COSC 223 Project 3: Treaps
An Analysis of Standard vs. Access Frequency Implementation

Ben Fleischmann, Mieke McLaughlin, Sarah Tam, Zhiyuan Jia

Introduction:

Binary Search Trees are a very important class of data structures in Computer Science. They provide a framework to store a dynamic data set in memory. The basic structure of a Binary Search Tree is as follows. Data points, or nodes, are organized by their unique key. Each node holds on to pointers to a left child and a right child. A node's left child must have a key strictly less than that of the node, and it's right child must have a key strictly greater than that of the node. We denote the number of nodes in a tree "n". When a Binary Search Tree is "balanced", it's depth will be approximately lgn, where n is the number of data points in the tree. A balanced tree therefore performs inserts, searches, and deletes in O(lgn) time, proving to be an efficient all around data structure.

However, problems arise when a Binary Search Tree becomes unbalanced. Based on the order of inserts and deletions, one unique set of data can lead to the creation of multiple different BSTs. For example, if a set of data is inserted in increasing key order, the resulting tree will resemble a linked list and perform inserts, searches, and deletes in linear time. This aspect of BSTs becomes a problem when operations are not randomly distributed. For example, if you want your tree to store a group of students by name, and you insert the students from an alphabetical list. This series of operations would be considered a worst case input.

Computer Scientists create randomized algorithms and data structures to eliminate the possibility of a worst case input. A Treap is a Binary Search Tree variation that utilizes randomization to eliminate any worst case input and therefore guarantee average runtimes of O(lgn) for its operations. Each node in a Treap is assigned a randomly generated priority. Data stored in a Treap follows criteria from both BST and Heap: data is organized by BST criteria based on key, and organized by Heap criteria based on priority. Heap organization dictates that each node must have a lower priority than its parent. This double organization ensures that a given set of data, regardless of the order of inserts and deletions, will be stored in a unique binary tree. A Treap essentially takes on the structure of its respective BST where data is inserted in order of priority. Since priorities are generated randomly, a Treap can be thought of as a BST constructed from a random series of operations. In a random binary tree, the expected

distance from a node to the root is proportional to lgn, therefore a Treap ensures average runtimes of O(lgn) for its operations. Other BST variations such as Red-Black Trees guarantee worst case runtimes of O(lgn) but are more difficult to implement. Treaps are highly regarded for their elegance and efficiency. They are used in production applications ranging from wireless networking to memory allocation.

In this project, we compared the performance of a standard Treap, and a Treap that rebalances based on access frequency. The idea of the Access Frequency Treap, or AFTreap, is that each time a node is accessed, a new random priority is generated. If the new priority is higher than the original priority, the node takes on the new priority and is sifted up to the correct spot in the Treap. Over a series of accesses, nodes with higher rates of access will appear higher up in the Treap. Our goal was to test if rebalancing based on access frequency leads to faster average search() times. We will use two data request distributions, uniform and zipf. To understand why the Treap and AFTreap perform the way they do, we will analyze the graphs of the Treap and AFTreap's average search time against the range of Keys in the treap. Note that the range of Keys is also the range of possible Keys used as the parameter when calling search(Key).

Experimental Setup:

We decided to run tests off of accesses drawn from both a uniform and zipf distribution. Therefore, we started by implementing our standard Treap data structure, our Access Frequency Treap, and a request generator that output a series of requests drawn from either a uniform or zipf distribution.

Data Structures Setup:
Because there are many variations of Treap, we specify the operations of the treap we used for this experiment.
Our Treap data structure performs three operations:

1. Insert
   insert() takes two Integer inputs, Key and value, and returns nothing.
2. Delete
   delete() takes an Integer Key and returns an Integer value associated with the Key;
3. Search
   search() takes an Integer Key and returns a Node associated with the Key.

To maintain the heap property, we perform right rotation and left rotations:

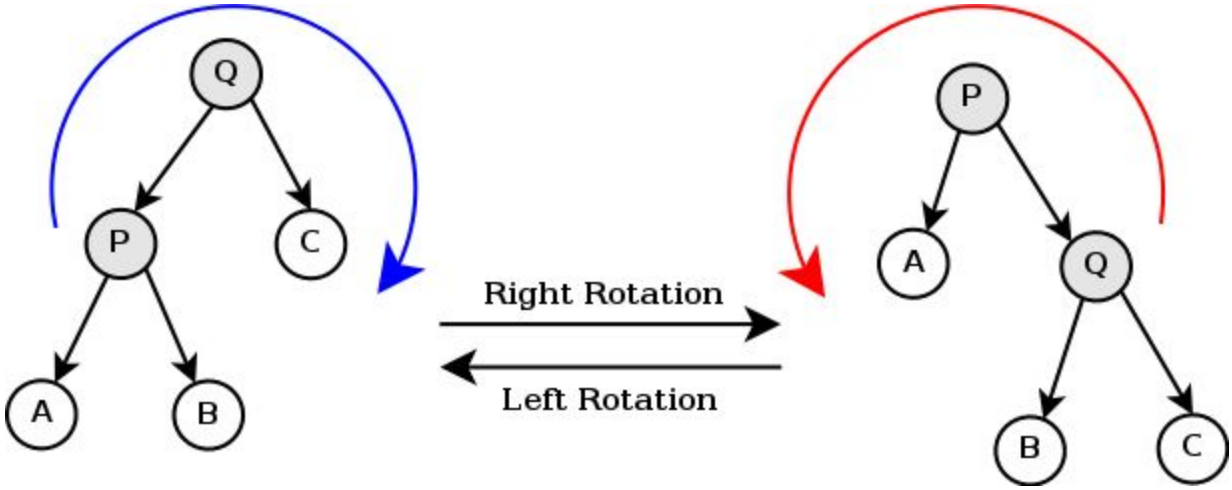

Figure 1[1]Left Rotation and Right Rotation

Access Frequency Treap provides the same three operations and the same rotations as the Treap. It's only different from the Treap that its search method generates a new random priority for the node associated with the Key, which will replace the old priority if the old one is smaller.

Distributions:

For a uniform distribution, each data item is requested at the same probability.

In a Zipf distribution, each item's probability to be requested is inversely proportional to its index, which follows a general formula:

$$p_D(i) = \frac{(1/i)^{\alpha}}{\sum_{j=1}^{\text{Range}} (1/j)^{\alpha}}$$

In this experiment, we specifically consider the case α = 1, so

$$p_D(i) = \frac{1/i}{\sum_{j=1}^{\text{Range}} 1/j}.$$

[1] https://commons.wikimedia.org/wiki/File:Tree_rotation.png

To simulate uniform requests, we use the JAVA random class to generate integers between 1 and range with a same probabilities of 1/range.

To simulate Zipf requests, we used an inverse transform method to map y~unif (0,1), in which y is equally likely to take on any value in [0,1], onto X, the requested integer as following:

1. Sort X1, … , Xrange, so that X1<X2<X3<…<Xrange, in which Xi has a probability of $p_D(i) = \frac{1/i}{\sum_{j=1}^{\text{Range}} 1/j}$,

   Create an int array with size range, fill the array's (i-1)'s space with pD(i).

2. Generate y~unif (0,1) using JAVA random class.
3. If 0 < y < pD(1), then generate the request X = X1 ;
4. If pD(1) < y < pD(1) + pD(2), then generate the request X = X2.
5. Follow the general formula in 3 and 4 to generate the X for all requests.

To fill both data structures, we did a series of inserts of every integer (shuffled with the Fisher-Yates algorithm) between one and a chosen range, inclusive. After the Treap and AFTreap were filled, we called the search(Key) method on each 10^6 times. We repeated this process twice, once generating the Key following uniform distribution, the other time following zipf distribution. We then repeated these two experiments on ranges of numbers between 100 and 10000. Specifically, we chose ranges 100, 500, 1000, 2500, 3000, 4500, 6000, 8000, 9000, and 10000. For each range and distribution, we ran 20 trials and took the average runtime, excluding the first five trials from our calculation. We found that the first five trials tended to have higher runtimes, which skewed the data (perhaps this is caused by the Garbage Collector).

Results and Discussion:

We were surprised to find that the access frequency treap generally performed worse than the regular treap under uniform distribution, as we initially expected it to perform more efficiently across the board. From our experiments we gained insight about the type of input and conditions in which the access frequency treap might be a better choice than the regular treap.

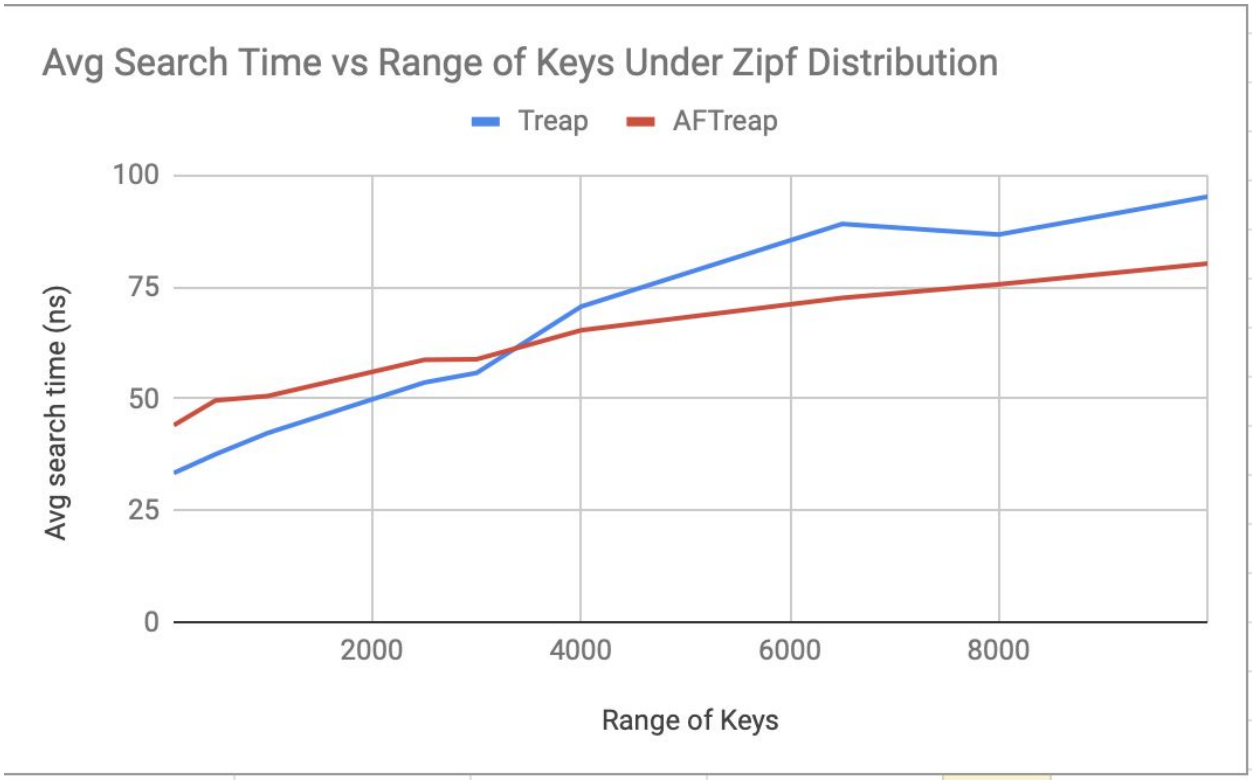

Figure 2: Avg Search Time vs Range of Keys Under Zipf Distribution

For the zipf distribution, as shown in Figure 2, the access frequency treap performed less efficiently than the standard treap for accesses with a smaller range. However, it began to perform better with ranges greater than 3000. Intuitively, this makes sense. As the range increases, so does the height of the tree. The benefit of the access frequency treap is that it moves the most frequently accessed elements higher up in the tree over time. The most common numbers are searched for more times, and a new priority is generated each time. After a number has been accessed k times, its priority will be the maximum of k random variables. Thus, these values are more likely to have a higher priority, which places them higher up in the tree, allowing for faster subsequent searches.

With the zipf distribution, the first dozens of integers dominate the sample space. For example, getting an integer from 1 to 10 from a Zipf distribution when range is 1000 has a 39.13% probability (as shown in Figure 3). Given these predictable frequencies, the access frequency treap is generally able to quickly search for values, since it has moved the most accessed data higher up in the tree.

But the question of why Treap outperforms AFTreap when the range is small will not be resolved until we take a look at the result from the uniform distribution experiments.

Integer	Pr{Being Requested in Zipf}
1	13.36%
2	6.68%
3	4.45%
4	3.34%
5	2.67%
6	2.23%
7	1.91%
8	1.67%
9	1.48%
10	1.34%
Sum	39.13%

Figure 3

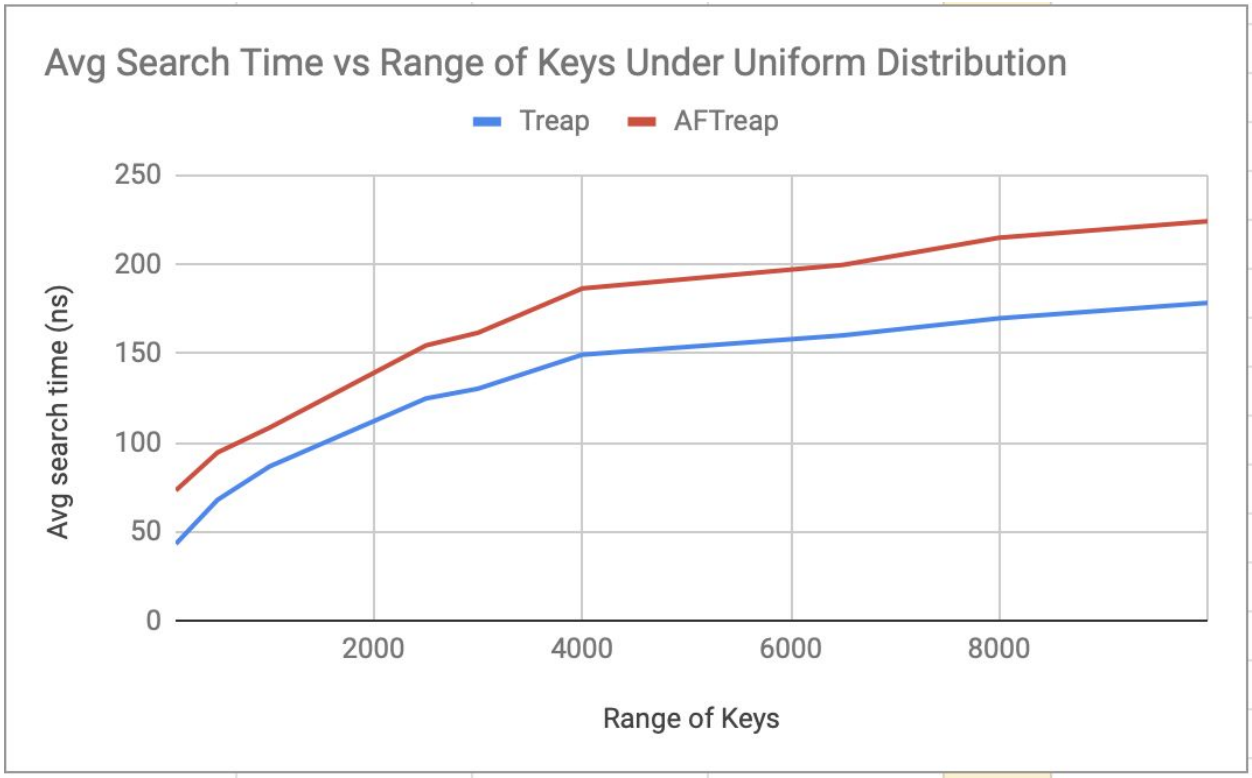

Figure 4: Avg Search Time vs. Range of Keys Under Uniform Distribution

For the uniformly-distributed accesses, the access frequency treap does not enjoy the advantage it has when access is zipfly distributed. Moving items up in the tree does not have a long-term payoff because all numbers are equally likely to be accessed. A number deeply buried in the tree, which requires a longer access time, is just as likely to be accessed next as a number that was recently more frequently accessed previously. On the other side, a number that was moved up may not be called again for a long time, sinking the cost of its upward rotation. From looking at the graph, there appears to be some operation that is costing the access frequency treap consistently more time than the regular treap. Our hypothesis is that this is all due to the rotations happening when moving a node up, which offers no real benefit when it comes to the uniform distribution. This hypothesis finds support in the graph. The gap between the average search time of the two Treaps widens as range of values increase. As the range rises, the height of the treap rises as well, which can potentially increase the cost of rotating a node up because the node might have to rotate up through more levels of nodes.

The rotation-cost hypothesis can also explain why AFTreap search performed worse than Treap search when range of values is small, when key is requested following the zipf distribution. The cost of rotations can only be offset when the average search cost of Treap is high enough. We hypothesize the following general relationship:

Range of Keys (N)	Relationship
N < 3000	average rotation cost of AFTreap + average search cost of AFTreap > average search cost of Treap
N = 3000	average rotation cost of AFTreap + average search cost of AFTreap = average search cost of Treap
N > 3000	average rotation cost of AFTreap + average search cost of AFTreap < average search cost of Treap

Conclusion:

In our experiments, we evaluated and compared the performances of the standard Treap versus the modified Access Frequency Treap. Although we expected the Access Frequency Treap to have an advantage over the standard Treap given input of any range or distribution, we found that it only performed better given zipf-distributed accesses and a high range of values. The cost of the upward rotations only pays off when the same handful of elements among a large pool are accessed more of the time. More frequently accessed elements are pushed up the tree, since a new priority is generated each time it is looked up (and only assigned if it is greater than the existing priority). Therefore, subsequent lookups become much faster. In the long term, the AFTreap is doing much less of the costly upward rotations because the most accessed elements are already at the top. With the uniformly distributed accesses, the standard Treap consistently performed better. Since all numbers had an equal chance of being accessed, rotating them up the tree had no real benefit. Therefore, in deciding whether or not to use an Access Frequency Treap, we should consider the kind of input we have and whether or not the cost of rotating elements up the tree will pay off in the long run.

Our experiment assumed an environment in which no new values is added, nor is any node deleted. This assumption is not realistic in applications. These operations can disrupt the optimal arrangement of nodes the AFTreap creates after many calls of search() method. Thus, we think with more calls of insert and delete, the AFTreap will perform worse than in this experiment. In a future experiment, we can explore how

different patterns of insert, delete, and search can influence the overall runtime of an AFTreap, as compared to a standard Treap.

We proposed a hypothetical relationship between average rotation cost of AFTreap, average search cost of AFTreap, and average search cost of Treap under different ranges. We wonder how is the number of nodes in a tree is related to the range, and thus related to the three costs. We want to establish a more mathematically accurate relationship and run experiments to prove the relationship.