Binary Node

```
class BinaryNode
{
    public BinaryNode( )  {  this( null, null, null ); }
    public BinaryNode( Object theElement, BinaryNode lt, BinaryNode rt );
    public static int size( BinaryNode t );   // size of subtree rooted at t
    public static int height( BinaryNode t );
    public void printPreOrder( );
    public void printPostOrder( );
    public void printInOrder( );
    public BinaryNode duplicate( ); // make a duplicate tree and return root
    public Object getElement( );
    public BinaryNode getLeft( );
    public BinaryNode getRight( );
    public void setElement( Object x );
    public void setLeft( BinaryNode t );
    public void setRight( BinaryNode t );

    private Object     element;
    private BinaryNode left;
    private BinaryNode right;
}
```

Binary Trees

```
public class BinaryTree
{
    public BinaryTree( );
    public BinaryTree( Object rootItem );
    public void printPreOrder( );
    public void printInOrder( );
    public void printPostOrder( );
    public void makeEmpty( );
    public boolean isEmpty( );
   /** Forms a new tree from rootItem, t1 and t2. t1 not equal to t2. */
    public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 );
    public int size( );
    public int height( );
    public BinaryNode getRoot( );

    private BinaryNode root;
}
```

Binary Trees Cont'd

```java
public class BinaryTree
{
   static public void main( String [ ] args )
   {
      BinaryTree t1 = new BinaryTree( "1" );   BinaryTree t3 = new BinaryTree( "3" );
      BinaryTree t5 = new BinaryTree( "5" );  BinaryTree t7 = new BinaryTree( "7" );
      BinaryTree t2 = new BinaryTree( );        BinaryTree t4 = new BinaryTree( );
      BinaryTree t6 = new BinaryTree( );
      t2.merge( "2", t1, t3 );   t6.merge( "6", t5, t7 );   t4.merge( "4", t2, t6 );

      System.out.println( "t4 should be perfect 1-7; t2 empty" );
      System.out.println( "---------------" );
      System.out.println( "t4" );
      t4.printInOrder( );
      System.out.println( "---------------" );
      System.out.println( "t2" );
      t2.printInOrder( );
      System.out.println( "---------------" );
      System.out.println( "t4 size: " + t4.size( ) );
      System.out.println( "t4 height: " + t4.height( ) );
   }
}
```
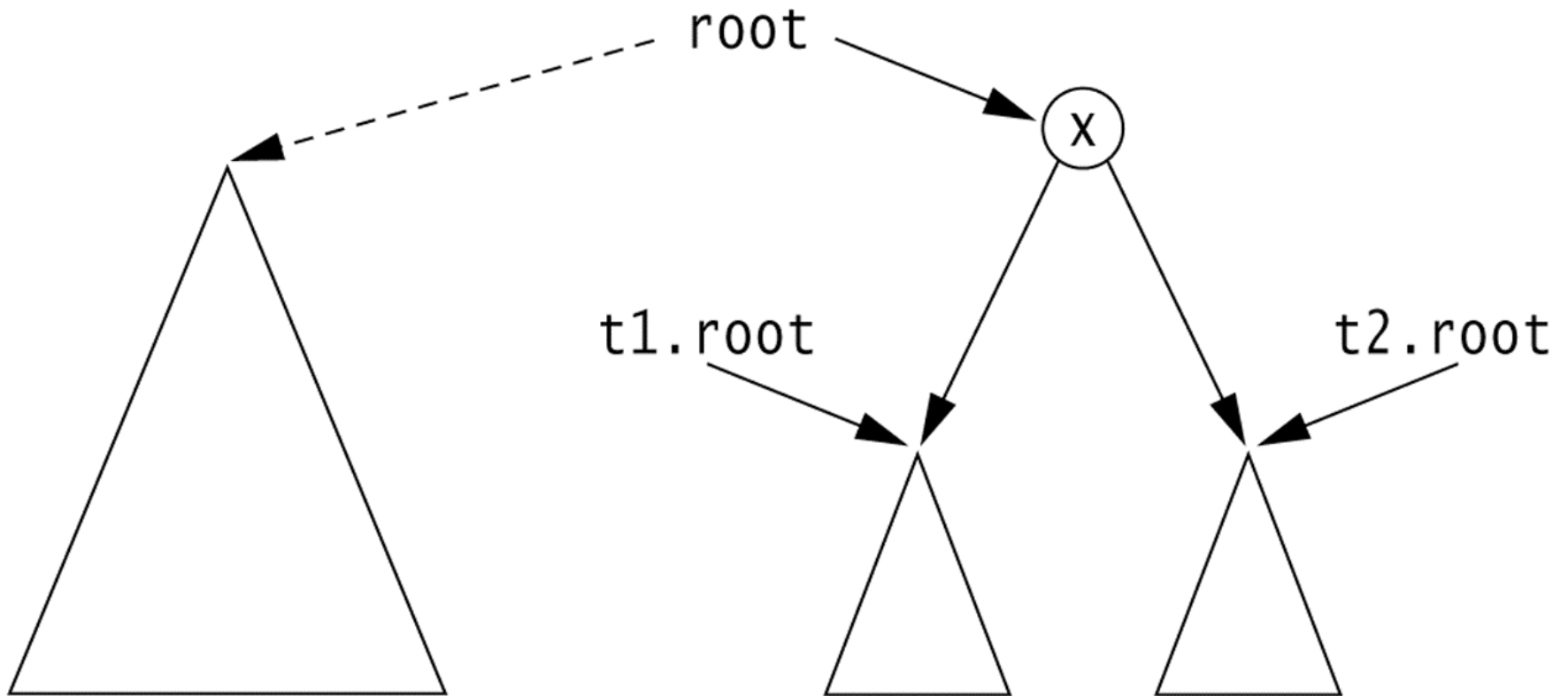
Binary Trees Cont'd

```java
public void printPreOrder( )
  {
    System.out.println( element );       // Node
    if( left != null )  left.printPreOrder( );        // Left
    if( right != null )  right.printPreOrder( );         // Right
  }
  public void printPostOrder( )
  {
    if( left != null )   left.printPostOrder( );        // Left
    if( right != null )   right.printPostOrder( );        // Right
    System.out.println( element );       // Node
  }
  public void printInOrder( )
  {
    if( left != null )  left.printInOrder( );        // Left
    System.out.println( element );      // Node
    if( right != null )  right.printInOrder( );        // Right
  }
```

Figure 18.14

Result of a naive merge operation: Subtrees are shared.

Binary Trees Cont'd

```
public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 )
 {
    if( t1.root == t2.root && t1.root != null )  {
       System.err.println( "leftTree==rightTree; merge aborted" );
       return;
    }
   root = new BinaryNode( rootItem, t1.root, t2.root );
    if( this != t1 )   t1.root = null;
    if( this != t2 )   t2.root = null;
 }
```

Figure 18.15

Aliasing problems in the merge operation; t1 is also the current object.
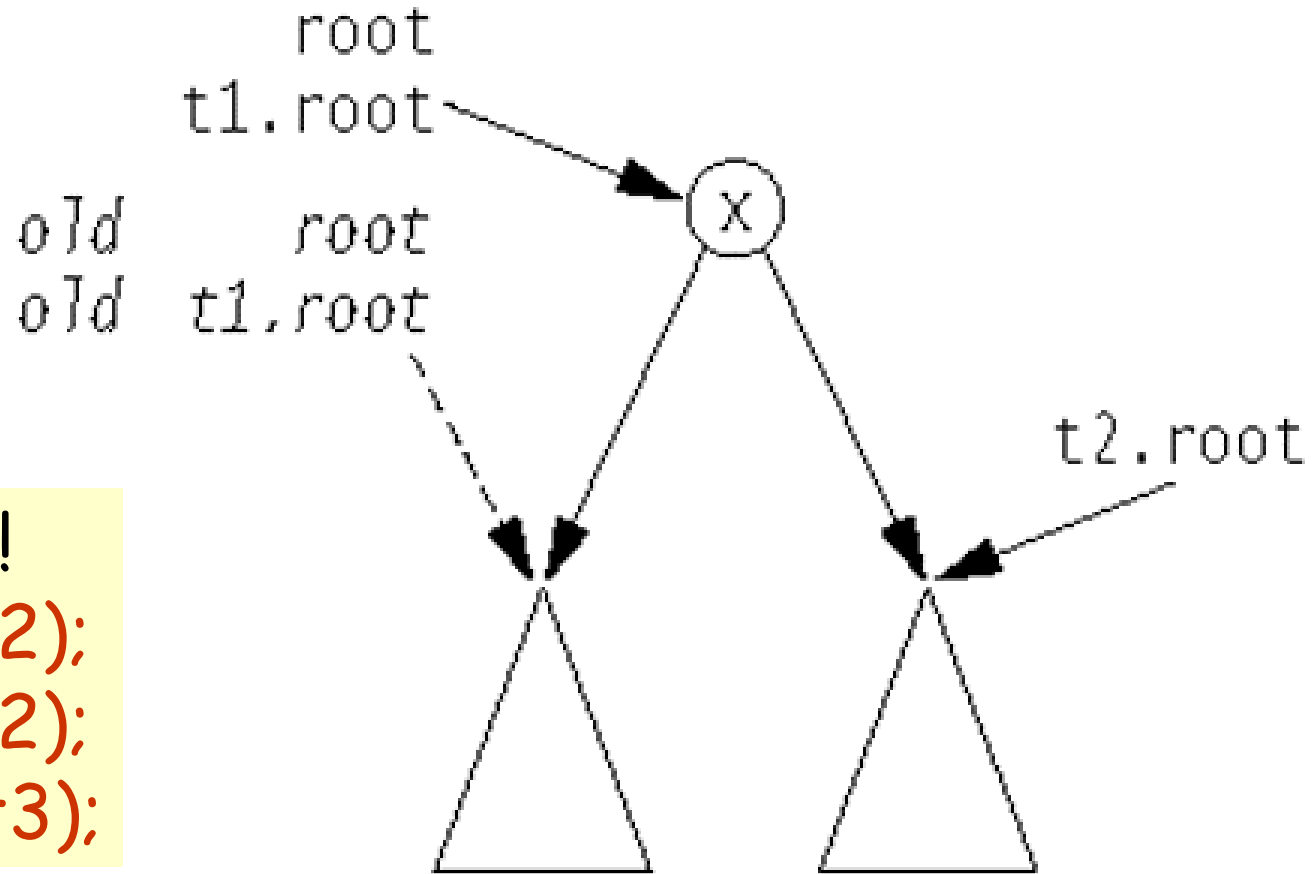
Consider these!
t1.merge(x,t1,t2);
t1.merge(x,t1,t2);
t1.merge(x,t3,t3);

# Binary Trees Cont'd

```
public BinaryNode duplicate( )
  {
      BinaryNode root = new BinaryNode( element, null, null );
      if( left != null )   root.left = left.duplicate( );
      if( right != null )  root.right = right.duplicate( );
      return root;                      // Return resulting tree
  }
```

Binary Search Trees

Values in the left subtree are smaller than the value stored at root.
Values in the right subtree are larger than the value stored at root.

Figure 19.1
Two binary trees: (a) a search tree;
(b) not a search tree
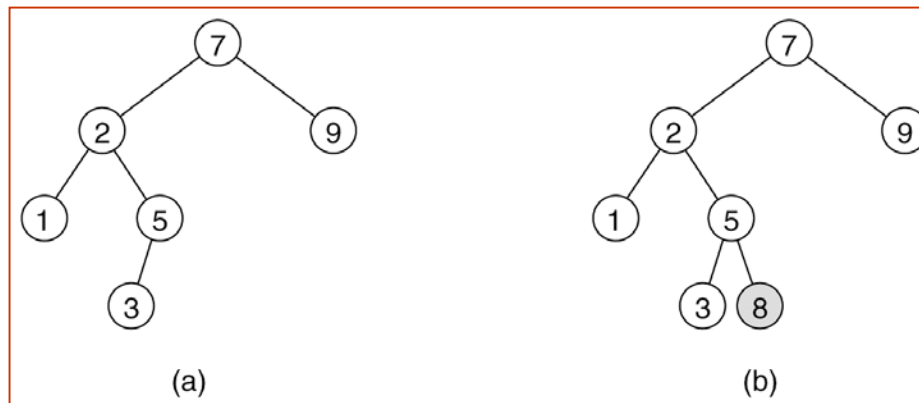
Binary Search Trees

```
// BinarySearchTree class
//
// void insert( x )      --> Insert x
// void remove( x )      --> Remove x
// void removeMin( )     --> Remove minimum item
// Comparable find( x )  --> Return item that matches x
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( ) --> Return largest item
// boolean isEmpty( )    --> Return true if empty; else false
// void makeEmpty( )     --> Remove all items
public class BinarySearchTree
{
  private Comparable elementAt( BinaryNode t )  { return t == null ? null : t.element; }
  protected BinaryNode insert( Comparable x, BinaryNode t )
  protected BinaryNode remove( Comparable x, BinaryNode t )
  protected BinaryNode removeMin( BinaryNode t )
  protected BinaryNode findMin( BinaryNode t )
  private BinaryNode findMax( BinaryNode t )
  private BinaryNode find( Comparable x, BinaryNode t )

  protected BinaryNode root;
}
```

Binary Search Trees

```java
public static void main( String [ ] args ) {
    BinarySearchTree t = new BinarySearchTree( );
    final int NUMS = 4000;
    final int GAP  =   37;
    System.out.println( "Checking... (no more output means success)" );
    for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
       t.insert( new Integer( i ) );
    for( int i = 1; i < NUMS; i+= 2 )
       t.remove( new Integer( i ) );
    if( ((Integer)(t.findMin( ))).intValue( ) != 2 ||
       ((Integer)(t.findMax( ))).intValue( ) != NUMS - 2 )
       System.out.println( "FindMin or FindMax error!" );
    for( int i = 2; i < NUMS; i+=2 )
        if( ((Integer)(t.find( new Integer( i ) ))).intValue( ) != i )
          System.out.println( "Find error1!" );
    for( int i = 1; i < NUMS; i+=2 )
    {
       if( t.find( new Integer( i ) ) != null )
          System.out.println( "Find error2!" );
    }
  }
}
```

Binary Search Trees

```
protected BinaryNode insert( Comparable x, BinaryNode t )   {
    if( t == null )
      t = new BinaryNode( x );
    else if( x.compareTo( t.element ) < 0 )
      t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
      t.right = insert( x, t.right );
    else   throw new DuplicateItemException( x.toString( ) );  // Duplicate
    return t;
  }
protected BinaryNode remove( Comparable x, BinaryNode t )  {
    if( t == null )   throw new ItemNotFoundException( x.toString( ) );
    if( x.compareTo( t.element ) < 0 )    t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )   t.right = remove( x, t.right );
    else if( t.left != null && t.right != null )   {
      t.element = findMin( t.right ).element;
      t.right = removeMin( t.right );
    }
    else
      t = ( t.left != null ) ? t.left : t.right;
    return t;
  }
```

Binary Search Trees

```
protected BinaryNode removeMin( BinaryNode t )
  {
    if( t == null )
      throw new ItemNotFoundException( );
    else if( t.left != null )
    {
      t.left = removeMin( t.left );
      return t;
    }
    else
      return t.right;
  }
```

```
protected BinaryNode findMin( BinaryNode t )
  {
    if( t != null )
      while( t.left != null )
        t = t.left;
    return t;
  }

private BinaryNode
        find( Comparable x, BinaryNode t )
  {
    while( t != null )
    {
      if( x.compareTo( t.element ) < 0 )
        t = t.left;
      else if( x.compareTo( t.element ) > 0 )
        t = t.right;
      else
        return t;    // Match
    }
    return null;        // Not found
  }
```

Figure 19.1
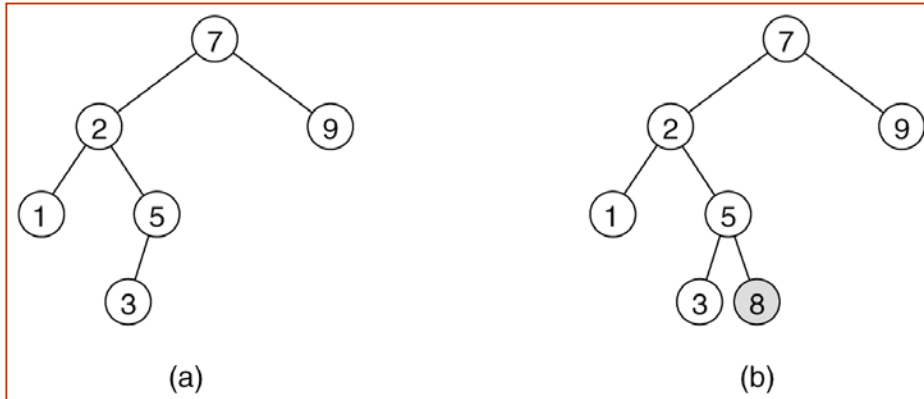
Two binary trees: (a) a search tree;
(b) not a search tree

(a)   (b)

Figure 19.2

Binary search trees
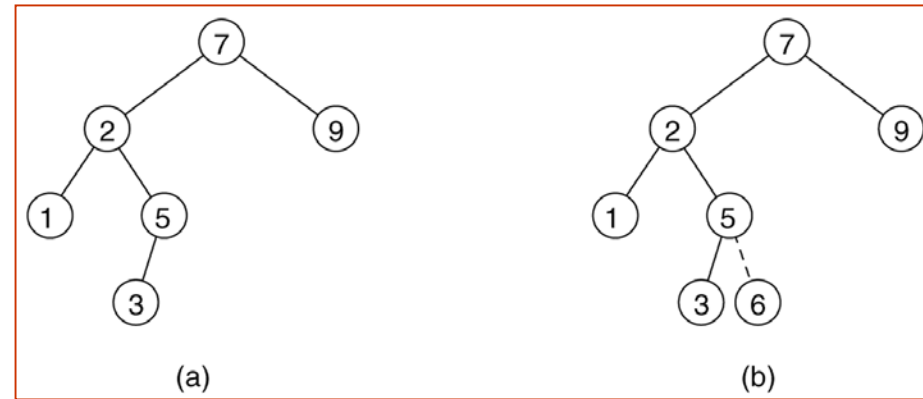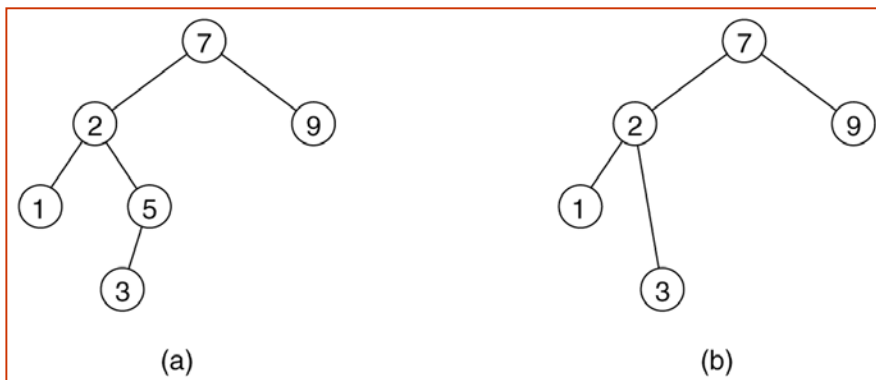(a) before and (b) after the insertion of 6

(a)   (b)

Figure 19.3
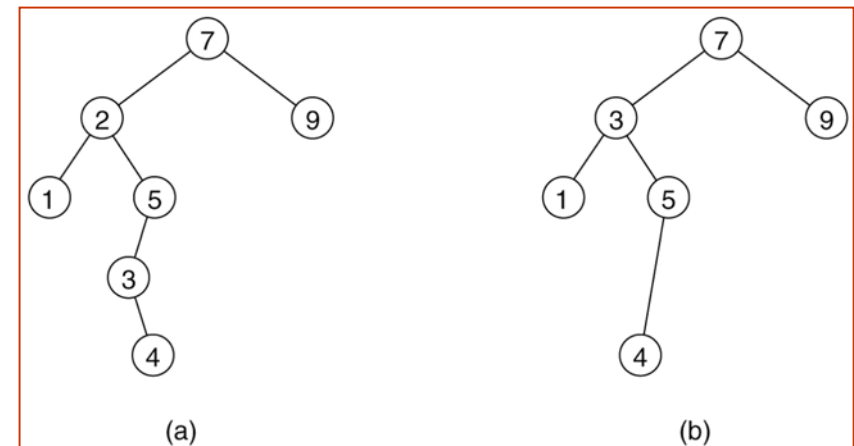
Deletion of node 5 with one child:
(a) before and (b) after

(a)   (b)

Figure 19.4

Deletion of node 2 with two children:
(a) before and (b) after

(a)   (b)

Figure 8.3
Basic action of insertion sort (the shaded part is sorted)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

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).

Array Position	0	1	2	3	4	5
Initial State	8	5				
After a[0..1] is sorted	5	8	9			
After a[0..2] is sorted	5	8	9	2		
After a[0..3] is sorted	2	5	8	9	6	
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

Insertion Sort

```
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

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;
        }
  }
```

Merge Sort

```java
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 );
    }
}
```

Merge in Merge Sort

```
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 ];
  }
```
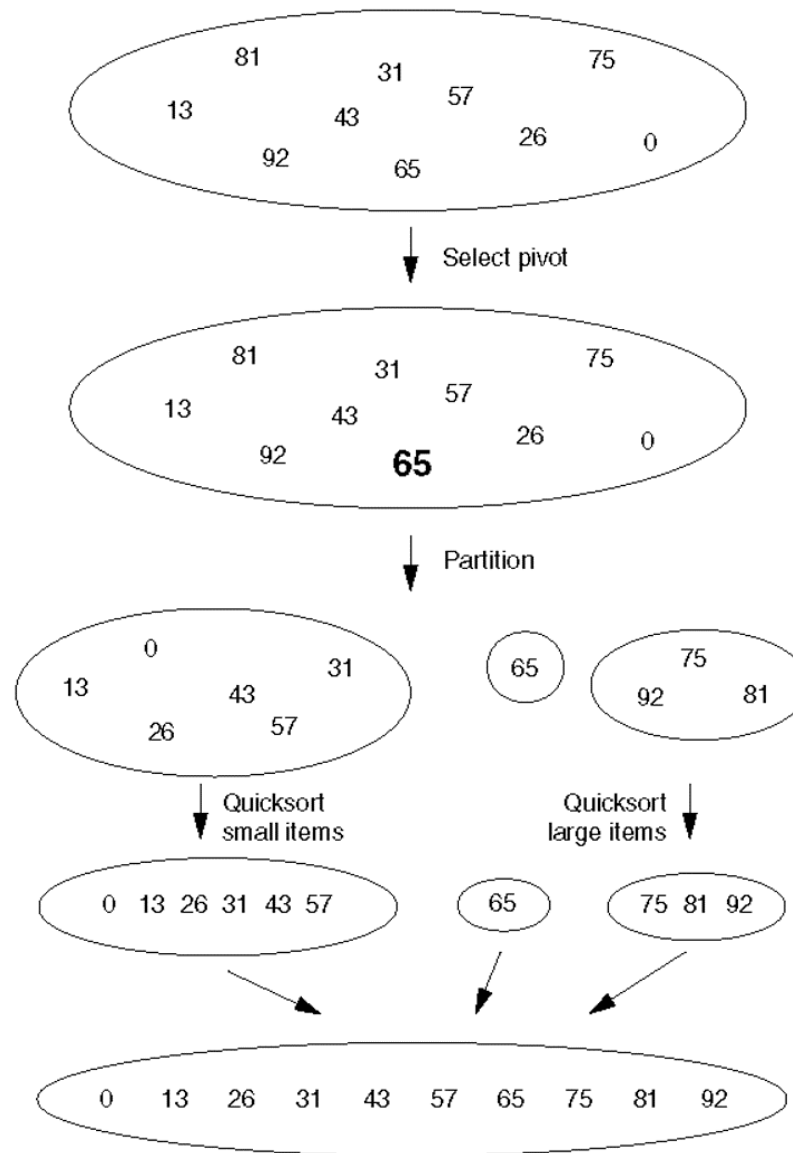
Figure 8.10 Quicksort

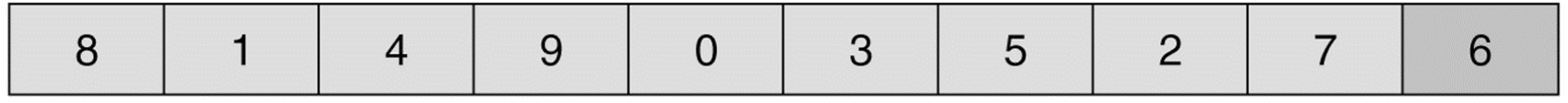**Figure 8.11** Partitioning algorithm: Pivot element 6 is placed at the end.

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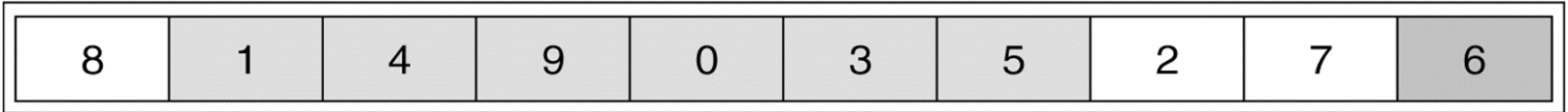
Figure 8.13 Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

2	1	4	9	0	3	5	8	7	6
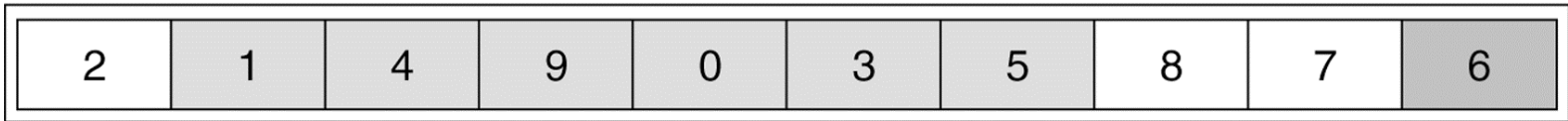
Figure 8.14 Partitioning algorithm: i stops at large element 9; j stops at small element 5.

2	1	4	9	0	3	5	8	7	6
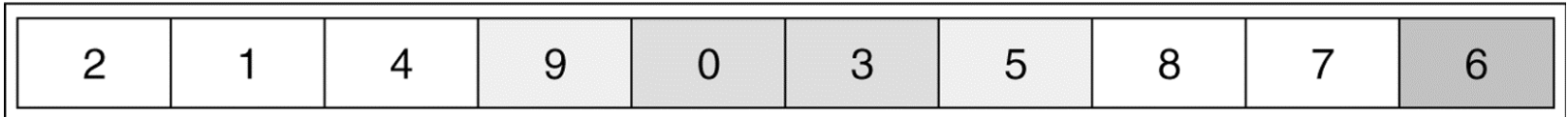
Figure 8.15 Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

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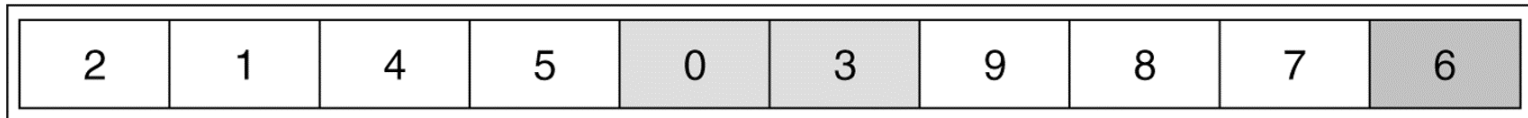
Figure 8.16 Partitioning algorithm: i stops at large element 9; j stops at small element 3.

2	1	4	5	0	3	9	8	7	6
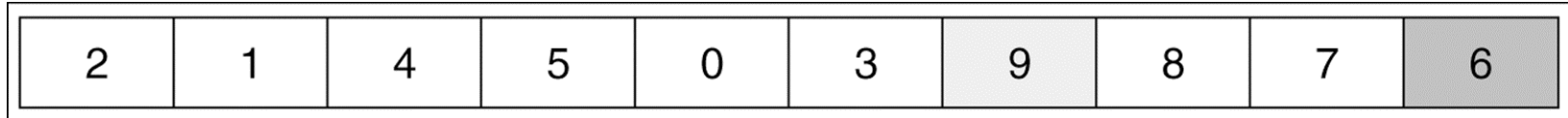
Figure 8.17 Partitioning algorithm: Swap pivot and element in position i.

2	1	4	5	0	3	6	8	7	9

Figure 8.18 Original array

8	1	4	9	6	3	5	2	7	0
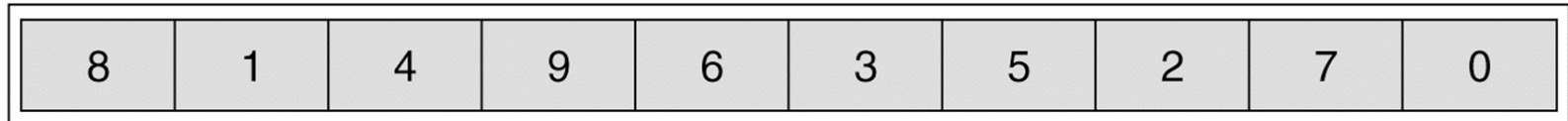
Figure 8.19 Result of sorting three elements (first, middle, and last)

0	1	4	9	6	3	5	2	7	8
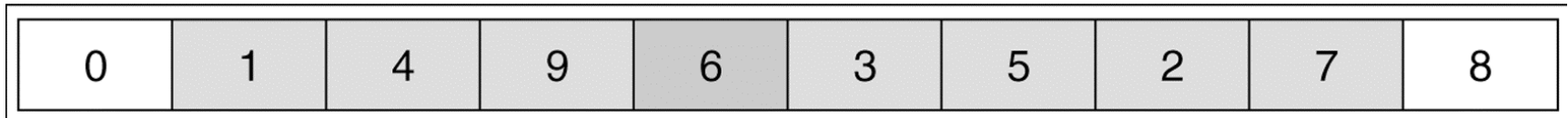
Figure 8.20 Result of swapping the pivot with the next-to-last element

0	1	4	9	7	3	5	2	6	8
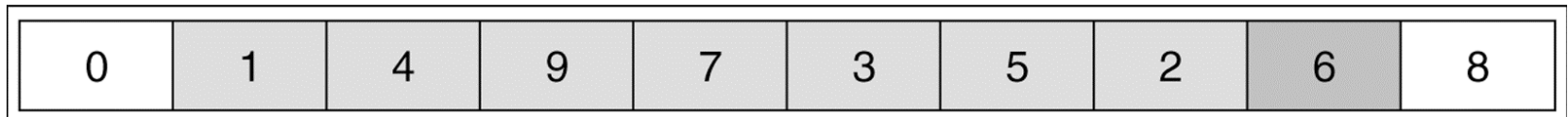
Quicksort

```
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
     }
   }
```