AP® Computer Science AB
2008 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.
- 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.

GO ON TO THE NEXT PAGE.

1. A *key string* for a word is the string obtained when the letters in the word are arranged in alphabetical order. For example, the words "poodle" and "looped" both have the key string "deloop". A set of words form an *anagram set* if all words in the set have the same key string. Some examples of anagram sets include {"poodle", "looped"}, {"nastier", "retains", "retinas"}, and {"discounter", "introduces", "reductions"}.

   Consider the problem of creating a group of anagram sets from a list of words. A word with no anagrams in the list will be contained in a singleton set. The example below shows a list of words and the anagram sets that are produced from that list.

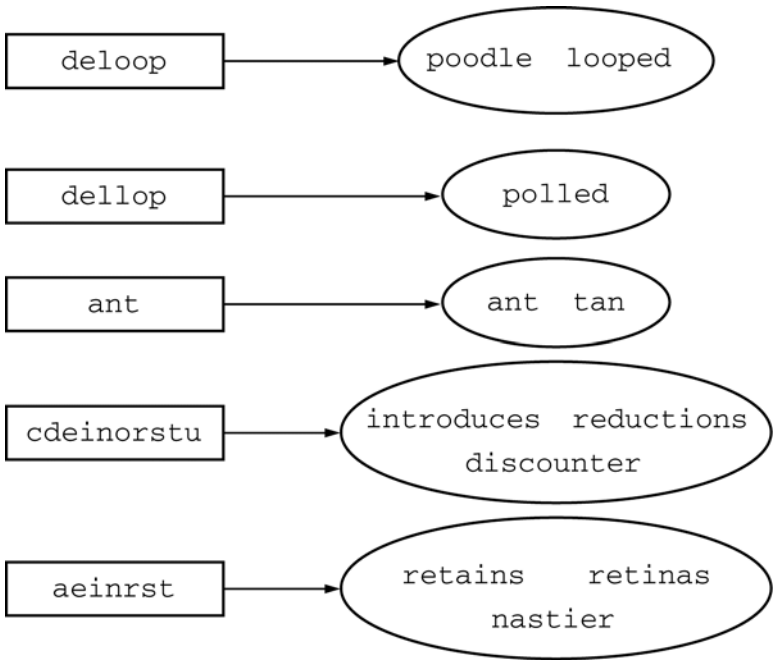
Original list of words

[ant, introduces, poodle, tan, looped, discounter, nastier, polled, retains, retinas, reductions]

Anagram sets from the list of words

{tan, ant}
{introduces, reductions, discounter}
{poodle, looped}
{retains, retinas, nastier}
{polled}

The anagram sets will be organized as a map in which each key is a key string and the associated value is the set of words that each have that key string. The following diagram shows a map with the key strings and associated anagram sets that would be created from the list in the previous example.



A partial declaration of the `AnagramGrouper` class is as follows: You will implement two methods in the `AnagramGrouper` class. The class contains a private helper method that can be used to create a key string from a word.

```
public class AnagramGrouper
{
  // Maps a key string to a corresponding anagram set
  private HashMap<String, HashSet<String>> groups;


  /** Constructs a map from words in which the keys are key strings and the
   *  value associated with a key string is the set of anagrams having that key string.
   *  Postcondition: each entry of words is contained in an anagram set
   *  @param words a list of strings to be grouped into anagram sets
   *         Precondition: words.size() > 0
   */
  public AnagramGrouper(List<String> words)
  {  /* to be implemented in part (a) */  }


  /** @return a set of all anagram sets of largest size in this AnagramGrouper
   */
  public HashSet<HashSet<String>> findLargestSets()
  {  /* to be implemented in part (b) */  }


  /** @param s a word
   *  @return a string with the same letters as s, arranged in alphabetical order
   */
  private String createKeyString(String s)
  {  /* implementation not shown */  }

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

(a) Write the `AnagramGrouper` constructor. The constructor takes a list of words and constructs a map in which each key is a key string and the associated value is the set of words that each have that key string. The map should contain all the anagram sets that are generated from the list of words.

Complete the `AnagramGrouper` constructor below.

```
/** Constructs a map from words in which the keys are key strings and the
 *  value associated with a key string is the set of anagrams having that key string.
 *  Postcondition: each entry of words is contained in an anagram set
 *  @param words a list of strings to be grouped into anagram sets
 *         Precondition: words.size() > 0
 */
public AnagramGrouper(List<String> words)
```

(b) Write the `AnagramGrouper` method `findLargestSets`. This method analyzes the instance variable `groups` and returns a set containing the largest anagram set(s); that is, the set(s) with the most elements. In the example shown at the beginning of the question, the method would return a set containing the sets {introduces, reductions, discounter} and {retains, retinas, nastier}.

Complete method `findLargestSets` below.

```
/** @return a set of all anagram sets of largest size in this AnagramGrouper
 */
public HashSet<HashSet<String>> findLargestSets()
```

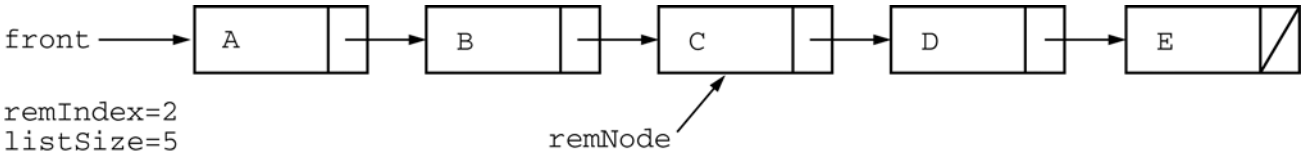
2. Suppose `myList` is a linked list. A loop such as

```
for (int k = 0; k < myList.size(); k++)
  Object a = myList.get(k);
```

can be inefficient if the `get` method always starts at the front of the list to locate each element.

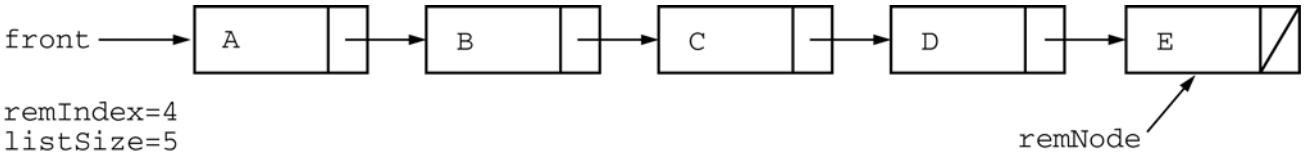
This question describes a variant of a linked list data structure that can improve the efficiency of the loop. The `ListNode` class is used as part of the implementation of a class `APList` that remembers the node and index of the most recently visited element of the list. If the index for the next `get` is greater than or equal to the remembered index, the traversal starts at the remembered node instead of the front of the list.

For example, consider the following `APList myList` that contains five elements. The call `myList.get(2)` returns the value C. The remembered node and remembered index will refer to the node at index 2 as shown in the diagram.

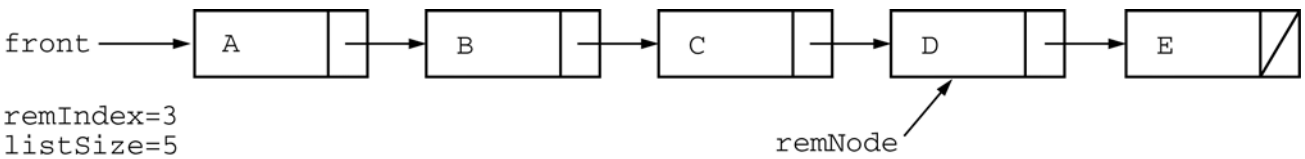


If the call `myList.get(2)` is executed again, the value C is returned again, but the traversal begins at the `remNode` position, instead of `front`, because the remembered index is 2. No other nodes are traversed because `remNode` is already at the correct position.

A subsequent call `myList.get(4)` will start the traversal at the `remNode` position instead of `front` because the remembered index is 2. It will then move forward two nodes to retrieve the node at index 4 (the value E), set `remNode` to the new position, and update `remIndex` as shown in the following diagram.



Finally, a subsequent call `myList.get(3)` will start the traversal at `front` because the remembered index is greater than the desired index. Starting at `front`, it will move forward three nodes to retrieve the node at index 3 (the value D), set `remNode` to the new position, and update `remIndex` as shown in the following diagram.



The partial declaration of the `APList` class is shown below.

```
public class APList
{
  private ListNode front;    // first node of this list (null if empty)

  private int listSize;      // the number of elements in this list

  private int remIndex;      // the index of the remembered node

  private ListNode remNode; // a reference to the node accessed by most recent call to get


  /** Constructs an empty APList.
   */
  public APList()
  {
    front = null;
    remIndex = -1;
    remNode = null;
    listSize = 0;
  }


  /** Gets a value at a given index in this list.
   *  @param n the index at which to get a value
   *           Precondition: 0 ≤ n < size()
   *  @return the object at the given index
   *  Postcondition: The remembered node and index refer to the node at index n
   */
  public Object get(int n)
  {  /* to be implemented in part (a) */ }


  /** Adds a new node containing obj to the front of this list.
   *  @param obj the value to add to the list
   */
  public void addFirst(Object obj)
  {  /* to be implemented in part (b) */  }


  /** @return the size of this list
   */
  public int size()
  {  return listSize;  }

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

(a) Write the `APList` method `get`. This method returns the value contained in the list node at index `n`. If the index `n` is greater than or equal to the remembered index, the method should start its traversal at the remembered node; otherwise, the method should start at the front of the list. The remembered node and index should be updated to refer to the node at the given index.

Complete method `get` below.

```
/** Gets a value at a given index in this list.
 *  @param n the index at which to get a value
 *           Precondition: 0 ≤ n < size()
 *  @return the object at the given index
 *  Postcondition: The remembered node and index refer to the node at index n
 */
public Object get(int n)
```

(b) Write the `APList` method `addFirst`. Instance variables should be updated as necessary. This method should not change the value of `remNode` since there is no advantage to remembering a node at the front of the list.

Complete method `addFirst` below.

```
/** Adds a new node containing obj to the front of this list.
 *  @param obj the value to add to the list
 */
public void addFirst(Object obj)
```

(c) Consider the following methods.

```
public static void printForward(SomeListType myList)
{
  int n = myList.size();
  for (int k = 0; k < n; k++)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}


public static void printReverse(SomeListType myList)
{
  int n = myList.size();
  for (int k = n - 1; k >= 0; k--)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}
```

Give the big-Oh running time (in terms of n) of these methods for the following list types, where n is the number of elements in the list.

SomeListType	printForward	printReverse
LinkedList<Object>		
APList		

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

   Consider the implementation of a *multigrid*, a gridlike structure in which each location can hold multiple occupants. Instead of storing a single occupant, a grid location stores a set of occupants. An empty location is always indicated by `null`, not an empty set.

   A partial declaration of the `Multigrid` interface is as follows. Four of the `Grid` methods require changes in this interface.

```
public interface Multigrid
{
  Set<Object> get(Location loc);

  void put(Location loc, Object obj);

  ArrayList<Object> getNeighbors(Location loc);

  void remove(Location loc, Object obj);

  // other methods identical to the Grid interface
}
```

A partial declaration of the `UnboundedMultigrid` class is as follows. You will complete three of the methods in the `UnboundedMultigrid` class.

```
public class UnboundedMultigrid implements Multigrid
{
  private UnboundedGrid<Set<Object>> grid;   // no other instance variables


  public UnboundedMultigrid()
  {  grid = new UnboundedGrid<Set<Object>>();  }


  /** @param loc a valid location in this grid
   *  @return a set of all objects at loc; an empty set, if no objects at loc
   *  Postcondition: the contents of this grid remain unchanged
   */
  public Set<Object> get(Location loc)
  {  /* to be implemented in part (a) */  }


  /** Puts an object at a given location in this grid.
   *  Precondition: (1) loc is valid in this grid. (2) obj is not null.
   *  @param loc the location at which to put the object
   *  @param obj the new object to be added
   */
  public void put(Location loc, Object obj)
  {  /* to be implemented in part (b) */  }


  /** Gets the neighboring occupants in all eight compass directions
   *  (north, northeast, east, southeast, south, southwest, west, and northwest).
   *  @param loc a location in this grid
   *         Precondition: loc is valid in this grid
   *  @return an array list of the objects in the occupied locations adjacent to loc in this grid
   */
  public ArrayList<Object> getNeighbors(Location loc)
  {  /* to be implemented in part (c) */  }

  // other methods not shown
}
```

(a) Complete the `UnboundedMultigrid` method `get` below.

```
/** @param loc a valid location in this grid
 *  @return a set of all objects at loc; an empty set, if no objects at loc
 *  Postcondition: the contents of this grid remain unchanged
 */
public Set<Object> get(Location loc)
```

(b) Assume that the `UnboundedMultigrid` method `get` works as specified, regardless of what you wrote in part (a).

Complete the `UnboundedMultigrid` method `put` below.

```
/** Puts an object at a given location in this grid.
 *  Precondition: (1) loc is valid in this grid. (2) obj is not null.
 *  @param loc the location at which to put the object
 *  @param obj the new object to be added
 */
public void put(Location loc, Object obj)
```

(c) Assume that the `UnboundedMultigrid` methods `get` and `put` work as specified, regardless of what you wrote in parts (a) and (b).

Complete the `UnboundedMultigrid` method `getNeighbors` below.

```
/** Gets the neighboring occupants in all eight compass directions
 *  (north, northeast, east, southeast, south, southwest, west, and northwest).
 *  @param loc a location in this grid
 *         Precondition: loc is valid in this grid
 *  @return an array list of the objects in the occupied locations adjacent to loc in this grid
 */
public ArrayList<Object> getNeighbors(Location loc)
```

4. A *filter* is an object that examines strings and accepts those that meet a particular criterion.

The `Filter` interface is defined as follows.

```
public interface Filter
{
   /** @param text a string to consider for acceptance
    *  @return true if this Filter accepts text; false otherwise
    */
   boolean accept(String text);
}
```

In this question, you will implement one type of `Filter` and write a method to build a variety of filters. Many different filters can implement the `Filter` interface. The `SimpleFilter` and `NotFilter` are both implementations of the `Filter` interface and are described as follows:

The `SimpleFilter` class accepts any string that contains a particular substring. It has one constructor that takes a single parameter that contains the string to be found in the text. The `NotFilter` is constructed with a single `Filter` parameter and accepts text if and only if the `Filter` it was constructed with rejects the text.

For example, the following code segment creates a filter `veggieFilter` that accepts all strings that contain the substring `"vegetable"` and another filter `noVeggie` that accepts all strings that do NOT contain the substring `"vegetable"`.

```
Filter veggieFilter = new SimpleFilter("vegetable");
Filter noVeggie = new NotFilter(veggieFilter);
```

The following table illustrates the results of several calls to the `veggieFilter accept` method and the `noVeggie accept` method.

Method Call	Result
`veggieFilter.accept("vegetable soup")`	`true`
`veggieFilter.accept("fruit salad")`	`false`
`noVeggie.accept("vegetable soup")`	`false`
`noVeggie.accept("fruit salad")`	`true`

(a) An `OrFilter` is a `Filter` that contains two or more objects that implement the `Filter` interface (such as a `SimpleFilter`, an `OrFilter`, a `NotFilter`, or other types of `Filter` objects). The `OrFilter accept` method accepts a given string if and only if one or more of the filters contained in the `OrFilter` accepts the string. An `OrFilter` is constructed with two filters. More filters can be included in the `OrFilter` by calling the `add` method. The following code segment illustrates a filter that accepts all strings that contain either one or both of the substrings `"vegetable"` and `"fruit"` or that do not contain the string `"poison"`.

```
Filter veggieFilter = new SimpleFilter("vegetable");
Filter fruitFilter = new SimpleFilter("fruit");
Filter noPoison = new NotFilter(new SimpleFilter("poison"));

OrFilter healthyFood = new OrFilter(veggieFilter, fruitFilter);
  //  healthyFood will accept strings containing "vegetable" or "fruit"

healthyFood.add(noPoison);
  //  healthyFood will accept strings containing "vegetable" or "fruit"
  //  or strings without "poison"
```

The following table illustrates the results of several calls to the `healthyFood accept` method.

Method Call	Result
`healthyFood.accept("vegetable soup is not poison")`	`true`
`healthyFood.accept("fruit salad")`	`true`
`healthyFood.accept("a poisoned apple")`	`false`
`healthyFood.accept("salad")`	`true`

Write the `OrFilter` class below.

(b) Write the `buildFilter` method that can be used to build filters that meet a set of criteria as described below. The `buildFilter` method is part of an unrelated class.

You may assume the existence of an `AndFilter` class that is similar to the `OrFilter` class except that it accepts a string only if all of its component filters accept the string. Consider the following code segment.

```
Filter appleFilter = new SimpleFilter("apple");
Filter peachFilter = new SimpleFilter("peach");
Filter fruits = new AndFilter(appleFilter, peachFilter);

boolean b1 = fruits.accept("peach cobbler");      // b1 is set to false
boolean b2 = fruits.accept("peaches and apples"); // b2 is set to true
```

The method `buildFilter` takes two parameters: `desirable`, a list of two or more strings, and `notAllowed`, a string. The filter returned by `buildFilter` should accept all strings that contain at least one string from the `desirable` array and do not contain the `notAllowed` string.

The following code segment and table show an example of using the `buildFilter` method.

```
String[] primary = {"red", "blue", "yellow"};
Filter primaryColors = buildFilter(primary, "green");
```

Method Call	Result
`primaryColors.accept("Roses are red, violets are blue")`	`true`
`primaryColors.accept("blue grass is really green")`	`false`
`primaryColors.accept("The rainbow has many colors")`	`false`

In writing `buildFilter`, you may use any of the four filters described (`SimpleFilter`, `NotFilter`, `OrFilter`, and `AndFilter`). Assume that these classes work as specified, regardless of what you wrote in part (a).

Complete method `buildFilter` below.

```
/** @param desirable contains strings that are allowed
 *         Precondition: desirable.length > 1
 *  @param notAllowed the string that is not allowed
 *  @return a Filter that accepts strings that contain at least one string
 *          in desirable and do not contain notAllowed.
 */
public static Filter buildFilter(String[] desirable,
                                 String notAllowed)
```

STOP

END OF EXAM