SPLAY TREES:
Self-Adjusting Binary Search Trees

In 1983, D.D. Sleator and R.E. Tarjan invented an interesting data structure for implementing the dictionary abstract data type, called a *splay tree*, a self-adjusting version of binary search trees. A self-adjusting BST is a BST maintained in a particular way.

Throughout this handout, m denotes the number of dictionary operations (INSERT, DELETE or SEARCH) that are processed and $n(\leq m)$ is the maximum number of keys in the dictionary at any time.

In balanced BSTs (e.g. red-black trees) we are very careful to maintain the BST height-balanced at all times. This ensures that the tree will have height at most $O(lg\ n)$. Since each operation takes time at most proportional to the height of the tree, each individual operation takes at most $O(lg\ n)$ time. The time needed to process m operations is therefore $O(m\ lg\ n)$.

Self-adjusting trees have the interesting property that they achieve the same time bound, even though it is *not* the case that each individual operation takes $O(lg\ n)$ time. Some take more (in fact, some may take as long as $\Theta(n)$) and some take less, so that the overall time complexity comes to $O(m\ lg\ n)$. We cannot hope to derive this bound simply by obtaining an upper bound for the worst-case time complexity of each individual operation and multiplying that bound by the number of operations performed: Since some operations may take as long as $\Theta(n)$ time, this method would yield an upper bound of $O(mn)$ — much higher than the claimed $O(m\ lg\ n)$.

The key idea in self-adjusting BSTs is the so-called *splay operation*, which modifies the structure (but not the contents) of a BST. A splay operation at node x of a BST consists of performing the following actions repeatedly, until x has become the root of the tree†; each application of these actions will be called a *step* of the splay:

- **Case 1:** if x is the left (resp. right) child of the root y then perform a right (resp. left) rotation at y (*cf.* Figure 1(a)). This is called the *zig* case.
- **Case 2:** if x is the left (resp. right) child of y and y is the left (resp. right) child of z then perform a right (resp. left) rotation at z, followed by another right (resp. left) rotation at y (*cf.* Figure 1(b)). This is called the *zig-zig* case.
- **Case 3:** if x is the right (resp. left) child of y and y is the left (reps. right) child of z then perform a left (resp. right) rotation at y, followed by a right (resp. left) rotation at z (*cf.* Figure 1(c)). This is called the *zig-zag* case.

Note that after the splay operation at node x is completed, x is the root of the resulting tree. An example of the effect of an entire splay is shown in Figure 2. Note the drastic effect of that operation on the structure of the tree.

Exercise 1: Write a simple algorithm for the *splay* operation. □

† Note the resemblance of this idea with the move-to-front heuristic on linear lists.

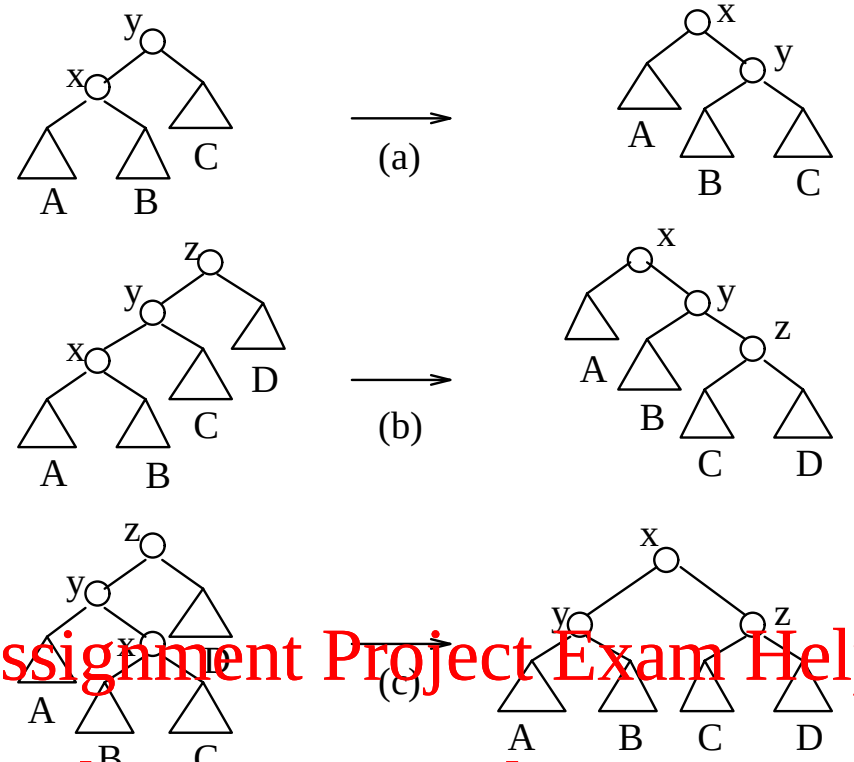

Figure 1: A splaying step. The node accessed is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig-zig: two single rotations. (c) Zig-zag: double rotation.

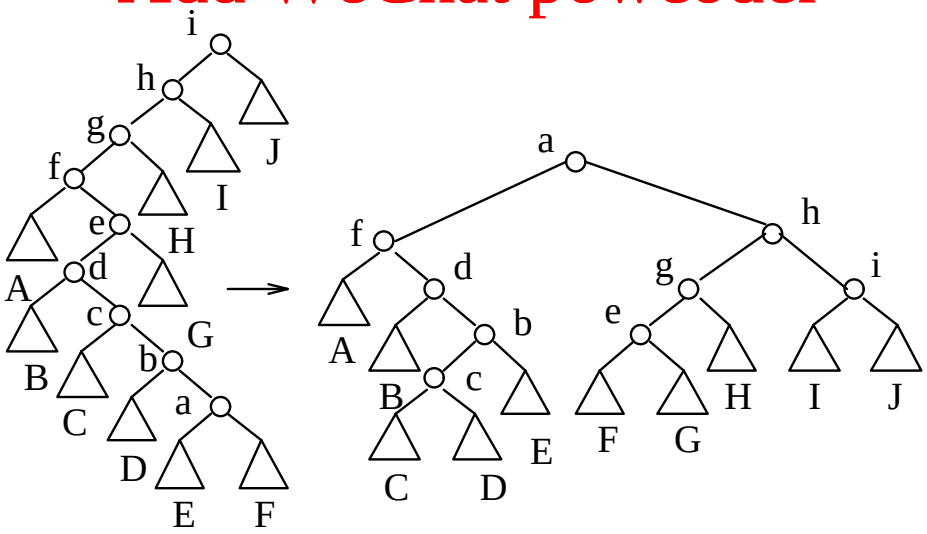

Figure 2: A splay operation at node a.

The INSERT, DELETE and SEARCH operations on self-adjusting BSTs are implemented as in plain BSTs, except that after each such operation a splay operation is applied to the deepest accessed *node* which is still in the BST. Specifically, after an INSERT or SEARCH operation, a splay is applied at the node inserted or searched (in the event of an unsuccessful search, the splay

is applied at the last node accessed during the search); and after a DELETE operation, a splay is applied at the node that was the parent of the deleted *node* (which is not necessarily the deleted *key*!).

It is important to note that in self-adjusting BSTs the structure of the tree is modified every time a search is performed. Thus, the shape of the tree depends not only on the order in which keys are inserted or deleted but also on the order in which keys are searched. (In contrast, the structure of red-black trees is modified only when nodes are inserted or deleted.) In this sense, self-adjusting BSTs are reminiscent of self-adjusting linear lists. The splay operation can be viewed as a heuristic that makes the most recently searched key the root of the BST, much the same way the Move-to-Front heuristic achieves the analogous property in self-adjusting linear lists. The amazing property of self-adjusting BSTs is that they achieve the $O(m \lg n)$ time complexity for any sequence of m operations as long as there are at most n keys in the tree at any point in time!

Some Motivation

Let T be a BST with n nodes. We will associate with T a non-negative real number, called the *potential* of T, which intuitively is a measure of the "balance" of T, i.e. a small number indicates T is almost a complete binary tree with the length of the longest path approximately $\lg n$, and a large number indicates T is almost a singly linked list with the length of the longest path approximately n. One such natural measure on T is the following. For each node u in T, let $l_T(u)$ be the number of nodes along the path from the root to u inclusive. (Note that $l_T(u)$ is the number of comparisons to reach node u in T.) Define $L(T) = \sum_{u \in T} l_T(u)$ (If T is the empty tree then define $L(T) = 0$). $L(T)$ is called the internal path length of T, and $L(T)/n$ is the average number of comparisons for a successful search in T assuming every node is equally probable. If $L(T)$ is $O(n \lg n)$ then T is almost a complete binary tree, whereas if $L(T)$ is $\Omega(n^2)$ then T is almost a singly linked list.

A measure equivalent to $L(T)$ is the following. For each node u in T, the weight of u, $w_T(u)$, is the number of nodes in the subtree rooted at u (including node u). Define $W(T) = \sum_{u \in T} w_T(u)$. Let us also use the notation $|T|$ to denote the number of nodes in tree T, i.e. the weight of the root of T.

Exercise 2: Prove that $W(T) = L(T)$. □

Thus, $W(T)$ is also a measure of the balance of T.

Definition: It turns out that for self-adjusting trees, the analysis requires a different measure of the balance of T. For each node u in T, define the *rank* of u, $r_T(u)$, to be the logorithm of its weight, i.e., $r_T(u) = \lg w_T(u)$. Define the *potential* (or the *rank*) of T to be $\Phi(T) = \sum_{u \in T} r_T(u)$ (If T is the empty tree then $\Phi(T) = 0$). The potential of a forest is the sum of the potentials of its trees. We may omit the subscript T from $r_T(u)$ or $w_T(u)$ of a node u when no confusion arises. □

Exercise 3: Prove that a BST T which has the smallest (largest) $\Phi(T)$ among all BSTs on n nodes is completely balanced (completely unbalanced). What is the minimum (maximum) value of $\Phi(T)$ possible? □

To further motivate the definition of the potential of a BST, suppose we have a BST T with the following *rank property*: For each node u in T with children v and w, $|r_T(v) - r_T(w)| \leq 1$. Intuitively, each node in a BST with this rank property has at most twice as many nodes in the left subtree as in the right subtree and vice versa.

Exercise 4: Prove that a BST T which has the smallest $\Phi(T)$ among all BSTs with n nodes has the above rank property. □

Exercise 5: Prove that the height of an n –node BST with the above rank property is $O(lg\ n)$. □

Thus, $\Phi(T)$ is a measure of how close T is to having the rank property, and is thus a measure of the balance of T. The idea behind self adjusting trees is that when a dictionary operation ends at a node which is a long way from the root, the ensuing splay operation which moves the node to the root substantially reduces the potential of the tree, i.e. the resulting tree is much better balanced.

The Analysis of Self-Adjusting BSTs

We claim that the total time for all operations is proportional to the total number of splay steps performed in splay operations. To justify this claim, note that each splay step only takes a constant number of pointer changes. Furthermore, the time for an INSERT, DELETE or SEARCH operation in a self-adjusting BST is dominated by the time for the corresponding splay operation, and the total time for the splay operations is the total time for all splay steps within the splay operations. Thus, the upper bound we derive on the total number of splay steps is to within a constant multiplicative factor an upper bound on the total time. Hence, in what follows we will count the number of single rotations as a measure of the running time. Consider a single splay step which changes tree T to tree T'. This splay step has actual running time $t = 1$ if it is a zig, and $t = 2$ if it is a zig-zig or a zig-zag. The amortized running time is $t + \Phi(T') - \Phi(T)$. We first need the following observation:

Lemma 0 (THE LOG LEMMA): Consider positive real numbers a, b, and c, such that $a + b \leq c$. Then, $lg\ a + lg\ b \leq 2\ lg\ c - 2$.

Proof: One way to prove this lemma is to observe that the logarithm function is concave (i.e., the second derivative is non-positive). If $f(x)$ is a concave function, then

$$\frac{f(a) + f(b)}{2} \leq f(\frac{a+b}{2}).$$

The above inequality says the graph of function $f(.)$ bends down, and hence the function value at the average point $(a + b)/2$ is at or above the line segment that passes through the points $f(a)$ and $f(b)$. Now, simply plug in $lg(.)$ for $f(.)$ and simplify the above eniquality.

Another way to obtain the lemma is using the following: $c^2 \geq (a + b)^2 = (a - b)^2 + 4ab \geq 4ab$. Thus, $4ab \leq c^2$. Now take logarithm from both sides. □

Lemmas 1, 2, and 3 below are used in the proof of the main theorem, Theorem 1.

Lemma 1 : The amortized time of a *zig* splay step at node x, which transforms tree T to T' is

$$a = 1 + \Phi(T') - \Phi(T) \leq 1 + 3[\ r_{T'}(x) - r_T(x)\ ]$$

Proof: Referring to Figure 1(a), we see that the splay step may only change the ranks of nodes x and y, the ranks of other nodes remain unchanged. Thus we have

$$a = 1 + \Phi(T') - \Phi(T) = 1 + [\, r_{T'}(x) + r_{T'}(y)\,] - [\, r_T(x) + r_T(y)\,]$$

$$\leq 1 + [\, r_{T'}(x) - r_T(x)\,] \qquad \textit{since } r_{T'}(y) \leq r_T(y)$$

$$\leq 1 + 3[\, r_{T'}(x) - r_T(x)\,] \qquad \textit{since } r_{T'}(x) \geq r_T(x)$$

This completes the proof. □

Lemma 2 : The amortized time of a *zig-zig* splay step at node x, which transforms tree T to tree T' is

$$a = 2 + \Phi(T') - \Phi(T) \leq 3[\, r_{T'}(x) - r_T(x)\,]$$

Proof: Referring to Figure 1(b), we have

$$a = 2 + \Phi(T') - \Phi(T) = 2 + [\, r_{T'}(x) + r_{T'}(y) + r_{T'}(z)\,] - [\, r_T(x) + r_T(y) + r_T(z)\,]$$

$$= 2 + r_{T'}(y) + r_{T'}(z) - r_T(x) - r_T(y) \qquad \textit{since } r_{T'}(x) = r_T(z)$$

$$\leq 2 + r_{T'}(x) + r_{T'}(z) - 2r_T(x) \qquad \textit{since } r_T(x) \leq r_T(y) \;\&\; r_{T'}(x) \geq r_{T'}(y)$$

Now, from Figure 1(b) we notice that $w_T(x) + w_{T'}(z) < w_{T'}(x)$. Using Lemma 0 (the log lemma), we get $r_T(x) + r_{T'}(z) \leq 2r_{T'}(x) - 2$. Hence, $r_{T'}(z) \leq 2r_{T'}(x) - r_T(x) - 2$. Plugging this inequality above, we get:

$$a \leq 2 + r_{T'}(x) + 2r_{T'}(x) - r_T(x) - 2 - 2r_T(x)$$

$$= 3[\, r_{T'}(x) - r_T(x)\,]$$

This completes the proof. □

Lemma 3 : The amortized time of a *zig-zag* splay step at a node x, which transforms tree T to tree T' is

$$a = 2 + \Phi(T') - \Phi(T) \leq 3[\, r_{T'}(x) - r_T(x)\,]$$

Proof: Referring to Figure 1(c), we have

$$a = 2 + \Phi(T') - \Phi(T) = 2 + [\, r_{T'}(x) + r_{T'}(y) + r_{T'}(z)\,] - [\, r_T(x) + r_T(y) + r_T(z)\,]$$

$$= 2 + r_{T'}(y) + r_{T'}(z) - r_T(x) - r_T(y) \qquad \textit{since } r_{T'}(x) = r_T(z)$$

$$\leq 2 + [\, r_{T'}(y) + r_{T'}(z)\,] - 2r_T(x) \qquad \textit{since } r_T(x) \leq r_T(y)$$

Now, from Figure 1(c) we notice that $w_{T'}(y) + w_{T'}(z) < w_{T'}(x)$. Using Lemma 0 (the log lemma), we get $r_{T'}(y) + r_{T'}(z) \leq 2r_{T'}(x) - 2$. Plugging this inequality above, we get:

$$a \le 2 + [\, 2r_{T'}(x) - 2\,] - 2r_T(x)$$

$$= 2[\, r_{T'}(x) - r_T(x)\,]$$

$$\le 3[\, r_{T'}(x) - r_T(x)\,] \qquad \textit{since } r_{T'}(x) \ge r_T(x)$$

This completes the proof. □

Theorem 1 : The amortized number of single rotations in a splay operation on a BST with n nodes is at most $3 \lg n + 1$.

Proof: Suppose the splay operation occurs at node $x = w_0$ in T as shown in Figure 3. Suppose T' is the tree after the splay operation.

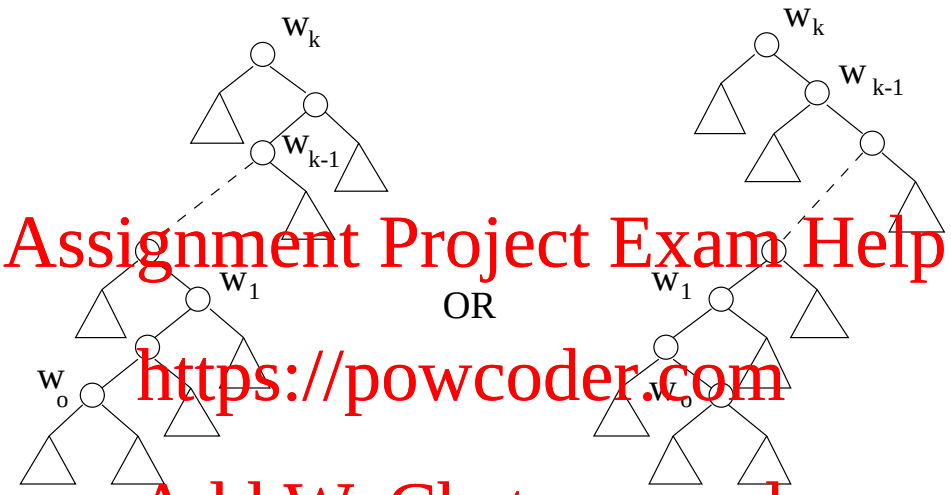

Figure 3. The last splay step may or may not be a zig operation. Node w_{k-1} may be a child or grandchild of the root w_k.

Node $x = w_0$ is moved to the root w_k by a series of k splay steps, successively moving through nodes $w_1, \cdots, w_k$. Let $T_0 = T$, and T_j be the tree after the j^{th} splay step, for $j = 1, \cdots, k$. (Note that $T' = T_k$.) By looking at how the splay steps work (Figure 1) it is easy to see that after the j^{th} splay step, the rank of x is

$$r_{T_j}(x) = r_T(w_j) \qquad \text{for } \; j = 0, 1, \cdots, k \tag{1}$$

Let t denote the actual number of single rotations in this splay operation, and t_j denote the number of single rotations (i.e., 1 or 2) in the j-th splay step. (Note that $t = \sum_{j=1}^{k} t_j$.) Then, the amortized number of single rotations is:

$$t + \Phi(T') - \Phi(T) = t + \Phi(T_k) - \Phi(T_0) = \sum_{j=1}^{k} [\, t_j + \Phi(T_j) - \Phi(T_{j-1})\,] \tag{2}$$

Note how the above summation telescopes and that the summand is the amortized time of the j-th splay step. From Lemmas 1, 2, and 3, we see that the amortized time of the j-th splay step is

$$t_j + \Phi(T_j) - \Phi(T_{j-1}) \le 3[\, r_{T_j}(x) - r_{T_{j-1}}(x)\,] \qquad \text{for } \; j = 1, 2, \cdots, k-1 \tag{3}$$

and

$$t_k + \Phi(T_k) - \Phi(T_{k-1}) \le 1 + 3[\, r_{T_k}(x) - r_{T_{k-1}}(x) \,] \qquad (4)$$

The difference between equations (3) and (4) is that only the last splay step can be a *zig*, while the others are either *zig-zig* or *zig-zag* steps.

Now if we sum up equations (3) and (4), and appropriately telescope the summation, we get:

$$\sum_{j=1}^{k} [\, t_j + \Phi(T_j) - \Phi(T_{j-1}) \,] \le 1 + 3[\, r_{T_k}(x) - r_{T_0}(x) \,] \qquad (5)$$

But from equation (1) we see that $r_{T_k}(x) = r_T(w_k) = lg\ n$ and $r_{T_0}(x) = r_T(w_0) \ge 0$. Therefore, from equations (2) and (5) we get:

$$t + \Phi(T') - \Phi(T) \le 1 + 3\ lg\ n$$

This completes the proof of the theorem. □

Exercise 6: Prove that in any BST T with n nodes, an INSERT operation, which adds a new leaf node, increases the potential of T by at most $lg\ n$, and a DELETE operation cannot increase the potential. (These bounds do not include the corresponding splay operation that takes place right after the INSERT/DELETE.) Use this to show that the amortized time of any dictionary operation on a splay tree of n nodes is at most $4\ lg\ n + 1$.

Corollary : The total number of single rotations is at most $m\,(\,4\ lg\ n + 1\,)$ for any sequence of m dictionary operations which starts with an empty splay tree and for which the splay tree has at most n keys at any point in time.

Definition: Suppose A is a BST and x is an item (which may or may not be in A). The operation *split(x,A,B,C)* partitions the items of A into two BSTs B and C such that B contains items of A that are $\le x$, and C contains items of A that are $> x$. The BST A is destroyed in the process. □

Definition: Suppose A and B are two BSTs such that every item in A is less than every item in B (i.e. the maximum item in A is smaller than the minimum item in B). Then the operation *join(A,B)* (or *concatenate*) returns a BST which contains the union of items in A and B. This operation destroys A and B in the process. □

Note that we can use *split* and *join* operations to "cut-and-paste" various BSTs.

Exercise 7: Write algorithms for *split* and *join* on splay trees using the splay operation. □

Comparison of Balanced BSTs and Self-Adjusting BSTs

One advantage of self-adjusting BSTs over balanced BSTs (e.g. red-black trees) is that they do not require any balance information to be maintained. This not only saves space but, possibly more significantly, makes the algorithms for the dictionary operations simpler to programme.

Another advantage of self-adjusting trees is that, true to their name, their structure adapts to the access pattern dynamically. This means that (as in self-adjusting linear lists) keys frequently searched for will tend to be close to the root of the tree. If the access pattern changes over time, so will the structure of the tree. In fact, it is possible to prove that the time to process any (sufficiently long) sequence of search operations on a self-adjusting binary search tree is within a constant factor of the time needed to process that sequence on an optimal (static) binary search tree! You can find a proof of this fact in the original paper of Sleator and Tarjan [ST85]. This paper,

besides this static optimality theorem, contains a number of other claims and conjectures (e.g., the dynamic optimality conjecture).

On the negative side, operations on self-adjusting trees cause more rotations than on red-black trees. Another potential disadvantage of self-adjusting trees is that, even though the overall time to perform a sequence of m operations is (asymptotically) the same as in red-black trees, some individual operations may take much longer ($\Omega(n)$, rather than $O(\lg n)$). This would be a problem in real-time applications where it is important to produce the result of each individual operation quickly because a user is waiting for it; it is not an issue in ‘‘batch’’ applications where the user is only interested in the result of the entire sequence of operations, not each one separately.

Let us also mention that the splay operation as defined in this note is a bottom-up operation. There are other variations of the operation with similar amortized results. One such example is the top-down spaly which has a much simpler iterative implementation. See the paper [ST85] or [Weiss] for details.

We have seen that the move-to-front heuristic has a rather strong optimality property (to within a constant factor) among a class of algorithms that operate on linear lists. In their original paper Sleator and Tarjan [ST85] proposed the following optimality conjecture for splay trees:

Dynamic Optimality Conjecture: Consider any sequence of dictionary operations starting with an n-node binary search tree. Let A be any (or the optimal off-line) algorithm that carries out each operation by traversing the corresponding search path at a cost of one per node along the path, and that between each operation performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform the same sequence of operations by splaying is no more than $O(n + T)$, where T is the time taken by algorithm A to perform the same sequence of operations. □

This is a very strong claim. Some important special cases of it have already been proven, but the general conjecture remains an open question. For instance, the paper [Tar85] shows that it takes only linear time to access/splay each of the n nodes of an n-node BST in symmetric order (i.e., inorder). The balanced BST’s do not have this property (they take $\Theta(n \lg n)$ time). For a more recent account on the (special case of the) dynamic conjecture see [CMSS00].

There are numerous animations of splay trees that can be obtained from the web. For instance, see [WEB].

References

[WEB] Here are a couple of splay tree animations on the web: http://gs213.sp.cs.cmu.edu/cgi-bin/splay and http://langevin.usc.edu/BST/

[CMSS00] R. Cole, B. Mishra, J. Schmidt, A. Siegel, "On the Dynamic Finger Conjecture for Splay Trees. Parts I and II", pages 1-85, SIAM J. on Computing 30(1), April 2000.

[ST85] D.D. Sleator and R.E. Tarjan, ‘‘Self-adjusting binary search trees’’, Journal of ACM, vol. 32, no. 3, 1985, pp. 652-686.

[Tar85] R.E. Tarjan, ‘‘Sequential access in splay trees takes linear time’’, Combinatorica 5(4), 1985, pp. 367-378.

[Weiss] M.A. Weiss, ‘‘Data Structures and Algorithm Analysis in C++’’, Third Edition, Addison Wesley, 2006.