

15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 10

Due: Monday November 9, 2015 by 6:00 PM

Name: _____

Andrew ID: _____

Section: _____

This written homework covers heaps and priority queues.

Print out this PDF double-sided, **staple** pages in order, and write your answers on these pages *neatly*. You can hand in the assignment to your TA during lab or in the box outside of GHC 4117 (in the CS Undergraduate Program suite). **Warning: The box is removed promptly at 6PM.**

You must hand in your homework yourself;
do not give it to someone else to hand in.

Question	Points	Score
1	6	
2	4	
3	5	
Total:	15	

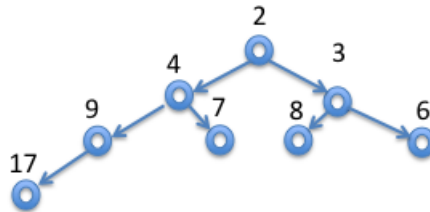
1. Heaps

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:



1pt

- (a) Draw a picture of the final state of the min-heap after an element with value 5 is inserted. Satisfy the shape invariant first, then restore the order invariant while maintaining the shape invariant.

Solution:

1pt

- (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. Satisfy the shape invariant first, then restore the order invariant while maintaining the shape invariant.

Solution:

2pts

- (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.

24, 16, 49, 20, 3, 21, 54, 12

Solution:

1pt

- (d) In a min-heap with n nodes, $n > 0$, how many nodes are leaves? Write one mathematical expression (**not** a C0 expression; you may use $\lfloor x \rfloor$ to round x down or $\lceil x \rceil$ to round x up) that expresses the number of leaves regardless of whether n is even or odd.

Solution:

1pt

- (e) 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.

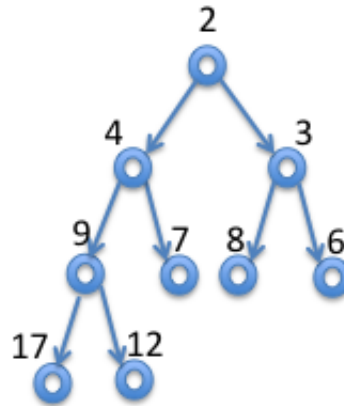
What is the worst-case runtime complexity of this sorting algorithm, using Big-O notation? Briefly explain your answer.

Solution: $O(\quad)$

2. Array Implementation of Heaps

1pt

- (a) Assume a heap is stored in an array as discussed in class. Using the min-heap pictured below, 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

1pt

- (b) Suppose we have a nonempty priority queue of n elements represented using the array implementation of heaps. Give the exact range (inclusive), in terms of n , of array indexes where any element of lowest priority might occur. You may use mathematical notation or C0 notation.

Solution:

2pts

(c) Here is the `heap_add` function discussed in class:

```
void heap_add(heap* H, elem e)
//@requires is_heap(H) && !heap_full(H);
//@ensures is_heap(H);
{
    int i = H->next;
    H->data[H->next] = e;
    (H->next)++;
    /*** LOCATION 1 ***/
    while(i > 1)
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant is_heap_except_up(H, i);
    //@loop_invariant grandparent_check(H, i);
    {
        if (ok_above(H, i/2, i)) {
            return;
        }
        swap_up(H, i);
        i = i/2;
    }
}
```

Write “OK” to the right of each assertion below, if it provably always holds at LOCATION 1; write “NO” otherwise.

```
//@assert is_safe_heap(H);

//@assert is_heap(H);

//@assert grandparent_check(H, i);

//@assert is_heap_except_up(H, i/2);

//@assert is_heap_except_down(H, i);

//@assert ok_above(H, i, i);
```

3. Using Priority Queues

You are working an exciting desk job as a stock market analyst. You want to be able to determine the total price increase of the stocks that have seen the highest price increases over the last day (of course, on a bad day, these might simply be the least negative price changes). 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_t` data type with the following interface:

```
// typedef _____ stream_t;
typedef struct stock_report report;
struct stock_report {
    string company;
    int current_price; // stock price in cents
    int old_price;     // previous day's price in cents
};

// Returns true if the data stream is empty
bool stream_empty(stream_t S);
// Retrieve the next stock report from the data stream
report* get_report(stream_t S) /*@requires !stream_empty(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 Grace just finished a stellar priority queue implementation with the interface below. You think you should be able to use Grace's priority queue to keep track of only the stock reports on the stocks that have increased the most, discarding the others as necessary.

```
// Client Interface
// f(x,y) returns true if x is STRICTLY higher priority than y
typedef bool higher_priority_fn(void* x, void* y);

// Library Interface
typedef _____* pq_t;
pq_t pq_new(int capacity, higher_priority_fn* priority)
    /*@requires capacity > 0 && priority != NULL; @*/
    /*@ensures \result != NULL; @*/ ;
bool pq_full(pq_t Q) /*@requires Q != NULL; @*/ ;
bool pq_empty(pq_t Q) /*@requires Q != NULL; @*/ ;
void pq_add(pq_t Q, void* x) /*@requires Q != NULL && !pq_full(Q); @*/
    /*@requires x != NULL; @*/ ;
void* pq_rem(pq_t Q) /*@requires Q != NULL && !pq_empty(Q); @*/ ;
void* pq_peek(pq_t Q) /*@requires Q != NULL && !pq_empty(Q); @*/ ;
```

3pts

- (a) Complete the functions `client_priority` and `total_increase` below. The function `total_increase` returns the sum of the price increases of the `n` stocks with the highest price increases from the data stream `S`.

```

use <util>

bool client_priority(void* x, void* y)
//@requires x != NULL && \hastag(report*, x);
//@requires y != NULL && \hastag(report*, y);
{
    return _____;
}

int total_increase(stream_t S, int n)
//@requires 0 < n && n < int_max();
{
    pq_t 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 price increase

        if (_____)
            _____
    }

    // Add up the price increases of everything in the priority queue
    int total = 0;

    while (_____) {
        report* r = _____;
        total += _____;
    }

    return total;
}

```


1pt

- (b) Assuming that Grace's priority queues are based on the heap data structure, what is the running time of `total_increase(S, n)` if the stream S ultimately contains m elements? (Give an answer in big- O notation.)

1pt

- (c) Suppose a sequence of n elements are inserted into a priority queue so that the priority of each element inserted is strictly decreasing. Afterwards, the elements are removed one at a time based on priority. What common data structure does this priority queue implement?

If the priorities are strictly increasing instead, what common data structure does this priority queue implement?