**Functional Stack Implementation**

**1-Introduction:**

This document explores the implementation of a functional stack in two programming languages: Haskell and C++.

Haskell is a purely functional programming language and C++ is an imperative language.

The purpose of this analysis is to evaluate the trade-offs between the two implementations based on:

1-Memory usage

2-Performance

3-Benefit of parallelization

A functional stack is a persistent data structure where every operation (like push or pop) creates a new version of the stack without modifying the existing version. This behavior aligns naturally with Haskell's immutability but poses challenges in C++ due to its imperative nature.

**2-Implementation Details:**

a)Haskell Implementation

The Haskell stack uses immutability by nature, with operations (push, pop, peek) returning new instances of the stack while retaining references to older ones.

The garbage collector automatically cleans unused references, reducing memory overhead.

Key Features:

-Persistent data structure using immutability.

-Lazy evaluation minimizes immediate computation but may delay memory usage.

C++ Implementation

The C++ stack uses **shared\_ptr** to mimic immutability. Each operation creates new stack nodes, ensuring references to previous versions remain valid.

Manual memory management is required, with **shared\_ptr** introducing overhead for reference counting.

Key Features:

-Simulates immutability in an imperative style.

-Immediate execution of operations.

**3-Comparaison:**

a)Memory Usage:

|  |  |  |
| --- | --- | --- |
| Aspect | Haskell | C++ |
| Allocation Model | Managed by the garbage collector | Explicit allocation using **shared\_ptr** |
| Overhead | Minimal, as unchanged data is shared | Higher due to reference counting overhead |
| Garbage Collection | Automatic cleanup of unused object | Requires explicit handling of memory |
| Persistence Handling | Efficient sharing of unchanged structure | Manual duplication of node references |

b)Performance:

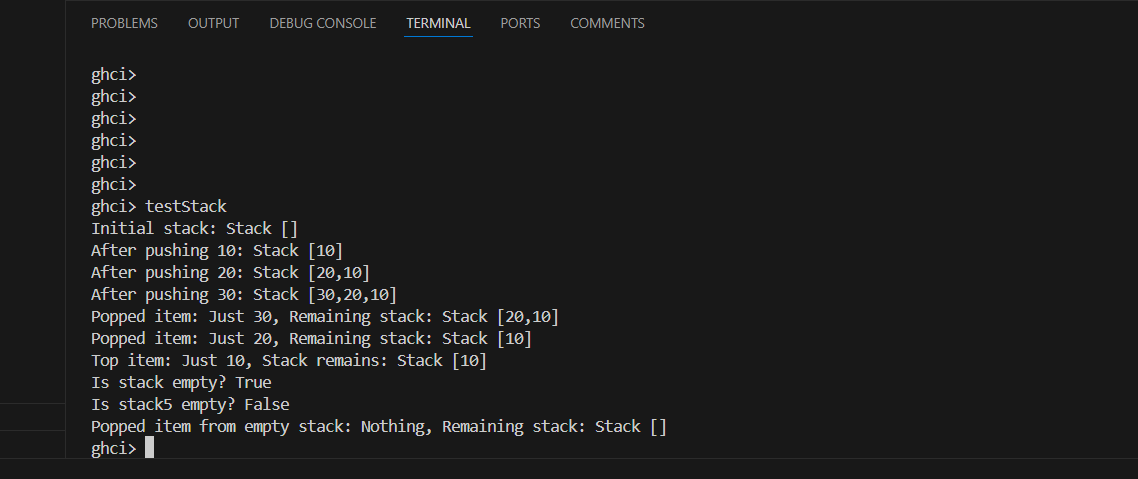
|  |  |  |
| --- | --- | --- |
| Aspect | Haskell | C++ |
| Execution Model | Lazy evaluation delays operations until needed | Eager execution processes operations immediately |
| Push Operation | O(1), creates a new head node | O(1), creates a new shared\_ptr node |
| Pop Operation | O(1), returns the next reference | O(1), adjusts pointers and reference counts |
| Runtime Overhead | Higher due to lazy evaluation | Lower, direct execution without abstractions |

c)Benefit of Parallelization:

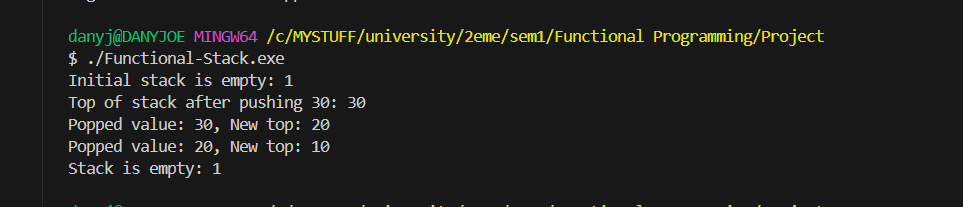
|  |  |  |
| --- | --- | --- |
| Aspect | Haskell | C++ |
| Immutability | Guarantees thread safety, as no data is mutated | Simulated immutability ensures partial thread safety |
| Concurrency Model | Easily parallelizable with no race conditions | Requires explicit locks or synchronization |
| Synchronization | No need for locks | Mutexes or atomic operations may be necessary |

**4-Output:**

Haskell:



C++:



**5-Observations:**

a)Memory Usage:

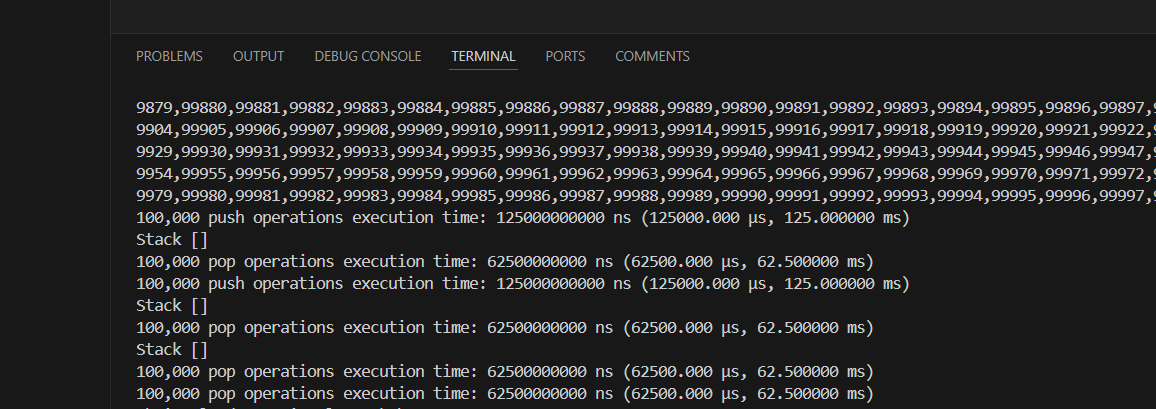
-Haskell's memory usage is more efficient for unused stack versions.

-C++ shows higher overhead due to shared\_ptr reference counting.

b)Performance:

|  |  |  |
| --- | --- | --- |
| Operation (100,000 operation) | Haskell time (ms) | C++ time (ms) |
| Push | 125 | 12.07 |
| Pop | 62.5 | 9.5135 |

Haskell:



C++:

A screen shot of a computer

Description automatically generated

NB: Those results depend from the device that is used to execute the code

**Parallelization**

* Haskell demonstrates faster multi-threaded execution due to immutability.
* C++ requires explicit synchronization mechanisms, increasing complexity.

**6-Conclusion:**

- **Haskell** excels in scenarios requiring **concurrency** and **thread safety**, making it ideal for highly parallel applications or functional programming paradigms.

- **C++** outperforms in raw execution speed and provides fine-grained control over memory and performance but requires more effort to simulate functional behavior.

Haskell is used for simplicity and parallelism.

C++ is used when performance and resrouce control are critical.