15-122 : Principles of Imperative Computation, Spring 2013

Homework 6 Theory [UPDATE 1]

Due: Thursday, April 4, 2013, at the **beginning** of lecture

Name: ______________________

Andrew ID: ______________________

Recitation: ______________________

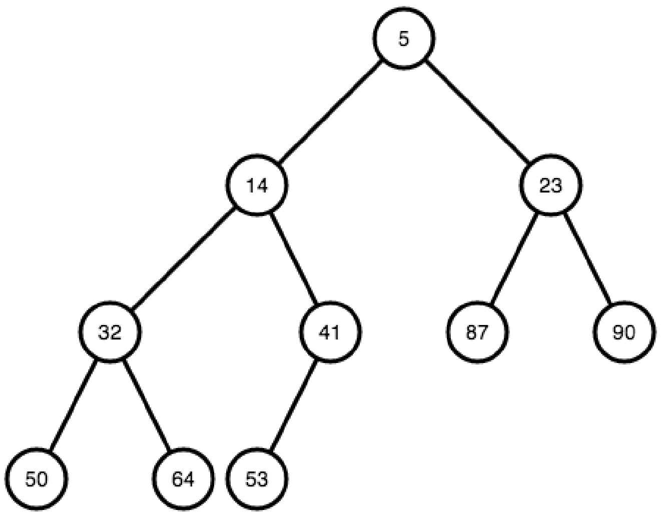
The written portion of this week's homework will give you some practice working with heaps, priority queues, BSTs, and AVL trees, as well as begin our transition to full C. You can either type up your solutions or write them *neatly* by hand in the spaces provided. You should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	5	
2	6	
3	6	
4	8	
5	5	
Total:	30	

You *must* use this printout, include this cover sheet, and staple the whole thing together before turning it in. Either type up the assignment using 15122-theory6.tex, or print this PDF and write your answers *neatly* by hand.

1. **Heaps.**

We represent heaps, conceptually, as trees. For example, consider the min-heap below.[1]



(2) (a) Assume a heap is stored in an array as discussed in class where the root is stored at index 1. Using the above min-heap, 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) (b) 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]Diagram courtesy of Hamilton (http://hamilton.herokuapp.com)

(1) (c) 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 a 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) (d) 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:

2. **Heaps and BSTs.**

Though heaps and binary search trees (BSTs) are very different in terms of their invariants and uses, they are both conceptually represented as trees. This question asks about three invariants of trees: the BST ordering invariant, the heap shape invariant, and the heap ordering invariant (for min-heaps, where higher-priority keys are lower integer values). For the first part of this question, we assume that each element has a single C0 `int` that is used as both the BST key and the heap priority.

(1) (a) Draw a tree with five elements that is a BST and satisfies the heap shape invariant.

Solution:

(1) (b) Draw a tree with at least four elements that is a BST and satisfies the (min-)heap ordering invariant.

Solution:

(1) (c) Why is it not a good idea to have a data structure that enforces both the (min-)heap ordering invariant and the BST ordering invariant? (Be brief!)

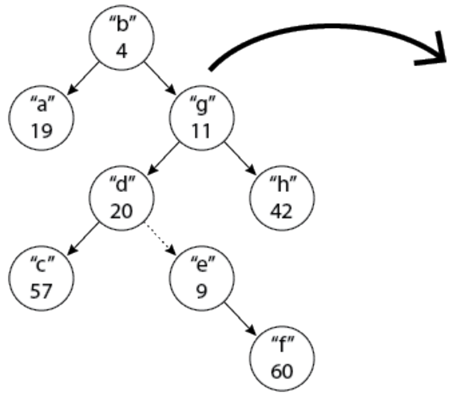
Solution:

(3) (d) Maintaining the BST ordering invariant and the heap invariant on the *same* set of values may not be a good idea, but it can be useful to have a tree structure where each node has two separate values – a key used for the BST ordering invariant and a priority used for the heap ordering invariant. Such trees are called *treaps*; we will use `string`s as keys and `int`s as priorities in this question.

The treap below satisfies the BST ordering invariant, but violates the heap ordering invariant because of the relationship between the "e"/9 node and its parent. In a heap, we restore the heap shape invariant using swaps. But in a treap, such a swap would violate the BST ordering invariant. However, by using the same local rotations we learned about for AVL trees, it is possible to restore the heap ordering invariant while preserving the BST ordering invariant.

The heap ordering invariant for the tree below can be restored with two tree rotations. Draw the tree that results from each rotation. You should be drawing two trees.

Solution:



3. **Priority Queues.**

In a priority queue, each element has a priority value which is represented as an integer. As in the previous question, the lower the integer, the higher the priority. When we call `pq_delmin`, we remove the element with the highest priority.

(a) Consider the following ways that we can implement a priority queue. Using big-O notation, what is the worst-case runtime complexity for each implementation to perform `pq_insert` and `pq_delmin` on a priority queue with n elements?

(1) i. Using an *unsorted* array.

Solution:

`pq_insert`:

`pq_delmin`:

(1) ii. Using a *sorted* array, where the elements are stored from lowest to highest priority.

Solution:

`pq_insert`:

`pq_delmin`:

(1) iii. Using a heap.

Solution:

`pq_insert`:

`pq_delmin`:

(1) (b) Which implementation in (a) is preferable if the number of `pq_insert` and `pq_delmin` operations are relatively balanced? Explain in one sentence.

Solution:

(1) (c) Under what specific condition does a priority queue behave like a FIFO queue if it is implemented using a heap? *(Warning: if you words like "higher," "lower," "increasing," or "decreasing" in your answer, be clear whether you are talking about priority or integer value.)*

Solution:

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

Solution:

4. **AVL Trees.**

(4) (a) Draw the AVL trees that result after successively inserting the following keys into an initially empty tree, in the order shown:

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

Show the tree after each insertion and subsequent re-balancing (if any): the tree after the first element, `98`, is inserted into an empty tree, then the tree after `88` is inserted into the first tree, and so on for a total of seven trees. Make it clear what order the trees are in.

Be sure to maintain and restore the BST invariants and the additional balance invariant required for an AVL tree after each insert.

Solution:

(b) Recall our definition for the height h of a tree:

The height of a tree is the maximum length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, and a balanced tree with three nodes has height 2.

The minimum number of nodes n in a valid AVL tree is related to its height. The goal of this question is to quantify this relationship.

(2) i. Fill in the table below relating the variables h and n:

h	n
0	0
1	1
2	2
3	
4	
5	

(2) ii. Recall that the xth Fibonacci number $F(x)$ is defined by:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(x) &= F(x-1) + F(x-2), \qquad x > 1 \end{aligned}$$

Using the table in part (i), give an expression for $T(h)$, where $T(h) = n$. You may find it useful to use $F(n)$ in your answer, but your answer <u>does not</u> need to be a closed form expression.

Solution:

5. **Pass by reference**

We now begin our transition in 15-122 to full C!

At various points in our C0 programming experience we had to use somewhat awkward workarounds to deal with *functions that need to return more than one value*. The address-of operator (&) in C gives us a new way of dealing with this issue.

(2) (a) Sometimes, a function needs to be able to both 1) signal whether it can return a result, and 2) return that result if it is able to. One such function that we've seen is `peg_solve`. When a solution is found, `peg_solve` returns `1` and modifies the originally-empty stack passed in with the winning moves, and when no solution is found `peg_solve` simply returns the minimum number of pegs seen. Parsing also fits this pattern. Consider the following code:

```
bool my_int_parser(char *s, int *i);  // Returns true iff parse succeeds

void parseit(char *s) {
  REQUIRES(s != NULL);
  int *i = xmalloc(sizeof(int));
  if (my_int_parser(s, i))
    printf("Success: %d.\n", *i);
  else
    printf("Failure.\n");
  free(i);
  return;
}
```

Using the address-of operator, rewrite the body of the `parseit` function so that it does not heap-allocate, free, or leak any memory on the heap. You may assume `my_int_parser` has been implemented (its prototype is given above).

Solution:

```
void parseit(char *s) {
  REQUIRES(s != NULL);









  return;
}
```

(3) (b) In both C and C0, multiple values can be 'returned' by bundling them in a struct:

```
struct bundle { int x; int y; };
struct bundle *foo(int x) {
  ...
  struct bundle *B = xmalloc(sizeof(struct bundle));
  B->x = e1;
  B->y = e2;
  return B;
}
int main() {
  ...
  struct bundle *B = foo(e);
  int x = B->x;
  int y = B->y;
  free(B);
  ...
}
```

Rewrite the declaration and the last few lines of the function `foo`, as well as the snippet of `main`, to avoid heap-allocating, freeing, or leaking any memory on the heap. The rest of the code (...) should continue to behave exactly as it did before.

Solution:

```
_______________ foo(___________________________________________) {
  ...

  ________________________________________________________________

  ________________________________________________________________

  ________________________________________________________________
}

int main() {
  ...

  ________________________________________________________________

  ________________________________________________________________

  ________________________________________________________________

  ________________________________________________________________
  ...
}
```