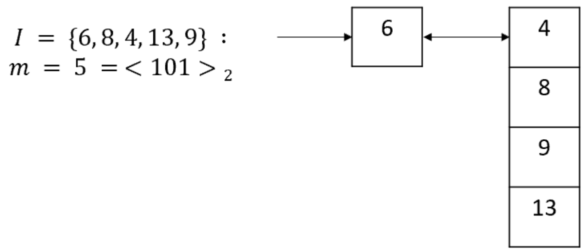
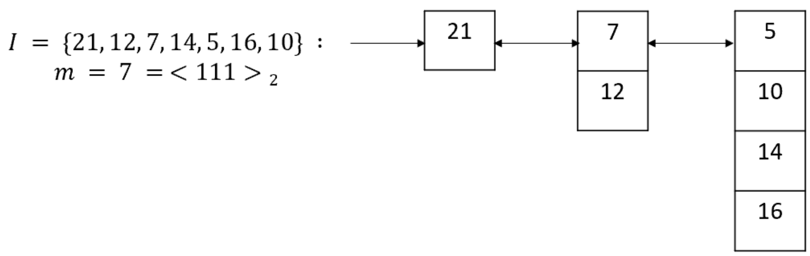
Solutions for Homework Assignment #5

Answer to Question 1.

a.





b. To do a SEARCH(x), one performs a binary search separately on each array of L until either x is found in some array, or all arrays have been considered and x is not found.

The worst-case time complexity of this SEARCH algorithm is $O(\log^2 n)$. To see this, note that if I contains n elements, there are $O(\log n)$ arrays: one array for each "1" digit in binary representation of n (this is similar to the $O(\log n)$ S_k trees that exist in a binomial heap with n elements). Moreover, the largest array contains at most n elements, and so the binary search of any array takes at most $O(\log n)$ time. Since the algorithm performs at most one binary search on each array, its worst-case time complexity is $O(\log^2 n)$.

Note that for an infinite number of values of n, the worst-case time complexity of the SEARCH algorithm is also $\Omega(\log^2 n)$. To see this, suppose that $n = 2^k - 1$ and we do SEARCH(x) for $x \notin I$. In this case, the list contains k arrays $A_0, A_1, \ldots, A_{k-1}$, where A_j has 2^j elements, and one must do a binary search in every array. This takes at least $\Omega((k-1) + (k-2) + \cdots + 2 + 1) = \Omega(k^2) = \Omega(\log^2 n)$ time.

Thus, for an infinite number of values of n, the worst-case time of the SEARCH algorithm is $\Theta(\log^2 n)$.

c. To do INSERT(x), one performs following algorithm:

```
(a) create a new array of size 1 containing x
(b) insert this new array at the beginning of the list L
(c) while L contains 2 arrays of the same size:
        merge the 2 sorted arrays into one sorted array of twice the size.
        To do each merging use a procedure similar to the one used in Mergesort.
```

In the worst case, $n = 2^k - 1$, the list L contains k arrays $A_0, A_1, \ldots, A_{k-1}$, where A_j has 2^j elements, and the INSERT(x) algorithm will merge all the arrays as follows: x with A_0, the resulting array (of size 2) with A_1, the resulting array (of size 4) with A_2, etc.

Merging x with A_0 takes at most 2 operations, merging the result with A_1 takes at most 4 operations, merging the result with A_2, takes at most 8 operations, and so on. So the total time taken is proportional to $2+4+8+\ldots+n < 2^1 + 2^2 + \ldots + 2^k = 2(2^k - 1) = 2n$, i.e., it is $O(n)$.

d. Aggregate analysis: From part (c), it is clear that to insert an element in a set I with n elements costs at most $O(2^r)$, where r is the position of the first 0 digit in the binary representation of n: this is because the linked list representing I contains arrays $A_0, A_1, \ldots, A_{r-1}$ but does not contain array A_r (where each A_j has 2^j elements), so the merging of arrays caused by an insertion stops when this merging creates A_r.

Note that when we start from an empty set I and we successively insert the n elements one by one we have (this is similar to the binary counter example that we did in class):

- $r = 0$ occurs at most $\lceil n/2 \rceil$ times, $r = 1$ occurs at most $\lceil n/4 \rceil$ times, and so on.
- $r = 0$ occurs at least $\lfloor n/2 \rfloor$ times, $r = 1$ occurs at least $\lfloor n/4 \rfloor$ times, and so on.

In fact, it turns out that $r = 0$ occurs exactly $\lfloor n/2 + 1/2 \rfloor$ times, $r = 1$ occurs exactly $\lfloor n/4 + 1/2 \rfloor$ times, and so on. Thus, the total cost is at most $O(1 \cdot n/2 + 2 \cdot n/4 + 4 \cdot n/8 + \ldots) = O(n \log n)$. The amortized cost per insertion is the total cost divided n, so it is $O(\log n)$.

Accounting method: We now switch our viewpoint to consider each element separately. Consider an element that is being inserted. At the moment of insertion, we (over) charge $\log n$. As more elements are being inserted, our element will be moved to other arrays. Every time our element is moved, it cost us 1 unit, and we pay 1 from the account of this element. Note that during mergesorts, our element is moved only to arrays that double in size. So our element cannot move more than $\log n$ times. Thus, the initial charge of $\log n$ for inserting this element is sufficient to cover all the costs of moving that element during the entire sequence of inserts. Since there are n elements, the total charge over all elements is $O(n \log n)$. Dividing by the number of operations, which is n, gives $O(\log n)$ amortized cost per insert operation.

e. The $\textsc{Delete}(x)$ algorithm works as follows. Assume x is in array A_s. Let A_r be the *smallest array* in the structure (so $r \leq s$).

If $r = s$, we first remove x from A_r; then we split the remaining $2^r - 1$ elements of A_r into (sorted) arrays $A_0, A_1, \ldots, A_{r-1}$ of sizes $1, 2, 4, \ldots, 2^{r-1}$, and enter these arrays in the linked list (after removing A_r).

If $r < s$, we first remove x from A_s; then, we pick the first element of A_r and insert it in A_s in a way that keeps A_s sorted (e.g., use binary search); finally, we split the remaining $2^r - 1$ elements of A_r into (sorted) arrays $A_0, A_1, \ldots, A_{r-1}$ of sizes $1, 2, 4, \ldots, 2^{r-1}$, and enter these arrays in the linked list (after removing A_r).

Since array A_r is already sorted, the splitting of A_r into the sorted arrays $A_0, A_1, \ldots, A_{r-1}$ can be done in at most $O(n)$ time. To insert an element into A_s while keeping A_s sorted also takes at most $O(n)$ time. So the worst-case time complexity of the above $\textsc{Delete}(x)$ algorithm is $O(n)$.

Answer to Question 2. In the following, we define the *distance* between two vertices u and v in an undirected graph G, denoted $\delta_G(u, v)$, to be the length of the *shortest path* between u and v in G. In our question, we assumed that each edge of G can be traversed in one unit of time, so the shortest time to reach a vertex v from a vertex u is simply $\delta_G(u, v)$.

Here our goal is to find for each house vertex u, the *shortest* distance between u and some hospital vertex of G. In other words, for each house vertex u, we want to compute $\min_{h \in H} \delta_G(u, h)$, where H is the set of all hospitals in G.

To do so, we use Breadth-First Search (BFS). Recall that a BFS on a graph G starting at a vertex s, computes for each node u an attribute $d(u)$, such that at the end of the BFS $d(u) = \delta_G(s, u)$ (*).

In every algorithm below, each node u has an attribute $\mathit{shortestDistance}(u)$ such that, at the end of the algorithm, $\mathit{shortestDistance}(u) = \min_{h \in H} \delta_G(u, h)$, i.e., it is the shortest distance between u and some hospital, as wanted.

a. Furio's algorithm for solving problem $\mathcal{P}$ uses BFS in the following simple way. For each house vertex u, do a BFS starting at u, until you discover the *first* hospital vertex h, upon which you set $\mathit{shortestDistance}(u) = d(h)$ and terminate the search.

Using (*), it is not difficult to see that, for each vertex u, $\mathit{shortestDistance}(u)$ contains the shortest distance between u and some hospital.

If c is the number of houses in the graph, then we do c BFS searches on the graph. Hence, the worst-case time complexity of this algorithm is $O(c(|V| + |E|))$, which is simply $O(|V| + |E|)$, since c is assumed to be a constant.

b. Paulie's algorithm for solving problem $\mathcal{P}$ uses BFS in a slightly more clever way. Instead of doing a BFS from each house vertex, we do a BFS from each hospital vertex, as shown below:

1. For every house vertex u, $shortestDistance(u)$ is initialized to ∞.

2. Repeat the following for each hospital vertex h:

   (a) Do a BFS starting from vertex h.

   (b) For each house vertex u, if $d(u) < shortestDistance(u)$, then set $shortestDistance(u) = d(u)$.

From (*), after the BFS starting from a hospital vertex h, for each house vertex u, we have $d(u) = \delta_G(h, u)$, in other words $d(u)$ is the distance between u and h. Thus, we have the following repeat loop invariant (at the end of the loop): for each house vertex u, we have $shortestDistance(u)$ is the shortest distance between u and all the hospitals from which we have done a BFS so far, i.e., $shortestDistance(u) = \min_{h \in H'} \delta_G(u, h)$, where H' is the set of all the hospitals from which we have done a BFS so far. Since we do a BFS from every hospital in G, at the end of the algorithm $H' = H$ and $shortestDistance(u)$ is the shortest distance between u and some hospital h in G.

If k is the number of hospitals in the graph, then we do k BFS searches on the graph. Hence, the worst-case time complexity of this algorithm is $O(k(|V| + |E|))$.

c. Tony, being the csc263 instructor, is **always** right. Tony's algorithm improves Paulie's algorithm in the following way. Instead of doing k BFS searches *sequentially* (one starting from each hospital vertex), do all of them *concurrently*, in an *interleaved* way: First visit all the houses that are at distance 1 from any hospital vertex, then visit all the houses that are at distance 2 from any hospital vertex, then visit all the houses that are at distance 3 from any hospital vertex, and so on. The following algorithm does this in a clean and efficient way :

1. Add a new vertex s to the graph G. Let $V' = V \cup \{s\}$.

2. Connect s to each hospital vertex of G. That is, for each hospital vertex h, add the edge (s, h) to the graph.

   Let E' denote the union of the set E with the set of these additional edges, and let $G' = (V', E')$.

3. Do a BFS on G' starting at vertex s.

   At the end of this BFS, for every house vertex u, we have $d(u) = \delta_{G'}(s, u)$ (the length of the shortest path between this house u and the new node s).

4. For each house vertex u, set $shortestDistance(u) = d(u) - 1$.

Since we do a single BFS search on the graph G', the worst-case time complexity of the algorithm is $O(|V'| + |E'|)$. Note that $|V'| = |V| + 1$, and $|E'| \le |E| + |V|$. Hence, the worst-case time complexity of the algorithm is $O(|V| + |E|)$.

To prove the algorithm's correctness (the question did *not* ask for this proof) we first relate the shortest distance between a house u and some hospital in G (i.e., $\min_{h \in H} \delta_G(u, h)$) to the distance between house u and the newly added vertex s of G' (i.e., $\delta_{G'}(u, s)$):

Theorem. *For every house vertex u of G, $\min_{h \in H} \delta_G(u, h) = \delta_{G'}(u, s) - 1$.*

Proof. Note that, since G is connected and it has at least one hospital h, G' is also connected (do you see why?).

Let u be any vertex of G.

- First, we prove $\min_{h \in H} \delta_G(u, h) \le \delta_{G'}(u, s) - 1$. Since G' is connected, by definition of $\delta_{G'}(u, s)$, G' has a shortest path P' between u and s of length $\delta_{G'}(u, s)$. Since s is connected *only* to hospital vertices in G', the path P' between u and s in G' is of the form $P' = u - \ldots - h' - s$, where $h' \in H$. So G' has a path $P = u - \ldots - h'$ of length $\delta_{G'}(u, s) - 1$ between u and h'. Note that s is *not* in P (because P' is a *shortest* path between u and s). Thus the path P is also in G. So G has a path between u and hospital $h' \in H$ of length $\delta_{G'}(u, s) - 1$. Therefore $\min_{h \in H} \delta_G(u, h) \le \delta_G(u, h') \le \delta_{G'}(u, s) - 1$.

- Next, we prove $\min_{h \in H} \delta_G(u,h) \geq \delta_{G'}(u,s) - 1$. Suppose, for contradiction, that $\min_{h \in H} \delta_G(u,h) < \delta_{G'}(u,s) - 1$. Then there is some $h' \in H$ such that $\delta_G(u,h') < \delta_{G'}(u,s) - 1$. Since G is connected, by the definition of $\delta_G(u,h')$, G has a shortest path $P = u - \ldots - h'$ between u and h' of length $\delta_G(u,h')$. Since s is connected to h' in G', this implies that G' has a path $P' = u - \ldots - h' - s$ between u and s of length $\delta_G(u,h') + 1$. Since $\delta_G(u,h') < \delta_{G'}(u,s) - 1$, we have $\delta_G(u,h') + 1 < \delta_{G'}(u,s)$. So the length $\delta_G(u,h') + 1$ of the path P' between u and s in G' is shorter than the length $\delta_{G'}(u,s)$ of the shortest path between u and s in G' — contradiction.

$\square$

Note that the algorithm performs a BFS of G' starting from s. At the end of this BFS, for each house vertex u, we have $d(u) = \delta_{G'}(u,s)$. So by above theorem, $d(u) - 1 = \delta_{G'}(u,s) - 1 = \min_{h \in H} \delta_G(u,h)$. Thus, for each house vertex u, the algorithm sets $shortestDistance(u) = d(u) - 1 = \min_{h \in H} \delta_G(u,h)$. In other words, the algorithms sets $shortestDistance(u)$ to the shortest distance between house u and some hospital, as wanted.