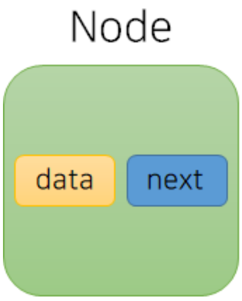
Lab05 - Linked Linear Data Structures

1. Implement Line Editor

Node

Node is data point. It includes data and pointer.

- data: simple data
- pointer(next): Having a location value for another node

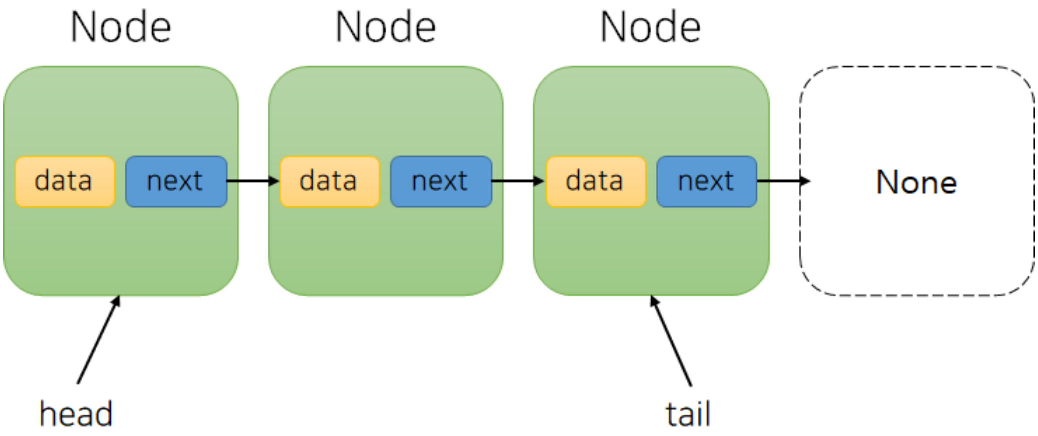


```
class Node:
    def __init__(self, data = None, nxt = None):
        self.data = data
        self.next = nxt

    def __str__(self):
        return "(" + str(self.data) + ")"

    def setData(self, data):
        self.data = data

    def getData(self):
        return self.data

    def setNext(self, nxt):
        self.next = nxt

    def getNext(self):
        return self.next
```

SinglyLinkedList

Singly linked list is data structure in which all nodes have **data** and **next pointer** and are **connected to other nodes through next pointer**.

The position of the first node is called the `head` pointer, and the position of the last node is called the `tail` pointer.



So what happens if there is no tail pointer?

Fortunately, there is no problem if there is no tail. Because**tail can be accessed from head through the node's next.**

init(self)

- Initialize head to None.

```
def __init__(self):
        self.head = None
```

str(self)

- Return node data in the form of "(data)". (through move to next of node)

```
def __str__(self):
        temp = self.head
        string_repr = ""
        while temp:
            string_repr += str(temp) + "->"
            temp = temp.next

        return string_repr + "END"
```

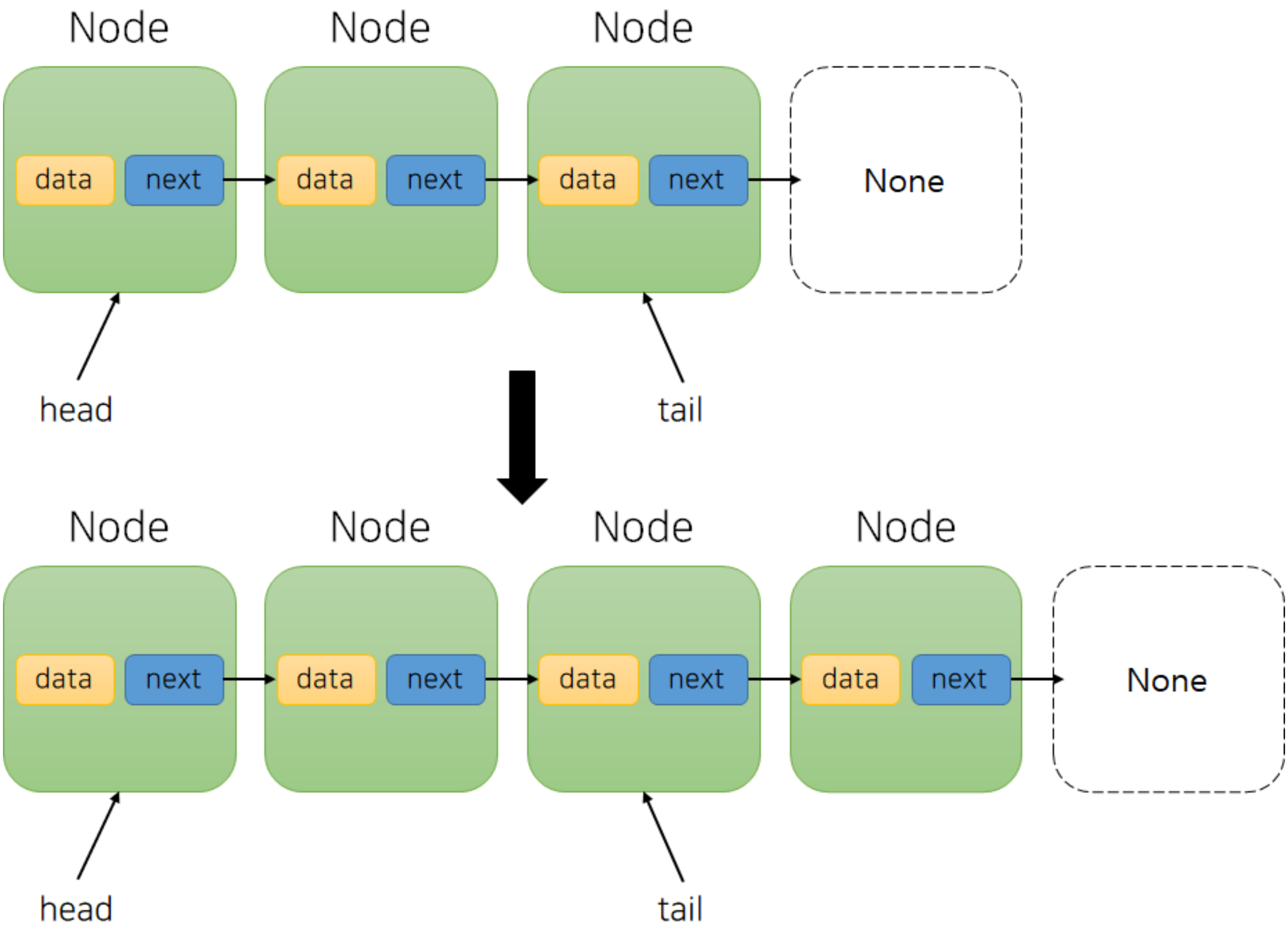
addFront(self, data)

- If head is None, insert new node into the head.
- If head is not None, insert the new node with the next of the head, and change the head to the new node.

```
def addFront(self, data):
        newNode = Node(data)
        if self.head == None:
            self.head = newNode
        else:
            newNode.next = self.head
            self.head = newNode
```

addRear(self, data)

- If head is None, add the node as addFront() method.
- If head is not None, the tail node is reached through 'while statement' and a new node is assigned to the next of the tail node.



```
def addRear(self, data):
        if self.head == None:
            self.addFront(data)
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = Node(data)
```

addAt(self, pos, elem)

- Get the previous node(before) of pos location by getNodeAt() method and do the following:
    - If before is None, set the next of new node to head and replace head with a new node.
    - If before is not None, set the next of new node to next of before and replace next of before to new node.

```
def addAt(self, pos, elem):
        before = self.getNodeAt(pos - 1)
        if before == None:
            self.head = Node(elem, self.head)
        else:
            node = Node(elem, before.next)
            before.next = node
```

deleteFront(self)

- Save head in temp, change head to next of head, change the next of temp to None, and return temp.

```
def deleteFront(self):
        temp = self.head
        if self.head:
            self.head = self.head.next
            temp.next = None
        return temp
```

deleteRear(self)

- if next of head is None

getNodeAt(self, pos)

- Move the node through next of the node by while statement, and return corresponding node.

```
def getNodeAt(self, pos):
        if pos < 0:
            return None
        node = self.head
        while pos > 0 and pos != None:
            node = node.next
            pos -= 1
        return node
```

printList(self, msg)

- Print data by moving from head node to last node through next of node.

replaceDataAt(self, pos, data)

- Use the ReplaceDataAt() method to obtain the node, and if the node is None, allocate the data.

```
def replaceDataAt(self, pos, data):
        node = self.getNodeAt(pos)
        if node != None:
            node.data = data
```

isEmpty(self)

- Return (head == None)

```
def clear(self):
        self.head = None
```

clear(self)

- Allocate None in head.

```
def isEmpty(self):
        return self.head == None
```

getSize(self)

- Count the number of nodes by moving through next of node.

```
def clear(self):
        self.head = None
```

reverseList(self)

- Move from head node to last node in turn, replacing the next of that node with the previous node.

```
def reverseList(self):
        prev = None
        temp = self.head

        while temp:
            next_node = temp.next
            temp.next = prev
            prev = temp
            temp = next_node

        self.head = prev
```

findData(self, val)

- Move from head node to last node through next of node, return the node if data exists and return None if not.

```
def findData(self, val):
        node = self.head
        while node is not None:
            if node.data == val:
                return node
            node = node.next
        return node
```

LineEditor

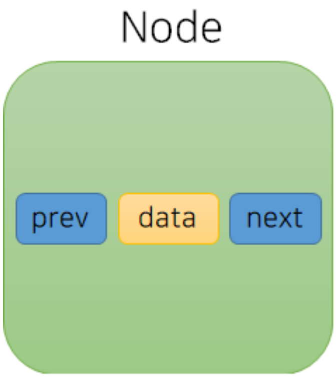
Implement LineEditor by singly linked list.

```
from SinglyLinkedList import *

class LineEditor:
    def __init__(self):
        self.list = SinglyLinkedList()

    def runLineEditor(self):
        while True:
            command = input("i-insert, d-delete, r- replace, p- print, l- loadfile, s- writefile, q-quit -> ")
            if command == 'i':
                self.addLine()
            elif command == 'd':
                self.deleteLine()
            elif command == 'r':
                self.replaceLine()
            elif command == 'p':
                self.printByLine()
            elif command == 'l':
                self.loadFromFile()
            elif command == 's':
                self.writeToFile()
            elif command == 'q' :
                return

    def addLine(self):
        pos = int(input(" input line number: ") )
        str = input( "input line text ")
        self.list.addAt(pos, str)

    def deleteLine(self):
        pos = int( input ("input line number: "))
        self.list.deleteAt(pos)

    def replaceLine(self):
        pos = int(input("input line number: "))
        str = input("input modified: ")
        self.list.replaceDFateAT(pos, str)

    def printByLine(self):
        self.list.printByLine()

    def loadFromFile(self):
        filename = 'test.txt'
        with open(filename, "r") as infile:
            lines = infile.readline()
            for line in lines:
                self.list.addAt(self.list.getSize(), line.rstrip('\n'))

    def writeToFile(self):
        filename = 'test.txt'
        with open(filename, 'w') as outfile:
            sz = self.list.getSize()
            print(sz)
            for i in range(sz):
                outfile.write(self.list.getDataAt(i) + '\n')
```

2. Implement Sparse Polynomial

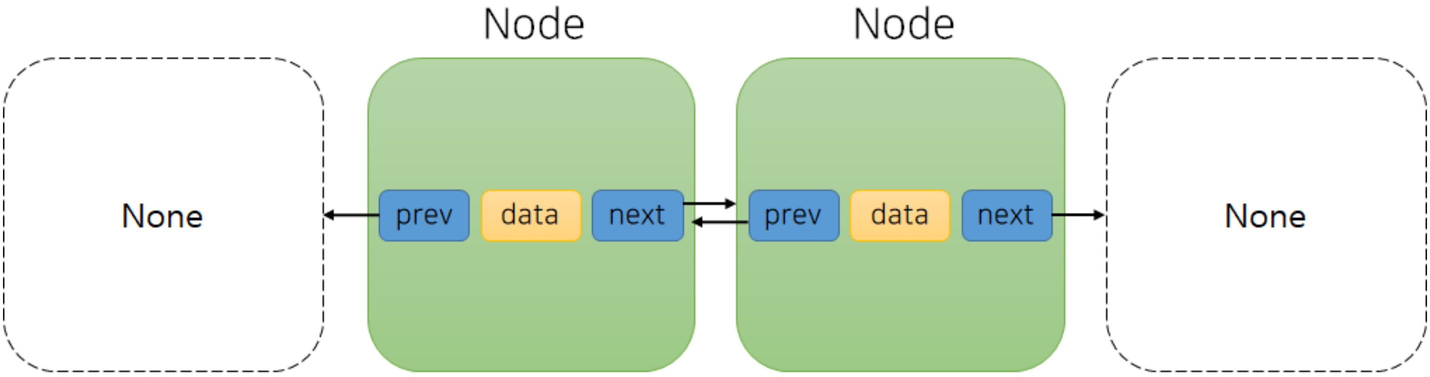
Node2

- Node2 is data point. It includes data and pointer.
    - data: simple data
    - pointer(**prev** and **next**): Having a location value for another node
    - prev: previous node pointer
    - next: next node pointer



DoublyLinkedList

Doubly linked list is data structure in which all nodes have **data** and **prev pointer**, **next pointer** and are **connected to other nodes through prev pointer and next pointer**. (prev pointer have location of previous node.)



All methods in double linked list are almost identical to all methods and logic in the singly linked list.

However, when working on nodes, not only 'next' but also 'prev' should be dealt with.

```
from Node2 import Node2

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

    def __str__(self):
        temp = self.head
        string_repr = ""
        while temp:
            string_repr += str(temp) + "->"
            temp += temp.next

        return string_repr + "END"

    def addFront(self, data):
        newNode = Node2(None, data, None)
        if self.isEmpty():
            self.head = newNode
        else:
            newNode.next = self.head
            self.head.prev = newNode
            self.head = newNode

    def addRear(self, data):
        newNode = Node2(None, data, None)
        if self.isEmpty():
            self.head = newNode
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = newNode
            newNode.prev = temp

    def addAt(self, pos, data):
        if pos == self.getSize():
            self.addRear(data)
        elif pos == 0:
            self.addFront(data)
        else:
            newNode = Node2(None, data, None)
            before = self.getNodeAt(pos - 1)
            if before == None:
                print("This node doesn't exist in DLL")

            newNode.next = before.next
            before.next= newNode
            newNode.prev = before
            if newNode.next == None:
                newNode.next.prev = newNode

    def deleteFront(self):
        if self.isEmpty():
            print("List is Empty..")
            return None

        temp = self.head
        if temp.next == temp.prev:
            self.head = None
            return temp
        else:
            self.head = temp.next
            self.head.prev = None
            return temp

    def deleteRear(self):
        if self.isEmpty():
            print("List is Empty..")
            return None

        temp = self.head
        if temp.next == temp.prev:
            self.head = None
            return temp
        else:
            while temp.next:
                temp = temp.next
            temp.prev.next = None
            temp.prev = None
            return temp

    def deleteAt(self, pos):
        temp = Node2()
        if pos == self.getSize():
            temp = self.deleteRear()
        elif pos == 0:
            temp=self.deleteFront()
        else:
            before = self.getNodeAt(pos - 1)
            if before is None:
                print("This node doesn't exist in DLL")
                return

            temp = before.next
            before.next=temp.next
            temp.next.prev=before
            temp.next=None
            temp.prev=None

        return temp

    def getNodeAt(self, pos):
        if pos<0 or pos> self.getSize():
            print("invalid position")
            return None
        temp = self.head
        while pos > 0 and temp != None:
            temp = temp.next
            pos -=1
        return temp

    def getDataAt(self, pos):
```

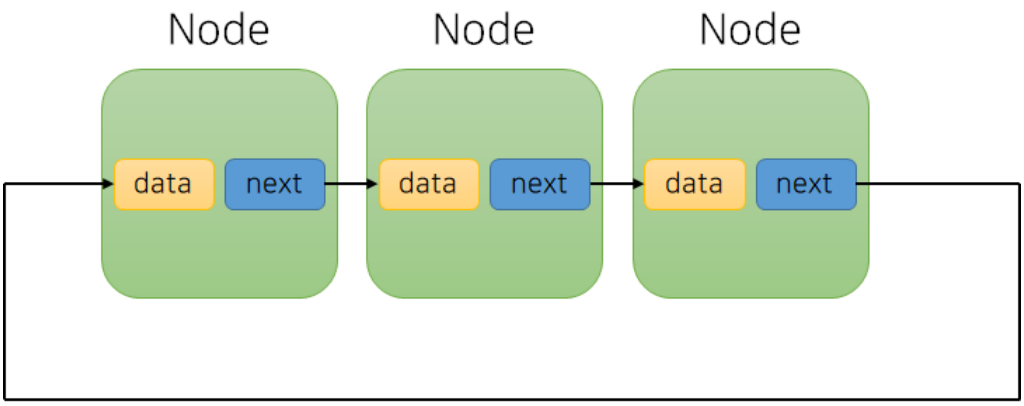
```
    def getDataAt(self, pos):
        node = self.getNodeAt(pos)
        if node == None:
            return None
        else:
            return node.data

    def replaceDataAt(self, pos, data):
        node = self.getNodeAt(pos)
        if node != None:
            node.data = data

    def isEmpty(self):
        return self.head == None

    def clear(self):
        self.head = None

    def getSize(self):
        node = self.head

        count = 0
        while node is not None:
            node = node.next
            count += 1
        return count

    def findData(self, val):
        node = self.head
        while node is not None:
            if node.getData() == val:
                return True
            node= node.next
        return False
```

3. Using circular linked list to solve Josephus Problem

CircularLinkedList

Circular linked list is data structure in which all nodes have **data** and **next pointer** and are **connected to other nodes through prev next pointer**.

And, **The last node and the first node are connected by pointer.**



The logic of methods is similar to the linked list, just like the double linked list.

```
from Node import Node

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

    def __str__(self):
        temp = self.head
        string_repr = ""
        string_repr += str(temp) + "->"
        temp = temp.next

        while temp != self.head:
            string_repr += str(temp) + "->"
            temp += temp.next
        return string_repr + "END"

    def addFront(self, data):
        newNode = Node(data)
        if self.isEmpty():
            self.head = newNode
            newNode.next = self.head
            self.head.next = self.head
        else:
            newNode.next = self.head.next
            self.head.next = newNode

    def addRear(self, data):
        newNode = Node(data)
        if self.head == None:
            self.head = newNode
            newNode.next = self.head
            self.head.next = self.head
        else:
            newNode.next = self.head.next
            self.head.next = newNode
            self.head = newNode

    def addAt(self, pos, elem):
        if pos == self.getSize():
            self.addRear(elem)
            return
        if pos == 0:
            self.addFront(elem)
            return

        newNode = Node(elem)
        before = self.getNodeAt(pos - 1)
        if before is None:
            print("This node doesn't exist in CLL")
            return

        newNode.next = before.next
        before.next = newNode

    def deleteFront(self):
        if self.isEmpty():
            print("List is Empty..")
            return None
        temp = self.head

        if (temp == temp.next):
            self.head = None
            return temp
        else:
            temp=self.head.next
            self.head.next = temp.next
            temp.next=None
            return temp

    def deleteRear(self):
        temp = self.head
        if self.isEmpty():
            print("List is Empty..")
            return None
        if self.head==self.head.next:
            self.head=None
            return temp
        else:
            before = self.getNodeAt(self.getSize()-2)
            self.head=before
            self.head.next=temp.next
            temp.next=None
            return temp

    def deleteAt(self, pos):
        temp = Node()
        if pos==self.getSize()-1:
            temp=self.deleteRear()
        elif pos == 0:
            temp = self.deleteFront()
        else:
            before = self.getNodeAt(pos-1)
            if before is None:
                print("This node doesn't exist in DLL")
                return
            temp = before.next
            before.next=temp.next
            temp.next=None
        return temp

    def getNodeAt(self, pos):
        if (pos <0 or pos > self.getSize()):
            return None
        temp = self.head
        if self.head is not None:
            while(True):
                temp = temp.next
                pos -= 1
                if(pos < 0):
                    break
        return temp

    def printList(self, msg = ""):
        print(msg, end='')
        temp = self.head

        if self.head is not None:
            while(True):
                print(temp, end='->')
                temp = temp.next
                if(temp == self.head):
                    break
        print()

    def replaceDataAt(self, pos, data):
        node = self.getNodeAt(pos)
        if node != None:
            node.data = data

    def isEmpty(self):
        return self.head == None

    def clear(self):
        self.head = None

    def getSize(self):
        temp = self.head
        count = 0
        if self.head is not None:
            while(True):
                count += 1
                temp = temp.next
                if (temp == self.head):
                    break
        return count

    def findData(self, val):
        node = self.head
        while node is not None:
            if node.getData() == val:
                return True
            node= node.next
        return False
```

JosephusProblem

Implement JosephusProblem by singly linked list.

```python
from CircularLinkedList import *

class JosephusProblem:
    def __init__(self, n=10, m=3):
        self.list = CircularLinkedList()
        self.n = n
        self.m = m
        for i in range(1, n + 1):
            self.list.addFront(i)

    def runJosephus(self):
        print(self.list)
        temp = self.list.head.next
        count = 0
        while (True):
            temp = temp.next
            count += 1

            if count == self.m:
                temp2 = temp.next
                pos = self.list.findPos(temp)
                print("Eliminated ->", self.list.deleteAt(pos))
                temp = temp2
                print(self.list)

                count = 0

            if (temp == temp.next):
                print("Selected ->", temp)
                break
```