Search trees

- Red-Black Tree
- Splay Tree
- 2-3-4 Tree
More general randomness

Choose a fraction $p$ between 0 and 1.
Rule: Fraction $p$ of elements with level $k$ pointers will have level $k+1$ elements as well.

On average: $\frac{1-p}{1-p}$ elements will be level 1 elements,
$(1-p) \cdot p$ elements will be level 2 elements,
$(1-p) \cdot p^2$ elements will be level 3 elements, etc.

This scheme corresponds to flipping a coin that has $p$ chance of coming up heads, $(1-p)$ chance of coming up tails.

Example of an experimental independent levels calculation with $p = 0.33$. 
Element level (probability):

\[
\begin{align*}
(1 - p) & \quad \text{level 1} \\
(1 - p) \cdot p & \quad \text{level 2} \\
\vdots & \\
(1 - p) \cdot p^{k-1} & \quad \text{level } k \\
\vdots & \\

\sum_{k=1}^{\infty} (1 - p) \cdot p^{k-1} &= \frac{1-p}{1-p} = 1
\end{align*}
\]

Expected number of pointers per element:

\[
\sum_{k=1}^{\infty} k \cdot (1 - p) \cdot p^{k-1} = (1 - p) \cdot \sum_{k=0}^{\infty} (k + 1) \cdot p^k = \frac{1-p}{(1-p)^2} = \frac{1}{1-p}
\]

This scheme corresponds to flipping a coin that has \( p \) chance of coming up heads, \((1-p)\) chance of coming up tails.
Experiment with Lehmer generator

\[ X_{n+1} = 16807 \, X_n \mod 2^{31} - 1 \]

seed = 23021905  // birth date of Derrick Henry Lehmer

Coin flipping:

\((X_n >> 16) & 1\)

Head = 1

128 nodes

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>Expected</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1/2</td>
<td>1/4</td>
</tr>
<tr>
<td></td>
<td>Actual</td>
<td>60</td>
<td>36</td>
<td>17</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Data structures and algorithms

Red-Black Trees

Petr Felkel

Exploited in Advanced Algorithms 2012-2020
Red-Black tree

Approximately balanced BST

\[ h_{RB} \leq 2 \cdot h_{BST} \quad (\text{height } \leq 2 \times \text{height of a balanced tree}) \]

Additional bit for COLOR = \{red | black\}

nil (non-existent child) = pointer to \textbf{nil} node
A binary search tree is a red-black tree if:

1. Every node is either red or black.
2. Every leaf (nil) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.
5. Root is black.

Black-height $bh(x)$ of a node $x$ is the number of black nodes on any path from $x$ to a leaf, not counting $x$.
Red-Black tree

Black-height $bh(x)$ of a node $x$ is the number of black nodes on any path from $x$ to a leaf, not counting $x$.

Black height $bh(x)$

Black height $bh(T) = 3$

Height $h(T) = 6$
Binary Search Tree -> RB Tree

black height $bh(T) = 4$

$\text{height } h(T) = 4$
Binary Search Tree -> RB Tree

black height $bh(T) = 3$

$h(T) = 4$
Binary Search Tree -> RB Tree

black height \( bh(T) = 3 \)

height \( h(T) = 4 \)
Binary Search Tree -> RB Tree

black height $bh(T) = 2$
$h(T) = 4$
Red-Black tree

Black-height $bh(x)$ of a node $x$
- is the number of black nodes on any path from $x$ to a leaf, not counting $x$
- is equal for all paths from $x$ to a leaf
- For given $h$ is $bh(x)$ in the range from $h/2$ to $h$
  - if $1/2$ of nodes red $\Rightarrow bh(x) \approx 1/2 \cdot h(x)$, $h(x) \approx 2 \lg(n+1)$
  - if all nodes black $\Rightarrow bh(x) = h(x) = \lg(n+1)$

Height $h(x)$ of a RB-tree rooted in node $x$
- is at maximum twice of the optimal height of a balanced tree
- $h \leq 2\lg(n+1)$ $\Rightarrow$ .... $h \in \Theta(\lg(n))$
RB-tree height proof  [Cormen, p.264]

A red-black tree with \( n \) internal nodes has height \( h \) at most \( 2\log(n+1) \)

Proof 1. Show that subtree starting at \( x \) contains at least \( 2^{bh(x)}-1 \) internal nodes.

By induction on height of \( x \):
I. If \( x \) is a leaf, then \( bh(x) = 0 \), \( 2^{bh(x)}-1 = 0 \) internal nodes  //... nil node
II. Consider \( x \) with height \( h \) and two children (with height at most \( h -1 \))
   – \( x \)'s children black-height is either \( bh(x) -1 \) or \( bh(x) \)  // \( x \) is black or red
   – Ind. hypothesis: \( x \)'s children subtree has at least \( 2^{bh(x)-1} -1 \) internal nodes
   – So subtree rooted at \( x \) contains at least
     \[ (2^{bh(x)-1} -1) + (2^{bh(x)-1} -1) + 1 = 2^{bh(x)} -1 \] internal nodes => proved

Proof 2. Let \( h = \) height of the tree rooted at \( x \)
   – min \( \frac{1}{2} \) nodes are black on any path to leaf  => \( bh(x) \geq h / 2 \)
   – Thus, \( n \geq 2^{h/2} - 1 \)  => \( n + 1 \geq 2^{h/2} \)  => \( \log(n+1) \geq h / 2 \)
   – \( h \leq 2\log(n+1) \)
Search is performed as in simple BST, node colors do not influence the search.

Search in R-B tree with N nodes takes
1. In general -- at most $2 \times \lg(N+1)$ key comparisons.
2. In best case when keys are generated randomly and uniformly -- cca $1.002 \times \lg(N)$ key comparisons,
   very close to the theoretical minimum.
Inserting in Red-Black Tree

Color new node \( x \) Red
Insert it as in the standard BST

If parent \( p \) is Black, stop. Tree is a Red-Black tree.
If parent \( p \) is Red (3+3 cases)…

resp.

While \( x \) is not root and parent is Red

if \( x \)'s uncle is Red then case 1 // propagate red up
else { if \( x \) is Right child then case 2 // double rotation
case 3 } // single rotation

Color root Black
If parent is Black, stop. Tree is a Red-Black tree.
Inserting in Red-Black Tree

\[ x \text{'s parent is Red} \]
\[ x \text{'s uncle } y \text{ is Red} \]
\[ x \text{ is a Left child} \]

Case 1a

- x is node of interest
- x's uncle is Red
- bh(x) increased by one

Loop: \( x = x.p.p \)

Recolor
Inserting in Red-Black Tree

- x’s parent is Red
- x’s uncle y is Red
- x is a Right child

Case 1b

Loop: x = x.p.p

- x is node of interest
- x's uncle is Red

bh(x) increased by one
Inserting in Red-Black Tree

- x’s parent is Red
- x's uncle y is Black
- x is a Right child

Case 2

x is a Right child
x's uncle is Black

transform to Case 3
Inserting in Red-Black Tree

x’s parent is Red
x's uncle y is Black
x is a Left child

Terminal case, tree is a Red-Black tree

Case 3

x is a Left child
x's uncle is Black
Inserting in Red-Black Tree

Cases Right from the grandparent are symmetric
RB-INSERT($T, x$)

1. **Tree-Insert**($T, x$)
2. $color[x] \leftarrow \text{RED}$
3. **while** $x \neq \text{root}[T]$ and $color[p[x]] = \text{RED}$
   
   **do if** $p[x] = \text{left}[p[p[x]]]$
   
   **then** $y \leftarrow \text{right}[p[p[x]]]$  
   
   **if** $color[y] = \text{RED}$
   
   **then** $color[p[x]] \leftarrow \text{BLACK}$  
   
   $color[y] \leftarrow \text{BLACK}$  
   
   $color[p[p[x]]] \leftarrow \text{RED}$  
   
   $x \leftarrow p[p[x]]$  
   
   **end if**

4. **else if** $x = \text{right}[p[x]]$
   
   **then** $x \leftarrow p[x]$  
   
   **end if**

5. **end do while**

6. **else** (same as then clause with "right" and "left" exchanged)

7. $color[root[T]] \leftarrow \text{BLACK}$

**Red uncle y -&gt; recolor up**

[Cormen90]
Inserting in Red-Black Tree

Insertion in $\Theta(\log(n))$ time
Requires at most two rotations
Deleting in Red-Black Tree

Find node to delete
Delete node as in a regular BST
Node \( y \) to be physically deleted will have at most one child \( x \)!!!

If we delete a Red node, tree still is a Red-Black tree, stop
Assume we delete a black node

Let \( x \) be the child of deleted (black) node \( y \)
If \( x \) is red, color it black and stop

while(\( x \) is not root) AND ( \( x \) is black) 
move \( x \) with virtual black mark through the tree 
(If \( x \) is black, mark it virtually double black \( \text{A} \))
//note that the whole \( x \) 's subtree lost 1 unit of black height
Deleting in Red-Black Tree

```plaintext
while(x is not root) AND ( x is black) {
    // move x with virtual black mark through the tree
    // just recolor or rotate other subtree up (decrease bh in R subtree)
    if (sibling is red)
        -> Case 1: Rotate right subtree up, color sibling black, and continue in left subtree with the new sibling
    if (sibling is black with both black children)
        -> Case 2: Color sibling red and go up
    else // black sibling with one or two red children
        if(red left child) -> Case 3: rotate to surface
    Case 4: Rotate right subtree up
}
```
Deleting in R-B Tree - Case 1

\( x \) is the child of the physically deleted black node \( \Rightarrow \) double black

\( x \)'s sibling \( w \) is red

\( x \)'s parent must be black

\( x \) stays at the same black height

[Possibly transforms to case 2a and terminates – depends on 3,4]
Deleting in R-B Tree - Case 2a

- x’s sibling w is black
- x’s parent is red
- x’s sibling left child is black
- x’s sibling right child is black

Case 2a

Terminal case, tree is Red-Black tree

Note that A's subtree had less by 1 black height than D's subtree
Deleting in R-B Tree - Case 2b

- x’s sibling w is black
- x’s parent is black
- x’s sibling left child is black
- x’s sibling right child is black

Decreases x black height by one
Note that A’s subtree had less by 1 black height than D’s subtree

Case 2b
Recolor(w) + black up

continue with new x
Deleting in R-B Tree - Case 3

- x’s sibling w is black
- x’s parent is either
- x’s sibling left child is red  // impossible to color w red
- x’s sibling right child is black

Transform to case 4
x stays at same black height
Deleting in R-B Tree - Case 4

- **x**’s sibling **w** is black
- **x**’s parent is either
- **x**’s sibling left child is either
- **x**’s sibling right child is red // impossible to color **w** red

Case 4

Terminal case, tree is Red-Black tree (D inherits the color of B)
Deleting in Red-Black Tree

\[
\text{RB-DELETE}(T, z)
\]

1. if \(\text{left}[z] = \text{nil}[T]\) or \(\text{right}[z] = \text{nil}[T]\)
2. then \(y \leftarrow z\)
3. else \(y \leftarrow \text{TREE-SUCCESSOR}(z)\)

4. if \(\text{left}[y] \neq \text{nil}[T]\)
5. then \(x \leftarrow \text{left}[y]\)
6. else \(x \leftarrow \text{right}[y]\)

7. \(p[x] \leftarrow p[y]\)

8. if \(p[y] = \text{nil}[T]\)
9. then \(\text{root}[T] \leftarrow x\)
10. else if \(y = \text{left}[p[y]]\)
11. then \(\text{left}[p[y]] \leftarrow x\)
12. else \(\text{right}[p[y]] \leftarrow x\)

13. if \(y \neq z\)
14. then \(\text{key}[z] \leftarrow \text{key}[y]\)
15. ▷ If \(y\) has other fields, copy them, too.

16. if \(\text{color}[y] = \text{BLACK}\)
17. then \(\text{RB-DELETE-FIXUP}(T, x)\)

18. return \(y\)

Notation similar to AVL
\(z = \text{logically} \) removed
\(y = \text{physically} \) removed
\(x = y\)'s only child

[Dsa]
[Cormen90]
RB-DELETE-FIXUP(T, x)

1. while x ≠ root[T] and color[x] = BLACK

2. do if x = left[p[x]]

3. then w ← right[p[x]]

4. if color[w] = RED

5. then color[w] ← BLACK

6. color[p[x]] ← RED

7. LEFT-ROTATE(T, p[x])

8. w ← right[p[x]]

9. ▷ Case 1

10. if color[left[w]] = BLACK and color[right[w]] = BLACK

11. then color[w] ← RED

12. x ← p[x]

13. ▷ Case 2

14. else if color[right[w]] = BLACK

15. then color[left[w]] ← BLACK

16. color[w] ← RED

17. RIGHT-ROTATE(T, w)

18. w ← right[p[x]]

19. ▷ Case 3

20. color[w] ← color[p[x]]

21. color[p[x]] ← BLACK

22. color[right[w]] ← BLACK

23. LEFT-ROTATE(T, p[x])

24. ▷ Case 4

25. x ← root[T]

26. ▷ Case 4

27. end

28. end

29. DSA

30. [Cormen90]
Deleting in R-B Tree

Delete time is \( \Theta(\log(n)) \)
At most three rotations are done
R-B Tree vs. AVL Tree

• Faster insertion and deletion operations (fewer rotations are done due to relatively relaxed balancing).
• Requires only 1 bit of information per node.
• AVL trees provide faster lookups.

=> Red-Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in databases where faster retrievals are required.
<table>
<thead>
<tr>
<th>Křížovková pauza</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>puška</th>
<th>červeno</th>
<th>české město</th>
<th>unavím</th>
<th>měkký kov</th>
<th>iniciály českého zpěváka a baviče</th>
<th>atmosférický světelný jev</th>
<th>obejda</th>
<th>mezinárodní značka Rumunska</th>
</tr>
</thead>
<tbody>
<tr>
<td>doprovod manželky šlechtice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. část tajenky</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>identifikátor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bývalý francouzský tenista</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>značka Tesly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>patřící obyvateli ráje</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>staré zájmeno programovací jazyk</td>
<td>Roosveltovy iniciály tlak krve (zkr.)</td>
<td>ušlechtilý kov</td>
<td>slovenská rocková skupina popěvek</td>
<td>buď nápomocný</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Splay tree, 2-3-4 tree

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.

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

Zig-Zag rotation is identical to the double (LR or RL) rotation in AVL tree.
Note the extremely inefficient shape of the resulting tree.
Find 1

Key 1 is the deepest key in the tree.

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

Example

Scheme - Progress of the two most unfavourable Find operations.

Note the relatively favourable shape of the resulting tree.

Pokročilá Algoritmizace, A4M33PAL, ZS 2012/2013, FEL ČVUT, 12/14
Delete(k)

1. Find(k); // This splay 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 splay 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
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**.
**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
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

1. Insert S
   - S

2. Insert E
   - ES

3. Insert A
   - AES

4. Insert R
   - A R S

5. Insert C
   - A C R S

6. Insert H
   - A C H R S

7. Insert I
   - A C H I S
Insert I

AC HI S

Insert N

AC HI NS

Insert G

AC GH NS

Insert K

AC GH KN S

Insert L

AC GH KLN S

Insert M

AC GH KMNS

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, ..., 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

2-3-4 tree

Relation to R-B tree