1. Singly linked list

```
# 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())
```

```
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
```

2. Doubly Linked List :

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


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

    def isEmpty(self):
        if self.head is None:
            return True
        return False

    def length(self):
        temp = self.head
        count = 0
        while temp is not None:
            temp = temp.next
            count += 1
        return count

    def search(self, value):
        temp = self.head
        isFound = False
        while temp is not None:
            if temp.data == value:
                isFound = True
                break
            temp = temp.next
        return isFound

    def insertAtBeginning(self, value):
        new_node = Node(value)
        if self.isEmpty():
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.previous = new_node
            self.head = new_node

    def insertAtEnd(self, value):
        new_node = Node(value)
        if self.isEmpty():
            self.insertAtBeginning(value)
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = new_node
            new_node.previous = temp

    def insertAfterElement(self, value, element):
        temp = self.head
        while temp is not None:
            if temp.data == element:
                break
            temp = temp.next
        if temp is None:
            print("{} is not present in the linked list. {} cannot be inserted into the list.".format(element, value))
        else:
            new_node = Node(value)
            new_node.next = temp.next
            new_node.previous = temp
            temp.next.previous = new_node
            temp.next = new_node

    def insertAtPosition(self, value, position):
        temp = self.head
        count = 0
        while temp is not None:
            if count == position - 1:
                break
            count += 1
            temp = temp.next
        if position == 1:
            self.insertAtBeginning(value)
        elif temp is None:
            print("There are less than {}-1 elements in the linked list. Cannot insert at {} position.".format(position,
                                                                                                               position))
        elif temp.next is None:
            self.insertAtEnd(value)
        else:
            new_node = Node(value)
            new_node.next = temp.next
            new_node.previous = temp
            temp.next.previous = new_node
            temp.next = new_node

    def printLinkedList(self):
        temp = self.head
        while temp is not None:
            print(temp.data, sep=",")
            temp = temp.next

    def updateElement(self, old_value, new_value):
        temp = self.head
        isUpdated = False
        while temp is not None:
            if temp.data == old_value:
                temp.data = new_value
                isUpdated = True
            temp = temp.next
        if isUpdated:
            print("Value Updated in the linked list")
        else:
            print("Value not Updated in the linked list")

    def updateAtPosition(self, value, position):
        temp = self.head
        count = 0
        while temp is not None:
            if count == position:
                break
            count += 1
            temp = temp.next
        if temp is None:
            print("Less than {} elements in the linked list. Cannot update.".format(position))
        else:
            temp.data = value
            temp.data = value
            print("Value updated at position {}".format(position))

    def deleteFromBeginning(self):
        if self.isEmpty():
            print("Linked List is empty. Cannot delete elements.")
        elif self.head.next is None:
            self.head = None
        else:
            self.head = self.head.next
            self.head.previous = None

    def deleteFromLast(self):
        if self.isEmpty():
            print("Linked List is empty. Cannot delete elements.")
        elif self.head.next is None:
            self.head = None
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.previous.next = None
            temp.previous = None

    def delete(self, value):
        if self.isEmpty():
            print("Linked List is empty. Cannot delete elements.")
        elif self.head.next is None:
            if self.head.data == value:
                self.head = None
        else:
            temp = self.head
            while temp is not None:
                if temp.data == value:
                    break
                temp = temp.next
            if temp is None:
                print("Element not present in linked list. Cannot delete element.")
            elif temp.next is None:
                self.deleteFromLast()
            else:
                temp.next = temp.previous.next
                temp.next.previous = temp.previous
                temp.next = None
                temp.previous = None

    def deleteFromPosition(self, position):
        if self.isEmpty():
            print("Linked List is empty. Cannot delete elements.")
        elif position == 1:
            self.deleteFromBeginning()
        else:
            temp = self.head
            count = 1
            while temp is not None:
                if count == position:
                    break
                temp = temp.next
            if temp is None:
                print("There are less than {} elements in linked list. Cannot delete element.".format(position))
            elif temp.next is None:
                self.deleteFromLast()
                temp.previous.next = temp.next
                temp.next.previous = temp.previous
                temp.next = None
                temp.previous = None


x = DoublyLinkedList()
print(x.isEmpty())
x.insertAtBeginning(5)
x.printLinkedList()
x.insertAtEnd(10)
x.printLinkedList()
x.deleteFromLast()
x.printLinkedList()
x.insertAtEnd(25)
x.printLinkedList()
x.deleteFromLast()
x.deleteFromBeginning()
x.insertAtEnd(100)
x.printLinkedList()
```

```
True
5
5
10
5
5
25
100
```

3. Circular Linked List :

```
# Python program to delete a given key from linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Function to insert a node at the
# beginning of a Circular linked list


def push(head, data):
    # Create a new node and make head as next of it.
    newP = Node(data)
    newP.next = head

    # If linked list is not NULL then
    # set the next of last node
    if head != None:
        # Find the node before head and
        # update next of it.
        temp = head
        while (temp.next != head):
            temp = temp.next
        temp.next = newP
    else:
        newP.next = newP
    head = newP
    return head

# Function to print nodes in a given circular linked list


def printList(head):
    if head == None:
        print("List is Empty")
        return
    temp = head.next
    print(head.data, end=' ')
    if (head != None):
        while (temp != head):
            print(temp.data, end=" ")
            temp = temp.next
    print()

# Function to delete a given node
# from the list


def deleteNode(head, key):
    # If linked list is empty
    if (head == None):
        return

    # If the list contains only a
    # single node
    if (head.data == key and head.next == head):
        head = None
        return

    last = head

    # If head is to be deleted
    if (head.data == key):
        # Find the last node of the list
        while (last.next != head):
            last = last.next

        # Point last node to the next of
        # head i.e. the second node
        # of the list
        last.next = head.next
        head = last.next
        return

    # Either the node to be deleted is
    # not found or the end of list
    # is not reached
    while (last.next != head and last.next.data != key):
        last = last.next

    # If node to be deleted was found
    if (last.next.data == key):
        d = last.next
        last.next = d.next
        d = None
    else:
        print("Given node is not found in the list!!!")


# Initialize lists as empty
head = None

# Created linked list will be
# 2->5->7->8->10
head = push(head, 2)
head = push(head, 5)
head = push(head, 7)
head = push(head, 8)
head = push(head, 10)

print("List Before Deletion: ")
printList(head)

deleteNode(head, 7)
print("List After Deletion: ")
printList(head)
```

```
List Before Deletion:
10 8 7 5 2
List After Deletion:
10 8 5 2
```

4. Stack Implementation using Linked List :

```
class Node:

    # Class to create nodes of linked list
    # constructor initializes node automatically
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack:

    # head is default NULL
    def __init__(self):
        self.head = None

    # Checks if stack is empty
    def isempty(self):
        if self.head == None:
            return True
        else:
            return False

    # Method to add data to the stack
    # adds to the start of the stack
    def push(self, data):

        if self.head == None:
            self.head = Node(data)

        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode

    # Remove element that is the current head (start of the stack)
    def pop(self):

        if self.isempty():
            return None

        else:
            # Removes the head node and makes
            # the preceding one the new head
            poppednode = self.head
            self.head = self.head.next
            poppednode.next = None
            return poppednode.data

    # Returns the head node data
    def peek(self):

        if self.isempty():
            return None

        else:
            return self.head.data
```

5. Conversion of infix to postfix expression, Evaluation of postfix expression

```
    def display(self):
```

```
def is_operator(char):
  return char in {'+','-','*','/'}

def precedence(operator):
  if operator == '+' or operator == '-':
    return 1
  elif operator == '*' or operator == '/':
    return 2
  else:
    return 0

def infix_to_postfix(infix_expresion):
  stack=[]
  postfix_expression=[]

  for char in infix_expression:
    if char.isalnum():
```