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：201700140056 | 姓名：李港 | | 班级：跟18.2（17.4） |
| 实验题目：实验十二 图 | | | |
| 实验学时：4h | | 实验日期：2019.12.16 | |
| 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。 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容**    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：       1. 插入一条边       2. 删除一条边       3. 遍历：BFS、DFS。       4. 判断全连通       5. 获取各连通分量       6. 获取最短路径    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。 2. **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）**   **总体思路：**   * 1. 为了简化各种算法的编写，本项目采用迭代器对结点访问进行封装，这样可以简化函数编写   2. 连通判断与连通子图的打印：   3. DFS深度优先遍历：可采用递归实现，当一条路径到尽头，函数将返回到上一层，上一层继续对下一个邻接结点调用该递归函数。   4. BFS广度优先搜索：类似于树的层次遍历，采用队列存储被遍历结点，当前结点的所有邻接结点全部存入队列后再读队列，进行下一个遍历   5. 获取最短路径：采用迪杰斯特拉算法（Dijkstra）算法对图进行遍历   6. 差异部分：      1. 邻接链表实现版本：采用链表类存储每个节点的后继结点和权值      2. 邻接矩阵实现版本：使用二维矩阵存储权值   **数据结构：**   * 1. 邻接链表实现版本：      1. 采用链表类存储每个节点的后继结点      2. 由于无权图是权相等的带权图，因此此处编写带权图，通过对权值赋值为1来实现无权图。      3. 不连通的结点无指针相连。   template<class T>  struct node {  T data;  T weight;  node\* next = nullptr;  node (T ele, T w, node\* p = nullptr) :data (ele), weight (w), next (p) {}  node () {}  node& operator=(node& A) {//结点赋值  data = A.data;  weight = A.weight;  next = A.next;  return \*this;  }  };   * 1. 邻接矩阵实现版本：      1. 使用二维矩阵存储矩阵      2. 二维矩阵中，应采用特殊权值表示两结点不相连      3. 由于权值可以是用户自定义的数据结构，因此不能用简单的0来代表无连接的权值,因此采用\_no\_edge这个成员变量存储无连接的值，矩阵中所有等于此值的位置均视为无连接。   **算法：**   * 1. 连通判断与连通子图的打印：      1. 采用某一遍历算法（DFS或BFS）对结点依次进行遍历      2. 每次遍历对结点设置不同的“标签”      3. 不同的连通子图会被设置不同标签      4. 若遍历完成后只有一个标签，则说明只有一个连通子图，该图全连通      5. 若遍历之后有多个标签，则有多个连通子图，只需按照标签编号对子图进行输出即可   2. DFS      1. 深度优先遍历      2. 类似于树的前序遍历      3. 可采用递归实现，当一条路径到尽头，函数将返回到上一层，上一层继续对下一个邻接结点调用该递归函数。   3. BFS      1. 广度优先搜索      2. 类似于树的层次遍历，采用队列存储被遍历结点，当前结点的所有邻接结点全部存入队列后再读队列，进行下一个遍历   4. 获取最短路径：      1. 采用迪杰斯特拉算法（Dijkstra）算法对图进行遍历，取得当前结点到所有结点的单源最短路径，然后通过目标结点获取所需要的那条路径。      2. 该算法为贪心算法      3. S集合为已经找到最短路的结点，U集合为未找到对短路的结点，另有dis数组保存距离值。      4. 初始化：初始情况下S集合只有源结点，U有剩下所有结点，dis数组源节点到源节点距离为0，与源节点相邻的结点距离为权值，到不相邻结点距离无穷大。      5. 进行循环取点：         1. 从dis数组中选取最小数值结点node\_min，从U集合移动到S集合         2. 将与node\_min相邻的所有结点到源点的距离进行更新 dis[node\_min的相邻结点]=dis[node\_min]+weight(node\_min，node\_min的相邻结点)         3. 再次进行本循环，直到U集合没有元素   最短路dijkstra算法详解：dijkstra（图解）（详 - mengxiang000000   1. **测试结果（测试输入，测试输出）**    * 1. 验收展示：      * 1. 平台提交      1. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**   本实验最终结果正确，在实验过程中有以下问题或心得：   * 1. 在编写代码时应尽量考虑重用。      1. 在本实验中，考虑到邻接链表表示与邻接矩阵表示的差异，采用迭代器进行结点访问，方便了迪杰斯特拉算法的编写也可方便下一个实验中普里姆算法和克鲁斯卡尔算法的编写。      2. 直接编写带权图，带权图可以当做无权图来使用，然而无权图却不能当做带权图来使用。   2. 本次实验我学习到了一个新的技巧：遍历贴标签。通过使用标签数组保存连通子图。  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）**   文件1 main.h  #include<iostream>  #include"linkedWGraph.h"  #define noEdge 1000000  using namespace std;  int main(){  #pragma warning(disable:4996)  //freopen ("input.txt", "r", stdin);  cout<<"请输入图的顶点数n和边数e"<<endl;  int n,e;  cin>>n>>e;  linkedWGraph<int> g(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;  g.insertEdge(v1,v2,w);  }  //cout<<"邻接链表各节点内容为："<<endl;  //g.output(cout);  if (g.connected () == true) {  cout << "本图是连通的。" << endl;  } else {  int\* c = new int[n + 1];  int ltfzs = g.getCC (c);  cout << "连通分支数是：" << ltfzs << endl;  for (int i = 1; i <= ltfzs; i++) {  for (int j = 1; j <= n; j++)  if (c[j] == i)cout << j << " ";  cout << "\n";  }  delete[] c;  }  cout<<"请输入一个顶点序号："; int num; cin>>num;  cout<<"BFS序列："; g.printBfs (num);  cout<<"DFS序列："; g.printDfs (num); cout<<endl;    int\* path=new int[n+1];  cout<<"请输入起始点和结束点："; int start,dest; cin>>start>>dest;  cout<<endl<<"最短路："<<g.dijkstra(start,dest,path)<<endl;  cout<<"最短路径为：";  n=dest; cout<<dest<<" ";  while(path[n]!=0){  cout<<path[n]<<" ";  n=path[n];  }  delete[] path;  return 0;  }  文件2 linkedWGraph.cpp  #include<iostream>  #include"stack.h"  #include"queue.h"  #include"chain.h"  #include"minHeap.h"  #include"fastUnionFind.h"  #include"edge.h"  #define until(re) while(!(re))  using namespace std;  //邻接链表加权无向图  template<class T>  class linkedWGraph {  protected:  int \_vertex\_num; //总顶点数  int \_edge\_num; //总边数  chain<T>\* \_chain\_heads; //链表数组  int\* visited; //作为递归遍历时的标记  int \_label = 1; //默认标记  //递归深度优先搜索  void \_dfs (int head) {  visited[head] = \_label;  int u;  myIterator\* iv = iterator (head);  //对该结点的后继结点进行遍历  while ((u = iv->moveNext ()) != 0) {  if (visited[u] != \_label)  \_dfs (u);  }  delete iv;  return;  }  void \_checkVertex (int theVertex) const {//确认是有效顶点  if (theVertex<1 || theVertex>\_vertex\_num) {  cerr << "the vertex:" << theVertex << " is not permissible";  exit (1);  } else return;  }  public:  //输出所有链表  void output (ostream& out) {  for (int i = 1; i <= \_vertex\_num; i++) {  out << "\_chain\_heads[" << i << "]="; \_chain\_heads[i].output ();  out << endl;  }  }  //初始化  linkedWGraph (int vnum) {  \_vertex\_num = vnum;  \_edge\_num = 0;  \_chain\_heads = new chain<int>[\_vertex\_num + 1];  }  //析构  ~linkedWGraph () {  delete[] \_chain\_heads; //这样就会自己调用chain的析构函数吗？  }  int numberOfVertices () const { return \_vertex\_num; }  int numberOfEdges () const { return \_edge\_num; }  bool directed () const { return false; }  bool weight () const { return true; }  void insertEdge (int v1, int v2, int weight) {  if (v1<1 || v2<1 || v1>\_vertex\_num || v2>\_vertex\_num || v1 == v2) {//越界处理  cerr << "(" << v1 << "," << v2 << ") is not a permissible edge" << endl;  exit (1);  }  //若此边不存在，则插入  if (\_chain\_heads[v1].indexOf (v2) == -1) {  \_chain\_heads[v1].insertAfterIndex (0, v2, weight);  \_chain\_heads[v2].insertAfterIndex (0, v1, weight);  \_edge\_num++;  }  }  void eraseEdge (int i, int j) {  if (i > 1 && j > 1 && i <= \_vertex\_num && j <= \_vertex\_num) {  int\* \_head = \_chain\_heads[i].eraseByElement (j);  int\* j = \_chain\_heads[j].eraseByElement (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 (\_head != nullptr && j != nullptr) {  \_edge\_num--;//该边存在  }  }  }  //返回一个结点的后继结点数  int degree (int Vertex) {  \_checkVertex (Vertex);  return \_chain\_heads[Vertex].size ();  }  class myIterator {  protected:  chain<T>\* \_head; //邻接表的点  node<T>\* \_current\_vertex;//当前搜索的顶点  public:  myIterator (chain<T>\* theVertex) {  \_head = theVertex;  \_current\_vertex = \_head->\_head;  }  ~myIterator () {}  //移动到下一个结点，返回索引，修改权重  int moveNext (T& theWeight) {  if (\_current\_vertex != nullptr) {  theWeight = \_current\_vertex->weight;  int vertex = \_current\_vertex->data;  \_current\_vertex = \_current\_vertex->next;  return vertex;  } else {  return 0;  }  }  int moveNext () {  //返回指定顶点的下一个结点的索引，迭代器也自动前移  if (\_current\_vertex != nullptr) {  int vertex = \_current\_vertex->data;  \_current\_vertex = \_current\_vertex->next;  return vertex;  }  return 0;  }  };  myIterator\* iterator (int theVertex) {  \_checkVertex (theVertex);  return new myIterator (&\_chain\_heads[theVertex]);  }  //判断无向图是否连通  bool connected () {  visited = new int[\_vertex\_num + 1];  for (int i = 1; i <= \_vertex\_num; i++) {  visited[i] = 0;  }  //给邻接于顶点1的可达顶点做标记  dfs (1, visited, 1);  //检查所有顶点是否已做标记，全标记为连通，否则不连通  for (int i = 1; i <= \_vertex\_num; i++) {  if (visited[i] == 0) {  return false;  }  }  return true;  }  //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  void bfs (int \_head, int visited[], int label) {  queue<int> q (10);  visited[\_head] = label;  q.push (\_head);  while (q.empty () != true) {  int vertex = q.front ();  q.pop ();  for (node<int>\* u = \_chain\_heads[vertex].\_head; u != nullptr; u = u->next) {  if (visited[u->data] == 0) {  q.push (u->data); visited[u->data] = label;  }  }  }  }  void dfs (int \_head, int visited[], int label) {  visited = visited;  label = label;  \_dfs (\_head);  }  int getCC (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 <= \_vertex\_num; i++) {  c[i] = 0;  }  int label = 0; //最后一个构建的编号  for (i = 1; i <= \_vertex\_num; 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 printBfs (int start) {  bool\* visited = new bool[\_vertex\_num + 1];  memset (visited, 0, sizeof (bool) \* (\_vertex\_num + 1));  visited[start] = true;  queue<int> q (10);  q.push (start);  while (q.empty () != true) {  int vertex = q.front ();  cout << vertex << " ";  q.pop ();  for (node<int>\* u = \_chain\_heads[vertex].\_head; u != nullptr; u = u->next) {  if (visited[u->data] == 0) {  q.push (u->data);  visited[u->data] = true;  }  }  }  }  //打印DFS遍历  void printDfs (int start) {  stack<int> s;  bool\* visited = new bool[\_vertex\_num + 1];  memset (visited, 0, sizeof (bool) \* (\_vertex\_num + 1));  visited[start] = true;  s.push (start);  cout << start << " ";  until (s.empty ()) {  int cur = s.top ();  s.pop ();  myIterator\* iv = iterator (cur);  int u = iv->moveNext ();  while (u != 0) {  if (visited[u] != true) {  visited[u] = true;  s.push (u);  cout << u << " ";  }  //迭代器前进  u = iv->moveNext ();  }  }  }  int dijkstra (int start, int dest, int pre[]) {  //返回最短路长，记录下最短路的路径，pre是从start到dest路径中dest前的那个顶点  //L[i]就是从start点到顶点i的距离  int\* L = new int[\_vertex\_num + 1];  fill (L, L + \_vertex\_num + 1, 100000);  L[start] = 0;  //设置初始距离与初始前驱结点  for (node<int>\* u = \_chain\_heads[start].\_head; u != nullptr; u = u->next) {  L[u->data] = u->weight;  pre[u->data] = start; //对于start邻接的点  }  pre[start] = 0; //源顶点没有前驱  //设置访问记录表  bool\* S = new bool[\_vertex\_num + 1];  fill (S, S + \_vertex\_num + 1, false);  until (S[dest] == true) {  int u, temp\_head;  //获得最短距离的函数  auto getMinDis = [](int L[], bool S[], int \_vertex\_num) {  int min = 100000;  for (int i = 1; i <= \_vertex\_num; i++) {  if (L[i] < min && S[i] == false)  min = L[i];  }  return min;  };  int min\_dis = getMinDis (L, S, \_vertex\_num);  //找到距离起点最近的那个结点，加入到S集合  for (u = 1; u <= \_vertex\_num; u++) {  if (S[u] == false && L[u] == min\_dis) {  S[u] = true;  temp\_head = u;  break;  }  }    //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，则更新距离  for (node<int>\* u = \_chain\_heads[temp\_head].\_head; u != nullptr; u = u->next) {  if (S[u->data] == false && L[temp\_head] + (u->weight) < L[u->data]) {  L[u->data] = L[temp\_head] + u->weight;  pre[u->data] = temp\_head; //顶点v的前驱是u  }  }  }  return L[dest];  }  //针对边，适用于稀疏矩阵的k算法  bool kruskal (edge<T>\* spanningTreeEdges) {  //用小根堆表示边集的效率较好,此处初始化所有边集合  minHeap<edge<T> > heap;  for (int i = 1; i <= \_vertex\_num; i++) {  myIterator\* ii = iterator (i);  int j;  T w;  while ((j = ii->moveNext (w)) != 0) {  if (i < j){//判断去重  heap.push (edge<int> (i, j, w));  }  }  }  //并查集避免环路，并查集，unite为列为集合，find查找集合的编号  fastUnionFind uf (\_vertex\_num);  int k = 0; //索引  while (\_edge\_num > 0 && k < \_vertex\_num - 1) {//生成树没有完成并且还有边存在  edge<T> x = heap.getTop ();  heap.pop ();  \_edge\_num--;  int seta = uf.find (x.getFrom ());  int setb = uf.find (x.getTo ());  if (seta != setb) {//保证在没有环路的情况下选取边x  spanningTreeEdges[k++] = x;  uf.unite (seta, setb);  }  }  return (k == \_vertex\_num - 1);  }  //使用小根堆的prim算法，适用于边多的图  bool prim (edge<T>\* spanningTreeEdges) {  //初始化小根堆，注意跟k不同，这里堆里存的不是所有边，而是当前集合到另一个集合所有点的直接距离  minHeap< edge<T> > heap;  myIterator\* i = iterator (1);  int j; T w;  while ((j = i->moveNext (w)) != 0) {  edge<T> tem (1, j, w);  heap.push (tem);  }  bool\* S = new bool[\_vertex\_num + 1];//TV是已在树中的顶点集  fill (S,S+ \_vertex\_num + 1,false);  S[1] = true;    int k = 0;  while (!heap.empty () && k < \_vertex\_num - 1) {  //添加一个点到S集合  edge<T> x = heap.getTop ();  heap.pop ();  int a = x.getFrom ();  int b = x.getTo ();  if (S[b] == false) {  spanningTreeEdges[k++] = edge<T> (a, b, x.getWeight ());  }  S[b] = true;      //更新堆，注意去重  heap.\_clearAndInit ();  for (int i = 1; i <= \_vertex\_num; i++) {  if (S[i] == true) {  myIterator\* cur = iterator (i);  while ((j = cur->moveNext (w)) != 0) {  if ( S[j]!=true) {  edge<T> tem (i, j, w);  heap.push (tem);  }  }  }  }  }  delete[] S;  return (k == \_vertex\_num - 1);  }  };  文件3 chain.h  #include<iostream>  using namespace std;  template<class T>  struct node {  T data;  T weight;  node\* next = nullptr;  node (T ele, T w, node\* p = nullptr) :data (ele), weight (w), next (p) {}  node () {}  node& operator=(node& A) {//结点赋值  data = A.data;  weight = A.weight;  next = A.next;  return \*this;  }  };  /\*template<class T>  class chain {  protected:  void \_checkIndex (int theIndex) const;  public:  chain ();  chain (int initialCapacity, T A[]);  chain (const chain<T>& A);  ~chain ();  bool empty () const;  int size () const;  T& get (int theIndex) const;  T get\_Weight (int ele) const;  int indexOf (T& theElement) const;  node<T>\* eraseByElement (int theVertex);  void eraseByIndex (int theIndex);  void insertAfterIndex (int theIndex, const T& ele, T weight);  void output () const;  void i\_print () const;  void clear ();  friend ostream& operator<<(ostream& out, const chain<T> A);  class iterator;  \*/  template<class T>  class chain {  public:  chain () {  \_head = nullptr;  \_size = 0;  }  chain (int initialCapacity, T A[]) {  if (initialCapacity > 0) {  \_head = new node<T>;  \_head->data = A[0];  node<T>\* p1 = \_head, \* p2;  for (int i = 1; i < initialCapacity; i++){ //等于1的话前面已经创建好了一个节点  p2 = new node<T>;  p2->data = A[i];  p1->next = p2;  p1 = p2;  }  \_size = initialCapacity;  }  if (initialCapacity <= 0) cerr << "初始长度过小！" << endl;  }  chain (const chain<T>& A) {  \_size = A.\_size;  if (\_size == 0) {//此表为空  \_head = nullptr;  } else {//TargetNode指向A的节点，SourceNode指向此链表的节点  //直接节点赋值会变成浅复制，ciao!  node<T>\* SourceNode = new node<T>;  node<T>\* pre\_node, \* TargetNode;  \_head = SourceNode;  TargetNode = A.\_head;  SourceNode->data = TargetNode->data;  for (int i = 1; i < \_size; i++) { //等于1的话前面已经创建好了一个节点  TargetNode = TargetNode->next;  pre\_node = new node<T> (TargetNode->data); //这里给pre\_node创建了一个新空间并初始化  SourceNode->next = pre\_node;  SourceNode = pre\_node;  }  }  }  ~chain () {  while (\_head != nullptr) {  node<T>\* nextnode = \_head->next;  delete \_head;  \_head = nextnode;  }  }  bool empty () const {return \_size == 0;}  int size () const {return \_size;}  T& get (int theIndex) const {//返回索引是theIndex节点的data内容  \_checkIndex (theIndex);  node<T>\* cur\_node = \_head;  for (int i = 0; i < theIndex; i++) {  cur\_node = cur\_node->next;  }  return cur\_node->data;  }  T getWeight (int ele) const {//返回元素是ele节点的weight  node<T>\* cur\_node;  cout << "vetrex=" << ele;  for (cur\_node = \_head; cur\_node != nullptr; cur\_node = cur\_node->next) {  if (cur\_node->data == ele)  return cur\_node->weight;  }  return -1;  }  int indexOf (T& theElement) const {//返回节点元素内容是theElement的节点索引，若未找到则返回-1  node<T>\* cur\_node = \_head;  for (int i = 0; i < \_size; i++) {  if (cur\_node->data == theElement)  return i;  else  cur\_node = cur\_node->next;  }  return -1;  }  node<T>\* eraseByElement (int theVertex) {//搜索链表并查找顶点等于theVertex的元素，若找到则删除它并返回这个元素的指针  node<T>\* p = \_head, \* tp = nullptr;  while (p != nullptr) {  if (p->data == theVertex) {  if (p == \_head) //删头结点  \_head = \_head->next;  else {  tp->next = p->next;  delete p;  return p;  }  } else {//元素不等于theVertex  tp = p;  p = p->next;  }  }  return nullptr; //找不到的话返回nullptr  }  void eraseByIndex (int theIndex) {//从链表中删除索引为theIndex的节点  \_checkIndex (theIndex);  node<T>\* deletenode;  if (theIndex == 0) {//删除头结点  deletenode = \_head;  \_head = \_head->next;  } else {  node<T>\* p = \_head;  for (int i = 0; i < theIndex - 1; i++) {  p = p->next;  }  //此时已经找到要删除的结点的前驱节点  deletenode = p->next;  p->next = p->next->next;  }  \_size--;  delete deletenode;  }  void insertAfterIndex (int theIndex, const T& ele, T weight) {//将ele元素插入索引为theIndex位置的新节点  \_checkIndex (theIndex);  if (theIndex == 0) {//插入头节点  node<T>\* p = new node<T> (ele, weight, \_head);  \_head = p;  } else {  node<T>\* p = new node<T> (ele, weight);//待插入节点  node<T>\* cur\_node = \_head;  for (int i = 0; i < theIndex - 1; i++) {  cur\_node = cur\_node->next;  }  //此时已经找到要插入的结点的前驱节点  p->next = cur\_node->next;  cur\_node->next = p;  }  \_size++;  }  void output () const {  if (\_size == 0) { cout << "null" << endl; return; } else {  node<T>\* p = \_head;  for (int i = 0; i < \_size; i++) {  cout << "vertex=" << p->data << ",weight=" << p->weight << "";  p = p->next;  }  }  }  void clear () {//清表操作  node<T>\* deletenode, \* p = \_head;  for (int i = 0; i < \_size; i++) {  deletenode = p;  p = p->next;  delete deletenode;  }  \_head = nullptr;  \_size = 0;  }  friend ostream& operator<<(ostream& out, const chain<T> A) {//这里要调用复制构造函数  if (A.\_size == 0) { out << "null" << endl; return out; } else {  node<T>\* p = A.\_head;  for (int i = 0; i < A.\_size; i++) {  out << p->data << " " << p->weight << " ";  p = p->next;  }  return out;  }  }  class iterator {//构造函数、\*&、前后自加、！=、==  protected:  node<T>\* \_node = nullptr;  public:  iterator (node<T>\* theNode) {  \_node = theNode;  }  iterator () {}  T& operator\*() { return \_node->data; } //取内容运算符，作为引用返回  T\* operator->() { return &(\_node->data); } //取地址运算符，作为指针返回  iterator& operator++() {//前加  \_node = \_node->next;  return \*this;  }  iterator operator++(int) {//后加  iterator temp = \*this;  \_node = \_node->next;  return temp;  }  bool operator!=(const iterator I)const {  if (\_node == I.\_node) return false; //这里我没有定义\_node的==，为什么没有报错？  else return true;  }  bool operator==(const iterator I)const {  if (\_node == I.\_node) return true;  else return false;  }  };  //protected:  void \_checkIndex (int theIndex) const {//确定索引的有效性，防止越界  if (theIndex<0 || theIndex>\_size) {  cerr << "index=" << theIndex << ",listSize=" << \_size << "请检查索引的有效性！" << endl;  }  }  node<T>\* \_head; //指向链表第一个元素的指针  int \_size;//链表的元素个数  };  文件4 edge.h  #pragma once  #include<iostream>  using namespace std;  template <class T>  class edge {  protected:  int from;  int to;  T w;  public:  edge () {}  edge (int v1, int v2, T weight):from(v1),to(v2),w(weight){}  ~edge () {};  int getFrom () const { return from; }  int getTo () const { return to; }  T getWeight () const { return w; }  operator T() const { return w; }  friend ostream& operator<<(ostream& out, const edge<T> A) {  out << "(" << A.from << ", " << A.to << ", " << A.w << ")";  return out;  }  };  文件5 fastUnionFind.h  #pragma once  #include <iostream>  using namespace std;  class UnionFind {//用树的链表描述来表示并查集，用到模拟指针  public:  UnionFind (int numberOfElements) {  parent = new int[numberOfElements + 1];  for (int i = 1; i <= numberOfElements; i++) {  parent[i] = 0;  }  }  int find (int ele) {//返回ele元素所在的树根  while (parent[ele] != 0) {  ele = parent[ele];  }  return ele;  }  void unite (int rootA, int rootB) {  parent[rootB] = rootA;  }  private:  int\* parent;  };  /\*\*\*\*\*\*\*\*\*\*\*\*一下是应用了重量规则和路径紧缩优化的快速并查集算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  struct UnionFindNode {  int parent;//若为根节点，则parent是树的重量，否则是父节点的模拟指针  bool root; //标志是否为根节点  UnionFindNode ():parent(1),root(true) {}  };  class fastUnionFind {//用重量规则和路径紧缩来优化并查集  public:  fastUnionFind (int numberOfElements) {  node = new UnionFindNode[numberOfElements + 1];  }  int find (int ele) {//路径紧缩增加了单个查找的操作时间，但它减少了此后查找操作的时间  int theRoot = ele;//theRoot是最终的根节点  while (!node[theRoot].root) {  theRoot = node[theRoot].parent;  }  //下面是紧缩路径  int currentNode = ele; //从ele开始  while (currentNode != theRoot) {  int k = node[currentNode].parent;  node[currentNode].parent = theRoot;//让模拟指针直接指向根节点  currentNode = k;  }  return theRoot;  }  void unite (int rootA, int rootB) {  //用重量规则合并根不同的数rootA和rootB  if (node[rootA].parent < node[rootB].parent) {  //A比较轻，把A作为子树  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;  };  文件6 minHeap.h  #pragma once  template<class T>  /\*分布式排序\*/  class minHeap {  public:  typedef enum { min\_head\_empty }err;  private:  int \_size;  int \_length;  T\* \_head;  void \_extLength () {  T\* temp = new T[\_length \* 2];  copy (\_head, \_head + \_length, temp);  delete[] \_head;  \_length \*= 2;  \_head = temp;  }  void \_clear () {  /\*delete[] \_head;\*/  }  public:  minHeap (int lengthi = 10) {  \_length = lengthi + 1;  \_head = new T[\_length];  \_size = 0;  }  void initialize (T\* arri, int sizei) {  \_clearAndInit ();  for (int i = 1; i <= sizei; i++) {  push (arri[i]);  }  }  ~minHeap () { \_clear (); }  void \_clearAndInit () {  \_clear ();  \_length = 11;  \_head = new T[11];  \_size = 0;  }  const T& getTop () {  if (\_size == 0)  throw min\_head\_empty;  return \_head[1];  }  void pop () {  if (\_size == 0) {  throw min\_head\_empty;  }  \_head[1].~T ();  T getTo\_be\_insert = \_head[\_size--];  int insert\_index = 1,  child\_index = 2; // child\_index of current\_node  //将新的头部元素逐层向下移动，向下移动到左子还是右子？这里需要判断  //起码有一个左子树，所以要<=  while (child\_index <= \_size) {  //如果左子比右子大，则根应当与右子交换，使新根小，这样可以保持最小堆特性  //如果左子树卡到了size位置，说明没有右子树，不必寻找左右中最小的元素  if (child\_index < \_size && \_head[child\_index] > \_head[child\_index + 1]) {  child\_index++;  }  //如果根比两个子都小，那直接退出就行了，不必再交换  if (getTo\_be\_insert <= \_head[child\_index]) {  break;  }  \_head[insert\_index] = \_head[child\_index];  insert\_index = child\_index;  child\_index \*= 2;  }  \_head[insert\_index] = getTo\_be\_insert;  }  void push (const T& datai) {  //进行越界检查  if (\_size == \_length - 1) {  \_extLength ();  }  int insert\_index = ++\_size;  while (insert\_index != 1 && \_head[insert\_index / 2] > datai) {//插入元素的父元素不小于插入元素，说明需要调整  \_head[insert\_index] = \_head[insert\_index / 2]; //该父元素放到子节点位置  insert\_index /= 2;//子节点位置指向原父节点那里去，也即发生父子交换，只不过子元素还没有插入  //继续循环查看新的父节点  }  \_head[insert\_index] = datai;  }  void pushShow (const T& datai) {  //进行越界检查  if (\_size == \_length - 1) {  \_extLength ();  }  int insert\_index = ++\_size;  \_head[insert\_index] = datai;  for (int i = 1; i <= \_size; i++) {  cout << \_head[i] << " ";  }  cout << "\n";  while (insert\_index != 1 && \_head[insert\_index / 2] > datai) {//插入元素的父元素不小于插入元素，说明需要调整  \_head[insert\_index] = \_head[insert\_index / 2]; //该父元素放到子节点位置  insert\_index /= 2;//子节点位置指向原父节点那里去，也即发生父子交换，只不过子元素还没有插入  \_head[insert\_index] = datai;  //继续循环查看新的父节点  for (int i = 1; i <= \_size; i++) {  cout << \_head[i] << " ";  }  cout << "\n";  }  \_head[insert\_index] = datai;  }  ostream& out (ostream& out) {  for (int i = 1; i <= \_size; i++) {  out << \_head[i] << " ";  }  return out;  }  bool empty () const { return \_size == 0; }  int size () const { return \_size; }  };  文件7 queue.h  template<class T>  class queue {  public:  queue (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  }  }  ~queue () { 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\* new\_queue = 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, new\_queue);  } else {//形成了环  std::copy (\_queue + start, \_queue + arrayLength, new\_queue);  //复制第2段(start,队列末端，新队列起点）  std::copy (\_queue, \_queue + qBack + 1, new\_queue + (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 = new\_queue;  }  //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  };  文件8 stack.h  template<class T>  class stack {  public:  typedef enum { pointer\_is\_null, newLength\_less\_than\_zero, stack\_empty } stack\_err;  protected:  void \_exLength () {  T\* temp = new T[bufferLength \* 2];  memcpy (temp, head, bufferLength \* sizeof (T));  delete[] head;  head = temp;  bufferLength = bufferLength \* 2;  }  void \_shLength () {  T\* temp = new T[bufferLength / 2];  memcpy (temp, head, bufferLength / 2 \* sizeof (T));  delete[] head;  head = temp;  bufferLength = bufferLength / 2;  }  int stackTop;  int bufferLength;  T\* head;  public:  stack (int initialCapacity = 10) {  bufferLength = initialCapacity;  head = new T[bufferLength];  stackTop = -1;  }  ~stack () { delete[] head; }  bool empty () const { return stackTop == -1; }  int size () const { return stackTop + 1; }  T top () {  //判断是否为空  //有的实现版本中getTop输入引用，栈空则返回原数值，这样很不符合职责单一原则  if (stackTop == -1) {  throw stack\_empty;  }  return head[stackTop];  }  void pop () {  if (stackTop == -1) {  throw stack\_empty;  }  head[stackTop--].~T ();  /\*用于缩小缓冲区的代码\*/  if (size () < bufferLength / 4) {  \_shLength ();  }  }  void push (const T theElement) {  //判断缓冲区长度并扩大缓冲区  if (stackTop == bufferLength - 1) {  \_exLength ();  }  head[++stackTop] = theElement;  }  }; | | | |