CMSC 420: Lecture 8
Treaps

Randomized Data Structures: A common design techlque in the field of algorithm design involves the notion of using randomization. A *randomized algorithm* employs a pseudo-random number generator to inform some of its decisions. Randomization has proved to be a remarkably useful technique, and randomized algorithms are often the fastest and simplest algorithms for a given application.

This may seem perplexing at first. Shouldn't an intelligent, clever algorithm designer be able to make better decisions than a simple random number generator? The issue is that a deterministic decision-making process may be susceptible to systematic biases, which in turn can result in unbalanced data structures. Randomness creates a layer of "independence," which can alleviate these systematic biases.

In this lecture, we will consider among the first randomized data structures for dictionary operations, called a *treap*. This data structure's name is a portmanteau (combination) of "tree" and "heap." It was developed by Raimund Seidel and Cecilia Aragon in 1989. (This 1-dimensional data structure is closely related to two 2-dimensional data structures, the *Cartesian tree* by Jean Vuillemin and the *priority search tree* of Edward McCreight, both discovered in 1980.)

Because the treap is a randomized data structure, its running time depend on the random choices made by the algorithm. We will see that all the standard dictionary operations take $O(\log n)$ *expected time*. The expectation is taken over all possible random choices that the algorithm may make. You might protest, since this allows for rare instances where the performance is very bad. While this is always a possibility, a more refined analysis shows that (assuming n is fairly large) the probability of poor performance is so insanely small that it is not worth worrying about.

Treaps: The intuition behind the treap is easy to understand. Recall back when we discussed standard (unbalanced) binary search trees that if keys are inserted in *random order*, the expected height of the tree is $O(\log n)$. The problem is that your user may not be so accommodating to insert keys in this order. A treap is a binary search tree whose structure arises "as if" the keys had been inserted in random order.

Let's recall how standard binary tree insertion works. When a new key is inserted into such a tree, it is inserted at the leaf level. If we were to label each node with a "timestamp" indicating its insertion time, as we follow any path from the root to a leaf, the timestamp values must increase monotonically (see Fig. 1(b)). From your earlier courses you should know a data structure that has this very property—such a tree is generally called *heap*.

This suggests the following simple idea: When first inserted, each key is assigned a *random priority*, call it `p.priority`. As in a standard binary tree, keys are sorted according to an inorder traversal. But, the priorities are maintained according to heap order. Since the priorities are random, it follows that the tree's structure is consistent with a tree resulting from a sequence of random insertions. Thus, we have the following:

Theorem: A treap storing n nodes has height $O(\log n)$ in expectation (over all $n!$ possible orderings of the random priorities present in the tree).

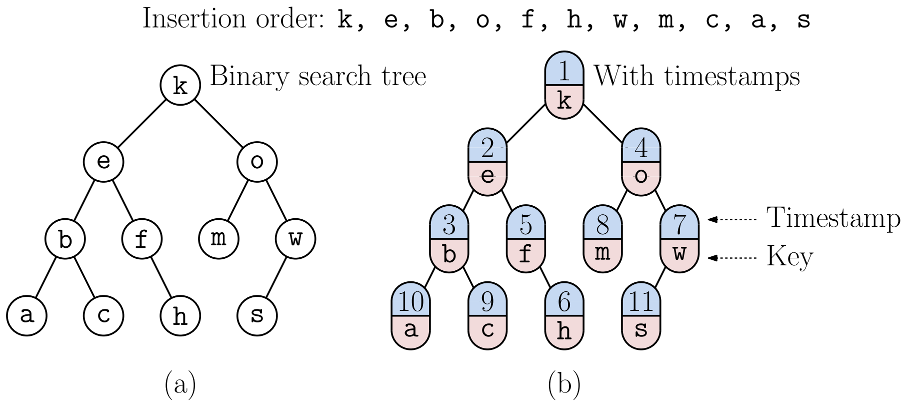

Fig. 1: (a) A binary search tree and (b) associating insertion timestamps with each node.

Since priorities are random, you might wonder about possibility of two priorities being equal. This might happen, but if the domain of random numbers if much larger than n (say at least n^2) then these events are so rare that they won't affect the tree's performance. We will show that it is possible to maintain this structure quite easily.

Geometric Interpretation: While Seidel and Aragon designed the treap as a 1-dimensional search structure, the introduction of numeric priorities suggests that we can interpret each key-priority pair as a point in 2-dimensional space. We can visualize a treap as a subdivision of 2-dimensional space as follows. Place all the points in rectangle, where the y-coordinates (ordered top to bottom) are the priorities and the x-coordinates are the keys, suitably mapped to numbers (see Fig. 2). Now, draw a horizontal line through the root. Because there are no points of lower priority, all the other points lie in the lower rectangle. Now, shoot a vertical ray down from this point. This splits the rectangle in two, with the points of the left subtree lying in the left rectangle and the points of the right subtree lying in the right subtree. Now, repeat the process recursively on each of the two halves.

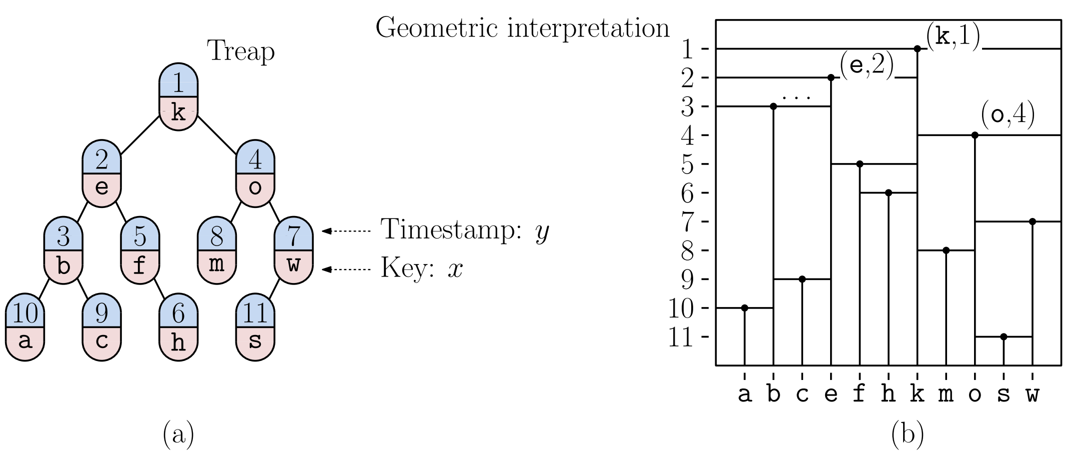

Fig. 2: (a) A treap and (b) geometric interpretation as a (randomized) priority search tree.

The resulting subdivision is closely related to two 2-dimensional data structures, one called the *Cartesian tree* and the other called the *priority search tree*. These structures can be used for answering various geometric range-searching queries.

Treap Insertion: Insertion into the treap is remarkably simple. First, we apply the standard binary-search-tree insertion procedure. When we "fall out" of the tree, we create a new node p, and set its priority, `p.priority`, to a random integer. We then retrace the path back up to the root (as we return from the recursive calls). Whenever we come to a node p whose child's priority is smaller than p's, we apply an appropriate single rotation (left or right, depending on which child it is), thus reversing their parent-child relationship. We continue doing this until the newly inserted key node is lifted up to its proper position in heap order. The code is very simple and is given below.

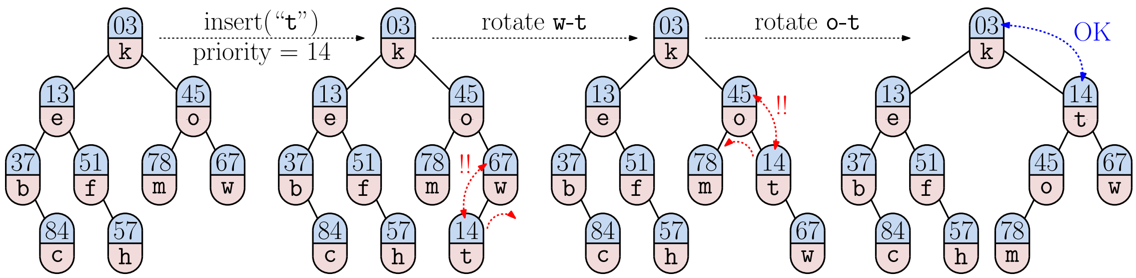

Fig. 3: Treap insertion.

Treap Deletion: Deletion is also quite easy, but as usual it is a bit more involved than insertion. If the deleted node is a leaf or has a single child, then we can remove it in the same manner that we did for binary trees, since the removal of the node preserves the heap order. However, if the node has two children, then normally we would have to find the replacement node, say its inorder successor and copy its contents to this node. The newly copied node will then be out of priority order, and rotations will be needed to restore it to its proper heap order. (The code presented at the end of the lecture notes uses this approach.)

There is, however, a cute trick for performing deletions. We first locate the node in the tree and then set its priority to ∞ (see Fig. 4). We then apply rotations to "sift it down" the tree to the leaf level, where we can easily unlink it from the tree.

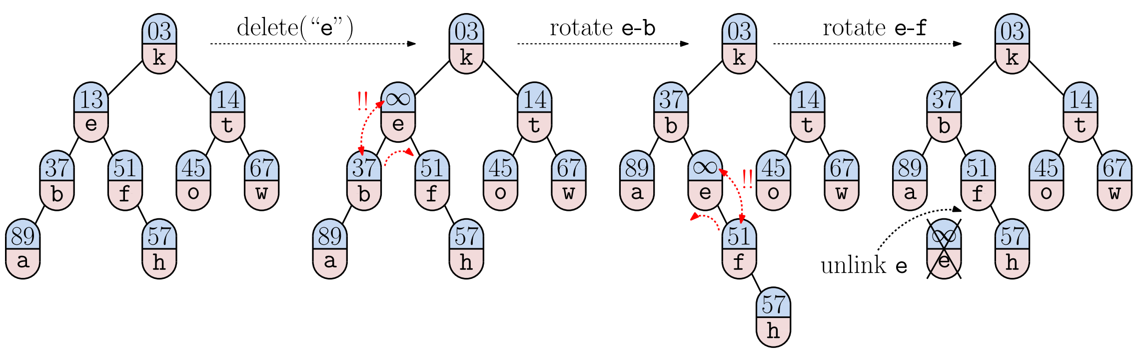

Fig. 4: Treap deletion.

The treap is particularly easy to implement because we never have to worry about adjusting the priority field. For this reason, treaps are among the fastest data tree-based dictionary structures.

Implementation: We can implement a treap by modifying our implementation of other binary search trees. First, the node structure is similar to that of a standard (unbalanced) binary search tree, but we include the priority value, called `priority`:

Treap Node Structure

```
private class TreapNode {
    Key key                     // key
    Value value                 // value
    int priority                // random priority (set when node is created)
    TreapNode left              // left child
    TreapNode right             // right child
}
```

The right and left rotation functions are the same as for AVL trees (and we omit them). We introduce three utility functions:

- `getPriority(p)`: If `p` is not `null` it returns the node's priority, and ∞ otherwise.
- `lowestPriority(p)`: Returns the node `p`, `p.left`, and `p.right` with the lowest priority.
- `restructure(p)`: Restructure the tree locally about `p` by rotating up a child having lower priority than `p`.

Treap Restructuring Operations

```
int getPriority(TreapNode p) { return (p == null ? MAX_PRIORITY : p.priority)  }

TreapNode lowestPriority(TreapNode p) {  // lowest priority of p, p.left, p.right
    TreapNode q = p
    if (getPriority(p.left) < getPriority(q))  q = p.left
    if (getPriority(p.right) < getPriority(q)) q = p.right
    return q
}

TreapNode restructure(TreapNode p) {       // restore priority at p
    if (p == null) return p                  // nothing to do
    TreapNode q = lowestPriority(p)          // get child to rotate
    if (q == p.left) p = rotateRight(p)      // rotate as needed
    else if (q == p.right) p = rotateLeft(p)
    return p                                 // return updated subtree
}
```

The insert function has exactly the same form the insert function for standard (unbalanced) binary search trees, but it invokes `restructure` as it returns up the tree. Notice that once we get to a node where a rotation is not needed, it will not be needed at any higher node.

The deletion function also follows the standard template. As in the case of deletion, the only difference from standard (unbalanced) search trees is the invocation of `restructure(p)` at the last line.

Treap Insertion

```
TreapNode insert(Key x, Value v, TreapNode p) {
    if (p == null) // fell out of the tree?
        p = new TreapNode(x, v, null, null)   // ... create a new leaf node here
    else if (x < p.key)                       // x is smaller?
        p.left = insert(x, v, p.left)         // ...insert left
    else if (x > p.key)                       // x is larger?
        p.right = insert(x, v, p.right)       // ...insert right
    else
        Error - Duplicate key!
    return restructure(p)                     // restructure (if needed)
}
```

Treap Deletion

```
TreapNode delete(Key x, TreapNode p) {
    if (p == null) Error - Nonexistant key!              // key not found
    else {
        if      (x < p.key) p.left  = delete(x, p.left)   // delete from left
        else if (x > p.key) p.right = delete(x, p.right)  // delete from right
                                                          // found it
        else if (p.left == null || p.right == null) {     // either child empty?
            if (p.left == null) return p.right            // return the other child
            else                return p.left
        } else {                                          // both children present
            TreapNode r = findReplacement(p.right)        // find replacement node
            copy contents from r to p
            p.right = delete(r.key, p.right)              // delete the replacement
        }
    }
    return restructure(p)                                 // restructure (if needed)
}
```