3.0 SORTING

Sorting refers to the operation of arranging data in ascending or descending order.

Example

Suppose an array DATA contains 8 elements as follows:

DATA: 77, 99, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 44, 55, 66, 77, 88, 99

There are several types of sorting algorithms:

- Insertion sort
- Selection sort
- Bubble sort

Correctness

An algorithm is correct with respect to a problem if for every instance of the inputs (that is, the specified properties of the inputs are satisfied) the algorithm halts, and the outputs produced satisfy the specified input/output relation.

3.1 Insertion sort

This is an in-place comparison-based sorting algorithm. An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works? We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. Return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Example2

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

```
Pass 1: 77, 33, 44, 11, 88, 22, 66, 55
Pass 2: 33, 77, 44, 11, 88, 22, 66, 55
Pass 3: 33, 44, 77, 11, 88, 22, 66, 55
Pass 4: 11, 33, 44, 77, 88, 22, 66, 55
Pass 5: 11, 33, 44, 77, 88, 22, 66, 55
Pass 6: 11, 22, 33, 44, 77, 88, 66, 55
Pass 7: 11, 22, 33, 44, 66, 77, 88, 55
Pass 8: 11, 22, 33, 44, 55, 66, 77, 88
```

Example 3

Write a java program to carry out an insertion sort on the following elements.
74, 47, 85,69,45,60,10,56,43

```
public class InsertionSort
{
 public static void main(String[ ] Args)
 {
 int[] Marks={74,47,85,69,45,60,10,56,43};
  int  i, j, temp, x;      /* i and j are control variables for the loops i.e determine indices of array. x
is for displaying passes */
 System.out.print("\nPass 1:");
        for(i=0;i<9;i++)
           {
           System.out.print(Marks[i] + "\t"); // prints elements of the array as they are i.e non sorted
                    }
                     //Insertion Sort Begins
                   for(i=0;i<8;i++)
                     {
                              if(Marks[i]>Marks[i+1])
                               {
                               temp=Marks[i+1];         //47 will be allocated to temp
                                Marks[i+1]=Marks[i];   // 47 will be replaced by 74
                                          Marks[i]=temp;              //74 will be replaced by 47
                                        for(j=i;j>0;j--)
```

```
                                        {
                                         if(Marks[j]<Marks[j-1])
                                                {
                                                        temp=Marks[j-1];
                                                        Marks[j-1]=Marks[j];
                                                        Marks[j]=temp;
                                                }
                                        }
                                }
                        System.out.print("\nPass"+(i+2)+ ":");
                        for(x=0;x<9;x++)
                        {
                                System.out.print(Marks[x] + "\t");//Display elements in each pass
                        }
                }
                //End of Insertion Sort
        }
}
```

Output

```
Pass 1: 74   47  85  69  45  60  10  56  43
Pass 2: 47   74  85  69  45  60  10  56  43
Pass 3: 47   74  85  69  45  60  10  56  43
Pass 4: 47   69  74  85  45  60  10  56  43
Pass 5: 45   47  69  74  85  60  10  56  43
Pass 6: 45   47  60  69  74  85  10  56  43
Pass 7: 10   45  47  60  69  74  85  56  43
Pass 8: 10   45  47  56  60  69  74  85  43
Pass 9: 10   43  45  47  56  60  69  74  85
```

Insertion Sort runtimes

Best case: insertion sort is efficient when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.

Worst case: insertion sort is inefficient if the numbers were sorted in reverse order.

3.2 selection Sort

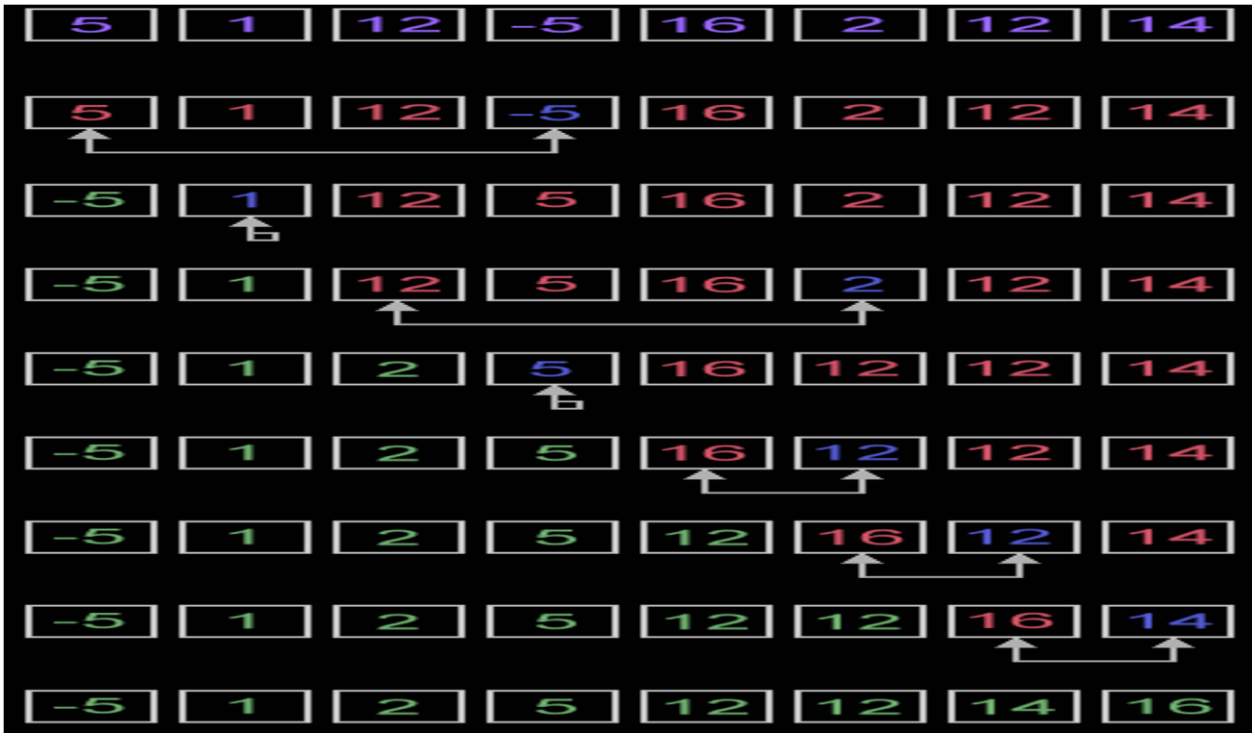
The idea behind selection sort is that we put a list in order by placing each item in turn. In other words, we put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

Selection sort is inefficient for sorting large data volumes.

Selection sort is notable for its programming simplicity (applying swaps) and it can over perform other sorts in certain situations

Example1.

Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



Example 2

Write a java program to carry out a selection sort on the following elements.

74, 47, 85, 69, 45, 60, 10, 56, 43

```
public class SelectionSort
{
public static void main(String[ ] Args)
        {
        int[] Marks={74,47,85,69,45,60,10,56,43};
```

```
        int i,j,temp,x,ind;     // ind keeps position of the smallest element in each pass. Others
remain the same

        for(i=0;i<8;i++)
                        {
                         ind=i;
                         temp=Marks[i];
                              for(j=i;j<9;j++)
                                  {
                                   if(Marks[j]<temp)
                                   {
                                   temp=Marks[j];
                                   ind=j;
                                   }
                                  }
                                  Marks[ind]=Marks[i];
                                  Marks[i]=temp;
                                  System.out.print("\nPass" + (i+1)+":");
                                  for(x=0;x<9;x++)
                                  {
                System.out.print(Marks[x] + "\t"); // prints elements in each pass
                                  }
                        }
        }
}
```

Output

```
Pass 1: 10   47  85  69  45  60  74  56  43
Pass 2: 10   43  85  69  45  60  74  56  47
Pass 3: 10   43  45  69  85  60  74  56  47
Pass 4: 10   43  45  47  85  60  74  56  69
Pass 5: 10   43  45  47  56  60  74  85  69
Pass 6: 10   43  45  47  56  60  74  85  69
Pass 7: 10   43  45  47  56  60  69  85  74
Pass 8: 10   43  45  47  56  60  69  74  85
```

Complexity analysis

Selection sort stops, when unsorted part becomes empty. On every step number of unsorted elements decreased by one.

Number of swaps may vary from zero (in case of sorted array) to n - 1 (in case array was sorted in reversed order).

Fact, that selection sort requires n - 1 number of swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

3.3. Bubble sort

The idea behind bubble sort is similar to the idea behind selection sort: on each pass through the algorithm, we place at least one item in its proper location.

The differences between bubble sort and selection sort lie in how many times data is swapped and when the algorithm terminates. Bubble sort performs more swaps in each pass

Like selection sort, bubble sort works by comparing two items in the list at a time.

Unlike selection sort, bubble sort will always compare two consecutive items in the list, and swap them if they are out of order.
If we assume that we start at the beginning of the list, this means that at each pass through the algorithm, the largest remaining item in the list will be placed at its proper location in the list.

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

Example 1

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57

Pass1: we have the following comparisons:

a) Compare A_1 and A_2. Since 32<51, the list is not altered

   32, 51, 27, 85, 66, 23, 13, 57

b) Compare A_2 and A_3. Since 51>27, interchange 51 and 27 as follows

   32, 27, 51, 85, 66, 23, 13, 57

c) Compare A_3 and A_4. Since 51<85, the list is not altered

   32, 27, 51, 85, 66, 23, 13, 57

d) Compare A_4 and A_5. Since 85>66, interchange 85 and 66 as follows

   32, 27, 51, 66, 85, 23, 13, 57

e) Compare A_5 and A_6. Since 85>23, interchange 85 and 23 as follows

   32, 27, 51, 66, 23, 85, 13, 57

f) Compare A_6 and A_7. Since 85>13, interchange 85 and 13 as follows

32, 27, 51, 66, 23, 13, 85, 57

g) Compare A_7 and A_8. Since 85>57, interchange 85 and 57 as follows

32, 27, 51, 66, 23, 13, 57, 85

At the end of the first pass, the largest number 85 has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions.

Pass2: only showing the interchanges

a) 27, 32, 51, 66, 23, 13, 57, 85
b) 27, 32, 51, 23, 66, 13, 57, 85
c) 27, 32, 51, 23, 13, 66, 57, 85
d) 27, 32, 51, 23, 13, 57, 66, 85

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.
Pass3: only showing the interchanges

a) 27, 32, 23, 51, 13, 57, 66, 85
b) 27, 32, 23, 13, 51, 57, 66, 85

Pass4: only showing the interchanges

a) 27, 23, 32, 13, 51, 57, 66, 85
b) 27, 23, 13, 32, 51, 57, 66, 85

Pass5: only showing the interchanges

a) 23, 27, 13, 32, 51, 57, 66, 85
b) 23, 13, 27, 32, 51, 57, 66, 85

Pass 6: 13, 23, 27, 32, 51, 57, 66, 85

Example 2

Write a java program to carry out a bubble sort on the following elements.

74,47,85,69,45,60,10,56,43

```
public class BubbleSort
{
        public static void main(String[ ] Args)
        {
        int[] Marks={74,47,85,69,45,60,10,56,43};
```

```
        int i,j, temp, x, y;// ind keeps position of the smallest element in each pass. others remain
the same
                for(i=0;i<9;i++)
                {
                 temp=Marks[0];
                    for(j=0;j<8;j++)
                        {
                                if(Marks[j+1]<temp)
                                {
                                 Marks[j]=Marks[j+1];
                                 Marks[j+1]=temp;
                                }
                                temp=Marks[j+1];
                        }
                        System.out.print("\nPass" + (i+1)+ ":");

                       for(x=0;x<9;x++)
                        {
                         System.out.print(Marks[x] + "\t");// prints elements in each pass
                        }
                        //Checks whether the elements are sorted if so the loop is terminated
                        y=0;
                        for (x=0;x<8;x++)
                        {
                                if(Marks[x]>Marks[x+1])
                                 {
                                        y=1; //y is set to 1 if the elements are not sorted
                                        break; // looping is terminated
                                }
                        }
                        if(y==0)
                        {
                        break;
                        }
                }
        }
}
```

Output

```
Pass 1: 47   74  69  45  60  10  56  43  85
Pass 2: 47   69  45  60  10  56  43  74  85
Pass 3: 47   45  60  10  56  43  69  74  85
Pass 4: 45   47  10  56  43  60  69  74  85
Pass 5: 45   10  47  43  56  60  69  74  85
Pass 6: 10   45  43  47  56  60  69  74  85
Pass 7: 10   43  45  47  56  60  69  74  85
```

4.0. NON-LINEAR STRUCTURES

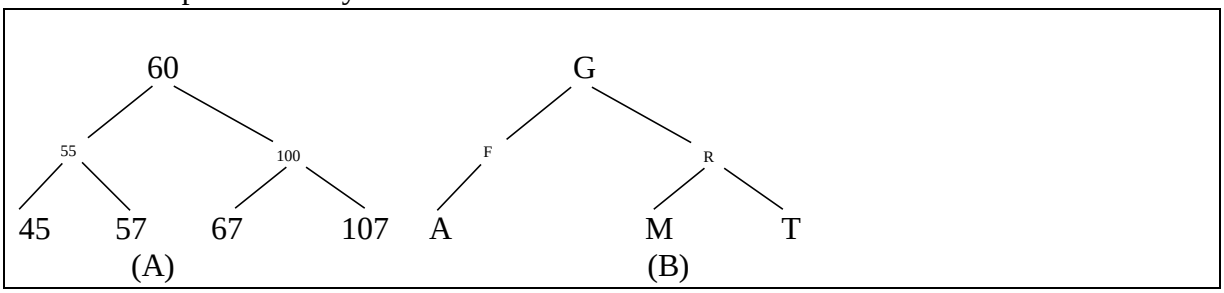
A non-linear data structure is a data structure in which a data item is connected to several other data items. Example are: graphs, trees, heaps

Trees

A tree is a collection of nodes called elements. A tree stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. The top element is called the root of the tree.

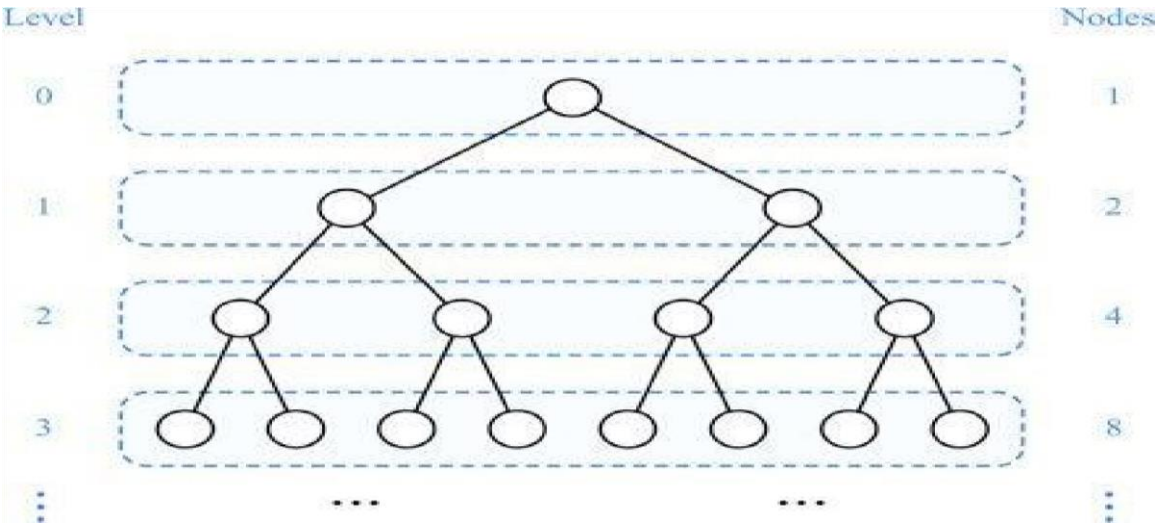
Binary trees

A binary tree is a hierarchical structure. It has a root node and 0-2 children. If it has 2 children, they are the left child and the right child. Each child is itself a tree, it may have a left child and a right child. Examples of binary trees are shown



A node without children is called a *leaf*.

- According to the definition of trees, a node can have any number of children.
- A binary tree is restricted to only having 0, 1, or at most 2 children.
- A complete binary tree is one where all the levels are full with exception to the last level and it is filled from left to right.
- A full binary tree is one where if a node has a child, then it has two children.



Differences between a general tree and binary tree

- A general tree is a **data structure** in that each node can have infinite number of children while a Binary tree is a data structure in that each node has at most **two nodes** left and right.
- Subtree of general tree are **not ordered** while Subtree of binary tree are **ordered**.

Binary search trees

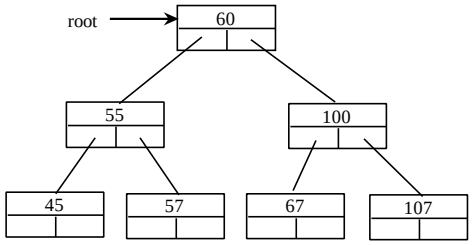
A special type of binary tree called a *binary search tree* is often useful. A binary search tree (with no duplicate elements) has the property that for every node in the tree the value of any node in its left subtree is less than the value of the node and the value of any node in its right subtree is greater than the value of the node.

Example:

Draw the binary search tree for the following inputs. 60, 25, 36, 45, 54, 20, 64.

Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure
below.



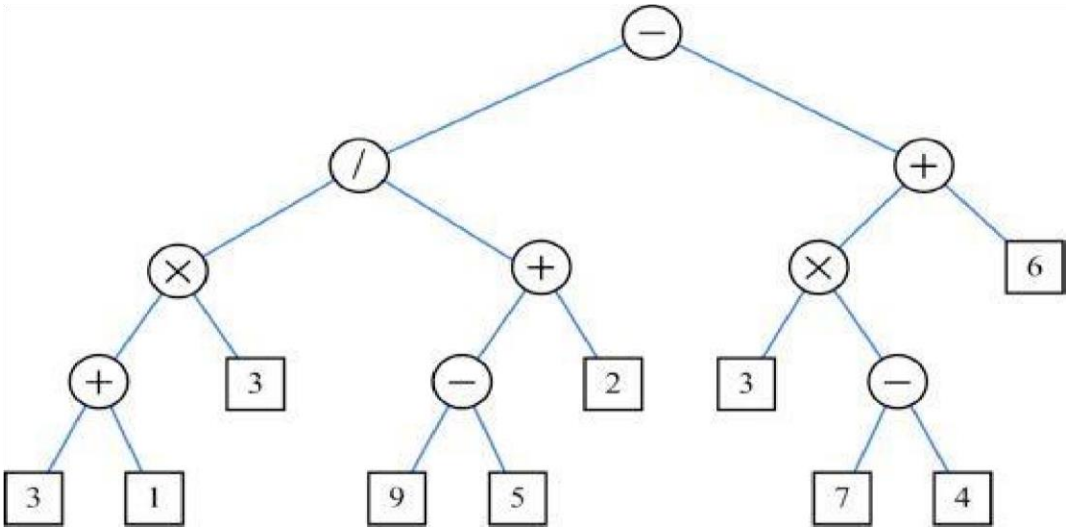

```
class TreeNode {
Object element;
  TreeNode left;
  TreeNode right;

  public TreeNode(Object o) {
    element = o;
  }
}
```

An arithmetic expression can be represented by a binary tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators, +, -, * and /. An arithmetic expression tree is a proper binary tree, since each operator, +, -, * and /, takes exactly two operands.

Example

A binary tree representing an arithmetic expression. The tree represents the expression

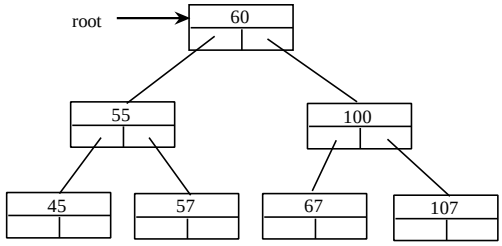
((((3 + 1) *3) / ((9 – 5) + 2)) – ((3 * (7 – 4)) + 6))



Inserting an Element to a Binary Tree

If a binary tree is empty, create a root node with the new element. Otherwise, locate the parent node for the new element node. If the new element is less than the parent element, the node for the new element becomes the left child of the parent. If the new element is greater than the parent element, the node for the new element becomes the right child of the parent. Here is the algorithm:

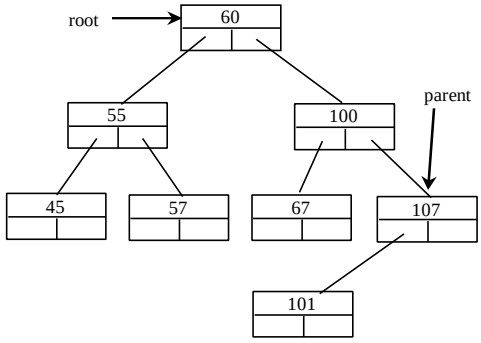
Insert 101 into the following tree.



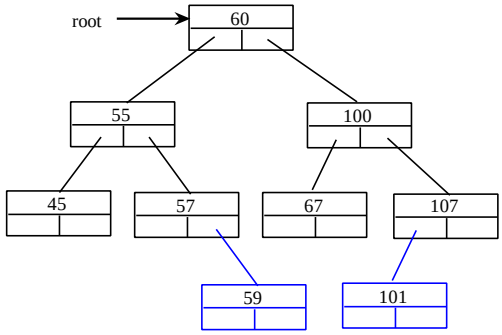

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
current = root;
while (current != null)
    if (element value < the value in current.element)
{       parent = current;
       current = current.left;
    }
    else if (element value > the value in current.element)
  {
        parent = current;
      current = current.right;
    }
```

```
    else
    return false; // Duplicate node not inserted
   // Create the new node and attach it to the parent node
if (element < parent.element)
  parent.left = new TreeNode(element);
else
parent.right = new TreeNode(element);
 return true; // Element inserted
}
```

The tree will be:



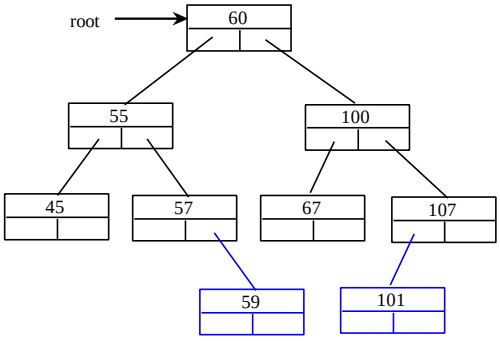
Inserting 59 into the Tree



Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree: ***inorder***, ***preorder***, ***postorder***, ***depth-first*** traversals.

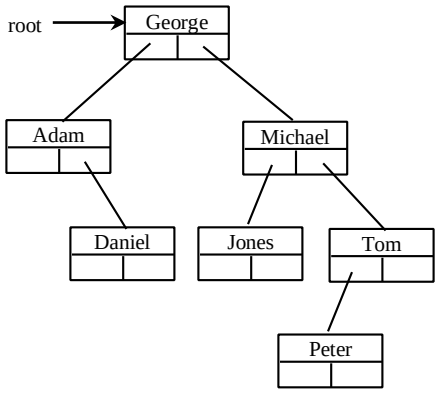
- The inorder traversal: visit the left subtree of the current node first, then the current node itself, and finally the right subtree of the current node (LNR).
- The preorder traversal: visit the current node first, then the left subtree of the current node, and finally the right subtree of the current node (NLR).
- The postorder traversal : visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself (LRN).
- The breadth-first traversal is to visit the nodes level by level. First visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on.



For example, in the tree in Figure above,

The inorder is: 45 55 57 59 60 67 100 101 107.
The postorder is: 45 59 57 55 67 101 107 100 60.
The preorder is: 60 55 45 57 59 100 67 107 101.
The breadth-first traversal is: 60 55 100 45 57 67 107 59 101.

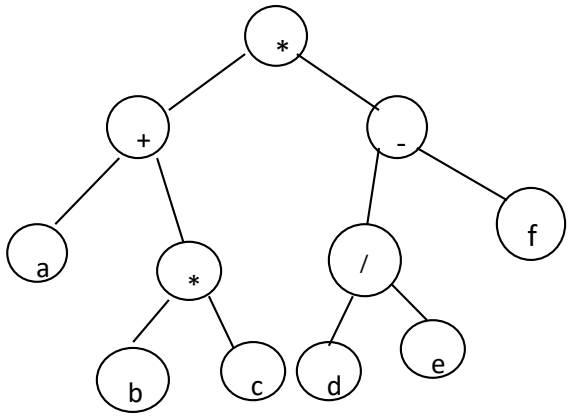
Traverse the following binary tree in inorder, postorder, and preorder



Inorder: Adam, Daniel George, Jones, Michael, Peter, Tom
Postorder: Daniel Adam, Jones, Peter, Tom, Michael, George
Preorder: George, Adam, Daniel, Michael, Jones, Tom, Peter

Example 1

Represent the following expression tree **(a+b*c)*(d/e-f)** and write the prefix and postfix expression.



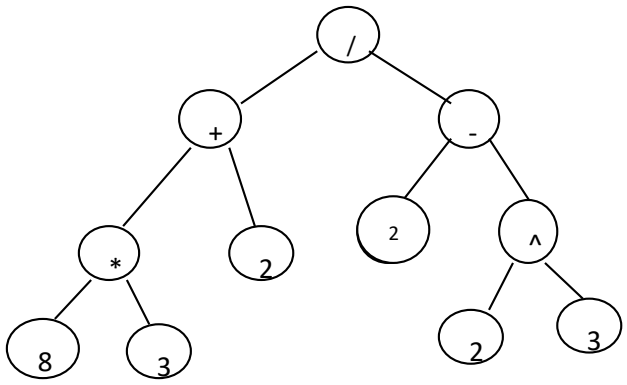
i) Prefix expression: *+a*bc-/def
ii) Postfix expression: abc*+de/f-*

Example 2

Draw a binary tree to evaluate the following fully parenthesized expression and evaluate it.

(((8*3) + 2)/ (21-(2^3)))

Binary Tree for the Above Expression



Evaluation of the fully parenthesized expression

(	(
(	((
(	(((
8	(((8
*	(((8*
3	(((8*3

)	(((8*3)
+	((24+
2	((24+2
)	((24+2)
/	(26/
(	(26/(
21	(26/(21
-	(26/(21-
(	(26/(21-(
2	(26/(21-(2
^	(26/(21-(2^
3	(26/(21-(2^3
)	(26/(21-(2^3)
)	(26/(21-8)
)	(26/13)
End Result	2

Hence, (((8*3)+2)/(21-(2^3))) = 2