Lecture 10: Introduction to Hash Tables
Administrivia

HW 2 part 1 grades went out

HW 2 part 2 due Friday, including part 1 regrade

HW 3 partner form due Friday by 12:00pm

Midterm Friday February 15th in class
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. \( \text{O}(\log n) \)
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node. \( \text{O}(1) \)

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)
Hashing
### Review: Dictionaries

#### Dictionary ADT

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of items &amp; keys</td>
<td>put(key, item) add item to collection indexed with key</td>
</tr>
<tr>
<td>Count of items</td>
<td>get(key) return item associated with key</td>
</tr>
<tr>
<td></td>
<td>containsKey(key) return if key already in use</td>
</tr>
<tr>
<td></td>
<td>remove(key) remove item and associated key</td>
</tr>
<tr>
<td></td>
<td>size() return count of items</td>
</tr>
</tbody>
</table>

#### ArrayDictionary\(<K, V>\)

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair(&lt;K, V&gt;)[] data</td>
<td>put create new pair, add to next available spot, grow array if necessary</td>
</tr>
<tr>
<td></td>
<td>get scan all pairs looking for given key, return associated item if found</td>
</tr>
<tr>
<td></td>
<td>containsKey scan all pairs, return if key is found</td>
</tr>
<tr>
<td></td>
<td>remove scan all pairs, replace pair to be removed with last pair in collection</td>
</tr>
<tr>
<td></td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>

#### LinkedDictionary\(<K, V>\)

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>put if key is unused, create new pair, add to front of list, else replace with new value</td>
</tr>
<tr>
<td>size</td>
<td>get scan all pairs looking for given key, return associated item if found</td>
</tr>
<tr>
<td></td>
<td>containsKey scan all pairs, return if key is found</td>
</tr>
<tr>
<td></td>
<td>remove scan all pairs, skip pair to be removed</td>
</tr>
<tr>
<td></td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>

#### TreeDictionary\(<K, V>\)

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>overallRoot</td>
<td>put if key is unused, create new pair, place in BST order, rotate to maintain balance</td>
</tr>
<tr>
<td>size</td>
<td>get traverse through tree using BST property, return item if found</td>
</tr>
<tr>
<td></td>
<td>containsKey traverse through tree using BST property, return if key is found</td>
</tr>
<tr>
<td></td>
<td>remove traverse through tree using BST property, replace or nullify as appropriate</td>
</tr>
<tr>
<td></td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>
### Review: Dictionaries

#### Why are we so obsessed with Dictionaries?

It’s all about data baby!

When dealing with data:
- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

---

### Dictionary ADT

<table>
<thead>
<tr>
<th>state</th>
<th>Set of items &amp; keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count of items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>behavior</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, item)</td>
<td>add item to collection indexed with key</td>
</tr>
<tr>
<td>get(key)</td>
<td>return item associated with key</td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>return if key already in use</td>
</tr>
<tr>
<td>remove(key)</td>
<td>remove item and associated key</td>
</tr>
<tr>
<td>size()</td>
<td>return count of items</td>
</tr>
</tbody>
</table>

---

### Operations vs. Data Structures

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why are we so obsessed with Dictionaries?  

**It’s all about data baby!**

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

### Dictionary ADT

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
</tbody>
</table>

SUPER common in comp sci:

- Databases
- Network router tables
- Compilers and Interpreters
Can we do better?

What if we knew exactly where to find our data?

Implement a dictionary that accepts only integer keys between 0 and some value k

- -> Leverage Array Indices!

“Direct address map”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key,value)</td>
<td>best – O(1)</td>
</tr>
<tr>
<td></td>
<td>average – O(1)</td>
</tr>
<tr>
<td></td>
<td>worst – O(1)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best – O(1)</td>
</tr>
<tr>
<td></td>
<td>average – O(1)</td>
</tr>
<tr>
<td></td>
<td>worst – O(1)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best – O(1)</td>
</tr>
<tr>
<td></td>
<td>average – O(1)</td>
</tr>
<tr>
<td></td>
<td>worst – O(1)</td>
</tr>
</tbody>
</table>

DirectAccessMap<Integer, V>

- state
  Data[]
  size
- behavior
  put put item at given index
  get get item at given index
  containsKey if data[] null at index, return false, return true otherwise
  remove nullify element at index
  size return count of items in dictionary
Implement Direct Access Map

```java
public V get(int key) {
    this.ensureIndexNotNull(key);
    return this.array[key].value;
}

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

public void remove(int key) {
    this.ensureIndexNotNull(key);
    this.array[key] = null;
}
```

DirectAccessMap<Integer, V>

**state**
- Data[]
- size

**behavior**
- put: put item at given index
- get: get item at given index
- containsKey: if data[] null at index, return false, return true otherwise
- remove: nullify element at index
- size: return count of items in dictionary
Can we do this for any integer?

**Idea 1:**
Create a GIANT array with every possible integer as an index

Problems:
- Can we allocate an array big enough?
- Super wasteful

**Idea 2:**
Create a smaller array, but create a way to translate given integer keys into available indices

Problem:
- How can we pick a good translation?
Review: Integer remainder with % “mod”

The % operator computes the remainder from integer division.

\[
\begin{align*}
14 \% 4 & \quad \text{is} \quad 2 \\
218 \% 5 & \quad \text{is} \quad 3 \\
4 & \quad 14 \\
\underline{12} & \quad 43 \\
2 & \quad 20 \\
15 & \quad 18 \\
\end{align*}
\]

Applications of % operator:

- Obtain last digit of a number: \(230857 \% 10 \) is 7
- See whether a number is odd: \(7 \% 2 \) is 1, \(42 \% 2 \) is 0
- Limit integers to specific range: \(8 \% 12 \) is 8, \(18 \% 12 \) is 6

For more review/practice, check out [https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic](https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic)
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>“foo”</td>
<td>“biz”</td>
<td></td>
<td></td>
<td></td>
<td>“bar”</td>
<td></td>
<td></td>
<td></td>
<td>“bop”</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
public V get(int key) {
    int newKey = getKey(key);
    this.ensureIndexNotNull(key);
    return this.data[key].value;
}

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

public void remove(int key) {
    int newKey = getKey(key);
    this.ensureIndexNotNull(key);
    this.data[key] = null;
}

public int getKey(int value) {
    return value % this.data.length;
}
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>element s</td>
<td>&quot;poo&quot;</td>
<td>&quot;biz&quot;</td>
<td></td>
<td></td>
<td>&quot;bar&quot;</td>
<td></td>
<td></td>
<td></td>
<td>&quot;bop&quot;</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!
Hash Obsession: Collisions

When multiple keys translate to the same location of the array

The fewer the collisions, the better the runtime!
Strategies to handle hash collision
Strategies to handle hash collision

There are multiple strategies. In this class, we’ll cover the following three:

1. Separate chaining
2. Open addressing
   - Linear probing
   - Quadratic probing
3. Double hashing
Handling Collisions

Solution 1: Chaining

Each space holds a “bucket” that can store multiple values. Bucket is often implemented with a LinkedList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array w/ indices as keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key,value)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + (\lambda))</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + (\lambda))</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best: O(1)</td>
</tr>
<tr>
<td></td>
<td>average: O(1 + (\lambda))</td>
</tr>
<tr>
<td></td>
<td>worst: O(n)</td>
</tr>
</tbody>
</table>

Average Case:
Depends on average number of elements per chain

Load Factor \(\lambda\)
If \(n\) is the total number of key-value pairs
Let \(c\) be the capacity of array
Load Factor \(\lambda = \frac{n}{c}\)
Practice

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a) (5,b) (11,a) (7,d) (12,e) (17,f) (1,g) (25,h)
Can we do better?

**Idea 1: Take in better keys**
- Can’t do anything about that right now

**Idea 2: Optimize the bucket**
- Use an AVL tree instead of a Linked List
- Java starts off as a linked list then converts to AVL tree when collisions get large

**Idea 3: Modify the array’s internal capacity**
- When load factor gets too high, resize array
  - Double size of array
  - Increase array size to next prime number that’s roughly double the array size
    - Prime numbers reduce collisions when using % because of divisors
  - Resize when $\lambda \approx 1.0$
  - When you resize, you have to rehash
What about non integer keys?

Hash Function

An algorithm that maps a given key to an integer representing the index in the array for where to store the associated value

Goals

Avoid collisions
- The more collisions, the further we move away from O(1)
- Produce a wide range of indices

Uniform distribution of outputs
- Optimize for memory usage

Low computational costs
- Hash function is called every time we want to interact with the data
How to Hash non Integer Keys

Implementation 1: Simple aspect of values
public int hashCode(String input) {
    return input.length();
}

Implementation 2: More aspects of value
public int hashCode(String input) {
    int output = 0;
    for(char c : input) {
        out += (int)c;
    }
    return output;
}

Implementation 3: Multiple aspects of value + math!
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}
Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```java
public int hashCode(String input) {
    return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

Java and Hash Functions

Object class includes default functionality:
- equals
- hashCode

If you want to implement your own hashCode you MUST:
- Override BOTH hashCode() and equals()
- If a.equals(b) is true then a.hashCode() == b.hashCode() MUST also be true