15-122 : Principles of Imperative Computation

Fall 2012

Assignment 6

(Theory Part)

Due: Thursday, November 8, 2012 at the **beginning** of lecture

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework will give you some practice working with hash tables, heaps and priority queues. You can either type up your solutions or write them *neatly* by hand in the spaces provided. You should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	7	
2	15	
3	3	
Total:	25	

You *must* include this cover sheet.
Either type up the assignment using
15-122-theory6.tex, or print out this PDF.

1. **Heaps and BSTs.**

Though heaps and binary search trees (BSTs) are very different in terms of their invariants and uses, they are both conceptually represented as trees. This question asks about three invariants of trees: the BST ordering invariant, the heap shape invariant, and the heap ordering invariant (for min-heaps, where higher-priority keys are lower integer values). For the first part of this question, we assume that each element has a single C0 `int` that is used as both the BST key and the heap priority.

(1) (a) Draw a non-empty tree that is both a (min-)heap and BST. *Hint: is it possible for a tree with three elements to satisfy all three invariants?*

Solution:

(1) (b) Draw a tree with five elements that is a BST and satisfies the heap shape invariant.

Solution:

(1) (c) Draw a tree with at least four elements that is a BST and satisfies the (min-)heap ordering invariant.

Solution:

(1) (d) Why is it not a good idea to have a data structure that enforces both the (min-)heap ordering invariant and the BST ordering invariant? (Be brief.)

Solution:

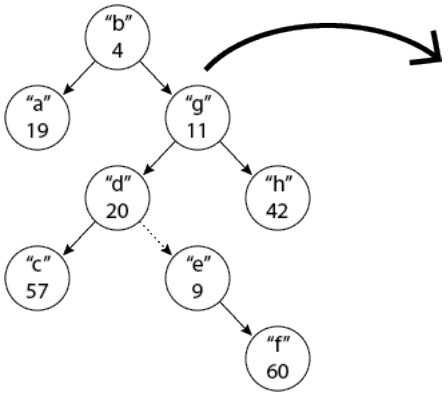
(3) (e) Maintaining the BST ordering invariant and the heap invariant on the *same* set of values may not be a good idea, but it can be useful to have a tree structure where each node has two separate values – a key used for the BST ordering invariant and a priority used for the heap ordering invariant. Such trees are called *treaps*; we will use strings as keys and integers as priorities in this question.

The treap below satisfies the BST ordering invariant, but violates the heap ordering invariant because of the relationship between the "e"/9 node and its parent. In a heap, we restore the heap shape invariant by using swaps; in a treap, such a swap would violate the BST ordering invariant. However, by using the same local rotations we learned about for AVL trees, it is possible to restore the heap ordering invariant while preserving the BST ordering invariant.

The heap ordering invariant for the tree below can be restored with two tree rotations. After one rotation, the heap ordering invariant will again hold except at one place. After a second rotation, the heap ordering invariant will be fully restored.

Draw the tree that results from each rotation. You should be drawing two trees.

Solution:



2. **BSTs.**

In this question, we will think about the implementation of binary search trees. The `is_bst` and `is_ordered` functions are implemented as follows:

```
bool is_ordered(tree* T, elem lower, elem upper) {
  if (T == NULL) return true;
  if (T->data == NULL) return false;
  key k = elem_key(T->data);
  if (!(lower == NULL || key_compare(elem_key(lower),k) < 0))
    return false;
  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);
}

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

(2) (a) It is possible to implement `bst_search` as an iterative function rather than a recursive one. Your loop condition should imply that the `@ensures` clause holds if and when the loop terminates.

Solution:

```
elem bst_search(bst B, key k)
//@requires is_bst(B);
/*@ensures \result == NULL
        || key_compare(k, elem_key(\result)) == 0; @*/
{
  tree T = B->root;

  while (_____________________________________________________)
  {
    if (_______________________________)
      T = T->left;
    else
      T = T->right;
  }

  if (T == NULL) return NULL;
  return T->data;
}
```

(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)*

```
tree* tree_insert(tree* T, elem e, elem lower, elem upper)
//P.C.1:
//@requires e != NULL;
//P.C.2:
//@requires ____________________________________________________;
//P.C.3:
/*@requires (lower == NULL)

   || ____________________________________________________; @*/
//P.C.4:
/*@requires (upper == NULL)

   || ____________________________________________________; @*/
//@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)
    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;
}
```

Solution: *(Proof)*
Let *klower* be either `elem_key(lower)` (if `lower` is not `NULL`) or $-\infty$ (if `lower` is `NULL`). Similarly, let *kupper* be either `elem_key(upper)` (if `upper` is not `NULL`) or ∞ (if `upper` is `NULL`).

There are two cases: either `T` is `NULL` or it is not.

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 [ ] and [ ]. 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 [ ].
- 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 [ ] and the definition of *kmid*.
  - The third and fourth preconditions of the recursive call follow from [ ] and [ ], 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 [ ], 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.

(4) (c) The ordering invariant of BSTs makes it possible to print the elements of the BST in descending order, without having to use a temporary data structure to store and/or sort any of the elements. Write a function `bst_print_desc(bst B)` that prints the elements of a BST in *descending order* using a helper function `tree_print_desc(tree*T)`.

You can assume that you have a client-provided function `elem_print(elem e)` that prints an element. Your solution should be simple and straightforward, so try hard to think of a way to do this elegantly; unnecessarily convoluted functions will lose points.

Solution:

```
void bst_print_desc(bst B)
//@requires is_bst(B);
{

      ____________________________________________________________;
      return;
}

void tree_print_desc(tree* T)
//@requires is_ordered(T, NULL, NULL);
{









}
```

3. **AVL Trees.**

(3) (a) Draw the AVL trees that results after successively inserting the following keys into an initially empty tree, in the order shown.

```
98, 88, 54, 67, 23, 72, 39
```

We want you to show the tree after each insertion and subsequent re-balancing (if any) – the tree after the first element, `98`, is inserted into an empty tree, then the tree after `88` is inserted into the first tree, and so on for a total of seven trees. Make it clear what order the trees are in.

Be sure to maintain and restore the BST invariants and the additional balance invariant required for an AVL tree after every insert.

Solution: