15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 8

Due: Thursday, October 24, 2013, at 10pm

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework will give you some practice working with hash tables, priority queues and heaps. 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	5	
2	6	
3	9	
Total:	20	

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. **Hash Tables: Data Structure Invariants**

   Refer to the C0 code below for `is_ht` that checks that a given hash table `ht` is a valid hash table.

```
struct chain_node {
  elem data;
  struct chain_node* chain;
};
typedef struct chain_node chain;

struct ht_header {
  chain*[] table;
  int m;     // m = capacity = maximum number of chains table can hold
  int n;     // n = size = number of elements stored in hash table
};
typedef struct ht_header* ht;

bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->m > 0)) return false;
  if (!(H->n >= 0)) return false;
  //@assert H->m == \length(H->table);
  return true;
}
```

   An obvious data structure invariant of our hash table is that every element of a chain hashes to the index of that chain. This specification function is incomplete, then: we never test that the contents of the hash table hold to this data structure invariant. That is, we test only on the struct `ht`, and not the properties of the array within.

   You may assume the existence of the following client functions as discussed in class:

```
int hash(key k);

bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;
  ;
```

(4) (a) Extend `is_ht` from above, adding code to check that every element in the hash table matches the chain it is located in, and that each chain is non-cyclic.

Solution:

```
bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->m > 0)) return false;
  if (!(H->n >= 0)) return false;
  //@assert H->m == \length(H->table);

  int nodecount = 0;

  for (int i = 0; i < ____________________________; i++)
  {
    // set p equal to a pointer to first node
    // of chain i in table, if any

    chain* p = ____________________________________________________;

    while (_____________________________________________________)
    {
      elem e = p->data;

      if ((e == NULL) || (_________________________________ != i))

        return false;

      nodecount++;

      if (nodecount > ____________________________________)

        return false;

      p = _____________________________________________________;

    }
  }

  if (_____________________________________________________________)

    return false;

  return true;
}
```

(1) (b) Consider the `ht_lookup` function given below:

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
  int i = abs(hash(k) % H->m);
  chain* p = H->table[i];
  while (p != NULL)
  //@loop_invariant is_chain(p, i, H->m);
  {
    //@assert p->data != NULL;
    if (key_equal(elem_key(p->data), k))
      return p->data;
    else
      p = p->next;
  }
  /* not in chain */
  return NULL;
}
```

Give a simple postcondition for this function.

Solution:

```
/*@ensures \result == ________________________________________

           || key_equal(k, ____________________________________);
@*/
```

2. **Priority Queues as an Abstract Data Type**

(2) (a) When does a priority queue behave like a FIFO queue? (HINT: Think about how priorities must be assigned to elements that are inserted into the priority queue.)

Solution:

(4) (b) Recall the client and library interfaces for a priority queue ADT (Abstract Data Type):

```
//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 P);
bool pq_empty(pq P);
void pq_insert(pq P, elem e)
//@requires !pq_full(P) && e != NULL;
   ;
elem pq_delmin(pq P)
//@requires !pq_empty(P);
   ;
elem pq_min(pq P)
//@requires !pq_empty(P);
   ;
```

Suppose our client needs to process a very long stream `S` of elements representing stock market reports:

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

The stream `S` is represented by the data type `stream` with the following two functions:

```
elem get_report(stream S);
// Returns the next stock report in the data stream
bool stream_empty(stream S);
// Returns true if the data stream has no more stock reports;
```

Since the stream is very, very long and we don't know how large it will eventually be, we can't just store all of the stock reports in a very large array.

Write a client function `total_value` that returns the total value of the 1000 stock reports with the highest stock values. You may assume that the stream has much more than 1000 stock reports. Your solution must use a priority queue. Remember

that you do not know how the priority queue is implemented since you only have its interface. Be sure to include a C0 definition for a suitable `elem_priority` function that the library can use for this problem.

Solution:

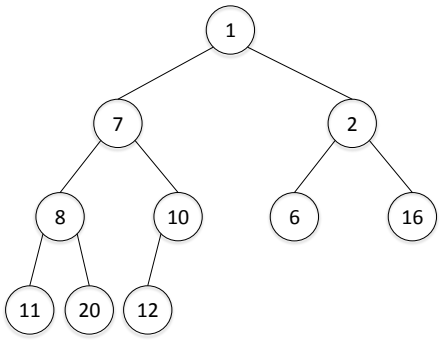
```
int elem_priority(elem e) {

    return ____________________________________;
}

int total_value(stream S) {

    pq P  = pq_new(____________________);

    while (!stream_empty(S)) {

        // Put the next stock report into the priority queue:

        ____________________________________________________

        // If the priority queue has more than 1000 reports,
        // delete the report with the smallest value.

        if (_______________________________________________)

             ____________________________________________________

    }

    // Add up the values of all 1000 reports in the priority queue
    int total = 0;

    while (_____________________________________________) {

           total += ____________________________________;

    }

    return total;

}
```

3. **Heaps**

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

> *Order*: For each non-root element, its value is greater than or equal to the value of 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 in the last level as possible.

Consider:



(1) (a) Draw a picture of the final state of the min-heap after an element with value 9 is inserted. Be sure that your final result satisfies 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 your final result satisfies 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 pictures.

`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:

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 ---------------------------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------------------------
```

(1) (e) In a non-empty min-heap (thought of as a tree which may not necessarily be stored as an array), where must the maximum value be? You should be able to give a precise answer in one clear sentence.

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:

(2) (g) We are given an array A of n integers that we wish to sort and we decide to use a min-heap to help with the sorting. Our min-heap will be represented using an array as described in class.

For each integer in the array, one at a time, we insert the integer into the min-heap. Then we delete (the minimum) from the min-heap repeatedly, storing each deleted value back into the array, one at a time, starting from the beginning of the array.

What is the worst-case runtime complexity of this sorting algorithm using big O notation? Explain your answer concisely.

Solution: $O($ $)$