Self-Balancing Binary Search Trees

EECS 214, Fall 2018
A self-balancing BST

Random binary search trees are very likely to be balanced
Self-balancing trees are guaranteed to be balanced
Balanced search tree survey
AVL trees

Due to: Georgy Adelson-Velsky & Evgenii Landis (1962)

Main idea: Maintain a balance factor giving the difference between each node’s subtrees’ heights

Local invariant: Balance factor between -1 and 1, maintained via rotations

Global invariant: Tree is approximately height-balanced
2–3 trees

**Due to:** John Hopcroft (1970)

**Main idea:** 2-nodes have one element and two children; 3-nodes have two elements and three children

**Local invariant:** All subtrees of a node have the same height

**Global invariant:** Every leaf is at the same depth

**Advantage:** Faster insertions, slower lookups (compared to AVL)
B-trees

Due to: Rudolf Bayer & Ed McCreight (1971)

Main idea: Generalization of 2–3 trees up to $k$ children.

Local invariant: Like 2–3 trees, but allow up to $k/2$ missing children.

Global invariant: Every leaf is at the same depth

Use: On-disk databases (or modern memory hierarchies)

Advantage: Larger nodes means fewer disk accesses (or cache misses)
2–3–4 trees (a/k/a 2–4 trees)

Due to: Rudolf Bayer (1972)

Main idea: B-tree of order 4.

Why interesting: Isometry of red–black tree
Red–black trees

Due to: Leonidas J. Guibas & Robert Sedgewick (1978)

Main idea: Every node has an extra bit marking it “red” or “black”

Local invariant: No red node has a red parent

Global invariant: Equal number of black nodes from root to every leaf

Advantage: Faster insertions, slower lookups (compared to AVL); easier representation than 2–3(–4) trees
Splay trees (randomized or amortized!)

Due to: Daniel Sleator & Robert Tarjan (1985)

Main idea: Cache recently accessed elements near the root of the tree

Local invariant: Complicated; required amortized analysis

Global invariant: Paths are very likely to be $O(\log n)$

Advantage: Self optimizing; no extra balance data
AVL trees
Example of an AVL tree
Local invariant maintains global property

- Balance factors are maintained locally
- Never recompute them from scratch
- Yet the whole tree stays reasonably balanced
AVL insertion

- First do a normal leaf insertion
- Track balance factors on the way back up to the root
- Adjust with rotations as necessary
AVL insertion example

Let’s insert H:
AVL insertion example

Let’s insert H:
AVL insertion example

Let’s insert H:

```
+1  -1  1  +1  +1  0  -1  0  0  0
```

```
Let's insert H:
```

```
AVL insertion example
```

```
+1  -1  1  +1  +1  0  -1  0  0  0
```
AVL insertion example

Let’s insert H:

```
        J
       /  
      F    P
     /  \
    D   G  L
   /  \
  C   H  N
```

Inserting H results in a balance factor of +1 for node J, which is the only violation of AVL property. To maintain balance, we need to perform rotations.
Another AVL insertion example

Let’s insert B:
Another AVL insertion example

Let’s insert B:
Another AVL insertion example

Let's insert B:
Another AVL insertion example

Let’s insert B:
Another AVL insertion example

Let’s insert B:

```
+1
```

```
-1
```

```
0
```

```
0
```

```
+1
```

```
0
```

```
-1
```

```
0
```

```
0
```

```
0
```

```
0
```

```
0
```
Another AVL insertion example

Let’s insert B:

```
+1
-1
0
0
+1
+1
0
0
0
0
0
0
0
```
Maintaining the AVL property

Suppose we have an AVL tree:

(Convention: triangles represent equal-height subtrees.)
Maintaining the AVL property

Suppose we have an AVL tree:

(Convention: triangles represent equal-height subtrees.)

Right now the balance factor is 0. So if we insert into A or C and that subtree grows in height, it becomes -1 or 1.
Maintaining the AVL property

Right now the balance factor at B is +1.
Suppose we insert into A. What happens to B’s balance factor?
Maintaining the AVL property

Right now the balance factor at B is +1.

Suppose we insert into A. What happens to B’s balance factor?

- If no change in A’s height then no change in B’s balance
- If A’s height grows then B’s balance factor goes to 0
Maintaining the AVL property

Right now the balance factor at B is +1.

Suppose we insert into C. What happens to B’s balance factor?
Maintaining the AVL property

Right now the balance factor at B is +1.

Suppose we insert into C. What happens to B’s balance factor?

- If no height change then B’s balance doesn’t change
- If C grows then B’s balance factor becomes +2
Maintaining the AVL property

Right now the balance factor at B is +1.

Suppose we insert into C. What happens to B’s balance factor?

- If no height change then B’s balance doesn’t change
- If C grows then B’s balance factor becomes +2—not okay!
Maintaining the AVL property

Right now the balance factor at B is +1.

Likewise, suppose we insert into E. What happens to B’s balance factor?

- If no height change then B’s balance doesn’t change
- If E grows then B’s balance factor becomes +2—not okay!
The right-right case

If the height of the right-right subtree (E) increases, we get a situation like this:
The right-right case

If the height of the right-right subtree (E) increases, we get a situation like this:
The right-left case

If the height of the right-left subtree (C) increases, we get a situation like this:

```
B +2
    /
   +1
  /   
A     C
```

But this is now the right-right case, which we know how to handle!
The right-left case

If the height of the right-left subtree (C) increases, we get a situation like this:
The right-left case

If the height of the right-left subtree (C) increases, we get a situation like this:

But this is now the right-right case, which we know how to handle!
Maintaining the AVL property

- We’ve seen the right-right and right-left cases
- The left-left and left-right cases are symmetrical
- Deletion is like ordinary BST deletion, with the same rebalancing cases

See `avl.rkt`. 
Red–black trees
The red–black tree rules

The rules:

1. Nodes are colored red or black.
The red–black tree rules

The rules:

1. Nodes are colored red or black.
2. The root is always black.
The red–black tree rules

The rules:

1. Nodes are colored red or black.
2. The root is always black.
3. “Dummy leaves” are black.
The red–black tree rules

The rules:

1. Nodes are colored red or black.
2. The root is always black.
3. “Dummy leaves” are black.
4. Every red node has a black parent.
The red–black tree rules

The rules:

1. Nodes are colored red or black.
2. The root is always black.
3. “Dummy leaves” are black.
4. Every red node has a black parent.
5. For every node, all paths to leaves have the same “black height.”
Red–black colorability
Red–black tree insertion

1. Leaf insert, like any other BST
Red–black tree insertion

1. Leaf insert, like any other BST
2. Color new node red.
Red–black tree insertion

1. Leaf insert, like any other BST
2. Color new node red.
3. If parent is also red (violating rule 4), color parent black and look for problems further up.
Next time: C and C++