CS 3410 – Ch 20 – Hash Tables

Sections	Pages
20.1-20.7	773-802

20.1 – Basic Ideas

1. A *hash table* is a data structure that supports insert, remove, and find in constant time, but there is no order to the items stored. We say a hast table supports the retrieval or removal of any *named* item. The *HashSet*, *HashMap*, and *Hashtable* are Java implementations. Our object is to understand the theory behind a hash table and how it is implemented.

2. Example: Suppose that we need to store integers between the values of 0 and 65,535. We could use an ordinary integer array to store the values. To insert a value is of course constant time. However, to find (or remove) an element takes linear time as we have to search the array.

   Here, is a different idea, we can define an array, *ht* with 65,536 positions, and initialize each position with value 0. This *value* represents whether an item is present or not. So, initially, the hash table is empty. See figure 20.1a. Suppose we want to insert the item 48, we do this by writing *ht[48]=1* and the result is shown in Figure 20.1b.

$$ht = \begin{matrix} value \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} index \\ 0 \\ 1 \\ 2 \\ \vdots \\ 65{,}535 \end{matrix}$$

Figure 20.1a – Empty hashtable

$$ht = \begin{matrix} value \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} index \\ 0 \\ 1 \\ 2 \\ \vdots \\ 48 \\ \vdots \\ 65{,}535 \end{matrix}$$

Figure 20.1b – Hashtable with the item 48

With this setup, we can remove 48 with: *ht[48]=0*. Thus, the basic operations are clearly are constant time:

Method	Algorithm
insert(item)	ht[item]++
remove(item)	if(ht[item] > 0) ht[item]--
find(item)	if(ht[item] > 0) return item

3. There are two problems with this approach:

   a. With larger numbers, we need much more storage. For instance, 32 bit integers (the size of Java's *int* which range from -2,147,483,648 to 2,147,483,647) would require an array with over 4 billion elements.
   b. This approach only works with storing integers. For instance, what if we want to store strings, or arbitrary objects?

4. The second problem is simply solved by simply mapping the items we want to store in the hash table to integers. For instance, consider the case of storing strings. An ASCII character can be represented by a number between 0 and 127. For example the string, "junk", could be represented as the integer:

$$'j' * 128^3 + 'u' * 128^2 + 'n' * 128^1 + 'k' * 128^0$$

$$= 106 * 128^3 + 117 * 128^2 + 110 * 128^1 + 107 * 128^0 = 224{,}229{,}227$$

However, even 4 character strings would require a very large array.

5. To solve the first problem, we use a function that maps large numbers to smaller numbers. Thus, we can use a smaller array. A function that maps an item to a small integer index is called a *hash function*.

6. A simple hash function – Suppose we decide on a reasonable array size, *tableSize*. Thus, if x is an arbitrary integer, then:

$$x \bmod tableSize$$

generates an index between 0 and $tableSize - 1$. For example, using $tableSize = 10{,}000$, "junk" would produce the index 9227:

$$224{,}229{,}227 \bmod 10{,}000 = 9227$$

and the hast table would be:

$$ht = \begin{matrix} value \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} index \\ 0 \\ 1 \\ 2 \\ \vdots \\ 'junk' = 9227 \\ \vdots \\ 10{,}000 \end{matrix}$$

7. Now, the problem with this is that *collisions* can occur. In other words, two or more items can hash to the same index. For instance, using a table size of 10, both 89 and 49 *hash* to 9:

$$89 \bmod 10 = 9 \text{ and } 49 \bmod 10 = 9$$

Collisions can be resolved by using the following methods: linear probing, quadratic probing, and separate chaining, which we will study in the following sections.

20.2 – Hash Function

1. Computing the hash functions for strings has a problem: the conversion of the string to an integer usually results in an integer that is too big for the computer to store conveniently. For instance, with a 6 character string, $[a_0 a_1 a_2 a_3 a_4 a_5]$

$$a_0 * 128^5 + a_1 * 128^4 + a_2 * 128^3 + a_3 * 128^2 + a_4 * 128^1 + a_5 * 128^0$$

the 128^5 would immediately overflow a Java *int*.

2. However, we remember, for instance, that a 3rd order polynomial:

$$a_0 * x^3 + a_1 * x^2 + a_2 * x^1 + a_3 * x^0$$

can be evaluated as:

$$[(a_0 x + a_1)x + a_2]x + a_3$$

This computation, in general involves *n* multiplications and *n* additions; however, it still produces overflow, albeit, more slowly.

3. An algorithm for computing the function above, which of course can cause an overflow, is:

```
hash( String key )
   hashVal = 0
   for( i=0; i<key.length(), i++ )
      hashVal = hashVal * 128 + key.charAt(i)
   return hashVal % tableSize
```

4. To solve the overflow problem, we could use the *mod* operator after each multiplication (or addition), but the computation of mod is expensive.

figure 20.1

A first attempt at a hash function implementation

```
1     // Acceptable hash function
2     public static int hash( String key, int tableSize )
3     {
4         int hashVal = 0;
5
6         for( int i = 0; i < key.length( ); i++ )
7             hashVal = ( hashVal * 128 + key.charAt( i ) )
8                                                 % tableSize;
9         return hashVal;
10    }
```

5. We can make this faster by performing a single mod at the end and by changing 128 to 37 to keep the numbers a bit smaller. Also, note that overflow can still occur generating negative numbers. We fix that by detecting it and making them positive

figure 20.2

A faster hash function that takes advantage of overflow

```
1       /**
2        * A hash routine for String objects.
3        * @param key the String to hash.
4        * @param tableSize the size of the hash table.
5        * @return the hash value.
6        */
7       public static int hash( String key, int tableSize )
8       {
9           int hashVal = 0;
10
11          for( int i = 0; i < key.length( ); i++ )
12              hashVal = 37 * hashVal + key.charAt( i );
13
14          hashVal %= tableSize;
15          if( hashVal < 0 )
16              hashVal += tableSize;
17
18          return hashVal;
19      }
```

6. An even quicker hash function would just simply take the sum of the characters. However, if all keys are 8 are fewer characters (there are approximately 208 billion such sequences), then the hash function will only generate indices between 0 and 1016 (127*8) and with a table size of 10,000 this would cause an extreme clustering of strings in positions 0 through 1016 and a high probability of collisions. Although collisions can be handled, as we see in the following sections, we want to use hash functions that distribute the keys more equitably to improve performance.

```
1       // A poor hash function when tableSize is large
2       public static int hash( String key, int tableSize )
3       {
4           int hashVal = 0;
5
6           for( int i = 0; i < key.length( ); i++ )
7              hashVal += key.charAt( i );
8
9           return hashVal % tableSize;
10      }
```

figure 20.3

A bad hash function if `tableSize` is large

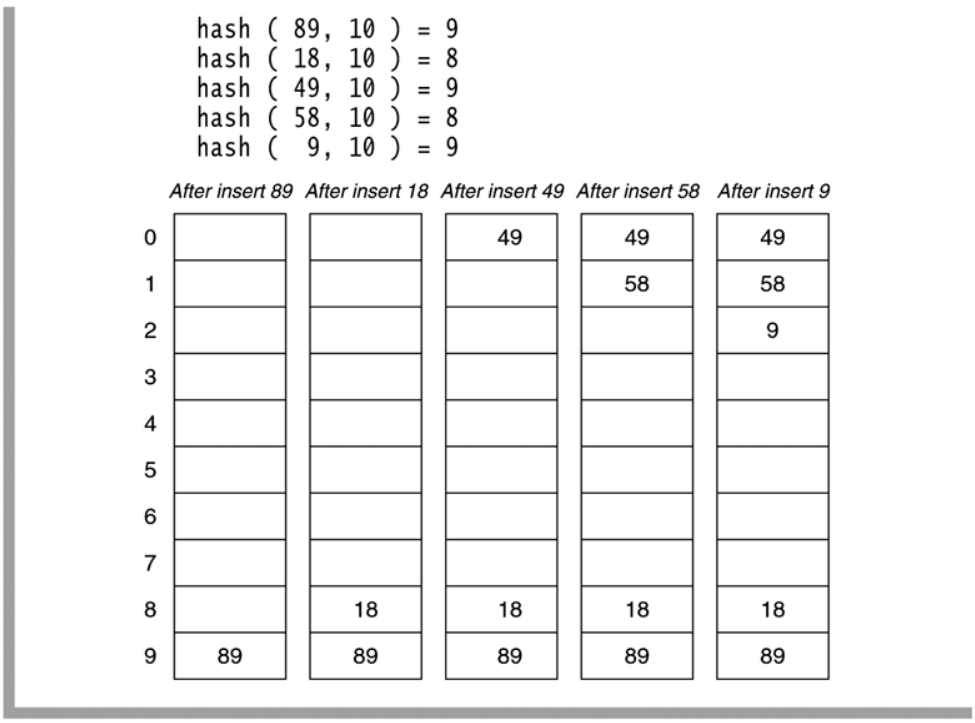
7. Early versions of Java essentially used the algorithm in Figure 20.2 for computing the hash code for strings, but without lines 14-16. Later, it was changed so that longer strings used just a subset of the characters, somewhat evenly spaced to compute the hashcode. This proved problematic in many applications because of situation where the keys were long and very similar, such as file path names or URLs. In Java 1.3, it was decided to store the hash code in the String class, because the expensive part was the computation of the hash code. Initially, the hash code is set to 0. The first time hashCode is called, it is computed and *cached* (remembered). Subsequent calls to hashCode simply retrieve the previously computed value. This technique is called *caching the hash code*. It works because strings are immutable.

20.3 – Linear Probing

1. Suppose that we are going to add an object to a hashtable. A collision occurs when the hash position of this object is already occupied. We must decide how we will handle this situation. The simplest solution is to search sequentially until we find an empty position. We call this *linear probing*.

figure 20.4

Linear probing hash table after each insertion



2. As long as the table is large enough, we can always find a free cell. If there is only one free cell left in the table, you might have to search the entire table to find it. On average, in the worst case, we might expect to have to search about half the table. This is not constant time! However, if we keep the table relatively empty, insertions should not be too costly.

3. The find operation is similar to the insert. We go to the hash position and see if that is the object we are looking for. If so, we return it. If not, we keep searching sequentially. If we find an empty cell, the object was not found. Otherwise, we will eventually find it.

4. The remove operation has a small twist: we can't actually remove the object because it is serving as a placeholder during collision resolution. Thus, we implement *lazy deletion* by marking an item as removed instead of physically removing it. We will introduce an extra data member to keep track of whether an item is *active* or *inactive* (removed).

Homework 20.1

1. Problem 20.5 a in text.
2. Problem 20.6 a in text.

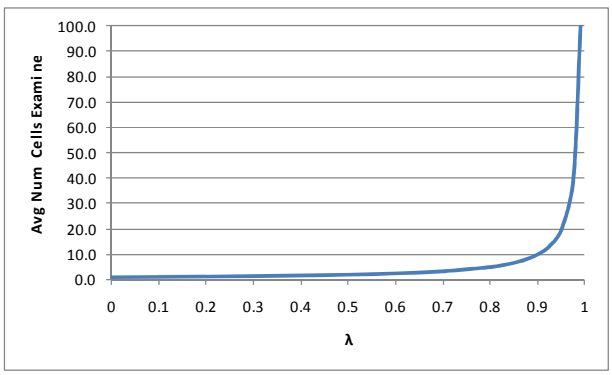
20.3.1 – Naive Analysis of Linear Probing

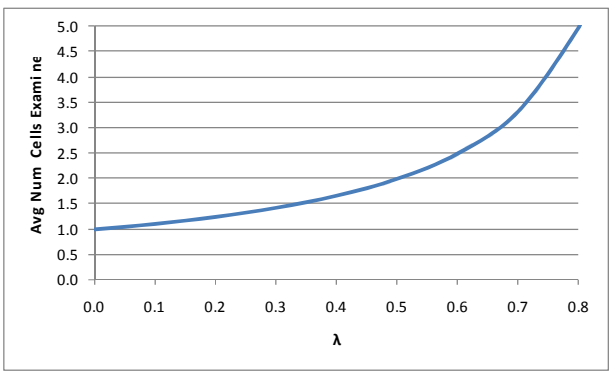
1. The *load factor,* λ of a hash table is the fraction of the table that is full. Thus, λ ranges from 0 (empty) to 1 (full).

2. If we make the assumptions:

    1. The hash table is large
    2. Each probe is independent of the previous probe

    then it can be shown that the average number of cells examined in an insertion using linear probing is $\frac{1}{1-\lambda}$. Thus when $\lambda = 0.5$, the average number of cells examined is 2; $\lambda = 0.75$, the average is 4; $\lambda = 0.95$, the average is 20. Consider the blue (middle) curve below which shows a graph of this function:





20.3.2 – Clustering

1. Unfortunately, this analysis is incorrect because assumption 2 above is not correct; probes are not independent. However, the result is useful because it serves as sort of a best case. What happens in practice, is that *clustering* occurs, where large blocks of occupied cells are formed. Thus, any key that hashes into a cluster must traverse the cluster, and then add to the cluster.

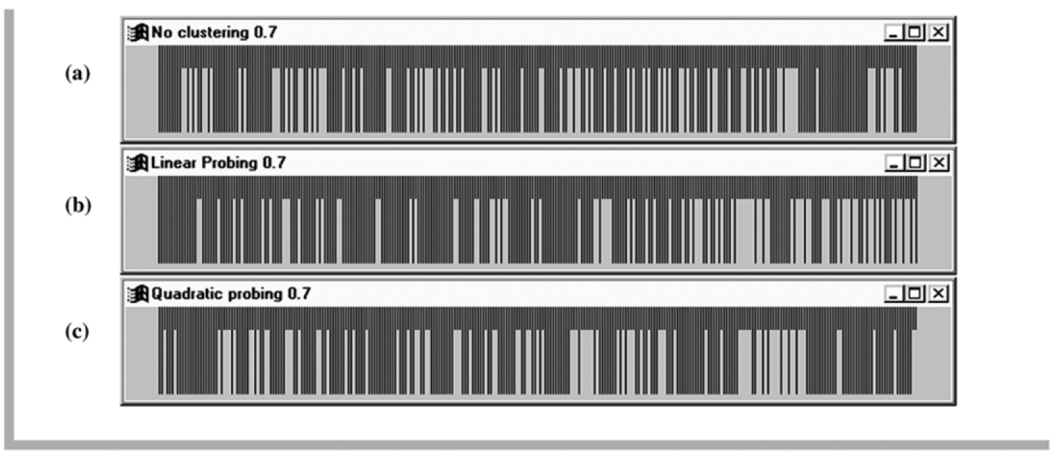
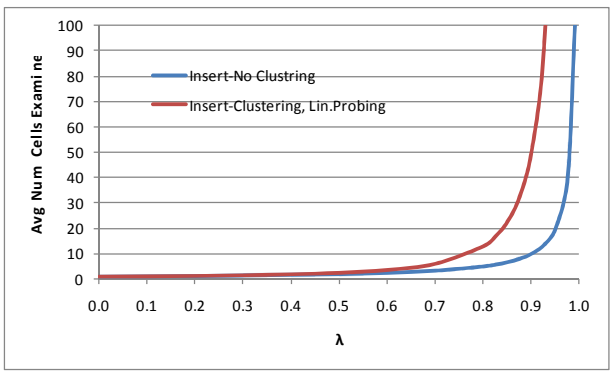
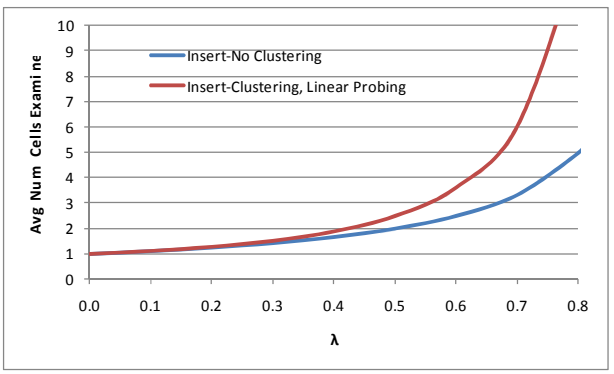


figure 20.5

Illustration of primary clustering in linear probing (b) versus no clustering (a) and the less significant secondary clustering in quadratic probing (c). Long lines represent occupied cells, and the load factor is 0.7.

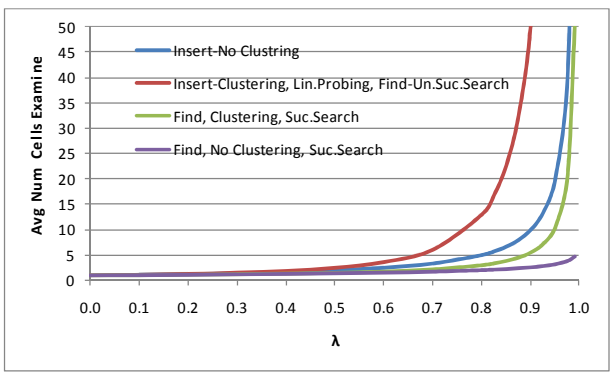
2. It can be shown that a better estimate of the number of cells examined in an insertion using linear probing is $\frac{1+\frac{1}{(1-\lambda)^2}}{2}$ which is shown in the red curve below. The main differences are seen when λ gets large. For instance, with $\lambda = 0.9$, the naive analysis shows 10 cells examined while the actual value is around 50.

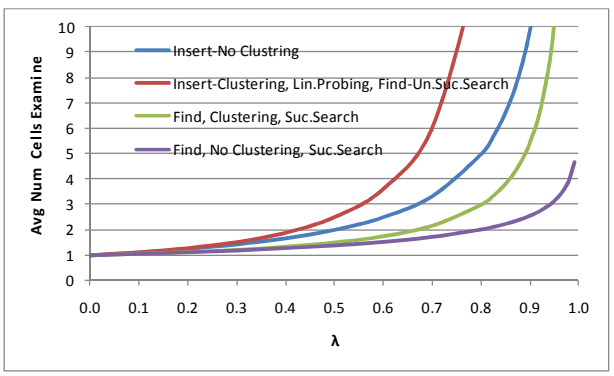




20.3.3 – Analysis of Find

1. There are two types of finds: successful and unsuccessful. The average number of cells examined for an unsuccessful find is the same as an insert. Thus, the cost of an unsuccessful find is the same as an insert. The cost of a successful search for X is equal to the cost of inserting X at the time X was inserted. It can be shown that the average number of cells examined in a successful search is $\frac{1+\frac{1}{1-\lambda}}{2}$ as shown by the green curve below. It can also be shown that the cost of a successful search when there is no clustering is $-\ln(1-\lambda)/\lambda$ as shown by the purple curve.





λ	Insert No Clustering	Insert & Unsuccessful Find Clustering	Successful Find Clustering	Successful Find No Clustering
0.000	1.0	1.0	1.0	1.0
0.100	1.1	1.1	1.1	1.1
0.200	1.3	1.3	1.1	1.1
0.300	1.4	1.5	1.2	1.2
0.400	1.7	1.9	1.3	1.3
0.500	2.0	2.5	1.5	1.4
0.600	2.5	3.6	1.8	1.5
0.700	3.3	6.1	2.2	1.7
0.800	5.0	13.0	3.0	2.0
0.825	5.7	16.8	3.4	2.1
0.850	6.7	22.7	3.8	2.2
0.875	8.0	32.5	4.5	2.4
0.900	10.0	50.5	5.5	2.6
0.925	13.3	89.4	7.2	2.8
0.950	20.0	200.5	10.5	3.2
0.975	40.0	800.5	20.5	3.8
0.990	100.0	5000.5	50.5	4.7

2. To reduce the number of probes, we need a collision resolution technique that avoids primary clustering. Note from the table above, that for $\lambda = 0.5$ not much is gained from such a strategy. Thus, the author concludes that linear probing is not a terrible strategy.

Homework 20.2

1. Problem 20.4 in text.

20.4 – Quadratic Probing

1. Quadratic Probing is a technique that eliminates the primary clustering problem of linear probing. If a hash function evaluates to H and it is not the appropriate cell, then we try $H + 1^2, H + 2^2, H + 3^2, \ldots$, wrapping around appropriately.

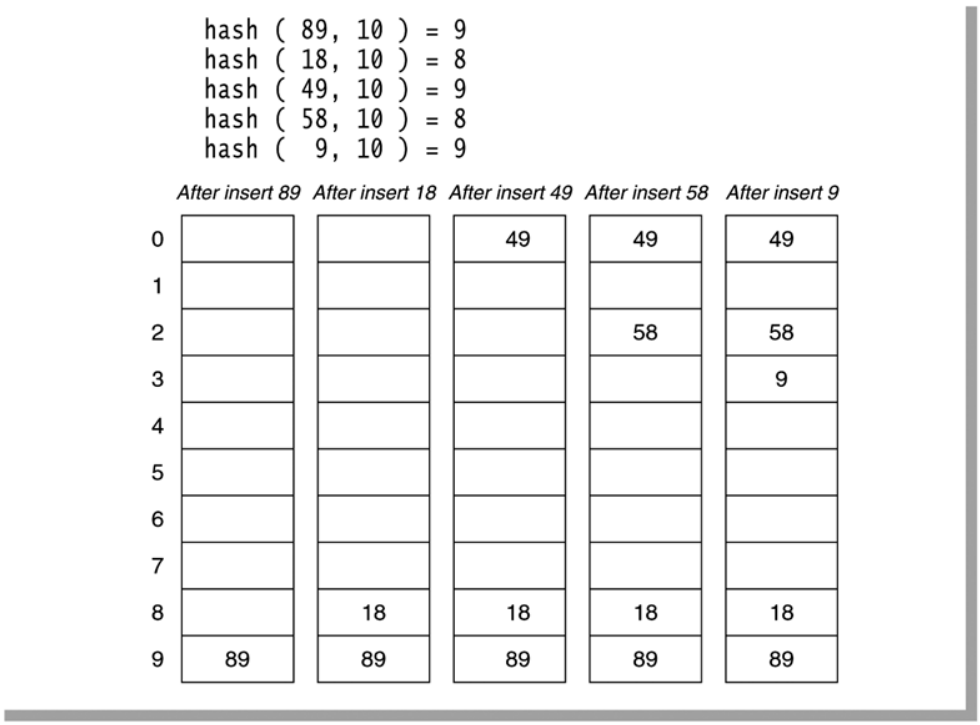


figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

2. Several questions arise: If the table is not full, will this method always insert a new value? Are we guaranteed of not revisiting a cell during the execution of this method? Answer: If the table size is a prime number and the load factor doesn't exceed 0.5 then the answers are "yes."

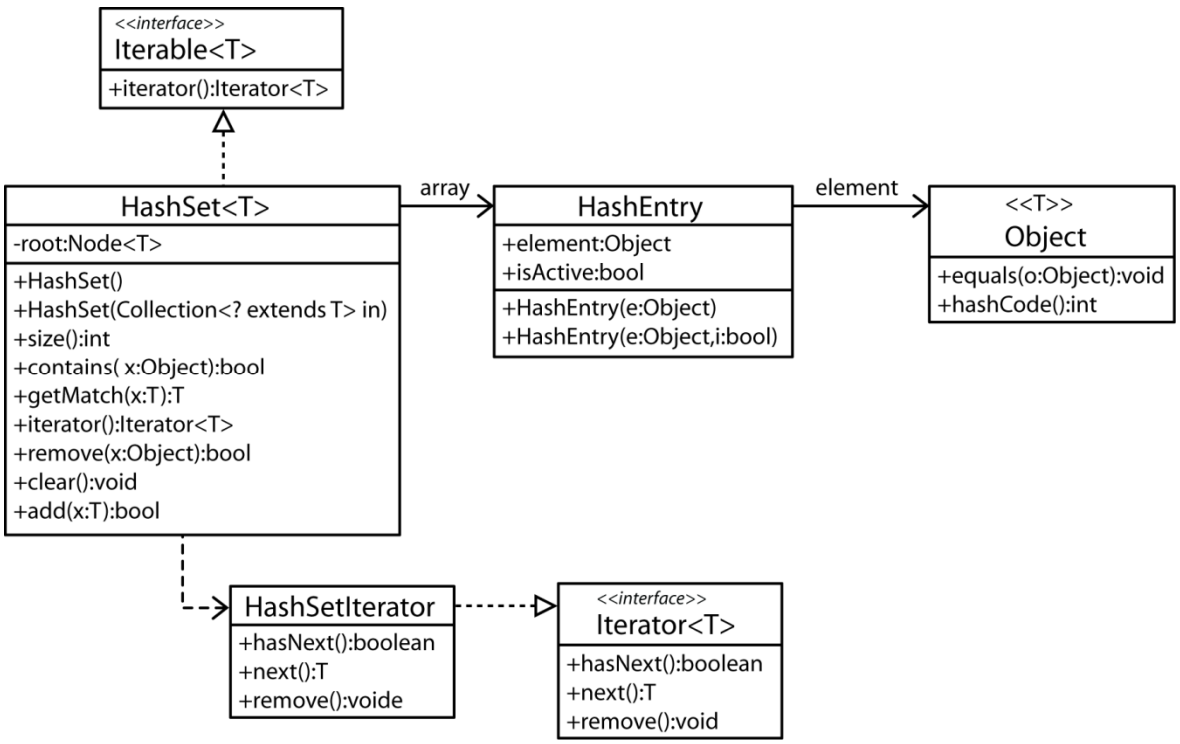
3. If there is just one more element in the table, above a load factor of 0.5, then the insert could fail (very unlikely, but still a reality). Thus, to make a new, larger table, we need table size that is larger and a prime number. This is easy to do. Finally, since we have a new table, we must now add the values in the old table to the new one using the new hash function. This process is called *rehashing*.

Homework 20.3

1. Problem 20.2 in text. Assume quadratic probing.
2. Problem 20.5 b in text.
3. Problem 20.6 b in text.

20.4.1 – Java Implementation

1. A class diagram for the implementation of HashSet:



2. The code for the HashSet class is shown below. First, we notice that a HashSet utilizes an array, *array* of HashEntry objects for storage (at bottom). We will focus on the *add* and *remove* methods. These methods rely on the *findPos* method, which is the only place that probing (quadratic) is implemented. In short, *findPos* will find the next available empty spot for an item to be added. Or, in the case of remove, it will find the location of the item to be removed. The field *currentSize* contains the (logical) number of items in the table, the number of *active* items. The field *occupiedSize* contains the total number of cells in *array* that contain a HashEntry, *e.g.* the number of non-null cells. When an item is removed, the HashEntry is not removed, it's *isActive* field is set to *false*. Remember, this is done to maintain the integrity of the probing. Thus, an entry that is removed, is not physically removed. We will see in the *add* algorithm that a removed entry location is never reused, except in the case when a previously removed item is added again.

figure 20.8

The class skeleton for a quadratic probing hash table

```
1  package weiss.util;
2
3  public class HashSet<AnyType> extends AbstractCollection<AnyType>
4                                 implements Set<AnyType>
5  {
6      private class HashSetIterator implements Iterator<AnyType>
7        { /* Figure 20.17 */ }
8      private static class HashEntry implements java.io.Serializable
9        { /* Figure 20.9 */ }
10
11     public HashSet( )
12       { /* Figure 20.10 */ }
13     public HashSet( Collection<? extends AnyType> other )
14       { /* Figure 20.10 */ }
15
16     public int size( )
17       { return currentSize; }
18     public Iterator iterator( )
19       { return new HashSetIterator( ); }
20
21     public boolean contains( Object x )
22       { /* Figure 20.11 */ }
23     private static boolean isActive( HashEntry [ ] arr, int pos )
24       { /* Figure 20.12 */ }
25     public AnyType getMatch( AnyType x )
26       { /* Figure 20.11 */ }
27
28     public boolean remove( Object x )
29       { /* Figure 20.13 */ }
30     public void clear( )
31       { /* Figure 20.13 */ }
32     public boolean add( AnyType x )
33       { /* Figure 20.14 */ }
34     private void rehash( )
35       { /* Figure 20.15 */ }
36     private int findPos( Object x )
37       { /* Figure 20.16 */ }
38
39     private void allocateArray( int arraySize )
40       { array = new HashEntry[ arraySize ]; }
41     private static int nextPrime( int n )
42       { /* Figure 20.7 */ }
43     private static boolean isPrime( int n )
44       { See online code  */ }
45
46     private int currentSize = 0;
47     private int occupied = 0;
48     private int modCount = 0;
49     private HashEntry [ ] array;
50 }
```

3. The HashEntry contains a field for the item being stored and a field to indicate if the item is active. In the code that follows, the second constructor is always used with *true* specified.

```
1     private static class HashEntry implements java.io.Serializable
2     {
3         public Object  element;   // the element
4         public boolean isActive;  // false if marked deleted
5
6         public HashEntry( Object e )
7         {
8             this( e, true );
9         }
10
11        public HashEntry( Object e, boolean i )
12        {
13            element  = e;
14            isActive = i;
15        }
16    }
```

figure 20.9

The HashEntry nested class

4. The HashSet is created by creating an array whose size is a prime number greater than or equal to the input value.

```
1     private static final int DEFAULT_TABLE_SIZE = 101;
2
3     /**
4      * Construct an empty HashSet.
5      */
6     public HashSet( )
7     {
8         allocateArray( DEFAULT_TABLE_SIZE );
9         clear( );
10    }
11
12    /**
13     * Construct a HashSet from any collection.
14     */
15    public HashSet( Collection<? extends AnyType> other )
16    {
17        allocateArray( nextPrime( other.size( ) * 2 ) );
18        clear( );
19
20        for( AnyType val : other )
21            add( val );
22    }
```

figure 20.10

Hash table initialization

```
private void allocateArray( int arraySize )
{
    array = new HashEntry[ nextPrime( arraySize ) ];
}
```

5. The method *findPos* is used by *add* and *remove* (and *getMatch* and *contains*). If called by *add* with an item to be added, *x* it will hash *x* and find the position, *currentPos* it belongs in (line 9). If there is no HashEntry in that location, then the loop at line 12 is aborted and *currentPos* is returned. If there is a HashEntry in *currentPos,* the probing sarts. If the item in the HashEntry is the same as the one we are trying to add (line 19), *x,* then we return that position. This is the case where the item already exists in the HashSet. If item is not the one we are trying to add, then we increment the *currentPos* and loop again. Eventually, we will either find the item or find an empty spot. The *remove* method will use *findPos* in a similar way. First, the *currentPos* if the item to be removed is found (line 9). If the position is empty (line 12), the *currentPos* is immediately returned. This occurs when the item is not found and no probing is needed. If *currentPos* is occupied, then we check to see if the item there is the one to be removed (line 19). If so, we immediately return that location. Otherwise, we increment the *currentPos* and continue. Eventually, we will find the item or find an empty spot.

```
1     /**
2      * Method that performs quadratic probing resolution.
3      * @param x the item to search for.
4      * @return the position where the search terminates.
5      */
6     private int findPos( Object x )
7     {
8         int offset = 1;
9         int currentPos = ( x == null ) ?
10                           0 : Math.abs( x.hashCode( ) % array.length );
11
12        while( array[ currentPos ] != null )
13        {
14            if( x == null )
15            {
16                if( array[ currentPos ].element == null )
17                    break;
18            }
19            else if( x.equals( array[ currentPos ].element ) )
20                break;
21
22            currentPos += offset;                  // Compute ith probe
23            offset += 2;
24            if( currentPos >= array.length )       // Implement the mod
25                currentPos -= array.length;
26        }
27
28        return currentPos;
29    }
```

figure 20.16

The routine that finally deals with quadratic probing

6. The *add* method first finds the position (line 8) where the item to be added belongs (a location that is currently null) or the location where the item already exists. If *currentPos* refers to a cell that is null, then *isActive* will return false and the item will be added (line 12). If *currentPos* refers to a position that is not null, then the item in the HashEntry must be the item we are trying to add. There are two cases. First, if the item is *active*, *e.g.* the item already exists, then we return *false* (line 10) without adding the item. Second, if the item is *not active, e.g.* the item previously existed in this location but was subsequently removed, then the item will be (re)added (line 12) in its previous position. Note that anytime an item is added, both *currentSize* and *occupied* are incremented. (It looks like to me this will over-count *occupied* in the case of adding a previously removed item).

figure 20.14

The add routine for a quadratic probing hash table

```
1       /**
2        * Adds an item to this collection.
3        * @param x any object.
4        * @return true if this item was added to the collection.
5        */
6       public boolean add( AnyType x )
7       {
8           int currentPos = findPos( x );
9           if( isActive( array, currentPos ) )
10              return false;
11
12          array[ currentPos ] = new HashEntry( x, true );
13          currentSize++;
14          occupied++;
15          modCount++;
16
17          if( occupied > array.length / 2 )
18              rehash( );
19
20          return true;
21      }
```

7. This is a static method in the HashSet class. It checks the internal array and returns true only when there is a HashEntry object at *pos* and it is active.

```
1   /**
2    * Tests if item in pos is active.
3    * @param pos a position in the hash table.
4    * @param arr the HashEntry array (can be oldArray during rehash).
5    * @return true if this position is active.
6    */
7   private static boolean isActive( HashEntry [ ] arr, int pos )
8   {
9       return arr[ pos ] != null && arr[ pos ].isActive;
10  }
```

figure 20.12

The isActive method for a quadratic probing hash table

8. The *remove* method first finds the position (line 8) where the item to be removed exists or a location that is null in the case where the item was not found. If the item in *currentPos* is null, *e.g.* the item was not found, then *isActive* will be *false* (line 9) and the *remove* method will immediately return (line 10). Similarly, if the item was found, but it is inactive (line 9), *e.g.* it was previously deleted, then we immediately return (line 10). Finally, if the item was found and it is *active* (line 9), then it is set to inactive (line 12) and *currentSize* is decremented.

   If the *currentSize* falls below a certain level (line 16), due to a removal, then the hash set is resized (line 17). (The condition for resizing doesn't seem judicious. If you add the first item to the hash set and then remove it before adding another, it is resized from 101 to 3. Then, if you add two more elements, the table is resized from 3 to 11.

```
1     /**
2      * Removes an item from this collection.
3      * @param x any object.
4      * @return true if this item was removed from the collection.
5      */
6     public boolean remove( Object x )
7     {
8         int currentPos = findPos( x );
9         if( !isActive( array, currentPos ) )
10            return false;
11
12        array[ currentPos ].isActive = false;
13        currentSize--;
14        modCount++;
15
16        if( currentSize < array.length / 8 )
17            rehash( );
18
19        return true;
20    }
21
22    /**
23     * Change the size of this collection to zero.
24     */
25    public void clear( )
26    {
27        currentSize = occupied = 0;
28        modCount++;
29        for( int i = 0; i < array.length; i++ )
30            array[ i ] = null;
31    }
```

figure 20.13

The `remove` and `clear` routines for a quadratic probing hash table

9. The *rehash* method creates a new array with a size that is at least four times the current size and a prime number (line 10). Then, active entries in the old table (line 16) are added to the new table (line 17). This cleans up the new table of inactive entries, *e.g.* items that have been removed. Of course, when you add an old entry (active) to the new table, a new hash location is computed in *findPos* as the length of the array is now larger. Thus, this potentially breaks up any clustering that may have existed in the old table.

figure 20.15

The rehash method for a quadratic probing hash table

```
1       /**
2        * Private routine to perform rehashing.
3        * Can be called by both add and remove.
4        */
5       private void rehash( )
6       {
7           HashEntry [ ] oldArray = array;
8
9               // Create a new, empty table
10          allocateArray( nextPrime( 4 * size( ) ) );
11          currentSize = 0;
12          occupied = 0;
13
14              // Copy table over
15          for( int i = 0; i < oldArray.length; i++ )
16              if( isActive( oldArray, i ) )
17                  add( (AnyType) oldArray[ i ].element );
18      }
```

10. The *getMatch* method essentially searches for a value, *x* and returns it if it was found and active. It has always seemed strange to me that this is not part of the Java HashSet interface. Or, that *contains* doesn't return them item. I suppose that if you need that behavior, you could use the HashMap class as it supports the *get* method that returns a value. However, you would have to use a key to use HashMap, which would increase storage requirements.

figure 20.11

The searching routines for a quadratic probing hash table

```
1       /**
2        * This method is not part of standard Java.
3        * Like contains, it checks if x is in the set.
4        * If it is, it returns the reference to the matching
5        * object; otherwise it returns null.
6        * @param x the object to search for.
7        * @return if contains(x) is false, the return value is null;
8        * otherwise, the return value is the object that causes
9        * contains(x) to return true.
10       */
11      public AnyType getMatch( AnyType x )
12      {
13          int currentPos = findPos( x );
14
15          if( isActive( array, currentPos ) )
16              return (AnyType) array[ currentPos ].element;
17          return null;
18      }
19
20      /**
21       * Tests if some item is in this collection.
22       * @param x any object.
23       * @return true if this collection contains an item equal to x.
24       */
25      public boolean contains( Object x )
26      {
27          return isActive( array, findPos( x ) );
28      }
```

20.4.2 – Analysis of Quadratic Probing

1. Quadratic probing has not yet been analyzed mathematically. In quadratic probing, elements that hash to the same position, probe the same cells which is known as *secondary clustering*. Empirical evidence suggests that quadratic probing is close to the no-clustering case.

20.5 – Separate Chaining Hashing

1. A popular and space-efficient alternative to quadratic probing is *separate chaining hashing* in which an array of linked lists is maintained. Thus, the hash function tells us which linked list to insert an item in, or which linked list to find an item in.
2. The appeal of separate chaining is that performance is not affected by a moderately increasing load factor. The Java API uses separate chaining hashing with a default load factor of 0.75.

Homework 20.4

1. Problem 20.5 b in text.
2. Problem 20.6 b in text.

20.6 – Hash Tables vs. Binary Search Trees

1. BST provides order at a complexity of $O(\log n)$; HT does not provide order but has a complexity of $O(1)$. Also, there is not an efficient way with HT to find the minimum, or other order statistics, nor to find a string unless the exact string is known. A BST can quickly find all items in a certain range.

20.7 – Hash Tables Applications

1. Compilers use hash tables to keep track of variables in source code. This data structure is called a *symbol table.*
2. Game programs commonly use a *transposition table* to keep track of different lines of play that it has already encountered.
3. Online spelling checkers can use a pre-built hash table of all the words in a dictionary. Thus, it only takes constant time to check to see if a word is misspelled.
4. Associative arrays are arrays that use strings (or other complicated objects) as indices. These are usually implemented with hash tables.
5. Some languages such as JavaScript, Python, and Ruby, implement objects with hash tables. The keys are the names of the class members and the values are points to the actual value.
6. Caches are frequently implemented as hash tables. A cache is use to speed up access to frequently used items.
7.

Supplemental – Hashing Custom Objects

1. This information discusses how to hash custom objects and provides an example. The source of this material is:

   http://www.idevelopment.info, by Jeffrey M. Hunter (The *programming/java* section has a lot of examples of Java techniques organized by category)

   http://www.idevelopment.info/data/Programming/java/object_oriented_techniques/HashCodeExample.java

   Keep in mind that two equal objects must return the same integer (hashcode). This is not a problem if the same class constructs the two equal objects. Both objects will have the same hashCode() method and hence, return the same integer. You may have a problem if you are trying to be smarter and force two objects from two different classes as being equal. Then, you must ensure that the hashCode() method of both classes returns the same integer.

   In a more complex world, hash codes that you return are supposed to be well-distributed over the range of possible integers. This reduces collisions and makes hash tables fast (by reducing chains/linked-lists). Remember that hash codes do not have to be unique. (It is not possible to guarantee that any way.)

   If you find the default hashCode() implementation based on the object identity too restrictive and returning a constant integer all the time too anti-performance, you can base a hashCode()on the data field values of the object. Beware though, that for mutable classes, a hashtable can lose track of keys if the data fields of the object used as a key are changed.

   So, If you insist on implementing your own hashCode() based on the data field values, can you make your class immutable? Just make all data fields private which can only be initialized once through the class constructor. Then, don't provide any setter methods or methods which change their values. Same thing in implementation of objects used as data fields of this class. If no one can change the data fields, the hash code will always remain the same.

If your class is immutable (the instance data cannot be modified once initialized), you can base the hash code on the data field values. You should even calculate the hashCode() just once for an instance (after all, no data is going to change after the object has been instantiated - the class is immutable) and store it in a private instance variable. Next time onwards, the hashCode() method can just return the private variable's value, making the implementation very fast.

Immutable classes may not be a practical solution though, for many cases. Most custom classes have non-private data or setter methods and MUST alter instance variables.

Anyway, immutable or not, here are some of the ways to get a custom hash code based on the data field values (apart from returning 0 or a constant integer discussed earlier which is not based on data fields).

The default hashCode() implementation on Sun's SDK returns a machine address.

```
class Team {

    private static final int HASH_PRIME = 1000003;
    private              String name;
    private              int wins;
    private              int losses;

    public Team(String name) {
        this.name = name;
    }

    public Team(String name, int wins, int losses) {
        this.name = name;
        this.wins = wins;
        this.losses = losses;
    }
```

```
/**
     * this overrides equals() in java.lang.Object
     */
    public boolean equals(Object obj) {
        /**
         * return true if they are the same object
         */
        if (this == obj)
            return true;

        /**
         * the following two tests only need to be performed
         * if this class is directly derived from java.lang.Object
         */
        if (obj == null || obj.getClass() != getClass())
            return false;

        // we know obj is of type Team
        Team other = (Team)obj;

        // now test all pertinent fields ...
        if (wins != other.wins || losses!= other.losses) {
            return false;
        }

        if (!name.equals(other.name)) {
            return false;
        }

        // otherwise they are equal
        return true;
    }


 /**

     * This overrides hashCode() in java.lang.Object
     */
    public int hashCode() {
        int result = 0;

        result = HASH_PRIME * result + wins;
        result = HASH_PRIME * result + losses;
        result = HASH_PRIME * result + name.hashCode();

        return result;
    }
```