Stacks and Queues

A Stacks

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The last element added to the stack is the first one to be removed. Stacks are often used in scenarios where you need to keep track of function calls, undo operations, or parsing expressions.

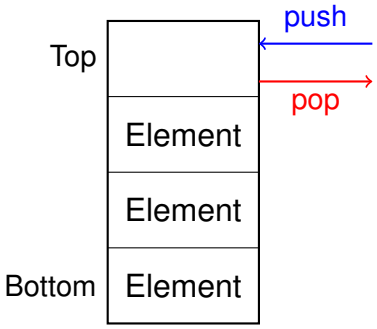

Figure 1: A stack

Figure 1 illustrates a stack with four elements. The top element can be removed using the pop operation, while new elements can be added using the push operation.

A.1 Operations

The main operations on a stack are:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.
- **Peek/Top**: Retrieve the top element without removing it.
- **IsEmpty**: Check if the stack is empty.
- **Size**: Get the number of elements in the stack.

A.2 Implementation

Stacks can be implemented using arrays or linked lists. The array-based implementation has a fixed size, while the linked list implementation can grow dynamically.

Example: A Stack Implementation in C++

```cpp
#include <iostream>
#include <vector>

class Stack : private std::vector<int> {
public:
    void push(int value) {
        this->push_back(value);
    }
    void pop() {
        if (!this->empty()) {
            this->pop_back();
        }
    }
    int top() const {
        if (!this->empty()) {
            return this->back();
        }
        throw std::out_of_range("Stack is empty");
    }
    bool isEmpty() const {
        return this->empty();
    }
    int size() const {
        return static_cast<int>(this->size());
    }
};
```

In this example, we define a simple stack class using C++'s 'std::vector' to store the elements. The stack supports basic operations like push, pop, and checking if it is empty. The base class 'std::vector' is privately inherited to encapsulate the stack's data structure and prevent direct access to the vector's methods. In this way, we ensure that the stack behaves as a proper abstract data type.

B Queues

A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element added to the queue is the first one to be removed. Queues are commonly used in scenarios like scheduling tasks, managing requests, or handling asynchronous data.

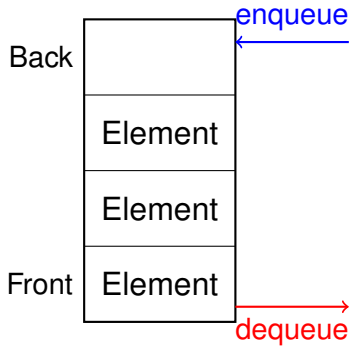

Figure 2: A queue

Figure 2 illustrates a queue with four elements. The front element can be removed using the dequeue operation, while new elements can be added using the enqueue operation.

B.1 Operations

The main operations on a queue are:

- **Enqueue**: Add an element to the back of the queue.
- **Dequeue**: Remove the front element from the queue.
- **Front/Peek**: Retrieve the front element without removing it.
- **IsEmpty**: Check if the queue is empty.
- **Size**: Get the number of elements in the queue.
- **Back**: Retrieve the last element in the queue.

B.2 Implementation

Queues can also be implemented using arrays or linked lists. The array-based implementation has a fixed size, while the linked list implementation can grow dynamically.

Example: A Queue Implementation in C++

```
#include <iostream>
#include <vector>
class Queue : private std::vector<int> {
public:
    void enqueue(int value) {
        this->push_back(value);
    }
    void dequeue() {
        if (!this->empty()) {
            this->erase(this->begin());
        }
    }
    int front() const {
        if (!this->empty()) {
            return this->at(0);
        }
        throw std::out_of_range("Queue is empty");
    }
    bool isEmpty() const {
        return this->empty();
    }
    int size() const {
        return static_cast<int>(this->size());
    }
};
```

In this example, we define a simple queue class using C++'s 'std::vector' to store the elements. The queue supports basic operations like enqueue, dequeue, and checking if it is empty. Similar to the stack implementation, we use private inheritance to encapsulate the queue's data structure and prevent direct access to the vector's methods.

B.3 Types of Queues

Queues can be categorized into several types based on their behavior and usage:

- **Simple Queue**: The basic FIFO queue as described above.

- **Circular Queue**: A queue where the last position is connected back to the first position, allowing for efficient use of space.
- **Priority Queue**: A queue where each element has a priority, and elements are dequeued based on their priority rather than their order of arrival.
- **Double-Ended Queue (Deque)**: A queue that allows insertion and deletion of elements from both ends.
- **Blocking Queue**: A queue that supports operations that block until the queue is not empty or not full, often used in concurrent programming.

C Stack vs Queue

Stacks and queues are both linear data structures, but they differ in their access patterns:

- **Access Pattern**: Stacks use LIFO (Last In First Out), while queues use FIFO (First In First Out).
- **Use Cases**: Stacks are often used for function call management, expression evaluation, and backtracking algorithms. Queues are used for task scheduling, request handling, and breadth-first search algorithms.
- **Operations**: Both structures support basic operations like adding and removing elements, but the order of these operations differs.
- **Implementation**: Both can be implemented using arrays or linked lists, but the choice of implementation may depend on the specific requirements of the application.

D Arrays vs Stacks and Queues

Arrays, stacks, and queues are all linear data structures, but they serve different purposes and have different characteristics:

- **Access Pattern**: Arrays allow random access to elements, while stacks and queues have restricted access patterns (LIFO for stacks and FIFO for queues).
- **Memory Management**: Arrays have a fixed size, while stacks and queues can grow dynamically (if implemented with linked lists) or have a maximum size (if implemented with arrays).
- **Use Cases**: Arrays are used for storing collections of elements that need to be accessed randomly, while stacks and queues are used for managing data in a specific order.
- **Performance**: Stacks and queues generally have O(1) time complexity for push/pop or enqueue/dequeue operations, while arrays may require O(n) time for insertion or deletion at arbitrary positions.
- **Implementation**: Arrays are a basic data structure, while stacks and queues are abstract data types that can be implemented using arrays or linked lists.