Lecture Notes on Randomized Binary Search Trees

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 16
October 19, 2010

1 Introduction

In the last lecture we have seen binary search trees. Time to insert and search for an element in the tree is $O(\log(n))$, where n is the number of elements in the tree, *when the tree is balanced*. In this lecture we discuss *randomized binary search trees* that achieve balance with a high probability. Alternatively, we might say that the probability that paths are much longer than the expected value of $\log(n)$ is very small.

Before we get to that, we return to the question on how to specify the ordering invariant for binary search trees so that we can use this effectively when reasoning about operations on trees.

2 Ordering Invariant Revisited

Recall the ordering invariant:

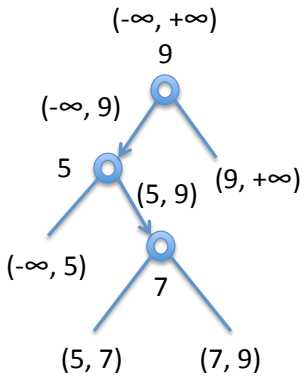
> **Ordering Invariant.** At any node with key k in a binary search tree, all keys of the elements in the left subtree are strictly less than k, while all keys of the elements in the right subtree are strictly greater than k.

In the last lecture we developed one way to check it: we collect the keys from the tree with an inorder traversal and then check if it is sorted in ascending order. Because this code is complex, it is difficult to reason about when we use it in pre- and post-conditions.

An alternative way of thinking about the invariant is as follows. Assume we are at a node with key k.

1. If we go to the *left* subtree, we establish an *upper bound* on the keys in the subtree: they must all be smaller than k.
2. If we go to the *right* subtree, we establish a *lower bound* on the keys in the subtree: they must all be larger than k.

The general idea then is to traverse the tree recursively, and pass down an interval with lower and upper bounds for all the keys in the tree. The following diagram illustrates this idea. We start at the root with an unrestricted interval, allowing any key, which is written as $(-\infty, +\infty)$. As usual in mathematics we write intervals as $(x, z) = \{y \mid x < y \text{ and } y < z\}$. At the leaves we write the interval for the subtree. For example, if there were a left subtree of the node with key 7, all of its keys would have to be in the interval $(5, 7)$.



The only difficulty in implementing this idea is the unbounded intervals, written above as $-\infty$ and $+\infty$. Here is one possibility: we pass a boolean flag to indicate if there is a lower bound or not, together with a lower bound. Similarly, we pass a boolean flag to indicate if there is an upper bound or not, together with an upper bound.

```
/* if lbd then lower is a lower bound */
/* if ubd then upper is an upper bound */
bool is_ordered(tree T, bool lbd, key lower, bool ubd, key upper)
{ key k;
  if (T == NULL) return true; /* an empty tree is an ordered tree */
  if (T->data == NULL) return false; /* data must be non-null */
```

```
  k = elem_key(T->data);
  return (!lbd || compare(lower,k) < 0)
    && (!ubd || compare(k,upper) < 0) /* root ok */
    && is_ordered(T->left, lbd, lower, true, k) /* left subtree ok? */
    && is_ordered(T->right, true, k, ubd, upper); /* right subtree ok? */
}
```

In the recursive calls we indicate through `true` when we know for sure we have a lower and upper bound. One small unpleasantness with this solution is that we still need to pass a key, even if there is no lower or upper bound at the beginning. Since the language does not support generating a default value of arbitrary type, we have to require the client to provide us with a function to get a default element.

```
key default_key();

bool is_ordtree(tree T) {
  return is_ordered(T, false, default_key(), false, default_key());
}
```

Another possibility is to pass the whole element, instead of just the key. This helps, because `NULL` is of type `elem` and therefore can be used to indicate the absence of a key, which means no lower or upper bound.

```
bool is_ordered(tree T, elem min, elem max) {
  if (T == NULL) return true; // empty tree is okay!
  if (T->data == NULL) return false; // no data -- bogus
  else {
    key k = elem_key(T->data);
    return
      // make sure the data is between min and max
         (min == NULL || compare(elem_key(min), k) < 0)
      && (max == NULL || compare(k, elem_key(max)) < 0)
      // check the subtrees
      && is_ordered(T->left, min, T->data)
      && is_ordered(T->right, T->data, max);
  }
}

bool is_ordtree(tree T) {
  return is_ordered(T, NULL, NULL);
```

```
}

bool is_bst(bst B) {
  return B != NULL && is_ordtree(B->root);
}
```

3 Balancing Trees

There are three basic techniques for keeping binary trees balanced. They reflect general computational thinking principles that are useful in a variety of circumstances in algorithm design.

Randomization If we inserted the elements into the tree in a random order, it would generally turn out to be balanced enough guarantee logarithmic insert and search times on the average. The question is how to guarantee the randomness. This is the subject of the rest of this lecture.

Dynamic Rebalancing After every insert operation we rebalance the tree enough so that insert and search are both logarithmic time. Obviously, this means that the rebalancing operation must also be bounded logarithmically. We will discuss means to achieve this in the next lecture.

Amortization Instead of rebalancing for every insert, we can also rebalance during search, as a kind of side effect. Then a sequence of inserts could be expensive, but we could amortize this cost during search. Splay trees are an example of this approach. We will not discuss them in this course.

4 Randomization

The easiest way to randomize would be to randomize the order of insertion. This works well if the data are presented to us, say, in an array. Often, however, data arrive in some predetermined sequential order. For example, when we read a dictionary or book from a file and want to insert the words into a binary search tree, then the order we read them in from a file is the natural order to insert them in. Storing them in an array may be difficult because we may not know ahead of time how many words there are.

A second possibility is to ensure that for every insertion of a new element into a tree, the probability that the element will end up at the root of the tree we are inserting to is $1/(n+1)$ if there are already n nodes in the tree. This requires some operations that can put a newly inserted element at the root of the tree without violating the ordering invariant. This method will be the subject of the rest of this lecture.

A third possibility is to assign an additional random key to each element. We use this second key to organize the tree as a heap. This is the idea behind *treaps*. It is clearly not straightforward because we have to maintain two sets of invariants: the ordering invariant for binary search trees on the given element keys and the heap ordering invariant on the additional random keys. We will not discuss treaps further in this course.

5 Root Insertion

In order to implement the second idea above we need a way to get a newly inserted element to the root. We do this with a recursive function that guarantees that a newly inserted element will be at the root of the subtree. We then apply a so-called *rotation* to lift up the new element by one level. Otherwise, the implementation is standard.

We begin by showing the pre- and post-conditions of the left and right rotation functions. In words: they preserve the elements of the tree and the ordering invariants. Further, a right rotation puts the left child at the root, while a left rotation puts the right child at the root.

```
tree rotate_right(tree T)
//@requires T != NULL && T->left != NULL;
//@requires is_ordtree(T);
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->data == \old(T->left->data);
  ;

tree rotate_left(tree T)
//@requires T != NULL && T->right != NULL;
//@requires is_ordtree(T);
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->data == \old(T->right->data);
  ;
```

```
tree root_insert(tree T, elem e)
//@requires is_ordtree(T);
//@requires e != NULL;
//@ensures is_ordtree(T);
//@ensures \result != NULL && \result->data == e;
//@ensures tree_search(\result, elem_key(e)) == e;
{
  if (T == NULL) {
    T = alloc(struct tree);
    T->data = e;
    T->left = NULL;
    T->right = NULL;
  } else {
    key kt = elem_key(T->data);
    key k = elem_key(e);
    if (compare(k, kt) == 0) {
      // don't rotate if updated in place
      T->data = e;
    } else if (compare(k, kt) < 0) {
      T->left = root_insert(T->left, e);
      T = rotate_right(T);
    } else {
      //@assert compare(k, kt) > 0;
      T->right = root_insert(T->right, e);
      T = rotate_left(T);
    }
  }
  return T;
}
```

The second `@ensures` clause shows the the new element e will be in the data field at the root of the tree. We can trace this invariant. Consider the last case:

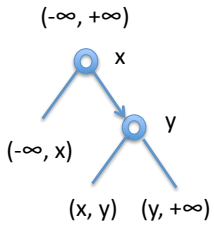
```
      //@assert compare(k, kt) > 0;
      T->right = root_insert(T->right, e);
      T = rotate_left(T);
```

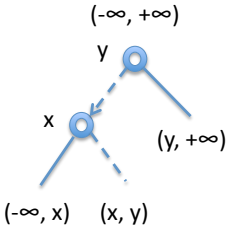
By the postcondition for `root_insert`, e will be the at the root of `T->right` after the assignment. Furthermore, it will not be null. This means we can

call `rotate_left`, which brings e to the root of its result, by its postcondition. After the assignment, e will therefore be at the root of T.

6 Left and Right Rotations

Below, we show the situation before a left rotation. We have generically denoted the crucial key values in question with x and y. Also, we have summarized whole subtrees with the intervals bounding their key values. Even though we wrote $-\infty$ and $+\infty$, when the whole tree is a subtree of a larger tree these bounds will be generic bounds α which is smaller than x and ω which is greater than y. The tree on the right is after the left rotation.





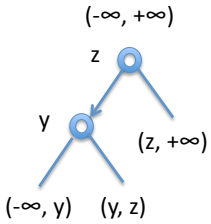
From the intervals we can see that the ordering invariants are preserved, as are the contents of the tree.

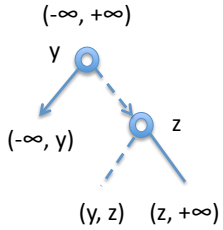
We implement this with some straightforward code. The main point to keep in mind is to use (or save) a component of the input before writing to it. We apply this idea systematically, writing to a location immediately after using it on the previous line.

```
tree rotate_left(tree T)
//@requires T != NULL && T->right != NULL;
//@requires is_ordtree(T);
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->data == \old(T->right->data);
{
  tree root = T->right;
  T->right = root->left;
  root->left = T;
  return root;
```

```
}
```

The right rotation is entirely symmetric. First in pictures:





Then in code:

```
tree rotate_right(tree T)
//@requires T != NULL && T->left != NULL;
//@requires is_ordtree(T);
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->data == \old(T->left->data);
{
  tree root = T->left;
  T->left = root->right;
  root->right = T;
  return root;
}
```

7 Randomization

Now that we have root insertion (which uses left and right rotation), we can implement the randomized insertion. To ensure that each element in every tree has equal probability to be at the root, we perform either a regular insertion or a root insertion, choosing root insertion with probability $1/(n+1)$ when there are n elements already in the tree. To get the right probability, we generate a random number and test whether it is 0 modulo $n+1$. To get n, we need to store the size of each subtree and update it during the insertion operations.

We leave the implementation of randomization as an exercise.

Exercises

Exercise 1 *Extend the implementation of binary search trees and rotations to track the size of each tree in a new field within its root node.*

Exercise 2 *Modify the implementation of* `root_insert` *using the sketch of randomized binary search trees in the last section. You may use the random number generator we have used in several lectures and test programs.*