Lecture 10: BST and AVL Trees

CSE 373: Data Structures and Algorithms
Warm Up

Is valid BST? No
Height? 2

Is valid BST? Yes
Height? 3
Administrivia
Trees
Binary Search Trees

A binary search tree is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.
Implement Dictionary

Binary Search Trees allow us to:
- quickly find what we’re looking for
- add and remove values easily

Dictionary Operations:
Runtime in terms of height, “h”
get() – O(h)
put() – O(h)
remove() – O(h)

What do you replace the node with?
Largest in left sub tree or smallest in right sub tree
Height in terms of Nodes

For “balanced” trees $h \approx \log_c(n)$ where $c$ is the maximum number of children

Balanced binary trees $h \approx \log_2(n)$

Balanced trinary tree $h \approx \log_3(n)$

Thus for balanced trees operations take $\Theta(\log_c(n))$
Unbalanced Trees

Is this a valid Binary Search Tree?
Yes, but...

We call this a **degenerate tree**

For trees, depending on how balanced they are,
Operations at worst can be $O(n)$ and at best
can be $O{\log n}$

How are degenerate trees formed?
- insert(10)
- insert(9)
- insert(7)
- insert(5)
Implementing Dictionary with BST

```java
public boolean contains(K key, BSTNode node) {
    if (node == null) {
        return false;  // +C1
    }
    int compareResult = compareTo(key, node.data);  // +C2
    if (compareResult < 0) {
        return contains(key, node.left);  // +T(n/2) best  + T(n-1) worst
    } else if (compareResult > 0) {
        return contains(key, node.right);  // +T(n/2) best  + T(n-1) worst
    } else {  // +C3
        return true;
    }
}
```

**Best Case (assuming key is at the bottom)**

\[
T(n) = \begin{cases} 
C & \text{when } n < 0 \text{ or key found} \\
T \left( \frac{n}{2} \right) + C & \text{otherwise}
\end{cases}
\]

**Worst Case (assuming key is at the bottom)**

\[
T(n) = \begin{cases} 
C & \text{when } n < 0 \text{ or key found} \\
T(n - 1) + C & \text{otherwise}
\end{cases}
\]
Measuring Balance

Measuring balance:

For each node, compare the heights of its two sub trees

Balanced when the difference in height between sub trees is no greater than 1
Meet AVL Trees

**AVL Trees** must satisfy the following properties:

- **binary trees**: all nodes must have between 0 and 2 children
- **binary search tree**: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- **balanced**: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

AVL stands for Adelson-Velsky and Landis (the inventors of the data structure)
Is this a valid AVL tree?

Is it...
- Binary  yes
- BST  yes
- Balanced?  yes
Is this a valid AVL tree?

Is it...
- Binary: yes
- BST: yes
- Balanced?: no

Height = 0
Height = 2
Is this a valid AVL tree?

Is it...
- Binary  yes
- BST    no
- Balanced? yes

9 > 8
Implementing an AVL tree dictionary

Dictionary Operations:
get() – same as BST
containsKey() – same as BST
put() - Add the node to keep BST, fix AVL property if necessary
remove() - Replace the node to keep BST, fix AVL property if necessary

Unbalanced!
Rotations!

Insert ‘c’

Unbalanced!

Balanced!
Rotate Left

parent’s right becomes child’s left, child’s left becomes its parent

Insert ‘c’

Unbalanced!

Balanced!
Rotate Right
parent’s left becomes child’s right, child’s right becomes its parent

put(16);
Rotate Right
parent’s left becomes child’s right, child’s right becomes its parent

```
put(16);
```

```
5 3 7 6 10 22 17 16 20 24 19
```

```
height
```

```
Rotate Right
parent’s left becomes child’s right, child’s right becomes its parent
```

```
3 15
  /
  2
  /
  8
  |
  1
  |
  4
  |
  0
  |
  3
  |
  0
  |
  6
  |
  0
  |
  10
  |
  1
  |
  17
  |
  1
  |
  16
  |
  0
  |
  20
  |
  0
  |
  24
```

```
height
```
So much can go wrong

Unbalanced!

Unbalanced!

Unbalanced!

Parent’s right becomes child’s left
Child’s left becomes its parent

Rotate Left

Parent’s left becomes child’s right
Child’s right becomes its parent

Rotate Right
Two AVL Cases

**Line Case**
Solve with **1** rotation

- **Rotate Right**
  - Parent’s left becomes child’s right
  - Child’s right becomes its parent

- **Rotate Left**
  - Parent’s right becomes child’s left
  - Child’s left becomes its parent

**Kink Case**
Solve with **2** rotations

- **Right Kink Resolution**
  - Rotate subtree left
  - Rotate root tree right

- **Left Kink Resolution**
  - Rotate subtree right
  - Rotate root tree left
Double Rotations 1

Insert 'c'

Unbalanced!
Double Rotations 2

Unbalanced!

- a
- d
- e
- c
- X
- Z

- W
- Y

- d
- a
- e
- c
- X
- Z

- W
- Y
- X
- Z
Implementing Dictionary with AVL

public boolean contains(K key, AVLNode node) {
    if (node == null) { +C1
        return false;
    }
    int compareResult = compareTo(key, node.data); +C2
    if (compareResult < 0) { +T(n/2)
        return contains(key, node.left);
    } else if (compareResult > 0) { +T(n/2)
        return contains(key, node.right);
    } else { +C3
        return true;
    }
}

Worst Case Improvement Guaranteed with AVL

\[ T(n) = \begin{cases} 
  C & \text{when } n < 0 \text{ or key found} \\
  T\left(\frac{n}{2}\right) + C & \text{otherwise}
\end{cases} \]
How long does AVL insert take?

AVL insert time = BST insert time + time it takes to rebalance the tree
   = \(O(\log n)\) + time it takes to rebalance the tree

How long does rebalancing take?
- Assume we store in each node the height of its subtree.
- How long to find an unbalanced node:
  - Just go back up the tree from where we inserted. \(\leftarrow O(\log n)\)
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node. \(\leftarrow O(1)\)

AVL insert time = \(O(\log n) + O(\log n) + O(1) = O(\log n)\)
AVL wrap up

Pros:
- $O(\log n)$ worst case for find, insert, and delete operations.
- Reliable running times than regular BSTs (because trees are balanced)

Cons:
- Difficult to program & debug [but done once in a library!]
- (Slightly) more space than BSTs to store node heights.
Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)