WEEK9B

JAVA II — BARRY

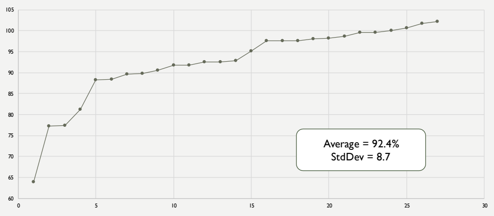
TOPICS

- Exam #2 debrief
- Project Q&A
- Hashing

EXAM #2 DEBRIEF

RESULTS, STATS, AND NOTES

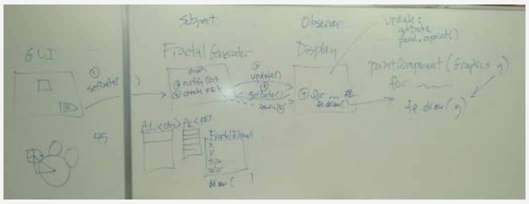
EXAM #2 STATS



PROJECT ARCHITECTURE/ COMMUNICATION REVIEW

GETTING CLEAR ABOUT HOW TO PUT TOGETHER THE FRACTAL PROJECT

IN-CLASS WHITEBOARD WORK



HASHING

SEARCHING FOR VALUES AT NEAR CONSTANT (O(1)) TIME

WHAT WE'RE LOOKING FOR: A WAY TO STORE INTEGERS

We need an approach that will provide…

- Ultra-fast searching (constant time (O(1)), or very near it)
- Fast operations including add and delete
- Scalability: ability to grow while maintaining required properties

What are some potential approaches to this?

CANDIDATE APPROACH #1: UNSORTED ARRAY OR ARRAYLIST

Approach

- Store values in an unsorted array or ArrayList
- Easily add values in at the end of the used space
- Instead of delete/shift, we could mark deleted items and compress periodically
- Periodically we'd need to grow the array (ArrayList does this)

Why it won't work: searching would be O(n)

CANDIDATE APPROACH #2: SORTED ARRAY OR ARRAYLIST

Approach

- Store values in an sorted array or ArrayList
- Add values in order
- Periodically we'd need to grow the array (ArrayList does this)

Why it won't work: searching would be $O(\log_2 n)$, additions/deletions would require shifting

CANDIDATE APPROACH #3: VALUE/INDEX PAIRING

Approach

- Store value at the index indicated by its value, e.g., value 37 goes at index 37

Why it won't work: to cover a large range of numbers, the array would have to be of a ridiculous size; it would be overly sparse, too

HASHING & HASH FUNCTIONS

Hashing provides a way to map a large range of numbers into indexes into a relatively small array, while maintaining all the desired efficiencies

A **Hash Function** does this mapping, turning an element value into an index at which to store the value; this is typically done with modulus math:

```
private int hashFunction(int value) {
    return Math.abs(value) % elementData.length;
}
```

HASH TABLES

A **Hash Table** is an array that stores its elements in indexes produced by a hash function

0	1	2	3	4	5	6
581		2		8768		

```
Value:    23  →   23 % 7   → 2
Value:  -581  →  581 % 7   → 0
Value:  8768  → 8768 % 7   → 4
```

COLLISIONS

What happens when we get the same hash for a different value?

A **Collision** occurs when two or more element values generate the same results from hash function

0	1	2	3	4	5	6
581		2		8768		

?

Value: 459 → 459 % 7 → 4

We can *reduce* clustering/collisions by making the array's size a prime number, but that doesn't *eliminate* collisions

COLLISION SOLUTION #1: PROBING

Probing: resolving hash collisions by placement elements at index other than their preferred ones

Linear Probing looks at the next available spot

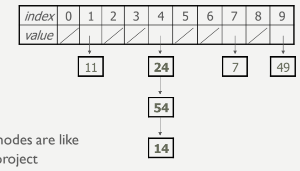
0	1	2	3	4	5	6
581		2		8768	459	

Value: 459 → 459 % 7 → 4

This means searching can be sub-optimal;contains( ) would need more logic

COLLISION SOLUTION #2: SEPARATE CHAINING

Separate Chaining: resolving hash collisions by having each index of the table store a list of values (in nodes)



Does this look familiar? These nodes are like our Dup Nodes from our BST project

HASHING ISSUE: PERFORMANCE AS SIZE APPROACHES CAPACITY

Regardless of how we solve our collision problem, performance starts to degrade as the table gets full

To solve this, we can't just create a larger array and copy data over; element values will often not have the same hash results in the larger table

Rehashing: resizing a table to increase capacity and maintain efficiency. It involves expanding the table and adding elements again via hash function

Typically we look at the table's load factor (size relative to capacity), and rehash when it exceeds some predetermined threshold (e.g., Java HashSet's 75%)

OBJECT HASHING

If you're providing an equals method, you should provide a hashCode method based on object state

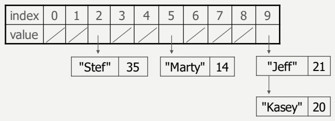
Two object you consider equal should produce the same hash results; ideally different objects would generally produce different ones, e.g.,

```
public class Point {
...
    public int hashCode( ) {
        return 31337 * y + x;
    }
}
```

HASH MAPS

We've so far discussed **HashSet** implementation

If you want to build a **HashMap**, just create a HashEntry class with a key and a value; hash the key and store the HashEntry object at the resulting index location. Usually we'd use generics so the types are flexible



END