

Lecture 15 Notes

Binary Search Trees

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning, André Platzer, Rob Simmons

1 Introduction

In this lecture, we will continue considering ways to implement the set (or associative array) interface. This time, we will implement this interface with *binary search trees*. We will eventually be able to achieve $O(\log n)$ worst-case asymptotic complexity for insert and lookup. This also extends to delete, although we won't discuss that operation in lecture.

This fits as follows with respect to our learning goals:

Computational Thinking: We discover binary trees as a way to organize information. We superimpose to them the notion of sortedness, which we examined in the past, as a way to obtain exponential speedups.

Algorithms and Data Structures: We present binary search trees as a space-efficient and extensible data structure with a potentially logarithmic complexity for many operations of interest — we will see in the next lecture how to guarantee this bound.

Programming: We define a type for binary trees and use recursion as a convenient approach to implement specification functions and operations on them.

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.) Arrays are not flexible enough for fast 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 the array-based binary search we've studied, each iteration calculates a midpoint

```
int mid = lower + (upper - lower) / 2;
```

and a new bound for the next iteration is (if the key we're searching for is smaller than the element at mid)

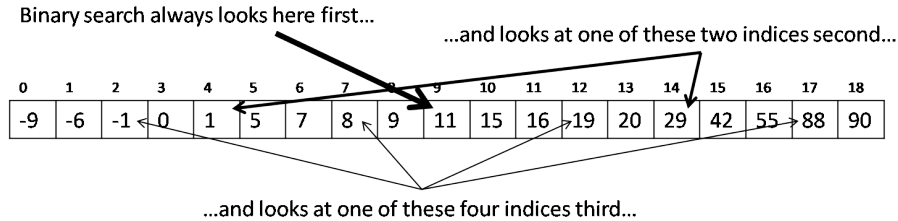
```
upper = mid;
```

or (if the key is larger)

```
lower = mid + 1;
```

So we know that the next value mid will be either $(lower + mid) / 2$ or $((mid + 1) + upper) / 2$ (ignoring the possibility of overflow).

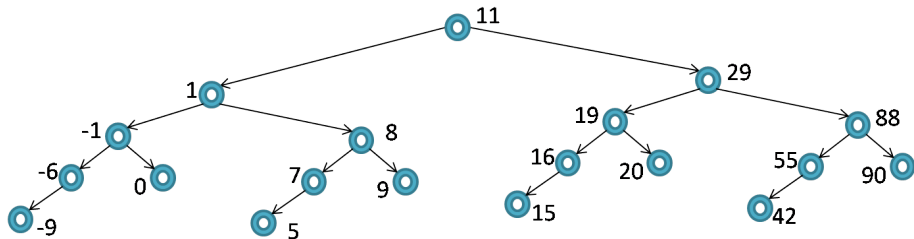
This pattern continues, and given any sorted array, we can enumerate all possible binary searches:



This pattern means that constant-time access to an array element at an arbitrary index isn't necessary for doing binary search! To do binary search on the array above, all we need is constant time access from array index 9 (containing 11) to array indices 4 and 14 (containing 1 and 29, respectively), constant time access from array index 4 to array indices 2 and 7, and so on. At each point in binary search, we know that our search will proceed in one of at most two ways, so we will explicitly represent those choices with a pointer structure, giving us the structure of a *binary tree*. The tree structure that we got from running binary search on this array...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-9	-6	-1	0	1	5	7	8	9	11	15	16	19	20	29	42	55	88	90

... corresponds to this binary tree:



4 Representing Binary Trees with Pointers

To represent a binary tree using pointers, we use a struct with two pointers: one to the left child and one to the right child. If there is no child, the pointer is NULL. A leaf of the tree is a node with two NULL pointers.

```

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

```

Rather than the fully generic data implementation that we used for hash tables, we'll assume for the sake of simplicity that the client is providing us with a type of `elem` that is known to be a pointer, and a single function `elem_compare`.

```

/* Client-side interface */
// typedef _____* elem;

int elem_compare(elem k1, elem k2)
    /*@requires k1 != NULL && k2 != NULL; @*/
    /*@ensures -1 <= \result && \result <= 1; @*/ ;

```

We require that valid values of type `elem` be non-NULL. — in fact an implementation of associative arrays based on trees would use NULL to signal that an `elem` is not there.

The `elem_compare` function provided by the client is different from the equivalence function we used for hash tables. For binary search trees, we 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 common approach to this 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 `elem_compare` to return no values other than -1, 0, and 1.

Trees are the second *recursive* data structure we've seen: a tree node has two fields that contain pointers to tree nodes. Thus far we've only seen recursive data structures as linked lists, either chains in a hash table or list segments in a stack or a queue.

Let's remember how we picture list segments. Any list segment is referred to by two pointers: `start` and `end`, and there are two possibilities for how this list can be constructed, both of which require `start` to be non-NULL (and `start->data` also to satisfy our constraints on `elem` values).

```


1 bool is_segment(list* start, list* end) {
2   if (start == NULL) return false;
3   if (start->data == NULL) return false;
4   if (start == end) return true;

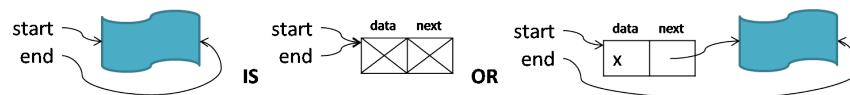
```

```

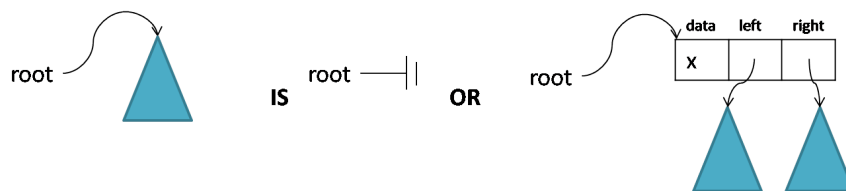
5  return is_segment(start->next, end);
6  }

```

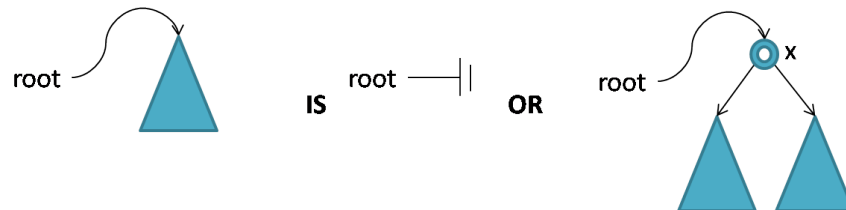
We can represent these choices graphically by using a picture like  to represent an arbitrary segment. Then we know every segment has one or two forms:



We'll create a similar picture for trees: the tree containing no elements is NULL, and a non-empty tree is a struct with three fields: the data and the left and right pointers, which are themselves trees.



Rather than drawing out the `tree_node` struct with its three fields explicitly, we'll usually use a more graph-like way of presenting trees:



This recursive definition can be directly encoded into a very simple data structure invariant `is_tree`. It checks very little: just that all the data fields are non-NULL, as the client interface requires. If it terminates, it also ensures that there are no cycles; a cycle would cause non-termination, just as it would with `is_segment`.

```

1  bool is_tree(tree* root) {
2    if (root == NULL) return true;
3    return root->data != NULL
4      && is_tree(root->left) && is_tree(root->right);
5  }

```

4.1 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. This *ordering invariant* is a core idea of binary search trees; it's what makes a binary tree into a binary *search* tree.

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

We won't write code for checking the ordering invariant just yet, as that turns out to be surprisingly difficult. We'll first discuss the lookup and insertion functions for binary search trees.

5 Searching for a Key

The ordering invariant lets us find an element e in a binary search tree the same way we found an element with binary search, just on the more abstract tree data structure. Here is a recursive algorithm for search, starting at the root of the tree:

1. If the tree is empty, stop.
2. Compare the key k of the current node to e . Stop if equal.
3. If e is smaller than k , proceed to the left child.
4. If e is larger than k , proceed to the right child.

The implementation of this search captures the informal description above. Recall that `elem_compare(x, y)` returns -1 if $x < y$, 0 if $x = y$, and 1 if $x > y$.

```
1 bool tree_lookup(tree* T, elem x)
2 //@requires is_tree(T);
3 {
4     if (T == NULL) return false;
5     int cmp = elem_compare(x, T->data);
6     if (cmp == 0) {
7         return true;
8     } else if (cmp < 0) {
9         return tree_lookup(T->left, x);
10    } else {
11        //@assert cmp > 0;
12        return tree_lookup(T->right, x);
13    }
14 }
```

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](#)).

6 Complexity

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. Every time we compare the element x with the root of a perfectly balanced tree, we either stop or throw out half the elements in the tree.

In general we can say that the cost of lookup is $O(h)$, where h is the *height* of the tree. We will define height to be the maximum number of nodes that can be reached by any sequence of pointers starting at the root. An empty tree has height 0, and a tree with two children has the maximum height of either child, plus 1.

7 The Interface

Before we talk about insertion into a binary search tree, we should specify the interface and discuss how we will implement it. Remember that we're assuming a single client definition of `elem` and a single client definition of

`elem_compare`, rather than the fully generic version using void pointers and function pointers.

```
/* Library interface */
// typedef _____* bst_t;

bst_t bst_new()
    /*@ensures \result != NULL; @*/ ;

void bst_insert(bst_t B, elem x)
    /*@requires B != NULL && x != NULL; @*/ ;

bool bst_lookup(bst_t B, elem x)
    /*@requires B != NULL && x != NULL; @*/ ;
```

We can't define `bst_t` to be `tree*`, for two reasons. One reason is that a new tree should be empty, but an empty tree is represented by the pointer `NULL`, which would violate the `bst_new` postcondition. More fundamentally, if `NULL` was the representation of an empty tree, there would be no way to write a function to imperatively insert additional elements in the tree. This is because a function call makes copies of the (small) values passed as arguments.

The usual solution here is one we have already used for stacks, queues, and hash tables: 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.


```
1 typedef struct bst_header bst;
2 struct bst_header {
3     tree* root;
4 };
5
6 bool is_bst(bst* B) {
7     return B != NULL && is_tree(B->root);
8 }
```

Lookup in a binary search tree then just calls the recursive function we've already defined:

```
10 bool bst_lookup(bst* B, elem x)
11 //@requires is_bst(B) && x != NULL;
12 {
13     return tree_lookup(B->root, x);
14 }
```

The relationship between both `is_bst` and `is_tree` and between `bst_lookup` and `tree_lookup` is a common one. The non-recursive function `is_bst` is given the non-recursive struct `bst_header`, and then calls the recursive helper function `is_tree` on the recursive structure of tree nodes.

8 Inserting an Element

With the header structure, it is straightforward to implement `bst_insert`. We just proceed as if we are looking for the given element. If we find a node with an equivalent element, we just overwrite its data field. Otherwise, we insert the new key 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 replace what it points to (it doesn't point to anything!). Instead, we *return* the new tree so that the parent can modify itself.

```
16 tree* tree_insert(tree* T, elem x)
17 //@requires is_tree(T) && x != NULL;
18 //@ensures is_tree(\result);
19 {
20     if (T == NULL) {
21         /* create new node and return it */
22         T = alloc(struct tree_node);
23         T->data = x;
24         T->left = NULL; // Not required (initialized to NULL)
25         T->right = NULL; // Not required (initialized to NULL)
26         return T;
27     } else {
28         int cmp = elem_compare(x, T->data);
29         if (cmp == 0) {
30             T->data = x;
31         } else if (cmp < 0) {
32             T->left = tree_insert(T->left, x);
33         } else {
34             //@assert cmp > 0;
35             T->right = tree_insert(T->right, x);
36         }
37     }
38
39     return T;
40 }
```

The returned subtree will also be stored as the new root:

```
42 void bst_insert(bst* B, elem x)
43 //@requires is_bst(B)
44 //@requires x != NULL;
45 //@ensures is_bst(B);
46 {
47     B->root = tree_insert(B->root, x);
48 }
```

9 Checking the Ordering Invariant

When we analyze the structure of the recursive functions implementing search and insert, we are tempted to try defining a simple, *but wrong!* ordering invariant for binary trees as follows: tree T is ordered whenever

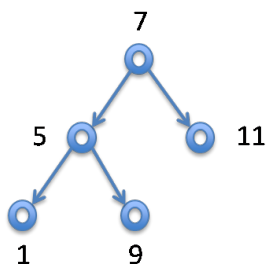
1. T is empty, or
2. T has key k at the root, T_L as left subtree and T_R as right subtree, and
 - T_L is empty, or T_L 's key is less than k and T_L is ordered; and
 - T_R is empty, or T_R 's key is greater than k and T_R is ordered.

This would yield the following code:

```
50 /* THIS CODE IS BUGGY */
51 bool is_ordered(tree* T) {
52     if (T == NULL) return true; /* an empty tree is a BST */
53     elem k = T->data;
54     return (T->left == NULL
55           || (elem_compare(T->left->data), k) < 0
56               && is_ordered(T->left))
57           && (T->right == NULL
58               || (elem_compare(k, T->right->data) < 0
59                   && is_ordered(T->right)));
60 }
```

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:

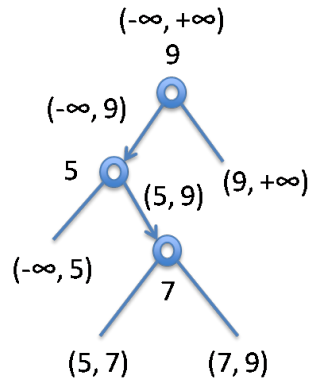


Even though, locally, the key of the left node is always smaller and on the right is always bigger, the node with key 9 is in the wrong place and we would not find it with our search algorithm since we would look in the right subtree of the root.

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 value, but the particular element from which we can extract the key that bounds the tree. Since `elem` must be a pointer type, this allows us to pass `NULL` in case there is no lower or upper bound.

```

50 bool is_ordered(tree* T, elem lower, elem upper) {
51     if (T == NULL) return true;
52     return T->data != NULL
53         && (lower == NULL || elem_compare(lower, T->data) < 0)
54         && (upper == NULL || elem_compare(T->data, upper) < 0)
55         && is_ordered(T->left, lower, T->data)
56         && is_ordered(T->right, T->data, upper);
57 }

```

This checks all the properties that our earlier `is_tree` checked, so we can just implement `is_tree` in terms of `is_ordered`:

```

59 bool is_tree(tree* T) {
60     return is_ordered(T, NULL, NULL);
61 }
62
63 bool is_bst(bst B) {
64     return B != NULL && is_tree(B->root);
65 }

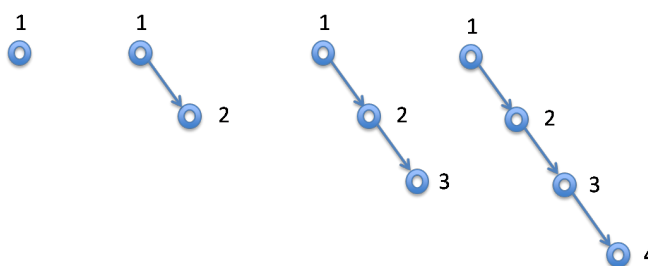
```

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 later in the course.

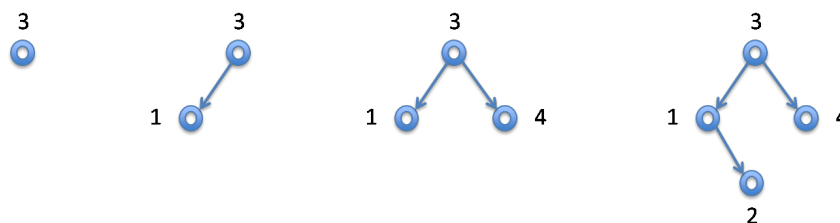
10 The Shape of Binary Search Trees

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 node 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?