CSC263, Assignment #5

Brendan Neal[1], Filip Tomin[2]

[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. The first part of σ will take k time because UNION takes constant time, so UNION $\in O(1) \implies$ k UNIONs $\in k * O(1) \in O(k)$. The second part of σ consists of k' finds. In the worst case, the resulting tree from the UNIONs would be a single chain of nodes. Since we will be using FIND path compression, FIND will set the parent/representative of each of the nodes in this chain to be the first node in the chain. Since the length of this chain is $O(n)$, iterating through each of the nodes and setting their parents/representatives will take $O(n)$ time.
   If FIND comes back to this tree, since the tree is now under path compression due to the first FIND, all nodes will be exactly one edge away from the root node. Thus, it will take $O(1)$ time to find x, as each node is at most one edge away from the root node. Using amortization, in the worst-case we can have one compression FIND, taking $O(n)$ time, and n single FINDs taking $O(1)$. Our total steps would be equal to $O(n+n(1)) = O(2n)$ steps. Averaging this over n FINDs gives us $O(\frac{2n}{n}) = O(1)$. Thus, with k' FINDs, we would take k' steps, which brings our running total to $O(k + k')$ steps for the entirety of σ.

2. (a) We will be organizing the arrays in a fashion similar to the organization of S_k trees in a binomial heap. For example, consider a binomial heap of 3 elements. In binary, 3_{10} is represented as 11_2. This means that there is one tree of size 2 (corresponding to the first 1) and one tree of size 1 (corresponding to the second 1). However, for this data structure, we will be using sorted arrays instead of S_k trees. We will refer to an array of size n as an n-array. We will be using the symbol $\rightleftharpoons$ to represent the relationships between the arrays in the doubly-linked list (top arrow represents successor, bottom arrow represents predecessor). Drawing out the data structure for the two given arrays:

      i. $I = \{3, 5, 1, 17, 10\}$ (5 elements)
      $5_{10} = 101_2 \implies$ one 4-array, one 1-array
      We are going to put 3 in the 1-array and the remaining elements in the 4-array. Thus, the data structure looks like:
      $\{(3) \rightleftharpoons (1, 5, 10, 17)\}$

ii. $I = \{17, 8, 3, 10, 1, 12, 6\}$ (7 elements)
$7_{10} = 111_2 \implies$ one 4-array, one 2-array, one 1-array
We are going to put 17 in the 1-array, 8 and 3 in the 2-array and the remaining elements in the 4-array. Thus, the data structure looks like:
$\{(17) \rightleftharpoons (3, 8) \rightleftharpoons (1, 6, 10, 12)\}$

(b) In the worst-case, when calling SEARCH(x), the element x will not exist within the arrays, thus the search algorithm would have to check every individual array for x. The question states we will be using a binary search algorithm on each array, so the total worst-case time complexity would be the worst-case time complexity of BinarySearch multiplied by the number of arrays.
Since the number of arrays is dictated by the binary representation of the total number of elements, the maximum number of arrays formed by n elements would be $log_2 n$. By the definition of BinarySearch, the worst-case time complexity of BinarySearch would be $O(log_2 n)$. Then calling it on every array would give us a total worst case time complexity of $O(log_2 n * log_2 n) = O((log_2 n)^2)$

(c) There are 3 steps to the algorithm for INSERT(x).
The first step is creating a singleton array containing just the element x to be inserted. This step is simple and can be done in $O(1)$ time.
The second step is inserting this singleton set into the beginning of the linked list, which is also simple and can be done in $O(1)$ time.
The final step is to iterate through the linked list while it contains two arrays of the same size, merging the two same sized arrays into one array while also using the Merge from MergeSort to make sure the merged array is still sorted in increasing order. In the worst-case, the algorithm would need to do this through the entire linked list, continuously inserting and merging until the end of the list is reached. We will assume that the linked list stores the sizes of the arrays at each index of the linked list, thus making a size comparison would take only one step, since we are comparing every element of the list at some point and the list has $O(log_2 n)$ elements, then the comparisons would add $O(log_2 n)$ steps. By the definition of Merge, the worst-case time complexity of Merge is $O(n)$, and since we are also calling it on every array, merging each array would take a total of $O(nlog_2 n)$ steps. The total worst-case time complexity would then be $O(nlog_2 n + log_2 n) = O(nlog_2 n)$.

(d) To help examine this question, we created a table which shows the cost and total cost of up to 16 inserts:

Inserts (i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Cost (c_i)	1	3	1	7	1	3	1	15	1	3	1	7	1	3	1	31
Total Cost	1	4	5	12	13	16	17	32	33	36	37	44	45	48	49	80

i. AGGREGATE METHOD:
As defined above, we already know the worst-case run time of INSERT(x) to be $O(nlog_2 n)$. If we wanted to do n of these operations, we would have a

worst-case of $O(n^2 log_2 n)$. While this is a correct upper bound, it is extremely inaccurate. In fact, this assumes that our list is somehow always full of arrays, when we are actually starting with an initially empty list.
When we account for starting with an initially empty list, calling INSERT(x) will not require merging sets as often, allowing us to remove the cost of Merge from INSERT(x), which was $O(n)$. Then taking our previous upper bound, we would have a new upper bound of $O(nlog_2 n)$.
Since we're looking for the amortized cost over n operations, we then divide this new upper bound by n to get the amortized cost of INSERT(x), which would be $O(\frac{nlog_2 n}{n}) = O(log_2 n)$.
Thus, the amortized cost of INSERT(x) on n elements from an initially empty list would be $O(log_2 n)$.

ii. ACCOUNTING METHOD:
Looking at the table, the biggest jumps in c_i happen whenever i is equal to a new power of 2. If we adjust our cost to be enough to break even on these values, the average cost will be enough to stay positive until the next new power of 2 is reached. If we look at $i = 4$, the total cost at that point is 12, the average cost would be $12/4 = 3$ for each INSERT(x), and we'd still be in the positive until we hit $i = 8$, which would require an average cost of $32/8 = 4$ until we hit $i = 16$ and so on.
Considering the binary representation of these numbers; $4_{10} = 100_2, 8_{10} = 1000_2, 16_{10} = 10000_2$, we can see that the average cost increases by 1 for each digit in the binary representation, no matter what the value of the digits are. (1011_2 inserts would have the same average cost as 1000_2 inserts). Then we can also relate the cost to the value of the highest bit, which gives us an average cost of $\lfloor log_2 n \rfloor + 1$, where n is the total number of Inserts to be done.
Thus, the upper bound for the amortized cost of INSERT(x), with n Inserts, is $O(\lfloor log_2 n \rfloor + 1) = O(log_2 n)$.

(e) The algorithm of DELETE(x) is as follows:
1) Remove x from the array that it belongs to. Refer to this array as A_x.
Since we are given a pointer to an element, we can find and remove the element in $O(1)$ time. After this first step, A_x is missing an element and therefore needs one more element to satisfy the required array-size property.

2) Extract some number y from the smallest array A_s in the linked list. Insert it into A_x.
Finding the smallest array will take $O(1)$ time, because in a linked list, the smallest element is the first one in the list. Inserting y into A_x will take $O(n)$ time (to find the insertion point).

3) A_s now has $2^k - 1$ elements, where $k \in O(\log_2 n)$ is some arbitrary value. Note $k \in O(\log_2 n) \implies 2^k \in O(2^{\log_2 n}) \implies 2^k - 1 \in O(n) \implies |A_s| \in O(n)$. We will

now "shatter" A_s into its constituent arrays. To do this, find $|A_s|_2$ to determine the number of n-arrays that A_s will be broken into. For example, if $|A_s| = 2$, $|A_s|_3 = 11_2$, which means that A_s will be broken into one 2-array and one 1-array.

To do this, we will loop through A_s, extract the first element and insert it into the linked list as a singleton array. Then we will extract the next two elements, insert them into the linked list as another array, then the next four elements, insert them into the linked list as another array and so on until the array is depleted. Since we have to loop through all the elements in A_s and $|A_s| \in O(n)$ and insert them into a linked list as an array (constant time, in $O(1)$ time), this step will take $O(n)$ time.

In summary, this algorithm's worst-case time complexity is $\in O(1)+O(n)+O(n) \in O(n)$ (because $O(n)$ is the most dominant term in the sequence).