Title: Symbol Table Implementation and Optimization

## Abstract

This report discusses the implementation of a symbol table for a programming language compiler, utilising two different approaches. The symbol table is a critical component of compilers, responsible for managing information about variables and functions in a program. The two implementations presented here differ in their data structures and optimization techniques.

## Introduction

A symbol table is a data structure that stores information about variables, functions, and other symbols in a programming language. It is essential for a compiler's various stages, including parsing, semantic analysis, and code generation. In this report, we present two different implementations of a symbol table.

## Implementation

1: Linked List

### Data Structure

The first implementation uses a linked list to store symbols. Each symbol is represented by a struct Variable` that contains fields for name, data type, size, dimensions (if it's an array), and address. A global pointer `head` points to the first element of the linked list.

### Operations
1. `add_variable`: This function adds a new symbol to the linked list by dynamically allocating memory for a new `struct Variable` and inserting it at the beginning of the list.
2. `display_symbol_table`: It prints the entire symbol table in a tabular format.

### Pros and Cons

Pros:
- Simplicity of implementation.
- Easy to understand and debug.

Cons:

- Inefficient for large symbol tables (O(n) search time).
- Not scalable for optimizing symbol lookup.

## Implementation

2: Hash Table

### Data Structure
The second implementation uses a hash table to optimize symbol lookup. It introduces the following changes:
- A global array of pointers to `struct Variable` called `hash_table`.
- A `hash` function that calculates the hash value of a symbol's name, distributing symbols across the hash table.
- Each entry in the hash table is a linked list of symbols sharing the same hash value.

### Operations
1. `add_variable`: Similar to the linked list implementation, it adds a new symbol to the hash table. However, it calculates the hash value and inserts the symbol into the corresponding linked list.

2. `display_symbol_table`: This function iterates over the entire hash table, printing each linked list of symbols.

### Pros and Cons

Pros:
- Optimized symbol lookup (average O(1) time complexity).
- Suitable for large symbol tables.
- Scalable and efficient for compiler optimization.

Cons:
- Slightly more complex to implement than the linked list approach.

## Comparison and Conclusion

The two implementations serve the purpose of managing a symbol table, but they have different strengths and weaknesses.

- The linked list approach is simple and easy to implement. It is suitable for educational purposes and small projects. However, it becomes inefficient for large symbol tables, as symbol lookup takes linear time.

- The hash table approach, on the other hand, offers efficient symbol lookup with average O(1) time complexity. It is the preferred choice for real-world compilers, where performance and scalability are crucial. It requires slightly more implementation effort but pays off in terms of runtime efficiency.

In conclusion, the choice between these implementations depends on the specific needs of the compiler project. For educational purposes and small projects, the linked list approach suffices. However, for larger and performance-critical projects, the hash table implementation is recommended.

In practice, compiler symbol tables often incorporate additional features like scope management and error checking, which were not covered in these basic implementations but are essential for a fully functional compiler.