Skip Lists, Treaps, and Other Randomized Data Structures for Maintaining Ordered Sets

Bairong Lei, Michael Wexler

University of Waterloo, David R. Cheriton School of Computer Science

Abstract. In this project, we will discuss several randomized data structures for maintaining ordered sets. These data structures use probabalistic methods to remain approximately balanced, which helps to avoid skewed distributions that can lead to excessive running times.

1 Introduction

The need for ordered sets is very common in computer science. By ordered set, we mean a typical set data structure (which provides membership information) which has the additional property of maintaining the order of its elements. At a minimum, an ordered set should contain the functions *insert*, *contains*, *delete*, and *successor*. Programmers often need such a structure in order to determine if a given element is in the set, and also to find the nearest successor of a given element. Given the functionality of an ordered set, it can be extended so that every key maps to a certain value. This would essentially be a dictionary data structure. One would hope that the most common dictionary operations would take $O(\log n)$, or better yet, $O(1)$ time.

There has been much work in developing dictionary data structures that provide fast runtime. Examples of these are hash-tables and binary-search-trees. Hash-tables actually provide $O(1)$ lookup time, compared to binary-search-trees which have $O(\log n)$ lookup time in the average-case. However, hash-tables are not ordered, so they do not provide the *successor* operation. On the other hand, binary-search-trees are ordered, so they are able to provide $O(\log n)$ time for the successor operation.

For binary-search-trees, there is often the occurence where a certain sequence of operations cause the data structure to be skewed. For instance, for a given n, if we were to insert the sequence $1, 2, \ldots, n$ into a normal binary search tree, we would end up having a very skewed structure, which would could take $O(n)$ time to find an element. For this reason self-adjusting binary-search-trees, such as AVL trees [AV62] and red-black trees [LJG78], have been developed. They use deterministic methods to perform self-balancing, so the skewed distribution cannot occur.

The afforementioned data structures, in their classical implementations, all use deterministic methods for organizing their data. Oftentimes, one can use randomization in order to avoid situations where their data structure becomes skewed. In this paper we will discuss various data structures that use randomization in order to maintain approximate balance.

This paper is organized as follows: section 2 will give an overview on several probabalistic set data structures. Section 3 will discuss several of the implementations that we have written. Section 4 will give an overview on some experiments we performed in order to compare these data structures. The last section draws a conclusion of our work.

All test files and program code we describe in this report can be found in our GitHub repository `https://github.com/wexlermi/cs840project`.

2 Randomized data structures

2.1 Treaps

The first randomized data structure we will discuss is the treap [AS89]. Treaps are at their foundation binary-search-trees, with a certain amount of randomness that allows them to remain balanced in the average case. Every element in a treap has the binary-search-tree property, such that all the nodes in its right subtree are greater than the root, and all the nodes in its left subtree are less than the root. However, every node in a treap contains a priority value, in which the priority of a treap node is always higher than that of its children. When we insert a new element into the treap, we randomly assign it a priority, which is a random integer from a sufficiently large range to ensure there are no collisions. Then we put the element in the treap, in the place it would go if it were considered only as a binary search tree. Once it is put in its proper place in the treap, a sequence of rotations are performed to restore the priority requirement, while still maintaining proper place to make sure its key is in the proper order.

To search for an element in a treap, we can simply use the binary-search-tree search algorithm.

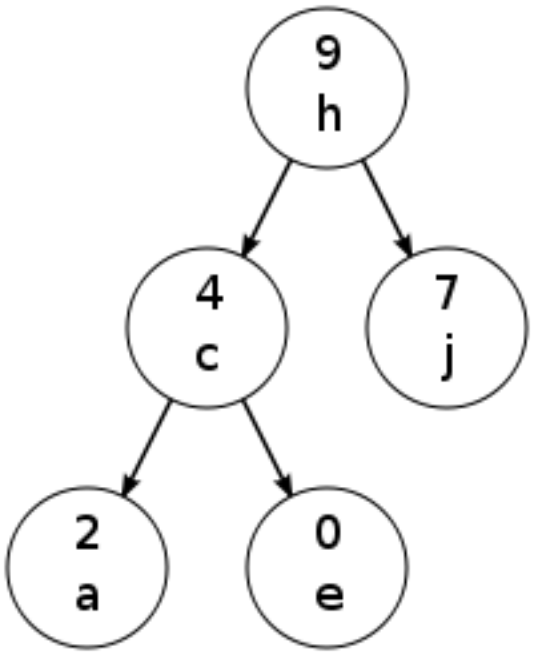

Fig. 1. Example of a treap from http://en.wikipedia.org/wiki/File:TreapAlphaKey.svg

Figure 1 is an example of a very small treap. The letters represent keys, and have the in-order property one would expect in a binary search tree. The numbers represent the priorities. In this case we have a max-heap, since each node has a priority which is a number greater than its children.

2.2 Skip lists

A skip-list [Pug90] is a randomized data structure that is inherently built upon linked-lists. Imagine we had a linked list of n elements, in sorted order. If we had to search for an element x in this linked-list, it would be quite inefficient to linearly search for it. To resolve this issue, we can create a linked-list above it which consists of every other element, or every 3rd element, etc. This way, when searching for an element, we can first search through the top level, and once we find an element greater than it, we can drop down and search our original linked list.

We can repeat the process for the top linked list: make a linked-list which consists of every other element in the linked-list below. We keep repeating this process, where each higher level has half the elements of the level below it, until we get a level which has 1 or 2 elements. Then when we search for an element x, we start at the top list, skipping over elements that are less, then dropping down to lower and lower levels, repeating the process, until we either find the element we are looking for, or reach the bottom level and overshoot it, in which case we determine that the element does not exist in the skip list.

Skip-lists use probabalistic methods in order to determine how many levels of a skip-list a given element can be found on. We specify a varible p which represents the probability that an element x on level i will appear on level $i+1$. Every element appears on level 0, as we go to higher levels, we will find less and less elements. The typical value of p is $\frac{1}{2}$. Every time we would like to insert a key x into the skip-list, we first put it at level 0. Then, we "flip a coin" repeatedly, and while we are getting heads, we keep adding x to higher and higher levels, until we get a tails, in which case we stop. This will mathematically ensure that our skip-list has approximately half the number of nodes at higher and higher levels, for $p = \frac{1}{2}$.

Figure 2 is an example skip list. Notice how each level has fewer nodes as we go higher and higher in the skip list. If we were to search for the element 13, we would start at the top level, and go to the right to 17, in which case we have overshot it. Then we drop down to the second level, on node 7, and go to the right to 14. Drop down to the bottom level, and head to 12 to 13. Notice how skip lists allow us to skip over many nodes, giving us an enormous performance increase as compared to an ordinary sorted linked-list, especially when the amount of elements gets large.

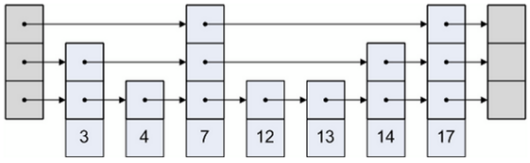

Fig. 2. Example of a skip-list from [Pug90]

2.3 Skip lifts

In [PBM11] Bose, Doueb, Morin write about an improvement to the skip list called the skip lift. A skip lift is a skip list in which every element only has the top two nodes. The rest are deleted. Copies of elements which are at the same level are connected by their left and right pointers. Also, the two copies of an elemented have up and down pointers which connect to each other. In Figure 3, we see a visual representation of the skip list (a) vs. the skip lift (b). Notice how the skip lift uses much less nodes than the skip list. The skip lift also has the property that if no nodes appear in a certain level, it will be empty except for the sentinel element which acts as the header.

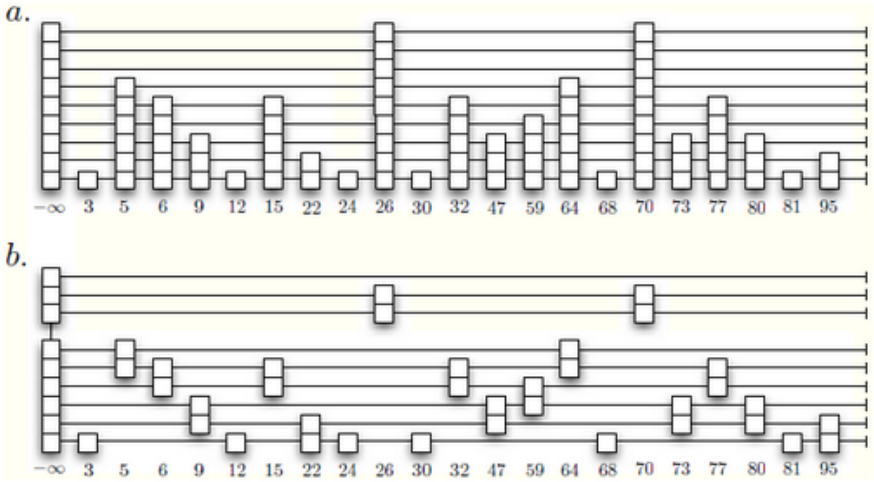

Fig. 3. Skip-list (a) vs. skip-lift (b)

2.4 Jump lists

Jump-lists [HB03] are sorted linked-lists, in which case search is expedited by putting "jump" pointers on several nodes, which serve as an "express-lane" for searching. During the course of this project, we came across three variations of the jumplist data structure. These three types of jumplists differ by their methods of establishing the jump links for each node in the list. In the original paper [HB03], a jumplist is a sorted single or double linked list. This linked list is a circular linked list with an empty value header. Each node has a key value and a next pointer to connect its immediate successor. Meanwhile, each node has a jump pointer that enables it to jump over several of its successors to reach one of its successors. The link between a node and the target node its jump pointer points to is known as an arch. The arch of the header node of the jumplist is called the *fundamental arch*. The jump pointers are created as follows: The node pointed by the jump pointer of the header of the list is selected randomly amongst the rest of the nodes of the list. This assignment splits the list into two independent sublists which are built recursively using the same random procedure. During this construction, the jump pointers do not cross each other. Each node of this jumplist can have more than one incoming jump pointers, but have only one jump pointer pointing to its successor.

The second variation of the jumplist data structure is found in [PBM11]. It has most of the same features as the jumplist featured in [HB03]. It maintains the core feature that the jump pointers do not cross over each other. Meanwhile, it ensures that each node has at most one jump pointer being pointed to it and only one jump pointer pointing to its successor. Figure 4 depicts a sample randomized jumplist with 21 nodes.

The third variation of the jumplist data structure comes from [Elm05]. Like the jumplist introduced by the original randomized jumplist paper, its jumplist maintains the set of common features such as the fact that jump pointers can not cross over each other. However, when the jump pointers are created for the jumplist, the procedure first selects the median node as the node pointed by the jump pointer of the jump list. This fundamental arch divides the whole list into two sublists evenly. These two sublists are then further split evenly and independently using the same process until it reaches the smallest sublists that have only one node and make its jump pointers point to itself. This process builds what they refer to as a *perfect jumplist*. Figure 5 depicts an example of a perfect jumplist with 23 elements.

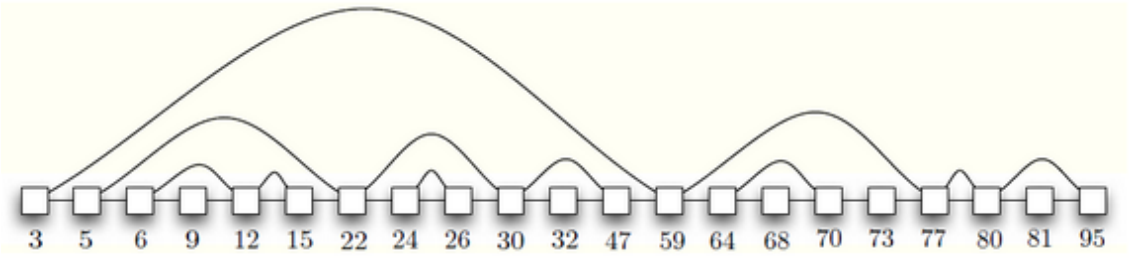

Fig. 4. A randomized jumplist with 21 nodes

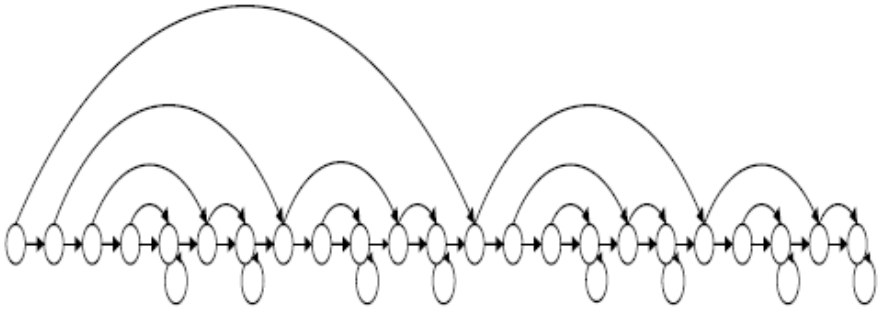

Fig. 5. A perfect jumplist with 23 elements

2.5 Skynet

Skynet is a scalable overlay network data-structure designed to eliminate the disadvantages of peer-to-peer systems that have no control of data storage location and do not satisfy the need to restrain the routing path within a certain range. Due to the limited time for this project, and the complexity of this data-structure, we were unable to implement SkyNet, but instead we will give a brief overview of the design of SkyNet.

Essentially, the structure of the Skynet is a circular double linked list. Each of node of this ring structure has a name ID field which is known as the data record key. From Figure 6, we can see a Skynet structure [NJHW03] with eight nodes, each of which has an alphabetic character. Each Skynet node store $2 \log N$ pointers, where N is the number of nodes in this structure. The pointers facilitate a set of rings as paths for routing through nodes in the rings to search an item. Figure 7 demonstrates the same Skynet of Figure 6, which shows a diverse set of rings by selecting nodes of every level to establish interconnections. For example, from the root ring, we may choose nodes of A, M, T and X to form a subring for the level 1 overlay. We may also further select nodes of A, T and M, X to form two subrings for level 2 overlay from its upper level ring. We continue to use this process to form smaller and smaller subrings until each node only has itself as the new ring. Meanwhile, each node is assigned a numeric ID representing the routing path to reach it.

Routing in the SkyNet: Routing can follow a clockwise direction or a counter-clockwise direction. There are two routing approaches to support lexicographic searching. One approach is called *Routing by name ID*. For example, if

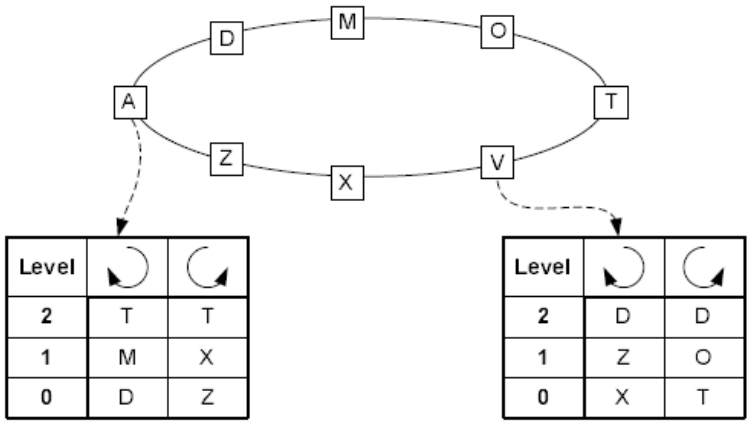

Fig. 6. A Skynet with 8 elements

we need to search node V from node A, this methodology begins routing from Ring 00. Since there are only nodes A and T in Ring 00, it goes down one level to level 1 to route around the subring formed with nodes A, M, T and X. At this moment, we can find node V, so it goes down one level to the level 0 to route around the root ring and eventually find the node V. Another routing approach is called *Routing by numeric ID*. There is a hash function that transfers the key of the target item into its corresponding numeric ID. Using the same example to search for node V, the hash function transfers the key V into 111. Therefore, to search node V from node A, the first digit of the numeric ID 111 is 1 and guides the routing from the root ring to the right subring Ring 1 in level 1. Then the second digit 1 directs the routing to the right subring Ring 11 in level 2. Eventually the last digit 1 guides the search routing to the right subring Ring 111 in level 3 and find the target node V. The run time for both Routing by name ID and numeric ID is $O(\log n)$, where n is the number of the nodes. Potential future work for this structure can be implementing the insertion and deletion methods.

3 Implementation

For this project, we decided to implement the skip lift [PBM11]. We built our implementation off an implementation of the skip list [Jiw]. We also implemented the jump list[HB03]. Our implementations can be found on our GitHub here: `http://github.com/wexlermi/cs840project`.

Due to the limited time for this project, we were unable to implement the operations of *delete* and *successor*. We were unfortunately only able to implement *insert* and *contains*, which are the minimum operations a set data structure must contain.

For the experiment we utilized previous implementations of the Treap [Weib] and the AVL Tree [Weia].

4 Experiment and Analysis

We thought it would be interesting to collect metrics on runtime and datasize of these probabilistic data structures, to see which ones perform most competitively from a time and space performance perspective. Since the AVL tree is deterministic, we decided to also measure the performance of it, and to use that as a baseline.

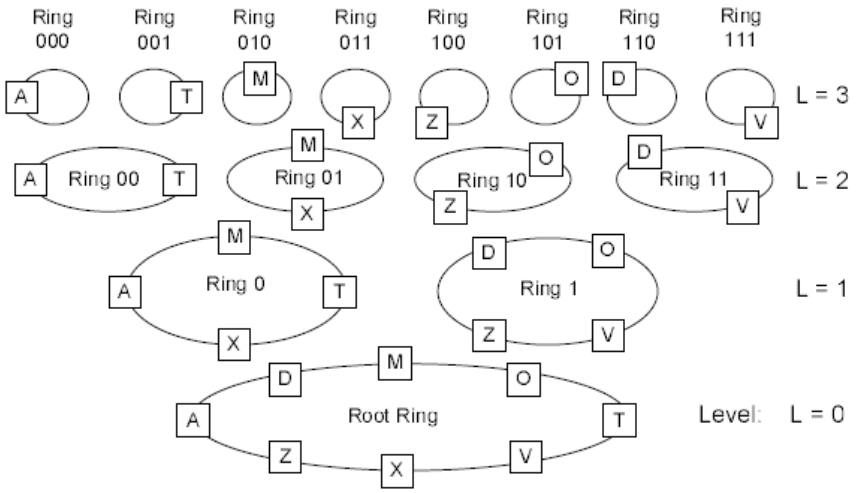

Fig. 7. A ring hierarchy for an 8-node Skynet

For our experiment, we generated a test data set of one million randomly chosen integers. This data-set exists in our GitHub with the name "testdata.txt". For each of our data structures, we measured the timing it would take to insert all the test data into the data structure, as well as the space occupied in memory of the data structure. Also, we made a random permutation of the elements in our data set, which represents a sequence of requests. For this sequence of requests, we measured the time it took to query all the elements for each data structure. Our results are in Table 1.

Table 1. Timings for each data structure in seconds

Data Structure	Time to insert (s)	Time to query (s)
AVLTree	1.462	1.227
Treap	2.745	2.101
SkipList	2.610	2.789
SkipLift	4.106	5.835
JumpList	4.056	4.919

Upon analyzing our timing results, we note that the AVL Tree, which is deterministic, had the best time performance for both types. Amongst the probabalistic data structures, we find the Treap and SkipList gave the best performance. The SkipLift and JumpList took approximately twice as long as the other probabalistic data structures, for both insert-time and query-time.

Upon analyzing our space results, we find that the JumpList took up the least space, by far. This is a testament to the space-efficiency of the JumpList data structure, which uses a similar skip mentality to the SkipList, but uses far less extra pointers, since only a fraction of the nodes in the JumpList contain jump pointers. The SkipList and SkipLift take up the most space, since their structure is conducive to using extra nodes and pointers. Comparing SkipList to

Table 2. Space occupied by the data structures in megabytes

Data Structure	Space (mb)
AVLTree	27.447
Treap	26.402
SkipList	57.601
SkipLift	68.439
JumpList	15.762

SkipLift, we were quite to surprised to see that the SkipLift takes up more space than the SkipList. This is surprising because the SkipLift only keeps the nodes on the top two levels for a given element, and gets rid of the rest. However, the SkipLift keeps track of the previous pointers for every node, which appears to have a greater effect on the total space than the deletion of extra nodes. This appears to demonstrate that the SkipLift does not offer any benefits as compared to the SkipList.

5 Conclusion

From the start of this project, we were interested in seeing if using randomization has any effect on the performance of a dictionary data structure. Using the deterministic AVLTree as our baseline, we come to the conclusion that for our situation, there is not too much benefit to using randomization, from a time-performance point of view. As mentioned before, the AVLTree performed the operations quicker than any of the other data structures. However, as we mentioned the JumpList takes up less space than the AVLTree, which makes it quite a useful data structure compared to the AVLTree if memory is limited.

We find the data structures presented to all be quite useful in various scenarios, and usually the level of their utility depends on the particular application. We note that randomization can also have benefits for algorithms such as QuickSort, in which the pivot is randomly chosen each time, or, in a hash-table, to keep elements from clustering. The importance of probabalistic methods in enhancing algorithm and data-structure performance should not be overlooked.

References

[AS89] C. Aragon and R. Seidel. Randomized search trees. *Proc. 30th IEEE Symposium on Foundations of Computer Science*, pages 540–546, 1989.

[AV62] G.; E. M. Landis Adelson-Velskii. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263266, 1962.

[Elm05] Amr Elmasry. Deterministic jumplists. *Nordic Journal of Computing*, Volume 12 Issue 1:27 – 39, March 2005.

[HB03] Marianne Durand Herv Brnnimann, Frdric Cazals. Randomized jumplists: A jump-and-walk dictionary data structure. *Lecture Notes in Computer Science*, 2607:283–294, 2003.

[Jiw] Kamil Jiwa. Skip list implementation in java. http://www.crimsonglow.ca/ kjiwa/src/java-skip-list/SkipList.java.

[LJG78] Robert Sedgewick Leo J. Guibas. A dichromatic framework for balanced trees. *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21, 1978.

[NJHW03] Michael B. Jones Stefan Saroiu Marvin Theimer Nicholas J.A. Harvey, John Dunagan and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 4, 2003.

[PBM11] Karim Doueb Prosenjit Bose and Pat Morin. Skip lifts: A probabilistic alternative to red-black trees. *Combinatorial Algorithms: Lecture Notes in Computer Science*, Volume 6460:226–237, 2011.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33 Issue 6:668–676, June 1990.

[Weia] Mark Allen Weiss. Avl tree implementation in java. http://users.cis.fiu.edu/ weiss/dsaajava/code/DataStructures/AvlTree.java.

[Weib] Mark Allen Weiss. Treap implementation in java. http://users.cis.fiu.edu/ weiss/dsaajava/code/DataStructures/Treap.java.