Treaps:

A Randomized Data Structure

Nicholas Carolan Connor Haugh Max Liu Ashira Mawji

I. Introduction

We strive for rapid data access in computational problem solving, especially with databases, parallel programing, and similar tasks. These applications frequently use binary search trees (BSTs) as a data structure. For operations on a binary search tree to be most efficient, the tree should be balanced such that its max height is logarithmically proportional to its number of nodes. With a balanced BST, we can expect a runtimes of O(lg n) for the *find()*, *insert()*, and *delete()* operations. However, a binary search tree does not guarantee an O(lg n) runtime, as a tree's shape and balance depends on its inputs. Consider when the inputs' keys inserted into the binary search are in strictly increasing or strictly decreasing order, causing the binary search tree to behave like a linked list - with a worse expected runtimes of O(n).Inputs can be unpredictable and sometimes an attacker might deliberately give a worst-case input to greatly increase the runtimes.

Variations of binary search trees that self-balance, independent of input, are therefore advantageous. In 1978, Guibas & Sedgewick proposed the Red-Black Tree as a way to improve binary search tree balancing. This data structure assigns a color, either red or black, to each BST node and maintains a balanced structure of the nodes based on rules regarding the sequence and number of red vs. black nodes in each root-to-leaf path. This stable arrangement guarantees a tree with *find(), delete(),* and *insert()* runtimes of O(lg n). However, this solution is complex because of the re-coloring associated with its structure, and yet still predictable because it is determined by its input.

In 1989, Aragon & Seidel proposed a novel alternative to Red-Black Trees called Treaps. Treaps combine the properties of binary search trees and heaps to produce expected runtimes of O(lg n) without the complexity of Red-Black Trees. Each Treap's position is uniquely determined by a combination of its key, according to BST properties, and a randomly assigned priority, according to heap rules. The randomization factor means that Treaps are not always balanced, and in a worst case of chaining would still have a runtime of O(n). However, for a large tree the probability of this is extremely low and we can expect the runtime of the average Treap operation to be O(lg n). Additionally, the randomization of Treaps prevents predictable arrangements, inhibiting attackers. Treaps have all of these benefits without the complexity of Red-Black Tree coloring and rotations, making them a useful data structure with clear practical advantages.

II. How Treaps Work

A Treap is a type of data structure that combines the properties of a binary search tree (BST) and a heap to form a unique arrangement of any given set of unique nodes. Each node contains two pieces of information: a distinct key as inputted by the user, and a distinct priority randomly generated from a uniform distribution. Treaps are organized with the keys obeying BST properties, such that every node in a given node's left subtree has a smaller key than it, and every node in its right subtree has a bigger key than it. Meanwhile the nodes also obey Heap properties based on their priorities, such that every descendant of a given node has a smaller priority than it.

Treaps have five standard operations: find, insert, delete, split, and join.

A. *Treap_Find (key, Treap)*

The *Treap_Find* method uses the Treap's BST properties to recursively locate the node with the given key, moving down the left subtree when looking for a smaller key and moving down the right subtree when looking for a bigger key.

//Use recursion to determine position of the given key, satisfying BST properties

if (key < tempTreapNode.key) then repeat with tempTreapNode.leftChild

else if (key > tempTreapNode.key) then repeat with tempTreapNode.rightChild

B. *Treap_Insert (Node (key, priority), Treap)*

As in the *Treap_Find* method, the *Treap_Insert* method first uses the BST ordering of the Treap's keys to recursively locate where the given Node can be inserted and maintain the Treap's BST properties. Then *Treap_Insert* performs left or right rotations on the Node so that its priority is smaller than that of its parent, meeting the Heap properties of the Treap.

//Use recursion to determine position of the given key, satisfying BST properties

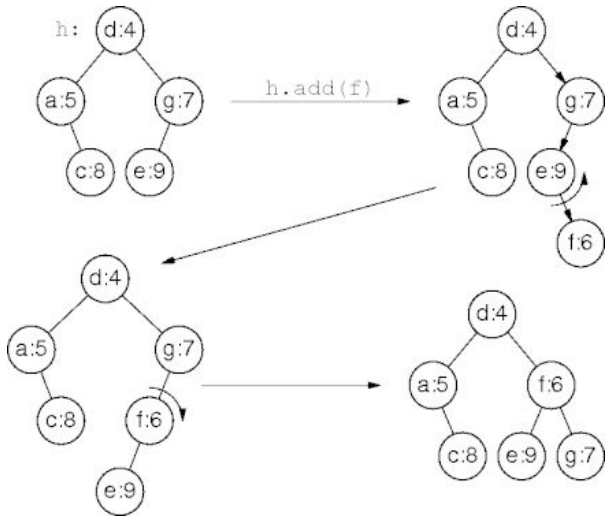
if (key < tempTreapNode.key) then repeat with tempTreapNode.leftChild

else if (key > tempTreapNode.key) then repeat with tempTreapNode.rightChild

//Use rotations to determine position of the given priority, satisfying Heap properties

if (priority < parentTreapNode.priority) then rotate subtree

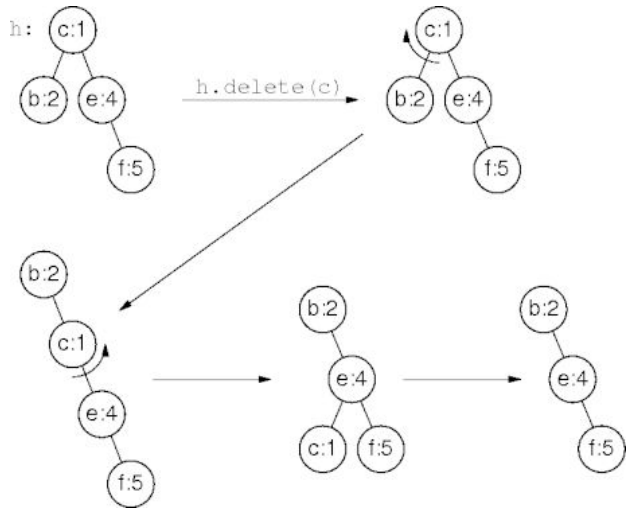
Figure 1: Treap_Insert



C. *Treap_Delete (key, Treap)*

As in the *Treap_Find* method, *Treap_Delete* traverses the Treap until it encounters the desired key to remove. Then through a series of rotations, it shifts the desired node to a leaf position and snips off the leaf.

Figure 2: Treap_Delete



D. *Treap_Split (Treap, key, leftSubTreap, rightSubTreap)*

In order to split a Treap into two subTreaps, one first inserts a node with the key along which the Treap is to be divided and an infinite priority. This ensures that this new node will be the root of Treap, and the Treap can be divided into its left and right subtrees as desired.

//Insert dummy node with infinite priority and key along which Treap is to be partitioned

Treap_Insert (Treap, Node(k, ∞), Treap)

//Divide Treap into the two subTreaps shooting off the root

leftSubTreap ← Treap.leftChild

rightSubTreap ← Treap.rightChild

E. *Treap_Join (leftSubtree, rightSubtree, Treap)*

Treap_Join is similar to *Treap_Split*, in that a dummy node of infinite priority is created. The two subTreaps are then appended to the dummy node as its left and right children. Then the dummy node is deleted, leaving the two subTreaps joined together in one happy Treap.

//Insert dummy Treap

N ← new Treap_Node (k, ∞)

//Append left and right subtrees as its children

N.leftChild ← leftSubtree

N.rightChild ← rightSubtree

//Delete dummy node

Treap_Delete (N.key, Treap)

III. Performance of Treaps

1. Proving the expected runtime of ln(n) for Treap *Insert()*, *Delete()*, and *Find()* operations.

To accomplish this task, we start with a Treap of n nodes, and calculate that the expected height of the tree is at least ln(n). We can consider the particular node *i* of the Treap where *i* has the uniformly drawn priority of p_i and therefore 1<*i*< n.

To determine the depth of a particular node, we simply want to find out the total number of ancestors a node has. Let's call the probability a node *j* is an ancestor of *i* 1 if *j* is a parent and 0 if not, and have that result be the random variable A_{ij}.

The expected value of A_{ij} will give us the number of ancestors a given node has.

In order to calculate A_{ij} we must first know that in considering all the nodes between *i* and *j, i* can only be an ancestor if it has the largest priority between *i* and *j,* not including j. We know this because this is a fundamental property of heaps, and the Treap is organized by heap properties. The probability this occurs is equivalent to the probability *i* takes on any priority between p_i and p_j as priorities are uniformly distributed.

Therefore expected value of A_{ij} must be equal to $\frac{1}{|j-i|+1}$

The expected number number of ancestors a node has, summed across all nodes will give the height of the Treap.

$$E[\text{height of a Treap}] = \sum_{1}^{n} E[A_{ij}]$$

We then insert $\frac{1}{|j-i|+1}$ for $E[A_{ij}]$, and fix *j* to be the root node. Which produces:

$$E[\text{height of a Treap with root node } j] = \sum_{1}^{n} \frac{1}{j-i+1}$$

Through our extensive knowledge of Harmonic Taylor series, we know that the expected height of a Treap will end up being less than 1 + 2ln(*n)*, which gives us the ln(n) runtime we desire.

2. Proving the Treap is almost always balanced.

We continue to use notations defined and used in part 1) above to compute the probability that the height of a Treap is more than logarithmic. In mathematical language, we want to compute the upper bound of $\Pr\{ D(x_l) >$ some logarithmic function $\}$ with x_l in X.

To choose a good inequality for the upper bound, we examine the information we have about $D(x_l)$:

i) $D(x_l) = \sum_{1 \le i \le n} A_{i,l}$ since the depth of a node is equal to the total number of its ancestors.

ii) For $1 \le i \le n$, $A_{i,l}$ are independent random variables. All the priorities are uniformly distributed continuous random variables, thus $A_{i,l}$ (the probability that x_i' priority p_i is greater than x_l's priority p_l) does not affect $A_{k,l}$ (the probability that x_k' priority p_k is greater than x_l's priority p_l).

iii) From part 1), we know that $Ex[D(x_l)] < 1 + 2\ln(n)$.

So, we use Chernoff inequality to derive the upper bound. In particular, we use the upper tail part of Chernoff bound. The upper tail of Chernoff bound says the following:

Let, $X_1, \ldots, X_n$ be independent random variables with $Pr[X_i = 1] = p_i$. Then if X is the sum of the X_i and if μ is E[X], for any $\delta > 0$:

$$P(X) > (1+\delta)\mu) \le e^{\mu\delta^2/2}$$

In Aragon and Seidel's original paper[1], they used Chernoff bound and derived a more precise upper bound as $Pr\{ D(x_l) \ge 1 + 2c \ln(n) \} < n\,(n/e)^{-c\ln(c/e)}$ with $c > 1$. This number is extremely small. For example[2], let n = 10000 and pick c = 5.429, then ln(n) = 9.21 and 2c ln(n) = 100. The upper bound $n\,(n/e)^{-c\ln(c/e)}$ is 4.081×10^{-10}. That is, for a Treap of size 10000, the probability that the height of the Treap consisting of 10,000 nodes is greater than 100 is less than one in two billion.

In conclusion, that the average depth of any node in a Treap is logarithmic and that the probability that the height of a Treap is more than logarithmic is extremely small prove Treap's effectiveness in balancing a binary search tree.

---

[1] Aragon, Cecilia R.; Seidel, Raimund (1989), "Randomized Search Trees" (PDF), *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C.: IEEE Computer Society Press, pp. 540–545, doi:10.1109/SFCS.1989.63531, ISBN 0-8186-1982-1.

[2] http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec6.treap/lec6.pdf, page 32 of 33.

IV. Citations

1. Aragon, Cecilia R.; Seidel, Raimund (1989), "Randomized Search Trees" (PDF), *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C.: IEEE Computer Society Press, pp. 540–545, doi:10.1109/SFCS.1989.63531, ISBN 0-8186-1982-1.
2. Blelloch, Guy; "Ordered Sets / Dictionaries Lecture" 2/12/19. Course Materials for *Parallel Programing*, Carnegie Mellon University.
3. Guibas, Leo J., and Robert Sedgewick. "A dichromatic framework for balanced trees." *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE, 1978.
4. "Lecture 6: Treaps." *University of California San Diego*, cseweb.ucsd.edu/~kube/cls/100/Lectures/ lec6.treap/lec6.pdf.