Self Adjusting Binary Search Trees

ECE 1762 Algorithms and Data Structures
Spring Semester, 2004 — U of Toronto

1 The weighted dictionary problem

In the weight dictionary problem, we are given a set of elements $S = \{x_1, x_2, \ldots, x_n\}$ and a weight, that is, an access frequency, w_i is associated with every element $x_i \in S$. The goal is to make accesses to high–frequency elements faster than to low–frequency elements without sacrificing the efficiency of operations such as INSERT, SEARCH and DELETE.

If keys and frequencies are given off–line and do not change throughout the execution of the algorithm, an optimal search tree, known as *static tree*, can be constructed. More precisely, for a tree T let d_i be the depth of item x_i. Then $P = \sum_i w_i d_i$ is called the weighted path length of T. It is not hard to construct an optimum tree, *i.e.* a tree that minimizes the weighted path length, in time $O(n^3)$ with the use of dynamic programming.

A search tree for a set S yields directly a prefix code for the symbols $x_1, x_2, \ldots, x_n$ over the binary alphabet (*go left*, *go right*). In view of the noiseless coding theorem it is therefore not surprising that the normalized weighted path length $\overline{P} = \sum_i (w_i/W) d_i$, where $W = \sum_i w_i$, is strongly related to the *entropy* $H = \sum_i (w_i/W) log(W/w_i)$ of the access frequency distribution.

In the *dynamic* case of the weighted dictionary problem there are two important facets: the underlying set S may change and the access frequencies of the elements of S might also change.

What kind of time bounds can we hope for? The entropy observation above says that the best we should hope for the *ideal search* time of element i is $O(\log(W/w_i))$. It can be actually proved that *inserting* an element has an $O(\log(W + w)/min(w^+, w, w^-))$ time bound, where w^+, w^- are the weights of the two neighbors of the new element x, and w is the weight of x.

The data structure that achieves an amortized time within a constant multiple of the information theoretic lower bound is **Splay Trees**, a *self–organizing* data structure. In self–organizing data structures, an item x is moved closer to the "entry point" [1] whenever it is accessed. This makes future accesses to the element cheaper. In this way, the elements of S compete for the places close to the entry point in the data structure and high–frequency elements are more likely to be there.

Note, however, that we do not maintain any explicit frequency counts or weights in the data structure; rather, we hope that the data structure self–organizes itself. The worst–case behavior of a single operation can always be horrible. On the other hand, the amortized behavior achieves the infomation theoretical lower bounds. In the presentation that follows, we will prove a weaker $O(\log n)$ amortized time per operation result for our operations.

For a detailed discussion on the weighted dictionary problem refer to [2].

2 Splay trees

The self-adjusting tree (or *splay tree*) is an elegant data structure invented by Sleator and Tarjan in 1984 [1]. The presentation below is adapted from Tarjan's monograph *Data Structures and Network Algorithms.*

A *splay tree* is an ordered binary tree: for every node x, every element in the left subtree of x is $\leq x$, and every node in the right subtree of x is $\geq x$. Yet there is *no explicit balance condition* on the tree. In

[1] The entry point for a binary search tree is obviously the root.

fact, they are *not* balanced trees, in any real sense of the word. So don't assume, as you read below, that the height of the tree is always $O(\log n)$ — there are cases where it will be $\Theta(n)$. This means that the worst–case complexity of an operation on the tree may in fact be $\Theta(n)$. But the *amortized* complexity of each operation will be $O(\log n)$. In other words, the cost of doing a sequence of m balanced tree operations will be $O(m \log n)$.

We implement all of the operations using the following simple subroutine called a `splay` (hence the name of the data structure). Splay takes as input a node in the tree, and then repeats a sequence of rotations at the parent of x. Each of these rotations will move x to the place where its parent was formerly located. The sequence of rotations is performed — each rotation moving x up one level in the tree — until finally x becomes the root of the tree. Although such a sequence of rotations will move x to the root of the tree, for the analysis we actually need to perform these rotations in pairs.

```
    SPLAY(x)
        while x is not the root do
            let p be the parent of x
            if p is the root then
(1)             rotate at p and stop
                    — now x is at the root
            else if x and p are both right children or left children then
(2)             rotate at parent(p)
(2)             rotate at p
                    — now x is where its grandparent was
            else { x is a left child and p a right child, or vice versa }
(3)             rotate at p
(3)             rotate at the new parent of x (its former grandparent)
                    — now x is where its grandparent was
```

The next few pages contain figures with the individual cases numbered above as (1), (2) and (3). Fig. 1 shows a SPLAY operation on node number 5. Note that the subtree rooted at node 5 is constantly moving up the tree. We also note that the initial "unbalanced" tree appears more "balanced" at the end of the SPLAY procedure; paths in the tree tend to be cut in half.

We will refer to each rotation or pair of rotations (in cases (1), (2) or (3)) as a *step* of the splay. If the path from x to the root of the tree has length h, then there are there are $\lceil h/2 \rceil$ steps, and exactly h rotations.

First, a simple observation.

Observation: The effect of a splay is to bring a node x up to the root of the search tree. If the tree was ordered before the splay, it is ordered after the splay.

Since the procedure uses only rotations, and rotations preserve the ordering of the keys in the tree, we know that splaying does not affect the order.

We implement the other operations as we would on any other binary search tree, with one or two modifications.

locate(x): Search for element x as in any ordered search tree. After we locate x in the tree, we *splay* the tree at x.

insert(x): Again, insert an element as in any ordered search tree by searching to the location where x should be, and then attaching x as a leaf. After inserting an element in the tree, we also *splay* the tree at at the newly inserted node.

delete(x): Delete an element just as in any other search tree. If the node to be deleted is a leaf, just remove it. If the node is an internal node, search for the successor or predecessor y of x, and remove y from the tree. Then replace x by y.

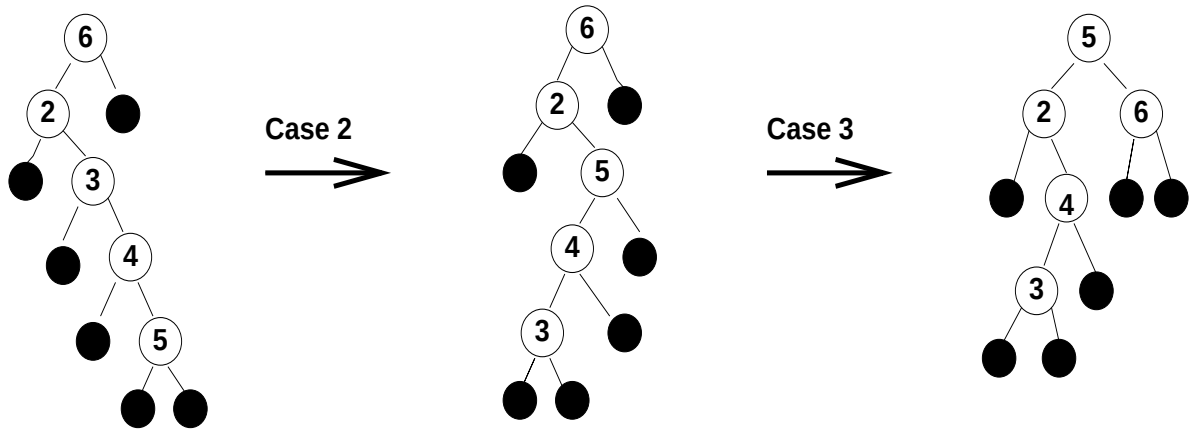

Figure 1: An example of the SPLAY procedure

As in the previous operations, we also splay the tree in a deletion. When a leaf is deleted, we splay the tree at the parent of the deleted leaf. If we replace an element by its successor or predecessor, then we splay the tree at the parent of the successor or predecessor.

An important use of balanced trees in implementing algorithms is in the representation of ordered lists which require fast random access (and perhaps local modifications). Often when the description of an algorithm says "list" or "ordered sequence", the implementation may call for a balanced tree[2]. In such cases, we may want to implement operations which **join** (concatenate) two sequences, or **split** one sequence into two. Self-adjusting trees support these two operations simply in amortized time $O(\log n)$. (Compare the simplicity here with the complexity of a join on red–black trees as described in CLR, problem 14–2, pp.278)

SPLIT(t, x)
— *this operation takes a tree t and splits it*
— *into three trees: one containing all elements $< x$,*
— *one containing all elements $> x$ and one containing just x*
locate(x)
splay(x)
— *this brings x to the root*
return left(x), right(x), x

JOIN(t_1, t_2, x)
— *every element in t_1 is $< x$*
— *every element in t_2 is $> x$*
— *join returns a tree containing x*
— *and all elements in t_1 and t_2*
make x into a node
left$(x) \leftarrow t_1$

[2]This happens in computational geometry and in several graph algorithms. See Tarjan's monograph for examples of the latter.

$\text{right}(x) \leftarrow t_2$

Another important property of splay trees is that they are "self-adjusting". In this case, we mean that those items which are accessed most frequently tend to move closer to the root. Subsequent accesses on these items will require less time (an important property when used for *on line* algorithms). For example, we can compare splay trees with static search trees — where the structure does not change, although their structure may be optimized for the expected sequence of queries. Then it can be shown that splay trees perform asymptotically at least as well as any static search trees (Static Optimality). Moreover, it has been conjectured that splay trees in fact perform as well as *any* adaptive search tree algorithm which uses rotations to restructure the tree. Partial results have been obtained on a weaker conjecture (the Dynamic Finger Conjecture, which compares splay trees with dynamic finger trees), but the more general conjecture is still open (the Dynamic Optimality Conjecture).

Overview

We claim is that these operations each have an $O(\log n)$ *amortized* time bound. We will concentrate first on the splay (since everything else depends on that). We will prove this using the accounting method. Analysis of the other operations will then be very simple.

At first, you can think of the splay as a heuristic. If it takes a long time to access an element x in the tree, then we make use of the time that we've spent in finding x by bringing it up to the root of the tree, while carrying some of x's children along with it. Now the next time we access the element x, or an element that was close to x, there's a better chance that it is nearer to the root, and will take less time to access.

3 Splaying a tree

Every node x in the tree has a *weight* — written $\text{WT}(x)$ — which is the number of nodes in the subtree rooted at x (including x itself). The weight defined here is related to the weight in Section 1 but you should not worry how and why.

Definition: Rank of a Splay Tree. Define the RANKof x to be $\lceil \log \text{WT}(x) \rceil$. I'll write this as $\text{RANK}(x)$.

Note please that the rank, as defined here, is different from the rank function we use in skewed heaps.

Now, since some of the operations will be more expensive than others, we want to guarantee that we always have some credits stored up for future use. We will keep these credits on the nodes of the tree, and will distribute them to the nodes according to the following credit invariant.

Credit Invariant: There are always $\text{RANK}(x)$ credits on node x, for every node x in the tree.

Note that this invariant is necessary only for bookkeeping purposes in the proof. It doesn't appear in the implementation of the data structure. We're going to charge 1 credit for each of the one or two rotations in a step of the splay operation on line (1), (2) or (3). We want to show that, for each splay, we need only $O(\log n)$ additional credits to maintain the invariant.

Now every step of the splay rearranges subtrees of the tree. In particular, when we splay at a node x, the weight of x changes (increases) at every step as it moves up the tree. Every time the weight of x doubles, its rank increases by 1. Since this can happen at most $\log n$ times, we'll need this many credits just to maintain the number of credits on x. But in general, we don't need many more credits than this for the entire sequence of rotations.

Claim: For *each step* in the splay operation, the number of additional credits that we need is at most $3(\text{NEW RANK}(x) - \text{OLD RANK}(x))$, plus maybe one additional credit for the last step. Any other credits we need will be taken from the account on the tree itself.

After each step in the splay operation, the rank of x may or may not change. We get a sequence of ranks for x, as this node moves up the tree. Let's say that there are k steps in this splay operation. Write the

sequence of ranks of x as RANK_0, RANK_1, RANK_2, ..., RANK_k, where RANK_0 is the original rank of x before the splay, and RANK_k is the final rank of x when it has become the root at the end of the splay. Now if the claim is true, then the total number of credits needed is at most

$$\begin{aligned} 1 + 3(\text{RANK}_k - \text{RANK}_{k-1}) \quad &+ \quad 3(\text{RANK}_{k-1} - \text{RANK}_{k-2}) \\ &\vdots \\ &+ \quad 3(\text{RANK}_2 - \text{RANK}_1) \\ &+ \quad 3(\text{RANK}_1 - \text{RANK}_0) \end{aligned}$$

and after cancelling out like terms ("telescoping"), this is just

$$1 + 3(\text{RANK}_k - \text{RANK}_0) \ .$$

But RANK_k is just the rank of x after the entire splay is done. Since x is moved to the root of the tree, every node in the tree is below x at this point, and its weight is n. That means that its rank is $\lceil \log n \rceil$. On the other hand, the original rank of x (RANK_0) is certainly ≥ 0. This means that

$$1 + 3(\text{RANK}_k - \text{RANK}_0) \leq 1 + 3 \log n \ .$$

Assuming the claim above, the cost of a splay operation (in credits) is at most $3 \log n + 1$. Therefore, the splay has amortized time complexity of $O(\log n)$. So we will assign an amortized cost of $3 \log n + 1$ credits to each splay.

4 Amortized complexity of the other operations

Assume that the claim is true, and that the splay requires only $O(\log n)$ amortized time. Here's how the other operations are analyzed. In each case, we increase the amount of work that can be paid for with one credit — but only by a constant factor. In this way, the amortized cost of any one of the operations will be no more than the splay. In some cases we will need extra credits to maintain the invariant.

locate(x)**:** To locate an element x, we will first search down a path, and then splay up that path. So the number of rotations and the number of comparisons are about equal. Since the amortized complexity of splaying is $O(\log n)$, there are no more than $O(\log n)$ comparisons (amortized!) as well. If we allowing 1 credit to pay for a rotation and a comparison, then the entire operation has $O(\log n)$ amortized time.

insert(x)**:** This is similar to locate, except that we must add a new leaf to the tree. This causes a small problem. Although just adding a leaf to the tree is a constant time ($O(1)$ time) operation, it *does not* cost a constant number of credits! By adding a new leaf, every node on the path from that leaf to the root has now gained weight (increased in weight by 1). And *some* of these nodes may increase in rank. So we need to add additional credits to these which increase in rank.

We make the following observations(similar to those for Skewed Heaps):

1. There are at most n nodes on any path, and and the weight of a node is always strictly smaller than the weight of its parent.
2. A node increases in RANK only if it had weight 2^k before adding x to the tree, and weight $2^k + 1$ after adding x to the tree. (Its rank increases from $k = \lceil \log 2^k \rceil$ to $k + 1 = \lceil \log 2^k + 1 \rceil$.) Here k is a number between 0 and $\log n$.
3. Therefore, there are at most $\log n$ nodes that increase in *rank*, and these increase in rank only by 1.

So when we attach a new leaf to the tree, we need to supply at most an extra $\log n$ credits to these nodes which have increased in rank. As noted above, these credits may not be used for the actual insertion. We are charging them to the insertion now, and distributing them to nodes on the access path to this new element. If this node causes the tree to become more unbalanced, these credits will be used in the splays which follow.

These extra credits costs an additional $\log n$. But the net amortized complexity is still $O(\log n)$.

delete(x)**:** We locate x and its successor as above, using only $O(\log n)$ credits. Now, replacing x by its successor y doesn't affect the credit scheme. We take any credits on x and place them on y.

Removing y from the tree *does* change the weight of all of the ancestors of y, but these ancestors all *decrease* in weight, and hence the ranks can never increase. So we don't need to supply any extra credits. In fact, some credits might be freed up. Since we splay at the successor or predecessor node – which is taken to be a leaf, and which is therefore at a greater depth in the tree than x — cost of the splay can be made to cover the cost of finding x and its successor as above.

The net cost is $O(\log n)$ amortized time.

join(t_1, t_2, x)**:** This is easy. We are given two splay trees t_1 and t_2 and an element x, where all elements in t_1 are less than x and all elements in t_2 greater than x. We also have $|t_1| + |t_2| + 1 \leq n$.

To join the two trees, merely make x the root, t_1 its left subtree, and t_2 its right subtree. To maintain the credit invariant, we just place $\leq \log n$ credits on the new root x. None of the other nodes have changed weight, so no other violations have occured. The cost is just $\log n$ credits, which is $O(\log n)$ amortized time.

split(t, x)**:** To split a tree t at a node x, we splay at x, and then remove x. The left and right subtrees still maintain the credit invariant and we are done. Since splaying is $O(\log n)$ amortized time, so is the split.

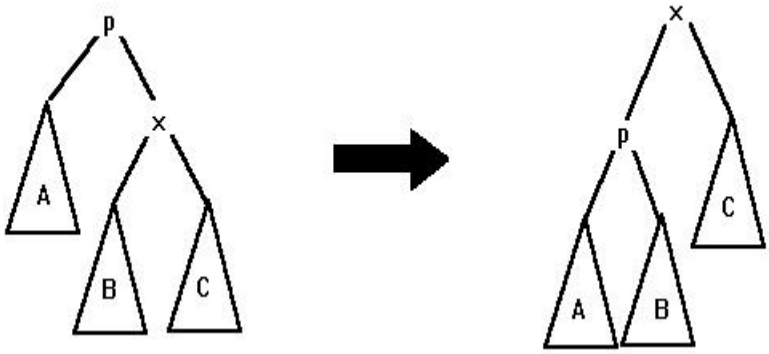

Figure 2: Case 1 of the SPLAY procedure

5 Proof of the Claim

Now, all we have to do is prove the claim. We're looking now at only one step in the splay operation (lines (1), (2) or (3)). Let's say that x is the node we are splaying at, p its parent and g its grandparent. We show that we only need to allocate

$$3(\text{NEW RANK}(x) - \text{OLD RANK}(x)) \text{ credits}$$

for any one step, except maybe for the last step (case (1), where p is the root), and in this case we may need $3(\text{NEW RANK}(x) - \text{OLD RANK}(x)) + 1$ credits.

Case 1. Suppose we're in the case at line (1) where we do just one rotation. This is the last step in the splay (Fig. 2).

$$\begin{aligned} \text{OLD RANK}(x) = \log(1 + |B| + |C|) &\leq \text{NEW RANK}(x) = \log(2 + |A| + |B| + |C|) \\ \text{OLD RANK}(p) = \log(2 + |A| + |B| + |C|) &\geq \text{NEW RANK}(p) = \log(1 + |A| + |B|) \end{aligned}$$

p can only decrease in rank, so we won't worry about it.

Since x already has $\text{OLD RANK}(x)$ credits, we need no more than $(\text{NEW RANK}(x) - \text{OLD RANK}(x))$ credits to add to x. This uses only $\frac{1}{3}$ of the credits we've allocated. Now we need one more to pay for the rotation; but note that we can't necessarily assume that we can use the remaining $2(\text{NEW RANK}(x) - \text{oldrank}(x))$ credits here, because it is possible that $\text{NEW RANK}(x) = \text{OLD RANK}(x)$. This is why we have allocated one extra credit for the last step. We can use that to pay for the rotation done here.

Case 2. Here x and its parent p are both left children, or both right children(Fig. 3). Taking all credits from these nodes, we have

$$\text{OLD RANK}(x) + \text{OLD RANK}(p) + \text{OLD RANK}(g)$$

credits available. We need

$$\text{NEW RANK}(x) + \text{NEW RANK}(p) + \text{NEW RANK}(g)$$

credits.

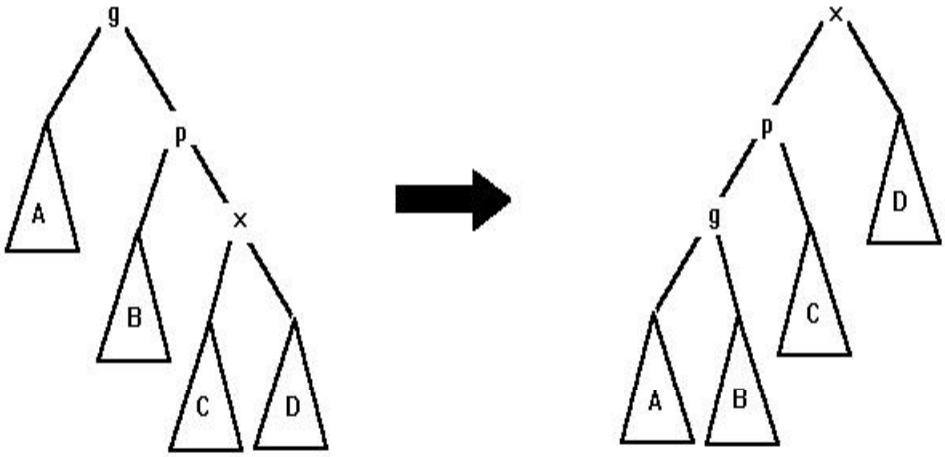

Figure 3: Case 2 of the SPLAY procedure

Since $\text{NEW RANK}(x) = \text{OLD RANK}(g)$

$$\text{NEW RANK}(p) + \text{NEW RANK}(g) - \text{OLD RANK}(x) - \text{OLD RANK}(p) \tag{1}$$

credits to what we have. But we note that

$$\begin{aligned} \text{NEW RANK}(p) &\leq \text{NEW RANK}(x) \\ \text{NEW RANK}(g) &\leq \text{NEW RANK}(x) \\ \text{OLD RANK}(p) &\geq \text{OLD RANK}(x) \end{aligned}$$

(by just observing where the subtrees A, B, C, D are moved), so the number of credits we need in this step (Eq. 1) is no larger than

$$2 \text{ NEW RANK}(x) - 2 \text{ OLD RANK}(x)$$

(maximize the positive quantities and minimize the negative quantities in Eq. 1). That uses only $\frac{2}{3}$ of the credits we have allocated for this step, and allows us to redistribute the credits properly.

Now, we need just one more credit to pay for the rotations we do here. If

$$\text{NEW RANK}(x) - \text{OLD RANK}(x) > 0$$

then we can use the other $\frac{1}{3}$ of the credits we have allocated to this step, and we are done. But if $\text{NEW RANK}(x) - \text{OLD RANK}(x) = 0$, we need to find this credit elsewhere.

So assume that

$$\text{NEW RANK}(x) = \text{OLD RANK}(x) \ .$$

Since

$$\text{NEW RANK}(x) = \text{OLD RANK}(g) \ ,$$

this implies that

$$\text{OLD RANK}(x) = \text{OLD RANK}(p) = \text{OLD RANK}(g)$$

and, of course, all of these are the same as the new rank of x. Remember that the rank is related to the log of the weight. So this means that, before the step, more than half of the elements in this tree are below x (in the trees C, D). If not, the weight of g would be twice that of x, and its rank would be larger.

So look at the weights:

$$\text{OLD WEIGHT}(x) + \text{NEW WEIGHT}(g) \leq \text{NEW WEIGHT}(x) \ .$$

On the left hand side of this equation, we have all of the elements in the subtrees A, B, C, D. On the right, is the entire subtree below x after the step is performed — also A, B, C, D. But we already said that more than half of the nodes are under x before the step. This means that after the step, fewer than half of the nodes are under g. In other words, this equation say that

$$(> \tfrac{1}{2} \text{ of all nodes here}) + \text{NEW WEIGHT}(g) \leq \text{all nodes here} \ ,$$

and so $\text{NEW WEIGHT}(g) < \frac{1}{2}$ of all nodes in this subtree, which is $< \frac{1}{2}\text{OLD WEIGHT}(g)$. But then

$$\begin{aligned} \log(\frac{1}{2}\text{OLD WEIGHT}(g)) &= \log(\text{OLD WEIGHT}(g)) + \log\frac{1}{2} \\ &= \log(\text{OLD WEIGHT}(g) - 1 \\ &= \text{OLD RANK}(g) - 1 \ . \end{aligned}$$

So g's rank has *decreased* by at least one.

Now by the assumptions, the rank of x does not change. p's rank can't increase at all, so we don't have to add any credits to p. The rank of g *does* decrease by at least 1, and this frees up one credit which pays for the rotations at this step.

So when $\text{NEW RANK}(x) > \text{OLD RANK}(x)$, we pay for the step using some of the credits we've allocated to pay for (1) rotations and (2) maintaining the credit invariant. When $\text{NEW RANK}(x) = \text{OLD RANK}(x)$, we can do the step for free, using credits in the account at g to pay for the operation while still maintaining the credit invariant.

Case 3. Now x is a left child and its parent p a right child, or vice versa(Fig. 4). This is very similar to case 2.

The only problem occurs when the new rank of x is the same as the old rank of x, as above. When this happens, the analysis is a little different.

Suppose that

$$\text{OLD RANK}(x) = \text{NEW RANK}(x),$$

so that we have 0 credits to do this step. It is still the case that this implies

$$\text{OLD RANK}(x) = \text{OLD RANK}(p) = \text{OLD RANK}(g)\ ,$$

so more than half of the nodes in this subtree are originally under x. But after the rotations, we see that some of the nodes under x are now under g (B) and some are under p (C). There are two possibilities.

Either: One of the nodes g and p now has $> \frac{1}{2}$ of the weight, which means that the other has $< \frac{1}{2}$ the weight and therefore drops in rank. That frees up one credit.

Or: If g and p have about the same weight, then *both* of them have dropped in rank (since each node has lost half of its weight), and we still get at least one credit freed. So, in this case too, we can take the cost of the rotations from the credits already on the tree.

Think of it this way. Let's say that the entire subtree has a bit more than 2^k nodes, and x, p, g originally had rank k. That means that most of these 2^k nodes are under x. Now after the rotation, we split these nodes between g and p. Either one of them has about 2^k nodes — which implies that the other has much less (and so a rank lower than k) — or both of them have about 2^{k-1} nodes — so both of them have a lower rank.

To be sure that you understand the proof, you should try filling in the rest of the details of this case.

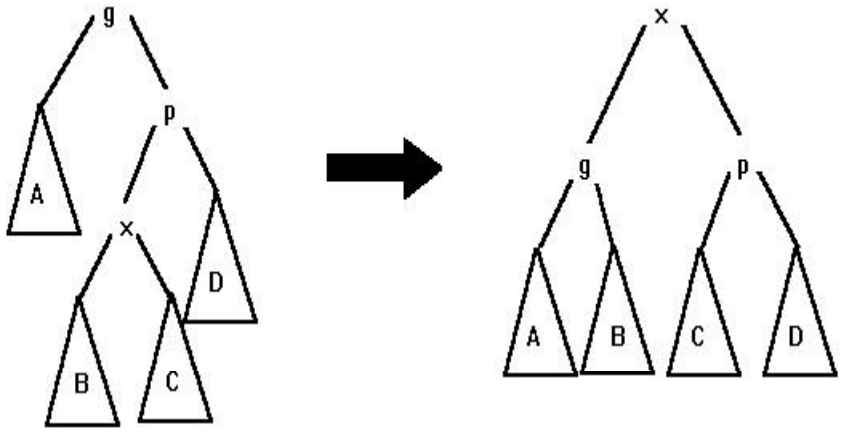

Figure 4: Case 3 of the SPLAY procedure

6 Discussion

So now that you've read the proof, what does it say? Informally, we allocate $O(\log n)$ credits for the splay operation. We use a credit immediately whereever the tree is nearly balanced — this happens at most about $\log n$ times. These credits pay for (1) rotations here and (2) places where the rotations make the tree more unbalanced. *But we can make the tree more unbalanced at most about* $\log n$ *times.* When we make the tree more unbalanced, we also place credits on the affected nodes, so that later, when we come back through here we can use those credits to rebalance the tree at this point.

On the other hand, where the tree is unbalanced, the rotations will make the tree more balanced. And we pay for this rebalancing with the credits we placed on these nodes earlier (when we first made the tree unbalanced here). That is: these credits come from (1) insertion and (2) rotations that disturbed the balance at this point.

References

[1] D.D.Sleator, and R.E.Tarjan, "Self–Adjusting Binary Search Trees," in *Journal of the ACM,* 32(3), pp.652-686, 1985.

[2] K.Mehlhorn, and A.Tsakalidis, "Data Structures," in *Handbook of Theoretical Computer Science,* Chapter 6, Vol. A, pp.303–341, Elsevier Science Publishers, 1990.