COMP322: Assignment 1 - Winter 2012

Due at 11:30pm EST, 6 Feb 2012

1 Introduction

For this assignment, we will investigate using a *linked list* representation of a matrix. The purpose of this assignment will be the following:

- Give you practice with C-style pointers
- Give you a bit of practice with memory allocation and management (since C++does not have a built in garbage collector)
- See a couple ideas that are specific to C++
- Learn about an implement a commonly used computer science data structure.
- Build the foundation of a system we'll use later on.

We will not be exploiting any advanced features of the C++ language yet.

The assignment is designed to be challenging. If you are having trouble with it, please contact the instructor, a TA, or post on the discussion boards your specific issues. Collaboration is strongly encouraged as long as your final submission is your own individual work. Part of the purpose of the assignment will be to make sure you feel comfortable enough with the C language.

Unfortunately, as we only have 1 hour per week in class, it is likely we won't get to every topic for this assignment. In this case, there are a few options: a)google search, b)email instructors or TAs, c)post on discussion boards, d)come to office hours, e)chat with each other.

2 Background Information

2.1 Storing a sparse matrix

The most natural way to store a matrix into a computer is create a two-dimensional array where one index represents a row and another index represents a value.

For example:

```
int main() {
int numbers[2][2];
numbers[0][0] = 1;
numbers[0][1] = 2;
numbers[1][0] = 3;
numbers[1][1] = 4;
}
```

could store the matrix:

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

While this is a very natural way to store a matrix, there are some applications for which this is not the most efficient means of storing a matrix.

Consider, in a movie recommender system such as Netflix uses, how to store a matrix representing what a user thinks of various items that he or she has rated. One way to do this would be to have every column represent a specific movie and every row represent a particular user. For example consider the following matrix:

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

Each column represents a particular movie and each row represents a particular user. For example, suppose the three columns represent the movies *Star Wars*, *Indiana Jones*, and *Strangers on a Train*. Suppose that the three rows represent the users (who are all robots) **Watson**, **Honda**, and **Marvin**.

In this case we'd interpret the 1 as meaning that Watson gave the movie *Star Wars* a 1. The 2 in the next column would represent that Watson gave the movie *Indiana Jones* a 2, etc. We interpret 0 as the user did not provide a rating.

Consider if we are designing a large scale system with a million users and a million movies in it. This would involve creating a 1000000 by 1000000 matrix of ints. Note that since most users only have the patience to rate dozens or at the most hundreds of movies, that we could conclude almost all of these values would be 0 (since 0 represents the movie wasn't rated). This is a huge waste of memory, and in fact if you try to create an array this large in C++, you may very well get an error (try this)

```
int largeMatrix[1000000][1000000]; //most likely causes compiler error
```

Since the matrix is very *sparse*, a smarter way to represent these matrices is to store a list of individual *elements*. Each element will have 3 things in it, a row number, a column number, and a value. (Note that in implementation row numbers and column numbers should start from 0 like in an array.) If a specific row number and column number pair appears in the list, then the value of the matrix is whatever the value is. If a specific row number and column number does NOT appear in the list, then the value of the matrix is 0

For example, the above 3x3 matrix would be stored:

row	*column*	*value*
0	0	1
0	1	2
0	2	3
1	0	2
2	0	2
2	1	4

Unfortunately, this sort of table is difficult to implement in practice because it's size is constantly changing. This brings us to the next idea:

2.2 Linked List

A *linked list* is a data structure used in computer programming that can change sizes quickly. (You can read more about it at `http://en.wikipedia.org/wiki/Linked_list`)

The idea is to define a type which contains a *link* to another element of the same type. Then you can keep track of the entire list by maintaining a pointer (in Java terms a reference) to the first element in the list. You can then access the 2nd element in the list by following the *link* from the first element to it's next element of the same type. If the value is NULL, then this means we have reached the end of the list. Each element in the list is often referred to as a *node*

To do this in C++(or C), you should define a new type by defining a `struct` which consists of the data you need to store as well as a pointer to the next element in the list. For example, to create a linkedlist of ints, you would define a new type:

```
struct IntNode {
       int value; // stores the actual value
       IntNode* next; //stores a pointer to the next element in the list
};
```

The equivalent definition in Java would be:

```
public class IntNode {
     public int value; //elements of a struct are public by default
     public IntNode next; //note that in Java an IntNode would be a reference by default
}
```

A couple key differences between the C style version and the Java style version:

- By default, everything in a struct in C is public. In Java by default it's package protected (closer to private)
- In Java, there is no way to create an IntNode that is NOT a reference type. In C, you can store an IntNode directly if you wanted, thus the * is necessary to denote that you want to create a pointer.
- In C++NULL is equivalent to null in Java
- You need a ; after the final closing brace in C++

In C++, C, and Java, the dot operator means the same thing. If you have an expression that evaluates to an element of a certain struct or class, you can access it's public properties via a dot operator. For example:

```
IntNode x;
x.value = 3;
x.next = NULL;
```

It is important to note though that in C or C++, by default the type is NOT a reference or pointer. In fact, with every variable in C or C++, you can specify whether you want to store it directly or using a pointer.

If we want to *traverse* a linked list (that is, if we want to visit every value), we'll use a loop and keep following the *next* pointer. We'll then check to make sure that current node is not null. For example:

```
//assume that root is a pointer to the head or first element of a linked list
IntNode* current = root;
while (current != NULL) {
     //do something with current. for example, print it's value:
     cout << current->value << " "; //see later for explanation on ->
     current = current->next; //update pointer
}
```

2.3 Pointers in C and C++

We'll talk about this in class, but it is useful to outline a few tips on using pointers in C.

In general, in C or C++, any type can be stored as a value or as a pointer. Storing something in C or C++as a pointer is equivalent to storing a reference in Java.

To create a pointer variable in C, you add a * to the type. For example, to create a pointer to an int, you would write:

```
int* x;
```

(One technical thing: to create 2 pointers at once you actually have to write `int *p, *q;` That is, a star for each variable)

To make this useful, you need to specify which address you want the int pointer to point to. There are two ways to do this:

1. Use the & operator to get the address of an existing int variable.

   ```
   int* p;
   int x = 3;
   p = &x; //p refers to the address of x
   ```

2. Use the *new* operator to create space in the computers memory free store to store an integer.

   ```
   int* p = new int;
   ```

   Of course in this case we have no idea what value is stored at the location pointed to by p. Since we used the new operator, it will be necessary to, before our program exits, use the `delete` operator as well, to assure there is no memory leak.

Any time you want to access the data pointed to by a pointer, you can use the * operator. This is known as *dereferencing a pointer*.

```
int x = 3;
int* p = &x; //p points to the variable x now
(*p) = 1; //change the value pointed to by p to be 1
cout << x ; // prints 1 now
```

When you have a pointer to a struct, you can still use the * operator to deference it.

```
IntNode foo;
IntNode* fooPointer = &foo;
(*fooPointer).value = 3;
```

In the above `fooPointer` is an expression of type `IntNode*`, this means it can be dereferenced (since it is a pointer). `*fooPointer` thus has the type `IntNode`. Since `*fooPointer` is an `IntNode` that means it has a property `value` and we can set it equal to 3. Note that the parenthesis ARE required as if you omit them and write simply `*fooPointer.value`, it will be interpreted as `*(fooPointer.value)`

Because this is used very frequently, a little of what's known as *syntactic sugar* (or perhaps more accurately in this case syntactic iodine) was introduced. If you have a pointer `p` to a struct or class in C++, you can write:

```
p->value = 3;
```

Where the `->` means the same thing as the properly parenthesized * and .

Since there is no automatic garbage collection in C++, you need to `delete` the data pointed at by a pointer when you are through with it. If the pointer is called `p` you can do this by writing `delete p;` You have to be careful though, because if `p` is NULL or if `p` has already been deleted you'll have problems. For example:

```
int *p = new int;
int *q = p; //creates variable q and makes q and p aliases
delete p;
delete q; //EEK! q has already been deleted. Program crashes!
```

In theory, every `new` should be paired with a `delete`.

3 Assignment Requirements

On this assignment, you will, in C++, implement a linked list in order to represent a sparse matrix. On a future assignment, we will expand this idea to implement a movie recommendation system.

You should do the following (make sure to read the section on headers that follows):

1. Define a new `struct MatrixElement` which contains 4 properties: 3 ints representing the row, column, and value of a spot in a matrix. The fourth property should be a pointer to a MatrixElement

2. Write a method `compareTo` which takes as input a `const MatrixElement& me1` and a `const MatrixElement& me2` and compares which element occurs first. If `me1` occurs before `me2`, then it returns -1, if `me2` occurs before `me1` it returns 1, if they occur at the same place, it returns 0. When comparing two elements we determine which one comes first based on the row number of the data. If the row numbers are equal then we determine it based on the column. (In other words if you printed the matrix by row, from left to right, which element would come first)

   Hint: The method header for this should look like:

   ```
   int compareTo(const MatrixElement& me1, const MatrixElement& me2)
   ```

   Although `me1` and `me2` are both references, they can be accessed "normally" with commands such as `me1.row`

3. Next define a function `append` which takes as input a pointer to the first `MatrixElement` in the list, and 3 ints, representing the row, column, and value of the new `MatrixElement` to add. The method should return a `MatrixElement*`. The method should add the new element (produced by row, column, and value) to the end of the list represented by the `MatrixElement*` passed as input. It should then return the result. **Clarification: By "result" I mean the head of the linked list. If the linked list is not empty to start (i.e. the pointer it takes in is not NULL) then this will just be the same as the original pointer. However, if it is NULL to start, then it will be the new element. The point of this is so you can call this method and write things like:**

   ```
   MatrixElement* root = append(NULL, 1,2,3);
   root = append(root,4,5,6);
   root = append(root,7,8,9);
   ```

   A few tips:

   (a) You must use the new operator to create your MatrixElement. If you don't you will run into issues after your method finishes.

   (b) You will need to double check that the list passed in as input is not NULL.

   (c) If the list passed in as input is NULL, then you can simply create a new MatrixElement and return a pointer to it.

   (d) If the list passed in as input is not NULL, then you should follow the links from the element passed in as input to the end (using a condition to check whether `element->next` is equal to NULL). Once you see that the next element is NULL, you can set the next element to be your newly created element.

4. A method `print` which takes as input a `MatrixElement* root` and prints all the contents of the linked list as a linked list. It should print for each element in the linked list, the row, the column, and the value.

5. A method `getMaxColumns` which takes as input a `MatrixElement* m`. The method should traverse the linked list by following the `next` pointers, and return the largest column value of any node. If `m` is NULL, you method should return a negative value.

6. A method `deleteFirst` which takes as input a `MatrixElement* root` and removes the first element of the list. It should then return what was previously the second element in the list (and now will be the first element). Your method **MUST** obey proper memory management by properly deleting the pointer to the old root. If `root` is NULL, the method should return NULL.

7. A method `deleteList` which takes as input a `MatrixElement* root` and removes every element in the list. You must make sure you perform proper memory management and do not create any memory leaks in doing this. Your method should return `void`

8. A method `getLast` which takes as input a `MatrixElement* root` and returns a pointer to the last MatrixElement in the linked list. If `root` is NULL, the method should return NULL

9. A method `insert` which takes as input a SORTED `MatrixElement* root` an `int row`, an `int column`, and an `int value`. The method should insert a new element, consisting of the data row, column, and value into the linkedlist represented by `root`. It should then return this. Note that your method should handle cases where `root` is NULL.

   Your method should use the `compareTo` method that you wrote above to figure out the proper place to insert the element.

   Your method may assume that the `root` is properly sorted to being with.

   If an element is already in the linked list, your method should ADD the result. For example, if you are trying to store a new element into row 1 and column 1 with value 2, and there already is an element with row 1 and column 1 that has value 4, you should end up with a linkedlist with an element having row 1, column 1, and value 6. (The size of this linked list should not change.)

   Hint: Think about how you would do this problem if you were working with an array instead of a linked list

10. A method `add` which takes as input a `MatrixElement* m1` and a `MatrixElement* m2`. It should add these two matrices together and return a pointer to the result. You can do this by using the insert method you have written above (since it will add results). (This method should not be very long.)

Header File

You should follow best practices by putting all necessary function header definitions into an appropriate .h header file. You should also use the appropriate preprocessor command to make sure there is no problem if the .h is included twice. That is, all code in the .h header file should be surrounded by something like:

```
#IFNDEF MATRIX_LINKED_LIST_H
#DEFINE MATRIX_LINKED_LIST_H
     code goes here
#ENDIF
```

4 Testing your code

It might be a bit difficult to test your add method of matrices as the format of the `print` method above is not user friendly. You may find the following method useful. Note that this method will only properly print the matrix if the `root` pointer points to a linked list already sorted in ascending order based on the `compareTo` definition.

```
void printMatrix(MatrixElement* m1) {
        if (m1 == NULL) {
                return;
        }
        int max = getMaxColumns(m1);
        int lastRow = 0;
        int lastColumn = -1;

        while (m1 != NULL) {
                //first print an appropriate number of 0s.
                if (lastRow != m1->row) {
                        //finish the line off
                        printZeros(max - lastColumn);
                        cout << endl;

                        //print complete rows of zeros
                        for (int i=0; i < m1->row - lastRow - 1; i++) {
                                printZeros(max);
                                cout << endl;
                        }

                        printZeros(m1->column);
                        cout << m1->value << " ";
                }
                else {
                        printZeros(m1->column - lastColumn - 1);
```

```
                        cout << m1->value << " ";
                }

                lastRow = m1->row;
                lastColumn = m1->column;
                m1 = m1->next;
        }

        //now just finish printing the column
        for (int i= lastColumn; i < max; i++) {
                cout << "0 ";
        }

        cout << endl;
}
```

For your convenience, this method also appears as a .cc file on the course website. The method comes with no warranties however, so please check the course website in case any mistake in the code is discovered.

5 Using/installing g++

g++ is the GNU C++ compiler. A recent version (4.3.2) is available on most of the Ubuntu lab computers. Our solution was developed using version 4.4.1.

We build our solution using the command line:

```
g++ -Wall -o MatrixLinkedList.exe MatrixLinkedList.cc
```

To do this, you'll have to add a main method. If you don't want to write a main method (you should as it's the best way to test things!), you could write We build our solution using the command line:

```
g++ -Wall -o MatrixLinkedList.o MatrixLinkedList.cc
```

to compile.

If you want to install g++ on your system, you should be able to find instructions around the web.

For the Mac, the instructions here look reasonably accurate and up-to-date:

`http://www.edparrish.com/common/macgpp.php`

For PC/Windows systems, there are multiple choices, but the Cygwin environment mimics most of the Linux command line on Windows, so it is a good option:

`http://www.cygwin.com`

For Ubuntu (and possibly Debian) users, you can install the free g++ package with the following command:

```
sudo apt-get install g++
```

If you have trouble, or you're running some other operating system, let us know and we'll try to help you.

6 Submitting your assignment

What To Submit

`MatrixLinkedList.h` Here you should put all function headers.

`MatrixLinkedList.cc` Here you should put all code. You should include the header file.

`Confession.txt` (optional) In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.