CSE 2320 - Homework 8 NAME: Gabriel de Sá 1001676832

Total Points 117 = 100 (required) + 17 (bonus). For this homework any point past 100 will be bonus. Updated 11/21/19

P1. (100 pts) Write a program that does the following:

1. Reads a graph. It must work with input redirection. (That is, read from the user, not explicitly from the data files.)
   You can assume the graph is undirected.
   You must represent the graph using a 2D matrix for the edges.
   The graph will have names for vertices. See people1.txt. If no special names are given, numbers 0 to (N-1) will be used. You should treat the numbers as if they were strings, NOT as a special case.
2. Prints the graph to verify it was loaded correctly.
   It prints: weighted, N, the mapping between the vertex *name* and the internal representation for that vertex which will be a number between 0 and N-1. E.g. see in run_people1.txt.

```
0-Sam
1-Thomas
2-Judy
```

   It also prints the matrices for the edges showing the row and column indices and using formatted printing with 4 reserved spaces (so that the numbers in the table align well).
3. Finds the connected components and labels them with numbers (starting at 1).
   Hint: In order to label the connected components you can adapt the DFS (Depth First Search).
4. Prints the connected components. In particular prints the *names* of the vertices in that connected component. See run_people1.txt and run_mst1_cities.txt .
5. Runs MST_Prim (starting at sVertex) if the graph is weighted and prints:
   - the edges as picked by MST_Prim starting at vertex sVertex and
   - the total cost of the MST tree built.

   If the graph is not weighted, print a message to say that the MST_Prim will not be applied to this graph. See run_people1.txt

MST_Prim is Prim's algorithm for finding Minimum Spanning Trees (MST). In class we discussed 2 implementations for it:

- the one from the book that uses a priority queue. Do NOT implement this one.
- The one that we wrote on paper that does not use a priority queue. *Implement this one*. ***It will need to be fixed as we have an easy-to-find bug in it.***

The files with the graph data have the following format:

6. weighted - if this number is 1, it means the graph is weighted, any other value means unweighted.
   A weighted graph is a graph where edges have weights.
7. sVertex - Starting vertex for MST. Even though this is only needed for the weighted graphs, in order to make reading from file easier, you can assume you have it in files for graphs with no weights.– 11/21/19
8. N - number of vertices in the graph
9. Vertices list:
   - N lines each containing ONE string (no spaces) with a vertex name (e.g. FlowerCity).
   - You can assume a vertex name has length at most 100.
10. Edges list. The edges will have 2 formats based on if the graph is weighted or not.
    - If the graph is NOT weighted: A number of lines in format: "name1 name2", ending with the line "-1 -1" to indicate the end of lines with edges. See file people1.txt.
    - If the graph IS weighted: A number of lines in format: "name1 name2 ***weight***", ending with the line "-1 -1 -1" to indicate the end of lines with edges. See file mst1.txt.

The program must not have errors when run with Valgrind. See posted sample runs.

Data files: people1.txt, mst1_7.txt, mst1_3.txt, mst1_cities_7.txt . .– Updated 11/21/19

Sample run files: run_people1.txt, run_mst1_7.txt, run_mst1_3.txt, run_mst1_cities.txt . – Updated 11/21/19

You do NOT need to match the printed text at the beginning: `Is it a weighted graph: 1-yes (edges have weights), else-no)?: … Enter name1 name2 (weight if applicable): _____`

You can use any of the code posted on the Code page for graphs. Refresh the pages.
Save the program in a file called **graph.c**.

P2. (5 pts) **Hashing**

The images below show the occupancy of two hash tables. Both tables have **the same size**, the **same items** hashed in them, and both use **open addressing**. However they differ in the way they find an available slot in the table. Which one is a better hash table and why? (You do NOT have to deduce how they find the next available slot. You simply have to judge which one would behave better based on this image.)

The bottom table is the better hashtable because it more evenly spreads the data. The top shows signs of open address fixing, which means its hash function isn't as effective as the bottom one.

P3. (12 pts) **Hashing**

a) (5 points) Give an example of a bad hash function for strings (that generates many collisions). Justify why it is bad: find some strings that will hash to the same cell.

Using only the first letter of a string and then modulo the size of the table. $\lfloor (str[0] + some\ num) \% N \rfloor$

This will cause many collisions. It is bad because a larger table size will mean that really there are only 26 variations to collide on.

b) (7 points) In the hash table below * and + indicate the cells probed when trying to insert two different items. These items are originally hashed to the same slot (see +* at index 5). The star, *, shows the probed cells for item and the plus, +, shows the probed cells for the other item. You can assume that the table size is very big (e.g. more than 1000) and the slots shown did not require a mod (%) operation (that is we did not have to wrap around).

1) What type of open addressing was used? Justify.

~~This open addressing appears to be linear addressing.~~

This addressing appears to be double hashing

2) Give the next slots to be checked for each item (show where the next * and where the next + will be).

Index	
…	…
5	+ *
…	…
8	+
9	*
…	…
11	+
12	
13	*
14	+
…	…
17	*
…	…

the next two slots will be a "+" then "*" most likely at ~~18, 19~~ respectively depending on the algorithm. (17, and 21)

Remember to include your name at the top.

Place the code, **graph.c**, and the written answers, **2320_H8.pdf,** in a folder called **2320_HW8**, zip that folder and upload it in Canvas.