石 家 庄 铁 道 大 学

**实 验 报 告**

班 级：信1901– 4

姓 名：闫竞存

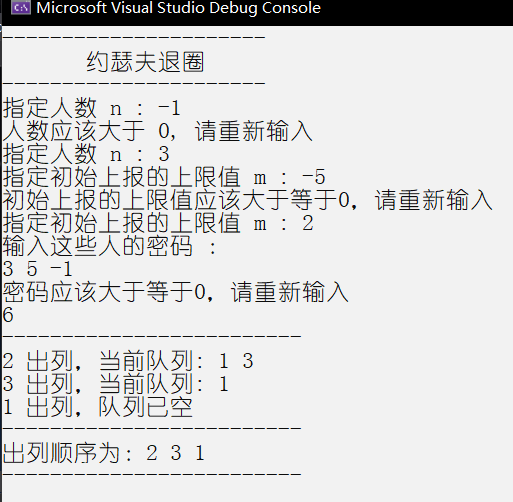
学 号：20194127

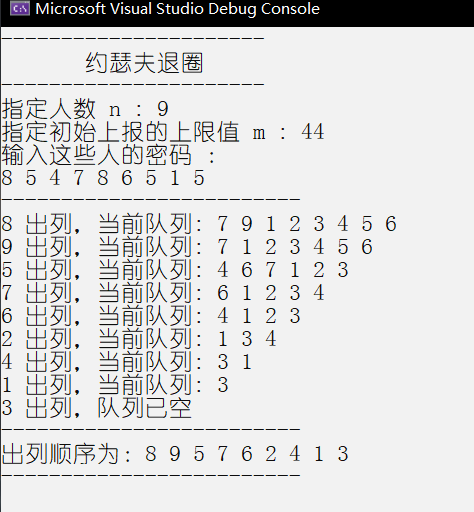
日 期：2020.12.18

**实习一 线性表应用—— 约瑟夫退圈**

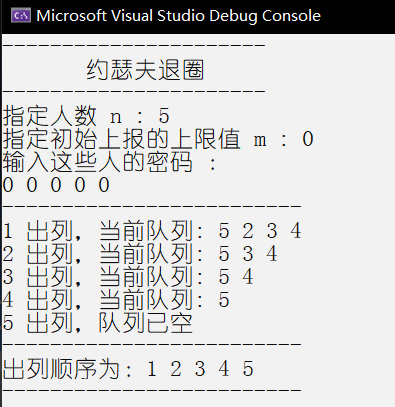
**1. 实验题目**

约瑟夫（Joeph）问题的一种描述是：编号为 1,2,…,n 的 n 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。一开始任选一个正整数作为报数上限值 m，从第一个人开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数。报 m 的人出列，将他的密码作为新的 m 值，从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直至所有人全部出列为止。试设计一个程序求出出列顺序。

**2. 程序测试**







**3. 核心算法的实现**

出队算法的实现

|  |
| --- |
| // 前面省略 …  int\* outOrder = new int[n]; // 记录出列顺序  int cntOutNumber = 0;  int totalNumber = n; //总人数  while (t && t->next != t) {  for (int i = 1; i < m; i++) t = t->next;  ListNode\* p = t->next;  if (p) {  m = p->data->pwd;  outOrder[cntOutNumber++] = p->data->ID; // 记录出列顺序  cout << p->data->ID << " 出列，当前队列: ";  t->next = p->next;  // 输出当前队列的值  int temp = --totalNumber;  ListNode\* pt = t;  while (temp--) {  cout << pt->data->ID << ' ';  pt = pt->next;  }  cout << endl;  delete p;  } else break;  }  if (t) cout << t->data->ID << " 出列" << "，队列已空" << endl;  outOrder[cntOutNumber++] = t->data->ID; //最后一个数字入队  //出列顺序  for (int i = 0; i < n; i++) cout << outOrder[i] << " ";  return 0;  } |

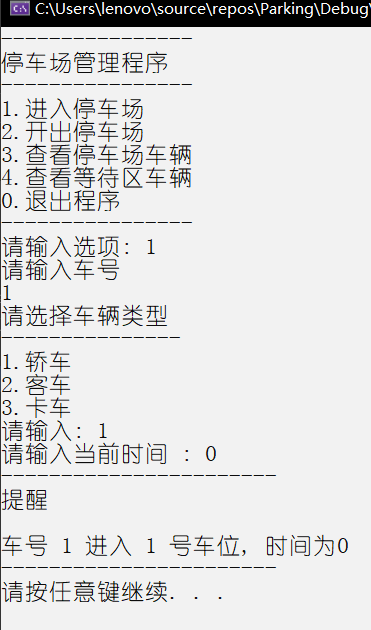
**实习二 栈和队列应用——停车场管理**

1. **实验题目**

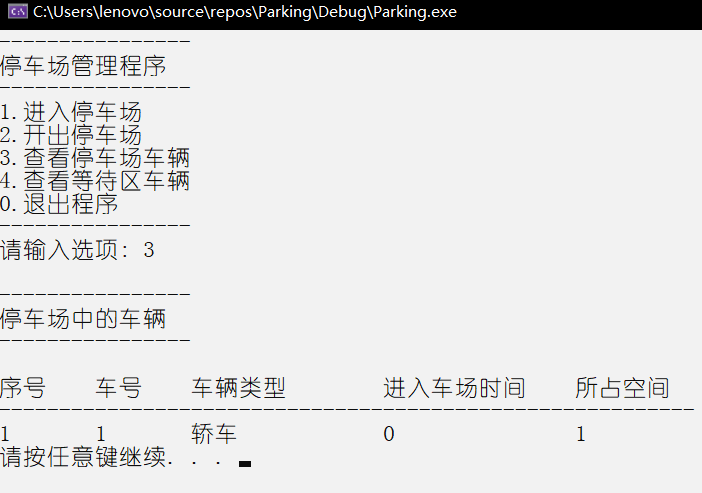
停车场程序，栈实现停车场，链表实现队列，当一辆车出停车场的时候，在这辆车前面的车应该先出去，等这辆车出去之后，让出去的车再进来。不同类型的车占地面积和收费标准不同。停在等候区的费用比再停车场中的费用低。

**2. 程序测试**

执行进停车场操作



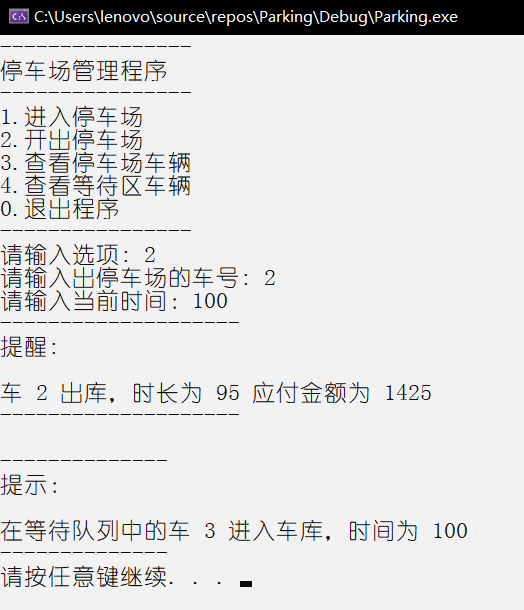
查看当前车辆



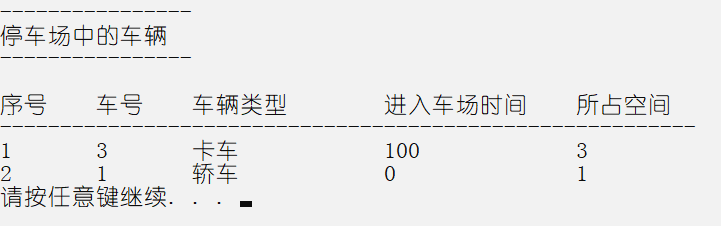


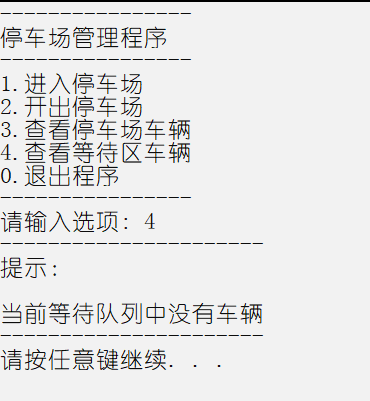


将停车场中的卡车开走，开走之后在等待队列中的车会进来

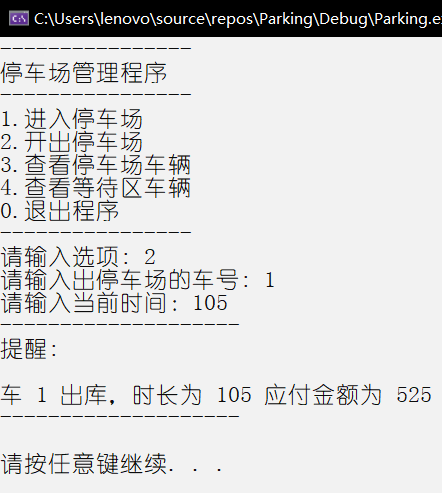
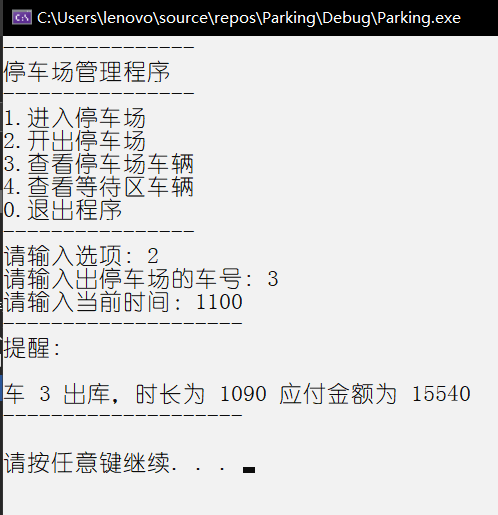


查看此时的停车场和等待队列的车辆

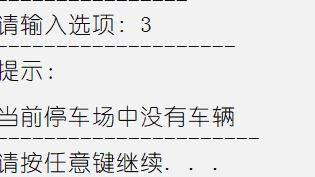
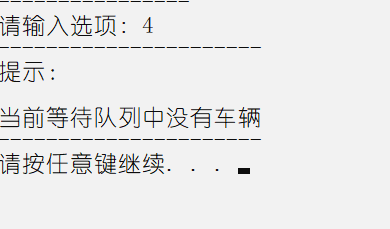




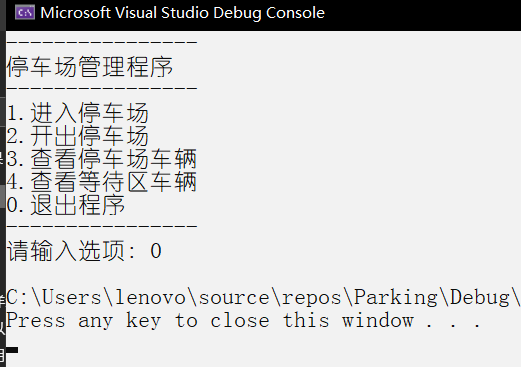
将剩余的车开走

查看此时停车场状态：

退出程序



**3. 核心算法的实现**

进停车场

|  |
| --- |
| void ParkingManager::enterPark() {  string id; int time; int type;  do { // 输入车号  cout << "请输入车号" << endl;  cin >> id;  if (this->currentCar.find(id) != currentCar.end()) {  // 车辆已经存在  cout << "--------------------\n提醒\n\n车辆已存在，请重新输入\n--------------------\n" << endl;  } else break;  } while (1);  this->currentCar.insert(id);  do { // 输入车辆类型  cout << "请选择车辆类型" << endl  << "---------------\n1.轿车\n2.客车\n3.卡车" << endl  cin >> type;  if (type < 1 || type > 3) {  cout << "----------------\n警告：\n\n请输入正确的数字\n----------------" << endl;  } else break;  } while (1);  do { // 输入当前时间  cout << "请输入当前时间 : "; cin >> time;  if (time < 0) {  cout << "----------------\n警告：\n\n请输入正确的时间\n----------------" << endl;  } else break;  } while (1);  if (this->size - OCCUPY[type] < 0) { // 检查车库有没有满  cout << "----------------------------\n提醒：\n\n车已停满，车辆进入等候区等候...\n----------------------------" << endl;  this->wait.push(new Car(time, id, type));  return;  }  this->parking.push\_0(new Car(id, type, time));  cout << "-----------------------\n提醒\n\n车号 " << id << " 进入 " << this->parking.size() << " 号车位, 时间为" << time << endl  << "-----------------------" << endl;  this->size -= OCCUPY[type];  } |

出停车场

|  |
| --- |
| void ParkingManager::outPark() {  string id; int time;  cout << "请输入出停车场的车号: "; cin >> id;  cout << "请输入当前时间："; cin >> time;  int flag = 0;  while (!this->parking.empty\_0()) {  if (this->parking.top\_0()->ID != id) {  //这辆车不是要出去的车  this->parking.push\_1(this->parking.top\_0());  this->parking.pop\_0();  } else {  // 这辆车是要出去的车  double toPay = OCCUPY[this->parking.top\_0()->type] \* (PARK\_PER\_TIME \* (time - this->parking.top\_0()->enTime) + WAIT\_PER\_TIME \* this->parking.top\_0()->enTime - this->parking.top\_0()->waitTime));  if (toPay < 0) {  cout << "--------------------\n警告:\n\n计算错误！出栈失败！\n--------------------" << endl; return; }  cout << "--------------------\n提醒: \n\n车 " << this->parking.top\_0()->ID << " 出库，时长为 " <<  (time - this->parking.top\_0()->enTime) +  (this->parking.top\_0()->enTime - this->parking.top\_0()->waitTime)  << " 应付金额为 " << toPay << endl << "--------------------\n" << endl;  Car\* todel = this->parking.top\_0();  this->parking.pop\_0();  //车站的容量加上刚刚出去的这辆车的容量  this->size += OCCUPY[todel->type];  //delete 刚出栈的车  delete todel; flag = 1; break;  }  }  if (flag) {  // 在车库中找到了这辆车，这辆车已经出去了的情况下  while (!this->parking.empty\_1()) {  // 将刚刚让路去的车弄进去  this->parking.push\_0(this->parking.top\_1());  this->parking.pop\_1();  }  if (!this->wait.empty()) { // 如果外面等待队列不空  // 如果车库中还有空间  if (this->size - OCCUPY[this->wait.front()->type] >= 0) {  this->size -= OCCUPY[this->wait.front()->type];  cout << "--------------\n提示：\n\n在等待队列中的车 " << this->wait.front()->ID << " 进入车库，时间为 " << time  << "\n--------------"<< endl;  this->wait.front()->enTime = time;  //设置等待队列最前面的车的入库时间  this->parking.push\_0(this->wait.front());  this->wait.pop();  }  }  } else { // 在车库中没有找到这辆车，从等候区寻找这辆车  flag = 0;  for (int i = 0; i < this->wait.size(); i++) {  if (this->wait.front()->ID == id) {// 找到了这辆车，  cout << "车号 " << this->wait.front()->ID  << " 出库，时长为 " << time - this->wait.front()->waitTime << " 应付金额为 " << OCCUPY[this->wait.front()->type] \* WAIT\_PER\_TIME \* (time - this->wait.front()->waitTime) << endl;  Car\* todel = this->wait.front();  this->currentCar.erase(todel->ID); // 在所有的车辆集合中删除这辆车  this->wait.pop();delete todel; flag = 1; break;;  } else {  // 当前这辆车不是要找的对象， 则将这辆车插入到队列尾部（就算以后找到了也是要插到队列的尾部的）  this->wait.push(this->wait.front());  this->wait.pop();  }  }  }  if (!flag) cout << "-----------\n提示: \n\n找不到这辆车\n------------" << endl;  } |

**实习三 二叉树应用——哈夫曼编译码**

1. **实验题目**

设计一个哈夫曼编码译码系统，实现对字符串的编码和解码

1. **程序测试**



1. **核心算法的实现**

构建哈夫曼树

|  |
| --- |
| void BiTreeApplication::init() {  cout << "----------------\n正在初始化二叉树\n----------------" << endl;  cout << "输入字符和其权值，中间以空格隔开，EOF 代表结束" << endl;  char tempChar; int tempWeight;  for (this->nodeCount = 1; cin >> tempChar >> tempWeight; nodeCount++) {  this->huffTree[nodeCount].data = tempChar;  this->huffTree[nodeCount].weight = tempWeight;  }  this->nodeCount--; //减去在循环的时候多出来的那个  this->maxLength = this->nodeCount \* 2 - 1;  cout << "----------------\n提示\n\n输入字母代码成功\n----------------" << endl;  this->constructHuffTree();  }  void BiTreeApplication::constructHuffTree() {  for (int i = ++this->nodeCount; i <= this->maxLength; i++) {  int s1 = 0, s2 = 0;  this->selectNode(s1, i - 1, s2);  this->huffTree[s1].parent = i;  this->huffTree[s2].parent = i;  this->huffTree[i].lch = s1;  this->huffTree[i].rch = s2;  this->huffTree[i].weight = this->huffTree[s1].weight + this->huffTree[s2].weight;  }  this->hasConstruct = 1;  } |

编码

|  |
| --- |
| void BiTreeApplication::encoding() {  if (!this->hasConstruct) {  cout << "---------------------------\n提示：\n\n二叉树尚未建立, 请执行选项2\n---------------------------" << endl;  return;  }  cout << "输入字符(不含空格): ";  string str; cin.clear(); cin >> str;  cout << "\n哈夫曼编码为 :" << endl;  for (int i = 0; i < str.length(); i++) {  // 遍历树，寻找数据节点  string tempCode = "";  int flag = 0; // 有没有找到这个字母所在的节点  for (int j = 1; j < this->nodeCount; j++) {  if (this->huffTree[j].data == str[i]) {  flag = 1; int tempNumber = j;  int parent = this->huffTree[j].parent;  while (parent != 0) {  // left child is the node  if (this->huffTree[parent].lch == tempNumber) tempCode += "0";  if (this->huffTree[parent].rch == tempNumber) tempCode += "1";  tempNumber = parent;  parent = this->huffTree[parent].parent;  }  break;  }  }  // 在树中没找到这个节点  if (!flag) {  cout << "\n\n警告：\n\n未找到节点" << str[i] << " 程序即将返回上一层\n" << endl; return;  }  // 反转10序列然后输出  reverse(tempCode.begin(), tempCode.end());  cout << tempCode;  }  } |

译码

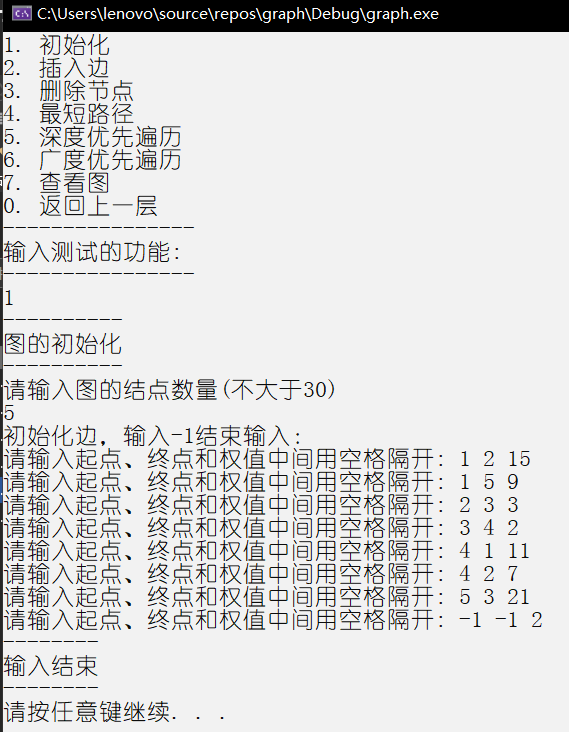
|  |
| --- |
| void BiTreeApplication::decoding() {  if (!this->hasConstruct) return;  cout << "\n输入Huffman code：";  string source;  cin >> source;  cout << "\n解码结果为 ：";  int ptr = this->maxLength;  for (int i = 0; i < source.length(); i++) {  if (this->huffTree[ptr].lch != 0 && this->huffTree[ptr].rch != 0) {  if (source[i] == '0') {  ptr = this->huffTree[ptr].lch;  } else if (source[i] == '1') {  ptr = this->huffTree[ptr].rch;  } else {  cout << "\n----------------------\n警告：\n\n遇到解码错误，即将返回\n----------------------"; return;  }  } else {  cout << this->huffTree[ptr].data;  ptr = this->maxLength;  i--; // 回退  }  }  } |

**实习四 图——图的基本操作**

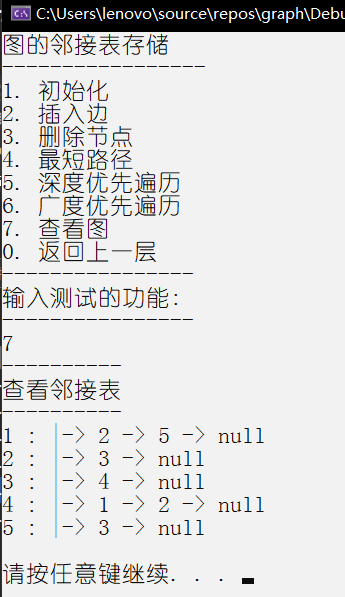
1. **实验题目**

分别用邻接矩阵和邻接表实现以下操作：图的创建、遍历、插入、删除、最短路径。

1. **程序测试**

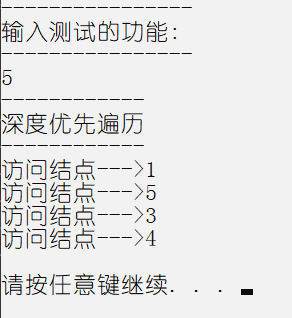


查看邻接表：





深度优先遍历：

1. **核心算法的实现**

最短路径-弗罗伊德算法

|  |
| --- |
| void ArrayMap::shortestPath() {  int dist[maxn][maxn] = {}; // 最短距离  // 初始化最短距离矩阵  for (int i = 0; i < this->size; i++) {  for (int j = 0; j < this->size; j++) {  if ((this->data[i][j] == 0 && i != j) || this->data[i][0] == -1) dist[i][j] = CANNOT\_REACH;  // data[i][0] == -1 的时候，看作这个点不存在  else dist[i][j] = this->data[i][j];  }  }  // 打表,即为所求  for (int k = 0; k < this->size; k++)  for (int i = 0; i < this->size; i++)  for (int j = 0; j < this->size; j++)  if (dist[i][j] > dist[i][k] + dist[k][j])  dist[i][j] = dist[i][k] + dist[k][j]; // 更新  // 输入点两点  cout << "输入两点, 判断其最短距离, 中间用空格隔开: ";  int v1 = 0, v2 = 0;  cin.clear();  // 输入检查  cin >> v1 >> v2;  while (v1 < 1 || v1 > 30 || v2 < 1 || v2 > 30) {  cout << "值不符合要求" << endl;  cin >> v1 >> v2;  }  if (v1 == v2) {  cout << "两点最短距离为 0 " << endl;  } else if (dist[v1 - 1][v2 - 1] == 0 || dist[v1 - 1][v2 - 1] == CANNOT\_REACH) {  cout << "两点不可达" << endl;  } else {  cout << "两点最短距离为 " << dist[v1 - 1][v2 - 1] << endl;  }  } |

深度优先遍历

|  |
| --- |
| void ArrayMap::DFSTraverse() {  // 已经遍历过的点  int\* visit = new int[this->size];  for (int i = 0; i < this->size; i++) {  if (this->data[i][0] == -1) visit[i] = 1;  else visit[i] = 0;  }  for (int i = 0; i < this->size; i++)  if (!visit[i]) this->DFS(i, visit);  }  void ArrayMap::DFS(int vtx, int visited[]) {  cout << "访问节点--->" << vtx + 1 << endl;  visited[vtx] = 1;  for (int i = 0; i < this->size; i++)  if (this->data[vtx][i] != 0 && !visited[i]) this ->DFS(i, visited);  } |

广度优先遍历

|  |
| --- |
| void ArrayMap::BFSTraverse() {  int\* visit = new int[this->size]; // 已经遍历过的点  for (int i = 0; i < this->size; i++) {  if (this->data[i][0] == -1) visit[i] = 1;  else visit[i] = 0;  }  queue<int> q; // 结点队列  for (int i = 0; i < this->size; i++) {  if (!visit[i]) {  visit[i] = 1;  cout << "访问结点--->" << i + 1 << endl;  q.push(i);  while (!q.empty()) {  i = q.front();  q.pop();  for (int j = 0; j < this->size; j++) {  if (this->data[i][j] != 0 && !visit[j]) {  visit[j] = 1;  cout << "访问结点--->" << j + 1 << endl;  q.push(j);  }  }  }  }  }  cout << endl;  } |

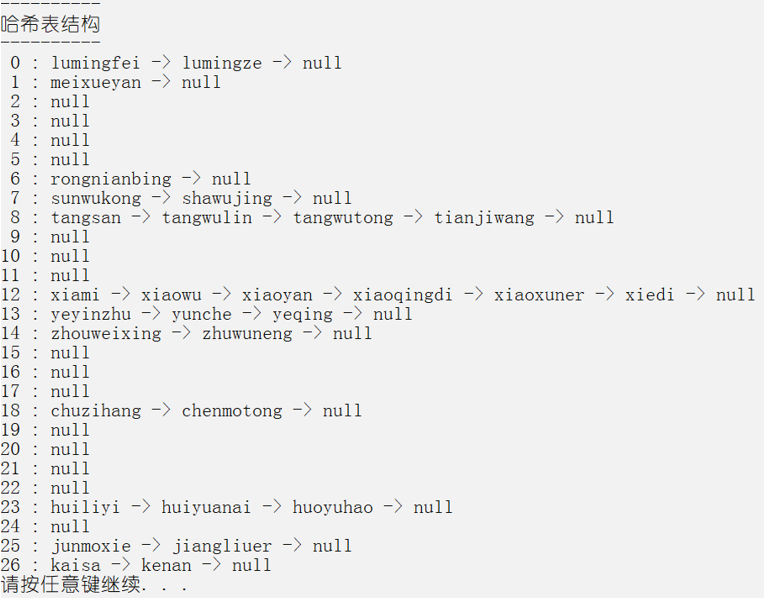
**实习五 哈希表——姓名查找哈希设计**

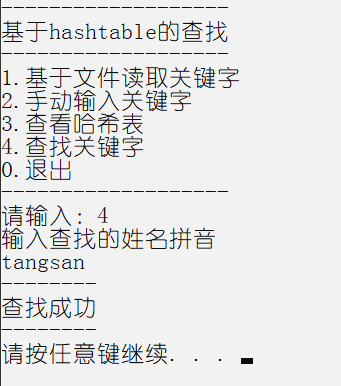
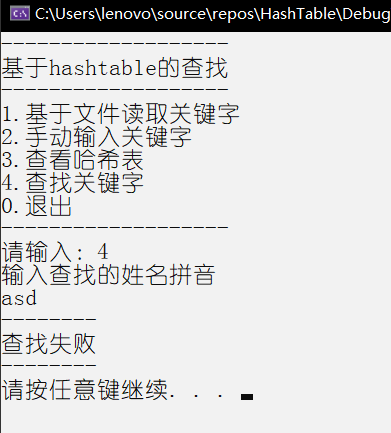
1. **实验题目**

为30个人名设计哈希表，哈希函数用除留余数法构造，用链地址法处理冲突，使平均查找长度不超过2.

1. **程序测试**

插入数据之后：



1. **核心算法的实现**

向哈希表中插入数据

|  |
| --- |
| // 在此之前，已经将姓名存到了string数组names中  void insert(ListNode\* hashtable, int posi, string data) {  // hashtable 是哈希表， posi是要插入的位置，data是要插入的数据  if (hashtable[posi].next == NULL) {  hashtable[posi].next = new ListNode(data, NULL);  } else {  ListNode\* temp = hashtable[posi].next;  while (temp->next) temp = temp->next;  temp->next = new ListNode(data, NULL);  }  } |

从哈希表中查找数据

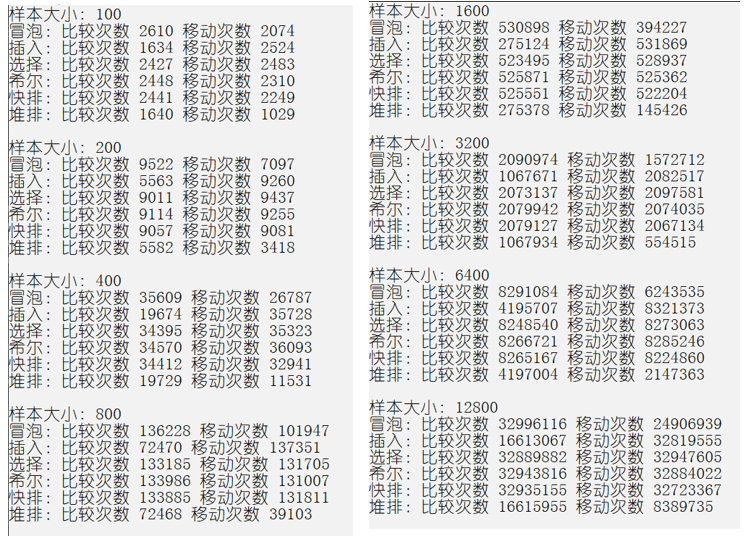
|  |
| --- |
| //在哈希表中查找指定字符串，成功返回1，否则返回0  int findInHashTable(string nameToFind) {  ListNode\* temp = nullptr;  if (!(temp = hashTable[hashCode(nameToFind)].next)) return 0;  while (temp) {  if (temp->data == nameToFind) {return 1;}  temp = temp->next;  }  return 0;  } |

**实习六 排序——排序算法对比分析**

1. **实验题目**

对以下常用的内部排序算法进行比较：起泡排序、直接插入排序、简单选择排序、快速排序、希尔排序、堆排序。

1. **程序测试**



1. **核心算法的实现**

快速排序算法实现：

|  |
| --- |
| void Sorts::quickSort(int\* nums, int first, int last) {  int low = first; int high = last; int key = nums[low];  if (low >= high) return;  while (low < high) {  while (this->cmpOfQuickSort++, low < high && nums[high] >= key) high--;  nums[low] = nums[high];  while (this->cmpOfQuickSort++, low < high && nums[low] <= key) low++;  nums[high] = nums[low];  this->movOfQuickSort += 2;  }  nums[low] = key;  this->movOfQuickSort++;  quickSort(nums, first, low - 1);  quickSort(nums, low + 1, last);  } |

希尔排序实现：

|  |
| --- |
| void Sorts::shellSort(int\* nums, int length) {  int gap; //gap是分组的步长  int temp; //希尔排序是在直接插入排序的基础上实现的,所以仍然需要哨兵  for (gap = length / 2; gap > 0; gap = gap / 2) {  for (int i = 0; i < gap; i++) {  for (int j = i + gap; j < length; j = j + gap) { //单独一次的插入排序  if (timeOfCompare++, nums[j] < nums[j - gap]) {  temp = nums[j]; //哨兵  int k = j - gap;  while (timeOfCompare++, k >= 0 && nums[k] > temp) {  nums[k + gap] = nums[k]; k = k - gap; timeOfMove++;  }  nums[k + gap] = temp;  }  }  }  }  } |

堆排序实现

|  |
| --- |
| void Sorts::heapAdjust(int\* arr, int start, int end) {  int dad = start; int son = dad \* 2 + 1;  while (son <= end) {  if (this->cmpOfHeapSort++, son + 1 <= end && arr[son] < arr[son + 1]) son++;  if (this->cmpOfHeapSort++, arr[dad] > arr[son]) return;  else {  this->movOfHeapSort += 3;  swap(arr[dad], arr[son]);  dad = son; son = dad \* 2 + 1;  }  }  }  void Sorts::heapSort(int\* nums, int n) {  for (int i = n / 2 - 1; i >= 0; i--) this->heapAdjust(nums, i, n - 1);  for (int i = n - 1; i > 0; i--) {  swap(nums[0], nums[i]);  this->movOfHeapSort += 3; // 交换看作3次移动  this->heapAdjust(nums, 0, i - 1);  }  } |