DATA STRUCTURE

SE06304

Part 1

A STACK ADT, A CONCRETE DATA STRUCTURE FOR A FIRST IN FIRST OUT (FIFO) QUEUE.

What is a Data structure algorithm ?

A data structure is a specific way to organize, manage, and store data so it can be accessed and modified efficiently. Common data structures include arrays, linked lists, stacks, queues, trees, graphs, and hash tables, each suited to particular kinds of operations.
An algorithm is a finite sequence of well-defined instructions to solve a problem or perform a computation. When combined with data structures, algorithms can help in managing data effectively, allowing for efficient search, sorting, insertion, deletion, and data manipulation.
Together, Data Structures and Algorithms (DSA) form the backbone of computer science, as they provide ways to handle and process data in an optimized manner. Key examples include:

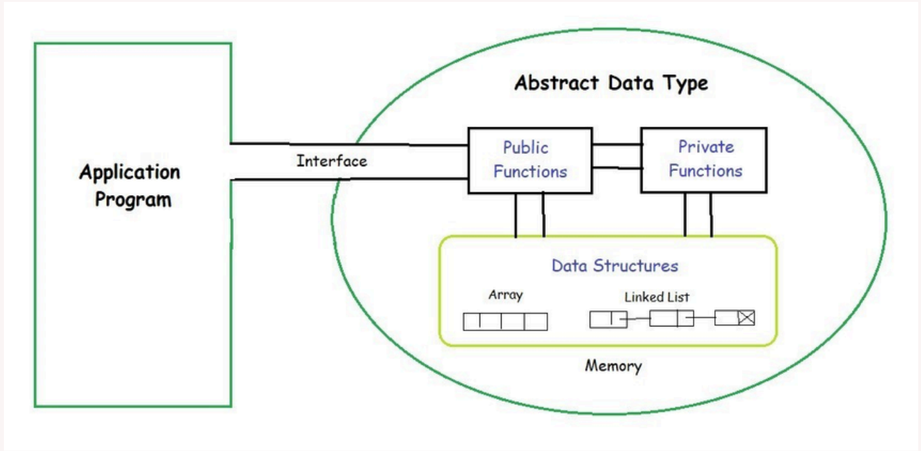
- Searching Algorithms: like binary search, which efficiently finds an element in a sorted array.
- Sorting Algorithms: such as quicksort or mergesort, used to sort data in various ways.
- Graph Algorithms: like Dijkstra's algorithm for finding the shortest path, useful in networking and pathfinding.
- Dynamic Programming: for breaking complex problems into simpler overlapping subproblems, often used in optimization.

What is ADT?

Abstract Data type (ADT) is a type (or class) for objects whose behavior 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.

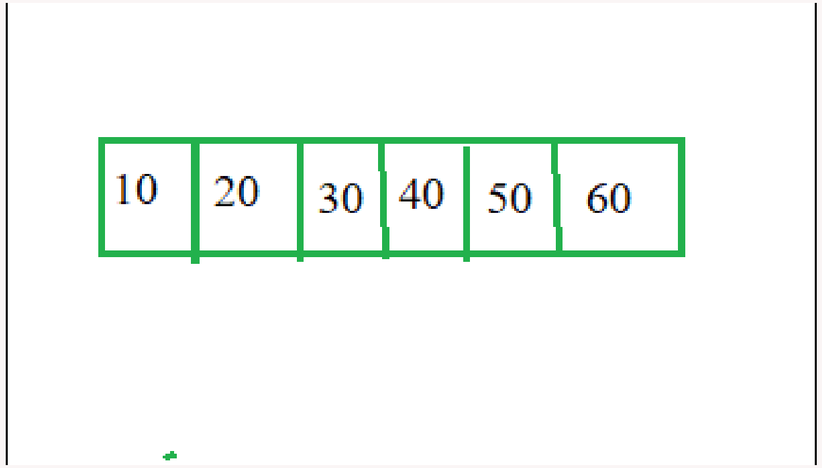
The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we’ll define three ADTs namely List ADT, Stack ADT, Queue ADT.



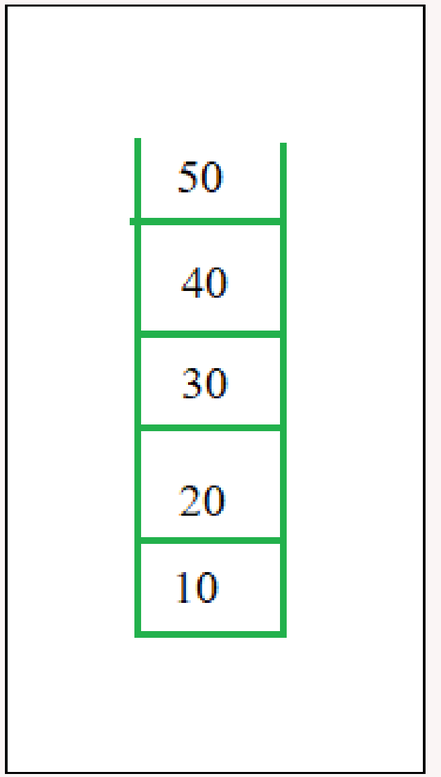
List ADT

- 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.
- 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.



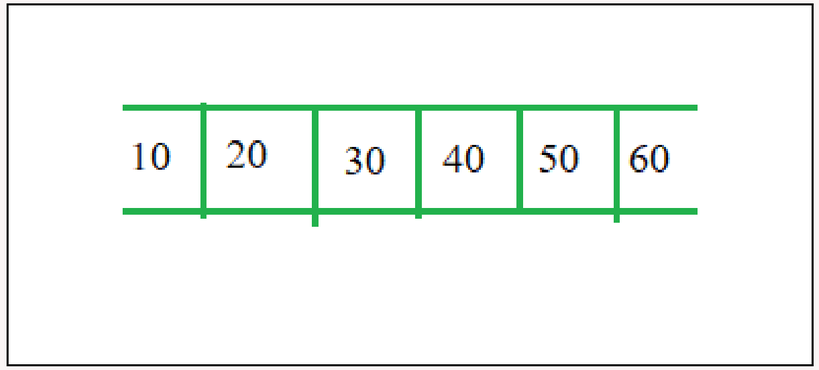
Stack ADT

- 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.
- 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.



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.
- 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.



The difference between queue and stack

Parameter	Stack Data Structure	Queue Data Structure
Basics	It is a linear data structure. The objects are removed or inserted at the same end.	It is also a linear data structure. The objects are removed and inserted from two different ends.
Working Principle	It follows the Last In, First Out (LIFO) principle. It means that the last inserted element gets deleted at first.	It follows the First In, First Out (FIFO) principle. It means that the first added element gets removed first from the list.
Pointers	It has only one pointer - the **top**. This pointer indicates the address of the topmost element or the last inserted one of the stack.	It uses two pointers (in a simple queue) for reading and writing data from both the ends - the **front** and the **rear**. The rear one indicates the address of the last inserted element, whereas the front pointer indicates the address of the first inserted element in a queue.
Operations	Stack uses **push** and **pop** as two of its operations. The pop operation functions to remove the element from the list, while the push operation functions to insert the element in a list.	Queue uses **enqueue** and **dequeue** as two of its operations. The dequeue operation deletes the elements from the queue, and the enqueue operation inserts the elements in a queue.

Structure	Insertion and deletion of elements take place from one end only. It is called the top.	It uses two ends- front and rear. Insertion uses the rear end, and deletion uses the front end.
Full Condition Examination	When top== max-1, it means that the stack is full.	When rear==max-1, it means that the queue is full.
Empty Condition Examination	When top==-1, it indicates that the stack is empty.	When front = rear+1 or front== -1, it indicates that the queue is empty.
Variants	A Stack data structure does not have any types.	A Queue data structure has three types- circular queue, priority queue, and double-ended queue.
Visualization	You can visualize the Stack as a vertical collection.	You can visualize a Queue as a horizontal collection.
Implementation	The implementation is simpler in a Stack.	The implementation is comparatively more complex in a Queue than a stack.

How many ways are there to implement Stack and Queue?

There are several ways to implement Stack and Queue data structures, each with its own trade-offs in terms of time and space complexity. Here's a rundown of the common implementations for each:

Stack Implementations

A stack is a Last-In-First-Out (LIFO) data structure, where the last element added is the first one removed. Here are the main ways to implement it:

1. Array-based Stack:
   - Uses a fixed-size or dynamically resizable array.
   - Operations (push, pop, peek) are generally fast (O(1)), but resizing an array can take longer in the worst-case scenario.
   - Suitable when you know the maximum size of the stack in advance or have a dynamic array that resizes automatically.
2. Linked List-based Stack:
   - Uses a linked list where each node points to the next one in line.
   - Push and pop operations are O(1), and the stack size can grow dynamically without resizing.
   - This approach is memory-efficient if the stack grows and shrinks unpredictably.
3. Deque-based Stack (using libraries):
   - Many programming languages offer a deque (double-ended queue) in standard libraries that supports stack operations.
   - Deques allow push and pop from both ends with O(1) time complexity.
   - This implementation is very flexible but relies on library support.

Stack example using array-based

```java
public class StudentStack {  2 usages  hientran123c *
    private Student[] stack;  7 usages
    private int top;  9 usages

    public StudentStack() {  1 usage  hientran123c
        stack = new Student[10];  // Initialize with a size of 10
        top = -1;
    }

    // Push: Add an element to the stack
    public void push(Student student) {  hientran123c
        if (top == stack.length - 1) {
            System.out.println("Stack is full! Cannot push more elements.");
            return;
        }
        stack[++top] = student;
        System.out.println("Pushed: " + student);
    }

    // Pop: Remove an element from the stack
    public Student pop() {  2 usages  hientran123c
        if (isEmpty()) {
            System.out.println("Stack is empty! Cannot pop any elements.");
            return null;
        }
        Student student = stack[top--];
        System.out.println("Popped: " + student);
        return student;
    }
```

```java
    // Peek: Look at the top element without removing it
    public Student peek() {  hientran123c
        if (isEmpty()) {
            System.out.println("Stack is empty! No elements to peek.");
            return null;
        }
        System.out.println("Peeked: " + stack[top]);
        return stack[top];
    }

    // Check if the stack is empty
    public boolean isEmpty() {  hientran123c
        return top == -1;
    }

    // Display all elements in the stack
    public void display() {  4 usages  new *
        if (isEmpty()) {
            System.out.println("Stack is empty.");
            return;
        }

        System.out.println("Stack elements (top to bottom):");
        for (int i = top; i >= 0; i--) {
            System.out.println(stack[i]);
        }
    }
    // Return the current size of the stack
    public int size() {  hientran123c
        return top + 1;
    }
}
```

Stack example using linkedlist-based

```java
public class StudentStack {  2 usages  hientran123c *
    private Node top;  // Top of the stack  8 usages
    private int size;  // Size of the stack  5 usages
    // Node class for the linked list
    private static class Node {  4 usages  hientran123c
        Student student;  4 usages
        Node next;  3 usages

        Node(Student student) {  1 usage  hientran123c
            this.student = student;
            this.next = null;
        }
    }

    public StudentStack() {  1 usage  hientran123c
        top = null;
        size = 0;
    }

    // Push: Add an element to the stack
    public void push(Student student) {  hientran123c
        Node newNode = new Node(student);
        newNode.next = top;  // Link the new node to the previous top
        top = newNode;       // Update top to the new node
        size++;
        System.out.println("Pushed: " + student);
    }
```

```java
    // Pop: Remove an element from the stack
    public Student pop() {  1 usage  hientran123c
        if (isEmpty()) {
            System.out.println("Stack is empty! Cannot pop any elements.");
            return null;
        }
        Student student = top.student;
        top = top.next;  // Move top to the next node
        size--;
        System.out.println("Popped: " + student);
        return student;
    }

    // Peek: Look at the top element without removing it
    public Student peek() {  hientran123c
        if (isEmpty()) {
            System.out.println("Stack is empty! No elements to peek.");
            return null;
        }
        System.out.println("Peeked: " + top.student);
        return top.student;
    }

    // Check if the stack is empty
    public boolean isEmpty() {  hientran123c
        return size == 0;
    }

    // Return the current size of the stack
    public int size() {  no usages  hientran123c
        return size;
    }
```

Queue Implementations

A queue is a First-In-First-Out (FIFO) data structure, where the first element added is the first one removed. Here are common ways to implement a queue:

1. Array-based Queue:
   - Uses a fixed-size or resizable array.
   - Elements are enqueued at one end and dequeued from the other.
   - Can suffer from "wasted space" if elements are simply dequeued from the front (unless a circular buffer or dynamic resizing is used).
   - Time complexity for enqueue and dequeue operations is typically O(1) when using a circular array or dynamic shifting.
2. Linked List-based Queue:
   - Uses a linked list where each node represents an element in the queue.
   - Enqueue operations add to the tail, and dequeue operations remove from the head, both O(1).
   - Suitable for scenarios where the queue size changes dynamically.
3. Circular Queue (using array or linked list):
   - Implements a queue in a circular way, especially useful for fixed-size arrays.
   - When the end of the array is reached, the queue wraps around to the beginning if there is space.
   - Avoids "wasted space" in the array but requires handling wrap-around logic.

4 Deque-based Queue (using libraries):

- Many languages offer a deque that supports both stack and queue operations efficiently.
- Deque allows both enqueue and dequeue from both ends, often with O(1) complexity.
- It's a versatile approach but depends on the availability of a deque in the language's standard library.

5 Priority Queue (or Heap-based Queue):

- Often implemented using a binary heap.
- Supports enqueuing elements with a priority, so elements with higher priority are dequeued first.
- Commonly used for applications where priority-based processing is required (like Dijkstra's shortest path).
- Enqueue and dequeue operations usually have O(log n) complexity.

Queue example using array-based

```java
// StudentQueue.java
public class StudentQueue {  2 usages  hientran123c
    private Student[] queue;  6 usages
    private int front;  5 usages
    private int rear;  3 usages
    private int size;  6 usages

    public StudentQueue() {  1 usage  hientran123c
        queue = new Student[10];  // Initialize with a size of 10
        front = 0;
        rear = -1;
        size = 0;
    }

    // Enqueue: Add an element to the queue
    public void enqueue(Student student) {  4 usages  hientran123c
        if (size == queue.length) {
            System.out.println("Queue is full! Cannot enqueue more elements.");
            return;
        }
        rear++;
        queue[rear] = student;
        size++;
        System.out.println("Enqueued: " + student);
    }
```

```java
// Dequeue: Remove an element from the queue
public Student dequeue() {  1 usage  hientran123c
    if (isEmpty()) {
        System.out.println("Queue is empty! Cannot dequeue any elements.");
        return null;
    }
    Student student = queue[front];
    front++;
    size--;
    System.out.println("Dequeued: " + student);
    return student;
}

// Peek: Look at the front element without removing it
public Student peek() {  hientran123c
    if (isEmpty()) {
        System.out.println("Queue is empty! No elements to peek.");
        return null;
    }
    System.out.println("Peeked: " + queue[front]);
    return queue[front];
}

// Check if the queue is empty
public boolean isEmpty() {  hientran123c
    return size == 0;
}

// Return the current size of the queue
public int size() {  no usages  hientran123c
    return size;
}
```

Queue example using linkedlist-based

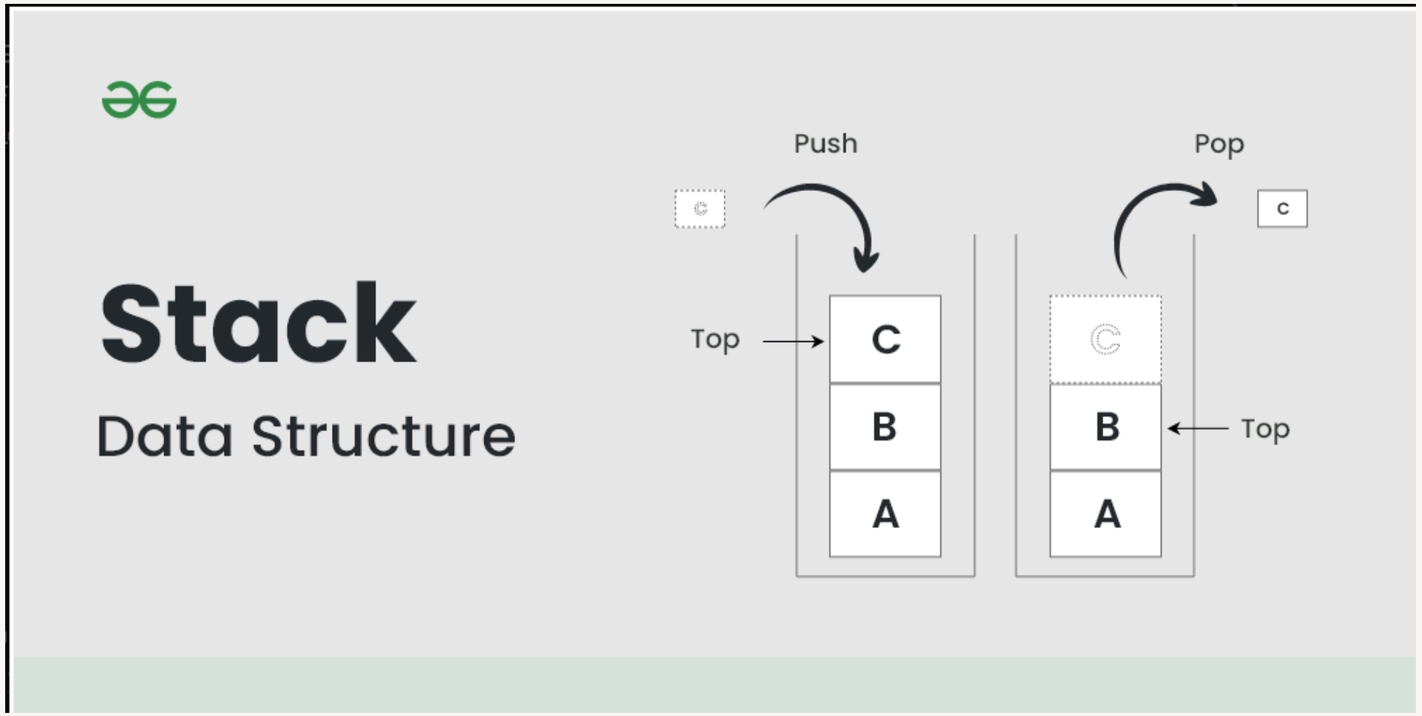
```java
// StudentQueue.java
public class StudentQueue {  2 usages  hientran123c
    private Node front;  9 usages
    private Node rear;  6 usages
    private int size;  4 usages

    // Node class for linked list
    private class Node {  5 usages  hientran123c
        Student student;  4 usages
        Node next;  3 usages

        public Node(Student student) {  1 usage  hientran123c
            this.student = student;
            this.next = null;
        }
    }

    public StudentQueue() {  1 usage  hientran123c
        front = null;
        rear = null;
        size = 0;
    }

    // Enqueue: Add an element to the end of the queue
    public void enqueue(Student student) {  4 usages  hientran123c
        Node newNode = new Node(student);
        if (rear == null) {  // Queue is empty
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        size++;
        System.out.println("Enqueued: " + student);
    }
```

```java
// Dequeue: Remove an element from the front of the queue
public Student dequeue() {  1 usage  hientran123c
    if (isEmpty()) {
        System.out.println("Queue is empty! Cannot dequeue any elements.");
        return null;
    }
    Student student = front.student;
    front = front.next;
    if (front == null) {  // Queue becomes empty after dequeue
        rear = null;
    }
    size--;
    System.out.println("Dequeued: " + student);
    return student;
}

// Peek: Look at the front element without removing it
public Student peek() {  hientran123c
    if (isEmpty()) {
        System.out.println("Queue is empty! No elements to peek.");
        return null;
    }
    System.out.println("Peeked: " + front.student);
    return front.student;
}

// Check if the queue is empty
public boolean isEmpty() {  hientran123c
    return front == null;
}

// Return the current size of the queue
public int size() {  no usages  hientran123c
    return size;
}
```

LIST ALL METHODS OF STACK STRUCTURE AND EXPLAIN HOW IT WORK

What is a stack ?

A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



Constructor method

The Stack constructor in Java is responsible for creating a new Stack object and initializing the initial properties of the stack

```java
public class StudentStack {
    private Student[] stack;
    private int top;

    public StudentStack() {
        stack = new Student[10];  // Initialize with a size of 10
        top = -1;
    }
```

The StudentStack class represents a stack specifically for Student objects. Here's a brief breakdown of each part:

1. Attributes:
   - stack: An array of Student objects, serving as the stack storage.
   - top: An integer that tracks the index of the top element. Initialized to -1, indicating the stack is empty.
2. Constructor (StudentStack):
   - Initializes the stack array with a size of 10, allowing up to 10 Student objects.
   - Sets top to -1 to start the stack in an empty state.

Push method

The push method adds an element to top the stack

```
// Push: Add an element to the stack
    public void push(Student student) {
        if (top == stack.length - 1) {
            System.out.println("Stack is full! Cannot push more elements.");
            return;
        }
        stack[++top] = student;
        System.out.println("Pushed: " + student);
    }
```

The push method adds a Student object to the top of a stack:

1. Check if Full: It first checks if the stack is full. If so, it prints a message and exits.
2. Add Element: If there's space, it increments the top index and places the Student at this new top position.
3. Confirmation: It then prints a message confirming the student was successfully added.

Pop method

The pop operation removes the top element from the stack and returns it. The top index is decremented, and the element is no longer part of the stack.

```
// Pop: Remove an element from the stack
public Student pop() {
    if (isEmpty()) {
        System.out.println("Stack is empty! Cannot pop any elements.");
        return null;
    }
    Student student = stack[top--];
    System.out.println("Popped: " + student);
    return student;
}
```

he pop method removes and returns the top Student object from the stack:

1. Check if Empty: It first checks if the stack is empty using isEmpty(). If empty, it prints a message and returns null.
2. Remove Element: If not empty, it retrieves the Student at the top index, then decrements top to remove it from the stack.
3. Confirmation and Return: It prints a confirmation message showing the popped student and returns the removed Student object.

Peek operation

The peek operation retrieves the top element of the stack without removing it. This allows you to see what the last element added to the stack is, while keeping it in the stack.

```
// Peek: Look at the top element without removing it
public Student peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty! No elements to peek.");
        return null;
    }
    System.out.println("Peeked: " + stack[top]);
    return stack[top];
}
```

The peek method lets you view the top Student object in the stack without removing it:

1. Check if Empty: It first checks if the stack is empty with isEmpty(). If it's empty, it prints a message and returns null.
2. View Top Element: If not empty, it prints the top element without modifying the stack.
3. Return Top Element: It returns the Student object at the top position.

isempty method

This method simply checks the value of the top index to determine if the stack contains any elements.

```
// Check if the stack is empty
public boolean isEmpty() {
    return top == -1;
}
```

The isEmpty method checks if the stack is empty:

1. Condition Check: It returns true if top is -1, meaning there are no elements in the stack.
2. Return Result: If top equals -1, isEmpty returns true; otherwise, it returns false.

This method provides a quick way to determine if the stack has any elements.

size method

This method provides a way to quickly determine how many elements are currently in the stack without needing to iterate through the elements

```
// Return the current size of the stack
public int size() {   hientran123c
    return top + 1;
}
```

The size method returns the current number of elements in the stack:

1. Calculate Size: It calculates the size by returning top + 1. Since top is the index of the last element in the stack, adding 1 gives the total count of elements.
2. Return Value: It returns this calculated size as an integer

The difference between pop() and peek()

Stack follow LIFO model that is Last In First Out, so any operation you perform on stack is basically on the top of stack or on the latest entered element in the stack. Hence pop() and peek() are also applied on the latest entered element in stack.

- pop(): Removes the latest entered element from stack and decrement top by 1. pop() doesn't return anything
- peek(): Returns the value of latest entered element.

If you stack contains SO1–SO2–SO3, where 3 is the latest entered element, followed by 2,1,0

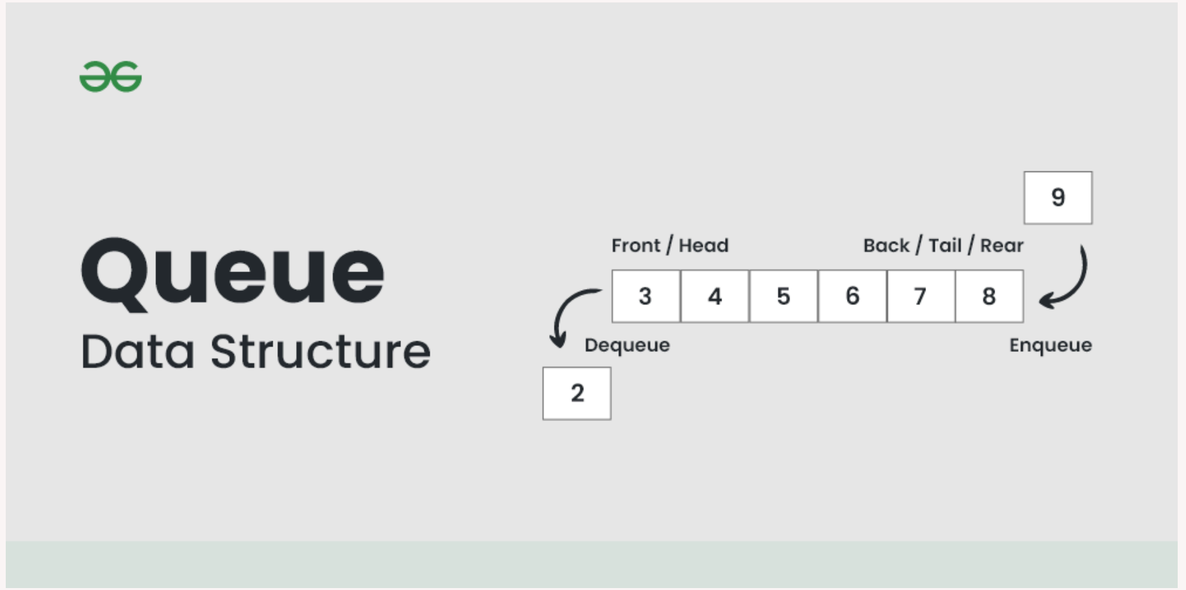
- Stack: SO1-SO2-SO3
- peek(): O/P: 3 Stack Status: SO1-SO2-SO3
- pop(): O/P: 3 Stack Status: SO1-SO2
- pop(): O/P: 3 Stack Status: SO1
- peek(): O/P: 3 Stack Status: SO1

```
Peeked: ID: S03, Name: Bob, Marks: 7.3
Stack elements (top to bottom):
ID: S03, Name: Bob, Marks: 7.3
ID: S02, Name: Alice, Marks: 9.2
ID: S01, Name: John, Marks: 8.5
Popped: ID: S03, Name: Bob, Marks: 7.3
Stack elements (top to bottom):
ID: S02, Name: Alice, Marks: 9.2
ID: S01, Name: John, Marks: 8.5
Popped: ID: S02, Name: Alice, Marks: 9.2
Stack elements (top to bottom):
ID: S01, Name: John, Marks: 8.5
Peeked: ID: S01, Name: John, Marks: 8.5
Stack elements (top to bottom):
ID: S01, Name: John, Marks: 8.5
```

LIST ALL METHODS OF QUEUE STRUCTURE AND EXPLAIN HOW IT WORK

What is a queue ?

A FIFO queue is a queue that operates on the first-in, first-out principle, hence the name. This is also referred to as the first-come, first-served principle.
In other words, FIFO queuing is when customers are served in the exact order in which they arrive. FIFO is the most common type of queuing, and it's generally believed to be the fairest way to manage queues.



Constructor method

The Queue constructor in Java is responsible for creating a new Queue object and initializing the initial properties of the queue

```java
public class StudentQueue {
    private Student[] queue;
    private int front;
    private int rear;
    private int size;

    public StudentQueue() {
        queue = new Student[10];  // Initialize with a size of 10
        front = 0;
        rear = -1;
        size = 0;
    }
```

The StudentQueue class represents a queue specifically for Student objects. Here's a brief explanation of each part:

1. Attributes:
   - queue: An array of Student objects that holds the elements of the queue.
   - front: An integer that indicates the index of the front element in the queue, initialized to 0.
   - rear: An integer that tracks the index of the last element added to the queue, initialized to -1, indicating that the queue is empty.
   - size: An integer that keeps track of the current number of elements in the queue, initialized to 0.
2. Constructor (StudentQueue):
   - Initializes the queue array with a size of 10, allowing up to 10 Student objects.
   - Sets front to 0, rear to -1, and size to 0, establishing an empty queue.

Enqueue method

Add a new element to the rear of queue . It requires the element to be added and returns nothing.

```java
// Enqueue: Add an element to the queue
public void enqueue(Student student) {
    if (size == queue.length) {
        System.out.println("Queue is full! Cannot enqueue more elements.");
        return;
    }
    rear++;
    queue[rear] = student;
    size++;
    System.out.println("Enqueued: " + student);
}
```

The enqueue method adds a Student object to the back of a queue.

- Check if Full: It first checks if the queue is full by comparing size to the length of the queue array. If the queue is full, it prints a message and exits the method.
- Add Element: If there is space, it increments the rear index and assigns the student object to that position in the queue.
- Update Size: It increments the size variable to reflect the addition of a new element.
- Confirmation: Finally, it prints a confirmation message indicating that the student has been successfully enqueued.

Dequeue method

Removes the elements from the front of queue. It does not require any parameters and returns the deleted item.

```java
// Dequeue: Remove an element from the queue
public Student dequeue() {  1 usage   hientran123c
    if (isEmpty()) {
        System.out.println("Queue is empty! Cannot dequeue any elements.");
        return null;
    }
    Student student = queue[front];
    front++;
    size--;
    System.out.println("Dequeued: " + student);
    return student;
}
```

The dequeue method removes and returns the front Student object from the queue:

1. Check if Empty: It first checks if the queue is empty using isEmpty(). If it is, it prints a message and returns null.
2. Remove Element: If not empty, it retrieves the Student at the front index of the queue.
3. Update Indices: The front index is incremented by 1 to point to the next element, and the size of the queue is decremented.
4. Confirmation and Return: It prints a confirmation message showing the dequeued student and returns the removed Student object.

Peek / Front method

The method accesses the element at the front of the queue and returns it without modifying the queue.

```
// Peek: Look at the front element without removing it
public Student peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty! No elements to peek.");
        return null;
    }
    System.out.println("Peeked: " + queue[front]);
    return queue[front];
}
```

The peek method allows you to view the front Student object in the queue without removing it:

1. Check if Empty: It first checks if the queue is empty using isEmpty(). If the queue is empty, it prints a message and returns null.
2. View Front Element: If the queue is not empty, it prints the front element, which is the Student at the front index.
3. Return Front Element: It returns the Student object located at the front position.

isEmpty method

Check the whether the queue is empty or not. It does not require any parameter and returns a Boolean value.

```
// Check if the queue is empty
public boolean isEmpty() {
    return size == 0;
}
```

The isEmpty method checks if the queue is empty:

1. Condition Check: It returns true if size is 0, indicating there are no elements in the queue.
2. Return Result: If size equals 0, the method returns true; otherwise, it returns false.

size method

it provides a straightforward way to get the size of the queue, enhancing usability and functionality.

```
// Return the current size of the queue
public int size() {
    return size;
}
```

The size method returns the current number of elements in the queue:

1. Return Size: It simply returns the value of the size variable, which keeps track of how many elements are currently in the queue.

LET'S PRESENT BUBBLE SORT ALGORITHM AND QUICK SORT

What is bubble sort algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.
Because bubble sorting is a simple process, it may be more efficient when you're sorting smaller data sets. In each pass of a bubble sort, an item at one end of the group finds its correct position. With each successive run, an additional item finds its place and reduces the number of items the next iteration checks until the entire set aligns with your sorting criteria.

How Bubble sort work

- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

VIBLO

Thuật toán sắp xếp **bubble sort**

Code

```java
public static void main(String[] args) {  hientran123c *
    Student[] students = new Student[5];
    // Add students
    students[0] = new Student( id: "S01",  name: "John",  marks: 8.5);
    students[1] = new Student( id: "S02",  name: "Alice",  marks: 9.2);
    students[2] = new Student( id: "S03",  name: "Bob",  marks: 7.3);
    students[3] = new Student( id: "S04",  name: "Charlie",  marks: 6.8);
    students[4] = new Student( id: "S05",  name: "Ropz",  marks: 4);

    // Print unsorted students
    System.out.println("Unsorted students:");
    for(Student x: students)
    {
    System.out.println(x);

    // Bubble sort to sort students by score
    int n = students.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (students[j].getMarks() > students[j + 1].getMarks()) {
                // Swap students
                Student temp = students[j];
                students[j] = students[j + 1];
                students[j + 1] = temp;
            }
        }
    }
    // Print sorted students
    System.out.println("\nSorted students by score(Bubble sort):");
    for(Student x: students) {
        System.out.println(x);
    }
```

Output

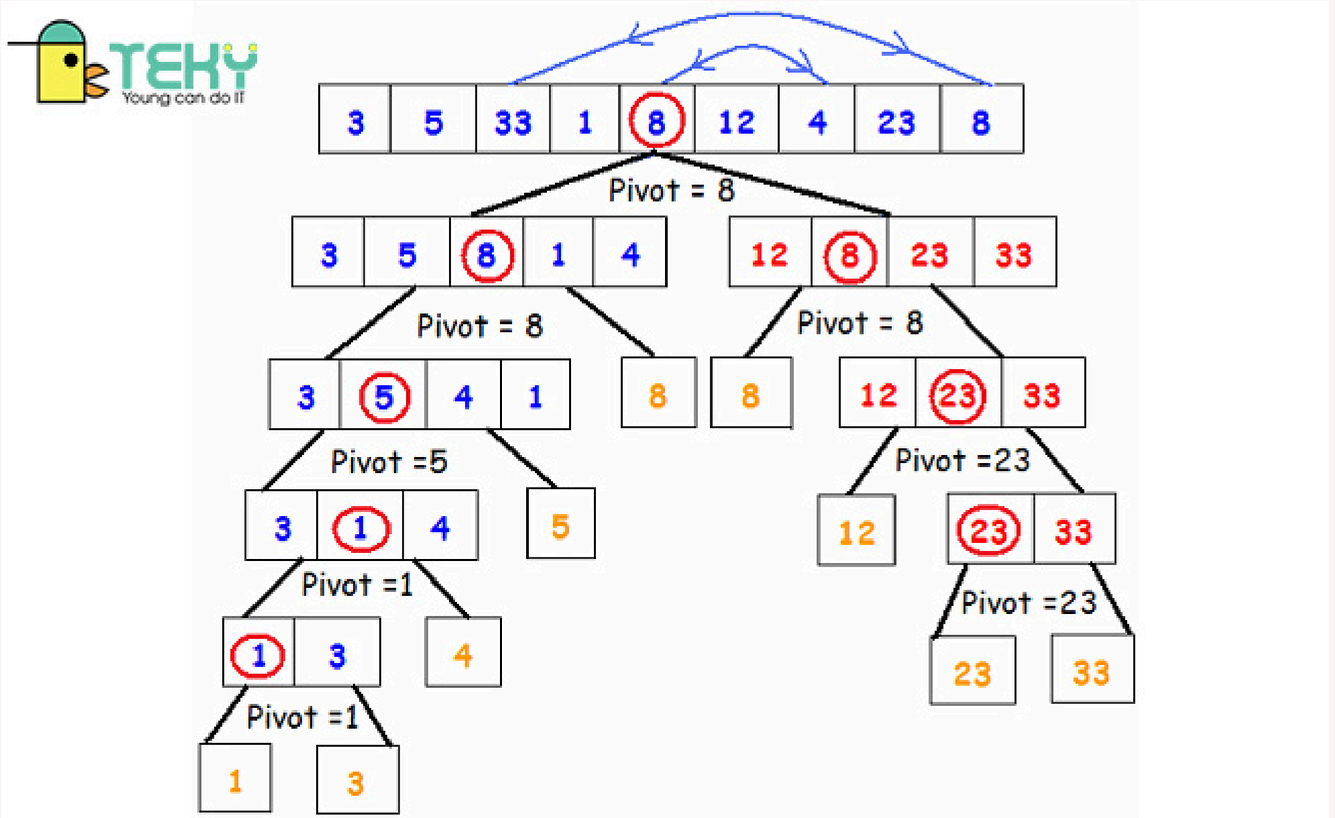
```
Unsorted students:
ID: S01, Name: John, Marks: 8.5
ID: S02, Name: Alice, Marks: 9.2
ID: S03, Name: Bob, Marks: 7.3
ID: S04, Name: Charlie, Marks: 6.8
ID: S05, Name: Ropz, Marks: 4.0

Sorted students by score(Bubble sort):
ID: S05, Name: Ropz, Marks: 4.0
ID: S04, Name: Charlie, Marks: 6.8
ID: S03, Name: Bob, Marks: 7.3
ID: S01, Name: John, Marks: 8.5
ID: S02, Name: Alice, Marks: 9.2
```

What is quick sort algorithm

Quick Sort is an efficient sorting algorithm based on dividing a data array into smaller groups of elements. The quick sort algorithm divides the array into two parts by comparing each element of the array with an element called the key element. One array consists of elements smaller than or equal to the key element and the other array consists of elements larger than the key element. This division process continues until the length of the sub-arrays is equal to 1. With the recursive method, we can quickly sort the sub-arrays after finishing the program, we get a completely sorted array. The quick sort algorithm proves to be quite effective with large data sets when the complexity is O(nlogn).

How Quick sort algorithm work

1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.


TEKY
Young can do it
3 5 33 1 8 12 4 23 8
Pivot = 8
3 5 8 1 4
12 8 23 33
Pivot = 8
Pivot = 8
3 5 4 1
8
8
12 23 33
Pivot =5
Pivot =23
3 1 4
5
12
23 33
Pivot =1
Pivot =23
1 3
4
23
33
Pivot =1
1
3

Code

```
    quickSort(students, low: 0, high: students.length - 1);
    // Print sorted students
    System.out.println("\nSorted students by score (Quick Sort):");
    for (Student x : students) {
        System.out.println(x);
    }
}
// Quick Sort method
private static void quickSort(Student[] students, int low, int high) {  3 usages  new *
    if (low < high) {
        // Choose the pivot from the middle
        int mid = low + (high - low) / 2; // Midpoint for the pivot
        double pivot = students[mid].getMarks();

        // Move the pivot to the end
        Student temp = students[mid];
        students[mid] = students[high];
        students[high] = temp;

        int i = low - 1;

        for (int j = low; j < high; j++) {
            // If current element is smaller than or equal to pivot
            if (students[j].getMarks() <= pivot) {
                i++;
                // Swap students[i] and students[j]
                temp = students[i];
                students[i] = students[j];
                students[j] = temp;
            }
        }
        // Swap students[i + 1] and students[high] (which is the pivot)
        temp = students[i + 1];
        students[i + 1] = students[high];
        students[high] = temp;
        // Recursively sort elements before and after partition
        quickSort(students, low, i);
        quickSort(students, low: i + 2, high);
    }
```

Output

```
Unsorted students:
ID: S01, Name: John, Marks: 8.5
ID: S02, Name: Alice, Marks: 9.2
ID: S03, Name: Bob, Marks: 7.3
ID: S04, Name: Charlie, Marks: 6.8
ID: S05, Name: Ropz, Marks: 4.0

Sorted students by score (Quick Sort):
ID: S05, Name: Ropz, Marks: 4.0
ID: S04, Name: Charlie, Marks: 6.8
ID: S03, Name: Bob, Marks: 7.3
ID: S01, Name: John, Marks: 8.5
ID: S02, Name: Alice, Marks: 9.2
```

Quick sort or Bubble sort is better

1. When comparing Quick Sort and Bubble Sort, Quick Sort is generally considered the superior algorithm for sorting due to its efficiency and performance characteristics. Quick Sort, which is a divide-and-conquer algorithm, has an average-case time complexity of O(n log n), making it significantly faster than Bubble Sort, which has an average-case time complexity of $O(n^2)$. This means that as the size of the dataset increases, Quick Sort will handle larger arrays more effectively than Bubble Sort.
2. Bubble Sort, while straightforward and easy to understand, is inefficient for large datasets because it repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted, which can result in numerous unnecessary comparisons and swaps. As a result, Bubble Sort tends to be more suited for small or nearly sorted datasets where its simplicity can be an advantage. However, for larger and more complex datasets, Quick Sort's efficiency and performance make it the preferred choice among experienced developers and computer scientists.
3. In addition to speed, Quick Sort also has better memory usage, as it operates in-place and requires less additional storage compared to other algorithms like Merge Sort, which can require O(n) additional space. Overall, for practical applications where performance is a key consideration, Quick Sort is typically favored over Bubble Sort.

Compare complexity between 2 sorting algorithms?

1 Bubble sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the list is sorted.

- Time Complexity:
  - Best Case: O(n) — When the array is already sorted, only one pass is needed (with an optimized version of Bubble Sort that stops when no swaps are needed).
  - Average Case: $O(n^2)$ — For a randomly ordered array, it requires multiple passes through the list, making it quadratic.
  - Worst Case: $O(n^2)$ — When the array is sorted in reverse order, it has to perform maximum swaps.
- Space Complexity: O(1) — Bubble Sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory regardless of input size.
- Stability: Stable — Bubble Sort preserves the relative order of equal elements.
- When to Use: Bubble Sort is rarely used in practice due to its inefficiency. It's mainly useful for educational purposes or very small datasets where simplicity is prioritized over speed.

2. Quick sort

Quick Sort is a more efficient, divide-and-conquer sorting algorithm. It works by selecting a "pivot" element and partitioning the array around the pivot, placing all smaller elements on one side and larger elements on the other. It then recursively applies the same process to each partition.

- Time Complexity:
  - Best Case: O(n log n) — When the pivot divides the array into two nearly equal parts at each step, achieving a balanced partition.
  - Average Case: O(n log n) — For a randomly ordered array, Quick Sort usually performs well due to balanced partitions.
  - Worst Case: $O(n^2)$ — If the pivot consistently divides the array into extremely unbalanced partitions (e.g., if the pivot is always the smallest or largest element in a sorted or reverse-sorted array), the algorithm degrades to quadratic complexity. However, this is rare and can often be mitigated by choosing a random or median pivot.
- Space Complexity:
  - In-place Quick Sort: O(log n) — For recursive calls and stack space in an optimized in-place implementation.
  - Out-of-place Quick Sort: O(n) — If implemented non-in-place, but this is rare.

- Stability: Not stable in its basic form — Quick Sort does not necessarily preserve the relative order of equal elements. However, it can be made stable with additional considerations.
- When to Use: Quick Sort is widely used in practice due to its efficient average-case performance and low memory overhead. It's suitable for large datasets, and many libraries and frameworks use it as a default sorting algorithm.

3. Summary

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Stability	Practical Use
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable	Rarely used; educational
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	Not stable	Commonly used; efficient for large datasets

- Quick Sort is much more efficient than Bubble Sort for larger datasets due to its O(n log n) average complexity.
- Bubble Sort is only suitable for small datasets or as an educational tool to understand sorting fundamentals.
- Quick Sort is generally the better choice for practical use, as it handles large data more efficiently and is used in many standard libraries and applications.

WHAT IS DIJKSTRA ALGORITHMS AND HOW IT WORK

What is Dijkstra’s algorithm

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

The need for Dijkstra’s algorithm arises in many applications where finding the shortest path between two points is crucial.

For example, It can be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting point and the Destination

How Dijkstra’s algorithm work

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

THANK YOU