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

**Instructions**

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

**CODE:**

#include <iostream>

#include <string>

#include <unordered\_map>

#define MAX\_SIZE 100

// Class for Static Array Implementation

class StaticArray {

int arr[MAX\_SIZE]; // Static array with fixed size

int size;

public:

StaticArray() : size(0) {}

// Insert element at specified index

void insert(int element, int index) {

if (size >= MAX\_SIZE || index < 0 || index > size) {

std::cout << "Insertion not possible." << std::endl;

return;

}

for (int i = size; i > index; --i) {

arr[i] = arr[i - 1];

}

arr[index] = element;

size++;

}

// Remove element at specified index

void remove(int index) {

if (index < 0 || index >= size) {

std::cout << "Deletion not possible." << std::endl;

return;

}

for (int i = index; i < size - 1; ++i) {

arr[i] = arr[i + 1];

}

size--;

}

// Traverse the static array

void traverse() const {

std::cout << "[";

for (int i = 0; i < size; ++i) {

std::cout << arr[i];

if (i < size - 1) std::cout << ", ";

}

std::cout << "]" << std::endl;

}

};

// Class for Dynamic Array Implementation

class DynamicArray {

int \*arr; // Pointer to the dynamic array

int size; // Number of elements

int capacity; // Current capacity of the array

// Resizes the array when capacity is reached

void resize() {

capacity \*= 2;

int \*newArr = new int[capacity];

for (int i = 0; i < size; ++i) {

newArr[i] = arr[i];

}

delete[] arr;

arr = newArr;

}

public:

DynamicArray() : size(0), capacity(2) {

arr = new int[capacity];

}

~DynamicArray() {

delete[] arr;

}

// Insert element in the dynamic array

void insert(int element) {

if (size == capacity) {

resize();

}

arr[size++] = element;

}

// Remove element at specified index

void remove(int index) {

if (index < 0 || index >= size) {

std::cout << "Deletion not possible." << std::endl;

return;

}

for (int i = index; i < size - 1; ++i) {

arr[i] = arr[i + 1];

}

size--;

}

// Traverse the dynamic array

void traverse() const {

std::cout << "[";

for (int i = 0; i < size; ++i) {

std::cout << arr[i];

if (i < size - 1) std::cout << ", ";

}

std::cout << "]" << std::endl;

}

};

// Class for String Operations

class StringOperations {

public:

// Concatenate two strings

std::string concatenate(const std::string &str1, const std::string &str2) {

return str1 + str2;

}

// Extract a substring from a given string

std::string substring(const std::string &str, int start, int length) {

if (start < 0 || start >= str.size() || start + length > str.size()) {

return "";

}

return str.substr(start, length);

}

// Compare two strings lexicographically

bool compare(const std::string &str1, const std::string &str2) {

return str1 == str2;

}

// Count frequency of each character in a string

std::unordered\_map<char, int> characterFrequency(const std::string &str) {

std::unordered\_map<char, int> frequency;

for (char c : str) {

frequency[c]++;

}

return frequency;

}

};

// Function to demonstrate the StringOperations class

void demonstrateStringOperations() {

StringOperations so;

std::string str1 = "hello";

std::string str2 = "world";

// Concatenate strings

std::cout << "Concatenation: " << so.concatenate(str1, str2) << std::endl;

// Extract a substring

std::cout << "Substring: " << so.substring(str1, 3, 2) << std::endl;

// Compare two strings

std::cout << "Comparison: " << (so.compare(str1, str2) ? "True" : "False") << std::endl;

// Calculate and print character frequency

std::string testString = "hello";

auto freq = so.characterFrequency(testString);

std::cout << "Character Frequencies:" << std::endl;

for (const auto &pair : freq) {

std::cout << pair.first << ": " << pair.second << std::endl;

}

}

// Main function demonstrating StaticArray, DynamicArray, and StringOperations

int main() {

// Static Array test case

std::cout << "Static Array Operations:" << std::endl;

StaticArray staticArr;

staticArr.insert(1, 0);

staticArr.insert(2, 1);

staticArr.remove(0);

std::cout << "Expected Output: [2]" << std::endl;

std::cout << "Desired Output: ";

staticArr.traverse();

// Dynamic Array test case

std::cout << "\nDynamic Array Operations:" << std::endl;

DynamicArray dynamicArr;

dynamicArr.insert(1);

dynamicArr.insert(2);

dynamicArr.remove(0);

std::cout << "Expected Output: [2]" << std::endl;

std::cout << "Desired Output: ";

dynamicArr.traverse();

// String Operations test case

std::cout << "\nString Operations:" << std::endl;

demonstrateStringOperations();

return 0;

}

**Report: Implementing and Analysing Basic Data Structures**

**Objective**

The objective of this assignment is to implement fundamental data structures such as static and dynamic arrays, along with basic string operations. This report explores the differences between static and dynamic arrays, analyzes their performance using asymptotic notations, and applies Big O notation to evaluate time and space complexities. Additionally, the assignment demonstrates how to analyze recurrence relations for algorithms.

**Problem Description**

1. **Static and Dynamic Arrays**: Implement both array types and perform insertion, deletion, and traversal operations. The time complexity of each operation is analyzed and compared.
2. **String Operations**: Implement functions for string concatenation, substring extraction, and string comparison. Additionally, a function to count character frequencies in a string is developed.
3. **Algorithm Analysis**: Evaluate the time and space complexity of the implemented data structures and solve recurrence relations for algorithms such as binary search and merge sort.

**Implementation**

**1. Static and Dynamic Array Operations**

Static Array

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

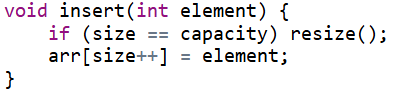
* **Insertion** at a specified index
* **Deletion** of an element by shifting remaining elements
* **Traversal** to output the current elements in the array

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

**Code Excerpt**:

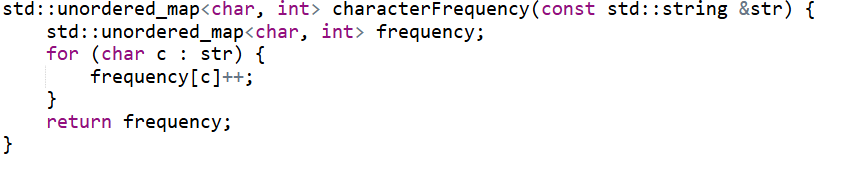


**2. String Operations**

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

**Code Excerpt**:



**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.

**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} |

**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.

The study of asymptotic notations (Big O) allowed for a clearer understanding of the time and space costs of these operations, which will help in selecting appropriate data structures in larger-scale applications.