Washington University in St. Louis

Engineering

Self-Adjusting Search Trees

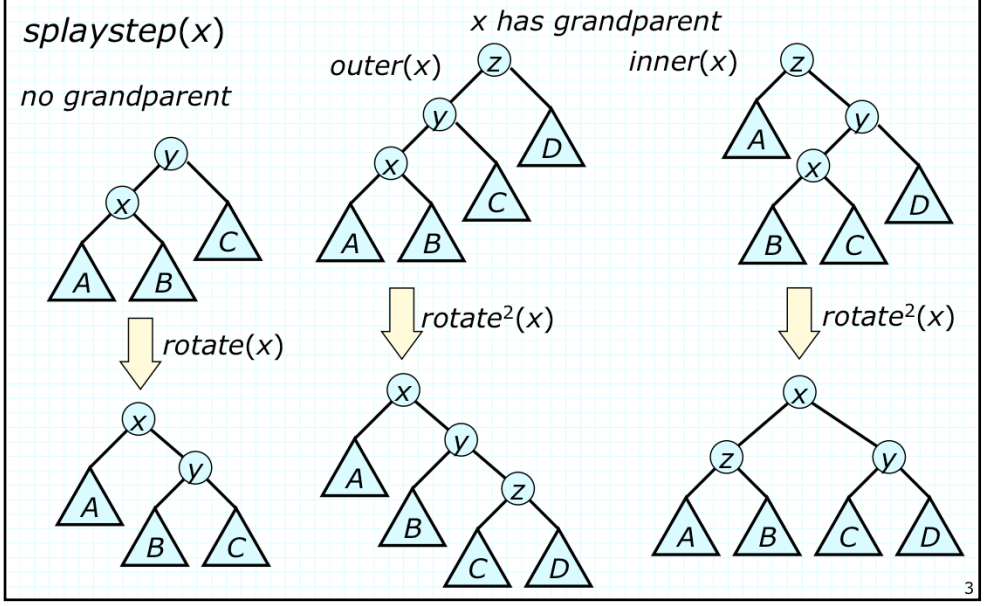
Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst

Self-Adjusting Binary Trees

- Self-adjusting binary search trees restructure search tree after each operation
  - » requires no explicit balance condition
  - » m operations take $O(m \log n)$ time
  - » in some contexts provides better overall performance than balanced binary trees
- Restructuring operation is called *splay*
  - » moves one vertex x to root of the tree by a sequence of rotations
  - » the rotations reduce depth of each vertex u, by roughly half the original depth of the nearest common ancestor of x and u

Illustration of Splay Steps



Implementation of Splay

```
sset function splay(item x);
    if x = null ⇒ return null fi;
    do p(x) ≠ null ⇒ splaystep(x) od;
    return x;
end;
procedure splaystep(item x);
    item y,z;
    if p(x) = null ⇒ return fi;
    y := p(x);
    if  p²(x) = null ⇒ rotate(x)
    |  p²(x) ≠ null ⇒ rotate²(x)
    fi;
    return
end;
```

last step of splay

each moves descendants of *x* one step closer to root; depth of original descendants roughly halved

Implementing Self-Adjusting BSTs

```
item function access(keytype k, sset s);
    if s = null ⇒ return null fi;
    do  k < key(s) and  left(s) ≠ null ⇒ s := left(s)
      | k > key(s) and  right(s) ≠ null ⇒ s := right(s)
    od;
    s := splay(s);
    if k = key(s) ⇒ return s;
     | k ≠ key(s) ⇒ return null;
    fi;
end;
[sset, sset] function split(item i, sset s);
    sset s1,s2;
    splay(i);
    s1,s2 := left(i),right(i); p(s1),p(s2) := null;
    left(i), right(i) := null;
    return [s1,s2];
end;
```

time bounded by number of splay steps

ditto

```
procedure insert(item i, sset s);
    item x;  x := s;
    do  key(i) < key(x) and  left(x) ≠ null ⇒ x := left(x)
      | key(i) > key(x) and  right(x)  ≠ null ⇒ x := right(x)
    od;
    if  key(i) = key(x) ⇒ return null
     | key(i) < key(x) ⇒ left(x) := i;
     | key(i) > key(x) ⇒ right(x) := i;
    fi;
    p(i) := x;
    splay(i);
end;
```

time bounded by number of splay steps

```
procedure delete(item i, sset s);
    item j;
    if left(i) ≠ null and right(i) ≠ null ⇒
        j := left(i);
        do right(j) ≠ null ⇒ j := right(j) od;
        swapplaces(i,j);
    fi;
    if left(i) = null ⇒ left(i) ↔ right(i) fi;
    p(left(i)) := p(i);
    if  i = left(p(i)) ⇒ left(p(i)) := left(i)
     | i = right(p(i)) ⇒ right(p(i)) := left(i)
    fi;
    splay(p(i));
    left(i),right(i),p(i) := null;
end;
```

time bounded by number of splay steps

Analysis of Self-Adjusting BSTs

- Objective is to show that sequence of m operations, on trees with total of n nodes takes $O(m \log n)$ time
- Use a credit scheme to account for running time
  - » all operations but *join* include a *splay*, so can account for their running time by bounding the time for all *splays*
  - » splay time is proportional to number of splay steps, so account for running time of splay by "spending" one credit per step
  - » allocate up to $C \lg n$ credits per splay and per *join* (C to be determined)
  - » credits not needed to pay for performing an operation are "stored" in the tree and can be used to "pay" for later steps
- To ensure we have enough credits on hand to pay for future operations, maintain the credit invariant
  - » keep $rank(x)$ credits on hand for each node x, where

$$rank(x) = \lfloor \lg(\#\text{ of descendants of } x) \rfloor$$

8

- Balanced trees need fewer credits than unbalanced trees, so splays generally "release" credits
- *Lemma 4.2*. Splaying a tree with root v at a node u while maintaining credit invariant requires at most $3(rank(v)-rank(u))+1$ new credits

Proof. The credits are divided among the different splay steps

- a splay step at node x with parent y and grandparent z is allocated $3(rank(z)-rank(x))$ credits
- a splay step at a node x with a parent y but no grandparent is allocated $3(rank(y)-rank(x))+1$ credits

Let *rank* and *rank'* be the rank functions before and after the step

- *Case 1*. x has no grandparent: this is last step, and extra credit pays for it; number of credits needed to maintain invariant is

$$(rank'(x) - rank(x)) + (rank'(y) - rank(y))$$
$$= rank'(y) - rank(x) \le rank(y) - rank(x)$$

which is one third of the available credits

- *Case 2*. $x=left(left(z))$ or $x=right(right(z))$: if $rank(z)=rank(x)=k$ we get no new credits for this step, but $rank'(z)<k$, so we can spend an existing credit while still maintaining the invariant

  if $rank(z)>rank(x)$, the number of credits needed to maintain the invariant is

  $$\begin{aligned}&(rank'(x)-rank(x)) + (rank'(y)-rank(y)) + (rank'(z)-rank(z))\\ &\quad = rank'(y) + rank'(z) - rank(x) - rank(y)\\ &\quad \le 2(rank(z) - rank(x)) < 3(rank(z) - rank(x))\end{aligned}$$

  releasing at least one extra credit to pay for the step
- *Case 3*. $x=left(right(z))$ or $x=right(left(z))$; if $rank(z)=rank(x)=k$ we get no new credits for this step, but either $rank'(z)<k$ or $rank'(y)<k$, so we can spend a credit while maintaining invariant

  if $rank(z)>rank(x)$, the number of credits needed to maintain the invariant is

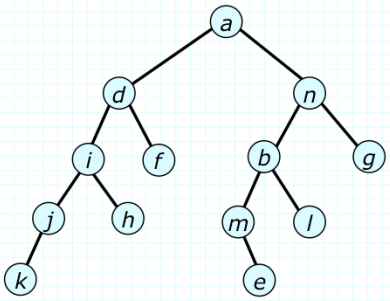
  $$\begin{aligned}&(rank'(x)-rank(x)) + (rank'(y)-rank(y)) + (rank'(z)-rank(z))\\ &\quad = rank'(y) + rank'(z) - rank(x) - rank(y)\\ &\quad \le 2(rank(z) - rank(x)) < 3(rank(z) - rank(x))\end{aligned}$$

  releasing at least one extra credit to pay for the step ■

- Lemma implies each splay requires $\leq 3\lfloor \lg n \rfloor + 1$ credits
  - » number of credits needed for an *insert* is this number plus the number of new credits needed to maintain the credit invariant, after the new item is inserted but before the splay is done
  - » the only nodes whose ranks can increase are those on the path from the root to the inserted node that have exactly $2^k - 1$ descendants before the operation (where $k \in [0..\lfloor \lg n \rfloor]$)
  - » there can be at most $\lfloor \lg n \rfloor + 1$ of these, so the total number of credits required for an *insert* is at most $4 \lfloor \lg n \rfloor + 2$
- The *join* operations requires at most $\lfloor \lg n \rfloor$ credits
- All other operations require no credits beyond those used by the *splay*
- *Theorem 4.1*. The total time required for a sequence of m sorted set operations on n vertices, using self-adjusting binary search trees is $O(m \log n)$

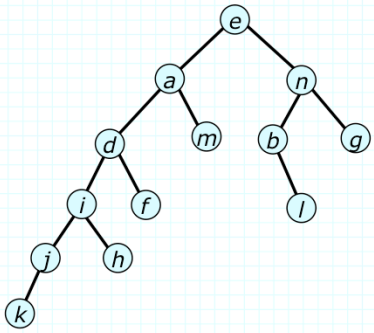
Exercises

1. Show the tree that results from performing a splay at node *e* in the self-adjusting BST shown below.



What is the largest increase in depth of any node? What is the largest decrease?

The new tree appears below



The largest increase in depth is 2 (all nodes in the subtree of d see an increase in depth of 2). The largest decrease is for node e, which goes from depth 4 to 0.

2. Prove the following statement. Let u be a descendant of x in a self-adjusting BST. If a splay is performed at x, then $D(u) \leq d(u) - (d(x)/2) + 1$, where $d(u)$ is the depth of u before the splay and $D(u)$ is the depth of u after the splay (recall that the depth of a node in a tree is the length of the path from the root).

If we look at the definition of a splay step on slide 3, we observe that each splay step except possibly the last, moves the descendants of the "splay node" at least one step closer to the root. The splay node moves two steps closer. So if $d(x)=2k$, there are k splay steps that reduce move u at least one step closer to the root, so

$$D(u) \leq d(u) - k \leq d(u) - (d(x)/2) + 1$$

The above inequality is also true if $d(x)=2k+1$.

Now, suppose that z is the nearest-common ancestor of x and u and that a splay is performed at x. Show that,

$$D(u) \leq d(u) - (d(z)/2) + 3$$

We'll assume that u is not a descendant of x, since that case has already been covered earlier. Now, let's first consider the effect of the splay on the depth of z. As long as x remains a descendant of z, the splay steps at x have no effect on the depth of z. The first splay step that makes z a descendant of x increases the depth of z by up to 2. Subsequent splay steps (except possibly the last) each decrease the depth of z by 1. Therefore,

$$D(z) \leq d(z) - (d(z)/2) + 3$$

Since u is a descendant of z, the splay steps will affect u in essentially the same way as z. The splay step that makes z a descendant of u can increase the depth of u by 2, but each subsequent step (except possibly the last) decreases its depth by 1. So, $D(u) \leq d(u) - (d(z)/2) + 3$.

3. Consider the self-adjusting BST in problem 1. How many credits are needed to satisfy the credit invariant before the splay at node *e*?

   Leaves in the tree require no credits. Nodes with a total of 2 or 3 descendants require 1, nodes with 4 to 7 require 2 and nodes with 8 to 15 require 3. So, in the tree from the previous problem, we need a total of 13.

   How many are needed after the splay?

   14

   How many credits were needed to pay for the splay?

   2

   How many new credits are allocated to the splay in the analysis?

   Node e has a rank of 0 initially and the tree root has a rank of 3, so 10 credits are allocated.

   How many "surplus credits" does this leave us with?

   10– ((14–13)+2)=7

4. Consider a highly unbalanced BST where each non-root node is the right child of its parent and there are n=15 nodes altogether. How many credits are needed to satisfy the credit invariant?

   2 nodes require 1 credit, 4 nodes require 2 credits, 8 require 3. This gives us 2+8+24=34.

   How many are needed after a splay at the one leaf node?

   The final tree has x at the root and all other nodes in its left subtree. Each internal node in this left subtree has a leaf as its left child. In this tree one node has a rank of 1, two have a rank of 2 and five have a rank of 3. The rest are leaves with rank 0. So the total number of credits needed is 1+4+15=20, a reduction of 13.

   How many credits were needed to pay for the splay?

   There are 7 splay steps, so 7 credits are needed to pay for the splay.

   How many new credits are allocated to the splay in the analysis?

   10 credits are allocated.

   How many "surplus credits" does this leave us with?

   10– ((20–34)+8)=16