AP® Computer Science AB
2003 Free-Response Questions

For the College Board's online home for AP professionals, visit AP Central at 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 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.

GO ON TO THE NEXT PAGE.

1. Periodically, a company processes the retirement of some of its employees. In this question, you will write functions to help the company determine whether an employee is eligible to retire and to process the retirement of employees who wish to retire. You will also analyze the runtime performance of one of the functions that you write.

   The `Employee` class is declared as follows.

```
class Employee
{
  public:
    int Age() const;
    // returns the age (in years) of this employee

    int YearsOnJob() const;
    // returns the number of years this employee has worked

    double Salary() const;
    // returns the salary of this employee in dollars

    int ID() const;
    // returns unique employee ID number

  // ... constructors, other member functions and data not shown
};
```

GO ON TO THE NEXT PAGE.

The Company class is declared as follows.

```
class Company
{
  public:
    void ProcessRetirements(const apvector<Employee> & claimants);
    // precondition:  claimants is sorted in ascending order
    //                by employee ID with no duplicates;
    //                all claimants are in empList
    // postcondition: all eligible employees in claimants have been
    //                removed from empList; empList has been resized to
    //                reflect retirements;
    //                empList remains sorted by employee ID;

    //                salaryBudget has been updated to reflect remaining
    //                employees

  private:
    bool EmployeeIsEligible(const Employee & emp) const;
    // postcondition: returns true if emp is eligible to retire;
    //                otherwise, returns false

    apvector<Employee> empList;
    // stores the employees sorted by employee ID in ascending order
    // empList.length() is the number of employees

    int retireAge;          // minimum age to retire
    int retireYears;        // minimum years on job to retire
    double retireSalary;    // minimum salary to retire

    double salaryBudget;    // total salary of all employees

};
```

GO ON TO THE NEXT PAGE.

The data member `empList` is sorted in ascending order by employee ID. The total of all salaries is maintained in the data member `salaryBudget`.

(a) An employee is eligible for retirement if (s)he meets at least two of the following requirements:

1. The employee is at least `retireAge` years old.

2. The employee has worked for at least `retireYears`.

3. The employee's salary is at least `retireSalary`.

Write the `Company` member function `EmployeeIsEligible`, which is described as follows. `EmployeeIsEligible` returns true if `Employee emp` is eligible for retirement, using the rules described above.

Complete function `EmployeeIsEligible` below.

```
bool Company::EmployeeIsEligible(const Employee & emp) const
// postcondition: returns true if emp is eligible to retire;
//                otherwise, returns false
```

(b) Write the `Company` member function `ProcessRetirements`, which is described as follows. The function takes as its only parameter an array, `claimants`, representing all employees that wish to retire. Assume that `claimants` is sorted in ascending order by ID number, contains no duplicates, and that all elements in `claimants` are also in `empList`. `ProcessRetirements` removes from `empList` only those employees listed in `claimants` that are eligible for retirement, resizes (shrinks) `empList` as appropriate, and decreases `salaryBudget` accordingly.

In writing `ProcessRetirements`, you may call `EmployeeIsEligible`, specified in part (a). Assume that `EmployeeIsEligible` works as specified, regardless of what you wrote in part (a).

Complete function `ProcessRetirements` below.

```
void Company::ProcessRetirements(const apvector<Employee> & claimants)
// precondition:  claimants is sorted in ascending order
//                by employee ID with no duplicates;
//                all claimants are in empList
// postcondition: all eligible employees in claimants have been
//                removed from empList; empList has been resized to
//                reflect retirements;
//                empList remains sorted by employee ID;
//                salaryBudget has been updated to reflect remaining
//                employees
```

(c) Assume that *N* is the number of employees in the company. Give the best Big-Oh expression (in terms of *N*) for the worst-case running time for your implementation of the function `ProcessRetirements`. Justify your answer with reference to the code you wrote in part (b). You will NOT receive full credit if you do not provide a justification.

GO ON TO THE NEXT PAGE.

2. Consider the problem of representing a filing system for student records that are stored in the drawers of a filing cabinet. The system uses a linked list in which each node represents a drawer of the filing cabinet. Each drawer in the list contains a pointer to the first node of a linked list of student records, a pointer to the next drawer in the filing cabinet, and the maximum student ID number that can be filed in that drawer. Within each drawer, the student records are stored by student ID number in ascending order. The drawers in the filing cabinet are ordered by the maximum student ID number that can be filed in each drawer. The maximum student ID number for the last drawer in the cabinet is greater than the largest possible student ID number.

The diagram below illustrates the structure of the filing system. The first node in the list of drawers would be pointed to by the private data member `drawerList` of the `FilingCabinet` class as declared on the next page.

drawerList

```
drawerMaxID:          idNum:   123        idNum:   178         idNum:   250
   250

drawerMaxID:
   428

drawerMaxID:          idNum:   542
   637


   . . .


drawerMaxID:          idNum:   834            . . .            idNum:   987
   999
```

GO ON TO THE NEXT PAGE.
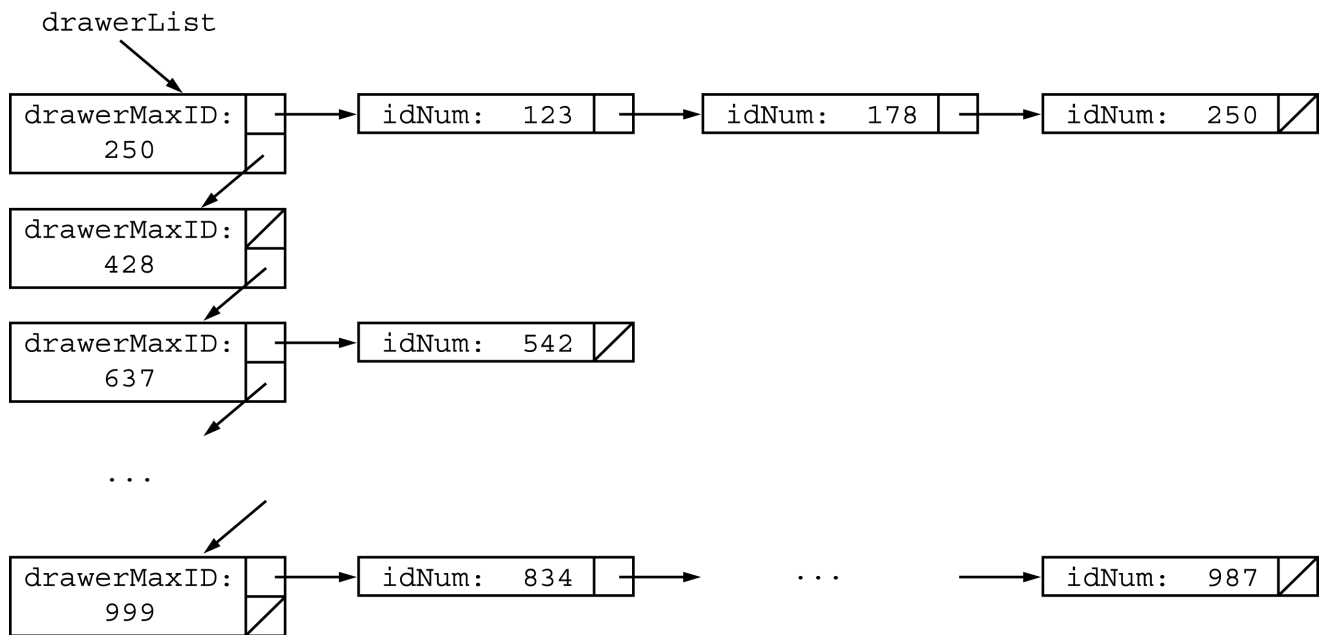
The following declarations represent this filing system.

```
struct StudentNode
{
  int idNum;
  StudentNode * next;

  // ... other student information not shown
};

struct DrawerNode
{
  int drawerMaxID;
  StudentNode * studentList; // if drawer is empty, studentList is NULL
  DrawerNode * next;
};

class FilingCabinet
{
  public:
    DrawerNode * FindDrawer(int studentID) const;
    // precondition:  this FilingCabinet has at least one drawer;
    //                studentID is less than or equal to drawerMaxID
    //                of the last drawer
    // postcondition: returns the first DrawerNode * d such that
    //                studentID is less than or equal to d->drawerMaxID

    void RemoveStudent(int studentID);
    // precondition:  this FilingCabinet has at least one drawer;
    //                studentID is less than or equal to drawerMaxID
    //                of the last drawer
    // postcondition: if there is a node containing studentID in this
    //                FilingCabinet, that node has been removed from its
    //                drawer; otherwise this FilingCabinet is unchanged


  private:
    DrawerNode * drawerList;

  // ... constructor, other member functions, and data not shown
};
```

GO ON TO THE NEXT PAGE.

(a) Write the `FilingCabinet` member function `FindDrawer`, which is described as follows. `FindDrawer` returns a pointer to the drawer in which `studentID` would be found. `FindDrawer` returns the first drawer in the list for which `studentID` is less than or equal to the maximum student ID number that can be filed in the drawer.

Complete function `FindDrawer` below.

```
DrawerNode * FilingCabinet::FindDrawer(int studentID) const
// precondition:  this FilingCabinet has at least one drawer;
//                studentID is less than or equal to drawerMaxID
//                of the last drawer
// postcondition: returns the first DrawerNode * d such that
//                studentID is less than or equal to d->drawerMaxID
```

(b) Write the `FilingCabinet` member function `RemoveStudent`, which is described as follows. `RemoveStudent` should find the drawer in which `studentID` should be located and remove the list node containing that student ID from the list associated with that drawer (calling `delete` as necessary). If a node containing `studentID` is not in the drawer then the `FilingCabinet` is unchanged.

In writing `RemoveStudent`, you may call `FindDrawer` specified in part (a). Assume that `FindDrawer` works as specified, regardless of what you wrote in part (a).

Complete function `RemoveStudent` below.

```
void FilingCabinet::RemoveStudent(int studentID)
// precondition:  this FilingCabinet has at least one drawer;
//                studentID is less than or equal to drawerMaxID
//                of the last drawer
// postcondition: if there is a node containing studentID in this
//                FilingCabinet, that node has been removed from its
//                drawer; otherwise this FilingCabinet is unchanged
```
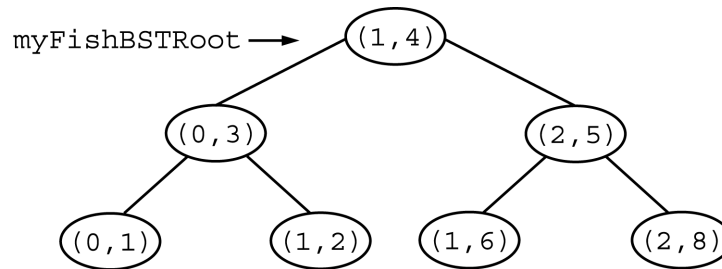
GO ON TO THE NEXT PAGE.

3. This question involves reasoning about the code from the Marine Biology Case Study. A copy of the code is provided in the Appendix.

   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 are stored in a binary search tree. `myWorld` is replaced by `myFishBSTRoot`, a pointer to the root node of the binary search tree. The fish are stored in the binary search tree according to the ordering of their positions, top-down, left-right (row-major order).

   The following overloaded operator has been added to the public section of the `Fish` class.

   ```
   bool operator < (const Fish & rhs) const;
   // postcondition: returns true if the location of this fish occurs
   //                before the location of rhs in row-major order
   ```

   In the example below, `myFishBSTRoot` points to the root node of a binary search tree containing seven fish (each indicated by the coordinates of its position).

   ```
   myFishBSTRoot ⟶  (1,4)
                   /      \
              (0,3)        (2,5)
             /    \        /    \
        (0,1)    (1,2)  (1,6)   (2,8)
   ```

   The binary search tree of fish will be implemented using the following declaration.

   ```
   struct Node
   {
     Fish theFish;
     Node * left;
     Node * right;

     Node();
       // sets theFish to emptyFish, left and right to NULL
     Node(const Fish & fsh);
       // sets theFish to fsh, left and right to NULL
   };
   ```

GO ON TO THE NEXT PAGE.

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

  void AllFishHelper(Node * root, apvector<Fish> & fishList,
                 int & index) const;

  void AddFishHelper(Node * & root, const Fish & fsh);

  Node * myFishBSTRoot;      // root pointer for binary search
                            // tree of fish

  int myNumRows;
  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) The `Environment` member function `AllFish` is modified to use the private helper function `AllFishHelper` as follows.

```
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 index = 0;
  apstring s = "";

  AllFishHelper(myFishBSTRoot, fishList, index);

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

Write the private `Environment` member function `AllFishHelper`, which is described as follows. `AllFishHelper` should add all the fish in the tree represented by `root` to `fishList`, starting from `index`. It is guaranteed that `fishList` is large enough to hold all the fish in the tree.

In writing `AllFishHelper`, you may use any `Environment` member functions or the public member functions of any other class used in this case study, including `Fish::operator <` specified at the beginning of the question. Assume that all member functions work as specified.

Complete function `AllFishHelper` below.

```
void Environment::AllFishHelper(Node * root, apvector<Fish> & fishList,
                                int & index) const
// precondition:  0 <= index < fishList.length();
//                there are no more than fishList.length() – index fish
//                in the subtree represented by root
// postcondition: All fish in the subtree represented by root have been
//                added to fishList in top-down, left-right order,
//                starting from index;
//                index has been increased by the number of fish
//                added to fishList
```

GO ON TO THE NEXT PAGE.

(b) The `Environment` member function `AddFish` is modified to use the private helper function `AddFishHelper` as follows.

```
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;
  }

  myFishCreated++;
  AddFishHelper(myFishBSTRoot, Fish(myFishCreated, pos));
  myFishCount++;
}
```

Write the private `Environment` member function `AddFishHelper`, which is described as follows. `AddFishHelper` should insert the fish into the binary search tree represented by `root`.

In writing `AddFishHelper`, you may use any `Environment` member functions or the public member functions of any other class used in this case study, including `Fish::operator <` specified at the beginning of the question. Assume that all member functions work as specified.

Complete function `AddFishHelper` below.

```
void Environment::AddFishHelper(Node * & root, const Fish & fsh)
// precondition:  root represents a subtree of a binary search tree;
//                nodes are ordered by fish position; no node in the
//                tree contains a fish at the same position as fsh
// postcondition: Fish fsh has been added to the subtree represented by
//                root, maintaining correct order of nodes
```

GO ON TO THE NEXT PAGE.

4. Consider the problem of separating a string into its component pieces, called tokens. The tokens of a string are its substrings that do not contain white space (spaces and tabs). For example:

```
string                    # tokens              tokens

""                          0
"apple"                     1                    "apple"
"big red house"             3                    "big", "red", "house"
" this  is  a test"         4                    "this", "is", "a", "test"
```

The functionality of separating a string into its tokens can be encapsulated in a `StringTokenizer` class. The class provides access to the tokens in a string and has the following characteristics.

- A constructor that allows an instance to be created using a string parameter as the source of tokens
- A member function that returns the number of tokens
- A member function that returns the string that is the kth token where the first token has index 0 and the last token has index one less than the number of tokens
- An appropriate data representation to support $O(1)$ implementation of these two member functions.

(a) Write the class declaration for `StringTokenizer` as it would appear in a `StringTokenizer.h` file. In writing the declaration, you must:

- choose appropriate identifiers for member functions and data members,
- provide the functionality specified above,
- use the `const` qualifier for functions and parameters where appropriate,
- provide a data representation consistent with the specification above, and
- make design decisions that are consistent with information-hiding principles.

YOU SHOULD NOT WRITE THE IMPLEMENTATIONS OF THE MEMBER FUNCTIONS OR THE CONSTRUCTOR OF THE `StringTokenizer` CLASS.

Complete the `StringTokenizer` class declaration on the following page in the skeleton provided.

```
class StringTokenizer
{
  public:




















  private:











};
```

(b) Write free function `CreateAcronym`, which is described as follows. `CreateAcronym` takes a string, `str`, as a parameter and returns a string that is the acronym formed from the first character of each token of `str`. The following table shows several examples of calls to `CreateAcronym`.

str	String returned by CreateAcronym(str)
"red orange yellow green blue indigo violet"	"roygbiv"
" as soon as possible"	"asap"
"Rolling on the floor laughing"	"Rotfl"
"As Far As I Know"	"AFAIK"

More formally, the acronym for a string `str` is formed by concatenating the first character of each token of `str` in the same order that the tokens appear in `str`.

In writing `CreateAcronym`, you must use the `StringTokenizer` class you designed in part (a). To receive full credit, the tokens of `str` must only be obtained by using the member functions of the `StringTokenizer` class that implement the specification given at the beginning of the question. Assume that the `StringTokenizer` has been implemented as specified.

Complete function `CreateAcronym` below.

```
apstring CreateAcronym(const apstring & str)
```

END OF EXAMINATION