# Trees

Until now we have studied linear data structures. But there are many applications where only the linear data structures are not sufficient. For such situations, we use non-linear data structures like trees and graphs.
In computer science, a tree is an abstract model of a hierarchical structure.

- It is used to represent a hierarchical relationship existing among several data items.
- A tree consists of nodes with a parent-child relationship.
- It is a non-linear data structure.
- Searching as fast as in ordered array.
- Insertion and deletion as fast as in linked list

**Graph theoretic definition of a Tree**
It is a finite set of one or more data items (nodes) such that
- There is a special data item called the root of the tree
- Its remaining data items are partitioned into number of disjoint subsets, each of which is itself a tree again, called subtrees.

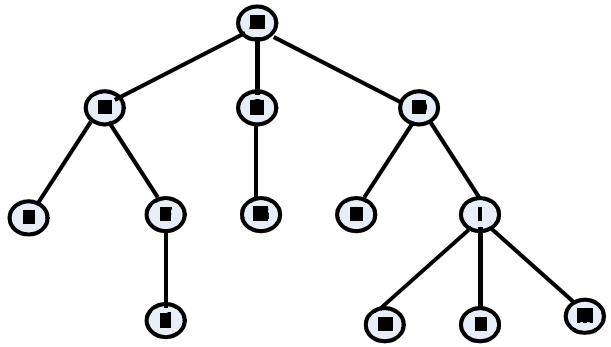


**fig: a tree structure**

**Basic Terminologies in Tree**

- **Root: Root** is a special data item, which is the first in the hierarchy. In the above tree,' **A**' is the root item

- **Node:** Each item in a tree is called a **node**. ***A, B, F, G*** etc. are examples of node.





- **Parent:** A node that points to (one or more) other nodes is the **parent** of those nodes while the nodes pointed to are the **children.** B is parent of E, F and E,F are children of B

- **Leaf (External Nodes):** Nodes with no children are **leaf** nodes or **external** nodes or **terminal** nodes. In the above tree E, J, G, H, K, L, and M are terminal nodes

- **Internal Nodes:** Nodes with children are **internal nodes** or **non-terminal nodes**. Note that the root node is also an internal node. Node such as **B, C, D**, are non-terminal nodes

- **Siblings (Brother):** Nodes that have the same parent are **siblings**.
  In the above tree:
  - **E** and **F** are siblings of parent node **B**
  - **K, L**, and **M** are siblings of parent node **I.**

- **Descendents:** The **descendents** of a node consist of its children, and their children, and so on. All nodes in a tree are descendents of the root node (except the root node itself).

- **Ancestors:** The **ancestors** of a node consist of the parent node and the parent of that node also. The root node is the ancestor of all nodes and the root does not have any ancestor.

- **Order (Degree): Order (degree) of a node** is given by the number of its children. If a node has two children, its degree is two.
  In the above tree.
  - degree of node A is 3.
  - degree of node B is 2.
  - degree of node C is 1.

  Similarly, **order (degree) of a tree** is given by the maximum of the degrees of its nodes. In other words, if the degree of a tree is n, its nodes can have, at most, n children. In the above tree the node A has degree 3 which is the maximum degree. **So the degree of the above tree is 3**.

- **Edge:** The link between any two nodes is called an **edge**. It is directed from the parent towards its child.

- **Path:** There is a **single, unique path** from the root to any node. For e.g. the path from node A to J is A, B, F, and J.

- **Height:** A **height of a node** is equal to the maximum path length **from that node to a leaf node**. A leaf node has a height of 0. The **height of a tree** is equal to the height of the **root**. For e.g. the height of node B from leaf node J is 2.

– **Depth:** A **depth of a node** is equal to the maximum path length from root to that node. A root has a depth of 0. **The depth of a tree** is equal to the depth of the deepest node in the tree. It, of course, is equal the tree height. For e.g. the depth of node J is 3.

– **Forest:** it is a set of disjoint tress. In a given tree, if you remove its node then it becomes a forest.

– **Level:** The **level of a node** is 0, if it is root. Otherwise, it is one more than its parent. For e.g.
  - Node **C** is at level 1
  - Node **K** is at level 3

**Binary Tree**
A binary tree is a finite set of data items which is either empty or consists of a single item called the root and the two disjoint binary trees called the **left subtree** and **right subtree.** In a binary tree the maximum degree of any node is **at most two.** That means, there may be a zero degree node (usually an empty tree) or one degree node or two degree node.

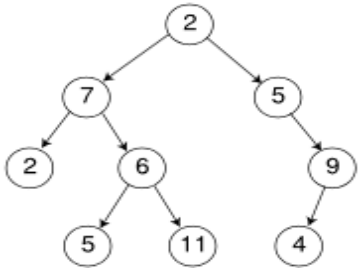



**Fig: A simple binary tree, in which every node has at most two children.**

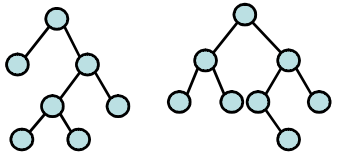
**Some properties of a binary tree**
– Number of external nodes ≤ number of internal nodes + 1
– Number of nodes at level i $\leq 2^i$
– Number of external nodes $\leq 2^{height}$ (Here, height means height of the tree)
– Number of nodes in a tree $\leq 2^{height+1}-1$ (Here, height means height of the tree)
– height $\geq \log_2$(number of external nodes)
– height $\geq \log_2$ (number of nodes) – 1





**Types of binary trees**
There are two types of binary trees.

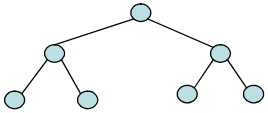
A **Strictly binary tree** or **Full binary tree**, or **Proper binary tree**, is a tree in which every node has zero or two children. i.e. Every internal node has nonempty left and right subtrees. A strictly binary tree with n leaves contains 2n – 1 nodes.



Here the first tree is a Strictly binary tree, while the second one is not.

A **Complete binary tree** or a **Perfect binary tree** is a strictly binary tree in which all leaves are at the same depth. It has following properties.

– Every internal node has 2 children
– Height of n will have $2^{n+1} – 1$ nodes
– Height of n will have $2^n$ leaves
– No of nodes in level l have $2^l$ nodes
– No of external nodes = No of internal nodes + 1



**Tree traversals**
Visiting the nodes in certain order is called traversing. The reason for visiting the nodes is to process the contents of the node according to our requirement.
In a linear structure like an array or a linked list, there are two ways of traversal
–from front to end and from end to front
These traversals can be achieved, simply, by using loops.

Tree, being a non-linear data structure, there are three different ways for traversal.
**Preorder, Inorder and Postorder traversals**.

The general rules for these traversals are as follows:

**PreOrder traversal (also called depth-first traversal)**
In this traversal, the root node is visited prior to the left or right node in a subtree. The rule is
1. Visit the root node (**N**).
2. then, visit the nodes in the left subtree in preorder(**L**).
3. finally, visit the nodes in the right subtree in preorder(**R**).

**InOrder traversal**

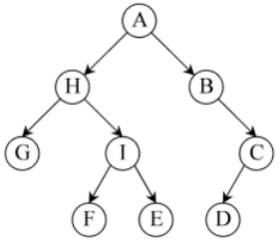
In this traversal, the root node is visited in between the left and the right node in a subtree. The rule is

1. Visit the nodes in the left subtree in inorder (**L**).
2. Visit the root node (**N**).
3. Finally, visit the nodes in the right subtree in inorder(**R**).

**PostOrder traversal (also called breadth first traversal)**

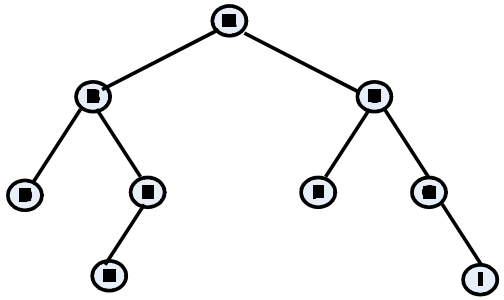
In this traversal, the root node is visited after the left and the right node in a subtree. The rule is

3. visit the nodes in the left subtree in postorder(**L**)
3. then, visit the nodes in the right subtree in postorder(**R**).
3. Finally, visit the root node (**N**).




- Preorder traversal : A, H, G, I, F, E, B, C, D
- Postorder traversal : G, F, E, I, H, D, C, B, A
- In-order traversal : G, H, F, I, E, A, B, D, C

**More examples of tree traversal**



The order of **Preorder traversal** is: *A B D E H C F G I*
The order of **Inorder traversal** is: *D B H E A F C G I*
The order of **Postorder traversal** is *D H E B F I G C A*
An expression tree is used to represent arithmetic and logical expressions. It is built up from the simple operands and operators of an expression by placing the simple operands as the leaves of a binary tree, and the operators as the interior nodes.





## Expression Tree

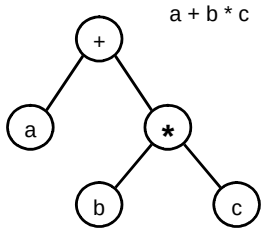
An expression tree is used to represent arithmetic and logical expressions. It is built up from the simple operands and operators of an expression by placing the simple operands as the leaves of a binary tree, and the operators as the interior nodes.
For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator and the right subtree contains everything in the right operand
For a unary operator, one subtree will be empty.

**Expression tree traversals**

When an expression tree is traversed in inorder, we get an infix expression. When it is traversed in preorder, we get the equivalent prefix expression. Similarly, when it is traversed in postorder, we get the equivalent postfix expression.

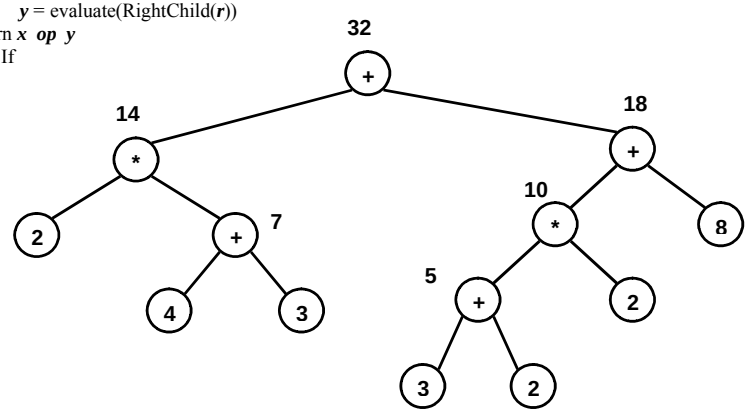
**Evaluation of Expression Tree**

Evaluation of an expression tree is done by applying the operator at the root to the values obtained by recursively evaluating left and right subtrees



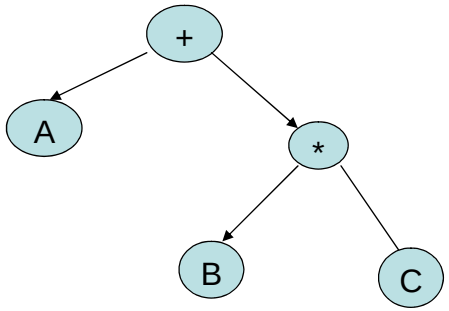

**An Algorithm**

**evaluate** (***r***)
– If ***r*** is a leaf
  return the value stored in ***r***
– Else
  Let op be the operator stored in ***r***
  ***x*** = evaluate(LeftChild(***r***))
  ***y*** = evaluate(RightChild(***r***))
  return ***x op y***
– End If



**Example of an expression tree from a given expression**

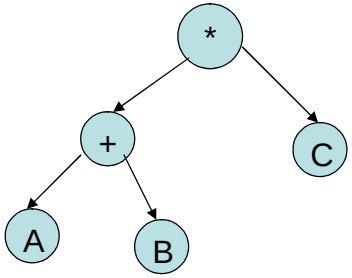
**1. A + B * C**




As, already said, When an expression tree is traversed in inorder, we get an infix expression. When it is traversed in preorder, we get the equivalent prefix expression. Similarly, when it is traversed in postorder, we get the equivalent postfix expression.So, now let us check:

**The preorder traversal is: + A * B C (equivalent to prefix expression)**
**The postorder traversal is: A B C * + (equivalent to postfix expression)**

**2. (A + B) * C**

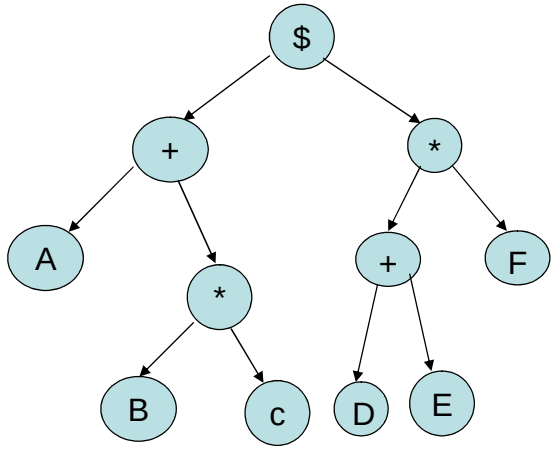



**The preorder traversal is: * + A B C (equivalent to prefix expression)**
**The postorder traversal is: A B + C * (equivalent to postfix expression)**





**3. (A + B * C) $ ((D + E) * F)**



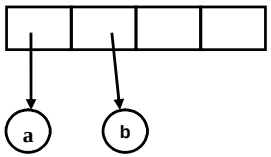

**The preorder traversal is: $+A*BC*+DEF (**equivalent to prefix expression**)**
**The postorder traversal is: ABC*+DE+F*$ (**equivalent to postfix expression**)**
**Algorithm for construction of an expression tree out of a postfix expression**

**Given a postfix expression, following is the algorithm to build an expression tree**. The inorder and preorder traversals of this tree will give infix and prefix expressions respectively.

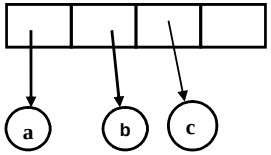
1. Make an empty stack.

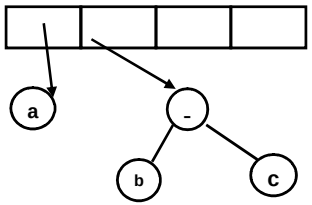
2. For each character from left to right
- if the character is an operand.
  make a single node tree for this operand, push its address into the stack.
  elseif it is an operator, say op
  make a single node tree for this operator
  pop the stack, make it right child of op
  pop the stack, make it left child op
  push the tree (whose root is op) into the stack
- endif

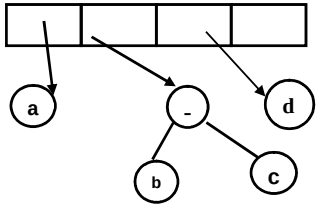
3. Pop the stack to get the required expression tree

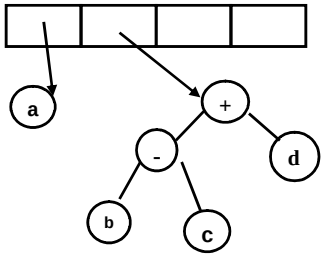


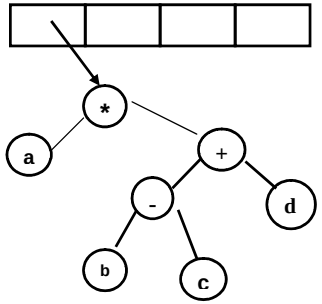

Let the postfix expression be **abc-d+***








## Representation of Binary Tree

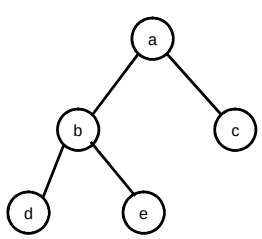
A binary tree can be represented in two different ways.

- Using linked list
- Using array

## Linked representation and implementation

A binary tree is a collection of nodes. Each node in a binary tree has three fields, the data part, and two pointers that point to the root node of left and right subtree. For a leaf the left and right pointers will be null. Also for an empty subtree, the corresponding pointers will be null. An external pointer is used to point to the root node of the tree.

```
struct bnode
{
        int info;                       //for storing data. Currently it is of type int.
        struct bnode *leftchild;  //pointer to the root of the left subtree
        struct bnode *rightchild; //pointer to the root of the right subtree
};
```



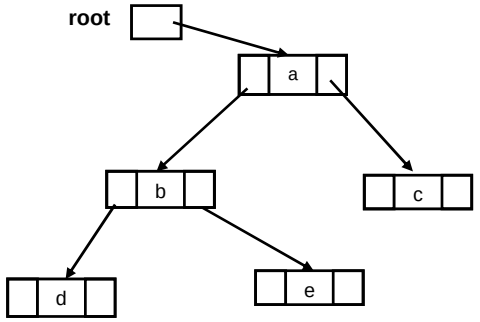




Fig: A binary tree and its linked list representation

**Array Representation of Binary Tree**

A Binary Tree can be represented by means of a linear array having index ‘k’, 0< k<= n, where n is the number of nodes.

The root node is stored at the index 0.
Parent(k) : The Parent of the node at index k is given by floor((k-1)/2) if k is not equal to 0

LeftChild(k) : Left child of the node at k is given by 2k+1.

RightChild(k) : Right child of the node at k is given by 2k+2.

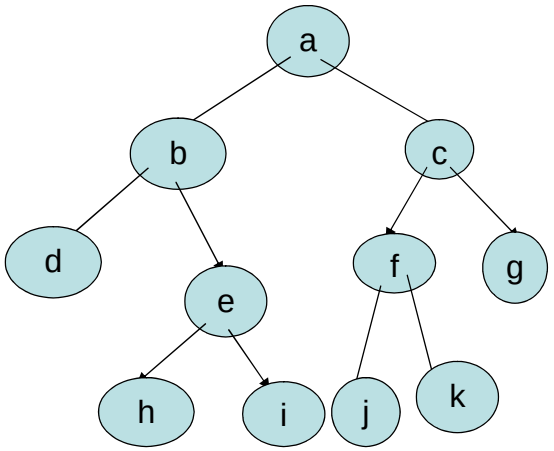
Siblings(k) : If the left child at index k is given, then its right sibling is at k+1. Similarly, if right child at index k is given, its left child is at k-1.





Some Binary Trees along with their sequential representation are as follows:

| | |
|---|---|
| **0** | **a** |
| **1** | **b** |
| **2** | **c** |
| **3** | **d** |
| **4** | **e** |
| **5** | **f** |
| **6** | **g** |
| **7** | |
| **8** | |
| **9** | **h** |
| **10** | **i** |
| **11** | **j** |
| **12** | **k** |
| **13** | |
| **14** | |



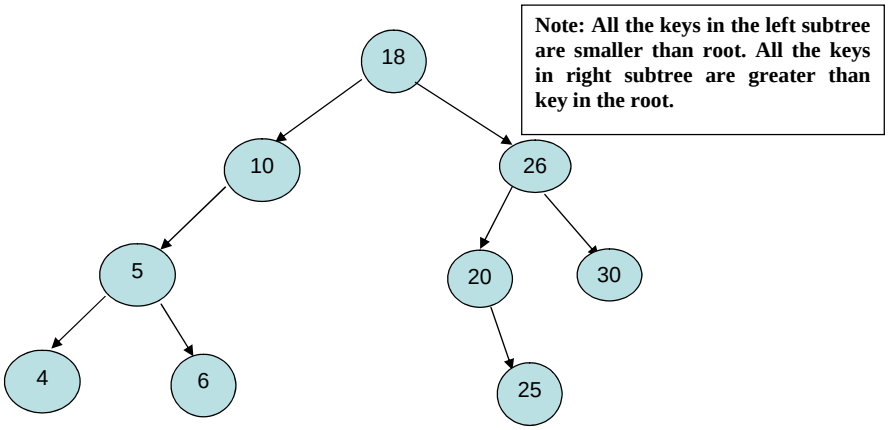

A sequential representation consumes more space for representation binary trees. But for representing complete binary tree, it proves to be efficient as no space is wasted. But, the amount of spaces has to be determined at first.

**Binary Search Tree (BST)**

A Binary Search Tree is a binary tree which is either empty, or in which every node contains a key and satisfies the following criteria:

- All keys (if any) in the left –subtree is smaller than the key in the root.
- All keys (if any) in the right subtree is larger than the key in the root.
- The left and right subtrees again BST.

**E.g. 18 10 26 5 20 30 4 6 25**




**Note: All the keys in the left subtree are smaller than root. All the keys in right subtree are greater than key in the root.**

- With the above definition, we can also conclude that, no two nodes in a BST have same value.
- The inorder traversal of a BST yields the keys in a sorted order (ascending).
- If the tree is reasonably full, searching for a key takes **$O(\log_2(n))$** time, where n is the number of nodes.

**Operations on a Binary Search Tree**

**Search(x, T):** Search for key x in the BST T.if x is found in some node n of T return a pointer to node n or NULL otherwise.

**Insert (x,T):** Insert a new node with x in the info part in the tree T such that the property of BST is maintained.

**Delete(x,T):** Delete a node x in the info part from the tree T such that the property of BST is maintained.

**FindMin(T):** Returns pointer to a node having minimum key in tree T,NULL if T is empty.

**FindMax(T):** Returns pointer to a node with maximum key in T,NULL if T is empty.





**Searching**

In BST, searching of the desired data item can be performed by branching into left or right subtree until the desired data item is found. We proceed from the root node.

- If the tree is empty, then we failed to find x, so return NULL.
- Else, compare x with the key stored in the root of T.
  - ✓ If x is same as the key in root, return address of the root.
  - ✓ If x is smaller than the key in root, search for x in the left subtree of T recursively.

If x is greater than the key in root, search for x in right subtree recursively.

**Insert**

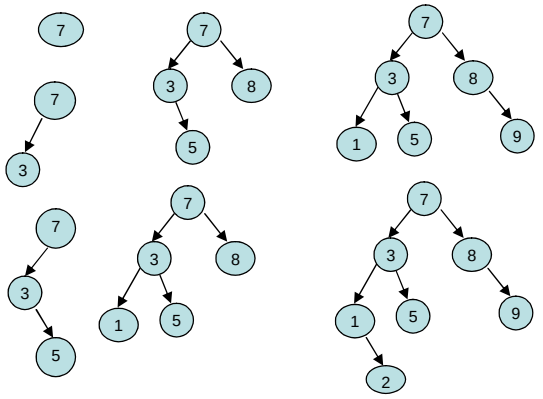
In BST, we don't allow duplicate data items. So, to insert a data item having key 'K' in a binary search tree, we must check that its key is different from those of existing data items by performing a search for data item with the same key K. If the search is unsuccessful, the data item is inserted into BST at the point the search is terminated.

**Algorithm**

Given a new item with key x, then following algorithm will insert it in BST T.

- If the tree is empty, create a new node n storing the key x and make this new node n the root of the tree. Otherwise, compare x to the key stored at root R.
- If x is identical to key in R, don't change the tree.
- If x is smaller, insert new item into left subtree of the root recursively.
- If x is larger, insert new item into right subtree of the root recursively.

**E.g. Trace 7,3,5,8,1,9,2**



**Now trace 15,10,20,17,25,5,13,33,9,21**

**Deletion**

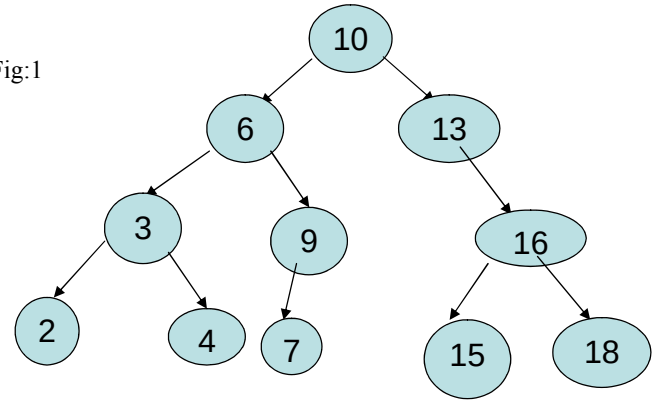
To delete a given node (key), it is searched first. If search is unsuccessful, the algorithm terminates. Otherwise, the following possibilities arise:

- If node is a leaf, it can be deleted immediately.
- If the node has one subtree, the node can be deleted by replacing it with the root of the non-empty subtree.

If the node has two subtrees, we replace the node with the rightmost node of the left subtree $T_r$ or the leftmost node of the right subtree $T_r$

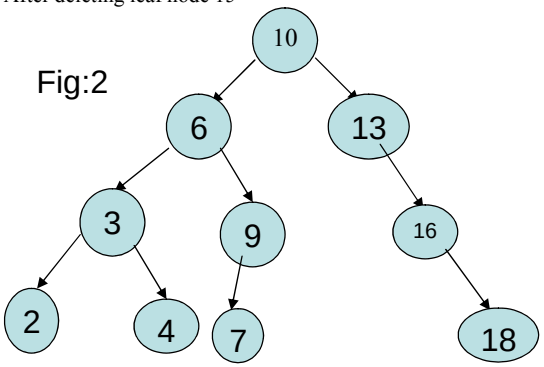
**E.g.**

Fig:1




After deleting leaf node 15

Fig:2

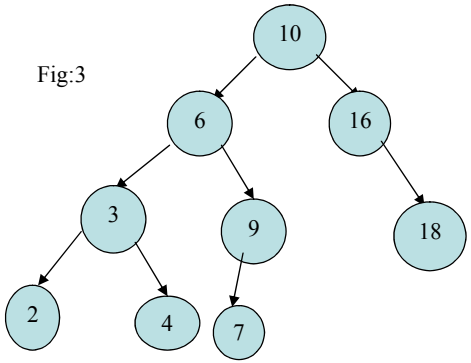



Now lets delete node with key 13 from the above tree.

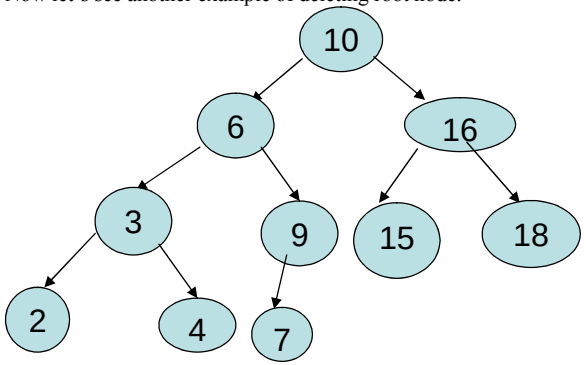




Fig:3

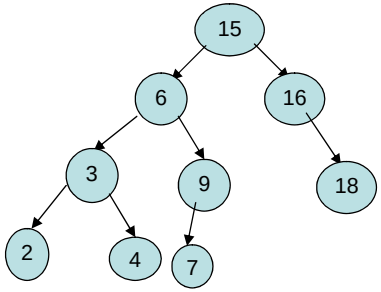



Now let's see another example of deleting root node.

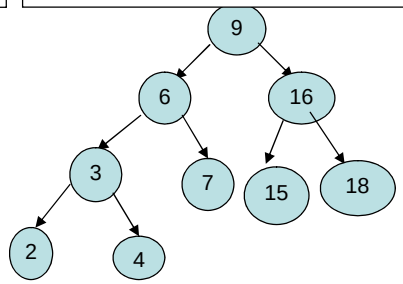



**Deleting root node with key 10**

**Leftmost node of right subtree takes its place**



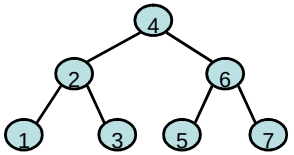

**Rightmost node of left subtree takes its place**

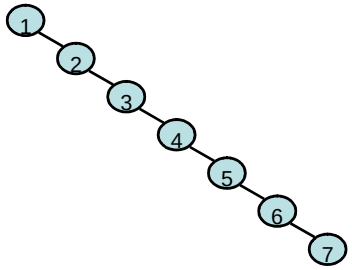


**Efficiency of Binary Search Tree**

1. Complexity of BST operations is proportional to the length of the path from the root to the node being manipulated, i.e., height of the tree
2. In a well balanced tree, the length of the longest path is roughly log n, where n is the no of nodes in the tree
    a. E.g., 1 million entries → longest path is log21,048,576 = 20
3. For a thin, unbalanced tree, operations become O(n):
    a. E.g., elements are added to tree in sorted order
4. So, Balancing is important to decrease the computational complexity

Example:




In the left tree, the *Search* operation requires 3 comparisons to find 7

In the right tree, the *Search* operation requires 7 comparisons to find 7

1. We know the **complete binary** tree has the shortest overall path length for any binary tree.
2. The longest path in a complete binary tree with n elements is guaranteed to be no longer than log(n).
3. If we can keep our tree complete, we're set for fast search times but the cost is very high to maintain a complete binary tree.
4. Instead, use **height-balanced binary trees**: for each node, the height difference between the left and right sub trees is at most one.



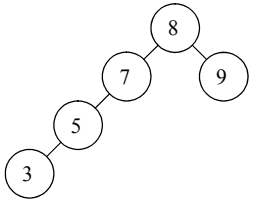


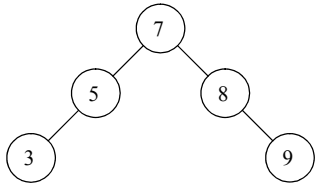
# Heap

A **binary heap** or simply a heap is an **almost complete binary tree** with the key in each node, such that

- All the leaves of the tree are on two adjacent levels.
- All the leaves on the lowest level occur to the left and all levels except possibly the lowest are filled.
- The key in the root is at least as large as the keys in its children (if any), and the left and rights subtrees are again heaps.

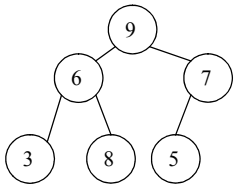
Above definition is of max heap (descending partially ordered tree). For min heap the root has the smallest key.



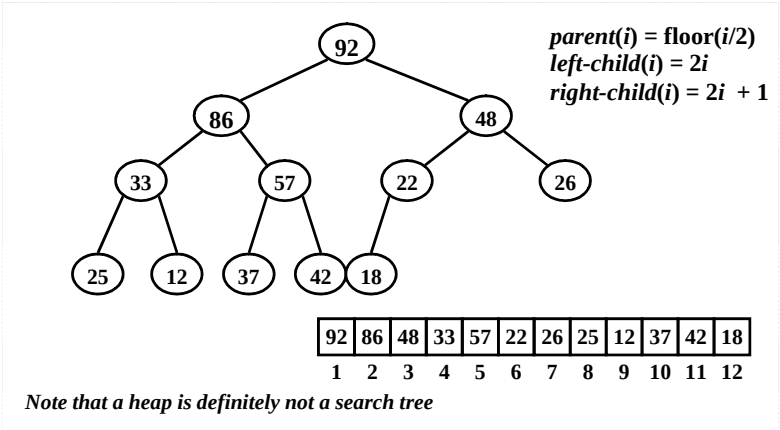




This is not a Heap since it violates rule 1

This is not a Heap since it violates rule 2




This is not a Heap since it violates rule 3




***Note that a heap is definitely not a search tree***

**Types of Heap:**

**Max Heap** : For max heap, the root has the largest key. The key at the parent node is larger than the keys at their children and descendents.

**Min Heap** : The root has the smallest key. The key at the parent node is smaller than the keys at their children and descendents.

Heaps are represented in array rather than linked structures since it is a special kind of binary tree that have no gaps in an array implementation.

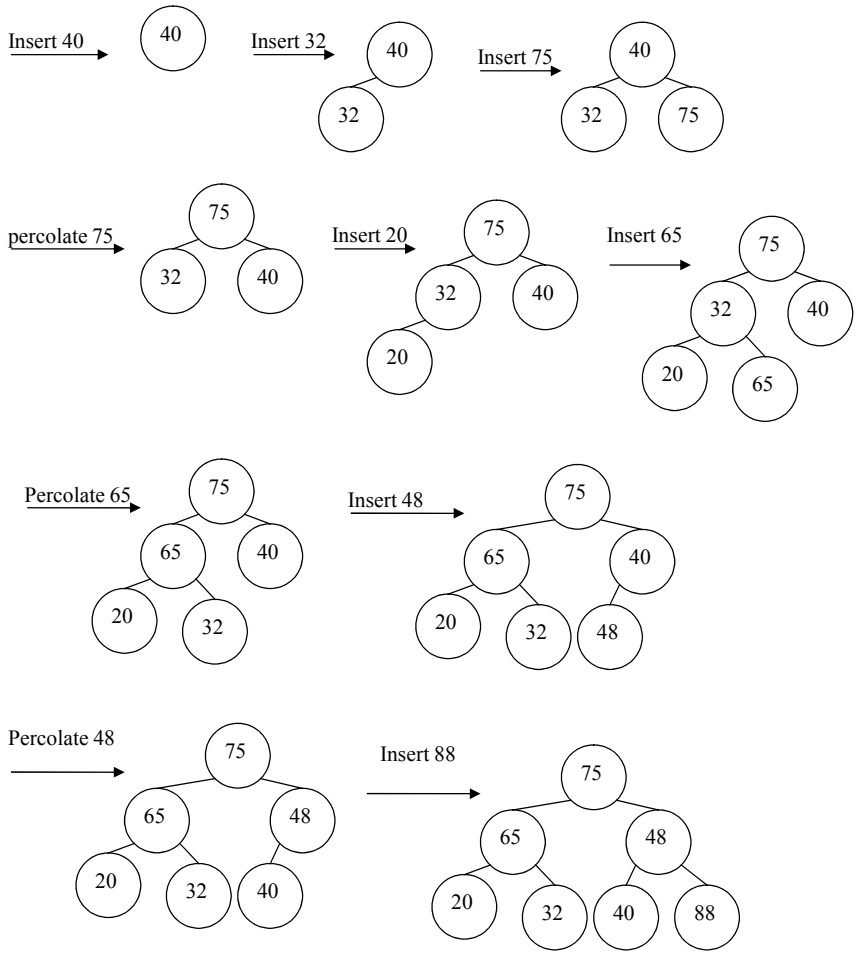
**Creating a Heap and Insertion:**

A Binary tree with only one node satisfies the properties of a heap.
To insert a new element in to the heap, we create a hole in the next available location, since otherwise the tree will not be complete tree.
If new item can be placed in the hole without violating heap order, then we do insert there. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up towards the root. The process is continued until the new item can be placed in the hole. This strategy is known as a percolate up. The new element is percolated up the heap until the correct location is formed.





Eg: Inserting following numbers in the max Heap: 40, 32, 75, 20, 65, 48, 88.



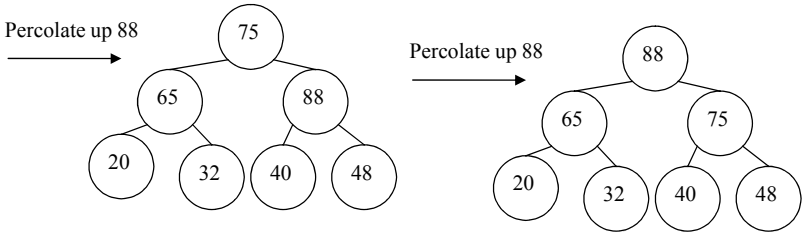



**Fig: Creating Max Heap**

**Deletions in Heap**

Deleting in heap is always applied for the item at the root (i.e minimum in min heap and maximum in max heap). When the root item is removed, a hole is created at the root. Since, the heap now becomes one smaller, it follows that the last element 'x' in the heap is placed in the hole. If the heap property is violated, we slide the smaller (larger) hole's children into the hole in the case of min (max) heap. This process of pushing up the smaller item is continued until the item 'x' finds its proper place. For deleting in heap lets take above example:





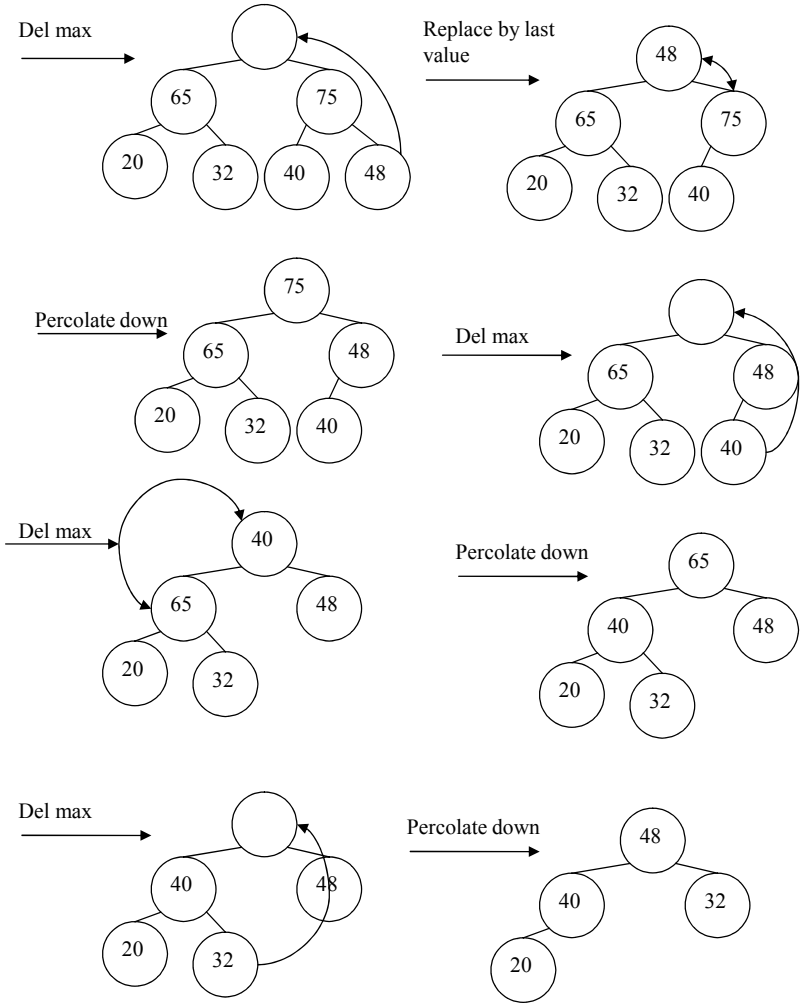



**Fig: Deletion in Max Heap**

**Huffman Algorithm**

Lots of compression systems use Huffman Algorithm for the compression of text, images, audio, data etc. The general idea behind the Huffman algorithm is that, use less number of bits to represent mostly used key.
For example:

Let's say we get four kind of signal in a system.
**A B C D**
For such system, we might use 2-bit representation as;
**A 00**
**B 01**
**C 10**
**D 11**
And we have data string as,

**AAABCAABBCDDAAAABAAABABABADDAAAABC**
If the number of A is 10,000, B is 234, C is 45 and D is 134 then the total storage required will be (10,000 + 234 + 45 + 134) * 2 = 20,826 bits to store.

Since, the data 'A' is repeated ten thousand times, if the storage for A can be reduced, then the total space required will automatically be reduced. Therefore, in Huffman algorithm, the data (symbol) that is used most is represented by a single bit.

Now, while assigning the code word, the most frequently used symbol is assigned a code word of length '1', and the final one always gets code word with all ones.

Let's assign code word for the symbols as;
**A 0**
**B 10**
**C 110**
**D 111**

Now, for the given data string, the binary value will be,

**000101100010101101111110000100001001001001111111000010110**

Now, if we calculate the total space required to store the data, it will be:

**(10,000 * 1 + 234 * 2 + 45 * 3 + 134 * 3) =11,005 bits.**

Here, we saved **20,826 – 11,005 = 9,821** bits.
If the data to be stored is very large, then more bits will be saved.



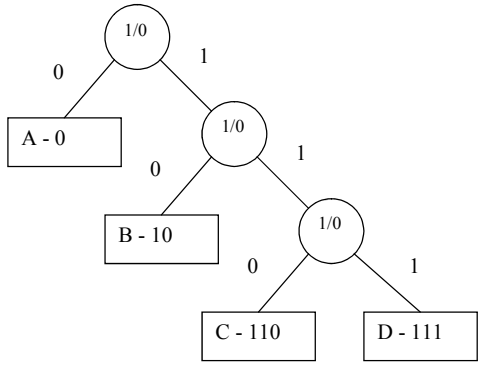


Now the problem is how we recover the original data from compressed string;

**000101100010101101111110000100001001001001111111000010110**

Here, a tree is made according to the code word assignment.

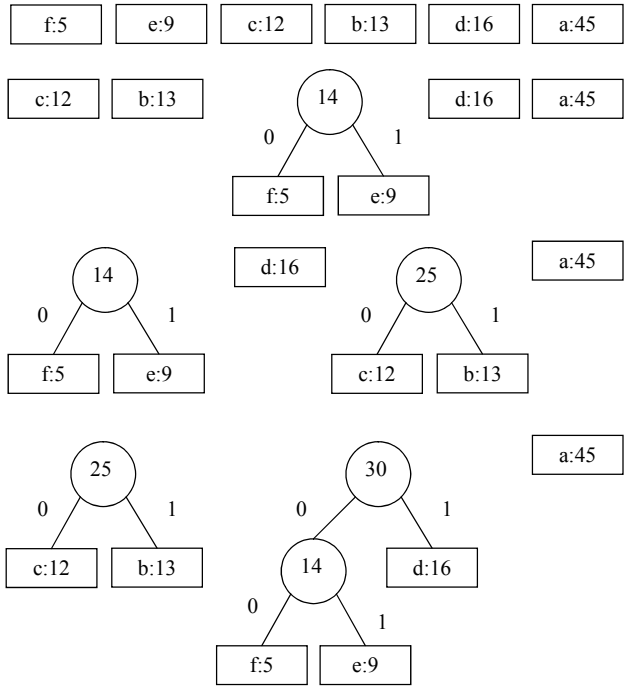
For our example, we have to make a tree like;




Now, to decode the encoded string, read a bit at a time. Compare with root node. If it is 0, then we are at the end decision point. So, it must be 'A'. If it was 1, then we have to read the next bit again and check. If the next bit is 0 then it is B. Otherwise, read next bit and check until we do not reach a decision point.

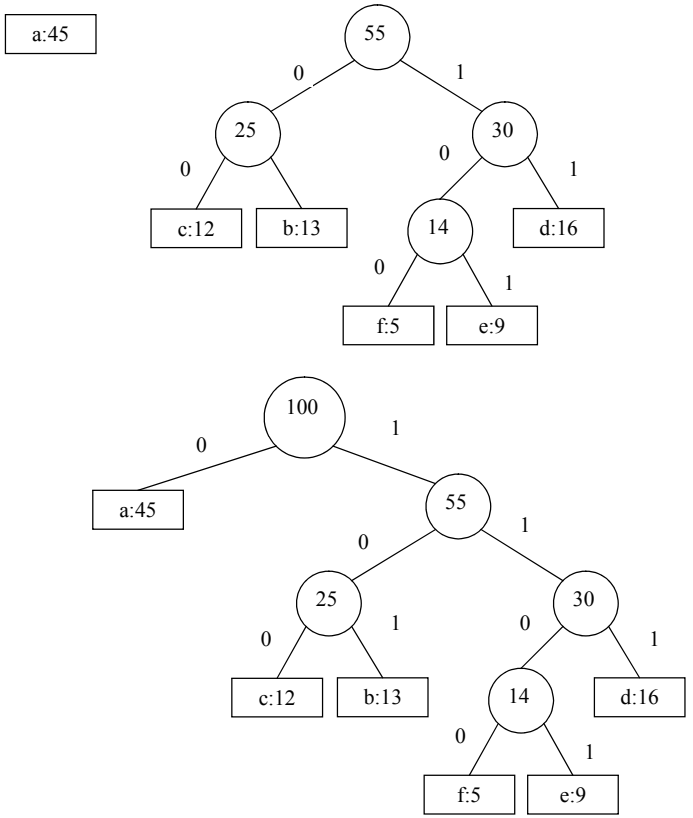
**Constructing a Huffman code**

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| **Frequency (in thousands)** | **45** | **13** | **12** | **16** | **9** | **5** |
| **Fixed-length codeword** | **000** | **001** | **010** | **011** | **100** | **101** |
| **Variable-length codeword** | **0** | **101** | **100** | **111** | **1101** | **1100** |








Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged.

Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a lift child and 1 if it is an edge to a right child.

The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.