```
#ifndef UTILS_H
#define UTILS_H
#include <unordered_map>
#include <chrono>
#include <vector>
#include <string>
#include <list>
#include <iostream>
#include "data_src/Game.h"
#include "GraphAdjMatrix.h"
#include "GraphAdjList.h"
#include "DataSource.h"
#include "Bridges.h"
#include "Style.h"
using std::unordered_map;
using std::vector;
using std::string;
using std::list;
using std::cout;
using std::cin;
using namespace chrono;
using chrono::system_clock;
using chrono::duration;
using bridges::Bridges;
using bridges::DataSource;
using bridges::GraphAdjMatrix;
using bridges::GraphAdjList;
using bridges::Game;

/// Immutable unordered mapping of a numerical key to each graph type name
const static unordered_map<int, string> graph_keymap = { {1, "matrix"}, {2, "list"} };

/// Immutable unordered mapping of a platform string key to a numerical value
const static unordered_map<string, int> platform_keymap =
{{"Arcade", 0}
,{"Atari 2600", 1}
,{"Atari 5200", 2}
,{"NES", 3}
,{"Super NES", 4}
,{"Nintendo 64", 5}
,{"Nintendo 64DD", 6}
,{"GameCube", 7}
,{"Wii", 8}
,{"Wii U", 9}
,{"Nintendo DS", 10}
,{"Nintendo DSi", 11}
,{"Nintendo 3DS", 12}
,{"Game Boy", 13}
,{"Game Boy Color", 14}
,{"Game Boy Advance", 15}
,{"PlayStation", 16}
,{"PlayStation 2", 17}
,{"PlayStation 3", 18}
,{"PlayStation 4", 19}
,{"PlayStation Vita", 20}
,{"PlayStation Portable", 21}
,{"Xbox", 22}
,{"Xbox 360", 23}
,{"Xbox One", 24}
,{"Master System", 25}
```

```
,{"Genesis", 26}
,{"Sega 32X", 27}
,{"Sega CD", 28}
,{"Saturn", 29}
,{"Dreamcast", 30}
,{"Dreamcast VMU", 31}
,{"Commodore 64/128", 32}
,{"Lynx", 33}
,{"NeoGeo", 34}
,{"NeoGeo Pocket Color", 35}
,{"TurboGrafx-16", 36}
,{"TurboGrafx-CD", 37}
,{"WonderSwan", 38}
,{"WonderSwan Color", 39}
,{"N-Gage", 40}
,{"Vectrex", 41}
,{"PC", 42}
,{"Pocket PC", 43}
,{"Linux", 44}
,{"Macintosh", 45}
,{"iPhone", 46}
,{"iPad", 47}
,{"iPod", 48}
,{"Android", 49}
,{"Windows Phone", 50}
,{"Windows Surface", 51}
,{"Wireless", 52}
,{"Game.Com", 53}
,{"Web Games", 54}
,{"DVD / HD Video Game", 55}
,{"String", 56}};

/// Immutable vector array of Game objects sourced from the Bridges Game dataset
/// @dataset: https://bridgesdata.herokuapp.com/api/games
const static vector<Game> game_data()
{
  Bridges bridge_src(0, "jbasil", "59727335315");
  DataSource ds (&bridge_src);
  return ds.getGameData();
}

/**
 * @brief: Checks a game title against a list of Game objects and updates a Game object
reference if found
 * @param: games   a list of Game objects
 * @param: title   the game title to validate
 * @param: g       Game object reference corresponding to parameter title
 * @return: true if game title found; otherwise false
 */
const bool validate_game(const vector<Game>& games, const string& title, Game& g)
{
  auto it = games.begin();
  while (it != games.end())
  { if (it->getTitle() == title) break; it++; }

  if (it == games.end())
  { cout << "Invalid input - Game title not found.\n\n"; return false; }

  g = *it;
  return true;
```

```
}

/**
 * @brief: Reads in a game title string from the terminal and updates the corresponding title
reference
 * @param: title  a game title reference
 */
void title_prompt(string& title)
{
  cout << "Enter a game title:\n";
  if (!title.empty()) title.clear();
  getline(cin, title);
  cout << endl;
}

/**
 * @brief: Reads in a game minimum rating from the terminal and updates the corresponding
rating reference
 * @param: min_rating  a game lower bound rating reference
 * @throws: invalid_argument ex if input is not floating point
 * @return: true if input is valid; otherwise false
 */
const bool rating_prompt(float& min_rating)
{
  cout << "Enter a rating lower bound (0.0 - 10.0):\n";
  string in_str;
  getline(cin, in_str);

  try
  {  min_rating = stof(in_str); }
  catch (invalid_argument const& ex)
  { cout << "Invalid input - Not a floating point number.\n\n"; return false; }

  if (min_rating > 10.0 || min_rating < 0.0)
  { cout << "Invalid input - Rating out of bounds.\n\n"; return false; }
  cout << endl;
  return true;
}

/**
 * @brief: Reads in a graph preference from the terminal and updates the corresponding pref
reference
 * @param: graph_pref  a graph type key
 * @return: true if input is valid; otherwise false
 */
const bool type_prompt(int& graph_pref)
{
  cout << "Enter a command key:\n";
  cout << "1 to graph an adjacency matrix\n";
  cout << "2 to graph an adjacency list\n";
  cout << "3 to restart program\n";
  cout << "0 to quit program\n";

  string in_str;
  getline(cin, in_str);
  if (in_str.length() != 1) { cout << "Invalid input - Too many digits.\n\n"; return false; }

  char c = in_str.at(0);
  if (c > 51 || c < 48) { cout << "Invalid input - Not a valid digit.\n\n"; return false; }
  cout << endl;
```

```
  graph_pref = stoi(in_str, nullptr, 10);
  return true;
}

/**
 * @brief: Reads in a density preference from the terminal and updates the corresponding pref
reference
 * @param: is_dense    a density preference
 * @param: graph_name  the name of the preferred graph
 * @return: false if input is invalid or go back is selected; otherwise true
 */
const bool density_prompt(bool& is_dense, const string& graph_name)
{
  cout << "Enter a command key:\n";
  cout << "1 to graph a sparse adjacency " << graph_name << endl;
  cout << "2 to graph a dense adjacency " << graph_name << endl;
  cout << "0 to go back\n";

  string in_str;
  getline(cin, in_str);
  if (in_str.length() != 1) { cout << "Invalid input - Too many digits.\n\n"; return false; }

  char c = in_str.at(0);
  if (c > 50 || c < 48) { cout << "Invalid input - Not a valid digit.\n\n"; return false; }
  if (c == 48) return false;
  cout << endl;

  is_dense = stoi(in_str, nullptr, 10) - 1; // decrement input for boolean alignment
  return true;
}

/**
 * @brief: Writes out the graph statistics to the terminal
 * @param: graph_name  the name of the preferred graph
 * @param: node_count  the number of nodes in the graph adjacency structure
 * @param: timer_start a point in time before the graph is created
 * @param: timer_end   a point in time after the graph is created
 */
static void print_stats(
  const string& graph_name,
  const bool& is_dense,
  const int& node_count,
  const time_point<system_clock>& timer_start,
  const time_point<system_clock>& timer_end)
{
  chrono::duration<double> elapsed = timer_end - timer_start;
  cout << "creation time: " << elapsed.count() << " seconds\n";
  cout << "struct size: " << node_count << " nodes\n";
  cout << "graph type: " << (is_dense ? "dense " : "sparse ") << graph_name << "\n\n";
}

/**
 * @brief: Compares two genre lists and increments weight for each match
 * @param: g1_genres  a first list of genres
 * @param: g2_genres  a second list of genres
 * @return: pointer to weight variable
 * @complexity: O(|M|log(|N|s)) where |M|=g1_genres.size(), |N|=g2_genres.size(),
s=genre.length() (arbitrary)
 */
```

```
int* weigh_genres(const vector<string>& g1_genres, const vector<string>& g2_genres)
{
  int wt = 0;
  int* wt_ptr = &wt;

  list<string> ll;
  for (string g2_genre : g2_genres) ll.push_front(g2_genre);

  for (auto arr_it = g1_genres.begin(); arr_it != g1_genres.end(); arr_it++) // O(|M|)
  {
    string genre = *arr_it;
    for (auto ll_it = ll.begin(); ll_it != ll.end(); ll_it++) // O(log|N|)
      if (*ll_it == genre) // O(s)
      { ll.erase(ll_it); wt++; break; } // O(1) erasure
  }
  return wt_ptr;
}

/**
 * Creates and visualizes an undirected adjacency matrix
 * @param: games       a list of Game objects
 * @param: src         Game object reference corresponding to parameter title
 * @param: src_title   the game title to validate
 * @param: min_rating  a game lower bound rating reference
 * @param: is_dense    a density preference
 * @param: graph_name  the name of the preferred graph
 * @complexity: O(|A|log|A|+|G||M|log(|N|s)), where |A|=src.adj.size(), G=games.size(); see
also: weigh_genres()
 */
void create_matrix(
  const vector<Game>& games,
  const Game& src,
  const string& src_title,
  const int& min_rating,
  const bool& is_dense,
  const string& graph_name)
{
  Style s; // initialize style object

  int node_count = 0;
  list<Game> src_ll;
  stack<Game> src_stk;

  const string src_platform = src.getPlatformType();
  const vector<string> src_genres = src.getGameGenre();

  auto start = chrono::system_clock::now(); // start graph creation timer
  GraphAdjMatrix<string, string> graph;
  graph.addVertex(src_title, "");
  for (auto g_it = games.begin(); g_it != games.end(); g_it++) // iterate through games O(|G|)
  {
    const Game g = *g_it;
    const string g_title = g.getTitle();
    const string g_platform = g.getPlatformType();
    const float g_rating = g.getRating();
    if (g_rating < min_rating || g_title == src_title) continue;

    int* weight = weigh_genres(g.getGameGenre(), src_genres); // O(|M|log(|N|s))
    if (*weight)
    {
```

```
      bool platforms_match = (platform_keymap.at(g_platform) ==
platform_keymap.at(src_platform)); // O(1)
      if (platforms_match) (*weight)++;

      graph.addVertex(g_title, ""); // O(1)*
      graph.addEdge(src_title, g_title, *weight); // O(1)*
      graph.addEdge(g_title, src_title, *weight); // O(1)*
      s.style_graph_matrix(graph, src_title, g_title, *weight, platforms_match, min_rating,
g_rating); // O(1)*

      src_ll.push_front(g); // push source adjacents to linked list O(1)
      src_stk.push(g); // push source adjacents to stack O(1)
    } // * if no hash collisions; graph class backed by unordered_map
  }
  const int adj_count = src_stk.size();
  node_count = pow(adj_count + 1, 2); // n = |V|^2 = (|A|+1)^2 where A = set of adjacents to
src

  if (is_dense)
  {
    while (!src_stk.empty()) // O(|A|)
    {
      Game g1 = src_stk.top();
      src_stk.pop();

      const string g1_title = g1.getTitle();
      const vector<string> g1_genres = g1.getGameGenre();
      const string g1_platform = g1.getPlatformType();
      for (auto ll_it = src_ll.begin(); ll_it != src_ll.end(); ll_it++) // O(log|A|)
      {
        Game g2 = *ll_it;
        const string g2_title = g2.getTitle();
        if (g1_title == g2_title) { src_ll.erase(ll_it--); continue; } // O(1)

        int* weight = weigh_genres(g1_genres, g2.getGameGenre());
        if (*weight)
        {
          const string g2_platform = g2.getPlatformType();
          if (platform_keymap.at(g1_platform) == platform_keymap.at(g2_platform)) (*weight)++;

          graph.addEdge(g1_title, g2_title, *weight);
          graph.addEdge(g2_title, g1_title, *weight);
        }
      }
    }
  }
  auto end = chrono::system_clock::now(); // end graph creation timer


  Bridges bridge_out(1 + is_dense, "jbasil", "59727335315");
  bridge_out.setDataStructure(&graph);
  bridge_out.visualize();

  print_stats(graph_name, is_dense, node_count, start, end);
}

/**
 * Creates and visualizes an undirected adjacency list
 * @param: games       a list of Game objects
 * @param: src         Game object reference corresponding to parameter title
```

```
 * @param: src_title   the game title to validate
 * @param: min_rating  a game lower bound rating reference
 * @param: is_dense    a density preference
 * @param: graph_name  the name of the preferred graph
 * @complexity: O(|A|log|A|+|G||M|log(|N|s)), where |A|=src.adj.size(), G=games.size(); see
also: weigh_genres()
 */
void create_list(
  const vector<Game>& games,
  const Game& src,
  const string& src_title,
  const int& min_rating,
  const bool& is_dense,
  const string& graph_name)
{
  Style s; // initialize style object

  int node_count = 0;
  list<Game> src_ll;
  stack<Game> src_stk;

  const string src_platform = src.getPlatformType();
  const vector<string> src_genres = src.getGameGenre();

  auto start = chrono::system_clock::now(); // start graph creation timer
  GraphAdjList<string, string, unsigned> graph;
  graph.addVertex(src_title, "");
  node_count++;
  for (auto g_it = games.begin(); g_it != games.end(); g_it++) // iterate through games O(|G|)
  {
    const Game g = *g_it;
    const string g_title = g.getTitle();
    const string g_platform = g.getPlatformType();
    const float g_rating = g.getRating();
    if (g_rating < min_rating || g_title == src_title) continue;

    int* weight = weigh_genres(g.getGameGenre(), src_genres); // O(|M|log(|N|s))
    if (*weight)
    {
      bool platforms_match = (platform_keymap.at(g_platform) ==
platform_keymap.at(src_platform)); // O(1)
      if (platforms_match) (*weight)++;

      graph.addVertex(g_title, ""); // O(1)*
      graph.addEdge(src_title, g_title, *weight); // O(1)*
      graph.addEdge(g_title, src_title, *weight); // O(1)*
      node_count+=3; // n = |V|+|E| = (1+|A|)+2|A| where A = set of adjacents to src

      src_ll.push_front(g); // push source adjacents to linked list O(1)
      src_stk.push(g); // push source adjacents to stack O(1)
      s.style_graph_list(graph, src_title, g_title, *weight, platforms_match, min_rating,
g_rating); // O(1)*
    } // * if no hash collisions; graph class backed by unordered_map
  }

  if (is_dense)
  {
    while (!src_stk.empty()) // O(|A|)
    {
      Game g1 = src_stk.top();
```

```
      src_stk.pop();

      const string g1_title = g1.getTitle();
      const vector<string> g1_genres = g1.getGameGenre();
      const string g1_platform = g1.getPlatformType();
      for (auto ll_it = src_ll.begin(); ll_it != src_ll.end(); ll_it++) // O(log|A|)
      {
        Game g2 = *ll_it;
        const string g2_title = g2.getTitle();
        if (g1_title == g2_title) { src_ll.erase(ll_it--); continue; } // O(1)

        int* weight = weigh_genres(g1_genres, g2.getGameGenre());
        if (*weight)
        {
          const string g2_platform = g2.getPlatformType();
          if (platform_keymap.at(g1_platform) == platform_keymap.at(g2_platform)) (*weight)++;

          graph.addEdge(g1_title, g2_title, *weight);
          graph.addEdge(g2_title, g1_title, *weight);
          node_count+=2; // n = |V|+|E| = (1+|A|)+(2|A|+2|EA*|)
        } // * EA = set of edges of (A = set of adjacents to src)
      }
    }
  }
  auto end = chrono::system_clock::now(); // end graph creation timer

  Bridges bridge_out(3 + is_dense, "jbasil", "59727335315");
  bridge_out.setDataStructure(&graph);
  bridge_out.visualize();

  print_stats(graph_name, is_dense, node_count, start, end);
}

#endif // UTILS_H
```