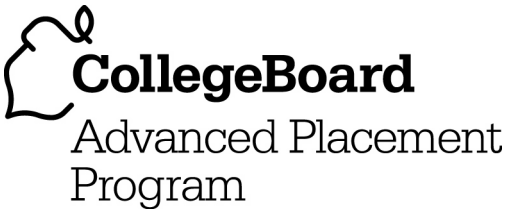



# AP® Computer Science AB
# 2004 Free-Response Questions

**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.
- 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.

1. Consider designing and implementing a set of classes and interfaces to represent books, library items, and library books.

   (a) A library item contains two pieces of information: a unique ID (a `String`) and a holder (also a `String`). If the library item has been checked out, the holder is the name of the person who has checked out this item, otherwise the holder is `null`. The ID of a library item cannot change after it is created, but the holder can change. Write the `LibraryItem` interface that abstracts this functionality.

   (b) A book consists of an author and a title, neither of which can change once the book is created. The `Book` class is represented as follows.

```
public class Book
{
  private String theAuthor;
  private String theTitle;

  public Book(String author, String title)
  { theAuthor = author;  theTitle = title; }

  public String getAuthor()
  { return theAuthor; }

  public String getTitle()
  { return theTitle; }

}
```

   A library book is a book that is also a library item. Write the complete class `LibraryBook`, implementing the required methods.

(c) A library is a collection of library items and supports the following operations.

1. Add a library item to the library.
2. Check out a library item by specifying its ID and new holder.
3. Determine the current holder of a library item, given its ID.

Consider the following incomplete class declaration.

```
public class Library
{
  private SomeClass items;

  public Library()
  {  /* implementation not shown */  }

  public void add(LibraryItem theItem)
  {  /* implementation not shown */  }

  public void checkout(String id, String holder)
  {  /* implementation not shown */  }

  public String getHolder(String id)
  {  /* implementation not shown */  }
}
```

Choose a data representation for `SomeClass` from the `java.util` class library that allows the operations `add`, `checkout`, and `getHolder` to be performed efficiently. In doing so, you must explain how you are using your chosen `java.util` class to organize your data and give the expected Big-Oh running time for each of these three operations in terms of `n`, the number of items in the library.

| Your data representation (Circle one) | How data is organized |
|---|---|
| ArrayList LinkedList<br>HashMap TreeMap<br>HashSet TreeSet | |

| Operations | Expected Big-Oh efficiency |
|---|---|
| add | |
| checkout | |
| getHolder | |

2. In approval voting there are at least two candidates and there are many voters. Each voter votes by submitting a ballot that represents a set of candidates of whom the voter approves. The winner of the election is the candidate that is listed on the most ballots, that is, the candidate who is approved by the most voters.

   For example, suppose there are four candidates: Chris, Jamie, Pat, and Sandy. Seven voters vote as follows (order is unimportant, each ballot represents the set of candidates chosen by that voter).

| Voter | Ballot |
|---|---|
| 0 | Chris, Jamie |
| 1 | Chris, Sandy |
| 2 | Chris, Sandy, Pat, Jamie |
| 3 | Pat |
| 4 | Sandy, Jamie |
| 5 | Sandy, Pat, Jamie |
| 6 | Jamie, Chris |

   The number of ballots on which each candidate was listed is as follows.

| Candidate | Number of ballots |
|---|---|
| Chris | 4 |
| Jamie | 5 |
| Pat | 3 |
| Sandy | 4 |

   In this example, Jamie is the winner. If more than one candidate is tied with the most votes, then we consider the set of such candidates to be the winning set for this election. For example, if Chris appeared on voter 3's ballot along with Pat, then Chris would tie Jamie with five votes and the winning set would be {Chris, Jamie}.

We represent one voter's ballot as a `Set` that contains the names (as `Strings`) of the candidates approved by that voter. We represent the set of ballots for all voters in the election by the following partial class declaration.

```
public class VoterBallots
{
  private Map voteCount;  // key is the candidate name, value is the
                          // number of votes received by that candidate

  // precondition:  each entry in ballotList is a Set representing
  //                one voter's ballot
  // postcondition: voteCount.get(candidate) is the total number of
  //                times candidate appears on ballots in ballotList
  public VoterBallots(List ballotList)
  {  /* to be implemented in part (a) */  }

  // postcondition: returns an Integer object with value equal to the
  //                maximum number of votes associated with any
  //                key in the Map voteCount
  private Integer maxVotes()
  {  /* implementation not shown */  }

  // postcondition: returns a set containing the candidate(s)
  //                with the most votes
  public Set candidatesWithMost()
  {  /* to be implemented in part (b) */  }

  // other methods not shown
}
```

(a) Write the `VoterBallots` constructor. The constructor creates the `Map voteCount` and initializes it to hold one entry for each candidate that appears on at least one ballot in `ballotList`. The associated value for each candidate is the total number of ballots on which that candidate appears.

A solution that creates an unnecessary new instance of a `List` or `Set` will not receive full credit.

Complete the constructor below.

```
// precondition:  each entry in ballotList is a Set representing
//                one voter's ballot
// postcondition: voteCount.get(candidate) is the total number of
//                times candidate appears on ballots in ballotList
public VoterBallots(List ballotList)
```

(b) Write the `VoterBallots` method `candidatesWithMost`. Method `candidatesWithMost` returns a `Set` of the name(s) of the candidate(s) in `voteCount` with the most votes.

In writing method `candidatesWithMost`, you may call the private helper method `maxVotes`. Solutions that reimplement functionality provided by `maxVotes`, rather than invoking `maxVotes`, will not receive full credit. A solution that creates an unnecessary new instance of a `List` or `Set` will not receive full credit.

Complete method `candidatesWithMost` below.

```
// postcondition: returns a set containing the candidate(s)
//                with the most votes
public Set candidatesWithMost()
```

(c) Assume that there are $C$ candidates and $V$ voters. What is the expected time complexity, in Big-Oh notation in terms of $C$ and $V$, for your implementation of method `candidatesWithMost`?

3. 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 defining a new type of fish, called `PredatorFish`, that eats other fish. At the beginning of a simulation step, a `PredatorFish` examines the environment location directly in front of it. If a fish of any kind is present, the `PredatorFish` eats that fish, removing it from the environment. Otherwise, if this is the fifth consecutive turn in which the `PredatorFish` has not eaten, it dies. After the attempt to eat, if the `PredatorFish` is still alive, it takes an ordinary turn, acting as an ordinary `Fish` in terms of breeding, moving, and dying.

   A `PredatorFish` can be defined by inheriting behavior from the `Fish` class and adding or overriding methods as appropriate.

   (a) Write a partial class declaration for `PredatorFish` that includes the heading for the class, any instance variables that must be declared in the class, and a two-parameter constructor that takes the environment and the initial location of the `PredatorFish` as parameters. Do not show other constructors and methods.

   (b) Write the new `eat` method for the `PredatorFish` class. This method will remove a `Fish` that is directly in front of the `PredatorFish` if that environment location is not empty. The `PredatorFish` remains in the same location. The `boolean` result that is returned from the `eat` method indicates whether the `PredatorFish` was able to eat.

   Complete the `PredatorFish` method `eat` below.

```
// if the PredatorFish is able to eat, the eaten fish is removed and
// true is returned; otherwise, false is returned
protected boolean eat()
```

   (c) One way to implement `PredatorFish` is to override the `act` method. A `PredatorFish` will first eat if possible. If it is the fifth consecutive turn in which the `PredatorFish` has not eaten, the `PredatorFish` dies. Otherwise, it acts as an ordinary `Fish` in terms of breeding, moving, and dying.

   You may use any of the accessible methods of the `PredatorFish` class and other public classes from the case study program. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that the `eat` method works correctly, regardless of what you wrote in part (b).

   Complete the `PredatorFish` method `act` below.

```
// acts for one step in the simulation
public void act()
```

4. In some applications of priority queues, each data value can be added to the priority queue multiple times. In this case, time and space can be saved by storing each data value only once, with a count field to keep track of the number of times that the data value is in the priority queue.

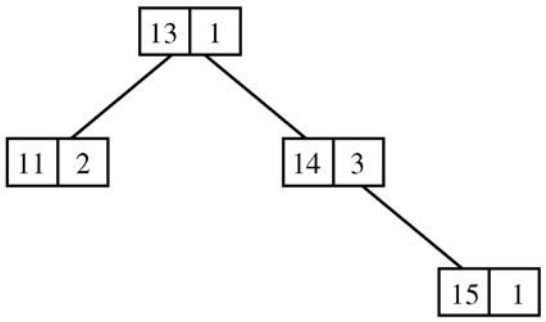
   Class `TreePriorityQueue` is implemented using a binary search tree. Each `TreeNode` of the binary search tree consists of an `Item` object and the links to the left and right subtrees. An `Item` object contains a `Comparable` data value and a count of the number of occurrences of the data. When the count goes to zero as a result of a deletion, the `Item` object is removed from the tree.

   The declarations for the `TreePriorityQueue` class and the `Item` class are shown below.

```
public class TreePriorityQueue implements PriorityQueue
{
  private TreeNode root;


  public TreePriorityQueue()
  { root = null; }


  //postcondition: returns true if the priority queue is empty;
  //               otherwise, returns false
  public boolean isEmpty()
  {  /* implementation not shown */  }


  // postcondition: obj has been added to the priority queue
  public void add(Object obj)
  { root = addHelper((Comparable)obj, root); }


  // postcondition: obj has been added to the subtree rooted at t;
  //                the updated subtree is returned
  private TreeNode addHelper(Comparable obj, TreeNode t)
  {  /* to be implemented in part (b) */  }


  // precondition:  the priority queue is not empty
  // postcondition: a data value of smallest priority has been
  //                removed and returned
  public Object removeMin()
  {  /* implementation not shown */  }


  // precondition:  the priority queue is not empty
  // postcondition: a data value of smallest priority is returned;
  //                the priority queue is unchanged
  public Object peekMin()
  {  /* to be implemented in part (a) */  }
}
```

```
public class Item
{
  private Comparable data;
  private int count;

  public Item(Comparable d)
  { data = d;  count = 1; }

  public void incrementCount()
  { count++; }

  // precondition: the count of this item is greater than 0.
  public void decrementCount()
  { count--; }

  public int getCount()
  { return count; }

  public Comparable getData()
  { return data; }
}
```

For example, if the items 13, 11, 14, 11, 15, 14, and 14 are added to an initially empty `TreePriorityQueue`, then the resulting binary search tree is as shown below (with 11 occurring two times, 14 occurring three times, and 13 and 15 each occurring one time).



(a) Write the `TreePriorityQueue` method `peekMin`. Method `peekMin` returns the minimum data value in the priority queue.

Complete method `peekMin` below.

```
// precondition:  the priority queue is not empty
// postcondition: a data value of smallest priority is returned;
//                the priority queue is unchanged
public Object peekMin()
```

(b) Write the `TreePriorityQueue` private method `addHelper`, which adds `obj` to the subtree rooted at `t`. If a node containing an `Item` with data `obj` is already in the subtree, `addHelper` increments the count; otherwise, `addHelper` adds a new node containing an `Item` with data `obj` and a count of 1 to the subtree, maintaining the binary search tree property. Method `addHelper` returns a reference to the root of the resulting subtree.

Complete method `addHelper` below.

```
// postcondition: obj has been added to the subtree rooted at t;
//                the resulting subtree is returned
private TreeNode addHelper(Comparable obj, TreeNode t)
```

**END OF EXAMINATION**