Amitdeb Prasad Bhattacharya A20402789

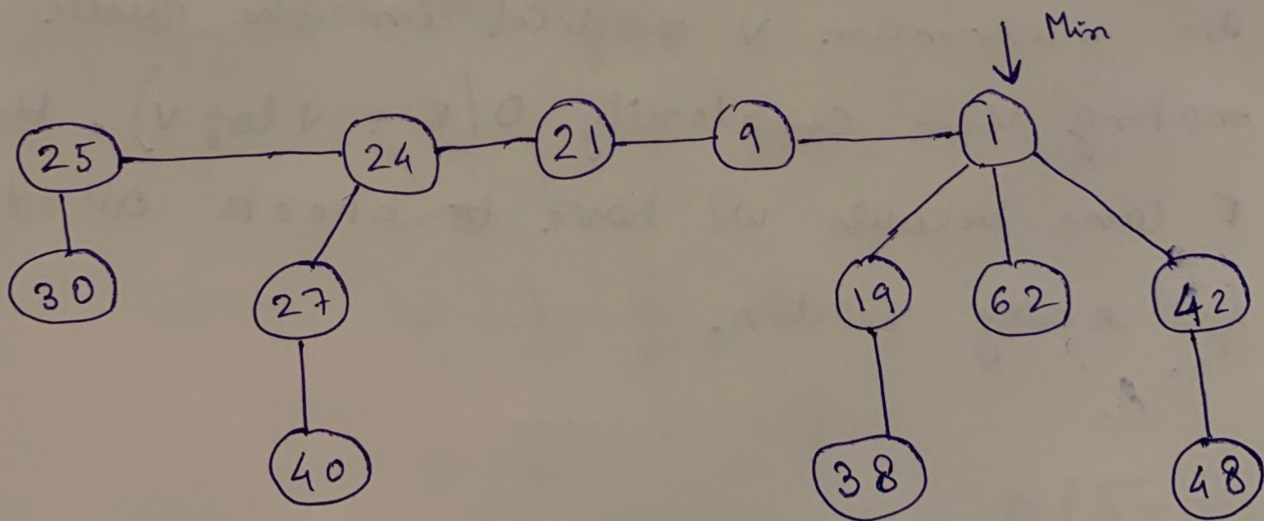
Problem 1

Let us say $G = (V, E)$ be a simple undirected graph with weights $w : E \to Z^+$. The inductivity of a vertex ordering (permutation π of V) $\langle v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)} \rangle$ is defined by

$$\max_{2 \le j \le n} \sum_{1 \le \pi(i) < \pi(j)} w\left(v_{\pi(i)} v_{\pi(j)}\right)$$

Like Binomeal heap, Fibonacci heap of a tree has min-heap or max-heap property. In Fibonacci heap a tree can have any shape.

Below is an Example of Fibonnacci heap,



Fibonacci maintains a pointer to minimum value. With fibonacci heap the time complexity is improved from binary heap to $O(v \log v + E)$

To make a tree we need to connect V vertices using edges from your initial graph. We can start from any node let us say 'v'. We need to add all vertices that can be reached from V with cost of the edges connecting it. We will definitely ~~take~~ go with cheapest cost. If we want to put vertex 'u' in my queue which is already present then we have to check wage on it's edge. If new one is smaller we need to take out older one from queue to insert new one, if not then we can skip it. If we are already connected node u to the tree then also we can skip it. So, there will be maximum V vertices in the queue making time complexity $O(E + V \log V)$. Here E came because we have to check all edges for every vertex.

Problem 2

The height of tree is $O(\sqrt{n \log n})$ if the average depth of a node ~~is~~ in a 'n' node binary tree is $\theta(\log n)$.

Let us say for n- node binary search tree has average dept $\theta(\log n)$ and height is h. So, there is a path formed from root to a node at depth h, and depths of the nodes on this path formed as a series starting from number 0 to h i.e.

$0, 1, 2, \ldots \ldots n.$

Let us assume A be the set of nodes on path and B be the rest of the node in this binary search tree.

so, average depth of a node is

$$\frac{1}{n}\left(\sum_{x \in A} \text{depth}(x) + \sum_{y \in B} \text{depth}(y)\right)$$

$$\geq \frac{1}{n} \sum_{x \in A} \text{depth}(x)$$

$$= \frac{1}{n} \sum_{d=0}^{h} d$$

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

For contradiction, let us say $h \neq O(\sqrt{n \log n})$, so that $h = \omega(\sqrt{n \log n})$ [let us assume]

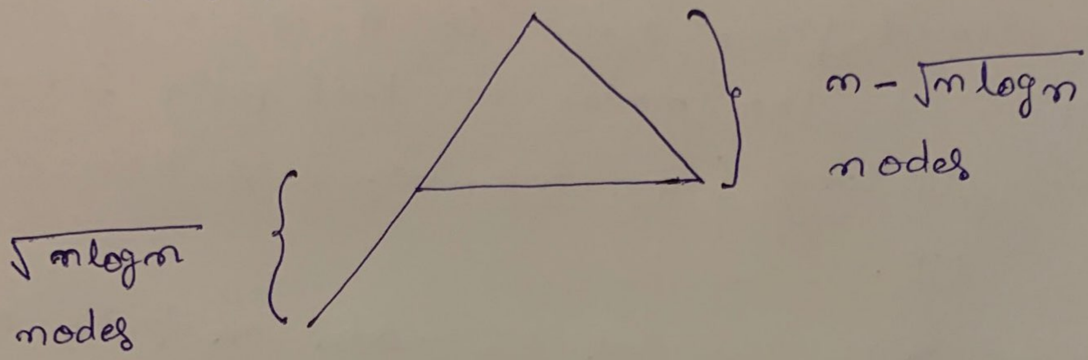
We have,

$$\frac{1}{n} \cdot \theta(h^2) = \frac{1}{n} \omega(n \log n)$$

$$= \omega(\log n)$$

This contradicts the assumption that the average depth is $\Theta(\log n)$. Thus the height is $O(\sqrt{n \log n})$.

Following is an example of binary search tree with average node depth $\Theta(\log n)$ but height is $\omega(\log n)$.



In this tree $(n - \sqrt{n \log n})$ nodes are complete binary tree, and the other $\sqrt{n \log n}$ nodes protude from below as a single chain.

The height of above tree is

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

To compute an upper bound on the average depth of a node, we use $O(\log n)$ as an upper bound on the depth 'p' of each of $n - \sqrt{n \log n}$ nodes in the complete binary tree part and $O(\log n + \sqrt{n \log n})$ as an upper bound on the depth of each of the $\sqrt{n \log n}$ nodes in the protruding chain.

So, the average depth of a node is bounded from above by

$$\frac{1}{n} \cdot O\left(\sqrt{n \log n}\left(\log n + \sqrt{n \log n}\right) + \left(n - \sqrt{n \log n}\right) \log n\right)$$

$$= \frac{1}{n} O(n \log n)$$

$$= O(\log n)$$

To find the lower bound of average depth of a node.

Bottomost level of the complete binary tree has $\Theta\left(n - \sqrt{n \log n}\right)$ nodes, and each of those nodes has depth of $\Theta(\log n)$.

The average node depth is at least

$$\frac{1}{n} \cdot \Theta\left(\left(n - \sqrt{n \log n}\right) \log n\right)$$

$$= \frac{1}{n} \cdot \Omega(n \log n)$$

$$= \Omega(\log n)$$

Since the average depth of a node is both $O(\log n)$ and $\Omega(\log n)$, it can give $\Theta(\log n)$ which means our assumption is correct.

Problem 3

Let v be the given node of binary search tree and w be the inorder successor of node v. We are considering v and root as the input to the algorithm to get node successor of v as output which is w. Here we have to consider two cases on basis of right subtree of input node as empty or non-empty.

1. If right subtree of node v, is not empty then inorder successor of node v, is present in right sub tree of binary search tree.

```
if v.right != NULL:
```

2. Go to right sub-tree and return node which is having minimum value in right sub-tree.

```
return min(v.right)
```

3. If right subtree of node v, is empty then successor of node is one among the parents.

4. Traverse back using parent pointer as mentioned in problem 3, until we get a node which is left child's parent and we can say that parent of such node as successor.

```
p = v.parent
while (p != NULL):
    if v != p.right :
        break
    v = p
```

p=p.parent

return p

We can traverse number of branches in Binary Search tree for only once can not exceed more than 2h times and this worst case senario occurs when we start from a leaf in bottom left side of a binary search tree, i.e. we have to traverse all the way up to root and then we have to go down to to bottom-leaf to find successor. We have to visit some of these branches again before we can go to other branches for the first time to find more successor. Hence, total number of branches we can travel for one time can not be more than 2h times. In the 2^{nd} part of Problem 3 for 's' consecutive call, we can traverse all the branches for 2s times. So, for a worst case senario, branch traversal count of $(2h+2s)$ is $O(h+s)$.

Problem 4

Let us implement a binary number as a bit vector so that any sequence of n INCREMENT and RESET operation takes time as O(n) on an initially zero number. Here we have to increment a number and also we have to reset ~~the~~ all bits of the number to zero. We are going to take a new field max[A] which holds the index of high order 1 in A. Initially we need to set max[A] to −1. Now, we need to update max[A] when the number is incremented or reseted. The cost of RESET can be limited to an amount that can be covered from earlier INCREMENTS.

INCREMENT(A)

i = 1
while i < length[A] and A[i] = 1
do A[i] = 0
i = i + 1
if i < length [A]
~~then~~ ~~max[A] = i~~
then A[i] = 1
if i > max[A]
then max[A] = i
else max[A] = − i

RESET (A)

for i = 0 to max[A]

do A[i] = 0

max[A] = -1

Let us assume, to update max[A] the cost is $1, and cost to flip a bit as $1. We need to pay $1 to set bit to 1 and place another $1 on same bit as credit, so that credit on each bit will pay to reset the bit. We will use $1 to pay update max[A] and if max[A] increases, we will place an additional $1 of credit on the new high-order 1. Every bit seen by RESET has $1 credit as RESET manipulates bit at sometime before the higher order 1 got up to max[A]. So, if we reset the bit, cost can be paid from stored credit. We need to pay $1 to reset max[A].

Thus, charging $4 for each INCREMENT and $1 for each RESET is sufficient. The sequence of 'n' number of INCREMENTS and RESET operation takes O(n) time.