1.Perform Linear Search on an array.

```
def linearSearch(array, n, x):

    # Going through array sequencially
    for i in range(0, n):
        if (array[i] == x):
            return i
    return -1


array = [2, 4, 0, 1, 9]
x = int(input())
n = len(array)
result = linearSearch(array, n, x)
if(result == -1):
    print("Element not found")
else:
    print("Element found at index: ", result)
```

```
4
Element found at index:  1
```

2.Perform Binary Search on a list stored in an array.

```
def binarySearch(array, x, low, high):
    while low <= high:

        mid = low + (high - low)//2

        if array[mid] == x:
            return mid

        elif array[mid] < x:
            low = mid + 1

        else:
            high = mid - 1

    return -1
array = [3, 4, 5, 6, 7, 8, 9]
x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```

```
Element is present at index 1
```

3.Develop a program to implement bubble sort technique.

```
def bubbleSort(array):
  for i in range(len(array)):

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):
      if array[j] > array[j + 1]:
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp


data = [-2, 45, 0, 11, -9]

bubbleSort(data)

print('Sorted Array in Ascending Order:')
print(data)
```

```
Sorted Array in Ascending Order:
[-9, -2, 0, 11, 45]
```

4.Develop a program to implement selection sort technique.

```
def selectionSort(array, size):
   for step in range(size):
        min_idx = step
        for i in range(step + 1, size):
            if array[i] < array[min_idx]:
                min_idx = i
        (array[step], array[min_idx]) = (array[min_idx], array[step])
data = [-2, 45, 0, 11, -9]
size = len(data)
selectionSort(data, size)
print(data)
```

```
[-9, -2, 0, 11, 45]
```

5.Develop a program to implement insertion sort technique.

```
def insertionSort(array):
  for step in range(1, len(array)):
    key = array[step]
    j = step - 1
    while j >= 0 & key < array[j]:
      array[j + 1] = array[j]
      j = j - 1
    array[j + 1] = key
data = [9, 5, 1, 4, 3]
insertionSort(data)
print("InsertionSort", data)
```

```
InsertionSort [3, 4, 1, 5, 9]
```

6.Develop a program to implement quick sort technique.

```
def partition(array, low, high):
  pivot = array[high]

  # pointer for greater element
  i = low - 1
  for j in range(low, high):
    if array[j] <= pivot:
      i = i + 1

      # swapping element at i with element at j
      (array[i], array[j]) = (array[j], array[i])

  # swap the pivot element with the greater element specified by i
  (array[i + 1], array[high]) = (array[high], array[i + 1])

  # return the position from where partition is done
  return i + 1

# function to perform quicksort
def quickSort(array, low, high):
  if low < high:
    pi = partition(array, low, high)

    # recursive call on the left of pivot
    quickSort(array, low, pi - 1)

    # recursive call on the right of pivot
    quickSort(array, pi + 1, high)


data = [8, 7, 2, 1, 0, 9, 6]
print("Unsorted Array")
print(data)

size = len(data)
```

```
quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

```
Unsorted Array
[8, 7, 2, 1, 0, 9, 6]
Sorted Array in Ascending Order:
[0, 1, 2, 6, 7, 8, 9]
```

7.Develop a program to implement merge sort technique.

```
def merge_sort(unsorted_array):
    if len(unsorted_array) > 1:
        mid = len(unsorted_array) // 2  # Finding the mid of the array
        left = unsorted_array[:mid]  # Dividing the array elements
        right = unsorted_array[mid:]  # into 2 halves

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0

        #  data to temp arrays L[] and R[]
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                unsorted_array[k] = left[i]
                i += 1
            else:
                unsorted_array[k] = right[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(left):
            unsorted_array[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            unsorted_array[k] = right[j]
            j += 1
            k += 1


def print_list(array1):
    for i in range(len(array1)):
        print(array1[i], end=" ")
    print()

if __name__ == '__main__':
    array = [20, 30, 60, 40, 10, 50]
    print("Given array is", end="\n")
    print_list(array)
    merge_sort(array)
    print("Sorted array is: ", end="\n")
    print_list(array)
```

```
Given array is
20 30 60 40 10 50
Sorted array is:
10 20 30 40 50 60
```

8. Design a program to create a singly linked list for the following operations

#• Insert a Node at Beginning, at Ending and at a given Position

#• Delete a Node at Beginning, at Ending and at a given Position

#• Search, Count the Number of Nodes and Display

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkedList:

    def __init__(self):
        self.head = None

    # Insert at the beginning
    def insertAtBeginning(self, new_data):
        new_node = Node(new_data)

        new_node.next = self.head
        self.head = new_node

    # Insert after a node
    def insertAfter(self, prev_node, new_data):

        if prev_node is None:
            print("The given previous node must inLinkedList.")
            return

        new_node = Node(new_data)
        new_node.next = prev_node.next
        prev_node.next = new_node

    # Insert at the end
    def insertAtEnd(self, new_data):
        new_node = Node(new_data)

        if self.head is None:
            self.head = new_node
            return

        last = self.head
        while (last.next):
            last = last.next

        last.next = new_node

    # Deleting a node
    def deleteNode(self, position):

        if self.head is None:
            return

        temp = self.head

        if position == 0:
            self.head = temp.next
            temp = None
            return

        # Find the key to be deleted
        for i in range(position - 1):
            temp = temp.next
            if temp is None:
                break

        # If the key is not present
        if temp is None:
            return

        if temp.next is None:
            return

        next = temp.next.next

        temp.next = None

        temp.next = next

    # Search an element
    def search(self, key):

        current = self.head

        while current is not None:
```

```
            if current.data == key:
                return True

            current = current.next

        return False

    # Sort the linked list
    def sortLinkedList(self, head):
        current = head
        index = Node(None)

        if head is None:
            return
        else:
            while current is not None:
                # index points to the node next to current
                index = current.next

                while index is not None:
                    if current.data > index.data:
                        current.data, index.data = index.data, current.data

                    index = index.next
                current = current.next

    # Print the linked list
    def printList(self):
        temp = self.head
        while (temp):
            print(str(temp.data) + " ", end="")
            temp = temp.next


if __name__ == '__main__':

    llist = LinkedList()
    llist.insertAtEnd(1)
    llist.insertAtBeginning(2)
    llist.insertAtBeginning(3)
    llist.insertAtEnd(4)
    llist.insertAfter(llist.head.next, 5)

    print('linked list:')
    llist.printList()

    print("\nAfter deleting an element:")
    llist.deleteNode(3)
    llist.printList()

    print()
    item_to_find = 3
    if llist.search(item_to_find):
        print(str(item_to_find) + " is found")
    else:
        print(str(item_to_find) + " is not found")

    llist.sortLinkedList(llist.head)
    print("Sorted List: ")
    llist.printList()
```

```
linked list:
3 2 5 1 4
After deleting an element:
3 2 5 4
3 is found
Sorted List:
2 3 4 5
```

9.Design a program to create a doubly linked list for the following operations

#• Insert a Node at Beginning, at Ending and at a given Position

#• Delete a Node at Beginning, at Ending and at a given Position

#• Search, Count the Number of Nodes and Display

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None


class DoublyLinkedList:

    def __init__(self):
        self.head = None

    # insert node at the front
    def insert_front(self, data):

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)

        # make newNode as a head
        new_node.next = self.head

        # assign null to prev (prev is already none in the constructore)

        # previous of head (now head is the second node) is newNode
        if self.head is not None:
            self.head.prev = new_node

        # head points to newNode
        self.head = new_node

    # insert a node after a specific node
    def insert_after(self, prev_node, data):

        # check if previous node is null
        if prev_node is None:
            print("previous node cannot be null")
            return

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)

        # set next of newNode to next of prev node
        new_node.next = prev_node.next

        # set next of prev node to newNode
        prev_node.next = new_node

        # set prev of newNode to the previous node
        new_node.prev = prev_node

        # set prev of newNode's next to newNode
        if new_node.next:
            new_node.next.prev = new_node

    # insert a newNode at the end of the list
    def insert_end(self, data):

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)

        # assign null to next of newNode (already done in constructor)

        # if the linked list is empty, make the newNode as head node
        if self.head is None:
            self.head = new_node
            return

        # store the head node temporarily (for later use)
        temp = self.head

        # if the linked list is not empty, traverse to the end of the linked list
        while temp.next:
            temp = temp.next

        # now, the last node of the linked list is temp
```

```
        # assign next of the last node (temp) to newNode
        temp.next = new_node

        # assign prev of newNode to temp
        new_node.prev = temp

        return

    # delete a node from the doubly linked list
    def deleteNode(self, dele):

        # if head or del is null, deletion is not possible
        if self.head is None or dele is None:
            return

        # if del_node is the head node, point the head pointer to the next of del_node
        if self.head == dele:
            self.head = dele.next

        # if del_node is not at the last node, point the prev of node next to del_node to the previous of del_node
        if dele.next is not None:
            dele.next.prev = dele.prev

        # if del_node is not the first node, point the next of the previous node to the next node of del_node
        if dele.prev is not None:
            dele.prev.next = dele.next

        # free the memory of del_node
        gc.collect()

    # print the doubly linked list
    def display_list(self, node):

        while node:
            print(node.data, end="->")
            last = node
            node = node.next


# initialize an empty node
d_linked_list = DoublyLinkedList()

d_linked_list.insert_end(5)
d_linked_list.insert_front(1)
d_linked_list.insert_front(6)
d_linked_list.insert_end(9)

# insert 11 after head
d_linked_list.insert_after(d_linked_list.head, 11)

# insert 15 after the seond node
d_linked_list.insert_after(d_linked_list.head.next, 15)

d_linked_list.display_list(d_linked_list.head)

# delete the last node
d_linked_list.deleteNode(d_linked_list.head.next.next.next.next.next)

print()
d_linked_list.display_list(d_linked_list.head)
```

```
6->11->15->1->5->9->
6->11->15->1->5->
```

10.Create a Circular singly linked list for adding and deleting a Node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class CircularLinkedList:
    def __init__(self):
        self.last = None
```

```
    def addToEmpty(self, data):

        if self.last != None:
            return self.last

        # allocate memory to the new node and add data to the node
        newNode = Node(data)

        # assign last to newNode
        self.last = newNode

        # create link to iteself
        self.last.next = self.last
        return self.last

    # add node to the front
    def addFront(self, data):

        # check if the list is empty
        if self.last == None:
            return self.addToEmpty(data)

        # allocate memory to the new node and add data to the node
        newNode = Node(data)

        # store the address of the current first node in the newNode
        newNode.next = self.last.next

        # make newNode as last
        self.last.next = newNode

        return self.last

    # add node to the end
    def addEnd(self, data):
        # check if the node is empty
        if self.last == None:
            return self.addToEmpty(data)

        # allocate memory to the new node and add data to the node
        newNode = Node(data)

        # store the address of the last node to next of newNode
        newNode.next = self.last.next

        # point the current last node to the newNode
        self.last.next = newNode

        # make newNode as the last node
        self.last = newNode

        return self.last

    # insert node after a specific node
    def addAfter(self, data, item):

        # check if the list is empty
        if self.last == None:
            return None

        newNode = Node(data)
        p = self.last.next
        while p:

            # if the item is found, place newNode after it
            if p.data == item:

                # make the next of the current node as the next of newNode
                newNode.next = p.next

                # put newNode to the next of p
                p.next = newNode

                if p == self.last:
                    self.last = newNode
                    return self.last
                else:
```

```
                return self.last
            p = p.next
            if p == self.last.next:
                print(item, "The given node is not present in the list")
                break

    # delete a node
    def deleteNode(self, last, key):

        # If linked list is empty
        if last == None:
            return

        # If the list contains only a single node
        if (last).data == key and (last).next == last:

            last = None

        temp = last
        d = None

        # if last node is to be deleted
        if (last).data == key:

            # find the node before the last node
            while temp.next != last:
                temp = temp.next

            # point temp node to the next of last i.e. first node
            temp.next = (last).next
            last = temp.next

        # travel to the node to be deleted
        while temp.next != last and temp.next.data != key:
            temp = temp.next

        # if node to be deleted was found
        if temp.next.data == key:
            d = temp.next
            temp.next = d.next

        return last

    def traverse(self):
        if self.last == None:
            print("The list is empty")
            return

        newNode = self.last.next
        while newNode:
            print(newNode.data, end=" ")
            newNode = newNode.next
            if newNode == self.last.next:
                break


# Driver Code
if __name__ == "__main__":

    cll = CircularLinkedList()

    last = cll.addToEmpty(6)
    last = cll.addEnd(8)
    last = cll.addFront(2)
    last = cll.addAfter(10, 2)

    cll.traverse()

    last = cll.deleteNode(last, 8)
    print()
    cll.traverse()
```

```
2 10 6 8
2 10 6
```

11.Create a stack and perform various operations on it.

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()
s = Stack()
while True:
    print('push <value>')
    print('pop')
    print('quit')
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'push':
        s.push(int(do[1]))
    elif operation == 'pop':
        if s.is_empty():
            print('Stack is empty.')
        else:
            print('Popped value: ', s.pop())
    elif operation == 'quit':
        break
print('\nElements popped from stack:')
print(s.pop())
print(s.pop())
print(f"Stack:{s}")
```

```
push <value>
pop
quit
What would you like to do? push 1
push <value>
pop
quit
What would you like to do? push 2
push <value>
pop
quit
What would you like to do? push 3
push <value>
pop
quit
What would you like to do? push 4
push <value>
pop
quit
What would you like to do? pop 4
Popped value:  4
push <value>
pop
quit
What would you like to do? pop 2
Popped value:  3
push <value>
pop
quit
What would you like to do? quit

Elements popped from stack:
2
1
Stack:<__main__.Stack object at 0x7f7f2c8d4a10>
```

12.Convert the infix expression into postfix form.

```
class infix_to_postfix:
    precedence={'^':5,'*':4,'/':4,'+':3,'-':3,'(':2,')':1}
    def __init__(self):
        self.items=[]
        self.size=-1
    def push(self,value):
```

```
        self.items.append(value)
        self.size+=1
    def pop(self):
        if self.isempty():
            return 0
        else:
            self.size-=1
            return self.items.pop()
    def isempty(self):
        if(self.size==-1):
            return True
        else:
            return False
    def seek(self):
        if self.isempty():
            return False
        else:
            return self.items[self.size]
    def isOperand(self,i):
        if i in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            return True
        else:
            return False
    def infixtopostfix (self,expr):
        postfix=""
        print('postfix expression after every iteration is:')
        for i in expr:
            if(len(expr)%2==0):
                print("Incorrect infix expr")
                return False
            elif(self.isOperand(i)):
                postfix +=i
            elif(i in '+-*/^'):
                while(len(self.items)and self.precedence[i]<=self.precedence[self.seek()]):
                    postfix+=self.pop()
                self.push(i)
            elif i is '(':
                self.push(i)
            elif i is ')':
                o=self.pop()
                while o!='(':
                    postfix +=o
                    o=self.pop()
            print(postfix)
                #end of for
        while len(self.items):
            if(self.seek()=='('):
                self.pop()
            else:
                postfix+=self.pop()
        return postfix
s=infix_to_postfix()
expr=input('enter the expression ')
result=s.infixtopostfix(expr)
if (result!=False):
    print("the postfix expr of :",expr,"is",result)
```

```
enter the expression G+A+(U-R)^I
postfix expression after every iteration is:
G
G
GA
GA+
GA+
GA+U
GA+U
GA+UR
GA+UR-
GA+UR-
GA+UR-I
the postfix expr of : G+A+(U-R)^I is GA+UR-I^+
```

13.Perform String reversal using stack

```
class  Stack_to_reverse  :
    # Creates  an  empty  stack.
    def __init__(  self  ):
```

```
        self.items  =  list()
        self.size=-1
    #Returns  True  if  the  stack  is  empty  or  False  otherwise.
    def  isEmpty(  self  ):
        if(self.size==-1):
            return True
        else:
            return False
    # Removes  and  returns  the  top  item  on  the  stack.
    def  pop(  self  ):
        if  self.isEmpty():
            print("Stack is empty")
        else:
            return self.items.pop()
            self.size-=1

    # Push  an  item  onto  the  top  of  the  stack.
    def  push(  self,  item  ):
        self.items.append(item)
        self.size+=1

    def reverse(self,string):
        n = len(string)
 # Push all characters of string to stack
        for i in range(0,n):
            S.push(string[i])

 # Making the string empty since all characters are saved in stack
        string=""

 # Pop all characters of string and put them back to string
        for i in range(0,n):
            string+=S.pop()
        return string
S=Stack_to_reverse()
seq=input("Enter a string to be reversed:")
sequence = S.reverse(seq)
print("Reversed string is: " + sequence)
```

```
Enter a string to be reversed:Tharun
Reversed string is: nurahT
```

14.Evaluation of postfix expression

```
class evaluate_postfix:
    def __init__(self):
        self.items=[]
        self.size=-1
    def isEmpty(self):
        return self.items==[]
    def push(self,item):
        self.items.append(item)
        self.size+=1
    def pop(self):
        if self.isEmpty():
            return 0
        else:
            self.size-=1
            return self.items.pop()
    def seek(self):
        if self.isEmpty():
            return False
        else:
            return self.items[self.size]
    def evalute(self,expr):
        for i in expr:
            if i in '0123456789':
                self.push(i)
            else:
                op1=self.pop()
                op2=self.pop()
                result=self.cal(op2,op1,i)
                self.push(result)
        return self.pop()
    def cal(self,op2,op1,i):
        if i is '*':
```

```
            return int(op2)*int(op1)
        elif i is '/':
            return int(op2)/int(op1)
        elif i is '+':
            return int(op2)+int(op1)
        elif i is '-':
            return int(op2)-int(op1)
        elif i is '%':
            return int(op2)%int(op1)
s=evaluate_postfix()
expr=input('enter the postfix expression')
value=s.evalute(expr)
print('the result of postfix expression',expr,'is',value)
```

```
enter the postfix expression56/45*23++
the result of postfix expression 56/45*23++ is 25
```

15.Create a queue and perform various operations on it.

```
# Queue implementation in Python
class Queue:

    def __init__(self):
        self.queue = []

    # Add an element
    def enqueue(self, item):
        self.queue.append(item)

    # Remove an element
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    # Display  the queue
    def display(self):
        print(self.queue)

    def size(self):
        return len(self.queue)
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")
q.display()
```

```
[1, 2, 3, 4, 5]
After removing an element
[2, 3, 4, 5]
```

16.Construct a binary tree and perform various traversals.

```
class Node:
    def __init__(self, item):
        self.left = None
        self.right = None
        self.val = item
def inorder(root):

    if root:
        # Traverse left
        inorder(root.left)
        # Traverse root
        print(str(root.val) + "->", end='')
        # Traverse right
        inorder(root.right)
```

```
def postorder(root):

    if root:
        # Traverse left
        postorder(root.left)
        # Traverse right
        postorder(root.right)
        # Traverse root
        print(str(root.val) + "->", end='')
def preorder(root):

    if root:
        print(str(root.val) + "->", end='')
        preorder(root.left)
        preorder(root.right)
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print("Inorder traversal ")
inorder(root)
print("\nPreorder traversal ")
preorder(root)
print("\nPostorder traversal ")
postorder(root)
```

```
Inorder traversal
4->2->5->1->3->
Preorder traversal
1->2->4->5->3->
Postorder traversal
4->5->2->3->1->
```

17.Construct a binary search tree and perform a search operation.

```
class GFG :
    def main( args) :
        tree = BST()
        tree.insert(25)
        tree.insert(60)
        tree.insert(75)
        tree.insert(20)
        tree.insert(10)
        tree.insert(30)
        tree.insert(60)
        tree.inorder()
class Node :
    left = None
    val = 0
    right = None
    def __init__(self, val) :
        self.val = val
class BST :
    root = None
    def insert(self, key) :
        node = Node(key)
        if (self.root == None) :
            self.root = node
            return
        prev = None
        temp = self.root
        while (temp != None) :
            if (temp.val > key) :
                prev = temp
                temp = temp.left
            elif(temp.val < key) :
                prev = temp
                temp = temp.right
        if (prev.val > key) :
            prev.left = node
        else :
            prev.right = node
    def inorder(self) :
        temp = self.root
        stack = []
```

```
        while (temp != None or not (len(stack) == 0)) :
            if (temp != None) :
                stack.append(temp)
                temp = temp.left
            else :
                temp = stack.pop()
                print(str(temp.val) + " ", end ="")
                temp = temp.right

if __name__=="__main__":
    GFG.main([])
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-3-d7e1d174c03e> in <module>
     49
     50 if __name__=="__main__":
---> 51         GFG.main([])

1 frames

<ipython-input-3-d7e1d174c03e> in insert(self, key)
     26                 temp = self.root
     27                 while (temp != None) :
---> 28                         if (temp.val > key) :
     29                                 prev = temp
     30                                 temp = temp.left

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW

19.Implement Dijkstra's Shortest Path Algorithm

```
import heapq


def calculate_distances(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0

    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)

        # Nodes can get added to the priority queue multiple times. We only
        # process a vertex the first time we remove it from the priority queue.
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # Only consider this new path if it's better than any path we've
            # already found.
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances


example_graph = {
    'U': {'V': 2, 'W': 5, 'X': 1},
    'V': {'U': 2, 'X': 2, 'W': 3},
    'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5},
    'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1},
    'Y': {'X': 1, 'W': 1, 'Z': 1},
    'Z': {'W': 5, 'Y': 1},
}
print(calculate_distances(example_graph, 'X'))
```

```
{'U': 1, 'V': 2, 'W': 2, 'X': 0, 'Y': 1, 'Z': 2}
```

18.deapth search

```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

```
Following is the Depth-First Search
5
3
2
4
8
7
```

18.breath search

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

20.heap sort

```
# heapify
def heapify(arr, n, i):
   largest = i # largest value
   l = 2 * i + 1 # left
   r = 2 * i + 2 # right
   # if left child exists
   if l < n and arr[i] < arr[l]:
      largest = l
   # if right child exits
   if r < n and arr[largest] < arr[r]:
```

```
            largest = r
        # root
        if largest != i:
            arr[i],arr[largest] = arr[largest],arr[i] # swap
            # root.
            heapify(arr, n, largest)
    # sort
    def heapSort(arr):
        n = len(arr)
        # maxheap
        for i in range(n, -1, -1):
            heapify(arr, n, i)
        # element extraction
        for i in range(n-1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i] # swap
            heapify(arr, i, 0)
    # main
    arr = [2,5,3,8,6,5,4,7]
    heapSort(arr)
    n = len(arr)
    print ("Sorted array is")
    for i in range(n):
        print (arr[i],end=" ")
```

```
Sorted array is
2 3 4 5 5 6 7 8
```