15-122: Principles of Imperative Computation, Fall 2010 Assignment 6: Sample Solution

Karl Naden (kbn@cs)

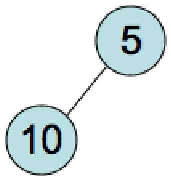
March 29, 2011

1 Written: (25 points)

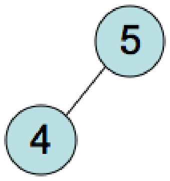
1.1 Heaps and BSTs

Exercise 1 (3 pts)**.** Draw a tree that matches each of the following descriptions.

a) Draw a (min) heap that is not a BST.



b) Draw a BST that is not a (min) heap.



c) Draw a non-empty BST that is also a (min) heap.

5

1.2 Binary Search Trees

Refer to the binary search tree code posted on the course website for Lecture 17 if you need to remind yourself how BSTs work.

Exercise 2 (4 pts)**.** The height of a binary search tree (or any binary tree for that matter) is the number of levels it has. (An empty binary tree has height 0.)

(a) Write a function `bst_height` that returns the height of a binary search tree. You will need a wrapper function with the following specification:

```
int bst_height(bst B);
```

This function should not be recursive, but it will require a helper function that will be recursive. See the BST code for examples of wrapper functions and recursive helper functions.

Solution:

```
int bst_height(bst B)
//@requires is_bst(B);
{
   return tree_height(B->root);
}

int tree_height(tree T)
{
   if (T == NULL) return 0;
   int left_height = tree_height(T->left);
   int right_height = tree_height(T->right);
   return left_height < right_height ? right_height+1 : left_height+1;
}
```

NOTE: we use the ternary operator, which has the general form `e ? e1 : e2`, where `e` has type `bool` and `e1` and `e2` can have any type, but they must the same. The semantics are that if the boolean expression before the '?' evaluates to true, the ternary expression evaluates and takes the value of the first subexpression after the '?'. Otherwise, it evaluates and takes the value of the second expression which follows the ':'. For example, `1 > 0 ? 1 : 0` would return 0 because `1 > 0` is false and 0 is the second expression.

(b) Why is recursion preferred over iteration for this problem?

Solution:

1. The code is simpler because recursion matches the structure of the tree so you do not need to explicitly keep track of how you traverse a path to each leaf.
2. easier to parallelize

Exercise 3 (5 pts)**.** In this exercise you will re-implement `bst_search` in an iterative fashion.

a) Complete the `bst_search` function below that uses iteration instead of recursion.

Solution:

```
elem bst_search(bst B, key k)
//@requires is_bst(B);
//@ensures \result == NULL || compare(k, elem_key(\result)) == 0;
{
    tree T = B->root;
    while (___T != NULL && compare(k,elem_key(T->data)) != 0___)
    {
        if (___compare(k,elem_key(T->data)) < 0___)
            T = T->left;
        else
            T = T->right;
    }
    if (T == NULL) return NULL;
    return T->data;
}
```

b) Using your loop condition, prove that the ensures clause of `bst_search` must hold when it returns.

Solution:

When the loop exits, you know that the negation of the loop condition is true, or:

```
T == NULL || compare(k,elem_key(T->data)) == 0
```

Now, executing from there, we have two cases:

- if `T == NULL`, then the `if` condition is true and so `\result == NULL` and the `@ensures` clause is satisfied.
- if `T != NULL`, then `\result == T->data`, but since `T != NULL`, we know by the negation of the loop condition that `compare(k,elem_key(T->data)) == 0`, thus, we know `compare(k,elem_key(\result)) == 0` and so the `@ensures` clause is satisfied in this case as well.

1.3 Rotations and AVL Trees

Exercise 4 (3 pts)**.** Consider the following function to perform a rotate right operation:

```
tree tree_rotate_right(tree T) {
    tree newRight = alloc(Struct tree);
    newRight-> data = T->data;
    newRight->right = T->right;
    newRight->left = T->left->right;

    tree newRoot = alloc(struct tree);
    newRoot->data = T->left->data;
    newRoot->left = T->left->left;
    newRoot->right = newRight;

    return newRoot;
}
```

Give `requires` and `ensures` annotations for the `tree_rotate_right` function. Your annotations should include tests on the ordering of the tree and the height of the tree, before and after the rotation. The function should not work only on AVL trees, but on BSTs in general. You may assume the following functions are defined for you:

```
bool is_ordered(tree T, elem min, elem max);
int height(tree T);
```

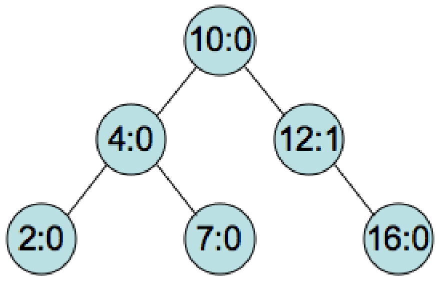
Solution:

```
//@requires is_ordered(T, NULL, NULL);
//@requires T != NULL && T->left != NULL;
//@ensures is_ordered(\result, NULL, NULL);
//@ensures \result != NULL && \result->right != NULL;
//@ensures tree_height(\result) == tree_height(T) ||
           tree_height(\result) == tree_height(T)+1 ||
           tree_height(\result) == tree_height(T)-1;
```

Exercise 5 (3 pts)**.** Draw the AVL tree that results after successively inserting the following keys (in the order shown) into an initially empty tree, maintaining and restoring the invariants of a BST and the additional balance invariant required for an AVL tree after every insert.

```
7  12  10  16  4  2
```

Your answer should show the tree after each key is successfully inserted. Also be sure to label each node with its *balance factor*, which is defined as the height of the right subtree minus the height of the left subtree.



Exercise 6 (5 pts)**.** During lecture 18, we showed that each rotation of a binary tree takes constant time. Since the height of an AVL tree is guaranteed to be $O(\log n)$ and the balance invariant can only be broken along the path from the location a new element was inserted to the root, we concluded that to restore the balance invariant we will need to do at most $O(\log n)$ rotations. However, it is an important property of AVL trees that restoring the balance invariant is achieved in one (double or single) rotation. It takes $O(\log n)$ time to find the place in the tree to insert the new element, so the `insert` operation is still $O(\log n)$ overall, but fewer rotations mean a smaller constant factor of overhead, which is an important practical consideration.

In this exercise we will go through several steps to prove that we only need to perform at most one rotation following an insertion into an AVL tree. The basic idea is to show that any rotation of an unbalanced tree created from an insertion of a new element into a balanced AVL tree must reduce the height of the rotated tree. In lecture, we considered three cases after insertion on the left. In two of these (not shown here), the height of the tree is reduced by one during the rotation. The third has the form show in Figure 1.

a) Draw the result of rotating the tree in Figure 1 to the right and explain why the overall height of the tree is not reduced.

Solution:

See the rotated tree in figure 2. The height of the tree didn't change because b is of height $h + 1$ and after a single rotation b will still be a child of both x and y and so x must still have height $h + 3$.

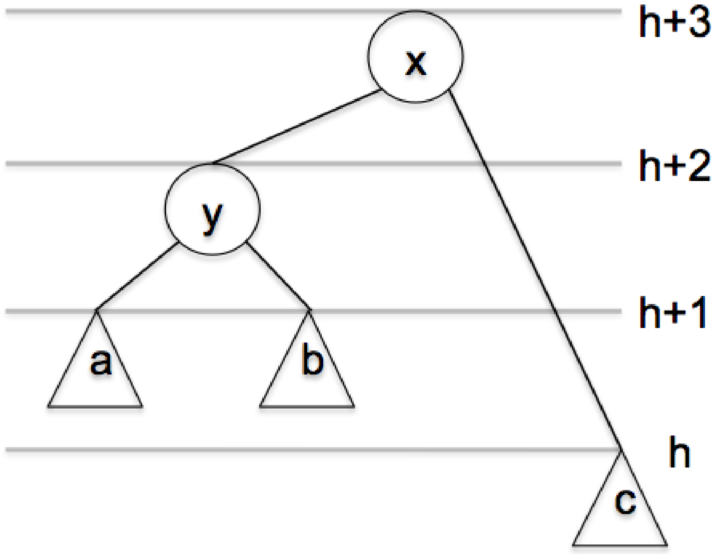


Figure 1: An unbalanced AVL tree

b) Explain why the case depicted above could not occur after the insertion of a single element into a balanced AVL tree prior to any rebalancing.

Solution: The situation depicted in Figure 1 could not occur after a single insertion into a balanced AVL tree because before the insertion, the tree must have been balanced. If it was balanced, then both sub trees a and b must have had height h or less. But since we only inserted a single element, we could have increased the height of a or b, but not both. Thus, the situation could never occur

c) Given that the other rotations reduce the height of the tree, argue that the parent of the rotated tree cannot be unbalanced. (**Hint:** Think about how the height of the subtree has changed between during the insertion and rotation)

Solution:

We know that before the insertion, all nodes in the tree must be balanced. Thus, if a node is unbalanced, it must be unbalanced towards the side where the new element was inserted. However, since the rotation reduced the height of the subtree in which the new element was inserted, we know that the height of that subtree must be the same as it was prior to the insertion. Thus, the node must still be balanced.

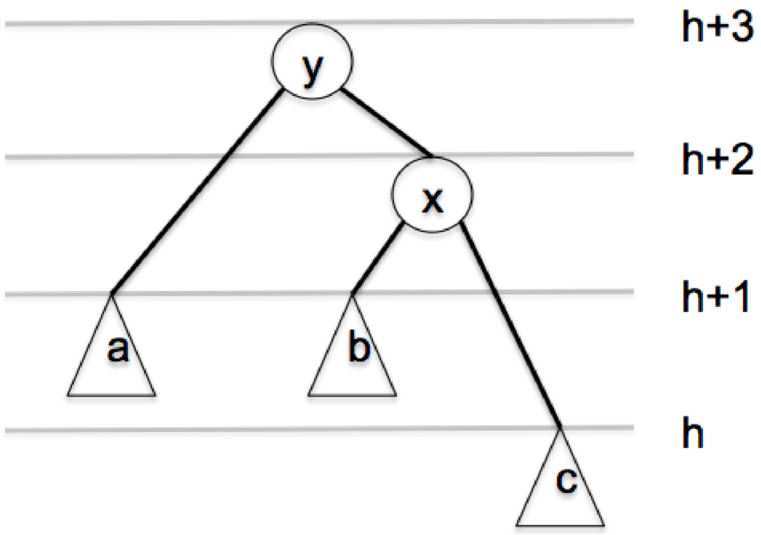


Figure 2: 6.a) Solution - still the same height but balanced

1.4 Testing

Exercise 7 (2 pts)**.** Identify the primary difference between black box and glass box testing in terms of what information is used to design the tests. How does this difference impact what can be effectively tested with each approach?

Solution: The main difference between black box and glass box testing is that in glass box testing, you know exactly what code is executed. Several potential differences in what you can test include:

- a set of glass box tests can only be used to test a particular implementation, not any code where as black box tests can test any code that fits the specification the tests were written for.
- Glass box testing can test details of the implementation that black box testing cannot
- If the specification is under-specified, then glass box testing can be much more thorough in its tests