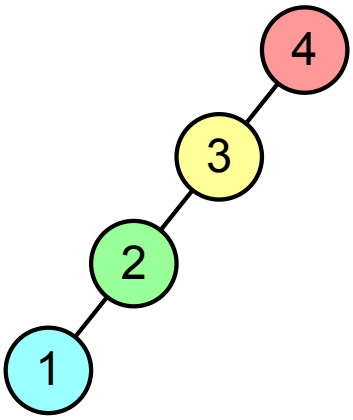
Growing with the Web

Splay tree

Published 9 June 2013, updated 16 August 2015

Computer science Data structure Java JavaScript Tree

The splay tree is a type of self-adjusting binary search tree like the red-black tree. What makes the splay tree special is its ability to access recently accessed elements faster. Whenever an operation is performed, the tree performs an operation called *splaying* which pulls the element to the top of the tree.



The worst case height of a splay tree is n, this could be the case if all nodes were accessed in ascending order for example.

This makes the worst case complexity of the splay tree's operations $O(n)$. Since all operations also splay the tree on the node, the tree ends up roughly balancing itself, this results in a $O(\log n)$ amortised worst case time complexity for all operations.

The splay tree is a particularly good choice as a data structure when it's likely that the same nodes will be accessed multiple times in a short period. This is where the real power in the splay tree lies, in its ability to hoist nodes up to the root when they are accessed, giving speedy access for nearby successive accesses.

This article assumes knowledge of the binary search tree (BST) data structure.

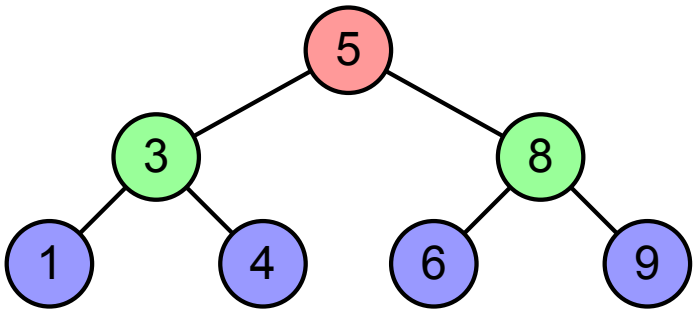
Complexity

Operation	Description	Complexity
Delete	Deletes a node given a key	$O(\log n)^*$
Insert	Inserts a node with an associated key	$O(\log n)^*$
Search	Searches for and returns a node using its key	$O(\log n)^*$
Splay	Reorganises the tree, moving a particular node to the top	$O(\log n)^*$

** Amortised*

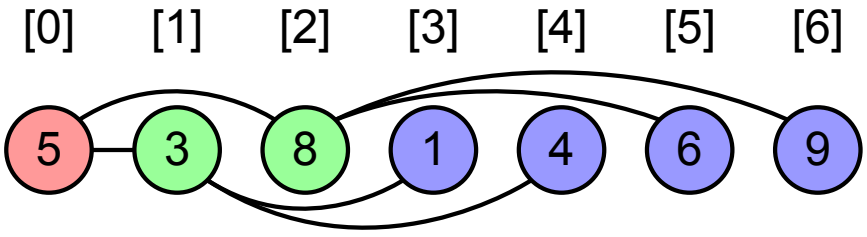
Representation

There are two main ways of representing a binary tree. The first is using node objects that have references to their children.



Tree representation

The second is using a regular array and manipulating the index of the node to find its children. The index of the left child of a node is $2i + 1$ and the index of the right is $2i + 2$ where i is the index of the parent.



Array representation

The index of node's parent can also be retrieved with $\lfloor (i - 1)/2 \rfloor$.

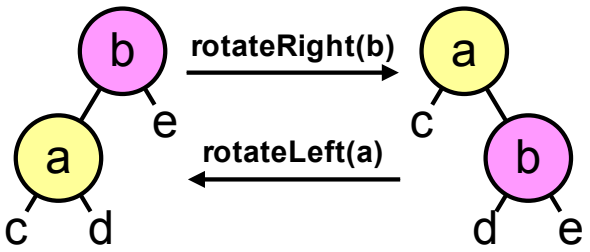
Parallelism

Due to the splay tree adjusting itself even after a “read-only” operation, the splay tree is probably not ideal in a multi-threaded application. If parallelism is desired, additional guards need to be put in place to protect against race conditions.

Operations

Rotation

The generic tree rotation operation is used to perform the below splaying operations. Here is an illustration of the process.



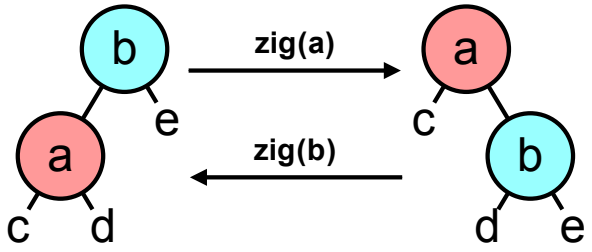
Splay(*a*)

The splay operation is performed to bring the node *a* that is being worked on to the root of the tree. It performs a series of operations called zig, zig-zig and zig-zag depending on the characteristics of *a*.

Each operation has two variants depending on whether *a* or its parent are left or right children.

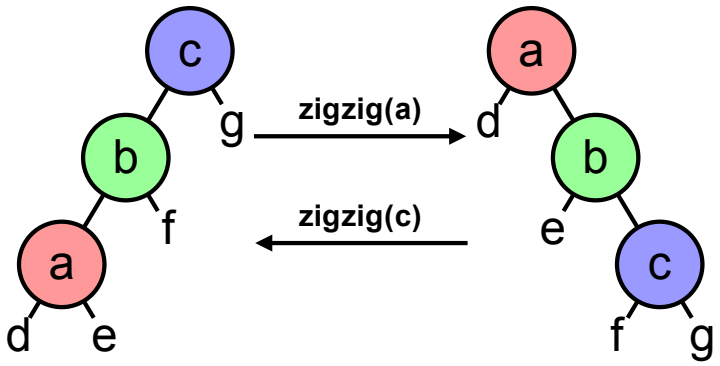
Zig(*a*)

This operation is performed when the parent of *a* is the root of the tree. A left or right rotate is performed to bring *a* to the root of the tree.



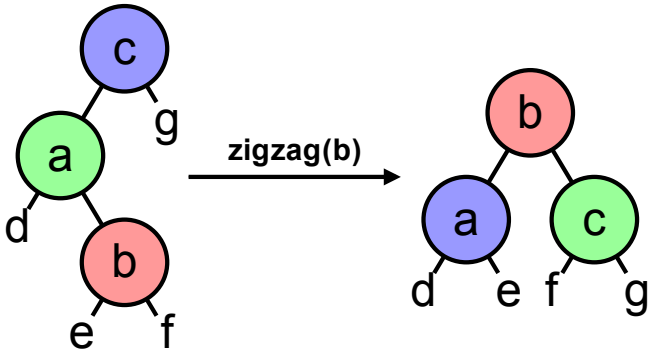
Zig-zig(*a*)

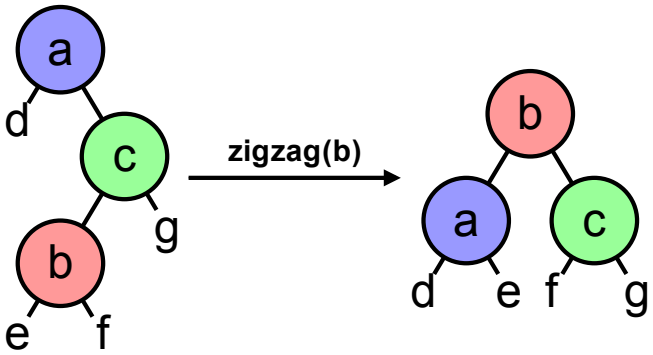
This operation is performed when *a* and its parent are the same child type as each other (both left children or both right children). It performs either two right rotations or two left rotations depending on the side. Its name is derived from the fact that the rotations performed are of the same type.



Zig-zag(*a*)

This operation is performed when *a* is a different child type to its parent. It performs a rotation of both types (left then right, or right then left) depending on the child type of *a*.





Delete(*a*)

Delete can be implemented two different ways:

- by performing a regular BST delete on the node *a* and then splaying the tree on what was *a*'s parent, or
- by splaying the node *a* and then performing a regular BST delete on the *a*.

Insert(*a*)

Insert performs a regular BST insert and then splays the tree on the node *a*, adjusting the tree so that the inserted node is at the root of the tree.

Search(*a*)

Search performs a regular BST search and then splays the tree on the node *a*, adjusting the tree so that the searched node is at the root of the tree.

Which binary search tree is best?

The AVL tree, red-black tree and splay tree are all self-adjusting binary search trees, so which one is better in which situation? And when is it better to use a regular BST?

A paper by Ben Pfaff of Stanford University performs an in-depth study of the performance characteristics of each tree under various circumstances. Each data structures excels based on runtime patterns in the input and calling of operations. It comes to the following conclusions:

- *Regular BSTs* excel when randomly ordered input can be relied

upon.

- *Splay trees* excel when data is often inserted in a sorted order and later accesses are sequential or clustered.
- *AVL trees* excel when data is often inserted in a sorted order and later accesses are random.
- *Red-black trees* excel when data is often inserted in random order but occasional runs of sorted order are expected.

Code

Java JavaScript

```
public class SplayTree<T extends Comparable<T>> {

    private SplayTreeNode<T> root;

    public SplayTree() { }

    private void splay(SplayTreeNode<T> node) {
        while (node.parentExists()) {
            SplayTreeNode parent = node.getParent();
            if (!parent.parentExists()) {
                if (parent.getLeft() == node) {
                    rotateRight(parent);
                } else {
                    rotateLeft(parent);
                }
            } else {
                SplayTreeNode gparent = parent.getParent();
                if (parent.getLeft() == node && gparent.getLeft() ==
parent) {
                    rotateRight(gparent);
                    rotateRight(node.getParent());
                } else if (parent.getRight() == node &&
                        gparent.getRight() == parent) {
                    rotateLeft(gparent);
```

```
                rotateLeft(node.getParent());
            } else if (parent.getLeft() == node &&
                    gparent.getRight() == parent) {
                rotateRight(parent);
                rotateLeft(node.getParent());
            } else {
                rotateLeft(parent);
                rotateRight(node.getParent());
            }
        }
    }
}

private void rotateLeft(SplayTreeNode<T> x) {
    SplayTreeNode y = x.getRight();
    x.setRight(y.getLeft());
    if (y.getLeft() != null) {
        y.getLeft().setParent(x);
    }
    y.setParent(x.getParent());
    if (x.getParent() == null) {
        root = y;
    } else {
        if (x == x.getParent().getLeft()) {
            x.getParent().setLeft(y);
        } else {
            x.getParent().setRight(y);
        }
    }
    y.setLeft(x);
    x.setParent(y);
}

private void rotateRight(SplayTreeNode<T> x) {
    SplayTreeNode y = x.getLeft();
    x.setLeft(y.getRight());
    if (y.getRight() != null) {
        y.getRight().setParent(x);
    }
    y.setParent(x.getParent());
    if (x.getParent() == null) {
        root = y;
    } else {
        if (x == x.getParent().getLeft()) {
            x.getParent().setLeft(y);
        } else {
            x.getParent().setRight(y);
        }
    }
    y.setRight(x);
    x.setParent(y);
}
```

```
    public void insert(T key) {
        if (root == null) {
            root = new SplayTreeNode(key, null);
            return;
        }

        insert(key, root);
        search(key);
    }

    private void insert(T key, SplayTreeNode<T> node) {
        if (key.compareTo( node.getKey() ) < 0) {
            if (node.leftExists()) {
                insert(key, node.getLeft());
            } else {
                node.setLeft(new SplayTreeNode(key, node));
            }
        }

        if (key.compareTo(node.getKey())>0) {
            if (node.rightExists()) {
                insert(key, node.getRight());
            } else {
                node.setRight(new SplayTreeNode(key, node));
            }
        }
    }

    public void delete(T key) {
        if (root == null) {
            return;
        }

        search(key);
        delete(key, root);
    }

    private void delete(T key, SplayTreeNode<T> node) {
        if (key.compareTo(node.getKey())< 0) {
            if (node.leftExists()) {
                delete(key, node.getLeft());
            }
            if (node.getLeft().isDeleted()) {
                node.setLeft(null);
            }
            return;
        }

        if (key.compareTo(node.getKey()) > 0) {
            if (node.rightExists()) {
                delete(key, node.getRight());
            }
            if (node.getRight().isDeleted()) {
```

```
                node.setRight(null);
            }
            return;
        }

        delete(node);
    }

    private void delete(SplayTreeNode<T> node) {
        if (!(node.leftExists() || node.rightExists())) {
            node.setDeleted(true);
            return;
        }

        if (node.leftExists() && !node.rightExists()) {
            node.setKey(node.getLeft().getKey());
            if (node.getLeft().rightExists()) {
                node.setRight(node.getLeft().getRight());
            }
            if (node.getLeft().leftExists()) {
                node.setLeft(node.getLeft().getLeft());
            } else {
                node.setLeft(null);
            }
            return;
        }

        if (node.rightExists() && !node.leftExists()) {
            node.setKey(node.getRight().getKey());
            if (node.getRight().leftExists()) {
                node.setLeft(node.getLeft().getLeft());
            }
            if (node.getRight().rightExists()) {
                node.setRight(node.getLeft().getRight());
            } else {
                node.setRight(null);
            }
            return;
        }

        // both exist, replace with minimum from right sub-tree
        T min = findMin(node.getRight());
        node.setKey(min);
    }

    private T findMin(SplayTreeNode<T> node) {
        if (!node.leftExists()) {
            node.setDeleted(true);
            return node.getKey();
        }

        T min = findMin(node.getLeft());
        if (node.getLeft().isDeleted()) {
```

```
            node.setLeft(null);
        }
        return min;
    }

    public boolean search(T key) {
        if (root == null) {
            return false;
        }

        SplayTreeNode<T> node = search(key, root);
        splay(node);
        return node != null;
    }

    private SplayTreeNode<T> search(T key, SplayTreeNode<T> node) {
        if (key == node.getKey()) {
            return node;
        }

        if (key.compareTo(node.getKey()) < 0) {
            if (!node.leftExists()) {
                return null;
            }
            return search(key, node.getLeft());
        }

        if (key.compareTo(node.getKey()) > 0) {
            if (!node.rightExists()) {
                return null;
            }
            return search(key, node.getRight());
        }

        return null;
    }

    public String toString() {
        return root.toString();
    }

}



public class SplayTreeNode<T extends Comparable<T>> {

    private final String nullNodeString = "_";
    private SplayTreeNode<T> left;
    private SplayTreeNode<T> right;
    private SplayTreeNode<T> parent;

    private T key;
```

```
    private boolean isDeleted = false;

    public SplayTreeNode(T key, SplayTreeNode<T> parent) {
        this.key = key;
        this.parent = parent;
    }

    @Override
    public String toString() {
        return key + " : { " +
                (leftExists() ? left.toString() : nullNodeString) + " , "
+
                (rightExists() ? right.toString() : nullNodeString) + "
}";
    }

    public boolean leftExists() {
        return left != null;
    }

    public boolean rightExists() {
        return right != null;
    }

    public boolean parentExists() {
        return parent != null;
    }

    public T getKey() {
        return key;
    }

    public void setKey(T key) {
        this.key = key;
    }

    public SplayTreeNode<T> getLeft() {
        return left;
    }

    public void setLeft(SplayTreeNode<T> left) {
        this.left = left;
    }

    public SplayTreeNode<T> getRight() {
        return right;
    }

    public void setRight(SplayTreeNode<T> right) {
        this.right = right;
    }

    public boolean isDeleted() {
```

```
        return isDeleted;
    }

    public void setDeleted(boolean isDeleted) {
        this.isDeleted = isDeleted;
    }

    public SplayTreeNode<T> getParent() {
        return parent;
    }

    public void setParent(SplayTreeNode<T> parent) {
        this.parent = parent;
    }

}
```

View source on GitHub

More posts tagged Data structure

- AVL tree
- Binary heap
- Binary search tree
- Binomial heap
- Fibonacci heap
- Red-black tree

Comments

Follow me