CS 840 **Fall 2020**

Self Organizing Linear Search and Binary Search Trees: Unit2

We will consider searching under the comparison model

Binary tree – lg n upper and lower bounds
This also holds in "average case"
Updates also in O(lg n)

Linear search – worst case n
If all elements have equal probability of search, expected time is (n+1)/2 in successful case (n for unsuccessful search)

Real Questions:

1. What if probabilities of access differ? How well can we do under stochastic model? (Easiest example: p_i is probability of requesting element i, probabilities are independent)

2. What if these (independent) probabilities are not given?

3. How do these methods compare with the best we could do?
   - given the frequencies (and later – given the sequence of requests)

   This leads to issues of

   - expected behaviour (given independent probabilities)

   - what if probabilities change

   - amortized behaviour (running versus an adversary so we are considering worst-case but for a sequence of operations)

   - amortized or expected cost could be compared with but possible for the given probabilities/sequence … perhaps compare with optimal static structure

   - consider our structures that will adapt, (self-adjust, though perhaps on-line and compare with the optimal adjusting structure that knows the sequence in advance (off-line)

     Model is a key issue

4. How can we develop self-adjusting data structures? – Adapt to changing probabilities.

Self Organizing Linear Search

Start with linear search. As noted

Worst case n (succ); also amortized
"Average" (all equal probabilities) (n+1)/2

– it doesn't matter how you change the structure for worst case or for all probabilities same and independent.

But

1) Start with $\{P_i\}$ $i = 1,n$ $P_i > P_{i+1}$ independent (Stochastic process)

2,3) Clearly the best order is $a_1 \ldots a_n$. Expected cost would be $Sopt = \sum_{i=1}^{n} i\, p_i$

Clearly, we could count accesses and converge
– But count and "chase" probability changes.

4) How can we adapt to changes in probability or "self adjust" to do better? -- other than count

Move elements forward as accessed

- Move to Front MTF
- Transpose TR
- Move halfway ---

Theorem: Under the model in which the probabilities of access are independent, and fixed, the transpose rule performs better than move to front unless
– $n \leq 2$ or
– all p_i's are the same, in which case the rules have the same expected performance (Rivest, *CACM '76*)

Theorem: Under the model in which probabilities of access are independent (and fixed) the move to front gives an expected behaviour asymptotically of $< 2 \Sigma\, i\, p_i$, i.e. less than a factor of 2 of the optimal (Rivest, *CACM '76*).

Can this be improved?

If all probabilities equal MTF = Sopt, the actual result is harder (much).

With $P_i = i^{-2} /(\pi^2/6)$ as $n \to \infty$ ($\pi^2/6$ – fudge factor)

MTF/Sopt = $\pi/2$ (Gonnet, Munro, Suwanda *SICOMP '81*)

and that is the worst case (Chung, Hajela, Seymour *JCSS '88*) (uses Hilbert's inequalities)

But how about – an amortized result, comparing these with Sopt in <u>worst case</u>. Note Sopt takes into account the frequencies

Transpose is a problem: alternate request for last 2 values - n is cost; MTF uses only 2 .

Move to Front – although "disaster" may be better

But how do we handle the "startup"
– note Rivest result asymptotic.

Bentley-McGeough (*CACM* '85).

The model:
Start with empty list
Scan for element requested, if not present,
Insert (and charge) as if found at end of list (then apply heuristic)

Theorem: Under the "insert on first request" startup model, the cost of MTF is $\leq$ 2 Sopt for <u>any sequence</u> of requests.

Proof. Consider an arbitrary list, but focus only on searches for b and for c, and "unsuccessful" comparisons where we compare query value "b or c" versus the other "c or b".

Assume sequence has k b's
and m c's $k \le m$

Sopt order is cb
and there are k "unsuccessful comparisons"

What order of requests maximizes this number under MTF? Clearly

$$c^{m-k}(bc)^k$$

So there are 2k "unssuccessful compares" (one for each b & one for each of the last k c's)

Now observe that this holds in any list for any pair of elements.

Sum over all

$$\text{Cost} \le 2\ \text{Sopt}$$

Note: This bound is tight.
Given $a_1, a_2, \ldots, a_n$, repeatedly ask for last in list, so all requested equally often.

Cost is n per search
Whereas "do nothing" = Sopt ≈ (n+1)/2

Note again Transpose is a disaster if we always ask for the last in its list.

Observe, for some sequences we do a lot better than static optimal $a^n_1\, a^n_2\, a^n_3 \ldots a^n_n$

Sopt - $(n+1)/2$

MTF - $$\left(\frac{n+1}{2} + n - 1\right)/\ n = \frac{n+1+2n-2}{2n} = \frac{3}{2} - O(1/n)$$

Next – Suppose you are given the entire sequence in advance. Sleator & Tarjan (1985).

The model we will discuss may or may not be realistic, but it is a benchmark.
We consider the idea of a "dynamic offline" method. The issue is:

on line: Must respond as requests come
vs
offline: Get to see entire schedule an determine how to move values

$$\text{Competitive Ratio of Alg} \equiv \text{Worst Case of } \frac{\text{Online time of Alg}}{\text{Optimal Offline time}}$$

A method is "competitive" if this ratio is a constant.

But the model becomes <u>extremely</u> important

Basics – Search for or change element in position i: scan to location i at cost i.

Unpaid exchange – can move element i closer to front of list free (keep others in same order)

Paid – can swap adjacent pairs at cost 1
Borodin and El-Yaniv prohibit going past location i
Sleator-Tarjan proof seems ok though.

Issues – Long/Short scan (ie passing location i)
Exchanging only adjacent values

Further

Access costs i if element in position i

Delete costs i if element in position i

Insert costs n+1 if n elements already there.

After any we can apply update.

Theorem: Under the model described, MTF is within a factor of 2 of offline optimal i.e. is 2-competitive.

Let A denote any algorithm, and MF denote the move to front heuristic.

We will consider the situation in which we deal with a sequence, S, having a total of M queries and a maximum of n data values. By convention we start with the empty list. The cost model for a search that ends by finding the element in position i is

i + # paid exchanges

Recall the element sought may be moved forward in the list at no charge (free exchange) while any other moves must be made by exchanging adjacent values.

Notation

$C_A(S)$ = total cost of all operations in S with algorithm A

$X_A(S)$ = # paid exchanges

$F_A(S)$ = # free exchanges

Note: $X_{MF}(S) = X_T(S) = X_{FC}(S) = 0$

(T denotes the transpose heuristic, FC denotes frequency count)

$F_A(S) \le C_A(S) - M$

(Since after accessing the i^{th} element there are at most i-1 free exchanges)

Theorem: $C_{MF}(S) \le 2C_A(S) + X_A(S) - F_A(S) - M$, for any algorithm A starting with the empty set.

Proof: The key idea is the use of a potential function Φ. We run algorithm A and MF, in parallel, on the same sequence, S.

Φ maps the configuration of the current status of the two methods onto the reals.

Running an operation (or sequence) maps Φ to Φ' and the amortized time of the operation is

$$T + \Phi' - \Phi$$

(i.e. amortized time = real time + $\Delta\Phi$)
(so we will aim at amortized time as an overestimate)
The j^{th} operation takes actual time (cost) t_j and amortized cost a_j

$$\Sigma_j t_j = \Phi - \Phi' + \Sigma_j a_j$$

where Φ is the initial potential and Φ', the final.

Φ is defined as the number of inversions between the status of A and MF, at the given time.

So $\quad \Phi \le n(n-1)/2$

We want to prove that the amortized time to access element i in A's list is at most 2i - 1 in MF's.

Similarly inserting in position i+1 has amortized cost 2(i+1) – 1. (Deletions are similar).

Furthermore: we can make the amortized time charged to MF when A does an exchange

-1 for free exchanges
at most +1 for paid exchanges

Initial configuration – empty; so $\Phi = 0$

Final value of Φ is nonnegative

So actual MF cost $\le$ Σ amortized time $\le$ our bound

{ access or insertion amortized time $\le 2C_A - 1$;
amortized delete time $\le 2C_A - 1$.
The -1's, one per operation, sum to -M }

Now we must bound the amortized times of operations.

Consider access by A to position <u>i</u> and assume we go to position <u>k</u> in MF.

x_i = # items preceding <u>it</u> in MF, but not in A

so # items preceding <u>it</u> in both is $(k - 1 - x_i)$

Moving <u>it</u> to front in MF creates

$k - 1 - x_i$ inversions

and destroys x_i others

so amortized time is

$$k + (k - 1 - x_i) - x_i = 2(k - x_i) - 1$$

But $(k - x_i) < i$ as $k-1$ items precede <u>it</u> in MF and only $i-1$ in A.

So amortized time $\leq$ <u>$2i - 1$.</u>

The same argument goes through for insert and delete.

An exchange by A has zero cost to MF,
so amortized time of an exchange is just increase in # inversions caused by exchange,
i.e. 1 for paid, -1 for free.

Extension

Let MF(d) $(d \geq 1)$ be a rule by which the element inserted or accessed in position k is moved at least $k/d - 1$ units closer to the front. Then

$$C_{MF(d)}(S) \leq d\,(2C_A(S) + X_A(S) - F_A(S) - M)$$

{ eg. MF $\equiv$ MF(1) }

<u>Also</u>

Theorem: Given any algorithm A running on a sequence S, there exists another algorithm for S that is no more expensive and does no paid exchanges.

Proof Sketch: Move elements only after an access, to corresponding position. There are details to work through.

Further applications can be made to paging.

However – there is a question about the model.

Given the sequence

$$1 \rightarrow 2 \rightarrow \cdots \rightarrow n/2 \rightarrow n/2+1 \rightarrow \cdots \rightarrow n$$

suppose we want to convert it to

$$n/2 \rightarrow \cdots \rightarrow n \rightarrow 1 \rightarrow \cdots \; n/2$$

what “should” I pay?
Observe that all only 3 pointers are changed, though presumably I have to scan the list ($\Theta(n)$).

Sleator and Tarjan model says $\Theta(n^2)$, perhaps $\Theta(n)$ is a fairer charge.

So for an offline algorithm: To search for element in position i, we probe up to position k ($k \geq i$) and can reorder elements in positions 1 through k. Cost is k.

J.I. Munro: On the Competitiveness of Linear Search. ESA ‘00 (LNCS 1879 pp 338-345)

Consider the following rule, Order by Next Request (ONR):
To search for element in position i, continue scan to position $2^{\lceil \lg i \rceil}$.
Then reorder these $2^{\lceil \lg i \rceil}$ elements according to the time until their next requests.
(The next of these to be accessed goes in position 1)

$$\text{Cost } 2^{\lceil \lg i \rceil} = \Theta(i)$$

How does this do? First try a permutation, say 1, …, n (let n = 16) and assume 1 is initially in the last half).
(To simplify diagram we move requested value to front)

Request	Cost	New Ordering
1	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2	2	2 1 • •
3	4	3 4 1 2 • •
4	2	4 3 • •
5	8	5 6 7 8 1 2 3 4 • •
6	2	6 5 • •
7	4	7 8 5 6 • •
8	2	8 7 • •
9	16	9 10 11 12 13 13 14 15 16 1 2 3 4 5 6 7 8

Clearly the same cost applies to any permutation

Under our model this cost is ~ n lg n.

Optimal and Self Organizing Binary Search Trees

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 $q_{i-1}, \ldots, q_j$ or $p_i, \ldots, p_j$.

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 almost 50 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.

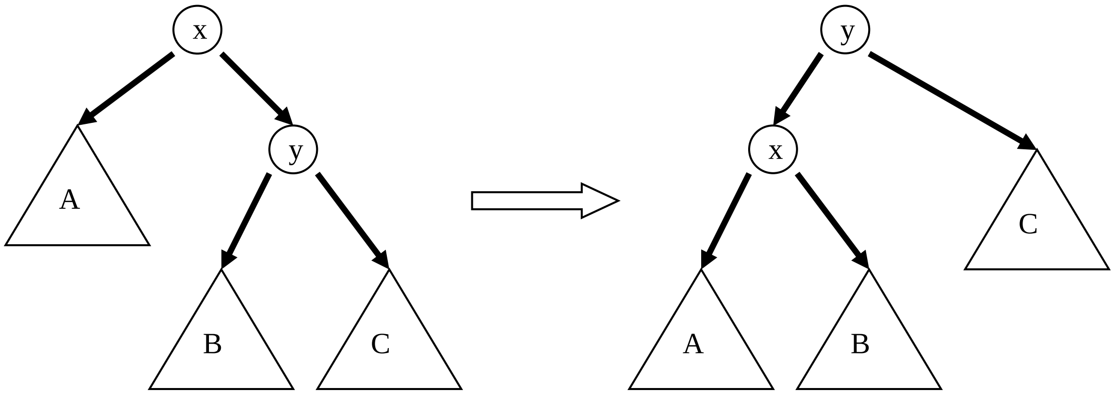
We also note that the optimal tree is expensive to compute and ask whether it can be approximated more cheaply. A natural greedy algorithm would be to put the key with the largest probability at the root. This can be a serious problem, even if all q_i values are 0. Suppose, for example, that all p_i 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.

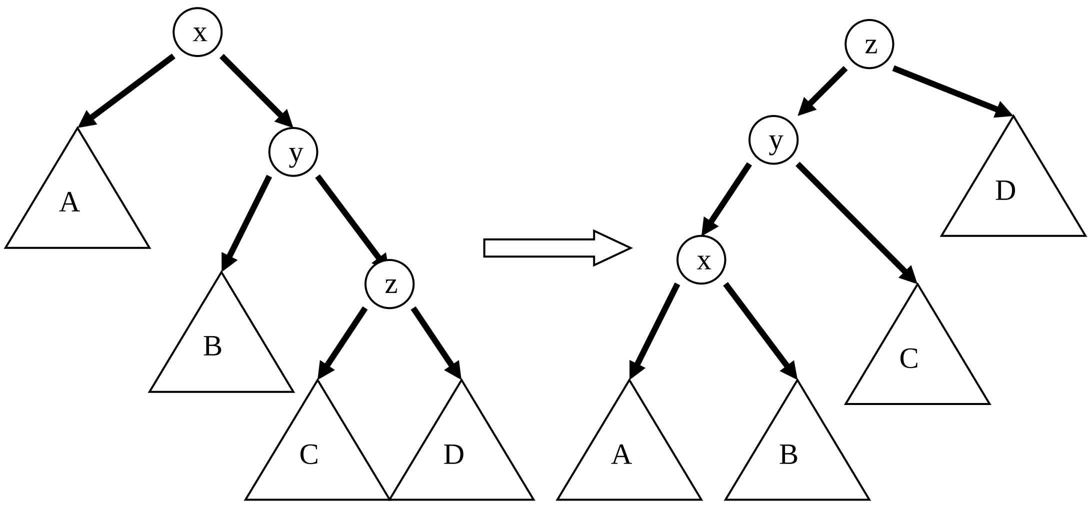
Unfortunately, this appealing scheme can be quite a disaster. If all key values have the same



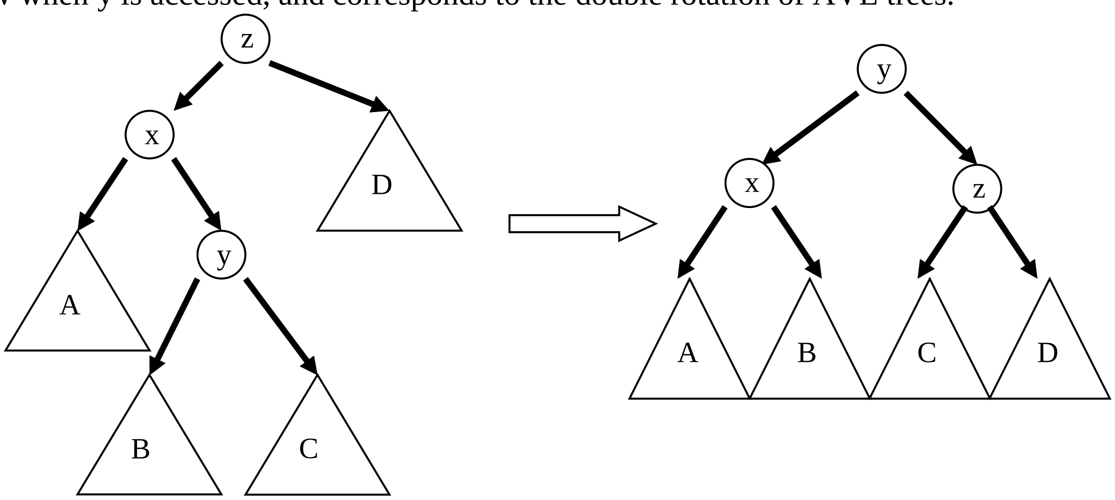
probability of request, then all binary trees are generated with the same probability. Unfortunately, most binary trees are very bad and the average search cost becomes $\Theta(n^{1/2})$.
[Note that this distribution is very different from what we would get by inserting elements into a binary tree in 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 $O(\lg n)$ 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.