

**本科实验报告**

课程名称： 算法设计与分析

实验项目： 算法设计与分析实验

实验地点： 文德楼201

专业班级： 学号： 19124349

学生姓名： 王昆

指导教师： 喻钢

2021年5月 20日

|  |  |
| --- | --- |
| **实验名称** | 实验八 贪心法算法设计-活动安排问题和哈夫曼编码问题 |
| **实验目的** | |
| 1. 理解贪心法的核心处理机制，理解解决问题的贪心法算法设计与实现过程；  2. 掌握基于贪心法的活动安排问题和哈夫曼编码问题的算法设计与分析； | |
| **问题描述** | |
| 1. 设计基于贪心法的算法实现活动安排问题，测试数据如下。  1. 设计基于贪心法的算法实现哈夫曼编码问题，测试数据如下。 | |
| **实验环境** | |
| 程序设计语言：C++/C#  编程工具：microsoft visual studio 2010 | |
| **程序代码** | |
| 实验1  //创建结构体  void swap(int ar\_a[], int a, int b)  {  int temp = ar\_a[a];  ar\_a[a] = ar\_a[b];  ar\_a[b] = temp;  }  //冒泡排序  //输入数组，以及排序头尾  //输出排序后的数组  void bubble\_sort(int ar\_f[], int ar\_s[], int left, int right)  {  for (int i = left; i < right; i++){  for (int j = left + 1; j <= right; j++){  if (ar\_f[j] < ar\_f[j - 1]){  swap(ar\_s, j, j - 1);  swap(ar\_f, j, j - 1);  }  }  }  }  //贪婪算法解决活动安排问题  //输入数组，以及数组长度  //输出安排结果  void GreedySelect(int ar\_s[], int ar\_f[], bool ar\_a[], int lenght\_ar\_s)  {  int n = lenght\_ar\_s - 1;  ar\_a[1] = true;  int j = 1, count = 1;  cout << "1";  for (int i = 2; i <= n; i++){  if (ar\_s[i] >= ar\_f[j]){  ar\_a[i] = true;  j = i;  count++;  cout << i << endl;  }  else{  ar\_a[i] = false;  }  }  cout << "count=" << count << endl;  }  实验2  //哈夫曼树生成  //cha:当前输入字符，m:每种字符出现的次数,n:字符总个数  //输出字符频率，以及每个字符的哈夫曼编码  void huf(char cha[], int m[], int n)  {  int c, i, j, m1, m2, x1, x2, p;  hnodetype\* huffnode = new hnodetype[2 \* n - 1];  hcodetype\* huffcode = new hcodetype[n], cd;  //初始化哈夫曼树  for (i = 0; i < 2 \* n - 1; i++){  huffnode[i].flag = 0;  huffnode[i].lchild = -1;  huffnode[i].rchild = -1;  huffnode[i].parent = 0;  huffnode[i].weight = 0;  }  for (i = 0; i < n; i++){  huffnode[i].weight = m[i];  huffcode[i].leaf = cha[i];  }  //遍历寻找左右子树,用x1,x2记录  for (i = 0; i < n - 1; i++){  m1 = m2 = INT\_MAX;  x1 = x2 = 0;  for (j = 0; j < n + i; j++){  if (huffnode[j].weight <= m1 && huffnode[j].flag == 0){  x2 = x1;  x1 = j;  m2 = m1;  m1 = huffnode[j].weight;  }  else if (huffnode[j].weight <= m2 && huffnode[j].flag == 0){  x2 = j;  m2 = huffnode[j].weight;  }  }  huffnode[x1].parent = n + i;  huffnode[x2].parent = n + i;  huffnode[x1].flag = 1;  huffnode[x2].flag = 1;  huffnode[n + i].weight = huffnode[x1].weight + huffnode[x2].weight;  huffnode[n + i].lchild = x1;  huffnode[n + i].rchild = x2;  }  //生成哈夫曼树的编码  for (i = 0; i < n; i++){  cd.start = n - 1;  c = i;  p = huffnode[c].parent;  while (p != 0){  if (huffnode[p].lchild == c) cd.bit[cd.start] = 0;  else cd.bit[cd.start] = 1;  cd.start--;  c = p;  p = huffnode[c].parent;  }  cout << huffcode[i].leaf << ":";  for (j = cd.start + 1; j < n; j++){  huffcode[i].bit[j] = cd.bit[j];  cout << cd.bit[j];  }  cout << endl;  huffcode[i].start = cd.start;  }  delete[] huffcode;  delete[] huffnode;  } | |
| **运行结果（截屏）** | |
|  | |
| **分析体会(复杂性分析&程序的优劣&改进)** | |
| 实验1  *O( nlog2 n)*  假定各个活动已按结束时间从小到大排序，算法只需O(n)的时间安排n个活动。对于未排序的情况，排序过程将消耗主要的时间。为了在每一次贪心选择时快速查找具有最早时间的相容活动，先把n个活动按结束时间非减序排列。这样，贪心选择时取当前活动集合中结束时间最早的相容活动就归结为取当前活动集合中排在最前面的且与已安排活动相容的活动  实验2  *O(nlogn)*  最小堆的最小元素就是根节点。由于最小堆需要经常性的做抽取最小元素和插入操作，因此实际上为了维持堆的特征，每次插入和抽取都要进行节点的调整 | |