Advanced Tree Structures

CMSC132
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”

Insert: 12
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”
Degenerate Search Trees

- Standard BST only as good as its insertion order
- Very easy to make “degenerate”
- Congratulations, you have a linked list
- How long to find/insert/delete?
Modified Binary Search Trees

The real world often doesn't use stock BSTs

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Tree Type</th>
<th>Use case</th>
<th>Tree Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation</td>
<td>AVL trees, red-black trees</td>
<td>Strings</td>
<td>Tries</td>
</tr>
<tr>
<td>Multi-way</td>
<td>2-3 trees</td>
<td>File systems/db</td>
<td>B/B+ trees</td>
</tr>
<tr>
<td>Rebuild</td>
<td>Scapegoat trees</td>
<td>Usage frequency</td>
<td>Splay trees</td>
</tr>
<tr>
<td>Randomization</td>
<td>Treaps, Skiplists</td>
<td>Multi-dimensional</td>
<td>K-d trees</td>
</tr>
</tbody>
</table>

Many of these covered in detail in CMSC420: Advanced Data Structures
AVL Trees

- Invented by Adelson-Velsky and Landis in 1962
- **Approach:** keep height of subtrees roughly equal
- **Strategy:** when unequal, rebalance tree with rotations
- **Outcome:** worst case $O(\log n)$ performance
Left Rotation

A moves down
C moves up

5

A

B

C

8

\[ v < 5 \]

\[ 5 < v < 8 \]

\[ v > 8 \]
Left Rotation

A moves down
C moves up
Left Rotation

A moves down
C moves up

Fixes:
- C too tall
- A too short
Right rotation

A moves up
C moves down
Right rotation

A moves up
C moves down
Right rotation

A moves up
C moves down
Fixes:
- C too short
- A too tall
Problem

Left and right rotation move A & C up & down, but what about B?
Problem

Left and right rotation move A & C up & down, but what about B?

Rotate twice!
Left-Right Rotation

Let’s fix the height of B1/B2 - We’ve already seen this can’t be done with a single rotation
Left-Right Rotation

First, rotate left so that B2 moves up and A moves down.
Left-Right Rotation

First, rotate left so that B2 moves up and A moves down.
Then, rotate right so that C moves down and the subtree rooted at 1 moves up.
Specific Left-Right Rotation Example

- Insert 20, then 10, then 15 into an empty tree.
- Insertion of 15 unbalances the tree, so we must perform a LR rotation.

Starting with unbalanced tree

Rotated left around “10”

Rotated right around “15” to restore balance
Right-Left Rotation

The same, but in reverse

```
5
/   \
/     \
/       \
v < 5    v > 9
7
/   \
/     \
/       \
v < 5    v > 9
A       C
/   \
/     \
/       \
v < 5    v > 9
B_1     B_2
/ \
/  \
v < 5 5 < v < 9
5 < v < 7

5
/   \
/     \
/       \
v < 5    v > 9
A       C
/   \
/     \
/       \
v < 5    v > 9
B_1     B_2
/ \
/  \
v < 5 5 < v < 9
5 < v < 7

7
/   \
/     \
/       \
v < 5    v > 9
A       B_1
/ \
/  \
v < 5 5 < v < 9
5 < v < 7

5
/   \
/     \
/       \
v < 5    v > 9
B_2     C
/ \
/  \
v < 5 5 < v < 9
5 < v < 7

9
/   \
/     \
/       \
v < 5    v > 9
B_2     C
/ \
/  \
v < 5 5 < v < 9
5 < v < 7
```

```
Rotation Rules

- These 4 rotations allow an AVL tree to self rebalance
- Rotate based off of which grandchild is too tall
  1. Left-left: right rotation
  2. Left-right: left-right rotation
  3. Right-left: right-left rotation
  4. Right-right: left rotation
Implementation Details

Code for AVL trees is “relatively” simple:

1. Add extra field for keeping track of height in Node class
2. After modification, update appropriate height fields
3. After modification, rebalance at each level if needed
4. Key search as normal

Extra functions: rebalance, updateHeight, and rotations
Result

- Other trees can be more involved
- AVL isn’t perfect - Java uses red-black trees
  - AVL provides faster lookup and slower insert/remove
  - R-B provides faster insertion/removal and slower lookup
  - R-B uses slightly less storage
  - R-B is harder to do in a 30 minute presentation
- AVL Demo:
Resources

These are the notes from Professor David Mount’s CMSC420 class from Fall 2020.


AVL trees on Wikipedia:

https://en.wikipedia.org/wiki/AVL_tree
References

End of presentation

(Subsequent slides include some notes for TAs & extra material on keeping track of node heights)
Keeping track of height

class Node {
  T data;
  Node left;
  Node right;
  int height;
}

int height(Node node) {
  if (node == null) return 0;
  return Math.max(height(node.left), height(node.right)) + 1;
}
Keeping track of height

class Node {
    T data;
    Node left;
    Node right;
    int height;
}

void updateHeight(Node node) {
    if (node == null) return;
    node.height = Math.max(height(node.left), height(node.right)) + 1;
}

int height(Node node) {
    return node == null ? 0 : node.height;
}
TAs:

- Subtree heights and the height difference of subtrees is favored over the terminology “balance factor”
- TAs can theme the presentation however they wish, but colors of diagrams may be adversely affected