In [5]:

```
# Linear search
import array as arr
def lin_search(a,x):
    for i in range(len(a)):
        if a[i]==x:
            return (f"Element x is present at index {i}")
    else:
        print("Element x is not present in the arr[]")
arr1=arr.array('i',[10,20,80,30,60,50,110,100,130,170])
lin_search(arr1,100)
```

Out[5]:

```
'Element x is present at index 7'
```

In [2]:

```
# Binary search
import array as arr
def bin_search(a,x):
    s=0
    e=len(a)-1
    while s<=e:
        m=(s+e)//2
        if a[m]==x:
            return m
        elif a[m]>x:
            e=m-1
        elif a[m]<x:
            s=m+1

arr1=arr.array('i',[12,24,32,39,45,50,54])
b=bin_search(arr1,45)
if b==None:
    print("Element is not present in the array")
else:
    print(f"Element is present at index {b}")
```

```
Element is present at index 4
```

In [3]:

```
# Bubble sort
import array as arr
def bubble_sort(arr):
    for i in range(len(arr)-1):
        for j in range(len(arr)-1):
            if arr[j]>arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
    return arr
length=int(input("Enter the length of the array: "))
list1=[]
for i in range(length):
    c=int(input(f"Enter element-{i}: "))
    list1.append(c)
a=arr.array('i',list1)
# a=arr.array('i',[5,3,8,6,7,2])
print(f"The unsorted list is: {list(a)}")
print(f"The sorted list is: {list(bubble_sort(a))}")
```

```
Enter the length of the array: 6
Enter element-0: 5
Enter element-1: 3
Enter element-2: 8
Enter element-3: 6
Enter element-4: 7
Enter element-5: 2
The unsorted list is: [5, 3, 8, 6, 7, 2]
The sorted list is: [2, 3, 5, 6, 7, 8]
```

In [14]:

```
# Selection sort
import array as arr
def selection_sort(arr):
    for i in range(len(arr)):
        min_index=i
        for j in range(i+1,len(arr)):
            if arr[j]<arr[min_index]:
                min_index=j
        arr[min_index],arr[i]=arr[i],arr[min_index]
    return arr
length=int(input("Enter the length of the array: "))
list1=[]
for i in range(length):
    c=int(input(f"Enter element-{i}: "))
    list1.append(c)
a=arr.array('i',list1)
# a=arr.array('i',[747,45,88,11,-202,-97,-9,-2,0])
print(f"Given array is {list(a)}")
print(f"Sorted array is {list(selection_sort(a))}")
```

```
Enter the length of the array: 9
Enter element-0: 747
Enter element-1: 45
Enter element-2: 88
Enter element-3: 11
Enter element-4: -202
Enter element-5: -97
Enter element-6: -9
Enter element-7: -2
Enter element-8: 0
Given array is [747, 45, 88, 11, -202, -97, -9, -2, 0]
Sorted array is [-202, -97, -9, -2, 0, 11, 45, 88, 747]
```

In [15]:

```
# Merge sort
import array as arr
def merge_sort(arr):
    if len(arr)<=1:
        return arr
    else:
        m=len(arr)//2
        l_half=arr[:m]
        r_half=arr[m:]
        r_sort=merge_sort(r_half)
        l_sort=merge_sort(l_half)
        i,j=0,0
        result=[]
        while i<len(l_sort) and j<len(r_sort):
            if l_sort[i]<r_sort[j]:
                result.append(l_sort[i])
                i+=1
            else:
                result.append(r_sort[j])
                j+=1
        result.extend(l_sort[i:])
        result.extend(r_sort[j:])
        return result
length=int(input("Enter the length of the array: "))
list1=[]
for i in range(length):
    c=int(input(f"Enter element-{i}: "))
    list1.append(c)
a=arr.array('i',list1)
# a=arr.array('i',[12,11,13,5,6,7])
print(f"Given array is {list(a)}")
print(f"Sorted array is {list(merge_sort(a))}")
```

```
Enter the length of the array: 6
Enter element-0: 12
Enter element-1: 11
Enter element-2: 13
Enter element-3: 5
Enter element-4: 6
Enter element-5: 7
Given array is [12, 11, 13, 5, 6, 7]
Sorted array is [5, 6, 7, 11, 12, 13]
```

In [16]:

```
# Quick sort
import array as arr
def quick_sort(arr):
    if len(arr)<=1:
        return arr
    else:
        small,equal,large=[],[],[]
        pivot=arr[0]
        for i in arr:
            if i<pivot:
                small.append(i)
            elif i==pivot:
                equal.append(i)
            elif i>pivot:
                large.append(i)
        return quick_sort(small)+equal+quick_sort(large)
length=int(input("Enter the length of the array: "))
list1=[]
for i in range(length):
    c=int(input(f"Enter element-{i}: "))
    list1.append(c)
a=arr.array('i',list1)
# a=arr.array('i',[1,8,2,7,4,6,5])
print(f"The unsorted array is: {list(a)}")
print(f"The sorted array is: {quick_sort(a)}")
```

```
Enter the length of the array: 7
Enter element-0: 1
Enter element-1: 8
Enter element-2: 2
Enter element-3: 7
Enter element-4: 4
Enter element-5: 6
Enter element-6: 5
The unsorted array is: [1, 8, 2, 7, 4, 6, 5]
The sorted array is: [1, 2, 4, 5, 6, 7, 8]
```

In [11]:

```
class Stack:
    def __init__(self,n):
        self.arr=[]
        self.n=n
    def push(self,e):
        if len(self.arr)>self.n:
            raise Exception("Stack Overflow")
        else:
            self.arr.append(e)
    def pop(self):
        if len(self.arr)==0:
            raise Exception("Stack Underflow")
        else:
            return self.arr.pop()
    def top(self):
        if len(self.arr)==0:
            raise Exception("Stack Underflow")
        else:
            return self.arr[-1]
    def is_empty(self):
        return len(self.arr)==0
    def size(self):
        return len(self.arr)
s=Stack(3)
s.push('a')
s.push('b')
s.push('c')
print(f"Initial stack\n{s.arr}")
print(f"Elements popped from the stack:\n{s.pop()}\n{s.pop()}\n{s.pop()}")
print(f"Stack after elements are popped:\n{s.arr}")
```

```
Initial stack
['a', 'b', 'c']
Elements popped from the stack:
c
b
a
Stack after elements are popped:
[]
```

In [15]:

```
# Queue
class Queue():
    def __init__(self,n):
        self.queue=[None]*n
        self.front=self.rear=-1
        self.n=n
    def is_empty(self):
        return self.rear==-1
    def is_full(self):
        return (self.rear+1)%self.n==0
    def enqueue(self,e):
        if self.is_empty():
            self.front=self.rear=0
        elif self.is_full():
            print("Queue is full")  #raise Exception("Queue is full")
        else:
            self.rear+=1
        self.queue[self.rear]=e
    def dequeue(self):
        if self.is_empty():
            print("Queue is empty")  #raise Exception("Queue is empty")
        elif self.front==self.rear:
            temp=self.queue[self.front]
            self.queue[self.front]=None
            self.front=self.rear=-1
            return temp
        else:
            temp=self.queue[self.front]
            self.queue[self.front]=None
            self.front+=1
            return temp
    def size(self):
        count=0
        for i in self.queue:
            if i!=None:
                count+=1
        return count
    def Front(self):
        return self.queue[self.front]
q=Queue(4)
q.dequeue()
q.enqueue(20)
q.enqueue(30)
q.enqueue(40)
q.enqueue(50)
print(q.queue)
q.enqueue(50)
print(q.queue)
q.dequeue()
q.dequeue()
print("After two nodes deletion")
print(q.queue)
print(f"Front element is: {q.Front()}")
print(q.size())
```

```
Queue is empty
[20, 30, 40, 50]
Queue is full
[20, 30, 40, 50]
After two nodes deletion
[None, None, 40, 50]
Front element is: 40
2
```

In [16]:

```
# Circular Queue
class C_Queue():
    def __init__(self,n):
        self.queue=[None]*n
        self.front=self.rear=-1
        self.n=n
    def is_empty(self):
        return self.rear==-1
    def is_full(self):
        return (self.rear+1)%self.n==self.front
    def enqueue(self,e):
        if self.is_empty():
            self.front=self.rear=0
        elif self.is_full():
            print("Queue is full")  #raise Exception("Queue is full")
        else:
            self.rear=(self.rear+1)%self.n
        self.queue[self.rear]=e
    def dequeue(self):
        if self.is_empty():
            print("Queue is empty")  #raise Exception("Queue is empty")
        elif self.front==self.rear:
            temp=self.queue[self.front]
            self.queue[self.front]=None
            self.front=self.rear=-1
            return temp
        else:
            temp=self.queue[self.front]
            self.queue[self.front]=None
            self.front=(self.front+1)%self.n
            return temp
    def size(self):
        count=0
        for i in self.queue:
            if i!=None:
                count+=1
        return count
    def Front(self):
        return self.queue[self.front]
cq=C_Queue(5)
cq.enqueue(14)
cq.enqueue(22)
cq.enqueue(13)
cq.enqueue(-6)
print(f"Elements in the circular queue are: {cq.queue}")
print(f"Deleted value: {cq.dequeue()}")
print(f"Deleted value: {cq.dequeue()}")
print(f"Elements in the circular queue are: {cq.queue}")
cq.enqueue(9)
cq.enqueue(20)
cq.enqueue(5)
print(f"Elements in the circular queue are: {cq.queue}")
cq.enqueue(10)
print(cq.size())
```

```
Elements in the circular queue are: [14, 22, 13, -6, None]
Deleted value: 14
Deleted value: 22
Elements in the circular queue are: [None, None, 13, -6, None]
Elements in the circular queue are: [20, 5, 13, -6, 9]
Queue is full
5
```

In [3]:

```
# Singly linked list
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
class SLL:
    def __init__(self):
        self.head=None
    def insert_begin(self,data):
        new_node=Node(data)
        new_node.next=self.head
        self.head=new_node
    def insert_end(self,data):
        new_node=Node(data)
        if self.head==None:
            self.insert_begin(data)
        else:
            current_node=self.head
            while current_node:
                current_node=current_node.next
            current_node.next=new_node
    def insert_index(self,index,data):
        new_node=Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            prev_node=self.head
            c_p=0
            while c_p<(index-1) and prev_node:
                prev_node=prev_node.next
                c_p+=1
            new_node.next=prev_node.next
            prev_node.next=new_node
    def delete_begin(self):
        self.head=self.head.next
    def delete_end(self):
        last_node=self.head
        while last_node.next.next:
            last_node=last_node.next
        last_node.next=None
    def delete_index(self,index):
        if index==0:
            self.delete_begin()
        else:
            prev_node=self.head
            c_p=0
            while c_p<(index-1) and prev_node:
                prev_node=prev_node.next
                c_p+=1
            prev_node.next=prev_node.next.next
    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end="-->")
            current_node=current_node.next
ll=SLL()
ll.insert_begin("Wed")
ll.insert_begin("Tue")
ll.insert_begin("Mon")
ll.insert_begin("Sun")
print("Created Linked list is:")
ll.display()
```

```
Created Linked list is:
Sun-->Mon-->Tue-->Wed-->
```

In [1]:

```
# Doubly linked list
class Node():
    def __init__(self,data):
        self.data=data
        self.next=None
        self.prev=None
class DLL():
    def __init__(self):
        self.head=None
    def insert_begin(self,data):
        new_node=Node(data)
        if self.head!=None:
            self.head.prev=new_node
            new_node.next=self.head
            self.head=new_node
        else:
            self.head=new_node
    def insert_end(self,data):
        new_node=Node(data)
        if self.head!=None:
            last_node=self.head
            while last_node:
                last_node=last_node.next
            last_node.next=new_node
            new_node.prev=last_node
        else:
            self.head=new_node
    def insert_index(self,index,data):
        if index==0:
            self.insert_begin(data)
        else:
            prev_node=self.head
            c_p=0
            while c_p<(index-1) and prev_node:
                prev_node=prev_node.next
                c_p+=1
            prev_node.next.prev=new_node
            new_node.next=prev_node.next
            prev_node.next=new_node
            new_nod.prev=prev_node
    def delete_begin(self):
        if self.head!=None:
            self.head.next.prev=None
            self.head=self.head.next
    def delete_end(self):
        if self.head!=None:
            last_node=self.head
            while last_node.next.next!=None:
                last_node=last_node.next
            last_node.next.prev=None
            last_node.next=None
    def delete_index(self,index):
        if index==0:
            self.delete_begin()
        else:
            prev_node=self.head
            c_p=0
            while c_p<(index-1) and prev_node.next!=None:
                prev_node=prev_node.next
                c_p+=1
            prev_node.next.next.prev=prev_node
            prev_node.next=prev_node.next.next
    def display_forward(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end="-->")
            current_node=current_node.next

    def display_backward(self):
        last_node=self.head
        while last_node.next:
            last_node=last_node.next
        temp=last_node
        while temp.prev!=None:
            print(temp.data,end="-->")
            temp=temp.prev
        print(temp.data,end="-->") #To print the first node
dl=DLL()
dl.insert_begin(5)
dl.insert_begin(4)
dl.insert_begin(3)
dl.insert_begin(2)
dl.insert_begin(1)
dl.display_forward()
print()
dl.display_backward()
```

```
1-->2-->3-->4-->5-->
5-->4-->3-->2-->1-->
```

In [2]:

```
# Circular Singly Linked List
class Node():
    def __init__(self,data):
        self.data=data
        self.next=None
class Circular_Singly_Linked_List():
    def __init__(self):
        self.head=None
    def get_last_node(self):
        last_node=self.head
        while last_node.next!=self.head:
            last_node=last_node.next
        return last_node
    def insert_begin(self,data):
        new_node=Node(data)
        if self.head==None:
            self.head=new_node
            new_node.next=self.head
        else:
            new_node.next=self.head
            last_node=self.get_last_node()
            last_node.next=new_node
            self.head=new_node
    def insert_end(self,data):
        new_node=Node(data)
        if self.head==None:
            self.insert_begin(data)
        else:
            last_node=self.get_last_node()
            last_node.next=new_node
            new_node.next=self.head
    def insert_index(self,index,data):
        new_node=Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            current_node=self.head
            c_p=0
            while current_node.next!=self.head and c_p<(index-1):
                current_node=current_node.next
                c_p+=1
            new_node.next=current_node.next
            current_node.next=new_node
    def delete_begin(self):
        last_node=self.get_last_node()
        if self.head!=None:
            self.head=self.head.next
            last_node.next=self.head
    def delete_end(self):
        if self.head!=None:
            last_node=self.head
            while last_node.next.next!=self.head:
                last_node=last_node.next
            last_node.next=self.head
    def delete_index(self,index):
        if index==0:
            self.delete_begin()
        else:
            current_node=self.head
            c_p=0
            while current_node.next and c_p<(index-1):
                current_node=current_node.next
                c_p+=1
            current_node.next=current_node.next.next
    def display(self):
        current_node=self.head
        while current_node.next!=self.head:
            print(current_node.data,end="-->")
            current_node=current_node.next
        print(current_node.data) # To print the head

cll=Circular_Singly_Linked_List()
cll.insert_begin("Sat")
cll.insert_begin("Fri")
cll.insert_begin("Thu")
cll.insert_begin("Wed")
cll.insert_begin("Tue")
cll.insert_begin("Mon")
cll.insert_begin("Sun")
cll.display()
```

```
Sun-->Mon-->Tue-->Wed-->Thu-->Fri-->Sat
```

In [9]:

```
# Circular Doubly Linked List
class Node():
    def __init__(self,data):
        self.data=data
        self.next=None
        self.prev=None
class Circular_Doubly_Linked_List():
    def __init__(self):
        self.head=None
    def get_last_node(self):
        last_node=self.head
        while last_node.next!=self.head:
            last_node=last_node.next
        return last_node
    def insert_begin(self,data):
        new_node=Node(data)
        if self.head==None:
            self.head=new_node
            new_node.next=self.head
            new_node.prev=self.head
        else:
            new_node.next=self.head
            self.head.prev=new_node
            last_node=self.get_last_node()
            last_node.next=new_node
            new_node.prev=last_node
            self.head=new_node
    def insert_end(self,data):
        new_node=Node(data)
        if self.head==None:
            self.insert_begin(data)
        else:
            last_node=self.get_last_node()
            last_node.next=new_node
            new_node.prev=last_node
            new_node.next=self.head
            self.head.prev=new_node
    def insert_index(self,index,data):
        new_node=Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            current_node=self.head
            c_p=0
            while current_node.next and c_p<(index-1):
                current_node=current_node.next
                c_p+=1
            current_node.next.prev=new_node
            new_node.next=current_node.next
            current_node.next=new_node
            new_node.prev=current_node
    def delete_begin(self):
        last_node=self.get_last_node()
        if self.head!=None:
            last_node.next=self.head
            self.head.next.prev=last_node
            self.head=self.head.next
    def delete_end(self):
        if self.head!=None:
            last_node=self.get_last_node()
            last_node.prev.next=self.head
            self.head.prev=last_node.prev.next
    def delete_index(self,index):
        if index==0:
            self.delete_begin()
        else:
            current_node=self.head
            c_p=0
            while current_node.next!=self.head and c_p<(index-1):
                current_node=current_node.next
                c_p+=1
            current_node.next.next.prev=current_node
            current_node.next=current_node.next.next
    def display_forward(self):
        current_node=self.head
        while current_node.next!=self.head:
            print(current_node.data,end="-->")
            current_node=current_node.next
        print(current_node.data) #To print the last node

    def display_backward(self):
        last_node=self.head
        while last_node.next!=self.head:
            last_node=last_node.next
        temp=last_node
        while temp.prev!=self.head:
            print(temp.data,end="-->")
            temp=temp.prev
        print(temp.data,end="-->") # To print the second node
        print(temp.prev.data) # To print the first node

cdl=Circular_Doubly_Linked_List()
cdl.insert_begin("Sat")
cdl.insert_begin("Fri")
cdl.insert_begin("Thu")
cdl.insert_begin("Wed")
cdl.insert_begin("Tue")
cdl.insert_begin("Mon")
cdl.insert_begin("Sun")
cdl.display_forward()
cdl.display_backward()
```

```
Sun-->Mon-->Tue-->Wed-->Thu-->Fri-->Sat
Sat-->Fri-->Thu-->Wed-->Tue-->Mon-->Sun
```

In [10]:

```
# Implementation of stacks using linked lists
class Node():
    def __init__(self,data):
        self.data=data
        self.next=None
class stack():
    def __init__(self):
        self.head=None
    def push(self,data): #Similar to insert_begin() function in SLL
        new_node=Node(data)
        new_node.next=self.head
        self.head=new_node
    def pop(self): #Similar to the delete_begin() function in SLL
        if self.head==None:
            raise Exception("Stack Underflow")
        else:
            temp=self.head.data
            self.head=self.head.next
            return temp
    def top(self):
        return self.head.data
    def is_empty(self):
        return self.head==None
    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end="-->")
            current_node=current_node.next

s=stack()
s.push(11)
s.push(22)
s.push(33)
s.push(44)
s.push(55)
s.display()
```

```
55-->44-->33-->22-->11-->
```

In [6]:

```
# Implementation of queueus using linked lists
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
class Queues():
    def __init__(self):
        self.head=None
    def enqueue(self,data): #Similar to the insert_end() function in SLL
        new_node=Node(data)
        if self.head!=None:
            last_node=self.head
            while last_node.next:
                last_node=last_node.next
            last_node.next=new_node
        else:
            self.head=new_node
    def dequeue(self): #Similar to the delete_begin() function in SLL
        if self.head==None:
            raise Exception("Queue is empty")
        else:
            temp=self.head.data
            self.head=self.head.next
            return temp
    def front(self):
        return self.head.data

    def is_empty(self):
        return self.head==None

    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end="-->")
            current_node=current_node.next
q=Queues()
q.enqueue(50)
q.enqueue(40)
q.enqueue(30)
q.enqueue(20)
q.enqueue(10)
print(q.dequeue())
q.display()
```

```
50
40-->30-->20-->10-->
```

In [5]:

```
# Binary Search Tree
class Node():
    def __init__(self,key):
        self.key=key
        self.left=None
        self.right=None
root=None
def insert(root,key):
    if root is None:
        return Node(key)
    else:
        if key<root.key:
            root.left=insert(root.left,key)
        elif key>root.key:
            root.right=insert(root.right,key)
    return root

def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.key,end="-->")
    inorder(root.right)

def preorder(root):
    if root is None:
        return
    print(root.key,end="-->")
    preorder(root.left)
    preorder(root.right)

def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.key,end="-->")

def find_min_value_node(node):
    current=node
    while current.left is not None:
        current=current.left
    return current

def delete_node(root,key):
    if root is None:
        return root
    if key<root.key:
        root.left=delete_node(root.left,key)
    elif key>root.key:
        root.right=delete_node(root.right,key)
    else:
        if root.left is None:
            temp=root.right
            root=None
            return temp
        elif root.right is None:
            temp=root.left
            root=None
            return temp
        temp=find_min_value_node(root.right)
        root.key=temp.key
        root.right=delete_node(root.right,temp.key)
    return root

def search(root,key):
    if root is None or root.key==key:
        return root
    elif key<root.key:
        return search(root.left,key)
    elif key>root.key:
        return search(root.right,key)

root=insert(root,50)
insert(root,30)
insert(root,70)
insert(root,20)
insert(root,40)
insert(root,60)
insert(root,80)
print("Inorder traversal")
inorder(root)
print()
print("Preorder traversal")
preorder(root)
print()
print("Postorder traversal")
postorder(root)
print()
delete_node(root,30)
print("Inorder traversal after deleting a node")
inorder(root)
```

```
Inorder traversal
20-->30-->40-->50-->60-->70-->80-->
Preorder traversal
50-->30-->20-->40-->70-->60-->80-->
Postorder traversal
20-->40-->30-->60-->80-->70-->50-->
Inorder traversal after deleting a node
20-->40-->50-->60-->70-->80-->
```

In [1]:

```
# Binary Tree
class Node():
    def __init__(self,key):
        self.key=key
        self.left=None
        self.right=None
root=None
def insert(root,key):
    if root is None:
        return Node(key)
    else:
        queue=[root]
        while queue:
            current_node=queue.pop(0)
            if current_node.left is None:
                current_node.left=Node(key)
                break
            else:
                queue.append(current_node.left)
            if current_node.right is None:
                current_node.right=Node(key)
                break
            else:
                queue.append(current_node.right)

def delete_deepest(root,d_node):
    q=[root]
    while q:
        c_n=q.pop(0)
        if c_n is d_node:
            c_n=None
            return
        if c_n.left:
            if c_n.left is d_node:
                c_n.left=None
                return
            else:
                q.append(c_n.left)
        if c_n.right:
            if c_n.right is d_node:
                c_n.right=None
                return
            else:
                q.append(c_n.right)

def delete(root,key):
    if root==None:
        return None
    if root.left==None and root.right==None:
        if root.key==key:
            return None
        else:
            return root
    key_node=None
    c_n=None
    q=[root]
    while q:
        c_n=q.pop(0)
        if c_n.key==key:
            key_node=c_n
        if c_n.left:
            q.append(c_n.left)
        if c_n.right:
            q.append(c_n.right)
    Right_most_node=c_n
    if key_node is not None:
        temp=Right_most_node
        delete_deepest(root,Right_most_node)
        key_node.key=temp.key
    return root

def preorder(root):
    if root is None:
        return
    print(root.key,end="-->")
    preorder(root.left)
    preorder(root.right)

def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.key,end="-->")
    inorder(root.right)

def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.key,end="-->")

root=insert(root,1)
insert(root,2)
insert(root,3)
insert(root,4)
insert(root,5)
print("Preorder traversal of binary tree is\n")
preorder(root)
print("\n")
print("Inorder traversal of binary tree is\n")
inorder(root)
print("\n")
print("Postorder traversal of binary tree is\n")
postorder(root)
print()
delete(root,4)
print("Preorder traversal after deleting a node")
preorder(root)
```

```
Preorder traversal of binary tree is

1-->2-->4-->5-->3-->

Inorder traversal of binary tree is

4-->2-->5-->1-->3-->

Postorder traversal of binary tree is

4-->5-->2-->3-->1-->
Preorder traversal after deleting a node
1-->2-->5-->3-->
```

In [13]:

```
# Graphs
# Breadth First Search --> Analogous to Level order traversal
def bfs(graph,start):
    visited=set()
    queue=[start]
    visited.add(start)
    while queue:
        vertex=queue.pop(0)
        print(vertex,end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Depth First Search --> Analogous to inorder, preorder and postorder traversals
def dfs(graph,start):
    visited=set()
    stack=[start]
    while stack:
        vertex=stack.pop()
        if vertex not in visited:
            print(vertex,end=" ")
            visited.add(vertex)
            stack.extend(reversed(graph[vertex]))

# Example usage
graph={"A":["B","C"],"B":["A","D","E"],"C":["A","F"],"D":["B"],"E":["B","F"],"F":["C","E
start_vertex="A"
print("Breadth First Traversal: ",end="")
bfs(graph,start_vertex)
print()
print("Depth First Traversal: ",end="")
dfs(graph,start_vertex)
```

```
Breadth First Traversal: A B C D E F
Depth First Traversal: A B D E F C
```