**Project idea 1: Linked list**

Elias Chalhoub (220438)

Georgio Mouawad (223064)

**Introduction:**

This documentation provides a detailed comparison of linked list implementations in **C++** and **Haskell**, highlighting their performance, time complexity, and how each language handles this data structure. Linked lists are a fundamental data structure where each element (node) points to the next, making it ideal for situations where dynamic memory allocation and efficient insertion/removal operations are required. This project compares the two languages in terms of implementation efficiency, memory management, and time complexity for common operations such as insertion, deletion, search, and traversal.

**Code explanation:**

**C++ code:**

In C++, the linked list is implemented using classes and pointers. Each **Node** contains an integer and a pointer to the next node, allowing dynamic creation and modification. The **LinkedList** class includes:

* **insertHead**: Adds a node at the beginning.
* **insertTail**: Adds a node at the end.
* **deleteNode**: Deletes a node by adjusting pointers.
* **search**: Traverses to find a node.
* **traverse**: Prints all nodes.

C++ allows manual memory management, providing flexibility but requiring explicit memory handling.

**Haskell code:**

In Haskell, the linked list uses recursion with the Maybe type to represent nodes. Operations include:

* **insertHead**: Recursively constructs a new list with the new node at the front.
* **search**: Recursively checks each node for the target.
* **traverseList**: Recursively prints the nodes.

Haskell’s immutability creates new lists for modifications, leading to higher memory usage and overhead compared to C++.

**How each language handles the linked list:**

**C++ approach:**

C++ is an imperative programming language, meaning the programmer has full control over the state and memory management of the program. This is reflected in the linked list implementation, where each node is manually allocated in memory using new and deallocated using delete. The mutable nature of C++ means that the list can be modified in place without creating new copies, which results in faster performance and lower memory usage compared to Haskell.

**Haskell approach:**

Haskell, being a functional programming language, handles the linked list differently. Immutability is a key feature of Haskell, which means that every modification to the list results in the creation of a new list rather than modifying the existing one. This leads to additional memory usage and slower performance. However, Haskell’s approach provides safety, as it prevents accidental changes to the list during operations. Haskell’s garbage collector also handles memory management automatically, preventing memory leaks but at the cost of some performance overhead.

**Performance comparison:**

1. The linked list operations are demonstrated by inserting 30,000 elements at the head of the list.
2. The program performs a search for the value 5000 and measures the time taken.
3. It then traverses the entire list and measures the time taken for that operation.

**Time complexity:**

Both the C++ and Haskell implementations exhibit **O(n)** time complexity for search and traversal operations. This means the time taken to search or traverse the list increases linearly with the number of nodes.

**Execution time:**

C++: Traversal time: 

Search time: 

Haskell: Traversal time:

 Search time:

**Memory usage:**

C++: Since C++ allows manual memory management, it can be more memory-efficient when handling large linked lists. However, improper memory management can lead to memory leaks.

Haskell: Due to its immutability, Haskell tends to use more memory. Each modification (even insertion) creates a new copy of the list, which leads to increased memory consumption. However, this approach simplifies memory management by relying on garbage collection, avoiding issues like memory leaks.

**Drawbacks:**

**C++**:

**Memory Management**: Manual memory management can be error-prone, and failure to deallocate memory can lead to memory leaks.

**Pointer Arithmetic**: The use of raw pointers makes the code more complex and harder to maintain, especially when dealing with large and complicated data structures.

**Less Safety**: Since C++ is less restrictive about memory access and manipulation, it can lead to undefined behavior if pointers are mishandled.

**Haskell**:

**Performance Overhead**: Immutability, although beneficial for maintaining code safety and predictability, introduces overhead due to the creation of new lists for each operation. This can lead to slower performance, especially when dealing with large datasets.

**Memory Overhead**: Haskell’s functional approach increases memory usage since each modification of the list involves creating a new version of the list instead of modifying it in place.

**Learning Curve**: Haskell’s functional style and immutable data structures can be difficult for developers accustomed to imperative programming paradigms like C++.

**Conclusion:**

In this project, we have compared the performance, time complexity, and handling of linked lists in **C++** and **Haskell**.

**C++** is better suited for performance-critical applications, where low-level control over memory and data structures is required. Its manual memory management provides flexibility and efficiency, but with the downside of potential errors such as memory leaks and pointer issues.

**Haskell**, on the other hand, prioritizes safety and simplicity with its immutable data structures and automatic memory management. While this approach simplifies code and avoids common memory management issues, it introduces overhead in both time and memory usage.

Both languages have their strengths and weaknesses when handling linked lists, with **C++** offering more control and better performance, and **Haskell** providing a more declarative, functional approach that emphasizes safety and ease of use.