Skip Lists, Jump Lists, 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 probabalistic data structures for maintaining ordered sets. These data structures use probabalistic methods to keep 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 has the additional property of maintaining the order of its elements. Programmers often need such a structure in order to determine if a given element is in the set, and also to find the nearest predecessor for 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 set data structures that provide O(log n) lookup time. Examples of these are hash-tables, AVL trees, and red-black trees. In addition to quick lookup times, AVL trees and red-black trees have the additional property of being able to provide O(log n) time for the next-predecessor operation, which, for a given key, will give the smallest key that is greater than that key.

For the afforementioned dictionary data structures, there is often the occurence where

For instance, for a given n, if we were to insert the sequence 1 2 ... 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. This is the reason AVL and red-black trees were developed. They use deterministic methods to perform self-balancing, so the skewed distribution never occurs.

The afforementioned data structures in their classical implementations all use deterministic methods for organizing their data. Oftentimes, one wants to put a certain element of randomness into their dictionary data structures in order to avoid situations where their data structure becomes skewed. In this paper we will discuss various data structures that use randomness in order to maintain a level of balance, to avoid skewed distributions.

This paper is organized as follows: section 2 will give an overview on several probabalistic set data structures. Section **??** 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 talk about is the treap. 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.

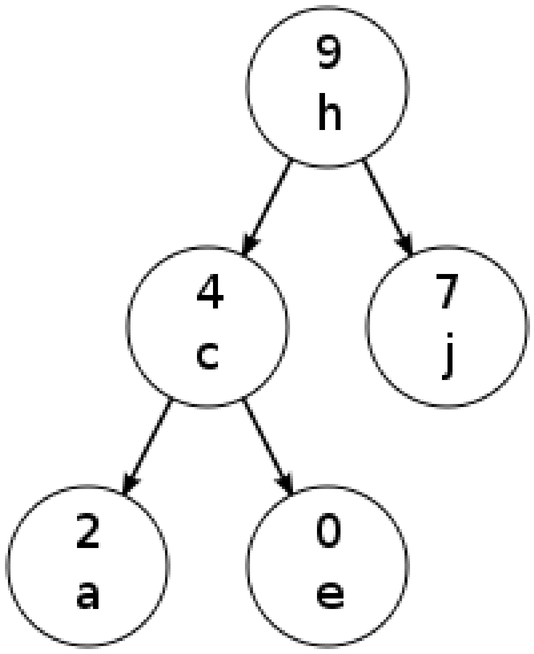

Fig. 1. Example of a treap from http://en.wikipedia.org/wiki/File:TreapAlphaKey.svg

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 which case we have a max-heap, since each node has a priority greater than its children.

2.2 Skip lists

Skip lists are a randomized data structure that is inherently built upon ordinary linked-lists. Imagine we had a linked list of n numbers, 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. Instead, we can put pointers 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: put pointers to every other element. We keep repeating this process, where each higher level has half the pointers of the level below it, until we get a level which has 1 or 2 pointers. 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.

While the above description of a linked-list seems good at first, we reach a problem for insertion and deletion, in which case the length of the levels no longer have length 2^i. Instead, we decide to use probabalistic methods to keep our levels approximately of length 2^i. We specify a varible p which represents the probability that an element on level i will appear on level $i+1$. The typical value is $p = \frac{1}{2}$. 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" a certain amount of times, and while we are getting heads, we keep adding the element to higher and higher levels, until we get a tails, and then we stop. This will mathematically ensure that our skip-list has approximately half the number of nodes at higher and higher levels, and provides us with good performance for insert and delete operations, something that is quite hard to maintain in deterministic skip-lists.

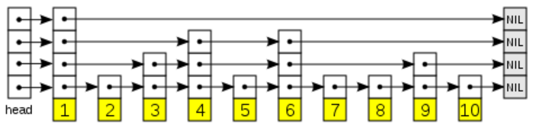

Fig. 2. Example of a skip-list from [Pug90]

2.3 Skip lifts

In [PBM11] Bose, Doueb, Morin talk 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. This greatly reduces the size of the skip list.

2.4 Jump lists

We have found three invariants of the jumplist data structures. These three types of jumplists distinguish each other by their approaches to establish the jump links for each node in the list. In the original paper Randomized jumplists - a Jump-and-Walk Dictionary Data Structure by Frederic Cazals and Marianne Durand, 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 successor. 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 the followow: The node pointed by the jump pointer of the header of the list is selected randomly among the rest 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 invariant of the jumplist data structure is found in the paper Bose, Prosenjit, Karim Doueb, and Pat Morin. "Skip lift: a probabilistic alternative to red-black trees. Combinatorial Algorithms. Springer Berlin Heidelberg, 2011. 226-237. It has most of the same features as the previous paper for a jumplist. 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

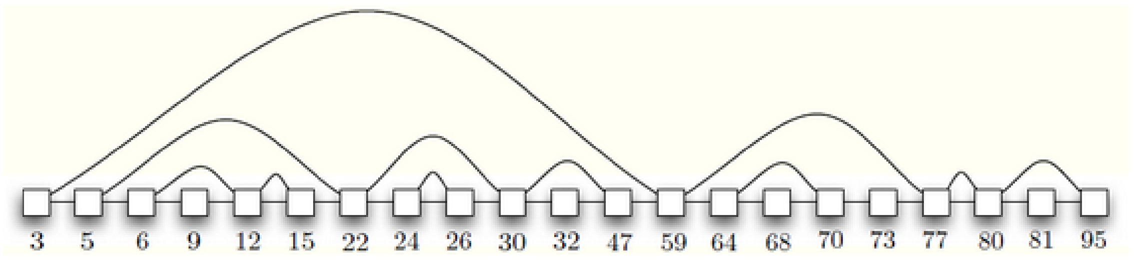

Fig. 3. A randomized jumplist with 21 nodes

being pointed and only one jump pointer pointing to its successor. The below figure depicts a sample randomized jumplist with 21 nodes.

The third invariant of the jumplist data structure comes from the paper Elmasry, Amr. "Deterministic jumplists." Nordic Journal of Computing 12.1 (2005): 27-39. Like the jumplist introduced by the original randomized jumplist paper, its jumplist maintains the set of common features such as the jump pointers can not cross over each other. However, when the jump pointers are created for the jumplist, the procedure firstly 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 pointing to itself. This process is called building a perfect jumplist.

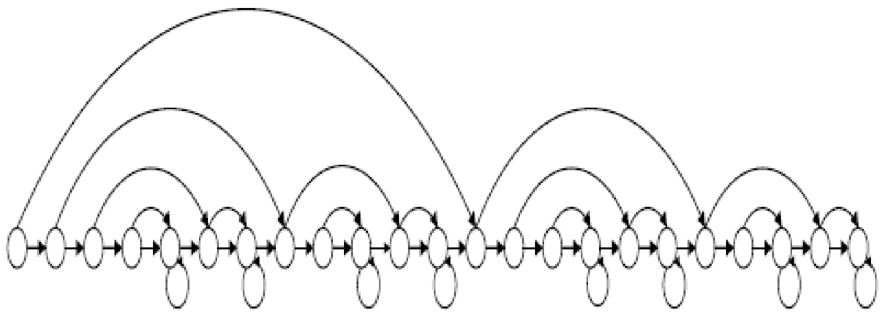

Fig. 4. A perfect jumplist with 23 elements

Support methods:

A jumplist usually has a set of common methods a dictionary data structure contains. Search, successor, insertion and deletion are the most popular methods a dictionary data structure supports. Our project aims to implement some of these methods for the jumplist data structure and evaluates and verify its theoretical running time estimation suggested from the Deterministic jumplists paper.

Implementation:

There are some challenges to implement the insertion and deletion methods for the jumplist, since the pseudo code for these two methods are missing in the original randomized jumplist paper. Although there has been a C plus plus implementation for the jumplist data structure in H. Brnnimanns research work (`http://photon.poly.edu/~hbr/publi/jumplist.html`) from the Polytechnic University, unfortunately its source code is too old and severely handicapped for its missing search and insertion methods so that we have to pursure another solution. Another challenge is when we strive to use the insertion algorithm from the deterministic jumplist paper, we find there is a defect in that algorithm so that whenever there is an inserted item whose value is less than the value of the header node, the insertion method always places it to the second place of the list.

We decide to implement our data structures in Java with the Eclipse IDE. For the time being, we only implement the search and insertion methods for the jumplist. The jumplist is initialized as an ordinary linked list at the beginning. A method called buildperfectjumplist is invoked to create the jump pointers for each node using the algorithm introduced in the deterministic jumplist paper. Meanwhile, each node updates its corresponding sizes of its next and jump sublists. The search method is implemented basing on its algorithm in the randomized jumplist paper to use the jump links as many as possible to skip nodes to speed up the search. For the insertion method we have implemented two versions to compare their running time. The first insert method is called djinsert which uses exactly the same algorithm introduced in the deterministic jumplist paper and inherits the first item insertion defect stated previously. After a new item is inserted, its jump pointer is updated as follows: The whole list is iterated to check whether there exists a node violating the balancing condition to rebuild the jump pointers for the entire list. If the jump pointer of the predecessor of the new item points to itself, its jump pointer is changed to point to the new item and the new item also inherits the jump pointer of its immediate successor. Meanwhile, the new items sizes of its next and jump sublists is updated respectively. On the other hand, the other insertion method follows the basic search algorithm to find the insertion point and inserts the new item as well as calling the buildperfectjumplist method to rebuild all the jump pointer for the entire list. This method resolves the first item insertion defect and does not need to maintain its sizes of next and jump sublists, but we expect the its running time is longer due to the redundancy of its jump pointer rebuilding procedure.

Running result:

Skynet:

Data structure:

For the time being of our project, we determine to consider the implementation for Skynet data structure as a potential future work. Skynet is a scalable overlay network 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. 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 the above figure, we can see a Skynet structure [Harvey, Nicholas JA, et al. "Skipnet: A scalable overlay network with practical locality properties." Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems. Vol. 4. 2003.

] with eight nodes each of which has an alphabetic character. Each Skynet node store 2log N pointers, where N is the number of nodes in this structure. This pointers facilitates a set of rings as paths for routing through nodes in the rings to search an item. The below figure demonstrates the same Skynet of the previous figure, 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 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. Using this process to form smaller and smaller subrings until each node only has itself to form new ring. Meanwhile, each node assigned a numeric ID representing the routing path to reach it.

Routing:

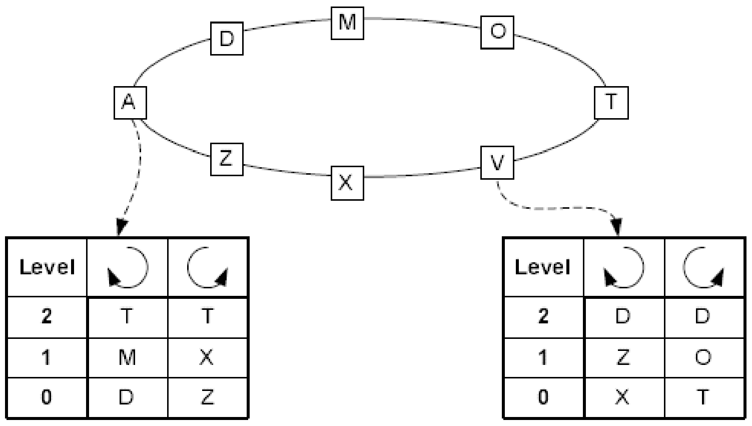

Fig. 5. A Skynet with 8 elements

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 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 the Ring 00, it goes down one level to the level 1 to route around the subring formed with nodes A, M, T and X. At this moment we can find node V either, 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 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. 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`.

4 Experiment and Analysis

We thought it would be interesting to collect metrics on runtime and datasize of these data structures, to see which ones perform most competitively from a time and space performance perspective. We conjectured that the skip lift would take less space than the skip list.

Remark 1. Hello

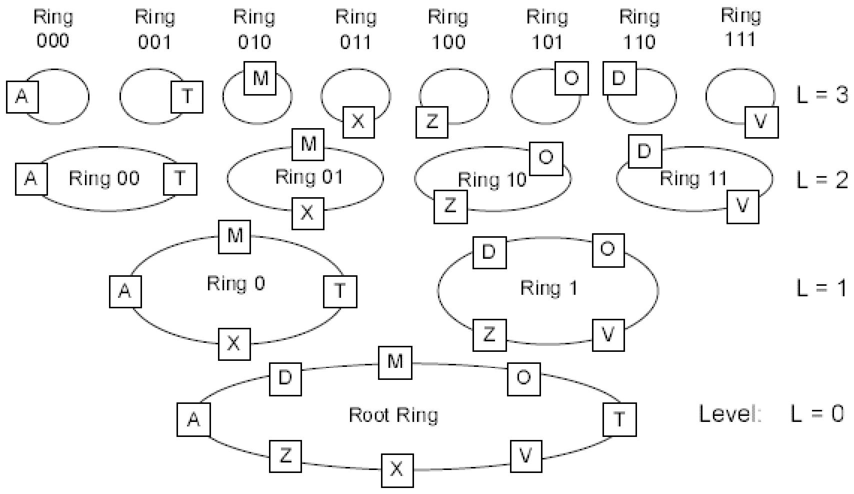

Fig. 6. A ring hierarchy for an 8-node Skynet

- The executable file that was compiled by `crestc`
- A number of maximal iterations through the control flow graph we want to allow (i.e. a limitation to the paths we want to consider. Otherwise the program might run too long)
- A strategy CREST should use to traverse the control flow graph. Possible options here are in the set

$$\{\texttt{dfs}, \texttt{cfg}, \texttt{random}, \texttt{uniform\_random}, \texttt{random\_input}, \texttt{cfg\_baseline}, \texttt{hybrid}\}$$

Table 1. DFS branch-coverage-ratio

1	1	1	1	1	1
2	0.75	0.75	0.25	0.25	0
3	0.667	0.667	0.667	0.333	0
4	0.625	0.75	0.25	0	0
5	0.6	0.7	0.6	0	0

5 Conclusion and Future Work

There are several conclusions we can draw in this report.

References

[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.

[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.