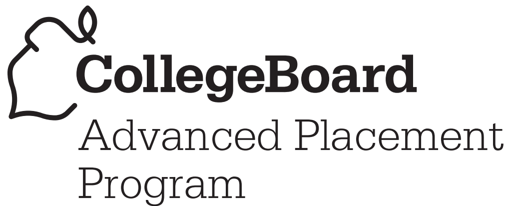


AP® Computer Science AB

Practice Exam

The questions contained in this AP® Computer Science AB Practice Exam are written to the content specifications of AP Exams for this subject. Taking this practice exam should provide students with an idea of their general areas of strengths and weaknesses in preparing for the actual AP Exam. Because this AP Computer Science AB Practice Exam has never been administered as an operational AP Exam, statistical data are not available for calculating potential raw scores or conversions into AP grades.

This AP Computer Science AB Practice Exam is provided by the College Board for AP Exam preparation. Teachers are permitted to download the materials and make copies to use with their students in a classroom setting only. To maintain the security of this exam, teachers should collect all materials after their administration and keep them in a secure location. Teachers may not redistribute the files electronically for any reason.

© 2008 The College Board. All rights reserved. College Board, Advanced Placement Program, AP, AP Central, SAT, and the acorn logo are registered trademarks of the College Board. PSAT/NMSQT is a registered trademark of the College Board and National Merit Scholarship Corporation. All other products and services may be trademarks of their respective owners. Visit the College Board on the Web: www.collegeboard.com.

Contents

Directions for Administration ii
Section I: Multiple-Choice Questions 1
Section II: Free-Response Questions 38
Quick Reference 48
Student Answer Sheet for Multiple-Choice Section 75
Multiple-Choice Answer Key 76
Free-Response Scoring Guidelines 77

The College Board: Connecting Students to College Success

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 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in 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.

AP® Computer Science AB
Directions for Administration

The AP Computer Science AB Exam is three hours in length and consists of a multiple-choice section and a free-response section.

- The 75-minute multiple-choice section contains 40 questions and accounts for 50 percent of the final grade.

- The 105-minute free-response section contains 4 questions and accounts for 50 percent of the final grade.

Students should be given a 10-minute warning prior to the end of each section of the exam. A 10-minute break should be provided after Section I is completed.

The actual AP Exam is administered in one session. Students will have the most realistic experience if a complete morning or afternoon is available to administer this practice exam. If a schedule does not permit one time period for the entire practice exam administration, it would be acceptable to administer Section I one day and Section II on a subsequent day.

Many students wonder whether or not to guess the answers to the multiple-choice questions about which they are not certain. It is improbable that mere guessing will improve a score. However, if a student has some knowledge of the question and is able to eliminate one or more answer choices as wrong, it may be to the student's advantage to answer such a question.

- The use of calculators, or any other electronic devices, is not permitted during the exam.

- It is suggested that the practice exam be completed using a pencil to simulate an actual administration.

- Teachers will need to provide paper for the students to write their free-response answers. Teachers should provide directions to the students indicating how they wish the responses to be labeled so the teacher will be able to associate the student's response with the question the student intended to answer.

- The AP Computer Science AB Exam Appendix is included with the exam materials, and each student should have a copy of this document for use with both Section I and Section II. Previously used copies of the appendix should not be distributed for the practice exam administration because students should not have access to any notes that may have been previously written into the appendix.

- Remember that students are not allowed to remove any materials, including scratch work and the appendix, from the testing site.

Section I

Multiple-Choice Questions

COMPUTER SCIENCE AB
SECTION I
Time—1 hour and 15 minutes
Number of questions—40
Percent of total grade—50

Directions: Determine the answer to each of the following questions or incomplete statements, using the available space for any necessary scratch work. Then decide which is the best of the choices given and place the letter of your choice in the corresponding box on the student answer sheet. No credit will be given for anything written in the examination booklet. Do not spend too much time on any one problem.

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- 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.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name and are not shown within a complete class definition appear within the context of an enclosing class.

1. Consider the following code segment.

```
int[] arr = {4, 2, 9, 7, 3};

for (int k : arr)
{
  k = k + 10;
  System.out.print(k + "  ");
}

for (int k : arr)
  System.out.print(k + "  ");
```

What is printed as a result of executing the code segment?

(A) `0   1   2   3   4   0   1   2   3   4`

(B) `4   2   9   7   3   4   2   9   7   3`

(C) `10   11   12   13   14   0   1   2   3   4`

(D) `14   12   19   17   13   4   2   9   7   3`

(E) `14   12   19   17   13   14   12   19   17   13`

---

2. Consider the following method.

```
public int mystery(int num)
{
  int x = num;

  while (x > 0)
  {
    if (x / 10 % 2 == 0)
      return x;

    x = x / 10;
  }

  return x;
}
```

What value is returned as a result of the call `mystery(1034)` ?

(A) 4

(B) 10

(C) 34

(D) 103

(E) 1034

3. Consider the following method.

```
public int pick(boolean test, int x, int y)
{
  if (test)
    return x;
  else
    return y;
}
```

What value is returned by the following method call?

```
pick(false, pick(true, 0, 1), pick(true, 6, 7))
```

(A) 0

(B) 1

(C) 3

(D) 6

(E) 7

GO ON TO THE NEXT PAGE.

Questions 4-5 refer to the following classes.

```
public class First
{
  public String name()
  {
    return "First";
  }
}


public class Second extends First
{
  public void whoRules()
  {
    System.out.print(super.name() + " rules");
    System.out.println(" but " + name() + " is even better");
  }

  public String name()
  {
    return "Second";
  }
}


public class Third extends Second
{
  public String name()
  {
    return "Third";
  }
}
```

GO ON TO THE NEXT PAGE.

4. Consider the following code segment.

```
Second varSecond = new Second();
Third varThird = new Third();

varSecond.whoRules();
varThird.whoRules();
```

What is printed as a result of executing the code segment?

(A)
```
First rules but Second is even better
First rules but Second is even better
```

(B)
```
First rules but Second is even better
First rules but Third is even better
```

(C)
```
First rules but Second is even better
Second rules but Second is even better
```

(D)
```
First rules but Second is even better
Second rules but Third is even better
```

(E)
```
Second rules but Second is even better
Second rules but Second is even better
```

---

5. Consider the following code segment.

```
/* SomeType1 */ varA = new Second();
/* SomeType2 */ varB = new Third();

varA.whoRules();
varB.whoRules();
```

Which of the following could be used to replace `/* SomeType1 */` and `/* SomeType2 */` so that the code segment will compile without error?

	`/* SomeType1 */`	`/* SomeType2 */`
I.	`First`	`Third`
II.	`Second`	`Second`
III.	`Third`	`Third`

(A) I only

(B) II only

(C) III only

(D) I and II

(E) II and III

6. Consider the following classes.

```
public class Base
{
  public Base()
  {
    System.out.print("Base" + "   ");
  }
}


public class Derived extends Base
{
  public Derived()
  {
    System.out.print("Derived" + "   ");
  }
}
```

Assume that the following statement appears in another class.

```
Derived d1 = new Derived();
```

What is printed as a result of executing the statement?

(A) Nothing is printed because the statement is a variable declaration.

(B) `Base`

(C) `Derived`

(D) `Base  Derived`

(E) `Derived  Base`

GO ON TO THE NEXT PAGE.

7. Consider the following method.

```
/** @param a  a queue of Integer values in nondecreasing order
 *  @param b  a queue of Integer values in nondecreasing order
 *  @return  a queue containing all values from a and b sorted in nondecreasing order
 *  Postcondition: a and b are empty
 */
public Queue<Integer> merge(Queue<Integer> a, Queue<Integer> b)
{
  Queue<Integer> result = new LinkedList<Integer>();

  while (!a.isEmpty() || !b.isEmpty())
  {
    if ( /* missing code */ )
      result.add(a.remove());
    else
      result.add(b.remove());
  }

  return result;
}
```

Which of the following expressions could be used to replace `/* missing code */` so that `merge` will work as intended?

(A) `!a.isEmpty() && b.isEmpty()`

(B) `a.peek().intValue() < b.peek().intValue()`

(C) `b.isEmpty() || (a.peek().intValue() < b.peek().intValue())`

(D) `!b.isEmpty() && (a.peek().intValue() < b.peek().intValue())`

(E)
```
b.isEmpty() ||
  (!a.isEmpty() && (a.peek().intValue() < b.peek().intValue()))
```

GO ON TO THE NEXT PAGE.

8. Consider the following method.

```
public int getTheResult(int n)
{
  int product = 1;
  for (int number = 1; number < n; number++)
  {
    if (number % 2 == 0)
      product *= number;
  }
  return product;
}
```

What value is returned as a result of the call `getTheResult(8)` ?

(A) 48

(B) 105

(C) 384

(D) 5040

(E) 40320

9. Consider the following code segment.

```
int[] oldArray = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int[][] newArray = new int[3][3];

int row = 0;
int col = 0;
for (int index = 0; index < oldArray.length; index++)
{
  newArray[row][col] = oldArray[index];
  row++;
  if ((row % 3) == 0)
  {
    col++;
    row = 0;
  }
}

System.out.println(newArray[0][2]);
```

What is printed as a result of executing the code segment?

(A) 3

(B) 4

(C) 5

(D) 7

(E) 8

GO ON TO THE NEXT PAGE.

10. Consider the following code segment.

```
ArrayList<Integer> nums = new ArrayList<Integer>();

nums.add(new Integer(37));
nums.add(new Integer(3));
nums.add(new Integer(0));
nums.add(1, new Integer(2));
nums.set(0, new Integer(1));
nums.remove(2);
System.out.println(nums);
```

What is printed as a result of executing the code segment?

(A) `[1, 2, 0]`

(B) `[1, 3, 0]`

(C) `[1, 3, 2]`

(D) `[1, 37, 3, 0]`

(E) `[37, 0, 0]`

---

11. Assume that the `boolean` variables `a`, `b`, `c`, and `d` have been declared and initialized. Consider the following expression.

```
!( !( a && b ) || ( c || !d ))
```

Which of the following is equivalent to the expression?

(A) `( a && b ) && ( !c && d )`

(B) `( a || b ) && ( !c && d )`

(C) `( a && b ) || ( c || !d )`

(D) `( !a || !b ) && ( !c && d )`

(E) `!( a && b ) && ( c || !d )`

GO ON TO THE NEXT PAGE.

12. Consider the following class declarations.

```
public class A
{
  private int x;

  public A()
  {  x = 0;  }

  public A(int y)
  {  x = y;  }

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


public class B extends A
{
  private int y;

  public B()
  {
    /* missing code */
  }

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

Which of the following can be used to replace `/* missing code */` so that the statement

```
B temp = new B();
```

will construct an object of type `B` and initialize both `x` and `y` with 0 ?

I.
```
y = 0;
```

II.
```
super(0);
y = 0;
```

III.
```
x = 0;
y = 0;
```

(A) I only

(B) II only

(C) I and II only

(D) II and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

13. Consider the following method.

```
// Precondition: b > 0
public int surprise(int b)
{
  if ((b % 2) == 0)
  {
    if (b < 10)
      return b;
    else
      return ((b % 10) + surprise(b / 10));
  }
  else
  {
    if (b < 10)
      return 0;
    else
      return surprise(b / 10);
  }
}
```

Which of the following expressions will evaluate to `true` ?

I. `surprise(146781) == 0`

II. `surprise(7754) == 4`

III. `surprise(58216) == 16`

(A) I only

(B) II only

(C) III only

(D) II and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

14. Consider the following method that is intended to return `true` if an array of integers is arranged in decreasing order and return `false` otherwise.

```
/** @param nums  an array of integers
 *  @return true  if the values in the array appear in decreasing order
 *          false  otherwise
 */
public static boolean isDecreasing(int[] nums)
{
  /* missing code */
}
```

Which of the following can be used to replace `/* missing code */` so that `isDecreasing` will work as intended?

I.
```
for (int k = 0; k < nums.length; k++)
{
  if (nums[k] <= nums[k + 1])
    return false;
}
return true;
```

II.
```
for (int k = 1; k < nums.length; k++)
{
  if (nums[k - 1] <= nums[k])
    return false;
}
return true;
```

III.
```
for (int k = 0; k < nums.length - 1; k++)
{
  if (nums[k] <= nums[k + 1])
    return false;
  else
    return true;
}
return true;
```

(A) I only

(B) II only

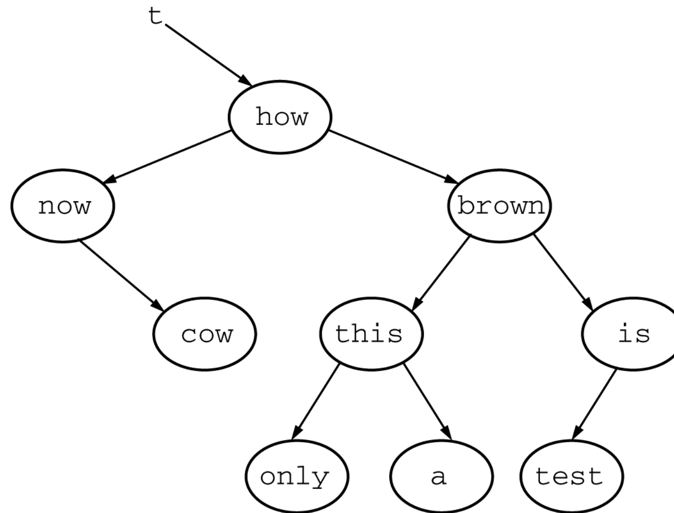
(C) III only

(D) I and III

(E) II and III

GO ON TO THE NEXT PAGE.

15. Assume that `TreeNode t` refers to the following binary tree that contains `String` objects.



Consider the following method.

```
public void funny(TreeNode p)
{
  if (p.getLeft() != null)
  {
    funny(p.getLeft());
  }
  else if (p.getRight() != null)
  {
    funny(p.getRight());
  }
  else System.out.print(p.getValue() + "." );
}
```

What is printed as a result of the call `funny(t)` ?

(A) `how.now.brown.cow.this.is.only.a.test.`

(B) `how.now.cow.brown.this.only.a.is.test.`

(C) `cow.only.a.test.`

(D) `cow.only.a.is.test.`

(E) `cow.`

16. A Web browser must keep track of the sites that you have visited so that when you click the “back” button it will return you to the most recent site. Which of the following data structures has a functionality that best supports the described display of previously visited sites?

(A) Stack

(B) Queue

(C) PriorityQueue

(D) TreeNode

(E) TreeMap

GO ON TO THE NEXT PAGE.

17. Consider the following class that stores information about temperature readings on various dates.

```
public class TemperatureReading implements Comparable
{
  private double temperature;
  private int month, day, year;

  public int compareTo(Object obj)
  {
    TemperatureReading other = (TemperatureReading) obj;
    /* missing code */
  }

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

Consider the following code segments that are potential replacements for `/* missing code */`.

I.
```
Double d1 = new Double(temperature);
Double d2 = new Double(other.temperature);
return d1.compareTo(d2);
```

II.
```
if (temperature < other.temperature)
  return -1;
else if (temperature == other.temperature)
  return 0;
else
  return 1;
```

III.
```
return (int) (temperature - other.temperature);
```

Which of the code segments could be used to replace `/* missing code */` so that `compareTo` can be used to order `TemperatureReading` objects by increasing temperature value?

(A) II only

(B) I and II only

(C) I and III only

(D) II and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

18. Consider the following method.

```
public String recScramble(String str, int[] positions, int k)
{
  if (str == null || str.length() == 0)
    return "";

  if (str.length() == 1)
    return str;

  int pos = positions[k];
  String nStr = str.substring(pos, pos + 1);
  str = str.substring(0, pos) + str.substring(pos + 1);

  return nStr + recScramble(str, positions, k + 1);
}
```

Consider the following code segment.

```
int[] indexes = {2, 1, 1};
System.out.println(recScramble("epic", indexes, 0));
```

What is printed as a result of executing the code segment?

(A) `cepi`

(B) `epci`

(C) `iecp`

(D) `iepc`

(E) `ipce`

GO ON TO THE NEXT PAGE.

19. Consider the following code segment.

```
Stack<Integer> stackOne = new Stack<Integer>();
Stack<Integer> stackTwo = new Stack<Integer>();

for (int k = 0; k < 5; k++)
{
  stackOne.push(new Integer(k));
}

while (!stackOne.isEmpty())
{
  stackTwo.push(stackOne.pop());
}

for (int k = 5; k < 10; k++)
{
  stackTwo.push(new Integer(k));
}

while (!stackTwo.isEmpty())
{
  System.out.print(stackTwo.pop() + " ");
}
```

What is printed as a result of executing the code segment?

(A) 0 1 2 3 4 5 6 7 8 9

(B) 0 1 2 3 4 9 8 7 6 5

(C) 5 6 7 8 9 0 1 2 3 4

(D) 9 8 7 6 5 0 1 2 3 4

(E) 9 8 7 6 5 4 3 2 1 0

GO ON TO THE NEXT PAGE.

20. Consider the following method.

```
public int addFun(int n)
{
  if (n <= 0)
    return 0;
  if (n == 1)
    return 2;
  return addFun(n - 1) + addFun(n - 2);
}
```

What value is returned as a result of the call `addFun(6)` ?

(A) 10

(B) 12

(C) 16

(D) 26

(E) 32

GO ON TO THE NEXT PAGE.

Questions 21-25 refer to the code from the GridWorld case study. A copy of the code is provided in the Appendix.

21. A `CornerBug` behaves like a `Bug` except that a `CornerBug` makes all turns at right angles rather than 45-degree angles. Of the following, which is the best design for the `CornerBug` class?

    (A) Create an abstract class called `RightAngleBug` that is a `Bug` that only turns 90 degrees, and then create a class `CornerBug` that inherits from `RightAngleBug`.

    (B) Create an interface called `RightTurn` that includes the specification of a `turnRight` method, and then create a class `CornerBug` that implements `RightTurn`.

    (C) Create a class `CornerBug` that inherits from `Bug` and adds a constructor that has an `int` parameter that determines if the bug should turn 90 degrees or 45 degrees.

    (D) Create a class `CornerBug` that inherits from `Bug` and overrides the `Bug` `turn` method to turn the bug 90 degrees instead of 45 degrees.

    (E) Create an interface `CornerBug` that includes the definition of a `turnRight` method that is automatically used by the `Bug` `act` method for objects that are instantiated as `CornerBug` objects.

---

22. Consider the following class that inherits from `Bug` and overrides the `turn` method.

```
public class LeftTurningBug extends Bug
{
  public void turn()
  {
    /* missing code */
  }
}
```

Which of the following can be used to replace `/* missing code */` so that a `LeftTurningBug` object will always turn 90 degrees to the left?

I. `setDirection(Location.LEFT);`

II. `setDirection(getDirection() + Location.LEFT);`

III. `setDirection(Location.HALF_LEFT + Location.HALF_LEFT);`

(A) I only

(B) II only

(C) III only

(D) I and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

23. Consider the following class declaration.

```
public class MysteryBug extends Bug
{
  private int count;

  public MysteryBug()
  {
    setDirection(Location.NORTHEAST);
    count = 0;
  }

  public void act()
  {
    if (count < 3 && canMove())
    {
      move();
      count++;
    }
    else
    {
      turn();
      count = 0;
    }
  }
}
```

Assume that a `MysteryBug` object has been placed in the center of an otherwise empty grid that is 20 rows by 20 columns. What pattern of flowers is formed if the `act` method of the `MysteryBug` is called many times in a row?

(A) No pattern is formed because the `MysteryBug` never puts any flowers into the world.

(B) A square pattern is formed just as with a `BoxBug` object.

(C) A diamond pattern is formed (a square that has been rotated 45 degrees).

(D) A six-sided pattern is formed.

(E) An eight-sided pattern is formed.

GO ON TO THE NEXT PAGE.

24. A `PouncingCritter` behaves like a `Critter` except that when it moves, it randomly selects one of the grid locations that contains an `Actor` and moves to that location.

Which of the following methods must be changed to allow a `PouncingCritter` to behave this way while also preserving the `Critter` postconditions?

I. `getMoveLocations`

II. `makeMove`

III. `selectMoveLocation`

(A) I only

(B) II only

(C) III only

(D) I and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

25. Consider the following method that is intended to move the parameter `anActor` to a different grid that is referred to by the parameter `newGrid`. The location of `anActor` in `newGrid` should be the same as the location that `anActor` had occupied in its original grid.

```
/** Moves anActor to newGrid in the same location it occupied in its original grid.
 *  @param anActor the actor to be moved
 *  @param newGrid the grid in which the actor is to be placed
 */
public void moveActorToNewGrid(Actor anActor, Grid<Actor> newGrid)
{
  Grid<Actor> oldGrid = anActor.getGrid();
  Location loc = anActor.getLocation();
  /* missing code */
}
```

Which of the following can be used to replace `/* missing code */` so that `moveActorToNewGrid` will work as intended?

(A)
```
anActor.putSelfInGrid(newGrid, loc);
anActor.removeSelfFromGrid();
```

(B)
```
oldGrid.remove(loc);
anActor.putSelfInGrid(newGrid, loc);
```

(C)
```
anActor.removeSelfFromGrid();
anActor.putSelfInGrid(newGrid, loc);
```

(D)
```
oldGrid.remove(loc);
newGrid.put(anActor, loc);
```

(E)
```
newGrid.put(anActor, loc);
oldGrid.remove(loc);
```

GO ON TO THE NEXT PAGE.

Questions 26-27 refer to the following method.

```
/** @param count  the number of elements to place in the queue
 *          Precondition: count > 0
 *  @return  a queue of integers
 */
public static Queue<Integer> fillQ(int count)
{
  Queue<Integer> intQ = new LinkedList<Integer>();
  int x = 3;
  int y = 2;
  int t = 1;

  if (count == 1)
    intQ.add(new Integer(y));
  else
  {
    intQ.add(new Integer(y));
    intQ.add(new Integer(t));
    for (int k = 2; k < count; k++)
    {
      x = y + t - k;
      y = t;
      t = x + k;

      intQ.add(new Integer(t));    // Point A
    }
  }

  return intQ;
}
```

GO ON TO THE NEXT PAGE.

26. What will be printed as a result of executing the following statement?

```
System.out.println(fillQ(6));
```

(A) `[1, 2, 3, 5, 8, 13]`

(B) `[2, 1, 3, 4, 7, 11]`

(C) `[2, 1, 3, 6, 10, 15]`

(D) `[2, 1, 3, 6, 12, 24]`

(E) `[2, 3, 5, 8, 12, 17]`

---

27. Which of the following best describes the value added to the queue at `// Point A` ?

(A) It is the sum of all the previous values that have been added to the queue.

(B) It is the sum of all the previous odd values that have been added to the queue.

(C) It is the sum of the two immediately previous values that have been added to the queue.

(D) It is the sum of the loop index and the immediately previous value that has been added to the queue.

(E) It is the square of the loop index.

GO ON TO THE NEXT PAGE.

28. Consider the following method.

```
/** @param t  a reference to the root of a binary tree of integers
 *           Precondition:  t  is not  null
 */
public int mystery(TreeNode t)
{
  int keep = ((Integer) t.getValue()).intValue();
  int leftKeep = keep;
  int rightKeep = keep;

  if (t.getLeft() != null)
    leftKeep = mystery(t.getLeft());

  if (t.getRight() != null)
    rightKeep = mystery(t.getRight());

  if (keep >= leftKeep)
    keep = leftKeep;

  if (keep >= rightKeep)
    keep = rightKeep;

  return keep;
}
```

Assume that `TreeNode root` is a reference to a nonempty binary tree of integers. Which of the following best describes the value returned by the call `mystery(root)` ?

(A) The smallest integer in the tree

(B) The largest integer in the tree

(C) The value in the root of the tree

(D) The value in the rightmost leaf of the tree

(E) The value in the leftmost leaf of the tree

29. Consider the following partial implementation of a `ListNodeIterator` that can be used to iterate over a linked list that has been constructed with `ListNode` objects.

```
public class ListNodeIterator implements Iterator
{
  private ListNode current;   // the next item to be returned from the list

  public ListNodeIterator(ListNode first)
  {  current = first;  }

  public Object next()
  {
    Object value = current.getValue();
    current = current.getNext();
    return value;
  }

  public boolean hasNext()
  {
    /* missing code */
  }

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

Which of the following can be used to replace `/* missing code */` so that method `hasNext()` will work as specified in the `Iterator` interface?

(A) `return current == null;`

(B) `return current != null;`

(C) `return current.getValue() != null;`

(D) `return current.getNext() != null;`

(E) `return current.getNext().getValue() != null;`

GO ON TO THE NEXT PAGE.

30. Consider the following method.

```
/** Removes all occurrences of nameToRemove from nameList.
 *  @param nameList a list of names
 *  @param nameToRemove a name to be removed from nameList
 */
public void removeName(List<String> nameList, String nameToRemove)
{
  /* missing implementation */
}
```

Which of the following can be used to replace `/* missing implementation */` so that `removeName` will work as intended?

I.
```
for (String name : nameList)
{
  if (name.equals(nameToRemove))
    name.remove();
}
```

II.
```
for (int k = 0; k < nameList.size(); k++)
{
  if (nameList.get(k).equals(nameToRemove))
    nameList.remove(k);
}
```

III.
```
for (int k = nameList.size() - 1; k >= 0; k--)
{
  if (nameList.get(k).equals(nameToRemove))
    nameList.remove(k);
}
```

(A) I only

(B) II only

(C) III only

(D) II and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

31. Assume that an inorder traversal of a particular binary tree of letters produces the following output.

```
T  C  L  S  O  N  A  D
```

A preorder traversal of the same binary tree produces the following output.

```
S  C  T  L  O  A  N  D
```

Which of the following letters is in the right child of the root of the tree?

(A) `A`

(B) `L`

(C) `N`

(D) `O`

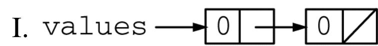
(E) `S`

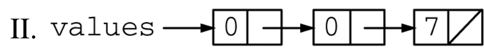
GO ON TO THE NEXT PAGE.

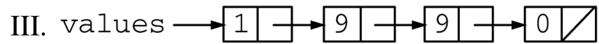
32. Consider the following method.

```
/** @param nums  a reference to a linked list of  Integer  objects
 *           Precondition:  nums  is not  null
 *  @return  the sum of all values in  nums
 */
public int sum(ListNode nums)
{
  if (nums.getNext() == null)
    return 0;
  else
    return ((Integer) nums.getValue()).intValue() + sum(nums.getNext());
}
```

For which of the following lists will the call `sum(values)` return the correct total of all the values in the list `values` ?

I. 

II. 

III. 

(A) I only

(B) I and II only

(C) I and III only

(D) II and III only

(E) I, II, and III

GO ON TO THE NEXT PAGE.

33. Assume that the map instantiated below is used to organize information about employees such that the employee number is the key and the employee name is the associated value.

```
Map<String, String> empMap = new TreeMap<String, String>();
```

Which of the following code segments will correctly print the employee number and name of each employee?

I.
```
for (String key : empMap.keySet())
{
  System.out.println(key + " " + empMap.get(key));
}
```

II.
```
Set<String> keys = empMap.keySet();
Iterator<String> itr = keys.iterator();
while (itr.hasNext())
{
  System.out.println(keys + " " + itr.next());
}
```

III.
```
Set<String> keys = empMap.keySet();
Iterator<String> itr = keys.iterator();
while (itr.hasNext())
{
  System.out.println(itr.next() + " " +  empMap.get(itr.next()));
}
```

(A) I only

(B) II only

(C) III only

(D) I and II

(E) I and III

GO ON TO THE NEXT PAGE.

34. Assume that the `boolean` variables `a` and `b` have been declared and initialized. Consider the following expression.

```
(a && (b || !a)) == a && b
```

Which of the following best describes the conditions under which the expression will evaluate to `true`?

(A) Only when `a` is `true`

(B) Only when `b` is `true`

(C) Only when both `a` and `b` are `true`

(D) The expression will never evaluate to `true`.

(E) The expression will always evaluate to `true`.

---

35. When a mergesort is used to sort 5,000 items on a particular computer, the code executes for 45 time units. Approximately how long would the mergesort on the computer take to sort 20,000 items?

(A) 90 time units

(B) 150 time units

(C) 220 time units

(D) 500 time units

(E) 720 time units

GO ON TO THE NEXT PAGE.

36. The following letters are inserted into an empty binary search tree in the following order.

```
T  R  U  L  Y  B  A  S  I  C
```

Which of the following will be printed by a preorder traversal of the tree?

(A) A B C I L R S T U Y

(B) A C I B L S R Y U T

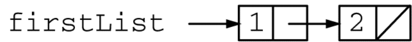
(C) T R L B A I C S U Y

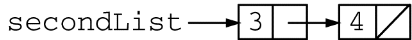
(D) T R U L S Y B A I C

(E) Y U S C I A B L R T

---

37. Assume that variables `firstList` and `secondList` of type `ListNode` have been initialized to refer to the following lists of `Integer` objects.

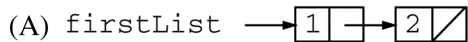


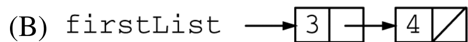


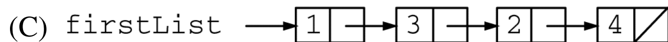
Consider the following code segment.

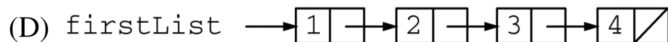
```
firstList.getNext().setNext(secondList.getNext());
secondList.setNext(firstList.getNext());
firstList.setNext(secondList);
secondList = null;
```

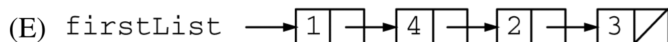
Which of the following represents the list referenced by `firstList` as a result of executing the code segment?

(A) 

(B) 

(C) 

(D) 

(E) 

GO ON TO THE NEXT PAGE.

38. Consider the following code segment.

```
PriorityQueue<String> pq = new PriorityQueue<String>();

pq.add("to");
pq.add("be");
pq.add("or");
pq.add("not");
pq.add("to");
pq.add("be");
while (!pq.isEmpty())
{
  System.out.print(pq.remove() + " ");
}
```

What is printed as a result of executing the code segment?

(A) `to to or not be be`

(B) `to be or not to be`

(C) `be to not or be to`

(D) `be not or to`

(E) `be be not or to to`

GO ON TO THE NEXT PAGE.

39. Consider the following method.

```
public int computeSum(int num)
{
  int sum = 0;
  int delta = 2;

  while (delta < 2 * num)
  {
    for (int k = delta / 2; k <= num; k = k + delta)
    {
      sum += k;
    }

    delta = delta * 2;
  }
  return sum;
}
```

Which of the following best characterizes the running time of the call `computeSum(n)` ?

(A) $O(\log n)$

(B) $O\left((\log n)^2\right)$

(C) $O(n)$

(D) $O(n \log n)$

(E) $O\left(n^2\right)$

GO ON TO THE NEXT PAGE.

40. Consider the following code segment.

```
Set<String> cities = new TreeSet<String>();

String[] stringArray = {"Philadelphia", "Minneapolis", "Houston"};
for (String str : stringArray)
{
  cities.add(str);
}

String[] stringArray2 = {"Philadelphia", "Chicago"};
for (String str : stringArray2)
{
  cities.remove(str);
}

String[] stringArray3 = {"Des Moines", "Houston"};
for (String str : stringArray3)
{
  cities.add(str);
}

System.out.println(cities);
```

What is printed as a result of executing the code segment?

(A) `[Chicago, Des Moines, Houston, Minneapolis, Philadelphia]`

(B) `[Des Moines, Houston, Houston, Minneapolis]`

(C) `[Des Moines, Houston, Houston, Minneapolis, Philadelphia]`

(D) `[Des Moines, Houston, Minneapolis]`

(E) `[Houston, Minneapolis, Philadelphia]`

END OF SECTION I

IF YOU FINISH BEFORE TIME IS CALLED,
YOU MAY CHECK YOUR WORK ON THIS SECTION.

DO NOT GO ON TO SECTION II UNTIL YOU ARE TOLD TO DO SO.

Section II

Free-Response Questions

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. The class `Gradebook` is used by a teacher to record and process student scores. One of the instance variables in the `Gradebook` class is a map that associates each student name with a score. You will implement two methods of the `Gradebook` class.

```
public class Gradebook
{
  private Map<String, Integer> scores;   // key is student name, value is score


  /** @return a map that is the inverse of the scores map. In the returned map, each key
   *          is a score, and the associated value is a Set containing the names of those
   *          students who earned that score.
   */
  private Map<Integer, Set<String>> inverseMap()
  {  /* to be implemented in part (a) */  }


  /** Precondition: scores.size() > 0
   *  @return a set of students who earned the mode score. The mode score is the score earned
   *          by the largest number of students. If there is more than one score tied for the
   *          mode, any one of those scores can be chosen as the mode.
   */
  public Set<String> modeSet()
  {  /* to be implemented in part (b) */  }

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

GO ON TO THE NEXT PAGE.

(a) Write the `Gradebook` method `inverseMap`, which returns a map that is the inverse of the `scores` map. Each score from the original map becomes a key in the inverse map. In the inverse map, the value associated with a score is a `Set` containing the names of the students who received that score.

Complete method `inverseMap` below.

```
/** @return a map that is the inverse of the scores map. In the returned map, each key
 *          is a score, and the associated value is a Set containing the names of those
 *          students who earned that score.
 */
private Map<Integer, Set<String>> inverseMap()
```

(b) Write the `Gradebook` method `modeSet`, which returns the `Set` of students who received the mode score. The mode score is the score received by the largest number of students. If there is more than one score tied for the mode, any one of those scores can be chosen as the mode.

In writing `modeSet`, you may assume that `inverseMap` works as specified, regardless of what you wrote in part (a).

Complete method `modeSet` below.

```
/** Precondition: scores.size() > 0
 *  @return a set of students who earned the mode score. The mode score is the score earned
 *          by the largest number of students. If there is more than one score tied for the
 *          mode, any one of those scores can be chosen as the mode.
 */
public Set<String> modeSet()
```

GO ON TO THE NEXT PAGE.

2. A bag is a generalization of the `Set` container. A `Set` contains at most one occurrence of a particular element, whereas a `Bag` can contain any number of occurrences of each element. The following declaration specifies the interface for a `StringBag`, a bag that holds elements of type `String`. In part (a) of this question, you will analyze a particular implementation of the `StringBag` interface. In part (b) of this question, you will write an implementation of the `StringBag` interface that must satisfy certain performance specifications.

```
public interface StringBag
{
  /** Adds element into this StringBag
   */
  public void add(String element);


  /** @return an array of the unique elements from this StringBag, arranged in sorted order
   */
  public String[] getUniqueElements();


  /** @return the number of times that element has been added into this StringBag
   */
  public int getNumOccurrences(String element);

}
```

(a) One implementation of the `StringBag` interface uses an `ArrayList` to hold the elements that have been added into the bag. Each unique `String` value will appear only once and a count is maintained for the number of times that the `String` has been added into the bag. Information about the elements in this implementation of the `StringBag` interface will be represented by objects of the `Entry` class, as shown below.

```
public class Entry
{
  private String element;
  private int count;

  public Entry(String str)
  {
    element = str;
    count = 1;
  }

  public String getElement()
  {  return element;  }

  public int getCount()
  {  return count;  }

  public void increment()
  {  count++;  }

}
```

GO ON TO THE NEXT PAGE.

The class `ListStringBag` below correctly implements the `StringBag` interface.

```
public class ListStringBag implements StringBag
{
  private ArrayList<Entry> entries;

  public ListStringBag()
  {  entries = new ArrayList<Entry>();  }


  /** Adds element into this StringBag
   */
  public void add(String element)
  {
    for (Entry cur : entries)
    {
      if (cur.getElement().equals(element))
      {
        cur.increment();
        return;
      }
    }
    entries.add(new Entry(element));
  }


  /** @return an array of the unique elements from this StringBag, arranged in sorted order
   */
  public String[] getUniqueElements()
  {
    String[] result = new String[entries.size()];
    for (int k = 0; k < entries.size(); k++)
      result[k] = entries.get(k).getElement();

    sortArray(result);

    return result;
  }
```

GO ON TO THE NEXT PAGE.

```
  /**  Uses a Mergesort algorithm to sort  stringArray  in alphabetical order.
   *   Postcondition:  stringArray  is in sorted order
   */
  private void sortArray(String[] stringArray)
  {   /*  implementation not shown  */   }


  /**  @return  the number of times that  element  has been added into this  StringBag
   */
  public int getNumOccurrences(String element)
  {
    for (Entry cur : entries)
    {
      if (cur.getElement().equals(element))
        return cur.getCount();
    }

    return 0;
  }
}
```

Give the average-case Big-Oh running time for the `ListStringBag` methods `add` and `getUniqueElements` in terms of n, the number of unique elements.

	Average-case Big-Oh
`add`	
`getUniqueElements`	

(b) A potential client requests an efficient implementation of the `StringBag` interface that meets the following Big-Oh running time specifications.

	Average-case Big-Oh
`add`	$O(\log n)$
`getUniqueElements`	$O(n)$

Write a class `FastStringBag` that correctly implements the `StringBag` interface and meets the requested running time specifications outlined in the table. You are not required to use the `Entry` class in your solution.

GO ON TO THE NEXT PAGE.

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

   A `TimidCritter` is a `Critter` that prefers to stay close to its original location within the grid. For the first ten calls to `act`, a `TimidCritter` moves in the same manner as an ordinary `Critter`. In the next ten calls to `act`, the `TimidCritter` retraces its movement back to its original location. Retracing means that the critter moves to the locations that it occupied before each of the first ten moves, but in the opposite order. After the ten retracing moves have been completed, the `TimidCritter` ends up at its original location. The `TimidCritter` will repeat the cycle indefinitely; that is, it will move normally for ten steps, retrace to its original location in the ten steps that follow, then move again normally for ten steps, and so on.

   When the `TimidCritter` is retracing, it will move even if the desired location is occupied. (This will cause the other occupant to be removed from the grid.) Before moving, the `TimidCritter` processes its neighbors in the same way as a `Critter`.

   Write the complete `TimidCritter` class, including all instance variables, a constructor, and the `getMoveLocations` and `makeMove` methods. Do NOT override the `act` method.

   Note: The `getMoveLocations` method has as a postcondition that the state of all actors is unchanged. Be sure not to change the state of your `TimidCritter` in that method. You may update the state in the `makeMove` method.

GO ON TO THE NEXT PAGE.

4. A retail store maintains a list of items that it has in stock. The items are represented by objects of the `Item` class as declared below.

```
public class Item
{
  /** @return the name of this Item
   */
  public String getName()
  {  /* implementation not shown */  }

  /** @return the price of this Item
   */
  public double getPrice()
  {  /* implementation not shown */  }

  /** @return true if this Item is in stock; false otherwise
   */
  public boolean isInStock()
  {  /* implementation not shown */  }

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

}
```

The `Inventory` class stores `Item` objects in a linked list that is implemented using the standard `ListNode` class. Each `Item` object is stored as the data value in a `ListNode` object. The `Inventory` class has an instance variable called `front` that stores a reference to the first node in the list. An empty list is represented as `null`. A partial declaration of the `Inventory` class is as follows. You will implement two methods in the `Inventory` class.

```
public class Inventory
{
  private ListNode front;   // a reference to first node in the list

  /** Constructs an empty list of items
   */
  public Inventory()
  {  front = null;  }


  /** @param newItem an Item to add to this Inventory
   */
  public void addItem(Item newItem)
  {  /* to be implemented in part (a) */  }


  /** Removes from this Inventory all items that are not in stock
   */
  public void removeUnavailableItems()
  {  /* to be implemented in part (b) */  }

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

GO ON TO THE NEXT PAGE.

(a) Write the `Inventory` method `addItem`. A new node containing the `Item` is added to the front of the existing list of items.

Complete method `addItem` below.

```
/** @param newItem an Item to add to this Inventory
 */
public void addItem(Item newItem)
```

(b) Write the `Inventory` method `removeUnavailableItems`. The method should remove all nodes from the list that contain items that are not currently in stock.

Complete method `removeUnavailableItems` below.

```
/** Removes from this Inventory all items that are not in stock
 */
public void removeUnavailableItems()
```

STOP

END OF EXAM

Quick Reference
AP® Computer Science AB

© 2008 The College Board. All rights reserved. Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

Content of Appendixes

Appendix A . . . . AB Exam Java Quick Reference
Appendix B . . . . Testable API
Appendix C . . . . Testable Code for APCS A/AB
Appendix D . . . . Testable Code for APCS AB Only
Appendix E . . . . Quick Reference A/AB
Appendix F . . . . Quick Reference AB Only
Appendix G . . . . Index for Source Code

Appendix A — AB Exam Java Quick Reference

Accessible Methods from the Java Library That May Be Included on the Exam

class java.lang.Object

- `boolean equals(Object other)`
- `String toString()`
- `int hashCode()`

interface java.lang.Comparable*

- `int compareTo(Object other)` // returns a value < 0 if `this` is less than `other`
  // returns a value = 0 if `this` is equal to `other`
  // returns a value > 0 if `this` is greater than `other`

class java.lang.Integer implements java.lang.Comparable*

- `Integer(int value)`
- `int intValue()`

class java.lang.Double implements java.lang.Comparable*

- `Double(double value)`
- `double doubleValue()`

class java.lang.String implements java.lang.Comparable*

- `int length()`
- `String substring(int from, int to)` // returns the substring beginning at `from`
  // and ending at `to-1`
- `String substring(int from)` // returns `substring(from, length())`
- `int indexOf(String str)` // returns the index of the first occurrence of `str;`
  // returns `-1` if not found

class java.lang.Math

- `static int abs(int x)`
- `static double abs(double x)`
- `static double pow(double base, double exponent)`
- `static double sqrt(double x)`
- `static double random()` // returns a `double` in the range [0.0, 1.0)

* The AP Java subset uses the "raw" `Comparable` interface, not the generic `Comparable<T>` interface.

interface java.util.List<E>

- int size()
- boolean add(E obj) // appends obj to end of list; returns true
- void add(int index, E obj) // inserts obj at position index (0 ≤ index ≤ size), // moving elements at position index and higher // to the right (adds 1 to their indices) and adjusts size
- E get(int index)
- E set(int index, E obj) // replaces the element at position index with obj // returns the element formerly at the specified position
- E remove(int index) // removes element from position index, moving elements // at position index + 1 and higher to the left // (subtracts 1 from their indices) and adjusts size // returns the element formerly at the specified position
- Iterator<E> iterator()
- ListIterator<E> listIterator()

class java.util.ArrayList<E> implements java.util.List<E>

class java.util.LinkedList<E> implements java.util.List<E>, java.util.Queue<E>

- void addFirst(E obj)
- void addLast(E obj)
- E getFirst()
- E getLast()
- E removeFirst()
- E removeLast()

interface java.util.Queue<E> // implemented by LinkedList<E>

- boolean add(E obj) // enqueues obj at the end of the queue; returns true
- E remove() // dequeues and returns the element at the front of the queue
- E peek() // returns the element at the front of the queue; // null if the queue is empty
- boolean isEmpty()

class java.util.PriorityQueue<E> // E should implement Comparable*

- boolean add(E obj) // adds obj to the priority queue; returns true
- E remove() // removes and returns the minimal element from the priority queue
- E peek() // returns the minimal element from the priority queue; // null if the priority queue is empty
- boolean isEmpty()

class java.util.Stack<E>

- E push(E item) // pushes item onto the top of the stack; returns item
- E pop() // removes and returns the element at the top of the stack
- E peek() // returns the element at the top of the stack; // throws an exception if the stack is empty
- boolean isEmpty()

*The AP Java subset uses the “raw” Comparable interface, not the generic Comparable<T> interface.

```
interface java.util.Iterator<E>
•  boolean hasNext()
•  E next()
•  void remove()                  // removes the last element that was returned by next
```

```
interface java.util.ListIterator<E> extends java.util.Iterator<E>
•  void add(E obj)                // adds obj before the element that will be returned by next
•  void set(E obj)                // replaces the last element returned by next with obj
```

```
interface java.util.Set<E>
•  int size()
•  boolean contains(Object obj)
•  boolean add(E obj)             // if obj is not present in this set, adds obj and returns true;
                                  // otherwise, returns false
•  boolean remove(Object obj)     // if obj is present in this set, removes obj and returns true;
                                  // otherwise, returns false
•  Iterator<E> iterator()
```

```
class java.util.HashSet<E> implements java.util.Set<E>
class java.util.TreeSet<E> implements java.util.Set<E>
```

```
interface java.util.Map<K,V>
•  int size()
•  boolean containsKey(Object key)
•  V put(K key, V value)          // associates key with value
                                  // returns the value formerly associated with key
                                  // or null if key was not present
•  V get(Object key)              // returns the value associated with key
                                  // or null if there is no associated value
•  V remove(Object key)           // removes and returns the value associated with key;
                                  // returns null if there is no associated value
•  Set<K> keySet()
```

```
class java.util.HashMap<K,V> implements java.util.Map<K,V>
class java.util.TreeMap<K,V> implements java.util.Map<K,V>
```

Implementation classes for linked list and tree nodes

Unless otherwise noted, assume that a linked list implemented from the `ListNode` class does not have a dummy header node.

```
public class ListNode
{
  private Object value;
  private ListNode next;

  public ListNode(Object initValue, ListNode initNext)
    { value = initValue; next = initNext; }

  public Object getValue() { return value; }
  public ListNode getNext() { return next; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setNext(ListNode theNewNext) { next = theNewNext; }
}
```

Unless otherwise noted, assume that a tree implemented from the `TreeNode` class does not have a dummy root node.

```
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(Object initValue)
    { value = initValue; left = null; right = null; }

  public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)
    { value = initValue; left = initLeft; right = initRight; }

  public Object getValue() { return value; }
  public TreeNode getLeft() { return left; }
  public TreeNode getRight() { return right; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }
  public void setRight(TreeNode theNewRight) { right = theNewRight; }
}
```

Appendix B — Testable API

`info.gridworld.grid.Location` class (`implements Comparable`)

`public Location(int r, int c)`
constructs a location with given row and column coordinates

`public int getRow()`
returns the row of this location

`public int getCol()`
returns the column of this location

`public Location getAdjacentLocation(int direction)`
returns the adjacent location in the direction that is closest to `direction`

`public int getDirectionToward(Location target)`
returns the closest compass direction from this location toward `target`

`public boolean equals(Object other)`
returns `true` if `other` is a `Location` with the same row and column as this location; `false` otherwise

`public int hashCode()`
returns a hash code for this location

`public int compareTo(Object other)`
returns a negative integer if this location is less than `other`, zero if the two locations are equal, or a positive integer if this location is greater than `other`. Locations are ordered in row-major order.
Precondition: `other` is a `Location` object.

`public String toString()`
returns a string with the row and column of this location, in the format (row, col)

Compass directions:

```
public static final int NORTH = 0;
public static final int EAST = 90;
public static final int SOUTH = 180;
public static final int WEST = 270;
public static final int NORTHEAST = 45;
public static final int SOUTHEAST = 135;
public static final int SOUTHWEST = 225;
public static final int NORTHWEST = 315;
```

Turn angles:

```
public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;
```

`info.gridworld.grid.Grid<E>` interface

`int getNumRows()`
returns the number of rows, or -1 if this grid is unbounded

`int getNumCols()`
returns the number of columns, or -1 if this grid is unbounded

`boolean isValid(Location loc)`
returns `true` if `loc` is valid in this grid, `false` otherwise
Precondition: `loc` is not `null`

`E put(Location loc, E obj)`
puts `obj` at location `loc` in this grid and returns the object previously at that location (or `null` if the location was previously unoccupied).
Precondition: (1) `loc` is valid in this grid (2) `obj` is not `null`

`E remove(Location loc)`
removes the object at location `loc` from this grid and returns the object that was removed (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`E get(Location loc)`
returns the object at location `loc` (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedLocations()`
returns an array list of all occupied locations in this grid

`ArrayList<Location> getValidAdjacentLocations(Location loc)`
returns an array list of the valid locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getEmptyAdjacentLocations(Location loc)`
returns an array list of the valid empty locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedAdjacentLocations(Location loc)`
returns an array list of the valid occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<E> getNeighbors(Location loc)`
returns an array list of the objects in the occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`info.gridworld.actor.Actor` class

`public Actor()`
constructs a blue actor that is facing north

`public Color getColor()`
returns the color of this actor

`public void setColor(Color newColor)`
sets the color of this actor to `newColor`

`public int getDirection()`
returns the direction of this actor, an angle between 0 and 359 degrees

`public void setDirection(int newDirection)`
sets the direction of this actor to the angle between 0 and 359 degrees that is equivalent to `newDirection`

`public Grid<Actor> getGrid()`
returns the grid of this actor, or `null` if this actor is not contained in a grid

`public Location getLocation()`
returns the location of this actor, or `null` if this actor is not contained in a grid

`public void putSelfInGrid(Grid<Actor> gr, Location loc)`
puts this actor into location `loc` of grid `gr`. If there is another actor at `loc`, it is removed.
Precondition: (1) This actor is not contained in a grid (2) `loc` is valid in `gr`

`public void removeSelfFromGrid()`
removes this actor from its grid.
Precondition: this actor is contained in a grid

`public void moveTo(Location newLocation)`
moves this actor to `newLocation`. If there is another actor at `newLocation`, it is removed.
Precondition: (1) This actor is contained in a grid (2) `newLocation` is valid in the grid of this actor

`public void act()`
reverses the direction of this actor. Override this method in subclasses of `Actor` to define types of actors with different behavior

`public String toString()`
returns a string with the location, direction, and color of this actor

info.gridworld.actor.Rock class (extends Actor)

`public Rock()`
constructs a black rock

`public Rock(Color rockColor)`
constructs a rock with color `rockColor`

`public void act()`
overrides the `act` method in the `Actor` class to do nothing

info.gridworld.actor.Flower class (extends Actor)

`public Flower()`
constructs a pink flower

`public Flower(Color initialColor)`
constructs a flower with color `initialColor`

`public void act()`
causes the color of this flower to darken

Appendix C — Testable Code for APCS A/AB

Bug.java

```
package info.gridworld.actor;

import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A Bug is an actor that can move and turn. It drops flowers as it moves.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Bug extends Actor
{
  /**
   * Constructs a red bug.
   */
  public Bug()
  {
    setColor(Color.RED);
  }

  /**
   * Constructs a bug of a given color.
   * @param bugColor the color for this bug
   */
  public Bug(Color bugColor)
  {
    setColor(bugColor);
  }

  /**
   * Moves if it can move, turns otherwise.
   */
  public void act()
  {
    if (canMove())
      move();
    else
      turn();
  }

  /**
   * Turns the bug 45 degrees to the right without changing its location.
   */
  public void turn()
  {
    setDirection(getDirection() + Location.HALF_RIGHT);
  }
```

```
  /**
   * Moves the bug forward, putting a flower into the location it previously occupied.
   */
  public void move()
  {
    Grid<Actor> gr = getGrid();
    if (gr == null)
      return;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (gr.isValid(next))
      moveTo(next);
    else
      removeSelfFromGrid();
    Flower flower = new Flower(getColor());
    flower.putSelfInGrid(gr, loc);
  }


  /**
   * Tests whether this bug can move forward into a location that is empty or contains a flower.
   * @return true if this bug can move.
   */
  public boolean canMove()
  {
    Grid<Actor> gr = getGrid();
    if (gr == null)
      return false;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (!gr.isValid(next))
      return false;
    Actor neighbor = gr.get(next);
    return (neighbor == null) || (neighbor instanceof Flower);
    // ok to move into empty location or onto flower
    // not ok to move onto any other actor
  }
}
```

BoxBug.java

```
import info.gridworld.actor.Bug;

/**
 * A BoxBug traces out a square "box" of a given size.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class BoxBug extends Bug
{
  private int steps;
  private int sideLength;


  /**
   * Constructs a box bug that traces a square of a given side length
   * @param length the side length
   */
  public BoxBug(int length)
  {
    steps = 0;
    sideLength = length;
  }


    /**
     * Moves to the next location of the square.
     */
  public void act()
  {
    if (steps < sideLength && canMove())
    {
      move();
      steps++;
    }
    else
    {
      turn();
      turn();
      steps = 0;
    }
  }
}
```

Critter.java

```
package info.gridworld.actor;

import info.gridworld.grid.Location;
import java.util.ArrayList;

/**
 * A Critter is an actor that moves through its world, processing
 * other actors in some way and then moving to a new location.
 * Define your own critters by extending this class and overriding any methods of this class except for act.
 * When you override these methods, be sure to preserve the postconditions.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Critter extends Actor
{
  /**
   * A critter acts by getting a list of other actors, processing that list, getting locations to move to,
   * selecting one of them, and moving to the selected location.
   */
  public void act()
  {
    if (getGrid() == null)
      return;
    ArrayList<Actor> actors = getActors();
    processActors(actors);
    ArrayList<Location> moveLocs = getMoveLocations();
    Location loc = selectMoveLocation(moveLocs);
    makeMove(loc);
  }


  /**
   * Gets the actors for processing. Implemented to return the actors that occupy neighboring grid locations.
   * Override this method in subclasses to look elsewhere for actors to process.
   * Postcondition: The state of all actors is unchanged.
   * @return a list of actors that this critter wishes to process.
   */
  public ArrayList<Actor> getActors()
  {
    return getGrid().getNeighbors(getLocation());
  }
```

```
/**
 * Processes the elements of actors. New actors may be added to empty locations.
 * Implemented to "eat" (i.e., remove) selected actors that are not rocks or critters.
 * Override this method in subclasses to process actors in a different way.
 * Postcondition: (1) The state of all actors in the grid other than this critter and the
 * elements of actors is unchanged. (2) The location of this critter is unchanged.
 * @param actors the actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
  for (Actor a : actors)
  {
    if (!(a instanceof Rock) && !(a instanceof Critter))
      a.removeSelfFromGrid();
  }
}


/**
 * Gets a list of possible locations for the next move. These locations must be valid in the grid of this critter.
 * Implemented to return the empty neighboring locations. Override this method in subclasses to look
 * elsewhere for move locations.
 * Postcondition: The state of all actors is unchanged.
 * @return a list of possible locations for the next move
 */
public ArrayList<Location> getMoveLocations()
{
  return getGrid().getEmptyAdjacentLocations(getLocation());
}


/**
 * Selects the location for the next move. Implemented to randomly pick one of the possible locations,
 * or to return the current location if locs has size 0. Override this method in subclasses that
 * have another mechanism for selecting the next move location.
 * Postcondition: (1) The returned location is an element of locs, this critter's current location, or null.
 * (2) The state of all actors is unchanged.
 * @param locs the possible locations for the next move
 * @return the location that was selected for the next move.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
{
  int n = locs.size();
  if (n == 0)
    return getLocation();
  int r = (int) (Math.random()  * n);
  return locs.get(r);
}
```

```
    /**
     * Moves this critter to the given location loc, or removes this critter from its grid if loc is null.
     * An actor may be added to the old location. If there is a different actor at location loc, that actor is
     * removed from the grid. Override this method in subclasses that want to carry out other actions
     * (for example, turning this critter or adding an occupant in its previous location).
     * 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)
    {
        if (loc == null)
            removeSelfFromGrid();
        else
            moveTo(loc);
    }
}
```

ChameleonCritter.java

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;

import java.util.ArrayList;

/**
 * A ChameleonCritter takes on the color of neighboring actors as it moves through the grid.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class ChameleonCritter extends Critter
{
    /**
     * Randomly selects a neighbor and changes this critter's color to be the same as that neighbor's.
     * If there are no neighbors, no action is taken.
     */
    public void processActors(ArrayList<Actor> actors)
    {
        int n = actors.size();
        if (n == 0)
            return;
        int r = (int) (Math.random()  * n);

        Actor other = actors.get(r);
        setColor(other.getColor());
    }

    /**
     * Turns towards the new location as it moves.
     */
    public void makeMove(Location loc)
    {
        setDirection(getLocation().getDirectionToward(loc));
        super.makeMove(loc);
    }
}
```

Appendix D — Testable Code for APCS AB

AbstractGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

/**
 * AbstractGrid contains the methods that are common to grid implementations.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public abstract class AbstractGrid<E> implements Grid<E>
{

  public ArrayList<E> getNeighbors(Location loc)
  {
    ArrayList<E> neighbors = new ArrayList<E>();
    for (Location neighborLoc : getOccupiedAdjacentLocations(loc))
      neighbors.add(get(neighborLoc));
    return neighbors;
  }


  public ArrayList<Location> getValidAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();

    int d = Location.NORTH;
    for (int i = 0; i < Location.FULL_CIRCLE / Location.HALF_RIGHT; i++)
    {
      Location neighborLoc = loc.getAdjacentLocation(d);
      if (isValid(neighborLoc))
        locs.add(neighborLoc);
      d = d + Location.HALF_RIGHT;
    }
    return locs;
  }


  public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    for (Location neighborLoc : getValidAdjacentLocations(loc))
    {
      if (get(neighborLoc) == null)
        locs.add(neighborLoc);
    }
    return locs;
  }
```

```
  public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    for (Location neighborLoc : getValidAdjacentLocations(loc))
    {
      if (get(neighborLoc) != null)
        locs.add(neighborLoc);
    }
    return locs;
  }


  /**
   * Creates a string that describes this grid.
   * @return a string with descriptions of all objects in this grid (not
   * necessarily in any particular order), in the format {loc=obj, loc=obj, ...}
   */
  public String toString()
  {
    String s = "{";
    for (Location loc : getOccupiedLocations())
    {
      if (s.length() > 1)
        s += ", ";
      s += loc + "=" + get(loc);
    }
    return s + "}";
  }
}
```

BoundedGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

/**
 * A BoundedGrid is a rectangular grid with a finite number of rows and columns.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public class BoundedGrid<E> extends AbstractGrid<E>
{
  private Object[][] occupantArray; // the array storing the grid elements


  /**
   * Constructs an empty bounded grid with the given dimensions.
   * (Precondition: rows > 0 and cols > 0.)
   * @param rows number of rows in BoundedGrid
   * @param cols number of columns in BoundedGrid
   */
  public BoundedGrid(int rows, int cols)
  {
    if (rows <= 0)
      throw new IllegalArgumentException("rows <= 0");
    if (cols <= 0)
      throw new IllegalArgumentException("cols <= 0");
    occupantArray = new Object[rows][cols];
  }


  public int getNumRows()
  {
    return occupantArray.length;
  }


  public int getNumCols()
  {
    // Note: according to the constructor precondition, numRows() > 0, so
    // theGrid[0] is non-null.
    return occupantArray[0].length;
  }


  public boolean isValid(Location loc)
  {
    return 0 <= loc.getRow() && loc.getRow() < getNumRows()
            && 0 <= loc.getCol() && loc.getCol() < getNumCols();
  }
```

```
  public ArrayList<Location> getOccupiedLocations()
  {
    ArrayList<Location> theLocations = new ArrayList<Location>();

    // Look at all grid locations.
    for (int r = 0; r < getNumRows(); r++)
    {
      for (int c = 0; c < getNumCols(); c++)
      {
        // If there's an object at this location, put it in the array.
        Location loc = new Location(r, c);
        if (get(loc) != null)
          theLocations.add(loc);
      }
    }

    return theLocations;
  }


  public E get(Location loc)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc + " is not valid");
    return (E) occupantArray[loc.getRow()][loc.getCol()]; // unavoidable warning
  }


  public E put(Location loc, E obj)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc+ " is not valid");
    if (obj == null)
      throw new NullPointerException("obj == null");

    // Add the object to the grid.
    E oldOccupant = get(loc);
    occupantArray[loc.getRow()][loc.getCol()] = obj;
    return oldOccupant;
  }


  public E remove(Location loc)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc + " is not valid");

    // Remove the object from the grid.
    E r = get(loc);
    occupantArray[loc.getRow()][loc.getCol()] = null;
    return r;
  }
}
```

UnboundedGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

import java.util.*;

/**
 * An UnboundedGrid is a rectangular grid with an unbounded number of rows and columns.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public class UnboundedGrid<E> extends AbstractGrid<E>
{
  private Map<Location, E> occupantMap;

  /**
   * Constructs an empty unbounded grid.
   */
  public UnboundedGrid()
  {
    occupantMap = new HashMap<Location, E>();
  }


  public int getNumRows()
  {
    return -1;
  }

  public int getNumCols()
  {
    return -1;
  }


  public boolean isValid(Location loc)
  {
    return true;
  }


  public ArrayList<Location> getOccupiedLocations()
  {
    ArrayList<Location> a = new ArrayList<Location>();
    for (Location loc : occupantMap.keySet())
      a.add(loc);
    return a;
  }


  public E get(Location loc)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    return occupantMap.get(loc);
  }
```

```
  public E put(Location loc, E obj)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    if (obj == null)
      throw new NullPointerException("obj == null");
    return occupantMap.put(loc, obj);
  }

  public E remove(Location loc)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    return occupantMap.remove(loc);
  }
}
```

Quick Reference A/AB

Location Class (implements Comparable)

```
public Location(int r, int c)
public int getRow()
public int getCol()
public Location getAdjacentLocation(int direction)
public int getDirectionToward(Location target)
public boolean equals(Object other)
public int hashCode()
public int compareTo(Object other)
public String toString()

NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST
LEFT, RIGHT, HALF_LEFT, HALF_RIGHT, FULL_CIRCLE, HALF_CIRCLE, AHEAD
```

Grid<E> Interface

```
int getNumRows()
int getNumCols()
boolean isValid(Location loc)
E put(Location loc, E obj)
E remove(Location loc)
E get(Location loc)
ArrayList<Location> getOccupiedLocations()
ArrayList<Location> getValidAdjacentLocations(Location loc)
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
ArrayList<E> getNeighbors(Location loc)
```

Actor Class

```
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
```

Rock Class (extends Actor)

```
public Rock()
public Rock(Color rockColor)
public void act()
```

Flower Class (extends Actor)

```
public Flower()
public Flower(Color initialColor)
public void act()
```

Bug Class (extends Actor)

```
public Bug()
public Bug(Color bugColor)
public void act()
public void turn()
public void move()
public boolean canMove()
```

BoxBug Class (extends Bug)

```
public BoxBug(int n)
public void act()
```

Critter Class (extends Actor)

```
public void act()
public ArrayList<Actor> getActors()
public void processActors(ArrayList<Actor> actors)
public ArrayList<Location> getMoveLocations()
public Location selectMoveLocation(ArrayList<Location> locs)
public void makeMove(Location loc)
```

ChameleonCritter Class (extends Critter)

```
public void processActors(ArrayList<Actor> actors)
public void makeMove(Location loc)
```

Quick Reference AB Only

AbstractGrid Class (implements Grid)

```
public ArrayList<E> getNeighbors(Location loc)
public ArrayList<Location> getValidAdjacentLocations(Location loc)
public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
public String toString()
```

BoundedGrid Class (extends AbstractGrid)

```
public BoundedGrid(int rows, int cols)
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```

UnboundedGrid Class (extends AbstractGrid)

```
public UnboundedGrid()
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```

Appendix G: Index for Source Code

This appendix provides an index for the Java source code found in Appendix C and Appendix D.

Bug.java

Method	Page
Bug()	**C1**
Bug(Color bugColor)	**C1**
act()	**C1**
turn()	**C1**
move()	**C2**
canMove()	**C2**

BoxBug.java

Method	Page
BoxBug(int length)	**C3**
act()	**C3**

Critter.java

Method	Page
act()	**C4**
getActors()	**C4**
processActors(ArrayList<Actor> actors)	**C5**
getMoveLocations()	**C5**
selectMoveLocation(ArrayList<Location> locs)	**C5**
makeMove(Location loc)	**C6**

ChameleonCritter.java

Method	Page
processActors(ArrayList<Actor> actors)	**C6**
makeMove(Location loc)	**C6**

AbstractGrid.java

`getNeighbors(Location loc)`	D1
`getValidAdjacentLocations(Location loc)`	D1
`getEmptyAdjacentLocations(Location loc)`	D1
`getOccupiedAdjacentLocations(Location loc)`	D2
`toString()`	D2

BoundedGrid.java

`BoundedGrid(int rows, int cols)`	D3
`getNumRows()`	D3
`getNumCols()`	D3
`isValid(Location loc)`	D3
`getOccupiedLocations()`	D4
`get(Location loc)`	D4
`put(Location loc, E obj)`	D4
`remove(Location loc)`	D4

UnboundedGrid.java

`UnboundedGrid()`	D5
`getNumRows()`	D5
`getNumCols()`	D5
`isValid(Location loc)`	D5
`getOccupiedLocations()`	D5
`get(Location loc)`	D5
`put(Location loc, E obj)`	D6
`remove(Location loc)`	D6

Name: ______________________________

AP® Computer Science AB
Student Answer Sheet for Multiple-Choice Section

No.	Answer
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	

No.	Answer
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	

AP® Computer Science AB
Multiple-Choice Answer Key

No.	Correct Answer
1	D
2	D
3	D
4	B
5	B
6	D
7	E
8	A
9	D
10	A
11	A
12	C
13	D
14	B
15	E
16	A
17	B
18	E
19	D
20	C
21	D
22	B
23	E
24	A
25	C
26	B
27	C
28	A
29	B
30	C

No.	Correct Answer
31	D
32	C
33	A
34	E
35	C
36	C
37	C
38	E
39	C
40	D

AP® Computer Science AB
Free-Response Scoring Guidelines

Question 1: Gradebook

Part A:	`inverseMap`	**4 1/2 points**

+1/2 create new map

+1/2 iterate over `scores` map for each key

+1/2 determine if score is in inverted map

+1/2 correctly create new set when needed

+1 add new set to inverse map under score key
- **+1/2** attempt
- **+1/2** correct

+1 add student name to appropriate set
- **+1/2** attempt
- **+1/2** correct

+1/2 return correct value

Part B:	`modeSet`	**4 1/2 points**

+1 call to `inverseMap`
- **+1/2** attempt
- **+1/2** correct

+3 iterate over inverse map for each key to find mode
- **+1/2** prime loop by setting initial value for largest set (and its size if saved separately)
- **+1** iterate over inverse map
  - **+1/2** attempt
  - **+1/2** correct
- **+1/2** compare sizes of sets to determine larger size
- **+1** update largest set when new max found (and its size if saved separately)
  - **+1/2** attempt
  - **+1/2** correct

+1/2 return largest set

AP® Computer Science AB
Free-Response Scoring Guidelines

Question 2: StringBag

Part A:	Big-Oh of `ListStringBag`	**1 point**

+1/2 Big-Oh of `ListStringBag add`

+1/2 Big-Oh of `ListStringBag getUniqueElements`

Part B:	`FastStringBag` class	**8 points**

+1/2 class header

+1/2 private instance variable declaration

+1 choice of data structure (such as `TreeMap`) to get specified $O(\log n)$ performance

+1/2 constructor initializes private data

+2 `add` method
- **+1** add new element with count 1 if first occurrence of value
  - **+1/2** attempt
  - **+1/2** correct
- **+1** increment count of existing element on subsequent occurrences of value
  - **+1/2** attempt
  - **+1/2** correct

+2 `getUniqueElements` method
- **+1/2** allocate array for result
- **+1/2** iterate over elements
- **+1/2** add key to proper index of array
- **+1/2** return array

+1 1/2 `getNumOccurrences` method
- **+1** return count for existing element
  - **+1/2** attempt
  - **+1/2** correct
- **+1/2** return 0 for nonexistent element

AP® Computer Science AB
Free-Response Scoring Guidelines

Question 3: TimidCritter (GridWorld)

Class and constructor declaration	**2 points**

+1/2 class header, `extends`

+1/2 declares private instance variables

+1/2 identifies data structure/logic that allows stack-like behavior (This may be a Stack, an array + an integer position, an array list, a linked list, or some other appropriate structure.)

+1/2 constructor initializes instance variables properly (Peek ahead at the next two methods to see how any integer or boolean instance variables are used.)

`getMoveLocations`	**3 points**

+1/2 has logic that distinguishes two cases (retracing and not)
(**Note:** If the logic is faulty because the student uses a mechanism that cannot capture the retrace state, such as the stack size, award this 1/2 point but penalize the student in `makeMove`.)

+1/2 when not retracing, just invokes `super`

When retracing:
+1/2 creates new empty list and uses it as method's return value

+1/2 populates list with a location from data structure (see `makeMove` for determining the correct location)

+1/2 `getMoveLocations` gets the next location in reverse order (This could be a `peek` or a `get` or an array access.)

+1/2 `getMoveLocations` does not mutate the data structure

Question 3: TimidCritter (GridWorld) (continued)

`makeMove`	**4 points**

+1 always invokes `super` to make move

+1/2 has logic that distinguishes two cases (retracing and not) (Same comment as above.)

Normal move:

+1/2 determines the correct location (before move) for adding to the data structure

+1/2 adds the location to the data structure consistent with use in `getMoveLocations` (This is likely to be a `push`, but it could be an `add` operation, or an array insert + position update.)

+1/2 correct logic to turn on retracing when max steps have been taken
Note: A student who has no instance variable for differentiating between retracing and normal moves will lose this 1/2 point and the 1/2 point in the next section; the intent is a 1-point penalty for failing to track the retracing state.
Note: If the student has a step counter from which he or she deduces the state, as in sample solution 2, this 1/2 point will be awarded simply for incrementing the step counter. If the student messes up the arithmetic for recovering the state or index, make appropriate deductions there.)

When retracing:

+1/2 removes the location from the data structure (**Note:** Students lose this 1/2 point if they mutated in `getMoveLocations`; the intent is a 1-point penalty for mutating in `getMoveLocations`.)

+1/2 correct logic to turn off retracing when the retracing process is complete (Same note as before.)

Special Usage:

-1 overrides other methods in `TimidCritter` such that postconditions are violated

Question 4: Inventory (ListNode)

Part A:	`addItem`	**2 points**

+1/2 create a new `ListNode` object containing `newItem`

+1/2 attach current `front` to follow the new node

+1 update `front` to point to the new node that was added

Part B:	`removeUnavailableItems`	**7 points**

+2 correctly traverse and consider all nodes in the list
- **+1** attempt to visit all nodes
- **+1** correct

+1 correct test for whether node should be removed
- **+1/2** attempt to get value of node and call `isInStock`
- **+1/2** correct cast to `Item` and call to `isInStock`

+1 handle node(s) that should not be removed
- **+1/2** attempt
- **+1/2** correct (advances to next node without changing any links)

+3 remove node(s) that should be removed
- **+1** modify `front` iff item is first in list
  - **+1/2** attempt to assign to `front`
  - **+1/2** correct (must handle case of removing multiple nodes at front of list)
- **+1** link from previous node is set to current's next
  - **+1/2** attempt
  - **+1/2** correct
- **+1** previous node stays the same for next iteration
  - **+1/2** attempt
  - **+1/2** correct (does not skip any nodes)

Question 1: Gradebook

PART A:

```
private Map<Integer, Set<String>> inverseMap()
{
  Map<Integer,Set<String>> inv = new HashMap<Integer,Set<String>>();

  for (String name : scores.keySet())
  {
    Integer score = scores.get(name);
    if (!inv.containsKey(score))
      inv.put(score, new HashSet<String>());
    inv.get(score).add(name);
  }

  return inv;
}
```

PART B:

```
public Set<String> modeSet()
{
  Map<Integer, Set<String>> inverted = inverseMap();
  Set<String> largestSet = new HashSet<String>();

  for (Integer score : inverted.keySet())
  {
    Set<String> curSet = inverted.get(score);
    if (curSet.size() > largestSet.size())
      largestSet = curSet;
  }

  return largestSet;
}
```

AP® Computer Science AB
Free-Response Canonical Solutions

Question 2: StringBag

PART A:

	Average-case Big-Oh
`add`	$O(n)$
`getUniqueElements`	$O(n \log n)$

PART B:

```
public class FastStringBag implements StringBag
{
  private Map<String, Integer> entries;

  public FastStringBag()
  {  entries = new TreeMap<String, Integer>();  }

  public void add(String s)
  {
    if (entries.containsKey(s))
      entries.put(s, new Integer(entries.get(s).intValue() + 1));
    else
      entries.put(s, new Integer(1));
  }

  public String[] getUniqueElements()
  {
    String[] result = new String[entries.size()];
    int count = 0;

    for (String key : entries.keySet())
    {
      result[count] = key;
      count++;
    }

    return result;
  }

  public int getNumOccurrences(String element)
  {
    Integer count = entries.get(element);
    if (count == null)
      return 0;
    else
      return count.intValue();
  }
}
```

AP® Computer Science AB
Free-Response Canonical Solutions

Question 3: TimidCritter (GridWorld)

<u>Solution 1 using stack and boolean retrace indicator</u>

```
public class TimidCritter extends Critter
{
  private Stack<Location> moves;
  private int steps;
  private boolean retracing;

  public TimidCritter()
  {
    moves = new Stack<Location>();
    steps = 0;
    retracing = false;
  }


  public ArrayList<Location> getMoveLocations()
  {
    if (retracing)
    {
      ArrayList<Location> moveList = new ArrayList<Location>();
      moveList.add(moves.peek());
      return moveList;
    }
    else
      return super.getMoveLocations();
  }


  public void makeMove(Location loc)
  {
    if (retracing)
    {
      steps--;
      if (steps > 0)
        moves.pop();
      else
        retracing = false;
    }
    else
    {
      steps++;
      if (steps < 10)
        moves.push(getLocation());
      else
        retracing = true;
    }
    super.makeMove(loc);
  }
}
```

Question 3: TimidCritter (GridWorld) (continued)

Solution 2 using array and step count

```
public class TimidCritter extends Critter
{
  private Location[] visitedLocations;
  private int steps;


  public TimidCritter()
  {
    visitedLocations = new Location[10];
    steps = 0;
  }


  public ArrayList<Location> getMoveLocations()
  {
    if ((steps / 10) % 2 == 1)
    {
      ArrayList<Location> moveLocations = new ArrayList<Location>();
      moveLocations.add(visitedLocations[9 - steps % 10]);
      return moveLocations;
    }
    else
    {
      return super.getMoveLocations();
    }
  }


  public void makeMove(Location loc)
  {
    if ((steps / 10) % 2 == 0)
      visitedLocations[steps % 10] = getLocation();
    steps++;
    super.makeMove(loc);
  }
}
```

Question 3: TimidCritter (GridWorld) (continued)

Solution 3 using 2 stacks

```
public class TimidCritter extends Critter
{
  private Stack<Location> visitedLocations;
  private Stack<Location> reverseLocations;


  public TimidCritter()
  {
    visitedLocations = new Stack<Location>();
    reverseLocations = null;
  }


  public ArrayList<Location> getMoveLocations()
  {
    if (reverseLocations != null)
    {
      ArrayList<Location> moveLocations = new ArrayList<Location>();
      moveLocations.add(reverseLocations.peek());
      return moveLocations;
    }
    else
    {
      return super.getMoveLocations();
    }
  }


  public void makeMove(Location loc)
  {
    if (reverseLocations == null)
    {
      visitedLocations.push(getLocation());
      if (visitedLocations.size() == 10)   // ***
        reverseLocations = visitedLocations;
    }
    else
    {
      reverseLocations.pop();
      if (reverseLocations.isEmpty())
      {
        visitedLocations = reverseLocations;
        reverseLocations = null;
      }
    }
    super.makeMove(loc);
  }
}
```

`***` would lose 1/2 point for using `size` because AP subset doesn't include `Stack size` method

Question 4: Inventory (ListNode)

PART A:

```
public void addItem(Item newItem)
{
  front = new ListNode(newItem, front);
}
```

PART B:

Iterative solution

```
public void removeUnavailableItems()
{
  ListNode prev = null;
  ListNode cur = front;

  while (cur != null)
  {
    if (!((Item)cur.getValue()).isInStock())
    {
      if (prev != null)
        prev.setNext(cur.getNext());
      else
        front = cur.getNext();
    }
    else
    {
      prev = cur;
    }

    cur = cur.getNext();
  }
}
```

Question 4: Inventory (ListNode) (continued)

Recursive solution

```
public void removeUnavailableItems()
{
  front = removeHelper(front);
}


private ListNode removeHelper(ListNode node)
{
  if (node == null)
    return null;
  else
  {
    Item current = (Item) node.getValue();
    if (current.isInStock())
    {
      node.setNext(removeHelper(node.getNext()));
      return node;
    }
    else
      return removeHelper(node.getNext());
  }
}
```