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：图 | | | |
| 实验学时：4 | | 实验日期： 2018.12.16 | |
| 实验目的：  1、掌握图的基本概念，图的描述方法；图上的操作方法实现。  2、掌握图结构的应用。 | | | |
| 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）创建无向图类。存储结构分别使用邻接矩阵和邻接链表。提供操作: 插入一条边、删除一条边、遍历、BFS、DFS 等。  2）键盘输入图中顶点的个数 n 和边的数目 e，以顶点对（i，j）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）表示顶点 i 和顶点 j 之间有边相连，建立图。  3）判断图是否连通。若不连通，输出该图的连通分量的个数及每个连通分量中的顶点；  4）对建立好的连通图，键盘输入一顶点，输出从该顶点开始的一个 DFS 序列和 BFS 序列；一个 DFS 生成树和 BFS 生成树（树可以文本形式输出）  5）键盘输入两顶点，输出两顶点之间的最短路径。   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）   1）创建邻接矩阵和临接链表描述的无向图类，并提供基本删除删除遍历操作。  2）确定一个无向图是否连通可从任意一个顶点执行DFS或BFS，然后检验是否所有顶点都被标记为已到达顶点，由此可以判断一个无向图图是否为连通的。若不连通则要判断它的连通分支数和每个连通分支中的顶点。具体方法是反复调用DFS或BFS来给连通分支做标记，从每一个尚未被标记的顶点开始进行搜索，并用新的标号来标记新到达的顶点。这样可以确定每个顶点所在的连通分支，从而也确定了每个连通分支所在的顶点。  3)求BFS树和DFS树，只需稍微修改下BFS和DFS代码即可，但是难点貌似在于树的表示形式。  4)求两点之间的最短路，可以考虑Dijkstra算法实现。先求两点间最短距离，再通过一个数组predecessor保存从源顶点到当前顶点最短路径中的倒数第二的点，从而间接求的最短路径。   1. 测试结果（测试输入，测试输出，结果分析）     左图是用邻接链表描述的加权无向图的结果，右图是用邻接矩阵描述的加权无向图的结果。经检查各项输出均正确。   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   1）operator操作符的隐式转换：如edge<T>::operator T()表名edge的对象可以隐式转换为T类型。  2)由于v1,v2,w在edge类中是保护型成员，所以只能在初始化的时候赋值，那么只有在for循环中初始化并直接插入图中了。  3)使用dfs判断是否连通是出错，应该是数组下标0,1问题，之后再查。现在确定问题是出在迭代器的next函数上。经发现是( u=iv->next() ) !=0少加了一个括号OTZ  4）为什么教材上的dfs功能要分成2个函数来写？  5）noEdge的大小应设为一个相当大的数，但不可以是2147483647，这样稍加运算就会造成溢出。  6）对于邻接链表描述的无向加权图，插入操作总是失败，所有节点的weight都是0？已查出问题，构造函数中，形参与类中声明的变量重名了。  7）对于邻接链表描述的无向加权图，迭代器设计不合理，要是把当前节点指向firstNode，下一次next就直接跳过了首节点。所以在迭代器中，应该先返回当前节点的值，在指向下一个节点。   1. 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"  #include"linkedWGraph.h"  #define noEdge 1000000  using namespace std;  int main1()  {  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);  }  if( AWGraph.connected() ==true) cout<<"本图是连通的。"<<endl;  else{  int c[n+1];  cout<<"连通分支数是："<<AWGraph.labelComponents(c)<<endl;  for(int i=1;i<=n;i++)  cout<<"顶点"<<i<<"属于第"<<c[i]<<"连通分支"<<endl;  }  cout<<"请输入一个顶点序号："; int num; cin>>num;  int reach[n+1];  cout<<"BFS序列："; AWGraph.print\_bfs(num,reach);  cout<<"DFS序列："; AWGraph.print\_dfs(num,reach); cout<<endl;      int predecessor[n+1]; cout<<"请输入起始点和结束点："; int start,dest; cin>>start>>dest;  cout<<endl<<"最短路："<<AWGraph.Dijkstra(start,dest,predecessor)<<endl;  cout<<"最短路径为：";  n=dest; cout<<dest<<" ";  while(predecessor[n]!=0)  {  cout<<predecessor[n]<<" ";  n=predecessor[n];  }  return 0;  }  int main2()  {  cout<<"请输入图的顶点数n和边数e"<<endl;  int n,e; cin>>n>>e;  linkedWGraph<int> LWGraph(n);  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);  LWGraph.insertEdge(Edge);  }  cout<<"邻接链表各节点内容为："<<endl;  LWGraph.output();  if( LWGraph.connected() ==true) cout<<"本图是连通的。"<<endl;  else{  int c[n+1];  cout<<"连通分支数是："<<LWGraph.labelComponents(c)<<endl;  for(int i=1;i<=n;i++)  cout<<"顶点"<<i<<"属于第"<<c[i]<<"连通分支"<<endl;  }  cout<<"请输入一个顶点序号："; int num; cin>>num;  int reach[n+1];  cout<<"BFS序列："; LWGraph.print\_bfs(num,reach);  cout<<"DFS序列："; LWGraph.print\_dfs(num,reach); cout<<endl;      int predecessor[n+1]; cout<<"请输入起始点和结束点："; int start,dest; cin>>start>>dest;  cout<<endl<<"最短路："<<LWGraph.Dijkstra(start,dest,predecessor)<<endl;  cout<<"最短路径为：";  n=dest; cout<<dest<<" ";  while(predecessor[n]!=0)  {  cout<<predecessor[n]<<" ";  n=predecessor[n];  }    return 0;  }  int main()  {  main1();  main2();  return 0;  }  adjacencyWGraph.h  #pragma once  #include<iostream>  #include"edge.h"  #include"arrayQueue.h"  #include"栈.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) )  {  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];  }    };  linkedWGraph.h  #include<iostream>  #include"graphChain.h"  #include"arrayQueue.h"  #include"edge.h"  #include<string.h>  #include"栈.h"  using namespace std;  template<class T>  class linkedWGraph  {//加权无向图  protected:  int n; //顶点数  int e; //边数  graphChain<T> \*aList; //邻接表  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:  void output()  {  for(int i=1;i<=n;i++)  {  cout<<"aList["<<i<<"]="; aList[i].output();  cout<<endl;  }  }  linkedWGraph(int v)  {  if(v>0)  n=v;  e=0;  aList=new graphChain<int> [n+1];  }  ~linkedWGraph()  {  delete [] aList; //这样就会自己调用graphChain的析构函数吗？  }  int numberOfVertices() const { return n; }  int numberOfEdges() const { return e; }  bool directed() const { return false; }  bool weight() const { return true; }  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;  }  void insertEdge(edge<T> \*theEdge)  {  int v1=theEdge->vertex1();  int v2=theEdge->vertex2();  int weight=theEdge->weight();  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(aList[v1].indexOf(v2)==-1)  {//新边  aList[v1].insert(0,v2,weight);  aList[v2].insert(0,v1,weight);  e++;  }  }  void eraseEdge(int i,int j)  {  if( i>1 && j>1 && i<=n && j<=n)  {  int \*v=aList[i].eraseElement(j);  int \*j=aList[j].eraseElement(i); //v,j一定同时为空或者非空，不然说明一致性出了问题  if(v!=NULL && j!=NULL ) e--; //该边存在  }  }  int Degree(int Vertex)  {  checkVertex(Vertex);  return aList[Vertex].size();  }    class myIterator  {  public:  myIterator(graphChain<T> \*theVertex)  {  v=theVertex;  currentVertex=v->firstnode;  }  ~myIterator(){ }  int next(T & theWeight)  {//返回指定顶点的下一个邻接点的序号和它的的权值  if(currentVertex!=NULL)  {  theWeight=currentVertex->weight;  int vertex=currentVertex->element;  currentVertex=currentVertex->next;  return vertex;  }  else  return 0;  }  int next()  {//返回指定顶点的下一个邻接点    if(currentVertex!=NULL)  {  int vertex=currentVertex->element;  currentVertex=currentVertex->next;  return vertex;  }  return 0;  }  protected:  graphChain<T> \*v; //邻接表的点  ChainNode<T> \*currentVertex; //当前搜索的顶点  };  myIterator\* iterator(int theVertex)  {  checkVertex(theVertex);  return new myIterator( &aList[theVertex] );  }    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(ChainNode<int> \*u=aList[vertex].firstnode; u!=NULL;u=u->next)  {  if( reach[u->element]==0 )  { q.push(u->element); reach[u->element]=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(ChainNode<int> \*u=aList[vertex].firstnode; u!=NULL;u=u->next)  {  if( reach[u->element]==0 )  { q.push(u->element); reach[u->element]=label; }  }  }  }  void print\_dfs(int v,int reach[])  {//使用栈实现的DFS搜索算法  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  }  }  }  }  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]=100000; predecessor[i]=-1; } L[start]=0;  for(ChainNode<int> \*u=aList[start].firstnode; u!=NULL;u=u->next)  {  L[u->element]=u->weight;  predecessor[u->element]=start; //对于start邻接的点  }  predecessor[start]=0; //源顶点没有前驱  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    /\*…………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,tag;  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？  { //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？  S[u]=true; //把u加入顶点集S中  tag=u;  break;  }  }    for(ChainNode<int> \*u=aList[tag].firstnode; u!=NULL;u=u->next)  {//遍历刚找到的点的邻接点,如果它没有被访问过，且距离可被更新  //这样写，复杂度是O(链表长)，较好  if(S[u->element]==false && L[tag]+ (u->weight) <L[u->element] )  {  L[u->element]=L[tag]+ u->weight ;  predecessor[u->element]=tag; //顶点v的前驱是u  }  }  }  return L[dest];  }  };  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;    };  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  };  栈.h  #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  }; | | | |