**实验6**

学号： 姓名： 专业：

**推荐使用“Web版式视图”浏览**

**6.1 实验目的**

掌握二叉树的动态链表存储结构及表示。

掌握二叉树的三种遍历算法（递归和非递归两类）。

运用二叉树三种遍历的方法求解有关问题。

**6.2 实验要求**

按照C++面向对象方法编写二叉树类；二叉树的测试数据可用多种方式进行输入，如键盘输入、静态写入、文件读入等。//**最难的是从文件把数据读进去！**

设计二叉树的二叉链表存储结构，编写算法实现下列问题的求解。

<1>打印出二叉树的三种遍历序列。

<2>设计算法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。

<3>求二叉树的高度。

<4>求二叉树的结点数。

<5>求二叉树的叶子结点数。

<6>求二叉树的度为2的结点数。

<7>键盘输入一个元素x，求其父节点、兄弟结点、子结点的值，不存在时给出相应提示信息。对兄弟结点和孩子结点，存在时要明确指出是左兄弟、左孩子、右兄弟或右孩子。

<8>键盘输入一个元素x，求其在树中的层次，不存在时给出相应提示信息。

<9>将按顺序方式存储在数组中的二叉树转换为二叉链表形式。（数组中要扩展为完全二叉树）。

<10>交换二叉树中每个结点的左右孩子指针的值。（即：左子树变为右子树，右子树变为左子树）。

**（下面为选做实验，有兴趣的同学完成）**

<11>复制一棵二叉树T到T1。

<12>输出二叉树从每个叶子结点到根结点的路径（经历的结点）。

<13>对二叉链表表示的二叉树，按从上到下，从左到右打印结点值，即按层次打印。（提示：需要使用队列）

<14>对二叉链表表示的二叉树，求2个结点最近的共同祖先。

实验测试数据基本要求：

<15>求二叉树中一条最长的路径长度（边数），并输出路径上的个结点值。

实验测试数据基本要求：

**6.3 实验数据要求**

自我编写测试样例，要求每个功能函数的测试样例不少于两组

**6.4 运行结果截图及说明**

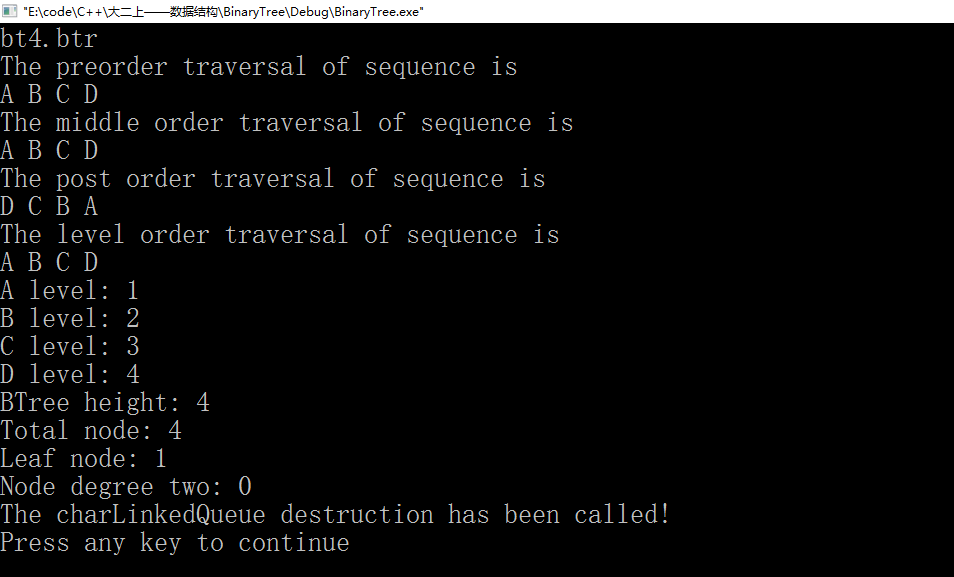


图1 测试（1）、（2）、（3）、（4）、（5）、（6）

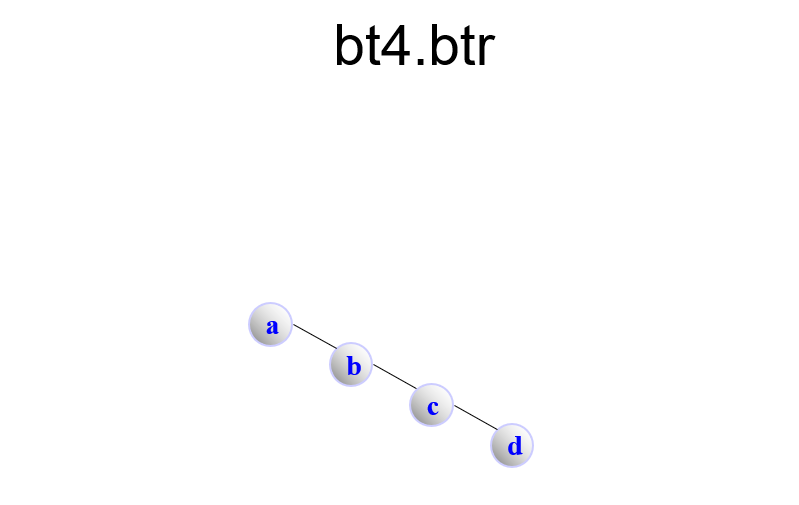


图2 测试（1）、（2）、（3）、（4）、（5）、（6）

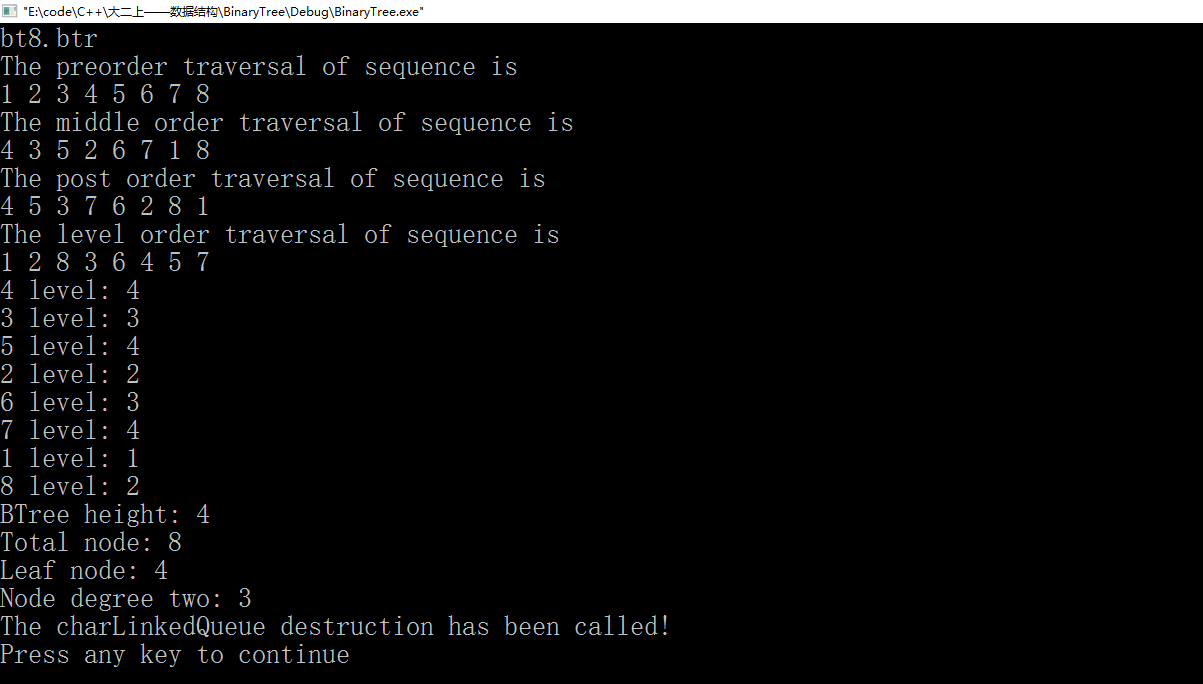


图3 测试（1）、（2）、（3）、（4）、（5）、（6）

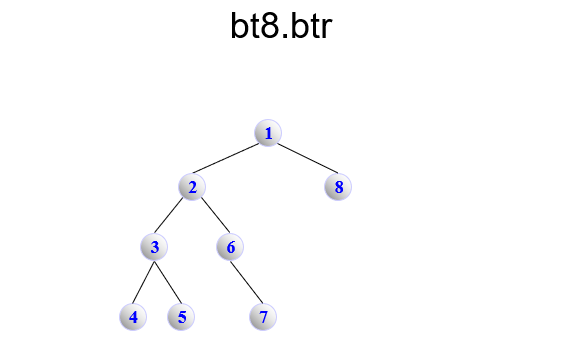


图4 测试（1）、（2）、（3）、（4）、（5）、（6）

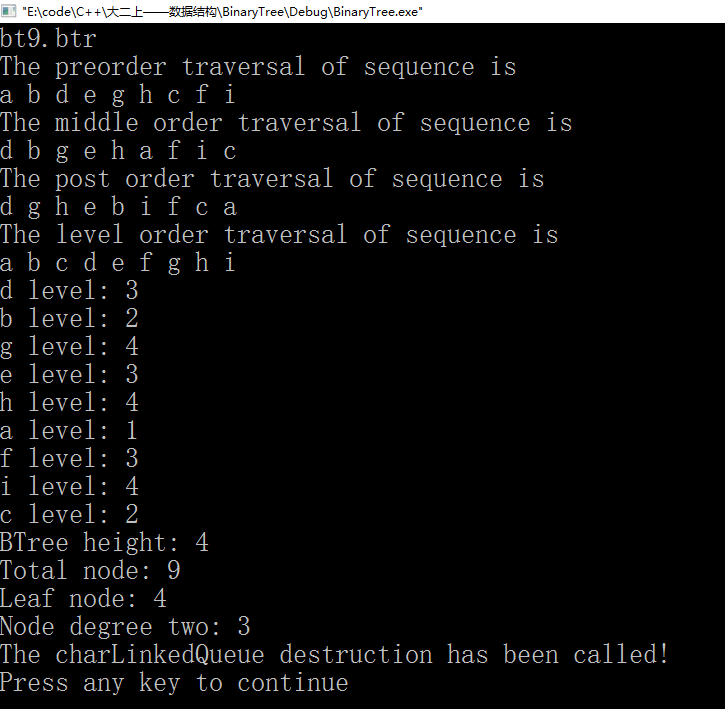


图5 测试（1）、（2）、（3）、（4）、（5）、（6）

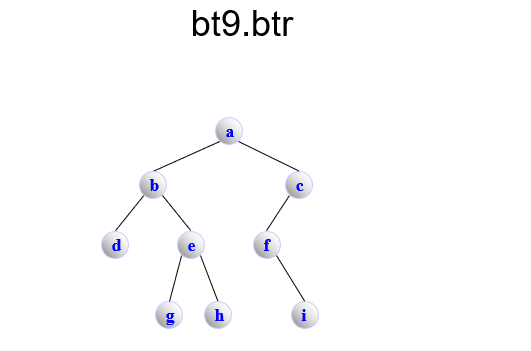


图6 测试（1）、（2）、（3）、（4）、（5）、（6）

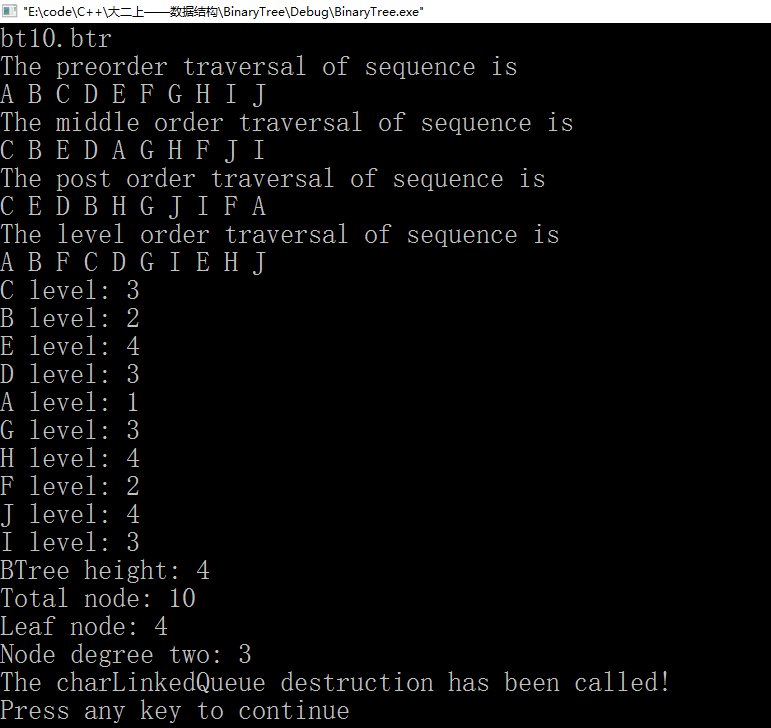


图7 测试（1）、（2）、（3）、（4）、（5）、（6）

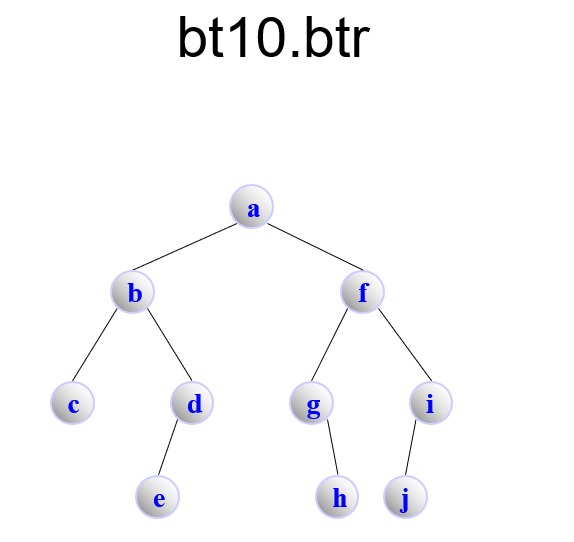
****

图8 测试（1）、（2）、（3）、（4）、（5）、（6）

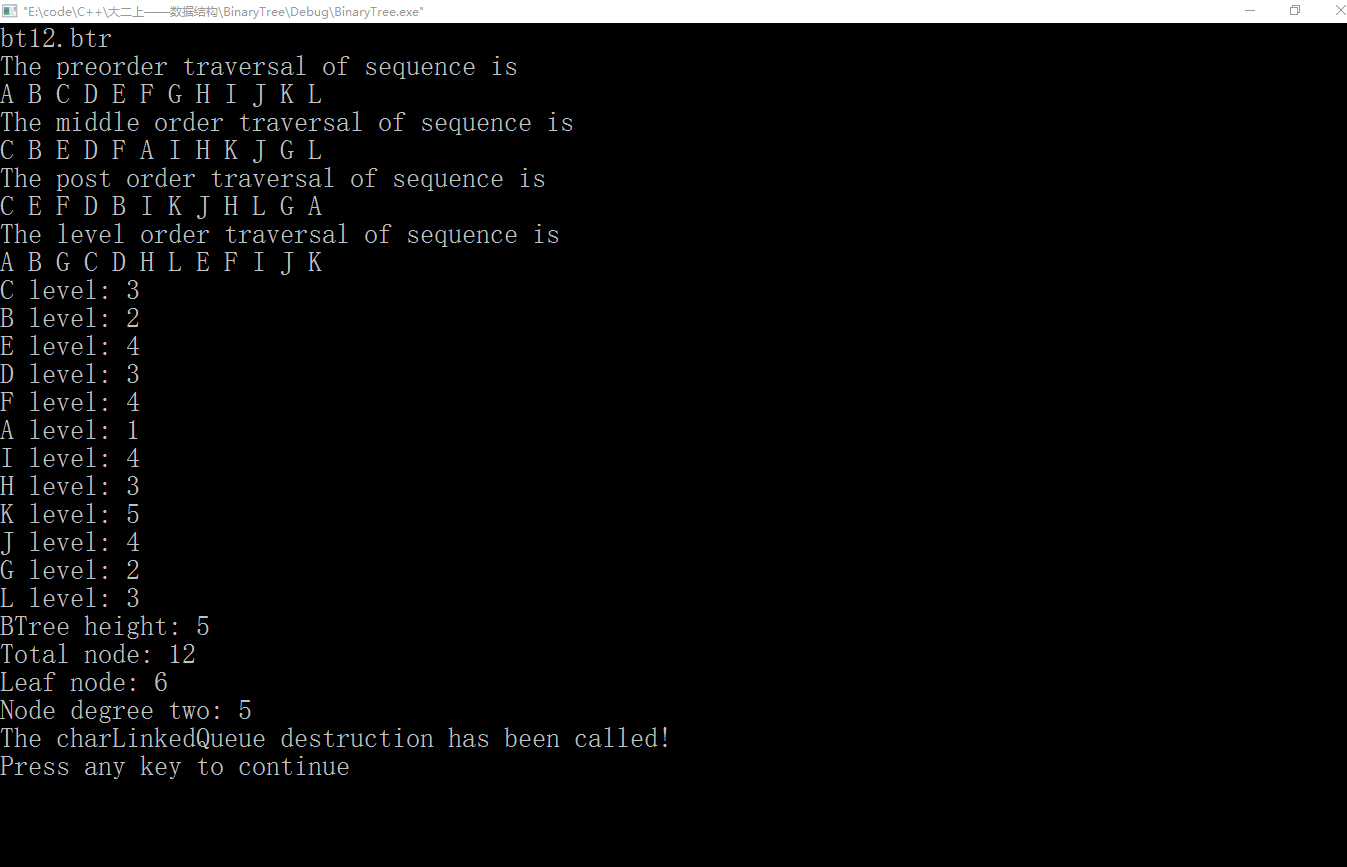
****

图9 测试（1）、（2）、（3）、（4）、（5）、（6）

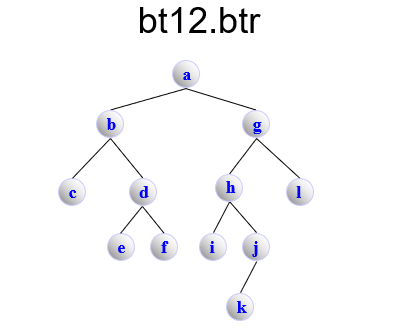


图10 测试（1）、（2）、（3）、（4）、（5）、（6）

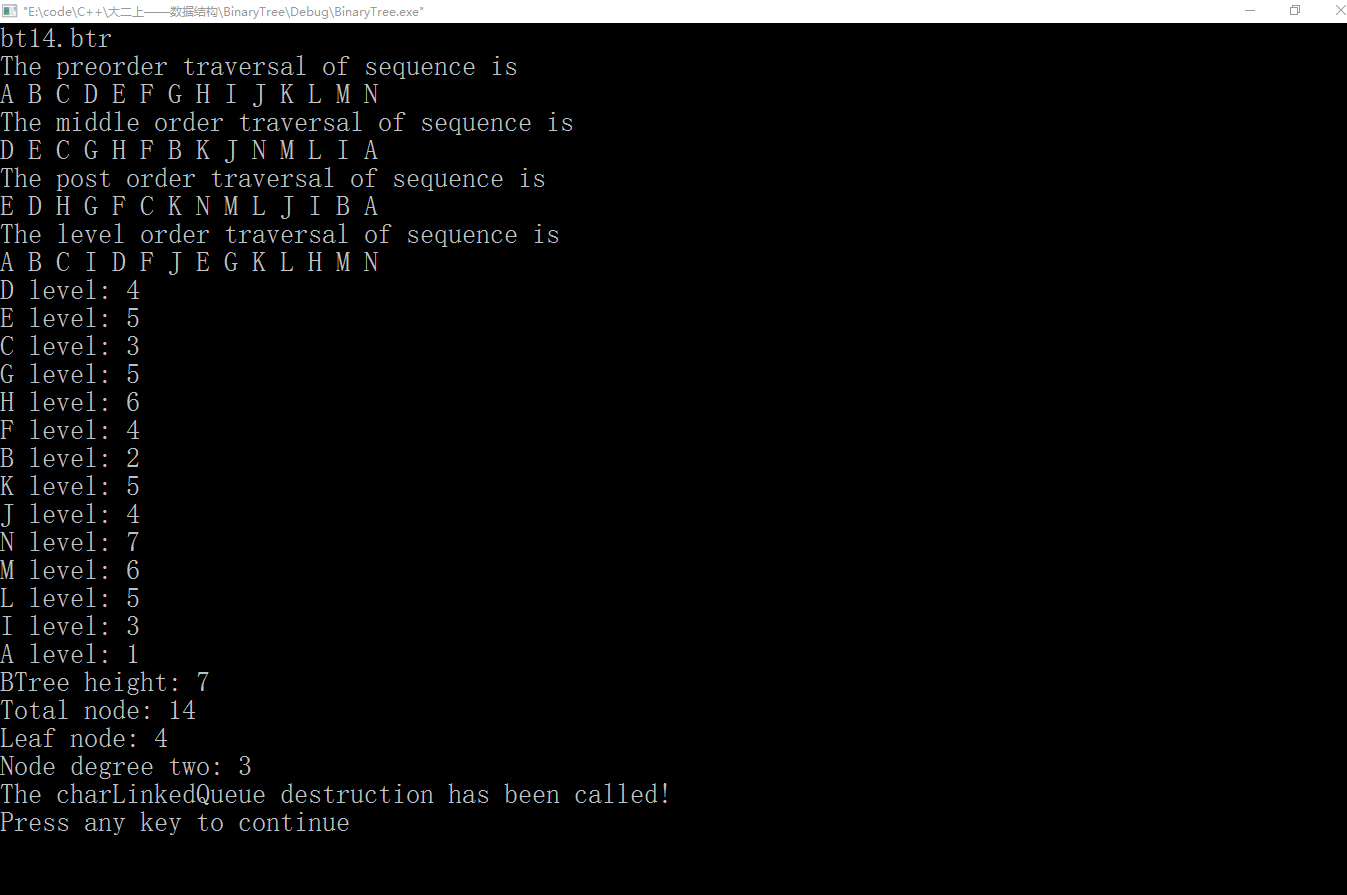


图11 测试（1）、（2）、（3）、（4）、（5）、（6）

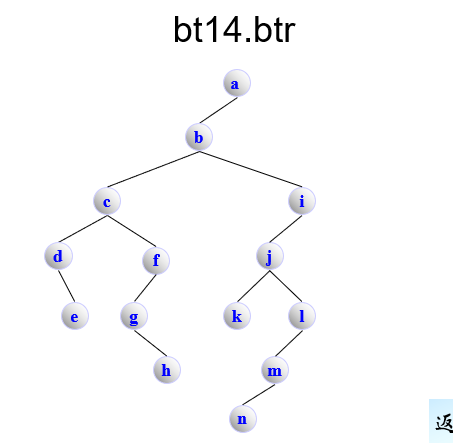


图12 测试（1）、（2）、（3）、（4）、（5）、（6）

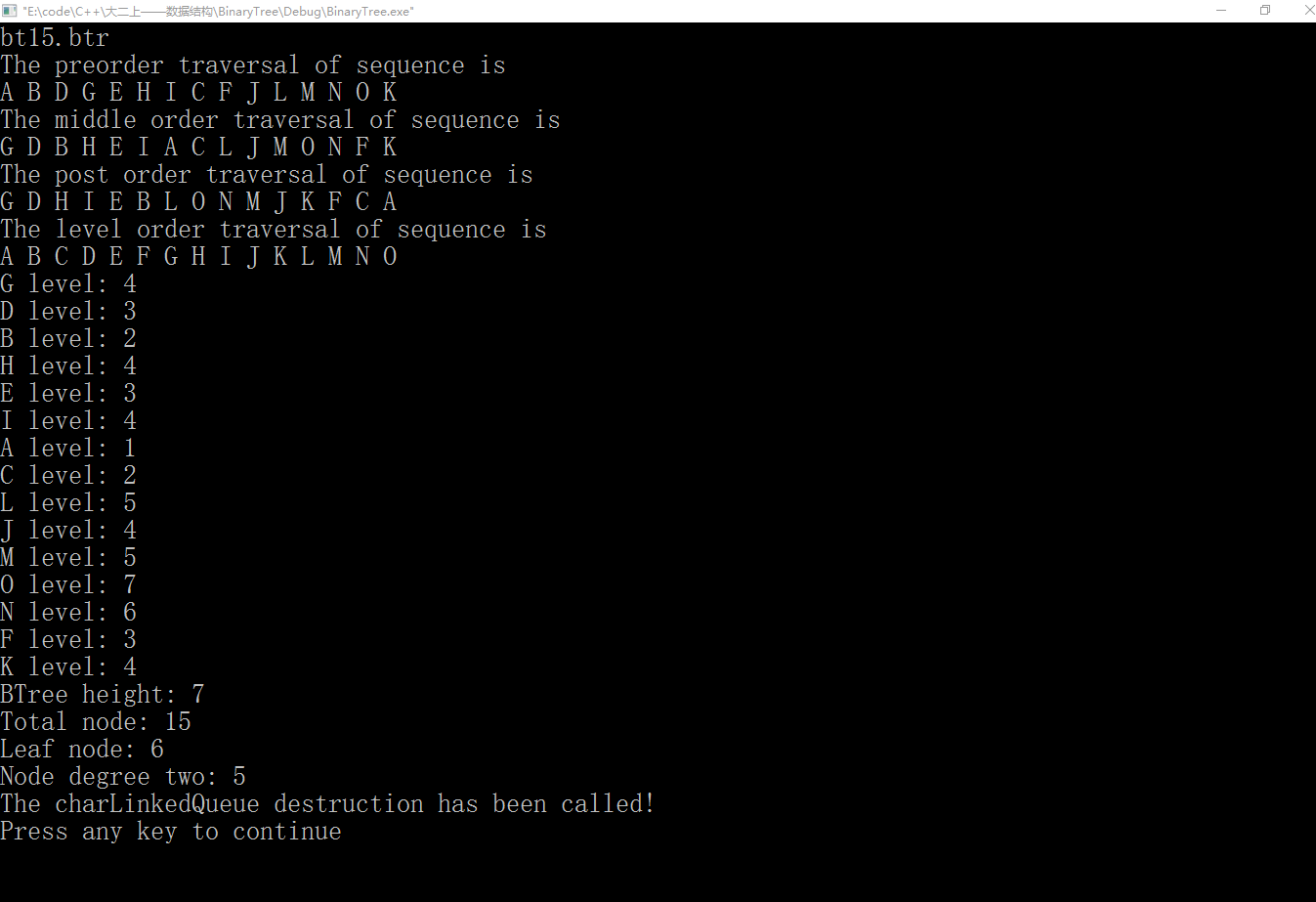


图13 测试（1）、（2）、（3）、（4）、（5）、（6）

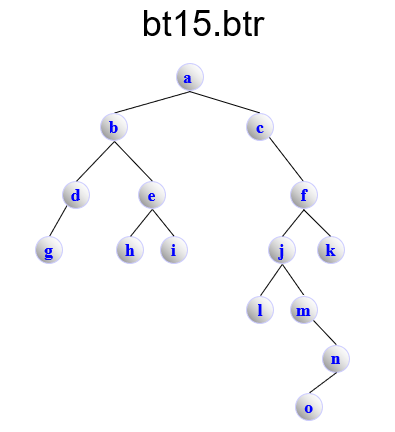


图14 测试（1）、（2）、（3）、（4）、（5）、（6）

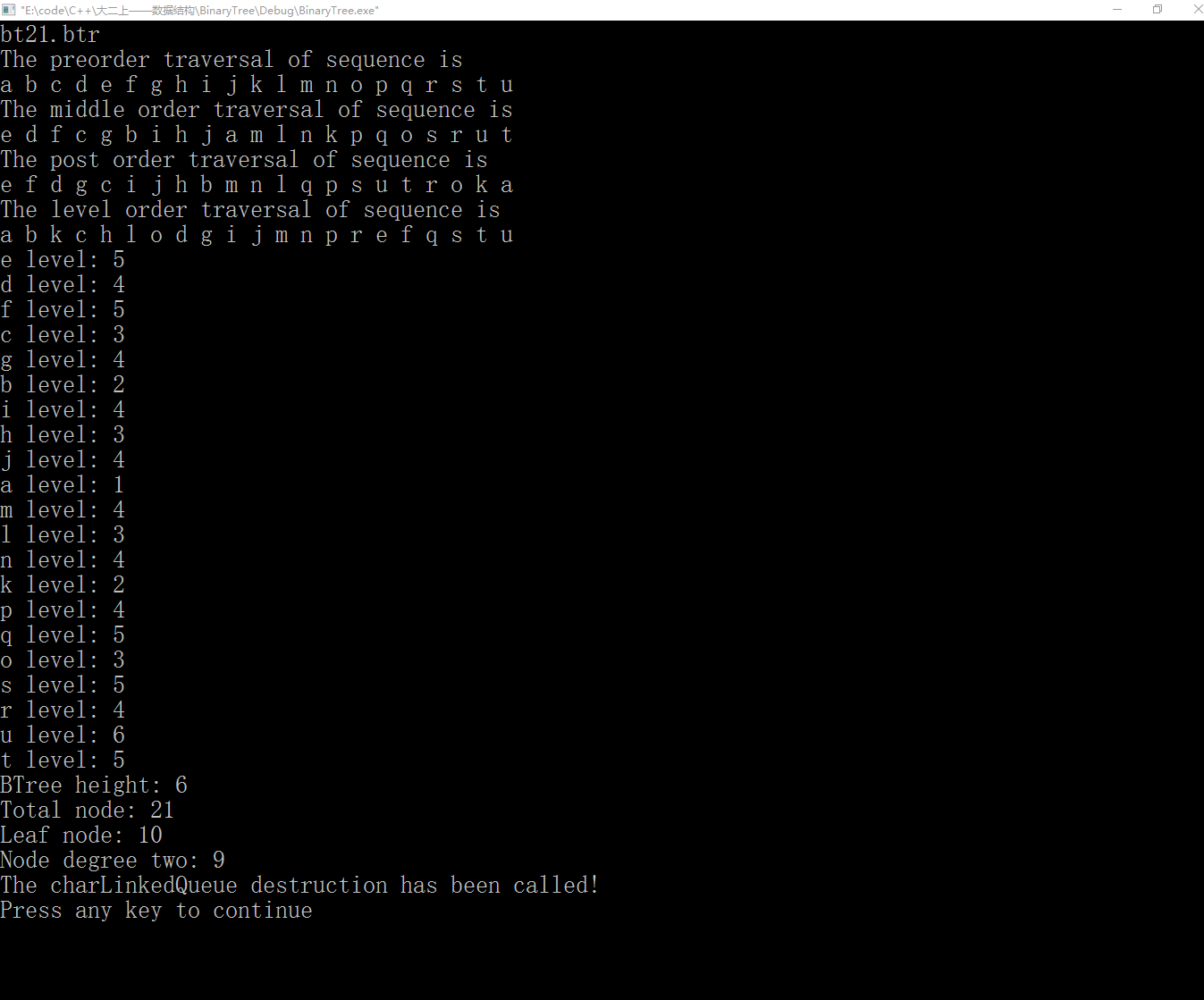


图15 测试（1）、（2）、（3）、（4）、（5）、（6）

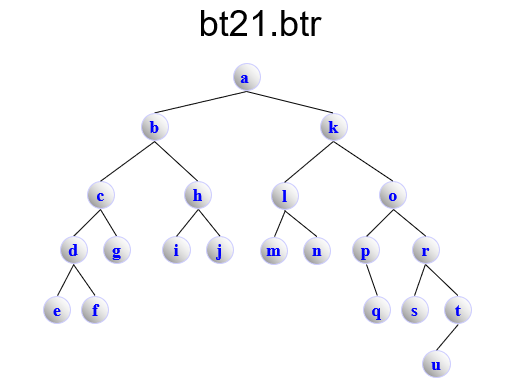


图16 测试（1）、（2）、（3）、（4）、（5）、（6）

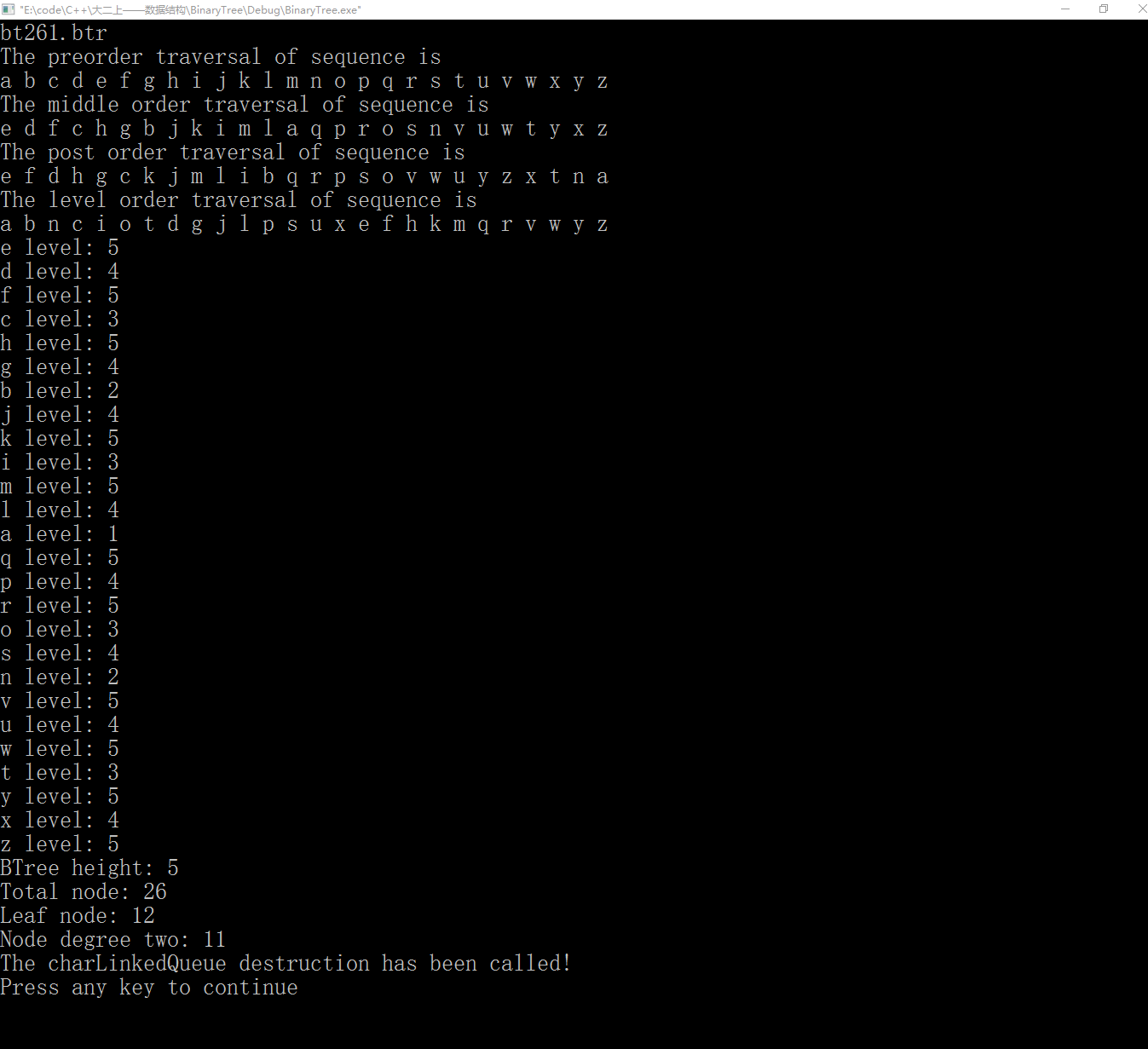


图17 测试（1）、（2）、（3）、（4）、（5）、（6）

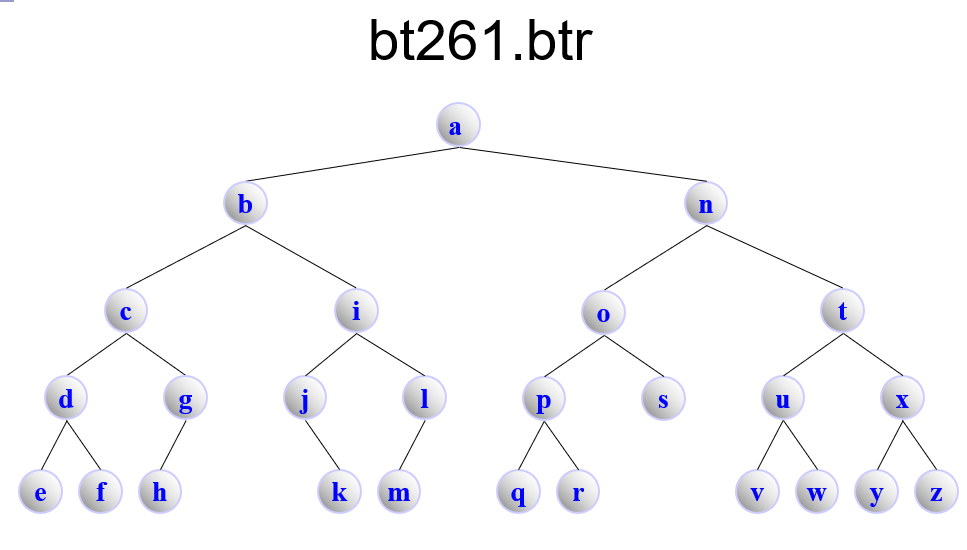


图18 测试（1）、（2）、（3）、（4）、（5）、（6）

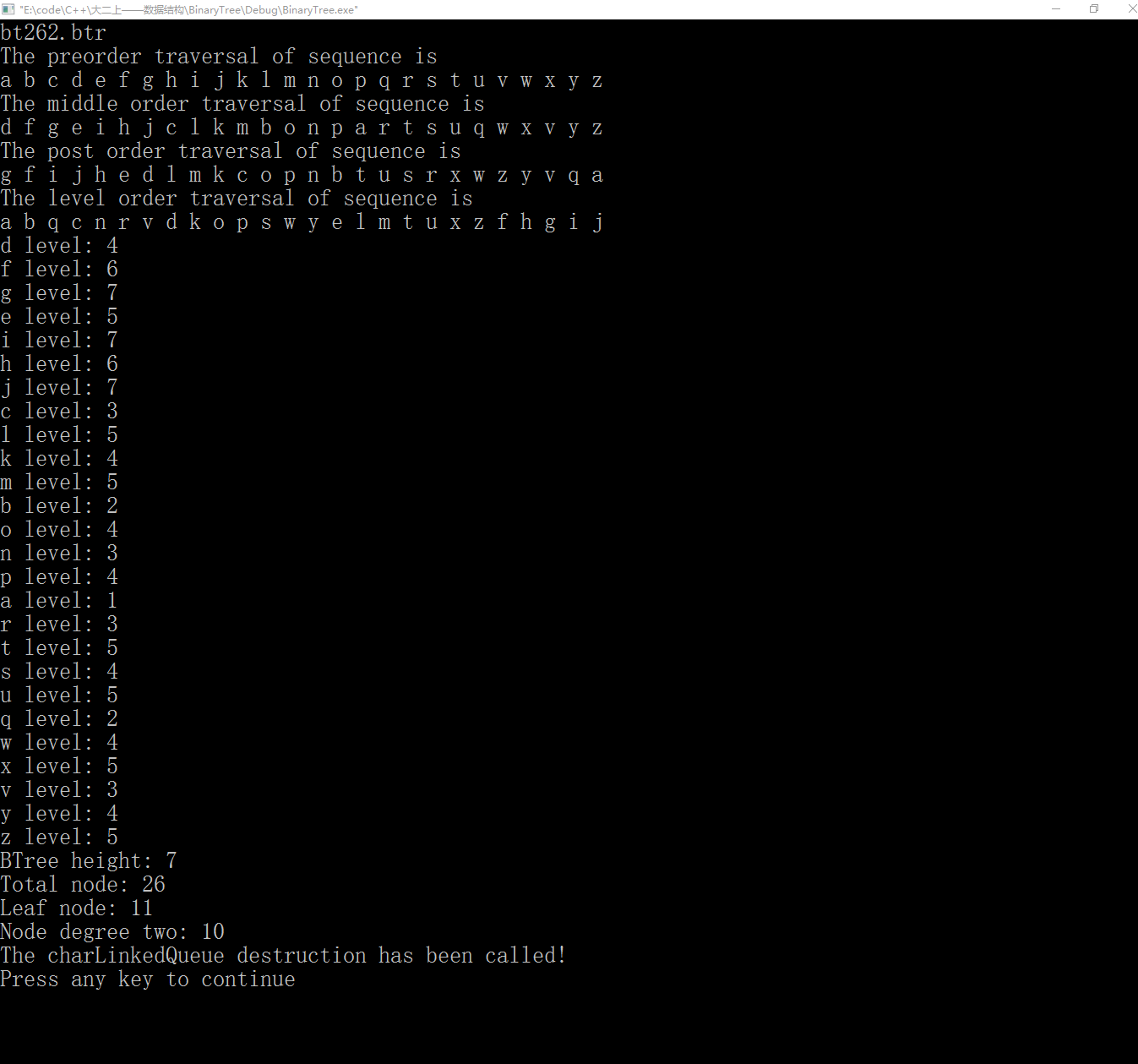


图19 测试（1）、（2）、（3）、（4）、（5）、（6）

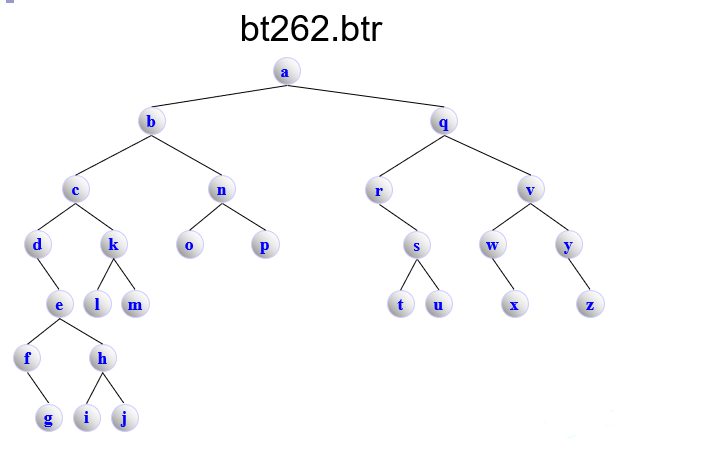


图20 测试（1）、（2）、（3）、（4）、（5）、（6）

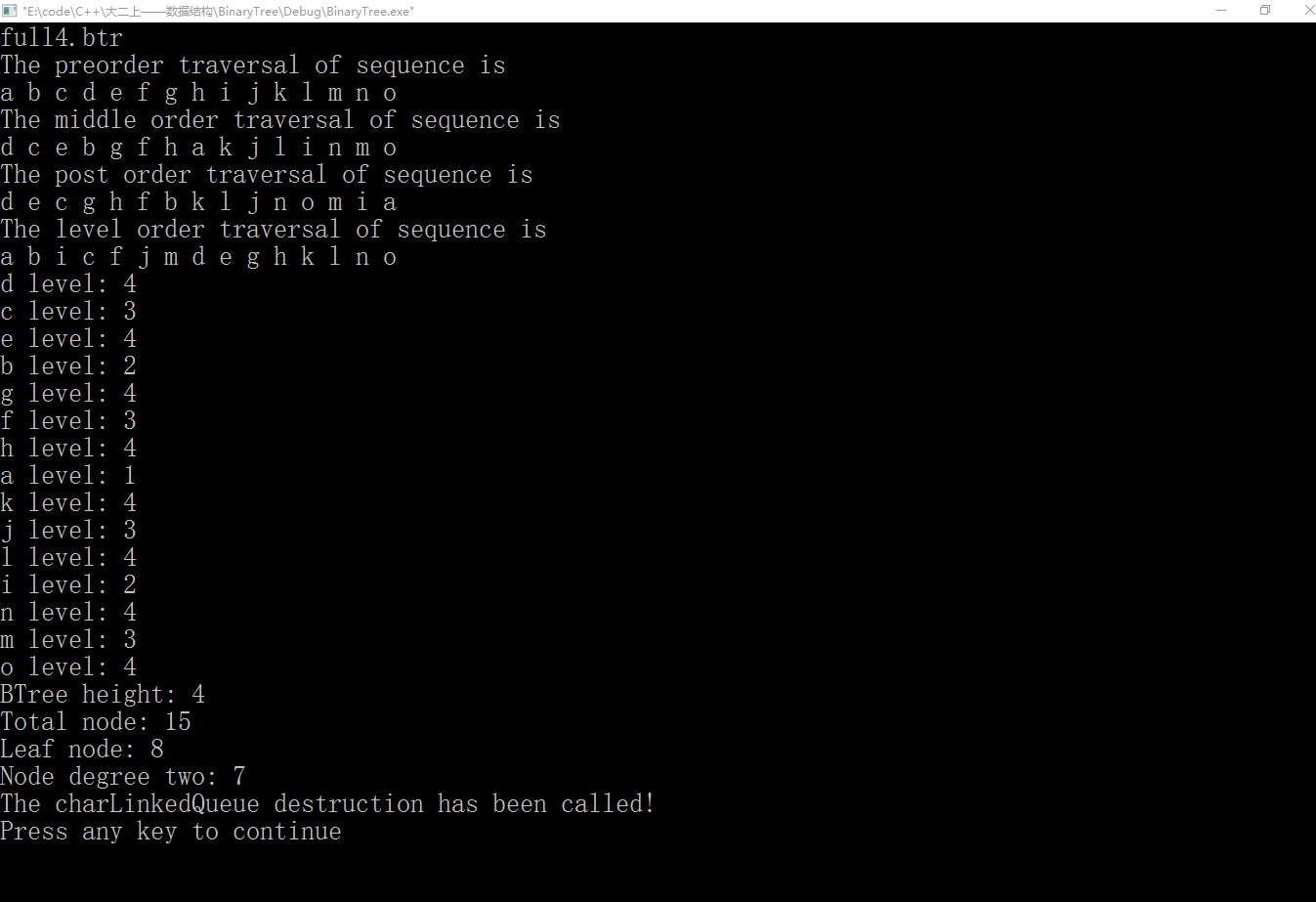


图21 测试（1）、（2）、（3）、（4）、（5）、（6）

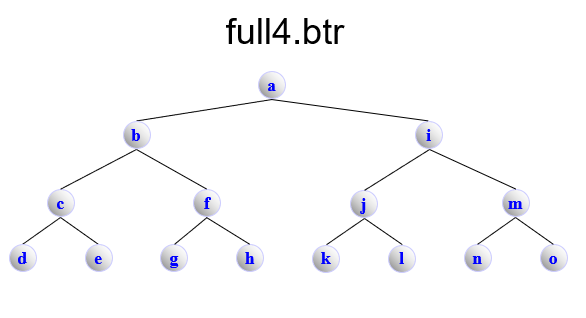


图22 测试（1）、（2）、（3）、（4）、（5）、（6）

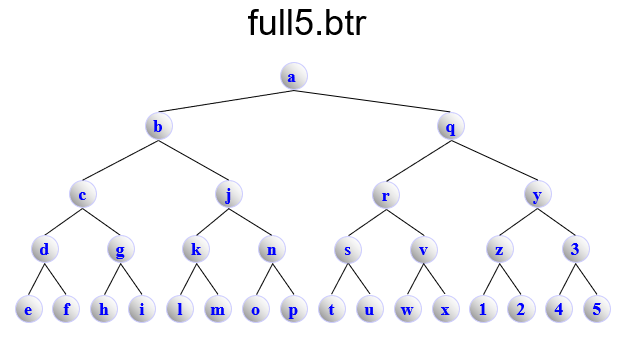


图23 测试（1）、（2）、（3）、（4）、（5）、（6）

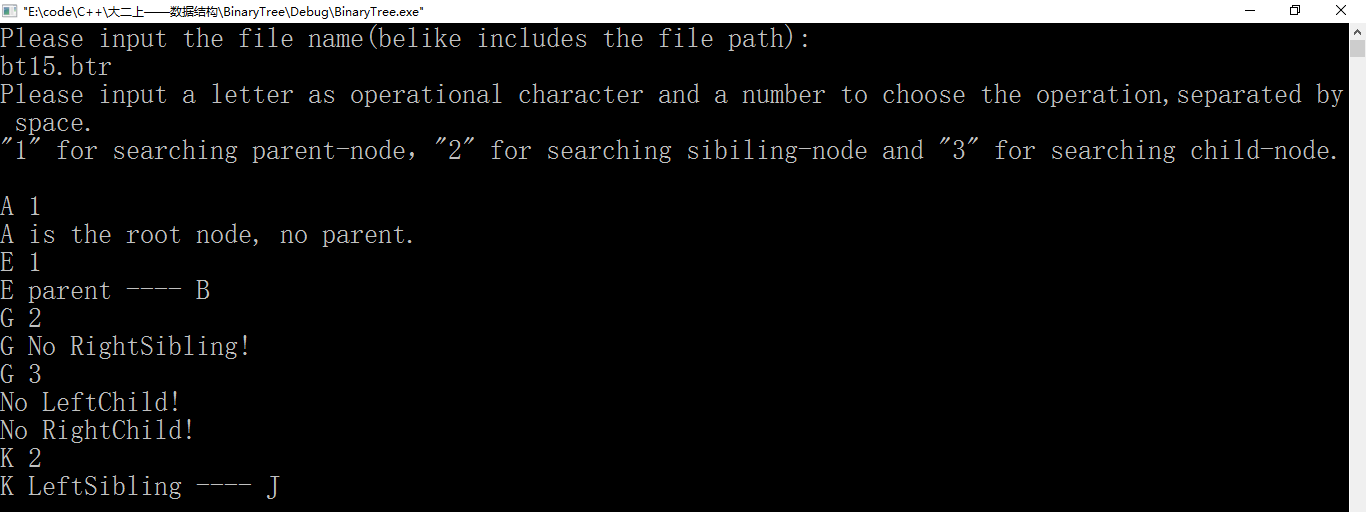


图24 测试（7）



图25 测试（7）

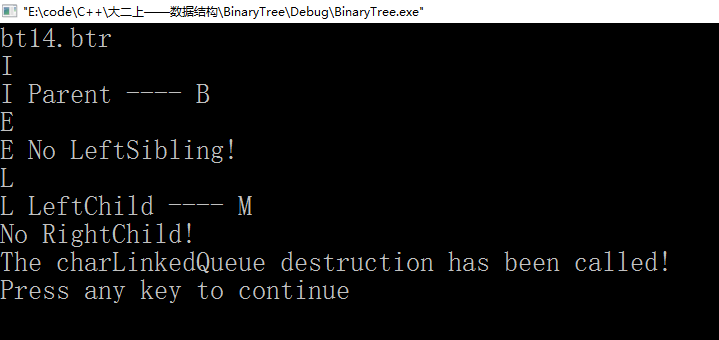


图26 测试（7）

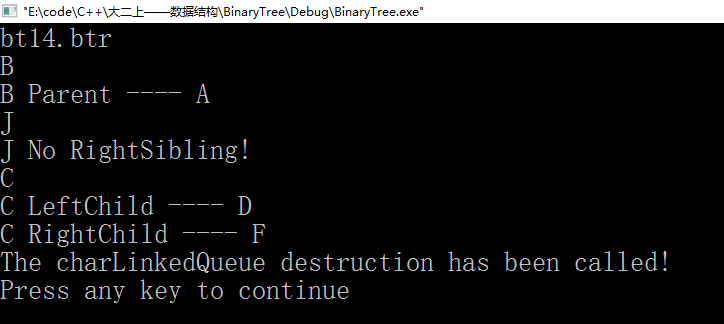


图27 测试（7）

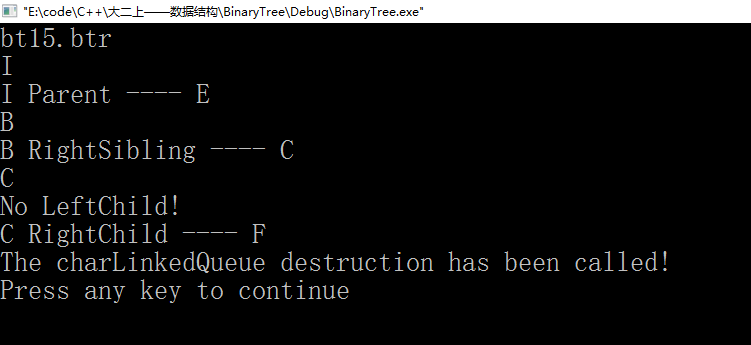


图28 测试（7）

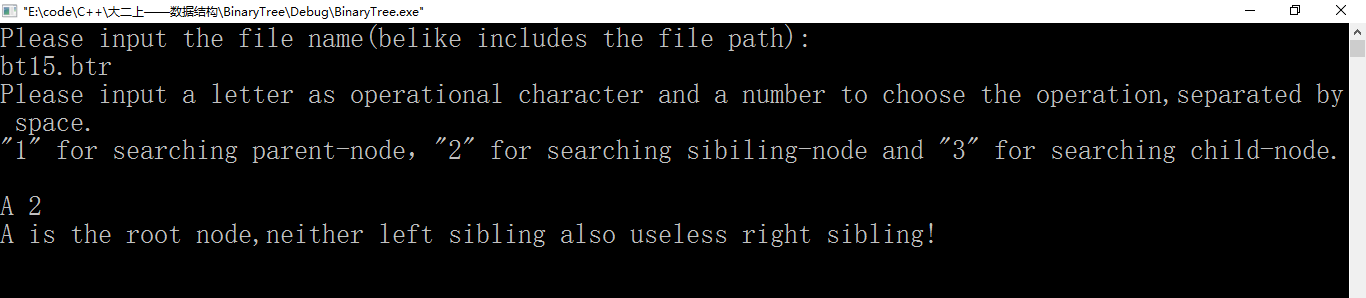


图29 测试（7）

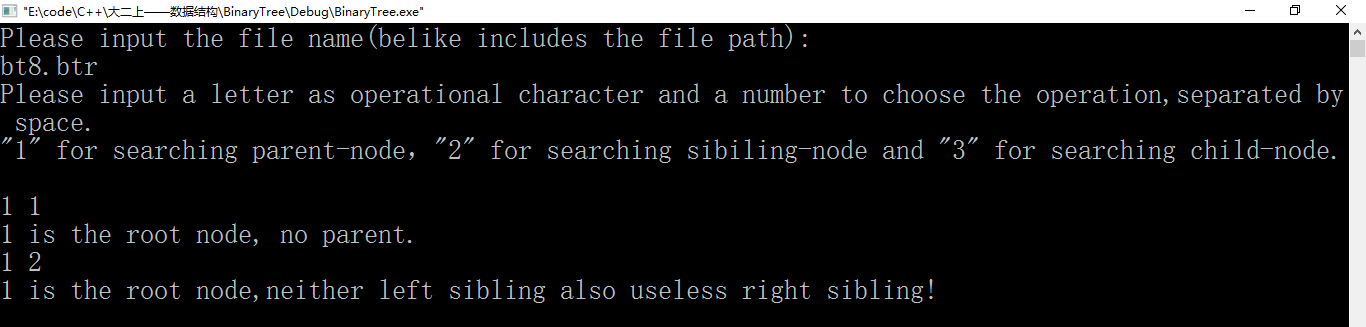


图30 测试（7）

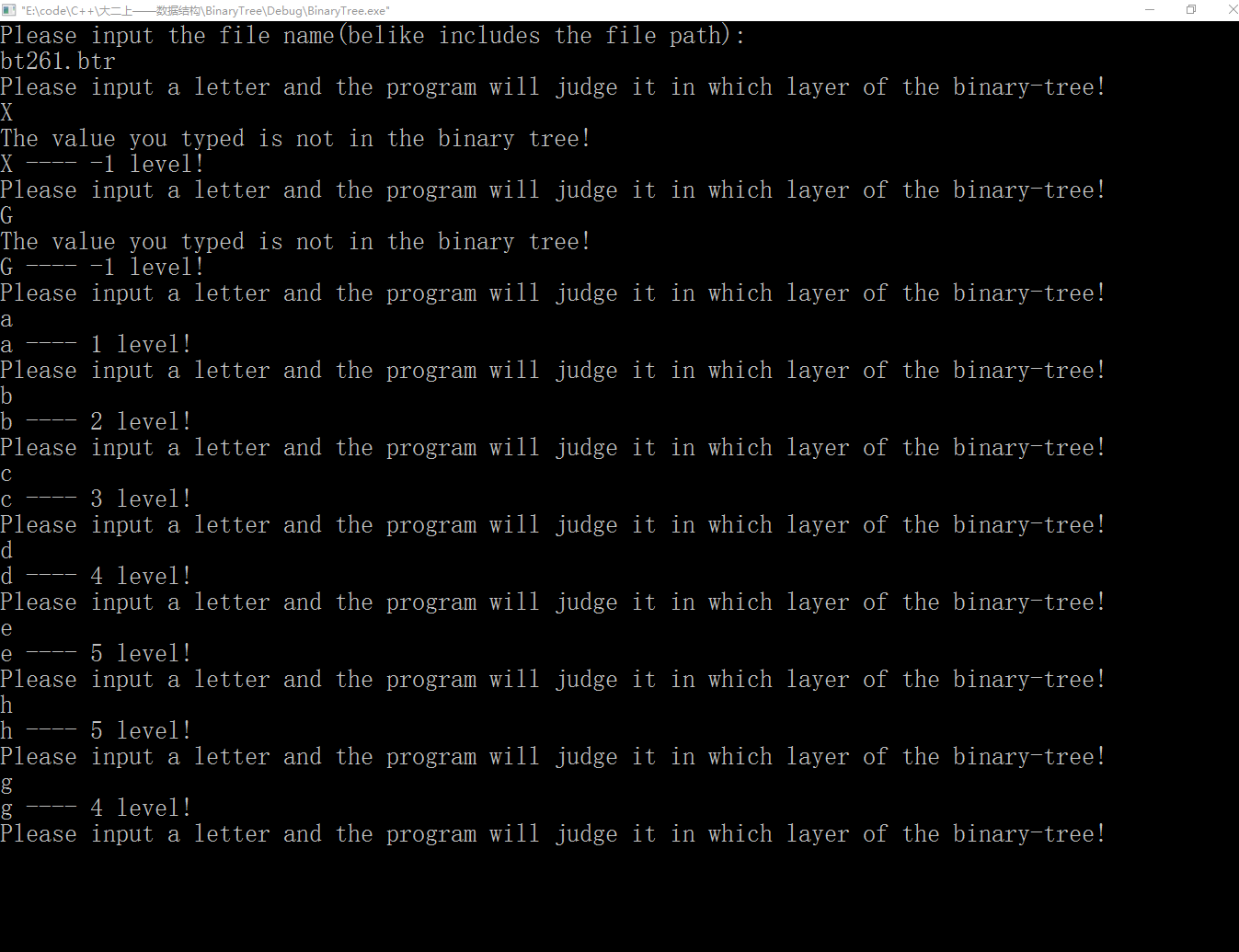


图31 测试（8）

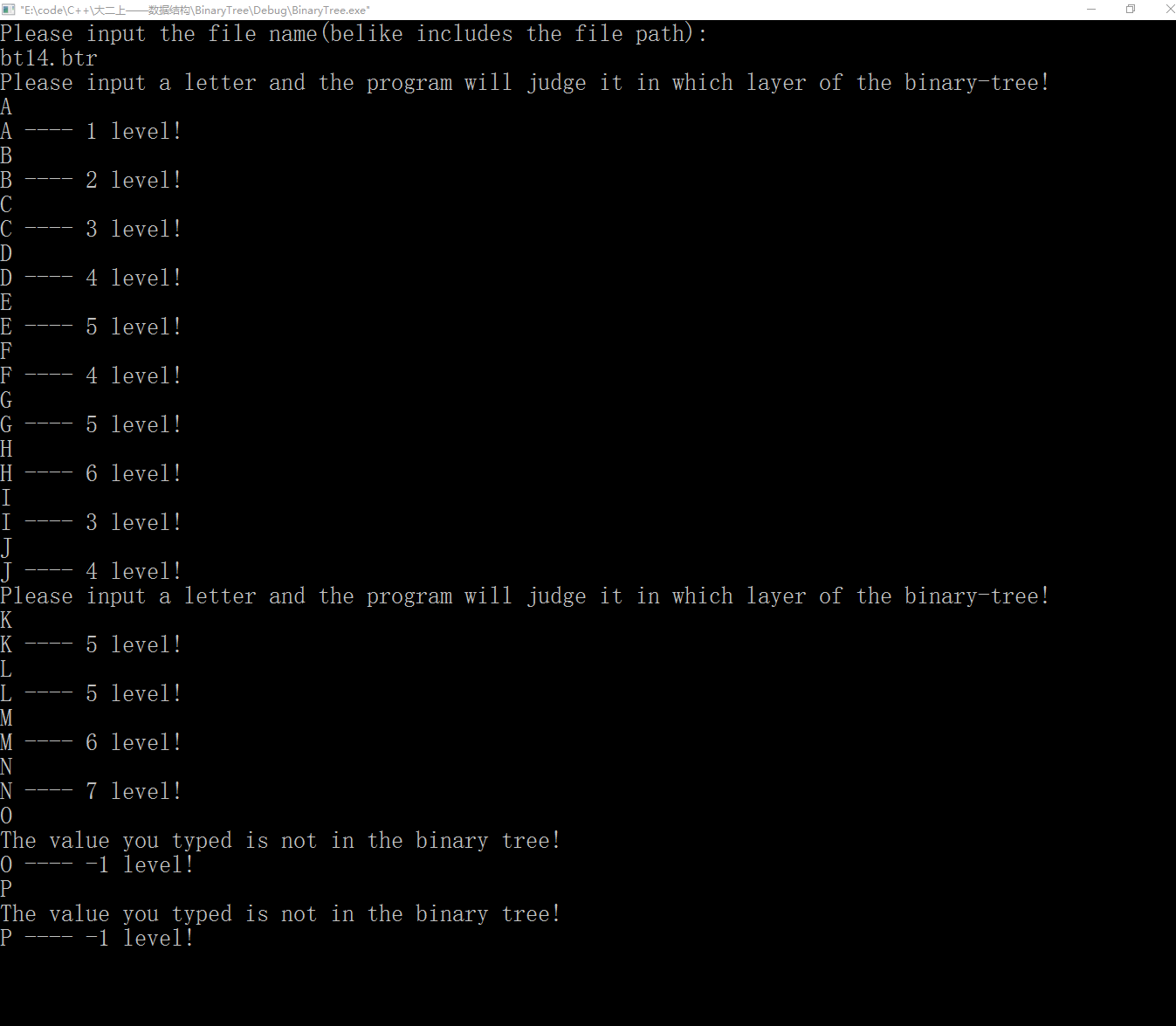


图32 测试（8）

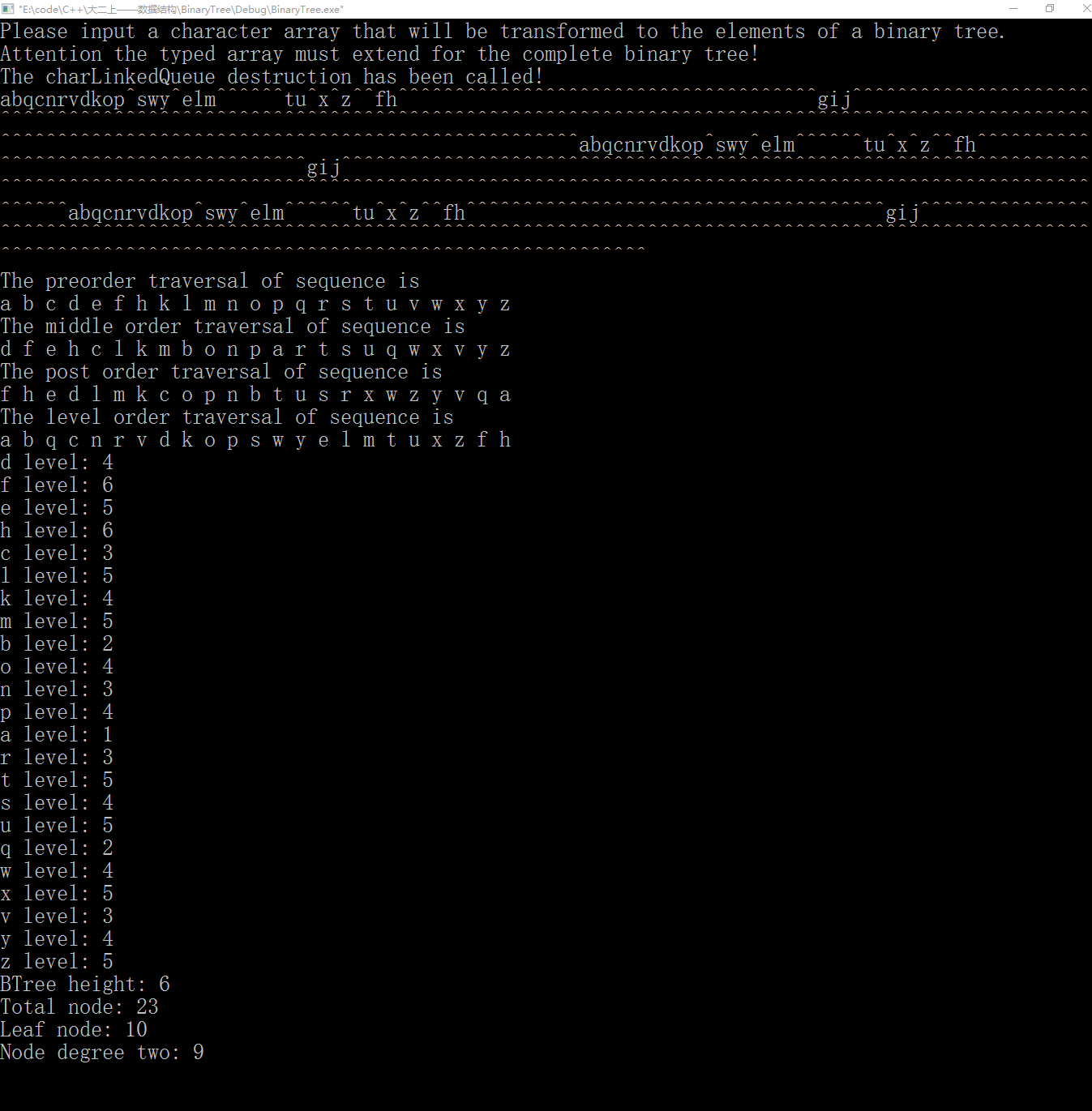


图33 测试（9）

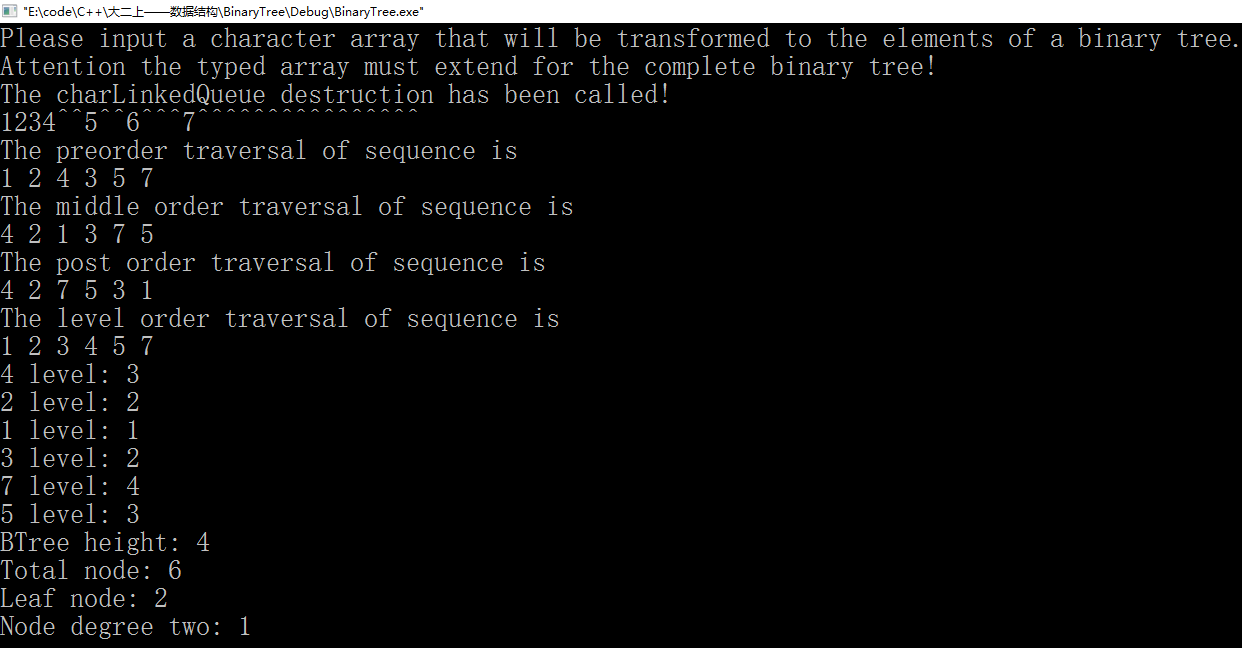


图34 测试（9）

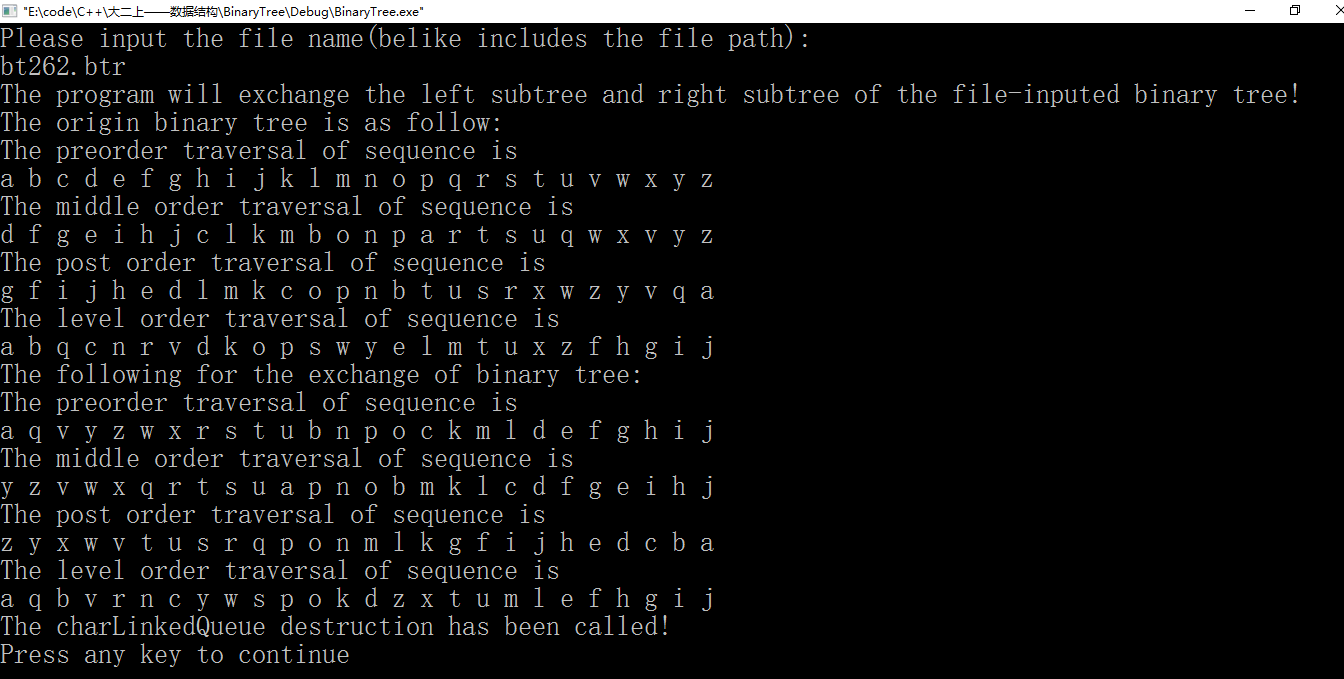


图35 测试（10）

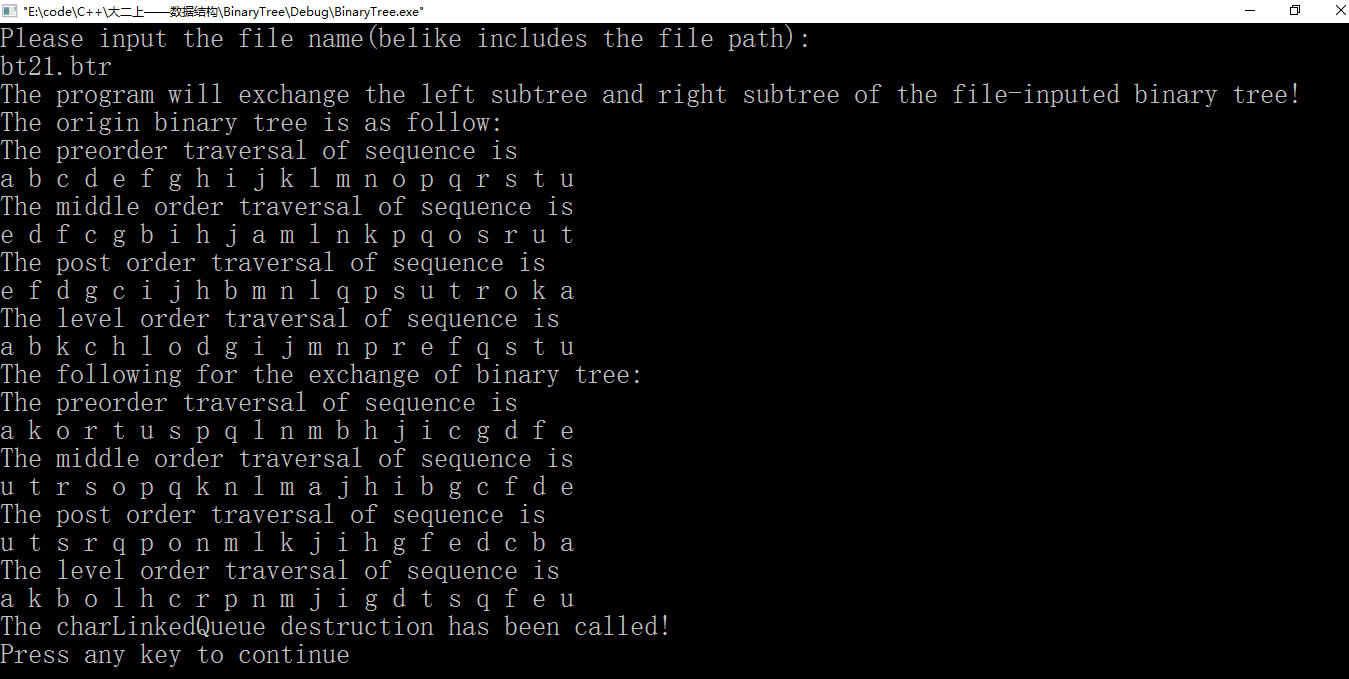


图36 测试（10）

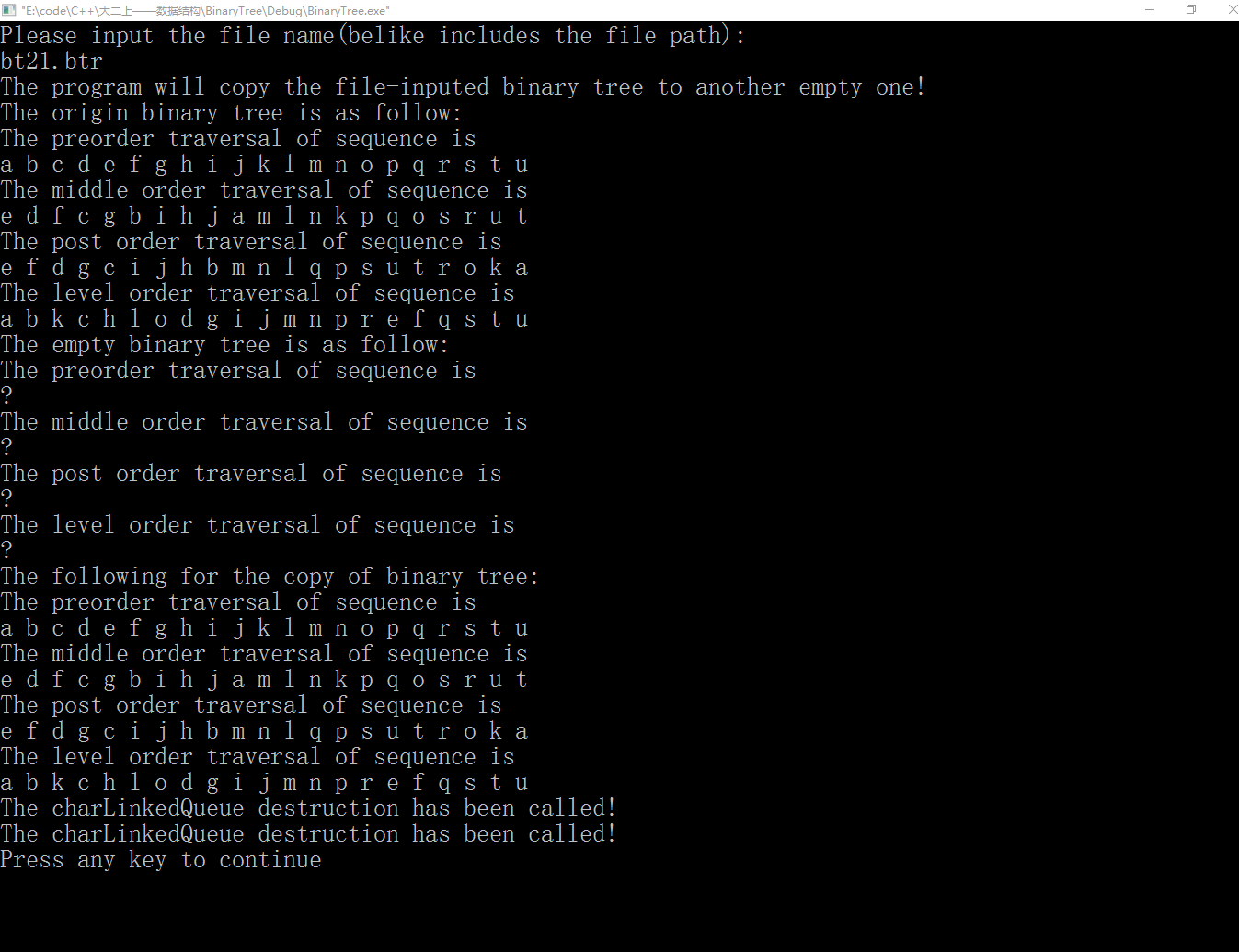


图37 测试（11）

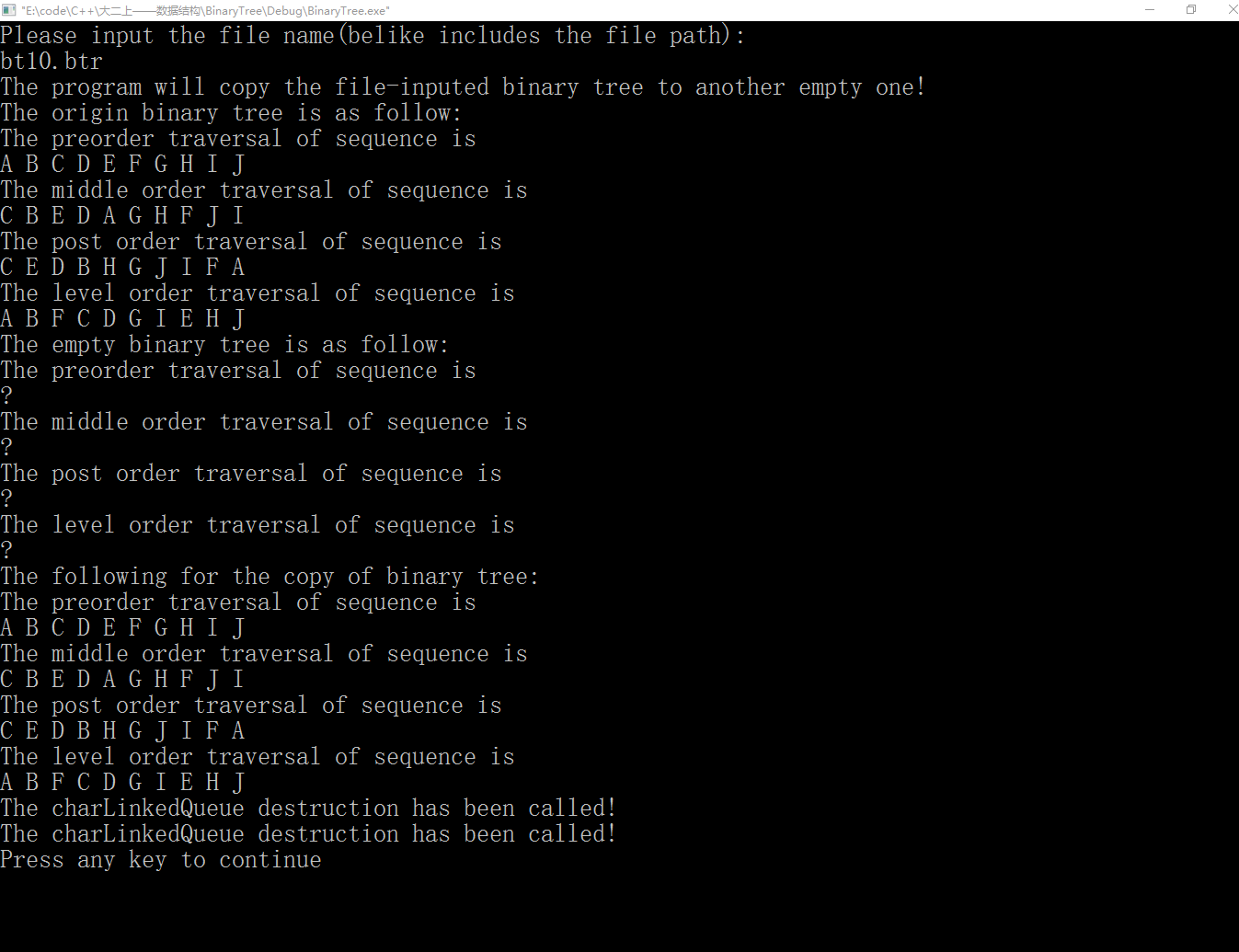


图38 测试（11）

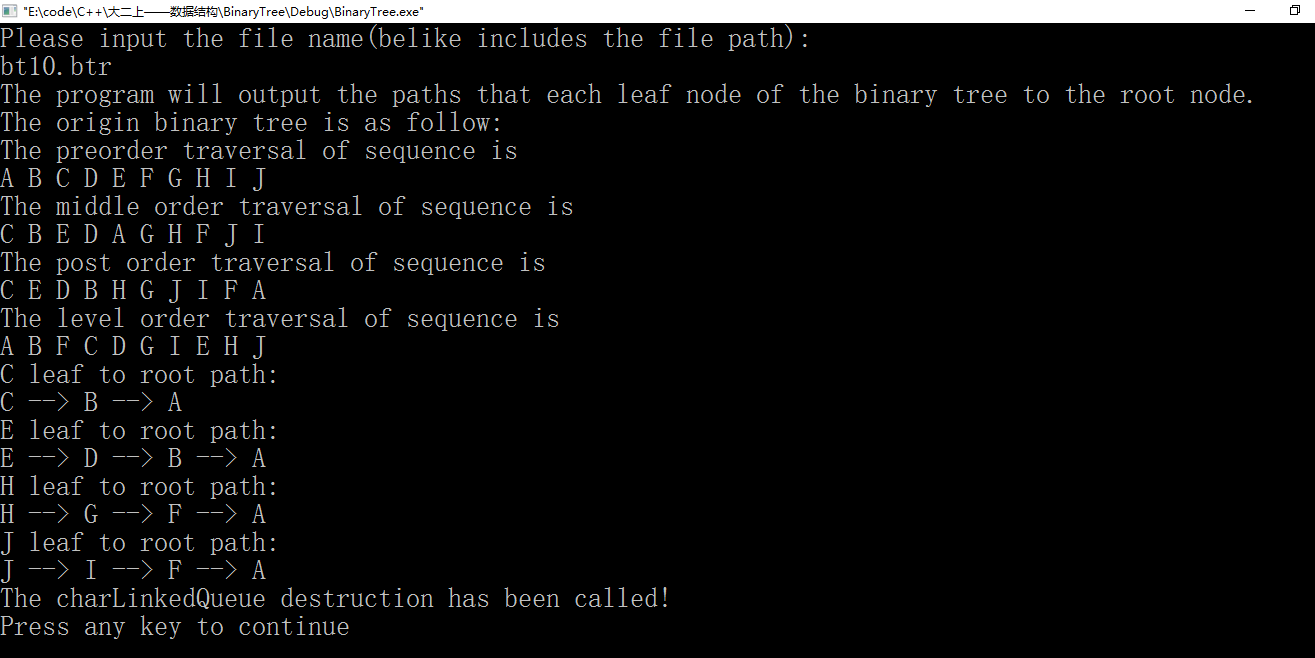


图39 测试（12）

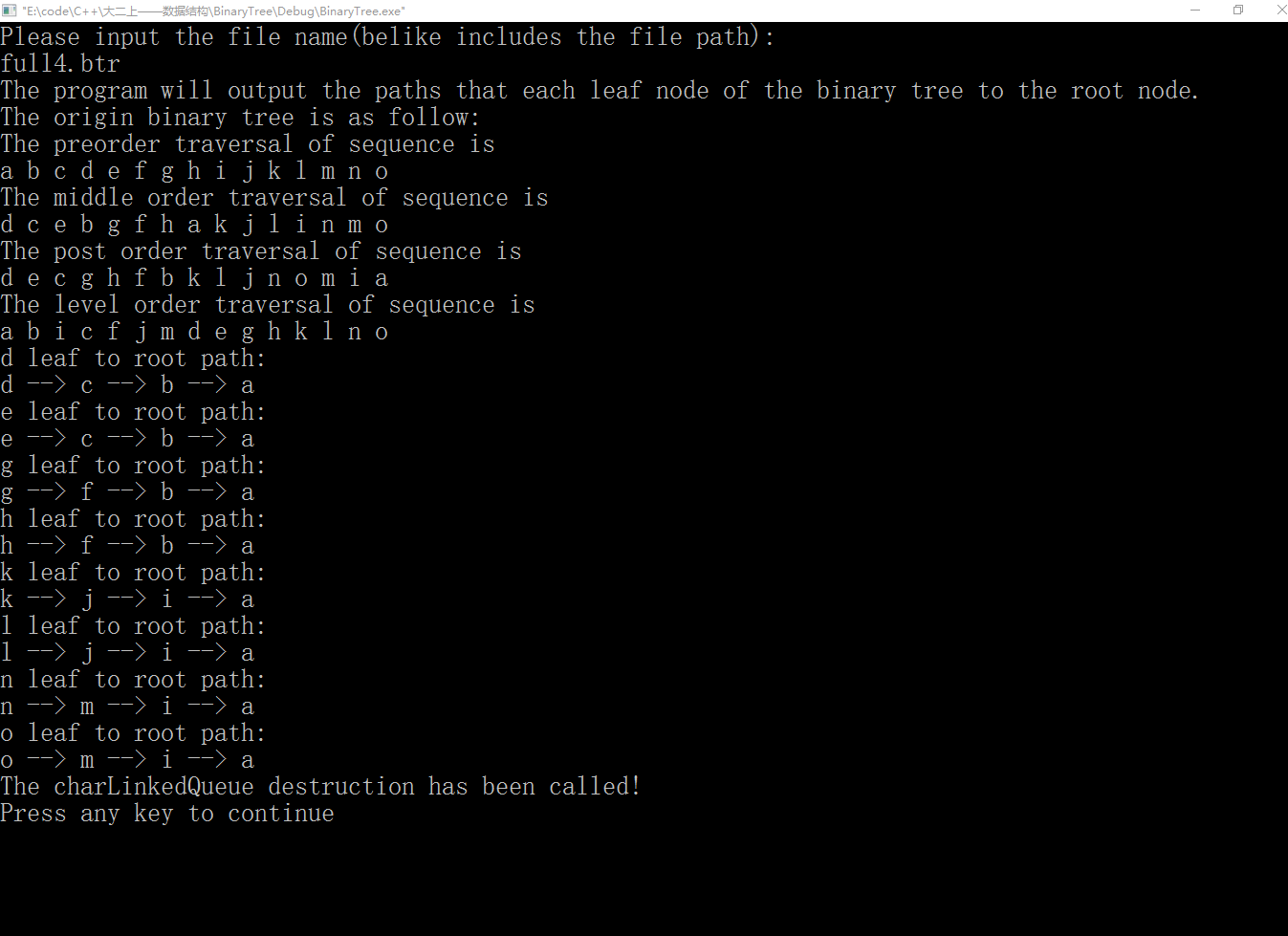


图40 测试（12）

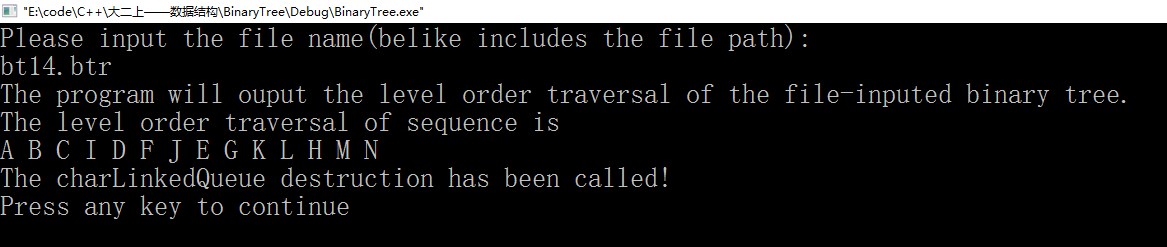


图41 测试（13）

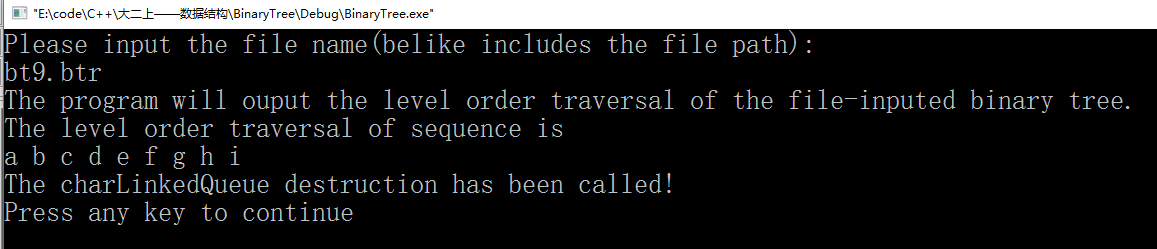


图42 测试（13）

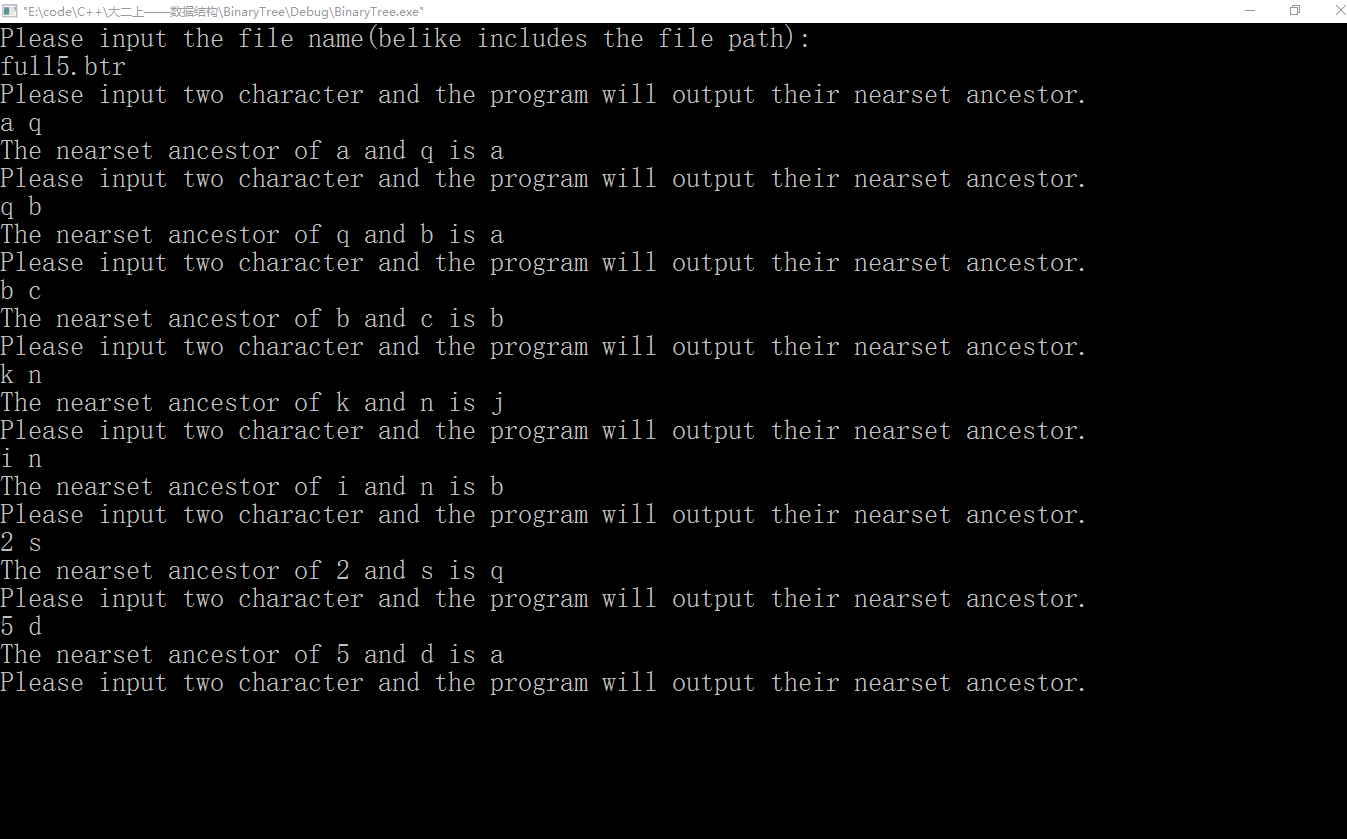


图43 测试（14）

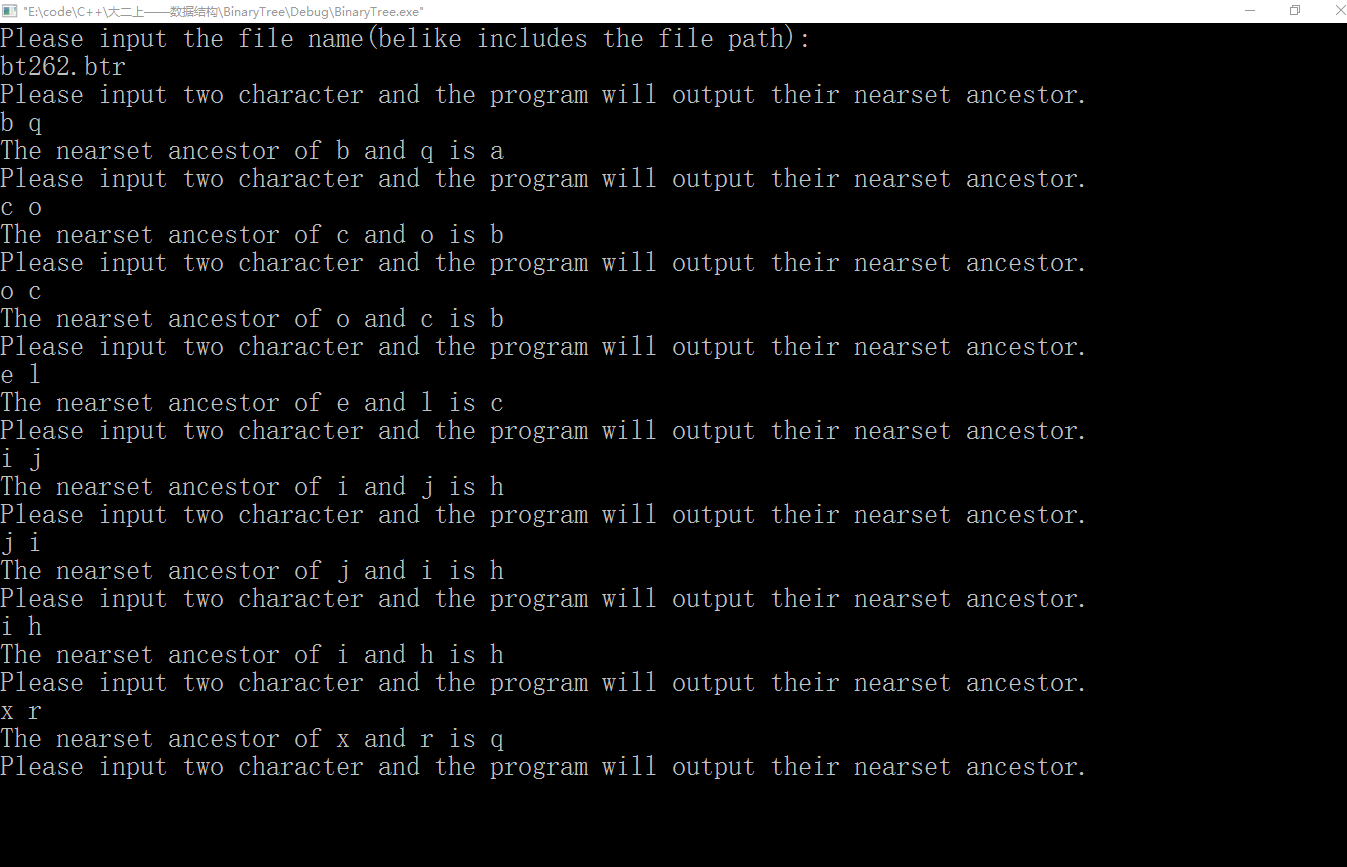


图44 测试（14）

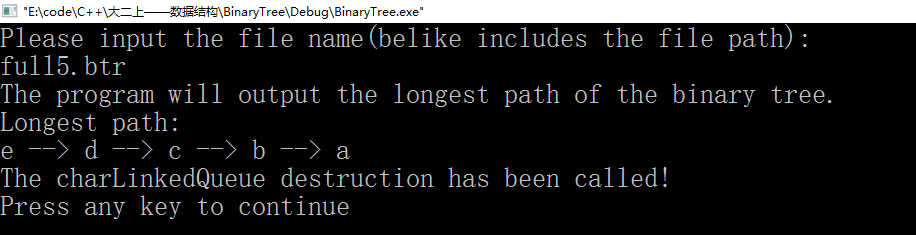


图45 测试（15）

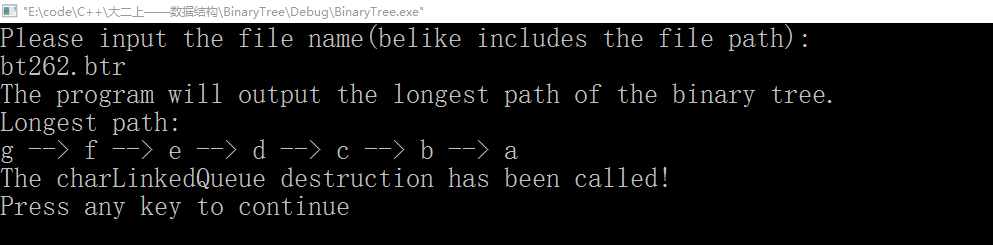


图46 测试（15）

**6.5 附源代码**

1. // stdafx.h : include file for standard system include files,
2. //  or project specific include files that are used frequently, but
3. //      are changed infrequently
4. //
6. #if !defined(AFX\_STDAFX\_H\_\_02F8C78B\_9F6E\_45FF\_BFCE\_7F99B5AC9359\_\_INCLUDED\_)
7. #define AFX\_STDAFX\_H\_\_02F8C78B\_9F6E\_45FF\_BFCE\_7F99B5AC9359\_\_INCLUDED\_
9. #if \_MSC\_VER > 1000
10. #pragma once
11. #endif // \_MSC\_VER > 1000
13. #include <stdc++.h>
14. #include <windows.h>
16. **using** **namespace** std;
18. **typedef** **char** elementType;
19. **typedef** **int** elementType1;
21. **typedef** **struct** node
22. {
23. elementType data;//刚开始应该写成将data写成string或者直接将整个函数写成模板的，写完了最后测试时
24. //才发现现在的写法有诸多不便；但修改的话就又要重构一遍，懒得整了。
25. **struct** node \*leftChild, \*rightChild;
26. }bitNode, \*binTree;
28. **typedef** **struct** charNode
29. {
30. //elementType data;
31. bitNode \*data;//the type must be bitNode\*
32. **struct** charNode \*link;
33. }CLNode, \*CPNode;

36. //typedef struct charNode
37. //{
38. //elementType data;
39. //struct charNode \*leftChild, \*rightChild;
40. //}charBitNode, \*charBinTree;
42. // TODO: reference additional headers your program requires here
44. //{{AFX\_INSERT\_LOCATION}}
45. // Microsoft Visual C++ will insert additional declarations immediately before the previous line.
47. #endif // !defined(AFX\_STDAFX\_H\_\_02F8C78B\_9F6E\_45FF\_BFCE\_7F99B5AC9359\_\_INCLUDED\_)

50. // charLinkedQueue.h: interface for the charLinkedQueue class.
51. //
52. //////////////////////////////////////////////////////////////////////
54. #if !defined(AFX\_CHARLINKEDQUEUE\_H\_\_13C2F642\_81C0\_4489\_9CF2\_3D58D8B48EA9\_\_INCLUDED\_)
55. #define AFX\_CHARLINKEDQUEUE\_H\_\_13C2F642\_81C0\_4489\_9CF2\_3D58D8B48EA9\_\_INCLUDED\_
57. #if \_MSC\_VER > 1000
58. #pragma once
59. #endif // \_MSC\_VER > 1000
61. //刚开始尝试写英文注释的，后面知难而退了；不过原来的英文注释我保留了
63. **class** charLinkedQueue
64. {
65. **public**:
66. charLinkedQueue();
67. **virtual** ~charLinkedQueue();
68. **bool** emptyCharLinkedQueue();
69. //bool fullSeqCircleQueue();
70. **bool** enQueue( bitNode \*value );//the type must be bitNode\*
71. **bool** deQueue( /\*bitNode \*value\*/ );
72. **bool** getFront( bitNode \*&value );//the type must be bitNode\*&
73. **int** length();
74. **friend** ostream &operator<<( ostream &os, charLinkedQueue &clq )
75. {
76. /\*
77. if( ( scq.\_front - 1 ) % maxn == scq.\_rear )
78. return os;
79. int column  = 0;
80. for( int i = scq.\_front; i % maxn != scq.\_rear; i = ( i + 1 ) % maxn )
81. {
82. os << setw(3) << setiosflags(ios::left) << scq.data[i] << " ";
83. column ++;
84. if( column % 10 == 0 )
85. os << endl;
86. }
87. os << endl;
88. \*/
89. **if**( clq.\_front == NULL )
90. **return** os;
91. CLNode \*tmp = clq.\_front;
92. **int** column = 0;
93. **while**( tmp != clq.\_rear->link )
94. {
95. os << setw(4) << setiosflags(ios::left) << tmp->data << " ";
96. column ++;
97. tmp = tmp->link;
98. **if**( column % 10 == 0 )
99. os << endl;
100. }
101. os << endl;
102. }
103. //为了能顺利使用原来的这个代码块来进行二叉树的层次便利，我主要的精力都放在\_front、\_rear类型、
104. //deQueue()、enQueue()、charNode的类型确定上，经过无数次尝试，总算结果对了----
105. //如果有Git，看了这个代码的每个版本你就会知道我付出了多少心血。。。。
106. **private**:
107. CLNode \*\_front;//the type must be CLNode\*
108. CLNode \*\_rear;//the type must be CLNode\*
109. };
111. #endif // !defined(AFX\_CHARLINKEDQUEUE\_H\_\_13C2F642\_81C0\_4489\_9CF2\_3D58D8B48EA9\_\_INCLUDED\_)


115. // \_Binary\_Tree.h: interface for the \_Binary\_Tree class.
116. //
117. //////////////////////////////////////////////////////////////////////
119. #if !defined(AFX\_\_BINARY\_TREE\_H\_\_9381B15F\_E185\_4489\_9415\_360A22C0A4E2\_\_INCLUDED\_)
120. #define AFX\_\_BINARY\_TREE\_H\_\_9381B15F\_E185\_4489\_9415\_360A22C0A4E2\_\_INCLUDED\_
122. #if \_MSC\_VER > 1000
123. #pragma once
124. #endif // \_MSC\_VER > 1000
126. #include "charLinkedQueue.h"
128. //刚开始尝试写英文注释的，后面知难而退了；不过原来的英文注释我保留了
130. **class** \_Binary\_Tree
131. {
132. **public**:
133. \_Binary\_Tree();//不带参数的构造函数
134. \_Binary\_Tree( elementType \*Arr );//带参数的构造函数
135. **void** build( elementType \*Arr );//从数组建立二叉树，相当于初始化；不带参数的构造函数无法适用于这里
136. **void** createNode( binTree BT, elementType \*Arr, **int** number );//根据从数组读到的数据先序递归建树
137. **virtual** ~\_Binary\_Tree();//析构函数
138. **bool** createBinaryTree( binTree &BT, elementType stringLine[100][3], **int** length, **int** &row );//根据文本数据
139. //先序构造二叉树
140. **bool** readFileToArray( elementType stringLine[100][3], **int** &length );//将文本数据读入二维数组中
141. **bool** emptyBinaryTree();//二叉树判空，仅适用于带参数构造函数建立的二叉树
142. **bool** \_exit( binTree BT, elementType value );//判断节点数据是否在二叉树中
143. binTree getNodePoint();//返回根节点地址
144. binTree getNodePoint( binTree BT, elementType value );//返回value在二叉树中的地址
145. binTree getParent( binTree BT, elementType value );//返回value的父母
146. **void** preOrderTraverse(binTree BT);//前序遍历
147. **void** inOrderTraverse(binTree BT);//中序遍历
148. **void** postOrderTraverse(binTree BT);//后序遍历
149. **void** levelOrderTraverse(binTree BT);//层次遍历
150. **void** destroy( binTree BT );//销毁二叉树
151. **void** level( binTree BT, **int** number );//求二叉树中各个节点的层次
152. **int** height( binTree BT );//求二叉树高度
153. **int** numberOfBTreeNode( binTree BT );//返回二叉树节点总数
154. **int** numberOfBTreeLeafNode( binTree BT, **int** &number );//返回二叉树叶节点个数
155. **void** numberOfNodeDegreeTwo( binTree BT, **int** &number );//求二叉树中度为2的节点个数
156. //void family( binTree BT, elementType1 number );
157. **void** getParent( binTree BT, elementType value, **bool** &flag );//求value的父节点
158. **void** getSibling( binTree BT, elementType value, **bool** &flag );//when call the function, the parameter flag
159. //must be assigned for false
160. //求value的兄弟节点，法1；有一个bug
161. **void** getSibling( binTree BT, elementType value );//求value的兄弟节点，法2
162. **void** getChild( binTree BT, elementType value, **bool** &flag );//求value孩子节点
163. **int** levelJudge( binTree BT, elementType value, **int** &number, **int** level );//返回value节点的层次
164. **void** exchangeLeftAndRightSibling( binTree BT );//交换左右子树
165. **void** copyBTree( binTree BT1, binTree BT );//复制二叉树
166. charLinkedQueue clq;//包含
167. **void** allLeafToRootPath( binTree BT, elementType \*path, **int** &pathLength );//求所有叶节点到根节点路径
168. **void** binaryTreeLongestPath( binTree BT, elementType \*path, **int** &pathLength,
169. elementType \*longestPath, **int** &longestLength );//求叶节点到根节点的最长路径
170. binTree nearestAncestor( binTree BT, bitNode \*BNode1, bitNode \*BNode2 );//求两个节点的最近祖先
171. //本来打算用elementType数据
172. //作为参数的，后面发现行不通
173. //可能是我太菜了吧
174. **private**:
175. bitNode \*BTree;
177. };
179. #endif // !defined(AFX\_\_BINARY\_TREE\_H\_\_9381B15F\_E185\_4489\_9415\_360A22C0A4E2\_\_INCLUDED\_)


183. // charLinkedQueue.cpp: implementation of the charLinkedQueue class.
184. //
185. //////////////////////////////////////////////////////////////////////
187. #include "stdafx.h"
188. #include "charLinkedQueue.h"
190. //////////////////////////////////////////////////////////////////////
191. // Construction/Destruction
192. //////////////////////////////////////////////////////////////////////
194. charLinkedQueue::charLinkedQueue()
195. {
196. \_front = \_rear = NULL;
197. }
199. charLinkedQueue::~charLinkedQueue()
200. {
201. CLNode \*tmp = NULL;
202. **while**( \_front != \_rear )
203. {
204. tmp = \_front;
205. \_front = \_front->link;
206. **delete** tmp;
207. }
208. cout << "The charLinkedQueue destruction has been called!" << endl;
209. }
211. **bool** charLinkedQueue::emptyCharLinkedQueue()
212. {
213. **return** \_front == NULL;
214. }
216. **bool** charLinkedQueue::enQueue( bitNode \*value )
217. {
218. CLNode \*newNode = **new** CLNode;
219. **if**( !newNode )
220. {
221. cerr << "Space allocating falied!Error in charLinkedQueue::enQueue()!" << endl;
222. **return** **false**;
223. }
224. newNode->data = value;
225. newNode->link = NULL;
226. **if**( emptyCharLinkedQueue() )
227. {
228. \_front = \_rear = newNode;
229. }
230. **else**
231. {
232. \_rear->link = newNode;
233. \_rear = newNode;
234. }
235. **return** **true**;
236. }
238. **bool** charLinkedQueue::deQueue( /\*elementType &value\*/ )
239. {
240. **if**( emptyCharLinkedQueue() )
241. {
242. cerr << "Node deleting falied!Error in charLinkedQueue::deQueue()!" << endl;
243. **return** **false**;
244. }
245. CLNode \*tmp = \_front;
246. //value = \_front->data;
247. \_front = \_front->link;
248. **delete** tmp;
249. **if**( \_front == NULL )
250. \_rear = NULL;
251. **return** **true**;
252. }
254. **bool** charLinkedQueue::getFront( bitNode \*&value )
255. {
256. **if**( emptyCharLinkedQueue() )
257. {
258. cerr << "Queue is empty!\nNode-data acquiring falied!Error in charLinkedQueue::deQueue()!" << endl;
259. **return** **false**;
260. }
261. value = \_front->data;//原来我是注释掉的，导致输出一直是A；
262. **return** **true**;
263. }
265. **int** charLinkedQueue::length()
266. {
267. **if**( emptyCharLinkedQueue() )
268. {
269. cerr << "Queue is empty!" << endl;
270. **return** -1;
271. }
272. CLNode \*tmp = \_front;
273. **int** \_size = 0;
274. **while**( tmp != NULL )
275. {
276. tmp = tmp->link;
277. \_size ++;
278. }
279. **return** \_size;
280. }

283. // \_Binary\_Tree.cpp: implementation of the \_Binary\_Tree class.
284. //
285. //////////////////////////////////////////////////////////////////////
287. #include "stdafx.h"
288. #include "\_Binary\_Tree.h"
289. #include "charLinkedQueue.h"
291. //////////////////////////////////////////////////////////////////////
292. // Construction/Destruction
293. //////////////////////////////////////////////////////////////////////

296. \_Binary\_Tree::\_Binary\_Tree()       //新建一个结点
297. {
298. //BTree = NULL;
299. BTree = **new** bitNode;
300. BTree->leftChild = BTree->rightChild = NULL;
301. }
303. \_Binary\_Tree::~\_Binary\_Tree()
304. {
305. destroy(BTree);//析构函数不能带参数，只能这么处理了
306. }
308. \_Binary\_Tree::\_Binary\_Tree( elementType \*Arr )
309. {
310. BTree = NULL;
311. build(Arr);
312. }
314. **void** \_Binary\_Tree::build( elementType \*Arr )
315. {
316. **if**(BTree)
317. destroy(BTree);
318. **if**( Arr[0] == '^' )
319. {
320. BTree = NULL;
321. **return**;
322. }
323. BTree = **new** bitNode;
324. BTree->leftChild = NULL;
325. BTree->rightChild = NULL;
326. BTree->data = Arr[0];
327. createNode( BTree, Arr, 0 );
328. }
330. **void** \_Binary\_Tree::createNode( binTree BT, elementType \*Arr, **int** number )
331. {
332. bitNode \*tmp = **new** bitNode;
333. **if**( Arr[ number \* 2 + 1 ] != '^' )
334. {
335. BT->leftChild =**new** bitNode ;
336. tmp = BT->leftChild;
337. tmp->data = Arr[ number \* 2 + 1 ];
338. tmp->leftChild = NULL;
339. tmp->rightChild = NULL;
340. createNode( tmp, Arr, number \* 2 + 1 );
341. }
342. **if**( Arr[ number \* 2 + 2 ] != '^' )
343. {
344. BT->rightChild =**new** bitNode ;
345. tmp = BT->rightChild;
346. tmp->data = Arr[ number \* 2 + 2 ];
347. tmp->leftChild = NULL;
348. tmp->rightChild = NULL;
349. createNode( tmp, Arr, number \* 2 + 2 );
350. }
351. }
353. **bool** \_Binary\_Tree::createBinaryTree( binTree &BT, elementType stringLine[100][3], **int** length, **int** &row )
354. {
355. **if** (row >= length || length == 0 )      //strlen存数据的二维数组，nRow结点所在的位数，nlen结点的个数
356. **return** **false**;
357. **if** ( row == 0 )
358. BT = BTree;
359. **else**
360. BT = **new** bitNode;//new下面是公用的，用if的目的是改变private里BTree里的值
361. BT->data = stringLine[row][0];
362. BT->leftChild = NULL;
363. BT->rightChild = NULL;
365. **int** nextRow = row;
366. **if** ( stringLine[nextRow][1] == '1' )
367. {
368. ++ row;
369. createBinaryTree( BT->leftChild, stringLine, length, row );
370. }
371. **if** ( stringLine[nextRow][2] == '1' )
372. {
373. ++row;
374. createBinaryTree( BT->rightChild, stringLine, length, row );
375. }
376. **return** **true**;
377. }
379. **bool** \_Binary\_Tree::readFileToArray( elementType stringLine[100][3], **int** &length )
380. {
381. **FILE** \*fp;
382. **char** str[100];
384. cout << "Please input the file name(belike includes the file path):" << endl;
385. **char** name[50];// = "bt10.btr";
386. cin >> name;
387. fp = fopen( name, "r" );
388. **if** (!fp)
389. {
390. cout << "Error!" << endl;
391. **return** **false**;
392. }
393. **if** (fgets(str, 1000, fp) != NULL)
394. {
395. **if** (strcmp(str, "BinaryTree\n") != 0)
396. {
397. cout << "Error!" << endl;
398. fclose(fp);
399. **return** **false**;
400. }
401. }
402. length = 0;
403. **while** (fscanf(fp, "%c %c %c\n", &stringLine[length][0], &stringLine[length][1], &stringLine[length][2]) != EOF)
404. {
405. length ++;
406. }
407. fclose(fp);
408. **return** **true**;
409. }

412. **bool** \_Binary\_Tree::emptyBinaryTree()
413. {
414. //if(BTree)
415. //return BTree->leftChild == NULL && BTree->rightChild == NULL;
416. //else
417. **return** BTree == NULL;
418. }
420. **bool** \_Binary\_Tree::\_exit( binTree BT, elementType value )
421. {
422. **if**(!BT)
423. **return** **false**;
424. //return NULL;
425. **if**( BT->data == value )
426. **return** **true**;
427. //return BT;
428. //bitNode \*index = \_exit( BT->leftChild, value );
429. **bool** flag = \_exit( BT->leftChild, value );
430. //if(!index)
431. **if**(!flag)
432. //\_exit( BT->leftChild, value );
433. \_exit( BT->rightChild, value );
434. }
436. binTree \_Binary\_Tree::getNodePoint()
437. {
438. //if( emptyBinaryTree() )
439. //{
440. //throw "Empty binary tree!Error in binTree \_Binary\_Tree::getNodePoint()!\n";
441. //return NULL;
442. //}
443. **return** (\***this**).BTree;
444. }
446. binTree \_Binary\_Tree::getNodePoint( binTree BT, elementType value )
447. {
448. /\*
449. if(!BT)
450. {
451. return NULL;
452. }
453. else
454. {
455. if( BT->data == value )
456. return BT;
457. else
458. {
459. bitNode \*tmp;
460. if( tmp = getNodePoint( BT->leftChild, value ) )
461. return tmp;
462. if( tmp = getNodePoint( BT->rightChild, value ) )
463. return tmp;
464. return NULL;
465. }
466. }
467. \*/
468. **if**(!BT)
469. {
470. **return** NULL;
471. }
472. **else**
473. {
474. **if**( BT->data == value )
475. {
476. **return**  BT;
477. }
478. //getNodePoint( BT->leftChild, value );
479. //getNodePoint( BT->rightChild, value );
481. bitNode \*tmp = getNodePoint( BT->leftChild, value );
482. **if**(!tmp)
483. {
484. getNodePoint( BT->rightChild, value );
485. }
486. //follow statement can't be added to the code
487. //return tmp;
488. }
489. }
491. **void** \_Binary\_Tree::preOrderTraverse(binTree BT)
492. {
493. //if( emptyBinaryTree() )
494. //  {
495. //throw "Empty binary tree!Error in void \_Binary\_Tree::preOrderTraverse(binTree BT) !\n";
496. //return;
497. //}
498. **if** (BT)
499. {
500. cout << BT->data << " ";
501. preOrderTraverse(BT->leftChild);
502. preOrderTraverse(BT->rightChild);
503. }
504. }
506. **void** \_Binary\_Tree::inOrderTraverse(binTree BT)
507. {
508. **if** (BT)
509. {
510. inOrderTraverse(BT->leftChild);
511. cout << BT->data << " ";
512. inOrderTraverse(BT->rightChild);
513. }
514. //return 0;
515. }
517. **void** \_Binary\_Tree::postOrderTraverse( binTree BT )
518. {
519. **if** (BT)
520. {
521. postOrderTraverse(BT->leftChild);
522. postOrderTraverse(BT->rightChild);
523. cout << BT->data << " ";
524. }
525. }
527. **void** \_Binary\_Tree::destroy( binTree BT )
528. {
529. **if**(BT)
530. {
531. destroy( BT->leftChild );
532. destroy( BT->rightChild );
533. **delete** BT;
534. BT = NULL;
535. }
536. }
538. **void** \_Binary\_Tree::level( binTree BT, **int** number )
539. {
541. **if**(BT)
542. {
543. level( BT->leftChild, number + 1 );
544. ///number +=3;
545. //cout << number << endl;
546. cout << BT->data << " level: " << number << endl;
548. level( BT->rightChild, number + 1 );
549. //number -=2;
550. }
551. //number --;
552. }
554. **int** \_Binary\_Tree::height( binTree BT )
555. {
556. **if**(!BT)
557. {
558. **return** 0;
559. }
560. **else**
561. {
562. **int** i = height( BT->leftChild );
563. **int** j = height( BT->rightChild );
564. **return** i < j ? j + 1 : i + 1;
566. }
567. }
569. **int** \_Binary\_Tree::numberOfBTreeNode( binTree BT )
570. {
571. **if**(!BT)
572. **return** 0;
573. **else**
574. {
575. **return** numberOfBTreeNode( BT->leftChild ) + numberOfBTreeNode( BT->rightChild ) + 1;
576. }
577. }
579. **int** \_Binary\_Tree::numberOfBTreeLeafNode( binTree BT, **int** &number )
580. {
581. **if**(!BT)
582. {
583. **return** 0;
584. }
585. **else**
586. {
587. **if**( !BT->leftChild && !BT->rightChild )
588. //number += 1;
589. number ++;
590. //return 1;
591. **else**
592. {
593. numberOfBTreeLeafNode( BT->leftChild, number );
594. numberOfBTreeLeafNode( BT->rightChild, number );
595. }
596. **return** number;
597. }
598. }
600. **void** \_Binary\_Tree::numberOfNodeDegreeTwo( binTree BT, **int** &number )
601. {
602. **if**(!BT)
603. {
604. **return**;
605. }
606. **else**
607. {
608. **if**( BT->leftChild && BT->rightChild )
609. //number += 1;
610. number += 1;
611. //return 1;
612. //else
613. //{
614. numberOfNodeDegreeTwo( BT->leftChild, number );
615. numberOfNodeDegreeTwo( BT->rightChild, number );
616. //return numberOfNodeDegreeTwo( BT->leftChild, number ) + numberOfNodeDegreeTwo( BT->rightChild, number );
617. //}
618. //return number;
619. }
620. }
622. /\*
623. void \_Binary\_Tree::family( binTree BT, elementType1 number )
624. {
625. if(!BT)
626. {
627. return;
628. }
629. if( BT->leftChild->data == number || BT->rightChild->data == number )
630. {
631. cout << "parent ---- " << BT->data << endl;
632. if( BT->leftChild->data == number && BT->rightChild )
633. {
634. cout << "rights sibling ---- " << BT->rightChild->data << endl;
635. }
636. if( BT->leftChild && BT->rightChild->data == number )
637. {
638. cout << "left sibling ---- " << BT->leftChild->data << endl;
639. }
640. }
641. if( BT->data == number && ( BT->leftChild || BT->rightChild ) )
642. {
643. cout << ( BT->leftChild ? "left child ---- " : true ) << endl;
644. cout << ( BT->rightChild ? "right child ---- " : true ) << endl;
645. }
646. family( BT->leftChild, number );
647. family( BT->rightChild, number );
648. //if( BT->data == number && BT-)
649. //if( BT->leftChild->data == number &&)
650. }
651. \*/
653. //bool \_Binary\_Tree::getParent( binTree BT, elementType number, bool flag )
654. **void** \_Binary\_Tree::getParent( binTree BT, elementType value, **bool** &flag )
655. {
656. **if**(!BT)
657. {
658. //return false;
659. **return**;
660. }
661. **if**( ( BT->leftChild && BT->leftChild->data == value ) || ( BT->rightChild ) && ( BT->rightChild->data == value ) )
662. {
663. flag = **true**;
664. cout << value << " Parent ---- " << BT->data << endl;
665. **return**;
666. //return true;
667. }
668. /\*
669. if( BT && BT->rightChild->data == number )
670. {
671. cout << "parent ---- " << BT->data << endl;
672. return true;
673. }
674. \*/
675. getParent( BT->leftChild, value, flag );
676. getParent( BT->rightChild, value, flag );
677. }
679. binTree \_Binary\_Tree::getParent( binTree BT, elementType value )
680. {
682. **if**( !\_exit( BT, value ) )
683. {
684. cerr << value << " is not in the binary tree!" << endl;
685. cerr << "Error in binTree \_Binary\_Tree::getParent( binTree BT, elementType value )!" << endl;
686. **return** NULL;
687. }
689. **if**(!BT)
690. {
691. **return** NULL;
692. }
693. **if**( BT->data == value )
694. {
695. **return** BT;
696. }
697. **if**( ( BT->leftChild && BT->leftChild->data == value ) || ( BT->rightChild && BT->rightChild->data == value ) )//|| BT->rightChild->data == value )
698. {
699. **return** BT;
700. }
701. bitNode \*tmp = getParent( BT->leftChild, value );
702. **if**(!tmp)
703. {
704. getParent( BT->rightChild, value );
705. }
706. }
708. **void** \_Binary\_Tree::getSibling( binTree BT, elementType value, **bool** &flag )
709. {
710. **if**(!BT)
711. {
712. cout << value << " No LeftSibling!" << endl << value << " No RightSibling!" << endl;
713. **return**;
714. }
715. **if**( !flag &&  !BT->leftChild )//|| !BT->rightChild )//write as "if(!BT)" would result error
716. {
718. **if**( BT->rightChild )
719. {
720. getSibling( BT->rightChild, value, flag );
721. //return;
722. }
723. **else**
724. {
725. //cout << value << " No LeftSibling!" << endl;
726. **return**;
727. }
728. **return**;//why would deleting the statement cause error!
729. }
731. **if**( !flag && !BT->rightChild )
732. {
734. **if**( BT->leftChild )
735. {
736. getSibling( BT->leftChild, value, flag );
738. }
739. **else**
740. {
741. //cout << value << "No LeftSibling!" << endl;
742. //cout << value << " No RightSibling!" << endl;
743. **return**;
745. }
746. **return**;//why would deleting the statement cause error!
747. }
748. **if**( BT->rightChild->data == value )
749. {
750. **if**( BT->leftChild )
751. {
752. flag = **true**;
753. cout << value << " LeftSibling ---- " << BT->leftChild->data << endl;
754. **return**;
755. }
756. **else** **if**( !BT->leftChild )
757. {
758. cout << value << " No LeftSibling!" << endl;
759. **return**;
760. }
762. }
763. **if**( BT->leftChild->data == value )
764. {
765. **if**( BT->rightChild )
766. {
767. flag = **true**;
768. cout << value << " RightSibling ---- " << BT->rightChild->data << endl;
769. **return**;
770. }
771. **else** **if**( !BT->rightChild )
772. {
773. cout << value << " No RightSibling!" << endl;
774. **return**;
775. }
776. }
777. getSibling( BT->leftChild, value, flag );
778. **if**( !flag && BT->rightChild )
779. getSibling( BT->rightChild, value, flag );
780. }
782. **void** \_Binary\_Tree::getSibling( binTree BT, elementType value )
783. {
784. bitNode \*parent = getParent( BT, value );
786. **if**( BT->data == value )
787. {
788. cout << value << " is the root node,neither left sibling also useless right sibling!" << endl;
789. **return**;
790. }
791. **if**( !\_exit( BT, value ) )
792. {
793. cout << value << " is not in the binary-tree!" << endl;
794. cerr << "Error in void \_Binary\_Tree::getSibling( binTree BT, elementType value )!" << endl;
795. **return**;
796. }
797. **if**( parent->leftChild && parent->leftChild->data == value )
798. {
799. **if**( parent->rightChild )
800. {
801. cout << value << " RightSibling ---- " << parent->rightChild->data << endl;
802. **return**;
803. }
804. **else**
805. {
806. cout << value << " No RightSibling!" << endl;
807. **return**;
808. }
809. }
810. **if**( parent->rightChild && parent->rightChild->data == value )
811. {
812. **if**( parent->leftChild )
813. {
814. cout << value << " LeftSibling ---- " << parent->leftChild->data << endl;
815. **return**;
816. }
817. **else**
818. {
819. cout << value << " No LeftSibling!" << endl;
820. **return**;
821. }
822. }
823. }
825. **void** \_Binary\_Tree::getChild( binTree BT, elementType value, **bool** &flag )//It costed me several minutes to
826. {                                                                       //write and almost one hour to
827. /\*                                                                  //perfect the function
828. if( BT->leftChild )
829. {
830. getChild( BT->leftChild, value, flag );
831. }
832. if( BT->rightChild )
833. {
834. getChild( BT->rightChild, value, flag );
835. }
836. \*/
838. /\*
839. if( !BT->leftChild )//|| !BT->rightChild )
840. {
841. if(flag)
842. {
843. cout << "No LeftChild! " << endl;
844. flag = true;
845. return;
846. }
847. else
848. {
849. cout << "No LeftChild! " << endl;
850. flag = false;
851. }
852. //return;
853. }
854. if( !BT->rightChild )
855. {
856. if(flag)
857. {
858. cout << "No RightChild! " << endl;
859. flag = true;
860. return;
861. }
862. else
863. {
864. cout << "No RightChild! " << endl;
865. flag = false;
866. }
867. }
868. \*/
869. //if(!BT)
870. //  {
871. //  return;
872. //  }
873. **if**( !\_exit( BT, value ) )
874. {
875. cerr << value << " is not in the binary tree!\nError in void \_Binary\_Tree::getChild( binTree BT, elementType value, bool &flag )" << endl;
876. **return**;
877. }
878. **if**( BT->data == value )//at first I neglected this detail that resulted wrong judgement at root-node
879. {
880. **if**( BT->leftChild )
881. {
882. flag = **true**;
883. cout << value << " LeftChild ---- " << BT->leftChild->data << endl;
885. }
886. **else**
887. {
888. cout << "No LeftChild!" << endl;
890. }
891. **if**( BT->rightChild )
892. {
893. flag = **true**;
894. cout << value << " RightChild ---- " << BT->rightChild->data << endl;
895. **return**;
896. }
897. **else**
898. {
899. cout << "No RightChild! " << endl;
900. **return**;
901. }
902. }
903. **if**( !BT->leftChild )
904. {
905. **if**( BT->rightChild )
906. {
907. getChild( BT->rightChild, value, flag );
908. }
909. **return**;
910. /\*
911. if(flag)
912. {
913. return;
914. }
915. else
916. {
917. flag = false;
918. return;
919. }\*/
920. }
921. **if**( !BT->rightChild )
922. {
923. **if**( BT->leftChild )
924. {
925. getChild( BT->leftChild, value, flag );
926. }
927. **return**;
928. //if(flag)
929. //{
930. //  flag = false;
931. //}
932. /\*
933. if(flag)
934. {
935. return;
936. }
937. else
938. {
939. flag = false;
940. return;
941. }\*/
942. }
943. /\*
944. if( BT->rightChild->data == value )
945. {
946. if( BT->rightChild->leftChild )
947. {
948. flag = true;
949. cout << value << " LeftChild ---- " << BT->rightChild->leftChild->data << endl;
950. //return;
951. }
952. else
953. {
954. cout << "No LeftChild!" << endl;
955. }
956. if( BT->rightChild->rightChild )
957. {
958. flag = true;
959. cout << value << " RightChild ---- " << BT->rightChild->rightChild->data << endl;
960. return;
961. }
962. else
963. {
964. cout << "No RightChild! " << endl;
965. return;
966. }
967. //else
968. //{
969. //flag = false;
970. //  return;
971. //}
972. }
973. if( BT->leftChild->data == value )
974. {
975. if( BT->leftChild->leftChild )
976. {
977. flag = true;
978. cout << value << " LeftChild ---- " << BT->leftChild->leftChild->data << endl;
979. //return;
980. }
981. else
982. {
983. cout << "No LeftChild!" << endl;
984. }
985. if( BT->leftChild->rightChild )
986. {
987. flag = true;
988. cout << value << " RightChild ---- " << BT->leftChild->rightChild->data << endl;
989. return;
990. }
991. else
992. {
993. cout << "No RightChild! " << endl;
994. return;
995. }
996. //else
997. //{
998. //flag = false;
999. //  return;
1000. //}
1001. }
1002. \*/
1003. getChild( BT->leftChild, value, flag );
1004. getChild( BT->rightChild, value, flag );
1005. }
1007. **int** \_Binary\_Tree::levelJudge( binTree BT, elementType value, **int** &number, **int** level )
1008. {
1009. bitNode \*position = getNodePoint( getNodePoint(), value );
1010. **if**(!position)
1011. {
1012. cout << "The value you typed is not in the binary tree!" << endl;
1013. //return -1;
1014. number = -1;
1015. **return** number;
1016. }
1017. /\*
1018. int level;
1019. if ( BT == NULL )
1020. return 0;
1021. else if ( BT->data == value )
1022. return number;
1023. else
1024. {
1025. number ++;
1026. level = levelJudge( BT->leftChild, value, number );
1027. if ( level != 0 )
1028. return number;
1029. else
1030. {
1031. //number ++;
1032. return levelJudge( BT->rightChild, value, number );
1033. }
1034. }
1035. \*/
1036. **if**(BT)
1037. {
1038. **if**( BT->data == value )
1039. {
1040. //number ++;
1041. number = level;
1042. }
1043. //number ++;
1044. levelJudge( BT->leftChild, value, number, level + 1 );
1045. //number ++;
1046. levelJudge( BT->rightChild, value, number, level + 1);
1047. }
1048. }
1050. **void** \_Binary\_Tree::exchangeLeftAndRightSibling( binTree BT )
1051. {
1052. **if**( BT && BT->leftChild && BT->rightChild )
1053. {
1054. bitNode \*tmp = BT->leftChild;
1055. BT->leftChild = BT->rightChild;
1056. BT->rightChild = tmp;
1057. exchangeLeftAndRightSibling( BT->leftChild );
1058. exchangeLeftAndRightSibling( BT->rightChild );
1059. }
1060. }
1062. **void** \_Binary\_Tree::copyBTree( binTree BT1, binTree BT )
1063. {
1064. bitNode \*tmp = NULL;//new bitNode;
1065. BT1->data = BT->data;
1066. **if**( BT->leftChild )
1067. {
1068. BT1->leftChild = **new** bitNode;
1069. tmp = BT1->leftChild;
1070. tmp->leftChild = NULL;
1071. tmp->rightChild = NULL;
1072. copyBTree( tmp, BT->leftChild );
1073. }
1074. **if**( BT->rightChild )
1075. {
1076. BT1->rightChild = **new** bitNode;
1077. tmp = BT1->rightChild;
1078. tmp->leftChild = NULL;
1079. tmp->rightChild = NULL;
1080. copyBTree( tmp, BT->rightChild );
1081. }
1082. }
1084. **void** \_Binary\_Tree::levelOrderTraverse( binTree BT )
1085. {
1086. clq.enQueue(BT);
1087. **while**( !clq.emptyCharLinkedQueue() )
1088. {
1089. //CLNode \*tmp = NULL;
1090. clq.getFront(BT);
1091. cout << BT->data << " ";
1092. clq.deQueue();
1093. **if**( BT->leftChild != NULL )
1094. {
1095. clq.enQueue( BT->leftChild );
1096. }
1097. **if**( BT->rightChild != NULL )
1098. {
1099. clq.enQueue( BT->rightChild );
1100. }
1101. }
1102. }
1104. **void** \_Binary\_Tree::allLeafToRootPath( binTree BT, **char** \*path, **int** &pathLength )
1105. {
1106. **if**(BT)
1107. {
1108. **if**( !BT->leftChild && !BT->rightChild )
1109. {
1110. path[pathLength] = BT->data;
1111. cout << BT->data << " leaf to root path: " << endl;
1112. **for**( **int** i = pathLength; i >= 0; i -- )
1113. {
1114. **if**( i != 0 )
1115. cout << path[i] << " --> ";
1116. **else**
1117. cout << path[i] << "\n";
1118. }
1119. //cout << endl;
1120. }
1121. **else**
1122. {
1123. path[ pathLength ++ ] = BT->data;
1124. allLeafToRootPath( BT->leftChild, path, pathLength );
1125. allLeafToRootPath( BT->rightChild, path, pathLength );
1126. pathLength --;
1127. }
1128. }
1129. }
1131. **void** \_Binary\_Tree::binaryTreeLongestPath( binTree BT, elementType \*path, **int** &pathLength,
1132. elementType \*longestPath, **int** &longestLength )
1133. {
1134. **if**(BT)
1135. {
1136. **if**( !BT->leftChild && !BT->rightChild )
1137. {
1138. path[pathLength] = BT->data;
1139. **if**( pathLength > longestLength)
1140. {
1141. //cout << BT->data << " leaf to root path: " << endl;
1142. //longestPath = pathLength;
1143. **for**( **int** i = pathLength; i >= 0; i -- )
1144. {
1145. /\*
1146. if( i != 0 )
1147. cout << path[i] << " --> ";
1148. else
1149. cout << path[i] << "\n";
1150. \*/
1151. longestPath[i] = path[i];
1152. }
1153. longestLength = pathLength;
1154. }
1155. //longestLength = pathLength;
1156. //cout << endl;
1157. }
1158. **else**
1159. {
1160. path[ pathLength ++ ] = BT->data;
1161. binaryTreeLongestPath( BT->leftChild, path, pathLength, longestPath, longestLength );
1162. binaryTreeLongestPath( BT->rightChild, path, pathLength, longestPath, longestLength );
1163. pathLength --;
1164. }
1165. }
1166. }
1168. binTree \_Binary\_Tree::nearestAncestor( binTree BT, bitNode \*BNode1, bitNode \*BNode2 )
1169. {
1170. //  if( !\_exit( BT, BNode1->data ) )
1171. //{
1172. //  cout << BNode1->data << " is not in the binary tree!" << endl;
1173. //  return NULL;
1174. //}
1175. //if( !\_exit( BT, BNode2->data ) )
1176. //{
1177. //  cout << BNode2->data << " is not in the binary tree!" << endl;
1178. //  return NULL;
1179. //}
1180. **if**( !BT || !BNode1 || !BNode2 )
1181. {
1182. //cout << "NO ANCESTOR!" << endl;
1183. **return** NULL;
1184. }
1185. **if**( BT == BNode1 || BT == BNode2 )
1186. {
1187. //cout << BT->data << endl;
1188. **return** BT;
1189. }
1190. bitNode \*left = nearestAncestor( BT->leftChild, BNode1, BNode2 );
1191. bitNode \*right = nearestAncestor( BT->rightChild, BNode1, BNode2 );
1192. **if**( left && right )
1193. {
1194. //cout << BT->data << endl;
1195. **return** BT;
1196. }
1197. **else** **if**(!left)
1198. {
1199. //cout << right->data << endl;
1200. **return** right;
1201. }
1202. **else**
1203. {
1204. //cout << left->data << endl;
1205. **return** left;
1206. }
1207. }


1211. // BinaryTree.cpp : Defines the entry point for the console application.
1212. //
1214. #include "stdafx.h"
1215. #include "\_Binary\_Tree.h"
1216. #include "charLinkedQueue.h"

1219. **void** test1()
1220. {
1221. \_Binary\_Tree BT1;
1222. elementType strLine[100][3];
1223. **int** nRow = 0, nLen = 0;
1224. binTree index;
1225. BT1.readFileToArray(strLine,nLen);
1226. //BT1.\_Binary\_Tree();
1228. BT1.createBinaryTree( index,strLine, nLen, nRow);
1230. cout << "The PreOrder traversal of sequence is" << endl;
1231. BT1.preOrderTraverse(BT1.getNodePoint());
1232. cout << endl;
1233. cout << "The middle order traversal of sequence is" << endl;
1234. BT1.inOrderTraverse(BT1.getNodePoint());
1235. cout << endl;
1236. cout << "The post order traversal of sequence is" << endl;
1237. BT1.postOrderTraverse(BT1.getNodePoint());
1238. cout << endl;
1239. cout << "The level order traversal of sequence is" << endl;
1240. BT1.levelOrderTraverse( BT1.getNodePoint() );
1241. cout << endl;
1242. **int** n = 1;
1243. BT1.level( BT1.getNodePoint() , n );
1244. cout << "BTree height: "<< BT1.height( BT1.getNodePoint() ) << endl;
1245. cout << "Total node: " << BT1.numberOfBTreeNode( BT1.getNodePoint() ) << endl;
1246. **int** m = 0;
1247. cout << "Leaf node: " << BT1.numberOfBTreeLeafNode( BT1.getNodePoint(), m ) << endl;
1248. **int** a = 0;
1249. BT1.numberOfNodeDegreeTwo( BT1.getNodePoint(), a );
1250. cout << "Node degree two: " << a << endl;
1251. }
1253. **void** test2()
1254. {
1255. \_Binary\_Tree BT1;
1256. elementType strLine[100][3];
1257. **int** nRow = 0, nLen = 0;
1258. binTree index;
1259. BT1.readFileToArray(strLine,nLen);
1260. //BT1.\_Binary\_Tree();
1262. BT1.createBinaryTree( index,strLine, nLen, nRow);
1263. **char** ch;
1264. **int** key;
1265. cout << "Please input a letter as operational character and a number to choose the operation,separated by space.\n\"1\" for searching parent-node£¬\"2\" for searching sibiling-node and \"3\" for searching child-node." << endl;
1266. **while**( cin>> ch >> key )
1267. {
1268. **if**( key == 1 )
1269. {
1270. **bool** flag = **false**;
1271. bitNode \*index = NULL;
1272. index = BT1.getParent( BT1.getNodePoint(), ch );
1273. **if**( index && index != BT1.getNodePoint() )
1274. cout << ch << " parent ---- " << index->data << endl;
1275. **else** **if**( index && index->data == ch )
1276. cout << ch << " is the root node, no parent." << endl;
1277. **else** **if**(index)
1278. cout << ch << " parent ---- " << index->data << endl;
1279. }
1280. **else** **if**( key == 2 )
1281. {
1282. BT1.getSibling( BT1.getNodePoint(), ch );
1283. }
1284. **else** **if**( key == 3 )
1285. {
1286. **bool** flag;
1287. BT1.getChild( BT1.getNodePoint(), ch, flag );
1288. }
1289. //Sleep( 1000 \* 60 );
1290. //system( "cls" );
1291. //cout << "Please input a letter as operational character and a number to choose the operation,separated by space.\n\"1\" for searching parent-node£¬\"2\" for searching sibiling-node and \"3\" for searching child-node." << endl;
1292. }
1293. }
1295. **void** test3()
1296. {
1297. \_Binary\_Tree BT1;
1298. elementType strLine[100][3];
1299. **int** nRow = 0, nLen = 0;
1300. binTree index;
1301. BT1.readFileToArray(strLine,nLen);
1302. //BT1.\_Binary\_Tree();
1304. BT1.createBinaryTree( index,strLine, nLen, nRow);
1306. elementType value;
1307. **int** cnt = 0;
1308. cout << "Please input a letter and the program will judge it in which layer of the binary-tree!" << endl;
1309. **while**( cin >> value )
1310. {
1311. **int** number = 1;
1312. BT1.levelJudge( BT1.getNodePoint(), value , number, 1 );
1313. cout << value << " ---- " << number << " level!" << endl;
1314. cnt ++;
1315. //Sleep( 1000 \* 60 );
1316. **if**( cnt % 10 == 0 )
1317. system( "cls" );
1319. cout << "Please input a letter and the program will judge it in which layer of the binary-tree!" << endl;
1320. }
1321. }
1323. **void** test4()
1324. {
1325. elementType str[10000];
1326. cout << "Please input a character array that will be transformed to the elements of a binary tree.\nAttention the typed array must extend for the complete binary tree!" << endl;
1327. **while**( cin >> str )
1328. {
1329. \_Binary\_Tree BT1(str);
1330. cout << "The PreOrder traversal of sequence is" << endl;
1331. BT1.preOrderTraverse(BT1.getNodePoint());
1332. cout << endl;
1333. cout << "The middle order traversal of sequence is" << endl;
1334. BT1.inOrderTraverse(BT1.getNodePoint());
1335. cout << endl;
1336. cout << "The post order traversal of sequence is" << endl;
1337. BT1.postOrderTraverse(BT1.getNodePoint());
1338. cout << endl;
1339. cout << "The level order traversal of sequence is" << endl;
1340. BT1.levelOrderTraverse( BT1.getNodePoint() );
1341. cout << endl;
1342. **int** n = 1;
1343. BT1.level( BT1.getNodePoint() , n );
1344. cout << "BTree height: "<< BT1.height( BT1.getNodePoint() ) << endl;
1345. cout << "Total node: " << BT1.numberOfBTreeNode( BT1.getNodePoint() ) << endl;
1346. **int** m = 0;
1347. cout << "Leaf node: " << BT1.numberOfBTreeLeafNode( BT1.getNodePoint(), m ) << endl;
1348. **int** a = 0;
1349. BT1.numberOfNodeDegreeTwo( BT1.getNodePoint(), a );
1350. cout << "Node degree two: " << a << endl;
1351. Sleep( 1000 \* 60 );
1352. system( "cls" );
1353. cout << "Please input a character array that will be transformed to the elements of a binary tree.\nAttention the typed array must extend for the complete binary tree!" << endl;
1354. }
1355. }
1357. **void** test5()
1358. {
1359. \_Binary\_Tree BT1;
1360. elementType strLine[100][3];
1361. **int** nRow = 0, nLen = 0;
1362. binTree index;
1363. BT1.readFileToArray(strLine,nLen);
1364. //BT1.\_Binary\_Tree();
1366. BT1.createBinaryTree( index,strLine, nLen, nRow);
1367. cout << "The program will exchange the left subtree and right subtree of the file-inputed binary tree!" << endl;
1368. cout << "The origin binary tree is as follow:" << endl;
1370. cout << "The PreOrder traversal of sequence is" << endl;
1371. BT1.preOrderTraverse(BT1.getNodePoint());
1372. cout << endl;
1373. cout << "The middle order traversal of sequence is" << endl;
1374. BT1.inOrderTraverse(BT1.getNodePoint());
1375. cout << endl;
1376. cout << "The post order traversal of sequence is" << endl;
1377. BT1.postOrderTraverse(BT1.getNodePoint());
1378. cout << endl;
1379. cout << "The level order traversal of sequence is" << endl;
1380. BT1.levelOrderTraverse( BT1.getNodePoint() );
1381. cout << endl;
1383. BT1.exchangeLeftAndRightSibling( BT1.getNodePoint() );
1384. cout << "The following for the exchange of binary tree:" << endl;
1385. cout << "The PreOrder traversal of sequence is" << endl;
1386. BT1.preOrderTraverse(BT1.getNodePoint());
1387. cout << endl;
1388. cout << "The middle order traversal of sequence is" << endl;
1389. BT1.inOrderTraverse(BT1.getNodePoint());
1390. cout << endl;
1391. cout << "The post order traversal of sequence is" << endl;
1392. BT1.postOrderTraverse(BT1.getNodePoint());
1393. cout << endl;
1394. cout << "The level order traversal of sequence is" << endl;
1395. BT1.levelOrderTraverse( BT1.getNodePoint() );
1396. cout << endl;
1397. }
1399. **void** test6()
1400. {
1401. \_Binary\_Tree BT1;
1402. elementType strLine[100][3];
1403. **int** nRow = 0, nLen = 0;
1404. binTree index;
1405. BT1.readFileToArray(strLine,nLen);
1406. //BT1.\_Binary\_Tree();
1408. BT1.createBinaryTree( index,strLine, nLen, nRow);
1409. cout << "The program will copy the file-inputed binary tree to another empty one!" << endl;
1410. \_Binary\_Tree BT2;
1412. cout << "The origin binary tree is as follow:" << endl;
1413. cout << "The PreOrder traversal of sequence is" << endl;
1414. BT1.preOrderTraverse(BT1.getNodePoint());
1415. cout << endl;
1416. cout << "The middle order traversal of sequence is" << endl;
1417. BT1.inOrderTraverse(BT1.getNodePoint());
1418. cout << endl;
1419. cout << "The post order traversal of sequence is" << endl;
1420. BT1.postOrderTraverse(BT1.getNodePoint());
1421. cout << endl;
1422. cout << "The level order traversal of sequence is" << endl;
1423. BT1.levelOrderTraverse( BT1.getNodePoint() );
1424. cout << endl;
1426. cout << "The empty binary tree is as follow:" << endl;
1428. cout << "The PreOrder traversal of sequence is" << endl;
1429. BT2.preOrderTraverse(BT2.getNodePoint());
1430. cout << endl;
1431. cout << "The middle order traversal of sequence is" << endl;
1432. BT2.inOrderTraverse(BT2.getNodePoint());
1433. cout << endl;
1434. cout << "The post order traversal of sequence is" << endl;
1435. BT2.postOrderTraverse(BT2.getNodePoint());
1436. cout << endl;
1437. cout << "The level order traversal of sequence is" << endl;
1438. BT2.levelOrderTraverse( BT2.getNodePoint() );
1439. cout << endl;
1441. cout << "The following for the copy of binary tree:" << endl;
1443. BT1.copyBTree( BT2.getNodePoint(), BT1.getNodePoint() );
1445. cout << "The PreOrder traversal of sequence is" << endl;
1446. BT2.preOrderTraverse(BT2.getNodePoint());
1447. cout << endl;
1448. cout << "The middle order traversal of sequence is" << endl;
1449. BT2.inOrderTraverse(BT2.getNodePoint());
1450. cout << endl;
1451. cout << "The post order traversal of sequence is" << endl;
1452. BT2.postOrderTraverse(BT2.getNodePoint());
1453. cout << endl;
1454. cout << "The level order traversal of sequence is" << endl;
1455. BT2.levelOrderTraverse( BT2.getNodePoint() );
1456. cout << endl;
1458. }
1460. **void** test7()
1461. {
1462. \_Binary\_Tree BT1;
1463. elementType strLine[100][3];
1464. **int** nRow = 0, nLen = 0;
1465. binTree index;
1466. BT1.readFileToArray(strLine,nLen);
1467. //BT1.\_Binary\_Tree();
1469. BT1.createBinaryTree( index,strLine, nLen, nRow);
1470. cout << "The program will output the paths that each leaf node of the binary tree to the root node." << endl;
1472. cout << "The origin binary tree is as follow:" << endl;
1473. cout << "The preOrder traversal of sequence is" << endl;
1474. BT1.preOrderTraverse(BT1.getNodePoint());
1475. cout << endl;
1476. cout << "The middle order traversal of sequence is" << endl;
1477. BT1.inOrderTraverse(BT1.getNodePoint());
1478. cout << endl;
1479. cout << "The post order traversal of sequence is" << endl;
1480. BT1.postOrderTraverse(BT1.getNodePoint());
1481. cout << endl;
1482. cout << "The level order traversal of sequence is" << endl;
1483. BT1.levelOrderTraverse( BT1.getNodePoint() );
1484. cout << endl;
1486. **int** pathLength = 0;
1487. elementType \*path = **new** **char**[ BT1.numberOfBTreeNode( BT1.getNodePoint() ) ];
1488. BT1.allLeafToRootPath( BT1.getNodePoint(), path, pathLength );
1490. }
1492. **void** test8()
1493. {
1494. \_Binary\_Tree BT1;
1495. elementType strLine[100][3];
1496. **int** nRow = 0, nLen = 0;
1497. binTree index;
1498. BT1.readFileToArray(strLine,nLen);
1499. //BT1.\_Binary\_Tree();
1501. BT1.createBinaryTree( index,strLine, nLen, nRow);
1502. cout << "The program will ouput the level order traversal of the file-inputed binary tree." << endl;
1503. cout << "The level order traversal of sequence is" << endl;
1504. BT1.levelOrderTraverse( BT1.getNodePoint() );
1505. cout << endl;
1506. }
1508. **void** test9()
1509. {
1510. \_Binary\_Tree BT1;
1511. **char** strLine[100][3];
1512. **int** nRow = 0, nLen = 0;
1513. binTree index;
1514. BT1.readFileToArray(strLine,nLen);
1515. //BT1.\_Binary\_Tree();
1517. BT1.createBinaryTree( index,strLine, nLen, nRow);
1519. cout << "Please input two character and the program will output their nearset ancestor." << endl;
1520. elementType ch1, ch2;
1521. **while**( cin >> ch1 >> ch2 )
1522. {
1523. //BT1.nearestAncestor( binTree BT, bitNode \*BNode1, bitNode \*BNode2 );
1524. bitNode \*index1 = BT1.getNodePoint( BT1.getNodePoint(), ch1 );
1526. **if**(!index1)
1527. {
1528. cout << ch1 << " is not in the binary tree!" << endl;
1529. }
1531. bitNode \*index2 = BT1.getNodePoint( BT1.getNodePoint(), ch2 );
1533. **if**(!index2)
1534. {
1535. cout << ch2 << " is not in the binary tree!" << endl;
1536. }
1538. **if**( index1 && index2 )
1539. {
1540. bitNode \*target = BT1.nearestAncestor( BT1.getNodePoint(), index1, index2 );
1541. cout << "The nearset ancestor of " << ch1 << " and " << ch2 << " is " << target->data << endl;
1542. }
1543. cout << "Please input two character and the program will output their nearset ancestor." << endl;
1544. }
1545. }
1547. **void** test10()
1548. {
1549. \_Binary\_Tree BT1;
1550. **char** strLine[100][3];
1551. **int** nRow = 0, nLen = 0;
1552. binTree index;
1553. BT1.readFileToArray(strLine,nLen);
1554. //BT1.\_Binary\_Tree();
1556. BT1.createBinaryTree( index,strLine, nLen, nRow);
1557. cout << "The program will output the longest path of the binary tree." << endl;
1558. **int** pathLength = 0, longestLength = 0;
1559. elementType \*path1 = **new** **char**[ BT1.numberOfBTreeNode( BT1.getNodePoint() ) ];
1560. elementType \*longestPath = **new** **char**[ BT1.numberOfBTreeNode( BT1.getNodePoint() ) ];
1561. BT1.binaryTreeLongestPath( BT1.getNodePoint(), path1, pathLength, longestPath, longestLength );
1562. cout << "Longest path:" << endl;
1563. **for**( **int** i = longestLength; i >= 0; i -- )
1564. {
1565. **if**( i!= 0 )
1566. cout << longestPath[i] << " --> ";
1567. **else**
1568. cout << longestPath[i] << endl;
1570. }
1571. }
1573. **int** main(**int** argc, **char**\* argv[])
1574. {
1575. //test1();
1576. test2();
1577. //test3();
1578. //test4();
1579. //test5();
1580. //test6();
1581. //test7();
1582. //test8();
1583. //test9();
1584. //test10();
1585. **return** 0;
1586. }

**6.6 调试过程中出现的bug总结**

往事不堪回首。

刚开始应该写成将data写成string或者直接将整个函数写成模板的，写完了最后测试时才发现现在的写法有诸多不便；但修改的话就又要重构一遍，懒得整了。

刚开始尝试写英文注释的，后面知难而退了；不过原来的英文注释我保留了。

**附：测试数据与对应的二叉树图形**

bt4.btr

BinaryTree

A 0 1

B 0 1

C 0 1

D 0 0



bt8.btr

BinaryTree

1 1 1

2 1 1

3 1 1

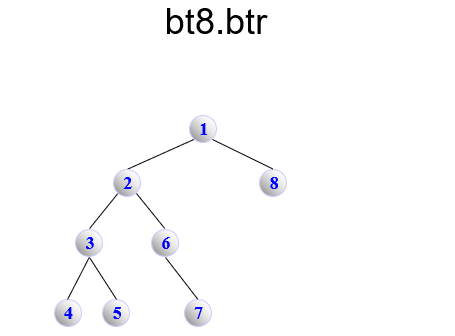
4 0 0

5 0 0

6 0 1

7 0 0

8 0 0



bt9.btr

BinaryTree

a 1 1

b 1 1

d 0 0

e 1 1

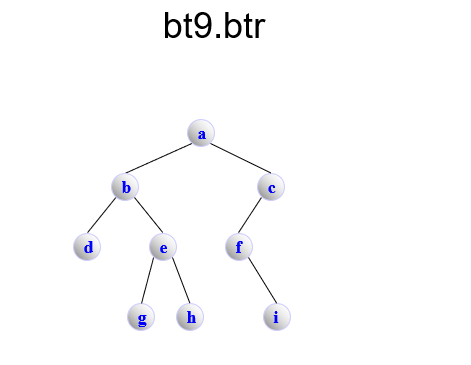
g 0 0

h 0 0

c 1 0

f 0 1

i 0 0



bt10.btr

BinaryTree

A 1 1

B 1 1

C 0 0

D 1 0

E 0 0

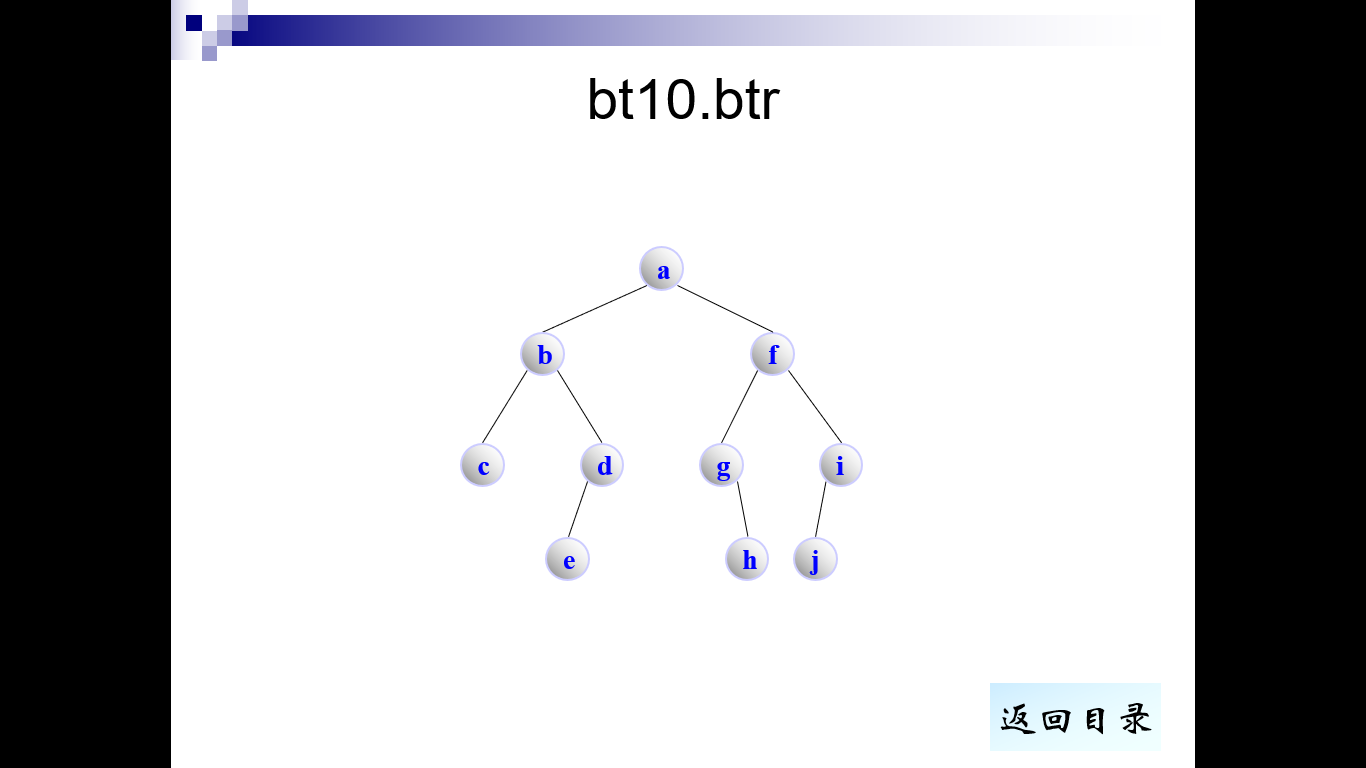
F 1 1

G 0 1

H 0 0

I 1 0

J 0 0



bt12.btr

BinaryTree

A 1 1

B 1 1

C 0 0

D 1 1

E 0 0

F 0 0

G 1 1

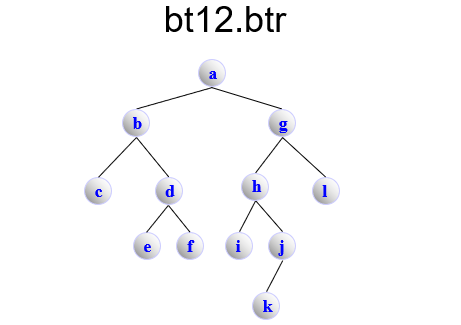
H 1 1

I 0 0

J 1 0

K 0 0

L 0 0



bt14.btr

BinaryTree

A 1 0

B 1 1

C 1 1

D 0 1

E 0 0

F 1 0

G 0 1

H 0 0

I 1 0

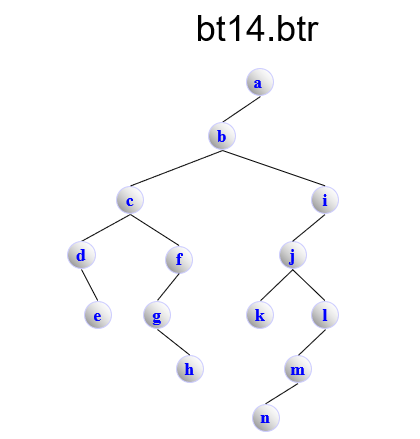
J 1 1

K 0 0

L 1 0

M 1 0

N 0 0



bt15.btr

BinaryTree

A 1 1

B 1 1

D 1 0

G 0 0

E 1 1

H 0 0

I 0 0

C 0 1

F 1 1

J 1 1

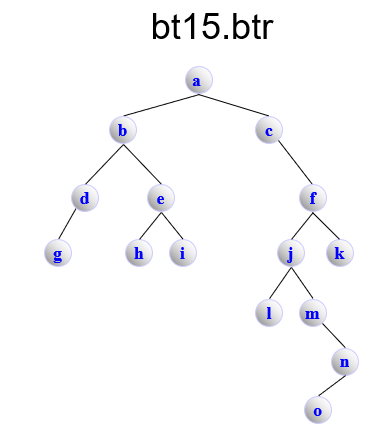
L 0 0

M 0 1

N 1 0

O 0 0

K 0 0



bt21.btr

BinaryTree

a 1 1

b 1 1

c 1 1

d 1 1

e 0 0

f 0 0

g 0 0

h 1 1

i 0 0

j 0 0

k 1 1

l 1 1

m 0 0

n 0 0

o 1 1

p 0 1

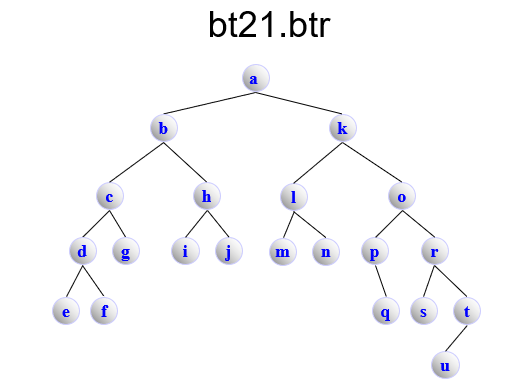
q 0 0

r 1 1

s 0 0

t 1 0

u 0 0



bt261.btr

BinaryTree

a 1 1

b 1 1

c 1 1

d 1 1

e 0 0

f 0 0

g 1 0

h 0 0

i 1 1

j 0 1

k 0 0

l 1 0

m 0 0

n 1 1

o 1 1

p 1 1

q 0 0

r 0 0

s 0 0

t 1 1

u 1 1

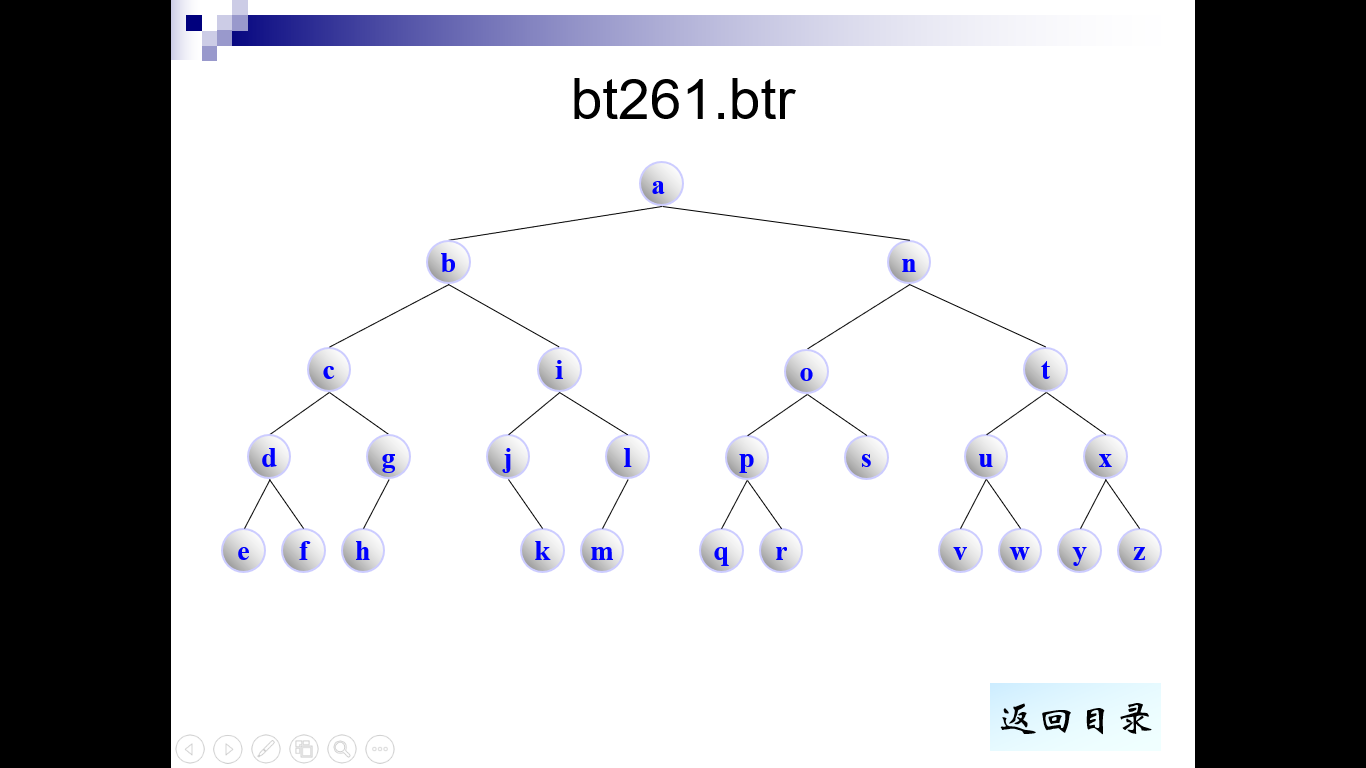
v 0 0

w 0 0

x 1 1

y 0 0

z 0 0



bt262.btr

BinaryTree

a 1 1

b 1 1

c 1 1

d 0 1

e 1 1

f 0 1

g 0 0

h 1 1

i 0 0

j 0 0

k 1 1

l 0 0

m 0 0

n 1 1

o 0 0

p 0 0

q 1 1

r 0 1

s 1 1

t 0 0

u 0 0

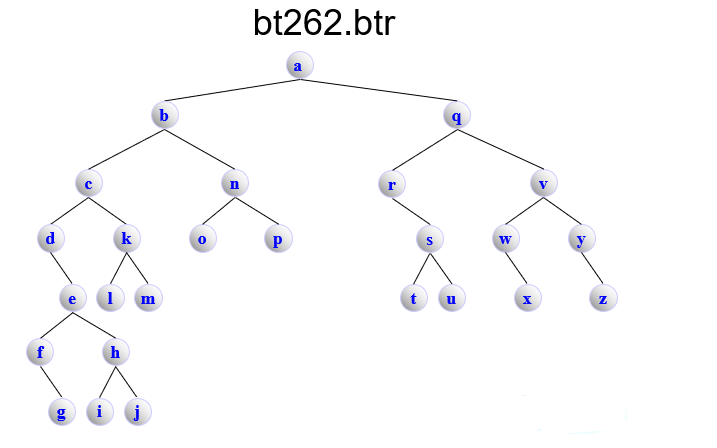
v 1 1

w 0 1

x 0 0

y 0 1

z 0 0



full4.btr

BinaryTree

a 1 1

b 1 1

c 1 1

d 0 0

e 0 0

f 1 1

g 0 0

h 0 0

i 1 1

j 1 1

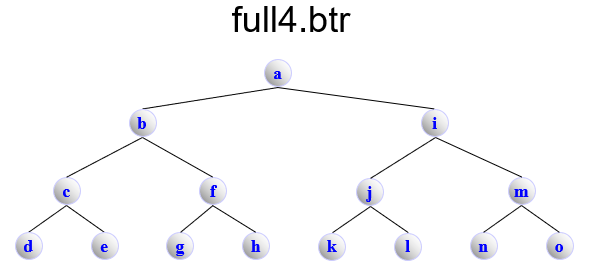
k 0 0

l 0 0

m 1 1

n 0 0

o 0 0



full5.btr

BinaryTree

a 1 1

b 1 1

c 1 1

d 1 1

e 0 0

f 0 0

g 1 1

h 0 0

i 0 0

j 1 1

k 1 1

l 0 0

m 0 0

n 1 1

o 0 0

p 0 0

q 1 1

r 1 1

s 1 1

t 0 0

u 0 0

v 1 1

w 0 0

x 0 0

y 1 1

z 1 1

1 0 0

2 0 0

3 1 1

4 0 0

5 0 0

