Figure 8.5

Shellsort after each pass if the increment sequence is {1, 3, 5}

ORIGINAL	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

ShellSort

```
public static void shellsort( Comparable [ ] a )
  {
     for( int gap = a.length / 2; gap > 0;
                gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
        for( int i = gap; i < a.length; i++ )
        {
           Comparable tmp = a[ i ];
           int j = i;

           for( ; j >= gap && tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
              a[ j ] = a[ j - gap ];
           a[ j ] = tmp;
        }
  }
```

Figure 8.10 Quicksort

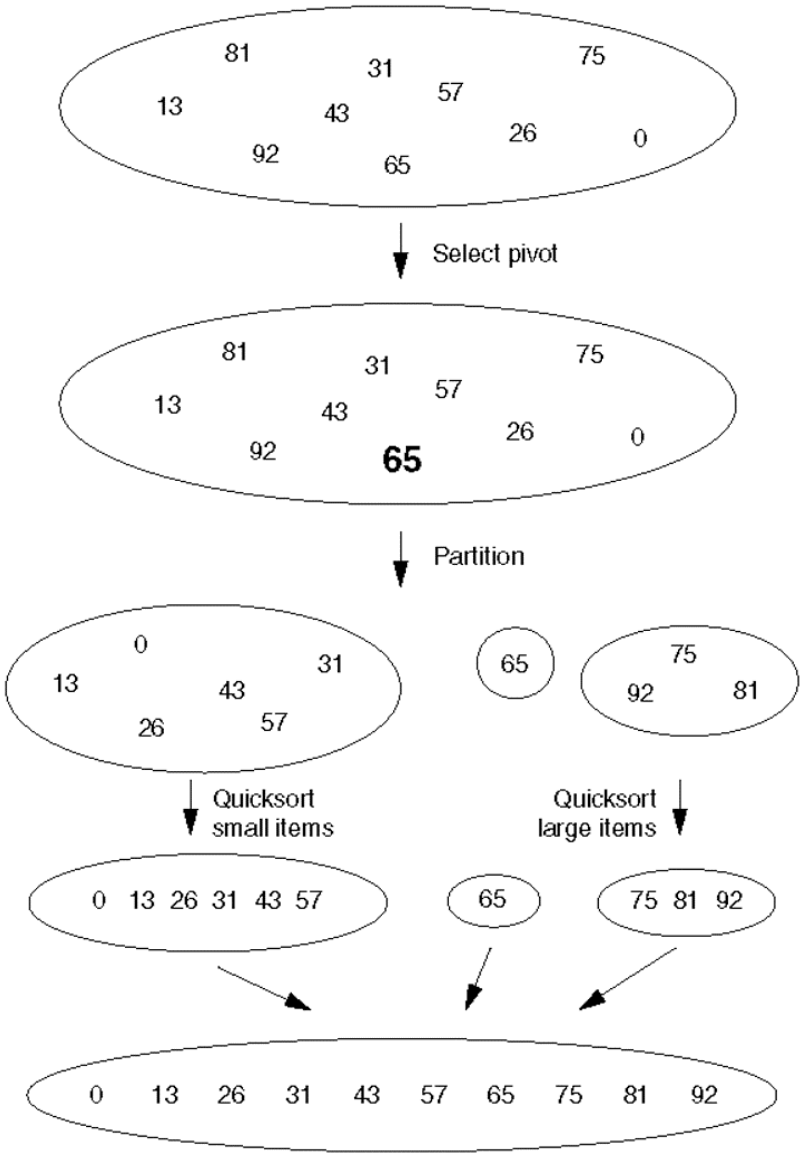


Figure 8.11 Partitioning algorithm: Pivot element 6 is placed at the end.

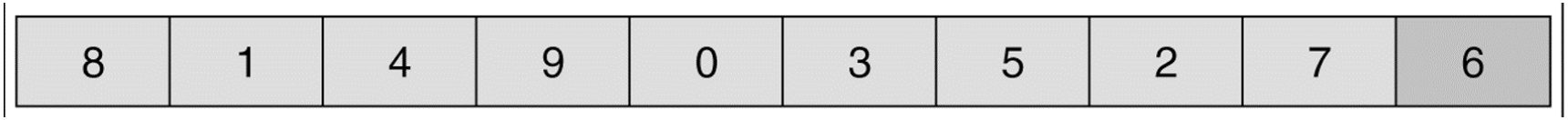


Figure 8.12 Partitioning algorithm: i stops at large element 8; j stops at small element 2.

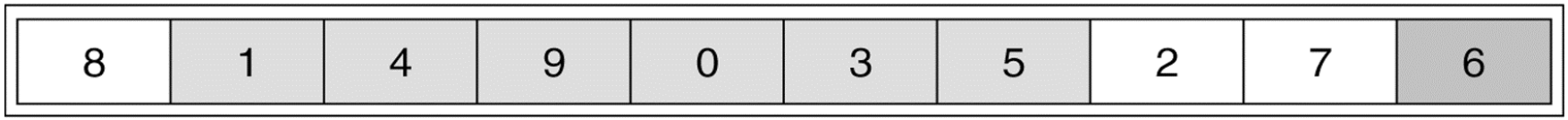


Figure 8.13 Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

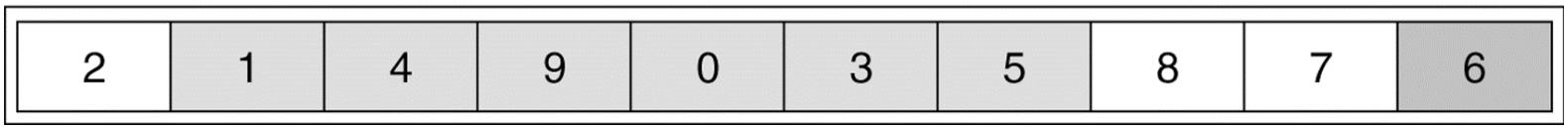


Figure 8.14 Partitioning algorithm: i stops at large element 9; j stops at small element 5.

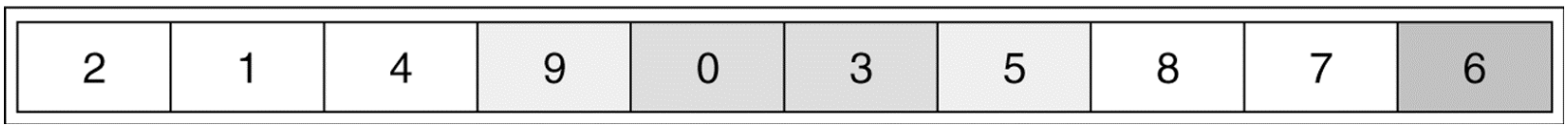


Figure 8.15 Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

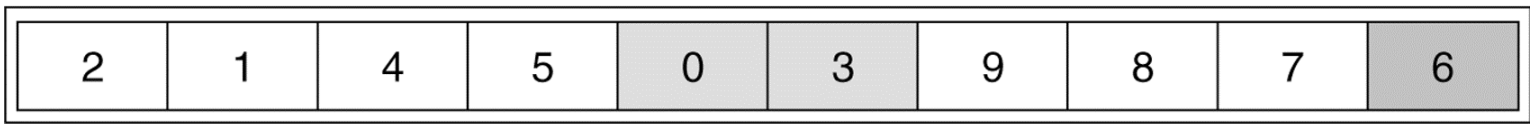


Figure 8.16 Partitioning algorithm: i stops at large element 9; j stops at small element 3.

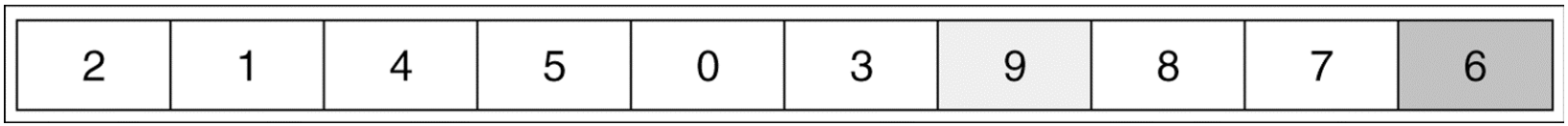


Figure 8.17 Partitioning algorithm: Swap pivot and element in position i.

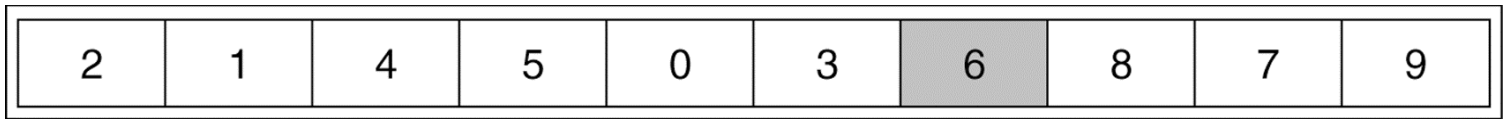


Figure 8.18 Original array

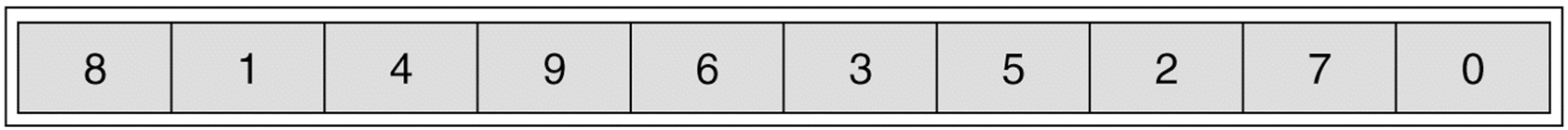


Figure 8.19 Result of sorting three elements (first, middle, and last)

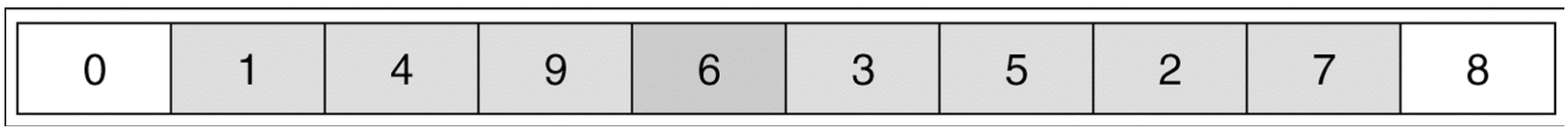
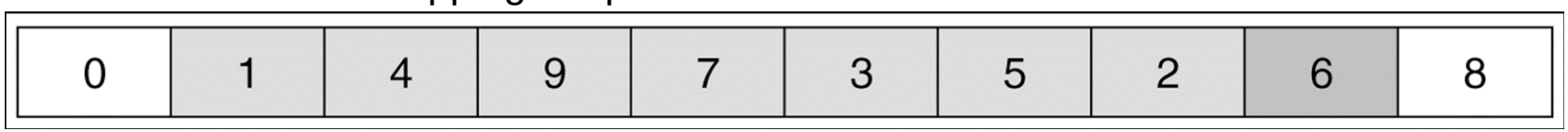


Figure 8.20 Result of swapping the pivot with the next-to-last element



Quicksort

```
public static void quicksort( Comparable [ ] a ) {  quicksort( a, 0, a.length - 1 );  }
private static void quicksort( Comparable [ ] a, int low, int high )
   {
      if( low + CUTOFF > high )  insertionSort( a, low, high );
      else  {  // Sort low, middle, high
         int middle = ( low + high ) / 2;
         if( a[ middle ].compareTo( a[ low ] ) < 0 )    swapReferences( a, low, middle );
         if( a[ high ].compareTo( a[ low ] ) < 0 )      swapReferences( a, low, high );
         if( a[ high ].compareTo( a[ middle ] ) < 0 )   swapReferences( a, middle, high );
         swapReferences( a, middle, high - 1 );        // Place pivot at position high - 1
         Comparable pivot = a[ high - 1 ];
         int i, j; // Begin partitioning
         for( i = low, j = high - 1; ; ) {
            while( a[ ++i ].compareTo( pivot ) < 0 ) /* Do nothing */     ;
            while( pivot.compareTo( a[ --j ] ) < 0 ) /* Do nothing */     ;
            if( i >= j )    break;
            swapReferences( a, i, j );
         }
        swapReferences( a, i, high - 1 );
         quicksort( a, low, i - 1 );    // Sort small elements
         quicksort( a, i + 1, high );   // Sort large elements
      }
   }
```

Collection

```
// Fig 6.9, 6.10, pg 192, 194.
package weiss.util;

public interface Collection extends java.io.Serializable
{
   int size( );
   boolean isEmpty( );
   boolean contains( Object x ); boolean containsAll(Collection c);
   boolean add( Object x ); boolean addAll(Collection c);
   boolean remove( Object x );  boolean removeAll(Collection c);
   void clear( );
   Iterator iterator( );
   int hashCode();
   Object [ ] toArray( ); Object[] toArray(Object[]);
}
```

Set & SortedSet

- A set is a container that contains no duplicates.
- It extends the Collection methods.

```
public interface Set extends Collection
{
}
```

```
public interface SortedSet extends Set // page 210
{
 Comparator comparator() ;
 Object first() ;
 Object last() ;
 SortedSet subSet(Object fromElement, Object toElement);
            // elements in range from fromElement, inclusive, to toElement, exclusive.
  SortedSet headSet(Object toElement) ; //returns items smaller than toElement
  SortedSet tailSet(Object fromElement); // returns items greater or equal
}
```

TreeSet

- Implements SortedSet using balanced BST

```
 // page 211
public static void main( String [ ] args )
    {
       // new TreeSet uses specified comparator instead of default
        Set s = new TreeSet( Collections.reverseOrder( ) );
        s.add( "joe" );
        s.add( "bob" );
        s.add( "hal" );
        printCollection( s );    // Figure 6.26
    }
```

HashSet

- implements Set
- Elements must have a hashCode method implemented

```
 // page 212
public static void main( String [ ] args )
    {
        Set s = new HashSet( );
        s.add( "joe" );
        s.add( "bob" );
        s.add( "hal" );
        printCollection( s );    // Figure 6.27
    }
```

Maps

- Map is used to store <**Key**, **Value**> pairs.
- It, therefore, maps **Key** to **Value**.
- **Key**s must be unique. **Value**s need not be unique.
- Implemented using HashMap or TreeMap.

```
public interface Map {
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Object get(Object key);   // Returns value to which key is mapped
    Object put(Object key, Object value) ;
    Object remove(Object key) ;
    Set entrySet();  //  Returns a set view of the mappings contained in this map.
    Collection values();    // Returns collection of values in this map.
     Set keySet();  //  Returns a set view of the keys contained in this map.
     int size();
     boolean isEmpty(); }
```

MapDemo

```
public static void main( String [ ] args )
   {
      Map phone1 = new TreeMap( );

      phone1.put( "John Doe", "212-555-1212" );
      phone1.put( "Jane Doe", "312-555-1212" );
      phone1.put( "Holly Doe", "213-555-1212" );

      System.out.println( "phone1.get(\"Jane Doe\"): " +
                          phone1.get( "Jane Doe" ) );
      System.out.println( );

      printMap( "phone1", phone1 );
   }
}
```

```
public class HashMap extends MapImpl
{  public HashMap( );
   public HashMap( Map other );
   protected Map.Entry makePair( Object key, Object value );
   protected Set makeEmptyKeySet( );
   protected Set clonePairSet( Set pairSet );
   private static final class Pair implements Map.Entry
   {  public Pair( Object k, Object v )
      public Object getKey( )
      public Object getValue( )
      public int hashCode( )
      public boolean equals( Object other )
      private Object key;
      private Object value;
   }
}
```

```
public class TreeMap extends MapImpl
{  public TreeMap( );
   public TreeMap( Map other );
   public TreeMap( Comparator cmp );
   protected Map.Entry makePair( Object key, Object value );
   protected Set makeEmptyKeySet( );
   protected Set clonePairSet( Set pairSet );
   private static final class Pair implements Map.Entry
   {  public Pair( Object k, Object v )
      public Object getKey( )
      public Object getValue( )
      public int compareTo( Object other)
      private Object key;
      private Object value;
   }
}
```

Priority Queue

```
public interface PriorityQueue
{
    public interface Position
    {
        Comparable getValue( );
    }
    Position insert( Comparable x );
    Comparable findMin( );
    Comparable deleteMin( );
    boolean isEmpty( );
    void makeEmpty( );
    int size( );
    void decreaseKey( Position p, Comparable newVal );
}
```

Priority Queue Demo

```
public static void main( String [ ] args )
  {
     PriorityQueue minPQ = new BinaryHeap( );

     minPQ.insert( new Integer( 4 ) );
     minPQ.insert( new Integer( 3 ) );
     minPQ.insert( new Integer( 5 ) );

     dumpPQ( "minPQ", minPQ );
  }
```

Hash Table

- Data Structure for:
  - Insert
  - Search or retrieve
  - Delete
- Very efficient
- Content-based data structure
  - Use value as an index
    - Works if range of values are small
  - Use HASH value as an index
    - Works if HASH function is "good"
- A COLLISION occurs when two values have the same HASH value
- A "good" HASH function is one that causes few or no COLLISIONS.

Simple hash functions

$$hashValue(x) = x \% tableSize$$

- Let tableSize = 100
  - X = 173, hashValue(X) = 73
  - X = 3452, hashValue(X) = 52
  - X = 9758, hashValue(X) = 58
  - X = 800, hashValue(X) = 0

$$hashValue(x) = x_3S^3 + x_2S^2 + x_1S^1 + x_0S^0 \% tableSize$$

- Let S = 128
  - X = comb,
    hashValue(X) = ('c' 128^3 + 'o' 128^2 + 'm' 128^1 + 'b' 128^0)%tableSize
  - X = eye,
    hashValue(X) = ('e' 128^2 + 'y' 128^1 + 'e' 128^0)% tableSize

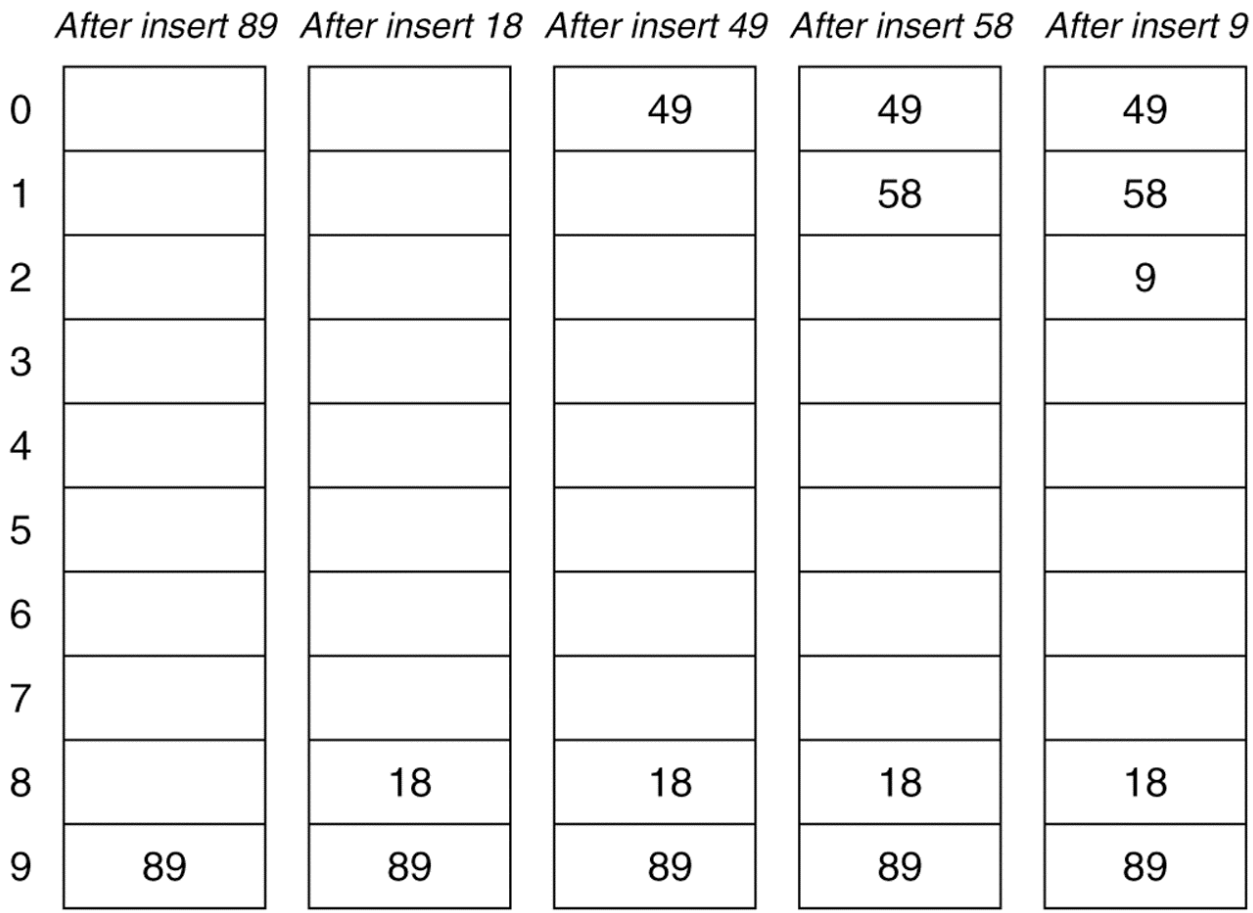
Collision Resolution

- Perfect hash functions: no collisions.
- Perfect hash functions can be built if the input data is known beforehand. But they are difficult to design.
- For perfect hash functions, all operations can be perfromed in O(1) time.
- If input is not known beforehand, then perfect hash functions are impossible to design.
- So collisions are inevitable.
- How to deal with collisions?
- LINEAR PROBING:
  - If the location where an item is to be inserted is already occupied (COLLISION), then scan sequentially until an empty location is found, and insert new item there.

Figure 20.4

Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Problems with Linear Probing

- PRIMARY CLUSTERING
  - Large blocks of occupied cells are formed.
  - Amount of clustering and size of clusters is dependent on LOAD FACTOR (fraction of table that is occupied).
  - It deteriorates the performance.
- NAÏVE ANALYSIS:
  - If load factor is F, and table size is T, then the average time for search is FT.
    - INCORRECT !!
  - If load factor is F, then the average time for search is:
    - $1 + 1/(1-F)^2)/2$
  - If F = 50%, then the average cluster time is 2.5
  - If F = 90%, then the average cluster time is 50.5

Clustering

- Linear Probing leads to primary clustering

- LINEAR PROBING: Try H, H+1, H+2, H+3, …
- QUADRATIC PROBING: Try H, $H+1^2$, $H+2^2$, $H+3^2$, …
  - Seems to eliminate primary clustering
- Linear Probing also leads to secondary clustering
  - This is when large clusters merge to become larger clusters.
  - It is not clear if quadratic probing eliminates it.

- DOUBLE HASHING: Try $H_1(x)$, $H_1(x) + H_2(x)$, $H_1(x) + 2H_2(x)$, $H_1(x) + 3H_2(x)$, …
- This is an improvement over quadratic probing. But more expensive to implement.
- SEPARATE CHAINING: need linked list or dynamic arrays.

Deletions & Performance

- DELETES:
  - Need to be careful to leave a "marker".

- OPTIMAL VALUES OF LOAD FACTORS
- Doubling table size if load factors become high.
- REHASHING

- Hashing works very well in practice, and is widely used.
- Used to implement SYMBOL TABLES in compilers and various software systems.
- How does it compare to BST?
  - O(log N) versus O(1)

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.

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).

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```