Henry Mound

AP® Computer Science A
2012 Free-Response Questions

About the College Board

The College Board is a mission-driven not-for-profit organization that connects students to college success and opportunity. Founded in 1900, the College Board was created to expand access to higher education. Today, the membership association is made up of more than 5,900 of the world's leading educational institutions and is dedicated to promoting excellence and equity in education. Each year, the College Board helps more than seven million students prepare for a successful transition to college through programs and services in college readiness and college success — including the SAT® and the Advanced Placement Program®. The organization also serves the education community through research and advocacy on behalf of students, educators, and schools.

Visit the College Board on the Web: www.collegeboard.org.
AP Central is the official online home for the AP Program: apcentral.collegeboard.org.

COMPUTER SCIENCE A
SECTION II

Time—1 hour and 45 minutes
Number of questions—4
Percent of total score—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the appendices have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. A mountain climbing club maintains a record of the climbs that its members have made. Information about a climb includes the name of the mountain peak and the amount of time it took to reach the top. The information is contained in the `ClimbInfo` class as declared below.

```
public class ClimbInfo
{
  /** Creates a ClimbInfo object with name peakName and time climbTime.
   *  @param peakName the name of the mountain peak
   *  @param climbTime the number of minutes taken to complete the climb
   */
  public ClimbInfo(String peakName, int climbTime)
  {  /* implementation not shown */  }

  /** @return the name of the mountain peak
   */
  public String getName()
  {  /* implementation not shown */  }

  /** @return the number of minutes taken to complete the climb
   */
  public int getTime()
  {  /* implementation not shown */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

The `ClimbingClub` class maintains a list of the climbs made by members of the club. The declaration of the `ClimbingClub` class is shown below. You will write two different implementations of the `addClimb` method. You will also answer two questions about an implementation of the `distinctPeakNames` method.

```
public class ClimbingClub
{
  /** The list of climbs completed by members of the club.
   *  Guaranteed not to be null. Contains only non-null references.
   */
  private List<ClimbInfo> climbList;

  /** Creates a new ClimbingClub object. */
  public ClimbingClub()
  {  climbList = new ArrayList<ClimbInfo>();  }

  /** Adds a new climb with name peakName and time climbTime to the list of climbs.
   *  @param peakName the name of the mountain peak climbed
   *  @param climbTime the number of minutes taken to complete the climb
   */
  public void addClimb(String peakName, int climbTime)
  {  /* to be implemented in part (a) with ClimbInfo objects in the order they were added  */
     /* to be implemented in part (b) with ClimbInfo objects in alphabetical order by name */
  }

  /** @return the number of distinct names in the list of climbs */
  public int distinctPeakNames()
  {  /* implementation shown in part (c) */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

Part (a) begins on page 4.

(a) Write an implementation of the `ClimbingClub` method `addClimb` that stores the `ClimbInfo` objects in the order they were added. This implementation of `addClimb` should create a new `ClimbInfo` object with the given name and time. It appends a reference to that object to the end of `climbList`. For example, consider the following code segment.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
```

When the code segment has completed executing, the instance variable `climbList` would contain the following entries.

Peak Name	"Monadnock"	"Whiteface"	"Algonquin"	"Monadnock"
Climb Time	274	301	225	344

Information repeated from the beginning of the question

```
public class ClimbInfo

public ClimbInfo(String peakName, int climbTime)
public String getName()
public int getTime()


public class ClimbingClub

private List<ClimbInfo> climbList
public void addClimb(String peakName, int climbTime)
public int distinctPeakNames()
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `addClimb` below.

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Postcondition: The new entry is at the end of climbList;
 *                 The order of the remaining entries is unchanged.
 */
public void addClimb(String peakName, int climbTime) {
    this.climbList.add(new ClimbInfo(peakName, climbTime));
}
```

Part (b) begins on page 6.

(b) Write an implementation of the `ClimbingClub` method `addClimb` that stores the elements of `climbList` in alphabetical order by name (as determined by the `compareTo` method of the `String` class). This implementation of `addClimb` should create a new `ClimbInfo` object with the given name and time and then insert the object into the appropriate position in `climbList`. Entries that have the same name will be grouped together and can appear in any order within the group. For example, consider the following code segment.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
```

When the code segment has completed execution, the instance variable `climbList` would contain the following entries in either of the orders shown below.

Peak Name	"Algonquin"	"Monadnock"	"Monadnock"	"Whiteface"
Climb Time	225	344	274	301

OR

Peak Name	"Algonquin"	"Monadnock"	"Monadnock"	"Whiteface"
Climb Time	225	274	344	301

You may assume that `climbList` is in alphabetical order by name when the method is called. When the method has completed execution, `climbList` should still be in alphabetical order by name.

Information repeated from the beginning of the question

```
public class ClimbInfo

public ClimbInfo(String peakName, int climbTime)
public String getName()
public int getTime()


public class ClimbingClub

private List<ClimbInfo> climbList
public void addClimb(String peakName, int climbTime)
public int distinctPeakNames()
```

Complete method `addClimb` below.

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  Alphabetical order is determined by the compareTo method of the String class.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Precondition: entries in climbList are in alphabetical order by name.
 *  Postcondition: entries in climbList are in alphabetical order by name.
 */
public void addClimb(String peakName, int climbTime)
{
    for (int x = 0; x < this.climbList.size(); x++)
    {
        if (!(peakName.compareTo(this.climbList.get(x).getName()) > 0))
        {
            this.climbList.add(x-1, new ClimbInfo(peakName, climbTime));
            break;
        }
    }
}
```

Part (c) begins on page 8.

(c) The `ClimbingClub` method `distinctPeakNames` is intended to return the number of different names in `climbList`. For example, after the following code segment has completed execution, the value of the variable `numNames` would be 3.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
int numNames = hikerClub.distinctPeakNames();
```

Consider the following implementation of method `distinctPeakNames`.

```
/** @return the number of distinct names in the list of climbs */
public int distinctPeakNames()
{
  if (climbList.size() == 0)
  {
    return 0;
  }

  ClimbInfo currInfo = climbList.get(0);
  String prevName = currInfo.getName();
  String currName = null;
  int numNames = 1;

  for (int k = 1; k < climbList.size(); k++)
  {
    currInfo = climbList.get(k);
    currName = currInfo.getName();
    if (prevName.compareTo(currName) != 0)
    {
       numNames++;
       prevName = currName;
    }
  }
  return numNames;
}
```

Assume that `addClimb` works as specified, regardless of what you wrote in parts (a) and (b).

(i) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in the order they were added as described in part (a)?

Circle one of the answers below.

YES NO

(ii) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in alphabetical order by name as described in part (b)?

Circle one of the answers below.

YES NO

2. This question involves reasoning about the GridWorld case study. Reference materials are provided in the appendices.

A retro bug behaves like a regular bug. It also has the ability to revert to its previous location and direction. When a retro bug acts, it maintains information about its location and direction at the beginning of the act. The retro bug has a `restore` method that restores it to the location (if possible) and direction it faced at the beginning of its previous act. A retro bug only maintains information about its most recent act; therefore, multiple calls to `restore` that occur before its next act will use the same information. The `restore` method has no effect if it is called before a retro bug's first act.

The `restore` method takes no parameters and does not return a value. The `restore` method has the following functionality.

- If the previous location of the retro bug is either unoccupied or contains a flower, the `restore` method places the retro bug in that previous location. The presence of any other type of actor in that location will prevent the retro bug from being placed in that location.
- The `restore` method always ends with the retro bug facing in the same direction that it had been facing at the beginning of its most recent act.

The following examples illustrate the behavior of the `restore` method.

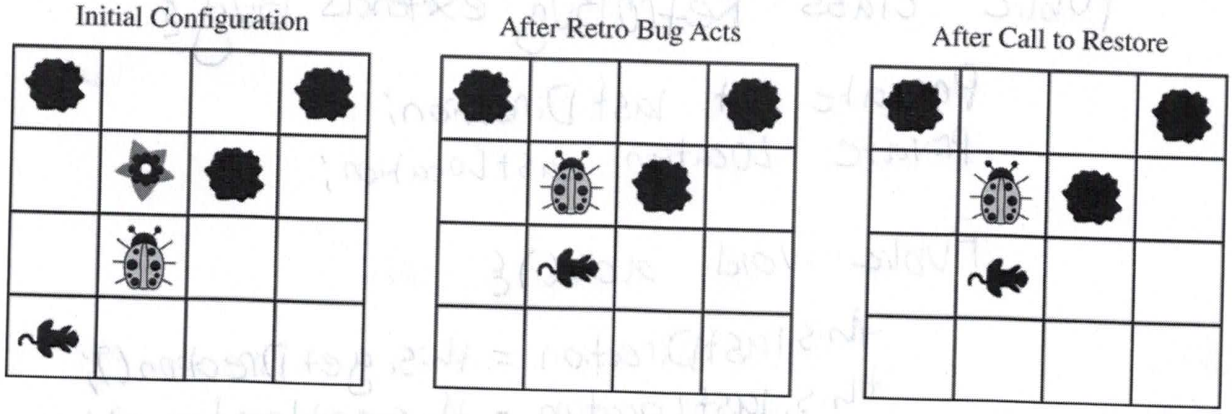
Example 1

The retro bug acts once and later calls `restore`. Note that the flower that was originally in front of the retro bug is not replaced as a result of the call to `restore`. The retro bug is returned to its previous direction, which, in this case, is the same as the current direction.

Question 2 continues on the next page.

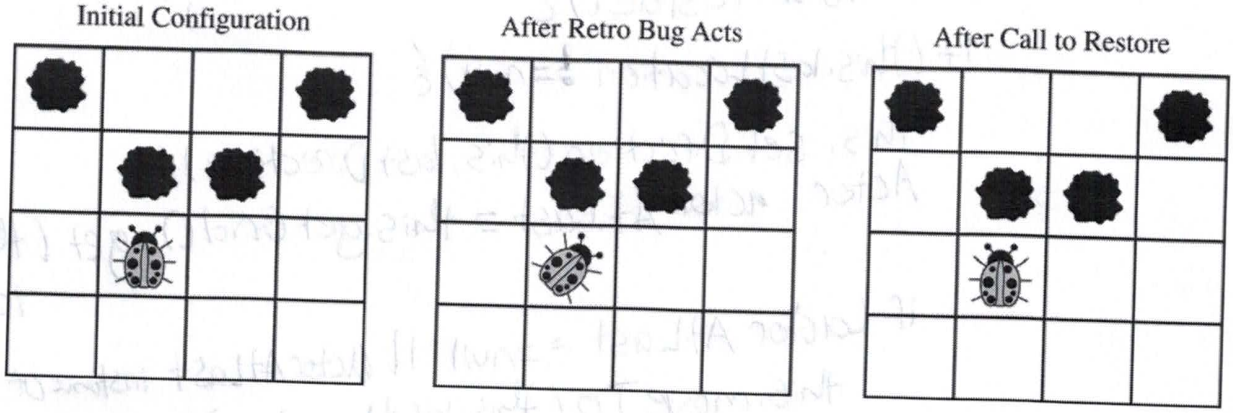
Example 2

The retro bug acts once and then some other actor moves into the location that the retro bug originally held. The call to `restore` results in the retro bug staying in its current location. The retro bug is returned to its previous direction (in this case it is the same as the current direction).



Example 3

The retro bug acts once and later calls `restore`. Because the retro bug is blocked from moving forward, it turns as its first act. The `restore` method results in the retro bug staying in its current location (the same as its previous location) and returning to its previous direction.



WRITE YOUR SOLUTION ON THE NEXT PAGE.

Write the entire `RetroBug` class, including all necessary instance variables and methods.

```
Public class RetroBug extends Bug {
    Private int lastDirection;
    Private Location lastLocation;

    Public void act() {
        this.lastDirection = this.getDirection();
        this.lastLocation = this.getLocation();
        Super.act();
    }

    Public void restore() {
        if (this.lastLocation != null) {
            this.setDirection(this.lastDirection);
            Actor actorAtLast = this.getGrid().get(this.lastLocation);
            if (actorAtLast == null || actorAtLast instanceof Flower)
                this.moveTo(this.lastLocation);
        }
    }
}
```

3. Consider a software system that models a horse barn. Classes that represent horses implement the following interface.

```
public interface Horse
{
  /** @return the horse's name */
  String getName();

  /** @return the horse's weight */
  int getWeight();

  // There may be methods that are not shown.
}
```

A horse barn consists of N numbered spaces. Each space can hold at most one horse. The spaces are indexed starting from 0; the index of the last space is N – 1. No two horses in the barn have the same name.

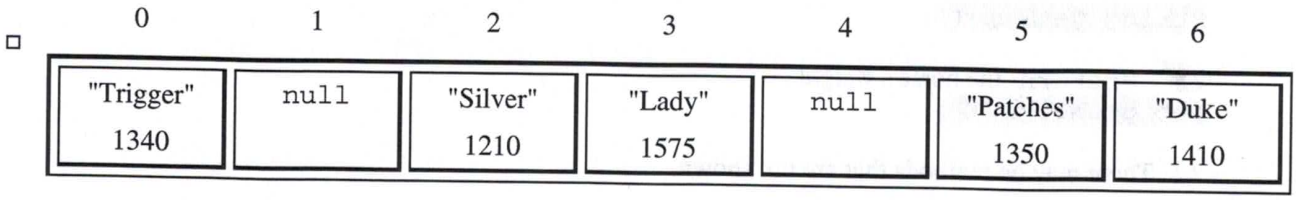
The declaration of the `HorseBarn` class is shown below. You will write two unrelated methods of the `HorseBarn` class.

```
public class HorseBarn
{
  /** The spaces in the barn. Each array element holds a reference to the horse
   *  that is currently occupying the space. A null value indicates an empty space.
   */
  private Horse[] spaces;


  /** Returns the index of the space that contains the horse with the specified name.
   *  Precondition: No two horses in the barn have the same name.
   *  @param name the name of the horse to find
   *  @return the index of the space containing the horse with the specified name;
   *          -1 if no horse with the specified name is in the barn.
   */
  public int findHorseSpace(String name)
  {  /* to be implemented in part (a) */  }


  /** Consolidates the barn by moving horses so that the horses are in adjacent spaces,
   *  starting at index 0, with no empty space between any two horses.
   *  Postcondition: The order of the horses is the same as before the consolidation.
   */
  public void consolidate()
  {  /* to be implemented in part (b) */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

Part (a) begins on page 14.

(a) Write the `HorseBarn` method `findHorseSpace`. This method returns the index of the space in which the horse with the specified name is located. If there is no horse with the specified name in the barn, the method returns -1.

For example, assume a `HorseBarn` object called `sweetHome` has horses in the following spaces.



The following table shows the results of several calls to the `findHorseSpace` method.

Method Call	Value Returned	Reason
`sweetHome.findHorseSpace("Trigger")`	0	A horse named Trigger is in space 0.
`sweetHome.findHorseSpace("Silver")`	2	A horse named Silver is in space 2.
`sweetHome.findHorseSpace("Coco")`	-1	A horse named Coco is not in the barn.

Information repeated from the beginning of the question

```
public interface Horse

String getName()
int getWeight()

public class HorseBarn

private Horse[] spaces
public int findHorseSpace(String name)
public void consolidate()
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

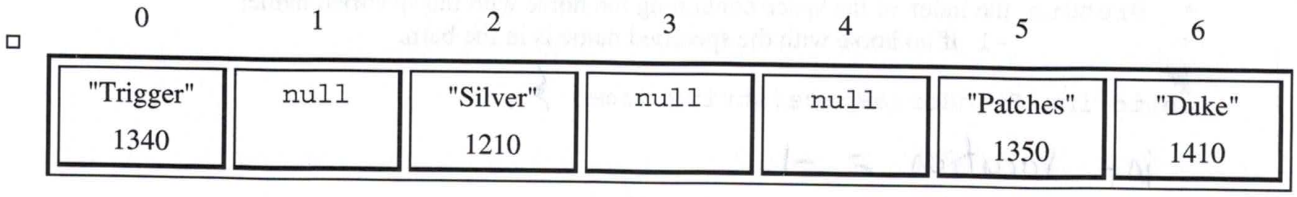
Complete method `findHorseSpace` below.

```
/** Returns the index of the space that contains the horse with the specified name.
 *  Precondition: No two horses in the barn have the same name.
 *  @param name the name of the horse to find
 *  @return the index of the space containing the horse with the specified name;
 *          -1 if no horse with the specified name is in the barn.
 */
public int findHorseSpace(String name) {
    int location = -1
    for (int x =0; x < spaces.length; x++){
        if (spaces[x] != null && (spaces[x].getName().equals(name))){
            location = x;
            break;
        }
    }
    return location;
}
```

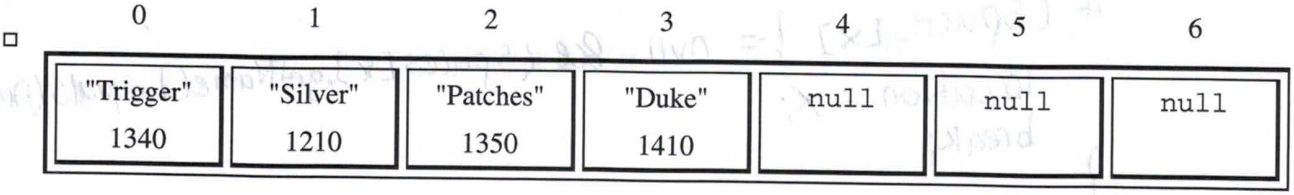
Part (b) begins on page 16.

(b) Write the `HorseBarn` method `consolidate`. This method consolidates the barn by moving horses so that the horses are in adjacent spaces, starting at index 0, with no empty spaces between any two horses. After the barn is consolidated, the horses are in the same order as they were before the consolidation.

For example, assume a barn has horses in the following spaces.



The following table shows the arrangement of the horses after `consolidate` is called.



Information repeated from the beginning of the question

```
public interface Horse

String getName()
int getWeight()

public class HorseBarn

private Horse[] spaces
public int findHorseSpace(String name)
public void consolidate()
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `consolidate` below.

```
/** Consolidates the barn by moving horses so that the horses are in adjacent spaces,
 *  starting at index 0, with no empty space between any two horses.
 *  Postcondition: The order of the horses is the same as before the consolidation.
 */
public void consolidate()
{
    Horse[] spaces2 = new Horse[spaces.length];
    int counter = 0;

    for(int x = 0; x < spaces.length; x++){
        if(spaces[x] != null){
            spaces2[counter] = spaces[x];
            counter++;
        }

    spaces = spaces2
}
```