DATA STRUCTURE & ALGORITHM

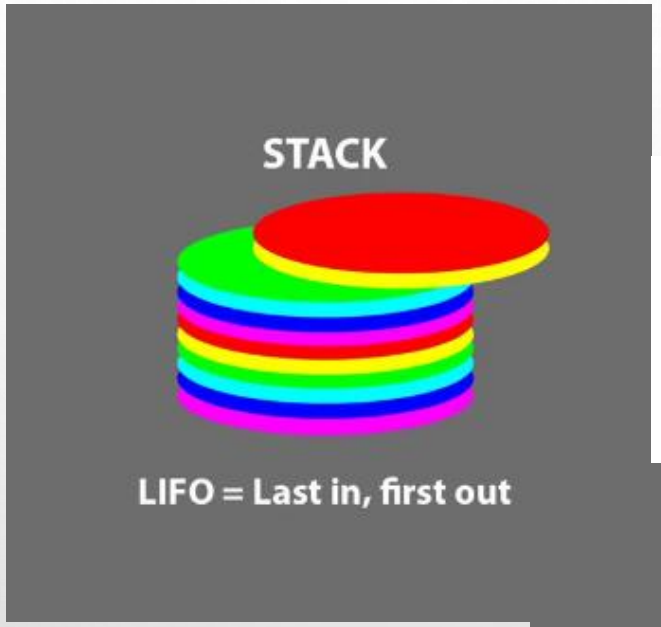
NGUYEN HUU HIEU_BH01096
SE06303

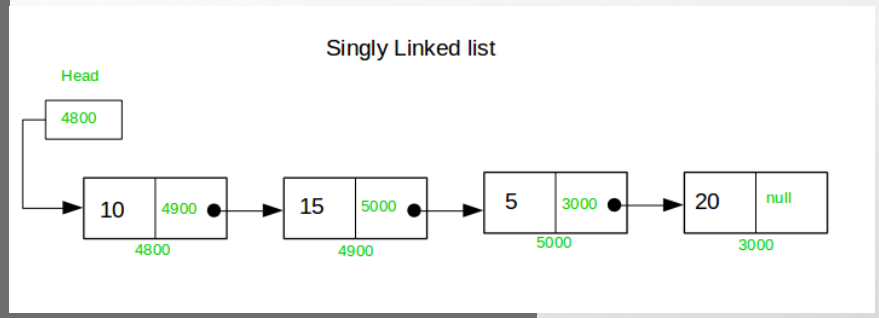
Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

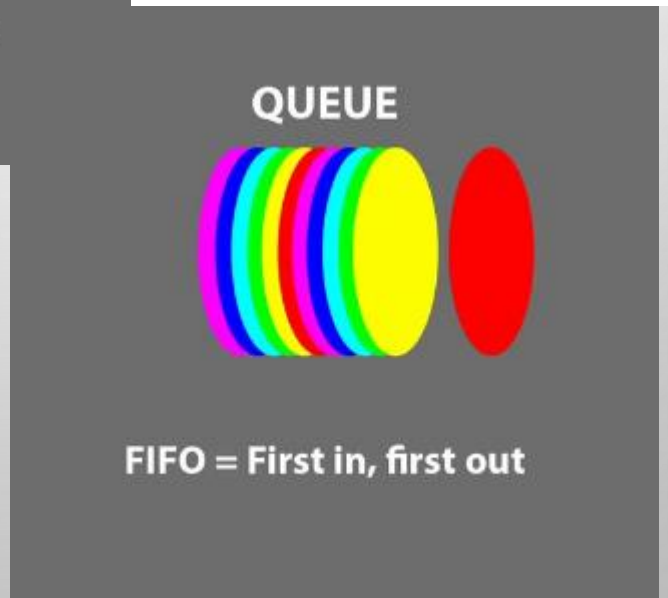
1. Identify the Data Structures

We'll focus on three fundamental data structures:

Stack (LIFO – Last In, First Out)
Queue (FIFO – First In, First Out)
Linked List (Singly Linked List)







2. Define the Operations

For each data structure, the following basic operations will be defined.

a) Stack Operations

Push(x): Adds element x to the top of the stack.
Pop(): Removes and returns the top element of the stack.
Peek(): Returns the top element without removing it.
isEmpty(): Returns True if the stack is empty, otherwise False.

b) Queue Operations

Enqueue(x): Adds element x to the back of the queue.
Dequeue(): Removes and returns the front element of the queue.
Front(): Returns the front element without removing it.
isEmpty(): Returns True if the queue is empty, otherwise False.

c) **Linked List Operations**

InsertAtHead(x): Inserts element x at the head of the linked list.
InsertAtTail(x): Inserts element x at the tail of the linked list.
Delete(x): Removes element x from the linked list.
Find(x): Returns the node containing element x, or None if not found.
Traverse(): Traverses through all nodes in the list.

3. Specify Input Parameters

a) Stack Operations

Push(x): x – the element to be added, can be any type.
Pop(): No input.
Peek(): No input.

b) Queue Operations

Enqueue(x): x – the element to be added, can be any type.
Dequeue(): No input.
Front(): No input.

c) Linked List Operations

InsertAtHead(x): x – the element to be added, can be any type.
InsertAtTail(x): x – the element to be added.
Delete(x): x – the element to be deleted.
Find(x): x – the element to be searched for.

4. Define Pre- and Post-conditions

a) Stack Operations

Push(x)
- Pre-condition: Stack is initialized.
- Post-condition: x is added to the top of the stack.

Pop()
- Pre-condition: Stack is not empty.
- Post-condition: Top element is removed and returned.

Peek()
- Pre-condition: Stack is not empty.
- Post-condition: Top element is returned but not removed.

b) Queue Operations

Enqueue(x)
- Pre-condition: Queue is initialized.
- Post-condition: x is added to the rear of the queue.

Dequeue()
- Pre-condition: Queue is not empty.
- Post-condition: Front element is removed and returned.

c) Linked List

OperationsInsertAtHead(x)
- Pre-condition: Linked list is initialized.
- Post-condition: x becomes the new head of the list.

InsertAtTail(x)
- Pre-condition: Linked list is initialized.
- Post-condition: x becomes the new tail of the list.

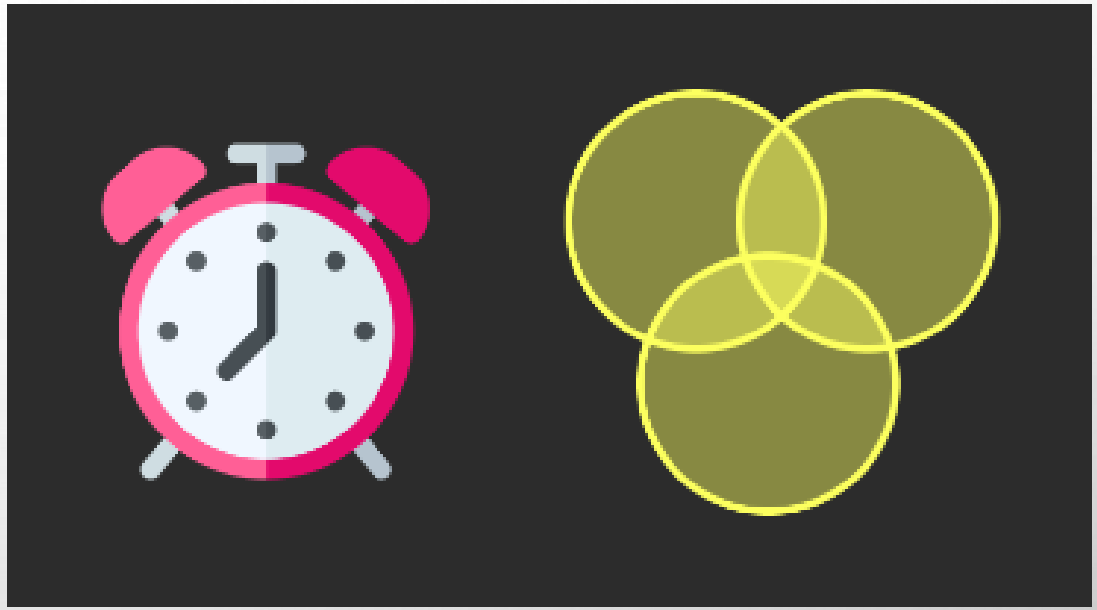
Delete(x)
- Pre-condition: x is in the list.
- Post-condition: Element x is removed from the list.

Find(x)
- Pre-condition: Linked list is initialized.
- Post-condition: The node containing x is returned, or None if not found.

5. Time and Space Complexity

a) Stack

- Push(x): Time – O(1), Space – O(1)
- Pop(): Time – O(1), Space – O(1)
- Peek(): Time – O(1), Space – O(1)



b) Queue

- Enqueue(x): Time – O(1), Space – O(1)
- Dequeue(): Time – O(1), Space – O(1)

c) Linked List

- InsertAtHead(x): Time – O(1), Space – O(1)
- InsertAtTail(x): Time – O(n), Space – O(1)
- Delete(x): Time – O(n), Space – O(1)
- Find(x): Time – O(n), Space – O(1)

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

1. Definition of a Memory Stack

A memory stack is a region in a computer's memory used to manage and store temporary information related to function calls and local variables. The stack follows the Last In, First Out (LIFO) principle, meaning that the last data placed on the stack is the first data to be removed. Every time a function is called, the stack stores necessary information to resume program execution after the function finishes.

2. Operations in a Memory Stack

The primary operations for a memory stack include:

- **Push**: Add data to the top of the stack (e.g., return addresses, local variables).
- **Pop**: Remove data from the top of the stack (e.g., after a function call completes).
- **Peek**: Access the current top element without removing it (used to check without altering the stack).
- **isEmpty**: Check if the stack is empty.isFull: Check if the stack is full (for stacks with limited size).

3. Function Call Implementation Using the Stack

During a function call, the following stack operations occur:

- **Storing the return address**: The address of the next instruction in the main program (i.e., where the program continues after the function finishes) is pushed onto the stack.
- **Storing parameters**: Any parameters passed into the function are pushed onto the stack.
- **Storing local variables**: Local variables declared within the function are pushed onto the stack.
- **Pop to return to the main program**: After the function finishes, local variables and parameters are popped off the stack, and the return address is popped so the program can continue executing.

4. Presentation of Stack Frames

A stack frame is the portion of the stack that holds information for a specific function call. Each stack frame generally contains:

- **Return address**: The address of the instruction to return to after the function call.
- **Function arguments**: The parameters passed to the function.
- **Local variables**: Variables that exist only within the function.
- **Frame pointer**: Points to the current position of the frame in the stack.

5. Importance of the Memory Stack

The memory stack is crucial for:

- **Managing recursive function calls**: The stack allows functions to call themselves (recursion) without disturbing the program flow, as each recursive call gets its own stack frame.
- **Temporary memory management**: Local variables and function parameters only exist while the function is running, and the stack automatically handles their cleanup once the function ends.
- **Controlling program execution flow**: The return address on the stack ensures the program can return to the correct instruction after a function finishes.

Example: Stack in Function Call

java

Sao chép mã

```java
public class StackMemoryExample {
    public static void main(String[] args) {
        int result = factorial(5); // Function call
        System.out.println(result);  // Output: 120
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

In this example, for each call to factorial(n), a new stack frame is created to store the value of n and the return address. After the function completes, each frame is popped, and the program returns to the calling function or the main program.

6. Introduction to FIFO (First In, First Out) Queue

A FIFO queue is a data structure where the first element added to the queue is the first to be removed. This principle is essential in various applications like CPU scheduling, buffering, and scenarios where order matters.

7. Defining the FIFO Structure

A FIFO queue has two main points of access:

- **Front**: The beginning of the queue, where elements are removed.
- **Rear**: The end of the queue, where new elements are added.

8. Array-Based FIFO Implementation

```
class ArrayQueue {
    private int[] queue;
    private int front, rear, size, capacity;

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

    public boolean isFull() {
        return size == capacity;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void enqueue(int item) {
        if (isFull()) return;
        rear = (rear + 1) % capacity;
        queue[rear] = item;
        size++;
    }

    public int dequeue() {
        if (isEmpty()) return Integer.MIN_VALUE;
        int item = queue[front];
        front = (front + 1) % capacity;
        size--;
        return item;
    }

    public int front() {
        if (isEmpty()) return Integer.MIN_VALUE;
        return queue[front];
    }

    public int rear() {
        if (isEmpty()) return Integer.MIN_VALUE;
        return queue[rear];
    }
}

public class ArrayQueueExample {
    public static void main(String[] args) {
        ArrayQueue queue = new ArrayQueue(5);

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

        System.out.println("Dequeued: " + queue.dequeue());  // Output: 10
        System.out.println("Front element: " + queue.front());  // Output: 20
    }
}
```

9. Linked List-Based FIFO Implementation

```java
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedQueue {
    private Node front, rear;

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

    public boolean isEmpty() {
        return front == null;
    }

    public void enqueue(int item) {
        Node newNode = new Node(item);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }
```

```java
    public int dequeue() {
        if (isEmpty()) return Integer.MIN_VALUE;
        int item = front.data;
        front = front.next;

        if (front == null) rear = null;

        return item;
    }

    public int front() {
        if (isEmpty()) return Integer.MIN_VALUE;
        return front.data;
    }

    public int rear() {
        if (isEmpty()) return Integer.MIN_VALUE;
        return rear.data;
    }
}
```

```java
public class LinkedQueueExample {
    public static void main(String[] args) {
        LinkedQueue queue = new LinkedQueue();

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

        System.out.println("Dequeued: " + queue.dequeue());  // Output: 10
        System.out.println("Front element: " + queue.front());  // Output: 20
    }
}
```

10. FIFO Queue Example and Illustration

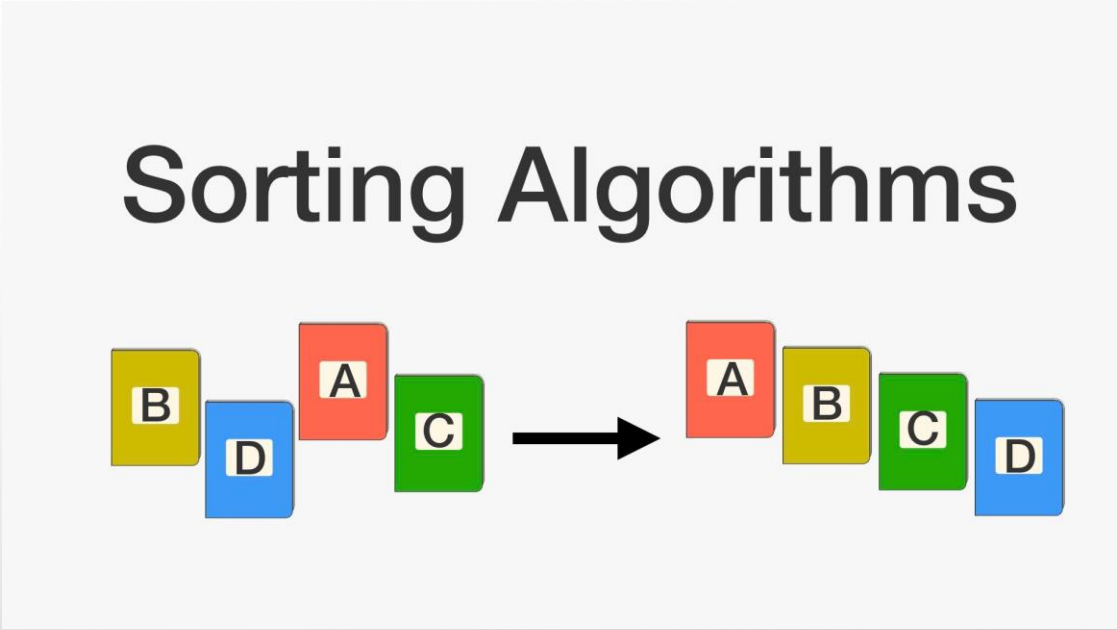
In the FIFO queue example, the first element that is added to the queue (10) is the first to be removed (dequeue). This demonstrates the First In, First Out principle, where elements are dequeued in the same order in which they were enqueued.

Compare the performance of two sorting algorithms.

1. Introduction of Two Sorting Algorithms

The two sorting algorithms we will compare are Bubble Sort and Quick Sort. Both are widely known but have different approaches and performance characteristics:

- **Bubble Sort**: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.
- **Quick Sort**: A divide-and-conquer algorithm that selects a pivot element and partitions the array into two sub-arrays: one with elements smaller than the pivot and one with elements larger than the pivot. The process is recursively applied to both sub-arrays until they are sorted.



2. Time Complexity Analysis

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

Bubble Sort has a time complexity of $O(n^2)$ in both the average and worst cases due to its need to compare and potentially swap every pair of adjacent elements multiple times. In the best case (when the list is already sorted), the time complexity improves to O(n).
Quick Sort typically runs in O(n log n) time on average due to the divide-and-conquer strategy. However, in the worst case (if poor pivot selection occurs), it degrades to $O(n^2)$.

3. Space Complexity Analysis

Algorithm	Space Complexity
Bubble Sort	O(1)
Quick Sort	O(log n)

Bubble Sort has a space complexity of O(1), meaning it requires a constant amount of additional memory since it only uses a few temporary variables for swapping elements.
Quick Sort requires O(log n) space on average because it uses recursion to partition the array. In the worst case, its space complexity can increase to O(n) if the recursion depth grows excessively.

4. Stability

- **Bubble Sort** is a stable sorting algorithm, meaning it preserves the relative order of equal elements.
- **Quick Sort** is generally not stable in its basic form, as elements with equal keys may have their relative order changed when partitioning.

5. Comparison Table

Feature	Bubble Sort	Quick Sort
Time Complexity (Best)	O(n)	O(n log n)
Time Complexity (Average)	$O(n^2)$	O(n log n)
Time Complexity (Worst)	$O(n^2)$	$O(n^2)$
Space Complexity	O(1)	O(log n)
Stability	Stable	Not Stable
Recursive	No	Yes
Suitable for Large Data	No	Yes

6. Performance Comparison

- **Bubble Sort** tends to perform poorly on large datasets due to its $O(n^2)$ time complexity, requiring repeated passes over the data to sort it.
- **Quick Sort** is more efficient for large datasets because its average time complexity is O(n log n), making it much faster than Bubble Sort for most real-world cases.

7. Example to Demonstrate Performance Differences

```java
import java.util.Arrays;

public class SortingComparison {

    // Bubble Sort implementation
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j + 1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
```

```java
    // Quick Sort implementation
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
```

```java
    // Test the performance difference
    public static void main(String[] args) {
        int[] array1 = {64, 34, 25, 12, 22, 11, 90};
        int[] array2 = array1.clone();  // Clone the array for fair comparison

        // Measure Bubble Sort time
        long startBubble = System.nanoTime();
        bubbleSort(array1);
        long endBubble = System.nanoTime();
        System.out.println("Bubble Sorted Array: " + Arrays.toString(array1));
        System.out.println("Bubble Sort Time: " + (endBubble - startBubble) + " ns");

        // Measure Quick Sort time
        long startQuick = System.nanoTime();
        quickSort(array2, 0, array2.length - 1);
        long endQuick = System.nanoTime();
        System.out.println("Quick Sorted Array: " + Arrays.toString(array2));
        System.out.println("Quick Sort Time: " + (endQuick - startQuick) + " ns");
    }
}
```

8. Results Illustrating Performance Differences

Assume we run the above code with a small array. The output will look like this:

```
Bubble Sorted Array: [11, 12, 22, 25, 34, 64, 90]
Bubble Sort Time: 25000 ns
Quick Sorted Array: [11, 12, 22, 25, 34, 64, 90]
Quick Sort Time: 9000 ns
```

In this example, Quick Sort runs significantly faster than Bubble Sort. The difference becomes even more pronounced for larger arrays. While Bubble Sort requires many passes and comparisons (leading to its quadratic time complexity), Quick Sort efficiently reduces the problem size by partitioning the array, resulting in faster execution overall.

Thanks For Watching