DATA STRUCTURES

LAB SHEET 1

TITLE: Implementing and Analyzing Basic Data Structures

Objective

The objective of this assignment is to introduce students to fundamental data structures and algorithms, their importance, and their classification. Students will implement basic data structures and analyze their performance using asymptotic notations.

Problem Description

1. **Implementing Static and Dynamic Arrays:**
    - Implement a static array and perform the following operations: insertion, deletion, and traversal.
    - Implement a dynamic array with similar operations and analyze the performance differences compared to the static array.
2. **String Operations:**
    - Implement basic string operations such as concatenation, substring, and comparison.
    - Write a program to count the frequency of each character in a given string.
3. **Algorithm Analysis:**
    - Analyze the time and space complexity of the implemented data structures and operations.
    - Solve and analyze recurrence relations for basic algorithms such as binary search and merge sort.

Instructions

1. **Static and Dynamic Array Implementation:**
    - Create a class StaticArray with methods for insertion, deletion, and traversal.
    - Create a class DynamicArray with methods for insertion, deletion, and traversal.
    - Provide a comparative analysis of time complexity for both implementations.
2. **String Operations Implementation:**
    - Create a class StringOperations with methods for concatenation, substring, and comparison.
    - Write a function characterFrequency that takes a string as input and returns the frequency of each character.
3. **Algorithm Analysis:**
    - Write a report analyzing the time and space complexity of the array and string operations using Big O notation.
    - Solve given recurrence relations and include the solutions in the report.

Code:

1.Static Array

```
// Class representing a static array with basic operations
class Array {
    private int[] array;
    private int size;
    private int length;

    // Constructor to initialize the array with the given size
    public Array(int size) {
        this.size = size;
        this.array = new int[size];
        this.length = 0;
    }

    // Method to insert a value into the array
    public void insert(int value) {

        if (length < size) {
            array[length] = value;
            length++;
        } else {
            System.out.println("Array is FULL");
        }
    }

    // Method to delete the first occurrence of a value from the array
    public void delete(int value) {

        for (int i = 0; i < length; i++) {
            if (array[i] == value) {

                for (int j = i; j < length - 1; j++) {
                    array[j] = array[j + 1];
                }
                length--;
                break;
            }
        }
    }

    // Method to print the elements of the array
    public void traverse() {
        System.out.print("[");
        for (int i = 0; i < length; i++) {
            System.out.print(array[i]);
            if (i < length - 1) {
                System.out.print(", ");
            }
```

```
        }
        System.out.println("]");
    }
}

// Main class to demonstrate the Array class operations
public class StaticArray {
    public static void main(String[] args) {
        // Static Array test case
        Array staticArr = new Array(5);
        staticArr.insert(1);
        staticArr.insert(2);
        staticArr.delete(1);
        System.out.println("Expected Output: [2]");
        System.out.print("Actual Output: ");
        staticArr.traverse();
    }
}
```

2. Dynamic Array

```
// Class for Dynamic Array Implementation in Java
class Array {
    private int[] arr; // Array to store elements
    private int size; // Current number of elements
    private int capacity; // Current capacity of the array

    // Constructor to initialize the dynamic array
    public Array() {
        capacity = 2;
        arr = new int[capacity];
        size = 0;
    }

    // Method to resize the array when capacity is reached
    private void resize() {
        capacity *= 2;
        int[] newArr = new int[capacity];
        for (int i = 0; i < size; i++) {
            newArr[i] = arr[i];
        }
        arr = newArr;
    }
```

```java
    // Method to insert an element into the array
    public void insert(int element) {
        if (size == capacity) {
            resize();
        }
        arr[size++] = element;
    }

    // Method to remove the first occurrence of an element by value
    public void delete(int value) {
        for (int i = 0; i < size; i++) {
            if (arr[i] == value) {
                for (int j = i; j < size - 1; j++) {
                    arr[j] = arr[j + 1];
                }
                size--;
                return; // Exit after deleting the first occurrence
            }
        }
        System.out.println("Value not found.");
    }

    // Method to traverse and print the array
    public void traverse() {
        System.out.print("[");
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i]);
            if (i < size - 1)
                System.out.print(", ");
        }
        System.out.println("]");
    }
}

// Main class to test the DynamicArray class
public class DynamicArray {
    public static void main(String[] args) {
        // Dynamic Array test case
        Array dynamicArray = new Array();

        // Performing operations
        dynamicArray.insert(1);
        dynamicArray.insert(2);
        dynamicArray.delete(1);
        System.out.println("Expected Output: [2]");
        System.out.print("Actual Output: ");
        dynamicArray.traverse();
    }
}
```

3. String Operations

```
import java.util.HashMap;

class Str {
    public String concatenate(String str1, String str2) {
            return (str1 + str2);
    }

    // Method to perform substring operation
    public String substring(String str, int start, int end) {
        return str.substring(start + 1, end + 1);
    }

    // Method to compare two strings
    public boolean compareStrings(String str1, String str2) {
        return str1.equals(str2);
    }

    // Method to calculate character frequency
    public HashMap<Character, Integer> characterFrequency(String str) {
        HashMap<Character, Integer> frequencyMap = new HashMap<>();
        for (char ch : str.toCharArray()) {
            frequencyMap.put(ch, frequencyMap.getOrDefault(ch, 0) + 1);
        }
        return frequencyMap;
    }
}

public class StringOperations {
    public static void main(String[] args) {
        Str strOps = new Str();

        System.out.println("Concatenation: " + strOps.concatenate("hello",
"world"));
        System.out.println("Substring: " + strOps.substring("hello", 2, 4));
        System.out.println("Comparison: " + strOps.compareStrings("hello",
"world"));
        System.out.println("Character Frequency: " +
strOps.characterFrequency("hello"));
    }
}
```

Report: Implementing and Analysing Basic Data Structures

Objective

This assignment focuses on implementing fundamental data structures, specifically static and dynamic arrays, and performing essential string operations. The report delves into the comparison between static and dynamic arrays, examining their performance through the lens of asymptotic analysis. The evaluation employs Big O notation to assess both time and space complexities comprehensively. Furthermore, the assignment includes an analysis of how to approach and solve recurrence relations in algorithmic contexts.

Problem Description

1. Static and Dynamic Arrays: Create both types of arrays and perform operations like adding (insertion), removing (deletion), and displaying (traversal) elements. The performance of each operation is analyzed using time complexity to understand how they compare.

2. String Operations: Build functions to join two strings, extract a part of a string (substring), and compare two strings. Additionally, a function is created to count how often each character appears in a string.

3.Algorithm Analysis: Study and measure the performance of the array operations by calculating how much time and memory they use. Also, solve and explain recurrence relations for common algorithms like binary search and merge sort.Implementation

Implementation

1. Static and Dynamic Array Operations

Static Array

A **Static Array** has a fixed size, so we define a class StaticArray that allows:

- **Insertion** at a specified index
- **Deletion** of an element by shifting remaining elements
- **Traversal** to output the current elements in the array

Dynamic Array

A **Dynamic Array** can resize itself when capacity is exceeded. The class DynamicArray includes:

- **Insertion** that doubles the array size if capacity is reached
- **Deletion** of elements, with a shift for remaining elements
- **Traversal** to display elements in the array

2. String Operations

The StringOperations class provides:

- **Concatenation** of two strings
- **Substring** extraction
- **Comparison** to check if two strings are identical
- **Character Frequency** analysis to count occurrences of each character

Test Cases and Results

Operation	Test Input	Expected Output	Actual Output
Static Array Operations	insert(1), insert(2), delete(1), traverse()	[2]	[2]
Dynamic Array Operations	insert(1), insert(2), delete(1), traverse()	[2]	[2]
String Operations	"hello", "world"	concatenation: "helloworld", substring: "lo", comparison: False	concatenation: "helloworld", substring: "lo", comparison: False
Character Frequency	"hello"	{'h': 1, 'e': 1, 'l': 2, 'o': 1}	{'h': 1, 'e': 1, 'l': 2, 'o': 1}

3. Algorithm Analysis

- **Time Complexity**: For each data structure operation, Big O notation was used to evaluate time complexity.
- **Space Complexity**: We analyzed memory requirements for both static and dynamic arrays.
- **Recurrence Relations**: Basic algorithms like binary search and merge sort were analyzed, with recurrence relations solved using the Master theorem.

Static Array:

Time Complexity

1. Insert Method insert(int value):
    - Description: Takes an integer value and adds it to the end of the array.
    - Best Case: O(1)
    - Worst Case: O(n)
2. Delete Method delete(int value):
    - Description: Takes an integer value and removes its first occurrence from the array.
    - Best Case: O(1)

- Worst Case: O(n)

3. Traversal traverse():
    - Description: Traverses through an array and return every element from the array.
    - Worst Case: O(n)

Dynamic Array:

Time Complexity

1. Insert Method insert(int value):
    - Description: Takes an integer value and adds it to the end of the array.If the array is full, it uses the resize() method to increase the size of the array, hence making it dynamic.
    - Best Case: O(1)
    - Worst Case: O(n)
2. Delete Method delete(int value):
    - Description: Takes an integer value and removes its first occurrence from the array.
    - Best Case: O(1)
    - Worst Case: O(n)
3. Traversal traverse():
    - Description: Traverses through an array and return every element from the array.
    - Worst Case: O(n)

String Operations

Concatenate:

Time Complexity: `O(n + m)` due to copying both strings to a new string.

Space Complexity: `O(n + m)` for creating a new string that combines `str1` and `str2`.

Substring:

Time Complexity: `O(k)` for copying `k` characters from the original string.

Space Complexity: `O(k)` for storing the new substring.

Compare Strings:

Time Complexity: `O(min(n, m))`, as the comparison stops at the first non-matching character or at the end of the shorter string.

Space Complexity: `O(1)` because no additional data structures are used.

Character Frequency:

Time Complexity: `O(n)` because it iterates through the string once, updating the `HashMap` for each character.

Space Complexity: `O(n)` in the worst case, as the `HashMap` may need to store up to `n` unique characters.

Time and Space Complexity Analysis of String Operations

Operation	Time Complexity (Big O)	Space Complexity (Big O)	Reason for Time Complexity	Reason for Space Complexity
Concatenate	O(n + m)	O(n + m)	Concatenation requires copying both strings str1 and str2, where n and m are their lengths.	A new string of size n + m is created to store the result.
Substring	O(k)	O(k)	Extracting a substring of length k requires copying k characters.	Space is required to store the new substring of length k.
Compare Strings	O(min(n, m))	O(1)	Comparison stops at the first differing character or at the end of the shorter string, where n and m are lengths.	Only constant space is used for comparison.
Character Frequency	O(n)	O(n)	Iterating through the string str of length n and updating a HashMap takes linear time.	The space required for the HashMap can be up to O(n) if all characters are unique.

Complexity Analysis

Operation	Time Complexity (Big O)	Space Complexity (Big O)
Static Array Insertion	O(n)	O(1)
Static Array Deletion	O(n)	O(1)
Dynamic Array Insertion	O(1) amortized	O(n)
Dynamic Array Deletion	O(n)	O(1)
String Concatenation	O(m + n)	O(m + n)
Character Frequency	O(n)	O(1)

Recurrence Relations

1. **Binary Search**: T(n)=T(n/2)+O(1)T(n) = T(n/2) + O(1)T(n)=T(n/2)+O(1)
    - Solution: O(log⁡n)O(\log n)O(logn)
2. **Merge Sort**: T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n)
    - Solution: O(nlog⁡n)O(n \log n)O(nlogn)

Conclusion

This assignment reinforced the importance of data structures and their efficient implementations:

- **Static Arrays** are efficient for fixed-size data, while **Dynamic Arrays** allow flexibility but require resizing.
- **String Operations** were effectively implemented, demonstrating string manipulation's importance in data handling.
- **Algorithm Analysis** provided insight into understanding algorithm efficiency, essential for optimizing performance.
