DATA STRUCTURE AND ALGORITHMS

LAB 4 AND 5

LUZIA MANUEL - 2021332
PRASHASTI GUPTA- 2021346

```python
class DoublyLinkedList:

    class _Node:
        def __init__(self, element):
            self._prev = None
            self._next = None
            self._element = element


    def __init__(self):
        self._head = None
        self._size = 0


    def is_empty(self):
        return self._size==0


    def __len__(self):
        return self._size

```

CREATED A LIST AND NODE

CHECKING IF LINKED LIST IS EMPTY
WILL RETURN BOOLEAN

LENGTH OF THE LINKED LIST
RETURNING THE SIZE OF THE LINKED LIST

```python
    def push_front(self, e):
        newNode = self._Node(e)
        newNode._next = self._head
        self._head = newNode
        self._size += 1


    def push_last(self, e):
        newNode = self._Node(e)
        tail = self._head
        while tail._next!=None:
            tail = tail._next
        newNode._prev = tail
        tail._next = newNode
        self._size += 1
```

PUSH ELEMENT TO THE FRONTAND CREATING NODE OF THE ELEMENT

SHIFTING THE INTIAL HEAD AND MAKING NEWNODE THE HEAD OF THE LIST AND ADDING NEWNODE TO THE LIST
PUSHING AN ELEMENT TO THE FRONT OF THE LINKED LIST

PUSH ELEMENT TO THE LAST INITIALIZING A TAIL VARIABLE AND GIVING IT HEAD VALUE, FINDING THE TAIL OF THE LINKED LIST AND ADDING NEWNODE TO THE LIST AFTER PUSHING AN ELEMENT TO THE LAST OF THE LINKED LIST

```
44
45      def delete_front(self):
46          if self.is_empty():
47              print ("List is empty")
48              return
49          next_node = self._head._next
50          if (next_node != None):
51              next_node._prev = None
52          item = self._head._element
53          self._head = next_node
54          self._size -= 1
55          return item
56
57      def delete_last(self):
58          if self.is_empty():
59              print ("List is empty")
60              return
61          tail = self._head
62          while tail._next != None:
63              tail = tail._next
64          item = tail._element
65          previous = tail._prev
66          if previous != None:
67              previous._next = None
68          tail._prev = None
69          tail = previous
70          self._size -= 1
71          return item
```

DELETE ELEMENT FROM THE FRONT, FIRST WILL CHECK IF THE LINKED LIST IS EMPTY, AFTER INITIALIZING AN ITEM VARIABLE AND GIVING IT HEAD VALUE
SHIFTING THE INTIAL HEAD AND MAKING NEXT_NODE THE HEAD OF THE LIST AND REDUCING SIZE
DELETE ELEMENT FROM THE LAST, WILL CHECK IF THE LINKED LIST IS EMPTY, FINDING THE TAIL OF THE LINKED LIST , PREVIOUS IS THE TAIL AND DETACHING THE TAIL AFTER THIS REDUCING SIZE

```
74
75      def printlist(self):
76          current = self._head
77          while current._next!=None:
78              print(current._element, end = " ")
79              current = current._next
80          print(current._element)
81
82      def insert_in_between(self, e, i):
83          node = self._Node(e)
84          curr = self._head
85          while curr != None and curr._element != i:
86              curr = curr._next
87          if curr == None:
88              print ("Key not found")
89              return
90          if curr._next == None:
91              curr._next = node
92              node._prev = curr
93          else:
94              next_node = curr._next
95              curr._next = node
96              node._prev = curr
97              node._next = next_node
98              next_node._prev = node
99          self._size += 1
100
```

FOR PRINTING THE LIST WE ITERATE THROUGH THE LIST AND KEEP PRINTING EACH VALUE AS WE GO THROUGH IT.
FOR INSERTING IN BETWEEN WE FIRST FIND THE LOCATION WHERE WE ARE SUPPOSED TO INSERT THEN IF IT THE HEAD WE ASSIGN HEAD._PREV, NODE VALUE. AND IF THE NODE IS TO BE INSERTED ANYWHERE IN BETWEEN THE LIST THEN WE CHANGE ITS PREV AND NEXT TO THE NODES IN BETWEEN WHICH WE ARE SUPPOSED TO INSERT THE ELEMENT

```python
#removing key from the linked list
def remove(self, key):
    if self.is_empty():         #checking if the linked list is empty
        print ("List is empty")
        return

    # find the position of the key
    curr = self._head           #initializing a curr variable and giving it head value
    while curr != None and curr._element != key:        #iterating to find the node which has the key
        curr = curr._next
    if curr == None:            #if key not found
        print ("key not found")
        return

    # if curr is head, delete the head
    if curr._prev == None:
        self.delete_front()
    elif curr._next == None: # if curr is last item
        self.delete_last()
    else: #anywhere between first and last node
        next_node = curr._next
        prev_node = curr._prev
        prev_node._next = next_node
        next_node._prev = prev_node
        curr._next = None
        curr._prev = None
        curr = None
```

FOR REMOVE FUNCTION WE FIRST CHECK IF THE FUNCTION IS EMPTY, IF IT IS THEN WE BREAK. IF NOT EMPTY, WE FIND THE POSITION OF THE KEY BY ITERATING THROUGH THE LINKED LIST. IF NOT FOUND WE RETURN "KEY NOT FOUND".

IF THE KEY IS AT THE HEAD THEN WE DELETE THE HEAD. IF THE KEY IS FOUND ANYWHERE BETWEEN THE FIRST OR THE LAST NODE WE REMOVE THE KEY BY CHANGING ITS NEXT AND PREV TO NONE AND CHANGING THE NEXT AND PREV OF THE NODES IN FRONT AND BEHIND THE CURRENT NODE.

```
#searching an element
def search(self, e):
    count = 0
    if self.is_empty():          #checking if the linked list is empty
        print ("List is empty")
        return False
    curr = self._head            #initializing a curr variable and giving it head value
    while curr != None and curr._element != e:      #finding the e value in the linekd list
        if curr._element==e:
            count+=1             #finding the number of times e is in the linked list
        curr = curr._next
    if curr == None:
        return count
    return count

#deleting all the elements
def del_all(self):
    while(self._head!=None):     #iterating through the list
        node = self._head        #initializing an item variable and giving it head value
        self._head = self._head._next       #shifting the head
        node = None              #assigning head None value
        self._size -= 1          #reducing the size
    return "Deleted all"
```

FOR SEARCH FUNSTION WE FIRST CHECK IF THE LINKED LIST IS EMPTY THEN FIND THE VALUE IN THE LINKED LIST BY ITERATING THROUGH THE LIST STARTING WITH THE HEAD. WE HAVE TO PRINT THE COUNT OF THE NUMBER OF TIMES AN ELEMENT WAS FOUND IN THE LINKED LIST.

DELETE ALL FUNCTION STARTS BY ITERATING THROUGH THE LIST AND WE INITIALISE NODE TO BE THE HEAD AND WE CHANGE THE POSITION OF THE HEAD. THEN NOD IS ASSIGNED THE VALUE NONE.