数据结构实验报告

姓名：阚东　　学号：919106840420

GitHub 链接:
https://github.com/KendrickKan/CPP_NJUST/tree/master/NJUSTHomework/DataStructure

一、一元多项式的相加算法

1．实验内容

编写相应的 C++算法实现一元多项式加法。

2．源代码

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    float coefficient;
    int exp;
    node *next;
};
node *add(node *a, node *b)
{
once:
    node *head = a->exp <= b->exp ? a : b; //如果 a 的首项的指数小于等于 b 那么头节点弄为 a，否则为 b
    node *ha = a->exp <= b->exp ? a : b;
    node *pa = ha;
    node *pb = a->exp <= b->exp ? b : a;
    while (pa != NULL && pb != NULL)
    {
        if (pa->exp < pb->exp)
        {
            ha = pa;
            pa = pa->next;
        }
        else if (pa->exp == pb->exp)
        {
            float tempCoe = pa->coefficient + pb->coefficient;
            //相加不为 0
            if (tempCoe != 0)
            {
                pa->coefficient = tempCoe;
```

```
                ha = pa;
                pa = pa->next;
                pb = pb->next;
            }
            //相加为0
            else
            {
                if (head == ha)
                {

                    //如果两个多项式的第一项相加为0，那么就可能会更换
head
                    node *tempdela = pa;
                    node *tempdelb = pb;
                    a->next = pa->next;
                    b->next = pb->next;
                    delete tempdela;
                    delete tempdelb;
                    goto once;//重新返回入口
                    // if (pa->next != NULL && pb->next != NULL)
                    // {
                    //     head = pa->next->exp <= pb->next->exp ?
pa->next : pb->next;
                    //     node *tempa = pa->next->exp <=
pb->next->exp ? pa->next : pb->next;
                    //     node *tempb = pa->next->exp <=
pb->next->exp ? pb->next : pa->next;
                    //     pa = tempa;
                    //     pb = tempb;
                    // }
                    // else
                    //     return pa->next == NULL ? pb->next :
pa->next;
                }
                ha->next = pa->next;
                node *temp = pa;
                pa = pa->next;
                delete (temp);
                pb = pb->next;
            }
        }
        else
        {
            node *temp = new node;
```

```
            temp->coefficient = pb->coefficient;
            temp->exp = pb->exp;
            temp->next = pa;
            ha->next = temp;
            ha = ha->next;
            pb = pb->next;
        }
    }
    if (pb != NULL)
    {
        if (ha != NULL)
            ha->next = pb;
    }
    return a;
}
void print(node *p) //打印多项式
{
    // cout << p->coefficient << "x^" << p->exp;
    // if (p->next != NULL)
    //     cout << (p->next->coefficient > 0 ? "+" : "-");
    p = p->next; //哨兵
    if (p == NULL)
    {
        cout << 0 << endl;
        return;
    }
    bool flag = false;
    while (p->next != NULL)
    {
        flag = true;
        cout << abs(p->coefficient) << "x^" << p->exp <<
(p->next->coefficient > 0 ? "+" : "-");
        p = p->next;
    }
    if (flag)
        cout << abs(p->coefficient) << "x^" << p->exp << "." << endl;
    else
        cout << p->coefficient << "x^" << p->exp << "." << endl;
}
int main()
{
    int anum, bnum;
    cout << "请输入多项式一的项数>>>";
    cin >> anum;
```

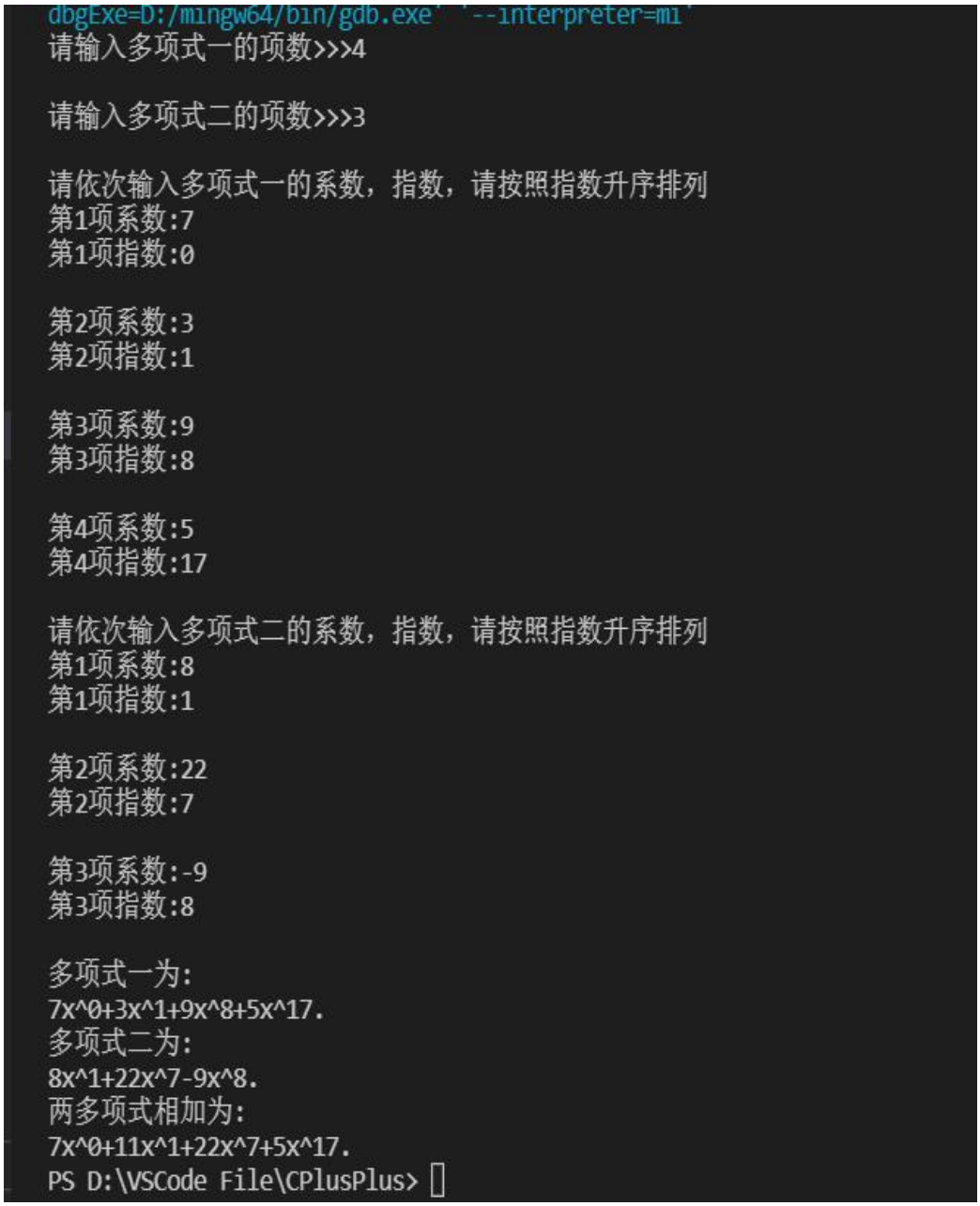
```
    cout << endl;
    cout << "请输入多项式二的项数>>>";
    cin >> bnum;
    cout << endl;
    node *shaobinga = new node;
    shaobinga->next = NULL;
    node *shaobingb = new node;
    shaobingb->next = NULL;
    node *a = shaobinga;
    node *nowa = shaobinga;
    node *b = shaobingb;
    node *nowb = shaobingb;
    cout << "请依次输入多项式一的系数，指数，请按照指数升序排列" <<
endl;
    for (int i = 0; i < anum; i++)
    {
        cout << "第" << i + 1 << "项系数:";
        float tempcoe;
        cin >> tempcoe;
        cout << "第" << i + 1 << "项指数:";
        int tempexp;
        cin >> tempexp;
        cout << endl;
        node *temp = new node;
        temp->coefficient = tempcoe;
        temp->exp = tempexp;
        temp->next = NULL;
        // if (i == 0)
        // {
        //     a = temp;
        //     nowa = a;
        // }
        // else
        // {
        nowa->next = temp;
        nowa = temp;
        // }
    }
    cout << "请依次输入多项式二的系数，指数，请按照指数升序排列" <<
endl;
    for (int i = 0; i < bnum; i++)
    {
        cout << "第" << i + 1 << "项系数:";
        float tempcoe;
```

```
        cin >> tempcoe;
        cout << "第" << i + 1 << "项指数:";
        int tempexp;
        cin >> tempexp;
        cout << endl;
        node *temp = new node;
        temp->coefficient = tempcoe;
        temp->exp = tempexp;
        temp->next = NULL;
        // if (i == 0)
        // {
        //     b = temp;
        //     nowb = b;
        // }
        // else
        // {
        nowb->next = temp;
        nowb = temp;
        // }
    }
    cout << "多项式一为:" << endl;
    print(a);
    cout << "多项式二为:" << endl;
    print(b);
    cout << "两多项式相加为:" << endl;
    print(add(a, b));
    return 0;
}
```

3．算法说明

用两个带有哨兵节点的链表实现两个一元多项式，两个一元多项式按照指数升序排序,通过算法判断将首项指数较小的链表设为head链表，将另一个链表加到 head 链表上从而实现一元多项式相加。在进行相加的时候需要处理（首项）相加是否为零等特殊情况，通过比较两个链表当前指针所指向的项的指数大小,从而实现相加以及相应的指针的后移，最后若非 head 链表还有剩余，将其连接到 head 链表上。

4．实验结果


dbgExe=D:/mingw64/bin/gdb.exe' '--interpreter=mi'
请输入多项式一的项数>>>4
请输入多项式二的项数>>>3
请依次输入多项式一的系数，指数，请按照指数升序排列
第1项系数:7
第1项指数:0
第2项系数:3
第2项指数:1
第3项系数:9
第3项指数:8
第4项系数:5
第4项指数:17
请依次输入多项式二的系数，指数，请按照指数升序排列
第1项系数:8
第1项指数:1
第2项系数:22
第2项指数:7
第3项系数:-9
第3项指数:8
多项式一为:
7x^0+3x^1+9x^8+5x^17.
多项式二为:
8x^1+22x^7-9x^8.
两多项式相加为:
7x^0+11x^1+22x^7+5x^17.
PS D:\VSCode File\CPlusPlus>

二、二叉树的遍历

1．实验内容

输入是一个基于链表存储的二叉树，基于非递归算法，实现一个二叉树的先序、中序以及后序遍历。

2．源代码

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    string data = "";
    node *leftChild;
    node *rightChild;
};
void PreOrderTraversal(node *T) //先序遍历
{
    cout << "PreOrderTraversal 先序遍历" << endl;
    if (T == NULL)
    {
        cout << "该二叉树为空" << endl;
        return;
    }
    stack<node *> sta; //栈
    sta.push(T);
    while (!sta.empty())
    {
        node *topNode = sta.top();
        sta.pop();
        cout << topNode->data << " ";
        //先右孩子入栈，再左孩子入栈
        if (topNode->rightChild != NULL)
            sta.push(topNode->rightChild);
        if (topNode->leftChild != NULL)
            sta.push(topNode->leftChild);
    }
    cout << endl;
    return;
}
void InOrderTraversal(node *T) //中序遍历
{
```

```
    cout << "InOrderTraversal 中序遍历" << endl;
    if (T == NULL)
    {
        cout << "该二叉树为空" << endl;
        return;
    }
    stack<node *> sta; //栈
    //sta.push(T);
    node *cur = T;
    while (!sta.empty() || cur != NULL)
    {
        while (cur != NULL)
        {
            sta.push(cur);
            cur = cur->leftChild; //一直向左，左孩子入栈
        }
        if (!sta.empty())
        {
            cur = sta.top();
            sta.pop();
            cout << cur->data << " ";
            cur = cur->rightChild;
        }
    }
    cout << endl;
    return;
}
void PostOrderTraversal(node *T) //后序遍历
{
    cout << "PostOrderTraversal 后序遍历" << endl;
    if (T == NULL)
    {
        cout << "该二叉树为空" << endl;
        return;
    }
    stack<node *> sta; //栈
    node *cur = T;
    node *pre = NULL;
    while (cur != NULL || !sta.empty())
    {
        while (cur != NULL)
        {
            sta.push(cur);
            cur = cur->leftChild;
```

```
        }
        if (sta.top()->rightChild == NULL || sta.top()->rightChild
== pre)
        {
            cout << sta.top()->data << " ";
            pre = sta.top();
            sta.pop();
        }
        else
            cur = sta.top()->rightChild;
    }
    cout << endl;
    return;
}
int main()
{
    cout << "该程序按照层序生成二叉树" << endl;
    cout << "若你想该节点为空请严格输入“NULL”" << endl;
    cout << "若你想结束输入请严格输入“END”" << endl;
    node *T = NULL;
    string cData;
    queue<node *> Q;
    cin >> cData;
    if (cData != "END" && cData != "NULL")
    {
        node *tempNode = new node;
        tempNode->data = cData;
        tempNode->leftChild = NULL;
        tempNode->rightChild = NULL;
        T = tempNode;
        Q.push(tempNode);
    }
    while (!Q.empty())
    {
        node *temp = Q.front();
        Q.pop();
        cin >> cData;
        if (cData == "END")
            break;
        if (cData == "NULL")
            temp->leftChild = NULL;
        else
        {
            node *tempNode = new node;
```

```
            tempNode->data = cData;
            tempNode->leftChild = NULL;
            tempNode->rightChild = NULL;
            temp->leftChild = tempNode;
            Q.push(tempNode);
        }
        cin >> cData;
        if (cData == "END")
            break;
        if (cData == "NULL")
            temp->rightChild = NULL;
        else
        {
            node *tempNode = new node;
            tempNode->data = cData;
            tempNode->leftChild = NULL;
            tempNode->rightChild = NULL;
            temp->rightChild = tempNode;
            Q.push(tempNode);
        }
    }
    cout << endl;
    cout << "该二叉树的先序遍历为" << endl;
    PreOrderTraversal(T);
    cout << "该二叉树的中序遍历为" << endl;
    InOrderTraversal(T);
    cout << "该二叉树的后序遍历为" << endl;
    PostOrderTraversal(T);
    return 0;
}
```

3. 算法说明

按照层序新建一棵二叉树，使用 STL 的栈辅助实现非递归实现二叉树的先序、中序和后序遍历。

先序：先根节点入栈，当栈非空，先输出根节点，再右孩子入栈，再左孩子入栈。

中序：根节点入栈，

While 栈非空或当前节点不为空：

若当前节点不为空，则当前节点入栈，当前节点指向一直向左，左孩子入栈；

否则若当前节点为空，但栈非空输出栈顶元素，出栈，当前节点为栈顶元素右孩子。

后序：根节点入栈，

While 栈非空或当前节点不为空：

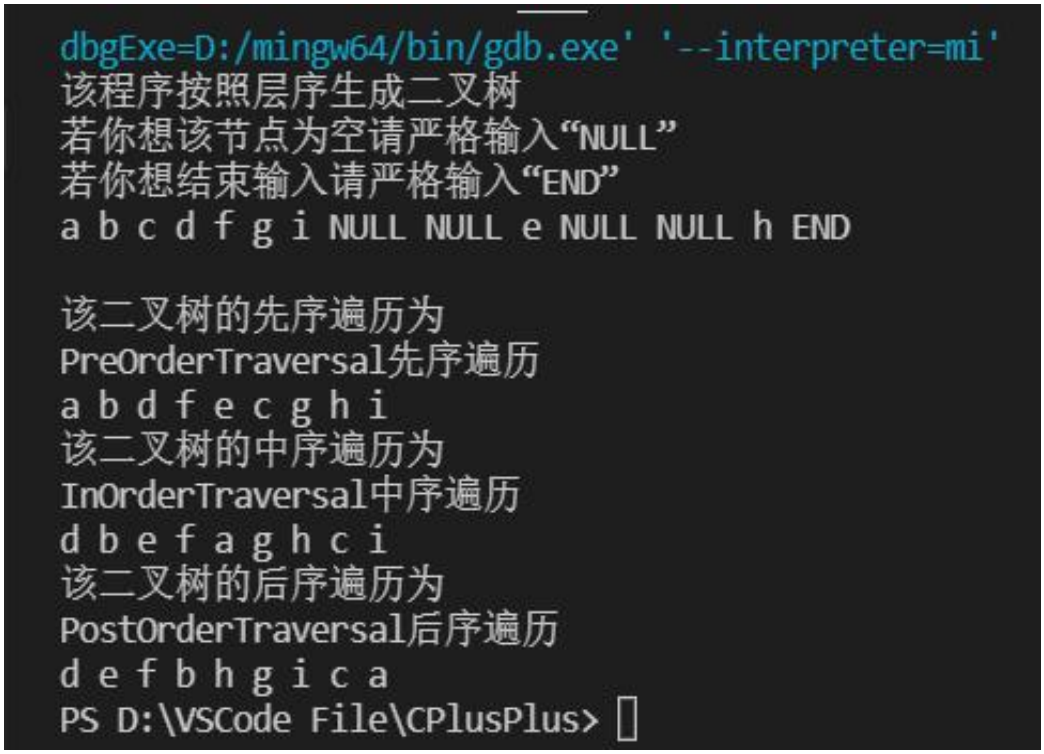
若当前节点不为空，则当前节点入栈，当前节点指向一直向左，左孩子入栈；

否则若当前节点为空，但栈非空输出栈顶元素：

If 栈顶节点为空或者栈顶节点为 pre 节点则输出栈顶节点，pre 指向栈顶节点，出栈，

Else 当前节点为栈顶结点右孩子。

4．运行结果



三、图的遍历及最小生成树算法

1．实验内容

输入是一个基于邻接表或者邻接矩阵存储的无向图，用非递归算法实现深度和者广度遍历一个无向图，并输出遍历结果，注意如果该图不连通，可能需要多次遍历，用非递归算法实现 Kruskal 算法和 Prim 算法。

2．源代码

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100;
const int MAXEDGE = 1e9 + 7;
//邻接矩阵实现图
//该图是一个无向图
struct Edge
{
    int u, v;
    int weight;
};
struct Graph
{
    int e[MAXN][MAXN];
    int edges;
    int ves;
    int DFSflag[MAXN];
    int BFSflag[MAXN];
    vector<Edge> edge; //Kruskal 用
};
int father[MAXN]; //Kruskal 并查集用
bool cmp(Edge a, Edge b)
{
    return a.weight < b.weight;
}
Graph *createGraph(Graph *G)
{
    int vi;
    int vj;
```

```
    cout << "请输入图的顶点个数和边数" << endl;
    cout << "顶点个数:";
    cin >> G->ves;
    cout << "边数:";
    cin >> G->edges;

    //初始化
    for (int i = 0; i < G->ves; i++)
    {
        for (int j = 0; j < G->ves; j++)
        {
            G->e[i][j] = MAXEDGE;
        }
        G->DFSflag[i] = 0; //标识全部置0,表示没有访问过结点
        G->BFSflag[i] = 0;
    }
    //创建邻接矩阵
    cout << "请输入边 edges(vi,vj) 以及该边长度 注意！！！顶点是从0
到VES-1" << endl;
    for (int i = 0; i < G->edges; i++)
    {
        cin >> vi >> vj;
        int weight;
        cin >> weight;
        G->e[vi][vj] = weight;
        G->e[vj][vi] = weight;
        Edge tempEdge;
        tempEdge.u = vi;
        tempEdge.v = vj;
        tempEdge.weight = weight;
        G->edge.push_back(tempEdge);
    }
    return G;
}
void dfs(Graph *G, int ves)
{
    if (G->DFSflag[ves] == 1)
        return;
    stack<int> sta; //创建一个栈
    cout << ves << " ";

    G->DFSflag[ves] = 1; //已经访问过结点ves了
    sta.push(ves);       //该点入栈
```

```
    while (!sta.empty())
    {
        int nowNode;
        int i;

        nowNode = sta.top();
        for (i = 0; i < G->ves; i++)
        {
            if (G->e[nowNode][i] < MAXEDGE && G->DFSflag[i] != 1)
            {
                cout << i << " ";
                G->DFSflag[i] = 1;
                sta.push(i); //该点入栈
                break;
            }
        }
        if (i == G->ves) //data 相邻的结点都访问结束了，就弹出 data
        {
            sta.pop();
        }
    }
    return;
}
void bfs(Graph *G, int ves)
{
    if (G->BFSflag[ves] == 1)
        return;
    queue<int> Q;
    cout << ves << " ";
    G->BFSflag[ves] = 1;
    Q.push(ves);
    while (!Q.empty())
    {
        int nowNode = Q.front();
        Q.pop();
        for (int i = 0; i < G->ves; i++)
        {
            if (G->e[nowNode][i] < MAXEDGE && G->BFSflag[i] != 1)
            {
                cout << i << " ";
                Q.push(i);
                G->BFSflag[i] = 1;
            }
        }
```

```
    }
    return;
}
void Prim(Graph *G)
{
    int start = 0; //开始节点,默认为0
    bool visited[MAXN];
    for (int i = 0; i < G->ves; i++)
        visited[i] = false;
    visited[start] = true;
    int Min[2][MAXN]; //第一行记录目前集合中哪个点到剩余每个点距离最
少，第二行记录目前集合中点到剩余每个点最少的距离
    for (int i = 0; i < G->ves; i++)
    {
        Min[0][i] = start;
        Min[1][i] = G->e[start][i];
    }
    vector<pair<int, int>> vec;
    for (int i = 0; i < G->ves; i++)
    {
        if (i == start)
            continue;
        int minedge = MAXEDGE;
        int node;
        for (int i = 0; i < G->ves; i++)
        {
            if (!visited[i] && Min[1][i] < minedge)
            {
                minedge = Min[1][i];
                node = i;
            }
        }
        vec.push_back(make_pair(Min[0][node], node)); //记录生成
树
        visited[node] = true;
        for (int i = 0; i < G->ves; ++i)
        {
            if (!visited[i] && Min[1][i] > G->e[node][i])
            {
                Min[1][i] = G->e[node][i];
                Min[0][i] = node;
            }
        }
    }
```

```
    //理论来说 vec.size()应该为 G->ves-1
    if (vec.size() != G->ves - 1)
    {
        cout << "该图不连通" << endl;
        return;
    }
    cout << "Prim 算法加入的边为:" << endl;
    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i].first << " " << vec[i].second << endl;
    }
    return;
}
int findfather(int a) //并查集判断是否存在环
{
    while (a != father[a])
    {
        a = father[a];
    }
    return a;
}
void Kruskal(Graph *G)
{
    vector<pair<int, int>> ans;
    sort(G->edge.begin(), G->edge.end(), cmp); //将边集合排序
    for (int i = 0; i < G->ves; i++)
        father[i] = i;
    for (int i = 0; i < G->edge.size() && ans.size() < G->ves - 1;
i++)
    {
        int u = G->edge[i].u;
        int v = G->edge[i].v;
        if (findfather(u) != findfather(v)) //判断父节点是否相同
        {
            ans.push_back(make_pair(u, v));
            father[findfather(u)] = father[findfather(v)]; //将两
点并入一个集合中
        }
    }
    if (ans.size() != G->ves - 1)
    {
        cout << "该图不连通" << endl;
        return;
    }
```

```
    cout << "Kruskal 算法加入的边为:" << endl;
    for (int i = 0; i < ans.size(); i++)
    {
        cout << ans[i].first << " " << ans[i].second << endl;
    }
    return;
}
int main()
{
    Graph *G = new Graph;
    G = createGraph(G);
    cout << "该图从第一个顶点开始，深度优先搜索为:" << endl;
    for (int i = 0; i < G->ves; i++) //遍历整个图，可能不是连通图
        dfs(G, i);
    cout << endl;
    cout << "该图从第一个顶点开始，广度优先搜索为:" << endl;
    for (int i = 0; i < G->ves; i++) //遍历整个图，可能不是连通图
        bfs(G, i);
    cout << endl;
    Prim(G);
    Kruskal(G);
    delete G;
    return 0;
}
```

3. 算法说明

通过邻接矩阵实现一个图。

借助 STL 的栈实现非递归 DFS；

借助 STL 的队列实现非递归 BFS；

借助 STL 的 vector，通过二维数组的第一行记录目前集合中哪个点到剩余每个点距离最少，第二行记录目前集合中点到剩余每个点最少的距离，从而实现非递归 Prim 最小生成树算法；

借助 STL 的 vector，运用并查集思想实现判断该节点是否已经加入最小生成树集合，从而实现 Kruskal 最小生成树算法

4．运行结果

```
dbgExe=D:/mingw64/bin/gdb.exe' '--interpreter=mi'
请输入图的顶点个数和边数
顶点个数:6
边数:10
请输入边 edges(vi,vj) 以及该边长度 注意！！！顶点是从0到VES-1
0 1 6
0 3 5
0 2 1
1 2 5
2 3 5
1 4 3
2 4 6
2 5 4
3 5 2
4 5 6
该图从第一个顶点开始，深度优先搜索为:
0 1 2 3 5 4
该图从第一个顶点开始，广度优先搜索为:
0 1 2 3 4 5
Prim算法加入的边为:
0 2
2 5
5 3
2 1
1 4
Kruskal算法加入的边为:
0 2
3 5
1 4
2 5
1 2
PS D:\VSCode File\CPlusPlus>
```