**数据结构实验报告—二叉树基本操作的实现**

设计者姓名：张帆

设计者班级：2班

设计者学号：20192131077

上机环境：DEV-C++

设计日期：2020-11-10

1. **实验题目**

选择二叉链式存储结构作为二叉树的存储结构，设计一个程序实现二叉树的

基本操作（包括建立、输出、前序遍历、中序遍历、后序遍历、求树高、统计叶

子总数等）。

程序的菜单功能项如下：

1------建立一棵二叉树（假设所有结点值唯一）

2------前序遍历递归算法

3------前序遍历非递归算法

4------中序遍历递归算法

5------中序遍历非递归算法

6------后序遍历递归算法

7------后序遍历非递归算法

8------求树高

9------求叶子总数

10-----求二叉树中值为 x 的结点的层次

11-----输出值为 x 的结点的所有祖先

12-----采用层次遍历方法输出二叉树的每一层的结点（每行输出一层的所有结

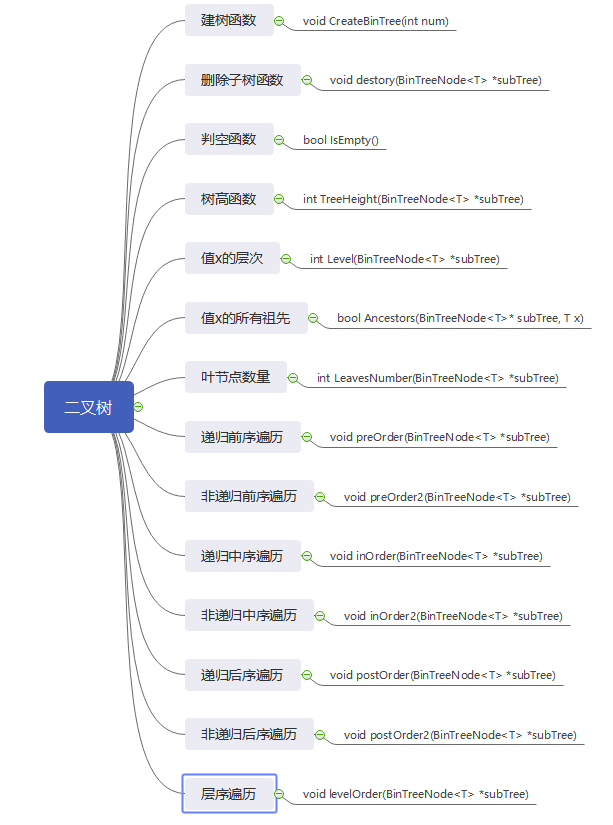
点）

13-----退出

1. **实验项目目的**

使学生领会二叉链式存储结构和掌握二叉链式存储中各种基本运算算法设计

1. **实验项目的程序结构**



**四、实验项目包含的各个文件中的函数的功能描述**

1. #include <iostream>
2. #include <cmath>
3. #include <stack>
4. #include <queue>
5. #include <vector>
6. #include <cstdio>
7. #include <cstdlib>
8. #include <cstring>
9. #include <algorithm>
10. **using** **namespace** std;
11. /\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/
13. **template** <**class** T>
14. **struct** BinTreeNode
15. {
16. T data;                //节点数据
17. **int** index;             //顺序存储二叉树的数组下标
18. BinTreeNode<T> \*leftChild, \*rightChild;  //左、右节点指针
19. BinTreeNode(): leftChild(NULL), rightChild(NULL) {}
20. BinTreeNode(T x, BinTreeNode<T> \*l = NULL, BinTreeNode<T> \*r = NULL):
21. data(x), leftChild(l), rightChild(r) {}
22. };
24. **template** <**class** T>
25. **class** BinaryTree
26. {
27. **public**:
28. BinTreeNode<T>\* root;        //指向根节点的指针

31. BinaryTree():root(NULL) {}  //构造函数
32. ~BinaryTree()               //析构函数
33. {
34. destory(root);
35. }
36. **bool** IsEmpty()              //判二叉树空否
37. {
38. **return** (root == NULL) ? **true** : **false**;
39. }
40. **void** CreateBinTree(**int** num);        //建立链式二叉树
41. **void** destory(BinTreeNode<T> \*subTree);    //删除当前节点的子树
42. **int** TreeHeight(BinTreeNode<T> \*subTree);   //树的高度
43. **int** LeavesNumber(BinTreeNode<T> \*subTree); //叶节点数量
44. **void** preOrder(BinTreeNode<T> \*subTree);   //递归前序遍历
45. **void** preOrder2(BinTreeNode<T> \*subTree);  //非递归前序遍历
46. **void** inOrder(BinTreeNode<T> \*subTree);    //递归中序遍历
47. **void** inOrder2(BinTreeNode<T> \*subTree);   //非递归中序遍历
48. **void** postOrder(BinTreeNode<T> \*subTree);  //递归后序遍历
49. **void** postOrder2(BinTreeNode<T> \*subTree); //非递归后序遍历
50. **void** levelOrder(BinTreeNode<T> \*subTree); //层次遍历
51. **int** Level(BinTreeNode<T> \*subTree);       //值x的层次
52. **bool** Ancestors(BinTreeNode<T>\* subTree, T x); //值x的所有祖先
54. };
56. **template** <**class** T>
57. **void** BinaryTree<T>::CreateBinTree(**int** num)  //建立链式二叉树
58. {
59. T x;
60. vector<T> Temp;
61. Temp.push\_back(T(0));
62. **for** (**int** i = 0; i < num; i++)
63. {
64. cin >> x;
65. Temp.push\_back(x);
66. }
67. BinTreeNode<T>\* t = **new** BinTreeNode<T>(Temp[1]);
68. t->index = 1;
69. root = t;
70. queue<BinTreeNode<T>\*> Q;
71. Q.push(root);
72. **for** (**int** i = 1; i <= num; i++)
73. {
74. **if** (Temp[i] != 0)
75. {
76. BinTreeNode<T>\* cur = Q.front();
77. Q.pop();
78. BinTreeNode<T>\* right = **new** BinTreeNode<T>();
79. BinTreeNode<T>\* left = **new** BinTreeNode<T>();
80. **int** lc = i << 1, rc = i << 1 | 1;
81. **if** (lc < num + 1 && Temp[lc] != 0)
82. {
83. left->data = Temp[lc];
84. left->index = lc;
85. cur->leftChild = left;
86. Q.push(left);
87. }
88. **if** (rc < num + 1 && Temp[rc] != 0)
89. {
90. right->data = Temp[rc];
91. right->index = rc;
92. cur->rightChild = right;
93. Q.push(right);
94. }
95. }
96. }
97. cout << "建立二叉树成功！" << endl;
98. }
100. **template** <**class** T>
101. **int** BinaryTree<T>::TreeHeight(BinTreeNode<T> \*subTree)  //树的高度
102. {
103. **if** (subTree == NULL)
104. {
105. **return** 0;
106. }
107. **int** a = TreeHeight(subTree->leftChild);
108. **int** b = TreeHeight(subTree->rightChild);
109. **return** max(a, b) + 1;   //返回左和右子树高度中更高的那个
110. }
112. **template** <**class** T>
113. **void** BinaryTree<T>::preOrder(BinTreeNode<T> \*subTree)  //递归前序遍历
114. {
115. **if** (subTree == NULL)  **return**;
116. cout << subTree->data << " ";
117. preOrder(subTree->leftChild);
118. preOrder(subTree->rightChild);
119. }
121. **template** <**class** T>
122. **void** BinaryTree<T>::preOrder2(BinTreeNode<T> \*subTree)  //非递归前序遍历
123. {
124. stack<BinTreeNode<T>\*> S;
125. BinTreeNode<T>\* p = subTree;
126. S.push(p);
127. **while** (S.size())
128. {
129. p = S.top();
130. S.pop();
131. cout << p->data << " ";
132. **if** (p->rightChild != NULL)
133. {
134. S.push(p->rightChild);
135. }
136. **if** (p->leftChild != NULL)
137. {
138. S.push(p->leftChild);
139. }
140. }
141. }
143. **template** <**class** T>
144. **void** BinaryTree<T>::inOrder(BinTreeNode<T> \*subTree)  //递归中序遍历
145. {
146. **if** (subTree == NULL)  **return**;
147. inOrder(subTree->leftChild);
148. cout << subTree->data << " ";
149. inOrder(subTree->rightChild);
150. }
152. **template** <**class** T>
153. **void** BinaryTree<T>::inOrder2(BinTreeNode<T> \*subTree)  //非递归中序遍历
154. {
155. stack<BinTreeNode<T>\*> S;
156. BinTreeNode<T>\* p = subTree;
157. **do** {
158. **while** (p != NULL)
159. {
160. S.push(p);
161. p = p->leftChild;
162. }
163. **if** (S.size())
164. {
165. p = S.top();
166. S.pop();
167. cout << p->data << " ";
168. p = p->rightChild;
169. }
170. } **while** (p != NULL || S.size());
171. }
173. **template** <**class** T>
174. **void** BinaryTree<T>::postOrder(BinTreeNode<T> \*subTree)  //递归后序遍历
175. {
176. **if** (subTree == NULL)  **return**;
177. postOrder(subTree->leftChild);
178. postOrder(subTree->rightChild);
179. cout << subTree->data << " ";
181. }
183. **template** <**class** T>
184. **void** BinaryTree<T>::postOrder2(BinTreeNode<T> \*subTree)  //非递归后序遍历
185. {
186. stack<pair<BinTreeNode<**int**>\*, **char**> > S;
187. pair<BinTreeNode<**int**>\*, **char**> w;
188. BinTreeNode<T>\* p = subTree;
189. **do** {
190. **while** (p != NULL)
191. {
192. w.first = p;
193. w.second = 'L';
194. S.push(w);
195. p = p->leftChild;
196. }
197. **bool** flag = **true**;
198. **while** (flag && S.size())
199. {
200. w = S.top();
201. S.pop();
202. p = w.first;
203. **switch**(w.second)
204. {
205. **case** 'L':
206. w.second = 'R';
207. S.push(w);
208. flag = **false**;
209. p = p->rightChild;
210. **break**;
211. **case** 'R':
212. cout << p->data << " ";
213. **break**;
214. }
215. }
216. } **while** (S.size());
217. }
219. **template** <**class** T>
220. **void** BinaryTree<T>::destory(BinTreeNode<T> \*subTree) //删除当前节点的子树
221. {
222. **if** (subTree != NULL)
223. {
224. destory(subTree->leftChild);
225. destory(subTree->rightChild);
226. **delete** subTree;    //释放空间
227. }
228. }
230. **template** <**class** T>
231. **int** BinaryTree<T>::LeavesNumber(BinTreeNode<T> \*subTree) //统计叶节点的数量
232. {
233. **static** **int** leafnum = 0;
234. **if** (subTree == NULL)  **return** 0;
235. **if** (subTree->leftChild == NULL && subTree->rightChild == NULL)
236. {
237. leafnum++;   //同时没有左、右节点的才是叶节点
238. }
239. LeavesNumber(subTree->leftChild);
240. LeavesNumber(subTree->rightChild);
241. **return** leafnum;
242. }
244. **template** <**class** T>
245. **void** BinaryTree<T>::levelOrder(BinTreeNode<T> \*subTree) //层次遍历
246. {
247. **if** (subTree == NULL)  **return**;
248. queue<BinTreeNode<T>\*> Q;
249. Q.push(subTree);
250. **int** level = 1;
251. **while** (Q.size())
252. {
253. BinTreeNode<T>\* cur = Q.front();
254. Q.pop();
255. **int** temp = log(cur->index) / log(2) + 1;
256. **if** (temp != level)
257. {
258. cout << endl;
259. level = temp;
260. }
261. cout << cur->data << " ";
262. **if** (cur->leftChild != NULL)  Q.push(cur->leftChild);
263. **if** (cur->rightChild != NULL)  Q.push(cur->rightChild);
264. }
265. }
267. **template** <**class** T>
268. **int** BinaryTree<T>::Level(BinTreeNode<T> \*subTree) //值x的层次
269. {
270. T x;
271. cin >> x;
272. **if** (subTree == NULL) **return** 0;
273. queue<BinTreeNode<T>\*> Q;
274. Q.push(subTree);
275. **int** level = 1;
276. **while** (Q.size())
277. {
278. BinTreeNode<T>\* cur = Q.front();
279. Q.pop();
280. **if** (cur->data == x)
281. {
282. **int** tnum = log(cur->index) / log(2) + 1;
283. **return** tnum;
284. }
285. **if** (cur->leftChild != NULL)  Q.push(cur->leftChild);
286. **if** (cur->rightChild != NULL)  Q.push(cur->rightChild);
287. }
288. **return** 0;
289. }
291. **template** <**class** T>
292. **bool** BinaryTree<T>::Ancestors(BinTreeNode<T>\* subTree, T x)  //值x的所有祖先
293. {
294. **if** (subTree == NULL)  **return** **false**;
295. **if** (subTree->data == x)  **return** **true**;
296. **if** (Ancestors(subTree->leftChild, x) || Ancestors(subTree->rightChild, x))
297. {
298. cout << subTree->data << " ";
299. **return** **true**;
300. }
301. **return** **false**;
302. }


306. **void** Menu()    //菜单函数
307. {
308. **int** choice;
309. BinaryTree<**int**> T;
310. **while** (1)
311. {
312. cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*二叉树程序菜单\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;
313. cout << "\*\*  1.建立二叉树    2.前序递归遍历   3.前序非递归遍历 \*\*" << endl;
314. cout << "\*\*  4.中序递归遍历  5.中序非递归遍历  6.后序递归遍历  \*\*" << endl;
315. cout << "\*\*  7.后序非递归遍历   8.树的高度     9.叶节点总数    \*\*" << endl;
316. cout << "\*\*  10.值x的层次    11.值x的所有祖先  12.层次遍历     \*\*" << endl;
317. cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;
318. cout << "请选择操作：";
319. cin >> choice;
320. **if** (choice == 1)
321. {
322. **int** num;
323. cout << "请输入要建立的二叉树的节点个数：";
324. cin >> num;
325. **if** (num > 0)
326. {
327. cout << "请依次输入节点数据：";
328. T.CreateBinTree(num);
329. }
330. }
331. **else** **if** (choice == 2)
332. {
333. **if** (!T.IsEmpty())
334. {
335. cout << "二叉树的递归前序遍历顺序为：";
336. T.preOrder(T.root);
337. cout << endl;
338. }
339. **else** cout << "二叉树为空！！！" << endl;
340. }
341. **else** **if** (choice == 3)
342. {
343. **if** (!T.IsEmpty())
344. {
345. cout << "二叉树的非递归前序遍历顺序为：";
346. T.preOrder2(T.root);
347. cout << endl;
348. }
349. **else** cout << "二叉树为空！！！" << endl;
350. }
351. **else** **if** (choice == 4)
352. {
353. **if** (!T.IsEmpty())
354. {
355. cout << "二叉树的递归中序遍历顺序为：";
356. T.inOrder(T.root);
357. cout << endl;
358. }
359. **else** cout << "二叉树为空！！！" << endl;
360. }
361. **else** **if** (choice == 5)
362. {
363. **if** (!T.IsEmpty())
364. {
365. cout << "二叉树的非递归中序遍历顺序为：";
366. T.inOrder2(T.root);
367. cout << endl;
368. }
369. **else** cout << "二叉树为空！！！" << endl;
370. }
371. **else** **if** (choice == 6)
372. {
373. **if** (!T.IsEmpty())
374. {
375. cout << "二叉树的递归后序遍历顺序为：";
376. T.postOrder(T.root);
377. cout << endl;
378. }
379. **else** cout << "二叉树为空！！！" << endl;
380. }
381. **else** **if** (choice == 7)
382. {
383. **if** (!T.IsEmpty())
384. {
385. cout << "二叉树的非递归后序遍历顺序为：";
386. T.postOrder2(T.root);
387. cout << endl;
388. }
389. **else** cout << "二叉树为空！！！" << endl;
390. }
391. **else** **if** (choice == 8)
392. {
393. cout << "二叉树的高度为：" << T.TreeHeight(T.root) << endl;
394. }
395. **else** **if** (choice == 9)
396. {
397. cout << "二叉树的叶子总数为：" << T.LeavesNumber(T.root) << endl;
398. }
399. **else** **if** (choice == 10)
400. {
401. cout << "请输入要查找层次的节点的值：";
402. **if** (T.Level(T.root))
403. {
404. cout << "层次为：" << T.Level(T.root) << endl;
405. }
406. **else** cout << "二叉树不存在此值！" << endl;
407. }
408. **else** **if** (choice == 11)
409. {
410. **int** tt;
411. cout << "请输入要查找所有祖先的节点的值：";
412. cin >> tt;
413. cout << tt << "的所有祖先为：";
414. T.Ancestors(T.root, tt);
415. cout << endl;
416. }
417. **else** **if** (choice == 12)
418. {
419. **if** (!T.IsEmpty())
420. {
421. cout << "二叉树的层次遍历顺序为：";
422. T.levelOrder(T.root);
423. }
424. **else** cout << "二叉树为空！！！" << endl;
425. }
426. **else** **break**;
427. cout << endl << endl;
428. }
429. }

432. **int** main(**int** argc, **char**\*\* argv)
433. {
434. Menu();    //菜单显示
435. **return** 0;
436. }

**五、算法描述或流程图**

建立二叉树采用的是数组下标建立二叉树的方法，第一个输入下标i的是根节点，2\*i和2\*i + 1分别是左、右子节点，不断建树

树的高度是如果本节点不为空，不断递归地向上返回左右子树更高的高度 + 1

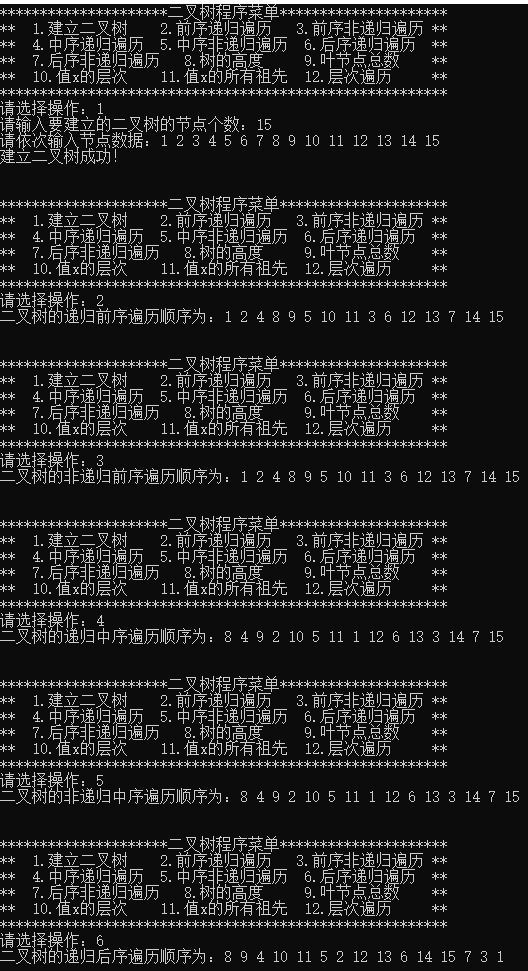
叶节点总数寻找的是左右子子节都为空的节点，然后静态变量不断 + 1

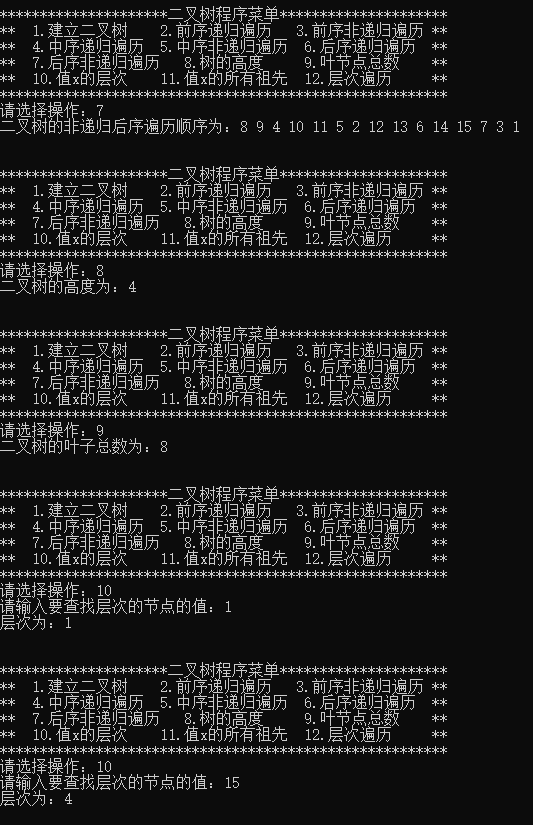
值x的层次不断寻找值为x的节点，然后返回层次

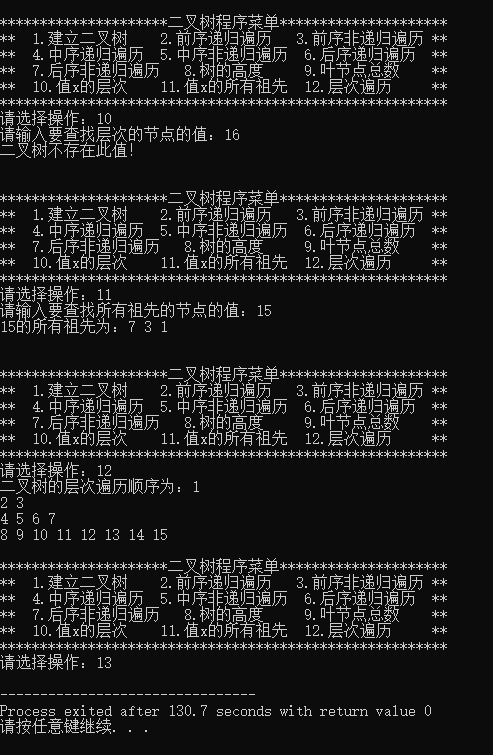
值x的所有祖先不断向上递归输出父节点，直到根节点

层次遍历通过一个队列来输出层次，当队列不为空的时候，不断加入节点和它的左右子节点，然后出队输出节点的数值

**六、实验数据和实验结果分析**







1. **实验体会**

这次实验，让我更加理解了二叉树内部的操作，对二叉树有了一个更深入的了解。