中北大学软件学院

**实验报告**

|  |  |
| --- | --- |
| 专业： | 软件工程 |
| 课程名称： | 数据结构与算法 |
| 班 级： | 21130420 |
| 学 号： | 2113042015 |
| 姓 名： | 程楚晋 |
| 辅导教师： | 马巧梅 |

2022年3月制

成绩：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验时间 | 2022年5月4日19时至21时 | 学时数 | 2学时 | |
| 1.实验名称  **树和二叉树的应用** | | | | |
| 2.实验目的  （1）掌握二叉树的结构特性，各种存储结构及其适用范围；  （2）掌握二叉树遍历算法。 | | | | |
| 3.实验内容  **基本要求：**  算法：输入字符序列，建立二叉树；按先序、中序、后序遍历二叉树（三种递归算法和中序遍历的非递归算法）；  **选作内容：**  求二叉树的高度和结点的个数。 | | | | |
| 4.拟遍历的二叉树（此处画出你计划遍历的二叉树和此二叉树对应的二叉链表存储结构）  二叉链表：二叉链表是指链表中的每个结点包含两个指针域和一个数据域，分别用来存储指向二叉树中的结点的左右孩子的指针及结点信息(三叉链表比二叉链表多出一个指针域用来指向该节点的双亲结点)  遍历的二叉树 相应的二叉链表存储结构 | | | | |
| 5.实验源代码   1. #include "iostream" 2. **using** **namespace** std; 3. #include<stdio.h> 4. #include<stdlib.h> 5. #define OK 1 6. #define ERROR 0 7. #define OVERFLOW -1 8. #define TRUE 1 9. #define FALSE 0 10. #define MAXSIZE 100 11. #define STACKINCREMENT 10 12. **typedef** **int** Status; 13. **typedef** **float** TElemType; 14. //二叉树的二叉链表存储结构定义 15. **typedef** **struct** BiTNode{ 16. **char** data; 17. **struct** BiTNode \*lchild,\*rchild; 18. }BiTNode,\*BiTree; 19. //顺序栈的存储结构 20. **typedef** BiTree SElemType; 21. **typedef** **struct** 22. { 23. SElemType \*base;  //栈底指针 24. SElemType \*top;  //栈顶指针 25. **int** stacksize;    //栈最大可用容量 26. }SqStack; 27. //创建二叉树 28. **void** InitBiTree(BiTree &T) 29. { 30. T=NULL; 31. } 32. //因为创建的时候就是从无到有，所以不需要InitBiTree() 33. //先序遍历的顺序创建二叉树 34. Status CreateBiTree(BiTree &T) 35. { 36. **char** ch; 37. ch = getchar(); 38. **if** (ch == '#') T = NULL; 39. **else** 40. { 41. **if** (!(T = (BiTree)malloc(**sizeof**(BiTNode))))  exit(OVERFLOW); 42. T->data = ch;              // 生成根结点 43. CreateBiTree(T->lchild);      // 构造左子树 44. CreateBiTree(T->rchild);      // 构造右子树 45. } 46. **return** OK; 47. } 48. /\*遍历二叉树\*/ 49. //前序遍历 50. **void** PreOrderTraverse(BiTree T) 51. { 52. **if**(T){ 53. printf("%c ",T->data); 54. PreOrderTraverse(T->lchild); 55. PreOrderTraverse(T->rchild); 56. } 57. } 58. //51ye 非递归 59. //中序遍历 60. **void** InOderTraverse(BiTree T) 61. { 62. **if**(T){ 63. InOderTraverse(T->lchild); 64. printf("%c ",T->data); 65. InOderTraverse(T->rchild); 66. } 67. } 68. //后序遍历 69. **void** PostOrderTraverse(BiTree T) 70. { 71. **if**(T){ 72. PostOrderTraverse(T->lchild); 73. PostOrderTraverse(T->rchild); 74. printf("%c ",T->data); 75. } 76. } 77. //中序遍历的非递归遍历 78. Status InitStack(SqStack &S) 79. {//构造一个空栈 80. S.base= **new** SElemType[MAXSIZE];  //为顺序栈动态分配一个最大容为MAXSIZE的数组空间 81. //  或者这样写 82. // S.base = (TElemType \*)malloc(MAXSIZE \* sizeof (TElemType)); 83. **if** (!S.base)  exit(OVERFLOW);   //储存分配失败 84. S.top=S.base;   //top初始为base，空栈 85. S.stacksize=MAXSIZE;  //stacksize置为栈的最大容量MAXSIZE 86. //if(S.base) printf("置空栈成功\n"); 87. **return** OK; 88. } 89. //入栈操作：在栈顶插入一个新的元素 90. Status Push(SqStack &S, SElemType e) 91. {//插入元素e为新的栈顶元素 92. **if**(S.top-S.base==S.stacksize)  **return** ERROR;     //栈满 93. **if** (S.top - S.base > S.stacksize) 94. { 95. S.base = (SElemType \*)realloc(S.base, (S.stacksize + STACKINCREMENT) \* **sizeof**(SElemType)); 96. **if** (!S.base) 97. { 98. printf("栈溢出!\n"); 99. **return** OVERFLOW; 100. } 101. S.top = S.base + S.stacksize; 102. S.stacksize += STACKINCREMENT; 103. }//若栈满，追加存储空间 104. \*S.top++ = e;   //元素e压入栈顶，栈顶指针+1 105. /\*上面的句子本质 106. \*S.top = e; 107. S.top++;\*/ 108. **return** OK; 109. } 110. //出栈操作：将栈顶元素删除 111. Status Pop(SqStack &S,SElemType &e) 112. {//删除S的栈顶元素，用e返回其值 113. **if**(S.base==S.top)   **return** ERROR;   //栈空 114. e=\*(--S.top);   //栈顶指针减1，将栈顶元素给e 115. **return** OK; 116. } 118. //判断栈是否为空 119. Status StackEmpty(SqStack S) 120. { 121. **if** (S.top == S.base) 122. **return** TRUE; 123. **else** 124. **return** FALSE; 125. } 127. **int** inorder(BiTree T) 128. { 129. SqStack S; 130. BiTree p=T; 131. InitStack(S); 132. **while**(p || !StackEmpty(S)) 133. { 134. **if**(p) { Push(S,p); p = p->lchild; } //非空指针进栈，继续左进 135. **else** { 136. //上层指针退栈，访问其所指结点，再右进 137. Pop(S,p); 138. // if(!Visit(p->data)) return ERROR; 139. printf("%c ",p->data); 140. p=p->rchild; 141. } 142. } 143. **return** OK; 144. } 145. //层次遍历二叉树 146. //层次遍历二叉树T的递归算法，对每个数据元素调用函数Visit，采用队列作为辅助结构 147. Status LevelOrderTraverse(BiTree T, Status(\*Visit)(**char**)) { 148. BiTNode \*Queue[100], \*p = T; 149. **int** front = 0, rear = 0; 150. **if** (p != NULL) { 151. Queue[++rear] = p;    /\*   根结点入队  \*/ 152. **while** (front < rear) { 153. p = Queue[++front]; 154. Visit(p->data); 155. **if** (p->lchild != NULL) 156. Queue[++rear] = p->lchild;    /\*   左结点入队  \*/ 157. **if** (p->rchild != NULL) 158. Queue[++rear] = p->rchild;    /\*   左结点入队  \*/ 159. } 160. **return** OK; 161. } 162. **return** ERROR; 163. } 164. Status PrintBiTree(**char** e) 165. { 166. printf("%c ",e); 167. **return** OK; 168. } 169. //统计叶子结点个数 170. **int** CountLeafs(BiTree root) { 171. **int** sum=0; 172. **if** (root == NULL) { 173. **return** 0; 174. } 175. **if** (root->lchild == NULL && root->rchild == NULL) { 176. **return** 1; 177. } 178. sum = CountLeafs(root->lchild) + CountLeafs(root->rchild); 179. **return** sum; 180. } 181. **int** CountRoot(BiTree ROOT) 182. { 183. **if**(ROOT ==NULL) **return** 0; 184. **else** **return** CountRoot(ROOT->rchild)+CountRoot(ROOT->lchild)+1; 185. } 187. //统计树的层次数 188. /\* 初始条件: 二叉树T存在。操作结果: 返回T的深度 \*/ 189. **int** CountLevels(BiTree T) 190. { 191. **if** (T == NULL) 192. **return** 0; 193. **int** l = 0, r = 0; 194. **if** (T->lchild) 195. l = CountLevels(T->lchild); 196. **if** (T->rchild) 197. r = CountLevels(T->rchild); 198. **return** l > r ? l + 1 : r + 1; 199. } 201. **int** main() 202. { 203. BiTree T; 204. **int** n = 0,b; 205. **char** s; 206. CreateBiTree(T); 207. cout<<"前序遍历递归\n"; 208. PreOrderTraverse(T); 209. cout<<endl<<"中序遍历递归\n"; 210. InOderTraverse(T); 211. cout<<endl<<"后序遍历递归\n"; 212. PostOrderTraverse(T); 213. cout<<endl<<"中序非递归\n"; 214. inorder(T); 215. cout<<endl<<"层次遍历"; 216. LevelOrderTraverse(T, PrintBiTree); 217. cout<<endl<<"叶子结点个数 "; 218. n = CountLeafs(T); 219. cout<<n; 220. cout<<endl<<"树高度 "; 221. cout<<CountLevels(T); 222. cout<<endl<<"结点个数 "; 223. cout<<CountRoot(T); 224. **return** 0; 225. } 226. //abd#e##fg###c##   运行结果如下： | | | |
| 6.实验结论及心得  1.遍历二叉树，就是按某种次序访问书中的结点，要求每个结点访问一次且仅访问一次  2.遍历的结果：产生一个关于结点的线性序列  3.先序遍历：DLR  中序遍历：LDR  后序遍历：LRD  4.遍历的算法分为递归和非递归，具体见源代码  5.在栈的数据存储类型中，需要将栈与二叉树相结合，所以需要在存储的指针类型中  将 栈的指针类型定义为BiTree类型，但是栈中需要的是栈底和栈顶指针，所以，要定义成BiTree \*类型（如图所示）  6.在创建二叉树的时候，只有中序不能单独创建二叉树，需要结合先序遍历 | | | |