ASSIGNMENT: HW5

NAME: PRASANTH BHAGAVATULA

CWID: A20355611

① During, the re-building operation of sub-tree rooted at x, we will create a new sub-tree with root as 'x' and all child nodes of x. While creating the new tree, we will create a balanced tree and of course, that is what we want to achieve during re-building. Now, there can be two cases, the subtree with root x can have odd number of child nodes (or) even number of child nodes.

Consider the case that of having 'x' having odd number of child nodes. In this case, either the left-sub-tree of 'x' or right sub-tree of x would be having the extra-one element

$$\Rightarrow \quad \text{if } |\text{size}(\text{left}(x)) - \text{size}(\text{right}(x))| = 1$$

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

$$= \max\{0, 1-1\}$$

$$= \max\{0, 0\} = 0$$

If we consider even number of child nodes, the a perfect balanced tree will be created ~~and so the height~~ and both the sub-trees will have the same number of elements and so,

$$|size(left(x)) - size(right(x))| = 0$$

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

$$= max\{0, 0-1\} = max\{0, -1\} = \underline{0}$$

Hence, after re-building, the imbalance of sub tree rooted at x will be zero.

ASSIGNMENT: HW5

NAME: PRASANTH BHAGAVATULA

FALL 2015

EWID: A20355611

② If $\beta = 1$, then ~~it means~~ $height(x) \leq \log(size(x))$, th

This means that all the leaf nodes are to be on the same level (height). If any one insert operation, violates this rule, the we will be re-building the tree.

If $\beta > 1$ and very large, say billion, then the height of one sub-tree can be very large than the height of the other sub-tree and still no re-building will occur. So, in worst case, there can only be a left sub-tree (i.e. only elements to the left of each and every node till leaf) and still it might satisfy the mentioned condition. So, in this case the ~~wor~~ time taken to access the leaf would be more of order $O(n)$ than $O(\log n)$.

ASSIGNMENT : HW5

NAME: PRASANTH BHAGAVATULA

CWID: A20355611

FALL 2015

(3) Th. When we consider the Imbalance function as mentioned below

$$I(x) = |\ size(left(x)) - size(right(x))\ |$$

I if we look at delete operation, without the re-building the tree, the Imbalance never changes as there is no physical deletion and so the delete-with-out-rebuild open operation is still $O(\log n)$

Consider the delete-with-rebuild operation. Let $I(T)$ be the sum of imbalances of the tree before deletion. Now after re-buiding, the tree can have odd number of children for the root and in that case one of the sub-trees will have one extra elements than the other. This increase can occure upto $\log\left(\frac{size(T)}{2}\right)$ elements. (since now we have only $|T/2|$ elements)

$$\therefore \Delta\phi \leq -\hat{c}\ size(T)/2 - I(T) + \log\left(\frac{size(T)}{2}\right)$$

$$\Rightarrow \Delta\phi = -\hat{c}\ size(T)/2 - I(T) + \log n - \log 2$$

And we know that the actual cost includes accessing the element $O(\log n)$ and $\Theta(n)$ - rebuilding the tree.

So, the amortized cost will increase by $\log n$ and so it is still $O(\log n)$

Now, consider the insert-with-out-rebuilding operation; In this operation the imbalance at each level just increases by atmost 1 and so the increase in potential will be same as mentioned in the material

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

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

Now, consider the insert-with-rebuild operation. This is also the Here also, it is similar to the rebuilding scenario in delete function, except that the increase in potential (of original) will only happen for the sub-tree rooted at 'x'. Meaning, consider only the sub-tree rooted at x. The Because of the new imbalance function, the change in value will be only in this subtree and so the maximum it can increase is height(x) (one at each node of subtree rooted at x)

$$\therefore \quad \text{height}(x) \leq \beta \log (\text{size}(x))$$

So, the final amortized cost increases by a constant factor of $\log n$ assuming that the x is same as the root of the tree; ie the entire tree becomes unbalanced. So the amortized cost is still $O(\log n)$

ASSIGNMENT: HW5

NAME: PRASANTH BHAGAVATULA FALL 2018

CWID: A20355611

(4) We need to concatenate two lazy weight balanced trees T_1 and T_2. Here as we know for sure that all elements of T_1 are less than all the elements of T_2. The first step, as mentioned, is to delete the largest-value node from T_1 which we will refer to as 'patch-node'. Note that, by deleting the patch node, we might decrease the height of tree T_1 (or) we might not decrease.

Now, we need to consider a certain point T_3 in the modified T_1. Instead of going down the right edge of modified T_1, we will consider T_3 as the root of T_1 itself. Since this does not have any root, we take the patch node and add the tree T_1 as its left subtree and T_2 as its right sub-tree.

Note that the value in patch node is the largest value in T_1 subtree and so all the elements of T_1 will be onto the left if we construct a tree with patch node as root.

Also, we know for sure that all elements in T_2 are greater than all elements in T_1 and so they all lie in the

right sub-tree. If the tree is having patch-node or the root.

So, the lexicographic condition holds good for the new tree (combined or joined tree). Since, by adding the patch node at root level the height of the new tree will be

$$height(T_3) = max\{height(T_1), height(T_2)\} + 1$$

and

$$size(T_3) = size(T_1) + size(T_2)$$

Note that we are not adding 1 to the size, because the patch-node is already present in the tree T_1, before the join.

If we look at the right sub-tree, which is the modified (T_1), it satisfies the balancing condition and similarly the left sub-tree, T_2, also satisfies the balancing condition. So, if at all the new tree has to re-built, it has to be because the root is out of balance.

So, if the tree goes out of balance, it will be at the root and we need to re-build the entire tree.

Consider the first scenario, where re-building is not required. So, the actual cost is finding the longest node in tree (T_1)

$\Rightarrow$ Actual cost = $O(\log \text{size}(T_1))$

Now, we need to compute the change in potential. The number of marked nodes in T_1 and T_2 are not going to change and so they cancel out. The change in imbalance is calculated as follows.

Consider, the sub tree (T_1') of the concatenated tree. The difference between this and the old tree T_1, would be that the patch node is missing. So, the potential at each and every node is increased by 1 for the elements in sub-tree T_1 in the new concatenated tree. In addition, we need to compute the imbalance at the root node. But there is no change in imbalance for right sub-tree T_2

$\Delta\phi \leq c\,I(T_3) + c\log(\text{size}(T_1'))$; where T_3 is root node of new tree

∴ The amortized cost is $O(\log(\text{size}(T_1)) + c\log(\text{size}(T_1')) + cI(T_3)$

and so $O(\log(\text{size}(T_1')))$

Consider the scenario, where re-building is required. So, now after re-building the imbalance at each and every nodes in the new tree will be '0'.

Hence

$$\Delta\phi = -c \sum_{k=1}^{size(T_1)} I(k) - c \sum_{j=1}^{size(T_2)} I(j)$$

And we know that the imbalance function is proportional to the size of that node

$$\therefore \Delta\phi = -\Theta(size(T_1)) - \Theta(size(T_2))$$

$$\Rightarrow \Delta\phi = -\Theta(size(T_1) + size(T_2))$$

Now, the actual cost includes finding the largest element in T_1 and then re-building the entire tree ($size(T_3) = size(T_1) + size(T_2)$)

$$\therefore \text{Actual cost} = \Theta(\log size(T_1)) + \Theta(size(T_1) + size(T_2))$$

$$\therefore \text{Amortized cost} = \text{Actual cost} + \Delta\phi$$

$$= \underline{\Theta(\log size(T_1))}$$

$\therefore$ The amortized cost of this join is

$$\underline{\Theta(\log(size(T_1)))}$$

ASSIGNMENT: HW5

NAME: PRASANTH BHAGAVATULA

CUID: A20355611

FALL 2015

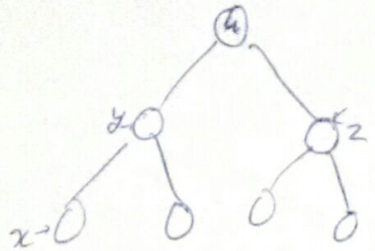
(5) Let us consider the scenario, where we don't need re-building ~~b~~ when we do the insertion. ~~When~~ we For all insertions, we compare the length we have gone into the tree with overall height of tree and update if we travelled more the height of tree. We simply increment the size of tree by 1. Now, we will validate balancing condition.

Since, ~~the~~ we are considering a scenario where re-building is not required, the balancing condition is satisfied and so, this is similar to the regular operation.

~~Even In the case we need to compute the size and height of each and every node from x to the root, to check~~

In case, the root goes out of balance, we need to find out, the lowest unbalanced node. We know for sure that, if 'u' is the node having unbalanced ~~tree~~ subtree, the cost of re-building that tree will be $\theta(size(u))$.

Now we need ~~to~~ the time to compute the size and height (u) so that we can say that the subtree at root u is unbalanced.



~~Lets say that~~

Consider the above figure, we now need to compute the size and and height at (u). Before reaching (u), we know for sure, rather we would have already computed the size(y) and height(y). So, we now need the size(z) and height(z) so that we can compute the size(u) and height(u).

Computing the size(z) requires accessing all the elements in the sub-tree 'z'. That means, in-order to determine the height(u) and size(u), we need to visit each and every element in 'u' and so the it take Θ(size(u)) to compute size(u) and height(u).

~~Since accessing and~~

Since computing and re-building can be done in $O(size(u))$, the entire re-building operation can be done in $O(size(u))$

∴ ~~The amortized cost of this event~~

In worst case scenario, 'u' will become the root of tree

⇒ Amortized cost: $O(\log size(T)) + O(size(T))$

⇒ ~~Amortized cost~~ =

ASSIGNMENT : HW5

NAME : PRASANTH BHAGAVATULA

CWID : A20355611

(6) Consider $k = 1$. That means, we will cut the node x from its parent as soon as 'x' loses its first child. So, this in turn, will cut the parent of 'x' from grand-parent (x) since, the parent of 'x' has lost a single child (x). So, this will cause a ripple effect and it will cascade till the root. Since, we don't do ~~any operation~~ have any cascading cut operation at root level, it will stop there.

~~Now we know that there can be a maximum of $\log n$ elements between~~

So, the number of trees going to be created is ~~same~~ proportional to the number of elements between 'x' and the root of sub-tree and so it is $O(\log n)$, where n is number of elements in the heap.

Now to consolidate all the trees at root level, we need an array $A[0 \ldots D(n)]$ and we know that the above operation has created a number of trees which is proportional to $O(\log n)$. Note that, when the above operation happens, the trees that are added at root can all be of different degree and so no consolidation can happen. So, the maximum array size required is

$$D(n) = O(\log n)$$

In fact, the $D(n) = O(\log n)$ is applicable for all k. For $k=1$, we can create maximum number of trees being inserted into the root level (it will surely ie. cascading-cut will happen till root level).

∴ The time bounds of all other heap operation remain some

Make-heap	$O(1)$
Insert	$O(1)$
Min	$O(1)$
Extract Min	$O(\log n)$
UNION	$O(1)$
Decrease key	$O(1)$
Delete	$O(\log n)$