AP® Computer Science AB
2002 Free-Response Questions

These materials were produced by Educational Testing Service® (ETS®), which develops and administers the examinations of the Advanced Placement Program for the College Board. The College Board and Educational Testing Service (ETS) are dedicated to the principle of equal opportunity, and their programs, services, and employment policies are guided by that principle.

The College Board is a national nonprofit membership association dedicated to preparing, inspiring, and connecting students to college and opportunity. Founded in 1900, the association is composed of more than 4,200 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 22,000 high schools, and 3,500 colleges, through major programs and services in college admission, 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 equity and excellence, and that commitment is embodied in all of its programs, services, activities, and concerns.

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 C++.

Note: Assume that the standard libraries (e.g., `iostream.h`, `fstream.h`, `math.h`, etc.) and the AP C++ classes are included in any program that uses a program segment you write. If other classes are to be included, that information will be specified in individual questions. Unless otherwise noted, assume that all functions are called only when their preconditions are satisfied. A Quick Reference to the AP C++ classes is included in the case study insert.

1. Consider the problem of assigning passengers to seats on airline flights. Three types of information are needed —passenger information, seat information, and flight information. Three classes will be used to represent this information, respectively: `Passenger`, `Seat`, and `Flight`.

   You will write three member functions for the `Flight` class:

   (a) `EmptySeatCount` that returns the number of empty seats of a specified type,

   (b) `FindBlock` that returns information about the location of an empty block of seats, and

   (c) `AssignGroup` that attempts to assign a group of passengers to adjacent seats.

   Passenger information is abstracted by a class and includes a name and other information. A default passenger, used to indicate "no passenger" in a seat, has the empty string `""` as its name. The declaration for class `Passenger` is as follows.

```
class Passenger
{
  public:
    Passenger();    // default passenger with name ""

    apstring GetName() const;
    // postcondition: returns passenger's name

    // ... other public and private members not shown
};
```

   Seat information includes the passenger assigned to the seat and the type of the seat ("window", "aisle", "middle"). The `Seat` function `GetPassenger` returns the passenger assigned to the seat; if the seat is empty, `GetPassenger` returns a default passenger. The declaration for the class `Seat` is as follows.

```
class Seat
{
  public:
    Passenger GetPassenger() const;
    // postcondition: returns passenger in this seat

    apstring GetType() const;
    // postcondition: returns the type of this seat

    void SetPassenger(const Passenger & p);
    // postcondition: assigns p to this seat (i.e., GetPassenger() == p)

    // ... constructors and other public and private members not shown
};
```

Seat assignments are processed by the public member functions of the class `Flight`. The seating arrangement is represented internally by a matrix of seats in the class `Flight`. The declaration for the class `Flight` is as follows.

```
class Flight
{
  public:
    int EmptySeatCount(const apstring & seatType) const;
    // postcondition: returns the number of empty seats
    //                whose type is seatType;
    //                if seatType is "any", returns the
    //                total number of empty seats

    int FindBlock(int row, int seatsNeeded) const;
    // postcondition: returns column index of the first (lowest index)
    //                seat in a block of seatsNeeded adjacent
    //                empty seats in the specified row;
    //                if no such block exists, returns -1

    bool AssignGroup(const apvector<Passenger> & group);
    // postcondition: if possible, assigns the group.length() passengers
    //                from group to adjacent empty seats in a single row
    //                and returns true;
    //                otherwise, makes no changes and returns false

    // ... constructors and other public member functions not shown

  private:
    apmatrix<Seat> mySeats;

    // ... other private data members not shown
};
```

(a) You will write the `Flight` member function `EmptySeatCount`, which is described as follows. `EmptySeatCount` returns the number of empty seats of the specified type `seatType`. Recall that an empty seat holds a default passenger whose name is `""`. If `seatType` is `"any"`, then every empty seat should be counted in determining the number of empty seats. Otherwise, only seats whose type is the same as `seatType` are counted in determining the number of empty seats.

For example, consider the diagram of passengers assigned to seats as stored in `mySeats` for `Flight ap2002` as shown below.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	window “Kelly”	middle “Robin”	aisle “”	aisle “Sandy”	middle “”	window “Fran”
[1]	window “Chris”	middle “Alex”	aisle “”	aisle “”	middle “Pat”	window “Sam”

The following table shows several examples of calling `EmptySeatCount` for this flight.

Function Call	Value Returned
`ap2002.EmptySeatCount("aisle")`	3
`ap2002.EmptySeatCount("window")`	0
`ap2002.EmptySeatCount("middle")`	1
`ap2002.EmptySeatCount("any")`	4

Complete function `EmptySeatCount` below.

```
int Flight::EmptySeatCount(const apstring & seatType) const
// postcondition: returns the number of empty seats
//                whose type is seatType;
//                if seatType is "any", returns the
//                total number of empty seats
```

(b) You will write the `Flight` member function `FindBlock`, which is described as follows. `FindBlock` searches for a block of `seatsNeeded` adjacent empty seats in the specified row. If such a block of seats is found, `FindBlock` returns the column index of the first (i.e., the lowest index) seat in the block; otherwise, it returns -1.

The seating diagram for passengers of `Flight ap2002` is repeated here for your convenience.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	window "Kelly"	middle "Robin"	aisle ""	aisle "Sandy"	middle ""	window "Fran"
[1]	window "Chris"	middle "Alex"	aisle ""	aisle ""	middle "Pat"	window "Sam"

The following table shows several examples of calling `FindBlock` for `Flight ap2002 as shown.`

Function Call	Value Returned
`ap2002.FindBlock(0, 1)`	2 or 4
`ap2002.FindBlock(0, 2)`	-1
`ap2002.FindBlock(1, 2)`	2

Complete function `FindBlock` below.

```
int Flight::FindBlock(int row, int seatsNeeded) const
// postcondition: returns column index of the first (lowest index)
//                seat in a block of seatsNeeded adjacent
//                empty seats in the specified row;
//                if no such block exists, returns -1
```

(c) You will write the `Flight` member function `AssignGroup`, which is described as follows. The parameter to the `Flight` member function `AssignGroup` is an array of passengers, `group`. These passengers require a block of adjacent seats in a single row. `AssignGroup` searches for `group.length()` adjacent empty seats in some row. If such a block of seats is found, the passengers in `group` will be assigned to those seats, and `AssignGroup` returns `true`. Otherwise, no passengers are assigned to seats, and `AssignGroup` returns `false`.

For example, the seats in `Flight ap314` are as shown in the first diagram below. If the array `adults` contains three passengers, the call `ap314.AssignGroup(adults)` makes no changes to `ap314` and returns `false`, because there is no block of three adjacent empty seats in a single row. On the other hand, suppose the array `kids` contains passengers `"Sam"` and `"Alex"`. The call `ap314.AssignGroup(kids)` will assign `"Sam"` and `"Alex"` to the seats shown in the second diagram below and return `true`.

Contents of `mySeats` for `ap314` before any call to `AssignGroup`

	[0]	[1]	[2]	[3]	[4]
[0]	window “Kelly”	aisle “”	aisle “Sandy”	middle “”	window “Fran”
[1]	window “Chris”	aisle “”	aisle “”	middle “Pat”	window “”

Contents of `mySeats` for `ap314` after call to `ap314.AssignGroup(kids)`

	[0]	[1]	[2]	[3]	[4]
[0]	window “Kelly”	aisle “”	aisle “Sandy”	middle “”	window “Fran”
[1]	window “Chris”	aisle “Sam”	aisle “Alex”	middle “Pat”	window “”

In writing `AssignGroup`, you may call `FindBlock` specified in part (b). Assume that `FindBlock` works as specified, regardless of what you wrote in part (b).

Complete function `AssignGroup` below.

```
bool Flight::AssignGroup(const apvector<Passenger> & group)
// postcondition: if possible, assigns the group.length() passengers
//                from group to adjacent empty seats in a single row
//                and returns true;
//                otherwise, makes no changes and returns false
```

2. Consider the following class declaration for representing cards to be used in a program that simulates a card game.

```
class Card
{
  public:

    int Value() const;
    // postcondition:  returns the value of this card

    // ... constructors and other public member functions not shown

  private:
    // ... private data members not shown
};
```

The card game to be simulated is a two-player game called "Compete" that is played with a set of cards, each having a numeric value. The object of the game is to win all of the cards.

Each player has a pile of cards and both piles start with the same number of cards. Play consists of a sequence of "rounds". Play continues until at the end of a round at least one player is out of cards. A discard pile is created and used during the rounds and is empty at the beginning of each round.

You will write code to move cards from one pile to another and to play one round. Each pile of cards will be represented by a queue.

A round is played by executing the following steps until the round ends.

1. If exactly one player's pile is empty, the other player adds all the cards in the discard pile to the bottom of his or her pile without changing the order. The round ends.
2. If both players' piles are empty at the same time, both players' piles remain empty. The round ends.
3. Each player turns over the top card from his/her pile.
4. If the cards match in value, both cards are added to the discard pile in either order and play returns to step 1.
5. If the cards do not match in value, the player whose card has the greater value adds any cards in the discard pile to the bottom of his/her pile without changing the order of the cards in the discard pile. He/she then adds both cards just played to the bottom of his/her pile in either order. The round ends.

(a) You will write free function `AppendQueue`, which is described as follows. `AppendQueue` should remove all the cards from the parameter `source` and add them to the parameter `destination` in the same order.

Complete function `AppendQueue` below.

```
void AppendQueue(apqueue<Card> & destination,
                 apqueue<Card> & source)
// postcondition: all cards have been removed from source
//                and added to destination in the same order
```

(b) You will write free function `OneRound`. `OneRound` takes the two players' piles of cards as parameters and carries out one round of the game. For your convenience the steps for one round are repeated here.

1. If exactly one player's pile is empty, the other player adds all the cards in the discard pile to the bottom of his or her pile without changing the order. The round ends.
2. If both players' piles are empty at the same time, both players' piles remain empty. The round ends.
3. Each player turns over the top card from his/her pile.
4. If the cards match in value, both cards are added to the discard pile in either order and play returns to step 1.
5. If the cards do not match in value, the player whose card has the greater value adds any cards in the discard pile to the bottom of his/her pile without changing the order of the cards in the discard pile. He/she then adds both cards just played to the bottom of his/her pile in either order. The round ends.

In writing `OneRound`, you may call `Card::Value()` and `AppendQueue` from part (a). Assume that `AppendQueue` works as specified, regardless of what you wrote in part (a).

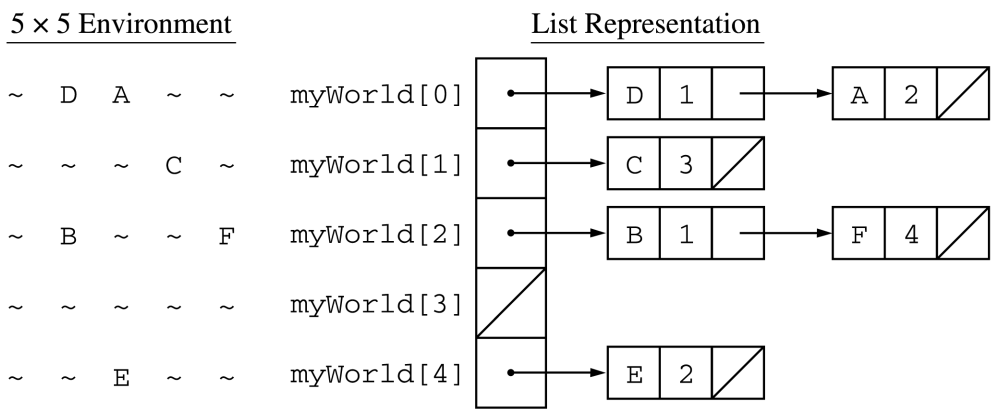
Complete function `OneRound` below.

```
void OneRound(apqueue<Card> & pile1,
              apqueue<Card> & pile2)
// precondition:  pile1.length() > 0;  pile2.length() > 0
// postcondition: pile1 and pile2 have been updated according to the
//                rules of the game
```

3. This question involves reasoning about the code from the Marine Biology Case Study. A copy of the code is provided as part of this exam.

The original version of the case study uses a two-dimensional matrix, `myWorld`, to represent the world in which the simulation takes place. Consider an alternate representation where the fish in each row are stored in a singly linked list. The implementation of `myWorld` becomes an array in which each element contains a pointer to the first node in the linked list for that row. If there are no fish in that row, the pointer is `NULL`. Each list node contains a fish, the column index for that fish, and a pointer to the node containing the next fish in that row. The linked list is ordered by column index, from smallest to largest.

In the example below, `myWorld[0]` is a pointer to the first node of a list containing two fish: fish `D` at column 1 and fish `A` at column 2. The element `myWorld[3]` is `NULL`, indicating that there are no fish in that row.



The list of fish will be implemented using the following declaration.

```
struct ListNode
{
  Fish theFish;
  int columnIndex;
  ListNode * next;

  ListNode();
    // sets theFish to emptyFish, columnIndex to -1, next to NULL

  ListNode(const Fish f, int c, ListNode * link);
    // sets theFish to f, columnIndex to c, next to link
};
```

Consider the following changes (shown in bold) to the private section of the `Environment` class.

```
private:

  bool InRange(const Position & pos) const;
  // postcondition: returns true if pos in grid,
  //                returns false otherwise

  apvector<ListNode *> myWorld;   // grid of fish

  int myNumCols;
    // from file input when environment constructed

  int myFishCreated;         // # fish ever created
  int myFishCount;           // # fish in current environment
```


GO ON TO THE NEXT PAGE.

(a) Modify the `Environment` member function `AllFish` to use the revised data structure. In writing `AllFish`, you may use any other `Environment` member functions or the public member functions of any other class used in this case study. Assume that all member functions work as specified.

Complete function `AllFish` below. Note the changes shown in bold.

```
apvector<Fish> Environment::AllFish() const
// postcondition: returned vector (call it fishList) contains all
//                fish in top-down, left-right order:
//                top-left fish in fishList[0],
//                bottom-right fish in fishList[fishList.length()-1];
//                # fish in environment is fishList.length()
{
  apvector<Fish> fishList(myFishCount);
  int r, k;    // c from original not needed
  int count = 0;
  apstring s = "";
  ListNode * tempPtr;

  // look at all grid positions, store fish found in vector fishList

  // insert code here










  // end of inserted code

  for (k = 0; k < count; k++)
  {
    s += fishList[k].Location().ToString() + " ";
  }
  DebugPrint(5, "Fish vector = " + s);
  return fishList;
}
```


GO ON TO THE NEXT PAGE.

(b) Modify the `Environment` member function `AddFish` to use the revised data structure. The new fish should be inserted into the correct row's linked list, maintaining the order of the list sorted by column index.

In writing `AddFish`, you may use any other `Environment` member functions or the public member functions of any other class used in this case study. Assume that all member functions work as specified.

Complete function `AddFish` below.

```
void Environment::AddFish(const Position & pos)
// precondition:  no fish already at pos, i.e., IsEmpty(pos)
// postcondition: fish created at pos
{
  if (! IsEmpty(pos))
  {
    cerr << "error, attempt to create a fish at non-empty: "
          << pos << endl;
  }

  // insert code here









}
```

4. Consider the problem of encoding words as a string of 0's and 1's using a codetree. A codetree is a binary tree containing letters in its leaves. The encoding of a letter is represented by the root-to-leaf path for the letter. The same codetree is used for encoding and decoding.

   The following properties hold for every codetree.

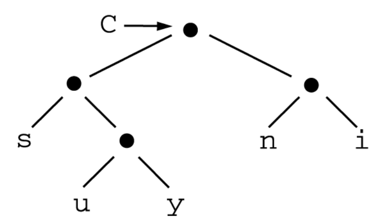
   (i) Every node is either a leaf or has exactly 2 children.

   (ii) Letters appear only at the leaves of the codetree.

   (iii) There are at least 2 letters in the codetree.

   (iv) Each letter appears at most once in the codetree; thus there is a unique root-to-leaf path encoding for each letter.

   For example, consider the following codetree, `C`.



   The code for each letter is a string formed by appending a `'0'` when taking a left branch and a `'1'` for a right branch when traversing the root-to-leaf path. In the codetree above, the code for `'u'` is `"010"` (left, right, left), the code for `'s'` is `"00"`, and the code for `'n'` is `"10"`. A word is encoded by appending the codes for letters in the word together. For example, the code for `"sun"` is `"0001010"`, which is formed by appending the codes for `'s'`, `'u'`, and `'n'`.

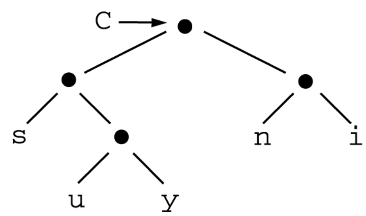
Consider the following declarations for a tree node and a class that represents the codetree.

```
struct Node
{
  char letter;    // value ignored except in leaves
  Node * left;    // link to left child
  Node * right;   // link to right child
};

class CodeTree
{
  public:

    apstring BitsToWord(const apstring & code) const;
    // precondition:  code is a string of 0's and 1's representing
    //                a valid encoded word
    // postcondition: returns decoded word for code

    apstring WordToBits(const apstring & word) const;
    // precondition:  each character in word is in a leaf
    //                of the codetree
    // postcondition: returns the code for word

    // ... constructor and other public member functions not shown

  private:

    apstring CharToBitsHelper(char ch, Node * T,
                              const apstring & pathSoFar) const;
    // postcondition: if ch is in subtree T, returns code for ch

    Node * myRoot;

    // ... other private data members and functions not shown
};
```


GO ON TO THE NEXT PAGE.

(a) You will write the `CodeTree` member function `BitsToWord`, which is described as follows. `BitsToWord` is given a coded word (a string of 0's and 1's) and returns the decoded word.

Each character of `code` represents a branch in the codetree, where `'0'` represents a left branch and `'1'` represents a right branch. To decode the word represented by `code`, begin at the root and follow a branch for each `'0'` or `'1'` character in `code`. When a leaf is reached, one letter in the decoded word has been found. The decoding process begins again at the root of the codetree with the next `'0'` or `'1'` character in `code`.



For example, using the `CodeTree C` as shown, if code is `"1110"`, the call `C.BitsToWord(code)` returns the word `"in"`. This result is obtained as follows. The path starts at the root and goes right for the first `'1'`, right again for the second `'1'`, and a leaf is reached, meaning the decoded word begins with `'i'`. Starting back at the root of the codetree and with the next `'1'` in `code`, the path goes right for `'1'` and left for `'0'`, reaching the leaf with the letter `'n'`. The decoded word is now `"in"`, and since all characters in `code` have been processed, `"in"` is returned. Similarly, `C.BitsToWord("000101010011")` returns the word `"sunny"`.

Complete function `BitsToWord` below.

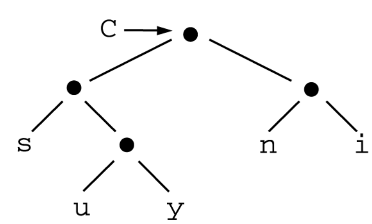
```
apstring CodeTree::BitsToWord(const apstring & code) const
// precondition:  code is a string of 0's and 1's representing
//                a valid encoded word
// postcondition: returns decoded word for code
```

(b) The implementation of `WordToBits` given below forms the code for `word` by appending the result of calling the private member function `CharToBitsHelper` once for each character in the parameter `word`.

```
apstring CodeTree::WordToBits(const apstring & word) const
// precondition:  each character in word is in a leaf of the codetree
// postcondition: returns the code for word
{
    apstring bits;
    for (int k = 0; k < word.length(); k++)
    {
      bits += CharToBitsHelper(word[k], myRoot, "");
    }
    return bits;
}
```

You will write the `CodeTree` private member function `CharToBitsHelper`.

`CharToBitsHelper` has a third parameter, `pathSoFar`, that can be used to keep track of the current path from `myRoot` to `T`, should you choose to do so. For this reason, the value of `pathSoFar` in the call from `WordToBits` is `""`.



Using `CodeTree C` as shown, `CharToBitsHelper('y', myRoot, "")` would return the string `"011"` and `CharToBitsHelper('n', myRoot, "")` would return `"10"`.

Complete function `CharToBitsHelper` below.

```
apstring CodeTree::CharToBitsHelper(char ch, Node * T,
                                    const apstring & pathSoFar) const
  // postcondition: if ch is in subtree T, returns code for ch
```

END OF EXAMINATION