Project 2

Treaps

Anthony Lupica

arl127@uakron.edu

Introduction

What is a treap? It's a "tree-heap," a structure that combines the binary search tree and heap. Recall that a BST follows that the "key" of any given node N, with children N_L and $N_{R,}$ satisfies the totally-ordered property: $N_L < N <= N_R$. A heap, on the other hand, is partially ordered. This means we define a relationship that only concerns the ordering for a node and its descendant and does not apply across sibling nodes. There are two variations: a min-heap and a max-heap. These utilize a "priority" value such that for a min heap, each node stores a priority that is numerically less than or equal to that of its children, and for a max heap, each node stores a priority that is numerically greater than or equal to that of its children. In both cases, the root node should store either the minimum or maximum value possibly assigned as a priority, respectively.[3]

Let's take a closer look at these properties individually…

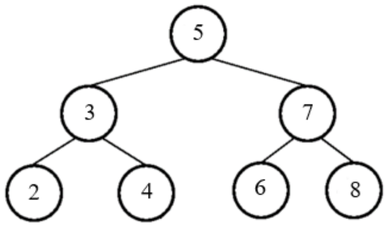


Figure 1

A BST is shown that has 7 keys inserted. Keys enter through the root, veering left or right in order to find their proper site of insertion.[2]

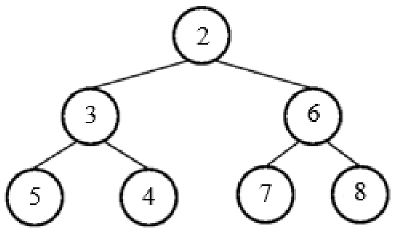


Figure 2

Using the same key values from figure 1 as priorities in a min-heap, figure 2 shows the resultant tree after heapification. See that subtrees are now ordered vertically, not horizontally.[2] The priority value that is the least is, for a min heap, considered to be of the highest priority.

We will examine what a treap looks like according to these two properties, how it operates, take note of what self-balancing is, and how it is managed. Our goal in the analysis of the treap is also to determine the benefit, or drawbacks, of combining these properties into a single structure, in terms of space overhead and how run-time scales with increased problem size, in the average, and worst cases. We will do this quantitatively by measuring the performance of the insert and search operations on a treap, versus that of a BST.

Discussion

Hypothesis: The treap implementation will scale significantly better than the BST for both the insert and search methods. The treap will exhibit little difference in runtime for “worst” and average case input.

The reason to care about the treap can be seen in the following depiction…

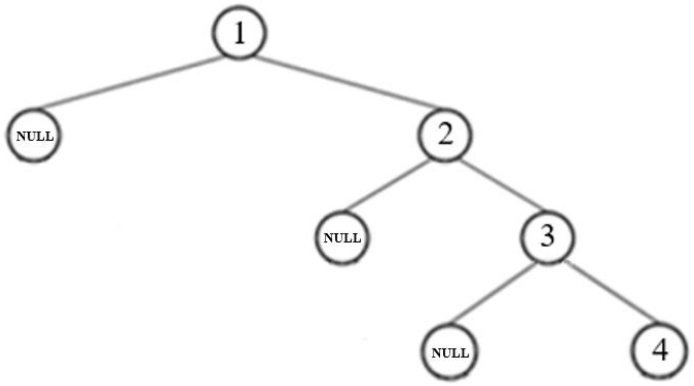


Figure 3

A valid BST. The <u>*order of insertion is relevant*</u> *to the shape of a BST. As the set of { 1, 2, 3, 4 } are inserted into the tree, the BST property causes the insert method to take all right child paths.*

As seen in figure 3, if the BST insertion process features keys in sequence that are all greater (or lesser) than the previous, the tree becomes unfortunately long in height. In fact, for n insertions, the BST can have a height as great as n (provided the root is counted as height 1).[4] In that case, the tree would be called “perfectly unbalanced,” which is the worst case. A worst case BST can have its operations run at O(n), or equal to the depth of the furthest node from the initial pointer, just as for a singly linked list.[4] The tree in figure 3 has effectively constructed itself into the form of a linked list, but with the wasted overhead of null-pointing left child pointers on each level. If this process were to continue in the same way, the tree would continue bypassing left children, and solely due to an unlucky insertion order chosen by the user.

A balanced tree is defined as a tree in which there are “roughly” an equal number of nodes in each subtree. Or, more rigorously, we say it is a tree where any given node’s subtrees have a height off by no more than 1.[5] Assuming the key inserts to a BST result, on average, in a tree of reasonably good balance, the height of a tree of n nodes can be expected to be roughly log n. That means operations on this reasonably balanced tree run in the more favorable O (log n) time. Thus, we would like our BST's overall height to be as short as possible.[4] If the same key values from figure 3 were instead inserted in the order { 3, 2, 4, 1 }, we get…

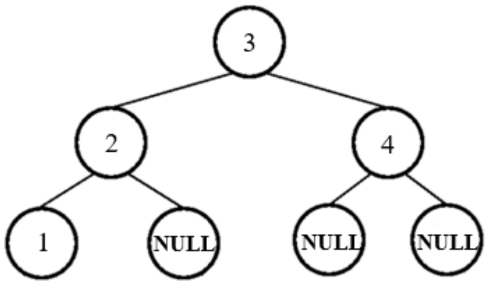


Figure 4

Balanced BST with height 3. This tree performs operations in O(log n).[2]

One of the few benefits of an unbalanced tree is O(1) insert if the key can be immediately inserted opposite the one-sided subtree. However, we may not want to accept this erratic behavior in runtime caused singularly by the insert order chosen by the user. The potential for unbalanced input to a BST is one of the structures' main concerns to be aware of. The treap data structure solves for this by abstracting away the user's concern for insert order, opting instead to implement self-balancing functionality to the BST. A treap will, in the average case, arrange itself into a balanced form by assigning random integer priorities across some range. In our implementation of a treap these priorities are used to maintain a min-heap ordering, in addition to satisfying the BST property.

Let's look at a visual representation of a treap containing integer keys and simulated random integer priorities. We will use the same keys as in figure 3, inserted in the same order to illustrate the self-balancing of a treap.

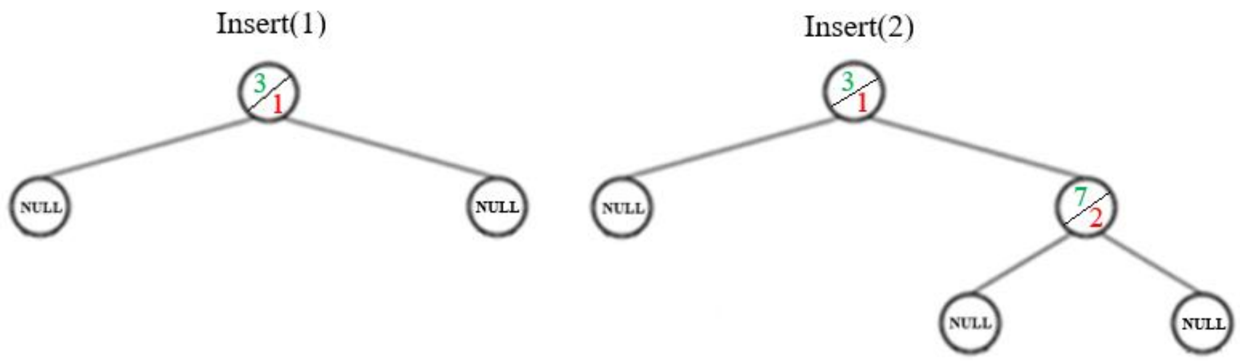


Figure 5

Here we see the key values 1, and then 2, are inserted according to BST rules. In this case, the "random" priorities assigned were 3 and 7 respectively. Following these operations, both the BST and min heap properties are satisfied.

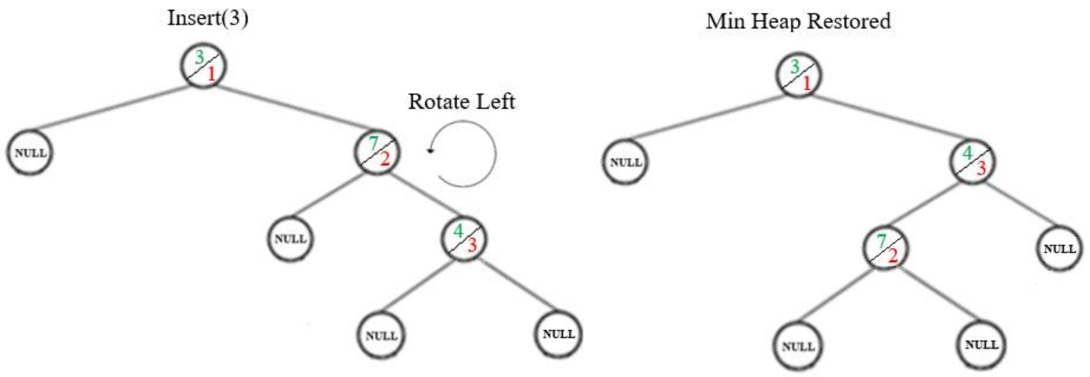


Figure 6

Now, key 3 is inserted following BST rules. However, this produces a min heap violation as the value assigned to this new node's priority was 4… which is of higher priority than its parent node according to min heap rules. We need to rotate the node containing 3 left with respect to its parent. Following this, it replaces the position which its parent node used to occupy in the BST. The old parent node containing 2 then slides down to fill in as the new parent's left child. Lastly, The node containing 3 "donates" its old left child (NULL in this case), to the demoted parent's right child. Performing these steps reshapes the tree to satisfy the min heap property, without violating the BST property.

In the case where instead the left child of some node were assigned a higher priority than its parent, a rotation right would be in order. Many combinations of left and right rotations may be necessary after an insert to restore the min heap order before another insert can be made.

Now we are ready to insert our final key value…

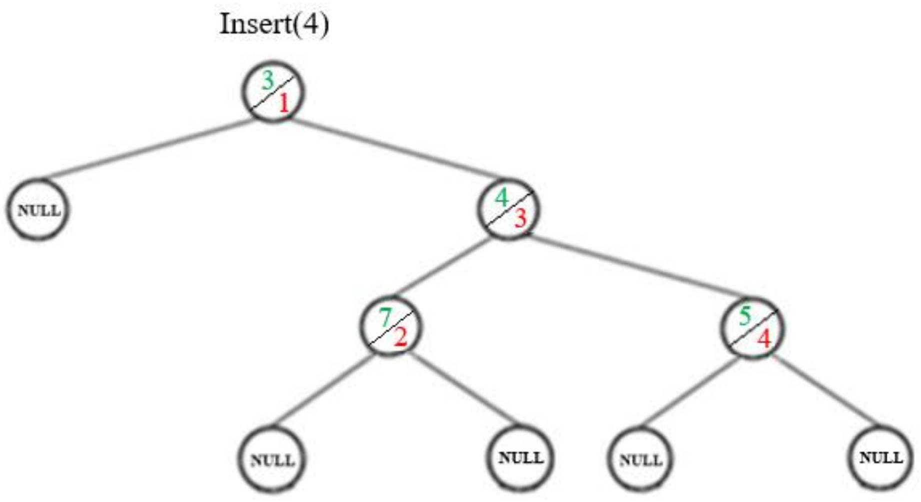


Figure 7

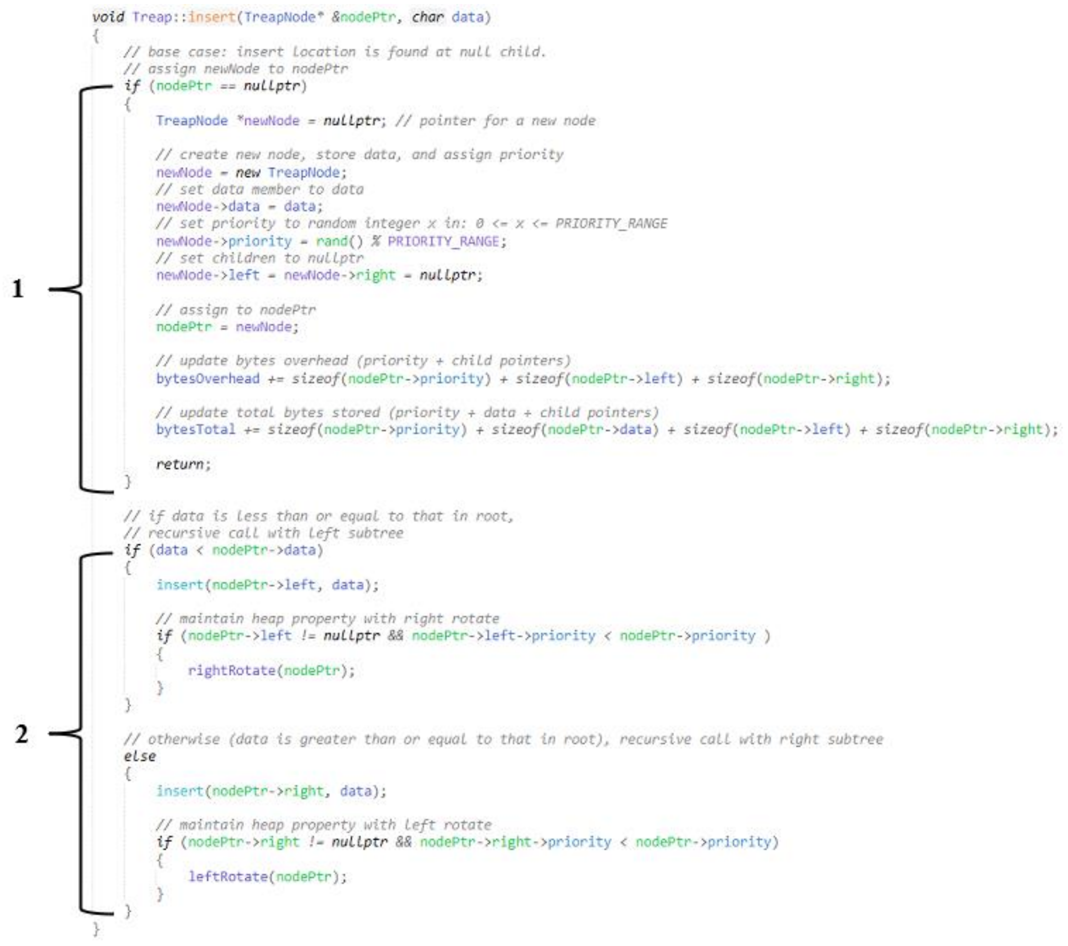
The priority-key pairing here is 5 and 4. The new node slots into the right of the node containing key 3. No min heap restoration is necessary as priority 5 is of lesser precedence than priority 4. Overall, we see key values in BST order, and priority values that ascend numerically as the tree descends, with the root node containing the least value numerically.

By utilizing random priority values in each node that must adhere to the min heap property, we were able to decrease the height of the original BST from figure 3 by 1, despite inserting the same key values in the same order. This is treap self-balancing at work.

Code and Analysis

The code for my implementation of a treap class uses the guidance of various resources, including the slides of unit 1 DS06 treaps[1], work by Techie Delight[6], and GeeksforGeeks[7]. It uses random integer priority values within the range 0 <= x <= PRIORITY_RANGE, where PRIORITY_RANGE is a globally defined constant of the class.

The insert method…



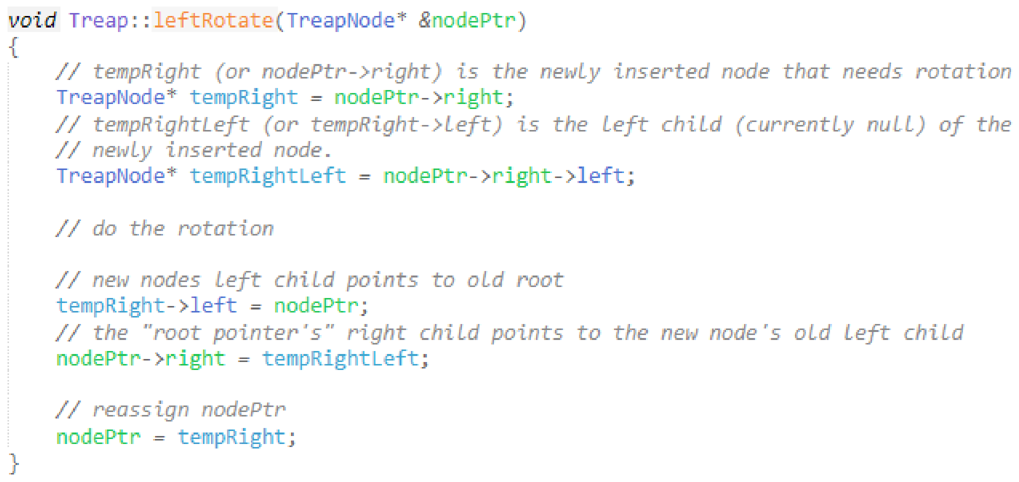
```cpp
void Treap::insert(TreapNode* &nodePtr, char data)
{
    // base case: insert location is found at null child.
    // assign newNode to nodePtr
    if (nodePtr == nullptr)
    {
        TreapNode *newNode = nullptr; // pointer for a new node

        // create new node, store data, and assign priority
        newNode = new TreapNode;
        // set data member to data
        newNode->data = data;
        // set priority to random integer x in: 0 <= x <= PRIORITY_RANGE
        newNode->priority = rand() % PRIORITY_RANGE;
        // set children to nullptr
        newNode->left = newNode->right = nullptr;

        // assign to nodePtr
        nodePtr = newNode;

        // update bytes overhead (priority + child pointers)
        bytesOverhead += sizeof(nodePtr->priority) + sizeof(nodePtr->left) + sizeof(nodePtr->right);

        // update total bytes stored (priority + data + child pointers)
        bytesTotal += sizeof(nodePtr->priority) + sizeof(nodePtr->data) + sizeof(nodePtr->left) + sizeof(nodePtr->right);

        return;
    }

    // if data is less than or equal to that in root,
    // recursive call with left subtree
    if (data < nodePtr->data)
    {
        insert(nodePtr->left, data);

        // maintain heap property with right rotate
        if (nodePtr->left != nullptr && nodePtr->left->priority < nodePtr->priority )
        {
            rightRotate(nodePtr);
        }
    }

    // otherwise (data is greater than or equal to that in root), recursive call with right subtree
    else
    {
        insert(nodePtr->right, data);

        // maintain heap property with left rotate
        if (nodePtr->right != nullptr && nodePtr->right->priority < nodePtr->priority)
        {
            leftRotate(nodePtr);
        }
    }
}
```

Figure 8

[1] The base case: when an empty space in the tree is found to insert data according to the BST order, a new TreapNode is allocated and its data members are set (null-pointing children, a random integer priority, and a character key set to the passed argument)

[2] The recursive calls: insert is called again, now with either the left or right subtree depending on if the character argument is less than or greater than the data present in the current node. After the insert, a check is made immediately to see if a right or left rotation is needed to maintain the min heap according to the new nodes priority.

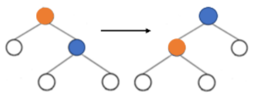
And now the leftRotate method, or one half of the heap managing methods in the class. The rightRotate method is effectively a mirror image, so we need only look at one…



```
void Treap::leftRotate(TreapNode* &nodePtr)
{
    // tempRight (or nodePtr->right) is the newly inserted node that needs rotation
    TreapNode* tempRight = nodePtr->right;
    // tempRightLeft (or tempRight->left) is the left child (currently null) of the
    // newly inserted node.
    TreapNode* tempRightLeft = nodePtr->right->left;

    // do the rotation

    // new nodes left child points to old root
    tempRight->left = nodePtr;
    // the "root pointer's" right child points to the new node's old left child
    nodePtr->right = tempRightLeft;

    // reassign nodePtr
    nodePtr = tempRight;
}
```

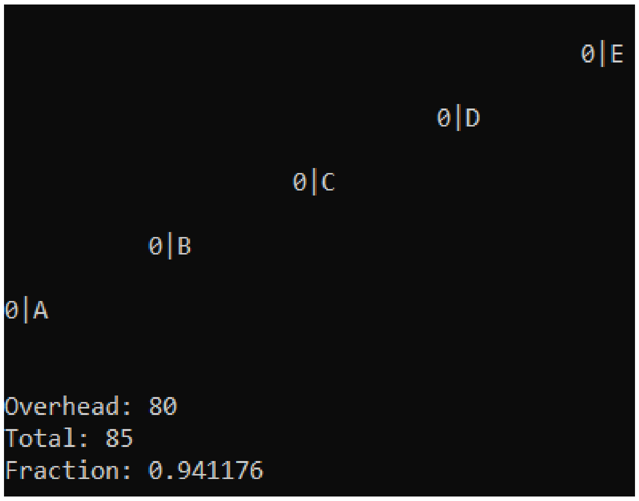
Figure 9

Declares temporary pointers for a swap. The goal is to leave the method with nodePtr pointing to the new node, with the left child of the old nodePtr. We also take the new node's left child and assign it to the old nodePtr's right child. Inserted node, and its parent are highlighted as they move for a left rotation.

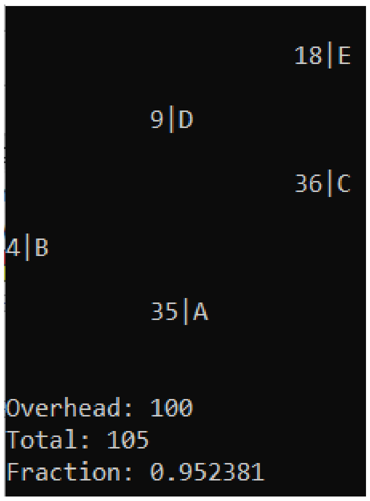


The other method of interest is search, but this operates in much the same way of a typical BST. However, mine returns the priority, if found, of the data requested to look for in the treap, and -1 otherwise.

My treap class implementation can be easily switched into a simple BST by removing the calls to rotate, or by forcing each node's priority to be of equal value so the calls don't trigger. Below is the output of the same sequence of inserts, { A, B, C, D, E }, by the program with, and without self-balancing. Recall this is a worst case insertion order for a BST, but our treap can't have bad input, so much as just an unlucky roll of random priorities.



```
                                        0|E

                              0|D

                    0|C

          0|B

0|A


Overhead: 80
Total: 85
Fraction: 0.941176
```



```
                    18|E

          9|D

                    36|C

4|B

          35|A


Overhead: 100
Total: 105
Fraction: 0.952381
```

Figure 10

The tree displayed is shown horizontally, with the root being the left most node. We see that without self-balancing, we get an identical output to what we would expect from a BST (left). The treap version shows a much different, shorter, and balanced form of the same collection of keys (right). This turned out to be a good set of random priorities for the sake of balance.

It's important to make the distinction that a treap is not incapable of shaping itself into a very unbalanced form. This would result if both the keys and priorities were sequential according to the BST and heap properties. Clearly, we can say that had I not forced the priorities in the left image to be 0, a random priority output of { 5, 20, 21, 40, 77 } would do just the same. As long as the priorities do not require any rotation, the tree will remain in the shape determined only by the BST property. The way to avoid this, to the best of our ability, is to use a wide enough range for priorities such that it is exceedingly unlikely that they will not require some rotating to restore the heap after insertions.

Something else to take note of here is the space overhead and total space required in units of bytes. For a BST, the space overhead typically includes two child pointers per node. This comes out to a total overhead of 2Pn, where P is the space required by a pointer, and n is the number of nodes. The total space required, which includes the data itself, is then n(2P + D), where D is the space requirement for a single data element. The overhead fraction can be found by 2P / (2P + D).[9] D for our purposes is 1 byte, as we are storing characters. On my 64-bit architecture, a pointer is stored as 8 bytes. This comes out to a startlingly high fraction of memory used to maintain the structure in both cases, and very little to store actual data. We do see a slight uptick in the treap version, as we would then include the integer priority in the calculation. That's n(2P + C) total overhead, where C is the space requirement of priorities, and n(2P + D + C) total space required. The overhead fraction for the treap is found with 2P + C / (2P + C + D). The overhead fraction of course depends on what size the data being stored is, and whether you use alpha or numeric priorities. No matter what, though, a treap will incur a slightly higher overhead cost than a traditional BST.

Test Cases

We will run the insert and search methods on both the BST and treap versions with worst case input, and "random" input. That is, input that is reasonably as sparse as what is on average produced for the random priorities of the treap. We will also see how these measures scale with increased problem size.

Worst Case			
BST		Treap	
Elements	Runtime (sec)	Elements	Runtime (sec)
10	0.000	10	0.000
100	0.000	100	0.000
1000	0.002	1000	0.000
10000	0.007	10000	0.015
100000	0.330	100000	0.032
1000000	62.843	1000000	1.224

Average Case			
BST		Treap	
Elements	Runtime (sec)	Elements	Runtime (sec)
10	0.000	10	0.000
100	0.000	100	0.000
1000	0.000	1000	0.000
10000	0.015	10000	0.015
100000	1.561	100000	0.046
1000000	Seg fault	1000000	4.796

Figures 11 and 12

The worst and average case runtime for the insert method in a BST and Treap. The method was scaled up to n = 1000000.

The worst case analysis inserted elements which were in ascending value, and the average case inserted elements which were randomly generated in the range 0 < x < 50.

We can see that across both cases, the treap performed significantly better, and scaled with ease. Due to the treap's self-balancing mechanism, it becomes irrelevant what and in what order the data is inserted. This enables it to perform well under more conditions than the BST.

Worst Case			
BST		Treap	
Elements	Runtime (sec)	Elements	Runtime (sec)
10	0.000	10	0.000
100	0.000	100	0.000
1000	0.008	1000	0.000
10000	0.015	10000	0.015
100000	0.345	100000	0.046
1000000	62.375	1000000	1.318

Average Case			
BST		Treap	
Elements	Runtime (sec)	Elements	Runtime (sec)
10	0.000	10	0.000
100	0.000	100	0.000
1000	0.000	1000	0.000
10000	0.014	10000	0.04
100000	0.333	100000	0.062
1000000	62.827	1000000	1.420

Figures 13 and 14

The worst and average case runtime for the search method in a BST and Treap. The method was scaled up to n = 1000000. In the worst case, the search method is called after each insert on the final element in the range. In the average case, the search method is called after each insert on some random element within the range.

Conclusion

The treap proves to be an effective structure to use when wanting to avoid O(n) operations, and the user cannot guarantee an insertion order that will produce a balanced tree. Overall, the treap does not sacrifice too much on additional overhead, and will trivialize worst case input by randomizing, and therefore balancing the shape of the tree with random priorities and using them to maintain the heap property.

References

1) Unit 1 DS06 Treaps Slides
2) Unit 1 DS05 Binary Trees Slides
3) Open DSA Assignment 5 - 05.05.05 Heaps and Priority Queues
4) Open DSA Assignment 5 – 05.05.04 Binary Search Trees
5) *Balanced binary tree*. Programiz. (n.d.). Retrieved March 14, 2022, from https://www.programiz.com/dsa/balanced-binary-tree
6) *Implementation of TREAP data structure (insert, search, and delete)*. Techie Delight. (2021, April 30). Retrieved March 15, 2022, from https://www.techiedelight.com/implementation-treap-data-structure-cpp-java-insert-search-delete/
7) Goel, A. *Print binary tree in 2-dimensions*. GeeksforGeeks. (2021, December 14). Retrieved March 15, 2022, from https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/
8) Moshiri, N. (2020, April 8). *Advanced Data Structures: TREAP insertion - youtube*. Youtube. Retrieved March 16, 2022, from https://www.youtube.com/watch?v=BQ9NBNnkJ5M
9) Open DSA Assignment 5 - 05.03.02 Binary Tree Space Requirements
10) Nair, K. K. (2021, April 30). *TREAP data structure*. Techie Delight. Retrieved March 16, 2022, from https://www.techiedelight.com/Treap-data-structure/