CSE 208

Graph

(For A1, A2, B1, B2)

Implement **Graph** data structure in two ways:

- adjacency matrix implementation
- adjacency list implementation

In order to maintain adjacency list and implement BFS, you should also implement two other classes: **ArrayList** and **Queue** (here Queue means FIFO queue).

You cannot use any built-in data structure for ArrayList and Queue.

All the three classes (Graph, ArrayList, Queue) should have their own constructor and destructor. In constructor, you should properly allocate memory dynamically (if needed). In destructor, you should properly deallocate all the dynamically allocated memory and perform other cleanup tasks.

The graph can be either directed or undirected. This choice will be given as input by the user.

Queue class should have the following functions (in addition to constructor and destructor):

- void enqueue(int item); //insert item in the queue
- int dequeue(); //returns the item according to FIFO
- bool empty(); //return true if Queue is empty, false otherwise

Queue should have a small initial size (say, 2) and double the size every time the queue overflows because of *enqueue* operation. Queue implementation should reuse the memory freed after deallocation, with minimal overhead.

ArrayList class maintains an ArrayList of integers. It should have the following functions (in addition to constructor and destructor):

- int searchItem(int item) ; // returns the index of item
- void insertItem(int item) ; // inserts item to the arraylist
- void removeItem(int item) ; // remove item from the arraylist

- void removeItemAt(int position); // remove item from the index *position*
- int getItem(int position) ; // returns the item at index *position*
- int getLength(); // returns the current length of the arraylist
- bool empty(); // returns true if the arraylist is empty, false otherwise
- void printList(); // print all the items of the arraylist

ArrayList should have a small initial size (say, 2) and double the size every time the ArrayList overflows because of *insertItem* operation.

Constructor of **Graph** class should take a parameter which denotes whether the graph will be directed or undirected. Graph class should have the following information inside it:

- number of vertices

- number of edges

- a pointer/reference to ArrayList, which will be dynamically allocated to create an array of ArrayList (only for adjacency list representation)

- a pointer/reference which will be used to create a 2D array or adjacency matrix (only for adjacency matrix representation)

Graph class should have the following functions (in addition to constructor and destructor):

- *void setnVertices(int n)*: Sets the number of vertices in the graph. This function initializes the graph data structure by allocating memory for the graph storage.
- *bool addEdge(int u, int v)*: Adds an edge (*u,v*) to the graph if there is currently no edge (u,v). In case of undirected graph, edge (*u,v*) is the same as edge (*v,u*). We are not allowing multiple edges between the same pair of vertices. The function will return true if an edge is added and false otherwise.
- *void printGraph()*: Prints the graph in adjacency list format.
- *void removeEdge(int u, int v)*: Removes the edge (*u, v*) from the graph.
- *bool isEdge(int u, int v):* Returns true if (*u, v*) is an edge, otherwise returns false.
- *int getOutDegree(int u)*: Returns the **out degree** of a vertex *u.*
- *int getInDegree(int u)*: Returns the in degree of a vertex *u*.
- *bool hasCommonAdjacent(int u, int v)*: Returns true if vertices *u* and *v* have some common adjacent vertices. Otherwise, it should return false.

- *void bfs(int source)*: Runs BFS algorithm on the graph using *source* as the source vertex. BFS statistics like parent, distance, and color information must be saved in class variables. So, you will need to define these as class private variables. Use the given ***Queue*** class in BFS. Define BFS related variables, i.e., parent, distance, and color, as class variables and initialize them memory appropriately in the ***Graph*** constructor.
- *void dfs(int source):* Runs DFS algorithm on the graph using *source* as the source vertex. Create separate variables to maintain DFS statistics such as color, parent.
- *int getDist(int u, int v)*: Returns the shortest path distance from vertex *u* to vertex *v*. You will first need to run BFS on the graph using *u* as the source vertex. Then, you will use the *dist* array to find the distance.

All the functions should consider **both directed and undirected graph**, whenever needed. Also, always check whether the vertex number is valid, i.e. between 0 and *nVertices* (inclusive).

You should implement all the above functions of Graph for both adjacency list representation and adjacency matrix representation in two separate files:

- *GraphAdjMatrix.cpp* or *GraphAdjMatrix.java*
- *GraphAdjList.cpp* or *GraphAdjList.java*

Runtime analysis of BFS for adjacency matrix and adjacency list representations:

You will test the runtime of BFS for the following number of vertices and edges:

Number of vertices: 1000, 2000, 4000, 8000, 16000

Number of edges: $|V|$, $2|V|$, $4|V|$, …. upto $(|V|^2-|V|)/8$

Edges will be created at random, i.e. randomly select two vertices and add an edge.

For each case (such as 1000 vertices & $2|V|=2000$ edges), create a Graph and select 10 source vertices at random and run BFS on those source vertices. Calculate the total time for these 10 runs in microseconds (or nanoseconds, if it gives more accurate measure). Then calculate the average time for these 10 runs of BFS. Report the runtimes in a doc/docx file, named **Report.doc** or **Report.docx**.

You may need the following functions if you use C++:

- srand(time(NULL))

- rand()
- chrono::steady_clock::now()
- chrono::duration_cast<chrono::microseconds>

Report all the runtimes found for both adjacency list and adjacency matrix representation. Also, answer the following questions in quantitative way:

- What is the impact on runtime if we keep |V| unchanged and double |E| for adjacency list? Why is it so?
- What is the impact on runtime if we keep |E| unchanged and double |V| for adjacency list? Why is it so?
- What is the impact on runtime if we keep |V| unchanged and double |E| for adjacency matrix? Why is it so?
- What is the impact on runtime if we keep |E| unchanged and double |V| for adjacency matrix? Why is it so?
- For the same |E| and |V|, why are the runtimes for adjacency list and adjacency matrix representation different? Which one is higher and why?

Answer the above questions in the same file **Report.doc** or **Report.docx.**

Put these files in a folder. The folder should be named with your 7-digit student ID (like 1705001). Create a zipped version of the folder naming it with your 7-digit student ID (like 1705001.zip) and upload it in moodle within 28 February (Friday) 11:55 pm.

You can use the code from your CSE 204 course.

There is no restriction on programming language. However, you should follow object-oriented approach. Some functions mentioned here have been written in C++ style, but you can use other object oriented language (such as Java) too.

Marks:

Adjacency matrix representation: 3.5 marks

Adjacency list representation: 3.5 marks

Report: 3 marks