CECS 277 – Lecture 10 – Hash Tables

A hash is used to find and store elements quickly. It is comprised of two parts: the Hash Table, which is where the elements are stored, and a Hash Function, which returns a location of where in the table the element is stored. The table's capacity can be set in the constructor. A good capacity is usually twice the expected number of elements. The default capacity is 16 and expands when the load factor (usually 75%) is exceeded.

A good hash function should return a unique, uniformly distributed integer that is calculated using the object's data. It should return different hash codes for objects with different values, and the same hash code for different objects with equal values.

The hash hunction, hashCode(), is overridden from the Object class. It uses the object's memory address to calculate the hash value. This is inadequate in most cases since it will result in different hash codes for all objects, including objects with equal values.

Several commonly used classes that override the hashCode() method are: String, Integer, Double, Character, Point, Rectangle, Color, along with all of the collection classes, such as: ArrayList, LinkedList, HashSet and TreeSet. When you create your own classes that are going to be placed in a hashable collection, you must override the hashCode() and equals() methods so that the collection is able to place it in the hash table.

Example: String's hashCode() - A hashCode method returns an integer given the value of the object. It will always return the same result given the same value, every time.

```
String s = "Hello";
int hashVal = s.hashCode(); //returns 69609650
```

The equation that String uses to find a hash code is:

$$hashCode = S_0 * 31^{(n-1)} + S_1 * 31^{(n-2)} + \ldots + S_{n-1}$$

where n = the number of characters in the string, and S is the Unicode value of that character in the string, 31 is a prime number which is often used for finding hash codes.

It is possible for two different string values to return the same hash value. This is called a collision. When this occurs, it tests the objects to see if they are equal, if they aren't then they both are stored at the same location in the table in a linked list.

```
int h1 = "FB".hashCode(); //returns 2236
int h2 = "Ea".hashCode(); //returns 2236
```

How An Object is Hashed – A hash table is a set of memory locations, often referred to as buckets (linked lists). Whenever an object is hashed, it follows this algorithm:

```
Call the object's hashCode() and get the location
If that bucket location is empty
        Store the object in the bucket at that location
Else
        Use the object's equals() to compare it to the other objects in the bucket
        If it is not equal to any other object in the bucket
                Store the object in the bucket
        Else
                Do not store the object
```

Creating a hashCode Method – When you need to create a hashCode method for your class, you should use the object's data to create a single value, by following these rules:

1. It should always return the same hash value for the same object data.
2. It should return a different hash value for different object data.
3. The hash code values it returns should be uniformly distributed.
4. It should minimize collisions.

Creating an equals Method – Whenever you create a hashCode method, you should also create an equals method. Compare the data of the two objects to see if they are the same. Always make sure your equals method follows these rules:

1. It should be reflexive – any object should equal itself (ex. x.equals(x) is true).
2. It is symmetric – the result should be the same no matter the order of the objects (ex. if x.equals(y) is true, then y.equals(x) should also be true).
3. It is transitive – if two objects are equal, and one of those is equal to another object, then the other should also be equal to the other object (ex. x.equals(y) is true, and y.equals(z) is true, then x.equals(z) should also be true).
4. It is consistent – as long as the object's values have not changed, then the equals method should return the same result every time it is called.
5. Comparing to a null should always return false.

Example: The hashCode and equals methods for a Person and Employee classes.

```
public class Person {
     private String name;
     private int age;
     public Person( String n, int a ) {
          name = n;
          age = a;
     }
     @Override
     public int hashCode() {
          final int prime = 31;
          int result = 1;
          result = prime * result + name.hashCode();
          result = prime * result + age;
          return result;
     }
     @Override
     public boolean equals( Object o ) {
          if( this == o )
               return true;
          if( o == null )
               return false;
          if( getClass() != o.getClass() )
               return false;
          Person p = (Person) o;
          return name.equals(p.name) && age == p.age;
     }
```

```
     @Override
     public String toString() {
          return "Name: " + name + ", Age: " + age;
     }
}
public class Employee extends Person {
     private int pay;
     public Employee( String n, int a, int p ) {
          super( n, a );
          pay = p;
     }
     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + pay;
          return result;
     }
     @Override
     public boolean equals( Object o ) {
          if( this == o )
               return true;
          if( !super.equals( o ) )
               return false;
          if( getClass() != o.getClass() )
               return false;
          Employee e = (Employee) o;
          return pay == e.pay
     }
     @Override
     public String toString() {
          return super.toString() + " Pay: " + pay;
     }
}
public class TestPerson {
     public static void main( String [] args ) {
          HashSet<Person> pSet = new HashSet<Person>();
          pSet.add( new Person( "Mary", 29 ) );
          pSet.add( new Person( "Roger", 63 ) );
          pSet.add( new Employee("Mary", 29, 65000) );
          for( Person p : pSet ) {
               System.out.println( p.hashCode() );
          }
     }
}
```