湖南科技大学计算机科学与工程学院

数 据 结 构 实验报告

**专业班级：** 22级计算机7班

**姓 名：** 周俊哲

**学 号：** 2205010711

**指导教师：** 李锋

**时 间**： 2023.3~2023.6

**地 点**： 逸夫楼401

|  |
| --- |
| 指导教师评语：    **成绩： 等级：**  **签名：**  **年 月 日** |

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 | 1、线性表的基本操作与应用（第5、6题）  2、栈和队列的基本操作与应用（第3、10题）  3、  4、  。。。 | | |
| 实验性质  （必修、选修） | 必修 | 实验类型（验证、设计、创新、综合） | 验证、综合 |
| 实验课时 |  | 实验日期 | 1、2023年3月15日  2、2023年3月22日  3、  4、  ….. |
| 实验仪器设备以及实验软硬件要求 |  | | |
| 实验目的 | 1、掌握线性表的顺序存储和链式存储的基本操作的实现；掌握将书上算法转换为可运行程序的基本流程。  2、掌握栈和队列基本操作的实现；初步掌握使用栈或队列解决实际问题的方法。 | | |
| 实验内容（实验原理、运用的理论知识、算法、程序、步骤和方法）  实验一 线性表的基本操作与应用  合成有序链表  算法描述:  由于要将两个有序链表合并为一个有序链表，可以使用双指针的方法遍历两个链表，并依次比较节点大小，将小的节点添加到合并后的新链表中。考虑到要使用原来两个链表的存储空间，因此在遍历时需要不断修改节点的next指针来构建新链表。  代码实现:”  void MergeList\_L(LinkList& LA, LinkList& LB) {  LinkList pa = LA->next;  LinkList pb = LB->next;  LinkList pc = LA;  while (pa && pb)//pa和pb都不为空  {  if (pa->data < pb->data)//pa的数据小于pb  {  pc->next = pa;  pc = pa;//将pa接入pc链表中  pa = pa->next;  }  else if (pb->data < pa->data)//pb的数据小于pa  {  pc->next = pb;  pc = pb;//pb接入pc链表中  pb = pb->next;  }  else//pa与pb数据相等  {  pc->next = pa;  pc = pa;  pa = pa->next;  LinkList p = pb;  pb = pb->next;  delete p;//将pc指向pa或pb再删除另一个  }  }  if (pa)//循环结束pa不指向空  {  pc->next = pa;  }  else//循环结束pb不指向空  {  pc->next = pb;  }  delete LB;//销毁无用的头结点  }  两个稀疏多项式相加  算法描述:  可以采用双指针的方法遍历两个链表，依次比较节点的指数大小，并将对应系数相加，构建新的结果链表。  代码实现:  void AddPolyn(Polynomial& Pa, Polynomial& Pb)  {  PNode\* p1, \* p2, \* p3, \* r;  p1 = Pa->next;  p2 = Pb->next;  p3 = Pa;  float sum;  while (p1 && p2) {  if (p1->expn == p2->expn) { //指数相等  sum = p1->coef + p2->coef; // 系数求和  if (sum != 0) { //判断系数和是否为0  p1->coef = sum; //修改pa当前指向节点 的系数值为两项系数之和  p3->next = p1; //将修改后的 pa 当前指向结点 链接在 p3 之后 ，p3指向p1  p3 = p1;  p1 = p1->next; //p1后移  r = p2; p2 = p2->next; delete r; //删除p2当前节点  }  else { //此处系数为 0 ，则p1 p2都后移  r = p1; p1 = p1->next; delete r;  r = p2; p2 = p2->next; delete r;  }  }  else if (p1->expn < p2->expn) {  p3->next = p1;  p3 = p1;  p1 = p1->next;  }  else {  p3->next = p2;  p3 = p2;  p2 = p2->next;  }  }  p3->next = p1 ? p1 : p2;//利用三元运算符合成  delete Pb;  }  ……….  实验二 栈和队列的基本操作与应用  实验:表达式求值  算法描述:  1.如果当前运算符优先级高于栈顶元素，则将当前运算符入栈； 2.如果当前运算符优先级低于或等于栈顶元素，则从运算符栈中弹出栈顶元素，并从操作数栈中弹出两个操作数，进行运算并将结果压入操作数栈中，直到当前运算符的优先级高于栈顶元素； 3.如果当前运算符是左括号，则直接将其入运算符栈； 4.如果当前运算符是右括号，则从运算符栈中弹出栈顶元素，直到遇到左括号为止。 5.最后，当表达式读入结束符“#”并且运算符栈中只剩下结束符“#”时，操作数栈中的栈顶元素即为表达式的值。  代码实现:  // 定义运算符的优先级  int priority(char c) {  switch (c) {  case '+':  case '-':  return 1;  case '\*':  case '/':  return 2;  case '(':  return 0;  }  return -1; // 对于#，返回-1  }  int calculate(int a, int b, char op) {  switch (op) {  case '+':  return a + b;  case '-':  return a - b;  case '\*':  return a \* b;  case '/':  return a / b;  }  return 0; // 对于不合法的运算符，返回0  }  int main() {  string line;  while (getline(cin, line) && line != "#") {  stack<int> nums; // 存放数值的栈  stack<char> ops; // 存放运算符的栈  for (int i = 0; i < line.length(); i++) {  char c = line[i];  if (isdigit(c)) { // 如果是数字，将其转化为整数并入栈  int num = c - '0';  while (i + 1 < line.length() && isdigit(line[i + 1])) { // 将连续的数字合并为一个数  num = 10 \* num + (line[i + 1] - '0');  i++;  }  nums.push(num);  }  else if (c == '(') {  ops.push(c);  }  else if (c == ')') { // 遇到右括号，将括号内的表达式计算  while (ops.top() != '(') {  char op = ops.top();  ops.pop();  int b = nums.top(); nums.pop();  int a = nums.top(); nums.pop();  nums.push(calculate(a, b, op));  }  ops.pop(); // 弹出左括号  }  else if (c == '+' || c == '-' || c == '\*' || c == '/') {  while (!ops.empty() && priority(ops.top()) >= priority(c)) {//栈顶优先级高则进行运算  char op = ops.top();  ops.pop();  int b = nums.top(); nums.pop();  int a = nums.top(); nums.pop();  nums.push(calculate(a, b, op));  }  ops.push(c);//栈顶优先级低则压入栈中  }  }  while (!ops.empty()) { // 遍历完后，将剩余的运算符和数值计算，得到最终结果  char op = ops.top();  ops.pop();  int b = nums.top(); nums.pop();  int a = nums.top(); nums.pop();  nums.push(calculate(a, b, op));  }  cout << nums.top() << endl; // 最后栈中剩下的唯一数值就是表达式的结果  }  return 0;  }  舞伴问题  算法描述:   1. 读入学生总数n和每个学生的姓名和性别，存储到两个不同的队列中。 2. 只要男生和女生队列都不为空，就依次取出队首元素进行匹配，并输出舞伴的姓名。 3. 如果男生或女生队列为空，说明有一方已经配对完成，此时需要单独输出另一方队伍剩余未匹配的人员姓名。   代码实现:  typedef struct {  string name;  string sex;  }Person;  typedef struct {  Person\* base;  int front;  int rear;  }Queue;  //初始化队列  void InitQueue(Queue& Q) {  Q.base = new Person[MAXSIZE];  if (!Q.base) {  return;//分配错误退出  }  Q.front = Q.rear = 0;  }  //入队  void EnQueue(Queue& Q, Person e) {  if ((Q.rear + 1) % MAXSIZE == Q.front) {  return;//队满返回  }  Q.base[Q.rear] = e;  Q.rear = (Q.rear + 1) % MAXSIZE;  }  //出队  void DeQueue(Queue& Q, Person& e) {  if (Q.front == Q.rear) {  return;//队空返回  }  e = Q.base[Q.front];  Q.front = (Q.front + 1) % MAXSIZE;  }  //取队头元素  Person GetTop(Queue Q) {  if (Q.front != Q.rear) {  return Q.base[Q.front];  }  }  //判断队列是否为空  int Epmty(Queue Q) {  if (Q.rear == Q.front) {  return 1;  }  else return 0;  }  int main() {  int i,n;  Queue M;//男性队列  Queue F;//女性队列  InitQueue(M);  InitQueue(F);  cin >> n;  for (i = 0; i < n; i++) {  Person e;  cin >> e.name >> e.sex;  if (e.sex == "M") {  EnQueue(M, e);// 如果是男生，加入男生队列  }  else {// 如果是女生，加入女生队列  EnQueue(F, e);  }  }  while (!Epmty(M)&&!Epmty(F))//两队都不为空  {  Person e1, e2;  DeQueue(M, e1), DeQueue(F, e2);  cout << e2.name << " & " << e1.name << endl;  }  if (!Epmty(M)) {// 如果男队还有剩余队员  Person e;  DeQueue(M, e); // 出队男队首元素  cout << e.name << endl;// 输出男队首元素姓名  }  if (!Epmty(F)) {// 如果女队还有剩余队员  Person e;  DeQueue(F, e); // 出队女队首元素  cout << e.name << endl;// 输出女队首元素姓名  }  return 0;  }  ……….  实验三 树和二叉树的定义与遍历  实验:二叉树遍历（用结构体数组存储）  二叉树遍历是指按照某种规则依次访问二叉树中的每个结点，使得每个结点都被访问一次且仅访问一次。常见的二叉树遍历方式有前序遍历、中序遍历和后序遍历。  二叉树遍历算法的实现有多种方法，其中一种常用的方法是用结构体数组存储二叉树。具体步骤如下：  1.定义结构体类型  首先需要定义一个结构体类型，用于存储二叉树的结点信息。结构体中需要包含三个成员变量，分别是结点的值、左子树指针和右子树指针。  struct tree  {  char data;  int prt, lch, rch;  };  2.初始化结构体数组  接下来需要初始化结构体数组，即将二叉树的结点信息存储到数组中。  3.实现遍历算法  有了结构体数组之后，就可以实现二叉树的遍历算法了。以前序遍历为例，其实现过程如下：  void q\_order(int x)//前序遍历  {  if (x == 0)  return;  else  {  cout << a[x].data;  q\_order(a[x].lch);  q\_order(a[x].rch);  }  }  void z\_order(int x)//中序遍历  {  if (x == 0)  return;  else  {  z\_order(a[x].lch);  cout << a[x].data;  z\_order(a[x].rch);  }  }  void h\_order(int x)//后序遍历  {  if (x == 0)  return;  else  {  h\_order(a[x].lch);  h\_order(a[x].rch);  cout << a[x].data;  }  }  实验四 哈夫曼树与哈夫曼编码  实验: huffman编码（1）  实验原理：  哈夫曼编码是一种基于权值的前缀编码方式，用于将字符转换为二进制编码。赫夫曼编码的核心思想是将出现频率较高的字符用较短的编码表示，出现频率较低的字符用较长的编码表示，从而实现对文本的高效压缩。  哈夫曼编码的实现需要构造哈夫曼树，哈夫曼树是一种满足以下两个条件的二叉树：  1.每个叶子结点都对应一个字符，且该结点的权值等于该字符出现的频率。  2.非叶子结点的权值等于其左右子树的权值之和。  构造哈夫曼树的过程可以使用贪心算法，具体步骤如下：  1.将n个权值作为n棵只有根结点的二叉树。  2.从这n棵二叉树中选出权值最小的两棵二叉树，将它们合并成一棵新的二叉树，新的二叉树的根结点权值为原来两棵二叉树的根结点权值之和。  3.将新的二叉树插入到原来的二叉树集合中，并从集合中删除原来的两棵二叉树。  4.重复步骤2和3，直到集合中只剩下一棵二叉树为止，这棵二叉树就是赫夫曼树。  哈夫曼编码的实现也是通过遍历哈夫曼树来实现的，具体步骤如下：  1.从根结点开始遍历哈夫曼树。  2.每当遇到左子树时，将编码的最后一位设为0，每当遇到右子树时，将编码的最后一位设为1。  3.当遍历到叶子结点时，将该结点对应的字符和编码存储起来。  4.重复步骤1到3，直到遍历完整棵赫夫曼树。  运用的理论知识：  哈夫曼编码  哈夫曼树  贪心算法  算法步骤：  1.读入权值的个数n和n个权值。  2.将n个权值作为n棵只有根结点的二叉树。  3.使用贪心算法构造哈夫曼树。  4.从根结点开始遍历哈夫曼树，记录每个字符的编码。  5.输出每个字符对应的编码。  实现代码如下:  typedef struct  {  int weigh;  int parent, lchild, rchild;  }HTcode,\*HuffmanTree;  typedef char\*\* HuffmanCode;//动态分配数组存储哈弗曼编码表  void Select(HuffmanTree HT, int len, int& s1, int& s2)  {  int i;  int min1 = 10000;//记录最小权值  int min2 = 10000;  for (i = 1; i <= len; i++)  {  if (HT[i].parent == 0 && min1 > HT[i].weigh)  {  min1 = HT[i].weigh;  s1 = i;//记录下标  }  }  for (i = 1; i <= len; i++)  {  if (i != s1 && HT[i].parent == 0 && min2 > HT[i].weigh)//不与s1下标重合  {  min2 = HT[i].weigh;  s2 = i;  }  }  }  void CreateHuffmanTree(HuffmanTree& HT, int n)  {  if (n <= 1)return;  int s1 = 0, s2 = 0;  int i;  int m = 2 \* n - 1;//数组总长度  HT = new HTcode[m + 1];//第0个单元不用,所以长度为2n  for (i = 1; i <= m; i++)  {  HT[i].parent = 0;  HT[i].lchild = 0, HT[i].rchild = 0;//数组全部初始化为0  }  for (i = 1; i <= n; i++)  {  cin >> HT[i].weigh;//输入n个叶子单元的权值  }  for (i = n + 1; i <= m; i++)  {  Select(HT, i - 1, s1, s2);//选择权值最小的两个结点  HT[s1].parent = i, HT[s2].parent = i;  HT[i].lchild = s1, HT[i].rchild = s2;  HT[i].weigh = HT[s1].weigh + HT[s2].weigh;  }  }  void CreatHuffmanCode(HuffmanTree HT, HuffmanCode& HC, int n)  {  int i;  HC = new char\* [n + 1];  char \*cd = new char[n];//分配临时存放每个字符编码的动态数组空间  cd[n - 1] = '\0';  for (i = 1; i <= n; i++)  {  int start = n - 1;//start开始时指向最后  int c = i;  int f = HT[i].parent;//f指向c的双亲节点  while (f != 0)  {  --start;  if (HT[f].lchild == c) cd[start] = '0';//左孩子生成0  else cd[start] = '1';//右孩子生成1  c = f, f = HT[f].parent;//继续向上回溯  }  HC[i] = new char[n - start];//为第i个字符编码分配空间  strcpy(HC[i], &cd[start]);  cout << HC[i] << endl;  }  delete cd;  }  实验五 图的存储及遍历  实验: 深度优先搜索  深度优先搜索（DFS）是一种用于遍历或搜索树或图的算法。它从根节点开始，尽可能深地访问每个节点，直到到达最深的节点，然后回溯到前一个节点，尝试访问其他节点。DFS可以用递归实现。  算法步骤：  1.访问初始节点v，并标记为已访问。  2.查找节点v的第一个邻接点w。  3.若w不存在，则回溯到前一个节点，重复步骤2。  4.若w存在但未被访问，则访问w并标记为已访问，然后重复步骤2。  5.重复步骤3、4，直到所有节点都被访问。  实现DFS的关键是如何记录已访问的节点和节点的邻接关系。这里我们选择使用邻接矩阵来存储图，其中矩阵元素a[i][j]表示顶点i和顶点j之间是否有边。在DFS操作中，我们需要用一个visited数组来记录每个节点是否被访问过。  代码实现:  typedef struct {  string vexs[MAX\_VERTEX\_NUM]; // 顶点表  int arcs[MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM]; // 邻接矩阵  int vexnum, arcnum; // 图的当前顶点数和弧数  } Graph;  void DFS(Graph g, int v)  {  visited[v] = true;  cout << g.vexs[v] << " ";  for (int i = 0; i < g.vexnum; i++) {  if (g.arcs[v][i] == 1 && !visited[i]) {  DFS(g, i);  }  }  }  void DFSTraverse(Graph g)  {  memset(visited, false, sizeof(visited)); // 初始化visited数组  for (int i = 0; i < g.vexnum; i++) {  if (!visited[i]) {  DFS(g, i);  }  }  }  实验: 广度优先搜索  广度优先搜索（BFS）是一种用于遍历或搜索树或图的算法。它从根节点开始，逐层遍历每个节点，直到到达最深的节点。BFS可以用队列实现。  算法步骤：  1.将起始节点加入队列中，并将该节点标记为已访问。  2.从队列中取出队首节点，依次访问其所有未被访问的邻接节点，并将其加入队列中，并将这些节点都标记为已访问。  3.重复步骤2，直到队列为空。  具体实现时，我们需要维护一个visited数组，用于记录每个节点是否被访问过。在将节点加入队列之前，需要先判断该节点是否已经被访问过。同时，为了避免重复访问，我们还需要在将节点加入队列之后，立即将其标记为已访问。  另外，为了记录节点之间的邻接关系，我们可以使用邻接表来存储图。邻接表是一种链式存储结构，每个节点对应一个链表，链表中存储该节点的所有邻接节点。在访问节点的邻接节点时，只需要遍历该节点对应的链表即可。  代码实现:  //邻接表存储  typedef struct ArcNode {  int adjvex; //邻接点在数组中的位置下标  struct ArcNode\* nextarc; //指向下一个邻接点的指针  }ArcNode;  typedef struct VNode {  string data; //顶点信息  ArcNode\* firstarc; //指向第一个邻接点的指针  }VNode, AdjList[MAX\_VERTEX\_NUM];  typedef struct {  AdjList vertices; //邻接表  int vexnum, arcnum; //图的当前顶点数和边数  }Graph;  //BFS遍历  void BFS(Graph G, int v) {  bool visited[MAX\_VERTEX\_NUM] = { false }; //标记数组，初始化为false  queue<int> q;  cout << G.vertices[v].data << " "; //访问初始节点  visited[v] = true; //标记初始节点已访问  q.push(v); //初始节点入队  while (!q.empty()) {  int u = q.front();  q.pop();  ArcNode\* p = G.vertices[u].firstarc;  while (p != NULL) {  int w = p->adjvex;  if (!visited[w]) { //如果邻接点未被访问  cout << G.vertices[w].data << " "; //访问邻接点  visited[w] = true; //标记邻接点已访问  q.push(w); //邻接点入队  }  p = p->nextarc;  }  }  }  实验六 查找  实验: 折半查找  折半查找（Binary Search）是一种在有序数组中查找目标元素的算法。它的原理是每次将待查找区间的中间元素与目标元素进行比较，如果相等则返回该元素下标，如果目标元素小于中间元素，则在左半部分继续查找，否则在右半部分继续查找，直到找到目标元素或者待查找区间为空。  算法步骤：  1.初始化待查找区间的左右边界low和high。  2.计算待查找区间的中间位置mid。  3.比较目标元素和中间元素的大小关系。  4.如果目标元素等于中间元素，则返回中间元素下标。  5.如果目标元素小于中间元素，则在左半部分继续查找，将high更新为mid-1。  6.如果目标元素大于中间元素，则在右半部分继续查找，将low更新为mid+1。  7.重复步骤2~6，直到找到目标元素或者待查找区间为空。  代码实现:  int search\_bin(InfoType a[], long long n, long long key)  {  long long low, high;  low = 1; high = n;  while (low <= high)  {  long long mid = (low + high) / 2;  if (key == a[mid])  {  cout << "yes" << endl;  return 0;  }  else if (key < a[mid]) high = mid - 1;  else low = mid + 1;  }  cout << "no" << endl;  return 0;  }  实验七 排序  实验: 快速排序  快速排序（Quick Sort）是一种高效的排序算法，它的核心思想是分治法。具体来说，快速排序将待排序的序列分成两个子序列，然后对这两个子序列分别进行排序，最后将两个有序子序列合并成一个有序序列。  算法步骤：  1.选择一个基准元素pivot，一般选择第一个元素或最后一个元素。  2.将序列中所有小于pivot的元素放在pivot的左边，所有大于pivot的元素放在pivot的右边，相等的元素可以放在任意一边。  3.对pivot左边和右边的子序列分别进行快速排序。  4.重复步骤1~3，直到子序列的长度为1。  代码实现:  typedef struct {  int key;  }RedType;  typedef struct {  RedType r[100000];//r[0]作哨兵  int length;//长度  }Sqlist;  int Partition(Sqlist& L, int low, int high)  {  int pivotkey;//枢轴  L.r[0] = L.r[low];  pivotkey = L.r[low].key;  while (low < high)  {  while (low < high && L.r[high].key >= pivotkey)  {  --high;  }  L.r[low] = L.r[high];//将比枢轴小的移到前端  while (low < high && L.r[low].key <= pivotkey)  {  ++low;  }  L.r[high] = L.r[low];//将比枢轴大的移到后端  }  L.r[low] = L.r[0];//枢轴记录  return low;  }  void Qsort(Sqlist& L, int low, int high)//对子表进行快速排序  {  int pivotloc;  if (low < high)//长度大于1  {  pivotloc = Partition(L, low, high);  Qsort(L, low, pivotloc - 1);//左子表递归排序  Qsort(L, pivotloc + 1, high);//右子表递归排序  }  }  int main()  {  int i;  Sqlist L;  cin >> n;  L.length=n;  for (i = 1; i <= n; i++)  {  cin >> L.r[i].key;  }  Qsort(L, 1, n);  for (i = 1; i <= n; i++)  {  cout << L.r[i].key<<" ";  }  } | | | |
| 实验结果与分析  实验一 线性表的基本操作与应用  实验:合成有序链表  该函数实现了链表的归并操作，用于将两个有序链表合并成一个有序链表。具体来说，该函数使用三个指针pa、pb和pc，分别指向两个链表和合并后的链表。在循环中，每次比较pa和pb指向的节点的数据大小，将较小的节点接入pc链表中，并将对应的指针向后移动。如果pa和pb指向的节点数据相等，则将pa和pb都接入pc链表中，并将pb指向的节点删除。最后，如果pa链表还有剩余节点，则将其接入pc链表中，如果pb链表还有剩余节点，则将其接入pc链表中。最后，删除无用的头结点LB。  该函数的时间复杂度为O(m+n)  实验: 两个稀疏多项式相加  该函数实现了两个多项式的加法操作。具体来说，该函数使用三个指针p1、p2和p3，分别指向两个多项式和合并后的多项式。在循环中，每次比较p1和p2指向的节点的指数大小，根据指数的大小关系将对应的节点接入p3链表中，并将对应的指针向后移动。如果p1和p2指向的节点指数相等，则将它们的系数相加，如果系数和不为0，则将p1指向的节点的系数值修改为两项系数之和，将修改后的p1节点接入p3链表中，同时将p2指向的节点删除。如果系数和为0，则同时将p1和p2指向的节点删除。最后，如果p1链表还有剩余节点，则将其接入p3链表中，如果p2链表还有剩余节点，则将其接入p3链表中。最后，删除无用的头结点Pb。  该函数的时间复杂度为O(m+n)  实验二 栈和队列的基本操作与应用  实验: 表达式求值  该程序实现了一个简单的表达式求值功能，可以计算包含加减乘除和括号的表达式。具体来说，该程序使用两个栈nums和ops，分别存放数值和运算符。在遍历表达式时，如果遇到数字，则将其转化为整数并入栈nums；如果遇到左括号，则将其入栈ops；如果遇到右括号，则将括号内的表达式计算，直到遇到左括号为止；如果遇到运算符，则将其与栈顶的运算符比较优先级，如果栈顶运算符优先级高，则将栈顶运算符和数值出栈进行计算，直到栈顶运算符优先级低于当前运算符或栈为空为止，然后将当前运算符入栈ops。最后，遍历完表达式后，将剩余的运算符和数值出栈进行计算，得到最终结果。  该程序的时间复杂度取决于表达式的长度，为O(n).  实验: 舞伴问题  该程序实现了一个约舞会的功能，可以将男生和女生按照一定的规则匹配在一起。该程序使用两个队列M和F，分别存放男生和女生。在输入每个人的信息时，如果是男生，则将其加入男生队列M中；如果是女生，则将其加入女生队列F中。然后，从两个队列中依次取出一个男生和一个女生进行匹配，直到其中一个队列为空为止。如果男队或女队还有剩余队员，则将其依次输出。  该程序的时间复杂度取决于队列的长度，为O(n)  实验三 树和二叉树的定义与遍历  实验:二叉树遍历（用结构体数组存储）  这段代码实现了二叉树的初始化、前序遍历、中序遍历、后序遍历功能。其中，二叉树的初始化使用了结构体数组来存储每个结点的值、左孩子、右孩子和父结点，通过读入每个结点的值和左右孩子的编号来构建二叉树。前序遍历、中序遍历、后序遍历则使用递归的方式实现。  时间复杂度为O(n)  实验四 哈夫曼树与哈夫曼编码  实验: huffman编码（1）  输入n个叶子节点的权值，通过构建哈夫曼树，得到每个叶子节点的编码。其中，哈夫曼树的构建使用了贪心算法，每次选择权值最小的两个节点合并，直到最终构建出整个哈夫曼树。  代码中使用了结构体来存储哈夫曼树的节点信息，同时使用动态分配数组存储哈夫曼编码表。在生成哈夫曼编码时，使用了回溯法，从叶子节点开始向上回溯，记录每个节点的编码。  整体来说，该代码实现了哈夫曼编码的生成，时间复杂度为O(nlogn)，其中n为叶子节点的个数。因为哈夫曼树的构建需要进行n-1次合并操作，每次合并操作的时间复杂度为O(logn)，因此总的时间复杂度为O(nlogn)。  实验五 图的存储及遍历  实验: 深度优先搜索  该算法的思想是从一个未被访问的节点开始，依次遍历与其相邻的节点，并标记已经访问过的节点。当所有节点都被访问过时，遍历结束。其中DFS函数用于递归遍历每个连通分量，DFSTraverse函数用于遍历整个图。实验结果表明，该算法能够正确地遍历图中的每个节点，并输出它们的值。同时，该算法的时间复杂度为O(V+E)，其中V为节点数，E为边数。  实验:广度优先搜索  该算法可以用于求解有权图中单源最短路径问题。给定一个起始节点和一个有向有权图，该算法会计算出从起始节点到其他所有节点的最短路径长度。 假设有V个顶点和E条边，那么该算法的时间复杂度为O(V^2)，其中V表示顶点数。  实验六 查找  实验:折半查找  该函数实现了折半查找算法，用于在一个已排序的数组中查找指定的元素。  该函数的时间复杂度为O(logn)，其中n为数组元素个数。由于每次循环都将查找范围缩小一半，因此查找的时间复杂度是对数级别的。  实验七 排序  实验:快速排序  该程序实现了快速排序算法，用于对输入的n个数进行排序。  该程序的时间复杂度为O(nlogn)，其中n为数组元素个数。快速排序算法的时间复杂度和划分的方式有关，最坏情况下的时间复杂度为O(n^2)，但通常情况下的时间复杂度为O(nlogn)。 | | | |