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.

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

```java
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

```java
// 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

```java
// 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.
• Keys must be unique. Values 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();
}
```java
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;
                }
            }
        }
    }
}
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 );
}
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 );
}
## Mid Term Exam 1

<table>
<thead>
<tr>
<th>Range</th>
<th>Number</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>70:75</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>60:69</td>
<td>2</td>
<td>A- to A</td>
</tr>
<tr>
<td>50:59:00</td>
<td>4</td>
<td>B to A-</td>
</tr>
<tr>
<td>40:49:00</td>
<td>4</td>
<td>B- to B+</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>AVERAGE = 40</strong></td>
</tr>
<tr>
<td>30:39:00</td>
<td>6</td>
<td>C to B-</td>
</tr>
<tr>
<td>20:29</td>
<td>10</td>
<td>D to C</td>
</tr>
<tr>
<td>10:19</td>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>
Selection Sort

```java
public static void selectionSort( Comparable [] a )
{
    for( int p = 0; p < a.length-1; p++ )
    {
        int minIndex = p;
        for( j = p+1; j < a.length-1; j++ )
            if (a[minIndex].compareTo( a[j] ) > 0)
                minIndex = j;

        Comparable tmp = a[p];
        a[p] = a[minIndex];
        a[minIndex] = tmp;
    }
}
```
**Figure 8.3**  
Basic action of insertion sort (the shaded part is sorted)

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial State</strong></td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After a[0..1] is sorted</strong></td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After a[0..2] is sorted</strong></td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After a[0..3] is sorted</strong></td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After a[0..4] is sorted</strong></td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td><strong>After a[0..5] is sorted</strong></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Figure 8.4
A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..1] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..2] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..3] is sorted</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>After a[0..4] is sorted</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..5] is sorted</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
public static void insertionSort( Comparable[] a )
{
    for( int p = 1; p < a.length; p++ )
    {
        Comparable tmp = a[ p ];
        int j = p;

        for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
Figure 8.5
Shell sort after each pass if the increment sequence is \{1, 3, 5\}

<table>
<thead>
<tr>
<th>Original</th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>95</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 5-sort</td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>28</td>
<td>12</td>
<td>41</td>
<td>75</td>
<td>15</td>
<td>96</td>
<td>58</td>
<td>81</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>After 3-sort</td>
<td>28</td>
<td>12</td>
<td>11</td>
<td>35</td>
<td>15</td>
<td>41</td>
<td>58</td>
<td>17</td>
<td>94</td>
<td>75</td>
<td>81</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>35</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>95</td>
<td>96</td>
</tr>
</tbody>
</table>
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;
    }
}
public static void mergeSort( Comparable [] a ) {
    Comparable [] tmpArray = new Comparable[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}

private static void mergeSort( Comparable [] a, Comparable [] tmpArray,
    int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray, 
   int leftPos, int rightPos, int rightEnd ) 
{
   int leftEnd = rightPos - 1;
   int tmpPos = leftPos;
   int numElements = rightEnd - leftPos + 1;
   while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[leftPos].compareTo( a[rightPos] ) < 0 )
         tmpArray[tmpPos++] = a[leftPos++];
      else
         tmpArray[tmpPos++] = a[rightPos++];
   while( leftPos <= leftEnd ) // Copy rest of first half
      tmpArray[tmpPos++] = a[leftPos++];
   while( rightPos <= rightEnd ) // Copy rest of right half
      tmpArray[tmpPos++] = a[rightPos++];
   for( int i = 0; i < numElements; i++, rightEnd-- )
      a[rightEnd] = tmpArray[rightEnd];
}
Figure 8.10  Quicksort

1. Select pivot
2. Partition
3. Quick sort small items
4. Quick sort large items
Figure 8.11  Partitioning algorithm: Pivot element 6 is placed at the end.

```
| 8 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 6 |
```

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

```
| 8 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 6 |
```

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

```
| 2 | 1 | 4 | 9 | 0 | 3 | 5 | 8 | 7 | 6 |
```

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

```
| 2 | 1 | 4 | 9 | 0 | 3 | 5 | 8 | 7 | 6 |
```

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

```
| 2 | 1 | 4 | 5 | 0 | 3 | 9 | 8 | 7 | 6 |
```
Figure 8.16  Partitioning algorithm: i stops at large element 9; j stops at small element 3.

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6
\end{array}
\]

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

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 6 & 8 & 7 & 9
\end{array}
\]

Figure 8.18  Original array

\[
\begin{array}{cccccccc}
8 & 1 & 4 & 9 & 6 & 3 & 5 & 2 & 7 & 0
\end{array}
\]

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

\[
\begin{array}{cccccccccc}
0 & 1 & 4 & 9 & 6 & 3 & 5 & 2 & 7 & 8
\end{array}
\]

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

\[
\begin{array}{cccccccccc}
0 & 1 & 4 & 9 & 7 & 3 & 5 & 2 & 6 & 8
\end{array}
\]
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
}