Hash Tables

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials...
AVL Trees: Four cases to consider

<table>
<thead>
<tr>
<th>Insert location</th>
<th>Case</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree of left child of y (A)</td>
<td>Left line case</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>Right subtree of left child of y (B)</td>
<td>Left kink case</td>
<td>Double (left-right) rotation</td>
</tr>
<tr>
<td>Left subtree of right child of y (C)</td>
<td>Right kink case</td>
<td>Double (right-left) rotation</td>
</tr>
<tr>
<td>Right subtree of right child of y (D)</td>
<td>Right line case</td>
<td>Single left rotation</td>
</tr>
</tbody>
</table>
## AVL Trees: Four cases to consider

<table>
<thead>
<tr>
<th>Insert location</th>
<th>Case (also called as)</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree of left child of y (A)</td>
<td>Left line case (case 1)</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>Right subtree of left child of y (B)</td>
<td>Left kink case (case 2)</td>
<td>Double (left-right) rotation</td>
</tr>
<tr>
<td>Left subtree of right child of y (C)</td>
<td>Right kink case (case 3)</td>
<td>Double (right-left) rotation</td>
</tr>
<tr>
<td>Right subtree of right child of y (D)</td>
<td>Right line case (case 4)</td>
<td>Single left rotation</td>
</tr>
</tbody>
</table>

![AVL Tree Diagram]
AVL Tree: Practice. Insert(6)
Unbalanced
<table>
<thead>
<tr>
<th>Insert location</th>
<th>Case (also called as)</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree of left child of y (A)</td>
<td>Left line case (case 1)</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>Right subtree of left child of y (B)</td>
<td>Left kink case (case 2)</td>
<td>Double (left-right) rotation</td>
</tr>
<tr>
<td>Left subtree of right child of y (C)</td>
<td>Right kink case (case 3)</td>
<td>Double (right-left) rotation</td>
</tr>
<tr>
<td>Right subtree of right child of y (D)</td>
<td>Right line case (case 4)</td>
<td>Single left rotation</td>
</tr>
</tbody>
</table>

AVL Trees: Four cases to consider

![Diagram showing AVL tree with nodes A, B, C, D, and X, Y, Z arranged to demonstrate the four cases of insertions and their solutions.]
AVL Tree: Practice

Unbalanced
AVL Tree: Practice

Unbalanced
Worksheet Q1

```
insert(1)

Case: Line or Kink?
```
insert(1)

Case: Line or Kink?
Worksheet Q1

insert(1)

Case: Line or Kink?

insert(3)

Case: Line or Kink?
Worksheet Q1

insert(1)

Case: Line or Kink?

insert(3)

Case: Line or Kink?
AVL Tree insertions

1. Do a BST insert – insert a node as you would in a BST.
2. Check balance condition at each node in the path from the inserted node to the root.
3. If balance condition is not true at a node, identify the case.
4. Do the corresponding rotation for the case.

AVL Trees: Four cases to consider

<table>
<thead>
<tr>
<th>Insert location</th>
<th>Case (also called as)</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree of left child of y (A)</td>
<td>Left line case (case 1)</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>Right subtree of left child of y (B)</td>
<td>Left kink case (case 2)</td>
<td>Double (left-right) rotation</td>
</tr>
<tr>
<td>Left subtree of right child of y (C)</td>
<td>Right kink case (case 3)</td>
<td>Double (right-left) rotation</td>
</tr>
<tr>
<td>Right subtree of right child of y (D)</td>
<td>Right line case (case 4)</td>
<td>Single left rotation</td>
</tr>
</tbody>
</table>
Worksheet Q2

Draw the AVL tree that results from inserting the keys 1, 3, 7, 5, 6, 9 in that order into an initially empty AVL tree. (*Hint:* Drawing intermediate trees as you insert each key can help.)
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.
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node.

AVL insert time = \(O(\log n)\) + \(O(\log n)\) + \(O(1)\) = \(O(\log n)\)
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. $\Rightarrow O(\log n)$
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node. $\Rightarrow O(1)$

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)
Announcements

- HW3 out. Due this Friday (10/26) at Noon (not at the usual time 11:59pm)
- HW4 out later today (or latest by tomorrow morning). Due next Tuesday (10/30)

- Midterm coming up – Nov 2, 2:30-3:20pm, here in the class
- If you can’t take the midterm on Nov 2, let me know ASAP.
- Midterm practice material will be posted on the website tomorrow
- Midterm review next Wednesday
Hash tables
Revisiting Dictionaries

- data = (key, value)
- operations: put(key, value); get(key); remove(key)

- $O(n)$ with Arrays and Linked List
- $O(\log n)$ with BST and AVL trees.

- Can we do better? Can we do this in $O(1)$?
Motivation

Why we are so obsessed with making dictionaries fast?

Dictionaries are extremely most common data structures.
- Databases
- Network router tables
- Compilers and Interpreters
- Faster than $O(\log n)$ search in certain cases
- Data type in most high level programming languages
Question

How would you implement a dictionary such that dictionary operations are O(1)?

(Assume all keys are non-zero integers)
Question

How would you implement a dictionary such that dictionary operations are O(1)?
(Assume all keys are non-zero integers)

Idea: Create a giant array and use keys as indices
Question

How would you implement at dictionary such that dictionary operations are O(1)? (Assume all keys are non-zero integers)

Idea: Create a giant array and use keys as indices

Problems?
1. ?
2. ?
How would you implement a dictionary such that dictionary operations are $O(1)$?

(Assume all keys are non-zero integers)

**Idea:** Create a giant array and use keys as indices

Problems?
1. Can only work with integer keys?
2. Too much wasted space

**Idea 2:** Can we convert the key space into a smaller set that would take much less memory
Solve problem: Too much wasted space
Review: Integer remainder with %

The % operator computes the remainder from integer division.

\[
\begin{align*}
14 \mod 4 &= 2 \\
218 \mod 5 &= 3 \\
3 &
\end{align*}
\]

\[
\begin{array}{c}
3 \\
14 \\
\hline \\
12 \\
2 \\
\end{array}
\quad 
\begin{array}{c}
43 \\
218 \\
\hline \\
20 \\
18 \\
\hline \\
15 \\
3 \\
\end{array}
\]

Applications of % operator:
- Obtain last digit of a number: \(230857 \mod 10\) is 7
- See whether a number is odd: \(7 \mod 2\) is 1, \(42 \mod 2\) is 0
- Limit integers to specific range: \(8 \mod 12\) is 8, \(18 \mod 12\) is 6
public V get(int key) {
    // input validation
    return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key] = value;
}

public void remove(int key) {
    // input validation
    this.array[key] = null;
}
public V get(int key) {
    // input validation
    int newKey = key % this.array.length;
    return this.array[newKey].value;
}

public void put(int key, V value) {
    this.array[key % this.array.length] = value;
}

public void remove(int key) {
    // input validation
    int newKey = key % this.array.length;
    this.array[newKey] = null;
}
First Hash Function: % table size

<table>
<thead>
<tr>
<th>indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements</td>
<td>&quot;foo&quot;</td>
<td>&quot;biz&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;poo&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
put(0, "foo"); 0 % 10 = 0
put(5, "bar"); 5 % 10 = 5
put(11, "biz"); 11 % 10 = 1
put(18, "bop"); 18 % 10 = 8
put(20, "poo"); 20 % 10 = 0
```

Collision!