**Maria Sabbagh – 220451**

**Charbel Khaddaa -221283**

**Alessandro Matta – 221262**

**Data Structures**

**Imperative VS Functional**

We have implemented different operations on linked lists. We will use Python and Haskell to compare the time and memory usage for both implementations.

1. **Linked Lists**  
   The operations we performed:  
     
   1) Append.  
   2) Insert at the beginning.  
   3) Insert at the end.  
   4) Delete the first element.  
   5) Delete the last element.  
   6) Find an element by index.  
   7) Merge two linked lists.  
   8) Reverse the linked list.  
     
   We have used a linked list of 10,000 random elements.  
   1. **Python Results:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Execution** **Time** | **Memory Change** |
| Append | |  | | --- | | 1916.688 ms |  |  | | --- | |  | | 480,000 bytes |
| Insert at the beginning | 0.007 ms | |  | | --- | | 48 bytes |  |  | | --- | |  | |
| Insert at the end | 0.313 ms | 48 bytes |
| Delete the first element | 0.007 ms | 48 bytes |
| Delete the last element | 0.671 ms | 48 bytes |
| Find an element by index. | 0.021 ms | 0 bytes |
| Merge 2 linked lists. | 0.291 ms | 480,000 bytes |
| Reverse the linked list. | 0.886 ms | 1. bytes |

* 1. **Haskell Results**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Execution Time** | **Memory Change** |
| Append | |  | | --- | | 12027.6491 ms |  |  | | --- | |  | | 8,909,207,376 bytes |
| Insert at the beginning | 12.5194 ms | |  | | --- | | 0 bytes |  |  | | --- | |  | |
| Insert at the end | 12113.475 ms | 8,908,964,048 bytes |
| Delete the first element | 9.7436 ms | |  | | --- | | 0 bytes |  |  | | --- | |  | |
| Delete the last element | 6426.9388 ms | 2,390,135,440 bytes |
| Find an element by index. | 3.5622 ms | 0 bytes |
| Merge 2 linked lists. | 18.4227 ms | 8,909,604,096 bytes |
| Reverse the linked list. | 12101.1792 ms | 8,909,604,096 bytes |

* 1. **Time Execution**
     1. **Python** is faster for all operations except append due to iterative traversal. All other operations are executed in under 1 ms because of Python’s efficient dynamic memory.
     2. **Haskell** has an immutable structure, making operations take longer. Operations such as append, insertLast, and reverse take much more time. Simpler operations like insertFirst perform faster due to direct pointer manipulation.
  2. **Memory Usage**
     1. **Python:** Memory usage is minimal, and as we see, most operations use 48 bytes per node. Merge and append have greater memory usage due to the large number of elements.
     2. **Haskell:** Memory usage is higher due to Haskell’s lazy evaluation and immutability. Append, InsertLast, and Reverse consume over 8GB, while Merge and InsertFirst do not allocate memory because they use nodes efficiently.
  3. **Performance**
     1. **Python** has faster performance and lower memory overhead due to its mutable structure and dynamic memory.
     2. **Haskell’s** lazy evaluation makes list traversal operations take more time and use more memory.
  4. **Scalability**
     1. **Python's** performance scales well for most operations, except for the “append” operation.
     2. **Haskell** struggles with scalability for operations that require the use of memory.
  5. **Memory**
     1. **Python** Due to its garbage collection, operations such as DeleteLast or DeleteFirst don’t require large memory usage.
     2. **Haskell** allocates a large amount of memory for each operation. It reuses nodes efficiently in some operations, such as InsertFirst and InsertLast.

1. **Binary Trees**The operations we performed:  
     
   1) Insert.  
   2) In-order traversal.  
   3) Pre-order traversal.  
   4) Post-order traversal.  
   5) Remove an element.  
   6) Search an element.  
     
   We have used a binary tree of 1,000,000 random elements.  
   1. **Python Results**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Execution** **Time** | **Memory Change** |
| Insert | |  | | --- | | 6.17333 s |  |  | | --- | |  | | 48,000,000 bytes |
| In-order traversal | 0.36091 s | |  | | --- | | 2,338 bytes |  |  | | --- | |  | |
| Pre-order traversal | 0.37974 s | 986 bytes |
| Post-order traversal | 0.35290 s | 986 bytes |
| Search | 0.00002 s | - |
| Remove | 0.00002 s | - |

* 1. **Haskell Results**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Execution** **Time** | **Memory Change** |
| Insert | |  | | --- | | 7.96 s |  |  | | --- | |  | | 24,000,000 bytes |
| In-order traversal | 565 ms | |  | | --- | | 952 bytes |  |  | | --- | |  | |
| Pre-order traversal | 567.26 ms | 953 bytes |
| Post-order traversal | 722 ms | 952 bytes |
| Search | 14 microseconds | - |
| Remove | 273 ms | - |

* 1. **Conclusion**Haskell is good for memory, but has issues in performance when the amount of data gets bigger. Had we put 10,000,000 it would have given us stack overflow (using eager evaluation). This is due to the deep recursion, though we can fix it with deepseq, or by using tail recursion.  
       
     Python can handle large datasets better without stack overflows but has higher memory use. Performance can also be improved with better efficient-memory data structures.

1. **Array :**

We have implemented different operations on an array. Using Python and Haskell, we are going to compare the time and memory usage for both implementations.

The operations we performed:

1. Append.
2. Insert at the beginning.
3. Insert at the end.
4. Delete the first element.
5. Delete the last element.
6. Find an element by index.
7. Merge two arrays.
8. Reverse an array.
9. Prepend an array
10. Length of an array
11. Slice an array ( FP operations)
12. Map an array( FP operations)
13. Filter an array ( FP operations)
14. Fold an array ( FP operations)
15. Update
16. Get

We have used an array of 10000 random elements.

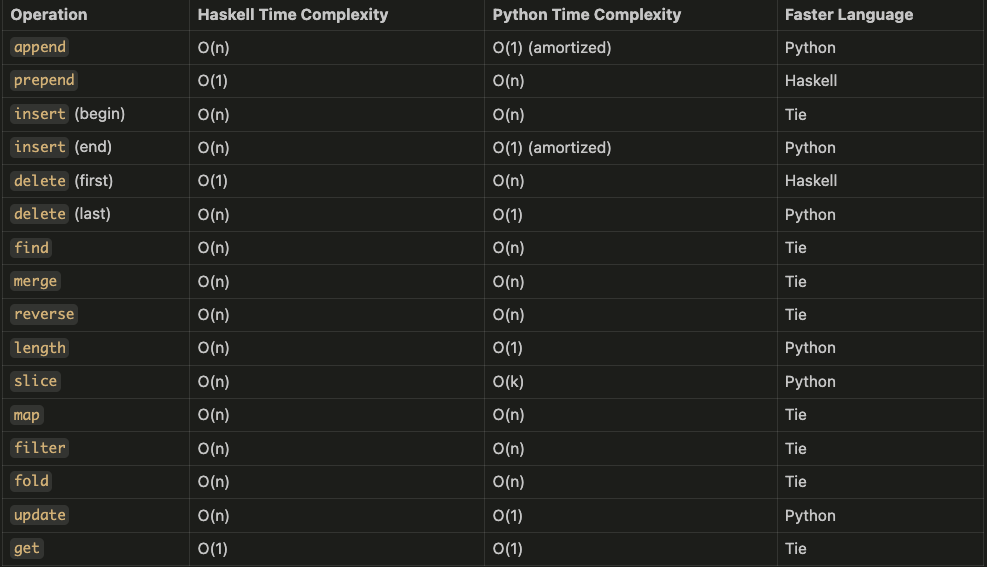
1. **Python Results:**

|  |  |  |
| --- | --- | --- |
| Operations | Execution time | Memory Change |
| Append | 95.08ms | 160776 bytes |
| Insert at the beginning | 104.43 ms | 160776 bytes |
| Insert at the end | 95.78 ms | 160336 bytes |
| Delete the first element | 95.90 ms | 160336 bytes |
| Delete the last element | 0.02 ms | 80080 bytes |
| Find an element by index | 105.94 ms | 161216 bytes |
| Merge two arrays | 0.01 ms | 28 bytes |
| Reverse an array | 0.03 ms | 160080 bytes |
| Prepend an array | 95.08 ms | 160776 bytes |
| Length of an array | 0.00 ms | 28 bytes |
| Slice an array | 0.00 ms | 28 bytes |
| Map an array | 0.52 ms | 42128 bytes |
| Filter an array | 0.01 ms | 8080 bytes |
| Fold an array | 94,42 ms | 160776 bytes |
| Update | 98.77 ms | 161216 bytes |
| Get | 0.05 ms | 160000 bytes |

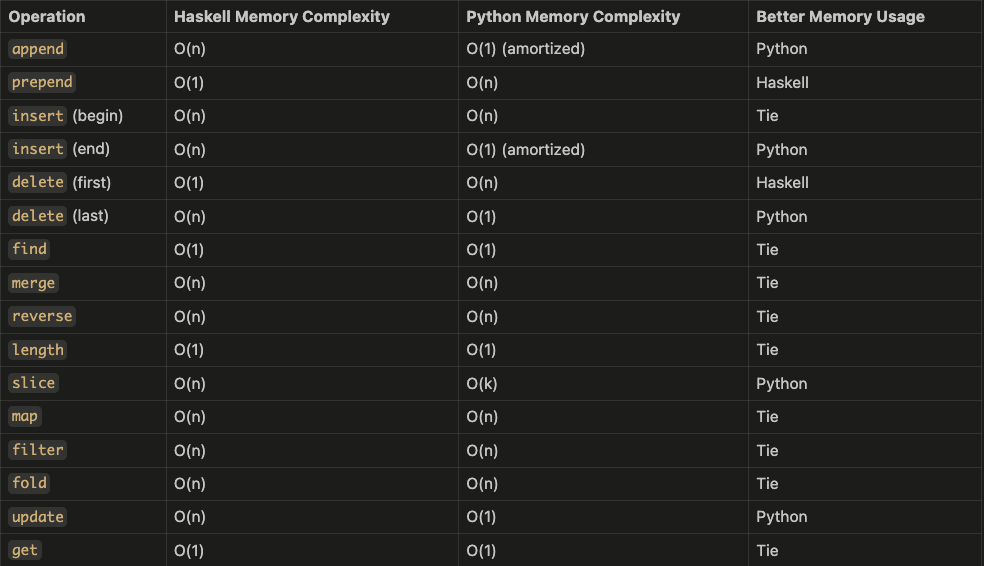
1. Haskell Results :

|  |  |  |
| --- | --- | --- |
| Operations | Execution time | Memory Change |
| Append | 1000ms | 8600 bytes |
| Insert at the beginning | 1.43 ms | 8776 bytes |
| Insert at the end | 1224.78 ms | 8336 bytes |
| Delete the first element | 0.090 ms | 6 bytes |
| Delete the last element | 1.02 ms | 8080 bytes |
| Find an element by index | 1.94 ms | 8 bytes |
| Merge two arrays | 1.01 ms | 8028 bytes |
| Reverse an array | 1.03 ms | 6080 bytes |
| Prepend an array | 0.008 ms | 7 bytes |
| Length of an array | 1.00 ms | 10 bytes |
| Slice an array | 1.00 ms | 8028 bytes |
| Map an array | 0.52 ms | 9128 bytes |
| Filter an array | 1.011 ms | 8080 bytes |
| Fold an array | 4,42 ms | 17 bytes |
| Update | 3531.77 ms | 7216 bytes |
| Get | 0.005 ms | 16 bytes |

Execution time comparison :



Memory Usage comparison:



**Conclusion :**

Execution time : Python generally has better execution time for operations like append, insert at the end, delete last , length , slice and update due to its dynamic array implementation,

Memory usage : Python also tends to have better memory usage for similar operations due to its efficient handling of dynamic arrays.

However, Haskell performs better in operations like prepend and delete first due to its list-based implementation. The actual performance can vary based on the specific use case and the size of the data being processed.