2000 Advanced Placement Program®
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 3,800 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 5,000 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®, the Advanced Placement Program® (AP®), and Pacesetter®. 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.

The College Board

COMPUTER SCIENCE AB

SECTION II

Time—1 hour and 45 minutes

Number of questions—4

Percent of total grade—50

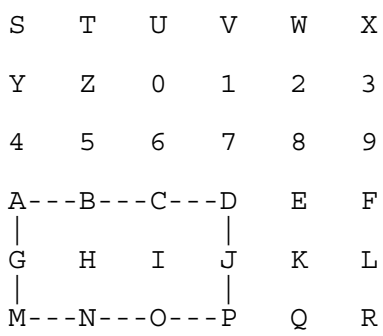
Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN C++.

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. One way of encrypting a word is to encrypt pairs of letters in the word together. A scheme to do this is to fill a 6 x 6 square with the 26 capital letters of the alphabet and the ten digits `'0'` through `'9'`. Each letter and digit appears exactly once in the square.

   To encrypt a letter pair, the rectangle formed by the two letters is used. Each letter of the original pair is replaced by the letter located on the same row and in the other corner of the rectangle. If both letters happen to be in the same row or column, the letters are swapped.

   For example, in the following arrangement `AP` is encrypted as `DM`.



   Consider the following declaration for a class that uses this scheme to encrypt a word.

```
struct Point
{
  int row;
  int col;

  Point();                          // default constructor
  Point(int newRow, int newCol); // sets row to newRow, col to newCol
};

class Encryptor
{
  public:
    Encryptor();
       // fills the matrix with the 26 letters of the alphabet
       // and the 10 digits '0' through '9'

    apstring EncryptWord(const apstring & word) const;
       // returns an encrypted form of the word

  private:
    apmatrix<char> myMat;

    apstring EncryptTwo(const apstring & pair) const;
       // returns an encrypted form of the pair

    Point GetCoordinates(char ch) const;
       // returns the coordinates of ch in the 2-dimensional array
};
```

(a) Write member function `GetCoordinates`, as started below. `GetCoordinates` takes a given letter or digit and returns its row and column in the 2-dimensional array. Assume that the parameter `ch` is a capital letter in the range `'A'` through `'Z'` or a digit in the range `'0'` through `'9'`.

The following example shows the point locations of character `ch` in the given matrix.

Encryptor.myMat						ch	Point coordinates	
S	T	U	V	W	X	P	row = 5	col = 3
Y	Z	0	1	2	3	8	row = 2	col = 4
4	5	6	7	8	9	M	row = 5	col = 0
A	B	C	D	E	F			
G	H	I	J	K	L			
M	N	O	P	Q	R			

Complete function `GetCoordinates` below.

```
Point Encryptor::GetCoordinates(char ch) const
// precondition:  'A' ≤ ch ≤ 'Z' or '0' ≤ ch ≤ '9'
// postcondition: returns the row and column number of the
//                location of ch in myMat
```

(b) Write member function `EncryptTwo`, as started below. `EncryptTwo` is passed a two-character string and returns an encoded two-character string.

The encoding of a letter pair is formed as follows.

1. If both letters are in the same row or column, swap the two letters.
2. Otherwise, find the other two corners of the rectangle formed by the two letters. Each letter of the original pair is replaced by the letter located on the same row and in the other corner of the rectangle.

For example, to encrypt a letter pair, say `NE`, look at the rectangle with corners `N` and `E`. The encrypted letter pair is `QB` because `Q` is the letter at the other corner on the same row as `N`, and `B` is the letter at the other corner on the same row as `E`.

```
S   T   U   V   W   X

Y   Z   0   1   2   3

4   5   6   7   8   9

A   B---C---D---E   F
    |           |
G   H   I   J   K   L
    |           |
M   N---O---P---Q   R
```

```
  Letters: BR  NE  ET  RE  TH  PR  GG
Encrypted: FN  QB  BW  QF  HT  RP  GG
```

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

Complete function `EncryptTwo` below.

```
apstring Encryptor::EncryptTwo(const apstring & pair) const
// precondition:  pair.length() is 2
// postcondition: returns an encoded two-character string
```

(c) Write member function `EncryptWord`, as started below. `EncryptWord` takes a word parameter and returns a string that contains the encryption of that word. Every two letters of the word are examined and encrypted by replacing the original letters with those located in the opposite corners of the rectangle formed by the two letters. If the original word contains an odd number of letters the last letter is unchanged.

The following are examples of encrypted words using the 2-dimensional array shown below.

```
S   T   U   V   W   X

Y   Z   0   1   2   3

4   5   6   7   8   9

A   B   C   D   E   F

G   H   I   J   K   L

M   N   O   P   Q   R
```

```
     Word:    COMPUTER    SCIENCE    STUDENTS
Encrypted:    OCPMTUFQ    UAKCOBE    TSVCBQST
```

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

Complete function `EncryptWord` below.

```
apstring Encryptor::EncryptWord(const apstring & word) const
// precondition:  word contains only capital letters 'A' through 'Z'
//                and digits '0' through '9'.
// postcondition: returns an encrypted version of word, in which every
//                two letters have been examined and encrypted by
//                replacing the original letters with those located
//                in the opposite corners of the rectangle formed by
//                the two letters. If the original word contains an odd
//                number of letters, the last letter is left unchanged.
```

2. An *infix expression* is an expression in which the operator is placed between the two operands, as in traditional algebraic notation. (For this problem, the only operations that we will consider are addition and multiplication.) Ambiguities in the expression are resolved by precedence rules (multiplication before addition) and left-to-right evaluation of operators of equal precedence. For example, in the following expression, the two multiplications would be evaluated first, then the additions would be evaluated from left to right.

```
3 * 4 + 5 * 2 + 8
```

A *postfix expression* is a sequence of numbers and arithmetic operators (`+`, `*`) that is evaluated from left to right by applying each operator to the two subexpressions immediately preceding it. For example, the following postfix expression is equivalent to the infix expression given above.

```
3  4  *  5  2  *  +  8  +
```

In this postfix expression,

the first `"*"` applies to `3` and `4`;	`3 * 4 = 12`
the second `"*"` applies to `5` and `2`;	`5 * 2 = 10`
the first `"+"` applies to the results of these two subexpressions,	`12 + 10 = 22`
and the last `"+"` applies to the last subexpression and the number `8`, resulting in the final evaluation.	`22 + 8  = 30`

Numbers and operators in the expression are called *tokens*. A token is either an integer or one of the operators `+` or `*`. Tokens are represented using the following definitions.

```
enum TokenType {PLUS, TIMES, NUMBER};
struct Token
{
  TokenType op;      // op == NUMBER means token represents a value
  int value;         // the value represented if op is NUMBER;
                     // otherwise, value is undefined
};
```

If a token has `NUMBER` for its `op`, then the token represents a value.

(a) Write overloaded `operator<`, as started below. If the precedence of the token `lhs` is less than the precedence of `rhs`, then `operator<` should return `true`. Tokens of type `PLUS` should have lower precedence than tokens of type `TIMES`, which should have lower precedence than tokens of type `NUMBER`.

Complete `operator<` below.

```
bool operator< (const Token & lhs, const Token & rhs)
```

(b) Write function `InfixToPostfix`, as started below. `InfixToPostfix` should fill the queue of tokens, `postQ`, with the postfix expression that is equivalent to the infix expression given in `infixQ`. The token at the front of `infixQ` represents the first token in the infix expression (`3` in the example above), and the token at the back of the `infixQ` represents the last token in the expression (`8` in the example above).

An infix expression can be translated to a postfix expression as follows. Use a stack to store operator tokens during processing. Process each token from the infix expression in turn.

- If a token is an operand (a number), then put it into the postfix expression.
- If a token is an operator and the stack is empty, then push the token on the stack. If the stack is not empty and the precedence of the token is less than or equal to the precedence of the token on top of the stack, then pop the token off the stack and put it into the postfix expression. Do this until the stack is empty or the precedence of the operator is greater than the precedence of the operator on top of the stack. Then push the current operator onto the stack.
- When the infix expression has been processed, put the operators remaining on the stack into the postfix expression.

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

Complete function `InfixToPostfix` below.

```
void InfixToPostfix(const apqueue<Token> & infixQ,
                          apqueue<Token> & postQ)
// precondition:  the sequence of tokens in infixQ represents a
//                valid infix expression using +, *, and integers;
//                postQ is empty
// postcondition: the sequence of tokens in postQ represents a
//                valid postfix expression equivalent to the
//                infix expression represented by infixQ

{
  apqueue<Token> tempQ = infixQ;
```

3. Consider changing the internal representation of a `BigInt` to a linked list implementation. The linked list nodes are implemented using the following declarations.

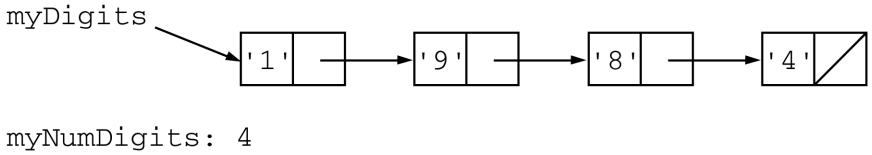
```
struct DigitNode
{
  char digit;
  DigitNode * next;
  DigitNode(char newDigit, DigitNode * newNext);
};

DigitNode::DigitNode(char newDigit, DigitNode * newNext)
  : digit(newDigit), next(newNext)
{}
```

In the declaration of the `BigInt` class, the private data member `myDigits` is changed from an `apvector` to a pointer to a `DigitNode`, as follows.

```
DigitNode * myDigits;
```

Note that the private data member `myNumDigits` continues to hold the number of digits in the `BigInt`.

The diagram below shows the linked list representation of the number `1984` stored as a `BigInt`.



`myNumDigits: 4`

Note that the least significant digit in this linked list implementation is stored in the last node of the list.

(a) Write a new implementation for private member function `AddSigDigit` to reflect the change to the linked list implementation of `BigInt`.

Complete function `AddSigDigit` below.

```
void BigInt::AddSigDigit(int value)
// postcondition:  value added to BigInt as most significant digit
```

(b) Write a new implementation for private member function `GetDigit()` to reflect the change to the linked list implementation of `BigInt`. For this function only, if the precondition is false then `GetDigit` returns zero.

Complete function `GetDigit` below.

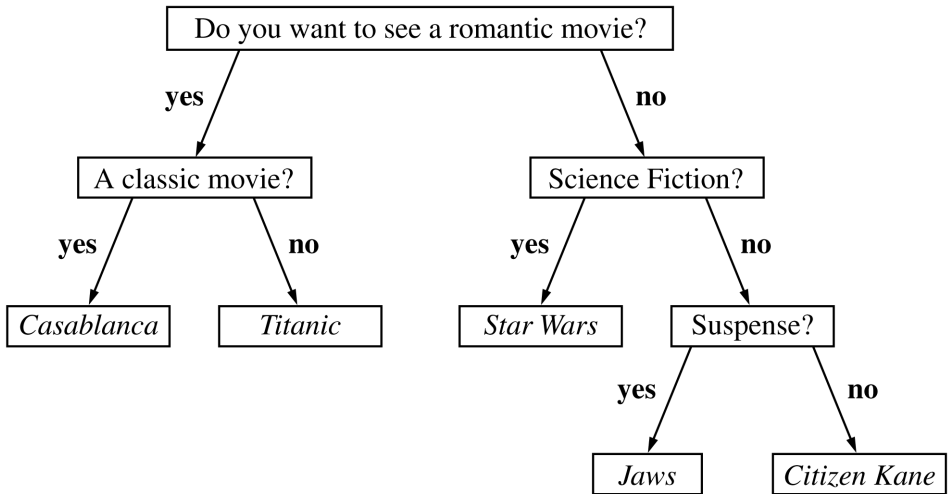
```
int BigInt::GetDigit(int k) const
// precondition:  0 ≤ k < NumDigits()
// postcondition: returns k-th digit (0 if precondition is false)
//                Note: GetDigit(0) returns the least significant digit
```

(c) Write the destructor for the `BigInt` class with the new linked list implementation. The destructor deletes all nodes and returns them to memory.

Complete the destructor below.

```
BigInt::~BigInt()
// postcondition: all the nodes in myDigits are returned to the heap
```

4. Consider a binary tree that implements a decision tree to give advice on some subject. Internal (nonleaf) nodes contain decisions to be made; leaf nodes represent the final advice or recommendation. For example, the following decision tree could be used to provide advice on choosing a movie.



Consider the following representation for the nodes of a tree.

```
struct AdviceNode
{
  apstring QorA;         // a question or an answer
  AdviceNode * yes;      // yes branch
  AdviceNode * no;       // no branch

  AdviceNode(const apstring & s); // constructor
};
```

Each node contains either a question or an answer (final recommendation). If `QorA` is a question, the `Advice` node has two non-empty subtrees, representing Yes and No answers to the question. If `QorA` is an answer, the node has no children. Function `IsQuestionNode`, whose specification is given below, is provided to determine whether a node is a question or an answer node.

```
bool IsQuestionNode(AdviceNode * T);
// precondition:  T is not NULL
// postcondition: returns true if T points to a question node;
//                otherwise, returns false
```

Do not write the body of `IsQuestionNode`.

(a) Write function `GiveAdvice`, as started below. If parameter `T` points to a question node, `GiveAdvice` should print the question, read a `'Y'` or `'N'` response from `cin`, and then continue down the appropriate path in the tree. (Assume that users always respond with either `'Y'` or `'N'`.) If parameter `T` points to an answer node, `GiveAdvice` should print the answer.

For example, if `T` is the tree given above and the user always answers `'Y'`, then the output would be as follows (where user input is given in boldface).

```
Do you want to see a romantic movie? Y
A classic movie? Y
Casablanca
```

In writing `GiveAdvice`, you may call function `IsQuestionNode` specified at the beginning of this question. Assume that function `IsQuestionNode` works as specified.

Complete function `GiveAdvice` below.

```
void GiveAdvice(AdviceNode * T)
// precondition: T is not NULL
```

(b) Write function `TracePath`, as started below. `TracePath` should search for the parameter `movie` in decision tree `T`, recursively tracing the path from the root of the tree to the movie. If the movie is in tree `T`, `TracePath` pushes the questions and answers along that path onto the parameter `pathStack` and returns `true`. If the movie is not in tree `T`, `TracePath` returns `false`. For question nodes, the question and its answer should be stored together as a single string in the stack.

For example, if `T` is a pointer to the root of the decision tree shown at the beginning of this question and `S` is an empty stack, then after the call `TracePath(T, "Jaws", S)`, `S` would contain the following elements.

```
top -> "Do you want to see a romantic movie? No"
       "Science Fiction? No"
       "Suspense? Yes"
       "Jaws"
```

In writing `TracePath`, you may call function `IsQuestionNode` specified at the beginning of this question. Assume that function `IsQuestionNode` works as specified. You may also assume that a given movie appears at most once in the decision tree.

Complete function `TracePath` below.

```
bool TracePath(AdviceNode * T, const apstring & movie,
               apstack<string> & pathStack)
// precondition: T is not NULL
```

END OF EXAMINATION