

# Lecture Notes on Binary Search Trees

15-122: Principles of Imperative Computation  
Tom Cortina

Lecture 15  
October 14, 2010

## 1 Introduction

In the previous two lectures we have seen how to exploit the structure of binary trees in order to efficiently implement priority queues. A priority queue is an abstract type where we can insert an arbitrary element and delete the minimal element. The  $O(\log(n))$  worst-case time for insert and delete arose from using a balanced binary tree, which has maximal depth  $\log(n) + 1$  for storing  $n$  elements. With *binary search trees* we try to obtain efficient insert and search times for associative arrays, which we have previously implemented as hash tables. Even though the operations turn out to be quite different from those on priority queues, we will indeed eventually be able to achieve  $O(\log(n))$  worst-case asymptotic complexity for insert and search. This also extends to delete, although we won't discuss that operation in lecture.

## 2 The Interface

We assume that the client defines a type `elem` of elements and a type `key` of keys, as well as functions to extract keys from elements and to compare keys. Then the implementation of binary search trees will provide a type `bst` and functions to insert an element and to search for an element with a given key.

```
bst bst_new();  
void bst_insert(bst B, elem x);
```

```
elem bst_search(bst B, key k); /* return NULL if not in tree */
```

We stipulate that `elem` is some form of pointer type so we can return null if no element with the given key can be found. Generally, some more operations may be requested at the interface, such as the number of elements in the tree or a function to delete an element with a given key.

### 3 The Ordering Invariant

At the core of binary search trees is 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  $k$ .

We assume here that no key occurs more than once in a tree, and we have to make sure our insertion function maintains this invariant.

If our binary search tree were perfectly balanced, that is, had the same number of nodes on the left as on the right for every subtree, then the ordering invariant would ensure that search for an element with a given key has asymptotic complexity  $O(\log(n))$ , where  $n$  is the number of elements in the tree. Why? When searching for a given key  $k$  in the tree, we just compare  $k$  with the key  $k'$  of the entry at the root. If they are equal, we have found the entry. If  $k < k'$  we recursively search in the left subtree, and if  $k' < k$  we recursively search in the right subtree. This is just like binary search, except that instead of an array we traverse a tree data structure.

### 4 A Representation with Pointers

Unlike heaps, we cannot easily represent binary search trees with arrays and keep them balanced in the way we preserved the heap invariant. This is because with heaps there was a lot of flexibility where to insert new elements, while with binary search trees the position of the new element seems rather rigidly determined<sup>1</sup>. So we will use a pointer-based implementation where every node has two pointers: one to its left child and one to its right child. A missing child is represented as `NULL`, so a leaf just has two null pointers.

---

<sup>1</sup>although we will see next lecture that this is not strictly true

```
typedef struct tree* tree;
struct tree {
    elem data;
    tree left;
    tree right;
};
```

As usual, we have a *header* which in this case just consists of a pointer to the root of the tree. We often keep other information associated with the data structure in these headers, such as the size.

```
struct bst {
    tree root;
};
```

## 5 Searching for a Key

In this lecture, we will implement insertion and search first before considering the data structure invariant. This is not the usual way we proceed, but it turns out finding a good function to test the invariant is a significant challenge—meanwhile we would like to exercise programming with pointers in a tree a little. For now, we just assume we have two functions

```
bool is_ordtree(tree T);
bool is_bst(bst B);
```

Search is quite straightforward, implementing the informal description above. Recall that `compare(k1,k2)` returns  $-1$  if  $k_1 < k_2$ ,  $0$  if  $k_1 = k_2$ , and  $1$  if  $k_1 > k_2$ .

```
elem tree_search(tree T, key k)
//@requires is_ordtree(T);
//@ensures \result == NULL || compare(elem_key(\result), k) == 0;
{
    if (T == NULL) return NULL;
    { key kt = elem_key(T->data);
      if (compare(k, kt) == 0) return T->data;
      else if (compare(k, kt) < 0)
          return tree_search(T->left, k);
      else
          return tree_search(T->right, k);
    }
}
```

```

    }
}

elem bst_search(bst B, key k)
//@requires is_bst(B);
//@ensures \result == NULL || compare(elem_key(\result), k) == 0;
{
    return tree_search(B->root, k);
}

```

We chose here a recursive implementation, following the structure of a tree, but in practice an iterative version may be preferable (see Exercise ??).

We can check the invariant: if  $T$  is ordered when `tree_search(T)` is called (and presumably `is_bst` would guarantee that), then both subtrees should be ordered as well and the invariant is preserved.

## 6 Inserting an Element

Inserting an element is almost as simple. We just proceed as if we are looking for the key of the given element. If we find a node with that key, we just overwrite its data field. If not, we insert it in the place where it would have been, had it been there in the first place. This last clause, however, creates a small difficulty. When we hit a null pointer (which indicates the key was not already in the tree), we cannot just modify `NULL`. Instead, we *return* the new tree so that the parent can modify itself.

```

tree tree_insert(tree T, elem x)
//@requires is_ordtree(T);
//@ensures is_ordtree(T);
//@ensures tree_search(\result, elem_key(x)) != NULL;
{
    if (T == NULL) { /* create new node */
        T = alloc(struct tree); T->data = x;
        T->left = NULL; T->right = NULL;
    } else {
        key kt = elem_key(T->data);
        key k = elem_key(x);
        if (compare(k, kt) == 0) {
            T->data = x;
        } else if (compare(k, kt) < 0) {

```

```

        T->left = tree_insert(T->left, x);
    } else {
        //@assert compare(k, kt) > 0;
        T->right = tree_insert(T->right, x);
    }
}
return T;
}

```

For the same reason as in `tree_search`, we expect the subtrees to be ordered when we make recursive calls. The result should be ordered because for analogous reasons. The return subtree will also be useful at the root.

```

void bst_insert(bst B, elem x)
//@requires is_bst(B);
//@ensures is_bst(B);
{
    B->root = tree_insert(B->root, x);
    return;
}

```

## 7 Checking the Ordering Invariant

When we analyze the structure of the recursive functions implementing search and insert, we are tempted to say that a binary search is ordered if either it is null, or the left and right subtrees have a key that is smaller. This would yield the following code:

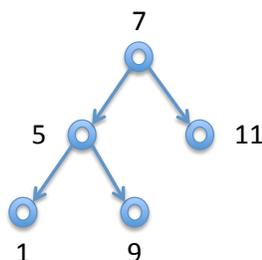
```

bool is_ordtree(tree T) {
    key k;
    if (T == NULL) return true; // an empty tree is a BST
    k = elem_key(T->data);
    return (T->left == NULL || (compare(elem_key(T->left->data), k) < 0
                                && is_ordtree(T->left)))
        && (T->right == NULL || (compare(k, elem_key(T->right->data)) < 0
                                && is_ordtree(T->right)));
}

```

While this should always be true for a binary search tree, it is far weaker than the ordering invariant stated at the beginning of lecture. Before reading on, you should check your understanding of that invariant to exhibit a tree that would satisfy the above, but violate the ordering invariant.

There is actually more than one problem with this. The most glaring one is that following tree would pass this test:



Even though, locally, the key of the left node is always smaller and on the right is always bigger, the node with key 9 is in the wrong place and we would not find it with our search algorithm since we would look in the right subtree of the root.

We will see another solution in the next lecture. For this lecture, we note that a so-called *inorder traversal* of the tree should return a sorted list. In-order traversal means that we first visit the left subtree, then the node itself, and then the right subtree. This is in contrast to a *preorder traversal* where we visit the node *before* its left and right subtrees, and a *postorder traversal* where we visit the node *after* its left and right subtrees.

Here is code for the inorder traversal. The particular operation we perform is to enqueue the key of the data elements in the tree.

```
void tree_traverse(tree T, queue keyqueue)
{
    if (T == NULL) return;
    tree_traverse(T->left, keyqueue); /* first left subtree */
    enq(keyqueue, elem_key(T->data)); /* then key of node */
    tree_traverse(T->right, keyqueue); /* then right subtree */
}
```

We can combine this with another function to check if the elements of the queue are in increasing order.

While this is correct and precise, the invariant checked is quite indirect, which makes it difficult to use this information in verifying the correctness of transformations or operations. In the next lecture, we will discuss a better way to describe and check the invariant.

Meanwhile, however, there is still a problem with the revised way of checking the invariant. Try to spot it before reading on.

Warning bells should always go off when we dereference a pointer. Could it be null? The reference to `T->left` and `T->right` are safe, because we have already ascertained that  $T$  is non-null. But what about `T->data`? Question: can the argument to `elem_key` be null? Checking the function

```
/* extracting keys from elements */
string elem_key(struct elem* e)
//@requires e != NULL;
{
    return e->word;
}
```

reveals that it is not permitted to be null. This makes sense: a data entry only has a key if it is not null. So we need to check it, which we do here is an assert annotation. It would be better to check it in a separate pass, or return a boolean, but since we will have a better solution in the next lecture, we will not bother with such refinements.

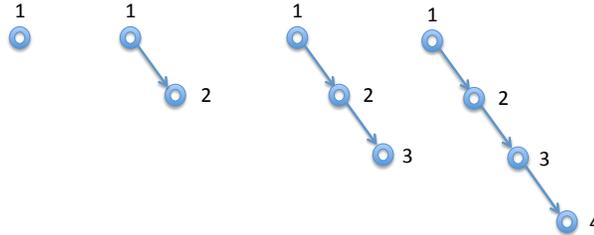
```
/* in-order traversal, adding all keys to keyqueue */
void tree_traverse(tree T, queue keyqueue)
{
    if (T == NULL) return;
    //@assert T->data != NULL;
    tree_traverse(T->left, keyqueue); /* first left subtree */
    enq(keyqueue, elem_key(T->data)); /* then key of node */
    tree_traverse(T->right, keyqueue); /* then right subtree */
}
```

## 8 The Shape of Binary Search Trees

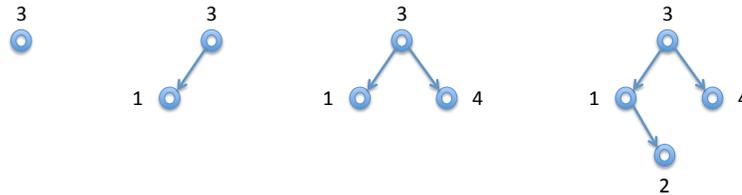
We have already mentioned that a balanced binary search trees has good properties, such as logarithmic time for insertion and search. The question is if binary search trees will be balanced. This depends on the order of insertion. Consider the insertion of numbers 1, 2, 3, and 4.

If we insert them in increasing order we obtain the following trees in

sequence.



Similarly, if we insert them in decreasing order we get a straight line along, always going to the left. If we instead insert in the order 3, 1, 4, 2, we obtain the following sequence of binary search trees:



Clearly, the last tree is much more balanced. In the extreme, if we insert elements with their keys in order, or reverse order, the tree will be linear, and search time will be  $O(n)$  for  $n$  items.

These observations mean that it is extremely important to pay attention to the balance of the tree. We will discuss two ways to generate balanced trees in the next two lectures: one based on randomness (similar to hash functions), the other one based on maintaining a balance invariant. There are also data structures that amortize the cost of rebalancing the tree over multiple operations such as splay trees, which we will not discuss in this course.

## Exercises

**Exercise 1** Rewrite `tree_search` to be iterative rather than recursive.

**Exercise 2** Rewrite `tree_insert` to be iterative rather than recursive. [**Hint:** The difficulty will be to update the pointers in the parents when we replace a node that is null. For that purpose we can keep a “trailing” pointer which should be the parent of the node currently under consideration.]