Assignment 3

Version 2a

Introduction

Your task is to implement Prim’s algorithm for undirected graphs, using a binomial heap as the basis for a priority queue.

Input

Your program will process a file containing a description of a graph. A graph description contains an arbitrary number of edge descriptions. An edge description consists of two vertices (optionally followed by a weight) followed by a semicolon. A vertex is simply a non-negative integer. If a weight is omitted, a weight of 1 should be assumed. A weight is a non-negative integer.

The file should be free format; whitespace may appear anywhere. Here a sample graph description:

```
1 5 ;
2
10
23 ;
214 33 1
;
```

which is equivalent to:

```
1 5 ;
2 10 23 ;
214 33 1 ;
```

In this example there are six vertices, named 1, 2, 5, 10, 33, and 214, and three edges, 1 to 5, 2 to 10 and 214 to 33, with weights 1, 23, and 1, respectively.

The name of your executable must be `prim` and the name of the file describing the graph will be passed to your program as a command line argument, as in:

```
$ cat g1
1 2 1 ;
2 3 2 ;
3 1 3 ;
$ prim g1
[output appears here]
```

Your program should interpret the graph description as an undirected graph and should report the minimum spanning tree with the first vertex mentioned in the graph description as the source vertex. Any unconnected vertices are ignored.

Output

The output of your program should be a spanning tree, displayed as a breadth-first traversal followed by the tree’s total weight. Here is an example display:

```
$ cat g2
10 0 9 ;
6 7 11 ;
5 9 1 ;
4 8 7 ; 10 8 6 ;
11 12 2 ;
1 6 5 ;
0 6 10 ; 0 5 3 ; 0 4 8 ; 0 3 2 ; 0 1 4 ;
$ prim g2
0: 10
```
Each level of the traversal starts with the level number (level 0 is the first level), followed by a colon, a space, and a space separated list of vertex descriptions. A vertex description is the vertex followed by its breadth-first-search predecessor (in parentheses) followed by the weight of the edge from the predecessor to the vertex in question. The vertex descriptions in a level are to be ordered by increasing vertex number. You must follow the format exactly as diff will be used to assess your output.

Options
Your executable must handle a -v option, as in previous assignments.

Program organization
You must implement the your binomial heap as a separate module named binomial.c and binomial.h (http://beastie.cs.ua.edu/cs201/assign-binomial.html). Likewise, you will need a vertex class (http://beastie.cs.ua.edu/cs201/assign-vertex.html) and an edge class (http://beastie.cs.ua.edu/cs201/assign-vertex.html).
You may use my vertex and edge classes:

```
wget beastie.cs.ua.edu/cs201/testing/3/vertex.c
wget beastie.cs.ua.edu/cs201/testing/3/vertex.h
wget beastie.cs.ua.edu/cs201/testing/3/edge.c
wget beastie.cs.ua.edu/cs201/testing/3/edge.h
```

Modifications to existing classes
You will need to add the following methods to the DLL class:

- `void removeDLLall(DLL *)` - sets the head and tail to null, the size to zero. Only freeing of list nodes, not values, is to be done.
- `void *removeDLLnode(DLL *,void *)` - removes the given node from the linked list, return the value of the removed node
- `void firstDLL(DLL *)` - start a DLL iterator, setting an object variable to point to the head node
- `void lastDLL(DLL *)` - start a DLL iterator, setting an object variable to point to the tail node
- `int moreDLL(DLL *)` - returns true if the current node is not null, false otherwise
- `void nextDLL(DLL *)` - moves the current node to the next node in the list
- `void prevDLL(DLL *)` - moves the current node to the previous node in the list
- `void *currentDLL(DLL *)` - returns the value at the current node

You will also need to modify the `insertDLL` method to return the newly created node holding the inserted value. Return the node as a `void *` pointer.

`decreaseKey` in \( \log n \) time
In order to decrease the value of a vertex key quickly, the vertex object needs to hold a pointer to the node in the binomial heap that holds the vertex. Suppose that field in a vertex object is called `owner`. Then, when inserting a vertex into the heap, one saves the return value of `insertBINOMIAL`:

```
setVERTEXowner(v,insertBINOMIAL(heap,v));
```

By saving this information, when the key of a vertex \( v \) is decreased, the heap can be informed efficiently:

```
setVERTEXkey(v,w);
decreaseKeyBINOMIAL(heap,getVERTEXowner(v),v);
```

Also, when a `decreaseKey` operation is called, the vertex may bubble up a subheap. In such cases, the binomial heap node holding the vertex will change. To keep up with these changes, an update function is passed to the binomial heap constructor:
static void
update(void *v, void *n) // v is a vertex, n is a binomial heap node
{
    VERTEX *p = v;
    p->owner = n;
}

Constraints
In order for your program to run on a randomly created graph, if an edge is given more than once, ignore subsequent occurrences. If a \( u, v \) edge is given, also ignore \( v, u \) edges, since the presence of a \( u, v \) edge implies the presence of a \( v, u \) edge.

You must use an adjacency list for storing the graph. You must be able to identify a duplicate edge in \( \theta(\log E) \) time. You must be able to add a non-duplicative edge to the adjacency list in \( \theta(\log V) \) time. Use AVL trees to store edges and vertices to achieve these time bounds.

If more than one vertex is eligible to be added to the growing minimum spanning tree at any given point, chose the vertex with the larger vertex number.

You cannot use any fixed-sized data structures for storing information about the graph. You many not use built-in data structures (other than arrays for sorting the display output).

All operations must run as efficiently as commonly expected.

When given an empty graph, the executable should display the word **EMPTY** followed by a newline.

Other details
You must implement your program in C. Only the most foolish student would not recompile and thoroughly test the implementation on a Linux system.

You must provide a makefile which responds properly to the commands `make`, `make test`, `make valgrind`, and `make clean`. The `make` command must compile the `prim` executable no errors or warnings and it must compile with the highest level of error checking (the `-Wall` and `-Wextra` options plus `-std=c99`). The `make test` command should run your program through some test files of your choosing. The `make valgrind` command should run your tests through `valgrind`. The `make clean` command should remove all intermediate files, including the executable `prim`.

Documentation
All code you hand in should be attributed to its author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Grading
You must pass all tests for your program to be graded.

Handing in the application
For preliminary testing, run a `make clean` and then send me all the files in your directory by running the command:

```
submit cs201 lusth test3
```

For your final submission, run a `make clean` and use the command:

```
submit cs201 lusth assign3
```

Again, your implementation may be developed on other hardware and operating systems, but it must also compile and run cleanly and correctly on a Linux system.