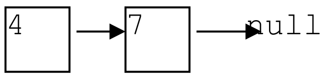
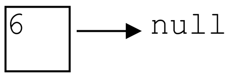
CIS 308 Exam 2 (50 points)
Tuesday, May 1, 2007 –OR-
Wednesday, May 2, 2007

Name:_______KEY__________

1. (6 pts) Write the declaration for the C++ class **`Table`**. This class stores elements in an array, and can add and access these elements using an array index. More than one element can be stored in the same array index, so each position in the array should be a linked list of values. It should have the following private instance variables:
   - A **pointer** for the dynamic array of linked lists. (Each array element will be a `Node*`.)
   - The **size** of the array

   You should also include:
   - A **constructor** that takes the size of the dynamic array, allocates memory for the array, and initializes the size variable to this array size
   - A **destructor** that releases the memory for each node in the table
   - **`add`** – takes an integer value and an index and adds the value to that spot in the array
   - **`contains`** – takes an integer value and returns whether it is in the array. This function does not take an index, so you should look through the entire array.

   For example, if the array size was 3 and we added 4 to index 0, 6 to index 1, and 7 to index 0, then the array should look like this:

Index	**Value**
0	
1	
2	`null`

You must use the `Node` class below to represent elements in the table:

```
class Node {
     private:
          int data;
          Node *next;
     public:
          Node(int elem) {data = elem; next = NULL;}
          friend class Table;
};
```

```
     class Table {
          private:
               Node** arr;
               int size;
          public:
               Table(int);
               ~Table();
               void add(int, int);
               bool contains(int);
     };
```

2. (11 pts) Implement the **constructor** and **each function** in the `Table` class. You do not need to implement the destructor.

```
Table::Table(int s) {
     int i;
     size = s;
     arr = new Node*[s];
     for (i = 0; i < size; i++) arr[i] = NULL;
}

void Table::add(int val, int index) {
     if (index < 0 || index >= size) throw "out of bounds";
     Node* temp = arr[index];
     Node* newnode = new Node(val);
     if (temp == NULL) arr[index] = newnode;
     else {
          while (temp->next != NULL) {
               temp = temp->next;
          }
          temp->next = newnode;
     }
}

bool Table::contains(int val) {
     int i;
     for (i = 0; i < size; i++) {
          Node* temp = arr[i];
          while (temp != NULL) {
               if (temp->data == val) return true;
               temp = temp->next;
          }
     }

     return false;
}
```

3. (3 pts) Explain (in words) what the **destructor** in the `Table` class would need to do to release all the dynamic memory.

   The destructor would need to loop through each linked list in the array. For each linked list, it would need to step through the list with a temporary `Node*`, deleting the nodes one at a time. After deleting every node in the list, the destructor would need to delete the array of node pointers.

4. (4 pts) Consider the following class:

```
class Message {
     protected:
          string msg;
     public:
          Message(string m) {msg = m;}
          virtual void printMessage(void) = 0;
};
```

   Write the class declaration for `EncryptedMessage`, which extends `Message`. It should contain an integer instance variable for the **rotation amount** of this encryption. It should also contain a **constructor** that takes values for the message and rotation amount. Finally, it should override the **`printMessage`** function.

```
class EncryptedMessage : public Message {
     private:
          int rotation;
     public:
          EncryptedMessage(string, int);
          void printMessage(void);
};
```

5. (8 pts) In this question, you will implement the `EncryptedMessage` class.
   a) (3 pts) Implement the **constructor** from `EncryptedMessage`.

```
EncryptedMessage::EncryptedMessage(string m, int r) : Message(m) {
     rotation = r;
}
```

   b) (5 pts) Implement the **`printMessage`** function from `EncryptedMessage`. `printMessage` should **print the encrypted message** using a rotation cipher. If the first character in the message is 'A' (which has ASCII value 65) and the rotation amount is 3, then the first encrypted character should be 'D' (which has ASCII value 68). If the next character is a 'Y' (ASCII value 89) with rotation amount 3, then the next encrypted character should be 'B' (ASCII value 66) – it should wrap back around to the beginning.

   You may assume that the message is made up of upper-case characters, which have ASCII values from 65-90. Recall that you can get the ASCII value for a character by casting the char to an int, and that you can convert an ASCII value to a character by casting the int to a char.

```
void EncryptedMessage::printMessage(void) {
     string result = "";
     int i;
     for (i = 0; i < message.length(); i++) {
          int ascii = (int) message[i];
          int ascii2 = ascii + (rotation % 26);
          if (ascii2 > 90) {
               ascii2 -= 26;
          }
          char newC = (char) ascii2;
          result += newC;
     }

     cout << result << endl;
}
```

6. (5 pts) Write the declaration for the C++ class `Double`. This class represents a double value with at most two digits before the decimal place and two digits after the decimal place (e.g., 12.34). It should have **four integer instance variables** – `dig1`, `dig2`, `dig3`, and `dig4` – that represent the four possible digits. (You should not store any other instance variables.) It should have a **constructor** that takes values for each digit initializes the variables. You should also overload the following operators:
   - ⑤ **`operator+`** - takes another `Double` object, and returns a new `Double` that is the sum of this `Double` and the function argument
   - ⑤ **`operator<<`** - adds the value of the `Double` to the output stream (like "12.34") and returns the updated stream (should be overloaded as a **non-member friend function**)

```
class Double {
     private:
          int dig1, dig2, dig3, dig4;
     public:
          Double(int, int, int, int);
          Double operator+(const Double&);
          friend ostream& operator<<(ostream&, const Double&);
};
```

7. (9 pts) **Implement the two overloaded operators** from #5. You do not need to implement the constructor. In the `operator+` function, you may assume that adding the two numbers will not result in a number with more than the four digits.

```
Double Double::operator+(const Double& d) {
     int new1, new2, new3, new4, carry = 0;
     new4 = dig4 + d.dig4;
     if (new4 > 9) {
          new4 -= 10;
          carry = 1;
     }
     new3 = dig3 + d.dig3 + carry;
     if (new3 > 9) {
          new3 -= 10;
          carry = 1;
     }
     new2 = dig2 + d.dig2 + carry;
     if (new2 > 9) {
          new2 -= 10;
          carry = 1;
     }
     new1 = dig1 + d.dig1 + carry;

     //assume new1 is not > 9
     Double result(new1, new2, new3, new4);
     return result;
}

ostream& operator<<(ostream& out, const Double& d) {
     out << d.dig1 << d.dig2 << ".” << d.dig3 << d.dig4;
     return out;
}
```

8. (4 pts) Suppose that we want to be able to call the function `getResults` **by reference** as follows, so that at the end of the function call, `area` holds the area of a square with side length `side`, and `perim` holds the perimeter of that square.

```
int area, perim;
int side = 4;

getResults(side, area, perim);
```

Write the C++ function `getResults` that will satisfy these requirements.

```
void getResults(int s, int& a, int& p) {
     a = s*s;
     p = 4*s;
}
```