Module 4- Linked Lists

Singly List Program to (a) Insert an element at front end, (b) Insert an element at rear end and (c) Insert an element at random, (d) Display elements and (e) count number of nodes in list.

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int data;
   struct node *next;
};

struct node *head;

void begin_insert()
{
       struct node *newnode;
   int item;
   newnode = (struct node *) malloc(sizeof(struct node *));
   if(newnode == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      newnode->data = item;
      newnode->next = head;
      head = newnode;
      printf("\nNode inserted");
   }

}

void last_insert()
{
   struct node *newnode,*temp;
   int item;
   newnode = (struct node*)malloc(sizeof(struct node));
   if(newnode == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter value?\n");
      scanf("%d",&item);
      newnode->data = item;

      if(head == NULL)
      {
         newnode-> next = NULL;
         head = newnode;
         printf("\nNode inserted");
      }
      else
      {
         temp = head;
         while (temp -> next != NULL)
         {
            temp = temp -> next;
         }
         temp->next = newnode;
         newnode->next = NULL;
         printf("\nNode inserted");

      }
   }
}

void random_insert()
{
   int i,loc,item;
   struct node *newnode, *temp;
   newnode = (struct node *) malloc (sizeof(struct node));
   if(newnode == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter element value");
      scanf("%d",&item);
      newnode->data = item;

      printf("\nEnter the location after which you want to insert ");
      scanf("\n%d",&loc);

      temp=head;
      for(i=0;i<loc;i++)
      {
         temp = temp->next;
         if(temp == NULL)
         {
            printf("\ncan't insert\n");
            return;
         }

      }
      newnode ->next = temp ->next;
      temp ->next = newnode;
      printf("\nNode inserted");
   }
}
void display()
{
   struct node *ptr;
   ptr = head;
   if(ptr == NULL)
   {
      printf("Nothing to print");
   }
   else
```

```
    {
        printf("\nprinting values . . . . .\n");
        while (ptr!=NULL)
        {
            printf("%d\t",ptr->data);
            ptr = ptr -> next;
        }
    }
}


//count() will count the nodes present in the list
void count() {
  int count = 0;
    //Node current will point to head
    struct node *current = head;

    while(current != NULL) {
        //Increment the count by 1 for each node
        count++;
        current = current->next;
    }
    printf("Count of nodes present in the list: %d", count);
}


void main()
{
    int choice =0;
    while(choice != 6)
    {
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4:Display\n5:count\n6:Exit\n");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:begin_insert();
                   break;
            case 2:last_insert();
                   break;
            case 3:random_insert();
                   break;
            case 4:display();
                   break;
            case 5:count();
                   break;
            case 6:exit(0);
                   break;

            default:
            printf("Please enter valid choice..");
        }

    }
}
```

Doubly List Program to (a) Insert an element at front end, (b) Insert an element at rear end and (c) Insert an element at random, (d) Display linked list elements and (e) count number of nodes in list.

Doubly Linked List Program

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void display();
void count();

void main ()
{
int choice =0;
    while(choice != 6)
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4.Display\n5.Count\n6.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            insertion_beginning();
            break;
            case 2:
                insertion_last();
            break;
            case 3:
            insertion_specified();
            break;
            case 4:
            display();
            break;
            case 5:
            count();
            break;
            case 6:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertion_beginning()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
       printf("\nOVERFLOW");
   }
   else
   {
    printf("\nEnter Item value");
    scanf("%d",&item);

   if(head==NULL)
   {
       ptr->next = NULL;
       ptr->prev=NULL;
       ptr->data=item;
       head=ptr;
   }
   else
   {
       ptr->data=item;
       ptr->prev=NULL;
       ptr->next = head;
       head->prev=ptr;
       head=ptr;
   }
   printf("\nNode inserted\n");
}

}
void insertion_last()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node));
   if(ptr == NULL)
   {
       printf("\nOVERFLOW");
   }
   else
   {
       printf("\nEnter value");
       scanf("%d",&item);
        ptr->data=item;
       if(head == NULL)
       {
           ptr->next = NULL;
           ptr->prev = NULL;
           head = ptr;
       }
       else
       {
          temp = head;
          while(temp->next!=NULL)
          {
              temp = temp->next;
          }
          temp->next = ptr;
          ptr ->prev=temp;
          ptr->next = NULL;
          }

       }
     printf("\nnode inserted\n");
    }
void insertion_specified()
{
   struct node *ptr,*temp;
   int item,loc,i;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
       printf("\n OVERFLOW");
   }
   else
   {
       temp=head;
       printf("Enter the location");
       scanf("%d",&loc);
       for(i=0;i<loc;i++)
       {
           temp = temp->next;
           if(temp == NULL)
           {
               printf("\n There are less than %d elements", loc);
               return;
           }
       }
       printf("Enter value");
       scanf("%d",&item);
       ptr->data = item;
       ptr->next = temp->next;
       ptr -> prev = temp;
       temp->next = ptr;
       temp->next->prev=ptr;
       printf("\nnode inserted\n");
   }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void count () {
    int counter = 0;
    //Node current will point to head
    struct node *current = head;

    while(current != NULL) {
        //Increment the counter by 1 for each node
        counter++;
        current = current->next;
    }
    printf("\nCount of nodes present in the list: %d", counter);
}

}
```

Module 5- Trees

A tree is **non-linear** and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the "children").

Basic Terminology In Tree Data Structure:

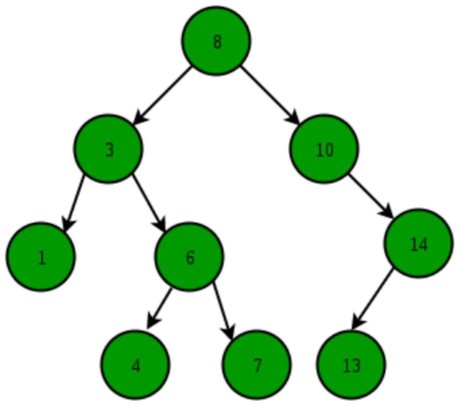
- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes.
- **Depth of a node:** The count of edges from the root to the node.
- **Height of a tree:** The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node.
- **Level of a node:** The count of edges on the path from the root node to that node.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant

Binary Search Tree (BST)

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.

- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

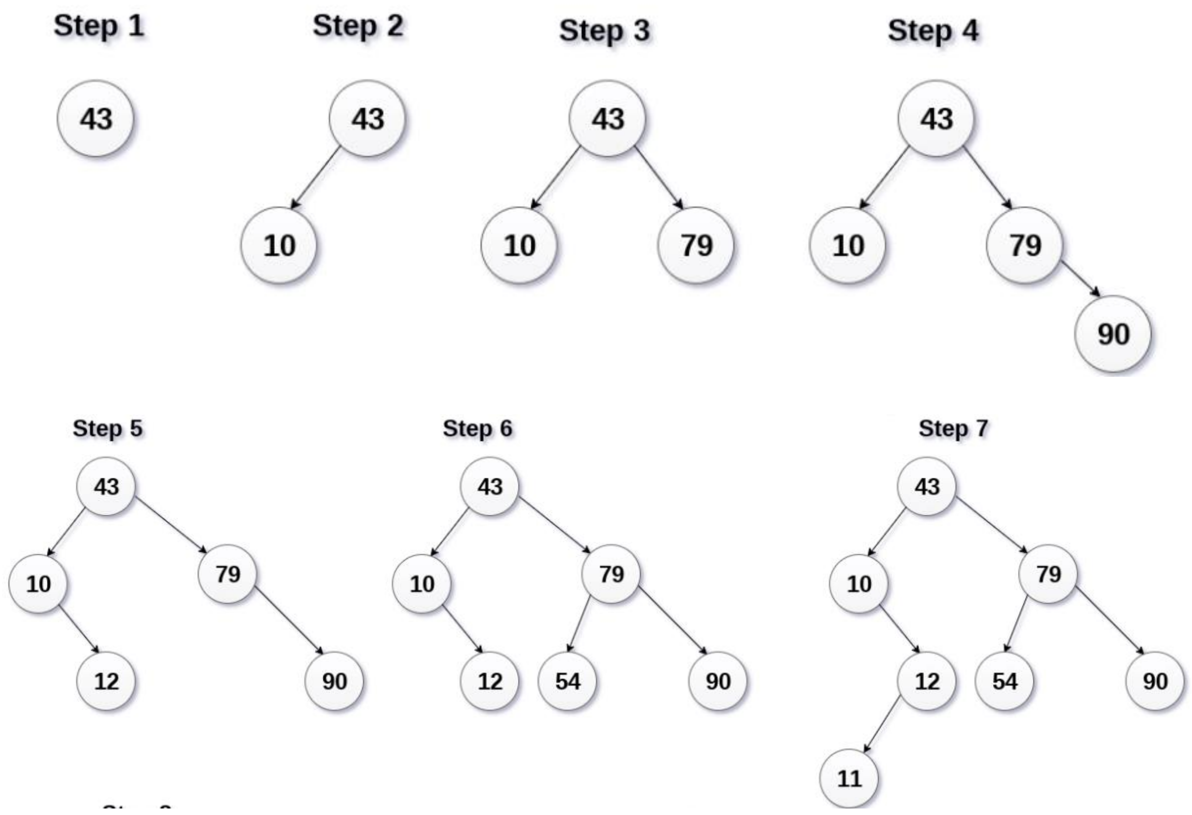


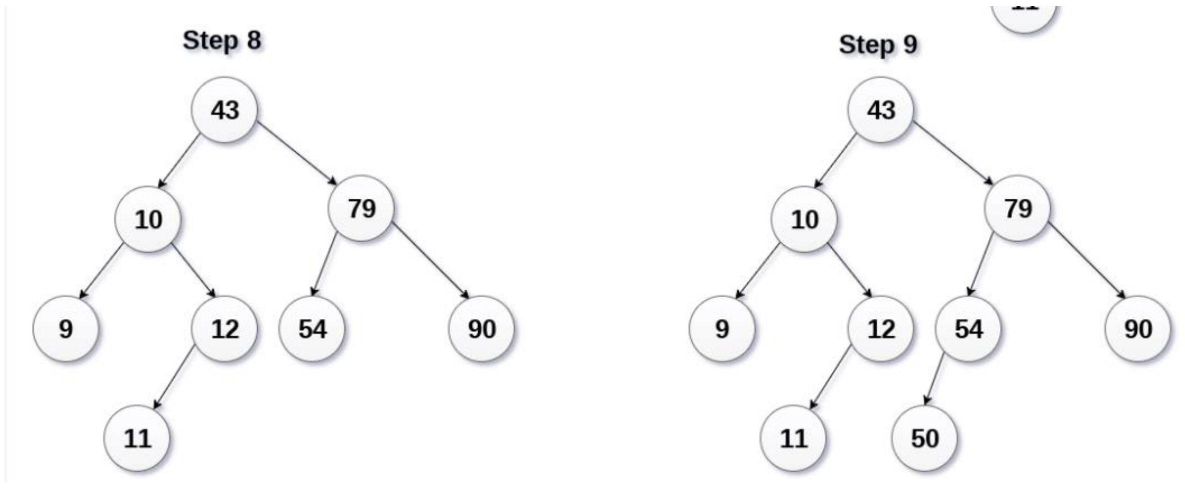
Creation of a Binary Search tree.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



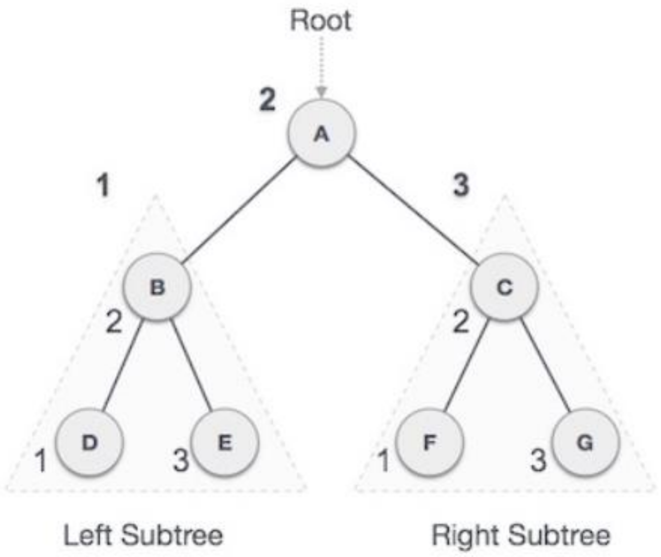


Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed in-order, **the output will produce sorted key values in an ascending order**.

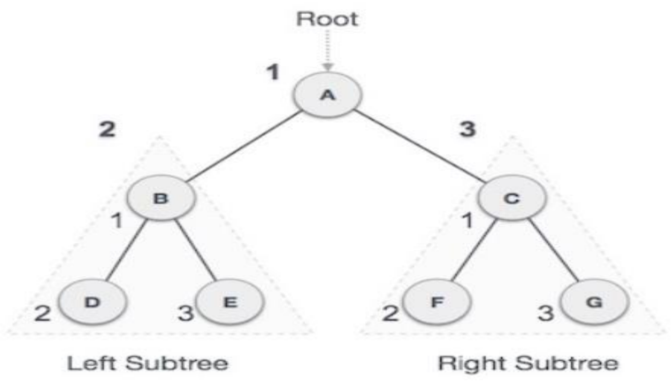


We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be −

D → B → E → A → F → C → G

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

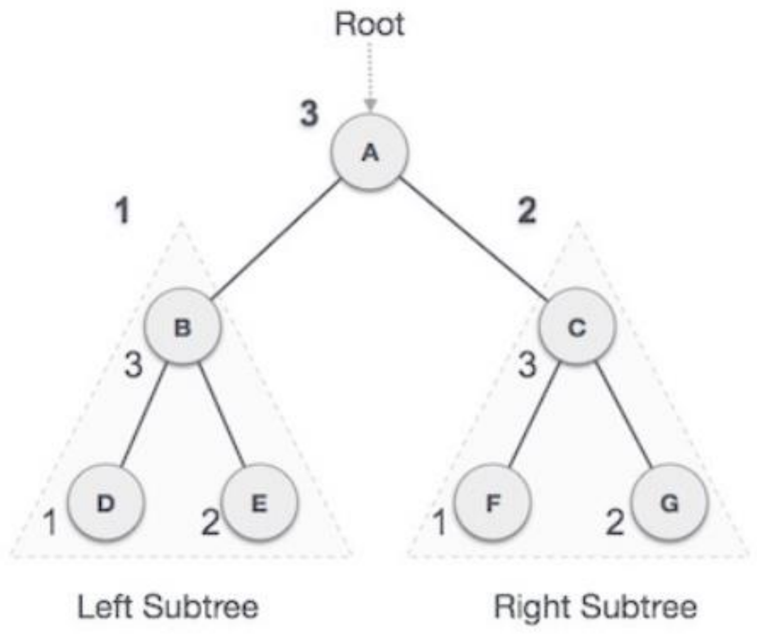


We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

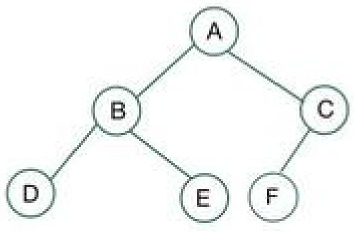
D → E → B → F → G → C → A

Complete Binary Tree

We know a **tree** is a non-linear data structure. It has no limitation on the number of children. A binary tree has a limitation as any node of the tree has at most two children: a left and a right child.

What is a Complete Binary Tree?

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.

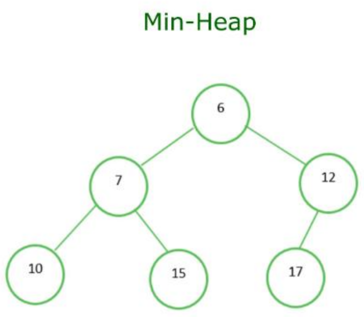


Heap

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

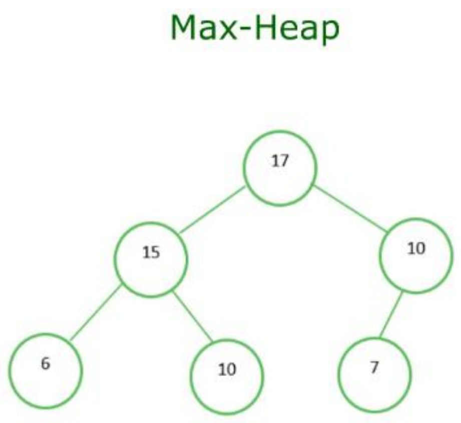
Min-Heap

In a Min-Heap the key present at the root node must be less than or equal among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. In a Min-Heap the minimum key element present at the root. Below is the Binary Tree that satisfies all the property of Min Heap.



Max Heap

In a Max-Heap the key present at the root node must be greater than or equal among the keys present at all of its children. The same property must be recursively **true** for all sub-trees in that Binary Tree. In a Max-Heap the maximum key element present at the root. Below is the Binary Tree that satisfies all the property of Max Heap.



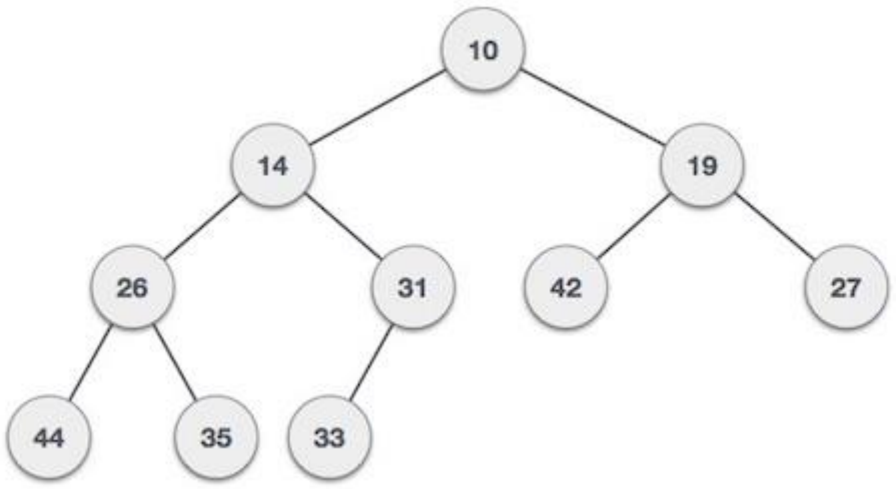
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

key(α) ≥ key(β)

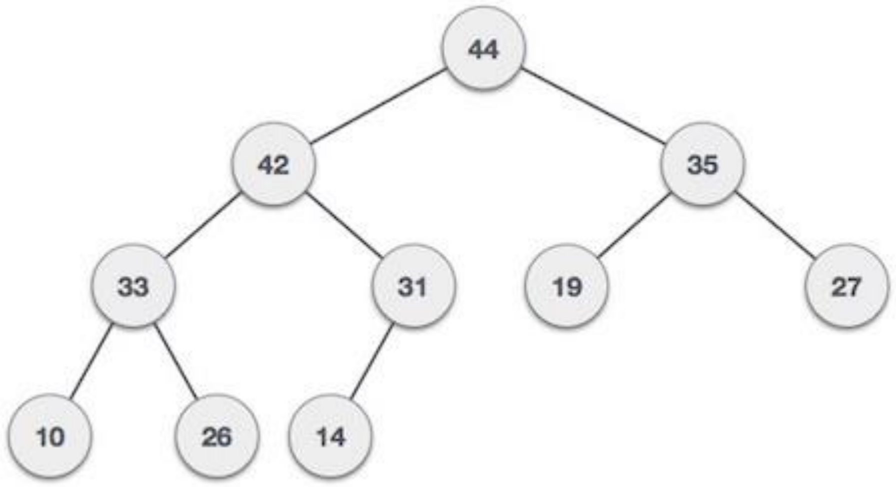
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

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

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



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



Both trees are constructed using the same input and order of arrival.