HOMEWORK 2

Ritika Kumawat
A20414073

Problem 1:

Soln:

Let G is undirected graph where one of those vertices is referred as source s and each of whose edges (u, v) has a non-negative length l(u, v). The minimum-length path from u to v is the shortest path.

Dijkstra's algorithm solves the shortest path problem using a tentative distance function d from vertices to real numbers with below properties:

1) For any vertex u, d(u) is finite and there is a path from s to u of length d(u).
2) When the algorithm terminates, d(u) is the distance from s to u.

Initially d(s) = 0 and d(u) = ∞ for u ≠ s

During the run of this algorithm, each vertex is in one of three states: unlabeled, labeled, or scanned.

Initially s is labeled and all other vertices are unlabeled. The algorithm consists of repeating the following steps until there are no labeled vertices.

Algorithm:

Assumptions:

$(G = (V, E, length); s \in V)$

dist : array[V] of integers, initialized to ∞

H : empty minheap

A : array[V] of vertices, initialized to null

```
for all vertices u ∈ V
    {
        H.insert(u, ∞)    // It will insert u with key ∞
    }
H.decreasekey(s, 0)
dist[s] := 0
while (H ≠ ϕ)
{
    (u, u.key) := H.deletemin()
    for (u, v) ∈ E
        {
            if dist[v] > dist[u] + l(u, v)
                {
                    A[v] := u
                    dist[v] := dist[v] + l(u, v)
                    H.decreasekey(v, dist[u] + l(u, v))
                }
        }
} return (dist, A)
```

In this algorithm, there are total 1 insert, $|V|$ deletemin and $|E|$ decreasekey operations.

The time complexity for insert operation will be $O(1)$

The time-complexity for 1 deletemin $= O(\log|V|)$

$\therefore$ time complexity for V deletemin $= O(|V|\log|V|)$

The time-complexity of 1 decreasekey $= O(1)$

$\therefore$ time-complexity for E decreasekey $= O(|E|)$

$\therefore$ Total time complexity

$$= O(1) + O(|V|\log|V| + O(|E|)$$

$$= O(|V|\log|V|) + O(|E|)$$

$$= O(|V|\log|V| + |E|)$$

So, the algorithm satisfies the time complexity that we needed for the algorithm ask in the

question which will produce a least-inductivity vertex ordering of G.

Problem 2:

Soln:

If a average depth of a node is $\Theta(\lg n)$ then the height of the tree is $O(\sqrt{n \lg n})$. We are going to prove this.

Suppose for, n-node binary search tree, the average depth is $\Theta(\lg n)$

and height = h

In this case, there will be a path from root to node at depth h, and the depths of the nodes on this path will be 0, 1, 2, 3, ..., h respectively.

Let there are two sets of nodes:

P = set of nodes on this path and

Q = " " " on other path.

∴ the avg. depth of node will be

$$\frac{1}{n}\left(\sum_{x \in P} \text{depth}(x) + \sum_{y \in Q} \text{depth}(y)\right) \geq \frac{1}{n} \sum_{x \in P} \text{depth}(x)$$

$$= \frac{1}{n} \sum_{d=0}^{h} d = \frac{1}{n}(0+1+2+3+\ldots+h)$$

$$= \frac{1}{n} \Theta(h^2)$$

For contradiction, suppose that h is not $O(\sqrt{n \lg n})$ so that $h = \omega(\sqrt{n \lg n})$

then

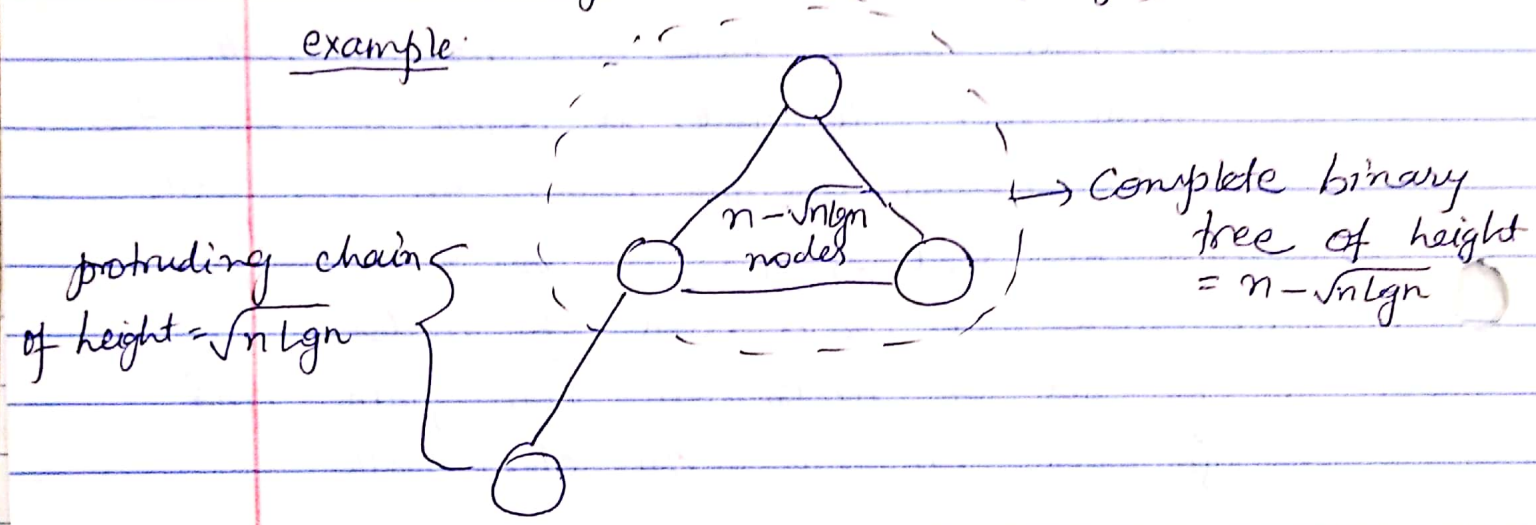
$$\frac{1}{n} \cdot \Theta(h^2) = \frac{1}{n} \cdot \omega(n \lg n)$$

$$= \omega(\lg n)$$

It contradicts the assumption that the avg. depth is $\Theta(\lg n)$.

∴ The height is ~~$\Theta\Theta$~~ $O(\sqrt{n \lg n})$.

example:



Height of above tree will be

$$= \Theta(\lg(n - \sqrt{n \lg n})) + \sqrt{n \lg n}$$

$$= \Theta(\sqrt{n \lg n}) = \omega(\lg n).$$

For calculating the upper bound ~~of~~ on the avg. depth of a node, we use $O(\lg n)$ as an upper bound on the depth of each of the $n - \sqrt{n \lg n}$ in complete binary tree and $O(\lg n + \sqrt{n \lg n})$ as an upper bound on the depth of each

of the $\sqrt{n \lg n}$ nodes in the protruding chain.

∴ The avg. height will be

$$\frac{1}{n} \cdot O(\sqrt{n \lg n}(\lg n + \sqrt{n \lg n}) + (n - \sqrt{n \lg n}) \lg n)$$

$$= \frac{1}{n} \cdot O(n \lg n) = \Theta(\lg n).$$

We can see in the tree that the bottommost level of the complete binary tree has $\Theta(n - \sqrt{n \lg n})$ nodes and each of these nodes has depth $\Theta(\lg n)$.

∴ the minimum avg. node depth will be

$$\frac{1}{n} \cdot \Theta(n - \sqrt{n - \lg n}) \lg n) = \frac{1}{n} \cdot \Omega(n \lg n)$$

$$= \Omega(\lg n)$$

Since the avg. node depth is $O(\lg n)$ and $\Omega(\lg n)$, ∴ it will be $\Theta(\lg n)$.

Problem 3:

Soln:

This algorithm will have two conditions:

1. Node will have right subtree: In this case, we will traverse to the least value of the right subtree. This least value will be the successor of the node
2. If the node does not have the right subtree: In this case, we will search the node from the root of the binary search tree and the successor will be the last-left node taken. If there is no last-left node, it will be the right-most node of the binary search tree and for that right-most node, there will be no successor.

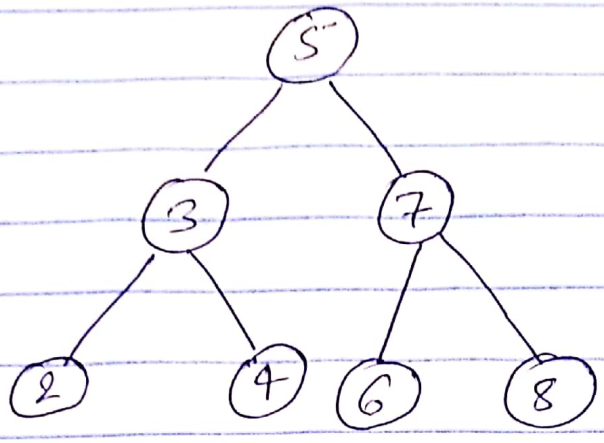
Here is the algorithm for these two cases mentioned above:

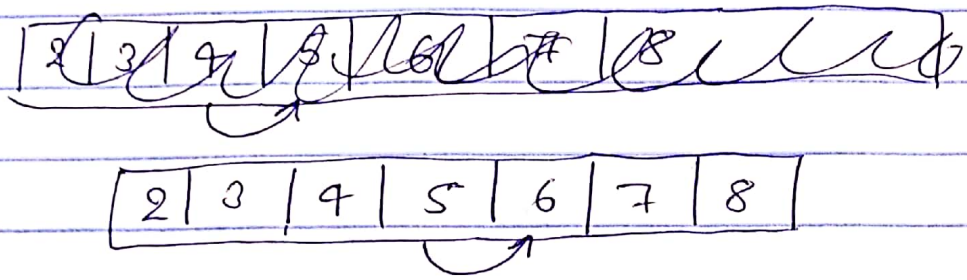
```
TREE-SUCCESSOR(v)
    IF (v.right ≠ NULL)
        {
            w = v.right
            while (w → left != NULL)
                w = w → left
            return w
        }
    ELSEIF (v.right = NULL)
```

```
{
    w = root;
    while (w -> value != v -> value)
    {
        IF (v -> value < w -> value)
        {
            w = w -> left
        }
        ELSE
        {
            w = w -> right
        }
    }
    return w
}
```

The above algorithm is explained here with the two cases considered:

Case I: If the node has a right subtree, we will traverse to the least value of the right subtree.



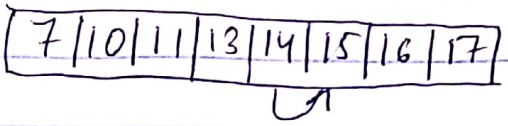


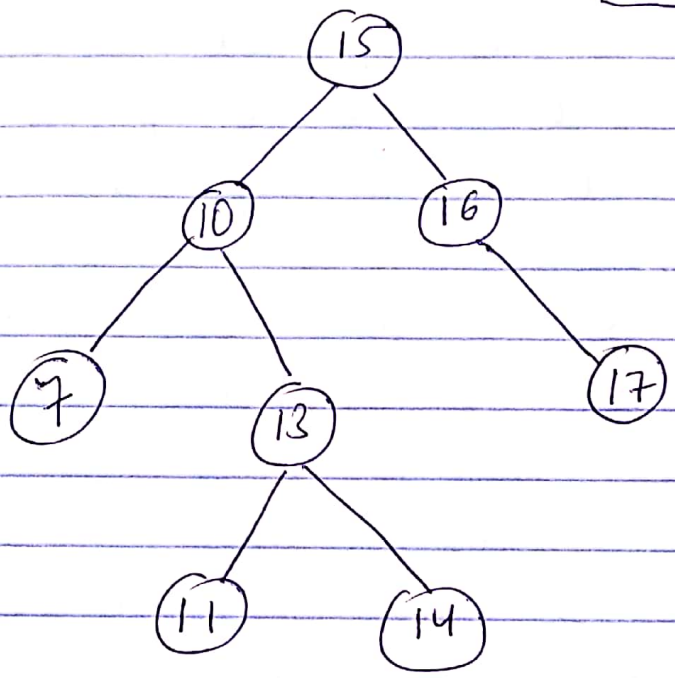
We can see that if we have right subtree, then we traverse to the right of that node, then we will check that the node has left node. If there is left node then we move to the left of that node until we reach the final left most node.

Here if we will find the successor of node 5, will first move to right node 7, and then we traverse the left node of 7 and we will find node 6.

∴ 6 will be the successor of node 5.

Case II → If the node will not have right subtree then we will search for the root node. and the last-left node will be the successor for the node.





Here, we are trying to find the successor of 14. First we will traverse to the root node which is 15 in this case. Since 15 > 14, we will take left turn and goto node 10.
∵ 10 < 14, we will take right turn and reach to node 13. Since 13
∴ The last-left node which is 15 in this case will be the successor of node 14.

Running-Time:

Now, we will find the running time of S-consecutive calls to successor in terms of S and h, the height of the tree.

In a binary search tree, we can traverse only two times through the branch. One will be upward and 2nd will be downward. With every double visit, we will be finding atleast one more successor.

∴ You will traverse the number of branches only once which can not be more than 2h. This is the worst-case situation happens when you start from the root and traverse to the leaf downward and again backward to the root.

∴ For k successors, you will need to traverse k branches two times (upward & downward) and 2h branches once.

∴ Running time with tight analysis will be $= 2h + 2k$

$= 2(h+k)$

$= O(h+k)$ Answer.

Problem 4

Soln:

Let's assume that A.max is a new field to hold the index of a high-order 1 in A. A.max is set to -1 initially, because there are initially no 1's in A and the low-order bit of A is at index(). When the counter is incremented or reset, A.max is updated accordingly and we use this value to limit how much of A must be looked at to reset it. In this way, we can control the cost of RESET and can limit it to an amount covered by credit from INCREMENTs.

Below are the algorithm for INCREMENT and RESET :-

```
INCREMENT(A)
    i=0
    while i< A.length and A[i] == 1
        A[i] = 0
        i = i+1
    if i< A.length
        A[i] = 1
        if i > A.max
            A.max = i
    else A.max = A.max - 1
```

~~RESET(A)~~

```
RESET(A)
    for (i=0 to A.max)
     A[i] = 0
    A.max = A.max - 1
```

We assume that it costs $1 to flip a bit. Also, we assume it costs $1 to update A.max.

Setting and resetting of bits by INCREMENT will work same as for the original counter, ie, $1 will pay to set one bit to 1 as credit and credit on each 1 bit will pay to reset the bit during the incrementing.

In addition, we will use $1 to pay to update max, and if max increases, we will place an additional $1 of credit on the new high-order 1. If max does not increase, we can just waste that $1, so it is not required.

Since RESET controls bits at positions only upto A.max and each bit upto there must have become the high-order 1 at the some time before the higher-order 1 got up to A.max, every bit seen

by RESET has $1 of credit on it.
So, we just need $1 to pay for resetting max.

Therefore, charging $4 for each INCREMENT and $1 for each RESET is enough, so the sequence of n INCREMENT and RESET operations will takes O(n) time.