CPS 100E **Exam 1 Solutions** **Fall 1997**

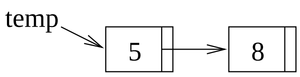
Rodger and Duvall

PROBLEM 1 : (*It's a Mystery To Me:* (6 pts))

Consider the following definitions and function *Mystery*.

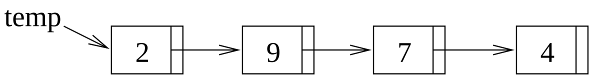
```
struct MysNode
{
   int number;
   MysNode * next;
};

MysNode * Mystery(MysNode * list)
{
   if (list != NULL && list->next != NULL)
   {
       return (Mystery(list->next->next));
   }
   else
       return list;
}

MysNode * temp;
```

1. Assume `temp` points to the following list.



   Show the list `temp` points to after the call: `temp = Mystery(temp)`

2. Assume `temp` points to the following list.



   Show the list `temp` points to after the call: `temp->next = Mystery(temp->next)`

3. Write a recurrence relation for the function *Mystery*. DO NOT SOLVE the recurrence relation. Assume the linked list has N nodes.

PROBLEM 2 : (*Race to Win:* (18 pts))

Consider the following definition and AthleteIterator iterator class. The AthleteIterator class contains information on different types of athletes. Instead of iterating over all athletes, you can specify a sport and iterate over athletes who only play that sport.

```
struct Player
{
   string name;
   string sport;
};

class AthleteIterator
{
 public:

   AthleteIterator();         // Constructor

   void First(const string &);
   void Next();
   bool IsDone();
   Player Current();

 private:
   void ReadInData();          // reads in Athletes to myAthletes

   Vector <Player> myAthletes;
   int mySize;                 // number of athletes in myAthletes
};
```

Here is a sample program that uses the AthleteIterator class to list out all the basketball players, followed by all the tennis players.

```
  AthleteIterator Duke;
  Player temp;

  for (Duke.First("basketball"); ! Duke.IsDone(); Duke.Next())
  {
     temp = Duke.Current();
     cout << temp.name << endl;
  }
  for (Duke.First("tennis"); ! Duke.IsDone(); Duke.Next())
  {
     temp = Duke.Current();
     cout << temp.name << endl;
  }
```

Complete the functions *First, Next, IsDone*, and *Current.* In writing these functions you may need to add code to the constructor and you may need to add items to the private section of the class. **Show these changes below.**

The function *ReadInData* reads in athletes and stores them in the Vector myAthletes in no particular order. (all tennis players may not be together, but rather spread throughout the Vector). You may assume *ReadInData* already exists, you **DO NOT** need to write it.

- Show additions to the private section of the class here *(3 pts)*.

```
private:
```

- Show additions to the constructor here *(3 pts)*.

```
AthleteIterator::AthleteIterator(): myAthletes(0), mySize(0)
{
   ReadInData();    // reads in and stores athletes in myAthletes






}
```

- Complete the function *First* below *(3 pts)*.

```
void AthleteIterator::First(const string & type)
// postcondition: sets iterator to first athlete of specified type.
{

}
```

- Complete the function *IsDone* below *(3 pts)*.

```
bool AthleteIterator::IsDone()
// postcondition: returns true if there are no more athletes of this type
//               to process, otherwise returns false
{

}
```

- Complete the function *Next* below *(3 pts)*.

```
void AthleteIterator::Next()
// postcondition: sets iterator to next athlete of specific type
{

}
```

- Complete the function *Current* below *(3 pts)*.

```
Player AthleteIterator::Current()
// precondition: IsDone() == false
// postcondition: returns current athlete
{

}
```

PROBLEM 3 : (*Multiplying Matrices:* (10 pts))

To multiply two matrices, A, an $m \times n$ matrix, and B, an $n \times p$ matrix, the product matrix $C = AB$ is an $m \times p$ matrix whose $[i][j]$ entry is given by

$$C[i][j] = A[i][0]B[0][j] + ... + A[i][n-1]B[n-1][j] = \sum_{k=0}^{n-1} A[i][k]B[k][j] \;\; for 0 \leq i \leq m-1, 0 \leq j \leq p-1$$

More intuitively, to the find the entry in the ith row, jth column of C, multiply the corresponding elements of the ith row of A and the jth column of B, and add each result.

For example, to find the entry in the first row, first column of C, take the first row of A and the first column of B, multiply the corresponding elements and add each result.

Here is an example, using two matrices of integers showing ($A * B = C$):

$$\begin{pmatrix} 2 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 5 \\ 3 & 4 & 5 \end{pmatrix}$$

In this example, $C[0][0] = A[0][0]B[0][0] + A[0][1]B[1][0] + A[0][2]B[2][0] = 2*1 + 0*0 + 1*1 = 3$

PART A *Doing the Math* (8 pts):

Write the function, with the header provided below, that multiplies the two matrices provided and places the result in a third matrix (also provided).

```
void MultiplyMatByMat (const Matrix<int>& a, const Matrix<int>& b, Matrix<int>& c)
// pre: matrices a, b, c are properly constructed
// post: c holds the result of multiplying a by b
{

}
```

PART B *Analyzing the Runtime* (2 pts):

What is the worst case running time (big-oh) of multiplying an $m \times n$ matrix by an $n \times p$ matrix?

PROBLEM 4 : (*Fishy Business:* (14 pts))

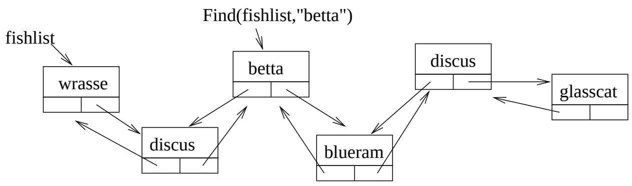
Consider the following definition for a node in a doubly linked list.

```
struct Node
{
   string fish;  // type of fish
   Node * next;  // forward pointer
   Node * prev;  // backward pointer
};
```

PART A *(6 pts)*:

Write the function *Find* whose header is shown below. *Find* returns a pointer to the first occurence of a specified type of fish. *Find* returns NULL if the type of fish does not appear in the given list.

In the example below, the result of the call `Find(fishlist,"betta")` is illustrated by showing a pointer to a node in fishlist with the value `"betta"`.



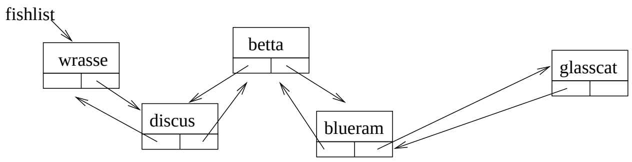
Complete function *Find* below the following header.

```
Node * Find(Node * list, const string & type)
// postcondition: returns a pointer to the first occuring node with "type" value
//                returns NULL if the type does not appear in the list.
{

}
```

PART B *(8 pts)*:

Write the function *RemoveDuplicates* whose header is shown below. *RemoveDuplicates* is given a linked list of fish and a type of fish, and removes all but one occurrence of this type of fish.

Let fishlist represent the list shown in Part A. The figure below shows one possible result of the call `RemoveDuplicates(fishlist,"discus")`. All but one node containing `"discus"` have been removed from the list.



In writing *RemoveDuplicates*, you may use the function *Find* from part A. Assume the function *Find* works correctly regardless of what you wrote for part A.

Complete function *RemoveDuplicates* below the following header.

```
void RemoveDuplicates(Node * & list, const string & type)
// postcondition: remove all but one occurrence of "type" from the list
{
```

```
}
```

PROBLEM 5 : (*Random Sentences:* (18 pts))

A text file stores a list of words that might be used in creating random sentences. Each line stores a number corresponding to a part of speech followed by a set of words of that part of speech. An example data file is shown below:

```
0 piano computer glass
1 ate moved broke
2 lazy heavy big dark smug
0 table tree dinosaurs
3 noisly happily sadly colorfully
```

Notice that there can be multiple sets of words for each part of speech, your program should combine each set of words (allowing duplicates) into a single collection. To help you, you can assume that the numbers given in the data file correspond directly to a vector index for that part of speech. So, in this case 0 = noun, 1 = verb, 2 = adjective, and 3 = adverb.

The data structure you will use to store these words is a vector of linked lists of strings. Each entry in the vector should represent the collection of words for a particular part of speech. This is given in the struct definition below:

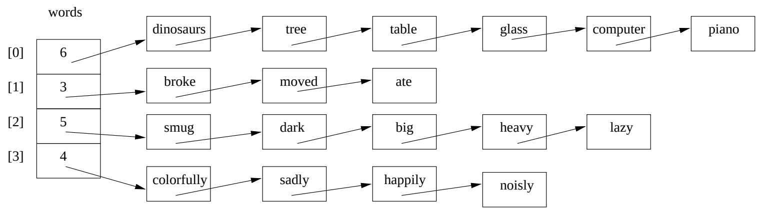
```
struct WordList
{
   int        numWords;    // count of words in linked list
   WordNode * list;

   WordList() : numWords(0), list(NULL)
   {
      // all data initialized
   }
};
```

The linked list represents the actual words read in from the data file. The following struct represents a single node in the list (and is held by the `WordList` struct):

```
struct WordNode
{
   string     word;
   WordNode * next;

   WordNode(string newWord, WordNode * newNext = NULL)
     : word(newWord), next(newNext)
   {
      // all data initialized
```

```
    }
 };
```

The diagram below shows the data structure after reading in the data file given above. Note, you may store the words in any order you choose, so your ordering may not be exactly that in the diagram.



PART A *Reading the Input* (12 pts):

Write the function *ReadInput*, with the header provided below, that reads in the data from the given input stream and fills in the given Vector of Linked Lists. You can assume that the Vector passed to the function has been constructed so that the numbers in the data file correspond to the vector indicies that will hold the words for that part of speech.

```
void ReadInput (istream & input,  Vector<WordList>& words)
// pre: input is open for reading, format as specified in problem
//      words is set up so that the Vector indicies corresponds to part of speech
// post: words holds data given in input, organized by their part of speech
{


   string line;
   while (getline(input, line))
   {


   }
}
```

PART B *Choosing a Single Random Word* (6 pts):

In this part of the problem, you will be writing a function that could be used by another function to create random sentences based on the words given in the data file. The following are example sentences that might be generated:

```
The big dark piano noisly colorfully moved the lazy big dark heavy computer.
```

Write the function *ChooseOneWord*, with the header provided below, that chooses a single word randomly from the list of words specified by the index. This function should use the `Dice` class to select a random word from the list and return it to the calling function. For example, if *words* is the Vector in the previous figure, then *ChooseOneWord(words,2)* might return `"heavy"`.

```
string ChooseOneWord (const Vector<WordList>& words, int index)
// pre: words holds lists of words organized by their part of speech
// post: a single word is returned from the list specified by index
{

}
```

Vector class

```
// for a vector of Items use Vector<Item>, e.g., Vector <int> intvector;
//
//                           note: Item must have a default constructor
//
// constructors:
//   Vector()                   -- default, vector of size 0 (no entries)
//   Vector(int size)           -- vector with size entries
//   Vector(int size,
//          Item fillValue)     -- vector w/ size entries all == fillValue
//   Vector(const Vector & vec) -- copy constructor
//
//   int Length()                -- returns size of vector (capacity)
//   void SetSize(int newSize)   -- resizes the vector to newSize elements
//                                  (can result in losing elements if
//                                   new size < old size)
//   void resize(int newSize)     -- synonym for SetSize
//
//   void Fill(Item fillValue)    -- set all entries == fillValue
//
//   operator =                   -- assignment operator works properly
//   operator []                  -- indexes both const and non-const vectors
//
//
//  examples of use:
//            Vector<double> dlist(100);       // a list of 100 doubles
//            Vector<double> dzlist(100,0.0);  // initialized to 0.0
//
//            Vector<String> slist(300);       // 300 strings
//
//            Vector<int> ilist;               // has room for 0 ints
```

Matrix class

```
// for a matrix of Items use Matrix<Item>, e.g., Matrix <int> intmatrix;
//
//                           note: Item must have a default constructor
//
// constructors:
//   Matrix()                   -- default, matrix of size 0x0
//   Matrix(int rows,int cols)  -- matrix with dimensions rows x cols
//   Matrix(int rows,int cols,
//          Item fillValue)     -- matrix w/ entries all == fillValue
```

```
//   Matrix(const Matrix & vec) -- copy constructor
//
//   int Rows()                  -- returns # of rows (capacity)
//   int Cols()                  -- returns # of columns (capacity
//
//   void SetSize(int rows, int cols) -- resizes the matrix to rows x cols
//                                   can lose entries if size is smaller
//                                   in either dimension
//
//   void Fill(Item fillValue)    -- set all entries == fillValue
//
//   operator =                   -- assignment operator works properly
//   operator []                  -- indexes const and non-const matrixs
//
//
//  examples of use:
//        Matrix<double> dmat(100,80);      // 100 x 80 matrix of doubles
//        Matrix<double> dzmat(100,80,0.0); // initialized to 0.0
//
//        Matrix<String> smat(300,1);       // 300 strings
//
//        Matrix<int> imat;                 // has room for 0 ints
```

Dice class

```
//  Dice(int sides) -- constructor, sides specifies number of "sides"
//              for the die, e.g., 2 is a coin, 6 is a 'regular' die
//
//  int Roll() -- returns the random "roll" of the die, a uniformly
//               distributed random number between 1 and # sides
//
//  int NumSides() -- access function, returns # of sides
//
//  int NumRolls() -- access function, returns # of times Roll called
//                   for an instance of the class

class Dice{
  public:
    Dice(int sides);            // constructor
    int Roll();                 // return the random roll
    int NumSides();             // how many sides this die has
    int NumRolls();             // # times this die rolled
  private:
    // not shown
};
```

Example Using String Stream

```
string str, name;
istrstream strIn(str);
strIn >> name;
```