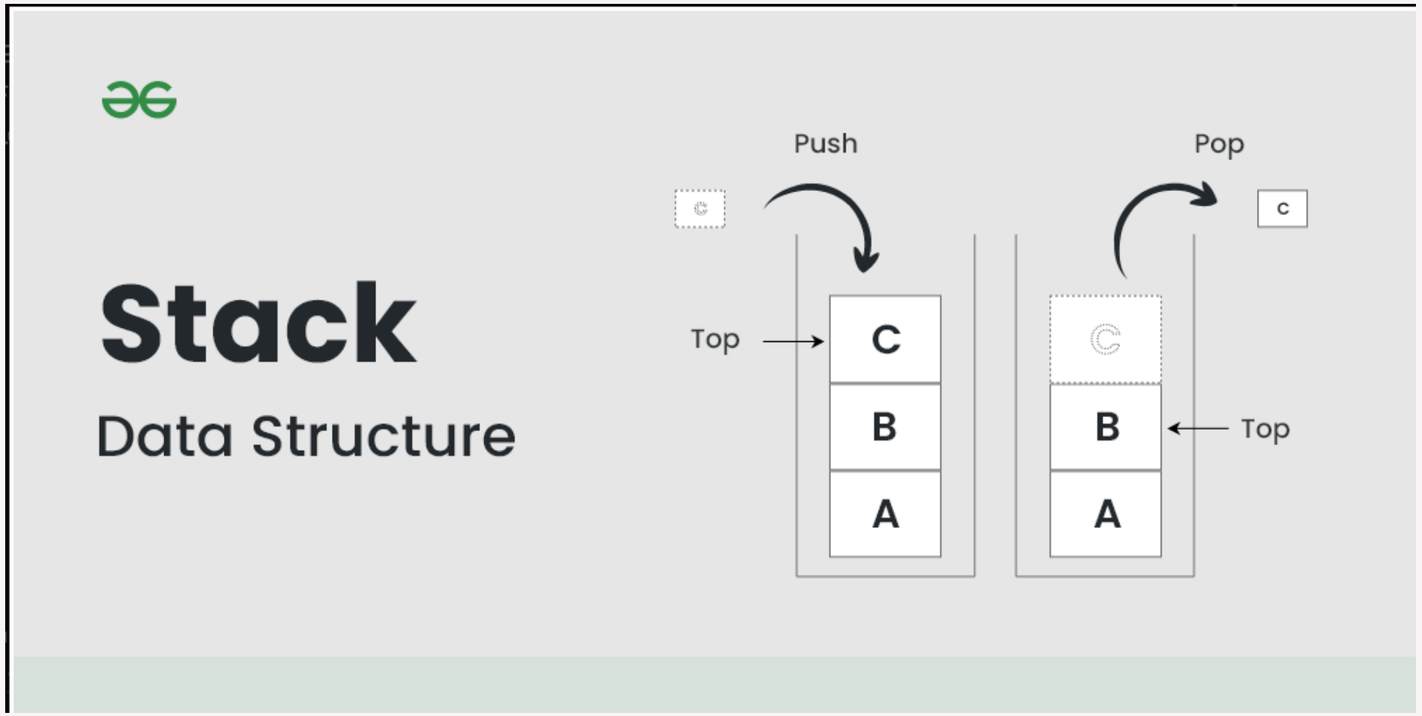
DATA STRUCTURE

SE06304

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

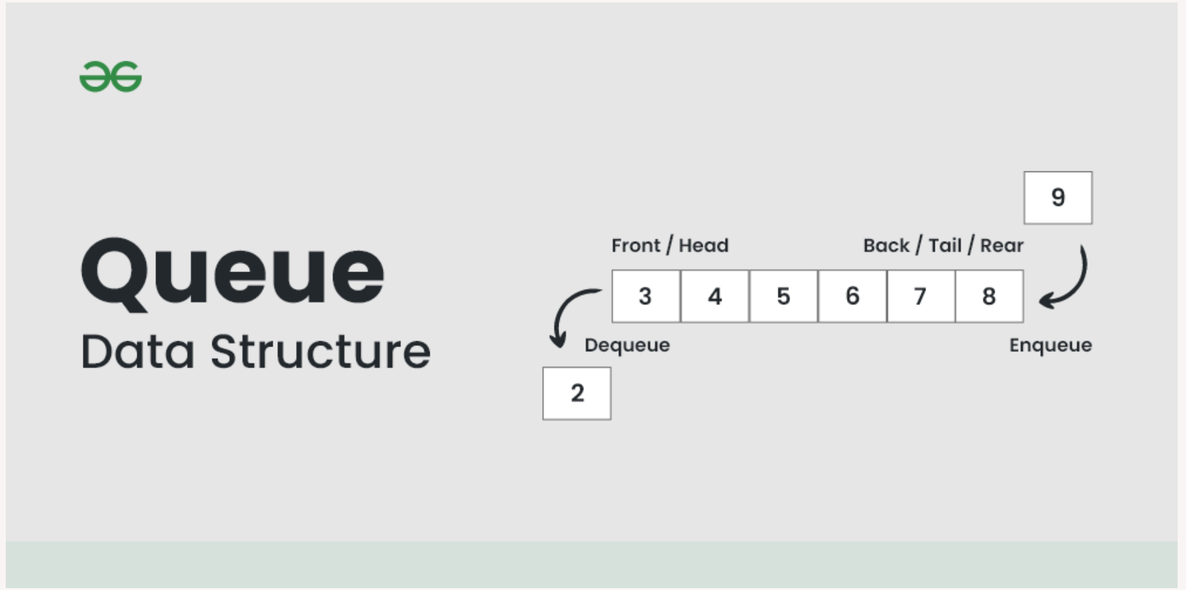
- Stack: SO1-SO2-SO3
- peek(): O/P: 3 Stack Status: SO1-SO2-SO3
- pop(): O/P: 3 Stack Status: SO1-SO2
- pop(): O/P: 3 Stack Status: SO1
- peek(): O/P: 3 Stack Status: SO1

```
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.

```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 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.

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

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.

- 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.

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. Select the key element.
2. Declare 2 pointer variables to point to the 2 sides of the key element.
3. The variable on the left points to each sub-array element on the left of the key element.
4. The variable on the right points to each sub-array element on the right of the key element.
5. When the variable on the left is smaller than the key element, move to the right.
6. When the variable on the right is smaller than the key element, move to the left.
7. If cases 5 and 6 do not occur, swap the values of the 2 variables on the left and right.
8. If the left is larger than the right, this is the new key value.

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);
    }
```

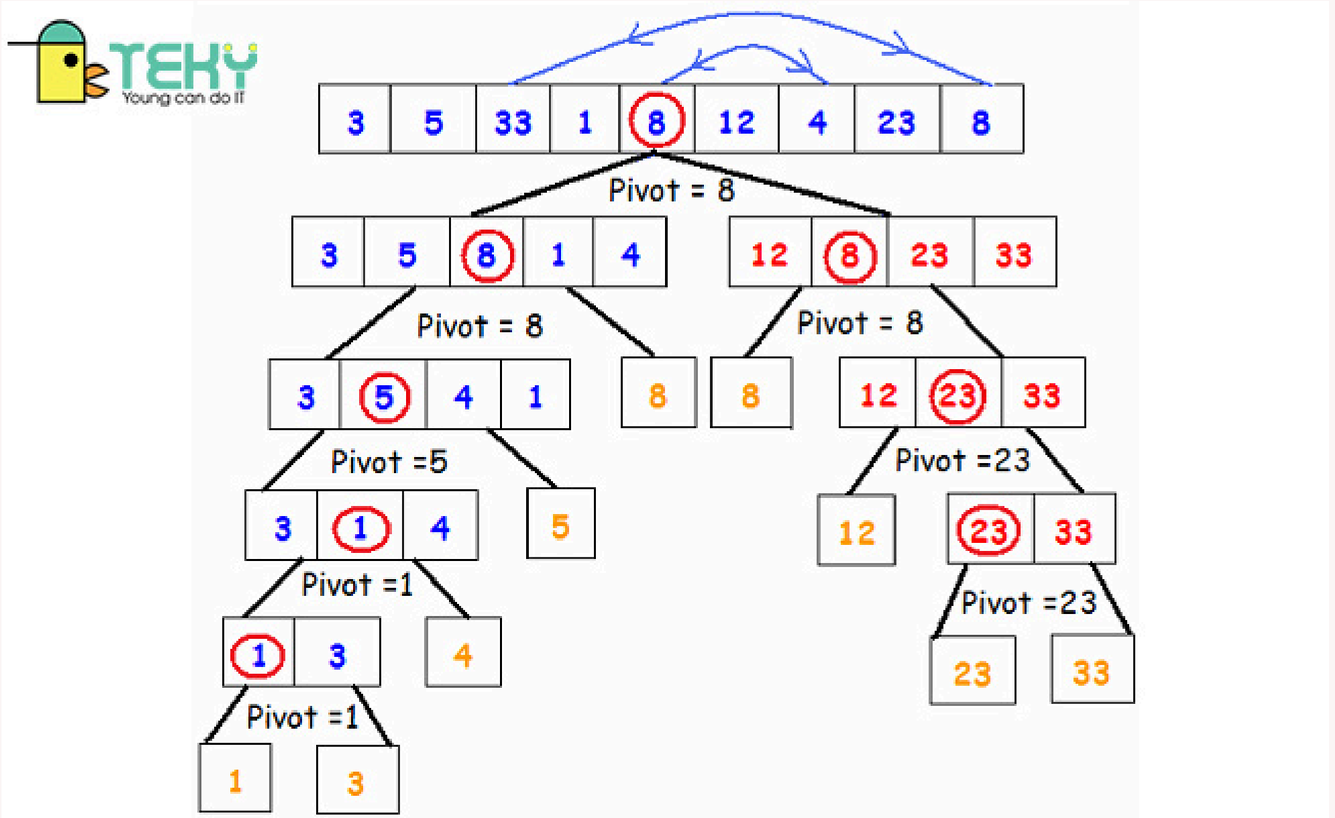
Output

```
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.


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

THANK YOU