Data Structures and Algorithm using Python

Lab Programs:

Program1

Recursion

Recursion is the process of defining something in terms of itself.

Recursion Function:

Program 1:

# Write Python Program to Perform Recursion operation to find factorial for given integers.

```
def factorial(x):
   "This is a recursive function to find the factorial of an integer"
   if x == 1:
      return 1
   else:
      return (x * factorial(x-1))
     num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

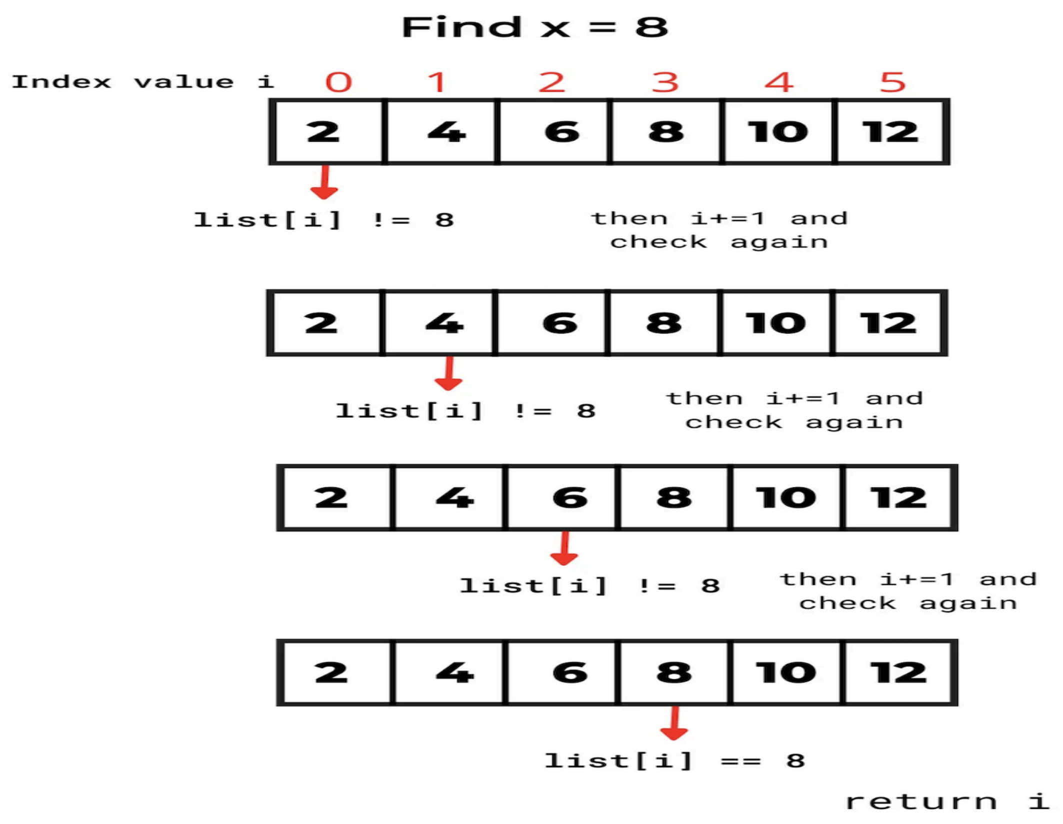
```
The factorial of 3 is 6
```

Reference:

```
factorial(3)  #1st call with 3
3 * factorial(2)  #2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 *1 # return from 3rd call as number=1
3 * 2 #return from 2nd call
6 #return from 1st call
```

Linear recursion:

Linear Search is a searching algorithm in which we sequentially search for a presence of a particular element inside a list or array.



Program 2:

# Write Python Program to Perform Linear Search using Recursion to find the element is found or not and its position.

```
L1= [2,4,6,8,10,12]
x=8 # element that we want to be found
i=0 # pointer
while i<len(L1):
    if L1[i]==x:
        print(f'element {x} present at {i}th position')
        break
    i+=1
if i>=len(L1):
    print('Element is not present')
```

Output:

```
Element 8 present at 3th position
```

Program 3:

# Write Python Program to Perform Linear Search using Recursion to display the element in the array and find the element using its index.

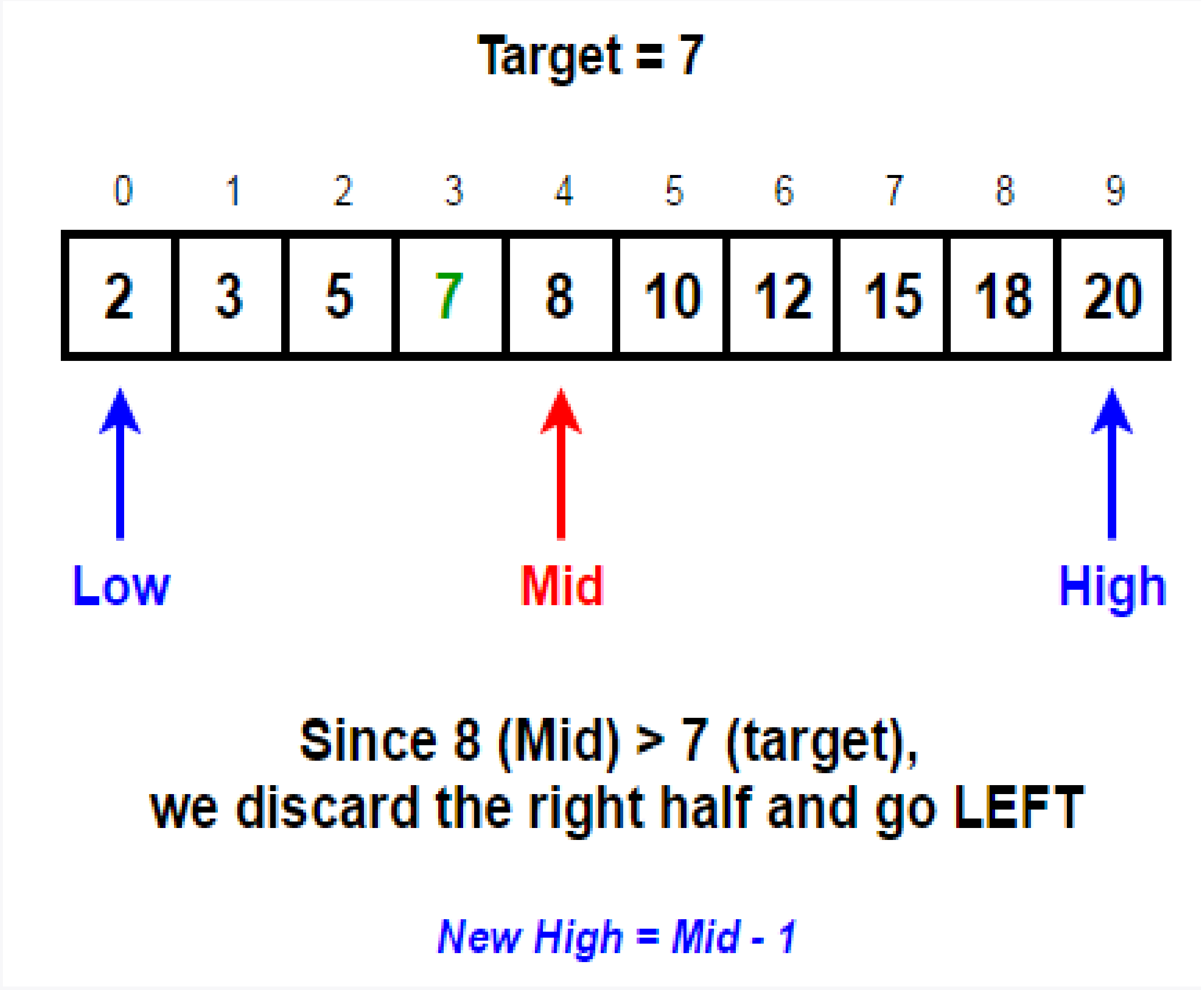
```
def linear_search(L, key, i):
    if i >= len(L):
        return -1
    if L[i] == key:
        return i
    return linear_search(L, key, i + 1)
L = [1, 2, 3, 4, 5, 6, 7]
key = 5
x = linear_search(L, key, 0)
if x != -1:
    print('List : ', L)
    print(f'Element {key} is available on index : {x}')
else:
    print(‘The element is not present’)
```

Output:

```
List: [1,2,3,4,5,6,7]

Element 5 is available on index : 4
```

Binary Recursion:

Given a sorted array of n integers and a target value, determine if the target exists in the array in logarithmic time using the binary search algorithm. If target exists in the array, print the index of it.



Samples:

Input: nums[] = [2, 3, 5, 7, 9]
target = 7
Output: Element found at index 3
Input: nums[] = [1, 4, 5, 8, 9]target = 2
Output: Element not found

Program 4:

```
        # Write Python Program to Perform Binary Search using Recursion using
a array to find the position of an element in an index.

# Returns index of x in arr if present, else -1
def binary_search(arr, low, high, x):

        # Check base case
        if high >= low:

                mid = (high + low) // 2

                # If element is present at the middle itself
                if arr[mid] == x:
                        return mid

                # If element is smaller than mid, then it can only
                # be present in left subarray
                elif arr[mid] > x:
                        return binary_search(arr, low, mid - 1, x)

                # Else the element can only be present in right subarray
                else:
                        return binary_search(arr, mid + 1, high, x)
        else:
                # Element is not present in the array
                return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binary_search(arr, 0, len(arr)-1, x)
if result != -1:
        print("Element is present at index", str(result))
else:
        print("Element is not present in array")
```

Output:
Element is present at index 3

Program 5:

# Write Python Program to Perform Binary Search using Recursion to display the element in the array and find the element using its index.

```
# Write Python Program to Perform Binary Search using Recursion
def binary_search(L, key, low, high):
        if high >= low:
                mid = (low + high) // 2
                if key == L[mid]:
                        return mid
                elif key > L[mid]:
                        return binary_search(L, key, mid + 1, high)
                else:
                        return binary_search(L, key, low, mid - 1)
        else:
                return -1
L = [11, 22, 33, 44, 55, 66, 77]
key = 44
idx = binary_search(L, key, 0, len(L) - 1)
if idx!=-1:
        print(f'In list {L} the key {key} lies on index : {idx} ')
else:
        print('The element is not present in array')
```

Output:

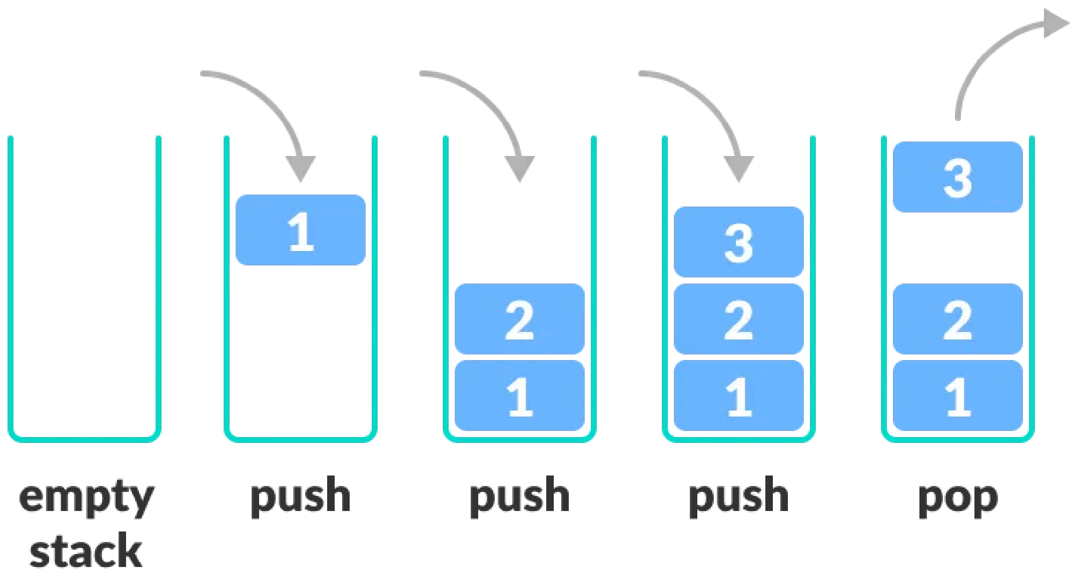
```
In list [11,22,33,44,55,66,77] the key 44 lies on index : 3
```

Stack Data Structure

A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means the last element inserted inside the stack is removed first. You can think of the stack data structure as the pile of plates on top of another.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.

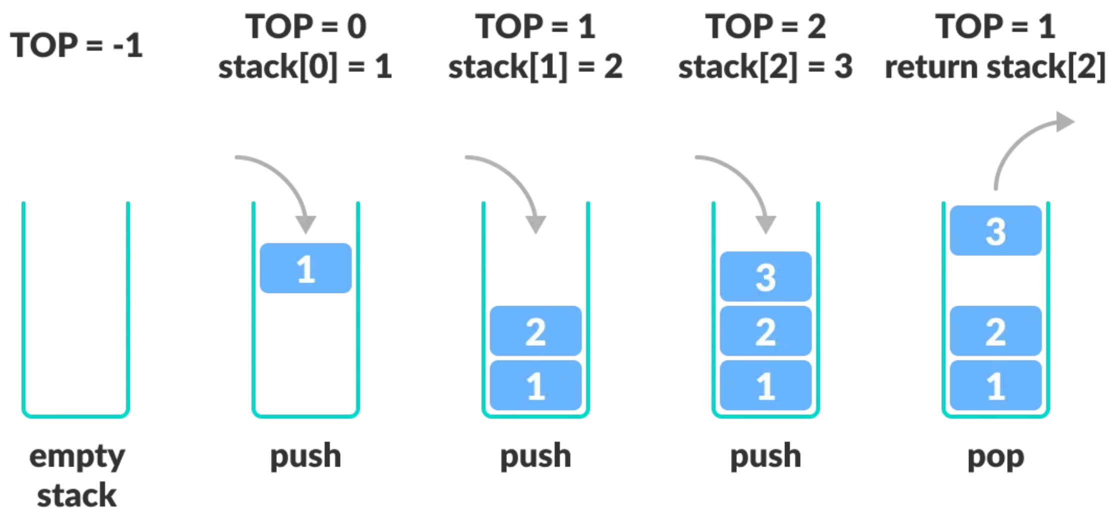


Basic Operations of Stack

There are some basic operations that allow us to perform different actions on a stack.

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)

- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element ‘a’ at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)



Program 6:

# Write Python Program to Perform Stack operation to print in reverse sort.

```
# Python program to
# demonstrate stack implementation
# using list

stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Output

```
Initial stack
['a', 'b', 'c']


Elements popped from stack:
c
b
a


Stack after elements are popped:
[]
```

Program 7:

# Write Python Program to Perform Stack pop operation to print the values in reverse order.

```
# Python program to
# demonstrate stack implementation
# using collections.deque
```

```
from collections import deque

stack = deque()

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack:')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

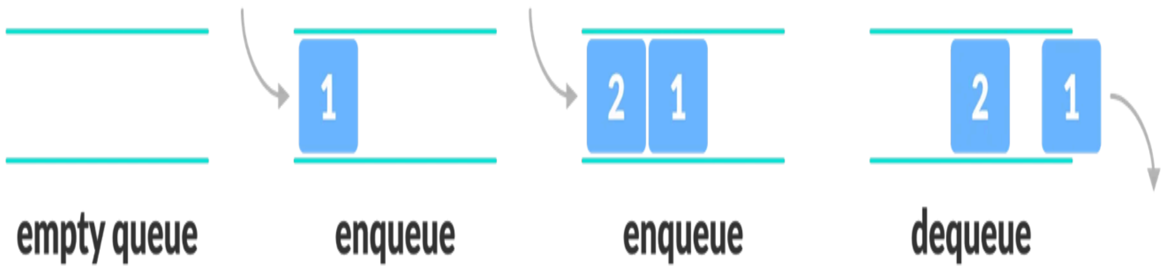
Output

```
Initial stack:
deque(['a', 'b', 'c'])

Elements popped from stack:
c
b
a

Stack after elements are popped:
deque([])
```

Queue Data Structure

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.



Basic Operations of Queue

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full

- **Peek**: Get the value of the front of the queue without removing it

Working of Queue

Queue operations work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Enqueue Operation

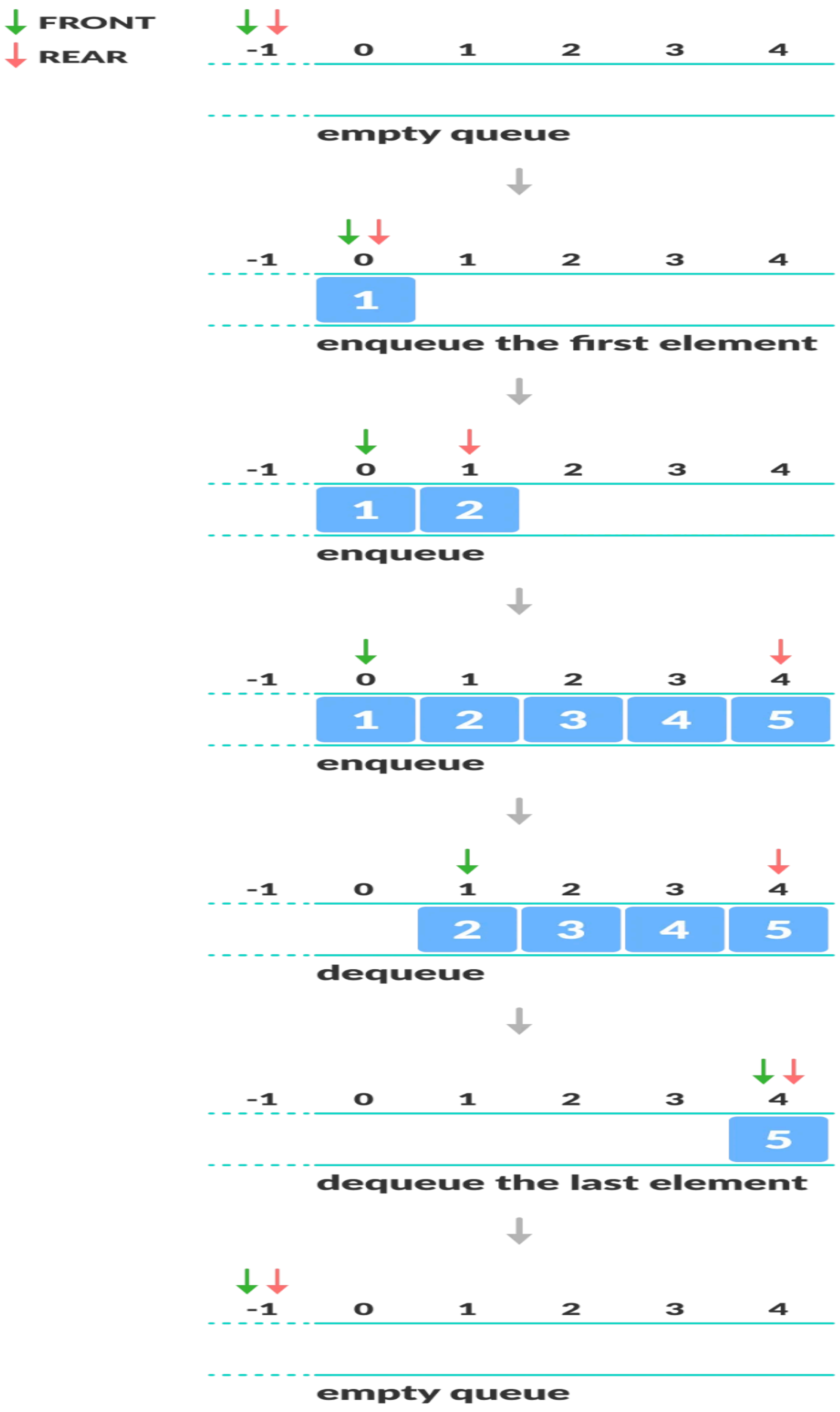
- check if the queue is full

- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Dequeue Operation

- check if the queue is empty

- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue
-1 0 1 2 3 4
2 3 4 5
dequeue
-1 0 1 2 3 4
5
dequeue the last element
-1 0 1 2 3 4
empty queue

Program 8:

# Write Python Program to Perform Queue operation to do insert and remove method.

```
queue = []
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
```

Output:

```
Initial queue
['a', 'b', 'c']
Elements dequeued from queue
a
b
c
Queue after removing elements
[]
```

Program 9

```
from collections import deque
q = deque()
q.append('a')
q.append('b')
q.append('c')
print("Initial queue")
print(q)
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())
print("\nQueue after removing elements")
print(q)
```

Output:

```
Initial queue
deque(['a', 'b', 'c'])
Elements dequeued from the queue
a
b
c
Queue after removing elements
deque([])
```

Program 10:

# Write Python Program to Perform Queue operation to check the queue is empty and print the elements.

```
from queue import Queue
q = Queue(maxsize = 3)
print(q.qsize())
q.put('a')
q.put('b')
q.put('c')
print("\nFull: ", q.full())
print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())
print("\nEmpty: ", q.empty())
q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())
```

Output:

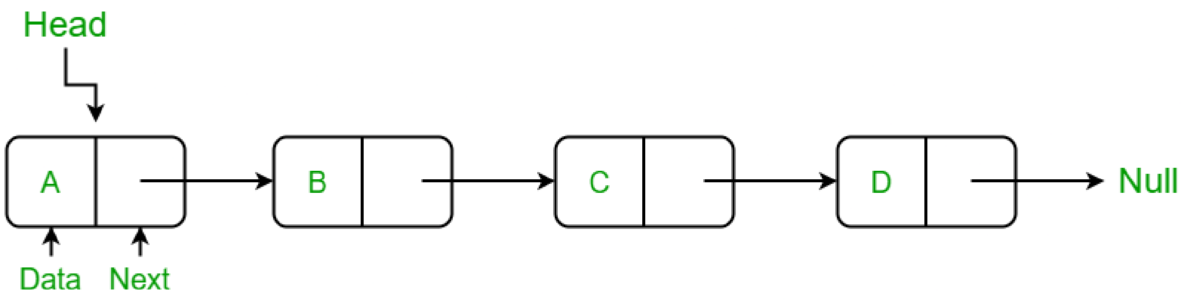
```
0
Full:  True
```

```
Elements dequeued from the queue
a
b
c
Empty:  True
Empty:  False
Full:  False
```

Linked List in Python

What is Linked List in Python

A linked list is a type of linear data structure similar to arrays. It is a collection of nodes that are linked with each other. A node contains two things first is data and second is a link that connects it with another node. Below is an example of a linked list with four nodes and each node contains character data and a link to another node. Our first node is where **head** points and we can access all the elements of the linked list using the **head.**



Creating a linked list in Python

In this LinkedList class, we will use the Node class to create a linked list. In this class, we have an

__init__ method that initializes the linked list with an empty head.
insertAtBegin() method to insert a node at the beginning of the linked list.
insertAtIndex() method to insert a node at the given index of the linked list.
insertAtEnd() method inserts a node at the end of the linked list.
remove_node() method takes the data as an argument to delete that node.
sizeOfLL() method to get the current size of the linked list.
printLL() which traverses the linked list and prints the data of each node.

Insertion in Linked List

1. Insertion at Beginning in Linked List
2. Insert a Node at a Specific Position in a Linked List
3. Insertion in Linked List at End
4. Update the Node of a Linked List

Delete Node in a Linked List

1. Remove First Node from Linked List
2. Remove Last Node from Linked List
3. Delete a Linked List Node at a given Position
4. Delete a Linked List Node of a given Data

Linked List Traversal
Length of a Linked List

Program 11:
# Using Python write a Linked List Program to create a node class to create a node using different operations and print the elements.

```
    # Create a Node class to create a node
class Node:
        def __init__(self, data):
                self.data = data
                self.next = None

# Create a LinkedList class


class LinkedList:
        def __init__(self):
                self.head = None

        # Method to add a node at begin of LL
        def insertAtBegin(self, data):
                new_node = Node(data)
                if self.head is None:
                        self.head = new_node
                        return
                else:
                        new_node.next = self.head
                        self.head = new_node

        # Method to add a node at any index
        # Indexing starts from 0.
        def insertAtIndex(self, data, index):
                new_node = Node(data)
                current_node = self.head
                position = 0
                if position == index:
                        self.insertAtBegin(data)
                else:
                        while(current_node != None and position+1 != index):
                                position = position+1
                                current_node = current_node.next

                        if current_node != None:
                                new_node.next = current_node.next
                                current_node.next = new_node
                        else:
                                print("Index not present")

        # Method to add a node at the end of LL

        def insertAtEnd(self, data):
                new_node = Node(data)
                if self.head is None:
                        self.head = new_node
                        return

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

                current_node.next = new_node

        # Update node of a linked list
                # at given position
        def updateNode(self, val, index):
                current_node = self.head
                position = 0
                if position == index:
                        current_node.data = val
                else:
                        while(current_node != None and position != index):
                                position = position+1
                                current_node = current_node.next

                        if current_node != None:
                                current_node.data = val
                        else:
                                print("Index not present")

        # Method to remove first node of linked list

        def remove_first_node(self):
                if(self.head == None):
```

```
        return

    self.head = self.head.next

# Method to remove last node of linked list
def remove_last_node(self):

    if self.head is None:
        return

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

    current_node.next = None

# Method to remove at given index
def remove_at_index(self, index):
    if self.head == None:
        return

    current_node = self.head
    position = 0
    if position == index:
        self.remove_first_node()
    else:
        while(current_node != None and position+1 != index):
            position = position+1
            current_node = current_node.next

        if current_node != None:
            current_node.next = current_node.next.next
        else:
            print("Index not present")

# Method to remove a node from linked list
def remove_node(self, data):
    current_node = self.head

    if current_node.data == data:
        self.remove_first_node()
        return

    while(current_node != None and current_node.next.data != data):
        current_node = current_node.next

    if current_node == None:
        return
    else:
        current_node.next = current_node.next.next
        # Print the size of linked list
        def sizeOfLL(self):
                size = 0
                if(self.head):
                        current_node = self.head
                        while(current_node):
                                size = size+1
                                current_node = current_node.next
                        return size
                else:
                        return 0

        # print method for the linked list
        def printLL(self):
                current_node = self.head
                while(current_node):
                        print(current_node.data)
                        current_node = current_node.next


# create a new linked list
llist = LinkedList()

# add nodes to the linked list
llist.insertAtEnd('a')
llist.insertAtEnd('b')
llist.insertAtBegin('c')
llist.insertAtEnd('d')
llist.insertAtIndex('g', 2)

# print the linked list
print("Node Data")
llist.printLL()

# remove a nodes from the linked list
print("\nRemove First Node")
llist.remove_first_node()
print("Remove Last Node")
llist.remove_last_node()
print("Remove Node at Index 1")
llist.remove_at_index(1)


# print the linked list again
print("\nLinked list after removing a node:")
llist.printLL()

print("\nUpdate node Value")
llist.updateNode('z', 0)
llist.printLL()

print("\nSize of linked list :", end=" ")
print(llist.sizeOfLL())
```

OUTPUT:

```
Node Data
c
a
g
b
d

Remove First Node
Remove Last Node
Remove Node at Index 1

Linked list after removing a node:
a
b

Update node Value
z
b

Size of linked list : 2
```

Program 12:

# Using Python write a Double Linked List Program to perform Insert, Delete operation and print the elements.

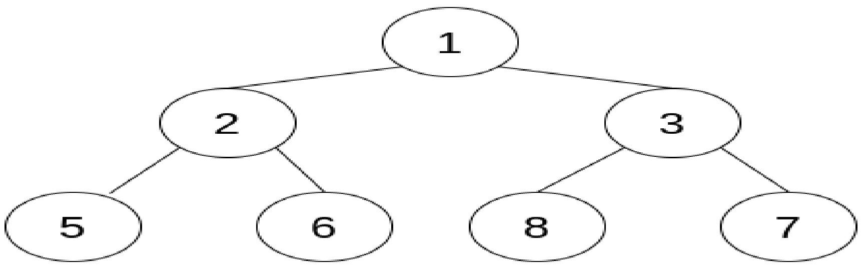
```
# Initialise the Node
class Node:
  def __init__(self, data):
      self.item = data
      self.next = None
      self.prev = None
# Class for doubly Linked List
class doublyLinkedList:
  def __init__(self):
      self.start_node = None
  # Insert Element to Empty list
  def InsertToEmptyList(self, data):
      if self.start_node is None:
          new_node = Node(data)
          self.start_node = new_node
      else:
          print("The list is empty")
  # Insert element at the end
  def InsertToEnd(self, data):
      # Check if the list is empty
      if self.start_node is None:
          new_node = Node(data)
          self.start_node = new_node
          return
      n = self.start_node
      # Iterate till the next reaches NULL
      while n.next is not None:
          n = n.next
      new_node = Node(data)
      n.next = new_node
      new_node.prev = n
  # Delete the elements from the start
  def DeleteAtStart(self):
      if self.start_node is None:
          print("The Linked list is empty, no element to delete")
          return
      if self.start_node.next is None:
          self.start_node = None
          return
```

```
        self.start_node = self.start_node.next
        self.start_prev = None;
    # Delete the elements from the end
    def delete_at_end(self):
        # Check if the List is empty
        if self.start_node is None:
            print("The Linked list is empty, no element to delete")
            return
        if self.start_node.next is None:
            self.start_node = None
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        n.prev.next = None
    # Traversing and Displaying each element of the list
    def Display(self):
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                print("Element is: ", n.item)
                n = n.next
        print("\n")
# Create a new Doubly Linked List
NewDoublyLinkedList = doublyLinkedList()
# Insert the element to empty list
NewDoublyLinkedList.InsertToEmptyList(10)
# Insert the element at the end
NewDoublyLinkedList.InsertToEnd(20)
NewDoublyLinkedList.InsertToEnd(30)
NewDoublyLinkedList.InsertToEnd(40)
NewDoublyLinkedList.InsertToEnd(50)
NewDoublyLinkedList.InsertToEnd(60)
# Display Data
NewDoublyLinkedList.Display()
# Delete elements from start
NewDoublyLinkedList.DeleteAtStart()
# Delete elements from end
NewDoublyLinkedList.DeleteAtStart()
# Display Data
NewDoublyLinkedList.Display()
```

HEAPS USING PRIORITY QUEUE

Abstract data structures specify operations and the relationships between them. The priority queue abstract data structure, for example, supports three operations:

1. **is_empty** checks whether the queue is empty.
2. **add_element** adds an element to the queue.
3. **pop_element** pops the element with the highest priority.



There are three rules that determine the relationship between the element at the index kand its surrounding elements:

1. Its first child is at 2*k + 1.
2. Its second child is at 2*k + 2.
3. Its parent is at (k - 1) // 2.

Program 13:

# Using Python write a Heaps Program to insert the element in a node, Sort, and print the elements.

```
# import modules
import heapq as hq
# list of students
list_stu = [(5,'Rina'),(1,'Anish'),(3,'Moana'),(2,'cathy'),(4,'Lucy')]
# Arrange based on the roll number
hq.heapify(list_stu)
print("The order of presentation is :")
for i in list_stu:
print(i[0],':',i[1])
```

Output:

```
The order of presentation is :
1 : Anish
2 : cathy
```

```
3 : Moana
5 : Rina
4 : Lucy
```

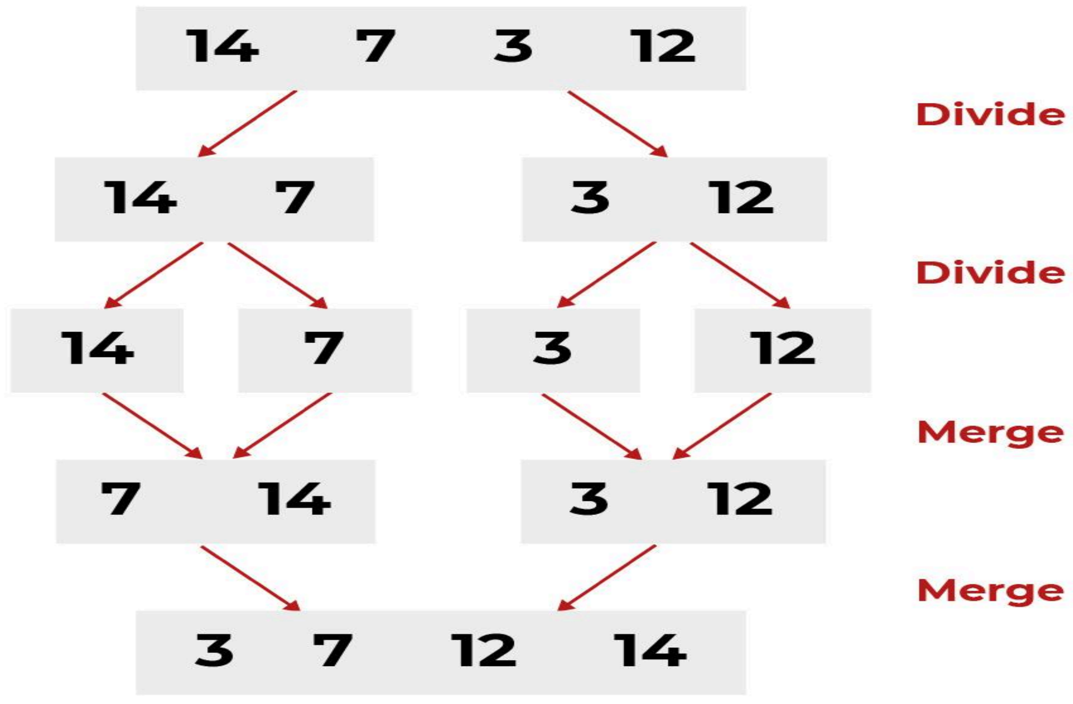
Program 14:

# Using Python write a Heaps Program to perform Insert, Swap,Delete, and print the elements.

```
# Priority Queue implementation in Python
# Function to heapify the tree
def heapify(arr, n, i):
# Find the largest among root, left child and right child
largest = i
l = 2 * i + 1
r = 2 * i + 2
if l < n and arr[i] < arr[l]:
largest = l
if r < n and arr[largest] < arr[r]:
largest = r

# Swap and continue heapifying if root is not largest
if largest != i:
arr[i], arr[largest] = arr[largest], arr[i]
heapify(arr, n, largest)

# Function to insert an element into the tree
def insert(array, newNum):
size = len(array)
if size == 0:
array.append(newNum)
else:
array.append(newNum)
for i in range((size // 2) - 1, -1, -1):
heapify(array, size, i)

# Function to delete an element from the tree
def deleteNode(array, num):
    size = len(array)
    if size == 0:
        print("Heap is empty")
        return

    i = 0
    for i in range(size):
        if num == array[i]:
            break
    else:
        print("Element not found")
        return  # Exit if the element is not found
    array[i], array[size - 1] = array[size - 1], array[i]
    array.pop()

    for i in range((len(array) // 2) - 1, -1, -1):
        heapify(array, len(array), i)


arr = []
insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)
print ("Max-Heap array: " + str(arr))
deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

Output:

Merge Sort



Program 15:

To write a python program to implement the Merge Sort algorithm to sort a list of numbers, displaying the list before and after sorting.

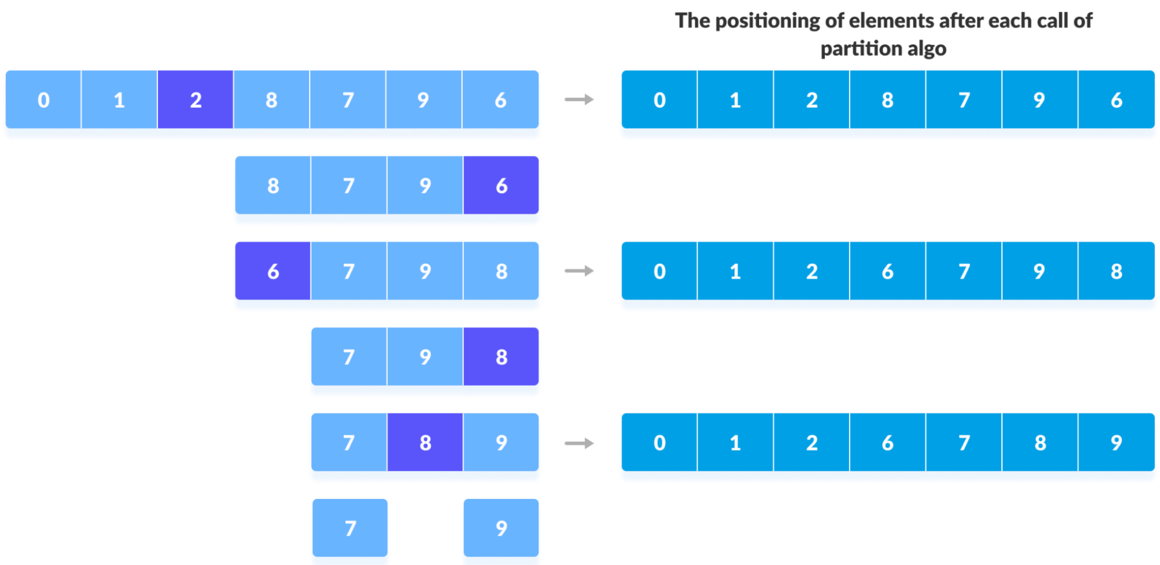
Source Code:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # Add remaining elements
    merged += left[i:]
    merged += right[j:]
    return merged

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
print("Before sorting:", arr)
sorted_arr = merge_sort(arr)
print("After sorting:", sorted_arr)
```

Output:

```
Before sorting: [38, 27, 43, 3, 9, 82, 10]
After sorting: [3, 9, 10, 27, 38, 43, 82]
```

Quick Sort



Program 16:

To write a python program to implement the Quick Sort algorithm to sort a list of numbers, displaying the list before and after sorting.

Source Code:

```
# function to find the partition position
def partition(array, low, high):

  # choose the rightmost element as pivot
  pivot = array[high]

  # pointer for greater element
  i = low - 1
  # traverse through all elements
  # compare each element with pivot
  for j in range(low, high):
    if array[j] <= pivot:
      # if element smaller than pivot is found
      # swap it with the greater element pointed by i
      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:

    # find pivot element such that
    # element smaller than pivot are on the left
    # element greater than pivot are on the right
    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)
```

Output:

—--

Binary Search Tree Traversal

Program 17:

To write a python program to implement the Binary Search Tree Traversal algorithm to perform Insert, Search,and Delete operations.

```
# Create a node
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Inorder traversal
def inorder(root):
    if root is not None:
        # Traverse left
        inorder(root.left)
        # Traverse root
        print(str(root.key) + "->", end=' ')

        # Traverse right
        inorder(root.right)


# Insert a node
def insert(node, key):

    # Return a new node if the tree is empty
    if node is None:
        return Node(key)

    # Traverse to the right place and insert the node
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    return node

# Find the inorder successor
def minValueNode(node):
    current = node

    # Find the leftmost leaf
    while(current.left is not None):
        current = current.left
    return current

# Deleting a node
def deleteNode(root, key):
```

```
# Return if the tree is empty
if root is None:
    return root

# Find the node to be deleted
if key < root.key:
    root.left = deleteNode(root.left, key)
elif(key > root.key):
    root.right = deleteNode(root.right, key)
else:
    # If the node is with only one child or no child
    if root.left is None:
        temp = root.right
        root = None
        return temp

    elif root.right is None:
        temp = root.left
        root = None
        return temp

    # If the node has two children,
    # place the inorder successor in position of the node to be deleted
    temp = minValueNode(root.right)

    root.key = temp.key

    # Delete the inorder successor
    root.right = deleteNode(root.right, temp.key)
    return root
root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)

print("Inorder traversal: ", end=' ')
inorder(root)

print("\nDelete 10")
root = deleteNode(root, 10)
print("Inorder traversal: ", end=' ')
inorder(root)
```

Binary Search Tree

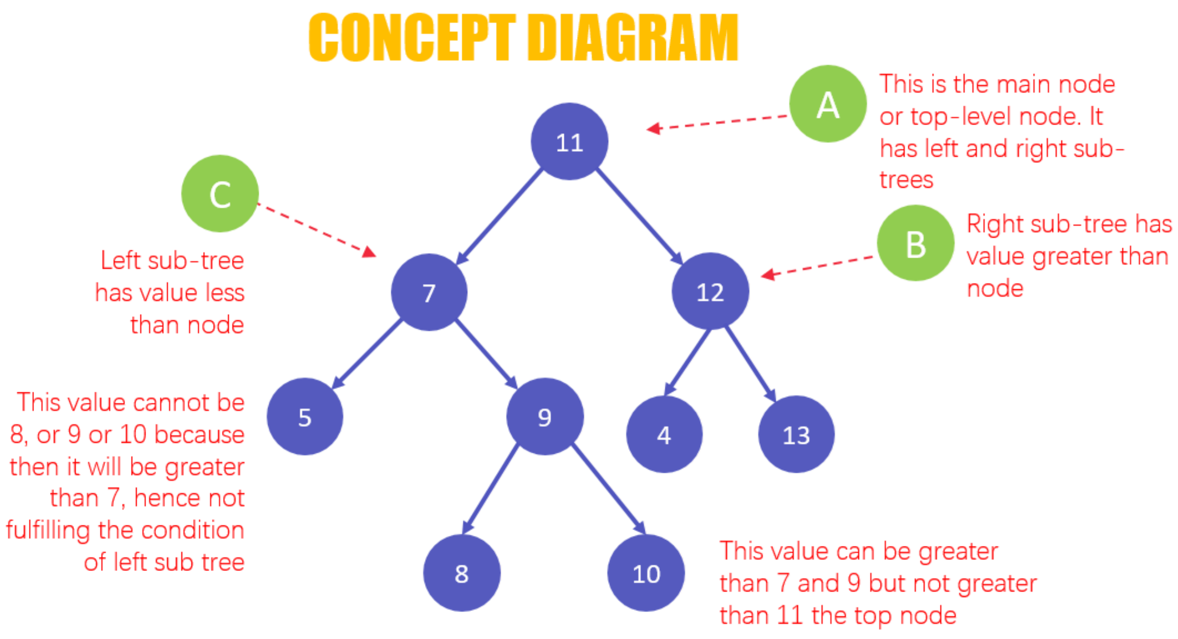
A **Binary Search Tree (BST)** is a tree data structure in which each node has at most two children, referred to as the left and right child.

The key feature of a BST is that for each node:

- The left subtree contains only nodes with values less than the node's value.
- The right subtree contains only nodes with values greater than the node's value.
- This property makes the Binary Search Tree useful for efficient searching, insertion, and deletion operations.

Operations in BST:

- **Search:** Find a value in the tree.
- **Insertion:** Insert a value into the tree while maintaining the BST property.
- **Deletion:** Remove a node while maintaining the BST property.
- **Traversal:** Visit all nodes in a specific order (in-order, pre-order, post-order)


CONCEPT DIAGRAM
A
This is the main node or top-level node. It has left and right sub-trees
B
Right sub-tree has value greater than node
C
Left sub-tree has value less than node
This value cannot be 8, or 9 or 10 because then it will be greater than 7, hence not fulfilling the condition of left sub tree
This value can be greater than 7 and 9 but not greater than 11 the top node
11
7
12
5
9
4
13
8
10

Program 18:

To write a python program to implement the Binary Search Tree algorithm to perform Insert, Search,and Delete operation

Source Code:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self._insert(value, self.root)

    def _insert(self, value, node):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert(value, node.left)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert(value, node.right)

    def search(self, value):
        return self._search(value, self.root)

    def _search(self, value, node):
        if not node or node.value == value:
            return node
        if value < node.value:
            return self._search(value, node.left)
        return self._search(value, node.right)

    def delete(self, value):
        self.root = self._delete(value, self.root)

    def _delete(self, value, node):
        if not node:
            return None
        if value < node.value:
            node.left = self._delete(value, node.left)
        elif value > node.value:
            node.right = self._delete(value, node.right)
        else:
            if not node.left:
                return node.right
            if not node.right:
                return node.left
            temp = node.right
            while temp.left:
                temp = temp.left
            node.value = temp.value
            node.right = self._delete(temp.value, node.right)
        return node

# Example usage
if __name__ == "__main__":
    bst = BST()
    for i in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(i)
    print("Search 40:", "Found" if bst.search(40) else "Not Found")
    print("Search 90:", "Found" if bst.search(90) else "Not Found")
    bst.delete(20)
    print("After deleting 20, search 20:", "Found" if bst.search(20) else "Not
Found")
    bst.delete(30)
    print("After deleting 30, search 30:", "Found" if bst.search(30) else "Not
Found")
    bst.delete(50)
    print("After deleting 50, search 50:", "Found" if bst.search(50) else "Not
Found")
```

Depth First search tree traversal

Depth-First Search (DFS)

Depth-First Search (DFS) is an algorithm used to traverse or search through graphs and trees. It explores as far as possible along each branch before backtracking. DFS is commonly implemented using recursion or an explicit stack (when using an iterative approach).

Types of DFS Orders in a Binary Tree

DFS can be performed in multiple orders when traversing a binary tree:

Preorder (Root → Left → Right)

Visit the current node first.

Recursively traverse the left subtree.

Recursively traverse the right subtree.

Example Output: [1, 2, 4, 5, 3, 6, 7]

Inorder (Left → Root → Right)

Recursively traverse the left subtree.

Visit the current node.

Recursively traverse the right subtree.

Example Output: [4, 2, 5, 1, 6, 3, 7]

Used in BSTs to retrieve elements in sorted order

Postorder (Left → Right → Root)

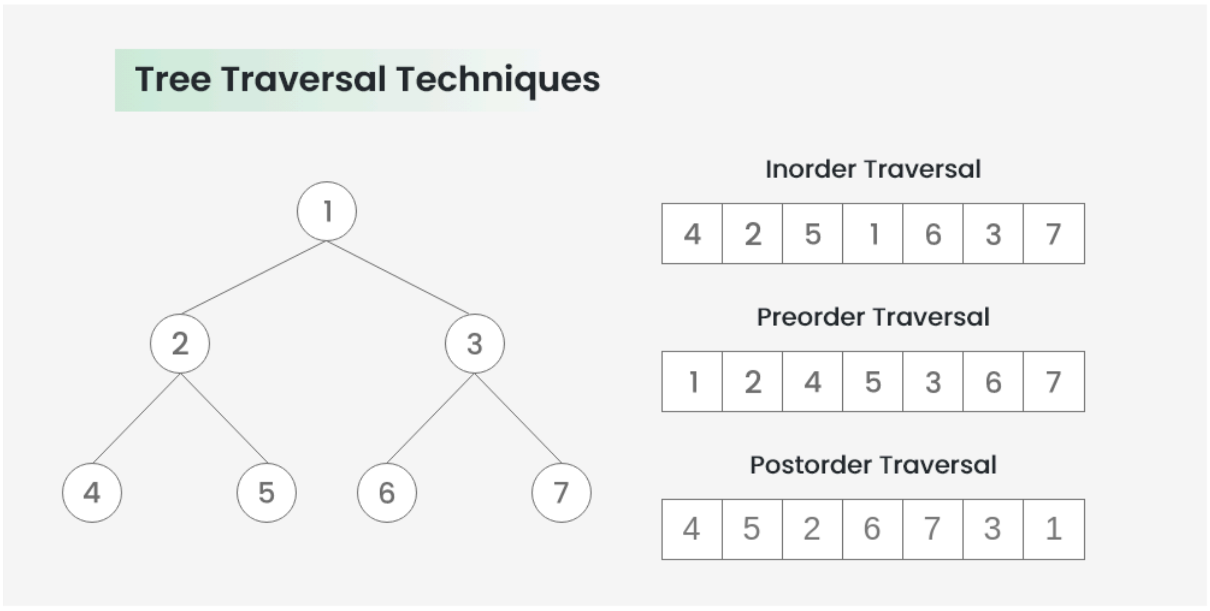
Recursively traverse the left subtree.

Recursively traverse the right subtree.

Visit the current node.

Example Output: [4, 5, 2, 6, 7, 3, 1]

Used in deleting trees (because children are processed before parents).



Program 18:

Source code:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def preorder_dfs(root):
    if root is None:
        return []
    return [root.val] + preorder_dfs(root.left) + preorder_dfs(root.right)
```

```
def inorder_dfs(root):
    if root is None:
        return []
    return inorder_dfs(root.left) + [root.val] + inorder_dfs(root.right)

def postorder_dfs(root):
    if root is None:
        return []
    return postorder_dfs(root.left) + postorder_dfs(root.right) + [root.val]



# Example usage:
if __name__ == "__main__":
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)
    root.right.left = TreeNode(6)
    root.right.right = TreeNode(7)

    print("Preorder DFS:", preorder_dfs(root))
    print("Inorder DFS:", inorder_dfs(root))
    print("Postorder DFS:", postorder_dfs(root))
```