CS271: Data Structures

Instructor: Dr. Stacey Truex
Due: Friday April 7, 2023 5:00pm

Project #5

This project is meant to be completed in pairs. You should work in your Unit 3 groups. Solutions should be written in `C++` with one submission per group. Submissions should be a compressed file following the naming convention: `NAMES_cs271_project5.zip` where `NAMES` is replaced by the first initial and last name of each group member. For example, if Dr. Truex and our TA were in a group they would submit one file titled `STruexCKim_cs271_project5.zip`. **You will lose points if you do not follow the course naming convention**. Your `.zip` file should contain a *minimum* of 3 files:

1. `makefile`
2. `graph.cpp`
3. `test_graph.cpp`

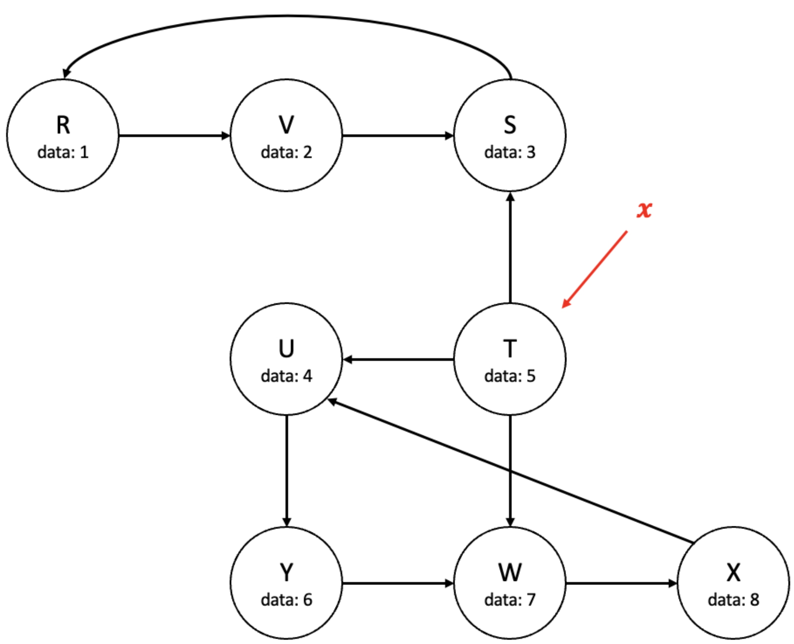
Additional files such as a `graph.h` header file or `README.md` are welcome. The above merely represent the minimum files required for project completion. Details for each are as follows.

Specifications

Graphs

Implement a `Graph` class using **two** templates - one for the data associated with each `Graph` vertex and one for the key associated with each vertex. In your implementation, denote the data template first. Your class should, at a minimum, support the following operations:

- get(k): `G.get(k)` should return a *pointer* to the vertex corresponding to the key `k` in the graph `G`. For example, consider the following graph:



Given the above,

```
Graph<int,string> G;
// populate graph G with vertices and edges corresponding to graph above
cout << G.get("T") -> data << endl;
cout << G.get("T") -> key << endl;
```

should print the integer value 5 and the string "T" respectively as the command `G.get("T")` returns the equivalent of x from the image above.

- reachable(u, v): `G.reachable(u, v)` should indicate if the vertex corresponding to the key `v` is reachable from the vertex corresponding to the key `u` in the graph `G`. For example, using the graph `G` from above:

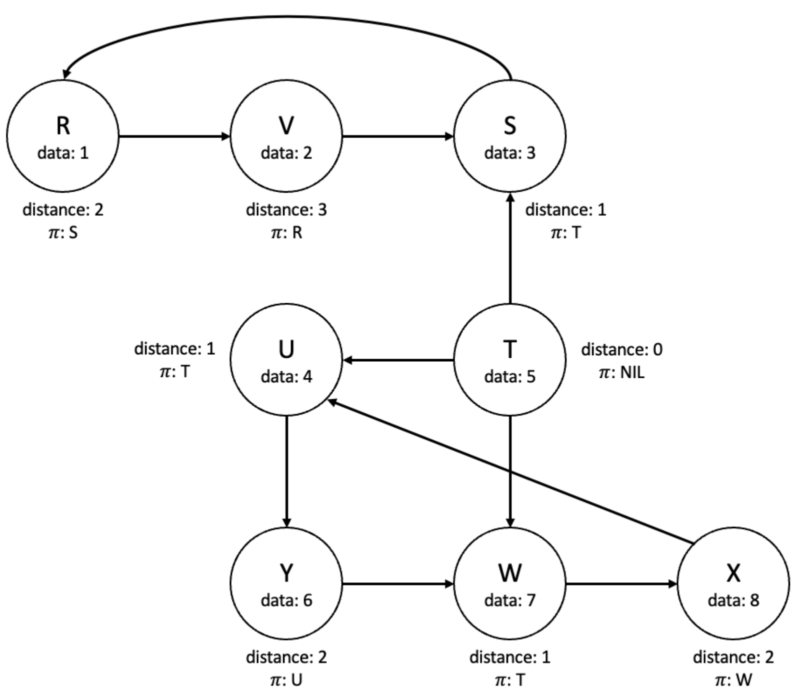
```
if(G.reachable("T", "X")) {
    cout << "X is reachable from T" << endl;
}
if(G.reachable("X", "T")) {
    cout << "T is reachable from X" << endl;
}
```

should *only* print the string `"X is reachable from T"`

- bfs(s): `G.bfs(s)` should execute the breadth-first search algorithm for the graph `G` from the source vertex corresponding to the key value `s`. For example, using the graph above:

```
G.bfs("T");
```

should result in the following internal representation of the graph:



- print_path(u, v): `G.print_path(u, v)` should print a shortest path from the vertex in `G` corresponding to the key `u` to the vertex in `G` corresponding to the key `v`. For example:

```
G.print_path("U", "X");
```

  should result in the printing of the string `"U -> Y -> W -> X"`. Paths should be printed as the keys of the vertices along the path separated by a space, `->`, and another space. Adhering to this formatting is required to pass testing.

- edge_class(u, v): `G.edge_class(u, v)` should return the string representation of the edge classification (tree edge, back edge, forward edge, cross edge, or no edge) of the edge from vertex `u` to `v`. For example, using the graph `G` from above:

```
cout << G.edge_class("X", "U") << endl;
cout << G.edge_class("V", "S") << endl;
```

  should print `"back edge"` and `"tree edge"` respectively. Note that returning the exact string `"tree edge"`, `"back edge"`, `"forward edge"`, `"cross edge"`, or `"no edge"` is required to pass testing.

- bfs_tree(s): `G.bfs_tree(s)` should **print** the bfs tree for the source vertex corresponding to the key `s`. Vertices in the bfs tree should be represented by their keys. Each depth level of the tree should be printed on a separate line with each vertex at the same depth being separated by a single space. For example, using the graph above:

```
G.bfs_tree("T");
```

  should result in the printing of the following string:

```
T
S U W
R Y X
V
```

Unit Testing

In addition to the functionality above, you are expected to implement the following constructor:

```
Graph(vector<K> keys, vector<D> data, vector<vector<K>> edges)
```

where `keys` is a vector of vertex keys, `data` is a vector of the corresponding vertex data in matching order, and `edges` is a vector of vectors representing the adjacency lists of the vertices in matching order. For example, you should be able to construct the graph `G` from the specifications above via the following:

```
vector<string> keys = {"R", "V", "S", "T", "U", "Y", "W", "X"};
vector<int> data = {1, 2, 3, 5, 4, 6, 7, 8};
vector<vector<string>> edges = {{"V"},{"S"},{"R"},{"S","U","W"},{"Y"},{"W"},{"X"},{"U"}};
Graph<int,string> G(keys, data, edges);
```

Your implementation of this constructor will be required for your class to pass testing. Both $G.V$ and $G.Adj[u]$ for all $u \in G.V$ should be in ascending order by key. You may assume the `keys` vector passed to your constructor is also in such order.

For each `Graph` method included in your `Graph` class, write a unit test method in a separate unit test file that *thoroughly* tests that method. Think, in addition to common cases: what are my boundary cases? edge cases? disallowed input? Each method should have *its own* test method.

An example test file test_graph_example.cpp has been provided along with an example graph specified in graph_description.txt. The example test file demonstrates (1) a general outline of what is expected in a test file and (2) a guide on how your projects will be tested after submission. The tests included in test_graph_example.cpp are not exhaustive. The unit testing in your test_graph.cpp file should be much more complete. Additionally, for grading purposes, your code will be put through significantly more thorough testing than what is represented by test_graph_example.cpp. Passing the tests in this example file should be viewed as a lower bound.

Documentation

The expectation of all coding assignments is that they are well-documented. This means that logic is documented with line comments and method pre- and post- conditions are properly documented immediately after the method's parameter list.

Pre-conditions and post-conditions are used to specify precisely what a method does. However, a pre-condition/post-condition specification does not indicate how that method accomplishes its task (if such commenting is necessary it should be done through line level comments). Instead, pre-conditions indicate what must be true before the method is called while the post-condition indicates what will be true when the method is finished.

Makefile

With each project you should be submitting a corresponding makefile. Once unpacking your .zip file, the single command make should create a test executable. The command ./test should then run all the unit tests in your test_graph.cpp file evaluating your Graph class.

Efficiency

Each project in this course will additionally be evaluated for efficiency. Each method detailed in the Graph class methods section of this document will be called 1 time using a very large example. The total time to execute all methods will be clocked.

Peer Evaluation

Each partner should additionally complete a pair evaluation form reflecting on the performance of both themselves and their partner with respect to professionalism and effort on the project. Reflections should be submitted individually, separate from the project file. Note that reflections will always remain confidential.

Rubric

Code	**Completeness** met submission requirements	21
	Correctness passes unit testing	48
	Correctness implementation deductions ex: `to_string` does not compile with `string` template	0
Efficiency	time test	15
Documentation	detailed comments pre- and post-conditions	6
Testing	thoroughness of unit tests	10