Day 1 Programs

Stack_array.java

algorithm for implementing a stack using an array:

Start by defining a class for the stack, which includes the following variables:
- `maxSize`: Maximum size of the stack.
- `stackArray`: Array to hold the stack elements.
- `top`: Index of the top element.

Create a constructor to initialize the stack:
- Accept the maximum size as a parameter.
- Initialize the `maxSize` variable with the given size.
- Create an array of size `maxSize` to store the stack elements.
- Initialize `top` to -1 to indicate an empty stack.

Implement the `push` operation:
- Check if the stack is full (`top` is equal to `maxSize - 1`).
- If the stack is full, display an error message or throw an exception.
- Otherwise, increment `top` by 1 and add the new element to `stackArray[top]`.

Implement the `pop` operation:
- Check if the stack is empty (`top` is equal to -1).
- If the stack is empty, display an error message or throw an exception.
- Otherwise, retrieve the element at `stackArray[top]` and decrement `top` by 1.
- Return the retrieved element.

Implement the `peek` operation:
- Check if the stack is empty (`top` is equal to -1).
- If the stack is empty, display an error message or throw an exception.
- Otherwise, return the element at `stackArray[top]` without modifying `top`.

Implement the `isEmpty` operation:
- Check if `top` is equal to -1.
- Return `true` if the stack is empty, and `false` otherwise.

Implement the `isFull` operation:
- Check if `top` is equal to `maxSize - 1`.
- Return `true` if the stack is full, and `false` otherwise.

That's the basic algorithm for implementing a stack using an array. This algorithm ensures that elements are added and removed from the top of the stack while maintaining the correct stack size and order.

```java
public class stack_array {
    private int maxSize;        // Maximum size of the stack
    private int[] stackArray;   // Array to hold the stack elements
    private int top;            // Index of the top element

    // Constructor to initialize the stack
    public stack_array(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;  // Stack is initially empty
    }

    // Push operation: add an element to the top of the stack
    public void push(int value) {
        if (isFull()) {
            System.out.println("Stack is full. Cannot push element: " + value);
            return;
        }

        top++;
        stackArray[top] = value;
        System.out.println("Pushed element: " + value);
    }

    // Pop operation: remove and return the element from the top of the stack
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop element.");
            return -1;
        }

        int poppedElement = stackArray[top];
        top--;
        return poppedElement;
    }

    // Peek operation: return the element at the top of the stack without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. No top element.");
            return -1;
        }

        return stackArray[top];
    }

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

    // Check if the stack is full
    public boolean isFull() {
        return (top == maxSize - 1);
```

```
    }

    public static void main(String[] args) {
        stack_array stack = new stack_array(5);

        stack.push(10);
        stack.push(20);
        stack.push(30);

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

        System.out.println("Popped element: " + stack.pop());
        System.out.println("Popped element: " + stack.pop());

        stack.push(40);

        System.out.println("Is stack empty? " + stack.isEmpty());
        System.out.println("Is stack full? " + stack.isFull());
    }
}
```

Complexity analysis of loops and recursive algorithms.

Complexity analysis is a fundamental concept in computer science that helps us understand the efficiency of algorithms. When analyzing the complexity of loops and recursive algorithms, we primarily focus on their time complexity, which describes how the execution time of an algorithm grows as the input size increases.

Loops:

Loops are iterative constructs that repeat a block of code until a certain condition is met. The time complexity of a loop depends on the number of iterations it performs. Let's consider a few common scenarios:

- Single loop: If a loop iterates 'n' times, the time complexity is O(n). This is known as linear time complexity because the execution time grows linearly with the input size.
- Nested loops: If we have multiple nested loops, each iterating 'm' and 'n' times, the time complexity becomes O(m * n). The time complexity grows with the product of the loop iterations.
- Nested loops with variable iterations: In some cases, the number of iterations may vary based on the input. Analyzing the time complexity in such cases requires considering the worst-case scenario, which gives us an upper bound on the execution time.

Recursive Algorithms:

Recursive algorithms are functions that call themselves to solve a problem by breaking it down into smaller subproblems. Analyzing the time complexity of recursive algorithms involves understanding how the number of recursive calls grows with the input size.

- Recurrence relations: Recursive algorithms are often described using recurrence relations, which express the time complexity in terms of the number of recursive calls. Solving these relations helps determine the overall time complexity.
- Master theorem: In some cases, we can apply the Master theorem to analyze the time complexity of recursive algorithms with a specific structure. The Master theorem provides a framework for determining the time complexity based on the form of the recurrence relation.
- Examples: Recursive algorithms like binary search, quicksort, and merge sort have well-known time complexities. Binary search has a time complexity of O(log n) as it halves the search space in each recursive call. Quicksort and merge sort both have an average-case time complexity of O(n log n) and worst-case time complexity of O(n^2) and O(n log n), respectively.

In addition to time complexity, it's also important to consider space complexity when analyzing algorithms. Space complexity refers to the amount of memory an algorithm requires. Recursive algorithms, in particular, can have significant space complexity due to the recursive function calls creating additional stack frames.

Implement queues with inserting element at different location (First, Last)

algorithm:

Initialize the queue with an array of a fixed size, along with variables to keep track of the first and last indices, and the size of the queue.
To insert an element at the first location (enqueueFirst):

- Check if the queue is full by comparing the size with the capacity of the array. If it is full, throw an exception or handle it accordingly.
- Decrement the first index by 1, taking care of wrapping around to the end of the array if necessary.
- Store the new element at the updated first index.
- Increment the size of the queue.

To insert an element at the last location (enqueueLast):

- Check if the queue is full by comparing the size with the capacity of the array. If it is full, throw an exception or handle it accordingly.
- Increment the last index by 1, taking care of wrapping around to the start of the array if necessary.
- Store the new element at the updated last index.
- Increment the size of the queue.

To remove an element from the queue (dequeue):

- Check if the queue is empty by comparing the size with 0. If it is empty, throw an exception or handle it accordingly.
- Retrieve the element at the first index.
- Set the element at the first index to null to remove it from the queue.
- Increment the first index by 1, taking care of wrapping around to the start of the array if necessary.
- Decrement the size of the queue.
- Return the retrieved element.

```
public class CustomQueue<T> {
    private Object[] queue;
    private int firstIndex;
    private int lastIndex;
    private int size;

    public CustomQueue(int capacity) {
        queue = new Object[capacity];
        firstIndex = -1;
        lastIndex = -1;
        size = 0;
    }

    public void enqueueFirst(T element) {
        if (isFull()) {
            throw new IllegalStateException("Queue is full");
```

```
        }

        if (isEmpty()) {
            firstIndex = 0;
            lastIndex = 0;
        } else {
            firstIndex = (firstIndex - 1 + queue.length) % queue.length;
        }

        queue[firstIndex] = element;
        size++;
    }

    public void enqueueLast(T element) {
        if (isFull()) {
            throw new IllegalStateException("Queue is full");
        }

        if (isEmpty()) {
            firstIndex = 0;
            lastIndex = 0;
        } else {
            lastIndex = (lastIndex + 1) % queue.length;
        }

        queue[lastIndex] = element;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }

        T element = (T) queue[firstIndex];
        queue[firstIndex] = null;

        if (firstIndex == lastIndex) {
            firstIndex = -1;
            lastIndex = -1;
        } else {
            firstIndex = (firstIndex + 1) % queue.length;
        }

        size--;
        return element;
    }

    public int size() {
        return size;
    }

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

```
    public boolean isFull() {
        return size == queue.length;
    }

    public static void main(String[] args) {
        CustomQueue<Integer> queue = new CustomQueue<>(5);
        queue.enqueueLast(10);
        queue.enqueueFirst(5);
        queue.enqueueLast(20);

        while (!queue.isEmpty()) {
            System.out.println(queue.dequeue());
        }
    }
}
```

Implement circular queue

Algorithm

circular queue:

Initialize the circular queue with an array of a fixed size, along with two pointers - `front` and `rear`.

- `front` represents the index of the front element in the queue.
- `rear` represents the index of the rear element in the queue.
- Initially, both `front` and `rear` are set to -1 to indicate an empty queue.

Enqueue an element into the circular queue:

- Check if the queue is full by comparing `(rear + 1) % capacity` with `front`. If they are equal, the queue is full.
- If the queue is full, throw an exception or handle it accordingly.
- Otherwise, increment `rear` by 1 using the modulus operator `%` to wrap around the array if necessary.
- Store the new element at the `rear` index in the array.

Dequeue an element from the circular queue:

- Check if the queue is empty by comparing `front` with -1. If it is -1, the queue is empty.
- If the queue is empty, throw an exception or handle it accordingly.
- Otherwise, retrieve the element at the `front` index from the array.
- Set the value at the `front` index to null or any appropriate placeholder value.
- If `front` and `rear` are equal, this indicates that the queue is now empty. Set `front` and `rear` to -1.
- Otherwise, increment `front` by 1 using the modulus operator `%` to wrap around the array if necessary.
- Return the retrieved element.

Additional operations:

- You can include operations like `isEmpty` to check if the queue is empty, `isFull` to check if the queue is full, and `size` to get the number of elements in the queue.
- For `isEmpty`, check if `front` is -1.
- For `isFull`, compare `(rear + 1) % capacity` with `front`.
- For `size`, calculate the size based on the positions of `front` and `rear`.

This algorithm provides a basic outline for implementing a circular queue using an array.

```
public class CircularQueue<T> {
    private T[] queue;
    private int front;
    private int rear;
    private int size;
    private int capacity;

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

    public void enqueue(T element) {
        if (isFull()) {
            throw new IllegalStateException("Queue is full");
        }

        rear = (rear + 1) % capacity;
        queue[rear] = element;

        if (front == -1) {
            front = rear;
        }

        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }

        T element = queue[front];
        queue[front] = null;

        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % capacity;
        }

        size--;
```

```
        return element;
    }

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

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

    public int size() {
        return size;
    }

    public static void main(String[] args) {
        CircularQueue<Integer> queue = new CircularQueue<>(5);
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.dequeue();
        queue.enqueue(40);

        while (!queue.isEmpty()) {
            System.out.println(queue.dequeue());
        }
    }
}
```