Problem – Add PRUNE to Fibonacci Heaps

Give an efficient implementation of an operation PRUNE(H, r), which deletes min(r, n[H]) nodes from H. Which nodes are deleted is up to you as the implementer. Analyze the running time of your implementation.

Solution Overview

The key ideas are: a) to only prune leaf nodes; b) to maintain a doubly-linked circular list of the leaf nodes (and a global pointer to that list); c) to prune r nodes by pruning one node at a time; and d) to charge the cost of pruning a node x to the INSERT operation that placed x in the data structure. We note also that if pruning converts some parent nodes into leaf nodes, they are added to the leaf list as well. Also, when a node loses its second child during the pruning process, cascading cuts will occur.

A Pruning Algorithm

To implement the leaf list we add two pointer fields to each node of the Fibonacci Heap. Those pointers allow the node to point to its left and right neighbors in the doubly linked circular leaf list. A global pointer leaf[H] points to some node in the leaf list. The left and right pointers are NIL if the node is not a leaf.

The PRUNE operation will place MARKs on nodes. These are the same MARKs as are placed by DECREASE_KEY.

In what follows, we assume that r < n, otherwise the entire heap should be deleted without the need for pruning. An algorithm for pruning a single node is given below. To prune r nodes, we simply call this algorithm r times.

```
PRUNE(H)  {
    x = leaf[H] // Get a pointer to a leaf node from the main leaf pointer.
    if (x == "big pointer")  // x is the root node with minimum value
       x = x.right // In the leaf list, skip the min node
      endif
   Remove x from the leaf list.

   y = PARENT[x]
   if (y == NIL)    // x is in the root list
       Remove x from the root list and discard x.
   else
      Remove x from the child list of y and discard x.
      Decrement the rank of y.
      CASCADING-CUT(H,  y)      // Given in the CLRS text
   endif
   }
```

Charging for the PRUNE of a single node

Aside from the leaf list maintenance considerations (next section), the entire pruning process will actually cost O(r) plus the cost for the resulting cascading cuts that might occur. How can we assign charges that will leave us with an amortized cost of O(1) for PRUNE? We cannot charge the pruning process or we end up with the process being O(r) which is still O(n).

Thus, we charge an INSERT operation for the cost of pruning the node that it inserted. That is, to prune a node X, we charge the INSERT that placed X into the Fibonacci heap. A node can only be pruned once, so its corresponding INSERT can only be charged once for this. Further, since the prune of X may result in a cascading cut, the INSERT of X also becomes responsible for the node Z where that cascading cut ends. That means that the INSERT of X will pay for: placing the mark on Z; a subsequent cut of Z; and later, a link of Z. In addition, the INSERT of X is charged O(1) if the node that becomes Z's parent in a link operation, had been in the leaf list. But these are all O(1) operations and the INSERT of X can only be charged once each in this fashion.

Note also that when the pruning operation results in the cascading cut of a node W, we charge the operation that is responsible for W. That is the operation charged for the operation whose cascading cut stopped at W. That will be either a DECREASE-KEY or as described above, the INSERT of the node the prune of which caused a cascading cut that stopped at W. So, in summary, for cascading cuts, either DECREASE_KEYS or INSERTS will be charged, but they are only charged O(1) in this fashion and only one time, so their total amortized costs are O(1).

Finally, the PRUNE(H, r) operation itself is charged just O(1) – this is for testing if $r \geq n[H]$, and if so then deleting the entire heap by making the root pointer NIL.

Changes to existing operations

The following operations need to be modified since we now have a leaf list to maintain:

- UNION - link the leaf lists of the two Fibonacci Heaps that are being unioned. This adds O(1) to the cost of UNION, making the total UNION cost to be O(1).

- INSERT – will also add the inserted node to the leaf list, adding O(1) to the cost of INSERT, both actual and amortized.

- DELETE_MIN - when two nodes of rank zero are linked, one leaf is removed from the leaf list. Whatever operation is charged for the link is also charged for that leaf list removal. This means that if Y becomes the child of X, then the operation charged is the one responsible for Y (for instance, the INSERT of Y) and that "Y" operation is the one paying for X to be removed from the leaf list. This adds O(1) to the cost of the operation that is responsible for Y.

- DECREASE_KEY - recall that a DECREASE_KEY is responsible for the costs associated with two nodes - the node X where the DECREASE_KEY is being done, and the node Z where the cascading cut ends. When X is placed into the root list, nothing new happens to it in so far as the leaf list is concerned. But, when Z loses its second child, and goes to the root list, it may also be that Z’s parent Y becomes a leaf, in which case Y is added to the leaf list and the operation (DECREASE_KEY or INSERT) responsible for Z is charged O(1). In the text there is a CUT procedure that is called to make the cut of a node – that will need to be modified to add a node to the leaf list if the cut removes the last child of the node. Again this is an O(1) addition to the cost.