Implementing Hash and AVL

Data Structures and Algorithms

Warm Up

Announcements

1. Go look at your HW 1 scores, seems a lot are missing

2. Look at your HW 2 scores
   - If you got 0/5 for check style, you can get those points back
   - If you got 0/12 for delete tests, your tests didn't pass on working input
   - Regrade policy: when resubmitted you can earn up to ½ missed points back

3. Must use same partners for part 2 of project
   - Can pick new partners for next project
   - EXTREMELY HIGH overlap between those working alone and late submitted projects

4. Kasey is presenting the "No BS CS Career Talk" for 14X on Thursday April 19th 4:30-5:20 in Gug 220
   - It's a good time, come hang out

Coming Up

Monday	Wednesday	Thursday	Friday
4/16 Lecture: Open Addressing in Hash Tables	4/18 Lecture: Implementing AVL Trees and Hash Tables	4/19 Section: AVL Trees and Hash Tables	4/20 Lecture: How Memory Works HW2 PT2 due HW3: Midterm Review assigned
4/23 Lecture: B-Trees	4/25 Lecture: Midterm Review	4/26 Section: Midterm Review	4/27 Midterm HW3: Midterm Review due

TA Lead Review Session: TBA

What's going to be on the Midterm?

ADTs and data structures

- Difference between an ADT and a data structure.
- Stacks, queues, lists, dictionaries: common implementations, runtimes, and when to use them.
- Iterators: what they are, how to implement basic ones (e.g. for array lists and linked lists).

Asymptotic analysis

- Big-O, Big-Omega, and Big-Theta.
- Finding c and n0 to show that one function is in Big-O, Big-Omega, or Big-Theta of another
- Modeling runtime of a piece of code as a function possibly including a summation or a recurrence.
- Understand the difference between best-case, average-case, and worst-case runtime.

Trees

- How to implement and manipulate trees including Binary Search and AVL types
- Runtimes for tree operations.
- Performing AVL rotations when inserting values

Hash tables

- Closed vs open addressing.
- Collision resolution: separate chaining, linear probing, quadratic probing, double hashing.
- Basics of good hash function design.
- Load factor.
- Runtimes (best, average, and worst-case).

Testing

- How to construct different test cases
- Reading and evaluating code to debug

NOT on the exam

- Java generics and Java interfaces
- JUnit
- Java syntax
- Finding the closed form of summations and recurrences

Implementing a Dictionary

Dictionary ADT

state

Set of Key, Value pairs

- Keys must be unique!
- No required order

Count of data pairs

behavior

Add pair to collection

Get value for given key

Change value for given key

Remove data pair from collection

public interface Dictionary {

state

unspecified

behavior

```
void put(key, value)

value get(key)

void set(key, value)

void remove(key)
```

HashMap<K, V>

state

```
Data[]
  - Pair<K, V>[]
  - LinkedList<E>[]
size
```

behavior

```
put() pair into array
based on hash
- Resize when appropriate

get() value from array index
based given key's hash

set() update value in pair
for given key's hash to
array index

remove() take data out
of array
```

TreeMap<K, V>

state

```
overallRoot<K,V>
```

behavior

```
put() add node for new
pair in correct location
- Balance when appropriate

get() value based on node
location in tree

set() update value in pair
for given key

remove() delete given node
- replace with appropriate
existing node
```

Implementing Hash Map

HashMap<K, V>

state

```
LinkedList<E>[]
size
```

behavior

```
put() pair into array
based on hash
- Resize when appropriate

get() value from array index
based given key's hash

set() update value in pair
for given key's hash to
array index

remove() take data out
of array
```

LinkedList<E>

state

```
ListNode<K, V> front
```

behavior

```
Add() add a new node that
stores Key and Value to list

get() return value from node
with given key

set() changes value in
node with given key

remove() deletes node with
given key from list

Contains() is the given key
stored in list

iterator() returns an
iterator to move over list
```

ListNode<K, V>

state

```
K key
V value
ListNode<K, V> next
```

behavior

```
Construct a new Node
```

Implementing a Hash Map

```
v get(k key) {
   bucketAddress = get hash for key % table size
   bucketList = data[bucketAddress]
   loop (bucketList) {
      if (this node's key is what I am looking for)
         return this node's value
   }
   return not found :(
}
```

```
void put(k key, v value) {
   create new Node
   bucketAddress = get hash for key % table size
   bucketList = data[bucketAddress]
   loop(bucketList)
      if (this node's key is what I am trying to add)
         replace this node with new pair
         stop work
   if (load factor is about 1)
      increase array capacity to next prime number
      rehash existing values into new array
      add node to bucket
   update size
}
```

HashMap<K, V>

state
```
LinkedList<E>[]
size
```
behavior
```
void put(key, value)
value get(key)
void set(key, value)
void remove(key)
```

LinkedList<E>

state
```
ListNode<K, V> front
```
behavior
```
void add(key, value)
value get(key)
void set(key, value)
void remove(key)
boolean contains(key)
iterator<E> iterator()
```

ListNode<K, V>

state
```
K key
V value
ListNode<K, V> next
```
behavior
```
Construct a new Node
```

Implementing Tree Map

TreeMap<K, V>

state

overallRoot<K,V>

behavior

put() add node for new pair in correct location
- Balance when appropriate

get() value based on node location in tree

set() update value in pair for given key

remove() delete given node
- replace with appropriate existing node

ListNode<K, V>

state

K key

V value

ListNode<K, V> left

ListNode<K, V> right

int height

behavior

Construct a new Node

Implementing Tree Map

```
v get(k key) {
   start at top of tree
}
ListNode<K, V> getHelper(key, Node) {
   if(node is null)
      data isn't in collection
   if data at current node > what I'm looking for
      go left
   if data at current node < what I'm looking for
      go right
   else
      found it!
}
```

TreeMap<K, V>

state

```
overallRoot<K,V>
```

behavior

```
void put(key, value)
value get(key)
void set(key, value)
void remove(key)
```

ListNode<K, V>

state

```
K key
V value
ListNode<K, V> left
ListNode<K, V> right
int height
```

behavior

```
Construct a new Node
```