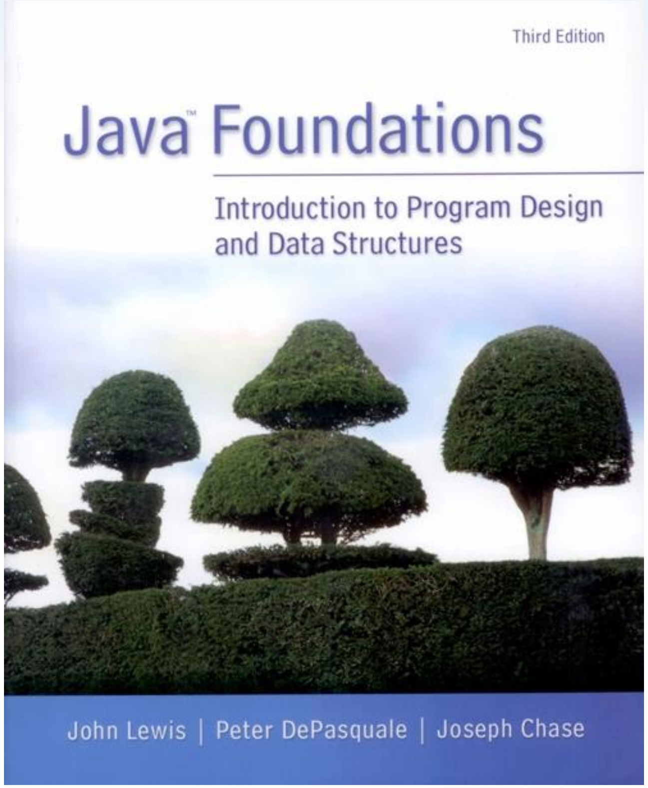


Chapter 18

Searching and Sorting

Chapter Scope

- Linear search and binary search algorithms
- Several sorting algorithms, including:
  - selection sort
  - insertion sort
  - bubble sort
  - quick sort
  - merge sort
- Complexity of the search and sort algorithms

Searching

- *Searching* is the process of finding a *target element* among a group of items (the *search pool*), or determining that it isn't there
- This requires repetitively comparing the target to candidates in the search pool
- An efficient search performs no more comparisons than it has to

Searching

- We'll define the algorithms such that they can search any set of objects, therefore we will search objects that implement the `Comparable` interface
- Recall that the `compareTo` method returns an integer that specifies the relationship between two objects:

```
obj1.compareTo(obj2)
```

- This call returns a number less than, equal to, or greater than 0 if `obj1` is less than, equal to, or greater than `obj2`, respectively

Generic Methods

- A class that works on a generic type must be instantiated
- Since our methods will be static, we'll define each method to be a *generic method*
- A generic method header contains the generic type before the return type of the method:

```
public static <T extends Comparable<T>> boolean
    linearSearch(T[] data, int min, int max, T target)
```

Generic Methods

- The generic type can be used in the return type, the parameter list, and the method body

Generic Method

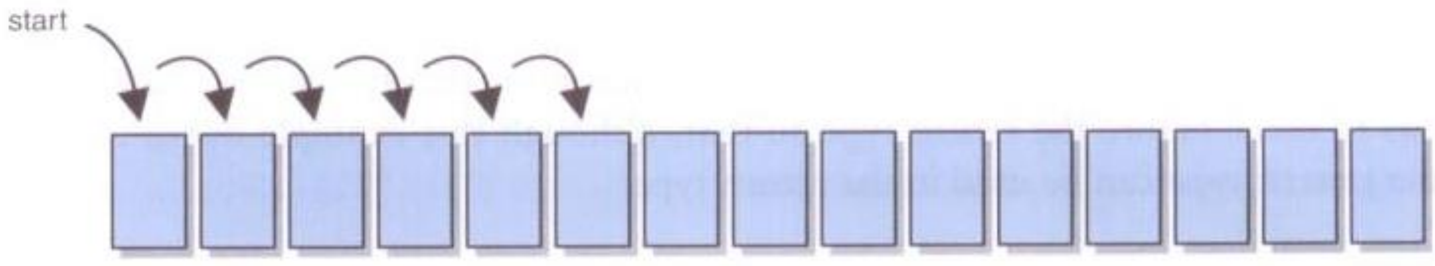
generic type parameter applies to this method

```
public static <T extends Comparable<T>> boolean
  linearSearch(T[] data, int min, int max, T target)
```

generic type can be used in
method parameters and return type

Linear Search

- A *linear search* simply examines each item in the search pool, one at a time, until either the target is found or until the pool is exhausted
- This approach does not assume the items in the search pool are in any particular order



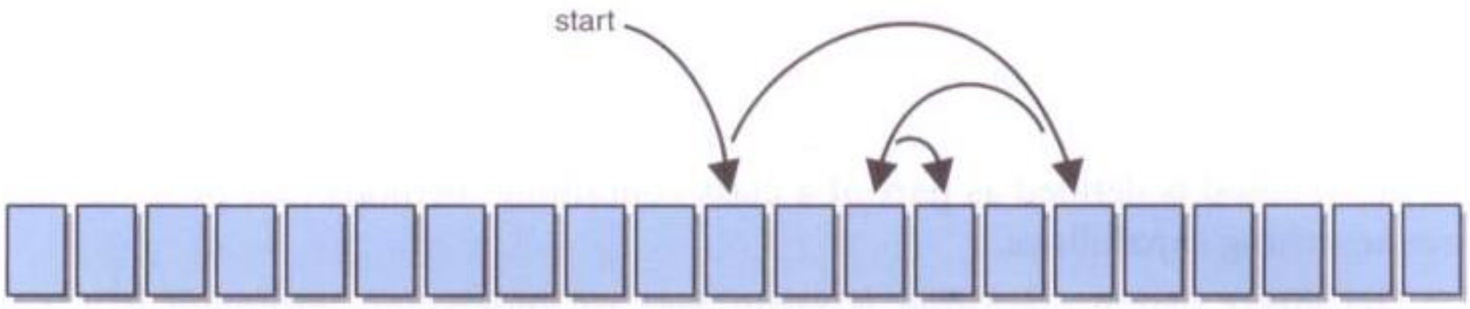
```java
/**
 * Searches the specified array of objects using a linear search
 * algorithm.
 *
 * @param data   the array to be searched
 * @param min    the integer representation of the minimum value
 * @param max    the integer representation of the maximum value
 * @param target the element being searched for
 * @return       true if the desired element is found
 */
public static <T>
    boolean linearSearch(T[] data, int min, int max, T target)
{
    int index = min;
    boolean found = false;

    while (!found && index <= max)
    {
        found = data[index].equals(target);
        index++;
    }

    return found;
}
```

Binary Search

- If the search pool is sorted, then we can be more efficient than a linear search
- A *binary search* eliminates large parts of the search pool with each comparison
- Instead of starting the search at one end, we begin in the middle
- If the target isn't found, we know that if it is in the pool at all, it is in one half or the other
- We can then jump to the middle of that half, and continue similarly

Binary Search

- Each comparison in a binary search eliminates half of the viable candidates that remain in the search pool:



Binary Search

- For example, find the number 29 in the following sorted list of numbers:

8 15 22 29 36 54 55 61 70 73 88

- First, compare the target to the middle value 54
- We now know that if 29 is in the list, it is in the front half of the list
- With one comparison, we've eliminated half of the data
- Then compare to 22, eliminating another quarter of the data, etc.

Binary Search

- A binary search algorithm is often implemented recursively
- Each recursive call searches a smaller portion of the search pool
- The base case is when there are no more viable candidates
- At any point there may be two “middle” values, in which case the first is used

```java
/**
 * Searches the specified array of objects using a binary search
 * algorithm.
 *
 * @param data   the array to be searched
 * @param min    the integer representation of the minimum value
 * @param max    the integer representation of the maximum value
 * @param target the element being searched for
 * @return       true if the desired element is found
 */
public static <T extends Comparable<T>>
    boolean binarySearch(T[] data, int min, int max, T target)
{
    boolean found = false;
    int midpoint = (min + max) / 2;  // determine the midpoint

    if (data[midpoint].compareTo(target) == 0)
        found = true;

    else if (data[midpoint].compareTo(target) > 0)
    {
        if (min <= midpoint - 1)
            found = binarySearch(data, min, midpoint - 1, target);
    }

    else if (midpoint + 1 <= max)
        found = binarySearch(data, midpoint + 1, max, target);

    return found;
}
```

Comparing Search Algorithms

- The expected case for finding an element with a linear search is n/2, which is O(n)
- Worst case is also O(n)
- The worst case for binary search is $(\log_2 n) / 2$ comparisons
- Which makes binary search O(log n)
- Keep in mind that for binary search to work, the elements must be already sorted

Sorting

- *Sorting* is the process of arranging a group of items into a defined order based on particular criteria
- Many sorting algorithms have been designed
- *Sequential sorts* require approximately n^2 comparisons to sort n elements
- *Logarithmic sorts* typically require $n\log_2 n$ comparisons to sort n elements
- Let's define a generic sorting problem that any of our sorting algorithms could help solve
- As with searching, we must be able to compare one element to another

```java
/**
 * SortPhoneList driver for testing an object selection sort.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class SortPhoneList
{
    /**
     * Creates an array of Contact objects, sorts them, then prints
     * them.
     */
    public static void main(String[] args)
    {
        Contact[] friends = new Contact[7];

        friends[0] = new Contact("John", "Smith", "610-555-7384");
        friends[1] = new Contact("Sarah", "Barnes", "215-555-3827");
        friends[2] = new Contact("Mark", "Riley", "733-555-2969");
        friends[3] = new Contact("Laura", "Getz", "663-555-3984");
        friends[4] = new Contact("Larry", "Smith", "464-555-3489");
        friends[5] = new Contact("Frank", "Phelps", "322-555-2284");
        friends[6] = new Contact("Marsha", "Grant", "243-555-2837");

        Sorting.insertionSort(friends);

        for (Contact friend : friends)
            System.out.println(friend);
    }
}
```

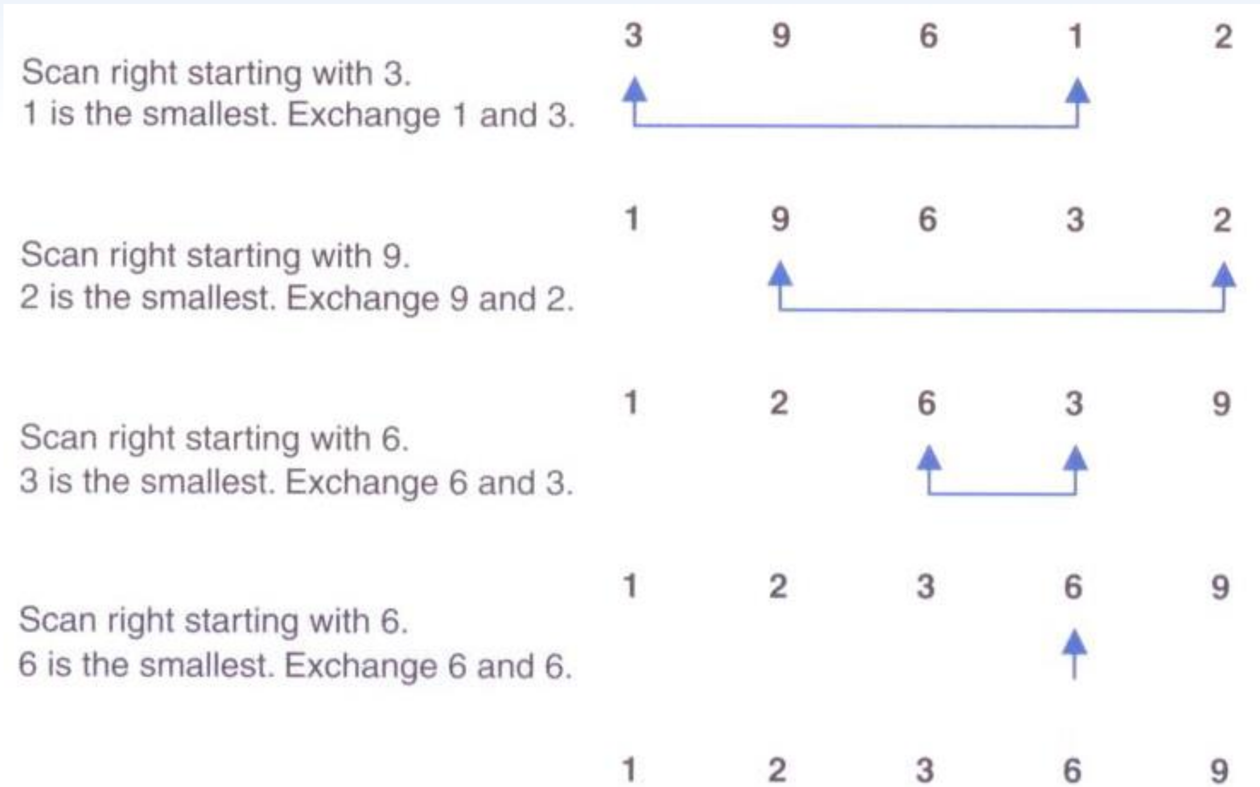
```java
/**
 * Contact represents a phone contact.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Contact implements Comparable<Contact>
{
    private String firstName, lastName, phone;

    /**
     * Sets up this contact with the specified information.
     *
     * @param first     a string representation of a first name
     * @param last      a string representation of a last name
     * @param telephone a string representation of a phone number
     */
    public Contact(String first, String last, String telephone)
    {
        firstName = first;
        lastName = last;
        phone = telephone;
    }
```

```java
    /**
     * Returns a description of this contact as a string.
     *
     * @return a string representation of this contact
     */
    public String toString()
    {
        return lastName + ", " + firstName + "\t" + phone;
    }

    /**
     * Uses both last and first names to determine lexical ordering.
     *
     * @param other the contact to be compared to this contact
     * @return      the integer result of the comparison
     */
    public int compareTo(Contact other)
    {
        int result;

        if (lastName.equals(other.lastName))
            result = firstName.compareTo(other.firstName);
        else
            result = lastName.compareTo(other.lastName);

        return result;
    }
}
```

Selection Sort

- *Selection sort* orders a list of values by repetitively putting a particular value into its final position
- More specifically:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection Sort



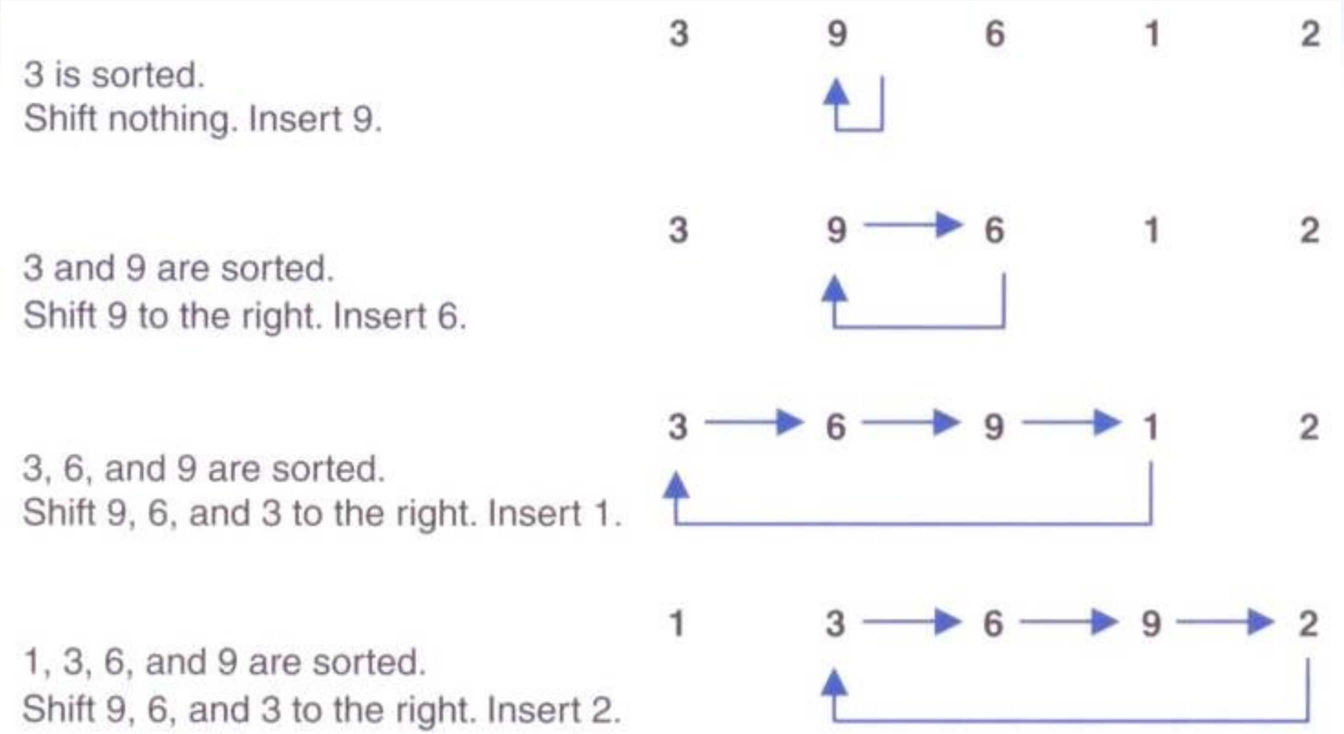
```java
/**
 * Sorts the specified array of integers using the selection
 * sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void selectionSort(T[] data)
{
    int min;
    T temp;

    for (int index = 0; index < data.length-1; index++)
    {
        min = index;
        for (int scan = index+1; scan < data.length; scan++)
            if (data[scan].compareTo(data[min])<0)
                min = scan;

        swap(data, min, index);
    }
}
```

```java
/**
 * Swaps to elements in an array. Used by various sorting algorithms.
 *
 * @param data   the array in which the elements are swapped
 * @param index1 the index of the first element to be swapped
 * @param index2 the index of the second element to be swapped
 */
private static <T extends Comparable<T>>
    void swap(T[] data, int index1, int index2)
{
    T temp = data[index1];
    data[index1] = data[index2];
    data[index2] = temp;
}
```

Insertion Sort

- *Insertion sort* orders a values by repetitively inserting a particular value into a sorted subset of the list
- More specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion Sort



```java
/**
 * Sorts the specified array of objects using an insertion
 * sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void insertionSort(T[] data)
{
    for (int index = 1; index < data.length; index++)
    {
        T key = data[index];
        int position = index;

        // shift larger values to the right
        while (position > 0 && data[position-1].compareTo(key) > 0)
        {
            data[position] = data[position-1];
            position--;
        }

        data[position] = key;
    }
}
```

Bubble Sort

- *Bubble sort* orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- More specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

```java
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void bubbleSort(T[] data)
{
    int position, scan;
    T temp;

    for (position =  data.length - 1; position >= 0; position--)
    {
        for (scan = 0; scan <= position - 1; scan++)
        {
            if (data[scan].compareTo(data[scan+1]) > 0)
                swap(data, scan, scan + 1);
        }
    }
}
```

Quick Sort

- *Quick sort* orders values by partitioning the list around one element, then sorting each partition
- More specifically:
  - choose one element in the list to be the partition element
  - organize the elements so that all elements less than the partition element are to the left and all greater are to the right
  - apply the quick sort algorithm (recursively) to both partitions

```java
/**
 * Sorts the specified array of objects using the quick sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void quickSort(T[] data)
{
    quickSort(data, 0, data.length - 1);
}
```

```java
/**
 * Recursively sorts a range of objects in the specified array using the
 * quick sort algorithm.
 *
 * @param data the array to be sorted
 * @param min  the minimum index in the range to be sorted
 * @param max  the maximum index in the range to be sorted
 */
private static <T extends Comparable<T>>
    void quickSort(T[] data, int min, int max)
{
    if (min < max)
    {
        // create partitions
        int indexofpartition = partition(data, min, max);

        // sort the left partition (lower values)
        quickSort(data, min, indexofpartition - 1);

        // sort the right partition (higher values)
        quickSort(data, indexofpartition + 1, max);
    }
}
```

```java
/**
 * Used by the quick sort algorithm to find the partition.
 *
 * @param data the array to be sorted
 * @param min  the minimum index in the range to be sorted
 * @param max  the maximum index in the range to be sorted
 */
private static <T extends Comparable<T>>
    int partition(T[] data, int min, int max)
{
    T partitionelement;
    int left, right;
    int middle = (min + max) / 2;

    // use the middle data value as the partition element
    partitionelement = data[middle];
    // move it out of the way for now
    swap(data, middle, min);

    left = min;
    right = max;
```

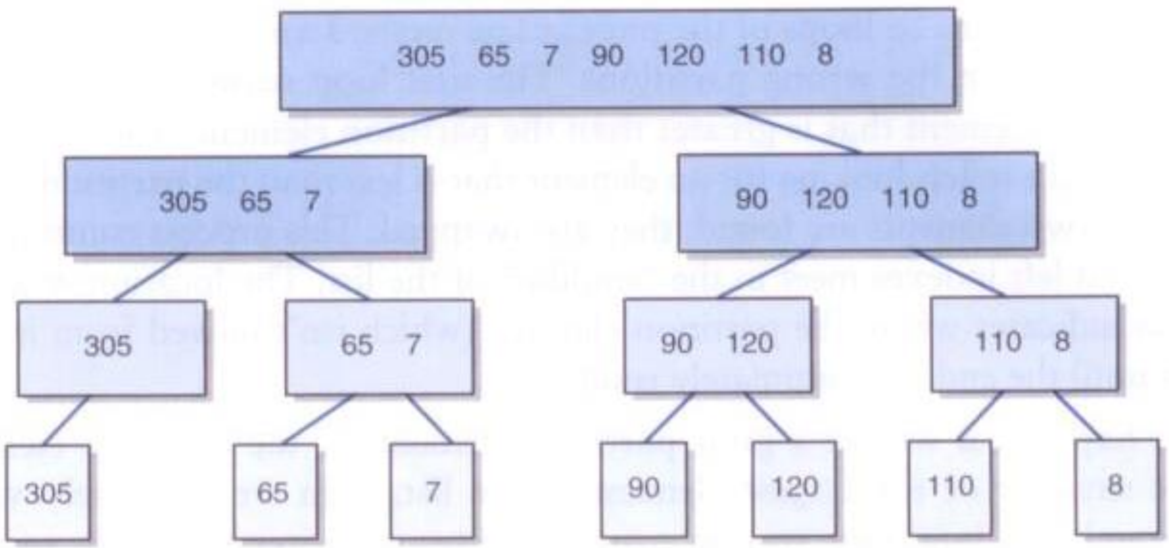
```java
      while (left < right)
      {
          // search for an element that is > the partition element
          while (left < right && data[left].compareTo(partitionelement) <= 0)
              left++;

          // search for an element that is < the partition element
          while (data[right].compareTo(partitionelement) > 0)
              right--;

          // swap the elements
          if (left < right)
              swap(data, left, right);
      }

      // move the partition element into place
      swap(data, min, right);

      return right;
  }
```

Merge Sort

- *Merge sort* orders values by recursively dividing the list in half until each sub-list has one element, then recombining
- More specifically:
  - divide the list into two roughly equal parts
  - recursively divide each part in half, continuing until a part contains only one element
  - merge the two parts into one sorted list
  - continue to merge parts as the recursion unfolds

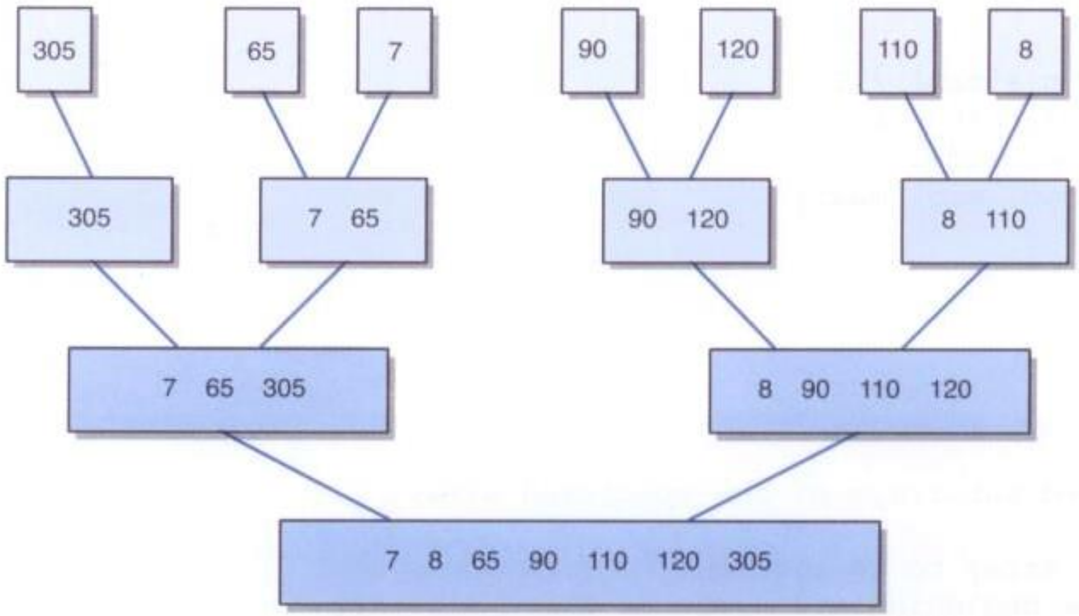
Merge Sort

- Dividing lists in half repeatedly:



Merge Sort

- Merging sorted elements



```java
/**
 * Sorts the specified array of objects using the merge sort
 * algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void mergeSort(T[] data)
{
    mergeSort(data, 0, data.length - 1);
}

/**
 * Recursively sorts a range of objects in the specified array using the
 * merge sort algorithm.
 *
 * @param data the array to be sorted
 * @param min  the index of the first element
 * @param max  the index of the last element
 */
private static <T extends Comparable<T>>
    void mergeSort(T[] data, int min, int max)
{
    if (min < max)
    {
        int mid = (min + max) / 2;
        mergeSort(data, min, mid);
        mergeSort(data, mid+1, max);
        merge(data, min, mid, max);
    }
}
```

```java
/**
 * Merges two sorted subarrays of the specified array.
 *
 * @param data the array to be sorted
 * @param first the beginning index of the first subarray
 * @param mid the ending index fo the first subarray
 * @param last the ending index of the second subarray
 */
@SuppressWarnings("unchecked")
private static <T extends Comparable<T>>
    void merge(T[] data, int first, int mid, int last)
{
    T[] temp = (T[])(new Comparable[data.length]);

    int first1 = first, last1 = mid;  // endpoints of first subarray
    int first2 = mid+1, last2 = last;  // endpoints of second subarray
    int index = first1;  // next index open in temp array

    //  Copy smaller item from each subarray into temp until one
    //  of the subarrays is exhausted
    while (first1 <= last1 && first2 <= last2)
    {
        if (data[first1].compareTo(data[first2]) < 0)
        {
            temp[index] = data[first1];
            first1++;
        }
```

```
            else
            {
                temp[index] = data[first2];
                first2++;
            }
            index++;
        }

        //  Copy remaining elements from first subarray, if any
        while (first1 <= last1)
        {
            temp[index] = data[first1];
            first1++;
            index++;
        }

        //  Copy remaining elements from second subarray, if any
        while (first2 <= last2)
        {
            temp[index] = data[first2];
            first2++;
            index++;
        }

        //  Copy merged data into original array
        for (index = first; index <= last; index++)
            data[index] = temp[index];
    }
```

Comparing Sorts

- Selection sort, insertion sort, and bubble sort use different techniques, but are all $O(n^2)$
- They are all based in a nested loop approach
- In quick sort, if the partition element divides the elements in half, each recursive call operates on about half the data
- The act of partitioning the elements at each level is $O(n)$
- The effort to sort the entire list is $O(n \log n)$
- It could deteriorate to $O(n^2)$ if the partition element is poorly chosen

Comparing Sorts

- Merge sort divides the list repeatedly in half, which results in the O(log n) portion
- The act of merging is O(n)
- So the efficiency of merge sort is O(n log n)

- Selection, insertion, and bubble sorts are called *quadratic sorts*
- Quick sort and merge sort are called *logarithmic sorts*

Radix Sort

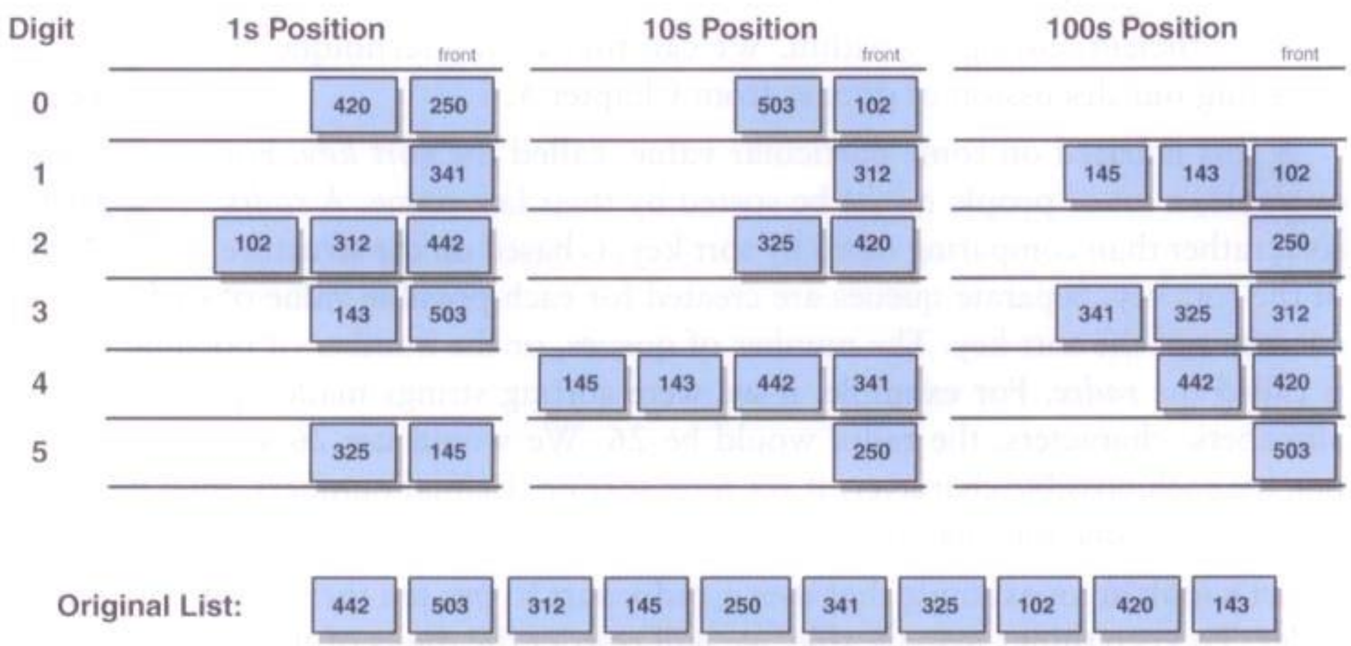
- Let's look at one other sorting algorithm, which only works when a *sort key* can be defined
- Separate queues are used to store elements based on the structure of the sort key
- For example, to sort decimal numbers, we'd use ten queues, one for each possible digit (0 – 9)
- To keep our example simpler, we'll restrict our values to the digits 0 - 5

Radix Sort

- The radix sort makes three passes through the data, for each position of our 3-digit numbers
- A value is put on the queue corresponding to that position's digit
- Once all three passes are finished, the data is sorted in each queue

Radix Sort

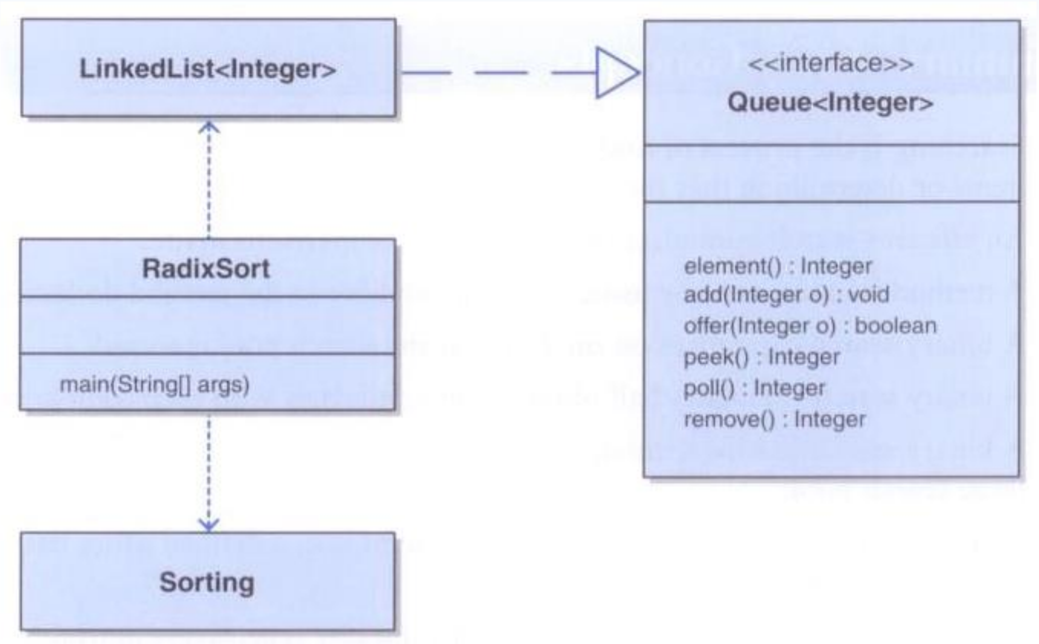
- An example using six queues to sort 10 three-digit numbers:



```java
import java.util.*;

/**
 * RadixSort driver demonstrates the use of queues in the execution of a radix sort.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class RadixSort
{
    /**
     * Performs a radix sort on a set of numeric values.
     */
    public static void main(String[] args)
    {
        int[] list = {7843, 4568, 8765, 6543, 7865, 4532, 9987, 3241,
                      6589, 6622, 1211};

        String temp;
        Integer numObj;
        int digit, num;

        Queue<Integer>[] digitQueues = (LinkedList<Integer>[])(new LinkedList[10]);
        for (int digitVal = 0; digitVal <= 9; digitVal++)
            digitQueues[digitVal] = (Queue<Integer>)(new LinkedList<Integer>());
```

```
        // sort the list
        for (int position=0; position <= 3; position++)
        {
            for (int scan=0; scan < list.length; scan++)
            {
                temp = String.valueOf(list[scan]);
                digit = Character.digit(temp.charAt(3-position), 10);
                digitQueues[digit].add(new Integer(list[scan]));
            }

            // gather numbers back into list
            num = 0;
            for (int digitVal = 0; digitVal <= 9; digitVal++)
            {
                while (!(digitQueues[digitVal].isEmpty()))
                {
                    numObj = digitQueues[digitVal].remove();
                    list[num] = numObj.intValue();
                    num++;
                }
            }
        }

        // output the sorted list
        for (int scan=0; scan < list.length; scan++)
            System.out.println(list[scan]);
    }
}
```

Radix Sort


LinkedList<Integer>
<<interface>>
Queue<Integer>
RadixSort
main(String[] args)
element() : Integer
add(Integer o) : void
offer(Integer o) : boolean
peek() : Integer
poll() : Integer
remove() : Integer
Sorting