3. A `WordCollection`, shown in the class declaration below, stores a group of words. The collection may store multiple instances of any word. In this question, you will not implement any of the member functions of class `WordCollection`.

```
class WordCollection
{
  public:
    int Size() const;
       // returns the total number of items stored in the collection

    void Insert(const apstring & word);
       // adds word to the collection (duplicates allowed)

    void Remove(const apstring & word);
       // removes one instance of word from the collection if word is
       // present; otherwise, does nothing

    apstring FindKth(int k) const;
       // returns kth word in alphabetical order, where
       // 1 ≤ k ≤ Size()

    // other public member functions not shown

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

The public member function `FindKth` returns the *k*th word in alphabetical order from the collection (the word with rank *k*), even though the underlying implementation of `WordCollection` may not be sorted. The rank ranges from 1 (first in alphabetical order) to *N*, where *N* is the number of words in the collection. For example, assume that `WordCollection C` stores the following words.

```
{ "at", "bad", "all", "at" }
```

The following table illustrates the results of calling `C.FindKth(k)`.

| k | C.FindKth(k) |
|---|---|
| 1 | "all" |
| 2 | "at" |
| 3 | "at" |
| 4 | "bad" |

(a) Write free function `Occurrences`, as started below. `Occurrences` returns the number of times that `word` appears in `WordCollection C`. If `word` is not in `C`, `Occurrences` should return 0.

In writing `Occurrences`, you may call any of the member functions of the `WordCollection` class. Assume that the member functions work as specified.

Complete function `Occurrences` below.

```
int Occurrences(const WordCollection & C, const apstring & word)
// postcondition: returns the number of occurrences of word in C
```

(b) Write free function `RemoveDuplicates`, as started below. `RemoveDuplicates` removes all but one occurrence of `word` from `C`. If `word` is not in collection `C`, then `RemoveDuplicates` does nothing.

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

Complete function `RemoveDuplicates` below.

```
void RemoveDuplicates(WordCollection & C, const apstring & word)
// postcondition: if word is present in C, all but one occurrence
//                is removed; otherwise, C is unchanged
```

(c) Write free function `MostCommon`, as started below. `MostCommon` returns the word that appears most often in the collection. If there is more than one such word, return any one of them. You may assume that `C` is not empty.

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

Complete function `MostCommon` below.

```
apstring MostCommon(const WordCollection & C)
// precondition:  C is not empty
// postcondition: returns the word that appears most often in C;
//                if there is more than one such word,
//                returns any one of those words
```

4. 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`.

```
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
```

   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
};
```