

Solutions for Homework Assignment #1

**Answer to Question 1.** (16 marks)  $T(n)$  is  $\Theta(2^n)$ . To prove this, first note that an integer input  $i$  of size  $n$  bits is such that  $i \leq 2^n - 1$ .

1.  $T(n)$  is  $O(2^n)$ . This is because for *every* input size  $n \geq 1$ : (a) executing the for loop of Line 3 consists of doing *at most*  $i \leq 2^n - 1$  iterations, each one taking a constant time, and (b) executing Lines 1, 2, and 4 takes a constant time.

Therefore there is a constant  $c > 0$  such that for all  $n \geq 1$ : for *every* input of size  $n$ , executing F on that input takes *at most*  $c \cdot 2^n$  time.

2.  $T(n)$  is  $\Omega(2^n)$ . This is because for every input of size  $n \geq 1$ , there is an integer  $i$  of size  $n$  *that is odd*, namely  $i = 2^n - 1$ , and executing the code of F on *this* input causes the execution of  $2^n - 1$  iterations in the for loop of Line 3 (each one taking a constant time). Note that  $2^n - 1 \geq (2^n)/2$ .

Therefore there is a constant  $c > 0$  (namely,  $c = 1/2$ ) such that for all  $n \geq 1$ : there is *some* input of size  $n$  such that executing F on that input takes *at least*  $c \cdot 2^n$  time.

Since  $T(n)$  is both  $O(2^n)$  and  $\Omega(2^n)$ , it is  $\Theta(2^n)$ .

**Answer to Question 2.** (14 marks)

- a. (7 marks) [16,14,10,11,0,5,3,1]
- b. (7 marks) [16,11,14,5,1,3,10,0]

**Answer to Question 3.** (20 marks) We'll give algorithms for min-heaps. Algorithms for max-heaps are completely symmetric.

a. (14 marks)

- CHANGE-KEY( $A, i, key$ ), where  $1 \leq i \leq A.heap-size$ , changes the priority of element  $A[i]$  to  $key$  and restores the min-heap ordering property.

*Algorithm sketch:* Let  $x$  be the element in  $A[i]$ .

- If CHANGE-KEY( $A, i, key$ ) decreases the (key of)  $x$ , then “percolate  $x$  up” until it settles to the right place, i.e., until the parent of  $x$  is less or equal to  $x$ . To do so, keep comparing  $x$  with its parent, and swap the two if  $x$  is smaller. This procedure is similar to MIN-HEAP-INSERT.
- If CHANGE-KEY( $A, i, key$ ) increases the (key of)  $x$ , then “drip  $x$  down” until it settles to the right place, i.e., until  $x$  is less or equal to its children. To do so, keep comparing  $x$  with its children, and if one of them is smaller, then swap  $x$  with the smallest of its children. This procedure is similar to MIN-HEAPIFY.

- DELETE( $A, i$ ), where  $1 \leq i \leq A.heap-size$ , deletes the element  $A[i]$  from the heap.

*Algorithm sketch:* Let  $x$  be the element in  $A[i]$ . One way to delete  $x$  is to first use the CHANGE-KEY( $A, i, key$ ) procedure to change the key of  $x$  to “minus infinity” (an element smaller than anything in  $A$ ). This will make  $x$  percolate up all the way to the root of the min-heap  $A$ . Then execute HEAP-EXTRACT-MIN( $A$ ) to remove  $x$ .

**b.** (6 marks) Let  $h$  be the height of the min-heap  $A$  (note that  $h = \lfloor \log n \rfloor$ , where  $n = A.\text{heap-size}$ ). The worst-case time complexity the above algorithms is both  $O(h)$  and  $\Omega(h)$ , because: (1) they never take more than time proportional to  $h$ , and (2) they each have at least one execution that does take time proportional to  $h$  (e.g., for  $\text{CHANGE-KEY}(A, i, \text{key})$ , such an execution occurs when  $i = n$ , and the new  $\text{key}$  is smaller than any other key in  $A$ : this execution makes the leaf  $x = A[n]$  percolate up all the way to the root of the heap). So the worst-case time complexity of the above algorithms is  $\Theta(h)$ , i.e.,  $\Theta(\log n)$ .

**Answer to Question 4.** (20 marks)

**a.** (8 marks) A binomial heap  $H$  with  $n$  vertices consists of  $\alpha(n)$  trees. Let  $T_i$ ,  $1 \leq i \leq \alpha(n)$ , denote the trees of  $H$ . A tree  $T_i$  with  $n_i$  vertices has  $n_i - 1$  edges. So the total number of edges in  $H$  is  $\sum_{i=1}^{\alpha(n)} (n_i - 1) = (\sum_{i=1}^{\alpha(n)} n_i) - \alpha(n) = n - \alpha(n)$

**b.** (12 marks) Binomial heap  $H$  has  $n$  nodes before the insertions. By Part (a), it has  $n - \alpha(n)$  edges before the insertions. After  $k$  consecutive insertions,  $H$  has  $n + k$  nodes, hence it now has  $(n + k) - \alpha(n + k)$  edges. So the number of new edges created during the  $k$  consecutive insertions is:  
 $[(n + k) - \alpha(n + k)] - [n - \alpha(n)] = k + \alpha(n) - \alpha(n + k) \leq k + \alpha(n)$  edges.

As we explained in class, the number of pairwise comparisons between the elements of  $H$  needed to execute  $k$  consecutive insertions is equal to the number of new edges created during these insertions (each new edge is the result of a pairwise comparison, and each pairwise comparison creates a new edge in  $H$ ). So  $k$  consecutive insertions require at most  $k + \alpha(n)$  comparisons. By definition  $\alpha(n)$  is the number of 1's in the binary representation of  $n$ , therefore,  $\alpha(n) \leq \lfloor \log_2 n \rfloor + 1$ . So  $k$  consecutive insertions require at most  $k + \lfloor \log_2 n \rfloor + 1$  comparisons. Note that if  $k > \log_2 n$ ,  $k$  is the dominant factor in  $k + \lfloor \log_2 n \rfloor + 1$ . So, when  $k > \log_2 n$ ,  $k$  consecutive insertions require just  $O(k)$  pairwise comparisons (a constant number of comparisons per insertion on the average).