Homwork 2

Q1

1/3

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

struct queue
{
    int data;
    queue *next;
    //為什麼next都是指向０？
    queue():data(0),next(0){};
    queue(int x):data(x),next(0){};
};

class queuelist
{
private:
    //為什麼不用說front, back指向哪裡？
    queue *front;
    queue *back;
    int size;
public:
    queuelist():front(0),back(0),size(0){};
    void Push(int x);
    void Pop();
    bool IsEmpty();
    int getFront();
    int getBack();
    int getSize();
};

void queuelist::Push(int x)
{
    if(IsEmpty())
    {
        front = new queue(x);
        back = front;
        size++;
        return;
    }
    queue *newnode = new queue(x);
    back->next = newnode;
    back = newnode;
    size++;
    if(IsEmpty())
        cout << "The queue is empty.\n";
    else
        cout << "FRONT: " << getFront() << " BACK: " << getBack();
}
```

2/3

```cpp
void queuelist::Pop()
{
    if(IsEmpty())
    {
        cout << "The queue is empty.\n";
        return;
    }
    queue *deletenode = front;
    front = front->next;
    delete deletenode;
    deletenode = 0;
    size--;
    if(IsEmpty())
        cout << "The queue is empty.\n";
    else
        cout << "FRONT: " << getFront() << " BACK: " << getBack();
}

int queuelist::getFront()
{
    if(IsEmpty())
    {
        cout << "The queue is empty.\n";
        return -1;
    }
    return front->data;
}

int queuelist::getBack()
{
    if(IsEmpty())
    {
        cout << "The queue is empty.\n";
        return -1;
    }
    return back->data;
}

bool queuelist::IsEmpty()
{
    return ((front && back) == 0);
}

int queuelist::getSize()
{
    return size;
}
```

3/3

```cpp
int main()
{
    queuelist q;
    if(q.IsEmpty())
    {
        cout << "The queue is empty now.";
    }
    cout << "\n\nPush 1, 2 inorder to the queue.\n";
    //為什麼者邊不會輸出兩行FRONT跟BACK?
    q.Push(1);
    q.Push(2);
    cout << "\n\nThe size of the queue now is: " << q.getSize();
    cout << "\n\nPop the first element\n";
    q.Pop();
    cout << "\n\nPush 3 to the queue.\n";
    q.Push(3);
    cout << "\n\nPop the first element\n";
    q.Pop();
    cout << "\n\nPop the first element\n";
    q.Pop();
    return 0;
}
```

```
The queue is empty now.

Push 1, 2 inorder to the queue.
FRONT: 1 BACK: 2

The size of the queue now is: 2

Pop the first element
FRONT: 2 BACK: 2

Push 3 to the queue.
FRONT: 2 BACK: 3

Pop the first element
FRONT: 3 BACK: 3

Pop the first element
The queue is empty.
Program ended with exit code: 0
```

print out

Q2

1/2

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

struct Node
{
    char data;
    struct Node *left, *right;
};

Node* newNode(char data)
{
    Node *temp = new Node;
    temp->data= data;
    temp->left = temp->right = NULL;
    return temp;
}

void swap(Node **a, Node **b)
{
    Node *temp = *a;
    *a = *b;
    *b = temp;
}

void swap_level(Node *root, int level)
{
    //if it is a leaf, then don't have to change
    if(root == NULL || (root->left == NULL && root->right == NULL))
        return;
    //if ( (level + 1) % k == 0)
    swap(&root->left, &root->right);

    //遞迴下去
    swap_level(root->left, level+1);
    swap_level(root->right, level+1);
}

void print(Node *root)
{
    if (root == NULL)
        return;
    print(root->left);
    cout << root->data << " ";
    print(root->right);
}
```

2/2

```cpp
int main()
{

    /*      A
          /   \
         B     C
       /  \   /
      D    E F
       \      \
        G      H */
    struct Node *root = newNode('A');
    root->left = newNode('B');
    root->right = newNode('C');
    root->left->left = newNode('D');
    root->left->right = newNode('E');
    root->right->left = newNode('F');
    root->left->left->right = newNode('G');
    root->right->left->right = newNode('H');

    //所以這個 k=2 是哪來的
    int k = 2;
    cout << "Before swapping the nodes:\n";
    print(root);
    cout << endl;

    swap_level(root, k);
    /*     A
          /   \
         C     B
          \   / \
          F E    D
         /      /
        H      G */

    cout << "\nAfter swapping the nodes:\n";
    print(root);
    return 0;
}
```

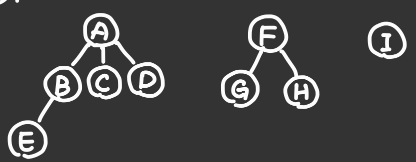
```
Before swapping the nodes:
D G B E A F H C

After swapping the nodes:
C H F A E B G D Program ended with exit code: 0
```

→ print out

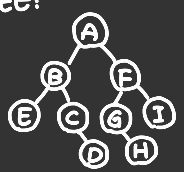
Q3 Counter example :

FOREST:



level order : AFIBCDGHE

BINARY TREE:



level order : ABFECGIDH

→ the level order of the forest and its corresponding binary tree are different #

Q4 preorder & inorder → unique tree

preorder → root → left subtree → right subtree

inorder → left subtree → root → right subtree

① find out the root R_1 from the preorder sequence (the first element of it)

② look at inorder sequence

→ elements in the left side of R_1 will be R_1's left subtree's element

→ right R_1's right subtree's element

③ Repeat ① and ②, and we can find out the roots of every subtree (by ①), find its corresponding subtrees into left and right of it (by ②)

By ①.②.③ → We can defined an unique binary tree by its preorder and inorder sequences. #

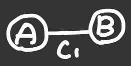
Q5 (Suppose the graph is undirected)

The edge provides both vertex it leads to 1 degree,

Besides, every edge leads to 2 vertex, so an edge can provide 2 degree.

According to that, we can find out that the sum of the degree of vertices of an undirected graph is twice the number of edges. #

(Take a two vertex graph for example



degree(A) = 1

degree(B) = 1 } both provide by C_1)

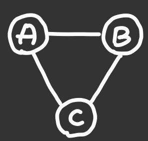
Q6

vertex=1

$\frac{1 \cdot 0}{2} = 0$

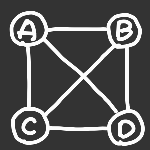
vertex=2

$\frac{2 \cdot 1}{2} = 1$



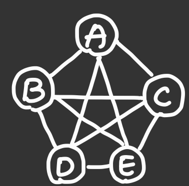
vertex=3

$\frac{3 \cdot 2}{2} = 3$



vertex=4

$\frac{4 \cdot 3}{2} = 6$



vertex=5

$\frac{5 \cdot 4}{2} = 10$

the number of edges:

① every vertex has an edges to all the other vertex → $(n-1)$ for every vertex

② the graph has n vertices → $n(n-1)$

However, we need to divided $n(n-1)$ by 2 to deal with double counting.

⇒ the number of edges in an n-vertex complete graph will be $\frac{n(n-1)}{2}$ #

(

for Ⓐ → (2-1) edge

Ⓑ → (2-1) edge

If we just add Ⓐ's edge and Ⓑ's edge,

the total number of edges will be $2(2-1) = 2$. ——— (*)

However, Ⓐ and Ⓑ share the same edge, we double count it.

So we have to $(\div 2)$ for the (*).

Therefore, the answer will be $2 \div 2 = 1$, and it's correct.)

Q7

1/3

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <queue>

using namespace std;

class graph
{
private:
    int vexNum;
    vector< list<int> > AdjList;
    bool *visit; //0:還沒走, 1:走過了

public:
    graph(int N):vexNum(N)
    {
        AdjList.resize(vexNum);
    };

    void AddEdge(int from, int to);
    void BFS(int Start);
};

void graph::AddEdge(int from, int to)
{
    AdjList[from].push_back(to);
}
```

2/3

```cpp
void graph::BFS(int Start)
{
    visit = new bool[vexNum];
    //先初始化每個位子都還沒走過
    for (int i = 0; i < vexNum; i++)
        visit[i] = 0;

    queue<int> q;

    int s = Start;
    for (int j = 0; j < vexNum; j++)
    {
        if (visit[s] == 0) //還沒走過
        {
            visit[s] = 1;
            q.push(s);
            while (!q.empty())
            {
                int n = q.front(); //新的搜尋起點
                cout << n << " ";
                for (list<int>::iterator k = AdjList[n].begin(); k != AdjList[n].end(); k++)
                {
                    if (visit[*k] == 0) //找到的vertex還沒走過
                    {
                        visit[*k] = 1;
                        q.push(*k); //把vertex推進queue
                    }
                }
                q.pop(); //把u移出queue
            }
        }
        //檢查有沒有沒被走到的
        s = j;
    }
}
```

3/3

```cpp
int main()
{
    graph g(7);
    g.AddEdge(0, 1); g.AddEdge(0, 2); g.AddEdge(0, 3);
    g.AddEdge(1, 0); g.AddEdge(1, 4); g.AddEdge(1, 5);
    g.AddEdge(2, 0); g.AddEdge(2, 6); g.AddEdge(2, 7);
    g.AddEdge(3, 0); g.AddEdge(3, 6);
    g.AddEdge(4, 1); g.AddEdge(4, 5);
    g.AddEdge(5, 1); g.AddEdge(5, 4);
    g.AddEdge(6, 2); g.AddEdge(6, 3);

    cout << "The order of breadth-first search in this graph:\n";
    g.BFS(0);

    return 0;
}
```

```
The order of breadth-first search in this graph:
0 1 2 3 4 5 6 7
```

print out

Q8 ① n=1 → A

n=2 → A—B

n=3 → A—B

C

To connect C with A, B, there're 2 choices (i) connect A—C

(ii) connect B—C

Besides, we have 2 options of each of them (i) connect

(ii) don't connect

In total, we'll have 2^2 ways.

However, when C connects to A and B at the same time, it isn't allowed.

(It'll be against to the definition of spanning tree, because there'll be a circle.)

So, in (n=3) we can have $2^{3-1}-1=3$ spanning trees.

② Suppose that when n=m, there'll be $2^{m-1}-1$ spanning trees.

③ For n=m+1,

(i) From ②, it'll have $2^{(m+1)-1}-1=2^m-1$ spanning trees.

(ii) From graph, there are m previous dots and a new one.

To connect them, we'll have 2^m-1 ways. (The same method with ①)

The answer of (i) = (ii)

④ By Mathematical Induction,

we can find out that the number of spanning trees in a complete graph with n vertices is at least $2^{n-1}-1$ #

Q9

1/3

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <queue>

using namespace std;

class TopoIterator
{
private:
    int vexNum;
    vector< list<int> > AdjList;
    bool *visit; //0:還沒走, 1:走過了

public:
    TopoIterator(int N):vexNum(N)
    {
        AdjList.resize(vexNum);
    };

    void AddEdge(int from, int to);
    void print(int Start);
};

void TopoIterator::AddEdge(int from, int to)
{
    AdjList[from].push_back(to);
}
```

2/3

```cpp
void TopoIterator::print(int Start)
{
    visit = new bool[vexNum];
    //先初始化每個位子都還沒走過
    for (int i = 0; i < vexNum; i++)
        visit[i] = 0;

    queue<int> q;

    int s = Start;
    for (int j = 0; j < vexNum; j++)
    {
        if (visit[s] == 0) //還沒走過
        {
            visit[s] = 1;
            q.push(s);
            while (!q.empty())
            {
                int n = q.front(); //新的搜尋起點
                cout << n << " ";
                for (list<int>::iterator k = AdjList[n].begin(); k != AdjList[n].end(); k++)
                {
                    if (visit[*k] == 0) //找到的vertex還沒走過
                    {
                        visit[*k] = 1;
                        q.push(*k); //把vertex推進queue
                    }
                }
                q.pop(); //把u移出queue
            }
        }
        //檢查有沒有沒被走到的
        s = j;
    }
}
```

```cpp
int main()
{
    TopoIterator g(6);
    g.AddEdge(0, 3); g.AddEdge(0, 2); g.AddEdge(0, 1);
    g.AddEdge(1, 4);
    g.AddEdge(2, 4); g.AddEdge(2, 5);
    g.AddEdge(3, 4); g.AddEdge(3, 4);

    cout << "The topologiacl order of the example graph is:\n";
    g.print(0);

    return 0;
}
```

3/3

```
The topologiacl order of the example graph is:
0 3 2 1 4 5 Program ended with exit code: 0
```

→ print out

Q 10 (a) The method of SHORTESTPATH is greedy,
which means that it'll tend to find out everytime's best solution.
But the length of this graph isn't all positive or all negative,
so SHORTESTPATH might be wrong. #

ex. the shortest path from 0 → 1 :

SHORTESTPATH: $0 \xrightarrow{2} 1$ (2)

($\because 2 < 3$)

the truth: $0 \xrightarrow{3} 2 \xrightarrow{-2} 1$ $(3-2=1)$

(b) The shortest length between 0 and 6

<i> $0 \xrightarrow{3} 2 \xrightarrow{-2} 1 \xrightarrow{4} 3 \xrightarrow{1} 4 \xrightarrow{2} 6$

<ii> $0 \xrightarrow{3} 2 \xrightarrow{-2} 1 \xrightarrow{4} 3 \xrightarrow{2} 5 \xrightarrow{1} 6$

$3-2+4+1+2=8$ #