ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	9/8/2023	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyễn Hải Việt	**Student ID**	GCD210136
Class	GCD1101	**Assessor name**	Phạm Thành Sơn
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.			
		Student's signature	

Grading grid

P1	P2	P3	M1	M2	M3	D1	D2

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

Table of Contents

I. Create a design specification for data structures explaining the valid operations that can be carried out on the structures.(P1)

A. Definition of ADT

An object's behaviour may be described by a collection of values and a set of operations, and this behaviour is known as an abstract data type (ADT). The definition of ADT merely specifies the actions that must be taken, not how they must be carried out. It is unclear what algorithms will be utilised to carry out the operations and how the data will be organised in memory. Because it provides an implementation-independent perspective, it is dubbed "abstract."

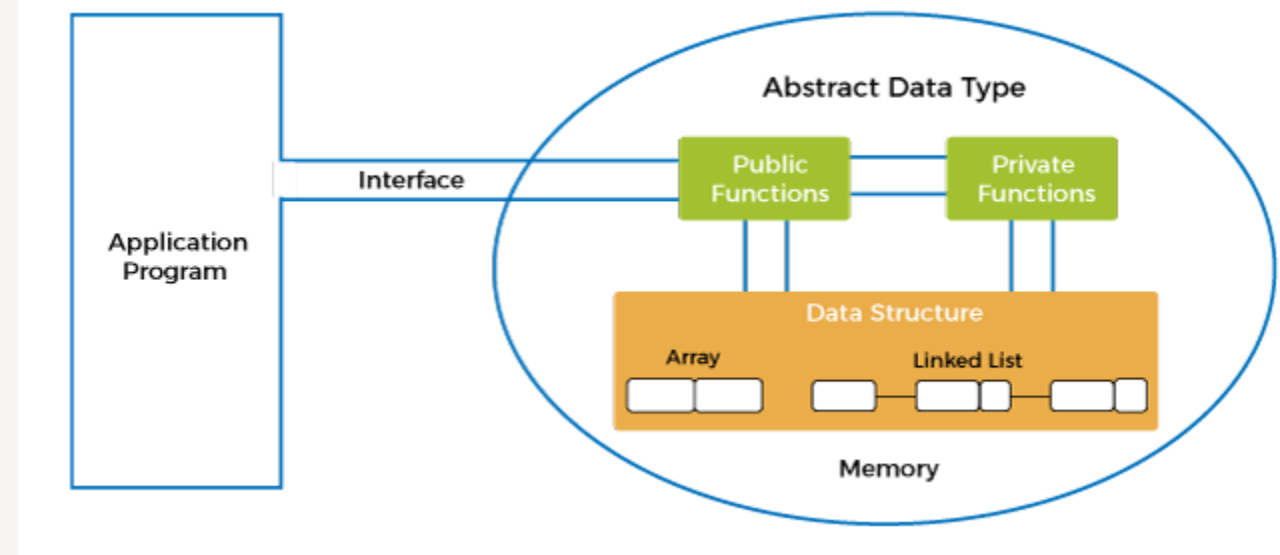

Figure 1: ADT

B. Feature of ADT

Data and actions on that data are combined into a single entity using abstract data types (ADTs). Key characteristics of ADTs include:

- **Abstraction**: Only the fundamentals are given, thus the user is not required to understand how the data structure is implemented.
- **Better Conceptualization**: The actual world is better understood for us thanks to ADT.
- **resilient**: The software is capable of detecting mistakes and is resilient.

- **Encapsulation**: ADTs encapsulate the underlying workings of the data and provide a user-interactive public interface. The data structure may now be modified and maintained more easily as a result.
- **Data Abstraction**: ADTs provide a degree of abstraction from the specifics of the data's implementation. Users do not need to be aware of the details around the execution of the operations that may be carried out on the data.
- **Data Structure Independence**: ADTs may be implemented using various data structures, such as arrays or linked lists, without having an impact on how well they work.
- **Information concealment**: By limiting access to only authorized individuals and processes, ADTs can safeguard the accuracy of the data. This lessens the chance of mistakes and data abuse.
- **Modularity**: ADTs are modular in that they may be used to create bigger, more complicated data structures. Programming may now be more flexible and modular as a result.

C. Advantage of ADT

- **Encapsulation:** Data and actions may be combined into one unit using ADTs, which makes the data structure simpler to maintain and alter.
- **Abstraction**: ADTs make working with data structures possible without requiring users to be familiar with implementation specifics, which may make programming simpler and less error-prone.
- **Data Structure Independence**: Different data structures may be used to implement ADTs, which can make it simpler to adapt to shifting demands and requirements.
- **Information concealment**: ADTs may safeguard data integrity by limiting access and blocking unwanted changes.
- **Modularity**: Programming may become more flexible and modular because to the modularity of ADTs, which can be coupled to create more sophisticated data structures.

D. Disadvantage of ADT

- **Overhead**: Implementing ADTs may result in additional memory and processing overhead, which might have an impact on speed.
- **Complexity**: Implementing ADTs may be challenging, particularly for big and intricate data structures.
- **Learning curve**: Understanding the implementation and use of ADTs is necessary for their use, which might take some time and effort to understand.
- **Limited Flexibility**: Some ADTs may not be suited for all sorts of data structures or may have limited capabilities.
- **Cost**: Putting ADTs into practice would call for more money and resources, which would raise the price of development.

E. examples of a general ADT

1. **Definition of Double LinkedList**

A Doubly Linked List is a variant of a Linked List, where traversing the nodes can be done in two directions: forward and backward easily when compared to a singly Linked List.

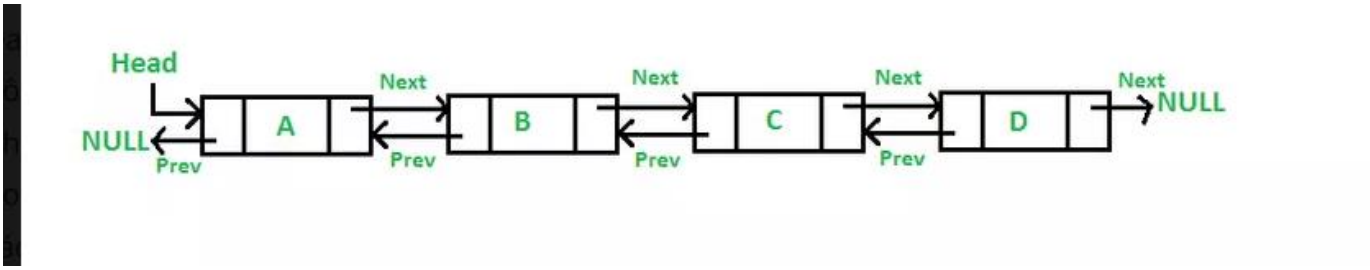

Figure 2: Double Linked List

2. Basic operation on Double LinkedList

- AddFirst(element): Add an element to the beginning of the list.

```
public void addFirst(E element) {
    Node<E> newNode = new Node<>(element, prev: null, head);
    if (head != null) {
        head.prev = newNode;
    }
    head = newNode;
    if (tail == null) {
        tail = newNode;
    }
    size++;
}
```

Figure 3: AddFirst1

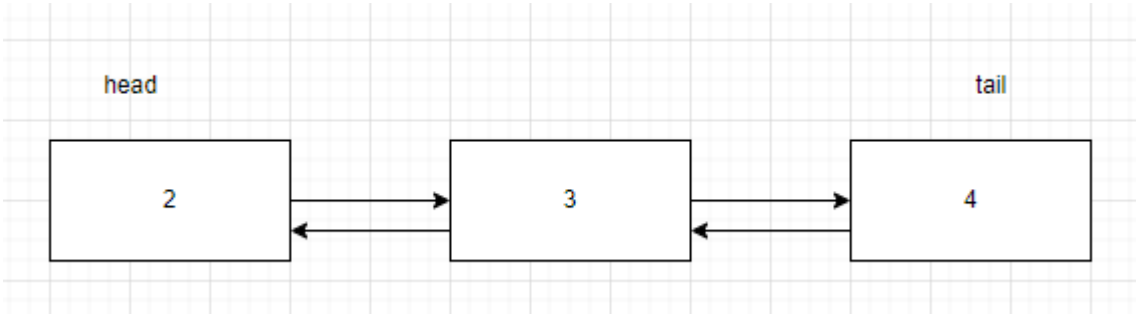

Figure 4: addfirst 2

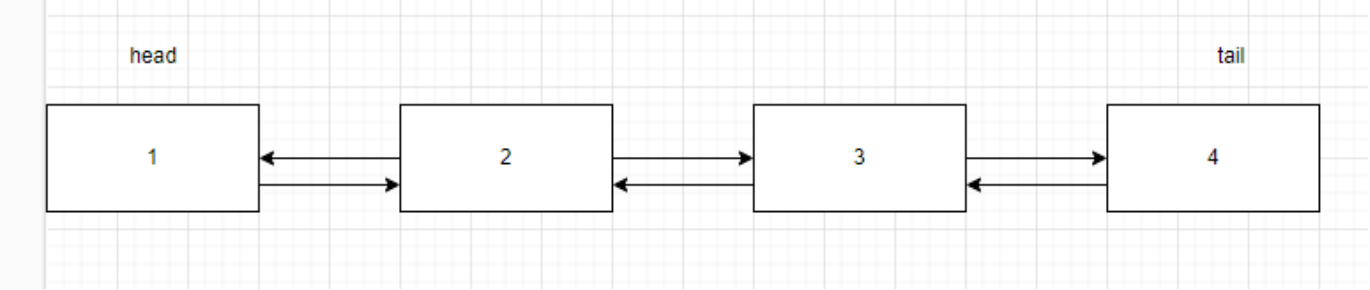

Figure 5: addfirst 3

- AddLast(element): Add an element to the end of the list.

```
public void addLast(E element) {
    Node<E> newNode = new Node<>(element, tail,  next: null);
    if (tail != null) {
        tail.next = newNode;
    }
    tail = newNode;
    if (head == null) {
        head = newNode;
    }
    size++;
}
```

x

Figure 6: Add Last 1

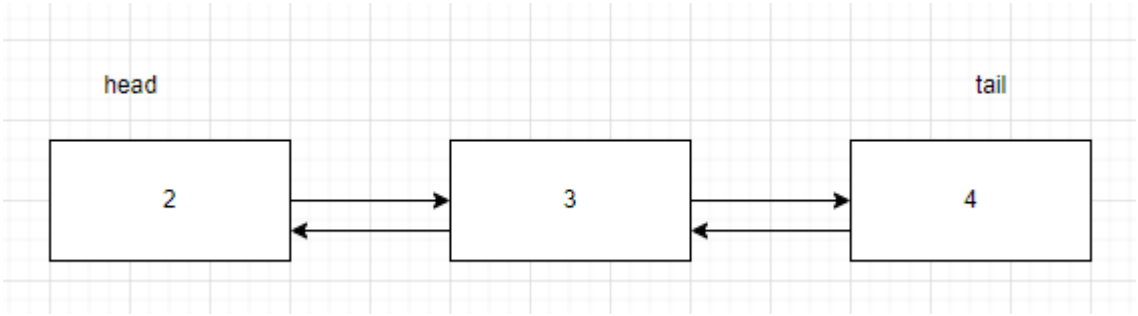

Figure 7: add last 2

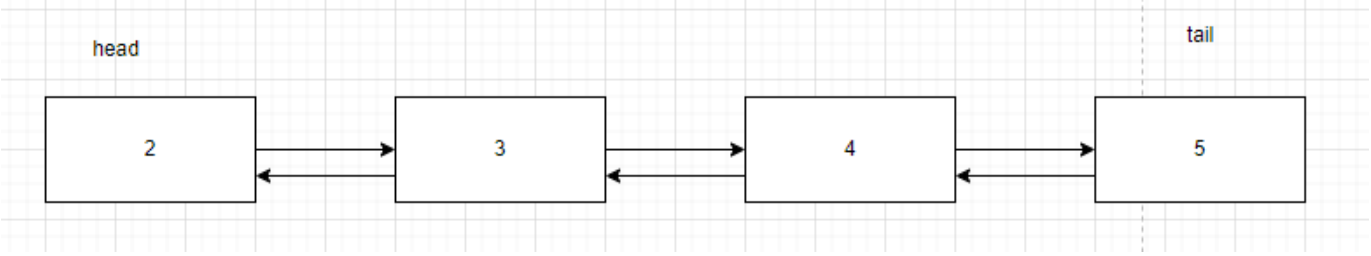

Figure 8: add last 3

- RemoveFirst(): Remove and return the element at the beginning of the list.

```
public E removeFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    E element = head.element;
    head = head.next;
    if (head != null) {
        head.prev = null;
    }
    size--;
    if (size == 0) {
        tail = null;
    }
    return element;
}
```

Figure 9: Remove Firsst 1

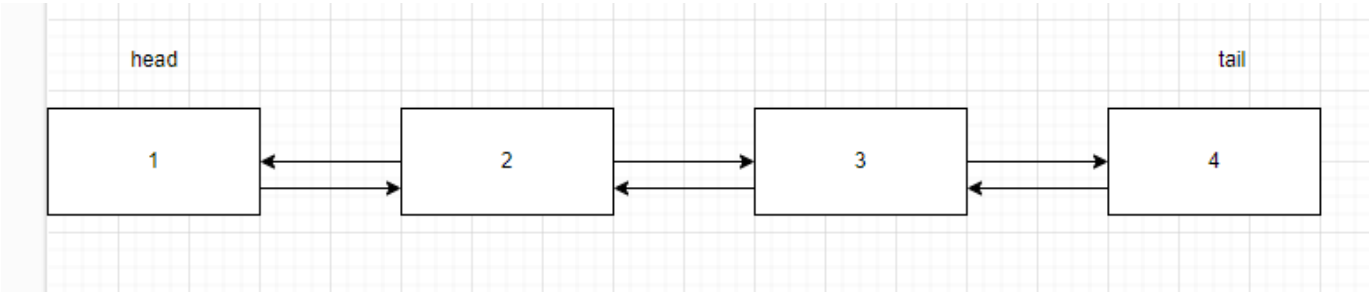

Figure 10: remove first 2

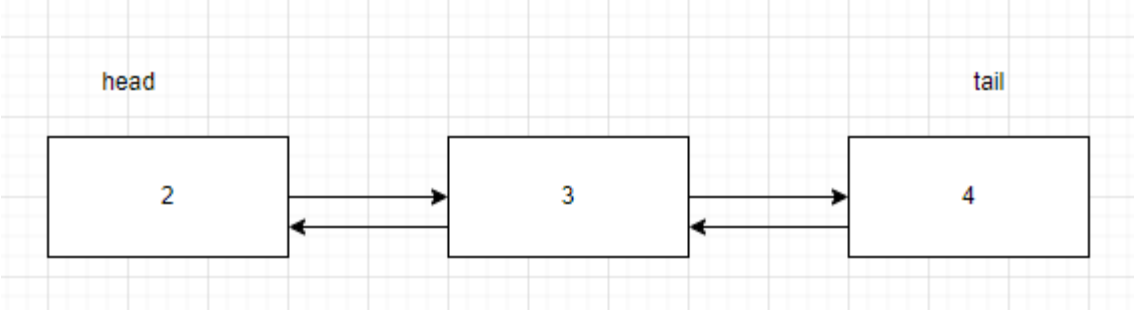

Figure 11: remove first 3

- RemoveLast(): Remove and return the element at the end of the list.

```
public E removeLast() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    E element = tail.element;
    tail = tail.prev;
    if (tail != null) {
        tail.next = null;
    }
    size--;
    if (size == 0) {
        head = null;
    }
    return element;
}

1 usage
```

Figure 12: Remove Last 1

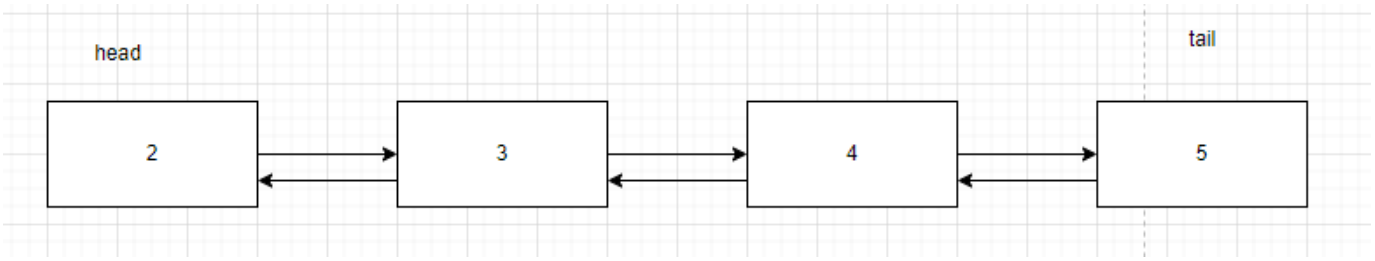

Figure 13: remove last 2

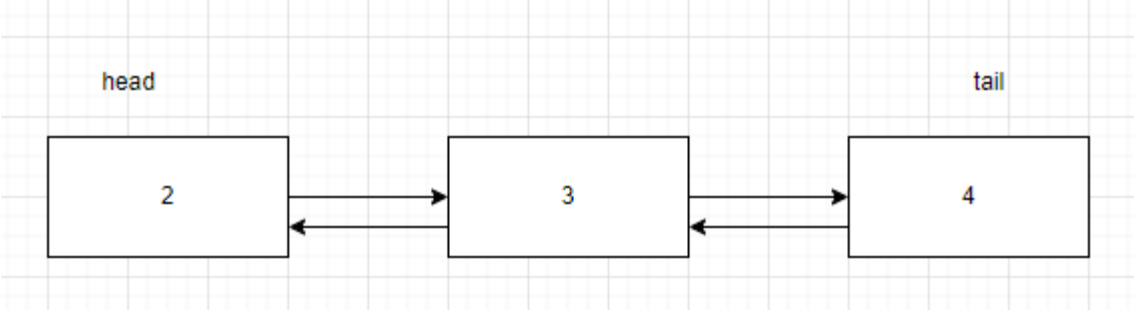

Figure 14: remove last 3

- GetFirst(): Return the element at the beginning of the list.

```
public E getFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    return head.element;
}
```

Figure 15: Get First

- GetLast(): Return the element at the end of the list.

```
public E getLast() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    return tail.element;
}
```

Figure 16: GetLast

- Size(): Return the number of elements in the list.

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

Figure 17: Size

- IsEmpty(): Check if the list is empty.

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

Figure 18: IsEmpty

3. **Advantage of Double linkedList**
   - A DLL may be walked through both forward and backward.
   - If a reference to the node that has to be destroyed is provided, the delete operation in DLL is more effective.
   - Before a certain node, a new node may be added fast.
   - A reference to the preceding node is required in a singly linked list in order to remove a node. Sometimes the list is traversed to get this preceding node. Using the previous pointer in DLL, we can get the previous node.
4. **Disadvantage of Double linkedList**
   - Every DLL node needs additional space for a prior pointer. However, DLL may be implemented with only one pointer.
   - An additional prior pointer must be preserved for all operations. For instance, while doing insertion, we must adjust both the preceding and subsequent pointers. For instance, setting the previous pointer requires one or two more steps in the following routines for insertions at various places.
5. **Example code**

```java
package implementations;

public class DoubleLinkedList<E> {
    10 usages
    private static class Node<E> {
        5 usages
        private E element;
        4 usages
        private Node<E> prev;
        4 usages
        private Node<E> next;

        2 usages
        public Node(E element, Node<E> prev, Node<E> next) {
            this.element = element;
            this.prev = prev;
            this.next = next;
        }
    }

    14 usages
    private Node<E> head;
    14 usages
    private Node<E> tail;
    9 usages
    private int size;

    1 usage
    public DoubleLinkedList() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }
```

Figure 19: Code Double Linked List

```
1 usage
public void addFirst(E element) {
    Node<E> newNode = new Node<>(element, prev: null, head);
    if (head != null) {
        head.prev = newNode;
    }
    head = newNode;
    if (tail == null) {
        tail = newNode;
    }
    size++;
}

3 usages
public void addLast(E element) {
    Node<E> newNode = new Node<>(element, tail, next: null);
    if (tail != null) {
        tail.next = newNode;
    }
    tail = newNode;
    if (head == null) {
        head = newNode;
    }
    size++;
}
```

Figure 20: Code Double Linked List 2

```java
public E removeFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    E element = head.element;
    head = head.next;
    if (head != null) {
        head.prev = null;
    }
    size--;
    if (size == 0) {
        tail = null;
    }
    return element;
}

public E removeLast() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    E element = tail.element;
    tail = tail.prev;
    if (tail != null) {
        tail.next = null;
    }
    size--;
    if (size == 0) {
        head = null;
    }
    return element;
}

1 usage
public E getFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("List is empty");
    }
    return head.element;
}
```

Figure 21: Code Double Linked List 3

```
    public E getLast() {
        if (isEmpty()) {
            throw new IllegalStateException("List is empty");
        }
        return tail.element;
    }

    public int size() {
        return size;
    }

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

Figure 22:Code Double Linked List 4

***Result code**:

```java
import implementations.*;
public class Main {
        public static void main(String[] args) {
            testAddFirst();
            testAddLast();
            testRemoveFirst();
            testRemoveLast();
            testGetFirst();
            testGetLast();
            testSize();
            testIsEmpty();
        }

        1 usage
        public static void testAddFirst() {
            DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
            list.addFirst( element: 10);
            list.addFirst( element: 5);
            list.addFirst( element: 20);

            System.out.println("Test Add First:");
            while (!list.isEmpty()) {
                System.out.print(list.removeFirst() + " ");
            }
            System.out.println();
        }

        1 usage
        public static void testAddLast() {
            DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
            list.addLast( element: 10);
            list.addLast( element: 5);
            list.addLast( element: 20);

            System.out.println("Test Add Last:");
            while (!list.isEmpty()) {
                System.out.print(list.removeFirst() + " ");
            }
            System.out.println();
        }
```

Figure 23: main double 1

```java
public static void testRemoveFirst() {
    DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
    list.addLast( element: 10);
    list.addLast( element: 5);
    list.addLast( element: 20);

    System.out.println("Danh sách ban đầu: " + list.toString());

    int removedElement = list.removeFirst();
    System.out.println("Phần tử đã bị xóa: " + removedElement);

    System.out.println("Danh sách sau khi xóa phần tử đầu: " + list.toString());
}

1 usage
public static void testRemoveLast() {
    DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
    list.addLast( element: 10);
    list.addLast( element: 5);
    list.addLast( element: 20);

    System.out.println("Danh sách ban đầu: " + list.toString());

    int removedElement = list.removeLast();
    System.out.println("Phần tử đã bị xóa: " + removedElement);

    System.out.println("Danh sách sau khi xóa phần tử cuối: " + list.toString());
}

1 usage
public static void testGetFirst() {
    DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
    list.addLast( element: 10);
    list.addLast( element: 5);
    list.addLast( element: 20);

    System.out.println("Test Get First: " + list.getFirst());
}
```

Figure 24:main double 2

```java
    1 usage
    public static void testGetLast() {
        DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
        list.addLast( element: 10);
        list.addLast( element: 5);
        list.addLast( element: 20);

        System.out.println("Test Get Last: " + list.getLast());
    }

    1 usage
    public static void testSize() {
        DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
        list.addLast( element: 10);
        list.addLast( element: 5);
        list.addLast( element: 20);

        System.out.println("Test Size: " + list.size());
    }

    1 usage
    public static void testIsEmpty() {
        DoubleLinkedList<Integer> list = new DoubleLinkedList<>();
        System.out.println("Test Is Empty: " + list.isEmpty());

        list.addLast( element: 10);
        list.addLast( element: 5);
        list.addLast( element: 20);
        list.removeFirst();
        list.removeFirst();
        list.removeFirst();

        System.out.println("Test Is Empty: " + list.isEmpty());
    }
}
```

Figure 25:main double 3

```
Test Add First:
20 5 10
Test Add Last:
10 5 20
Danh sách ban đầu: [10, 5, 20]
Phần tử đã bị xóa: 10
Danh sách sau khi xóa phần tử đầu: [5, 20]
Danh sách ban đầu: [10, 5, 20]
Phần tử đã bị xóa: 20
Danh sách sau khi xóa phần tử cuối: [10, 5]
Test Get First: 10
Test Get Last: 20
Test Size: 3
Test Is Empty: true
```

Figure 26: resul code double linked list

II. Determine the operations of a memory stack and how it is used to implement function calls in a computer(P2)

1. What is stack memory

Stack memory is a temporary memory area managed by the Last In First Out (LIFO) principle, meaning that the latest data placed on the stack will be accessed and removed first.

Every time the program makes a function call, a new stack frame is created and placed on top of the stack. This stack frame contains information about the current function call, including the return address, local variables, and function arguments.

When the function call completes, the corresponding stack frame is dropped from the top of the stack, and the program returns to execute from the return address stored in the previous frame.

2. Application in Stack memory

The stack memory is a portion of memory that utilises a Last-In-First-Out (LIFO) method in computer science and programming. During programme execution, it is mostly used to control function calls and local variables. Local variables and other relevant data are placed into the stack when a function is invoked. Control is sent back to the caller function when the function's execution is complete and its stack frame is popped.

Function Call:

- When a function is called, its return address (the address to which the control should return after the function finishes), parameters, and local variables are pushed onto the stack.
- This creates a new stack frame for the called function.

Nested Function Calls:

- If a function calls another function (nested function call), the return address and variables for the nested function are pushed on top of the previous function's stack frame.
- This creates a stack of stack frames, with the topmost frame representing the currently executing function.

Function Return:

- When a function completes its execution, its stack frame is popped from the stack, and control returns to the return address stored in the previous frame.
- The memory occupied by the local variables of the function is freed, and the stack pointer moves back to the previous frame.

Recursion:

- Recursion is a special case of function calls where a function calls itself.
- Each recursive call creates a new stack frame, and a stack of frames is formed until the base case is reached and the recursion unwinds.

3. Example code

```java
public class Example {

    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int result = add(a, b);
        System.out.println("Result: " + result);
    }

    public static int add(int x, int y) {
        int temp = x + y;
        int squared = square(temp);
        return squared;
    }

    1 usage
    public static int square(int num) {
        return num * num;
    }
}
```

Figure 27: code stack memory

In the main function, the program starts executing. The data frame for main is created and pushed to the top of the stack. Two variables a and b are initialized and assigned the values 5 and 10.

When add(a, b) is called, a new data frame is created and pushed to the top of the stack. The x and y variables in the add function are assigned the values of a and b. A temporary variable temp is created and calculates the sum of x and y.

Then, when calling square(temp), a new dataframe is created and pushed to the top of the stack. The variable num in the square function is assigned the value of temp. In the square function, the squared value of num is calculated and returned.

When every function completes, the corresponding data frames are removed from the top of the stack (pop) and the memory is released.

The result of the square of the sum of the two numbers is printed to the screen.

*__Result__:

```
C:\Users\PC\.jdks\openjdk-20.0.2\bin\java.exe ..
Result: 225
```

Figure 28: result

*__Illustration__:

Step 1: first stack memory will be empty and nothing

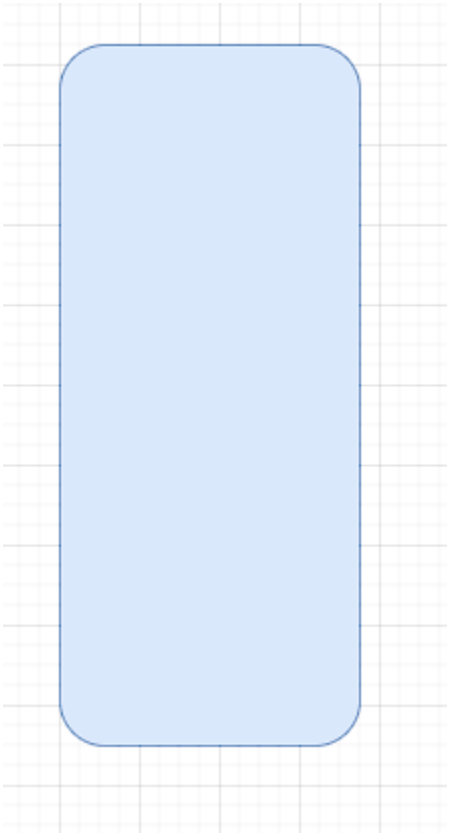

Figure 29: stack memory 1

Step 2: Next, we will call the main function and it will be stored in memory.

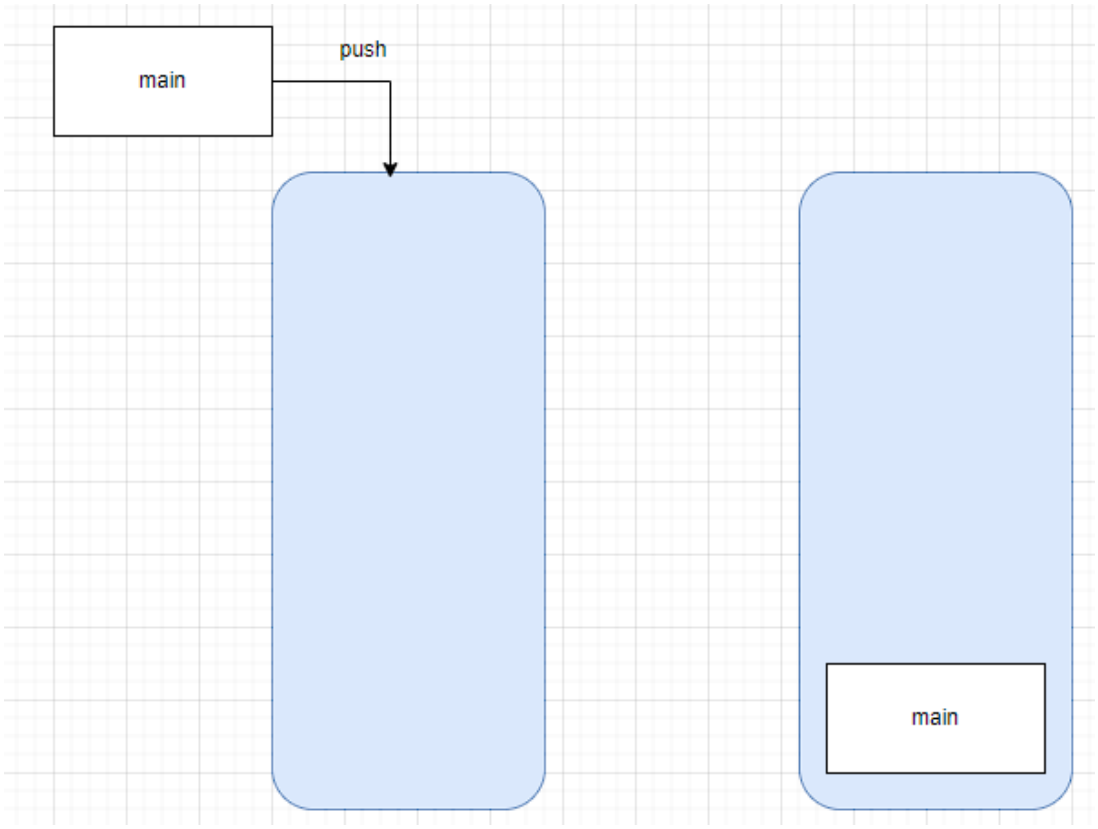

Figure 30:stack memory 2

Step 3: in the main function, the add function will be called so that the user can calculate the sum of x and y input by the user. Then add function will continue to call square function to be able to calculate the next step

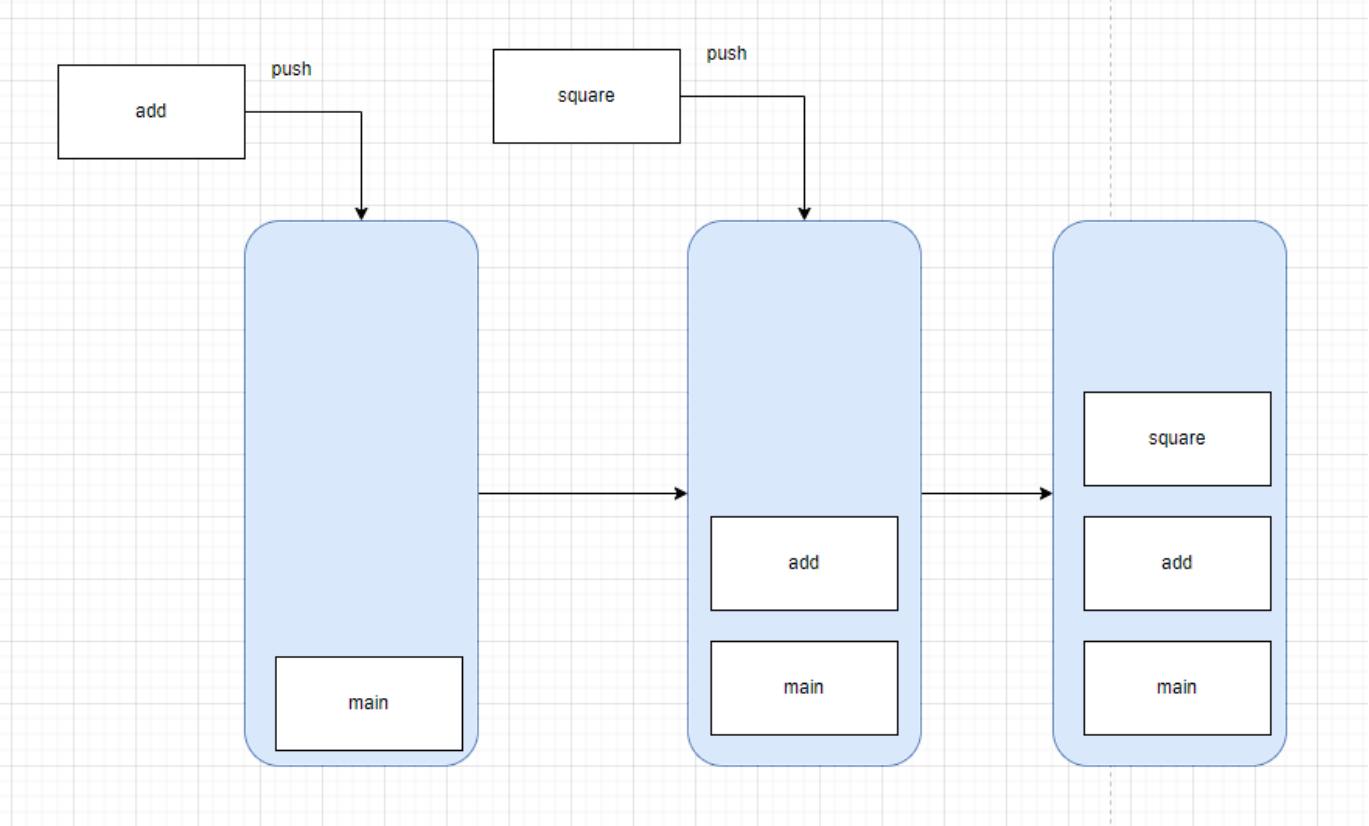

Figure 31:stack memory 3

Step 4: After finishing the calculation, the add and square functions will be pop according to LIFO rule

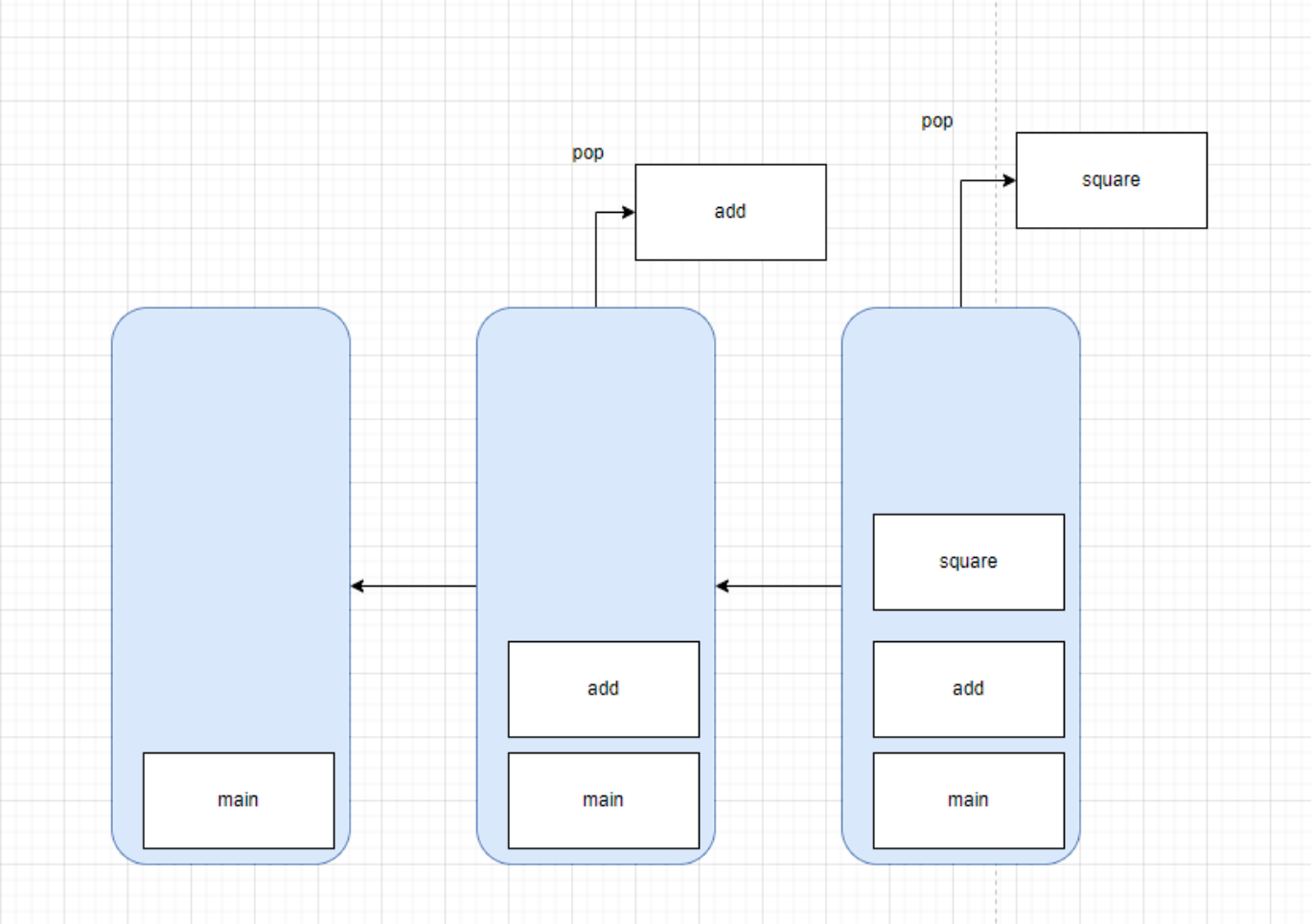

Figure 32: stack memory 4

Step 5: Finally, the program will remove the main function from the stack memory and return to the original stack memory

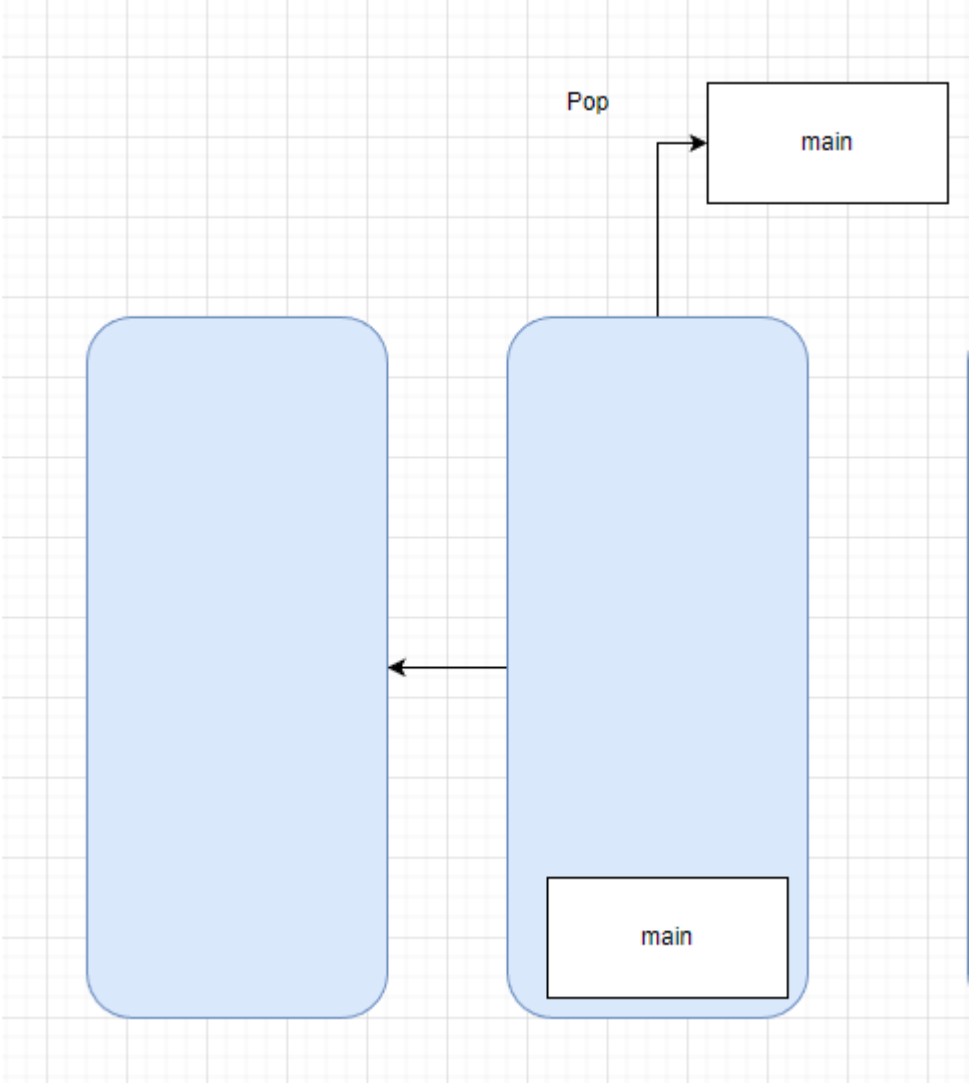

Figure 33: stack memory 5

III. Using an imperative definition, specify the abstract data type for a software stack.(P3)

1. Secenario

Currently, watching movies outside the theater is nothing new and the revenue of cinemas is gradually increasing. A small movie theater hired me to design an app for cinema management. To be able to complete this project, I need to use a data structure to be able to save to movie lists. And the array that I will use is suitable for cinema management and is part of the ADT that is ArrayList. ArrayList will be able to help me a lot in completing the cinema management project

2. What is ArrayList

The ArrayList class in Java is a class that implements the List Interface in the Collections Framework and inherits the AbstractList class, therefore it shares several traits and methods with List. A dynamic array called ArrayList is used to hold items.

Things to remember about ArrayList:

- Duplicate items are possible in Java's ArrayList class.
- The order of newly inserted items is preserved by the ArrayList class.
- It is not synchronized (asynchronous) to use the ArrayList class.
- Because it stores information by index, the ArrayList class enables random access.
- Java's ArrayList class operates slowly because it shifts data often when an element is deleted.

3. Basic Operation of ArrayList

- **add(E element):** This method is used to add a new element to the end of the list. If the list is full (the size is equal to the length of the storage array), the array will be expanded by creating a new array of double length, then the old elements will be copied into the new array. Then the new element will be added to the last position of the list.

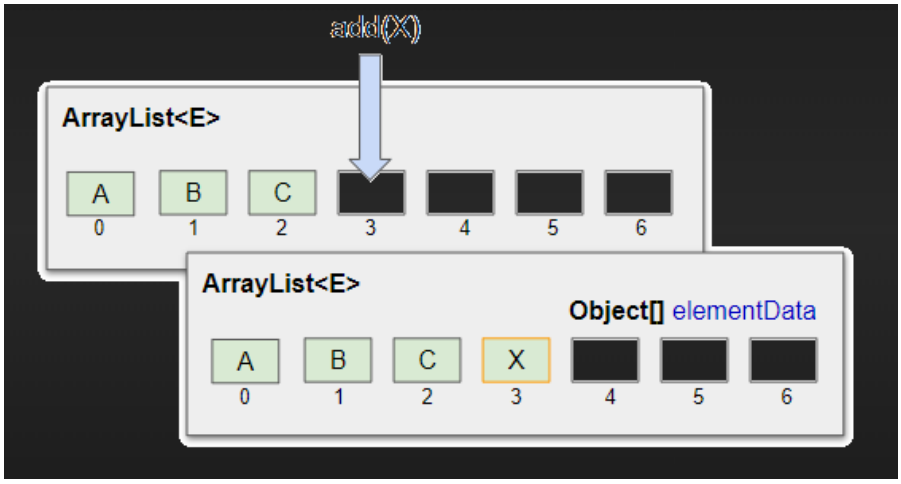

Figure 34: add function

```
@Override
public boolean add(E element) {
    if (this.size == this.elements.length) {
        this.elements = grow();
    }
    this.elements[this.size++] = element;
    return true;
}
```

Figure 35: add oparation

- **get(int index):** This method is used to retrieve the element at the specified index position in the list. Before retrieving, the method will check if the index is valid (between 0 and size - 1). If invalid, it will throw an IndexOutOfBoundsException exception. If the index is valid, the method will return the element at that position in the list.

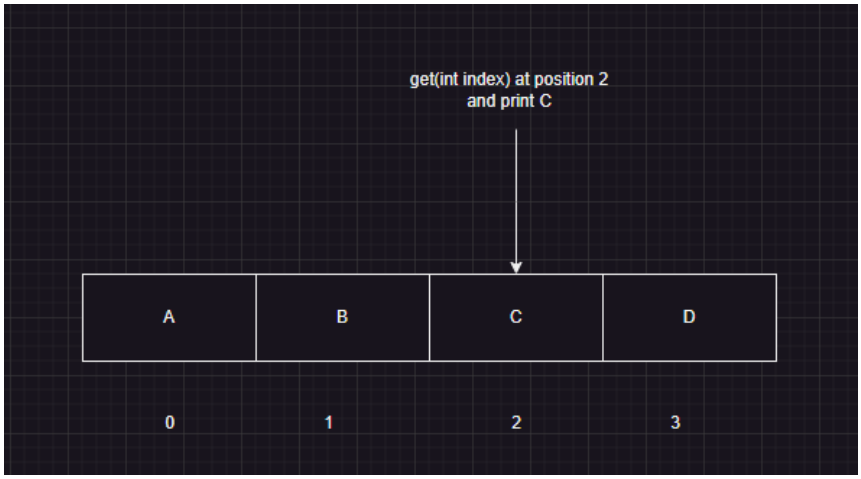

Figure 36: get function

```
@Override
public E get(int index) {
    checkIndex(index);
    return this.getElement(index);
}
```

Figure 37: get operation

- **remove(int index):** This method is used to remove the element at the specified index position from the list. Before deleting, the method also checks if the index is valid. Then the element at the index position is removed from the list by shifting all the elements after it to the left one position. Finally, the list size is reduced by 1. If, after deletion, the list size is less than 1/3 of the storage array size, the array will be shrunk by creating a new array of half the length. current and copies the remaining elements into the new array.

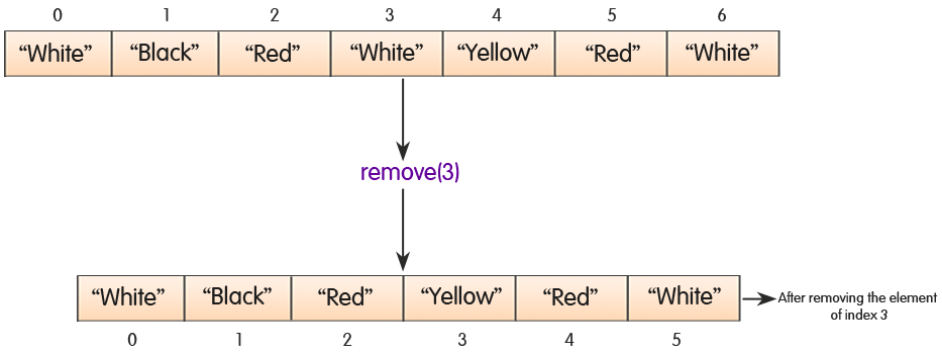

Figure 38: remove function

```java
@Override
public E remove(int index) {
    checkIndex(index);
    E element = this.getElement(index);
    shift(index);
    this.size--;
    ensureCapacity();
    return element;
}
```

Figure 39: remove operation

- **size():** This method returns the number of elements currently in the list.

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

Figure 40: size operation

- **contains(E element):** This method is used to check if the given element exists in the list. This method traverses the list and compares each element with the given element. If the given element appears in the list, the method returns true, otherwise it returns false.

```java
@Override
public int indexOf(E element) {
    for (int i = 0; i < this.size; i++) {
        if (this.elements[i].equals(element)) {
            return i;
        }
    }
    return -1;
}

4 usages
@Override
public boolean contains(E element) {
    return indexOf(element) != -1;
}
```

Figure 41: contains operation

4. Implement program

My program will be a movie list manager, so in this program it will have the basic functions to manage the movie list such as adding, deleting, printing out how many movies, and printing. Is the movie on the list of movies

a. Code ArrayList

```java
package implementations;

import interfaces.List;

import java.util.Arrays;
import java.util.Iterator;

public class ArrayList<E> implements List<E> {
    15 usages
    private int size;
    1 usage
    private static final int DEFAULT_CAPACITY = 4;
    21 usages
    private Object[] elements;
    public ArrayList() {
        this.elements = new Object[DEFAULT_CAPACITY];
    }
    2 usages
    private Object[] grow() {
        return Arrays.copyOf(this.elements,  newLength: this.elements.length * 2);
    }

    1 usage
    private Object[] shrink() {
        return Arrays.copyOf(this.elements,  newLength: this.elements.length / 2);
    }
```

Figure 42: code Arraylist 1

```java
1 usage
private void shift(int index) {
    for (int i = index; i < this.size - 1; i++) {
        this.elements[i] = this.elements[i + 1];
    }
    this.elements[this.size - 1] = null; // Xóa phần tử cuối cùng sau khi dịch chuyển
}

1 usage
private void ensureCapacity() {
    if (this.size < this.elements.length / 3) {
        this.elements = shrink();
    }
}

1 usage
private void insert(int index, E element) {
    if (this.size == this.elements.length) {
        this.elements = grow();
    }
    for (int i = this.size; i > index; i--) {
        this.elements[i] = this.elements[i - 1];
    }
    this.elements[index] = element;
    this.size++;
}

3 usages
private E getElement(int index) {
    if (index >= 0 && index < this.size) {
        return (E) this.elements[index];
    } else {
        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + this.size);
    }
}
```

Figure 43: code Arraylist 2

```java
@Override
public boolean add(E element) {
    if (this.size == this.elements.length) {
        this.elements = grow();
    }
    this.elements[this.size++] = element;
    return true;
}

@Override
public boolean add(int index, E element) {
    checkIndex(index);
    insert(index, element);
    return true;
}

6 usages
@Override
public E get(int index) {
    checkIndex(index);
    return this.getElement(index);
}

@Override
public E set(int index, E element) {
    checkIndex(index);
    E oldElement = this.getElement(index);
    this.elements[index] = element;
    return oldElement;
}
```

Figure 44: code Arraylist 3

```java
@Override
public E remove(int index) {
    checkIndex(index);
    E element = this.getElement(index);
    shift(index);
    this.size--;
    ensureCapacity();
    return element;
}

@Override
public int size() {
    return this.size;
}

3 usages
@Override
public int indexOf(E element) {
    for (int i = 0; i < this.size; i++) {
        if (this.elements[i].equals(element)) {
            return i;
        }
    }
    return -1;
}

4 usages
@Override
public boolean contains(E element) {
    return indexOf(element) != -1;
}

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

Figure 45: code Arraylist 4

```java
@Override
public Iterator<E> iterator() {
    return new Iterator<E>() {
        2 usages
        private int index = 0;

        @Override
        public boolean hasNext() {
            return index < size();
        }

        @Override
        public E next() {
            return get(index++);
        }
    };
}
```

Figure 46: code Arraylist 5

b. Class Movie

```java
package implementations;

public class Movie {
    6 usages
    private String title;
    6 usages
    private String genre;
    4 usages
    public Movie(String title, String genre) {
        this.title = title;
        this.genre = genre;
    }

    2 usages
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }
    @Override
    public String toString() {
        return "Title: " + title + ", Genre: " + genre;
    }
```

Figure 47: class Movie

```
    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof Movie)) {
            return false;
        }
        Movie otherMovie = (Movie) obj;
        return this.title.equals(otherMovie.title) && this.genre.equals(otherMovie.genre);
    }
}
```

Figure 48: class Movie 2

We have a Movie class to represent a movie. This class has two properties title and genre, representing the title and genre of the movie.

Methods in the Movie class:

- **Constructor**: Movie(String title, String genre): This is the constructor to create a new Movie object with the provided title and genre.
- **Getter and Setter**: For the title and genre properties, we have getter and setter methods to access and update their values.
- **toString() method**: This method is overridden from the Object class and returns a string representing information about the movie, including the title and genre.
- **equals(Object obj):** This method is overridden to compare two Movie objects. It checks if the supplied object is another Movie object and then compares the titles and genres of the two to determine if they are the same.

This code allows us to create Movie objects, store information about the titles and genres of the movies, and compare them to determine if two movies are the same.

c. Class Movie Manage

```java
package implementations;

public class MovieManager {
    6 usages
    private ArrayList<Movie> movieList;

    1 usage
    public MovieManager() {
        this.movieList = new ArrayList<>();
    }

    3 usages
    public void addMovie(Movie movie) {
        movieList.add(movie);
    }

    1 usage
    public void removeMovie(int index) {
        movieList.remove(index);
    }

    3 usages
    public Movie getMovie(int index) {
        return movieList.get(index);
    }

    2 usages
    public int getMovieCount() {
        return movieList.size();
    }

    1 usage
    public boolean containsMovie(Movie movie) {
        return movieList.contains(movie);
    }
}
```

Figure 49: class Movie Manage

I have a MovieManager class to manage a list of movies. This class has a property movieList, which is an ArrayList object containing Movie objects.

Methods in the MovieManager class:

- **Constructor**: MovieManager(): This is the constructor to create a new MovieManager object. In this method, we initialize the movieList with an ArrayList object.
- **addMovie(Movie movie) method**: This method allows to add a Movie object to the movieList list. The Movie object is provided as an argument to the method.
- **removeMovie(int index):** This method allows to remove a movie from the movieList, based on the index of the movie you want to remove. The index is provided as a parameter to the method.
- **getMovie(int index):** This method returns the Movie object from the movieList, based on the index of the movie to get. The index is provided as a parameter to the method.
- **getMovieCount() method:** This method returns the number of movies currently in the movieList.
- **containsMovie(Movie movie) method:** This method checks if a given Movie object exists in the movieList. The method returns true if the given object exists in the list, and false if it does not.

The MovieManager class allows adding, deleting, retrieving information and checking the existence of movies in the management list. This makes it easy for me to manage and work with my list of movies.

d. Code main

```java
import implementations.*;
public class Main {
    public static void main(String[] args) {
        // Tạo đối tượng quản lý danh sách phim
        MovieManager movieManager = new MovieManager();

        // Thêm phim vào danh sách
        movieManager.addMovie(new Movie( title: "The Shawshank Redemption", genre: "Drama"));
        movieManager.addMovie(new Movie( title: "Inception", genre: "Sci-Fi"));
        movieManager.addMovie(new Movie( title: "The Dark Knight", genre: "Action"));

        // Hiển thị số lượng phim trong danh sách
        System.out.println("Number of movies in the list: " + movieManager.getMovieCount());

        // Hiển thị thông tin phim
        System.out.println("Movie at index 0: " + movieManager.getMovie( index: 0));
        System.out.println("Movie at index 1: " + movieManager.getMovie( index: 1));
        System.out.println("Movie at index 2: " + movieManager.getMovie( index: 2));

        // Kiểm tra xem một phim đã cho có tồn tại trong danh sách hay không
        Movie movieToCheck = new Movie( title: "Inception", genre: "Sci-Fi");
        if (movieManager.containsMovie(movieToCheck)) {
            System.out.println("The movie '" + movieToCheck.getTitle() + "' exists in the list.");
        } else {
            System.out.println("The movie '" + movieToCheck.getTitle() + "' does not exist in the list.");
        }

        // Xóa một phim khỏi danh sách
        movieManager.removeMovie( index: 1);

        // Hiển thị số lượng phim sau khi xóa
        System.out.println("Number of movies in the list after removal: " + movieManager.getMovieCount());
    }
}
```

Figure 50: code Main

This is the class that runs the main program:

- Create a MovieManager object to manage the movie list.
- Add three movies to the list via addMovie().
- Display the number of movies in the list by calling getMovieCount().
- Retrieve and display the information of the first three movies in the list using getMovie() and print the information to the screen.
- Check if a specific movie (eg "Inception") already exists in the list using containsMovie() and print the result to the screen.
- Removes a movie from the list and shows the number of movies remaining after deletion.

e. Result code

a) create management objects, import movie, print number of movie and information about movie

```
// Tạo đối tượng quản lý danh sách phim
MovieManager movieManager = new MovieManager();

// Thêm phim vào danh sách
movieManager.addMovie(new Movie( title: "The Shawshank Redemption", genre: "Drama"));
movieManager.addMovie(new Movie( title: "Inception", genre: "Sci-Fi"));
movieManager.addMovie(new Movie( title: "The Dark Knight", genre: "Action"));

// Hiển thị số lượng phim trong danh sách
System.out.println("Number of movies in the list: " + movieManager.getMovieCount());

// Hiển thị thông tin phim
System.out.println("Movie at index 0: " + movieManager.getMovie( index: 0));
System.out.println("Movie at index 1: " + movieManager.getMovie( index: 1));
System.out.println("Movie at index 2: " + movieManager.getMovie( index: 2));
```

Figure 51: print infor movie

```
Number of movies in the list: 3
Movie at index 0: Title: The Shawshank Redemption, Genre: Drama
Movie at index 1: Title: Inception, Genre: Sci-Fi
Movie at index 2: Title: The Dark Knight, Genre: Action
The movie 'Inception' exists in the list
```

Figure 52: result 1

b) find Movie in list, remove movie, and print List after remove

```
        // Kiểm tra xem một phim đã cho có tồn tại trong danh sách hay không
        Movie movieToCheck = new Movie( title: "Inception", genre: "Sci-Fi");
        if (movieManager.containsMovie(movieToCheck)) {
            System.out.println("The movie '" + movieToCheck.getTitle() + "' exists in the list.");
        } else {
            System.out.println("The movie '" + movieToCheck.getTitle() + "' does not exist in the list.");
        }

        // Xóa một phim khỏi danh sách
        movieManager.removeMovie( index: 1);

        // Hiển thị số lượng phim sau khi xóa
        System.out.println("Number of movies in the list after removal: " + movieManager.getMovieCount());
    }
}
```

Figure 53: find movie and remove movie

```
The movie 'Inception' exists in the list.
Number of movies in the list after removal: 2
```

Figure 54: result 2

IV. Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue (M1)

1. Scenario

Based on part P3, then I will do a cinema management project for a small cinema that has hired me. I used ArrayList to be able to manage the movie list easily. And next I will use queue to be able to manage the list of customers that will be served in FIFO order, which means that customers who have booked and come first will be served first.

2. What is Queue

Queue is an abstract data structure that resembles a queue in real-world situations (queuing).

The queue is open on both ends, in contrast to the stack. Data is always added to one end (also known as rowing) and removed from the other end (also known as departing). The data inserted initially will be accessible first since the queue data structure uses the initially-In-First-Out approach.

Figure 55: queue

3. Why Queue is suitable for this problem

Queue is suitable for managing a list of pre-booked customers because it supports the implementation of a "First In First Out" (FIFO) mechanism, i.e. requests that come first will be served first. Later requests will have to wait until their turn.

Reasons why queues are appropriate in this case:

- **Fair Processing**: With queues, booking requests will be processed in the order they arrive, with no priority or preference for any customer. This helps to ensure fairness and avoid bias.
- **Convenience and efficiency**: Queue provides simple and efficient methods to add a request to a waitlist and retrieve the next request for processing. This process helps to

reduce the time and effort of management staff and ensures the service process takes place efficiently.

- **Automatic order management**: Queue automatically maintains the order of requests, without the need for manual intervention from the management staff. This helps to avoid confusion and errors in the processing of requests.
- **Easily extensible**: If the management program needs to be extended to support more customers, using queues provides a flexible mechanism for efficient waitlist management and easy system scalability.

4. **Basic operation of Queue**
   - **Offer(E element):** With the help of the offer(E element) method, an element may be added at the very end of the queue. The element is handed in and used to generate a new node, which is then added to the end of the linked list. The new node is assigned as head if the list is empty. The size of the Queue grows after addition.

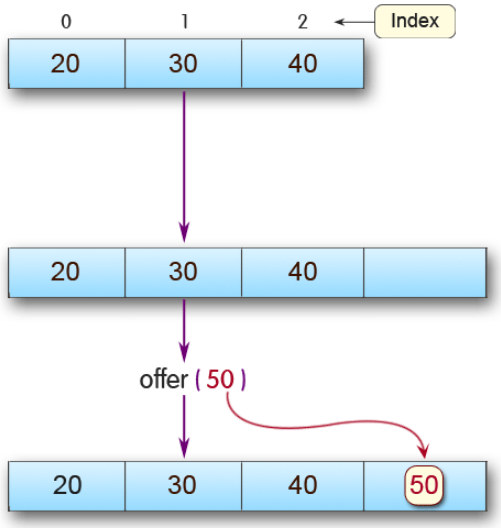

Figure 56: offer function

```java
@Override
public void offer(E element) {
    Node<E> newNode = new Node<>(element);
    if (this.head == null) {
        this.head = newNode;
    } else {
        Node<E> current = this.head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
    this.size++;
}
```

Figure 57: code offer

- **Poll():** Using the poll() function, the first member in the queue is taken out and returned. The function uses ensureNonEmpty() to determine if the Queue is empty before dismissing. The head is then transferred to the next node, if any, after the current head is discarded. The size of the queue shrinks after removal.

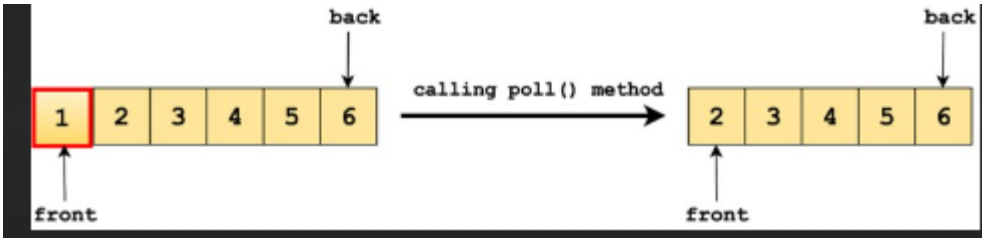

Figure 58: poll function

```java
@Override
public E poll() {
    ensureNonEmpty();
    E element = this.head.element;
    Queue.Node<E> temp = this.head.next;
    this.head.next = null;
    this.head = temp;
    this.size--;
    return element;
}
```

Figure 59: code poll

- **Peek():** Without removing it from the Queue, the peek() function returns the element at the top of the list. Using ensureNonEmpty(), the function determines if the Queue is empty before returning.

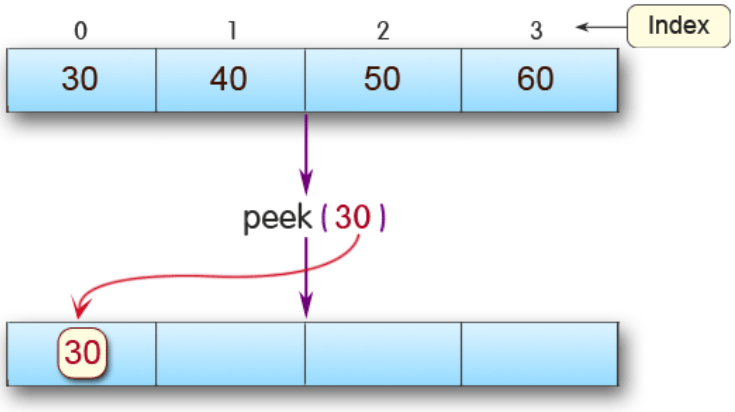

Figure 60: peek function

```java
@Override
public E peek() {
    ensureNonEmpty();
    return this.head.element;
}
```

Figure 61: code peek

- **Size():** the quantity of entries in the queue is returned by the size() function.

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

Figure 62: code size

- **isEmpty():** The isEmpty() function determines if the Queue is empty. The Queue is empty if head is null.

```
@Override
public boolean isEmpty() { return this.head ==null; }
2 usages
```

Figure 63: code isEmpty

5. Implement Code

a. Code Queue

```java
package implementations;

import ...

17 usages
public class Queue<E> implements AbstractQueue<E> {
    11 usages
    private Node<E> head;
    4 usages
    private int size;
    7 usages
    private static class Node<E> {
        4 usages
        private final E element;
        7 usages
        private Queue.Node<E> next;


        1 usage
        public Node(E element) {
            this.element = element;
            this.next = null;
        }
    } // Node class
    8 usages
    public Queue() {
        this.head = null;
        this.size= 0;
    }
```

Figure 64: code queue 1

```java
@Override
public void offer(E element) {
    Node<E> newNode = new Node<>(element);
    if (this.head == null) {
        this.head = newNode;
    } else {
        Node<E> current = this.head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
    this.size++;
}

4 usages
@Override
public E poll() {
    ensureNonEmpty();
    E element = this.head.element;
    Queue.Node<E> temp = this.head.next;
    this.head.next = null;
    this.head = temp;
    this.size--;
    return element;
}

4 usages
@Override
public E peek() {
    ensureNonEmpty();
    return this.head.element;
}
```

Figure 65: code queue 2

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

    @Override
    public boolean isEmpty() { return this.head ==null; }
    2 usages
    private void ensureNonEmpty() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
    }

    @Override
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            4 usages
            private Queue.Node<E> current = head;
            @Override
            public boolean hasNext() { return current != null; }

            @Override
            public E next() {
                E element = current.element;
                this.current = this.current.next;
                return element;
            }
        };
    }
}
```

Figure 66: code queue 3

b. Code class Customer

```java
package implementations;

public class Customer {
    3 usages
    private String name;
    3 usages
    private int age;
    3 usages
    private String phoneNumber;
    3 usages
    private String email;

    3 usages
    public Customer(String name, int age, String phoneNumber, String email) {
        this.name = name;
        this.age = age;
        this.phoneNumber = phoneNumber;
        this.email = email;
    }

    public String getName() { return name; }

    public int getAge() {
        return age;
    }

    public String getPhoneNumber() { return phoneNumber; }

    public String getEmail() { return email; }

    @Override
    public String toString() {
        return "Name: " + name + ", Age: " + age + ", Phone: " + phoneNumber + ", Email: " + email;
    }
}
```

Figure 67: class Customer

In the above code, Customer is a class that represents a customer's information. This class has properties and methods to manage customer information such as name, age, phone number, and email address.

The properties of the Customer class include:

- **name**: This is the name of the customer. Initialized via constructor and has a getName() method to return the value of this property.
- **age**: This is the age of the customer. Also initialized via constructor and has a getAge() method to return the value of this property.

- **phoneNumber**: This is the customer's phone number. Initialized via constructor and has a getPhoneNumber() method to return the value of this property.
- **email**: This is the email address of the customer. Initialized via constructor and has getEmail() method to return the value of this property.

In addition, the Customer class has a toString() method to create a string representing the customer's information. This method is called when you want to display the Customer object's information as a string.

c. Code class Customer List

```java
public class CustomerList {
    7 usages
    private Queue<Customer> customers;

    1 usage
    public CustomerList() { customers = new Queue<>(); }

    3 usages
    public void addCustomer(Customer customer) {
        customers.offer(customer);
    }

    public Customer getNextCustomer() {
        if (isEmpty()) {
            return null; // Trả về null nếu danh sách khách hàng rỗng
        }
        return customers.peek();
    }

    1 usage
    public Customer serveNextCustomer() {
        if (isEmpty()) {
            return null; // Trả về null nếu danh sách khách hàng rỗng
        }
        return customers.poll();
    }

    public int getSize() { return customers.size(); }

    public boolean isEmpty() { return customers.isEmpty(); }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Customer customer : customers) {
            sb.append(customer.toString()).append("\n");
        }
        return sb.toString();
    }}
```

Figure 68: class Customer List

In the above code, we have a CustomerList class to manage the customer list. This class uses a Queue to store the clients and implements the following functionality:

- **addCustomer(Customer customer):** This method is used to add a new customer to the list. The customer will be added to the end of the queue (enqueue) using the offer method of the Queue.
- **getNextCustomer():** This method returns the next customer in the list without removing it from the list (i.e. only viewing the first customer information). It uses the Queue's peek method to get the customer information at the top of the queue without removing that customer from the queue.
- **serveNextCustomer():** This method returns and removes the next customer from the list. It uses the poll method of the Queue to get and remove the customers at the top of the queue.
- **getSize():** This method returns the number of existing customers in the list. It uses the size method of the Queue to get the number of elements in the queue.
- **isEmpty():** This method checks if the customer list is empty. It uses Queue's isEmpty method to check if the queue is empty.
- **toString():** This method is overridden to return a string containing the information of all customers in the list. It traverses all the elements in the queue and adds the customer information to the StringBuilder. Each customer information is separated by a newline \n. The end result is a string containing the information of all the customers in the list.

d. Code Main

```java
public static void main(String[] args) {
    CustomerList customerList = new CustomerList();

    // Thêm khách hàng vào danh sách
    customerList.addCustomer(new Customer( name: "Khách hàng A",  age: 30,  phoneNumber: "123456789",  email: "customerA@example.com"));
    customerList.addCustomer(new Customer( name: "Khách hàng B",  age: 25,  phoneNumber: "987654321",  email: "customerB@example.com"));
    customerList.addCustomer(new Customer( name: "Khách hàng C",  age: 40,  phoneNumber: "456789123",  email: "customerC@example.com"));

    // In ra số lượng khách hàng trong danh sách
    System.out.println("Số lượng khách hàng trong danh sách: " + customerList.getSize());
    System.out.println("khách hàng ban đầu: "+ customerList.toString());

    // Phục vụ khách hàng tiếp theo và in thông tin khách hàng đã được phục vụ
    Customer nextCustomer = customerList.serveNextCustomer();
    if (nextCustomer != null) {
        System.out.println("Khách hàng đã được phục vụ:");
        System.out.println(nextCustomer);
    } else {
        System.out.println("Danh sách khách hàng rỗng.");
    }

    // In ra số lượng khách hàng còn lại trong danh sách sau khi phục vụ
    System.out.println("Số lượng khách hàng còn lại trong danh sách: " + customerList.getSize());
    System.out.println("khách hàng còn lại:"+customerList.toString());
}
```

Figure 69: code main

In the main function, the program has already entered the customers. The program will use addCustomer to be able to add customers to the list, then it can print out a list of customers with how many people and customer information. Next, the program will choose the first customer to come to serve, after the service is completed, the first customer will be removed from the list and after calling back the number of the list is only 2 customers.

e. Result

```
Số lượng khách hàng trong danh sách: 3
khách hàng ban đầu: Name: Khách hàng A, Age: 30, Phone: 123456789, Email: customerA@example.com
Name: Khách hàng B, Age: 25, Phone: 987654321, Email: customerB@example.com
Name: Khách hàng C, Age: 40, Phone: 456789123, Email: customerC@example.com

Khách hàng đã được phục vụ:
Name: Khách hàng A, Age: 30, Phone: 123456789, Email: customerA@example.com
Số lượng khách hàng còn lại trong danh sách: 2
khách hàng còn lại:Name: Khách hàng B, Age: 25, Phone: 987654321, Email: customerB@example.com
Name: Khách hàng C, Age: 40, Phone: 456789123, Email: customerC@example.com
```

Figure 70: result code queue

When running the program, information and number of customers will appear. Then the first served customer will be selected and deleted. Finally, update the list and print out the remaining customers

V. Compare the performance of two sorting algorithms (M2)

1. Insert sort

a. What is insert sort

Insertion sort is an in-place comparison-based sorting algorithm. Here, a sublist is always maintained in sorted form. Insertion sort is to insert an element into a sorted sublist. The element is inserted in the appropriate position so that the sublist remains in order.

With an array data structure, we imagine an array consisting of two parts: a sorted sublist and the other unordered elements. The insertion sort algorithm will perform a sequential search through the array, and the unordered elements will be moved and inserted into the appropriate position in the sublist (of the same array).

This algorithm is not suitable for use with large data sets when the average and worst case complexity is O(n2) where n is the number of elements.

b. How insert sort work

- **Step 1**: See the first element of the array as sorted.
- **Step 2**: Move to the next element. If the element is smaller than the previous element, move the previous element up and place the current element in the appropriate position in the sorted list.
- **Step 3**: Repeat Step 2 for all the elements in the unsorted list.
- **Step 4**: The sorted list will ascend from left to right.

c. Example with an unsorted array: [5, 2, 4, 6, 1, 3]:

- **Step 1**: Unsorted Array: [5, 2, 4, 6, 1, 3]

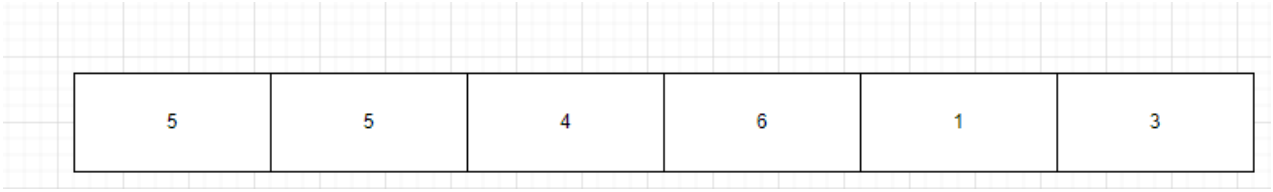

Figure 71: unsorted array

- **Step 2**: We will do insertion sort with the first 2 word parts 2 and 5. We can see that 2 will be less than 5 and from there we will swap them

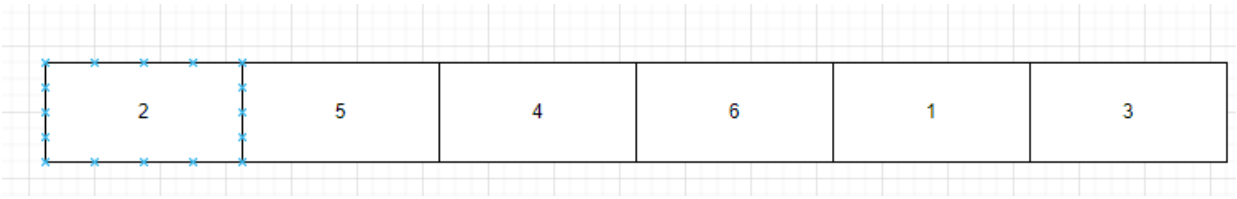

Figure 72: example insert sort 1

- **Step 3**: Continue comparing 2 elements 5 and 4. See 5 > 4 and we will swap 2 elements. Then continue to compare with the remaining elements that are number 2, then 4 > 2, so it won't change places

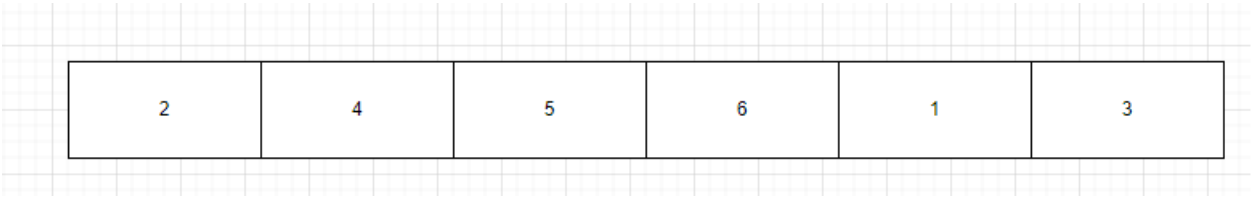

Figure 73:example insert sort 2

- **Step 4**: Continuing to compare, 5<6 so these 2 elements do not need to be interchanged

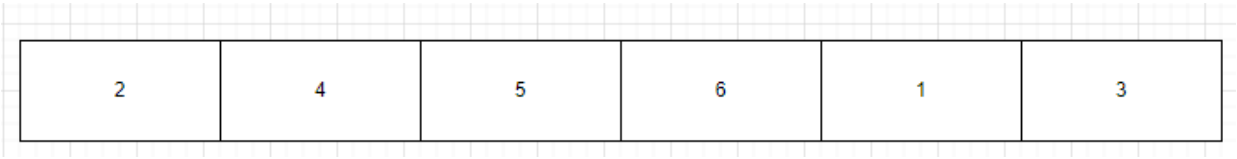

Figure 74:example insert sort 3

- **Step 5**: Continuing to compare, 6>1 so we will swap the 2 elements for each other. Then we will continue to compare the number 1 and the remaining elements are 2,4,5 then we can see that the number 1 is the smallest and we can insert the number 1 at the beginning

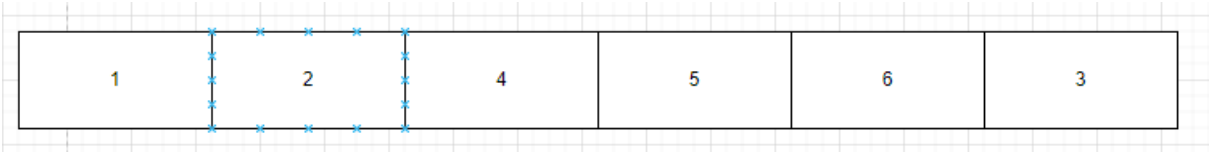

Figure 75:example insert sort 4

- **Step 6**: Keep comparing, then 6<3 and we will change their position then keep comparing 3 with remaining elements 1,2,4,5 and we can see that 3 will be small more than 4.5 and greater than 1.2 and we will insert the number 3 between 2 and 4

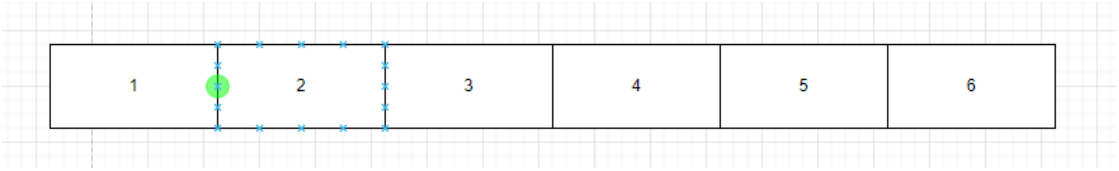

Figure 76:example insert sort 5

d. Analyze its performance (time, memory)

a) Time

- Best case: O(n)

In the ideal scenario, the array is already sorted, making finding each element's precise location as easy as comparing adjacent items. Therefore, there will be n-1 comparisons, and the execution time will be O(n).

- Average and worst case: O(n^2)

Each element must be placed in the proper location in the sorted array whether the array is unordered or reverse ordered. To precisely locate the present element, this calls for comparing and shifting bigger elements. (n-1) + (n-2) +... + 1 = n*(n-1)/2, or O(n2). The total number of comparisons will be (n-1) + (n-2) +... + 1.

b) Memory

Without needing extra memory for the new array, Insertion Sort is carried out on the existing array. As a result, the space utilisation is O(1) (constant), meaning that it is independent of the size of the input array.

e. Example code

```java
import java.util.Arrays;

public class InsertionSort {

    1 usage
    public static void insertionSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 6, 1, 3, 4};
        System.out.println("Before");
        System.out.println(Arrays.toString(arr));
        insertionSort(arr);
        System.out.println("After");
        System.out.println(Arrays.toString(arr));
    }
}
```

Figure 77: example code insert sort

***result:**

```
Before
[5, 2, 6, 1, 3, 4]
After
[1, 2, 3, 4, 5, 6]
```

Figure 78: result insert sort

2. Quick sort

a. What is quick sort

Quick Sort is a highly efficient algorithm and is based on dividing arrays of data into smaller arrays. The quick sort algorithm divides the array into two parts by comparing each element of the array with a selected element called the pivot element (Pivot): an array consisting of elements less than or equal to the peg and the array the rest consists of elements greater than or equal to the pin element.

This division process continues until the lengths of the sub-arrays are all equal to 1. The quick sort algorithm is quite effective for large data sets where the average and bad case complexity at most O(nlogn) where n is the number of elements.

b. How quick sort work

- Select a pivot element from the array. In the above code example, the latch element is selected as the last element of the array.
- Split the array into two parts: a part consisting of elements less than or equal to the latch element (left) and a part consisting of elements greater than the pin element (right).
- Continue performing Quick Sort for both subsections (left and right) by recursion.
- The recursion continues until the array has only one element left (array of size 1), at which point the array is considered sorted.
- Combine the results from the two sorted subsections to create the complete sorted array

c. Example with an unsorted array: [5, 2, 4, 6, 1, 3]:

- **Step 1**: Unsorted Array: [5, 2, 4, 6, 1, 3]

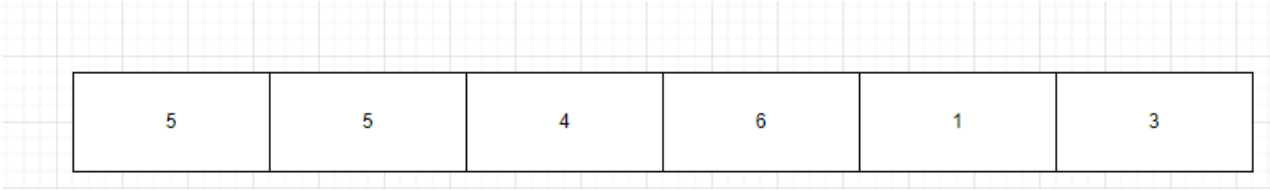

Figure 79: unsorted array

- **Step 2**: Select pivot as the last element in the array as 3. Then perform a comparison of the elements in the array so that the left side will be less than 3 and the right side will be greater than 3. Assign j to the first element element 5 and assign i before the number 5.

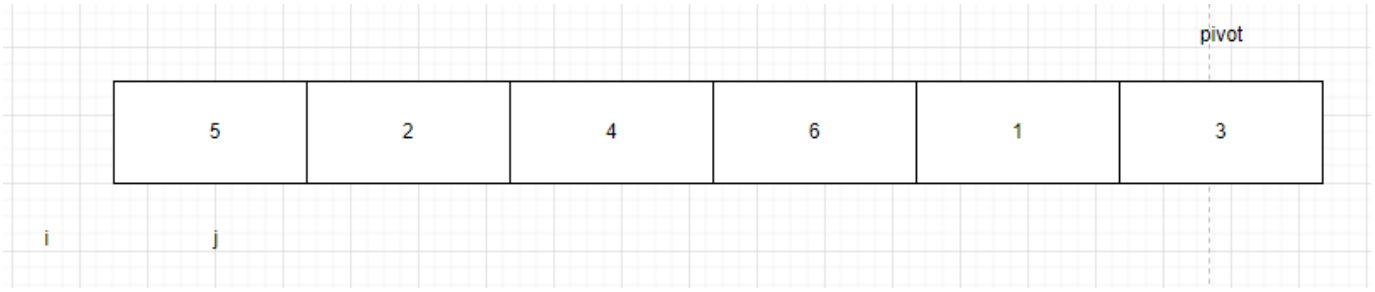

Figure 80: example quick sort 1

- **Step 3**: compare j =5 with pivot =3, then 5>3 should be ignored and continue to move j up 1 element. Then j = 2 and pivot =3, and 2<3. When j<pivot then raise i by one element, this time i = 5 and change position i and j here to 5 and 2.

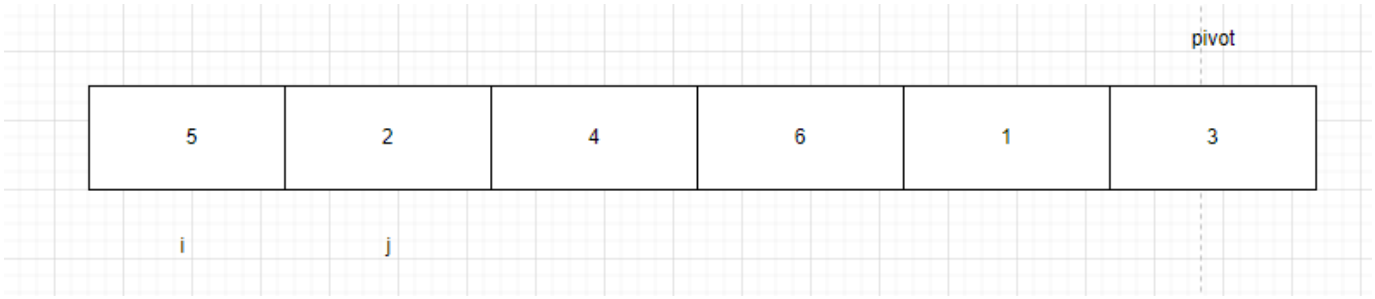

Figure 81: example quick sort 2

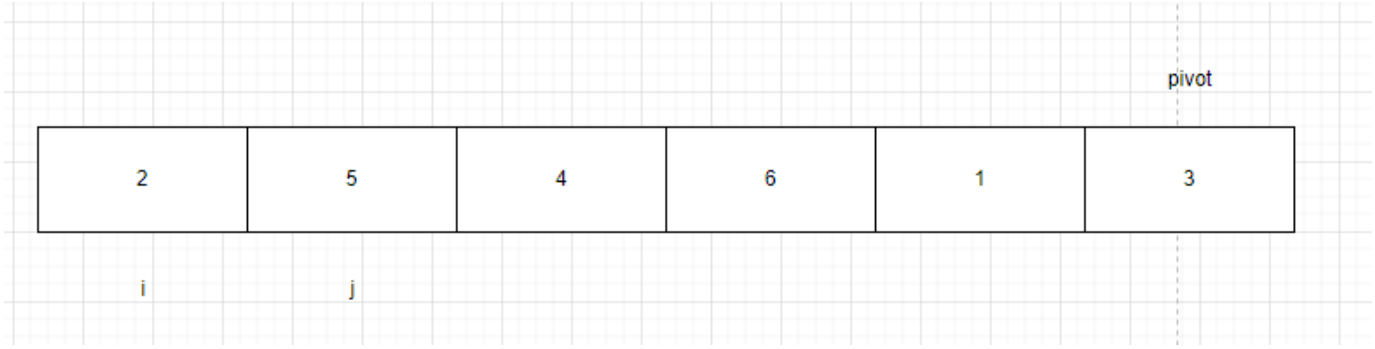

Figure 82: example quick sort 3

- **Step 4**: keep raising until j = 1, at this time 1<3 will increase the position of i by 1 element and change the position of i and j by 5 and 1

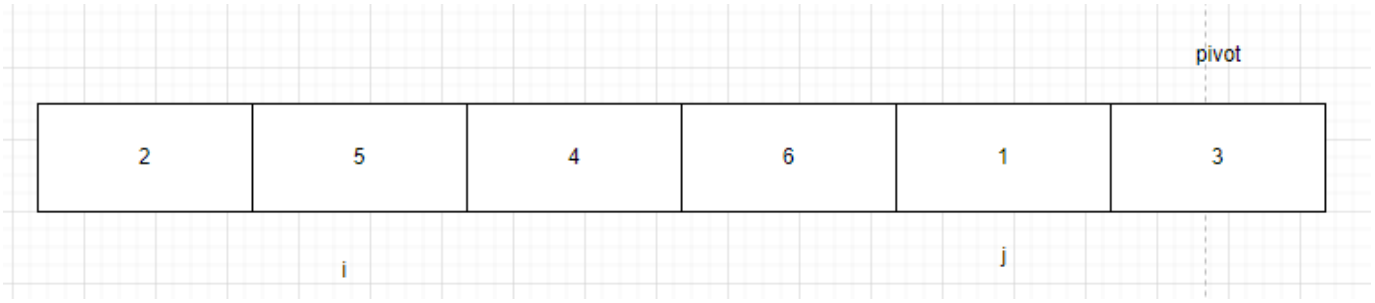

Figure 83: example quick sort 4

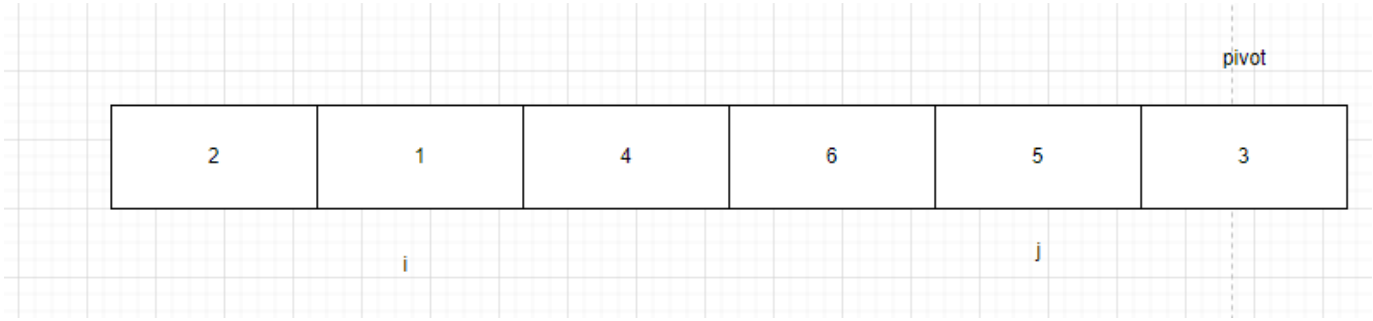

Figure 84: example quick sort 5

- **Step 5**: keep moving j to new position, when j = pivot = 3 then i will be moved up 1 position. Then change the position of j with i being 3 and 4

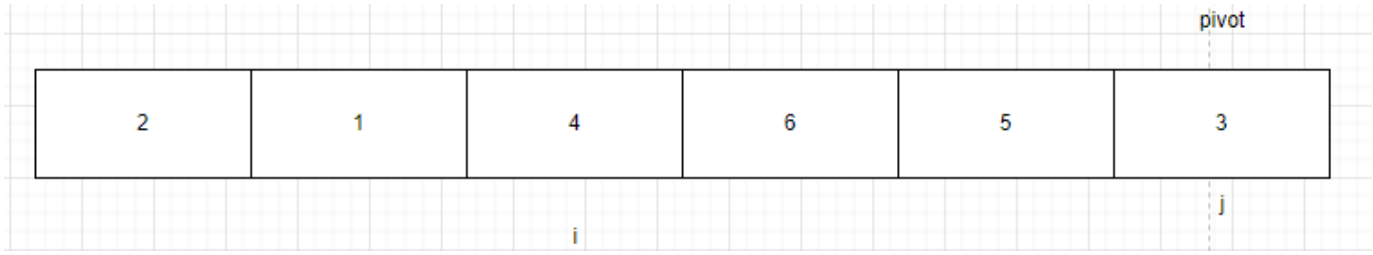

Figure 85: example quick sort 6

After changing positions, we can see, numbers less than 3 will be on the left and numbers higher than 3 will be on the right.

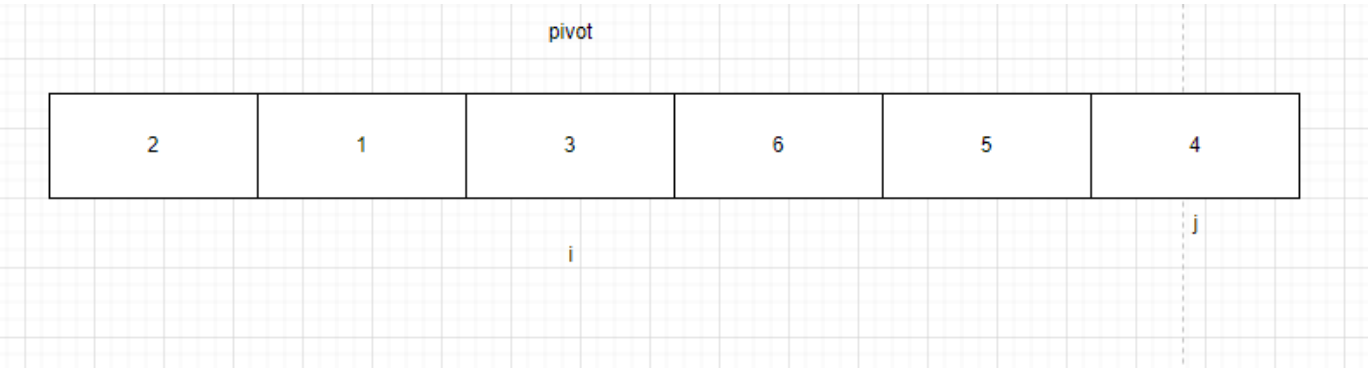

Figure 86: example quick sort 7

- **Step 6**: Continue recursing the left part, set pivot = 1; Then j = 2 and i is in place before the number 2; Compare j=2 > pivot =1, move j up 1 position. Now j =pivot will move i up 1 position. Then change the position of i and j

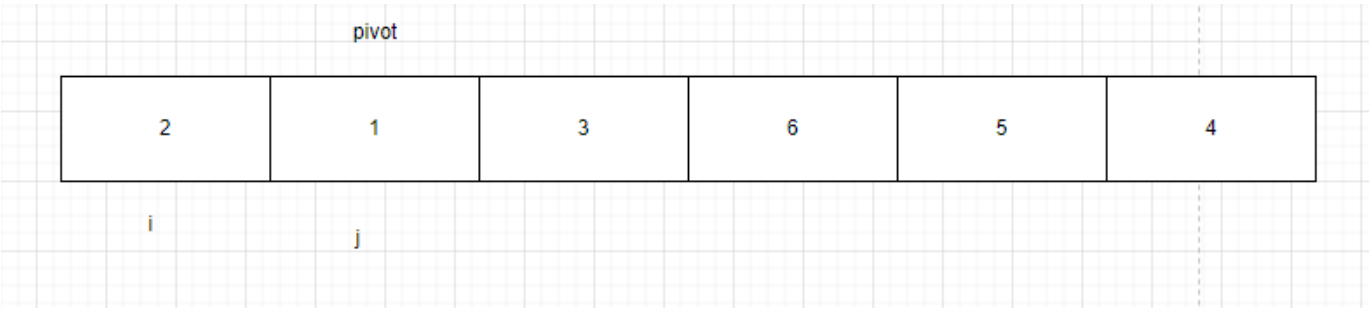

Figure 87: example quick sort 8

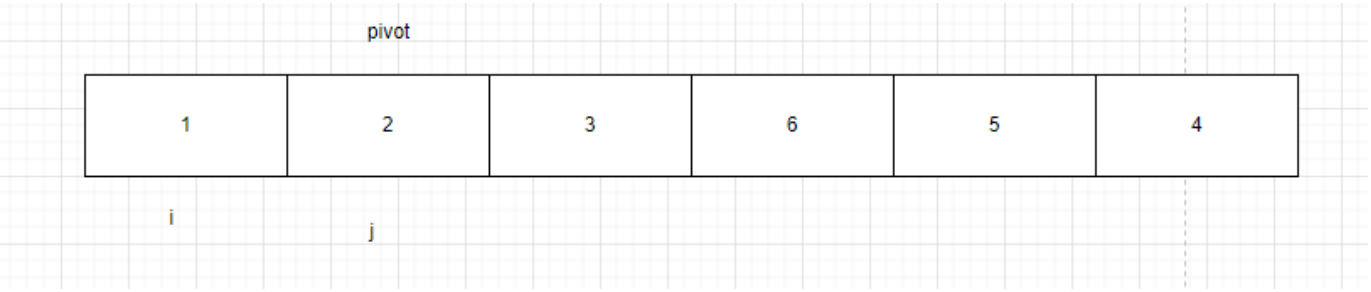

Figure 88: example quick sort 9

Step 7: Continue recursing the left part, having pivot = 4, j = 6 and i = 3. Compare j = 6 > pivot = 4. Continue raising j's position by 1 unit until j < pivot or j = pivot. When j = pivot, move i's position up one unit and swap j and i

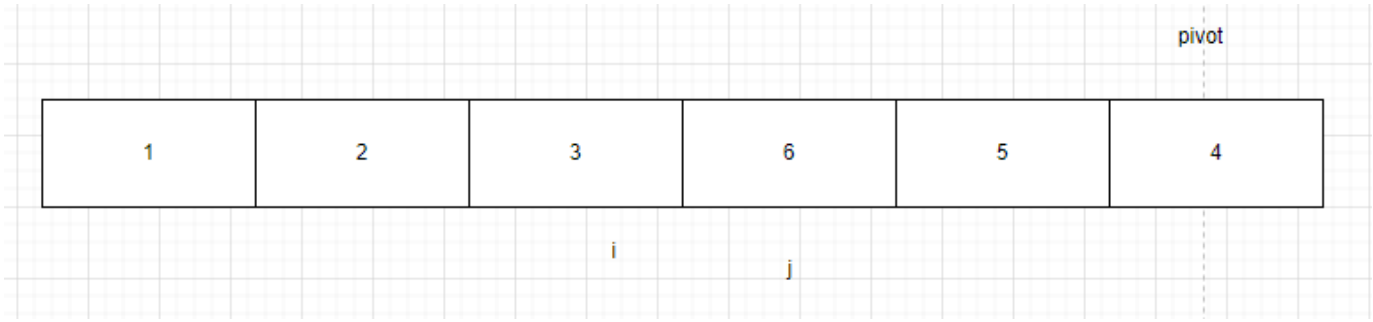

Figure 89: example quick sort 10

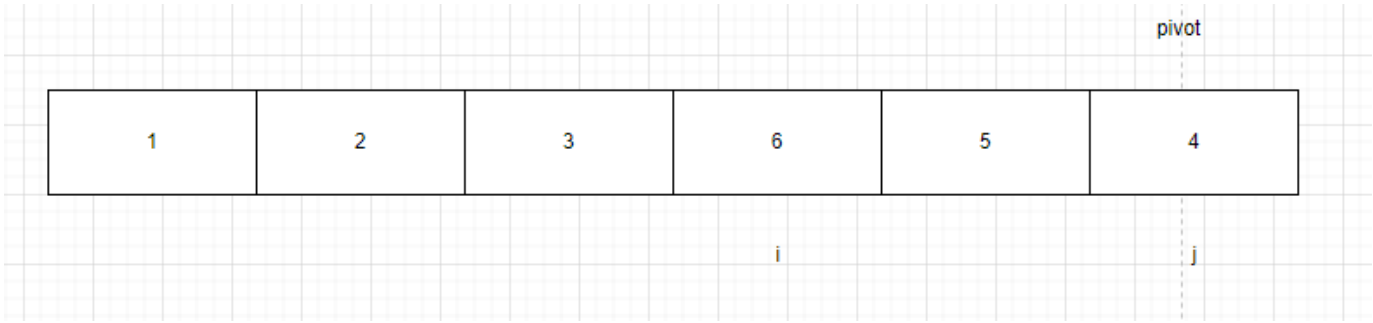

Figure 90: example quick sort 11

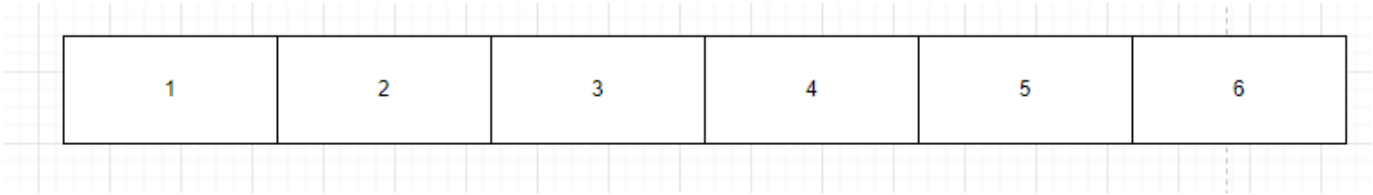

Figure 91: example quick sort 12

After sorting, the array is completely sorted

d. Analyze its performance (time, memory)

a) Time Complexity:

Usually, the execution time of the Quick Sort algorithm depends on the size of the input array and how the data is divided during the sorting process.

- **Worst case**: The worst case execution time is O(n^2), occurs when the array is sorted or near sorted and the pivot is chosen badly, resulting in the division of the array into elements smaller and larger than the unbalanced pivot.
- **Average case**: The average time of Quick Sort is O(n log n). This is the most common case and the average complexity will be more optimal than the worst case.
- **Best case**: In the best case, the execution time of Quick Sort is also O(n log n). This happens when every split of the array divides it into two equal parts.

b) Memory

The average complexity of Quick Sort is O(log n), which corresponds to the height of the recursion tree in the algorithm. However, in the worst case, the space used can be O(n), since the recursion continues in an irregularly smaller side. This can happen if the pivot selection algorithm is not good, resulting in the generation of consecutively smaller pivot elements.

e. Example code

```
import java.util.Arrays;

public class testSort {

    3 usages
    public static void quickSort(int[] arr, int start, int end) {
        if (start < end) {
            // Bước 1: Lấy phần tử chốt là phần tử ở cuối danh sách
            int pivotIndex = partition(arr, start, end);

            // Bước 3: Sử dụng sắp xếp nhanh đệ qui với mảng con bên trái
            quickSort(arr, start,  end: pivotIndex - 1);

            // Bước 4: Sử dụng sắp xếp nhanh đệ qui với mảng con bên phải
            quickSort(arr,  start: pivotIndex + 1, end);
        }
    }
    1 usage
    public static int partition(int[] arr, int start, int end) {
        // Bước 2: Chia mảng bằng phần tử chốt
        int pivot = arr[end];
        int i = start - 1;
        for (int j = start; j <= end - 1; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr,  i: i + 1, end);
        return i + 1;
    }
```

Figure 92: example code quick sort

```
    public static void main(String[] args) {
        int[] arr = {5, 2, 6, 1, 3, 4};
        System.out.println("Before");
        System.out.println(Arrays.toString(arr));
        quickSort(arr, start: 0, end: arr.length - 1);
        System.out.println("After");
        System.out.println(Arrays.toString(arr));
    }
```

Figure 93: main quick sort

Result :

```
Before
[5, 2, 6, 1, 3, 4]
After
[1, 2, 3, 4, 5, 6]
```

Figure 94: result quick sort

VI. Examine the advantages of encapsulation and information hiding when using an ADT(M3)

- **Modularity and abstraction**: Encapsulation enables the outside world to remain unaware of the ADT's implementation specifics. The ADT creates a distinct separation between interface and implementation by encapsulating internal data and operations. As a result, the ADT becomes modular and abstract, which makes it simple to use, maintain, and comprehend.
- **Data protection**: Information encapsulation and concealing prohibit users from having direct access to ADT's internal data from outside sources, protecting data. In order to regulate access to and modification of data, preserve data integrity, and guard against alterations or improper entry of information, ADT may specify particular access methods (getters and setters). destination.
- **Code reuse**: ADT may be used in many sections of a programme or even in many separate programmes without changing its internal settings because to the encapsulation of implementation details. This enhances development efficiency by reducing code duplication and increasing code reuse.
- **Encourage developers to adhere to good programming practises**: Encapsulation and hiding incentivizes programmers to adhere to best practises such providing lucid and consistent interfaces and adhering to the data abstraction principle. Additionally, by separating different components of the programme, this makes the code easier to maintain and lowers the likelihood of problems.

- **Installation adaptability**: Because of encapsulation, changes to the internal data structure of the ADT may be made without impacting its outward interface. As a result, it is possible to change to a more effective data structure or to optimise the implementation without having an impact on the code's other ADT-using sections.
- **Security and stability**: Information masking prohibits external programmes from directly accessing sensitive data or internal configuration specifics, ensuring security and stability. This adds an extra layer of protection to the programme and lowers the possibility of vital data being accessed or altered without authorization. By lowering the risk of unneeded side effects brought on by direct data modifications, it also increases code stability.

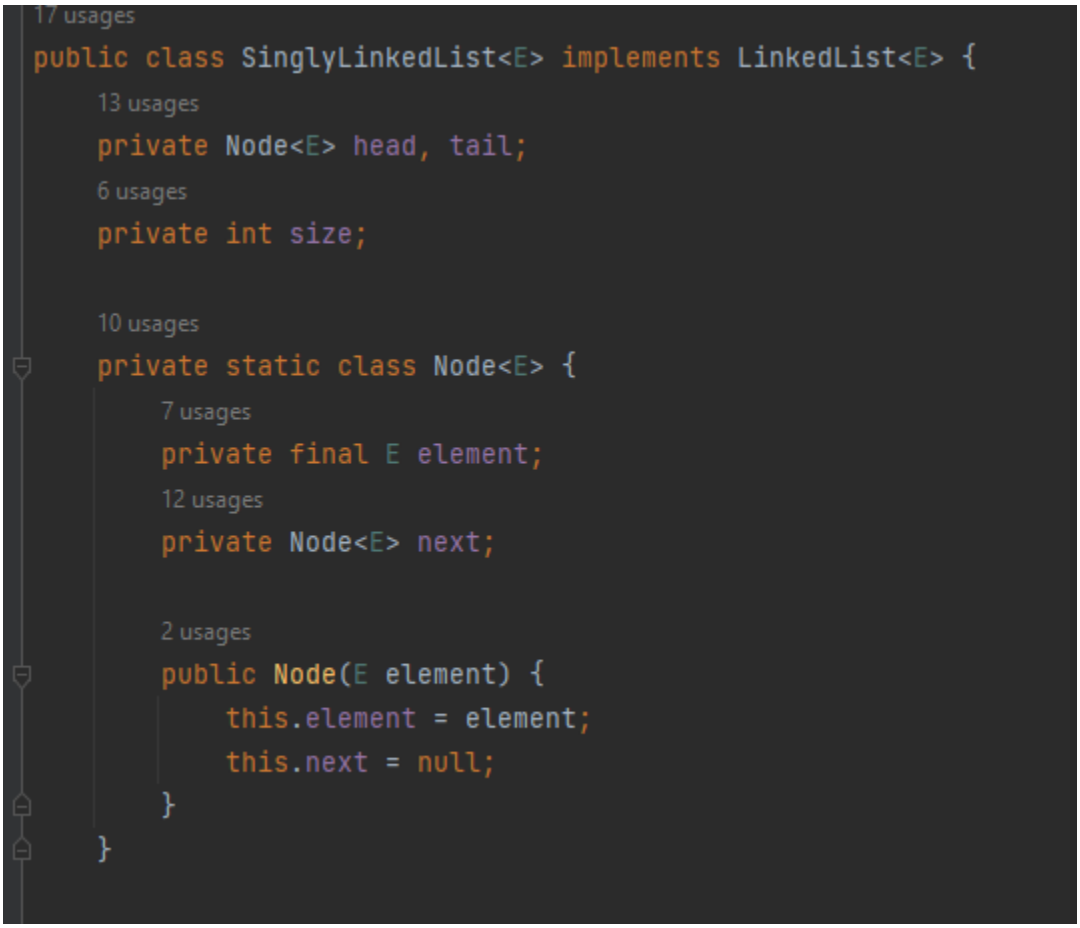

Figure 95: encapsulation in ADT

VII. References

Anon (2023a) Abstract data types, GeeksforGeeks, GeeksforGeeks, [online] Available at: https://www.geeksforgeeks.org/abstract-data-types/ (Accessed 2 August 2023).

Anon (2023b) Abstract data types, GeeksforGeeks, GeeksforGeeks, [online] Available at: https://www.geeksforgeeks.org/abstract-data-types/ (Accessed 2 August 2023).

Anon (2023c) Applications, advantages and disadvantages of Stack, GeeksforGeeks, GeeksforGeeks, [online] Available at: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-stack/ (Accessed 2 August 2023).

Anon (2023d) Introduction to doubly linked list – data structure and algorithm tutorials, GeeksforGeeks, GeeksforGeeks, [online] Available at: https://www.geeksforgeeks.org/data-structures/linked-list/doubly-linked-list/ (Accessed 2 August 2023).

Anon (n.d.) Cấu Trúc dữ Liệu Hàng đợi (queue), Quantrimang.com - Kiến Thức Công Nghệ Khoa Học và Cuộc sống, [online] Available at: https://quantrimang.com/cong-nghe/cau-truc-du-lieu-hang-doi-queue-156381 (Accessed 2 August 2023a).

Anon (n.d.) Cấu Trúc dữ Liệu Ngăn xếp (stack), VietTuts, [online] Available at: https://viettuts.vn/cau-truc-du-lieu-va-giai-thuat/cau-truc-du-lieu-ngan-xep-stack (Accessed 2 August 2023b).

Anon (n.d.) Giải Thuật sắp xếp chèn (insertion sort), Quantrimang.com - Kiến Thức Công Nghệ Khoa Học và Cuộc sống, [online] Available at: https://quantrimang.com/cong-nghe/giai-thuat-sap-xep-chen-insertion-sort-156393 (Accessed 2 August 2023c).

ITNavi (2022) Quick sort LÀ GÌ? Tất Tần Tật TỪ A-Z về thuật toán sắp xếp nhanh, ITNavi, ITNavi, [online] Available at: https://itnavi.com.vn/blog/quick-sort-la-gi (Accessed 2 August 2023).