CSE341T/CSE549T **11/05/2014**

Treaps

Lecture 19

1 Binary Search Trees (BSTs)

Search trees are tree-based data structures that can be used to store and search for items that satisfy a total order. There are many types of search trees designed for a wide variety of purposes. Probably, the most common use is to implement sets and tables (dictionaries, mappings). What's common among search trees is that they store a collection of elements in a tree structure and use values (most often called keys) in the internal nodes to navigate in search of these elements. A *binary search tree* (BST) is a search tree in which every node in the tree has at most two children.

If search trees are kept "balanced" in some way, then they can usually be used to get good bounds on the work and span for accessing and updating them. We refer to such trees as *balanced search trees*. If trees are never updated but only used for searching, then balancing is easy—it needs only be done once—but what makes balanced trees interesting is their ability to efficiently maintain balance even when updated. To allow for efficient updates, balanced search trees do not require that the trees be strictly balanced, but rather that they are approximately balanced in some way. You should convince yourself that it would be impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

There are dozens of balanced search trees that have been suggested over the years, dating back at least to AVL trees in 1962. The trees mostly differ in how they maintain approximate balance. Most trees either try to maintain height balance (the two children of a node are about the same height) or weight balance (the two children of a node are about the same size, i.e., the number of elements). Here we list a few balanced trees:

1. *AVL trees.* Binary search trees in which the two children of each node differ in height by at most 1.

2. *Red-Black trees.* Binary search trees with a somewhat looser height balance criteria — no path is more than twice as long as any path.

3. *2–3 and 2–3–4 trees.* Trees with perfect height balance but the nodes can have different number of children so they might not be weight balanced. These are isomorphic to red-black trees by grouping each black node with its red children, if any.

4. *B-trees.* A generalization of 2–3–4 trees that allow for a large branching factor, sometimes up to 1000s of children. Due to their large branching factor, they are well-suited for storing data on disks.

5. *Splay trees.* Binary search trees that are only balanced in the amortized sense (i.e. on average across multiple operations).

6. *Weight balanced trees.* Trees in which the children all have the same size. These are most typically binary, but can also have other branching factors.

7. *Treaps.* A binary search tree that uses random priorities associated with every element to keep balance.

8. *Random search trees.* A variant on treaps in which priorities are not used, but random decisions are made with probabilities based on tree sizes.

9. *Skip trees.* A randomized search tree in which nodes are promoted to higher levels based on flipping coins. These are related to skip lists, which are not technically trees but are also used as a search structure.

Traditional treatments of binary search trees concentrate on three operations: search, insert and delete. Out of these, search is naturally parallel since any number of searches can proceed in parallel with no conflicts[1] However, insert and delete are inherently sequential, as normally described. For this reason, in later lectures, we'll discuss union and difference, which are more general operations that are useful for parallel updates and of which insert and delete are just a special case.

2 Treaps

A Treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. In the next couple lectures, we will show that with high probability, a treap with n keys will have depth $O(\log n)$. In a Treap, a random priority is assigned to every key. In practice, this is often done by performing a hash on a key, or by assigning a different random number to each key. We will assume that each priority is unique although it is possible to remove this assumption.

The nodes in a treap must satisfy two properties:

- **BST Property.** Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).

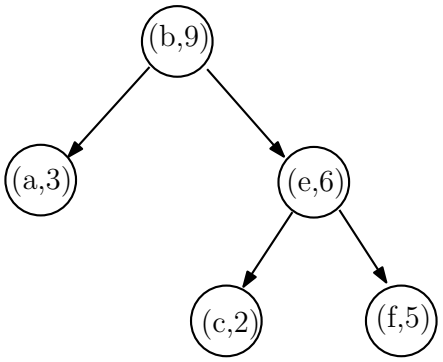
[1] In splay trees and other self-adjusting trees, this is not true since a searches can modify the tree.

- **Heap Property.** The associated priority satisfy the *heap* property. The (max) heap property requires for every node that the priority at a node is greater than the priority of its two children.

Consider the following key-value pairs:

(a,3), (b,9), (c, 2), (e,6), (f, 5)

These elements would be placed in the following treap.



Theorem 1 *For any set S of key-value pairs, there is exactly one treap T containing the key-value pairs in S which satisfies the treap properties.*

Proof. The key k with the highest priority in S must be the root node, since otherwise the tree would not be in heap order. Then, to satisfy the property that the treap is ordered with respect to the nodes' keys, all keys in S less than k must be in the left subtree, and all keys greater than k must be in the right subtree. Inductively, the two subtrees of k must be constructed in the same manner.

Now, if you are given a set of elements with keys and priorities, how would you build a treap out of them? Take the element with the largest priority, make that the root. Then partition around that root based on keys to find the elements in the right and left subtrees and recurse. What does this remind you of?

This procedure is exactly the procedure you do for quicksort. There is a straightforward relationship between the analysis of quicksort and the height of a treap.

3 A Different Analysis of Randomized Quicksort

We now turn to analyzing quicksort. As mentioned earlier, the motivation for analyzing this now is that this analysis is a nice segue into the Treap analysis. The argument here is essentially the same as the analysis we needed to show that the expected depth of a node in a Treap is $O(\log n)$.

For the analysis, we'll consider a completely equivalent algorithm which will be slightly easier to analyze. Before the start of the algorithm, we'll pick for each element a random priority uniformly at random from the real interval $[0, 1]$. At each step, instead of picking a pivot randomly, we'll instead pick the key with the highest priority (sound familiar?). Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of the algorithm are fully equivalent (modulo the technical details about how we might store the priority values).

We're interested in counting how many comparisons QUICKSORT makes. This immediately bounds the work for the algorithm because this is where the bulk of work is done. That is, if we let

$$X_n \quad = \quad \#\text{ of comparisons QUICKSORT makes on input of size } n,$$

our goal is to find an upper bound on $\mathrm{E}[X_n]$ for any input sequence S. For this, we'll consider the final sorted order[2] of the keys $T = \textsc{sort}(S)$. In this terminology, we'll also denote by p_i the priority we chose for the element T_i.

We'll derive an expression for X_n by breaking it up into a bunch of random variables and bound them. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence T. We use the random indicator variables A_{ij} to indicate whether we compare the elements T_i and T_j during the algorithm—i.e., the variable will take on the value 1 if they are compared and 0 otherwise.

Looking closely at the algorithm, we have that if some two elements are compared, one of them has to be a pivot in that call. So, then the other element will be either in the left partition or the right partition, but the pivot won't be the part of any partition. Therefore, once an element is a pivot, we never compare it to anything ever again. This gives the following observation:

Observation 1 *In the quicksort algorithm, if some two elements are compared in a* QUICKSORT *call, they will never be compared again in other call.*

Therefore, with these random variables, we can express the total comparison count X_n as follows:

$$X_n \quad \leq \quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} A_{i,j}$$

By linearity of expectation, we have $\mathrm{E}[X_n] \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[A_{i,j}]$. Furthermore, since each $A_{i,j}$ is an indicator random variable, $\mathrm{E}[A_{i,j}] = \Pr\{A_{i,j} = 1\}$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\Pr\{A_{i,j} = 1\}$) and working out the sum.

Computing the probability $\Pr\{A_{i,j} = 1\}$. The crux of the matter is in describing the event $A_{i,j} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result,

[2]Formally, there's a permutation $\pi \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of S and T.

let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the top level takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run QUICKSORT recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further. With this view, the following observation is not hard to see:

Claim 2 *For $i < j$, T_i and T_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.*

Proof. We'll show this by contradiction. Assume there is a key T_k, $i < k < j$ with a higher priority between them. In any collection of keys that include T_i and T_j, T_k will become a pivot before either of them. Since T_k "sits" between T_i and T_k (i.e., $T_i \leq T_k \leq T_j$) , it will separate T_i and T_j into different buckets, so they are never compared.

Therefore, for T_i and T_j to be compared, p_i or p_j has to be bigger than all the priorities in between. Since there are $j - i + 1$ possible keys in between (including both i and j) and each has equal probability of being the highest, the probability that either i or j is the greatest is $2/(j-i+1)$. Therefore,

$$\begin{aligned}
\mathrm{E}\left[A_{i,j}\right] &= \Pr\left\{A_{i,j} = 1\right\} \\
&= \Pr\left\{p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}\right\} \\
&= \frac{2}{j - i + 1}.
\end{aligned}$$

Hence, the expected number of comparisons made is

$$\begin{aligned}
\mathrm{E}\left[X_n\right] &\leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}\left[A_{i,j}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} (k = j - i) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\
&< \sum_{i=1}^{n-1} 2H_n \\
&= O(n \lg n)
\end{aligned}$$

4 Back to Treaps

How does this help with our analysis of treaps? Let us assume that you are given a set of elements with randomly generated priorities and you create a treap out of them. What is the expected search cost? The expected search cost is the expected depth of a node in the tree. Let us analyze that.

Consider a set of keys K and associated priorities $p : key \to int$. We assume the priorities are unique. Consider the keys laid out in order (i.e., by taking an inorder traversal of the Treap), and as with the analysis of quicksort we use T_i and T_j to refer to two positions in the sorted order.

If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors in the tree. So we want to know how many ancestor. We use the random indicator variable B_{ij} to indicate that T_j is an ancestor of T_i. Now the expected depth can be written as:

$$\mathrm{E}\left[\text{depth of } i \text{ in } T\right] = \sum_{j=1, j\neq i}^{n} \mathrm{E}\left[B_{ij}\right].$$

To analyze B_{ij} lets just consider the $|j-i|+1$ keys and associated priorities from T_i to T_j inclusive of both ends. (Note that keys outside of this range is irrelevant, because they don't effect the probability of B_{ij}.) We consider three cases:

1. The element T_i has the highest priority.
2. One of the elements T_k in the middle has the highest priority (i.e., neither T_i nor T_j.
3. The element T_j has the highest priority.

What happens in each case?

In the first case T_j cannot be an ancestor of T_i since T_i has a higher priority. In the second case, note that T_k will be picked to be the root of the subtree containing T_i and T_j before either T_i or T_j is picked, and separate T_i and T_j into two different subtrees based on the BST property. Thus, T_j cannot possibly be T_i's ancestor in this case. Finally, in the third case T_j is an ancestor of T_i, since there is no key between them with a higher priority to separate them into two subtrees.

We therefore have that T_j is an ancestor of T_i exactly when it has a priority greater than all elements from T_i to T_j (inclusive on both sides). Since priorities are selected randomly, this has a chance of $1/(j-i+1)$ and we have $\mathrm{E}\left[B_{ij}\right] = \frac{1}{|j-i|+1}$.

Now we have

$$\begin{aligned}
\mathrm{E}\left[\text{depth of } i \text{ in } T\right] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
&= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= \frac{1}{i} + \frac{1}{i-1} + ... + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n-i+1} \\
&= H_i + H_{n-i+1} \\
&= O(\log n)
\end{aligned}$$

Exercise 1 *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key.*

Is this similar to the bound you got for red-black trees? What is the red-black tree guarantee? It is that the height of the tree is $O(\lg n)$. The answer is no. The fact that the expected depth of a key (and hence its expected search cost) is $O(\lg n)$ does not imply that the expected height of the tree is $O(\lg n)$. To see why, think about a family of trees where every key has depth $O(\lg n)$ expect for one key which has depth of $\sqrt{n}$. For this family of trees, the expected depth of a node is $O(\lg n)$, but the expected height of a tree is $\sqrt{n}$. Or, putting it differently — the height of the tree depends on the depth of the key that has maximum depth. Even though we have the expected depth of a key, that doesn't tell us the expected value of the height of the tree. Recall that $\mathrm{E}\left[max\ depth\ across\ all\ keys\right]$ is not the same as $max\{\mathrm{E}\left[depthofkey1\right], \mathrm{E}\left[depthofkey2\right], ...\}$.

It turns out that in order to argue that the height of a tree is $O(\lg n)$, we need a high-probability bound, which we will study in the next lecture.