BACS2063 Data Structures and Algorithms

ASSIGNMENT 202205

Student Name	:	Chuah Shee Yeap
Student ID	:	22WMR05642
Programme	:	RSF2
Tutorial Group	:	G4
Assignment Title	:	Internship Application System (For Student to Build Resume & Submit Application Only)

Declaration

- I confirm that I have read and complied with all the terms and conditions of Tunku Abdul Rahman University College's plagiarism policy.
- I declare that this assignment is free from all forms of plagiarism and for all intents and purposes is my own properly derived work.

Student's signature

11/9/2022

Date

Table of Contents

1. Introduction

This project will be carried out by myself (individual basis) to build an **Internship Application System** in **console-based** structure that mainly for student usage. This system will be developing functionable modules that involves several functionalities and features specifically for student to go through approximately the whole flowing of internship application submission that represents the real-world internship placement services provided by the relevant school authority.

The **aim** of this system is to do its best to assist students to search through available posted jobs offered by some specific company and so, the students can select one of the jobs in an internship application as well as they can attempt to submit different application details with different job selected according to application that is for the usage of *[company Human Resource (HR) Management – This System Will Not Cover].* In other words, it will primarily take care of front-end design structure using custom-made collection of **Abstract Data Type (ADT)** which has been learned in Data Structure and Algorithm's Lessons to bear up the responsibility in implementing some of the basic methods and attempt to create as much characteristics and operations as possible in compliance with the standard Java Platform Documentation that are able to be applied and resolved the system requirements within the whole implementation of **Internship Application System**.

The system primarily involved 3 modules, which are Student Management Module, Resume Module and Application Module. I will be implementing all of these modules as shown in *Table* **1.1** *and Figure 1.1* below:

Table 1.1 Modules Distribution

Modules	Description
Student Management	- Allow user to sign-up/login an account - Allow user to edit their personal details
Resume Management	- Allow user to build-up a resume - Allow user to view resume details - Allow user to edit resume details - Allow user to clear personal resume
Skill Management	- Allow user to view several available skills - Allow user to add skill to their resume - Allow user to edit skill level
Application Management	- Allow user to pre-view application details - Allow user to submit application - Allow user to clear current application details (no submitted yet)

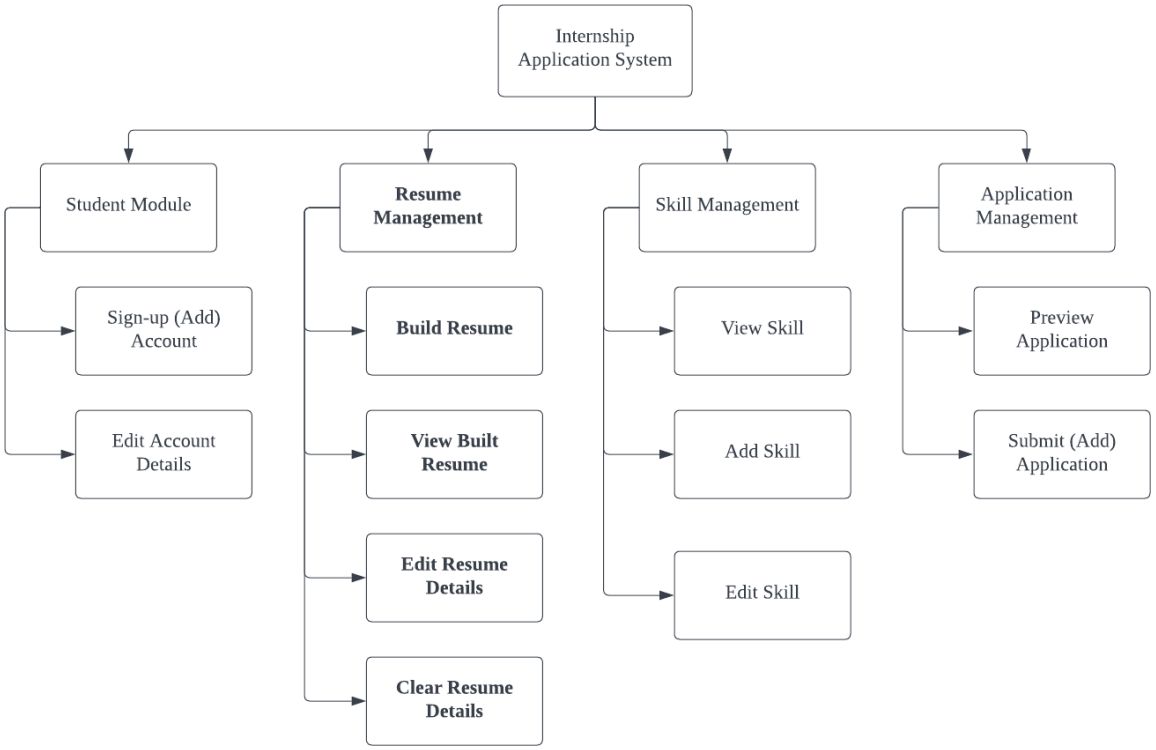


Figure 1.1 Modules Distribution

All of these modules might involve one or more entities engagement: **Student Management Module** will only focus solely to its edit profile features as a user; **Resume Management Module** will involve entities for Resume, Student and Skill; **Skill Management Module** is a necessary component for student to complete their resume building; **Application Management Module** will involve entities for Resume and Job. Everything considered, **Resume Management Module** will be my first and best one module to represent my understanding according to ADT collection being used, which is Doubly Linked List within the implementation of Internship Application System.

Overall saying, the user can attempt to: sign-up their account, view job menu, build resume, add skill highlight to their resume, select desire job to submit application. The system will eliminate the change of duplicate records when the user is conducting skill selection on resume creation. Only a job can be selected to be submitted as a new application together with resume and the user can view the application history via main system menu. There will be sorting and searching features to be applied for job menu and application history for the ease of viewing purpose based on the users' preference.

There will be 5 packages in this project's file structure: ***adt package*** contains the ADT's Interface with its implementer; ***client package*** contains various kinds of entities' maintenance to be processed specifically for an entity; ***dao*** (Data Access Object with Data Write Object) package will be used to retrieving all the data stored in the text files (.txt) as well as writing every single data back to the text files; ***entity package*** will be acted as a domain between system and user; ***utility package*** contains features for printing colors, validating data entry and displaying logo design.

2. Abstract Data Type (ADT) Specification

ADT DLL - Doubly Linked List

A **DLL (Doubly Linked List)** is a variation type of Linked List that consists of a set of sequentially linked records (nodes) and there will be two nodes, which are head node and tail node placed and used within the DDL implementation. As each node has references pointing to the previous and next nodes, therefore, DDL provides navigation flexibility in two-ways directions (forward and backward). Any node adjustments (add or remove) can be done with the help of references changing in terms of node traversal. DDL is suitable to be used in this Internship Application System because it supports fast data retrieval for records (jobs posted, skill selection and elimination of duplicated skill records), the breeze of data searching and sorting behaviors as it implements the way quite similar to Singly Linked List, but allowing for traversing through both ends of the linked list, as well as size of the data records will not be a critical point which it can be changed literally.

boolean add(T newEntry)	
Description:	Add a new entry at tail of the list
Precondition:	New entry cannot be null
Postcondition:	Entry is added at tail of the list
Return:	True if success added at tail

boolean add(int index, T newEntry)	
Description:	Add a new entry with index given to the list
Precondition:	Index must be valid within the list's size. New entry cannot be null.
Postcondition:	Entry is added at specified index of the list
Return:	True if success added to index

boolean addFront(T newEntry)	
Description:	Add a new entry at head of the list
Precondition:	New entry cannot be null
Postcondition:	Entry is added at head of the list
Return:	True if success added at head

boolean addBack(T newEntry)	
Description:	Add a new entry at tail of the list
Precondition:	New entry cannot be null
Postcondition:	Entry is added at tail of the list
Return:	True if success added at tail

boolean addAll(T[] newEntries)	
Description:	Add all new entries one-by-one to the tail of the list
Precondition:	All new entries cannot be null
Postcondition:	All new entries are added at tail of the list
Return:	True if success added all entries at tail

boolean remove(T anEntry)	
Description:	Remove a specified entry from the list
Precondition:	Entry cannot be null. List cannot be empty.
Postcondition:	Entry is removed from the list
Return:	True if success removed specified entry

boolean remove(int index)	
Description:	Remove an element with specified index from the list
Precondition:	Index must be valid within the list's size. List cannot be empty.
Postcondition:	Element is removed from the list
Return:	True if success removed element from specified index

boolean removeFront()	
Description:	Remove an element at head from the list
Precondition:	List cannot be empty
Postcondition:	Element is removed from the list
Return:	True if success removed element from head of the list

boolean removeBack()	
Description:	Remove an element at tail from the list
Precondition:	List cannot be empty
Postcondition:	Element is removed from the list
Return:	True if success removed element from tail of the list

boolean removeAll(T[] entries)	
Description:	Remove all elements one-by-one from the list
Precondition:	List cannot be empty
Postcondition:	All elements are removed from the list
Return:	True if success remove all elements from the list

int size()	
Description:	Get the number of total elements in the list
Precondition:	N/A
Postcondition:	N/A
Return:	Number of elements in the list

boolean isEmpty()	
Description:	Check whether the list is empty
Precondition:	N/A
Postcondition:	N/A
Return:	True if the list is empty

T get(int index)	
Description:	Get element with specified index
Precondition:	Index must be valid within the list's size. List cannot be empty.
Postcondition:	N/A
Return:	Specific element found

T get(T anEnty)	
Description:	Get element with specified entry
Precondition:	Index must be valid within the list's size. List cannot be empty.

Postcondition:	N/A
Return:	Specific element found

boolean set(int index, T anEntry)	
Description:	Set element at specified index with a new entry attributes' value.
Precondition:	Index must be valid within the list's size. Entry cannot be null. List cannot be empty.
Postcondition:	Specific element attributes' value has changed in the list
Return:	True if element attributes' value has changed

T peekFront()	
Description:	Get element at head of the list
Precondition:	List cannot be empty
Postcondition:	N/A
Return:	Element at head

T peekBack()	
Description:	Get element at tail of the list
Precondition:	List cannot be empty
Postcondition:	N/A
Return:	Element at tail

boolean contains(T anEntry)	
Description:	Check whether the element is existed in the list
Precondition:	N/A
Postcondition:	N/A
Return:	True if element is existed in the list

int indexOf(T anEntry)	
Description:	Get index with specific entry given
Precondition:	Entry cannot be null. List cannot be empty.
Postcondition:	N/A
Return:	Index found

int lastIndexOf(T anEntry)	
Description:	Get index of last occurrence with specific entry given
Precondition:	Entry cannot be null. List cannot be empty.
Postcondition:	N/A
Return:	Index of last occurrence

clear()	
Description:	Clear all the elements from the list
Precondition:	N/A
Postcondition:	N/A
Return:	N/A

ListInterface<T> clone()	
Description:	Deep-copying all the elements of current list to a new list
Precondition:	List cannot be empty
Postcondition:	N/A

Return:	A deep copy of all elements within current list

sort(Comparator<T> comparator)	
Description:	Sort the elements in the list in ascending order
Precondition:	N/A
Postcondition:	N/A
Return:	N/A

reverse(Comparator<T> comparator)	
Description:	Sort the elements in the list in descending order
Precondition:	N/A
Postcondition:	N/A
Return:	N/A

Object[] toArray()	
Description:	Convert a list of elements to array
Precondition:	List cannot be empty.
Postcondition:	N/A
Return:	An array of elements converted from the list

Iterator<T> getIterator()	
Description:	Enable elements in the list to be traversed
Precondition:	N/A
Postcondition:	N/A
Return:	List’s iterator

3. ADT Implementation

3.1 Overview of ADT

The only one collection ADT that will be used for this assignment is **Doubly Linked List (DDL)** and I will be the one who take advantage of this ADT to be applied throughout the implementation of **Internship Application System**. A private **Node** inner-class will be created as a data holding block and each node created contains three fundamental components: **data** to store the generic type value, T; **next** to store a link address (pointer) pointing to the next node of current node in the linked list; **prev** to store a link address (pointer) pointing to the previous node from current node in the linked list.

Figure 3.1 Diagram Structure – Doubly Linked List (DDL)

The *List interface* which originally inherits the *Collection interface* will be customized in this assignment to create a new structure of methods and attributes that simulate the *List interface* operation within *ListInterface.java*, instead of inheriting extant, pre-defined interface's properties. There will be 3 member variables used in implementer class, *DoublyLinkedList.java* that implements the *List interface*. The *size* variable indicates the total number of entries within the linked list. The other two variables are typed of ***Node*** which one will be containing the pointer to the **head node** (first node), while another contains the pointer to the **tail node** (last node). Also, there will be a customized ***Iterator class*** that allows nodes (generic type of data contains object's record) searching to be traversed forward or backward within the list and to perform necessary operations based on demands, but conform to Doubly Linked List implementation standard.

3.2 ADT Implementation

ListInterface.java

```
1.  package adt;
2.  import java.util.Comparator;
3.  import java.util.Iterator;
4.
5.  /**
6.   * List Interface for Doubly Linked List
7.   *
8.   * @author Chuah Shee Yeap
9.   * @param <T> type of data (object) to be stored
10. */
11.
12. public interface ListInterface<T> {
13.    /**
14.     * Task: Add a new entry at tail of the list
15.     *
16.     * @param newEntry the object to be added as a new entry
17.     * @return true if success added at tail
18.     */
19.    public boolean add(T newEntry);
20.
21.    /**
22.     * Task: Add a new entry with index given to the list
23.     *
24.     * @param index the position specified by user and make index here
25.     * @param newEntry the object to be added as a new entry
26.     * @return true if success added to index
27.     */
28.    public boolean add(int index, T newEntry);
29.
30.    /**
31.     * Task: Add a new entry at head of the list
32.     *
33.     * @param newEntry the object to be added as a new entry
34.     * @return true if success added at head
35.     */
36.    public boolean addFront(T newEntry);
37.
38.    /**
39.     * Task: Add a new entry at tail of the list
40.     *
41.     * @param newEntry the object to be added as a new entry
42.     * @return true if success added at tail
43.     */
```

```
44.  public boolean addBack(T newEntry);
45.
46.  /**
47.   * Task: Add all new entries one-by-one to the tail of the list
48.   *
49.   * @param newEntries the array object that all to be added
50.   * @return true if success added all at tail
51.   */
52.  public boolean addAll(T[] newEntries);
53.
54.  /**
55.   * Task: Remove a specified entry from the list
56.   *
57.   * @param anEntry the object to be removed
58.   * @return true if success removed specified entry
59.   */
60.  public boolean remove(T anEntry);
61.
62.  /**
63.   * Task: Remove an object with specified index from the list
64.   *
65.   * @param index the position specified by user and make index here
66.   * @return true if success removed entry from specified index
67.   */
68.  public boolean remove(int index);
69.
70.  /**
71.   * Task: Remove object at head from the list
72.   *
73.   * @return true if success removed from head
74.   */
75.  public boolean removeFront();
76.
77.  /**
78.   * Task: Remove object at tail from the list
79.   *
80.   * @return true if success removed from tail
81.   */
82.  public boolean removeBack();
83.
84.  /**
85.   * Task: Remove all entries one-by-one from the list
86.   *
87.   * @param entries the array object that all to be removed
88.   * @return true if success removed all from the list
89.   */
90.  public boolean removeAll(T[] entries);
```

```
91.
92.    /**
93.     * Task: Get the number of total elements in the list
94.     *
95.     * @return number that indicates total elements in the list
96.     */
97.    public int size();
98.
99.    /**
100.       * Task: Check whether the list is empty
101.       *
102.       * @return true if empty list
103.       */
104.      public boolean isEmpty();
105.
106.      /**
107.       * Task: Get element with specified index
108.       *
109.       * @param index the position specified by user and make index here
110.       * @return specific element
111.       */
112.      public T get(int index);
113.
114.      /**
115.       * Task: Get element with specified entry
116.       *
117.       * @param anEntry the object that has a key to get specific record
118.       * @return whole attributes of specific element
119.       */
120.      public T get(T anEntry);
121.
122.      /**
123.       * Task: Set element at specified index with a new entry attributes' value
124.       *
125.       * @param index the position specified by user and make index here
126.       * @param anEntry the object that has changed its attributes' value
127.       * @return true if element attributes' value has changed
128.       */
129.      public boolean set(int index, T anEntry);
130.
131.      /**
132.       * Task: Get element at head of the list
133.       *
134.       * @return element at head of the list
135.       */
136.      public T peekFront();
137.
```

```
138.    /**
139.     * Task: Get element at head of the list
140.     *
141.     * @return element at tail of the list
142.     */
143.    public T peekBack();
144.
145.    /**
146.     * Task: Check whether the element is existed in the list
147.     *
148.     * @param anEntry the object to be checked its existence
149.     * @return true if an element is existed in the list
150.     */
151.    public boolean contains(T anEntry);
152.
153.    /**
154.     * Task: Get index with specific entry given
155.     *
156.     * @param anEntry the object to be traversed to find its location in the list
157.     * @return index
158.     */
159.    public int indexOf(T anEntry);
160.
161.    /**
162.     * Task: Get index of last occurrence with specific entry given
163.     * @param anEntry the object to be traversed to find its last location in the list
164.     * @return index of last occurrence
165.     */
166.    public int lastIndexOf(T anEntry);
167.
168.    /**
169.     * Task: Clear all the elements from the list
170.     */
171.    public void clear();
172.
173.    /**
174.     * Task: Deep-copying all the elements of current list to a new list
175.     *
176.     * @return a new list that contains all elements
177.     */
178.    public ListInterface<T> clone();
179.
180.    /**
181.     * Task: Sort the elements in the list in ascending order
182.     *
183.     * @param comparator specify attributes to be compared
184.     */
```

```
185.     public void sort(Comparator<T> comparator);
186.
187.     /**
188.      * Task: Sort the elements in the list in descending order
189.      *
190.      * @param comparator specify attributes to be compared
191.      */
192.     public void reverse(Comparator<T> comparator);
193.
194.     /**
195.      * Task: Convert a list of elements to array
196.      *
197.      * @return an array of elements from the list
198.      */
199.     public Object[] toArray();
200.
201.     /**
202.      * Task: Enable elements in the list to be traversed
203.      *
204.      * @return list's iterator
205.      */
206.     public Iterator<T> getIterator();
207. }
208.
```

DoublyLinkedList.java

3.2.1 Methods as Defined in Interface

This method will add a new entry at tail of the list. It will check for the null of the new entry and determine how the new entry will be added based on list's empty condition.

```
1.  @Override
2.     public boolean add(T newEntry) {
3.        if (newEntry == null)
4.           throw new RuntimeException("Null Entry");
5.
6.        Node newNode = new Node(newEntry);
7.
8.        if       (isEmpty())       {
9.           directAdd(newNode);
10.       } else {
11.          newNode.prev = tail;
12.          tail.next = newNode;
13.          tail = newNode;
14.       }
```

```
15.       size++;
16.       return true;
17.   }
```

This method will add a new entry at the index location specified by the client program to the list. It will check for the null of the new entry and index range within list's size. It determines how the new entry will be added based on index value condition.

```
1.  @Override
2.    public boolean add(int index, T newEntry) {
3.       if (checkRange(index))
4.          throw new IllegalArgumentException("Invalid Position Entry");
5.       if (newEntry == null)
6.          throw new IllegalArgumentException("Null Entry");
7.
8.       Node newNode = new Node(newEntry);
9.       Node temp = head;
10.
11.      if (index == 0) {
12.         addFront(newEntry);
13.      } else if (index == size) {
14.         addBack(newEntry);
15.      } else {
16.         for (int j = 0; j < index && temp.next != null; j++) {
17.            temp = temp.next;
18.         }
19.         newNode.next = temp;
20.         temp.prev.next = newNode;
21.         newNode.prev = temp.prev;
22.         temp.prev = newNode;
23.         size++;
24.      }
25.      return true;
26.   }
```

This method will add a new entry at head of the list. It will check for the null of the new entry and determine how the new entry will be added based on list's empty condition.

```
1.  @Override
2.    public boolean addFront(T newEntry) {
3.       Node newNode = null;
4.       if (newEntry != null) {
5.          newNode = new Node(newEntry);
6.          if (isEmpty()) {
```

```
7.              directAdd(newNode);
8.           } else {
9.              newNode.next = head;
10.             head.prev = newNode;
11.             head = newNode;
12.          }
13.          size++;
14.          return true;
15.       }
16.       return false;
17.    }
```

This method will add a new entry at tail of the list. It will check for the null of the new entry and determine how the new entry will be added based on list's empty condition.

```
1.  @Override
2.     public boolean addBack(T newEntry) {
3.        Node newNode = null;
4.        if (newEntry != null) {
5.           newNode = new Node(newEntry);
6.           if (isEmpty()) {
7.              directAdd(newNode);
8.           } else {
9.              tail.next = newNode;
10.             newNode.prev = tail;
11.             tail = newNode;
12.          }
13.          size++;
14.          return true;
15.       }
16.       return false;
17.    }
```

This method will add all new entries one follow by one entry to the tail of the list. It will check for the null of all the new entries.

```
1.  @Override
2.     public boolean addAll(T[] newEntries) {
3.        for (T data : newEntries) {
4.           if (data == null)
5.              return false;
6.        add(data);7.
          }
8.        return true;
```

```
9.    }
```

This method will remove a specified entry from the list. It will check for the null of the new entry and the list empty condition. It determines how the entry will be removed based on the location of the node that contains the generic data type.

```
1.  @Override
2.    public boolean remove(T anEntry) {
3.      Node getNode = null;
4.
5.      if (isEmpty())
6.        throw new RuntimeException("Empty List!");
7.      if (anEntry == null)
8.        throw new RuntimeException("Null Entry");
9.
10.     if (head == tail) {
11.       directRemove();
12.       return true;
13.     }
14.     getNode = travel(anEntry);
15.     if (getNode != null) {
16.       if (getNode == head) {
17.         removeFront();
18.       } else if (getNode == tail) {
19.         removeBack();
20.       } else {
21.         getNode.prev.next = getNode.next;
22.         getNode.next.prev = getNode.prev;
23.         size--;
24.       }
25.     }
26.     return true;
27.   }
```

This method will remove an entry according to the given index specified by the client program from the list. It will check for the index range within list's size and the list empty condition. It determines how the entry will be removed based on index value condition.

```
1.  @Override
2.    public boolean remove(int index) {
3.      if (!checkRange(index))
4.        throw new IndexOutOfBoundsException("Invalid Position Entry");
5.      if (isEmpty())
```

```
6.          throw new RuntimeException("Empty List!");
7.
8.       Node temp = head;
9.       if (index == 0) {
10.         removeFront();
11.      } else if (index == size) {
12.         removeBack();
13.      } else {
14.         for (int j = 0; j < index && temp.next != null; j++) {
15.            temp = temp.next;
16.         }
17.         temp.prev.next = temp.next;
18.         temp.next.prev = temp.prev;
19.         size--;
20.      }
21.      return true;
22.   }
```

This method will remove a head element from the list. It will check for the list empty condition and determines how the element will be removed based on null condition for the node next to the head node and start removing around head node.

```
1.  @Override
2.     public boolean removeFront() {
3.        if (isEmpty())
4.           throw new RuntimeException("Empty List!");
5.
6.        if (head.next == null) {
7.           tail = null;
8.        } else {
9.           head.next.prev = null;
10.       }
11.       head = head.next;
12.       size--;
13.       return true;
14.   }
```

This method will remove a tail element from the list. It will check for the list empty condition and determines how the element will be removed based on null condition for the node next to the head node and start removing around tail node.

```
1.  @Override
2.     public boolean removeBack() {
3.        if (isEmpty())
```

```
            throw new RuntimeException("Empty List!");

        if (head.next == null) {
            head = null;
        } else {
            tail.prev.next = null;
        }
        tail = tail.prev;
        size--;
        return true;
    }
```

This method will remove all elements specified in an array, one follow by another from the list. It will check for list empty condition and determines how the element will be removed based on null condition for every entry.

```
@Override
    public boolean removeAll(T[] entries) {
        if (isEmpty())
            throw new RuntimeException("Empty List!");

        for (T data : entries) {
            if (data == null)
                return false;
            remove(data);
        }
        return true;
    }
```

This method will return an integer value for the size of the list.

```
@Override
    public int size() {
        return size;
    }
```

This method will return a Boolean value that indicates the list's empty condition if there is a value of zero size.

```
@Override
    public boolean isEmpty() {
        return size == 0;
    }
```

This method will return a generic type of data according to the index specified by the client program by performing node searching functions.

```
1.  @Override
2.    public T get(int index) {
3.      T data = null;
4.      if (isEmpty())
5.        throw new RuntimeException("Empty List");
6.      if (!checkRange(index))
7.        throw new IndexOutOfBoundsException("Invalid Position");
8.
9.      Node temp = getSpecificNode(index);
10.     data = temp.data;
11.     return data;
12.   }
```

This method will return a generic type of data according to the specified generic type of object, which contains the primary key that overrides the ***equals()*** method in its entity class to perform node searching functions.

```
1.  @Override
2.    public T get(T anEntry) {
3.      if (isEmpty())
4.        throw new RuntimeException("Empty List!");
5.      if (anEntry == null)
6.        throw new RuntimeException("Null Entry");
7.
8.      Node temp = travel(anEntry);
9.      return temp.data;
10.   }
```

This method will make a change to the existing data record reside within the node according to the index and data record to be updated which are specified by the client program.

```
1.  @Override
2.    public boolean set(int index, T anEntry) {
3.      if (isEmpty())
4.        throw new RuntimeException("Empty List");
5.      if (!checkRange(index))
6.        throw new IndexOutOfBoundsException("Invalid Position");
7.      if (anEntry == null)
8.        throw new RuntimeException(("Null Entry"));
```

```
9.
10.       Node setNode = getSpecificNode(index);
11.       setNode.data = anEntry;
12.       return true;
13.   }
```

This method will return a generic type of data for the head node in the list.

```
1.  @Override
2.    public T peekFront() {
3.       if (isEmpty())
4.          throw new RuntimeException("Empty List");
5.       return head.data;
6.    }
```

This method will return a generic type of data for the tail node in the list.

```
1.  @Override
2.    public T peekBack() {
3.       if (isEmpty())
4.          throw new RuntimeException("Empty List");
5.       return tail.data;
6.    }
```

This method will check an entry specified by the client program to know whether it exists in the list.

```
1.  @Override
2.    public boolean contains(T anEntry) {
3.
4.       Node temp = travel(anEntry);
5.       return temp != null;
6.    }
```

This method will return first occurrence of integer value indicating the exact index location of a generic type of data specified by the client program by searching through the matches of data properties overridden by the ***equals()*** method in its entity class.

```
1.  @Override
2.    public int indexOf(T anEntry) {
3.       if (isEmpty())
4.          throw new RuntimeException("Empty List");
5.       if (anEntry == null)
```

```
6.          throw new RuntimeException(("Null Entry"));
7.
8.       int index = 0;
9.       Node temp = head;
10.
11.      // search for first matching value
12.      while (temp != null && !temp.data.equals(anEntry)) {
13.         temp = temp.next;
14.         index++;
15.      }
16.
17.      if (temp == null)     // value not found, return indicator
18.         return -1;
19.      else                  // value found, return index
20.         return index;
21.   }
```

This method will return last occurrence of integer value indicating the exact index location of a generic type of data specified by the client program by searching through the matches of data properties overridden by the ***equals()*** method in its entity class.

```
1.  @Override
2.     public int lastIndexOf(T anEntry) {
3.        if (isEmpty())
4.           throw new RuntimeException("Empty List");
5.        if (anEntry == null)
6.           throw new RuntimeException(("Null Entry"));
7.
8.        int index = size - 1;
9.        Node temp = head;
10.
11.       // search for last matching value
12.       while (temp != null && !temp.data.equals(anEntry)) {
13.          temp = temp.prev;
14.          index--;
15.       }
16.
17.       if (temp == null)     // value not found, return indicator
18.          return -1;
19.       else                  // value found, return index
20.          return index;
21.    }
```

This method will clear all the nodes in the list by setting head and tail node to null indicating no records within the list.

```
1.  @Override
2.     public void clear() {
3.        head = tail = null;
4.        size = 0;
5.     }
```

This method will return a list of cloned generic type of data to the client program by using technique of deep-copying.

```
1.   @Override
2.     public ListInterface<T> clone() {
3.        return deepCopy();
4.     }
```

This method will sort and arrange the node in ascending order in the list according to the attribute-based comparator passed in and customized by the client program.

```
1.  @Override
2.     public void sort(Comparator<T> comparator) {
3.        orderByComparator(comparator);
4.     }
```

This method will sort and arrange the node in descending order in the list according to the attribute-based comparator passed in and customized by the client program.

```
1.  @Override
2.     public void reverse(Comparator<T> comparator) {
3.        descOrderByComparator(comparator);
4.     }
```

This method will return an array of typed **Object** which is converted from the records of the list and it will check for the list's empty condition.

```
1.  @Override
2.     public Object[] toArray() {
3.        int index = 0;
4.        if (isEmpty())
5.           throw new RuntimeException("Empty List");
6.
```

```
7.          Object[] arr = (T[]) new Object[size];
8.          Iterator<T> itr = getIterator();
9.          while (itr.hasNext()) {
10.             arr[index++] = itr.next();
11.         }
12.         return arr;
13.     }
```

3.2.2 Utility methods

This method will keep looping the node next to the previous temporary/head node to find a generic type of data record which is matched with an entry specified by the implementer's method that is taken from the client program. This utility method will be supported for several methods: *remove(T anEntry), get(T anEntry), contains(T anEntry).*

```
1.  private Node travel(T anEntry) {
2.          Node temp = head;
3.          while (temp != null && !temp.data.equals(anEntry))
4.                temp = temp.next;
5.          return temp;
6.      }
```

This method will attempt to simulate the concept of binary search, but this method will only divide the size of the list into half once to determine where the node searching the best to start either from the front node to or back node to the index's designated destination, depending on the index value as compared to the middle index value of the list's size. This utility method will be supported for several methods: *get(int index), set(int index, T anEntry).*

```
1.  private Node getSpecificNode(int indexToFind) {
2.          Node temp = null;
3.          int mid = size / 2;
4.
5.          if (indexToFind < mid)
6.             temp = travelFromFrontToIndex(indexToFind);
7.          else
8.             temp = travelFromBackToIndex(indexToFind);
9.          return temp;
10.     }
```

This method will keep looping to reach the destination of the index which is taken from the **getSpecificNode(int indexToFind)** method that has determined the node searching starting from front node to the index's destination.

```
1.  private Node travelFromFrontToIndex(int indexToFind) {
2.          int firstIndex = 0;
3.          Node temp = head;
4.
5.          while (firstIndex < indexToFind) {
6.             temp = temp.next;
7.             firstIndex++;
8.          }
9.          return temp;
```

```
10.    }
```

This method will keep looping to reach the destination of the index which is taken from the **getSpecificNode(int indexToFind)** method that has determined the node searching starting from back node to the index's destination.

```
1.  private Node travelFromBackToIndex(int indexToFind) {
2.        int lastIndex = size - 1;
3.        Node temp = tail;
4.
5.        while (lastIndex > indexToFind) {
6.           temp = temp.prev;
7.           lastIndex--;
8.        }
9.        return temp;
10.   }
```

This method will perform the adding function by directly assign the new node to the head and tail node variables declared as attributes in this implementer class because there are currently no records/nodes for the list. This utility method will be supported for several methods: *add(T newEntry), addFront(T newEntry), addBack(T newEntry).*

```
1.  private void directAdd(Node newNode) {
2.        head = newNode;
3.        tail = newNode;
4.    }
```

This method will perform the removing function by directly assign the head and tail node variables declared as attributes in this implementer class to the null value because there are no more records/nodes except for the only node which has been assigned as head and tail node previously. This utility method will be supported for method: *remove(T anEntry).*

```
1.  private void directRemove() {
2.        head = null;
3.        tail = null;
4.        size--;
5.    }
```

This method will perform the index range checking as index should be a minimum value of 0 indicates the first location of the node and it should not be exceeding the

size of the list. This utility method will be supported for several methods: *add(int index, T newEntry), remove(int index), get(int index), set(int index, T anEntry).*

```
1.  private boolean checkRange(int index) {
2.          return index >= 0 && index <= size;
3.      }
```

This method will perform a looping function that one-by-one putting the each element of the list into the ***add(T anEntry)*** method until there is no more next record of the node. This utility method will be supported for method: *clone().*

```
1.  private ListInterface<T> deepCopy() {
2.          Node temp = head;
3.          ListInterface<T> newLists = new DoublyLinkedList<>();
4.          while (temp != null) {
5.              newLists.add(temp.data);
6.          temp = temp.next;7.
            }
8.          return newLists;
9.      }
```

This method will simulate **bubble sorting algorithm** that repeatedly using two looping functions, which is a nested loop to traverse and swap the node, two nodes per one time, into the order specified by the client program using a customized comparator in its entity class. It compares each pair of two nodes as to whether which is greater to another and perform swapping accordingly to which the first node is greater than the second node. This utility method will be supported for method: *sort(Comparator<T> comparator).*

```
1.  private void orderByComparator(Comparator<T> comparator) {
2.          Node curNode = head;
3.          while (curNode != null) {
4.              for (Node curNodeNext = curNode.next; curNodeNext != null; curNodeNext
    = curNodeNext.next) {
5.                  if (comparator.compare(curNode.data, curNodeNext.data) > 0) {
6.                  swap(curNode, curNodeNext);7.
                    }
8.              }
9.              curNode = curNode.next;
10.         }
11.     }
```

This method will be the same algorithm same specified in description of ***orderByComparator()*** method. However, it compares each pair of two nodes as to whether which is lower to another and perform swapping accordingly to which the first node is smaller than the second node so that the result will be a descending order of data records reside the node within the list. This utility method will be supported for method: *reverse(Comparator<T> comparator).*

```
1.  private void descOrderByComparator(Comparator<T> comparator) {
2.        Node curNode = head;
3.        while (curNode != null) {
4.          for (Node curNodeNext = curNode.next; curNodeNext != null; curNodeNext
    = curNodeNext.next) {
5.            if (comparator.compare(curNode.data, curNodeNext.data) < 0) {
6.            swap(curNode, curNodeNext);7.
              }
8.          }
9.          curNode = curNode.next;
10.       }
11.   }
```

This method will be used to swap the current node and the node next to the current node so that the data records can be exchanged accordingly to a specific order it should be as used by the ***orderByComparator(Comparator<T> comparator)*** method and ***desOrderByComparator(Comparator<T> comparator)*** method.

```
1.  private void swap(Node curNode, Node curNodeNext) {
2.        T tempData = curNode.data;
3.        curNode.data = curNodeNext.data;
4.        curNodeNext.data = tempData;
5.    }
```

This is a **Node** inner-class defined within the implementer class that simulates the structure of Node that contains data for generic type of data objects, pointer to the next node and pointer to the previous node from the current node.

```
1.  private class Node {
2.        private T data;
3.        private Node next;
4.        private Node prev;
5.
6.        public Node(T data) {
7.          this.data = data;
8.          this.prev  =  null;
9.          this.next = null;
10.       }
```

```
11.
12.     public Node(T data, Node prev, Node next) {
13.         this.data = data;
14.         this.prev = prev;
15.         this.next = next;
16.     }
17. }
```

3.2.3 Overridden Java Standard Methods

This method will return an iterator that allows the client program to easily perform records traversal of the list without index usage.

```
1.  @Override
2.     public Iterator<T> getIterator() {
3.        return new DoublyLinkedListIterator();
4.     }
```

This is a **Iterator** type of inner-class defined within the implementer class to simulate the pre-defined Iterator class by overriding the ***hasNext()*** method that indicates the continuity and existence of the node next to the current node as well as ***next()*** method that return the data record of the node.

```
1.  public class DoublyLinkedListIterator implements Iterator<T> {
2.        Node temp = head;
3.
4.        @Override
5.        public boolean hasNext() {
6.           return temp != null;
7.        }
8.
9.        @Override
10.       public T next() {
11.          T data = null;
12.          if (hasNext()) {
13.             data = temp.data;
14.             temp = temp.next;
15.          }
16.          return data;
17.       }
18.   }
```

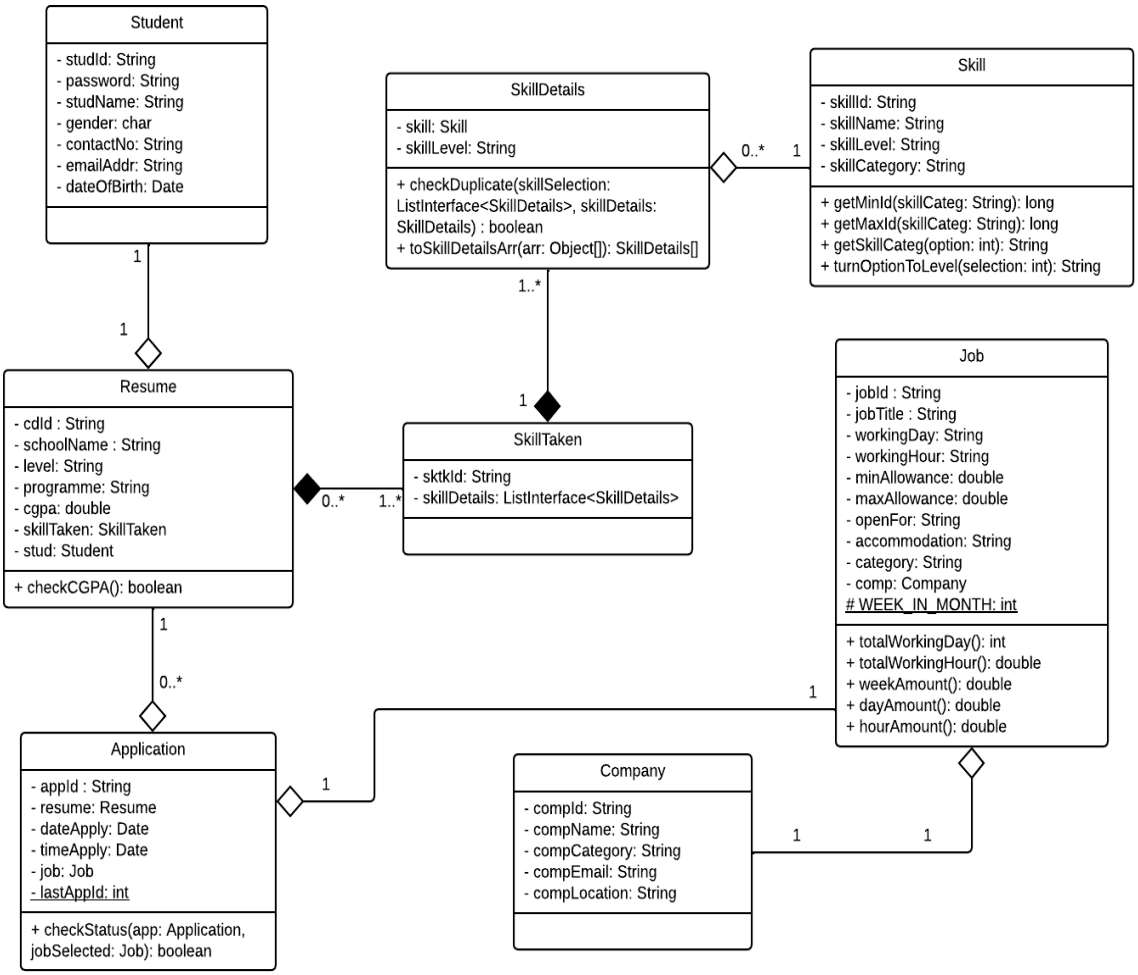
3.2.4 Package, java.util.*, Implementer and variables Declaration

This is about the overall structure of the implementer class and "……" representing the methods as described above.

```
1.  package adt;
2.  import java.util.Comparator;
3.  import java.util.Iterator;
4.
5.  /**
6.   * Implementer class: Doubly Linked List that implements List Interface
7.   *
8.   * @author Chuah Shee Yeap
9.   * @param <T> type of data (object) to be stored
10. */
11.
12. public class DoublyLinkedList<T> implements ListInterface<T> {
13.     private Node head;
14.     private Node tail;
15.     private int size;
16.
17.     public DoublyLinkedList() {
18.         this.head = null;
19.         this.tail = null;
20.         this.size = 0;
21.     }
22.
23. ......
24. ......
25. ......
26.
27. }
```

4. Entity Classes

4.1 Entity Class Diagram



4.2 Entity Class Implementation

Entity Class: *Student.java*

```
1.  package entity;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.  import java.util.Objects;
6.
7.  public class Student {
8.     private String studId;
9.     private String password;
10.    private String studName;
11.    private char gender;
12.    private String contactNo;
13.    private String emailAddr;
14.    private Date dateOfBirth;
15.
16.    public Student() { }
17.
18.    public Student(String studId, String password, String studName, char gender, String
    contactNo, String emailAddr, Date dateOfBirth) {
19.       this.studId = studId;
20.       this.password = password;
21.       this.studName = studName;
22.       this.gender = gender;
23.       this.contactNo = contactNo;
24.       this.emailAddr = emailAddr;
25.       this.dateOfBirth = dateOfBirth;
26.    }
27.
28.    public String getStudId() {
29.       return studId;
30.    }
31.
32.    public void setStudId(String studId) {
33.       this.studId = studId;
34.    }
35.
36.    public String getPassword() { return password; }
37.
38.    public void setPassword(String password) { this.password = password; }
39.
40.    public String getStudName() {
41.       return studName;
42.    }
43.
```

```
44.    public void setStudName(String studName) {
45.        this.studName = studName;
46.    }
47.
48.    public char getGender() { return gender; }
49.
50.    public void setGender(char gender) { this.gender = gender; }
51.
52.    public String getContactNo() {
53.        return contactNo;
54.    }
55.
56.    public void setContactNo(String contactNo) {
57.        this.contactNo = contactNo;
58.    }
59.
60.    public String getEmailAddr() {
61.        return emailAddr;
62.    }
63.
64.    public void setEmailAddr(String emailAddr) {
65.        this.emailAddr = emailAddr;
66.    }
67.
68.    public Date getDateOfBirth() { return dateOfBirth; }
69.
70.    public void setDateOfBirth(Date dateOfBirth) { this.dateOfBirth = dateOfBirth; }
71.
72.    public String contentWriteToFile() {
73.        return this.getStudId() + "/" + this.getPassword() + "/" + this.getStudName() + "/"
    + this.getGender() + "/" + this.getContactNo() + "/" + this.getEmailAddr() + "/"
74.                + new SimpleDateFormat("dd-MM-
    yyyy").format(this.getDateOfBirth()) + "\n";
75.    }
76.
77.    @Override
78.    public boolean equals(Object o) {
79.        if (this == o) return true;
80.        if (!(o instanceof Student)) return false;
81.        Student student = (Student) o;
82.        return studId.equals(student.studId) && password.equals(student.password);
83.    }
84.
85.    @Override
86.    public int hashCode() {
87.        return Objects.hash(studId, password);
88.    }
```

```
89.
90.     @Override
91.     public String toString() {
92.         return String.format("%25s| %-14s | %-14s | %-22s | %3c%-3s | %-15s | %-
    23s | %-
    11s |\n", "", this.getStudId(), this.getPassword(), this.getStudName(), this.getGender()
    , "", this.getContactNo(), this.getEmailAddr(), new SimpleDateFormat("dd-MM-
    yyyy").format(this.getDateOfBirth()));
93.
94.     }
95. }
96.
```

Entity Class: *Company.java*

```
1. package entity;
2.
3. import java.util.Objects;
4.
5.  public class Company {
6.     private String compId;
7.     private String compName;
8.     private String compCategory;
9.     private String compEmail;
10.    private String compLocation;
11.
12.    public Company() { }
13.
14.    public Company(String compId, String compName, String compCategory, String com
    pEmail, String compLocation) {
15.        this.compId = compId;
16.        this.compName = compName;
17.        this.compCategory = compCategory;
18.        this.compEmail = compEmail;
19.        this.compLocation = compLocation;
20.    }
21.
22.    public String getCompId() {
23.        return compId;
24.    }
25.
26.    public void setCompId(String compId) {
27.        this.compId = compId;
28.    }
29.
30.    public String getCompName() {
31.        return compName;
32.    }
33.
34.    public void setCompName(String compName) {
35.        this.compName = compName;
36.    }
37.
38.    public String getCompCategory() {
39.        return compCategory;
40.    }
41.
42.    public void setCompCategory(String compCategory) {
43.        this.compCategory = compCategory;
44.    }
```

```
45.
46.     public String getCompEmail() {
47.         return compEmail;
48.     }
49.
50.     public void setCompEmail(String compEmail) {
51.         this.compEmail = compEmail;
52.     }
53.
54.     public String getCompLocation() {
55.         return compLocation;
56.     }
57.
58.     public void setCompLocation(String compAddr) {
59.         this.compLocation = compAddr;
60.     }
61.
62.     @Override
63.     public boolean equals(Object o) {
64.         if (this == o) return true;
65.         if (!(o instanceof Company)) return false;
66.         Company company = (Company) o;
67.         return compId.equals(company.compId);
68.     }
69.
70.     @Override
71.     public int hashCode() {
72.         return Objects.hash(compId);
73.     }
74.
75.     @Override
76.     public String toString() {
77.         return "Company{" +
78.                 "compId='" + compId + '\'' +
79.                 ", compName='" + compName + '\'' +
80.                 ", compCategory='" + compCategory + '\'' +
81.                 ", compEmail='" + compEmail + '\'' +
82.                 ", compLocation='" + compLocation + '\'' +
83.                 '}';
84.     }
85. }
86.
```

Entity Class: *Job.java*

```
1. package entity;
2.
3.  import java.util.Comparator;
4.  import java.util.Objects;
5.
6.  public class Job {
7.      private String jobId;
8.      private String jobTitle;
9.      private String workingDay;
10.     private String workingHour;
11.     private double minAllowance;
12.     private double maxAllowance;
13.     private String openFor;
14.     private String accommodation;
15.     private String category;
16.     private Company comp;
17.     static final int WEEK_IN_MONTH = 4;
18.     static final int WORK_HOUR = 8;
19.
20.     public Job() { }
21.
22.     public Job(String jobId, String jobTitle, String workingDay, String workingHour, doubl
    e minAllowance, double maxAllowance, String openFor, String accommodation, String
    category, Company comp) {
23.         this.jobId = jobId;
24.         this.jobTitle = jobTitle;
25.         this.workingDay = workingDay;
26.         this.workingHour = workingHour;
27.         this.minAllowance = minAllowance;
28.         this.maxAllowance = maxAllowance;
29.         this.openFor = openFor;
30.         this.accommodation = accommodation;
31.         this.category = category;
32.         this.comp = comp;
33.     }
34.
35.     public String getJobId() {
36.         return jobId;
37.     }
38.
39.     public void setJobId(String jobId) {
40.         this.jobId = jobId;
41.     }
42.
43.     public String getJobTitle() {
```

```
44.     return jobTitle;
45.  }
46.
47.  public void setJobTitle(String jobTitle) {
48.     this.jobTitle = jobTitle;
49.  }
50.
51.  public String getWorkingDay() {
52.     return workingDay;
53.  }
54.
55.  public void setWorkingDay(String workingDay) {
56.     this.workingDay = workingDay;
57.  }
58.
59.  public String getWorkingHour() {
60.     return workingHour;
61.  }
62.
63.  public void setWorkingHour(String workingHour) {
64.     this.workingHour = workingHour;
65.  }
66.
67.  public double getMinAllowance() {
68.     return minAllowance;
69.  }
70.
71.  public void setMinAllowance(double minAllowance) {
72.     this.minAllowance = minAllowance;
73.  }
74.
75.  public double getMaxAllowance() {
76.     return maxAllowance;
77.  }
78.
79.  public void setMaxAllowance(double maxAllowance) {
80.     this.maxAllowance = maxAllowance;
81.  }
82.
83.  public String getOpenFor() {
84.     return openFor;
85.  }
86.
87.  public void setOpenFor(String openFor) {
88.     this.openFor = openFor;
89.  }
90.
```

```
91.    public String getAccommodation() {
92.        return accommodation;
93.    }
94.
95.    public void setAccommodation(String accommodation) {
96.        this.accommodation = accommodation;
97.    }
98.
99.    public String getCategory() { return category; }
100.
101.        public void setCategory(String category) { this.category = category; }
102.
103.        public Company getComp() { return comp; }
104.
105.        public void setComp(Company comp) { this.comp = comp; }
106.
107.        public int totalWorkingDay(int firstDay, int lastDay) {
108.            return lastDay - firstDay + 1;
109.        }
110.
111.        public double totalWorkingHour(double startHour, double endHour) { return en
    dHour - startHour; }
112.
113.        public double weekAmount(double amount, int totalWorkDay) { return amount
    / WEEK_IN_MONTH; }
114.
115.        public double dayAmount(double amount, int totalWorkDay) { return amount / (
    totalWorkDay * WEEK_IN_MONTH); }
116.
117.        public double hourAmount(double amount, int totalWorkDay, double totalWork
    Hour) { return amount / (totalWorkDay * WEEK_IN_MONTH * totalWorkHour); }
118.
119.        public static Comparator<Job> JobIdComparator = (Job j1, Job j2) -> {
120.            return j1.getJobId().compareToIgnoreCase(j2.getJobId());
121.        };
122.
123.        public static Comparator<Job> JobTitleComparator = (Job j1, Job j2) -> {
124.            return j1.getJobTitle().compareToIgnoreCase(j2.getJobTitle());
125.        };
126.
127.        public static Comparator<Job> JobWorkingDayComparator = (Job j1, Job j2) -> {
128.            return j1.getWorkingDay().compareToIgnoreCase(j2.getWorkingDay());
129.        };
130.
131.        public static Comparator<Job> JobWorkingHourComparator = (Job j1, Job j2) -> {
132.            return j1.getWorkingHour().compareToIgnoreCase(j2.getWorkingHour());
133.        };
```

```
134.
135.        public static Comparator<Job> JobMinAllowanceComparator = (Job j1, Job j2) ->
    {
136.            double preciseCheck = j1.getMinAllowance() - j2.getMinAllowance();
137.            //If direct compare and the subtraction will be cast to int (round by), then the
    result incorrect
138.            if (preciseCheck > 0.00001)
139.                return 1;
140.            if (preciseCheck < -0.00001)
141.                return -1;
142.            return 0;
143.        };
144.
145.        public static Comparator<Job> JobMaxAllowanceComparator = (Job j1, Job j2) ->
     {
146.            double preciseCheck = j1.getMaxAllowance() - j2.getMaxAllowance();
147.            //If direct compare and the subtraction will be cast to int (round by), then the
    result incorrect
148.            if (preciseCheck > 0.00001)
149.                return 1;
150.            if (preciseCheck < -0.00001)
151.                return -1;
152.            return 0;
153.        };
154.
155.        public static Comparator<Job> JobOpenForComparator = (Job j1, Job j2) -> {
156.            return j1.getOpenFor().compareToIgnoreCase(j2.getOpenFor());
157.        };
158.
159.        public static Comparator<Job> JobAccommodationComparator = (Job j1, Job j2)
    -> {
160.            return j1.getAccommodation().compareToIgnoreCase(j2.getAccommodation()
    );
161.        };
162.
163.        public static Comparator<Job> JobCategoryComparator = (Job j1, Job j2) -> {
164.            return j1.getCategory().compareToIgnoreCase(j2.getCategory());
165.        };
166.
167.        public static Comparator<Job> JobCompanyComparator = (Job j1, Job j2) -> {
168.            return j1.getComp().getCompName().compareToIgnoreCase(j2.getComp().get
    CompName());
169.        };
170.
171.        @Override
172.        public boolean equals(Object o) {
173.            if (this == o) return true;
```

```
174.            if (!(o instanceof Job)) return false;
175.            Job job = (Job) o;
176.            return jobId.equals(job.jobId);
177.        }
178.
179.        @Override
180.        public int hashCode() {
181.            return Objects.hash(jobId);
182.        }
183.
184.        @Override
185.        public String toString() {
186.            String categAbbr = switch (category) {
187.                case "E-Commerce" -> "EC";
188.                case "Computing-Based" -> "CB";
189.                case "Graphic Design" -> "GP";
190.                default -> "";
191.            };
192.
193.            return String.format("  |  %-6s |  %-24s(%s)  |  %-15s %-16s |  RM%4.0f-
    RM%4.0f  |  %-15s |  %-13s |  %-27s |\n",
194.                    this.getJobId(), this.getJobTitle(), categAbbr, this.getWorkingDay(), this.ge
    tWorkingHour(), this.getMinAllowance(), this.getMaxAllowance(),
195.                    this.getOpenFor(), this.getAccommodation(), this.getComp().getCompNa
    me());
196.
197.        }
198.    }
```

Entity Class: *Skill.java*

```
1. package entity;
2.
3.  import adt.ListInterface;
4.  import client.HostSystem;
5.
6.  import java.util.Comparator;
7.  import java.util.Iterator;
8.  import java.util.Objects;
9.
10. public class Skill {
11.     private String skillId;
12.     private String skillName;
13.     private String category;
14.
15.     public Skill() {
16.     }
17.
18.     public Skill(String skillId) {
19.         this.skillId = skillId;
20.     }
21.
22.     public Skill(String skillId, String skillName) {
23.         this.skillId = skillId;
24.         this.skillName = skillName;
25.     }
26.
27.     public Skill(String skillId, String skillName, String category) {
28.         this.skillId = skillId;
29.         this.skillName = skillName;
30.         this.category = category;
31.     }
32.
33.     public String getSkillId() {
34.         return skillId;
35.     }
36.
37.     public void setSkillId(String skillId) {
38.         this.skillId = skillId;
39.     }
40.
41.     public String getSkillName() {
42.         return skillName;
43.     }
44.
45.     public void setSkillName(String skillName) {
```

```
46.     this.skillName = skillName;
47.  }
48.
49.  public String getCategory() {
50.     return category;
51.  }
52.
53.  public void setCategory(String category) {
54.     this.category = category;
55.  }
56.
57.  public long getMinId(String categ) {
58.     ListInterface<Skill> skillList = HostSystem.skillList;
59.     Iterator<Skill> skillItr = skillList.getIterator();
60.     while (skillItr.hasNext()) {
61.        Skill sk = skillItr.next();
62.        if (Objects.equals(sk.getCategory(), categ)) {
63.           return Long.parseLong(sk.getSkillId().replaceAll("[a-zA-Z]", ""));
64.        }
65.     }
66.     return 0;
67.  }
68.
69.  public long getMaxId(String categ) {
70.     long id = 0;
71.     ListInterface<Skill> skillList = HostSystem.skillList;
72.     Iterator<Skill> skillItr = skillList.getIterator();
73.     while (skillItr.hasNext()) {
74.        Skill sk = skillItr.next();
75.        if (Objects.equals(sk.getCategory(), categ)) {
76.           id = Long.parseLong(sk.getSkillId().replaceAll("[a-zA-Z]", ""));
77.        }
78.     }
79.     return id;
80.  }
81.
82.  public String getSkillCateg(int option) {
83.     String skillCateg = "";
84.     switch (option) {
85.        case 1 -> skillCateg = "Language Skill";
86.        case 2 -> skillCateg = "Computer Skill";
87.        case 3 -> skillCateg = "Personality";
88.     }
89.     return skillCateg;
90.  }
91.
92.  public String turnOptionToLevel(int selection) {
```

```
93.     String level = "";
94.     switch (selection) {
95.       case 1 -> level = "MASTER";
96.       case 2 -> level = "PROFESSIONAL";
97.       case 3 -> level = "INTERMEDIATE";
98.       case 4 -> level = "ADVANCED";
99.       case 5 -> level = "BEGINNER";
100.        }
101.       return level;
102.    }
103.
104.    public Comparator<Skill> SkillIdComparator = (Skill sk1, Skill sk2) -> {
105.       return sk1.getSkillId().compareToIgnoreCase(sk2.getSkillId());
106.    };
107.
108.    public Comparator<Skill> SkillNameComparator = (Skill sk1, Skill sk2) -> {
109.       return sk1.getSkillName().compareToIgnoreCase(sk2.getSkillName());
110.    };
111.
112.    public Comparator<Skill> SkillCategoryComparator = (Skill sk1, Skill sk2) -> {
113.       return sk1.getCategory().compareToIgnoreCase(sk2.getCategory());
114.    };
115.
116.    @Override
117.    public boolean equals(Object o) {
118.       if (this == o) return true;
119.       if (!(o instanceof Skill)) return false;
120.       Skill skill = (Skill) o;
121.       return skillId.equals(skill.skillId);
122.    }
123.
124.    @Override
125.    public int hashCode() {
126.       return Objects.hash(skillId);
127.    }
128.
129.    @Override
130.    public String toString() {
131.       return "Skill{" +
132.             "skillId='" + skillId + '\'' +
133.             ", skillName='" + skillName + '\'' +
134.             ", category='" + category + '\'' +
135.             '}';
136.    }
137. }
```

Entity Class: *SkillDetails.java*

```
1. package entity;
2.
3. import adt.ListInterface;
4.
5.  import java.util.Comparator;
6.  import java.util.Objects;
7.
8.  public class SkillDetails {
9.     private String skdId;
10.    private Skill skill;
11.    private String level;
12.
13.    public SkillDetails() { }
14.
15.    public SkillDetails(String skdId, Skill skill) {
16.       this.skdId = skdId;
17.       this.skill = skill;
18.    }
19.
20.    public SkillDetails(String skdId, Skill skill, String level) {
21.       this.skdId = skdId;
22.       this.skill = skill;
23.       this.level = level;
24.    }
25.
26.    public String getSkdId() {
27.       return skdId;
28.    }
29.
30.    public void setSkdId(String skdId) {
31.       this.skdId = skdId;
32.    }
33.
34.    public Skill getSkill() {
35.       return skill;
36.    }
37.
38.    public void setSkill(Skill skill) {
39.       this.skill = skill;
40.    }
41.
42.    public String getLevel() {
43.       return level;
44.    }
45.
```

```
46.    public void setLevel(String level) {
47.        this.level = level;
48.    }
49.
50.    public Comparator<SkillDetails> SkillDetailsSkillIdComparator = (SkillDetails skd1, Ski
       llDetails skd2) -> {
51.        return skd1.getSkill().getSkillId().compareToIgnoreCase(skd2.getSkill().getSkillId())
       ;
52.    };
53.
54.    public boolean checkDuplicate(ListInterface<SkillDetails> skillSelection, SkillDetails s
       killDetails) {
55.        return skillSelection.contains(skillDetails);
56.    }
57.
58.    public SkillDetails[] toSkillDetailsArr(Object[] arr) {
59.        SkillDetails[] skdArr = new SkillDetails[arr.length];
60.        for (int i = 0; i < arr.length; i++) {
61.            skdArr[i] = (SkillDetails) arr[i];
62.        }
63.        return skdArr;
64.    }
65.
66.    public String contentWriteToFile() {
67.        return this.getSkdId() + "/"  + this.getSkill().getSkillId() + "/" + this.getLevel() + "\n"
       ;
68.    }
69.
70.    @Override
71.    public boolean equals(Object o) {
72.        if (this == o) return true;
73.        if (!(o instanceof SkillDetails)) return false;
74.        SkillDetails that = (SkillDetails) o;
75.        return skdId.equals(that.skdId) && skill.equals(that.skill);
76.    }
77.
78.    @Override
79.    public int hashCode() {
80.        return Objects.hash(skdId, skill);
81.    }
82.
83.    @Override
84.    public String toString() {
85.        return "SkillDetails{" +
86.                "skdId='" + skdId + '\'' +
87.                ", skill=" + skill +
88.                ", level='" + level + '\'' +
```

```
89.             '}';
90.     }
91. }
```

Entity Class: *SkillTaken.java*

```
1. package entity;
2.
3. import adt.ListInterface;
4.
5. import java.util.Objects;
6.
7.  public class SkillTaken {
8.     private String skTkId;
9.     private ListInterface<SkillDetails> skillDetails;
10.
11.    public SkillTaken() {  }
12.    public SkillTaken(String skTkId, ListInterface<SkillDetails> skillDetails) {
13.       this.skTkId = skTkId;
14.       this.skillDetails = skillDetails;
15.    }
16.
17.    public String getSkTkId() {
18.       return skTkId;
19.    }
20.
21.    public void setSkTkId(String skTkId) {
22.       this.skTkId = skTkId;
23.    }
24.
25.    public ListInterface<SkillDetails> getSkillDetails() {
26.       return skillDetails;
27.    }
28.
29.    public void setSkillDetails(ListInterface<SkillDetails> skillDetails) {
30.       this.skillDetails = skillDetails;
31.    }
32.
33.    public String contentWriteToFile() {
34.       return this.getSkTkId() + "/";
35.    }
36.
37.    @Override
38.    public boolean equals(Object o) {
39.       if (this == o) return true;
40.       if (!(o instanceof SkillTaken)) return false;
41.       SkillTaken that = (SkillTaken) o;
42.       return skTkId.equals(that.skTkId);
43.    }
44.
45.    @Override
```

```
46.    public int hashCode() {
47.        return Objects.hash(skTkId);
48.    }
49.
50.    @Override
51.    public String toString() {
52.        return "SkillTaken{" +
53.                "skTkId='" + skTkId + '\'' +
54.                ", skillDetails=" + skillDetails +
55.                '}';
56.    }
57. }
```

Entity Class: *Resume.java*

```
1. package entity;
2.
3.  import java.util.Comparator;
4.  import java.util.Objects;
5.
6.  public class Resume {
7.      private String cvId;
8.      private String schoolName;
9.      private String level;
10.     private String programme;
11.     private double cgpa;
12.     private SkillTaken skillTaken;
13.     private Student stud;
14.
15.     public Resume() { }
16.
17.     public Resume(String cvId, String schoolName, String level, String programme, doubl
    e cgpa, SkillTaken skillTaken, Student stud) {
18.         this.cvId = cvId;
19.         this.schoolName = schoolName;
20.         this.level = level;
21.         this.programme = programme;
22.         this.cgpa = cgpa;
23.         this.skillTaken = skillTaken;
24.         this.stud = stud;
25.     }
26.
27.     public String getCvId() {
28.         return cvId;
29.     }
30.
31.     public void setCvId(String cvId) {
32.         this.cvId = cvId;
33.     }
34.
35.     public String getSchoolName() {
36.         return schoolName;
37.     }
38.
39.     public void setSchoolName(String schoolName) {
40.         this.schoolName = schoolName;
41.     }
42.
43.     public String getLevel() {
44.         return level;
```

```
45.    }
46.
47.    public void setLevel(String level) {
48.        this.level = level;
49.    }
50.
51.    public String getProgramme() {
52.        return programme;
53.    }
54.
55.    public void setProgramme(String programme) {
56.        this.programme = programme;
57.    }
58.
59.    public double getCgpa() { return cgpa; }
60.
61.    public void setCgpa(double cgpa) {
62.        this.cgpa = cgpa;
63.    }
64.
65.    public SkillTaken getSkillTaken() { return skillTaken; }
66.
67.    public void setSkillTaken(SkillTaken skillTaken) { this.skillTaken = skillTaken; }
68.
69.    public Student getStud() { return stud; }
70.
71.    public void setStud(Student stud) { this.stud = stud; }
72.
73.    public boolean checkCGPA(double cgpa) {
74.        return cgpa >= 0.0 && cgpa <= 4.0;
75.    }
76.
77.    public Comparator<Resume> ResumeIdComparator = (Resume resume1, Resume re
     sume2) -> {
78.    return resume1.getCvId().compareToIgnoreCase(resume2.getCvId());
79.    };
80.
81.    public String contentWriteToFile() {
82.        return String.format("%s/%s/%s/%s/%.4f/%s/%s\n", this.getCvId(), this.getSchool
     Name(), this.getLevel(), this.getProgramme(), this.getCgpa(), this.getSkillTaken().getSk
     TkId(), this.getStud().getStudId());
83.    }
84.
85.    @Override
86.    public boolean equals(Object o) {
87.        if (this == o) return true;
88.        if (!(o instanceof Resume)) return false;
```

```
89.         Resume resume = (Resume) o;
90.         return cvId.equals(resume.cvId);
91.     }
92.
93.     @Override
94.     public int hashCode() {
95.         return Objects.hash(cvId);
96.     }
97.
98.     @Override
99.     public String toString() {
100.            return "Resume{" +
101.                    "cvId='" + cvId + '\'' +
102.                    ", schoolName='" + schoolName + '\'' +
103.                    ", level='" + level + '\'' +
104.                    ", programme='" + programme + '\'' +
105.                    ", cgpa=" + cgpa +
106.                    ", skillTaken=" + skillTaken +
107.                    ", stud=" + stud +
108.                    ", ResumeIdComparator=" + ResumeIdComparator +
109.                    '}';
110.        }
111.    }
```

Entity Class: *Application.java*

```
1. package entity;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Comparator;
5.  import java.util.Date;
6.  import java.util.Objects;
7.
8.  public class Application {
9.     private String appId;
10.    private Resume resume;
11.    private Date dateApply;
12.    private Date timeApply;
13.    private String status;
14.    private Job job;
15.    private static int lastAppID;
16.
17.    public Application() { }
18.
19.    public Application(String appId, Resume resume, Date dateApply, Date timeApply, S
    tring status, Job job) {
20.       this.appId = appId;
21.       this.resume = resume;
22.       this.dateApply = dateApply;
23.       this.timeApply = timeApply;
24.       this.status = status;
25.       this.job = job;
26.    }
27.
28.    public String getAppId() {
29.       return appId;
30.    }
31.
32.    public void setAppId(String appId) {
33.       this.appId = appId;
34.    }
35.
36.    public Resume getResume() {
37.       return resume;
38.    }
39.
40.    public void setResume(Resume resume) {
41.       this.resume = resume;
42.    }
43.
44.    public Date getDateApply() {
```

```
45.         return dateApply;
46.     }
47.
48.     public void setDateApply(Date dateApply) {
49.         this.dateApply = dateApply;
50.     }
51.
52.     public Date getTimeApply() {
53.         return timeApply;
54.     }
55.
56.     public void setTimeApply(Date timeApply) {
57.         this.timeApply = timeApply;
58.     }
59.
60.     public String getStatus() {
61.         return status;
62.     }
63.
64.     public void setStatus(String status) {
65.         this.status = status;
66.     }
67.
68.     public Job getJob() {
69.         return job;
70.     }
71.
72.     public void setJob(Job job) {
73.         this.job = job;
74.     }
75.
76.     public static int getLastAppID() {
77.         return lastAppID;
78.     }
79.
80.     public static void setLastAppID(int lastAppID) {
81.         Application.lastAppID = lastAppID;
82.     }
83.
84.     public static Comparator<Application> ApplicationIdComparator = (Application app1
    , Application app2) -> {
85.     return app1.getAppId().compareToIgnoreCase(app2.getAppId());
86.     };
87.
88.     public static Comparator<Application> ApplicationDateComparator = Comparator.c
    omparing(Application::getDateApply);
89.
```

```
90.    public static Comparator<Application> ApplicationTimeComparator = Comparator.c
    omparing(Application::getTimeApply);
91.
92.    public static Comparator<Application> ApplicationStatusComparator = (Application
    app1, Application app2) -> {
93.    return app1.getStatus().compareToIgnoreCase(app2.getStatus());
94.    };
95.
96.    public static Comparator<Application> ApplicationJobTitleComparator = (Applicatio
    n app1, Application app2) -> {
97.        return app1.getJob().getJobTitle().compareToIgnoreCase(app2.getJob().getJobTitl
    e());
98.    };
99.
100.        public static Comparator<Application> ApplicationJobMinAllowanceComparator
    = (Application app1, Application app2) -> {
101.            double preciseCheck = app1.getJob().getMinAllowance() - app2.getJob().getM
    inAllowance();
102.            //If direct compare and the subtraction will be cast to int (round by), then the
    result incorrect
103.            if (preciseCheck > 0.00001)
104.                return 1;
105.            if (preciseCheck < -0.00001)
106.                return -1;
107.            return 0;
108.        };
109.
110.        public static Comparator<Application> ApplicationJobMaxAllowanceComparator
     = (Application app1, Application app2) -> {
111.            double preciseCheck = app1.getJob().getMaxAllowance() - app2.getJob().getM
    axAllowance();
112.            //If direct compare and the subtraction will be cast to int (round by), then the
    result incorrect
113.            if (preciseCheck > 0.00001)
114.                return 1;
115.            if (preciseCheck < -0.00001)
116.                return -1;
117.            return 0;
118.        };
119.
120.        public boolean checkStatus(Application app, Job jobSelected) {
121.            return app.getJob().equals(jobSelected) && Objects.equals(app.getStatus(), "
    Rejected");
122.        }
123.
124.        public String contentWriteToFile() {
```

```
125.            return this.getAppId() + "/" + this.getResume().getCvId() + "/" + new SimpleDa
    teFormat("dd-MM-yyyy").format(this.getDateApply()) + "/" +
126.                    new SimpleDateFormat("HH:mm:ss").format(this.getTimeApply()) + "/" +
    this.getStatus() + "/" + this.getJob().getJobId() + "\n";
127.        }
128.
129.        @Override
130.        public boolean equals(Object o) {
131.            if (this == o) return true;
132.            if (!(o instanceof Application)) return false;
133.            Application that = (Application) o;
134.            return appId.equals(that.appId);
135.        }
136.
137.        @Override
138.        public int hashCode() {
139.            return Objects.hash(appId);
140.        }
141.
142.        @Override
143.        public String toString() {
144.            return String.format("%30s|| %-16s || %s %13s || %-11s || %-
    25sRM%4.0f-RM%4.0f ||\n",
145.                    "", this.getAppId(), new SimpleDateFormat("dd-MM-
    yyyy").format(this.getDateApply()),
146.                    new SimpleDateFormat("HH:mm:ss").format(this.getTimeApply()), this.ge
    tStatus(),
147.                    this.getJob().getJobTitle(), this.getJob().getMinAllowance(), this.getJob().g
    etMaxAllowance());
148.        }
149.    }
```

5. Client Program

5.1 ADT Usage and Justification

Doubly Linked List (DLL) is used in this Internship Application System because it allows for searching specific data records in both directions of forward and backward. Therefore, it can somehow assist in saving the searching time by traversing the linked list in a specific direction that the best to find out the anticipated data records which increases the overall system efficiency and performance.

ADT in Entity Class

SkillTaken.java

There will be several skills to be owned by student when he or she is building a resume for later application used. There will be several available skills provided for students to select from and put into resume and every student has to define their skill level. It means that the one resume might contains multiple skills that has integrated within, while any one of the skills elected by the student will also possible to be used by other student who own the same skill as the only difference is about the skill level.

Since it implies a many-to-many relationship between entity resume and entity skill, an associative entity, which is SkillDetails is created to resolve the record mapping as to keep the record structure organized in file and also defines a key for easy mapping process for getting the specific records as well as avoid massive duplication of records. This SkillDetails will be declared as a ListInterface that is typed of DDL and placed within entity SkillTaken which this entity acts as a data holding block for indicating a list of skills selected by a specific students is belongs to which resume.

```
.....
public class SkillTaken {
    private String skTkId;
    private ListInterface<SkillDetails> skillDetails;

    public SkillTaken() {  }

    public SkillTaken(String skTkId, ListInterface<SkillDetails> skillDetails) {
        this.skTkId = skTkId;
        this.skillDetails = skillDetails;
    }
.....
}
```

(entity.SkillTaken.java: Line7-Line15 shows the declaration and initialization of SkillDetails into a ListInterface of typed Doubly Linked List.)

ADT in dao (Data Access Object)

Code below for *public ListInterface<Student> getStudDetails()* and public *ListInterface<Skill> getSkillDetails()* shows how I read the data from file, line-by-line to get all the data attributes and using adt add(T anEntry) method to add a record of type Student into ListInterface and continue the process until there is no more new line for reading the data records. Same as to apply for other reading files functions is to get all the records and find and modify the matched records based on all records obtained from any of the methods inside *entity.DataAccess.java.*

```
public ListInterface<Student> getStudDetails() {
      ListInterface<Student> studDetails = new DoublyLinkedList<>();
      int index = 0;
      try {
         Scanner fileScan = new Scanner(new File("Student.txt"));
         while (fileScan.hasNextLine()) {
            Student getStud = new Student();
            Scanner lineScan = new Scanner(fileScan.nextLine()).useDelimiter("/");
            while (lineScan.hasNext()) {
               String data = lineScan.next();
               switch (index) {
                  case 0 -> getStud.setStudId(data);
                  case 1 -> getStud.setPassword(data);
                  case 2 -> getStud.setStudName(data);
                  case 3 -> getStud.setGender((data).charAt(0));
                  case 4 -> getStud.setContactNo(data);
                  case 5 -> getStud.setEmailAddr(data);
                  case 6 -> getStud.setDateOfBirth(new SimpleDateFormat("dd-MM-
yyyy").parse(data));
               }
               index++;
            }
            index = 0;
            studDetails.add(getStud);
            lineScan.close();
         }
         fileScan.close();
      } catch (FileNotFoundException | ParseException e) {
         e.printStackTrace();
      }
      return studDetails;
   }
```

(dao.DataAccess.java: Line17-Line47)

(Method name: public ListInterface<Student> getStudDetails()*))*

```
public ListInterface<Skill> getSkillDetails() {
      ListInterface<Skill> skillDetails = new DoublyLinkedList<>();
      try {
         String line;
         BufferedReader readFile = new BufferedReader(new FileReader("Skill.txt"));
```

```
        while ((line = readFile.readLine()) != null) {
            String[] data = line.split("/");
            skillDetails.add(new Skill(data[0], data[1], data[2]));
        }
        readFile.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return skillDetails;
}
```

(dao.DataAccess.java: Line65-Line79))

(Method Name: public ListInterfac<Skill> getSkillDetails()*)*

[Both using add(T anEntry) method]

ADT in client maintenance

client.JobMaintenance.java

The student can view the job menu especially when they want to sort the job lists based on attributes desire or find some specific jobs according to keywords inputted. Code below showing the structure DDL sorting as to whether sort by ascending or descending on different data attributes that is also selected by student to have a view on the job list.

```
        ……
        case 1 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobIdComparator);
            else
                tempSortedJobList.reverse(Job.JobIdComparator);
            printJobList(tempSortedJobList);
        }
        case 2 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobTitleComparator);
            else
                tempSortedJobList.reverse(Job.JobTitleComparator);
            printJobList(tempSortedJobList);
        }
        case 3 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobWorkingDayComparator);
            else
                tempSortedJobList.reverse(Job.JobWorkingDayComparator);
            printJobList(tempSortedJobList);
        }
        case 4 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobWorkingHourComparator);
            else
```

```
                tempSortedJobList.reverse(Job.JobWorkingHourComparator);
            printJobList(tempSortedJobList);
        }
        case 5 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobMinAllowanceComparator);
            else
                tempSortedJobList.reverse(Job.JobMinAllowanceComparator);
            printJobList(tempSortedJobList);
        }
        case 6 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobMaxAllowanceComparator);
            else
                tempSortedJobList.reverse(Job.JobMaxAllowanceComparator);
            printJobList(tempSortedJobList);
        }
        case 7 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobOpenForComparator);
            else
                tempSortedJobList.reverse(Job.JobOpenForComparator);
            printJobList(tempSortedJobList);
        }
        case 8 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobAccommodationComparator);
            else
                tempSortedJobList.reverse(Job.JobAccommodationComparator);
            printJobList(tempSortedJobList);
        }
        case 9 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobCategoryComparator);
            else
                tempSortedJobList.reverse(Job.JobCategoryComparator);
            printJobList(tempSortedJobList);
        }
        case 10 -> {
            if (jobOrder() == 1)
                tempSortedJobList.sort(Job.JobCompanyComparator);
            else
                tempSortedJobList.reverse(Job.JobCompanyComparator);
            printJobList(tempSortedJobList);
        }
         .....
```

(client.JobMaintenance.java Line80-Line149 shows the ascending sorting and descending sorting that selected by the student by using ListInterface.sort(Comparator<T> comparator) or ListInterface.reverse(Comparator<T> comparator))

(Method name: public void sortedJobMenu())

The student can search a specific job or get a list of job if there are multiple records matches with the *ListInterface<Job> jobList* which contains a list of available jobs for student to search from. Code below show the structure how a job's iterator can be obtained by using *ListInterface.getIterator()* method so that it allows for records traversal and if the job's data field is matched what user is inputted, then that job will be added into a DDL for later records printing functions.

```
        ……
       switch (searchType) {
         case "Job ID" -> {
            System.out.printf("%60s[ Example: J1002, J1010 ]\n", "");
            System.out.printf("%60s==========================================================
=====\n", "");
            System.out.printf("%60sPlease Key-in search word ---> ", "");
            str = scanner.nextLine().toLowerCase();
            jobItr = jobList.getIterator();
            while (jobItr.hasNext()) {
               Job job = jobItr.next();
               if (job.getJobId().toLowerCase().contains(str))
                  jobsSearch.add(job);
            }
         }
        ……
```

(client.JobMaintenance.java Line327-Line440 shows the fields searching process using ListInterface.getIterator())

(Method name: public void fieldSearch)

The student will make a job selection and the number entered must be within the size of DDL by using *ListInterface.size()* method to pass on the within-range condition. The selection entered by student will be minus 1 and put into *ListInterface.get(int index)* method to indicate the exact index location of the DDL so to obtain that specific record.

```
 ……
        printJobList(jobList);
         int selection = DataValidation.intValidation("%69sPlease Enter Selection (in number) ---> ");
         if (selection >= 1 && selection <= jobList.size()) {
            for (int j = 0; j < jobList.size(); j++) {
               if (j == selection - 1) {
                  jobSelect = jobList.get(j);
               }
            }
            return jobSelect;
         }
 ……
```

(client.JobMaintenance.java Line502-Line511 shows the get(int index) method)

(Method name: public Job jobSelection())

client.ResumeMaintenance.java

If the student is agreed to remove current or previously built resume, then *ListInterface.remove(T anEntry)* method is used to remove the specific resume that is to be compared with all the other resume records and the correct data record removal will be done.

```
……
            if (exist) {
                boolean toRemove = removeResumeDetails(resume);
                if(toRemove) {
                    ListInterface<Application> appList = dataRetrieve.getAllAppDetails();
                    Iterator<Application> appItr = appList.getIterator();
                    while(appItr.hasNext()) {
                        Application app = appItr.next();
                        if(app.getResume().equals(resume))
                            appList.remove(app);
                    }
                    dataWrite.applicationWriteToFile(appList);

                    ListInterface<Resume> resumeList = dataRetrieve.getResumeDetails();
                    resumeList.remove(resume);
                    dataWrite.resumeWriteToFile(resumeList);
                    resume  = null;
                }
            }
……
```

(client.ResumeMaintenance.java Line83-Line99 shows the remove(T anEntry) method)

(Method name: public Resume resumeMenu(Student userLogin, Resume resume)

A resume to be edited will be passed to this method and *ListInterface.indexOf(T anEntry)* is used to find the exact index location of this edited resume details can be set to the found index location and replace with the old resume details by using *ListInterface.set(int index, T anEntry)* method.

```
public ListInterface<Resume> editResumeDetails(Resume resume) {
    ListInterface<Resume> resumeList = dataRetrieve.getResumeDetails();
    int index = resumeList.indexOf(resume);
    resumeList.set(index, resume);
    return resumeList;
}
```

(client.ResumeMaintenance.java Line301-Line306 shows the indexOf(T anEntry) method and set(int index, T anEntry))

(Method name: public ListInterface<Resume> editResumeDetails(Resume resume))

client.SkillDetailsMaintenance.java

Skill selected by a student cannot be duplicated because each skill will only have a single meaning of its level, instead of a skill with multiples level indication. ListInterface.contains(T anEntry) can help to check for the existence of data record and impede the adding skill process if it does contain.

```
public boolean checkDuplicate(ListInterface<SkillDetails> skillSelection, SkillDetails skillDetails) {
    return skillSelection.contains(skillDetails);
  }
```

(client.SkillDetailsMaintenance.java Line54-Line56 shows the contains(T anEntry) method)

(Method name: public boolean checkDuplicate(ListInterface<SkillDetails> skillSelection, SkillDetails skillDetails))

There will be many skills have been selected by a student and to be added to the text file. Before adding all the skills with level, ListInterface.isEmpty() is checked to ensure no empty or null value will be passed to adt. As addAll(T[] entries) has been implemented in DDL and the argument is an array, therefore current list of selected jobs will be first converted to object array using ListInterface.toArray() method and perform relevant array copying technique to enable addAll(T[] entries) successful completed and added to the list.

```
public void updateSkill(ListInterface<SkillDetails> skillSelection, int option) {
    ListInterface<SkillDetails> tempSkillDetails = skillSelection(option);
    Iterator<SkillDetails> skillDetailsItr = tempSkillDetails.getIterator();

    while (skillDetailsItr.hasNext()) {
      SkillDetails temp1 = skillDetailsItr.next();
      if(temp1.checkDuplicate(skillSelection, temp1)) {
        tempSkillDetails.remove(temp1);
      }
    }

   if(!tempSkillDetails.isEmpty()) {
      Object[] arr = tempSkillDetails.toArray();
      skillSelection.addAll(new SkillDetails().toSkillDetailsArr(arr));
    }
  }
```

(client.SkillDetailsMaintenance.java Line81-Line96 shows the isEmpty() method, toArray() method, addAll(T[] entries) method)

(Method name: public void updateSkill(ListInterface<SkillDetails> skillSelection, int option))

ListInterface.get(T anEntry) method is used here to obtain an object that has all value placed within. Even an object with a key which is an id here that is enough to find that specific skill selected.

```
{……
            Skill skillSelect = new Skill();
             String skillCateg = skillSelect.getSkillCateg(option);
             printSkillCategory(option);
             String skillId = DataValidation.strValidation("%69sPlease Enter Skill ID ---> ", 7);

             long skillIdNumberOnly = Long.parseLong(skillId.replaceAll("[a-zA-Z]", ""));
             if (skillIdNumberOnly >= skillSelect.getMinId(skillCateg) && skillIdNumberOnly <= skillSel
ect.getMaxId(skillCateg)) {
                skillSelect = skillList.get(new Skill(skillId));
                skillDetails = new SkillDetails("skd" + student.getStudId(), skillSelect);
```

```
.....
}
```

(client.SkillDetailsMaintenance.java Line108-Line116 shows get(T anEntry) method

(Method name: public void ListInterface<SkillDetails> skillSelection(int option))