Maps
Hash Tables

CS206
March 5
Java Inner Classes

- A class defined WITHIN another class
- Cannot be public (so private or protected)
- Reason
  - inner classes allow a grouping of related data — just like regular classes
  - BUT, it is not a grouping that anyone outside the class knows about
- So, class writer can change it as needed
Zip Code (assignment 1)

- The Place class could / should have been a private class.
- no users require access to it
- Its presence slightly violates encapsulation
  - I suspect that somehow, internally, places is probably using the Place class to store data.
- (Alternative, a 2-d array where the second dimension is 3 with
  - 0==zip
  - 1==cite
  - 2==state
Internal classes

• Are real classes
  • they could inherit for other external classes or other internal classes
• variables are “public” to the containing class
  • they are only “public” to the containing class so no encapsulation violation
public class OutCl {

    private class InnCl {
        private int value1;
        private String value2;
        public InnCl(int v1, String v2) {
            this.value1 = v1;
            this.value2 = v2;
        }
        public String toString() {
            return value1 + " " + value2;
        }
    }

    public void worker() {
        InnCl icl1 = new InnCl(1, "Bob");
        InnCl icl2 = new InnCl(2, "Carol");
        icl1.value1 = 3;
        icl2.value2 = "Alice";
        System.out.println(icl1 + "\n" + icl2);
    }

    public static void main(String[] args) {
        OutCl ocl = new OutCl();
        ocl.worker();
    }
}
public class OutCLGen {

    /**
     * The inner class, Generically
     */
    private class InnCl<Y,Z> {
        private Y value1; // a value
        private Z value2; // another value

        public InnCl(Y v1, Z v2) {
            this.value1 = v1;
            this.value2 = v2;
        }

        public String toString() {
            return value1 + " " + value2;
        }
    }

    public void worker() {
        InnCl<Integer, String> icl1 = new InnCl<>(1, "Bob");
        InnCl<String, String> icl2 = new InnCl<>("Alice", "Carol");
        icl1.value1 = 3;
        icl2.value2 = "Alice"
        System.out.println(icl1 + "\n" + icl2);
    }

    public static void main(String[] args) {
        OutCl ocl = new OutCl();
        ocl.worker();
    }
}
Complexity Analysis for ArrayList

- In groups discuss
- What is the O() for each of these
- size()
- get()
- add()
- remove()
Maps

- A searchable collection of key-value pairs
- A lot of this class will be about key value pairs
  - A lot of life is about key value pairs
    - SSN, tax history
    - BMID no, student record
    - .....
- Multiple entries with the same key are not allowed
- Also known as dictionary (python), associative array (perl)
Map Interface

- https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

```java
class MapInterface {
    public void put(K key, V val);
    public V get(K key);
    public boolean containsKey(K key);
    public int size();
    public Set<K> keySet();
}
```
public class Map206<K,V> {

    private ArrayList<Pair<K,V>> underlying = new ArrayList<>();

    private class Pair<L,W> {
        public L ky;
        public W vl;
        Pair(L key, W value) {
            ky=key;
            vl=value;
        }
    }
}
public boolean containsKey(K key) {
    return null != iContainsKey(key);
}

private Pair<K,V> iContainsKey(K ky) {
    for (Pair<K,V> pair : underlying) {
        if (pair.key.equals(ky)) {
            return pair;
        }
    }
    return null;
}

/**
 * The number of items in the map
 * @return The number of items in the map
 */
public int size() {
    return underlying.size();
}

public void put(K key, V val) {
    Pair<K,V> pair = iContainsKey(key);
    if (pair==null) {
        Pair<K,V> np = new Pair<>(key, val);
        underlying.add(np);
    } else {
        pair.value=val;
    }
}

public V get(K key) {
    Pair<K,V> pair = iContainsKey(key);
    if (pair!=null)
        return pair.value;
    return null;
}
Maps - time complexity

• Put
  • need to search the existing items (iContainsKey)
  • so time to add 1 item in $O(n)$
  • BUT if you are adding $n$ items the you are doing an $O(n)$ process $n$ times so your time is $O(n^2)$

• Get
  • similar analysis so $O(n^2)$ to do $n$ gets
  • So maps get slow when $n$ gets large.
private Map206<String, Integer> counts = new Map206<>();

void countFile(String filename) {
  try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
    String line;
    while (null != (line = br.readLine())) { // read line and test if there is a line to read
      line = line.toLowerCase().replace(".", ").replace("", ").replace("?", ").replace("!", ").replace("-", "\s+"");
      String[] ss = line.split("\s+"); // split the line by spaces
      for (String token : ss) {
        if (token.length() > 0) {
          int tokencount = 0;
          if (counts.containsKey(token)) {
            tokencount = counts.get(token);
          }
          counts.put(token, tokencount+1);
        }
      }
    }
  } catch (FileNotFoundException e) {
    System.err.println("Error in opening the file:" + filename);
    System.exit(1);
  }
  catch (IOException ioe) {
    System.err.println("Error reading file " + ioe);
    System.exit(1);
  }
}

two getCount() {
  return counts.size();
}

public static void main(String[] args) {
  WordCounter wc = new WordCounter();
  wc.countFile("janeausten.txt");
  System.out.println("Words " + wc.getCount());
}
Hash Tables

• A hash table is a form of a map that has better time complexity

• A hash table consists of
  • an array of size $N$
    - an associated hash function $h$ that maps keys to integers in $[0, N-1]$
    - item $(k, v)$ is stored at index $h(k)$
  • $h(x) = x \% N$ is such a function for integers
Simple Hashtable Implementation

```java
public class SimpleHT {
    private class Pair {
        // the key. Once set it cannot be changed
        public final Integer key;
        // the value
        public String value;

        // Create a key value pair.
        Pair(Integer ky, String val) {
            key = ky;
            value = val;
        }
    }

    private Pair[] backingArray;

    public SimpleHT() {
        backingArray = new Pair[4];
    }

    private int h(int k) {
        return k%4;
    }

    public void put(Integer key, String value) {
        backingArray[h(key)] = new Pair(key, value);
    }

    public String get(Integer key) {
        return backingArray[h(key)].value;
    }
}
```
HashTable Example

```java
public static void main(String[] args) {
    SimpleHT sht = new SimpleHT();
    for (int i=0; i<10; i++) {
        System.out.println("adding item with key=" + i + " value=" + String.format("%c", 'a'+i));
        sht.put(i, String.format("%c", 'a'+i));
    }
    for (int i=0; i<10; i++)
        System.out.println("getting key="+i+" value="+sht.get(i));
}
```

Two problems:
1. a poor hashing function.
2. Storing more than there is room for

adding item with key=0 value=a
adding item with key=1 value=b
adding item with key=2 value=c
adding item with key=3 value=d
adding item with key=4 value=e
adding item with key=5 value=f
adding item with key=6 value=g
adding item with key=7 value=h
adding item with key=8 value=i
adding item with key=9 value=j
getting key=0 value=i
generating key=1 value=j
generating key=2 value=g
generating key=3 value=h
generating key=4 value=i
generating key=5 value=j
generating key=6 value=g
generating key=7 value=h
generating key=8 value=i
generating key=9 value=j
Hash Functions

- The goal is to “disperse” the keys in an appropriately random way
- A hash function is usually specified as the composition of two functions:
  - hash code: key $\rightarrow$ integers
  - compression: integers $\rightarrow [0, N-1]$

see SepChainHT.java
Hash Codes

- Polynomial accumulation: partition bits of key into a sequence of components of fixed length $a_0a_1\ldots a_{n-1}$

- Evaluate the polynomial
  
  $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
Polynomial accumulation on Strings

Recommended by textbook

Handles really large numbers

Array storing the hashtable

static int POLY_MULT=33;
public int stringHasher(String ss) {
    BigInteger ll = new BigInteger("0");
    for (int i=0; i<ss.length(); i++) {
        BigInteger bb = BigInteger.valueOf(POLY_MULT).pow(i).multiply(BigInteger.valueOf((int)ss.charAt(i)));
        ll = ll.add(bb);
    }
    ll = ll.mod(BigInteger.valueOf(backingArray.length));
    return ll.intValue();
}
Collisions

drawing 500 unique words from Oliver Twist and assuming a hashtable size of 1009, get these collisions

16 probable child when
42 fagins xxix importance that xv administering
104 stage pledge near
132 surgeon can night
271 things fang birth
341 alone sequel life
415 maylie check circumstances
418 mentioning containing growth
625 meet she first
732 there affording encounters
749 possible out acquainted
761 never xviii after goaded where
833 marks jew gentleman
985 adventures inseparable experience
Realistic hash codes computation in Java

- Use the hashcode function defined on Java Object.
- So put into hashtable is just

```java
private int h(Object k) {
    return k.hashCode() % backingArray.length;
}

public void put(Object key, Object value) {
    backingArray[h(key)] = value;
}
```
Collisions

• Handling of collisions is one of the most important topics for hashtables

• Rehashing
  • make the table bigger
  • $O(n)$ time so want to avoid

• Alternative to rehashing
  • Separate Chaining
  • Probing
Separate Chaining

• Idea: each spot in hashtable holds a array list of key value pairs when the key maps to that hashvalue.
• Replace the item if the key is the same
• Otherwise, add to list
• Generally do not want more than about number of objects as size of table
• Chains can get long
Hash tables get crowded, chains get long

**HT_SIZE=1009**

Using unique words drawn from “Oliver Twist”. Unique count at top of table

<table>
<thead>
<tr>
<th>278</th>
<th>473</th>
<th>1550</th>
<th>2510</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>762</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>217</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
In class exercise

- Show the final contents of the hashtable using separate chaining assuming
  - table size is 7
  - $h(t) = t \mod 7$
  - Data: <0,a> <32,b> <39,c> <12,d> <14,e> <35,f> <27,g> <13,h> <15,i> <5,j> <12,k> <13,l> <4,m> <0,n> <35,o>
- What is the longest chain?
Open Addressing
Linear Probing

• Store only \(<K,V>\) at each location in array
  • No awkward lists
• If key is different and location is in use then go to next spot in array
  • repeat until free location found
Linear Probing Example

- Suppose
  - hashtable size is 7
  - $h(t) = t \mod 7$
  - add:
    - $<3,A>$
    - $<10,B>$
    - $<17,C>$
    - $<17,C>$
    - $<24,Z>$
    - $<3,D>$
    - $<4,E>$
Linear Probing

- Store only <K,V> at each location in array
- No awkward linked lists
- If key is different and location is in use then go to next spot in array
- If key is same, replace value
- Repeat until free location found
Probing Distance

- 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
  - Use a second hash function to determine jumps
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
  • NOTE: cheating here $O()$ is about true worst case
Open Addressing 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 linked list
- Efficient probing requires soft/lazy deletions – tombstoning
- De-tombstoning
In class exercise

• Show the final contents of the hashtable using linear probing assuming
  • table size is 13
  • \( h(t) = t \% 13 \)
  • Data: <0,a> <32,b> <39,c> <12,d> <14,e> <35,f> <27,g> <13,h> <15,i> <5,j> <12,k> <13,l> <4,m> <0,n> <35,o>
  • What is the most number of steps you needed to take to find a free location?
Using Hashtables

• No worries about hashing functions, rehashing, ...
  • Someone else responsibility
• Example: who is visiting my site, and how often?
  • for instance, hackers?
• web servers keep access logs
• java.util.HashMap
Lab

• What is the time complexity, big O, O(?) of these 2 functions?

```java
long complex(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length / 2; i++) {
        for (int j = data.length / 2; j < data.length; j++) {
            res++;
        }
    }
    return res;
}

long complex2(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length; i++) {
        for (int j = 1; j < data.length; j=j*2) {
            res++;
        }
    }
    return res;
}
```