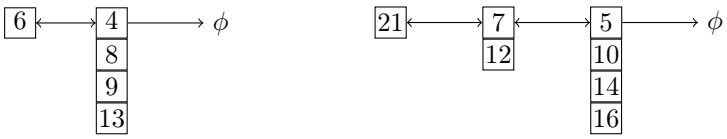
CSC263 Assignment 5

Angela Zhu, Xinyi Ji, Zhuozi Zou

March 14, 2019

1 Problem 1

(Parts a, b, c written by Zhuozi Zou, parts d, e written by Xinyi Ji, full quetion read by everyone)
a.



b. Search(x):
For each sorted array A_i of size 2^i in L (where $i = 0, 1, ..., k-1$ such that $b_i = 1$), call Search(A_i, x) (the search operation for sorted arrays). If x is found in any array A_i, which means x is currently in L, Search(x) returns TRUE immediately. Otherwise x is not in any array A_i, which means x is currently not in L, so Search(x) returns FALSE.

Worst-case time complexity:
Search(A_0, x) is $\mathcal{O}(\log 2^0)$ (since A_0 is a sorted array of size 2^0), Search(A_1, x) is $\mathcal{O}(\log 2^1)$ (since A_1 is a sorted array of size 2^1), ..., Search(A_{k-1}, x) is $\mathcal{O}(\log 2^{k-1})$ (since A_{k-1} is a sorted array of size 2^{k-1}).

$\log 2^0 + \log 2^1 + ... + \log 2^{k-1} = \log(2^0 \times 2^1 \times ... \times 2^{k-1}) = \log(2^{0+1+...+k-1})$
$= (0 + 1 + ... + k - 1) \cdot \log 2 = 0 + 1 + ... + k - 1 = \frac{(k-1)k}{2}$

Since $k = \mathcal{O}(\log n)$, then:
$\frac{(k-1)k}{2} = \frac{(\mathcal{O}(\log n)-1)(\mathcal{O}(\log n))}{2} = \frac{\mathcal{O}((\log n)^2 - \log n)}{2} = \frac{\mathcal{O}((\log n)^2)}{2} = \mathcal{O}((\log n)^2)$.

In the worst-case: we need to search for x in every array A_i of size 2^i in L (where $i = 0, 1, ..., k-1$ such that $b_i = 1$) (in this case $b_i = 1$ for each $i = 0, 1, ..., k-1$). So the total time used to determine whether x is in any array A_i is: $\mathcal{O}((\log n)^2)$. Also, returning TRUE/FALSE is $\mathcal{O}(1)$ and we only return once.
Therefore, it is now clear that there exists a constant $c > 0$ such that for all $n \geq 0$: for every input I of size n, Search(x) takes at most $c \cdot ((\log n)^2 + 1)$ time. So Search(x) is $\mathcal{O}((\log n)^2)$.

c. Insert(x) (assume x is not already in I):

1. Insert x into an empty array M. So now M is a sorted array of size 1.
2. Let $p = 0$. Note that the size of M is $2^p = 1$.
3. If A_p is empty: set M to be the new A_p of L (we can do so since M is a sorted array of size 2^p), and the algorithm ends.
   Otherwise: use the merge part of MergeSort to merge M with A_p (since both M and A_p are sorted arrays), and let Q be the resulting sorted array after the merge. So Q is a sorted array of size 2^{p+1} (since M is of size 2^p and A_p is of size 2^p). Then remove A_p from L.

4. Let $p = p + 1$, let $M = Q$. Now M is a sorted array of size 2^p. If $p = k$, let $A_k = M$, then add A_k to L and end the algorithm. Otherwise, do nothing.
5. Repeat steps 3-4 until the algorithm ends.

Worst-case time complexity:
Step 1: Creating a new sorted array of size 1 is $\mathcal{O}(1)$. Since step 1 runs only once, step 1 is thus $\mathcal{O}(1)$.
Step 2: Fixing the value of p is $\mathcal{O}(1)$. Since step 2 runs only once, step 2 is $\mathcal{O}(1)$.
In the worst-case, we need to do the merge part for each A_i of size 2^i in L (where $i = 0, 1, ..., k-1$ such that $b_i = 1$) (e.g. consider the case that $b_i = 1$ for each $i = 0, 1, ..., k-1$). In this case, steps 3-4 iterate k times and the algorithm ends on the k^{th} iteration where $p = k$.
Step 3: In the worst-case, the if statement in step 3 will never be true, so the worst-case time complexity of step 3 only depends on the total run time of all the merges + the total run time to remove all A_i from L. Use the merge part of MergeSort to merge two sorted arrays of size 2^i is $\mathcal{O}(2^i)$. In the worst-case, $i = 0, 1, \ldots, k-1$. Then $2^0 + 2^1 + \cdots + 2^{k-1} = \frac{1(1-2^k)}{1-2} = 2^k - 1$. Since $k = \mathcal{O}(\log n)$, $2^k - 1 = 2^{\mathcal{O}(\log n)} - 1 = \mathcal{O}(n-1) = \mathcal{O}(n)$. So the total run time of all the merges is $\mathcal{O}(n)$. Since removing any A_i from L takes constant time and step 3 runs $k = \mathcal{O}(\log n)$ times in the worst-case, the total run time to remove all A_i from L is $\mathcal{O}(\log n)$. Therefore, the worst-case time complexity of step 3 is $\mathcal{O}(n + \log n)$, which is $\mathcal{O}(n)$.
Step 4: : Defining p, M takes $\mathcal{O}(1)$ time. Checking if $p = k$ is $\mathcal{O}(1)$. Defining A_k is $\mathcal{O}(1)$. Adding A_k to L is $\mathcal{O}(1)$ since we have removed all A_i from L, which means L is empty (we only do the add operation when $p = k$). Since in the worst-case, step 4 runs $k = \mathcal{O}(\log n)$ times and we only do the addition part once, the worst-case time complexity of step 4 is $\mathcal{O}(3 \log n + 1)$, which is $\mathcal{O}(\log n)$.
For $n \geq 0$: $1 + 1 + n + \log n \leq n + n + n + n = 4n$
Therefore, it is now clear that there is a constant $c > 0$ such that for all $n \geq 0$: for every input I of size n, Insert(x) takes at most $c \cdot (4n)$ time. So Insert(x) is $\mathcal{O}(n)$.

d. The amortized time of an Insert operation is of $\mathcal{O}(\log(n))$, the proof is as of the following:

Aggregate analysis:
Since we use the Insert algorithm described in c, our execution of a sequence of n Inserts starting from an empty set is just like the increment of a binary number. That is, we need to create a sorted array of size 1 for every $2^0 = 1$ Insert, which takes $2^0 \times c'$ time(let the time of creation be constant c'), and we need to merge 2 sorted arrays of size $2^0 = 1$ for every $2^1 = 2$ Inserts. And since we use the merge part of MergeSort, this takes $2 \times 2^0 \times c = 2^1 \times c$ time (let the time of comparison between two nodes be c). We also need to merge 2 sorted arrays of size $2^1 = 2$ for every $2^2 = 4$ Inserts, and since we use merge part of MergeSort, this takes $2 \times 2^1 \times c = 2^2 \times c$ time, and so on... Lastly, we would need to merge 2 sorted arrays of size $2^{\log(n)-1}$ for every $2^{(\log n)}$ Inserts, and since we use the merge part of MergeSort, this takes $2 \times 2^{\log(n)-1} \times c = 2^{\log(n)} \times c$ time.
Hence the total time $T(n)$ is:
$\frac{n}{1} \times c' + \lfloor \frac{n}{2^1} \rfloor \times 2^1 \times c + \lfloor \frac{n}{2^2} \rfloor \times 2^2 \times c + ... + \lfloor \frac{n}{2^{\log(n)}} \rfloor \times 2^{\log(n)} \times c$
$= n \times c' + n \times c + n \times c + ... + n \times c = n \times c' + \log n \times n \times c.$
Hence $\frac{T(n)}{n} = c' + \log n \times c$, which is of $\mathcal{O}(\log n)$ (since we insert one node into the array and always create an array of size 1 which is the smallest at first, and do the Union operation starting from small arrays to larger ones, hence we don't need to change the position of the array in L).

Accounting method:
We stored $1 + \log n$ credits in each node when we insert it into I. We use the credits as follows: When the node is first inserted into I, we create a sorted array of size 1 to contain this node, and this operation takes 1 credit. And the rest of the credits are used when we merge the array that contains this node with another array of the same size. And I am going to show that for every node, the node goes through at most $\log n$ merge.

Let a be an arbitrary node we inserted into I. After n insertions, a must belong to a sorted array of size no more than n, let this array be b. And for every sorted array, each node in it goes through log(the size of the array) merge operations to finally result this array. Hence a has experienced $\log b$ merge operations, which is at most $\log n$. Hence, every node experiences at most $\log n$ merge operations (since we insert one node into the array and always create an array of size 1 which is the smallest at first, and and do the Union operation starting from small arrays to larger ones, hence we don't need to change the position of the array in L).
And $1 + \log n$ is of $O(\log n)$.

e. The algorithm and the explanation of the run time of Delete(x) is as of following:
First, we split the array that contains x (let the size of this array be a, we know that $a < n$) into two arrays of the same size by splitting the original array from the middle. Ex: $1- > 2- > 3- > 4$, we split this array into $1- > 2$ and $3- > 4$. Do the same operation to the one of the two arrays which contains x, and we do this again and again until we have x in an array of size 1. This takes $\log a$ time, which is of $\mathcal{O}(\log n)$.
Second, we delete x, by omitting the array of size 1 which it belongs to. This takes constant time which is of $\mathcal{O}(1)$.
Third, we perform Union and merge the rest of the arrays from smaller same size arrays to bigger same size arrays. First, by the way we split the array which x is contained in, we know that the set of the small arrays formed in step 1 that do not contain x is in the form of L. And this set of arrays contains $a-1$ nodes which is smaller than n. And the biggest array in this set is of size at most $\frac{n}{2}$. Second, the set of the original arrays that do not contain x is also of form L, and this set of arrays contains $n-a$ nodes which is smaller than n. And the biggest array in this set is of size at most $\frac{n}{2}$. Hence we merge these two sets of arrays by merging same size arrays, starting from the smallest ones. And this takes at most $2+4+..+2^{(\log n)} = \frac{2\cdot(1-2^{(\log n)})}{(1-2)} = (n-1)\times 2 = 2n-2$ steps, which is of $\mathcal{O}(n)$ (supposing that we have 2 same size arrays for each size from size 1 to size $n/2$. Obviously we cannot have two arrays of size $\geq n$ and we only merge the arrays of one size one time. Since we just have two arrays of the same size at the beginning, even after all the merge operations, we cannot find a fourth array of the same size).
Hence the worst-case run time of Delete is $\mathcal{O}(\log n) + \mathcal{O}(n) + \mathcal{O}(1)$ which is of $\mathcal{O}(n)$.

2 Problem 2

(Written by Angela Zhu, read by Zhuozi)

a. Furio is using BFS (Breadth First Search) which we learned in class.
Since the number of houses is constant, let this number be c. Furio can simply use a loop to go though all the houses, and for each house s, call $BFS(G, s)$. For each hospital v discovered when doing BFS, we know from the theorem learned in class that $d[v]$ is the shortest distance to reach from s to v. Having this in mind, the first hospital h that is discovered is the nearest hospital to the house s, since we know that every hospital v that is discovered later than h must have $d[v] \geq d[h]$ (as described in lemma 1 learned in class). Therefore the shortest time to reach a hospital from a house s must be $d[h]$
Since the worst-case runtime of BFS is $\mathcal{O}(|V| + |E|)$, doing BFS for all the houses which is a total of c times takes $c \cdot \mathcal{O}(|V| + |E|)$ steps. Since c is constant, the worst-case runtime of this algorithm that solves $\mathcal{P}$ is still $\mathcal{O}(|V| + |E|)$.

b. Paulie also uses BFS to solve the problem. Instead of looping through all the hosues, BFS is performed once for each of the k hospitals.
For each house s, let $s.time$ be the current shortest time to reach a hospital vertex. Assume this value to be ∞ at the beginning of the algorithm. For each hospital h, call $BFS(G, h)$. Upon discovery of a house s, compare $d[s]$ with $s.time$ (we know from the theorem learned in class

that $d[s]$ is the shortest distance to reach house s from the current hospital). If $d[s] < s.time$, let $s.time = d[s]$. This occurs when the current hospital h is the first hospital of the loop, or that the shortest distance between s and h is less than the shortest distance between s and all previous hospitals. After looping through all the hospitals, for each house s, $s.time$ should store the shortest time to reach any hospital from s.
Since the worst-case runtime of BFS is $\mathcal{O}(|V| + |E|)$, doing BFS for all the hospitals which is a total of k times takes $k \cdot \mathcal{O}(|V| + |E|)$ steps. Since k is not a constant, the worst-case runtime of this algorithm that solves $\mathcal{P}$ is $\mathcal{O}(k(|V| + |E|))$.

c. Tony is right. Tony's algorithm is similar to a normal BFS, but with the following changes:

1. Before starting to explore the vertices (as of the BFS code, this is before the loop), instead of enqueuing one single vertex into Q, enqueue all hospital vertices into Q. This means that Q should contain k grey vertices before exploring.

2. For each house s, let $s.time$ denote the shortest time to reach a hospital. This field is updated upon first discovery of a house s. If $p[s]$ is a hospital, then $s.time = 1$, otherwise, $s.time = p[s].time + 1$.

This results in the effect of exploring all k hospital vertices at the same time, by doing BFS only once. Since the algorithm is "first discovered first explored", the exploration order in this algorithm can be seen as:
Explore all the k hospitals and discover all houses that are within distance 1 of any hospital.
Explore all the houses that are distance 1, and discover all house that are within distance 1 of these explored houses, which means they can reach a hospital within distance 2.
And the algorithm continues until all houses are explored. This ensures that upon first discovery of a house s, the field $s.time$ as described above must be the shortest time to reach a hospital.

Time Complexity:
The worst-case time complexity of this algorithm is $\mathcal{O}(|V| + |E|)$.
As described above, the algorithm has 2 major differences from a regular BFS, with maybe some small extra operations (assignment, arithmetic, etc.) that take constant time. We know that the time complexity of a regular BFS is $\mathcal{O}(|V| + |E|)$. For the two differences:

1. All k hospital vertices are enqueued into Q. Each ENQ operation takes constant time, therefore this step takes $\mathcal{O}(k)$ time.
2. This is just assigning value to the variable $s.time$ upon discovery of a house s, which takes constant time and happens at the same time as the assignment of $d[s]$ in a regular BFS. Therefore this step does not affect the time complexity of the algorithm.

Therefore, the worst-case time complexity of this algorithm would be $\mathcal{O}(|V|+|E|)+\mathcal{O}(k)$, which is $\mathcal{O}(k + |V| + |E|)$.
Let n denote the number of houses. Then the number of vertices $|V| = n + k$. Then the time complexity of this algorithm can be written as:
$\mathcal{O}(k + |V| + |E|) = \mathcal{O}(k + k + n + |E|) = \mathcal{O}(2k + n + |E|)$.
Since $2k + n$ is of $\mathcal{O}(k + n)$, the time complexity of this algorithm is therefore of $\mathcal{O}(k + n + |E|)$, which is $\mathcal{O}(|V| + |E|)$.