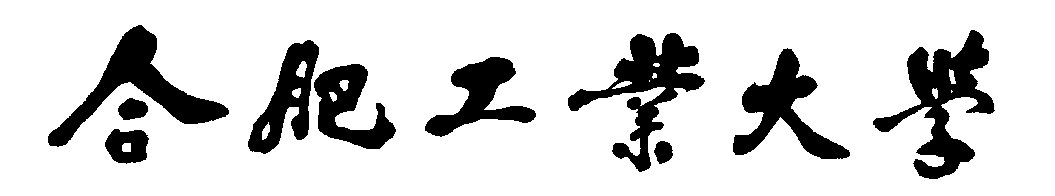
****

计算机与信息学院

数据结构实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 计算机科学与技术21-1 |
| 学生姓名及学号 | 杨程锦 2021214710 |
| 课程教学班号 |  |
| 任 课 教 师 | 胡学刚 |
| 实验指导教师 | 李培培 |
| 实验地点 | C201 |
| 2021 ~2022 学年第 二 学期 | |

说 明

实验报告是关于实验教学内容、过程及效果的记录和总结，因此，应注意以下事项和要求：

1．每个实验单元在4页的篇幅内完成一份报告。“实验单元”指按照实验指导书规定的实验内容。若篇幅不够，可另附纸。

2、各实验的**预习部分的内容是进入实验室做实验的必要条件**，请按要求做好预习。

3．实验报告要求：**书写工整规范，语言表达清楚，数据和程序真实。理论联系实际，认真分析实验中出现的问题与现象，总结经验。**

4．参加实验的每位同学应独立完成实验报告的撰写，其中程序或相关的设计图纸也可以采用打印等方式粘贴到报告中。严禁抄袭或拷贝，否则，一经查实，按作弊论取，并取消理论课考试资格。

5．实验报告作为评定实验成绩的依据。

实验序号及名称：实验 五 树与二叉树

实验时间∶ 年 月 日

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  **（1）掌握树与二叉树的动态链表存储结构及表示；**  **（2）掌握二叉树、线索二叉树、树与森林的遍历算法与相互转换；**  **（3）掌握基于树与二叉树的实际应用问题求解的算法设计** |
| 二、实验任务∶  **<1> 设计一个二叉树系统，除包含二叉树的相关运算功能外，还要实现以下功能：**  **（1）按中序次序输出二叉树中各结点的值及其所对应的层次数；**  **例如：构建如图所示的二叉树，输出结果：**  **D:3, B:2, E: 3, A: 1 F: 3, C:2 G:3**  **（2）实现三种次序的线索化算法；**  **（3）在线索二叉树中指定位置上插入一个用户自定义的结点；**  **要求：给出测试案例验证相关运算求解结果。**  **<2> 设计一个树和森林的小系统，包含以下功能，并可采用菜单方式来选择相应功能：**  **（1）采用多种方式建树或森林（指定输入、读入文件等）；**  **（2）实现各遍历算法；**  **（3）与二叉树的相互转如图2所示的森林，后序遍历森林的序列输出：**KEFBGCHIJDALNM    **<3>设计一个压缩与解压缩系统：**  **（1）通过读取文件获得文件中相关字符的权重；**  **（2）设计哈夫曼树实现对文件的压缩，并输出每个字符的哈夫曼编码 ；**  **（3）构建出相应的解压缩功能，以还原所压缩文件（选作\*）；** |
| 三、实验准备方案，包括以下内容：  （硬件类实验：实验原理、实验线路、设计方案等）  （软件类实验：所采用的核心方法、框架或流程图及程序清单）  任务一：二叉树  二叉树的构造：先序遍历二叉树，递归构造    二叉树的遍历：  递归遍历，和构造差不多  二叉树的线索化：  先序遍历二叉树，若结点左子树不为空，pre指向左子树，否则那么左子树应指向前驱pre。右子树为空，右子树指向后继。  二叉树的结点插入：  遍历线索二叉树，定位选取结点，    任务二：树和森林：  用孩子兄弟法储存结点，用fristchild替换左子树，nextbrother替换右子树。  任务三：哈夫曼树 |

|  |
| --- |
| 实验内容 |
| 一、实验用仪器、设备：  Legion R9000p，cLion |
| 二、实验内容与步骤（过程及数据记录）：  任务一：二叉树  //二叉树  #include<iostream>  using namespace std;  typedef struct Node  {  char data;  int depth;  Node\* left, \* right;  int LTag, RTag;  Node(char data)  {  this->data = data;  this->left = left;  this->right = right;  this->LTag = 1;  this->RTag = 1;  }  }Node;  class btree {  public:  btree();  void createBinaryTree(Node\*& cur);  void midorder() { midorder(root); };  void getdepth() { getdepth(root, 1); }  void Threading() { Threading(root); }  void ThreadingPrint() { ThreadingPrint(root); }  void midorder(Node\* cur);  void getdepth(Node\* p, int depth);  void ThreadingPrint(Node\* T);  void Threading(Node\* T);  void insert(char data1, char data2);  protected:  Node\* root;  };  btree::btree() {  createBinaryTree(root);  }  void btree::createBinaryTree(Node\*& cur)  {  char data;  cout << "input the root if you donot want continue the nod input enough '#'" << endl;  cin >> data;  if (cur == root) {  root = new Node(data);  createBinaryTree(root->left);  createBinaryTree(root->right);  }  else {  if (data == '#') {  cur = NULL;  return;  }  else  {  //按前序遍历输入构建二叉树  cur = new Node(data);  createBinaryTree(cur->left);  createBinaryTree(cur->right);  }  }  }  void btree::midorder(Node\* cur) {  if (cur != NULL) {  midorder(cur->left);  cout << cur->data << ":" << cur->depth << " ";  midorder(cur->right);  }  return;  }  void btree::getdepth(Node\* p, int depth)  {  if (!p)  return;  p->depth = depth;  //if (p->left == NULL && p->right == NULL)  // ans = max(ans, p->depth);  getdepth(p->left, depth + 1);  getdepth(p->right, depth + 1);  }  Node\* pre = NULL; //pre表示前驱节点，初始化为NULL  /////////////////////////////////////////////////////////////////////////////////////////////先序  void btree::Threading(Node\* T) {  if (T == NULL)  return; //当前节点是空  if (T->left == NULL) { //当前节点左为空，那么左孩子应指向前驱pre  T->left = pre;  T->LTag = 0;  }  if (pre != NULL && pre->right == NULL) { //pre结点右孩子为空，那么右孩子应指向后继节点，也就是当前节点T  pre->right = T;  pre->RTag = 0;  }  pre = T; //更新pre为当前节点  if (T->LTag == 1) Threading(T->left); //有左孩子递归线索化左子树  if (T->RTag == 1) Threading(T->right); //有右孩子递归线索化右子树  return;  }  ////根据前序线索二叉树输出前序序列  void btree::ThreadingPrint(Node\* T) {  while (T != NULL) { //当前节点不为空  cout << T->data << " "; //输出当前节点的内容  if (T->LTag == 1) T = T->left; //如果有左孩子，那么左孩子必为后继节点  else T = T->right; /\*如果没有左孩子，那么左孩子指向了当前节点的前驱。  如果右孩子存在，那么右孩子就是后继，  如果右孩子不存在，那么右孩子肯定指向后继节点，  除非是最后一个节点右孩子会指向NULL，  那么此时恰好跳出while循环，输出结束\*/  }  cout << endl;  return;  }  //////////////////////////////////////////////////////////////////////////////////////////中序  //void btree::Threading(Node\* T) {  // if (T == NULL)  // {  // return;  // }  // Threading(T->left); // 左  // if (T->left == NULL) { //当前节点左为空，那么左孩子应指向前驱pre  // T->left = pre;  // T->LTag = 0;  // }  // if (pre != NULL && pre->right == NULL) { //pre结点右孩子为空，那么右孩子应指向后继节点，也就是当前节点T  // pre->RTag = 0;  // pre->right = T;  // }  // Threading(T->right); //右  //}  /////////////////////////////////////////////////////////////////////////////////////////后序  //void btree::Threading(Node\* T) {  // if (T == NULL)  // {  // return;  // }  // Threading(T->left); // 左  // Threading(T->right); //右  // if (T->left == NULL) { //当前节点左为空，那么左孩子应指向前驱pre  // T->left = pre;  // T->LTag = 0;  // }  // if (pre != NULL && pre->right == NULL) { //pre结点右孩子为空，那么右孩子应指向后继节点，也就是当前节点T  // pre->RTag = 0;  // pre->right = T;  // }  //}  void btree::insert(char data1, char data2) {  Node\* T = root;  while (T->data != data1) {  if (T->LTag == 1)  T = T->left; //如果有左孩子，那么左孩子必为后继节点  else T = T->right;  }  Node\* p = T;  Node\* s;  s = new Node(data2);  s->left = p->left;  s->LTag = 1;  s->right = p;  s->RTag = 1;  p->left = s;  p->LTag = 1;  if (s->left->LTag == 1)  s->left->right = p;  }    实验结果正常  任务二：树和森林  #include<iostream>  #include<fstream>  #include<iostream>  #include"..\btree.h"  fstream q1;  using namespace std;  typedef struct node  {  char data;  node\* fristchild = NULL;  node\* nextbrother = NULL;  node(char data)  {  this->data = data;  }  };  class tree {  public:  tree();  void createtree(node\*& cur);  void order(node\* cur);  void order();  protected:  node\* fristroot;  };  tree::tree() {  createtree(fristroot);  }  void tree::createtree(node\*& cur)  {  char data;  cout << "input the root if you donot want continue the nod input enough '#'" << endl;  cin >> data;  /\*q1>>data;\*/  if (cur == fristroot) {  fristroot = new node(data);  createtree(fristroot->fristchild);  createtree(fristroot->nextbrother);  }  else {  if (data == '#') {  cur = NULL;  return;  }  else  {  //按前序遍历输入构建树  cur = new node(data);  createtree(cur->fristchild);  createtree(cur->nextbrother);  }  }  }  void tree::order(node\* cur) {  if (cur != NULL) {  order(cur->fristchild);  cout << cur->data << " ";  order(cur->nextbrother);  }  return;  }  void tree::order() {  order(fristroot);  }  //主函数    实验结果正常  任务三：哈夫曼树  #include<iostream>  #include<cstring>  using namespace std;  #pragma warning (disable:4996)  #define maxSize 100  /\*  赫夫曼树的存储结构，它也是一种二叉树结构，  这种存储结构既适合表示树，也适合表示森林。  \*/  typedef struct Node  {  int weight; //权值  int parent; //父节点的序号，为-1的是根节点  int lchild, rchild; //左右孩子节点的序号，为-1的是叶子节点  }HTNode, \* HuffmanTree; //用来存储赫夫曼树中的所有节点  typedef char\*\* HuffmanCode; //用来存储每个叶子节点的赫夫曼编码  HuffmanTree create\_HuffmanTree(int\* wet, int n);  void select\_minium(HuffmanTree HT, int k, int& min1, int& min2);  int min(HuffmanTree HT, int k);  void HuffmanCoding1(HuffmanTree HT, HuffmanCode& HC, int n);  void HuffmanCoding2(HuffmanTree HT, HuffmanCode& HC, int n);  int countWPL1(HuffmanTree HT, int n);  int countWPL2(HuffmanTree HT, int n);  int main()  {  int w[] = { 5,4,3,2,1 };//用数组w存储各个权值  int n = 5;//表示数组w中的个数  HuffmanCode HC = NULL;  HuffmanTree hTree = create\_HuffmanTree(w, n);  int wpl1 = countWPL1(hTree, n);  printf("从叶子结点开始遍历二叉树求最小带权路径长度WPL=%d\n", wpl1);  int wpl2 = countWPL2(hTree, n);  printf("从根结点开始遍历二叉树求最小带权路径长度WPL=%d\n", wpl2);  printf("\n从叶子到根结点编码结果为：\n");  HuffmanCoding1(hTree, HC, n);  printf("\n从根结点到叶子结点编码结果为：\n");  HuffmanCoding2(hTree, HC, n);  system("pause");  return 0;  }  /\*根据给定的n个权值构造一棵赫夫曼树,wet中存放n个权值\*/  HuffmanTree create\_HuffmanTree(int\* wet, int n)  {  //一棵有n个叶子节点的赫夫曼树共有2n-1个节点  int total = 2 \* n - 1;  HuffmanTree HT = (HuffmanTree)malloc(total \* sizeof(HTNode));  if (!HT)  {  printf("HuffmanTree malloc faild!");  exit(-1);  }  int i;  //以下初始化序号全部用-1表示，  //这样在编码函数中进行循环判断parent或lchild或rchild的序号时，  //不会与HT数组中的任何一个下标混淆  //HT[0],HT[1]...HT[n-1]中存放需要编码的n个叶子节点  for (i = 0; i < n; i++)  {  HT[i].parent = -1;  HT[i].lchild = -1;  HT[i].rchild = -1;  HT[i].weight = \*wet;  wet++;  }  //HT[n],HT[n+1]...HT[2n-2]中存放的是中间构造出的每棵二叉树的根节点  for (; i < total; i++)  {  HT[i].parent = -1;  HT[i].lchild = -1;  HT[i].rchild = -1;  HT[i].weight = 0;  }  int min1, min2; //用来保存每一轮选出的两个weight最小且parent为0的节点  //每一轮比较后选择出min1和min2构成一课二叉树,最后构成一棵赫夫曼树  for (i = n; i < total; i++)  {  select\_minium(HT, i, min1, min2);  HT[min1].parent = i;  HT[min2].parent = i;  //这里左孩子和右孩子可以反过来，构成的也是一棵赫夫曼树，只是所得的编码不同  HT[i].lchild = min1;  HT[i].rchild = min2;  HT[i].weight = HT[min1].weight + HT[min2].weight;  }  return HT;  }  /\*  从HT数组的前k个元素中选出weight最小且parent为-1的两个，分别将其序号保存在min1和min2中  \*/  void select\_minium(HuffmanTree HT, int k, int& min1, int& min2)  {  min1 = min(HT, k);  min2 = min(HT, k);  }  /\*  从HT数组的前k个元素中选出weight最小且parent为-1的元素，并将该元素的序号返回  \*/  int min(HuffmanTree HT, int k)  {  int i = 0;  int min; //用来存放weight最小且parent为-1的元素的序号  int min\_weight; //用来存放weight最小且parent为-1的元素的weight值  //先将第一个parent为-1的元素的weight值赋给min\_weight,留作以后比较用。  //注意，这里不能按照一般的做法，先直接将HT[0].weight赋给min\_weight，  //因为如果HT[0].weight的值比较小，那么在第一次构造二叉树时就会被选走，  //而后续的每一轮选择最小权值构造二叉树的比较还是先用HT[0].weight的值来进行判断，  //这样又会再次将其选走，从而产生逻辑上的错误。  while (HT[i].parent != -1)  i++;  min\_weight = HT[i].weight;  min = i;  //选出weight最小且parent为-1的元素，并将其序号赋给min  for (; i < k; i++)  {  if (HT[i].weight < min\_weight && HT[i].parent == -1)  {  min\_weight = HT[i].weight;  min = i;  }  }  //选出weight最小的元素后，将其parent置1，使得下一次比较时将其排除在外。  HT[min].parent = 1;  return min;  }  /\*  从叶子节点到根节点逆向求赫夫曼树HT中n个叶子节点的赫夫曼编码，并保存在HC中  \*/  void HuffmanCoding1(HuffmanTree HT, HuffmanCode& HC, int n)  {  //用来保存指向每个赫夫曼编码串的指针  HC = (HuffmanCode)malloc(n \* sizeof(char\*));  if (!HC)  {  printf("HuffmanCode malloc faild!");  exit(-1);  }  //临时空间，用来保存每次求得的赫夫曼编码串  //对于有n个叶子节点的赫夫曼树，各叶子节点的编码长度最长不超过n-1  //外加一个'\0'结束符，因此分配的数组长度最长为n即可  char\* code = (char\*)malloc(n \* sizeof(char));  if (!code)  {  printf("code malloc faild!");  exit(-1);  }  code[n - 1] = '\0'; //编码结束符，亦是字符数组的结束标志  //求每个字符的赫夫曼编码  int i;  for (i = 0; i < n; i++)  {  int current = i; //定义当前访问的节点  int father = HT[i].parent; //当前节点的父节点  int start = n - 1; //每次编码的位置，初始为编码结束符的位置  //从叶子节点遍历赫夫曼树直到根节点  while (father != -1)  {  if (HT[father].lchild == current) //如果是左孩子，则编码为0  code[--start] = '0';  else //如果是右孩子，则编码为1  code[--start] = '1';  current = father;  father = HT[father].parent;  }  //为第i个字符的编码串分配存储空间  HC[i] = (char\*)malloc((n - start) \* sizeof(char));  if (!HC[i])  {  printf("HC[i] malloc faild!");  exit(-1);  }  //将编码串从code复制到HC  strcpy(HC[i], code + start);  }  for (int i = 0; i < n; ++i) {  printf("%s\n", HC[i]);  }  free(code); //释放保存编码串的临时空间  }  /\*  从根节点到叶子节点无栈非递归遍历赫夫曼树HT，求其中n个叶子节点的赫夫曼编码，并保存在HC中  \*/  void HuffmanCoding2(HuffmanTree HT, HuffmanCode& HC, int n)  {  //用来保存指向每个赫夫曼编码串的指针  HC = (HuffmanCode)malloc(n \* sizeof(char\*));  if (!HC)  {  printf("HuffmanCode malloc faild!");  exit(-1);  }  //临时空间，用来保存每次求得的赫夫曼编码串  //对于有n个叶子节点的赫夫曼树，各叶子节点的编码长度最长不超过n-1  //外加一个'\0'结束符，因此分配的数组长度最长为n即可  char\* code = (char\*)malloc(n \* sizeof(char));  if (!code)  {  printf("code malloc faild!");  exit(-1);  }  int cur = 2 \* n - 2; //当前遍历到的节点的序号，初始时为根节点序号  int code\_len = 0; //定义编码的长度  //构建好赫夫曼树后，把weight用来当做遍历树时每个节点的状态标志  //weight=0表明当前节点的左右孩子都还没有被遍历  //weight=1表示当前节点的左孩子已经被遍历过，右孩子尚未被遍历  //weight=2表示当前节点的左右孩子均被遍历过  int i;  for (i = 0; i < cur + 1; i++)  {  HT[i].weight = 0;  }  //从根节点开始遍历，最后回到根节点结束  //当cur为根节点的parent时，退出循环  while (cur != -1)  {  //左右孩子均未被遍历，先向左遍历  if (HT[cur].weight == 0)  {  HT[cur].weight = 1; //表明其左孩子已经被遍历过了  if (HT[cur].lchild != -1)  { //如果当前节点不是叶子节点，则记下编码，并继续向左遍历  code[code\_len++] = '0';  cur = HT[cur].lchild;  }  else  { //如果当前节点是叶子节点，则终止编码，并将其保存起来  code[code\_len] = '\0';  HC[cur] = (char\*)malloc((code\_len + 1) \* sizeof(char));  if (!HC[cur])  {  printf("HC[cur] malloc faild!");  exit(-1);  }  strcpy(HC[cur], code); //复制编码串  }  }  //左孩子已被遍历，开始向右遍历右孩子  else if (HT[cur].weight == 1)  {  HT[cur].weight = 2; //表明其左右孩子均被遍历过了  if (HT[cur].rchild != -1)  { //如果当前节点不是叶子节点，则记下编码，并继续向右遍历  code[code\_len++] = '1';  cur = HT[cur].rchild;  }  }  //左右孩子均已被遍历，退回到父节点，同时编码长度减1  else  {  HT[cur].weight = 0;  cur = HT[cur].parent;  --code\_len;  }  }  for (int i = 0; i < n; ++i) {  printf("%s\n", HC[i]);  }  free(code);  }  /\*  从叶子结点开始遍历二叉树直到根结点，根结点为HT[2n-1],且HT[2n-1].parent=-1;  各叶子结点为HT[0]、HT[1]...HT[n-1]。  关键步骤是求出各个叶子结点的路径长度，用此路径长度\*此结点的权值就是  此结点带权路径长度，最后将各个叶子结点的带权路径长度加起来即可。  \*/  int countWPL1(HuffmanTree HT, int n)  {  int i, countRoads, WPL = 0;  /\*  由creat\_huffmanTree（）函数可知，HT[0]、HT[1]...HT[n-1]存放的就是各个叶子结点，  所以挨个求叶子结点的带权路径长度即可  \*/  for (i = 0; i < n; i++)  {  int father = HT[i].parent; //当前节点的父节点  countRoads = 0;//置当前路径长度为0  //从叶子节点遍历赫夫曼树直到根节点  while (father != -1)  {  countRoads++;  father = HT[father].parent;  }  WPL += countRoads \* HT[i].weight;  }  return WPL;  }  /\*  以下是从根结点开始遍历二叉树，求最小带权路径长度。关键步骤是求出各个叶子  结点的路径长度，用此路径长度\*此结点的权值就是此结点带权路径长度，最后将  各个叶子结点的带权路径长度加起来即可。  \*/  int countWPL2(HuffmanTree HT, int n)  {  int cur = 2 \* n - 2; //当前遍历到的节点的序号，初始时为根节点序号  int countRoads = 0, WPL = 0;//countRoads保存叶子结点的路径长度  //构建好赫夫曼树后，把visit[]用来当做遍历树时每个节点的状态标志  //visit[cur]=0表明当前节点的左右孩子都还没有被遍历  //visit[cur]=1表示当前节点的左孩子已经被遍历过，右孩子尚未被遍历  //visit[cur]=2表示当前节点的左右孩子均被遍历过  int visit[maxSize] = { 0 };//visit[]是标注数组,初始化为0  //从根节点开始遍历，最后回到根节点结束  //当cur为根节点的parent时，退出循环  while (cur != -1)  {  //左右孩子均未被遍历，先向左遍历  if (visit[cur] == 0)  {  visit[cur] = 1; //表明其左孩子已经被遍历过了  if (HT[cur].lchild != -1)  { //如果当前节点不是叶子节点，则路径长度+1，并继续向左遍历  countRoads++;  cur = HT[cur].lchild;  }  else  { //如果当前节点是叶子节点，则计算此结点的带权路径长度，并将其保存起来  WPL += countRoads \* HT[cur].weight;  }  }  //左孩子已被遍历，开始向右遍历右孩子  else if (visit[cur] == 1)  {  visit[cur] = 2;  if (HT[cur].rchild != -1)  { //如果当前节点不是叶子节点，则记下编码，并继续向右遍历  countRoads++;  cur = HT[cur].rchild;  }  }  //左右孩子均已被遍历，退回到父节点，同时路径长度-1  else  {  visit[cur] = 0;  cur = HT[cur].parent;  --countRoads;  }  }  return WPL;  }    实验结果正常 |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶  实验结果正常，成功构造了二叉树、树和森林、哈夫曼树，实现了二叉树的线索化、插入结点。 |
| 四、感想、体会、建议∶  这个实验任务极多，花了很多时间，二叉树的构造上刚开始花了些时间，到森林时对递归就已经很熟悉了，二叉树的线索化查阅了一些资料，刚开始不是很理解具体的操作。 |
| 实验成绩∶  指导教师签名：  年 月 日 |