AP® Computer Science AB
2009 Free-Response Questions

The College Board

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 5,600 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools and 3,800 colleges through major programs and services in college readiness, college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT® and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities and concerns.

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

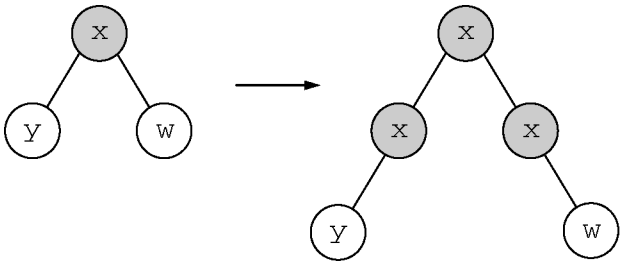
COMPUTER SCIENCE AB
SECTION II
Time—1 hour and 45 minutes
Number of questions—4
Percent of total grade—50

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

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- The `java.util.Stack` and `java.util.PriorityQueue` classes and the `java.util.Queue` interface (page A2 in the Appendix) each inherit methods that access elements in a way that violates their abstract data structure definitions. Solutions that use objects of types `Stack`, `Queue`, and `PriorityQueue` should use only the methods listed in the Appendix for accessing and modifying those objects. The use of other methods may not receive full credit.
- Assume that the implementation classes `ListNode` and `TreeNode` (page A4 in the Appendix) are used for any questions referring to linked lists or trees, unless otherwise specified.
- `ListNode` and `TreeNode` parameters may be `null`. Otherwise, unless 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.
- When Big-Oh running time is required for a response, you must use the most restrictive Big-Oh expression. For example, if the running time is $O(n)$, a response of $O(n^2)$ will not be given credit.

1. A `GrowingTree` is a binary tree that can expand at certain nodes. When a given node expands, two new nodes are created that contain the same value as contained in the given node. These new nodes become the left and right child nodes of the given node. The original left child of the given node becomes the left child of the new left child, and the original right child of the given node becomes the right child of the new right child. In the following figure, the tree on the left is expanded at the root node and becomes the tree on the right.



A partial declaration for the `GrowingTree` class is shown below.

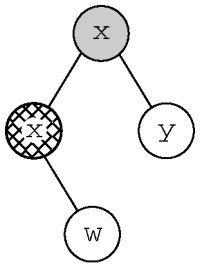
```
public class GrowingTree
{
  /** the root node of this tree */
  private TreeNode root;


  /** Creates two new nodes that each contain the same value as contained in t.
   *  These new nodes become the left and right child nodes of t.
   *  The original left child of t becomes the left child of the new left child,
   *  and the original right child of t becomes the right child of the new right child.
   *  @param t a reference to the node to be expanded
   *           Precondition: t is not null
   */
  private static void expandNode(TreeNode t)
  {  /* to be implemented in part (a) */  }


  /** Grows the tree by expanding all nodes that contain the value val
   *  @param val the value to be matched
   */
  public void growTree(Object val)
  {  growTreeHelper(root, val);  }


  /** Grows the tree by expanding all nodes containing val in the subtree rooted at current
   *  @param current the root of the subtree to be processed
   *  @param val the value to be matched
   */
  private static void growTreeHelper(TreeNode current, Object val)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `GrowingTree` method `expandNode`. The `expandNode` method creates two new nodes that contain the same value as contained in `t`. These new nodes become the left and right child nodes of `t`. The original left child of `t` becomes the left child of the new left child, and the original right child of `t` becomes the right child of the new right child.

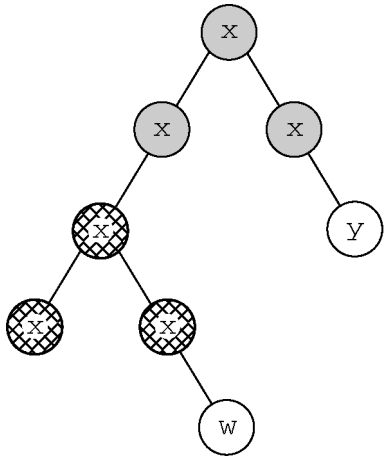
Complete method `expandNode` below.

```
/** Creates two new nodes that each contain the same value as contained in t.
 *  These new nodes become the left and right child nodes of t.
 *  The original left child of t becomes the left child of the new left child,
 *  and the original right child of t becomes the right child of the new right child.
 *  @param t a reference to the node to be expanded
 *           Precondition: t is not null
 */
private static void expandNode(TreeNode t)
```

(b) The `GrowingTree` method `growTree` expands all the nodes in the tree that contain a given value. For example, assume that the variable `GrowingTree g` has been initialized with values that represent the following tree. Note that there are 2 nodes in the tree that contain the value `x` (indicated by the two different shaded nodes).



If the statement `g.growTree(x);` is executed, the `GrowingTree g` will be modified as shown in the following tree representation. Note that the new nodes added to the tree are *not* expanded.



Write the `GrowingTree` method `growTreeHelper`. Assume that `expandNode` works as specified, regardless of what you wrote in part (a).

Complete method `growTreeHelper` below.

```
/** Grows the tree by expanding all nodes containing val in the subtree rooted at current
 *  @param current the root of the subtree to be processed
 *  @param val the value to be matched
 */
private static void growTreeHelper(TreeNode current, Object val)
```

(c) For the most efficient implementation of the `expandNode` and `growTreeHelper` operations as described above, state the Big-Oh efficiency in terms of n, where n is the number of nodes in the tree.

Method	Big-Oh
`expandNode`	
`growTreeHelper`	

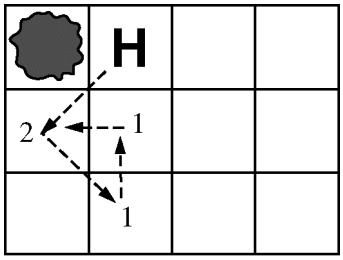
2. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

A `Historian` is a `Critter`. In addition to behaving like a regular `Critter`, it keeps track of the number of times it visits locations. A `Historian` visits a location when it moves to that location from a different location. When a `Historian` object is placed in the grid, its initial location is not considered to be visited because the `Historian` has not moved into that location from a different location.

The declaration of the `Historian` class is given below. The instance variable `places` is used to map each visited location to an integer. The integer represents the number of times the `Historian` has visited that location. The `Historian` class has a method that returns a list of locations that were visited most often. You will implement two methods of this class.

```
public class Historian extends Critter
{
  /** maps each visited location to the number of times it was visited */
  private Map<Location, Integer> places;


  public Historian()
  {
    super();
    places = new HashMap<Location, Integer>();
  }


  /** Moves this critter to the given location loc. or removes this critter
   *  from its grid if loc is null. Increments the count for location loc if and only if
   *  moving to loc from a different location and loc is not null.
   *  Postcondition: (1) getLocation() == loc.
   *  (2) The state of all actors other than those at the old and new locations is unchanged.
   *  @param loc the location to move to
   */
  public void makeMove(Location loc)
  {  /* to be implemented in part (a) */  }


  /** Precondition: this Historian has changed its location at least once.
   *  @return a list containing the location(s) this Historian
   *          has moved to most frequently.
   */
  public List<Location> mostPopularPlaces()
  {  /* to be implemented in part (b) */  }
}
```

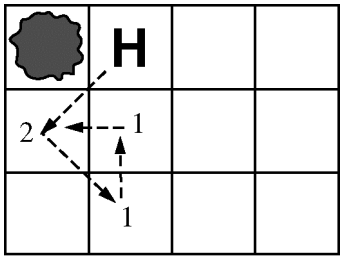
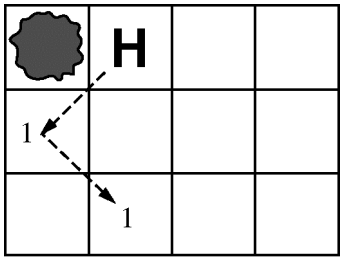
(a) Override the `makeMove` method for the `Historian` class. A `Historian` makes its move like a `Critter` and updates the instance variable `places` to increment the number of times that location `loc` has been visited. For example, assume that the `Historian` starts in location (0,1) and makes a series of moves to the following locations: (1,0), (2,1), (1,1), and (1,0). The counts for each location visited are indicated in the picture below. Note that location (1,0) has a count of 2 because the `Historian` visited that location twice. Note also that location (0,1) would not have a count associated with it because the `Historian` did not move into that location from another location.



Complete method `makeMove` below.

```
/** Moves this critter to the given location loc. or removes this critter
 *  from its grid if loc is null. Increments the count for location loc if and only if
 *  moving to loc from a different location and loc is not null.
 *  Postcondition: (1) getLocation() == loc.
 *  (2) The state of all actors other than those at the old and new locations is unchanged.
 *  @param loc the location to move to
 */
public void makeMove(Location loc)
```

(b) Write the `mostPopularPlaces` method for the `Historian` class. This method returns a list containing each location that was visited the highest number of times by this `Historian`. If only one location was visited the highest number of times, then the returned list contains only that location. If more than one location has the same highest number of visits, each of those locations is returned in the list. For example, in the first row of the table below, the returned list contains only the location (1,0) because that was the only location that was visited twice. In the second row of the table, the returned list contains location (1,0) and location (2,1) because both of them were visited once.

Moves Made by the `Historian`	`List` Returned by `mostPopularPlaces()`
	`[ (1,0) ]`
	`[ (1,0), (2,1) ]`

Assume that `makeMove` works as specified, regardless of what you wrote in part (a).

Complete method `mostPopularPlaces` below.

```
/** Precondition: this Historian has changed its location at least once.
 *  @return a list containing the location(s) this Historian
 *          has moved to most frequently.
 */
public List<Location> mostPopularPlaces()
```

3. An increasing number stream is a nonterminating sequence of integers. The first term in the stream is a positive integer. Each subsequent term in the stream is strictly greater than the previous term. Two increasing number streams are shown below.

   2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, . . .

   45, 96, 112, 221, 326, . . .

   The interface `IncreasingNumberStream`, given below, can be used to represent an increasing number stream. Assume that the Java class constant `Integer.MAX_VALUE` represents the largest value that an `int` can represent.

```
public interface IncreasingNumberStream
{
  /** @return the next term in the number stream or Integer.MAX_VALUE if the
   *          next term would be larger than the maximum int value.
   */
  int nextTerm();


  /** Restarts the number stream such that the next call to nextTerm() will return the first term
   *  in the number stream
   */
  void restart();
}
```

An arithmetic number stream is an increasing number stream in which a constant positive increment value is added to the current term to calculate the next term. For example, the positive even integers can be expressed as an arithmetic number stream with a starting value of 2 and an increment value of 2. The following table shows two examples of arithmetic number streams.

Starting Value	Increment Value	First Five Terms in the Number Stream
2	2	2, 4, 6, 8, 10
5	3	5, 8, 11, 14, 17

Write a class definition for `ArithmeticNumberStream` that implements the `IncreasingNumberStream` interface. Your class definition must include a constructor that takes a starting value and an increment value for the stream. Assume that both values passed to the constructor are positive. The first call to `nextTerm` after an `ArithmeticNumberStream` object has been constructed will return the first term in the stream.

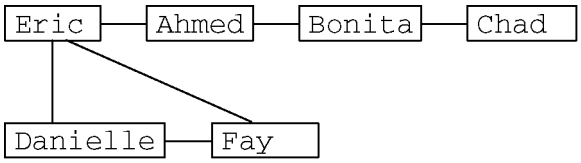
The following code segment shows an example of an `ArithmeticNumberStream` object.

```
IncreasingNumberStream stream = new ArithmeticNumberStream(12, 5);
System.out.print(stream.nextTerm() + " ");
System.out.print(stream.nextTerm() + " ");
System.out.print(stream.nextTerm() + " ");
stream.restart();
System.out.print(stream.nextTerm() + " ");
System.out.println(stream.nextTerm());
```

The output of the code segment is the terms `12 17 22 12 17`. Notice that the call to `restart` restarts the stream back to its beginning.

Write a class definition for `ArithmeticNumberStream` including all instance variables, the constructor, and all required methods.

4. Consider the following diagram that represents contact relationships within a group of people. A line between two people in the diagram indicates that they are immediate contacts of each other.



For example, the diagram shows that Eric and Ahmed are immediate contacts of each other, but Eric and Bonita are not. An immediate contact is also called a contact at distance 1. The following table shows the immediate contacts of each person in the diagram.

Person	Immediate Contacts (distance 1)
Ahmed	Bonita, Eric
Bonita	Ahmed, Chad
Chad	Bonita
Danielle	Eric, Fay
Eric	Ahmed, Danielle, Fay
Fay	Danielle, Eric

Once one has the network of contacts within distance d for a particular person, the network of contacts within distance $d + 1$ can be found by adding all the immediate contacts of members of the network within distance d. Note that all contacts within a distance less than d are included in the network of contacts within distance d. A person is never included in his or her own network of contacts.

For example, suppose that Eric wants to establish a network of contacts beyond his immediate contacts. In the diagram above, Eric's immediate contacts include Ahmed, Danielle, and Fay. Eric's network of contacts within distance 2 includes the immediate contacts of each of those members. Bonita is an immediate contact of Ahmed, who in turn is an immediate contact of Eric, making Bonita a contact within distance 2 for Eric. The following table shows each person's network of contacts within distance 2.

Person	Network of Contacts within Distance 2
Ahmed	Bonita, Chad, Danielle, Eric, Fay
Bonita	Ahmed, Chad, Eric
Chad	Ahmed, Bonita
Danielle	Ahmed, Eric, Fay
Eric	Ahmed, Bonita, Danielle, Fay
Fay	Ahmed, Danielle, Eric

The `Person` class, given below, is used to represent contact information about an individual person. The class provides methods to access a person's set of immediate contacts and a network of contacts at a specified distance.

```
public class Person
{
  /** @return a set containing all of this person's immediate contacts (distance 1)
   */
  public Set<Person> getContacts()
  {  /* implementation not shown */  }


  /** @param people a set of Person objects
   *  @return a set containing all members of people and all the
   *          immediate contacts of all members of people
   *  Postcondition: the set people is unchanged
   */
  private static Set<Person> expandContactSet(Set<Person> people)
  {  /* to be implemented in part (a) */  }


  /** Creates and returns a set containing all members of this person's network of contacts
   *  who are within distance dist away
   *  @param dist the maximum distance between this person and a network contact
   *         Precondition: dist > 0
   *  @return a set of contacts who are within distance dist away from this person
   *         Postcondition: this person is not included in the returned set
   */
  public Set<Person> getNetwork(int dist)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `Person` method `expandContactSet`. The `expandContactSet` method is a static method that when given a set of `Person` objects returns a set containing the members of the given set and each person's immediate contacts.

For example, using the relationships illustrated at the beginning of the question, if the method `expandContactSet` were called with the set containing {Ahmed, Bonita} as its parameter, the set returned would contain {Bonita, Eric, Ahmed, Chad}.

Complete method `expandContactSet` below.

```
/** @param people a set of Person objects
 *  @return a set containing all members of people and all the
 *          immediate contacts of all members of people
 *  Postcondition: the set people is unchanged
 */
private static Set<Person> expandContactSet(Set<Person> people)
```

(b) Write the `Person` method `getNetwork` that generates a network of contacts that includes all contacts who are within a given distance away from this person. The second table at the beginning of this question shows for each person the set of contacts that would be returned from the call `getNetwork(2)`.

Assume that `expandContactSet` works as specified, regardless of what you wrote in part (a).

Complete method `getNetwork` below.

```
/** Creates and returns a set containing all members of this person's network of contacts
 *  who are within distance dist away
 *  @param dist the maximum distance between this person and a network contact
 *         Precondition: dist > 0
 *  @return a set of contacts who are within distance dist away from this person
 *         Postcondition: this person is not included in the returned set
 */
public Set<Person> getNetwork(int dist)
```

STOP

END OF EXAM