Charbel Aoun – 220599

Functional programming project: Linked Lists

Introduction :

This report presents the implementation of **Linked Lists** using two programming paradigms: **functional programming** and **object-oriented programming**. The implementations are distributed across three files:

- **linked_list.cpp**: A **functional programming implementation** of a linked list in C++.
- **linked_list.hs**: A **functional programming implementation** of a linked list in Haskell.
- **linkedList.cpp**: An **object-oriented programming (OOP) implementation** of a linked list in C++.

The primary objective of this report is to conduct a **performance comparison** between these implementations in terms of **execution time** and **memory usage**. By analyzing and comparing these metrics, we aim to understand the causes behind the observed performance differences. This analysis will provide insight into how the choice of paradigm (functional vs object-oriented) and the choice of programming language (C++ vs Haskell) affect the efficiency of linked list operations such as insertion, deletion, traversal, and memory allocation.

The report also highlights the **advantages and trade-offs** of each paradigm and language. The comparison will focus on three key areas:

1. **Functional Programming in Haskell** vs **Functional Programming in C++** (linked_list.hs vs linked_list.cpp).
2. **Functional Programming in Haskell** vs **Object-Oriented Programming in C++** (linked_list.hs vs linkedList.cpp).

In order to **test** the performance in terms of **execution time**, we provided a file called **random_values.py**, when you run it, you will be asked to choose a file name and number of the values for testing.

Then, in the **linked_list.hs** and **linked_list.cpp,** there is a part called "**TESTING EXECUTION TIME"** where we implemented a main function that you can use to do the test scenarios. When you run the compiled file, you have to provide the command with a second argument, which is the .txt file generated by the python code above.

First, let's take a look at the structure of the functional programming linked list code to view its key components and design approach.

General Structure of a Linked List (Functional Programming in Haskell and C++):

Core Data Structures

- **Linked List (LL)**
    - **Node Structure**:
        - Represents nodes of the linked list with:
            - A **value** (data of the node).
            - A **next** reference pointing to the next node in the list.
        - **Immutable** in both Haskell and C++ implementations.
    - **Implemented in**:
        - **C++**: As a struct with std::shared_ptr for immutability.
        - **Haskell**: As an **algebraic data type** (data LL), where Node contains a value and a reference to the next node, or Empty represents the end of the list.

Core Operations on the Linked List

Creation

- **emptyList**:
    - Creates an empty linked list.
    - **(C++)** Implemented as nullptr (or Empty in functional implementations).
    - **(Haskell)** Implemented as Empty.
- **createNode**:
    - Creates a single node with a specified value.
    - **(C++)** Uses std::make_shared to create a new node.
    - **(Haskell)** Uses Node with a value and the next reference.

Basic List Manipulations

- **addFirst**:
    - Adds a new element at the beginning of the list.

- **(C++)** Creates a new node with the given value and points it to the existing list.
- **(Haskell)** Uses Node to prepend an element to the front of the list.

- **addLast**:
  - Adds a new element at the end of the list.
  - **(C++)** Recursively traverses to the end of the list, creating a new node.
  - **(Haskell)** Recursively creates a new list where the new node is appended to the end.
- **addAtIndex**:
  - Adds a new element at a specific index.
  - **(C++)** Recursively traverses to the specified index, then inserts the new node.
  - **(Haskell)** Recursively calls the addAtIndex function while decrementing the index until it reaches 0, where it inserts the node.
- **removeFirst**:
  - Removes the first element from the list.
  - **(C++)** Returns the next node of the first node.
  - **(Haskell)** Returns the next part of the list (i.e., skips the first node).
- **removeLast**:
  - Removes the last element from the list.
  - **(C++)** Recursively traverses the list and removes the last node.
  - **(Haskell)** Recursively traverses to the end, where the last node is removed.
- **removeAtIndex**:
  - Removes an element at a specific index.
  - **(C++)** Traverses to the index, then links the previous node to the node after the removed node.
  - **(Haskell)** Recursively decrements the index until it reaches 0, then removes the node at that position.

Basic Queries

- **size**:
  - Returns the total number of elements in the list.
  - **(C++)** Recursively counts the nodes.
  - **(Haskell)** Recursively counts the nodes and returns the total count.
- **sum**:

- Returns the sum of all elements in the list.
- **(C++)** Uses an accumulator and recursion to compute the sum.
- **(Haskell)** Uses an accumulator with a recursive call to compute the sum.

- **check**:
  - Checks if a specific value exists in the list.
  - **(C++)** Recursively checks each node to see if the value matches.
  - **(Haskell)** Uses pattern matching to compare the current node's value with the target value.
- **index**:
  - Returns the index of a specific value in the list (returns -1 if not found).
  - **(C++)** Uses recursion and a counter to find the first occurrence of the value.
  - **(Haskell)** Tracks the current index and recursively checks each node for the value.

Functional Operations

- **map**:
  - Applies a function to each element in the list and returns a new list.
  - **(C++)** Uses a recursive call to apply the function to each node's value and returns a new linked list.
  - **(Haskell)** Uses recursion to apply the function to each node, returning a new list.
- **filter**:
  - Filters the list based on a predicate function.
  - **(C++)** Uses recursion and a predicate function to construct a new list containing only elements that satisfy the condition.
  - **(Haskell)** Recursively applies the predicate to each node, keeping only the elements that satisfy it.

Aggregation and Higher-Order Operations

- **foldLeft**:
  - Folds the list from left to right, using an accumulator.
  - **(C++)** Uses recursion to accumulate the result from left to right.
  - **(Haskell)** Uses recursion to combine the current element with the accumulator from left to right.
- **foldRight**:
  - Folds the list from right to left, using an accumulator.

- **(C++)** Recursively calls the function to the end of the list, then processes it backward.
- **(Haskell)** Uses recursion to accumulate from the end of the list to the start.

List Reversal

- **reverse**:
  - Reverses the list by switching the direction of links between nodes.
  - **(C++)** Uses a recursive helper function with an accumulator to reverse the list.
  - **(Haskell)** Uses an accumulator to "prepend" nodes to a new list during recursion.

List Queries

- **maxi**:
  - Finds the maximum value in the list.
  - **(C++)** Uses recursion to track the maximum value seen so far.
  - **(Haskell)** Uses recursion to compare and store the maximum value at each step.
- **mini**:
  - Finds the minimum value in the list.
  - **(C++)** Uses recursion to track the minimum value seen so far.
  - **(Haskell)** Uses recursion to compare and store the minimum value at each step.

List Merging and Sorting

- **mergeTwoSorted**:
  - Merges two sorted linked lists into a single sorted list.
  - **(C++)** Recursively merges two lists by selecting the smaller head element.
  - **(Haskell)** Uses recursion to select the smaller of the two head elements and continue merging.
- **mergeSort**:
  - Sorts a linked list using the merge sort algorithm.
  - **(C++)** Recursively splits the list into halves, sorts each half, and merges them.
  - **(Haskell)** Recursively divides the list, sorts each half, and merges them.

Visualization and Display

- **display**:
    - Displays the contents of the linked list as a string.
    - **(C++)** Recursively builds a string representation of the list.
    - **(Haskell)** Uses recursion to construct a string with "->" between elements.

In each file, there is a part called TESTING FUNCTIONS where we wrote a main function in order to test the functions of these codes.

Test Scenario:

OBSERVATIONS:

During testing, it was observed that the **C++ functional programming implementation** of the linked list failed to handle operations involving **large numbers of elements** due to **deep recursion**, ultimately leading to a **stack overflow**. Here's an analysis of the issue:

Recursive Nature of Functional Implementation:

- Functional programming heavily relies on **recursion** instead of loops for operations like **addLast**, **size**, **sum**, and **removeAtIndex**.
- Each **recursive function** call allocates memory on the stack for the function's execution context. When the number of recursive calls grows too large (e.g., processing 10,000 or more elements), the program exhausts the stack space, causing a stack overflow.

Limitations of Tail-Call Optimization in C++ Compared to Functional Languages:

- Unlike some purely functional languages like Haskell or Lisp, which optimize recursion using techniques like **tail-call optimization (TCO)**, most C++ compilers do not guarantee such optimizations.
- Even tail-recursive functions in C++ still have limitations due to the language's imperative nature and how compilers handle recursion.
- The **proof** lies in our attempt to rewrite **addLast** and other functions in a tail-recursive manner. Despite this, stack overflow persisted when testing with large numbers of elements.

Behavior in Our Testing Scenarios:

Adding Large Numbers of Elements (addLast):

- When testing with large inputs (e.g., adding 100,000 elements), the recursive addLast function caused a deeply nested stack of function calls. The attempt to refactor it into a tail-recursive version did not resolve the issue, as stack overflow still occurred.

Iterative vs. Recursive Implementations:

- The object-oriented C++ implementation, which uses loops for operations, handled the same inputs efficiently because loops do not consume additional stack space for each iteration.

Functional Programming in C++:

- Functional programming is not a native paradigm in C++ and often does not benefit from the same optimizations available in functional-first languages. This makes it less suitable for handling very large datasets or deep recursion.

Haskell vs. C++ Tests:

- In contrast, when testing the same scenarios in Haskell, we did not encounter the same stack overflow issues, even with very large datasets. This difference arises from fundamental distinctions in how Haskell and C++ handle recursion and manage memory. This is because Haskell fully supports tail-call optimization (TCO) and employs lazy evaluation, allowing it to process only the necessary parts of large datasets and optimize recursive calls into iterative loops internally. These mechanisms enable Haskell to handle deep recursion efficiently without consuming excessive stack space.

Now we can compare the implementations. To compare, we have to look at two essential criteria: Execution time and Memory Consumption

Behavior in Our Testing Scenarios:

In order to compare the performance, we created linked lists from random values generated by the python file, and we increased the number of values and observed the results.

EXECUTION TIME:

We get it after executing the code under "**TEST EXECUTION TIME**"

- **Time :** (seconds)

Number of values	1 000	5 000	10 000	50 000	100 000	500 000	1 000 000
Haskell Functional	0.004	0.01	0.02	0.11	0.26	1.03	2.4
C++ Functional	0.16	1	Stack overflow	Stack overflow	Stack overflow	Stack overflow	Stack overflow
C++ Imperative	0.005	0.01	0.2	4	19	--	--

MEMORY CONSUMPTION:

To check memory consumption, run the file and use the Windows Task Manager. Navigate to Memory (Resource Monitor) to view the memory usage of all running processes.

- **Memory consumption :** (MB)

Number of values	1 000	5 000	10 000	50 000	100 000	500 000	1 000 000
Haskell Functional	1	2	5	30	55	250	430
C++ Functional	1	3	Stack overflow	Stack overflow	Stack overflow	Stack overflow	Stack overflow
C++ Imperative	0.4	0.7	1.4	3.2	4	10	12

- Let's compare between the OOP implementation in C++ and the functional implementation in Haskell. The complexity in time for the set of basic operations is clearly O(n) for the Functional Implementation. We can see that in the proportionality between the number of values and the execution time.
- Also, we conclude that in memory consumption, the imperative programming is way better.

Conclusion:

In this report, we analyzed the performance of linked list implementations using functional programming in C++, functional programming in Haskell, and object-oriented programming (OOP) in C++. The results revealed that the OOP implementation in C++ outperformed the others in terms of efficiency, memory usage, and scalability, especially for large datasets, as it relies on iterative loops instead of recursion. The Haskell functional implementation excelled at handling large datasets without stack overflow due to its support for tail-call optimization (TCO) and lazy evaluation, though it consumed more memory for large lists due to thunks (unevaluated expressions). The functional implementation in C++ faced significant challenges, including stack overflow during recursive calls, as C++ compilers do not guarantee TCO. Attempts to refactor the C++ functional code to use tail recursion did not prevent stack overflow, highlighting the limitations of recursion in C++. Ultimately, the OOP approach in C++ is recommended for practical applications, while Haskell remains a suitable option for functional programming, thanks to its efficient handling of recursion and large datasets.