15-122 : Principles of Imperative Computation

Fall 2012

Assignment 5

(Theory Part)

Due: Tuesday, October 30, 2012 at the **beginning** of lecture

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this weeks 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	4	
2	7	
3	8	
4	6	
Total:	25	

You *must* include this cover sheet.
Either type up the assignment using
15-122-theory5.tex, or print out this PDF.

1. **Hash Tables using Separate Chaining.**

Refer to the C_0 code below for `is_ht` that checks that a given hash table `ht` is a valid hash table.

```
struct list_node {
     elem data;
     struct list_node* next;
};
typedef struct list_node list;

struct ht_header {
     list*[] 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.

(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.

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

```
int hash(key k, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;

bool key_equal(key k1, key k2);

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

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 numnodes = 0;

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

    list* p = ___________________;

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

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

        return false;

      numnodes++;

      if (numnodes > _____________________________)

        return false;

      p = _________________________________;

    }
  }

  if (_____________________________________________________)

    return false;

  return true;
}
```

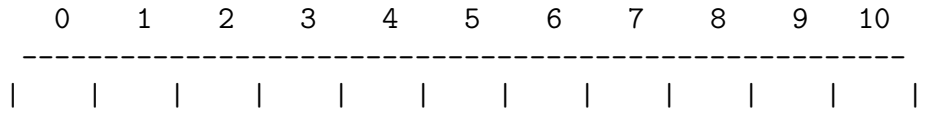
2. **Collision Resolution.**

Consider an alternate implementation of hash tables that uses an array only (no chains) and linear probing to resolve collisions. Recall that in linear probing, if a collision occurs, we search through the array linearly (with wraparound if necessary) for the first available position.

(2) (a) Assume that we hash a set of integer keys into a hash table of capacity m = 11 using a hash function hash(k) = k modulo 11 using linear probing to resolve collisions. Show where the following sequence of keys are stored in the hash table if they are inserted in the order shown using linear probing to resolve collisions.

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

Solution:



(2) (b) Suppose we have a hash table that resolves collisions using linear probing. Each table cell either holds a pointer to an element or NULL if the table cell is not occupied. Besides the insert and lookup functions, we provide a delete function: if the element is referenced in the table, we replace the pointer to the element to be deleted with NULL.

Explain why this implementation fails. Hint: Consider what happens after a number of elements are inserted that all collide, and then one of them is removed from the hash table.

Solution:

(3) (c) Consider an update to the hash table that adds a boolean array to the hash table data structure such that cell i is true if the corresponding table cell is or has been occupied, and false otherwise. When a new hash table is created, all cells in the `occupied` array are initially `false`.

```
struct ht_header {
    elem[] table;
    bool[] occupied;
    int m;          // m = capacity
};
typedef struct ht_header* ht;
```

Complete the functions below for `ht_insert`, `ht_lookup` and `ht_delete` for the hash table that uses linear probing to resolve collisions and the `occupied` array as described above. You may assume there is an appropriate `is_ht` function already defined, and you may assume the existence of the client functions `hash`, `elem_key`, and `key_equal` as described in problem 1.

```
void ht_insert(ht H, elem e)
//@requires is_ht(H);
//@requires e != NULL;
//@ensures is_ht(H);
{
    key k = elem_key(e);
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        if (____)   // write missing condition below in solution box
           H->table[i] = e;
           H->occupied[i] = true;
           return;
        }
        i = (i + 1) % m;
        j++;
    }
    return;     // table full, element gets dropped
}
```

Solution:

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
//@ensures \result == NULL || key_equal(elem_key(\result), k);
{
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        elem e = H->table[i];
        if (____)   // write missing condition below in solution box
            return e;
        i = (i + 1) % m;
        j++;
    }
    return NULL;
}
```

Solution:

```
void ht_delete(ht H, key k)
//@requires is_ht(H);
//@ensures is_ht(H);
{
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        elem e = H->table[i];
        if (____)    // write missing condition #1 below in solution box
           return;
        if (____) {  // write missing condition #2 below in solution box
           H->table[i] = NULL;
           return;
        }
        i = (i + 1) % m;
        j++;
    }
}
```

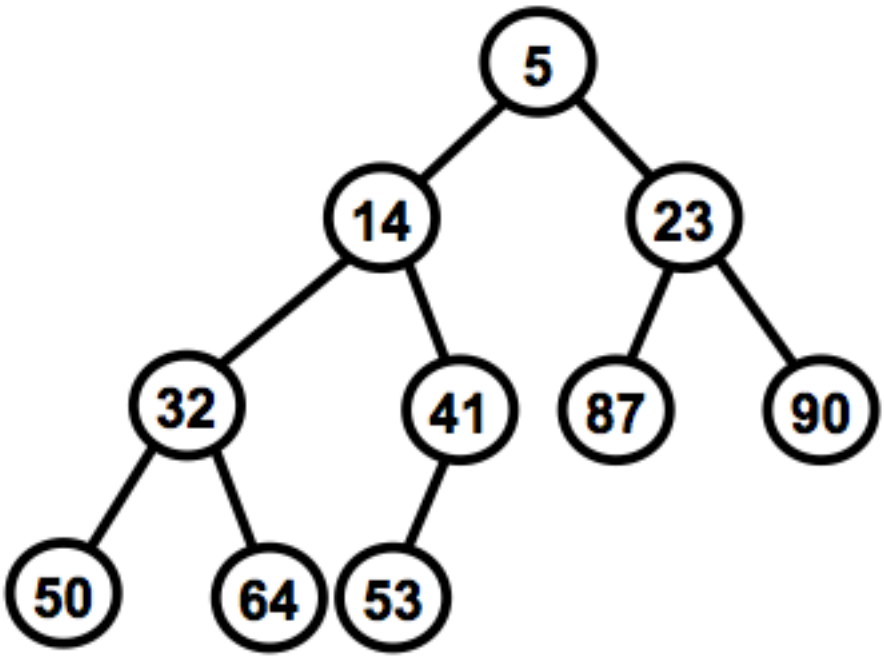
Solution:

condition 1:

condition 2:

3. **Heaps.**

We represent heaps, conceptually, as trees. For example, take the min-heap below.



(1) (a) What is the result of inserting an element with value 29 into the min-heap shown above using the standard algorithm for heap insertion as discussed in class? Draw your answer as a tree, of the same form.

Solution:

(1) (b) Using the *original* min-heap from the previous page, what is the result of deleting the minimum element from the min-heap using the standard O(log n) algorithm for heap deletion as discussed in class? Draw your answer as a tree, of the same form.

Solution:

(2) (c) Assume a heap is stored in an array as discussed in class where the root is stored at index 1. Using the *original* min-heap on the previous page, at what index is the element with value 32 stored? At what index is its parent stored? At what indices are its left and right children stored?

Solution:

```
The value 32 is stored at index ___________.

The parent of value 32 is stored at index ____________.

The left child of value 32 is stored at index ____________.

The right child of value 32 is stored at index ____________.
```

(1) (d) A heap is a binary tree. The root is at level 1, the children of the root are at level 2, etc. For a heap with 10 levels, what is the maximum number of elements it can store?

Solution:

(1) (e) Suppose we have a non-empty min-heap of integers of size n and we wish to find the maximum integer in the heap. Describe precisely where the maximum must be in the min-heap. (You should be able to answer this question with one short sentence.)

Solution:

(1) (f) Using the following C0 definition for a heap of integers (position 0 in the array is not used):

```
struct heap_header {
  int limit; // size of the array of integers
  int next;  // next available array position for an integer
  int[] value;
};
typedef struct heap_header* heap;
```

Write the C0 function `find_max` that takes a non-empty min-heap and returns the maximum value. Your code should examine only those cells that could possibly hold the maximum.

Solution:

```
int find_max(heap H)
//@requires is_heap(H);
//@requires H->next > 1;
{









}
```

(1) (g) What is the worst-case runtime complexity in big O notation of your `find_max` function on a non-empty min-heap of n elements from the previous problem?

Solution:

4. **Priority Queues.**

In a priority queue, each element has a priority value which is represented as an integer. Recall that the lower the integer is, the higher the priority is. When we perform a `delmin` operation, we remove the element with the highest priority (i.e. lowest `int` value for priority).

(1) (a) If we used an unsorted array to represent the priority queue, what is the worst-case runtime complexity using big O notation for `insert` and `delmin` operations on a priority queue with n elements?

Solution:
insert:
delmin:

(1) (b) If we used a sorted array to represent the priority queue, where the elements are stored from lowest priority to highest priorty, what is the worst-case runtime complexity using big O notation for `insert` and `delmin` operations on a priority queue with n elements?

Solution:
insert:
delmin:

(1) (c) If we used a heap to represent the priority queue, what is the worst-case runtime complexity using big O notation for `insert` and `delmin` operations on a priority queue with n elements?

Solution:
insert:
delmin:

(1) (d) Which implementation above is preferable if the number of `insert` operations and `delmin` operations are relatively balanced? Explain in one sentence.

Solution:

(1) (e) Under what specific condition does a priority queue behave like a FIFO queue if it is implemented using a heap?

Solution:

(1) (f) Under what specific condition does a priority queue behave like a LIFO stack if it is implemented using a heap?

Solution: