Midterm 4 Review Solutions

```
void Graph::BFTraversal(string startingCity){
    vertex *start;
    for(int i = 0; i < vertices.size(); i++){
        vertices[i].visited = false;
        if(vertices[i].name == startingCity){
            start = &vertices[i];
        }
    }
    queue<vertex*> Q;
    Q.push(start);
    start->visited = true;
    cout << start->name << endl;
    while(!Q.empty()){
        vertex *node = Q.front();Q.pop();
        for(int i = 0; i < node->adj.size(); i++){
            if(!node->adj[i].v->visited){
                cout << node->adj[i].v->name << endl;
                Q.push(node->adj[i].v);
                node->adj[i].v->visited = true;
            }
        }
    }
}
```

```
void HashTable::findAndMoveMisplacedMovies (){
    for (int i = 0; i < tableSize; i++){
        if (hashTable[i] != NULL){
            if(hashTable[i]->next == NULL){
                if(hashSum(hashTable[i]->title, tableSize) != i){
                    insertMovieAtIndex(hashTable[i]->title, hashSum(hashTable[i]->title, tableSize));
                    hashTable[i] = hashTable[i]->next;
                }
                continue;
            }
            else {
                Movie *temppre = hashTable[i];
                while(temppre->next!=NULL){
                    Movie *temp = temppre->next;
                    int tempindex = hashSum(temp->title, tableSize);
                    if(tempindex != i){
                        insertMovieAtIndex(temp->title, tempindex);
                        temppre->next = temppre->next->next;
                        delete temp;
                    }
                    temppre = temppre->next;
                    if(temppre == NULL) break;
                }
            }
        }
    }
}

bool Graph::pathExists(string path[], int length) {
    vertex v;
    for (int i = 0; i < vertices.size(); i++) {
        if (vertices[i].name == path[0]) {
            v = vertices[i];
        }
    }
    for (int i = 1; i < length; i++) {
        bool vertexFound = false;
        for (int j = 0; j < v.adj.size(); j++) {
            if (v.adj[j].v->name == path[i]) {
                v = *v.adj[j].v;
                vertexFound = true;
                break;
            }
        }
        if (!vertexFound) {
            return false;
        }
    }
    return true;
}

void HashTable::createNewHashTable()

{
for(int i = 0; i < 10; i++)
{
Movie *tmp = hashTable[i];
while(tmp != NULL)
{
string title = tmp->title;
int newHashSum = hashSum2(title, 10);
```

```
if(newHashTable[newHashSum] == NULL){
Movie *m = new Movie(title);
newHashTable[newHashSum] = m;
}
else{
Movie *temp = newHashTable[newHashSum];
Movie *m = new Movie(title);
newHashTable[newHashSum] = m;
m->next = temp;
}
tmp = tmp->next;
}
}
}
```

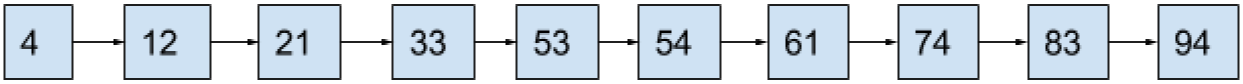
```
void Graph::shortestPath(string source, string destination, string intermediate)
{
    vertex *v = NULL;
    int i = 0;
    for(int k = 0; k < vertices.size(); k++)
    {
        vertices[k].visited = false;
    }

    for(i = 0; i < vertices.size(); i++)
    {
        if(vertices[i].name == source)
        {
            v = &vertices[i];
            break;
        }
    }

    v->visited = true;

    queue<vertex*> bfq;

    //cout<<v.name<<endl;
    vertices[i].visited = true;
    bfq.push(&vertices[i]);

    vertex *end = NULL;
    bool found = false;
```

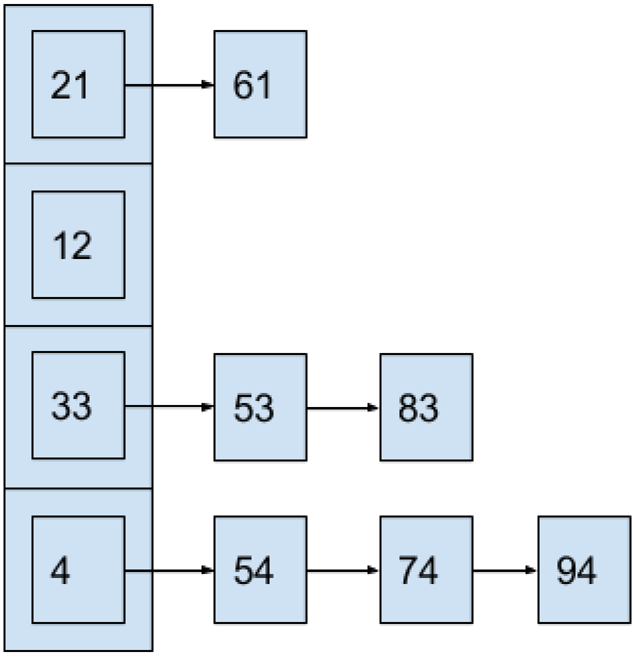
```
        while (!bfq.empty()) {
            v = bfq.front();
            bfq.pop();
            for(i=0;i<v->adj.size();i++) {
                if (v->adj[i].v->visited==false) {
                    v->adj[i].v->prev = v;
                    v->adj[i].v->visited = true;
                    if(v->adj[i].v->name == destination)
                    {
                        end = v->adj[i].v;
                        found = true;
                        break;

                    }
                    if(found)
                        break;
                    bfq.push(v->adj[i].v);
                }
            }

        }
        while(end->name != source)
        {
            if(end->name == intermediate)
            {
                cout << "Yes" << endl;
                return;
            }
            end = end->prev;
        }
        cout << "No" << endl;
    }

    void Graph::countBFTraversal(){
        int count = 0;
        for(int i=0; i<vertices.size();i++) {
            if (vertices[i].visited == false){
                 BFTraversal(i);
                 count++;
            }
        }
        cout << count << endl;
```

}

Conceptual Questions:

1) **Consider following linked list and hash table,**

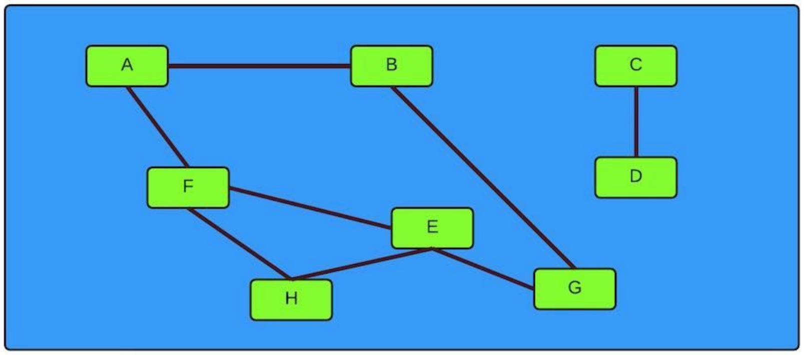




1. Number of operations required to search for element 94 in linked list?
   -> 10

2. Number of operations required to search for element 94 in hash table (Assume that hash function has been already written for you which will return the required index)?
   -> 5

2. Using the BFTraversal algorithm and the following graph, list the order that the vertices are visited starting from G and the distance to the vertex. Adjacent nodes are visited alphabetically and should be answered accordingly.



(G,0)

(B,1)

(E,1)

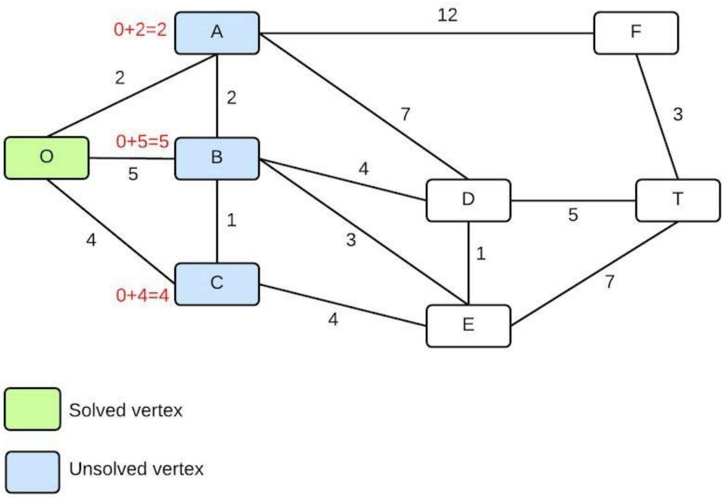
(A,2)

(F,2)

(H,2)

Answer: G 0 B 1 E 1 A 2 F 2 H 2

3. Using Dijkstra's algorithm and the following graph, find the shortest path from O to T. The first step has already been taken, showing that O has been marked as solved.



Shortest path from O to T ?

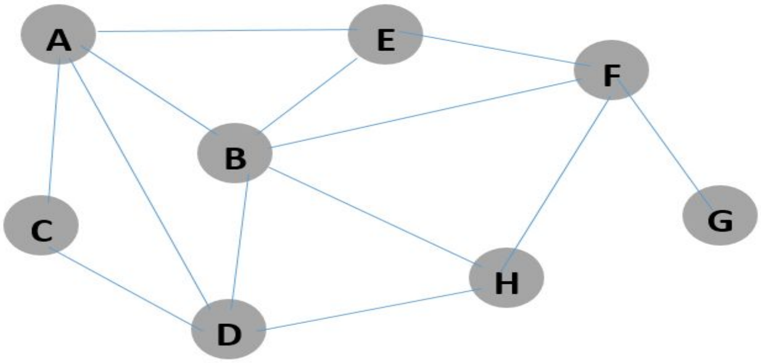
O A B D T

Are there any vertices in the graph that won't be marked solved before T is solved?

F

4) Using the following DFS algorithm and the graph, what is the order in which the vertices will be printed. Start the algorithm with vertex A and push the adjacent vertices in alphabetical order. NOTE: Answer should be in the example format A B C D E F G H where the vertices are separated by space.

```
depthFirstSearchNonRecursive(value)
    vertex = search(value)
    vertex.visited = true
    vertex.distance = 0
    stack.push(vertex)
    while(!stack.isEmpty())
        ve = stack.pop()
        print(v.key)
        for x = 0 to ve.adjacent.end
            if(!ve.adjacent[x].v.visited)
                ve.adjacent[x].v.visited = true
                ve.adjacent[x].v.distance = ve.distance + 1
                stack.push(ve.adjacent[x].v)
```



Solution : A E F H G D C B