Random Treaps

Abdeselam El-Haman Abdeselam

December 7, 2017

Treaps are interesting data structures that act as trees and heaps at the same time with its 2 keys per node. A randomized implementation of these structures are very appealing in the focus of this course. In this document an analysis is done, showing that some properties about tree balance are true and it helps to prove that in expectance, a random treap is a well-balanced binary search tree.

1 Treaps vs Quicksort

The *memoryless* property of a tree shows us that, as a priority for a node is chosen independently for every node, we would know the structure of the tree if we knew those random values before inserting them.

This property makes us able to make a parallelism with the behaviour of quicksort with 2 important characteristics of random treaps, the priority and its binary search similarity. The priority of a node will position the node up on the tree if it's higher, or down if it's lower.

If we take a treap, we'll see that every element in the node has been compared to the root (x) once, and the root is the element with the highest priority. Then, the x node has a sub-tree as his left child and another one as his right child. We can see then that the elements of the left sub-tree have been compared with the root of that sub-tree, but never been compared with the trees in the other subtree. If we iterate like this, the root of every sub-tree in the treap, or the node at the ith level, can be seen as the *pivot* chosen in the ith iteration of the quicksort algorithm. The higher the random value of the priority is, the sooner it'd be appear in quicksort as a pivot. And the comparaisons are the same in both cases because of the binary search properties of the random treap.

We can also say that if we want to sort a sequence, we can insert every element in an empty tree, and then read the tree in a prefix order, this operation will take have a complexity of $O(n \log n)$, the same as quicksort.

2 Implementation

This project has been implemented in Python, with 2 main classes: `Treap` and `TreapNode`, that represent respectively the treap and a node of the treap.

The `TreapNode` doesn't have a lot of interesting things to talk about, as it has only getters and setters to its attributes (left child, right child, parent, key and priority). In the other hand, the `Treap` class includes every implementation of every algorithm useful to implement the random treap. Left and Right rotation routines have been implemented, as the find method too (just a BST search), but there's nothing special about them, nevertheless we'll check out more closely the `insert` and `delete` code.

The `insert` method has been implemented in 2 parts. A treap has to add the node as it was just a simple binary search tree, and then has to apply some rotations so that it satisfies also the heap property.

```
# BST step
attach = self.find(elem, False)

# node creation with random priority
node = TreapNode(elem, random.random())

# in case it's an empty tree
if attach == None:
    self.root = node
    return

node.setParent(attach)
if elem <= attach.getBkey():
    attach.setLeft(node)
else:
    attach.setRight(node)

# Heap step
while node.getParent() != None and node.getHkey() > node.getParent().getHkey():
    if node.isRightChild():
        self.leftRotation(node)
    else:
        self.rightRotation(node)
```

The `delete` method cannot be the same as the BSL, as it could break some priorities if not done well. So, for the chosen node, we will have to move it down the tree by rotations until it's a leaf, so we can delete it without problems. If we don't want to break the priorities, we'll do a rotation with the child who has the more priority (for example, if priority of left child = 0.5 and right child = 0.4, we'll chose the left child, so he'll be on top of the current right child and it won't break the priorities).

```
node = self.find(elem)

if node.getBkey() != elem:
    return

# moves node down til' it's a leaf
while node.getLeft() != None or node.getRight() != None:
    left = -1 if node.getLeft() == None else node.getLeft().getHkey()
    right = -1 if node.getRight() == None else node.getRight().getHkey()

    if left > right:
        self.rightRotation(node.getLeft())
    else:
        self.leftRotation(node.getRight())
```

```
15
16  # deletes node
17  if node.isRightChild():
18      node.getParent().setRight(None)
19  else:
20      node.getParent().setLeft(None)
```

3 Results

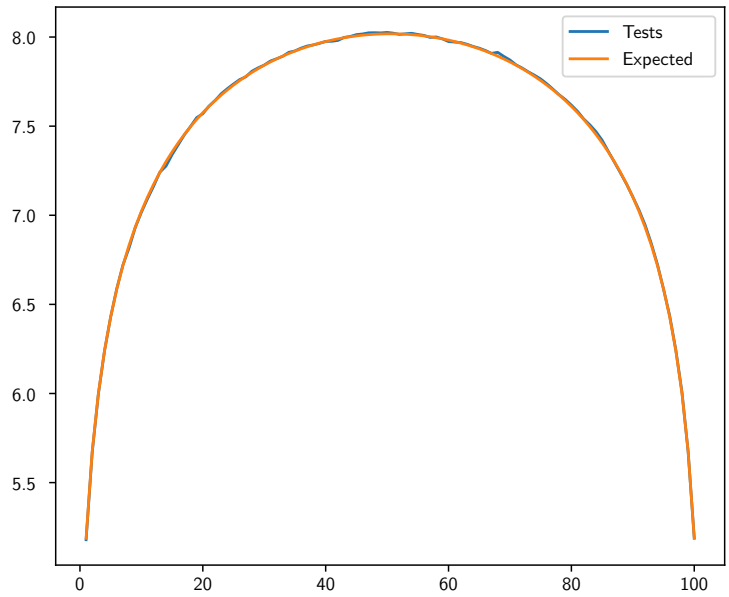
Lemma 8.6 states: Let T be a random treap for a set S of size n. For an element $x \in S$ having rank k,

$$E[\text{depth}(x)] = H_k + H_{n-k+1} - 1$$

Being H_n the n-th harmonic number.

For the tests I've taken every integer in the range $[1, 100]$ (inclusive), so we can know that the element x is has the rank x in the set. Then, I've created 100000 different treaps, each one inserting the elements of the set in a random order (after a shuffle for example).

In expectance, the results clearly prove that the property stated in the lemma is true, as it's seen in the graph with the results:



The average error is 0.05, which is pretty little and can shows us the acceptance of the lemma 8.6.

4 Conclusion

In conclusion, the treaps solve with an elegant twist some drawbacks of traditional tree-based data structures with the power of randomness, and in this document it has been proved that in expectance, random treaps will not have the problem of the worst-case scenario that have simple BSTs, with the operations having **the same complexity** ($O(\log n)$).