Singly Linked List and different functions

Tuan Hung Nguyen
Student ID: 104048305
Bachelor of Computer Science
Swinburne University of Technology
Hawthorn, VIC 3122
Email: nthung2k4@gmail.com

0. Abstract

When it comes to data structure, we have a bunch of way to organize data, linked list is one of the most popular and effective way to manage and implement. Simultaneously, the application of linked list is universal. This report will discuss how it works, the complexity of linked list and giving the insight of linked list's application.

1. Introduction

1.1 What is data structure?

Based on the objective, data structure is the way of organizing data so that it could be used effectively in different way. The idea for data structure is reducing the space and time complexity of different tasks.

1.2 Category of linked list in Abstract Data Types (ADT)

While ADT is an abstraction of a data structure which provides only interface to which a data structure must adhere and it is divided into different type including list, queue, map, vehicle, linked list belongs to list type.

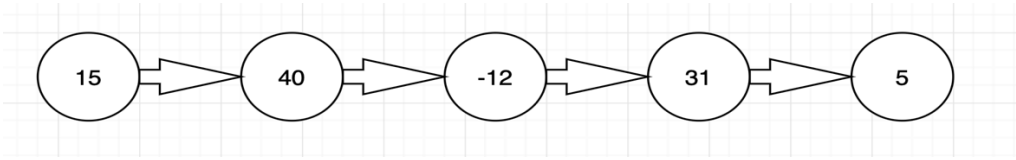
1.3 Why is linked list?

The reason for using linked list because it is dynamic and easily to insert or delete each element in the list.

2. Theoretical background

2.1 Linked list definition

Linked list is a linear data structure. While each element does not link with other at a contiguous location, themselves are nodes and connected by the pointer.

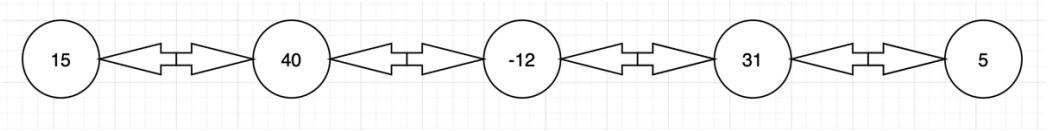


2.1.1 Linked list node

The 2 keys node of the basic linked list is head and tail node which are 2 circles with the value 15 and 5 in the illustration 2.1.1. These two nodes take the role as the start orientation for a list.

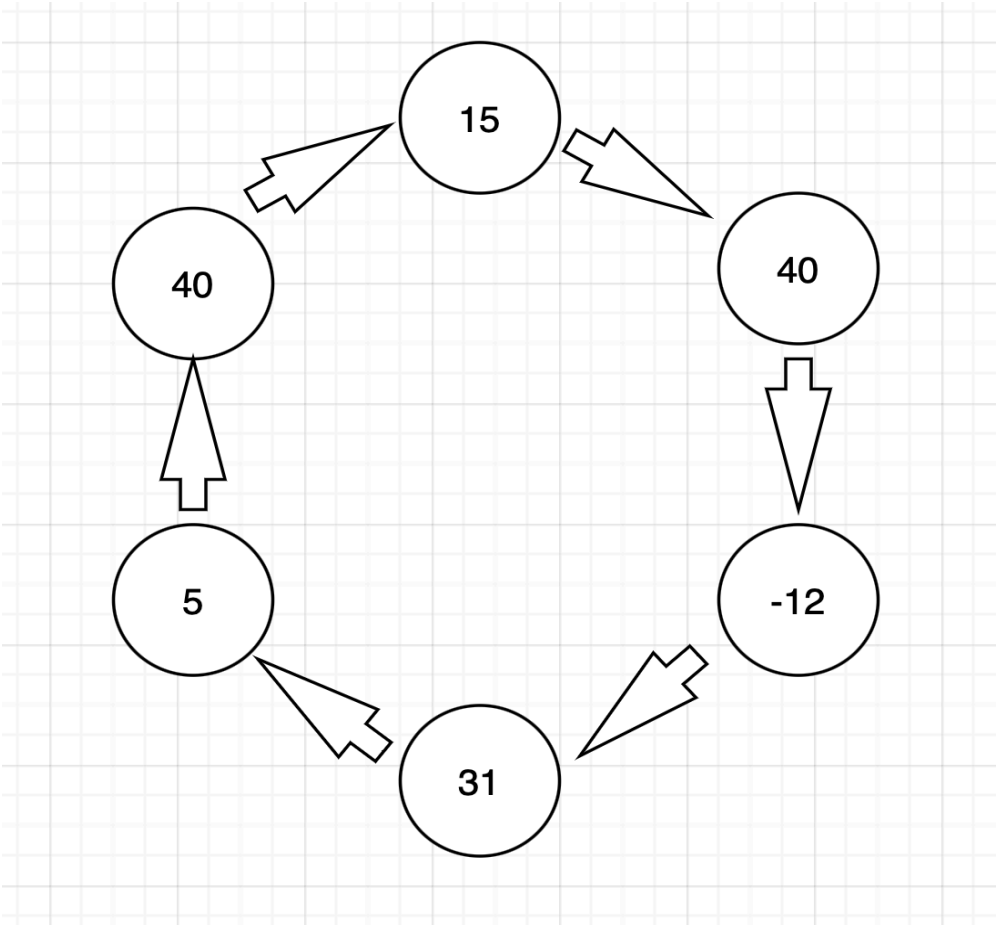
2.2 Variations of Linked list

Basically, linked list has 3 main variations except the basic singly linked lists. The first variation is doubly linked lists. In this variation. While in singly linked list, the pointer will point from the before the next ones and has not vice versa, but doubly linked list could do the opposite thing. From the next node, it could point in both directions.



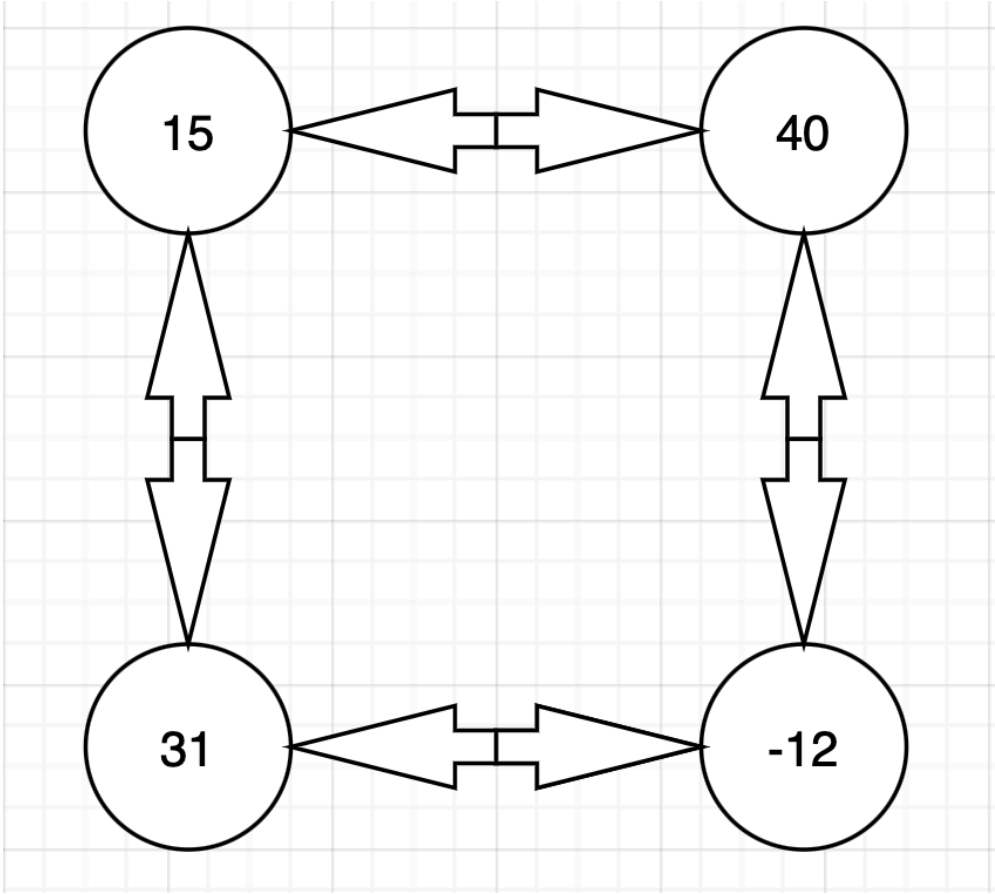
2.2.1 Doubly linked list node

The second variation of linked list is circular linked list. With the tail node point to the head node instead of null as the singly linked list, circular linked lists could return from the tail to the head continuously.



2.2.2 Circular linked lists node

The third variation is doubly circular linked lists. This variation is the combination between doubly linked lists and circular linked lists. The tail node point to the head node and it also could point back to the before node.



2.2.3 Doubly circular linked lists node

3. Fundamental modification of singly linked list

3.0 Generating a singly linked list with node

Basically, linked list bases on creating multiple node class and assemble each other with next_node accessor. (the codes are built on python language)

```python
class Node:
    def __init__(self, data =None, next = None):
        self.data = data
        self.next = next
class LinkedList:
    def __init__(self):
        self.head = None
```

With a list of data, user will expect to modify it in several way such as addition, insertion, deletion, and reversion

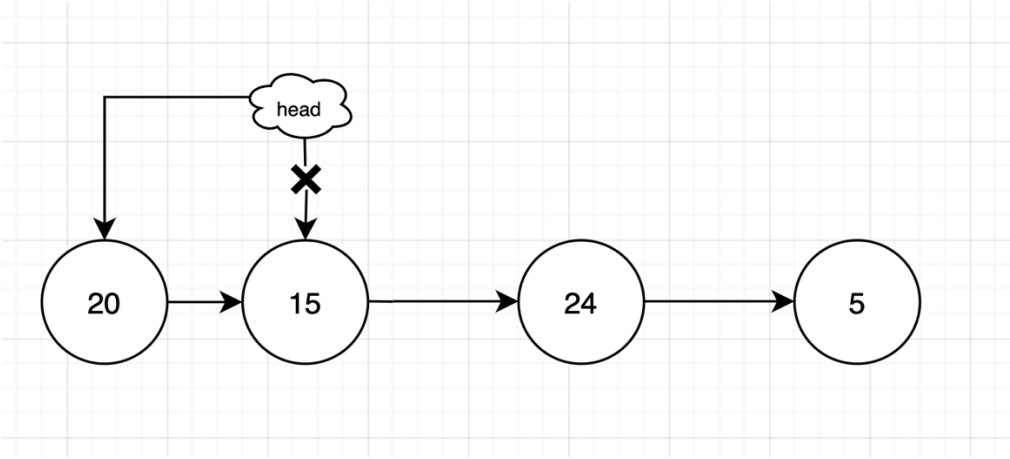
3.1 Addition and insertion

Considering to an empty linked list, to add the new data to the node, we must choose the data and the location of the node. Firstly, when adding the new data to the tail of the list, we must assign a new node for the next node of the old last node with the next node of the new node is null.

```ruby
  def add_last(node)
    if @head.nil?
      @head = node
      return
    end

    iterate do |curr_node|
      if curr_node.next_node.nil?
        curr_node.next_node = node
        return
      end
    end
  end
```

Furthermore, we could modify the code from adding at the location of the last element to adding in anywhere in the list. From here, we have several different cases. The first case is insert at the front of the list. The head will point to the new node instead of the old head node, and the new head node will point to the old head

node.

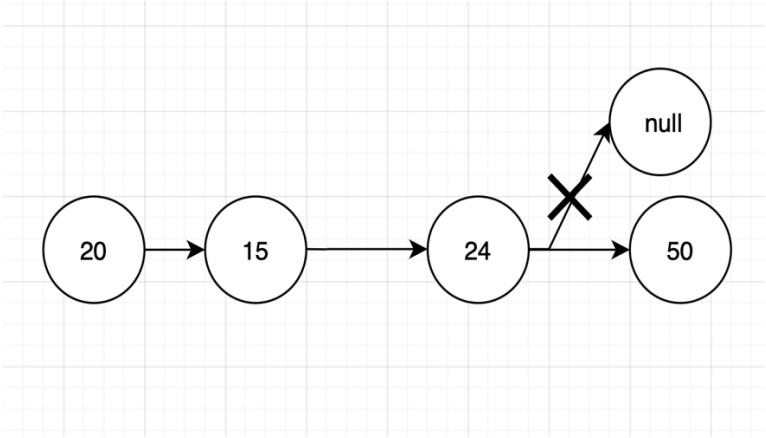


3.1.2 Insert at the front of the list

```
def insert_at_begining(self,data):
        node = Node(data,self.head)
        self.head = node
```

*Complexity analysis: We have a pointer to the head, and we can directly attach a node and change the pointer. So the Time complexity of inserting a node at the head position is O(1) as it does a constant amount of work.

The second case is insert at the end of the list. At this time, we must move the temp node to the last position and generate a new node which is pointed by the old last node.

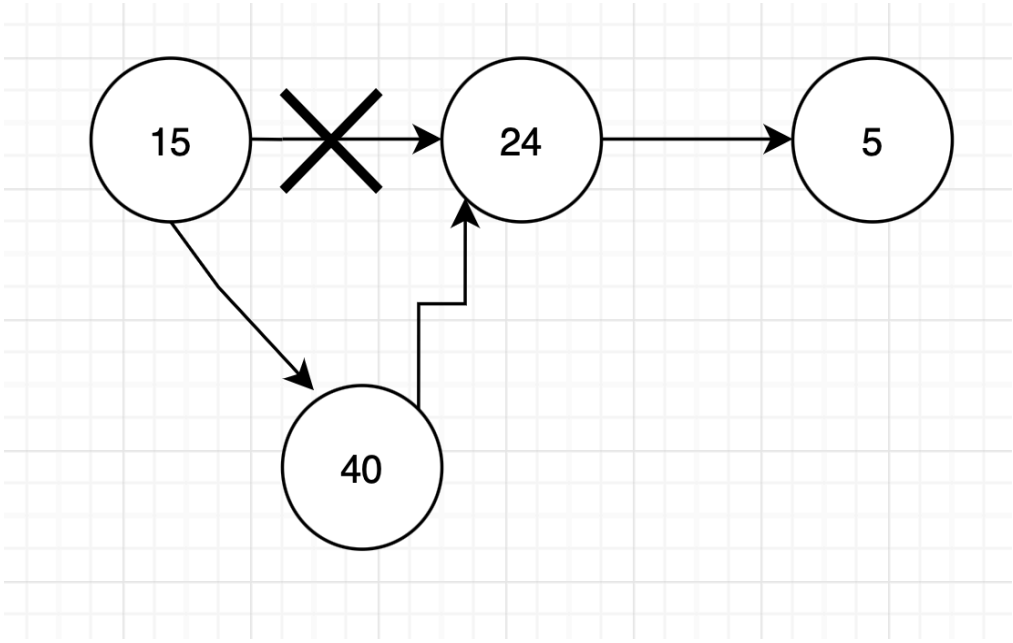


3.1.3 Inserting at the end of list.

```
def insert_at_end(self,data):
        if self.head is None:
            self.head = Node(data,None)
            return

        itr = self.head
        while itr.next:
```

```
            itr = itr.next
        itr.next = Node(data,None)
```

The last case is inserting from anywhere. The basic logic of this is from the location of node people want to insert a new node, it must point to the new node rather the old next node, and the new node will point to the next node in the list.



3.1.4 Insert a new node

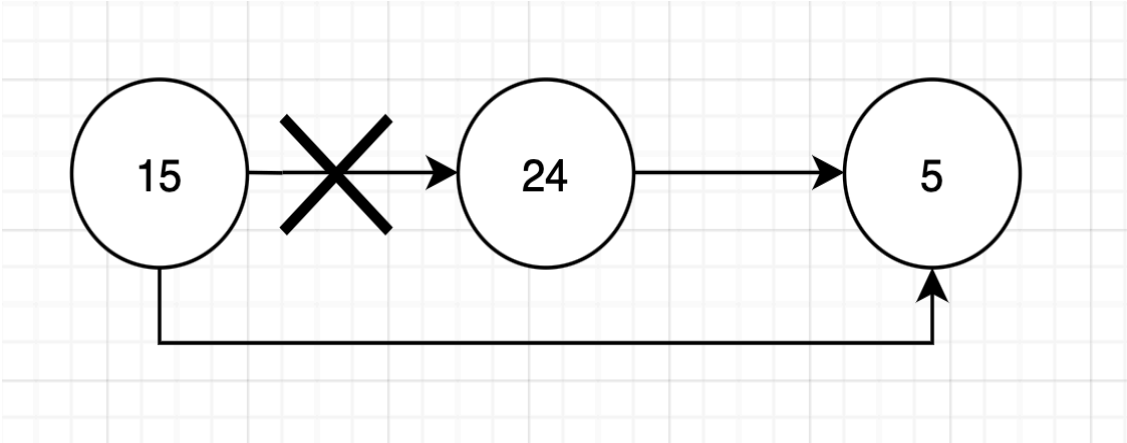
```
def insert(idx, node)
    if idx.zero?
      add_first(node)
      return
    end

    iterate do |curr_node, count|
      if count == idx - 1
        old_next = curr_node.next_node
        curr_node.next_node = node
        node.next_node = old_next
        return
      end
    end
  end
```

*Complexity analysis: Because in both of these cases, linked list must run from the head to the indicated to be able to add the

new node, therefore, the complexity of adding a new data to the last element will be O(n) with n is the index of the list.

3.2 Deletion and replacement

When it comes to deleting a node in the list, in the position of chosen node, the next_node accessor is simply jumps over the node that user want to delete.



3.2.1 Delete a node

In a more specific insight to the code, it must also run from the head to the chosen node, then the node.next will be replaced with node.next.next

```
        def remove_at(self, index):
    if index < 0 or index>= self.get_length():
        raise Exception('Invalid index')
    if index ==0:
        self.head = self.head.next
        return
    count = 0
    itr = self.head
    while itr:
        if count == index -1 :
            itr.next = itr.next.next
            break
        itr = itr.next
        count += 1
```

4.Traversal in singly linked list

4.1 Length of the list

Basically, we have 2 way to get the length of the list, the first one is iterative way and the other one is recursive way.

The first way is simply by run the list from the head until next node is null with a count variable.

```
def get_length(self):
        count = 0
        itr = self.head
        while itr:
            count +=1
            itr = itr.next
        return count
```

The second way is by using recursion, calling the function of get count by itself.

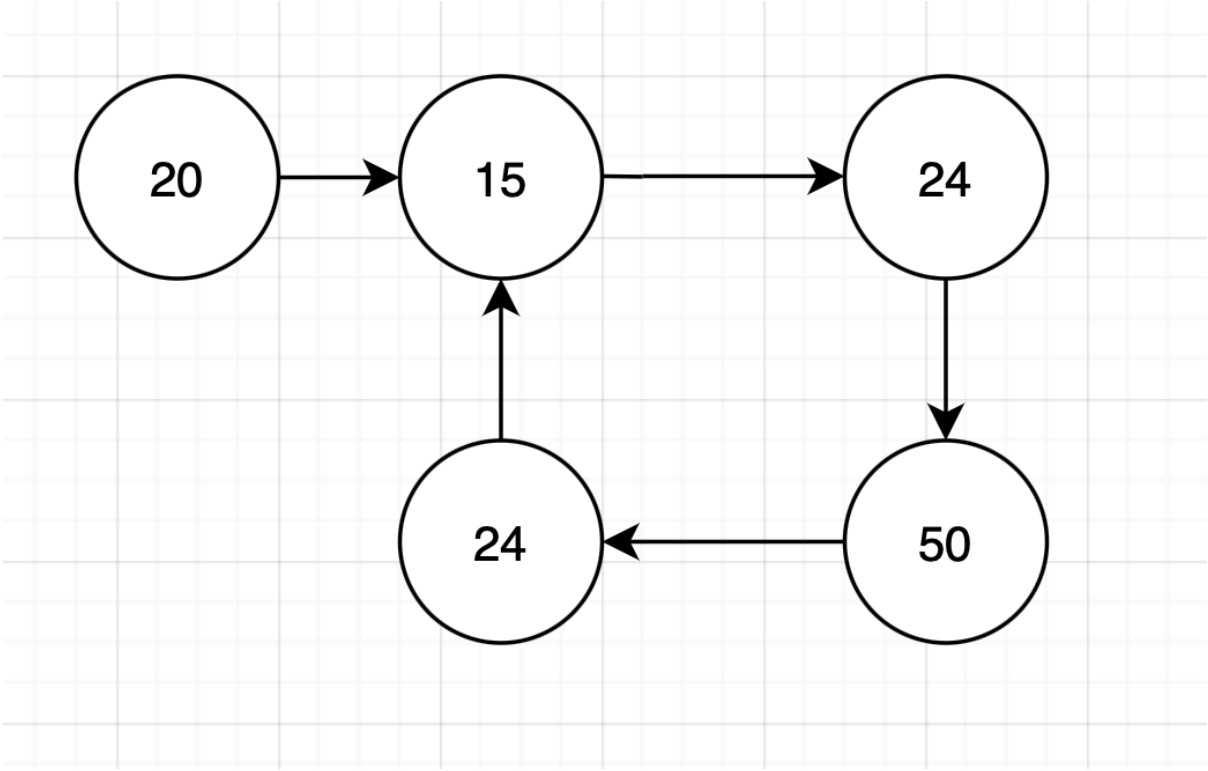
```
 def getCountRec(self, node):
        if (not node):
            return 0
        else:
            return 1 + self.getCountRec(node.next)
    def getCount(self):
        return self.getCountRec(self.head)
```

*Complexity analysis: while the fundamental technical way is different, the basic logic is almost the same that run until the last node is null, therefore, the big O notation is O(n) with n is the length of the list.

4.2 Loop detection in linked list

One of the errors could occur in linked list that the loop could happen when a node point to a previous node which is already exist. That cause several problems related to the length of the list or accessing

previous nodes.



4.2.1 Looping in linked list.

The solution is given here is using a hash map. By pushing each next node into the hash map, they could check if the node is already existed in the hash map or not.

```
def detectLoop(self):
        s = set()
        temp = self.head
        while (temp):
            if (temp in s):
                return True
            s.add(temp)
            temp = temp.next

        return False
```

Complexity analysis: O(N) The loop only run until a node is founded in the hash map.

5. Search in linked list

5.1 Find the node number N in the list from head

To implement a function to search the node from number N, by put a loop throughout the list, until the index equal the number N gave by user, return the value of the node.

```
def getN(self, index):
        current = self.head
        count = 0
        while (current):
            if (count == index):
                return current.data
            count += 1
            current = current.next
        assert(false)
        return 0
```

*Complexity analysis: Because it have to run through the list with N times, therefore the O notation will be O(n).

5.2 Find the node number N in the list from tail

When search from the opposite side of the list, we will do almost the same with search from the head with the different that the index will equal length – n + 1

```
def getN(self, index):
        current = self.head
        count = 0
        len = getLength(self):
        while (current):
            if (count == len - index + 1):
                return current.data
            count += 1
            current = current.next
        assert(false)
        return 0
```

*Complexity analysis: Because it have to run through the list with N times, therefore the O notation will be O(M) with M = Length of list – N + 1.

5.3 Find the node in the middle of the list

In this function, I use a method with 2 parallel pointers named fast and slow to find the middle node. By moving the slow pointer one and fast pointer two nodes, that means, when the fast pointer come to the tail, the slow pointer will move exactly to the middle of the list.

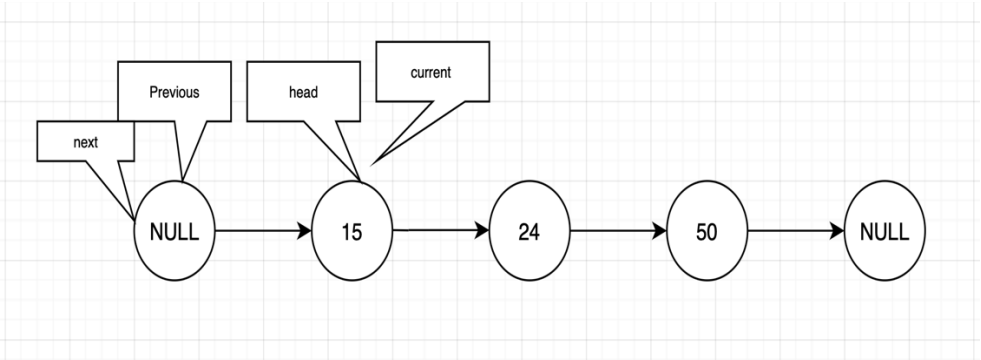
```
def get_middle(self,head):
        if head is None:
            return head
        slow = head
        fast = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        return slow
```

*Complexity analysis: By just moving half of the list without taking the length of the list, the time complexity is reduced significantly, therefore the big O notation is just O(N/2) with N is the length of the list.

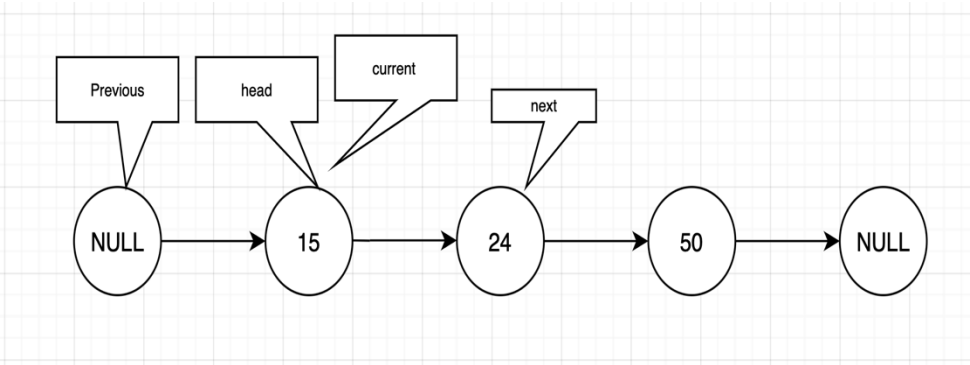
6. Reversal and rotation in linked list

6.1 Reversing a linked list

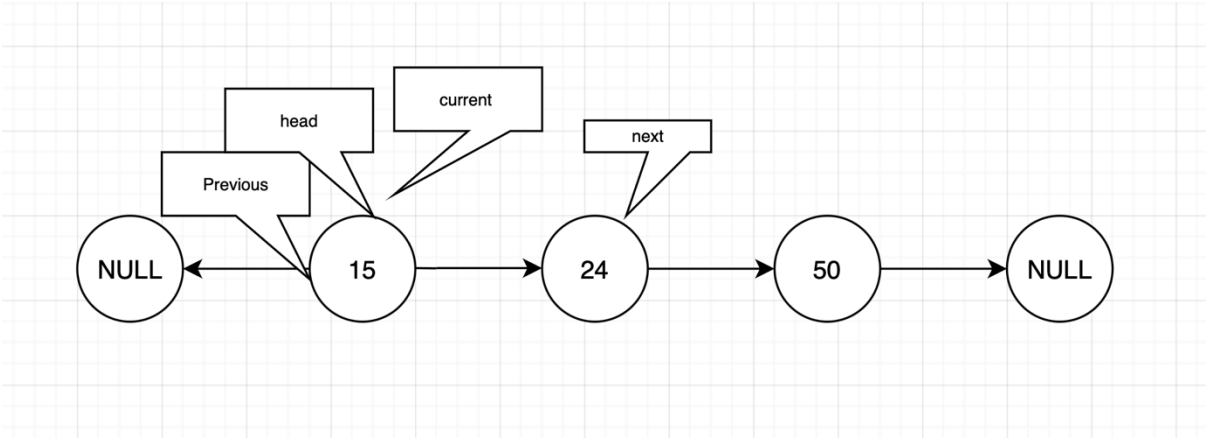
The idea for reversing a linked list is make each node point to a previous node and change the head node to the old last node of the list. To implement this, I use 3 pointers to keep track of nodes to update reverses links. Below is the demonstration of how it work.



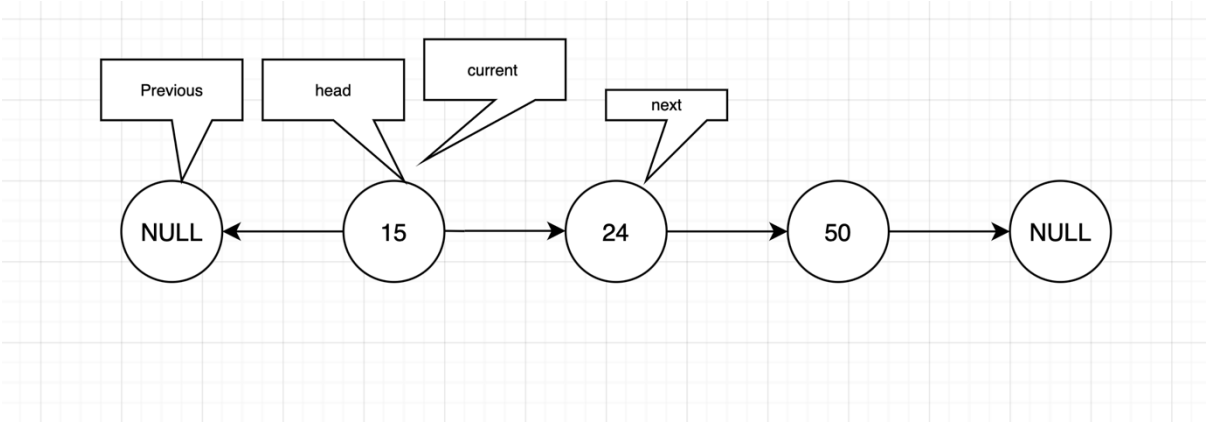
6.1 Start of the function



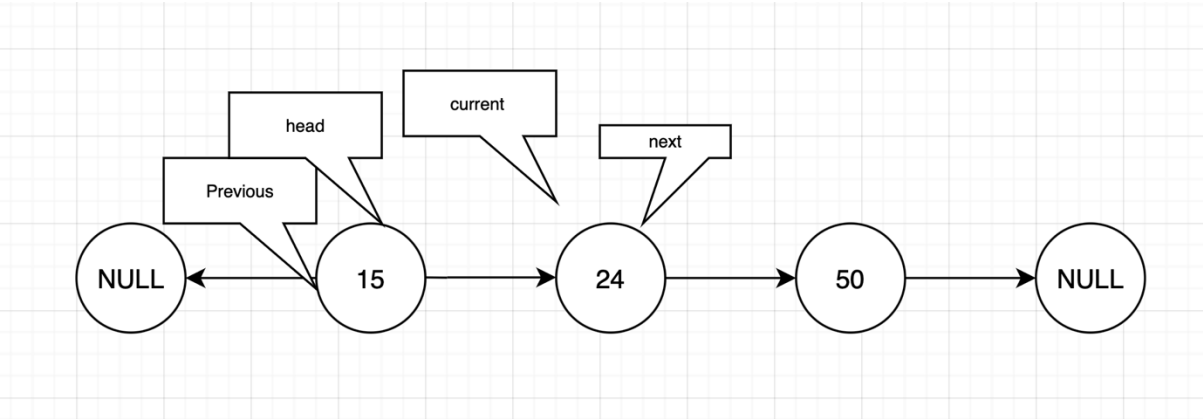
6.2 Next = current.next



6.3 Current.next = previous



6.4 Previous = current



6.5 Current = next

After finishing a step, by doing it several times until the last node change the pointed direction, then we will have a reversed list.

```
def reverse(self):
        itr = self.head
        prev = None
        while itr :
            temp = itr.next
            itr.next = prev
            prev = itr
            itr = temp
        self.head = prev
```

*Complexity analysis: by traversing over the whole list, the big O notation will be O(N) with N is the length of the list.

6.2 Recursive reversal

In this way, by using recursion, we will be every node then we will point it to the previous node which is pretended to the head node. After calling every single node at the end of the list, it will return the head node which is the variable named “second” holding the temp head roles, finally start to point to the previous node.

```
def reverse_recursive(head):
    if head is None:
        return None
    if head.next is None:
        return head
    second = head.next
    head.next = None
    reverse = reverse_recursive(second)
    second.next = head
    return reverse
```

*Complexity analysis: in this algorithm, the list still run through the length of linked list, therefore the big O notation is O(N)

7. Sorting algorithm with linked list

7.1 Merging 2 sorted linked list

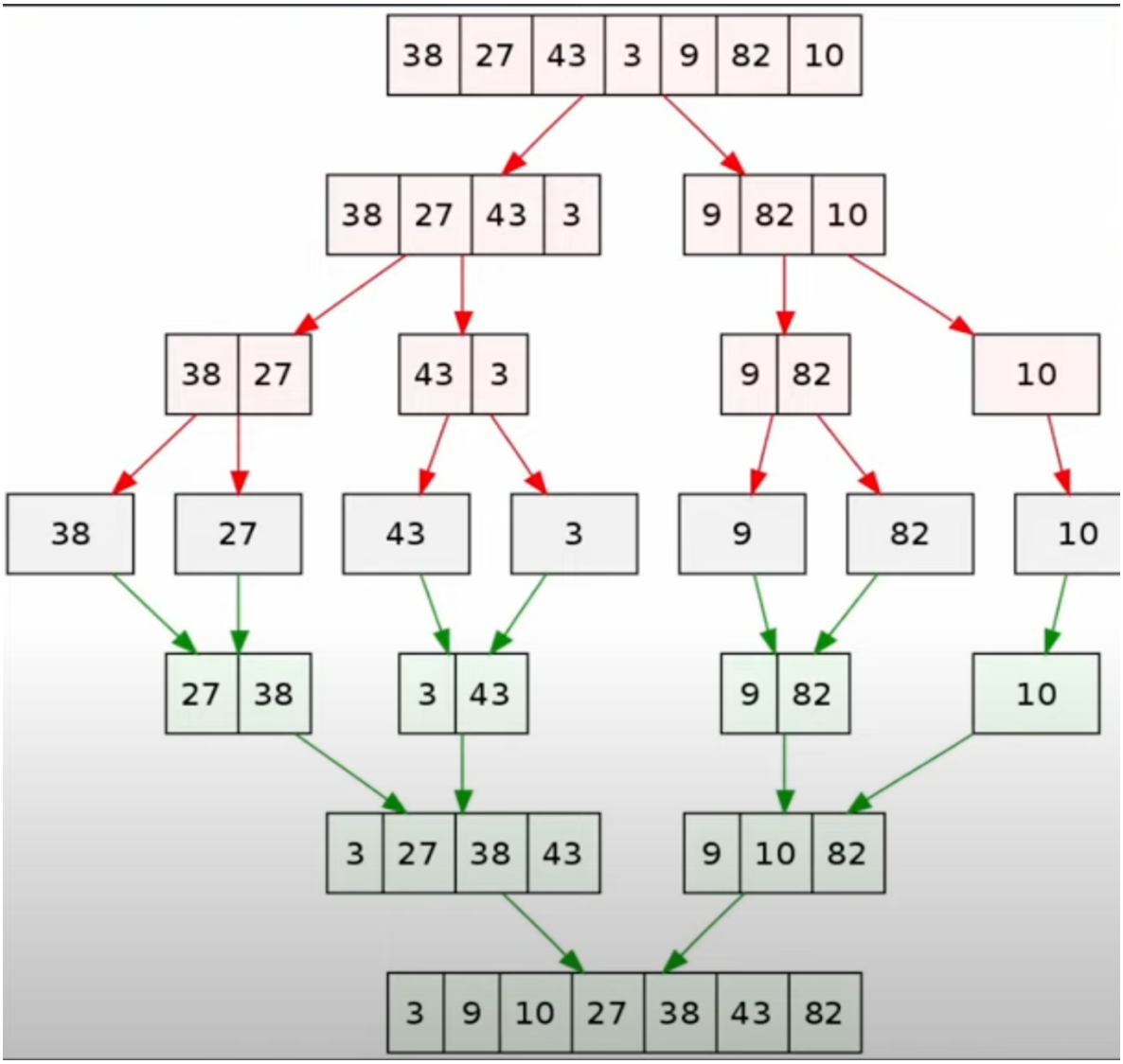
Continuing using recursion, I parallel call 2 linked list and compare each node's data in the list, if one of 2 data in 2 nodes higher, I will put it into a temp linked list, when it run through all of 2 list, the temp linked list will be returned.

```
def sortedMerge(self,ll1,ll2):
        if ll1 is None:
            return ll2
        if ll2 is None:
            return ll1
        if ll1.data < ll2.data:
            temp = ll1
            temp.next = self.sortedMerge(ll1.next,ll2)
        else:
            temp = ll2
            temp.next = self.sortedMerge(ll1,ll2.next)
        return temp
```

*Complexity analysis: by run through 2 linked lists, that means the complexity of this will be O(M) with M equal the length of list 1 plus list 2.

7.2 Merge sort algorithm

The basic logic of merge sort is continuous dividing half the list to become 2 parts left and right, then it will be merged with each other by the sortedMerge function to sort 2 smaller lists.



7.2.2 Merge sort logic

```
def mergeSort(self,head):
        if head is None or head.next is None:
            return head
        middle = self.get_middle(head)
        head2 = middle.next

        middle.next = None
        left = self.mergeSort(head)
        right = self.mergeSort(head2)
        final_list = self.sortedMerge(left,right)
        return final_list
```

*Complexity analysis: because a list is devided by 2 n times, that means a node is browsed through n times. Therefore, the number a node

is browsed is log2(N), and we have n node, that means the complexity will be O(NlogN)

8. Conclusion

8.1 Linked list application discussion

With multiple ways that we could interact with linked list, that means we also got numerous chances to utilize the application of linked list. One of the applications is using to implement stacks and queues, various representations of trees and graphs, dynamic memory allocation (linked list of free blocks), etc…. Also linked list has both side and the downside of linked list is because the use of pointers is more in linked lists hence, complex and requires more memory, searching an element is costly and requires O(n) time complexity.

8.2 Further research

In the future, I will get the specific insight in variation of linked lists which could be circular linked list, doubly linked list, double circular linked list, etc…. This approach could bring more application, for example, the alt tab in window is using the circular linked list, when people move to the final tab and continue, it will move to the first tab.

9. Reference

https://www.geeksforgeeks.org/what-is-linked-list/?ref=lbp
https://github.com/VietDungTran0412/Data-Structure-and-Algorithms