15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Ordered Sets / Dictionaries **Date:** February 12, 2009
Scribe: Brian Coltin

A dictionary contains a set of keys, with the ability to search for keys and recover data associated with them. Dictionaries can be implemented using trees, such as red-black trees, splay trees, and treaps. Searching on these trees is embarrassingly parallel since we are not modifying the binary tree, so we can perform as many searches at a time as we wish. The interesting problem is how to modify these trees in parallel.

10.1 Definition of Treaps

Treaps are especially easy to parallelize, so we will focus our discussion on them. It is possible to parallelize other types of trees, including AVL trees and red-black trees. However, it is much messier.

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:

- They are in order with respect to their keys, as in a typical binary search tree.
- They are in heap order with respect to their priorities, that is, no key has a key of lower priority as an ancestor.

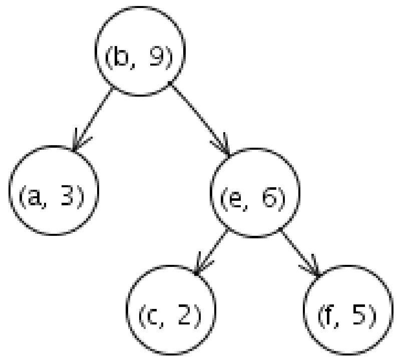
This is why this type of tree is called a treap, because the nodes are in search tree order and in heap order.

10.1.1 Example : Treap Construction

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.



10.2 Uniqueness of Treaps

Theorem 10.2.1 *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. ■

10.3 Basic Operators

We will define some building blocks used to describe higher-level operations on treaps.

10.3.1 `split`$(T, k) \rightarrow (T_l, T_r)$

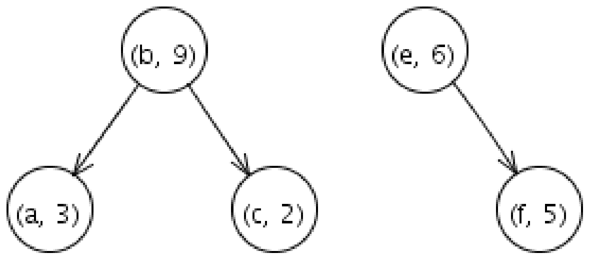
`split`(T, k) returns a pair of trees, T_l and T_r, splitting the original tree T at the key k.

- $T_l \subset T$ such that $\forall x \in T_l$, $x < k$
- $T_r \subset T$ such that $\forall x \in T_r$, $x \geq k$

If k is in T, it will be returned in T_r.

10.3.1.1 Example: `split`

We consider the treap T shown in the earlier example of treap construction. `split`(T, d) will split the tree around the key d, as shown below.



The tree with b as a root is returned as T_l, and the tree with e as a root is returned as T_r.

10.3.2 `join`$(T_l, T_r) \rightarrow T$

While `split` cuts a tree in two, `join` combines two trees into one. For `join`, each element in T_l must be less than each element in T_r.

These two primitives will be used to define our other operations. `split` and `join` both take $O(\log n)$ work and depth.

10.4 Other Treap Operations

The most common operations performed on treaps are search, insertion, and deletion. Note that search is easily parallelized, since the tree is not modified. This means that many searches can be run simultaneously without race conditions.

We will discuss three related operations, union, intersection, and building a tree. We will look at how to parallelize these operations.

- `union` takes two ordered sets and merges them into one. Duplicate elements are removed.
- `intersection` takes two treaps, and keeps only the elements which appear in both.
- `buildTree` takes as input an unordered set of values, such as an array, and builds a treap containing the values.

These operations also allow us to insert an unsorted set of values I into a tree T, by calling `union`$(T,$ `buildTree`$(I))$.

The algorithms we will describe for union, intersection, and build tree are optimal in terms of work under the comparison model. The depth, however, is not optimal. We will improve the depth of these operations later in the course when we discuss pipelining.

10.4.1 `buildTree`$(I) \to T$

`buildTree`(I) constructs a treap out of the values in I. We first generate a priority for each element in I, possibly by a hash. Then, the element a with the highest priority becomes the root. The remaining elements are partitioned into those with keys less than a and those with keys greater than a. `buildTree` is then called recursively to build these two subtrees.

```
fun BT(I) =
    if |I| = 0 then empty
    else a = maxprioritykey(I)
    e = {e in I | e < a}
    g = {e in I | e > a}
    return node(BT(e), a, BT(g))
```

This algorithm is very similar to quicksort. Picking the element with the maximum priority key is equivalent to picking out a random element. Then we identify the elements which are less than and greater than the selected element, and either sort or build the tree recursively on the two halves.

The complexity of this algorithm is identical to quicksort. We assume `maxprioritykey` takes linear work and logarithmic depth. So this algorithm, like quicksort, has $O(n \log n)$ expected work and $O(\log^2 n)$ depth with high probability.

Since we proved that the recursion depth of quicksort is $O(\log n)$ with high probability, this means that `buildTree` has $O(\log n)$ recursion depth with high probability, and hence a treap has depth $O(\log n)$ with high probability.

10.4.2 `union`$(T_1, T_2) \rightarrow T$

The `union` operation combines two treaps into one.

```
union(T1, T2) =
    if empty(T1) then T2
    else if empty(T2) then T1
    else if priority(root(T1)) < priority(root(T2)) then
        union(T2, T1) (* so T1 has the greater root *)
    else let
        (Tl1, r, Tr1) = T1
        (Tl2, Tr2) = split(T2, r)
        in node(union(Tl1, Tl2), r, union(Tr1, Tr2))
```

The root of the new tree will be the key with highest priority of the roots of the two original trees. Then, the other tree is split around this root. The two trees on each side of the root node are the joined together recursively.

`union` is purely functional, so there are no race conditions, and the two recursive calls to `union` can be made in parallel.

With this algorithm, we have a parallel merge algorithm, and also a parallel insertion algorithm if we union a tree with a single element.

10.4.2.1 Depth and Work of `union`

In each recursive call to `union`, we reduce the depth of one of the two trees by one. So the recursion depth is at most $D_{T_1} + D_{T_2} \leq O(\log |T_1| + \log |T_2|)$. Each level of recursion does at most $O(\log n)$ work, so we would estimate the depth to be $O(\log^2 n)$. In fact, the depth is $O(\log |T_1| \log |T_2|)$, but we will not prove this.

If we assume $|T_1| > |T_2|$, then the work is $O(|T_2| \log \frac{|T_1|+|T_2|}{|T_2|})$. This bound shows that union is an efficient implementation of both merge and sort. If we are doing an insertion, $|T_2| = 1$ and we do $O(\log n)$ work. In merge, where $|T_1| = |T_2| = n$, we do $O(n)$ work. This amount of work is optimal under the comparison model. The lower bound can be shown under the comparison model to be $\log \binom{n+m}{n}$, which matches our bound. This algorithm does no more work than if we inserted the items one by one sequentially.

10.5 Analysis of Treap Depth

Consider the random indicator variables A_{ij}, where A_{ij} is 1 if i is ancestor of j, and i and j are the positions of two keys in sorted order. Assume there is a key k between i and j which has a higher priority than both. Then i is to the left of k in the tree and j is to the right of k, so i is not an ancestor of j. If j is the highest priority of all the keys from i to j, then j is an ancestor of i. Similarly, if i has the highest priority, then i is an ancestor of j. So the only way that i is an ancestor of j is if i has the highest priority of all the keys from position i to j. Hence, the probability that i is an ancestor of j is the probability that i has the highest priority of the keys from i to j.

Each key has an equal probability of having the highest priority, so the probability that i is an ancestor of j is $\frac{1}{j-i+1}$. Therefore $E[A_{ij}] = \frac{1}{j-i+1}$.

Then we find the expected depth, D_i, of the subtree with the node at position i as the root.

$$\begin{aligned} E[D_i] &= \sum_{j=1}^{n} E[A_{ij}] \\ &= \sum_{j=1}^{n} \frac{1}{j-i+1} \\ &= O(\log n) \end{aligned}$$

We conclude that $E[D_i] = O(\log n)$ because $E[D_i]$ is the sum of two harmonic series heading in different directions from index i. Therefore, the expected depth of a treap is $\log n$.

We use Chernoff bounds to show that the depth is $\log n$ with high probability. Recall the Chernoff bound, that

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

where X is a sum of independent indicator variables.

Each of the random variables A_{ij} is independent. The probability the priority of j is greater than i does not affect the probability that the priority of k is greater than i, so these random variables are independent. This allows us to find a Chernoff bound to show that the depth is $\log n$ with high probability.