Lecture 21 — Treaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Margaret Reid-Miller — 6 November 2012

1 Quick Review: Binary Search Trees

For a quick recap, recall that in the last lecture, we were talking about binary search trees (BST). In particular, we looked at the following:

- *Many ways to keep a search tree almost balanced.* Such trees include red-black trees, 2-3 trees, B-trees, AVL trees, Splay trees, Treaps, weight balanced trees, skip trees, among others.

  Some of these are binary, some are not. In general, a node with k children will hold $k-1$ keys. But in this course, we will restrict ourselves to binary search trees.

- *Using* `split` *and* `join` *to implement other operations.* The `split` and `join` operations can be used to implement most other operations on binary search trees, including: `search`, `insert`, `delete`, `union`, `intersection` and `difference`.

- *An implementation of* `split` *and* `join` *on unbalanced trees.* We claim that the same idea can also be easily implemented on just about any of the balanced trees.

- *Cost of* `union` We showed that when both trees are split evenly at every level of the recursion tree, the work for `union` is $O(m \log(1 + \frac{n}{m})$, where $m \leq n$. If they don't split evenly, it only makes less work.

2 Today

Today we introduce a balanced binary search tree that is closely related to randomized quicksort. First we will look at the connection between quicksort and binary search trees, and then we will introduce treaps, which is a binary search tree that uses randomization to maintain balance.

3 Quicksort and BSTs

Can we think of binary search trees in terms of an algorithm we already know? As is turns out, the quicksort algorithm and binary search trees are closely related: if we write out the recursion tree for quicksort and annotate each node with the pivot it picks, what we get is a BST.

Let's try to convince ourselves that the function-call tree for quicksort generates a binary search tree when the keys are distinct. To do this, we'll modify the quicksort code from a earlier lecture

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

to produce the tree as we just described. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

```
1  fun qs_tree(S) =
2    if |S| = 0 then LEAF
3    else let
4           val p = pick a pivot from S
5           val S1 = ⟨ s ∈ S | s < p ⟩
6           val S2 = ⟨ s ∈ S | s > p ⟩
7           val (TL, TR) = ( qs_tree(S1) || qs_tree(S3) )
8         in
9           NODE(TL, p, TR)
10        end
```

Notice that this is clearly a binary tree. To show that this is a binary *search* tree, we only have to consider the ordering invariant. But this, too, is easy to see: for `qs_tree` call, we compute S_1, whose elements are strictly smaller than p—and S_2, whose elements are strictly bigger than p. So, the tree we construct has the ordering invariant. In fact, this is an algorithm that converts a sequence into a binary search tree.

It clear that, whatever the pivot strategy is, the maximum depth of the binary search tree resulting from `qs_tree` is the same as the maximum depth of the recursion tree for quicksort using that strategy. As shown in lecture, the expected depth of the recursion tree for *randomized* quicksort is $O(\log n)$

Can we maintain a tree data structure that centers on this random pivot-selection idea? If so, we automatically get a nice BST.

4 Treaps

Unlike quicksort, when building a BST we don't necessarily know all the elements that will be in the BST at the start, so we can't randomly pick an element (in the future) to be the root of the BST. So how can we use randomization to help maintain balance in a BST?

A treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. 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:

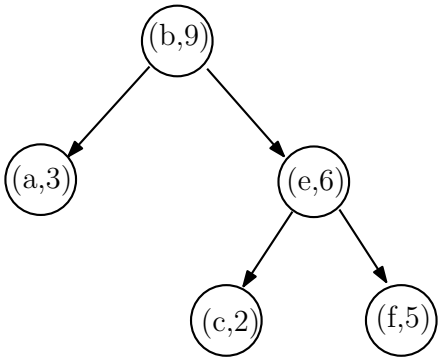
BST Property: Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).

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

Consider the following key-priority pairs:

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

These elements would be placed in the following treap.



Theorem 4.1. *For any set S of unique key-priority pairs, there is exactly one treap T containing the key-priority 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. Only one key has the highest priority. 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. □

Note, there is a subtle distinction here with respect to randomization. With quicksort the algorithm is randomized. With treaps, none of the functions for treaps are randomized. It is the data structure itself that is randomized[1].

Split and Join on Treaps

As mentioned in the last lecture, for any binary tree all we need to implement is split and join and these can be used to implement the other BST operations. Recall that split takes a BST and a key and splits the BST into two BST and an optional value. One BST only has keys that are less than the given key, the other BST only has keys that are greater than the given key, and the optional value is the value of the given key, if it is the tree. Join takes two BSTs and an optional middle (key,value) pair, where the maximum key on the first tree is less than the minimum key on the second tree. It returns a BST that contains all the keys the given BSTs and middle key.

We claim that the split code given in the last lecture for unbalanced trees does not need to be modified for treaps.

Exercise 1. *Convince yourselves than when doing a split none of the priority orders change (i.e. the code will never put a larger priority below a smaller priority).*

[1] In contrast, for the so called Randomize Binary Search Trees, it is the functions that update the tree that are randomized.

The join code, however, does need to be changed. The new version has to check the priorities of the two roots, and use whichever is greater as the new root. In the algorithm shown below, we assume that the priority of a key can be computed from the key (e.g., priorities are a hash of the key).

```
1  fun join(T1, m, T2) =
2  let
3    fun singleton(k, v) = Node(Leaf, Leaf, k, v)
4    fun join'(T1, T2) =
5      case (T1, T2) of
6        (Leaf, _) ⇒ T2
7      | (_, Leaf) ⇒ T1
8      | (Node(L1, R1, k1, v1), Node(L2, R2, k2, v2)) ⇒
9          if (priority(k1) > priority(k2)) then
10           Node(L1, join'(R1, T2), k1, v1)
11         else
12           Node(join'(T1, L2), R2, k2, v2)
13 in
14   case m of
15     NONE ⇒ join'(T1, T2))
16   | SOME(k, v) ⇒ join'(T1, join'(singleton(k, v), T2))
17 end
```

In the code join′ is a version of join that has no middle element as an argument. Note that line 9 compares the priorities of the two roots and then places the key with the larger priority in the new root causing a recursive call to join on one of the two sides.

We refer to the *left spine* of the tree as the path from the root to the leftmost node in the tree, and the *right spine* as the path from the root to the rightmost node in the tree. What join′(T_1, T_2) does is to interleave pieces of the left spine of T_1 with pieces the right spine of T_2, where the size of each piece depends on the priorities.

Because the keys and priorities determines a treap uniquely, repeated splits and joins on the same key, results in the same treap. This property is not always true of most other kinds of balanced trees; the order that operations are applied can change the shape of the tree.

Because the cost of split and join depends on the depth of the i^{th} element in a treap, we now analyze the expected depth of a key in the tree.

5 Expected Depth of a Key in a Treap

Consider a set of keys K and associated priorities p : key $\rightarrow int$. For this analysis, we assume the priorities are unique. Consider the keys laid out in order, and as with the analysis of quicksort, we use i and j to refer to the i^{th} and j^{th} keys in this ordering. Unlike quicksort analysis, though, when analyzing the depth of a node i, i j can be in any order, since an ancestor in a BST can be either less than or greater than node i.

If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors it has in the tree. So we want to know how many ancestors a particular node i has. We use the indicator random variable A_i^j to indicate that j is an ancestor of i. (Note that the superscript here does not mean A_i is raised to the power j; it simply is a reminder that j is the ancestor of i.) By the linearity of expectatons, the expected depth of i can be written as:

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \mathbf{E}\left[\sum_{j=1}^{n} A_i^j\right] = \sum_{j=1}^{n} \mathbf{E}\left[A_i^j\right].$$

To analyze A_i^j let's just consider the $|j-i|+1$ keys and associated priorities from i to j inclusive of both ends. As with the analysis of quicksort, if an element k has the highest priority and k is less than both i and j or greater than both i and j, it plays no role in whether j is an ancestor of i or not. The following three cases do:

1. The element i has the highest priority.
2. One of the elements k in the middle has the highest priority (i.e., neither i nor j).
3. The element j has the highest priority.

What happens in each case?

1. If i has the highest priority then j cannot be an ancestor of i, and $A_i^j = 0$.
2. If k between i and j has the highest priority, then $A_i^j = 0$, also. Suppose it was not. Then, as j is an ancestor of i, it must also be an ancestor of k. That is, since in a BST every branch covers a contiguous region, if i is in the left (or right) branch of j, then k must also be. But since the priority of k is larger than that of j this cannot be the case, so j is not an ancestor of i.
3. If j has the highest priority, j must be an ancestor of i and $A_i^j = 1$. Otherwise, to separate i from j would require a key in between with a higher priority. We therefore have that j is an ancestor of i exactly when it has a priority greater than all elements from i to j (inclusive on both sides).

Therefore j is an ancestor of i if and only if it has the highest priority of the keys between i and j, inclusive. Because priorities are selected randomly, there a chance of $1/(|j-i|+1)$ that $A_i^j = 1$ and we have $\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1}$. (Note that if we include the probability of either j being an ancestor of i or i being an ancestor of j then the analysis is identical to quicksort. Think about why.)

Now we have

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

Recall that the harmonic number is $H_n = \sum_{i=1}^{n} \frac{1}{n}$. It has the following bounds: $\ln n < H_n < \ln n + 1$, where $\ln n = \log_e n$. Notice that the expected depth of a key in the treap is determined solely by it relative position in the sorted keys.

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

Theorem 5.1. *For treaps the cost of* `join`(T_1, m, T_2) *returning* T *and of* `split`$(T, (k, v))$ *is* $O(\log |T|)$ *expected work and span.*

Proof. The `split` operation only traverses the path from the root down to the node at which the key lies or to a leaf if it is not in the tree. The work and span are proportional to this path length. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$.

For `join`(T_1, m, T_2) the code traverses only the right spine of T_1 or the left spine of T_2. Therefore the work is at most proportional to the sum of the depth of the rightmost key in T_1 and the depth of the leftmost key in T_2. The work of `join` is therefore the sum of the expected depth of these nodes. Since the resulting treap T is an interleaving of these spines, the expected depth is bound by $O(\log |T|)$. □

5.1 Expected overall depth of treaps

Even though the expected depth of a node in a treap is $O(\log n)$, it does not tell us what the expected maximum depth of a treap is. As you have saw in lecture 15, $\mathbf{E}\left[\max_i\{A_i\}\right] \neq \max_i\{\mathbf{E}\left[A_i\right]\}$. As you might surmise, the analysis for the expected depth is identical to the analysis of the expected span of randomized quicksort, except the recurrence uses 1 instead of $c \log n$. That is, the depth of the recursion tree for randomized quicksort is $D(n) = D(Y_n) + 1$, where Y_n is the size of the larger partition. Thus, the expected depth is $O(\log n)$.

It turns out that is possible to say something stronger: For a treap with n keys, the probability that any key is deeper than $10 \ln n$ is at most $1/n^2$[2]. That is, for large n a treap with random priorities

[2]The bound base on Chernoff bounds which relies on events being independent.

has depth $O(\log n)$ with *high probability*. It also implies that randomized quicksort $O(n \log n)$ work and $O(\log^2 n)$ span bounds hold with high probability.

Being able to put high probability bounds on the runtime of an algorithm can be critical in some situations. For example, suppose my company DontCrash is selling you a new air traffic control system and I say that in expectation, no two planes will get closer than 500 meters of each other—would you be satisfied? More relevant to this class, let's say you wanted to run 1000 jobs on 1000 processors and I told you that in expectation each finishes in an hour—would you be happy? How long might you have to wait?

There are two problems with expectations, at least on their own. Firstly, they tell us very little if anything about the variance. And secondly, as mentioned in an earlier lecture, the expectation of a maximum can be much higher than the maximum of expectations. The first has implications in real time systems where we need to get things done in time, and the second in getting efficient parallel algorithms (e.g., span is the max span of the two parallel calls). Proving these high probability bounds is beyond the scope of this course.

Summary

Earlier we showed that randomized quicksort has worst-case expected $O(n \log n)$ work, and this expectation was independent of the input. That is, there is no bad input that would cause the work to be worse than $O(n \log n)$ all the time. It is possible, however, (with extremely low probability) we could be unlucky, and the random chosen pivots could result in quicksort taking $O(n^2)$ work.

It turns out the same analysis shows that a deterministic quicksort will on average have $O(n \log n)$ work. Just shuffle the input randomly, and run the algorithm. It behaves the same way as randomized quicksort on that shuffled input. Unfortunately, on some inputs (e.g., almost sorted) the deterministic quicksort is slow, $O(n^2)$, every time on that input.

Treaps take advantage of the same randomization idea. But a binary search tree is a dynamic data structure, and it cannot change the order in which operations are applied to it. So instead of randomizing the input order, it adds randomization to the data structure itself.