15-122 : Principles of Imperative Computation, Spring 2014

Written Homework 9

Due before class: Thursday, March 27, 2014

Name: 高棱霖

Andrew ID: U201514613

~~Recitation~~ class: ACM1501

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

Question	Points	Score
1	5	
2	6	
3	9	
Total:	20	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

1) Write your answers *neatly* on a printout of this PDF.

2) Use the TeX template at `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory9.tgz`

1. **Heaps**

   Refer to the implementation of heaps discussed in class that is available on our course website. Keep in mind that throughout this homework, when we say "priority value", we mean the integer representing the priority. So, higher priority value is really lower priority.

(2) (a) Add a meaningful assertion about H to each of the functions below.

Solution:

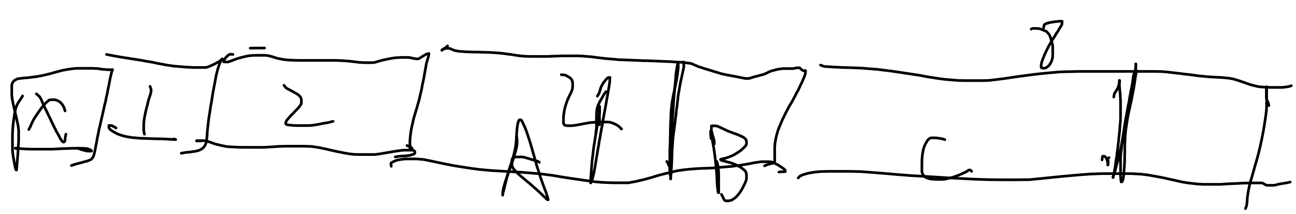
```
void pq_insert(heap H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert H->next <= H->limit; ;
  int i = H->next - 1;
  while (i > 1 && priority(H,i) < priority(H,i/2))
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant is_heap_except_up(H, i);
    {
      swap(H->data, i, i/2);
      i = i/2;
    }
  //@assert is_heap(H);
  return;
}

elem pq_delmin(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
  int n = H->next;
  elem min = H->data[1];
  H->data[1] = H->data[n-1];
  H->next = n-1;
  if (H->next > 1) {
      //@assert !pq_empty(H);
      sift_down(H, 1);
  }
  return min;
}
```

(2) (b) Complete an additional library function, `pq_max`, that returns, but does not remove, the element with the maximum priority value from our array-based min-heap. We have provided part of the function for you. You should examine only those elements that might contain the maximum. (Note that this is not an operation you would want to provide for a min-heap priority queue due to its runtime complexity.)

Solution:

```
elem pq_max(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
   int max = (H->next+1)/2;

   for (int i = max+1; i < H->next; i++)

       if (priority(H, i) > priority(H, max))  max = i;

   return max;
}
```

proof:



from 1 to 4A+2B ← next.

$C = 2A$

$Candinate = B + C = B + 2A$

$total = (A+B)\times 2 - 1 + C$

$= 4A + 2B - 1$

$\therefore Candinate = \frac{total+1}{2}$

(1) (c) The library function, `pq_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 in our heap structure and does not allocate a new array.

```
heap pq_build(elem[] elements, int arraylength)
//@requires \length(elements) > 0 && \length(elements)==arraylength;
//@ensures is_heap(\result);
{
  heap H = alloc(struct heap_header);
  H->limit = arraylength;
  H->next = 1;
  H->data = elements;
  for (int i = 1; i < arraylength; i++)
      pq_insert(H, elements[i]);
  return H;
}
```

The function above does not respect the boundary between the client and the library. Complete the following client code so that the `pq_empty` function will likely abort by failing its precondition. In your solution, do not deference `H` or set `H` to `NULL`.

Solution:

```
:
:
//@assert \length(E) = 16;
heap H = pq_build(E, 16);
  E[1] = 6;
----------------------------------------------------------------
  E[2] = 5;
----------------------------------------------------------------

----------------------------------------------------------------

if (pq_empty(H)) return;
:
:
```

2. **Priority Queues as an Abstract Data Type**

(2) (a) When does a priority queue behave like a stack? (HINT: Think about how priorities must be assigned to elements that are inserted into the priority queue.)

Solution:

when data inserted is non-decreasing.

(4) (b) Recall the client and library interfaces for a priority queue ADT (Abstract Data Type):

```
//Client Interface

typedef _________ elem;
int elem_priority(elem e)
//@requires e != NULL;
   ;

// Library Interface

typedef _________ pq;
pq pq_new(int capacity)
//@requires capacity > 0;
   ;
bool pq_full(pq P);
bool pq_empty(pq P);
void pq_insert(pq P, elem e)
//@requires !pq_full(P) && e != NULL;
   ;
elem pq_delmin(pq P)
//@requires !pq_empty(P);
   ;
elem pq_min(pq P)
//@requires !pq_empty(P);
   ;
```

Suppose our client needs to process a very long stream `S` of elements representing stock market reports:

```
struct stock_report {
    string company;
    int value;   // stock value in whole dollars
};
typedef struct stock_report* elem;
```

The stream `S` is represented by the data type `stream` with the following two functions:

```
elem get_report(stream S);
// Returns the next stock report in the data stream
bool stream_empty(stream S);
// Returns true if the data stream has no more stock reports;
```

Since the stream is very, very long and we don't know how large it will eventually be, we can't just store all of the stock reports in a very large array.

Write a client function `total_value` that returns the total value of the 1000 stock reports with the highest stock values. You may assume that the stream has much more than 1000 stock reports. Your solution must use a priority queue. Remember that you do not know how the priority queue is implemented since you only have its interface. Be sure to include a C0 definition for a suitable `elem_priority` function that the library can use for this problem.

Solution:

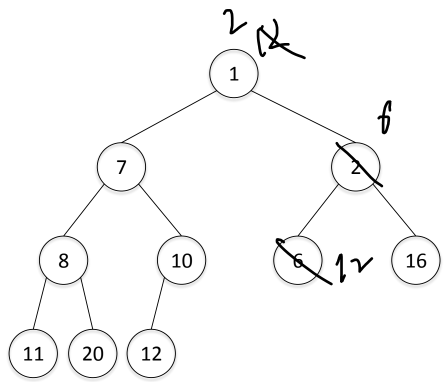
```
int elem_priority(elem e) {
    return ___e->value___;
}

int total_value(stream S) {
    pq P  = pq_new(___1001___);

    while (!stream_empty(S)) {

        // Put the next stock report into the priority queue:
        ___pq_insert(get_report(S));___

        // If the priority queue has more than 1000 reports,
        // delete the report with the smallest value.
        if (___pq_full(P)___)
            ___pq_delmin(P)___
    }

    // Add up the values of all 1000 reports in the priority queue
    int total = 0;
    while (___pq_empty(P)___) {
        total += ___pq_delmin(P)___;
    }

    return total;
}
```

3. **Heaps**

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

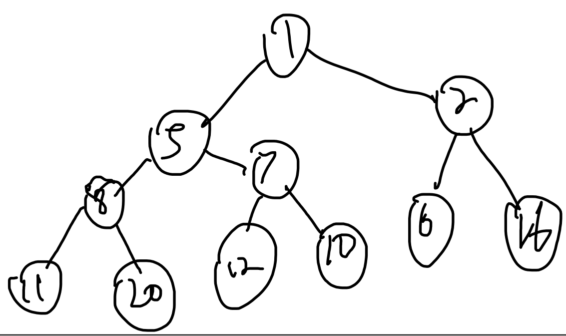
> *Order*: For each non-root element, its value is greater than or equal to the value of its parent.
>
> *Shape*: Each level of the min-heap is completely full except possibly the last level, which has all of its elements stored as far left in the last level as possible.

Consider:



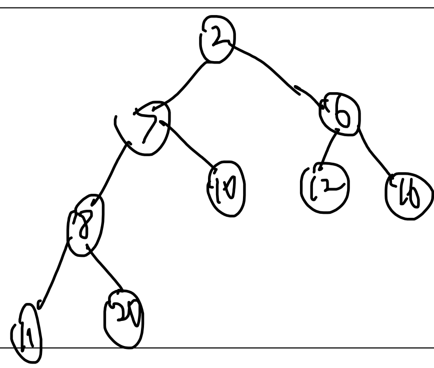
(1) (a) Draw a picture of the final state of the min-heap after an element with value 5 is inserted. Be sure that your final result satisfies both of the data structure invariants for a min-heap.

Solution:



(1) (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. Be sure your final result satisfies both of the data structure invariants for a min-heap.

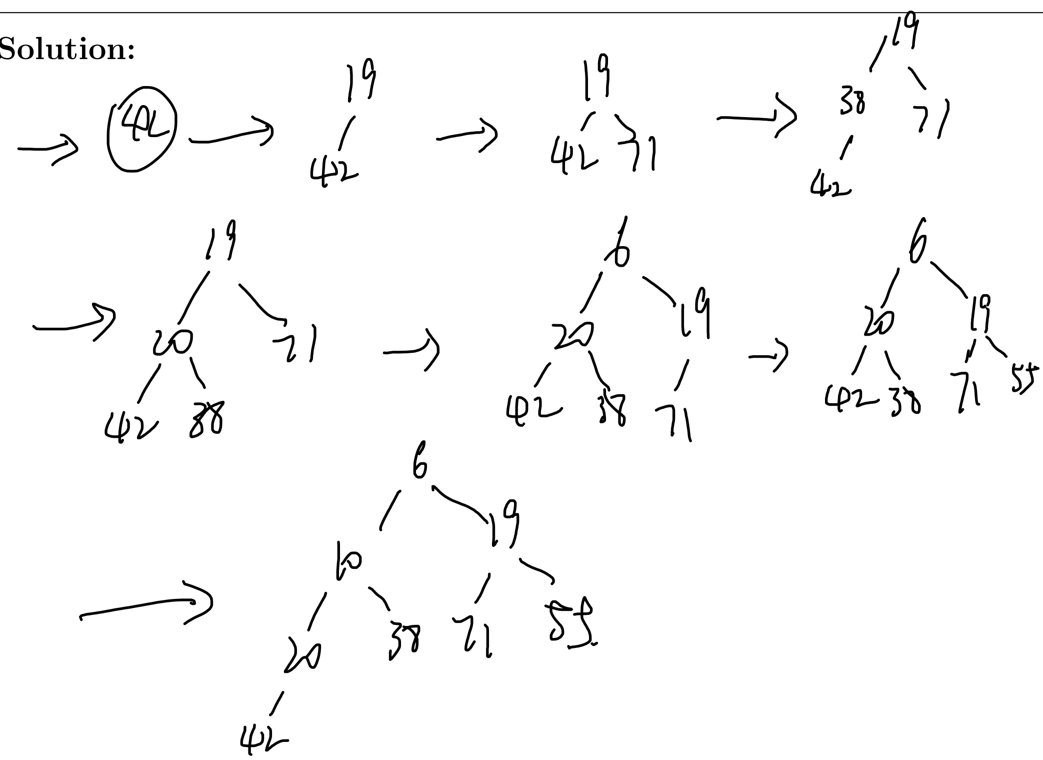
Solution:



(2) (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 pictures.

`42, 19, 71, 38, 20, 6, 55, 10`

Solution:



(1) (d) Assume a heap is stored in an array as discussed in class. Using the final min-heap from your previous answer, 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
X	6	10	19	20	38	71	55	42							

(1) (e) In a non-empty min-heap (thought of as a tree which may not necessarily be stored as an array), where must the maximum value be? You should be able to give a precise answer in one clear sentence.

Solution: suppose we have $N = 1+2+4+\cdots+2^r+R, = 2^{r+1}+R-1$ where $0 \leq R < 2^{r+1}$. Then there are R nodes in the last layer, and $2^r - \lceil \frac{R}{2} \rceil$ can be maximum in the second last layer. So the potential maximum is $R+2^r-\lceil\frac{R}{2}\rceil = 2^r+\lfloor\frac{R}{2}\rfloor = \lceil\frac{N}{2}\rceil$, $= \left\lfloor \frac{2^{r+1}+R}{2} \right\rfloor = \lceil\frac{N}{2}\rceil$

So the last $\lceil\frac{N}{2}\rceil$ element can be maximum.

(1) (f) What is the worst-case runtime complexity of finding the maximum in a min-heap if the min-heap has n elements? Why?

Solution: $O(N)$
because the number of values that need to be examined is: $\lceil\frac{N}{2}\rceil$

(2) (g) We are given an array A of n integers that we wish to sort and we decide to use a min-heap to help with the sorting. Our min-heap will be represented using an array as described in class.

For each integer in the array, one at a time, we insert the integer into the min-heap. Then we delete (the minimum) from the min-heap repeatedly, storing each deleted value back into the array, one at a time, starting from the beginning of the array.

What is the worst-case runtime complexity of this sorting algorithm using big O notation? Explain your answer concisely.

Solution: $O(N \log N)$

for each insertion, cost is $O(\log n)$

for each deletion, cost is $O(\log n)$

since there is N insertion and deletion, the total cost is $O(n \log n)$