Due Date: April 17, 2020 @ 11:59:59

Generic Programming – Union Find

This assignment will focus on the introduction of C++ Templates and Generics. We're going to write a program to implement the Union-Find (or Disjoint-set) data structure and use it to check whether an undirected graph contains a cycle or not. We will be writing 3 generic classes to implement this functionality. We will be using union by rank and path compression in find to improve time complexity for our use case. A video tutorial and slides are uploaded on Canvas which give an overview of Union-Find. Be sure to check that out.

You will submit your work to Gradescope and it will be both auto graded and human graded. The autograder will compile your code with g++-7.4.0. Be sure to watch Piazza for any clarifications or updates to the assignment.

Don't forget the Assignment 6 Canvas Quiz!
Your assignment submission is not complete without the quiz.

Your submission must consist of the following files: **SetUF.h**, **UnionFind.h, Graph.h, demo.cpp, README.md** and a **Makefile**. There are 3 generic classes to be implemented (**SetUF**, **UnionFind** and **Graph**). **demo.cpp** implements the **main()** function which will test the overall functionality.

For header files, in general *(*.h)*:

- Be sure to have include guards.
- Never put "using namespace ..." in a header file
- Don't import anything that is unnecessary.
- Include file header comments that have your name, student id, a brief description of the functionality, and be sure to cite any resource (site or person) that you used in your implementation.
- Each function should have appropriate function header comments.
  - You can use javadoc style (see doxygen) or another style but be consistent.
  - Be complete: good description of the function, parameters, and return values.
  - In your class declarations (and header files in general), only inline appropriate functions.
- All the templated code should go in a header file.

For source files, in general *(*.cpp)*:

- Don't clutter your code with useless inline comments (it is a code smell [Inline Comments in the Code is a Smell, but Document the Why], unless it is the why).
- Follow normal programming practices for spacing, naming, etc.: Be reasonable and consistent.

- Be sure to avoid redundant code

For classes, in general:

- Remember the rule of 3: Explicitly define the destructor, copy operator=, and copy constructor if needed.
- Reuse the code better by calling base class functions and constructors from derived classes.
- Do not make a member variable public or protected, unless it makes sense to do so.

For *SetUF.h*

- SetUF<T> represents the concept of a set in the union find data structure. A set is a collection of items. In the union find data structure, sets are likes nodes in a tree structure.
- SetUF<T> requires 3 private member variables
    1. ***T value***
        - A member variable of generic type T
        - The value of one item of a set.
    2. ***unsigned rank***
        - A member variable to store the rank of a set.
        - Rank roughly gives a measure of the depth of the tree for a set.
        - By using rank, we maintain balanced sets if we start with balanced sets.
        - The concept of rank will become clearer once you go through the algorithm for union of 2 sets.
    3. ***SetUF *parent***
        - A member variable to store the parent node of this node of the set.
        - For a set, the node whose parent is itself can be termed as the **root node** of the set. The root node is representative of a given set and the rank of the root node gives us an idea about the depth of this set which is used to maintain balanced sets.

- SetUF<T> should have the following member functions
    1. ***SetUF(T v)***
        - A Parametrized constructor to set the value of the node.
        - The default value of the rank should be zero and of the parent should be the node itself (this).
        - The constructor should make the casting from T to SetUF<T> implicit.
    2. ***SetUF(const SetUF &a)***
       SetUF& operator=(const SetUF &a)
        - A copy constructor and assignment operator to create a new singleton.

- Both methods copy a value, set rank to *0* and parent to *this*.

3. ***operator T () const { return value; }***
   - A conversion function to implicitly cast SetUF object to type T.
   - 2 SetUF objects (or singletons) are the same if their node values are equal.
   - Hence, if we have this operator in place which implicitly casts SetUF<T> to T and as long as the comparison operators (== and *!=*) are overloaded in T, we have a way to compare the values of the nodes and thereby to compare SetUF objects.

- Remember UnionFind<T> class should be a **friend** of SetUF<T> as union and find operations need access to private members of SetUF<T>, e.g. rank and parent.

For *UnionFind.h*

- UnionFind <T> requires 1 private member variable
  1. ***std::vector<SetUF<T>> sets***
     - A container to store all the set singletons.

- UnionFind <T> should have the following member functions
  1. ***UnionFind(const std::vector<T> &singletons)***
     - A parametrized constructor to add the singletons to the container sets
  2. ***SetUF<T>& find(T node)***
     - find gives the root of the set to which the node belongs to.
     - Remember, we are also doing path compression to improve the time complexity.
     - The algorithm is as follows

       Find(x)
       1. *Starting from x, traverse to the root of the set*
       2. *For each node traversed, set its parent to the root (path compression)*
       3. *Return root*
  3. ***void unionOp(SetUF<T> &x, SetUF<T> &y)***
     void unionOp(T x, SetUF<T> &y)
     void unionOp(SetUF<T> &x, T y)
     void unionOp(T x, T y)
     - Note, we cannot use **union** as the function name as it is a reserved keyword in C++.
     - Remember, we are doing union by rank. The rank of a set is the rank of its root. The algorithm basically finds root of both the sets (using **find if argument is a node of the set and not the set itself, hence 4 methods**) and by looking at the

rank of each set, one is made the parent of the other and rank is adjusted accordingly.

- The algorithm is as follows

  UnionOp(x, y)

  1. *Make the higher rank set the parent of the lower rank set.*
  2. *If x and y have same rank, increment the rank of the new set by 1.*

For *Graph.h*

- Graph.h is fully implemented for you.
- The class definition is provided to you, but it has not been commented properly.
- Your task is to add the file, class, function and member comments.
- Graph.h uses the union find data structure we implemented above to detect cycle in an undirected graph.
- For the ***containsCycle*** function, give the algorithm detail in function header comments.

demo.cpp

- demo.cpp is already provided to you. Go through demo.cpp implementation to check how we are testing our overall functionality.
- We have created a graph where the vertices are objects of Student class and we are using our union find data structure to detect presence of a cycle in this graph.
- **What you should do is** create 2 additional graphs where vertices can be of any data type (built-in or user-defined) other than Student. One of the graphs should have a cycle with at least 5 nodes while the other graph should be a full binary tree of height 3. (which means a total of 15 nodes – 1 + 2 + 4 + 8)
- While this file **will be checked during manual grading**, it won't be used as a part of auto grading **and hence no fixed output is required**.

Makefile

Your submission must include a *Makefile*. The minimal requirements are:

- Compile using the demo executable with the *make* command.
- Use appropriate variables for the compiler and compiler options.
- It must contain a clean rule.
- The demo executable must be named something meaningful (not *a.out*).

README.md

Your submission must include a README.md file (see https://guides.github.com/features/mastering-markdown/). The file should give a brief description of the program, the organization of the code, how to compile the code, and how to run the program.

Submission

You will upload all the required files to Gradescope. Reminder to complete the Assignment 6 Canvas quiz. There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, half of the marks will come from the auto grader. For this assignment, none of the test details have been hidden, but there may be hidden tests in future assignments.

The other half of the marks will come from human grading of the submission. Your files will be checked for style and to ensure that they meet the requirements described above. In addition, all submitted files should contain a header (comments or otherwise) that, at a minimum, give a brief description of the file, your name and wisc id.

HAPPY CODING!