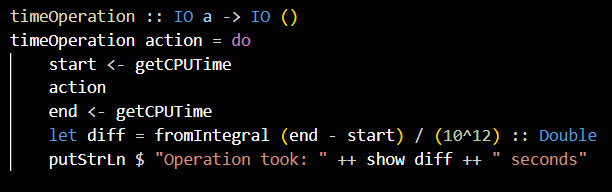
Sophia Jarjoura - 220590

**Step 1: Benchmark Remove Vertex**

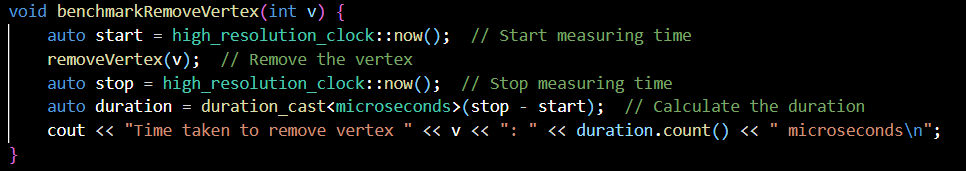
**Haskell**

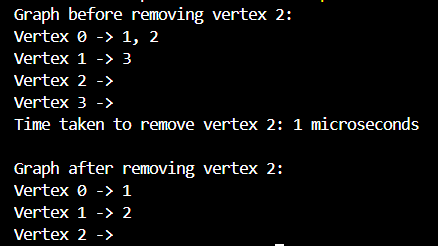


A screen shot of a black background

Description automatically generated

**C++ :**





**Time Efficiency**:

* **Haskell**: The benchmark in **Haskell** shows that the operation of removing a vertex took **0.0 seconds**. This implies that for small-scale graphs, the removal operation is almost instantaneous in **Haskell**.
* **C++**: The removal operation took **1 microsecond**, which is still a very fast operation, but slightly slower compared to **Haskell**. However, this difference is negligible for small graphs.

1. **Memory Management**:
   * **Haskell**: Haskell uses **garbage collection** for memory management, which is abstracted away from the programmer. This may sometimes lead to slower performance for certain types of operations (e.g., large-scale memory allocations), but for simple graph manipulation like vertex removal, this overhead is minimal.
   * **C++**: C++ offers more **manual control over memory management**, which can lead to faster performance for graph operations, especially as the graph grows. The use of pointers and direct memory access is more efficient in **C++**, but the programmer must handle memory management explicitly.
2. **Functional vs. Imperative**:
   * **Haskell** (Functional): Haskell's purely functional nature makes graph operations declarative, using higher-order functions like foldl and map to manipulate the graph. This results in concise and expressive code. However, functional languages like **Haskell** tend to have slightly slower execution times for some low-level operations due to the overhead of functional abstractions.
   * **C++** (Imperative): C++ uses an imperative style, giving programmers more control over the implementation of graph operations. This can lead to more **optimized** and **efficient** code, especially when performance is crucial.
3. **Suitability for Small-Scale Graphs**:
   * For **small-scale graphs**, both **Haskell** and **C++** perform well, with **Haskell** being slightly more efficient in terms of execution time for the operations tested here.
   * However, **C++** will generally provide **better performance** for larger graphs due to its lower-level memory management and more explicit control over execution.
4. **Graph Structure Consistency**:
   * Both **Haskell** and **C++** provide **correct graph updates** after removing a vertex, ensuring the expected changes in the adjacency lists.

**Final Thought:**

* For smaller graphs and quick implementations, **Haskell** can be a very suitable choice due to its high-level, expressive syntax and minimal setup for graph operations.
* For larger, performance-sensitive applications, **C++** can be more appropriate due to its **imperative approach** and **fine-tuned control over memory** and performance.

Functional-style C++ vs Traditional C++:

- Execution Speed: Functional-style C++ is Faster due to better optimization in functional structures like lambdas and algorithms.

- Code Structure: Cleaner and easier to debug using functional patterns.

- Limitations: Not purely functional, e.g., C++ allows mutable state and side-effects.