15-122: Principles of Imperative Computation, Fall 2017

Written Homework 10

Due: Monday 6th November, 2017 by 9pm

Name: Shaojie Zhang

Andrew ID: shaojiez

Section: O

This written homework covers heaps and priority queues.

Instructions

You can prepare your submission in one of two ways:

Just edit (preferred) Use any PDF editor (e.g., Preview on Mac, iAnnotate on mobile, Acrobat Pro installed on all non-CS cluster machines and most platforms) to typeset your answers in the given spaces — you can even draw pictures. *That's it.*

Print and Scan Alternatively, print this file, write your answers *neatly* by hand, and then scan it into a PDF file. *This is pretty labor-intensive.*

Once you have prepared your submission, submit it on Gradescope. You have unlimited submissions.

Question:	1	2	3	Total
Points:	4	2.5	2.5	9
Score:				

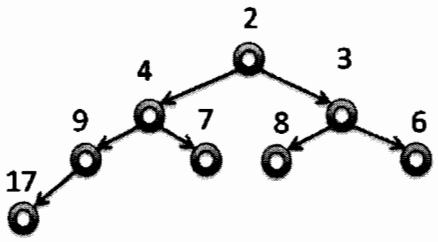
Evaluation Summary Once this homework is graded, you will be able to find a summary of your performance on Gradescope.

1. **Heaps**

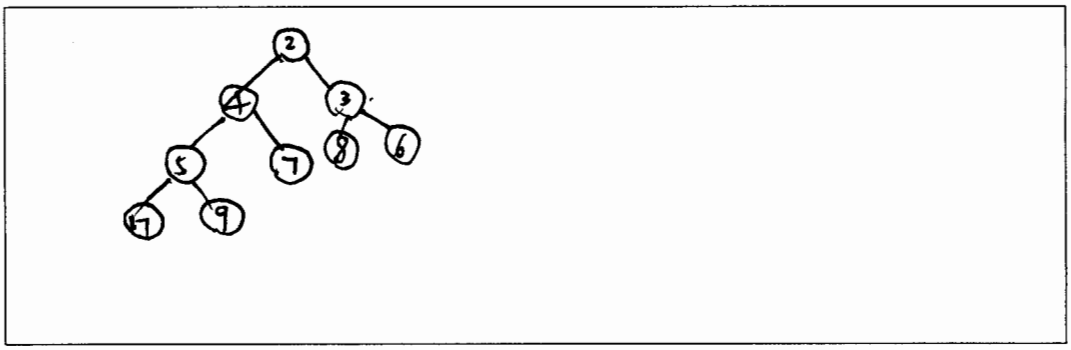
As discussed in class, a *min-heap* is a hierarchical data structure that satisfies two invariants:

> **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).

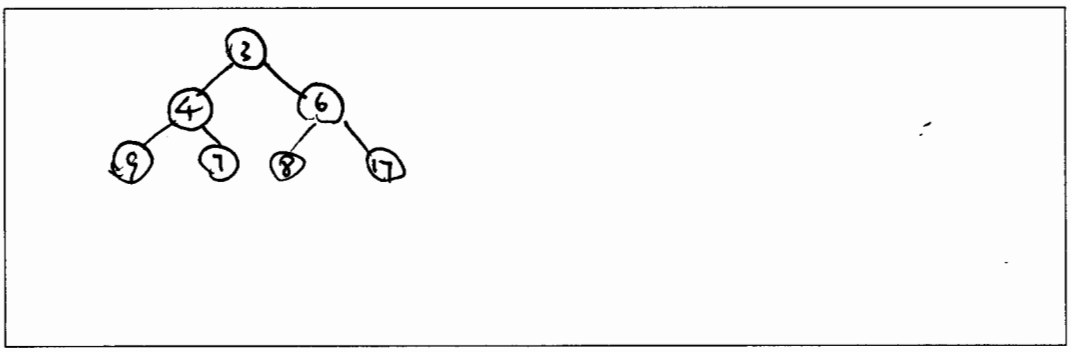
Consider:



0.5pts **1.1** 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. Draw all branches in your tree *clearly* so we can distinguish left branches from right branches.

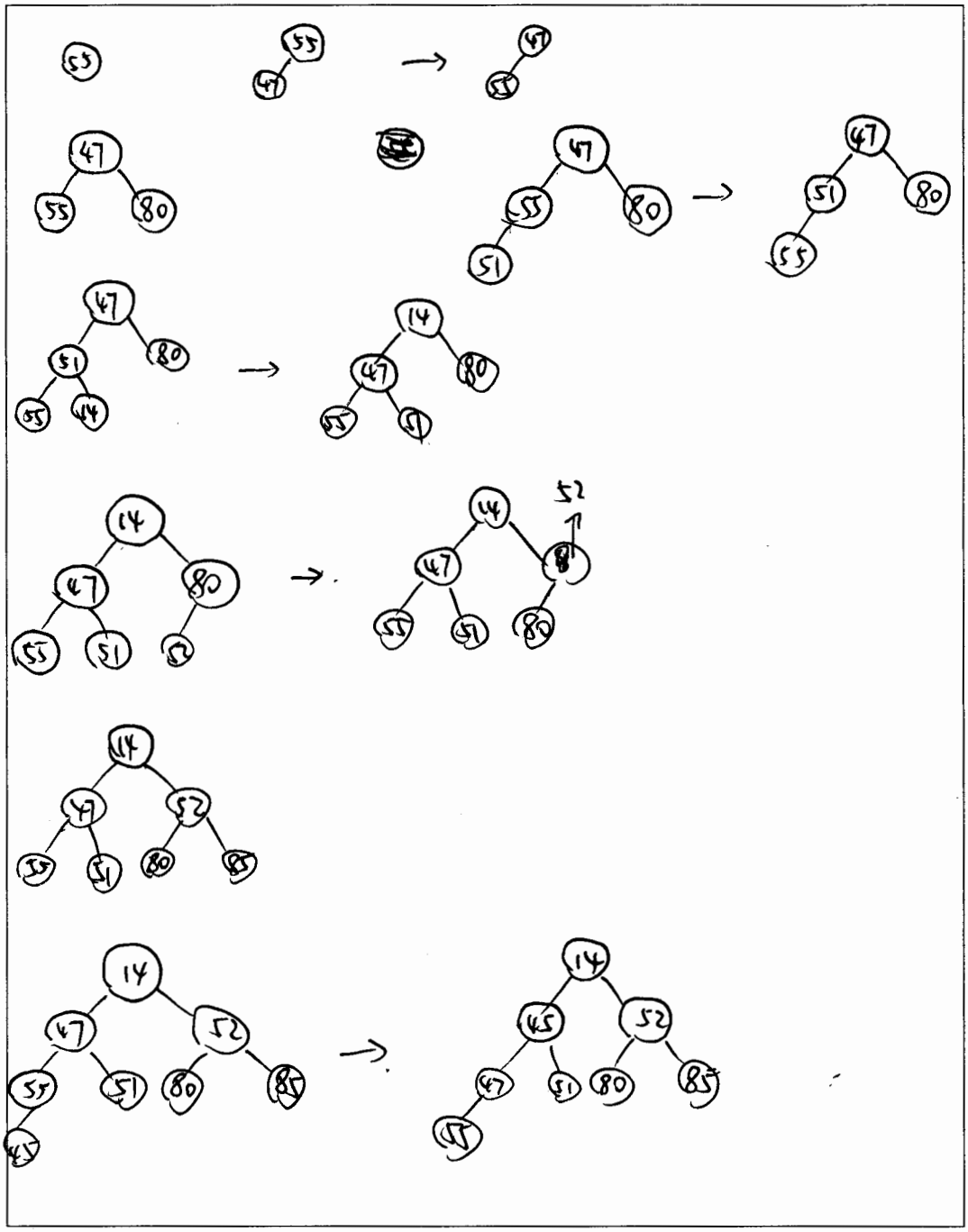


0.5pts **1.2** Starting from the *original* min-heap above, draw a picture of the final state of the min-heap after the element with the minimum value is deleted. Satisfy the shape invariant first, then restore the order invariant while maintaining the shape invariant.



1.5pts **1.3** 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.

```
55, 47, 80, 51, 14, 52, 85, 45
```



0.5pts **1.4** 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.

$O(\ n \log n.\ )$

Because: Insert take $O(\log n)$. $\log 1 + \log 2 + \ldots + \log n. = n \log n$

Delete takes, similarly, $O(\log n)$.

0.5pts **1.5** 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.

The max value must be on the last level.

0.5pts **1.6** What is the worst-case runtime complexity of finding the maximum in a min-heap if the min-heap has n elements? Why?

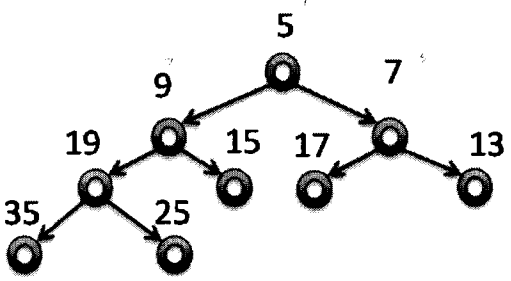
$O(\ n.\ )$

Because the number of values that need to be examined is $\frac{n+1}{2}$.

2. Implementing Priority Queues as Heaps

0.5pts

2.1 Assume a heap is stored in an array as discussed in class. Using the min-heap pictured below, show where each element is stored in the array. You may not need to use all of the array positions shown below.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	5	9	7	19	15	17	13	35	25						

1pt **2.2** Here is the pq_add function discussed in class:

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_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);                    // Answer: OK
//@assert is_heap(H);                         // Answer: No
//@assert grandparent_check(H, i);            // Answer: OK
//@assert is_heap_except_up(H, i/2);          // Answer: No
//@assert is_heap_except_down(H, i);          // Answer: No
//@assert ok_above(H, i, i);                  // Answer: OK
```

1pt **2.3** The library function `heap_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* heap_build(elem[] E, int n, higher_priority_fn* priority)
//@requires 0 < n && n <= \length(E) && priority != NULL;
//@ensures is_heap(\result);
{
  heap* H = alloc(heap);
  H->limit = n;
  H->next = 1;
  H->data = E;
  H->prior = priority;
  for (int i = 1; i < n; i++) {
    pq_add(H, E[i]);
  }
  return H;
}
```

This code disrespects the boundary between the client and the library. (Do you see why?) Use the functions `int_to_elem` and `int_compare` to create a test case where the postcondition of `build_broken_heap` will always succeed. Assume that the function `int_compare` treats lower integers as higher priority, so the heap we're building will be a min-heap.

```
elem int_to_elem(int n)
  /*@ensures \result != NULL && \hastag(int*, \result); @*/
  /*@ensures n == *(int*)\result; @*/ ;
bool int_compare(elem x, elem y)
  /*@requires x != NULL && \hastag(int*, x); @*/
  /*@requires y != NULL && \hastag(int*, y); @*/ ;

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

  E[1] = int_to_elem(4);

  E[2] = int_to_elem(3);

  return H;
}
```

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); @*/ ;
```

2pts **3.1** 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 (((report*)x)->current_price - ((report*)y)->old_price) >
         (((report*)y)->current_price - ((report*)y)->old_price);
}

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

  pq_t Q  = pq_new(n, &client_priority);

  while (!stream_empty(S)) {
    // Put the next stock report into the priority queue

    pq_add(Q, (void*)get_report(S));
    // If the priority queue is at capacity, delete the
    // report with the smallest price increase

    if (pq_full(Q))

      pq_rem(Q);
  }

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

  while (!empty(Q)) {

    report* r = (report*)pq_rem(Q);

    total += r->current_price - r->old_price;
  }

  return total;
}
```

0.5pts **3.2** 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.)

$\max(O(m \log n), o(n \log n))$