15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 9

Due: Thursday, October 31, 2013, at 10pm

Name: _____

Andrew ID: _____

Recitation: _____

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

Question	Points	Score
1	3	
2	3	
3	9	
Total:	15	

Write your answers *neatly* on this PDF (or fill out the TeX handout), and then submit the stapled printout to the handin box Thursday before lecture or on Thursday afternoon outside of Tom Cortina's office (GHC 4117).

1. **Heaps**

   Refer to the implementation of heaps discussed in class that is available on our course website.

(1) (a) Add a meaningful assetion about H to each of the functions below.

Solution:

```
void pq_insert(heap H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert ________________________________________________;
  int i = H->next - 1;
  while (i > 1 && priority(H,i) < priority(H,i/2))
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant is_heap_except_up(H, i);
    {
      swap(H->data, i, i/2);
      i = i/2;
    }
  //@assert is_heap(H);
  return;
}

elem pq_delmin(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
  int n = H->next;
  elem min = H->data[1];
  H->data[1] = H->data[n-1];
  H->next = n-1;
  if (H->next > 1) {
      //@assert ____________________________________________;
      sift_down(H, 1);
  }
  return min;
}
```

(1) (b) Complete an additional library function, `pq_max`, that returns, but does not remove, the element with the maximum priority value from our array-based min-heap. We have provided part of the function for you. You should examine only those elements that might contain the maximum. (Note that this is not an operation you would want to provide for a min-heap priority queue due to its runtime complexity.)

Solution:

```
elem pq_max(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
   int max = ______________________________;

   for (int i = ____________________; i < ___________________; i++)

       if (priority(H, i) > priority(H, max))  max = i;

   return ______________________________;
}
```

(1) (c) The library function, `pq_build`, shown below, takes an array of data elements (ignoring index 0 of the array) and builds our array-based min-heap *in place*. That is, it uses the given array in our heap structure and does not allocate a new array.

```
heap pq_build(elem[] elements, int arraylength)
//@requires \length(elements) > 0 && \length(elements)==arraylength;
//@ensures is_heap(\result);
{
  heap H = alloc(struct heap_header);
  H->limit = arraylength;
  H->next = 1;
  H->data = elements;
  for (int i = 1; i < arraylength; i++)
      pq_insert(H, elements[i]);
  return H;
}
```

The function above does not respect the boundary between the client and the library. Complete the following client code so that the `pq_empty` function will likely abort by failing its precondition. In your solution, do not deference `H` or set `H` to `NULL`.

Solution:

```
:
:
//@assert \length(E) = 16;
heap H = pq_build(E, 16);

----------------------------------------------------------------

----------------------------------------------------------------

----------------------------------------------------------------

if (pq_empty(H)) return;
:
:
```

2. **Binary Search Trees and Heaps**

(1) (a) Draw the binary search tree that results from inserting the following keys in the order given:

`75 92 99 13 84 42 71 98 73 20`

Be sure all branches in your tree are clearly drawn so we can distinguish left branches from right branches.

Solution:

(2) (b) How many different binary search trees can be constructed using the following five keys: `73, 28, 52, -9, 104` if they can inserted in any arbitrary order?

Solution:

How many different min-heaps can be constructed using the following five keys: `73, 28, 52, -9, 104` if they can be inserted in any arbitrary order?

Solution:

Consider the shape of a binary tree. How many non-empty binary search tree configurations can also be min-heaps?

Solution:

Consider the shape of a binary tree. How many non-empty binary search tree configurations can also be max-heaps?

Solution:

3. **Binary Search Trees: Library Functions**

Refer to the implementation of binary search trees discussed in class that is available on our course website.

(3) (a) Write an implementation of a new library function, `bst_height`, that returns the height of a binary search tree. The height of a binary search tree is defined as the maximum number of nodes as you follow a path from the root to a leaf. As a result, the height of an empty binary search tree is 0. Your function must include a **recursive** helper function `tree_height`.

HINT: In general, the height of a tree rooted at node T is one more than the height of its deepest subtree.

Solution:

```
int tree_height(tree* T)
//@requires is_ordered(T, NULL, NULL);
{









}

int bst_height(bst B)
//@requires is_bst(B);
//@ensures is_bst(B);
{
       return ____________________________________________;
}
```

(6) (b) Consider extending the BST library implementation with the following function which deletes an element from the tree with the given key.

```
void bst_delete(bst B, key k)
//@requires is_bst(B);
//@ensures is_bst(B);
{
        B->root = tree_delete(B->root, key k);
}
```

Complete the code for the recursive helper function `tree_delete` which is used by the `bst_delete` function. This function should return a pointer to the tree rooted at `T` once the key is deleted (if it is in the tree).

You will need to complete an additional helper function `largest_child` that removes and returns the largest child rooted at a given tree node T.

Solution:

```
tree* tree_delete(tree* T, key k)
{
    if (T == NULL) {                    // key is not in the tree

        return _______________________________;

    }

    if (key_compare(k, elem_key(T->data)) < 0) {

       ________________________ = tree_delete(T->left, k);

       return T;

    } else if (key_compare(k, elem_key(T->data)) > 0) {

       ________________________ = tree_delete(T->right, k);

       return T;

    } else {      // key is in current tree node T

        if (T->left == NULL)        // node has only right child

            return ___________________________;

        else if (T->right == NULL)   // node has only left child

            return ___________________________;
```

```
        else {         // Node to be deleted has two children

            if (T->left->right == NULL) {

                // Replace the data in T with the data
                // in the left child.

                ____________________________________________;

                // Replace the left child with its left child.

                ____________________________________________;

               return T;
            }
            else {
                // Search for the largest child in the
                // left subtree of T and replace the data
                // in node T with this data after removing
                // the largest child in the left subtree.
                T->data = largest_child(T->left);
                return T;
            }
        }
    }
}

elem largest_child(tree* T)
//@requires T != NULL && T->right != NULL;
{
    if (T->right->right == NULL) {

        elem e = ___________________________________;

        T->right = ___________________________________;

        return e;
    }

    return largest_child(_____________________________________);
}
```