Finish Hash Tables
Review
Growing Probe Hashtables

- O(1) get and put when lightly loaded so want to keep the table lightly loaded.
- Need to add a private “Grow” function to put
  - Grow:
    - make a new array bigger than old array (2x)
    - copy each item from old array into new array (into the correct location)
    - forget old array
Growing Hashtables

```java
public class ProbeHTInc<K, V> implements Map151Interface<K, V> {

    private Pair<K, V>[] backingArray;

    private int hash(K key) {
        return Math.abs(key.hashCode()) % backingArray.length;
    }

    private void grow() {
        // write me
    }
}
```
Probing Distance (Summary)

- Given a hash value $h(x)$, linear probing generates $h(x)$, $h(x) + 1$, $h(x) + 2$, ...
  - Primary clustering – the bigger the cluster gets, the faster it grows

- Quadratic probing –
  $h(x)$, $h(x) + 1$, $h(x) + 4$, $h(x) + 9$, ...
  - Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

- Double hashing
  - has neither primary nor secondary clustering
Performance Analysis for probing

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide

- The load factor $\alpha$ affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1 - \alpha}$

- Expected time of all operations is $O(1)$ provided $\alpha$ is not close to 1

- Rule of thumb:
  - small hashtables -- $\alpha < 0.5$
  - larger hashtables -- $\alpha < 0.66$
Removing Items

• In separate chaining just remove.
• Probing: cannot simply delete as positions are dependent on what was there are time inserted

• So rather than set position empty on delete, replace item with "tombstone"
Probing vs Chaining

• Probing is significantly faster in practice
• locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a list
• Efficient probing requires tombstoning
  • de-tombstoning??
Sample Hashtable use

• Problem I have a random string generator and I want to see how "random" it is.
• Concept, generate lots of random strings, put them in hashtable, find out how many unique strings I actually saw
• Complexity Analysis
  • Generate a string: O(1)
  • Add N strings into hashtable: O(N)
  • Count number of things in hashtable: O(1)
public class UseHT {
    private SepChainHT<String, Integer> hashT;
    private Random rand;

    public UseHT() {
        hashT = new SepChainHT<>(10001);
        rand = new Random();
    }

    private String randomString(int len) {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < len; i++) {
            sb.append('a' + rand.nextInt(26));
        }
        return sb.toString();
    }
}
public void querrier() {
    for (int i = 2; i < 5; i++) { // length of random string
        for (int j = 1; j < 100001; j = j * 10) { // number of strings to make
            hashT = new SepChainHT<>(j*2);
            for (int k = 0; k < j; k++) { // actually to the work specified by the other loops
                String s = randomString(i);
                hashT.put(s, 1);
            }
        }
    }
}
System.out.println("Random String length: "+i+
" number generated " + j+
" number unique "+hashT.size());
if (!hashT.containsKey(s)) {
}
Java

- Classes and Inheritance
  - Overloading
    - method with same name but different parameters
      - equals(Object ob) vs equals(String st)
  - Overriding of methods
    - same name, same args but in extending class
    - marked by @Override
- Exceptions  Chapter: Interlude 2,3
- UML and Java Interfaces  Chapter: Prelude
- Generics  Chapter Interlude 1,8
- Inner classes
Data Structures

- Arrays
- Bags **Chapter 1,2**
- ArrayList **Chapter 10**
- Maps **Chapter 20,21**
  - key-value pairs
- Hashtables **Chapter 22,23**
Theory

- Complexity Analysis — Big-O — Chapter 4
  - drop constants
  - focus on dominant term
  - always look at worst case
  - Look for loops
    - loops incrementing using + or -: O(n)
    - loops incrementing using * or /: O(lg n)
    - loops inside loops (inside loops): multiply
    - loops next to loops: add
- Modularity, Abstraction and Encapsulation — Chapter: Prelude
Study suggestion

• Do not just read notes / book.
• Instead, be active
  • Read notes / book describing one algorithm (or data structure)
  • Write code for that algorithm
  • Do complexity analysis for that algorithm
Practice

• Write a class for Car
  • it should have several instance variables (eg color, manufacturer, size of engine)
    • write an equals method for Car
    • write a toString method for Car
  • Create several instances of Car and add them to a List151Impl (or ArrayList)
  • Write a user interaction that allows people to ask if a car is in your list.