Splay tree

From Wikipedia, the free encyclopedia

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 nonrandom 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 Dominic Sleator and Robert Endre Tarjan in 1985.[1]

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

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 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.

Contents [hide]

1 Advantages
2 Disadvantages
3 Operations
3.1 Splaying
3.2 Insertion
3.3 Deletion
4 Code in C language
4.1 Splay operation in BST
5 Analysis
6 Performance theorems
7 Dynamic optimality conjecture
8 See also
9 References
10 External links

Advantages

[edit]

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 nearly all practical applications (also see Locality of reference),[*citation needed*] and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Simple implementation—simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Comparable performance—average-case performance is as efficient as other trees.[*citation needed*]
- Small memory footprint—splay trees do not need to store any bookkeeping data.
- Possibility of creating a persistent data structure version of splay trees—which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized O(log *n*) space per update.
- Working well with nodes containing identical keys—contrary to other types of self-balancing trees. Even with identical keys, performance remains amortized O(log *n*). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation

can return the leftmost or rightmost node of a given key.

Disadvantages [edit]

Perhaps 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 slow. 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[2].

A splay tree can be worse than a static tree by at most a constant factor.

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.

Operations [edit]

Splaying [edit]

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.

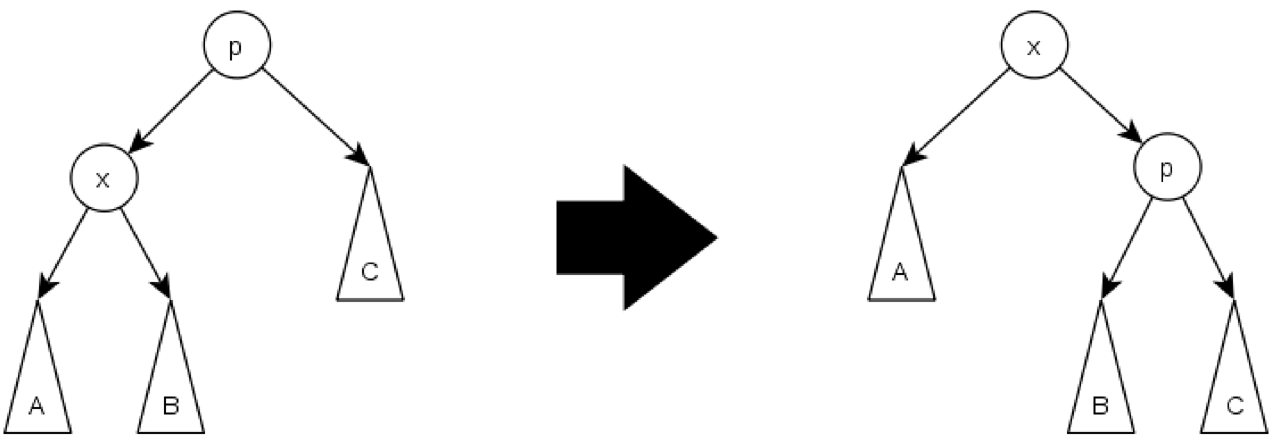
Each particular step depends on three factors:

- 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).

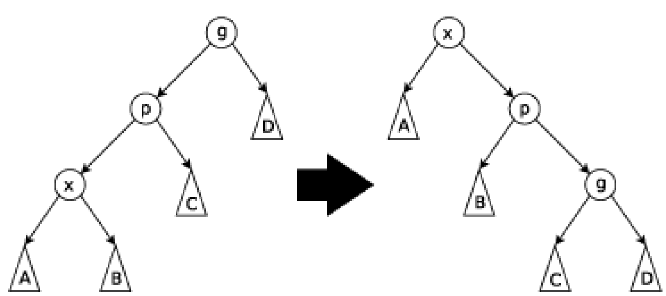
It is important to remember to set *gg* (the *great-grandparent* of x) to now point to x after any splay operation. If *gg* is null, then x obviously is now the root and must be updated as such.

The three types of splay steps are:

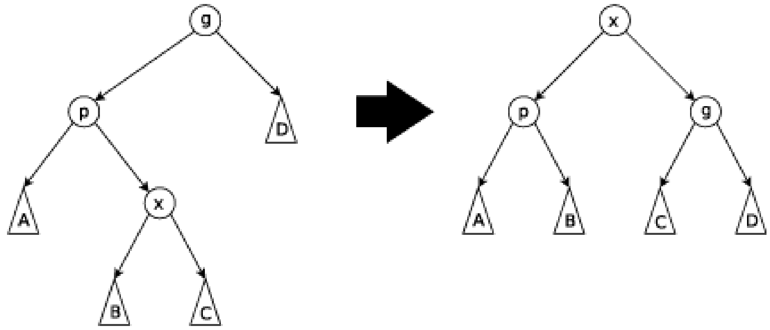
Zig Step: This step is done when *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when *x* has odd depth at the beginning of the operation.



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[3] 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 *x* and *p*, then rotated on the edge between *x* and its new parent *g*.



Insertion

[edit]

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

1. First insert the node as with a normal binary search tree.
2. Then splay the newly inserted node *x* to the top of the tree.

Deletion

[edit]

To delete a node *x*, we use the same method as with a binary search tree: if *x* has two children, we 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 we 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. OR The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. This leaves the tree with two sub trees. The maximum element of the left sub tree (: METHOD 1), or minimum of the right sub tree (: METHOD 2) is then splayed to the root. The right sub tree is made the right child of the resultant left sub tree (for METHOD 1). The root of left sub tree is the root of melded tree.

Code in C language

[edit]

Splay operation in BST

[edit]

Here x is the node on which the splay operation is performed and root is the root node of the tree.

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *parent;
        struct node *left;
        struct node *right;
};
int data_print(struct node *x);
struct node *rightrotation(struct node *p,struct node *root);
struct node *leftrotation(struct node *p,struct node *root);
void splay (struct node *x, struct node *root);
struct node *insert(struct node *p,int value);
struct node *inorder(struct node *p);
struct node *delete(struct node *p,int value);
struct node *successor(struct node *x);
struct node *lookup(struct node *p,int value);

void splay (struct node *x, struct node *root)
{
        struct node *p,*g;
        /*check if node x is the root node*/
        if(x==root)
                return;
        /*Performs Zig step*/
        else if(x->parent==root)
        {
                if(x==x->parent->left)
                        root=rightrotation(root,root);
                else
                        root=leftrotation(root,root);
        }
        else
        {
                p=x->parent; /*now points to parent of x*/
                g=p->parent; /*now points to parent of x's parent*/
                /*Performs the Zig-zig step when x is left and x's parent is left*/
                if(x==p->left&&p==g->left)
                {
```

```
				root=rightrotation(g,root);
				root=rightrotation(p,root);
			}
			/*Performs the Zig-zig step when x is right and x's parent is right*/
			else if(x==p->right&&p==g->right)
			{
				root=leftrotation(g,root);
				root=leftrotation(p,root);
			}
			/*Performs the Zig-zag step when x's is right and x's parent is left*/
			else if(x==p->right&&p==g->left)
			{
				root=leftrotation(p,root);
				root=rightrotation(g,root);
			}
			/*Performs the Zig-zag step when x's is left and x's parent is right*/
			else if(x==p->left&&p==g->right)
			{
				root=rightrotation(p,root);
				root=leftrotation(g,root);
			}
			splay(x, root);
		}
}
struct node *rightrotation(struct node *p,struct node *root)
{
		struct node *x;
		x = p->left;
		p->left = x->right;
		if (x->right!=NULL) x->right->parent = p;
		x->right = p;
		if (p->parent!=NULL)
			if(p==p->parent->right) p->parent->right=x;
			else
				 p->parent->left=x;
		x->parent = p->parent;
		p->parent = x;
		if (p==root)
			return x;
		else
			return root;
}
struct node *leftrotation(struct node *p,struct node *root)
{
		struct node *x;
		x = p->right;
		p->right = x->left;
		if (x->left!=NULL) x->left->parent = p;
		x->left = p;
		if (p->parent!=NULL)
			if (p==p->parent->left) p->parent->left=x;
			else
				 p->parent->right=x;
		x->parent = p->parent;
		p->parent = x;
		if(p==root)
			return x;
		else
			return root;
}
struct node *insert(struct node *p,int value)
{
		struct node *temp1,*temp2,*par,*x;
		if(p == NULL)
		{
			p=(struct node *)malloc(sizeof(struct node));
			if(p != NULL)
			{
				p->data = value;
				p->parent = NULL;
				p->left = NULL;
				p->right = NULL;
			}
			else
			{
				printf("No memory is allocated\n");
				exit(0);
			}
			return(p);
		} //the case 2 says that we must splay newly inserted node to root
		else
		{
				temp2 = p;
				while(temp2 != NULL)
				{
					temp1 = temp2;
					if(temp2->data > value)
						temp2 = temp2->left;
					else if(temp2->data < value)
						temp2 = temp2->right;
					else
```

```
                                        if(temp2->data == value)
                                                return temp2;
                                }
                                if(temp1->data > value)
                                {
                                        par = temp1;//temp1 having the parent address,so that's it
                                        temp1->left = (struct node *)malloc(sizeof(struct node));
                                        temp1= temp1->left;
                                        if(temp1 != NULL)
                                        {
                                                temp1->data = value;
                                                temp1->parent = par;//store the parent address.
                                                temp1->left = NULL;
                                                temp1->right = NULL;
                                        }
                                        else
                                        {
                                                printf("No memory is allocated\n");
                                                exit(0);
                                        }
                                }
                                else
                                {
                                        par = temp1;//temp1 having the parent node address.
                                        temp1->right = (struct node *)malloc(sizeof(struct node));
                                        temp1 = temp1->right;
                                        if(temp1 != NULL)
                                        {
                                                temp1->data = value;
                                                temp1->parent = par;//store the parent address
                                                temp1->left = NULL;
                                                temp1->right = NULL;
                                        }
                                        else
                                        {
                                                printf("No memory is allocated\n");
                                                exit(0);
                                        }
                                }
        }
        splay(temp1,p);//temp1 will be new root after splaying
        return (temp1);
}
struct node *inorder(struct node *p)
{
        if(p != NULL)
        {
                inorder(p->left);
                printf("CURRENT %d\t",p->data);
                printf("LEFT %d\t",data_print(p->left));
                printf("PARENT %d\t",data_print(p->parent));
                printf("RIGHT %d\t\n",data_print(p->right));
                inorder(p->right);
        }
}
struct node *delete(struct node *p,int value)
{
        struct node *x,*y,*p1;
        struct node *root;
        struct node *s;
        root = p;
        x = lookup(p,value);
        if(x->data == value)
        {       //if the deleted element is leaf
                if((x->left == NULL) && (x->right == NULL))
                {
                        y = x->parent;
                        if(x ==(x->parent->right))
                                y->right = NULL;
                        else
                                y->left = NULL;
                        free(x);
                }
                //if deleted element having left child only
                else if((x->left != NULL) &&(x->right == NULL))
                {
                        if(x == (x->parent->left))
                        {
                                y = x->parent;
                                x->left->parent = y;
                                y->left = x->left;
                                free(x);
                        }
                        else
                        {
                                y = x->parent;
                                x->left->parent = y;
                                y->right = x->left;
                                free(x);
                        }
                }
```

```
                        //if deleted element having right child only
                        else if((x->left == NULL) && (x->right != NULL))
                        {
                                if(x == (x->parent->left))
                                {
                                        y = x->parent;
                                        x->right->parent = y;
                                        y->left = x->right;
                                        free(x);
                                }
                                else
                                {
                                        y = x->parent;
                                        x->right->parent = y;
                                        y->right = x->right;
                                        free(x);
                                }
                        }
                        //if the deleted element having two children
                        else if((x->left != NULL) && (x->right != NULL))
                        {
                                if(x == (x->parent->left))
                                {
                                        s = successor(x);
                                        if(s != x->right)
                                        {
                                                y = s->parent;
                                                if(s->right != NULL)
                                                {
                                                        s->right->parent = y;
                                                        y->left = s->right;
                                                }
                                                else y->left = NULL;
                                                s->parent = x->parent;
                                                x->right->parent = s;
                                                x->left->parent = s;
                                                s->right = x->right;
                                                s->left = x->left;
                                                x->parent->left = s;
                                        }
                                        else
                                        {
                                                y = s;
                                                s->parent = x->parent;
                                                x->left->parent = s;
                                                s->left = x->left;
                                                x->parent->left = s;
                                        }
                                        free(x);
                                }
                                else if(x == (x->parent->right))
                                {
                                        s = successor(x);
                                        if(s != x->right)
                                        {
                                                y = s->parent;
                                                if(s->right != NULL)
                                                {
                                                        s->right->parent = y;
                                                        y->left = s->right;
                                                }
                                                else y->left = NULL;
                                                s->parent = x->parent;
                                                x->right->parent = s;
                                                x->left->parent = s;
                                                s->right = x->right;
                                                s->left = x->left;
                                                x->parent->right = s;
                                        }
                                        else
                                        {
                                                y = s;
                                                s->parent = x->parent;
                                                x->left->parent = s;
                                                s->left = x->left;
                                                x->parent->right = s;
                                        }
                                        free(x);
                                }

                        }
                        splay(y,root);
                }
                else
                {
                        splay(x,root);
                }
}
struct node *successor(struct node *x)
{
```

```
        struct node *temp,*temp2;
        temp=temp2=x->right;
        while(temp != NULL)
        {
                temp2 = temp;
                temp = temp->left;
        }
        return temp2;
}
//p is a root element of the tree
struct node *lookup(struct node *p,int value)
{
        struct node *temp1,*temp2;
        if(p != NULL)
        {
                temp1 = p;
                while(temp1 != NULL)
                {
                        temp2 = temp1;
                        if(temp1->data > value)
                                temp1 = temp1->left;
                        else if(temp1->data < value)
                                temp1 = temp1->right;
                        else
                                        return temp1;
                }
                return temp2;
        }
        else
        {
                printf("NO element in the tree\n");
                exit(0);
        }
}
struct node *search(struct node *p,int value)
{
        struct node *x,*root;
        root = p;
        x = lookup(p,value);
        if(x->data == value)
        {
                printf("Inside search if\n");
                splay(x,root);
        }
        else
        {
                printf("Inside search else\n");
                splay(x,root);
        }
}
main()
{
        struct node *root;//the root element
        struct node *x;//x is which element will come to root.
        int i;
        root = NULL;
        int choice = 0;
        int ele;
        while(1)
        {
                printf("\n\n 1.Insert");
                printf("\n\n 2.Delete");
                printf("\n\n 3.Search");
                printf("\n\n 4.Display\n");
                printf("\n\n Enter your choice:");
                scanf("%d",&choice);
                if(choice==5)
                        exit(0);
                switch(choice)
                {
                        case 1:
                                printf("\n\n Enter the element to be inserted:");
                                scanf("%d",&ele);
                                x = insert(root,ele);
                                if(root != NULL)
                                {
                                        splay(x,root);
                                }
                                root = x;
                                break;
                        case 2:
                                if(root == NULL)
                                {
                                        printf("\n Empty tree...");
                                        continue;
                                }
                                printf("\n\n Enter the element to be delete:");
                                scanf("%d",&ele);
                                root = delete(root,ele);
                                break;
                        case 3:
```

```
                                        printf("Enter the element to be search\n");
                                        scanf("%d",&ele);
                                        x = lookup(root,ele);
                                                splay(x,root);
                                        root = x;
                                        break;
                                case 4:
                                        printf("The elements are\n");
                                        inorder(root);
                                        break;
                                default:
                                        printf("Wrong choice\n");
                                        break;
                        }
                }
}
int data_print(struct node *x)
{
        if ( x==NULL )
                return 0;
        else
                return x->data;
}
```

Analysis

[edit]

A simple amortized analysis of static splay trees can be carried out using the potential method. Suppose that size(*r*) is the number of nodes in the subtree rooted at *r* (including *r*) and rank(*r*) = $\log_2$(size(*r*)). Then the potential function P(*t*) for a splay tree *t* is the sum of the ranks of all the nodes in the tree. This will tend to be high for poorly balanced trees, and low for well-balanced trees. We can bound the amortized cost of any zig-zig or zig-zag operation by:

amortized cost = cost + $P(t_f) - P(t_i) \le 3(\mathrm{rank}_f(x) - \mathrm{rank}_i(x))$,

where *x* is the node being moved towards the root, and the subscripts "f" and "i" indicate after and before the operation, respectively. When summed over the entire splay operation, this telescopes to 3(rank(root)) which is O(log *n*). Since there's at most one zig operation, this only adds a constant.

Performance theorems

[edit]

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[1]

The cost of performing the sequence *S* is $O\left(m(1+\log n)+n\log n\right)$. In other words, splay trees perform as well as static balanced binary search trees on sequences of at least *n* accesses.

Static Optimality Theorem[1]

Let q_i be the number of times element *i* is accessed in *S*. The cost of performing *S* is $O\left(m+\sum_{i=1}^n q_i\log\frac{m}{q_i}\right)$. In other words, splay trees perform as well as optimum static binary search trees on sequences of at least *n* accesses.

Static Finger Theorem[1]

Let i_j be the element accessed in the j^{th} access of *S* and let *f* be any fixed element (the finger). The cost of performing *S* is $O\left(m+n\log n+\sum_{j=1}^m \log(|i_j-f|+1)\right)$.

Working Set Theorem[1]

Let $t(j)$ be the number of distinct elements accessed between access *j* and the previous time element i_j was accessed. The cost of performing *S* is $O\left(m+n\log n+\sum_{j=1}^m \log(t(j)+1)\right)$.

Dynamic Finger Theorem[4][5]

The cost of performing *S* is $O\left(m+n+\sum_{j=1}^m \log(|i_{j+1}-i_j|+1)\right)$.

Scanning Theorem[6]

Also known as the Sequential Access 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. The tightest upper bound proven so far is $4.5n$.[7]

Dynamic optimality conjecture

[edit]

Unsolved problems in computer science

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

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.

> 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][6][9] 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:[10] Let S be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order S is $O(n)$.

See also

[edit]

- Knuth, Donald. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Page 478 of section 6.2.3.
- Finger tree
- Link/cut tree
- Scapegoat tree
- Zipper (data structure)
- Trees
- Tree rotation
- AVL tree
- B-tree
- T-tree
- List of data structures

References

[edit]

1. ^ *a b c d e f g* Sleator, Daniel D.; Tarjan, Robert E. (1985), "Self-Adjusting Binary Search Trees", *Journal of the ACM (Association for Computing Machinery)* 32 (3): 652–686, DOI:10.1145/3828.3835
2. ^ "Randomized Splay Trees: Theoretical and Experimental Results". Retrieved 31 May 2011.
3. ^ Allen, Brian; and Munro, Ian (1978), "Self-organizing search trees", *Journal of the ACM* 25 (4): 526–535, DOI:10.1145/322092.322094
4. ^ Cole, Richard; Mishra, Bud; Schmidt, Jeanette; and Siegel, Alan (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences", *SIAM (Society for Industrial and Applied Mathematics) Journal on Computing* 30: 1–43
5. ^ Cole, Richard (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof", *SIAM Journal on Computing* 30: 44–85, DOI:10.1137/S009753979732699X
6. ^ *a b* Tarjan, Robert E. (1985), "Sequential access in splay trees takes linear time", *Combinatorica* 5 (4): 367–378, DOI:10.1007/BF02579253
7. ^ Elmasry, Amr (2004), "On the sequential access theorem and Deque conjecture for splay trees", *Theoretical Computer Science* 314 (3): 459–466, DOI:10.1016/j.tcs.2004.01.019
8. ^ Pettie, Seth (2008), "Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture", *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*: 1115–1124

9. ^ Sundar, Rajamani (1992), "On the Deque conjecture for the splay algorithm", *Combinatorica* 12 (1): 95–124, DOI:10.1007/BF01191208
10. ^ Lucas, Joan M. (1991), "On the Competitiveness of Splay Trees: Relations to the Union-Find Problem", *Online Algorithms, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Series in Discrete Mathematics and Theoretical Computer Science Vol. 7*: 95–124

External links [edit]

- NIST's Dictionary of Algorithms and Data Structures: Splay Tree
- Implementations in C and Java (by Daniel Sleator)
- Pointers to splay tree visualizations
- Fast and efficient implentation of Splay trees
- Top-Down Splay Tree Java implementation
- Zipper Trees

v · t · e ·	Trees in computer science [hide]
Binary trees	Binary search tree (BST) · Cartesian tree · Top tree · T-tree ·
Self-balancing binary search trees	AA tree · AVL tree · LLRB tree · Red–black tree · Scapegoat tree · Splay tree · Treap ·
B-trees	B+ tree · B*-tree · B^x-tree · UB-tree · 2-3 tree · 2-3-4 tree · (a,b)-tree · Dancing tree · Htree ·
Tries	Suffix tree · Radix tree · Ternary search tree · X-fast trie · Y-fast trie ·
Binary space partitioning (BSP) trees	Quadtree · Octree · *k*-d tree · Implicit *k*-d tree · vp-tree ·
Non-binary trees	Exponential tree · Fusion tree · Interval tree · PQ tree · Range tree · SPQR tree · Van Emde Boas tree ·
Spatial data partitioning trees	R-tree · R+ tree · R* tree · X-tree · M-tree · Segment tree · Hilbert R-tree · Priority R-tree ·
Other trees	Heap · Hash tree · Finger tree · Metric tree · Cover tree · BK-tree · Doubly chained tree · iDistance · Link-cut tree · Fenwick tree ·

Categories: Binary trees