Java 1996 AP Computer Science Question 3

Assume that binary trees are implemented using the standard TreeNode class:

```
public class TreeNode
{
    public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)
    {
        // not shown
    }

    public Object getValue()   { // not shown }
    public TreeNode getLeft()  { // not shown }
    public TreeNode getRight() { // not shown }

    public void setValue(Object theNewValue)   { // not shown }
    public void setLeft(TreeNode theNewLeft)   { // not shown}
    public void setRight(TreeNode theNewRight) { // not shown }
};
```

**Part A**

Write the method valsLess whose header is given below. ValsLess returns true if its parameter t is null or if all values stored in the binary tree pointed to by t are less than obj; otherwise valsLess returns false. Assume all values stored in t implement the Comparable interface.

Complete ValsLess below the following header.

```
// precondition: all values stored in t implement Comparable
// postcondition: returns true if t is null or if
// all values stored in tree represented
// by t are less than obj; otherwise
// returns false

public boolean valsLess(TreeNode t, Comparable obj)
```

**Part B**

Recall that a binary tree $T$ that contains no duplicate values is a search tree if and only if

1. $T$ is empty or
2. all of the following are true
   - The value stored at the root of $T$ is greater than all of the values stored in $T$'s left subtree.
   - The value stored at the root of $T$ is less than all of the values stored in $T$'s right subtree.
   - $T$'s left subtree is a binary search tree that contains no duplicate values.
   - $T$'s right subtree is a binary search tree that contains no duplicate values.

Write function isBST whose header is given below. isBST should return true if the binary tree represented by its parameter $t$ is a binary search tree that contains no duplicate values; otherwise isBST should return false.

In writing isBST you may include calls to function valsLess from part (A). Assume that valsLess works as specified, regardless of what you wrote in part (A). You may also include calls to function valsGreater whose specification is given below. (You do not need to write the body of valsGreater.)

```
// precondition: all values stored in t implement Comparable
```
// postcondition: returns true if t is null or if
//                all values stored in tree represented
//                by t are greater than obj; otherwise
//                returns false

public boolean valsGreater(TreeNode t, Comparable obj)

Complete function isBST below the following header.

// postcondition: returns true if t represents a binary search
//                tree containing no duplicate values;
//                otherwise, returns false.

public boolean isBST(TreeNode t)

answer

Owen L. Astrachan
Last modified: Sat Jul 5 10:45:01 EDT 2003
Assume that binary trees are implemented using the following declarations.

```java
public class TreeNode {
    public TreeNode(Off Object initValue, TreeNode initLeft, TreeNode initRight) {
        // not shown
    }

    public Object getValue() { // not shown }
    public TreeNode getLeft() { // not shown }
    public TreeNode getRight() { // not shown }

    public void setValue(Object theNewValue) { // not shown }
    public void setLeft(TreeNode theNewLeft) {  // not shown}
    public void setRight(TreeNode theNewRight) {  // not shown}
}
```

You may find the following method useful in writing solutions for this problem.

```java
// returns number of nodes in t
public int nodeCount(TreeNode t) {
    if (t == null) return 0;
    return 1 + nodeCount(t.getLeft()) + nodeCount(t.getRight());
}
```

**Part A**

Write method `setSize`, whose header is given below. `setSize` should replace the value stored in each node of the tree with an `Integer` object whose value is the number of nodes in that node's subtree (including the node itself). The number of nodes in a tree is equal to the number of nodes in its left subtree plus the number of nodes in its right subtree plus one.

For example, the following picture shows the result of the call `setSize(root)`. The values labelled `data` could represent the values stored before calling `setSize`, the values labelled `size` are the values stored in each node after `setSize` has completed.
Complete function `setSize` below.

```java
public class AB4 {
    // precondition: t != null
    // postcondition: The value of every node in the tree t
    //                has been set to an Integer whose value is the
    //                the size of that node's subtree

    public void setSize(TreeNode t) {
    }
}
```

**answer**

**Part B**

Assume that tree \( t \) is a binary search tree ordered by the node's values, and that there are no duplicate values. The \( k \)th value in a binary search tree is the \( k \)th smallest value in the tree. For example, the tree shown in part (a) includes the data values 12, 25, 30, 37, 50, 62, 75, 88.

- The 1st value is 12.
- The 4th value is 37.
- The 7th value is 75.
- The 8th value is 88.

Write method `findKth`, whose header is given below. `findKth` returns the \( k \)th value in a binary search tree. Assume that the `size` fields in all nodes of the tree have been correctly initialized. One way to determine the location of the \( k \)th value is as follows:

Consider the size of the left subtree of the current node
- If \( k \) is equal to the size of the left subtree + 1, the \( k \)th value is in the current node.
- If \( k \) is less than the size of the left subtree + 1, the \( k \)th value is in the left subtree.
- Otherwise, the \( k \)th value is in the right subtree.

Complete function `kthValue` below.

```java
Object kindKth(TreeNode * t, int k) {
    // precondition: t is not NULL, the size fields of all nodes in t
    //                are correctly initialized.
    // 1 <= k <= t->size
    // postcondition: returns the kth value in t
```
Consider the problem of encoding words as a string of 0's and 1's using a codetree. A codetree is a binary tree containing letters (strings of length one) in its leaves. The encoding of a letter is represented by the root-to-leaf path for the letter. The same codetree is used for encoding and decoding.

The following properties hold for every codetree.

1. Every node is either a leaf or has exactly 2 children.
2. Letters appear only at the leaves of the codetree.
3. There are at least 2 letters in the codetree.
4. Each letter appears at most once in the codetree; thus there is a unique root-to-leaf path for each letter.

For example, consider the following codetree, C.

```
C
/\  
S  u
/ \  /  
/   \ /   
/     N  
```

The code for each letter is a string formed by appending a "0" when taking a left branch and a "1" for a right branch when traversing the root-to-leaf path. In the codetree above, the code for "u" is "010" (left, right, left), the code for "s" is "00", and the code for "n" is "10". A word is encoded by appending the codes for letters in the word together. For example, the code for "sun" is "0001010", which is formed by appending the codes for "s", "u", and "n".

Consider the following declarations for a class that represents a codetree.

```java
public class CodeTree
{
    /**
     * construct a codetree
     */
    public CodeTree()
    {
        // not shown
    }

    /**
     * Returns a decoded word for code.
     * @precondition code valid string of 0's and 1's
     * @return the decoded word for code
     */
    public String bitsToWord(String code)
    {
        // not shown
    }
}
```
/**
 * Returns the code for word
 * @precondition each character of word is in a leaf of codetree
 * @return the code for word
 */

public String wordToBits(String word)
{
    // not shown
}

/**
 * Returns the code for s in codetree, returns "" if s not in tree
 * @precondition pathSoFar is path of 0's and 1's from myRoot to t
 * @return the path from myRoot to leaf containing s,"" if no path
 */

private String wordToBitsHelper(String s, TreeNode t, String pathSoFar)
{
    // not shown
}

private TreeNode myRoot;  // root of codetree, nodes store Strings

---

**Part A**

You will write the `CodeTree` method `bitsToWord`, which is described as follows. Method `bitsToWord` is given a coded word (a string of 0's and 1's) and returns the decoded word.

Each character (substring of length one) of code represents a branch in the codetree where "0" represents a left branch and "1" represents a right branch. To decode the word represented by `code`, begin at the root and follow a branch for each "0" or "1" character in `code`. When a leaf is reached, one letter in the decoded word has been found. The decoding process begins again at the root of the codetree with the next "0" or "1" in `code`.

![CodeTree Diagram]

For example, using the `CodeTree C` as shown, if `code` is "1110", the call `C.bitsToWord(code)` returns the word "in". This result is obtained as follows. The path starts at the root and goes right for the first "1" character in `code`, right again for the second "1", and a leaf is reached, meaning the decoded word begins with "i" (note that each leaf `TreeNode` stores a String). Starting back at the root of the codetree and with the next "1" in `code`, the path goes right for the "1" and left for the "0", reaching the leaf containing "n". The decoded word is now "in", and since all the characters in `code` have been processed, "in" is returned. Similarly, `C.bitsToWord("000101010011")` returns the word "sunny".

Complete method `bitsToWord` below. Assume that instance field `myRoot` has been initialized to the root of a codetree when `bitsToWord` is called.

/**
public String bitsToWord(String code) {

}
Using CodeTree C as shown, \texttt{wordToBitsHelper("y",myRoot,"")} would return the string "011" and \texttt{wordToBitsHelper("n",myRoot,"")} would return "01". Note that \texttt{wordToBitsHelper("x",myRoot,"")} would return "" since "x" is not in the codetree.

Complete method \texttt{wordToBitsHelper} below.

```java
/**
 * Returns the code for \texttt{s} in codetree, returns "" if \texttt{s} not in tree
 * @precondition \texttt{pathSoFar} is path of 0's and 1's from \texttt{myRoot} to \texttt{t}
 * @return the path from \texttt{myRoot} to leaf containing \texttt{s} ,"" if no path
 */

private String wordToBitsHelper(String s, TreeNode t, String pathSoFar) {

}
```

\textit{Owen L. Astrachan}

Last modified: Thu Jul 11 13:57:35 EDT 2002
Periodically, a company processes the retirement of some of its employees. In this question, you will write methods to help the company determine whether an employee is eligible to retire and to process the retirement of employees who wish to retire. You will also analyze the runtime performance of one of the methods that you write.

The Java interface **Employee** is implemented by objects that represent employees. The interface is declared as follows.

```java
public interface Employee {
    public int getAge();        // returns age of employee
    public int getYearsOnJob(); // returns number of years on job
    public double getSalary();  // returns salary in dollars
    public int getID();         // returns unique employee ID number
}
```

The **Company** class is declared as follows.

```java
import java.util.ArrayList;

public class Company {
    private final static int RETIRE_AGE   = 65;
    private final static int RETIRE_YEARS = 30;
    private final static double RETIRE_SALARY = 10000.0;

    private ArrayList myEmployees;  // list of employees
    private double myTotalSalary;   // total salary of all employees

    // ... constructor and other methods not shown

    // precondition: claimants is sorted in ascending order by
    // employee ID, contains no duplicates, and each
    // element of claimants is in myEmployees

    private boolean employeeIsEligible(Employee emp) {
        // you will write this function
    }
}
```
// postcondition: all retirement-eligible employees have been
// removed from myEmployees; myEmployees remains
// sorted by employee ID; myTotalSalary has been
// updated to maintain invariant that it represents
// total of all employee salaries

public void processRetirements(Employee[] claimants) {
    // you will write this function
}

Two class invariants must be maintained by Company objects:

- The instance variable myEmployees is sorted by employee ID.
- The instance variable myTotalSalary is the total of all employee salaries.

The Company constructor will establish these invariants as true initially. Each Company method must ensure that the invariants remain true after the method's execution.

Part A

An employee is eligible for retirement if (s)he meets at least two of the following requirements.

1. The employee is at least RETIRE_AGE years old.
2. The employee has worked for at least RETIRE_YEARS years.
3. The employee's salary is at least RETIRE_SALARY.

Write the private Company method employeeIsEligible, which is described as follows. Method employeeIsEligible returns a boolean value that indicates whether the employee represented by parameter emp is eligible for retirement, using the rules above.

Complete method employeeIsEligible below.

// postcondition: returns true if emp is eligible to retire;
// otherwise, returns false

private boolean employeeIsEligible(Employee emp) {
}

Part B

( Assume all import statements you need are made. You do not need to write import statements).

Write the Company method processRetirements, which is described as follows. Method
processRetirements has one parameter, claimants representing all employees that wish to retire. Assume
claimants is sorted in ascending order by ID number, contains no duplicates, and that all elements in
claimants are also in private instance variable myEmployees. Method processRetirements removes from
ArrayList myEmployees only those employees listed in claimants that are eligible for retirement and
maintains the two class invariants described above: the ArrayList is maintained in order by employee ID and
myTotalSalary is the total of all salaries of the remaining employees.

In writing processRetirements, you may call method employeeIsEligible, specified in part (a). Assume
that employeeIsEligible works as specified, regardless of what you wrote in part (a).

Complete method processRetirements below.

```java
// precondition: claimants is sorted in ascending order by
// employee ID, contains no duplicates, and each
// element of claimants is in myEmployees
// postcondition: all retirement-eligible employees in claimants have been
// removed from myEmployees; myEmployees remains
// sorted by employee ID; myTotalSalary has been
// updated to maintain invariant that it represents
// total of all employee salaries

public void processRetirements(Employee[] claimants) {

}
```

Part C

Assume that N is the number of employees in the company. Give the best Big-Oh expression (in terms of N)
for the worst-case running time for your implementation of the function processRetirements. Justify your
answer with reference to the code you wrote in part (b). You will NOT receive full credit if you do not
provide a justification.

Owen L. Astrachan
Last modified: Mon May 19 14:03:52 EDT 2003
Periodically, a company processes the retirement of some of its employees. In this question, you will write methods to help the company determine whether an employee is eligible to retire and to process the retirement of employees who wish to retire. You will also analyze the runtime performance of one of the methods that you write.

The Java interface Employee is implemented by objects that represent employees. The interface is declared as follows.

```java
public interface Employee extends Comparable {
    public int getAge();        // returns age of employee
    public int getYearsOnJob(); // returns number of years on job
    public double getSalary();  // returns salary in dollars
    public int getID();         // returns unique employee ID number
}
```

The Company class is declared as follows.

```java
import java.util.ArrayList;

public class Company {
    // minimum age, years on job, and salary needed to retire
    private final static int RETIRE_AGE   = 65;
    private final static int RETIRE_YEARS = 30;
    private final static double RETIRE_SALARY = 10000.0;

    private ArrayList myEmployees; // list of employees
    private double myTotalSalary;   // total salary of all employees

    // ... constructor and other methods not shown

    // postcondition: returns true if emp is eligible to retire;
    //                 otherwise, returns false
    private boolean employeeIsEligible(Employee emp) {
        // you will write this function
    }

    // precondition: claimants is sorted in ascending order,
    //                contains no duplicates, and each
    //                element of claimants is in myEmployees
```
Two class invariants must be maintained by Company objects:

- The instance variable myEmployees is sorted (e.g., by employee ID), note that interface Employee extends the Comparable interface.
- The instance variable myTotalSalary is the total of all employee salaries.

The Company constructor will establish these invariants as true initially. Each Company method must ensure that the invariants remain true after the method's execution.

Part A

An employee is eligible for retirement if (s)he meets at least two of the following requirements.

1. The employee is at least RETIRE_AGE years old.
2. The employee has worked for at least RETIRE_YEARS years.
3. The employee's salary is at least RETIRE_SALARY.

Write the private Company method employeeIsEligible, which is described as follows. Method employeeIsEligible returns a boolean value that indicates whether the employee represented by parameter emp is eligible for retirement, using the rules above.

Complete method employeeIsEligible below.

```java
// postcondition: returns true if emp is eligible to retire; otherwise, returns false
private boolean employeeIsEligible(Employee emp) {
    // your implementation here
}
```
**Part B**

*(Assume all import statements you need are made. You do not need to write import statements)*

Write the `Company` method `processRetirements`, which is described as follows. Method `processRetirements` has one parameter, `claimants` representing all employees that wish to retire. Assume `claimants` is sorted in ascending order, contains no duplicates, and that all elements in `claimants` are also in private instance variable `myEmployees`. Method `processRetirements` removes from ArrayList `myEmployees` only those employees listed in `claimants` that are eligible for retirement and maintains the two class invariants described above: the ArrayList is maintained in ascending order and `myTotalSalary` is the total of all salaries of the remaining employees.

Assume that the class used to implement the `Employee` interface has an overridden method `equals` consistent with its method `compareTo`.

In writing `processRetirements`, you may call method `employeeIsEligible`, specified in part (a). Assume that `employeeIsEligible` works as specified, regardless of what you wrote in part (a).

Complete method `processRetirements` below.

```java
// precondition: claimants is sorted in ascending order,
// contains no duplicates, and each
// element of claimants is in myEmployees
// postcondition: all retirement-eligible employees in claimants have been
// removed from myEmployees; myEmployees remains
// sorted in ascending order; myTotalSalary has been
// updated to maintain invariant that it represents
// total of all employee salaries
public void processRetirements(Employee[] claimants)
{
}
```

**Part C**

Assume that $N$ is the number of employees in the company. Give the best Big-Oh expression (in terms of $N$) for the worst-case running time for your implementation of the function `processRetirements`. Justify your answer with reference to the code you wrote in part (b). You will NOT receive full credit if you do not provide a justification.

---

*Owen L. Astrachan*

Last modified: Mon May 19 11:23:07 EDT 2003
Consider the problem of representing a filing cabinet with drawers of student records. A filing cabinet is implemented using a linked list of drawers. Each drawer is implemented using a linked list of student records. All student records in a drawer have a student ID less than or equal to the drawer's maximum student ID, and student records are stored in a drawer in ascending order by student ID.

The diagram below illustrates the structure of a filing cabinet as implemented by the class `FilingCabinet`. The data member `myDrawerList` is an instance of `java.util.LinkedList` that stores `Drawer` objects in ascending order by maximum ID.

The class `Student` is declared as follows.
public class Student {
    // constructor and data members not shown
    // returns id of this student
    public int getID() {
        // not shown
    }
    // returns name of this student
    public String getName() {
        // not shown
    }
    // precondition: o is an instance of student
    // postcondition: returns true if o equals this student
    // otherwise returns false
    public boolean equals(Object o) {
        // not shown
    }
}

The class **Drawer** is declared as follows.

```java
public class Drawer {
    private int myMaxID;            // maximum ID in this drawer
    private LinkedList myStudents;  // all students in this drawer
    // constructor and some methods not shown
    // returns maximal ID for this drawer
    public int getMaxID() {
        return myMaxID;
    }
    // add s to this drawer so students are in ascending order by ID
    public void addStudent(Student s) {
        // you will write this
    }
    // return an iterator for the students in this drawer
    public Iterator iterator() {
        return myStudents.iterator();
    }
}
```

The class **FilingCabinet** is declared as follows.
public class FilingCabinet
{
    private LinkedList myDrawerList;

    // precondition: this filing cabinet has at least one Drawer;
    // studentID is less than or equal to maximum ID
    // of last Drawer
    // postcondition: returns the first Drawer d such that
    // d.getMaxID() >= studentID

    public Drawer findDrawer(int studentID)
    {
        // you will write this
    }

    // precondition: student.getID() <= maximum ID of last Drawer
    // postcondition: student added to proper Drawer

    public void addStudent(Student student)
    {
        Drawer d = findDrawer(student.getID());
        d.addStudent(student);
    }

    // precondition: student.getID() is less than or equal to
    // maximum ID of last Drawer
    // postcondition: if there is a Student s in this filing cabinet
    // equal to student, then s is removed from the
    // drawer in which it is located; otherwise this
    // FilingCabinet is unchanged

    public void removeStudent(Student student)
    {
        // you will write this
    }
}

Part A

Write FilingCabinet method findDrawer, which is described as follows. Method findDrawer returns the
Drawer object in which studentID would be found. Method findDrawer returns the first Drawer in the list
myDrawerList for which studentID is less than or equal to the maximum student ID number that can be
filed in the drawer.

Complete findDrawer below.

    // precondition: this filing cabinet has at least one Drawer;
    // studentID is less than or equal to maximum ID
    // of last Drawer
    // postcondition: returns the first Drawer d such that
    // d.getMaxID() >= studentID

    public Drawer findDrawer(int studentID)
    {

    }
**Part B**

Write the `FilingCabinet` method `removeStudent`, which is described as follows. Method `removeStudent` removes the `Student` object equal to `student` from the `FilingCabinet` if there is such an object. If there is no such object then the `FilingCabinet` is unchanged.

In writing `removeStudent`, you may call `findDrawer` specified in part (a). Assume that `findDrawer` works as specified, regardless of what you wrote in part (a).

Complete method `removeStudent` below.

```java
// precondition: student.getID() is less than or equal to
//                maximum ID of last Drawer
// postcondition: if there is a Student s in this filing cabinet
//                equal to student, then s is removed from the
//                drawer in which it is located; otherwise this
//                FilingCabinet is unchanged

public void removeStudent(Student student) {

}
```

**Part C**

Write the `Drawer` method `addStudent` which is described as follows. Method `addStudent` inserts the `Student` object `s` into `LinkedList` object `myStudents` so that the linked list is maintained in increasing order by student ID.

Assume that the `Drawer` constructor initializes `myStudents` to be a `LinkedList` containing no objects.

You may find the following algorithm helpful in implementing `Drawer` method `addStudent`.

- If the linked list `myStudents` is empty, add the new student anywhere in the linked list.
- If the new student's ID is less than the ID of the first student in `myStudents`, then add the new student at the beginning of the linked list.
- Similarly, if the new student's ID is greater than the ID of the last student in `myStudents`, then add the new student to the end of the linked list.
- (Otherwise there at least two objects in the linked list.) Use two `ListIterator` objects for `myStudents` and the `ListIterator` method `add`. The call to `add` should occur between the calls of `next` on the `ListIterator` objects.

Complete `Drawer` method `addStudent` below.
public class Drawer
{
    private LinkedList myStudents; // list of students in this drawer

    // add s to this drawer so students are in ascending order by ID
    public void addStudent(Student s)
    {
    
    }
}

Owen L. Astrachan
Last modified: Wed May 14 17:11:46 EDT 2003
Consider the problem of representing a filing cabinet with drawers of student records. A filing cabinet is implemented using a linked list of drawers. Each drawer is implemented using a linked list of student records. All student records in a drawer have a student ID less than or equal to the drawer's maximum student ID, and student records are stored in a drawer in ascending order by student ID.

The diagram below illustrates the structure of a filing cabinet as implemented by the class `FilingCabinet`. The data member `myDrawerList` is an instance of `ListNode` that implements a linked list of `Drawer` objects in ascending order by maximum ID.

```
myDrawerList

myMaxID = 250
  myStudents

myMaxID = 428
  myStudents

myMaxID = 637
  myStudents

... ...

myMaxID = 999
  myStudents

myID = 123

myID = 178

myID = 250

myID = 542

myID = 834

... ...

myID = 987
```

The class `Student` is declared as follows.
public class Student
{
    // constructor and data members not shown
    // returns id of this student
    public int getID()
    {
        // not shown
    }

    // returns name of this student
    public String getName()
    {
        // not shown
    }

    // precondition: o is an instance of student
    // postcondition: returns true if o equals this student
    //                otherwise returns false
    public boolean equals(Object o)
    {
        // not shown
    }
}

The class **Drawer** is declared as follows.

```java
public class Drawer
{
    private int myMaxID;            // maximum ID in this drawer
    private ListNode myStudents;    // all students in this drawer

    // constructor and some methods not shown
    // returns maximal ID for this drawer
    public int getMaxID()
    {
        return myMaxID;
    }

    // remove Student equal to s from this drawer
    public void removeStudent(Student s)
    {
        // you will write this
    }

    // return first node in this drawer's linked list
    public ListNode getFirst()
    {
        return myStudents;
    }
}
```

The class **FilingCabinet** is declared as follows.
public class FilingCabinet
{
    private ListNode myDrawerList;

    // precondition: this filing cabinet has at least one Drawer;
    // studentID is less than or equal to maximum ID
    // of last Drawer
    // postcondition: returns the first Drawer d such that
    // d.getMaxID() >= studentID

    public Drawer findDrawer(int studentID)
    {
        // you will write this
    }

    // precondition: student.getID() is less than or equal to
    // maximum ID of last Drawer
    // postcondition: if there is a Student s in this filing cabinet
    // equal to student, then s is removed from the
    // drawer in which it is located; otherwise this
    // FilingCabinet is unchanged

    public void removeStudent(Student student)
    {
        Drawer d = findDrawer(student.getID());
        d.removeStudent(student);
    }
}

Part A

Write FilingCabinet method findDrawer, which is described as follows. Method findDrawer returns the object in which studentID would be found. Method findDrawer returns the first Drawer in the list myDrawerList for which studentID is less than or equal to the maximum student ID number that can be filed in the drawer.

Complete findDrawer below.

    // precondition: this filing cabinet has at least one Drawer;
    // studentID is less than or equal to maximum ID
    // of last Drawer
    // postcondition: returns the first Drawer d such that
    // d.getMaxID() >= studentID

    public Drawer findDrawer(int studentID)
    {
    }

Part B

Write the Drawer method removeStudent, which is described as follows. Method removeStudent removes the Student object equal to student from the Drawer if there is such an object. If there is no such object then the Drawer is unchanged.
Complete method `removeStudent` below.

```java
// precondition: student.getID() is less than or equal to
//               maximum ID of this Drawer
// postcondition: if there is a Student s in this drawer
//               equal to student, then s is removed from the
//               drawer; otherwise this drawer is unchanged

public void removeStudent(Student student) {
}
```

`Owen L. Astrachan`

Last modified: Wed May 14 10:15:37 EDT 2003