To read


See also PAL webpage for references
AVL trees and red-black trees are binary search trees with logarithmic height. This ensures all operations are $O(\ln(n))$.

An alternative idea is to make use of an old maxim: **Data that has been recently accessed is more likely to be accessed again in the near future.**

Accessed nodes are splayed (= moved by one or more rotations) to the root of the tree:

**Find:** Find the node like in a BST and then splay it to the root.

**Insert:** Insert the node like in a BST and then splay it to the root.

**Delete:** Splay the node to the root and then delete it like in a BST.

Splay tree

- A binary search tree.

- No additional tree shape description (no additional memory!) is used.

- Each node access or insertion *splays* that node to the root.

- Rotations are *zig*, *zig-zig* and *zig-zag*, based on BST single rotation.

- All operations run times are $O(n)$, as the tree height can be $\Theta(n)$.

- Amortized run times of all operations are $O(\ln(n))$. 
Zig rotation is the same as a rotation (L or R) in AVL tree.

Afected nodes and edges

The terms "Zig" and "Zag" are not chiral, that is, they do not describe the direction (left or right) of the actual rotations.

Note
Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.
Both simple rotations are performed at the top of the current subtree, the splayed node (with key A) is not involved in the first rotation.
Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.
Zig-Zag rotation is identical to the double (LR or RL) rotation in AVL tree.

Note:
Splay Tree - Insert

Example

Insert 1  Insert 2  Splay  Insert 3  Splay  Insert 4  Splay

1 1 2 1 2 1 3 4

etc...

Insert 5  Splay  etc...  Insert 12  Splay

1 2 3 4 5 6 7 8 9 10 11 12

Note the extremely inefficient shape of the resulting tree.
Splay Tree - Find

Find 1

Key 1 is the deepest key in the tree.

Find operation is of $\Theta(n)$ complexity in this case :-(.

Pokročilá Algoritmizace, A4M33PAL, ZS 2012/2013, FEL ČVUT, 12/14
Splay Tree - Insert

Note that the tree height is roughly halved. \( H \rightarrow \frac{H + 3}{2} \)
Splay Tree - Find

Find 3

Key 3 is the deepest key in the tree.

The Find operation would be again of $\sim n$ complexity. :-(

Pokročilá Algoritmizace, A4M33PAL, ZS 2012/2013, FEL ČVUT, 12/14
Scheme - Progress of the two most unfavourable Find operations.

Note the relatively favourable shape of the resulting tree.
**Delete(k)**

1. Find(k);  // This splays k to the root
2. Remove the root;  // Splits the tree into L and R subtree of the root.
3. y = Find max in L subtree;  // This splays y to the root of L subtree
4. y.right = R subtree;

---

1. Find k
2. Split = remove root
3. FindMax(L)
4. y.right = R
It is difficult to demonstrate the amortized logarithmic behaviour of splay trees using only small trees with few nodes.

The original ACM article [2] proves the *balance theorem*: The run time of performing a sequence of $m$ operations on a splay tree with $n$ nodes is $O( m(1 + \ln(n)) + n \ln(n) )$.

Therefore, the run time for a splay tree is comparable to any balanced tree assuming at least $n$ operations.
From the time of introducing splay trees (1985) up till today the following conjecture (among others) remains unproven.

**Dynamic optimality conjecture**[^2]

Consider any sequence of successful accesses on an $n$-node search tree. Let $A$ be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses $A$ performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation.

Then the total time to perform all the accesses by splaying is no more than $O(n)$ plus a constant times the time required by algorithm $A$.  

[^2]: Dynamic optimality conjecture
Advantages:
  – The amortized run times are similar to that of AVL trees and red-black trees
  – The implementation is easier
  – No additional information (height/colour) is required

Disadvantages:
  – The tree will change with read-only operations
A **2-3-4 search tree** is structurally a **B-tree of min degree 2 and max degree 4**.

A node is a **2-node** or a **3-node** or a **4-node**.

If a node is not a leaf it has the corresponding number (2, 3, 4) of children.

All leaves are at the same distance from the root, the tree is **perfectly balanced**.
**2-3-4 tree**

**Find:** As in B-tree

**Insert:** As in B-tree: Find the place for the inserted key $x$ in a leaf and store it there. If necessary, split the leaf and store the median in the parent.

**Splitting strategy**
Additional insert rule (like single phase strategy in B-trees):
In our way down the tree, whenever we reach a 4-node (including a leaf), we split it into two 2-nodes, and move the middle element up to the parent node. This strategy prevents the following from happening:
After inserting a key it might be necessary to split all the nodes going from inserted key back to the root. Such outcome is considered to be time consuming.

Splitting 4-nodes on the way down results in sparse occurrence of 4-nodes in the tree, thus the nodes never have to be split recursively bottom-up.

**Delete:** As in B-tree
2-3-4 tree

Split node is the root. Only the root splitting increases the tree height.

Split node is the leftmost or the rightmost child of either a 2-node or a 3-node. (Only the leftmost case is shown, the rightmost case is analogous.)

Split node is the middle child of a 3-node.

The node being split cannot be a child of a 4-node, due to the splitting strategy.
Insert keys into initially empty 2-3-4 tree: S E A R C H I N G K L M

Insert S

Insert E

Insert A

Insert R

Insert C

Insert H

Insert I
Note the seemingly unnecessary split of E,I,R 4-node during insertion of K.
Results of an experiment with \( N \) uniformly distributed random keys from range \( \{1, \ldots, 10^9\} \) inserted into initially empty 2-3-4 tree:

<table>
<thead>
<tr>
<th>( N )</th>
<th>Tree depth</th>
<th>2-nodes</th>
<th>3-nodes</th>
<th>4-nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>39</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>7</td>
<td>414</td>
<td>257</td>
<td>24</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
<td>4451</td>
<td>2425</td>
<td>233</td>
</tr>
<tr>
<td>100000</td>
<td>13</td>
<td>43583</td>
<td>24871</td>
<td>2225</td>
</tr>
<tr>
<td>1000000</td>
<td>15</td>
<td>434671</td>
<td>248757</td>
<td>22605</td>
</tr>
<tr>
<td>10000000</td>
<td>18</td>
<td>4356849</td>
<td>2485094</td>
<td>224321</td>
</tr>
</tbody>
</table>
Relation of a 2-3-4 tree to a red-black tree

![Diagram showing the relation between a 2-3-4 tree and a red-black tree.](image-url)
**Ben Pfaff**: *Performance Analysis of BSTs in System Software, 2004, [3]*

Conclusions:

- Unbalanced BSTs are best when randomly ordered input can be relied upon;
- If random ordering is the norm but occasional runs of sorted order are expected, then red-black trees should be chosen.
- On the other hand, if insertions often occur in a sorted order, AVL trees excel when later accesses tend to be random,
- And splay trees perform best when later accesses are sequential or clustered.

Some consequences:

Managing virtual memory areas in OS kernel:

... Many kernels use BSTs for keeping track of virtual memory areas (VMAs): Linux before 2.4.10 used AVL trees, OpenBSD and later versions of Linux use red-black trees, FreeBSD uses splay trees, and so does Windows NT for its VMA equivalents...
<table>
<thead>
<tr>
<th>Tree / Time in msec / Order</th>
<th>Memory Management Supporting Web Browser</th>
<th>Artificial Uniformly Random Data</th>
<th>Secondary Peer Cache Tree</th>
<th>Compilation Identifiers Cross-References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BST   AVL    RB    splay</td>
<td>BST   AVL    RB    splay</td>
<td>BST   AVL    RB    splay</td>
<td>BST   AVL    RB    splay</td>
</tr>
<tr>
<td></td>
<td>15.67 3.65 3.78 2.63</td>
<td>1.63 1.67 1.64 1.94</td>
<td>3.94 4.07 3.78 7.19</td>
<td>4.97 4.47 4.33 4.00</td>
</tr>
<tr>
<td></td>
<td>4      2      3      1</td>
<td>1      3      2      4</td>
<td>2      3      1      4</td>
<td>4      3      2      1</td>
</tr>
</tbody>
</table>