ASSIGNMENT FINAL REPORT

Qualification	**Pearson BTEC Level 5 Higher National Diploma in Computing**		
Unit number and title	**Unit 19: Data Structures and Algorithms**		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Tuan Anh	**Student ID**	BH00667
Class	SE06203	**Assessor name**	Ta Quang Hieu

Student’s signature	Anh

Grading grid

P1	P2	P3	P4	P5	P6	P7	M1	M2	M3	M4	M5	D1	D2	D3	D4

❐ Summative Feedback:	❐ Resubmission Feedback:	
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
Signature & Date:		

Table of Contents

I. Introduction

Soft Development ABK, a software workshop dedicated to providing tailored solutions for small and medium enterprises, is continuously striving to improve the quality and efficiency of its software projects. As part of this effort, abstract data types (ADTs) have been identified as a crucial tool for enhancing design, development, and testing processes. The purpose of this report is to introduce ADTs and illustrate their benefits, particularly in the context of a real-life application: a student ranking system.

The student ranking system will allow users to manage student information, including ID, name, and marks. Based on these marks, students will be classified into ranks ranging from 'Fail' to 'Excellent'. The system will support essential operations such as adding, editing, deleting, sorting, and searching for students. By leveraging ADTs, we aim to create a robust, maintainable, and efficient application that meets the needs of our users.

Additionally, the report will compare the current algorithmic approach used in the system with an alternative, evaluating the effectiveness of each to ensure we are using the most suitable method for our requirements.

II. Body

LO1. Examine abstract data types, concrete data structures and algorithms

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

1. Data Structure

Figure 1: Data Structure

Definition

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed.

Type of data structure

Data structures are divided into 2 types:

- Linear data structure
- Non-linear data structure

Linear data structure

A linear data structure is a data structure where data elements are organized sequentially. In a linear data structure, each element has a unique predecessor and successor, except for the first and last elements. The elements are accessed and processed in a linear order, typically from one end to the other.

Common examples of linear data structures include arrays, linked lists, stacks, and queues. Let's briefly discuss these structures:

Array

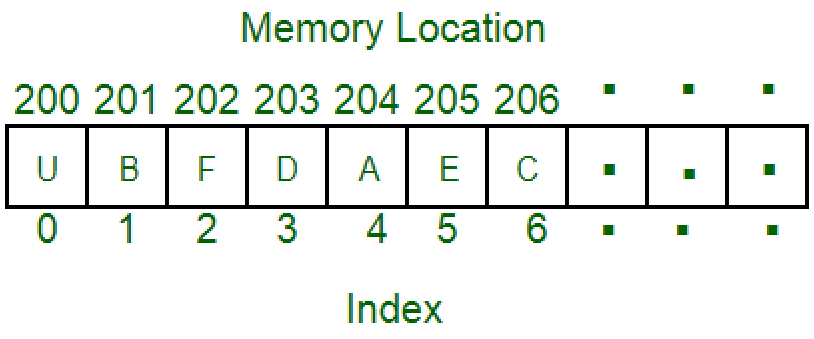
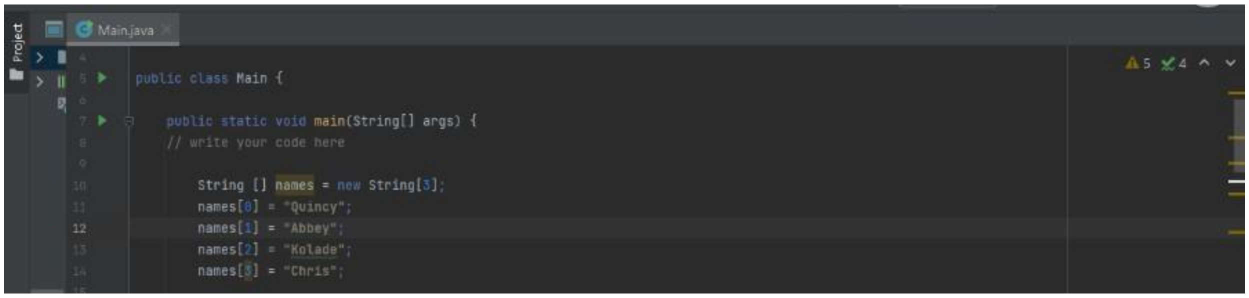


Figure 2: Array

Definition

An array is a fixed-size collection of elements stored in contiguous memory locations. Elements can be accessed using their index. Arrays offer constant-time access to elements but have a fixed size, which cannot be easily changed.

Example:



```
public class Main {

    public static void main(String[] args) {
    // write your code here

        String [] names = new String[3];
        names[0] = "Quincy";
        names[1] = "Abbey";
        names[2] = "Kolade";
        names[3] = "Chris";
```

Figure 3: Array

Operation

- An Array are different operations possible in an array, like Searching, Sorting, Inserting, Traversing, Reversing, and Deleting.
- Initialization: An array can be initialized with values at the time of declaration or later using an assignment statement.
- Accessing elements: Elements in an array can be accessed by their index, which starts from 0 and goes up to the size of the array minus one.
- Searching for elements: Arrays can be searched for a specific element using linear search or binary search algorithms.
- Sorting elements: Elements in an array can be sorted in ascending or descending order using algorithms like bubble sort, insertion sort, or quick sort.
- Inserting elements: Elements can be inserted into an array at a specific location, but this operation can be time-consuming because it requires shifting existing elements in the array.
- Deleting elements: Elements can be deleted from an array by shifting the elements that come after it to fill the gap.
- Updating elements: Elements in an array can be updated or modified by assigning a new value to a specific index.
- Traversing elements: The elements in an array can be traversed in order, visiting each element once.

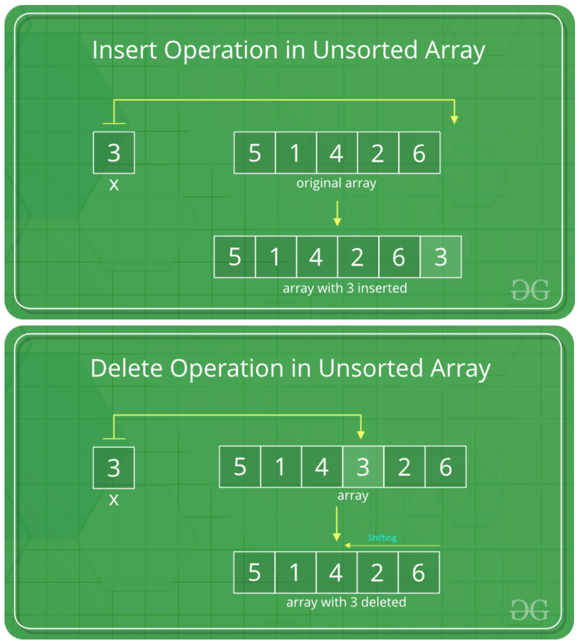


Figure 4: Operation in array

Linked List

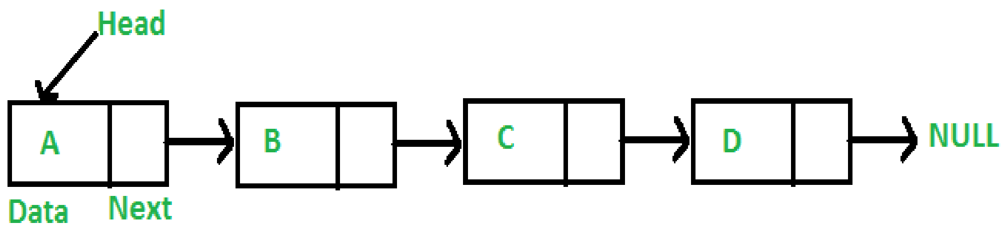
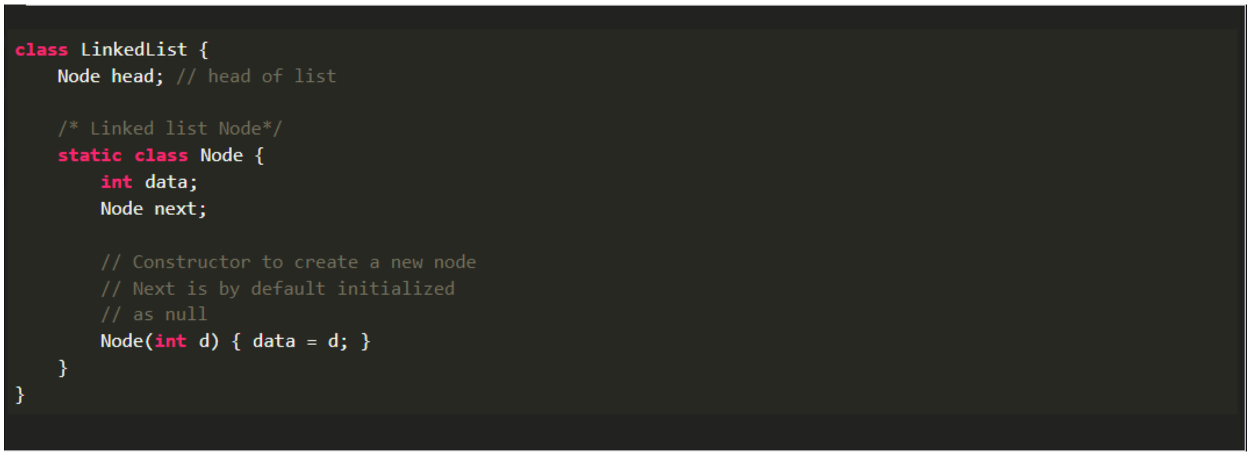


Figure 5: Linked List

Definition

- A linked list is a linear data structure in which elements are not stored at contiguous memory locations.
- A linked list consists of nodes containing a data element and a reference (pointer) to the next node in the sequence. Linked lists provide dynamic memory allocation and efficient insertion/deletion at the beginning or end of the list. However, random access to elements is slower compared to arrays.
- Linked lists use additional storage to store links. During the initialization of the linked list, it is not necessary to know the size of the elements. Linked lists are used to implement stacks, queues, graphs, and more. The first node of the linked list is called Head.
- The next cursor of the last node always points to NULL. In a linked list, can be inserted and deleted easily. Each node of the linked list includes a pointer/link that is the address of the next node. The linked list can be minimized or grown at any time easily.

Example:



```
class LinkedList {
    Node head; // head of list

    /* Linked list Node*/
    static class Node {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized
        // as null
        Node(int d) { data = d; }
    }
}
```

Figure 6: LinkedList

Operation

- A linked list is a linear data structure where each node contains a value and a reference to the next node. Here are some common operations performed on linked lists:
- Initialization: A linked list can be initialized by creating a head node with a reference to the first node. Each subsequent node contains a value and a reference to the next node.
- Inserting elements: Elements can be inserted at the head, tail, or at a specific position in the linked list.
- Deleting elements: Elements can be deleted from the linked list by updating the reference of the previous node to point to the next node, effectively removing the current node from the list.
- Searching for elements: Linked lists can be searched for a specific element by starting from the head node and following the references to the next nodes until the desired element is found.
- Updating elements: Elements in a linked list can be updated by modifying the value of a specific node.
- Traversing elements: The elements in a linked list can be traversed by starting from the head node and following the references to the next nodes until the end of the list is reached.
- Reversing a linked list: The linked list can be reversed by updating the references of each node so that they point to the previous node instead of the next node.

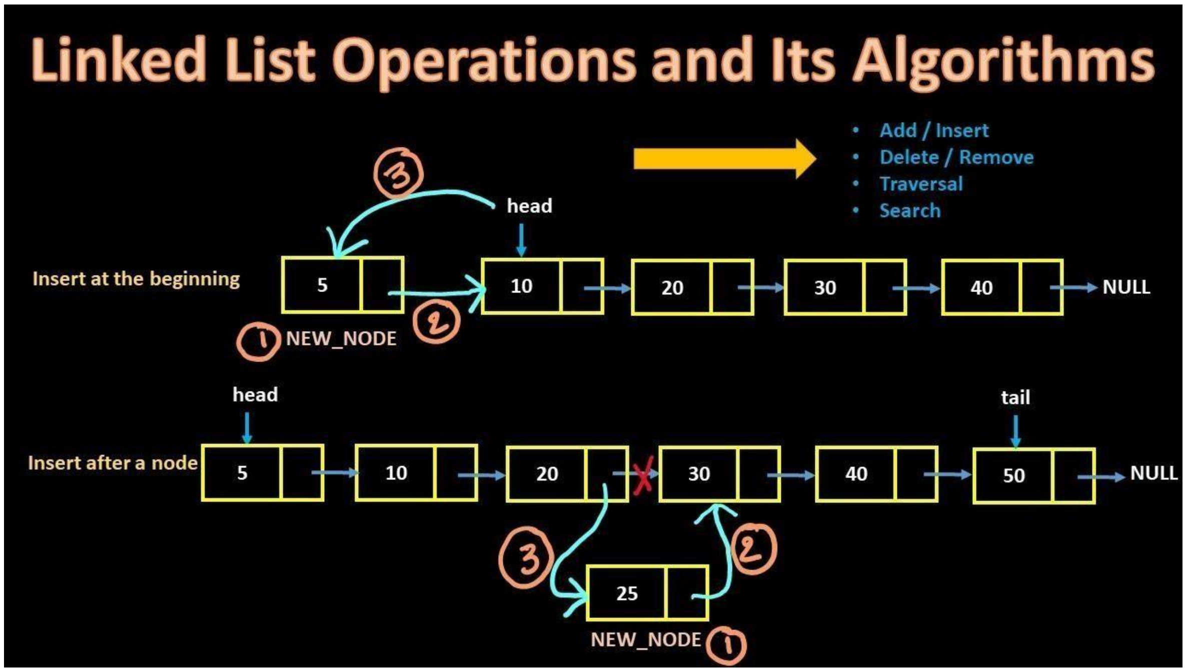


Figure 7: Operation in LinkedList

Stack

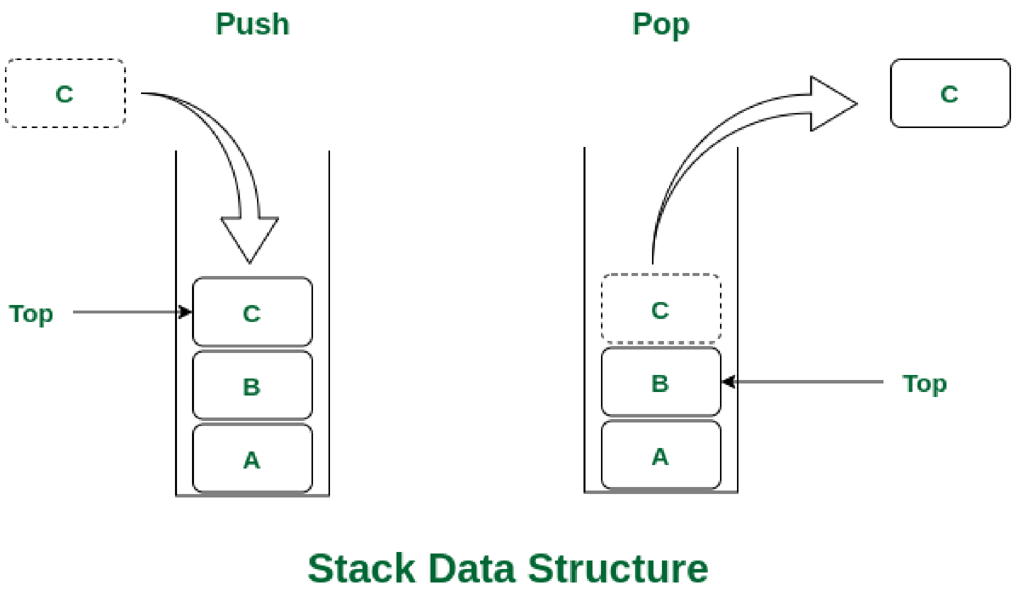
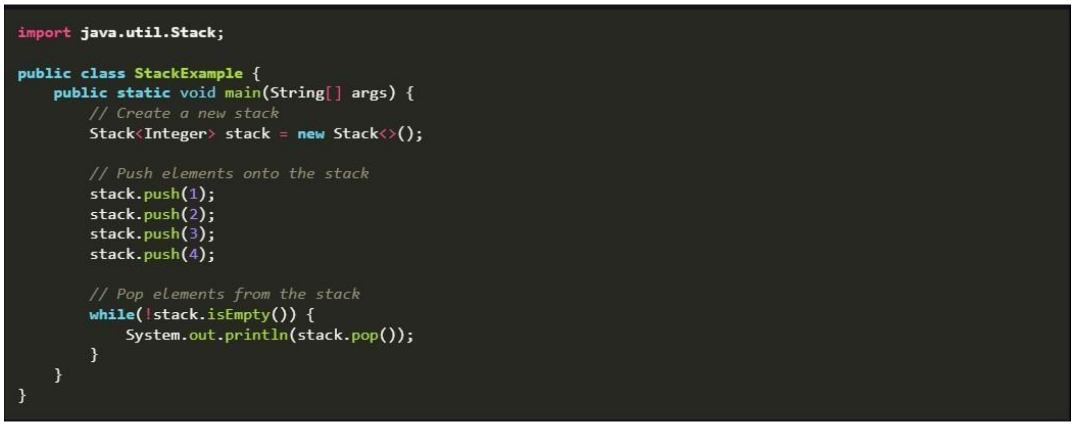


Figure 8: Stack

Definition

- A stack is a Last In First Out (LIFO) data structure, where elements are added or removed from one end called the top. It follows a "push" operation to add an element and a "pop" operation to remove the topmost element. Stacks are commonly used in algorithms involving recursion, backtracking, and expression evaluation.

  Example:



```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Create a new stack
        Stack<Integer> stack = new Stack<>();

        // Push elements onto the stack
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);

        // Pop elements from the stack
        while(!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

Figure 9: Stack code implementation

Operation

- A stack is a linear data structure that implements the Last-In-First-Out (LIFO) principle.
- Here are some common operations performed on stacks:

  + Push: Elements can be pushed onto the top of the stack, adding a new element to the top of the stack.

  + Pop: The top element can be removed from the stack by performing a pop operation, effectively removing the last element that was pushed onto the stack.

  + Peek: The top element can be inspected without removing it from the stack using a peek operation.

  + IsEmpty: A check can be made to determine if the stack is empty.

  + Size: The number of elements in the stack can be determined using a size operation.

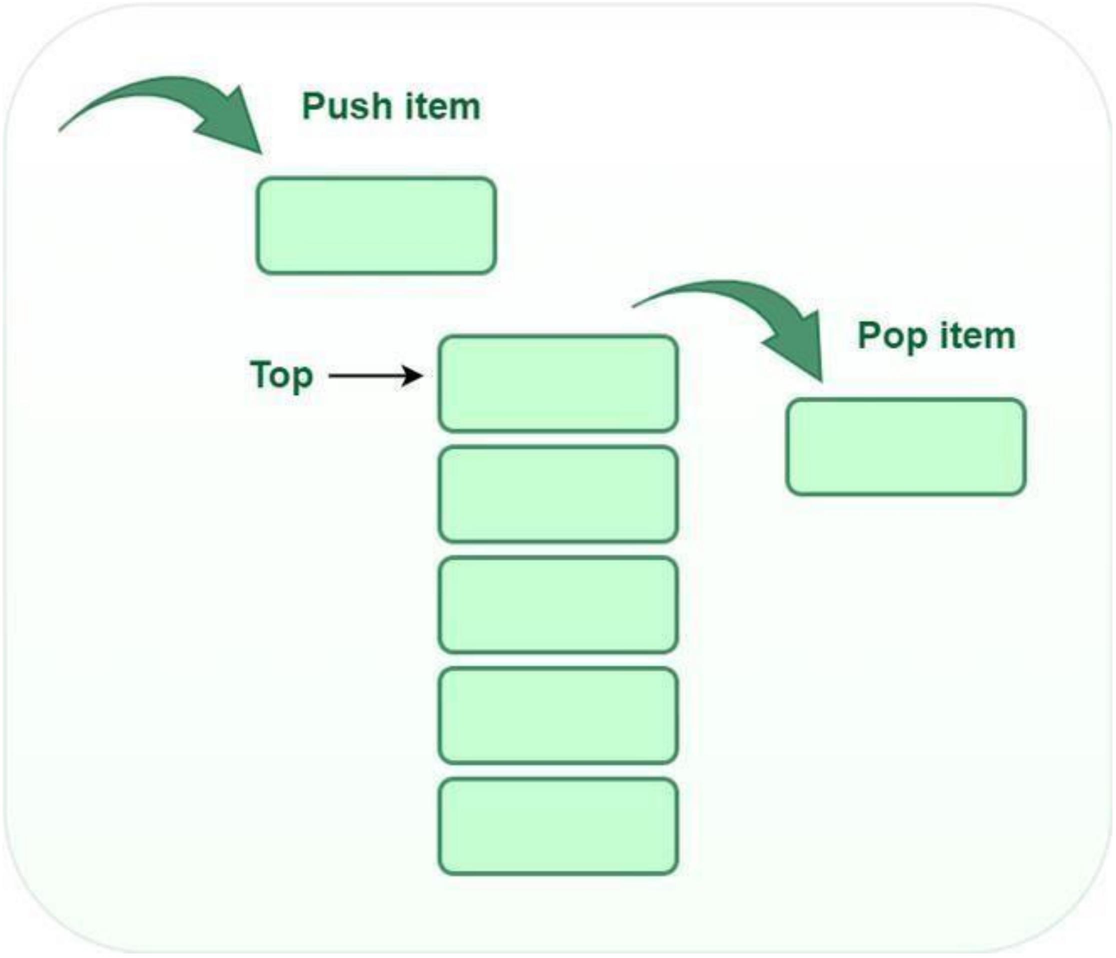
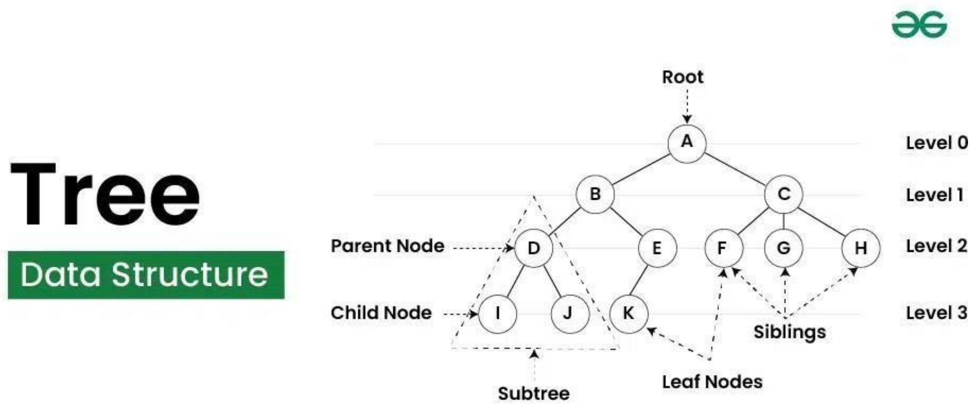


Figure 10: Operation in Stack

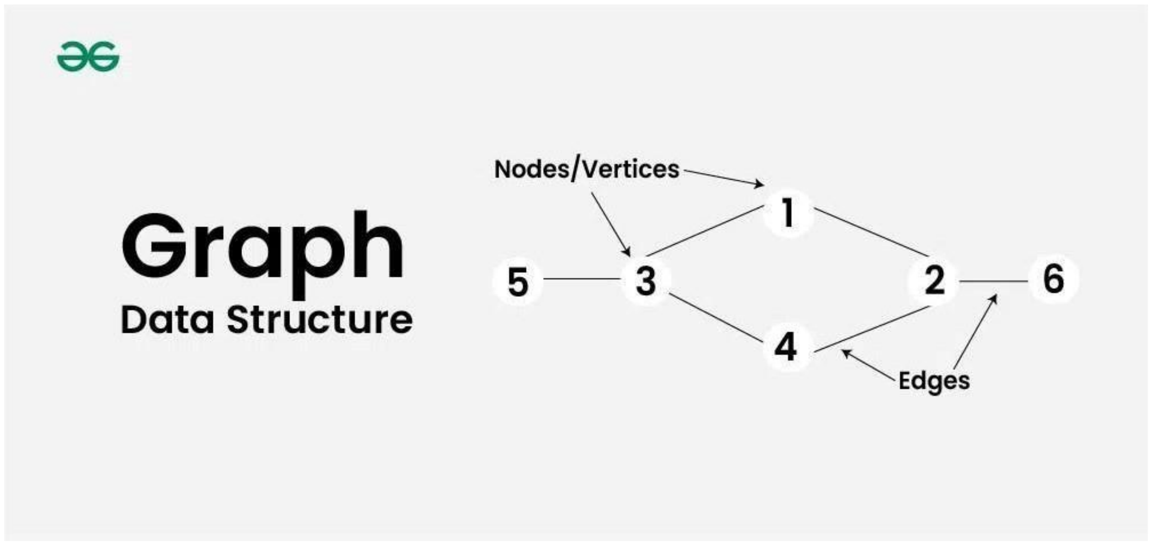
Non-linear data structure

- A non-linear data structure is a type of data organization where data elements are not arranged sequentially. Unlike linear data structures, non-linear structures do not have a strict predecessor-successor relationship between elements. Instead, they allow for more complex relationships and connections among elements.
- Here are some common examples of non-linear data structures:

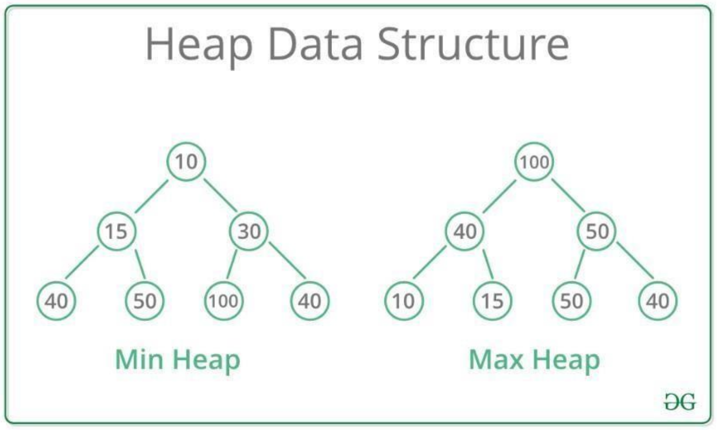
- Tree: A tree is a hierarchical data structure composed of nodes connected by edges. It has a root node at the top and child nodes branching out from the root or other nodes. Each node can have zero or more child nodes. Trees are widely used for organizing hierarchical data, such as file systems, organization structures, and decision-making scenarios.



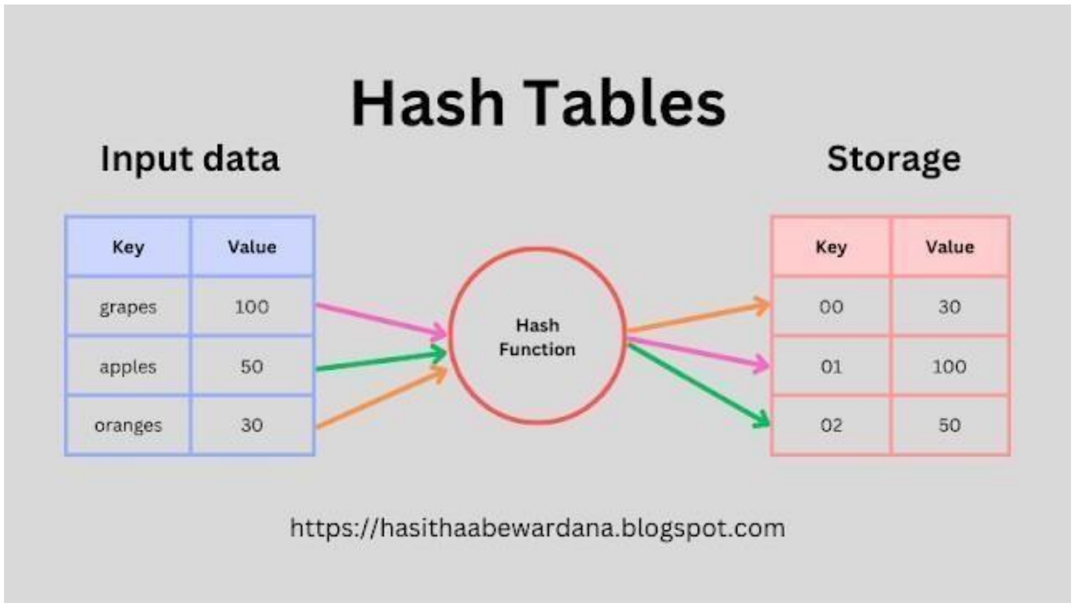
- Graph: A graph is a collection of nodes (vertices) and the connections between them (edges). Unlike trees, graphs can have arbitrary connections between any pair of nodes. Graphs are used to represent relationships, networks, social connections, transportation systems, and more. They can be directed (edges have a specific direction) or undirected (edges have no specific direction).



- Heap: A heap is a specialized tree-based data structure that satisfies the heap property. It is commonly used to implement priority queues. A heap ensures that the highest (or lowest) priority element is always at the root, allowing efficient access to the most important element.



- Hash Table: A hash table is a data structure that uses a hash function to map keys to values. It provides fast insertion, deletion, and retrieval operations. Hash tables are useful when quick key-based access to data is required. They are commonly used in database indexing, caches, and lookup tables.



Non-linear data structures offer flexibility in representing relationships and solving complex problems. They provide efficient access and manipulation of data in various scenarios. The choice of a non-linear data structure depends on the specific requirements of the problem and the relationships among the data elements.

Input parameters

- An input parameter in data structure refers to the data passed to a function, method, or algorithm that influences its operation. These parameters allow for the customization and dynamic operation of functions, methods, and algorithms.
- In the context of data structures, input parameters are often used to specify the values to be stored, the keys for lookup operations, or any other data needed to perform specific operations.
- Some types of input parameter are include:
    + Primitive Types: These include basic data types such as integers, floats, characters, and booleans.
    + Composite Types: These include arrays, lists, tuples, and other collections.
    + User-defined Types: These include objects and instances of classes that are created by the user.
    + Function References: Functions themselves can be passed as parameters to other functions (higher-order functions).

Time and Space Complexity

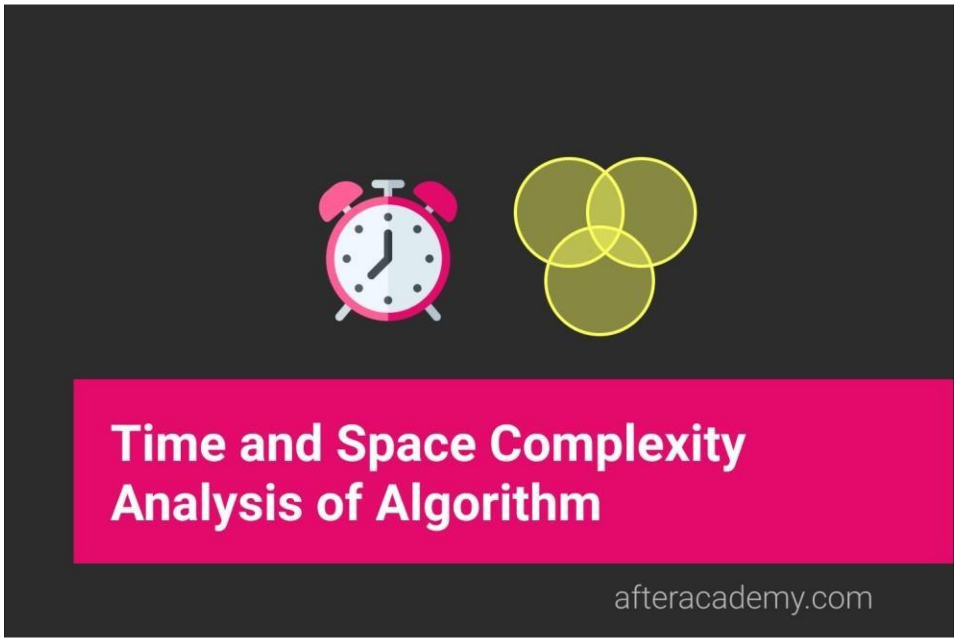


Figure 11: Time and Space Complexity

Time complexity

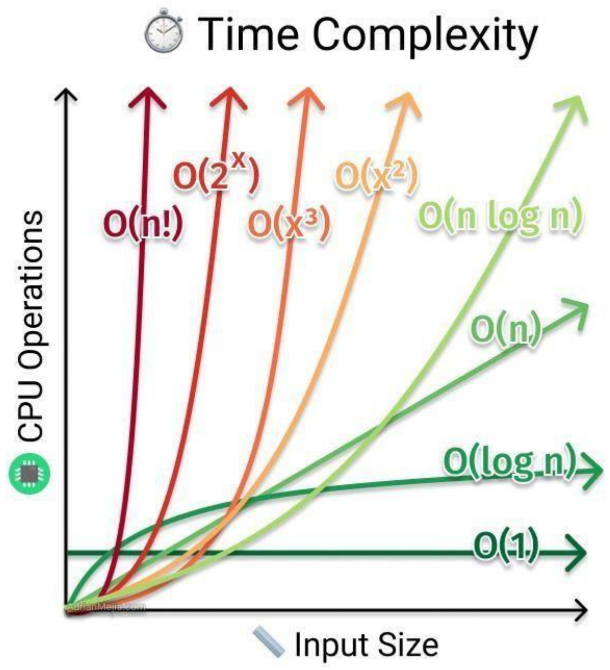


Figure 12: Time complexity

- Time complexity refers to the amount of time required by an algorithm to run as a function of the input size. It helps us understand how the algorithm's runtime grows with the increase in input size. The time complexity is typically expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm.
- Some types of time complexity:

  + Constant Time (O (1)): The execution time remains constant regardless of the input size.

  + Linear Time (O (n)): The execution time increases linearly with the input size. For instance, finding an item in an unsorted list.

+ Logarithmic Time (O (log n)): The execution time increases logarithmically with an increase in input size. Binary search is a classic example.
+ Quadratic Time (O(n^2)): The time increases quadratically with the input size. This is common in algorithms with nested loops over the input data.
+ Exponential Time: O(2^n): The execution time doubles with each addition to the input data set. This is typical of algorithms that solve problems by computing all possible combinations.
Space complexity

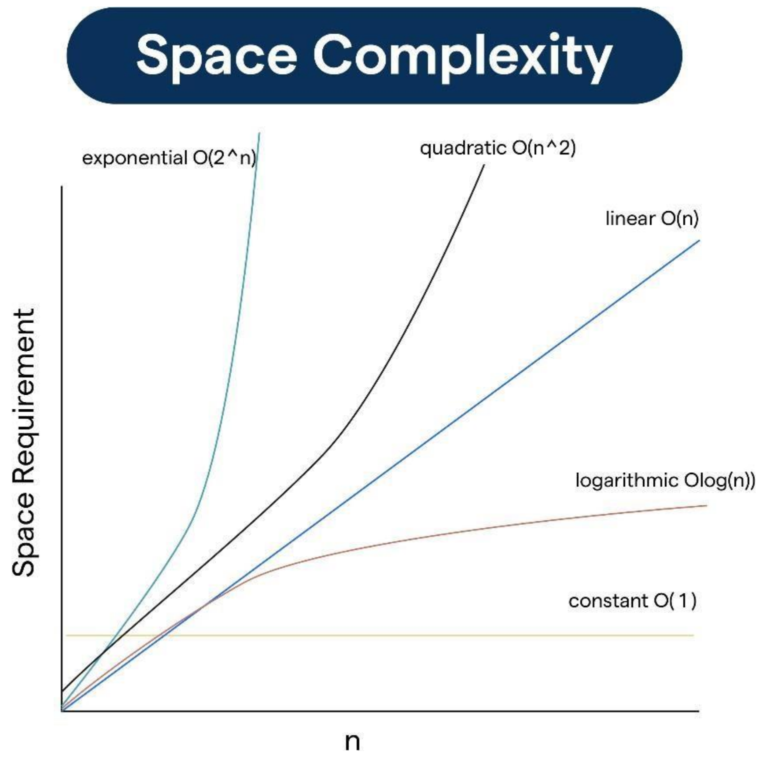


Figure 13: Space complexity

- Space complexity refers to the amount of memory required by an algorithm to run as a function of the input size. It helps us understand how the algorithm's memory usage grows with the increase in input size. Like time complexity, space complexity is also expressed using Big O notation.
- Some examples of time complexity:

+ Constant Space (O (1)): The algorithm uses a fixed amount of memory space regardless of the input size. For example, an algorithm that swaps two numbers.

+ Linear Space (O(n)): The memory required grows linearly with the input size. An example is creating a list of 'n' elements.

+ Quadratic Space: O(n^2): The space requirement grows quadratically with the input size, commonly seen in algorithms that create two-dimensional arrays based on the input size.

Advantages and disadvantages of data structure

- Advantages of Data Structures:

- Efficient Data Access: Well-designed data structures provide efficient access and retrieval of data elements. They enable quick search, insertion, deletion, and modification operations, which can improve the overall performance of algorithms and programs.
- Optimal Memory Utilization: Data structures help in optimizing memory usage by organizing and storing data in a structured manner. Efficient data organization reduces memory overheads, improves cache utilization, and minimizes memory fragmentation.
- Scalability: Data structures enable handling large amounts of data and facilitate scalability. They allow for the efficient management of growing datasets without significant performance degradation.
- Code Reusability: Well-defined data structures promote code reusability. Once implemented, data structures can be reused across multiple programs or modules, saving development time and effort.
- Problem Solving: Different data structures are designed to solve specific types of problems efficiently. By choosing the appropriate data structure, developers can solve complex problems more easily and effectively.

- Disadvantages of Data Structures:

- Complexity: Some advanced data structures can be complex to understand, implement, and maintain. They may require a deep understanding of algorithms and data organization principles.
- Overhead: Certain data structures may introduce additional overhead in terms of memory usage or computational complexity. For example, maintaining self-balancing properties in a tree data structure requires extra computational resources.
- Trade-offs: Different data structures have their own trade-offs. While some structures may provide excellent search performance, they may have slower insertion or deletion operations. Developers need to consider these trade-offs based on the specific requirements of their applications.
- Learning Curve: Understanding and effectively utilizing various data structures can require a learning curve. Developers need to invest time and effort in learning the characteristics, operations, and best practices associated with different data structures.
- Implementation Complexity: Implementing certain data structures from scratch can be challenging, especially for complex ones like balanced trees or graphs. It may require advanced programming skills and expertise

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

2. Memory Stack

2.1 Definition

- The stack is a segment of memory that stores temporary variables created by a function. In stack, variables are declared, stored and initialized during runtime.
- When we compile a program, the compiler enters through the main function and a stack frame is created on the stack. A structure, also known as an activation record, is the collection of all data on the stack associated with one subprogram call. The main function and all the local variables are stored in an initial frame.
- It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. The stack section mostly contains methods, local variables, and reference variables.

2.2 Operations

- The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated.

- This all happens using some predefined routines in the compiler. A programmer does not have to worry about memory allocation and de-allocation of stack variables. This kind of memory allocation is also known as Temporary memory allocation because as soon as the method finishes its execution all the data belonging to that method flushes out from the stack automatically.

- This means any value stored in the stack memory scheme is accessible if the method hasn't completed its execution and is currently in a running state.

2.3 Call function

- The basic operations of Stack memory include: Push and Pop. These two operations are as follows:

+ Push:
  - Adding an element on the top of the stack is termed a push operation. Push operation has the following two steps:
  - Increment the top variable of the stack so that it can refer to the next memory location.
  - Add a data element at the increment top position.
  - Stack data structure states an overflow condition when you try to insert an element into the stack when complete.

+ Pop:
  - Removing a data element from the stack data structure is called a pop operation. The pop operation has two following steps:
  - The value of the top variable will be incremented by one whenever you delete an item from the stack.
  - The topmost variable of the stack is stored in another variable, and then the value of the top variable will be decremented by one.
  - The pop operation returns the deleted element that was stored in another variable as a result.
  - Stack data structure states an underflow condition when you try to delete a data element when the stack is already empty.
+ Peek:
  - Peek operations involve returning the topmost data element of the stack without removing it from the stack.
  - Underflow conditions may occur if you try to return the topmost element if the stack is already empty.

2.4 Stack frame

2.4.1 Definition

- Stack is one of the segments of application memory that is used to store the local variables, function calls of the function. Whenever there is a function call in our program the memory to the local variables and other function calls or subroutines get stored in the stack frame. Each function gets its own stack frame in the stack segment of the application's memory.
- Stack pointer always points to the top and frame pointer stores address of whole stack frame of the subroutine. Each stack frame of a subroutine or a function contains as follows:

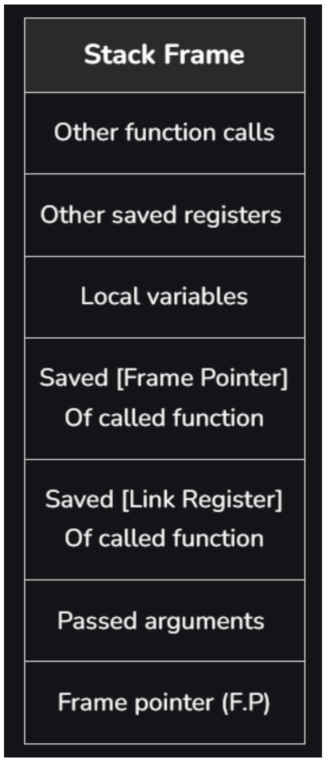


Figure 14: Stack frame

2.3.2 Features

- The memory allocated for a function call in the stack lives only the time when the function is executing once the function gets completed, we can't access the variables of that function.
- Once the calling function completes its execution its stack frame is removed and the thread of execution of called function resumes from that position where it was left.
- Stack is used for storing function calls so in the case when we are using a lot of recursive calls in our program the stack memory gets exhausted by the function calls or subroutines which may result in stack overflow because the stack memory is limited.
- Each stack frame maintains the Stack Pointer (SP), and the frame pointer (FP). Stack pointer and frame pointer always point to the top of the stack. It also maintains a program counter (PC) which points to the next instruction to be executed.
- Whenever a function call is made a stack frame is created in the stack segment the arguments that were given by the calling function get some memory in the stack frame of called function, and they get pushed into the stack frame of the called function. When their execution is finished, they get popped from the stack frame. And the thread of execution gets continues in the called function.

2.4 The importance of Stack

- Stacks are crucial in programming and software development for several reasons:
    + Function call management: Stacks are used to manage function calls in programs. When a function is called, the program pushes the function's context onto the stack, including parameters and local variables. When the function completes execution, its context is popped off the stack, allowing the program to return to the previous point of execution.
    + Memory management: Stacks are used in memory management to store variables and function call information. Local variables, function parameters, return addresses, and other important data are stored on the stack during program execution. This helps in efficient memory allocation and deallocation.
    + Expression evaluation: Stacks are used in evaluating expressions, such as infix, postfix, and prefix expressions. For example, the postfix notation (also known as Reverse Polish Notation) can be evaluated efficiently using a stack.
    + Undo mechanisms: Stacks are used in implementing undo mechanisms in applications. Each action performed by the user is pushed onto the stack, and the user can undo these actions by popping items off the stack in reverse order.
    + Backtracking: Stacks are used in algorithms that require backtracking, such as depth-first search in graphs and tree traversal algorithms. The stack keeps track of the nodes to be visited or explored, allowing the algorithm to backtrack when necessary.

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

3. First In First Out (FIFO) approach

3.1 Definition

- FIFO is an abbreviation for first in, first out. It is a method for handling data structures where the first element is processed first and the newest element is processed last.
- Example:
    + Assume a company purchased 100 items for $10 each, then purchased 100 more items for $15 each. The company sold 60 items. Under the FIFO method, the COGS for each of the 60 items is $10/unit because the first goods purchased are the first goods sold. Of the 140 remaining items in inventory, the value of 40 items is $10/unit, and the value of 100 items is $15/unit because the inventory is assigned the most recent cost under the FIFO method.
    + With this remaining inventory of 140 units, the company sells an additional 50 items. The cost of goods sold for 40 of the items is $10, and the entire first order of 100 units has been fully sold. The other 10 units that are sold have a cost of $15 each, and the remaining 90 units in inventory are valued at $15 each, or the most recent price paid.
- Certain data structures like Queue and other variants of Queue uses FIFO approach for processing data.

4. Queue Data Structure

- A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order.

- It follows the principle of “First in, First out” (FIFO), where the first element added to the queue is the first one to be removed.

- Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

4.1 Queue

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It operates like a line where elements are added at one end (rear) and removed from the other end (front).

4.2 Operation in Queue

4.2.1 Enqueue

- Inserts an element at the end of the queue i.e. at the rear end.
- The following steps should be taken to enqueue (insert) data into a queue:
- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.

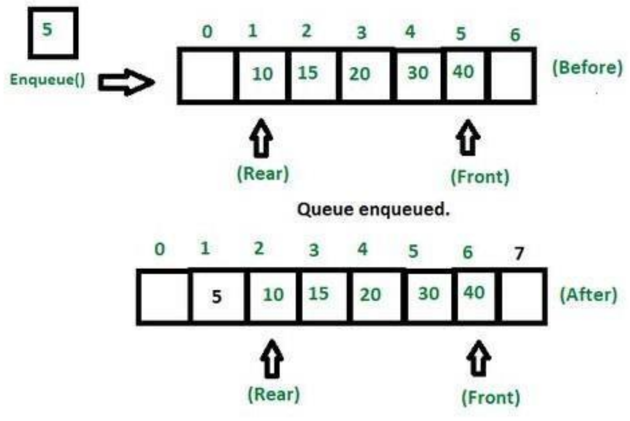


Figure 15: Enqueue Operation

4.2.2 Dequeue

- This operation removes and returns an element that is at the front end of the queue.
- The following steps are taken to perform the dequeue operation:
- Check if the queue is empty.
- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.

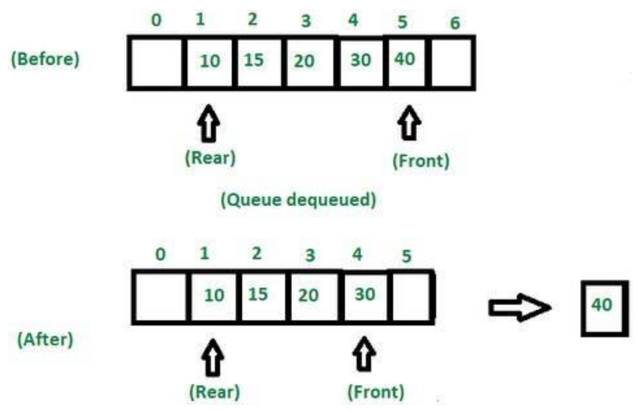


Figure 16: Dequeue Operation

4.3 Implementing a FIFO Queue Using Arrays

- To implement queue using array we need to follow these steps:
- Step 1: Initialize the Queue
- Create an array 'arr' of a fixed size 'n' to hold the elements of the queue. Use two variables, 'front' and 'rear', both initialized to 0. These variables will help in managing the indices of the array:
- 'rear' is the index where the next element will be inserted.
- 'front' is the index of the first element in the queue.
- Step 2: Define the Enqueue Operation
- The enqueue operation adds an element to the end of the queue:
- Check if the queue is full. If 'rear' is equal to n, it means the queue has reached its maximum capacity, and no more elements can be added.
- If the queue is not full, insert the element at the index 'rear' and then increment the 'rear' index by 1.
- Step 3: Define the Dequeue Operation
- The dequeue operation removes an element from the 'front' of the queue:
- Check if the queue is empty. If 'front' is equal to 'rear', it means the queue is empty, and there are no elements to remove.

- If the queue is not empty, remove the element at the index ‘front’ and then increment the ‘front’ index by 1.
- Optionally, reset the queue when all elements are dequeued by setting front and rear back to 0.
- Step 4: Define Auxiliary Operations
- Other useful operations include checking if the queue is empty, checking if the queue is full, getting the size of the queue, and accessing the front element without removing it.
- Example:

Class Queue:

```
class Queue {
    static private int front, rear, capacity;
    static private int[] queue;
    public Queue(int c)
    {
        front = rear = 0;
        capacity = c;
        queue = new int[capacity];
    }
    public void queueEnqueue(int data)
    {
        if (capacity == rear) {
            System.out.print("\nQueue is full\n");
        }
        else {
            queue[rear] = data;
            rear++;
        }
    }
    public void queueDequeue()
    {
        if (front == rear) {
            System.out.print("\nQueue is empty\n");
        }
        else {
            for (int i = 0; i < rear - 1; i++) {
                queue[i] = queue[i + 1];
            }
            if (rear < capacity)
                queue[rear] = 0;
            rear--;
        }
      }
}
```

```
public class Main {
  public static void main(String[] args) {
    Queue q = new Queue(4);

    // print Queue elements
    q.queueDisplay();

    // inserting elements in the queue
    q.queueEnqueue(20);
    q.queueEnqueue(30);
    q.queueEnqueue(40);
    q.queueEnqueue(50);

    // print Queue elements
    q.queueDisplay();

    // insert element in the queue
    q.queueEnqueue(60);

    // print Queue elements
    q.queueDisplay();

    q.queueDequeue();
    q.queueDequeue();
    System.out.print(
        "\n\nafter two node deletion\n\n");

    // print Queue elements
    q.queueDisplay();

    // print front of the queue
    q.queueFront();
  }
}
```

- The specific methods in the above code are as follows:
- Enqueue: Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If rear == size, then it is said to be an Overflow condition as the array is full. Else which indicates that the array is not full then store the element at arr[rear] and increment rear by 1.
- Dequeue: Removal of an element from the queue. An element can only be deleted when there is at least an element to delete. If front == rear, then it is said to be an Underflow condition as the array is empty. else, the element at arr[front] can be deleted but all the remaining elements must shift to

the left by one position for the dequeue operation to delete the second element from the left on another dequeue operation.

- Front: Get the front element from the queue i.e. arr[front] if the queue is not empty.
- Display: Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.

4.4 Implementing a FIFO Queue Using LinkedList

- To implement a queue using a linked list, we need to follow these steps:
- Step 1: Create the QNode Class
- The QNode class will represent each node in the queue. It will have two data members:
- data of type int, which stores the value of the node.
- next of type QNode, which is a pointer to the next node in the queue.
- We will also define a parameterized constructor that initializes the node with a given value and sets the next pointer to null.
- Step 2: Create the Queue Class
- The Queue class will manage the front and rear pointers of the queue:
- front is a pointer to the front node of the queue.
- rear is a pointer to the rear node of the queue.
- Step 3: Define the Enqueue Operation
- The enqueue operation adds an element to the end of the queue:
- Create a new node (temp) with the given value.
- If rear is null, it means the queue is empty. Set both front and rear to temp and return.
- If the queue is not empty, set the next pointer of the current rear to temp and update rear to temp.
- Step 4: Define the Dequeue Operation
- The dequeue operation removes an element from the front of the queue:
- If front is null, it means the queue is empty, and there is nothing to dequeue.
- Otherwise, create a temporary node (temp) pointing to front.
- Update front to point to the next node.
- If front becomes null, set rear to null as well.
- Finally, delete the temporary node to free up memory.
- Example:

QNode Class:

```
class QNode {
   int key;
   QNode next;
   public QNode(int key)
   {
      this.key = key;
      this.next = null;
   }
}
```

Queue Class:

```
class Queue {
    QNode front, rear;

    public Queue() { this.front = this.rear = null; }

    // Method to add an key to the queue.
    void enqueue(int key)
    {

        // Create a new LL node
        QNode temp = new QNode(key);

        // If queue is empty, then new node is front and
        // rear both
        if (this.rear == null) {
            this.front = this.rear = temp;
            return;
        }

        // Add the new node at the end of queue and change
        // rear
        this.rear.next = temp;
        this.rear = temp;
    }

    // Method to remove an key from queue.
    void dequeue()
    {
        // If queue is empty, return NULL.
        if (this.front == null)
            return;

        // Store previous front and move front one node
        // ahead
        QNode temp = this.front;
        this.front = this.front.next;

        if (this.front == null)
            this.rear = null;
    }
}
```

Main Class:

```
public class Main {
    public static void main(String[] args) {
        Queue q = new Queue();
        q.enqueue(10);
        q.enqueue(20);
        q.dequeue();
        q.dequeue();
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);
        q.dequeue();
        System.out.println("Queue Front : " + ((q.front != null) ? (q.front).key : -1));
        System.out.println("Queue Rear : " + ((q.rear != null) ? (q.rear).key : -1));
    }
}
```

- The methods in the above code work as follows:
- enQueue(): The enQueue operation adds a new element to the rear of the queue. A new node is created and linked to the current rear node. The rear pointer is then updated to the new node. If the queue was initially empty, both front and rear pointers are set to the new node.
- deQueue(): The deQueue operation removes an element from the front of the queue. The front pointer is moved to the next node, and the value of the removed node is processed. If the queue becomes empty after this operation, both the front and rear pointers are set to null.

4.5 How does Queue work?

- Above, we learned that queues include operations such as Enqueue, Dequeue and a number of other operations. So how do they work? The code snippets below will demonstrate in detail how they work.
- Example:

Queue Class:

```
class Queue {
    int front, rear, size;
    int capacity;
    int[] array;

    public Queue(int capacity)
    {
        this.capacity = capacity;
        front = this.size = 0;
        rear = capacity - 1;
        array = new int[this.capacity];
    }
```

```
// Queue is full when size becomes
// equal to the capacity
boolean isFull(Queue queue)
{
    return (queue.size == queue.capacity);
}

// Queue is empty when size is 0
boolean isEmpty(Queue queue)
{
    return (queue.size == 0);
}

// Method to add an item to the queue.
// It changes rear and size
void enqueue(int item)
{
    if (isFull(this))
        return;
    this.rear = (this.rear + 1) % this.capacity;
    this.array[this.rear] = item;
    this.size = this.size + 1;
    System.out.println(item + " enqueued to queue");
}

// Method to remove an item from queue.
// It changes front and size
int dequeue()
{
    if (isEmpty(this))
        return Integer.MIN_VALUE;

    int item = this.array[this.front];
    this.front = (this.front + 1) % this.capacity;
    this.size = this.size - 1;
    return item;
}

// Method to get front of queue
int front()
{
    if (isEmpty(this))
        return Integer.MIN_VALUE;
        return  this.array[this.front];
    }

    // Method to get rear of queue
    int rear()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        return this.array[this.rear];
    }
}
```

Main Class:

```
public class Main {
    public static void main(String[] args)
    {
        Queue queue = new Queue(1000);

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.enqueue(40);

        System.out.println(queue.dequeue()
                + " dequeued from queue");

        System.out.println("Front item is "
                + queue.front());

        System.out.println("Rear item is " + queue.rear());
    }
}
```

- Operations are simulated by methods in the Queue class, specifically as follows:
- Enqueue:
- Enqueue() operation in Queue adds (or stores) an element to the end of the queue.
- The following steps should be taken to enqueue (insert) data into a queue:
- Step 1: Check if the queue is full.
- Step 2: If the queue is full, return overflow error and exit.
- Step 3: If the queue is not full, increment the rear pointer to point to the next empty space.
- Step 4: Add the data element to the queue location, where the rear is pointing.
- Step 5: return success.

- Dequeue:
- removes (or access) the first element from the queue.
- The following steps are taken to perform the dequeue operation:
- Step 1: Check if the queue is empty.
- Step 2: If the queue is empty, return the underflow error and exit.
- Step 3: If the queue is not empty, access the data where the front is pointing.
- Step 4: Increment the front pointer to point to the next available data element.
- Step 5: The Return success.
- Front: This operation returns the element at the front end without removing it.
- Rear: This operation returns the element at the rear end without removing it.
- isEmpty: This operation returns a boolean value that indicates whether the queue is empty or not.
- isFull: This operation returns a boolean value that indicates whether the queue is full or not.

M2 Compare the performance of two sorting algorithms.

5. Sort Algorithms

5.1 Selection Sort

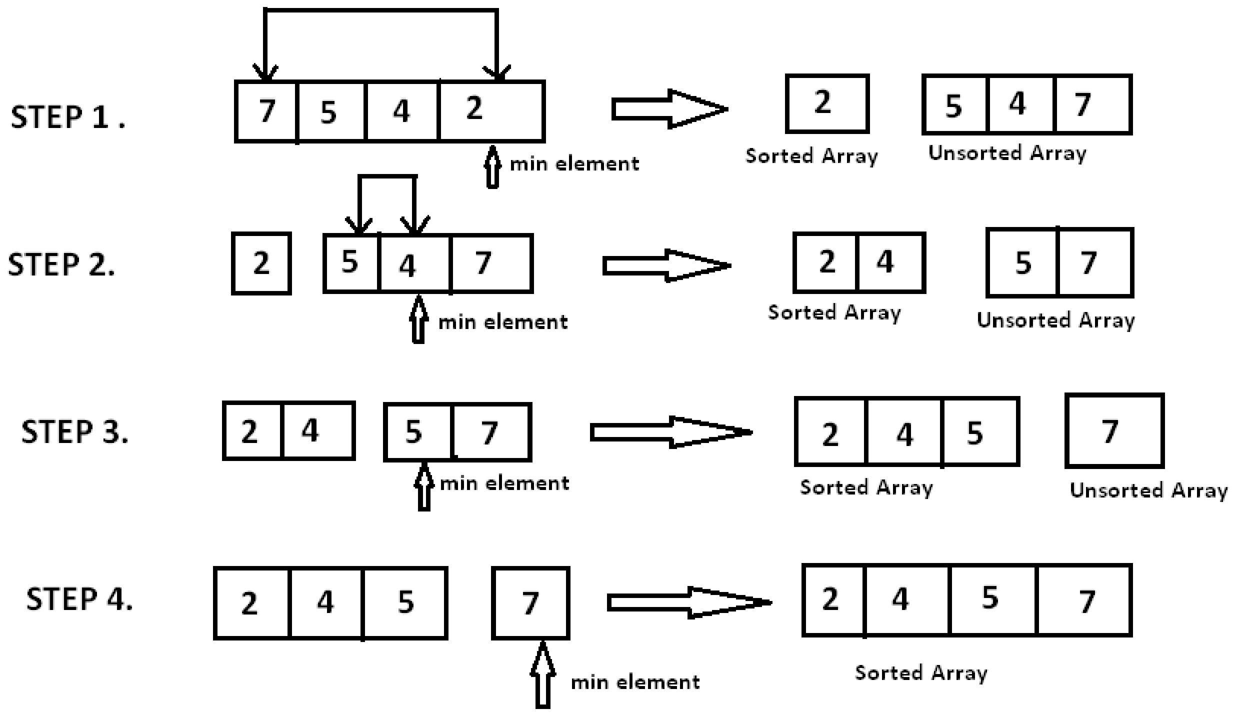


Figure 17: Selection Sort

5.1.1 Definition

- Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

5.1.2 Time Complexity

- The time complexity of the Selection sort remains constant regardless of the input array's initial order. At each step, the algorithm identifies the minimum element and places it in its correct position. However, the minimum element cannot be determined until the entire array is traversed.
- Best-case: O(n2), best case occurs when the array is already sorted. (where n is the number of integers in an array)
- Average-case: O(n2), the average case arises when the elements of the array are in a disordered or random order, without a clear ascending or descending pattern.
- Worst-case: O(n2), The worst-case scenario arises when we need to sort an array in ascending order, but the array is initially in descending order.

5.1.3 Space Complexity

- The space complexity of Selection Sort is O(1).
- This is because we use only constant extra space such as:
- 2 variables to enable swapping of elements.
- One variable to keep track of smallest element in unsorted array.
- Hence, in terms of Space Complexity, Selection Sort is optimal as the memory requirements remain same for every input.

5.1.4 Stability

- Selection sort works by finding the minimum element and then inserting it in its correct position by swapping with the element which is in the position of this minimum element. This is what makes it unstable.
- Swapping might impact in pushing a key(let's say A) to a position greater than the key(let's say B) which are equal keys. which makes them out of desired order.
- Selection sort can be made Stable if instead of swapping, the minimum element is placed in its position without swapping i.e. by placing the number in its position by pushing every element one step forward(shift all elements to left by 1).

5.2 Quick Sort

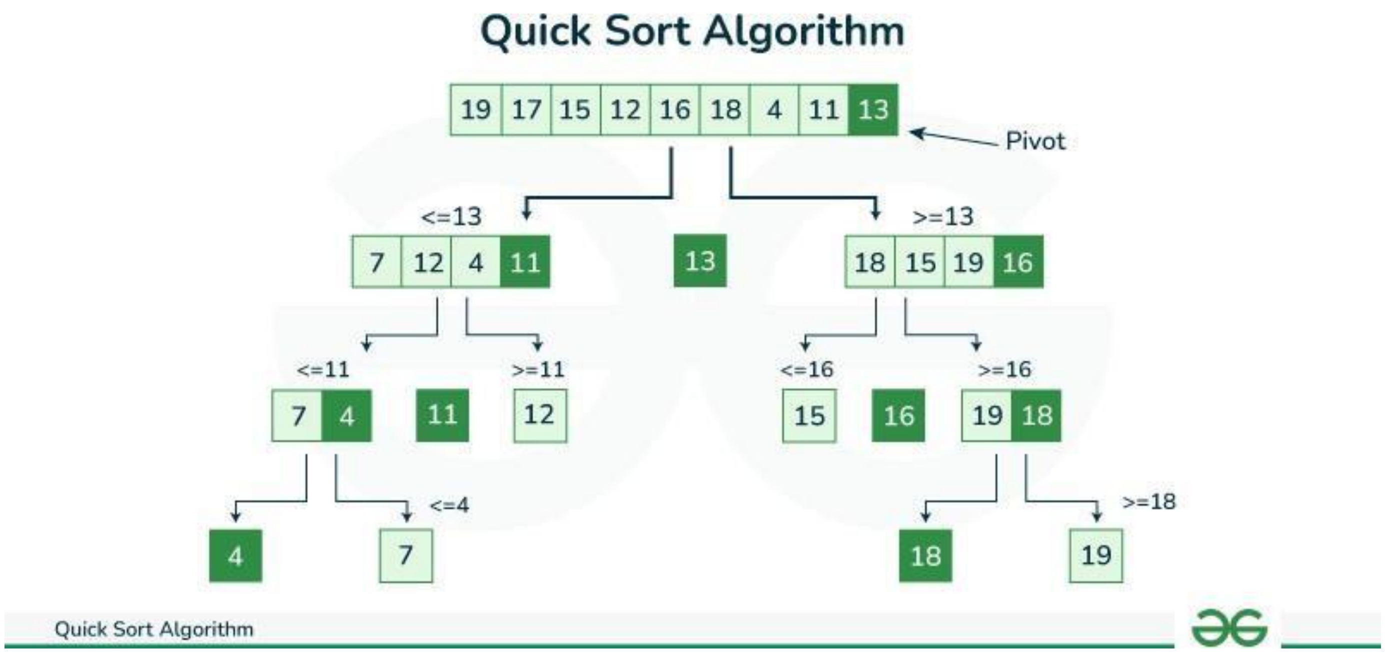


Figure 18: Quick Sort

5.2.1 Definition

- Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

- The key process in Quick Sort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
- Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

5.2.2 Time Complexity

- Best-case:
- The best case of quick sort is when we will select pivot as a mean element. Therefore, the time complexity is O(N * logN).
- T(N) = 2 * T(N / 2) + N * constant
- Now T(N/2) is also 2*T(N / 4) + N / 2 * constant. So,
- T(N) = 2*(2*T(N / 4) + N / 2 * constant) + N * constant
  = 4 * T(N / 4) + 2 * constant * N.
- So, we can say that
- T(N) = 2k * T(N / 2k) + k * constant * N
- then, 2k = N
- k = log2N
- So T(N) = N * T(1) + N * log2N. Therefore, the time complexity is O(N * logN).
- Average-case: The worst case will occur when the array gets divided into two parts, one part consisting of N-1 elements and the other and so on. So:
- T(N) = T(N – k) + T(k)
- 1 / N * [$\sum_{i=1}^{N-1} T(i) + \sum_{i=1}^{N-1} T(N-i)$]
- As $\sum_{i=1}^{N-1} T(i)$ and $\sum_{i=1}^{N-1} T(N-i)$ are equal likely functions, we can say

  T(N) = 2/N * [$\sum_{i=1}^{N-1}$ T(i)]

  N * T(N) = 2 * [$\sum_{i=1}^{N-1} T(i)$]
- Also we can write:
- (N – 1) * T(N – 1) = 2 * [$\sum_{i=1}^{N-1}$ T(i)]
- If we subtract the above two equations, we get
- N * T(N) – (N – 1) * T(N – 1) = 2 * T(N – 1) + N2 * constant – (N – 1)2 * constant
- N * T(N) = T(N – 1) * (2 + N – 1) + constant + 2 * N * constant – constant
- = (N + 1) * T(N – 1) + 2 * N * constant
- Divide both side by N*(N-1) and we will get

- T(N) / (N + 1) = T(N – 1)/N + 2 * constant / (N + 1) — (i)
- If we put N = N-1 it becomes
- T(N – 1) / N = T(N – 2)/(N – 1) + 2*constant/N
- Therefore, the equation (i) can be written as
- T(N) / (N + 1) = T(N – 2)/(N – 1) + 2*constant/(N + 1) + 2*constant/N
- Similarly, we can get the value of T(N-2) by replacing N by (N-2) in the equation (i). At last it will be like
- T(N) / (N + 1) = T(1)/2 + 2*constant * [1/2 + 1/3 + . . . + 1/(N – 1) + 1/N + 1/(N + 1)]
- T(N) = 2 * constant * log2N * (N + 1)
- If we ignore the constant it becomes
- T(N) = log2N * (N + 1)
- So the time complexity is O(N * logN).
- Worst-case: The worst case will occur when the array gets divided into two parts, one part consisting of N-1 elements and the other and so on. So,
- T(N) = T(N – 1) + N * constant
- = T(N – 2) + (N – 1) * constant + N * constant = T(N – 2) + 2 * N * constant – constant
- = T(N – 3) + 3 * N * constant – 2 * constant – constant

  . . .
- = T(N – k) + k * N * constant – (k – 1) * constant – . . . – 2*constant – constant
- = T(N – k) + k * N * constant – constant * (k*(k – 1))/2
- If we put k = N in the above equation, then
- T(N) = T(0) + N * N * constant – constant * (N * (N-1)/2)
- = N2 – N*(N-1)/2
- = N2/2 + N/2
- So the worst case complexity is O(N2)

5.2.3 Space Complexity

- Worst-case scenario: O(n) due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size O(n).

- Best-case scenario: O(log n) as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size O(log n).

5.2.4 Stability

- QuickSort is an unstable algorithm because we do swapping of elements according to pivot's position (without considering their original positions).
- Quicksort can be stable but it typically isn't implemented that way. Making it stable either requires order N storage (as in a naive implementation) or a bit of extra logic for an in-place version.
- In the best and average cases, where the array is partitioned into two equal subarrays, the recursion stack in quicksort takes up to O(log n) space, as its height will be log n. At each recursion level, extra space of "n" is required for the smaller and greater arrays. Thus, the total space is the sum of the recursion stack space and the extra arrays space, which equals O(log n) + n * O(log n) = O(n log n). In these cases, the common unstable implementation of quicksort uses only O(log n) space for the recursion stack.
- In the worst case, where the array is partitioned into subarrays of sizes n-1 and 1, the recursion tree height becomes "n," and extra space is needed at each level. Consequently, the worst-case space complexity is O(n^2). For the common unstable implementation, the worst-case space complexity is O(n).

5.3 Performance between Selection Sort and Quick Sort

- Based on all the information we talked about above, we will have the following comparison table:

Criterial	Quick Sort	Selection Sort
Time Complexity	Best: O(n log n)	Best: O(n^2)
	Average: O(n log n)	Average: O(n^2)
	Worst: O(n^2)	Worst: O(n^2)
Space Complexity	Best and Average: O(log n)	Best and Average: O(1)
	Worst: O(n)	Worst: O(1)
Divide and Conquer	Yes	No
Stability	No (unstable)	No (unstable)

LO2 Specify abstract data types and algorithms in a formal notation

P3 Specify the abstract data type for a software stack using an imperative definition.

6. Abstract Data Type

6.1 Definition

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.

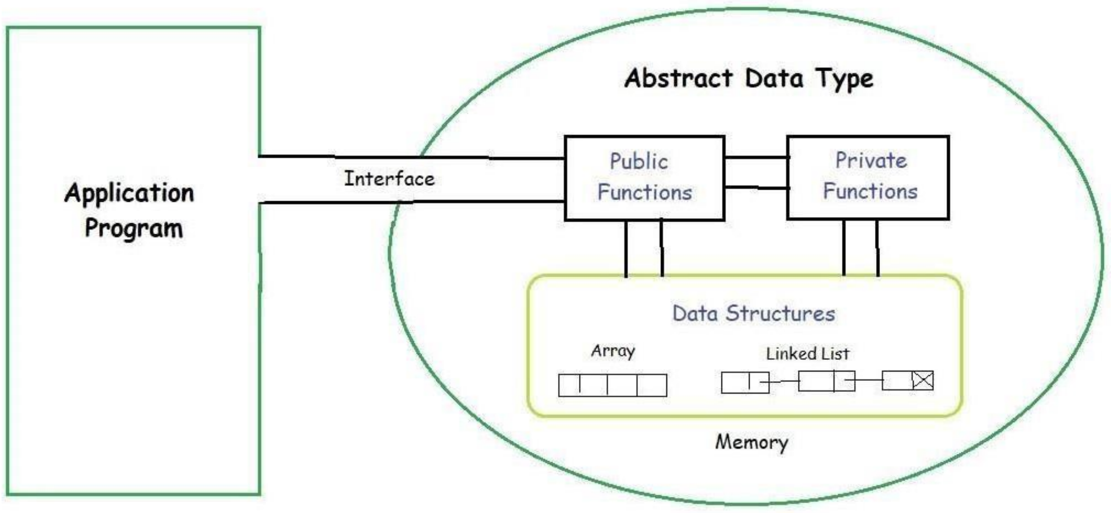


Figure 19: Abstract data type

6.2 Type of Abstract Data Type

6.2.1 List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.
- The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

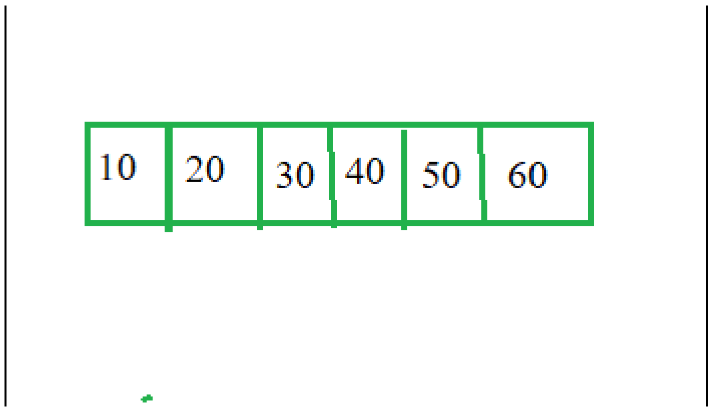


Figure 20: List ADT

- The List ADT Functions is given below:
  + get() – Return an element from the list at any given position.
  + insert() – Insert an element at any position of the list.
  + remove() – Remove the first occurrence of any element from a non-empty list.
  + removeAt() – Remove the element at a specified location from a non-empty list.
  + replace() – Replace an element at any position by another element.
  + size() – Return the number of elements in the list.

+ isEmpty() – Return true if the list is empty, otherwise return false.
+ isFull() – Return true if the list is full, otherwise return false.

6.2.2 Stack ADT

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and count of number of entries currently in stack.

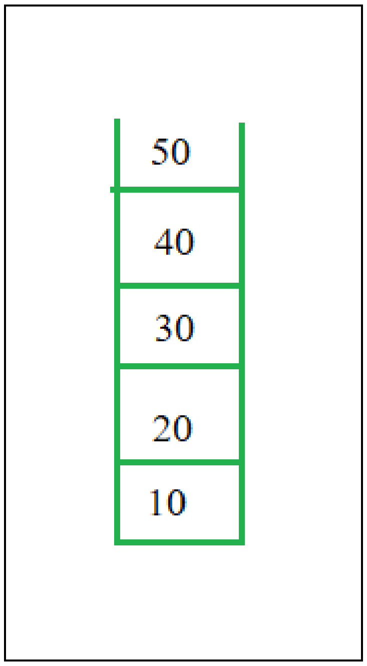


Figure 21: Stack ADT

- The Stack ADT Functions is given below:
  + push() – Insert an element at one end of the stack called top.
  + pop() – Remove and return the element at the top of the stack, if it is not empty.
  + peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
  + size() – Return the number of elements in the stack.
  + isEmpty() – Return true if the stack is empty, otherwise return false.
  + isFull() – Return true if the stack is full, otherwise return false.

6.2.3 Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

- Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

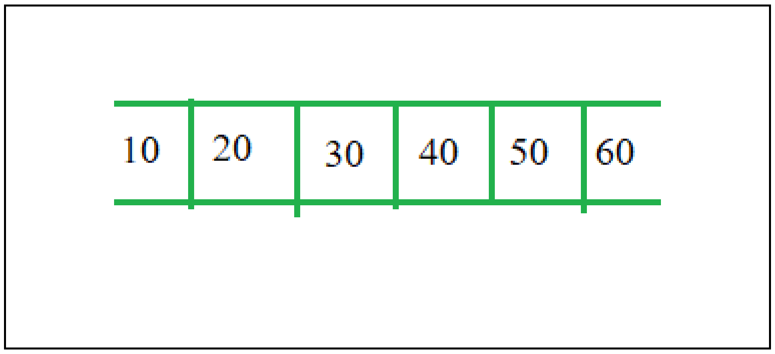


Figure 22: Queue ADT

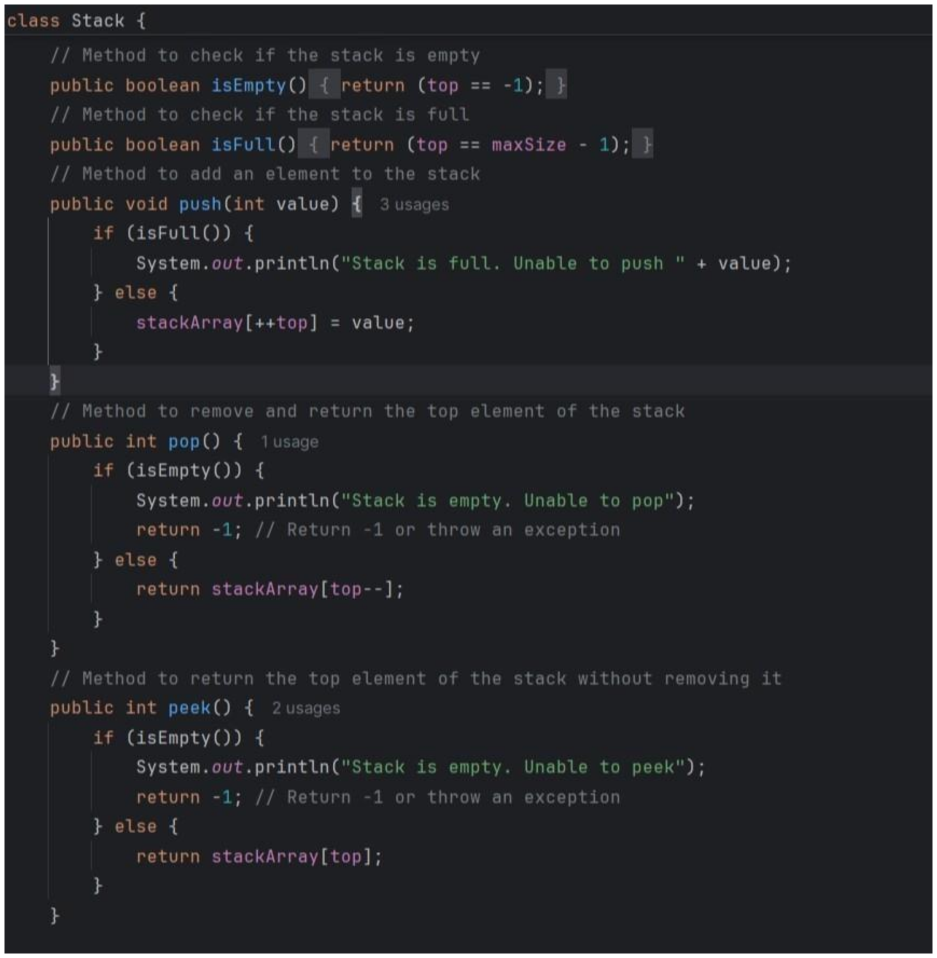
- The Queue ADT Functions is given below:
  + enqueue() – Insert an element at the end of the queue.
  + dequeue() – Remove and return the first element of the queue, if the queue is not empty.
  + peek() – Return the element of the queue without removing it, if the queue is not empty.
  + size() – Return the number of elements in the queue.
  + isEmpty() – Return true if the queue is empty, otherwise return false.
  + isFull() – Return true if the queue is full, otherwise return false.

7. Stack in abstract data type

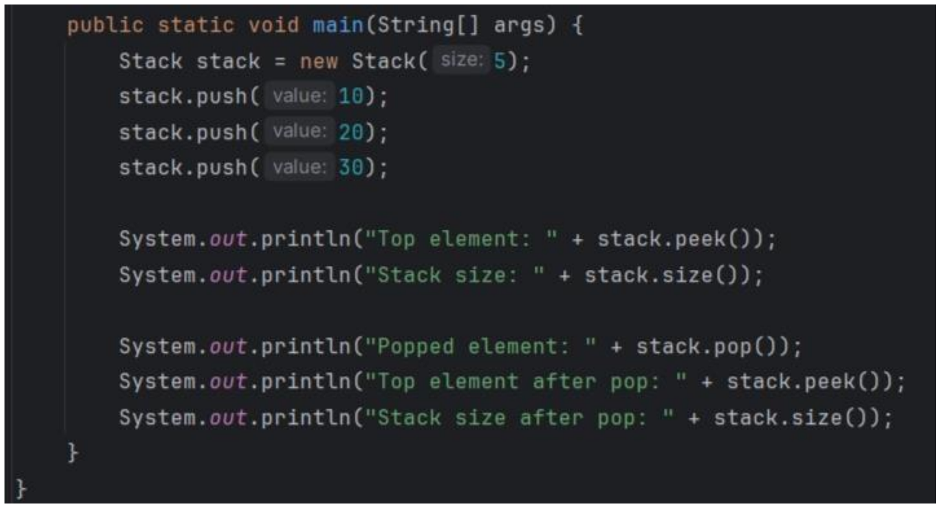
   7.1 Definition

   - A stack is an abstract data type (ADT) that follows the Last In, First Out (LIFO) principle, meaning that the last element added to the stack will be the first one to be removed. It is commonly used in various algorithms and applications, such as expression evaluation, backtracking, and function call management in programming languages.

7.2 Initialize the Stack



```
class Stack {
    // Method to check if the stack is empty
    public boolean isEmpty() { return (top == -1); }
    // Method to check if the stack is full
    public boolean isFull() { return (top == maxSize - 1); }
    // Method to add an element to the stack
    public void push(int value) {  3 usages
        if (isFull()) {
            System.out.println("Stack is full. Unable to push " + value);
        } else {
            stackArray[++top] = value;
        }
    }
    // Method to remove and return the top element of the stack
    public int pop() {  1 usage
        if (isEmpty()) {
            System.out.println("Stack is empty. Unable to pop");
            return -1; // Return -1 or throw an exception
        } else {
            return stackArray[top--];
        }
    }
    // Method to return the top element of the stack without removing it
    public int peek() {  2 usages
        if (isEmpty()) {
            System.out.println("Stack is empty. Unable to peek");
            return -1; // Return -1 or throw an exception
        } else {
            return stackArray[top];
        }
    }
```



```
    public static void main(String[] args) {
        Stack stack = new Stack( size: 5);
        stack.push( value: 10);
        stack.push( value: 20);
        stack.push( value: 30);

        System.out.println("Top element: " + stack.peek());
        System.out.println("Stack size: " + stack.size());

        System.out.println("Popped element: " + stack.pop());
        System.out.println("Top element after pop: " + stack.peek());
        System.out.println("Stack size after pop: " + stack.size());
    }
}
```

Figure 23: Initialize the Stack

- In the above code, Stack is initialized and the specific operation of each function is explained as follows:
    + isEmpty: Checks if the stack is empty by comparing the top index to -1.
    + isFull: Checks if the stack is full by comparing the top index to the maximum size minus one.
    + push: Adds an element to the top of the stack. If the stack is full, it prints a message and does not add the element.
    + pop: Removes and returns the top element of the stack. If the stack is empty, it prints a message and returns -1 (or you could throw an exception).
    + peek: Returns the top element of the stack without removing it. If the stack is empty, it prints a message and returns -1 (or you could throw an exception).
    + size: Returns the number of elements in the stack.
    + main: Demonstrates the usage of the stack by performing push, pop, peek, and size operations.

7.3 Operation

7.3.1 Push

- Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.
- Algorithm for Push Operation:
    + Before pushing the element to the stack, we check if the stack is full .
    + If the stack is full (top == capacity-1) , then Stack Overflows and we cannot insert the element to the stack.
    + Otherwise, we increment the value of top by 1 (top = top + 1) and the new value is inserted at top position.
    + The elements can be pushed into the stack till we reach the capacity of the stack.

7.3.2 Pop

- Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Algorithm for Pop Operation:
    + Before popping the element from the stack, we check if the stack is empty.
    + If the stack is empty (top == -1), then Stack Underflows and we cannot remove any element from the stack.
    + Otherwise, we store the value at top, decrement the value of top by 1 (top = top – 1) and return the stored top value.

7.3.3 Peek

- Returns the top element of the stack.
- Algorithm for Top Operation:
    + Before returning the top element from the stack, we check if the stack is empty.
    + If the stack is empty (top == -1), we simply print “Stack is empty”.
    + Otherwise, we return the element stored at index = top.

7.3.4 IsEmpty

- Returns true if the stack is empty, else false.
- Algorithm for isEmpty Operation:
    + Check for the value of top in stack.
    + If (top == -1), then the stack is empty so return true.
    + Otherwise, the stack is not empty so return false.

LO3: Implement complex data structures and algorithms

P4: Implement a complex ADTand algorithm in an executable programming language to solve a well-defined problem.

1. An overview of previous developments.

   1.1 Identify problems that need to be solved

   After our previous implementation, we realized that our program encountered a number of issues that needed to be improved. We have raised some of the following issues:

   Data Entry and Management: Allow users to input, edit, and delete student information dynamically, including student ID, name, and marks.

   Student Ranking Calculation: Calculate and assign a ranking to each student based on predefined mark intervals whenever marks are added or edited.

   Data Storage and Retrieval: Use an appropriate data structure to store student records and implement methods for efficient retrieval based on different criteria.

   Sorting: Implement a stable, efficient sorting algorithm (e.g., MergeSort) to sort student records in descending order of marks.

   Searching: Implement an efficient search algorithm to quickly retrieve student records using their ID.

   User Interface: Design an intuitive command-line or graphical user interface that provides clear prompts and feedback for all operations and handles errors gracefully.

   Data Validation: Implement validation checks to ensure unique student IDs, non-empty names, and marks within the range of 0.0 to 10.0, handling invalid inputs appropriately.

   Performance Optimization: Evaluate and implement efficient algorithms for sorting and searching, ensuring the system handles large datasets without significant performance issues.

   Evaluation of Algorithms: Compare the time and space complexity of previous and proposed algorithms (e.g., Bubble Sort vs. MergeSort) with empirical data supporting the analysis.

1.2 Programming language

We use the Java language to execute the project. Java is an object-oriented programming language that is quite popular in the world. The use of Java for development will also become more flexible and Java also supports a lot of libraries and can be developed across platforms perfectly. Linked list elements are not stored at the contiguous location, the elements are linked using pointers as shown below.

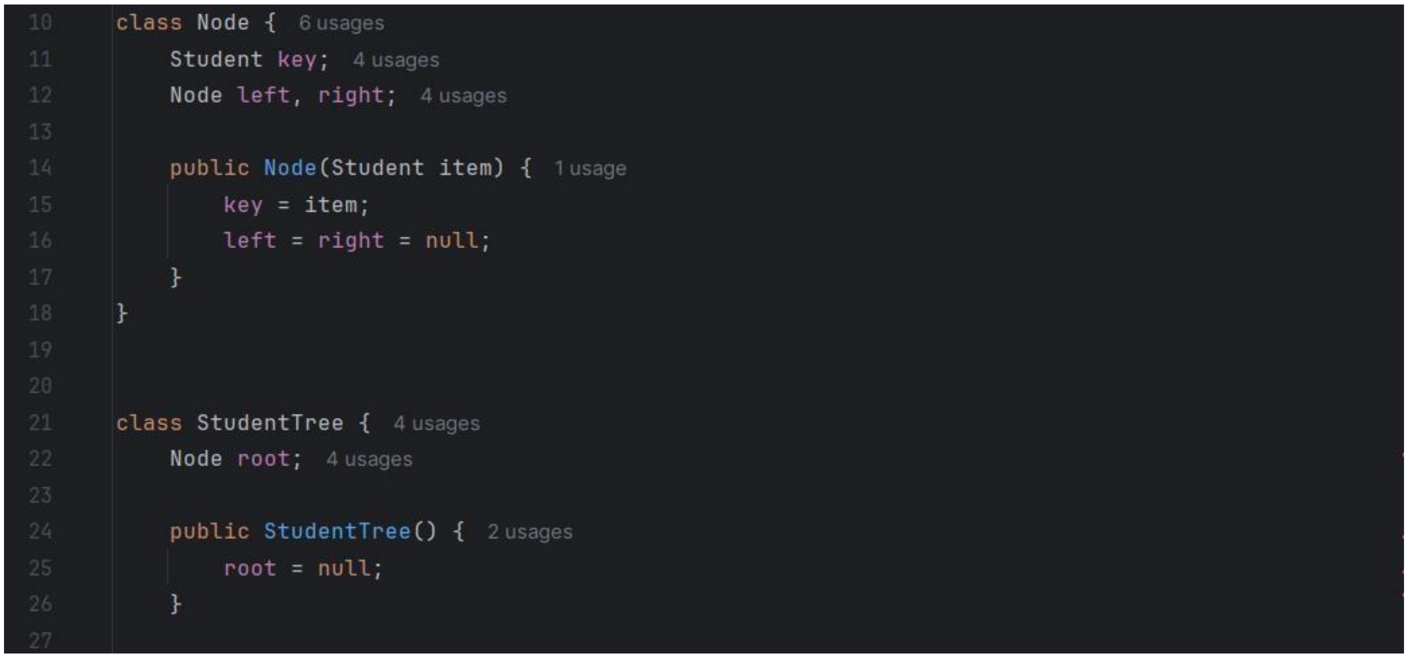
1.3 Abstract Data Structure

1.3.1 Tree

Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

We'll implement this structure in the app as follows:



```
class Node {  6 usages
    Student key;  4 usages
    Node left, right;  4 usages

    public Node(Student item) {  1 usage
        key = item;
        left = right = null;
    }
}


class StudentTree {  4 usages
    Node root;  4 usages

    public StudentTree() {  2 usages
        root = null;
    }

```

Figure 24: Tree in Student management app

- The **Node** class represents a node in a binary tree, but instead of storing an int value, it stores a **Student** object.
- Student key: This field is used to store a **Student** object. This object might contain information such as a student's name, ID, grades, etc.

- Node left, right: These are two references to the left and right child nodes of the current node in the binary tree.
- public **Node**(Student item): This is a constructor for the Node class. When you create a Node object, you need to pass in a **Student** object. This object is assigned to key. The left (left) and right (right) child nodes are initialized to null, indicating that the current node has no children yet.
- The **StudentTree** class represents a binary tree that contains Node objects.
- Node root: This is a reference to the root node of the tree. The root node is the first node in the binary tree, from which other nodes branch out.
- public **StudentTree**(): This is a constructor for the **StudentTree** class. When a **StudentTree** object is created, the root node (root) is initialized to null, indicating that the tree is initially empty with no nodes.

Here's how classes work:

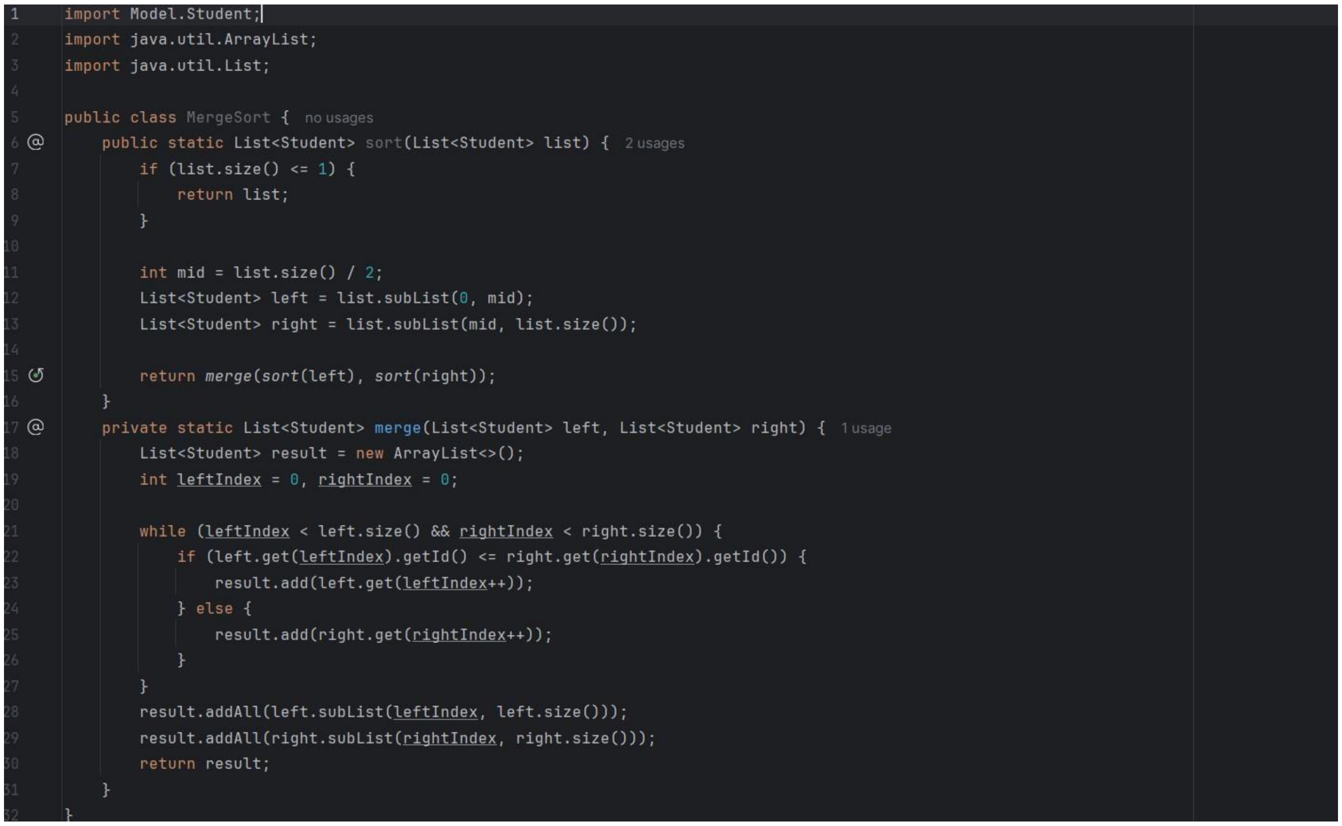
- **Node**: Each Node in the binary tree contains a **Student** object and can have two child nodes (left and right). The **Student** object can hold various student-related information.
- **StudentTree**: The **StudentTree** class is the structure that manages the binary tree, starting from the root node (root). Initially, when the tree is created, it will have no nodes.

1.4 Algorithm

1.4.1 Sort (Quick sort and Merge sort)

In the application, I will implement improved sorting algorithms from the previous one: Merge sort and Quick Sort. These two algorithms are evaluated more optimally than some other types of sorting algorithms.

- Merge Sort:



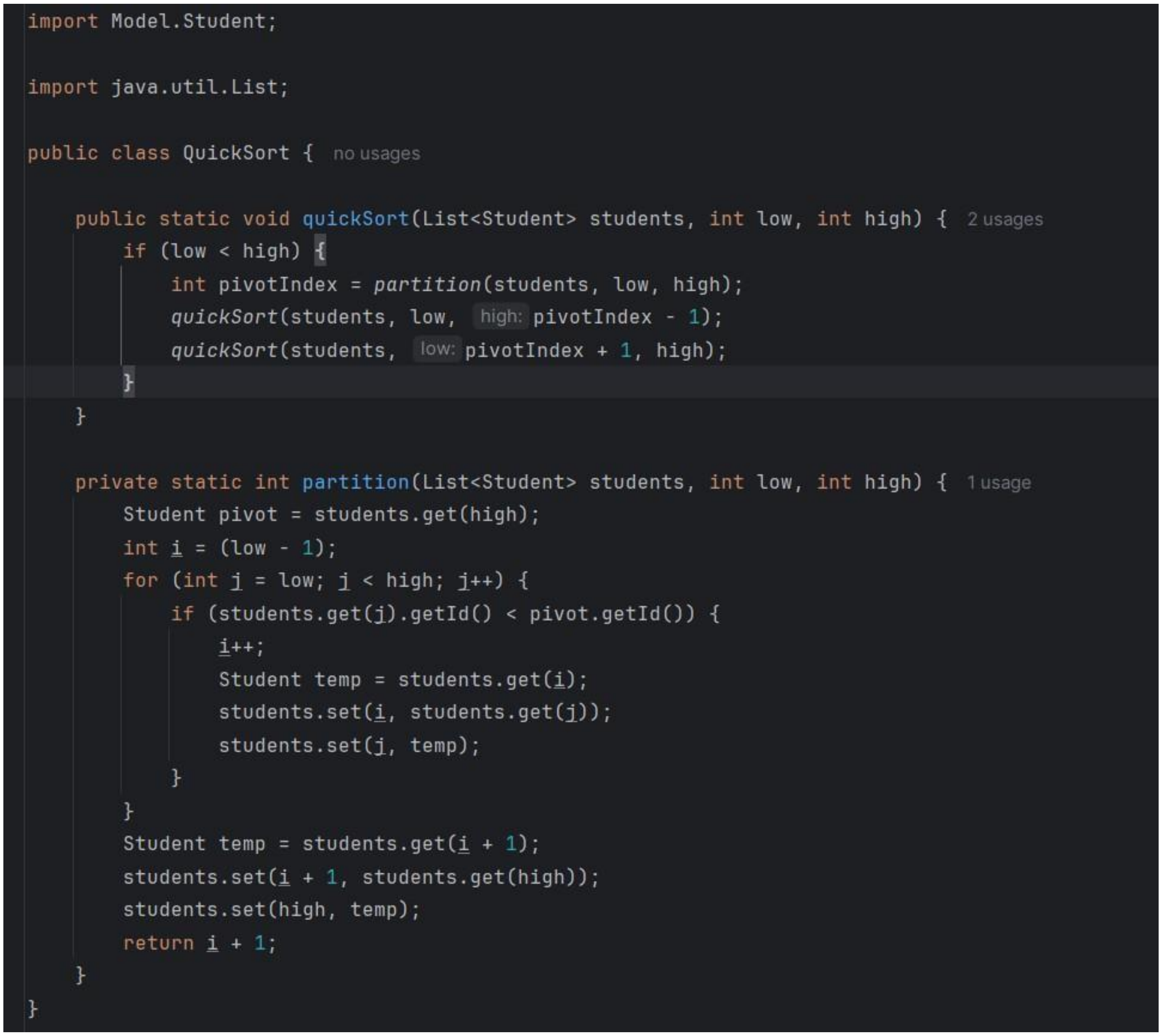
```java
import Model.Student;
import java.util.ArrayList;
import java.util.List;

public class MergeSort {  no usages
    public static List<Student> sort(List<Student> list) {  2 usages
        if (list.size() <= 1) {
            return list;
        }

        int mid = list.size() / 2;
        List<Student> left = list.subList(0, mid);
        List<Student> right = list.subList(mid, list.size());

        return merge(sort(left), sort(right));
    }
    private static List<Student> merge(List<Student> left, List<Student> right) {  1 usage
        List<Student> result = new ArrayList<>();
        int leftIndex = 0, rightIndex = 0;

        while (leftIndex < left.size() && rightIndex < right.size()) {
            if (left.get(leftIndex).getId() <= right.get(rightIndex).getId()) {
                result.add(left.get(leftIndex++));
            } else {
                result.add(right.get(rightIndex++));
            }
        }
        result.addAll(left.subList(leftIndex, left.size()));
        result.addAll(right.subList(rightIndex, right.size()));
        return result;
    }
}
```

Figure 25: Merge sort

Here's how the above snippet works:

- The list is split into two halves. left contains the first half of the list, and right contains the second half. This division is done using the subList() method.
- The method recursively sorts both halves (left and right) and then merges them into a single sorted list using the merge() method.
- A new list result is created to hold the merged and sorted elements. Two indices, leftIndex and rightIndex, are used to track positions in the left and right lists, respectively.
- The while loop iterates as long as there are elements in both left and right lists.
- It compares the id of the current elements from left and right.
- The smaller element (or equal element) is added to the result list, and the corresponding index (leftIndex or rightIndex) is incremented.
- After the while loop, there may still be elements left in either left or right.
- These remaining elements are added to the result list using the addAll() method.
- subList(leftIndex, left.size()) and subList(rightIndex, right.size()) get the remaining elements in the left and right lists, respectively.
- Finally, the result list, which now contains all elements sorted, is returned.

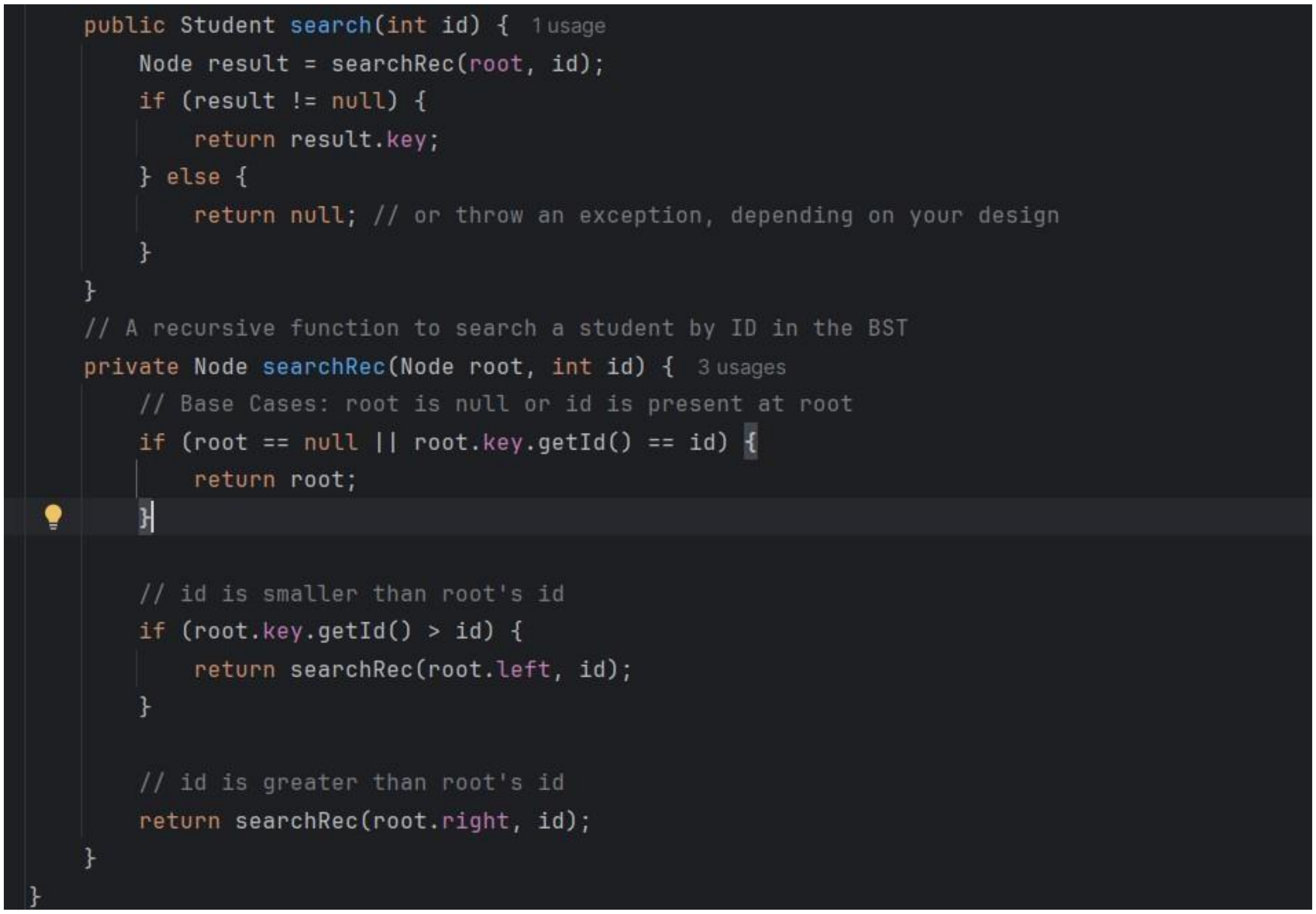
Quick Sort:



```java
import Model.Student;

import java.util.List;

public class QuickSort {  no usages

    public static void quickSort(List<Student> students, int low, int high) {  2 usages
        if (low < high) {
            int pivotIndex = partition(students, low, high);
            quickSort(students, low,  high: pivotIndex - 1);
            quickSort(students,  low: pivotIndex + 1, high);
        }
    }

    private static int partition(List<Student> students, int low, int high) {  1 usage
        Student pivot = students.get(high);
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (students.get(j).getId() < pivot.getId()) {
                i++;
                Student temp = students.get(i);
                students.set(i, students.get(j));
                students.set(j, temp);
            }
        }
        Student temp = students.get(i + 1);
        students.set(i + 1, students.get(high));
        students.set(high, temp);
        return i + 1;
    }
}
```

Figure 26: Quick Sort

1.4.2 Search (Binary Search Tree)



```java
    public Student search(int id) {  1 usage
        Node result = searchRec(root, id);
        if (result != null) {
            return result.key;
        } else {
            return null; // or throw an exception, depending on your design
        }
    }
    // A recursive function to search a student by ID in the BST
    private Node searchRec(Node root, int id) {  3 usages
        // Base Cases: root is null or id is present at root
        if (root == null || root.key.getId() == id) {
            return root;
        }

        // id is smaller than root's id
        if (root.key.getId() > id) {
            return searchRec(root.left, id);
        }

        // id is greater than root's id
        return searchRec(root.right, id);
    }
}
```

Figure 27: Binary Search Tree

Students is the list of Student objects to be sorted. low is the starting index of the sub-array to be sorted. high is the ending index of the sub-array to be sorted.

Checks if the low index is less than the high index (i.e., there are at least two elements to sort). Calls the partition method to find the index where the pivot is placed correctly.
Recursively applies quickSort to the sub-arrays on the left and right of the pivot.

Students is the list of Student objects. low is the starting index of the sub-array. High is the ending index of the sub-array (also the index of the pivot).

The pivot is chosen as the element at the high index. ‘i’ is initialized to one less than low and is used to track the boundary of elements less than the pivot. The for loop iterates from low to high - 1:

- If the current element's ID is less than the pivot's ID, ‘i’ is incremented, and the current element is swapped with the element at index ‘i’.

After the loop, the pivot is placed in its correct position by swapping it with the element at ‘i’ + 1. Finaly, Returns the index of the pivot, which is now correctly positioned.

P5: Implement error handling andreporttestresults.

1. Error

1.1 Error determination

During the development process, when the implementation is not tight, it can cause some potential errors when developing the system. Specifically, as follows:

Null Pointer Dereference: Accessing or modifying a null pointer can cause crashes or undefined behavior.

Memory Leaks: Failing to properly manage memory allocation and deallocation, especially during node creation and deletion, can lead to memory leaks.

Data Inconsistency: Incorrect manipulation of the list can result in data inconsistency, such as duplicate nodes or lost nodes.

Boundary Conditions: Errors handling edge cases like empty lists, single-node lists, or lists with duplicate values can cause unexpected behavior or crashes.

1.2 Error handling method

To effectively overcome the above potential errors, we have come up with the following error handling strategies:

Thorough Testing: Write comprehensive test cases to cover various scenarios, including edge cases.

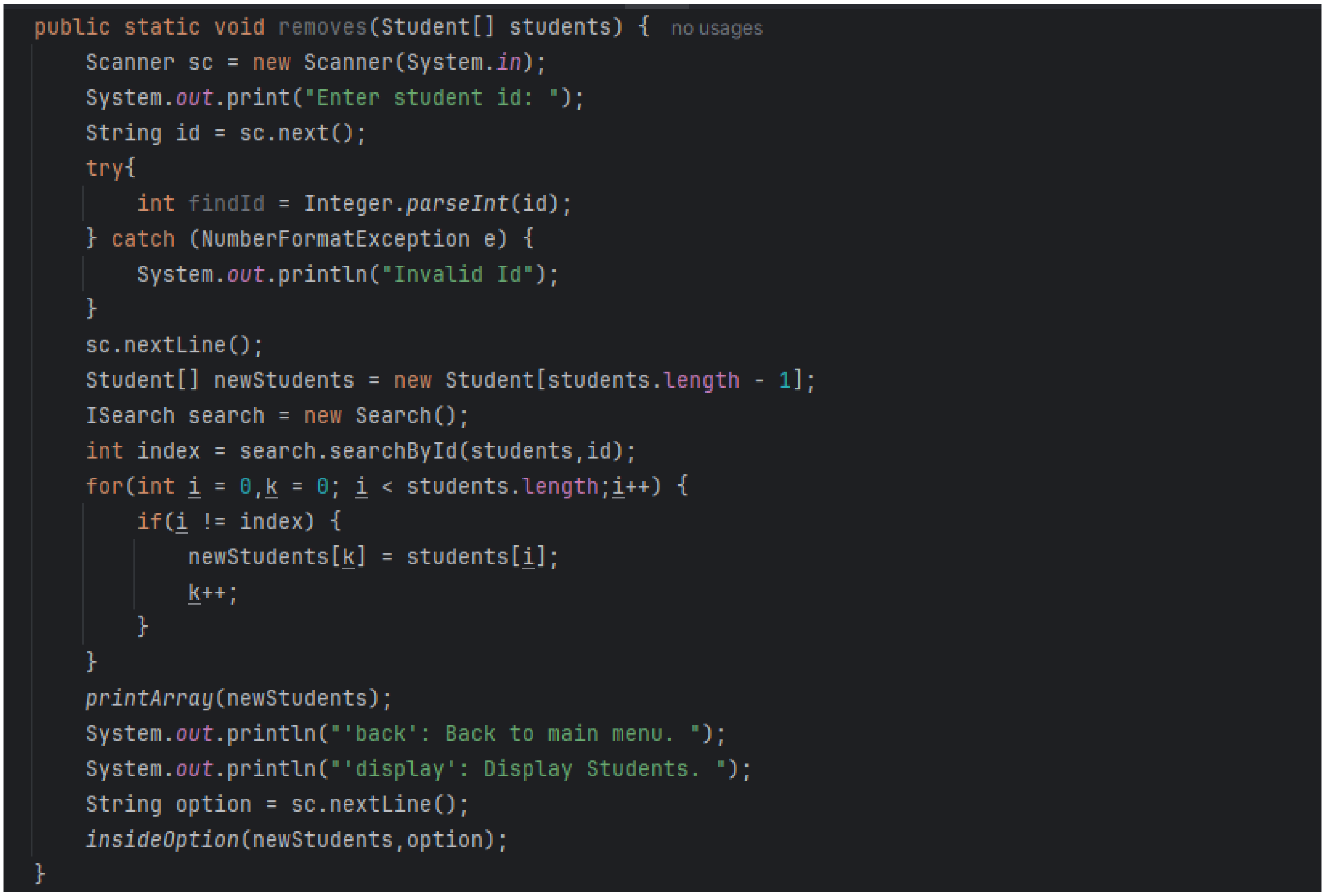
Debugging and Logging: Use debugging tools and logging to trace and identify issues during list manipulation.

Code Review: Conduct code reviews to catch logical errors and improve code quality.

Documentation: Maintain clear documentation of the algorithm implementations and list manipulation processes to ensure understanding and correct usage.

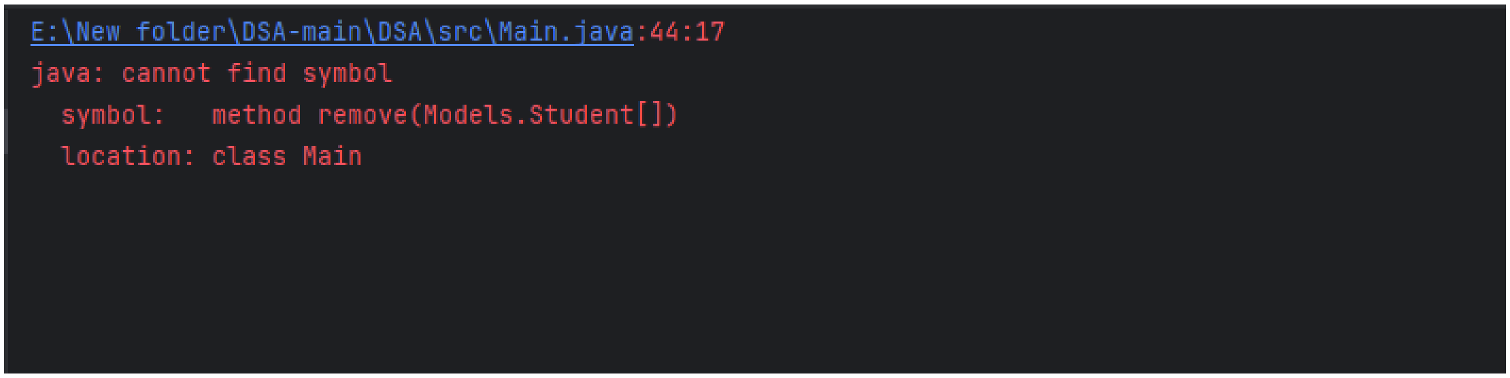
Implement error handling code

Here’s an example of having error and how I handle it:



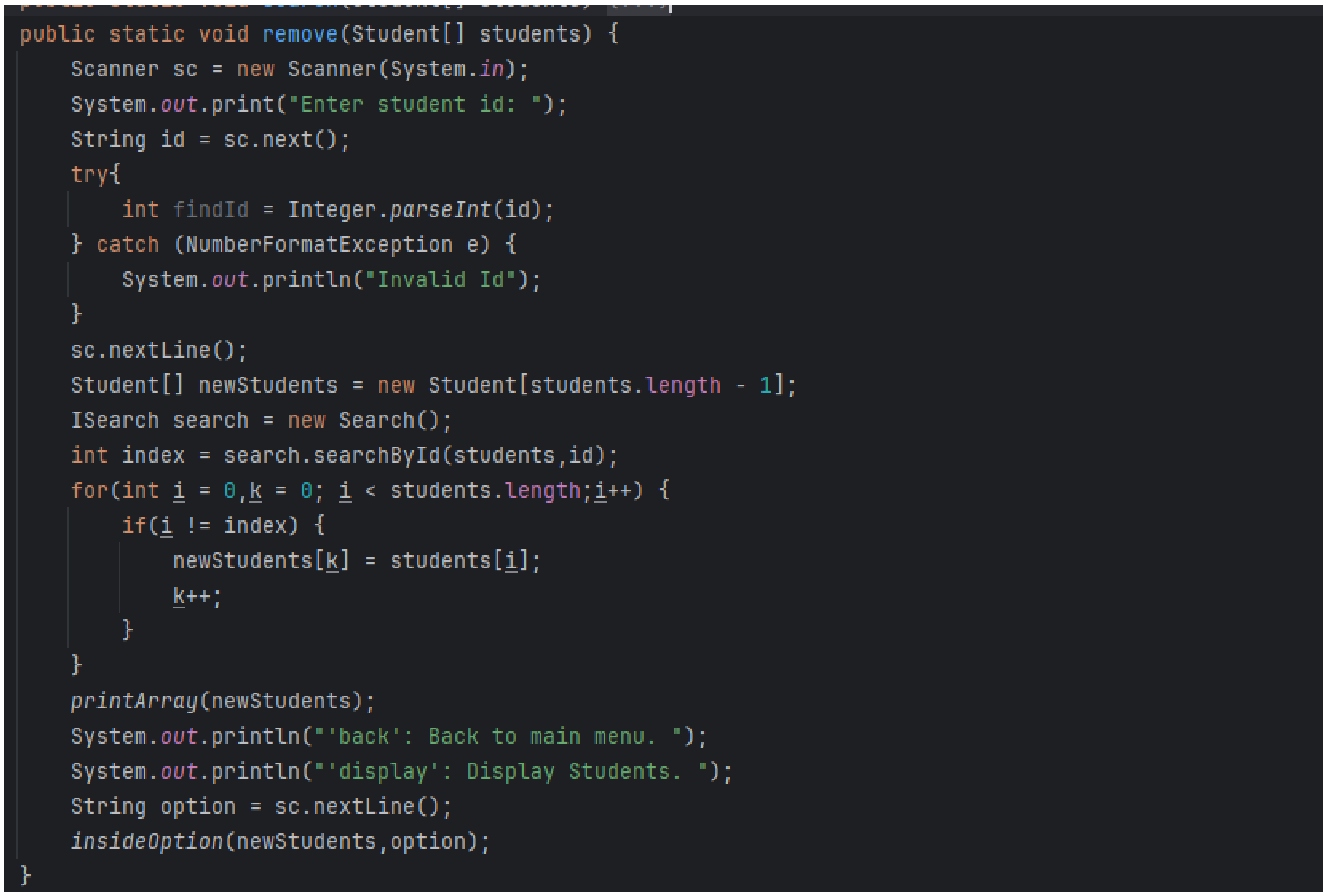
```java
public static void removes(Student[] students) {  no usages
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter student id: ");
    String id = sc.next();
    try{
        int findId = Integer.parseInt(id);
    } catch (NumberFormatException e) {
        System.out.println("Invalid Id");
    }
    sc.nextLine();
    Student[] newStudents = new Student[students.length - 1];
    ISearch search = new Search();
    int index = search.searchById(students,id);
    for(int i = 0,k = 0; i < students.length;i++) {
        if(i != index) {
            newStudents[k] = students[i];
            k++;
        }
    }
    printArray(newStudents);
    System.out.println("'back': Back to main menu. ");
    System.out.println("'display': Display Students. ");
    String option = sc.nextLine();
    insideOption(newStudents,option);
}
```

As you can see, the value that I wrote here was removes. This will cause an error like this:



```
E:\New folder\DSA-main\DSA\src\Main.java:44:17
java: cannot find symbol
  symbol:   method remove(Models.Student[])
  location: class Main
```

And how do I handle it? Just by refactor the name:



```
public static void remove(Student[] students) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter student id: ");
    String id = sc.next();
    try{
        int findId = Integer.parseInt(id);
    } catch (NumberFormatException e) {
        System.out.println("Invalid Id");
    }
    sc.nextLine();
    Student[] newStudents = new Student[students.length - 1];
    ISearch search = new Search();
    int index = search.searchById(students,id);
    for(int i = 0,k = 0; i < students.length;i++) {
        if(i != index) {
            newStudents[k] = students[i];
            k++;
        }
    }
    printArray(newStudents);
    System.out.println("'back': Back to main menu. ");
    System.out.println("'display': Display Students. ");
    String option = sc.nextLine();
    insideOption(newStudents,option);
}
```

LO4: Assess the effectiveness of data structure sand algorithms

P6: Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm.

Definition of asymptotic analysis

Asymptotic analysis is a method used in computer science and mathematics to assess algorithm efficiency and function behavior as the input size approaches infinity. It focuses on understanding how an algorithm's or a function's growth acts for huge input values rather than the technical specifics for specific input quantities.

Asymptotic analysis' major purpose is to classify algorithms based on their efficiency and provide a high level understanding of their performance characteristics. Typically, the study takes into account the top and lower bounds of the algorithm's time complexity or space complexity.

Asymptotic analysis is especially useful for comparing and contrasting algorithms since it provides a high level knowledge of their efficiency without delving into specific hardware, programming languages, or constant variables. It aids in making informed algorithm selection selections based on predicted performance for big input sizes.

Time Complexity Analysis.

Inserting a Student: If we insert a Student into an ArrayList, the average time complexity is O(1) (amortized).

Editing a Student: To change a student's information, you must first locate the student in the data structure and update their information. O(1) is the average time complexity for searching and updating student information.

Deleting a Student: Deleting a student entails locating the student in the data structure and updating their information. O(1) is the average time complexity for searching and updating student information.

Sorting Students: Depending on the sorting method employed, sorting students based on a certain criterion can have varying time complexity. In general, sorting algorithms have an average temporal complexity of O(n log n).

Searching for Students: Finding a student based on their ID or other criteria frequently necessitates iterating through student lists or employing data structures. The temporal complexity of searching will be O(1) on average.

Growth Rate Comparison Efficiency and Scalability.

Constant Time (O(1)): o Algorithms with constant time complexity have a constant growth rate regardless of input size. This is the most efficient scenario because the execution time is independent of input size. It has a high degree of scalability.

Logarithmic Time (O(log n)): o Logarithmic time algorithms have a growth rate that increases logarithmically with input size. In each step, these algorithms partition the input space, resulting in efficient performance even for enormous inputs. They are highly efficient and widely regarded as scalable.

Linear Time (O(n)): o The growth rate of algorithms with linear time complexity is exactly proportional to the size of the input. The execution time grows linearly with the size of the input. Although linear-time algorithms are often efficient, they may not scale well for very large input sizes.

Time in Linear Form (O(n log n)): o The growth rate of algorithms with linearithmic time complexity is somewhat faster than that of linear time. They frequently appear in algorithms for sorting and combining, such as Quick Sort and Merge Sort. They are less effective than linear time algorithms, but they are nevertheless widely used in practice and offer good scalability.

O(n^2) Quadratic Time: o The growth rate of algorithms with quadratic time complexity rises quadratically with the size of the input. They frequently involve nested loops. When dealing with big inputs, quadratic time algorithms can become exceedingly inefficient and unscalable. They are less efficient than linear or linearithmic time algorithms.

O(2^n) or Exponential Time:

The growth rate of algorithms with exponential time complexity rises exponentially with the size of the input. They are not scalable for even moderately sized inputs and are incredibly wasteful. Exponential time algorithms ought to be avoided as they are typically unfeasible.

Identifying Dominant Operations Optimization Opportunities.

Time Complexity Analysis:

Examine how long each of the many algorithms or data structures in your program takes to execute. Because they are more likely to be dominant, identify procedures having a higher time complexity.

Benchmarking:

Produce test cases or benchmarks that closely mimic real-world circumstances. Calculate how long your program takes to execute and note which operations add the most to the total runtime.

Data Structures and Algorithms:

Assess your program's selection of data structures and algorithms. Examine if there are any more effective options available that can lower the dominating operations' time complexity.

Algorithmic Optimization: o Look closely at the main operations and algorithms that your code use.

Seek methods to improve them, such as implementing memoization or dynamic programming techniques, employing more effective sorting algorithms, or cutting down on pointless iterations.

I/O Optimization:
Make regular I/O operations, like file or network access, more efficient.
To cut down on waiting times, minimize pointless I/O calls, employ buffered I/O, or take a look at asynchronous I/O methods.

Limitations

Although optimization can significantly increase a program's performance, it's vital to understand its limitations. When optimizing code, keep these restrictions in mind:

Trade-offs: Trade-offs between several parameters, including time complexity, memory utilization, code readability, and maintainability, are frequently necessary during optimization. It's possible that improving one thing will hurt other things.

Algorithmic Complexity: Performance benefits can be achieved by optimizing individual procedures and code snippets, but some issues are inherently complex and cannot be fully solved.

External Factors: Outside of your control, a program's performance may be impacted by things like input data qualities, network latency, or hardware restrictions. The degree to which optimization efforts can raise overall performance may be constrained by several variables.

Maintenance Difficulties: Code that has been well optimized may be more complicated and demanding to maintain. Low-level optimizations and manual memory management are examples of optimization approaches that can increase the likelihood of errors being introduced or make the code more difficult for future developers to understand.

Development Time: Implementing, testing, and validating optimization frequently takes more time and effort. This may not be appropriate for every project or situation, and it may affect the total development schedule.

Practical Considerations.

Establish Specific Performance Goals and Metrics: Clearly state the performance objectives and metrics you hope to attain through optimization. This will serve as a guide for your optimization efforts and a standard by which to judge how well the optimizations are working.

Determine the Important Code Paths: Concentrate your optimization efforts on the crucial code routes that affect system performance the most. These code segments can be located with the use of benchmarking and profiling.

Put the user experience first: In the end, optimization should enhance the user experience. Think about the effects that the improvements will have on parameters like latency, reaction time, and UI responsiveness. Give top priority to optimizations that improve the user experience directly.

Test Extensively: Extensive testing is necessary to guarantee that optimizations do not bring up new problems or bugs. Create thorough test cases that address a variety of scenarios, edge cases, and input variants. Conduct thorough testing to confirm the streamlined code's accuracy and stability.

Measure Total Performance Impact: Evaluate the total impact of optimizations on the system as a whole, rather than only concentrating on optimizing specific activities. Examine the effects of optimizations on scalability, system resource utilization, and overall performance.

Think about Readability and Maintenance: Code should be optimized while being readable and manageable. Steer clear of overly complex code and smart optimizations, as these could impede future development and maintenance efforts.

To make optimizations easier to grasp for next developers, clearly comment on them and document them.

P7 Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example.

Time Complexity and example.

Definition:

The term "time complexity" refers to how long an algorithm takes to execute in relation to the length of the input. It calculates how long it takes to run each algorithmic code statement. It won't look at an algorithm's overall execution time. Instead, it will provide details regarding variations (increases or decreases) in algorithm execution time with the number of operations (increases or decreases). Indeed, as stated in the definition, the length of the input alone determines how long it takes.

Here are some different types of a time complexity:

Constant Time (O(1)):

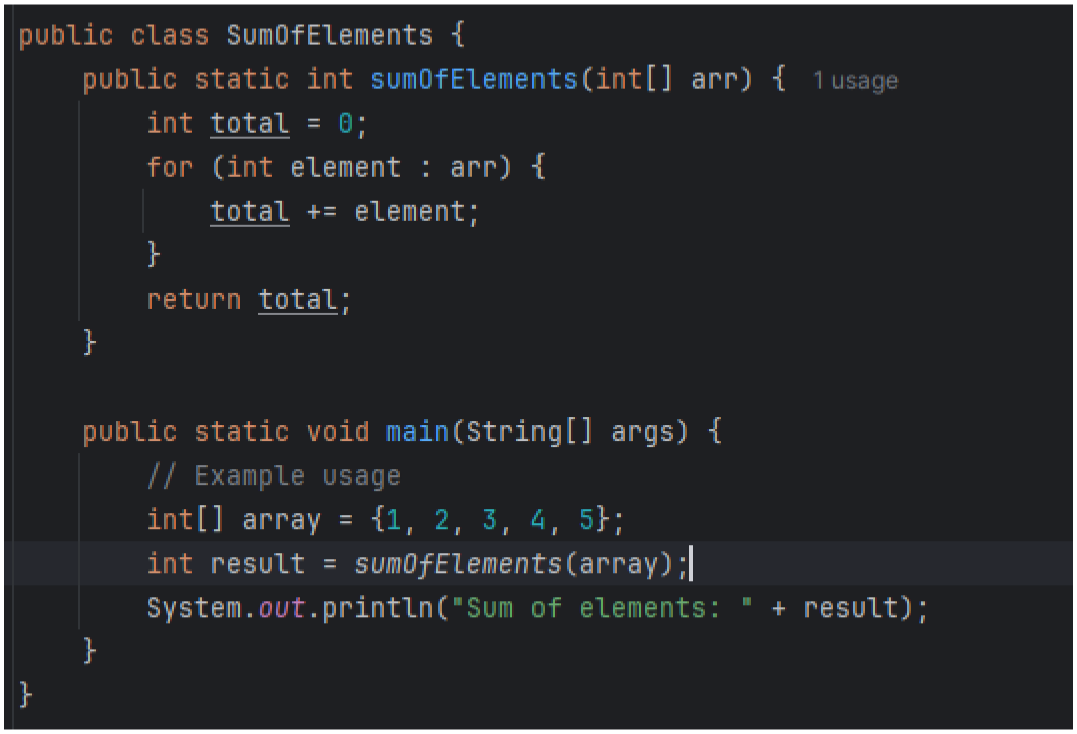
Algorithms with constant time complexity have a fixed runtime that does not depend on the input size. It means the algorithm takes the same amount of time to execute regardless of the input. An example of an algorithm with constant time complexity is accessing an element in an array by its index.

Linear Time (O(n)):
Algorithms with linear time complexity have a runtime that grows linearly with the input size. This means the execution time increases proportionally with the size of the input. An example of an algorithm with linear time complexity is finding the maximum element in an unsorted array.

Quadratic Time (O(n^2)):

Algorithms with quadratic time complexity have a runtime that grows quadratically with the input size. This means the execution time increases exponentially as the input size increases. An example of an algorithm with quadratic time complexity is the Bubble Sort algorithm. Logarithmic Time (O(log n)): Algorithms with logarithmic time complexity have a runtime that grows logarithmically with the input size. This means the execution time increases slowly even with a large input size. An example of an algorithm with logarithmic time complexity is the Binary Search algorithm.

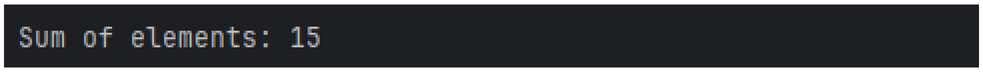
Here is an example of a time complexity:



```java
public class SumOfElements {
    public static int sumOfElements(int[] arr) {  1 usage
        int total = 0;
        for (int element : arr) {
            total += element;
        }
        return total;
    }

    public static void main(String[] args) {
        // Example usage
        int[] array = {1, 2, 3, 4, 5};
        int result = sumOfElements(array);
        System.out.println("Sum of elements: " + result);
    }
}
```

Time Complexity Example

In this example, an array of integers array is created with values {1, 2, 3, 4, 5}. The sumOfElements method is then called with this array, and the result is printed to the console. When you run this Java program, it will output:



```
Sum of elements: 15
```

Time Complexity Result

Space Complexity and example.
Definition:

An algorithm requires a specific quantity of memory space in order to operate on a computer. The space complexity of a program indicates how much memory it need to run. A program's memory needs for storing temporal values and input data while it is operating result in auxiliary and input space space complexity.

Here are different types of space complexity in an algorithm:

Constant Space (O(1)):

Algorithms with constant space complexity use a fixed amount of memory that does not depend on the input size. Regardless of the input size, the algorithm uses the same amount of memory. An example of an algorithm with constant space complexity is swapping two variables.

Linear Space (O(n)):

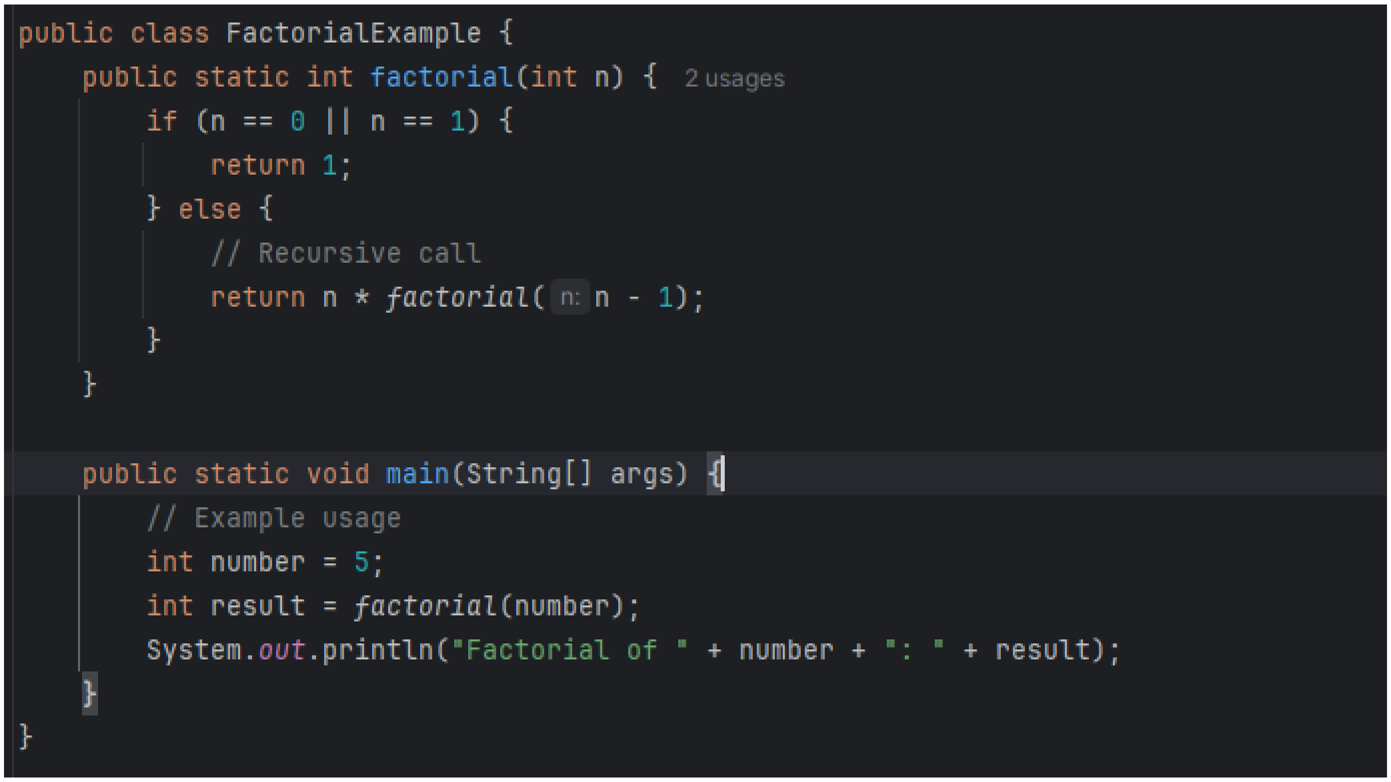
Algorithms with linear space complexity use additional memory that grows linearly with the input size. The amount of memory used by the algorithm increases proportionally with the input size. An example of an algorithm with linear space complexity is creating a new array to store the elements of an existing array.

Quadratic Space (O(n^2)):

Algorithms with quadratic space complexity use additional memory that grows quadratically with the input size. The amount of memory used by the algorithm increases exponentially as the input size increases. An example of an algorithm with quadratic space complexity is generating a matrix of size n x n.

Logarithmic Space (O(log n)):

Algorithms with logarithmic space complexity use additional memory that grows logarithmically with the input size. The amount of memory used by the algorithm increases slowly, even with a large input size. An example of an algorithm with logarithmic space complexity is recursive binary search.
Here's an example of it:



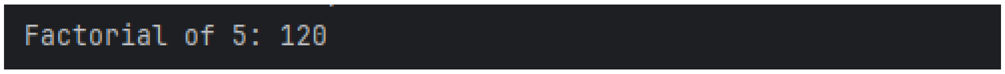
```java
public class FactorialExample {
    public static int factorial(int n) {  2 usages
        if (n == 0 || n == 1) {
            return 1;
        } else {
            // Recursive call
            return n * factorial( n: n - 1);
        }
    }

    public static void main(String[] args) {
        // Example usage
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + ": " + result);
    }
}
```

Space Complexity Example

In this example:

• The factorial of an integer n is computed recursively using the factorial method.
• When n is either 0 or 1, the factorial is 1 in the base case.
• The method uses n - 1 in a recursive call for other values of n.

Space on the call stack is used up by each recursive call. When n is big, the number of recursive calls increases in a linear fashion, resulting in an O(n) space complexity. This occurs as a result of the space needed on the call stack building up with each recursive call until the base case is reached.
Output:



```
Factorial of 5: 120
```

Space Complexity Result

In this example, the program calculates the factorial of 5, which is 5 × 4 × 3 × 2 × 1, and prints the result, which is 120. The recursive calls are made to calculate the factorial, and the space complexity is O(n) due to the recursive nature of the algorithm.

III. Conclusion

The introduction and application of abstract data types (ADTs) have significantly impacted the development of the student ranking system at Soft Development ABK. By encapsulating data and operations, ADTs have facilitated a modular design, improved code maintainability, and enhanced testing capabilities. The student ranking system, which allows users to manage student information and classify students based on their marks, has benefited from these advantages, resulting in a robust and user-friendly application.

Our evaluation of the current algorithm, which employs linear search and sort techniques, against an alternative approach using binary search and merge sort, has provided valuable insights. While the current algorithm is sufficient for small datasets, the alternative offers better performance for larger datasets, demonstrating the importance of selecting appropriate algorithms based on specific requirements.

In conclusion, the use of ADTs in conjunction with careful algorithm selection has proven to be a powerful strategy for developing efficient and maintainable software. As we continue to refine the student ranking system and other projects, these principles will guide us in delivering high-quality solutions that meet the evolving needs of our clients.

IV. References

P1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures. – Page 8 – 18

P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer. – Page 18 - 21

P3 Specify the abstract data type for a software stack using an imperative definition. – Page 36 - 42

P4 Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem. – Page 42 - 48

P5 Implement error handling and report test results. – Page 48 - 50

P6 Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm. – Page 50 - 53

P7 Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example. – Page 53 - 56

M1 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.
– Page 21 - 31

M2 Compare the performance of two sorting algorithms. – Page 31 - 36

V. References

Link GitHub: https://github.com/nguyentuananhBH667/DSM-Main.git

adservio.fr. (n.d.). *Data Structure | Types | Operations*. [online] Available at: https://www.adservio.fr/post/data-structure-types-operations#el6.

Gupta, E. (2024). *Difference Between Time Complexity and Space Complexity*. [online] Shiksha.com. Available at: https://www.shiksha.com/online-courses/articles/difference-between-time-complexity- and-space-complexity-blogId-151433#2

GeeksforGeeks. (2022). *Introduction to Stack memory*. [online] Available at: https://www.geeksforgeeks.org/introduction-to-stack-memory/.

GeeksforGeeks. (2021). *Stack Frame in Computer Organization*. [online] Available at: https://www.geeksforgeeks.org/stack-frame-in-computer-organization/.

Quora. (2019). *What is the importance of stack in data structure?* [online] Available at: https://www.quora.com/What-is-the-importance-of-stack-in-data-structure

GeeksforGeeks. (2022). *What is Data Structure: Types, Classifications and Applications*. [online] Available at: https://www.geeksforgeeks.org/what-is-data-structure-types-classifications-and-applications/.

GeeksforGeeks (2017). *Abstract Data Types - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/abstract-data-types/.