HW2

1.

part (1)

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

template <typename T>
struct Qnode{
    T data;
    Qnode *next;

    Qnode(){
        data = 0;
        next = NULL;
    }

    Qnode(T x){
        data = x;
        next = NULL;
    }
};

class Queue{
    int length;
    int capacity;
    Qnode<int> *front;
    Qnode<int> *rear;

public:
    Queue(int cap);

    int size();
    bool isEmpty();
    bool isFull();

    void enqueue(int x);
    int dequeue();
    void print();
};
```

part (2)

```cpp
Queue::Queue(int cap){
    capacity = cap;
    front = NULL;
    rear = NULL;
    length = 0;
}

int Queue::size(){
    return length;
}

bool Queue::isEmpty(){
    return length == 0;
}

bool Queue::isFull(){
    return length == capacity;
}

void Queue::enqueue(int x){
    if(isFull()){
        cout << "The queue is full, terminating program.\n";
        exit(EXIT_FAILURE);
    }
```

part (3)

```cpp
    Qnode<int> *newNode = new Qnode(x);
    length ++;

    if(front == NULL) {
        front = newNode;
        rear = newNode;
    }
    else{
        rear->next = newNode;
        rear = newNode;
    }
    cout << "Enqueued " << x << ".\n";
}

int Queue::dequeue(){
    if(isEmpty()){
        cout << "The queue is empty, terminating program.\n";
        exit(EXIT_FAILURE);
    }

    int value = front->data;
    Qnode<int> *deleteNode = front;
    front = front->next;
    delete deleteNode;
    length--;

    cout << "Dequeued " << value << ".\n";
    return value;
}
```

part (4)

```cpp
void Queue::print(){
    Qnode<int> *current = front;
    cout << "Queue:[\t";
    while(current != NULL){
        cout << current->data << "\t";
        current = current->next;
    }
    cout << "]\n";
}

int main(void){
    Queue q(5);
    cout << "isEmpty = " << q.isEmpty() << endl;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    q.print();
    q.enqueue(4);
    q.enqueue(5);
    q.print();
    cout << "size = " << q.size() << endl;
    cout << "isFull = " << q.isFull() << endl;
    q.dequeue();
    q.dequeue();
    q.print();
    cout << "isFull = " << q.isFull() << endl;
    q.enqueue(6);
    q.enqueue(7);
    q.print();
    q.dequeue();
    q.dequeue();
```

Testing Results

```
isEmpty = 1
Enqueued 1.
Enqueued 2.
Enqueued 3.
Queue:[ 1   2   3   ]
Enqueued 4.
Enqueued 5.
Queue:[ 1   2   3   4   5   ]
size = 5
isFull = 1
Dequeued 1.
Dequeued 2.
Queue:[ 3   4   5   ]
isFull = 0
Enqueued 6.
Enqueued 7.
Queue:[ 3   4   5   6   7   ]
Dequeued 3.
Dequeued 4.
Dequeued 5.
Dequeued 6.
Dequeued 7.
Queue:[ ]
isEmpty = 1
Program ended with exit code: 0
```

2.

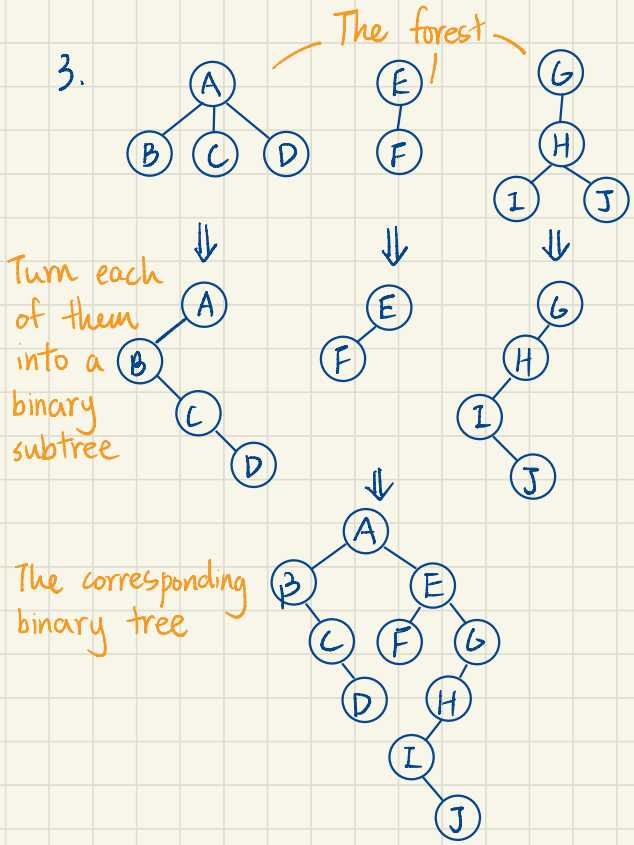
```cpp
void BT::SwapTree(BT *root){
    if(root != NULL){
        BT *temp = root->left;
        root->left = root->right;
        root->right = temp;
        SwapTree(root->left);
        SwapTree(root->right);
    }
}
```

This is the function that swaps the left and right children of every node of a binary tree.


```
4   2   5   1   6   3   7
7   3   6   1   5   2   4
Program ended with exit code: 0
```

This is the result before and after of the swap in inorder traversal.

3.



The original level-order traversal of the forest:
[AEGBCDFHIJ]

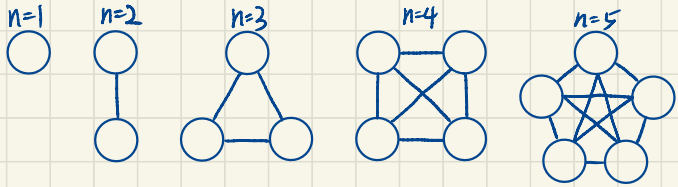
Level-order traversal of its corresponding binary tree:
[ABECFGDHIJ]

From the example, we can see that the level-order traversal of a forest and that of its corresponding binary tree does not always yield the same result.

4. As we all know, the first element in the preorder traversal is the tree's root. We can then look for this element in the inorder traversal. On the left of that element contains the left subtree and vice versa for the right side. We can then continue the process of finding the roots of each subtree in the preorder traversal and part its corresponding subtrees into left and right in the inorder traversal. Therefore, we can construct a unique binary tree based on it preorder and inorder traversal.

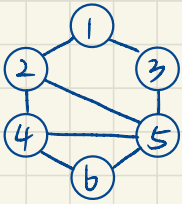
5. Since each edge is connected to two different vertices and the degree of a vertex is counted as the number of edges incident to the vertex, so each edge would be counted twice by the two touching vertices. Thus the sum of the degree of vertices of an undirected graph is twice the number of edges.

6.



A complete graph means that every vertex is connected with every other vertex. So each vertex has $(n-1)$ edges and there are n vertices, thus there will be $n \times (n-1)$ edges in total. However, each edge is counted twice since every edge going out of a vertex is one going into another vertex. Therefore, the final number of edges in an n-vertex complete graph is $n(n-1)/2$.

7.



Take the graph above for example.

A BFS starting from ① would be:
1 → 2 → 3 → 4 → 5 → 6

A BFS starting from ② would be:
2 → 1 → 4 → 5 → 3 → 6

part (1)

```cpp
#include<iostream>
#include <list>
using namespace std;

class Graph{
    int V; // # of vertices
    list<int> *adj;

public:
    Graph(int);

    void addEdge(int, int);
    void printBFS(int);
};

Graph::Graph(int x){
    this->V = x;
    adj = new list<int>[V];
}

void Graph::addEdge(int x, int y){
    adj[x].push_back(y);
}

void Graph::printBFS(int first){
    bool *visited = new bool[V];
    for(int i=0; i<V; i++) visited[i] = false;

    list<int> queue;

    visited[first] = true;
```

part (2)

```cpp
    queue.push_back(first);

    list<int>::iterator i;

    while(!queue.empty()){
        int current = queue.front();
        cout << current << " ";
        queue.pop_front();

        for(i = adj[current].begin(); i != adj[current].end();
            ++i){
            if(!visited[*i]){
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
    cout << "\n";
}

int main(){
    Graph g(7);
    g.addEdge(1, 2); g.addEdge(1, 3);
    g.addEdge(2, 1); g.addEdge(2, 4); g.addEdge(2, 5);
    g.addEdge(3, 1); g.addEdge(3, 5);
    g.addEdge(4, 2); g.addEdge(4, 5); g.addEdge(4, 6);
    g.addEdge(5, 2); g.addEdge(5, 3); g.addEdge(5, 4);
        g.addEdge(5, 6);
    g.addEdge(6, 4); g.addEdge(6, 5);

    int start = 1;
```

part (3)

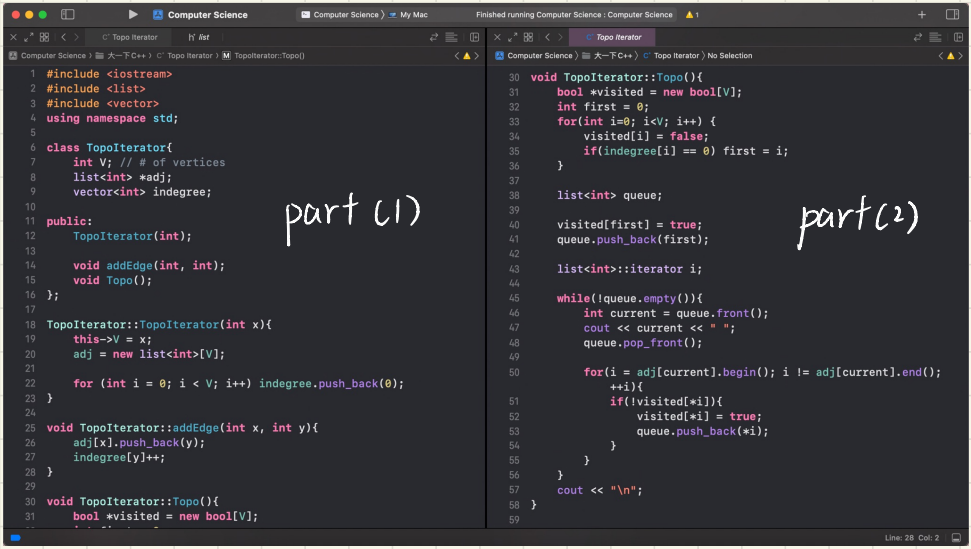
```cpp
int main(){
    Graph g(7);
    g.addEdge(1, 2); g.addEdge(1, 3);
    g.addEdge(2, 1); g.addEdge(2, 4); g.addEdge(2, 5);
    g.addEdge(3, 1); g.addEdge(3, 5);
    g.addEdge(4, 2); g.addEdge(4, 5); g.addEdge(4, 6);
    g.addEdge(5, 2); g.addEdge(5, 3); g.addEdge(5, 4); g.addEdge(5, 6);
    g.addEdge(6, 4); g.addEdge(6, 5);

    int start = 1;
    cout << "Breadth First Search starting from " << start << "\n";
    g.printBFS(start);

    start = 2;
    cout << "Breadth First Search starting from " << start << "\n";
    g.printBFS(start);

    return 0;
}
```

the results

```
Breadth First Search starting from 1
1 2 3 4 5 6
Breadth First Search starting from 2
2 1 4 5 3 6
Program ended with exit code: 0
```

8. Based on Cayley's formula, the number of spanning trees in a n-vertex complete graph is n^{n-2}. When $n \geq 2$, $n^{n-2} \geq 2^{n-1}-1$. Thus the number of spanning trees in a complete graph with n vertices is at least $2^{n-1}-1$.

9.



```cpp
#include <iostream>
#include <list>
#include <vector>
using namespace std;

class TopoIterator{
    int V; // # of vertices
    list<int> *adj;
    vector<int> indegree;

public:
    TopoIterator(int);

    void addEdge(int, int);
    void Topo();
};

TopoIterator::TopoIterator(int x){
    this->V = x;
    adj = new list<int>[V];

    for (int i = 0; i < V; i++) indegree.push_back(0);
}

void TopoIterator::addEdge(int x, int y){
    adj[x].push_back(y);
    indegree[y]++;
}

void TopoIterator::Topo(){
    bool *visited = new bool[V];
```

part (1)

```cpp
void TopoIterator::Topo(){
    bool *visited = new bool[V];
    int first = 0;
    for(int i=0; i<V; i++) {
        visited[i] = false;
        if(indegree[i] == 0) first = i;
    }

    list<int> queue;

    visited[first] = true;
    queue.push_back(first);

    list<int>::iterator i;

    while(!queue.empty()){
        int current = queue.front();
        cout << current << " ";
        queue.pop_front();

        for(i = adj[current].begin(); i != adj[current].end();
            ++i){
            if(!visited[*i]){
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
    cout << "\n";
}
```

part (2)

```cpp
int main(){

    TopoIterator f(6);
    f.addEdge(0, 3); f.addEdge(0, 2); f.addEdge(0, 1);
    f.addEdge(1, 4);
    f.addEdge(2, 4); f.addEdge(2, 5);
    f.addEdge(3, 4); f.addEdge(3, 5);

    cout << "Topological Order: ";
    f.Topo();

    return 0;
}
```

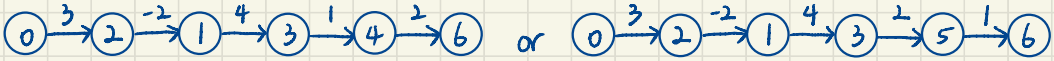
part (3)

```
Topological Order: 0 3 2 1 4 5
Program ended with exit code: 0
```

The result

10. Some Shortest Path Algorithms are greedy algorithms that work with mainly positive numbers because it is the most common case where each weight of the paths are positive. Therefore, these algorithms will not work properly when there exists a negative number in the weight of a path.

The shortest path from 0 to 6 is:



Path length = $3-2+4+1+2 = 8$