15-122 : Principles of Imperative Computation

Fall 2012

Assignment 6 – Selected Solutions

(Theory Part)

The following solutions are provided to you to help you study this semester. They are not to be distributed to others outside of the class nor are they intended to be used by students in future semesters as a substitute for completing one's own homework assignments.

1. **BSTs.**

(0) (a) Omitted.

(9) (b) The pre- and post-conditions for the `tree_insert` function in `bst.c0` that we discussed in lecture and recitation are not actually strong enough to prove the correctness of the recursive function! (See page **L17.9** in the lecture notes.) Proving the correctness of `tree_insert` requires us to add two *auxiliary arguments* to the function that are used only to reason about the function's correctness.

Trying to understand the proof on the next page will help you figure out the missing code below, and the code will help you fill out the missing parts of the proof. Every blank in the proof must be filled by writing `P.C.#` (for preconditions) or `B.C.#` (for branch conditions) for some number `#`.

This problem asks you to write down very little. That does not mean it is easy to get correct. Make sure everything you reason through everything you write down.

Solution: *(Code)* 3 points total

```
tree* tree_insert(tree* T, elem e, elem lower, elem upper)
//P.C.1:
//@requires e != NULL;
//P.C.2:
//@requires ____is_ordered(T, lower, upper)____;
//P.C.3:
/*@requires (lower == NULL)
   || __key_compare(elem_key(lower), elem_key(e)) < 0__; @*/
//P.C.4:
/*@requires (upper == NULL)
   || __key_compare(elem_key(e), elem_key(upper)) < 0__; @*/
//@ensures is_ordered(\result, lower, upper);
{
  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) //B.C.1
    T->data = e; /* modify in place */
  else if (r < 0) //B.C.2
    T->left = tree_insert(T->left, e, __lower__, __T->data___);
  else //@assert r > 0;
    T->right = tree_insert(T->right, e, __T->data__, __upper__);
  return T;
}
```

Strategy for the code part: Whenever you are writing preconditions for a function, your primary concern should be whether or not the preconditions (along with the loop invariant, if there is one) are **strong enough** to prove the postcondition. While there are many possible weak preconditions that would still remain true when entering the function `tree_insert`, weaker preconditions will not allow you to reason that `is_ordered(\result, lower, upper)` holds upon completion. It is also helpful to remember the types of data structure invariants used on other tree functions presented in class. The correct preconditions are only small variations on usage patterns seen in class.

Solution: *(Proof)*

In the first case where `T` is `NULL`, we return a tree with one element, `e`. We know this key is in the interval (*klower*, *kupper*) by P.C.3 and P.C.4. This means that the postcondition is satisfied.

In the other case where `T` is not `NULL`, let *kmid* be `elem_key(T->data)` – we know `T->data` is not `NULL` as a consequence of `P.C.2`. There are three sub-cases, depending on the value of `r`: either `r` $= 0$, `r` < 0, or `r` > 0. *(We omit the third case where* `r` > 0 *here. You are welcome to think through and write out how that case would work as practice, but do not include this in the homework.)*

- If `r` $= 0$, then `elem_key(e)` $=$ *kmid*; we refer to this fact as `B.C.1`. In this sub-case, we return a tree with the exact same keys as the tree we got, so the postcondition holds immediately by P.C.2.
- If `r` < 0, we know `elem_key(e)` $<$ *kmid*; we refer to this fact as `B.C.2`.

  First we must show that the preconditions of the recursive call are met.

  – The first precondition of the recursive call is true immediately because of `P.C.1` (`e` has not changed).

  – The second precondition of the recursive call demands that all the keys in `T->left` are in the interval (*klower*, *kmid*). This is a consequence of P.C.2 and the definition of *kmid*.

  – The third and fourth preconditions of the recursive call follow from P.C.3 and B.C.2, respectively.

  We return our original tree where the left subtree has been replaced by the tree we got from the recursive call. Because of the recursive call's postcondition, we know the modified `T->left` is in the interval (*klower*, *kmid*). Because of P.C.2, we know the unchanged `T->right` is in the interval (*kmid*, *kupper*). We didn't change `T->data` in this sub-case, so when we return `T` it is in the interval (*klower*, *kupper*) as required.

Strategy for the proof part: The key in this part was devising the correct contracts in the previous code section; those decisions should have been informed both by the structure of the proof given to you and reasoning about boundary cases that must be addressed to ensure that each statement in the proof always holds. A good strategy for checking the correctness of your proof is to read over your answers and consider what pathological cases you must guard against to make each claim in the proof true for all inputs that satisfy the preconditions – in some sense, "unit test" your proof. If there exists any input that satisfies a precondition that you mention in your proof but that does not match the claim being made in the proof, then you have chosen the wrong precondition or your precondition is not strong enough.