山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201700130009 | 姓名： 张愈博 | | 班级： 计科17.3 |
| 实验题目：贪婪算法 | | | |
| 实验学时：4 | | 实验日期： 2018.12.22 | |
| 实验目的：  掌握最小生成树的 Prim 算法和 Kruskal 算法的及其实现。 | | | |
| 软件环境：  Windows 10  Dev C++ 5.11 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求）   1）创建加权无向图类。设图没有重边和自环，存储结构分别使用邻接矩阵或邻接链表。提供必要的基本操作；  2）键盘输入图中顶点的个数 n 和边的数目 e，以三元组（i,j,w）形式依次输入图的每一条边或随机生成含 e 条边的图，其中（i,j,w）表示顶点 i 和顶点 j之间拥有权值为 w 的边，建立图。  3)对建立好的图，分别使用 Prim 算法和 Kruskal 算法求最小生成树，输出求得的最小生成树（以文本形式输出生成树中的各条边及对应的权值）。   1. 数据结构与算法描述（整体思路描述，所需要的数据结构与算法）   本题使用的加权无向图在实验12中就已经写好，存储结构选择邻接矩阵。只需在adjacencyWGraph中加入kurskal算法的代码即可。这里kurskal算法的思想较为简单，关键是用什么数据结构实现它。因为其中需要不断提取最小边和判断两个顶点是否在同一顶点集中（判断是否存在环路），所以可选小根堆和树描述的并查集作为数据结构能达到较好的时间复杂度。而Prim算法的思想是以一个顶点开始将未加到最小生成树中的点进行逐步短接。因为需要找一个点集到另一个点集的最短边，所以考虑用小根堆实现，每加入一个新顶点就把该点的所有邻边加入小根堆中。然后不断弹出（与已收录点集外的点之间的）最小边直到堆为空或者顶点数等于n-1为止。   1. 测试结果（测试输入，测试输出，结果分析）     Kurskal Prim  经检查，各项输出正确。   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）     经分析发现在小根堆初始化之前是可以成功返回的，但在初始化之后return就会出现未知断点。但是处理的数据完全正确，不知道为什么会出现bug。经反复查验，发现是kurskal函数返回的时候出现了断点，过程中一切状态正常，错误原因仍然未知。发现在初始化Edge数组时把空间错误的设置为了[顶点数+1]，改为[边数+1]则一切正常。  附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）  Main.cpp  #include<iostream>  #include"adjacencyWGraph.h"  #define noEdge 1000000  using namespace std;  int main()  {  cout<<"请输入图的顶点数n和边数e"<<endl;  int n,e; cin>>n>>e;  adjacencyWGraph<int> AWGraph(n,noEdge);  cout<<"请以顶点对(顶点1,顶点2，权值)的形式输入图的每一条边"<<endl;    for(int i=0;i<e;i++)  {  int v1,v2,w; cin>>v1>>v2>>w;  edge<int> \*Edge=new edge<int> (v1,v2,w);  AWGraph.insertEdge(Edge);  }    edge<int> \*STE=new edge<int> [n-1];  cout<<"最小生成树中每条边(v1,v2,w)如下:"<<endl;  if( AWGraph.Kruskal(STE) ==true)  {  for(int i=0;i<=n-2;i++)  cout<<STE[i]<<endl;  }    return 0;  }  Edge.h  #pragma once  #include<iostream>  using namespace std;  template <class T>  class edge  {  public:  edge() { }  edge(int V1,int V2,T weight)  {  v1=V1; v2=V2; w=weight;  }  ~edge() {};  int vertex1() const { return v1; }  int vertex2() const { return v2; }  T weight() const { return w; }  operator T() const { return w; } //隐式转换，we'll see.  friend ostream& operator<<(ostream& out,const edge<T> A)  {  out << "(" << A.v1 << ", " << A.v2 << ", " << A.w << ")";  return out;  }    protected:  int v1;  int v2;  T w;    };    fastUnionFind.h  #pragma once  #include <iostream>  using namespace std;  class UnionFind  {//ÓÃÊ÷µÄÁ´±íÃèÊöÀ´±íÊ¾²¢²é¼¯£¬ÓÃµ½Ä£ÄâÖ¸Õë  public:  UnionFind(int n)  {  initialize(n);  }  void initialize(int numberOfElements)  {  parent = new int [numberOfElements+1];  for(int e=1;e<=numberOfElements;e++)  parent[e]=0;  }  int find(int ele)  {//·µ»ØeleÔªËØËùÔÚµÄÊ÷¸ù  while(parent[ele]!=0)  {  ele=parent[ele];  }  return ele;  }  void unite(int rootA,int rootB)  {  parent[rootB]=rootA;  }    private:  int \*parent;  };  struct UnionFindNode  {  UnionFindNode()  {  parent=1; root=true;  }  int parent; //ÈôÎª¸ù½Úµã£¬ÔòparentÊÇÊ÷µÄÖØÁ¿£¬·ñÔòÊÇ¸¸½ÚµãµÄÄ£ÄâÖ¸Õë  bool root; //±êÖ¾ÊÇ·ñÎª¸ù½Úµã    };  class fastUnionFind  {//ÓÃÖØÁ¿¹æÔòºÍÂ·¾¶½ôËõÀ´ÓÅ»¯²¢²é¼¯  public:  fastUnionFind(int n)  {  initialize(n);  }  void initialize(int numberOfElements)  {  node=new UnionFindNode [numberOfElements+1];  }  int find(int ele)  {//Â·¾¶½ôËõÔö¼ÓÁËµ¥¸ö²éÕÒµÄ²Ù×÷Ê±¼ä£¬µ«Ëü¼õÉÙÁË´Ëºó²éÕÒ²Ù×÷µÄÊ±¼ä  int theRoot=ele; //theRootÊÇ×îÖÕµÄ¸ù½Úµã  while( !node[theRoot].root )  {  theRoot=node[theRoot].parent;  }  //ÏÂÃæÊÇ½ôËõÂ·¾¶  int currentNode=ele; //´Óele¿ªÊ¼  while( currentNode!=theRoot )  {  int k=node[currentNode].parent;  node[currentNode].parent=theRoot; //ÈÃÄ£ÄâÖ¸ÕëÖ±½ÓÖ¸Ïò¸ù½Úµã  currentNode=k;  }  return theRoot;  }  void unite(int rootA,int rootB)  {//ÓÃÖØÁ¿¹æÔòºÏ²¢¸ù²»Í¬µÄÊýrootAºÍrootB  if(node[rootA].parent<node[rootB].parent)  {//A±È½ÏÇá£¬°ÑA×÷Îª×ÓÊ÷  node[rootB].parent+=node[rootA].parent;  node[rootA].parent=rootB;  node[rootA].root=false;  }  else  {  node[rootA].parent+=node[rootB].parent;  node[rootB].parent=rootA;  node[rootB].root=false;  }  }      private:  UnionFindNode \*node;  };  小根堆.h  #pragma once  #include<iostream>  using namespace std;  template<class T>  class minHeap  {//用数组描述的二叉树表示  public:  minHeap(const int initialCapacity=10)  {  heap=new T [initialCapacity];  arrayLength=heapSize=0;  }  ~minHeap()  {  delete [] heap;  }  bool empty() const  {  if(heapSize==0) return true;  else return false;  }  int size() const  {  return heapSize;  }  const T& top()  {//返回最小元素的引用  return heap[1];  }  void pop()  {//删除最小元素 ,这里只是把最后一个元素踢出heapSize的范围但是并没有删除，有隐患？  if(heapSize==0)  {  cerr<<"堆为空！"<<endl;  return;  }  heap[1].~T(); //删除最小元素，这句话根本没用啊？  //cout<<"heap[1] is"<<heap[1]<<endl;  T lastElement=heap[heapSize]; //删除一个元素，然后重新建堆    //重新堆化：从根开始，为最后一个元素寻找位置  int currentNode=1,child=2;  while(child<=heapSize)  {//首先选出currentNode较小的孩子，然后看看能不能插入  if(child<heapSize && heap[child]>heap[child+1] )  child++;  if(lastElement<=heap[child])  break; //找到一个可以插入的位置，跳出循环  if(lastElement>heap[child] )  {  heap[currentNode]=heap[child];  currentNode=child;  child\*=2;  }  }  heap[currentNode]=lastElement;  heapSize--;  }  void push(T& theElement)  {//把元素theElement加入堆  if(heapSize == arrayLength-1)  {//快满的时候倍长数组  T \*p=new T[2\*arrayLength];  for(int i=0;i<heapSize;i++)  {  p[i]=heap[i];  }  delete [] heap;  heap=p;  arrayLength\*=2;  }  //为元素寻找插入位置  int pos = heapSize+1;  while( pos!=1 && heap[pos/2]>theElement )  {//父节点较大且没有走到根  heap[pos]=heap[pos/2]; //把元素向下移动  pos/=2; //检查节点向上一层  }  heap[pos]=theElement;  heapSize++;  }  void erase(T element)  {//删除方法：首先在二叉树中找到element对应的位置，然后类似pop操作即可。  int i;  for(i=1;i<=heapSize;i++)  {  if(element==heap[i])  break;  }  if(i>heapSize) { cerr<<"erase failed."<<endl; return; }  //成功找到对应位置的情况  heap[i].~T();  T lastElement=heap[heapSize];    int currentNode=i,child=2\*i;  while(child<=heapSize)  {  if(child<heapSize && heap[child]>heap[child+1] )  child++; //这是较小的孩子  if(lastElement<=heap[child])  break; //找到一个可以插入的位置，跳出循环  if(lastElement>heap[child] )  {  heap[currentNode]=heap[child];  currentNode=child;  child\*=2;  }  }  heap[currentNode]=lastElement;  heapSize--;  }  void initialize(T \*theHeap,int theSize)  {//在数组theHeap[1:theSize]中建小根堆  //首先收回原heap空间  delete [] heap;  heap=theHeap;  heapSize=theSize;    //从有叶节点的节点开始堆化  for(int root=heapSize/2;root>=1;root--)  {  T rootElement=heap[root];  int child=root\*2; //child是root的左孩子  int currentNode=root;  while(child<=heapSize) //保证以root为根节点的树是小根堆  {  if(child<heapSize && heap[child] > heap[child+1] )  child++; //保证child是较小子节点  if(heap[child]>=rootElement)  break;  if(heap[child]<rootElement) //相对顺序有讲究的！这里也只能写rootElement，不能写成heap[currentNode]，因为在循环中cur的值会改变  {  heap[currentNode]=heap[child]; //把孩子向上移  currentNode=child;  child\*=2;  }  }  heap[currentNode]=rootElement;  }cout<<"堆初始化正常。"<<endl;  }  void deactivateArray()  {//从析构函数中保留heap  heap = NULL;  arrayLength = heapSize = 0;  }  friend ostream& operator<<(ostream &out,minHeap &x)  {  for(int i=1;i<=x.heapSize;i++)  {  out<<x.heap[i]<<" ";  }  return out;  }      //private:  T \*heap; //保存堆中元素的数组，从1开始存放  int arrayLength; //数组heap的容量  int heapSize; //堆的元素个数  };  template<class T>  void heapSort(T a[],int n)  {//使用堆排序方法给a[1:n]排序  //在数组上建立小根堆  minHeap<T> heap(1);  heap.initialize(a,n);  T b[n+1]; //保存变量的临时数组    //逐个从小根堆中提取元素，从大到小排序  for(int i=1;i<=n-1;i++)  {  T x=heap.top();  heap.pop();  b[i]=x; //把最小元素放在b数组中    cout<<"第"<<i<<"次排序后的结果是：";  for(int j=1;j<=i;j++)  cout<<b[j]<<" ";  cout<<endl;  }  for(int i=1;i<=n-1;i++)  a[i]=b[i];  heap.deactivateArray(); //从堆的析构函数中保留数组a  }  栈.h  //Question:~int()  #pragma once  #include<iostream>  template<class T>  class arrayStack  {  public:  arrayStack(int initialCapacity=10)  {//构造函数  if(initialCapacity<1)  {  std::cerr<<"初始化非法！"<<std::endl;  }  stack=new T[initialCapacity];  stackTop=-1; //代表该栈中没有元素  arrayLength=initialCapacity;  }  ~arrayStack()  {//析构函数  delete [] stack;  }  bool empty() const  {//返回true，当且仅当栈为空  if(stackTop==-1) return true;  else return false;  }  int size() const  {//返回栈中元素个数  return (stackTop+1);  }  T& top()  {//返回栈顶元素的引用  if(stackTop==-1)  {//如果为空栈  std::cerr<<"本栈为空"<<std::endl;  exit(1);  }  return stack[stackTop];  }  T pop()  {//删除栈顶元素,提供改进策略：当pop操作将栈中元素减少到不足原来1/4时，将容量减半  if(stackTop==-1)  {//如果为空栈  std::cerr<<"本栈为空"<<std::endl;  exit(1);  }  T Return=stack[stackTop];  stack[stackTop].~T(); //这里有一个问题，如果是int型变量也会被顺利删除么？  stackTop--;    if(stackTop<arrayLength/4)  {//空间过多，容量减半  T \*temp=new T[arrayLength/2];  for(int i=0;i<=stackTop;i++)  { temp[i]=stack[i]; }  delete [] stack;  stack=temp;  arrayLength=arrayLength/2;  }  return Return;  }  void push(const T& ele)  {//将元素ele压入栈顶  if(stackTop==arrayLength-1)  {//空间不足,容量加倍  T \*temp=new T[2\*arrayLength];  for(int i=0;i<=stackTop;i++)  { temp[i]=stack[i]; }  delete [] stack;  stack=temp;  /\* changeLengthlD(stack,arrayLength,2\*arrayLength); \*/  arrayLength\*=2;  }  stackTop++;  stack[stackTop]=ele;  }  private:  int stackTop; //当前栈顶  int arrayLength; //栈容量  T\* stack; //元素数组  };  adjacencyWGraph.h  #pragma once  #include<iostream>  #include"edge.h"  #include"arrayQueue.h"  #include"栈.h"  #include"小根堆.h"  #include"fastUnionFind.h"  #include<string.h>  using namespace std;  int Minium(int L[],bool S[],int n)  {//在L[1:n]数组中返回不在顶点集S中最小的元素  int min=100000;  for(int i=1;i<=n;i++)  {  if( L[i]<min && S[i]==false )  min=L[i];  }  return min;  }  template<class T>  class adjacencyWGraph  {//加权无向图  protected:  int n; //顶点个数  int e; //边的个数  T \*\*a; //邻接数组  T noEdge; //表示不存在的边  int \*reach; //数组指针  int label=1;    void rDfs(int v)  {//递归实现深度优先搜索  reach[v]=label; int u;  //cout<<"当前走到顶点："<<v<<endl;  myIterator \*iv=iterator(v); //这里加<T>就会报错，为什么？？  while( ( u=iv->next() ) !=0 )  {//以v的下一个邻接点u为新起点递归搜索  //cout<<v<<"的下一个邻接点是："<<u<<" "<<endl;  if(reach[u]!=label)  rDfs(u); //u是一个没有到达过的顶点  }  delete iv;  return;  }    public:  adjacencyWGraph(int v=0,T theNoEdge=0)  {//构造函数  if(v<0) cerr<<"请检验顶点数的合法性！"<<endl;  n=v;  noEdge=theNoEdge;  e=0;    a=new T\*[n+1];  for(int i=0;i<n+1;i++)  {//分配空间  a[i]=new T [n+1];  }  for(int i=1;i<=n;i++) //初始化邻接矩阵  {  for(int j=1;j<=n;j++)  {  a[i][j]=noEdge;  }  }  }  ~adjacencyWGraph()  {  for(int i=0;i<n+1;i++)  {  delete [] a[i];  }  delete [] a;  a=NULL; //不置为NULL的话会变野指针，但是这里也没事  }  class myIterator  {  public:  myIterator(T\* theRow,T theEdge,int v)  {  Row=theRow;  noEdge=theEdge;  n=v;  currentVertex=1;  }  ~myIterator(){ }  int next(T & theWeight)  {//返回指定顶点的下一个邻接点和它的的权值  for(int i=currentVertex;i<=n;i++)  {  if(Row[i]!=noEdge)  {//发现一个邻接点  currentVertex=i+1;  theWeight=Row[i];  return i;  }  }  //不存在下一个邻接点了  currentVertex=n+1;  return 0;  }  int next()  {//返回指定顶点的下一个邻接点  for(int i=currentVertex;i<=n;i++)  {  if(Row[i]!=noEdge)  {//发现一个邻接点  currentVertex=i+1;  //cout<<"i="<<i<<endl;  return i;  }  }  //不存在下一个邻接点了  currentVertex=n+1;  return 0;  }  protected:  T \*Row; //邻接矩阵的的行  T noEdge; //theRow[i]==noEdge，当且仅当没有关联与顶点i的边  int n; //顶点数  int currentVertex; //当前搜索的顶点  };  myIterator\* iterator(int theVertex)  {//返回顶点theVertex的迭代器  checkVertex(theVertex); //cout<<"a[theVertex]="<<a[theVertex];  return new myIterator( a[theVertex], noEdge, n);  }  void checkVertex(int theVertex) const  {//确认是有效顶点  if(theVertex<1 || theVertex>n)  {  cerr<<"the vertex:"<<theVertex<<" is not permissible";  exit(1);  }  else return;  }  bool directed() const { return false; }  void insertEdge(edge<T> \*theEdge)  {//插入边，如果该边已存在，则用theEdge->weight()修改边的权  int v1=theEdge->vertex1();  int v2=theEdge->vertex2();  if( v1<1 || v2<1 || v1>n || v2>n || v1==v2 )  {//越界处理  cerr<<"("<<v1<<","<<v2<<") is not a permissible edge"<<endl;  exit(1);  }  if(a[v1][v2]==noEdge) //新边  e++;  a[v2][v1]=a[v1][v2]=theEdge->weight();  }  void eraseEdge(int i,int j)  {//删除边(i,j),如果不存在该边就什么也不做  if(i>=1 && j>=1 && i<=n && j<=n && a[i][j]!=noEdge )  {  a[i][j]=a[j][i]=noEdge;  e--;  }  }  int Degree(int Vertex)  {//计算顶点Vertex的度  checkVertex(Vertex);  int sum=0;  for(int j=1;j<=n;j++)  if( a[Vertex][j]!=noEdge ) //遍历第Vertex行  sum++;  return sum;  }  bool connected()  {//判断无向图是否连通  if( directed() )  {  cerr<<"本图是无向图！"<<endl; exit(2);  }    reach=new int [n+1];  for(int i=1;i<=n;i++)  reach[i]=0;    dfs(1,reach,1); //给邻接于顶点1的可达顶点做标记    for(int i=1;i<=n;i++)  {//检查所有顶点是否已做标记  if(reach[i]==0)  return false;  }  return true;  }  void bfs(int v,int reach[],int label)  {//广度优先算法，reach[i]用来标记所有邻接于顶点v的可到达的顶点  arrayQueue<int> q(10);  reach[v]=label;  q.push(v);    while(q.empty()!=true)  {  int vertex=q.front();  q.pop();    for(int j=1;j<=n;j++)  {  if( a[vertex][j]!=noEdge && reach[j]==0 )  { q.push(j); reach[j]=label; }  }  }  }  void dfs(int v,int reach[],int label)  {  reach=reach;  label=label;  rDfs(v);  }  int labelComponents(int c[])  {//返回连通分支数，c[i]是顶点i所属的分支序号  int i=0;  for(i=0;i<=n;i++)  c[i]=0;    int label=0; //最后一个构建的编号  for(i=1;i<=n;i++)  {  if( c[i]==0 )  {//对所有未到达的顶点，都进行一次bfs标记  label++;  bfs(i,c,label); //给新分支做标记  }  }  return label;  }  void print\_bfs(int v,int reach[])  {  memset( reach,0,sizeof(int)\*(n+1) );  int label=1; reach[v]=label;  arrayQueue<int> q(10);  q.push(v);    while(q.empty()!=true)  {  int vertex=q.front();  cout<<vertex<<" ";  q.pop();    for(int j=1;j<=n;j++)  {  if( a[vertex][j]!=noEdge && reach[j]==0 )  { q.push(j); reach[j]=label; }  }  }  }  void print\_dfs(int v,int reach[])  {  arrayStack<int> s(10); int u;  memset( reach,0,sizeof(int)\*(n+1) );  reach[v]=label;  s.push(v); cout<<v<<" ";    while(!s.empty()) //当栈不空时  {  int w=s.top();  s.pop();  myIterator \*iv=iterator(w); u=iv->next();  while( u !=0 ) //对上一个到达的点w的所有邻接点  {  if( reach[u]!=label ) //如果存在一个未标记的点j  { reach[u]=label;  s.push(u);  cout<<u<<" ";  iv=iterator(u); //让iv指向以u为起点的迭代器  }  else{//该顶点已被标记  u=iv->next();  if( u ==0 && s.empty()!=true ) s.pop(); //该顶点没有其他邻接点了  }  }  }  }  /\* void bfSpanningTree(int v)  {//生成以v顶点的BFS生成树  arrayQueue<int> q(10);  reach[v]=label;  q.push(v);  TreeNode Node(v); TreeNode \*root=&Node;    while(q.empty()!=true)  {  int vertex=q.front(); TreeNode \*pp=Tree\_search(vetrex,root); //这里应该写一个在树中搜索关键字的函数，返回指向节点的指针  q.pop();    for(int j=1;j<=n;j++)  {  if( a[vertex][j]!=noEdge && reach[j]==0 )  { q.push(j);  reach[j]=label;  TreeNode \*p=new TreeNode(j); //这里建一个新节点  linkNode(pp,p); //将pp与p连接起来  }  }  }  }\*/  int Dijkstra(int start,int dest,int predecessor[])  {//返回最短路长，记录下最短路的路径，predecessor是从start到dest路径中dest前的那个顶点  int L[n+1]; //L[i]就是从start点到顶点i的距离  for(int i=1;i<=n;i++) L[i]=a[start][i]; L[start]=0;  bool S[n+1];  for(int i=1;i<=n;i++) S[i]=false; //S[i]表示start到顶点i的最短路已求得  for(int i=1;i<=n;i++)  {  if(L[i]==noEdge) //对不邻接的点  predecessor[i]=-1;  else  predecessor[i]=start; //邻接点的前驱就是start  }  predecessor[start]=0; //源顶点没有前驱  /\*…………以上是初始化操作…………\*/    while( S[dest]!=true )  {/\*下面找一个不在顶点集S中的u且L[u]标号最小，可以借助一下线性表吗？\*/  int u;  for(u=1;u<=n;u++)  {  if( S[u]==false && L[u]==Minium(L,S,n) ) //这里找一个不在顶点集S且标号最小的点，复杂度可以降到O(1)吗？  {//这里比较粗暴，直接扫描了所有顶点，其实可以用一个collected数组优化成扫描所有已收录顶点  //按我的写法复杂度好像是O(n^3)，可以通过把两个循环判断条件糅合来降低复杂度嘛？不，你的复杂度是O(N^2)  S[u]=true; //把u加入顶点集S中  for(int v=1;v<=n;v++)  {//对每一个不属于S的顶点v  if( S[v]==false && L[u]+a[u][v]<L[v] )  {  L[v]=L[u]+a[u][v];  predecessor[v]=u; //顶点v的前驱是u  }  }  break; //跳到最外层的while循环  }  }  }  return L[dest];  }  bool Kruskal(edge<T> \*spanningTreeEdges)  {//求最小生成树，若该图连通则返回true，且sTE[0:n-2]中存有最小生成树的边  //n是图中的顶点数，e是边数  int n=this->n;  int e=this->e;  edge<T> \*Edge=new edge<T> [e+1];  int k=1; //数组Edge的索引    //下面对edge数组初始化：把所有边加入edge数组  for(int i=1;i<=n;i++)  {//取所有关联顶点i的边  myIterator \*ii=iterator(i);  int j; T w;  while( (j=ii->next(w)) !=0)  {  if(i<j) //避免重复加边  Edge[k++]=edge<int> (i,j,w);  }  }    //用小根堆表示边集的效率较好  minHeap< edge<T> > heap(1);  heap.initialize(Edge,e); cout<<heap<<endl;  fastUnionFind uf(n);  k=0; //索引  while( e>0 && k<n-1 )  {//生成树没有完成并且还有边存在  edge<T> x=heap.top();  heap.pop();  e--;  int a=uf.find(x.vertex1());  int b=uf.find(x.vertex2());  if(a!=b)  {//保证在没有环路的情况下选取边x  spanningTreeEdges[k++]=x;  uf.unite(a,b);  }  }  cout<<"k="<<k<<",n-1="<<n-1<<endl;  if(k==n-1) return true;  else return false;  }    bool Prim(edge<T> \*spanningTreeEdges)  {//spanningTreeEdgesÊÇÒÑ¾­ÈëÑ¡µÄ±ß¼¯  int n=this->n;  int e=this->e;  minHeap< edge<T> > heap;  bool TV [n+1]; //TVÊÇÒÑÔÚÊ÷ÖÐµÄ¶¥µã¼¯  for(int i=1;i<=n;i++)  TV[i]=false;  TV[1]=true;    myIterator \*i = iterator(1);  int j; T w;  while ((j = i->next(w)) != 0)  {  edge<T> tem(1,j,w);  heap.push(tem);  }      int k=0; //±íÊ¾±ß¼¯ÖÐË÷Òý  while( !heap.empty()>0 && k<n-1 )  {  edge<T> x=heap.top(); //µ¯³öµÄ±ßÊÇ×îÐ¡µÄ  heap.pop();  int a=x.vertex1();  int b=x.vertex2();  if( TV[b]==false ) //ÕâÀïTV[a]==true  spanningTreeEdges[k++]=edge<T>(a,b,x.weight());  TV[b]=true;  i=iterator(b);  while( (j=i->next(w))!=0 )  {  if(j>b) //±£Ö¤²»ÖØ¸´¼ÓÈë±ß  {  edge<T> tem(b,j,w);  heap.push(tem);  }  }  }  return (k==n-1);    }  };  arrayQueue.h  //本队列使用映射公式：location(i)=(location(队列首元素)+i)%arrayLength 即环形数组表示法  #pragma once  #include<iostream>  #include<cstdlib>  #include<algorithm>  template<class T>  class arrayQueue  {  public:  arrayQueue(int initialCapacity = 10)  {//构造函数  if(initialCapacity<0) std::cerr<<"队列长度必须大于0！"<<std::endl;  else{  Queue=new T[initialCapacity];  arrayLength=initialCapacity;  qFront=qBack=0; //这里是从Queue[1]开始插入元素  }  }  ~arrayQueue() {delete [] Queue;}  bool empty() const  {  if(qFront==qBack) return true;  else return false;  }  int size() const  {  return (arrayLength+qBack-qFront)%arrayLength;  }  T& front()  {  if(empty()!=true)  return Queue[(qFront+1)%arrayLength];  else  { std::cerr<<"队列为空"<<std::endl; }  }  T& back()  {  if(empty()!=true)  return Queue[qBack];  else  { std::cerr<<"队列为空"<<std::endl; exit(1); }  }  T pop()  {//从队首删除元素  T \*p=&front(); //这里已经判断了队列是否为空  T temp=(\*p);  qFront=(qFront+1)%arrayLength;  (\*p).~T(); //析构首元素好像不能表示int的删除...，就是无法恢复到初始化以前的状态  return temp;  }  void push(const T& ele)  {//从队尾添加元素  if( (qBack+1)%arrayLength==qFront )  {//队列将满，加倍数组长度  T \*newQueue=new T[2\*arrayLength];  int start=(qFront+1)%arrayLength;  if(start==0||start==1)  {//未形成环  std::copy(Queue+start,Queue+qBack+1,newQueue);  }  else  {//形成了环  std::copy(Queue+start,Queue+arrayLength,newQueue);  //复制第2段(start,队列末端，新队列起点）  std::copy(Queue,Queue+qBack+1,newQueue+(arrayLength-start));  //复制第1段（原队列首端，qback,新队列第arraylength-start个位置）  }  qFront=(arrayLength)\*2-1;  qBack=arrayLength-1-1; //重新设置首尾游标  arrayLength=arrayLength\*2;  delete [] Queue;  Queue=newQueue;  }  //把元素插入队列的尾部  qBack=(qBack+1)%arrayLength;  Queue[qBack]=ele;    }  void output()  {  for(int i=qFront;i<qBack;i++)  std::cout<<Queue[i];  std::cout<<std::endl;  }  private:  int qFront; //队列中第一个元素的前一个未知  int qBack; //队列最后一个元素的位置  int arrayLength; //队列的容量  T \*Queue; //队列元素  }; | | | |