Computer Science CSC263H — January 30, 2020
St. George Campus — University of Toronto

Solutions for Homework Assignment #2

Answer to Question 1.

Whiz Warehouses needs can *mostly* be satisfied by using max binomial heaps to implement mergeable priority queues. Each warehouse stores its set of orders in a separate max binomial heap, where for order x, key$[x]$ is the integer amount of x's incentive. You can find descriptions of the operations on ordinary binomial heaps in the CLRS (2nd Edition) hand-out in course materials. The one operation that this approach will not satisfy with the required performance guarantee is finding the current maximum incentive in constant time: MAXIMUM on a max binomial heap with n nodes has worst-case $O(\log n)$ running time.

Provide constant-time maximum incentive look-up by augmenting a max binomial heap H with a field *max* to record the current maximum incentive. The augmented data structure is simply $H_A = (H, max)$.

In what follows assume that key[NIL] returns special value $-\infty$. The operations required by Whiz Warehouses are below, with a subscript "A". Each operation's correctness follows from the correctness of the component binomial max heap operations, and from the properties of the **max** function, in particular binomial heap operation MAXIMUM is called to find a maximum priority after operations that may re-arrange the heap.

- Make-Heap$_A$()

  allocate a new object H_A
  $H_A.H \leftarrow$ MAKE-HEAP()
  $H_A.max \leftarrow -\infty$
  return H_A

  WORST-CASE TIME COMPLEXITY:
  The new operation inherits worst-case time complexity $O(1)$ from MAKE-HEAP.

- Extract-Max$_A(H_A)$

  tmp $\leftarrow$ EXTRACT-MAX$(H_A.H)$
  $H_A.max \leftarrow$ key[MAXIMUM$(H_A.H)$]
  return tmp

  RUNNING TIME:
  Calling one procedure with $O(\log n)$ worst-case time complexity and then another with $O(\log n)$ worst-case time complexity produces one with $O(\log n)$ worst-case time complexity.

- Union$_A(H'_A, H''_A)$

  $H_A \leftarrow$ MAKE-HEAP$_A$()
  $H_A.max \leftarrow$ maximum of $H'_A.max$ and $H''_A.max$
  $H_A.H \leftarrow$ UNION$(H'_A.H, H''_A.H)$
  return H_A

  RUNNING TIME:
  The new operation inherits worst-case time complexity $O(\log n)$ from UNION, where n is the sum of the number of nodes in H' and H'', plus worst-case constant time complexity to calculate and update $H_A.max$. Thus the worst-case time complexity is $O(\log n)$.

- Insert$_A(H_A, x)$

  INSERT$(H_A.H, x)$
  $H_A.max \leftarrow$ maximum of $H_A.max$ and key$[x]$.

  RUNNING TIME:
  The new operation inherits worst-case time complexity $O(\log n)$ from Insert, plus worst-case constant time to access and update $H_A.max$, so the result has worst-case time complexity $O(\log n)$.

- Max-Incentive$_A(H_A)$
  1 **return** $H_A.max$

  RUNNING TIME:
  The new operation has worst-case time complexity $O(1)$.

Answer to Question 2.

The binary representation of 56 is $\langle 111000 \rangle_2$. Thus the 56-node binomial heap has three binomial trees: one with 2^5 nodes, one with 2^4 nodes, and the smallest with 2^3 nodes (note that these correspond to the 1s in the binary representation of 56). We know that each binomial tree with 2^k nodes, has height k, and the degree of its root is k, so the degree of the smallest tree's root is 3. After an EXTRACT-MIN operation the heap holds 55 nodes, and the binary representation of 55 has a "1" in the right-most position. So the smallest tree has $2^0 = 1$ node and its degree and height is 0.

Answer to Question 3.

ALGORITHM:
Search for the maximum key in B_1 *and* the minimum key in B_2 *in parallel* and *stop as soon as one of them is found*, as follows. Traverse down the rightmost path of B_1 and down the leftmost path of B_2, by following the right-child pointers in B_1 and the left-child pointers in B_2, *in parallel*, i.e., by alternating steps; stop this traversal as soon as you find a node x such that:

(*) x is in B_1 and x has no right child, ***or*** x is in B_2 and x has no left child.

Remove x from its tree (note that this takes only constant time as x is simply replaced by its only child), and then make x the root of the new, merged tree T with b_1 (the root of B_1) as its left child and b_2 (the root of B_2) as its right child.

CORRECTNESS:
From the ordering of keys in a binary search tree, the key at x is either the largest key of B_1 or the smallest key of B_2. Since every key in B_1 is smaller than every key in B_2, every key in B_1 is smaller than or equal to the key of x, *and* every key in B_2 is greater than or equal to the key of x.

Since: (a) each of B_1 and B_2 are binary search trees, and (b) the keys in x's left subtree are smaller than or equal to the key of x, and the keys in x's right subtree are greater than or equal to the key of x, the new tree rooted at x is also a binary search tree.

RUNNING TIME:
Note that the parallel traversals from the roots of B_1 and B_2 to the *first* node x that satisfies condition (*) can take at most $2 \cdot \min\{h_1, h_2\}$ steps. Removing x takes $O(1)$ time because it has only one child. Creating the new tree T and assigning the root's children also takes $O(1)$ time. Therefore, in total, the algorithm runs in $O(\min\{h_1, h_2\})$ time in the worst-case.