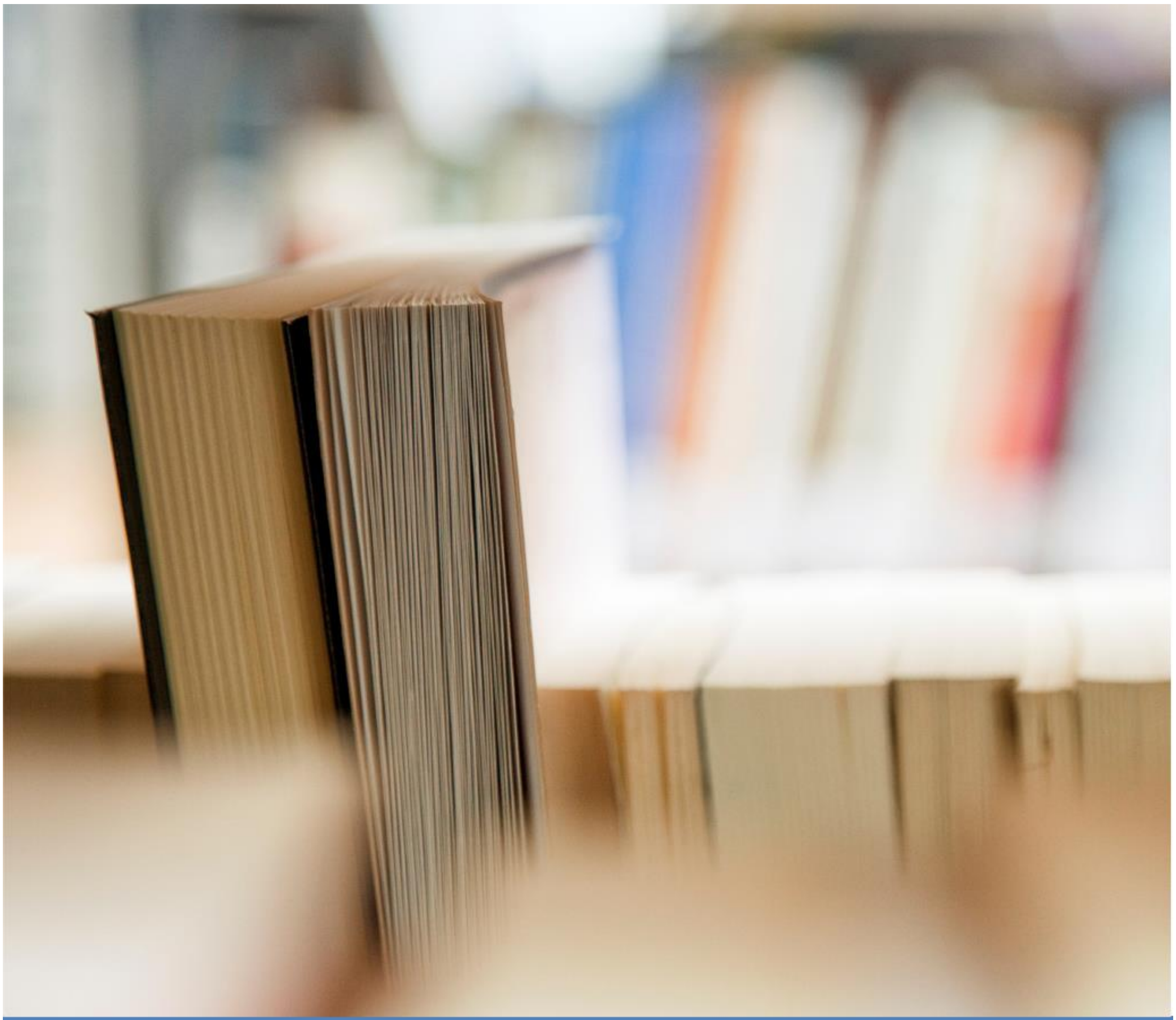
USM's Shriram Mantri Vidyanidhi Info Tech Academy



DATA STRUCTURES

BY : AMAR PANCHAL

INDEX

CHAPTER 5

TREES

INTRODUCTION

In the previous chapters, we have studied various data structures such as arrays, stacks, queues and linked list. All these are linear data structures. In these data structures the elements are arranged in a linear manner, i.e. one after another. Tree is an equally useful data structure of non-linear type. In tree, elements are arranged in non-linear fashion. A tree structure means data is organized in branches. The following Fig. 5.1 is a sample tree.

A tree is a non-linear data structure and its elements are arranged in sorted order.

Tree has several practical applications. It is immensely useful in manipulating data and to protect hierarchical relationship among data. The fundamental operations such as insertion, deletion etc. is easy and efficient in tree data structure than in linear data structures. Fig. 5.2 represents family hierarchy, which keeps relations among them. The hierarchy gives relations between associates of family members. In this tree, node 'A' can be assumed as a parent of 'B' and 'C', 'D' and 'E' are the children of 'B'. 'F' is the child of 'C'. This tree represents the relation between the family members.

Figure 5.1. Sample tree

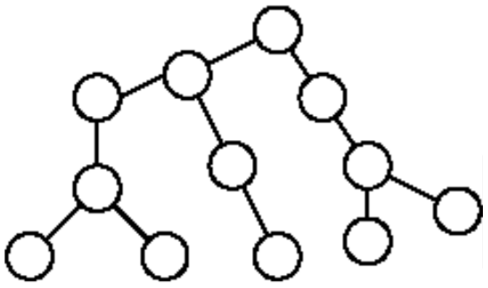
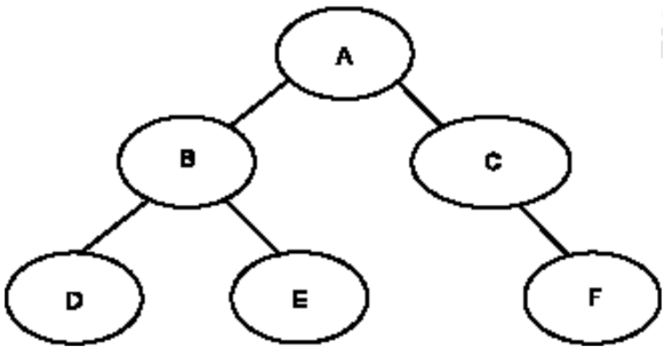


Figure 5.2. A family hierarchy in tree structure



For example, suppose the left side member is male and right side member is female. Then, various relations such as sister, brother, grandfather, grandmother can also be implied.

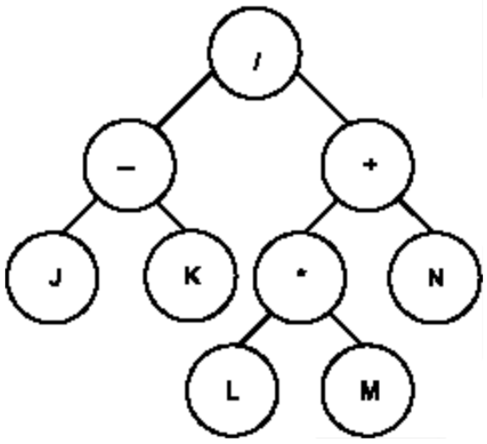
The algebraic expression can be represented with a tree. Consider the following example of an algebraic expression.
Z=(J-K)/((L*M)+N)

The operators of the above equation have different priority. The priority of the operators can be represented by the tree structure. The operators of high priority are at low level and operator and associated operands are represented in tree structure. The Fig. 5.3 illustrates the representation of an algebraic expression.

BASIC TERMS

Some of the basic concepts relevant to trees are described in this section. These are node, parents, roots, child, link, leaf, level, height, degree of node, sibling, terminal nodes, path length, and forest.

Figure 5.3. An algebraic expression in tree form



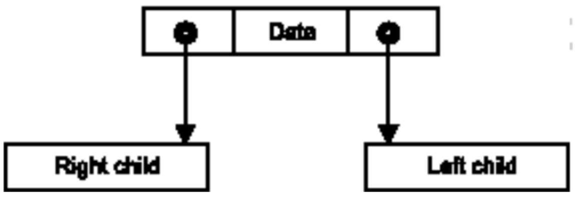
Root

It is the mother node of a tree structure. This is the most important node of any tree. This node does not have parent. It is the first node in the hierarchical arrangement.

Node

It is the main component of the tree. The node of a tree stores the data and its role is same as the linked list. Nodes are connected by means of links with other nodes. This is shown in Fig. 5.4.

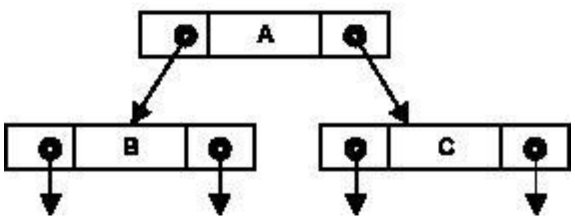
Figure 5.4. Right and left nodes of a tree



Parent

It is an immediate predecessor of a node. In the Fig. 5.5 A is parent of B and C.

Figure 5.5. Parent nodes with child nodes



Child

When a predecessor of a node is parent then all successor nodes are called child nodes. In Fig. 5.5, B and C are child nodes of A. The node at left side is called left child node and node at right side is called right child node.

Link

The link is nothing but pointer to node in a tree structure. In other words, link connects the two nodes. The line drawn from one node to other node is called a link. Fig. 5.5 shows left and right child. Here, two links are shown from node A. More than two links from a node may be drawn in a tree. In a few textbooks the term edge is used instead of link. Functions of both of them are same.

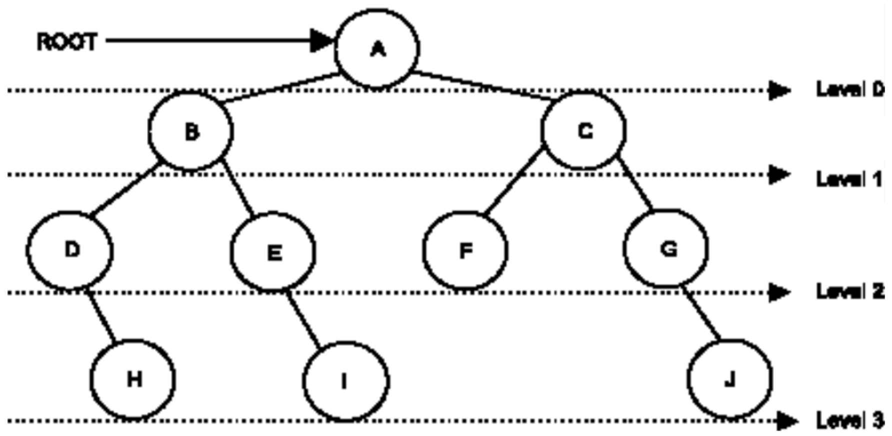
Leaf

This node is located at the end of the tree. It does not have any child hence it is called as leaf node. Here, 'H', 'I', 'F', and 'J' are leaf nodes in Fig. 5.6.

Level

Level is a rank of tree hierarchy. The whole tree structure is levelled. The level of root node is always at 0. The immediate children of root are at level 1 and their immediate children are at level 2 and so on. If-children nodes are at level n+1 then parent node would be at level n. The Fig. 5.6 shows the levels of tree.

Figure 5.6. Levels of tree



Height

The highest number of nodes that is possible in a way starting from the first node (root) to a leaf node is called the height of tree. In Fig. 5.6, the height of tree is 4. This value can be obtained by referring three different paths from the source node to leaf node. The paths A-B-D-H, A-B-E-I, and A-C-G-J have the same height. The height can be obtained from the number of levels, which exists in the tree. The formula for finding the height of the tree h = i_{max} +1, where h is the height and i_{max} is maximum level of the tree. In the above Fig. 5.6 the maximum level of the tree is 3(i_{max}=3). By substituting the value into the formula the h will be 4. The term depth can be used in place of height.

Degree of a Node

The maximum number of children that can exist for a node, is called as the degree of the node. In Fig. 5.6 the node A, B and C have maximum two Children. So, the degree of A, B and C is same and it is equal to 2.

Sibling

The child nodes of same parent are called sibling. They are also called brother nodes. A, B and C nodes in the Fig. 5.6 have two child nodes. B and C are the siblings of the node A, whereas D and E are the siblings of the node B.

Terminal Node

A node with degree zero is called terminal node or leaf. Fig. 5.6 shows 4 terminal nodes and they are H, I, F and J.

Path Length

It is the number of successive edges from source node to destination node. In the above Fig. 5.6 the path length from the root node A to H is three because there are three edges.

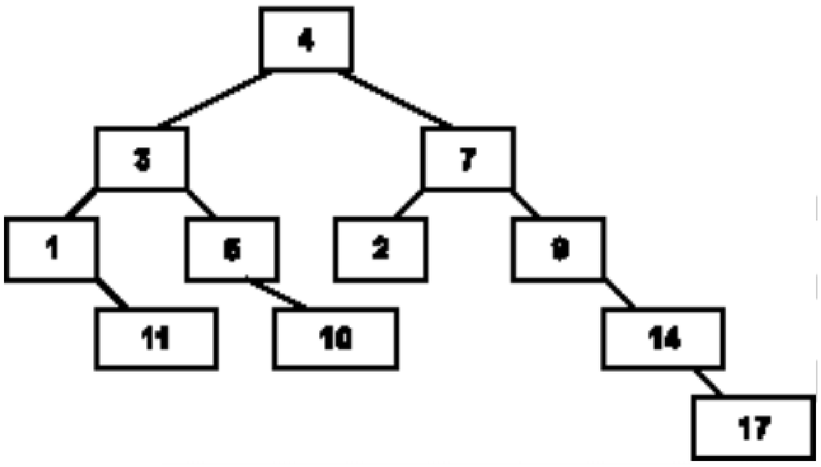
Forest

It is a group of disjoint trees. If we remove a root node from a tree then it becomes the forest. If we remove root node A then the two disjoint sub-trees will be observed. They are left sub-tree B and right sub-tree C.

Labelled Trees

In the labelled tree the nodes are labelled with alphabetic or numerical values. The labels are the values allotted to the nodes in the tree. Fig. 5.7 shows the diagram of a labelled tree. Finally obtained tree is called as labelled tree.

Figure 5.7. Labelled tree



The Fig. 5.7 shows the labelled tree having 11 nodes; root node is 4 and leaf nodes 11,10,17, and 2. The parent and children relationship between them is shown in Table 5.1.

Table 5.1. Parent and children relationship

Parent nodes	Children nodes
4	3,7
3	1,5
1	11
5	10

Table 5.1. Parent and children relationship

Parent nodes	Children nodes
7	2,9
9	14
14	17

There is one more relationship, which is called left node left child and right node right child. This is useful for traversing the binary tree.

Table 5.2. Relationship between parent, left and right nodes

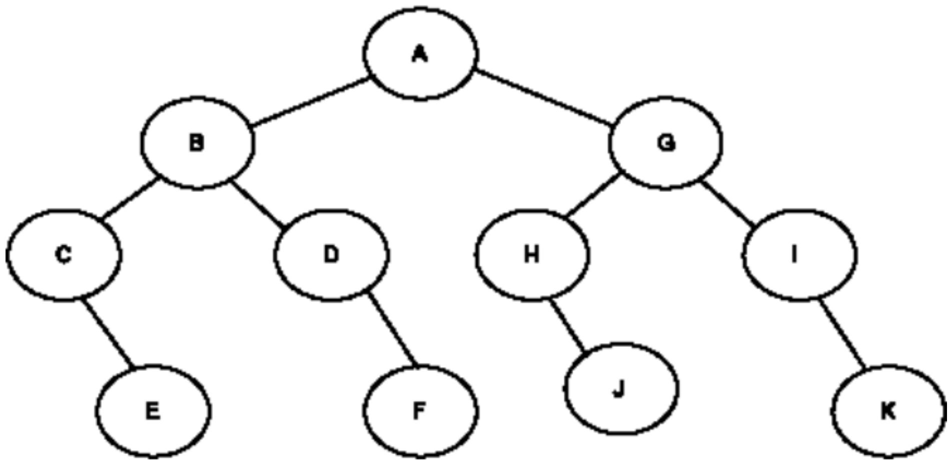
Parent node	Left child	Right child
4	3	7
3	1	5
1	-	11
5	-	10
7	2	9
9	-	14
14	-	17

Table 5.2 describes the relationship between the parents, left and right nodes, which is drawn from the Fig. 5.7. The '—' indicate that root node (parent) does not have child node. The 11,10 and 17 labelled nodes are the leaf nodes of the tree.

BINARY TREES

A binary tree is a finite set of data elements. A tree is binary if each node of it has a maximum of two branches. The data element is either empty or holds a single element called root along with two disjoint trees called left sub-tree and right sub-tree, i.e. in a binary tree the maximum degree of any node is two. The binary tree may be empty. However, the tree cannot be empty. The node of tree can have any number of children whereas the node of binary tree can have maximum two children. Fig. 5.8 shows a sample binary tree.

Figure 5.5. A sample binary tree



In Fig. 5.8, A is the root and B and G are its child nodes. The nodes B and G are non-empty nodes, hence they are called left successor and right successor of the root A. The node root without successor is called the terminal node of that root. In Fig. 5.8 the node E, F, J, and K are the terminal nodes.

The right tree is G and left tree is B. Next B has left tree C and right tree D. The right tree further has left tree H and right tree I. This will be continued up to last level.

COMPLETE BINARY TREE

A tree is called complete binary tree if each of its nodes has two children, except the last nodes. In other words, every non-terminal node of it must have both children except the last leaf nodes. So, at any level the maximum number of nodes is equal to 2. At level 0, there must be only one node and that is the root node. A at level 1 the maximum nodes must be 2. At level 3 the maximum nodes must be equal to 8. A complete binary tree can be obtained from Fig. 5.8.

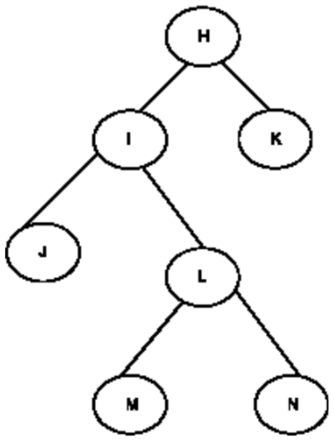
The advantage of this complete binary tree is that one can very easily locate the position of the parent and also left and right child nodes of a complete binary tree. Left child node and right child nodes are located at 2N and 2N+1. Similarly, the parent node of any node would be at floor (N/2).

Parent of D would be floor (5 / 2)=2, i.e. B is the parent and its left and right child are 2*2=4 and 2*2+1=5. 4 and 5 are the C and D child nodes in Fig. 5.8.

STRICTLY BINARY TREE

When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree. It is shown in Fig. 5.9.

Figure 5.9. Strictly binary trees



In the strictly binary tree as shown in Fig. 5.9, L and I are non-terminal nodes with non-empty left and right sub trees.

EXTENDED BINARY TREE

When every node of a tree has either 0 or 2 children then such a tree is called extended binary tree or 2-tree. The nodes with two children are called internal nodes. The nodes without children are known as external nodes. At some places in order to identify internal nodes in figures 5.10 to 5.13 circles are used. To identify external nodes squares are used. The nodes in binary tree that have only one child can be extended with one more child. This extended binary tree can be used for implementing the algebraic equation because in the algebraic equation the left and right child nodes are operands and the parent of the child represents the operator.

Figure 5.10. Binary tree

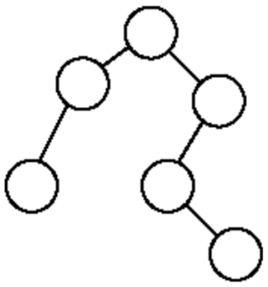


Figure 5.11. Extended 2-tree

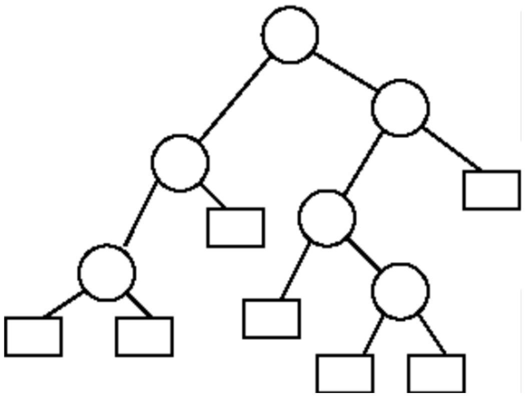


Figure 5.12. Binary trees

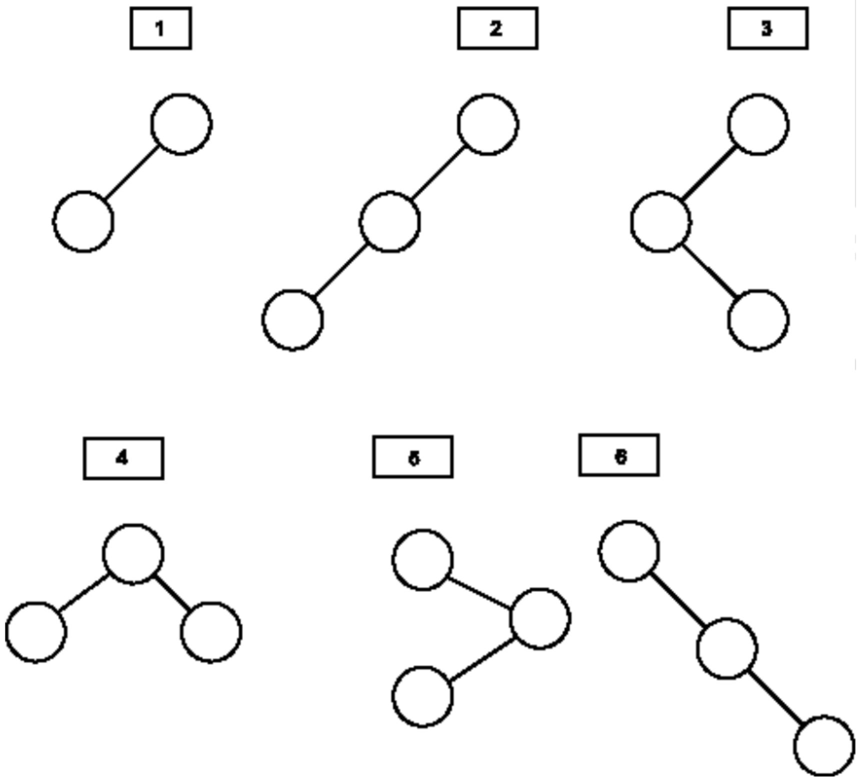
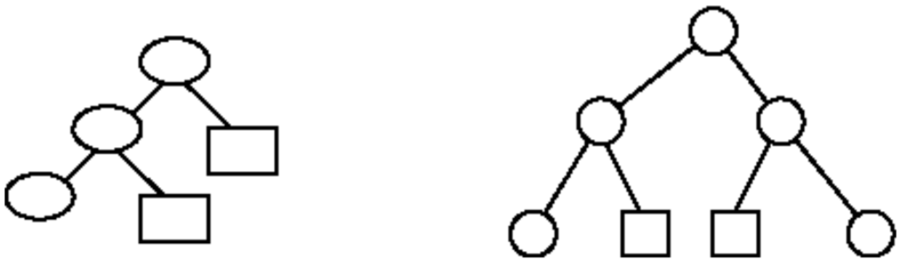


Figure 5.13. 2-trees

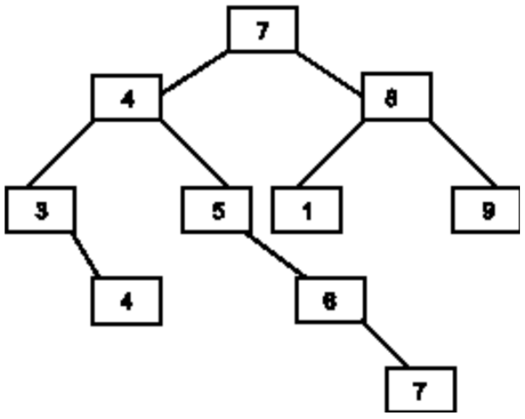


BINARY SEARCH TREE

A binary search tree is also called as binary sorted tree. Binary search tree is either empty or each node N of tree satisfies the following property:

1. The key value in the left child is not more than the value of root.
2. The key value in the right child is more than or identical to the value of root.
3. All the sub-trees, i.e. left and right sub-trees follow the two rules mentioned above.

Binary search tree is shown in Fig. 5.33.
In Fig. 5.33 number 7 is the root node of the binary tree. There are two sub-trees to root 7. The left sub-tree is 4 and right sub-tree is 8. Here, the value of left sub-tree is lower than root and value of right sub-tree is higher than root node. This property can be observed at all levels in the tree.
Figure 5.33. Binary search tree



5.11.1 Searching an Element in Binary Search Tree

The item which is to be searched is compared with the root node. If it is less than the root node then the left child of left sub tree is compared otherwise right child is compared. The process would be continued till the item is found.

Insertion of an Element in Binary Search Tree

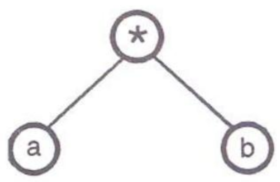
Insertion of an element in binary search tree needs to locate the parent node. The element to be inserted in the tree may be on the left sub-tree or right sub-tree. If the inserted number is lesser than the root node then left sub-tree is recursively called, otherwise right sub-tree is chosen for insertion.

Expression Tree

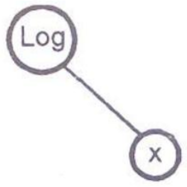
Definition :

An expression tree is a tree built up from the simple operands as the leaves of binary tree and operators as the interior nodes.

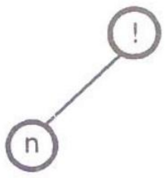
Example :Some o the expression trees are given in Fig.



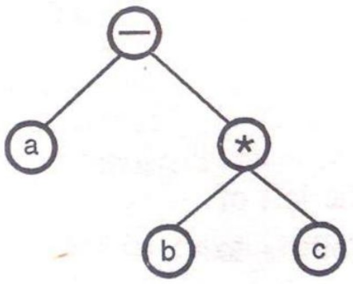
*a * b*



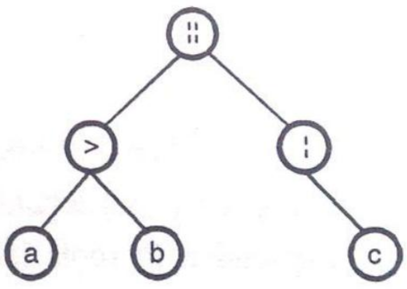
Log x



n! (factorial n)



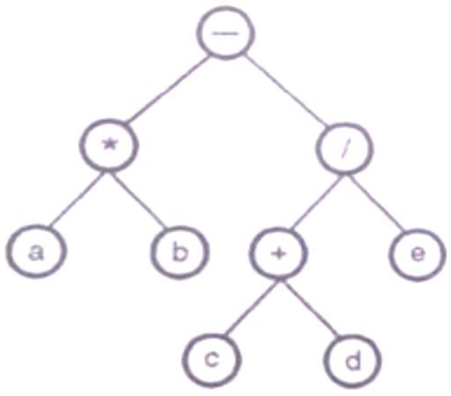
*a – b * c*



(a > b) || (!c)

Let us see one more example, construct an expression tree for a * b – c + d/e.

This is shown in Fig



a * b – c + d/e

But one may ask the question is it the only possible tree for the given expression? Definitely, one may select any operator as root node, and which will result in different results. Thus to obtain the correct result, one has to consider the priority of operators. But, if we are given with infix and one of the other notations, then we can construct uniquely the expression tree.

Construction of an Expression tree from infix and prefix expressions :
Consider the inorder/infix sequence :
a – b * c + d
and preorder sequence,
* –ab + cd
To construct the equivalent binary tree follow the steps given below :
Step 1 : Read the preorder sequence, the first element becomes the root.
Step 2 : Now scan the infix sequence, till you get an element found in step 1. Place all the elements left of this element (of infix expression) to the left of root and others to right.
Step 3 : Repeat steps 1 and 2, till all the elements from infix sequence gets placed in tree.
Step 4 : Stop.

The conversion process is given below :
Infix : a – b*c + d
Prefix : *–ab + cd
↑
* becomes root :
Place all elements (from leftside) to the left of '*' from infix expression to roots left and others to the right.Now scan next element from prefix, which becomes root for subtree. Check the subexpression(subtree). Again separate left is right subexpression.

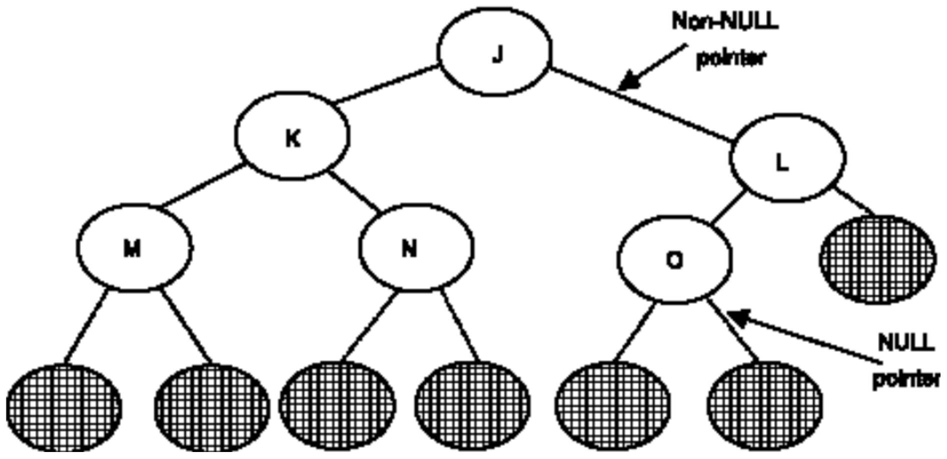
We now get, When we read next element from prefix we get (a, b) which have been already dealt with (they have become leaves). So no further processing required, i.e. a, b an operand.
Next we get '+' in prefix, which is not yet processed, (it has not become leaf nor a root of subtree). Hence scan the sub expression, and rearranging the elements we get, Thus it shows that we get the desired result.

Construction of an expression tree from a given postfix expression :
Let us consider the postfix expression
abc +*,
Now to construct an expression tree, we follow the steps given below :
Step 1 : Read the input character.
Step 2 : If it is an operand then push address of this node onto stack and goto step 4.
Step 3 : If the character is an operator, then pop twice, which gives the address of two operands. Create a new node for this operator and attach the operands to its left and right branch. Push the address of this node onto stack.
Step 4 : If all the characters from postfix expression are not read, then goto step 1.
Step 5 : Pop the content of stack which gives the root address of expression tree.
Step 6 : Stop.

THREADED BINARY TREE

While studying the linked representation of a binary tree, it is observed that the number of nodes that have null values are more than the non-null pointers. The number of left and right leaf nodes has number of null pointer fields in such a representation. These null pointer fields are used to keep some other information for operations of binary tree. The null pointer fields are to be used for storing the address fields of higher nodes in tree, which is called thread. Threaded binary tree is the one in which we find these types of pointers from null pointer fields to higher nodes in a binary tree. Consider the following tree:

Figure 5.34. A binary tree with null pointers

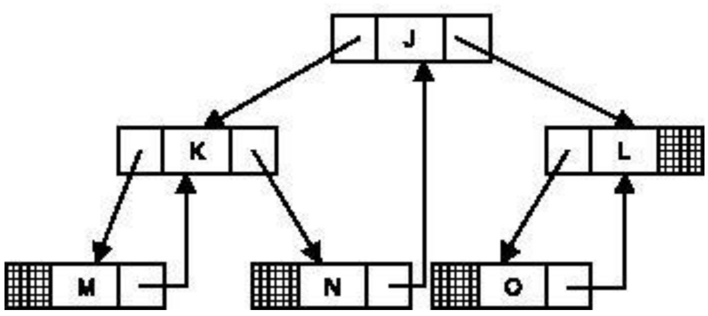


In Fig. 5.34, in the binary tree there are 7 null pointers. These are shown with the dotted lines. There are total 12 node pointers out of which 5 are actual node pointers, i.e. non-null pointer (solid lines). For any binary tree having n nodes there will be (n+1) null pointers and 2n total pointers. All the null pointers can be replaced with appropriate pointer value known as thread. The binary tree can be threaded according to appropriate traversal method. The null pointer can be replaced as follows:

Threaded binary tree can be traversed by any one of the three traversals, i.e. preorder, postorder and inorder. Further, in inorder threading there may be one-way inorder threading or two-way inorder threading. In one way inorder threading the right child of the node would point to the next node in the sequence of the inorder traversal. Such a threading tree is called right in threaded binary tree. Also, the left child of the node would point to the previous node in the sequence of inorder traversal. This type of tree is called as left in threaded binary tree. In case both the children of the nodes point to other nodes then such a tree is called as fully threaded binary tree.

Fig. 5.35 describes the working of right in threaded binary tree in which one can see that the right child of the node points to the node in the sequence of the inorder traversal method.

Figure 5.35. Right in threaded binary tree



The inorder traversal shown in the Fig. 5.35 will be as M-K-N-J-O-L. Two dangling pointers are shown to point a header node as shown below:

- rchild of M is made to point to K
- rchild of N is made to point to J
- rchild of O is made to point to L.

Similarly, the working of the left in binary threaded tree is illustrated in Fig. 5.36. In this case the left child of node points to the previous node in the sequence of inorder traversal.
As shown in Fig. 5.36, thread of N points to K. Here, K is the predecessor of N in inorder traversal. Hence, the pointer points to K. In this type of tree the pointers pointing to other nodes are as follows:

- lchild of N is made to point to K
- lchild of O is made to point to J.

Figure 5.36. Left in threaded binary tree

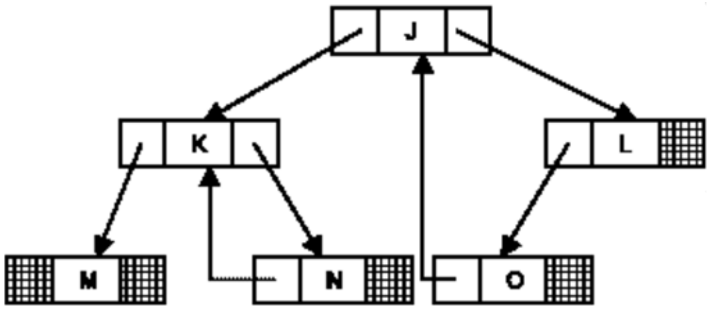
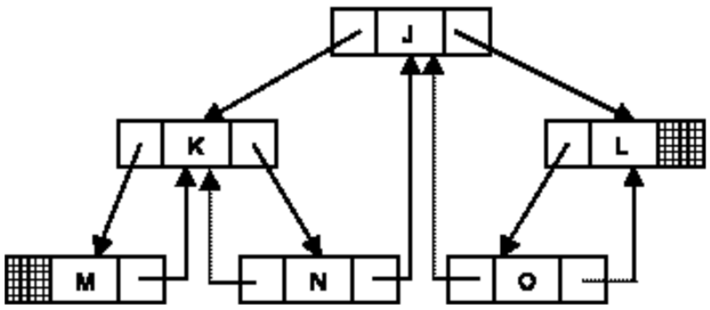


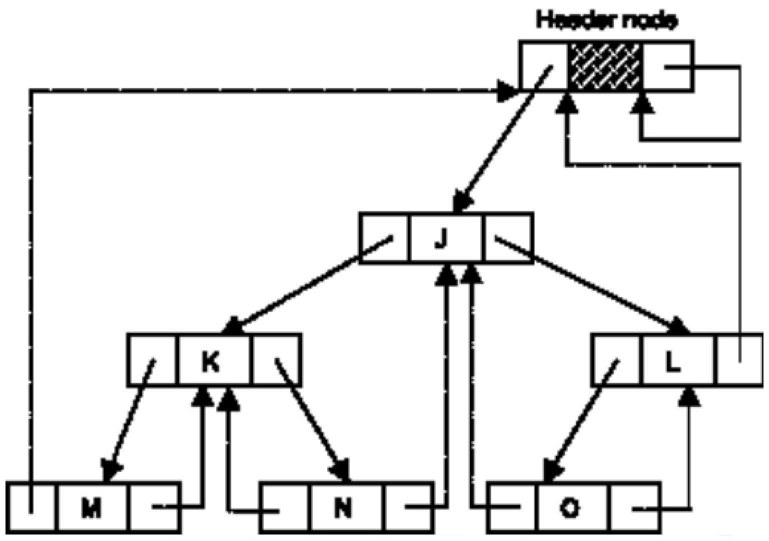
Fig. 5.37 illustrates the operation of fully threaded binary tree. Right and left children are used for pointing to the nodes in inorder traversal method.

- rchild of M is made to point to K
- lchild of N is made to point to K
- rchild of N is made to point to J
- lchild of O is made to point to J
- rchild of O is made to point to L.

Figure 5.37. Fully threaded binary tree



Fully threaded binary tree with header is described in Fig. 5.38.
rchild of M is made to point to K
lchild of M is made to point to Header
lchild of N is made to point to K
rchild of N is made to point to J
lchild of O is made to point to J
rchild of O is made to point to L
rchild of L is made to point to Header.
Figure 5.38. Fully threaded tree with header



The working of the fully threaded binary tree is illustrated in Fig. 5.38. In this case the left child of node points to the previous node in the sequence of inorder traversal and right child of the node points to the successor node in the inorder traversal of the node. In the previous two methods left and right pointers of the first and last node in the inorder list are NULL. But in this method the left pointer of the first node points to the header node and the right pointer of the last node points to the header node. The header node's right pointer points to itself, and the left pointer points to the root node of the tree. The use of the header is to store the starting address of the tree. In the fully threaded binary thread each and every pointer points to the other nodes. In this tree we do not find any NULL pointers.

In the Fig. 5.38 the first node in the inorder is M and its left pointer points to the left pointer of the header node. Similarly, the last node in the inorder is L and its right pointer points to the left pointer of the header.

In memory representation of threaded binary tree, it is very important to consider the difference between thread and normal pointer. The threaded binary tree node is represented in Fig. 5.39.

Figure 5.39. Representation of the node

LTHREAD	DATA	RTHREAD

Each node of any binary tree stores the three fields. The left field stores the left thread value and the right field stores the right thread value. The middle field contains the actual value of the node, i.e. data.

AVL TREE

HEIGHT-BALANCED TREE

The efficiency of searching process in binary tree depends upon the method in which the data is organised. A binary tree is said to be completely balanced binary tree if all its leaves present at nodes of level h or h-1 and all its nodes at level less than h-1 contain two children.

Figure 5.40. Full binary tree

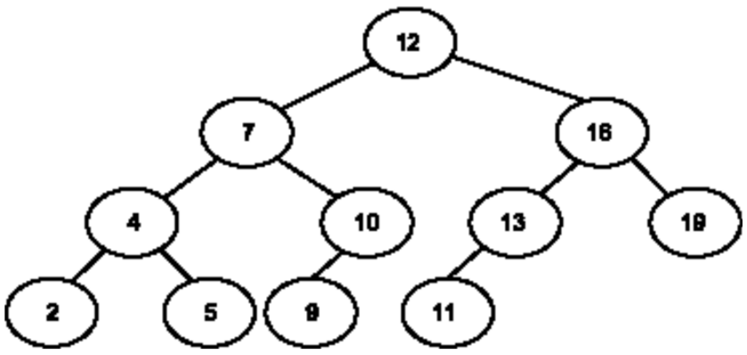
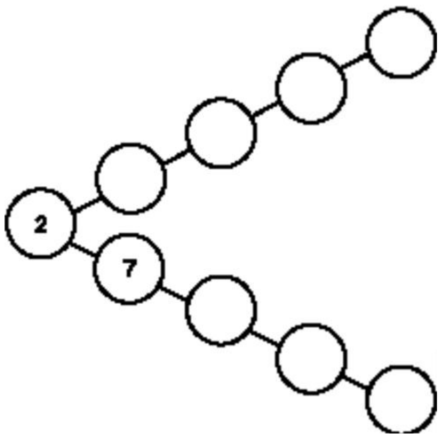


Figure 5.41. A degenerate binary tree

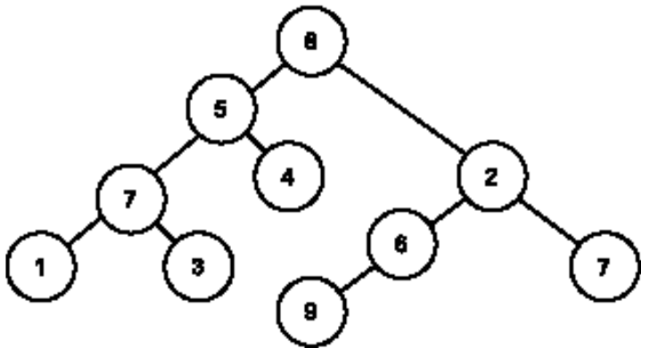


A tree is height balanced tree when each node in the left sub-tree varies from right sub-tree by not more than one.

A nearly height balanced tree is called an AVL. This form of tree is studied and defined by Russian mathematician G.M. Adel'son-Velskii and E.M. Landis in 1962.

We can conclude number of nodes might be present in a balanced tree having height h. At level one there is only one node, i.e. root. In every successive level the number of nodes increases i.e. 2 nodes at level 2, 4 nodes at level 3, and so on. There will be 2^{1-1} nodes at level 1. Thus, we can calculate total number of nodes from level 1 through level h-1 will be $1+2+2^2+2^3+.....+2^{h-2} = 2^{h-1} -1$. The number of nodes at level h may be from 1 to 2h-1 nodes. The total number of nodes (n) of tree range from 2h-1 to 2h-1 or (2h-1-1+1).

Figure 5.42. Completely balanced tree

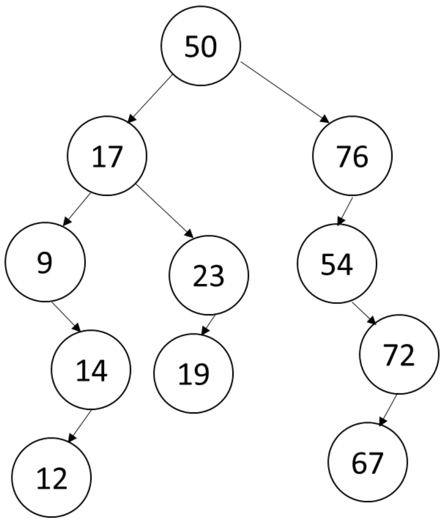


An AVL tree should satisfy the following rules:

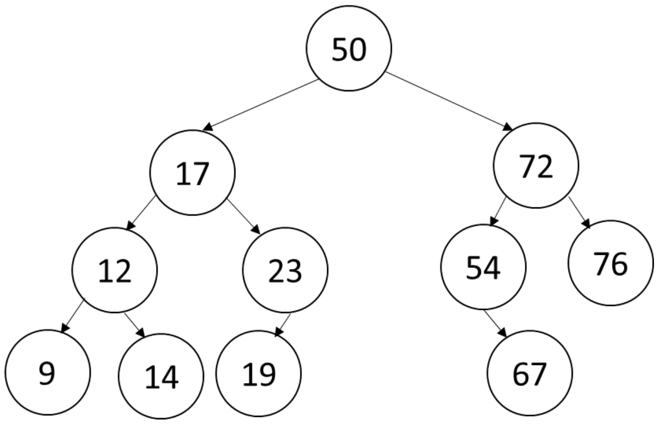
1. A node is known as left heavy if the longest path in its left child (left sub-tree) is longer than the longest path in right sub-tree.
2. A node is known as right heavy if the longest path in its right sub-tree is longer than left sub-tree.
3. A node is known as balanced if the longest path in left and right sub-trees are identical.

AVL tree

In computer science, an AVL tree is a self-balancing binary search tree, and it is the first such data structure to be invented.[1] In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
he balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.



Unbalanced tree



Balanced

Operations

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

Insertion

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

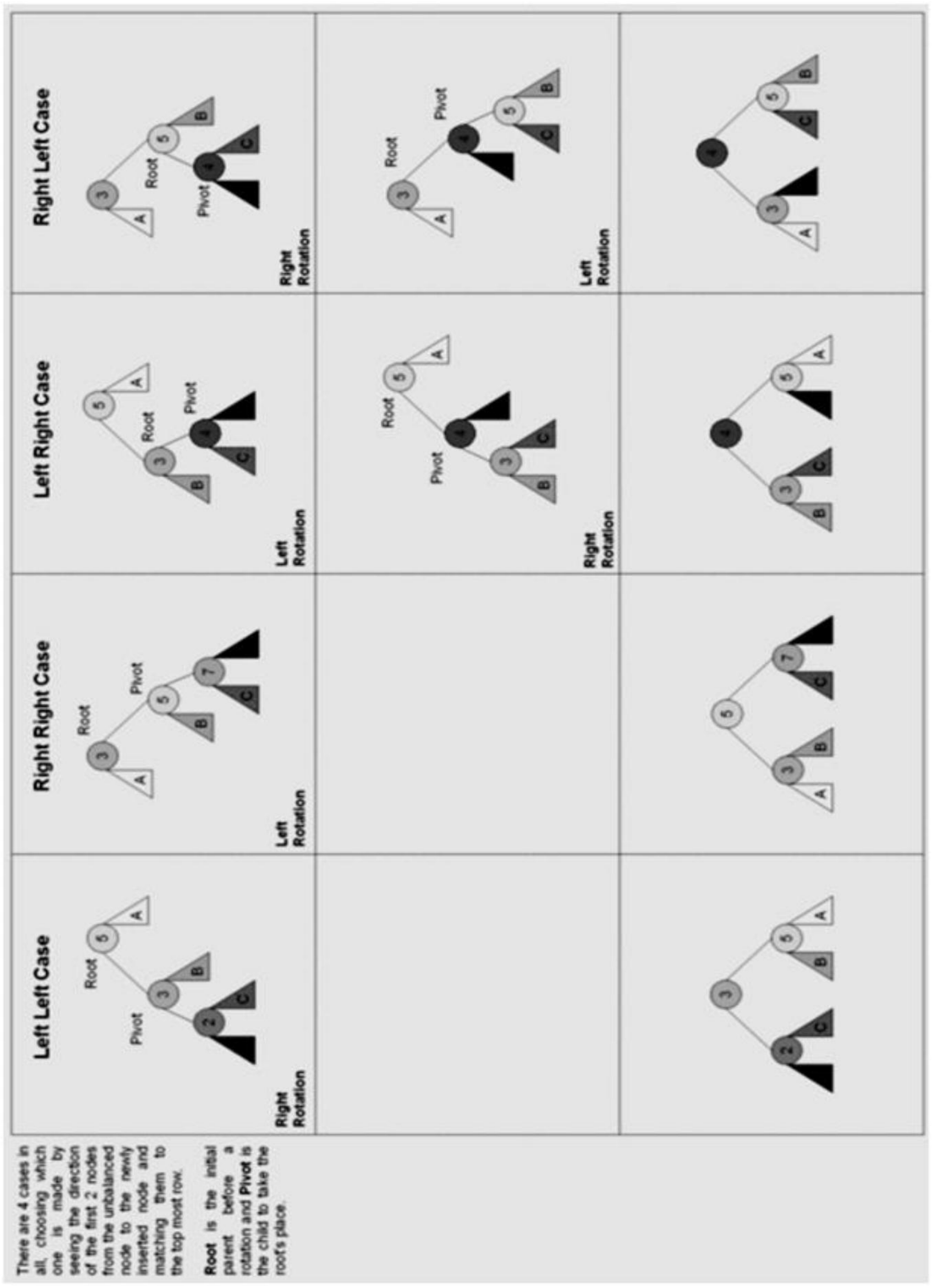
If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary.

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

There are basically four cases which need to be accounted for, of which two are symmetric to the other two. For simplicity, the root of the unbalanced subtree will be called P, the right child of that node will be called R, and the left child will be called L. If the balance factor of P is 2, it means that the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must then be checked. If the bala nce factor of R is 1, it means the insertion occurred on the (external) right side of that node and a left rotation is needed (tree rotation) with P as the root. If the balance factor of R is -1, this means the insertion happened on the (internal) left side of that node. This requires a double rotation. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

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

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



There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.

Deletion

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

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

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

B-TREE (BALANCED MULTI-WAY TREE)

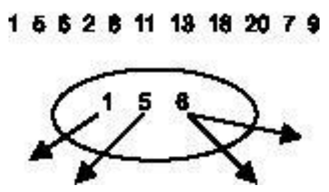
Binary search tree is, in general called multi-way search tree. The integer m is called the order of the tree. Each node should have maximum m children. If k ≤ m, where m is number of children, then node has accurately k-1 keys which divides all the keys into k number of sets. In case some sets are empty, the children are also empty. The B-tree is also known as the balanced sort tree. The B-tree is used in external sorting. The B-tree is
not a binary tree. While implementing B-tree following conditions are followed:

1. The height of the tree must be minimum.
2. There should be no empty sub-trees after the leaves of the tree.
3. The leaves of the tree should be at the same level.
4. All nodes excepting the leaves should have at least few children.

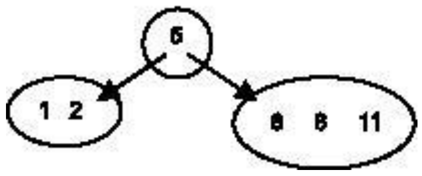
5.14.1 B-Tree Insertion

In B-tree insertion at first search is made where the new element is to be placed. If the node is suitable to the given element then insertion is straightforward. The element is inserted by using an appropriate pointer in such an order that number of pointers will be one more than the number of records. In case, the node overflows due to upper bound of node, splitting is mandatory. The node is divided into three parts. The middle part is passed upward. It will be inserted into the parent. Partition may spread the tree. This is because the parent into which element is to be inserted spilts into its child nodes. If the root is needed to be split, a new root is created with two children. The tree grows by one level.

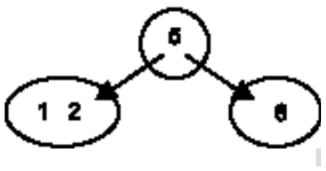
Example: Consider the following B-tree of degree 4. It can be balanced in four ways. Here, each node holds elements. It also has four branches. Suppose, it has the following values:



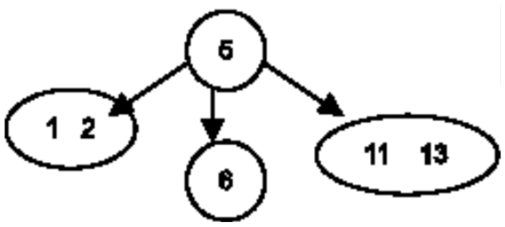
The value 1 is put in a new node. This node can also hold next two values.



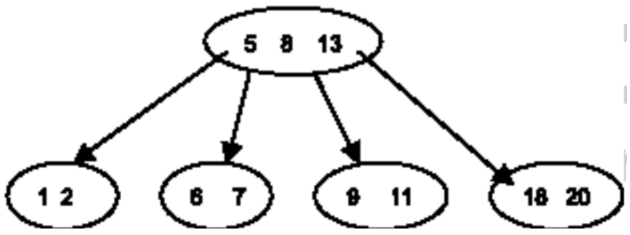
When value 2 (4th value) is put, the node is split at 5 into leaf nodes. Here, 5 is parent. The element 8 is added in leaf node. The search for its accurate position is done in the node having value 6. The element 8 also is present in the same node.



The element 13 is to be inserted. However, the right leaf node, in which 1 to 3 values have appropriate plane, is occupied. Hence, the node splits at median 8 and this moves it up to the parent.



By following the above procedure the remaining nodes can be included. The final figure would be as follows:

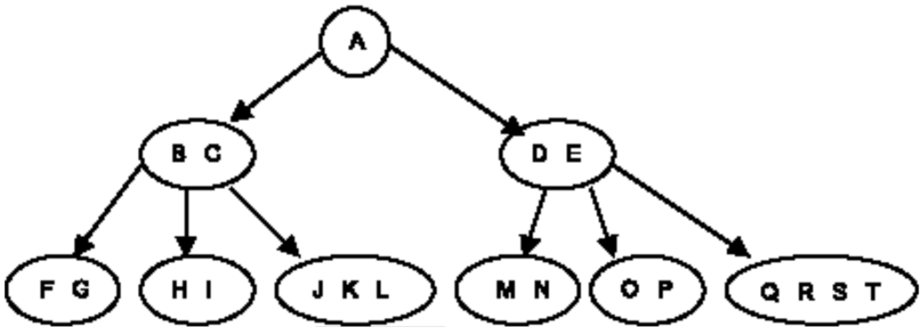


B-Tree Deletion

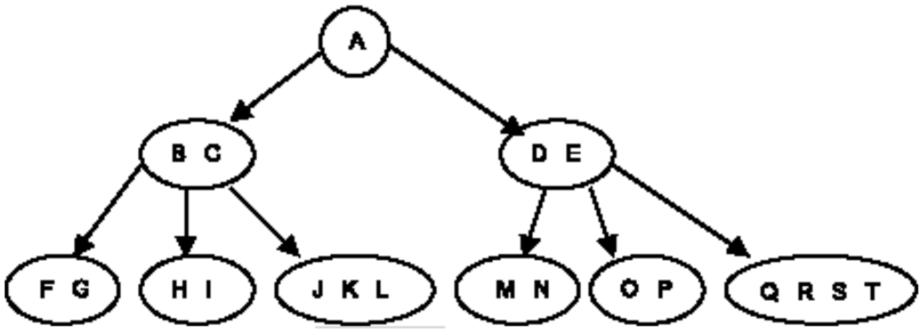
In B-Tree deletion the element which is to be deleted is searched. In case the element is terminal node, the deletion process is straightforward. The element with a suitable pointer is deleted.

If the element fixed for deletion is not a terminal node, it is replaced by its successor element. Actually, a copy of successor is taken. The successor is an element with higher value. The successor of any node which is not at lowest level, be a terminal node. Thus, deletion is nothing but removing of a particular element or record from a terminal node. While deleting the record the new node size is more

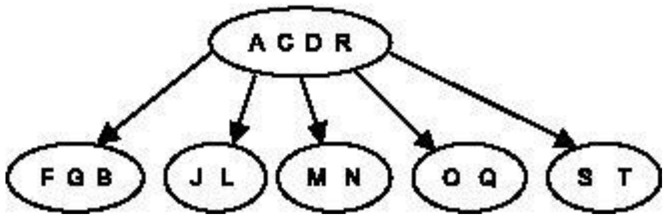
than minimum, i.e. the deletion is complete. If the node size is less than minimum, an underflow happens.
Rearrangement is done if either of adjacent siblings has more than the minimum elements (records). For rearrangement, the contents of the node (only those nodes having less than minimum records) along with sorting out records from parent node are gathered. The central record is written back to the parent and left and right halves are written back to two siblings.
Concatenation is applied if the node with less than minimum number of records has no adjacent sibling. The node is combined with its neighbouring sibling and element is moved from its parent.



1. Deleting K is straightforward because it is a leaf node.
2. Deleting E is not simple. Hence, its successor is moved up. E is moved down and deleted.



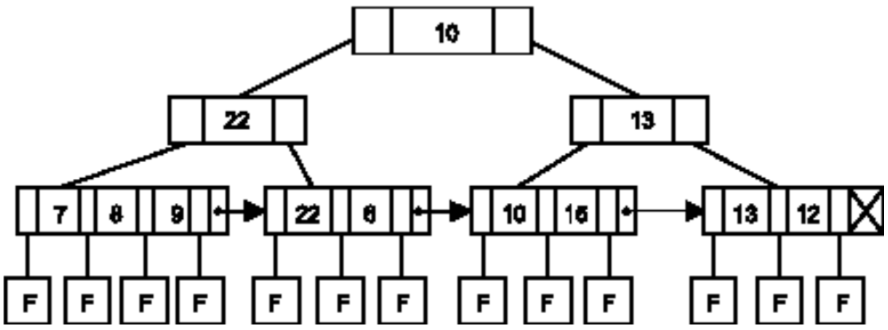
3. To delete P, the node has less than minimum numbers of keys. The sibling is carried. R moves up and Q moves down.
4. Deleting H, again node has less than minimum keys than required. The parent is left with only one key. Here, sibling cannot be applied. Hence, A, C, D and R form a new node.



B+ TREE

A B^+ tree can be obtained with slight modification of indexing in B tree. A B^+ tree stores key values repeatedly. Fig. 5.43 shows B^+ tree indexing.

Figure 5.43. B+ tree



By observing Fig. 5.43, we will come to know that the key values 10, 22 and 13 are stored repeatedly in the last level (terminal nodes). In these leaf nodes a link is maintained so that the traversing can be done from the leaf node at the extreme left to the leaf node at the extreme right. B tree and B^+ tree are same except the above differences.

Every leaf node of B^+ tree has two parts:

1. Index part It is the interior node stored repeatedly.
2. Sequence set It is a set of leaf nodes.

The interior nodes or keys can be accessed directly or sequentially. B^+ tree is useful data structure in-indexed sequential file organization.

Huffman Tree/ Encoding :-

Suppose that we have an algorithm of n symbols and a long message consisting of symbols from this alphabet. We wish to encode the message as a long bit string (a bit is either 0 or 1) by assigning a bit string code to each symbol of the alphabet and concatenating the individual codes of the symbols making up the message to produce an encoding for the message.

For example, suppose that the message is ABACCDA. Each of the letters B and D appears only once in the message, whereas the letter A appears three times. If a code is chosen so that the letter A is assigned a shorted bit string than the letters B and D, the length of the encoded message would be small. This is because the short code (representing the letter A) would appear more frequently than the long code. Indeed the code can be represented as follows :

Symbol	Code
A	0
B	110
C	10
D	111

Using this code, the message ABACCDA is encoded as 0110010101110, which requires only 13 bits. In very long messages containing symbols that appear very infrequently the savings are substantial. Ordinarily, codes are not constructed on the basis of frequency of characters within a single message alone, but on the basis of their frequency within a whole set of messages.

In our example decoding proceeds by scanning a bit from left to right. If a 0 is encountered as the first bit, the symbol is an A; otherwise it is a B,C, or D, and the next bit is examined. If the second bit is 0, the symbol is C; otherwise it must be a B or a D, and the third bit must be examined. If the third bit is 0, the symbol is a B; if it is a 1, the symbol is a D. As soon as the first symbol has been identified the process is repeated starting at the next bit to find the second symbol.

This suggests a method for developing an optimal encoding scheme, given the frequency of occurrence of each symbol in a message.

Construction of Huffman Tree :

Find the two symbols that appear least frequently. In our example, these are B and D. The last bit of their codes differentiates one from the other: 0 for B and 1 for D. combine these two symbols into the single symbol BD, whose code represents the knowledge that a symbol is either a B or d. The frequency of occurrence of this new symbol is the sum of the frequencies of its two constituent symbols. Thus the frequency of BD is 2.

There are now three symbols : A(frequency 3), C(frequency 2) and BD (frequency 2). Again choose the two symbols with smallest frequency : C and BD. The last bit of there codes again differentiates one from the other: 0 for C and 1 for BD. The two symbols are then combined into the single symbol CBD with frequency 4. there are now only two symbols remaining . A and CBD. These are combined into the single symbol ACBD. The last bits of the codes for A and CBD differentiate one from the other: 0 for A and 1 for CBD.

The symbol ACBD contains the entire alphabet; it is assigned the null bit string of length 0 as its code. At the start of the decoding, before any bits have been examined it is certain that any symbol is contained in ACBD. The two symbols that make up ACBD (A and CBD) are assigned the code 0 and 1, respectively. If a 0 is encounted the encoded symbol is in A if a 1 is encounted, it is a C or B or D. Similarly, the two symbols that constitute CBD (C abd BD) are assigned the codes 10 and 11, respectively.

The first bit indicates that symbol is one of the constituents of CBD and the second bit indicates whether it is a C or BD. The symbols that make up BD(B and D) are then assigned the codes 110 and 111. By this process, symbols that appear frequently in the message are assigned shorter codes than symbols that appear infrequently.

The action of combining two symbols into one suggests the use of a binary tree. Each node of the tree represents a symbol and each leaf represents a symbol of the original alphabet. Fig shows the binary tree constructed using the previous example. Each node contains a symbol and its frequency. Such trees are called Huffman Trees after the discoverer of this encoding method. Huffman tree is strictly binary.

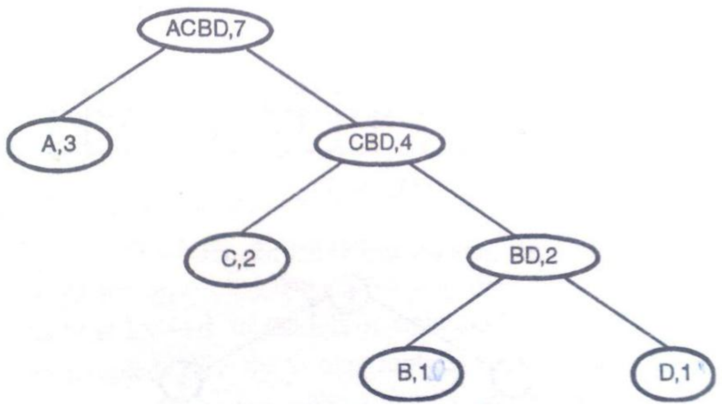


Fig. Huffman tree

Once the Huffman tree is constructed, the code of any symbol in the alphabet can be constructed by starting at the leaf representing that symbol and climbing up to the root. The code is initialized to null. Each time that a left branch is climbed, 0 is appended to the beginning of the code; each time that a right branch a climbed, 1 is appended to the beginning of the code.

BINARY TREE REPRESENTATION

Binary tree can be represented by two ways:

1. Array representation
2. Linked representation.

1. Array Representation of Binary Tree--

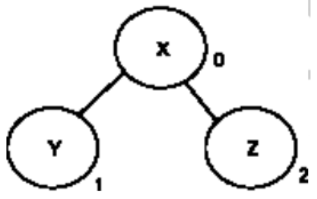
In any type of data structure array representation plays an important role. The nodes of trees can be stored in an array. The nodes can be accessed in sequence one after another. In array, element-counting starts from zero to (n-1) where n is the maximum number of nodes. In other words, the-numbering of binary tree nodes will start from 0 rather than 1.-

Assume an integer array. Its declaration is as follows:-

int array tree[n];

The root node of the tree always starts at index zero. Then, successive memory locations are used for storing left and right child nodes. Consider the following Fig. 5.14,-

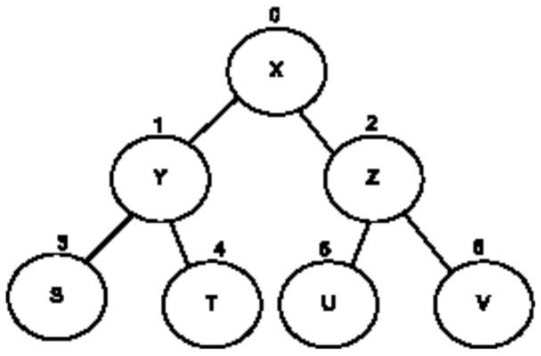
Figure 5.14. Array representation of tree



TREE[0]	*X*
TREE[1]	*Y*
TREE[2]	*Z*

In the above figure, X is the root and Y and Z are children. X is father of child Y and Z. Consider the following Fig. 5.15 with one more level.

Figure 5.15. Array representation of a tree



The array representation with one more level would be as follows:

0	X
1	Y
2	Z
3	S
4	T
5	U
6	v

It is very easy in this representation to identify the father, left and right child of an arbitrary node. For any node n, 0 ≤ n ≤ (MAXSIZE -1), the following can be used to identify the father and child nodes.

Father (n)

The location of the father node can be identified in a given tree by using the ((n-1)/2) where n is the index of child node, provided that n! =0 (not equal to zero). In case if n=0, the said node is root node. It has no father node. Consider the node 3 in Fig. 5.15 i.e. S. The father of node S is Y and the index of Y is 1. By substituting these values in the equation we identify the index of the father node.

Floor ((3-1)/2)) i.e. (2/2)=1

Lchild(n)

The left child of a node (n) is at position (2n+1).

1. Lchild(X)
   =lchild(0) =2 * 0+1
   =1
   The node with index 1 is Y.

2. lchild (Z) = lchild(2)
   = 2*2+1 = 5
   The node with index 5 is U.

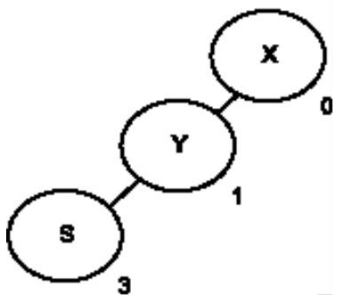
rchild (n)
The right child of node n can be recognized by (2n+2)

1. rchild (X)=rchild (0)
   = 2 * 0+2
   = 2

2. rchild (Y): rchild(1)
   = 2 *1+2=4
   The node with index 4 is T.

Siblings
In case the left child at index n is known then its right sibling is at (n+1). Likewise, if the right child at index n is known then is left sibling is at (n-1). The array representation is perfect for complete binary tree. However, this is not appropriate for other tree types and if we use array for their representation, it results in inefficient use of memory. Consider the binary trees shown in Figs 5.16(a) and 5.16(b).

Figure 5.16(a). Left skewed binary tree



The above is skewed binary tree. Here, every left sub-tree again represents left sub-tree. Such type of binary tree is called left skewed binary tree. Similarly, right skewed binary tree is also present [See Figs 5.16(a) and 5.16(b)]. Fig. 5.17 shows array representation of left and right skewed trees.

Figure 5.16(b). Right skewed binary tree

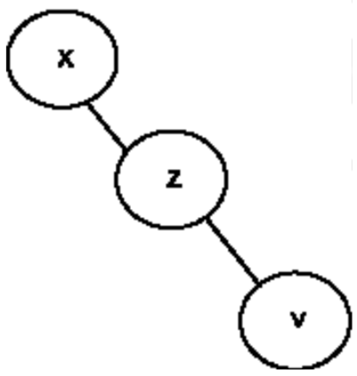
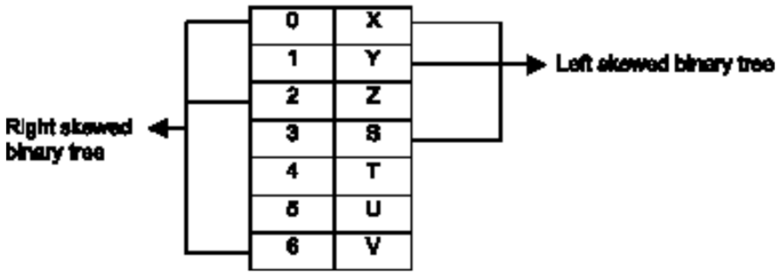


Figure 5.17. Left-right skewed binary tree

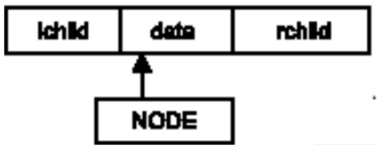


2.Linked Representation of Binary Tree

Another way of representation of binary tree is linked list, which is known to be more memory efficient than array representation. The fundamental component of binary tree is node. We know that the node consists of three fields, which is shown in Fig. 5.18.

- Data
- Left child
- Right child.

Figure 5.18. Link list representation of binary tree



We have already learnt about nodes in Chapter 6. The data field stores the given values. The lchild field is link field and holds the address of its left node. Similarly, the rchild holds the address of right node. The node can be logically represented as follows. Figs 5.19 and 5.20 show the linked list representation of binary tree.

```
struct node
{
  int data;
  struct node *rchild;
  struct node *lchild;
};
```

Figure 5.19. Binary tree

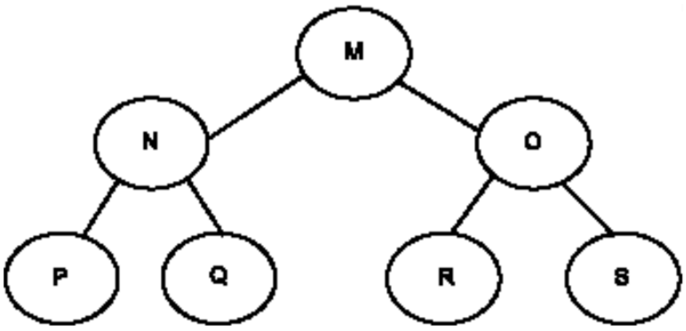
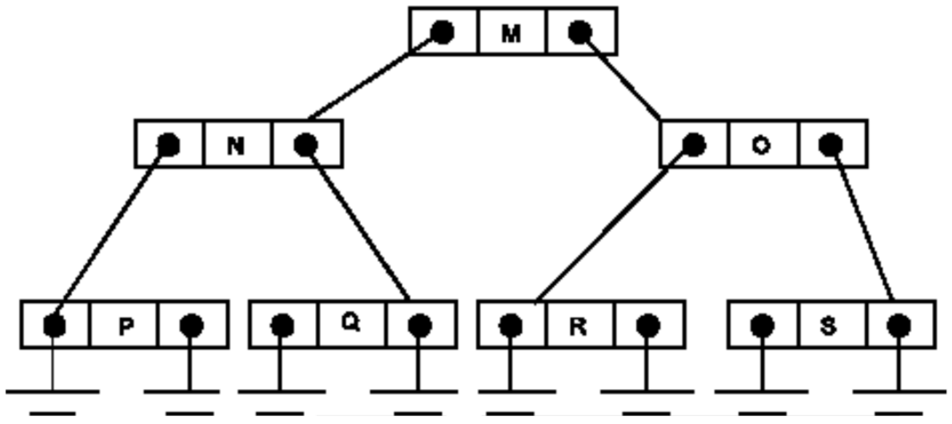


Figure 5.20. Linked list representation of binary tree



Binary tree has one root node and few non-terminal and terminal nodes. The terminal nodes are called leafs. The non-terminal nodes have their left and right child nodes. However, the terminal nodes are without child. While implementing this, fields of lchild and rchid are kept NULL. The non-terminal nodes are known as internal nodes and terminal nodes are known as external nodes.

OPERATION ON BINARY TREE-

Following fundamental operations are performed on binary tree:

Create

This operation creates an empty binary tree.

Make

This operation creates a new binary tree. The data field of this node holds some value.

Empty

When binary tree is empty, this function will return true else it will return false.-

Lchild

A pointer is returned to left child of the node. When the node is without left child, a NULL pointer is returned.

Rchild

A pointer is returned to right child and if the node is without right child a NULL pointer is returned.

Father

A pointer to father of the node is returned or else the NULL pointer is returned.

Sibling (Brother)-

A pointer to brother of the node is returned or else NULL pointer is returned.

Data

Value stored is returned.

In addition to above mentioned operations, following operations can also be performed on binary tree:-

1.Tree traversa

2.Insertion of nodes

3.Deletion of node

4.Searching for a given node

5.Copying the binary tree.

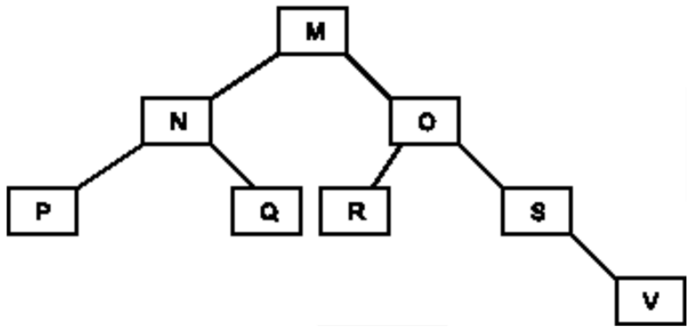
TRAVERSAL OF A BINARY TREE

Three parameters are needed for formation of binary tree. They are node, left and right sub-trees. Traversing is one of the most important operations done on binary tree and frequently this operation is carried on data structures. Traversal means passing through every node of the tree one by one. Every node is traversed only once. Assume, root is indicated by O, left sub-tree as L and right sub-tree as R. Then, the following traversal combinations are possible:

1. ORL - ROOT - RIGHT-LEFT
2. OLR - ROOT - LEFT-RIGHT
3. LOR - LEFT - ROOT- RIGHT
4. LRO - LEFT - RIGHT- ROOT
5. ROL- RIGHT - ROOT - LEFT
6. RLO- RIGHT - LEFT-ROOT

Out of six methods only three are standard and are discussed in this chapter. In traversing always right sub-tree is traversed after left sub-tree. Hence, the OLR is preorder, LOR is inorder and LRO is postorder.

Figure 5.21. A model tree

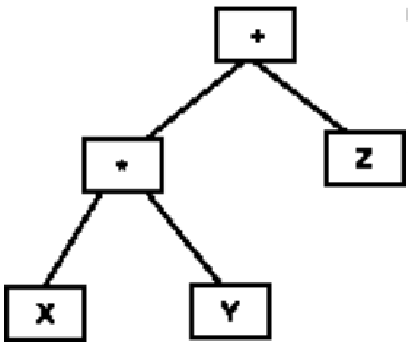


The inorder representation of above tree is P-N-Q-M-R-O-S-V.

Traversing is a common operation on binary tree. The binary tree can be used to represent an arithmetic expression. Here, divide and conquer technique is used to convert an expression into a binary tree. The procedure to implement it is as follows.

The expression for which the following tree has been drawn is (X*Y)+Z. Fig. 5.22 represents the expression.

Figure 5.22. An arithmetic expression in binary tree form



Using the following method,the traversing operation can be performed. They are:

1. Preorder traversal

2. Inorder traversal
3. Postorder traversal.

All the above three types of traversing methods are explained below.

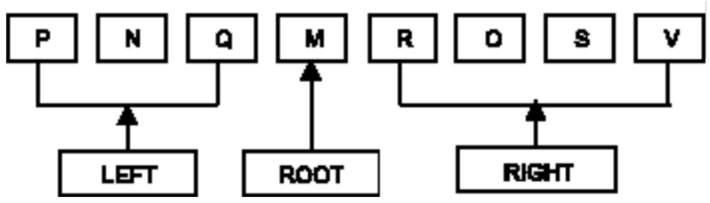
5.9.1 Inorder Traversal

The functioning of inorder traversal of a non-empty binary tree is as follows:

1. Firstly, traverse the left sub-tree in inorder.
2. Next, visit the root node.
3. At last, traverse the right sub-tree in inorder.

In the inorder traversal firstly the left sub-tree is traversed recursively in inorder. Then the root node is traversed. After visiting the root node, the right sub-tree is traversed recursively in inorder. Fig. 5.21 illustrates the binary tree with inorder traversal. The inorder traversing for the tree is P-N-Q-M-R-O-S-V. It can be illustrated as per Fig. 5.23.

Figure 5.23. Inorder traversal



The left part constitutes P, N, and Q as the left sub-tree of root and R, O, S, and V are the right sub-tree.

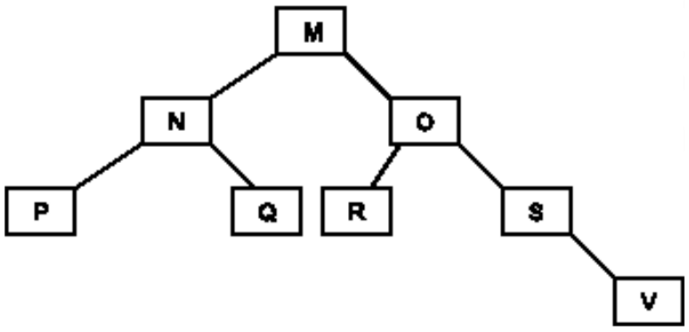
Preorder Traversal

The node is visited before the sub-trees. The following is the procedure for preorder traversal of non empty binary tree.

1. Firstly, visit the root node (N).
2. Then, traverse the left sub-tree in preorder (L).
3. At last, traverse the right sub-tree in preorder R.

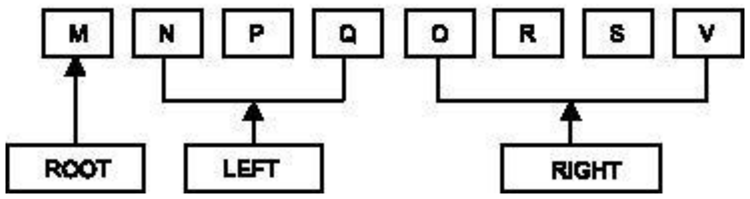
The preorder is recursive in operation. In this type, first root node is visited and later its left and right sub-trees. Consider the following Fig. 5.25 for preorder traversing.

Figure 5.25. Tree for preorder traversal



The preorder traversing for Fig. 5.25 is M, N, P, Q, O, R, S, and V. This can also be shown in Fig. 5.26. In this traversing the root comes first and the left sub-tree and right sub-tree at last.

Figure 5.26. Preorder traversal



In the preorder the left sub-tree appears as N, P, and Q and right sub-tree appears as O,R,S, and V.

Postorder Traversal

In the postorder traversal the traversing is done firstly left and right sub-trees before visiting the root. The postorder traversal of non-empty binary tree is to be implemented as follows:

1. Firstly, traverse the left sub-tree in postorder style.
2. Then, traverse the right sub tree in postorder.
3. At last, visit the root node (N).

In this type, the left and right sub-trees are processed recursively. The left sub-tree is traversed first in postorder. After this, the right sub-tree is traversed in post order. At last, the data of the root node is shown. Fig. 5.28 shows the postorder traversal of a binary tree.

The postorder traversing for the above Fig. 5.28 is P, Q, N, R, V, S, O, and M. This can also be shown as in Fig. 5.29. In this traversing the left sub-tree is traversed first, then right sub-tree and at last root. In the postorder the left sub-tree is P, Q and N and the right sub-tree is R, V, S and O.

Figure 5.28. Tree for post order

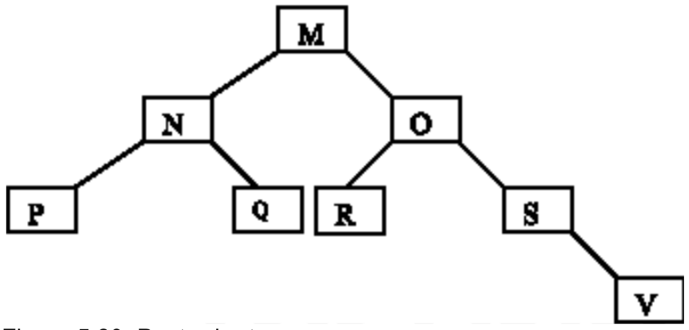
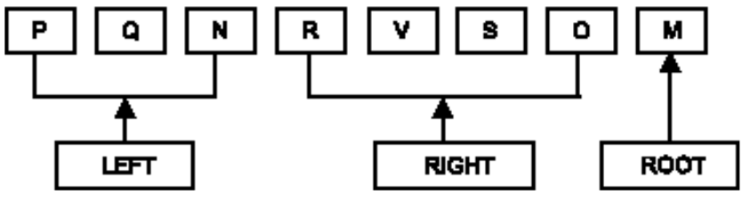


Figure 5.29. Postorder traverse



CHAPTER 6
GRAPH

INTRODUCTION

Graphs are frequently used in every walk of life. Every day we come across various kinds of graphs appearing in newspapers or television. The countries in a globe are seen in a map. A map depends on the geographic location of the places or cities. As such, a map is a well-known example of a graph. In the map, various connections are shown between the cities. The cities are connected via roads, rail or aerial network. How to reach a place is indicated by means of a graph. Using the various types of the links the maps can be shown.

Fig. 6.1 illustrates a graph that contains the cities of India connected by means of road. Assume that the graph is the interconnection of cities by roads. As per the graph, Mumbai, Hyderabad and Kolkata are directly connected to all the other cities by road. Delhi is directly connected to Mumbai, Hyderabad, and Kolkata. Delhi is connected to Chennai via Hyderabad.

Generally, we provide the address of our office/residence to a stranger who is not aware of our address and location of city. At this juncture we use the graph for the easiest representation of our residential location. Fig. 6.2 shows the location of place by graph. By using the graph any stranger can easily find location. For example, a pilgrim wishes to reach the Gurudwara in Nanded. The path is shown in the figure for reaching the Gurudwara. The devotee has to reach the destination via Shivaji statue, Mahatma Gandhi statue and then Gurudwara as per the graph. Once a map is provided, any stranger can reach any destination by using appropriate conveyance.

Figure 6.1. Map representation of connection of cities

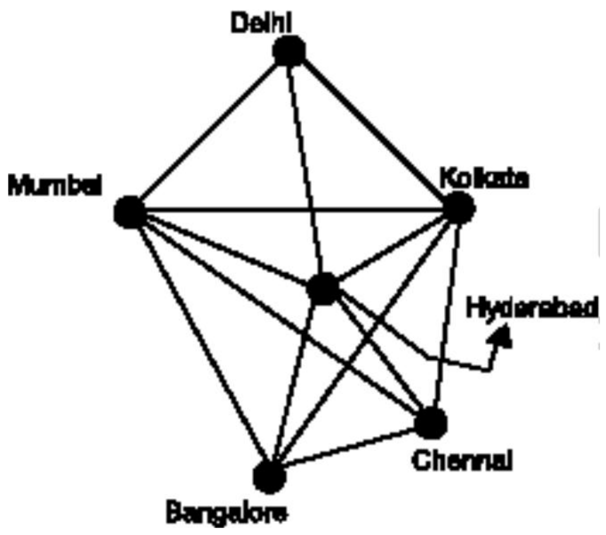
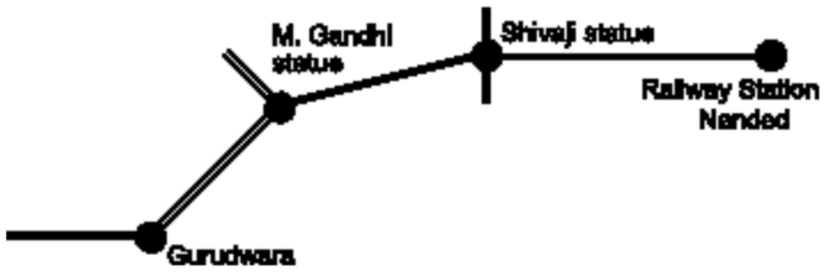


Figure 6.2. Representation of a map



In Fig. 6.2, four places are connected by road links. In the graph the road links are called as edges and the places are called as vertices. The graph is a collection of the vertices and the edges, hence a map is treated as graph. The following section describes the graph and relevant theories.

Like tree, graphs are nonlinear data structures. Tree nodes are arranged hierarchically from top to bottom like a parent is placed at the top and child as successor at the lower level. Tree has some specific structure whereas graph does not have a specific structure. It varies from application to application.

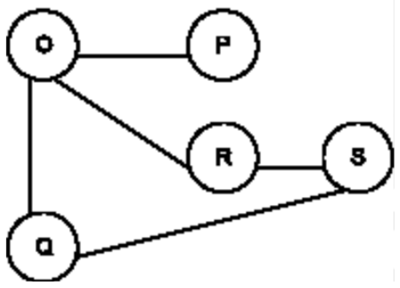
GRAPH

A graph is set of nodes and arcs. The nodes are also termed as vertices and arcs are termed as edges. The set of nodes as per the Fig. 6.3 is denoted as {O, P, R, S, Q}. The set of arcs in the following graph are {(O,P), (O,R), (O,Q), (R,S), (Q,S)}. Graph can be represented as,

G= (V,E) and V(G)= (O, Q, P, R, S) or group of vertices.

Similarly, E(G)= ((O,P), (O,R), (O,Q),(R,S),(Q,S)) or group of edges.

Figure 6.3. Graph



A graph is linked if there is pathway between any two nodes of the graph, such a graph is called connected graph or else, it is non-connected graph. Fig. 6.4 and 6.5 are connected and non-connected graphs, respectively. In both the figures four nodes are depicted. In the latter all the nodes are not connected by links whereas in the former case all the nodes are joined by paths or links.

Figure 6.4. Connected graph

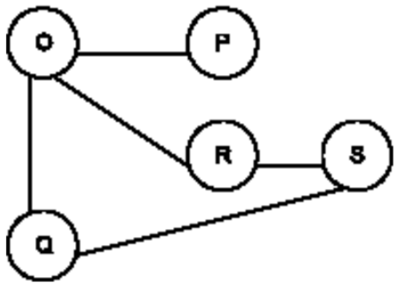
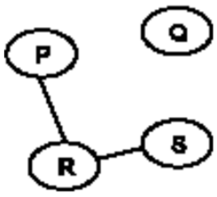


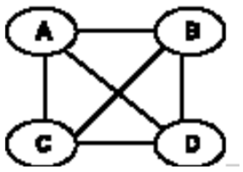
Figure 6.5. Non-connected graph



Undirected Graph

A graph containing unordered pair of nodes is termed as undirected graph.
The vertices are {A, B, C, D} and edges are {(A, B), (A, D), (A, C), (B, D), (B, C), (D, C)}. The graph has four nodes and six edges. This type of graph is known as completely connected network, in which every node is having out going path to all nodes in the network. For a complete network the number of links =N (N-1)/2, where N is the number of vertices or nodes. In the Fig. 6.6 N is 4. By substituting its value the number of edges obtained will be equal to 6.

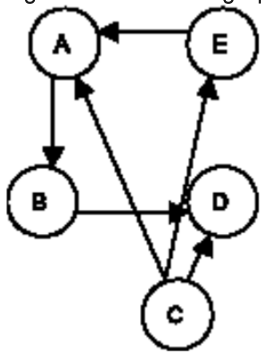
Figure 6.6. Undirected graph



Directed Graph

This kind of graph contains ordered pairs of vertices. For example, graph vertices are {A,B,C,D,E} and edges are {(A,B),(B,D),(C,D)(C,A), (C,E), (E, A) }. Fig. 6.7 represents a graph having five nodes and six edges.
A direction is associated with each edge. The directed graph is also known as digraph.

Figure 6.7. Directed graph



V(G) = {A, B, C, D, E} and
group of directed edges = {(A,B), (B,D), (C,D)(C,A), (C,E), (E, A)}.

TERMINOLOGIES OF GRAPH

Weighted Graph

A graph is supposed to be weighted if its every edge is assigned some value which is greater than or equal to zero, i.e. non-negative value. The value is equal to the length between two vertices. Weighted graph is also called as network. Weighted graph is shown in Fig. 6.8.

Adjacent Nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

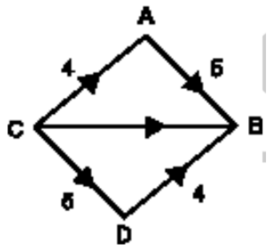
Incidence

In an undirected graph the edges v0 , v1 is incident on nodes. In a direct graph the edge v0, v1 is incident from node v0. It is incident to node v1.

Path

A path from edges u_0 to node u_n is a sequence of nodes u_0, u_1,u_2, u_3.. .u_{n-1}, u_n. Here, u_0 is adjacent to u_1, u_1 is adjacent to u_2 and u_{n-1} is adjacent to u_n.

Figure 6.8. A weighted graph



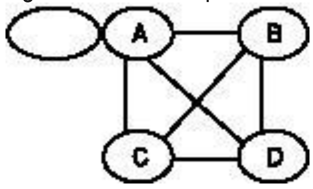
Length of Path

Length of path is nothing but total number of edges included in the path from source node to destination node.

Closed Path

When first and last nodes of the path are same, such path is known as closed path. In Fig. 6.9 closed path at node A is shown.

Figure 6.6. Closed path for node A



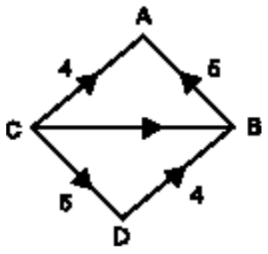
Simple Path

In this path all the nodes are different with an exception that the first and last nodes of the path can be similar.

Cycle

Cycle is a simple path. The first and last nodes are same. In other words, a closed simple path is a cycle. In a digraph a path is known as cycle if it has one or more nodes. The starting node is connected to the last node. In an undirected graph a path is called cycle if it has at least three nodes. The starting node is connected to last node. In the following figure path ACDBA is a closed path. Example of a cycle is shown in Fig. 6.10.

Figure 6.10. Example of a cycle



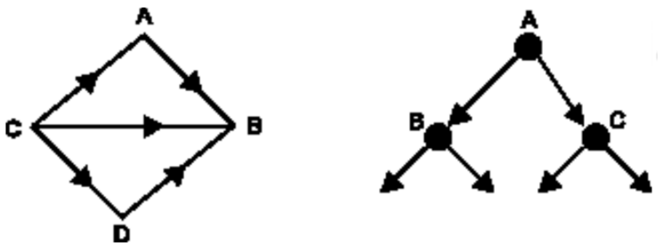
Cycle Graph

A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle. This is same as closed path shown in Fig. 6.10.

Acyclic Graph

A graph without cycle is called acyclic graph. Examples of acyclic graphs are shown in Fig. 6.11.

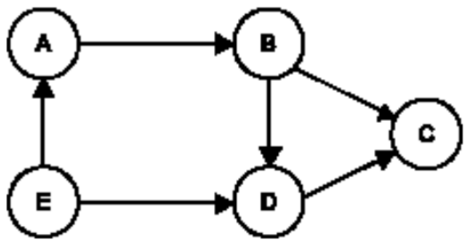
Figure 6.11. Acyclic graphs



Dag

A directed acyclic graph is called dag after its acronym (reduction). Fig. 6.12 is a graph showing the dag.

Figure 6.12. Dag



Degree

In an undirected graph, the total number of edges linked to a node is called degree of that node. In a digraph there are two degrees for every node called indegree and outdegree. In the above Fig. 6.12, E has two edges hence degree is 2. Similarly, D has degree three and so on.

Indegree

The indegree of a node is the total number of edges coming to that node. In Fig. 6.12, C is receiving two edges hence, the indegree is two.

Outdegree:

The outdegree of a node is the total number of edges going outside from that node. In the above Fig. 6.12 the outdegree of D is one.

Source

A node, which has only outgoing edges and no incoming edges, is called a source. The indegree of source is zero. In Fig. 6.12 the node E is the source since it does not have any incoming edges. It has only the outgoing edges.

Sink

A node having only incoming edges and no outgoing edges is called sink node. Node C in Fig. 6.12 is a sink node because it has only incoming edges but no outgoing edges.

Pendant Node

When indegree of node is one and outdegree is zero then such a node is called pendant node.

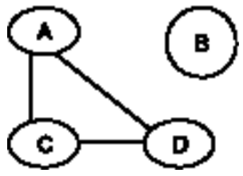
Reachable

If a path exists between two nodes it will be called reachable from one node to other node.

Isolated Node

When degree of node is zero, i.e. node is not connected with any other node then it is called isolated node. In Fig. 6.13 B node is the isolated node.

Figure 6.13. Isolated node

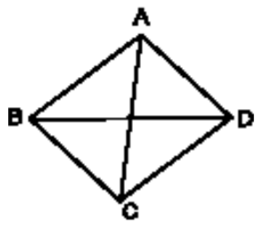


Successor and Predecessor

In digraph if a node V_0 is adjacent to node V_1 then V_0 is the predecessor of V_1 and V_1 is the successor of V_0.

Complete Graph

The graph in which any V_0 node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges= n(n-1)/2. The following figure shows the complete graph. The 'n' is the number of vertices present in the graph.



Articulation Point

On removing the node the graph gets disconnected, then that node is called the articulation point.

Biconnected Graph

The biconnected graph is the graph which does not contain any articulation point.

Multigraph

A graph in which more than one edge is used to join the vertices is called multigraph. Edges of multigraph are parallel to each other.

Figure 6.14. Multigraph

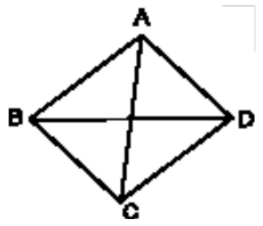


Fig. 6.14 shows the multigraph in which one can see the parallel edges between A and D, D and C, B and C, and A and B.

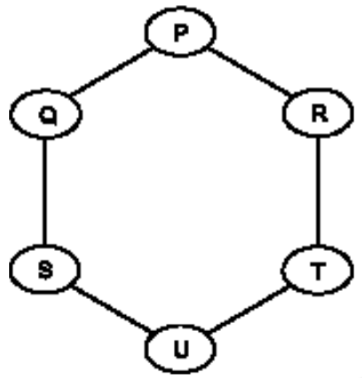
Regular Graph

Regular graph is the graph in which nodes are adjacent to each other i.e. each node is accessible from any other node.

GRAPH REPRESENTATION

The graph can be implemented by linked list or array. Fig. 6.15 illustrates a graph and its representation and implementation is also described.

Figure 6.15. Modle graph

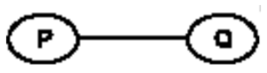


Different possibilities of graph representations are dependent on two cases:

1. If there is no edge between two nodes.

   There is no edge between nodes P and Q.

2. If there is an edge between any two nodes.



The nodes P and Q are having an edge.
Hence, following Table 6.1 provides the representation of the graph (Fig. 6.15).

Table 9.1. Representation of a graph

Nodes	*P*	*Q*	*R*	*S*	*T*	*U*
P	x	✓	✓	x	x	x
Q	✓	x	x	✓	x	x
R	✓	x	x	x	✓	✓
S	x	✓	x	x	x	✓
T	x	x	✓	x	x	✓
U	x	x	x	✓	✓	x

As per the Table 6.1, there is an edge in between the nodes P and Q, P and R, and there is no edge between nodes P and S. The symbol indicates existence of edge and indicates absence of edge between two nodes.

From the above table one can predict the path to reach a particular node. For example, initial node is P and the destination node is U. We have to find the path to reach node U from P.

There is no edge between P and U. Then, find out the edge for nearest node in forward direction. By observing, we know there are two edges from P to Q and P to R. We can select either Q or R. Suppose, we have selected node Q, again find out next nearest successive node to Q by observing column Q. The next successive forward node will be S. Then, refer column S and it provides two edges Q and U. The node U is our solution. Thus, by using the above table, paths between any two nodes can be determined. The path should be P-Q-S-U. The graph can be represented by sequential representation and linked list representation.

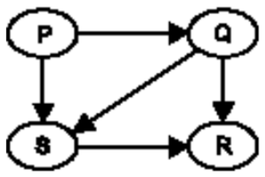
Adjacency Matrix

The matrix can be used to represent the graph. The information of adjacent nodes will be stored in the matrix. Presence of edges from a particular node can be determined easily. The matrix can be represented by two-dimensional array. In a two-dimensional array [][], the first sub-script indicates row and second, column. For example, there are five nodes in the graph then the 0th row indicates node1 and so on. Likewise, column represents node1, node2, and so on. For example, consider a two-dimensional array.

Nodes[j[k]

1 indicates presence of edge between two nodes j and k.
0 indicates absence of an edge between two nodes j and k.
Thus, the matrix will contain only 0 and 1 values.

Figure 6.16. An example of graph

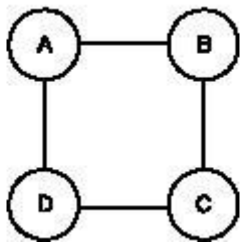


The matrix for the graph given in Fig. 6.16 would be

$$\text{Matrix } X = \begin{array}{c|cccc} & P & Q & R & S \\ \hline P & 0 & 1 & 0 & 1 \\ Q & 0 & 0 & 1 & 1 \\ R & 0 & 0 & 0 & 0 \\ S & 0 & 0 & 1 & 0 \end{array}$$

In the above matrix ,Mat[0][1]=1, which represents an edge between node P and Q. Entry of one in the matrix indicates that there is an edge and 0 for no edge. Thus, adjacency is maintained in the matrix X. One can also represent the undirected graph with adjacency matrix. Fig. 6.17 is an undirected graph.

Figure 6.17. Undirected graph



The adjacency matrix for the above graph would be as follows:

$$\text{Matrix } X = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 1 \\ D & 1 & 0 & 1 & 0 \end{array}$$

The above matrix is symmetric since x[i][j]= x[j][i].
In undirected graph the sum of row elements and column elements is the same. The sum represents the degree of the node. In this matrix the sum of any row or any column is 2, which is nothing but the degree of each node is 2.
We can also represent in the same way a weighted graph with adjacency matrix. The contents of the matrix will not be only 0 and 1 but the value is substituted with the corresponding weight. For a null graph, that contains n vertices but no edges, all the elements of such null graph in an adjacency matrix are 0.

Figure 6.18. Null matrix

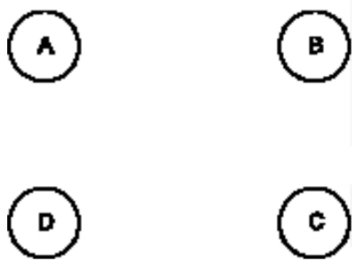


Fig. 6.18 represents the null graph and the adjacency matrix is as follows:

$$X = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 \end{array}$$

Adjacency List

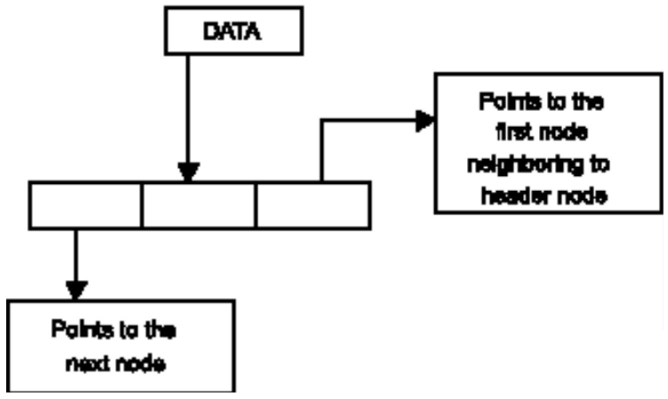
Two lists are maintained for adjacency of list. The first list is used to store information of all the nodes. The second list is used to store information of adjacent nodes for each node of a graph.
In case a graph comprises of N nodes, then two lists are to be prepared.

1. First list keeps the track of all the N nodes of a graph.

2. The second list is used to keep the information of adjacent nodes of each and every node of a graph. As such there will be N lists that would keep the information of adjacent nodes.

Header node is used to represent each list, which is assumed to be the first node of a list. Fig. 6.19 represents a header node.

Figure 6.19. Header node

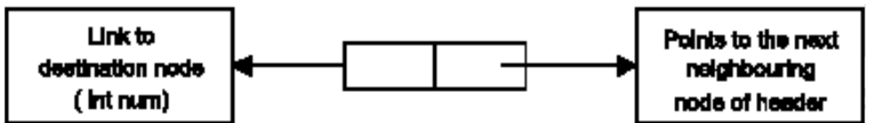


```
struct
node {
  struct *  next
  int num;
  struct edge *aj;
};
```

Structure of an Edge

Fig. 6.20 is the representation of an edge.

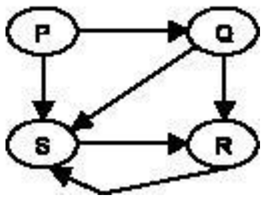
Figure 6.20. An edge of graph


Link to destination node (int num)
Points to the next neighbouring node of header

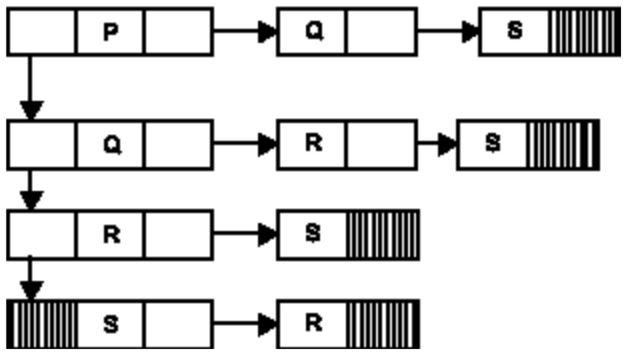
```
struct edge
{
  int num;
  struct edge *ptr;
};
```

Consider a graph cited in Fig. 6.21.

Figure 6.21. An example of graph



Adjacency list for the above graph would be given in Fig. 6.22.
Figure 6.22. Adjacency list



TRAVERSAL IN GRAPH-

Traversing in a graph is nothing but visiting each and every node of a graph. The following points are to be noted in a graph:

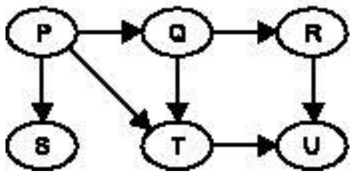
1. The graph has no first node or root. Therefore, the graph can be started from any node.
2. In graph, only those nodes are traversed which are accessible from the current node. For complete traversing of graph, the path can be determined by traversing nodes step by step.
3. In the graph, the particular node can be visited repeatedly. Hence, it is necessary to keep the track of the status of every node whether traversed or not.
4. In graph to reach a particular node, more paths are available.

Two techniques are used for traversing nodes in a graph. They are depth first and breadth first. These techniques have been elaborated in detail in the following sections.

Breadth First Search

This is one of the popular methods of traversing graph. This method uses the queue data structure for traversing nodes of the graph. Any node of the graph can act as a beginning node. Using any node as starting node, all other nodes of the graph are traversed. To shun repeated visit to the same node an array is maintained which keeps status of visited node.

Figure 6.26. A model graph



Take the node P of Fig. 6.26 as a beginning node. Initially, the node P is traversed. After this, all the adjacent nodes of P are traversed, i.e. Q, T and S. The traversal of nodes can be carried in any sequence. For example, the sequence of traverse of nodes is Q, S and T. The traversal will be
P Q S T

First, all the nodes neighbouring Q are traversed, then neighbouring nodes of S and finally T are taken into account. The adjacent node of Q is R and T is U. Similarly, the adjacent node of T is U and S does not have any adjacent node. Hence, in this step the traversal now should be in the following way:
P Q S T R U

Now, the new nodes obtained are R and U after traversing. The new adjacent node of R is U and U node does not have any adjacent node. Node U has been visited in the previous case hence it must be ignored in this step.

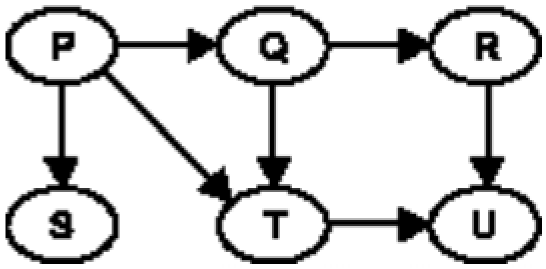
Algorithm

1. Put the root node on the queue.
2. Pull a node from the beginning of the queue and examine it.
   - If the searched element is found in this node, quit the search and return a result.
   - Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
3. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
4. Repeat from Step 2.

6.6.2 Depth First Search

In this method, also a node from graph is taken as a starting node. Traverse through all the possible paths of the starting node. When the last node of the graph is obtained and path is not available from the node; then control returns to previous node. This process is implemented using stack. Consider the following graph shown in Fig. 6.27

Figure 6.27. A model graph



Consider, P as starting node. Then, traverse the node adjacent to P and we will get Q and then R (adjacent to Q) and U (adjacent to R). The traversal will be
P Q R U

The search is always carried in forward direction. After reaching to U, we reach the end of the path and further movement in forward direction is not possible. Hence, the controls go to the previous node and again traverse through the available paths for non-traversed nodes.

In reverse direction, we get the node R and it has unvisited node. Hence, Q is taken and it gives T. The node T gives U, but it is already visited. Therefore, control in reverse direction checks all the nodes. It takes P and it gives node S. The sequence of traversal will be
P Q R U T S

Algorithm:-

1. Set the starting point of traversal and push it inside the stack.
2. Pop the stack and add the popped vertex to the list of visited vertices.
3. Find the adjacent vertices for the most recently visited vertex(from the Adjacency Matrix).
4. Push these adjacent vertices inside the stack (in the increasing order of their depth) if they are not visited and not there in the stack already.
5. Step-5: Go to step-2 if the stack is not empty.