MODULE CODE AND NAME:

DA3306 – DATA STRUCTURES AND ALGORTIHMS

TITLE OF ASSIGNMENT:

Assignment 1**:** *Data Structures – Circular Doubly Linked List*

NAME:

Muhammad Hidayat Bin Mohd Yusof (20FTT2910)

SUBMISSION DATE:

29th March 2022

MODULE LECTURER:

Norfarrah Muhd Masdi

NAME OF INSTITUTE:

School of Information, Communication and Technology, Polytechnic Brunei

PROGRAMME:

Level 5 Diploma in Data Analytics

Session:

2021 – 2023

Table of Contents

1. Introduction 3
1.1. Background 3
1.2. Purpose 3
2. Circular Doubly Linked List 3
2.1. Singly and Doubly Linked List 3
2.2. Circular Doubly Linked List 4
2.3. Differences from a normal list 4
3. Sorted and Unsorted Circular Doubly Linked List 4
3.1. Unsorted 4
3.2. Sorted 5
3.3. Methods implemented 5
4.Conclusions 14
References 15

1. Introduction

1.1. Background

A list is a common tool for storing data for later use in operations. It is usually mutable allowing items to be inserted and removed easily. On the other hand, as the amount of data increases, making modifications to the list would become cumbersome as it would need to be constantly restructured which would necessitates a different method to store data in a list form. One of those methods is called a Linked list and is the topic of this report specifically a Circular Doubly Linked List.

1.2. Purpose

This report will further elaborate on the topic as well as its creation in python code including all of its variations.

Hence by the end of this report the user will:

- Have a complete understanding of what is a Circular Doubly Linked List alongside its sorted and unsorted variations.
- Understand the differences between a Circular Doubly Linked List and a normal list.
- Understand how they are made in python, with complete explanation of the methods involved.

2. Circular Doubly Linked List

2.1. Singly and Doubly Linked List

In comparison to a normal list, a linked list stores data in nodes that are connected with each other through ‘Links’. A singly linked list is the most basic form as the nodes within it have only one link which is to the next node. These links are used to traverse from one node to another within the correct order. Hence, regardless of how and where the nodes are stored in memory, they will always accurately point to the next following node.

A doubly linked list takes a step further by including a link to the previous node for each node allowing for traversal in both ways of the list. Although it consumes more memory,

it has better performance, than a singly linked list. It also simplifies operations such as deletion because backward traversal is possible.

2.2. Circular Doubly Linked List

Now for the main topic of this report, a Circular Doubly Linked List is similar to a Doubly Linked List except instead of the previous link for the first node(head) pointing to a ‘None’ object, it points to the last node(tail) and instead of the next link for the last node(tail) being pointed to a ‘None’ object, it points to the first node(head). Linking the specified nodes this way essentially eliminates the ‘None’ object from the list allowing for instant traversal from the first node directly to the last node or vice-versa. Additionally, the list would be able to be iterated as many times as needed because after the last node will always lead to the first node.

2.3. Differences from a normal list

A normal list can only traverse forward until it reaches the end. A Circular Doubly Linked List or CDLL on the other hand, may traverse in both directions due to each node possessing a connection to its previous and next node. It may also iterate through each item of the list as many times needed as a CDLL loops to the beginning when it reaches the end.

Furthermore, a normal list stores the references for its elements in the form of a single contiguous (adjacent) memory block whereas a CDLL does not need to because it stores the references within each element (Pregueiro, 2022).

3. Sorted and Unsorted Circular Doubly Linked List

3.1. Unsorted

An Unsorted Circular Doubly Linked List store its elements without any particular order. Items are added at the beginning or top end of the list with each new item added becoming the new head of the list.

3.2. Sorted

A Sorted Circular Doubly Linked List store its elements in ascending order. When an item is being added to the list, it is first compared with all the stored items before being placed.

3.3. Methods implemented

Add(item)

```
def add(self, item):
    if self.len<self.limit:
        self.len+=1
    elif self.limit==0:
        self.len+=1
    else:
        print('\nSize limit reached')
        return

    temp = Node.Node(item)
    if self.head==None:
        self.head=temp
        self.tail=temp
        temp.set_next(self.head)
        temp.set_prev(self.tail)
        self.tail.set_next(temp) #or self.head
        self.head.set_prev(temp)

    else:

        temp.set_next(self.head)
        temp.set_prev(self.tail)
        self.head.set_prev(temp)
        self.tail.set_next(temp)
        self.head=temp
```

Arguably the most important method, this one adds items to the list. When the list is unsorted, items are added at the beginning of the list. Prior to becoming the new head of the list, the item is stored in a node called temp where it is then linked with the tail

node as its previous node and the old head node as it is next node. The temp node is then designated as the new head of the list.

```
    def add(self, item):
        if self.len<self.limit:
            self.len+=1
        elif self.limit==0:
            self.len+=1
        else:
            print('\nSize limit reached')
            return

        current=self.head
        previous=self.tail
        temp = Node.Node(item)

        if self.head==None:
            self.head=temp
            self.tail=temp
            self.tail.set_next(self.head)
            self.head.set_prev(self.tail)
            self.head.set_next(self.tail)
            self.tail.set_prev(self.head)
            return

        elif current.get_next()==self.head:
            if current.get_data()>item:
                self.head=temp
                self.tail=current

            else:
                self.head=current
                self.tail=temp

            self.tail.set_next(self.head)
            self.head.set_prev(self.tail)
            self.head.set_next(self.tail)
            self.tail.set_prev(self.head)
            return
```

```
else:

    while current.get_next()!=self.head:
        if current.get_data()>item:
            if current==self.head:
                self.tail.set_next(temp)
                temp.set_prev(self.tail)
                temp.set_next(current)
                current.set_prev(temp)
                self.head=temp
                return
            else:
                previous.set_next(temp)
                temp.set_prev(previous)
                temp.set_next(current)
                current.set_prev(temp)
                return
        else:
            previous=current
            current=current.get_next()

    if current.get_data()>item:
        temp.set_next(current)
        temp.set_prev(previous)
        current.set_prev(temp)
        previous.set_next(temp)
        self.tail=current
        return
    else:
        current.set_next(temp)
        temp.set_prev(current)
        temp.set_next(self.head)
        self.head.set_prev(temp)
        self.tail=temp
        return
```

For a sorted list it is a bit more complex. The same step where the item is placed in a temp node still occurs but instead of immediately placed at the beginning of the list it is first compared one by one with items in other nodes until a bigger item is encountered or the tail of the list is reached. If a bigger item has been encountered, its node is set as the next node for them temp node and the bigger item's previous node becomes the previous for the temp node. If the tail is reached without a bigger item being found, the item to be

added is compared with the tail. If the item is bigger than the tail, the item is made as the new tail of the list. If it is smaller than the tail, it is placed behind the tail.

Delete(item)

```
def delete(self,item):
    current = self.head
    previous = self.tail
    found = False
    while not found:
        if current.get_data() == item:
            found = True
        else:
            previous = current
            current = current.get_next()

    previous.set_next(current.get_next())
    current.set_prev(previous)
```

This method deletes the first match to a specified item within the list. If current node matches the specified item, the previous node is connected with the current's next node and vice versa. Modifying the links this way essentially skips the match node, removing it from the list.

Deleteall(item)

```
def deleteall(self,item):

    current = self.head
    previous= self.tail
    while current.get_next()!=self.head:
        if current.get_data() == item:
            if current.get_prev()==self.tail:
                self.head=current.get_next()
                self.tail.set_next(self.head)
                current=current.get_next()


            else:
                previous.set_next(current.get_next())
                current=current.get_next()
        else:
            previous=current
            current = current.get_next()
    if current.get_data() == item:
        previous.set_next(current.get_next())
        print('\ndelete all item '+"'"+str(item)+"'"+' finished')
    else:
        print('\ndelete all item '+"'"+str(item)+"'"+' finished')
```

This method deletes/removes all items within the list that matches the specified item until the tail is reached. Once the tail is reached, it is also compared with the specified item where if it matches it is deleted and the text ‘delete all item ‘item’ finished’. If the tail does not match the same text is printed because the end of the list has been reached.

DeletebyIndex(index)

```
def deletebyindex(self,index):
    current=self.head
    previous=self.tail
    pos=1
    if index>self.len:
        print('\nIndex',index,'does not exist')
        return
    else:
        while pos!=index:
            pos=pos+1
            previous=current
            current=current.get_next()
        if current==self.head:
            self.head=current.get_next()
            self.tail.set_next(self.head)
        else:
            previous.set_next(current.get_next())
            current.set_prev(previous)
```

This method removes the node/item at the specified index. If the specified index is greater than the size of the list, the text ‘Index n does not exist’ is printed and the method is returned. If the index is smaller than the size of the list, then the items in the list are iterated until the index is reached and the item at the index is deleted.

AddtoFront(item)

```
def addtofront(self,item):
    if self.len<self.limit:
        self.len+=1
    elif self.limit==0:
        self.len+=1
    else:
        print('\nSize limit reached')
        return
    current=self.head
    previous=self.tail

    temp=Node.Node(item)

    previous.set_next(temp)
    temp.set_prev(previous)
    temp.set_next(current)
    current.set_prev(temp)

    self.tail=previous.get_next()
```

This method adds items at the tail end of the list. Similar to the add(item) method, if the size limit of the list has already been reached when the addtofront(item) method is executed then it is returned. If not, the item is added to the right most end of the list where it is designated as the new tail.

```
def addtofront(self,item):
    if self.len<self.limit:
        self.len+=1
    elif self.limit==0:
        self.len+=1
    else:
        print('\nSize limit reached')
        return
    current=self.head
    previous=self.tail

    temp=Node.Node(item)
    if previous.get_data()<item:
        previous.set_next(temp)
        temp.set_prev(previous)
        temp.set_next(current)
        current.set_prev(temp)

        self.tail=previous.get_next()
    else:
        self.add(item)
```

For a sorted list it is mostly the same except, if the tail of the list is bigger than the item to be added, the item is added to the list with the sorted list add(item) method because it would need to be placed into the list in the right order.

Search(item)

```
def search(self,item):
    current=self.head
    previous=self.tail
    while current.get_next()!=self.head:
        if current.get_data()==item:
            print('')
            return print(True)
        else:
            previous=current
            current=current.get_next()
    if current.get_data()==item:
        print('')
        return print(True)
    else:
        return print(False)
```

This method returns TRUE if the item exists within the list and returns False if no matches have been when it reaches the end of the list.

SearchIndex(item)

```
def searchindex(self,item):
    current=self.head
    previous=self.tail
    index=1
    while current.get_next()!=self.head:

        if current.get_data()==item:
            print('')
            return print(item,'is found at index',index)
        else:
            index=index+1
            previous=current
            current=current.get_next()
    if current.get_data()==item:
        print('')
        return print(item,'is found at index',index)
    else:
        return print('Item not found')
```

This method iterates through each item in the list until an item within the list matches the item being searched where the matched item is returned alongside the index it was found in. After the tail is compared and concluded as not a match, the string ‘Item not found’ is returned.

SearchbyIndex(index)

```
def searchbyindex(self,index):
    current=self.head
    previous=self.tail
    pos=0
    if index>self.len:
        print('\nIndex',index,'does not exist')
        return
    else:
        while current.get_next()!=self.head:
            pos=pos+1
            if pos==index:
                return print(current.get_data(),'was found at position',pos)
            else:
                previous=current
                current=current.get_next()
        pos=pos+1
        if pos==index:
            print(current.get_data(),'was found at position',pos)
        else:
            return print('Index is not found')
```

Similar to deletebyindex() but instead of deleting the item at the specified index, this method searches for the node at the specified index and returns the item within it. Of course, if the index that is searched for is greater than the size of the list the method would not be fully executed.

Size()

```
    def __init__(self):
        self.head = None
        self.tail = None
        self.limit= 0
        self.len=0
```

```
def size(self):
    return print('\nSize=',self.len)
```

The Size() method returns ‘self.len’ which is specified in the _init_ function. It counts the length/size of the list so will increase when items are added to the list.

SizeLimit()

```
    def __init__(self):
        self.head = None
        self.tail = None
        self.limit= 0
        self.len=0
```

```
def sizelimit(self):
    return print('\nSizelimit=',self.limit)
```

The SizeLimit() method returns ‘self.limit’ that was also specified in the _init_ function. ‘self.limit’ defines the size limit of the list therefore, ‘self. len’, as in the current size of the list is compared to it when new items are added where after the limit is reached, further items will not be added.

SetSizeLimit(number)

```
    def __init__(self):
        self.head = None
        self.tail = None
        self.limit= 0
        self.len=0
```

```
def setsizelimit(self,number=10):
    self.limit=number
```

This method imposes a specified size limit onto the list. This is done by changing the value in self.limit to the specified number. If the number argument is empty when the method is executed, self.limit will be set to a default value of 10.

4.Conclusions

As shown a CDLL is much more complicated than a normal list, having said that, this complexity brings with it many new ways for data to be utilized. With the elements able to always find its way to the next, it expands the possibilities to what can be done within python which was deemed impossible to achieve with a normal list. Queues, stacks, infinite looping without losing track of each individual element and much more are to name a few to what can be done.

Hence, with this report as a guide, the reader can create their own CDLL by using the methods explained above and use the list for whatever they see as fit.

References

i. Pregueiro, P. (2022, 3 29). *Linked Lists in Python: An Introduction*. Retrieved from Realpython: https://realpython.com/linked-lists-python/