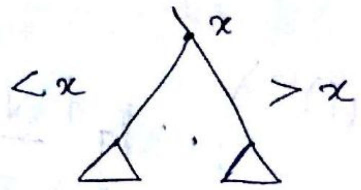
Homework Assignment-5 Dhayalini Nagaraj
A20359686

1. At the top of page 4 in the notes on lazy weight balanced trees, it is claimed that "after rebuilding all imbalances are zero in the subtree". Prove this statement.

Lexicographic Trees.



When imbalance occurs during deletion and insertion, the tree is rebuilt.

According to the given notes the subtree is rebuilt using divide and conquer algorithm.

Divide and conquer algorithm works by recursively breaking down a problem into two or more sub problems.

Similarly in this, the middle element is kept at the root and the left and right trees are constructed recursively.

Therefore, the tree can have a height difference of atmost 1 between two subtrees after rebuilding it. We know.

$$I(x) = \max\{0, |size(left(x)) - size(right(x))| - 1\}.$$

So, the imbalance of the rebuilt subtree is zero. The remainder of the entire tree has 0 imbalance before rebuilding the tree. So, the whole new tree that is rebuilt has imbalance of 0.

Hence Proved.

2. Redo the amortized analysis of insertion/deletion in lazy weight balanced trees with $I(x) = \{|size(left(x)) - size(right(x))|\}$ in the potential function.

<u>Deletion:</u>

If <u>rebuilding</u> <u>is not required</u>, then the amortized analysis is same as in the lecture notes: The imbalance is not changed.

(ie) Actual cost $= O(height(T)) = O(\log size(T))$

$= O(\log num(T))$ for the search

$$\Delta \Phi(T) = \hat{c} = O(1)$$

Since $m(T)$ increases by 1.

REBUILD TREE

If the tree is rebuilt, then the actual cost is $O(\log n) + \Theta(n)$

After rebuilding, the imbalance could be ~~ab~~ at most $\Theta(n)$ and

$$\Delta\Phi \leq -\hat{c}\ \text{size}(T)/2 + \Theta(n)$$

By scaling $\hat{c}$, the n term in actual cost and potential change is cancelled.

Then the amortized cost is $O(\log n)$

~~KRE~~ INSERTION

NO REBUILD

If rebuilding the tree is not required then imbalance change of path from root node to the new inserted node is atmost

$\text{height}(T) \leq \beta \log n$

The number of marked nodes will not change.

$\therefore\ \Delta\Phi \leq \beta \log n$

Actual cost is depth search and it is $O(\log n)$

So, amortized cost is $O(\log n)$

TREE REBUILD:

If the tree is rebuilt, then

actual cost $= O(\log n) + \Theta(n)$

Change in potential is similar to the one in the notes. We need to calculate the imbalance $I(x)$ before the insertion.

$$I(x) \geq size(left(x)) - size(right(x))$$

$$\geq \frac{size(x)}{2^{1/\beta}} - \left[\left(1 - \frac{1}{2^{1/\beta}}\right) size(x) - 1\right]$$

$$= \left(2^{1-1/\beta} - 1\right) size(x) + 1$$

$$I(x) > \left(2^{1-1/\beta} - 1\right) size(x)$$

After rebuilding the tree

$$I(x) = O(size(x))$$

Potential change $\Delta \Phi = O(n)$

Amortized cost = Actual cost + Potential change

$$= O(n)$$

Scanned by CamScanner

... cost is ...

4. Professor Pinocchio claims that the height of an n-node, Fibonacci heap is ~~[illegible]~~ $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\lg n)$

Let us consider an empty Fibonacci heap F. The ~~[illegible]~~ idea of the algorithm is to create recursively a Fibonacci heap that comprises of only one tree, a linear chain of $n-1$ nodes. Then one more node is added to the chain

Fibonacci heap of heigh 1 should be created with the root key as k. Then elements are added to it, in such a way

k-1 - a value less than the root key

k+1 – a value more than the root key.

k-2 - a value, two times less than the root key.

They k-2 is deleted. This will generate the required fibonacci heap

We assume that the number of nodes in the tree is greater than two.

<u>Pseudocode</u>.

~~Linear-heap (F, k, n)~~

```
Linear-heap(F, n, K)  // start with empty F
    Linear-heap (F, n-1, K+1) // add node with value
                                 more than root
    Insert (F, min(F)+1)   // add a node with value
    Insert (F, min(F)-1)   // less than root node min value
    Insert (F, min(F) - 2)
  Deletemin(F)             // deletenode with minimum
                                                  key
  a = min(F).secondchild  // assign minimum
                                          node's
                          child node to variable
  Decreasekey(a, min(F)-2)        a
  Deletemin (F)
  return
```

Proof of correctness:

The hypothesis is correct for $n=1$ as the Fibonacci heap contains one tree with a linear chain of n-nodes.

The proof is done by induction.

Assume that the above statement is true for $n = l$. Then for $n = l+1$, the algorithm creates a linear chain containing l nodes.

After creating, it adds two child nodes x and y. The child node x has a key which is less than the minimum key.

The child node y has a key which is greater than the minimum key.

Finally it adds the node k, such that the node k's key is smaller than the minimum key.

When the node k is deleted, then there remains a chain of l nodes containing x and y. Since the degree of the nodes x and y ~~so~~ is zero, the nodes are joined.

Now we obtain a chain with two elements, where x becomes the root node and has degree 1.

Thus the chain of height two and the chain of height 1 are combined. Now removing the node y, we obtain the linear chain with n nodes.

Hence a Fibonacci heap, containing one tree with a linear chain of n nodes, can be created from a sequence of Fibonacci heap operations. This proves that the Professor Pinocchio is wrong.

6.a) The operation FIB-HEAP-CHANGE-KEY(H, x, k) changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-~~PRUNE(H,x)~~ CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less or equal to $x.key$.

i) $K > x \cdot key$

K can be greater than keys of some children of x. So update the key

$x \cdot key \leftarrow k$ and push the x down until the min heap property is preserved.

Worst case actual cost is $O(\log n)$. There is no potential change.

Therefore amortized cost is

$= $ Actual cost + change in potential

$= O(\log n)$

ii) $K = x \cdot key$

Nothing is done in this. So no potential change.

$\therefore$ The amortized and actual cost are equal and it is $O(1)$

iii) $K < x \cdot key$

It calls FIB-HEAP-DECREASE-KEY (H, x, k)

The amortized cost is $O(1)$

Scanned by CamScanner

6b) Given an efficient implementation of FIB-HEAP-PRUNE (H, r), which deletes $q = \min(r, H, n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation.

The amortized cost of deleting a given node is $O(\log n)$. In this problem, any q nodes can be deleted, ~~which is like we try deleting q leaves~~ That is we try deleting q leaves.

Worst case is every leaf that is deleted will incur cascading cut untill the root which implies the actual cost of

FIB-HEAP-PRUNE (H, r) is $c = q \log n$.

$\therefore$ The potential change can be found using

$$\Phi(H) = t(H) + 2m(H)$$

$$t(H_{after}) - t(H_{before}) = q \log n$$

In the worst case $q \log n$ new trees are created due to the promotion during the cascading cut. H_{after} has $q \log n$ more trees

$m(H_{after}) - m(H_{before}) = -q \log n$

$-q \log n \Rightarrow 2m(H_{after}) - 2m(H_{before}) = -2q \log n$

This is because, in the worst case $q \log n$ marked nodes are promoted as new ~~nodes~~. roots. This means $q \log n$ less marked nodes are there in H_{after}.

Then

$$\hat{c} = c + \Phi(H_{after}) - \Phi(H_{before})$$
$$= q \log n + q \log n - 2q \log n$$
$$= O(1)$$

Therefore, removing leaf nodes amortized cost is $O(1)$

Scanned by CamScanner