15-122: Principles of Imperative Computation

Recitation 16 **Josh Zimmerman**

Binary search trees—a quick recap

Binary search trees are an implementation of *associative arrays* in a tree structure. We maintain the invariant (an *ordering* invariant) that everything to the left of a node is smaller than it and that everything to the right of a node is larger than it.

A brief aside: why would we use BSTs instead of hash tables? There are several reasons. As you may notice when working on peg solitaire, hash tables can be tricky to get right. Beyond that, it's also very difficult to write good hash functions that work in all cases. If we use a hash function that distributes keys poorly, we lose most of the benefit we get from using hash tables (since we'd have a lot of collisions). In addition, hash tables require us to allocate an array that will have many empty elements, which wastes memory.

BSTs also excel if you want to get a sorted list of the elements in your data structure.

Basically, there are some applications in which hash tables are preferable and some in which BSTs are preferable, so it's a matter of evaluating the specific case and deciding which data structure works better for it.

Let's play around with a binary search tree visualization so we can see how the ordering invariants work out in practice. We'll be using `http://www.cs.usfca.edu/~galles/visualization/BST.html`.

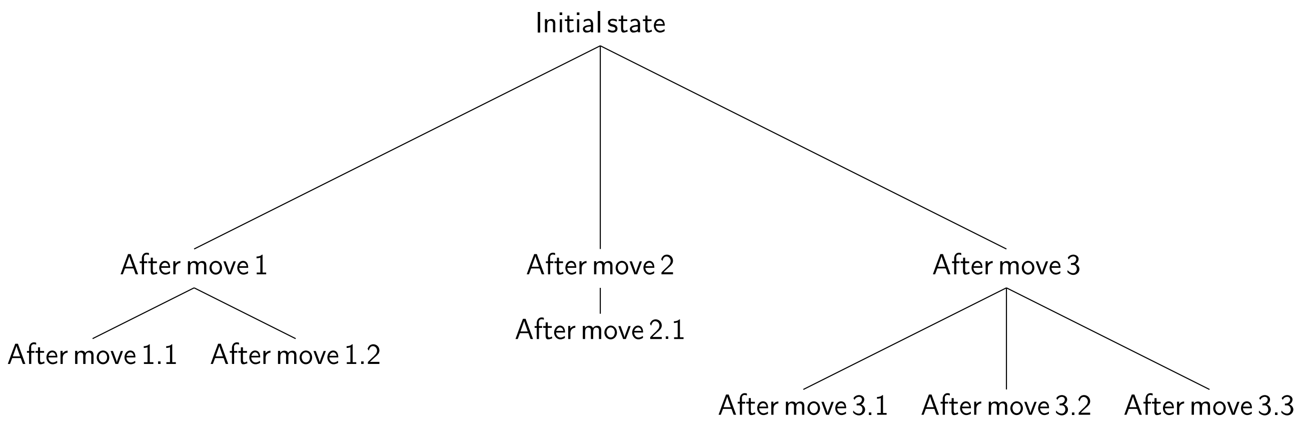
One important thing to note is that BSTs are not always balanced, and so we won't necessarily always have $O(\log(n))$ runtime for the common operations. For example, if we insert an increasing sequence of elements (for example, insert 1, then 2, then 3, then 4...) we don't have a balanced tree—in fact, what we have more closely resembles a linked list—so we don't have the good runtime we want. This is obviously not good, and we'll see how to fix the problem next week when we talk about self-balancing binary trees.

Backtracking

Backtracking search is very important to the next homework, so we're going to take a few minutes to review it.

The key idea behind backtracking search is that we try possibilities in some order (more on how we choose the order in a bit) and then *backtrack* if we get to a case where we can't possibly solve the problem we're trying to solve. In the case of peg solitaire, this could correspond to a board with no legal moves that isn't a winning board.

It's often conceptually useful to visualize backtracking search with a tree diagram, but we *do not* normally actually build a tree. It's useful for talking about how algorithms work, but in practice we use some combination of recursion, stacks, queues, or priority queues to implement a backtracking search.



We can traverse this tree of possible states in several different ways. We'll discuss *depth-first*, *breadth-first*, and *best-first*.

Depth-first simply goes as deep as possible: It will check move 1, then move 1.1, then move 1.2, then move 2 from the original, and so on. This can be implemented by putting possible states onto a stack: we see all possible moves from the initial state, push the states that result from making those moves onto a stack, and then pop something from the stack and start to examine it.

We are effectively traveling as far down the game tree as possible, and then backtrack back up it if we get to an unsolvable state.

One important thing to note about depth-first search is that it won't produce the answer that requires the fewest steps to get to. For instance, in peg solitaire, depth-first search will very frequently produce a sequence of moves to solve the game that is far longer than it needs to be. This doesn't mean it's inherently bad, but it's definitely something that's important to consider when choosing a search algorithm.

Breadth-first search traverses each level of the tree one at a time: It checks move 1, then move 2, then move 3, then move 1.1, then move 1.2, move 2.1, and so on. It will always find the solution with the smallest number of moves. This can be implemented by putting possible states onto a queue: the sooner we put things onto the queue, the sooner we'll evaluate them.

However, there's an algorithm that can often be better than either of these: best-first search. In best-first search, we have some function that estimates how good a node in the tree is, and use a priority queue to evaluate the *best* option first (functions that do this type of estimation are normally referred to as *heuristics*). It's worth noting that writing good heuristics is really, really hard in general, but you might be able to come up with one for peg solitaire that helps your code run more efficiently!

Note that best-first search is not necessary for the peg solitaire homework, but it's interesting to think about.

`is_ordered`

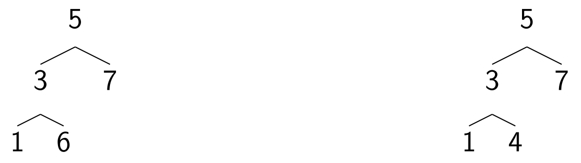
Here's the `is_ordered` code from lecture. Remember that we require that `elem` be a pointer to something. This allows us to use `NULL` as a special value that we can use in `bst_lookup` to indicate that the key was not found in the table.

Also remember that `key_compare(k1, k2)`'s return value is less than 0 if $k1 < k2$, is equal to 0 if $k1 = k2$, and is greater than 0 if $k1 > k2$. (You can remember this by thinking of the special case when $k1$ and $k2$ are integers and we can just return $k1 - k2$.)

```c
bool is_ordered(tree* T, elem lower, elem upper) {
   // Basic NULL checks
   if (T == NULL) return true;
   if (T->data == NULL) return false;
   key k = elem_key(T->data);
   // Check that the key is larger than the lower bound
   if (!(lower == NULL || key_compare(elem_key(lower), k) < 0)) {
      return false;
   }
   // Check that the key is smaller than the upper bound
   if (!(upper == NULL || key_compare(k, elem_key(upper)) < 0)) {
      return false;
   }
   return is_ordered(T->left, lower, T->data)
      && is_ordered(T->right, T->data, upper);
}
```

When we define the function this way we essentially establish a lower and an upper bound for what a BST key can be to be valid in the part of the BST we're currently examining. This at first seems like it's a more complicated invariant than what we actually need, but that is not the case.

Consider the following trees. Only the right one is a valid BST: the left one has 6 to the left of 5, which shouldn't happen, since $5 < 6$.



If we were to just check a parent against its two children, we'd say that both trees are valid, when in reality they aren't. So, when we traverse the tree, we need to maintain lower and upper bounds for the possible values of nodes. For instance, the node to the right of 3 must be between 3 and 5. And indeed, we see that the left tree violates this condition but the right one does not.

On the other hand, the node to the left of 3 must be between $-\infty$ and 3. In other words, there is no lower bound for it. We represent this by passing a lower bound of `NULL`. Similarly, when there is no upper bound, we pass an upper bound of `NULL`.

This is why our `is_ordtree` function is written as it is—we pass in `NULL` for both the lower and upper bound since we don't have a lower or upper bound for the first node: it can be anything.

```c
bool is_ordtree(tree* T) {
 // initially, we have no bounds - pass in NULL
 return is_ordered(T, NULL, NULL);
}
```

The `is_ordered` function is tricky, so let's spend a bit of time going over why it correctly checks the order invariant. Let k be the key that the node we're currently examining has.

The ordering invariant as we stated it in lecture is "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."

If we slightly rephrase this, we see that it's saying that if x is a key in the left subtree, then $x < k$, and that if y is a key in the right subtree, then $k < y$. In this way, we can see that the current node we're looking at establishes an *upper bound* for the nodes to the left of it and a *lower bound* for the nodes to the right of it.

If either the upper or lower bound is violated, we don't have a valid BST and can stop checking further. Otherwise, we'll only ever narrow our range as we get further down the BST, so we won't accidentally say that something invalid is valid.

`tree_insert`

Speaking of recursive functions, let's consider `tree_insert`.

```
 1 tree* tree_insert(tree* T, elem e)
 2 //@requires is_ordtree(T);
 3 //@requires e != NULL;
 4 //@ensures is_ordtree(\result);
 5 {
 6    if (T == NULL) {
 7       /* create new node and return it */
 8       T = alloc(struct tree_node);
 9       T->data = e;
10       T->left = NULL;
11       T->right = NULL;
12       return T;
13    }
14    int r = key_compare(elem_key(e), elem_key(T->data));
15    if (r == 0) {
16       T->data = e;   /* modify in place */
17    }
18    else if (r < 0) {
19       T->left = tree_insert(T->left, e);
20    }
21    else {
22       //@assert r > 0;
23       T->right = tree_insert(T->right, e);
24    }
25    return T;
26 }
```

"Hold on... why are we returning `T`?" you might ask. Note that on lines 19 and 23, we assign the right and left sub-trees to the result of calling the function recursively. If we didn't return `T` in all cases, that assignment wouldn't make sense. Also, we need to return `T` in the case where we enter into the `if` statement on line 6, since we have no way of modifying the tree that was passed in as an argument.

When we're inserting a new element into the BST, we traverse the tree until we find the location that the element we're inserting should go. Then, when we're at the base case (intuitively, when we've traversed until the correct location in the tree), we create a new node and return it. Then, we update the new node's parent to point to it and return back up the call stack updating pointers until we finally finish inserting the new element.

If you're confused about how exactly that works, I highly recommend playing with the animation I linked above and doing the proof of `tree_insert`'s correctness.

Proof that `tree_insert`'s postcondition holds.

To help you understand how `tree_insert` works, write a proof that it works.

As a reminder, when showing that recursive functions are correct, we first show partial correctness by first showing that in the base case the function is correct and then showing that if the function is correct on a smaller case it is correct on a larger case. We then show termination.

In this case, our steps will be as follows:

1. Show that the base case of the code (the case when `T == NULL`) is correct.
2. Show that if the recursive calls we make in the function (lines 19 and 23) are correct, then the function as a whole is correct.
3. Show that the function terminates on all input.