Name: Animesh Patni

A#: A20403240

Design Analysis of Algorithms Assignment 3

Question 1:→

In this problem, if the fibonacci heap contains one tree with a linear chain of n nodes. First, create a Fibonacci heap with the height of one with the root key → N. Adding nodes with keys:→

N−1 → a value less than current root

N−2 → a value which is twice less than current node.

N+1 → a value one more than current root.

N+2 → a value two more than current root.

Delete the minimum, decrease the key of node next to the minimum to −∞, then delete that element. Will repeat the above steps and the last step completely deletes it.

As we can obtain n-nodes from c operations, therefore we can construct $\Omega(n)$-nodes chain with c operations.

Proof of correctness:→

Base Case:→

As $n = 1$;
So, in this case the fibonacci heap contains one tree, with a linear chain of n-nodes.

Inductive Case:→

The base condition makes it true for $n = z$. Now checking for $n = z+1$, we create the linear chain following the steps above. Now, adding the other two child nodes: c1 → less than min key
c2 → greater than min key. At last the

Root (R) is added such that it is the minimum key. After removing R then the chain would of x & y, which proves the above case. is correct. Using the above approach we will always have a linear chain of nodes attached, which gives a running time of $\Omega(n)$. After every operation we add two nodes to the list. So, after adding, deleting & re-adding elements such the number of elements remain same. So, as the operation increases constantly $\rightarrow c \cdot n$, thus $\Omega(c \cdot n) = \Omega(n)$.

Question 2: →

So, in this problem, I am going to store the value of the pointer of a node in a linked list, that will point to the next element or node if it is attached too. It will contain a single tail pointer. The only information added will be a pointer. So, when we do the make set operation, we will also create a new linked list for every element that is added, & add x-pointer to the element' label. This is done in O(1) time for every node. Find-Set would remain the same. Union (x, y) would merge the linked list of element x & y and every other operation in the union algorithm would run exactly the same; thus

the running time for union won't change. The PRINT-SET operation would call FIND-SET

SET = FIND-SET(X), then print the element in any order present starting with element pointed by X. Since, the list contains the same number of elements as in the SET operation, & it is given that print would take O(1) time. This would also be done linear time.

MAKE-SET(x)

1. $x.p = x$
2. $x \rightarrow head = x$
3. $x \rightarrow tail = NULL$.
4. $x.rank = 0$

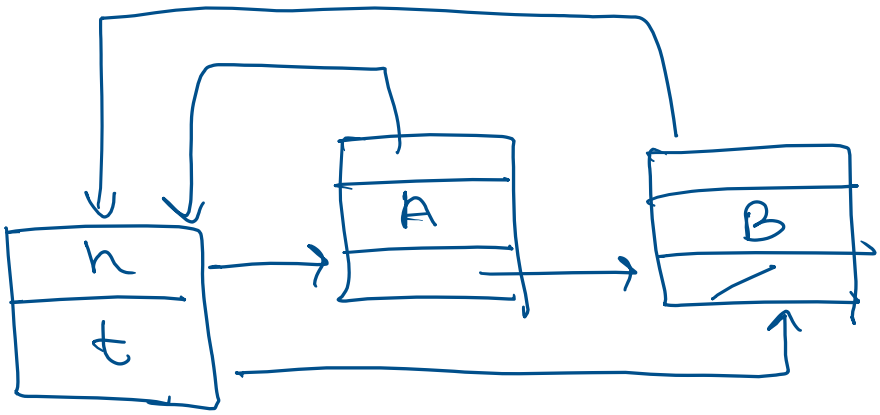
```
UNION (x, y):
1. LINK (FIND-SET(x), FIND-SET(y))

LINK (x, y):
1. IF  x.RANK > y.RANK :
2.          y.p = x
3.          x → tail → next = y → head
4.          x → tail  = y.tail.
5.          y → head  = x → head.
6. ELSE     x.p → y
7.          y → tail → next = x → head
8.          y → tail = x.tail
9.          x → head = y → head.
10.         IF x.RANK == y.RANK
11.                 y.RANK = y.RANK + 1

FIND-SET (x):
1. IF x ≠ x.p
2.          x.p = FIND-SET (x.p)
3. RETURN x.p.
```

```
PRINT-SET(X):
1.  S = FIND-SET(X)
2.  WHILE(S → NEXT IS NOT NULL)
3.          PRINT(S→VALUE)
4.          S = S → NEXT.
```

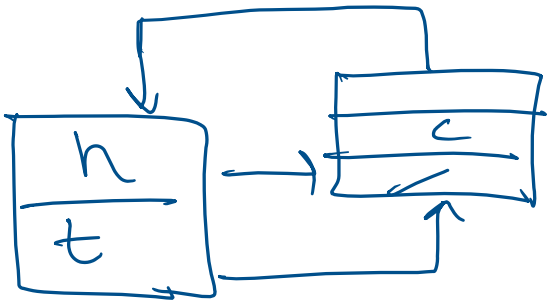
So, in the above example each node will have it's head & tail (linked list object). After each merging / union of this linked list object that object of the linked list getting merged will be destroyed. After this in the print set I am finding the set representative and printing the elements in the set in a linear time. Showing the example of how linked list would work below:→

SET 1:→

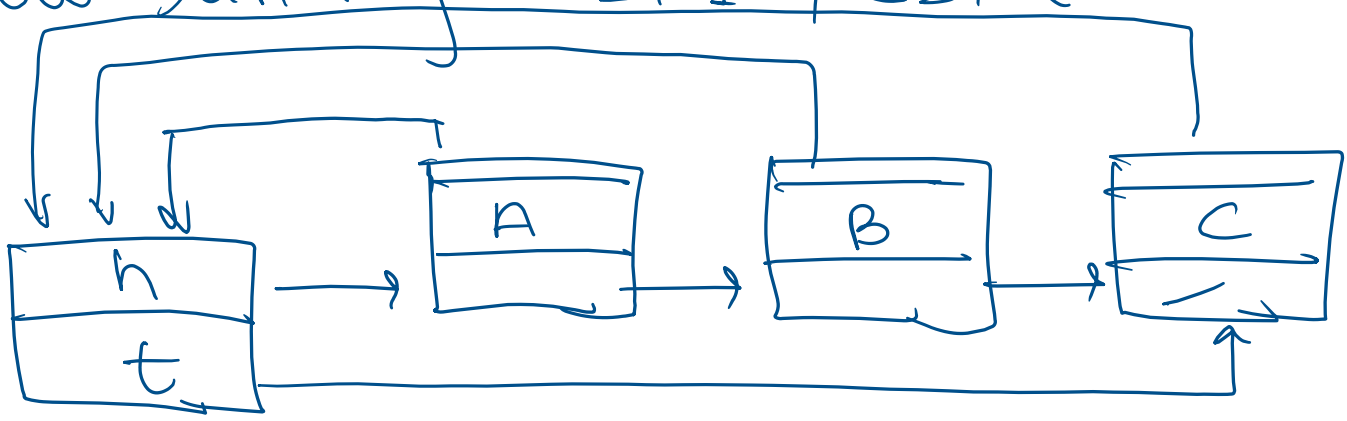
h → head
t → tail.



SET 2:→



Now linking SET 1 & SET 2:→



SET 2 will be destroyed.

Question 3 :→

I am going to use the credit scheme to to obtain $O(m)$ bound.

Suppose Make Set takes 2 credits.

1 pays for make set & 1 remains on the node as indicated in the hint.

The 1 credit on the node pays for the first time the node appears in the findset & is turned into the child of the root (path compression).

Then for the union operation 1 credit is charged, which pays for union operation.

Then comes the FIND-SET operation, in which we have 3 cases :→

(i) The node is the representative of it's own set. In this case, it clearly takes one constant time to run.

(ii) In this case the path from node to it's set representative is already compressed. So, it only takes constant time as it is just one step to find the representative.

(iii) In this case the node is not the representative & the path compression is not being done. I put 1 credit for find-set, this credit pays for visiting the root & the child, & for the path compression of these two. And all the other nodes traversed on this find path will use the dollar stored on them. Since, atmost of 2 credits is charged, so

a sequence of m operations will at most take 2m credits or $O(2m)$ time. ∴ Amortized cost will be $O(m)$. Hence proved.
From the book (CLRS), the worst case running time is $O(m\alpha(n))$ where $\alpha(n) \leq 4$

∴ In constant time it would eventually take $O(m)$. Hence proved.

PART-B :→

In the worst case scenario, we traverse the whole tree, till we reach the set representative. So, suppose we have n elements

we traverse through n-elements & to reach the set-representative we took total of q operations that is FIND() is called q times till we reach the set representative. So, the total time is $O(q+n)$.

Question 4 :→

I am going to prove it by contradiction
So, first looking at the successor of the given node A.

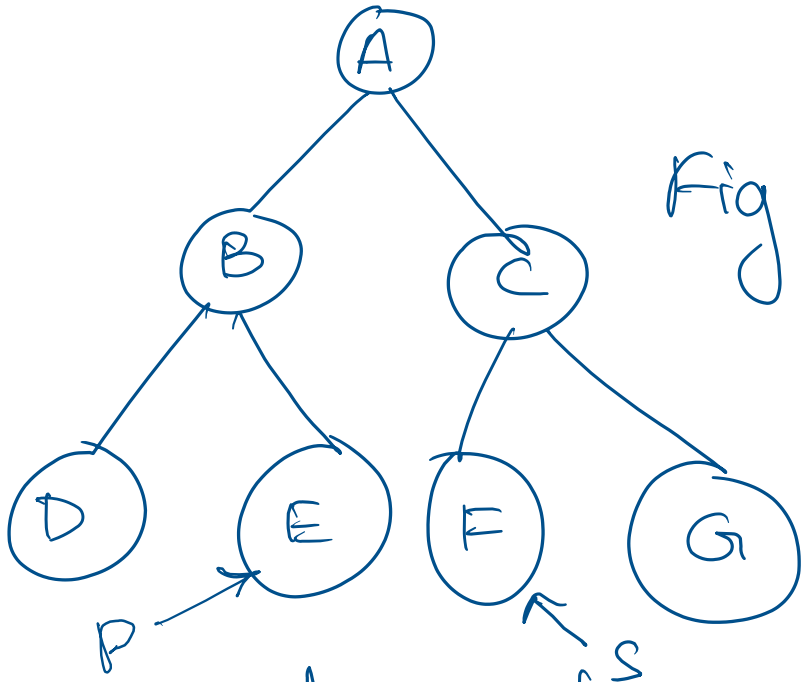
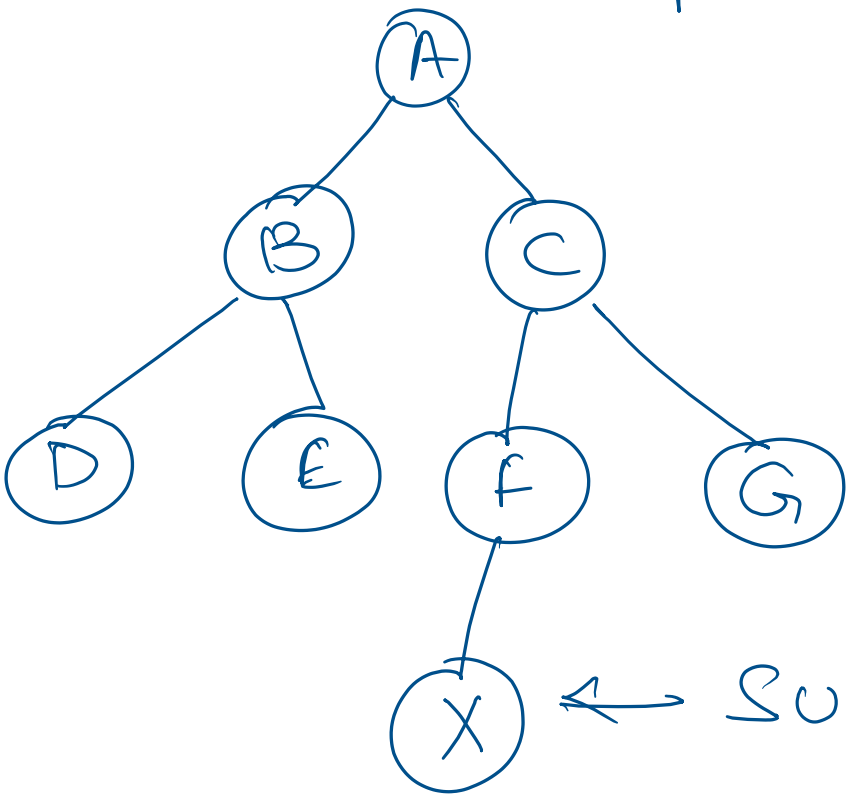


Fig 1.

So, for any node in this case A, F would be the successor of A denoted by 's'. Then F cannot have a left child cause for a left child of s would come in

between s & A in the inorder walk. If F has a left node then:-



then the successor of A would become X which contradicts the statement given in the question.

So, generalizing the above example, let s be a successor of node N. Then s cannot have a

left child, as the left child would be the new successor & would come in between S & N in the inorder traversal. So, s won't be the successor of N as supposed which contradicts since S is the successor of N

KEY[S] ≥ KEY[Z] ≥ KEY[N]

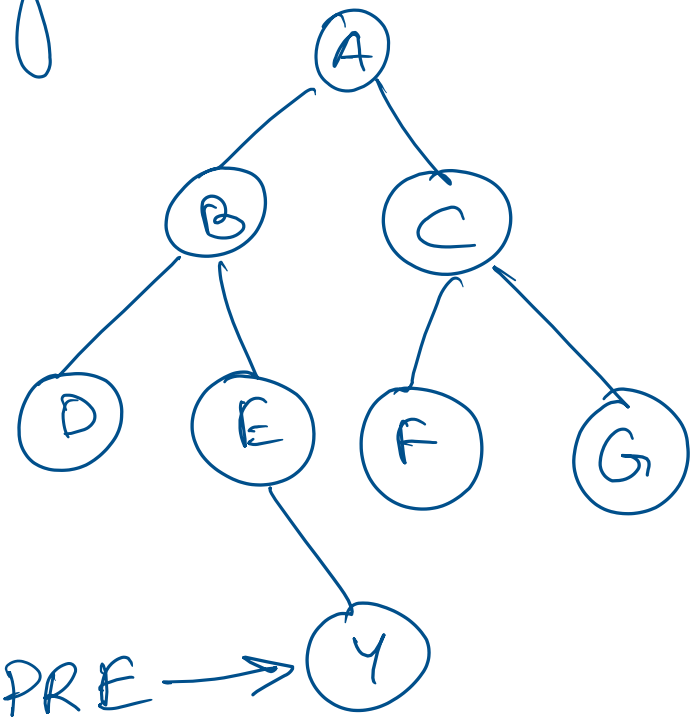
↓

Z being the left node of S.

Hence proved by contradiction that s would have no left child.

So, second step for the <u>predecessor</u>, looking at the same fig 1. Suppose E has a right child as shown

in the figure below.



In figure 1 we have E as the predecessor of A, but adding Y would contradict that. Hence proved by contradiction. Now, generalizing the above. Suppose P be a predecessor of node N. Now, if P have a right child it would come in between P & N, in this

case the new node would become the predecessor, hence contradicting that p is the predecessor of N.

$$key[p] \leq key[z] \leq key[N]$$

z being the right node of p

Hence, proved by contradiction that p would have no right child.