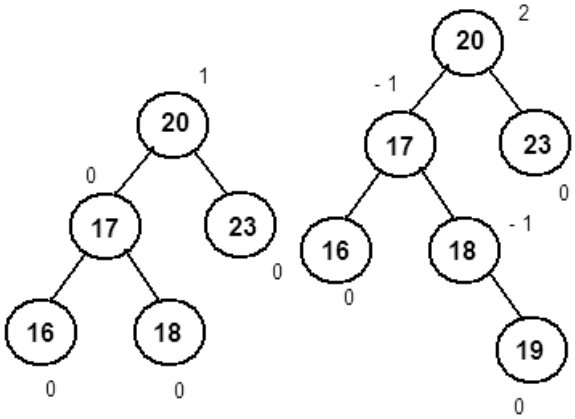
RIGHT-OF-RIGHT(pivot)

P = right(pivot)
Q = left(P)
Root = P
Left(P) = pivot
Right(pivot) = Q

End RIGHT-OF-RIGHT

Right-of-Left Rotation

In this following tree, a new node 19 is added. This is added as the right child to the left subtree of the pivot node. The node 20 fixed as the pivot node, as it disturbs the balance of the tree. In the first two steps the pointers P and Q are positioned. In the next four steps, tree is rotated. In the remaining steps, the new balance factors are calculated.



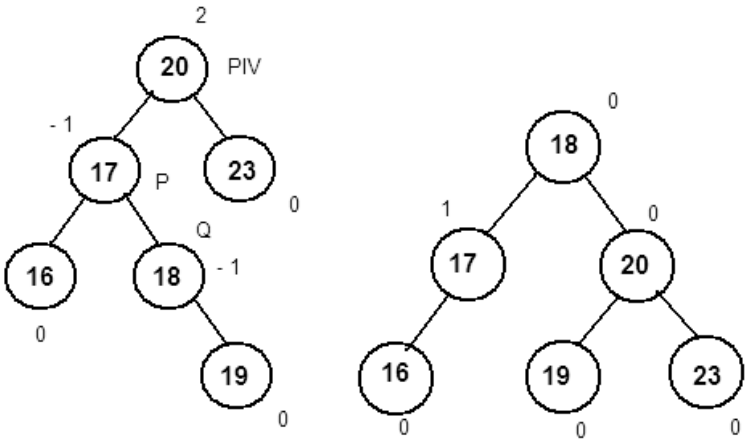


Fig 2.7: Right-of-Left Rotation

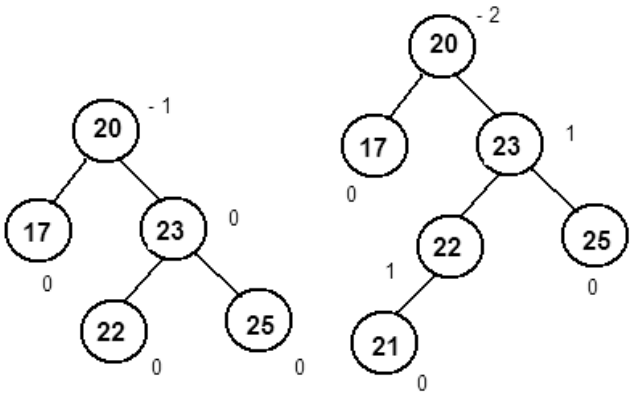
RIGHT-OF-LEFT(pivot)

P = left(pivot)
Q = right(P)
Root = Q
Left(Q) = P
Right(Q) = Pivot
Left(pivot) = right(Q)
Right(P) = left(Q)

End RIGHT-OF-LEFT

Left-of-Right

In the following tree, a new node 21 is added. The tree becomes unbalanced and the node 20 is the node which has a deviated balance factor and hence fixed as the pivot node. In the first two steps of the algorithm, the pointers P and Q are positioned. In the next 4 steps the tree is rotated to make it balanced. The remaining steps calculate the new balance factors for the nodes in the tree.



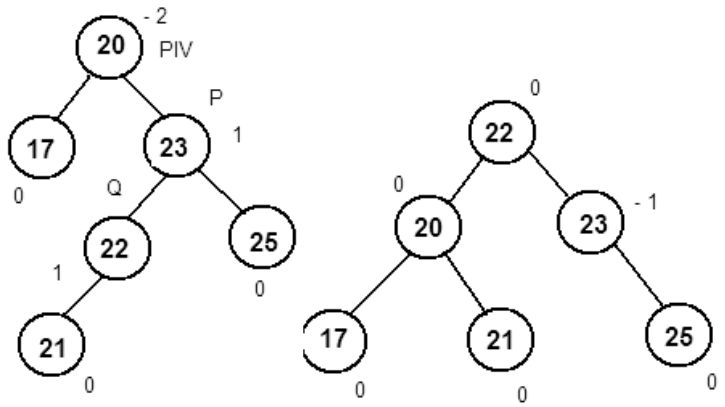


Fig 2.8: Left-of-Right Rotation

LEFT-OF-RIGHT(pivot)

P = right(pivot)
Q = left(P)
Root= Q
Right(Q) = P
Left(Q) = Pivot
Right(pivot) = left(Q)
Left(P) = right(Q)

End LEFT-OF-RIGHT

B – TREES

Multiway search tree (m-way search tree): Multiway search tree of order n is a tree in which any node may contain maximum n-1 values and can have maximum n children.

Consider the following tree. Every node in the tree has one or more than one values stored in it. The tree shown is of order 3. Hence this tree can have maximum 3 children and each node can have maximum 2 values. Hence it is an m-way search tree.

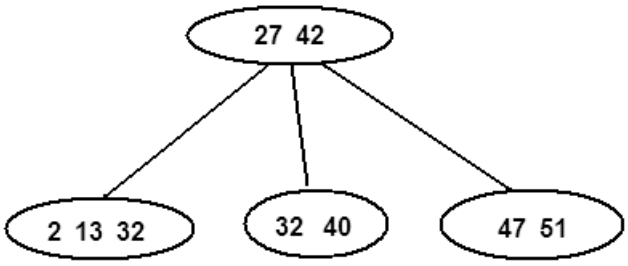


Fig 2.9: 3-way Search Tree

B – Tree:

B – tree is a m-way search tree of order n that satisfies the following conditions.

(i) All non-leaf nodes (except root node) have at least n/2 children and maximum n children.

(ii) The non-leaf root node may have at least 2 children and maximum n children.

(iii) B-Tree can exist with only one node. i.e. the root node containing no child.

(iv) If a node has n children then it must have n-1 values.

(v) All the values that appear on the left most child of a node is smaller than the first value of that node. All values that appears on the right most child of a node is greater than the last values of that node.

(vi) If x and y are any two ith and (i+1)th values of a node, where x < y, then all the values appearing on the (i+1)th sub-tree of that node are greater than x and less than y.
(vii) All the leaf nodes should appear on the same level. All the nodes except root node should have minimum n/2 values.

Consider the following tree. Clearly, it is a m-way search tree of order 3. Let us check whether the above conditions are satisfied. It can be seen that root node has 3 children and therefore has only 2 values stored in it. Also it is seen that the elements in the first child (3, 17) are lesser than the value of the first element (23) of the root node. The value of the elements in the second child (31) is greater than the value of the first element of the root node (23) and less than the value of the second element (39) in the root node. The value of the elements in the rightmost child (43, 65) is greater than the value of the rightmost element in the root node. All the three leaf nodes are at the same level (level 2). Hence all the conditions specified above is found to be satisfied by the given m-way search tree. Therefore, it is a B-Tree.

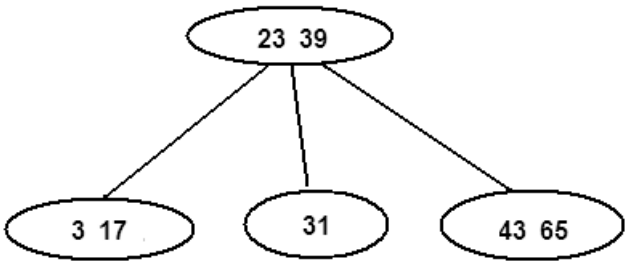


Fig 2.10: B-Tree

Let us say the number to be searched is k = 64. A temporary pointer temp is made to initially point to the root node. The value k = 64 is now compared with each element in the node pointed by temp. If the value is found then the address of the node where it is found is returned using the temp pointer. If the value k is greater than ith element of the node, then the temp is moved to the i+1th node and the search process is repeated. If the k value is lesser than the first value in the node, then the temp is moved to the first child. If the k value is greater than the last value of the node, then temp is moved to the rightmost child of the node and the search process is repeated.

After the particular node where the value is found is located (now pointed by temp), then a variable LOC is initialized to 0, indicating the position of the value to be searched within that node. The value k is compared with each and every element of the node. When the value of the k is found within the node, then the search comes to an end position where it is found is stored in LOC. If not found the value of LOC is zero indicating that the value is not found.

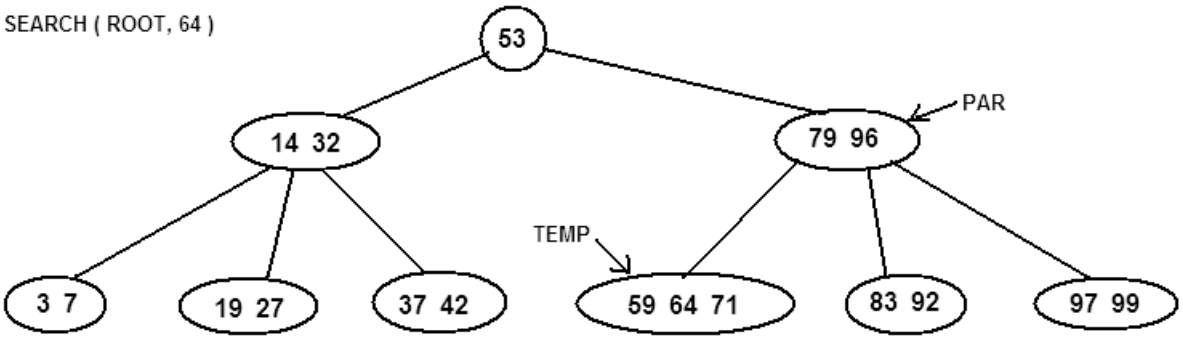


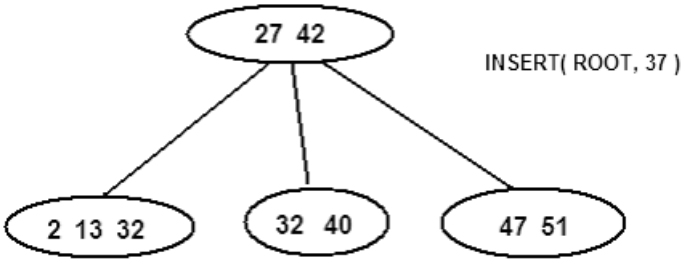
Fig 2.11: Search Operation on B-Tree

```
SEARCH( ROOT, k )
    Temp = ROOT, i = 1, pos = 0
    While i ≤ count(temp) and child[i](temp) ≠ NULL
        If k = info(temp[i])
            Pos = i
            Return temp
        Else If k < info(temp[i])
            SEARCH(child[i](temp), k)
        Else If i = count(temp)
            Par = temp
            Temp = child[i+1](temp)
        Else
            i = i + 1
        End if
    End While
    While i ≤ count(temp)
        If k = info(temp[i])
            Pos = i
            Return temp
        End if
    End while
End SEARCH()
```

Insert Operation in a B-Tree

One of the conditions in the B-Tree is that, the maximum number of values that can be present in the node of a tree is n – 1, where n is the order of the tree. Hence, it should be taken care that, even after insertion, this condition is satisfied. There are two cases: In the first case, the element is inserted into a node which already had less than n- 1 values, and the in the second case, the element is inserted into a node which already had exactly n-1 values. The first case is a simple one. The insertion into the node does not violate any condition of the tree. But in the second case, if the insertion is done, then after insertion, the number values exceeds the limit in that node.

Let us take the first case. In both the cases, the insertion is done by searching for that element in the tree which will give the node where it is to be inserted. While searching, if the value is already found, then no insertion is done as B-Tree is used for storing the key values and keys do not have duplicates. Now the value given is inserted into the node. Consider the figure which shows how value 37 is inserted into correct place.



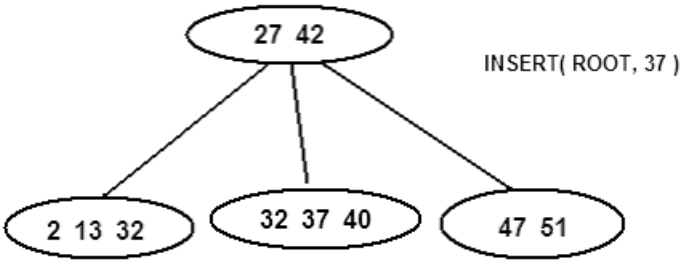


Fig 2.12: Steps for case1of Insertion on B-Tree

In the second case, insertion is done as explained above. But now, it is found that, the number of values in the node after insertion exceeds the maximum limit. Consider the same tree shown above. Let us insert a value 19 into it. After insertion of the value 19, the number of values (2, 13, 19, 22) in that node has become 4. But it is a B-Tree of order 4 in which there should be only maximum 3 values per node. Hence the node is split into two nodes, the first node containing the numbers starting from the first value to the value just before the middle value (first node: 2). The second node will contain the numbers starting just after the mid value till the last value (second node: 19, 22). The mid value 13 is pushed into the parent. Now the adjusted B-Tree appears as shown.

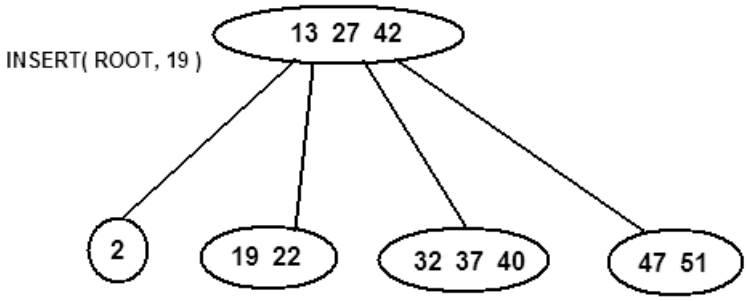


Fig 2.13: Steps for case2 of Insertion on B-Tree

INSERT(ROOT, k)

```
    Temp = SEARCH( ROOT, k )
    If count(temp) < n-1
        Ins(temp, k)
        Return
    Else
        Repeat for i = n/2 +1 to n-1
            Info(R[i-n/2]) = info(temp[i])
            Count(R) = count(R) + 1
        End repeat
        Count(temp) = n/2 – 1
        Ins(par, info(temp[m/2])
    End if
    INSERT(ROOT, k )
```

End INSERT

Delete Operation in B-Tree

When the delete operation is performed, we should take care that even after deletion, the node has minimum n/2 value in it, where n is the order of the tree.

There are three cases:

Case 1: The node from which the value is deleted has minimum n/2 values even after deletion. Let us consider the following B-Tree of order 5. A value 64 is to be deleted. Even after the deletion of the value 64, the node has minimum n/2 values (i.e., 2 values). Hence the rules of the B-Tree are not violated.

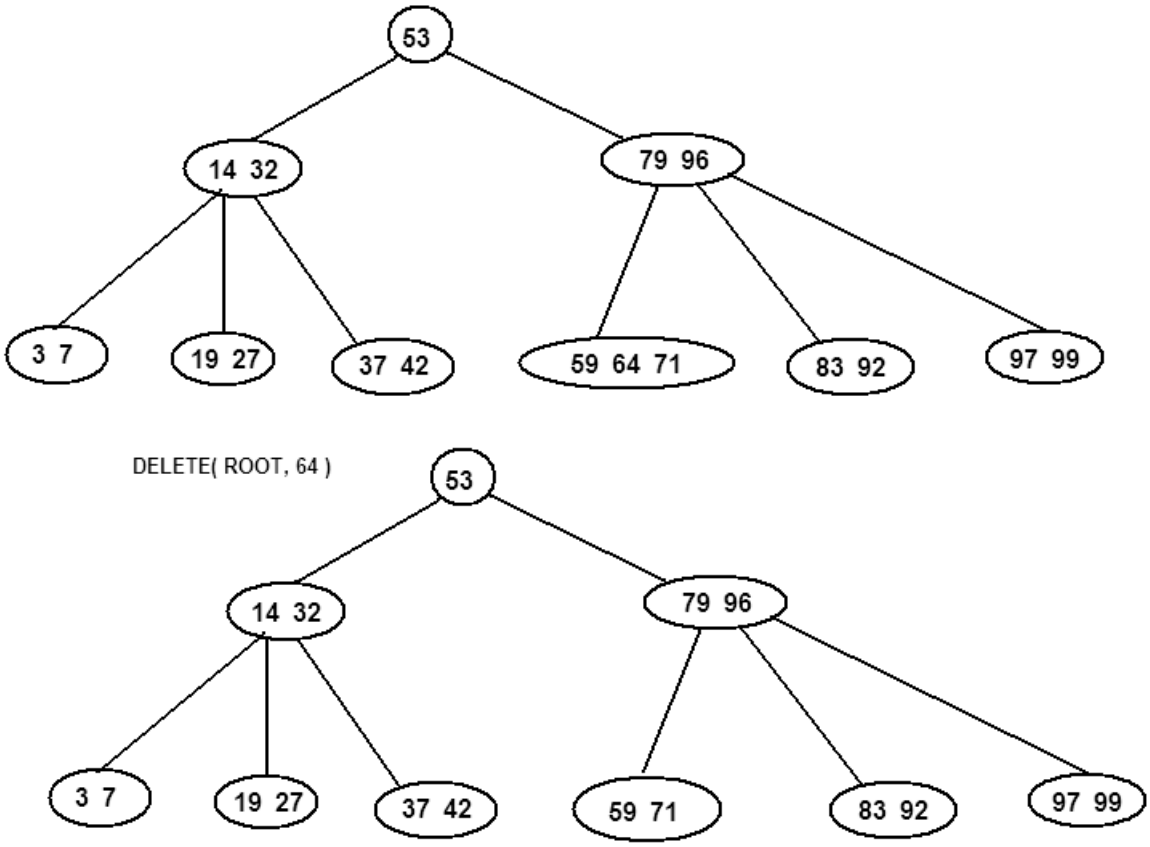


Fig 2.14: case1 of Deletion

Case 2: In the second case, after the deletion the node has less than minimum n/2 values. Let us say we delete 92 from the tree. After 92 is deleted, the node has only one value 83. But a node adjacent to it consist 3 values (i.e., there are extra values in the adjacent node). Then the last value in that node 71 is pushed to its parent and the first value in the parent namely 79 is pushed into the node which has values less than minimum limit. Now the node has obtained the minimum required values.

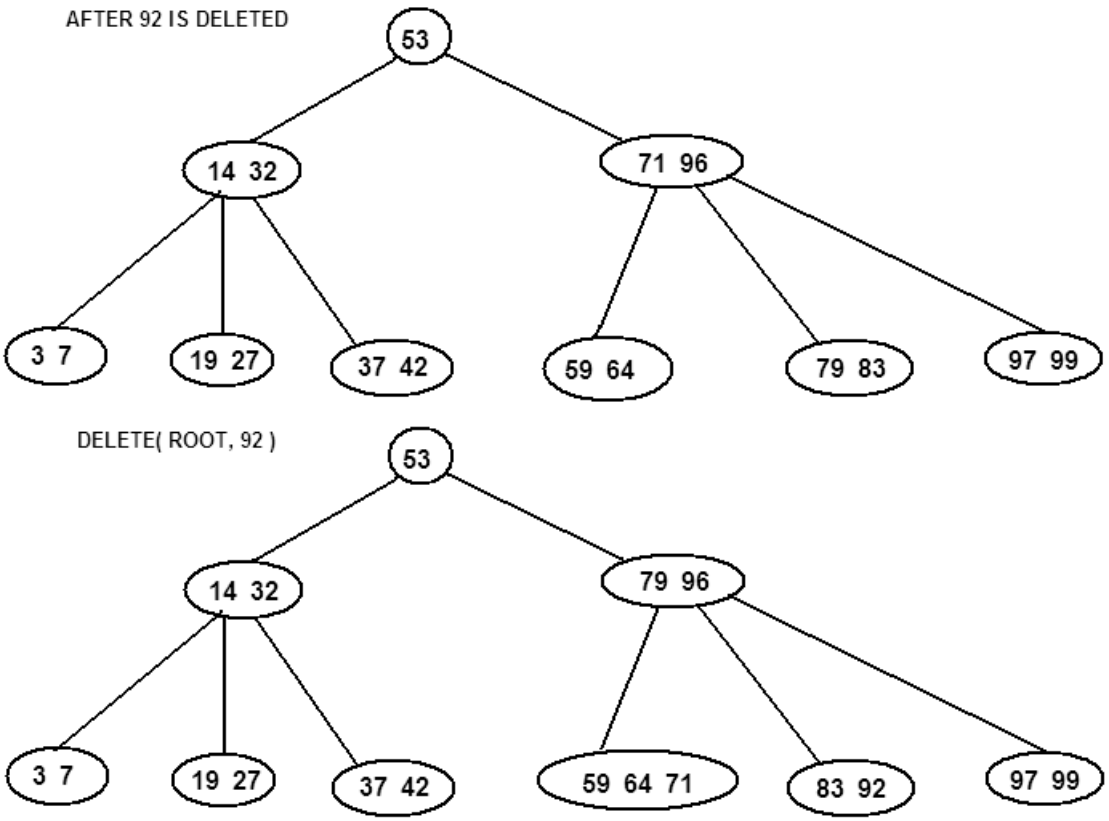
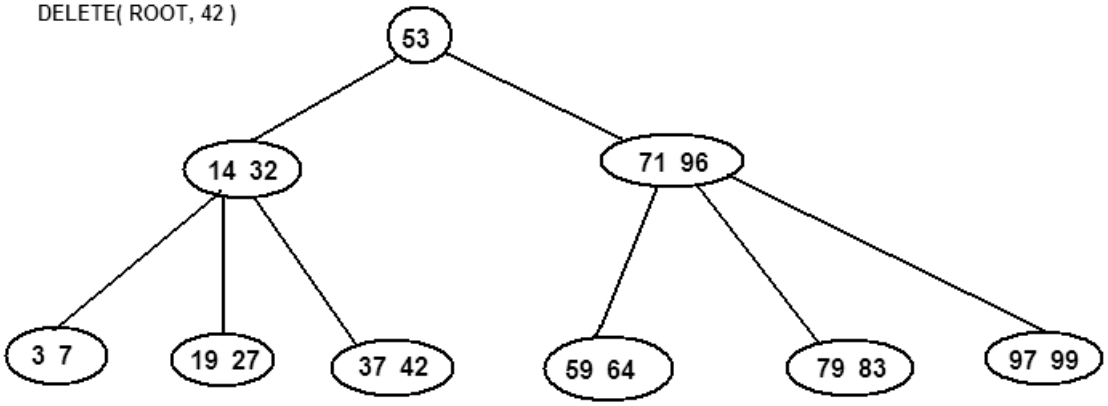


Fig 2.15: Case2 of Deletion

Case 3: In the previous case, there was an adjacent node with extra elements and hence the adjustment was made easily. But if all the nodes have exactly the minimum required, and if now a value is deleted from a node in this, then no value can be borrowed from any of the adjacent nodes. Hence as before a value from the parent is pushed into the node (in this case 32 is pushed down). Then the nodes are merged together. But we see that the parent node has insufficient number of values. Hence same process of merging takes place recursively till the entire tree is adjusted.



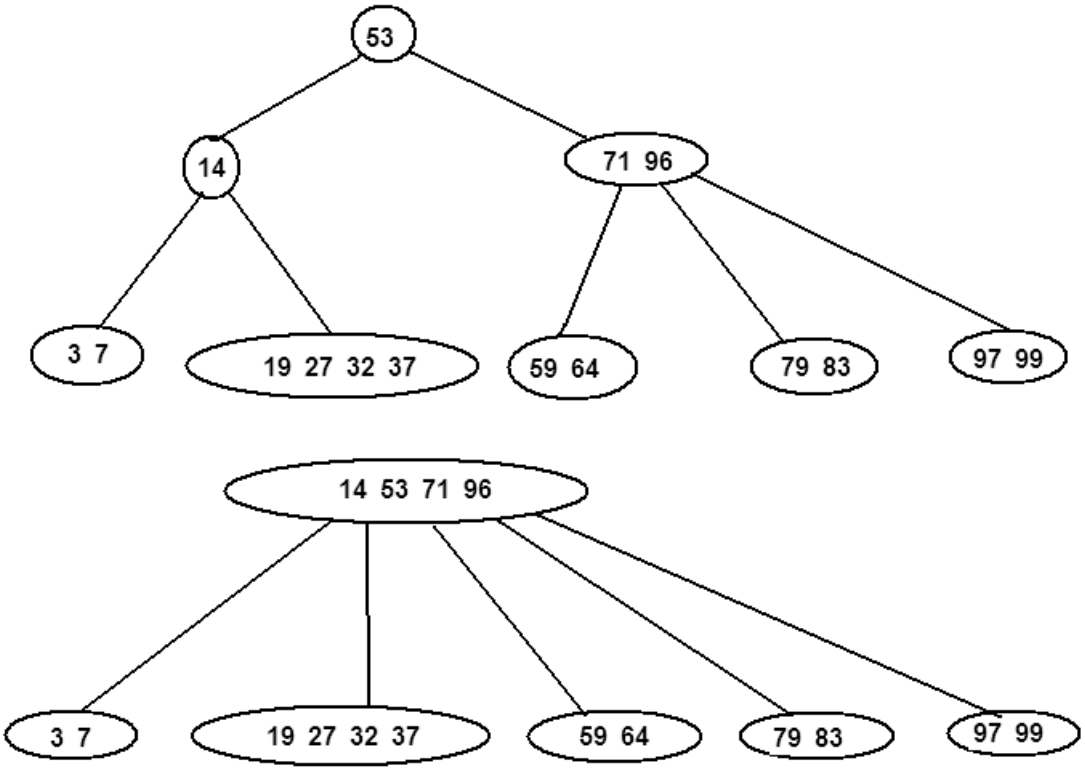


Fig 2.16: Case3 of Deletion

DELETE(ROOT, K)

```
Temp = SEARCH( ROOT, k ), DELETED = 0, i = 1
While i < = count(temp)
        If (k = info(temp[i])
                DELETED = 1
                Delete temp[i]
        End if
End while
If DELETED = 0
        Print "Item not found"
        Return
End if
If count(temp) < n / 2
        i = 1
        While i <= count(par)
                If count(child[i](par)) > n/2
                        s = child[i](par)
                        break
                Else
                        i = i + 1
                End if
        End while
        If info(temp[1]) > info(s[count(s)])
                Ins(temp, info(par[1]))
                Ins(par, info(s[count(s)]))
        Else
```

```
                Ins(temp, info(par[count(par)]))
                Ins(par, info(s[1]))
            End if
        End if
End DELETE
```

Splay trees

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) time.

Tree Splaying

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.

Advantages:

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly.

Tree rotations

To bring the recently accessed node closer to the tree *root*, a splay tree uses tree rotations. There are six types of tree rotations, three of which are symmetric to the other three. These are as follows:

- Left and right rotations
- Zig-zig left-left and zig-zig right-right rotations
- Zig-zag left-right and zig-zag right-left rotations

The first type of rotations, either left or right, is always a *terminal rotation*. In other words, the splaying is complete when we finish a left or right rotation.

Deciding which rotation to use

The decision to choose one of the above rotations depends on three things:

- Does the node we are trying to rotate have a grand-parent?
- Is the node left or right child of the parent?
- Is the parent left or right child of the grand-parent?

If the node does not have a grand-parent, we carry out a left rotation if it is the right child of the parent; otherwise, we carry out a right rotation.

If the node has a grand-parent, we have four cases to choose from:

If node is left of parent and parent is left of grand-parent, we do a *zig-zig rightright* rotation.

If node is left of parent but parent is right of grand-parent, we do a *zig-zag rightleft* rotation.

If node is right of parent and parent is right of grand-parent, we do a *zig-zig leftleft* rotation.

Finally, if node is right of parent but parent is left or grand-parent, we do a *zig-zag leftright* rotation.

The actual rotations are described in the following sections.

Left and right rotations

The following shows the intermediate steps in understanding a right rotation. The left rotation is symmetric to this.

As we can see, each of the left or right rotations requires **five** pointer updates:

```
if (current == parent->left) {
/* right rotate */
parent->left = current->right;
if (current->right)
current->right->parent = parent;
parent->parent = current;
current->right = parent;
} else {
/* left rotate */
parent->right = current->left;
if (current->left)
current->left->parent = parent;
parent->parent = current;
current->left = parent;
}
current->parent = 0;
```

Zig-zig right-right and left-left rotations

The following shows the intermediate steps in understanding a zig-zig right-right rotation. The zig-zig left-left rotation is symmetric to this. Note in the following that with zig-zig rotations, we first do a right or left rotation on the **parent**, before doing a right or left rotation on the **node**.

As we can see, zig-zig right-right rotation requires **nine** pointer updates.

```
/* zig-zig right-right rotations */
if (current->right)
current->right->parent = parent;
if (parent->right)
parent->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->left = parent->right;
parent->right = grandParent;
parent->left = current->right;
current->right = parent;
```

The same number of pointer updates for zig-zig left-left rotation.

```
/* zig-zig left-left rotations */
if (current->left)
current->left->parent = parent;
if (parent->left)
parent->left->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->right = parent->left;
parent->left = grandParent;
parent->right = current->left;
current->left = parent;
```

Zig-zag left-right and right-left rotations

The following shows the intermediate steps in understanding a zig-zag left-right rotation. The zig-zag right-left rotation is symmetric to this. Note in the following that with zigzag rotations, we do both rotations on the node, in contrast to zig-zig rotations.

As we can see, zig-zag left-right rotation requires **nine** pointer updates.

```
/* zig-zag right-left rotations */
if (current->left)
```



```
current->left->parent = grandParent;
if (current->right)
current->right->parent = parent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->right = current->left;
parent->left = current->right;
current->right = parent;
current->left = grandParent;
```

The same number of pointer updates for zig-zag right-left rotation.

```
/* zig-zag left-right rotations */
if (current->left)
current->left->parent = parent;
if (current->right)
current->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->left = current->right;
parent->right = current->left;
current->left = parent;
current->right = grandParent;
```

Heap Trees

Heap is a special case of balanced binary tree data structure where root-node value is compared with its children and arranged accordingly. Heap trees are of two types- Max heap and Min heap.

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − where the value of root node is less than or equal to either of its children.

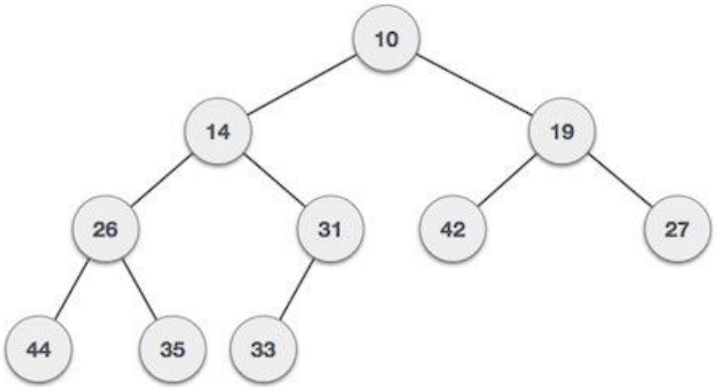


Fig 2.17: Min Heap

Max-Heap − where the value of root node is greater than or equal to either of its children.

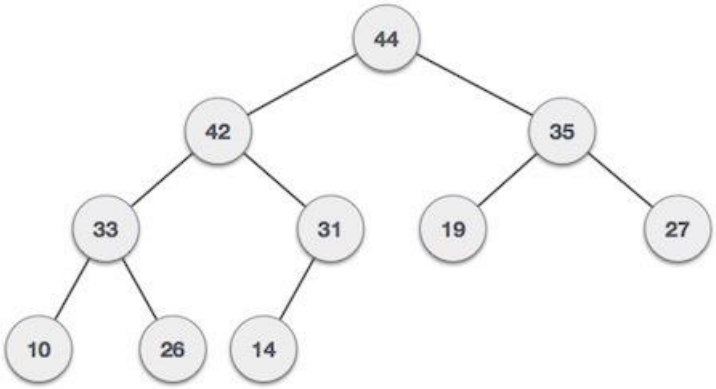


Fig 2.18: Max Heap

If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.** To check whether the right child exists or not, use the condition **R ≤ N.** If true, Right child exists otherwise not. The last node of the tree is **N/2.** After this position tree has only leaves.

Procedure HEAPIFY(A,N)
// A is the list of elements
//N is the number of elements
For (I = N/2 to 1)
WALKDOWN (A,I,N)
END FOR
End Procedure
Procedure WALKDOWN(A, I,N)
//A is the list of unsorted elements
//N is the number of elements in the array
//I is the position of the node where the walkdown procedure is to be applied.
While I ≤ N/2
L=2I, R=2I + 1
If A[L] > A[I] Then
M=L
Else

M=I
End If
If A[R] > A[M] and R ≤ N Then

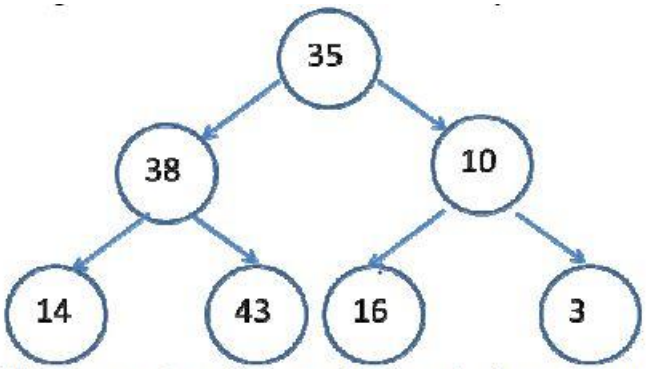
M=R
End If
If M ≠ I Then
A[I] « A[M]
I=M
Else
Return
End If
End While
End WALKDOWN

Example:

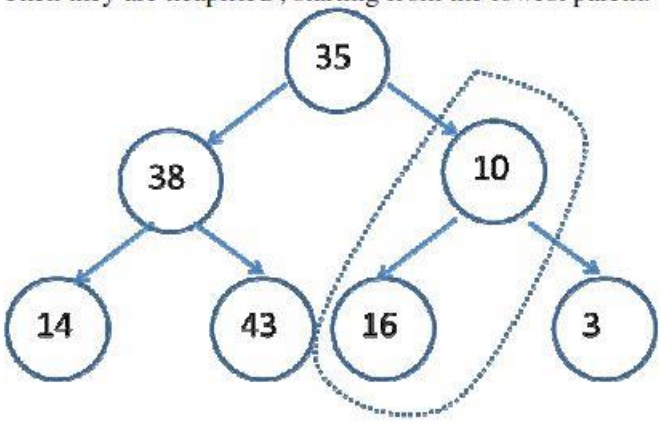
Given a list A with 8 elements:

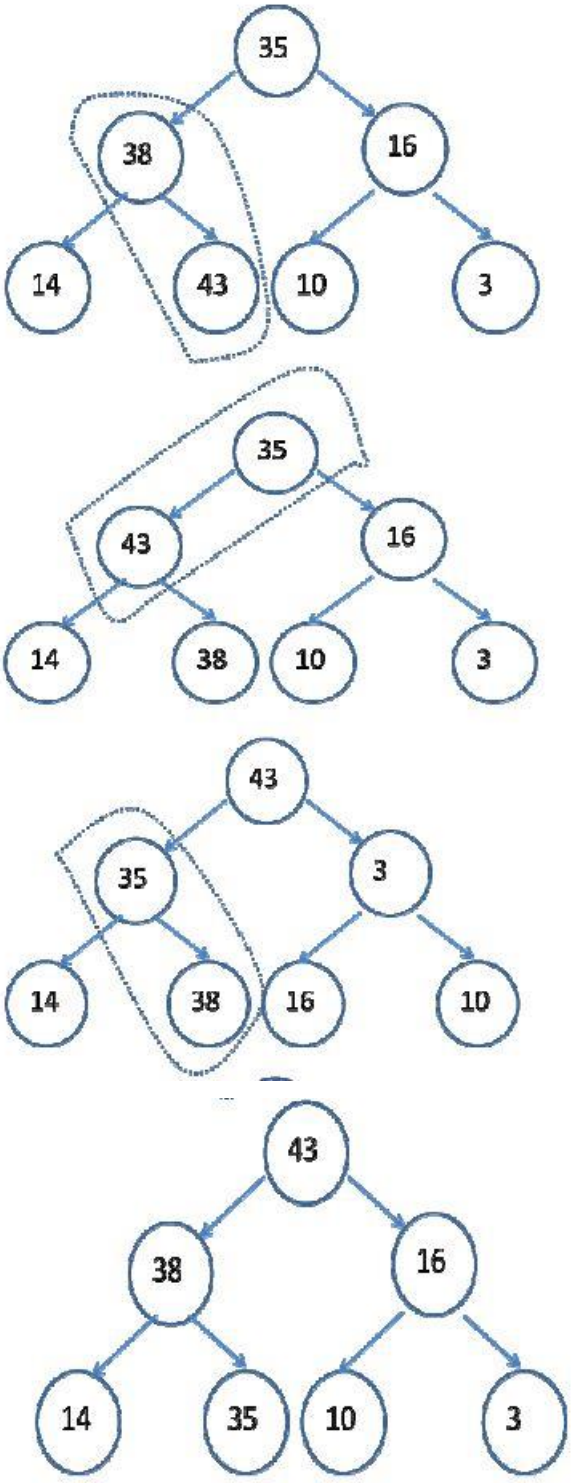
35 38 10 14 43 16 3

The given list is first converted into a binary tree as shown.



Then they are heapified , starting from the lowest parent.





The obtained tree is a Max heap tree.

Fig 2.19: Max Heap Construction

TRIES

-Also called digital tree or radix tree or prefix tree.

-Tries are an excellent data structure for strings.

-Tries is a tree data structure used for storing collections of strings.

-Tries came from the word retrieval.

-Nodes store associative keys (strings) and values.

Tries Structure

Let us consider the case of a tries tree of order 3.Let the key value in this tries is constituted from three letters namely a, b and c. Each node has the following structure:

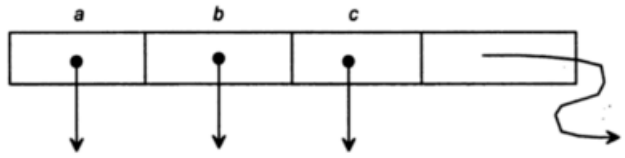


Fig 2.20 : Trie Node Structure

```
// Trie node
struct TrieNode
{
struct TrieNode *children[ALPHABET_SIZE];
// isLeaf is true if the node represents
// end of a word
bool isLeaf;
};
```

Here 3 link fields' points to three nodes in the next level and the last field are called the information field. The information field has the value either TRUE or FALSE. If the value is TRUE then traversing from the root node to this node yields some information. A tries of order 3 is given below:

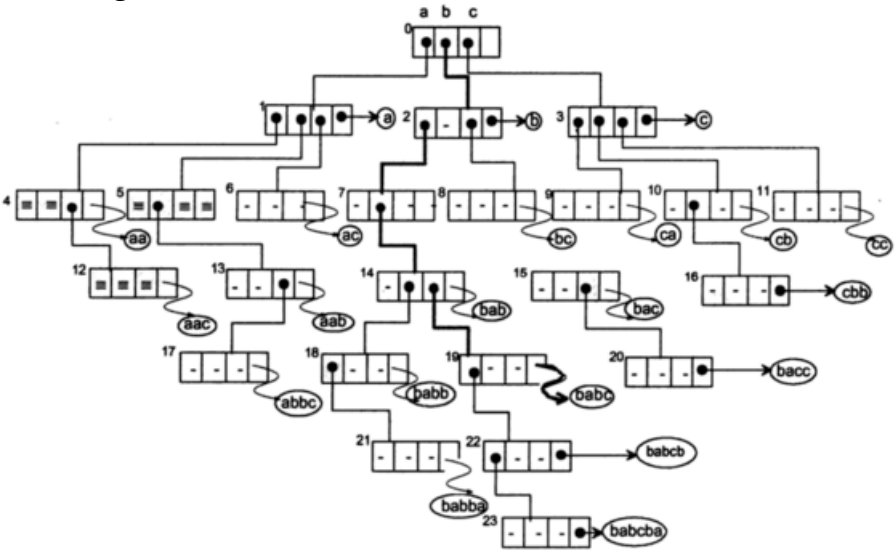


Fig 2.21 :Trie Structure of order 3

For example, let us assume the key value 'bab'. Starting from the root node, and branching based on each letter, traversal will be 0-2-7-14 and in node 14, the information field TRUE implies that 'bab' is a word.

NOTE:

_ Tries indexing is suitable for maintaining variable sized key values.
_ Actual key value is never stored but key values are implied through links.
_ If English alphabets are used, then a trie of order 26 can maintain whole English dictionary.

Operations on Trie

Searching

Searching for a key begins at the root node, compare the characters and move down. The search can terminate due to end of string or lack of key in tries. In the former case, if the *value* field of last node is non-zero then the key exists in tries. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

PSEUDOCODE. The search algorithm involves the following steps:

1. For each character in the string, see if there is a child node with that character as the content.
2. If that character does not exist, return false.
3. If that character exist, repeat step 1.
4. Do the above steps until the end of string is reached.
5. When end of string is reached and if the marker (NotLeaf) of the current Node is set to false, return true, else return false.

Insertion

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the children are an array of pointers to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

PSEUDOCODE: Any insertion would ideally be following the below algorithm:

1. Find the place of the item by following bits.
2. If there is nothing, just insert the item there as a leaf node.
3. If there is something on the leaf node, it becomes a new internal node. Build a new sub tree to that inner node depending how the item to be inserted and the item that was in the leaf node differs.
4. Create new leaf nodes where you store the item that was to be inserted and the item that was originally in the leaf node.

Deletion

Deletion procedure is same as searching and insertion with some modification. To delete a key from a trie, trace down the path corresponding to the key to be deleted, and when we reach the appropriate node, set the TAG field of this node as FALSE. If all the field entries of this node are NULL, then return this node to the pool of free storage. To do so, maintain a stack of PATH to store all the pointers of nodes on the path from the root to the last node reached.

Application of Tries

- Retrieval operation of lexicographic words in a dictionary.
- Word processing packages to support the spelling check.
- Useful for storing a predictive text for auto complete.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF Electrical and Electronics

DEPARTMENT OF Electronics and Communication Engineering

Advanced Data Structures – SCS1201

III. BASIC GRAPH CONCEPTS

Graph: A graph G is a defined as a set of objects called nodes and edges.
G= (V, E) , Where V is a finite and non-empty set at vertices. E is a set of pairs of vertices called edges. Each edge '*e*' in E is identified with a unique pair (*a*, *b*) of nodes in V, denoted by *e* = [*a*, *b*].
Example:

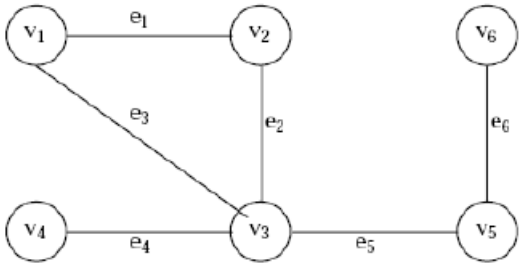


Fig 3.1: Graph

Consider the above graph 'G'. Then the vertex V and edge E can be represented as:
Vertex V = {v1, v2, v3, v4, v5, v6}
E = {e1, e2, e3, e4, e5, e6}
E = {(v1, v2) (v2, v3) (v1, v3) (v3, v4), (v3, v5) (v5, v6)}.
There are six edges and vertex in the graph
Node: A node is a data element of the graph.
Edge: An edge is a path between two nodes.

TYPES OF GRAPH

There are two types of graph. They are
1. Undirected graph
2. Directed graph

Undirected graph: An undirected graph is a graph in which the edges are not directionally oriented towards a node.
Directed graph: A Directed graph or a Digraph is a graph in which the edges are directionally oriented towards any node.

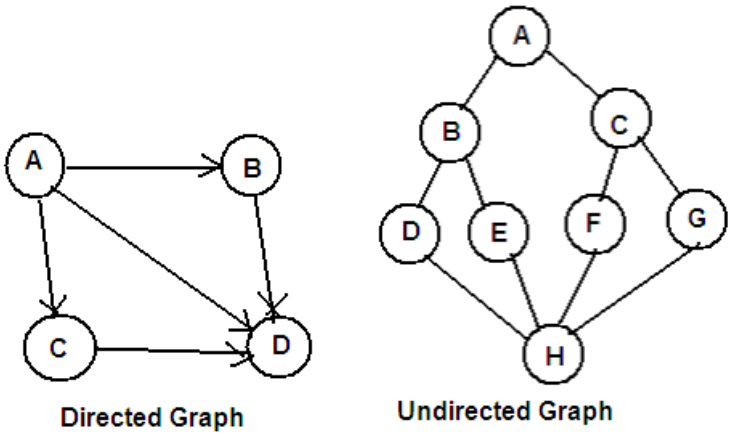


Fig 3.2: Example Graph Structure

Weighted and Unweighted

Graphs can be classified by whether or not their edges have **weights**.

Weighted graph: Edges have a weight

_ Weight typically shows cost of traversing.

_ Example: weights are distances between cities

Unweighted graph: Edges have no weight

_ Edges simply show connections

BASIC TERMINOLOGIES

Arc: The directed edge in a directed graph is called an arc.

Strongly connected graph: A directed graph is called strongly connected if there is a directed path from any vertex to any other vertex.

Example:

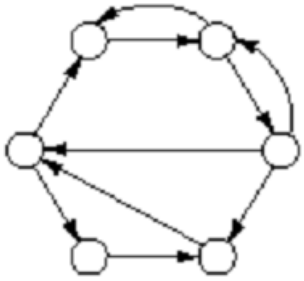


Fig 3.3: Strongly Connected Graph

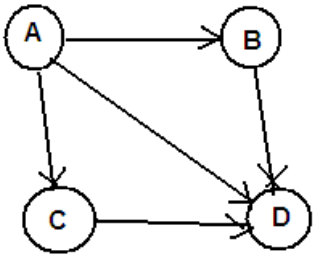
In the above graph we have path from any vertex to any other vertex

Weakly connected graph: A Directed graph is called a weakly connected graph if for any two nodes I and J, there is a directed path from I to J or from J to I.

Out degree: The number of arcs exiting from the node is called out degree of that node.

In degree: The number of arcs entering the node is called in degree of that node.

Example:



Directed Graph

Nodes	Indegree	OutDegree
A	0	3
B	1	1
C	1	1
D	3	0

Source node: A node where the indegree is 0 but has a positive value for outdegree is called a source node. That is there are only outgoing arcs to the node and no incoming arcs to the node.

Example:

Node ‘A’ is the source node.

Sink node: A node where the outdegree is 0 and has a positive value for indegree is called the sink node. That is there is only incoming arcs to the node and no outgoing arcs the node.

Example:

Node ‘D’ is the Sink node.

Cycle: A cycle in a directed graph is a directed path that originates and terminates at the same node ie some number of vertices connected in a closed chain.
Example:

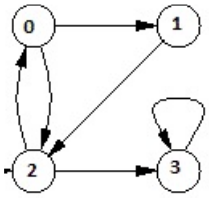


Fig 3.4: Graph

This graph contains three cycles 0->2->0, 0->1->2->0 and 3->3.

REPRESENTATION OF GRAPHS

There are two possible ways by which the graph can be represented.
1. Matrix representation (Array Representation)
2. Linked representation

The graphs can be represented using **Adjacency matrix** or otherwise called the **incidence matrix**. Since the matrix is so sparse it is also called as sparse matrix.

The adjacency matrix is a N X N matrix where N is the number of nodes in the graph. Each entry (I, J) in the matrix has either 1 or 0. An entry 1 indicates that there is a direct connection from I to J. An entry 0 indicates that there is no direct connection from I to J. If an adjacency matrix is written for the above directed graph as shown:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation

Since the matrix is so sparse in nature the second method of representation will be preferred if the number of edges is very less compared to the number of vertices.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

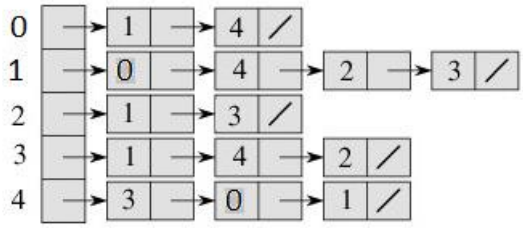


Fig 3.5: Adjacency List Representation

GRAPH TRAVERSALS

There are two methods for traversing through the nodes of the graph. They are:
(1) Breadth First Search Traversal (BFS)
(2) Depth First Search Traversal (DFS)

Breadth First Search Traversal (BFS)

As the name implies, this method traverses the nodes of the graph by searching through the nodes breadth-wise. Initially let the first node of the graph be visited. This node is now considered as node u. Now find out all the nodes which are adjacent to this node. Let all the adjacent nodes be called as w. Add the node u to a queue. Now every time an adjacent node w is visited, it is added to the queue. One by one all the adjacent nodes w are visited and added to the queue. When all the unvisited adjacent nodes are visited, then the node u is deleted from the queue and hence the next element in the queue now becomes the new node u. The process is repeated on this new node u. This is continued till all the nodes are visited.
The Breadth First Traversal (BFT) algorithm calls the BFS algorithm on all the nodes.

Algorithm

BFT(G, n)

```
Repeat for i = 1 to n
Visited[i] = 0
End Repeat
Repeat for i = 1 to n
If visited[i] = 0
BFS(i)
End if
End Repeat
```

BFS(v)

```
u = v
visited[v] = 1
Repeat while(true)
Repeat for all vertices w adjacent to u
If visited[w] = 0
Add w to queue
Visited[w] = 1
End if
End Repeat
If queue is empty
Return
End if
Delete u from queue
End while
End BFS
```

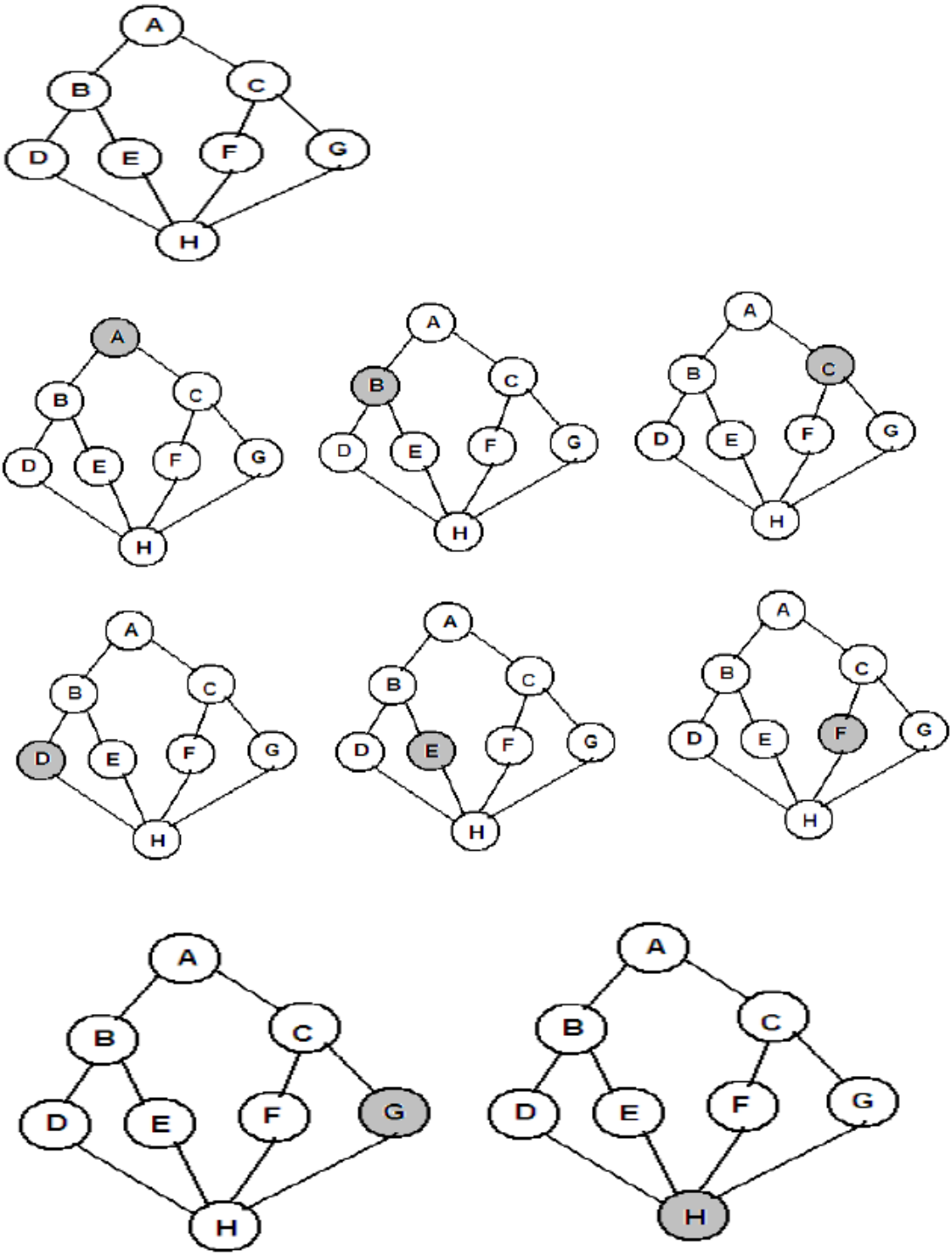


Fig 3.5 BFS Traversal

Depth First Search Traversal (DFS)

In the Depth First Search Traversal, as the name implies the nodes of the graph are traversed by searching through all the nodes by first going to the depth of the graph. The first node is visited first. Let this be node u. Find out all the adjacent nodes of u. Let that be w. Apply the DFS on the first adjacent node recursively. Since a recursive approach is followed, the nodes are traversed by going to the depth of the graph first. The DFT algorithm calls the DFS algorithm repeatedly for all the nodes in the graph.

Algorithm
DFT(G, n)
Repeat for i = 1 to n
Visited[i] = 0
End Repeat
Repeat for i = 1 to n
If visited[i] = 0
DFS(i)
End if
End Repeat

DFS(v)
Visited[v] = 1
Repeat for each vertex w adjacent from v
If visited[w] = 0
DFS(w)
End if
End for

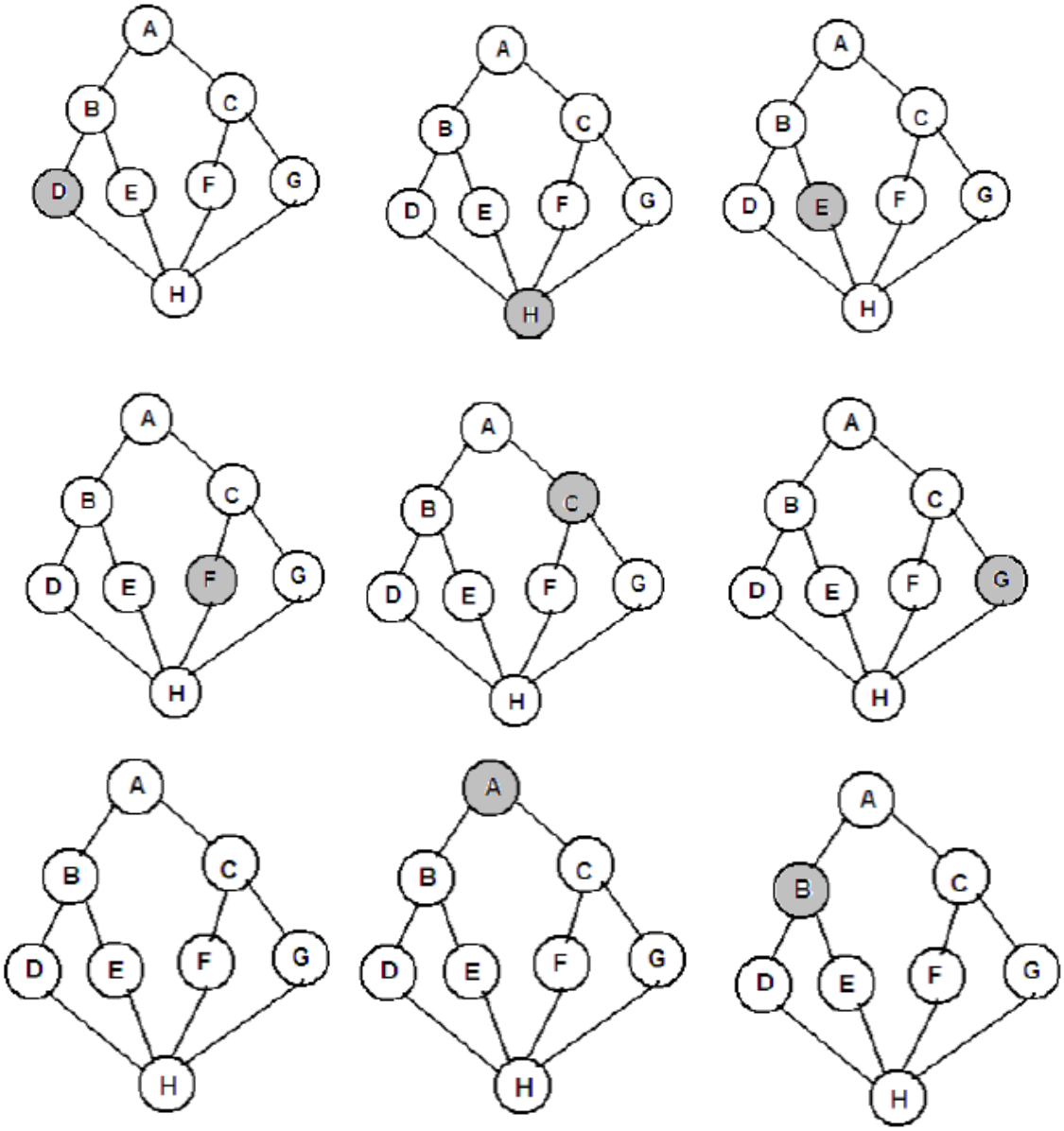


Fig 3.6: DFS Traversal

Applications of depth First Traversal

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph : A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) Path Finding:

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

4) Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers .
5**) Finding** Strongly Connected Components **of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
6) Solving puzzles with only one solution, such as mazes. DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.

Applications of Breadth First Traversal
1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.
6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
10) To test if a graph is Bipartite: We can either use Breadth First or Depth First Traversal.
11) Path Finding: We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node. Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure similar to Breadth First Search.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

Advanced Data Structures – SCS1201

IV. ADVANCED GRAPH CONCEPTS

MINIMUM SPANNING TREES

Weight of an edge: Weight of an edge is just of the value of the edge or the cost of the edge. For example, a graph representing cities, has the distance between two cites as the edge cost or its weight.
Network: A graph with weighted edges is called a network.
Spanning Tree: Any tree consisting of edges in the graph G and including all vertices in G is called a spanning tree. Given a network, we should try to connect all the nodes in the nodes in the graph with minimum number of edges, such that the total weight is minimized. To solve this problem, we shall devise an algorithm that converts a network into to tree structures called the minimum spanning tree of the network.
Given a network, the edges for the minimum spanning tree are chosen in such a way that:
(1) Every node in the network must be included in the spanning tree.
(2) The overall edge weight of the spanning tree is the minimum possible that will allow the existence of a path between any 2 nodes in the tree.
The two algorithms which are used for finding the minimum spanning tree for a graph are:
1. Kruskal's Algorithm
2. Prim's Algorithm
3. Sollin's Algorithm

KRUSKAL'S ALGORITHM

The Kruskal's algorithm follows greedy approach. At every stage of the solution, it takes that edge which has the minimum cost and builds the minimum spanning tree.
Example:

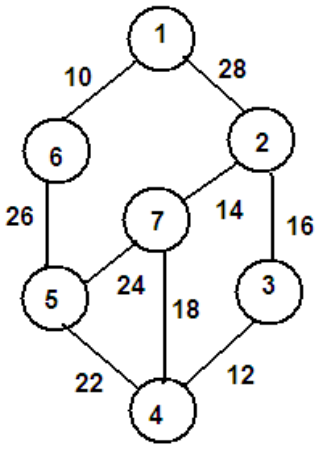


Fig 4.1: Graph

Consider the above graph. Now let us apply the Kruskal's algorithm to construct a minimum spanning tree.

Step 1:
Construct a queue with the cost of edges, such that the edges are placed in the queue in the ascending order of the cost as shown.
Queue of edge costs

10	12	14	16	18	22	24	26	28

Step 2:

Create N sets each consisting one node. N is the number of nodes in the graph. Then for the above problem, the sets which will be created are

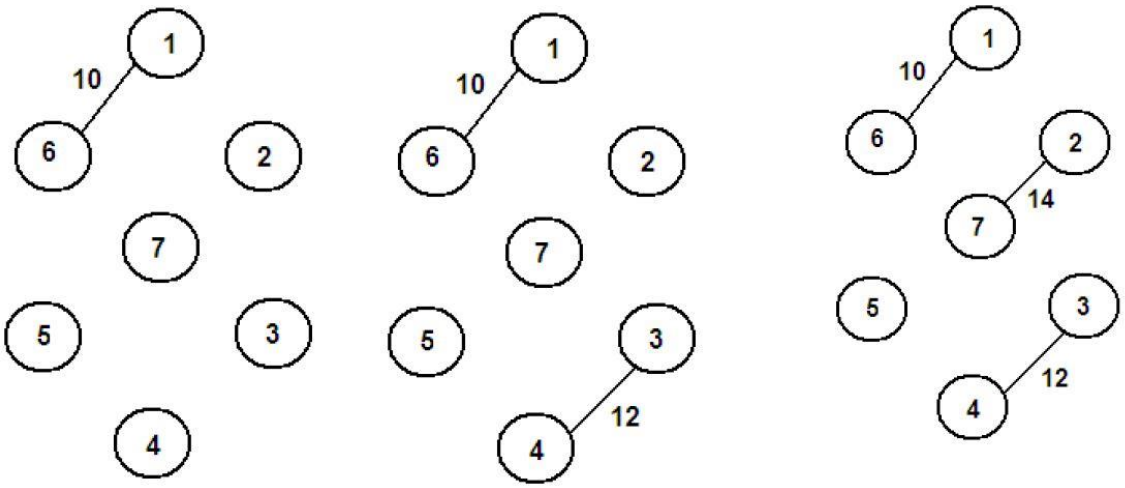
S1 = {1}
S2 = {2}
S3 = {3}
S4 = {4}
S5 = {5}
S6 = {6}
S7 = {7}

Step 3:

Delete a cost from the queue. Let the nodes associated with that edge be (u,v). Now, 10 is deleted first from the queue. The nodes associated with 10 is (u,v) = (1,6). Check if u and v belong to the same set or different set. If they belong to the different set then enter that into the output matrix as shown. Since 1 belongs to S1 and 6 belong to S6, they can be entered into the T matrix. If the nodes belong to the same set, then entering them into the matrix will give an output which may form a cycle. Hence that is avoided. The T matrix has n-1 rows and 2 columns.

Step 4:

Using the edges in the T matrix connect the nodes of the graph. The resulting tree is the required minimum spanning tree.



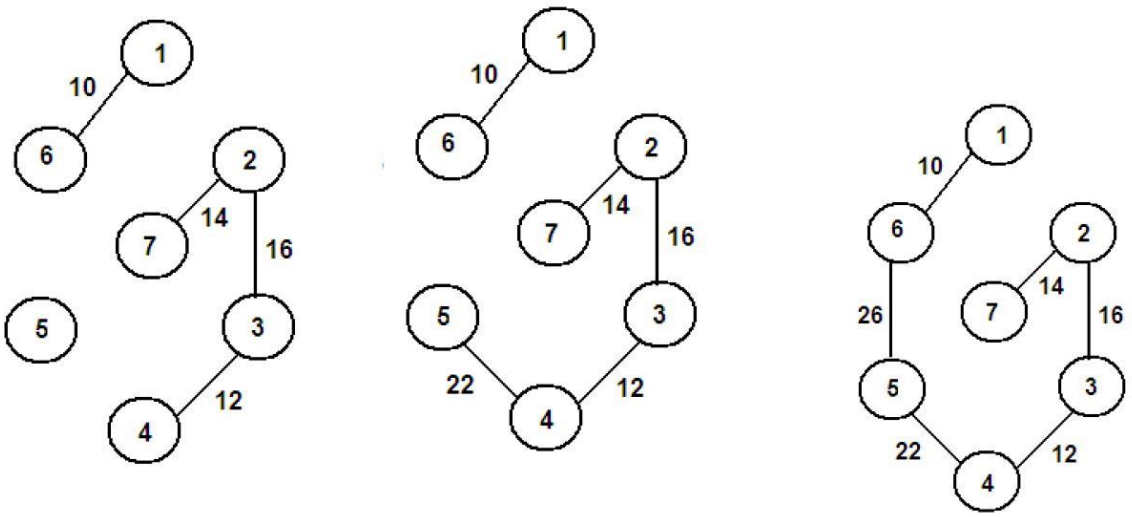


Fig 4.2: Steps for Kruskal's Algorithm

Algorithm

KRUSKAL(E, cost, n, t)
Construct a queue with edge costs such that they are in ascending order
i = 0, mincost = 0
while i < n – 1 and queue is not empty
    Delete minimum cost edge (u, v) from queue
    j = Find(u), k = Find(v)
    If j ≠ k
        i = i + 1
        t[i, 1] = u, t[i, 2] = v
        mincost = mincost + cost[u, v]
        Union(j, k)
    End if
End while
If i ≠ n – 1
    Print "No spanning tree"
Else
    Return mincost
End if
End KRUSKAL

PRIM'S ALGORITHM

The other popular algorithm used for constructing the minimum spanning tree is the Prim's algorithm, which also follows the greedy approach. We can consider the same example as above and solve it using Prim's algorithm.

Example:

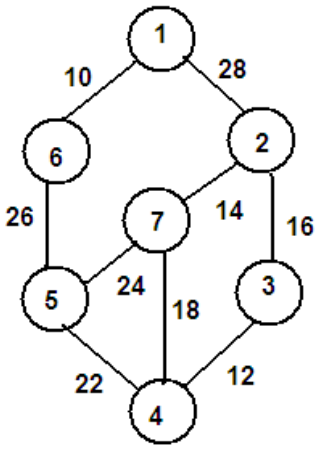


Fig 4.3: Graph

The Cost Matrix is,

1	2	3	4	5	6	7
0	28	∞	∞	∞	10	∞
28	0	16	∞	∞	∞	14
∞	16	0	12	∞	∞	∞
∞	∞	12	0	22	∞	18
∞	∞	∞	22	0	26	24
10	26	∞	∞	∞	0	∞
∞	14	∞	18	24	∞	0

Step 1:

Select the least cost edge from the graph and enter into the T matrix. The least cost edge is (1, 6) with cost 10.

T matrix

	u	v
1	1	6
2		
3		
4		
5		
6		

Let us consider an array NEAR[ ], which is filled as follows:

If cost[i, l] < cost[i, k]

Near[i] = l

Else

Near[i] = k

In the first iteration i = 1 and (k, l) = (1, 6). Using the above condition the NEAR array is filled as follows.

NEAR

1	0	
2	1	28
3	1	∞
4	1	∞
5	6	26
6	0	
7	1	∞

Among the costs, 26 is minimum. Hence (5, 6) is entered into the T matrix. The corresponding entry into the NEAR array is made 0.

T matrix

	u	v
1	1	6
2	5	6
3		
4		
5		
6		

Step 3:
Now in every iteration the NEAR array is updated using the following condition and procedure in step 2 is followed to fill up the T matrix. The solution is as follows:

If Near[k] ≠ 0 and cost[k, Near[k]] > cost[k, j]
    Near[k] = j

Updated NEAR

1	0	
2	1	28
3	1	∞
4	5	22
J=5	0	
6	0	
7	5	24

Among the cost computed, 22 is minimum and hence (4,5) is selected as the minimum edge.
T matrix

	u	*v*
1	1	6
2	5	6
3	4	5
4		
5		
6		

Updated NEAR

1	0	
2	1	28
3	4	12
J=4	0	
5	0	
6	0	
7	4	18

Among the cost computed, 12 is minimum and hence (3, 4) is selected as the minimum edge.

T matrix

	u	v
1	1	6
2	5	6
3	4	5
4	3	4
5		
6		

Updated NEAR

1	0	
2	3	16
J=3	0	
4	0	
5	0	
6	0	
7	4	18

Among the cost computed, 16 is minimum and hence (2, 3) is selected as the minimum edge.
T matrix

	u	v
1	1	6
2	5	6
3	4	5
4	3	4
5	2	3
6		

Updated NEAR

1	0	
J=2	0	
3	0	
4	0	
5	0	
6	0	
7	2	14

The last edge (7, 2) is selected and entered into the T matrix.
T matrix

	u	v
1	1	6
2	5	6
3	4	5
4	3	4
5	2	3
6	7	2

Step 4:

Now using the edges in the T matrix connect the nodes in the graph. The resulting tree is the minimum spanning tree.

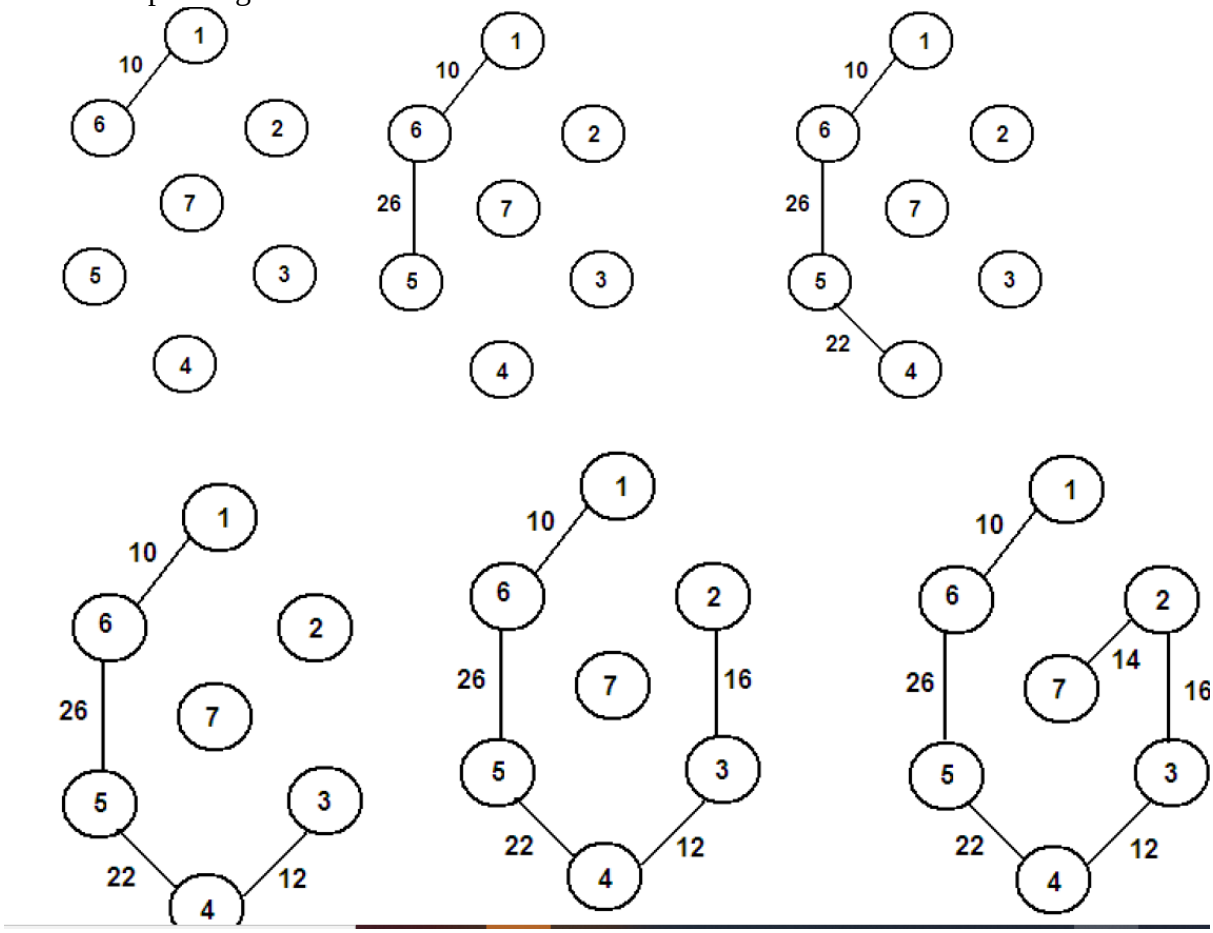


Fig 4.4: Steps of Prim's Algorithm

Algorithm

PRIM(E, cost, n, t)

Let (k, L) be an edge of minimum cost in E
mincost = cost[k, L]
t[1, 1] = k, t[1, 2] =L
For i = 1 to n
    If cost[i, L] < cost[i, k]
        Near[i] = L
    Else
        Near[i] = k
    End if
End for
Near[k] = Near[L] = 0

```
For i = 2 to n -1
        Let j be an index such that near[j] ≠ 0 and cost[j, near[j]] is minimum
        T[i, 1] = j, t[i, 2] = Near[j]
        mincost = mincost + cost[j, near[j]]
        Near[j] = 0
        For k = 1 to n
                If Near[k] ≠ 0 and cost[k, Near[k]] > cost[k, j]
                        Near[k] = j
                End if
        End for
End For
Return mincost
End PRIM
```

SOLLIN'S ALGORITHM

A minimum spanning tree (MST) of a weighted graph G is a spanning tree of G whose edges sum to minimum weight. In other words, a minimum spanning tree is a tree formed from a subset of the edges in a given undirected graph, with two properties: (1) it spans the graph, i.e., it includes every vertex in the graph, and (2) it is a minimum, i.e., the total weight of all the edges is as low as possible.

Sollin's algorithm selects several edges at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in this forest. The edge is a minimum-cost edge that has exactly one vertex in the tree. These selected edges are added to the spanning tree being constructed. Note that it is possible for two trees in the forest to select the same edge. So, multiple copies of the same edge are to be eliminated. Also, when the graph has several edges with the same cost, it is possible for two trees to select two different edges that connect them together. At the start of the first stage, the set of selected edges is empty. The algorithm terminates when there is only one tree at the end of a stage or when no edges remain to be selected.

The Sollin's Algorithm based on two basic operations:

_ Nearest Neighbor – This operation takes a an input a tree spanning the nodes Nk and determines an arc (ik , jk) with the minimum cost among all arcs emanating from Nk.

_ Merge (ik jk) – This operation takes as an input two nodes ik and jk,and if the two nodes belong to two different trees, then merge these two trees into a single tree

Algorithm

Sollin's Algorithm

{

Form a forest consisting of the nodes of the graph while the forest has more than one tree

For each tree in the forest

Choose the cheapest edge

Form a vertex in the tree to a vertex not in the tree

Merge trees with common vertices

}

This algorithm keeps a forest of minimum spanning trees which it continuously connects via the least cost arc live in each tree. To begin each node is made its minimum spanning tree from here the shortest path live in each tree (which doesn't connect to a node already belonging to the current tree) is added along the minimum spanning tree it connect to . This continues until exit a single spanning tree.

Example:

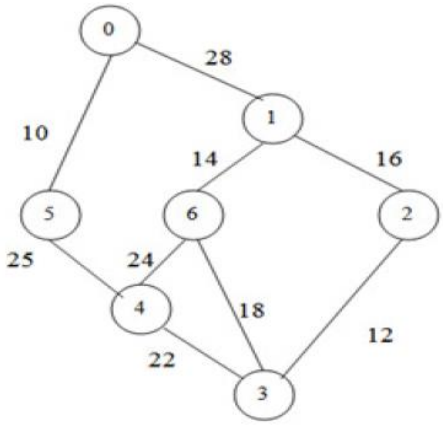


Figure (a)

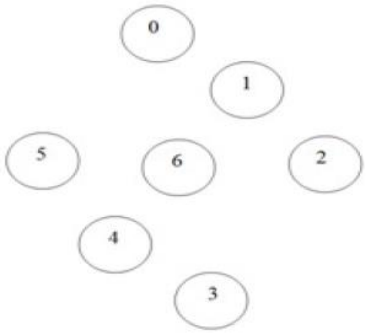
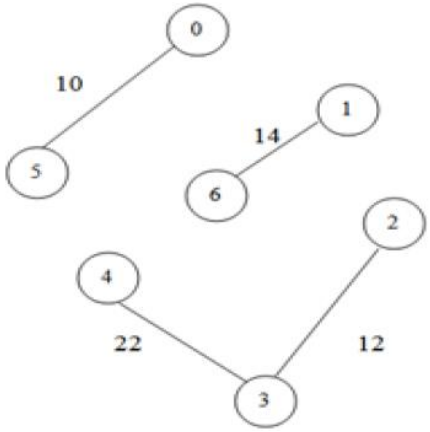


Figure (b)



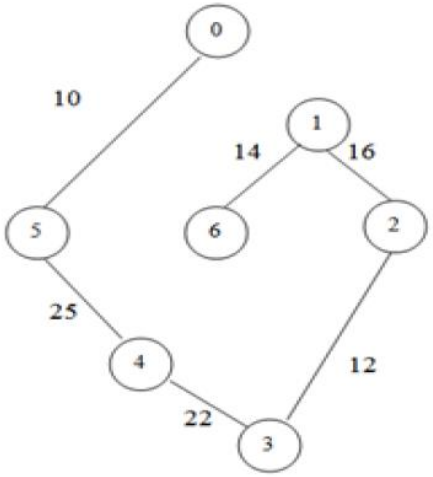


Fig 4.5: Sollin's Algorithm

Figures shows the stages in Sollin's algorithm when it begins with the graph of fig (a). The initial configuration of zero selected edges is the same as that shows in fig(b) .Each tree in this spanning forest is a single vertex.

The edges selected by vertices 0,1,…,6 are , respectively , (0,5),(1,6),(2,3),(3,2),(4,3),(5,0), and (6,1). The distinct edges in this selection are (0,5),(1,6),(2,3), and (4,3). Adding these to the set of selected edges results in the configuration of fig(c). In the next stage, the tree with vertex set {0,5} selects the edge (5,4),and the remaining two trees select the edge (1,2). Following the addition of these two edges to the set of selected edges, construction of the spanning tree is complete. The resulting spanning tree is shown in fig.

SINGLE SOURCE SHORTEST PATH ALGORITHM

DJIKSTRA'S ALGORITHM

The Djikstra's algorithm finds out the shortest path between the single source and every other node in the graph. For example consider the following graph. Let the node 5 be the source. Let solve this using Djiksta's algorithm to find the shortest paths between 5 and every other node in the graph.

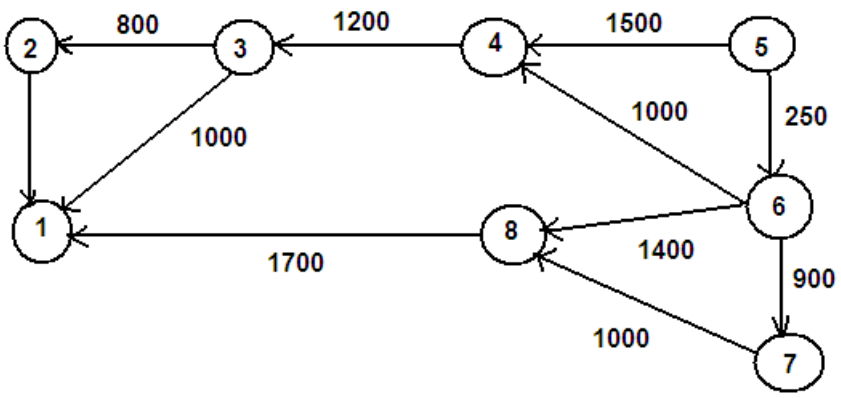


Fig 4.6: Graph

A Distance array Dist[] is initially filled with infinity. The entry corresponding to the source node 5 alone is made 0. Now find out the adjacent nodes of 5 and update their values using the cost matrix. In the next iteration the node with minimum distance is the vertex selected and again the above process is repeated. The Column S shows the set of vertices already selected. In every iteration the node with minimum distance and which is not yet selected is taken as the new vertex.

The solution is obtained as follows.

Iteration	S	Vertex Selected	Dist[ ]							
			1	2	3	4	5	6	7	8
Initial	-	-	∞	∞	∞	1500	0	250	∞	∞
	5	6	∞	∞	∞	1250	0	250	1150	1650
	5,6	7	∞	∞	∞	1250	0	250	1150	1650
	5,6,7	4	∞	∞	2450	1250	0	250	1150	1650
	5,6,7,4	8	3350	∞	2450	1250	0	250	1150	1650
	5,6,7,4,8	3	3350	3250	2450	1250	0	250	1150	1650
	5,6,7,4,8,3	2	3350	3250	2450	1250	0	250	1150	1650

Now we can see that the values in the last row, gives the distance of the shortest path between the source node 5 and every other node in the graph.

The shortest paths corresponding to this can also be generated. The paths are represented using linked lists. Whenever a value in the distance array is updated, the shortest path linked lists are also adjusted.

The shortest paths are represented using linked lists as shown.

Algorithm

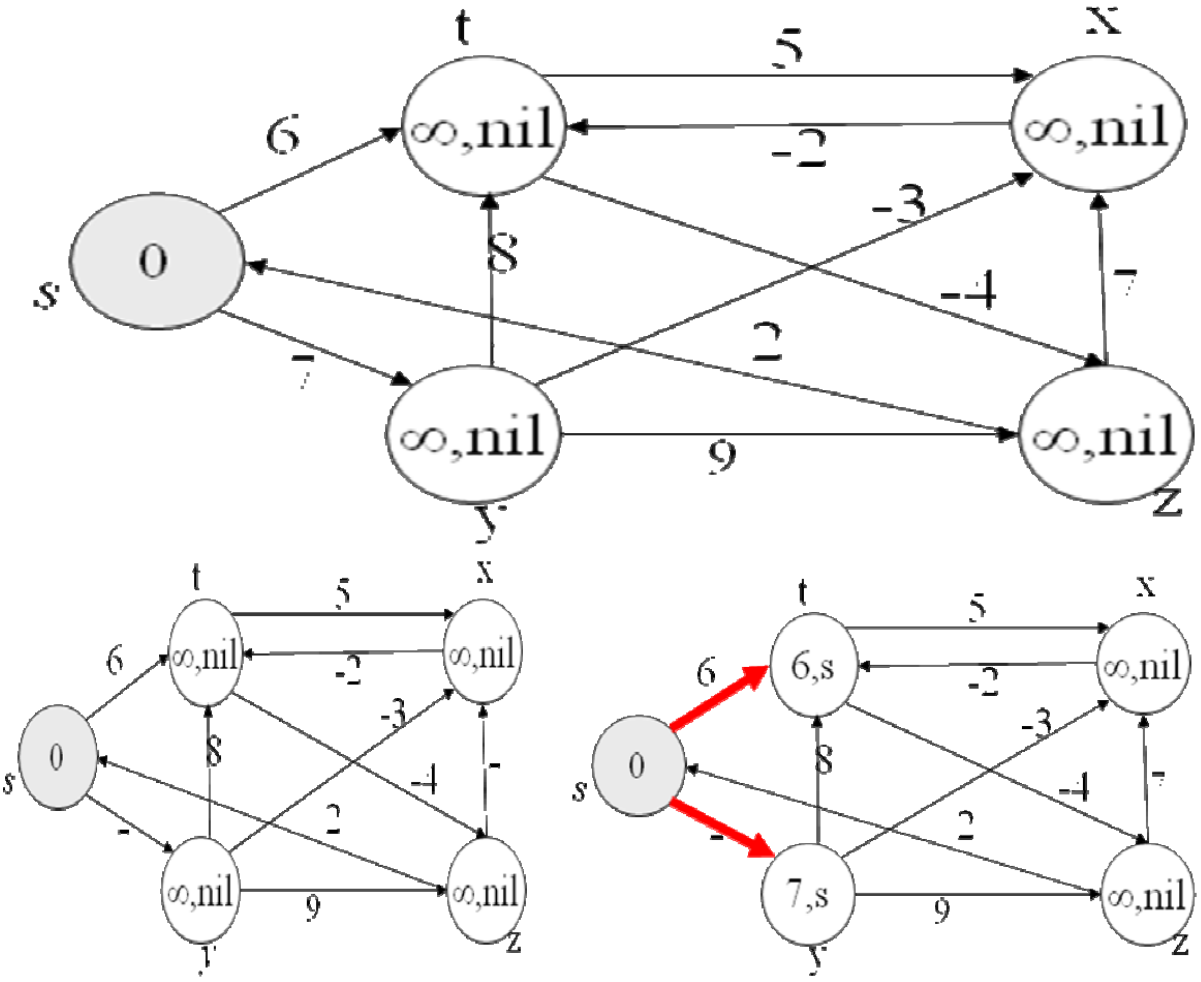
DJIKSTRA(v, cost, dist, n)

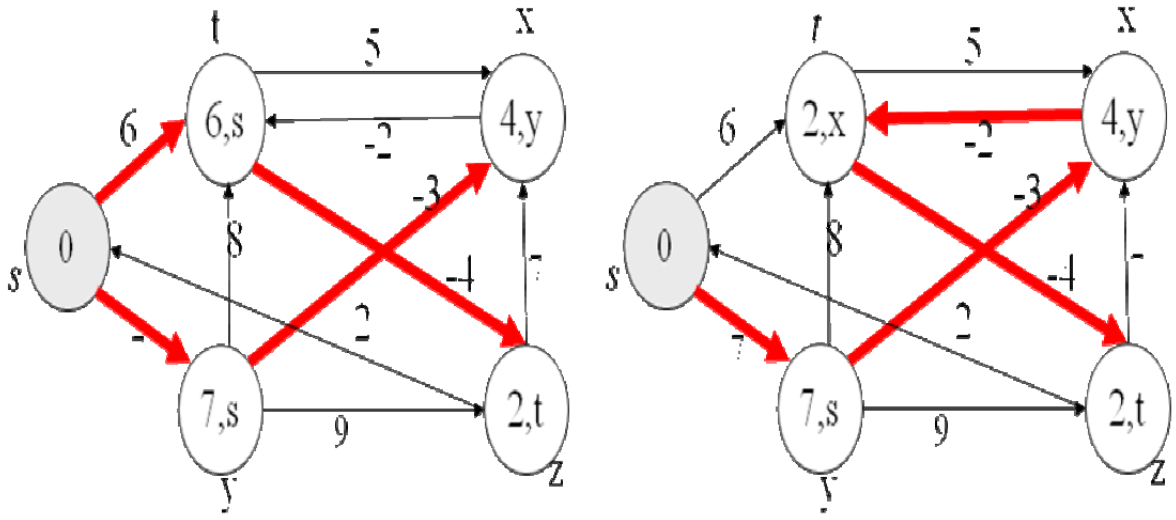
For i = 1 to n
S[i] = false
Dist[i] = ∞
End for
S[v] = true
Dist[v] = 0
Create n lists each beginning with v
For num = 1 to n – 1
Choose u from among those vertices not in S such that dis[u] is minimum
S[u] = true

For each w adjacent to u with s[w] = false
If Dist[w] > Dist[u] + cost[u, w]
Dist[w] = Dist[u] + cost [u, w]
List[w] = List[u] + w
End if
End for
End for

THE BELLMAN-FORD SHORTEST PATH ALGORITHM

Given a weighted graph G and a source vertex s, Bellman-Ford algorithm finds the shortest (minimum cost) path from s to every other vertex in G. The weighted path length (cost) is the sum of the weights of all links on the path.





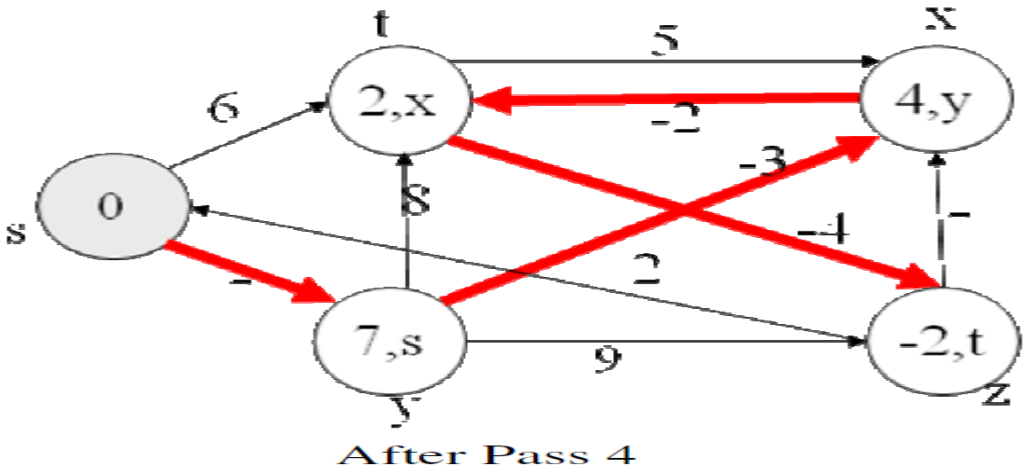


Fig 4.7: Bellman-Ford Algorithm

The order of edges examined in each pass:
(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)

Algorithm

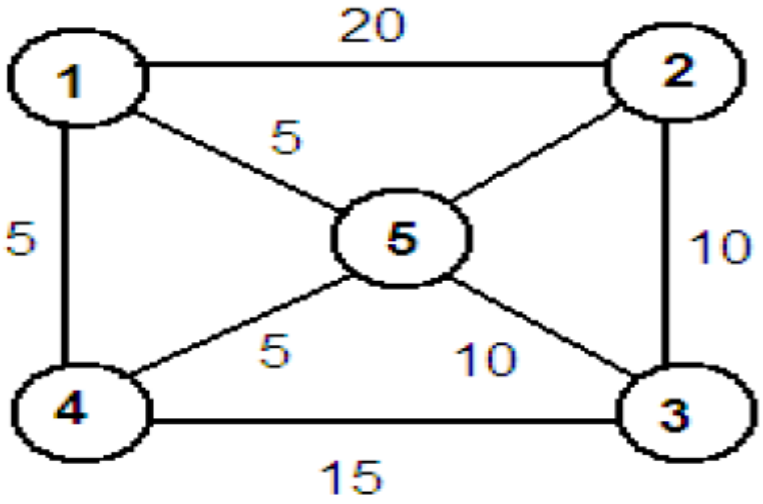
Bellman-Ford(G, w, s)

1. Initialize-Single-Source(G, s)
2. for i := 1 to |V| - 1 do
3. for each edge (u, v) Î E do
4. Relax(u, v, w)
5. for each vertex v Î u.adj do
6. if d[v] > d[u] + w(u, v)
7. then return False // there is a negative cycle
8. return True

Relax(u, v, w)
if d[v] > d[u] + w(u, v)
then d[v] := d[u] + w(u, v)
parent[v] := u

FLOYD WARSHALL ALGORITHM

The All pairs shortest path algorithm is used to find the shortest path between every pair of nodes in the graph. Consider the following example.
We will use the following condition solve the problem
A[i, j] = min(A[i, j], A[i, k] + A[k, j])
Solution is derived as follows using the above condition.



Cost Matrix

	1	*2*	*3*	*4*	*5*
1	0	20	∞	5	5
2	20	0	10	∞	10
3	∞	10	0	15	10
4	5	∞	15	0	5
5	5	10	10	5	0

Iteration1

	1	2	3	4	5
1	0	20	∞	5	5
2	20	0	10	∞	10
3	∞	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration2

	1	2	3	4	5
1	0	20	30	5	5
2	20	0	10	∞	10
3	30	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration3

	1	2	3	4	5
1	0	20	30	5	5
2	20	0	10	25	10
3	30	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration5

	1	2	3	4	5
1	0	20	20	5	5
2	20	0	10	25	10
3	20	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

The output matrix gives the shortest path distance between all pairs of nodes.

Algorithm

ALLPAIRSHORTESTPATH(cost, A, n)

```
For i = 1 to n
For j = 1 to n
A[i, j] = cost[i, j]
End for
For k = 1 to n
For i = 1 to n
For j = 1 to n
A[i, j] = min(A[i, j], A[i, k] + A[k, j])
End for
End for
End for
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

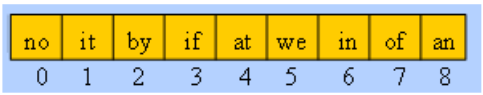
SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

Advanced Data Structures – SCS1201

V. TABLE DATA STRUCTURES

Table is a data structure which plays a significant role in information retrieval. A set of n distinct records with keys K1, K2, …., Kn are stored in a file. If we want to find a record with a given key value, K, simply access the index given by its key k.**Network:** A graph with weighted edges is called a network.



The table lookup has a running time of O(1). The searching time required is directly proportional to the number of number of records in the file. This searching time can be reduced, even can be made independent of the number of records, if we use a table called Access Table.

Some of possible kinds of tables are given below:

- Rectangular table
- Jagged table
- Inverted table.
- Hash tables

Rectangular Tables

Tables are very often in rectangular form with rows and columns. Most programming languages accommodate rectangular tables as 2-D arrays. Rectangular tables are also known as matrices. Almost all programming languages provide the implementation procedures for these tables as they are used in many applications.

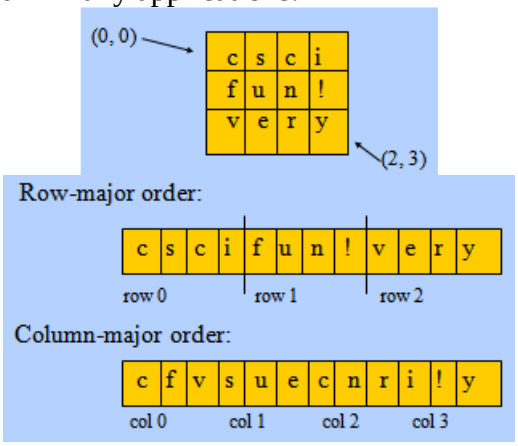


Fig 5.1: Rectangular Tables

Logically, a matrix appears as two-dimensional, but physically it is stored in a linear fashion. In order to map from the logical view to physical structure, an indexing formula is used. The compiler must be able to convert the index (i, j) of a rectangular table to the correct position in the sequential array in memory.

For an m x n array (m rows, n columns):

Each row is indexed from 0 to m-1

Each column is indexed from 0 to n – 1

Item at (i, j) is at sequential position i * n + j

Row major order:

Assume that the base address is the first location of the memory, so the

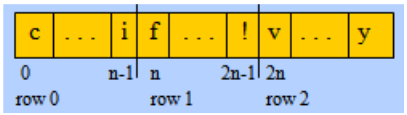
Address aij =storing all the elements in the first(i-1)th rows +

the number of elements in the ith row up to the jth coloumn

= (i-1)*n+j

Column major order:

Address of aij = storing all the elements in the first(j-1)th column +
The number of elements in the jth column up to the ith rows.
=(j-1)*m+i



Jagged table

Jagged tables are nothing but the special kind of sparse matrices such as triangular matrices, band matrices, etc… In the jagged tables, we put a restriction that in the row (or in a column) if elements are present then they are contiguous. Thus, in fig (a) - (e), all are jagged tables except the table in fig (f).

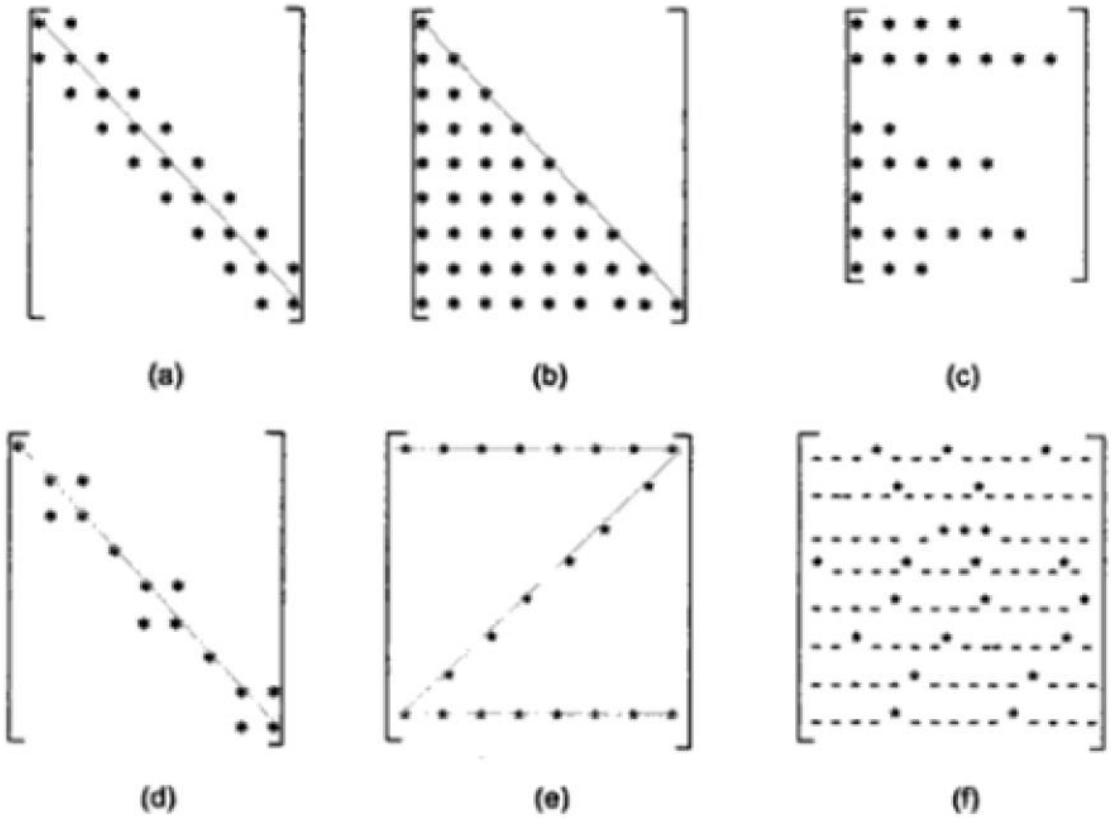


Fig 5.2: Jagged Tables

Symmetric sparse matrices stored as one dimensional arrays can be retrieved using the indexing formula

$$\text{Address}(a_{ij}) = \frac{i \times (i-1)}{2} + j$$

This formula involves multiplication and division which are infact inefficient from the computational point of view. So, the alternative technique is by setting up an access table whose entries correspond to the row indices of the jagged table, such that the ith entry in the access table is

$$\frac{i \times (i-1)}{2}$$

The access table is calculated only at the time of initiation and a=can be stored in memory, it can referred each time the access of element in the jagged table occur. We can also calculate by pure addition rather than multiplication or division such as 0,1, (1+2), (1+2)+3,…

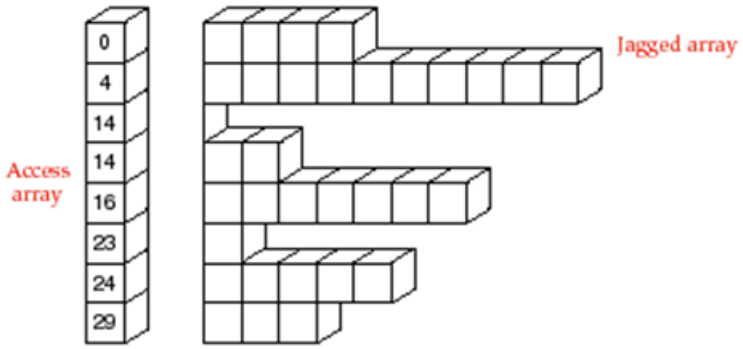
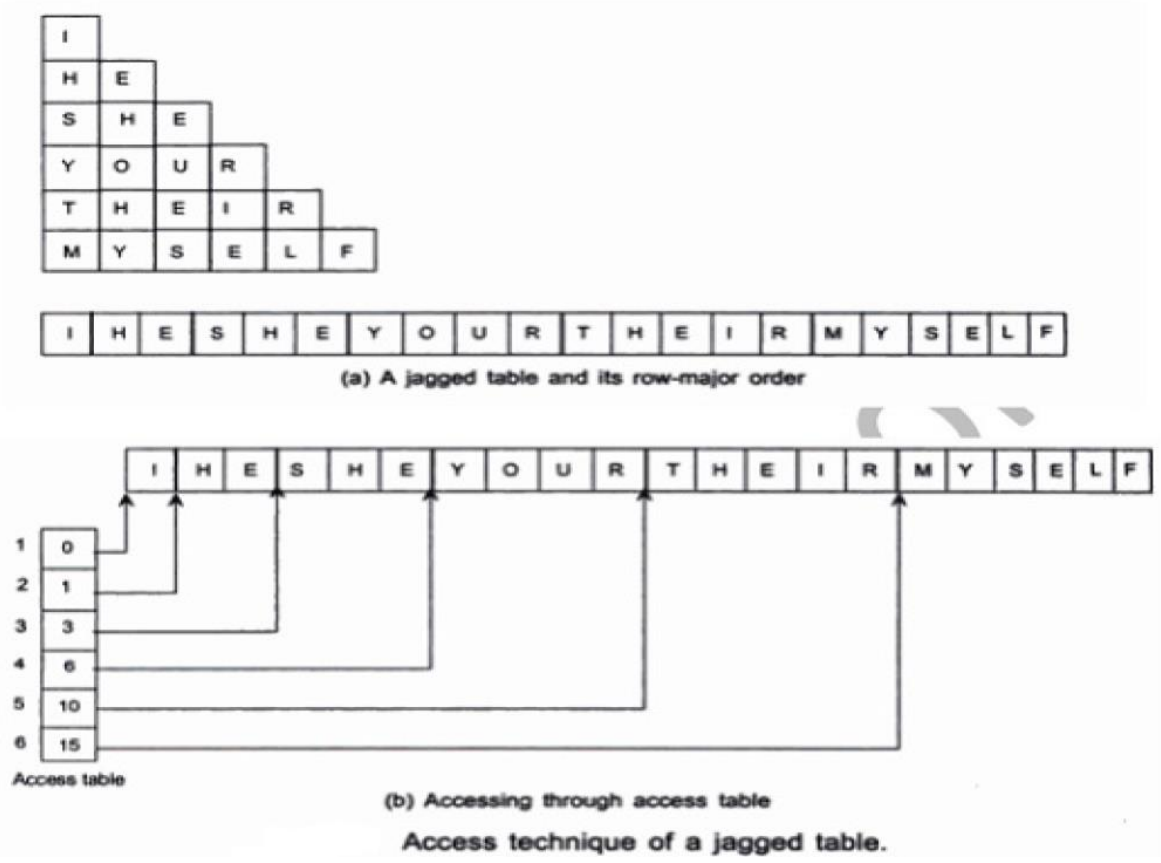


Fig 5.3: Access Array for the Jagged Tables

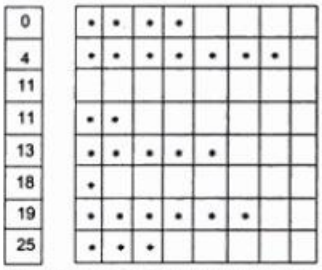
For e.g., if we want to access a54 (element in 5th row and 4th column) then at 5th location of the access table, we see the entry is 10; hence desire element is at 14 (=10+4) location of the array which contains the elements. It is assumed that the 1st element of the table is located at the location of the array.



Access technique of a jagged table.

Above mentioned accessing technique has another advantage over the indexing formula. We can find the indexing formula even if the jagged tables are asymmetric with respect to the arrangement of elements in it. For e.g., in fig (d), it is difficult find index formula. In this case, we can easily maintain its storage in an array and can obtain faster access of elements

from it. In the fig access of elements in the asymmetric matrix, its use is same as that of the symmetric sparse matrices. An entry in the ith location of the access table can be obtained by adding the number of elements in (i-1)th row of the jagged table and (i-1)th entry of the access table, assuming that entry 0 as the first entry in the access table, and as before, the starting location of the array storing the elements is 1.



(a) Accessing of elements in an asymmetric matrix (jagged table).

Inverted tables

The concept of inverted tables can be explain with an example. Suppose, a telephone company maintains records of all the subscribers of a telephone exchange as shown in the fig given below. These records can be used to serve several purposes. One of them requires the alphabetical ordering of name of the subscriber. Second, it requires the lexicographical ordering of the address of subscriber.

Third, it also requires the ascending order of the telephone numbers in order to estimate the cabling charge from the telephone exchange to the location of the telephone connection etc.... To serve these purposes, the telephone company should maintain 3 sets of records: one in alphabetical order of the NAME, second, the lexicographical ordering of the ADDRESS and third, the ascending order of the phone numbers.

(a) Records of a Telephone Exchange

Index	*Name*	*Address*	*Phone*
1	K.R. Narayana	Maker Towers #6	257696
2	A.B. Vajpayee	9 Vivekananda Road	257459
3	L.K. Advani	11 Von Kasturba Marg	257583
4	Mamta Banerjee	342 Patel Avenue	257423
5	Y. Sinha	5 SBI Road	257504
6	D. Kulkarni	369 Faculty Colony	257564
7	T. Krishnamurthy	185 Faculty Colony	257579
8	N. Puranjay	409 Medical Colony	257409
9	Tadi Tabi	Officers Mess #52	257871

This way of maintaining records leads to the following drawbacks,

a. Requirement of extra storage: three times the actual memory

b. Difficulty in modification of records: if a subscriber changes his address, then we have to modify this in three storages to maintain consistency in information.

Using the concept of inverted tables, we can avoid the multiple set of records, and we can still retrieve the records by any of the three keys almost as quickly as if the records are fully sorted by that key.
Therefore, we should maintain an3 inverted table. In this case, this table comprise of three columns:
NAME, ADDRESS, and PHONE as shown in below figure. Each column contains the index numbers of records in the order based on the sorting of the corresponding key. This inverted table, therefore, can be consulted to retrieve information.

(b) Inverted Table

Name	*Address*	*Phone*
2	7	8
6	6	4
1	1	2
3	8	5
4	9	6
8	4	7
7	5	3
9	2	1
5	3	9

Symbol Table:
Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.
A **symbol table** is a major data structure used in a compiler:
o Associates **attributes** with identifiers used in a program
o For instance, a **type attribute** is usually associated with each identifier
o A symbol table is a necessary component
_ Definition (declaration) of identifiers appears once in a program
_ Use of identifiers may appear in many places of the program text
o Identifiers and attributes are entered by the analysis phases
_ When processing a definition (declaration) of an identifier
_ In simple languages with only global variables and implicit declarations:
_ The scanner can enter an identifier into a symbol table if it is not already there
_ In block-structured languages with scopes and explicit declarations:
_ The parser and/or semantic analyzer enter identifiers and corresponding attributes
o Symbol table information is used by the analysis and synthesis phases
_ To verify that used identifiers have been defined (declared)
_ To verify that expressions and assignments are semantically correct – **type checking**

_ To generate intermediate or target code
Symbol Table Interface
_ The basic operations defined on a symbol table include:
o **allocate** – to allocate a new empty symbol table
o **free** – to remove all entries and free the storage of a symbol table
o **insert** – to insert a name in a symbol table and return a pointer to its entry
o **lookup** – to search for a name and return a pointer to its entry
o **set_attribute** – to associate an attribute with a given entry
o **get_attribute** – to get an attribute associated with a given entry
_ Other operations can be added depending on requirement
o For example, a **delete** operation removes a name previously inserted
_ Some identifiers become invisible (out of scope) after exiting a block
_ This interface provides an abstract view of a symbol table
_ Supports the simultaneous existence of multiple tables
_ Implementation can vary without modifying the interface
A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:
<symbol name, type, attribute>
For example, if a symbol table has to store information about the following variable declaration:
static int interest;
then it should store the entry such as:
<interest, int, static>
The attribute clause contains the entries related to the name.
Implementation
If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:
• Linear (sorted or unsorted) list
• Binary Search Tree
• Hash table
Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.
• **Unordered List**
o Simplest to implement
o Implemented as an array or a linked list
o Linked list can grow dynamically – alleviates problem of a fixed size array
o Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average
• **Ordered List**
o If an array is sorted, it can be searched using binary search – $O(\log 2\ n)$
o Insertion into a sorted array is expensive – $O(n)$ on average
o Useful when set of names is known in advance – table of reserved words
• **Binary Search Tree**
o Can grow dynamically
o Insertion and lookup are $O(\log 2\ n)$ on average
Operations
• First consideration is how to **insert** and **lookup** names Variety of implementation techniques
• A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

int a;

should be processed by the compiler as:

insert(a, int);

lookup()

lookup() operation is used to search a name in the symbol table to determine:

• if the symbol exists in the table.
• if it is declared before it is being used.
• if the name is used in the scope.
• if the symbol is initialized.
• if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

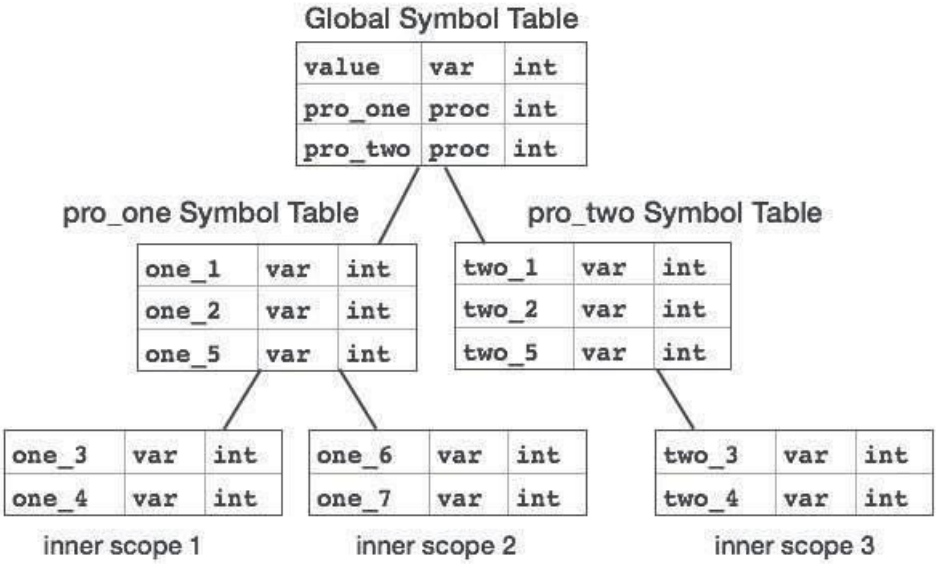
A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
int value=10;
void pro_one()
{
int one_1;
int one_2;
{ \
int one_3; |_ inner scope 1
int one_4; |
} /
int one_5;
{ \
int one_6; |_ inner scope 2
int one_7; |
} /
}
void pro_two()
{
int two_1;
int two_2;
```

```
{ \
int two_3; |_ inner scope 3
int two_4; |
} /
int two_5;
}
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

• first a symbol will be searched in the current scope, i.e. current symbol table.

• if a name is found, then search is completed, else it will be searched in the parent symbol table until, either the name is found or global symbol table has been searched for the name.

Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in

array format where each data value has its own unique index value. Access of data becomes very fast

if we know the index of desired data.

Hash table: *The memory area where the keys are stored is called hash table.*

Properties of Hash table:

• Hash table is partitioned into b buckets, HT(0),..HT(b-1).

• Each bucket is capable of holding ‘s’ records.

• A bucket is said to consist of s slots, each slot being large enough to hold 1 record.

• Usually s=1 and each bucket can hold exactly 1 record.

11

It becomes a data structure in which insertion and search operations are very fast irrespective of size

of data. Hash Table uses array as a storage medium and uses hash technique to generate index where

an element is to be inserted or to be located from.

HASHING TECHNIQUES

Hashing Technique: The hashing technique is the method in which the address or location of a key 'k' is obtained by computing some arithmetic function 'f' of k. f(k) gives the address of k in the table. The address will be referred to as hash address or home address.

Hashing function f(k): The hashing function used to perform an identifier transformation on k. f(k) maps the set of possible identifiers onto the integer 0 through b-1.

Overflow: An overflow is said to occur when a new identifier k is mapped or hashed by f(k) into a full bucket.

Collision: A collision occurs when two non-identical identifiers are hashed into the same bucket. When bucket size s=1, collisions and overflow occur simultaneously. Choose f which is easy to compute and results in very few collisions.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and following items are to be stored. Item are in (key, value) format.

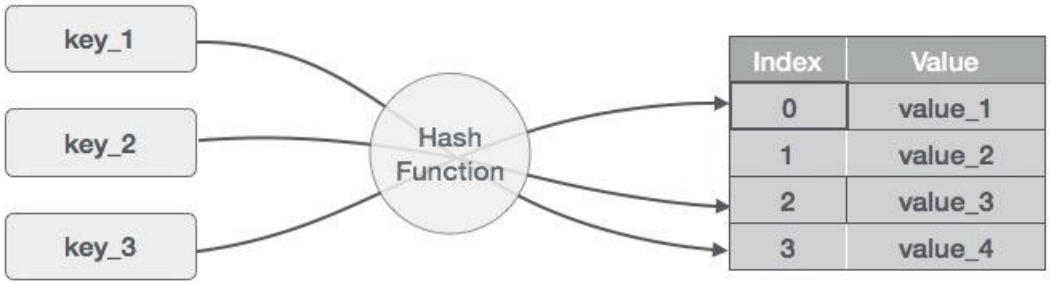


Fig 5.4: Hash Function

(1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17

Factors affecting Hash Table Design:

• Hash function

• Table size-usually fixed at the start

• Collision handling scheme

Hashing functions

A hash function is one which maps an element's key into a valid hash table index h(key) => hash table index.

Hash Function Properties:

• A hash function maps key to integer

Constraint: Integer should be between [0, TableSize-1]

• A hash function can result in a many-to-one mappingn (causing collision) Collision occurs when hash function maps two or more keys to same array index.

• A "good" hash function should have the properties:

1. Reduced chance of collision

Different keys should ideally map to different indices

Distribute keys uniformly over table

2. Should be fast to compute

Different types:

(a) Mid square method.

The key k is squared then the required hash value is obtained by deleting some digits from both ends of k2.

f (k) = specific digits(k2)

For example, let us the compute the hash address for the following key value: 334, 567, 239.

(i) f(334) = 3rd , 4th digits (111556) = 11.

(ii) f(567) = 3rd , 4th digits (321489) = 21.

(iii) f(239) = 3rd , 4th digits (57121) = 57

(b) Division method

In the division method the hash address is the remainder after key k is divided by m, where m is the number of buckets.

f(k) = k mod m.

Let m = 10.

f(334) = 334 mod 10 = 4

f(567) = 567 mod 10 = 7

(c) Folding method

This method involves chopping the key k into two parts and adding them to get the hash address.

f(334) = 03 + 34 =37

f(567) = 05 + 67 = 72

Collision resolution

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, there is approximately a 95% chance of at least two of the keys being hashed to the same slot. Therefore, almost all hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

Techniques to Deal with Collisions:

Chaining

Open addressing

Double hashing, etc.

Chaining

In chaining, the buckets are implemented using linked lists. If there is a collision, then a new node is created and added at the end of the bucket. Hence all records in T with the same hash address h may be linked together to form a linked list. It is also known as open hashing.

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

Separate chaining

In the method known as *separate chaining*, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred.

Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable.

If these cases happen often, the hashing function needs to be fixed.

Separate chaining with linked lists

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the *average* cost of a lookup depends only on the average number of keys per bucket—that is, it is roughly proportional to the load factor.

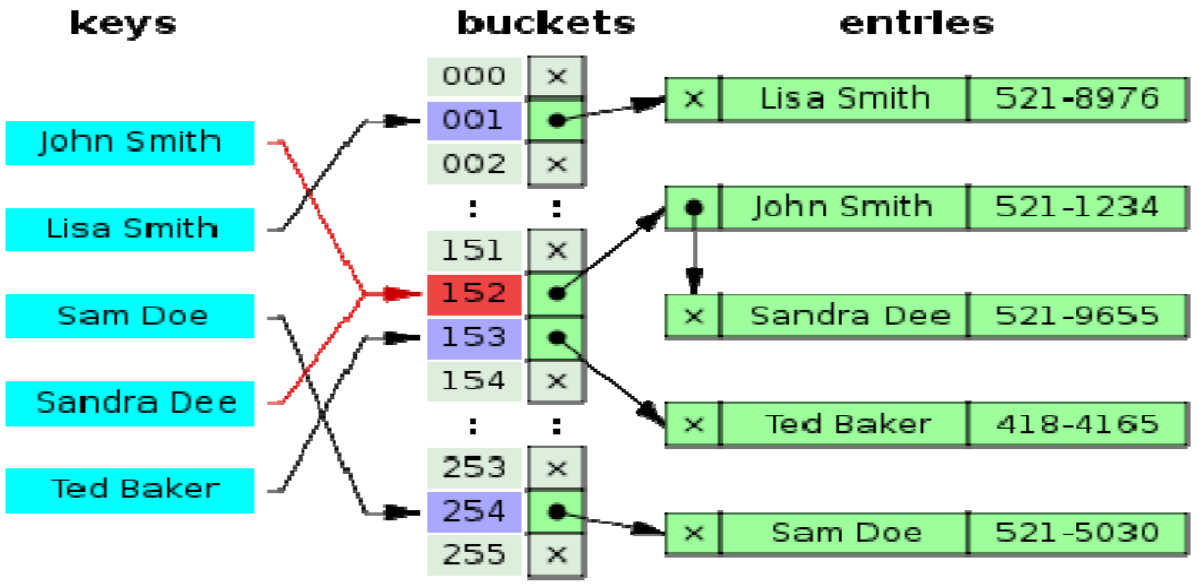


Fig. Hash collision resolved by separate chaining

For this reason, chained hash tables remain effective even when the number of table entries *n* is much higher than the number of slots. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket

data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number *n* of entries in the table.
The bucket chains are often searched sequentially using the order the entries were added to the bucket. If the load factor is large and some keys are more likely to come up than others, then rearranging the chain with a move-to-front heuristic may be effective. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in a worst-case scenario. However, using a larger table and/or a better hash function may be even more effective in those cases.
Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache ineffective.
The advantages of using chaining are
• Insertion can be carried out at the head of the list at the index.
• The array size is not a limiting factor on the size of the table.
The prime disadvantage is the memory overhead incurred if the table size is small.

Potential disadvantages of Chaining

Linked lists could get long
_ Especially when N approaches M
_ Longer linked lists could negatively impact performance
More memory because of pointers
Absolute worst-case (even if N << M):
_ All N elements in one linked list!
_ Typically the result of a bad hash function

Open addressing

In open hashing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. This method is also called closed hashing.
Wellknown probe sequences include:
• Linear probing, in which the interval between probes is fixed (usually 1)
• Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
• Double hashing, in which the interval between probes is computed by a second hash function Collision Resolution by Open Addressing When a collision occurs, look elsewhere in the table for an empty slot
_ Advantages over chaining
_ No need for list structures
_ No need to allocate/deallocate memory during insertion/deletion (slow)
_ Disadvantages
_ Slower insertion – May need several attempts to find an empty slot
_ Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance, Load factor $\lambda \approx 0.5$

Linear Probing

In linear probing, if there is a collision then we try to insert the new key value in the next available free space. As we can see, it may happen that the hashing technique used to create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

Quadratic Probing:

Quadratic Probing is similar to Linear probing. The difference is that if you were to try to insert into a space that is filled you would first check 1^2 = 1 element away then 2^2 = 4 elements away, then 3^2 =9 elements away then 4^2=16 elements away and so on.

With linear probing we know that we will always find an open spot if one exists (It might be a long search but we will find it). However, this is not the case with quadratic probing unless you take care in the choosing of the table size. For example consider what would happen in the following situation:

Table size is 16. First 5 pieces of data that all hash to index 2

• First piece goes to index 2.
• Second piece goes to 3 ((2 + 1)%16
• Third piece goes to 6 ((2+4)%16
• Fourth piece goes to 11((2+9)%16
• Fifth piece dosen't get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven't searched all empty spots.

Double Hashing:

Double Hashing is works on a similar idea to linear and quadratic probing. Use a big table and hash into it. Whenever a collision occurs, choose another spot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. For example, given hash function H1 and H2 and key.

Steps:

• Check location hash1(key). If it is empty, put record in it.
• If it is not empty calculate hash2(key).
• check if hash1(key)+hash2(key) is open, if it is, put it in
• repeat with hash1(key)+2*hash2(key)*, *hash1(key)*+*3*hash2(key) and so on, until an opening is found.

Basic Operations

Following are basic primary operations of a hashtable which are following.

• **Search** − search an element in a hashtable.
• **Insert** − insert an element in a hashtable.
• **Delete** − delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
int data;
int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
//get the hash
int hashIndex = hashCode(key);
//move in array until an empty
while(hashArray[hashIndex] != NULL){
if(hashArray[hashIndex]->key == key)
return hashArray[hashIndex];
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}
return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hash code as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
item->data = data;
item->key = key;
//get the hash
int hashIndex = hashCode(key);
//move in array until an empty or deleted cell
while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}
hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, Compute the hash code of the key passed and locate the index using that hash code as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hash table intact.

```
struct DataItem* delete(struct DataItem* item){
int key = item->key;
//get the hash
int hashIndex = hashCode(key);
//move in array until an empty
while(hashArray[hashIndex] !=NULL){
if(hashArray[hashIndex]->key == key){
struct DataItem* temp = hashArray[hashIndex];
//assign a dummy item at deleted position
```

```
hashArray[hashIndex] = dummyItem;
return temp;
}
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}
return NULL;
}
```

Sets

A Set is a collection of objects need not to be in any particular order. It is just applying the mathematical concept in concept. The set with no element is called null set or empty set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and deletion of elements from the set.

Rules:

Elements should not be repeated.

Each element should have a reason to be in the set.

Example:

Assume we are going to store indian cricketers in a set. The names should not be repeated, as well the name who is not in the team can't be inserted. This is the restriction which need to be followed
in the set.

Representation of Sets:

List

Tree

Hash Table

Bit Vectors

List Representation:

Its a simple and straight forward representation, which is best suited for dynamic storage facility. This representation allows multiplicity of elements i.e., Bag structures. All the operations can be easily implemented and performance of these operations are as good as compared to other representations. A set S={5,6,9,3,2,7,1}

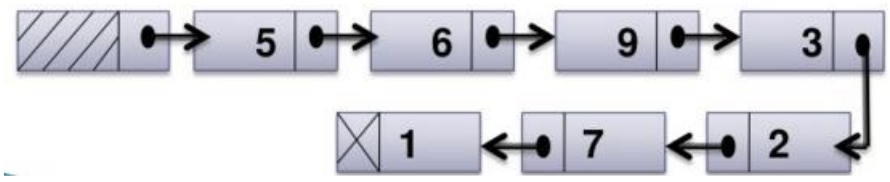


Fig : List Representation

Tree Representation:

Tree is used to represent a set, and each element in the set has the same root. Each element in the set has a pointer to its parent. For example a set S1={1,3,5,7,9,11,13} can be represented as

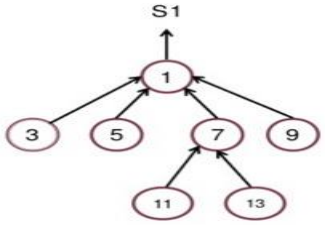


Fig : Tree Representation

Hash Table Representation:

In this representation the elements in collection are seperated into number of buckets. Each bucket can hold arbitrary number of elements. Consider the set S={2,5,7,16,17,23,34,42}, the hash table with 4 buckets and H(x) hash function can store wchi can place element from S to any of the four buckets.

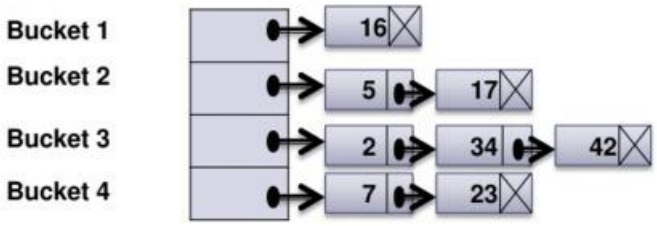


Fig : Hash Table

Bit Vectors Representation:

Two variations are proposed which are maintaining the actual data values, maintaining the indication of presence or absence of data. A set, giving the records about the age of cricketer less than or equal to 35 is as given {0,0,0,0,1,1,1,1,0,1,1}. Here 1 indicates the presence of records having age less than or equal to 35. 0 indicates the absence of records having the age less than or euqla to 35. As 22 we have to indicate presence or absence of an element only, so 0 or 1 can be used for indication of saving storage space.

Operations on Sets:

Basic operations:

• union(S,T): returns the union of sets S and T.
• intersection(S,T): returns the intersection of sets S and T.
• difference(S,T): returns the difference of sets S and T.
• subset(S,T): a predicate that tests whether the set S is a subset of set T.

Operations on a static set structure S:

• is_element_of(*x*,*S*): checks whether the value *x* is in the set *S*.
• is_empty(*S*): checks whether the set *S* is empty.
• size(*S*) or cardinality(*S*): returns the number of elements in *S*.
• iterate(*S*): returns a function that returns one more value of *S* at each call, in some arbitrary order.
• enumerate(*S*): returns a list containing the elements of *S* in some arbitrary order.
• build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.
• create_from(*collection*): creates a new set structure containing all the elements of the given collection or all the elements returned by the given iterator.

Operations on a Dynamic set structure:

• create(): creates a new, initially empty set structure.
• create_with_capacity(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.
• add(*S*,*x*): adds the element *x* to *S*, if it is not present already.
• remove(*S*, *x*): removes the element *x* from *S*, if it is present.
• capacity(*S*): returns the maximum number of values that *S* can hold.

Other operations:
• pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.
• pick(*S*): returns an arbitrary element of *S*. Functionally, the mutator pop can be interpreted as the pair of selectors (pick, rest), where rest returns the set consisting of all elements except for the arbitrary element. Can be interpreted in terms of iterate.
• map(*F*,*S*): returns the set of distinct values resulting from applying function *F* to each element of *S*.
• filter(*P*,*S*): returns the subset containing all elements of *S* that satisfy a given predicate *P*.
• fold(*A0*,*F*,*S*): returns the value *A*|*S*| after applying *Ai+1* := *F*(*Ai, e*) for each element *e* of *S*, for some binary operation *F*. *F* must be associative and commutative for this to be well-defined.
• clear(*S*): delete all elements of *S*.
• equal(*S1, S2*): checks whether the two given sets are equal (i.e. contain all and only the same elements).
• hash(*S*): returns a hash value for the static set *S* such that if equal(*S1, S2*) then hash(*S1*) = hash(*S2*)

Applications:
1. Hash function.
2. Spelling Checker.
3. Information storage and retrieval.
4. Filtering the records from the huge database.

Hash function:
Hash function is a function that generate an integer number with a particular logic. This function is like a one-way entry. We can't reverse this function. But by doing the process with same number we may get an answer. Sometimes the same valyue may be generated for the different values called collision. In that case, we need to check for the data in that location. In some cases, same values will generate the different outout. Both are possible.

Spell Checker:
This concept applied by using the hash table data structure. The two files are provided to us for process. One is 'dictionary file', which is the collection of meaningful words, and another one is 'input file', that contains the word to be validated for spell. Initially the word from the dictionary file and also from input file be inserted into the hash tables seperately. Intersection operation will be done between the two hash tables. If the set with one element is the result of the intersection, then the word is in dictionary. Spell is perfect. If, null set returned from the intersection, then the spell is wrong.

Static tree tables:
• When symbols are known in advance and no insertion and deletion is allowed, it is called a static tree table.
• An example of this type of table is a reserved word table in a compiler.

Dynamic tree tables:
• A dynamic tree tables are used when symbols are not known in advance but are inserted as they come and deleted if not required.
• Dynamic keyed tables are those that are built on-the-fly.
• The keys have no history associated with their use.