Lecture 15 — Quicksort and Treaps

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

Lectured by Margaret Reid-Miller — 6 March 2012

Today:
- Quicksort and Binary Search Trees
- Work Analysis of Randomized Quicksort
- Treaps

1 Today

Today we will be introducing 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 consider the worst-case expected work of randomized quicksort. Finally, we will give the definition of a binary search tree that uses randomization to maintain balance.

2 Quick Review: Binary Search Trees

For a quick recap, recall that in the last lecture, we were talking about binary search trees. 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.

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 function-call tree

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

for quicksort and annotate each node with the pivot it picks, what we get is a BST. Conversely, if we pick the pivot according to a BST, the function-call tree for quicksort will be that BST.

Let's look at this in more detail. Consider the following implementation of quicksort. 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.

```
fun quicksort($S$) =
  if $|S| = 0$ then $S$
  else let
        val $p$ = pick a pivot from $S$
        val $S_1 = \langle s \in S \mid s < p \rangle$
        val $S_2 = \langle s \in S \mid s = p \rangle$
        val $S_3 = \langle s \in S \mid s > p \rangle$
      in
        quicksort($S_1$) @ $S_2$ @ quicksort($S_3$)
      end
```

Let's try to convince ourselves that the function-call tree for quicksort is a binary search tree. To do this, we'll modify the quicksort code to produce the tree as we just described:

```
fun qs_tree($S$) =
  if $|S| = 0$ then LEAF
  else let
        val $p$ = pick a pivot from $S$
        val $S_1 = \langle s \in S \mid s < p \rangle$
        val $S_2 = \langle s \in S \mid s = p \rangle$
        val $S_3 = \langle s \in S \mid s > p \rangle$
      in
        NODE(qs_tree($S_1$), $p$, qs_tree($S_3$))
      end
```

Notice that this is clearly a binary tree. To show that this is a binary *search* tree, we only have to worry about 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.

The next question to ask is: *How does the pivot choice effect the costs of quicksort and the quality of the BST*? A reasonable question to start is how deep this tree is. If the tree is about $O(\log n)$ deep, then it must be a reasonably balanced tree. In addition, the sum the depths of all the nodes in the tree is the same as the number of comparisons in quicksort. So then, what determines the depth of the nodes in the tree? As we discussed already, the depth of the BST is the same as the depth of the recursion tree for quicksort, so the choice of pivots determines the depth of the BST. Let's consider some strategies for picking a pivot:

- *Always pick the first element:* If the sequence is sorted in increasing order, then picking the first element is the same as picking the smallest element. We end up with a lopsided tree. In fact, each node only has a right child (and no left child). This is a linked-list, and the work to create this tree is the same as appending one element at a time to a linked list, $O(n^2)$. Similarly, if the sequence is sorted in decreasing order, each node in the tree has only a left child and no right child. In practice, it is not uncommon for a sort function input to be a sequence that is already sorted or nearly sorted.

- *Pick the median of three elements:* Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this strategy avoids the pitfall with sorted sequences, it is still possible to be unlucky, and in the worst-case the costs and tree depth are the same as the first strategy.
- *Pick an element randomly:* It is not immediately clear what the depth of this is, but intuitively, when we choose a random pivot, the size of each side is not far from $n/2$ in expectation. This doesn't give us a proof but it gives us hopes that this strategy will result in a tree of depth $O(\log n)$ in expectation. Indeed, picking a random pivot gives us expected $O(n \log n)$ work and $O(\log^2 n)$ span for quicksort and an expected $O(\log n)$-depth tree, as we will show.

Next we look at the expected work for randomized quicksort on all inputs, and from there conclude that the non-randomized quicksort has same work when averaged over all inputs.

4 Expected work for randomized quicksort

As discussed above, if we always pick the first element then the worst-case work is $O(n^2)$, for example when the array is already sorted. The *expected work*, though, is $O(n \log n)$ as we will prove below. That is, the work averaged over all possible input ordering is $O(n \log n)$. In other words, on most input this naive version of quicksort works well on average, but can be slow on some (common) inputs.

On the other hand, if we choose an element randomly to be the pivot, the *expected worst-case work* is $O(n \log n)$. That is, for input in **any** order, the expected work is $O(n \log n)$: No input has expected $O(n^2)$ work. But with a very small probability we can be unlucky, and the random pivots result in unbalanced partitions and the work is $O(n^2)$.

For the analysis of randomized quicksort, we'll consider a completely equivalent algorithm that 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 $[0, 1]$—and in Line 4 in the above algorithm, we'll instead pick the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of randomized quicksort 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

$$X_n \;=\; \text{\# of comparisions } \texttt{quicksort} \text{ makes on input of size } n,$$

our goal is to find an upper bound on $\mathbf{E}\left[X_n\right]$ for any input sequence S. For this, we'll consider the final sorted order[1] of the keys $T = \texttt{sort}(S)$. 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.

The crux of the matter is in describing the event $A_{ij} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result, let's take a closer look at the quicksort algorithm to gather some intutions. 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.

For any one call to `quicksort` there are three possibilities for A_{ij}, where $i < j$:

- The pivot (highest priority element) is either T_i or T_j, in which case T_i and T_j are compared and $A_{ij} = 1$.
- The pivot is element between T_i and T_j, in which case T_i is in S_1 and T_j is in S_3 and T_i and T_j will never be compared and $A_{ij} = 0$.
- The pivot is less than T_i or greater than T_j. Then T_i and T_j are either both in S_1 or both in S_3, respectively. Whether T_i and T_j are compared will be determined in some later recursive call to `quicksort`.

In the first case above, when two elements are compared, the non-pivot element is part of S_1, S_2, or S_3—but the pivot element is part of S_2, on which we don't recurse. This gives the following observation:

Observation 4.1. *If two elements are compared in a* `quicksort` *call, they will never be compared again in other call.*

Also notice in the corresponding BST, when two elements are compared, the pivot element become the root of two subtrees, one of which contains the other element.

Observation 4.2. *In the quicksort algorithm, two elements are compared in a* `quicksort` *call if and only if one element is an ancestor of the other in the corresponding BST.*

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

$$X_n \;\leq\; 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{ij}$$

The constant 3 is because our not-so-optimized quicksort compares each element to a pivot 3 times. By linearity of expectation, we have $\mathbf{E}\left[X_n\right] \leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right]$. Furthermore, since each A_{ij} is

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

an indicator random variable, $\mathbf{E}\left[A_{ij}\right] = \mathbf{Pr}\left[A_{ij} = 1\right]$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\mathbf{Pr}\left[A_{ij} = 1\right]$) and working out the sum.

Computing the probability $\mathbf{Pr}\left[A_{ij} = 1\right]$. Let us first consider the first two cases when the pivot is one of $T_i, T_{i+1}, ..., T_j$. With this view, the following observation is not hard to see:

Claim 4.3. *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 contradition. Asssume 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,

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

Notice that element T_i is compared to T_{i+1} with probability 1. It is easy to understand why if we consider the corresponding BST. One of T_i and T_{i+1} must be an ancestor of the other in the BST: There is no element that could be the root of a subtree that has T_i in its left subtree and T_{i+1} in its right subtree. On the other hand, if we consider T_i and T_{i+2} there could be such an element, namely T_{i+1}, which could have T_i in its left subtree and T_{i+2} in its right subtree. That is, with probability $1/3$, T_{i+1} has the highest probability of the three and T_i is not compared to T_{i+2}, and with probability $2/3$ one of T_i and T_{i+2} has the highest probability and, the two are compared. In general, the probability of two elements being compared is inversely proportional to the number of elements between them when sorted. The further apart the less likely they will be compared. Analogously, the further apart the less likely one will be the ancestor of the other in the related BST.

Hence, the expected number of comparisons made in randomized `quicksort` is

$$\begin{aligned}
\mathbf{E}\left[X_n\right] &\leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right] \\
&= 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&= 3\sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{2}{k} \\
&\leq 6\sum_{i=1}^{n}H_n \\
&= 6nH_n \in O(n\log n)
\end{aligned}$$

Because we can use the random priorities to shuffle the input so that the priorities are in decreasing order and then call the non-randomized quicksort, we have also shown the average work for the basic quicksort is also $O(n\log n)$. That is, the expected work for the basic quicksort is $O(n\log n)$ when averaged over all permutations of the input.

5 Treaps

So far, we know that whatever the pivot strategy is, the resulting function-call tree for quicksort has the same depth as the BST constructed using that strategy. As we will show picking a pivot at random gives us a BST that is only $O(\log n)$ deep in expectation. *Can we maintain a tree data structure that centers on this random pivot-selection idea?* If so, we automatically get a nice BST.

Unlike quicksort, when building a BST we don't necessarily know all the elements that will be in the BST at the start. But we can use the random priority idea in the analysis of quicksort to get the same BST that randomized quicksort would build.

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).

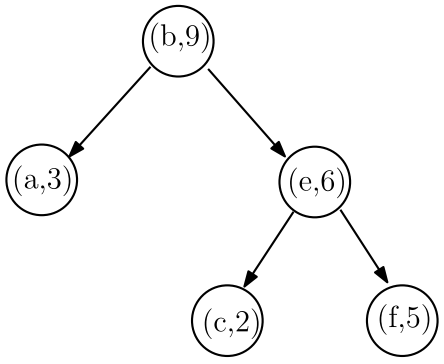
Heap Property: The associated priorities satisfy the *heap* property. The (max) heap property requires for every node that the value at a node is greater than the value of its two children.

In quick sort terminology, what the heap property says is that we are going to pick a pivot at random from the set of keys. This is because we have random priorities on the keys and the heap property says that the pivot (the root of that subtree) is the key with the highest priority.

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 5.1. *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. □

We claim that the split code given in the last lecture for unbalanced trees doesn't need to be modified at all 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).*