AP Computer Science AB
2001 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 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. A window is represented by an *M*-by-*N* matrix filled with integers representing colors. Operations on a window include the following.

   - Determine if a point lies within the window.
   - Place a square of a single color in the window, ignoring those points in the square that are not within the window.

   Consider the following declarations for `Window`.

```
class Window
{
  public:

    // ... constructors not shown

    bool IsInBounds(int row, int col) const;
    // postcondition: returns true if the point (row, col) is
    //                in this window;
    //                otherwise, returns false

    void ColorSquare(int ULrow, int ULcol, int N, int val);
    // postcondition: all points in this window that are also in the
    //                N-by-N square with upper left corner
    //                (ULrow, ULcol) have been set to val;
    //                points in the square that are not in this
    //                window are ignored

    int ValAt(int row, int col) const;
    // postcondition: returns color value at position row, col
    //                in this window

    // ... other public member functions not shown

  private:
    int myNumRows;
    int myNumCols;
    apmatrix<int> myMat;
};
```


GO ON TO THE NEXT PAGE.

(a) Write the `Window` member function `IsInBounds`, as started below. `IsInBounds` checks whether a single point is in the window.

For example, for any 5-by-4 `Window W`, the following table shows the results of several calls to `IsInBounds`.

Call	Return value
`W.IsInBounds(0, 0)`	`true`
`W.IsInBounds(2, 1)`	`true`
`W.IsInBounds(4, 3)`	`true`
`W.IsInBounds(5, 3)`	`false`
`W.IsInBounds(3, -1)`	`false`
`W.IsInBounds(8, 8)`	`false`

Complete function `IsInBounds` below.

```
bool Window::IsInBounds(int row, int col) const
// postcondition: returns true if the point (row, col) is
//                in this window;
//                otherwise, returns false
```

(b) Write the `Window` member function `ColorSquare`, as started below. `ColorSquare` sets all the integers in a specified square to a particular color value. `ULrow` and `ULcol` specify the location of the upper left corner of the square, `N` is the number of rows and columns in the square, and `val` is the color value. Points that are in the specified square but do not lie in the `Window` are ignored.

For example, consider the following 5-by-6 `Window W`.

```
10 10 10 10 10 10
10 10 10 20 20 20
20 20 30 30 30 30
30 30 40 40 40 40
40 40 50 50 50 50
```

After the call `W.ColorSquare(2, 1, 3, 66)`, `W` is changed to

```
10 10 10 10 10 10
10 10 10 20 20 20
20 66 66 66 30 30
30 66 66 66 40 40
40 66 66 66 50 50
```

After an additional call, `W.ColorSquare(2, 4, 3, 77)`, `W` is changed to

```
10 10 10 10 10 10
10 10 10 20 20 20
20 66 66 66 77 77
30 66 66 66 77 77
40 66 66 66 77 77
```

Note that the third column of the square added is not in `W`.

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

Complete function `ColorSquare` below.

```
void Window::ColorSquare(int ULrow, int ULcol, int N, int val)
// postcondition: all points in this window that are also in the
//                N-by-N square with upper left corner
//                (ULrow, ULcol) have been set to val;
//                points in the square that are not in this
//                window are ignored
```


GO ON TO THE NEXT PAGE.

(c) A rectangular area in a window can be specified using the `Rectangle` structure as declared below.

```
struct Rectangle
{
  int ULrow;   // row position of upper left corner of rectangle
  int ULcol;   // column position of upper left corner of rectangle
  int numRows; // number of rows in rectangle (height)
  int numCols; // number of columns in rectangle (width)
};
```

The following example shows a 5-by-4 window in which the 3-by-2 rectangle with upper left corner (2,1) is highlighted.

10	20	30	40
10	20	30	40
10	**99**	**55**	40
10	**44**	**33**	40
10	**77**	**66**	40

Write the free function `Enlarge`, as started below. `Enlarge` magnifies a `Rectangle` in the `Window` by replacing each point with a `factor-by-factor` square of points of the same color. The upper left corner of the magnified `Rectangle` is the same as the upper left corner of the original `Rectangle`. Each square of color is placed in the `Window` at the same relative position in the magnified `Rectangle` as the original point in the `Rectangle`. Conceptually, the enlarged rectangle may run off the window, but only points in the window are modified by `Enlarge`.

For example, consider the 10-by-11 `Window W`, and `Rectangle R`, where `R.ULrow = 2`, `R.ULcol = 1`, `R.numRows = 2`, and `R.numCols = 4`. The following table shows the original version of `W` with `R` highlighted and the result of magnifying `R` in `W` by a `factor` of `3`.

<u>`Window W` with `R` highlighted</u>

00	00	00	10	10	10	10	10	10	10	10
10	10	10	20	20	20	20	20	20	20	20
20	**55**	**99**	**33**	**66**	20	20	20	30	30	30
30	**22**	**88**	**77**	**44**	30	30	30	40	40	40
40	40	40	40	30	30	30	50	50	50	50
50	50	50	40	40	40	40	30	30	30	30
60	60	50	50	50	40	40	40	30	30	20
70	70	70	50	50	50	40	40	40	30	30
80	80	70	70	60	60	50	50	40	40	30
90	80	70	60	60	50	50	40	40	30	20

<u>Result of the call `Enlarge(W, R, 3)`</u>

00	00	00	10	10	10	10	10	10	10	10
10	10	10	20	20	20	20	20	20	20	20
20	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
30	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
40	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
50	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
60	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
70	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
80	80	70	70	60	60	50	50	40	40	30
90	80	70	60	60	50	50	40	40	30	20

In writing `Enlarge`, you may call any public `Window` member functions. Assume that `IsInBounds` and `ColorSquare` work as intended, regardless of what you wrote in parts (a) and (b).

Complete function `Enlarge` below.

```
void Enlarge(Window & W, const Rectangle & rect, int factor)
// precondition:  factor > 0
```

2. 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.

   Consider modifying the Marine Biology Case Study to have fish breed, age, and die. The `Fish` class will have the following changes:

   - A new private data member, `myAge`, will store the age of the fish.
   - A new private data member, `myProbDie`, will store the probability (between 0.0 and 1.0) that the fish dies in any given time step.
   - A new constructor will take the fish's starting age and probability of dying as parameters, in addition to the id and position parameters.
   - The original constructors will set the starting age and probability of dying to default values.
   - A new public member function, `Act`, will take actions for the fish for one step in the simulation.
   - A new private member function, `Breed`, will reproduce new fish.
   - The `Move` function will become a private member function, called by `Act`. (Note that `Simulate::Step` will now call `Fish::Act` rather than `Fish::Move`.)

GO ON TO THE NEXT PAGE.

The modified `Fish` class declaration is shown below with additions in **boldface**.

```
class Fish
{
  public:

    // constructors

    Fish();
    // postcondition: IsUndefined() == true

    Fish(int id, const Position & pos);
    // postcondition: Location() returns pos, Id() returns id,
    //                IsUndefined() == false

    Fish(int id, const Position & pos, int age, double probDie);
    // precondition:  id not used for any other fish;
    //                probDie is between 0.0 and 1.0
    // postcondition: Location() returns pos, Id() returns id,
    //                IsUndefined() == false,
    //                this fish's probability of dying is probDie

    // accessing functions

    int Id() const;
    Position Location() const;
    bool IsUndefined() const;

    apstring ToString() const;
    char ShowMe() const;

    // modifying functions

    void Act(Environment & env);
    // precondition:  this fish is stored in env at Location()
    // postcondition: this fish has moved, bred, or died

  private:

    Neighborhood EmptyNeighbors(const Environment & env,
                                const Position & pos) const;
    void AddIfEmpty(const Environment & env,
                    Neighborhood & nbr, const Position & pos) const;

    void Breed(Environment & env);
    // precondition:  this fish is stored in env at Location();
    //                this fish is old enough to breed
    // postcondition: the neighboring empty positions of this fish have
    //                been filled with new fish, each with age 0 and
    //                the same probability of dying as this fish

    void Move(Environment & env); // now a private member function

    int myId;
    Position myPos;
    bool amIDefined;

    int myAge;         // age of this fish
    double myProbDie;  // probability that this fish dies on a given
                       // step as a probability between 0.0 and 1.0
};
```

The `Environment` class will have the following changes.

- The constructor will read and initialize fish ages and probabilities of dying, along with their positions.
- The `AddFish` member function will take the fish's age and probability of dying as additional parameters.
- A new public member function, `RemoveFish`, will remove an existing fish from the environment.

The modified `Environment` class declaration is shown below with additions in **boldface**.

```
class Environment
{
  public:
    // constructor

    Environment(istream & input);

    // accessing functions

    int NumRows() const;
    int NumCols() const;
    apvector<Fish> AllFish() const;
    bool IsEmpty(const Position & pos) const;

    // modifying functions

    void Update(const Position & oldLoc, Fish & fish);

    void AddFish(const Position & pos, int age, double probDie);
    // precondition:  no fish already at pos, i.e., IsEmpty(pos)
    // postcondition: fish created at pos with the specified age and
    //                probability of dying

    void RemoveFish(const Position & pos);
    // precondition:  there is a fish at pos (IsEmpty(pos) is false)
    // postcondition: fish removed from pos; IsEmpty(pos) is true

  private:
    bool InRange(const Position & pos) const;

    apmatrix<Fish> myWorld; // grid of fish
    int myFishCreated;      // # fish ever created
    int myFishCount;        // # fish in current environment
};
```


GO ON TO THE NEXT PAGE.

(a) Write the `Environment` member function `RemoveFish`, as started below. `RemoveFish` checks its precondition and prints an error message if the precondition is not met. Otherwise, `RemoveFish` removes the fish in position `pos` from the environment and updates `myFishCount`.

In writing `RemoveFish`, you do not need to include calls to `DebugPrint`.

Complete function `RemoveFish` below.

```
void Environment::RemoveFish(const Position & pos)
// precondition:  there is a fish at pos (IsEmpty(pos) is false)
// postcondition: fish removed from pos; IsEmpty(pos) is true
{

  if (IsEmpty(pos))
  {
    cerr << "error - attempt to remove nonexistent fish at:"
         << pos << endl;
    return;
  }
```

(b) Write the `Fish` member function `Breed`, as started below. `Breed` asks the environment, `env`, to add a new fish in every one of the fish's empty neighboring positions, each with age 0 and with the same probability of dying as this fish.

In writing `Breed`, you do not need to include calls to `DebugPrint`. Assume that all member functions of the `Environment` class work as specified above.

Complete function `Breed` below.

```
void Fish::Breed(Environment & env)
// precondition:  this fish is stored in env at Location();
//                this fish is old enough to breed
// postcondition: the neighboring empty positions of this fish have
//                been filled with new fish, each with age 0 and
//                the same probability of dying as this fish
```

(c) Write the `Fish` member function `Act`, as started below. `Act` will, with probability `myProbDie`, cause the fish to die by calling `env.RemoveFish`. If the fish does not die, it should increment its age. If its new age is three, it should breed; otherwise, it should attempt to move. You will not receive full credit if you reimplement `Move` and `Breed` within function `Act`.

In writing `Act`, you do not need to include calls to `DebugPrint`. Assume that all member functions of the `Environment` and `Fish` classes work as specified above. You may also assume that `Environment` member function `RemoveFish` and the `Fish` member function `Breed` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `Act` below.

```
void Fish::Act(Environment & env)
// precondition:  this fish is stored in env at Location()
// postcondition: this fish has moved, bred, or died
```

3. Consider using a Radix Sort to order a list of positive integers. A Radix Sort makes as many passes through the list as there are digits in the largest number to be sorted. For example, if the largest integer in the list were 492, then the algorithm would make three passes through the list to sort it.

   In each pass through the list, the Radix Sort algorithm sorts the numbers based on a different digit, working from the least to the most significant digit. To do this, it uses an intermediate data structure `QA`, which is an array of ten queues. Each number is placed into the queue corresponding to the value of the digit being examined. For example, in the first pass the digit in the ones' place is considered, so the number 34**5** would be enqueued into `QA[5]`. The number 26**0** would be enqueued into `QA[0]`. In each pass, the algorithm moves the numbers to be sorted from the list to the array of queues and then back to the list, as described below. After the last pass, the list is in sorted order, from smallest to largest.

   Radix Sort Algorithm: In each pass through the list, do the following two steps.

   <u>Step 1</u>

   Taking each integer in the list in order, insert the integer into the queue corresponding to the value of the digit currently being examined. If the integer being examined does not have a digit at a given place value, 0 is assumed for that place value. For example, 95 has no digit in the hundreds' place, so when examining the hundreds' digit, the algorithm would assume that the value in the hundreds' place is 0 and enqueue 95 into `QA[0]`.

   <u>Step 2</u>

   After all integers have been inserted into the appropriate queues, each queue is emptied in order back into the list, starting with `QA[0]`.

GO ON TO THE NEXT PAGE.

For example, assume that the `apvector L` contains the list of integers 380, 95, 345, 382, 260, 100, and 492. The sort will take three passes, because the largest integer in the list has 3 digits. The following diagram shows the sorting process. (For passes II and III, only the nonempty queues are shown in order to save space.)

```
        List before          QA   front        rear              List after
           Pass                     ↓            ↓                  Pass
Pass    [0]   380            [0]  380    260   100               [0]   380
 I      [1]    95            [1]                                 [1]   260
        [2]   345            [2]  382    492                     [2]   100
        [3]   382      =>    [3]                           =>    [3]   382
        [4]   260    Step 1  [4]                         Step 2  [4]   492
        [5]   100            [5]   95    345                     [5]    95
        [6]   492            [6]                                 [6]   345
                             [7]
                             [8]
                             [9]

Pass    [0]   380            [0]  100                            [0]   100
 II     [1]   260            [4]  345                            [1]   345
        [2]   100      =>    [6]  260                      =>    [2]   260
        [3]   382    Step 1  [8]  380    382             Step 2  [3]   380
        [4]   492            [9]  492     95                     [4]   382
        [5]    95                                                [5]   492
        [6]   345                                                [6]    95

Pass    [0]   100            [0]   95                            [0]    95
 III    [1]   345            [1]  100                            [1]   100
        [2]   260      =>    [2]  260                      =>    [2]   260
        [3]   380    Step 1  [3]  345    380   382       Step 2  [3]   345
        [4]   382            [4]  492                            [4]   380
        [5]   492                                                [5]   382
        [6]    95                                                [6]   492
```

The Radix sort uses the following data structures.

```
apvector<int> L;                 // the list to be sorted

apvector<apqueue<int> > QA(10);  // vector of queues, corresponding
                                 // to the digits 0 through 9
```

(a) Write function `ItemsToQueues`, as started below. `ItemsToQueues` corresponds to step 1 of each pass of the Radix sort algorithm, creating the intermediate array of ten queues. Each integer in `L` is inserted into the queue corresponding to the value of the digit currently being examined. If an integer does not have a digit at the given place value, 0 is assumed for that place value.

In writing `ItemsToQueues`, you may call function `GetDigit`, which returns the `k`th digit of its parameter, `number`. The least significant digit is indicated by a value of 0 for `k`. If `k` is greater than the number of digits in `number`, then `GetDigit` returns `0`.

```
int GetDigit(int number, int k);
// precondition:  number ≥ 0; k ≥ 0
// postcondition: returns kth digit of number
```

The following table illustrates the results of several calls to `GetDigit`.

number	k	GetDigit(number, k)
95	0	5
95	1	9
95	2	0

You do not need to implement `GetDigit`.

Complete function `ItemsToQueues` below.

```
apvector<apqueue<int> > ItemsToQueues(const apvector<int> & L, int k)
// precondition:  all values in L are positive;
//                k ≥ 0
// postcondition: Step 1 of the Radix sort algorithm for the digit
//                at position k has been completed
```

(b) Write function `QueuesToArray`, as started below. `QueuesToArray` corresponds to step 2 of each pass of the Radix sort algorithm, creating a new list from the values in the intermediate array of queues.

Complete function `QueuesToArray` below.

```
apvector<int> QueuesToArray(apvector<apqueue<int> > & QA, int numVals)
// precondition:  QA.length() == 10; numVals is the total number of
//                integers in all the queues in QA
// postcondition: returns an apvector of length numVals that contains
//                the integers from QA[0] through QA[9] in the order
//                in which they were stored in the queues;
//                the queues in QA are empty
```

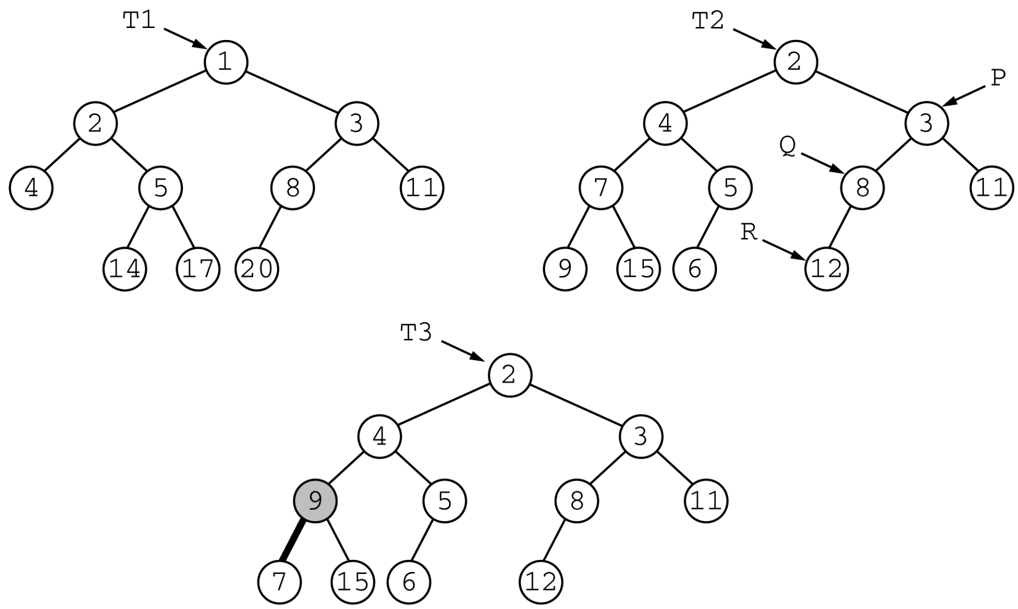
(c) Write function `RadixSort`, as started below.

In writing `RadixSort`, you may call functions `GetDigit`, `ItemsToQueues`, and `QueuesToArray` specified in parts (a) and (b). Assume that `ItemsToQueues` and `QueuesToArray` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `RadixSort` below.

```
void RadixSort(apvector<int> & L, int numDigits)
// precondition:  L.length() > 0; all values in L are positive;
//                the largest value in L has numDigits digits;
// postcondition: L contains the same list of values, sorted in
//                nondecreasing order
```

4. A *minheap* with no duplicate values is a binary tree in which the value in each node is smaller than the values in its children's nodes. For example, trees `T1` and `T2` are minheaps, but tree `T3` is not, because the shaded node has a larger value than one of its children (as shown by the thick line).



Consider the following representation for the nodes of a minheap.

```
struct HeapNode
{
  int val;           // value in this node
  HeapNode * left;   // left child
  HeapNode * right;  // right child
};
```

(a) Write function `MinChild`, as started below. `MinChild` returns a pointer to the child node of `T` that contains the smaller value (or `NULL` if `T` is a leaf node). If `T` has only one child, `MinChild` should return a pointer to that child. If `T` is `NULL`, `MinChild` should return `NULL`.

The following table shows the results of several calls to `MinChild`.

Function call	Return value
`MinChild(T2)`	`P`
`MinChild(P)`	`Q`
`MinChild(Q)`	`R`
`MinChild(R)`	`NULL`
`MinChild(NULL)`	`NULL`

Complete function `MinChild` below.

```
HeapNode * MinChild(HeapNode * T)
```

(b) Write function `IsHeapOrdered`, as started below. `IsHeapOrdered` returns `true` if `T` is a minheap and `false` otherwise.

`T` is a minheap under the following conditions.

- `T` is `NULL`, or
- `T` is a leaf node, or
- `T` is a nonleaf node, the value in `T` is smaller than the values in its children's nodes and its children are minheaps.

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

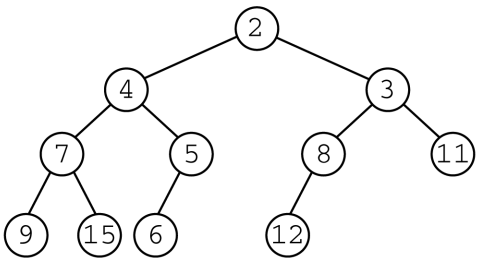
Complete function `IsHeapOrdered` below.

```
bool IsHeapOrdered(HeapNode * T)
```

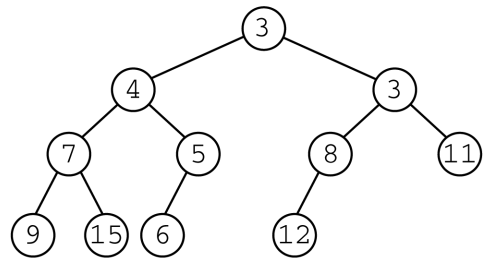
(c) The `RemoveMin` function removes the minimum value from a minheap. To do so, replace the value in the root node with the smaller of its children's values and then recursively call `RemoveMin` on the subtree rooted at the smaller child.

For example, the following diagram illustrates the steps to restore the heap after removing 2 from the root node.

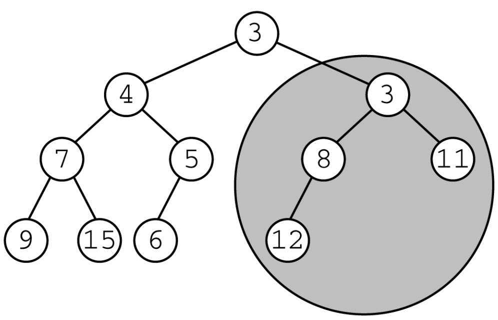
1. Initial



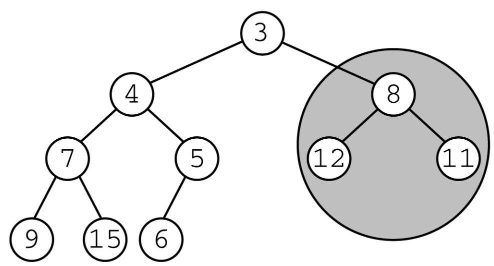
2. After replacing root node with smaller child's value



3. Subtree to be used for recursive call



4. After recursive call



Write function `RemoveMin`, as started below. `RemoveMin` should remove the minimum item from the heap, calling `delete` as necessary.

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

Complete function `RemoveMin` below.

```
void RemoveMin(HeapNode * & T)
// precondition: T is not NULL
```

END OF EXAMINATION