Announcements

• HW 1 due tonight, 11PM
• HW 2 out: due Friday, July 8\textsuperscript{th} at 11PM

• Lilian and Dan holding office hours today
Two cases to go

Unfortunately, single rotations are not enough for insertions in the **left-right** subtree or the **right-left** subtree.

Simple example: \texttt{insert(1), insert(6), insert(3)}

– First wrong idea: single rotation like we did for left-left.
Two cases to go
Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: \texttt{insert(1), insert(6), insert(3)}

Second wrong idea: single rotation on the child of the unbalanced node
Sometimes two wrongs make a right

- First idea violated the BST property
- Second idea didn’t fix balance
- But if we do both single rotations, starting with the second, it works! (And not just for this example.)
- Double rotation:
  1. Rotate problematic child and grandchild
  2. Then rotate between self and new child

Intuition: 3 must become root
The general right-left case

Rotation 1:
- \( b.left = c.right \)
- \( c.right = b \)
- \( a.right = c \)

Rotation 2:
- \( a.right = c.left \)
- \( c.left = a \)
- \( \text{root} = c \)
Comments

• Like in the left-left and right-right cases, the height of the subtree after rebalancing is the same as before the insert
  – So no ancestor in the tree will need rebalancing
• Does not have to be implemented as two rotations; can just do:

Easier to remember than you may think:

1) Move c to grandparent’s position

2) Put a, b, X, U, V, and Z in the only legal positions for a BST
The last case: left-right

- Mirror image of right-left
  - Again, no new concepts, only new code to write
Insert, summarized

• Insert as in a BST

• Check back up path for imbalance, which will be 1 of 4 cases:
  – Node’s left-left grandchild is too tall (left-left single rotation)
  – Node’s left-right grandchild is too tall (left-right double rotation)
  – Node’s right-left grandchild is too tall (right-left double rotation)
  – Node’s right-right grandchild is too tall (right-right double rotation)

• Only one case occurs because tree was balanced before insert

• After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  – So all ancestors are now balanced
Now efficiency

- Worst-case complexity of **find**: $O(\log n)$
  - Tree is balanced

- Worst-case complexity of **insert**: $O(\log n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there’s an $O(\log n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced

- Worst-case complexity of **buildTree**: $O(n \log n)$

Takes some more rotation action to handle **delete**...
Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of insert and delete

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. Most large searches are done in database-like systems on disk and use other structures (e.g., B-trees, a data structure in the text)
5. If *amortized* (later, I promise) logarithmic time is enough, use splay trees (also in text)
Dictionary Runtimes: More motivation

For a **dictionary** with $n$ key, value pairs

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Balanced</strong> tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Magic array</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Sufficient “magic”:

- Use key to compute array index for an item in $O(1)$ time [doable]
- Have a different index for every item [magic]
Motivating Hash Tables

For a dictionary with \( n \) key, value pairs

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>( O(1)  )</td>
<td>( O(n)   )</td>
<td>( O(n)   )</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>( O(1)  )</td>
<td>( O(n)   )</td>
<td>( O(n)   )</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>( O(n)  )</td>
<td>( O(n)   )</td>
<td>( O(n)   )</td>
</tr>
<tr>
<td>Sorted array</td>
<td>( O(n)  )</td>
<td>( O(\log n) )</td>
<td>( O(n)   )</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Magic array</td>
<td>( O(1)  )</td>
<td>( O(1)   )</td>
<td>( O(1)   )</td>
</tr>
</tbody>
</table>

Sufficient “magic”:

- Use key to compute array index for an item in \( O(1) \) time [doable]
- Have a different index for every item [magic]
Motivating Hash Tables

• Let’s say you are tasked with counting the frequency of integers in a text file. You are guaranteed that only the integers 0 through 100 will occur:

For example: 5, 7, 8, 9, 9, 5, 0, 0, 1, 12
Result: 0 ↦ 2  1 ↦ 1  5 ↦ 2  7 ↦ 1  8 ↦ 1  9 ↦ 2

What structure is appropriate?
  Tree?
  List?
  Array?

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th></th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Motivating Hash Tables

Now what if we want to associate name to phone number?

Suppose keys are first, last names
   – how big is the key space?

Maybe we only care about students
Hash Tables

• Aim for constant-time (i.e., $O(1)$) **find**, **insert**, and **delete**
  – “On average” under some often-reasonable **assumptions**

• A hash table is an array of some fixed size

• Basic idea:

  ![Diagram of hash table and hash function]

  - **Key space** (e.g., integers, strings)
  - **Table size** (e.g., TableSize - 1)
  - **Hash function**: $\text{index} = h(\text{key})$
Hash Tables vs. Balanced Trees

• In terms of a Dictionary ADT for just insert, find, delete, hash tables and balanced trees are just different data structures
  – Hash tables $O(1)$ on average (assuming we follow good practices)
  – Balanced trees $O(\log n)$ worst-case

• Constant-time is better, right?
  – Yes, but you need “hashing to behave” (must avoid collisions)
  – Yes, but $\text{findMin}$, $\text{findMax}$, $\text{predecessor}$, and $\text{successor}$ go from $O(\log n)$ to $O(n)$, $\text{printSorted}$ from $O(n)$ to $O(n \log n)$
    • Why your textbook considers this to be a different ADT
Hash Tables

• There are $m$ possible keys ($m$ typically large, even infinite)
• We expect our table to have only $n$ items
• $n$ is much less than $m$ (often written $n << m$)

Many dictionaries have this property

– Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
– Database: All possible student names vs. students enrolled
– AI: All possible chess-board configurations vs. those considered by the current player
– …
Hash functions

An ideal hash function:
• Fast to compute
• “Rarely” hashes two “used” keys to the same index
  – Often impossible in theory but easy in practice
  – Will handle collisions later

hash function: \[ \text{index} = h(\text{key}) \]

key space (e.g., integers, strings)

hash table
0
...
TableSize – 1
Simple Integer Hash Functions

- key space $K = \text{integers}$
- TableSize = 7
- $h(K) = K \mod 7$
- **Insert**: 7, 18, 41
Simple Integer Hash Functions

- key space \( K = \text{integers} \)
- TableSize = 10

- \( h(K) = ?? \)

- **Insert**: 7, 18, 41, 34
  - What happens when we insert 44?
Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

\[ h(K) = \text{function}(K) \% \text{TableSize} \]

(In the previous examples, function(K) = K.)

Useful properties of mod:

- \((a + b) \% c = [(a \% c) + (b \% c)] \% c\)
- \((a \ b) \% c = [(a \% c) (b \% c)] \% c\)
- \(a \% c = b \% c \rightarrow (a - b) \% c = 0\)
Designing Hash Functions

Often based on **modular hashing**: 

\[ h(K) = f(K) \% P \]

P is typically the TableSize

P is often chosen to be prime:

- Reduces likelihood of collisions due to patterns in data
- Is useful for guarantees on certain hashing strategies (as we’ll see)

Equivalent objects MUST hash to the same location
Some String Hash Functions

key space = strings

\[ K = s_0 \ s_1 \ s_2 \ ... \ s_{m-1} \] (where \( s_i \) are chars: \( s_i \in [0, 128] \))

1. \[ h(K) = s_0 \% \text{TableSize} \]

2. \[ h(K) = \left( \sum_{i=0}^{m-1} s_i \right) \% \text{TableSize} \]

3. \[ h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 37^i \right) \% \text{TableSize} \]

H(“batman”) = H(“ballgame”)

H(“spot”) = H(“pots”)

H(“spot”) = H(“pots”)

UW CSE 332, Spring 2016
What to hash?

We will focus on the two most common things to hash: \textit{ints} and \textit{strings}

- For objects with several fields, usually best to have most of the “identifying fields” contribute to the hash to avoid collisions

- Example:
  
  class Person {
    String first; String middle; String last;
    Date birthdate;
  }

- An inherent trade-off: hashing-time vs. collision-avoidance
  
  - Bad idea(?): Use only first name
  - Good idea(?): Use only middle initial? Combination of fields?
  - Admittedly, what-to-hash-with is often unprincipled 😞
Deep Breath

• Recap
Hash Tables: Review

• Aim for constant-time (i.e., $O(1)$) **find, insert, and delete**
  – “On average” under some reasonable assumptions

• A hash table is an array of some fixed size
  – But growable as we’ll see

```
    hash table
      0
      ...

  client: E
  hash table library
  int: table-index
  collision?
  collision resolution

  TableSize −1
```
Collision resolution

Collision:
When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution
– Ideas?
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and \textbf{TableSize} = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing
and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing
and TableSize = 10
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}}$$

← number of elements

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:

• Each “unsuccessful” find compares against $\lambda$ items

So we like to keep $\lambda$ fairly low (e.g., 1 or 1.5 or 2) for chaining