Question 1.

Zijin Zhang wrote the solution and Quan Xu read this solution to verify its clarity and correctness.

(a)

set $I_1 = \{6, 8, 4, 13, 9\}$
data structure L_1 for I_1: $[6] \leftrightarrow [\ ] \leftrightarrow [4, 8, 9, 13]$
set $I_2 = \{21, 12, 7, 14, 5, 16, 10\}$
data structure L_2 for I_2: $[21] \leftrightarrow [7, 12] \leftrightarrow [5, 10, 14, 16]$

(b)

Assume n be the size of I and $n \in \mathbb{N}$
To perform a $SEARCH(x)$ operation with this data structure, we will begin with searching each individual array in the double linked list structure. Since each array insides the double linked list structure is sorted, we can use *binary search* to search for x.

Now, consider the worst-case running time for the $SEARCH(x)$ operation:
- the worst-case running time for a *binary search* on array A_i with length of 2^i is $\mathcal{O}(log_2 2^i)$.
- In worst-case scenario, assume every array$(A_{k-1}, A_{k-2}, ..., A_2, A_1)$ in the double linked list are full and x is not in this data structure. So we will have at most k (since $k = \mathcal{O}(\log_2 n)$ as given) arrays to search.

$$\begin{aligned}
T(n) &= \mathcal{O}(\Sigma_{i=0}^{k-1} \log_2 2^i) \\
&= \mathcal{O}(\Sigma_{i=0}^{k-1} i) \\
&= \mathcal{O}(\frac{k(k-1)}{2}) \\
&= \mathcal{O}(\frac{k^2 - k}{2}) \\
&\in \mathcal{O}(k^2) \qquad \#\ \mathrm{k} \geq 0 \\
&= \mathcal{O}((\log_2 n)^2) \qquad \#\ \mathrm{k} = \mathcal{O}(\log_2 n) \\
&= \mathcal{O}(\log^2 n)
\end{aligned}$$

So the worst-case running time for $SEARCH(x)$ operation is $\mathcal{O}(\log^2 n)$.

(c)

Assume n be the size of I and $n \in \mathbb{N}$
To perform a $INSERT(x)$ operation with this data structure, we will use an algorithm that is similar to the $Insert$ operation of Binomial Heap.
1. Create a new array A' of size 1 that contains the new element x.
2. Check if A_0 is empty. If A_0 is empty, the newly created array A' will be the new A_0. If A_0 is not empty, we use the Merge part of MergeSort to merge A_0 with A' into a new sorted array A'_1, and make A_0 be empty array.
3. Repeat Step 2 for all A_i *s.t.* $i \in \{1, 2, ..., k-1\}$. If A_i is not empty, we repeat Step 2 for crating A'_{i+1} after merging arrays. And set A_i be empty array. If A_i is empty, we fills A_i and stop.

Now, consider the worst-case running time for the $INSERT(x)$ operation:
- Create a new array of size 1 takes constant time $\mathcal{O}(1)$
- For the worst-case scenario, we assume that each array in the double linked list is full. So Step 2 will be repeated for at most k times. Each merge by the Merge part of MergeSort operation for two array of size m

will takes at most $2m$ times. So

$$\begin{aligned} T(n) &= 2 \times \Sigma_{i=0}^{k-1} 2^i \\ &= 2 \times (2^k - 1) \\ &= 2(n-1) \\ &= 2n - 2 \\ &\in \mathcal{O}(n) \end{aligned}$$

So the worst-case running time for $INSERT(x)$ operation is $\mathcal{O}(n)$.

(d)

Assume n be the size of I and $n \in \mathbb{N}$
Determine amortized time of an $INSERT$ operation by Accounting Method.
According to the algorithm we described in part (c), in total of k times of merge will be made in a single $INSERT$ operation in worst-case scenario. We know that each merge by merge sort operation for two array of size m will takes at most $2m$ times. So each element in the two array will take constant time $\mathcal{O}(1)$ in a merge operation. Therefore, a single element will take at most $k \times \mathcal{O}(1) = \mathcal{O}(k) \in \mathcal{O}(\log n)$ in worst-case scenario.
- Thus, the amortized time of an $INSERT$ operation by Accounting Method $\in \mathcal{O}(logn)$

Determine amortized time of an $INSERT$ operation by Aggregate Analysis.
We compute worst case cost of n insertions from empty set:
- Calculate the merge cost for each array:

1. First insertion fills the A_0 array, and the second insertion all a merge on A_0 with the newly insertion element(length 1). So for every twice insertion, A_0 has a merge call. And the cost of the merge call for A_0 is $2 * 1$. In general, the merge cost would be 2m where m is the length of the array.

2. For A_1, merge call costs $2 * 2^1 = 2^2$.

3. For A_2, merge call costs $2 * 2^2 = 2^3$.

... ... (continues)

- So the cost for each array A_i, where $i = 0, 1, ..., k-1$, is $2m = 2 * 2^i = 2^{i+1}$ (# Every array, A_i has length 2^i)

- calculate the number of merge call on each array:

1. First insertion fills the A_0 array, and the second insertion all a merge on A_0 with the newly insertion element(length 1). So for every twice insertion, A_0 has a merge call.

2. Merge call would be called on A_1 every four times insertion.

3. Merge call would be called on A_2 every eight times insertion.

... ... (continues)

- So the number of calling Merge call for each array, A_i, where $i = 0, 1, 2, ..., k-1$, is $\lfloor \frac{n}{2^{i+1}} \rfloor$

So the total cost of insertion is:

$$\begin{aligned} T(n) &= \Sigma_{i=0}^{k-1}\ (2^{i+1})(\left\lfloor \frac{n}{2^{i+1}} \right\rfloor) \\ &\leq \Sigma_{i=0}^{k-1}\ (2^{i+1})(\frac{n}{2^{i+1}}) \\ &= \Sigma_{i=0}^{k-1}\ n \\ &\leq n(k-1) \\ &\leq n((alogn + C) - 1) \qquad a, C \in \mathbb{R},\ k \in \mathcal{O}(logn) \end{aligned}$$

- So the cost is $\frac{T(n)}{n} \in \mathcal{O}(logn)$
- Thus, the amortized time of an $INSERT$ operation by Aggregate Analysis $\in \mathcal{O}(logn)$

(e)

Assume n be the size of I and $n \in \mathbb{N}$
- To perform a $DELETE(x)$ operation with this data structure, we first find the position of x by using the $SEARCH(x)$ operation we defined in part (b). Assume we found x in array A_i. Then we want to find the array A_m that is full and has the least amount of elements among all array in our data structure.
- Then find the last element of A_m and denoted as y. We delete x in A_i and insert y into the correct position of A_i. And then delete y from A_m.

Now, the array A_i still have 2^i elements but A_m has $2^m - 1$ elements. So we split A_m into another m arrays, $A_0, A_1, ..., A_{m-1}$, simply by put the first element of A_m into A_0, next two element into A_1 and so on. We can do this without any merge or merge sort since the original $A_0, A_1, ..., A_{m-1}$ are all empty because A_m was sorted, and A_m still sorted after removing the last element.

After this operation, all m arrays $A_0, A_1, ..., A_{m-1}$ will be full because $\Sigma_{k=0}^{m-1} 2^k = 2^k - 1$. After reorganizing, A_m will be empty now.

Consider the worst-case running time for the $DELETE(x)$ operation:

- In order to find x, using the $SEARCH(x)$ operation will take $\mathcal{O}(\log^2 n)$ in worst-case scenario to find x as we proved in part (b). Then we can locate which array that x is located. Let's say A_i

- Finding the full array, denote as A_m, that contains least number of elements, takes $\in \mathcal{O}(k-1) \in \mathcal{O}(logn)$, and find y, which is the largest element in A_m takes $\mathcal{O}(1)$, since we know the size of A_m and y is the last element in this array. Delete x, y will take constant time $\mathcal{O}(1)$. Insert y in to the correct position of A_i will takes $\mathcal{O}(2^i) = \mathcal{O}(2^{k-1}) \in \mathcal{O}(n)$ since there are total 2^i element in A_i.

- The operation that split A_m takes $\mathcal{O}(2^m - 1) \in \mathcal{O}(2^m) \in \mathcal{O}(2^{k-1}) \in \mathcal{O}(n)$. Since there are total $2^m - 1$ elements remains in A_m, we have to move all $2^m - 1$ elements into those empty arrays $(A_{m-1}, A_{m-2}, ..., A_1, A_0)$ and each move operation will take constant time $\mathcal{O}(1)$. We don not need to use merge operation since all elements in A_m are sorted. For example, the first element in A_m move to A_0, and the second and third element move to A_1 as ordered, and so on. So we move $2^m - 1 \in \mathcal{O}(2^{k-1}) \in \mathcal{O}(n)$ elements to fill those empty arrays, $A_0, A_1, ..., A_{m-1}$.

Therefore, the worst-case running time of $DELETE(x)$ operation will be $\mathcal{O}(n)$.

Question 2.

Quan Xu wrote the solution and Zijin Zhang read this solution to verify its clarity and correctness.

(a)

We want to use Breadth First Search to solve this problem. Let $C \in \mathbb{R}$
We assume that the number of houses is a constant, C. Also, we have a set V that contains all vertex of graph G and a set E that contains all edges of graph G.

- We augment the house vertices: add one more attribute *shortest_time* to each vertex that represent the shortest time to reach the hospital vertex.

- We also augment $BFS(G, S_{house})$: From the original $BFS(G, S)$ algorithm, we have distance, $d(v)$, which is the length of the discovery path to v from S. Since each edge can be traversed in one unit of time, $d(v) \times 1 = d(v)$ is the reach time of v from the starting vertex S. So for each S_{house}, if *shortest_time* of S is NIL, we set the attribute *shortest_time* as the $d(v_{hosp})$ where v_{hosp} is the hospital vertex. When a house vertex has shortest_time which means the vertex is already reach a hospital, V_{hosp1}. And the value of shortest_time is the actually shortest reach time form S_{house} to some hospital. If we continues discovering vertex, we might reach another hospital, V_{hosp2}. But $d(V_{hosp1}) < d(V_{hosp2})$ by Lamma 1 in lecture. So when we reach the first hospital vertex, we should stop discovering for saving time. All the assignment take constant time $\mathcal{O}(1)$. So adding these extra steps will not effect the original worst-case running time of BFS. Therefore, the worst-case running time is still $\mathcal{O}(|V| + |E|)$.

Algorithm:

In order to solve problem P, we will call the augmented BFS on each house vertex. Since the number of houses is constant C, the total running time of this in worst-case scenario is $C \times \mathcal{O}(|V| + |E|) \in \mathcal{O}(|V| + |E|)$.

(b)

We want to use Breadth First Search to solve this problem.
- We augment the house vertices: add one more attribute *shortest_time* to each vertex that represent the shortest time to reach the hospital vertex.
- We also augment $BFS(G, S_{hosp})$:

1. From the original $BFS(G, S)$ algorithm, we have distance, $d(v)$, which is the length of the discovery path to v from S.

2. Starting at S_{hosp}, for every house vertex v_{house} that is discovered, we set the *shortest_time* of v_{house} as $d(v_{house})$ from S_{hosp} if *shortest_time* of v_{house} is NIL. If *shortest_time* of v_{house} is not NIL, then we compare the $d(v_{house})$ with *shortest_time* and set *shortest_time* to be $Min(shortest\_time,\ d(v_{house}))$.

3. Starting at S_{hosp}, if we reach another hospital vertex, V_{hosp2}, we do not need to discover new vertex from the hospital vertex, V_{hosp2}, since the newly discovered hospital vertex, V_{hosp2}, must have a shorter reach time than S_{hosp} to a house vertex that will be discovered later. We will explain this more specifically in following two cases.

Case 1: There is no more house vertex to be discovered after V_{hosp2}. If there is no house vertex can be discovered from the hospital vertex V_{hosp2}, then we do not need to continue discovering from the hospital vertex.

Case 2: If there is at least one house vertex can be discovered after the hospital vertex, V_{hosp2}. The reach time from S_{hosp} to that house vertex will be the sum of the reach time from S_{hosp} to the newly discovered hospital vertex, V_{hosp2}, and the reach time from the newly discovered hospital vertex, V_{hosp2}, to that house vertex. Apparently, the *shortest_time* should be the reach time from the newly discovered hospital vertex, V_{hosp2}, to that house vertex. So we should stop discovering new vertex from that hospital vertex and continue with all other vertices on the *queue*.

Since all the value comparison and assignment take constant time $\in \mathcal{O}(1)$, adding these extra steps will not effect the original worst-case running time of BFS. So the worst-case running time is still $\mathcal{O}(|V|+|E|)$.

Algorithm:

- Making an empty set, *S_hosp*, and looping over all vertices from S. Then adding all k hospital vertices into *S_hosp*. Making an empty set takes $\mathcal{O}(1)$, and adding into *S_hosp* takes $\mathcal{O}(1)$. So total run-time in worst-case of this step is $\mathcal{O}(|V|)$

- We will loop over the set, *S_hosp*, and call augmented BFS(G, V_{hosp}) on each hospital vertex. Looping over all vertices in *S_hosp* takes $\in \mathcal{O}(k) \in \mathcal{O}(|V|)$ and there are k hospital vertices. So the total running time will be $k \times \mathcal{O}(|V| + |E|) \in \mathcal{O}(k(|V| + |E|)$ in worst-case scenario.

- Therefore worst case running time is $\in \mathcal{O}(k(|V| + |E|)$.

(c)

We use BFS to solve it. Let $a, c, C \in \mathbb{R}$
We add one attributes to each house vertex, which called 'shortest_time'

We augments the BFS(G, S) as $BFS(G,\ S_{set\_hosp})$:

1. S_{set_hosp} is a set of all hospital vertices.

2. We let the second parameter of BFS as a set, which means that we set all hospital vertices as level 1 (putting all the elements of S_{set_hosp} in the 'queue' of BFS initially) instead of setting only one node in the queue initially. And all the vertex in the set should have BFS_COLOUR grey, since they are in the queue now.

3. The house vertex that discovered by a hospital vertex has on edge distance to a hospital vertex. And traversed each edge in one unit time. So these house vertices(discovered directly by a hospital vertex) has shortest_time as 1. Also, for other house vertices, the shortest_time = $1 + shortest\_time\ of\ its\ parent$. When a vertex is discovered, its BFS_COLOUR is grey. And when a vertex's neighbours are all discovered, the vertex's BFS_COLOUR is Black and the vertex is explored.

4. When a vertex is discovered(BFS_COLOUR is grey) or explored(BFS_COLOUR is black), it would not be added into the 'queue' of BFS again. So each vertex would be visited once only, and each edge would also be visited once only.

5. Assigning value to 'shortest_time', and calculate the 'shortest_time' by addition would take $\mathcal{O}(1)$ time.

6. Thus, the worst-case run time of augmented BST is still $\mathcal{O}(|V| + |E|)$

Algorithm:

1. Making an empty set, S_{set_hosp}, and looping over all vertices from V. Then adding all hospital vertices into S_{set_hosp}. Making an empty set takes $\mathcal{O}(1)$, and adding each element into S_{set_hosp} takes $\mathcal{O}(1)$. Looping over all vertices take $\mathcal{O}(|V|)$. So total run-time in worst-case of this step is $\mathcal{O}(|V|)$

2. Calling augmented BFS(G, S_{set_hosp}) on set, S_{set_hosp}, then we can update the shortest reach time to some hospital for each house vertex. Notice that, we only call augmented BFS(G, S_{set_hosp}) once, and the second parameter is a set. This step takes $\mathcal{O}(|V| + |E|)$ as shown above.

3. So in worst-case running time in total is $a|V| + c(|V| + |E|) + C \in \mathcal{O}(|V| + |E|)$ as what we want.