Student: 陳玉山
ID： 0710185

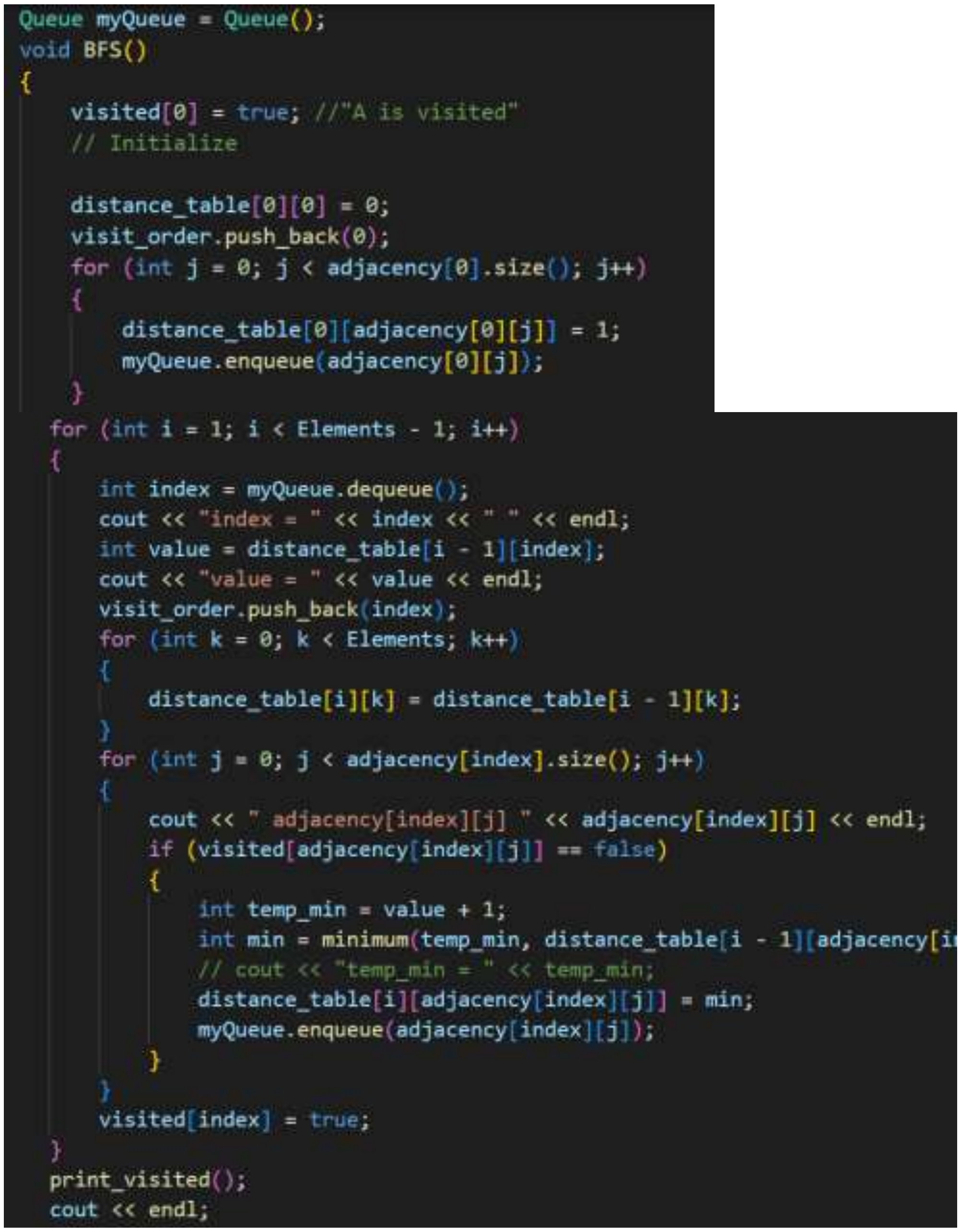
REPORT HOMEWORK 7

I. Introduction.

- In this homework, we try to use two different methods to visit all the nodes of the graph, DFS, and BFS. With DFS, we can use a stack data structure to discover the deepest of the graph and then move to the other branches. While BFS uses a queue to discover nodes layer by layer.

II. Methodology.

Breadth-First Search



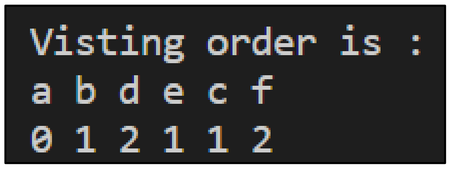
```cpp
Queue myQueue = Queue();
void BFS()
{
    visited[0] = true; //"A is visited"
    // Initialize

    distance_table[0][0] = 0;
    visit_order.push_back(0);
    for (int j = 0; j < adjacency[0].size(); j++)
    {
        distance_table[0][adjacency[0][j]] = 1;
        myQueue.enqueue(adjacency[0][j]);
    }
  for (int i = 1; i < Elements - 1; i++)
  {
      int index = myQueue.dequeue();
      cout << "index = " << index << " " << endl;
      int value = distance_table[i - 1][index];
      cout << "value = " << value << endl;
      visit_order.push_back(index);
      for (int k = 0; k < Elements; k++)
      {
          distance_table[i][k] = distance_table[i - 1][k];
      }
      for (int j = 0; j < adjacency[index].size(); j++)
      {
          cout << " adjacency[index][j] " << adjacency[index][j] << endl;
          if (visited[adjacency[index][j]] == false)
          {
              int temp_min = value + 1;
              int min = minimum(temp_min, distance_table[i - 1][adjacency[i
              // cout << "temp_min = " << temp_min;
              distance_table[i][adjacency[index][j]] = min;
              myQueue.enqueue(adjacency[index][j]);
          }
      }
      visited[index] = true;
  }
  print_visited();
  cout << endl;
```

Figure 1: BFS function.

Student: 陳玉山
ID：0710185

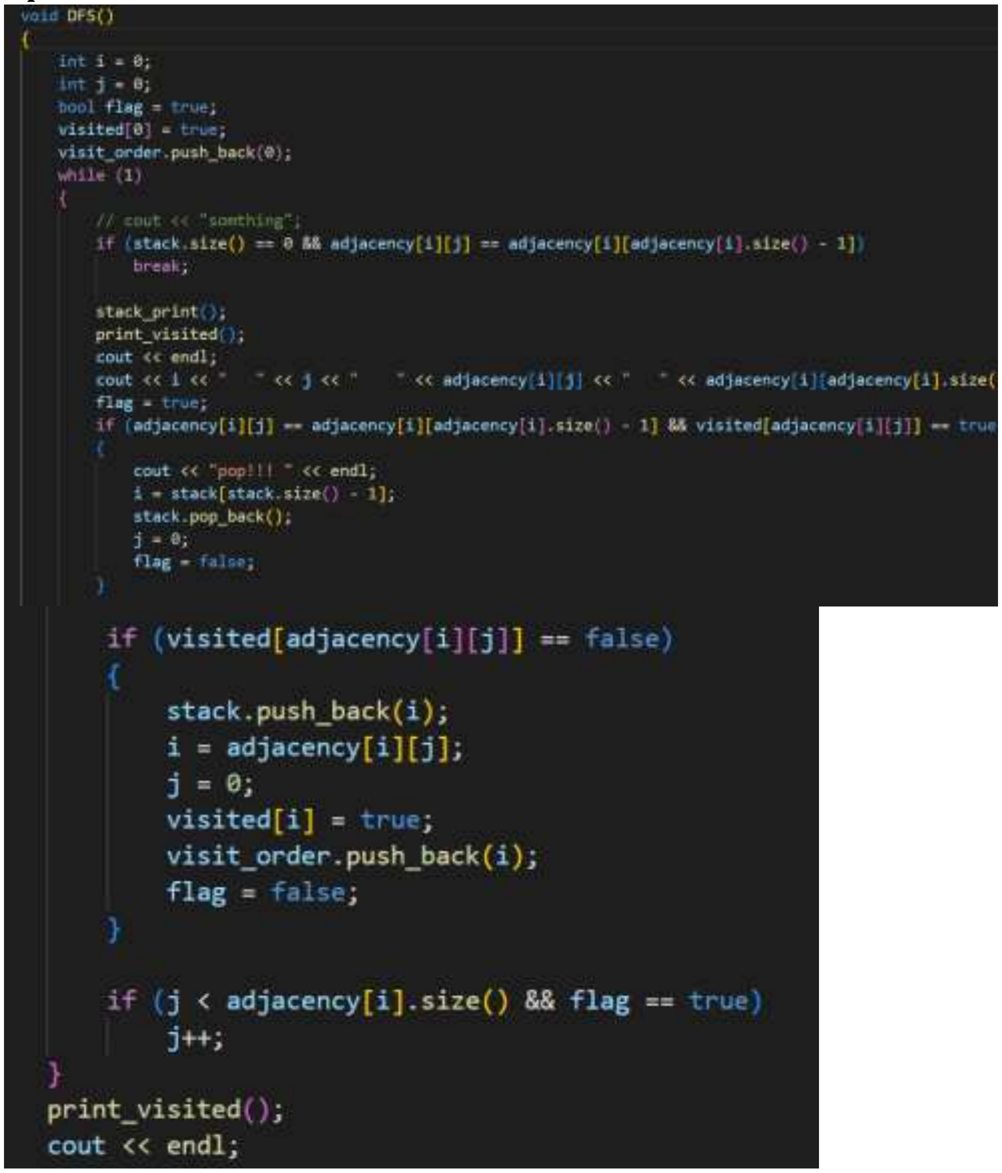
Above figure is my BFS function, this function will use queue to add all the connected nodes with the current node. After visiting a node, the next node will be popped out from the queue. With the property of the queue that “first in, first out”, we can visit the nodes by layer by layer.



```
Visting order is :
a b d e c f
0 1 2 1 1 2
```

Figure 2: The result from running the BFS function.

Depth-first search.



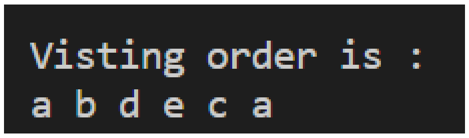
```
void DFS()
{
    int i = 0;
    int j = 0;
    bool flag = true;
    visited[0] = true;
    visit_order.push_back(0);
    while (1)
    {
        // cout << "somthing";
        if (stack.size() == 0 && adjacency[i][j] == adjacency[i][adjacency[i].size() - 1])
            break;

        stack_print();
        print_visited();
        cout << endl;
        cout << i << "    " << j << "    " << adjacency[i][j] << "    " << adjacency[i][adjacency[i].size(
        flag = true;
        if (adjacency[i][j] == adjacency[i][adjacency[i].size() - 1] && visited[adjacency[i][j]] == true
        {
            cout << "pop!!! " << endl;
            i = stack[stack.size() - 1];
            stack.pop_back();
            j = 0;
            flag = false;
        }
        if (visited[adjacency[i][j]] == false)
        {
            stack.push_back(i);
            i = adjacency[i][j];
            j = 0;
            visited[i] = true;
            visit_order.push_back(i);
            flag = false;
        }

        if (j < adjacency[i].size() && flag == true)
            j++;
    }
    print_visited();
    cout << endl;
```

Student: 陳玉山
ID：0710185

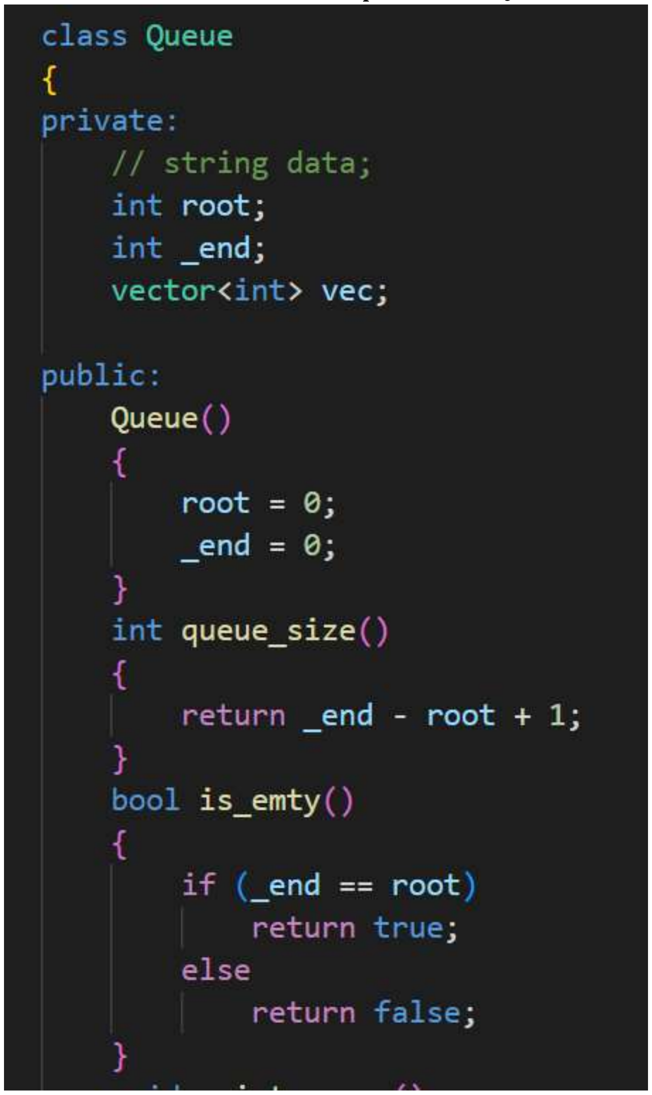
Depth-first search method will make use of stack data structure for the implementation. It will first store the connected node to the stack, then chase the node all the way to the bottom. It chooses the later node to visit by popping out from the stack. With the property of the stack that "last in, first out", we can discover the graph by going deeper into it until reaching the bottom, then moving to the next branch of nodes. Below figure are the results from running the DFS function.



```
Visting order is :
a b d e c a
```

Figure 3: DFS function's result.

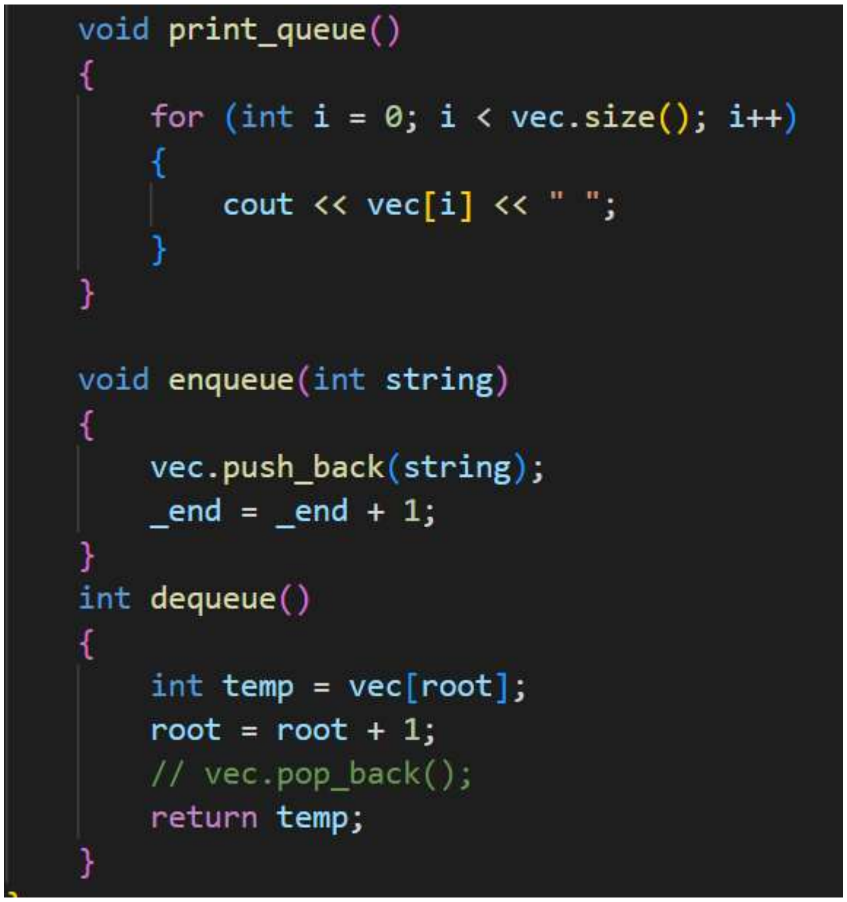
Queue.
For this homework, I also implement a Queue that suits for my purpose.



```
class Queue
{
private:
    // string data;
    int root;
    int _end;
    vector<int> vec;

public:
    Queue()
    {
        root = 0;
        _end = 0;
    }
    int queue_size()
    {
        return _end - root + 1;
    }
    bool is_emty()
    {
        if (_end == root)
            return true;
        else
            return false;
    }
```

Student: 陳玉山
ID： 0710185



```
void print_queue()
{
    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i] << " ";
    }
}

void enqueue(int string)
{
    vec.push_back(string);
    _end = _end + 1;
}
int dequeue()
{
    int temp = vec[root];
    root = root + 1;
    // vec.pop_back();
    return temp;
}
```

III. Discussion.

1. The time complexity of these two methods DFS and BFS are the same for both adjacency matrix (O(V^2)) and adjacency list (O(V+E)).
2. BFS will take more memory than the DFS method because it has to store all the connected nodes of the current node. Meanwhile, the DFS method only stores the node that leads to the bottom of the graph.
3. Application

BFS	DFS
1) GPS navigation systems 2) Find the shortest path/ Minimum spanning tree for the unweighted graph. 3) Broadcasting.	1. Detecting cycle in a graph. 2. Pathfinding. 3. Solving puzzles with only one solution.