Splay tree

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan in 1985.[1]

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this with the basic search operation is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Splay tree		
Type	tree	
Invented	1985	
Invented by	Daniel Dominic Sleator and Robert Endre Tarjan	
Time complexity in big O notation		
Algorithm	**Average**	**Worst case**
Space	O(n)	O(n)
Search	O(log n)	amortized O(log n)
Insert	O(log n)	amortized O(log n)
Delete	O(log n)	amortized O(log n)

Contents

Advantages

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height—though unlikely—is O(n), with the average being O(log *n*). Having frequently used nodes near the root is an advantage for many practical applications (also see Locality of reference), and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Comparable performance: Average-case performance is as efficient as other trees.[2]
- Small memory footprint: Splay trees do not need to store any bookkeeping data.

Disadvantages

The most significant disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all n elements in non-decreasing order. Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be high. However the amortized access cost of this worst case is logarithmic, O(log n). Also, the expected access cost can be reduced to O(log n) by using a randomized variant.[3]

The representation of splay trees can change even when they are accessed in a 'read-only' manner (i.e. by *find* operations). This complicates the use of such splay trees in a multi-threaded environment. Specifically, extra management is needed if multiple threads are allowed to perform *find* operations concurrently. This also makes them unsuitable for general use in purely functional programming, although even there they can be used in limited ways to implement priority queues.

Operations

Splaying

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

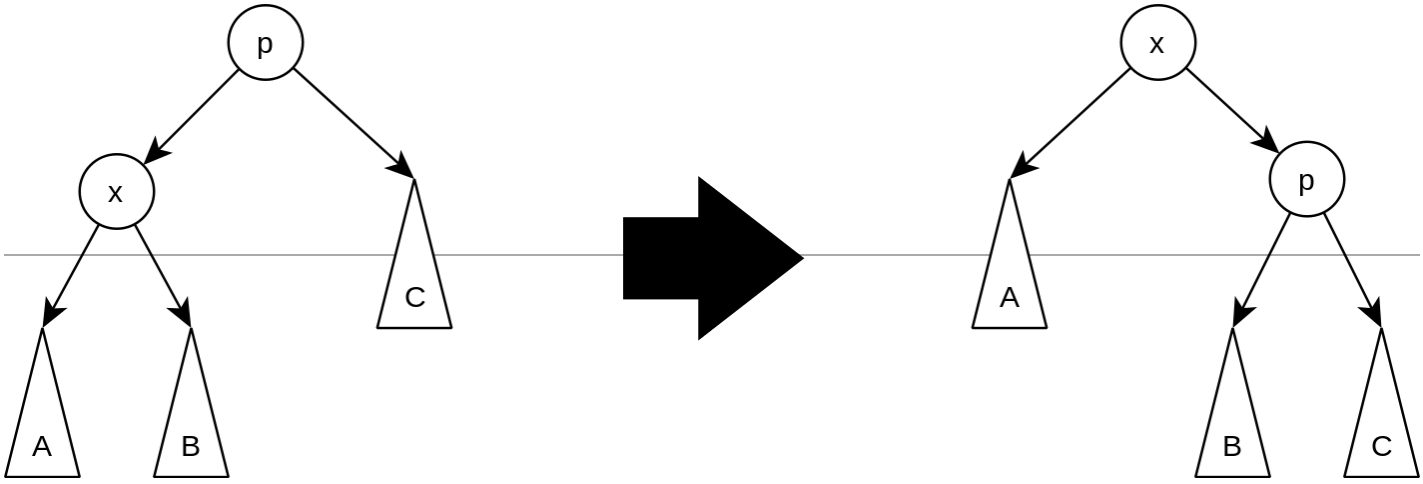
Each particular step depends on three factors:

- Whether x is the left or right child of its parent node, p,
- whether p is the root or not, and if not
- whether p is the left or right child of *its* parent, g (the *grandparent* of x).

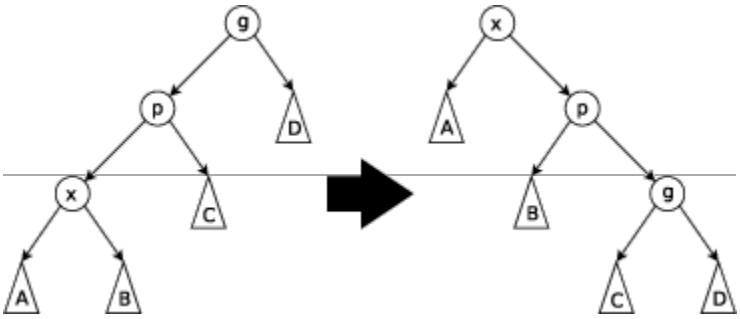
It is important to remember to set *gg* (the *great-grandparent* of x) to now point to x after any splay operation. If *gg* is null, then x obviously is now the root and must be updated as such.

There are three types of splay steps, each of which has a left- and right-handed case. For the sake of brevity, only one of these two is shown for each type. These three types are:

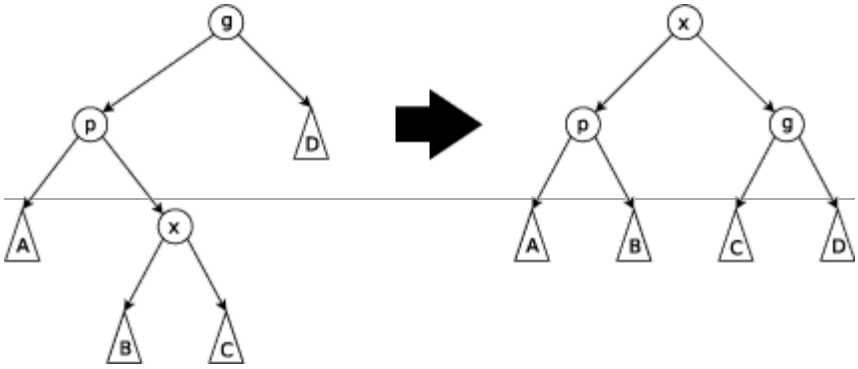
Zig step: this step is done when p is the root. The tree is rotated on the edge between x and p. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when x has odd depth at the beginning of the operation.



Zig-zig step: this step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and *p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro[4] prior to the introduction of splay trees.



Zig-zag step: this step is done when *p* is not the root and *x* is a right child and *p* is a left child or vice versa. The tree is rotated on the edge between *p* and x, and then rotated on the resulting edge between *x* and g.



Join

Given two trees S and T such that all elements of S are smaller than the elements of T, the following steps can be used to join them to a single tree:

- Splay the largest item in S. Now this item is in the root of S and has a null right child.
- Set the right child of the new root to T.

Split

Given a tree and an element *x*, return two new trees: one containing all elements less than or equal to *x* and the other containing all elements greater than *x*. This can be done in the following way:

- Splay *x*. Now it is in the root so the tree to its left contains all elements smaller than *x* and the tree to its right contains all element larger than *x*.
- Split the right subtree from the rest of the tree.

Insertion

To insert a value *x* into a splay tree:

- Insert *x* as with a normal binary search tree.
- when an item is inserted, a splay is performed.
- As a result, the newly inserted node *x* becomes the root of the tree.

ALTERNATIVE:

- Use the split operation to split the tree at the value of *x* to two sub-trees: S and T.
- Create a new tree in which *x* is the root, S is its left sub-tree and T its right sub-tree.

Deletion

To delete a node x, use the same method as with a binary search tree: if x has two children, swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

ALTERNATIVE:

- The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. leaves the tree with two sub trees.
- The two sub-trees are then joined using a "join" operation.

Implementation and variants

Splaying, as mentioned above, is performed during a second, bottom-up pass over the access path of a node. It is possible to record the access path during the first pass for use during the second, but that requires extra space during the access operation. Another alternative is to keep a parent pointer in every node, which avoids the need for extra space during access operations but may reduce overall time efficiency because of the need to update those pointers.[1]

Another method which can be used is based on the argument that we can restructure the tree on our way down the access path instead of making a second pass. This top-down splaying routine uses three sets of nodes - left tree, right tree and middle tree. The first two contain all items of original tree known to be less than or greater than current item respectively. The middle tree consists of the sub-tree rooted at the current node. These three sets are updated down the access path while keeping the splay operations in check. Another method, semisplaying, modifies the zig-zig case to reduce the amount of restructuring done in all operations.[1][5]

Below there is an implementation of splay trees in C++, which uses pointers to represent each node on the tree. This implementation is based on bottom-up splaying version and uses the second method of deletion on a splay tree. Also, unlike the above definition, this C++ version does *not* splay the tree on finds - it only splays on insertions and deletions, and the find operation, therefore, has linear time complexity.

```
#include <functional>

#ifndef SPLAY_TREE
#define SPLAY_TREE

template<typename T, typename Comp = std::less<T>>
class splay_tree {
private:
  Comp comp;
  unsigned long p_size;

  struct node {
    node *left, *right;
    node *parent;
    T key;
    node( const T& init = T( ) ) : left( nullptr ), right( nullptr ), parent( nullptr ), key( init ) { }
    ~node( ) {

    }
  } *root;

  void left_rotate( node *x ) {
    node *y = x->right;
    if(y) {
      x->right = y->left;
      if( y->left ) y->left->parent = x;
      y->parent = x->parent;
    }

    if( !x->parent ) root = y;
    else if( x == x->parent->left ) x->parent->left = y;
    else x->parent->right = y;
    if(y) y->left = x;
    x->parent = y;
```

```
  }

  void right_rotate( node *x ) {
    node *y = x->left;
    if(y) {
      x->left = y->right;
      if( y->right ) y->right->parent = x;
      y->parent = x->parent;
    }
    if( !x->parent ) root = y;
    else if( x == x->parent->left ) x->parent->left = y;
    else x->parent->right = y;
    if(y) y->right = x;
    x->parent = y;
  }

  void splay( node *x ) {
    while( x->parent ) {
      if( !x->parent->parent ) {
        if( x->parent->left == x ) right_rotate( x->parent );
        else left_rotate( x->parent );
      } else if( x->parent->left == x && x->parent->parent->left == x->parent ) {
        right_rotate( x->parent->parent );
        right_rotate( x->parent );
      } else if( x->parent->right == x && x->parent->parent->right == x->parent ) {
        left_rotate( x->parent->parent );
        left_rotate( x->parent );
      } else if( x->parent->left == x && x->parent->parent->right == x->parent ) {
        right_rotate( x->parent );
        left_rotate( x->parent );
      } else {
        left_rotate( x->parent );
        right_rotate( x->parent );
      }
    }
  }

  void replace( node *u, node *v ) {
    if( !u->parent ) root = v;
    else if( u == u->parent->left ) u->parent->left = v;
    else u->parent->right = v;
    if( v ) v->parent = u->parent;
  }

  node* subtree_minimum( node *u ) {
    while( u->left ) u = u->left;
    return u;
  }

  node* subtree_maximum( node *u ) {
    while( u->right ) u = u->right;
    return u;
  }
public:
  splay_tree( ) : root( nullptr ), p_size( 0 ) { }

  void insert( const T &key ) {
    node *z = root;
    node *p = nullptr;

    while( z ) {
      p = z;
      if( comp( z->key, key ) ) z = z->right;
      else z = z->left;
    }

    z = new node( key );
    z->parent = p;

    if( !p ) root = z;
    else if( comp( p->key, z->key ) ) p->right = z;
    else p->left = z;

    splay( z );
    p_size++;
  }

  node* find( const T &key ) {
    node *z = root;
    while( z ) {
```

```
      if( comp( z->key, key ) ) z = z->right;
      else if( comp( key, z->key ) ) z = z->left;
      else return z;
    }
    return nullptr;
  }

  void erase( const T &key ) {
    node *z = find( key );
    if( !z ) return;

    splay( z );

    if( !z->left ) replace( z, z->right );
    else if( !z->right ) replace( z, z->left );
    else {
      node *y = subtree_minimum( z->right );
      if( y->parent != z ) {
        replace( y, y->right );
        y->right = z->right;
        y->right->parent = y;
      }
      replace( z, y );
      y->left = z->left;
      y->left->parent = y;
    }

    delete z;
    p_size--;
  }

/* //the alternative implementation
    void erase( const T &key) {
        node *z = find( key );
        if( !z ) return;

        splay( z );

        node *s = z->left;
        node *t = z->right;
        delete z;

        node *sMax = NULL;
        if(s) {
            s->parent = NULL;
            sMax = subtree_maximum(s);
            splay(sMax);
            root = sMax;
        }
        if(t) {
            if(s)
                sMax->right = t;
            else
                root = t;
            t->parent = sMax;
        }

        p_size--;
    }
*/

  const T& minimum( ) { return subtree_minimum( root )->key; }
  const T& maximum( ) { return subtree_maximum( root )->key; }

  bool empty( ) const { return root == nullptr; }
  unsigned long size( ) const { return p_size; }
};

#endif // SPLAY_TREE
```

Analysis

A simple amortized analysis of static splay trees can be carried out using the potential method. Define:

- size(r) = the number of nodes in the sub-tree rooted at node r (including r).
- rank(r) = $\log_2(\text{size}(r))$.

- Φ = the sum of the ranks of all the nodes in the tree.

Φ will tend to be high for poorly balanced trees and low for well-balanced trees.

To apply the potential method, we first calculate ΔΦ: the change in the potential caused by a splay operation. We check each case separately. Denote by rank′ the rank function after the operation. x, p and g are the nodes affected by the rotation operation (see figures above).

Zig step:

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x) && \text{[since only p and x change ranks]} \\
&= \text{rank}'(p) - \text{rank}(x) && \text{[since rank}'(x)=\text{rank}(p)\text{]} \\
&\leq \text{rank}'(x) - \text{rank}(x) && \text{[since rank}'(p)<\text{rank}'(x)\text{]}
\end{aligned}$$

Zig-Zig step:

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x) \\
&= \text{rank}'(g) + \text{rank}'(p) - \text{rank}(p) - \text{rank}(x) && \text{[since rank}'(x)=\text{rank}(g)\text{]} \\
&\leq \text{rank}'(g) + \text{rank}'(x) - 2\,\text{rank}(x) && \text{[since rank}(x)<\text{rank}(p) \text{ and rank}'(x)>\text{rank}'(p)\text{]} \\
&\leq 3(\text{rank}'(x) - \text{rank}(x)) - 2 && \text{[due to the concavity of the log function]}
\end{aligned}$$

Zig-Zag step:

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x) \\
&\leq \text{rank}'(g) + \text{rank}'(p) - 2\,\text{rank}(x) && \text{[since rank}'(x)=\text{rank}(g) \text{ and rank}(x)<\text{rank}(p)\text{]} \\
&\leq 2(\text{rank}'(x) - \text{rank}(x)) - 2 && \text{[due to the concavity of the log function]}
\end{aligned}$$

The amortized cost of any operation is ΔΦ plus the actual cost. The actual cost of any zig-zig or zig-zag operation is 2 since there are two rotations to make. Hence:

$$\begin{aligned}
\text{amortized-cost} &= \text{cost} + \Delta\Phi \\
&\leq 3(\text{rank}'(x) - \text{rank}(x))
\end{aligned}$$

When summed over the entire splay operation, this telescopes to 3(rank(root)−rank(x)) which is O(log n). The Zig operation adds an amortized cost of 1, but there's at most one such operation.

So now we know that the total *amortized* time for a sequence of m operations is:

$$T_{\text{amortized}}(m) = O(m \log n)$$

To go from the amortized time to the actual time, we must add the decrease in potential from the initial state before any operation is done (Φ_i) to the final state after all operations are completed (Φ_f).

$$\Phi_i - \Phi_f = \sum_x \text{rank}_i(x) - \text{rank}_f(x) = O(n \log n)$$

where the last inequality comes from the fact that for every node x, the minimum rank is 0 and the maximum rank is log(n).

Now we can finally bound the actual time:

$$T_{\text{actual}}(m) = O(m \log n + n \log n)$$

Weighted analysis

The above analysis can be generalized in the following way.

- Assign to each node *r* a weight *w*(*r*).
- Define size(*r*) = the sum of weights of nodes in the sub-tree rooted at node *r* (including *r*).
- Define rank(*r*) and Φ exactly as above.

The same analysis applies and the amortized cost of a splaying operation is again:

$$\mathbf{rank}(root) - \mathbf{rank}(x) = O(\log W - \log w(x)) = O\left(\log \frac{W}{w(x)}\right)$$

where *W* is the sum of all weights.

The decrease from the initial to the final potential is bounded by:

$$\Phi_i - \Phi_f \leq \sum_{x \in tree} \log \frac{W}{w(x)}$$

since the maximum size of any single node is *W* and the minimum is *w(x)*.

Hence the actual time is bounded by:

$$O\left(\sum_{x \in sequence} \log \frac{W}{w(x)} + \sum_{x \in tree} \log \frac{W}{w(x)}\right)$$

Performance theorems

There are several theorems and conjectures regarding the worst-case runtime for performing a sequence *S* of *m* accesses in a splay tree containing *n* elements.

Balance Theorem — The cost of performing the sequence *S* is $O\left[m \log n + n \log n\right]$.

> **Proof**
>
> Take a constant weight, e.g. $w(x) = 1$ for every node *x*. Then $W = n$.

This theorem implies that splay trees perform as well as static balanced binary search trees on sequences of at least *n* accesses.[1]

Static Optimality Theorem — Let q_x be the number of times element *x* is accessed in *S*. If every element is accessed at least once, then the cost of performing *S* is $O\left[m + \sum_{x \in tree} q_x \log \frac{m}{q_x}\right]$

> **Proof**
>
> Let $w(x) = q_x$. Then $W = m$.

This theorem implies that splay trees perform as well as an optimum static binary search tree on sequences of at least *n* accesses. They spend less time on the more frequent items.[1]

Static Finger Theorem — Assume that the items are numbered from 1 through n in ascending order. Let f be any fixed element (the 'finger'). Then the cost of performing S is $O\left[m + n\log n + \sum_{x \in sequence} \log(|x - f| + 1)\right]$.

Proof

Let $w(x) = 1/(|x - f| + 1)^2$. Then $W = O(1)$. The net potential drop is $O(n \log n)$ since the weight of any item is at least $1/n^2$.[1]

Dynamic Finger Theorem — Assume that the 'finger' for each step accessing an element y is the element accessed in the previous step, x. The cost of performing S is $O\left[m + n + \sum_{x,y \in sequence}^{m} \log(|y - x| + 1)\right]$.[6][7]

Working Set Theorem — At any time during the sequence, let $t(x)$ be the number of distinct elements accessed before the previous time element x was accessed. The cost of performing S is $O\left[m + n\log n + \sum_{x \in sequence} \log(t(x) + 1)\right]$

Proof

Let $w(x) = 1/(t(x) + 1)^2$. Note that here the weights change during the sequence. However, the sequence of weights is still a permutation of $1, \frac{1}{4}, \frac{1}{9}, \cdots, \frac{1}{n^2}$. So as before $W = O(1)$. The net potential drop is $O(n \log n)$.

This theorem is equivalent to splay trees having key-independent optimality.[1]

Scanning Theorem — Also known as the **Sequential Access Theorem** or the **Queue theorem**. Accessing the n elements of a splay tree in symmetric order takes $O(n)$ time, regardless of the initial structure of the splay tree.[8] The tightest upper bound proven so far is $4.5n$.[9]

Dynamic optimality conjecture

In addition to the proven performance guarantees for splay trees there is an unproven conjecture of great interest from the original Sleator and Tarjan paper. This conjecture is known as the *dynamic optimality conjecture* and it basically claims that splay trees perform as well as any other binary search tree algorithm up to a constant factor.

Unsolved problem in computer science:

Do splay trees perform as well as any other binary search tree algorithm?

(more unsolved problems in computer science)

Dynamic Optimality Conjecture:[1] Let A be any binary search tree algorithm that accesses an element x by traversing the path from the root to x at a cost of $d(x) + 1$,

and that between accesses can make any rotations in the tree at a cost of 1 per rotation. Let $A(S)$ be the cost for A to perform the sequence S of accesses. Then the cost for a splay tree to perform the same accesses is $O[n + A(S)]$.

There are several corollaries of the dynamic optimality conjecture that remain unproven:

Traversal Conjecture:[1] Let T_1 and T_2 be two splay trees containing the same elements. Let S be the sequence obtained by visiting the elements in T_2 in preorder (i.e., depth first search order). The total cost of performing the sequence S of accesses on T_1 is $O(n)$.

Deque Conjecture:[8][10][11] Let S be a sequence of m double-ended queue operations (push, pop, inject, eject). Then the cost of performing S on a splay tree is $O(m + n)$.

Split Conjecture:[5] Let S be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order S is $O(n)$.

Variants

In order to reduce the number of restructuring operations, it is possible to replace the splaying with *semi-splaying*, in which an element is splayed only halfway towards the root.[1][12]

Another way to reduce restructuring is to do full splaying, but only in some of the access operations - only when the access path is longer than a threshold, or only in the first *m* access operations.[1]

See also

- Finger tree
- Link/cut tree
- Scapegoat tree
- Zipper (data structure)
- Trees
- Tree rotation
- AVL tree
- B-tree
- T-tree
- List of data structures
- Iacono's working set structure
- Geometry of binary search trees
- Splaysort, a sorting algorithm using splay trees

Notes

1. Sleator & Tarjan 1985.
2. Goodrich, Tamassia & Goldwasser 2014.
3. Albers & Karpinski 2002.
4. Allen & Munro 1978.
5. Lucas 1991.
6. Cole et al. 2000.
7. Cole 2000.
8. Tarjan 1985.
9. Elmasry 2004.
10. Pettie 2008.

11. Sundar 1992.
12. Brinkmann, Degraer & De Loof 2009.

References

- Albers, Susanne; Karpinski, Marek (28 February 2002). "Randomized Splay Trees: Theoretical and Experimental Results" (http://www14.in.tum.de/personen/albers/papers/ipl02.pdf) (PDF). *Information Processing Letters*. **81** (4): 213–221. doi:10.1016/s0020-0190(01)00230-7 (https://doi.org/10.1016%2Fs0020-0190%2801%2900230-7).
- Allen, Brian; Munro, Ian (October 1978). "Self-organizing search trees". *Journal of the ACM*. **25** (4): 526–535. doi:10.1145/322092.322094 (https://doi.org/10.1145%2F322092.322094).
- Brinkmann, Gunnar; Degraer, Jan; De Loof, Karel (January 2009). "Rehabilitation of an unloved child: semi-splaying" (http://caagt.ugent.be/preprints/splay_spe.pdf) (PDF). *Software—Practice and Experience*. **39** (1): 33–45. CiteSeerX 10.1.1.84.790 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.790). doi:10.1002/spe.v39:1 (https://doi.org/10.1002%2Fspe.v39%3A1). "The results show that semi-splaying, which was introduced in the same paper as splaying, performs better than splaying under almost all possible conditions. This makes semi-splaying a good alternative for all applications where normally splaying would be applied. The reason why splaying became so prominent while semi-splaying is relatively unknown and much less studied is hard to understand."
- Cole, Richard; Mishra, Bud; Schmidt, Jeanette; Siegel, Alan (January 2000). "On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences". *SIAM Journal on Computing*. **30** (1): 1–43. CiteSeerX 10.1.1.36.4558 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.4558). doi:10.1137/s0097539797326988 (https://doi.org/10.1137%2Fs0097539797326988).
- Cole, Richard (January 2000). "On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof". *SIAM Journal on Computing*. **30** (1): 44–85. CiteSeerX 10.1.1.36.2713 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.2713). doi:10.1137/S009753979732699X (https://doi.org/10.1137%2FS009753979732699X).
- Elmasry, Amr (April 2004), "On the sequential access theorem and Deque conjecture for splay trees" (https://www.researchgate.net/profile/Amr_Elmasry2/publication/220150614_On_the_sequential_access_theorem_and_deque_conjecture_for_splay_trees/links/0c96052a84e4fe1eba000000.pdf?disableCoverPage=true) (PDF), *Theoretical Computer Science*, **314** (3): 459–466, doi:10.1016/j.tcs.2004.01.019 (https://doi.org/10.1016%2Fj.tcs.2004.01.019)
- Goodrich, Michael; Tamassia, Roberto; Goldwasser, Michael (2014). *Data Structures and Algorithms in Java* (6 ed.). Wiley. p. 506. ISBN 978-1-118-77133-4.
- Knuth, Donald (1997). *The Art of Computer Programming. 3: Sorting and Searching (3rd ed.). Addison-Wesley. p. 478. ISBN 0-201-89685-0.*
- Lucas, Joan M. (1991). "On the Competitiveness of Splay Trees: Relations to the Union-Find Problem". *On-line Algorithms: Proceedings of a DIMACS Workshop, February 11–13, 1991*. Series in Discrete Mathematics and Theoretical Computer Science. **7**. Center for Discrete Mathematics and Theoretical Computer Science. pp. 95–124. ISBN 0-8218-7111-0.
- Pettie, Seth (2008), "Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture" (http://web.eecs.umich.edu/~pettie/papers/Deque.pdf) (PDF), *Proc. 19th ACM-SIAM Symposium on Discrete Algorithms*, **0707**: 1115–1124, arXiv:0707.2160 (https://arxiv.org/abs/0707.2160), Bibcode:2007arXiv0707.2160P (http://adsabs.harvard.edu/abs/2007arXiv0707.2160P)
- Sleator, Daniel D.; Tarjan, Robert E. (1985). "Self-Adjusting Binary Search Trees" (http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf) (PDF). *Journal of the ACM*. **32** (3): 652–686. doi:10.1145/3828.3835 (https://doi.org/10.1145%2F3828.3835).
- Sundar, Rajamani (1992). "On the Deque conjecture for the splay algorithm". *Combinatorica*. **12** (1): 95–124. doi:10.1007/BF01191208 (https://doi.org/10.1007%2FBF01191208).
- Tarjan, Robert E. (1985). "Sequential access in splay trees takes linear time". *Combinatorica*. **5** (4): 367–378. doi:10.1007/BF02579253 (https://doi.org/10.1007%2FBF02579253).

External links

- NIST's Dictionary of Algorithms and Data Structures: Splay Tree (https://xlinux.nist.gov/dads/HTML/splaytree.html)
- Implementations in C and Java (by Daniel Sleator) (http://www.link.cs.cmu.edu/link/ftp-site/splaying/)

- Pointers to splay tree visualizations (http://wiki.algoviz.org/search/node/splay)
- Fast and efficient implementation of Splay trees (https://github.com/fbuihuu/libtree)
- Top-Down Splay Tree Java implementation (https://github.com/cpdomina/SplayTree)
- Zipper Trees (https://arxiv.org/abs/1003.0139)
- splay tree video (https://www.youtube.com/watch?v=G5QIXywcJlY)
