MODULE - V

Tree Graphs and Their Applications

MODULE 5

Tree Graphs and Their Applications

Module Description

Trees and graphs are typical examples of non-linear data structure as discussed in Module 1. Non-linear data structure unlike linear data structure, is a structure where in an element is permitted to have any number of adjacent elements.

Trees are non-linear data structures which are very useful for representing hierarchical relationships among the data elements. ***For example,*** for representation of relationship between members of the family we can uses non-linear data structures like trees. Data organization in hierarchical forms or structures is very essential for many applications involving searching of data elements. Trees are most important and useful data structures in Computer Science in the areas of data parting, compiler designs, expression evaluation and managing storages.

Similar to trees, graph is a powerful tool used for representing a physical problem in mathematical form. One of the famous problems where graphs are used is for finding optimum shortest path for a travelling salesman while travelling from one city to another city, so as to minimize the cost. A graph may also have unconnected node. Moreover, there may be more than one path between two nodes. Graphs and directed graphs are powerful tools used in computer science for many real world applications. ***For example,*** building compilers and also in modelling physical communication networks. A graph is an abstract notion of a set of nodes (vertices or points) and connection relations (edges or arcs) between them.

Chapter Table of Contents

Chapter 5.1

Tree Fundamentals

Aim

To equip the students with the techniques of Trees so that it can be used in the program to search and sort the elements in the list

Instructional Objectives

After completing this chapter, you should be able to:

- Explain tree and its different types
- Outline various tree terminologies with example
- Illustrate max heap and min heap technique
- Demonstrate binary search tree with an example
- Illustrate preorder, postorder and inorder

Learning Outcomes

At the end of this chapter, you are expected to:

- Discuss tree and various types with example
- Write a code to demonstrate max heap and min heap
- Explain array representation of binary tree
- Outline the steps to traverse inorder, preorder and postorder
- Construct binary tree from inorder, preorder and postorder

Introduction

So far in the previous chapters, we have come across linear data structures like stack, queues and linked lists. This chapter focuses on non-linear data structures such as trees. In this chapter, we will come across the basic structure of trees, its types and its various terminologies. We will then focus on heaps, its types, then the concept of binary search tress and its various types of traversals.

5.1.1 Definition of Tree

Tree can be defined as a non-linear data structure consisting of a root node and other nodes present at different levels forming a hierarchy. A tree essentially has one node called its root and one or more nodes adjacent below, connected to it. A tree with no nodes is called an empty tree.

Therefore, a tree is a finite set of one or more nodes such that:

- There is specially designated node called a root.
- The remaining nodes are partitioned into n>=0 disjoint set T1, T2,……., Tn where each of these sets is a tree. T1, T2… Tn are called sub-trees of the root.

A node in the definition of a tree depicts an item of information and the links between the nodes are called as its branches which represents an association between these items of information.

The figure 5.1.1 below shows pictorial representation of a Tree.

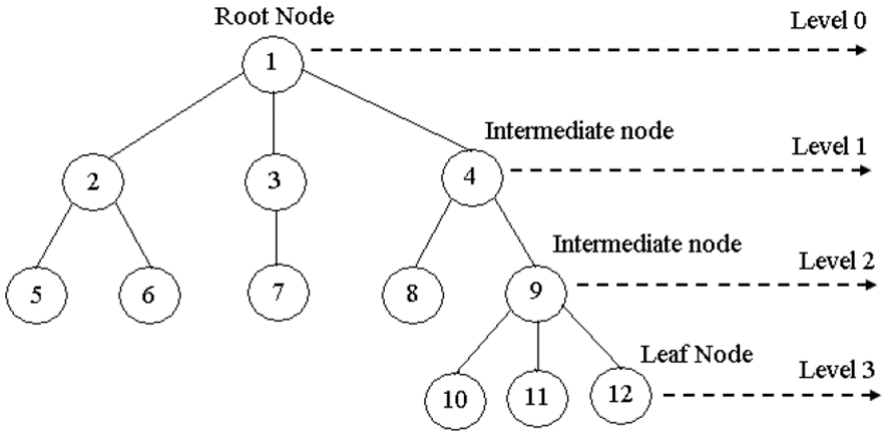


Figure 5.1.1: Pictorial Representation of a Tree

In the above figure, node 1 is the root of the tree, nodes 2, 3, 4 and 9 are called intermediate nodes and nodes 5, 6, 7, 8, 10, 11 and 12 are its leaf nodes. It is important to note that the tree emphasizes on the aspect of (i) connectedness and (ii) absence of loops or cycles. Starting from the root, the tree structure allows connectivity of the root to each of the node in the tree. Generally, any node can be reached from any part of the tree. Moreover, with all the branches providing links between the nodes, the tree structure makes the point that there are no sets of nodes forming a closed loop or cycle.

Tree data structure is widely used in the field of computer science. The following points shows its use in many applications, they are

- Folder/directory structure in operating systems like windows and linux.
- Network routing
- Syntax tree used in compilers, etc.

Example:

The figure 5.1.2 below shows directory structure in Windows OS.

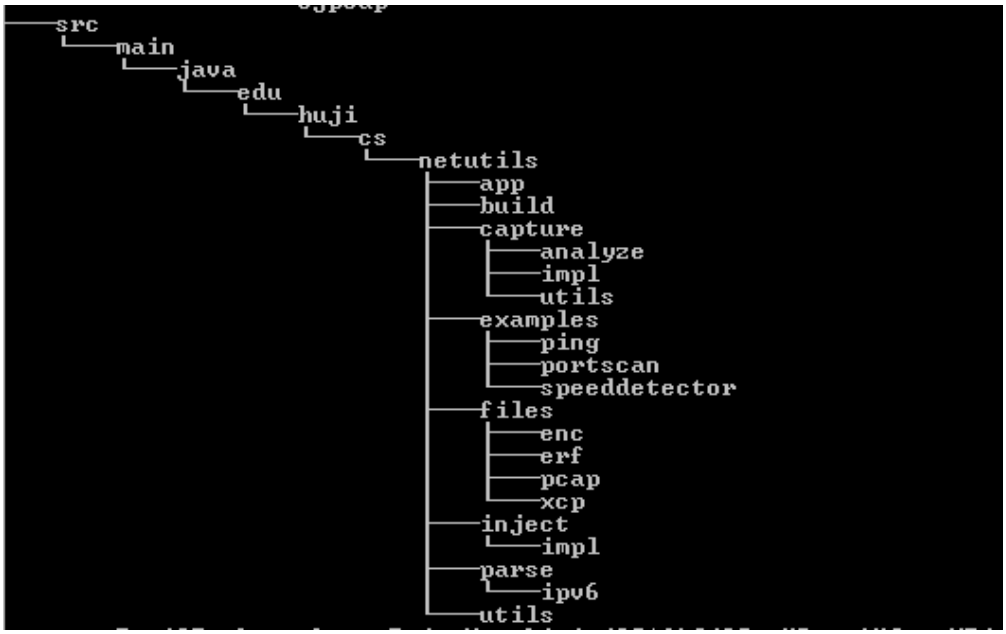


Figure 5.1.2: Directory Structure in Windows OS.

The figure 5.1.3 below shows directory structure in Linux OS.

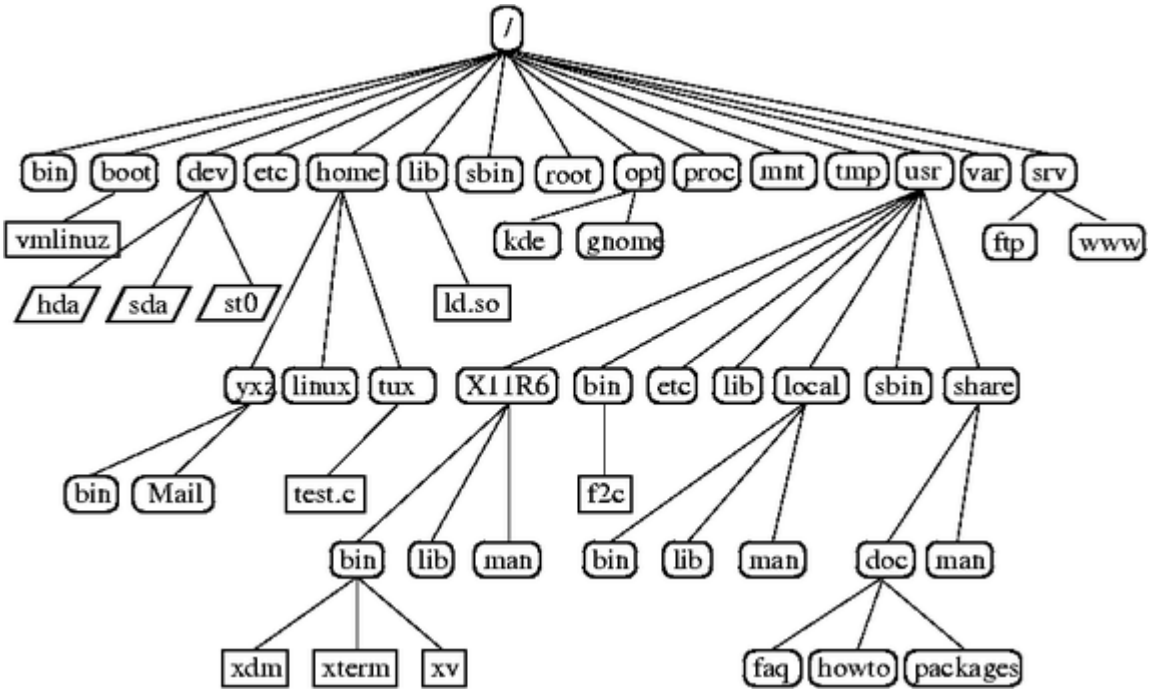


Figure 5.1.3: Directory Structure in Linux OS.

Advantages of trees

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub-trees around with minimum efforts

Did you know?

Portable Document Format (PDF) is a tree based format. It has a root node followed by a catalog node (these are often the same) followed by a pages node which has several child page nodes. Producers/consumers often use a balanced tree implementation to store a document in memory.

Self-assessment Questions

1) A Tree is _____ type of data structure

   a) Linear
   b) Advanced
   c) Non-Linear
   d) Data Driven

2) The tree with no nodes is called an empty tree

   a) True
   b) False

3) Nodes in a tree can be infinite.

   a) True
   b) False

5.1.2 Tree Terminologies

Terminologies are ways of giving names to any part of data structure. Therefore this section covers the various terminologies that are used to define and help us to identify the components of the tree. This will make us easy to analyze and solve various complex mathematical problems. Some of the very important terminologies are root, node, degree of a node or a tree, terminal nodes, siblings, levels, edge, path, depth, parent node and ancestral of the node.

(i) Root

Definition: In trees, the origin node or the first node is called a root.

This is the node from which all the other nodes gets evolved and sometimes referred to as a seed node. In any given tree there is at least one root node and the entire structure of the tree is built on this node.

The figure 5.1.4 shows the root node in a tree data structure.

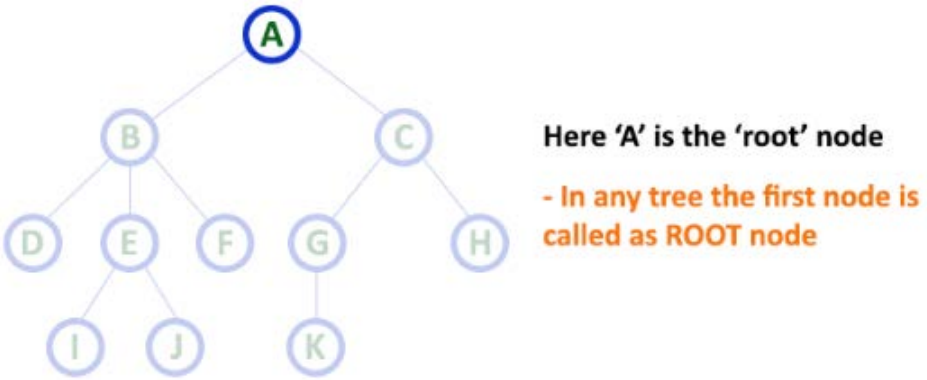


Figure 5.1.4: Root Node Representation

(ii) Node

Definition: Node is a data point which forms a unit in a tree data structure.

In real time applications, node can be a computer system connected to network which is in turn connected to a computer server. Different nodes connect together in specific structure to form a tree data structure.

(iii) Degree of a Node or a Tree

Definition: Degree of a node is the total number of child node connected to a particular node. The highest degree of a node among all the nodes is called 'Degree of a Tree'.

The degree of a node varies as at times a node many be connected to more than two nodes. However, in case of a binary tree, degree is always 2.

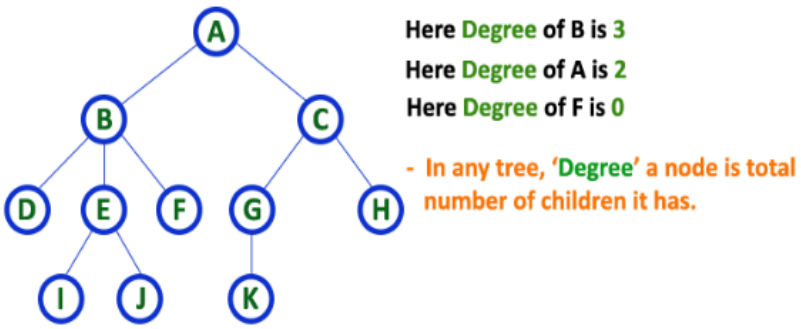


Figure 5.1.5: Degree of a Node

(iv) Terminal Node

Definition: A node having no child node is called as a terminal node or a leaf node.

These nodes are also called as External nodes. These are the nodes having no further connectivity and form leafs of the tree. Figure 5.1.6 below shows leaf nodes.

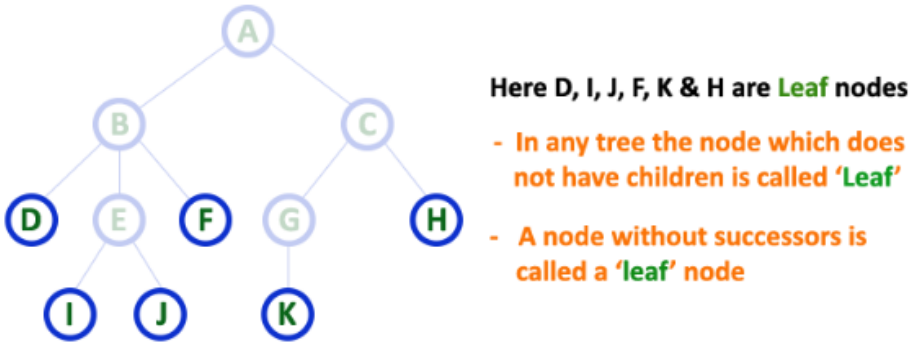


Figure 5.1.6: Leaf Nodes

(v) Non-terminal Nodes

Definition: All the intermediate nodes which have at least one child node are called non-terminal nodes or internal nodes.

All the nodes which are non-terminal nodes are non-terminal nodes. They can have degree greater than zero. Figure 5.1.7 below shows internal or non-terminal nodes.

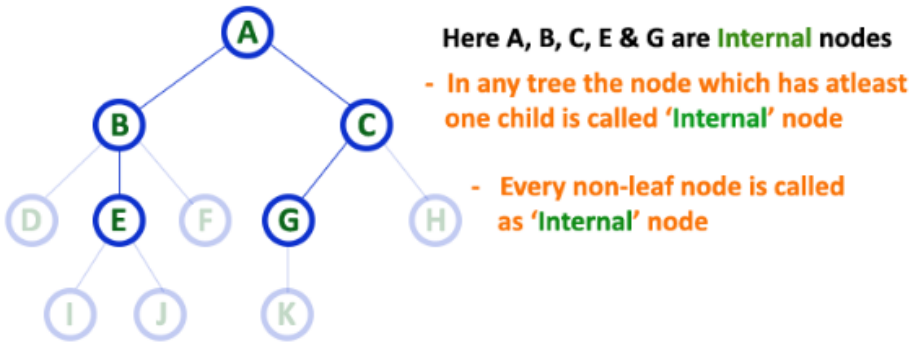


Figure 5.1.7: Internal or Non-Terminal Nodes.

(vi) Siblings

Definitions: Nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.

The following Figure 5.1.8 shows sibling nodes in a tree.

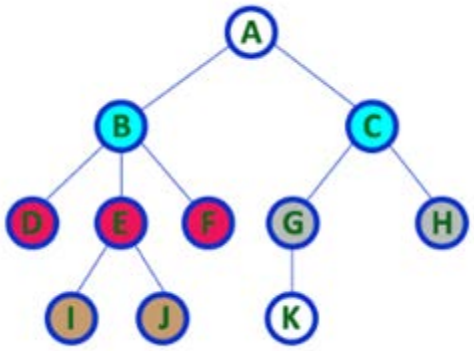


Figure 5.1.8: Sibling Nodes

Here B & C are siblings

Here D E & F are siblings

Here G & H are siblings

Here I & J are siblings

In any tree, the nodes which has same parent are called 'Siblings'.

The children of a parent are called 'Siblings'.

(vii) Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

Figure 5.1.9 below shows levels in a tree.

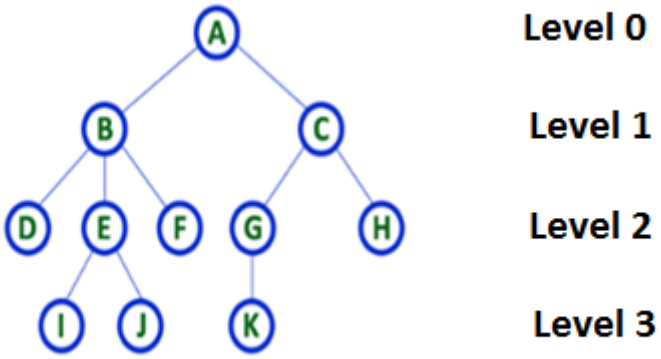


Figure 5.1.9: Levels in the Tree

(viii) Edge

Definition: Connecting link between two nodes is called an Edge.

An edge is basically representation of links which help us understand which node is connected to what other node in the tree. It also helps us determine if the node is not a leaf node. In a tree with N number of nodes there are maximum of N-1 numbers of edges. Figure 5.1.10 below shows representation of edge.

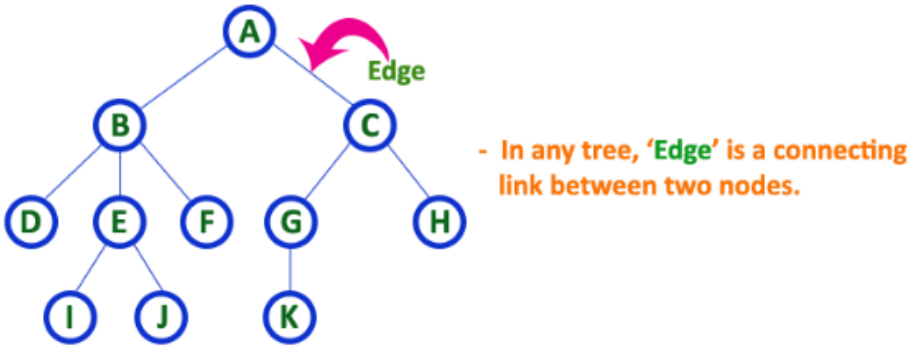


Figure 5.1.10: Representation of an Edge

(ix) Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **Path** between those two Nodes. **Length of a Path** is total number of nodes in that path. In below ***example*** **the path A - B - E - J has length 4**. Figure 5.1.11 shows path in the tree.

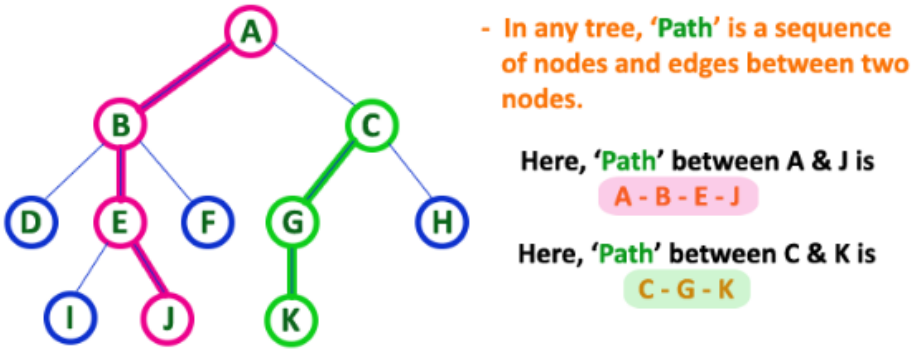


Figure 5.1.11: Path in the Tree.

(x) Depth

In a tree data structure, the total number of edges from root node to a particular node is called as **Depth** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.** Figure 5.1.12 below shows paths in a tree.

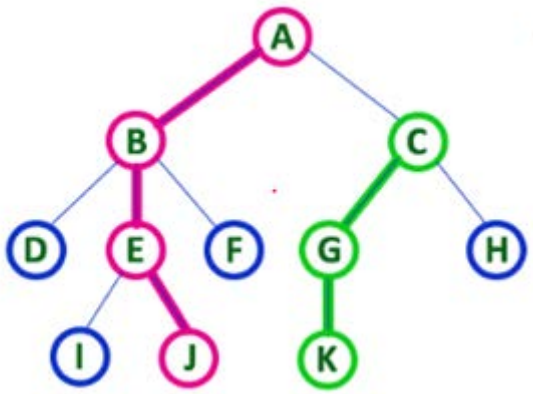


Figure 5.1.12: Paths in a Tree

In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here 'Path' between A and J is A – B – E – J

Here, 'Path' between C and K is C – G - K

(xi) Parent Node

Definition: Parent is a converse notion of a child

In figure 5.1.13 below 2 is parent node of 7 and 5.

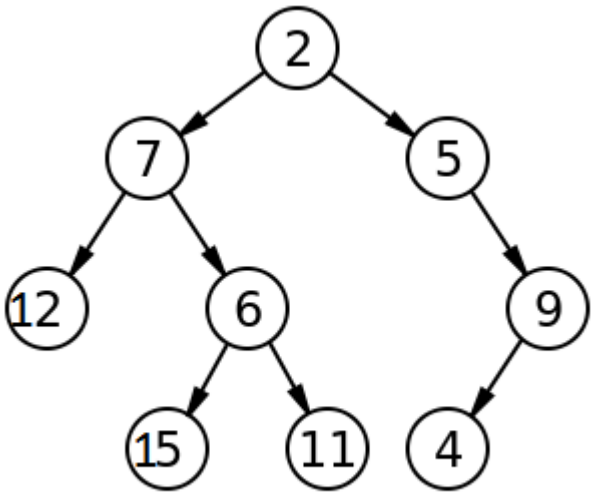


Figure 5.1.13: General Tree Architecture

(xii) Ancestor of a Node

Definition: A node reachable by repeated proceedings from the child to the parent is called as an ancestor node.

In figure 5.1.13, 2 is an ancestor node of 7, 5, 12, 6, 9, 15, 11 and 4

Self-assessment Questions

4) In trees, the origin node or the first node is

a) Intermediate node b) Root node

c) Leaf node d) Ancestor node

5) A node having no child node is called _____

a) Intermediate node b) Root node

c) Leaf node d) Ancestor node

6) A node reachable by repeated proceedings from the child to the parent is called as an ancestor node.

a) Intermediate node b) Root node

c) Leaf node d) Ancestor node

5.1.3 Types of Trees

Based on the structure, properties, operational permeability and applications, trees can be of various types. Some of the most important types of tree data structures are discussed in this section. Following are the three most common and important types of tree data structures which are widely used in different applications of Computer Science.

1. Binary trees
2. Binary search trees
3. Complete binary trees

(i) Binary Trees

A binary tree is a type of tree in which node can have only two child node. The figure 5.1.14 below shows binary tree consisting of a root node and two child nodes or subtrees T_l and T_r. Both of these two child nodes can also be empty.

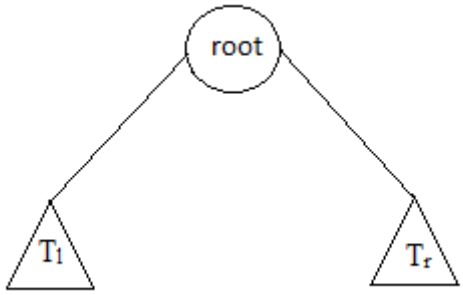


Figure 5.1.14: Generic Binary Tree

One of the important properties of binary tree is that the depth of the average binary tree is considerably lesser than n. According to the analysis the average depth is and that for a special type of binary tree is called binary search tree, the average depth is O (log n). In fact, the depth can be as large as n -1, as the ***example*** in Fig. 5.1.15 shows.

Here, the depth of the tree is 4 and height of the tree is also 4

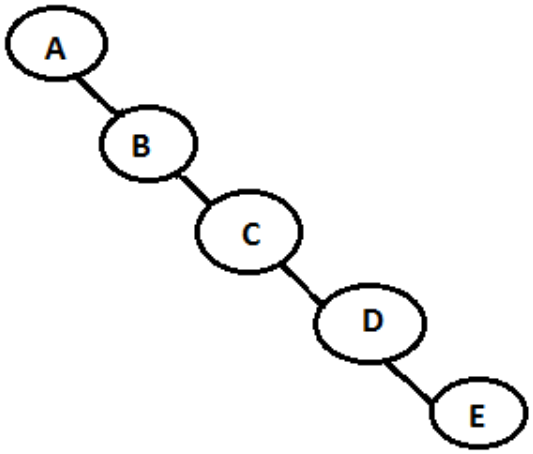


Figure 5.1.15 Worst-case Binary Tree

Most of the rules that implies to linked lists will also apply for trees. Particularly, while performing an insertion operation, a node must be created by allocating memory by calling malloc function. Memory allocated to nodes can be freed after their deletion by calling free function.

The trees could also be drawn using rectangular (boxes) representation which we usually use for representing linked lists. However circles are used for representation as trees are in fact

graphs. Also NULL pointer is not drawn as every binary tree with n nodes would require (n+1) NULL pointers.

We could draw the binary trees using the rectangular boxes that are customary for linked lists, but trees are generally drawn as circles connected by lines, because they are actually graphs. We also do not explicitly draw NULL pointers when referring to trees, because every binary tree with n nodes would require n + 1 NULL pointers.

Terminologies of Binary tree

- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

(ii) Binary Search Trees

The binary search trees (BST) are basically binary trees which are aimed at providing efficient way of data searching, sorting, and retrieving.

A binary search tree can be defined as a binary tree that is either empty or in which every node has a key (within its data entry) and satisfies the following conditions:

- The key of the root (if it exists) is greater than the key in any node in the left sub-tree of the root.
- The key of the root (if it exists) is less than the key in any node in the right sub-tree of the root.
- The left and right sub-trees of the root are again binary search trees.

In above definition, the property 1 and 2 describes ordering relative to the key of the root node and property 3 reaches out to all the nodes of the tree; therefore we can easily implement recursive structure of binary tree. After examining the root of the tree, we move to the either

left or right sub-tree which in-turn is binary search tree. Therefore, we can use the same method again on this smaller tree.

This algorithm is used in such a way that no two entries in the binary tree can have equal key as key of the left sub tree are always smaller than that if root and those of right sub tree are always greater. It is possible to change the definition to allow entries with equal keys; however doing so makes the algorithms somewhat more complicated. Therefore, we always assume: *No two entries in a binary search tree may have equal keys.*

Insertion in Binary Search Tree:

- Check whether root node is present or not (tree available or not). If root is NULL, create root node.
- If the element to be inserted is less than the element present in the root node, traverse the left sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left(key in new node < key in T)/T->right (key in new node > key in T).
- If the element to be inserted is greater than the element present in root node, traverse the right sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left/T->right.

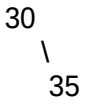
Algorithm for insertion in Binary Search Tree:

```
TreeNode insert(int data, TreeNode T) {
          if T is NULL {
                  T = (TreeNode *)malloc(sizeof (StructTreeNode));
                  (Allocate Memory of new node and load the data into
it)
                  T->data = data;
                  T->left   = NULL;
                  T->right = NULL;
          } else if T is less than T->left {
                  T->left = insert(data, T->left);
                  (Then node needs to be inserted in left sub-
tree.So,
                    recursively traverse left sub-tree to find the
place
                    where the new node needs to be inserted)
          } else if T is greater than T->right {
                   T->right = insert(data, T->right);
                   (Then node needs to be inserted in right sub-tree
                    So, recursively traverse right sub-tree to find
the
                     place where the new node needs to be inserted.)
         }
         return T;
}
```

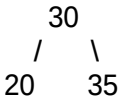
Example:

Insert 30 into the Binary Search Tree.
Tree is not available. So, create root node and place 30 into it.

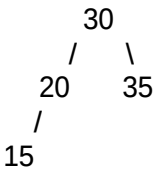
Insert 23 into the given Binary Search Tree. 35> 30 (data in root). So, 35 needs to be inserted in the right sub-tree of 30.



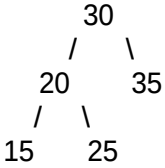
Insert 13 into the given Binary Search Tree. 20<30(data in root). So, 20 needs to be inserted in left sub-tree of 30.



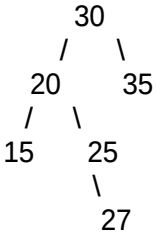
Insert 15 into the given Binary Search Tree.



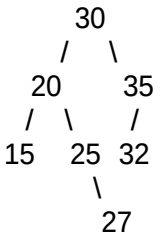
Inserting 25.



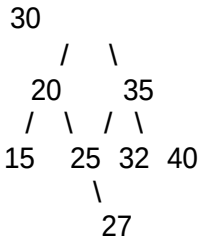
Inserting 27.



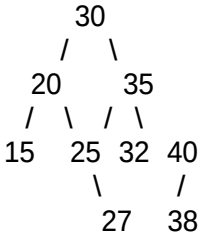
Inserting 32.



Inserting 40.



Inserting 38.

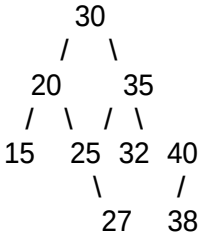


Deletion in Binary Search Tree:

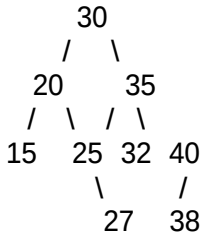
How to delete a node from binary search tree?

There are three different cases that need to be considered for deleting a node from binary search tree.

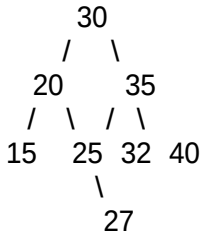
case 1: Node with no children (or) leaf node
case 2: Node with one child
case 3: Node with two children.



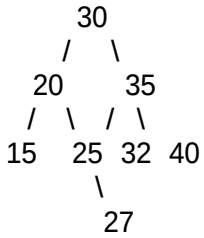
Case 1: Delete a leaf node/ node with no children.



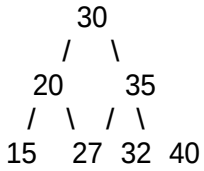
Delete 38 from the above binary search tree.



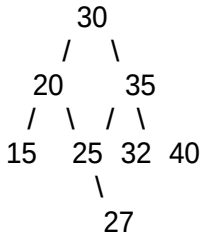
Case 2: Delete a node with one child.



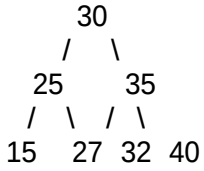
Delete 25 from above binary search tree.



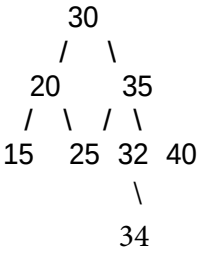
Case 3: Delete a node with two children.
Delete a node whose right child is the smallest node in the right sub-tree. (25 is the smallest node present in the right sub-tree of 20).



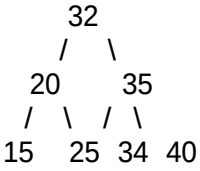
Delete 20 from the above binary tree. Find the smallest in the left subtree of 20. So, replace 20 with 25.



Delete 30 from the below binary search tree.



Find the smallest node in the right sub-tree of 30. And that smallest node is 32. So, replace 30 with 32. Since 32 has only one child(34), the pointer currently pointing to 32 is made to point to 34. So, the resultant binary tree would be the below.



(iii) Complete Binary Tree

If the nodes of a complete binary tree are labeled in order sequence, starting with 1, then each node is exactly as many levels above the leaves as the highest power of 2 that divides its label.

The following figure 5.1.16 depicts how a complete binary tree looks like.

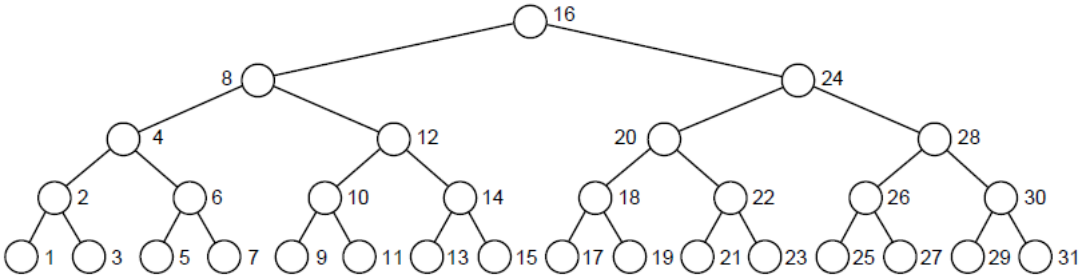


Figure 5.1.16: An Example of Complete Binary Tree

In the above figure 5.1.4, the nodes are labeled as per the execution or traversal of called in-order traversal. Here the left most extreme node is chosen for traversal and the tree is traversed as per the sequence of numbers labeled adjacent to the nodes.

Self-assessment Questions

7) How many child nodes a node in a binary tree can have?

a) One
b) Two
c) Three
d) Four

8) The maximum depth of a binary tree can be _____

a) n+1
b) n
c) n-1
d) 0

9) Binary search trees are aimed at providing efficient calculations.

a) True
b) False

5.1.4 Heap

In computer science, a heap is a specialized tree based data structure that satisfies a heap priority. *i.e.,* if A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap.

Like binary trees, heap has two properties, first is structure property and second is heap order property. A heap is basically a binary tree which is completely filled (exception for bottom level) from left to right. Such tree is also referred as a complete binary tree. Figure 5.1.17 shows an ***example*** of a complete binary tree.

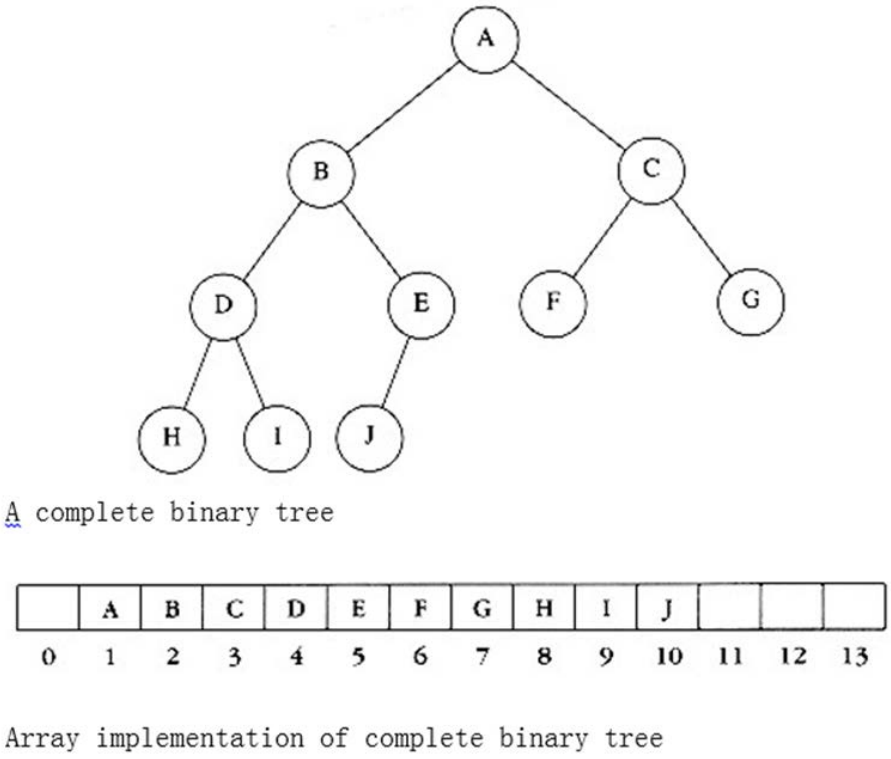


Figure 5.1.17: Complete Binary Tree

In a complete binary tree, for any array element i, its left child is in position 2i, and the right child is in cell after left child (2i+1). Therefore, there is no requirement of a pointer and tree traversal is very simple and fast on many of the computers. The only limitation of this type of implementation is that there must be an estimate of a heap size in advance, which is also not a major issue.

(i) Heap Order Property

This property allows operations to be performed quickly on binary trees. Suppose, the operation is to find the minimum of the elements quickly, then it makes logical sense to have smallest element at the root. If we assume, any sub-tree should be heap, then it must be noted that any node should be smaller than its descendants.

If this logic is applied to a binary tree then the result gives heap order property. Each node X present in heap, the key in the parent of X is always smaller (or equal to) than the key X. Expect for the root which has no parents. In Figure 5.1.18 the tree on the left is a heap, but the tree on the right is not (the dashed line shows the violation of heap order). As usual, we will assume that the keys are integers, although they could be arbitrarily complex.

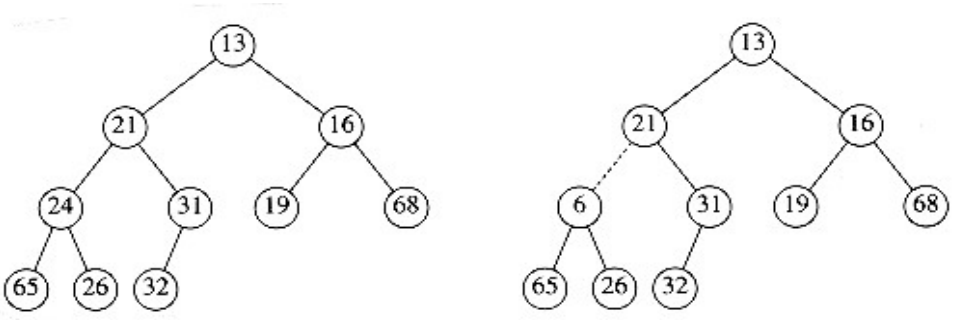


Figure 5.1.18: Two Complete Trees (only the Left Tree is a Heap)

On similar lines, max heap can be declared, which can efficiently find and remove the maximum element, by changing the heap order property. Thus, a priority queue can be used to find either a minimum or a maximum, but this need to be decided ahead of time.

By the heap order property, the minimum element can always be found at the root. Thus, we get the extra operation, find_min, in constant time.

A min-heap is a binary tree such thatλ - the data contained in each node is less than (or equal to) the data in that node's children. - The binary tree is complete

The following figure 5.1.19 depicts a min heap property.

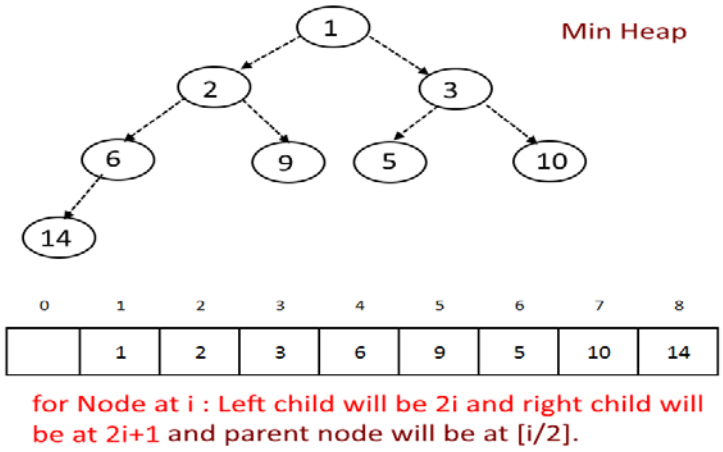


Figure 5.1.19: Min Heap

A max-heap is a binary tree such thatλ - the data contained in each node is greater than (or equal to) the data in that node's children. - The binary tree is complete.

The following figure 5.1.20 depicts a max heap property.

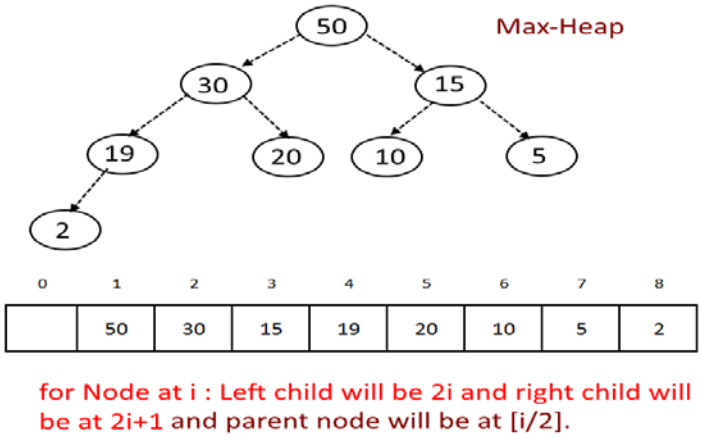


Figure 5.1.20: Max Heap

A binary heap is a heap data structure created using a binary tree.

Binary tree has two rules -

1. Binary Heap has to be complete binary tree at all levels except the last level. This is called ***shape property***.

2. All nodes are either greater than equal to (***Max-Heap***) or less than equal to (***Min-Heap***) to each of its child nodes. This is called ***heap property***.

Implementation:

- Use array to store the data.
- Start storing from index 1, not 0.
- For any given node at position i:
- Its **Left Child** is at ***[2*i]*** if available.
- Its **Right Child** is at ***[2*i+1]*** if available.
- Its **Parent Node** is at ***[i/2]***if available.

Min heap

Here the value of the root is less than or equal to either (left or right) of its children.

The following figure 5.1.21 shows a min heap example.

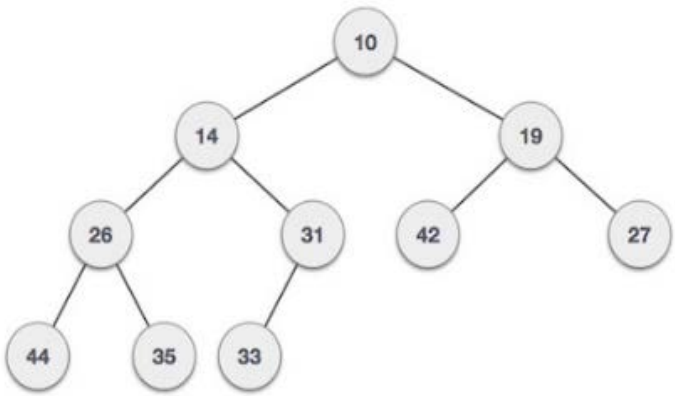


Figure 5.1.21: Min Heap Example

Max heap

Here the value of the root is greater than or equal to either (left or right) of its children.

The following figure 5.1.22 shows a max heap example.

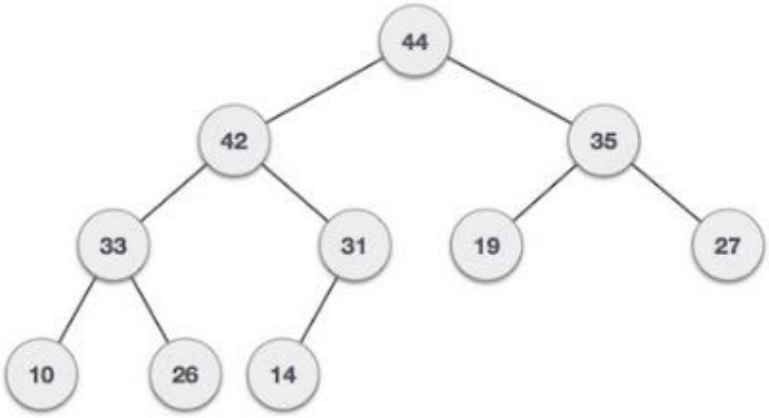


Figure 5.1.22: Max Heap Example

Below is the algorithm used for maintaining the heap property

```
// Input: A: an array where the left and right children of i root heaps (but
i may not), i: an array index
// Output: A modified so that i roots a heap
// Running Time: O(log n) where n = heap-size[A] – i
l ← Left(i)
r ← Right(i)  if l ≤ heap-size[A] and A[l] > A[i]
      largest ← l
else largest ← i
if r ≤ heap-size[A] and A[r] < A[largest]
      largest ← r
if largest != i
      exchange A[i] and A[largest]
      Max-Heapify(A, largest)
```

Heap build algorithm

```
BUILD-HEAP(A)
// Input: A: an (unsorted) array
// Output: A modified to represent a heap.
// Running Time: O(n) where n = length[A]
heap-size[A] ← length[A]
for i ← blength[A]/2  downto 1
Max-Heapify(A, i)
```

Heap build algorithm

```
BUILD-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        do HEAPIFY(A, i)
```

The following figure 5.1.23 shows Example for building a heap

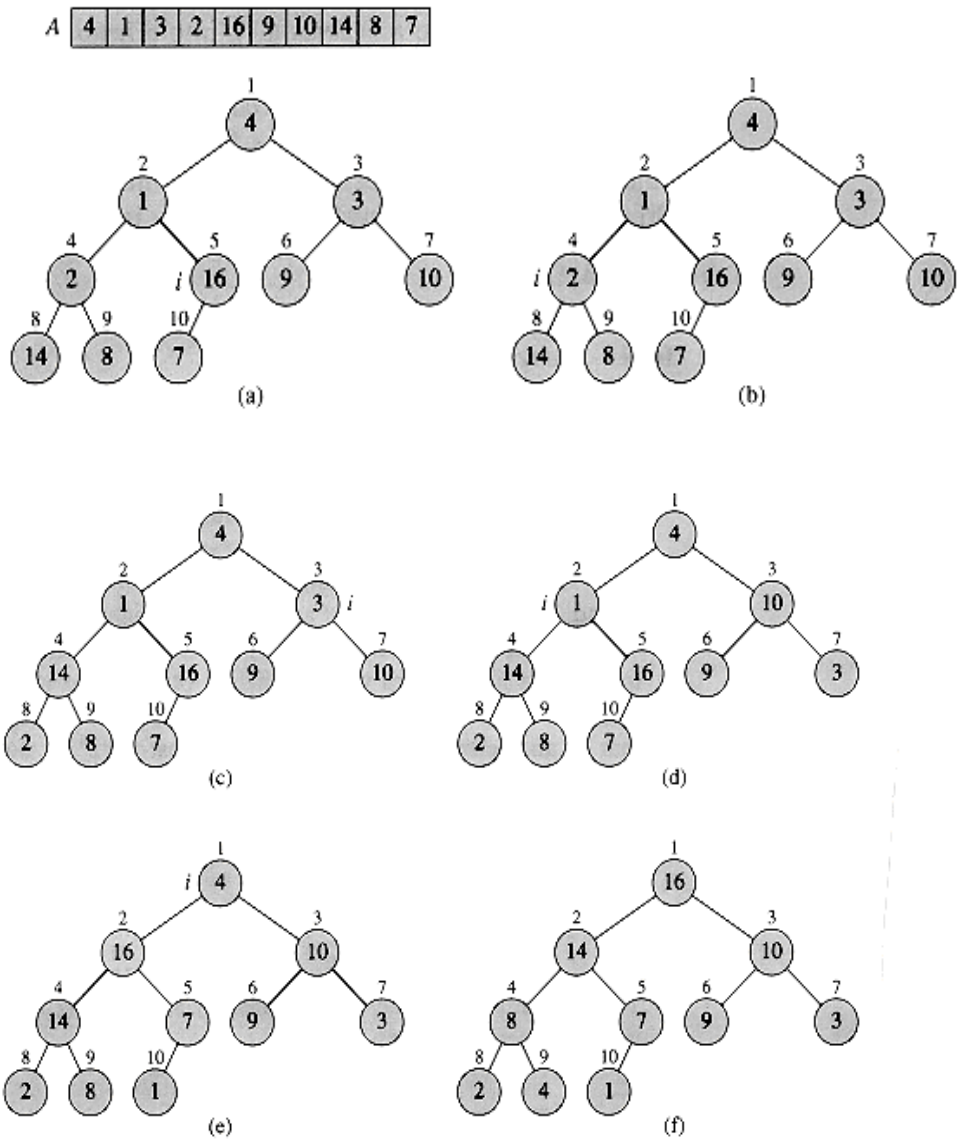


Figure 5.1.23: Example for Building a Heap

(ii) Heap Sort

The heap sort algorithm starts by using BUILD-HEAP to build a heap on the input array $A[1 . . n]$, where $n = length[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap (by decrementing $heap\text{-}size[A]$), we observe that $A[1 . . (n - 1)]$ can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property (7.1). All that is needed to restore the heap property, however, is one call to HEAPIFY(A, 1), which leaves a heap in $A[1 . . (n - 1)]$. The heap sort algorithm then repeats this process for the heap of size n - 1 down to a heap of size 2.

```
HEAPSORT(A)
1  BUILD-HEAP(A)
2  for i ← length[A] downto 2
3        do exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] -1
5           HEAPIFY(A, 1)
```

Program for heap sort:

```
/*
 * C Program to sort an array based on heap sort algorithm(MAX heap)
 */
#include <stdio.h>

int main()
{
  int heap[10], n, i, j, cnt, root, temp;

  printf("\nEnter the number of elements: ");
  scanf("%d",&n);
  printf("\nEnter the elements: ");
  for(i =0; i< n; i++)
  scanf("%d",&heap[i]);
  for(i =1; i< n; i++)
  {
    cnt = i;
    do
    {
      root =(cnt -1)/2;
      if(heap[root]< heap[cnt])/* to create MAX heap array */
      {
        temp = heap[root];
```

```
        heap[root]= heap[cnt];
        heap[cnt]= temp;
      }
      cnt = root;
    }while(cnt !=0);
  }

  printf("Heap array elements are: ");
  for(i =0; i< n; i++)
  printf("%d  ", heap[i]);
  for(j = n -1; j >=0; j--)
  {
    temp = heap[0];
    heap[0]= heap[j];    /* swap max element with rightmost leaf element */
    heap[j]= temp;
    root =0;
    do
    {
      cnt =2* root +1;/* left node of root element */
      if((heap[cnt]< heap[cnt +1])&&cnt< j-1)
      cnt++;
      if(heap[root]<heap[cnt]&&cnt<j)/* again rearrange to max heap array
      */
      {
        temp = heap[root];
        heap[root]= heap[cnt];
        heap[cnt]= temp;
      }
      root = cnt;
    }while(cnt< j);
  }
  printf("\nThe sorted array elements after Heap sort are: ");
  for(i =0; i< n; i++)
  printf("%d  ", heap[i]);
}
```

Output:

```
sh-4.3$ gcc -o main *.c
sh-4.3$ main

Enter the number of elements: 7

Enter the elements: 16
5
33
41
18
27
12
Heap array elements are: 41  33  27  5  18  16  12
The sorted array elements after Heap sort are: 5  12  16  18  27  33  41  sh-4.3$
```

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-HEAP takes time $O(n)$ and each of the n - 1 calls to HEAPIFY takes time $O(\lg n)$. The Figure 5.1.24 shows heap sort.

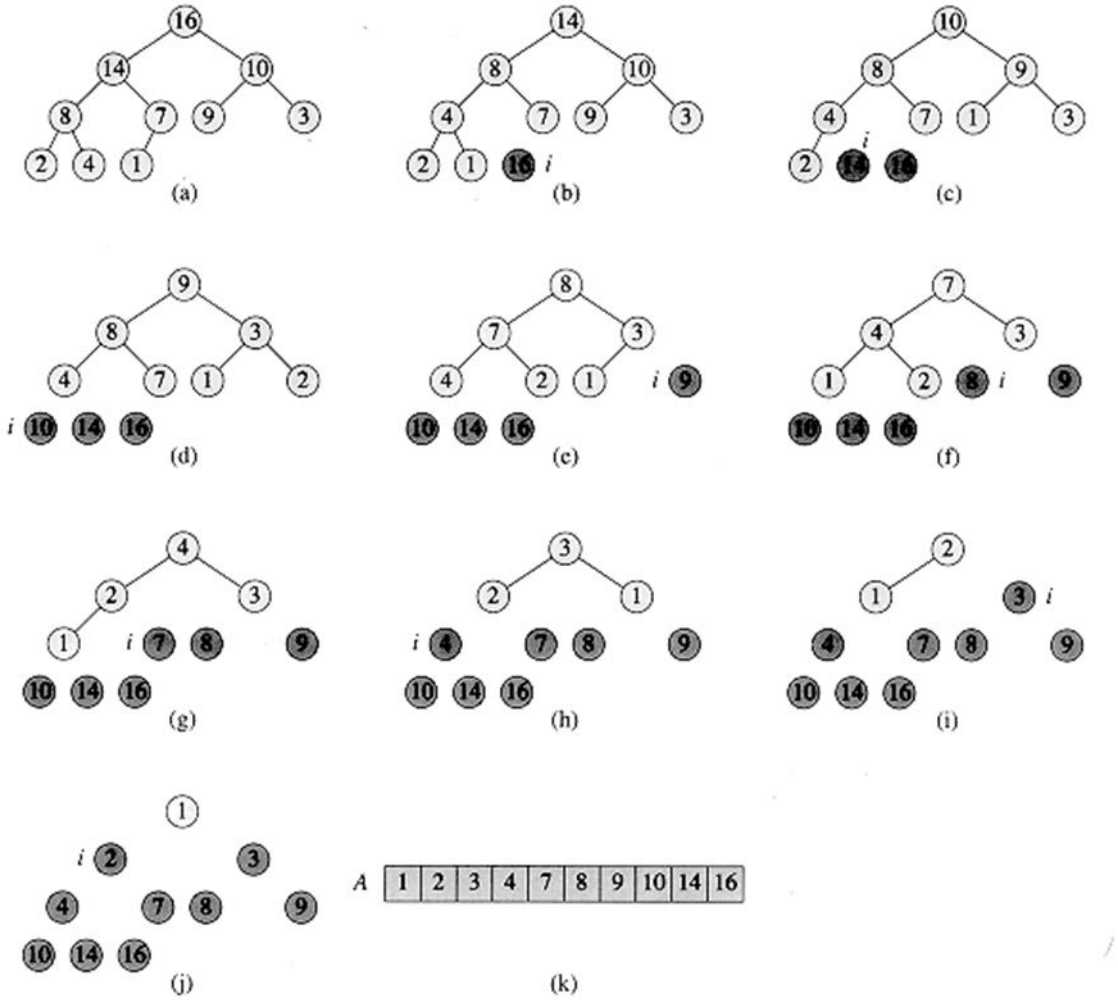


Figure 5.1.24: HeapSort

Did you know?

Data structure "heap" might be used in various places. Heap used for dynamic memory allocation wherever it is needed. There are two types of heap "ascending heap" and "descending heap". In ascending heap root is the smallest one and in descending heap root is the largest element of the complete or almost complete binary tree.

Self-assessment Questions

9) If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap.

a) True b) False

10) As per the heap property it makes logical sense to have smallest element at _____

a) Leaf or terminal nodes b) Intermediate nodes

c) Root nodes d) Non-terminal nodes

11) We can easily implement priority queue using heap order property.

a) True b) False

5.1.5 Binary Tree

As discussed in section 5.1.2.a, a binary tree is a type of tree in which node can have only two child node. That is maximum degree of a binary tree can be 2. Let us discuss how a binary tree can be represented in the form of an array and also how to create binary trees.

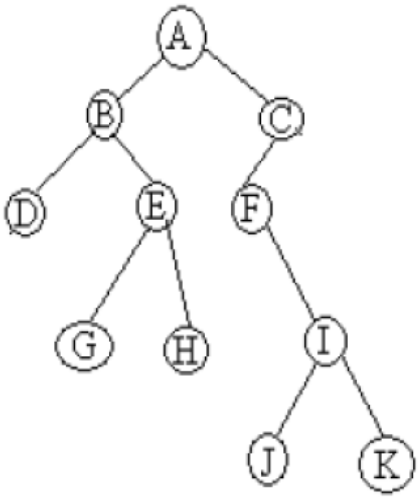
(i) Array Representation

A binary tree can be represented as both single dimension as wee as two dimensional array called as Adjacency Matrix.

Adjacency Matrix representation:

A two dimensional array can be used to store the adjacency relations very easily and can be used to represent a binary tree. In this representation, to represent a binary tree with n vertices we use n×n matrix.

Figure 5.1.25(a) shows a binary tree and Figure 5.1.24(b) shows its adjacency matrix representation.



	A	B	C	D	E	F	G	H	I	J	K
A		L	R								
B				L	R						
C						L					
D											
E							L	R			
F									R		
G											
H											
I										L	R
J											
K											

(a) A binary tree (b) Adjacency matrix representation

Figure 5.1.25: Representation of a Binary Tree in the Form of Adjacency Matrix

From the above representation, we can understand that the storage space utilization is not efficient. Now, let us see the space utilization of this method of binary tree representation. Let '*n*' be the number of vertices. The space allocated is *n* x *n* matrix. i.e., we have *n*2number of locations allocated, but we have only *n*-1 entries in the matrix. Therefore, the percentage of space utilization is calculated as follows:

$$\frac{n-1}{n^2} \cong n = \frac{1}{n} \times 100\%$$

The percentage of space utilized decreases as n increases. For large 'n', the percentage of utilization becomes negligible. Therefore, this way of representing a binary tree is not efficient in terms of memory utilization.

Single dimension array representation

Since the two dimensional array is a sparse matrix, we can consider the prospect of mapping it onto a single dimensional array for better space utilization.

In this representation, we have to note the following points:

- The left child of the ith node is placed at the 2ith position.
- The right child of the ith node is placed at the (2i+1)th position.
- The parent of the ith node is at the (i/2)th position in the array.

If *l* is the depth of the binary tree then, the number of possible nodes in the binary tree is 2*l*+1-1. Hence it is necessary to have 2*l*+1-1 locations allocated to represent the binary tree.

If 'n' is the number of nodes, then the percentage of utilization is

$$\frac{n-1}{2^{l+1}-l}\times 100$$

Figure 5.1.26 shows a binary tree and Figure 5.1.27 shows its one-dimensional array representation.

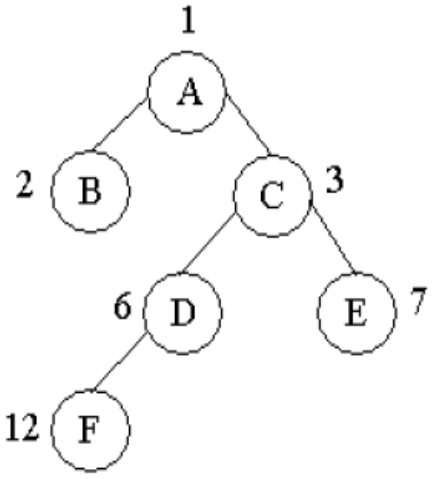


Figure 5.1.26: A Binary Tree

1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			D	E					F

Figure 5.1.27: One- dimensional Array Representation

For a complete and full binary tree, there is 100% utilization and there is a maximum wastage if the binary tree is right skewed or left skewed, where only l+1 spaces are utilized out of the 2l+1 – 1 spaces.

$$\text{i.e., } \frac{l+1}{2^{l+1}-1} \times 100$$

An important observation to be made here is that the organization of the data in the binary tree decides the space utilization of the representation used.

(ii) Creation of a Binary Tree

Following is an algorithm for creation of a binary tree.

Step 1: Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.

Step 2: Create a new tree node tNode with the data as picked element.

Step 3: Find the picked element's index in Inorder. Let the index be in Index.

Step 4: Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.

Step 5: Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.

Step 6: Return tNode.

Below is the C program for implementing the above algorithm.

```
  intinIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     rightsubtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);

  returntNode;
}
/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
```

```
intsearch(chararr[], intstrt, intend, charvalue)
{
  inti;
  for(i = strt; i<= end; i++)
  {
    if(arr[i] == value)
      returni;
  }
}
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
structnode* newNode(chardata)
{
  structnode* node = (structnode*)malloc(sizeof(structnode));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}
/*
}
```

Program:

```
#include <stdlib.h>
typedefstructtnode
{
   int data;
   structtnode *right,*left;
}TNODE;

TNODE *CreateBST(TNODE *,int);
voidInorder(TNODE *);
void Preorder(TNODE *);
voidPostorder(TNODE *);
main()
{
      TNODE *root=NULL;       /* Main Program */
  intopn,elem,n,i;
  do
  {
    clrscr();
    printf("\n ### Binary Search Tree Operations ### \n\n");
    printf("\n Press 1-Creation of BST");
    printf("\n       2-Traverse in Inorder");
    printf("\n       3-Traverse in Preorder");
    printf("\n       4-Traverse in Postorder");
    printf("\n       5-Exit\n");
    printf("\n       Your option ? ");
    scanf("%d",&opn);
```

```
    switch(opn)
    {
      case1: root=NULL;
      printf("\n\nBST for How Many Nodes ?");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
      {
        printf("\nRead the Data for Node %d ?",i);
        scanf("%d",&elem);
        root=CreateBST(root,elem);
      }
      printf("\nBST with %d nodes is ready to Use!!\n",n);
      break;
      case2:printf("\n BST Traversal in INORDER \n");
      Inorder(root);break;
      case3:printf("\n BST Traversal in PREORDER \n");
      Preorder(root);break;
      case4:printf("\n BST Traversal in POSTORDER \n");
      Postorder(root);break;
      case5:printf("\n\n Terminating \n\n");break;
      default:printf("\n\nInvalid Option !!! Try Again !! \n\n");
      break;
    }
    printf("\n\n\n\n  Press a Key to Continue . . . ");
    getch();
  }while(opn !=5);
}
TNODE *CreateBST(TNODE *root,intelem)
{
  if(root == NULL)
  {
    root=(TNODE *)malloc(sizeof(TNODE));
    root->left= root->right = NULL;
    root->data=elem;
    return root;
  }
  else
  {
    if(elem< root->data )
    root->left=CreateBST(root->left,elem);
    else
    if(elem> root->data )
    root->right=CreateBST(root->right,elem);
    else
    printf(" Duplicate Element !! Not Allowed !!!");

    return(root);
  }
}
voidInorder(TNODE *root)
```

```
{
   if( root != NULL)
   {
      Inorder(root->left);
      printf(" %d ",root->data);
      Inorder(root->right);
   }
}

void Preorder(TNODE *root)
{
   if( root != NULL)
   {
      printf(" %d ",root->data);
      Preorder(root->left);
      Preorder(root->right);
   }
}

voidPostorder(TNODE *root)
{
   if( root != NULL)
   {
      Postorder(root->left);
      Postorder(root->right);
      printf(" %d ",root->data);
   }
}
```

```
sh-4.3$ gcc -o main *.c
sh-4.3$ main

Binary Search Tree Operations
Press 1-Creation of BST      2-Inorder Traversal      3-Preorder Traversal      4-Postorder Traversal      5-Exit
Enter your choice: 1
Enter the number of nodes for BST: 5
Enter the data for the node 1 15
Enter the data for the node 2 25
Enter the data for the node 3 10
Enter the data for the node 4 30
Enter the data for the node 5 20
BST with 5 nodes is created.

Press a Key to Continue . . .

Binary Search Tree Operations
Press 1-Creation of BST      2-Inorder Traversal      3-Preorder Traversal      4-Postorder Traversal      5-Exit
Enter your choice: 2
Inorder Traversal for BST  10  15  20  25  30

Press a Key to Continue . . .

Binary Search Tree Operations
Press 1-Creation of BST      2-Inorder Traversal      3-Preorder Traversal      4-Postorder Traversal      5-Exit
Enter your choice: 3
Preorder Traversal for BST  15  10  25  20  30

Press a Key to Continue . . .

Binary Search Tree Operations
Press 1-Creation of BST      2-Inorder Traversal      3-Preorder Traversal      4-Postorder Traversal      5-Exit
Enter your choice: 4
Postorder Traversal for BST  10  20  30  25  15

Press a Key to Continue . . .

Binary Search Tree Operations
Press 1-Creation of BST      2-Inorder Traversal      3-Preorder Traversal      4-Postorder Traversal      5-Exit
Enter your choice: 5
Terminating

Press a Key to Continue . . .
sh-4.3$
```

Self-assessment Questions

12) A tree can only be represented as a 2 dimensional array.

a) True b) False

13) A two dimensional representation of an array is a sparse matrix

a) True b) False

5.1.6 Traversal of Binary Tree

A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of nodes or information represented by them. There are three simple ways to traverse a tree. They are called *preorder*, *inorder*, and *postorder*. In each technique, the left sub-tree is traversed recursively, the right sub-tree is traversed recursively, and the root is visited. What distinguishes the techniques from one another is the order of those three tasks. The following sections discuss these three different ways of traversing a binary tree.

(i) Preorder Traversal

In this traversal, the nodes are visited in the order of root, left child and then right child.

- Process the root node first.
- Traverse left sub-tree.
- Traverse right sub-tree.

Repeat the same for each of the left and right sub-trees encountered. Here, the leaf nodes represent the stopping criteria. The pre-order traversal sequence for the binary tree shown in Figure 5.1.28 is: A B D E H I C F G

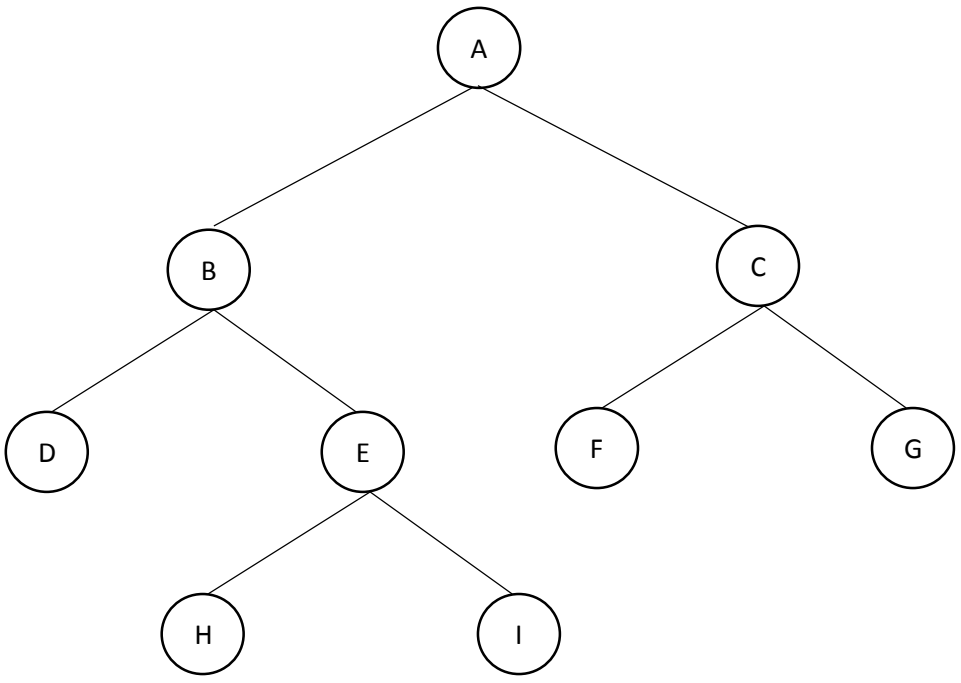


Figure 5.1.28: A Binary Tree

Consider the following example

The following figure 5.1.29 shows pre-order traversal example

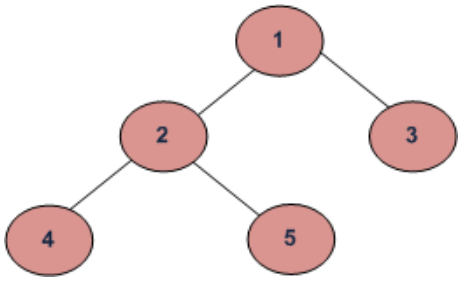


Figure 5.1.29: Pre-order Traversal Example

The pre-order traversal for the above tree is 1->2->4->5->3.

(ii) Inorder Traversal

In this traversal, the nodes are visited in the order of left child, root and then right child. *i.e.,* the left sub-tree is traversed first, then the root is visited and then the right sub-tree is traversed. The function must perform only three tasks.

- Traverse the left subtree.
- Process the root node.
- Traverse the right subtree

Remember that visiting a node means doing something to it: displaying it, writing it to a file and so on. The Inorder traversal sequence for the binary tree shown in Figure 5.1.27 is: D B H E I A F C G.

Consider the following example:

The following figure 5.1.30 shows in-order traversal example

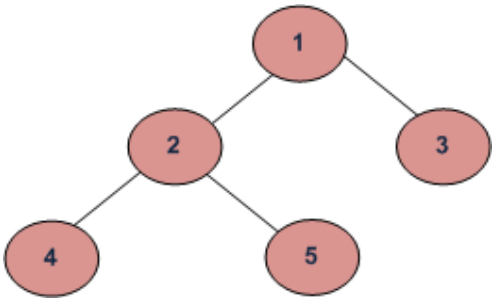


Figure 5.1.30: In-order Traversal Example

The in-order traversal for the above tree is 4->2->5->1->3.

(iii) Postorder Traversal

In this traversal, the nodes are visited in the order of left child, right child and then the root. *i.e.,* the left sub-tree is traversed first, then the right sub-tree is traversed and finally the root is visited. The function must perform the following tasks.

- Traverse the left subtree.
- Traverse the right subtree.
- Process the root node.

The post order traversal sequence for the binary tree shown in Figure 5.1.27 is: D H I E B F G C A.

Consider the following example:

The following figure 5.1.31 shows post-order traversal example

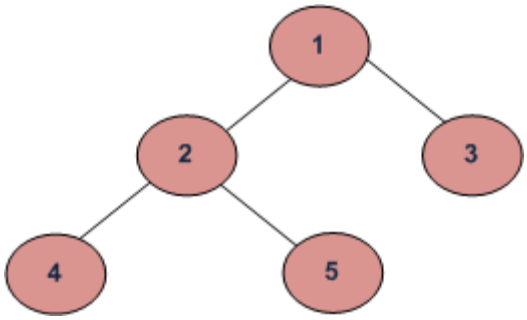


Figure 5.1.31: Post-order Traversal Example

The post-order traversal for the above tree is 4->5->2->3->1.

Self-assessment Questions

14) Traversal in a binary tree is order in which we visit the nodes in a tree.

a) True b) False

15) In preorder traversal nodes are visited in order of _____

a) Life child – Root – Right child b) Left child – Right child – Root

c) Root – Left child – Right child d) Root – Right child – Left child

Summary

- Tree can be defined as a non-linear data structure consisting of a root node and other nodes present at different levels forming a hierarchy.
- There are three different types of trees; Binary trees, Binary search trees and complete binary trees.
- A binary tree is a type of tree in which a node can have only two child nodes.
- A binary search tree is a binary tree that is either empty or in which every node has a key.
- A complete binary tree is a binary tree in which every level of the tree is completely filled except the last nodes towards the right.
- A heap is a specialized tree based data structure that satisfies a heap priority which makes it suitable for implementing priority queues.
- There are various terminologies used for identification and analysis of various types of trees like root, nodes, degree of node/tree, terminal nodes, non-terminal nodes, siblings, level, edge, path, depth, parent node and ancestral node.
- Binary tree can be implemented in both single and two dimensional arrays.
- Binary tree can be traversed using three orders or traversal; inorder, preorder and postorder.

Terminal Questions

1. List and explain advantages of a tree.
2. Explain different types of tree.
3. Explain various terminologies used in context of a tree.
4. Explain different types of tree traversals.

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	a
3	b
4	b
5	c
6	d
7	b
8	a
9	c
10	a
11	c
12	b
13	a
14	a
15	a
16	c

Activity

Activity Type: Offline **Duration:** 15 Minutes

Description:

Ask all the students to solve the given problem,

Convert an array [10,26,52,76,13,8,3,33,60,42] into a maximum heap.

Bibliography

e-References

- cs.cmu.edu, (2016). *Binary Trees.* Retrieved on 19 April 2016, from, https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html
- comp.dit.ie, (2016). *Heap Sort.* Retrieved on 19 April 2016, from, http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/heap%20sort.pdf

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C'* (2nd ed.). Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth* (2nd ed.). BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.

Video Links

Topic	Link
Tree Terminologies	https://www.youtube.com/watch?v=nq7m0Gll-60
Binary Tree Traversal	https://www.youtube.com/watch?v=-aIcPlIQ_MI
Binary Tree Representation	https://www.youtube.com/watch?v=1EsBpPmGEEE

Notes: