15-122 : Principles of Imperative Computation, Summer 1 2014

Written Homework 5

Due: Thursday, June 19 before recitation

Name: _____

Andrew ID: _____

Recitation: _____

Binary search trees, AVL trees, heaps, priority queues, and more heaps.

Question	Points	Score
1	10	
2	7	
3	8	
4	4	
5	6	
Total:	35	

You *must* print this PDF and write your answers *neatly* by hand.
You should hand in the assignment before recitation begins.

Part I: High Up in a Tree

For questions 1 and 2, we define the height of a tree as **the number of nodes on the longest path between the root and a leaf, inclusive**. For example, the empty tree has height 0, a tree with 1 node has height 1, and a balanced tree with 3 nodes has height 2.

1. **Binary Search Trees**

(1) (a) Draw the binary search tree that results from inserting the following keys in the order given. Be sure all branches in your tree are drawn *clearly* so we can distinguish left branches from right branches.

$$75,\ 92,\ 99,\ 13,\ 84,\ 42,\ 71,\ 98,\ 73,\ 20$$

Solution:

(1) (b) How many different binary search trees can be constructed using the following five keys if they can be inserted in any order?

$$73,\ 28,\ 52,\ -9,\ 104$$

Solution:

(c) For the following questions, you should refer to the implementation of binary search trees discussed in class. The code is available on the course website.

(3) i. Write the function `bst_height` which returns the height of the given BST, as per the definition given on page 2. Your function must include a **recursive** helper function `tree_height`.
(*Hint*: the height of a tree 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 ____________________________________________________;
}
```

(1) ii. Complete the code for the function `largest_child` which removes and returns the largest child rooted at a given tree node `T`.

Solution:
```
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(_____________________________);
}
```

(4) iii. 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).

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)

      return ________________________;
    else if (T->right == NULL)

      return ________________________;
    else {

      // continued on next page...
```

```
      // T has two children
      if (T->left->right == NULL) {
          // Replace T's data with the left child's data.

          ____________________________________________;
          // 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;
      }
    }
  }
}
```

2. **AVL Trees.**

(3) (a) Insert the following values into an *initially empty* AVL tree one at a time in the order shown. Draw the final state of the AVL tree after each insert is completed and the tree is restored back to its proper invariants (do not draw the intermediate steps of any rebalancing operations). Your answer should show exactly 7 clearly drawn trees.

89, 79, 45, 58, 10, 63, 31

Solution:

(b) We want to determine upper and lower bounds on the number of nodes in an AVL tree of a certain height. Let $m(h)$ be the **minimum** number of nodes in a valid AVL tree of height h. Let $M(h)$ be similarly defined for the **maximum** number of nodes.

(2) i. Fill in the table below relating the values h and $m(h)$:

h	$m(h)$
0	0
1	1
2	2
3	
4	
5	
6	

(1) ii. Guided by the table in part (i), complete the *recursive* definition of $m(h)$. The base cases have been completed for you. (*Hint*: think Fibonacci.)

Solution:

$$m(h) = \begin{cases} 0 & \text{if } h = 0 \\ 1 & \text{if } h = 1 \\ \underline{\qquad\qquad\qquad\qquad\qquad\qquad} & \text{if } h \geq 2 \end{cases}$$

(1) iii. Give a closed form expression for $M(h)$.

Solution:

$M(h) =$

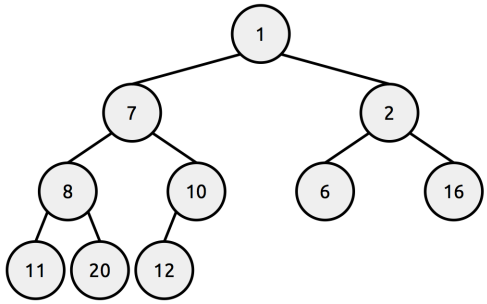
Part II: Heaps of Work

3. **Piles of Theory**

As discussed in class, a *min-heap* is a hierarchical data structure that satisfies two invariants:

> **Order**: Every child has value greater than or equal to its parent.
>
> **Shape**: Each level of the min-heap is completely full except possibly the last level, which has all of its elements stored as far left as possible. (Also known as a *complete* binary tree).

Consider:



(1) (a) Draw a picture of the final state of the min-heap after an element with value 5 is inserted. Be sure to satisfy both of the data structure invariants for a min-heap.

Solution:

(1) (b) Starting from the *original* min-heap above, draw a picture of the final state of the min-heap after the element with the minimum value is deleted. Be sure to satisfy both of the data structure invariants for a min-heap.

Solution:

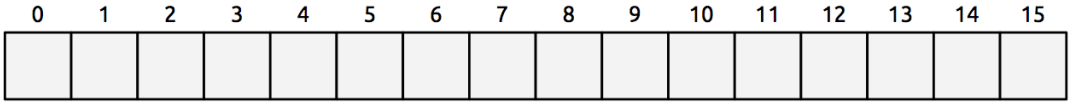
(2) (c) Insert the following values into an *initially empty* min-heap one at a time in the order shown. Draw the final state of the min-heap after each insert is completed and the min-heap is restored back to its proper invariants. Your answer should show 8 clearly drawn heaps.

$$42,\ 19,\ 71,\ 38,\ 20,\ 6,\ 55,\ 10$$

Solution:

(1) (d) Assume a heap is stored in an array as discussed in class. Using the final min-heap from your previous answer, show where each element would be stored in the array. You may not need to use all of the array positions shown below.

Solution:



(1) (e) You are given a non-empty min-heap. In one sentence, describe precisely where the maximum value must be located. Do not assume the heap is implemented as an array. (Your vocabulary should pertain only to the tree definition of a heap).

Solution:

(1) (f) What is the worst-case runtime complexity of finding the maximum in a min-heap if the min-heap has n elements? Why?

Solution: $O($ $)$

Because the number of values that need to be examined is:

(1) (g) We are given an array A of n integers. Consider the following sorting algorithm:

- Insert every integer from A into a min-heap.
- Repeatedly delete the minimum from the heap, storing the deleted values back into A from left to right.

What is the worst-case runtime complexity of this algorithm, using Big-O notation? Briefly explain your answer.

Solution: $O($ $)$

4. **Jumbles of Code**

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

(2) (a) Add a meaningful assertion 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 the function `pq_max`, that returns (but does not remove) the element with the maximum value from a min-heap stored as an array. *You should examine only those elements that might contain the maximum.*

Solution:

```
elem heap_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 inside of the heap structure rather than allocating a new array.

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

This code disrespects the boundary between the client and the library. Complete the function `build_broken_heap` below such that the postcondition will always succeed.

Solution:

```
heap build_broken_heap(int[] E, int n)
//@requires 3 <= n && n <= \length(E);
//@ensures !is_heap(\result);
{
  heap H = pq_build(E, n);

  E[1] = ____________________;

  E[2] = ____________________;
  return H;
}
```

5. **Priority Queues**

(4) (a) You are working an exciting desk job as a stock market analyst. You want to be able to determine the total sum value of all of the top stocks at any time. However, since the year is 1983, your Commodore 64 can only offer up about 30 KB of memory.

Stock reports are delivered to you via a `stream` data type with the following interface:

```
struct stock_report {
  string company;
  int value;   // stock value in whole dollars
};
typedef struct stock_report* report;

// Retrieve the next stock report from the data stream
report get_report(stream S);
// Returns true if the data stream is empty
bool stream_empty(stream S);
```

A stream of stock reports could be very, very large. Storing all of the reports in an array won't cut it – you don't have enough memory (30 KB isn't even enough to store 2000 reports). You'll need a more clever solution.

Luckily, your cubicle mate Jim just finished a stellar heap implementation with the interface below. You think you should be able to use Jim's priority queue to keep track of only the largest stock reports, discarding the smaller ones as necessary.

```
// Client Interface
typedef _________ elem;
int elem_priority(elem e) /*@requires e != NULL; @*/;

// Library Interface
typedef _________ pq;
pq pq_new(int capacity) /*@requires capacity > 0; @*/;
bool pq_full(pq Q);
bool pq_empty(pq Q);
void pq_insert(pq Q, elem e) /*@requires !pq_full(Q) && e != NULL; @*/;
elem pq_delmin(pq Q) /*@requires !pq_empty(Q); @*/;
elem pq_min(pq Q) /*@requires !pq_empty(Q); @*/;
```

Complete the function `total_value` on the next page, which returns the sum of the values of the top `n` reports from the data stream `S` (by "top" reports, we mean those with the largest monetary value). Oh, and don't forget to implement the client interface for the priority queue!

Solution:

```
typedef ___________________ elem;
int elem_priority(elem e) {

  return ____________________________________;
}

int total_value(stream S, int n) {

  pq Q  = pq_new(_____________________);
  while (!stream_empty(S)) {
    // Put the next stock report into the priority queue


    ____________________________________________________
    // If the priority queue is at capacity, delete the
    // report with the smallest value

    if (_______________________________________________)


      ________________________________________________
  }

  // Add up the values of all reports in the priority queue
  int total = 0;

  while (_____________________________________________) {

    total += ____________________________________;
  }
  return total;
}
```

(2) (b) When does a priority queue behave like a stack? (*Hint*: think about how priorities must be assigned to elements that are inserted into the priority queue.)

Solution: