Announcements

• Assignment 1 will be post tomorrow
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 $x \geq$ keys in its left subtree.
- Key of a node $x \leq$ keys in its right subtree.
Operations on BSTs

- Search(T, k): $\Theta(h)$
- Insert(T, k): $\Theta(h)$
- Delete(T, k): $\Theta(h)$

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 + \text{max}( \text{height(left(x))), \text{height(right(x))} )
Example

\[ h(a) = ? \]

\[ = 1 + \max ( h(b), h(g) ) \]
\[ = 1 + \max ( 1 + \max ( h(c), h(d) ), 1 + h(h) ) \]
\[ = 1 + \max ( 1 + \max ( 0, h(d) ), 1 + 0 ) \]
\[ = 1 + \max ( 1 + \max ( 0, 1 + h(e) ), 1 ) \]
\[ = 1 + \max ( 1 + \max ( 0, 1 + ( 1 + h(f) ) ), 1 ) \]
\[ = 1 + \max ( 1 + \max ( 0, 1 + ( 1 + 0 ) ), 1 ) \]
\[ = 1 + \max ( 3, 1 ) \]
\[ = 4 \]
Height vs. Depth

Good vs. Bad BSTs

Balanced
\[ h = \Theta(\log n) \]

Unbalanced
\[ h = \Theta(n) \]

This is technically a valid BST but in practice, it’s a sorted linked list 😞
AVL trees (Adelson-Velsky, Landis)

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

\[ |h_{\text{left}} - h_{\text{right}}| \leq 1 \]

- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take \( O(\log n) \) in average and worst cases.
- To satisfy the definition, the height of an empty subtree is \(-1\)

One node: height=0. Zero nodes: height=-1
Height of an AVL tree

N_h = minimum #nodes in an AVL tree of height h.

N_h > 2^k \cdot N_{(h-2k)}

Let k = h/2 − 1:
N_h > 2^{(h/2−1)} \cdot N_1
N_h > c \cdot 2^{(h/2−1)}

We can generalize this expression by multiplying shorter subtrees by higher exponents of 2.

N_h = 1 + N_{h-1} + N_{h-2}
> 2 \cdot N_{h-2}
⇒ N_h > \Theta(2^{h/2})
⇒ h < 2 \cdot \log N_h
⇒ h = O(\log n)

We confirmed the height grows with \log N in the worst case, unlike BSTs.

(Note: a tighter bound can be found using Fibonacci numbers)
Balance factor

\[ \begin{align*}
\text{Left tree is higher (left-heavy)} & \quad \Rightarrow & \quad N_{h-2} > N_{h-1} \\
\text{Balanced} & \quad = & \quad N_{h-1} = N_{h-1} \\
\text{Right tree is higher (right-heavy)} & \quad \Rightarrow & \quad N_{h-3} > N_{h-1} \\
\end{align*} \]

Violates AVL property
Insert in AVL trees

1. Insert as in standard BST
2. Restore AVL tree properties
Just like BSTs, the AVL definition is recursive. All children of the root of an AVL tree are the root of an AVL tree.

Insert(T, 15)
Insert in AVL trees

Insert(T, 15)

How to restore AVL property?  Bottom-up!
Rotations change the tree structure & **preserve the BST property**.

**Proof:** elements in B are $\geq x$ and $\leq y$...

In both cases, everything in $A < x < $ everything in $B < y < $ everything in $C$
Example (right rotation)
Example: Insert in AVL trees

Right rotation at 27

We call it a rotation AT node 27 because 27 is the root that gets “kicked”
We intervene at the deepest node that breaks AVL rules.
Example: Insert in AVL trees

Insert(T, 50)

RotateRight(T, 57)

How to restore AVL property?

Right rotation at 57

Insert(T, 50)

RotateRight(T, 57)

Rotating right didn’t fix the problem?! Let’s try something else.
Example: Insert in AVL trees

Left rotation at 43

We remove the zig-zag pattern

RotateLeft(T,43)
Example: Insert in AVL trees

RotateRight

Right rotation at 57

We needed to get rid of the « zig-zag » before doing the right rotation!

AVL property restored!

RotateRight(T,57)
Algorithm: Maintaining AVL

1. Suppose x is lowest node violating AVL
2. If x is right-heavy:
   - If x’s right child is right-heavy or balanced (no zig-zag):
     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 (no zig-zag):
     Right rotation (symmetric of case A)
   - Else: Left followed by right rotation (sym. of case B)
4. then continue up to x’s ancestors. (bottom-up approach)

Proving cases A and B is sufficient because all AVL operations are symmetric
Two cases:
The right child is
a) right-heavy or
b) balanced

Proof: Case A

Left rotation

Left rotation

Left rotation
Proof: Case B

The right child is left-heavy (zig zag)

Right rotation at y &
Left rotation at x

Intuition: here, notice that node z looks like it « belongs » in the center, and does end up as the root!
Proof: Case B

Right rotation at y

The first right rotation brings us back to case A

Left rotation at x
AVL insertion

• Insert key $k$ as in standard BST

• Starting from $k$, find the first ancestor of $k$ that is unbalanced

• Rebalance the tree performing the appropriate rotations
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.

Once we fix the unbalanced subtree its height will decrease by one. This means that it will be restored to its previous height before insertion. Hence all its ancestors will go back having their original heights.

Remember we already proved $h$ asymptotically grows with $\log n$ in the worst case.
Sorting with BSTs

1. BST sort
   - Simple method using BSTs
   - Problem: Worst case $O(n^2)$

2. AVL sort
   - Use AVL trees to get $O(n \cdot \log n)$

   AVL tree operations are guaranteed to be $O(\log n)$

   This happens because the BST worst case is basically a diagonal linked list
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 ≤ x, and keys in B ≥ x.
- In a BST, it prints first keys ≤ x, then x, and then keys ≥ x.
In-order traversal & BST

```
8, 12, 15, 20, 27, 36, 43, 57
```

All keys come out sorted!
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  12  8  57  43  27
```

```
→ 8, 12, 27, 36, 43, 57
```

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

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

**Best case:** The BST is always balanced for every insertion.

$$\Omega(n \log(n))$$

In the best case, a BST always respects AVL properties without being « forced » to do so

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

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

The BST worst case is basically a diagonal linked list
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.