CSC263 Assignment 5

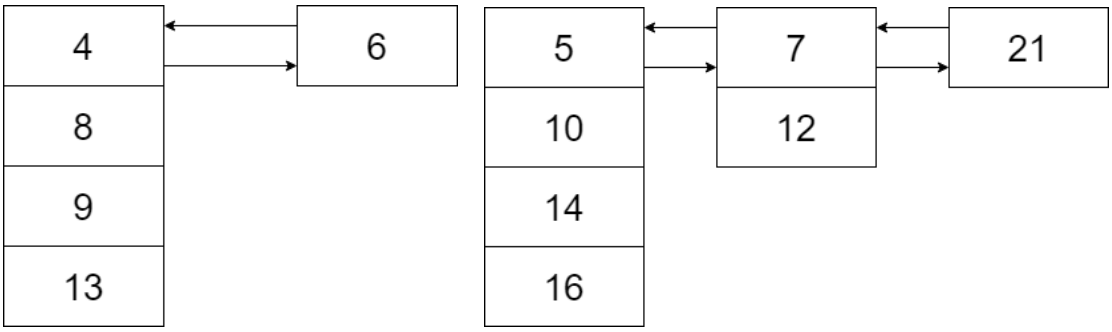
Yongzhen Huang, Xinyi Liu, Kewei Qiu

March 14, 2019

Question 1

Written by: Xinyi Liu, Kewei Qiu Checked by: Yongzhen Huang

a)



b)

Algorithm:

Suppose that the first elements of A_i are linked together as a doubly linked list. For each A_i starting from A_0, we perform binary search on it (since each A_i are sorted arrays) until we find x.

Worst-case time complexity:

The upper bound of this algorithm is $\mathcal{O}(\log^2 n)$. First, there are $\mathcal{O}(\log n)$ arrays. Then, in the worst case, we have to binary search on all the arrays, each costs $\mathcal{O}(\log n)$. Therefore, total is $\mathcal{O}(\log^2 n)$.

c)

Algorithm:

The algorithm we use is similar as INSERT function for binomial heap
1. Create a new linked list node storing only one element, which is the number we want to insert into the original list.
2. Merge the new linked list node with the original linked list by performing similar steps as INSERT function for binomial heap.

Worst-case time complexity:

The cost of creating a new linked list node which stores only one element is $\mathcal{O}(1)$.

We claim the cost of merging the new linked list node with the original one is $\mathcal{O}(n)$. Similar with INSERT function of binomial heap with length $log(n)$, which may create at most $log(n)$ key-comparisons, for a linked list of size $log(n)$, our algorithm may create at most $log(n)$ array-merges. Since arrays stored in original linked list are sorted, and length of them are 2^0, 2^1, ...,$2^{log(n)-1}$, and by the merge part of MERGESORT, merging two sorted arrays of size h needs at most 2h comparisons, the total cost of this part is 2 * 2^0 + 2 * 2^1 + ... + 2 * $2^{log(n)-1} = 2^{log(n)} - 2 = \mathcal{O}(n)$.

Hence the total worst-case time complexity of this algorithm is $\mathcal{O}(n)$.

d)

Claim: The upper bound on the amortized time of an INSERT is $\mathcal{O}(log(n))$.

Aggregate analysis:

Since the linked list is created based on the binary representation of number of elements, n, stores in it, we can view it as a binary counter.

As we discussed in class, the lowest bit of the counter changes every $2^0 = 1$ increment, the second lowest bit of the counter changes every $2^1 = 2$ increments, ..., the highest bit of the counter changes every $2^{log(n)-1} = \frac{n}{2}$ increments(since when $|I| =$ n the size of L id $log(n)$).

Hence for n insertions, the lowest bit will change n times, the second lowest bit will change $\lfloor \frac{n}{2} \rfloor$ times, ..., the highest bit will change $\lfloor \frac{n}{2^{log(n)-1}} \rfloor$ times.

For the ith bit each change either add an array of length 2^{i-1} to an empty position, which costs a constant time, or merge two arrays of length 2^{i-1}, which costs $2 * 2^{i-1} = 2^i$ as we discussed in part(c).

Hence by calculating, the total cost of executing a sequence of n INSERTS starting from an empty set is n (cost for inserting n single elements) + $\frac{n}{2^1}$ * 2^1 + $\frac{n}{2^2}$ * 2^2 + ... + $\frac{n}{2^{log(n)}}$ * $2^{log(n)} = n + nlog(n) = \mathcal{O}(n + nlog(n))$, and the amortized time of an INSERT is $\frac{\mathcal{O}(n+nlog(n))}{n} = \mathcal{O}(1 + log(n)) = \mathcal{O}(log(n))$.

Accounting method:

Just like what we did in accounting method, we consider the insertion process as the increment of a binary counter.

We charge \$1 + $log(n)$ for each insertion, \$1 simply for changing 0 to 1, because similarly as we discussed in class that each time increment a binary counter will cause exactly one bit of "0" to become "1", here each insertion will cause exactly one empty position in I to be filled with an merged array. The rest \$$log(n)$ is attached on this new array, to pay for its merge in future(if have one). We choose $log(n)$ here because for each single element stored in linked list of size $log(n)$, the array which contains it will go through $log(n) - 1$ array-merges to have the largest possible length, which means that every node will experience at most $(log(n) - 1)$ comparisons.

Hence in conclusion, the amortized time of an INSERT is $\mathcal{O}(log(n))$.

e)

Algorithm:

1. Delete x from the array it belongs.
2. Remove the array which contained x before from the original linked list.
3. Create a new linked list storing elements of this array using by adding pointers between first elements of arrays in the new linked list.
4. Merge the new linked list and the original one.

Worst-case time complexity:

Deleting an element from array and removing an array from a linked list both cost constant time.

Since the original linked list L stores n elements, by part(c) the largest array in it has size $2^{log(n)-1} = \frac{n}{2}$. Since this array is already sorted, to create a new linked list based on it, what we need to do is just split it into pieces of smaller arrays based on the binary representation of its size, like what we did in part(a), and add pointers between first elements of these smaller arrays. Based on the largest possible size of this array, which is $\frac{n}{2}$, the largest possible size of the new linked list is $log(\frac{n}{2}) = log(n) - 1$. Adding pointers between two element costs constant time. Hence the worst-case cost of creating a new linked list storing these elements is $\mathcal{O}(log(n))$.

The size of original linked list after removed one array is $log(n) - 1$, with biggest possible size of stored array $2^{log(n)-1} = \frac{n}{2}$. The biggest possible size of the new linked list is $log(\frac{n}{2}) = log(n) - 1$, also with biggest possible size of stored array $2^{log(n)-1} = \frac{n}{2}$. Similar as UNION algorithm of binomial heap, merging these two linked lists will cause at most $log(n) - 1$ array-merges, with largest possible array size $2^{log(n)-1}$, hence the total cost is $2 * 2^0 + 2 * 2^1 + ... + 2 * 2^{log(n)-1} = 2^{log(n)} - 2 = \mathcal{O}(n)$.

In conclusion, the total worst-case time complexity of this algorithm is $\mathcal{O}(1 + log(n) + n) = \mathcal{O}(n)$.

Question 2

Written by: Yongzhen Huang, Xinyi Liu, Kewei Qiu
Checked by: Yongzhen Huang, Xinyi Liu, Kewei Qiu

a)

Furio uses BREADTH_FIRST_SEARCH on each houses. From what we discussed in class, the time complexity of BREADTH_FIRST_SEARCH is $\mathcal{O}(|V| + |E|)$. Since the number of houses is a constant number, the time that this algorithm costs is equal to $\mathcal{O}(|V| + |E|)$ times a constant number which is still $\mathcal{O}(|V| + |E|)$. Hence Furio is right.

b)

Paulie uses BREADTH_FIRST_SEARCH on each hospital, and uses a linked list whose size is equal to the number of houses to store the shortest time to reach some hospital vertex.

For the first hospital, Paulie just runs BREADTH_FIRST_SEARCH on the graph by using it as the starting point, which creates a BFS Tree, and stores the length of the path from root (i.e. the first hospital) to each node (i.e. one house) in the BFS Tree into the corresponding position of this house of the list, during the process of each iteration. Then, for the rest of hospitals, run BREADTH_FIRST_SEARCH on each of them by using each of them as the starting point. For each time running the algorithm, every time Paulie discovers a node (house), he compares the path length from the current root (hospital) to this node (house) in BFS Tree with the value stores in the corresponding position of this house of the list. If the path length is smaller than the value stored in the list, Paulie replaces the original value with the path length; if the path length is longer or equals to the value stored in the list, do nothing to the list and continue BREADTH_FIRST_SEARCH.

This algorithm is correct because as we discussed in the lecture, in the BFS tree cerated by running BREADTH_FIRST_SEARCH on node n, the path length from one node to the root n is the length of the shortest path between these two nodes in the original graph. By comparing distances from each house to each hospital, we can finally find out the shortest time to reach some hospital vertex for each house./

The time complexity of BREADTH_FIRST_SEARCH is $\mathcal{O}(|V| + |E|)$. Since we run this algoritim k times, and compare and (possible) replacement in each iteration all take constant times, the total worst-case time complexity for this algorithm is $\mathcal{O}(k(|V| + |E|))$. Hence Paulie is right.

c)

Tony is correct. The algorithm will be the following,

First search through V to find all the hospitals. Then we apply BFS on each of the hospitals. However, each time we only explore one layer of nodes for each hospital (i.e. the discovery of houses through a path stops whenever one of the neighbouring nodes is hospital or it has been discovered). Whenever a house is discovered, we mark that house as discovered with a field called *discovered* set to true. Then, when a hospital is trying to discover houses, it only discover those that are not yet marked as discovered. All the hospitals take turns in this fashion. By the end, all the houses are discovered by one of the hospitals and the shortest paths can be found from the BFS trees.

First searching through V for hospitals takes $\mathcal{O}(|V|)$; the BFS process takes $\mathcal{O}(|V| + |E|)$ since it only goes through each house and edge once. Therefore, the total time is $\mathcal{O}(|V| + |E|)$ and hence Tony is correct.