AP® Computer Science AB
2006 Free-Response Questions

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.

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.
- Assume that the implementations of the interfaces for stacks, queues, and priority queues (pages A4-A5 in the Appendix) behave as specified.
- Assume that the implementation classes `ListNode` and `TreeNode` (page A3 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 thesaurus is a list of words in which each word is associated with a corresponding set of other words having the same meaning (synonyms).

   The following shows an example of words and synonyms that could be listed in a thesaurus.

Word	Synonym Set
excellent	{brilliant, great, outstanding, tremendous}
super	{excellent, fantastic, great, wonderful}
wonderful	{amazing, brilliant, fantastic, great}

   The thesaurus can be represented using a `Map` in which each key is a word, and the value associated with a particular word is a `Set` of synonyms. A partial definition of the `Thesaurus` class is shown below.

```
public class Thesaurus
{
  // stores (word, synonym set) pairs as (String, Set of Strings)
  private Map wordMap;


  public Thesaurus()
  {  wordMap = new HashMap();  }


  // adds syn to the set of synonyms associated with word
  // in this Thesaurus;
  // if word is not a key in this Thesaurus, adds an
  // entry with word as key and a set containing syn
  // as the associated value
  public void addSynonym(String word, String syn)
  {  /* to be implemented in part (a) */  }


  // removes the word syn from each synonym set in this Thesaurus;
  // returns a set of the words (keys) whose associated
  // synonym sets originally contained the word syn;
  // if syn was not contained in any synonym set, returns an empty set
  public Set removeSynonym(String syn)
  {  /* to be implemented in part (b) */  }


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

(a) Write the `Thesaurus` method `addSynonym`. If `word` is already a key in `wordMap`, `syn` is added to the set associated with `word`. Otherwise, a new entry is added to `wordMap` with `word` as the key and a set containing `syn` as the associated value.

For example, assume that the `Thesaurus` object `myThesaurus` has been declared and initialized with the values shown at the beginning of this question. The following code segment contains two calls to `addSynonym`.

```
myThesaurus.addSynonym("wonderful", "magnificent");
myThesaurus.addSynonym("awesome", "wonderful");
```

After the code segment has executed, the contents of `myThesaurus` are as shown below.

Word	Synonym Set
excellent	{brilliant, great, outstanding, tremendous}
super	{excellent, fantastic, great, wonderful}
wonderful	{amazing, brilliant, fantastic, great, magnificent}
awesome	{wonderful}

Complete method `addSynonym` below.

```
// adds syn to the set of synonyms associated with word
// in this Thesaurus;
// if word is not a key in this Thesaurus, adds an
// entry with word as key and a set containing syn
// as the associated value
public void addSynonym(String word, String syn)
```

(b) Write the `Thesaurus` method `removeSynonym`. The word `syn` should be removed from every synonym set in the thesaurus. Method `removeSynonym` returns a set (possibly empty) that contains the keys for which the associated synonym set was modified. No key is ever removed from the thesaurus, even if the synonym set associated with that key becomes empty.

For example, assume that the `Thesaurus` object `myThesaurus` has been declared and initialized with the following values.

Word	Synonym Set
excellent	{brilliant, great, outstanding, tremendous}
super	{excellent, fantastic, great, wonderful}
wonderful	{amazing, brilliant, fantastic, great, magnificent}
awesome	{wonderful}

Consider the following statement.

```
Set affectedWords = myThesaurus.removeSynonym("wonderful");
```

After the statement has executed, the contents of `myThesaurus` are as shown below, and the set `affectedWords` contains {super, awesome}. Note that the words "wonderful" and "awesome" still appear as keys in the thesaurus.

Word	Synonym Set
excellent	{brilliant, great, outstanding, tremendous}
super	{excellent, fantastic, great}
wonderful	{amazing, brilliant, fantastic, great, magnificent}
awesome	{}

Complete method `removeSynonym` below.

```
// removes the word syn from each synonym set in this Thesaurus;
// returns a set of the words (keys) whose associated
// synonym sets originally contained the word syn;
// if syn was not contained in any synonym set, returns an empty set
public Set removeSynonym(String syn)
```

2. A company sells products in the form of items, packs, and bundles as listed below.

   1) An item consists of a description and a price.
   2) A pack consists of a product and an integer quantity indicating how many of that product are included in the pack.
   3) A bundle consists of a list of products.

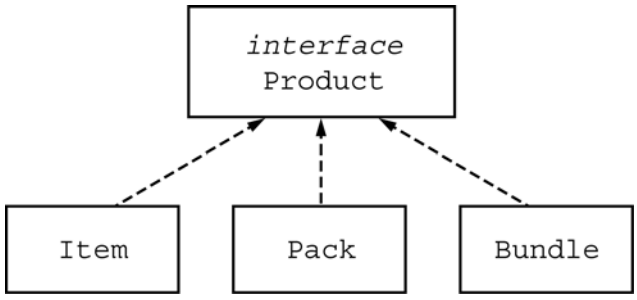
   Examples of each type of product are shown in the following table.

Product	Price	Product Type
ZapIt Microwave Oven	90.00	Item
NeverBurn Toaster	20.00	Item
4 x NeverBurn Toaster	80.00	Pack
Home Cooking Kit, containing ZapIt Microwave Oven NeverBurn Toaster	110.00	Bundle
Restaurant Starter Kit, containing Home Cooking Kit 4 x NeverBurn Toaster	190.00	Bundle
5 x Home Cooking Kit	550.00	Pack

The three types of products are represented by the three classes `Item`, `Pack`, and `Bundle`. All three classes implement the following `Product` interface.

```
public interface Product
{
  double getPrice();
}
```

The following diagram shows the relationship between the `Product` interface and the `Item`, `Pack`, and `Bundle` classes.



The definition of the `Item` class is as follows.

```
public class Item implements Product
{
  private String itemDescription; // description of this item
  private double unitPrice;       // price of this item, in dollars


  public Item(String description, double price)
  {
    itemDescription = description;
    unitPrice = price;
  }


  // returns the price of this item
  public double getPrice()
  {
    return unitPrice;
  }


  // sets the price of this item
  public void setPrice(double price)
  {
    unitPrice = price;
  }


  // returns the description of this item
  public String getDescription()
  {
    return itemDescription;
  }
}
```

In this problem, you will design and implement the `Pack` and `Bundle` classes.

(a) Write the `Pack` class that implements the `Product` interface. A `Pack` is used to represent multiple occurrences of a given product. Its constructor should have two parameters: the first represents the number of occurrences of a product in the pack, and the second represents the product.

The price of a `Pack` is the price of the product times the quantity. If the price of the product changes after the `Pack` has been constructed, the result of a call to `getPrice` for the `Pack` should reflect the updated product price.

The following code segment shows an example of how a `Pack` object can be declared and initialized.

```
Product toaster = new Item("NeverBurn Toaster", 20.0);
Product toasterPack = new Pack(4, toaster);
```

Write the `Pack` class below.

(b) Write the `Bundle` class that implements the `Product` interface. Its constructor should take no parameters and create a bundle that contains no products. There should be an `add` method that adds products to the bundle. It is possible for the same product to appear multiple times in a bundle. The price of the bundle is the sum of the prices of the products in the bundle. If the price of a product in a bundle changes after the `Bundle` has been constructed, the result of a call to `getPrice` for the `Bundle` should reflect the updated product price.

The following code segment creates instances of the products listed in the example at the beginning of this question.

```
Product oven = new Item("ZapIt Microwave Oven", 90.0);
Product toaster = new Item("NeverBurn Toaster", 20.0);
Product toasterPack = new Pack(4, toaster);

Bundle homeCookingKit = new Bundle();
homeCookingKit.add(oven);
homeCookingKit.add(toaster);

Bundle restaurantStarterKit = new Bundle();
restaurantStarterKit.add(homeCookingKit);
restaurantStarterKit.add(toasterPack);

Product homeKitPack = new Pack(5, homeCookingKit);
```

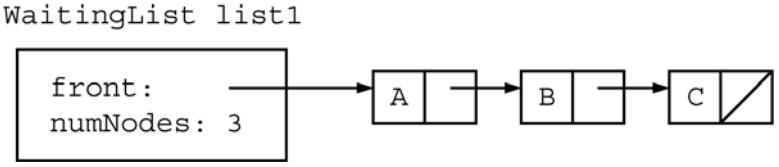
Write the `Bundle` class below.

3. Consider a class `WaitingList` that uses the `ListNode` class to manipulate linked lists. The definition of class `WaitingList` is as follows.

```
public class WaitingList
{
  private int numNodes;
  private ListNode front;

  // returns the number of nodes in this list
  public int size()
  {  return numNodes;  }


  // returns a reference to the node at index k,
  // where the indexes are numbered 0 through size()-1
  // precondition: 0 <= k < size()
  private ListNode getKthNode(int k)
  {  /* to be implemented in part (a) */  }


  // removes the last num nodes from other and attaches them
  // in the same order to the end of this WaitingList;
  // updates the number of nodes in each list to reflect the move
  // precondition:  size() > 0;
  //                0 < num <= other.size()
  public void transferNodesFromEnd(WaitingList other, int num)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `WaitingList` method `getKthNode`. This method should return a reference to the node at index `k`. Nodes in a `WaitingList` are indexed consecutively from the front, starting at 0.

In the diagram below, `list1.getKthNode(0)` returns a reference to the node containing `A`, `list1.getKthNode(1)` returns a reference to the node containing `B`, and `list1.getKthNode(2)` returns a reference to the node containing `C`.

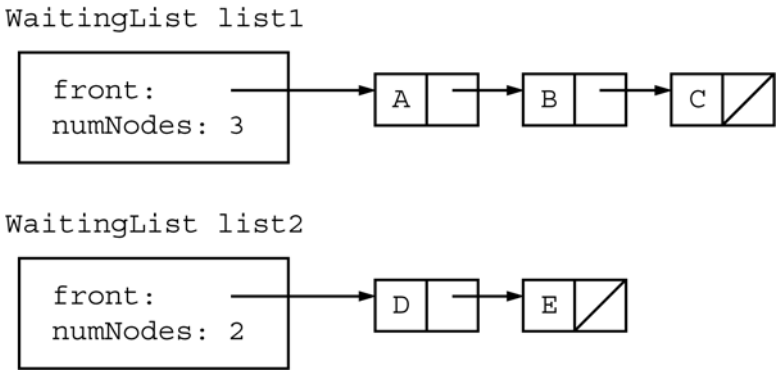


Complete method `getKthNode` below.

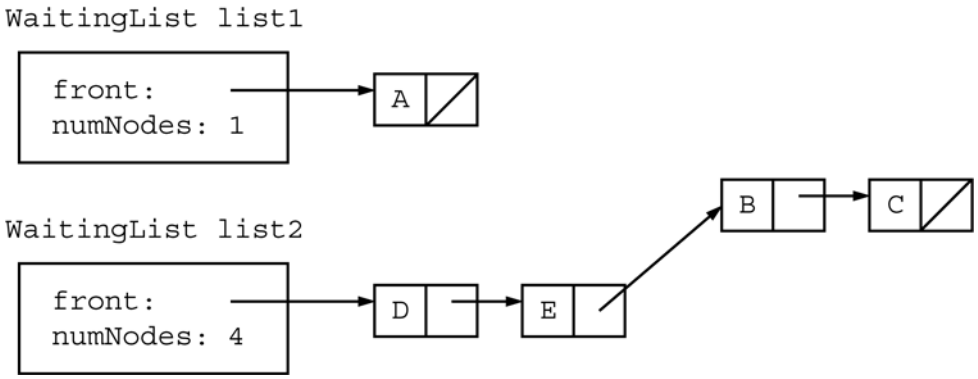
```
// returns a reference to the node at index k,
// where the indexes are numbered 0 through size()-1
// precondition: 0 <= k < size()
private ListNode getKthNode(int k)
```

(b) Write the `WaitingList` method `transferNodesFromEnd`. This method moves the last `num` nodes from the list `other` to the end of this list and updates the sizes of the two lists to reflect the results of the move. If `num` is equal to the size of `other`, the instance variable `front` in `other` will need to be updated as well.

For example, assume that two `WaitingList` objects have been defined and initialized as in the following diagram.



The result of the call `list2.transferNodesFromEnd(list1, 2)` would be as shown in the next diagram.



Note that it is not necessary to copy nodes. Solutions that allocate any new `ListNode` objects will not receive full credit.

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

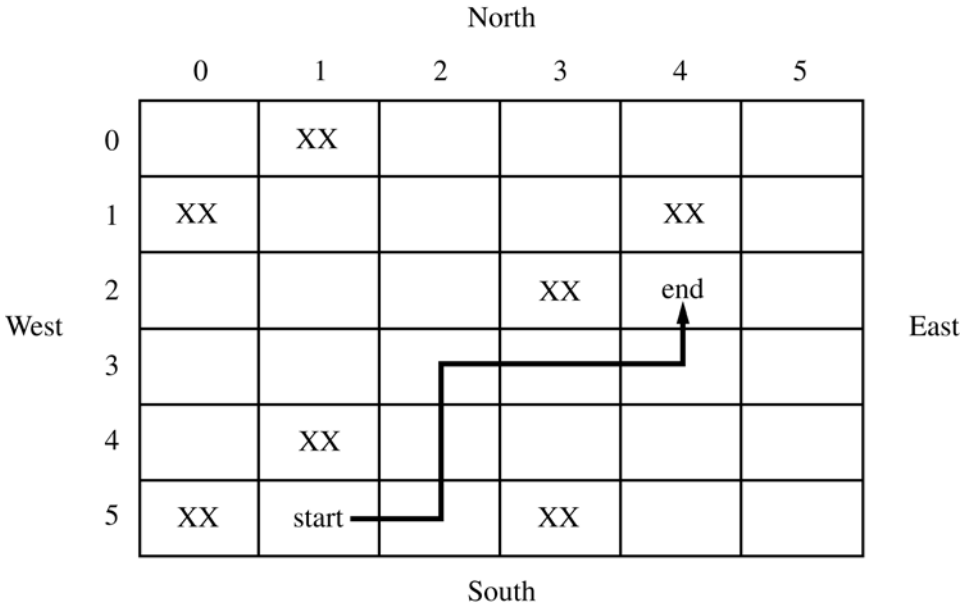
Complete method `transferNodesFromEnd` below.

```
// removes the last num nodes from other and attaches them
// in the same order to the end of this WaitingList;
// updates the number of nodes in each list to reflect the move
// precondition:  size() > 0;
//                0 < num <= other.size()
public void transferNodesFromEnd(WaitingList other, int num)
```

4. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

   Consider the problem of finding a particular kind of path, known as an NE-path, between two empty locations in an environment. An NE-path is an ordered list of empty locations in which each successive location must be either the north neighbor or the east neighbor of the previous location. For example, the diagram below shows an environment in which the occupied locations are indicated by XX. One possible NE-path between start location (5, 1) and end location (2, 4) as shown in the diagram is

```
[(5, 1), (5, 2), (4, 2), (3, 2), (3, 3), (3, 4), (2, 4)]
```



   Note that there may be more than one NE-path connecting two locations in an environment. In the above diagram, there are three distinct NE-paths that connect (5, 1) to (2, 4).

```
[(5, 1), (5, 2), (4, 2), (3, 2), (3, 3), (3, 4), (2, 4)]
[(5, 1), (5, 2), (4, 2), (4, 3), (3, 3), (3, 4), (2, 4)]
[(5, 1), (5, 2), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4)]
```

   Similarly, there may be locations that have no NE-paths connecting them. In the above diagram, (3, 3) and (0, 4) cannot be connected because all possible NE-paths are blocked by occupied locations. Likewise, (2, 4) cannot be the start of an NE-path to (5, 1) because (2, 4) is north and east of (5, 1).

An NE-path between a start location and an end location can be found recursively. For an NE-path between start and end locations to be possible, both locations must be empty and the start location must not be north or east of the end location. If the start and end locations are the same, then the NE-path will consist of that single location. Otherwise, an NE-path can be searched for by trying to recursively find an NE-path from the north neighbor of the start location to the end location. If such a path is not found, a similar recursive attempt is made from the east neighbor of the starting location. If either of these NE-paths exists, the start location can be added to the front of that NE-path to complete an NE-path from start to end.

For example, the path from (5, 1) to (2, 4) in the above diagram is found by first attempting to find an NE-path from (4, 1) to (2, 4). This attempt fails because the starting location, (4, 1), is not empty. Next, the attempt to recursively find an NE-path from (5, 2) to (2, 4) succeeds. Suppose it yields the following path.

```
[(5, 2), (4, 2), (3, 2), (3, 3), (3, 4), (2, 4)]
```

The start location (5, 1) is then added to the front of that path to yield the NE-path shown in the diagram.

The incomplete definition of the `PathFinder` class is given below. The class declares a private field `theEnv` to refer to the `Environment` being searched. You will implement two methods for the `PathFinder` class to complete the recursive path-finding algorithm.

```
public class PathFinder
{
  private Environment theEnv;


  // returns true if the specified locations are possible
  // starting and ending locations for an NE-path
  // otherwise, returns false
  private boolean possibleEnds(Location start, Location end)
  {  /* to be implemented in part (a) */  }


  // returns a list containing a sequence of empty locations
  // that form an NE-path from start to end;
  // returns null if no such path exists
  // postcondition: theEnv is unchanged
  public List findNEPath(Location start, Location end)
  {  /* to be implemented in part (b) */  }


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

(a) Write the `PathFinder` method `possibleEnds`, which returns `true` if the specified locations are possible end points for an NE-path. In order to be possible end points for an NE-path, both locations must be empty and the start location cannot be north or east of the end location.

Complete method `possibleEnds` below.

```
// returns true if the specified locations are possible
// starting and ending locations for an NE-path
// otherwise, returns false
private boolean possibleEnds(Location start, Location end)
```

(b) Write the `PathFinder` method `findNEPath`, which recursively searches the environment to find an NE-path between the locations `start` and `end`. If there is more than one NE-path connecting the locations, the method may return any one of those NE-paths. If there are no NE-paths connecting the locations, the method should return null.

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

Complete method `findNEPath` below.

```
// returns a list containing a sequence of empty locations
// that form an NE-path from start to end;
// returns null if no such path exists
// postcondition: theEnv is unchanged
public List findNEPath(Location start, Location end)
```

END OF EXAM