Comparative Data Structures for Ordered Dictionary

Mingji Han, Mingyu Chen, Bolun Liu, Haochuan Hu

Introduction

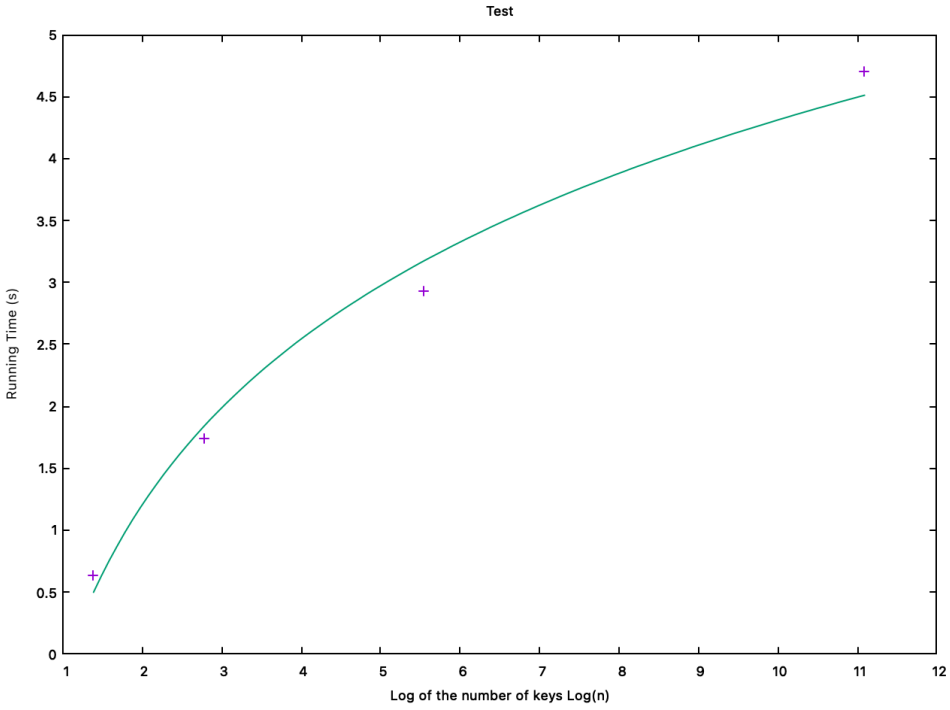
A ordered dictionary is a collection of key-value pairs,in which keys are ordered. It supported operations: insertion, deletion, and lookup. It is widely used in various applications: database system (MongoDB), Filesystem (Ceph), and In-Memory Cache (Redis). In this project, we study and implement 4 types of data structures for ordered dictionary: VEB Tree, AVL Tree, Skip List and binary serach tree. Our codebase can be found on Github: https://github.com/brower/EC504_2023

VEB Tree

VEB (Van Emde Boas) tree is an advanced data structure used for associative arrays with integer keys. It allows for efficient implementations of common operations such as search, insert, delete, minimum, and maximum. The main advantage of a VEB tree over other data structures like binary search trees and heaps is its fast performance, especially in the case of sparse keys.

The VEB tree is based on the divide-and-conquer strategy and has a recursive structure. The main idea is to divide the keys into **high** and **low** parts and recursively build VEB trees for each part. At the top level, a summary structure is maintained to keep track of which parts are non-empty. This enables the VEB tree to perform search, insert and delete operations in O(log log N) time, where N is the size of the universe of keys.

We implemented the VEB Tree in python and evaluate it on Macbook Pro21. The results are shown as follows, which implies a complexity O(loglogN).



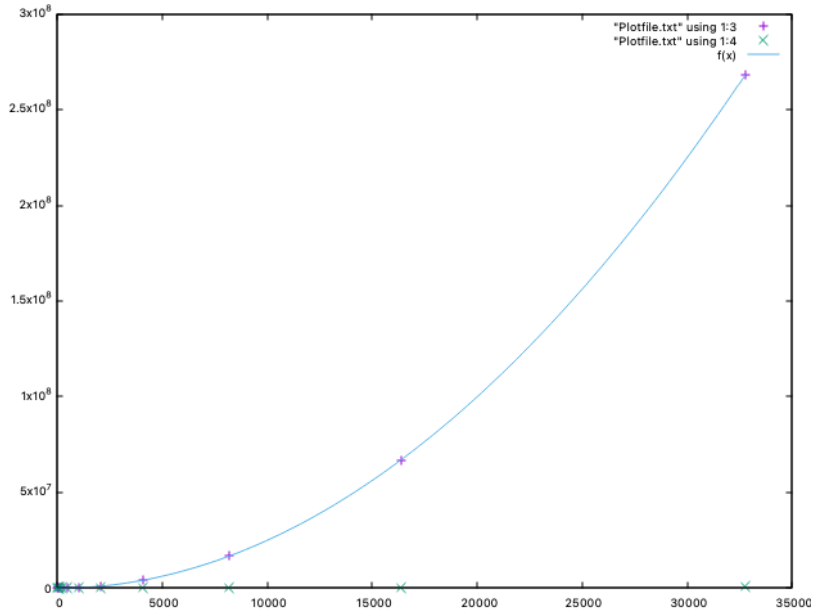
Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

This BST implementation for ordered dictionary supports key-value pairs. Each node in the binary search tree contains a key and a value, which are used to store the key-value pairs.
The insert method inserts a new key-value pair into the tree. If the key already exists in the tree, the associated value is updated.
The delete method removes the key-value pair with the given key from the tree.
The lookup method searches for a key-value pair with the given key by traversing the tree in a manner similar to binary search.
The findMin and deleteMin methods are helper methods used in the delete method to handle the case where a node to be deleted has two children.
Again, this implementation assumes that the keys are comparable

We implemented the BST in Java, and tested 33000 values inserted to an empty BST. Here is the result of time cost. The evaluation result shows that BST has good performance for inserting key values.



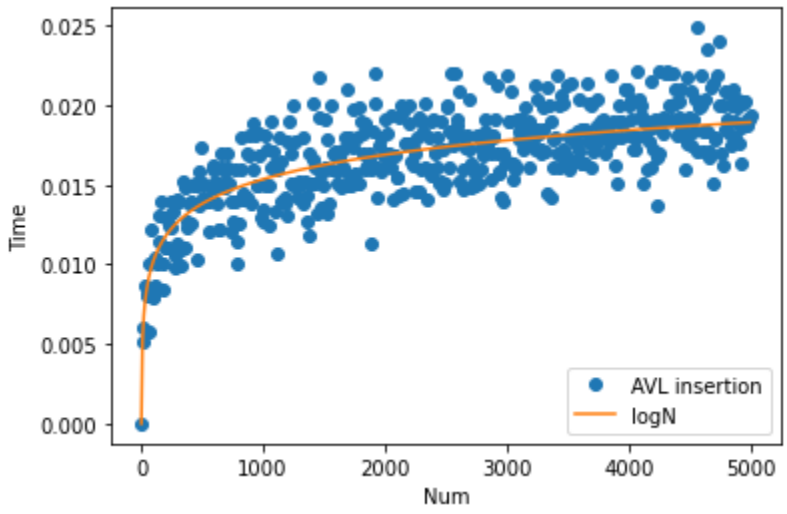
AVL Tree

AVL tree is a type of self-balancing binary search tree that was invented by Adelson-Velsky and Landis in 1962. The balance property of AVL trees ensures that they remain balanced even after a series of insertions and deletions, and it is implemented using nodes with three fields: key, left

child pointer, and right child pointer. The balance factor of a node is the difference between the height of its left subtree and the height of its right subtree. A node is balanced if its balance factor is -1, 0, or 1. If a node's balance factor is outside this range, it is considered unbalanced and must be rebalanced.

Rebalancing of the AVL tree is achieved through a series of rotations. There are four types of rotation: single left rotation, single right rotation, left right rotation, and right left rotation. Each rotation is aimed at maintaining the balanced properties of the tree while making necessary adjustments to its structure.

We implemented an AVL tree in python, and tested 5000 values inserted to an empty AVL tree. Here is the result of time cost.



In summary, an AVL tree is a self-balancing binary search tree that provides efficient insertion, deletion, and search operations with a complexity of O(log n). They are widely used in database indexing, computer graphics, and other applications that require frequent modification of the tree. However, their overhead of maintaining balance properties may make them less efficient in some specific applications than other types of trees.

Skip List

So far we have used tree structure for ordered dictionary. Can we use linked list to implement an ordered dictionary with O(logn) insertion / deletion / lookup cost? The answer is yes. Skip-List is a data structure extends ordinary linked list with multiple level pointers, which can improve the performance of lookup. When we do insertion, we add one more level pointer with probability p until we fail or reach the highest level, which is random process of geometric distribution. When we delete element, we need to delete both element and all pointers at

different level. When we lookup an element, we traverse from highest pointers to lowest pointers. Since the higher level has less pointers, the traversal can “skip” many elements compared with traverse on the linked list directly. Thus it improves the performance of lookup.

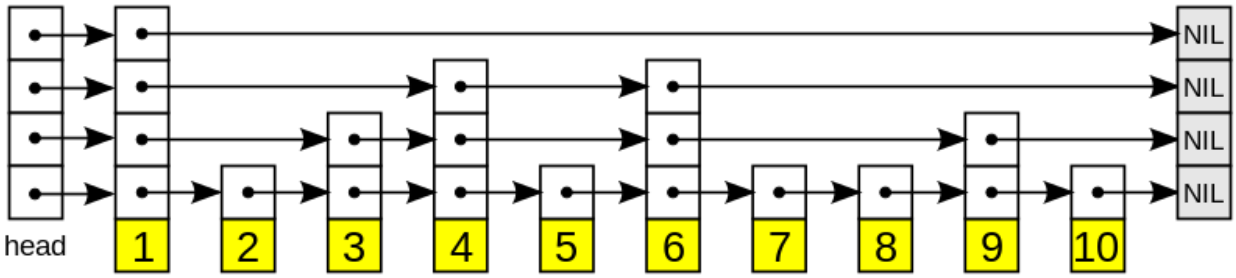
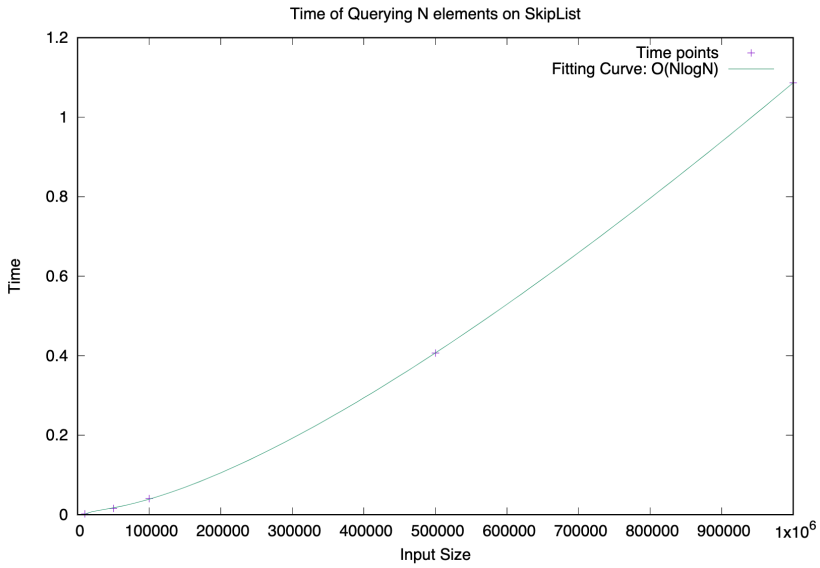


Figure 1. Skip List with Four Level Pointers

We implemented the skip list in C++17 and evaluate it on a server with a Intel(R) Xeon(R) Gold 5317 CPU and 256GB memory, Ubuntu 20.04. The evaluation result shows that skip list has good performance for lookup 1 million integers.



Conclusion

In this project, we studied four data structures for ordered dictionary. Many data structures can support ordered dictionaries including non-linear data structures (tree) and linear data structure (skip list) The key idea is to choose proper data structures for the applications.