MIT OpenCourseWare
http://ocw.mit.edu

6.854J / 18.415J Advanced Algorithms
Fall 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

18.415/6.854 Advanced Algorithms September 24, 2008

Lecture 6 - Splay Trees

Lecturer: Michel X. Goemans

1 Introduction

In this lecture, we investigate **splay trees**, a type of binary search tree (BST) first formulated by Sleator and Tarjan in 1985. Splay trees are self-adjusting BSTs that have the additional helpful property that more commonly accessed nodes are more quickly retrieved. They have good behavior when compared to many other types of self-balancing BSTs, even when the operations are unknown and non-uniform. While in the worst case, operations can take $O(n)$ time, splay trees maintain $O(\log n)$ amortized cost for basic BST operations, and are within a constant factor to the cost of any static BST.

We first give an overview of the operations used in splay trees, then give an amortized analysis of its behavior. We conclude by noting its behavior relative to other Binary Search Trees.

2 Splay Tree Structure

A splay tree is a dynamic binary search tree, meaning that it performs additional operations to optimize behavior. Because they are BSTs, given a node x in a splay tree and a node y in the left subtree of x, we have $\text{key}(y) < \text{key}(x)$. Similarly, for a node z in the right subtree of x, we have $\text{key}(x) < \text{key}(z)$. This is the *binary search tree property.* A well-balanced splay tree will have height $\Theta(\log(n)$, where n is the number of nodes.

Splay trees achieve their efficiency through use of the following operations:

2.1 Rotation

The basic operation used in splay trees (or any other dynamic BST) is the **rotation**. A rotation involves rearranging the nodes of a subtree rooted at y so that one of the children x of y becomes the new root of the subtree, while maintaining the binary search tree property. This is illustrated in Figure 1.

When the left child becomes the new root, the rotation is a right rotation. When the right child becomes the new root, the rotation is a left rotation. We call a right rotation a **zig** and a left rotation a **zag**.

The key idea of the splay tree is to bring node x to the root of the tree when accessing x via rotations. This brings the most recently accessed nodes closer to the top of the tree.

However, there are many ways of bringing a node to the root via rotations, and we must therefore specify in which order we perform them. Consider a linear tree (effectively a linked list) of the values $1, \ldots, n$, rooted at n. Suppose we access the value 1. If we use the naive (and most natural) method of repeatedly performing a zig to bring 1 at the top, we proceed as illustrated in Figure 2. The resulting tree has the same height as the original tree, and is clearly not better balanced. We must try a more clever approach than successive, single rotations.

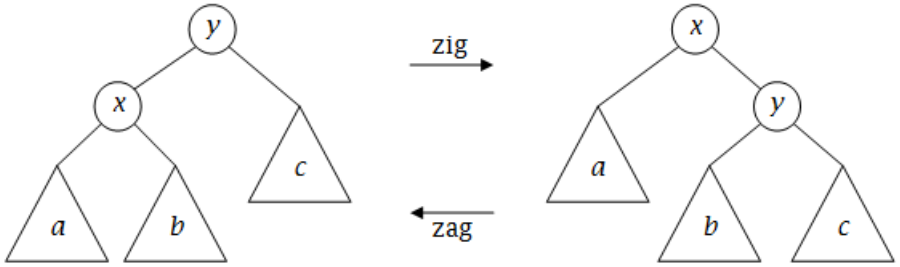

Figure 1: Rotation via zigs and zags.

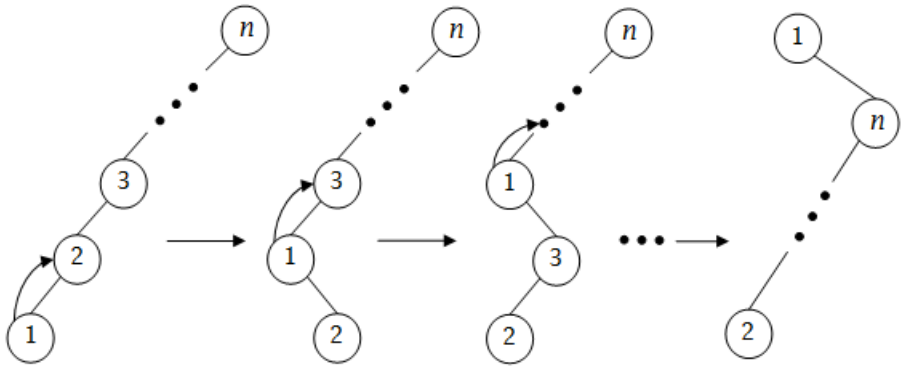

Figure 2: When we access node 1 and try to bring it up via pure rotations, the result is a tree that is just as unbalanced as before.

2.2 Splay-Step

We now define an operation called SPLAY-STEP. In one splay-step on a node x, x is brought up 2 levels with rotations (or just 1 level if x's parent is the root). When some node x is accessed in the splay tree, we bring x up with a series of splay-steps until it is the root.

We separate the actions performed for the splay-step into the following categories. Call the node that we are trying to access x, its parent y, and y's parent z.

- **Case 0**: x is the root. Do nothing in this case.
- **Case 1**: y is the root. If x is the left child of the root, perform a zig on x and y. If not, perform a zag.
- **Case 2**: x and y are both left children (or both right children). Let us look at the case when both x and y are left children. We first do a zig on the y-z connection. Then, we do a zig on the x-y connection. If x and y are right children, we do the same thing, but with zags instead. (See Figure 3.)
- **Case 3**: x is a left child and y is a right child, or vice versa. Consider the case where x is a right child, and y is a left child. We first do a zag on the x-y edge, and then a zig on the x-z edge. In the case where x is a left child and y a right child, we do the same thing, but with a zig on the first move, followed by a zag. (See Figure 4.)

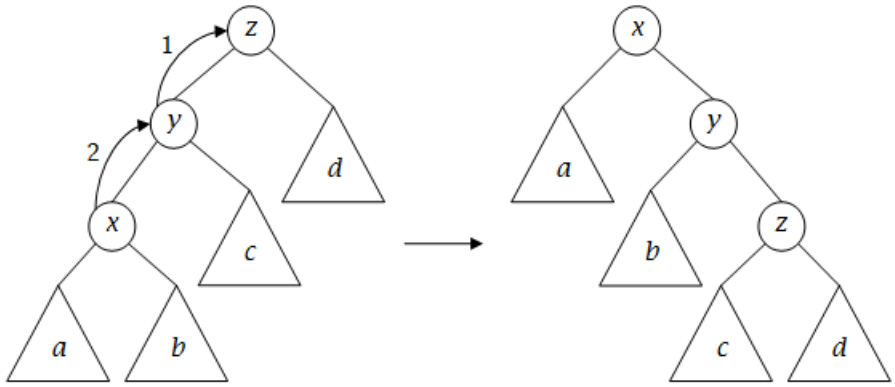

Figure 3: Case 2 of the splay-step is when x and y are the same type of children. In this figure, we first do a zig on $y - z$, and then a zig on $x - z$.

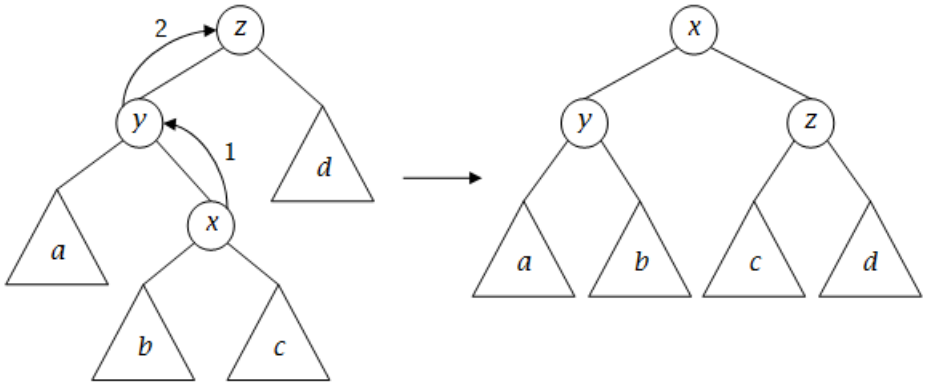

Figure 4: In Case 3, x and y are not the same type of children. In this case, we do a zag on the $x - y$ edge, and then a zig on the $x - z$ edge.

Note that in the case of the earlier example with the chain of nodes, using splay-step instead of direct rotations results in a much more balanced tree, see Figure 5.

2.3 Splay

With the splay-step operation, we can bring the node x to the root of the splay tree with the procedure:

```
splay(x):
   WHILE x≠root:
      DO splay-step(x)
```

The described procedure performs the splay operation in a bottom-up order. It is possible to perform the splay operation in a top down fashion, which would result in the same running time.

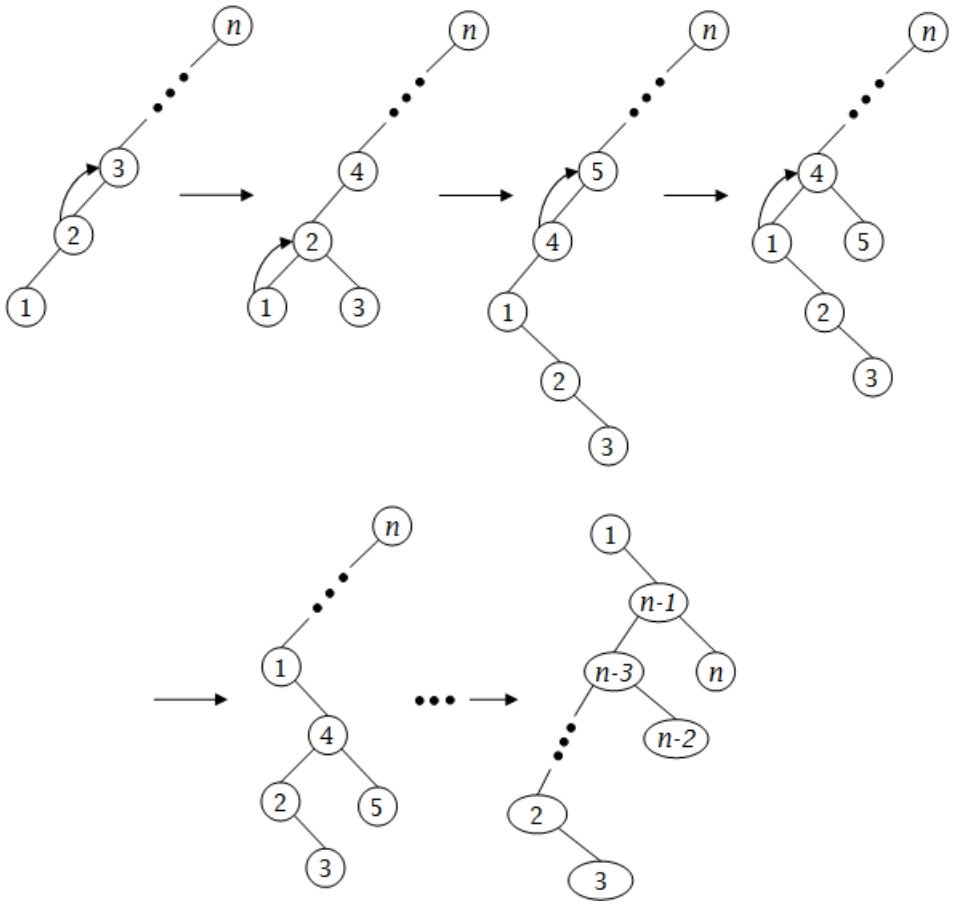

Figure 5: When splaying node 1, the resulting tree has half its original height.

3 Running-Time Analysis

3.1 Potential Function

We define a class of potential functions for the amortized analysis of operations on a splay tree. The potential function depends on weights that we can choose. For each node x in the tree, make the following definitions:

- $T(x)$ is the subtree rooted at x (and it includes teh node x itself),
- weight function: $w(x) > 0$ is the weight of node x (we can choose what this is; we'll often take $w(x) = 1$ for all nodes x)
- weight-sum function: $s(x) = \sum_{y \in T(x)} w(y)$,
- rank function: $r(x) = \log_2 s(x)$.

Then we define the potential function as:

$$\phi = \sum_{x \in T(root)} r(x).$$

3.2 Amortized Cost of Splay(x)

Using the potential function described above, we can show that the amortized cost of the splay operation is $O(\log n)$. For the purposes of cost analysis, we assume a rotation takes 1 unit of time.

Lemma 1 *For a splay-step operation on x that transforms the rank function r into r', the amortized cost is $a_i \leq 3(r'(x) - r(x)) + 1$ if the parent of x is the root, and $a_i \leq 3(r'(x) - r(x))$ otherwise.*

Proof of Lemma 1: Let the potential before the splay-step be ϕ and the potential after the splay-step be ϕ'. Let the worst case cost of the operation be c_i. The amortized cost a_i is $a_i = c_i + \phi' - \phi$. We consider the three cases of splay-step operations.

Case 1: In this case, the parent of x is the root of the tree. Call it y. After the splay-step, x becomes the root and y becomes a child of x. The operation involves exactly one rotation, so $c_i = 1$.

The splay step only affects the rank for x and y. Since y was the root of the tree and x is now the root of the tree, $r'(x) = r(y)$. Additionally, since y is now a child of x, (the new) $T(x)$ contains (the new) $T(y)$, so $r'(y) \leq r'(x)$. Thus the amortized cost is:

$$\begin{aligned} a_i &= c_i + \phi' - \phi \\ &= 1 + r'(x) + r'(y) - r(x) - r(y) \\ &= 1 + r'(y) - r(x) \\ &\leq 1 + r'(x) - r(x) \\ &\leq 1 + 3(r'(x) - r(x)), \end{aligned}$$

since $r'(x) \geq r(x)$.

Case 2: In this case, we perform two zigs or two zags, so $c_i = 2$. Let the parent of x be y and the parent of y be z. Node x takes the place of z after the splay-step, so $r'(x) = r(z)$. Also, we see in Figure 3 that $r(y) \geq r(x)$ (since y was the parent of x) and $r'(y) \leq r'(x)$ (since y is now a child of x). Then the amortized cost is:

$$\begin{aligned} a_i &= c_i + \phi' - \phi \\ &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= 2 + r'(y) + r'(z) - r(x) - r(y) \\ &\leq 2 + r'(x) + r'(z) - r(x) - r(x). \end{aligned}$$

Next, we use the fact that the log function is concave, or $\frac{\log a + \log b}{2} \leq \log\left(\frac{a+b}{2}\right)$. If the splay-step operation transforms the weight-sum function s into s', we have:

$$\frac{\log_2(s(x)) + \log_2(s'(z))}{2} \leq \log_2\left(\frac{s(x) + s'(z)}{2}\right).$$

The left side is equal to $\frac{r(x)+r'(z)}{2}$. On the right side, note that

$$s(x) + s'(z) \leq s'(x);$$

indeed the old subtree $T(x)$ and the new subtree $T'(z)$ cover all nodes of $T'(x)$, except y (thus $s(x) + s'(z) = s'(x) - w(y)$). Thus, we have:

$$\frac{r(x) + r'(z)}{2} \leq \frac{\log_2 (s'(x))}{2} = r'(x) - 1,$$

or

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

Therefore, the amortized cost is:

$$\begin{aligned} a_i &\leq 2 + r'(x) + 2r'(x) - r(x) - 2 - r(x) - r(x) \\ &= 3(r'(x) - r(x)). \end{aligned}$$

Case 3: In this case, we perform a zig followed by a zag, or vice versa, so $c_i = 2$. Let the parent of x be y and the parent of y be z. Again, $r'(x) = r(z)$ and $r(y) \geq r(x)$. Then the amortized cost is:

$$\begin{aligned} a_i &= c_i + \phi' - \phi \\ &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq 2 + r'(y) + r'(z) - r(x) - r(x). \end{aligned}$$

Note in Figure 4 that $s'(y) + s'(z) \leq s'(x)$. Using the fact that the log function is concave as before, we find that $r'(y) + r'(z) \leq 2r'(x) - 2$. Then we conclude

$$\begin{aligned} a_i &\leq 2 + 2r'(x) - 2 - r(x) - r(x) \\ &\leq 2(r'(x) - r(x)) \\ &\leq 3(r'(x) - r(x)). \end{aligned}$$

□

Lemma 2 *The amortized cost of the splay operation on a node x in a splay tree is $O(1+\log \frac{s(root)}{s(x)})$.*

Proof of Lemma 2: The amortized cost $a(\text{splay}(x))$ of the splay operation is the sum of all of the splay-step operations performed on x. Suppose that we perform k splay-step operations on x. Let $r_0(x)$ be the rank of x before the splay operation. Let $r_i(x)$ be the rank of x after the i^{th} splay-step operation. Then we have $r_k(x) = r_0(root)$ and:

$$\begin{aligned} a(\text{splay}(x)) &\leq 3(r_k(x) - r_{k-1}(x)) + 3(r_{k-1}(x) - r_{k-2}(x)) + ... + 3(r_1(x) - r_0(x)) + 1 \\ &= 3(r_k(x) - r_0(x)) + 1 \\ &= 3(r_0(root) - r_0(x)) + 1. \end{aligned}$$

The added 1 comes from the possibility of a case 1 splay-step at the end. The definition of r gives the result. □

The above lemma gives the amortized cost of a splay operation, for any settings of the weights. To be able to get good bounds on the total cost of any sequence of operations, we set $w(x) = 1$ for all nodes x. This implies that $s(root) \leq n$ where n is the total number of nodes ever in the BST, and by Lemma 2, the amortized cost of any splay operation is $a(\text{splay}(x)) = O(\log n)$.

3.3 Amortized Cost of BST operations

We now need to show how to implement the various BST operations, and analyze their (amortized) cost (still with the weights set to 1).

3.3.1 Find

Finding an element in the splay tree follows the same behavior as in a BST. After we find our node, we splay it, which is $O(\log n)$ amortized cost. The cost of going down the tree to find the node can be charged to the cost of splaying it. Thus, the total amortized cost of Find is $O(\log n)$. (Note: if the node is not found, we splay the last node reached.)

3.3.2 Find-Min

This operation will only go down the left children, until none are left, and this cost will be charged to the subsequent splay operation. After we find the min node, we splay it, which takes $O(\log n)$ amortized cost. The total amortized cost is then $O(\log n)$.

3.3.3 Find-Max

The process for this is the same as for Find-Min, except we go down the right child. The total amortized cost of this is $O(\log n)$ as well.

3.3.4 Join

Given two trees T_1 and T_2 with $\text{key}(x) < \text{key}(y)\ \forall x \in T_1, y \in T_2$, we can join T_1 and T_2 into one tree with the following steps:

1. Find-Max(T_1). This makes the max element of T_1 the new root of T_1.
2. Make T_2 the right child of this.

The amortized cost of the first step is $O(\log n)$. For the second step, the actual cost is 1, but we need to take into account in the amortized cost the increase in the potential function value. Before step 2, T_1 and T_2 had a potential function value of $\phi(T_1)$ and $\phi(T_2)$. After it, the resulting tree has a potential function value $\leq \phi(T_1) + \phi(T_2) + \log n$, since the rank of the new root is $\leq log(n)$. So the amortized cost of Join is $O(\log n)$.

3.3.5 Split

Given a tree T and a pivot i, the split operation partitions T into two BSTs:

$$T_1 : \{x \,|\, \text{key}(x) \leq i\},$$

$$T_2 : \{x \,|\, \text{key}(x) > i\}.$$

We split the tree T by performing Find(i). This Find will then splay on a node, call it x, which brings it to the root of the tree. We can then cut the tree; everything on the right of x belongs to

T_2, and everything on the left belongs to T_1. Depending on its key, we add x to either T_1 or T_2. Thus, we either make the right child or the left child of x a new root by simply removing its pointer to its parent.

The amortized cost of the FIND operation is $O(\log n)$. The actual cost of creating the second BST (by cutting off one of the children) is just $O(1)$, and the potential function does not increase (as the rank of the root does not increase). Thus the total amortized time of a SPLIT is also $O(\log n)$ time.

JOIN and SPLIT make insertion and deletion very simple.

3.3.6 INSERT

Let i be the value we want to insert. We can first split the tree around i. Then, we let node i be the new root, and make the two subtrees the left and right subtrees of i respectively. The amortized cost again is $O(logn)$.

3.3.7 DELETE

To delete a node i from a tree T, we first FIND(i) in the tree, which brings node i to the root. We then delete node i, and are left with its left and right subtrees. Because everything in the left subtree has key less than everything in the right subtree, we can then join them. It is easy to see that this has amortized cost $O(\log n)$ as well.

3.3.8 Total cost of m operations

The next theorem shows that the cost of any sequence of operations on a splay tree has worst-case time similar to any balanced BST (unless the number of operations m is $o(n)$ where n is the number of keys).

Theorem 3 *For any sequence of m operations on a splay tree containing at most n keys, the total cost is $O((m+n)\log n)$.*

Proof of Theorem 3: Let a_i be the amortized cost of the i^{th} operation. Let c_i be the real cost of the i^{th} operation. Let ϕ_0 be the potential before and ϕ_m be the potential after the m operations. The total amortized cost of m operations is:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + \phi_m - \phi_0.$$

Then we have:

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} a_i + \phi_0 - \phi_m.$$

Since we chose $w(x) = 1$ for all x, we have that, for any node x, $r(x) \leq \log n$. Thus $\phi_0 - \phi_m \leq n \log n$, so we conclude:

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} a_i + O(n \log n) = O(m \log n) + O(n \log n) = O((m+n)\log n).$$

□

4 Comparison to other BSTs

4.1 Static Optimality Property

We will show that splay trees are competitive against any binary search tree that does not involve any rotations. We consider BSTs containing n keys, and sequences of operations that contain only FIND operations (thus, no INSERT or DELETE for example).

Theorem 4 *Define a static binary search tree to be one that uses no rotation operations. Let m_i be the number of times element i is accessed for $i = 1, \ldots, n$. We assume $m_i \geq 1$ for all i. Then the total cost for accessing every element i m_i times is at most a constant times the total cost of any static binary search tree.*

Proof of Theorem 4: Consider any binary search tree T rooted at t. Let $l(i)$ be the height of of i in T, or the number of nodes on the path from i to the root of T, so $l(t) = 1$. In T, the cost for accessing an element i is $l(i)$, so the total cost for accessing every element i m_i times is $\sum_i l(i)m_i$. We want to show that the total cost of operations on a splay tree, irrespective of the starting configuration, is $O(\sum_i l(i)m_i)$.

We choose a different weight function that earlier. Here, we define the weights to be $w(i) = 3^{-l(i)}$ for all i. Note that $s(t) \leq \frac{1}{3} + 2(\frac{1}{3^2}) + 2^2(\frac{1}{3^3}) + \ldots = 1$. Then, by Lemma 2, the amortized cost of finding i is:

$$a(i) = O(1 + \log_2 \frac{s(t)}{s(i)}) = O(1 + \log_2 \frac{1}{3^{-l(i)}}) = O(1 + l(i)).$$

The total amortized cost of accessing every element i m_i times on a splay tree is thus:

$$O(m + \sum_i l(i)m_i) = O\left(\sum_i l(i)m_i\right).$$

This is the amortized cost, we now need to argue about the actual cost. Let ϕ be the potential before the beginning of the sequence, and ϕ' be the potential after the sequence of operations. For a node i, let $r(i)$ be the rank of i before and $r'(i)$ be the rank after the operations. Note that (since $r(i) \leq \log_2 1$ and $r'(i) \geq \log_2 w(i)$):

$$\phi - \phi' = \sum_i r(i) - r'(i) \leq \sum_i \log_2 \frac{1}{3^{-l(i)}} = O\left(\sum_i l(i)\right).$$

Then we have:

$$\sum c_i = \sum a_i + \phi - \phi' = O\left(\sum_i l(i)m_i\right) + O\left(\sum_i l(i)\right) = O\left(\sum_i l(i)m_i\right),$$

since our assumption $m_i \geq 1$ implies that $\sum_i l(i) \leq \sum_i l(i)m(i)$. □

4.2 Dynamic Optimality Conjecture

The Dynamic Optimality Conjecture claims that Splay Trees are efficient up to a constant factor to any self-adjusting Binary Search Tree (allowing an arbitrary number of (arbitrary) rotations between accesses). This conjecture was first put forth in the Tarjan and Sleater's original Splay Tree paper in 1985, and has withstood attempts to prove or disprove it since.

4.3 Scanning Theorem

The scanning theorem states that, for a splay tree that contains the values $[1, 2, \ldots, n]$, accessing all of those elements in sequential order takes $O(n)$ time, regardless of the initial arrangement of the tree. An interesting point is that, even though the Scanning Theorem has been proved, if the Dynamic Optimality Conjecture were true, then it would follow directly from the fact that one can create dynamic BST's that perform sequential access in linear time.