

main.cpp

```
1// Lina Kang
2// CS1D MW 2:30 - 5:00 PM
3// 09/23/2020
4// This program explores the concept of binary tree and
5// tests out its abilities through inserting items,
6// in-order, pre-order, post-order, breadth-first traversals
7// and hierarchical relationships
8
9// ----- OUTPUT -----
10/*
11 Lina Kang
12 CS1D MW 2:30 - 5:00 PM
13 09/23/2020
14 This program explores the concept of binary tree and
15 tests out its abilities through inserting items,
16 in-order, pre-order, post-order, breadth-first traversals
17 and hierarchical relationships
18
19
20**In-Order Traversal**
21
22 5 12 13 18 19 24 25 29 33 44 49 55 59 77 89 109 118 288 1001
23
24**Pre-Order Traversal**
25
26 89 59 19 13 5 12 18 25 24 33 29 44 55 49 77 288 109 118 1001
27
28**Post-Order Traversal**
29
30 12 5 18 13 24 29 49 55 44 33 25 19 77 59 118 109 1001 288 89
31
32**Breadth-First Traversal**
33
34 89 59 288 19 77 109 1001 13 25 118 5 18 24 33 12 29 44 55 49
35
36**Print By Level**
37
38Level 0: 89
39Level 1: 59 288
40Level 2: 19 77 109 1001
41Level 3: 13 25 118
42Level 4: 5 18 24 33
43Level 5: 12 29 44
44Level 6: 55
45Level 7: 49
46
47**Print Relationships of Nodes**
48
49Node: 5
50 - Parent: 13
51 - Children: 12
52
53Node: 12
54 - Parent: 5
55
56Node: 13
57 - Parent: 19
```

```
58 - Children: 5 18
59
60Node: 18
61 - Parent: 13
62
63Node: 19
64 - Parent: 59
65 - Children: 13 25
66
67Node: 24
68 - Parent: 25
69
70Node: 25
71 - Parent: 19
72 - Children: 24 33
73
74Node: 29
75 - Parent: 33
76
77Node: 33
78 - Parent: 25
79 - Children: 29 44
80
81Node: 44
82 - Parent: 33
83 - Children: 55
84
85Node: 49
86 - Parent: 55
87
88Node: 55
89 - Parent: 44
90 - Children: 49
91
92Node: 59
93 - Parent: 89
94 - Children: 19 77
95
96Node: 77
97 - Parent: 59
98
99Node: 89
100 - Children: 59 288
101
102Node: 109
103 - Parent: 288
104 - Children: 118
105
106Node: 118
107 - Parent: 109
108
109Node: 288
110 - Parent: 89
111 - Children: 109 1001
112
113Node: 1001
114 - Parent: 288
```

```

115 */
116 // ----- END OUTPUT -----
117
118 #include "binaryTree.h"
119
120 int main()
121 {
122     // Input & Initialization
123     NodeBinaryTree binaryTree;
124
125     binaryTree.insert(89);
126
127     Node * root = binaryTree.getroot();
128
129     binaryTree.insert(59, root);
130     binaryTree.insert(288, root);
131     binaryTree.insert(19, root);
132     binaryTree.insert(13, root);
133     binaryTree.insert(5, root);
134     binaryTree.insert(109, root);
135     binaryTree.insert(12, root);
136     binaryTree.insert(118, root);
137     binaryTree.insert(25, root);
138     binaryTree.insert(33, root);
139     binaryTree.insert(1001, root);
140     binaryTree.insert(18, root);
141     binaryTree.insert(44, root);
142     binaryTree.insert(77, root);
143     binaryTree.insert(55, root);
144     binaryTree.insert(24, root);
145     binaryTree.insert(49, root);
146     binaryTree.insert(29, root);
147
148
149
150     // Output & Processing
151     cout << " Lina Kang\n"
152            " CS1D MW 2:30 - 5:00 PM\n"
153            " 09/23/2020\n"
154            " This program explores the concept of binary tree and\n"
155            " tests out its abilities through inserting items,\n"
156            " in-order, pre-order, post-order, breadth-first traversals\n"
157            " and hierarchical relationships \n\n";
158
159     cout << "\n**In-Order Traversal**\n\n";
160
161     binaryTree.in_order(root);
162
163     cout << "\n\n**Pre-Order Traversal**\n\n";
164
165     binaryTree.pre_order(root);
166
167     cout << "\n\n**Post-Order Traversal**\n\n";
168
169     binaryTree.post_order(root);
170
171     cout << "\n\n**Breadth-First Traversal**\n\n";

```

```

172
173     binaryTree.breadth_first(root);
174
175     cout << "\n\n**Print By Level**\n\n";
176
177     binaryTree.printByLevel(root);
178
179     cout << "\n\n**Print Relationships of Nodes**\n\n";
180
181     binaryTree.printRelation(root);
182 }
183
184 #ifndef BINARYTREE_H_
185 #define BINARYTREE_H_
186
187 #include <vector>
188 #include <queue>
189 #include <iostream>
190
191 using namespace std;
192
193 struct Node
194 {
195     int value;
196     int level;
197     Node * left;
198     Node * right;
199     Node * parent;
200     Node() : value(0), level(0), left(NULL), right(NULL), parent(NULL) { }
201 };
202
203 //linked-list implementation
204 class NodeBinaryTree
205 {
206 public:
207     NodeBinaryTree();
208     int getsize() const;
209     bool isempty() const;
210     Node * getroot() const;
211
212     void insert(int);
213     void insert(int, Node*);
214
215     bool isExternal(Node * v) const { return v->left == NULL && v->right == NULL; }
216     bool isInternal(Node * v) const { return v->left != NULL || v->right != NULL; }
217
218     void in_order(Node*) const;
219     void post_order(Node*) const;
220     void pre_order(Node*) const;
221     void breadth_first(Node*) const;
222
223     void printByLevel(Node *) const;
224     void printRelation(Node *) const;
225
226 private:
227     Node *root;
228     Node *current;

```

```

229
230     int size;
231 };
232 NodeBinaryTree::NodeBinaryTree()
233 {
234     root = NULL;
235     current = NULL;
236     size = 0;
237 }
238 int NodeBinaryTree::getsize() const
239 {
240     return size;
241 }
242 bool NodeBinaryTree::isempty() const
243 {
244     return root == NULL ? true : false;
245 }
246 Node * NodeBinaryTree::getroot() const
247 {
248     return root;
249 }
250 // insert function for ONLY root
251 void NodeBinaryTree::insert(int item)
252 {
253     if(root!=NULL)
254         insert(item, root);
255     else
256     {
257         Node * newNode = new Node;
258         root = newNode;
259         root->value=item;
260         root->left=NULL;
261         root->right=NULL;
262         size++;
263     }
264 }
265 // insert function for descendants of root (recursive)
266 void NodeBinaryTree::insert(int item, Node * node)
267 {
268     // compare the item to current node
269     if(item < node->value)
270     {
271         //recursively repeat until an empty spot is found to insert
272         if(node->left != NULL)
273             insert(item, node->left);
274         else
275         {
276             Node * newNode = new Node;
277
278             node->left=newNode;
279
280             newNode->value = item;
281             newNode->left = NULL;
282             newNode->right = NULL;
283             newNode->parent = node;
284
285             size++;

```

```

286     }
287 }
288 else if(item >= node->value)
289 {
290     //recursively repeat until an empty spot is found to insert
291     if(node->right != NULL)
292         insert(item, node->right);
293     else
294     {
295         Node * newNode = new Node;
296
297         node->right = newNode;
298
299         newNode->value = item;
300         newNode->left = NULL;
301         newNode->right = NULL;
302         newNode->parent = node;
303
304         size++;
305     }
306 }
307 }
308
309 // Prints the binary tree from leftmost node to rightmost node
310 // (the output is from smallest to largest in increasing order)
311 void NodeBinaryTree::in_order(Node * p) const
312 {
313     if(p->left != NULL)
314         in_order(p->left);
315     cout << " " << p->value;
316     if(p->right != NULL)
317         in_order(p->right);
318 }
319
320 // Prints the binary tree where nodes are visited after its descendants
321 void NodeBinaryTree::post_order(Node * p) const
322 {
323     if(p->left != NULL)
324         post_order(p->left);
325     if(p->right != NULL)
326         post_order(p->right);
327     cout << " " << p->value;
328 }
329
330 // Prints the binary tree where nodes are visited before its descendants
331 void NodeBinaryTree::pre_order(Node * p) const
332 {
333     cout << " " << p->value;
334     if(p->left != NULL)
335         pre_order(p->left);
336     if(p->right != NULL)
337         pre_order(p->right);
338 }
339
340 // Prints the binary tree at a top-down approach visiting nodes by level
341 void NodeBinaryTree::breadth_first(Node * p) const
342 {

```

```

343     Node * current;
344
345     queue<Node *> que;           // utilize a queue to add nodes
346     que.push(p);               // and add the descendants
347
348     while(que.size() > 0)
349     {
350         // get first value from queue, print that value, pop the value
351         current = que.front();
352         que.pop();
353         cout << " " << current->value;
354
355         // enqueue descendants if they exist
356         if(current->left != NULL)
357             que.push(current->left);
358         if(current->right != NULL)
359             que.push(current->right);
360     }
361 }
362
363 // Prints the binary trees divided by levels and nodes within those levels
364 // (similar algorithm from breadth_first traversal)
365 void NodeBinaryTree::printByLevel(Node * p) const
366 {
367     Node * current;
368
369     queue<Node *> que;
370     que.push(p);
371
372     int level = 0;
373     cout << "Level " << level << ": ";
374
375     while(que.size() > 0)
376     {
377         current = que.front();
378         que.pop();
379         cout << current->value << " ";
380
381         if(current->left != NULL)
382         {
383             // designate a level to each node
384             // (since current->left is a child of current, their level is
385             // 1 + current's level)
386             current->left->level = current->level + 1;
387             que.push(current->left);
388         }
389         if(current->right != NULL)
390         {
391             current->right->level = current->level + 1;
392             que.push(current->right);
393         }
394     }
395
396     // to prevent the next if condition from bugging out when size == 0
397     if(que.size() == 0)
398         break;
399     // if the next node in queue is starting at next level,

```

```

400     // print a new line and print "Level"
401     else if(current->level < que.front()->level )
402     {
403         level++;
404         cout << endl << "Level " << level << ": ";
405     }
406
407 }
408 cout << endl;
409 }
410 // Print the relationship of the nodes
411 // (same algorithm from in-order traversal)
412 void NodeBinaryTree::printRelation(Node * p) const
413 {
414     if(p->left != NULL)
415         printRelation(p->left);
416
417     // ----- Output -----
418     cout << "Node: " << p->value;
419
420     if(p != root)
421         cout << "\n - Parent: " << p->parent->value;
422     if(!isExternal(p))
423     {
424         cout << "\n - Children: ";
425         if(p->left != NULL)
426             cout << p->left->value << " ";
427         if(p->right != NULL)
428             cout << p->right->value;
429     }
430     cout << endl << endl;
431     // ----- End Output -----
432
433     if(p->right != NULL)
434         printRelation(p->right);
435 }
436
437 #endif /* BINARYTREE_H_ */
438
439

```