Homework Assignment 4

Li Kunle

Groupmates: Han Zifei, Lin Jiakai, Su Chang (Alphabetically ordered)

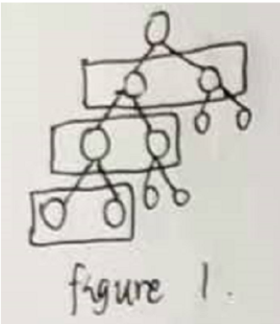
Exercise 1

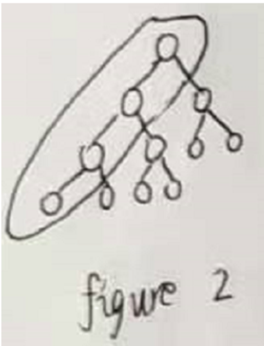
Ex1

(i) Because the depth of one heap is logn, siftUp(n) means compare the child with its parent from the bottom to its top and swap if necessary, so the max running time is 2*logn, and hence O(logn).

Inserting into a heap means inserting one element to the bottom of one heap and then run siftUp(n) for the element, so the time is also O(logn).

(ii) First we choose to compare elements whose position is like the figure 1 and put the max/min(**) one to the left, then the left path like in the figure 2 is what we need. Because the depth of one heap is logn, so such process needs logn comparisons, and such path has logn elements. Then use binary search to find the needed position and we know the complexity is O(logn). Then the total comparisons is logn+O(loglogn).





(**) If the heap is a max-heap, then we need the max one; if the heap is a min-heap, then we need the min one.

Exercise 2

1. We can set up a hash function $H(x, y) = p \cdot x + q \cdot y$, where p and q are two distinct prime numbers. For each pair (x_i, y_i) in R, compute $H(x_i, y_i)$ and store it in a **vector**. Then for each pair (x_i, y_i), compute $H(y_i, x_i)$ and check whether that value is in the **vector**. If for every pair (x_i, y_i) in R, $H(y_i, x_i)$ is also available in the **vector**, then R is symmetric. Otherwise, it is not symmetric.

Exercise 3

Ex3

(i) Now we take r_1 and r_2 as an example. For r_1, p_1 points to the oldest child of r_1, p_2 to r_2, p_3 to the parent. Then compare the key of r_1 and r_2. Then combine the root of the tree with the smaller key value to the parent of the root of the other tree. And the tree with the bigger root key becomes the oldest child of the other tree. For this new tree, p_1 points to the oldest child (r_1 or r_2), p_2 to r_3, p_3 to the parent. Then do the same for r_3 and r_4. But here p_3 points to the previous root (r_1 or r_2). Then after this process, p_2 points to r_3 or r_4.

After combining r_1 with r_2, r_3 with r_4, etc, we combine the youngest sibling with its next older sibling using the same way because we always have a pointer pointing to the next older sibling. After this process, we repeat this for many times until we combine all root nodes and after that, pairing heaps is finished.

(ii) For this time, we take r_1 and r_2 as an example. For r_1, p_1 points to the oldest child of r_1, p_2 to r_2. Then combine the root of the tree with smaller key value to the parent of the root of the other tree. Then we get a new tree with the root (smaller key root of r_1 or r_2). And the tree with bigger root key within r_1 and r_2 becomes the oldest child of the other tree.

For this new tree, p_1 points to the oldest child (r_1 or r_2), p_2 to r_3. Then do the same for r_3 and r_4. After combining r_3 and r_4, p_2 points to the new tree's root, and then we combine these two trees.

After that, combine r_5 and r_6 and combine the older tree with the new tree. Repeat this process until we combine all root nodes and after that, pairing heap is finished.

Exercise 4

To keep the property about degree for the Fibonacci heap (and thus also for *McGee Heap*), we need the function to consolidate, which has amortized time complexity $O(\log n)$. For Fibonacci Heap, *insertion* and *union* are followed by consolidation, but that function cannot be applied for *McGee Heap* as it supports just the mergeable heap operations. Besides, even if no nodes are consolidated, we still need to check all child root nodes have different degrees. So the worst-case running time of *insert* and *union* is $O(\log n)$ for *McGee Heap*.

Since the other operations, including *min* and *delete_min*, are not affected by this difference of *McGee Heap*, the worst-time complexity of other operations remain unchanged. Furthermore, as *McGee Heap* is binomial heap, a *McGee Heap* with n nodes has at most $\log n$ trees and its degree is at most $\log n$. As k trees need at most $k - 1$ basic consolidations, we need worst-case time complexity $O(\log n - 1) = O(\log n)$. That further shows that function "consolidate" has time complexity $O(\log n)$.