15-122 : Principles of Imperative Computation, Spring 2016

Written Homework A/B

Due: Tuesday 5th April, 2016

Name: _____

Andrew ID: _____

Section: _____

This written homework covers heaps and priority queues.

The assignment is due by 1:30pm on Tuesday 5th April, 2016.

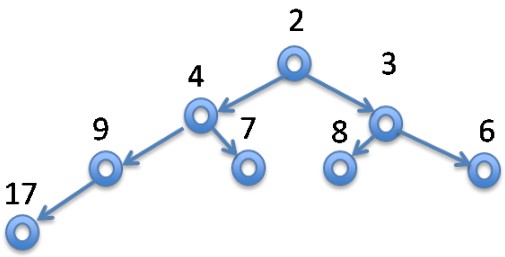
This assignment is to be completed by editing the PDF file,
printing it to another PDF file,
and then submitting through Gradescope.

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:



1pt (a) Describe 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.

In the next few questions, you can describe a tree by recursively putting parentheses around subtrees and writing _ for empty subtrees, so that for example

```
      5
     / \
    2   7
   / \   \
  1   4   9
     /
    3
```

is described as

```
((_ 1 _) 2 ((_ 3 _) 4 _)) 5 (_ 7 (_ 9 _)).
```

If you have access to a PDF editing application that allows you to draw, feel free to use that instead. In that case, be sure all branches in your tree are drawn *clearly* so we can distinguish left branches from right branches.

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.

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

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.

1pt (e) We are given an array A of n integers. Consider the following sorting algorithm:

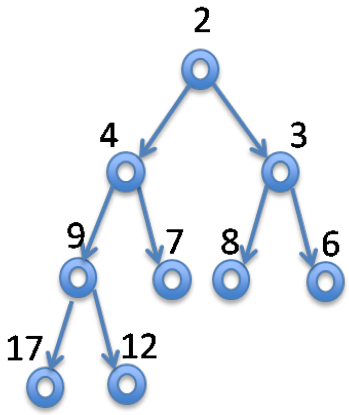
- Insert every integer from A into a min-heap.
- Repeatedly delete the minimum from the heap, storing the deleted values back into A from left to right.

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

Solution: $O($_____$)$

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.

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);              // Answer: _____

//@assert is_heap(H);                   // Answer: _____

//@assert grandparent_check(H, i);      // Answer: _____

//@assert is_heap_except_up(H, i/2);    // Answer: _____

//@assert is_heap_except_down(H, i);    // Answer: _____

//@assert ok_above(H, i, i);            // Answer: _____
```

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. Afterward, 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?

2pts

4. **Contracts in C**

The code below is taken from the lecture notes on hash sets in C0. This is also legal C code (assuming all the right definitions are available), but the contracts will not be checked in C.

```
1  elem hset_lookup(hset H, elem x)
2  //@requires is_hset(H);
3  //@requires x != NULL;
4  //@ensures \result==NULL || elem_equiv(\result, x);
5  {
6    int i = elemhash(H, x);
7    for (chain* p = H->table[i]; p != NULL; p = p->next) {
8      //@assert p->data != NULL;
9      if (elem_equal(p->data, x)) return p->data;
10   }
11   return NULL;
12 }
```

Rewrite the function in the box on the next page as follows:

- Insert assignment statements so that all return statements have the form `return result`. (In other words, use the variable `result`, defined on the next page, to hold the return value for all cases and use this variable in your postcondition.)
- Insert any necessary C contracts so that, when compiled with the flag `-DDEBUG`, contracts will be checked as they would be in C0 with the flag `-d`.

Do *not* simplify any contracts even if it is immediately obvious from the context that you could do so. You may omit the C0 contracts (lines beginning `//@`) even though in practice we might like to keep them.

```
elem hset_lookup(hset H, elem x) {









  elem result;

















}
```

3pts

5. **Allocating and freeing memory in C**

Here is a leaky C program that works with NULL-terminated linked lists. We've omitted the code for `print_list` because it can't leak any memory. Contracts have been omitted for the sake of space.

```
1  typedef struct list_node list;
2  struct list_node {
3    int data;
4    list* next;
5  };
6  void free_list(list* L) {
7    list* current = L;
8    while (current != NULL) {
9      list* next = current->next;
10     free(current);
11     current = next;
12   }
13   return;
14 }
15 void sum(list* L) {
16   list* sum = xmalloc(sizeof(list));
17   sum->data = 0;
18   list* current = L;
19   while (current != NULL) {
20     sum->data += current->data;
21     current = current->next;
22   }
23   L->data = sum->data;
24   L->next = NULL;
25   return;
26 }
27 int main() {
28   list* current = NULL;
29   for (int i=0 ; i<10 ; i++) {
30     ASSERT(0 <= i);
31     list* new = xmalloc(sizeof(list));
32     new->data = i;
33     new->next = current;
34     current = new;
35   }
36   printf("Initial list: "); print_list(current);
37   sum(current);
38   printf("Summed list: "); print_list(current);
39   return 0;
40 }
```

In the table below, give the line number of each line that leaks memory. A line is considered to leak memory if, as a result of executing it, some allocated memory has not been freed, and no further references to that memory are possible. Returning from the `main` function without deallocating everything that was allocated is considered a leak (even though the operating system will clean it up).

Indicate how to fix the leak(s) by writing any extra code that needs to be added, with the line numbers between which it should be inserted.

Line number of leak	Code that fixes it	Where to insert it