PEARSON NEW INTERNATIONAL EDITION

Data Structures and Problem Solving Using Java

Mark A. Weiss

Fourth Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

Data Structures and Problem Solving
Using Java
Mark A. Weiss
Fourth Edition

PEARSON

12 `intersect`, shown below, returns the number of elements that are in both lists. Assume both lists contain *N* items.

```
// Returns the number of elements in both c1 and c2
// Assumes no duplicates in either list.
public static int intersect ( List<Integer> c1, List<Integer> c2 )
{
    int count = 0;

    for( int i = 0; i < c1.size( ); i++ )
    {
        int item1 = c1.get( i );
        for( int j = 0; j < c2.size( ); j++ )
        {
            if( c2.get( j ) == item1 )
            {
                count++;
                break;
            }
        }
    }

    return count;
}
```

a. What is the running time of `intersect` when both lists are `ArrayLists`?
b. What is the running time of `intersect` when both lists are `LinkedLists`?
c. Suppose it takes 4 seconds to run `intersect` on two equally-sized 1,000-item `LinkedLists`. How long will it take to run `intersect` on two equally-sized 3,000-item `LinkedLists`?
d. Does rewriting the two loops using the enhanced for loop (i.e. `for( int x : c1 )`) make `intersect` more efficient? Provide an explanation of your reasoning.

13 `containsAll` returns true if the first list contains all the elements in the second list. Assume the lists are approximately the same size and have about *N* items each.

```
public static boolean containsAll( List<Integer> bigger,
                                   List<Integer> items )
{
  outer:
    for( int i = 0; i < bigger.size( ); i++ )
    {
        Integer itemToFind = bigger.get( i );

        for( int j = 0; j < items.size( ); j++ )
            if( items.get( j ).equals( itemToFind ) ) // match
                continue outer;

        // If we get here, no entry in items matches bigger.get(i)
        return false;
    }
    return true;
}
```

a. What is the running time of `containsAll` when both lists are `ArrayLists`?
b. What is the running time of `containsAll` when both lists are `LinkedLists`?
c. Suppose it takes 10 seconds to run `containsAll` on two equally-valued 1000-item `ArrayLists`. How long will it take to run `containsAll` on two equally-valued 2000-item `ArrayLists`?
d. Explain in one sentence how to make the algorithm efficient for all types of lists.

14 `containsSum`, shown below, returns true if there exists two unique numbers in the list that sum to *K*. Assume *N* is the size of the list.

```
public static boolean containsSum( List<Integer> lst, int K )
{
    for( int i = 0; i < lst.size( ); i++ )
        for( int j = i + 1; j < lst.size( ); j++ )
            if( lst.get( i ) + lst.get( j ) == K )
                return true;

    return false;
}
```

a. What is the running time of `containsSum` when the list is an `ArrayList`?
b. What is the running time of `containsSum` when the list is a `LinkedList`?
c. Suppose it takes 2 seconds to run `containsSum` on a 1,000-item `ArrayList`. How long will it take to run `containsSum` on a 3,000-item `ArrayList`? You may assume `containsSum` returns false in both cases.

15 Consider the following implementation of the `clear` method (which empties any collection).

```
public abstract class AbstractCollection<AnyType>
                      implements Collection<AnyType>
{
    public void clear( )
    {
        Iterator<AnyType> itr = this.iterator( );

        while( itr.hasNext( ) )
        {
            itr.next( );
            itr.remove( );
        }
    }
       ...
}
```

a. Suppose `LinkedList` extends `AbstractCollection` and does not override `clear`. What is the running time of `clear`?
b. Suppose `ArrayList` extends `AbstractCollection` and does not override `clear`. What is the running time of `clear`?
c. Suppose it takes 4 seconds to run `clear` on a 100,000-item `ArrayList`. How long will it take to run `clear` on a 500,000-item `ArrayList`?
d. As clearly as possible, describe the behavior of this alternate implementation of `clear`:

```
public void clear( )
{
    for( AnyType item : this )
        this.remove( item );
}
```

16 Static method `removeHalf` removes the first half of a `List` (if there are an odd number of items, then slightly less than one-half of the list is removed). One possible implementation of `removeHalf` is shown below:

```
public static void removeHalf( List<?> lst )
{
    int size = lst.size( );

    for( int i = 0; i < size / 2; i++ )
        lst.remove( 0 );
}
```

a. Why can't we use `lst.size( )/2` as the test in the for loop?

b. What is the Big-Oh running time if `lst` is an `ArrayList`.
c. What is the Big-Oh running time if `lst` is a `LinkedList`.
d. Suppose we have two computers, Machine A and Machine B. Machine B is twice as fast as Machine A. How would the running time for `removeHalf` on Machine B compare to Machine A if Machine B is given an `ArrayList` that is twice as large as the `ArrayList` given to Machine A?
e. Does the one line implementation:

```
public static void removeHalf( List<?> lst )
{
    lst.subList( 0, lst.size( ) / 2 ).clear( );
}
```

work, and if so, what is the Big-Oh running time for both an `ArrayList` and a `LinkedList`?

17 Static method `removeEveryOtherItem` removes items in even positions (0, 2, 4, etc.) from a `List`. One possible implementation of `removeEveryOtherItem` is shown below:

```
public static void removeEveryOtherItem( List<?> lst )
{
    for( int i = 0; i < lst.size( ); i++ )
        lst.remove( i );
}
```

a. What is the Big-Oh running time if `lst` is an `ArrayList`.
b. What is the Big-Oh running time if `lst` is a `LinkedList`.
c. Suppose we have two computers, Machine A and Machine B. Machine B is twice as fast as Machine A. Machine A takes 1 sec. on a 100,000 item list. How large a list can Machine B process in 1 second?
d. Rewrite `removeEveryOtherItem`, using an iterator, so that it is efficient for linked lists and does not use any extra space besides the iterator.

18 Consider the following implementation of the `removeAll` method (which removes all occurrences of any item in the collection passed as a parameter from this collection).

```
public abstract class AbstractCollection<AnyType>
                      implements Collection<AnyType>
{
    public boolean removeAll ( Collection<? extends AnyType> c )
    {
        Iterator<AnyType> itr = this.iterator( );
        boolean wasChanged = false;

        while( itr.hasNext( ) )
        {
            if( c.contains( itr.next( ) ) )
            {
                itr.remove( );
                wasChanged = true;
            }
        }
        return wasChanged;
    }
        ...
}
```

a. Suppose `LinkedList` extends `AbtractCollection` and does not override `removeAll`. What is the running time of `removeAll` when `c` is a `List`?
b. Suppose `LinkedList` extends `AbstractCollection` and does not override `removeAll`. What is the running time of `removeAll` when `c` is a `TreeSet`?
c. Suppose `ArrayList` extends `AbstractCollection` and does not override `removeAll`. What is the running time of `removeAll` when `c` is a `List`?
d. Suppose `ArrayList` extends `AbstractCollection` and does not override `removeAll`. What is the running time of `removeAll` when `c` is a `TreeSet`?
e. What is the result of calling `c.removeAll(c)` using the implementation above?
f. Explain how to add code so that a call such as `c.removeAll(c)` clears the collection.

19 Write a test program to see which of the following calls successfully clears a java `LinkedList`.

```
c.removeAll( c );
c.removeAll( c.subList ( 0, c.size( ) );
```

20 The `RandomAccess` interface contains no methods but is intended to serve as a marker: a `List` class implements the interface only if its `get` and `set` methods are very efficient. Accordingly, `ArrayList` implements the `RandomAccess` interface. Implement static method `removeEveryOtherItem`, described in Exercise 17. If `list` implements `RandomAccess` (use an `instanceof` test), then use `get` and `set` to reposition items at the front half of the list. Otherwise, use an iterator that is efficient for linked lists.

21 Write, in as few lines as possible, code that removes all entries in a `Map` whose values are `null`.

22 The `listIterator` method `set` allows the caller to change the value of the last item that was seen. Implement the `toUpper` method (that makes the entire list upper case) shown below using a `listIterator`:

```
public static void toUpper( List<String> theList )
```

23 Method `changeList` replaces each String in the list by both its lower case and upper case equivalents. Thus if the original list contained [`Hello`,`NewYork`], then new list will contain [`hello`, `HELLO`, `newyork`, `NEWYORK`]. Use the `listIterator` `add` and `remove` methods to write an efficient implementation for linked lists of method `changeList`:

```
public static void changeList( LinkedList<String> theList )
```

PROGRAMMING PROJECTS

24 A queue can be implemented by using an array and maintaining the current size. The queue elements are stored in consecutive array positions, with the front item always in position 0. Note that this is not the most efficient method. Do the following:

a. Describe the algorithms for `getFront`, `enqueue`, and `dequeue`.
b. What is the Big-Oh running time for each of `getFront`, `enqueue`, and `dequeue` using these algorithms?
c. Write an implementation that uses these algorithms using the protocol in Figure 28.

25 The operations that are supported by the `SortedSet` can also be implemented by using an array and maintaining the current size. The array elements are stored in sorted order in consecutive array positions. Thus `contains` can be implemented by a binary search. Do the following:

a. Describe the algorithms for `add` and `remove`.
b. What is the running time for these algorithms?