ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	Wednesday, 31 August, 2022	**Date Received 1st submission**	Wednesday, 31 August, 2022
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Hong Duc	**Student ID**	GCH200635
Class	GCH0907	**Assessor name**	Do Hong Quan

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	Duc

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

❐ Summative Feedback:		❐ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
IV Signature:		

Table of Contents

List of Figures

List of Table

I. Introduction

The following tasks' implementations and reviews are included in this article by the author.

(a) Describe how a single linked list and a doubly linked list differ from one another. Include the implementations that should contain the fundamental operations of these data structures, such as adding and deleting elements.

(b) How should an element be inserted into the middle of a linked list? Include his recommendation in the execution of his own linked list, as was done in task (a).

(c) Use a sorting algorithm on his linked list.

II. Implement complex data structures and algorithms

1. Singly liked list and doubly linked list.

The singly-linked list and the doubly-linked list are both applications of the linked list in which each singly-linked list item has some data and a connection to the subsequent element, allowing the structure to be preserved. In contrast, each node in a doubly-linked list is connected to the node before it (Sharma, 2019).

The key distinctions between a singly linked list and a doubly linked list are listed in the table below (Sharma, 2019).

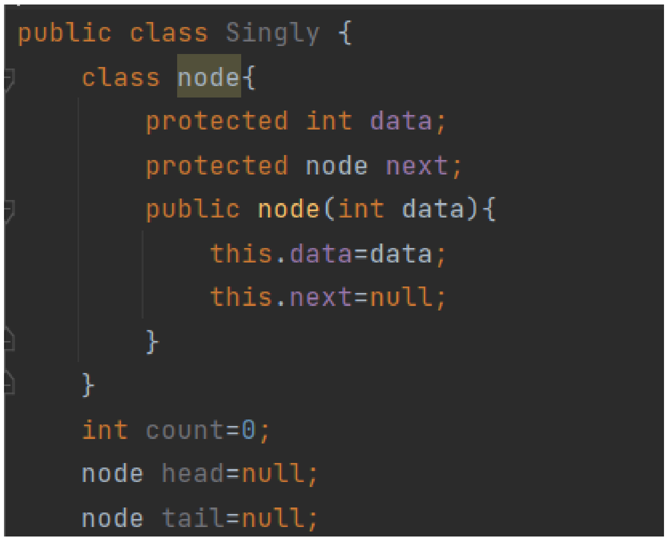
No	Key	Singly Linked List	Doubly Linked List
1	Complexity	The time complexity of adding a new element and deleting a new element is O(n).	The time complexity of adding a new element and deleting a new element is O(1).
2	Internal Implementation	Each node has a reference to the node after it as well as data.	It is more complicated since it contains data and a pointer to both the previous and subsequent nodes.
3	Usage	Used for implementation of stacks.	Binary trees, heaps, and stacks are all implemented using this.
4	Index performance	When memory is limited and searching is not required, a singly linked list is preferred. This is because a single index pointer is stored.	If memory is not a problem and we require better output while searching, we should utilize a doubly linked list.

5	Order of elements	Only one method of element traversal is permitted.	It allows two-dimension access.
6	Memory consumption	because only one node's reference is stored, it uses less memory (2 bytes).	Memory is used up more quickly since the preceding and subsequent nodes' pointers are stored. (4 bytes)

Table 1.Singly and Double Linked List

2. Singly Linked List

2.1 Attribute and constructor



```
public class Singly {
    class node{
        protected int data;
        protected node next;
        public node(int data){
            this.data=data;
            this.next=null;
        }
    }
    int count=0;
    node head=null;
    node tail=null;
```

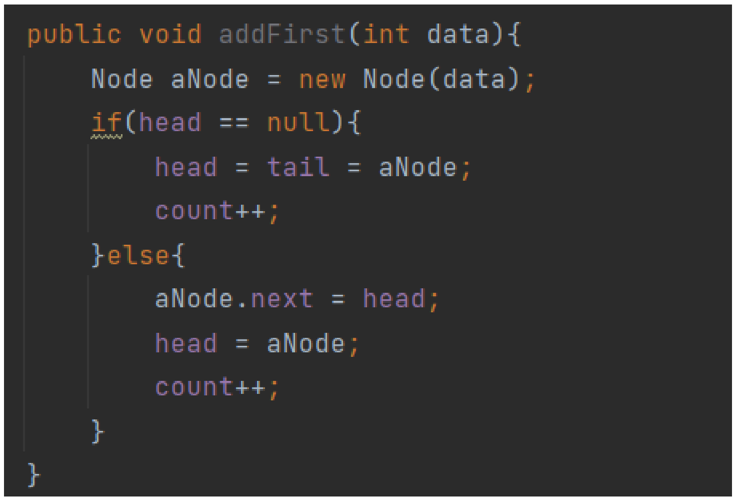
Figure 1.Attributes and constructor of Singly Linked List

The properties of a singly linked list are first populated, with the **data** attribute used to hold the nodes' contents and the pointer attribute used to refer to the **next** node. The constructor is **next**, with **data** as an argument.

Besides, the author also initializes an integer variable **count** to monitor the number of nodes in the **Singly Linked List**; **head** and **tail** is two variables to identify the first element and the last element(respectively) of the list.

2.2 Operations

2.2.1 Add new element to the first



```
public void addFirst(int data){
    Node aNode = new Node(data);
    if(head == null){
        head = tail = aNode;
        count++;
    }else{
        aNode.next = head;
        head = aNode;
        count++;
    }
}
```

Figure 2. Add operation

The operation above is used to add a new element to the first of the **Singly Linked List**. It has a return type is **void**, and it has 1 parameter that is the data of the element.

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: If there is no element in the list, assign head and tail are **aNode**.
- **Step 3**: If there is more than 1 element, assign the head to that **aNode**.
- **Step 4**: After adding, increase the size to 1.

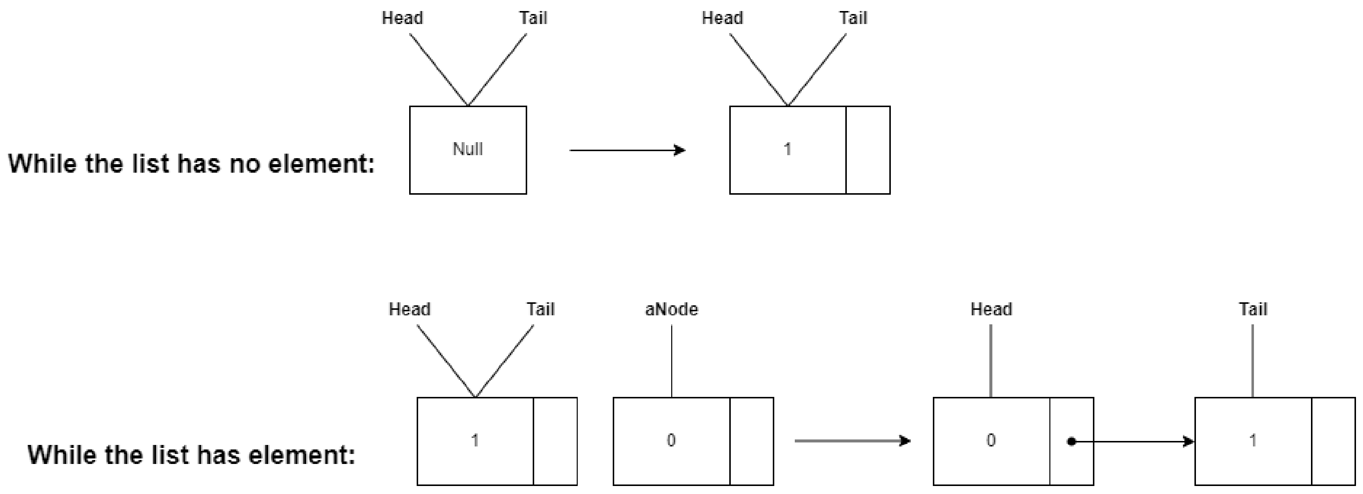
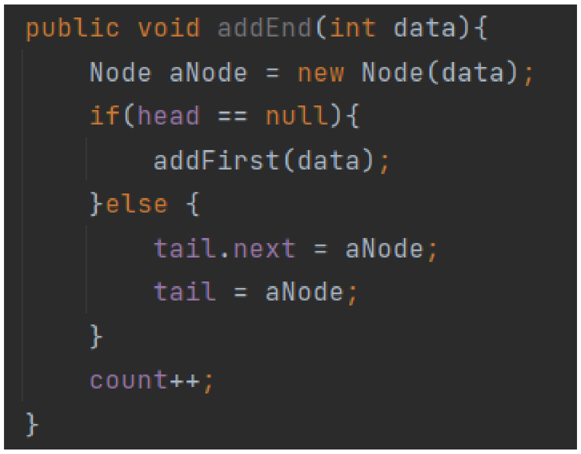


Figure 3. Illustration

2.2.2 Add new element to the end



```
public void addEnd(int data){
    Node aNode = new Node(data);
    if(head == null){
        addFirst(data);
    }else {
        tail.next = aNode;
        tail = aNode;
    }
    count++;
}
```

Figure 4. addFirst operation

The operation above is used to add a new element to the first of the **Singly Linked List**. It has a return type is **void**, and it has 1 parameter that is the data of the element.

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: If there is no element in the list, assign head and tail are **aNode** (addFirst operations).
- **Step 3**: If there is more than 1 element, assign the tail to that **aNode**.
- **Step 4**: After adding, increase the size to 1.

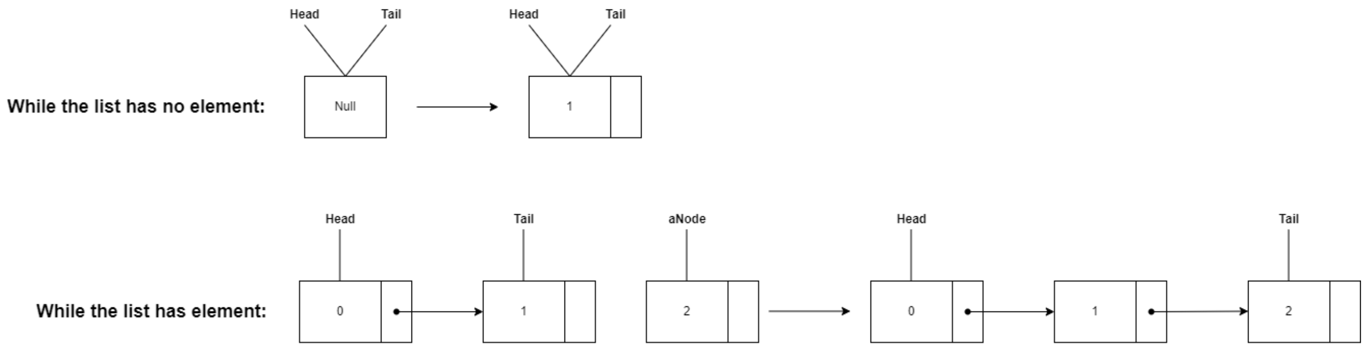
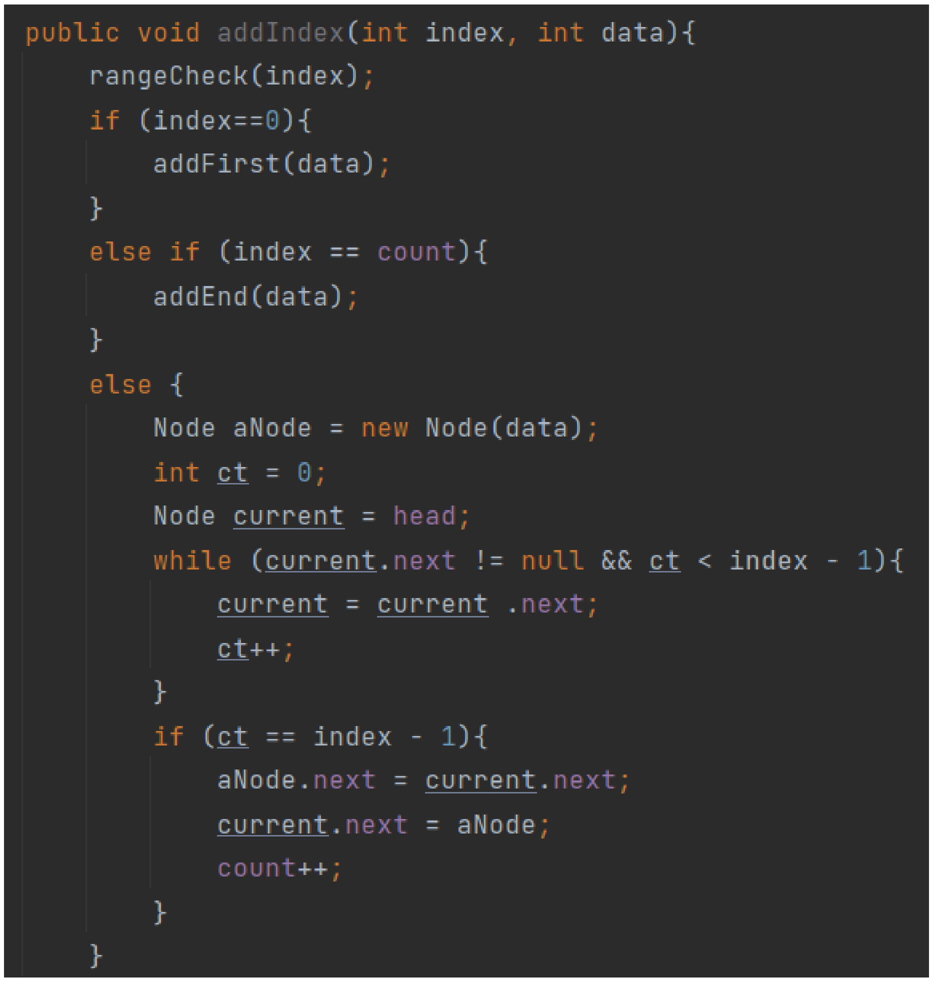


Figure 5. Illustration

2.2.3 Add new element at the given index



```java
public void addIndex(int index, int data){
    rangeCheck(index);
    if (index==0){
        addFirst(data);
    }
    else if (index == count){
        addEnd(data);
    }
    else {
        Node aNode = new Node(data);
        int ct = 0;
        Node current = head;
        while (current.next != null && ct < index - 1){
            current = current .next;
            ct++;
        }
        if (ct == index - 1){
            aNode.next = current.next;
            current.next = aNode;
            count++;
        }
    }
```

Figure 6. addIndex operation

The operation above is used to add a new element to the given index of the **Singly Linked List**. It has a return type is **void**, and it has 2 parameters are the index and the data of the element.

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: Create the variable **ct** to track the index.
- **Step 3**: Create the variable **ct** to track the element at the given index.
- **Step 4**: Run from the head to the index -1 with the condition are current.next!=null and ct<index-1.
- **Step 5**: When ct == index – 1, insert the new element into that index.
- **Step 6**: After adding, increase the size to 1.

The **addFirst** operation will be called with the argument is **data** if the index is equal to 0. If the index equals the count, the **add** operation will be called with the data. rangeCheck(index) is a function that determines whether the provided index is valid. Throw an error if it is invalid. The author will discuss this procedure in more detail in the following parts to help you understand it better.

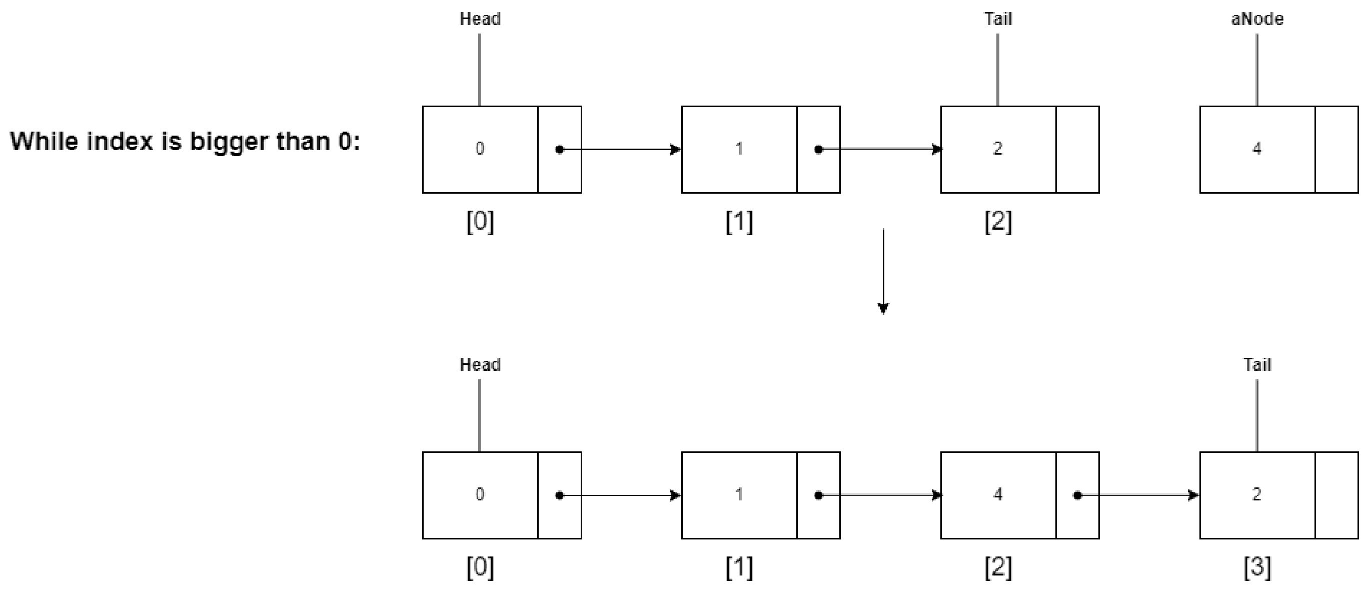
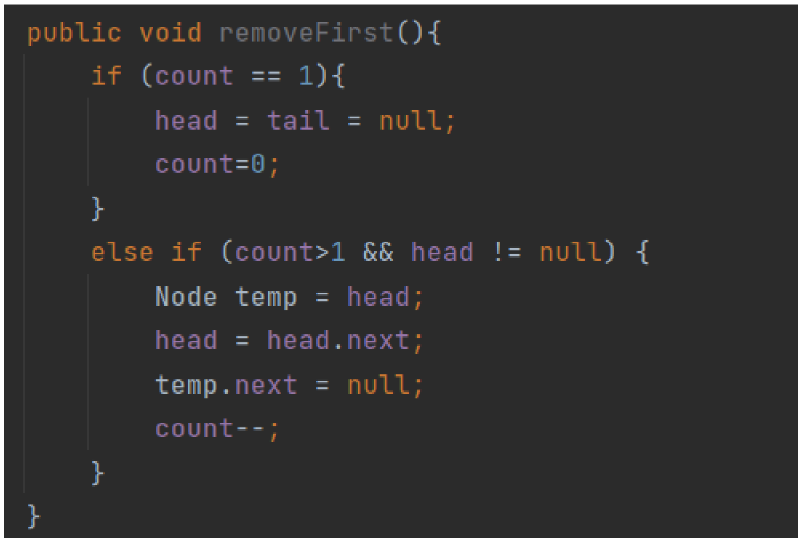


Figure 7. Illustration

2.2.4 Remove the first elements of the list



```
public void removeFirst(){
    if (count == 1){
        head = tail = null;
        count=0;
    }
    else if (count>1 && head != null) {
        Node temp = head;
        head = head.next;
        temp.next = null;
        count--;
    }
}
```

Figure 8. removeFirst operation

void removeFirst() is an operation to remove the first element of the list. Its return type is void, and it takes no parameter. In this operation, it will check the list has element or not, by checking **head! =null && count>1**, if it is true, then it will delete the first node and the count (the size of the list) is decreased by 1.

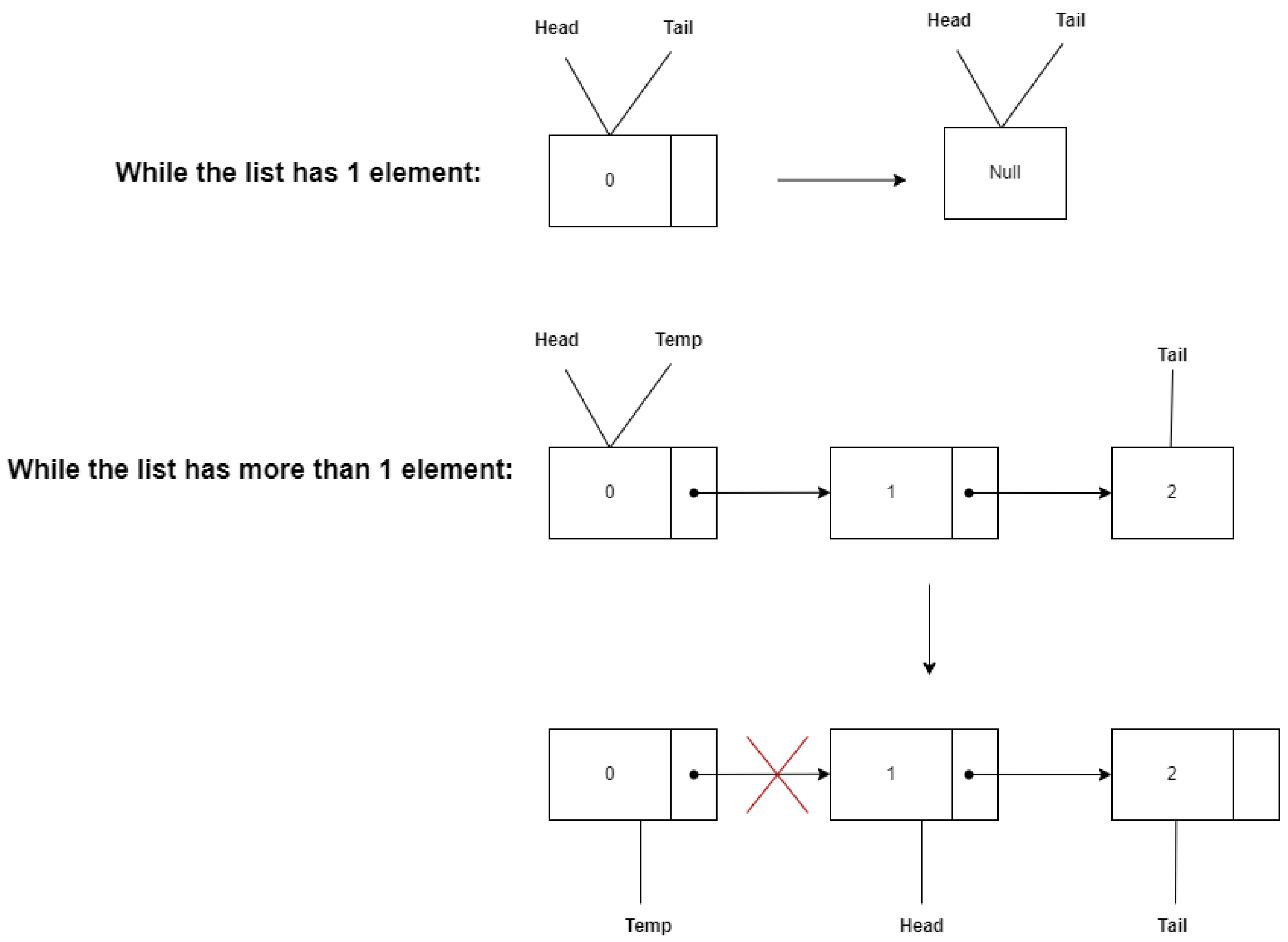
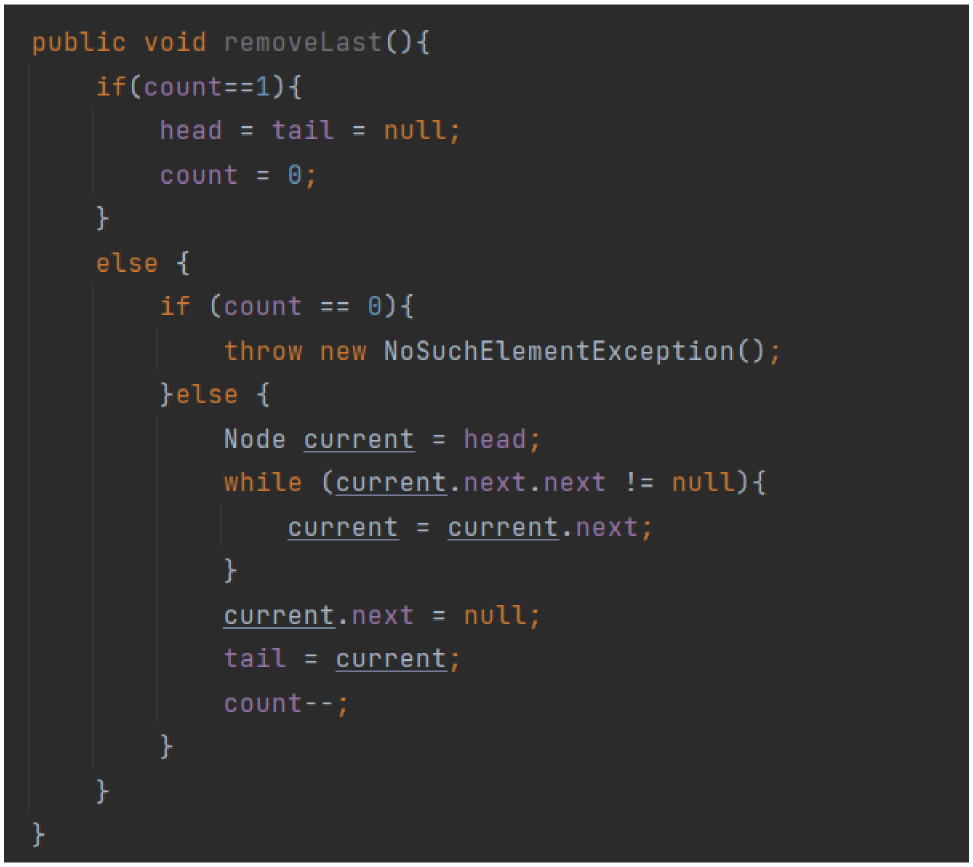

While the list has 1 element:
Head
Tail
0
Head
Tail
Null
While the list has more than 1 element:
Head
Temp
0
1
2
Tail
0
1
2
Temp
Head
Tail

Figure 9. Illustration

2.2.5 Remove the last elements of the list



```
public void removeLast(){
    if(count==1){
        head = tail = null;
        count = 0;
    }
    else {
        if (count == 0){
            throw new NoSuchElementException();
        }else {
            Node current = head;
            while (current.next.next != null){
                current = current.next;
            }
            current.next = null;
            tail = current;
            count--;
        }
    }
}
```

Figure 10. removeLast operation

void removeLast() is an operation removes the last element of the list, it has return type is void, and no parameter. It is similar with **removeFirst**, when the list has 1 element, **head** and **tail** will be equal to null and the count will be zero. If the list has no element, it will throw an error: **NoSuchElementException().** Otherwise, it will run from the **head** element to the element before the last element (right before **tail**) to remove the **tail** element.

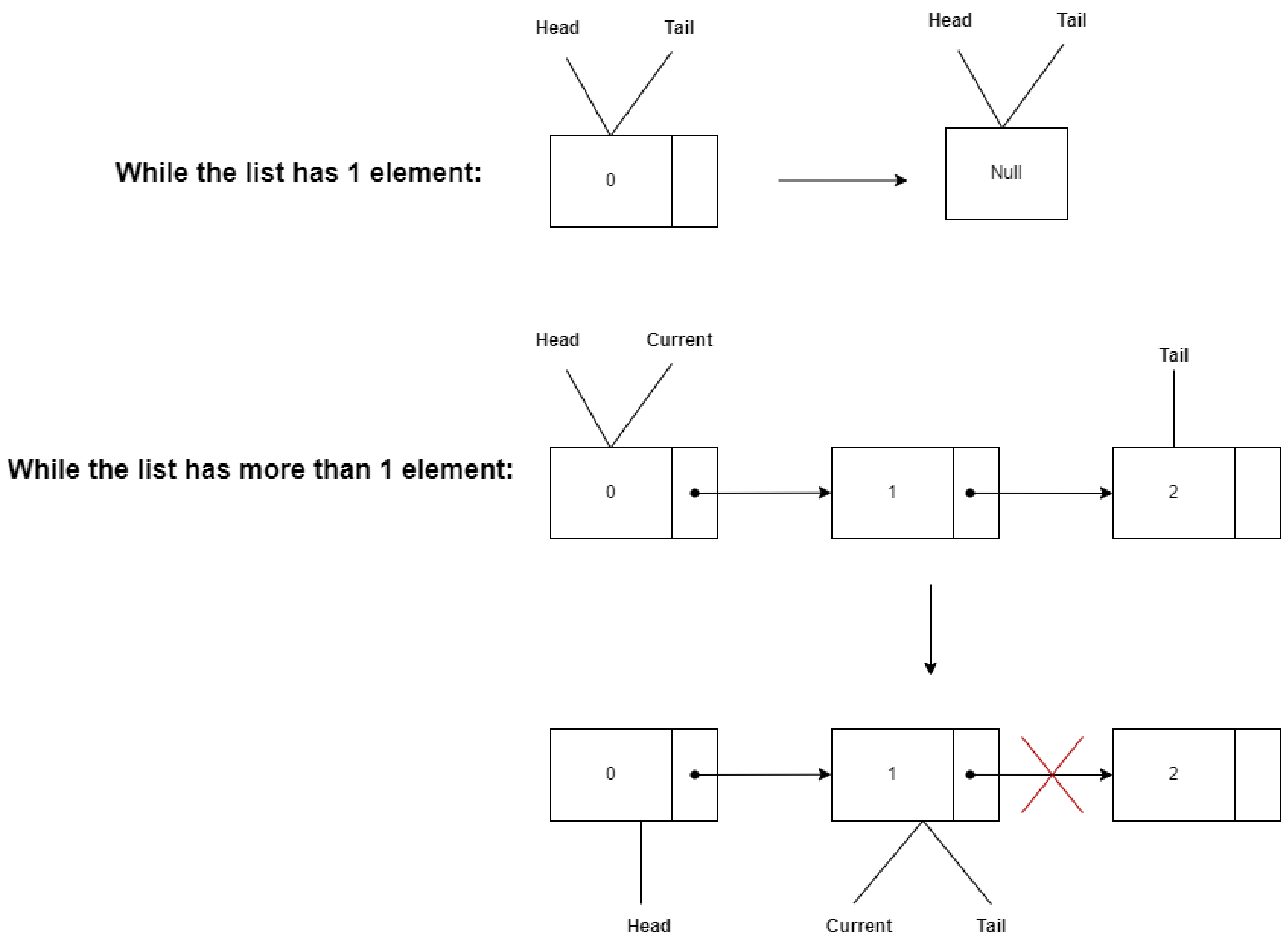
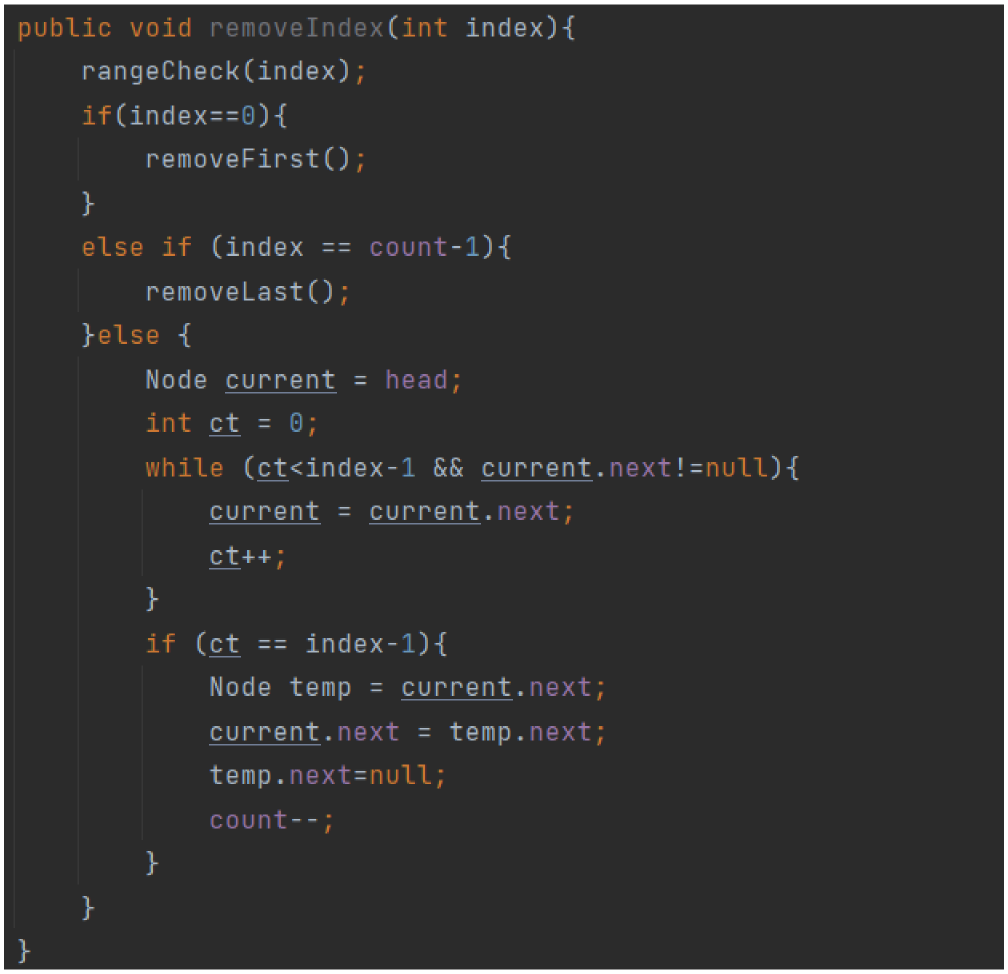

While the list has 1 element:
Head
Tail
0
Head
Tail
Null
While the list has more than 1 element:
Head
Current
Tail
0
1
2
0
1
2
Head
Current
Tail

Figure 11. Illustration

2.2.6 Remove the element by the given index



```
public void removeIndex(int index){
    rangeCheck(index);
    if(index==0){
        removeFirst();
    }
    else if (index == count-1){
        removeLast();
    }else {
        Node current = head;
        int ct = 0;
        while (ct<index-1 && current.next!=null){
            current = current.next;
            ct++;
        }
        if (ct == index-1){
            Node temp = current.next;
            current.next = temp.next;
            temp.next=null;
            count--;
        }
    }
}
```

Figure 12. removeIndex operation

This operation is used to remove an element at the given index. It has the return type is void, and it takes 1 parameter: **index**. In this operation, **rangeCheck(index)** is also being used like in addIndex operation, as it needs to check the index is valid or not. While the index is valid, it dives into solving problem. If the index == 0, it calls the **removeFirst()** operation, or if the index == count-1, which means it is pointing the last element, it calls the **removeLast()** operation. Otherwise, it runs from the head element to the element right before the selected element and remove the selected element.

While the index is valid:

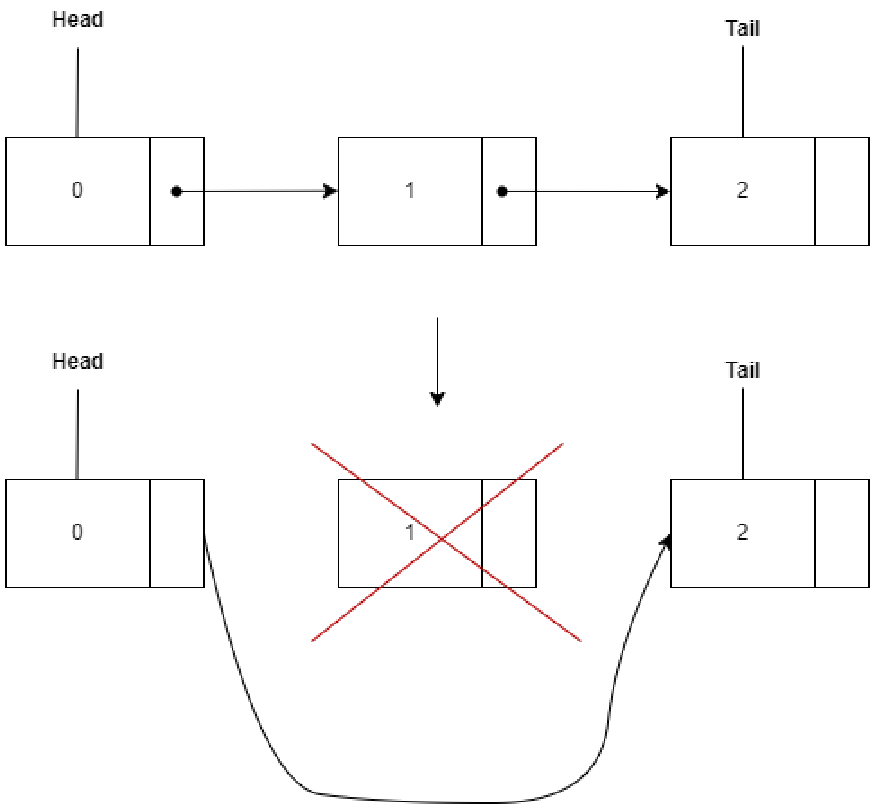
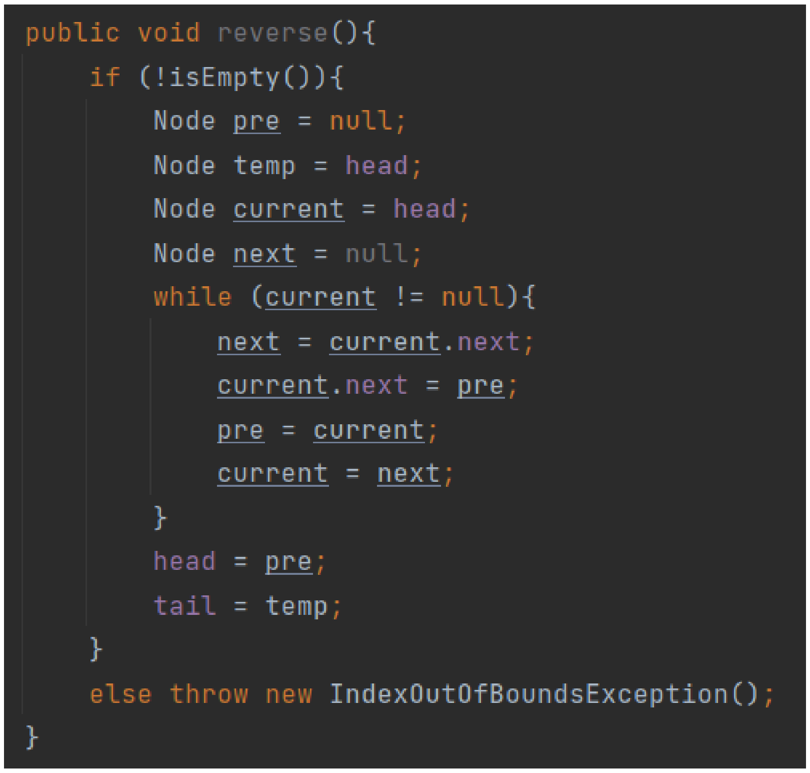


Figure 13. Illustration

2.2.7 Reverse a Singly Linked List



```
public void reverse(){
    if (!isEmpty()){
        Node pre = null;
        Node temp = head;
        Node current = head;
        Node next = null;
        while (current != null){
            next = current.next;
            current.next = pre;
            pre = current;
            current = next;
        }
        head = pre;
        tail = temp;
    }
    else throw new IndexOutOfBoundsException();
}
```

Figure 14. reverse operation

In this operation, if the list is not empty **(!isEmpty()),** it will reverse the list. The below picture is how the operation works:

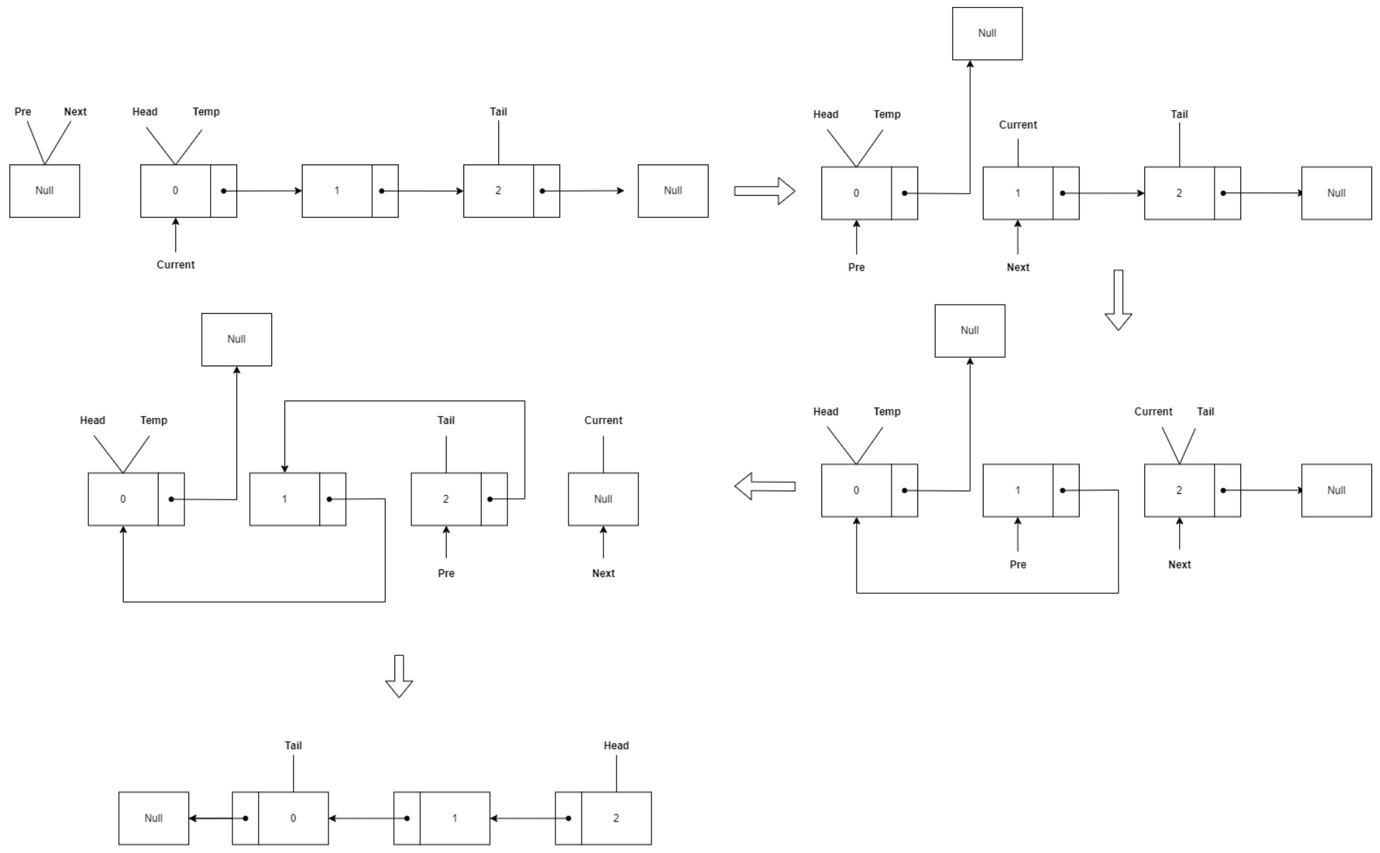


Figure 15. Illustration

2.2.8 Print the list in reverse order

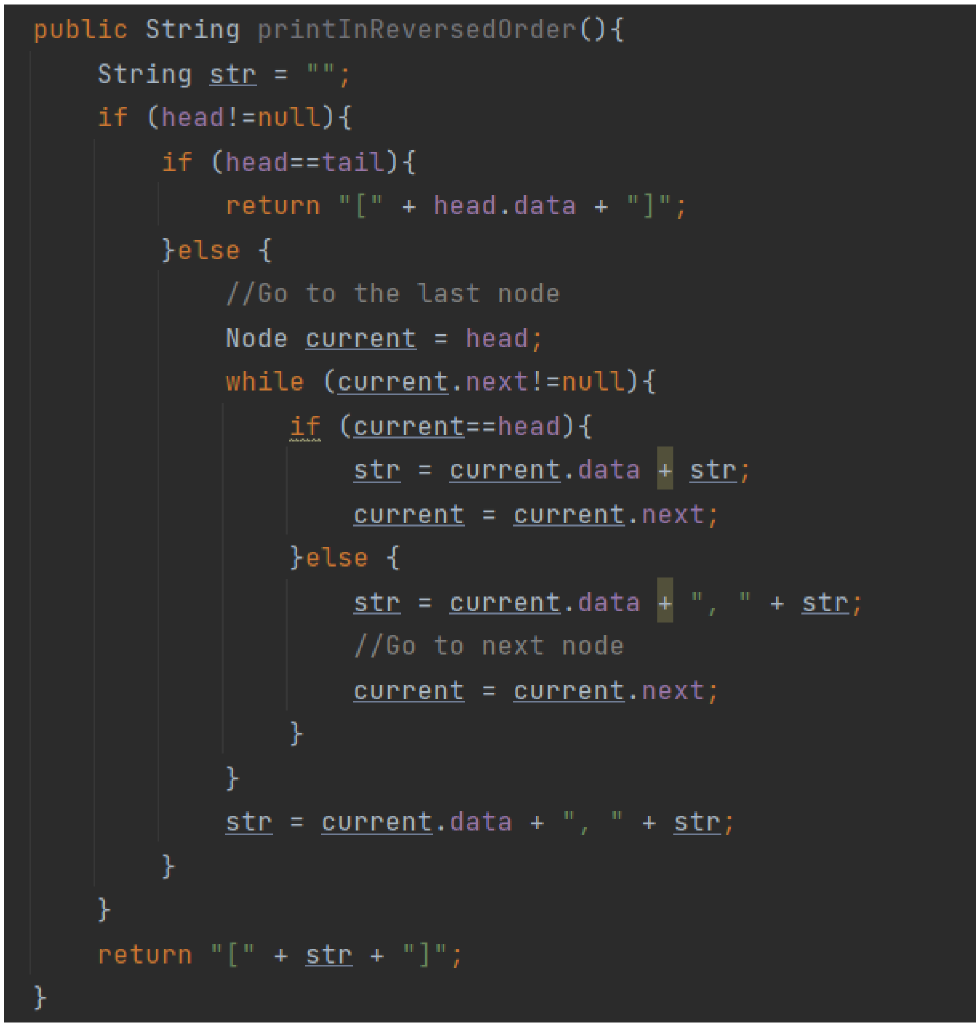

```
public String printInReversedOrder(){
    String str = "";
    if (head!=null){
        if (head==tail){
            return "[" + head.data + "]";
        }else {
            //Go to the last node
            Node current = head;
            while (current.next!=null){
                if (current==head){
                    str = current.data + str;
                    current = current.next;
                }else {
                    str = current.data + ", " + str;
                    //Go to next node
                    current = current.next;
                }
            }
            str = current.data + ", " + str;
        }
    }
    return "[" + str + "]";
}
```

Figure 16. printInReverseOrder operation

This function is simple printing the **Singly Linked List** in reverse order. Its return type is String, and it takes no parameter. The logic of this operation is, it stores the data of each node, from the begin to the end. And at each step, the result will be concatenated by the current’s data with a “,” and the previous result. However, if the current is **head** then the result simply be concatenated by the **current’s data** and the previous result. To be more understandable, the result will be packaged in “[]”.

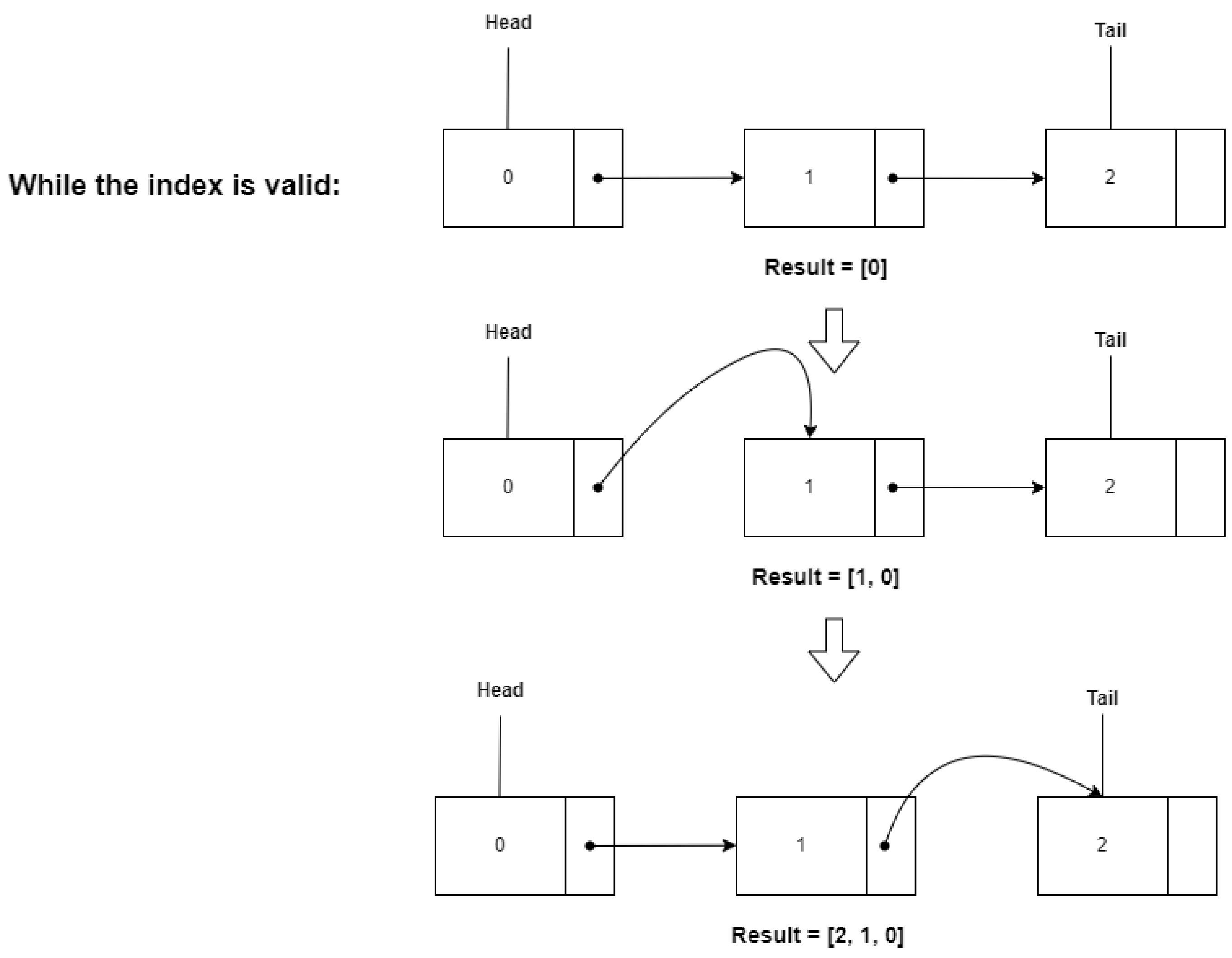
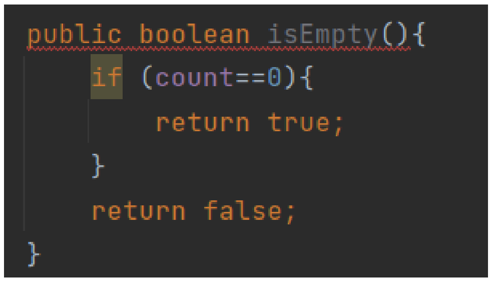


Figure 17. Illustration

2.2.9. Empty

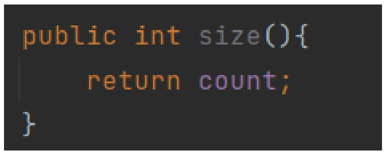


```
public boolean isEmpty(){
    if (count==0){
        return true;
    }
    return false;
}
```

Figure 18. isEmpty operation

The above operation is used to check the list has element or not. If it has element, it returns **false**. Otherwise, it returns **true**.

2.2.10. Size



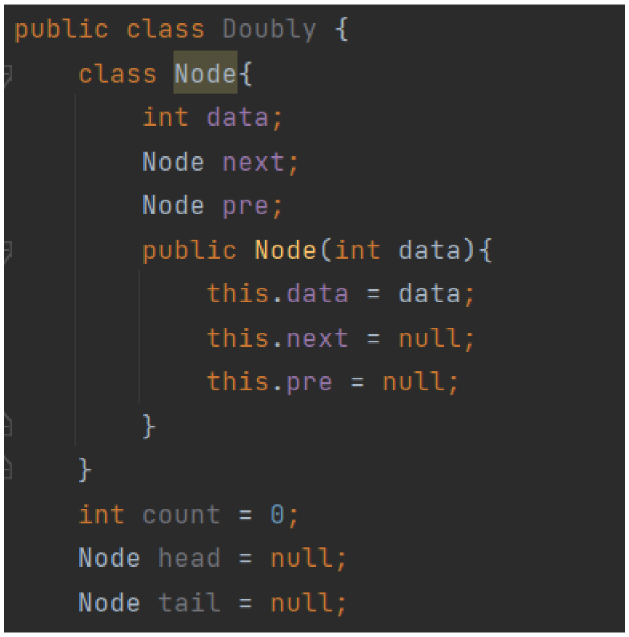
```java
public int size(){
    return count;
}
```

Figure 19. size operation

The above operation shows the size of the list (the total of elements in the list). Its return type is Integer.

3. Doubly Linked List

3.1 Attribute and constructor



```java
public class Doubly {
    class Node{
        int data;
        Node next;
        Node pre;
        public Node(int data){
            this.data = data;
            this.next = null;
            this.pre = null;
        }
    }
    int count = 0;
    Node head = null;
    Node tail = null;
```

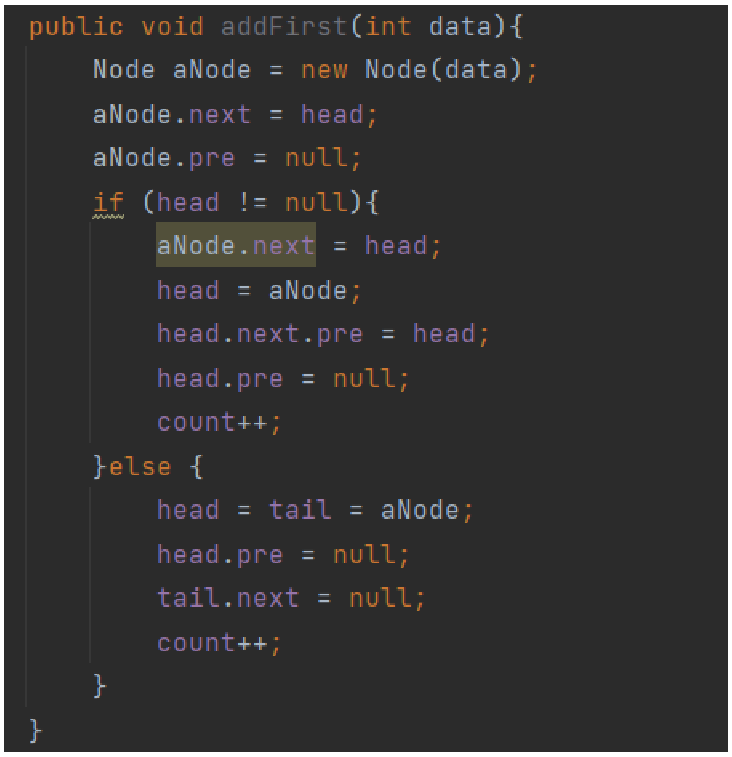
Figure 20. Attributes and constructors of Doubly Linked List

The attributes of **Doubly Linked List** will be initialized at the beginning, while the data attribute is used to store the data of the nodes, **next** is the pointer to point the next Node, and **prev** is the pointer to point the previous node. Then, it is the constructor, with an argument data.

Besides, the author also initializes an integer variable count to monitor the number of nodes in the **Doubly Linked List**; **head** and **tail** is two variables to identify the first element and the last element (respectively) of the list.

3.2 Operations

3.2.1 Add new element to the first



```
public void addFirst(int data){
    Node aNode = new Node(data);
    aNode.next = head;
    aNode.pre = null;
    if (head != null){
        aNode.next = head;
        head = aNode;
        head.next.pre = head;
        head.pre = null;
        count++;
    }else {
        head = tail = aNode;
        head.pre = null;
        tail.next = null;
        count++;
    }
}
```

Figure 21. addFirst operation

The operation above is used to add a new element to the first of the **Doubly Linked List**. It has a return type is **void**, and it has 1 parameter that is the data of the element.

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: If there is no element in the list, assign head and tail are **aNode**.
- **Step 3**: If there is more than 1 element, assign the head to that **aNode**.
- **Step 4**: After adding, increase the size to 1.

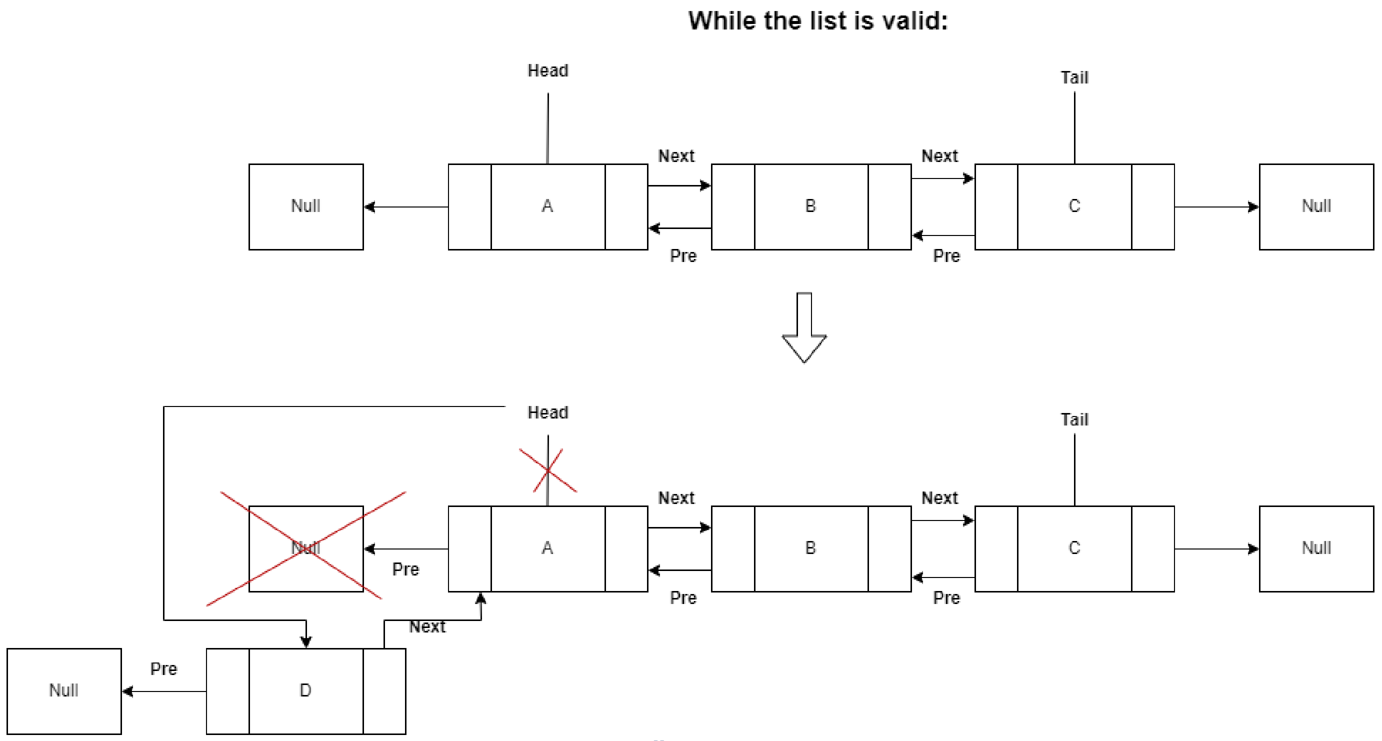
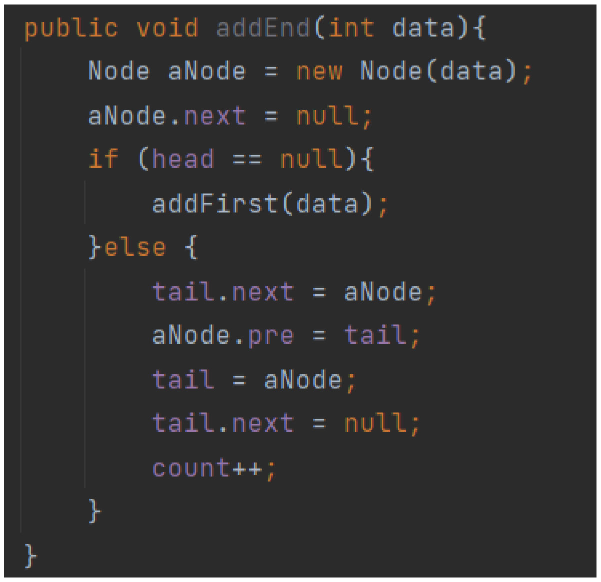


Figure 22. Illustration

3.2.2 Add new element to the end



```
public void addEnd(int data){
    Node aNode = new Node(data);
    aNode.next = null;
    if (head == null){
        addFirst(data);
    }else {
        tail.next = aNode;
        aNode.pre = tail;
        tail = aNode;
        tail.next = null;
        count++;
    }
}
```

Figure 23. add operation

The operation above is used to add a new element to the first of the **Doubly Linked List**. It has a return type is **void**, and it has 1 parameter that is the data of the element.

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: If there is no element in the list, assign **head** and **tail** are **aNode** (addFirst operations).
- **Step 3**: If there is more than 1 element, assign the **tail** to that **aNode**.
- **Step 4**: After adding, increase the size to 1.

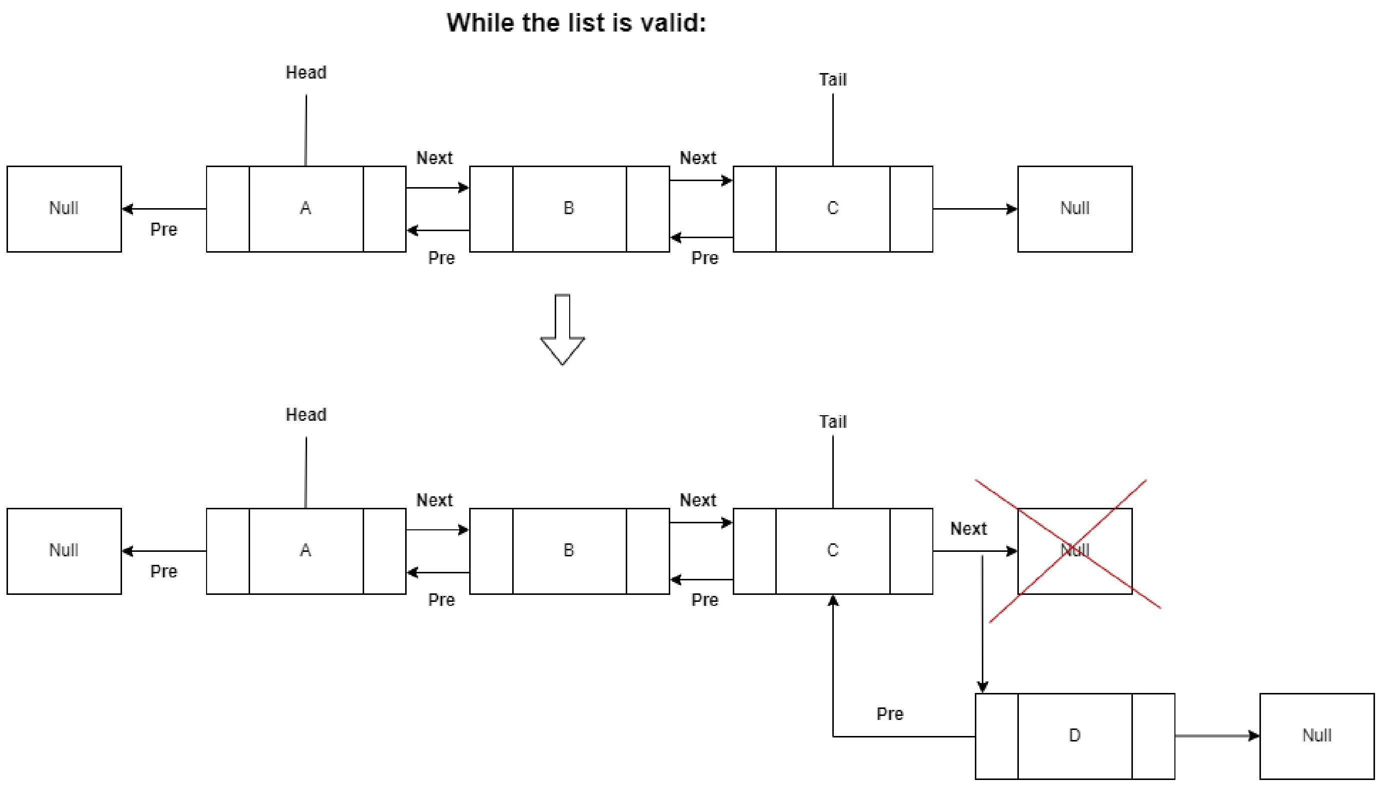
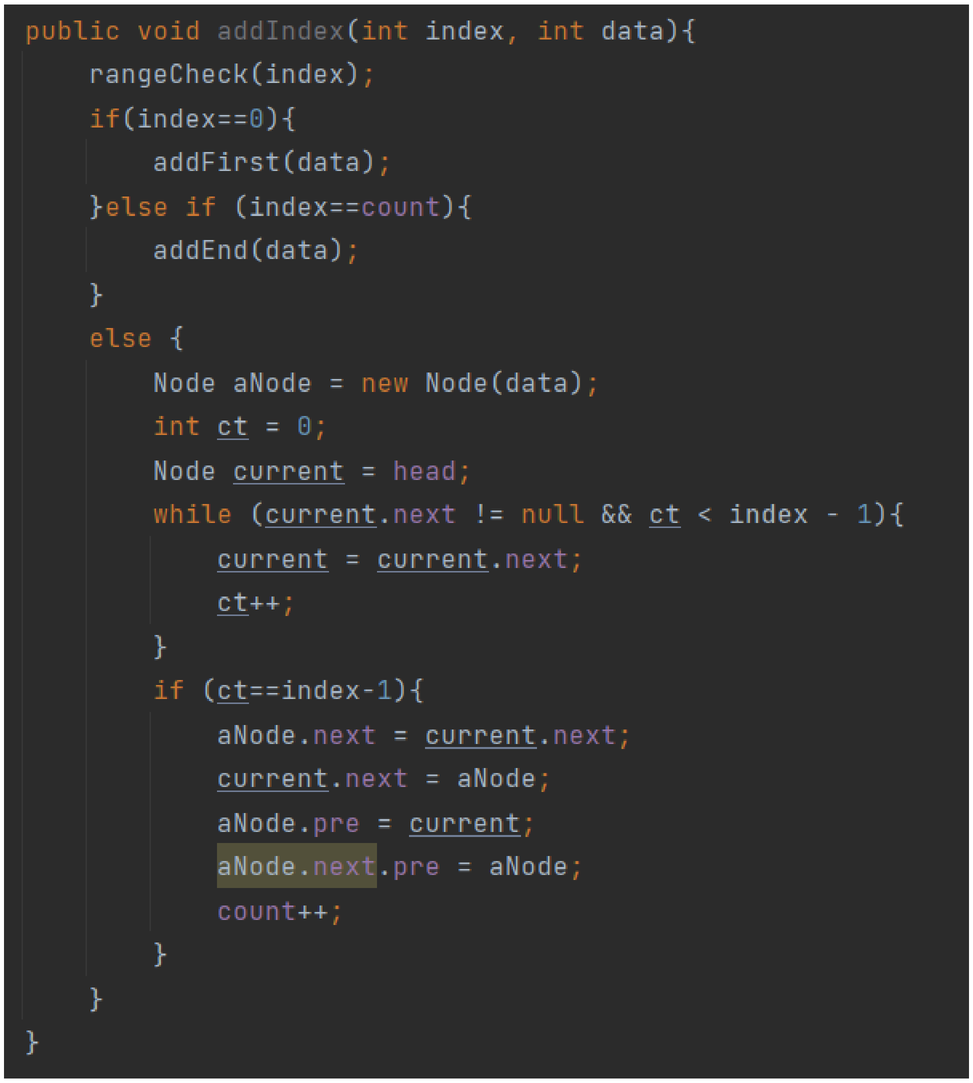


Figure 24. Illustration

3.2.3 Add new element at the given index



```
public void addIndex(int index, int data){
    rangeCheck(index);
    if(index==0){
        addFirst(data);
    }else if (index==count){
        addEnd(data);
    }
    else {
        Node aNode = new Node(data);
        int ct = 0;
        Node current = head;
        while (current.next != null && ct < index - 1){
            current = current.next;
            ct++;
        }
        if (ct==index-1){
            aNode.next = current.next;
            current.next = aNode;
            aNode.pre = current;
            aNode.next.pre = aNode;
            count++;
        }
    }
}
```

Figure 25. addIndex operation

The operation above is used to add a new element to the first of the **Doubly Linked List**. It has a return type is **void**, and it has 2 parameters are the index and the data of the element.
In this operation, if the index is valid, then the following step:

- **Step 1**: Create a new node **aNode** to get the data of the added element.
- **Step 2**: Create the variable **ct** to track the index.
- **Step 3**: Create the variable **current** to track the element at the given index.
- **Step 4**: Run from the head to the index -1 with the condition are current.next!=null and cnt<index-1
- **Step 5**: When ct == index – 1, insert the new element into that index.
- **Step 6**: After adding, increase the size to 1.

If the index is equal to 0, then it will call the **addFirst** operation with the parameter is **data**. In case of the index is equal to count, it will call the add operation with the parameter is **data**. **rangeCheck(index)** is an operation to check the given index is valid or not. If it is not valid, throw an error. It is exactly the same with void **rangeCheck(index)** at **Singly Linked List** as above.

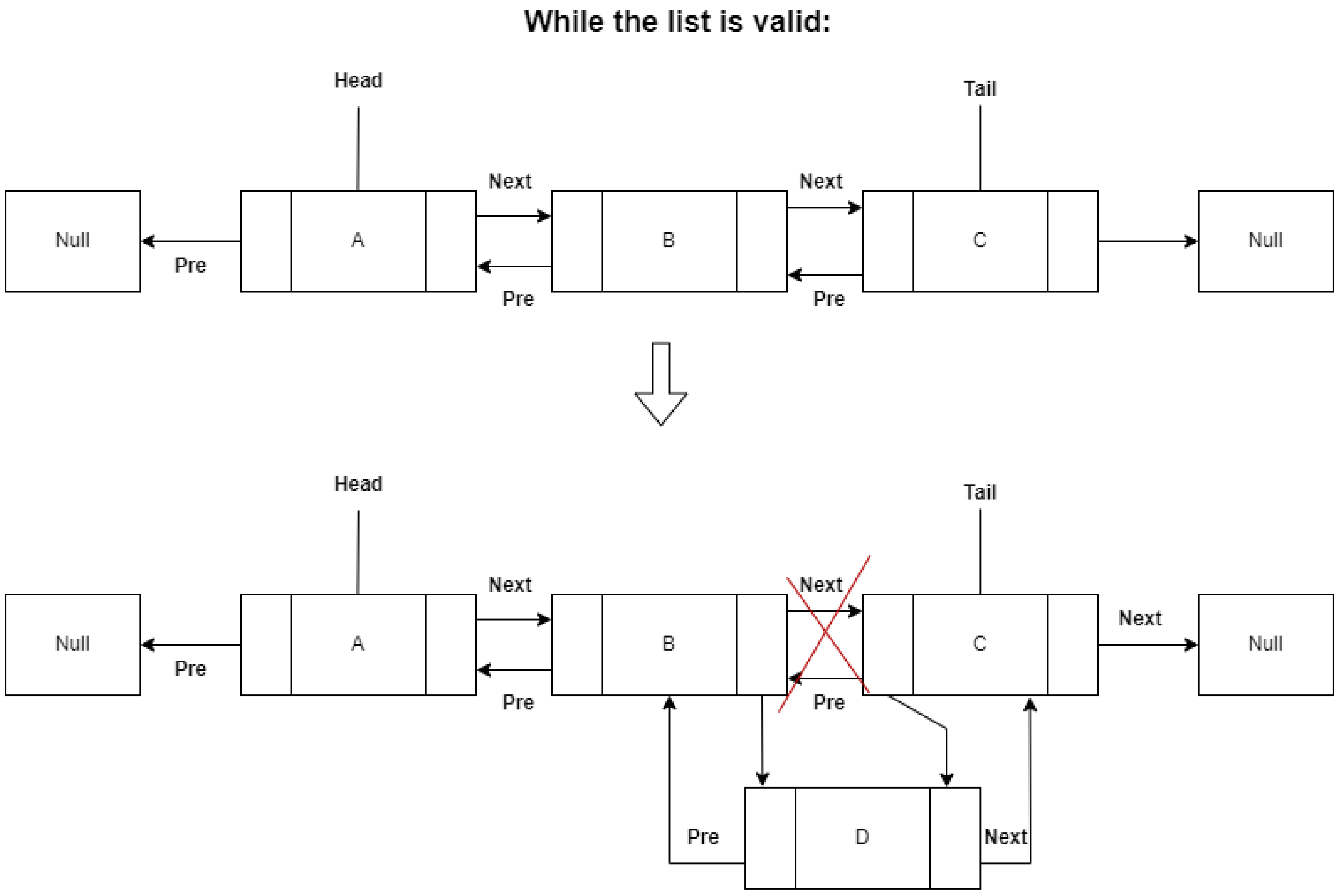
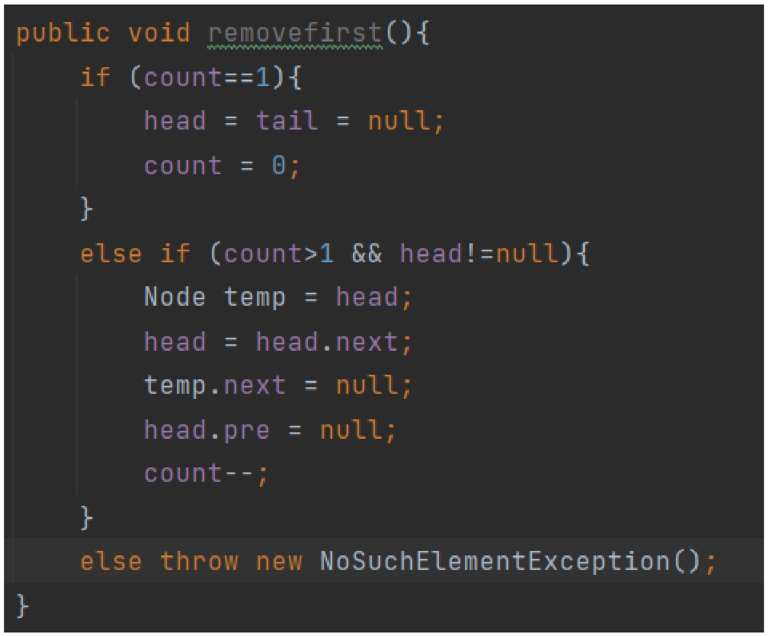


Figure 26. Illustration

3.2.4 Remove the first element of the list



```
public void removefirst(){
    if (count==1){
        head = tail = null;
        count = 0;
    }
    else if (count>1 && head!=null){
        Node temp = head;
        head = head.next;
        temp.next = null;
        head.pre = null;
        count--;
    }
    else throw new NoSuchElementException();
}
```

Figure 27. removeFirst operation

void **removeFirst()** is an operation to remove the first element of the list. Its return type is void, and it takes no parameter. In this operation, it will check the list has element or not, by checking **head! =null && count>1**, if it is true, then it will delete the first node and the count (the size of the list) is decreased by 1. Otherwise, it will throw an error: **NoSuchElementException()**, which means the element being requested does not exit. While count==1, which means the list has only 1 element, the **head** and **tail** will be equal to null and the **count** will be zero.

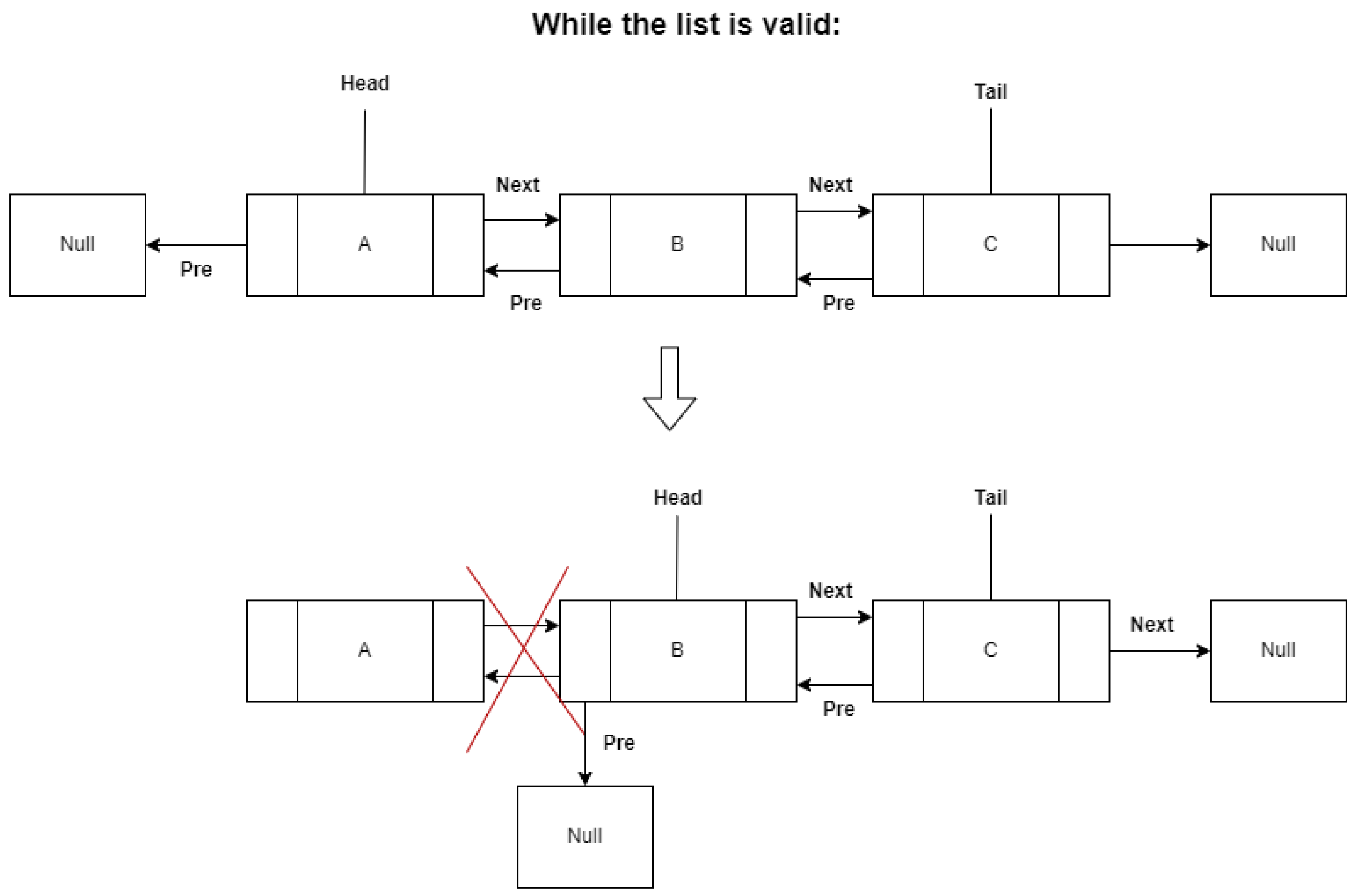
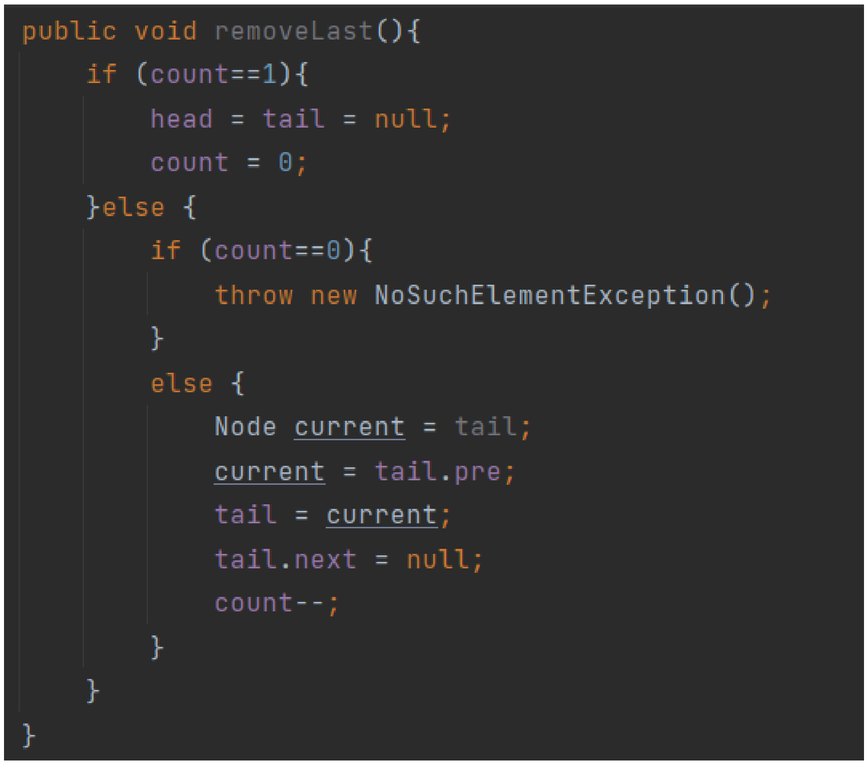


Figure 28. Illustration

3.2.5 Remove the last element of the list



```java
public void removeLast(){
    if (count==1){
        head = tail = null;
        count = 0;
    }else {
        if (count==0){
            throw new NoSuchElementException();
        }
        else {
            Node current = tail;
            current = tail.pre;
            tail = current;
            tail.next = null;
            count--;
        }
    }
}
```

Figure 29. removeLast operation

void removeLast() is an operation removes the last element of the list, it has return type is void, and no parameter. It is similar with **removeFirst**, when the list has 1 element, **head** and **tail** will be equal to null and the **count** will be zero. If the list has no element, it will throw an error**: NoSuchElementException().** This operation does not need to run from the beginning of the list like **Singly Linked List**, because the author leveraged the **tail** and the **prev** of **Doubly Linked List.**

So, it could go from the end to the previous one and remove the last element like the figure below:

While the list is valid:

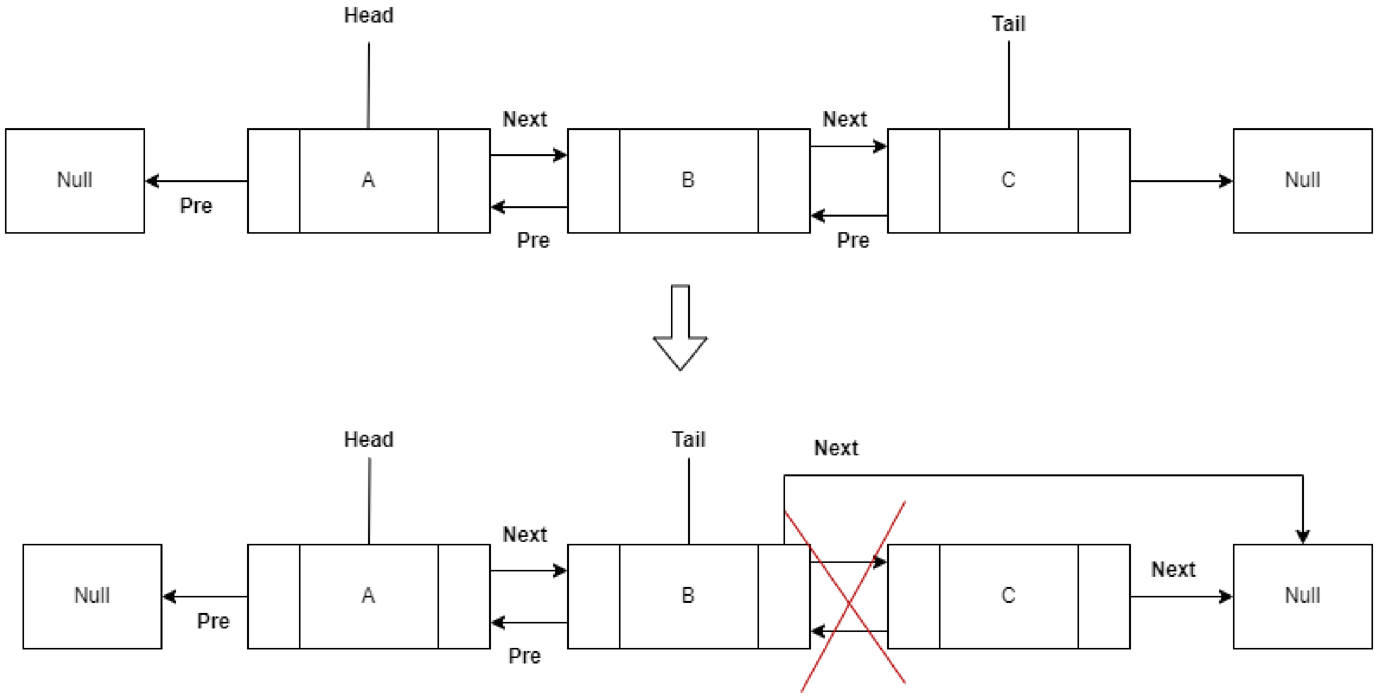
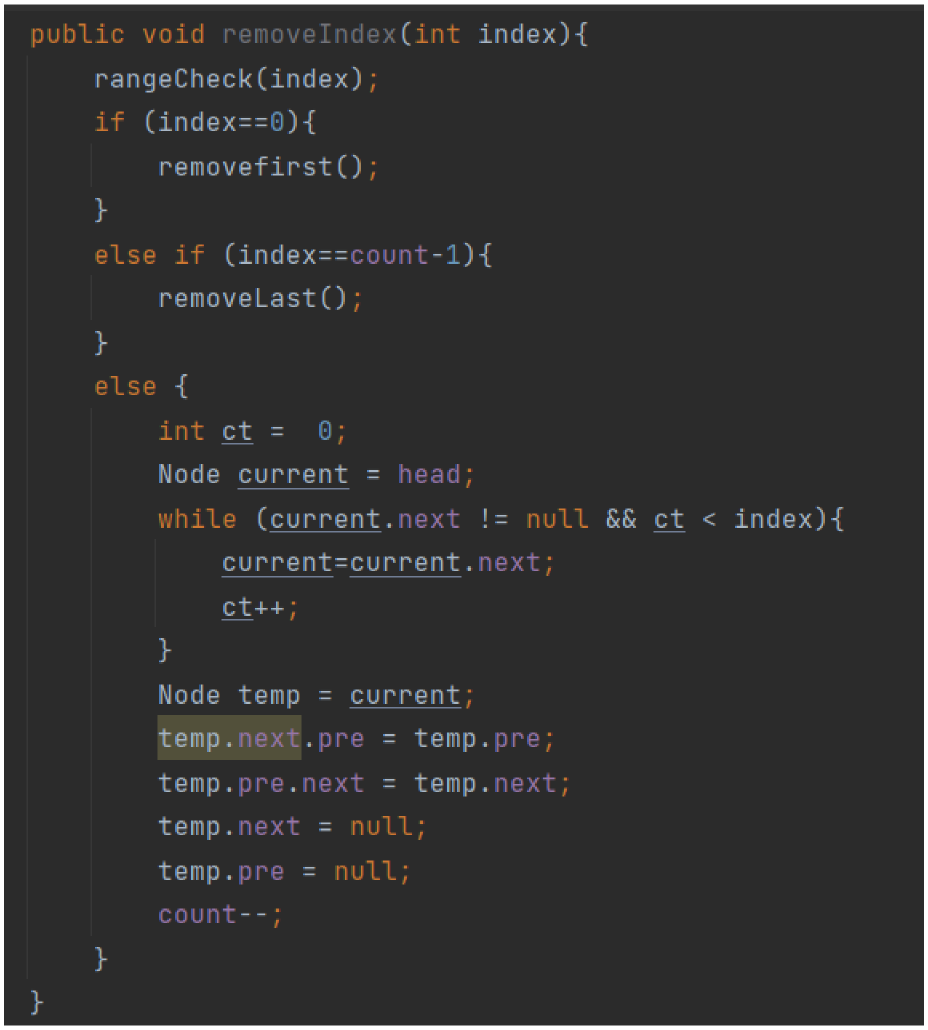


Figure 30. Illustration

3.2.6 Remove the element by the given index



```
public void removeIndex(int index){
    rangeCheck(index);
    if (index==0){
        removefirst();
    }
    else if (index==count-1){
        removeLast();
    }
    else {
        int ct =  0;
        Node current = head;
        while (current.next != null && ct < index){
            current=current.next;
            ct++;
        }
        Node temp = current;
        temp.next.pre = temp.pre;
        temp.pre.next = temp.next;
        temp.next = null;
        temp.pre = null;
        count--;
    }
}
```

Figure 31. removeIndex operation

This operation is used to remove an element at the given index. It has the return type is void, and it takes 1 parameter: **index**. In this operation, **rangeCheck(index)** is also being used like in **addIndex** operation, as it needs to check the index is valid or not. While the index is valid, it dives into solving problem. If the index == 0, it calls the **removeFirst()** operation, or if the index == count-1, which means it is pointing the last element, it calls the **removeLast()** operation. Otherwise, it runs from the head element to the selected element and remove the selected element.

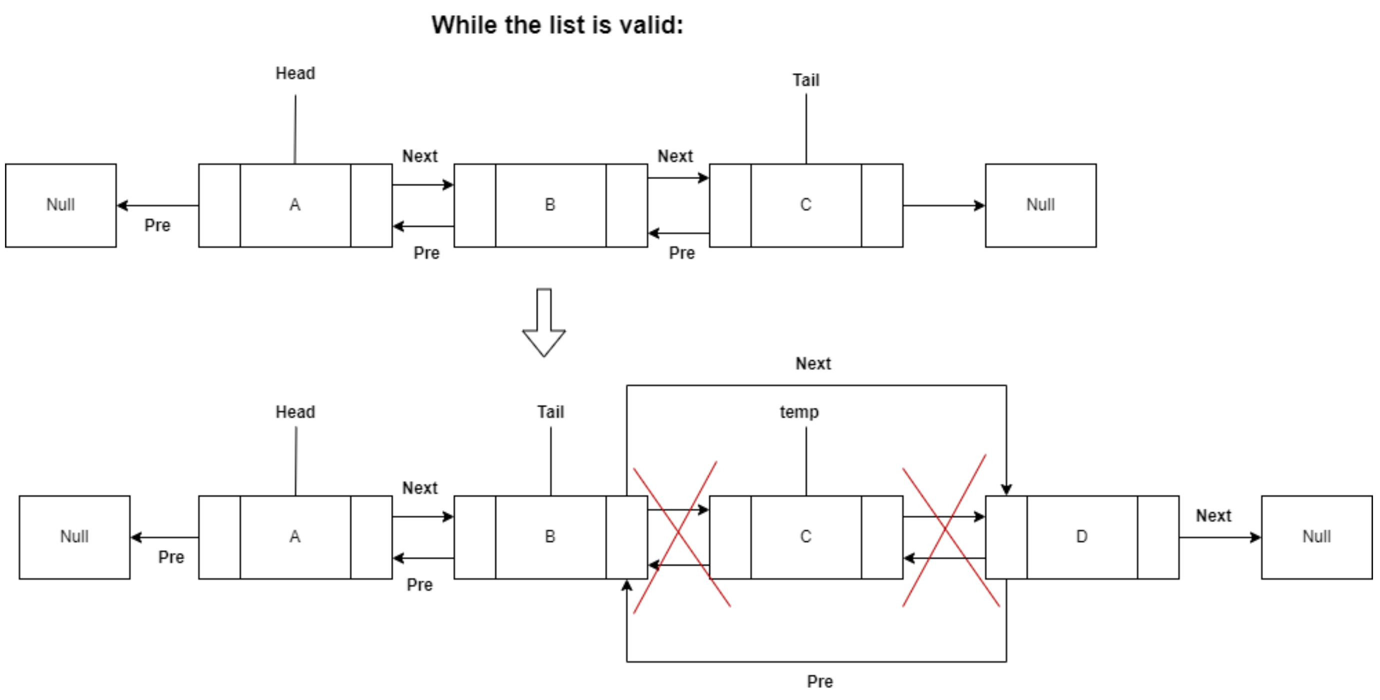
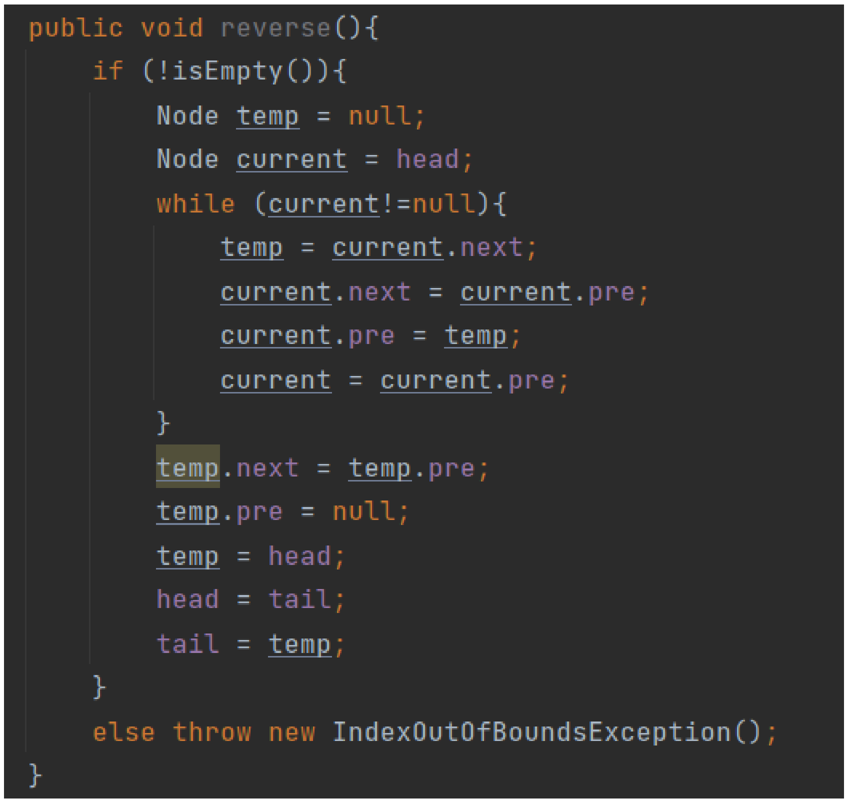


Figure 32. Illustration

3.2.7 Reverse a Doubly Linked List



```
public void reverse(){
    if (!isEmpty()){
        Node temp = null;
        Node current = head;
        while (current!=null){
            temp = current.next;
            current.next = current.pre;
            current.pre = temp;
            current = current.pre;
        }
        temp.next = temp.pre;
        temp.pre = null;
        temp = head;
        head = tail;
        tail = temp;
    }
    else throw new IndexOutOfBoundsException();
}
```

Figure 33. reverse operation

In this operation, if the list is not empty **(!isEmpty()),** it will reverse the list. The below picture is how the operation works:

While the list is valid:

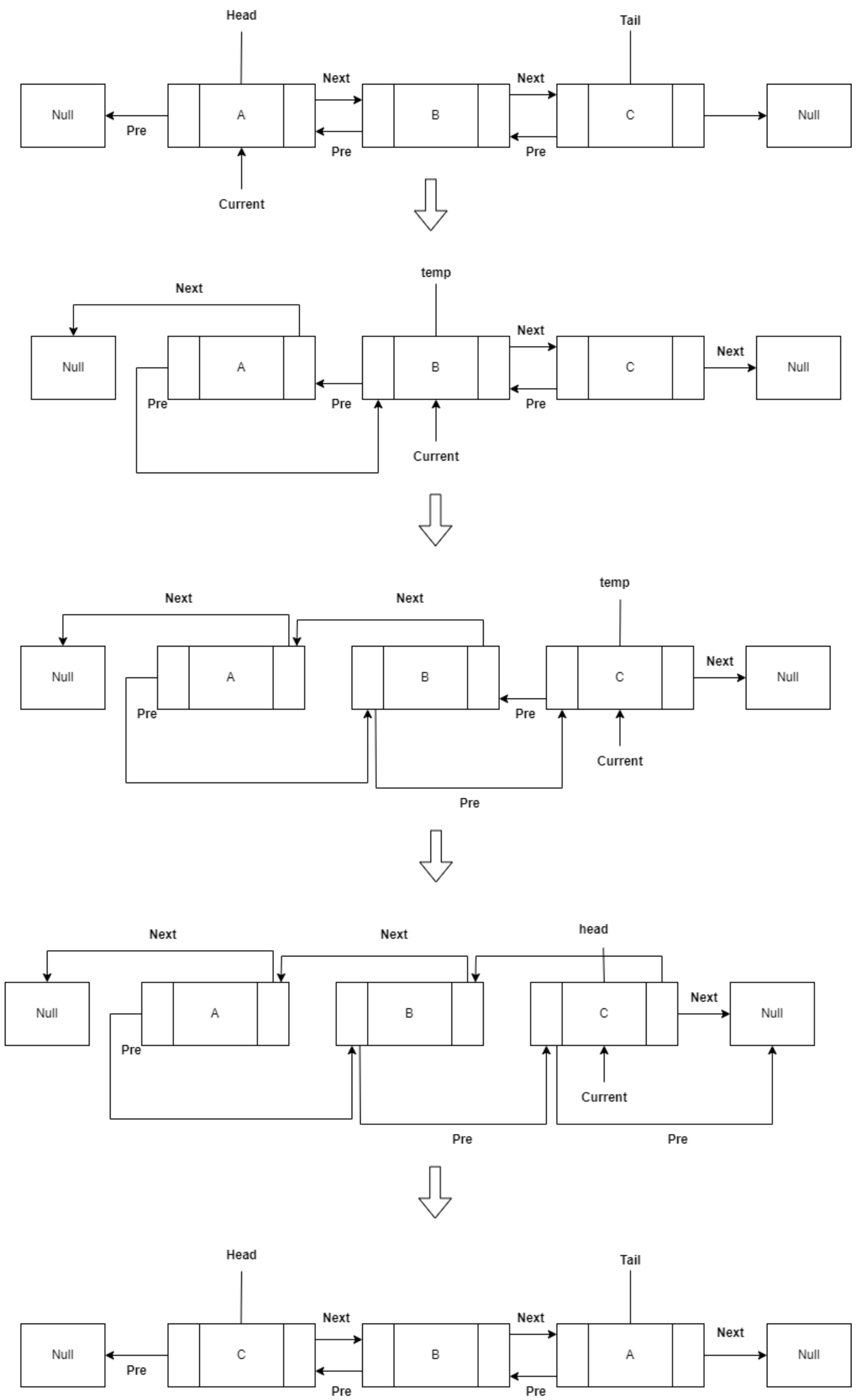
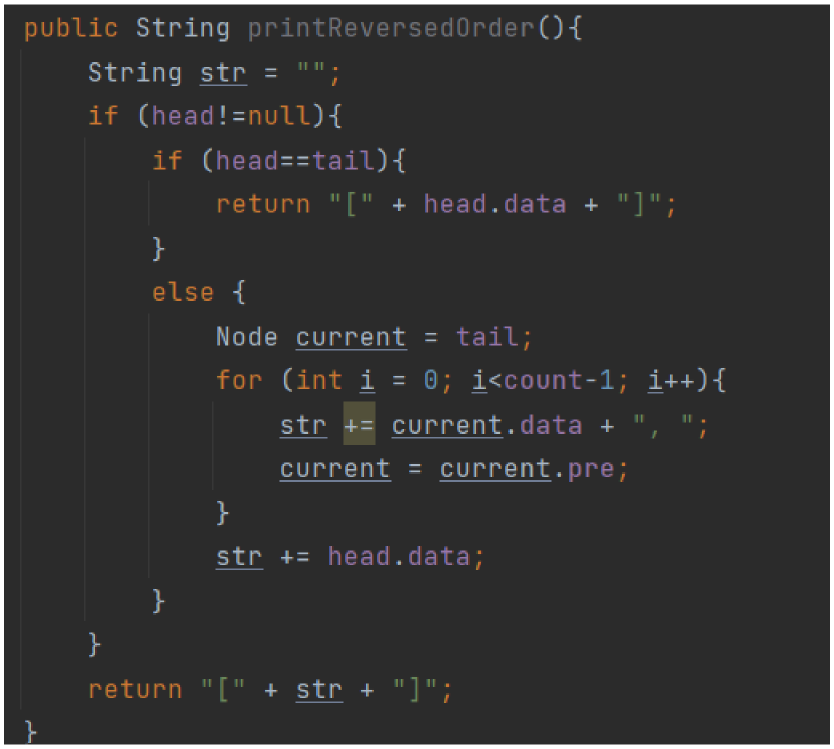


Figure 34. Illustration

3.2.8 Print the list in reserve order



```java
public String printReversedOrder(){
    String str = "";
    if (head!=null){
        if (head==tail){
            return "[" + head.data + "]";
        }
        else {
            Node current = tail;
            for (int i = 0; i<count-1; i++){
                str += current.data + ", ";
                current = current.pre;
            }
            str += head.data;
        }
    }
    return "[" + str + "]";
}
```

Figure 35. printInReversedOrder

This function is simple printing the **Singly Linked List** in reverse order. Its return type is **String**, and it takes no parameter. The logic of this operation is opposite to **Singly Linked List**. It runs from the last node to the first node. And at each step, the result will be concatenated by the current's data with a "," and the previous result. However, if the current is **head** then the result simply be concatenated by the current's data and the previous result. To be more understandable, the result will be packaged in "[]".

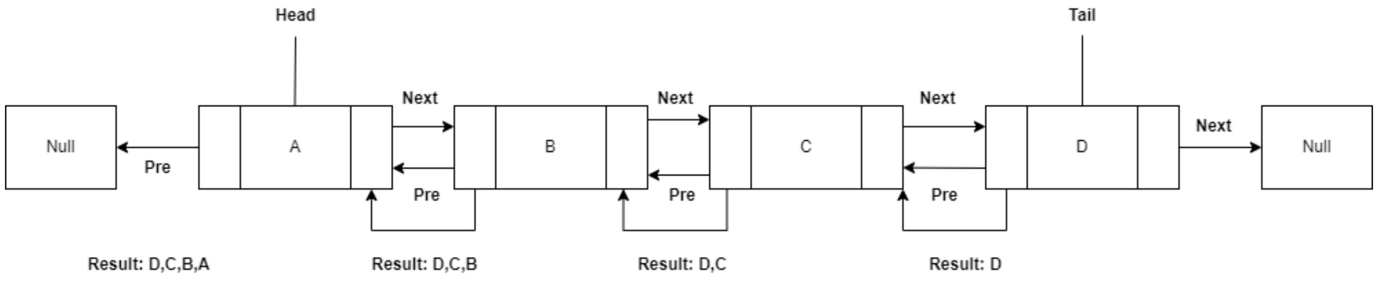
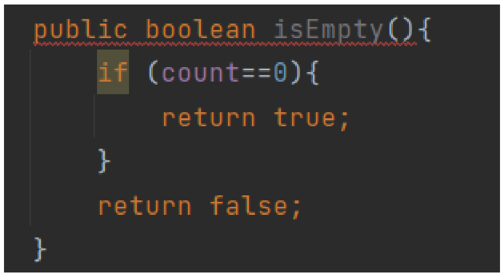


Figure 36. Illustration

3.2.9 Empty

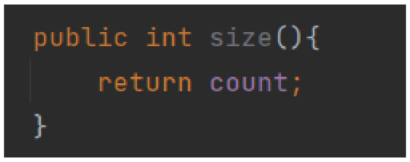


```
public boolean isEmpty(){
    if (count==0){
        return true;
    }
    return false;
}
```

Figure 37. isEmpty operation

The above operation is used to check the list has element or not. If it has element, it returns false. Otherwise, it returns true.

3.2.10 Size



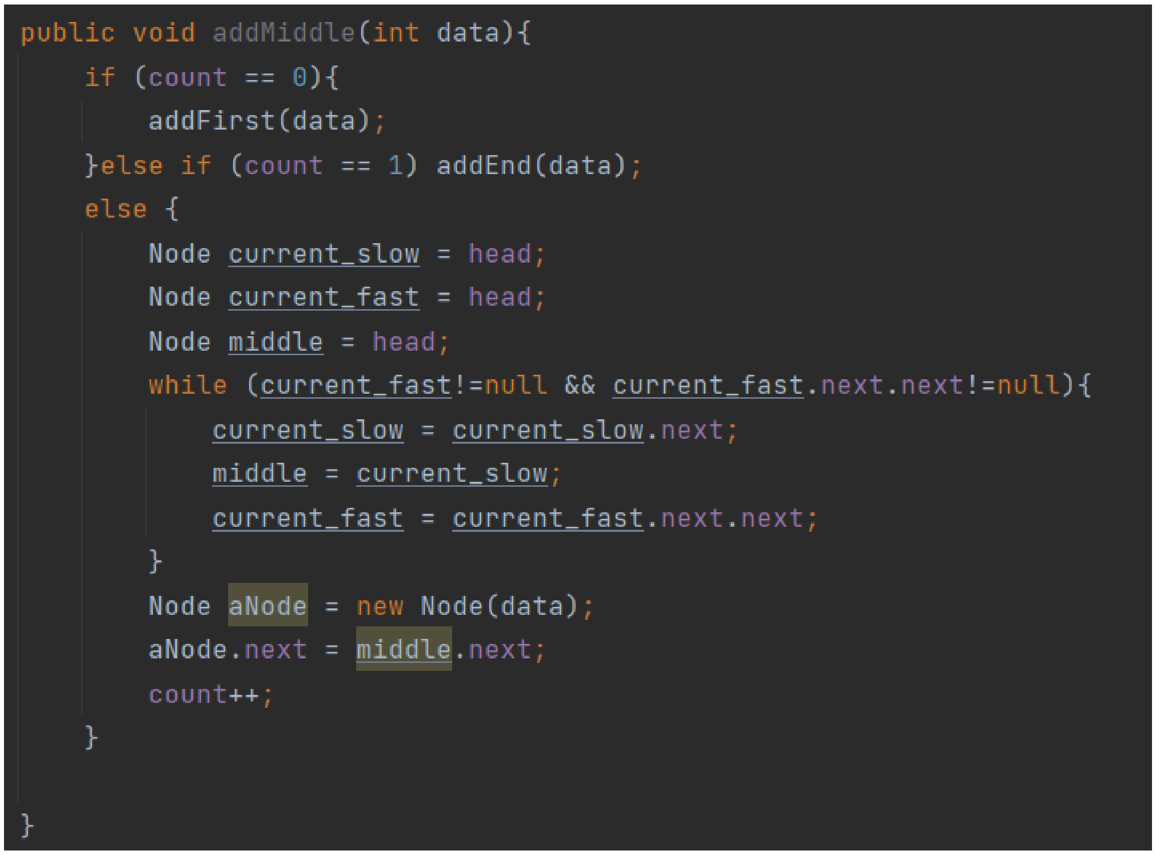
```
public int size(){
    return count;
}
```

Figure 38. size operation

The above operation shows the size of the list (the total of elements in the list). Its return type is **Integer**.

4. Insert an element in the middle of a Linked List

4.1 Singly Linked List



```
public void addMiddle(int data){
    if (count == 0){
        addFirst(data);
    }else if (count == 1) addEnd(data);
    else {
        Node current_slow = head;
        Node current_fast = head;
        Node middle = head;
        while (current_fast!=null && current_fast.next.next!=null){
            current_slow = current_slow.next;
            middle = current_slow;
            current_fast = current_fast.next.next;
        }
        Node aNode = new Node(data);
        aNode.next = middle.next;
        count++;
    }

}
```

Figure 39. addMiddle operation

This operation used to add a new element to the middle of the list. Firstly, if the size of the list (count) is zero, it calls **addFirst(data).** When the count == 1, it calls **add(data)** operation. In two cases above, the author cannot add the new element to the middle of the list as to do this, he needs at least 2 elements.

The idea of this operator is use in two pointers. The author initializes 3 variables including **current_slow**, **current_fast** and **middle**. **current_slow** and **current_fast** do the task of finding the middle position of the list by jumping **current_fast** 2 steps, while **current_slow** jumps 1 step. This repeats until **current_fast.next! = Null && current_fast.next.next! = Null**, which means **current_fast** cannot jump anymore and **current_slow** is in the middle now. At this point, the author just creates a new node, named **aNode**, and adds it to the middle position. It creates a new Node, creates the link from the new node (aNode) similar to the connection of the middle element. Then the middle element's connection will be appended to **aNode**. After adding, the size of the list increases by one.

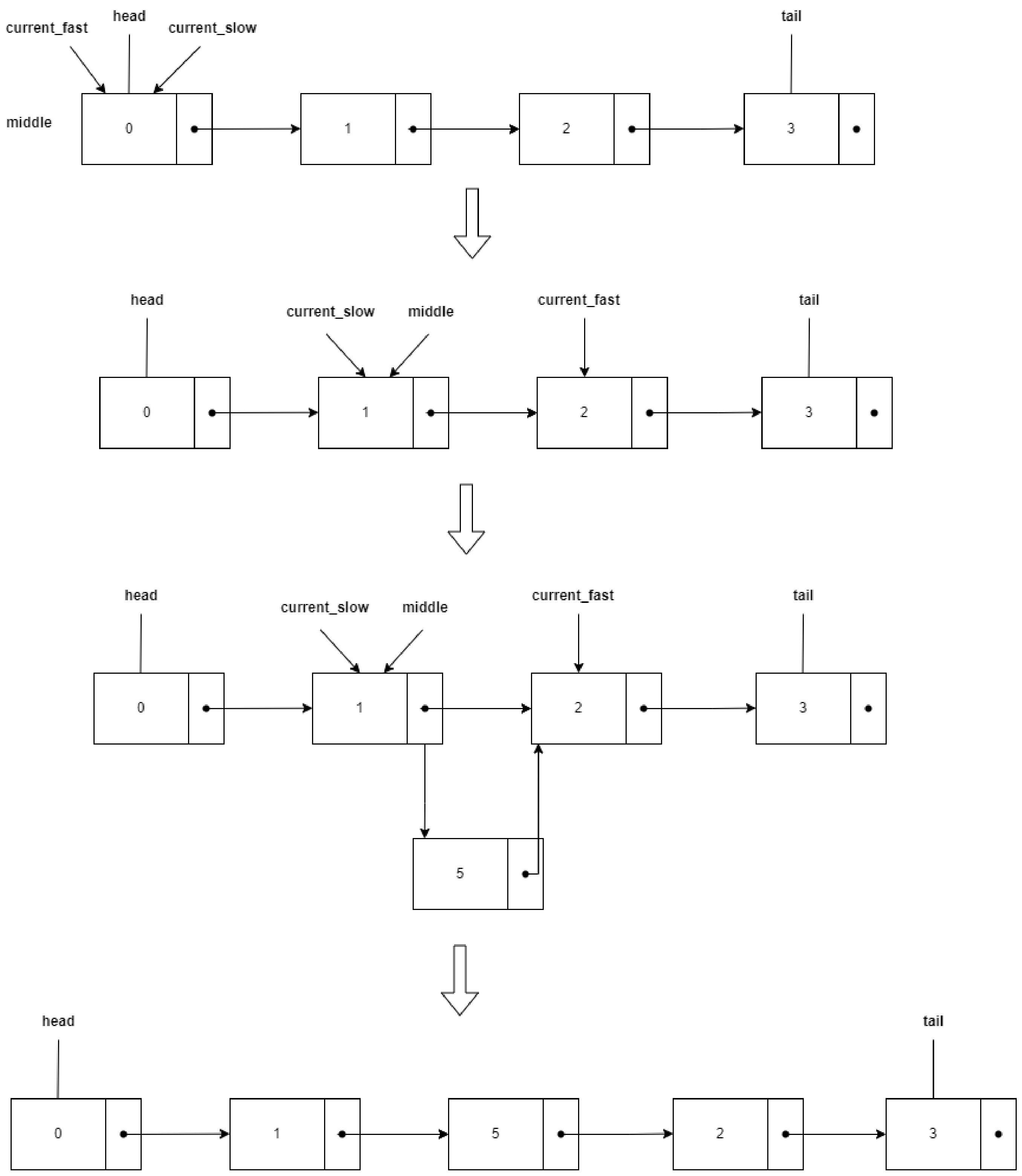


Figure 40. Illustration

4.1.1 Time complexity

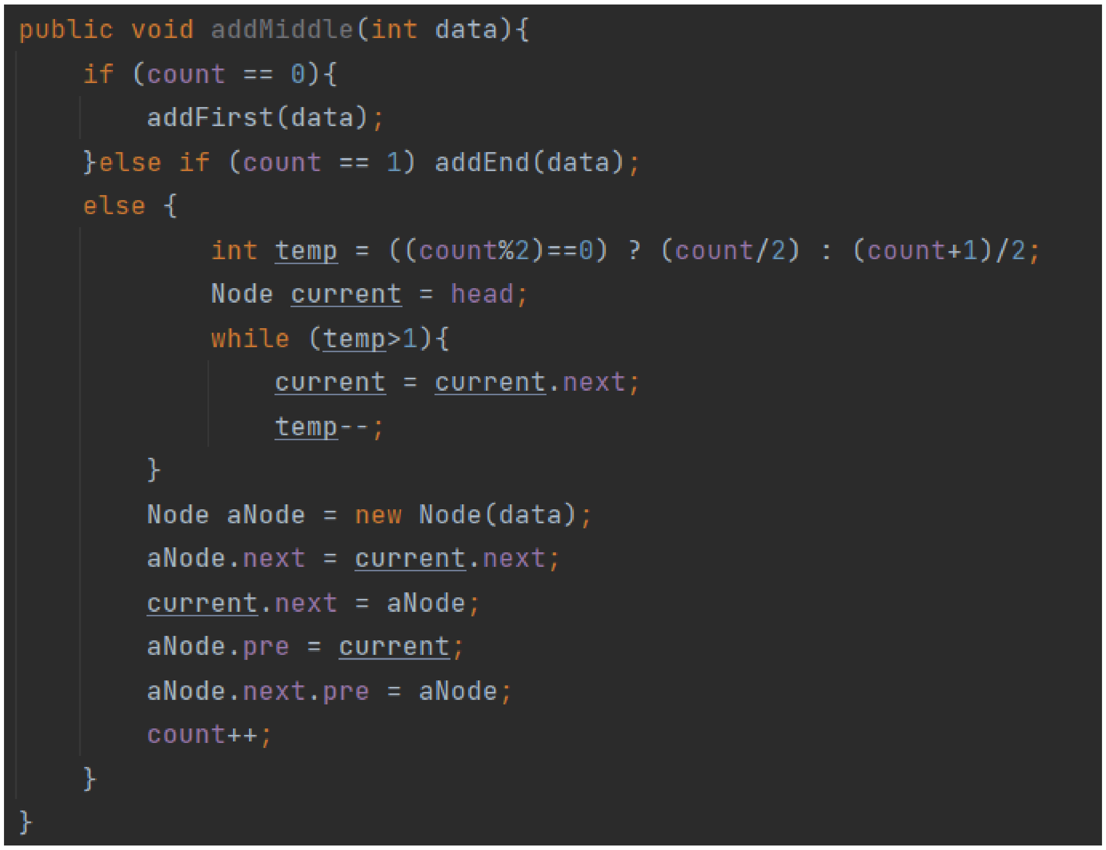
The **time complexity** of the operation above is **O(n)** in worst case, and **O(1)** in best case.

While the size of the list is smaller than 2 (equal to 1 or 0), it just adds a new element to the first (or the last) of the list by calling **addFirst(data)** or **addLast(data)** operation, which takes a constant time (**O(1))**. So it is the best case with **time complexity** is **O(1).**

4.1.2 Space complexity

The author just creates the new list head the node he wishes to insert and links it to the old list head to add a new node to the list (in case the list has zero element). Similar to the scenario when there is just one element in the list, if the author wishes to insert at the end of the list, he can do so by using the list's tail as a pointer and inserting the new node right after it. He only needs to make a new node, link it to the prior and subsequent nodes, and put it anywhere he wants in the list. Thus, this operation's space complexity is **O(1).**

4.2 Doubly Linked List



```java
public void addMiddle(int data){
    if (count == 0){
        addFirst(data);
    }else if (count == 1) addEnd(data);
    else {
            int temp = ((count%2)==0) ? (count/2) : (count+1)/2;
            Node current = head;
            while (temp>1){
                current = current.next;
                temp--;
        }
        Node aNode = new Node(data);
        aNode.next = current.next;
        current.next = aNode;
        aNode.pre = current;
        aNode.next.pre = aNode;
        count++;
    }
}
```

Figure 41. addMiddle operation

The task of this operation is similar to the operation **addMiddle** in **Singly Linked List**. However, the way of implementing is a bit different. Find the number of nodes or length of the linked and let it be count. Calculate **temp = count/2**, if temp is even, else **temp = (count+1)/2**, if temp is odd. Traverse again the first nodes and insert the new node after the temp-th node.

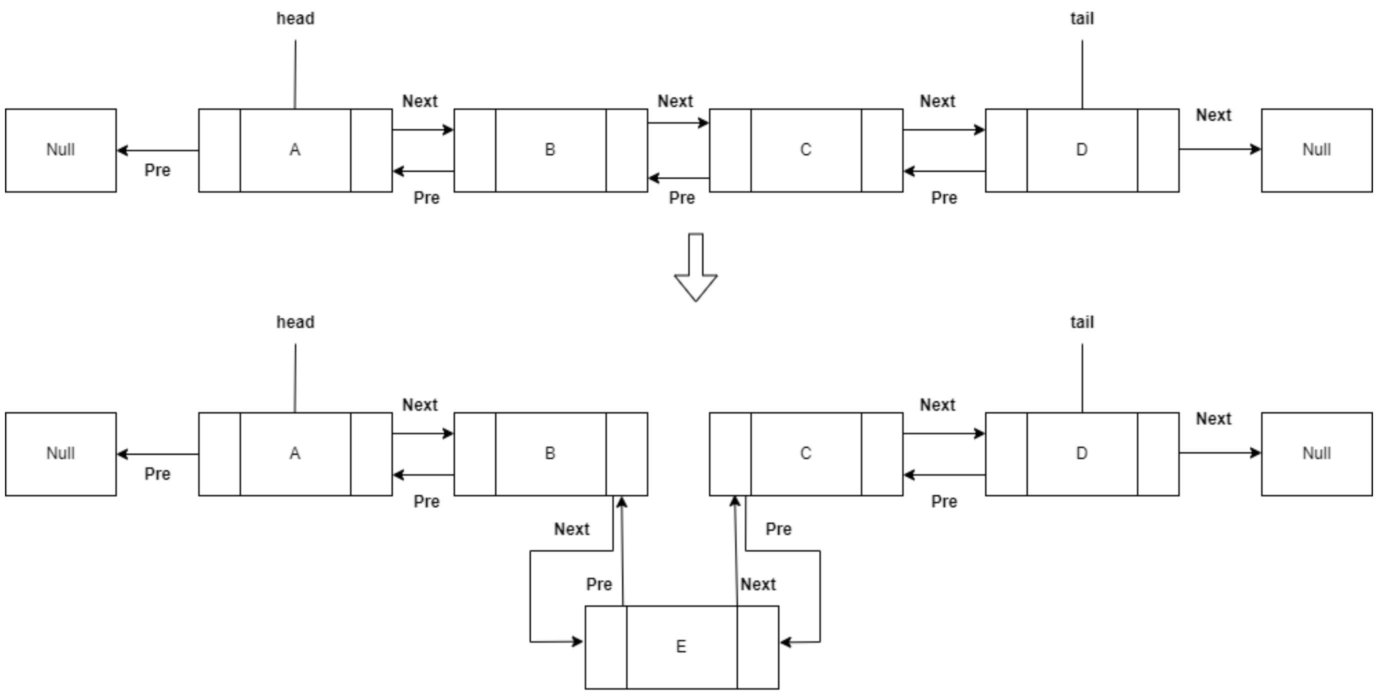


Figure 42. Illustration

4.2.1 Time complexity

The time complexity of this operation is similar with **addMiddle** of **Singly Linked List**, which is mentioned above. The time complexity is **O(1)** in best case, where the size (count) is equal to 0 or 1. While the size of the list is smaller than 2 (equal to 1 or 0), it just also adds a new element to the first (or the last) of the list by calling **addFirst(data)** or **addLast(data)** operation, which takes a constant time (**O(1)).** So it is the best case with **time complexity** is **O(1)**.

In case of the size of the list is bigger or equal to 2, it needs to traverse the first node (head) to the node after the **temp-th**. Which means, the step of the loop is depended on the size of the list. The bigger size the list is, the more times of the loop. So, when the size is bigger than 2, it would be the worst case with the **time complexity** is **O(n)**.

4.2.2 Space complexity

The **space complexity** of this operation is basically familiar with the space complexity of **addElement** operation of **Singly Linked List**. Thus, the **space complexity** is **O(1)**.

5. Sorting in Linked-List

Selected algorithm

The author will employ merge sort to order the linked list. Merge Sort was chosen by the author as the sorting technique since linked lists and arrays utilize different amounts of memory. Linkedlist nodes may not be close to one another in memory, unlike arras. People may access an element in an array with an

O(1) time complexity. For instance, if the author has an integer (4-byte) array A and the address of A[0] is x, he can directly access memory at (x + i*4) to access A[i].

An algorithm called merge sort utilizes a divide-and-conquer strategy to sort the items in an array. Recursively splitting the input into two halves, it rearranges and merges the two sorted halves. Merge Sort basically compares the two sizes of the array mathematically before merging them into a sorted array after dividing the array into two sizes recursively until it can no longer split.

It must alter the data at the first node if it is not the smallest value in the linked list since the code shown below sorts linked lists by changing next links rather than the data at the nodes.

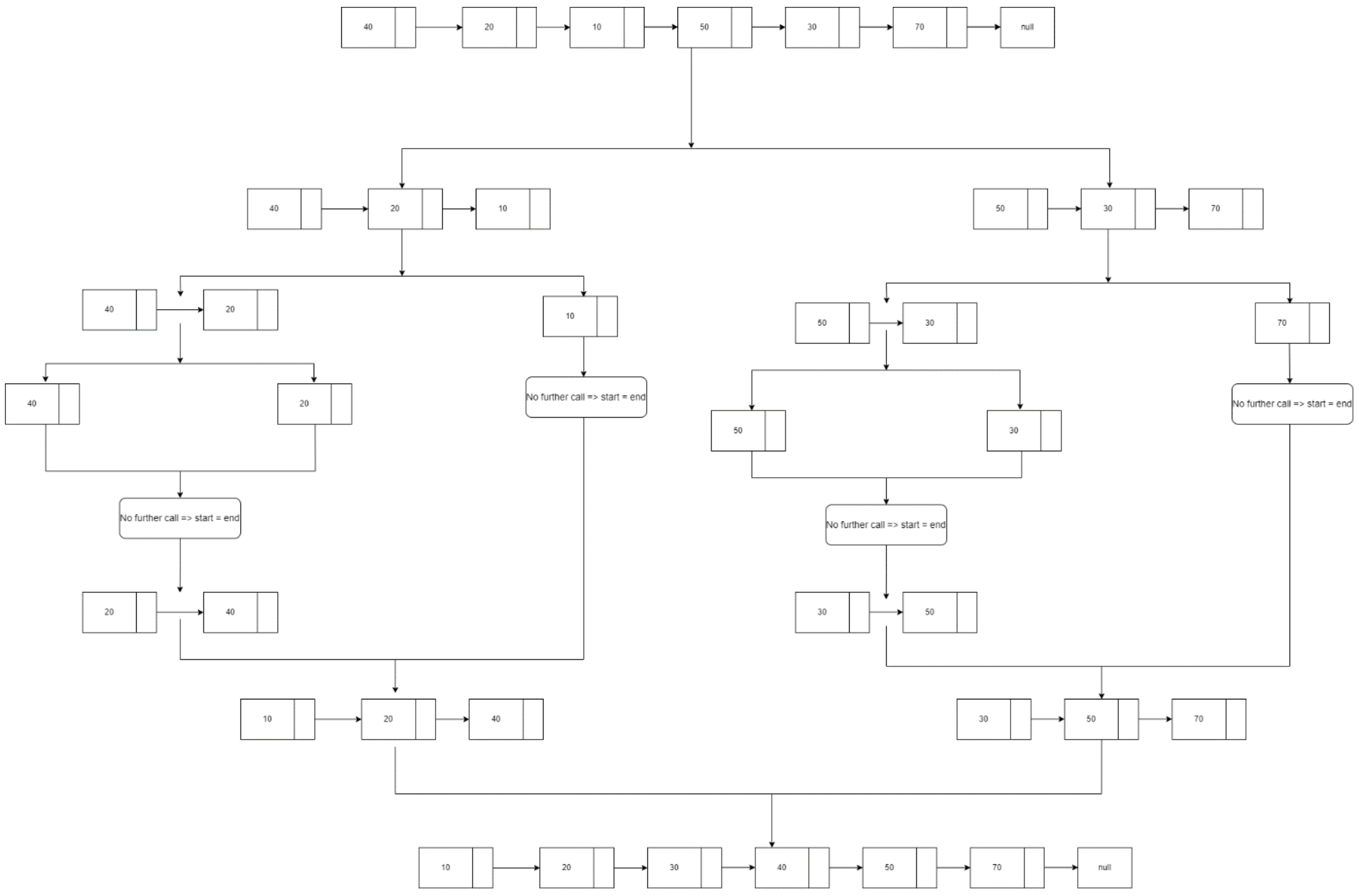

40
20
10
50
30
70
null
No further call => start = end
No further call => start = end
No further call => start = end
No further call => start = end

Figure 43. Merge Sort with Linked List

In the example above, the input linked list is split into two sub-linked lists each half the size of the previous step array, until no further division is possible. Consider a short-linked list with the items 40->20->10 to be more specific. The reasoning behind choosing the middle element is comparable to the reasoning behind **addMiddle**, which the author previously described. The connection between the middle element and middle.next is broken after selecting the middle element.

This list of three items is split into two sub-linked lists in the first stage. the first being made up of components (40–>20) and the second (10). The initial list of components (40->20) has now been further separated into two sub-linked lists, each of which contains elements (40) and (20), respectively.

Since there is no way to further decompose this list and each sub-linked list can only have one entry, the author will combine these linked lists. A new linked list (20->40) is created by merging the two sub-linked lists that were created in the previous phase in sorted order. In order to create the new sorted list, the author must combine this list with the list that has element (10) in the past (10-20-60).

The author's method for merging two independent linked lists makes use of fake nodes. He began the result list with a temporary fake node. It is simple to add (or append) new nodes since the pointer tail always links to the final node in the result list.

Implementation

In this case, the primary data structure used by **Merge Sort** is **Linked List**.

The **sortedMerge** will be the author's starting point. The linked list, which has a maximum size of n, is simply iterated through. In the remaining part of the section, we only compare and assign values; it is a constant-time operation. The **sortedMerge** function thus has a running time of **O(n)**.

It takes **O(n)** times to determine the middle element of the linked list (as explained in the previous section).

Then the **mergerSort** is breaking when it divides the size of the input into two sub-linked lists, each of which is the size of the linked list from the step before it. Assuming that it divides into one element at a time, the time may be represented as follows:

$$n/2_k = 1 \rightarrow k = \log_2 n$$

Furthermore, the whole steps will be repeated with every input (all types of given linked list, both sorted linked list or unsorted linked list.

So, the **MergeSort time complexity** is **O(n log(n)).**

The **space complexity of MergeSort is O(n): n** auxiliary space is required in Merge Sort implementation as all the elements are copied into an auxiliary linked list.

III. Implement error handling and Report test result

1. Testing Plan

The author will offer several test cases in this part to determine whether the program's performance and efficiency level are satisfactory or not. In doing so, he will provide his thoughts on the project's merits, shortcomings, mistakes, and suggestions for how to improve it.

1.1 Singly Linked List

No	Scope	Operation	Testing Type	Input	Expected Output	Actual Output	Status
1	Singly Linked List ADT: MySinglyLinkedList aList	**add(item i)**	Normal	aList: [4,6,8]; add(10)	aList: [4, 6, 8] Size of aList: 4 getLast() return 10	The same as expected output	Passed
			Data validation	aList: [4,6,8]; add(B)	aList: [4,6,8] print an error message	aList: [4,6,8] The system is stopped	Failed
2		**addFirst(item i)**	Normal	aList:[]; addFirst(0)	aList:[0] Size of aList: 1 getFirst() return 1=0	The same as expected output	Passed
			Data validation	aList:[]; addFirst(B)	aList:[] Print an error message	aList:[] The system is stopped	Failed
		addIndex(index a, item x)	Normal	aList:[4,5,6]; addIndex(4,7)	aList:[4,5,6,7] Size of aList: 4 Get(4) returns 1	The same as expected output	Passed
			Data validation	aList:[2,3,4] addIndex(-1,1)	aList:[2,3,4] The system is stopped Print error:"**IndexOutOfBoundsException**"	The same as expected output	Passed
				aList:[2,3,4] addIndex(-1,B)	aList:[2,3,4] Print an error message	aList:[2,3,4] The system is stopped	Failed
3		**removeFirst()**	Normal	aList:[1,2,3] removeFirst()	aList:[2,3] Size of aList: 2 getFirst() returns 1	The same as expected output	Passed
			Data validation	aList[] removeFirst()	aList[] The system is stopped Print error message:"**NoSuchElementsException**"	The same as expected output	Passed
4		**removeLast()**	Normal	aList[1,2,3] removeLast()	aList[1,2] Size of aList = 2 getLast() returns 1	The same as expected output	Passed
			Data validation	aList[] removeLast()	aList[] The system is stopped	The same as expected output	Passed

					Print an error message"**NoSuchElementsException**"		
5		**removeIndex(index i)**	Normal	aList[1,2,3] removeIndex(2)	aList[1,3] Size of aList = 2 get(2) return 2	The same as expected output	Passed
			Data validation	aList[] removeIndex(0)	aList[] The system is stopped Print an error message"**NoSuchElementsException**"	The same as expected output	Passed
				aList[] removeIndex(5)	aList[] The system is stopped Print an error message"**IndexOutOfBoundException**"	The same as expected output	Passed
				aList[] removeIndex(A)	aList[] Print an error message	aList[] The system is stopped	Failed
6		**rangeCheck(index i)**	Normal	aList[1,2,3] rangeCheck(1)	aList[1,2,3] returns nothing	The same as expected output	Passed
			Data validation	aList[1,2,3] rangeCheck(-1)	aList[1,2,3] The system is stopped Print an error message"**IndexOutOfBoundException**"	The same as expected output	Passed
				aList[1,2,3] rangeCheck(-5)	aList[1,2,3] The system is stopped Print an error message"**IndexOutOfBoundsException**"	The same as expected output	Passed
				aList[1,2,3] rangeCheck(-A)	aList[1,2,3] Print an error message	aList[1,2,3] The system is stopped	Failed
7		**printInReversedOrder()**	Normal	aList[1,2,3] printInReversedOrder()	Returns [3,2,1] getLast() returns 3 getFirst() returns 1	The same as expected output	Passed
			Data validation	aList[] printInReversedOrder()	Returns [] getLast() returns null	The same as expected outputs	Passed

					getFirst() returns null		🟩

Table 2. Test case of Singly Linked List

1.2 Doubly Linked List

No	Scope	Operation	Testing Type	Input	Expected Output	Actual Output	Status

1	Doubly Linked List ADT: MyDoublyLinkedList aList	**add(item i)**	Normal	aList: [4,6,8];
				add(10)
					aList: [4, 6, 8]
					Size of aList: 4
					getLast() return 10
						The same as expected output	Passed

			Data validation	aList: [4,6,8];
				add(B)
					aList: [4,6,8]
					print an error message
						aList: [4,6,8]
						The system is stopped
							Failed

2		**addFirst(item i)**	Normal	aList:[];
				addFirst(0)
					aList:[0]
					Size of aList: 1
					getFirst() return 1=0
						The same as expected output	Passed

			Data validation	aList:[];
				addFirst(B)
					aList:[]
					Print an error message
						aList:[]
						The system is stopped
							Failed

		addIndex(index a, item x)	Normal	aList:[4,5,6];
				addIndex(4,7)
					aList:[4,5,6,7]
					Size of aList: 4
					Get(4) returns 1
						The same as expected output	Passed

			Data validation	aList:[2,3,4] addIndex(-1,1)	aList:[2,3,4] The system is stopped Print error:"**IndexOutOfBoundsException**"	The same as expected output	Passed
				aList:[2,3,4] addIndex(-1,B)	aList:[2,3,4] Print an error message	aList:[2,3,4] The system is stopped	Failed
3		**removeFirst()**	Normal	aList:[1,2,3] removeFirst()	aList:[2,3] Size of aList: 2 getFirst() returns 1	The same as expected output	Passed
			Data validation	aList[] removeFirst()	aList[] The system is stopped Print error message:"**NoSuchElementsException**"	The same as expected output	Passed
4		**removeLast()**	Normal	aList[1,2,3] removeLast()	aList[1,2] Size of aList = 2 getLast() returns 1	The same as expected output	Passed

			Data validation	aList[] removeLast()	aList[] The system is stopped Print an error message”**NoSuchElementsException**”	The same as expected output	Passed
5		**removeIndex(index i)**	Normal	aList[1,2,3] removeIndex(2)	aList[1,3] Size of aList = 2 get(2) return 2	The same as expected output	Passed
			Data validation	aList[] removeIndex(0)	aList[] The system is stopped Print an error message”**NoSuchElementsException**”	The same as expected output	Passed
				aList[] removeIndex(5)	aList[] The system is stopped Print an error message”**IndexOutOfBoundException**”	The same as expected output	Passed
				aList[] removeIndex(A)	aList[] Print an error message	aList[] The system is stopped	Failed
6		**rangeCheck(index i)**	Normal	aList[1,2,3] rangeCheck(1)	aList[1,2,3] returns nothing	The same as expected output	Passed
			Data validation	aList[1,2,3] rangeCheck(-1)	aList[1,2,3] The system is stopped Print an error message”**IndexOutOfBoundException**”	The same as expected output	Passed
				aList[1,2,3] rangeCheck(-5)	aList[1,2,3] The system is stopped Print an error message”**IndexOutOfBoundsException**”	The same as expected output	Passed
				aList[1,2,3] rangeCheck(-A)	aList[1,2,3] Print an error message	aList[1,2,3] The system is stopped	Failed
7		**printInReversedOrder()**	Normal	aList[1,2,3] printInReversedOrder()	Returns [3,2,1] getLast() returns 3 getFirst() returns 1	The same as expected output	Passed

			Data validation	aList[] printInReversed Order()	Returns [] getLast() returns null getFirst() returns null	The same as expected outputs	Passed

Table 3. Test case of Doubly Linked List

2. Evaluation

There are **28** test cases examined for the Single Linked List and the Doubly Linked List, respectively. The author's test case had a failure rate of **6/28 (21,4%)**.

The operations have generally been examined, assessed, and carried out logically. The majority of the input, specifically the data and index, are not checked. As a result, the system is unable to rationally analyze the program and manage the data, which creates several faults.

There are 28 test cases (in Singly Linked List and Doubly Linked List, respectively) The application still needs several data validation functionalities after testing, according to the test case table. This is because the creation process only takes place over a short period of time, hence the creator did not give the software full validation. The author encountered issues with input processing in a few instances when the user's input was random and they could enter anything they wanted, including characters, numbers, symbols, etc.

But the author's application only permits **INTEGER** number input, and there is no filter other than **rangeCheck** to verify that the input is correct. It had the effect of making the test cases fail.

The author must create a filter to address this issue, such as one that ensures the system prints an error notice and asks users to reenter if the index they entered from the device is not a number. He must also check the data's validity when dealing with incorrect data formats, and he must forbid the user from entering any input other than integers.

IV. Discuss how asymptotic analysis can be used to access the effectiveness of an algorithm

Asymptotic notations are used to show how long an algorithm needs to execute when the input tends towards a certain value or a limiting value. Think about the following example: The procedure runs in linear time, which is the best-case situation, when the input array has already been sorted. The worst-case situation is that the method takes the longest (quadratic) time to sort the items when the input array is arranged in reverse. When the input list is unsorted or in reverse order, it takes an average amount of time. These durations are denoted using asymptotic notations (Programmiz, 2021).

There are mainly three asymptotic notations:

- Big-O Notation
- Omega Notation
- Theta Notation

1. Big O

Big-O notation is used to express an algorithm's maximum allowable running time. As a result, it provides an algorithm's worst-case complexity. Consider the instance of Insertion Sort. It takes linear time in the best scenario and quadratic time in the worst. It is fair to assume that the time complexity of the Insertion sort is O(n2). It's important to remember that O(n2) also accounts for linear time. The Big O notation is useful when we only know an upper bound on the time complexity of an algorithm. Additionally, we may establish an upper bound by just examining the method (GeeksforGeeks, 2021).

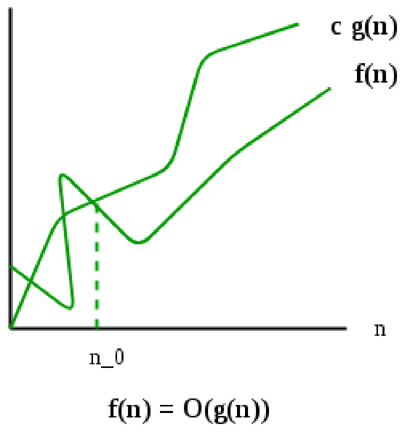


Figure 44. Big-O gives the upper bound of a function (GeeksforGeeks, 2021).

2. Omega Notation

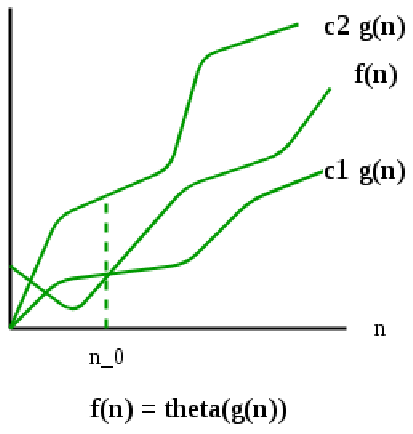


Figure 45. Theta bounds the function within constants factors (GeeksforGeeks, 2021).

Theta notation bounds a function from above and below to represent precise asymptotic action.

It is simple to convert an equation to Theta notation by dropping low order terms and disregarding leading constants. As an illustration, consider the following sentence: (GeeksforGeeks, 2021).

3n3 + 6n2 + 6000 = O (n3)

Regardless of the constants involved, dropping lower order terms is always ideal since there will always be a number (n) after which (n3) has greater values than (n2) (n2). We designate g(n) is the following collection of functions for a given function (GeeksforGeeks, 2021).

3. Theta Notation

The lowest bound of an algorithm's execution time is shown in omega notation. As a result, it offers an algorithm's best-case complexity (GeeksforGeeks, 2021).

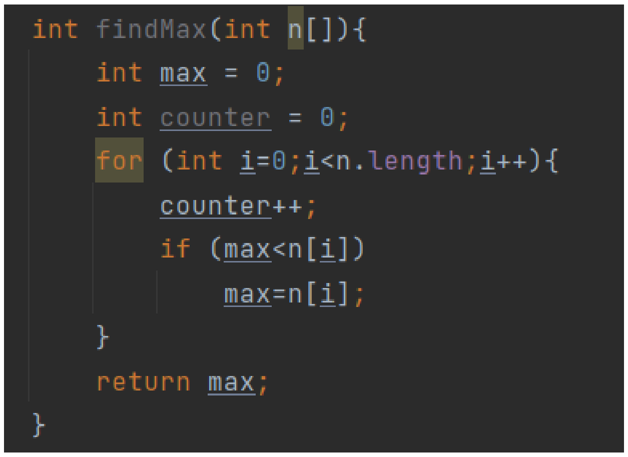
If there is a positive constant c that, for sufficiently big n, stands above cg(n), the aforementioned statement can be seen as a function f(n) belonging to the set (g(n)). Omega (g(n)) provides the algorithm's minimal time requirement for any value of n. (GeeksforGeeks, 2021).

V. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example

A characteristic that characterizes an algorithm's effectiveness in terms of the volume of data it must process is the algorithm's complexity. This function's domain and range are often represented in terms of natural units. The effectiveness of an algorithm is measured by two main complexity tests.

A function called **time complexity** indicates how long an algorithm takes in relation to the quantity of data it receives. The term "time" can be used to refer to any natural unit that is connected to how much time the method would take in real time, such as the number of memory accesses, integer comparisons, the frequency with which an inner loop is run, etc. Since actual time may be impacted by several factors unrelated to the algorithm, we attempt to maintain this idea of time distinct from "wall clock" time (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.). It turns out that if we pick our units carefully, everything else will work themselves out. It turns out that if we select the units carefully, everything else doesn't matter and we can obtain a measure of the algorithm's effectiveness on its own (educative, 2021).

For instance, assume that the author wants to find the maximum value of a non-negative unsorted array. The code is given below:



```
int findMax(int n[]){
    int max = 0;
    int counter = 0;
    for (int i=0;i<n.length;i++){
        counter++;
        if (max<n[i])
            max=n[i];
    }
    return max;
}
```

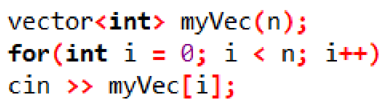
Figure 46. findMax function

So, how many operations will the **findMax** do? Basically, it checks every element from n. If the current element is bigger than max it will do an assignment. Here, the author added a **counter** so it can help him count how many times the inner block is executed.

Now, the time complexity of the above code would be:
- Line 47-48: 2 operations.
- Line 49: a loop of size n.
- Line 50-52: 3 operations inside the for loop.

So, this gets the author 3(n)+2 operations. Applying the asymptotic analysis, the author can only leave the most significant term, thus: n. Finally, he gets O(n). He can verify this using counter.

A function called "**space complexity**" describes how much memory (or "space") an algorithm requires in relation to how much input it receives. When we talk about "additional" memory needs, we don't include the RAM needed to store input. Again, we use conventional (but fixed-length) units to compute this. Even if we can utilize bytes, metrics like the quantity of integers or fixed-size structures used are more practical. Finally, the number of bytes required to represent the unit will have no impact on the functionality we design. Space complexity is frequently disregarded because it is not necessary and/or evident, although it may be just as important as time.



```
vector<int> myVec(n);
for(int i = 0; i < n; i++)
cin >> myVec[i];
```

Figure 47. Example

The author is making a vector of size n in the aforementioned example. Therefore, the given code has a space complexity on the order of "n," meaning that as n increases, so will the space need. AfterAcademy (2021) states that "In typical programming, you will be able to use 256MB of space for a specific task.

As a result, you are only permitted to utilize 256MB, hence you are unable to build an array larger than 108 bytes. Additionally, because a function may only utilize up to 4MB of space, it is not possible to generate an array larger than 106 in size. In order to use a larger array, you can establish a global array.

VI. Conclusion

The author of this paper has thoroughly described the software as well as the primary distinction between SLL and DLL. In addition, the author supplied a detailed implementation, pseudocode, example, and efficient sorting method for Linked Lists. Additionally, the testing strategy was provided and assessed. He talked on how asymptotic analysis may be used to analyze the three advantages of employing the implementation-independent data structure, trade-offs when describing an ADT, and algorithm efficacy.

Following this topic, the author learned a lot about ADT and how to apply it to make a practical application that addresses a practical issue. In order to make the system work successfully and efficiently, he frequently has varied priorities for time or space in particular scenarios. By measuring the program, he will make sure that the system meets the necessary requirements for operation.

VII. References

AfterAcademy, 2021. Time and Space Complexity Analysis of Algorithm. [online] Afteracademy.com. Available at: <https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm> [Accessed 1 May 2021].

educative, 2021. Time complexity vs. space complexity. [online] Educative: Interactive Courses for Software Developers. Available at: <https://www.educative.io/edpresso/time-complexity-vs-space-complexity> [Accessed 1 May 2021].

GeeksforGeeks, 2021. Analysis of Algorithms | Set 3 (Asymptotic Notations) - GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic   notations/> [Accessed 1 May 2021].

Programmiz, 2021. Big-O Notation, Omega Notation and Big-O Notation (Asymptotic Analysis). [online] Programiz.com. Available at: <https://www.programiz.com/dsa/asymptotic-notations> [Accessed 1 May 2021].

Sharma, N., 2019. Difference between Singly linked list and Doubly linked list in Java. [online] Tutorialspoint.com. Available at: <https://www.tutorialspoint.com/difference-between-singly-linked-list-and-doubly-linked-list-in   java> [Accessed 28 April 2021].

Suh, E., 2021. Space-time tradeoff in Computer Science - Cprogramming.com. [online] Cprogramming.com. Available at: <https://www.cprogramming.com/tutorial/computersciencetheory/space-time-tradeoff.html> [Accessed 2 May 2021].