CSE 589 Part VIII

Those who fall in love with practice without science are like a sailor who enters a ship without a helm or a compass, and who never can be certain whither he is going.

Leonardo da Vinci

Reading

D.D. Sleator and R.E. Tarjan "Self-Adjusting Binary Search Trees", Journal of the ACM 32 (1985), pp. 652-686

Skiena, Section 8.1.1

Weiss, "Data Structures and Algorithm Analysis in C++", 2nd Edition, Section 11.5

Lewis and Denenberg, "Data Structures and Their Algorithms", Section 7.3

Splay Trees – award winning

Danny Sleator of Carnegie Mellon University and Bob Tarjan of Princeton University have been awarded the 1999 Kanellakis Theory and Practice Award by the ACM for their invention of the Splay Tree data structure. This prestigious award "honors specific theoretical accomplishments that have had a significant and demonstrable effect on the practice of computing".

"Splay Trees are a simple, elegant and efficient self-adjusting data structure for storing ordered sets. They've found their way into the Linux and NT kernels, the GCC compiler, FORE routers, and a myriad of other important applications."

Another implementation of a Dictionary: use binary search tree

A binary search tree is a binary tree with a key at each node u such that, for each node with left subtree L and right subtree R, we have

- the key at u is greater than all the keys in L
- the key at u is less than all the keys in R

Splay Trees

self adjusting binary search trees

guarantee that any m consecutive tree operations starting from an empty tree take O(m log n) time, where n is the maximum number of elements in tree at any time.

Another way of saying this is that the average cost of an operation (averaged over all the operations) is O(logn). Note: this makes no assumption about the sequence of accesses.

Call this amortized logarithmic cost.

Amortized Analysis

in general, when a sequence of m operations has a total worst-case running time O(m f(n)), we say the amortized running time of each operation is O(f(n))

Definitions of amortize (from real dictionary)

1 : to provide for the gradual extinguishment of (as a mortgage) usually by contribution to a sinking fund at the time of each periodic interest payment

2 : to amortize an expenditure for <amortize intangibles> <amortize the new factory>

Idea

Expensive operations can only happen infrequently.

If you ever pay O(n) to access an element, you must move it.

List Update

Given a linked list L, sequence of requests σ to elements in list, serviced online

Cost of accessing an element x in L = position of element in list.

Also, can transpose two adjacent list elements at a cost of one unit.

Goal: minimize total cost of accessing elements in σ

Best offline strategy : no efficient algorithm for computing it known.

Sleator & Tarjan

Move-To-Front heuristic always within a factor of 2 of optimal offline (2-competitive)

Transpose and Frequency Count can be arbitrarily bad.

Application of these ideas here:

if a node is too deep, after access it is moved to root

nodes on path to accessed node (which are also deep) should be moved to less deep position

want restructuring to have side effect of balancing tree.

Splay trees

simpler data structure than those that explicitly maintain balanced tree (don't explicitly maintain any height or balance info)

excellent amortized running time -- guarantees do not depend on any assumption about distribution of accesses

practically even better.

- Locality of reference -- when a node is accessed it is likely to be accessed again in the near future.

Crucial Operation called "Splay"

to "splay" something means to spread it out and flatten it

Splay (K, T), where K is a key and T is a BST modifies T so that

- it remains BST with same set of items
- new tree has K at root if K was already in T
- if K was not in T, root contains key that would be predecessor (or successor) of K if K were in tree.

Call this operation "splaying the tree around K"

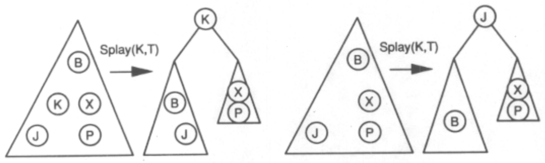


Figure 7.16 Effect of *Splay*(K, T). If key K is in tree T, it is brought to the root, otherwise a key in T that would neighbor K in the dictionary ordering is brought to the root.

Rest of dictionary operations

Given an implementation of splay operation, rest of dictionary operations can be implemented easily:

- Lookup(K,T)
- Insert(K, Info, T)
- Delete (K,T)
- Concat(T1, T2):
  - input -- two BST's T1 and T2 such that every key in T1 is less than every key in T2
  - output -- BST containing all keys in either T1 or T2

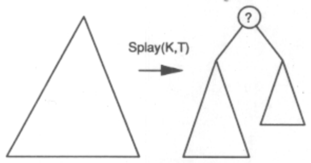


Figure 7.17 Implementation of *LookUp*(K, T) with the aid of *Splay*. Splay the tree around K, then see if K is at the root.

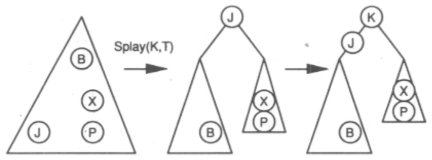


Figure 7.18 Implementation of *Insert*(K, T) with the aid of *Splay*. Splay the tree around K, then make K the root.

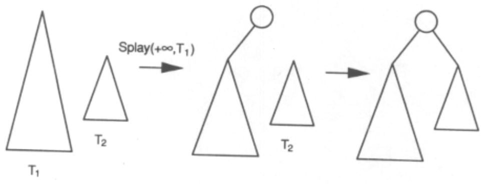


Figure 7.19 Implementation of *Concat*(T_1, T_2) with the aid of *Splay*. Splay the first tree around $+\infty$, then make the second tree the right subtree of the root.

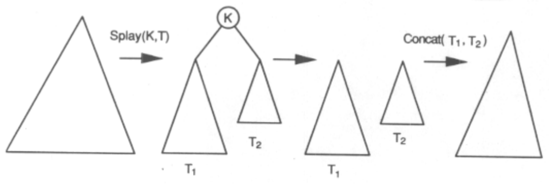


Figure 7.20 Implementation of *Delete*(K, T) with the aid of *Splay* and *Concat*. Splay the tree around K, then concatenate the two subtrees of the root.

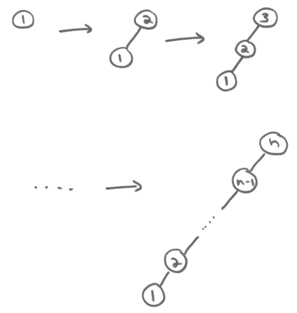
How to implement Splay (K,T)?

First search for K in usual way, remembering search path by stacking it. Let P be last node inspected. (If K is in tree, K is in node P, otherwise, P has empty child where search for K terminated)

Idea 1: perform single rotations bottom up -- return along path from P to root carrying out rotations that move P up the tree, so that when splay is completed, P will be the new root.

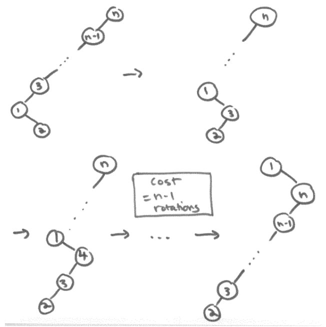
Bad Example

Insert(1) , Insert(2), ..., Insert(N)

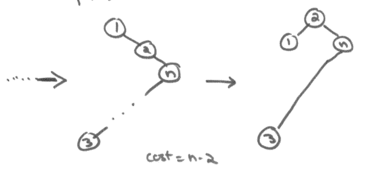


Lookup(1), Lookup(2), ..., Lookup(n)

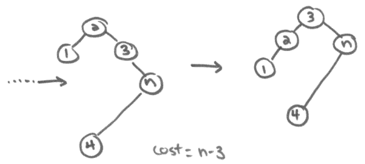
Lookup(1)



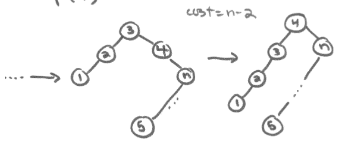
Lookup(2)



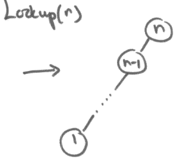
Lookup(3)



Lookup(4)



Lookup(n)



$\Omega(n^2)$ time total

Can repeat.

Better proposal

- Case I: P has no grandparent (that is, Parent(P) is the root). Perform a single rotation around the parent of P, as illustrated in Figure 7.21 or its mirror image.
- Case II: P and Parent(P) are both left children, or both right children: Perform two single rotations in the same direction, first around the grandparent of P and then around the parent of P as shown in Figure 7.22 or its mirror image.
- Case III: One of P and Parent(P) is a left child and the other is a right child: Perform single rotations in opposite directions, first around the parent of P and then around its grandparent, as illustrated in Figure 7.23 or its mirror image

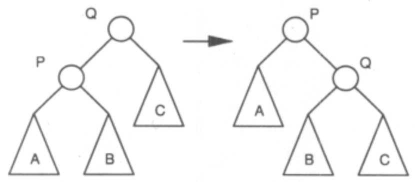


Figure 7.21 Rotation during splay, Case I: P has no grandparent.

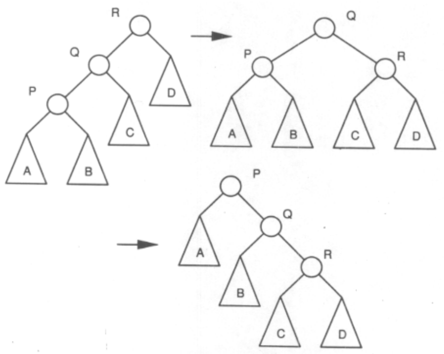


Figure 7.22 Rotation during splay, Case II: P and its parent are both left children.

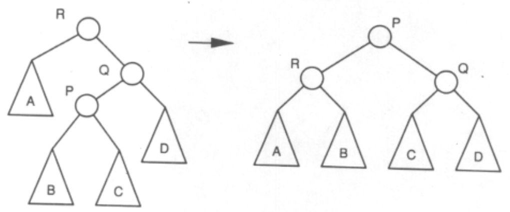


Figure 7.23 Rotation during splay, Case III: P is a left child and its parent is a right child.

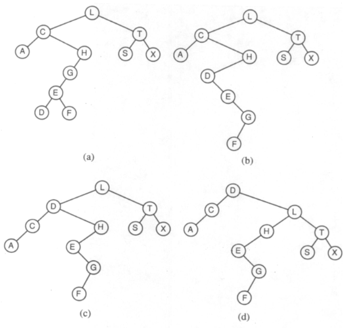


Figure 7.24 Splaying a tree around D. (a) Original tree; D is a left child of a left child, so Case II applies. (b) After applying the rotations of Figure 7.22 at D, E, and G. D is now a left child of a right child, so Case III applies. (c) After applying the rotation of Figure 7.23 at D, H, and C. D now has no grandparent, so Case I applies. (d) After applying the rotation of Figure 7.21 at D and L.

The big theorem

effect of rotations mysterious, subtle and actually not that well understood.

Splay Tree Theorem:

- Any sequence of m dictionary operations on a splay tree that is initially empty and never has more than n modes uses O(m log n) time in worst case. Therefore, the operations have amortized O(log n) time.

Analysis subtle: need to show that time "saved" while performing low-cost operations can be "saved up for use" during time-consuming operations

We'll spend the next several slides proving this.

Idea of Proof

use banking metaphor.

- You will give me O(log n) dollars per dictionary operation
- I will either save it or use it to pay for the rotations I do.
- If I do a Splay operation that consists of d basic steps, I will pay you d dollars.
- My only source of money is what you give me.
- If I can show that I never fail to pay you for the operations, then we can argue that the total cost of m operations is O(m log n)

More on idea of proof

I will save the money you give me in "bank accounts" at each node in the tree, each of which will be required to maintain a certain minimum balance.

Minimum balance in a bank account at node will be related to its number of descendents (money invariant below)

payment I make to you for each operation:

- some will come out of bank accounts at nodes
- some will come from new investment (what you paid me)

Definitions

for any vertex v in the tree:

- w(v): the weight of v = number of descendents of v (including v itself)
- r(v) : the rank of v = floor($\log_2$ w(v))

The Money Invariant: each vertex v has r(v) dollars in its account at all times.

Using money during splay process

Money is used in two ways during splay process

- we pay for the time used. $1 per operation (zig, zag, zigzig, zagzag, zigzag, zagzig)
- since the shape of the tree changes as the splay is carried out, we may have to add money to the tree, or redistribute the money already in the tree, in order to maintain the Money Invariant everywhere.

The Investment Lemma

It costs at most 3 floor(log(n)) + 1 new dollars (this is the money you pay me) to splay a tree with n nodes while maintaining the Money Invariant everywhere.

We'll do the proof of this later.

Proof of Splay Tree Theorem using Investment Lemma

Any dictionary operation on a tree T with at most n nodes costs O(log n) new dollars:

- Lookup(K,T) costs only what the splay costs, which is O(log n)
- Insert(K,Info, T) costs for splay plus what must be banked in new root to maintain invariant , for a total of O(log n)
- Concat(T1, T2), where T1 and T2 have at most n nodes, costs the splay at T1, plus what must be banked at root to make T2 a subtree, for a total of O(log n).
- Delete(K,T) costs splay of T, plus the costs to concatenate resulting subtrees, which is O(log n)

Using Investment Lemma O(m log n) dollars are enough to pay for the whole thing.

To prove investment lemma

Two observations about ranks

- rank of a node >= rank of any of its descendents
- Rank Rule: if a node has 2 children of equal rank, then its rank is greater than that of each child
- Proof of Rank Rule: node x has two children, u and v

$$w(u) >= 2^{r(u)}$$
$$w(v) >= 2^{r(v)}$$
$$w(x) > w(u) + w(v) >= 2^{r(u) + 1}$$

$$\log(w(x)) > r(u) + 1$$
$$\text{floor}(\log(w(x)) >= r(u) + 1$$

Another Lemma

consider single step of splay operation (case I, II or III)

- let r(P) denote the rank of P before the operation
- let r'(P) denote the rank of P after the operation

Cost of Splay Steps Lemma

- A splay step involving node P, the parent of P and possibly the grandparent of P can be done with an investment of at most 3(r'(P)-r(P)) new dollars, plus one more dollar if this was the last step in the splay

defer proof for a moment

Cost of Splay Steps Lemma => Investment Lemma

Let $r^{(i)}(P)$ denote the rank of P after i steps of the splay operation have been carried out.

By CSSL, the total investment of new money needed to carry out splay is <=

3(r'(P) - r(P))
+ 3($r^{(2)}$(P) - r'(P))
+ ...
+ 3($r^{(k)}$(P) - $r^{(k-1)}$(P)) + 1
where k is the number of steps needed to bring P to root.
But $r^{(k)}$(P) is the rank of the original root <= floor (log n)
Total <= 3($r^{(k)}$(P) - r(P)) + 1 <= 3 floor (log n) + 1

Proof of Cost of Splay Steps Lemma

Case 1: P has no grandparent (last step)

1 extra dollar pays for time to do rotation

# new dollars need to add
= r'(p) - r'(q) - r(p) - r(q)
= r'(q) - r(p)
<= r'(p) - r(p)

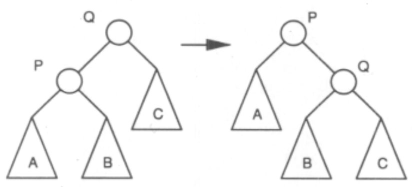


Figure 7.21 Rotation during splay, Case I: P has no grandparent.

Proof of Cost of Splay Steps Lemma, cont.

Case 2: zig zig

to maintain money invariant need to add
r'(p) + r'(q) + r'(r) - r(p) - r(q) - r(r)
= r'(p) + r'(r) - r(p) - r(q)
<= 2(r'(p) - r(p))

Case (a): r'(p) > r(p). Then there are $ left over to pay for rotations
Case (b): r'(p) = r(p)

CSSL, Case 2(b) r'(p) = r(p)

Fact 1: r'(p) = r(r)

Fact 2: r'(r) < r(p)

r'(r) <= r'(p) = r(p)
can't have r'(r) = r(p) by Rank Rule on middle tree

Fact 3: r'(q) <= r(q)

r'(q) <= r'(p) = r(p) <= r(q)

=> can move r's money to p, p's money to r and q's money to q to maintain invariant and still have $ leftover to pay for rotation.

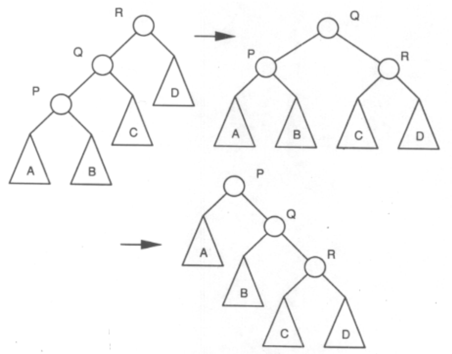


Figure 7.22 Rotation during splay, Case II: *P* and its parent are both left children.

Proof of Cost of Splay Steps Lemma, cont.

remaining cases require similar analysis

Summary on Splay Trees

one of the most popular ways to implement binary search tree

- relatively simple (splaying can also be done top down)
- no extra fields needed
- excellent temporal locality properties
- m>= n operations costs O(m log n), good amortized time complexity

on negative side

- all those rotations add a significant constant factor
- worst case for a single operation can be costly

big open problem: dynamic optimality conjecture