Lecture Notes on Binary Search Trees

15-122: Principles of Imperative Computation
Frank Pfenning André Platzer

Lecture 17
October 23, 2014

1 Introduction

In this lecture, we will continue considering associative arrays as we did in the hash table lecture. This time, we will follow a different implementation principle, however. With *binary search trees* we try to obtain efficient insert and search times for associative arrays dictionaries, which we have previously implemented as hash tables. We will 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. In particular, the worst-case complexity of associative array operations implemented as binary search trees is better than the worst-case complexity when implemented as hash tables.

2 Ordered Associative Arrays

Hashtables are associative arrays that organize the data in an array at an index that is determined from the key using a hash function. If the hash function is good, this means that the element will be placed at a reasonably random position spread out across the whole array. If it is bad, linear search is needed to locate the element.

There are many alternative ways of implementing associative arrays. For example, we could have stored the elements in an array, sorted by key. Then lookup by binary search would have been $O(\log(n))$, but insertion would be $O(n)$, because it takes $O(\log n)$ steps to find the right place, but

then $O(n)$ steps to make room for that new element by shifting all bigger elements over. We would also need to grow the array as in unbounded arrays to make sure it does not run out of capacity. In this lecture, we will follow similar principles, but move to a different data structure to make insertion a cheap operation as well, not just lookup. In particular, arrays themselves are not flexible enough for insertion, but the data structure that we will be devising in this lecture will be.

3 Abstract Binary Search

What are the operations that we needed to be able to perform binary search? We needed a way of comparing the key we were looking for with the key of a given element in our data structure. Depending on the result of that comparison, binary search returns the position of that element if they were the same, advances to the left if what we are looking for is smaller, or advances to the right if what we are looking for is bigger. For binary search to work with the complexity $O(\log n)$, it was important that binary search advances to the left or right *many steps at once*, not just by one element. Indeed, if we would follow the abstract binary search principle starting from the middle of the array but advancing only by one index in the array, we would obtain linear search, which has complexity $O(n)$, not $O(\log n)$.

Thus, binary search needs a way of comparing keys and a way of advancing through the elements of the data structure very quickly, either to the left (towards elements with smaller keys) or to the right (towards bigger ones). In arrays, advancing quickly is easy, because we just compute the new index to look at as either

```
int next_mid = (lower + mid) / 2;
```

or as

```
int next_mid = ((mid+1) + upper) / 2;
```

We use the first case if advancing from `mid` to the left (where `next_mid`$\leq$`mid`), because the element we are looking for is smaller than the element at `mid`, so we can discard all elements to the right of `mid` and have to look on the left of `mid`. The second case will be used if advancing from `mid` to the right (where `next_mid`$\geq$`mid`), because the element we are looking for is bigger than the one at `mid`, so we can discard all elements to the left of `mid`. In Lecture 6, we also saw that both computations might actually overflow in arithmetic, so we devised a more clever way of computing the midpoint,

but we will ignore this for simplicity here. In Lecture 6, we also did consider `int` as the data type. Now we study data of an arbitrary type `elem` provided by the client. In particular, as one step of abstraction, we will now actually compare elements in terms of their keys.

Unfortunately, inserting into arrays remains an $O(n)$ operation. For other data structures, insertion is easy. For example, insertion into a doubly linked list at a given list node is $O(1)$. But if we use a sorted doubly linked list, the insertion step will be easy, but finding the right position by binary search is challenging, because we can only advance one step to the left or right in a doubly linked list. That would throw us back into linear search through the list to find the element, which gives a lookup complexity of $O(n)$. How can we combine the advantages of both: fast navigation by several elements as in arrays, together with fast insertion as in doubly linked lists? Before you read on, try to see if you can find an answer yourself.

In order to obtain the advantages of both, and, thus, enable binary search on a data structure that supports fast insertion, we proceed as follows. The crucial observation is that arrays provide fast access to any arbitrary index in the array, which is why they are called a random access data structure, but binary search only needs very selective access from each element. Whatever element binary search is looking at, it only needs access to that element and one (sufficiently far away) left element and one (sufficiently far away) right element. If binary search has just looked at index `mid`, then it will subsequently only look at either `(lower + mid) / 2` or `(mid+1 + upper) / 2`. In particular, for each element, we need to remember what its key is, what its left successor is and what its right successor is, but nothing else. We use this insight to generalize the principle behind binary search to a more general data structure.

4 Binary Search in Binary Search Trees

The data structure we have developed so far results in a (binary) tree. A binary tree consists of a set of nodes and, for each node, its left and its right child. Finding an element in a binary search tree follows exactly the same idea that binary search did, just on a more abstract data structure:

1. Compare the current node to what we are looking for. Stop if equal.
2. If what we are looking for is smaller, proceed to the left successor.
3. If what we are looking for is bigger, proceed to the right successor.

What do we need to know about the binary tree to make sure that this principle will always lookup elements correctly? What data structure invariant do we need to maintain for the binary search tree? Do you have a suggestion?

5 The Ordering Invariant

Binary search was only correct for arrays if the array was sorted. Only then do we know that it is okay not to look at the upper half of the array if the element we are looking for is smaller than the middle element, because, in a sorted array, it can then only occur in the lower half, if at all. For binary search to work correctly on binary search trees, we, thus, need to maintain a corresponding data structure invariant. All elements to the right of a node have keys that are bigger than the key of that node. And all the nodes to the left of that node have smaller keys than the key at that node.

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.

This implies 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. Unlike in an array, however, we will see that insertion is quick as well.

6 The Interface

The basic interface for binary search trees is almost the same as for hash tables, because both implement the same abstract principle: associative arrays. Binary search trees, of course, do not need a hash function. 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.

```
/* Client-side interface */

typedef ______* elem;
typedef ______ key;

key elem_key(elem e)
//@requires e != NULL;
  ;

int key_compare(key k1, key k2)
//@ensures -1 <= \result && \result <= 1;
  ;

/* Library interface */

typedef ________ bst;

bst bst_new();
void bst_insert(bst B, elem e)
//@requires e != NULL;
  ;
elem bst_lookup(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.

The `key_compare` function provided by the client is different from the equality function we used for hash tables. For binary search trees, we actually need to compare keys k_1 and k_2 and determine if $k_1 < k_2$, $k_1 = k_2$, or $k_1 > k_2$. A standard approach to this in imperative languages is for a comparison function to return an integer r, where $r < 0$ means $k_1 < k_2$, $r = 0$ means $k_1 = k_2$, and $r > 0$ means $k_1 > k_2$. Our contract captures that we expect `key_compare` to return no values other than -1, 0, and 1.

7 A Representation with Pointers

We will use a pointer-based implementation for trees 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.

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

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_header {
  tree* root;
};
```

8 Searching for a Key

In this lecture, we will implement insertion and lookup 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 `key_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_lookup(tree* T, key k)
//@requires is_ordtree(T);
//@ensures \result == NULL || key_compare(elem_key(\result), k) == 0;
{
  if (T == NULL) return NULL;
  int r = key_compare(k, elem_key(T->data));
  if (r == 0)
    return T->data;
  else if (r < 0)
    return tree_lookup(T->left, k);
  else //@assert r > 0;
    return tree_lookup(T->right, k);
```

```
}

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

We chose here a recursive implementation, following the structure of a tree, but in practice an iterative version may also be a reasonable alternative (see Exercise 1).

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

9 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 e)
//@requires is_ordtree(T);
//@requires e != NULL;
//@ensures is_ordtree(\result);
{
  if (T == NULL) {
    /* create new node and return it */
    T = alloc(struct tree_node);
    T->data = e;
    T->left = NULL; T->right = NULL;
    return T;
  }
  int r = key_compare(elem_key(e), elem_key(T->data));
  if (r == 0)
```

```
    T->data = e; /* modify in place */
  else if (r < 0)
    T->left = tree_insert(T->left, e);
  else //@assert r > 0;
    T->right = tree_insert(T->right, e);
  return T;
}
```

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

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

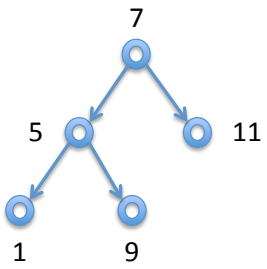
10 Checking the Ordering Invariant

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:

```
/* THIS CODE IS BUGGY */
bool is_ordtree(tree* T) {
  if (T == NULL) return true;  /* an empty tree is a BST */
  key k = elem_key(T->data);
  return (T->left == NULL
          || (key_compare(elem_key(T->left->data), k) < 0
              && is_ordtree(T->left)))
      && (T->right == NULL
          || (key_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.

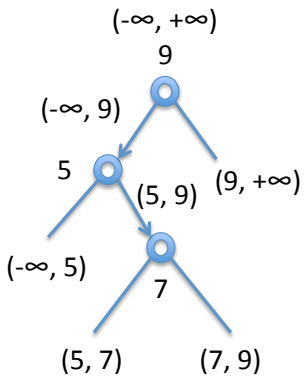
An alternative way of thinking about the invariant is as follows. Assume we are at a node with key k.

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 not just the key, but the particular element which bounds the tree from which we can extract the element. This allows us to pass `NULL` in case there is no lower or upper bound.

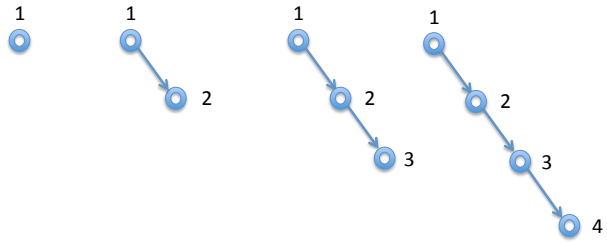
```
bool is_ordered(tree* T, elem lower, elem upper) {
  if (T == NULL) return true;
  return T->data != NULL
    && (lower == NULL || elem_compare(lower, T->data) < 0)
    && (upper == NULL || elem_compare(T->data, upper) < 0)
    && is_ordered(T->left, lower, T->data)
    && is_ordered(T->right, T->data, upper);
}

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

A word of caution: the `is_ordered(T, NULL, NULL)` pre- and post-condition of the function `tree_insert` is actually not strong enough to prove the correctness of the recursive function. A similar remark applies to `tree_lookup`. This is because of the missing information of the bounds. We will return to this issue in a later lecture.

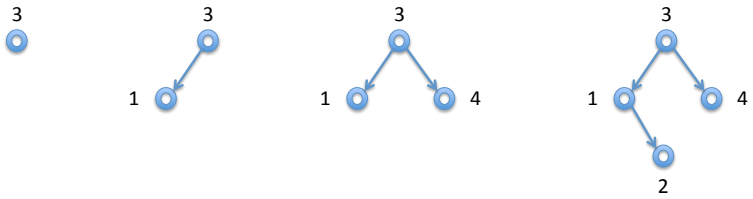
11 The Shape of Binary Search Trees

We have already mentioned that balanced binary search trees have 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 ways to keep binary search trees balanced in a later lecture.

Exercises

Exercise 1 *Rewrite* `tree_lookup` *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 note currently under consideration.]*

Exercise 3 *The binary search tree interface only expected a single function for key comparison to be provided by the client:*

```
int elem_compare(elem k1, elem k2);
```

An alternative design would have been to, instead, expect the client to provide a set of elem comparison functions, one for each outcome:

```
bool elem_equal(elem k1, elem k2);
bool elem_greater(elem k1, elem k2);
bool elem_less(elem k1, elem k2);
```

What are the advantages and disadvantages of such a design?