|  |  |  |  |
| --- | --- | --- | --- |
| **学生学号** | 0121710870520 | **实验课成绩** |  |



**学 生 实 验 报 告 书**

|  |  |
| --- | --- |
| **实验课程名称** | 数据结构与算法综合实验 |
| **开课学院** | 计算机科学与技术学院 |
| **指导教师姓名** | 李晓红 |
| **学生姓名** | 张宇航 |
| **学生专业班级** | 软件工程zy1701 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2018 | -- | 2019 | 学年 | 第 | 2 | 学期 |

实验课程名称： 数据结构与算法综合实验

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目名称 | 图与景区信息管理系统实践 | | | 报告成绩 |  |
| 实验者 | 张宇航 | 专业班级 | 软件工程zy1701 | 组别 |  |
| 同组者 |  | | | 完成日期 | 2019年5月21日 |
| 第一部分：实验分析与设计（可加页）   1. 实验目的和具体内容   1.实验目的   * **掌握图的定义和图的存储结构。** * **掌握图的创建方法和图的应用。**   **使用C++语言，定义图的数据结构，结合迭代开发思路实现“景区信息管理系统”**  2.实验内容  **2.1 任务背景**  **有一个景区，景区里面有若干个景点，景点之间满足以下条件：**  **1) 某些景点之间铺设了道路(相邻)**  **2) 这些道路都是可以双向行驶的(无向图)**  **3) 从任意一个景点出发都可以游览整个景区(遍历连通图)**    **2.2 景点数据**  **景区的数据包含景点信息和景点之间的道路信息。分别由两个文本文件存储。**  **Vex.txt文件用来存储景点信息；**  **Edge.txt文件用来存储道路信息。**     1. 分析与设计   **3.1 程序设计**  **使用Mircosoft Visual Studio 2010开发工具，创建一个空的控制台工程(Win32 Console Application)。**  **利用图的存储结构来保存景区景点图，使用C++语言开发景区信息管理系统，工程名为GraphCPro。**  **(1) 添加Graph.h和Graph.cpp文件，用来定义图的数据结构，实现图的相关操作。**  **(2) 添加Tourism.h和Tourism.cpp文件，用来实现景区信息管理系统的相关功能。**  **(3) 添加Main.cpp文件，在文件中创建程序入口函数int main(void)。**    **程序将使用两个主要文件：Graph.cpp与Tourism.cpp。**  **Graph用于实现图的数据结构和相关操作。**  **Tourism用于实现系统的主要功能。**  **(1)、Graph文件**  **Graph为图的数据结构，用于存储景区景点图，实现图的相关操作。**  **1) 数据结构**  **struct Graph**  **{**  **int m\_aAdjMatrix[20][20]; // 邻接矩阵**  **Vex m\_aVexs[20]; // 顶点数组**  **int m\_nVexNum; // 顶点个数**  **} m\_Graph; // Graph对象，用于存储景区景点图**  **2) 函数**  **void Init(void)，初始化图结构。**  **int InsertVex(Vex sVex)，将顶点添加到顶点数组中。**  **int InsertEdge(Edge sEdge)，将边保存到邻接矩阵中。**  **Vex GetVex(int nVex)，查询指定顶点信息。**  **int FindEdge(int nVex, Edge aEdge[])，查询与指定顶点相连的边。**  **int GetVexnum(void)，获取当前顶点数。**  **(2)、Tourism文件**  **Tourism为景点旅游信息管理功能，用于实现系统的主要功能。**  **void CreateGraph(void)，读取文件，创建景区景点图。**  **void GetSpotInfo(void)，查询指定景点信息，显示到相邻景点的距离。**    **1、图的存储**  **当保存图结构时，即要保存顶点信息，也要保存边。**  **① 数组表示，用一维数组来保存顶点的集合，用二维数组来保存边的集合；**  **② 链表表示，可用邻接表、十字链表等方式存储图的顶点和边的信息。**  **本程序中使用数组表示法存储图：**  **定义一维数组Vex m\_aVexs[20]保存顶点信息，最多允许有20个顶点。**  **定义二维数组(邻接矩阵)int m\_AdjMatrix[20][20]保存边的集合，数组中每个元素的值即为边的权值。**  **定义Vex结构体，存储图的顶点信息。**  **struct Vex**  **{ int num; // 景点编号**  **char name[20]; // 景点名字**  **char desc[1024]; // 景点介绍**  **};**  **定义Edge结构体，存储图的边的信息。**  **struct Edge**  **{ int vex1， vex2; //边的两个个顶点**  **int weight; //权值**  **};**  三、主要仪器设备及耗材   1. 安装了Windows XP或Windows 10或其它版本的Windows操作系统的PC机1台 2. PC机系统上安装了Microsoft Visual Studio开发环境   第二部分：实验过程和结果（可加页）   1. 源代码   Graph.cpp关键函数：  #include<iostream>  #include"Graph.h"  using namespace std;  //m\_Graph图结构已经在主函数中定义，此处调用其  extern Graph m\_Graph;  //初始化图结构  int Init()  {  for (int i = 0; i < 20; i++) {  for (int j = 0; j < 20; j++) {  m\_Graph.m\_aAdjMatrix[i][j] = 0; //邻接矩阵置零  }  m\_Graph.m\_nVexNum = 0; //景点数目置零  }  return OK;  }  //将顶点添加到数组中  int InsertVex(Vex sVex)  {  //顶点已满  if (m\_Graph.m\_nVexNum == 20)  return ERROR;  //  m\_Graph.m\_aVexs[m\_Graph.m\_nVexNum++] = sVex;  return OK;  }  //将边保存到邻接矩阵中  int InsertEdge(Edge sEdge)  {  //下标越界  if (sEdge.vex1<0 || sEdge.vex1 >= 20 || sEdge.vex2<0 || sEdge.vex2 >= 20)  return ERROR;  m\_Graph.m\_aAdjMatrix[sEdge.vex1][sEdge.vex2] = sEdge.weight;  m\_Graph.m\_aAdjMatrix[sEdge.vex2][sEdge.vex1] = sEdge.weight;  return OK;  }  //查询指定顶点信息  Vex GetVex(int nVex)  {  return m\_Graph.m\_aVexs[nVex];  }  //查询与指定顶点相连的边  int FindEdge(int nVex, Edge aEdge[])  {  int flag = 0; //与景点n相邻的边的条数  //便利整个图的邻接矩阵  for (int j = 0; j < 20; j++) {  if (m\_Graph.m\_aAdjMatrix[nVex][j] != 0 && nVex!=j) {  aEdge[flag].vex1 = nVex;  aEdge[flag].vex2 = j;  aEdge[flag].weight = m\_Graph.m\_aAdjMatrix[nVex][j];  flag++;  }  }  return flag;  }  //获取当前顶点数  int GetVexmun(){  return m\_Graph.m\_nVexNum;  }  //实现图的深度优先搜索遍历  void DFS(int nVex, bool bVisited[], int & nIndex, PathList & pList)  {  bVisited[nVex] = true; //改为已访问  pList->vexs[nIndex++] = nVex; //访问顶点nVex并赋值给链表，然后索引值自加  //判断所有的顶点是否都已经被访问过  int v\_num = 0;  for (int i = 0; i<m\_Graph.m\_nVexNum; i++)  {  //如果当前i节点被访问过，则V-Num自加  if (bVisited[i])  v\_num++;  }    //所有的顶点都已经被访问过,新增链表结点,保存此次的路径。必须保存，不然在后续的递归中会存在重复使用的vex，导致有的路径结点中vex没有值  if (v\_num == m\_Graph.m\_nVexNum)  {  //创建一个新链表，将当前的pList中的数据保存起来  pList->next = new Path;  for (int i = 0; i<m\_Graph.m\_nVexNum; i++)  {  pList->next->vexs[i] = pList->vexs[i];  }  pList = pList->next; //pList指针继续往下移动，寻找下一条路径  pList->next = NULL; //next赋值为空  }  //并没有全部访问，则进行寻找下一个相邻节点的操作  else  {  for (int i = 0; i<m\_Graph.m\_nVexNum; i++)  {  //如果i是nVex的的邻接点 并且未被访问  if (!bVisited[i] && m\_Graph.m\_aAdjMatrix[nVex][i]>0)  {  DFS(i, bVisited, nIndex, pList); //递归调用DFS  bVisited[i] = false; //改为未访问，回退  nIndex--; //索引值减一  }  }  }  }  //深度优先遍历  void DFSTraverse(int nVex, PathList & pList)  {  int nIndex = 0; //遍历深度  bool bVisited[20] = { false }; //所有的景点起始均为未访问  DFS(nVex, bVisited, nIndex, pList);  }  //寻找最短路径  int FindShortPath(int nVexStart, int nVexEnd, Edge aPath[])  {  int nShortPath[20][20]; //保存最短路径，其中行表示终点，列表示从起点到终点的最短路径的每一步  int nShortDistance[20]; //保存最短距离，保存从起点到任一顶点的最短距离  bool aVisited[20]; //判断某顶点是否已经加入到最短路径中  int v; //在下面的循环中，表示每一次找到的可以加入集合的顶点，即已经找到了从起点到该顶点的最短路径  //初始化工作  for (v = 0; v<m\_Graph.m\_nVexNum; v++)  {  aVisited[v] = false;  if (m\_Graph.m\_aAdjMatrix[nVexStart][v] != 0) {  //初始化该顶点到其他顶点的最短距离，默认为两顶点间的距离  nShortDistance[v] = m\_Graph.m\_aAdjMatrix[nVexStart][v];  }  else {  //如果顶点v和nVexStart不相连，则最短距离设置为最大值  nShortDistance[v] = 0x7FFFFFFF;  }  nShortPath[v][0] = nVexStart; //起始点为nVexStart  //初始化最短路径  for (int w = 1; w < m\_Graph.m\_nVexNum; w++) {  nShortPath[v][w] = -1;  }  }  //初始化，将nVexStart顶点加入到集合中  aVisited[nVexStart] = true;  int min; //暂存路径的最小值  for (int i = 1; i<m\_Graph.m\_nVexNum; i++)  {  min = 0x7FFFFFFF;  bool bAdd = false; //判断是否还有顶点可以加入集合  for (int w = 0; w<m\_Graph.m\_nVexNum; w++)  {  if (!aVisited[w] && nShortDistance[w]<min)  {  v = w; //w顶点距离nVexStart顶点最近  min = nShortDistance[w]; //w到nVexStart的最短距离为min  bAdd = true;  }  }  //若果没有顶点可以加入到集合，则跳出循环  if (!bAdd) break;  aVisited[v] = true; //将w顶点加入到集合  nShortPath[v][i] = v; //每次找到最短路径后，就相当于从源点出发到了终点，所以nShortPath[v][i]=v  for (int w = 0; w<m\_Graph.m\_nVexNum; w++)  {  //将w作为过渡顶点计算nVexStart通过w到每个顶点的距离  if (!aVisited[w] && (min + m\_Graph.m\_aAdjMatrix[v][w]<nShortDistance[w]) && (m\_Graph.m\_aAdjMatrix[v][w]>0))  {  //更新当前最短路径及距离  nShortDistance[w] = min + m\_Graph.m\_aAdjMatrix[v][w];  for (int i = 0; i < m\_Graph.m\_nVexNum; i++) {  //如果通过w达到顶点i的距离比较短，则将w的最短路径复制给i  nShortPath[w][i] = nShortPath[v][i];  }  }  }  }  int nIndex = 0;  int nVex1 = nVexStart;  //将最短路径保存为边的结构体数组  for (int i = 1; i<m\_Graph.m\_nVexNum; i++)  {  if (nShortPath[nVexEnd][i] != -1)  {  aPath[nIndex].vex1 = nVex1;  aPath[nIndex].vex2 = nShortPath[nVexEnd][i];  aPath[nIndex].weight = m\_Graph.m\_aAdjMatrix[nVex1][aPath[nIndex].vex2];  nVex1 = nShortPath[nVexEnd][i];  nIndex++;  }  }  return nIndex;  /\*  bool bVisited[20] = { false }; //初始化全部结点为未访问  int start = nVexStart,end= nVexStart; //将起点值赋值给中间起点start方便调用；终点同起点  int index = 0; //访问深度/路线长度  bVisited[start] = true; //起点设置为已访问  //循环，直到中间结果的终点与真正的终点nVexEnd重合，即寻找完毕  while (end != nVexEnd) {  int MaxWeight = 0x7fffffff; //定义一个初始最大路径  //寻找相邻接点 start->end 的最短路径  for (int j = 0; j < m\_Graph.m\_nVexNum; j++) {  //如果j->i直接相连，并且权值小于当前已知的最大权值，并且当前重点j未被访问，那么这两个点就更近一点  if (m\_Graph.m\_aAdjMatrix[start][j] > 0 && m\_Graph.m\_aAdjMatrix[start][j] < MaxWeight && !bVisited[j]) {  MaxWeight = m\_Graph.m\_aAdjMatrix[start][j];  end = j;  bVisited[j] = true;  if (end == nVexEnd) //如果此时的点已经是终点，那么久不需要继续进行  break;  }  }    //将与中间的起点start传值给aPath保存  aPath[index].vex1 = start;  aPath[index].vex2 = end;  aPath[index++].weight = MaxWeight;  //下一次循环的起点为这次的终点  start = end;  }  return index; //返回深度（即找到的路径个数）\*/  }  void FindMinTree(Edge aPath[])  {  bool aVisited[20] = { false }; //判断某顶点是否在最小生成树中，true表示已经添加到了最小生成树中  aVisited[0] = true; //从0号顶点开始，加入到集合中  int min;  int nVex1, nVex2;    for (int k = 0; k<m\_Graph.m\_nVexNum - 1; k++)  {  min = 0x7FFFFFFF;  for (int i = 0; i<m\_Graph.m\_nVexNum; i++)  {  //从集合中取一个顶点  if (aVisited[i])  {  for (int j = 0; j<m\_Graph.m\_nVexNum; j++)  {  //从不在集合中的顶点 中取出一个顶点  if (!aVisited[j])  {  if ((m\_Graph.m\_aAdjMatrix[i][j]<min) && (m\_Graph.m\_aAdjMatrix[i][j] != 0))  {  nVex1 = i;  nVex2 = j;  //找出最短边  min = m\_Graph.m\_aAdjMatrix[i][j];  }  }  }  }  }  //保存最短边的两个顶点  aPath[k].vex1 = nVex1;  aPath[k].vex2 = nVex2;  aPath[k].weight = m\_Graph.m\_aAdjMatrix[nVex1][nVex2];  //将两个顶点加入集合  aVisited[nVex1] = true;  aVisited[nVex2] = true;  }  }   1. 调试说明（调试手段、过程及结果分析）   实验结果截图如图：            第三部分：实验小结、收获与体会  在学期初，我觉得数据结构还是比较简单的，但可能由于之前c语言学习对指针掌握的不够熟练，导致在数据结构中接触到与指针有关的问题，例如线性表，堆栈，队列，二叉树等问题的时候，都会显得有些吃力。但是在不断学习数据结构的过程中我也不断加强了对指针的学习，现在我已经能够基本掌握指针的相关知识并且能够熟练运用了。这一学期的学习下来我发现想要学好数据结构有以下几点经验{虽然可能我的数据结构学的并不是很好}   1. **初步了解算法思想、原理**   想要弄清楚一个算法的实现，首先要知道这个算法的大致原理，这是最简单的一步，也是最基础的一步，只有明白算法想要干什么，才能弄清楚相应的代码段是为什么   1. **钻研课本代码段**   对于书上的算法代码，我们一定要仔细钻研每一步的具体含义和目的，在此基础上深入的了解算法的实现过程，而不是一味的四级硬背，不仅无聊，而且效率低下。   1. **查找各种算法资料**   例如排序算法，其实历史上有很多不同的排序算法，书上只列举出了一部分，我们通过查阅资料可以发现很多其他不同的排序算法，而且就算是同一个算法，也有很多不同的实现方法，这个过程是一个十分有趣的过程，同时也增长了自己的知识储备，我们可以根据已有的知识储备，从而稍加创新，对某个算法可以有自己不同的见解，从而写出一个“自己”的算法。这对于数据结构的学习是十分重要的   1. **坚持上级操作，用实践检验**   和所有计算机相关知识的学习一样，数据结构也是一项需要动手的课程，一味的学习书本知识，埋头拿笔演算，还不如在电脑上把代码敲进去自己亲自跑一遍，只有这样才能够最直接最深入的了解一个代码，这也是我这个学期也来最深刻的感受。只有多动手，才能找到写代码的感觉，才能将各种算法烂熟于心。   1. **勤于练习，寻找感觉**   算法是为了问题服务的，我们在掌握了书本上的算法以后，要去找一些综合性的题目来锻炼自己，这些问题通常融合了不同的知识点，例如同时蕴含了排序，二叉树，堆栈的相关知识，只有在解决问题的过程中，灵活运用所学知识，才能真正检验我们是否牢固掌握了书本上的内容。 | | | | | |