Implementation and analysis of several data structures

Nuno Neto
up201703898

May 13, 2021

Advanced Topics in Algorithms
Masters in Computer Science

1 Introduction

In this paper, we will be discussing various data structures, their ideas and goals, their complexities and their basic empirical running and memory costs of the implementations that were developed for this assignment. The data structures that will be discussed are:

- AVL Trees
- Red Black Trees
- Splay Trees
- Treaps
- Skip Lists (Concurrent and Sequencial implementations)
- Bloom Filters (Concurrent and Sequencial implementations)

2 Binary Search Tree based structures

The binary search tree based structures that will be discussed here are auto balancing. Auto balancing BSTs fix the inherent defect of imbalance in a BST with dynamic sets of information, making them an ideal structure for indexing large amounts of comparable data (Data that cannot be compared cannot be represented using a BST) and still maintain very efficient insert, lookup, delete operations with a complexity of $O(log(n))$ along with other operations like min and max that can be performed in $O(1)$, among other operations.

To maintain this balance, we use two constant time operations called rotations. There are 2 possible rotations on a given node X, a right rotation and a left rotation. These rotations can then be compounded and produce complex movements.

All binary trees that were implemented have the following functions:

- add (key, value) - Add a key value pair to the tree. $O(log(N))$
- hasKey (key) - Check if a given key is contained in the tree. $O(log(N))$
- get (key) - Returns the associated value to the key, if it's present. $O(log(N))$
- remove (key) - Removes the key from the tree and returns the associated value, if the key was in the tree. $O(log(N))$
- keys() - Get a vector containing all the keys in the tree (in ascending order). $\Theta(n)$
- values() - Get a vector containing all the values in the tree (in ascending order of their associated keys). $\Theta(n)$
- entries() - Get a vector containing all the key value pairs (in ascending order of the keys). $\Theta(n)$
- rangeSearch(base, max) - Get a vector containing all the key value pairs within the range [base, max] (in ascending order). The complexity of this operation will depend on the base and max parameters, the worst case is when $base <= peekSmallest() <= peekLargest() <= max$ and has a complexity of $O(n)$.
- peekSmallest() - Peek the minimum value in the tree. $O(1)$
- peekLargest() - Peek the largest value in the tree. $O(1)$
- popSmallest() - Pop and return the minimum value in the tree. $O(log(N))$
- popLargest() - Pop and return the largest value in the tree. $O(log(N))$

2.1 Implementation of the operations

All the BST approaches share the same base implementation of these operations, implemented on the BinarySearchTree class. In this section we will discuss how these operations are implemented and their complexity. If a particular implementation of a BST changes the way these operations are implemented, we will discuss the implementation in it's corresponding section.

2.1.1 Contains and get operations

Both the contains and get operations are based on finding a given key in our tree.

The find operation is implemented as follows, starting at the root node:

- Check if the node key is equal to the key we are searching, if so return.
- Check whether the key is larger than the node key, if so we recursively visit the right child, if not then we recursively visit the left child.
- If the child we mean to visit does not exist, then the key is not contained in the tree and we can return.

The complexity is determined by the height of the tree so given that our implementations are balanced, we have a worst case complexity of $O(log(N))$.

2.1.2 Add operation

The add operation performs a find operation on the key that we want to add to the tree. If the find operation returns a node, then the key is already in the tree so we just replace the value associated with it to the new value that we have received. If it does not find the node, then we get the last node that was visited (which has to be a leaf node), compare it's key $leafKey$ with the key we want to add. If the key we are adding is larger than $leafKey$, we make it the right child of the leaf if not then we make it the left child.

The complexity of this operation is determined by the height of the tree, so as all the BSTs implemented in this paper are **balanced**, we know that this operation has a worst case complexity of $O(log(N))$.

2.1.3 Remove operation

The remove operation starts by performing a find operation on the key.

If the node is contained in the tree, then we will remove it by performing the following operation:

- If the node that is to be removed has no children (it's a leaf node), then we just remove it from the tree.
- If the node that is to be removed only has one children (left or right), then we remove the node and replace it with it's child and the operation is done.

- If the node has two children, then we get the left most node (that is a leaf) of the right sub tree (so the smallest node that is larger than the key that we want to remove). We then replace the key and value of the node we want to remove with the key and value of the leaft most node in the right sub tree. After performing this replacement, we get a situation where there are 2 nodes that share the same key. To solve this, we perform a remove on the right sub tree for the repeated key which since the node is a leaf will remove it as per case 1.

Again we see that the complexity of this operation is determined by the height of the tree so since we are working with balanced BSTs the worst case complexity is $O(log(N))$.

2.1.4 Keys, Values and Entries operations

To implement these operations, we perform an in-order traversal of the tree. This traversal visits all of the nodes in the tree once in order (as the name states) by first visiting the left sub tree then the root node and afterwards the right sub tree. Since it always has to visit all the nodes in the tree, we have a complexity of $\Theta(N)$.

2.1.5 Peek Smallest and Largest Operations

To implement these operations, we store a reference to the largest and to the smallest node. This reference gets updated on every add and every remove operation (This just requires a comparison and swapping a value, so it's $O(1)$, meaning we maintain the $O(log(N))$ complexity on those operations, even with this addition). When we want to peek the nodes, we just use the references that we have stored originating a complexity of $O(1)$.

2.1.6 Pop Smallest and Largest Operations

To implement these operations, we use the references to the smallest and largest node as a shortcut. Since we know that the smallest and largest nodes have at most one child (The smallest node always has to be the left most node in the tree and the largest node always has to be the right most node meaning they can only have a right child and left child, respectively) we can remove it directly using the second case of the remove operation (Section 2.1.3).
After this, we have to replace the reference to the node by the now new smallest node or largest node, respectively. We will describe the operation for the smallest node, as the operation for the largest node is analogous.
First we start by checking if the smallest node has children. If it has children (has to be right children) then the left most node in the tree is the left most node in the right sub tree of the node. If the node has no children, then the new smallest node is it's parent (as that becomes the left most node). This originates a worst case complexity of $O(log(N))$.

2.2 AVL Tree

To maintain balance AVL trees keep track of the difference in height between each of it's children, known as the balance of the node.

$$B(X) = Height(RightSubTree(X)) - Height(LeftSubTree(X))$$

where $B(X)$ is the balance of the node.
A node that is balanced has a $B(X) = 0$, any value different from 0 means that the node is unbalanced. However, if we have a node that is unbalanced by 1 then we can never achieve a balanced state ($B(X) = 0$), as the number of nodes in the sub tree is odd. To reflect this, AVL trees only require a rotation whenever the $B(X) > 1$ or $B(X) < -1$. To decide the type of balancing we need to do we will look at the sign of $B(X)$.

2.2.1 Balancing types

We have two possible types of balancing: Left balancing and right balancing. To decide what case to apply for each node, we have to look at $B(X)$. If $B(X) < -1$ then we know that the right subtree is heavier, so we will require a left balancing. Analogously, if $B(X) > 1$ then we know that the left subtree is heavier, so we'll require a right balancing.

However, each type of balancing also has 2 possible cases which are solved by applying the corresponding rotations (at most 2 rotations are required at a given insert in an AVL tree). We will describe the possible cases for the left balancing type, as the right balancing is analogous:

- The left sub tree is balanced. In this case, we only need to apply a right rotation on the node X to fix the unbalance.
- The left sub tree is right heavy ($B(LeftSubTree(X)) > 0$). To solve this case, we must apply a left rotation on $LeftSubTree(X)$ followed by a right rotation on X.

We only require 2 rotations on any insertion/deletion because with a single operation we can at most unbalance a single node, which can always be solved by at most 2 rotations.

2.2.2 Operation complexity

We have seen that the complexity of the operations described earlier depends on whether the balancing implementation is capable of maintaining a tree height that is in the order of $log(N)$, which is guaranteed by the height limit of $1.44 * log(n)$ that is assured by the AVL tree.

Since we have seen that for any given change to the members of the tree, we require at most 2 rotations which is $O(1)$, then we can conclude that the complexities described in Section 2.1.

2.2.3 Advantages and disadvantages

Some advantages of using AVL trees:

- Searching, inserting and removing are guaranteed a worst case of $O(log(N))$
- Since the tree maintains a maximum height level of $1.44 * log(n)$, searching the tree will be very efficient.

Some disadvantages of using AVL trees:

- The implementation is complex.
- We require at least an extra two bits of memory per each node.
- Inserting and removing from the tree is less efficient since we have to constantly rotate and change the tree structure in order to maintain the very small height limit, which also contributes to making the tree not cache or disk friendly.

2.3 Red Black Trees

To maintain balance, Red Black trees assign a color to each node (either red or black) and require a set of properties to always be maintained. The properties are:

- The root node is black - Root property
- The leaves are empty black nodes - Leaf property
- The children of a red node are black. - Red property
- For each node, a path to any of its descending leaves has the same number of black nodes - Black property.

2.3.1 Maintaining the properties

Given that we know a tree will only get changed whenever we are inserting or removing nodes we know that these are the only possibilities for breaking the properties.

When inserting, we do the following:

Create the node as a red node (with the empty black leaf nodes) and insert it into the correct position (Following the regular BST insert operation). Since the created node is red, we know that we haven't broken the black property. Since the tree is assumed not to be empty, we haven't broken the root property (if it is empty, just color the node black and make it the new root). Since we always have the leaf empty nodes as black nodes, we haven't broken the leaf property.

So we conclude that the only property that can be broken by an insertion is the red property. This happens when the parent of the node that was inserted

is red, which means that we now have two consecutive red nodes. When the parent of the node is black, the property is not broken so nothing needs to be done.
When the red property is broken, it has to follow one of 3 cases:

- When the uncle of the node is black and the node we have just inserted is the left child we right rotate the grandfather and then swap the colors of the parent and the grandfather.
- When the uncle of the node is black and the node we have just inserted is the right child we left rotate the parent and then perform the moves described in the prior case. (Right rotate the grandfather and then swap the colors of the parent and grandfather)
- When the uncle of the node is red we swap the colors of the parent, uncle and grandfather. So the parent and uncle would become black nodes with the grandparent becoming a red node. If the father of the grandparent is red, then that would break the red property, which can be fixed by aplying these cases.

When removing, we do the following:
Remove the node as described in the standard BST delete. As described in the method, we always remove either a leaf or a node with a single child. This means that the property that will mostly be violated is the black height property as if we remove a black node, the black height for that subtree will be altered. We refer to the node that is going to be deleted as D and the node that will replace it R (R can be null when D is a leaf node, as no other node will take it's place). There are various possible scenarios we have to account for:

- If either R or D are red, then we just mark R black, maintaining the black height property.
- If both are black, then the black height property will be violated so we colour the node R as double-black and then focus on fixing this double black into a single black. The deletion of a black non-null leaf will also cause a double-black (as leaves have themselves two null black leaf nodes).

To convert this double-black into a single black, we do the following function for an argument node N while the node is double black and is not the root:
Let the sibling of the node N be called S.

- If S is black and at least one of it's children is red (let's call this red child R), we will perform rotations according to the 4 possible sub cases:
    1. Left Left Case: Where S is the left child of its parent and R is the left child of S (or both children of S are red).
    2. Left Right Case: Where S is the left child of its parent and R is the right child of S.

3. Right Left Case: Where S is the right child of its parent and R is the left child of S.
4. Right Right Case: Where S is the right child of its parent and R is the right child of S (or both children of S are red).

- If S is black and both its children are black perform recoloring and apply the function to the parent of N and S if it is also black.
- If S is red, perform a rotation and recolor both the sibling and the parent. The rotation that needs to be applied follows these 2 possible scenarios:
  1. Left case: Where S is the left child of its parent, we right rotate the parent P.
  2. Right case: Where S is the right child of its parent, we left rotate the parent P.

2.3.2 Advantages and disadvantages

Some advantages of using Red Black trees:

- Searching, inserting and removing are guaranteed a worst case of $O(log(N)$.
- Uses little memory as each node only needs 1 bit to store it's color.
- Inserting and removing is fast as the properties are less strict, so we perform less rotations.

Some disadvantages of using Red Black trees:

- Since the tree maintains a maximum height of $2 * log(n + 1)$, searching is slower than with AVL trees.
- Very complex insert and remove operations.

2.4 Splay Trees

Splay trees were built to use the notion that in most real world applications, 80% of the accesses to a dataset occur on 20% of the data. In order to take advantage of this splay trees work by *splaying* the node that was most recently accessed to the root of the tree. *Splaying* a node means moving the node to the root using tree rotations so we maintain the BST properties. Doing this not only modes the node that was most recently accessed to the top of the tree, but also moves nodes that are close to that one further up the tree, so accesses to keys that are close to each other will in theory get better performance.

The operations are implemented like a regular BST, except we splay the node we accessed at the end of the access (this is done in find, insert and remove).

2.4.1 Splaying a node

To splay a node, splay trees use simple rotations similar to the rotations previously viewed in Red Black trees and AVL trees. However in Splay trees they are named differently, **Zig** for a right rotation and **Zag** for a left rotation. We can also have combinations of the two with Zig-Zig, Zag-Zag, Zig-Zag and Zag-Zig which combines the two simple rotations into a more complex rotation. To splay a node, we must recursively apply combinations of these rotations until the accessed node has reached the root of the tree. If the node has a grandparent, that means it's still at least 2 rotations away from becoming the root, so we apply 2 rotations. When the node does not have a grandparent it means that it's at most 1 rotation away from becoming the root of the tree. To decide which rotation to use, we first check whether we have a grandparent or not. If we do have a grandparent, we have to decide which of the 4 rotations (Zig-Zig, Zag-Zag, Zig-Zag, Zag-Zig) is going to be chosen. The 4 possibilities are:

1. Left left case: When the parent is a left child of the grandparent and the node is a left child of the parent. In this situation, we will apply a Zig-Zig rotation.
2. Left right case: When the parent is a left child of the grandparent and the node is a right child of the parent. In this situation we will apply a Zag-Zig rotation.
3. Right left case: When the parent is a right child of the grandparent and the node is a left child of the parent. In this situation we will aplly a Zig-Zag rotation.
4. Right right case: When the parent is a right child of the grandparent and the node is a right child of the parent. In this situation we will apply Zag-Zag rotation.

If we do not have a grandparent, then there are just two possibilities:

1. We are a left child of the root, which means we will need a Zig rotation.
2. We are a right child of the root, which means we will need a Zag rotation.

By recursively applying these rotations to the node we want to splay it will rotate the tree until we have the node at the root.

2.4.2 Complexity of splaying

By splaying a node, we can come to a situation where we have a tree that does not have a height of $O(log(N))$, but instead $\theta(n)$. In this case, accessing and splaying the deepest node in the tree would take $\theta(n)$ time, resulting in a worst case complexity of $O(n)$ for insert, contains and remove. However, when we perform a splay operation on the deepest node, we will then get a tree that has an average node depth of roughly half compared to the tree before the access.

To get an accurate measurement of the complexity in splay trees, we must use Amortized Analysis. We will not discuss how this analysis is done, but by employing it the complexity we obtain for m operations on a given splay tree will result in a complexity of $O(mlog(n))$ which averages $O(log(n))$ complexity per operation.

2.4.3 Advantages and disadvantages

Some advantages of using Splay trees:

- No memory overhead on each node for storing state.
- If our use case follows the 80% of accesses accessing 20% of the dataset, we will see visible performance improvements.

Some disadvantages of using Splay trees:

- Randomized accesses might lead to very bad (Near linear) performance.
- We can get situations where we have $\theta(N)$ operations, instead of a constant worst case $O(log(n))$.

2.5 Treaps

A treap is a probabilistic data structure where every node has both a key and a priority. Treaps maintain both the properties of a BST according to their keys and of a min heap according to their priorities. Maintaining the properties of a min heap means that for every node N,

$$priority(N) < priority(LeftChild(N)) \text{ and}$$
$$priority(N) < priority(RightChild(N))$$

or in other words, for any tree the node with the highest priority (in this case highest priority means smallest $priority(N)$ is located at the root of the tree). Maintaining the properties of a BST means that similarly to all other BSTs, for any node N,

$$key(N) > key(LeftChild(N)) \text{ and } key(N) < key(RightChild(N))$$

. To achieve this property treaps use rotations, similarly to other balanced BSTs to maintain the heap property, as the BST property is assured by performing regular BST insert. The BST property does not get broken when performing any type of rotation.

The way this BST maintains its balance is by assignign each node a random number sampled from a random uniform distribution. This means that the tree is not guaranteed to be balanced but there is an overwhelmingly high probability of it being so meaning the expected height of any treap is $O(log(N))$. However, when a new node with a random probability is inserted into the tree, we have to heapify that node to assert the heap property is maintained.

2.5.1 Heapifying a node

As we have said before when inserting a node into a tree we perform a regular BST insert operation and then we heapify the node to maintain the heap property. We have already seen in Section 2.1 the insert operation depends on the height of the tree which is expected to be $O(log(N))$ for any treap so we know the insert operation has a complexity of $O(log(N))$. We will now evaluate the complexity of the heapify operation to make sure the insert operation is still $O(log(N))$.

We know that the heap property means that the highest priority has to be at the root, so in the worst case a new leaf we have just inserted is the highest priority node meaning it has to be moved to the root of the tree. It is trivial to see that since the height of the tree is expected to be $O(log(N))$, then the amount of rotations we have to perform to reach the root of the tree from a leaf is also $O(log(N))$ meaning the heapify operation has a complexity of $O(log(N))$.

2.5.2 Advantages and disadvantages

Some advantages of using treaps:

- Very easy to implement compared to some of the other balanced search trees.
- Order of insertion and removal of items is not determining of tree shape and therefore does not influence performance.

Some disadvantages of using treaps:

- Complexity of operations is not guaranteed, it is just expected.
- Using a poor random uniform distribution might lead to unbalanced trees.

3 Skip Lists

Skip lists are a probabilist alternative to balanced BSTs. Skip lists are basically linked lists with layers of shortcuts that get exponencially less populated by repeatedly flipping a coin until we obtain tails for every node in the list so for example the first layer of shortcuts will have around $\frac{n}{2}$ elements while the second layer will have around $\frac{n}{4}$ the third around $\frac{n}{8}$ and so on so forth. These shortcuts allow us to significantly reduce the amount of steps required to find a certain key in the list by roughly a factor of 2 per added shortcut level so in theory with a limitless amount of shortcut levels we would be able to achieve $O(log(N))$ search time for any N. In practice though the amount of shortcut levels is usually limited for performance reasons but it still allows for $O(log(N))$ operations on realistic data sets with millions of entries (the probability of a node getting level 20 is 0.000000954 meaning on average it would require millions of nodes to achieve a single node in 20th level).

3.1 Implemented operations

We have implemented the exact same operations that were described in Section 2.1.

3.2 Implementation of the operations

Almost all operations in skip lists require performing the find operation (Searching, adding, removing, etc) so if the complexity of the find operation is $O(log(N))$, then we can conclude that the complexity of those operations is also $O(log(N))$ as they just perform constant time operations after finding the correct position.

3.2.1 Find operation

The find operation consists of, for every level, finding the smallest node that is larger than the key that we are looking for. When it is found for a given level we store the previous node (the largest node that is smaller than the key we are searching for) and then descend into the lower level, starting at the stored node and repeat this operation until we reach the base level and we either find the key or not. Intuitively, since each level of the skip list has about half the nodes of the level bellow it the total number of levels should be around $O(log(N))$. Similarly, with each new level of random shortcusts we cut the search space in roughly half so $O(log(N))$ levels should give us an expected search time of $O(log(N))$.

This operation is used for both checking if a certain key is in the map and returning its associated value if it is.

3.2.2 Add operation

The add operation first performs a find on the key we want to add. If the key is found we just replace the associated value. If the key is not found, we will have to add the new node into the list. First we determine the level for this new node by flipping a coin until we obtain tails. Given this level, we use the information from the find operation (namely the largest node that is smaller than our key per level) and add this new node up to its level. Since in this implementation the level of the node is capped we can assume this is a constant time operation so the complexity of the operation is majored by the find operation making the add operation $O(log(N))$.

3.2.3 Remove operation

The remove operation, similarly to add, first performs a find on the key we want to remove. If the key is not found then we cannot remove it and the operation is finished. If the key is found then we will remove the node on every level the node is on (using the information of the largest node that is smaller than that key collected by the find operation). Again, since the amount of levels is

capped this operation is constant time so the complexity is majored by the find operation making the complexity $O(log(N))$.

3.2.4 Keys, Values and Entries operations

These operations are implemented by simply traversing the base level of the list as the list is ordered. This traversal visits all of the nodes, which originates a complexity of $\theta(N)$.

3.2.5 Peek smallest and largest operations

These operations are implemented similarly to the BST version by storing a reference to the largest node (as the smallest node is always the first node in the list so we don't have to keep another reference to it). These references are kept updated on every add / remove so when we need to peek the node it's always a constant time operation so complexity is $O(1)$.

3.2.6 Pop smallest and largest operations

These operations are implemented by just calling the remove operation on the reference to the largest/smallest node which is kept as a reference. Since the remove operation has a complexity of $O(log(N))$ these operations will also have a complexity of $O(log(N))$.

3.2.7 Advantages and disadvantages

Some advantages of using skip lists:

- Great performance on insertions as no rotations or reallocations are required.
- Simple to implement and extend.
- Retrieval of next element in constant time is simple.
- Easy and efficient to parallelize without recurring to a global lock.

Some disadvantages of using skip lists:

- Being a probabilistic data structure operation complexities are not guaranteed for every case.
- A bad coin flip entropy can result in a poor skip list.

4 Bloom Filter

A Bloom Filter is space efficient probabilistic data structure that is used to test whether an element is a member of a set. Unlike the other data structures described this data structure does not garantee that an element belongs to the set even when it returns that it does belong (false positives). We cannot however obtain false negatives. In other words:

When testing if an element is a member of a set using a bloom filter a **yes** response means that it **may** belong to the set however a **no** response means that the element **definitely** does not belong to the set.

A bloom filter supports adding new elements and checking if an element is contained in the set but by default bloom filters do not support the removal of elements from the set.

This type of data structure allows us to quickly filter out elements that are known to not be in the set preventing us from having to search less efficient structures like BSTs and Skip Lists. The fact that this structure also uses a constant memory allows us to deploy it to systems without a lot of memory for example filtering certain internet traffic at routers that have a small memory footprint and also have to handle large amounts of traffic quickly.

To achieve these properties, bloom filters use hash functions to calculate positions for each element. A bloom filter can use 1 or more hash function, usually more than one to reduce the chance of false positives. For an element to be present inside a bloom filter all of the positions that are generated by the hash functions have to be set to 1 (just 1 of the positions being set to 0 indicates that the element cannot be present whithin the set).

The chance of getting a false positive is dependant on the size of the bloom filter, the number of hash functions used and the amount of items stored in the bloom filter. If the number of items increases, so does the probability of a false positive. If the size of the bloom filter is reduced, the false positive rate also increases.

4.1 Hash functions used

In this bloom filter there were 2 separate hash functions used however these can be expanded into any amount by seeding the input of the hash functions, which generates a completly different output that can be used as if it were an entirely new hash function. The base functions used where MurmurHash and SpookyHash and the default number of hash functions used is 3, customizable when initializing the bloom filter.

4.2 Add operation

To add an element to a bloom filter we go through all of the hash functions and hash the element to obtain the positions we have to set to 1. After we have these positions, we set them all to 1 and the element is inserted. This has no

correlation to the current amount of items that is contained in the set and takes a constant amount of time. The complexity is $O(1)$.

4.3 Test operation

To test if an element is in a bloom filter we perform the same hash functions as in the add operation however instead of setting the values to 1, we check their current value. If any of the positions is not set to 1, then we know the element cannot possibly be in the set. If all the positions are set to 1, then we return true as the element **may** be in the set. Like in the add operation, this has no correlation to the amount of items stored in the bloom filter and is therefore constant time. The complexity is $O(1)$.

4.4 Advantages and disadvantages

Some advantages of using bloom filters:

- Fast insert and test speeds, dependant on the hash functions used and how many hash functions were used.
- Constant memory usage, independent of the amount of items stored in the bloom filter.

Some disadvantages of using bloom filters:

- Increasing the amount of items in the bloom filter increases the false positive chance
- No guarantees that a certain element belongs to the set stored in the bloom filter.

5 Performance Analysis

To analyse the performance of the data structures we used tests that mixed insert, lookups and removes to assess the speed of the data structures. We used tests with more inserts/deletes and tests with more lookups, along with varying orders (ascending, descending and randomized). For a given test size T the insert heavy tests (in ascending and descending order) are structured as follows:

- Insert $3 * T$ elements according to the order of the test.
- Perform T lookups.
- Delete T elements from the data structure according to the order of the test.
- Insert another $3 * T$ elements according to the order of the test.
- Perform another T lookups.

- Delete T elements from the data structure.

The lookup heavy tests (in ascending and descending order) are structured as follows:

- Insert T elements into the data structure.
- Perform $5 * T$ lookups on the data structure. (Repeat lookups from $0..T$ or $T..0$ depending on the order, 5 times)
- Insert another T elements into the data structure.
- Perform $10 * T$ lookups on the data structure. (Same as point number 2, but from $0..2T$)

The randomized insert heavy tests are structured as follows:

- First generate $2 * T$ random elements using a constant seed so we can always get the same generated numbers.
- Insert the T first elements into the data structure.
- Perform T operations using the final T generated elements where if the number is even, insert it into the data structure, if not remove it. If the number $\%3 === 0$ then also perform a lookup on the number.

This test is insert/remove heavy as all numbers generate either an insert or a remove, but only numbers that are a multiple of 3 generate a lookup.

The randomized lookup heavy tests are structured as follows:

- First generate $2 * T$ random elements using a constant seed so we can always get the same generated numbers.
- Insert the T first elements into the data structure.
- Perform T operations using the final T generated elements where if the number is even, then perform a lookup with that number. If it is not even, then check whether the number $\%3 == 0$, if so then remove that number, if not then insert that number into the data structure.

5.1 Results

The tests were performed with test sizes of 10^2, 10^3, 10^4, 10^5, 10^6, 10^7.

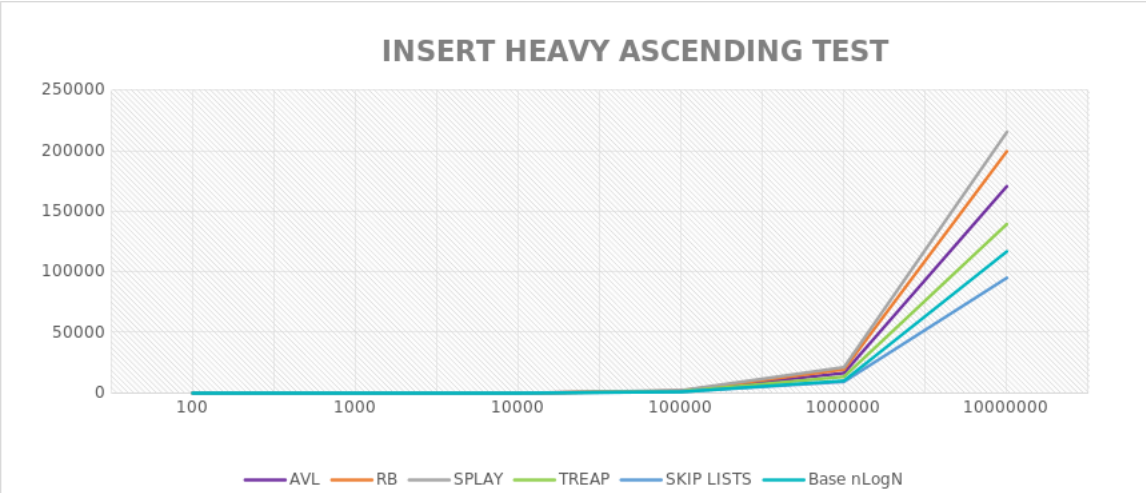

Figure 1: Insert Heavy Ascending order test results

5.1.1 Insert Heavy Ascending Test

Some notes on the results of the test in Figure 1:

- All data structures have a similar performance, following in line with the expected $O(Nlog(N))$ runtime.
- Splay trees are the slowest data structure, which makes sense since they excel at acessing a small portion of the data for most the accesses which is not the case.
- Skip lists are the fastest data structure given they are the most efficient for insert operations as they do not require rotations or changes of structure.
- Treaps perform better than AVL and RB trees because the test is done is ascending order, AVL trees and RB trees need lots of rotations while the Treaps do not care about insertion order.

5.1.2 Lookup heavy ascending order

Some notes on the results of the test in Figure 2:

- All data structures except for the splay have a similar performance, in line with the expected $O(Nlog(N))$ runtime.
- Splay trees are by far the slowest, which again makes sense as we are performing a lot of lookups on an ascending data set, which requires lots of splaying leading to degraded performance.

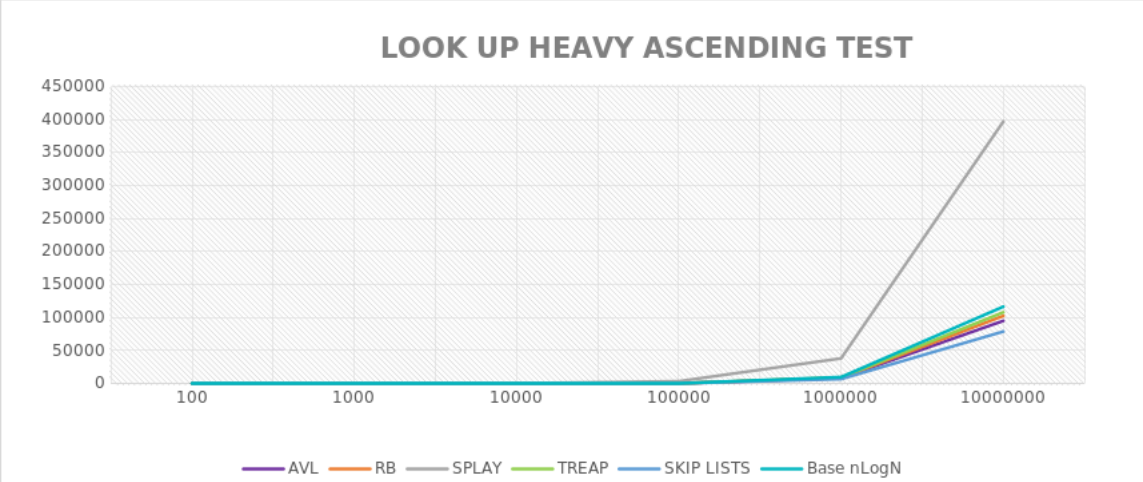

Figure 2: Lookup Heavy Ascending order test results

- Skip lists are still the fastest, but they lost a lot of the lead vs the balanced BSTs a lot due to the fact that we no longer perform so many inserts which the skip lists excel at.

5.1.3 Insert heavy descending order

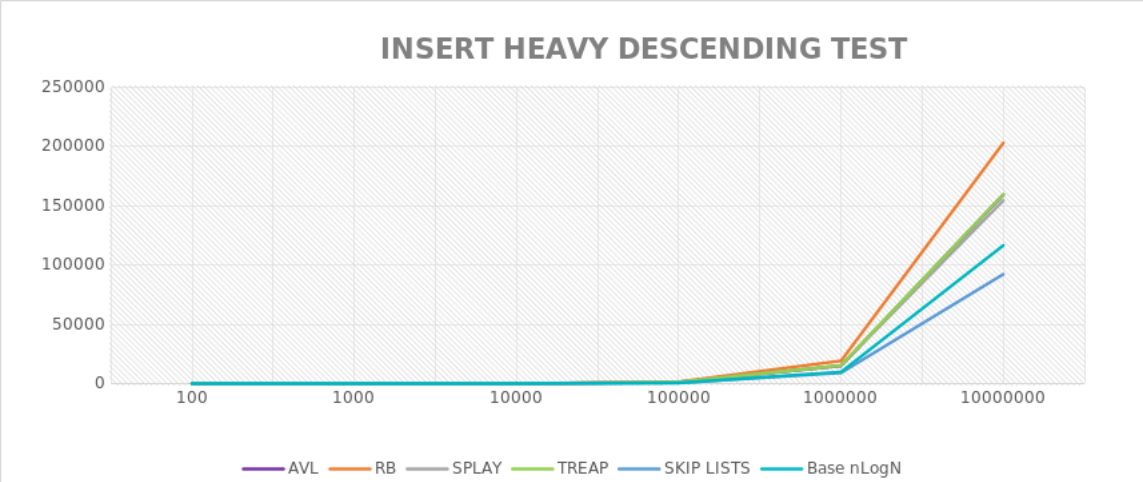

Figure 3: Insert Heavy descending order test results

Some notes on the result of the test in Figure 3:

- All data structures have a similar performance, following the excepted

runtime of $O(Nlog(N))$.

- Skip lists are again the fastest, since they are the most efficient for insert operations as they do not require rotations.
- Splay trees are again by far the slowest, as we do not exploit the property that they excel forcing them to perform a lot of splaying.

5.1.4 Lookup heavy descending order

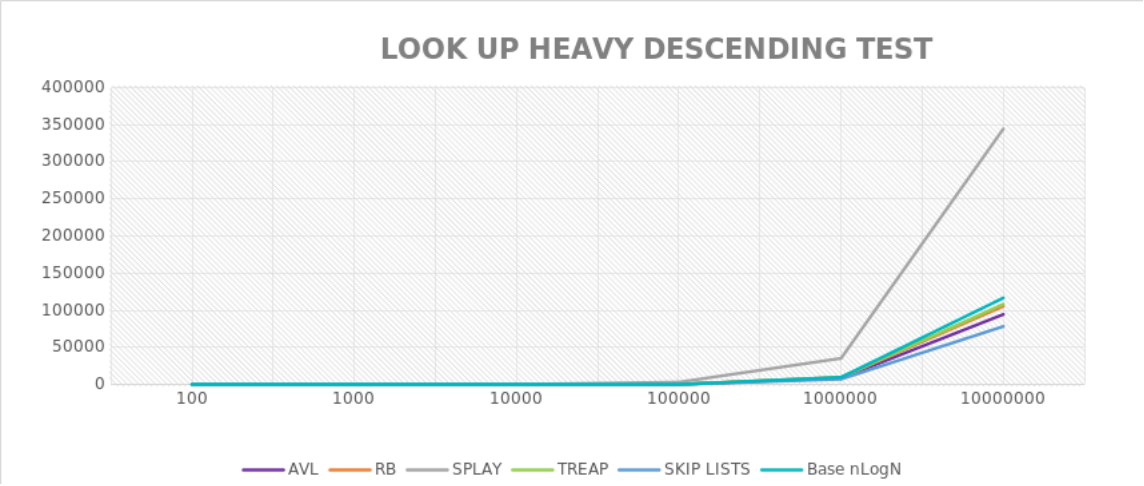

Figure 4: Lookup Heavy descending order test results

Some notes on the result of the test in Figure 4:

- Similar to the ascending order lookup heavy test, we have all but Splay Trees following the expected runtime of $O(Nlog(N))$.
- Splay trees are still the slowest data structure as the test does not follow the property splay trees excel at.

5.1.5 Lookup heavy randomized order

Some notes on the result of the test in Figure 5:

- We again see all data structures but the splay tree following in line with the $O(Nlog(N))$ runtime.
- Splay trees are again the slowest as they do not handle uniform random distributions very well.

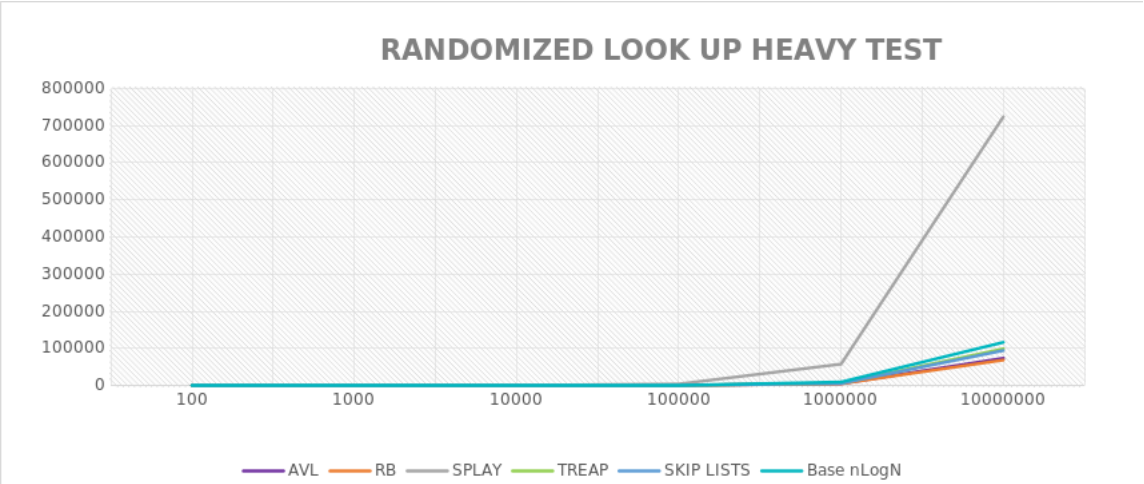

Figure 5: Lookup Heavy random order test results

- Skip lists lost the lead they had in the ascending order insert heavy tests, as expected since this test does not highlight its great insertion capabilities, since it is mostly made up of lookups.
- We see that RB trees and AVL trees have caught up and overtaken the Treap which makes sense seen as inserting in random order reduces the amount of rotations that would be required (inserting nodes in ascending/descending order is the worst possible case for AVL and RB trees as it normally creates trees with height of $\theta(N)$ if not externally balanced).

5.1.6 Insert heavy randomized order

Some notes on the results of the test in Figure 6:

- The scenario is repeated as all but the splay tree are in line with the expected $O(Nlog(N))$ runtime.
- We see a repetition of splay trees not handling uniform distributions well at all.
- Like the previous randomized order tests, RB trees and AVL trees have overtaken the Treap as we are no longer inserting in ascending/descending order.

5.2 Bloom Filter performance

To test the performance of bloom filters, we perform T inserts and then perform $2 * T$ contain tests.

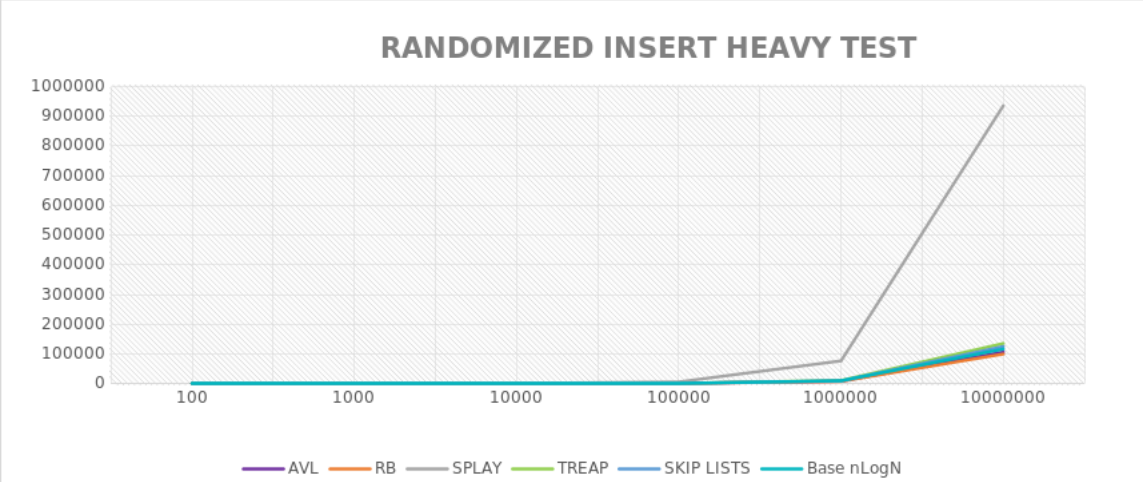

Figure 6: Insert Heavy random order test results

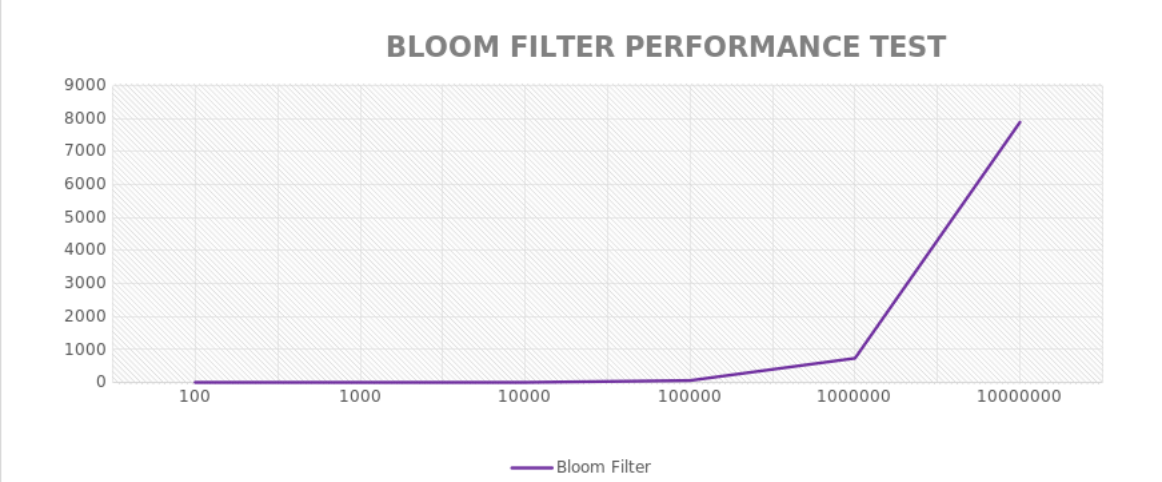

Figure 7: Bloom filter performance results

As we can see from Figure 7 the runtime grows linearly with the amount of operations performed as is expected given the constant time execution of the add and test operations.

6 Concurrent Data structures

As an extra to this assignment we implemented a concurrent lock free skip list and a concurrent bloom filter.

6.1 Concurrent Skip lists

To implement the concurrent skip lists, we added two flags for each node that serve to make insert and delete operations appear atomic. These flags are *fullyLinked* to make insert operations appear atomic and *marked* to make removal operations appear atomic. All nodes start with *fullyLinked* set to true and it gets set to false when we are done inserting the node. When we want to remove a node we first set *marked* to true to mark the node for removal. We also added a lock to each node so we can assure mutual exclusion when changing node information.

We will not go further in depth into the implementation of these data structures as they are not the subject of this paper, just an extra addition.

6.2 Concurrent Bloom Filters

To implement the concurrent bloom filters we added a lock for every 64 bit block of information to assure mutual exclusion when changing the values in the nodes. We do not require mutual exclusion when verifying if a node is contained in the set as if we check if it is contained when another thread is adding to the bloom filter we treat it as if the insert operation was not completed, since adding a node does not influence the bloom filter in any way that is illegal to its operation (it does not generate at any point any false negative and it does not generate any false positive that wouldn't also be generated by adding the node atomically).