TỔ CHỨC GIÁO DỤC FPT

# Higher Nationals - Summative Assignment Feedback Form

| | | | |
|---|---|---|---|
| **Student Name/ID** | Vo Dinh Nhat / GBD210218 | | |
| **Unit Title** | Data Structures and Algorithms | | |
| **Assignment Number** | Assignment 1 | **Assessor** | Nguyen The Nghia |
| **Submission Date** | | **Date Received 1st submission** | |
| **Re-submission Date** | | **Date Received 2nd submission** | |

**Grading grid**

| P1 | P2 | P3 | M1 | M2 | M3 | D1 | D2 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

**Assessor Feedback:**

| | | |
|---|---|---|
| **Grade:** | **Assessor Signature:** | **Date:** |

**Resubmission Feedback:**

| | | |
|---|---|---|
| **Grade:** | **Assessor Signature:** | **Date:** |

| |
|---|
| **Internal Verifier’s Comments:** |
| **Signature & Date:** |

* Please note that grade decisions are provisional. They are only confirmed once internal and external moderation has taken place and grades decisions have been agreed at the assessment.

## Contents

# 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

In programming languages like Java, C, C++, and Python, programmers must work with a variety of data types, including int, float, double, char, string, and others. Usually, these data types correspond to particular byte sizes, such 4 bytes for int and 1 byte for character types. For the ease of the user, certain data types are frequently abstracted or simplified during programming. Programmers routinely work with a variety of data types, including int, float, double, char, string, and more, in languages like Python, C, C++, and Java. These data types often have fixed byte sizes; for instance, an int takes up 4 bytes, while character types take up around 1 byte. To make them easier to grasp for the user, certain data types are frequently abstracted or simplified during the programming process.

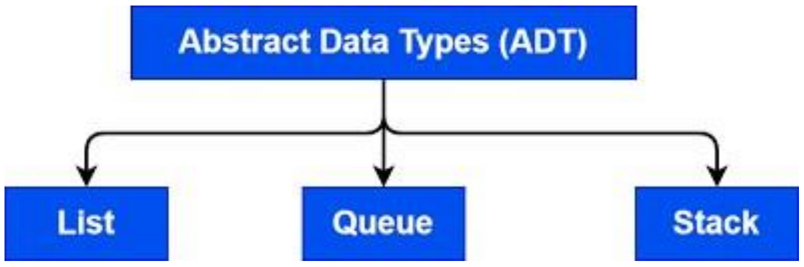


**Figure 1: Abstract Data Type**

### 2. Common ways to represent ADT.

- Using ArrayList

A dynamic array with resizing capabilities is an ArrayList. When one element is removed, it automatically compresses and extends to make room for other elements. An ArrayList internally keeps its elements in an array. An index can be used to access elements in an ArrayList, just like it can with conventional arrays. Null and duplicate values are supported by the Java ArrayList class. Since it is an ordered collection, each item is kept in the same order that it was inserted.

It's crucial to remember that primitive types like int, char, and so forth cannot be directly stored in an array list. Rather, you must use the equivalent boxed types (Character, Boolean, Integer, etc.).

- It's important to note that Java ArrayList lacks thread safety because it is not synchronized. When an ArrayList is being modified concurrently by several threads, the result is unexpected. To ensure thread safety, you should explicitly synchronize access to an ArrayList if multiple threads are altering it.

The procedures you stated have the following explanations:

- add(): Enables the list to have a new entry.
- addAll(): This function joins every element in one list to another.
- get(): Using its index, this function retrieves an entry from the list. Elements can be accessed at random.
- iterator(): Provides an iterator object that may be used to read a list's elements one after the other.
- set(): Adjusts the value of an element in a list at a certain index.
- remove(): Takes out a particular entry from the list.
- removeAll(): Gets rid of every element in the list.
- clear(): Gets rid of every item in the list. RemoveAll() is not as efficient as this method.
- size(): Gives back the total number of items in the list.
- toArray(): Generates an array from a list.
- contains(): This function determines whether a given element is present in the list and, if so, returns true.

- Using Singly Linked List

One kind of linked list is a singly linked list, which is made up of a series of nodes, each of which has data and a link to the node after it. It allows movement in a single direction and can be navigated from the list's initial node, also called the "head," to its last node, also called the "tail." The singly linked list is depicted in the above graphic. A "node" is any one of the list's elements. Two components make up a node: data and a pointer.

The link to the following node in the list is represented by the pointer, and data is the information kept in the node.

The term "head" refers to the first node in the list. The "tail" node, which is the last node in the list, links to NULL.

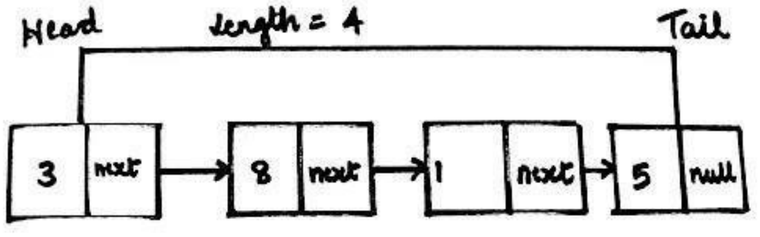


**Figure 2: Singly Linked List**

- Using Doubly Linked List

A doubly linked list is a kind of linked list that is made up of a series of nodes, each of which has two pointers—the previous pointer and the subsequent pointer—along with data. The next node that is included in the list is indicated by the following pointer, whereas the prior pointer refers to the node that came before it. One can explore this kind of linked list from the first node to the last node and the other way around. The doubly linked list is depicted in the above graphic. Each node has two pointers: the previous pointer and the next pointer, which indicate the data, which is the information contained in the node. The node that came before it in the list is indicated by the prior pointer.

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

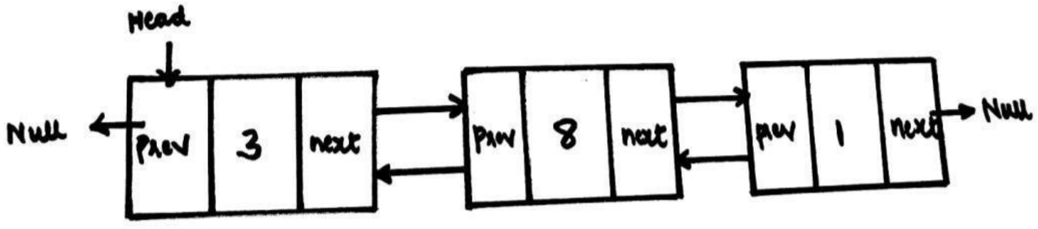


**Figure 3: Doubly Linked List**

- Using Circular Linked List

A circular chain is formed when the final node in a linked list points to the initial node, creating a circular linked list. This makes it possible to traverse the list in any direction—forward or backward—starting at any node and ending at the same node as before. A circular linked list lacks a clear beginning and finish as a result.

- Here's an example of a circular linked list in the image:

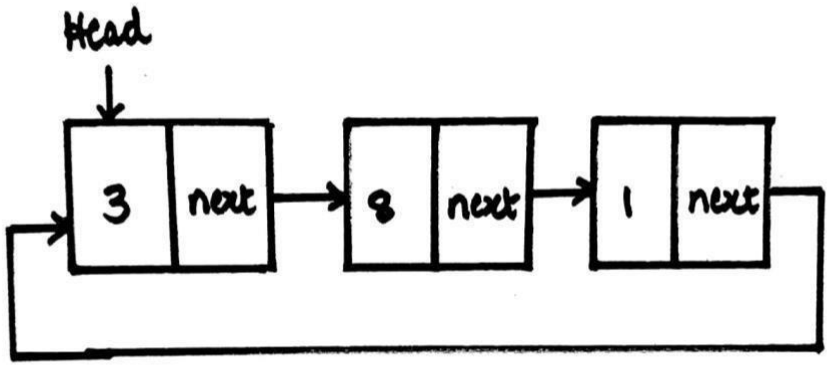


**Figure 4: Circular Linked List**

Except for the end node, which points back to the initial node, every node in the list has data and a pointer to the next node in the list. Many data structures, including as queues, scheduling algorithms, and circular buffers, can make use of circular linked lists.

- Using Stack

Last in, first out, or LIFO, is an abstract data structure that is based on stacks. The elements at the top of the stack are added and removed. It can be pushed, popped, and peeked at to see the top element without taking it off. Programming and applications frequently employ stacks to handle function calls, temporary variables, and undo and redo procedures.

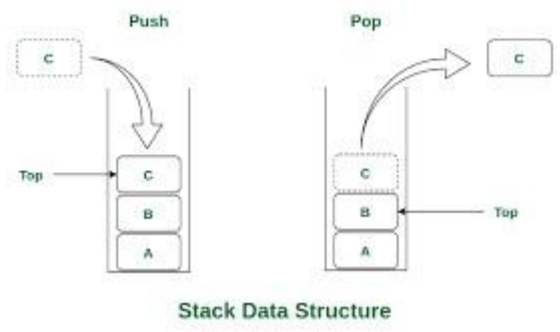


**Figure 5: Stack.**

Here are the methods of a stack data structure:

- empty(): This function determines whether the stack is empty and returns false if it does not contain any elements and true otherwise.
- peek(): This function retrieves, without eliminating, the element at the top of the stack.
- pop(): This function takes out the element at the top of the stack and puts it back.
- push(): This function returns the element that has been added to the top of the stack.
- search(): This function looks through the stack for a certain element. It returns the element's distance from the top of the stack if it is located. If not, it gives back -1.

- Using QUEUE

An abstract data structure called a queue resembles a line or queue in the real world. A queue is open at both ends, in contrast to a stack. Data is inserted (also called enqueued) using one end, and removed (dequeued) using the other end. The data that is placed first is the first to be accessed in a queue data structure that uses the First-In-First-Out (FIFO) method. There are several instances of queues in the real world, such as rows of cars on a one-way street where the first vehicle to enter is the first to exit (particularly during traffic jams). Additional instances comprise of student lines, passes, and so on.

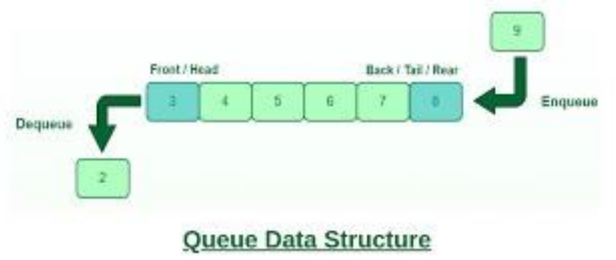


**Figure 6: Queue.**

Here are the methods of a queue data structure:

- enqueue(): This function appends a new element to the queue's end.
- dequeue(): This function takes out the element from the front of the queue and puts it back.
- peek(): This function keeps the element at the head of the queue in place while returning it.
- isFull(): This function determines whether the queue is full and returns false otherwise. If it is, it returns true.
- isEmpty(): This function determines whether the queue is empty and returns false otherwise. If it is, it returns true.

## 3. Examples

- Queue

The "first in, first out" (FIFO) data structure is used to hold things in a queue. Items can be added to the queue at any point, but the only item that can be deleted from the queue is the first one added. In a queue, adding an object always happens at the back, and removing an object always happens at the front. The terms "enqueue" and "dequeue," respectively, relate to these operations. Applications for the queue data structure in computing include work scheduling, keyboard buffer management, breadth-first and backtracking algorithms, recursion elimination, and exhaustive search.

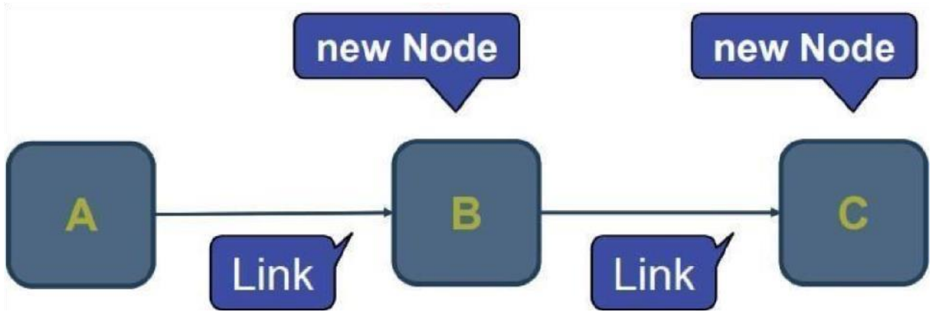


**Figure 7: Queue.**

Queue's advantages include:

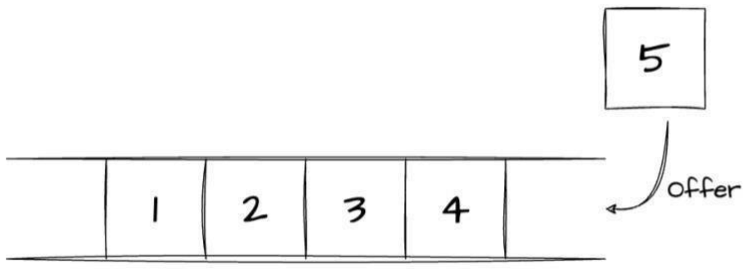
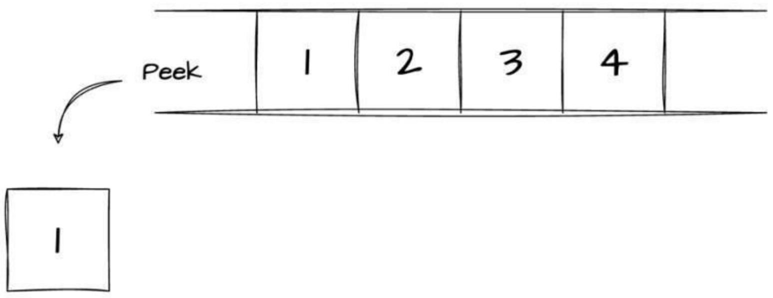
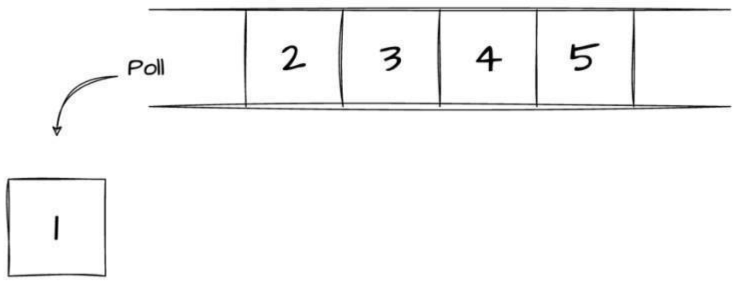
- Easy management of massive amounts of data.
- Simple insertion and deletion operations due to its adherence to the first-in, first-out principle.
- When several customers use a specific service, queues are helpful.
- When it comes to data interprocess communication, queues move quickly.
- Other data structures can be implemented using queues.

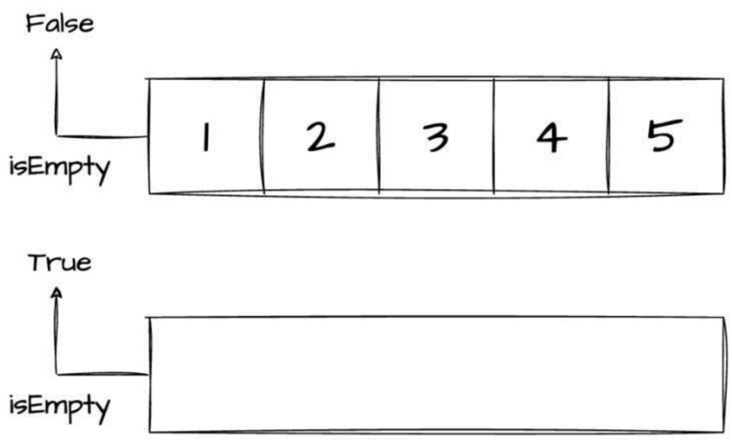
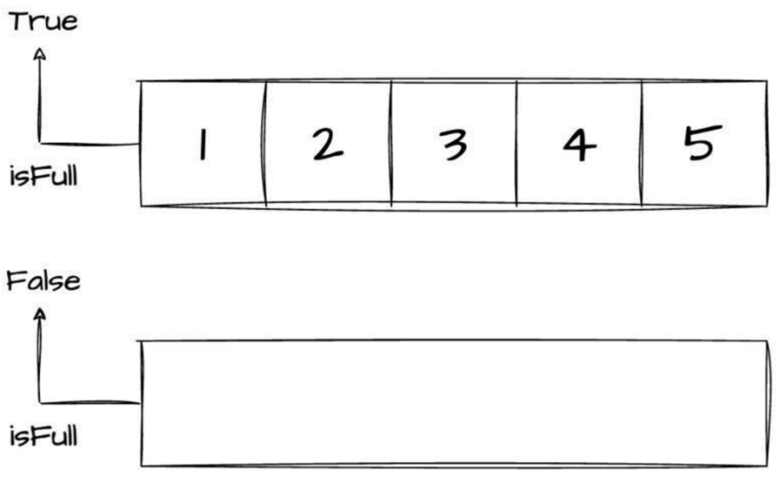
One of queue's drawbacks :

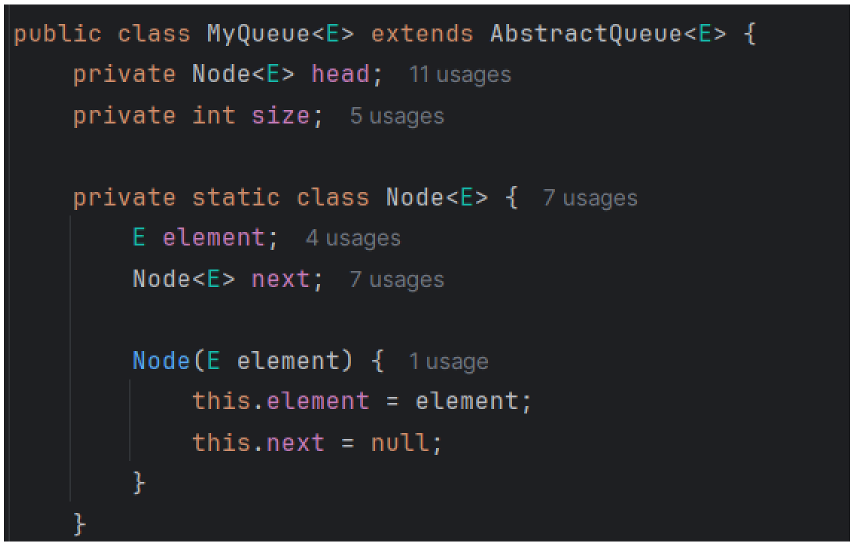
- time-consuming operations like adding and removing members from the middle must be performed.
- Restricted Room.
- In a traditional queue, the addition of a new element is contingent upon the removal of all current components from the queue.
- Finding an element requires O(N) processing time.

- A queue's maximum size needs to be specified beforehand.

- **Class Queue:**

| Method | Description | Illustractions |
|---|---|---|
| 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<br><br>front of the queue |  |

| | | |
|---|---|---|
| isEmpty | Determine whether the queue is empty |  False<br>isEmpty<br>1<br>2<br>3<br>4<br>5<br>True<br>isEmpty |
| isFull | Determine whether the<br><br>queue is full |  |



```
public class MyQueue<E> extends AbstractQueue<E> {
    private Node<E> head;  11 usages
    private int size;  5 usages

    private static class Node<E> {  7 usages
        E element;  4 usages
        Node<E> next;  7 usages

        Node(E element) {  1 usage
            this.element = element;
            this.next = null;
        }
    }
```

**Figure 8: MyQueue.**

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

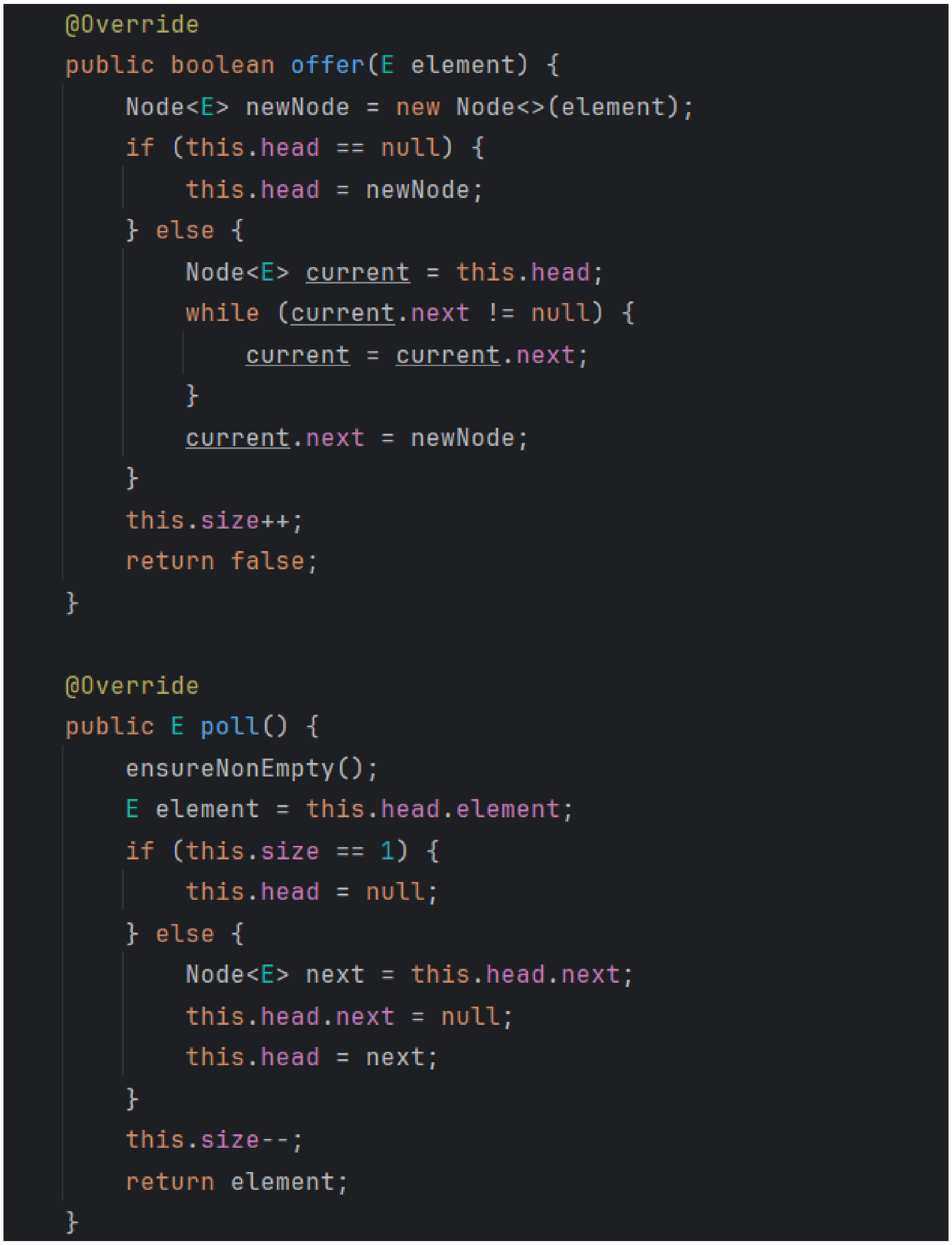
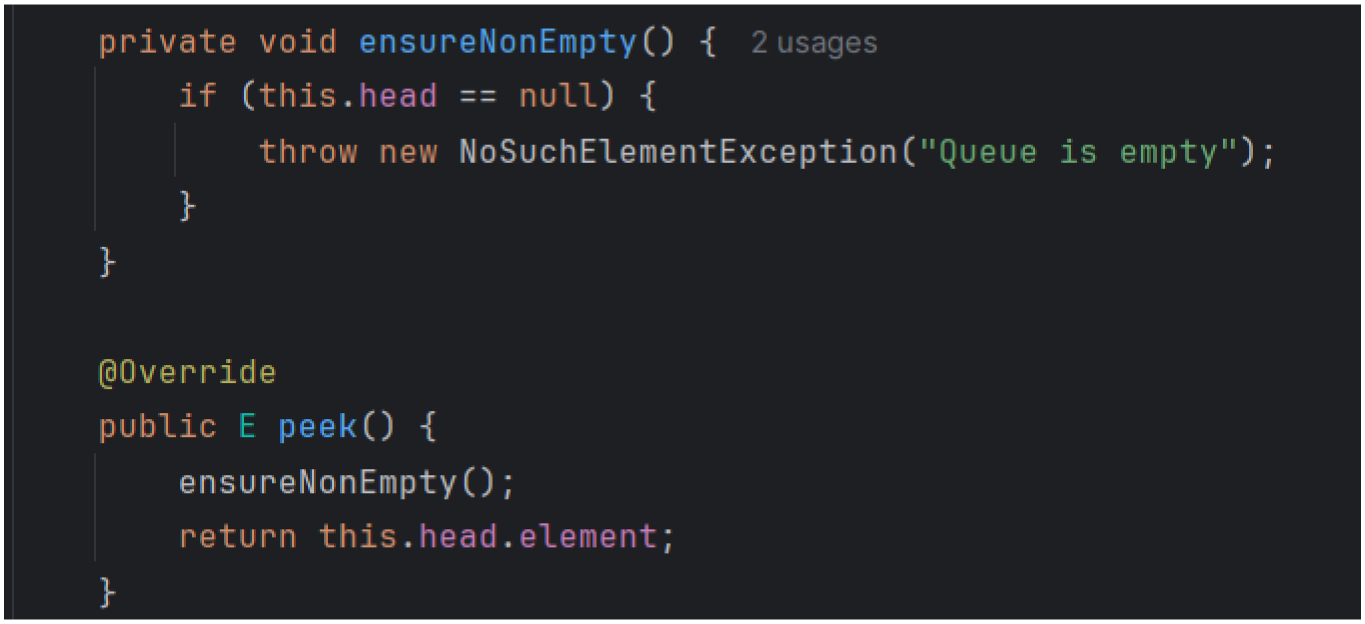
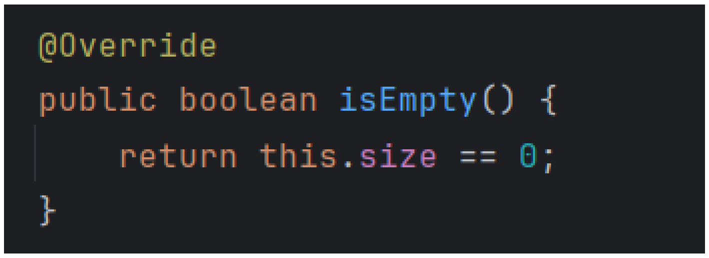

```
@Override
public boolean offer(E element) {
    Node<E> newNode = new Node<>(element);
    if (this.head == null) {
        this.head = newNode;
    } else {
        Node<E> current = this.head;
        while (current.next != null) {
            current = current.next;
        }
        current.next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

@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 9: Offer() and Poll()**



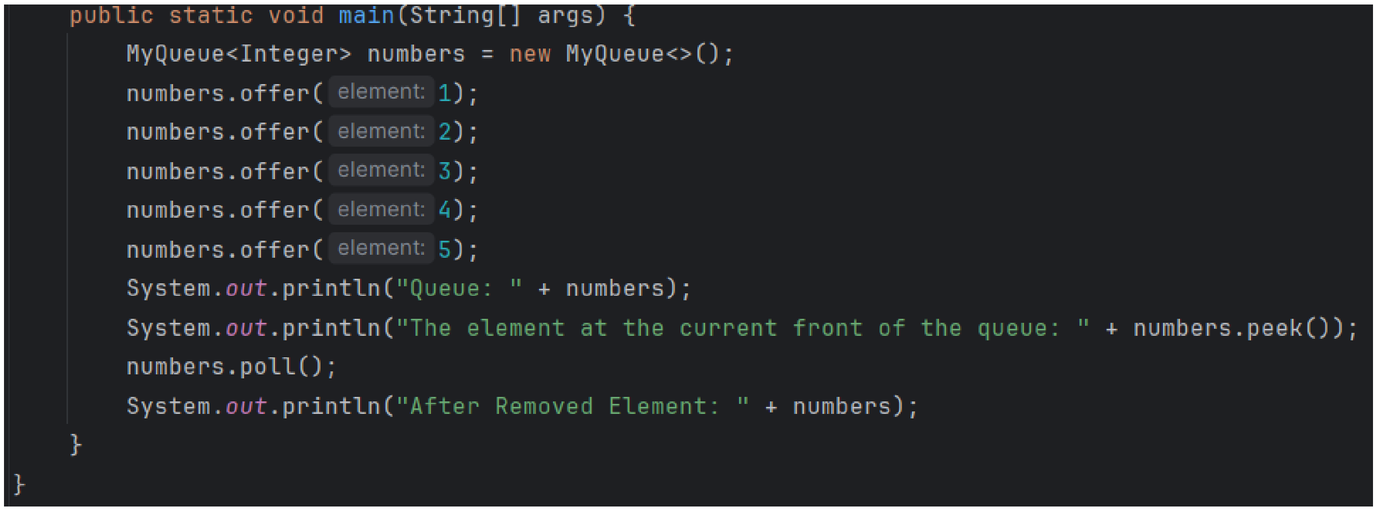
```
private void ensureNonEmpty() {  2 usages
    if (this.head == null) {
        throw new NoSuchElementException("Queue is empty");
    }
}

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

**Figure 10: ensureNonEmpty() and Peek()**



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

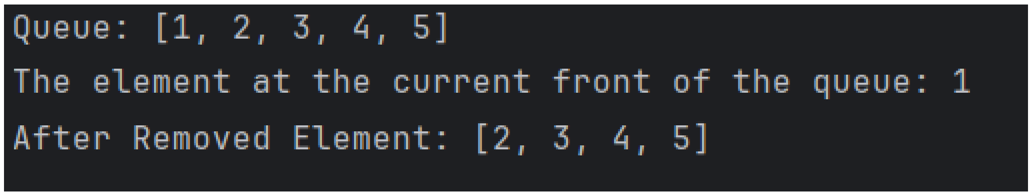
**Figure 11: isEmpty()**



```
    public static void main(String[] args) {
        MyQueue<Integer> numbers = new MyQueue<>();
        numbers.offer( element: 1);
        numbers.offer( element: 2);
        numbers.offer( element: 3);
        numbers.offer( element: 4);
        numbers.offer( element: 5);
        System.out.println("Queue: " + numbers);
        System.out.println("The element at the current front of the queue: " + numbers.peek());
        numbers.poll();
        System.out.println("After Removed Element: " + numbers);
    }
}
```

**Figure 12: 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 13: 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

The physical structure of memory is made up of several cells, each of which has a single bit of storage. They are rationally arranged as collections of units known as words, each having a distinct address. Instructions and data can be retrieved using these memory locations.

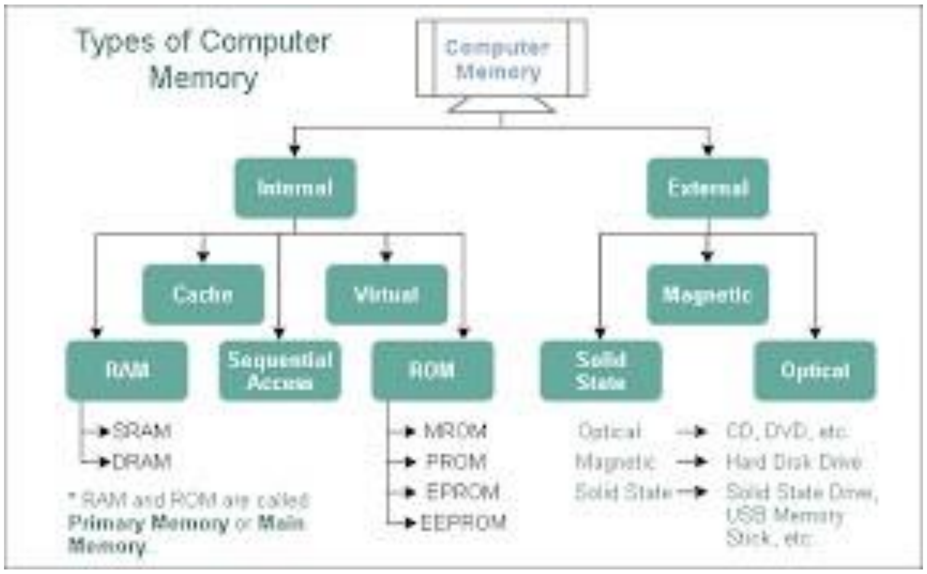


**Figure 14: Computer System Memory**

To be clear, an operating system loads a program into main memory before executing it. Computer instructions and data used by software are both stored in memory. The software partitioned its memory to execute distinct program operations. Applications still arrange their memory according to the functional units listed below, even though paged memory has mostly supplanted this technique.

Computers with paged memory dynamically manage memory, enabling a program's memory allocation to change in response to the demands of the application. Programs are allotted memory in fixed-size units called pages, which the operating system then recovers. On a computer with paged memory, when you load software, the operating system allots a certain number of pages to the program initially, and subsequently more RAM as needed. Pages containing code and data that are not needed right now might be returned to the operating system if they haven't been utilized in a long.

- An application of Stack in memory management

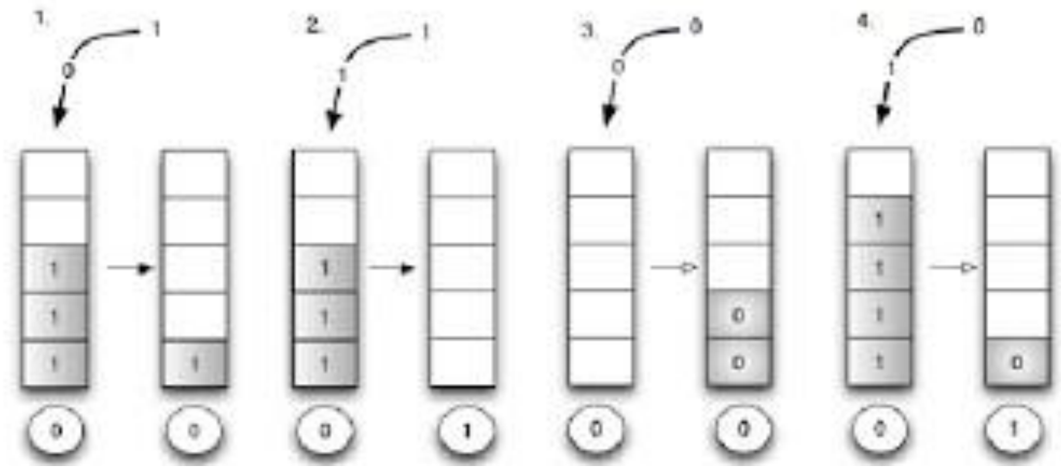


**Figure 15: Demonstrations of stack behavior.**

## 2. Operations

Elements of the same type are arranged gradually in stacks. The one end of the stack where all actions are completed is the top. utilizing the Last In, First Out (LIFO) tenet. Additionally, the following stack operations can be performed:

- PUSH:

In the PUSH process, a new element is inserted into a stack. Since the top of the stack is where a new element is always inserted, we must constantly verify that the top is empty, or that TOP=Max-1. In the event that this condition is false, the stack is full and cannot hold any more elements; an error message indicating a stack overflow occurs when we attempt to add an element.

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
```

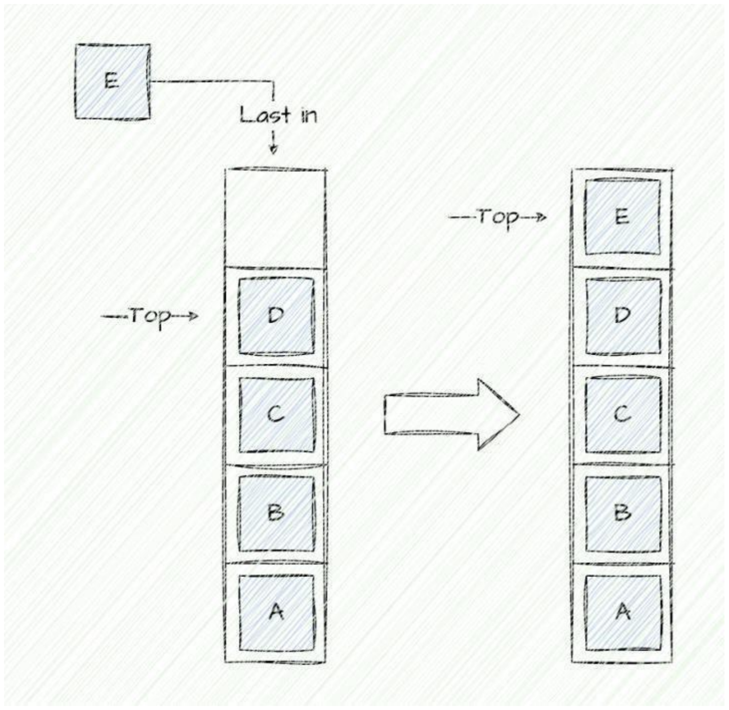


**Figure 16: Push operation of Stack.**

- POP:

The removal of an element from the stack is indicated by POP. Make that the Stack Top (i.e., TOP=NULL) is not NULL before deleting an element. The stack is empty, and no deletion actions may be carried out if this condition is satisfied. The Stack underflow notification appears if we attempt to delete.

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

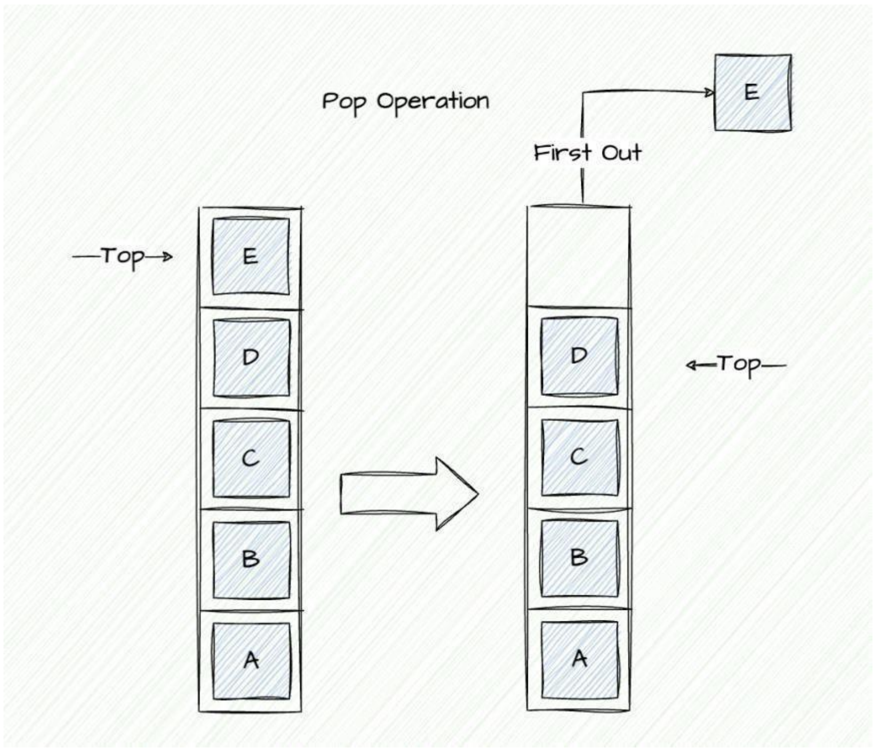


**Figure 17: Pop operation of Stack.**

- PEEK:

When we need to return the value of the top element in the stack without removing it, we use the Peek action. Initially, this action checks to see if the Stack is empty, i.e., TOP = NULL; if it is, a suitable message is shown; if not, 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

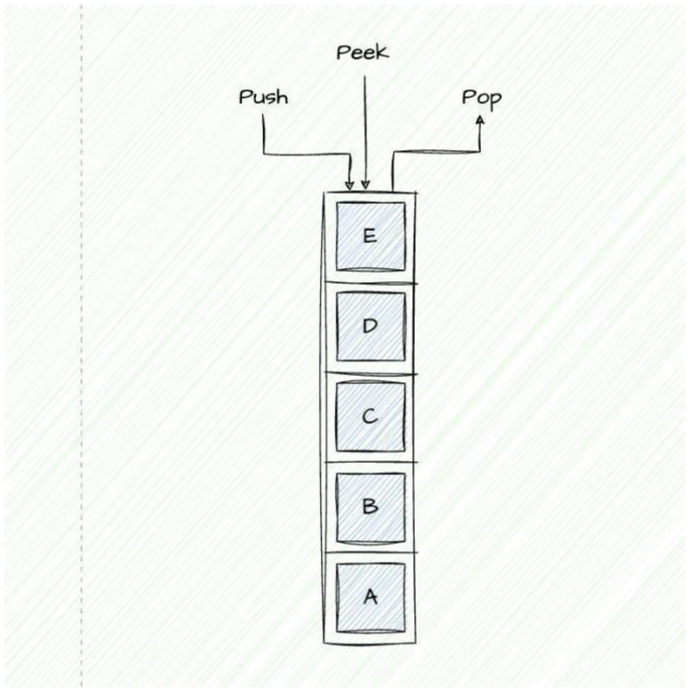


**Figure 18: Peek operation of Stack.**

- isEmpty

When we're notThe programmer must internally maintain the stack size, which will be updated for push and pop actions, to prevent operations from being carried out on an empty stack. Typically, isEmpty() yields a boolean result. True if size is 0; false otherwise.

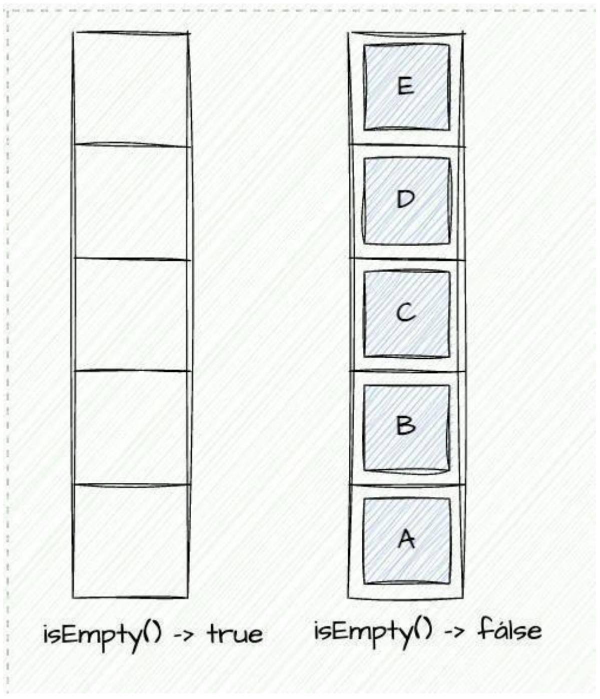
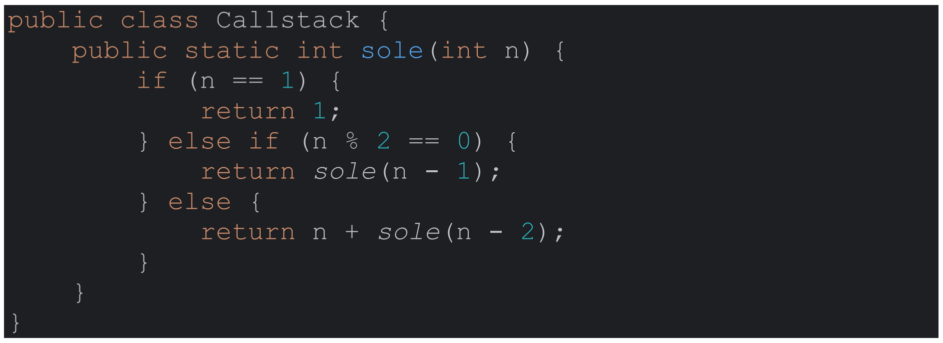


**Figure 19: is Empty operation of Stack.**

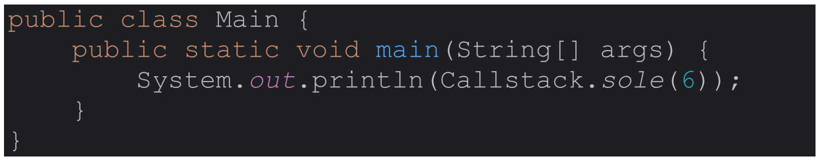
## 2. How a method (function) calls are implemented with stack

- A data structure called the method-call stack, sometimes referred to as the program-execution stack, is used in programming to implement the call/return mechanism. For every called procedure, it permits the establishment, upkeep, and deletion of local variables. A method must return to the calling method in order for the stack to exhibit Last In First Out (LIFO) behavior. The system has to remember each method's return address when it calls another method. The best data structure for this is the method call stack. Every time a method calls another, an activation record with the return address and extra data is added to the stack.

- The stack frame for the method call is popped and control is transferred to the return address in the popped stack frame when a called method returns without calling another method. Because any called method can always find the data it needs to return to its caller at the top of the stack, the call stack is also helpful. The return address of the newly called method is now at the top of the call stack when one method calls another, inserting a stack frame for the new method call into the call stack.
- Because a function needs access to its stack frame in order to function, this must be granted during execution. The sum of an integer's odd integers is computed in a little program that demonstrates how a method uses a stack.

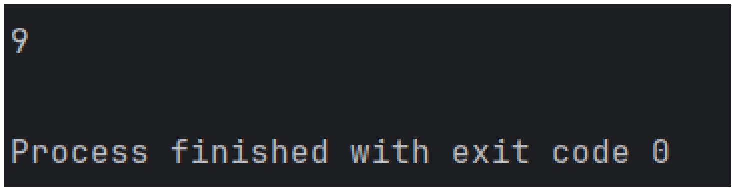


```
public class Callstack {
    public static int sole(int n) {
        if (n == 1) {
            return 1;
        } else if (n % 2 == 0) {
            return sole(n - 1);
        } else {
            return n + sole(n - 2);
        }
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        System.out.println(Callstack.sole(6));
    }
}
```

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



```
9

Process finished with exit code 0
```

**Figure 20: Result.**

The recursive function is used when n is more than 1, and if n is divisible by two, its value is decreased by two; if not, it is reduced by one. This procedure keeps going until n equals 1, which is the base case. Until the original function call is reached and the result is computed, the return value from each function call is now passed on to the function call that comes before it, and so on.

To make sure that there aren't any more recursive calls after a particular amount of time, the base case is crucial. The function would keep calling itself without a base case, which would eventually cause a stack overflow and program failure. As a result, for the recursive function to function properly, a base case is needed.

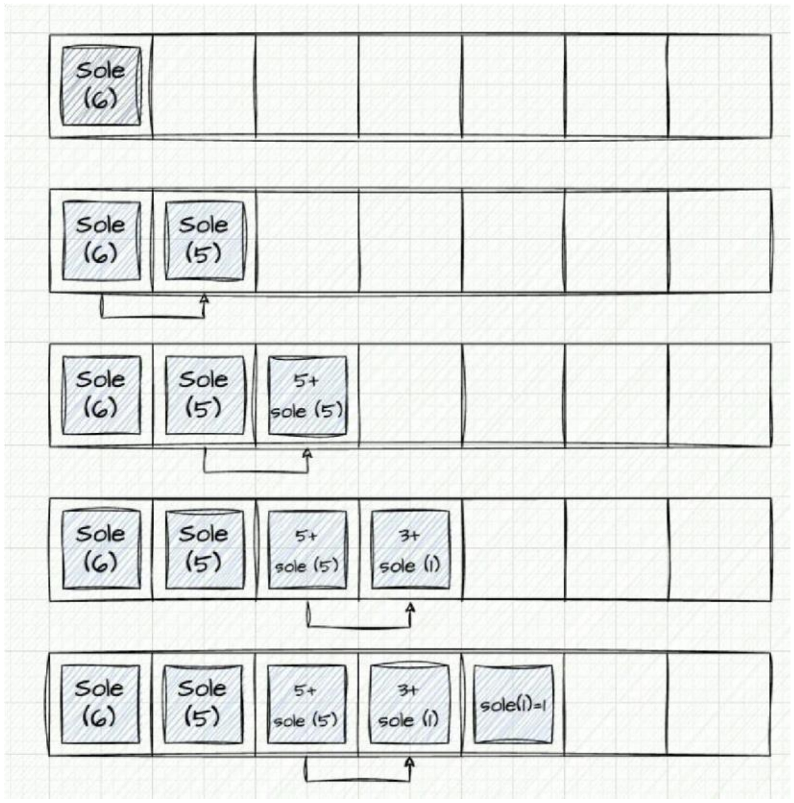


**Figure 21: The sum of the odd numbers of 6 (1)**

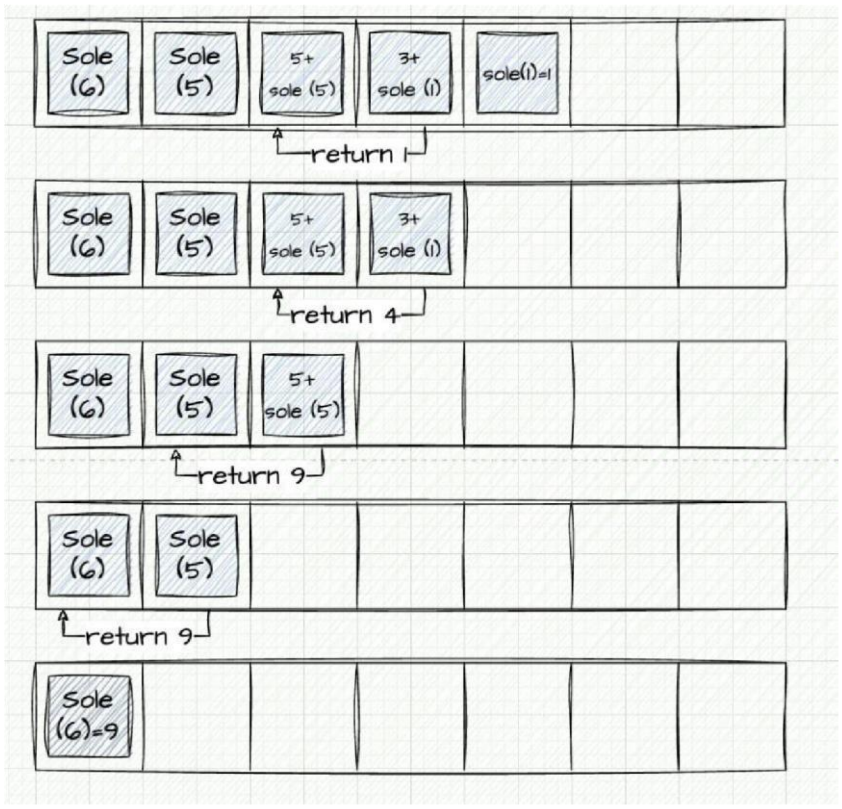


**Figure 22: The sum of the odd numbers of 6 (2)**

TỔ CHỨC GIÁO DỤC FPT

# 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. Scenario

It is true that overseeing employees typically requires a lot of work and time. If you handle a personnel list only on paper and do not use software, this process is quite laborious and time-consuming. Thus, I'll develop a program to manage staff members of a small business using Linked List to address such issues in this ADT application portion.

**Figure 23: Problem - Employee Management**

### 2. Basic Operations on linked list

- Traversal: The act of moving between nodes.
- Insertion: The addition of a node at the designated location.
- DELETION: A node that is eliminated.
- Searching: Using the element's value to locate the element or elements.
- Updating: The process of changing a node.
- Sorting: is the process of placing nodes in a linked list in a specific order.
- Merging: Combining two linked lists into a single one.

## 3. Part of a software stack

Software stacks can be simple or sophisticated, incorporating parts and services from an organization's on-premises resources, outside providers, or cloud providers, depending on the intended application functionality. Apart from the fact that its features and functions make it possible for an application to be developed, delivered, and used, there is no minimal standard for the services and components that must be part of a software stack.

## 4. Five software stack examples

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

This software stack for web development is well-known. At the base of the stack structure is the Linux operating system, which communicates with the Apache web server. The top of the hierarchy is the scripting language, in this case PHP. (Alternatively, the "P" could represent for the programming languages Perl or Python.) LAMP stacks are widely used because the components are all freely available and the stack can operate on commodity hardware. Unlike monolithic software stacks, which are often tightly coupled and designed for a particular operating system, a LAMP stack is loosely connected. This just means that even if the parts were not intended to work together at first, they have shown to be complementary and are often used in tandem. Nowadays, almost every version of Linux contains LAMP components. PostgreSQL can be used by developers in place of MySQL to create a LAPP stack. The LEAP stack (Linux, Eucalyptus, AppScale, Python) is an additional iteration of this open-source software stack that is utilized for cloud-based development and service delivery.

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

Language barriers that are commonly encountered in software development can be removed with the help of this set of development tools. A MEAN stack's base is a NoSQL document data store called MongoDB. The front-end JavaScript framework is called Angular, while the HTTP server is called Express. At the top of the stack is Node, a server-side programming platform. In addition to using JavaScript, a popular programming language, MEAN provides developers greater freedom by not depending on a particular operating system. Vue.js can be used as the front-end development framework in a MEVN stack, or Ember can be used in place of Angular to form a MEEN stack.

- **Apache CloudStack:**

Infrastructure as a service (IaaS) providers and major enterprise clients employ this opensource cloud management stack. Developers can choose from a variety of different service levels and hypervisor and application programming interface (API) compatibilities with CloudStack.

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

Built in C, this package of open-source web application tools is based on the HTTP web server and the OpenBSD operating system.

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

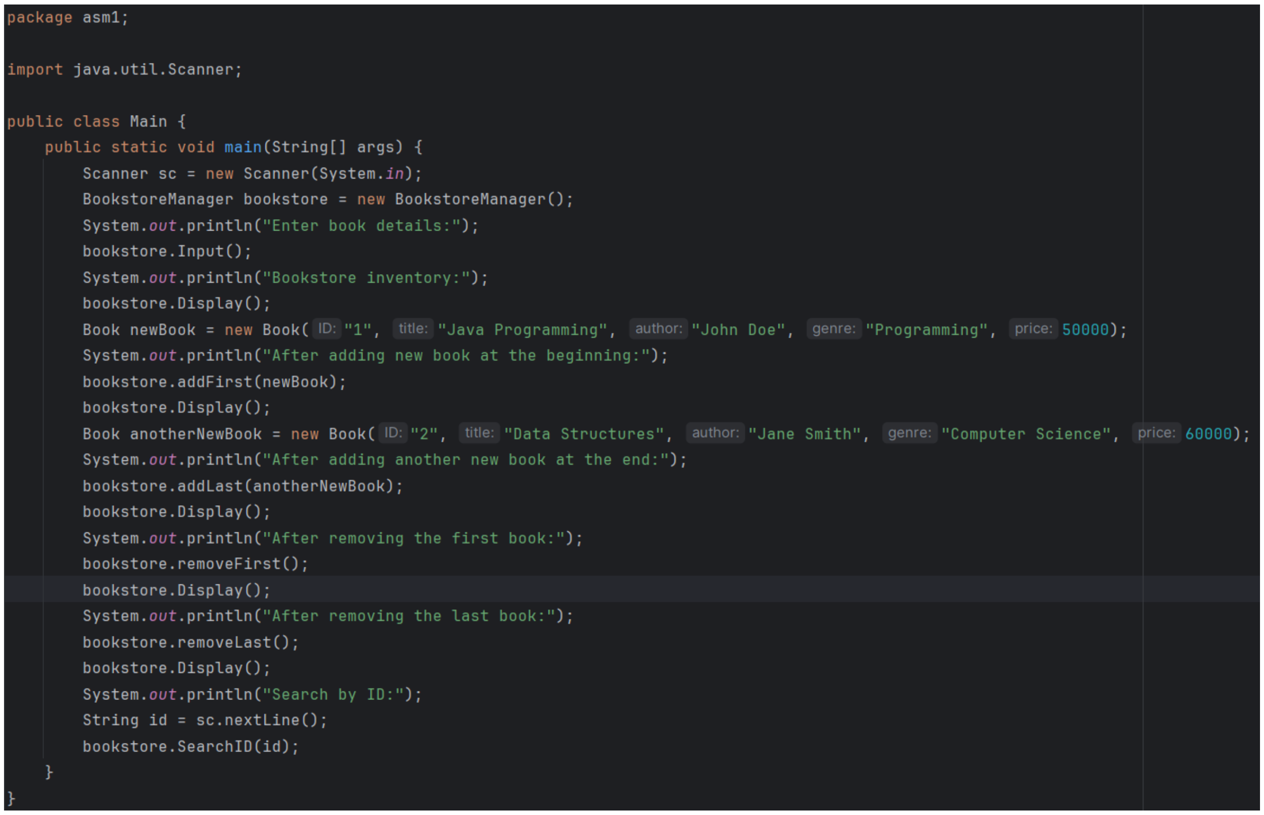
Seaside is the web framework, GemStone is the database and application server, and Smalltalk is the programming language used in this stack to create online applications. Software stacks can get more complicated as businesses adopt distributed architectures, containers, and connections with cloud platforms and services. Software configuration management facilitates resource assignment to development projects and enhances reporting and change management.

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

I agreed to a contract with a company. The company currently needs staff management software that can insert phone numbers, rooms, IDs, and names. Workers can be removed and located using their IDs.
I have to build an object Node first in order to declare the variables head and tail before I can start implementing code. Among the procedures I created are:

- 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.

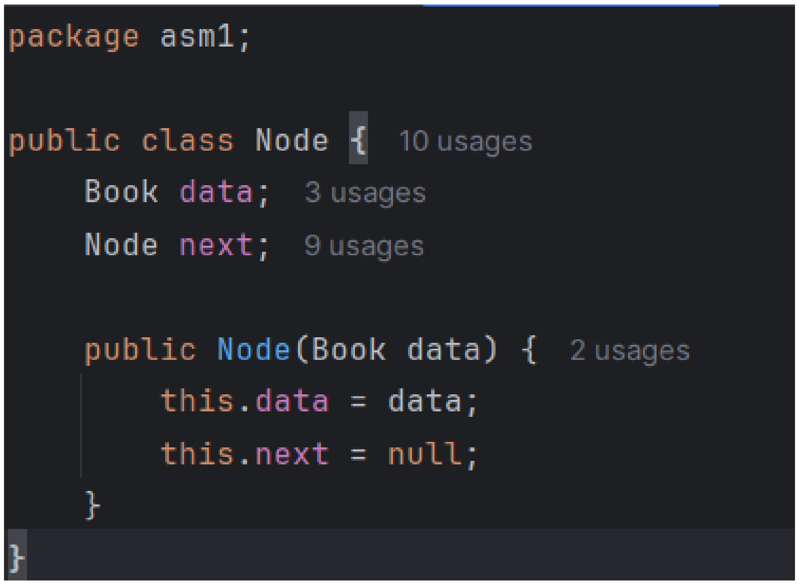
➢ Class Main:



```
package asm1;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        BookstoreManager bookstore = new BookstoreManager();
        System.out.println("Enter book details:");
        bookstore.Input();
        System.out.println("Bookstore inventory:");
        bookstore.Display();
        Book newBook = new Book( ID: "1",  title: "Java Programming",  author: "John Doe",  genre: "Programming",  price: 50000);
        System.out.println("After adding new book at the beginning:");
        bookstore.addFirst(newBook);
        bookstore.Display();
        Book anotherNewBook = new Book( ID: "2",  title: "Data Structures",  author: "Jane Smith",  genre: "Computer Science",  price: 60000);
        System.out.println("After adding another new book at the end:");
        bookstore.addLast(anotherNewBook);
        bookstore.Display();
        System.out.println("After removing the first book:");
        bookstore.removeFirst();
        bookstore.Display();
        System.out.println("After removing the last book:");
        bookstore.removeLast();
        bookstore.Display();
        System.out.println("Search by ID:");
        String id = sc.nextLine();
        bookstore.SearchID(id);
    }
}
```

**Figure 24: Source Code 1.**

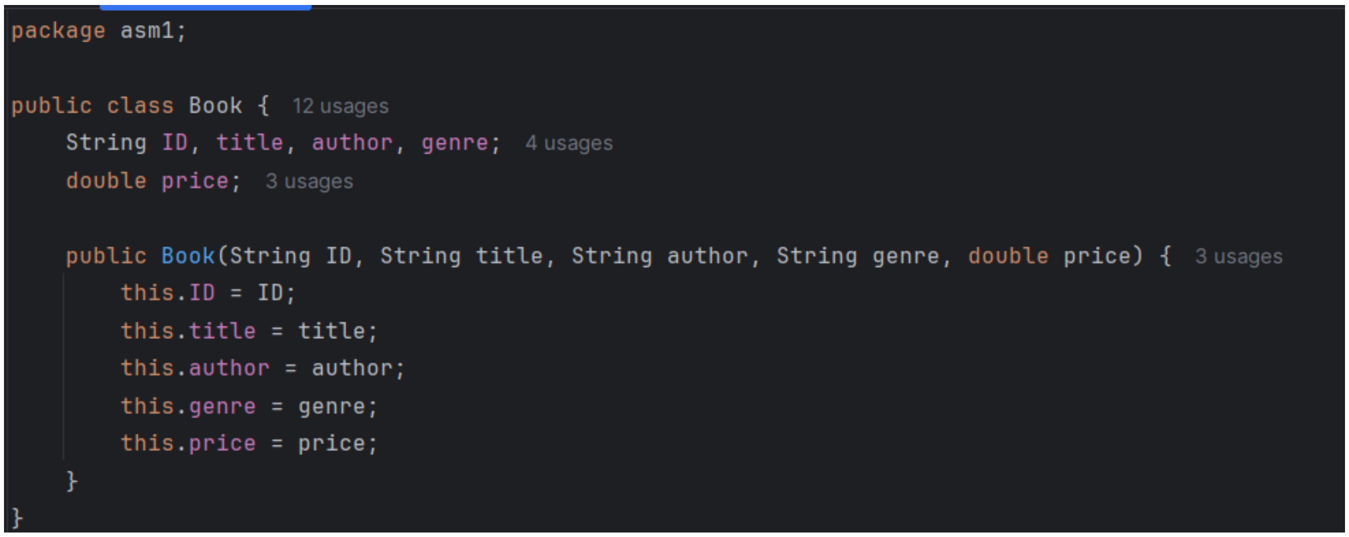
- Class Node:



```java
package asm1;

public class Node {  10 usages
    Book data;  3 usages
    Node next;  9 usages

    public Node(Book data) {  2 usages
        this.data = data;
        this.next = null;
    }
}
```

**Figure 25: Source Code 2.**

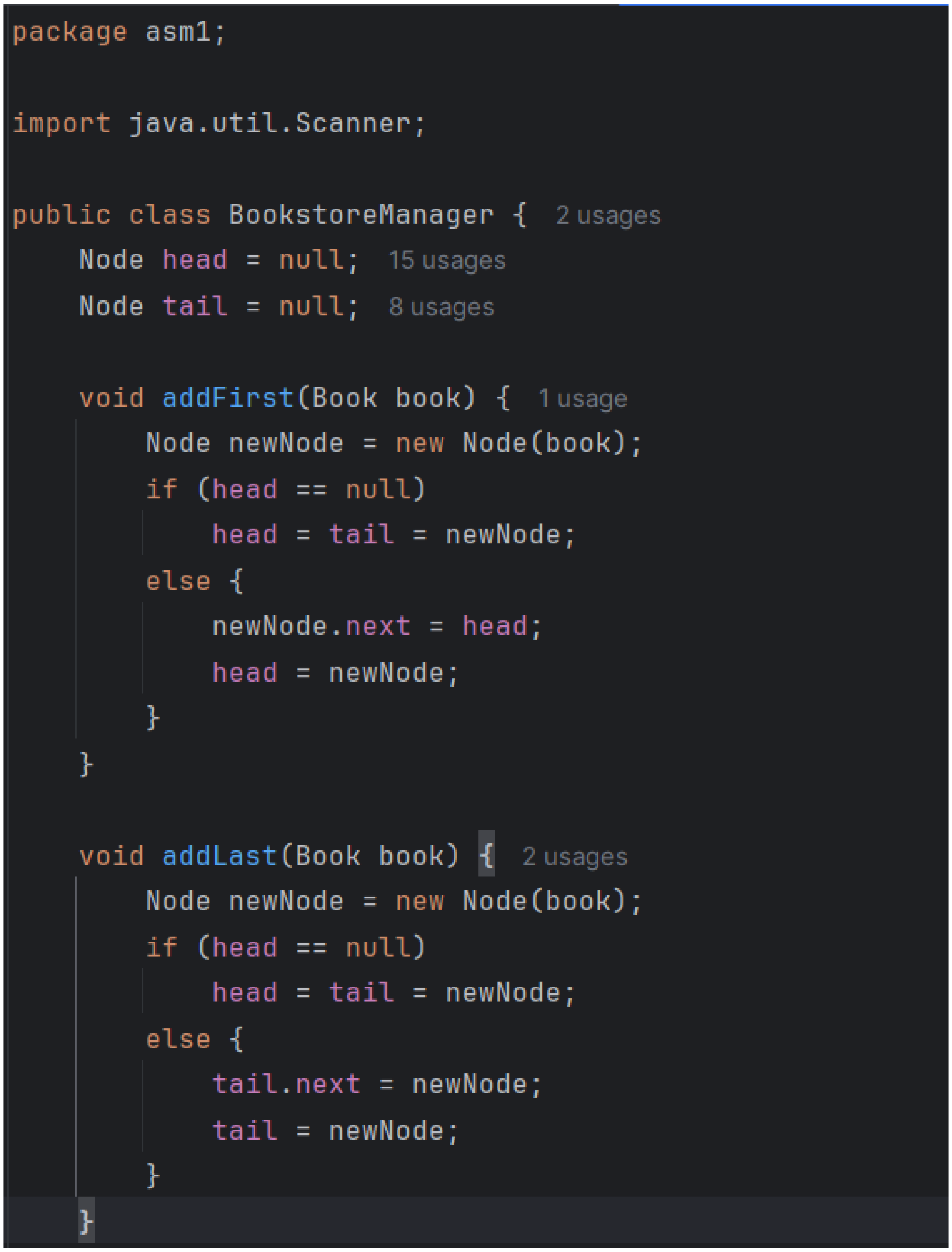
- Class Book:



```java
package asm1;

public class Book {  12 usages
    String ID, title, author, genre;  4 usages
    double price;  3 usages

    public Book(String ID, String title, String author, String genre, double price) {  3 usages
        this.ID = ID;
        this.title = title;
        this.author = author;
        this.genre = genre;
        this.price = price;
    }
}
```

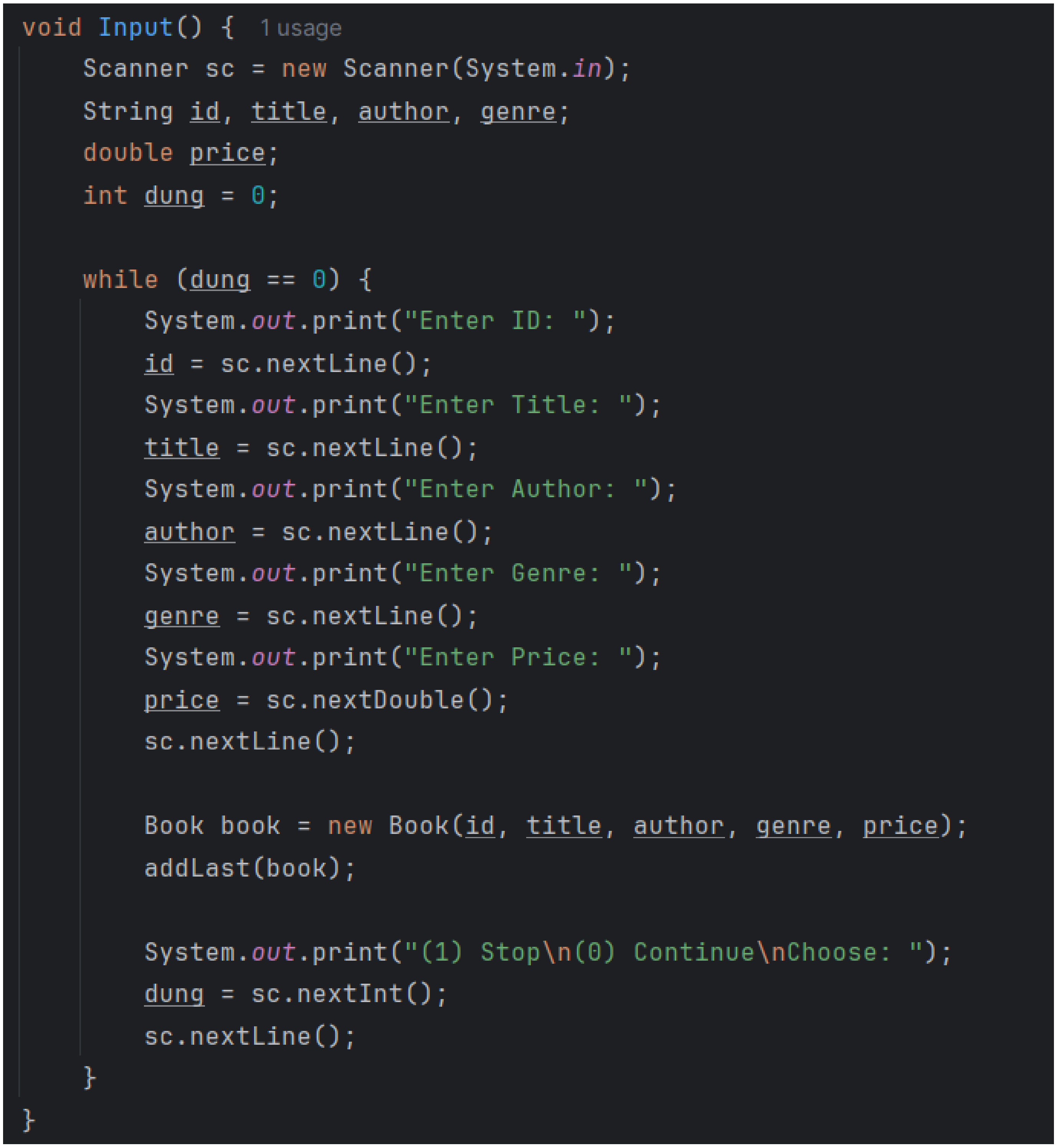
**Figure 26: Source Code 3.**

- Class BookstoreManager:



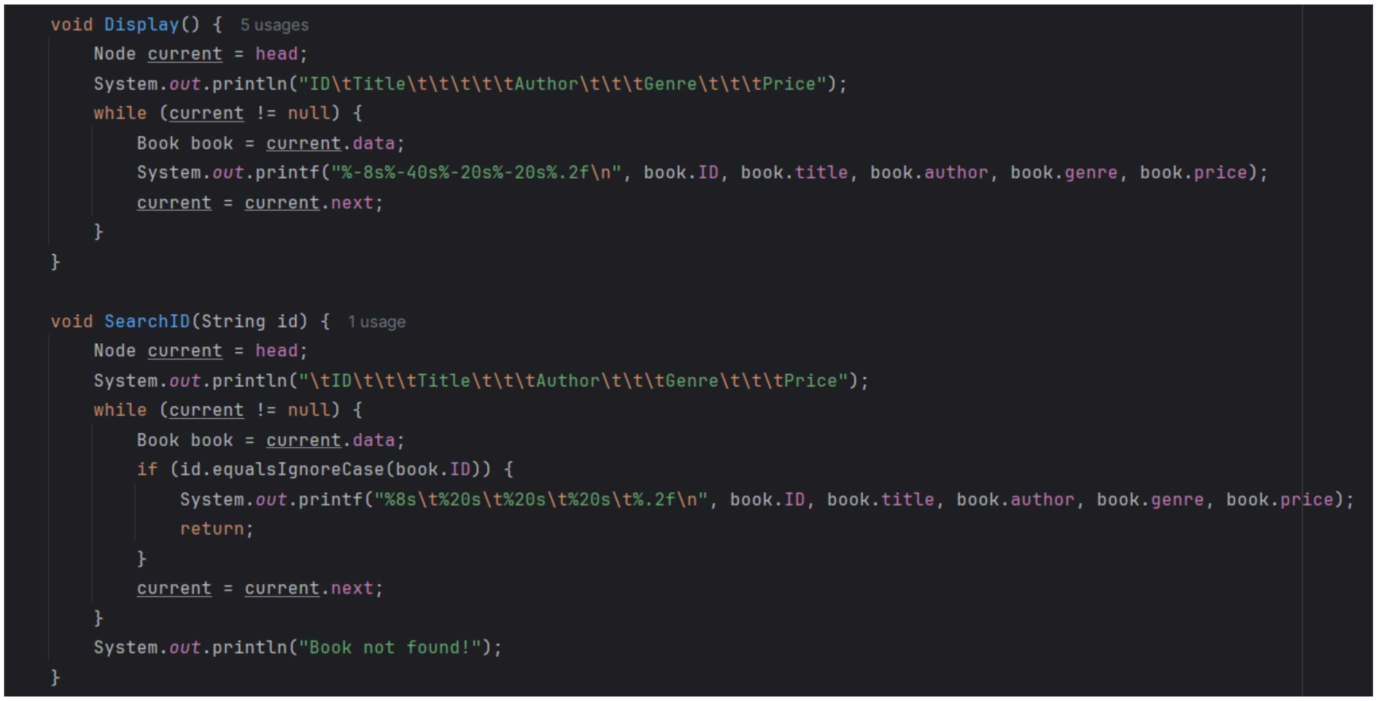
```
package asm1;

import java.util.Scanner;

public class BookstoreManager {  2 usages
    Node head = null;  15 usages
    Node tail = null;  8 usages

    void addFirst(Book book) {  1 usage
        Node newNode = new Node(book);
        if (head == null)
            head = tail = newNode;
        else {
            newNode.next = head;
            head = newNode;
        }
    }

    void addLast(Book book) {  2 usages
        Node newNode = new Node(book);
        if (head == null)
            head = tail = newNode;
        else {
            tail.next = newNode;
            tail = newNode;
        }
    }
```

**Figure 27: Source Code 4.**



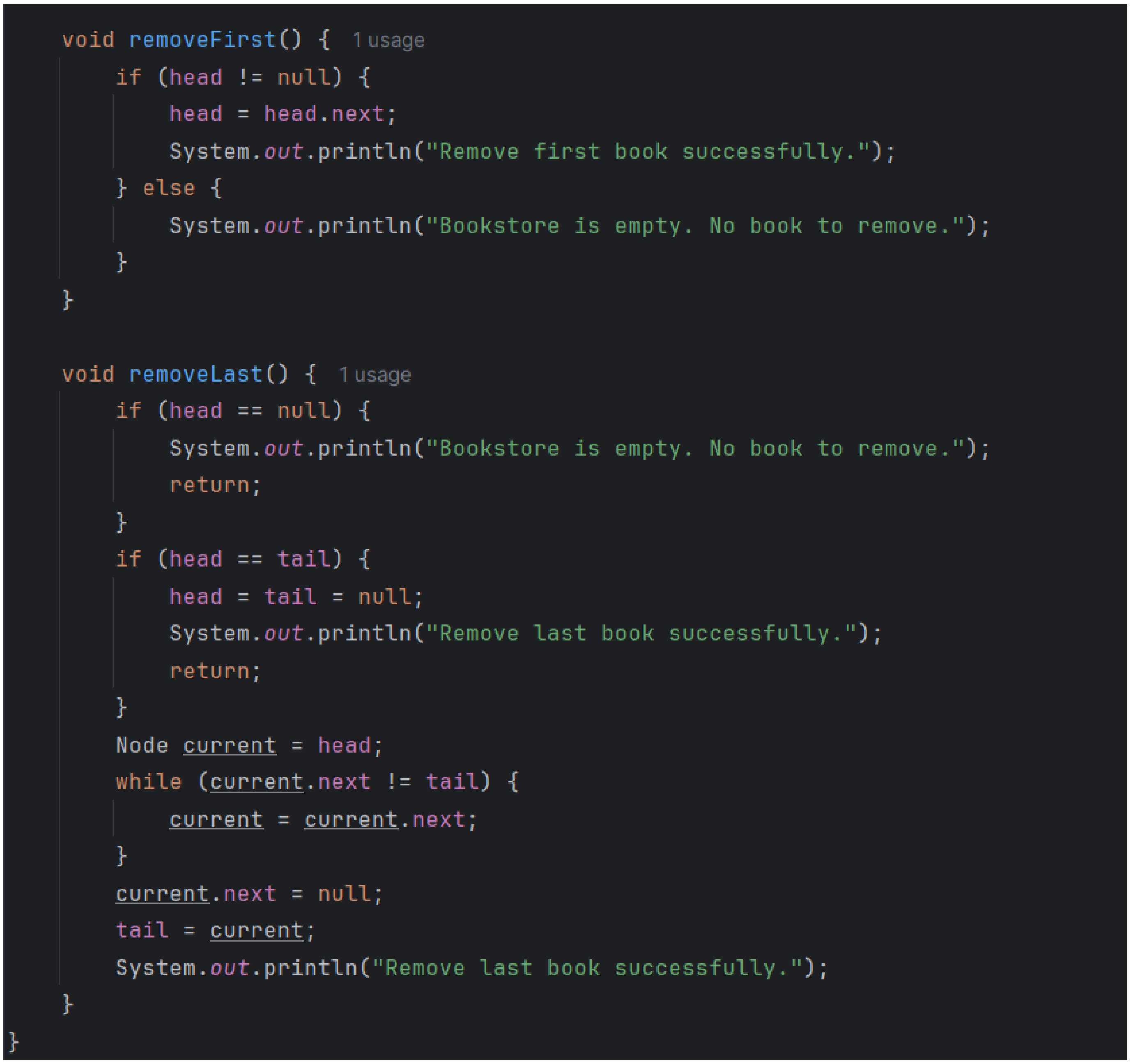
```java
void Input() {  1 usage
    Scanner sc = new Scanner(System.in);
    String id, title, author, genre;
    double price;
    int dung = 0;

    while (dung == 0) {
        System.out.print("Enter ID: ");
        id = sc.nextLine();
        System.out.print("Enter Title: ");
        title = sc.nextLine();
        System.out.print("Enter Author: ");
        author = sc.nextLine();
        System.out.print("Enter Genre: ");
        genre = sc.nextLine();
        System.out.print("Enter Price: ");
        price = sc.nextDouble();
        sc.nextLine();

        Book book = new Book(id, title, author, genre, price);
        addLast(book);

        System.out.print("(1) Stop\n(0) Continue\nChoose: ");
        dung = sc.nextInt();
        sc.nextLine();
    }
}
```

**Figure 28: Source Code 5.**



```
void Display() {  5 usages
    Node current = head;
    System.out.println("ID\tTitle\t\t\t\t\tAuthor\t\t\tGenre\t\t\tPrice");
    while (current != null) {
        Book book = current.data;
        System.out.printf("%-8s%-40s%-20s%-20s%.2f\n", book.ID, book.title, book.author, book.genre, book.price);
        current = current.next;
    }
}

void SearchID(String id) {  1 usage
    Node current = head;
    System.out.println("\tID\t\t\tTitle\t\t\tAuthor\t\t\tGenre\t\t\tPrice");
    while (current != null) {
        Book book = current.data;
        if (id.equalsIgnoreCase(book.ID)) {
            System.out.printf("%8s\t%20s\t%20s\t%20s\t%.2f\n", book.ID, book.title, book.author, book.genre, book.price);
            return;
        }
        current = current.next;
    }
    System.out.println("Book not found!");
}
```

**Figure 29: Source Code 6.**

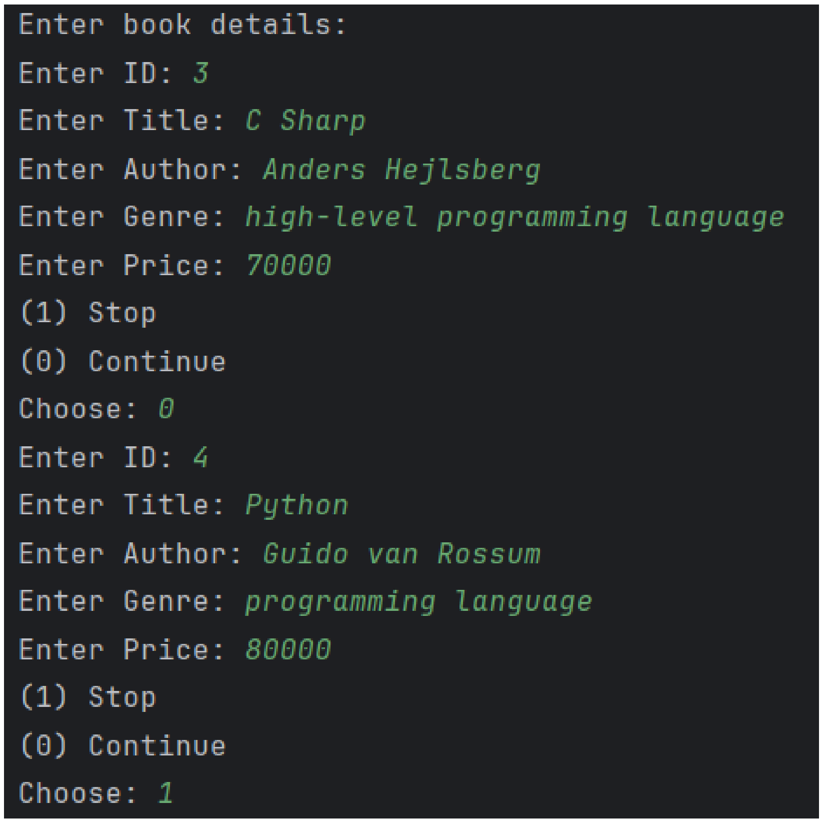


```
    void removeFirst() {  1 usage
        if (head != null) {
            head = head.next;
            System.out.println("Remove first book successfully.");
        } else {
            System.out.println("Bookstore is empty. No book to remove.");
        }
    }

    void removeLast() {  1 usage
        if (head == null) {
            System.out.println("Bookstore is empty. No book to remove.");
            return;
        }
        if (head == tail) {
            head = tail = null;
            System.out.println("Remove last book successfully.");
            return;
        }
        Node current = head;
        while (current.next != tail) {
            current = current.next;
        }
        current.next = null;
        tail = current;
        System.out.println("Remove last book successfully.");
    }
}
```

**Figure 30: Source Code 7.**

- **Output:**

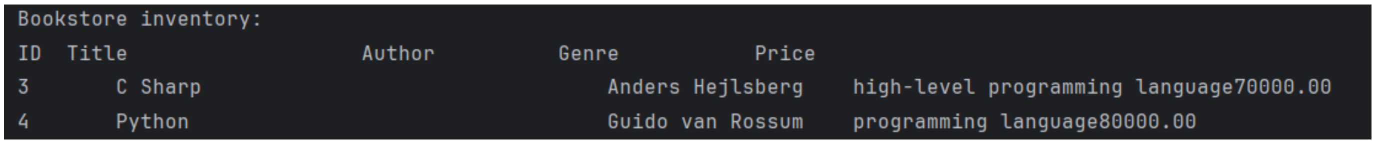
I use the keyboard to enter information such as ID, Title, Author, Genre, Price.



```
Enter book details:
Enter ID: 3
Enter Title: C Sharp
Enter Author: Anders Hejlsberg
Enter Genre: high-level programming language
Enter Price: 70000
(1) Stop
(0) Continue
Choose: 0
Enter ID: 4
Enter Title: Python
Enter Author: Guido van Rossum
Enter Genre: programming language
Enter Price: 80000
(1) Stop
(0) Continue
Choose: 1
```

**Figure 31: Output 1**

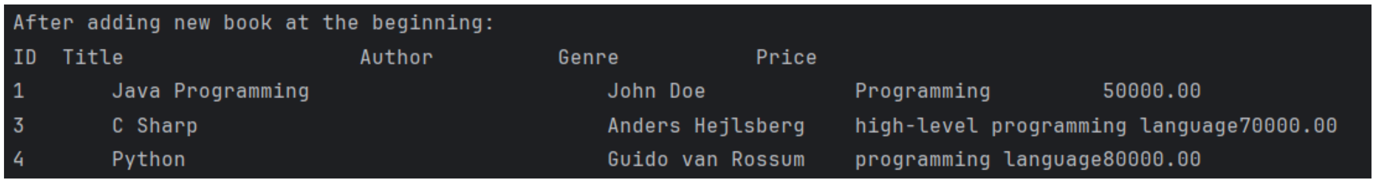
Displays the title of a book data table in an online bookstore. Columns include ID, Title, Author, Genre, and Price.



```
Bookstore inventory:
ID  Title                   Author          Genre          Price
3       C Sharp                                 Anders Hejlsberg    high-level programming language70000.00
4       Python                                  Guido van Rossum    programming language80000.00
```

**Figure 32: Output 2.**

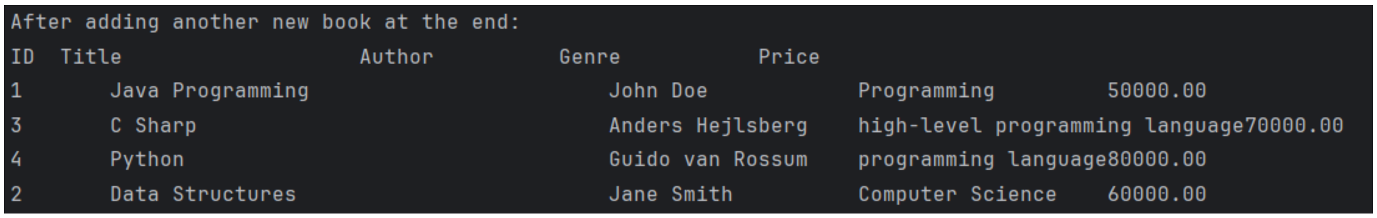
Displays a list of books after adding a new book to the top of the list. The new book added to the top of the list has ID as 1, title as "Java Programming", author as "John Doe", genre as "Programming", and price as 50000.



```
After adding new book at the beginning:
ID  Title                   Author          Genre          Price
1       Java Programming                        John Doe            Programming         50000.00
3       C Sharp                                 Anders Hejlsberg    high-level programming language70000.00
4       Python                                  Guido van Rossum    programming language80000.00
```

**Figure 33: Output 3.**

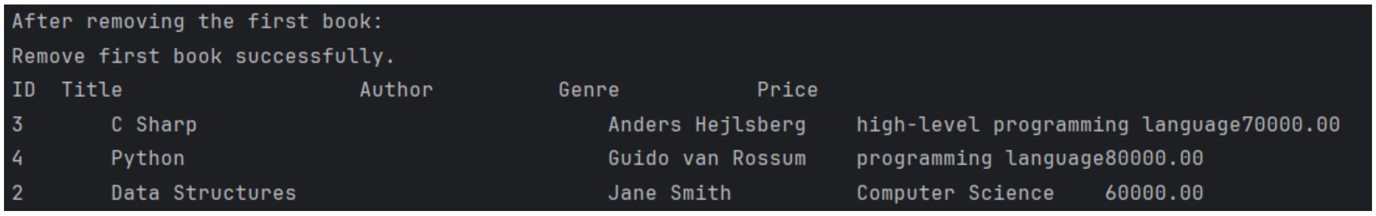
Shows the book list after adding a new book to the end of the list. The new book added to the bottom of the list has ID 2, title is "Data Structures", author is "Jane Smith", genre is "Computer Science", and price is 60000.



```
After adding another new book at the end:
ID  Title                   Author          Genre          Price
1       Java Programming                        John Doe            Programming         50000.00
3       C Sharp                                 Anders Hejlsberg    high-level programming language70000.00
4       Python                                  Guido van Rossum    programming language80000.00
2       Data Structures                         Jane Smith          Computer Science    60000.00
```

**Figure 34: Output 4.**

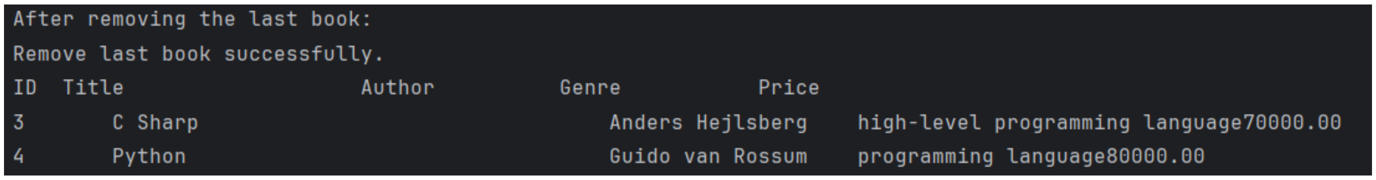
List of books after removing the first book from the list. In this case, the first book with ID 1 and title "Java Programming" has been removed from the list.



```
After removing the first book:
Remove first book successfully.
ID  Title                   Author          Genre          Price
3       C Sharp                                 Anders Hejlsberg    high-level programming language70000.00
4       Python                                  Guido van Rossum    programming language80000.00
2       Data Structures                         Jane Smith          Computer Science    60000.00
```

**Figure 35: Output 5.**

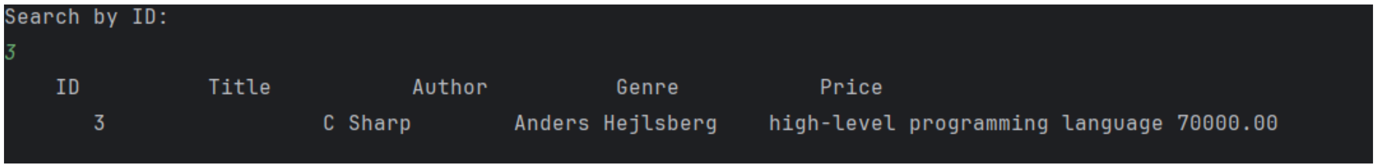
Book list after removing the last book from the list. In this case, the last book in the list with ID 2 and title "Data Structures" has been removed from the list.



```
After removing the last book:
Remove last book successfully.
ID  Title                   Author          Genre           Price
3       C Sharp                                 Anders Hejlsberg    high-level programming language70000.00
4       Python                                  Guido van Rossum    programming language80000.00
```

**Figure 36: Output 6.**

book with ID 3 was found. The title of the book is "C Sharp", the author is "Anders Hejlsberg", the genre is "high-level programming language", and the price is 70000.



```
Search by ID:
3
    ID          Title           Author          Genre           Price
       3                 C Sharp        Anders Hejlsberg    high-level programming language 70000.00
```

**Figure 37: Output 7.**

## REFERENCES:

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

Available at: https://www.tutorialspoint.com/basics_of_computers/basics_of_computers_primary_memory.htm

Available at: https://www.tutorialspoint.com/basics_of_computers/index.htm