By: Eric Koehli, Jacob Chmura, Conor Vedova

Question 1. [1 MARK]

For the following parts of this question, we define I to be a set of integers such that we keep the set of integers I in a doubly-linked list L of sorted arrays with the following properties:

Let $n = |I|$ be the number of elements in I and $\langle b_{k-1}, b_{k-2}, \ldots, b_0 \rangle$ be the binary representation of n. Then it follows that $k \in \mathcal{O}(\lg(n))$. Further, for $i = 0, 1, \ldots, k-1$, if $b_i = 1$, then the doubly-linked list L contains a sorted array A_i of size 2^i; That is, $|A_i| = 2^i$.

a. Draw two instances of the data structure L, one for set $I = \{6, 8, 4, 13, 9\}$ and one for set $I = \{21, 12, 7, 14, 5, 16, 10\}$.

i. For the first set, $I = \{6, 8, 4, 13, 9\}$, so $|I| = 5 = \langle 101 \rangle_2$. Thus, from the binary representation of $|I|$, we can see that L contains two nodes, each with a sorted array of size $|A_i| = 2^i$:

$$L = Node(A_0 : [9]) \Longleftrightarrow Node(A_2 : [4, 6, 8, 13])$$

ii. For the second set, $I = \{21, 12, 7, 14, 5, 16, 10\}$, so $|I| = 7 = \langle 111 \rangle_2$. Thus, from the binary representation of $|I|$, we can see that L contains three nodes, each with a sorted array:

$$L = Node(A_0 : [10]) \Longleftrightarrow Node(A_1 : [7, 16]) \Longleftrightarrow Node(A_2 : [5, 12, 14, 21])$$

b. Describe an algorithm to perform a `Search(x)` operation with this data structure. Give a good upper bound on the worst-case time complexity of your algorithm (using the $\mathcal{O}$ notation) and justify your answer.

Perform `search(L, x)` in the following way:

1. Loop through the A_i lists. If x $\geq$ root of A_i, then perform binary search on this list. Else, do not perform binary search on this list as we know x cannot be in it.

Let $T(n)$ be the worst-cast time complexity of calling `search(L, x)` on L. In the worst case, x is not contained in L at all and n is an integer such that the binary representation of $n = \langle 1, 1, \ldots, 1 \rangle_2$, so the algorithm would have to check every node in L. Since the number of nodes in L is *at most* k where $k \in \mathcal{O}(\lg(n))$ (by the way we defined the data structure for L), then we are iterating *at least* $\lg(n)$ times. But we must also search for the input x within each node. Since each array A_i is sorted, we can use binary search, which we know has a runtime in $\mathcal{O}(\lg(n))$. Therefore, the total worst-case time complexity would be the worst-case time complexity of `BinarySearch` multiplied by the number of nodes in L. Thus, we can see that the "outer loop" of checking each node has a runtime of $\mathcal{O}(\lg(n))$ and for each node, binary search has a runtime of $\mathcal{O}(\lg(n))$, so the total cost of our `search` algorithm has a time complexity of

$$T(n) = \lg(n) \times \lg(n) \in \mathcal{O}(\lg^2(n))$$

c. Describe an algorithm to perform an `Insert(x)` operation with this data structure. Give a good upper bound on the worst-case time complexity of your algorithm (using the $\mathcal{O}$ notation) and justify your answer. Hint: Think about how we insert an element in a Binomial Heap, and use the Merge part of MergeSort.

There are esentially three steps to perform when inserting an integer x into the data structure L:

```
1. Place x into an empty array, then wrap the array in a Node

2. Prepend the new node to the head of the doubly-linked list

3. while the array length of two consecutive nodes are equal:
        use Merge on the two sorted arrays
        remove the empty node that results from merge
```

Time complexity of insert

Let $n = |I|$ and $T(n)$ be the worst-case time complexity of calling `insert`. From our algorithm as described above, we can see that the first two steps take constant time and can be completed in $\mathcal{O}(1)$ time. In the third step, we iterate over all the nodes of L, which we know is at most $\mathcal{O}(\lg(n))$. But for each node, we may need to perform a `Merge` on the two sorted arrays to maintain the sorted order. The worst-case of `Merge` is $\mathcal{O}(n)$. Thus, in the worst-case of `insert`, the binary representation of $n = \langle 1, 1, \ldots, 1\rangle_2$, so the algorithm must iterate across all the nodes of L performing a `Merge` at each node. Since the number of nodes in L is at most $\mathcal{O}(\lg(n))$, and because we double the size of each array, the final result of L will be a single node containing an array of size $2\,|A_{k-1}|$. Hence, the total worst-case time complexity of `insert` is $T(n) = n\lg(n) \in \mathcal{O}(n\lg(n))$.

d. Suppose we execute a sequence of n Inserts starting from an empty set I. Determine a good upper bound on the amortized time of an `Insert` (i.e., the worst-case total time to execute these n Inserts divided by n). Justify your answer in two different ways, i.e., give two separate proofs, each proof using a different argument (e.g., use aggregate analysis and the accounting method).

For part **d.** of question 1, we start by creating a table to develop some intuition.

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

As shown in the table, when we call `insert`, we know that the cost is at least 1 for creating a new array for the new integer. For each `Merge` that occurs, the cost increases by 2 times the size of the merging arrays since the size of the array is doubling.

Aggregate Analysis

Since we already analysed that the worst-case runtime of `insert` was in $\mathcal{O}(n\lg(n))$ and since we are performing n inserts, the worst-case of executing a sequence of n inserts is at most $\mathcal{O}(n^2\lg(n))$. But after performing some inserts we can see that each operation does not necessarily reach close to $\mathcal{O}(n^2\lg(n))$.

In order to better understand the amortized runtime, we will understand how many times each bit flips (from 1 to 0) when performing n inserts. As demonstrated in the table, the first bit flips $n/2$ times. The second bit will flip $n/4$ times because the first bit will flip only $n/2$ times and half of those times, the second bit starts at 0. The $\lfloor logn \rfloor$ bit flips 1 time using the same logic. Each bit at position i flips $\frac{n}{2^i}$ times. The runtime of each bit flipping from 1 to 0 is $log(n)$ because of merge. Also, the runtime of each bit flipping from 0 to 1 is constant so it can be ignored. The amortized runtime will therefore equal:

$$\begin{aligned}
\frac{\sum_{i=0}^{i=\lfloor log(n)\rfloor} \frac{n}{2^i} * log(n)}{n} &= logn(n) * (\sum_{i=0}^{i=\lfloor log(n)\rfloor} \frac{1}{2^i}) \\
&= logn(n) * (1 + 1/2 + 1/3 + ... + 1/n) \\
&\leq log(n) * 2
\end{aligned}$$

So, the amortized runtime is $\mathcal{O}(\lg(n))$.

Accounting Method

As we can see from the table, the largest cost occurred happens whenever i is a power of 2. To amortize the cost, we would like to increase the average cost for each `insert` in order to "recharge" so that we will have just enough to break even when $i = 2^m$ for some positive integer m.

Consider the average cost when i is a power of 2. When $i = 4$, the average cost would be $12/4 = 3$. When $i = 8$, the average cost would be $32/8 = 4$. When $i = 16$, the average cost would be $80/16 = 5$. This pattern tells us that the average cost incurred increases by 1 for every power of 2 that i increases by. Thus, a good estimate based off this pattern is to charge $\lfloor \lg(i) \rfloor + c$, where c is a constant to be determined. Through some trial and error, we can see that if we choose $c = 3$, then we are always charging enough to break even whenever i is a power of two.

Therefore, a good upper bound on the amortized time complexity of n `Inserts` is $\lfloor \lg(n) \rfloor + 3$, which is $\mathcal{O}(\lg(n))$.

e. Describe an algorithm to perform a `Delete(x)` operation, i.e., given a pointer to integer x in one of the arrays of L, remove x, in $\mathcal{O}(n)$ time in the worst case. Explain why the worst-case time complexity of your algorithm is $\mathcal{O}(n)$.

Given a pointer to an element x, we perform delete from L as follows:

```
1.  Remove x from its array

2.  If x is not removed from the $A_i$, where i is the least significant bit that equals 1, find
the least significant bit at some position i that equals 1. Take the last element of this list
$A_i$ and insert it into the array that contained x such that the array remains sorted.

Else, Skip step 2 and carry on to Step 3, with i being the least significant bit that equals 1.

3. Take $A_i$ and remove a list, of minimal size, from the end of $A_i$ such that the number of
elements remaining in $A_i$ is a power of 2 (Sometimes, you do not need to remove any elements).
Slot the result of $A_i$ into its corresponding slot.

4. Repeat part 3 except use the list that you removed from $A_i$ as the new list you are removing
from.

5. Assign the least significant bit that equals 1 as the head and ensure all pointers are
corrected during steps 1 to 4.
```

Note: This process ensures that no merge operations must be performed as position i is the least significant bit that equals 1. Also, every split of the lists results in a number of elements less than any positions that are filled with lists already.

Time Complexity Analysis

For this time complexity analysis, we will analyse the runtime of each of the control blocks above:

Since we are given a pointer to x, step 1 can be performed in constant time.

In Step 2 (in the worst case), we are simply accessing the head of the Data Structure, removing the last element of A_i, and inserting it into the A_i that we removed x from, which is $\mathcal{O}(n)$ time.

Steps 3 to 4 take a total of $\mathcal{O}(log(n))$ time because the list A_i can be split at maximum $log(j)$ times where j equals the size of A_i.

Step 5 is constant time.

Thus, the overall worst-case of `delete` happens in $\mathcal{O}(n)$ time.

Answered by: Eric, Conor
Verified by: Jacob

Question 2. [2 MARKS]

a. The algorithm that Furio has in mind is Breadth First Search traversal of the graph G. To solve problem P, traverse every vertex, and run BFS on every vertex that's a house. As you run BFS, you can stop as soon as you encounter a hospital node in the BFS generated tree, since the first hospital node you encounter must be the closest hospital node to the current house, by invariant of BFS trees. After repeating the process for every house node, you will have the closest hospital node to each house on G.

Now, since we are running the BFS algorithm a total of h times, where h is the number of houses. This gives a running time of $O(h(|V| + |E|))$, since BFS is known to run in $O(|V| + |E|)$. However, by the assumption that the number of houses is constant, the is equivalent to $O(|V| + |E|)$. Thus P is solved in desired running time, and Furio is correct.

b. We can use Breadth First Search traversal of G, in a modified way to solve the problem in $O(k(|V|+|E|)$, where k represents the number of hospital nodes in G. Since we know the running time of BFS to be $O(|V|+|E|)$, the key idea is to call BFS once for every hospital node. More specifically, begin by traversing through all vertices. At each house vertex, create a pointer into a table. This table will hold the results of our BFS trees. Next, for each hospital vertex, run BFS, and for each house node generated by the BFS tree for that hospital, overwrite the distance from that hospital into the table entry for that house if the current entry for that house is either empty, or is larger than the current distance found. Repeat for all hospital nodes. Notice that after this procedure, our table will be as long as the number of houses, and each house position will have the smallest distance from all hospital nodes. To finish, simply go through each entry, and take the value at that entry as the smallest distance to any hospital in G.

Now an analysis of the complexity. To distinguish the house nodes and create pointers into our table takes constant amount of work for each house, and since there are no more houses then vertices, this step is $O(|V|)$. Running BFS k times (once for each hospital) is $O(k(|V| + |E|)$. Lastly, we loop through our table once again. Thus the total running time is $O(k(|V| + |E|))$, and Paulie is right.

c. Tony is correct. The intuition for solving the problem is that we don't care about *which* hospital provides the shortest distance for a house, rather we just need the shortest distance to *some* hospital. With this in mind, begin by adding a node to G. We call this node *hospital.mother*. Next we connect this hospital mother to all hospital nodes in G, by means of the adjacency list. Finally We call BFS from the *hospital.mother*. The distance from the "hospital mother" to any house is one more than the actual distance from that house to any given hospital due to the extra edge connecting the "hospital mother". So for each house in the BFS tree, take the minimum distance to some hospital as one less than the minimum distance found by BFS.

Now an analysis of the complexity. To add the extra vertex into G, and connect it to all other hospitals, only requires traversing through each vertex in order to find all hospital nodes. Since the number of hospitals in G is no more than the number of vertices, this step is $O(|V|)$. Next, calling BFS once takes $O(|V| + |E|)$. Then for any given house vertex, when asked for its shortest distance to any given hospital, we always make sure to subtract one before giving the answer to account for the extra vertex added into G. This step is constant. Thus, we maintain $O(|V| + |E|)$, and Tony is correct.

Answered by: Jacob
Verified by: Conor, Eric