

## ASSIGNMENT 1 FRONT SHEET

<b>Qualification</b>	<b>BTEC Level 5 HND Diploma in Computing</b>		
<b>Unit number and title</b>	Unit 19: Data Structures and Algorithms		
<b>Submission date</b>	2/8/2023	<b>Date Received 1st submission</b>	
<b>Re-submission Date</b>	9/8/2023	<b>Date Received 2nd submission</b>	
<b>Student Name</b>	Vo Dinh Nhat	<b>Student ID</b>	GBD210218
<b>Class</b>	GCD1101	<b>Assessor name</b>	Pham Thanh Son
<b>Student declaration</b> <p>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.</p>			
		<b>Student's signature</b>	Nhat

### Grading grid

P1	P2	P3	M1	M2	M3	D1	D2

☐ Summative Feedback:

☐ Resubmission Feedback:

Grade:

Assessor Signature:

Date:

Internal Verifier's Comments:

IV Signature:

## Contents

<b>Chapter 1: Create a design specification for data structures explaining the valid operations that can be carried out on the structures(P1).</b>	<b>5</b>
I. Abstract data type.....	5
1. Definition .....	5
2. Common ways to represent ADT .....	5
3. Examples .....	9
<b>Chapter 2: Determine the operations of a memory stack and how it is used to implement function calls in a computer(P2).</b>	<b>14</b>
I. Application of Stack in memory.....	14
1. How the memory is organized.....	14
2. Operations .....	15
3. How a method (function) calls is implemented with stack .....	19
<b>Chapter 3: Create a design specification for data structures explaining the valid operations that can be carried out on the structures(P3).</b>	<b>22</b>
I. Application of an ADT (P3):.....	22
1. Senerio .....	22
2. Basic Operations on linked list.....	22
3. Part of a software stack .....	22
4. Five software stack examples .....	22
5. The problem-specific use of Data Structure .....	23
<b>REFERENCES:</b>	<b>29</b>

## **FIGURE**

Figure 1: Abstract Data Type .....	5
Figure 2: : Singly Linked List .....	6
Figure 3: Doubly Linked List .....	7
Figure 4: Circular Linked List .....	7
Figure 5: Stack.....	8
Figure 6: Queue .....	8
Figure 7: Queue .....	9
Figure 8: Table class Queue .....	11
Figure 9: Interface AbstractQueue .....	11
Figure 10: Offer() and Poll() .....	12
Figure 11: ensureNonEmpty() and Peek() .....	12
Figure 12: isEmpty() .....	12
Figure 13: Main function.....	13
Figure 14: Result .....	13
Figure 15: Computer System Memory.....	14
Figure 16: Demonstrations of stack behavior.....	15
Figure 17: Push operation of Stack.....	16
Figure 18: Pop operation of Stack .....	17
Figure 19: Peek operation of Stack.....	18
Figure 20: isEmpty operation of Stack.....	19
Figure 21: Program of calculate the sum of the odd numbers of 6.....	19
Figure 22: Result .....	20
Figure 23: The sum of the odd numbers of 6 (1) .....	20
Figure 24: The sum of the odd numbers of 6 (2) .....	21
Figure 25: Problem - EMployee Management.....	22
Figure 26: Source code 1.....	24
Figure 27 : Source code 2.....	25
Figure 28: Source code 3.....	25
Figure 29: Source code 4.....	26
Figure 30: Source code 5.....	27
Figure 31: Source code 6.....	27
Figure 32: Output 1.....	27
Figure 33: Output 2.....	27
Figure 34: Output 3.....	28
Figure 35: Output 4.....	28

# Chapter 1: Create a design specification for data structures explaining the valid operations that can be carried out on the structures(P1).

## I. Abstract data type

### 1. Definition

Programmers must work with various data types, such as int, float, double, char, string, and others, in programming languages like Java, C, C++, and Python. These data types are typically associated with specific byte sizes, such as 4 bytes for int and 1 byte for character types. However, during programming, these data types are often abstracted or simplified for the user's convenience. In languages like Java, C, C++, and Python, programmers frequently manipulate different data types like int, float, double, char, string, and more. These data types generally have predetermined byte sizes, for example, an int occupies 4 bytes and character types occupy approximately 1 byte. However, throughout the programming process, these data types are often abstracted or simplified for the user to make them easier to understand.

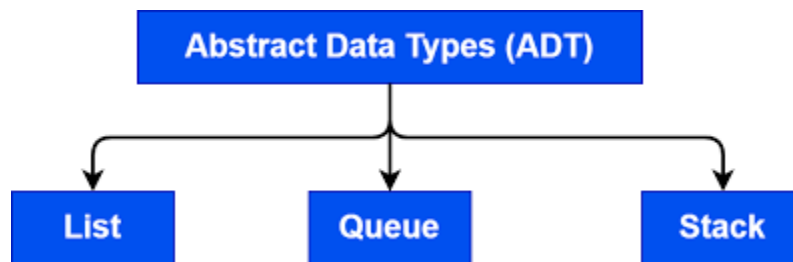


Figure 1: Abstract Data Type

### 2. Common ways to represent ADT

- **Using ArrayList**

An ArrayList is a dynamic array that can be resized. It automatically expands to accommodate more elements and contracts when elements are removed. Internally, an ArrayList stores its elements in an array. Similar to regular arrays, elements in an ArrayList can be accessed using an index. The Java ArrayList class supports duplicate and null values. It is an ordered collection, which means it maintains the order of insertion for its items.

However, it's important to note that ArrayList cannot directly store primitive types like int, char, etc. Instead, you need to use their corresponding boxed types such as Integer, Character, Boolean, etc.

It's worth mentioning that Java ArrayList is not synchronized, meaning it is not thread-safe. When multiple threads attempt to modify an ArrayList concurrently, the final outcome becomes unpredictable. If you have multiple threads modifying an ArrayList, you should explicitly synchronize access to it to ensure thread safety.

Here are the explanations for the methods you mentioned:

- `add()`: Adds an element to the list.

- `addAll()`: Adds all the elements of one list to another.
- `get()`: Retrieves an element from the list based on its index. It allows random access to elements.
- `iterator()`: Returns an iterator object that can be used to sequentially access the elements of a list.
- `set()`: Modifies the value of an element at a specific index in the list.
- `remove()`: Removes a specific element from the list.
- `removeAll()`: Removes all elements from the list.
- `clear()`: Removes all elements from the list. This method is more efficient than `removeAll()`.
- `size()`: Returns the number of elements in the list.
- `toArray()`: Converts a list into an array.
- `contains()`: Checks if the list contains a specified element and returns true if it does.

## • Using Singly Linked List

A Singly Linked List is a type of linked list consisting of a sequence of nodes, where each node contains data and a link to the next node. It can be traversed from the first node of the list (also referred to as the "head") to the last node of the list (also known as the "tail") and supports movement in only one direction. The above figure illustrates a singly linked list. Each element in the list is called a "node". A node is composed of two parts: data and a pointer.

Data represents the information stored in the node, and the pointer is the link to the next node in the list.

The first node in the list is referred to as the "head". The last node in the list is the "tail", and it points to NULL.

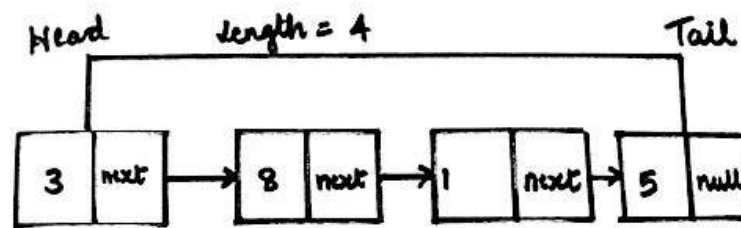


Figure 2: : Singly Linked List

## • Using Doubly Linked List

A Doubly Linked List is a type of linked list consisting of a sequence of nodes where each node contains data and two pointers: the previous pointer and the subsequent pointer. The previous pointer points to the previous node that is part of the list, while the subsequent pointer points to the next node that is part of the list. This type of linked list can be traversed from the first node to the last node and vice versa. The above figure illustrates a

doubly linked list. Data represents the information stored in the node, and each node contains two pointers: the previous pointer and the next pointer. The previous pointer points to the previous node that is part of the list.

The next pointer points to the subsequent node on the list.

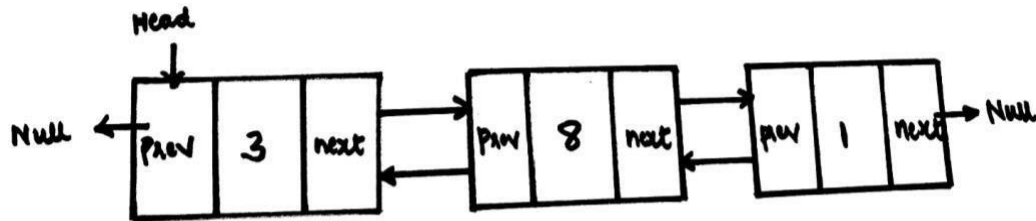


Figure 3: Doubly Linked List

### • Using Circular Linked List

A circular linked list is a type of linked list where the last node points to the first node in the list, forming a circular chain. This allows for traversal of the list in any direction (forward or backward) from any node until we reach the same node we started from. As a result, a circular linked list does not have a distinct beginning or end.

The image below illustrates an example of a circular linked list:

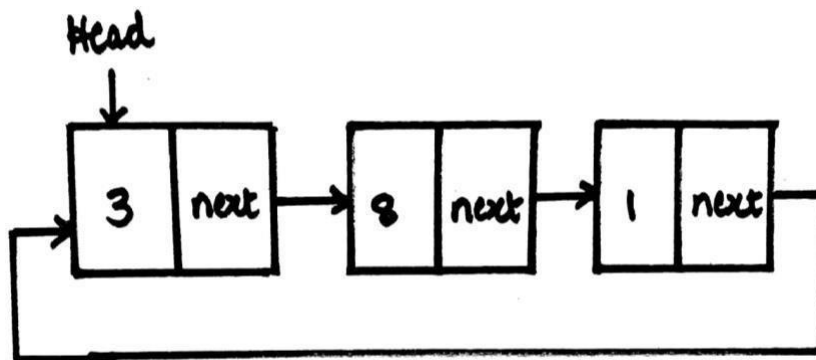


Figure 4: Circular Linked List

Each node in the list contains data and a pointer to the next node in the list, with the exception of the last node, which points back to the first node. Circular linked lists can be utilized in various data structures, including queues, scheduling algorithms, and circular buffers.

### • Using Stack

A stack is an abstract data structure that follows the last in, first out (LIFO) principle. Elements are added and removed from the top of the stack. It supports push and pop operations, and a peek operation to view the top element without removing it. Stacks are commonly used in programming and applications for managing function calls, temporary variables, and performing undo and redo operations.

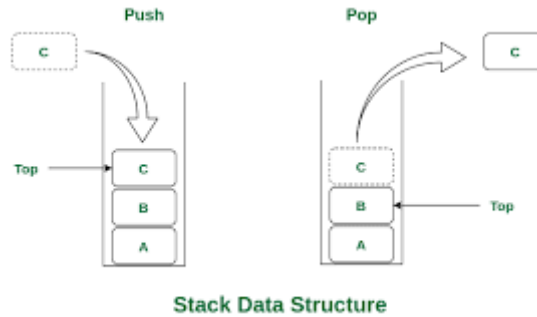


Figure 5: Stack

Here are the methods of a stack data structure:

- `empty()`: This method checks if the stack is empty and returns true if it is, and false if it contains elements.
- `peek()`: This method returns the element at the top of the stack without removing it.
- `pop()`: This method removes and returns the element at the top of the stack.
- `push()`: This method adds an element to the top of the stack and returns the added element.
- `search()`: This method searches for an element in the stack. If the element is found, it returns its distance from the top of the stack. Otherwise, it returns -1.

## 2.6. Using QUEUE

A queue is an abstract data structure that is similar to a real-life queue or line. Unlike a stack, a queue is open on both ends. One end is used for inserting data (also known as enqueueing), while the other end is used for removing data (dequeueing). The queue data structure follows the First-In-First-Out (FIFO) approach, which means that the data that is inserted first is the first to be accessed. There are many real-life examples of queues, such as lines of cars on a one-way street (especially during traffic jams), where the first car to enter is the first to leave. Other examples include queues for students, tickets, and so on.

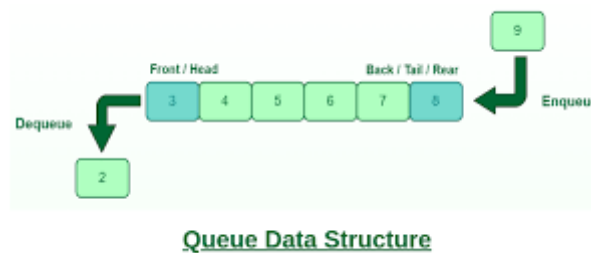


Figure 6: Queue

Here are the methods of a queue data structure:

- `enqueue()`: This method adds an element to the end of the queue.
- `dequeue()`: This method removes and returns the element at the front of the queue.



- peek(): This method returns the element at the front of the queue without removing it.
- isFull(): This method checks if the queue is full and returns true if it is, and false if it is not.
- isEmpty(): This method checks if the queue is empty and returns true if it is, and false if it is not.

### 3. Examples

- **Queue**

A queue is a data structure that stores items based on the "first in, first out" (FIFO) principle. Objects can be added to the queue at any time, but only the first object added can be removed from the queue. Adding an object always occurs at the end of the queue, while removing an object always occurs at the front of the queue. These operations are referred to as "enqueue" and "dequeue," respectively. The queue data structure has various applications in computing, including recursion elimination, breadth-first and backtracking algorithms, exhaustive search, job scheduling, and keyboard buffer management.

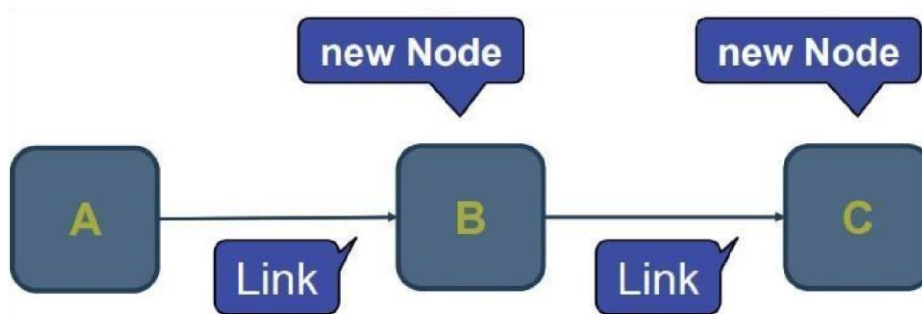


Figure 7: Queue

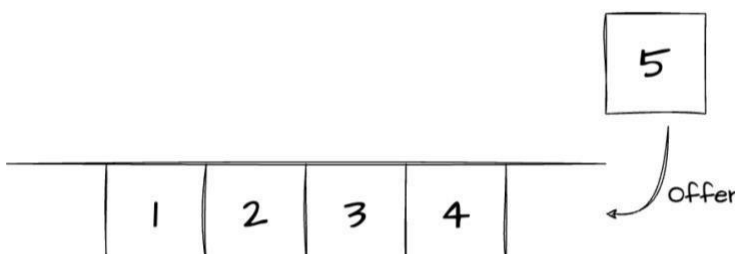
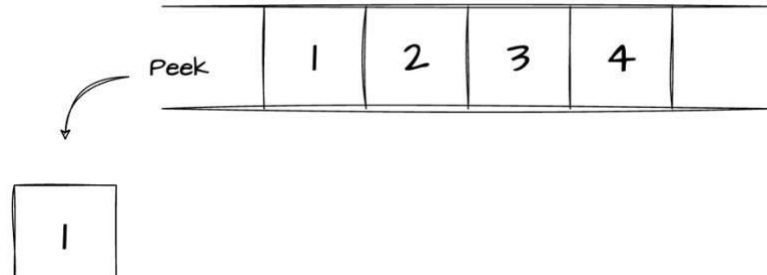
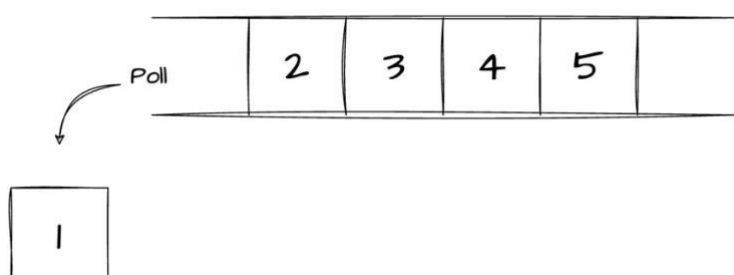
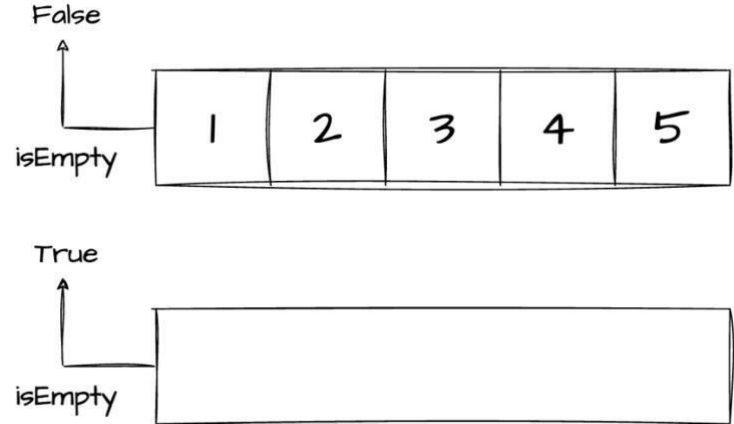
#### Advantages of Queue:

- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
- Queues are useful when a particular service is used by multiple consumers.
- Queues are fast in speed for data inter-process communication.
- Queues can be used in the implementation of other data structures.

#### Disadvantages of Queue:

- Operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching for an element takes  $O(N)$  time.
- The maximum size of a queue must be defined prior.

➤ **Class Queue:**

Method	Description	Illustrations
Offer	Add an element to the end of the queue	
Peek	Get the value of the queue's front without removing it.	
Poll	Move an element to the front of the queue	
isEmpty	Determine whether the queue is empty	

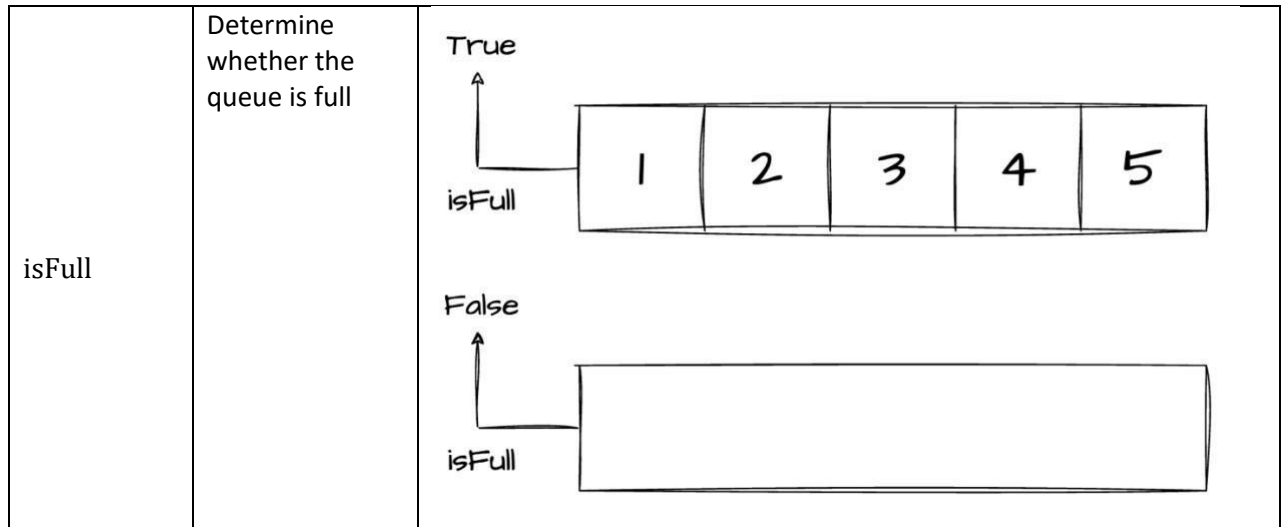


Figure 8: Table class Queue

```
package interfaces;

public interface AbstractQueue<E> extends Iterable<E> {
    void offer(E element);
    E poll();
    E peek();
    int size();
    boolean isEmpty();
}
```

Figure 9: Interface AbstractQueue

We construct the AbstractQueue interface, which has the operations Offer(), Poll(), Peek(), isEmpty(), and isFull().

```

@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++;
}

@Override
public E poll() {
    ensureNonEmpty();
    E element = this.head.element;
    if (this.size == 1) {
        this.head = null;
    } else {
        Node<E> next = this.head.next;
        this.head.next = null;
        this.head = next;
    }
    this.size--;
    return element;
}

```

Figure 10: Offer() and Poll()

```

private void ensureNonEmpty() {
    if (this.head == null) {
        throw new IndexOutOfBoundsException(String.format("Element have to not null"));
    }
}

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

```

Figure 11: ensureNonEmpty() and Peek()

```

@Override
public boolean isEmpty() {
    return false;
}

```

Figure 12: isEmpty()

```
private static void testQueue() {  
    Queue<Integer> numbers = new LinkedList<>();  
  
    // offer elements to the Queue  
    numbers.offer(1);  
    numbers.offer(2);  
    numbers.offer(3);  
    numbers.offer(4);  
    numbers.offer(5);  
    System.out.println("Queue: " + numbers);  
    System.out.println("The element at the current front of the queue: " + numbers.peek());  
    // Remove elements from the Queue  
    numbers.poll();  
    System.out.println("After Removed Element: " + numbers);  
}
```

Figure 13: Main function

The main function is used to put the Queue class's functions to the test.

```
Queue: [1, 2, 3, 4, 5]  
The element at the current front of the queue: 1  
After Removed Element: [2, 3, 4, 5]
```

Figure 14: Result

## Chapter 2: Determine the operations of a memory stack and how it is used to implement function calls in a computer(P2).

### I. Application of Stack in memory

#### 1. How the memory is organized

Memory is physically organized as a plethora of cells, each of which can store one bit. They are organized logically as groups of bits called words, each with its unique address. These memory addresses are utilized to retrieve information and instructions.

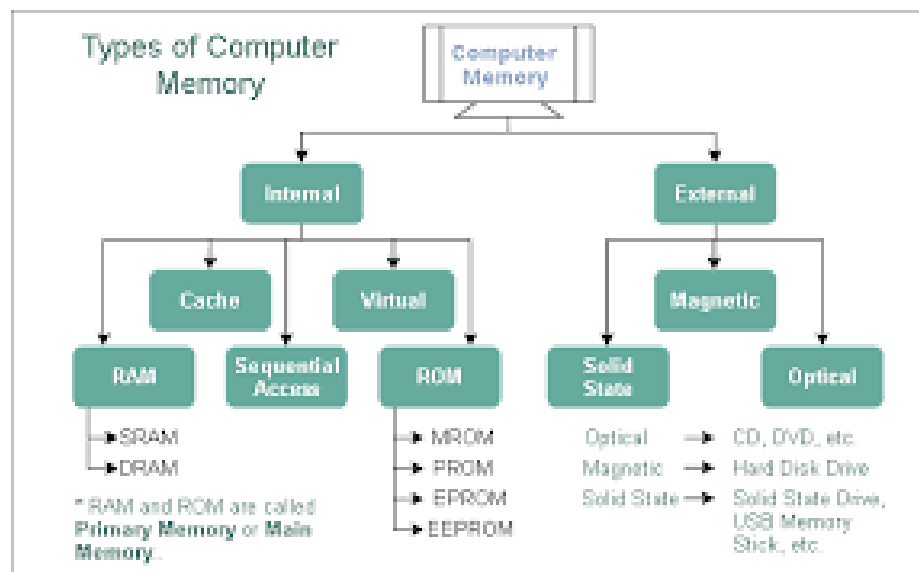


Figure 15: Computer System Memory

To clarify, when an operating system executes a program, it first loads it into the main memory. Memory is used to store both computer instructions and data used by software. The software divided its memory into portions that performed different program functions. Although paged memory has mostly replaced this memory management method, applications continue to organise their memory using the functional units described below.

Paged memory computers manage memory dynamically, allowing the amount of memory given to a program to alter as the program's demands change. Memory is allocated to the program in fixed-size chunks known as pages and reclaimed by the operating system. When you load a software onto a paged-memory computer, the operating system first allocates a small number of pages to the application and then allocates additional memory as needed. Pages holding machine code and data that have not been used in a while may be returned to the OS if they contain machine code and data that are not immediately required.

- An application of Stack in memory management

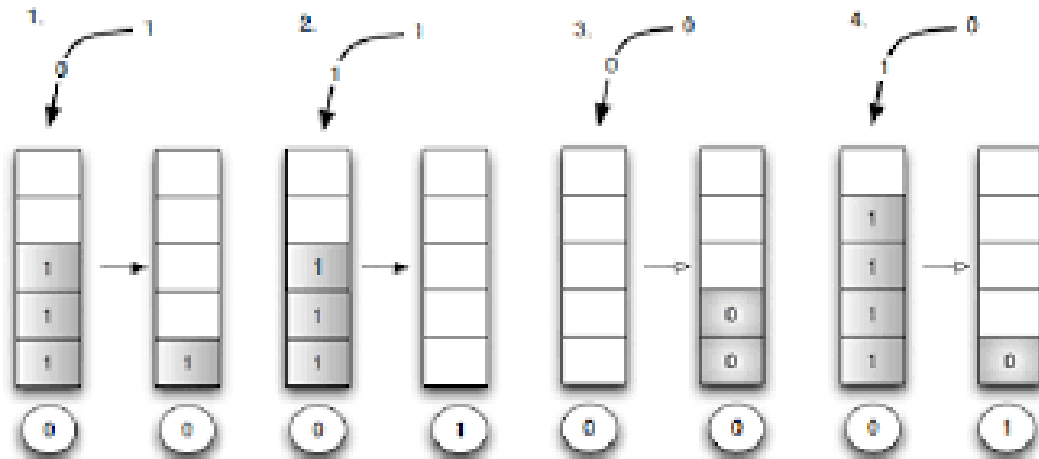


Figure 16: Demonstrations of stack behavior

## 2. Operations

In a stack, elements of the same type are placed progressively. The top of the stack is the one end where all activities are carried out. Working on the LIFO (Last In, First Out) principle. Furthermore, the following stack operations are possible

- **PUSH:**

The PUSH procedure involves inserting a new element into a Stack. Because a new element is always introduced from the topmost position of the Stack, we must always check to see if the top is empty, i.e.,  $TOP = Max - 1$ . If this condition is false, the Stack is full and no more elements can be put; if we try to insert the element, a Stack overflow notice is displayed.

Algorithm:

Step 1: If  $TOP = Max - 1$

Print "Overflow"

Goto Step 4

Step 2: Set  $TOP = TOP + 1$

Step 3: Set  $Stack[TOP] = ELEMENT$

Step 4: END

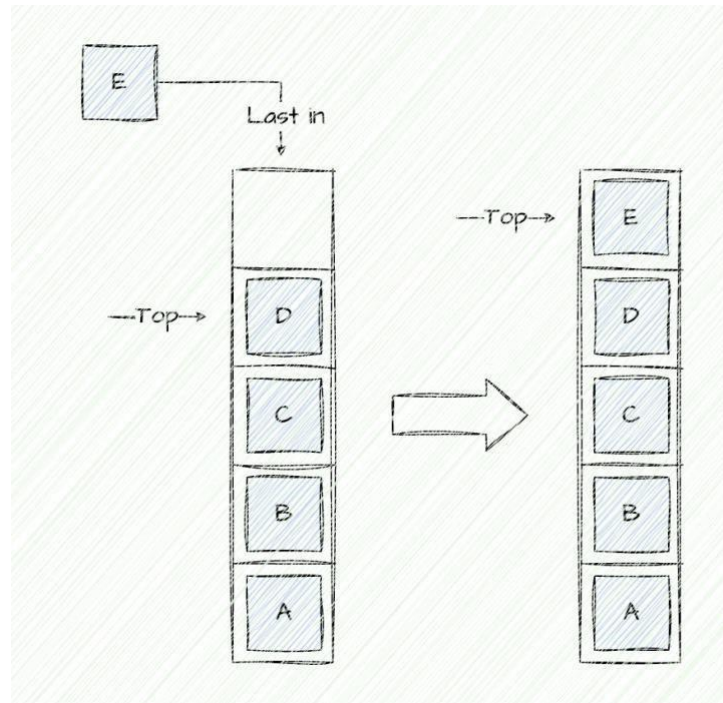


Figure 17: Push operation of Stack

- **POP:**  
POP denotes the removal of an element from the Stack. Before deleting an element, ensure that the Stack Top, i.e., TOP=NULL, is not NULL. If this condition is met, the Stack is empty, and no deletion operations may be performed. If we try to delete, the Stack underflow notification is generated.  
Algorithm:

Step-1: If TOP= NULL

Print "Underflow"

Goto Step 4

Step-2: Set VAL= Stack[TOP]

Step-3: Set TOP= TOP-1

Step-4: END



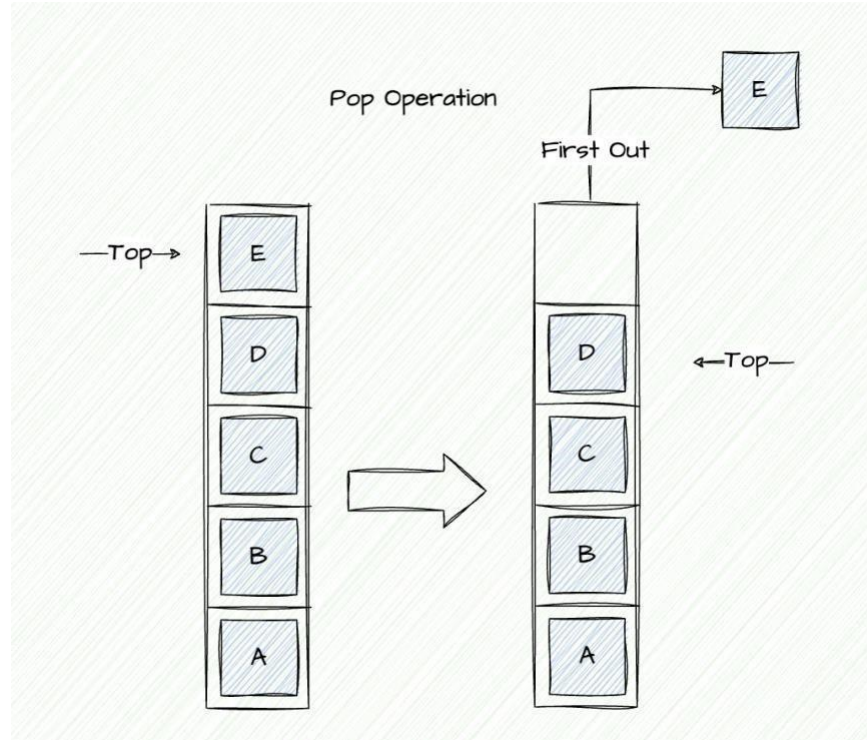


Figure 18: Pop operation of Stack

- PEEK:

The Peek action is utilized when we need to return the value of the topmost element of the Stack without deleting it from the Stack. This operation first determines whether the Stack is empty, i.e., TOP = NULL; if so, an appropriate message is displayed; otherwise, the value is returned.

Algorithm:

Step-1: If TOP = NULL PRINT "Stack is Empty" Goto Step 3

Step-2: Return Stack[TOP]

Step-3: END

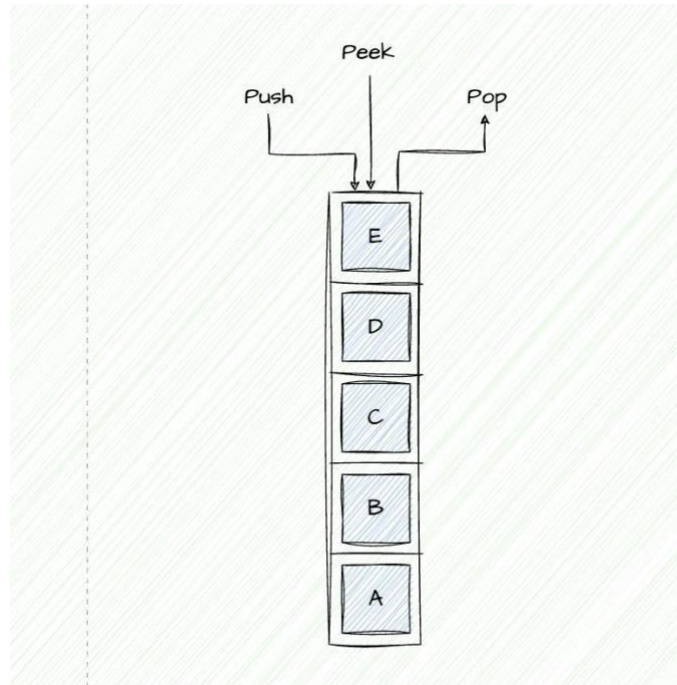


Figure 19: Peek operation of Stack

- isEmpty

When we are not to avoid activities from being conducted on an empty stack, the programmer must internally keep the stack size, which will be updated for push and pop actions. isEmpty() normally returns a boolean value. If size is zero, true; otherwise, false.

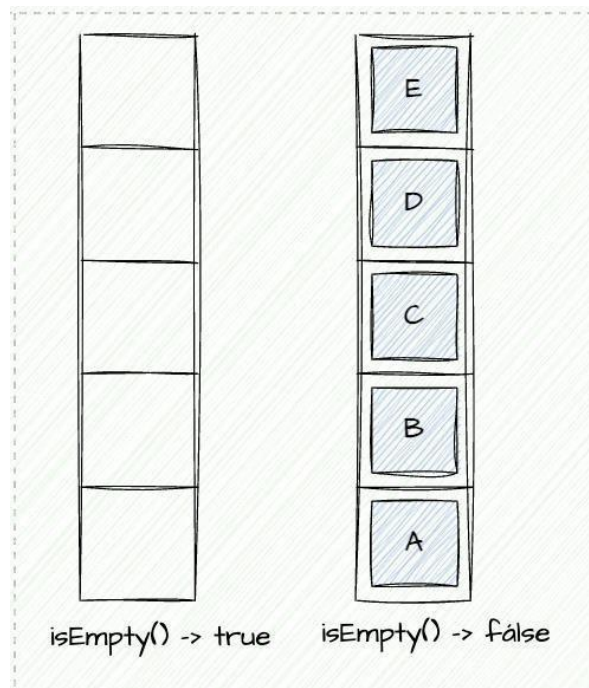


Figure 20: isEmpty operation of Stack

### 3. How a method (function) calls is implemented with stack

- The method-call stack, also known as the program-execution stack, is a data structure used to implement the call/return mechanism in programming. It enables the creation, maintenance, and deletion of local variables for each called method. The stack's Last In First Out (LIFO) behavior is essential for a method to return to the calling method. As methods call other methods, the system must keep track of the return addresses for each method. The method call stack is the optimal data structure for this purpose. An activation record, which includes the return address and additional information, is added to the stack every time one method calls another.
- When a called method returns without calling another method, the stack frame for the method call is popped, and control is sent to the return address in the popped stack frame. The call stack is also useful because every called method can always locate the data it needs to return to its caller at the top of the stack. When one method calls another, a stack frame for the new method call is inserted into the call stack, and the newly called method's return address is now at the top of the stack.
- It is crucial to provide a function access to its stack frame during execution, as it is necessary for the function to operate. An example of how a method uses a stack is shown in a short program that computes the sum of an integer's odd integers.

```
public class CallStack {
    public static int sole (int n) {
        if(n == 1) {
            return 1;
        } else if (n % 2 == 0) {
            return sole (n - 1);
        } else {
            return n + sole (n - 2);
        }
    }

    public static void main (String[] args) {
        System.out.println("x: sole ( n: 6)");
    }
}
```

Figure 21: Program of calculate the sum of the odd numbers of 6

When I add the odd numbers to 6, this is what the software produced.

```
-----[ jar ]-----

--- exec-maven-plugin:3.0.0:exec (default-cli) @ asml ---
9
-----

BUILD SUCCESS
```

Figure 22: Result

When the value of  $n$  is greater than 1, the recursive function is used, and the value of  $n$  is reduced by one if it is divisible by two and by two if it is not. This process continues until the base case is reached, which is when  $n$  equals 1. At this point, the return value from each function call is passed on to the preceding function call, and so on, until the original function call is reached, and the result is computed.

The base case is essential to ensure that there are no additional recursive calls beyond a certain point. Without a base case, the function would keep calling itself indefinitely, leading to a stack overflow and eventual program failure. Therefore, a base case is required for the recursive function to work correctly.

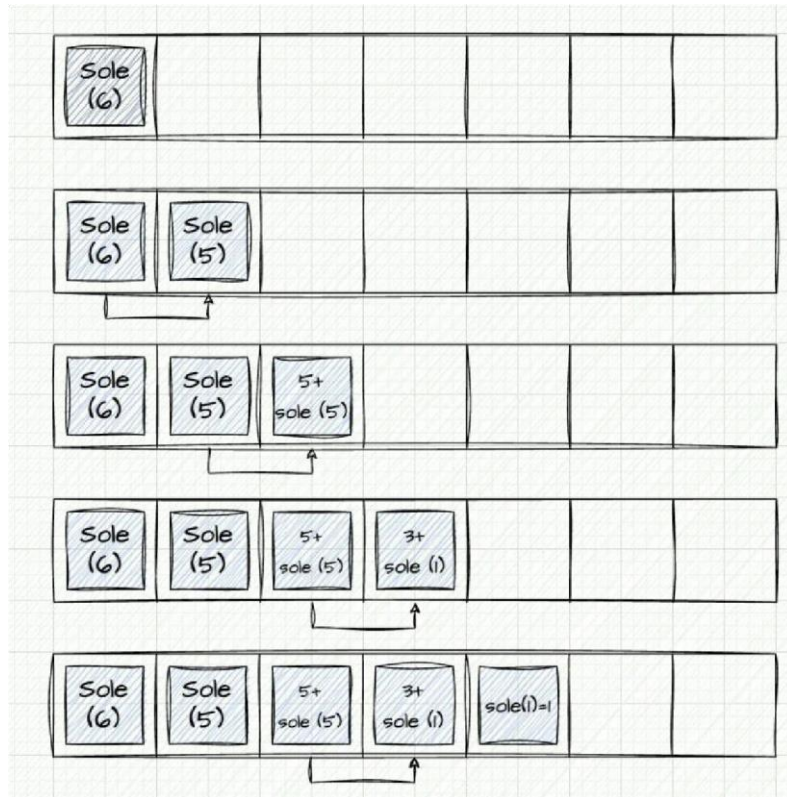


Figure 23: The sum of the odd numbers of 6 (1)

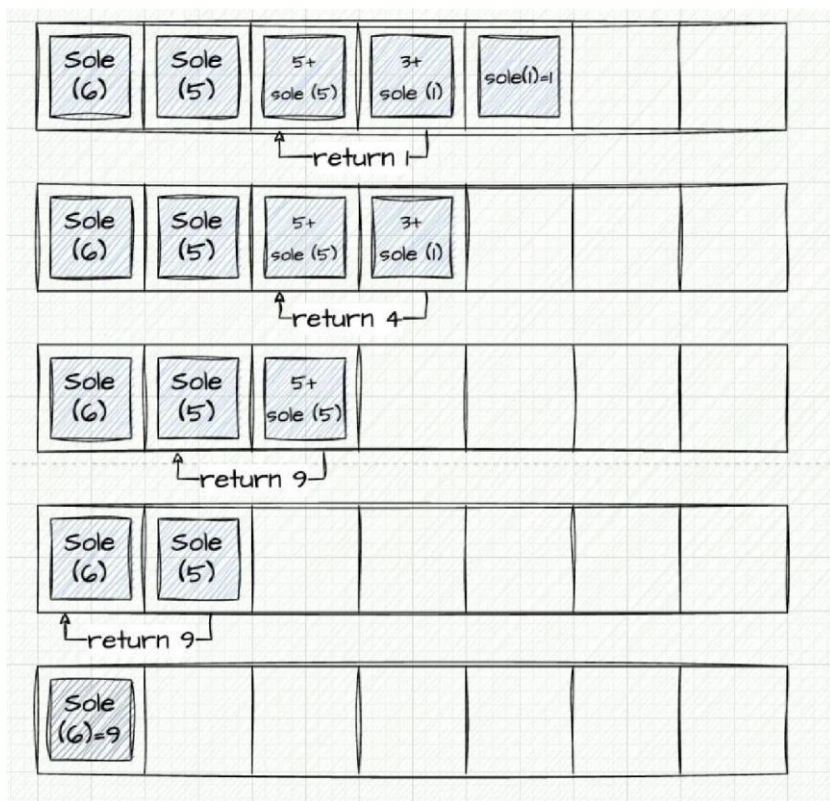


Figure 24: The sum of the odd numbers of 6 (2)



## Chapter 3: Create a design specification for data structures explaining the valid operations that can be carried out on the structures(P3).

### I. Application of an ADT :

#### 1. Senerio

It is true that managing personnel is usually a time-consuming and labor-intensive task. This work is extremely difficult and time-consuming if you do not use software and handle a personnel list entirely on paper. So, in this time's application section of ADT, I'll create a software to manage employees of a small firm utilizing Linked List to tackle such challenges.



Figure 25: Problem - Employee Management

#### 2. Basic Operations on linked list

- Traversal: The process of going from one node to the next.
- Insertion: The addition of a node at the specified place.
- Deletion: The removal of a node.
- Searching: Finding an element(s) based on its value.
- Updating: The act of updating a node.
- Sorting: The process of arranging nodes in a linked list in a certain order.
- Merging: Consolidating two linked lists into one.

#### 3. Part of a software stack

Depending on the intended application functionality, software stacks can be basic or complex, and can include components and services from an organization's on-premises resources, third-party providers, or cloud providers. There is no minimum standard for the components and services that must be included in a software stack, except that their features and functions enable the development, delivery, and operation of an application.

#### 4. Five software stack examples

- **LAMP (Linux, Apache, MySQL, PHP):**

This is a well-known web development software stack. The Linux operating system, which interfaces with the Apache web server, is at the bottom of the stack's hierarchy. The scripting language, in this case PHP, is at the top of the hierarchy. (The "P" might alternatively stand for Python or Perl, two programming languages.) Because the components are all open source and the stack can run on commodity hardware, LAMP stacks are popular. A LAMP stack is loosely connected, as opposed to monolithic software stacks, which are frequently tightly tied and built for a specific operating system. This simply indicates that, while the components were not planned to function together in the beginning, they have proven to be complementary and are frequently utilized together. LAMP components are currently present in practically all Linux variants. To establish a LAPP stack, developers can replace MySQL with PostgreSQL. Another variation of this open-source software stack is the LEAP stack (Linux, Eucalyptus, AppScale, Python), which is used for cloud-based development and service delivery.

- **MEAN (MongoDB, Express, Angular, Node.js):**

This set of development tools aids in the removal of language barriers that are frequently encountered in software development. MongoDB, a NoSQL document data store, serves as the foundation of a MEAN stack. Express is the HTTP server, and Angular is the front-end JavaScript framework. Node, a platform for server-side programming, is at the top of the stack. MEAN does not rely on a single operating system, which allows developers more flexibility; it also employs JavaScript, a widely used programming language. Ember can be used instead of Angular to build a MEEN stack, or Vue.js can be used as the front-end development framework in a MEVN stack.

- **Apache CloudStack:**

This opensource cloud management stack is used by large enterprise customers and service providers to provide infrastructure as a service (IaaS). CloudStack offers developers numerous levels of optional services, as well as compatibility for a wide range of hypervisors and application programming interfaces (APIs).

- **BHCS/BCHS (OpenBSD, C, HTTP, SQLite):**

This suite of open-source web application tools is built with the C programming language and is based on the OpenBSD operating system and HTTP web server.

- **GLASS (GemStone, Linux, Apache, Smalltalk, Seaside):**

Smalltalk is the programming language used in this stack for online application development, GemStone is the database and application server, and Seaside is the web framework. As enterprises increasingly embrace containers, distributed architectures, and integrations with cloud platforms and services, software stacks can grow complex. Software configuration management can be used to improve reporting and change management, as well as assign resources to development projects.

## 5. The problem-specific use of Data Structure

### Staff administration with Linked List

I signed a contract with a business. Currently, the organization need staff management software with full functionality such as entering id, name, phone, and room. Employees can be found by ID and deleted.

To begin implementing code, I must first construct an object Node to define variables head and tail. Some of the operations I developed include:

- AddFirst() method to put Staff adds on top of QLNv.
- AddLast() method to put Staff add last of QLNv.
- Input() method to input information of Staff on keyboard.
- Display() method to getSize() method to count the number of staffs in QLNv.
- SearchID() method to search staffs by ID.
- SearchName() method to search staffs by name.
- SearchClass() method to search staffs by class.
- DeleteFirst() method to delete staffs on top QLNv.
- DeleteLast() method to delete staff at last QLNv.

### ➤ Class Main:

```
package asm1;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner ( source: System.in);
        Staff_manager Staff = new Staff_manager();
        Staff.Input();
        Staff.Display();
        Staff st1 = new Staff( ID:"1", Name:"Nhat", phoneNumber:"0905233817", room:"101");
        System.out.println(x:"After AddFirst staff");
        Staff.addFrist( x:st1);
        Staff.Display();
        Staff st2 = new Staff( ID:"2", Name:"Hieu", phoneNumber:"0905123456", room:"102");
        System.out.println(x:"After AddLast staff");
        Staff.addFrist( x:st2);
        Staff.Display();
        Staff.removeFrist();
        System.out.println(x:"After remove first staff ");
        Staff.Display();
        Staff.removeLast();
        System.out.println(x:"After remove last staff ");
        Staff.Display();
        System.out.println(x:"Search by id: ");
        String id = sc.nextLine();
        Staff.SearchID(id);
    }
}
```

Figure 26: Source code 1

### ➤ Class Node:



```
package asm1;

class Node {
    Staff data;
    Node next;
}
```

Figure 27 : Source code 2

➤ **Class Staff\_manager:**

```
package asm1;

import java.util.Scanner;

public class Staff_manager {
    Node head = null;
    Node tail = null;
    void addFrist (Staff x) {
        Node p = new Node();
        p.data = x;
        p.next = null;

        if (head == null)
            head = tail = p;
        else {
            p.next = head;
            head = p;
        }
    }
    void addLast (Staff x) {
        Node p = new Node();
        p.data = x;
        p.next = null;
        if (head == null)
            head = tail = p;
        else {
            tail.next = p;
            tail = p;
        }
    }
    void Input() {
        String id, name, phone, cls;
        int dung = 0;

        Scanner sc = new Scanner( source: System.in);
        while (dung == 0) {
            System.out.print( s: "Enter ID: ");
```

Figure 28: Source code 3

```

        System.out.print(s:"Enter ID: ");
        id = sc.nextLine();
        System.out.print(s:"Enter Name: ");
        name = sc.nextLine();
        System.out.print(s:"Enter phoneNumbers: ");
        phone = sc.nextLine();
        System.out.print(s:"Enter Room: ");
        cls = sc.nextLine();
        Staff x = new Staff( ID:id, Name:name, phoneNumber:phone, room:cls);
        addLast(x);
        System.out.print(s:" (1) Stop ");
        System.out.print(s:"\n(0) Continue" );
        dung = sc.nextInt();
        sc.nextLine();
    }

}

void Display() {
    Node p = head;
    System.out.println(x:"\tID\t\tName\t\tPhone\t\t\tRoom");
    while (p != null) {
        System.out.printf(format:"%8s\t%12s\t", args:p.data.ID, args:p.data.Name);
        System.out.printf(format:"%8s\t%12s\t", args:p.data.phoneNumber, args:p.data.room);
        System.out.println();
        p = p.next;
    }
}

int getSize() {
    Node p = head;
    int count = 0;
    while (p != null) {
        count++;
        p =p.next;
    }
    return count;
}

```

Figure 29: Source code 4

```

}

int getSize() {
    Node p = head;
    int count = 0;
    while (p != null) {
        count++;
        p =p.next;
    }
    return count;
}

void SearchID(String id) {
    Node p = head;
    System.out.println(x:"\t\tID\t\t\tName\t\tRoom");
    while (p != null) {
        if (id.compareToIgnoreCase(str:p.data.ID) == 0) {
            System.out.printf(format:"%8s\t%12s\t", args:p.data.ID, args:p.data.Name);
            System.out.printf(format:"%8s\t%12s\t", args:p.data.phoneNumber, args:p.data.room);
            System.out.println();
        }
        p = p.next;
    }
}

void removeFrist() {
    head = head.next;
    System.out.println(x:"Remove first student Successfully");
}

void removeLast() {
    Node p = head;
    while (p.next != null) {
        p = p.next;
    }p.next = null;
}

```

Figure 30: Source code 5

➤ **Class Staff:**

```
package asm1;

class Staff {
    String ID, Name, phoneNumber, room;
    Staff(String ID, String Name, String phoneNumber, String room){
        this.ID = ID;
        this.Name = Name;
        this.phoneNumber = phoneNumber;
        this.room = room;
    }
}
```

Figure 31: Source code 6

➤ **OUTPUT:**

I use the keyboard to enter information such as Id, Name, Phone Numbers, and Staff Room:

```
Enter ID: 3
Enter Name: Viet
Enter phoneNumbers: 1234567890
Enter Room: 103
(1) Stop
(0) Continue0
Enter ID: 4
Enter Name: An
Enter phoneNumbers: 0123456789
Enter Room: 104|
```

Figure 32: Output 1

Next, I try to eliminate the employees at the top of the list and the employees at the bottom of the list. Then, display the personnel details once more.

```
Remove first student Successfully
After remove first staff
```

ID	Name	Phone	Room
1	Nhat	0905233817	101
3	Viet	1234567890	103
4	An	0123456789	104

```
After remove last staff
```

ID	Name	Phone	Room
1	Nhat	0905233817	101
3	Viet	1234567890	103

Figure 33: Output 2

The information will appear when I press "1" on the keyboard. Furthermore, attempt to Add First and Last employees, then display the information List of employees again:

After AddFirst staff

ID	Name	Phone	Room
1	Nhat	0905233817	101
3	Viet	1234567890	103
4	An	0123456789	104

After AddLast staff

ID	Name	Phone	Room
2	Hieu	0905123456	102
1	Nhat	0905233817	101
3	Viet	1234567890	103
4	An	0123456789	104

Figure 34: Output 3

I search for employees by id "1":

Search by id:

1

	Id	Name	Room
1	Nhat	0905233817	101

---

Figure 35: Output 4

## REFERENCES:

Available at: <https://www.simplilearn.com/tutorials/java-tutorial/linked-list-in-java>

Available at: <https://www.programiz.com/java-programming/linkedList>

Available at: [https://www.tutorialspoint.com/basics\\_of\\_computers/basics\\_of\\_computers\\_primary\\_memory.htm](https://www.tutorialspoint.com/basics_of_computers/basics_of_computers_primary_memory.htm)

Available at: [https://www.tutorialspoint.com/basics\\_of\\_computers/index.htm](https://www.tutorialspoint.com/basics_of_computers/index.htm)

Available at: [https://www.tutorialspoint.com/basics\\_of\\_computers/basics\\_of\\_computers\\_quick\\_guide.htm](https://www.tutorialspoint.com/basics_of_computers/basics_of_computers_quick_guide.htm)