1.00 Lecture 32

Hashing

Reading for next time: Big Java 18.1-18.3

Motivation

- Can we search in better than *O(lg n)* time, which is what a binary search tree provides?
- For example, the operation of a computer memory does considerably better than this. A computer memory takes a key (the memory address) to insert or retrieve a data item (the memory word) in constant (*O(1)*) time.
  - Access times for computer memory do not go up as the size of the computer memory or the proportion used increases.

Direct Addressing

- Computer memory access is a special case of a technique called *direct addressing* in which the key leads directly to the data item.
- Data storage in arrays is another example of direct addressing, where the array index plays the role of key.
- The problem with direct addressing schemes is that they require storage equal to the range of all possible keys rather than proportional to the number of items actually stored.

Direct Addressing Example

- Let's use the example of social security numbers.
- A direct addressing scheme to store information on MIT students based on social security number would require a table (array) of 1,000,000,000 entries since a social security number has 9 digits.
- It doesn't matter whether we expect to store data on 100 students or 100,000,000.
- A direct addressing scheme will still require a table that can accommodate all 1 billion potential entries.

Hashing

- **Hashing is a technique that provides speed comparable to direct addressing (*O(1)*) with lower memory requirements (*O(n)*, where *n* is the number of entries actually stored in the table).**
  - **If the number of entries actually stored in the table is comparable to the maximum possible number, don't use hashing. Just use an array.**
- **Hashing uses a function to generate a pseudo-random *hash code* from the object key and then uses this *hash code* (~direct address) to index into the *hash table*.**

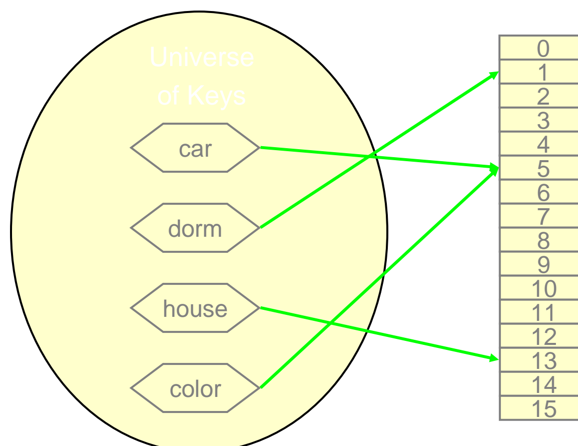
Hashing Example

- **Suppose that we want a small hash table with a capacity of 16 entries to store English words.**
- **Then we will need a hash function that will map English words to the integers 0, 1, ..., 15.**
- **We usually divide the task of creating a hash function into two parts:**
  1. **Map the key into an integer. *(hash1 function)***
  2. **Map the integer "randomly" or in a well distributed way to the range of integers (*{ 0, ..., m-1 }*, where m is the capacity (or number of entries) that will be used to index into the hash table. *(hash2 function)***

Hash Code Example

- As an example, consider the hash code that takes the numeric value of the first character of the word and adds it to the numeric value of the last character of the word (*hash1*), then takes the remainder *mod* 16 (*hash2*).
- For instance, the numeric value of a "c" is 99 and of a "r" is 114. So, "car" would hash to *(99 + 114) mod 16 = 5*.

Hash Code Diagram



Collisions

- "car" and "color" hash to the same value using this hash function because they have the same first and last letter. Our hash function may not be as "random" as it should be.
- But if *n* > *m*, duplicate hash codes, otherwise known as *collisions*, are inevitable.
- In fact, even if *n* < *m*, collisions are likely as a consequence of *von Mises* argument (also known as the birthday paradox: if there are 23 people in a room, the chance that at least two of them have the same birthday is greater than 50%).

Exercise 1

- Create a FirstHash class (no download)
  - Don't call it HashTest (will conflict with later download)
- Write a public static int hash1(String) method
  - Use c= charAt(int i) in a loop to get each character in String
  - Use int k= c to get the integer value of the character
  - Your hash1 value will be the sum of the k values
- Write a main() method:
  - Compute hash values (ints) using your hash1() method for the following Strings:
    - Red, blue, yellow, green, orange, black, brown, purple
  - Use the mod (%) operator to map your 8 ints into a hash table of size 10 (0-9)
    - This is the hash2 function: h2= h1 % 10
  - Print out your results
    - Are there any collisions?
    - Do you think your hash function is good?

Exercise 1

```
public class FirstHash {
   public static void main(String[] args) {
         String[] colors= {"red", "blue", "yellow", "green",
                "orange", "black", "brown", "purple"};
        // Loop thru colors array:
                // Compute hash1 value (call method below)
                // Compute hash2 value (hash1 mod 10)
                // Print out String, hash1, hash2
   }

   public static int hash1(String s) {
        // Initialize hash value to 0
        // Loop thru all characters in s (use s.length())
                // Get character at each position (s.charAt(i))
                // Convert character to int (int k= c)
                // Add k to hash value; return it when done
        }
}
```

Hashing Tasks

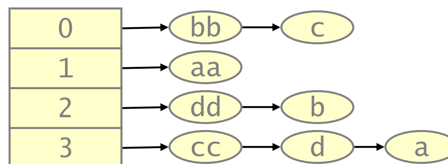
1. **Designing an appropriate hash function to assign hash codes to keys in such a way that a non-random set of keys generates a well-balanced, "random" set of hash codes;**
   - **If the hash codes aren't random, excessive collisions will "clump" the keys under the same direct address.**
2. **Coping with any collisions that arise after hashing.**

Chaining to Avoid Collisions

- *Chaining* is a simple and efficient approach to managing collisions.
- In a hash table employing chaining, the table entries, usually known as slots or buckets, don't contain the stored objects themselves, but rather linked lists of objects.
- Objects with colliding keys are inserted on the same list.
- Insertion, search, and deletion become 2 step processes:
  1. Use the hash function to select the correct slot.
  2. Perform the required operation on the linked list that is referenced at that slot.

Chaining Illustration

keys = { a, b, c, d, aa, bb, cc, dd }



Load Factor and Performance

- The ratio of the number of items stored, *n*, to the number of table slots, *m*, *n/m*, is called the table's *load factor*.
- Because the linked lists referenced by the hash slots can accommodate an arbitrary number of elements, there is no limit on the capacity of a hash table that employs chaining.
- If the hash function employed does not distribute the keys well, the performance of the table will degrade.
- The worst case for a hash table as for a binary search tree is that of the linked list. This occurs when all the keys hash to the same slot.
- Given a good hash function, however, it can be proved that a hash table employing chaining with a load factor of L can perform the basic operations of insertion, search, and deletion in O(1 + L) time.
- For efficiency, keep load factor ≤ 5 (when using chaining)

Java hashCode()

- In an object-oriented language like Java, the first phase of hashing, the $hash_1$ function, is the responsibility of the key class (the data type being stored in the hash table), not the hash table class.
- The hash table will be storing entries as `Objects`. It does not know enough to generate a hash code from the `Object`, which could be a `String`, an `Integer`, or a custom object.
- Java acknowledges this via the `hashCode()` method in `Object`. All Java classes implicitly or explicitly extend `Object`. And `Object` possesses a method `hashCode()` that returns an `int`.
- Caution: the `hashCode()` method can return a negative integer in Java; if we want a non-negative number, and we usually do, we have to take the absolute value of the `hashCode()`.

Hash Code Design

- There is more art than science in hashing, particularly in the design of $hash_1$ functions.
  - "Art" should terrify you in this context!
- The ultimate test of a good hash code is that it distributes its keys in an appropriately "random" manner.
- There are a few good principles to follow:
  1. A hash code should depend on as much of the key as possible.
  2. A hash code should assume that it will be further manipulated to be adapted to a particular table size, the $hash_2$ phase.

The $hash_2$ Function

- Once the `hashCode()` method returns an `int`, we must still distribute it, the $hash_2$ role, into one of the m slots, h, $0 \le h < m$. The simplest way to do this is to take the absolute value of the modulus of the hash code divided by the table size, m:

```
k = Math.abs( o.hashCode() % m );
```

- This method may not distribute the keys well, however, depending on the size of m. In particular, if m is a power of 2, 2^p, then this $hash_2$ will simply extract the low order p bits of the input hash.
- If you can rely on the randomness of the input $hash_1$, then this is probably adequate. If you can't, it is advisable to use a more elaborate scheme by performing an additional hash using the input hash as key.

Integer Hashing

A good method to hash an integer (including our hash codes) multiplies the integer by a number, *A*, *0<A<1*, extracts the fractional part, multiplies by the number of table slots, *m*, and truncates to an integer. In Java, if `n` is the integer to be hashed, this becomes

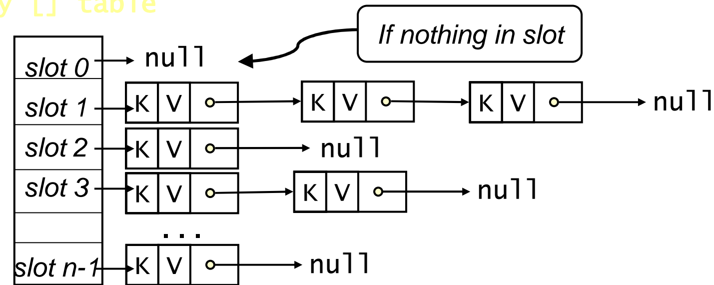
```
private int hashCode( int n ) {
   double t = Math.abs( n ) * A;
   return ( (int) (( t - (int)t ) * m ) );
}
```

Unintuitively, certain values of A seem to work much better than others. The literature suggests that the reciprocal of the golden ratio, *(sqrt(5.0) - 1.0) / 2.0)* works.

Hash Table Implementation

- **We are going to implement our hash table (`HashMap`) as a `Map` with keys and values.**
- **A `HashMap` is unordered. The items in the hash table have no order at all.**
- **We are using *singly linked lists* to resolve collisions.**
- **Since our single linked list implementation from the earlier lecture is not a map and does not accommodate keys and values, we have embedded a reduced implementation of a singly linked list map in the `HashMap` class itself.**

Sample Hashtable with Chaining



(You've seen this before: this is the same as the graph adjacency list!)

Map Members

```
public interface SimpleMap {
  // Inserts the key and value
  public void put(Object key, Object value);
  // Returns the value
  public Object get(Object key);
  // Empty the map
  public void clear();
  // Is map empty?
  public boolean isEmpty();
  // Number of entries
  public int size();
}
```

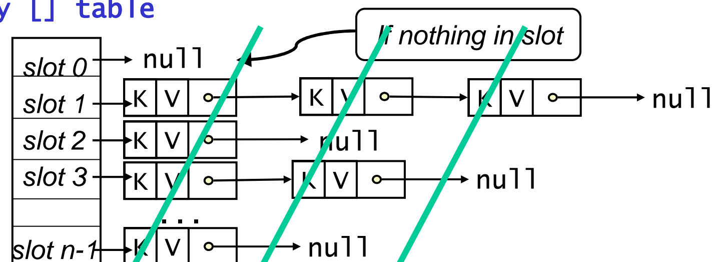
`SimpleHashMap` Members

```
public class SimpleHashMap implements SimpleMap {
   private int length = 0;
   private Entry [] table = null;     // Heads of chains

   public SimpleHashMap( int slots ) {
        table = new Entry[ slots ];
        clear();
   }
   private static class Entry {
        final Object key;      // Package access—remember that?
        Object value;
        Entry next;
        Entry( Object k, Object v, Entry n ) {
               key = k; value = v; next = n; }
   }
```

Exercise 2

- Download SimpleMap, SimpleHashMap, SimpleHashTest
- Write one of the utility methods for SimpleHashMap, clear()
  - clear() removes all Entries from the table, sets length=0



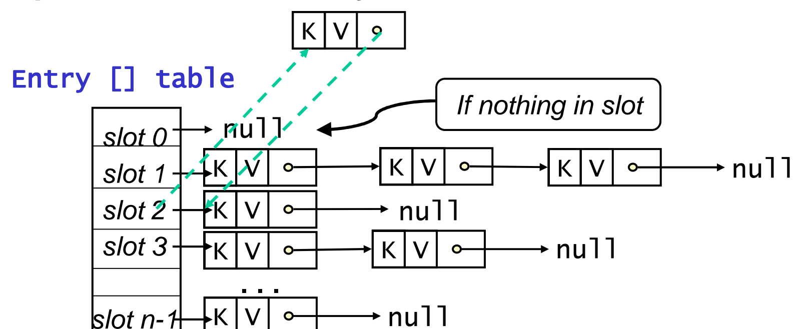
Exercise 3

- **Write the put() method to place an Entry in the table:**

```
public void put( Object k, Object v )
{
    // Invoke Object k's hashCode() method
    // Remember to get the absolute value
    // Use % operator to find index in table
    // Create new Entry with k, v, Entry ref
}
    // Assume k not null, no duplicate keys
```

Exercise 3, p.2

- **Insert the new Entry as the first Entry in the chain. Use the previous table[index] reference to point to the next Entry in the chain**



Exercise 4

- **Write the get() method to return whether a key exists in the table and, if so, return its value**

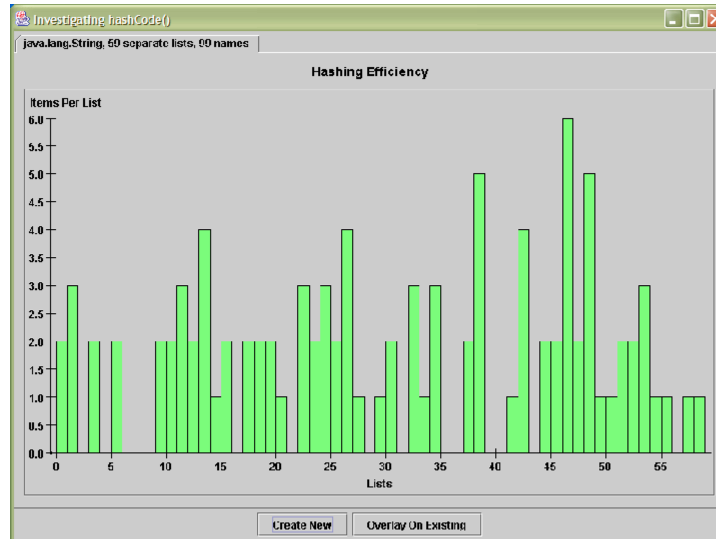
```
public Object get( Object k ) {
    // Assume k is not null, no duplicate keys
    // Find table index of k as before:
    //     find k's hash value and mod (%) it
    // Start at that table index and walk
    // the chain looking for k. If found, return it
    // It's just like walking a linked list:
    //    while (current != null) …
}
```

- **When this works, run SimpleHashTest**

SimpleHashTest

```
public class SimpleHashTest {
   public static void main(String[] args) {
        SimpleHashMap1 h= new SimpleHashMap1(20);
        h.put("Kenneth", "3-1975");
        h.put("Mary", "3-1325");
        h.put("Ailene", "3-5234");
        h.put("Michael", "3-6543");
        h.put("Ferd", "3-6350");
        String phone= (String) h.get("Ferd");
        System.out.println(phone);
        h.remove("Ferd");
        phone= (String) h.get("Ferd");
        if (phone != null)
               System.out.println(phone);
        else
               System.out.println("Ferd's not in the book");
   }
}
```

The HashTest Application



HashTest Introduction

`HashTest.java:`

- Allows a user to select <u>any</u> Java class with a constructor that takes a single String argument,
- Creates 235 instances of the class, using the names of past 1.00 students and the String constructor of the class,
- Inserts the resulting instances into a HashMap and creates a histogram representing the distribution of the objects in the HashMap.
- By examining this histogram, we can gain insight into the efficacy of the hashCode method defined in the class we used.

HashTest Download

- Download the following 10 files: ResultViewer.java, SimpleHash.java, ConstantHash.java, MapIterator.java, Map.java, HashMap.java, HashMain.java, FirstLastName.java, jas.jar, and name.txt
- Import them all into your Lecture32 project in Eclipse
  - Or you may want to create a new project (Lecture32Hash) to keep these separate from the code you just wrote
- Select (right-click) the project, and from its pop-up menu, select Properties.
  - In the Properties dialog, select the Java Build Path page.
  - Click the Libraries tab.
  - If jas.jar isn't shown:
    - Click the Add JARs button (internal JAR)
    - Choose jas.jar and hit 'OK'
- Save/compile the project

Experimenting with HashTest

- Create a histogram for java.lang.String by executing HashTest and clicking on the "Create New" button.
- What is the largest number of collisions for any single list?
- What is the smallest number?
- How many lists are empty?
- Click the overlay button and enter javax.swing.JFrame. Which class appears to have the better hashCode implementation, String or JFrame? Why?
- We made two classes to experiment with: ConstantHash and SimpleHash. Open the source for these classes and examine their hashCode methods.
- Overlay these classes with Swing classes JFrame, JLabel, JButton, JCheckBox, JFileChooser (very slow), JMenu, JPasswordText, or others (see Javadoc).
  - Which are better? Worse?

A Final Word

- Hashing doesn't preserve order in the data:
  - Hash table data isn't sorted, unlike binary search tree data
- Hashing is statistical.
  - If you're hashing billions of items per day (or night), you will be unlucky sometimes, and a system's search/retrieval time will slow to a crawl
  - In industry, a VP calls on your beeper at 2am to have you fix the system, if it dies at 2am. The VP is usually mad.
- If you make your hash tables very big, that offsets the benefits of its faster speed than trees
  - Memory allocation is slow and expensive
- So…choose carefully
  - I never used search hashing in a real system, always using a balanced tree, but your situations may differ.
- Other forms of hashing (linear probing for search, and cryptographic hashes) are somewhat different