15-122: Principles of Imperative Computation

Recitation 11 Solutions

Josh Zimmerman, Arjun Hans, Nivedita Chopra

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). For every node in the tree:
  (a) All the elements in the left subtree have keys smaller than that of the node
  (b) All elements in the right subtree have keys greater than than that of the node
- The asymptotic complexity of insertion/lookup in a BST is $O(\log n)$. While this is slower than the $O(1)$ insertion/lookup time with hash tables, there are some good reasons why we may want to use BSTs instead:
  (a) Hash tables only have $O(1)$ insertion/lookup time if the hashing function has a good key distribution. Otherwise, if most of the elements hash to a small number of chains, insertion/lookup will slow down to linear time.
  (b) Also, hash tables require us to allocate large backbone arrays; a number of these may be empty throughout the hash table's usage, which wastes memory.

  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.
- 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. This is largely dependent on the order in which the keys are inserted.

Checkpoint 0

Insert the keys 1,2 and 3 in the order 2, 1, 3 and draw the tree formed. Then insert the keys in the order 1,2,3. What do you notice?

Solution: we see that tree is not balanced; it resembles a linked list,which has linear insertion/lookup time. We'll see how to fix the problem next week when we talk about self-balancing binary trees.

Playing with binary search trees

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`.

Try inserting these keys:

$$2, 4, 1, 3, 5$$

Then try deleting 2 and then 4 from the tree.

Discussing the code for insertion into the tree

Now let's consider the code for `tree_insert`. It's a recursive function.

```
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;
}
```

Note that in the cases where `r < 0` and `r > 0` we assign the right and left sub-trees to the result of calling the function recursively. Also, we need to return `T` in the case where we enter the case where `T == NULL` (an empty tree), since we have no way of modifying the tree that was passed in as an argument.

We can understand `tree_insert` as follows. Suppose that the element we want to insert is e

(1) If we reach an empty tree (`T == NULL`), then we just create a new node, set the data to e and return the node.

(2) Otherwise, we compare the key e_k of e to the key n_k of the node n we're currently at in the tree. We then have the following three cases:

   (i) If k_e and k_n are equal, then we just write over the data in the node n with e.

   (ii) If $k_e < k_n$, then element e belongs in the left-subtree of n. We therefore call `tree_insert` on the subtree `T->left` and insert e somewhere in `T->left`

   (iii) Otherwise, $k_e < k_n$, so the element e belongs in the right-subtree of n. We therefore call `tree_insert` on the subtree `T->right` and insert e somewhere in `T->right`.

Checkpoint 1

Recursion on binary search trees can be a tricky concept to master. Here are some practice problems to help you. Note that some of these problems only rely upon the structure of a BST, while others rely upon the BST ordering invariant. Your solution should use the struct definitions and the client-side functions we defined in class (given below for your reference).

Your solutions should be *short* (around 10 lines at most). You may also need a helper function to access the internals of the tree (as we did in `bst_insert` with `tree_insert`.

```
key elem_key(elem e);

int key_compare(key k1, key k2);

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

struct bst_header {
  tree* root;
};
typedef struct bst_header* bst;
```

(a) Write a function `elem bst_max(bst B)` that returns the element with the maximum key in a given BST.

Solution:

```
// Solution 1: iterative.
elem bst_max(bst B)
//@requires is_bst(B);
{
  tree curr = B->root;
  if (curr == NULL) return NULL;  // Empty tree.
  while (curr->right != NULL) {
    curr = curr->right;
  }
  return curr->data;
}

// Solution 2: recursive.
elem tree_max(tree T)
//@requires T != NULL;
{
  if (T->right == NULL) return T->data;
  return tree_max(T->right);
```

```
}
elem bst_max(bst B)
//@requires is_bst(B);
{
  if (B->root == NULL) return NULL;  // Empty tree.
  return tree_max(B->root);
}
```

(b) Write a function `int count_leaves(bst B)` that counts the number of leaves in a given BST.

Solution:

```
int count_tree_leaves(tree T) {
  if(T == NULL) return 0;
  if(T->left == NULL && T->right == NULL) return 1;
  return count_tree_leaves(T->left) + count_tree_leaves(T->right);
}
int count_leaves(bst B)
//@requires is_bst(B);
{
  return count_tree_leaves(B->root);
}
```