

1 ONR Continued..

Recall that, the amortized cost of ONR to access an element v is $\leq 1 + 4\lceil \log r \rceil$, where r is the number of distinct elements accessed since v was last accessed, including v .

To get this amortized cost, we charge something to elements during access.

1. We want to charge a constant amount to each element charged.
→ This means that we must charge at least a constant fraction of the elements seen.
2. We want to bound the number of times an element is charged.
→ To do this, we will charge elements only when their state changes.

What should the charging scheme be? We look at a few ideas:

Idea 1: Charge the elements in the last block.

Analysis: Note that the elements in the last block may never be accessed. So, there may be an unbounded number of charges made to those elements between accesses of those elements. Thus, this scheme is not feasible.

Idea 2: Charge the elements in the front.

Analysis: There are not enough of these elements, and so they cannot absorb all the cost. Hence, this scheme is also infeasible.

We now study the actual charging scheme and also give a sketch of the proof for the amortized cost. The taxation policy is called "Soak the middle-class", and the charges in this scheme are as follows:

1. If the request is for the first element, we charge it 1.
2. Otherwise, the requested element is in position $q + 1$, where the last block inspected ended at position $q = 2^i$ (i.e the requested element must be the first in its block). The block that ends at position $q = 2^i$ is of length $q/2 = 2^{i-1}$. The cost of the search is $2q = 2^{i+1}$. We charge this cost to the $q/2 = 2^{i-1}$ elements in the penultimate block, b.

Claim 1 *An element v is charged at most once in any block, between 2 requests for v .*

Proof: Suppose v is in block b during a request in block $b + 1$. Then v is charged during the current request. We prove that v will not be charged again in block b before the next request for v .

There are 2 cases to consider. Either v remains in block b or moves to an earlier block $b' < b$ as a consequence of the re-ordering done by the current request, or v moves to block $b + 1$.

Case 1: v remains in block b or moves to $b' < b$ after current request.

Note that all the elements from v 's new position to end of block $b + 1$ (i.e. position $2q$) are ordered by next request. Therefore, their order will not change, though other elements ahead of v may be moved to positions among them, after being accessed. No elements after v up to the end of block $b + 1$ will be requested before v is requested again.

So, while v remains in any block $b' \leq b$, there is a buffer of elements between v and the end of block $b + 1$ that will not be requested. So, v will not be in the penultimate block, and will not get charged again.

If a request is made to an element in a block $b' > b + 1$, v will not be charged but it may move to block $b + 1$, or greater. In that case, again, there is a large buffer of elements after v that will not be accessed. So, if v moves back to block b , there will still be a buffer at the end of block $b + 1$, and no request for an element in block $b + 1$ will be made, and v will not be charged in block b .

Case 2: v moves to block $b + 1$.

v may now get charged if an element in block $b + 2$ is requested. We argue that if v moves back to block $b' \leq b$, it will not be charged again. Since elements are ordered by next request, all elements after v in block $b + 1$ will be requested after v 's next request. If v moves back to block $b' \leq b$, it is because more elements have been placed behind it, and again all elements after v to end of block $b + 1$ will be requested after v .

So, again there is a buffer to the end of block $b + 1$. Hence, v will not be in the penultimate block and therefore, not get charged. ■

How many blocks can v be charged in? Recall that r is the number of distinct elements accessed between 2 accesses of v . So, the farthest right position v could be in is position r . Thus,

- v can be in at most $\lceil \log r \rceil$ blocks.
- at most $\lceil \log r \rceil$ charges will be made to v .
- the charge at block $[2^{i-1} + 1, 2^i]$ is 2^{i+1} , giving a per element charge of $2^{i+1}/2^{i-1} = 4$.

Thus, the amortized cost per element is $4\lceil \log r \rceil + 1$, where the additional 1 is the cost of accessing the front of the list.

Corollary 1 *The total cost of ONR is $\leq m + 4 \sum_{i=1}^n \lceil \log \frac{m}{f_i} \rceil$, where f_i is the number of occurrences of i .*

Let $p_i = \frac{f_i}{m}$ be the probability of accessing element i . Then, the following are true:

1. Amortized cost for element i is $O(\log \frac{1}{p_i})$.
2. “Expected” cost for a random element is $O(\sum_{i=1}^n p_i \log \frac{1}{p_i})$. This cost approaches the entropy¹ bound, i.e. it approaches the information theoretic lower bound.
3. The total cost is $O(m \sum_{i=1}^n p_i \log \frac{1}{p_i})$.

Note: “Expected” is in quotes since the expected value is over the input distribution, not over random choices.

2 Binary Search Trees

We study the same questions here as we did for linear search linked lists.

Worst case search = $\Theta(\log n)$

Cost model = number of nodes inspected starting at root

2.1 Stochastic Model for Optimal BSTs

Refer to CLRS chapter 15.5 for details about optimal BSTs.

$x_1 < x_2 < \dots < x_n$,

p_i = probability of searching for element x_i , $1 \leq i \leq n$,

q_i = probability of searching for element y , such that $x_i < y < x_{i+1}$, $1 \leq i \leq n-1$,

q_0 = probability of searching for element $y < x_1$,

q_n = probability of searching for element $y > x_n$.

See figure 1. The d_i s are dummy keys.

Define $root[i, j]$ = root of optimal tree on $d_{i-1}, x_i, d_i, \dots, x_j, d_j$,

and, $cost[i, j]$ = cost of tree rooted at $root[i, j]$ \times probability of entering the tree.

Use dynamic programming (DP) to compute $cost[i, j], \forall i, j$.

Base: $root[i, i] = x_i$,

and, $cost[i, i] = q_{i-1} + p_i + q_i$.

Recursive step:

$$cost[i, j] = \min_{i \leq r \leq j} [(q_{i-1} + p_i + q_i + \dots + p_j + q_j) + cost[i, r-1] + cost[r+1, j]]$$

¹The entropy of a random variable X is:

$$H(X) = E \left[\log \frac{1}{Pr(X)} \right] = \sum_{\forall X} Pr(X) \log \frac{1}{Pr(x)},$$

where $Pr(X)$ is the probability of X .

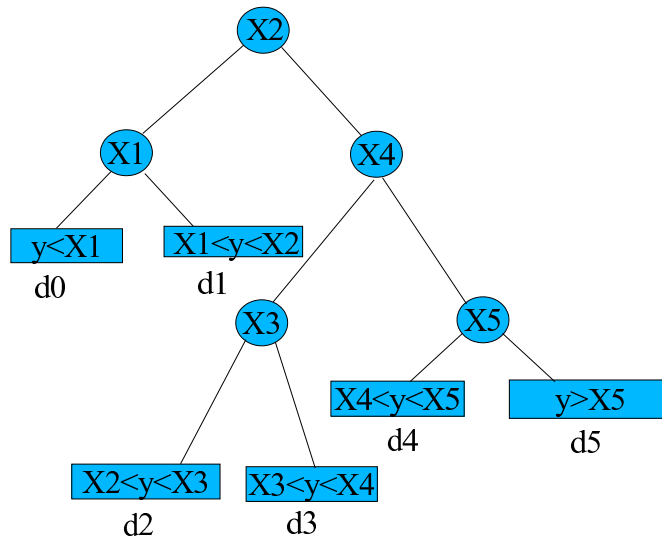


Figure 1: Optimal BST

Note: The first term, $(q_{i-1} + p_i + q_i + \dots + p_j + q_j)$, is the cost to access root.

And, $root[i, j] = \min r$ in $cost[i, j]$.

2.2 Self-organizing Trees

We assume an unknown distribution. Some possible approaches for self-organizing trees are:

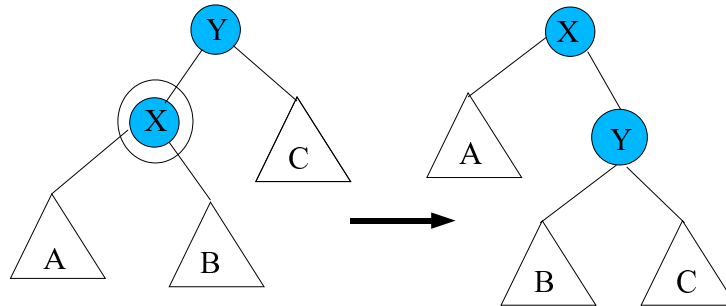


Figure 2: Transpose Analog

1. Transpose Analog: (Allen and Munro JACM'78 [2])

- Searches and moves the node up by 1 level (by performing rotations²) See figure 2.

The problems with this scheme are:

- It is bad for uniform searches.

²*Rotate(X)*: moves node X up by 1 level. For an example see figure 2

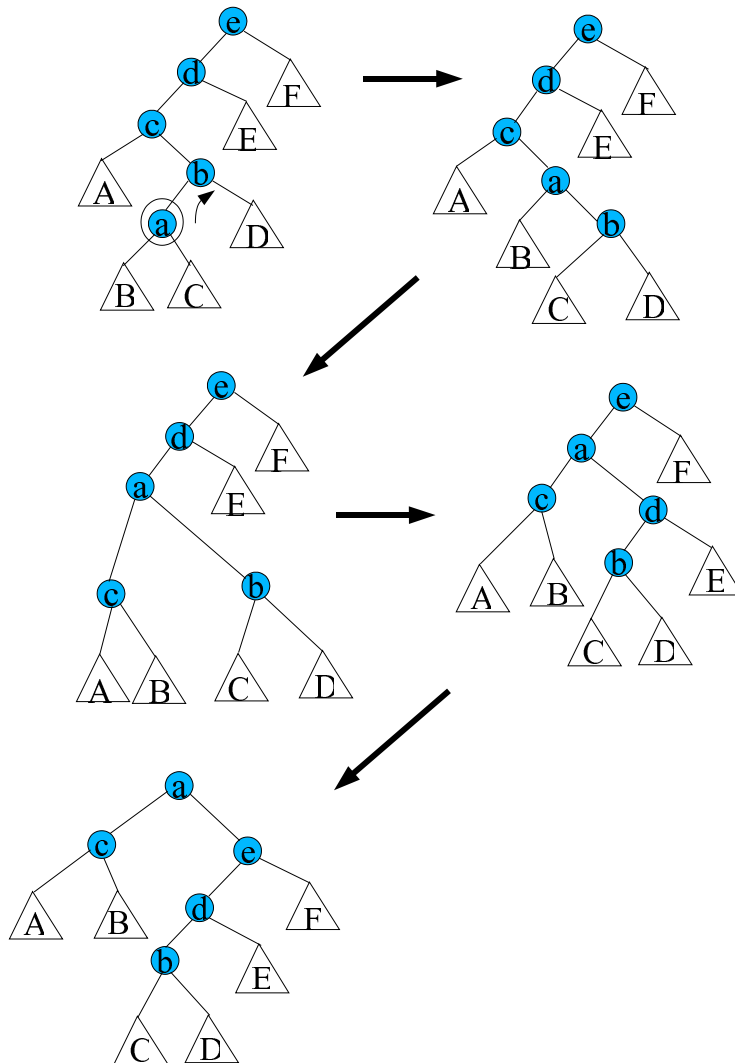


Figure 3: Move-to-root

- All binary search trees are generated with equal probability,
 \Rightarrow The average search cost is $\Theta(\sqrt{n})$.

Note that these trees are not necessarily balanced.

2. Move-to-root:

- Repeatedly rotate the searched element all the way to the root. See figure 3
- The search cost is $2 \ln 2 \approx 1.38$ of statistically optimal tree in a stochastic model.

A third type of self-organizing tree is the splay tree. We study splay trees in detail.

2.3 Splay Trees

Splay trees are a version of move-to-root. The main idea is to perform rotation in pairs. We perform the splaying step until the element reaches the root. Note that here $p(x)$ refers to the parent of x .

Splaying step:

Case 1: (*Zig*): if $p(x)$ is root, perform $Rotate(x)$. See figure 4.

Case 2: (*Zig-Zig*): If $p(x)$ is not root and x and $p(x)$ are both left children or both right children, then, perform $Rotate(p(x))$, and $Rotate(x)$. See figure 5.

Case 3: (*Zig-Zag*): If $p(x)$ is not the root and x is a left (right) child and $p(x)$ is the right (left) child, then, perform $Rotate(x)$, and $Rotate(p(x))$. See figure 6.

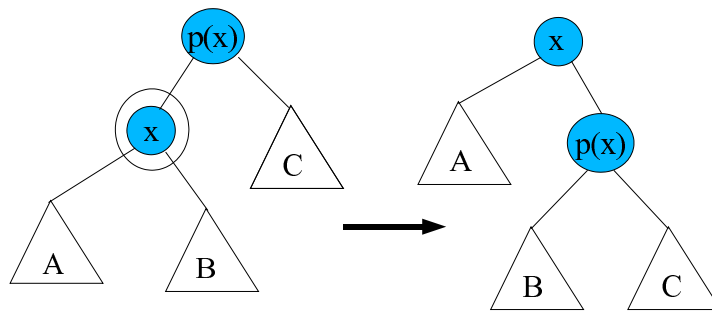


Figure 4: Zig

Note: Splaying roughly halves the depth of every node along the access path. See figure 7 for an example. Also, observe the difference between splay trees and the move-to-root scheme. It might at first appear that the move-to-root scheme which moves the node to root by repeated rotation of the searched element does the same thing as repeated splaying. However, in splaying, we do not always rotate just the searched element. The Zig-Zig step rotates $p(x)$ and then x . This is the crucial step that ensures that the height of every node in the splayed tree is roughly halved. This is not true for the move-to-root scheme.

The amortized cost per operation (for search, insert, delete) is $O(\log n)$. This approaches the entropy bound and thus splay trees are almost statistically optimal. Next, we analyze this amortized cost for the search operation.

Analysis: Define a potential function Φ on each configuration of data structure.

Each item i has weight w_i . The size of a node x ,

$$S(x) = \sum_{i \in \text{subtree rooted at } x} w_i ,$$

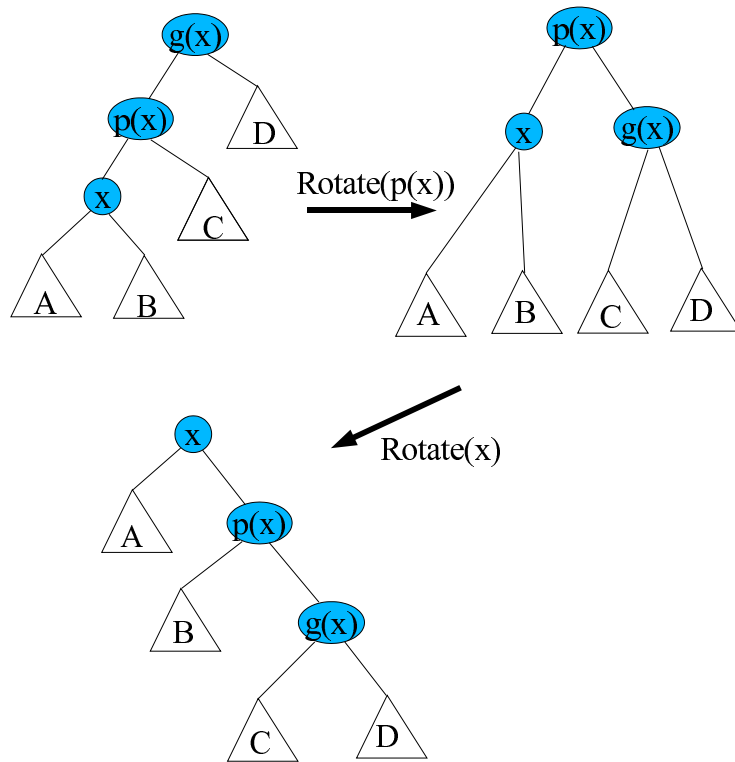


Figure 5: Zig-Zig

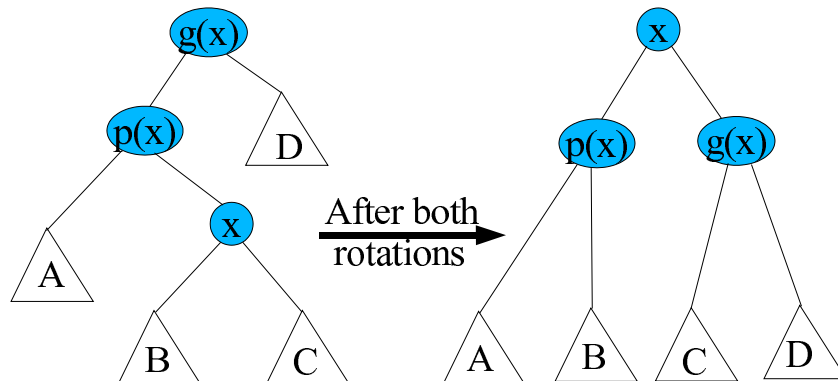


Figure 6: Zig-Zag

rank of a node x , $r(x) = \log S(x)$.

$$\Phi(T) = \sum_{i=1}^n r(x_i)$$

The running time of a splay operation is $\max\{i, \text{number of rotations}\}$.

Lemma 2 Access Lemma: *The amortized time to splay a node x in tree rooted at t is at most $3[r(t) - r(x)] + 1$. Note that this is equal to $O(\log \frac{S(t)}{S(x)})$.*

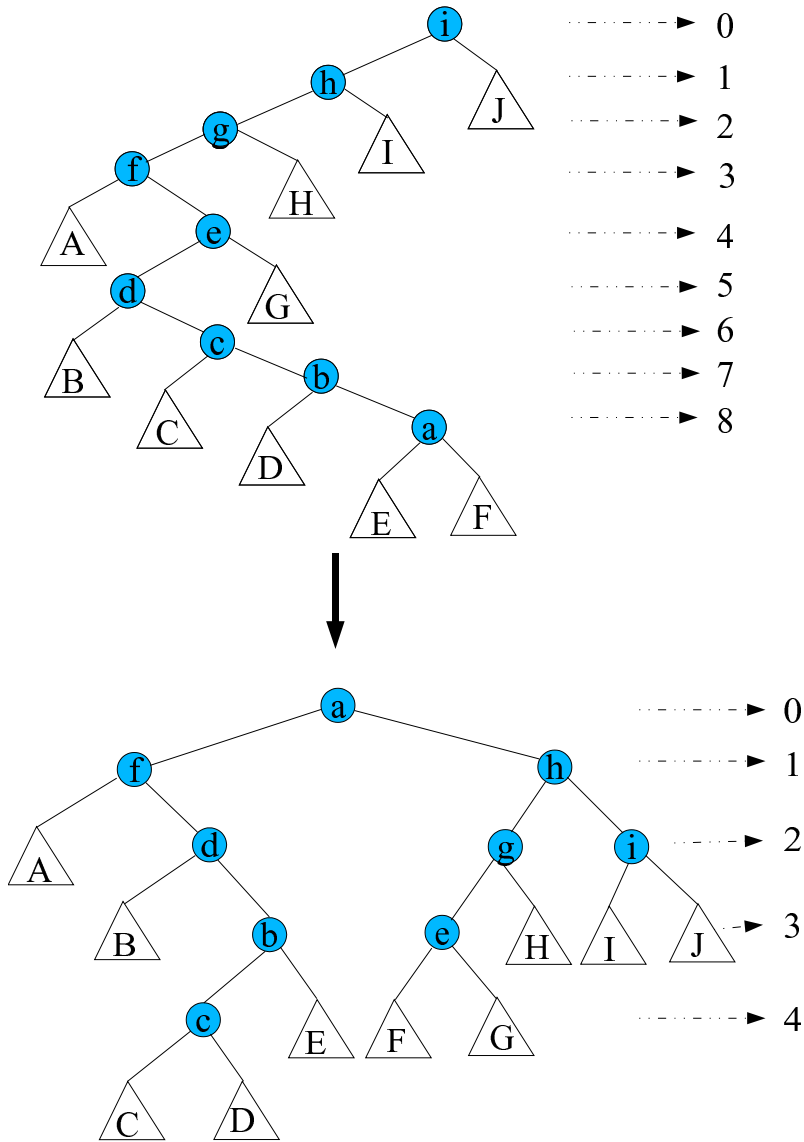


Figure 7: Splaying roughly halves the depth of each node along the access path

Proof: Suppose there is at least 1 rotation. Consider a splay step. Let S and S' , and, r and r' be the size and rank functions right before and after this step respectively. Let y be the parent of x , and z be the grandparent of x . We look at the 3 cases of the splaying step:

$$\begin{aligned}
 \text{Case 1: (Zip): Amortized cost} &= 1 + r'(x) + r'(y) - r(x) - r(y) \\
 &\leq 1 + r'(x) - r(x) \quad (\text{since, } r' \leq r(y)) \\
 &\leq 1 + 3(r'(x) - r(x)) \quad (\text{since, } r'(x) \geq r(x))
 \end{aligned}$$

Thus, the lemma holds for case 1.

Case 2: (Zig-Zig): Here, there are two rotations. Therefore,
 Amortized Cost = $2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$

$$\begin{aligned}
&= 2 + r'(y) + r'(z) - r(x) - r(y) && \text{(Since, } r'(x) = r(z)\text{)} \\
&\leq 2 + r'(x) + r'(z) - r(x) - r(x) && \text{(Since, } r'(y) \leq r'(x)\text{, and } r(y) \geq r(x)\text{)}
\end{aligned}$$

We need to prove that,

$$2 + r'(x) + r'(z) - 2r(x) \leq 3(r'(x) - r(x)).$$

To do this, we show that,

$$2 \leq 2r'(x) - r(x) - r'(z).$$

Note that for $x, y > 0$ and $x + y = 1$, $\log x + \log y$ is maximum at value -2 . This happens when $x = y = 1/2$.

$$\therefore \log\left(\frac{S(x)}{S'(x)}\right) + \log\left(\frac{S'(z)}{S'(x)}\right) \leq -2 \quad (\because S(x) + S'(z) \leq S'(x))$$

Now, we know that,

$$r(x) - r'(x) = \log\left(\frac{S(x)}{S'(x)}\right),$$

$$\text{and, } r'(z) - r'(x) = \log\left(\frac{S'(z)}{S'(x)}\right)$$

By adding these two equations we get,

$$-2r'(x) + r(x) + r'(z) \leq -2$$

$\therefore 2 \leq 2r'(x) - r(x) - r'(z)$, and we are done.

Case 3: (Zig-Zag):

$$\begin{aligned}
\text{Amortized Cost} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&\leq 2 + r'(y) + r'(z) - 2r(x) && \text{(Since, } r'(x) = r(z)\text{, and, } r(x) \leq r(y)\text{)}
\end{aligned}$$

Now, we have,

$$\log\left(\frac{S'(y)}{S'(x)}\right) + \log\left(\frac{S'(z)}{S'(x)}\right) \leq -2 \quad (\because S'(y) + S'(z) \leq S'(x))$$

$$\Rightarrow r'(y) - r'(x) + r'(z) - r'(x) \leq -2$$

$$\Rightarrow r'(y) + r'(z) \leq -2 + 2r'(x).$$

Using this in the amortized cost above, we get,

$$\text{Amortized cost} \leq 2[r'(x) - r(x)]$$

Thus, the amortized cost of one splay step is $\leq 3r'(x) - 3r(x)$. But we do it until the element becomes the root. Therefore, we have,

$$\begin{aligned}
\text{Total amortized cost} &\leq 3r(t) - 3r(x) + 1 && (\because \text{it is a telescoping sum}) \\
&= O\left(\log\left(\frac{S(t)}{S(x)}\right)\right)
\end{aligned}$$

For more information about splay trees see [3] (Sleator and Tarjan JACM'85).

Consequences of the splay tree analysis: Consider a sequence of m accesses on an n -node tree.

$$\Delta\Phi \leq \sum_{i=1}^n \log\left(\frac{W}{w(i)}\right) \quad (\because w(i) \leq s(i) \leq W)$$

where, $W = \sum_{i=1}^n w(i)$.

We prove the $\log n$ bound:

Assign $w(i) = 1$ for all i , then, $S(t) = n$.

From the access lemma we have,

Amortized cost for node x is $O(\log \frac{n}{k})$ and $k = S(x)$, which gives $O(\log n)$ amortized complexity.

2.3.1 Static Optimality Theorem

$$w(i) = P_i = \frac{f_i}{m}$$

$s(t) = 1$. Now, since $s(x) \geq w(x)$, the access to node x will be $O(\log \frac{1}{k})$

$$= O(mH)$$

\Rightarrow Static optimality.

2.3.2 Static Finger Theorem

Theorem 3 *Let f be a specific node in a tree, then the amortized cost of accessing node i is, $O(\log(1 + |i - f|))$ where $|i - f|$ is the difference between i and f in the total order of the keys.*

Proof: Let $w(i) = \frac{1}{(1+|i-f|)^2}$.

$$\begin{aligned} s(t) &\leq \sum_{k=-\infty}^{\infty} \frac{1}{k^2} \\ &= 2 \sum_{k=1}^{\infty} \frac{1}{k^2} = O(1) \end{aligned}$$

From access lemma, the cost of splaying node i is,

$$O\left(\log \frac{O(1)}{w(i)}\right)$$

$$= O(\log(1 + |i - f|)^2)$$

$= O(\log(1 + |i - f|))$. This proves the theorem. ■

References

- [1] J. Ian. Munro, *On the competitiveness of linear search*, ESA, 2000: 338–345.
- [2] Brian Allen, Ian Munro, *Self-Organizing Binary Search Trees*, JACM, 25: 526–535, 1978.
- [3] D. D. Sleator, R. E. Tarjan, *Self-adjusting binary trees*, JACM, 32: 652–686, 1985.