```cpp
/************************************************************
*                                                           *
*         A BINARY TREE PROGRAM BY JAMES WASHINGTON         *
*                                                           *
************************************************************/

#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

struct binaryTreeNode
{
    binaryTreeNode* leftSubtree;
    binaryTreeNode* rightSubtree;
    char key;
};
typedef binaryTreeNode* binaryTree;

void validateInput(string&);
void getPreorder(string&);
binaryTree newNode(char);
binaryTree buildTreeUtil(string, int&);
binaryTree buildTree(string);
void preorderTraversal(binaryTree, string&);
void inorderTraversal(binaryTree, string&);
void postorderTraversal(binaryTree, string&);
int height(binaryTree);
void getTreeLevel(binaryTree, int, string&);
void levelorderTraversal(binaryTree, string&);
bool isBSTUtil(binaryTree, binaryTree, binaryTree);
bool isBST(binaryTree);
int countNodes(binaryTree);
bool isFullBT(binaryTree);
bool isCompleteBT(binaryTree, unsigned int, unsigned int);
int pow(int, int);
void header();
void driver();

int main()
{
    header();
    driver();
}

/************************************************************
*       THIS FUNCTION VALIDATES THE INPUT OF THE USER       *
*                                                           *
```

```
50  **********************************************************/
51
52  void validateInput(string& input)
53  {
54      int size;
55      bool isValid = false;
56      bool check;
57      char ch;
58      string temp;
59
60      while (!isValid)
61      {
62          temp = "";
63
64          for (int i = 0; i < input.size(); i++)
65          {
66              if (input[i] != ' ')
67              {
68                  temp += input[i];
69              }
70          }
71
72          input = temp;
73          check = true;
74          size = input.size();
75
76          for (int i = 0; (i < size) && check; i++)
77          {
78              ch = input[i];
79
80              if ((ch < 'a' || ch > 'z') && (ch < 'A' || ch > 'Z') && (ch < '0'
                  || ch > '9') &&
81                  (ch < '!' || ch > '.') && (ch < ':' ||  ch > '@') && (ch < '['
                      || ch > '`'))
82              {
83                  cout << "Invalid input, please try again >> ";
84                  getline(cin, input);
85                  cout << endl;
86
87                  check = false;
88                  isValid = false;
89              }
90              else if (i == size - 1)
91              {
92                  isValid = true;
93              }
94          }
95      }
96  }
```

```
 97
 98 /*************************************************************
 99 *       THIS FUNCTION GETS THE INPUT FROM THE USER           *
100 *                                                           *
101 *************************************************************/
102
103 void getPreorder(string& preorder)
104 {
105     cout << "Please enter a preorder representation of a tree >> " ;
106     getline(cin, preorder);
107     cout << endl;
108
109     validateInput(preorder);
110 }
111
112 /*************************************************************
113 *    THIS FUNCTION CREATES A NEW NODE FOR THE BINARY TREE    *
114 *                                                           *
115 *************************************************************/
116
117 binaryTree newNode(char key)
118 {
119     binaryTree node = new binaryTreeNode;
120
121     node->key = key;
122     node->leftSubtree= NULL;
123     node->rightSubtree = NULL;
124
125     if (key == '.')
126     {
127         return NULL;
128     }
129
130     return node;
131 }
132
133 /*************************************************************
134 * THIS FUNCTION BUILDS A BINARY TREE FROM PREORDER TRAVERSAL *
135 *                                                           *
136 *************************************************************/
137
138 binaryTree buildTreeUtil(string input, int& index)
139 {
140     if (input.length() == index)
141     {
142         return NULL;
143     }
144     else
145     {
```

```
146         binaryTree node = newNode(input[index]);
147
148         if (input[index] != '.' && input[index + 1] == '.' && input[index + 2]
              == '.' && (input.length() - index == 3))
149         {
150             index = input.length();
151         }
152         else if (input[index] != '.' && input[index + 1] == '.' && input[index
              + 2] == '.')
153         {
154             index += 3;
155         }
156         else if (input[index] == '.')
157         {
158             index++;
159         }
160         else if (input[index] != '.')
161         {
162             index++;
163             node->leftSubtree = buildTreeUtil(input, index);
164             node->rightSubtree = buildTreeUtil(input, index);
165         }
166
167         return node;
168     }
169 }
170
171 /*************************************************************
172 * THIS FUNCTION USES THE UTILITY FUNCTIONT TO BUILD THE TREE *
173 *                                                           *
174 *************************************************************/
175
176 binaryTree buildTree(string input)
177 {
178     int index = 0;
179
180     return buildTreeUtil(input, index);
181 }
182
183 /*************************************************************
184 *         THIS FUNCTION GETS THE PREORDER FOR A TREE         *
185 *                                                           *
186 *************************************************************/
187
188 void preorderTraversal(binaryTree root, string& preorder)
189 {
190     if (root == nullptr)
191     {
192         return;
```

```
193     }
194
195     preorder += root->key;
196     preorderTraversal(root->leftSubtree, preorder);
197     preorderTraversal(root->rightSubtree, preorder);
198 }
199
200 /*************************************************************
201 *         THIS FUNCTION GETS THE INORDER FOR A TREE           *
202 *                                                             *
203 *************************************************************/
204
205 void inorderTraversal(binaryTree root, string& inorder)
206 {
207     if (root == nullptr)
208     {
209         return;
210     }
211
212     inorderTraversal(root->leftSubtree, inorder);
213     inorder += root->key;
214     inorderTraversal(root->rightSubtree, inorder);
215 }
216
217 /*************************************************************
218 *         THIS FUNCTION GETS THE POSTORDER FOR A TREE         *
219 *                                                             *
220 *************************************************************/
221
222 void postorderTraversal(binaryTree root, string& postorder)
223 {
224     if (root == nullptr)
225     {
226         return;
227     }
228
229     postorderTraversal(root->leftSubtree, postorder);
230     postorderTraversal(root->rightSubtree, postorder);
231     postorder += root->key;
232 }
233
234 /*************************************************************
235 *        THIS FUNCTION GETS THE LEVEL OF A GIVEN TREE         *
236 *                                                             *
237 *************************************************************/
238
239 void getTreeLevel(binaryTree root, int level, string& levelorder)
240 {
241     if (root == nullptr)
```

```
242     {
243         return;
244     }
245
246     if (level == 1)
247     {
248         levelorder += root->key;
249     }
250     else
251     {
252         getTreeLevel(root->leftSubtree, level - 1, levelorder);
253         getTreeLevel(root->rightSubtree, level - 1, levelorder);
254     }
255 }
256
257 /*************************************************************
258 *          THIS FUNCTION GETS THE HEIGHT FOR A TREE          *
259 *                                                            *
260 *************************************************************/
261
262 int height(binaryTree root)
263 {
264     if (root == nullptr)
265     {
266         return 0;
267     }
268     else
269     {
270         int leftHeight = height(root->leftSubtree);
271         int rightHeight = height(root->rightSubtree);
272
273         if (leftHeight > rightHeight)
274         {
275             return (leftHeight + 1);
276         }
277         else
278         {
279             return (rightHeight + 1);
280         }
281     }
282 }
283
284 /*************************************************************
285 *        THIS FUNCTION GETS THE LEVELORDER FOR A TREE        *
286 *                                                            *
287 *************************************************************/
288
289 void levelorderTraversal(binaryTree root, string& levelorder)
290 {
```

```
291     int numOfLevels = height(root);
292
293     for (int i = 1; i <= numOfLevels; i++)
294     {
295         getTreeLevel(root, i, levelorder);
296     }
297 }
298
299 /************************************************************
300 *          THIS FUNCTION DETERMINES IF A TREE IS A          *
301 *                     BINARY SEARCH TREE                    *
302 ************************************************************/
303
304 bool isBSTUtil(binaryTree root, binaryTree left, binaryTree right)
305 {
306     if (root == NULL)
307     {
308         return true;
309     }
310
311     if (left != NULL && root->key <= left->key)
312     {
313         return false;
314     }
315
316     if (right != NULL && root->key >= right->key)
317     {
318         return false;
319     }
320
321     return isBSTUtil(root->leftSubtree, left, root) && isBSTUtil(root-
      >rightSubtree, root, right);
322 }
323
324 /************************************************************
325 *          THIS FUNCTION DETERMINES IF A TREE IS A          *
326 *        BINARY SEARCH TREE USING THE UTILITY FUNCTION      *
327 ************************************************************/
328
329 bool isBST(binaryTree root)
330 {
331     return isBSTUtil(root, nullptr, nullptr);
332 }
333
334 /************************************************************
335 *      THIS FUNCTION GETS THE NUMBER OF NODES IN A TREE     *
336 *                                                           *
337 ************************************************************/
338
```

```
339 int countNodes(binaryTree root)
340 {
341     if (root == nullptr)
342     {
343         return 0;
344     }
345     else
346     {
347         return (countNodes(root->leftSubtree) + 1 + countNodes(root-
              >rightSubtree));
348     }
349 }
350
351 /*************************************************************
352 *          THIS FUNCTION DETERMINES IF A TREE IS A           *
353 *                      FULL BINARY TREE                      *
354 *************************************************************/
355
356 bool isFullBT(binaryTree root)
357 {
358     int h = height(root);
359     int numOfNodes = countNodes(root);
360
361     return (numOfNodes == (pow(2, h) - 1));
362 }
363
364 /*************************************************************
365 *          THIS FUNCTION DETERMINES IF A TREE IS A           *
366 *                     COMPLETE BINARY TREE                   *
367 *************************************************************/
368
369 bool isCompleteBT(binaryTree root, unsigned int index, unsigned int numOfNodes)
370 {
371     if (root == NULL)
372     {
373         return true;
374     }
375
376     if (index >= numOfNodes)
377     {
378         return false;
379     }
380
381     return (isCompleteBT(root->leftSubtree, 2 * index + 1, numOfNodes) &&
382             isCompleteBT(root->rightSubtree, 2 * index + 2, numOfNodes));
383 }
384
385 /*************************************************************
386 *        THIS FUNCTION WORKS AS AN EXPONENT FUNCTION         *
```

```
387 *                                                              *
388 ****************************************************************/
389
390 int pow(int base, int exp)
391 {
392     int ans = 1;
393
394     for (int i = 0; i < exp; i++)
395     {
396         ans *= base;
397     }
398
399     return ans;
400 }
401
402 /*************************************************************
403 *    THIS FUNCTION PRINTS OUT THE HEADER FOR THE PROGRAM      *
404 *                                                            *
405 *************************************************************/
406
407 void header()
408 {
409     cout << "/************************************************************** "
          << endl;
410     cout << "*                                                            * "
          << endl;
411     cout << "*                        BINARY TREE                         * "
          << endl;
412     cout << "*                                                            * "
          << endl;
413     cout << "**************************************************************/ "
          << endl << endl;
414
415     cout << "WELCOME USER!" << endl << endl;
416     cout << "# TO GET STARTED, ENTER A PREORDER REPRESENTATION OF A BINARY
          TREE." << endl;
417     cout << "# THE COMPUTER WILL THEN OUTPUT INFO ON THE TREE." << endl;
418     cout << "ENJOY THE PROGRAM!!!" << endl << endl;
419 }
420
421 /*************************************************************
422 *     THIS FUNCTION WORKS AS THE DRIVER FOR THE PROGRAM       *
423 *                                                            *
424 *************************************************************/
425
426 void driver()
427 {
428     string preorder;
429     string inorder;
```

```
430     string postorder;
431     string levelorder;
432     string input;
433     binaryTree tree;
434     bool isBst;
435     int numOfNodes;
436     int treeHeight;
437     bool isFull;
438     unsigned int index = 0;
439     bool isComplete;
440
441
442     getPreorder(input);
443     tree = buildTree(input);
444
445     preorderTraversal(tree, preorder);
446     inorderTraversal(tree, inorder);
447     postorderTraversal(tree, postorder);
448     levelorderTraversal(tree, levelorder);
449
450     isBst = isBST(tree);
451     numOfNodes = countNodes(tree);
452     treeHeight = height(tree);
453     isFull = isFullBT(tree);
454     isComplete = isCompleteBT(tree, index, numOfNodes);
455
456     cout << "TRAVERSALS" << endl << endl;
457     cout << "Preorder: " << preorder << endl << endl;
458     cout << "Inorder: " << inorder << endl << endl;
459     cout << "Postorder: " << postorder << endl << endl;
460     cout << "Levelorder: " << levelorder << endl << endl << endl;
461
462     cout << "GENERAL INFO" << endl << endl;
463     if (isBst)
464     {
465         cout << "This tree is a binary search tree." << endl << endl;
466     }
467     else
468     {
469         cout << "This tree is not a binary search tree." << endl << endl;
470     }
471
472     cout << "This tree has " << numOfNodes << " nodes." << endl << endl;
473     cout << "This tree has a height of " << treeHeight << "." << endl << endl;
474
475     if (isFull)
476     {
477         cout << "This tree is a full binary tree." << endl << endl;
478     }
```

```
479     else
480     {
481         cout << "This tree is not a full binary tree." << endl << endl;
482     }
483
484     if (isComplete)
485     {
486         cout << "This tree is a complete binary tree." << endl << endl;
487     }
488     else
489     {
490         cout << "This tree is not a complete binary tree." << endl << endl;
491     }
492 }
```