CS 840 Winter 2010
Self Organizing Linear Search: Unit 5

The same questions can be asked in binary search trees.

Given a sequence of access queries, what is the best way to organize the search tree [reference: Cormen Leiserson, Rivest and Stein]

Worst case behaviour: $\Theta(\lg n)$ upper and lower bounds.

Model: Searches must start at root of a binary search tree. Charge for each node inspected.

Static optimal binary search tree: easy dynamic programming.

Given: p_i = prob. of access for $A_i\ (i = 1, n)$
q_i = prob. of access for value between A_i and $A_{i+1}\ (i = 0, n)$
$[p_0 = p_{n+1} = 0]$

$T[i, j] =$ root of optimal tree on range q_{i-1} to q_j

$C[i, j] =$ cost of tree rooted at $T[i, j]$; this cost is the probability of looking for one of the values (or gaps) in the range times the expected cost of doing in that tree.

Algorithm computes *R's* and *C*'s by increasing size, i.e. by increasing value of *(j-i)*.

So $T[i,i] = i$ [Initialization, i is the only key value in the range, so it must be the root]
$C[i,i] = q_{i-1} + p_i + q_i$ [the probability of searching in this tree with one internal node]

If r = root; L = left subtree; R = right subtree;
W[tree] = probability of being in tree = probability of accessing root.

Then
C[tree rooted at r] = *W*[tree] + *C*[L] + *C*[R]

It will clearly be handy to have *W[i,j]*, the probability of accessing any node $q_{i-1}, \dots, q_j$ or $p_i, \dots, p_j$.

So $W[i, j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$

These are easy to compute in $\Theta(n^2)$ time by computing $W[i, j+1]$ as $W[i, j] + p_{j+1} + q_{j+1}$.
In our description, if $T[i, j]$ is defined, take that value, otherwise compute it recursively as:

$T[i, j] = i$

$C[i, j] = W[i, j] + q_{i-1} + C[i+1, j]$ {initialize with i as root}

for $r = i+1$ to j *do* {if r is a better root, take it .. this line will be editted later}

$r\cos t = W[i,j] + C[i,r] + p_r + C[r+1,j]$

if $r\cos t < C[i,j]$ *then*

begin

$C[i, j] = r\cos t$

$T[i, j] = r$

end

Clearly we do this by memoing, if a value of $T[i, j]$ or $C[i, j]$ has been computed ... use it; otherwise compute it and remember it.

Hence an $O(n^3)$ algorithm as each of the $O(n^2)$ values takes $O(n)$ time to compute given values on smaller ranges.

But one more thing –
Do we have to check entire *i,j* range for a root?

Check just from $T[i, j-1]$ to $T[i+1, j]$

Lemma: $T[i, j]$ cannot be to the right of $T[i+1, j]$ (similarly not to the left of $T[i, j-1]$)

Proof: Induction on range size (Basis 2 node trees)

Hence key loop is "shorter" if we just go between the subtree roots.

And indeed

Runtime of improved to $O(n^2)$ [working through some series telescope … essentially a $\sum_{i=1}^{n} (Cost_{i+1} - Cost_i) = Cost_{n+1} - Cost_0$]

Fact – after 30 years still best

$\Theta(n^2)$ time and space

Now known polynomial time $\Theta(n^{2-\epsilon})$ space method.

How good is the tree? Clearly the expected cost can be as high as lg *n*.

Definition: The entropy of a probability distribution $x_1, .. x_k$ is given by

$$H(x_1,...,x_k)=\sum_{i=1}^{k} x_i \lg(1/x_i)$$

Hence the entropy of our distribution is $H(p_1,...,p_n,q_0,...q_n)=\sum_{i=1}^{n} p_i \lg(1/p_i)+\sum_{i=0}^{n} q_i \lg(1/q_i)$

Theorem: The expected cost, *C*, of the optimal binary search tree satisfies the inequalities:

$$C \geq H - \lg H - \lg e + 1$$

and

$$C \leq H + 3.$$

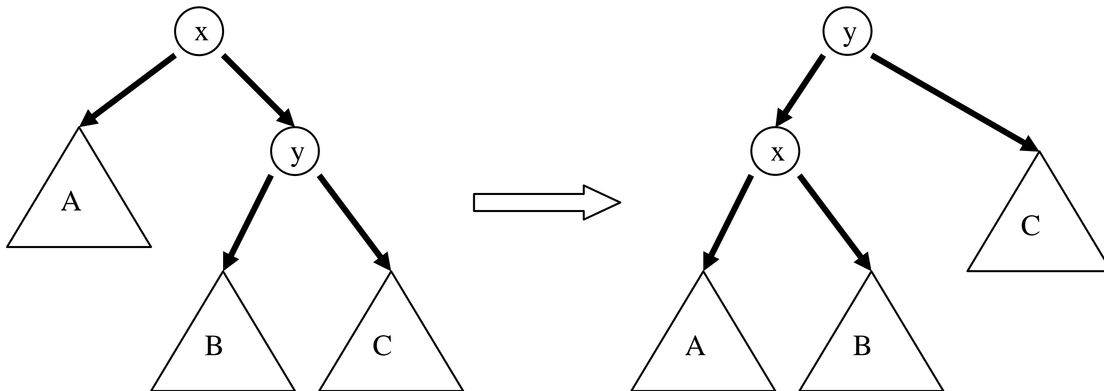
It is interesting to note these bounds can essentially be achieved for the same set of *p*'s and *q*'s by simply permuting the probabilities.

We also note that the optimal tree is expensive to compute and ask whether it can be approximated more cheaply. A natural greedy algorithm would be to put the key with the largest probability at the root. This can be a serious problem, even if all q_i values are 0. Suppose, for example, that all p_i values are virtually the same, but that $p_i > p_{i+1}$. Then the heuristic suggested will give a tree rooted at node 1, in which each node (except the last) has a right child but no left child. The search cost will be linear; whereas the optimal tree is balanced, with expected cost lg *n*.
A different approach is to try to balance the load at each node. That is to choose as root a node so that less than half the weight of the tree is in the left subtree and less than half is in the right. Clearly this is not always possible. For example, consider the case in which p_1=.2 and p_2=.2 and the *q* values are also .2. We must choose one of the internal nodes as root, so the weight of one subtree will be .6 while the other is .2. An achievable version of this approach is to choose as root the internal node that minimizes the weight of the largest subtree, or that minimizes the difference between the weights of the two subtrees. Ties are broken arbitrarily. Both approaches are effective. *The bounds quoted above for the optimal tree also apply to these approximate solutions.* Furthermore, while is may seem tat the minmax approach is the better of the two, it is easy to construct example in which this is not the case.

It is natural to ask to ask whether a binary search tree can be made to adapt to an unknown distribution, or to do well in a competitive sense with an offline approach.

The first results in this direction come from Allen and Munro (JACM 1978). Given the success of the simple exchange algorithm in the case of linear search given a fixed independent distribution of inputs, one may be inclined to try swapping the requested value with its parent as shown below when y is requested.

Unfortunately, this appealing scheme can be quite a disaster. If all key values have the same



probability of request, then all binary trees are generated with the same probability. Unfortunately, most binary trees are very bad and the average search cost becomes $\Theta(n^{1/2})$.

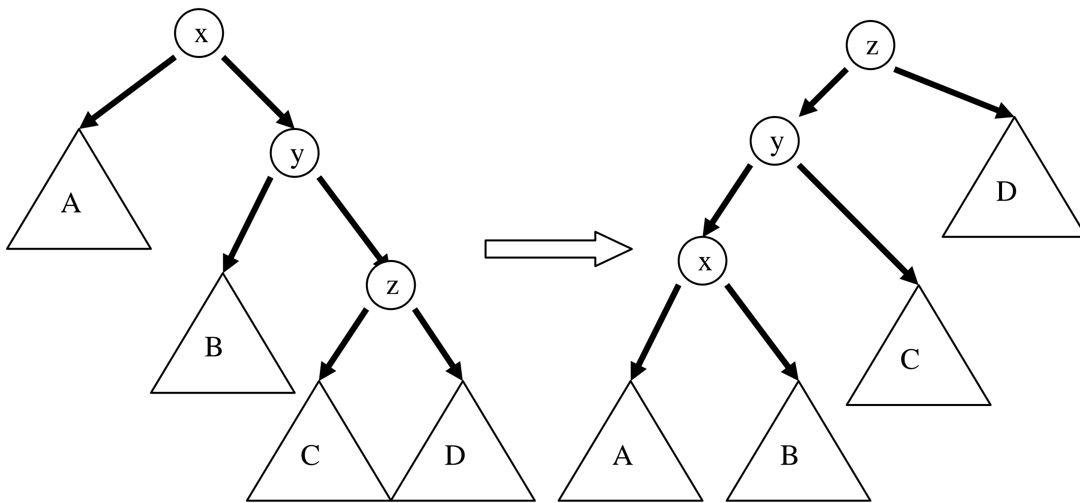
[Note that this distribution is very different from what we would get by inserting elements into a binary tree in random order, in which situation the average search cost $\Theta(\lg n)$.]

The next thing to try is to move the element requested to the root by a sequence of swaps with its newly acquired parent. This meets with much more success, and the expected cost ids within a factor or 2 ln 2 [about 1.38..] of the optimal static tree. The method, however, can be shown to do poorly on an amortized basis. The key reason for this is that on access, the requested element is moved to the root and each element on its path is moved down one level.

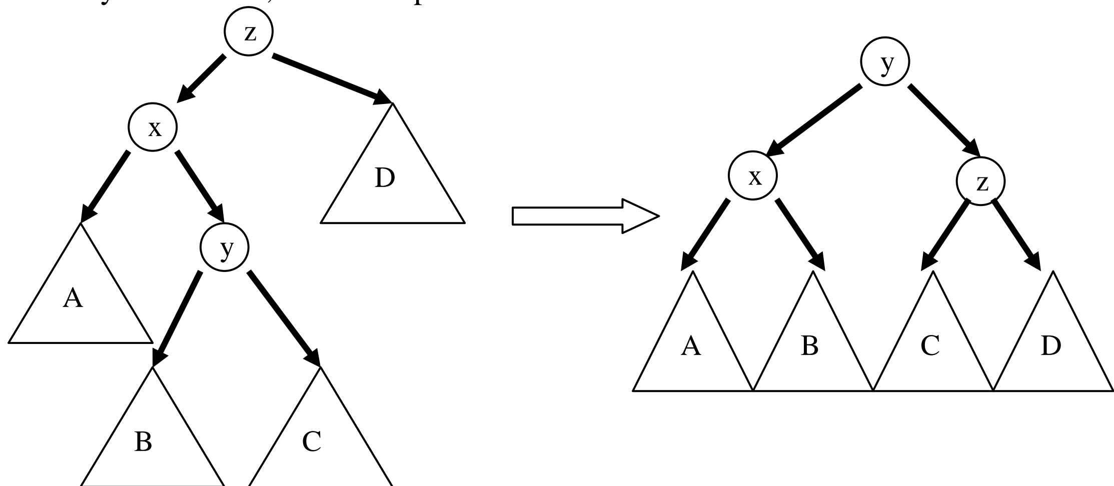
This led to the splay tree of Sleator and Tarjan, which takes a slightly more complex approach. On accessing a node, we again move it to the root by a sequence of local moves.

Splaying:

- If the element requested is the root, leave it as is.
- If the element requested is the child of the root, simply swap the two as indicated above (a single rotation, as per the AVL tree). The terminating single rotation is called a *zig* step.
- If the element requested is the left child of a left child (similarly right child of right child), the element is moved up in a manner that can be viewed as a single rotation between the parent and the grandparent of the accessed node, followed by a roation between the accessed node and its parent. This is called the *zig-zig* step, as shown below as z is accessed. Note that it does not correspond to either the single or double rotation of the AVL tree.



- Finally we have the case in which the requested element is an “inside” grandchild, either the left child of a right child or the right child of a left child. This *zig-zag* step is shown below when y is accessed, and corresponds to the double rotation of AVL trees.



On insertion or access to an element in the tree, we splay the node in question to the root. If the node requested is not present, we simply splay the lower of its neighbours to the root. (This is the node at which we discover our value is absent.) Deletions involve splitting he tree at the node in question and performing splays to put it back together. (We omit the details.)

Splay tree give a guarantee of $\Theta(\lg n)$ behaviour without and extra space constraints on the tree. We can use a pointer reversal trick to avoid the space of the stack as we go down the tree. Much more important, however, is how the structure does on a “distorted” distribution of inputs, as a function of the number of “other elements” requested since the last time the current one was sought, and ideally its competitive behaviour.

The proof are based on a potential function argument so we have $a = t + \Phi' - \Phi$, where a is the amortized time for an operation, t is the actual time, Φ' is the potential after the operation and

Φ is the potential before the operation. So the total cost of all *m* operations is given by
$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (a_j + \Phi_{j-1} - \Phi_j) = \sum_{j=1}^{m} a_j + \Phi_0 - \Phi_m$.
Each item *i* has a weight *w(i)*. The size of node *x*, s(x) is the sum of the weights of the nodes in the subtree weighted at x, and the rank *r(x)* is lg s(x). The potential of a tree is the sum of he ranks of all nodes. The following lemma gives a bound on the cost of splaying, charging 1 per rotation and 1 if there are no rotations.

Access Lemma: The amortized time to splay a tree with root *t* at a node *x* is at most $3(\mathrm{r(t)} - \mathrm{r(x)} + 1 = \mathrm{O(lg(s(t)/s(x)))}$.
Proof: omitted

This leads directly to several interesting theorems:
Static Optimality Theorem: If every item is accessed at least once, then the total access time is $O(m + \sum_{i=1}^{n} q_i \lg(m/q(i)))$ where $q(i)$ denotes the access frequency of element *i*.

Working Set Theorem: The total access time is $O(n \lg n + m + \sum_{j=1}^{m} \lg(t(j) + 1))$ where *t(j)* denotes the number of distinct items accessed since the last access of element i_j.

Scanning Theorem: Given an arbitrary *n*-node splay tree, the total time to splay once at each of the nodes, in increasing order, is *O(n)*.

There are many variants of these results. However, the main open question is whether or not the approach is competitive.