Gianna Julio

CMPT432 - Design of Compilers
Lab 6

1. [***Crafting a Compiler* - Exercise 8.1**] The two data structures most commonly used to implement symbol tables in production compilers are binary search trees and hash tables. What are the advantages and disadvantages of using each of these data structures for symbol tables?

Binary search trees combine the efficiency of a linked data structure for insertion with the efficiency of binary search for retrieval. On average, a name can be inserted or found in O(log n) time, but average-case performance does not hold for symbol tables, where name lookup can take O(n) time. The worst-case scenario for a binary search tree can be avoided if a search tree can be maintained in balanced form. The time spent balancing the tree happens over the many operations to build the tree, so a symbol can be inserted or found in O(log n) time. Examples of such trees include red-black trees and splay trees. Another advantage of binary search trees is their simple, widely-known implementation. This simplicity and the common perception of reasonable average-case performance make binary search trees a common technique for implementing symbol tables.

Hash tables are the most common structure for managing symbol tables due to their excellent performance. Given a sufficiently large table, a good hash function, and collision-handling techniques (like chaining), insertion or retrieval can performed in constant time, regardless of the number of entries in the table. Hash tables are so widely implemented that some languages, including Java, contain hash table implementations in their core library. Hash tables are considered elementary data structures, giving them the advantage of being widely known and implemented like BSTs.

In summary:

	Binary Search Trees	Hash Tables
Advantages	***Ordered retrieval:*** BSTs maintain a sorted order, allowing for efficient traversal in sorted order ***Efficient searching:*** average case time complexity of O(log n), efficient for searching large tables ***Dynamic operations:*** BSTs support dynamic operations like insertion, deletion, and updates ***Space efficiency:*** balanced BSTs typically use less memory compared to hash tables	***Constant-time average case:*** average-case complexity of O(1) for common operations ***No balancing required:*** do not require balancing like BSTs, simplifying implementation and avoiding associated performance pitfalls ***Insensitive to input distribution:*** perform consistently well regardless of the input distribution ***Adaptive sizing:*** dynamically resize tables to maintain space usage and performance
Disadvantages	***Potential for imbalance:*** if unbalanced, BSTs degrade into linked lists, leading to worst-case time complexity O(n) ***Complexity of balancing:*** balancing requires operations like rotations, increasing complexity ***Sensitive to input distribution:*** performance heavily depends on the input distribution; insertion or deletion can lead to significant imbalance and performance degradation	***Unordered traversal:*** do not inherently maintain any specific order ***Worst-case performance:*** worst-case performance degrades to O(n) when there are many collisions ***Memory overhead:*** may consume more memory, especially with hash buckets and collision resolution mechanisms ***Hash function sensitivity:*** performance is sensitive to quality of the hash function and distribution of hash values