Purely Functional Data Structures

Project Group :

Rayen Bou Nassif 222353
Tony Abd Sater 221379
Mada Faddoul 221676

Data Structures :

1. Binary Tree Functions
2. Linked List Functions
3. HashMap Functions
4. Graph Functions
5. Stack Functions
6. Queue Functions

Introduction :

An imperative language uses a sequence of statements to determine how to reach a certain goal. These statements are said to change the state of the program as each one is executed in turn.

Functional languages: The functional programming paradigm was explicitly created to support a pure functional approach to problem solving. Functional programming is a form of declarative programming. We don't tell it step by step how to solve things, but we tell it what things are, their definition.

Advantages of Pure Functions: The primary reason to implement functional transformations as pure functions is that pure functions are composable: that is, self-contained and stateless. These characteristics bring a number of benefits, including the following: Increased readability and maintainability. This is because each function is designed to accomplish a specific task given its arguments. The function does not rely on any external state.

Easier reiterative development. Because the code is easier to refactor, changes to design are often easier to implement. For example, suppose you write a complicated transformation, and then realize that some code is repeated several times in the transformation. If you refactor through a pure method, you can call your pure method at will without worrying about side effects.

Easier testing and debugging. Because pure functions can more easily be tested in isolation, you can write test code that calls the pure function with typical values, valid edge cases, and invalid edge cases.

Functional or Imperative languages:

Object-oriented languages are good when you have a fixed set of operations on things and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods and the existing classes are left alone.

Functional languages are good when you have a fixed set of things and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types and the existing functions are left alone.

In the following we will implement different Data structures in C++ and haskell and compare them

1-Binary Tree Functions

1. **`insert`**: Adds a new node with a specified value to the tree.
2. **`search`**: Checks if a value exists in the tree.
3. **`inorder`**: Performs an inorder traversal (left, root, right) and returns a vector of node values.
4. **`preorder`**: Performs a preorder traversal (root, left, right) and returns a vector of node values.
5. **`postorder`**: Performs a postorder traversal (left, right, root) and returns a vector of node values.
6. **`findMax`**: Finds and returns the maximum value in the tree.
7. **`findMin`**: Finds and returns the minimum value in the tree.
8. **`deleteValue`**: Deletes a node with a specified value from the tree.
9. **`height`**: Calculates the height of the tree.
10. **`mapTree`**: Applies a function to every value in the tree.
11. **`foldTree`**: Reduces the tree to a single value using an accumulator.
12. **`clearTree`**: Deletes all nodes in the tree and deallocates memory.

1.1-Comparison of Haskell and C++ Implementations for Data Structures:

1. Time Complexity

- **Insertion**: Insertion takes **O(log n)** for balanced trees and **O(n)** for unbalanced trees in both C++ and Haskell. C++ links the new node in place, while Haskell creates a new path from the root.
- **Search**: Searching takes **O(log n)** for balanced trees and **O(n)** for unbalanced trees in both C++ and Haskell. C++ uses iteration, while Haskell uses recursion to traverse from root to target.
- **Traversal**: Inorder, Preorder, and Postorder traversal take **O(n)** in both C++ and Haskell, as every node must be visited. C++ uses iteration or recursion, while Haskell relies on recursion.
- **Deletion**: Deletion takes **O(log n)** for balanced trees and **O(n)** for unbalanced trees. C++ rebalances the tree in place, while Haskell creates a new path from the root to the deleted node.
- **Find Min/Max**: Finding the minimum or maximum takes **O(log n)** for balanced trees and **O(n)** for unbalanced trees. C++ and Haskell follow the leftmost or rightmost path using iteration (C++) or recursion (Haskell).
- **Height**: Calculating tree height takes **O(n)** in both C++ and Haskell, as every node must be visited. C++ uses iteration or recursion, while Haskell relies on recursion.
- **Functional Operations** (`map`, `fold`): Both take **O(n)** in C++ and Haskell, as every node is visited. C++ uses iteration or recursion, while Haskell uses recursion.

2. Space Complexity

- **Memory for Nodes**: Both C++ and Haskell require **O(n)** space to store nodes for a tree of size **n**.
- **Modifications**: C++ modifies nodes in place, requiring **O(1)** extra space. Haskell's immutability requires **O(log n)** space for balanced trees and **O(n)** for unbalanced trees due to the creation of new paths.
- **Traversal Space**: Recursive traversals require **O(h)** space for the call stack, where **h** is the height of the tree (**O(log n)** for balanced and **O(n)** for unbalanced trees). This is true for both C++ and Haskell.
- **Memory Efficiency**: C++ is more memory-efficient since nodes are updated in place, requiring **O(1)** space for modifications. Haskell's immutability creates new versions of the tree, leading to higher space usage of **O(log n)** for balanced trees.

2-Linked List Functions

1. **`addFirst(int x)`**: Adds a new element at the beginning of the linked list.
2. **`addLast(int x)`**: Adds a new element at the end of the linked list.
3. **`insertAt(int x, int pos)`**: Inserts an element at a specified position.
4. **`updateAt(int index, int value)`**: Updates the value of an element at a specified position.
5. **`elementAt(int index)`**: Retrieves the value of an element at a specific position.
6. **`search(int value)`**: Checks if an element exists in the linked list.

7. **`concat(LinkedList& other)`**: Concatenates two linked lists.
8. **`map(void (*f)(int&))`**: Applies a function to each element of the linked list.
9. **`filter(bool (*p)(int))`**: Filters the elements of the list based on a predicate.
10. **`fold(int (*f)(int, int), int acc)`**: Reduces the list to a single value using a binary function.
11. **`zip(LinkedList& other)`**: Combines two linked lists into a single list of interleaved values.
12. **`zipWith(int (*f)(int, int), LinkedList& other)`**: Combines two linked lists using a binary function.
13. **`deleteValue(int value)`**: Deletes the first occurrence of a specified value from the list.
14. **`clear()`**: Deletes all nodes and clears the linked list.
15. **`toList(int* arr, int size)`**: Converts the linked list to an array.
16. **`fromList(int* arr, int size)`**: Converts an array into a linked list.
17. **`size()`**: Returns the number of elements in the linked list.
18. **`reverse()`**: Reverses the linked list.
19. **`print()`**: Prints all elements in the linked list.

2.1-Comparison of Haskell and C++ Implementations for Data Structures:

1.Time Complexity

- **Access & Search** (`elementAt`, `deleteValue`, `search`): Both C++ and Haskell require **O(n)** time as linked lists require sequential traversal from the head. C++ uses iteration, while Haskell uses recursion.
- **Insertion** (`addFirst`, `addLast`, `insertAt`):
    - **addFirst**: **O(1)** for both since a new head node is linked directly.
    - **addLast**: **O(n)** since traversal to the end is required. C++ links in place, but Haskell creates a new list.
    - **insertAt**: **O(n)** for both, as traversal to the insertion point is needed. C++ modifies in place; Haskell creates a new path.
- **Update** (`updateAt`): **O(n)** for both. C++ updates the node's value in place, while Haskell creates a new path up to the updated node.
- **Concatenation**: **O(n)** for both. C++ links lists directly, while Haskell creates a new list.
- **Functional Operations** (`filter`, `map`, `fold`): **O(n)** for both as every node must be processed.

2.Space Complexity

- **Memory for Nodes**: Both C++ and Haskell require **O(n)** space to store nodes for a list of size **n**. Each node has a value and a pointer.

- **In-Place Modifications**: C++ modifies nodes in place, using **O(1)** extra space for updates like `updateAt` and `insertAt`.
- **Immutable Modifications**: Haskell's immutability requires creating new nodes for modified paths. Operations like `insertAt`, `updateAt`, and `deleteValue` require **O(n) space** in the worst case.

3-HashMap Functions

1. **`empty :: HashMap k v`**: Creates an empty hash map with no key-value pairs.
2. **`insert :: Eq k => k -> v -> HashMap k v -> HashMap k v`**: Adds or updates a key-value pair in the hash map.
3. **`delete :: Eq k => k -> HashMap k v -> HashMap k v`**: Removes a key-value pair by key.
4. **`lookup :: Eq k => k -> HashMap k v -> Maybe v`**: Retrieves the value associated with a key, or returns `Nothing` if the key is not found.
5. **`toList :: HashMap k v -> [(k, v)]`**: Converts the hash map to a list of key-value pairs.
6. **`fromList :: Eq k => [(k, v)] -> HashMap k v`**: Converts a list of key-value pairs into a hash map.
7. **`keys :: HashMap k v -> [k]`**: Returns a list of all keys in the hash map.
8. **`values :: HashMap k v -> [v]`**: Returns a list of all values in the hash map.
9. **`size :: HashMap k v -> Int`**: Returns the number of key-value pairs in the hash map.
10. **`isEmpty :: HashMap k v -> Bool`**: Checks if the hash map is empty.
11. **`update :: Eq k => k -> v -> HashMap k v -> HashMap k v`**: Updates the value associated with a key (equivalent to `insert`).
12. **`merge :: Eq k => HashMap k v -> HashMap k v -> HashMap k v`**: Merges two hash maps, retaining values from the first hash map when keys overlap.
13. **`mapValues :: (v -> v') -> HashMap k v -> HashMap k v'`**: Applies a function to all values in the hash map.

3.1-Comparison of Haskell and C++ Implementations for Data Structures:

1. Time Complexity

- **Add Vertex**: In C++, adding a vertex takes **O(1)** since inserting into an unordered map is **O(1)** on average. In Haskell, adding a vertex takes **O(1)** since it prepends to the list.

- **Add Edge**: In C++, adding an edge takes **O(1)** as the vertex is inserted into an unordered set. In Haskell, adding an edge takes **O(n)** since the list must be traversed to find the vertex, and the entire list is traversed using `map` to update it.
- **Get Vertices**: In C++, getting all vertices takes **O(n)** since it traverses the unordered map's keys. In Haskell, getting vertices takes **O(n)** since it extracts the first element from each tuple in the list.
- **Get Edges**: In C++, getting all edges takes **O(n + m)** since it traverses the adjacency list for each vertex. In Haskell, getting edges takes **O(n + m)** as it must traverse the list of vertices and the adjacency lists within each vertex.
- **Vertex Existence**: In C++, checking vertex existence takes **O(1)** since unordered map lookups are **O(1)**. In Haskell, vertex existence takes **O(n)** since it requires a linear search through the list.
- **Edge Existence**: In C++, checking edge existence takes **O(1)** since the lookup in an unordered set is **O(1)**. In Haskell, edge existence takes **O(n)** since it must traverse the adjacency list for the vertex.
- **Remove Vertex**: In C++, removing a vertex takes **O(n)** since the vertex is removed from the unordered map, and its references are removed from adjacency lists. In Haskell, removing a vertex takes **O(n)** as it filters the list to remove the vertex and updates the adjacency lists of other vertices.
- **Remove Edge**: In C++, removing an edge takes **O(1)** since the unordered set allows fast deletions. In Haskell, removing an edge takes **O(n)** as the list of adjacent vertices is filtered.
- **Neighbors**: In C++, getting neighbors takes **O(1)** since it retrieves the unordered set for the vertex. In Haskell, getting neighbors takes **O(n)** since it requires locating the vertex and returning its adjacency list.
- **In-Degree**: In C++, calculating in-degree takes **O(n + m)** since it checks every vertex's adjacency list. In Haskell, in-degree takes **O(n + m)** as it also scans each vertex's adjacency list to count edges pointing to the vertex.
- **Out-Degree**: In C++, calculating out-degree takes **O(1)** since it returns the size of the unordered set for the vertex. In Haskell, out-degree takes **O(n)** as it finds the vertex and counts its neighbors.
- **Check if Empty**: In C++, checking if the graph is empty takes **O(1)** since it checks if the unordered map is empty. In Haskell, checking if the graph is empty takes **O(1)** since it checks if the list is empty.
- **Graph Size (Vertices and Edges)**: In C++, calculating graph size takes **O(n + m)** since it counts the vertices and edges. In Haskell, calculating graph size takes **O(n + m)** as it sums the vertex count and adjacency list sizes.
- **Merge Graph**: In C++, merging two graphs takes **O(n' + m')** where **n'** is the number of vertices and **m'** is the number of edges in the second graph. In Haskell, merging two graphs takes **O(n' + m')** since it concatenates the lists of vertices and edges from both graphs.
- **Check if Connected**: In C++, checking if a graph is connected takes **O(n + m)** since it performs a BFS or DFS. In Haskell, it takes **O(n + m)** since it performs DFS recursively.
- **Cycle Detection**: In C++, cycle detection takes **O(n + m)** using DFS and a recursion stack. In Haskell, cycle detection takes **O(n^2)** since it may traverse the graph multiple times while checking for back edges.
- **Pathfinding (Single Path)**: In C++, finding a path takes **O(n + m)** as it performs BFS. In Haskell, finding a path also takes **O(n + m)** since it performs DFS.
- **Get All Paths**: In C++, finding all paths takes **O(k * (n + m))** where **k** is the number of paths. In Haskell, it also takes **O(k * (n + m))** since it explores all possible paths recursively.

2. Space Complexity

- **Memory for Vertices and Edges**: Both C++ and Haskell require **O(n + m) space** for **n** vertices and **m** edges. C++ uses unordered sets for adjacency lists, while Haskell uses lists for adjacency lists.
- **Add Vertex**: In C++, adding a vertex requires **O(1) space** for the new entry in the unordered map. In Haskell, adding a vertex takes **O(1) space** to prepend it to the list.
- **Add Edge**: In C++, adding an edge requires **O(1) space** to store the edge in the unordered set. In Haskell, adding an edge requires **O(n) space** since it creates a new list with the updated adjacency list.
- **Remove Vertex**: In C++, removing a vertex takes **O(1) space** since entries are deleted in place. In Haskell, removing a vertex requires **O(n) space** since it builds a new list without the vertex.
- **Remove Edge**: In C++, removing an edge takes **O(1) space** to modify the set in place. In Haskell, removing an edge takes **O(n) space** to construct a new adjacency list for the vertex.
- **Neighbors**: In C++, getting neighbors requires **O(d) space** where **d** is the degree of the vertex. In Haskell, neighbors requires **O(d) space** to return the list of adjacent vertices.
- **Graph Size (Vertices and Edges)**: In C++, calculating size requires **O(1) space** to store the vertex and edge counts. In Haskell, size requires **O(1) space** for the counts.
- **Pathfinding**: In C++, finding a path requires **O(n) space** for BFS queue and parent map. In Haskell, it also requires **O(n) space** for the visited set and recursion stack.
- **Cycle Detection**: In C++, cycle detection requires **O(h) space** where **h** is the height of the recursion stack. In Haskell, it requires **O(n) space** since each vertex in the recursion stack is stored in the recursive calls.
- **Get All Paths**: In C++, getting all paths takes **O(k * n) space** to store paths. In Haskell, it also requires **O(k * n) space** to store the paths.

4-Graph Functions

1. **`vertices :: Graph a -> [a]`**: Returns a list of all vertices in the graph.
2. **`edges :: Graph a -> [(a, a)]`**: Returns a list of all edges in the graph.
3. **`vertexExists :: Eq a => a -> Graph a -> Bool`**: Checks if a vertex exists in the graph.
4. **`edgeExists :: Eq a => a -> a -> Graph a -> Bool`**: Checks if an edge exists between two vertices.
5. **`neighbors :: Eq a => a -> Graph a -> [a]`**: Retrieves all adjacent vertices of a given vertex.
6. **`inDegree :: Eq a => a -> Graph a -> Int`**: Calculates the number of incoming edges for a vertex.
7. **`outDegree :: Eq a => a -> Graph a -> Int`**: Calculates the number of outgoing edges for a vertex.
8. **`findPath :: Eq a => a -> a -> Graph a -> Maybe [a]`**: Finds a path between two vertices, if one exists.
9. **`getAllPaths :: Eq a => a -> a -> Graph a -> [[a]]`**: Finds all possible paths between two vertices.

10. **`isConnected :: Eq a => Graph a -> Bool`**: Checks if the graph is fully connected.
11. **`hasCycle :: Eq a => Graph a -> Bool`**: Checks if the graph contains any cycles.
12. **`graphInfo :: Graph a -> (Int, Int)`**: Returns the number of vertices and edges in the graph.
13. **`isEmpty :: Graph a -> Bool`**: Checks if the graph has no vertices or edges.
14. **`mergeGraphs :: Eq a => Graph a -> Graph a -> Graph a`**: Merges two graphs by combining their vertices and edges.

4.1-Comparison of Haskell and C++ Implementations for Data Structures:

1. Time Complexity

- **Insert**: In C++, inserting a key-value pair takes **O(1)** on average due to hash-based lookup. In Haskell, inserting a key-value pair takes **O(n)** since it filters the list to remove the existing key before inserting the new pair.
- **Lookup**: In C++, lookup takes **O(1)** on average since it performs a hash lookup in the unordered_map. In Haskell, lookup takes **O(n)** as it searches the list for the key using linear search.
- **Delete**: In C++, deleting a key-value pair takes **O(1)** on average since unordered maps support efficient erasure. In Haskell, delete takes **O(n)** as it filters the list to exclude the key.
- **FromList**: In C++, converting a list of key-value pairs to a hashmap takes **O(n)** since each pair is inserted in constant time. In Haskell, fromList takes **O(n^2)** since each insertion takes **O(n)**, and there are **n** insertions.
- **ToList**: In C++, toList takes **O(n)** since it iterates over the unordered map and copies each key-value pair to a list. In Haskell, toList takes **O(1)** since the list is stored directly as the hashmap's internal representation.
- **Update**: In C++, updating the value associated with a key takes **O(1)** since it reuses the insert operation. In Haskell, update takes **O(n)** since it reuses the insert operation, requiring a filter and insertion.
- **Merge**: In C++, merging two hashmaps takes **O(n' + m')**, where **n'** is the number of keys in the first hashmap and **m'** is the number of keys in the second hashmap. In Haskell, merge takes **O(n' + m')** since the two lists are concatenated and filtered for duplicates.
- **Keys**: In C++, retrieving all keys takes **O(n)** as it iterates over all elements. In Haskell, keys take **O(n)** since it maps over the list to extract keys.
- **Values**: In C++, retrieving all values takes **O(n)** as it iterates over all elements. In Haskell, values take **O(n)** since it maps over the list to extract values.
- **Size**: In C++, getting the size of the hashmap takes **O(1)** as unordered maps track the size directly. In Haskell, size takes **O(1)** since the list length is stored as part of the hashmap's structure.
- **MapValues**: In C++, mapping a function over values takes **O(n)** as it iterates and modifies each value. In Haskell, mapValues also takes **O(n)** since it maps a function over the list of key-value pairs.

2. Space Complexity

- **Memory for Key-Value Pairs**: Both C++ and Haskell require **O(n) space** for **n** key-value pairs. C++ stores the data in a hash table, while Haskell stores key-value pairs as a list of tuples.
- **Insert**: In C++, inserting a new key-value pair requires **O(1) space** for the new key and value. In Haskell, insertion takes **O(1) space** to store the new key-value pair, but filtering the existing list requires **O(n) space**.
- **Delete**: In C++, deleting a key-value pair requires **O(1) space** since the entry is deleted in place. In Haskell, delete requires **O(n) space** since it creates a new list excluding the deleted key.
- **FromList**: In C++, converting a list to a hashmap takes **O(n) space** to store the elements. In Haskell, fromList requires **O(n) space** to store the new list of key-value pairs.
- **ToList**: In C++, toList requires **O(n) space** to store the new list of key-value pairs. In Haskell, toList requires **O(1) space** as it returns the existing list directly.
- **Update**: In C++, updating a key's value requires **O(1) space** as it modifies the value in place. In Haskell, update takes **O(n) space** since it creates a new list.
- **Merge**: In C++, merging two hashmaps takes **O(n' + m') space** to store the combined hashmap. In Haskell, merge takes **O(n' + m') space** to concatenate the two lists and remove duplicates.
- **Keys**: In C++, retrieving keys requires **O(n) space** to store the list of keys. In Haskell, keys require **O(n) space** to store the list of extracted keys.
- **Values**: In C++, retrieving values requires **O(n) space** to store the list of values. In Haskell, values require **O(n) space** to store the list of extracted values.
- **Size**: In C++, size requires **O(1) space** since it only returns an integer. In Haskell, size requires **O(1) space** since it only returns the length of the list.
- **MapValues**: In C++, mapping a function over values requires **O(1) extra space** since the values are updated in place. In Haskell, mapValues requires **O(n) space** since a new list of key-value pairs is created.

5-Stack Functions:

1. **`emptyStack`**: Creates an empty stack.
2. **`push`**: Adds an element to the top of the stack, returning a new stack.
3. **`pop`**: Removes and returns the top element along with the updated stack.
4. **`peek`**: Returns the top element of the stack without removing it.
5. **`isEmpty`**: Checks if the stack is empty.
6. **`size`**: Returns the number of elements in the stack.
7. **`reverseStack`**: Reverses the elements in the stack, returning a new stack.
8. **`toList`**: Converts the stack into a list.
9. **`fromList`**: Creates a stack from a list.
10. **`mapStack`**: Applies a function to each element in the stack, returning a new stack.

5.1-Comparison of Haskell and C++ Implementations for Data Structures:

1. Time Complexity

- **Push**: In C++, push takes **O(1)** in the average case due to amortized vector growth. In Haskell, push takes **O(1)** since adding to the front of a list requires no traversal.
- **Pop**: In C++, pop takes **O(1)** as it removes the last element of the vector. In Haskell, pop also takes **O(1)** since removing the head of a list is a constant-time operation.
- **Peek**: In C++, peek takes **O(1)** as it accesses the last element of the vector. In Haskell, peek also takes **O(1)** since it accesses the head of the list.
- **IsEmpty**: In C++, isEmpty takes **O(1)** since vector emptiness is checked via size comparison. In Haskell, isEmpty takes **O(1)** as it checks if the list is empty.
- **Size**: In C++, size takes **O(1)** since the vector tracks its size. In Haskell, size takes **O(n)** since it must traverse the list to count elements.
- **Reverse**: In C++, reverse takes **O(n)** as it reverses the vector in place. In Haskell, reverse also takes **O(n)** since it must construct a new reversed list.
- **ToList**: In C++, toList takes **O(n)** as it creates a copy of the vector. In Haskell, toList takes **O(1)** as the list can be returned directly.
- **FromList**: In C++, fromList takes **O(n)** as it copies elements into the vector. In Haskell, fromList takes **O(1)** since the list is directly converted into a stack.
- **Map**: In C++, map takes **O(n)** since it updates each element of the vector in place. In Haskell, map also takes **O(n)** as each element is mapped into a new list.

2. Space Complexity

- **Memory for Elements**: Both C++ and Haskell require **O(n) space** for **n** elements. C++ uses contiguous memory for the vector, while Haskell uses linked nodes for each list element.
- **Push/Pop/Peek**: C++ requires **O(1) space** as elements are modified in place, while Haskell requires **O(1) space** as the new list head points to the existing list.
- **Reverse**: C++ reverses the vector in place, requiring **O(1) extra space**. In Haskell, reverse requires **O(n) space** since a new list must be created.
- **ToList**: In C++, toList requires **O(n) space** to store the new list. In Haskell, toList requires **O(1) space** since it returns the existing list.
- **FromList**: In C++, fromList requires **O(n) space** to copy elements into the vector. In Haskell, fromList requires **O(1) space** as it directly references the existing list.
- **Map**: C++ modifies the vector in place, requiring **O(1) extra space**, while Haskell creates a new list, requiring **O(n) space** for the new list.

6-Queue Functions:

1. **`emptyQueue`**: Creates an empty queue.
2. **`enqueue`**: Adds an element to the end of the queue, returning a new queue.
3. **`dequeue`**: Removes and returns the front element along with the updated queue.
4. **`peek`**: Returns the front element of the queue without removing it.
5. **`isEmpty`**: Checks if the queue is empty.
6. **`size`**: Returns the number of elements in the queue.
7. **`reverseQueue`**: Reverses the elements in the queue, returning a new queue.

8. **`toList`**: Converts the queue into a list.
9. **`fromList`**: Creates a queue from a list.
10. **`mapQueue`**: Applies a function to each element in the queue, returning a new queue.

6.1-Comparison of Haskell and C++ Implementations for Data Structures:

1. Time Complexity

- **Enqueue**: In C++, enqueue takes **O(1)** on average as it appends to the end of the vector. In Haskell, enqueue takes **O(n)** since appending to the end of a list requires copying the entire list.
- **Dequeue**: In C++, dequeue takes **O(1)** to access the front and **O(n)** to shift elements after removal, making the total time **O(n)**. In Haskell, dequeue takes **O(1)** since it removes the head of the list directly.
- **Peek**: In C++, peek takes **O(1)** to access the front of the vector. In Haskell, peek also takes **O(1)** to access the head of the list.
- **IsEmpty**: In C++, isEmpty takes **O(1)** since it checks if the vector's size is zero. In Haskell, isEmpty also takes **O(1)** by checking if the list is empty.
- **Size**: In C++, size takes **O(1)** as the vector tracks its size. In Haskell, size takes **O(n)** as it must traverse the list to count the elements.
- **Reverse**: In C++, reverse takes **O(n)** as it swaps elements in the vector in place. In Haskell, reverse takes **O(n)** as it constructs a new reversed list.
- **ToList**: In C++, toList takes **O(n)** to copy elements from the vector. In Haskell, toList takes **O(1)** as it returns the existing list.
- **FromList**: In C++, fromList takes **O(n)** to copy elements from a list into the vector. In Haskell, fromList takes **O(1)** as it directly wraps the list into the queue.
- **Map**: In C++, map takes **O(n)** as each element in the vector is updated in place. In Haskell, map also takes **O(n)** as each element is mapped into a new list.

2. Space Complexity

- **Memory for Elements**: Both C++ and Haskell require **O(n) space** to store **n** elements. C++ uses contiguous memory for the vector, while Haskell uses linked nodes for each list element.
- **Enqueue**: In C++, enqueue requires **O(1) space** since the vector resizes only when capacity is reached. In Haskell, enqueue requires **O(n) space** since a new list must be created.
- **Dequeue**: In C++, dequeue requires **O(1) space** since elements are shifted in place. In Haskell, dequeue requires **O(1) space** since the list's head is dropped.
- **Reverse**: In C++, reverse requires **O(1) extra space** since it swaps elements in place. In Haskell, reverse requires **O(n) space** as a new list is created.
- **ToList**: In C++, toList requires **O(n) space** to create a new list. In Haskell, toList requires **O(1) space** since it returns the existing list.

- **FromList**: In C++, fromList requires **O(n) space** to copy elements into the vector. In Haskell, fromList requires **O(1) space** as it wraps the list in a queue.
- **Map**: C++ modifies elements in place, requiring **O(1) extra space**, while Haskell creates a new list, requiring **O(n) space**.

Conclusion

It can be concluded that a comparison between C++ and Haskell implementations of different data structures shows contrasting design philosophy and performance characteristics:

Performance Trade-offs: The ability to modify structures in C++ results in a lower time complexity for operations which alter the structure, which is why many C++ programmers prefer it. In Haskell, data can't be modified thus existing structures have to be copied, resulting in a lot of data being needed, but this also means that the integrity of the data is ensured and no threads would cause any issues.

Implementation Characteristics Memory Management: C++ allows a straightforward approach to memory by making changes in place, in contrast, Haskell ensures several guarantees with garbage collection and immutability. **Time Efficiency:** C++ is the definitive winner in update-dependent operations (in the majority of cases O(1)), unlike Haskell, which will take a much longer time for the same operations due to its immutable stance. **Space Usage:** In most cases C++ requires minor or less memory by requiring in place modifications, while in Haskell due to making a new copy for each modification, requires more memory.

Practical Implications C++ is well-suited for applications that are a little sensitive regarding time as well as regards memory usage, as they can be very quick. Haskell's way is good in a concurrent system as well in a case where the integrity of the data is more important than the state and reasoning of it. These differences point out that the decision on the usage of either C++ or Haskell depends on the purpose in which the program is going to be used for.