COMP251: Binary search trees, AVL trees & AVL sort

Jérôme Waldispühl
School of Computer Science
McGill University
From Lecture notes by E. Demaine (2009)

Outline

• Review of binary search trees
• AVL-trees
• Rotations
• BST & AVL sort

Binary search trees (BSTs)

• T is a rooted binary tree
• Key of a node x ≥ keys in its left subtree.
• Key of a node x ≤ keys in its right subtree.

Operations on BSTs

• Search(T,k): Θ(h)
• Insert(T,x): Θ(h)
• Delete(T,x): Θ(h)

Where h is the height of the BST.

Height of a tree

Height(n): length (edges) of longest downward path from node n to a leaf.

Height(x) = 1 + max( height(left(x)), height(right(x)) )

Good vs. Bad BSTs

Balanced
h=Θ( log n )

Unbalanced
h=Θ( n )
AVL trees

**Definition:** BST such that the heights of the two child subtrees of any node differ by at most one.

$|h_{left} - h_{right}| \leq 1$

- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take $O(\log n)$ in average and worst cases.

Insert in AVL trees

1. Insert as in standard BST
2. Re-establish AVL tree properties

Insert in AVL trees

How to restore AVL property?

Rotations

```
Rotations preserve the BST property.
Proof: elements in B are $\geq x$ and $\leq y$.
```
15-01-19

**Insert in AVL trees**

Right rotation at 57

Insert(T, 50)

RotateRight(T, 57)

How to restore AVL property?

**Insert in AVL trees**

Left rotation at 43

Insert(T, 50)

RotateLeft(T, 43)

**Insert in AVL trees**

Right rotation at 57

Insert(T, 50)

RotateRight(T, 57)

**Insert in AVL trees**

1. Suppose x is lowest node violating AVL
2. If x is right-heavy:
   - If x’s right child is right-heavy or balanced: Left rotation (case A)
   - Else: Right followed by left rotation (case B)
3. If x is left-heavy:
   - If x’s left child is left-heavy or balanced: Right rotation (symmetric of case A)
   - Else: Left followed by right rotation (sym. of case B)
4. then continue up to x’s ancestors.

**Case A**

**Case B**
Running time AVL insertion

- Insertion in $O(h)$
- At most 2 rotations in $O(1)$
- Running time is $O(h) + O(1) = O(h) = O(\log n)$ in AVL trees.

In-order traversal & BST

```
inorderTraversal(treeNode x)
inorderTraversal(x.leftChild);
print x.value;
inorderTraversal(x.rightChild);
```

- Print the nodes in the left subtree (A), then node x, and then the nodes in the right subtree (B)
- In a BST, keys in $A \leq x$, and keys in $B \geq x$.
- In a BST, it prints first keys $\leq x$, then $x$, and then keys $\geq x$.

Running time of BST sort

- In-order traversal is $O(n)$
- Running time of insertion is $O(h)$

**Best case:** The BST is always balanced for every insertion. $O(\log(n))$

**Worst case:** The BST is always un-balanced. All insertions on same side.

$$\sum_{i=1}^{n} \frac{n(n-1)}{2} = O(n^2)$$

BST sort

1. Build a BST from the list of keys (unsorted)
2. Use in-order traversal on the BST to print the keys.

```
36 57 12 8
```

Running time of BST sort: insertion of $n$ keys + tree traversal.

AVL sort

Same as BST sort but use AVL trees and AVL insertion instead.

- Worst case running time can be brought to $O(n \log n)$ if the tree is always balanced.
- Use AVL trees (trees are balanced).
- Insertion in AVL trees are $O(h) = O(\log n)$ for balanced trees.