**MIDTERM-4 REVIEW SOLUTIONS**

Q1.

bool Graph::BFSWithDistance(std::string startVertex){

    int index = BFSFind(startVertex); //check if that vertex does/does not exists

    if (index == -1){

        return false;

    }

    vertex \*v = &vertices[index];

    v->visited = true;

    v->distance = 0;

    queue<vertex\*> q;

    q.push(v);

    vertex \*n;

    while(!q.empty()){

        n = q.front();

        q.pop();

        int nAdjSize = n->adj.size();

        for(int x = 0; x < nAdjSize; x++){

            if(!n->adj[x].v->visited){

                n->adj[x].v->distance = n->distance + n->adj[x].weight;

                n->adj[x].v->visited = true;

                if(n->distance > 0 && n->distance < n->adj[x].weight){

                    return true;

                }

                q.push(n->adj[x].v);

            }

        }

    }

    return false;

}

Q2.

void Graph::shortestPath(string source, string destination, string intermediate){

    int sIndex, dIndex;

    for(int i=0;i<vertices.size(); i++){                    // find the source and destination in vertices

        if (vertices[i].name == source){

            sIndex = i;

        }

        if (vertices[i].name == destination){

            dIndex = i;

        }

    }

    vertex \*v = &vertices[sIndex];                          // pointer to address of source

    v->visited = true;

    queue<vertex\*> q;                                       // push source pointer to the vertex

    q.push(v);

    vertex \*n;                                              // for adjacent vertices

    while(!q.empty()){

        n = q.front();

        q.pop();

        for(int x = 0; x < n->adj.size(); x++){             // loop through each adjacent

            if(!n->adj[x].v->visited){                      // if they haven't been visited

                n->adj[x].v->visited = true;                // then visit and mark as such

                n->adj[x].v->prev = n;                      // then set up the parent

                q.push(n->adj[x].v);

            }

        }

    }

    vertex \*v2 = &vertices[dIndex];                        // pointer to the address of the dest

    while(v2->prev->name != source){                       // while the prev isn't the source (making sure there is something between)

        if(v2->prev->name == intermediate){                // checking if the intermediate is in the path

            cout << "Yes" << endl;

        }

        v2 = v2->prev;

    }

    cout << "No" << endl;

}

Q3.

bool Graph::pathExists(string path[], int length)

{

 vertex\* tmp;

 size\_t len=vertices.size();

 for(std::size\_t i=0;i<len;i++)

 {

     if(vertices[i].name==path[0])

     {

         tmp=&vertices[i];

         break;

     }

 }

 for(int i=1;i<length;i++)

 {

     bool found=false;

     for(size\_t j=0;j<tmp->adj.size();j++)

     {

         if(tmp->adj[j].v->name==path[i])

         {

               tmp=tmp->adj[j].v;

               found=true;

               break;

         }

     }

     if(!found)

       return false;

 }

  return true;

}

Q4.

void HashTable::buildHashTable(string movies[], int length){

   for (int i = 0; i<length; i++){

       Movie \*mov = new Movie(movies[i]);

       int id = newHashSum(movies[i], tableSize);

       if (hashTable[id]){

           hashTable[id]->next = mov;

       }else{

           hashTable[id] = mov;

       }

   }

}

int HashTable::newHashSum(string inputString, int hashLen){

   int sum = 0;

   for (size\_t i = 0; i<inputString.size(); i++){

       if (i%2==0){

           sum+=(int)inputString[i];

       }

   }

   return sum%hashLen;

}

Q5.

void HashTable::createNewHashTable(){

for(int i = 0; i < tableSize; i++){ //check array if items in table

if(hashTable[i] != NULL){ // check if something at that index

Movie \*temp = hashTable[i];

while(temp != NULL){

int newHashIndex = hashSum2(temp->title, tableSize);

// If there is nothing in this location.

if (newHashTable[newHashIndex] == NULL)

{

newHashTable[newHashIndex] = new Movie(temp->title);

}

// If we need to add to a chain.

else

{

Movie \* temp2 = newHashTable[newHashIndex];

Movie \* newMovie = new Movie(temp->title);

newMovie->next = temp2;

newHashTable[newHashIndex] = newMovie;

//cout << newMovie->title << endl;

}

temp = temp->next;

}

}

}

}