Robert Barnes

20407 DSA 2013b

Maman 17

1) Given a red-red-black tree of height k, what are the minimum and maximum number of internal nodes?

A) We show that the minimum number of nodes is $2^k - 1$ via induction. For $k = 0$, we have a tree with only `nil` leaves and no internal nodes, and $2^0 - 1 = 0$. Assume x is some internal node in a rrb tree and $bh(x) = k$ and that the number of internal nodes is $2^b h(z) - 1$ for any $bh(z) < k$. By property four of a rrb tree, we have that all simple paths from a node to a leaf have the same black height. Thus the black height of `x.right` and `x.left` is either k or $k - 1$ depending on wether they are red or black, respectively.
We assume two children and since we are looking for the minimum we assume they are both black. Then we have by the induction hypothosis that the number of internal nodes is:

$$2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2 \cdot 2^{k-1} - 1 = 2^k - 1$$

B) We show that the maximum number of internal nodes is $2^{3k} - 1$. Assume that we have a complete rrb tree x with all black nodes such that $bh(x) = k$. Then this tree has k levels of black internal nodes, and one level of black leaves. We add two levels of red nodes *after* each level of internal black nodes. Since the root must be black, and by property three there can be no more than two levels of red nodes prior to a level of black nodes, then this is the largest valid rrb tree such that $bh(x) = k$. The height of this tree is clearly $3k$ not including the last row of black leaves, and thus this complete binary tree has $2^{3k} - 1$ internal nodes.

2) Given an accumulator tree A which represents a binary tree T:

A) Write an $O(n)$ algorithm which checks if A represents a BST.

We make the initial call with `isBST( root, -inf, +inf, 0)`. Basically, whenever we “turn a corner” we compare against the grandparent value, and when we go straight we compare against the parent.

```
int isBST(diffNode root, int min, int max, int accum)
if ( root == NULL )
    return true;
```

```
int key = root.key + root.accum + accum;

// assumes only unique values in bst
if ( key >= max || key <= min )
    return false;

return isBST(root.l, min, key, accum + root.accum)
    && isBST(root.r, key, max, accum + root.accum);
```

B) Write $O(h)$ routines for finding, inserting and deleting in an accumulator tree which is also a BST.

.

```
findNode( root, key )
accum = 0;
while ( root != NULL )
    accum = accum + root.accum
    key = key - accum
    if ( key == root.key ) return root
    if ( key < root.key ) root = root.left
    else root = root.right
return NULL
```

.

```
insertNode( T, x )
accum = 0;
root = T.root
if ( root == T.nil )
    T.root = x
    return T.root
while (1)
    x.key = x.key - root.accum
    if ( x.key < root.key )
        if ( root.left == T.nil )
            root.left = x
            x.p = root
            return x
        else
            root = root.left
    else
```

```
        if ( root.right == T.nil )
            root.right = x
            x.p = root
            return x
        else
            root = root.right
```

.

```
// replaces node u with node v without modifying children
replace(T, u, v)
if ( u.p == NULL )
    T.root = v
else if ( u = u.p.left )
    u.p.left = v
else
    u.p.right = v
if ( v != NULL )
    v.p = u.p
```

.

```
// assumes a valid accum bst
deleteNode( T , x )
// left child NULL - replace with right child
if ( x.left == NULL )
    x.right.accum = x.right.accum + x.accum
    replace(T, x, x.right)
// right child NULL - replace with left child
else if ( x.right == NULL )
    x.left.accum = x.left.accum + x.accum
    replace(T, x, x.left)
// two children
else
    // find successor using regular bst successor function since accum
    // doesn't affect the ordering of the keys.
    y = successor(x)
    // walk up the tree
    for ( temp = y; temp != x; temp = temp.p )
        y.key = y.key + temp.accum
    x.key = y.key
    // delete successor which has only one child
    deleteNode(T, y)
```

C) Show that we can make a red-black accumulator tree. Implement the rotation routines.

.

```
rotate-left(T, x)
    y = x.right
    x.right = y.left
    if ( y.left != T.nil )
        y.left.p = x
        y.left.accum = y.left.accum + y.accum
    y.p = x.p
    if ( x.p == T.nil )
        T.root = y
    else if ( x == x.p.left )
        x.p.left = y
    else
        x.p.right = y
    temp = x.accum
    x.accum = -y.accum
    y.accum = y.accum + temp
    y.left = x
    x.p = y

rotate-right(T, x)
    y = x.left
    x.left = y.right
    if ( y.right != T.nil )
        y.right.p = x
        y.right.accum = y.right.accum + y.accum
    y.p = x.p
    if ( x.p == T.nil )
        T.root = y
    else if ( x == x.p.left )
        x.p.left = y
    else
        x.p.right = y
    temp = y.accum
    y.accum = x.accum + y.accum
    x.accum = -temp
    y.left = x
    x.p = y
```

We can see from the rotation routines, and from the routines in part B, that in order to maintain the accum field we need only access a fixed number of descendants of the nodes involved, or the parent of the relevant node up to the root of the tree. Thus by Theorum 14.1 we have that we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

3) Given a set of n houses built at varying distances from the beach, storms come along and cause cumulative damage to all the houses at less than a certain distance from the beach.

We make the following assumptions:

1. That new houses may be built in a previously damaged area
2. That multiple houses may not be built at the same distance
3. We assume that all houses at less than the given distance all sustain the same amount of damage, and that all house beyond the given distance suffer zero damage.

We use a red-black accumulator tree similar to that from question 2. The keys for the tree are the distances. We have three additional fields, one for the house value, one for damage incurred at that distance, and a negative damage field. The damage field is the accum field, the value and negative damage fields are constant and never change once written.

A) `INSERT(dist, value)` Insertion of a new house works similar to in question 2, except that instead of adjusting the key value, we accumulate negative damage in a separate field as we go down the tree to find the new houses' insertion point. Thus the distance and value are inserted as is, and the `accum` field is initially set to 0. The rotation routines from question 2 are sufficient for any balancing, since once a node is inserted the only value that must be maintained is the accum field.

B) `DECREASE-VALUE(D, A)` We assume that a house exists at the given distance. If we can't make this assumption, then we simply prepend a search which returns either the node at the given distance, or what would be it's immediate predecessor. Given a tree `T`, we start at the root.

.

```
decrease(x, d, a )
    if ( d == x.key )
        x.accum += a
        if ( x.right != nil )
            x.right.accum -= a
    else if ( d < x.key )
```

```
        decrease(x.left, d, a)
    else
        x.accum += a
        decreaseRight( x.right, d, a )

decreaseRight(x, d, a)
    if ( d == x.key )
        if ( x.right != nil )
            x.right.accum -= a
    if ( d < x.key )
        x.accum -= a
        decrease(x.left, d, a)
    else
        decreaseRight(x.right, d, a)
```

This is the same routine written with a loop.

```
decrease-value( T, d, a )
    x = T.root
    wentRight = false
    while ( d != x.key ) {
        if ( d < x.key ) {
            if ( wentRight )
                x.accum -= a
            x = x.left
            wentRight = false
        } else {
            if ( !wentRight )
                x.accum += a
            x = x.right
            wentRight = true
        }
    }
    if ( !wentRight )
        x.accum += a
    if ( x.right != nil )
        x.right.accum -= a
```

There are three base cases whose behavior vary depending on if the previous action was to go left or go right. If the current key is the one we're looking for, then we want the damage to affect the current node and all nodes less than it. If we last went right, then there is an ancestor whose accum field already modifies the value of the current node. Since we don't want to modify nodes greater than the current node we decrement the right child by the same amount. If our last action wasn't to go right, then `x` is the local root. If we go left, and our last action was to go right, then we decrement accum so as to remove it's affect on larger values. If we go right, we add to accum to affect smaller values.

C) `ASSESS-DAMAGE(dist)` This works similarly to the find node routine from question 2.

   ASSESS-DAMAGE(dist) accum = 0; while (root != NULL) accum = accum + root.accum if (key == root.key) return root.value - accum + root.negDmg if (key < root.key) root = root.left else root = root.right

   ```
   return NULL
   ```

4) We support the requested operations using a combination of two order statistic trees and max-priority queue implemented with a binary tree heap. The first OS tree has the key's as keys, the second one uses the insertion time as the key. Counts are maintained in the priority queue. In the key OS tree we have two additional fields, a pointer to an entry in the time OS tree and a pointer to an entry in the priority queue. Similarly in the time OS tree we have a pointer to the corresponding entry in the key OS tree. The max priority queue uses the count as it's key, maintains a copy of the key it relates to in the key OS tree, and a pointer to the relevant entry in the time OS tree. We use an auxilary function, find to see if the value already exists in the tree. The `_p` indicates a pointer.

.

```
INSERT(S,k)
    y.key = k
    x = osFind(keyTree, k)
    if ( x != keyTree.nil )
        y.count_p = x.count_p
        pqIncrementKey(y.count_p)
    else
        newCount.count = 1
        newCount.key = k
        y.count_p = pqInsert(S.countPq, newCount)
    newTime.time = now()
    newTime.key_p = y
    y.time_p = newTime
    osInsert(timeTree, newTime)
    osInsert(keyTree, y)
```

The osFind, pqInsert, and osInsert functions each take $O(\lg n)$ time. `pqIncrementKey` takes $O(\lg n)$ in the worst case when all the count keys are identical and the entry is at the bottom of the heap. Thus `INSERT` takes $O(\lg n)$ time.

```
DELETE-MIN(S)
    y = osMinimum(S.keyTree)
    pqDecrementKey(S.countPq, y.count_p)
    osDelete(S.timeTree, y.time_p)
    osDelete(S.keyTree, y)
```

It takes $O(\lg n)$ time to locate the minimum element. `pqDecrementKey` decrements the count by 1 and calls `heapify`. If the count reaches 0, it deletes the entry. In the worst case, when all count keys are identical and the decremented count key is at the top of the heap this takes $O(\lg n)$ time. Deletion from an OS tree takes $O(\lg n)$ time, thus overall the function takes $O(\lg n)$ time.

```
DELETE-OLD(S)
    y = osMinimum(S.timeTree)
    pqDecrementKey(S.countPq, y.key_p.count_p)
    osDelete(S.keyTree, y.key_p)
    osDelete(S.timeTree, y)
```

This is completely analogous to `DELETE-MIN` and operates in the same time.

```
MAX-COUNT(S)
    y = pqMax(S.countPq)
    return y.key
```

Returning the max of a max priority queue takes $O(1)$ time.

```
COUNT(S, i)
    y = osSelect(S.keyTree, i)
    return y.count_p.count
```

`osSelect` takes $O(\lg n)$ time.

```
COUNT-OLD(S, t)
    y = osSelect(S.timeTree, i)
    return y.key_p.count_p.count
```