CHAPTER 5

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

HASH TABLE ADT

- The implementation of hash tables is frequently called hashing.
- Insertion, deletion and searches are performed in constant average time.
- Tree operations that require any ordering information among the elements are not supported efficiently.

HASH TABLE

- The ideal hash table data structure is an array of fixed size, containing the items.
- TableSize: size of the hash table
- Hash function is defined as mapping of keys into some number in the range 0 to TableSize – 1. Then data is placed in the appropriate cell .
- Collision : two keys hash to the same value

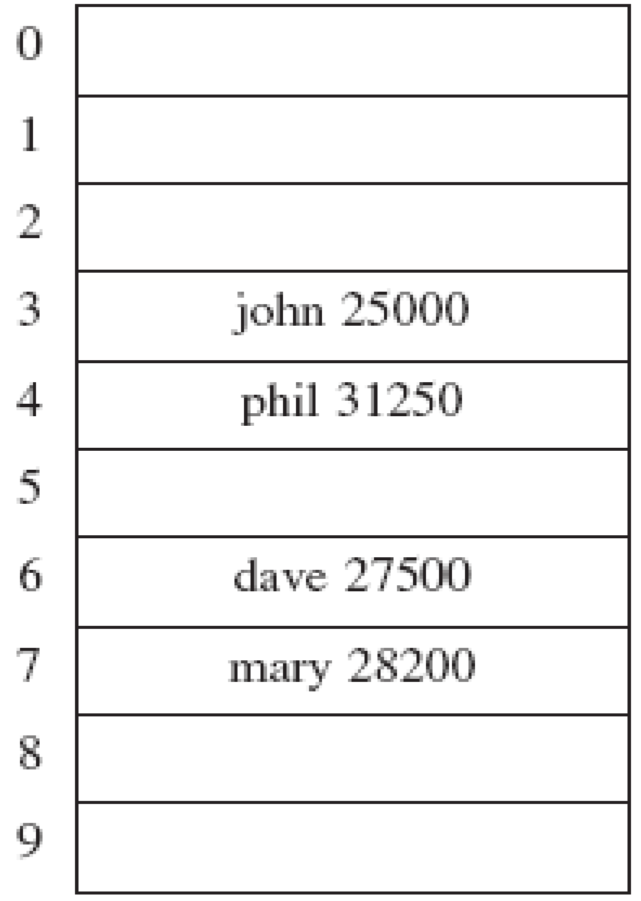


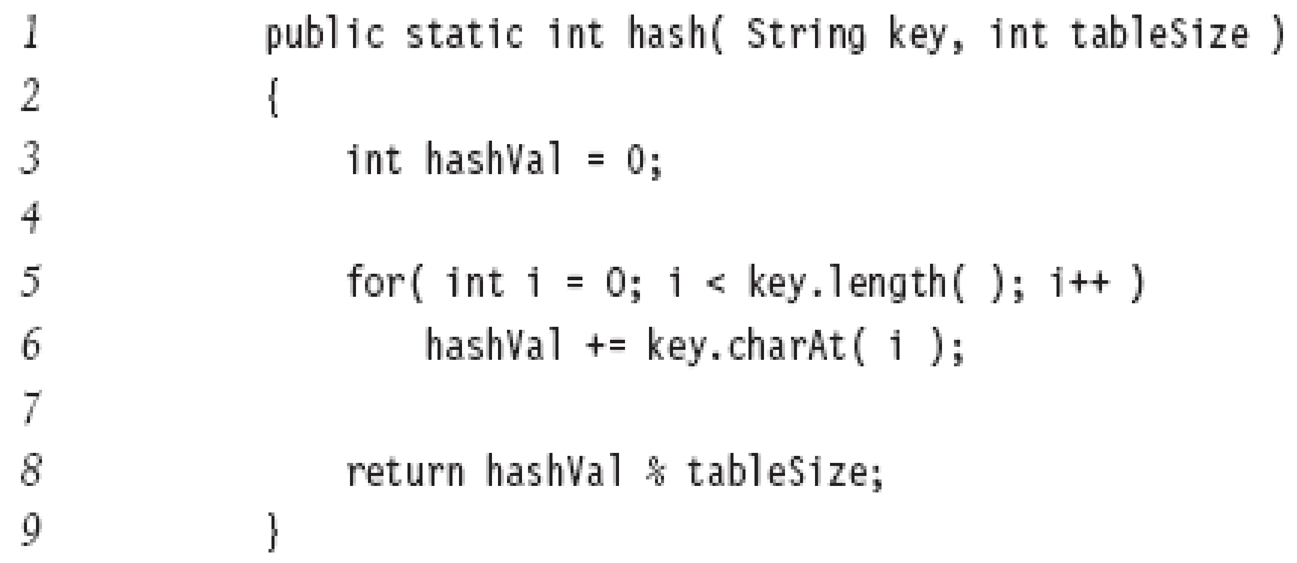
Figure 5.1 An ideal hash table

HASH FUNCTION

- Integer key
  - key mod TableSize is one choice.
  - Keeping TableSize as prime will distribute the keys more evenly.

- String Key
  - add up the Unicode values of the characters

HASH FUNCTION FOR STRING KEYS

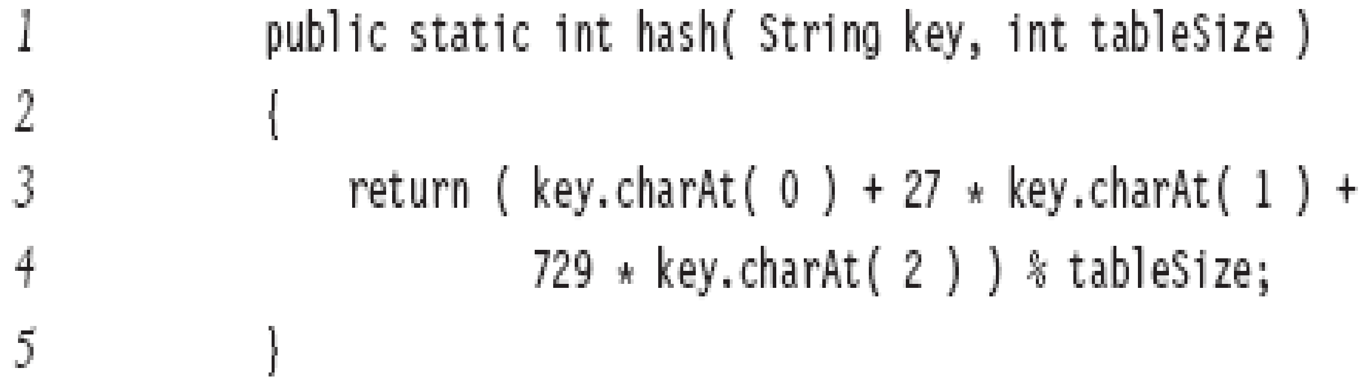


```java
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal += key.charAt( i );

    return hashVal % tableSize;
}
```

Figure 5.2 A simple hash function

The hash function can assume values between 0 and 127 * 8 = 1016 which is less if the TableSize = 10007

HASH FUNCTION



```
public static int hash( String key, int tableSize )
{
    return ( key.charAt( 0 ) + 27 * key.charAt( 1 ) +
                729 * key.charAt( 2 ) ) % tableSize;
}
```

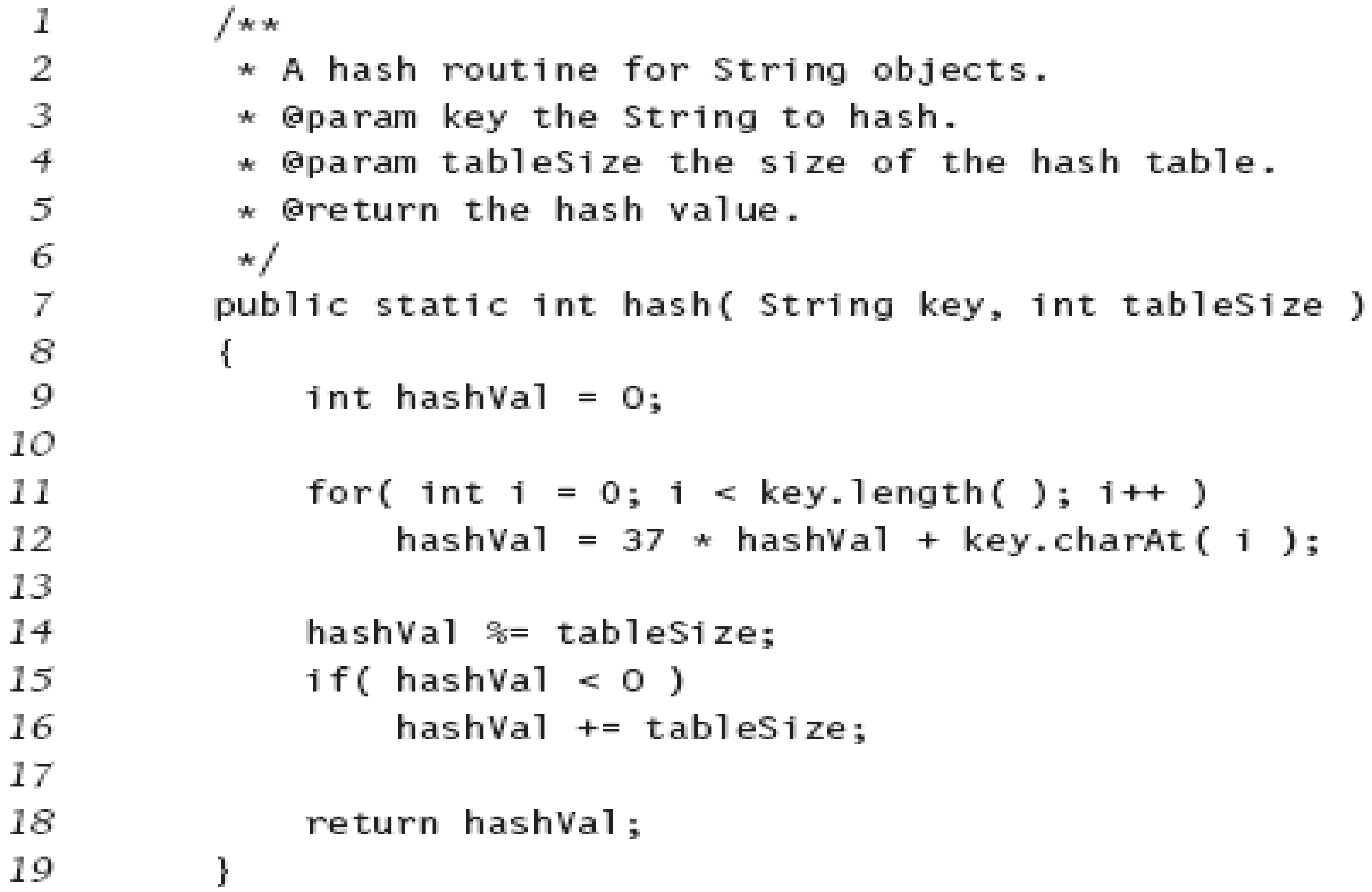
Figure 5.3 Another possible hash function—not too good

If keys are random we could expect a reasonable distribution ($26^3 = 17,578$)
But in English language number of different combination is only 2 851 (28 % of the TableSize)

GOOD HASH FUNCTION

- $\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] * \ 37^{i}$
- The code computes a polynomial function of 37 by use of Horner’s rule.
- A common practice is not to use all the characters, some programmers implement their hash function by using only the characters in the odd spaces.

GOOD HASH FUNCTION



```
/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

Figure 5.4 A good hash function

SEPARATE CHAINING

- To keep a list of all elements that hash to the same value.
- For example let the keys be first 10 perfect squares and hash function is

  hash(x) = x mod 10
- For insert determine the location in the hash table using the hash function then
  - check whether the element is already in the list if not insert at the front of the list

SEPARATE CHAINING

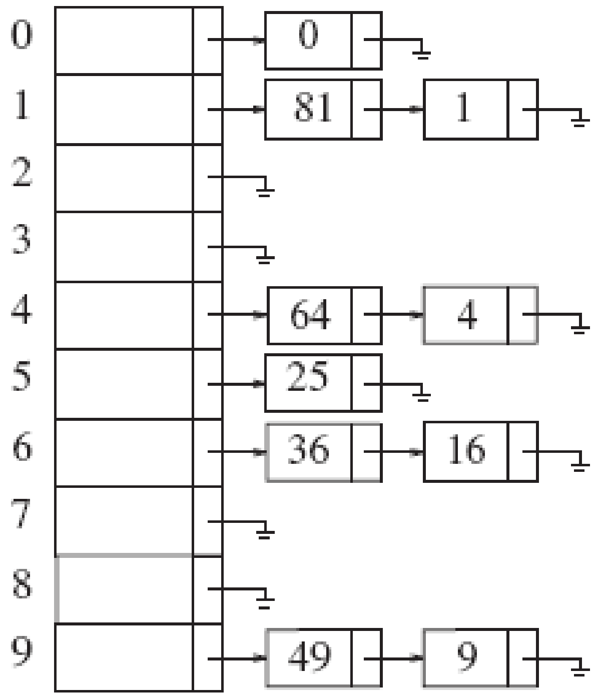
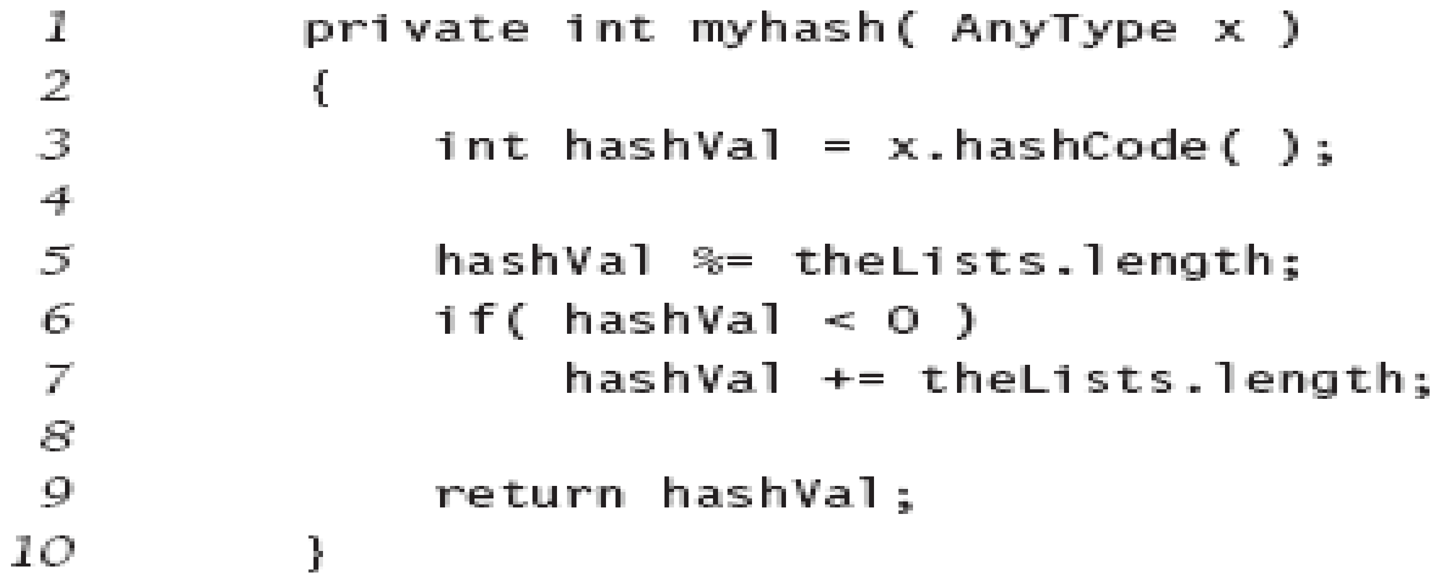


Figure 5.5 A separate chaining hash table

```
public class SeparateChainingHashTable<AnyType>
{
    public SeparateChainingHashTable( )
      { /* Figure 5.9 */ }
    public SeparateChainingHashTable( int size )
      { /* Figure 5.9 */ }

    public void insert( AnyType x )
      { /* Figure 5.10 */ }
    public void remove( AnyType x )
      { /* Figure 5.10 */ }
    public boolean contains( AnyType x )
      { /* Figure 5.10 */ }
    public void makeEmpty( )
      { /* Figure 5.9 */ }

    private static final int DEFAULT_TABLE_SIZE = 101;

    private List<AnyType> [ ] theLists;
    private int currentSize;

    private void rehash( )
      { /* Figure 5.22 */ }
    private int myhash( AnyType x )
      { /* Figure 5.7 */ }

    private static int nextPrime( int n )
      { /* See online code */ }
    private static boolean isPrime( int n )
      { /* See online code */ }
}
```

Figure 5.6 Class skeleton for separate chaining hash table

ROUTINE: *myhash*



```
private int myhash( AnyType x )
{
    int hashVal = x.hashCode( );

    hashVal %= theLists.length;
    if( hashVal < 0 )
        hashVal += theLists.length;

    return hashVal;
}
```

Figure 5.7 myHash method for hash tables

The hascode() routine returns an int number that is converted to suitable array index

For Example

```java
public class Employee
{
    public boolean equals( Object rhs )
      { return rhs instanceof Employee && name.equals( ((Employee)rhs).name ); }

    public int hashCode( )
      { return name.hashCode( ); }

    private String name;
    private double salary;
    private int seniority;

      // Additional fields and methods
}
```

Figure 5.8 Example of `Employee` class that can be in a hash table

```java
    /**
     * Construct the hash table.
     */
    public SeparateChainingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /**
     * Construct the hash table.
     * @param size approximate table size.
     */
    public SeparateChainingHashTable( int size )
    {
        theLists = new LinkedList[ nextPrime( size ) ];
        for( int i = 0; i < theLists.length; i++ )
            theLists[ i ] = new LinkedList<>( );
    }

    /**
     * Make the hash table logically empty.
     */
    public void makeEmpty( )
    {
        for( int i = 0; i < theLists.length; i++ )
            theLists[ i ].clear( );
        currentSize = 0;
    }
```

Figure 5.9 Constructors and `makeEmpty` for separate chaining hash table

```
    /**
     * Find an item in the hash table.
     * @param x the item to search for.
     * @return true if x is not found.
     */
    public boolean contains( AnyType x )
    {
        List<AnyType> whichList = theLists[ myhash( x ) ];
        return whichList.contains( x );
    }

    /**
     * Insert into the hash table. If the item is
     * already present, then do nothing.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        List<AnyType> whichList = theLists[ myhash( x ) ];
        if( !whichList.contains( x ) )
        {
            whichList.add( x );

                // Rehash; see Section 5.5
            if( ++currentSize > theLists.length )
                rehash( );
        }
    }

    /**
     * Remove from the hash table.
     * @param x the item to remove.
     */
    public void remove( AnyType x )
    {
        List<AnyType> whichList = theLists[ myhash( x ) ];
        if( whichList.contains( x ) )
        {
            whichList.remove( x );
            currentSize--;
        }
    }
```

Figure 5.10 contains, insert, and remove routines for separate chaining hash table

COLLISIONS

- Any scheme other than linked list can be used resolve collision, a binary search tree or another hash table.
- If the table is large and the hash function is good, all the list should be short.
- Load Factor λ ratio of the number of elements to tableSize
- Unsuccessful research examined λ nodes on average
- Successful search examines λ /2 + 1 on average

HASH TABLE WITHOUT LINKED LIST

- An alternative to resolving collisions with linked list is to try alternative cells until an empty cell is found.

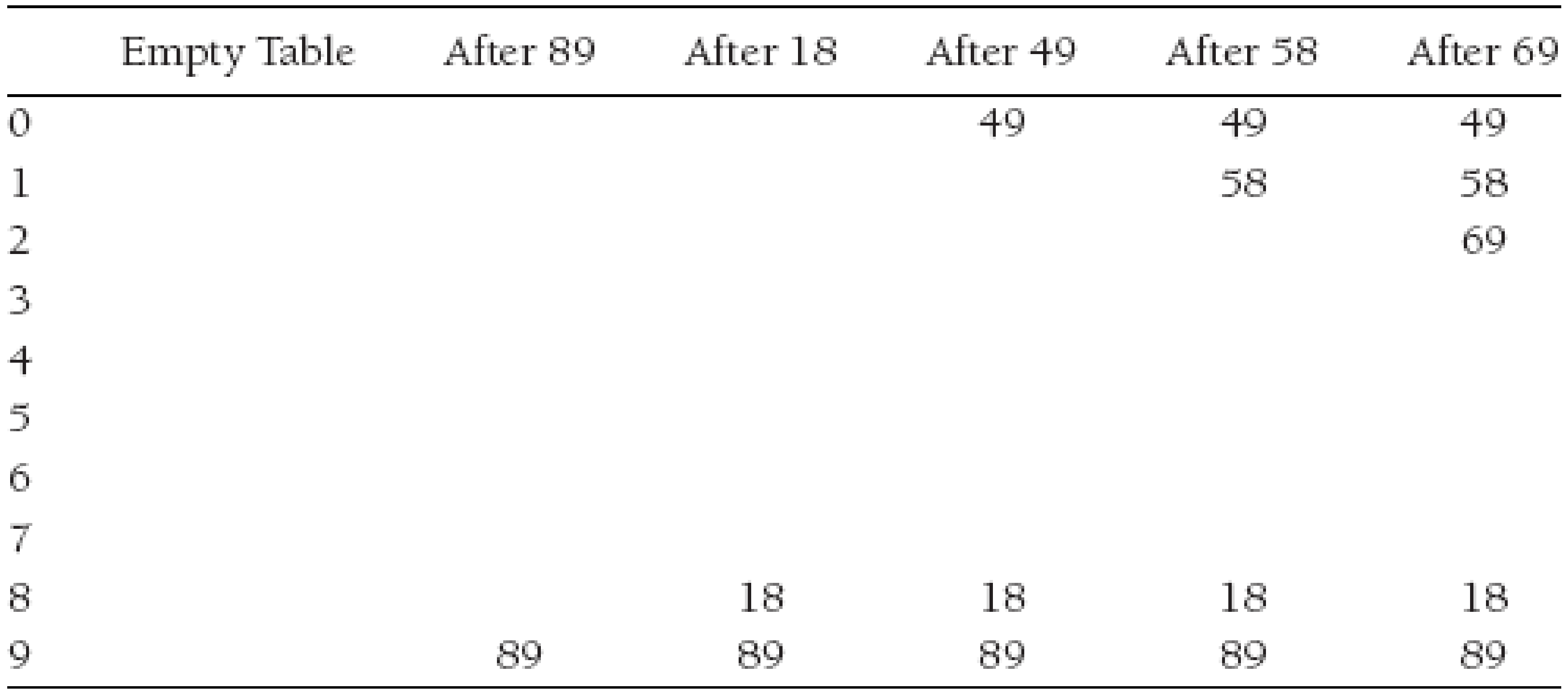
$$h_i = (\text{hash}(x) + f(i)) \bmod \text{TableSize}, \text{ with } f(0) = 0.$$

- The function f, is the collision resolution strategy.
- The load factor λ for such tables should be below 0.5
- Such tables are called probing hash tables.

LINEAR PROBING

- In linear probing f is an linear function of i, for example f(i) = i
- This amounts to trying cells sequentially in search of an empty cell.
- For example inserting keys { 89, 18, 49, 58, 69} into hash table with same hash function as before.
- As long as the table is big enough, a free cell can always be found.
- Disadvantage is primary clustering, where even when the table is relatively empty, blocks of occupied cells start forming.

LINEAR PROBING



	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

LINEAR PROBING

- Expected number of probes in an unsuccessful search is 1/(1- λ).
- The number of probes for a successful search is equal to the number of probes when the particular element was inserted.

$$I(\lambda) = \frac{1}{\lambda} \int_0^{\lambda} \frac{1}{1-x}\, dx = \frac{1}{\lambda} ln \frac{1}{1-\lambda}$$

LINEAR PROBING

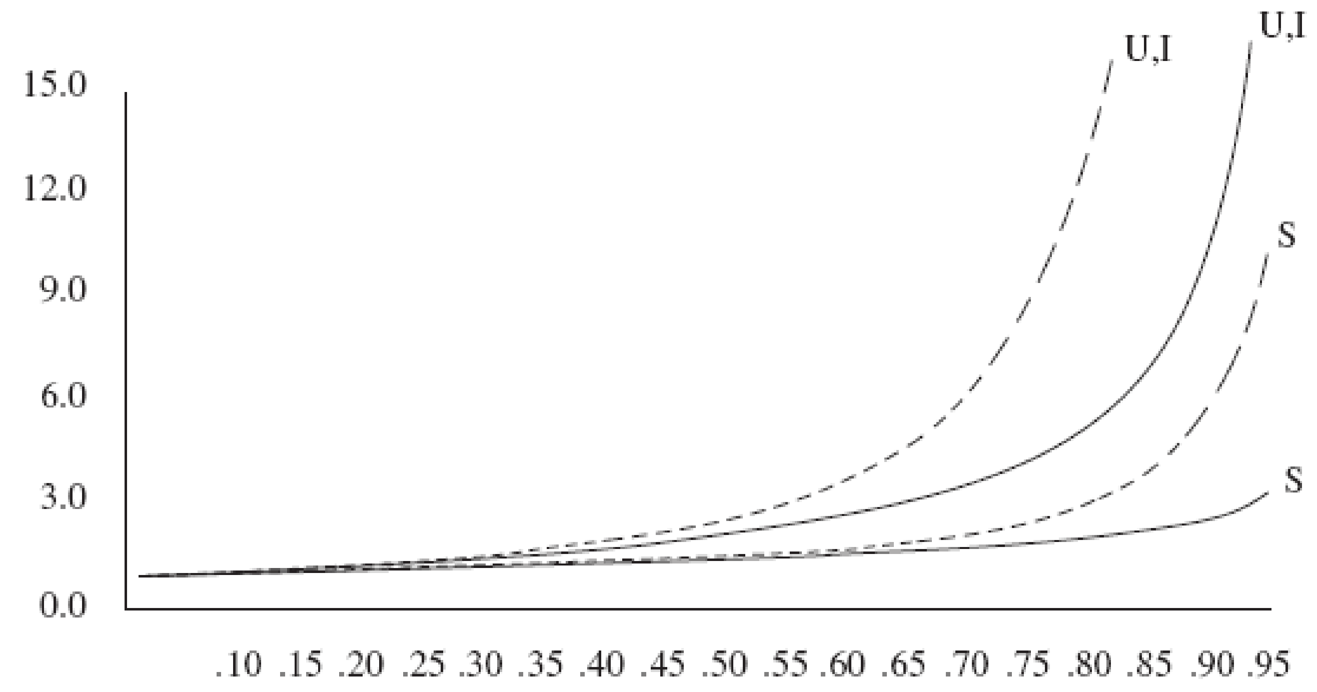
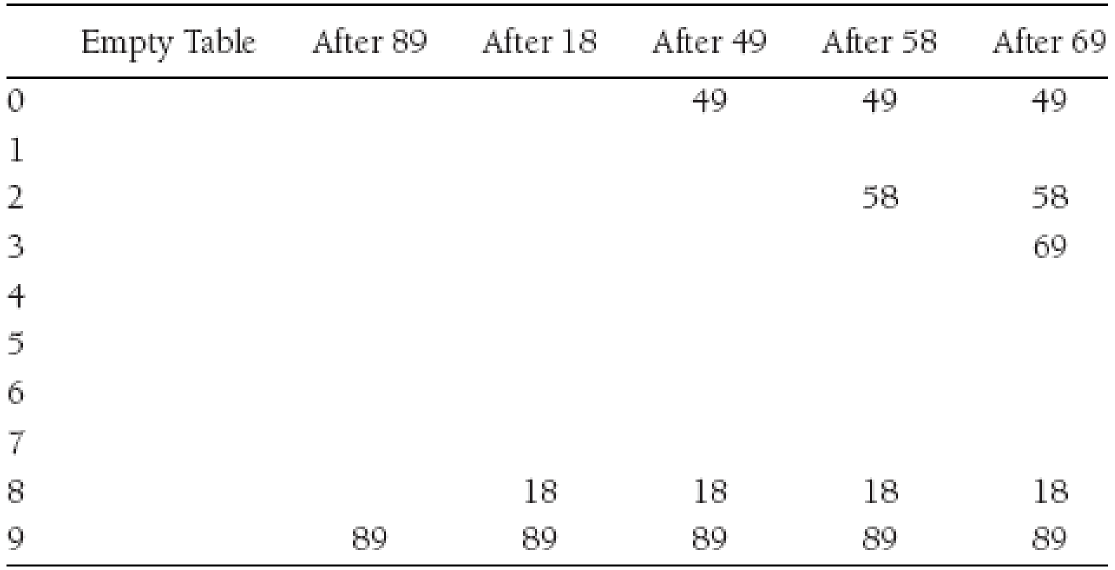


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full.

QUADRATIC PROBING

- The collision function is quadratic $f(i) = i^2$
- This eliminates primary clustering problem of linear probing.



	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Hash table with quadratic probing, after each insertion

QUADRATIC PROBING

- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- In case of quadratic probing the situation is even more drastic. There is no guarantee of finding an empty cell once the table gets half full. This is because at most half of the table can be used as alternative locations to resolve collisions.

DELETION

- Standard deletion cannot be performed in a probing hash table, because cell might have caused collision to go past it.
- Probing hash tables require lazy deletion.
- Each entry in the array of HashEntry references is either
  1. null
  2. Not null and the entry is active (isActive is true)
  3. Not null and the entry is marked deleted(isActive is false)

```
public class QuadraticProbingHashTable<AnyType>
{
    public QuadraticProbingHashTable( )
      { /* Figure 5.15 */ }
    public QuadraticProbingHashTable( int size )
      { /* Figure 5.15 */ }
    public void makeEmpty( )
      { /* Figure 5.15 */ }

    public boolean contains( AnyType x )
      { /* Figure 5.16 */ }
    public void insert( AnyType x )
      { /* Figure 5.17 */ }
    public void remove( AnyType x )
      { /* Figure 5.17 */ }

    private static class HashEntry<AnyType>
    {
        public AnyType  element;   // the element
        public boolean isActive;   // false if marked deleted

        public HashEntry( AnyType e )
          { this( e, true ); }

        public HashEntry( AnyType e, boolean i )
          { element  = e; isActive = i; }
    }

    private static final int DEFAULT_TABLE_SIZE = 11;

    private HashEntry<AnyType> [ ] array; // The array of elements
    private int currentSize;              // The number of occupied cells

    private void allocateArray( int arraySize )
      { /* Figure 5.15 */ }
    private boolean isActive( int currentPos )
      { /* Figure 5.16 */ }
    private int findPos( AnyType x )
      { /* Figure 5.16 */ }
    private void rehash( )
      { /* Figure 5.22 */ }

    private int myhash( AnyType x )
      { /* See online code */ }
    private static int nextPrime( int n )
      { /* See online code */ }
    private static boolean isPrime( int n )
      { /* See online code */ }
}
```

Figure 5.14 Class skeleton for hash tables using probing strategies, including the nested `HashEntry` class

```java
    /**
     * Construct the hash table.
     */
    public QuadraticProbingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /**
     * Construct the hash table.
     * @param size the approximate initial size.
     */
    public QuadraticProbingHashTable( int size )
    {
        allocateArray( size );
        makeEmpty( );
    }

    /**
     * Make the hash table logically empty.
     */
    public void makeEmpty( )
    {
        currentSize = 0;
        for( int i = 0; i < array.length; i++ )
            array[ i ] = null;
    }

    /**
     * Internal method to allocate array.
     * @param arraySize the size of the array.
     */
    private void allocateArray( int arraySize )
    {
        array = new HashEntry[ nextPrime( arraySize ) ];
    }
```

Figure 5.15 Routines to initialize hash table

```java
    /**
     * Find an item in the hash table.
     * @param x the item to search for.
     * @return the matching item.
     */
    public boolean contains( AnyType x )
    {
        int currentPos = findPos( x );
        return isActive( currentPos );
    }

    /**
     * Method that performs quadratic probing resolution in half-empty table.
     * @param x the item to search for.
     * @return the position where the search terminates.
     */
    private int findPos( AnyType x )
    {
        int offset = 1;
        int currentPos = myhash( x );

        while( array[ currentPos ] != null &&
                !array[ currentPos ].element.equals( x ) )
        {
            currentPos += offset;  // Compute ith probe
            offset += 2;
            if( currentPos >= array.length )
                currentPos -= array.length;
        }

        return currentPos;
    }

    /**
     * Return true if currentPos exists and is active.
     * @param currentPos the result of a call to findPos.
     * @return true if currentPos is active.
     */
    private boolean isActive( int currentPos )
    {
        return array[ currentPos ] != null && array[ currentPos ].isActive;
    }
```

Figure 5.16 contains routine (and private helpers) for hashing with quadratic probing

```
    /**
     * Insert into the hash table. If the item is
     * already present, do nothing.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
            // Insert x as active
        int currentPos = findPos( x );
        if( isActive( currentPos ) )
            return;

        array[ currentPos ] = new HashEntry<>( x, true );

            // Rehash; see Section 5.5
        if( ++currentSize > array.length / 2 )
            rehash( );
    }

    /**
     * Remove from the hash table.
     * @param x the item to remove.
     */
    public void remove( AnyType x )
    {
        int currentPos = findPos( x );
        if( isActive( currentPos ) )
            array[ currentPos ].isActive = false;
    }
```

Figure 5.17 insert routine for hash tables with quadratic probing

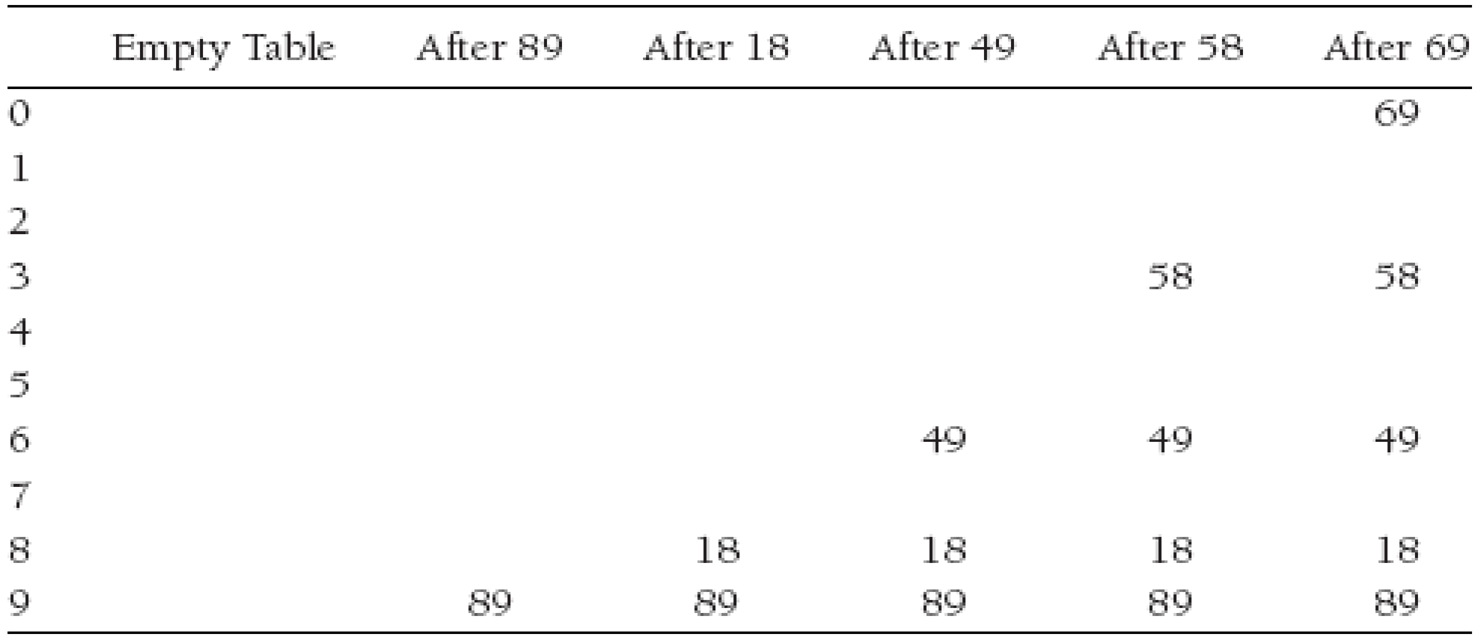
ROUTINE: *contains()*

- contains(x) invokes private methods isActive and findPos.
- findPos performs the collision resolution.
  - Quadratic resolution function is F(i) = f(i-1) +2i -1
  - If the location is past the array subtract it by TableSize

- Although quadratic probing eliminates primary clustering it introduces secondary clustering. But simulations show this is a theoretical blemish.

DOUBLE HASHING

- For double hashing one popular choice is F(i) = i. $hash_2(x)$
- A function such as $hash_2(x)$ = R – (x mod R) with R a prime smaller than TableSize is a good choice.



	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

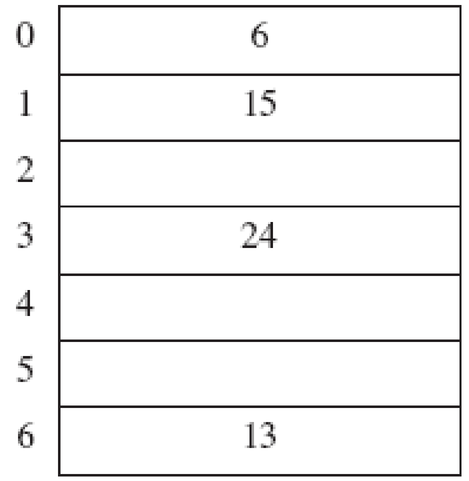
Figure 5.18 Hash table with double hashing, after each insertion

TableSize should be prime

- it is important for the table size to be prime when using double hashing.
- Excepted number of probes become same as random collision resolution strategy If the TableSize is prime.
- Quadratic probing however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice, especially for keys like string whose hash function are expensive to compute.

REHASHING

- Building a second table (twice as big) and computing the new hash value for each element and inserting into the new table is called rehashing.

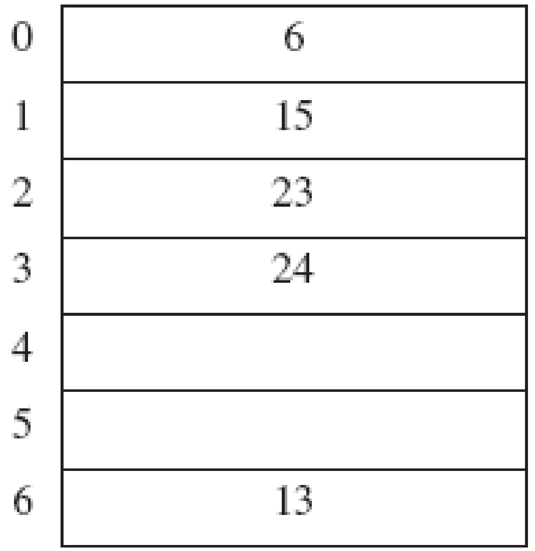


0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.19 Hash table with linear probing with input 13, 15, 6, 24

REHASHING

- If 23 inserted into the table, the resulting table will be over 70% full, so a new table is created with TableSize = 17



0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 5.20 Hash table with linear probing after 23 is inserted

This entire operation is called rehashing. This is very expensive O(N) but it happens very infrequently.

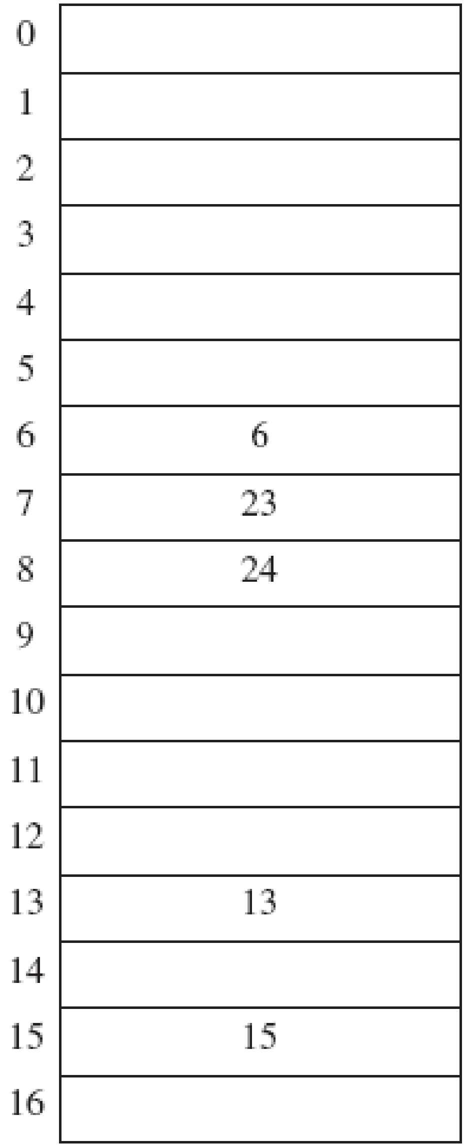


Figure 5.21 Linear probing hash table after rehashing

```
/**
 * Rehashing for quadratic probing hash table.
 */
private void rehash( )
{
    HashEntry<AnyType> [ ] oldArray = array;

        // Create a new double-sized, empty table
    allocateArray( nextPrime( 2 * oldArray.length ) );
    currentSize = 0;

        // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( oldArray[ i ] != null && oldArray[ i ].isActive )
            insert( oldArray[ i ].element );
}

/**
 * Rehashing for separate chaining hash table.
 */
private void rehash( )
{
    List<AnyType> [ ]  oldLists = theLists;

        // Create new double-sized, empty table
    theLists = new List[ nextPrime( 2 * theLists.length ) ];
    for( int j = 0; j < theLists.length; j++ )
        theLists[ j ] = new LinkedList<>( );

        // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldLists.length; i++ )
        for( AnyType item : oldLists[ i ] )
            insert( item );
}
```

Figure 5.22 Rehashing for both separate chaining and probing hash tables