Table of Contents

1. Introduction

This assignment is to develop a **Meals Service System** that allows meals ordering procedures to be smoothed. The system consists of Ordering Module, Menu Module and Voucher Module which apply Queue Abstract Data Type, List Abstract Data Type and Set Abstract Data Type respectively.

Module	Abstract Data Type	Person In Charge
1. Ordering	*Queue*	Nicholas Yue Qin Nam
2. Package	***List***	**Chean Kae Lun**
3. Voucher	*Set*	Jessie Liew

In my assignment scope, I am in charge of the **Package Module,** which applies **List ADT**. In my module, it provides the functionality to manage meal packages, which includes:

- *Manage Package Details (Managing meals in a package)*
- *Add Package*
- *Remove Package*
- *Edit Package*
- *Display Sorted Package*
- *Search Package*

The related files are as follows:

- `src\Client\`**`mainClient.java`**
- `src\Entity\`**`Meal.java`**
- `src\Entity\`**`MealPackage.java`**
- `src\SubClient\`**`PackageManager.java`** `(Module's subclient)`
- `src\SortingAlgo\`**`MealPackageSort.java`**
- `src\ADT\`**`ListInterface.java`**
- `src\ADT\`**`LinkedList.java`**

2. Abstract Data Type (ADT) Specification

The reason that I have chosen List Abstract Data Type(ADT) in this assignment is because it is suitable for handling a package list. To clarify,

- In a restaurant, there might be more than one package available for customers to choose from, whereby the packages can be stored in a list *regardless of the concern on rules to add and remove the package from the list*.
- Besides that, the *package list in this system is mainly to display the packages*, without affecting the core functionality which is ordering. Hence, by this reason, I can consider that List ADT is relevant and suitable for packages.
- By utilising List ADT, it enables *easier manipulation of entries in the list* without the restriction of rules or principle whereby direct access to a position in the list is allowed.

Title : List ADT

Description : A linear collection of entries with generic data type<T>, which allows duplicate entries. It allows adding an entry at a specific position or by default at the end of the list. Other operations like removing and searching are allowed with specified positions. The position of a list starts with 1

`add(T newEntry)`

Description : Adds **`newEntry`** to the end of the list

Postcondition : **`newEntry`** has been added to the end of the list

`boolean add(T newEntry, Integer newPosition)`

Description : Add **`newEntry`** to the specified **`newPosition`** within the list.

Position 1 indicates the first entry.

Precondition : **`newPosition`** must between 1 and total entries + 1 inclusively

Postcondition : **`newEntry`** has been added to the specified **newPosition**

Returns : **`true`** if newEntry is successfully added, **`false`** otherwise

`T get(Integer givenPosition)`

Description : Get the entry at specified **`givenPosition`**

Precondition : **`givenPosition`** must between 1 and total entries inclusively

Postcondition : The list remains unchanged

Returns : The entry at **`givenPosition`** if successfully get, **null** otherwise

`T remove(Integer givenPosition)`

Description : Remove the entry at specified **`givenPosition`**

Precondition : **`givenPosition`** must between 1 and total entries inclusively

Postcondition : The entry at **`givenPosition`** has been removed from the list

Returns : The entry that is successfully removed, **`null`** otherwise

`boolean replace(T newEntry, Integer givenPosition)`

Description : Replace the entry at specified **`givenPosition`** with **`newEntry`**

Precondition : List **`isEmpty()`** must be **`false`** and **`givenPosition`** must between 1 and total entries inclusively

Postcondition : The entry at **`givenPosition`** has been replaced by **`newEntry`**

Returns : **`true`** if successfully replace, **`false`** otherwise

`boolean isEmpty()`

Description : To check if the list is empty

Postcondition : The list remains unchanged

Returns : **`true`** if the list is empty, **`false`** otherwise

`boolean isFull()`

Description : To check if the list is full

Postcondition : The list remains unchanged

Returns : **`true`** if the list is full, **`false`** otherwise

`boolean contains(T entry)`

Description : To check if **`entry`** exists in the list

Precondition : List is **`empty()`** must be **`false`**

Postcondition : The list remains unchanged

Returns : **`true`** if **`entry`** is found, **`false`** otherwise

`void sort()`

Description : To sort all the entries in the list in ascending order

Precondition : List **`isEmpty()`** must be **`false`**

Postcondition : The list is sorted in ascending order

`void append(ListInterface newList)`

Description : Append **`newList`** to the current list

Postcondition : **`newList`** is appended to the current list

`clear()`

Description : To remove all entries from the list

Postcondition : The list is empty

3. ADT Implementation

3.1 Overview of ADT

Linked List has been chosen for this List ADT implementation. All the entries are stored in a linear link which starts with the first node and ends with the last node. In each of the nodes, it comprises two parts, that are a data part and a link part.

Data part(data)	Portion of node that stores reference to an entry
Link part(next)	Portion of node that stores reference to another node

In this implementation, three important variables to perform list's operations are declared as follows:

1. `firstNode` : as a reference to the first entry of the list
2. `lastNode` : as a reference to the last entry of the list
3. `numberOfEntries` : to track the total number of entries in the list

All the new entries will be appended to the end of the list without specifying the particular position. Therefore, by using *lastNode*, it increases the efficiency without traversing to the end of the list from the beginning.

The reasons for implementing Linked-List are as follows:

- To have flexible space allocation without emphasizing contiguous memory because all the entries are stored dynamically and being referred by link.
- To add and remove an entry without shifting the existing entries, which avoids overhead.
- To allow the list to grow as large as necessary without the waste of memory since the list will only occupy spaces for existing entries only, when there is a new entry, it will be appended to the link without the need of preallocating space.

3.2 ADT Implementation

3.2.1 Methods As Defined in Inteface

Method : add
Description : Allows adding a new entry into the list with or without specifying position. Without specifying the position, the new entry will be added to the end of the list.

```
1.    @Override
2.     public void add(T newEntry) {
3.         Node newNode = new Node(newEntry);
4.
5.         if (this.isEmpty()) {
6.             firstNode = newNode;
7.         } else {
8.             lastNode.next = newNode;
9.         }
10.        lastNode = newNode;
11.        numberOfEntries++;
12.    }
13.    @Override
14.    public boolean add(T newEntry, Integer newPosition) {
15.        Node newNode = new Node(newEntry);
16.
17.        if (newPosition >= 1 && newPosition <= (numberOfEntries + 1)) {
18.            if (newPosition == 1 && this.isEmpty()) {
19.                newNode.next = firstNode;
20.                firstNode = newNode;
21.                lastNode = newNode;
22.            } else if (newPosition == 1) {
23.                newNode.next = firstNode;
24.                firstNode = newNode;
25.            } else if (newPosition == numberOfEntries + 1) {
26.                lastNode.next = newNode;
27.                lastNode = newNode;
28.            } else {
29.                Node currentNode = firstNode;
30.                for (int currentPosition = 1; currentPosition < newPosition - 1;
    currentPosition++) {
31.                    currentNode = currentNode.next;
32.                }
33.                newNode.next = currentNode.next;
34.                currentNode.next = newNode;
35.            }
36.            numberOfEntries++;
37.            return true;
38.        }
39.        return false;
40.    }
```

Method : get
Description : Obtain an entry from the list by specifying the position.

```
1.    public T get(Integer givenPosition) {
2.         T result = null;
3.         if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
4.             Node currentNode = firstNode;
5.             for (int currentPosition = 1; currentPosition < givenPosition;
    currentPosition++) {
6.                 currentNode = currentNode.next;
7.             }
8.             result = currentNode.data;
9.         }
10.        return result;
11.    }
```

Method : remove
Description : Remove an entry from the list by specifying the position.

```
1.  @Override
2.    public T remove(Integer givenPosition) {
3.        T entryRemoved = null;
4.        //T entryRemoved = (T) "Entry not exist";
5.        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
6.            if (givenPosition == 1) {
7.                entryRemoved = firstNode.data;
8.                firstNode = firstNode.next;
9.            } else {
10.               Node currentNode = firstNode;
11.               for (int currentPosition = 1; currentPosition < givenPosition - 1;
    currentPosition++) {
12.                   currentNode = currentNode.next;
13.               }
14.               entryRemoved = currentNode.next.data;
15.               if (givenPosition == numberOfEntries) {
16.                   lastNode = currentNode;
17.               }
18.               currentNode.next = currentNode.next.next;
19.           }
20.
21.           numberOfEntries--;
22.       }
23.       return entryRemoved;
24.   }
```

Method : replace
Description : Replace an entry with a new entry by specifying the position.

```
1.  @Override
2.    public boolean replace(T newEntry, Integer givenPosition) {
3.
4.        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
5.            Node currentNode = firstNode;
6.            for (int currentPosition = 1; currentPosition < givenPosition;
    currentPosition++) {
7.                currentNode = currentNode.next;
8.            }
9.            currentNode.data = newEntry;
10.           return true;
11.       }
12.       return false;
13.   }
```

Method : isEmpty
Description : To check if the list is empty or not.

```
1.  @Override
2.    public boolean isEmpty() {
3.        return numberOfEntries == 0;
4.    }
```

Method : isFull
Description : To check if the list is fully occupied or not. In this implementation, the list will never be full as it can grow as much as possible when memory capacity is still sufficient.

```
1.  @Override
2.    public boolean isFull() {
3.        //Linked-List will never be full
4.        return false;
5.    }
6.
```

Method : clear
Description: To clear the list into an empty state.

```
1.    @Override
2.    public void clear() {
3.        firstNode = null;
4.        numberOfEntries = 0;
5.        lastNode = null;
6.    }
```

Method : getNumberOfEntries
Description : To get the total number of entries inside the list.

```
1.    @Override
2.    public int getNumberOfEntries() {
3.        return numberOfEntries;
4.    }
```

Method : append
Description : To append another list to the end of the current list.

```
1.   @Override
2.    public void append(ListInterface<T> newList) {
3.        Node newListCurrentNode = ((LinkedList<T>) newList).firstNode;
4.
5.        while (newListCurrentNode != null) {
6.            this.add(newListCurrentNode.data);
7.            newListCurrentNode = newListCurrentNode.next;
8.        }
9.    }
```

Method : contains
Desription : To check if an entry exists in the list or not.

```
1.  @Override
2.    public boolean contains(T entry) {
3.        Node currentNode = firstNode;
4.        while (currentNode != null) {
5.            if (currentNode.data.equals(entry)) {
6.                return true;
7.            }
8.            currentNode = currentNode.next;
9.        }
10.       return false;
11.   }
```

3.2.2 Overriden Java Standard Methods

Method : toString
Description : To return a string that contains all of the elements in a readable manner.

```
1.  @Override
2.   public String toString() {
3.       String strResult = "";
4.       Node currentNode = firstNode;
5.       while (currentNode != null) {
6.           strResult += currentNode.data;
7.           currentNode = currentNode.next;
8.       }
9.
10.      strResult += "\n";
11.      return strResult;
12.  }
```

Method : iterator
Description : To return an iterator of the list for traversing purpose.

```
1.  @Override
2.   public Iterator iterator() {
3.       return new LinkedListIterator(firstNode);
4.   }
5.
6.   private class LinkedListIterator implements Iterator<T> {
7.
8.       private Node currentNode;
9.
10.      private LinkedListIterator(Node firstNode) {
11.          this.currentNode = firstNode;
12.      }
13.
14.      @Override
15.      public boolean hasNext() {
16.          return currentNode != null;
17.      }
18.
19.      @Override
20.      public T next() {
21.          T result = null;
22.          if (currentNode != null) {
23.              result = currentNode.data;
24.              currentNode = currentNode.next;
25.          }
26.
27.          return result;
28.      }
29.  }
```

3.2.3 Inner Class of Linked-list implementation

Class : Node
Description : An object consists of a data part and a link part to form a data structure.

```
1.  private class Node implements Serializable {
2.
3.          private T data;
4.          private Node next;
5.
6.          private Node(T newEntry) {
7.              data = newEntry;
8.              next = null;
9.          }
10.
11.         private Node(T newEntry, Node next) {
12.             data = newEntry;
13.             this.next = next;
14.         }
15.     }
16. }
```

4. Entity Classes

4.1 Entity Class Diagram