

Functional Programming

Functional Data Structure – Haskell and C++

Binary Search Tree

Run the program :

In this directory, ./main/BinarySearchTree, you will find the implementation of a BinarySearchTree, in a functional programming paradigm.

Kindly find the following files :

- **BST.hs** : Implementation of Binary Tree in Haskell – Functional
- **BST.cpp** : Implementation of Binary Tree in C++ - Functional
- **BST_.cpp** : Implementation of Binary Tree in C++ - Imperative
- **generator.py** : A file that helps generating random values for testing

First, to run this part of the project, you have to run the file generator.py, when you will be asked to choose a file name and number of the values for testing.

Then, to run any of the basic files, **BST.hs**, **BST.cpp**, **BST_.cpp**, you have to compile them using GHC for the Haskell file and gcc for C++ files

When you want to run the compiled file, you have to provide the command with a second argument, which is the .txt file generated by the code python above.

The files cpp and hs contain a main part, which is basically a simple test for some implemented functions. You can change this part in **cpp** and **Haskell files**, and for Haskell specifically, you can ignore the main function and run any function or process from the **ghci** environment

Running the files with their **main** functions will show in the terminal a brief summary of what's happening in the program. Basically, we are taking numbers, randomly generated in python, and we are inserting them to the Tree, with some operations like count, add, remove, balance, depth ... At the end, the total execution time of the program is displayed.

Some problems :

- Because of the lazy initialization in Haskell, we were not able to detect the exact time of execution of a certain command. Even if the function is written and assigned to a variable, it is not executed if the program doesn't need its return value. And due to our purely functional implementation, it is impossible to measure the exact execution time of a command without directly using its value, except if we explicitly force the evaluation of the function by using its return value. For instance, we could achieve this by printing the result or passing it to another function to ensure its computation. However, we chose not to do so in order to maintain our focus on measuring the overall execution time of the program or command.
- In Haskell, we need many libraries and imports to be able to communicate with the system (like memory usage), and with the other files.
So, we decided to implement the python helper (generator.py) in order to manage data and file creation, and to generate the random values.
For the execution time, it can be calculated in C++ or Haskell using built in libraries
For memory usage, the solution to visualize the memory performance on Windows was to use Task Manager, and follow the execution of the related process.
- In C++, our goal was to optimize the execution time. So, we implemented the Data Structure in a purely functional way.
- Another challenge was ensuring that all operations, like balancing the tree or traversals, remained efficient. Recursive functions like `toList`, `addElement`, and `deleteElement` required careful management to avoid stack overflow for deeply nested trees.
- Since the goal of this project is not to manage input and outputs, we implemented some utility functions to facilitate data import and handling. For instance, the `readNumbersFromFile` function was designed to read numerical values from a text file into a vector, simplifying the process of feeding input into the data structures. Similarly, `measureTime` was introduced as a reusable utility to measure the execution time of operations, allowing us to evaluate the performance of various functions without modifying their core logic.

Structure of the functional program

Introduction :

Functional programming is a paradigm that emphasizes immutability, stateless functions, and the use of expressions over statements. In Binary Search Tree, functional programming was a natural fit for implementing immutable Tree, where operations like insertion, deletion, and modification do not alter the original structure but instead return a new version.

This immutability ensures that the original state is preserved, which reduces side effects and makes the program easier to reason about, debug, and test. Key principles we adhered to include:

1. **Immutability:** Each operation on the data structure (e.g., addition, removal) generates a new data structure without modifying the original.
2. **Pure Functions:** Functions depend only on their inputs and produce consistent outputs without side effects.
3. **Lazy Evaluation:** Although Haskell employs lazy evaluation by default, we handled strict evaluation manually where necessary to measure performance and manage system resources.

This design is reflected in both our C++ and Haskell implementations, where we crafted immutable linked lists, trees, and utility functions in a purely functional style.

General Structure :

Core Data Structures

- **Binary Search Tree (BST):**
 - **Node Structure:**
 - Represents nodes of the binary search tree with:
 - A value (data of the node).
 - Left and right child nodes.
 - Immutable in both languages.
 - **Implemented in:**
 - **C++:** As a struct with `std::shared_ptr` for immutability.
 - **Haskell:** As an algebraic data type (data BST).

Core Operations on the Tree

Creation

- **emptyTree:**
 - Creates an empty tree.
 - (C++) Implemented as returning nullptr.
 - (Haskell) Implemented as Empty.
- **singleTree:**
 - Creates a tree with a single node.
 - (C++) Utilizes createNode helper.
 - (Haskell) Returns Node with two Empty children.
- **addElement:**
 - Adds an element while maintaining the BST properties.
 - Recursively compares and inserts values into the left or right subtree.

Conversions

- **toList:** Converts the BST into a sorted list (in-order traversal).
- **fromSortedList:** Constructs a balanced BST from a sorted list (Start with the middle element as a root).
- **fromUnsortedList:** Creates a balanced BST from an unsorted list by sorting first.

Balancing

- **Balance:** Balances an unbalanced BST by converting it to a list and recreating it.

Basic Queries

- **sumTree and productTree:** Calculate the sum and product of all elements.
- **minElement and maxElement:** Find the smallest and largest elements
- **contains:** Checks whether a value exists in the BST.
- **count:** Counts the number of nodes.
- **depth:** Calculates the height of the tree.

Modifications

- **removeMin and removeMax:** Remove the smallest or largest element.
- **deleteElement:** Deletes a specific element while maintaining the BST structure.

Traversals

- **In-order (toList), Pre-order, and Post-order Traversals:** Standard tree traversal algorithms implemented recursively.

Higher-order Operations

- **mapTree:** Applies a function to each element and recreates a balanced BST.
- **filterTree:** Filters elements of the BST based on a predicate, returning a balanced tree.

Visualization

- **displayTree (C++):** Prints the BST in a structured format for debugging.
- **showTree (Haskell):** Uses a custom Show instance to visualize the tree.

Each modification in the structure creates a new structure while preserving immutability, ensuring that both the old and new structures remain accessible.

Binary Search Tree in imperative programming

In the file **BST_.cpp**, you will find a standard implementation of the Binary Search Tree, in an Object Oriented Programming style. This implementation follows the Von Neumann architecture, utilizing an Object-Oriented Programming style with mutable states. Most of the functions present in the purely functional implementation are also included here. This approach carefully manages memory and allows dynamic updates to the tree, though it may result in slightly higher execution times compared to the immutable, functional version.

Comparison :

Now we will move on comparing the algorithms, and discover the advantages and disadvantages of a functional implementation in the Binary Search Tree Data Structure.

To compare the implementations, we are interested in two critical aspects :

- The memory usage
- The execution time

Let's consider a process where we create a Binary Tree from random values generated in a file.

After creating the Tree, we add an element, then we remove an element, then we count the nodes, we balance the Tree, we find the Depth, the sum. A simple process that represents many daily uses Tree operations


Let's consider 3 different opponents :

- Functional Binary Search Tree in Haskell
- Functional Binary Search Tree in C++
- Object Oriented Binary Search Tree in C++

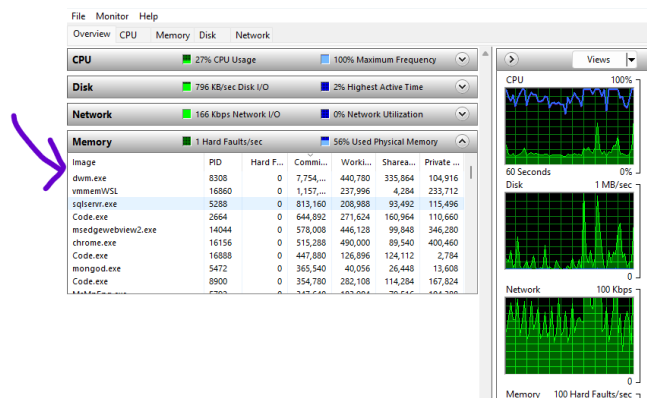
The execution time can be read from the command line after executing the corresponding program.

```

=== Final Results ===
Depth of Original Tree: 43
Depth of Balanced Tree: 17
Number of Nodes: 100000
Total execution time: 0.47554898262023926 seconds
  
```



However, concerning the memory usage, we have to execute the file. And during execution, on **Windows Task Manager -> Memory -> Resource Monitor -> Memory**, we can find the memory usage of all running processes.




So, to visualize Memory Usage, we have first to run the command to start executing the file, for example :

```

1\Programmation Fonctionnelle\0Projet\BinarySearchTree> ./hss.exe numbers.txt
  
```

The process hss.exe can be visualized, with its current memory usage

explorer.exe	19012	0	234,324	306,016	230,908	13,108
fontdrvhost.exe	18288	0	5,056	11,256	9,440	1,816
fontdrvhost.exe	1196	0	1,660	3,300	3,268	32
HerdHelper.exe	5336	0	10,380	33,040	33,000	40
hss.exe	18876	0	2,168,...	2,163,...	7,712	2,156,...
igfxCUIService.exe	1480	0	2,200	8,500	8,480	20
igfxEM.exe	20936	0	26,248	35,920	35,548	372
IntelCpHDCPSvc.exe	1688	0	1,532	6,912	6,888	24
IntelCpHDCPSvc.exe	1688	0	1,532	6,912	6,888	24



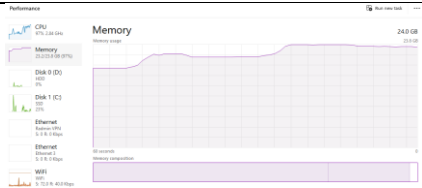
Here a summary of our testing process

Time : (In seconds)

Number of values	Haskell Functional	C++ Functional	C++ OOP
1 000	0.001	0.001	0.001
10 000	0.015	0.015	0.015
100 000	0.42	0.211	0.5
1 000 000	2.8	1.8	3.9
10 000 000	70	23	140
20 000 000	--	45	--
50 000 000	--	114	--

We can deduce here that the complexity of the processes (add, count, remove, balance, sum, etc.) in C++ functional programming is $O(n)$. This is evident because the execution time varies proportionally with the number of nodes.

Maximum Memory : (In MB)

Number of values	Haskell Functional	C++ Functional	C++ OOP
1 000	0.05	0.005	0
10 000	3	0.9	0.3
100 000	35	12	4
1 000 000	210	85	33
10 000 000	2 000	900	330
50 000 000	 Memory Full	4 800	1 600

In addition, the complexity in memory for the OOP C++ implementation is $O(n)$. Our implemented C++ functional Tree has also a complexity of $O(n)$ in memory, but consume 4 times more than the OOP implementation

In conclusion, we can confirm that functional programming is better than OOP imperative programming in terms of execution time, primarily because functional programming often focuses on immutability and avoids the overhead associated with mutable state management.

However, in terms of memory performance, OOP implementations tend to be more efficient. This is because:

- **Recursion Causes Stack Overflows:** In functional programming, recursion is a primary mechanism for iteration, which can lead to a deeper call stack. Each recursive call consumes stack memory, potentially leading to higher memory usage or even stack overflow in extreme cases.
- **Better Memory Management in OOP:** Imperative programming often allows fine-grained control over memory allocation and deallocation. For example, destructors and direct memory management techniques can free up unused memory more efficiently. Additionally, mutable data structures in OOP avoid the need to create new copies for every modification.
- **Functional Programming is Faster in Computation:** Functional programming avoids side effects and leverages immutability, which simplifies reasoning about code and allows for more aggressive optimizations by the compiler or runtime. Operations can be performed more quickly because data structures like trees are often implemented to reuse large portions of the original structure.

Finally, the main focus here was to create a functional structure in C++ which is efficient in memory management and also performant in execution time.

Haskell functional programming is fast, C++ OOP programming manages better the memory. In this C++ functional implementation, we got an optimal data structure, where the computation is very fast, and the memory management is not bad

Functional programming in Haskell demonstrated good computational speed due to its immutability and optimized tree operations, but the reliance on recursion led to higher memory usage. In contrast, C++ OOP implementations excelled in memory management by leveraging mutable states and direct memory handling, though at the cost of slightly higher execution times for larger datasets.

By implementing a functional approach in C++, we achieved a balanced data structure that combines efficient computation with reasonable memory performance, bridging the gap between functional purity and pragmatic memory usage. This makes the C++ functional implementation a robust choice for applications requiring both speed and resource optimization.