ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	1/9/2022	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Mai Thế Đức	**Student ID**	GCH200681
Class	GCH0907	**Assessor name**	Hong-Quan Do

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	[illegible]

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

TABLE OF FIGURES

INTRODUCTION

In keeping with the previous report, my role this time is to give appropriate implementations and evaluations for the corporate document. This article will discuss the distinctions between a singly linked list and a doubly linked list. I'll also give examples of how to implement key operations, such adding, deleting, and adding elements in the midst of a linked list. A sorting algorithm will be included in my linked list.

P4: Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem.

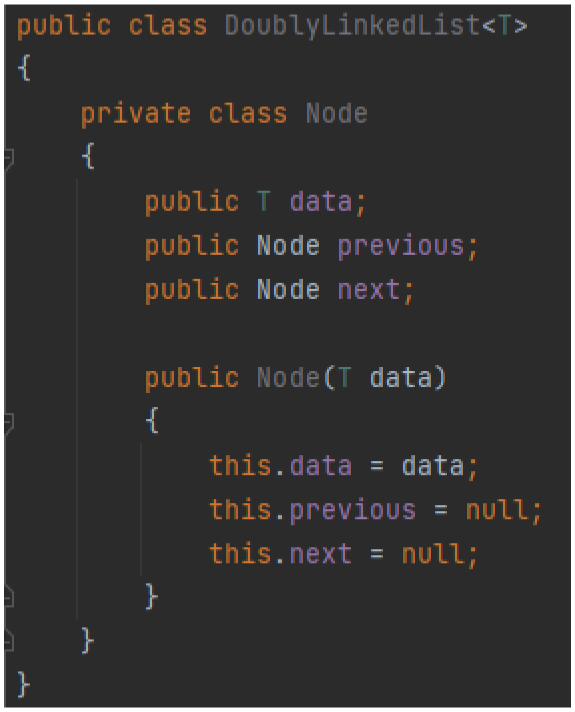
I. IMPLEMENT COMPLEX DATA STRUCTURES AND ALGORITHMS

A. Singly vs. Doubly Linked-List

According to Sharma (2021), singly and doubly linked-list have some differences:

Singly Linked List	Doubly Linked List
A Singly Linked has nodes with a data field and a next link field.	A Doubly Linked List has a previous link field along with a data field and a next link field.
The only way to traverse a singly linked list is by using the link to the node after it.	A Doubly Linked List allows for traversal utilising both the previous node link and the next node link.
Having only 2 fields, a singly linked list takes up less memory than a double linked list.	Due to its three fields, a doubly linked list takes up more memory than a singly linked list.

+) DOUBLY LINKED-LIST



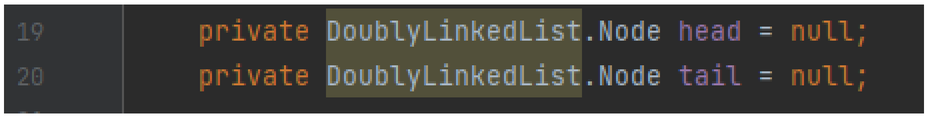
```
public class DoublyLinkedList<T>
{
    private class Node
    {
        public T data;
        public Node previous;
        public Node next;

        public Node(T data)
        {
            this.data = data;
            this.previous = null;
            this.next = null;
        }
    }
}
```

‘T’ is the type of element held in this collection

The data field contains the information or data, whereas the previous field contains the address of the node before it, and the next field contains the address of the node after it.

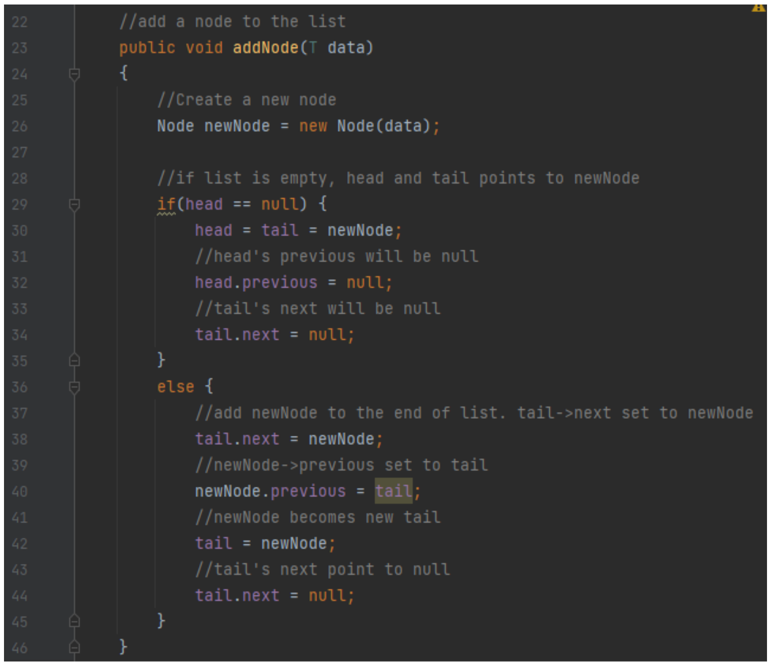
First step, we create a node class for doubly linked-list.



```
    private DoublyLinkedList.Node head = null;
    private DoublyLinkedList.Node tail = null;
```

To begin, we set the head and tail to null.

+) Operation: add an element to the end of the list



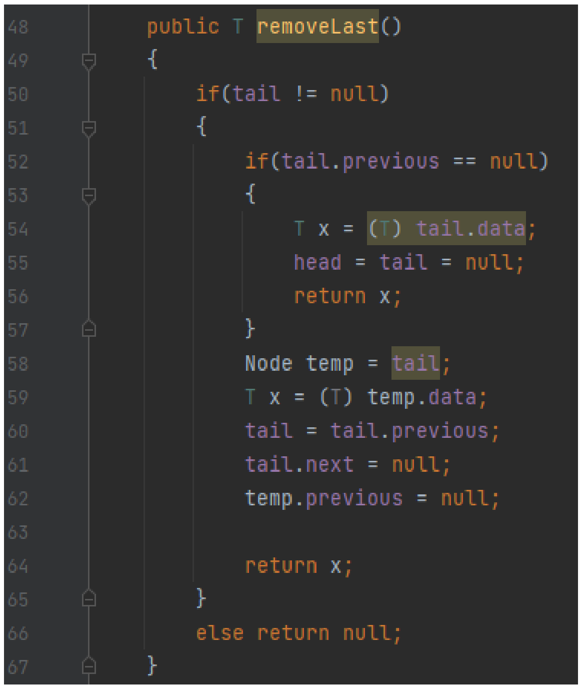
```
22      //add a node to the list
23      public void addNode(T data)
24      {
25          //Create a new node
26          Node newNode = new Node(data);
27
28          //if list is empty, head and tail points to newNode
29          if(head == null) {
30              head = tail = newNode;
31              //head's previous will be null
32              head.previous = null;
33              //tail's next will be null
34              tail.next = null;
35          }
36          else {
37              //add newNode to the end of list. tail->next set to newNode
38              tail.next = newNode;
39              //newNode->previous set to tail
40              newNode.previous = tail;
41              //newNode becomes new tail
42              tail = newNode;
43              //tail's next point to null
44              tail.next = null;
45          }
46      }
```

Line 26 we create a new Node. This new Node will be added to the linked-list.

From line 29 to 35, if the list is empty, we add in a new node and set the very first location of head is null, and also the very last location of tail is null. Now we have a linked-list with only 1 element.

From line 36 to 45, if there already some element in the linked-list then we add the new element in the last node of the linked-list. By making the new Node is the new tail, in line 38 the tail next location connects to the new node and in line 40 the new node previous location connects to the tail. In line 42, the new tail is now the new node. We also set the tail next location to be null.

+) Operation: remove the last element in the linked-list



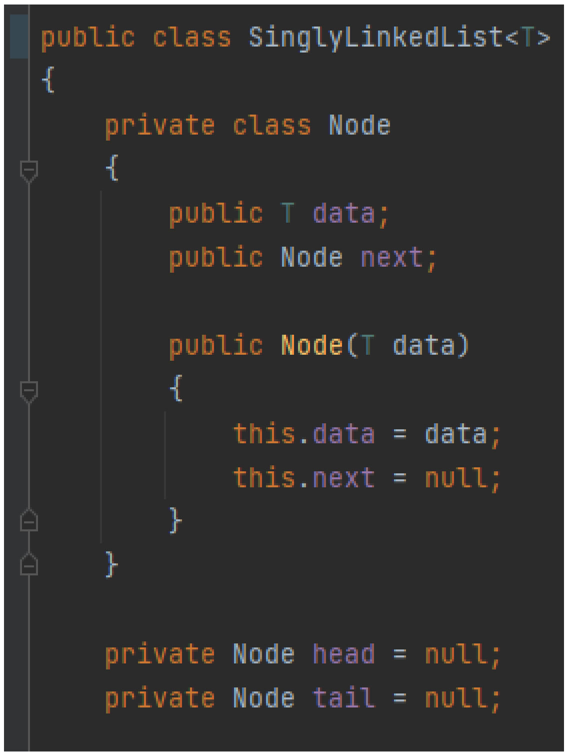
```
48      public T removeLast()
49      {
50          if(tail != null)
51          {
52              if(tail.previous == null)
53              {
54                  T x = (T) tail.data;
55                  head = tail = null;
56                  return x;
57              }
58              Node temp = tail;
59              T x = (T) temp.data;
60              tail = tail.previous;
61              tail.next = null;
62              temp.previous = null;
63
64              return x;
65          }
66          else return null;
67      }
```

Line 50, if the linked-list is not empty then this function will work normally and remove element as usual. But if the linked-list is empty then in line 66 this operation will return null.

Line 52 to 57, this is the case when the linked-list have only 1 element. Therefore, ‘x’ is the element we remove, and we set the linked-list to be null. After remove the element, this operation return ‘x’ to help user know what element has been removed.

Line 58 to 64, if the linked-list have more than 1 element then this operation removes the last element from the list. ‘x’ is the element we remove. We set the almost last node to be tail then we cut the connection with the last node. But also line 62, we must clear the previous location of the last node to disconnect it entirely.

+) SINGLY LINKED-LIST


```
public class SinglyLinkedList<T>
{
    private class Node
    {
        public T data;
        public Node next;

        public Node(T data)
        {
            this.data = data;
            this.next = null;
        }
    }

    private Node head = null;
    private Node tail = null;
```

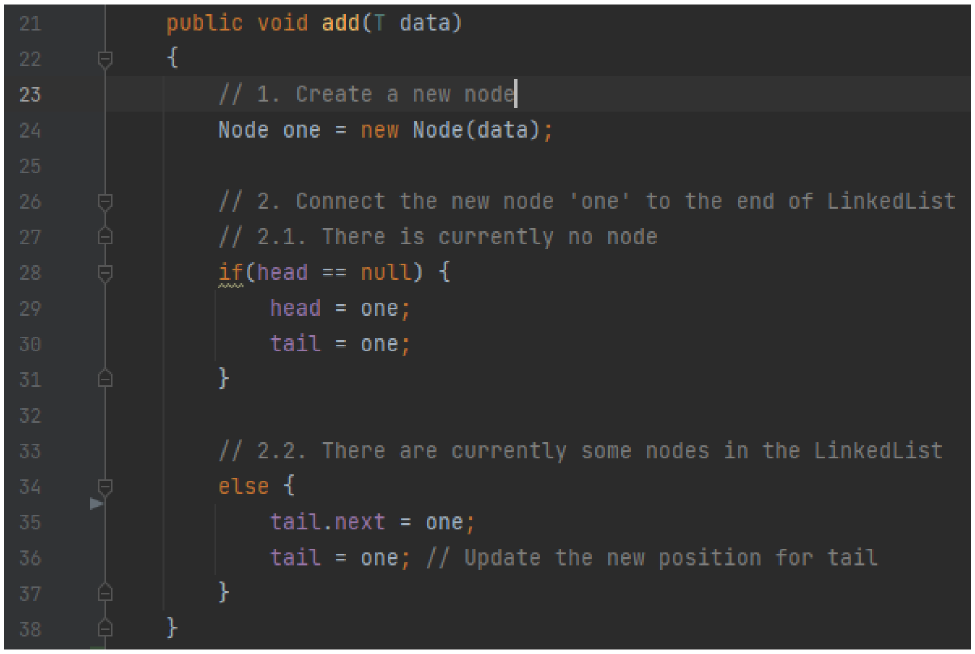
‘T’ is the type of element held in this collection

A Singly Linked has nodes with a data field and a next link field.

First step just like doubly list, we create node class for singly linked-list.

We also set the head and tail to be null.

+) Operation: add an element to the end of the list



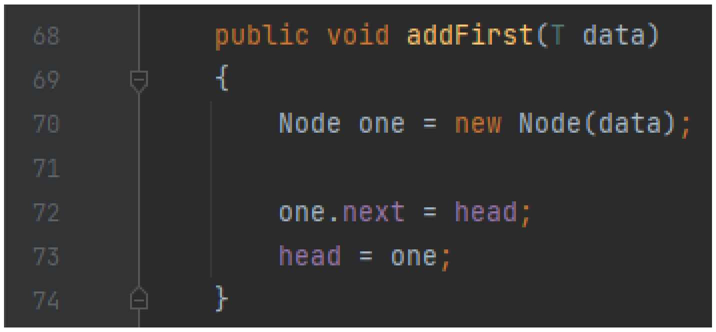
```
21      public void add(T data)
22      {
23          // 1. Create a new node
24          Node one = new Node(data);
25
26          // 2. Connect the new node 'one' to the end of LinkedList
27          // 2.1. There is currently no node
28          if(head == null) {
29              head = one;
30              tail = one;
31          }
32
33          // 2.2. There are currently some nodes in the LinkedList
34          else {
35              tail.next = one;
36              tail = one; // Update the new position for tail
37          }
38      }
```

Line 24 we create a new Node. This new Node will be added to the linked-list.

From line 28 to 31, if the list is empty, we add in a new node. Now we have a linked-list with only 1 element.

From line 34 to 37, if there already some element in the linked-list then we add the new element in the last node of the linked-list. Line 35, we move the tail to the newest node. And then, in line 36 we update the tail new position to the node we added.

+) Operation: add an element to the first of the list

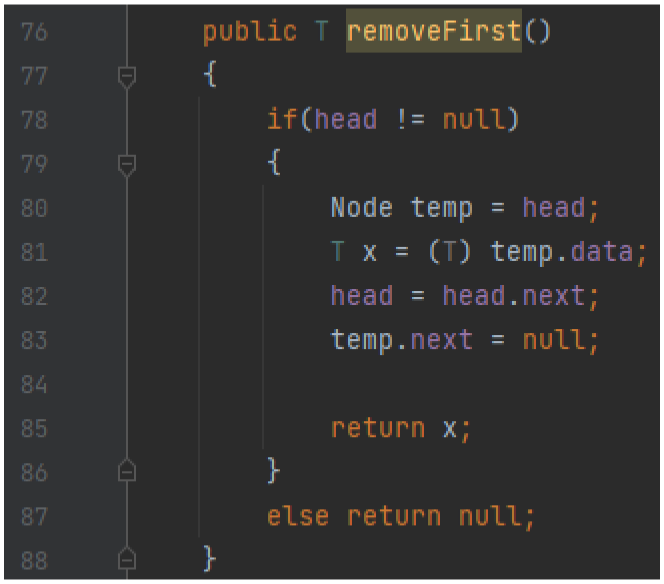


```java
    public void addFirst(T data)
    {
        Node one = new Node(data);

        one.next = head;
        head = one;
    }
```

Difference with the operation add() from above. Operation addFirst() doesn’t care the linked-list have any element or not. It just easy add the new node to the first of the linked-list.

+) Operation: remove the first element in the linked-list

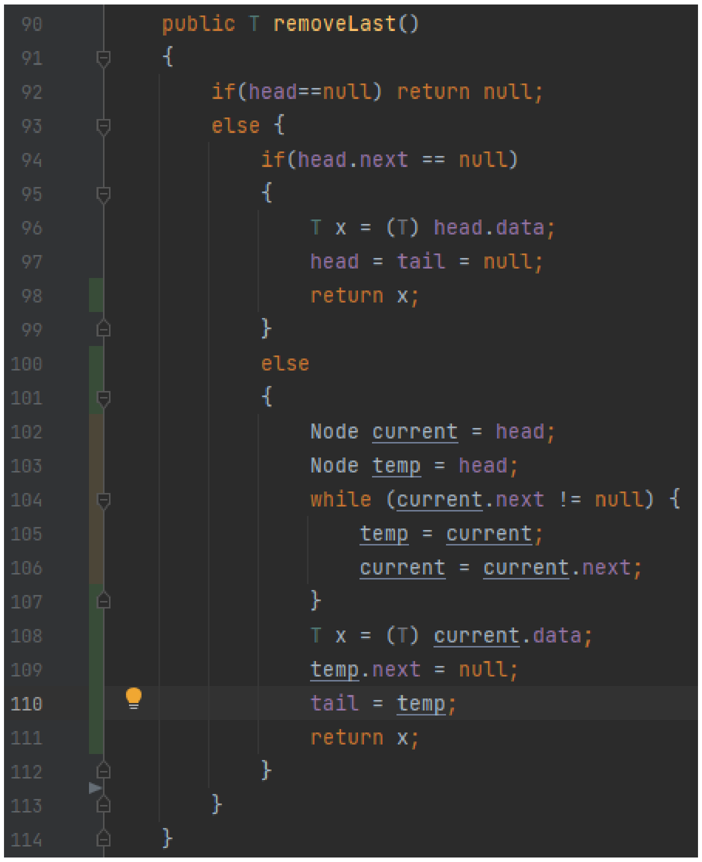


```java
76      public T removeFirst()
77      {
78          if(head != null)
79          {
80              Node temp = head;
81              T x = (T) temp.data;
82              head = head.next;
83              temp.next = null;
84
85              return x;
86          }
87          else return null;
88      }
```

From line 78 to 86, if the list is not null then the operation removes the first element. Therefore, ‘x’ is the element we remove, and we set the linked-list to be null. After remove the element, this operation return ‘x’ to help user know what element has been removed.

Line 87, if the list have no element then this operation return null.

+) Operation: remove the last element in the linked-list



```
90      public T removeLast()
91      {
92          if(head==null) return null;
93          else {
94              if(head.next == null)
95              {
96                  T x = (T) head.data;
97                  head = tail = null;
98                  return x;
99              }
100             else
101             {
102                 Node current = head;
103                 Node temp = head;
104                 while (current.next != null) {
105                     temp = current;
106                     current = current.next;
107                 }
108                 T x = (T) current.data;
109                 temp.next = null;
110                 tail = temp;
111                 return x;
112             }
113         }
114     }
```

Line 92, if the linked-list is empty then this function will return null. Else, it runs from line 93 to 113.

Line 94 to 99, this is the case when the linked-list have only 1 element. Therefore, ‘x’ is the element we remove, and we set the linked-list to be null. After remove the element, this operation return ‘x’ to help user know what element has been removed.

Line 100 to 112, if the linked-list have more than 1 element then this operation removes the last element from the list. ‘x’ is the element we remove. We have ‘current’ and ‘temp’, node ‘current’ will go first then followed by node ‘temp’.

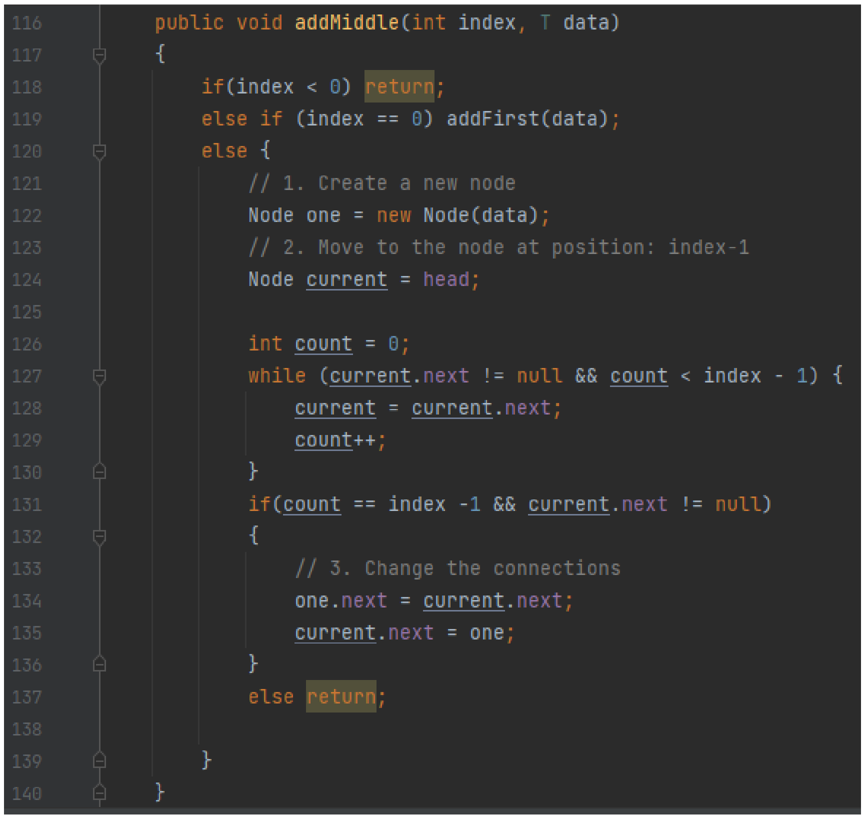
Line 104, until node ‘current’ stop at the last node in the list, the ‘temp’ will always behind ‘current’. So therefore in line 109 - 110, node ‘temp’ will be the new tail and then we cut the connection of it with node ‘current’.

Eventually line 111, we return ‘x’ to help the user know what element they have remove from the linked-list.

B. Insert an element in the middle of a Linked-List

Continue from Singly linked-list.

+) Operation: add an element into the middle of the list



```
116        public void addMiddle(int index, T data)
117        {
118            if(index < 0) return;
119            else if (index == 0) addFirst(data);
120            else {
121                // 1. Create a new node
122                Node one = new Node(data);
123                // 2. Move to the node at position: index-1
124                Node current = head;
125
126                int count = 0;
127                while (current.next != null && count < index - 1) {
128                    current = current.next;
129                    count++;
130                }
131                if(count == index -1 && current.next != null)
132                {
133                    // 3. Change the connections
134                    one.next = current.next;
135                    current.next = one;
136                }
137                else return;
138
139            }
140        }
```

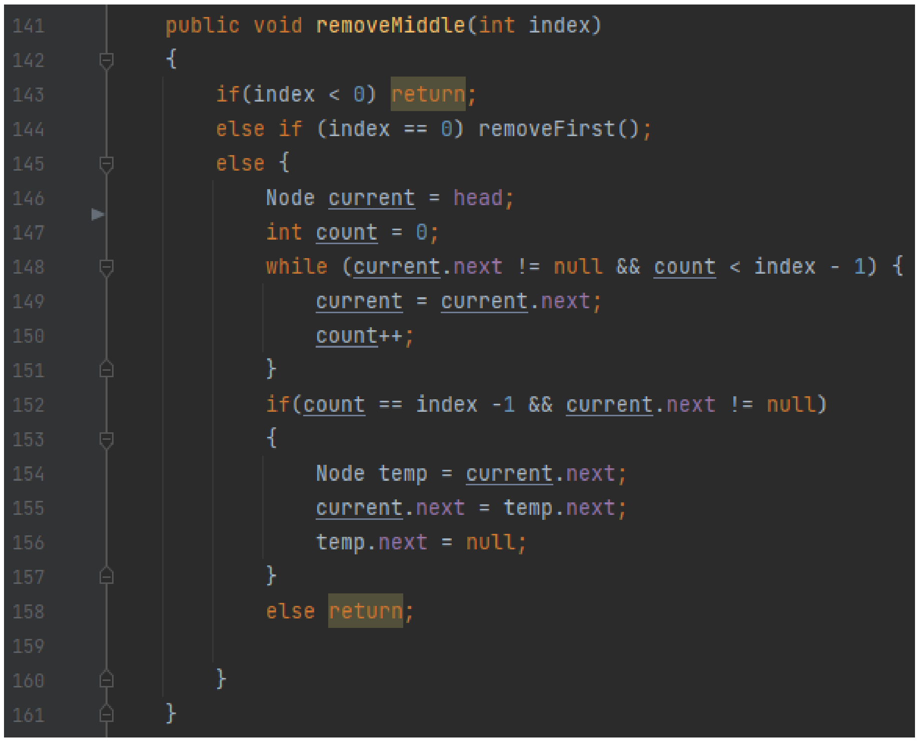
Line 116, we choose ‘index’ and the ‘data’ to add in the middle. So in line 118, if the ‘index’ is smaller than 0 then this operation will stop.

Line 119, if the ‘index’ equal 0 which mean the first element of the linked-list, we will use function addFirst() to operate the task. Line 120 to 139 are the case that ‘index’ > 1.

We use a node ‘current’ and start it at the head of the list. Then we use ‘count’ to locate the index that we want to add item. In line 127, if the ‘index’ we choose is bigger than the index of the list then the loop ‘while’ will stop. Line 137 will end the operation. Line 128 – 129, the node ‘current’ will follow ‘count’ one node behind.

But if ‘count’ found the index first then line 131 – 136 will execute adding the new node in that correct index. Line 134 – 135 will help node ‘one’ have the location of the next node, and also help the current node have the location of node ‘one’.

+) Operation: remove an element from the middle of the list



```
141    public void removeMiddle(int index)
142    {
143        if(index < 0) return;
144        else if (index == 0) removeFirst();
145        else {
146            Node current = head;
147            int count = 0;
148            while (current.next != null && count < index - 1) {
149                current = current.next;
150                count++;
151            }
152            if(count == index -1 && current.next != null)
153            {
154                Node temp = current.next;
155                current.next = temp.next;
156                temp.next = null;
157            }
158            else return;
159
160        }
161    }
```

Similar to operation addMiddle(), line 143 – 144 in charge of checking the index user choose. If ‘index’ smaller than 0, the operation stops. If ‘index’ equal 0 which is remove the first and only element, the operation will use function removeFirst() above.

Line 145 – 160, if the ‘index’ is bigger than 0, the operation will continue. We use a node ‘current’ to track the node we want to remove. We also use ‘count’ again to find the right index.

Line 148 – 151, using a loop, the ‘count’ “move” forward and the ‘current’ step behind. If the ‘index’ user choose is bigger than the list, then operation stop. Until ‘count’ found the target index, Node ‘current’ still move right behind it.

Line 152 - 157, we use a node ‘temp’ as a target to remove. Now if ‘count’ found the match index, then node ‘current’ stand in index – 1 which mean right behind. Now node ‘temp’ is assign to the node where match the target index. The node ‘current’ will connect to the node where index + 1 which mean the next node of node ‘temp’. Lastly, in line 156, we disconnect the node ‘temp’ from the list and this operation has finished it job.

C. Sorting in Linked-List

Heap sort is a comparison-based sorting technique based on the Binary Heap data structure. It is similar to selection sort, where we first find the minimum element and place it at the beginning. We repeat the same process for the remaining elements.

Example:

```
        120 (0)
        /    \
  210 (1)    180 (2)
```

Child 210 (1) is greater then the parent 120 (0)

So we swap child 210 (1) with the parent 120 (0)

```
        210 (0)
        /    \
    120 (1)   180 (2)
```

Heap Sort Algorithm for sorting in increasing order:

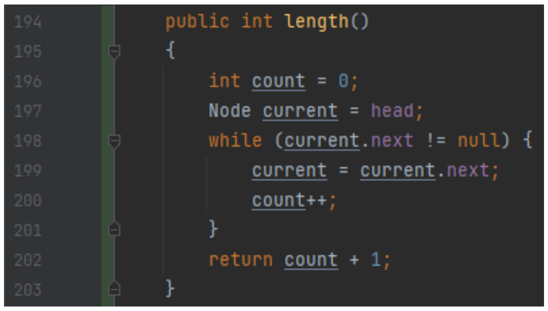
+) Build a max heap from the input data

+) At this point, the largest item is stored at the root of the heap. Replace it with last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.

+) Repeat step 2 while the size of the heap is greater than 1.

Continue from Singly linked-list. But before we sort an linked-list of integer number, we must prepare some support operation.

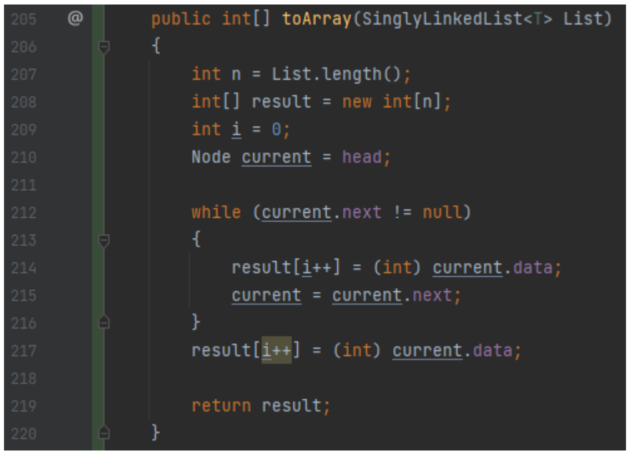
+) Operation: length of the linked-list



```java
public int length()
{
    int count = 0;
    Node current = head;
    while (current.next != null) {
        current = current.next;
        count++;
    }
    return count + 1;
}
```

This operation find the length of the list by using ‘count’. It return a integer ‘count’ for other operation to use.

+) Operation: Convert the linked-list to the array for easy sort

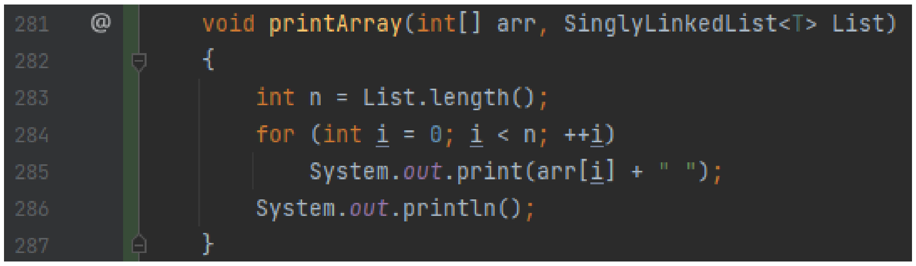


```java
205     public int[] toArray(SinglyLinkedList<T> List)
206     {
207         int n = List.length();
208         int[] result = new int[n];
209         int i = 0;
210         Node current = head;
211
212         while (current.next != null)
213         {
214             result[i++] = (int) current.data;
215             current = current.next;
216         }
217         result[i++] = (int) current.data;
218
219         return result;
220     }
```

Line 210, node ‘current’ starting with head. Our array will be integer array.

Line 212 – 216, we add the item in each node of the list to the array. But this while loop stop at the almost last node, so line 217 we add the last node to the last index of the array.

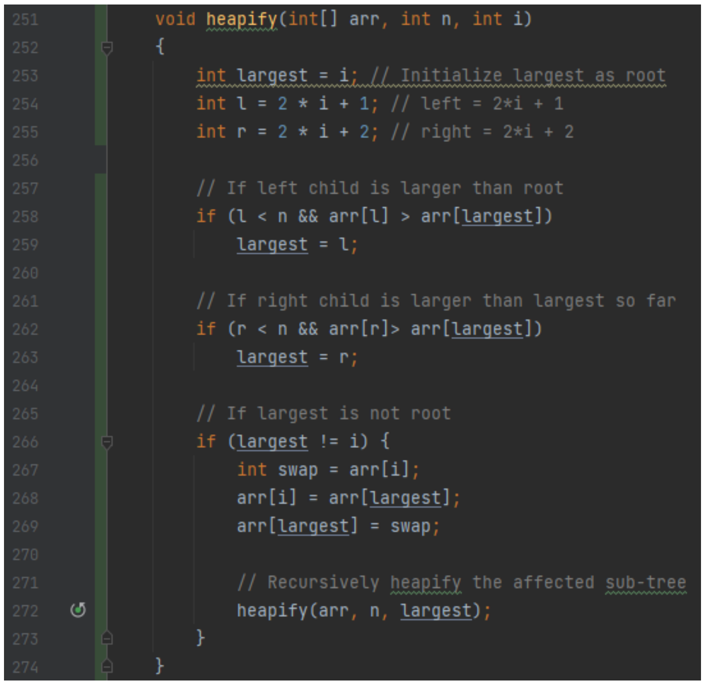
+) Operation: print the array



```
281     void printArray(int[] arr, SinglyLinkedList<T> List)
282     {
283         int n = List.length();
284         for (int i = 0; i < n; ++i)
285             System.out.print(arr[i] + " ");
286         System.out.println();
287     }
```

In line 283, because of list’s length is equal to array’s length so we use it instead of create another operation to find array’s length. This operation prints the array to console.

+) Operation: Heapify

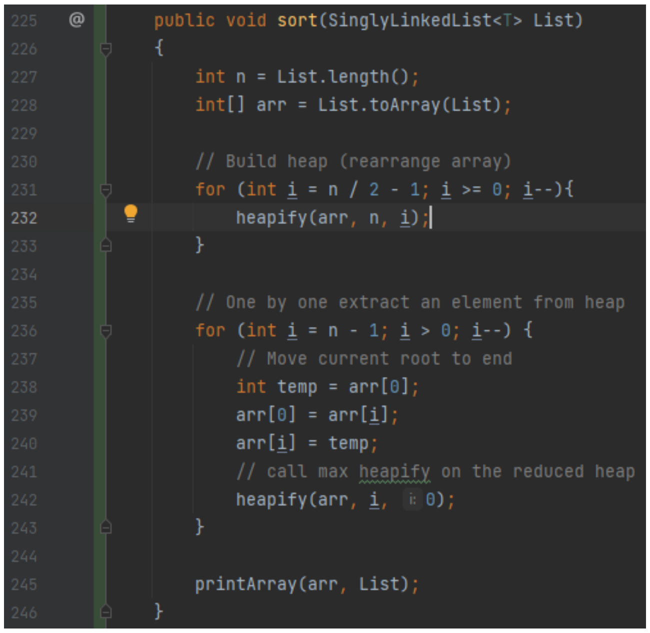


```
251     void heapify(int[] arr, int n, int i)
252     {
253         int largest = i; // Initialize largest as root
254         int l = 2 * i + 1; // left = 2*i + 1
255         int r = 2 * i + 2; // right = 2*i + 2
256 
257         // If left child is larger than root
258         if (l < n && arr[l] > arr[largest])
259             largest = l;
260 
261         // If right child is larger than largest so far
262         if (r < n && arr[r]> arr[largest])
263             largest = r;
264 
265         // If largest is not root
266         if (largest != i) {
267             int swap = arr[i];
268             arr[i] = arr[largest];
269             arr[largest] = swap;
270 
271             // Recursively heapify the affected sub-tree
272             heapify(arr, n, largest);
273         }
274     }
```

Line 251, ‘n’ is size of heap. We also heapify a subtree rooted with node ‘i’ which is an index in arr[].

From line 253 – 269, we heapify 3 number, then line 272 we move to the next node using recursive until the whole tree has become a complete Max heap.

+) Operation: Heap sort



```
225    public void sort(SinglyLinkedList<T> List)
226    {
227        int n = List.length();
228        int[] arr = List.toArray(List);
229
230        // Build heap (rearrange array)
231        for (int i = n / 2 - 1; i >= 0; i--){
232            heapify(arr, n, i);
233        }
234
235        // One by one extract an element from heap
236        for (int i = n - 1; i > 0; i--) {
237            // Move current root to end
238            int temp = arr[0];
239            arr[0] = arr[i];
240            arr[i] = temp;
241            // call max heapify on the reduced heap
242            heapify(arr, i, i: 0);
243        }
244
245        printArray(arr, List);
246    }
```

In this operation we can see line 227, again because of array's length is equal list's length so we use it.

Line 228, we convert this linked-list to array.

Line 231 – 233, this is building heap step but this time we build from the whole tree. This tree is the array, we rearrange array to start build it starting with the first node which have highest index.

With formula (i = n/2-1), the first node is the last internal node. The highest index is slightly below the middle point of the array.

Now after built the Max-heap, in line 236 – 243, we swap the first number with the last number in the array. Because Max-heap so the first number is the maximum number; we'll want to push it to the last of the array. After that we continue heapify the new array with index minus 1.

Because of operation heapify, after swap position of number it recursively heapify the affected sub-tree. So that our tree will remain correct order. The order is: parent node always bigger than childs node, but don't care about same level node.

P5 Implement error handling and report test results.

II. TESTING PLAN

#	Scope	Operation	Testing type	Input	Expected Output	Actual Output	Status
1	Doubly linked-list ADT: Doubly Linked List <Integer> S2	addNode(T data)	Normal	S2:[18,20,15]; addNode(24)	S2: [18,20,15,24];	The same as expected output	Passed
2		removeLast()	Data validation	S2:[]; removeLast()	S2: []; Return null;	The same as expected output	Passed
			Data validation	S2:[18]; removeLast()	S2: []; Return 18;	The same as expected output	Passed
			Normal	S2:[18,20,15]; removeLast()	S2: [18,20]; Return 15;	The same as expected output	Passed
3	Singly linked-list ADT: Singly Linked List <Integer> S1	add(T data)	Normal	S1:[27,30,15]; add(24)	S1:[27,30,15,24];	The same as expected output	Passed
4		addFirst(T data)	Normal	S1:[27,30,15]; addFirst(21)	S1:[21,27,30,15];	The same as expected output	Passed
5		removeFirst()	Data validation	S1:[]; removeFirst()	S1:[]; Return null;	The same as expected output	Passed
			Normal	S1:[27,30,15]; removeFirst()	S1:[30,15]; Return 27;	The same as expected output	Passed
6		removeLast()	Data validation	S1:[]; removeLast()	S1:[]; Return null;	The same as expected output	Passed
			Data validation	S1:[27]; removeLast()	S1:[]; Return 27;	The same as	Passed

						expected output	
			Normal	S1:[27,30,15]; removeLast()	S1:[27,30]; Return 15;	The same as expected output	Passed
7		addMiddle(int index, T data)	Data validation	S1:[27,30,15]; addMiddle(-1,11)	S1:[27,30,15];	The same as expected output	Passed
			Data validation	S1:[27,30,15]; addMiddle(0,11)	S1:[11,27,30,15];	The same as expected output	Passed
			Normal	S1:[27,30,15]; addMiddle(1,11)	S1:[27,11,30,15];	The same as expected output	Passed
			Data validation	S1:[27,30,15]; addMiddle(3,11)	S1:[27,30,15];	The same as expected output	Passed
8		removeMiddle (int index)	Data validation	S1:[27,30,15]; removeMiddle(-1)	S1:[27,30,15];	The same as expected output	Passed
			Data validation	S1:[27,30,15]; removeMiddle(0)	S1:[30,15];	The same as expected output	Passed
			Normal	S1:[27,30,15]; removeMiddle(1)	S1:[27,15];	The same as expected output	Passed
			Data validation	S1:[27,30,15]; removeMiddle(3)	S1:[27,30,15];	The same as expected output	Passed
9		Sort(Singly Linked List <Integer> List)	Normal	S1:[27,30,15,24]; Sort(S1)	S1:[15,24,27,30];	The same as expected output	Passed
			Data validation	S1:[]; Sort(S1)	S1:[]; Print an error message	S1:[]; The system is stopped!	Failed

The sort operation can be improve by manage to return null if there are no element in the linked-list.

P6 DISCUSS HOW ASYMPTOTIC ANALYSIS CAN BE USED TO ASSESS THE EFFECTIVENESS OF AN ALGORITHM

III. ASYPTOTIC ANALYSIS

Calculating the execution time of any operation in terms of mathematical computation units is known as asymptotic analysis. An algorithm's asymptotic analysis involves determining the mathematical bounds or frames of its run-time performance. We may easily draw conclusions about an algorithm's best case, average case, and worst case scenarios using asymptotic analysis (tutorialspoint, 2022).

According to tutorialspoint (2022), there are 3 commonly used asymptotic notations:

- O Notation
- Ω Notation
- θ Notation

+) Big Oh Notation: O

The formal notation for expressing the upper bound of an algorithm's execution time is O(n). It calculates the worst-case time complexity, or the maximum time an algorithm can take to run (tutorialspoint, 2022).

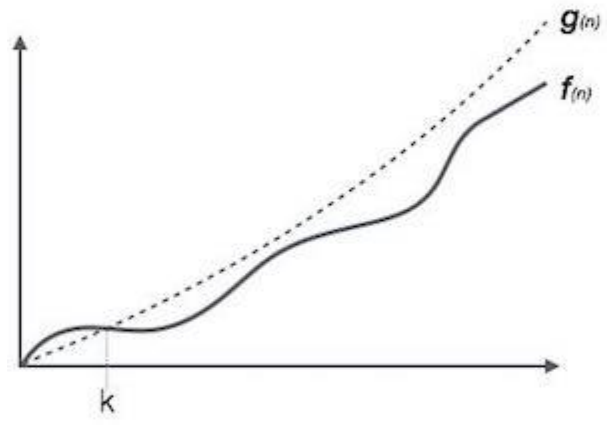


Figure 1: Big Oh Notation, O

Example:

Selection sort have O(n2). Quick sort have O(n*log n). Binary search have O(log n). Simple search have O(n).

+) Omega Notation: Ω

The formal notation for expressing the lower bound of the execution time of an algorithm is (n). It gauges the lowest possible level of time complexity or the fastest possible time an algorithm may run (tutorialspoint, 2022).

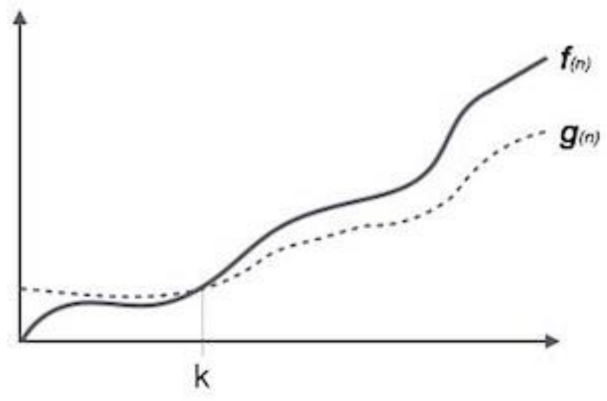


Figure 2: Omega Notation, Ω

Example:

Bubble sort have O(n), Heap sort have O(n*log n).

+) Theta Notation: θ

The formal notation for expressing the lower and upper bounds of an algorithm's execution time is (n). Also, we consider it average run time (tutorialspoint, 2022).

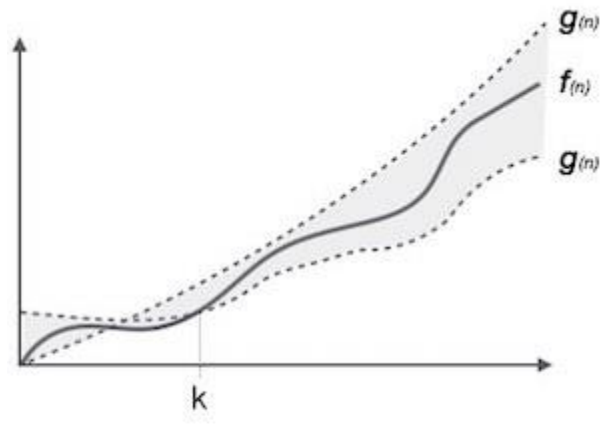


Figure 3: Theta Notation, θ

Example:

Tree sort have O(n*log n), Radix sort have O(nk).

P7 Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example.

IV. EFFICIENCY OF AN ALGORITHM

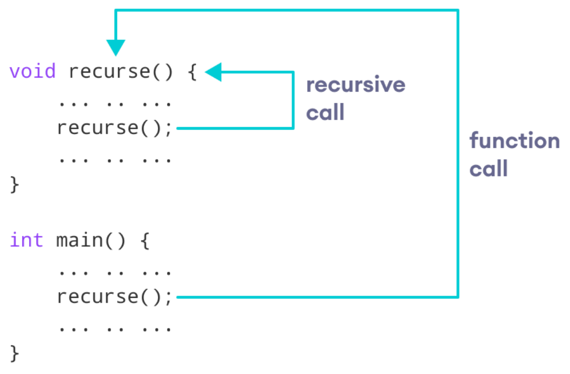
To ascertain an algorithm's resource utilisation, an analysis of the algorithm is required. The quantity of computer resources that an algorithm uses is referred to as the algorithm's efficiency. As a result, the effectiveness of an algorithm can be evaluated depending on how various resources are used. We want to use as little resources as possible for an algorithm to run as efficiently as possible (Kumari, 2022).

There are 2 type of algorithm’s complexity:

- Space complexity
- Time complexity

+) Time complexity

The amount of memory needed for a programme to run is referred to as the function input when describing the space complexity of an algorithm.



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

Figure 4: Recursive

For example, recursive method help save space of memory. If I want to sum: 100 + 99 + … + 1, is not hard but if the sum is n + (n-1) + … + 1 will be space consume. But with recursive we can call it again and again to save space.

Another easy example is try to make all operation work with each other with the same data set.

+) Time complexity

Time complexity is the amount of time required to execute an algorithm completely. The same variables that affect space difficulty also affect time complexity.

The best example is choosing the best case algorithm for the scenario. For example, Bubble sort have the best case O(n). So if you sort an array which not large, then Bubble sort will be very fast compare to other sorting algorithm.

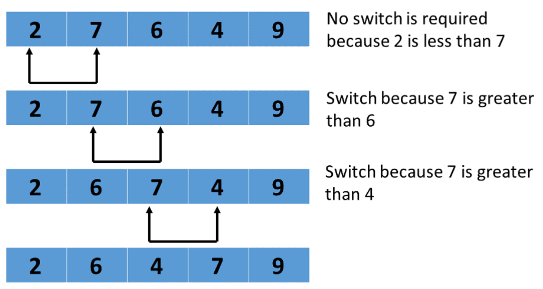

2 7 6 4 9
No switch is required because 2 is less than 7
2 7 6 4 9
Switch because 7 is greater than 6
2 6 7 4 9
Switch because 7 is greater than 4
2 6 4 7 9

Figure 5: Bubble sort

REFERENCES

Kumari, M., 2022. *Efficiency of an Algorithm: Time Complexity & Space Complexity.* [Online]
Available at: https://byjusexamprep.com/efficiency-of-an-algorithm-i
[Accessed 1 September 2022].

Sharma, A., 2021. *Difference between a Singly Linked List and a Doubly Linked List.* [Online]
[Accessed 1st August 2022].

tutorialspoint, 2022. *Data Structures - Asymptotic Analysis.* [Online]
[Accessed 1 September 2022].