COMP 322: Assignment 1 - Winter 2014

Due at 11:30pm, Feb 4th 2014

1 Introduction

For this assignment, we will investigate *the adjacency list* implementation of graphs. This will provide a bit of practice with C-style pointers/references and with memory allocation and management. Moreover, you will be exposed to a few commonly used computer science data structures.

Notes:

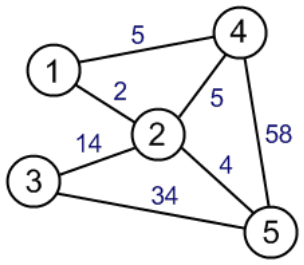
- It is normal to have trouble with this assignment, as it is intended to be challenging. Collaboration is strongly encouraged (but make sure your submission reflects individual work). Also, the Discussion Board, me and the TAs are there for you to use.
- It might happen that some topics in this homework won't be covered in class. This is normal as we only have one hour of lecture per week. You should not have trouble finding out answers in C++ online tutorials, and as always, don't forget to e-mail us or discuss with other students.
- We provide a `Makefile` and a header file `graph.h`. Your submission should compile using the `make` command on Trottier (3rd floor) machines when run on the original `Makefile`, `graph.h` and your submission `graphStudent.cpp` and `edge.h` (see details later).

2 Background on graphs

You are surely familiar with social network services like Facebook, Google+, Twitter etc. These and other complex networks are stored and processed using a data structure that some of you are already familiar with, and the others will definitely learn about in the near future: graphs. By this, we don't mean "graphs" as in plotting a function or a correlation. Rather a graph consists of

- A set of vertices V, indexed by integers $0, 1, 2, 3, 4, ..., n$
- A set of edges E, where each edge is a pair of vertices (i, j). This conveys the information that there is some significant relationship between nodes i and j.

Depending on the application that uses this data structure, there might be additional data/information attached to each edge/vertex. We will develop on this idea later this semester, but for this homework we consider the case where each edge comes with an integer value called a *weight* which describes how strong the relationship between two edges is.



What is the natural way to store graph data? A matrix containing the weights between each pair gives a straight forward solution. For the example above, we would have

$$\begin{pmatrix} 0 & 2 & 0 & 5 & 0 \\ 2 & 0 & 14 & 5 & 4 \\ 0 & 14 & 0 & 0 & 34 \\ 5 & 5 & 0 & 0 & 58 \\ 0 & 4 & 34 & 58 & 0 \end{pmatrix}$$

What would be the outcome if we are trying to store a large scale system, where the graph stores "connections" between millions of identities (i.e. vertices)? That would need a 1000000 by 1000000 matrix of ints, which seems quite wasteful given that this matrix is sparse (each vertex might be connected to a few (tens or hundreds) other vertices). Moreover, you can try your first line of code and see how C++ would handle such a matrix:

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

Fortunately, there are many alternatives to this trivial and non-efficient storage method. We consider the *adjacency list* implementation: for each vertex i we only store a list of all edges that connects i to some other vertex. We will implement these using the LinkedList data structure, which provides the flexibility that we'll apreciate in future assignments. Those of you not familiar with LinkedLists should read the appendix.

3 Assignment Requirements

On this assignment, you will implement an adjacency list implementation of a graph data structure. We will use this later in the course.

Please add all your code (except the `Edge struct`) in a file named `graphStudent.cpp`. The `Edge struct` should be added in a header file under the name `edge.h`. Test your code using these and the other files provided: `graph.h` and `simpleTest.cpp`. The TAs will test your code by compiling your submission using the provided `Makefile` and an additional `gradingTest.cpp` (not provided).

You are responsible for the following (make sure they respect the header definitions in `graph.h`):

1. Define a new `struct Edge` which contains 4 properties: two ints representing the two vertices connected by the edge, an int representing the weight of the edge, and a pointer to another `Edge`. This **should be defined** in `edge.h`.

2. Define a method `compareTo` which takes as input two edges `edge1` and `edge2` and compares the vertices that these connect. If the two edges connect the exact same vertices, the method should return 0. If the two edges have one vertex in common, then we compare the other two vertices. If the two edges don't share any vertex, then we compare the smaller vertex in each edge. That is:

$$(1,2) = (1,2) \quad (1,2) < (2,3) \quad (3,5) > (3,4) \quad (2,1) < (3,1)$$

If `edge1` $<$ `edge2`, your method should return -1, and if `edge1` $>$ `edge2`, your method should return 1.

3. Define a function `append` which takes as input a pointer to the first `Edge` in an adjacency list, three ints representing the vertices to be connected by a new edge and the weight of a new edge to be added to the list. The method should add the new `Edge` to the end of the list. The method should return a pointer to the first `Edge` in the list (i.e. the head of the linked list). If the list is `NULL` at the beginning, then the head of the list should be the newly created `Edge`.

4. A method `print` which takes as input a `Edge*` and prints all the contents of the linked list starting at the input `Edge*`. It should print for each element in the linked list, the vertices and the weight of the edge.

5. A method `deleteFirst` which takes as input a `Edge*` and removes the first element in 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`.

6. A method `deleteList` which takes as input a `Edge*` 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`.

7. A method `organizeList` which takes as input a `Edge*` and an `int` representing the number of vertices in the graph. It goes through the input list of edges and organizes it into separate lists for every vertex. That is, it creates an array where at index i we store a list of all edges that connects vertex i to some other vertex j. Your method should return the array, after all edges in the initial list has been transfered in the appropriate list of the array mentioned earlier. For the graph given as an example earlier should return the following:

0	`NULL`
1	$[(1,2,2),(1,4,5)]$
2	$[(2,1,2),(2,3,14),(2,4,5),(2,5,4)]$
3	$[(3,2,14),(3,5,34)]$
4	$[(4,1,5),(4,2,2),(4,5,58)]$
5	$[(5,2,4),(5,3,34),(5,4,58)]$

8. A method `printOrganized` which takes as input a graph that has been organized (i.e. output of `organizeList` of type `Edge**`) and prints line by line the neighbours of each vertex in the graph. For the graph given as an example, it should print (NOTE that the order of the neighbours should not matter):

```
Vertex 1 -> 4:5, 2:2
Vertex 2 -> 1:2, 4:5, 3:14, 5:4
Vertex 3 -> 2:14, 5:34
Vertex 4 -> 1:5, 2:5, 5:58
Vertex 5 -> 4:58, 2:4, 3:34
```

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 EDGE_H
#DEFINE EDGE_H
     code goes here
#ENDIF
```

4 Appendix

4.1 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. For details on how out solution is built, please see the Makefile. Note that to simply use the command `make` you have to add a main method to your `graphStudent.cc` file. If you dont want to write a main method (you should as its the best way to test things!), you can check compile time errors by using the command `make graphAdjList.o`.

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.claremontmckenna.edu/pages/faculty/alee/g++/g++_mac.html`

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 youre running some other operating system, let us know and well try to help you.

4.2 LinkedList

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 its 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 `int`s, 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 its 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 its 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), well use a loop and keep following the next pointer. Well 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 its value:
     cout << current->value << " "; //see later for explanation on ->
     current = current->next; //update pointer
}
```

4.3 Pointers in C and C++

Well 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 whats 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 youll 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`.