Answer Book for Classic Data Structures in Java published by Addison-Wesley

Timothy A. Budd
Oregon State University
Corvallis, Oregon, USA

October 4, 2000

1 Chapter 1 – Complexity

1. Inheritance Hierarchy: Biological hierarchies are a good example. Non inheritance hierarchy: Organizational charts.

2. Many possible answers.

3. Many possible answers

4. This proof given in Chapter 5.

2 Chapter 2 - ADT

1. Many different answers are possible, the instructor needs to look at the form of the answer. Here is one possible answer:

   insert: Post: Places a value into the container. This overwrites any previously inserted value.

   extract: Post: Removes the value most recently inserted into the container.

   Under this interpretation, the FortyTwoBox would not be said to properly implement the interface, since it does not remove the most recent value. To allow FortyTwoBox you need to disassociate the value returned from any of the values inserted – saying something like it returns an integer value.

2. For Box the conditions are as described in the previous answer. For OneTimeBox the conditions are as follows:

   insert: Post: The first time the method is called it places a value into the container. This method has no effect on subsequent calls.

   extract: Pre: A call in insert must have already occurred. Post: Removes the value first inserted into the box.

   For SuccessorBox the condition for extract is changed to the following:

   extract: Pre: increments the value held in the box by one, and returns the updated value.

3. The interface is something like this:

```
interface DataBoxInterface {
    Enumeration elements ();
    int count ();
    void add (Object v);
    Object find (Object v);
    void delete (Object v);
}
```

3 Chapter 3 – Algorithms

1. Many possible answers
2. Many possible answers
3. The three properties are integer, decreasing, and positve. Infinite sequences that satisfy two of the three include:
   (a) integer, decreasing, but not positive: negative integers
   (b) decreasing, positive, but not integers: 1/n as n increases
   (c) integer, positive, but not decreasing: integer numbers
4.

```
public int gcd (int n, int m) {
    if (m != n)
        if (m > m) return gcd(n-m, m);
        else return gcd(n, m-n);
    return n;
}
```

5. ```
   move disk from tower a to b
   move disk from tower a to c
   move disk from tower b to c
   move disk from tower a to b
   move disk from tower c to a
   move disk from tower c to b
   move disk from tower a to b
   move disk from tower a to c
   move disk from tower b to c
   move disk from tower b to a
   move disk from tower c to a
   move disk from tower b to c
   move disk from tower a to b
   move disk from tower a to c
   move disk from tower b to c
   move disk from tower a to b
   move disk from tower c to a
   move disk from tower c to b
   move disk from tower a to b
   move disk from tower c to a
   move disk from tower b to c
   move disk from tower b to a
   move disk from tower c to a
   move disk from tower c to b
   move disk from tower a to b
   move disk from tower a to c
   move disk from tower b to c
   move disk from tower a to b
   move disk from tower c to a
   move disk from tower c to b
   move disk from tower a to b
   ```

6. First, you must prove by induction that the towers of hanoi requires 2^n steps to move a tower of size n. (In hindsight, this should have been a separate exercise). The question then amounts to asking for value value of n is 2^n larger than 60×60. The answer is 12, so that at most 11 disks can be moved in an hour.
   ```

4 Chapter 4 – Time Management

1. (a) If $f(n)$ is $O(g(n))$ it means that there exists a constant, c_1, such that $f(n) < c_1 \times g(n)$ for all sufficiently large n. But then $f(n) + c < c_1 \times g(n) + c$. Since $g(n)$ is larger than 1, this is smaller than $c_1 \times g(n) + c \times g(n)$. But the latter is $(c_1 + c) \times g(n)$. Since $(c_1 + c)$ is constant (call it c_2), we have that $f(n) + c < c_2 \times g(n)$, and hence $f(n) + c$ is $O(g(n))$.

   (b) If $f(n)$ is $O(g(n))$ it means that there exists a constant, c_1, such that $f(n) < c_1 \times g(n)$ for all sufficiently large n. But then $c \times f(n) < c \times c_1 \times g(n)$. Let c_2 be $c \times c_1$ and we have the result.

   (c) Since $f_1(n) < f_2(n)$, $f_1(n) + f_2(n) < 2 \times f_2(n)$. Since $f_2(n)$ is $O(g(n))$, there exists a constant, c_1, such that $f_2(n) < c_1 \times g(n)$ for all sufficiently large n. Let c_2 be $2 \times c_1$. Then $f_1(n) + f_2(n) < c_2 \times g(n)$, and therefore is $O(g(n))$.

2. Prove by induction. Base case for j = 0, let a be the constant, and therefore $a \times x^i$ is $O(x^i)$. Base case for j = 1. For any value x larger than 1 $a \times x^i < a \times x^i \times x$, and hence $a \times x^i$ is $O(x^{i+1})$. Induction case. Assume it is true for $j = n$, prove for $n + 1$. But if we assume $a \times x^i$ is $O(x^{i+n})$, it means that there exists a constant c such that $a \times x^i < c \times x^{i+n}$. But for $x > 0$ the right hand side is less than $c \times x^{i+n+1}$, and hence the function is $O(x^{i+n+1})$.

3. Prove by induction on the degree of the polynomial.

4. By definition $\log_a(n)$ is $\log_2(n)/log_2(a)$. Since $1/log_2(a)$ is a constant, we have the result.

5. Base case of $n = 1$ is trivial, since they are the same. Assume it is true for n, prove for $n + 1$. But we have that

$$\sum_{i=1}^{n+1} c \times f(i) = \sum_{i=1}^{n} c \times f(i) + c \times f(n+1) = c \times (\sum_{i=1}^{n} f(i) + f(n+1) = c \times (\sum_{i=1}^{n+1} f(i)$$

6. This proof is given in Chapter 5.

7. (a) $O(n^2)$
   (b) $O(n \log n)$
   (c) $O(n)$
   (d) $O(n^2)$

(e) $O(n)$

(f) $O(n \log n)$

8. (a) $O(n)$

   (b) $O(n^2)$

   (c) $O(n^2)$

   (d) $O(\log n)$

   (e) $O(n \times \sqrt{n})$

   (f) $O(n \log n)$

9. Misprint, the 80 should be a 60. Easiest to work in seconds. This means the constant of proportinality is about 1. For n = 60 then $n \log n$ is about 60×8, or about 8 minuits.

10. `2 3 7 4 9`
    `2 3 4 7 9`
    `2 3 4 7 9`
    `2 3 4 7 9`

11. `2 7 3 9 4`
    `2 3 7 9 4`
    `2 3 7 9 4`
    `2 3 4 7 9`

5 Chapter 5 – Correceness

1. From the beginning of program to invariant 1. i is 2, hence $i - 1$ is 1, hence n has no factors between 2 and $i - 1$, since the latter set is empty.

   From invariant 1 to invariant 2. Since the condition evaluated to true, the invariant must be true (actually, independently of 1) and therefore n is not prime.

   From invariant 1 to invariant 3. We only reach invariant 3 if the condition was false. Since invariant 1 asserted there was no factor between 2 and $i - 1$, and since the condition is false we know that i is not a factor of n, then invariant 3 must be true.

   From invariant 3 to invariant 1. The only thing that changes is that i is incremented by 1, and so the two invariants assert the same thing.

   From invariant 3 to invariant 4. Since i is strictly larger than the ceiling of the square root, and we know that n has no factors smaller than $i - 1$ (note that i has been increated between the two invariants) then n can not have any factors.

   From the beginning of the program to invariant 4. This only happens if 2 is larger than n, so n must be 2 or 3, which are both prime.

2. From the beginning of the program to the loop invariant. Both sets are empty in the loop invariant, and so the conditions must be true.

   From the loop invariant to itself on the next iteration. We divide this argument into two cases. If data[mid] is smaller than the test value, then we know that all values with index positions smaller than or equal to mid are smaller than the test value. If we set low to mid+1, this tells us that all positions with index values smaller than low-1 are smaller than the test value, which is the condition we seek. (high hasn't changed, so the 2nd part of the condition is still true). On the other hand, if data[mid] is larger than or equal to the test value, we know that all elements with index values larger than or equal to mid are equal to the test value. If we set high equal to mid, then the 2nd part of the invariant is true. (Since low hasn't changed, the first part remains true).

   From the loop invaraint to the end of the program. The first clause is the same as the loop invariant. The second can be proved by contradiction. Assume it is not true, that is, that testValue is greater than data[low+1]. But we know that all elements with index values larger than high are greater than or equal to testValue. And we know that high is less than or equal to low (by the while loop). But these two can't both be true.

   Finally, from the precondition to the end assuming the loop is not executed. This than can only happen if the array has length zero, and thus the only legal index to return is zero.

3. Invariant 1. Result is base raised to zero.

   Invariant 2. Result is base raised to (i-1).

   Invariant 3. Result is base raised to i.

   Invariant 4. Result is base raised to n.

4. From invariant 1 to invariant 2. i is 1, $i - 1$ is zero, and thus invariant 2 asserts the same thing as invariant 1.

   From invariant 2 to invariant 3. Result has been multipled by base, and thus if invariant 2 was true invariant 3 must now be true.

   From invariant 3 back to invariant 2. i has been incremented by 1, and thus the two invariants are asserting the same thing.

   From invariant 3 to invariant 4. i must be equal to n on the last iteration, and thus the two invariants are asserting the same thing,

   From invariant 1 to invariant 4. n must be zero, and hence the value held by result is correct.

5. Invariant 1. sum is sum of value with index positions smaller than 0 (an empty set)

Invariant 2. sum is sum of values with index positions smaller than i-1.

Invariant 3. sum is sum of values with index positions smaller than i.

Invariant 4. sum is sum of values with index positions smaller than n.

6. From beginning of program to invariant 1, sum of an empty set is zero. From invariant 1 to invariant 2, i must be 0, set is empty and still has sum zero. From invariant 2 to invariant 3, all that has changed is that the value data[i] is added to sum, and so invariant 3 must be true. From invariant 3 back to invariant 2, all that changes is that i is incremented by 1, and so the two invariants are asserting the same thing. From invariant 3 to invariant 4, i must be equal to the size of the array, and thus invariant 3 asserts that all elements are in sum. Finally, from invariant 1 to invariant 4, this only happens if size is zero, and thus sum is also zero.

7. Invariant 1. result is 0 fac (zero factorial)

   Invariant 2. result is (i-1) fac

   Invaraint 3. result is i fac

   Invariant 4. result is val fac

8. From beginning of program to invariant 1, result is set to 1, which is zero factorial, so invariant must hold.

   From invariant 1 to invariant 2, i is equal to 1, so i-1 is zero, and so invariant 2 is asserting same thing as invariant 1.

   From invariant 2 to invariant 3. Result has been increased by i, and so if invariant 2 is true then invariant 3 must be true.

   From invariant 3 back to invariant 2, i is incremented by 1, and so the two invariants assert the same thing.

   From invariant 3 to invariant 4, i must get incremented past val, and so the previous value of i must have been val, and so invariant 3 asserts the same thing as invariant 4.

   From invariant 1 to invariant 4. Only happens of value is zero, and so result is zero factorial, which is correct.

9. Here is the code:

```
void insertionSort (double [ ] v) {
    int n = v.length;
    // inv 1: elements with index positions smaller than 0 are sorted
    for (int i = 1; i < n; i++) {
        // move element v[i] into place
        // inv 2: elements with index positions smaller than i are sorted
        double element = v[i];
        int j = i - 1;
```

```
            while (j >= 0 && element < v[j]) {
                v[j+1] = v[j]; // slide old value up
                j = j - 1;
            }
            // place element into position
            v[j+1] = element;
            // inv 3: elements with index positions smaller than
            // or equal to i are sorted.
        }
    }
```

all done?

10. At least one input that causes the loop to execute zero times (say an array with one element). At least one value that casues the loop to execute more than once (an array with several values). Elements in which the min is both the first value (so the conditional is always false) and not the first value (so that the conditional is sometimes true). An illegal value (say an empty array).

11. An input that causes the loop to execute zero times (either values 2 or 3). A value that causes the loop to execute more than once (say 12). A value in which the condition is always false (say 11). A value in which the condition is sometimes true (say 12). A value in which the test is true only on the last iteration (say 25).

12. An input in which the loop is never executed (zero length array). An input in which the loop is executed, but the test is always true (inserting at top). Same but always false (inserting at bottom). A loop with the element in the middle. An array where the element being sought occurrs several times. An array where the element being sought is in the middle but does not occur in the array.

13. Chapter 9?? What was I thinking??

6 Chapter 6 – Vectors

1. Table:

contents	size	capacity
(empty vector)	0	5
null, null, null, null	4	5
null, null, null, A	4	5
B, null, null, A	4	5
B, null, null	3	5
B, null, C	3	5
B, null, C, D	4	5
B, null, C, D, null, null	6	10

2. When the size is reduced an internal value is set and the elements above the new size are set to null. No recallocation occurs.

3. ```
   public synchronized void addElementAt (Object val, int index)
   {
       if ((index < 0) || (index > elementCount))
           throw new ArrayIndexOutOfBoundsException(index);
   // inv element has valid index
       setSize(elementCount+1);
   // there is room for the element
       for (int i = elementCount-1; i > index; i--)
           elementData[i] = elementData[i-1];
   // all elements above index have been moved up, leaving a hole
       elementData[index] = val;
   // element is inserted into the vector
   }
   ```

   Proof connecting invariants is left as exercise.

4. ```
   public synchronized void removeElementAt (int index) {
       if ((index < 0) || (index >= elementCount))
           throw new ArrayIndexOutOfBoundsException(index);
   // inv index is valid
       elementCount--;
   // size is one element smaller
       for (int i = index; i < elementCount; i++)
           elementData[i] = elementData[i+1];
   // values with index elements larger than size have been moved down
       elementData[elementCount] = null;
   // last position is now null
   }
   ```

	step	size	cost
	1	10	1
	2	10	1
	3	10	1
	4	10	1
	5	10	1
	6	10	1
	7	10	1
	8	10	1
	9	10	1
	10	20	20
	11	20	1
	12	20	1
5.	13	20	1
	14	20	1
	15	20	1
	16	20	1
	17	20	1
	18	20	1
	19	20	1
	20	40	40
	21	40	1
	22	40	1
	23	40	1
	24	40	1
	25	40	1
		Total	$83/25 = 3.32$

step	size	cost
1	10	1
2	10	1
3	10	1
4	10	1
5	10	1
6	10	1
7	10	1
8	10	1
9	10	1
10	11	11
11	12	12
12	13	13

6.

13	14	14
14	15	15
15	16	16
16	17	17
17	18	18
18	19	19
19	20	20
20	21	21
21	22	22
22	23	23
23	24	24
24	25	25
25	26	26
	Total	$289/25 = 11.56$

7.
```
class Vector {
    public Vector (Collection c) {
        setSize(c.size());
        int i = 0;
        for (Enumeration e = c.elements(); e.hasMoreElements();
)
            setElementAt(e.nextElement(), i++);
    }
}
```
Advantage, this form works for copying from other collections. Disadvantage, cannot be assigned to a Cloneable object.

8. Can generate a subscript error on collections of odd sizes

9. Element number 2 will be generated twice.

10. Assume the first method is inserting a 9 and the second is inserting a 6. At the point the first method is halted the array looks like 2 5 5 8 After the second halts it looks like 2 6 5 5 8. After the first completes again it will be 2 9 5 5 8. Note that the element 5 has been duplicated, and the element 6 lost.

11. Typo. Should be 36. The first will allocate a new array of size 20, which will overwrite the buffer. The second will generate a subscript error.

12.
```
3 2 4 1 0 5 6 7 8
2 3 1 0 4 5 6 7 8
2 1 0 3 4 5 6 7 8
1 0 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```

13. skipped

14. Example doesn't work. In fact, this version of bubble sort IS stable.

15. It is stable, since values move down until they reach a point where they are less than OR equal to another value. Two equal values will keep their relative ordering.

16. No it is not. Try sorting the array 4 2_1 2_2 6 5. After the sort with gap 2 we will have 2_2 2_1 4 6 5, and after the gap with 1 we will have 2_2 2_1 4 5 6, with the two elements interchanged.

17. Define the idea of "gap sorted" to mean that elements with the given gap are in order. Then using the proof similar to insertion sort, show that after each loop the values are gap sorted for the given gap. Once elements are gap sorted with gap one you are finished.

18. Assume that the array is held as a data field in a class, and we will use an inner class:
```
class DNAQuestion {
    private Vector data();  // original data values
    private Vector indirect(); // indirection values

    DNAQuestion () {
        int n = data.size();
        indirect.setSize(n);
        for (int i = 0; i < n; i++)
            indirect.setElementAt(new Integer(i), i);
    }
```

```
    public class Comp implements Comparator {
        public int compare(Object left, Object right) {
            Integer iLeft = (Integer) left;
            Integer iRight = (Integer) right;
            int i = iLeft.intValue();
            int r = iRight.intValue();
            for (int k = 0; k < m; k++)
                if (data.elementAt(i+k) != data.elementAt(j+k)
                    if (data.elementAt(i+k) < data.elementAt(j+k)
                        return -1;
                    else
                        return 1;
            return 0;  .// equal
        }
    }
}
```

So, sort the array using this comparator, then look at adjacent elements in the sorted array and see if they are equal.

7 Chapter 7 – Sorting Vectors

1. Here is the table

low	high	mid
0	15	7
0	7	3
0	3	1
2	3	2
3	3	done

2. log 10,000 or about 14.

3. The first, or lowest

4. Search for the lowest index, search for the highest index, return the values between the two index positions.

5. `3 1 4 1 5 break 9 2 6 5 3 5`
   `3 1 break 4 1 5 break 9 2 6 break 5 3 5`
   `3 break 1 break 4 break 1 5 break 9 break 2 6 break 5 break 3 5`
   `3 break 1 break 4 break 1 break 5 break 9 break 2 break 6 break 5 break 3 break 5 break`
   `1 3 break 4 break 1 5 break 9 break 2 6 break 5 break 3 5`
   `1 3 break 1 4 5 break 2 6 9 break 3 5 5`

```
1 1 3 4 5 break 2 3 5 5 6 9
1 1 2 3 3 4 5 5 5 6 9
```

6. Yes, it is stable. Two items will keep their relative order when they are merged back together.

7. Here are the invariants:

```
private void merge (int low, int mid, int high) {
// pre: elements from low to (mid-1) and from mid to high are sorted
    stk.setSize(0);
    int i = low;
    int j = mid;
    while ((i < mid) || (j < high)) {
        if (i < mid)
            if ((j < high) &&
                (test.compare(data.elementAt(j),
                    data.elementAt(i)) < 0))
// inv: the smallest (and hence next) element is at j
                stk.addLast(data.elementAt(j++));
            else
// inv: the smallest (and hence next) element is at i
// OR all of the 2nd part has been exhausted
                stk.addLast(data.elementAt(i++));
        else
// inv: elements in first part or exhausted, add elements from second
part
            stk.addLast(data.elementAt(j++));
    }
    j = stk.size(); // copy stack back to original area
    for (i = 0; i < j; i++)
        data.setElementAt(stk.elementAt(i), low+i);
}
```

8. If the data is sorted partition yields one group with size zero and another with size n-1, and repeating this gives the poor $O(n^2)$ performance for quick sort.

9. If the element indexed by low is properly in the first group, low is incremented. Otherwise the element indexed by low is larger than or equal to the pivot value. In that case, low will not be modified, but the 2nd if statement will be tested. If the element indexed by high is larger than the pivot element, then high can be decremented. Otherwise the element with index position high is also out of order, and the two elements can be swapped.

   Remainder of proof is left as exercise.

10.
```
private void quickSort (int low, int high) {
        // no need to sort zero or one elements
    if (low >= high) return;
        // select a pivot and partition
    int pivotIndex = (low + high) / 2;
    pivotIndex = pivot(low, high, pivotIndex);
// inv: elements less than pivotIndex are all smaller than all
// elements with index values larger than pivotIndex
        // then sort two subarrays
    quickSort(low, pivotIndex);
    quickSort(pivotIndex + 1, high);
}
```

11. No. Consider sorting the following 5 element vector 2 4_1 4_2 1 6. After the first partition the elements will be 2 1 4_2 4_1 6 and there is no way the two elements will ever be rearranged.

12. Here are some of the steps

```
Initially will pivot around the nine,
first moving it to the first location
9 1 4 1 5 3 2 6 5 3 5
then reorganizing, but the 2nd group is empty
and so after the last swap we have
5 1 4 1 5 3 2 6 5 3 9
Now the elements indexed 0 to 9 are sorted, pivot
is in position 4, which is 5.  Swap to first location
leaves same array.  After partitioning we have two
groups as follows
5 1 4 1 3 3 2 | 6 5 5 9
after final swap we have
2 1 4 1 3 3 | 5  6 5 5 9
Each group is then sorted (we will do it in parallel)
first moving pivot to front
4 1 2 1 3 3 | 5 6 5 5 9
then partitioning (not we get two empty sets)
4 1 2 1 3 3 | (empty) | 5 (empty) | 5 5 6 9
then swapping pivot back
3 1 2 1 3 | 4 | empty | 5 | 5 5 6 9
Again,
2 1 3 1 3 | 4 | | 5 | 5 5 6 9
after partition
2 1 1 | 3 3 | 4 | | 5 | 5 | empty + 5 6 9
after final swap
1 1 | 2 | 3 3 | 4 | | 5 | 5 | | 5 6 9
Last few steps are base case, leaving final answer
1 1 2 3 3 4 5 5 5 6 9
```

13. Quick sort will be $O(n^2)$, insertion sort is $O(n)$ Problem is one of the two partitions is always empty.

14. Quick sort is $O(nlogn)$ in this case would be $O(n^2)$ if we selected first element .

15. See question 13.

16. low + high (since either low is incremented or high is decremented)

17. Look for the kth smallest. The lower partition provides the answer. Expected complexity is $O(\log n)$.

18. Finding and removing largest would be both $O(1)$.

8 Chapter 8 – Linked Lists

1. ```
   ConsCell lst = new ConsCell(7, null)
   list = new ConsCell(2, lst)
   lst = new ConsCell(6, lst)
   lst = new ConsCell(3, lst)
   ```

2. Recursive calls would finally add 7 to original list of 6 and 3.

3. pictures

4. pictures

5. You get a NoSuchElementException. RemoveLast needs to explicitly check the condition before it access the element prev.

6. addElement is O(1) always containtsElement is O(n) worst case findElement is O(n) worst case removeelement is O(n) worst case

7. You get a null pointer exception. After hasMoreelements has said there are more elements, nextElement trys to read it, but by then there is a null pointer.

8. An enumeration doesn't allow us to get hold of the link we need to delete.

9. ```
   public synchronized void removeElement (Object newValue) {
       for (DoubleLink p = firstLink; p != sentinel; p = p.next)
   {
           // inv: have not seen element yet, but maybe it's p
           if (newValue.equals(p.value)) {
               // have found element, must remove it
               p.remove();
               return;
   ```

```
            }
            // inv: have not yet seen element
        }
        throw new NoSuchElementException(newValue.toString());
    }
```

10. ```
    class LinkedList {
        public boolean equals (Object right) {
            Collection rc = (Collection) right;
            for (Enumeration e = elements(); e.hasMoreElements();
    )
                if (! rc.contains(e.nextElement()))
                    return false;
        }
    }
    ```

    It is much harder if elements are not unique, as then you have to count the occurresnces of elements.

11. Here is elementAt, others are similar, but are all O(n)

    ```
    class LinkedList {
        public Object elementAt (int index) {
            for (Enumeration e = elements(); e.hasMoreElements();
    ) {
                Object obj = e.nextElement();
                if (index-- = 0) return obj;
            }
            throw NoSuchElementException();
        }
    }
    ```

9 Chapter 9 – List Variations

1. Since indexing is $O(n)$, searching for an element would take $O(N \log N)$, probably not a good idea since it is even slower than the regular $O(N)$ search.

2. This is relatively easy if you draw pictures that show the state of the computation at each point.

3. Can be done in O(n), and doesnt require any restriction on elements in the lists.

   ```
    class SortedList {
       boolean equals (Object right) {
           SortedList sright = (SortedList) right;
           if (size() != sright.size()) return false;
   ```
   ```

```
         Enumeration e = elements();
        Enumeration f = sright.elements();
        while (e.hasMoreElements() && f.hasMoreElements())
            if (! e.nextElement().equals(f.nextElement()))
                return false;
        return true;
    }
}
```

4. Would be $O(n^2)$ for a List, $O(n^2)$ also for a SortedVector.
5. Can't get access to the undelying links of you use an enumerator
6. Yes, use findElement and check if the result is non-null.
7. See chapter 15.
8. $O(n \log n)$
9. Here is the table:

Container	Addition	Search	Removal
SkipList	$O(\log n)$ Expected	$O(\log n)$ Expected	$O(\log n)$ Expected
SortedVector	$O(n)$ Worst	$O(\log n)$ Always	$O(n)$ Worst
SortedList	$O(n)$ Worst	$O(n)$ Worst	$O(n)$ Worst
LinkedList	$O(1)$ Always	$O(n)$ Worst	$O(n)$ Worst

10. Lots of different answers, depending upon the toss of the coin. All should look roughly like the picture in the book.
11. My counts are as follows: 9 8 9 8 9 8 9 8 7 7 6 5 9 8 7 6 6 5 5 5 for a total of 142 and average of 7.1

10 Chapter 10 – Stacks

1. (a) 3, a = 3, 3 5, 3

   (b) 3, 3 4, 3, empty, error
2. Here is the table

	Vector	LinkedList
addLast	$O(1)$ expected	$O(1)$ always
getFirst	$O(1)$ always	$O()$ always
removeFirst	$O(1)$ always	$O(1)$ always

3.
```
class StackAdapter {
      public StackAdapter (Stack d) { data = d; }
      private Stack data;

      public void  push (Object obj) { data.addLast(obj); }
      public void pop () { data.removeLast(); }
      public Object top () { return data.getLast(); }
  }
```

4.
```
private class MouseKeeper extends MouseAdapter {
      CardPile sourceDeck = null;

      public void mousePressed (MouseEvent e)
          { sourceDeck = findDeck(e.getX(), e.getY()); }

      public void mouseReleased (MouseEvent e) {
          if (sourceDeck == null) return;
          if (sourceDeck.isEmpty()) return;
          // inv: source deck is a valid deck with at least one card
          CardPile toDeck = findDeck(e.getX(), e.getY());
          if (toDeck == null) return;
          // to deck is a valid card deck
          Card playCard = sourceDeck.topCard();
          if (playCard == null) return;
          if (toDeck.canTake(playCard))
              // todeck can take the top card from source deck
              toDeck.addCard(playCard);
          else
              // todeck cannot take top card, so place it
              // back on source deck
              sourceDeck.addCard(playCard);
          repaint();
      }
  }
```

5. When you see a left paren, push the corresponding right paren (of the correct type). When you see a right paren, see if the correspodning item on the stack is the correct type. If so, pop stack and get next char, if not, report an error.

6. a b c + * b d / + a *

7. $-(7-5)$ or $7-(-5)$

8. Left as a programming exercise.

9. assume there is a number in the display, enter 0, enter 1, enter minus (leaving -1) enter multiply.

10. $O(n^2)$

11.
```
void shuffle (vector v) {
    int n = v.size();
    for (int i = 0; I  < n; i++) {
        int j = (int) ( n * Math.random());
        Object temp = v.elementAt(i);
        v.setElementat(v.elementAt(j), i);
        v.setElementAt(temp, j);
}
```

12. No., if it becomes full then he cannot get all cars. He must leave one less than the depth of the parking area (depth 3 in the picture, so he must leave 2 spaces).

13. As cars come from left, if they are not the next car in sequence then push on to siding, until car is reached. Thereafter, if the next car in sequence is already on siding, push back to right until it can be moved. Otherwise keep moving cars from left onto siding. Worst case is $O(n^2)$, since N cars may have to be moved to find next one.

14. Certainly if they are either strictly increasing or strictly decreasing they will work. Also any group that can be divided into parts that are strictly increasing or strictly descreasing (with no elements out of order within the group).

15. not done, simple trace of procedure calls.

16. not done, simple trace of procedure calls.

11 Chapter 11 – Deques

1. Put each letter into the deque until all letters are recognized. Then while the size of the deque is larger than 2, pull items from both front and back. If they differ, then word is not palendrome. If you get down to zero or one letters and have not found any difference, then word is palendrome.

2. Here is the table.

	LinkedList	IndexedDeque
addFirst	$O(1)$ always	$O(1)$ expected
addLast	$O(1)$ always	$O(1)$ expected
getFirst	$O(1)$ always	$O(1)$ always
getLast	$O(1)$ always	$O(1)$ always
removeFirst	$O(1)$ always	$O(1)$ always
removeLast	$O(1)$ always	$O(1)$ always

3. Both wrap around. However, one uses a linked list as the underlying data structure, the other uses an array. One is indexed, one is not.

4. The stack version works in one place until finished, the deque version moves around the entire snowflake doing a little bit at each point until finished.

5. Lots of different answers

6. (a) O(1): elementAt, setElementAt O(N): setSize, ensureCapacity, addElementAt, removeElementAt

   (b) The division operator is necessary to wrap around when the firstfilled field occurrs later than the first empty field.

   (c) i. The elementCount is smaller, but the firstFilled does not change.
   ii. the elementCount becomes larger, but the buffer is not changed
   iii. A new buffer is created and the first filled position becomes zero, the elementCount indicates the number of values.

   (d) The method simply returns

   (e) picture

   (f) You could get a capacity that was larger than the buffer size.

   (g) The worst case is $O(n)$, this happens when an insertion forces a reallocation of a new buffer.

   (h) This allows adding a method at the end of the data area.

   (i) Worst case is O(N), occurs when removing the 2nd element and all values smaller than must be moved down.

   (j) Lots of possible answers, all having the characteristic that they leave the buffer in an inconsistent state.

7. Doubling the size makes the next few additions $O(1)$. If we simply added 1 one element, each additional increase in size would be $O(N)$.

8. (a) Easy answers modelled on those in text.

   (b) because allElementAt is synchronized

(c) $O(1)$. using allElementAt would have moved values unnecessarily, rather than moving the firstFilled pointer down.

(d) $O(1)$. Using removeElementAt would have made it $O(N)$ worst case.

9. If the last value is removed, the hashMoreElements will have reported true, but the invocation of nextElement will try to access a value that is not there.

12 Chapter 12 – Queues

1. (a)
```
que.addLast(new Integer(2));
// queue is 2
Integer a = que.getFirst();
// a is 2, queue is 2
que.addLast(new Integer(3));
// a is 2, queue is 2 3
que.removeFirst();
// a is 2, queue is 3
```

(b)
```
que.addLast(new Integer(2));
Integer a = que.getFirst();
// a is 2, queue is 2
que.addLast(new Integer(4));
// a is 2, queue is 2 4
que.removeFirst();
// a is 2, queue is 4
que.addLast(new Integer(6));
// a is 2, queue is 4 6
que.removeFirst();
// a is 2, queue is 6
```

(c)
```
que.addLast(new Integer(2));
// queue is 2
que.getFirst();
// queue is 2
que.addLast(new Integer(3));
// queue is 2 3
que.removeFirst();
// queue is 3
que.removeFirst();
// queue is empty
Integer a = que.getFirst();
// error
```

2. Here is the table.

	LinkedList	IndexedDeque	RingBuffer
addLast	$O(1)$ always	$O(1)$ expected	$O(1)$ always
getFirst	$O(1)$ always	$O(1)$ always	$O(1)$ always
removeFirst	$O(1)$ always	$O(1)$ always	$O(1)$ always

3. The objects are pastries, boxes, machines and queues. not done

4. Remove each element from the front of the queue and place them into the stack, then remove each element from the stack and push them into the back of the queue.

5. As each character is read, push the character on to both the stack and the queue. After the last character, read pop values from both stack and queue. If they disagree, it is not a palendrome. If you reach the end without finding a disgreement, they match.

6. All the elements from 2nd position upwards get moved down.

7. The sum 1 + 2 + 3 + ... + n is still $O(n^2)$.

8. ```
   public class RingBufferQueue implements Queue {
       public Enumeration elements () { return new RingBufferEnumerator();
   }

       ...

       private RingBufferEnumerator implements Enumeration {
           public RingBufferEnumerator () { ptr = firstFilled; }
           private RingBufferNode ptr;

           public boolean hasMoreElements()
               { return ptr != firstFree; }

           public Object nextElement () {
               Object result = ptr.value;
               ptr = ptr.next;
           }
       }
   }
   ```

9. experimental results will differ depending upon initial conditions.

10. ```
    public class RingBufferQueue implements Queue {

        RingBufferQueue () {
            firstFilled = firstFree = new RingBufferNode(null);
            firstFree.next = firstFree;
    ```

```
        }
    private RingBufferNode firstFree, firstFilled;
    private count = 0;

    public boolean isEmpty () { return count == 0; }

    public int size () { return count; }

    public synchronized void addLast (Object val) {
        count++;
        if (firstFree.next == firstFilled)
            firstFree.next = new RingBufferNode(firstFree.next);
        firstFree.value = val;
        firstFree = firstFree.next;
    }

    public Object getFirst () {
        if (count == 0) throw new NoSuchElementException();
        return firstFilled.value; }

    public synchronized void removeFirst () {
        if (count == 0) throw new NoSuchElementException();
        count--;
        firstFilled = firstFilled.next;
    }
}
```

11. experimental results will differ depending upon initial conditions.

12. Programming exercise left to reader.

13. experimental results will differ depending upon initial conditions.

14. This is actually a bit tricky. Requires making double links that point both forward and backward.

15. Programming exercise left to reader.

13 Chapter 13 – Trees

1. Many different answers

2. picture

3. `prefix: * + a b + c * d e`
   `postfix: a b + c d e * + *`

4. picture
5. `preorder: A B D H I E C F G`
   `inorder: H D I B E A F C G`
   `postprder: H I D E B F G C A`
   `level order: A B C D E F G H I`
6. Need to compare the precidence of interior nodes.
7. Picture
8. The root has no sibling, so has no right node. The collection can be viewed as siblings of the root tree.

14 Chapter 14 – Search Trees

1. Minimum is the ceiling of the log, which is 100. Maximum is 100.
2. The tree is pretty unbalanced, with only a couple of 1's hanging down the left side of the right child. Removing the root doesn't improve balance any. picture
3. There are 24 different permutations. Of these, only six appear more than once. (Drawing the trees is left as an exercise, but I've divided into the equivalance groups).
   `1 2 3 4`
   `1 2 4 3`
   `1 3 2 4, 1 3 4 2`
   `1 4 2 3`
   `1 4 3 2`
   `2 1 3 4, 2 3 1 4, 2 3 4 1`
   `2 1 4 3, 2 4 3 1, 2 4 1 3`
   `3 1 2 4, 3 1 4 2, 3 4 1 2`
   `3 2 1 4, 3 4 2 1, 3 2 4 1`
   `4 1 2 3`
   `4 1 3 2`
   `4 2 1 3, 4 2 3 1`
   `4 3 1 2`
   `4 3 2 1`
4. $O(N)$ simply enumerate the two trees and compare elements. Don't think it can be done any faster.
5. Left as exercise
6. Opps. Same as question 3.

7. I believe it is 1, 2, ... n. (anybody have a proof?)

8. Part a is simple calculation. Here is part b.

$$\begin{aligned} M_{h+1} &= M_{h-1} + M_h + 1 && (1)\\ &= f_{h+2} - 1 + f_{h+3} - 1 + 1 && (2)\\ &= f_{h+2} + f_{h+3} - 1 && (3)\\ &= f_{h+3} - 1 && (4)\\ & && (5) \end{aligned}$$

9. Base cases are proved by inspection.

$$\begin{aligned} f_{n+1} &= f_{n-1} + f_n && (6)\\ &< 2^{n-1} + 2^n && (7)\\ &< 2^n + 2^n && (8)\\ &< 2 \times 2^n && (9)\\ &= 2^{n+1} && (10)\\ & && (11) \end{aligned}$$

10. Since we are fudging, we will use $\approx$ instead of $<$. We tosss out the constant term as it becomes unimportant as h gets large.

$$\begin{aligned} M_h &\approx 2^{h+3} && (12)\\ \log M_h &\approx (h+3)\log 2 && (13)\\ \log M_h &\approx h\log 2 + 3\log 2 && (14)\\ \log M_h &\approx h\log 2 && (15)\\ \frac{\log M_h}{\log 2} &\approx h && (16)\\ & && (17) \end{aligned}$$

11. (a) for n = 0 we get $1 - 1$ is 0, and thus FIB_n is 0. For n = 1 the inner formula reduces to $\frac{2\sqrt{5}}{2}$, and thus we have FIB_2 is 1. For n = 2 we have $\frac{1}{\sqrt{5}}(\frac{1+2\sqrt{5}+5}{4} - \frac{1-2\sqrt{5}+5}{4})$, which reduces to $\frac{1}{\sqrt{5}}(\frac{6+2\sqrt{5}}{4} - \frac{6-2\sqrt{5}}{4})$, and thus to $\frac{1}{\sqrt{5}}\frac{4\sqrt{5}}{4}$, or 1.

    (b) $(\frac{1+\sqrt{5}}{2})^2$ is $\frac{1+2\sqrt{5}+5}{4}$, which is $\frac{6+2\sqrt{5}}{4}$, or $\frac{4}{4} + \frac{2+2\sqrt{5}}{4}$, or $1 + \frac{1+\sqrt{5}}{2}$.

    (c) $Fib_{n+1} = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1})$
    $= \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n-1}(\frac{1+\sqrt{5}}{2})^2 - (\frac{1-\sqrt{5}}{2})^{n-1}(\frac{1-\sqrt{5}}{2})^2)$

$= \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n-1}(1 + \frac{1+\sqrt{5}}{2}) - (\frac{1-\sqrt{5}}{2})^{n-1}(1 + \frac{1-\sqrt{5}}{2}))$
$= \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n-1} - (\frac{1-\sqrt{5}}{2})^{n-1}) + \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{n} - (\frac{1-\sqrt{5}}{2})^{n})$
$= Fib_{n-1} + Fib_n$

(d) As the second term is less than 1, as n gets large it gets closer and closer to zero.

(e) c is approximately 1.618.

(f) $\log c$ is approximately 0.694242

(g) $\frac{1}{\log c}$ is approximately 1.44.

12. (a) We need to examine each case in turn.

    i. h(y) = h(z). Therefore the right size has height h(y)+1. Since we know this is equal to h(x)+2, we have h(y) = h(x)+1. After rotation the left child is balanced, since one child has height h(x)=h(y)-1, and one child height h(y). The height of the left child is h(y)+1, which is h(z)+1, ahd the right child h(z). Therefore we have balance.

    ii. h(y)+1 = h(z). Therefore the two children of B have heights h(y) and h(y)+1, and the height of B is h(y)+2. Since h(y)+2 = h(x)+2, we have h(x)=h(y). After rotation the left child is balanced (they have exactly the same height) and the height of A is h(y)+1. But this is the same as h(z), so everything is perfectly balanced.

    iii. h(y) = h(z)+1. Using the same logic as in the 2nd case, we have that h(x) = h(z). After rotation the two children of A will have heights h(z) and h(z)+1, That's ok. But the height of A is h(z)+2. This is 2 levels higher than the height of the right child.

    (b) Break Y into two subtrees of height Y1 and Y2, then consider the three cases that h(y1) = h(y2), h(y1)+1 = h(y2) and h(y1) = h(y2)+1. Analyze each of these three cases in turn and you will see that everything works fine.

13. No, it is not stable, the two values will have interchanged in the sort.

15 Chapter 15 – Priority Queues

1.
```
class ListPriorityQueue implements FindMin {
    public ListPriorityQueue (Comparator t) { test = t; }

    private LinkedList data = new LinkedList();
    private Comparator test;
```

```
    public void addElement (Object newElement) { data.addLast(newElement);
}

    public Object getFirst () {
        Object result = data.getFirst();
        Enumeration e = data.elements();
        while (e.hasMoreElements()) {
            Object obj = e.nextElement();
            if (test.compare(result, obj) < 0)
                result = obj;
        }
        return result;
    }

    public void removeFirst () {
        Object result = getFirst();
        data.removeElement(result);
    }

}
```

Nothing is said about it being fast!

2. Lots of answers

3. These just involve walking through the code. No paper answers can be given.

4. The subscript error will throw the exception

5. Solve by induction. For height 0 there is only 1 tree. For height 1 there are two trees, since the two children can be interchanged. For height 2 there are 8 trees. For height 3 there are 8 (or 2^3) rearragnements of the left subtree. For each of these there are 8 rearrangments of the right subtree. Since these can be done independently of each other, there are 2^6 possible orderings. But we can also interchange the right and left subtrees, therefore there are really 2^7 possible orderings. If we go one level further, we see that for the next level there are $2^7 \times 2^7 \times 2$ or 2^{15} possible orderings.

   Now the trick here is coming up with an induction hypothesis. To do that, let us look not at the values themselves, but at their powers (since they are all powers of two). We have that N(0)=2^0, N(1)=2^1, N(2)=2^3, N(3)=2^7, N(4)=2^{15}. To a computer scientist, the pattern 0, 1, 3, 7, 15 should have an obvious meaning, namely $2^n - 1$.

   So our induction hypothesis is that for a complete heap of height h there are 2^{2^h-2} possible different rearrangements that can be made simply by

reversing the order of children. Let us call this R(h). (Note that this isn't all the possible trees, because we haven't considered the possibility of moving children up or down to different levels).

For a heap of height $h+1$ we have R(h+1) rearragnements. But we can rearrange either child independently, and exchange the children. Thus the number of rearrangements is $R(h) \times R(h) \times 2$. By our induction hypothesis this is $2^{2^n-1} \times 2^{2^n-1} \times 2$, which is $2^{2^{n+1}-1}$ and we are done.

6. Its takes more execuition time.

7. The only way is to insert values one by one from the 2nd heap into the first, a $O(n \log n)$ process. This is much slower than the $O(\log n)$ merge process for the skew heap.

8. No, they will not be.

9. No heapsort is not stable.

10. Left as exercise, just drawing pictures of execution.

11. This will be $O(n \log n)$, but in practice not as fast as heap sort.

12. not done

13. The maximum could potentially be any leaf value. Since for a heap with N nodes there are $N/2$ leafs, finding the correct one must be $O(N)$.

14. Not done, but basically a heapify variation.

15. Decrease the priority of a task every time it is executed. Eventually it will become low priority, and another task will be performed.

16 Chapter 16 – Hash Tables

1. Because the vector index values, computed by taking the hash value and computing the remainder after dividing by the vector size, would then be wrong. In particular, both amy and andy would hash into location 3, and nobody would hash into location 2.

2. Here is the data

name	hash value	bucket if size 5	bucket if size 11
Abel	4	4	4
Aspen	15	0	4
Albert	1	1	1
Amanda	0	0	0
Angela	6	1	6
Andreas	3	3	3
Abigail	8	3	8
Adrian	17	2	6
Alastair	0	0	0
Alexis	4	4	4
Anita	8	3	8
Arthur	19	4	8
Abraham	17	2	6
Adrienne	17	2	6
Angelique	6	1	6
Andrew	3	3	3
Anne	13	3	2
Audrey	3	3	3
Ada	0	0	0
Agnes	13	3	2
Amina	8	3	8
Alaric	0	0	0
Antonia	19	4	8

For 5 buckets we have element counts 5, 3, 3, 8, 4. For 11 buckets we have element counts 4 1 2 3 3 0 5 0 5 0 0, which seems hardly random.

3. rank * 4 + suit will do it, although this then messes up getting the suit values.

4. Because the sin function returns a value between 0 and 1, all the index values would be zero.

5. Left as an exercise.

6. Too many possible answers depending upon the telephone numbers selected.

7. Experiment. Usually works for any group of 24 or more people.

8. The probability that a single person does not have your birthday is $\frac{364}{365}$. Since the probability of each person not sharing your birthday is the same, the total probability is $1 - (\frac{364}{365})^{24}$, which is about 0.07.

9. Use the difference between the smallest and largest as the counting array size, then subtract the smallest from each element to get an zero-based index.

10. Go through the array once to find the smallest and largest, then build the counts.

11. Yes, since equal elements will always hash the same, and their relative positions are always preserved.

12. Because all the values are in order in the first bucket already.

13. Because an element may not hash into the same bucket if the table size is changed.

14. Count the number of elements examined, when the count reaches zero you know you have examined the entire table.

15. If C is relative prime with the table size it will examine all elements before it returns to the same location.

16. The clusters are still there, but just not right next to each other.

17. Even though Anne is in the table you cannot find it, because you encounter a null value in the search before you find it.

18. All elements would hash into the same bucket.

17 Chapter 17 – Maps

1. Amy could use the name as the map key and the value as the amount paid.

2. To determine if a map has a value there is no easy way other than to search the entire map, $O(N)$.

3. lots of answers.

4. Easy to generate the parse trees.

5. The chance of selecting "I do something" is 1/4. The change of selecting "love" is 1/3. The change of selecting "computer" is 1/4, for a total of 1/48.

6. Here is the chart.

	get	set	containsKey	removeKey
LinkedList	$O(n)$	$O(n)$	$O(n)$	$O(n)$
SortedVector	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
SortedList	$O(n)$	$O(n)$	$O(n)$	$O(n)$
SkipList	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
AVLTree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
OpenHashtable	$O(1)$	$O(1)$	$O(1)$	NA
Hashtable	$O(1)$	$O(1)$	$O(1)$	$O(1)$

7. picture

8. picture

9. This can be done by copying the concordance information into a new set that is ordered by the word counts.

10. Easy to do by reading in another set and doing set operations.

18 Chapter 18 – Sets

1. ```
   public void unionWith (Set aSet) {
       for (Enumeration e = aSet.elements(); e.hasMoreElements();
   )
           addElement(e.nextElement());
       // inv: all elements from argument now appear in set
   }

   public boolean subsetOf (Set aSet) {
       for (Enumeration e = elements(); e.hasMoreElements(); )
           if (! aSet.containsElement(e.nextElement()))
               // there exists an element in our set
               // that is not in argument set
               return false;
       // all elements from our set are in argument set
       return true;
   }
   ```

2. Just one symbol different from intersectWith:
   ```
   public void differenceWith (Set aSet) {
       Bag removedItems = new LinkedList();
           // find elements to be removed
       Enumeration e;
       for (e = elements(); e.hasMoreElements(); ) {
           Object val = e.nextElement();
   ```
   ```

```
        if (aSet.containsElement(val))
            removedItems.addElement(val);
    }
    for (e = removedItems.elements(); e.hasMoreElements(); )
        removeElement(e.nextElement());
}
```

3. The chapter in Table 18.1 makes this easier. Should also ask for worst case, every case, average, etc.

	addElement	unionWith	intersectionWith	differenceWith	subsetOf
LinkedList	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
SortedVector	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
SortedList	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
SkipList	$O(\log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
AVLTree	$O(\log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
OpenHashtable	(1)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Hashtable	(1)	$O(n)$	$O(n)$	$O(n)$	$O(n)$

4. `union : 1 2 5 7 12 36 52`
   `intersection : 1 5 12`
   `difference: 2 7`

5. It's the union of the two asymmetric differences

6. If A is a subset of B then every element in A is in B, and the interseciton of A and B will at least have all the elements of A. Now assume that the inter section of A and B is equal to A. Then every element in A must be in B, and hence A is a subset of B. But this is a very slow way to compute the subset.

7. Here is the program augmented with invariants

```
public void differenceWith (Set newSet) { // assumes newSet is sorted
    DoubleLink ptr = elementData.firstLink;
    Sorted argSet = (Sorted) newSet;
    for (Enumeration e = argSet.elements(); e.hasMoreElements();
) {
        Object newElement = e.nextElement();
        while (true)
            if (ptr == elementData.sentinel)
                // have seen all elements in set, can quit
                break;
            else if (test.compare(ptr.value, newElement) < 0)
                // element referenced by pointer does not
```

```
                // occur in argument set, and therefore
                // should be maintained in the difference
                ptr = ptr.next;
            else {
                if (test.compare(newElement, ptr.value) < 0)
                    // element referenced by pointer
                    // does not occur in the argument set,
                    // and therefore should be maintained
                    // in the difference set
                    break;
                // element referenced by pointer is equal
                // to element in the argument set,
                // and therefore must be removed from
                // the difference set
                DoubleLink rptr = ptr;
                ptr = ptr.next;
                rptr.remove();
            }
        // ptr is either at sentinel or
        // ptr is larger than current enumeration element
        // so enumeration should be advanced.
    }
}
```

8. Here is the code with invariants:

```
public void intersectWith (Set newSet) { // assumes newSet is sorted
    DoubleLink ptr = elementData.firstLink;
    Sorted argSet = (Sorted) newSet;
    LinkedList intersect = new LinkedList();
    for (Enumeration e = argSet.elements(); e.hasMoreElements();
) {
        Object newElement = e.nextElement();
            // skip smaller elements
        while ((ptr != elementData.sentinel) &&
            (test.compare(ptr.value, newElement) < 0))
                ptr = ptr.next;
            // have either reached the sentinel, or
            // element is in both sets
            // save if in both sets
        if ((ptr != elementData.sentinel) &&
            ptr.value.equals(newElement))
                // element is in both sets, should be
                // added to intersection list
                intersect.addLast(ptr.value);
```

```
    }
        // change elements to intersection
    elementData = intersect;
}
```

9. Very similar to difference and union:

```
public boolean subsetOf (Set newSet) { // assumes newSet is sorted
    DoubleLink ptr = elementData.firstLink;
    Sorted argSet = (Sorted) newSet;
    for (Enumeration e = argSet.elements(); e.hasMoreElements();
) {
        Object newElement = e.nextElement();
        if (test.compare(newElement, ptr.value) < 0)
            // value referenced by ptr is larger than
            // enumeration, advance to next enumeration value
            break;
        if (test.compare(ptr.value, newElement)  < 0)
            // value referenced by ptr is smaller than
            // enumeration, and therefore cannot occur in
            // enueration set (since it is sorted).
            // found an element not in enumeration set
            return false;
        // value referenced is equal to enumeration value
        // therefore occurs in both sets, advance both
        // pointer and enumeration
        ptr = ptr.next;
    }
    // all values occur in argument set, therefore we are
    // a subset of argument set
    return true;
}
```

10. The subset relationship is less-than or equal, not less-than. To determine less-than the comparitor would have to test both subset and equality in order to determine which answer to give.

11. If there is an element of A not in B then the union will be larger than B. But this is a slow way to compute subset.

12. If either has an element not in the other then one of the subsets will fail – if neither fails they must be equal.

13. Yes, a skip list would be faster, but would require that elements be ordered and would require a comparator object.

14. Finding the root causes the trees to be wider and more shallow. If you don't find the root you can get deeper trees, which may slow down processing depending upon the size of the data.

15. Complexity is determined by the number of integers in the set, not by the number of values that are 1.

16. The hash value would continually change as the bit set changed.

17. No easy way to record more than whether or not the bit is set.

19 Chapter 19 – Matrices

1. element i, j is found at $i * m + j$. Notice that the number of rows has no impact on this calculation.

2. See errata. This question doesn't make sense if Matrix is an interface and not a class.

3. Not done

4. Simply check if $i < j$ then reverse and ask for element j, i.

5. The key insight is to see that if you examine the middle-middle element, either it is smaller than the value you see, in which case you have eliminated the entire upper left quadrant, or it is larger than the element you seek, in which case you have eliminated the entire lower right quandrant. However, you know nothing about the other three quandrants. Still, this can be used to find a fast algorithm. Simply make three recursive calls, and report true if any of them offer success.

20 Chapter 20 – Graphs

1. Every vertex must have at least one edge, so $O(n)$ is a lower bound.

2. Using a matrix it is $O(1)$. Using an edge list it could be as bad as $O(N)$. Using a sparse matrix could be $O(\log n)$.

3.

	0.	1.	2.	3.	4.	5.	6.
0.	0	1	1	0	0	0	0
1.	1	0	0	1	0	0	0
2.	1	0	0	0	1	0	1
3.	0	1	0	0	1	1	0
4.	0	0	1	1	0	1	0
5.	0	0	0	1	1	0	1
6.	0	0	1	0	0	1	0

4. 0: edges to 1 and 2
   1: edges to 0 and 3
   2: edges to 0, 4 and 6
   3: edges to 1, 4 and 5
   4: edges to 2, 3 and 5
   5: edges to 3, 4 and 6
   6: edges to 2 and 5

5. Since very subscript is potentially $O(\log n)$, then time time would be $O(n^2 \log n)$. Uses at most $O(n^2)$ space.

6. If there is a loop, snip it out and there is still a path.

7. The question is ambiguous if you don't specify a starting vertex. assume we start in Pendleton. 1. Pendelton, 2. Phonix, 3. Pittsburg, 4. Pensacola, 5. Peroria, 6. Peueblo, 7. Pierre

8. The assumption is that the elements of the set can be tested in $O(1)$ time.

9. Similar to earlier question.

10. Not done

11. This information can be copied from Table 20.1

12. Yes, the speed with which you can perform the containsKey operation will impact the running time. But speeding up this operations would require that WVertices be ordered, something they are not in this version.

13. Assign letters A to E to the vertices starting from the upper left and moving clockwise. Start from vertex A. Dijkstras algorithm will then travel from A to B, listing the cost of reaching vertex B as 10. It will then travel from B to C, and from C to D, yielding costs of 11 and 12, respectively. It is only then that it will travel from A to E (cost 14), and finally from E to B. The latter results in a path with cost 8, which is less than the previous cost for a path from A to B. So not only is the previously recorded cost of the path from A to B wrong, but so are the costs of the paths from A to C and from A to D.

14. Many possible answers

15. Start from a node, and move along edges. If the graph contains n nodes, then in n steps you must either find a node you have visited (therefore the graph has cycles) or a node with no successor (therefore there is a node with no edges).

16. $O(n \log n)$

17. Many possible answers.