```cpp
/*****************************************************************************
 * AUTHOR           : Nick Reardon
 * Assignment #5    : Binary Trees
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 02 / 19 / 20
 *****************************************************************************/
#ifndef _LINKEDBINARYTREE_H_
#define _LINKEDBINARYTREE_H_
#include <exception>
#include <sstream>
#include <string>
#include <queue>
#include "Except.h"

enum ERROR_TYPE
{
    DEFUALT,
    FULL,
    EMPTY
};

struct Trunk
{
    Trunk* prev;
    std::string str;

    Trunk(Trunk* prev, std::string str)
    {
        this->prev = prev;
        this->str = str;
    }
};

template <class Type>
struct Node
{
    Type value;

    Node<Type>* parent;

    Node<Type>* left;
    Node<Type>* right;

    Node<Type>(const Type& newValue, Node<Type>* parentNode, Node<Type>* leftNode,
Node<Type>* rightNode)
    {
        value = newValue;

        parent = parentNode;

        left = leftNode;
        right = rightNode;
    }

    Node<Type>(const Type& newValue, Node<Type>* leftNode, Node<Type>* rightNode)
    {
        value = newValue;
```

```
58
59          left = leftNode;
60          right = rightNode;
61      }
62
63  };
64
65  template <class Type>
66  class LinkedBinaryTree
67  {
68  private:
69      Node<Type>* root;
70
71      int capacity;
72      int currentSize;
73
74
75  protected:
76      void insertRecursion(const Type& newValue, Node<Type>* node)
77      {
78          if (newValue == node->value)
79          {
80              Node<Type>* tempPtr = new Node<Type>(newValue, node, node->left, nullptr);
81
82              if (node->left != nullptr)
83              {
84                  node->left->parent = tempPtr;
85              }
86
87              node->left = tempPtr;
88
89              tempPtr = nullptr;
90
91          }
92          else if (newValue > node->value)
93          {
94              if (node->right == nullptr)
95              {
96                  node->right = new Node<Type>(newValue, node, nullptr, nullptr);
97              }
98              else
99              {
100                 insertRecursion(newValue, node->right);
101             }
102         }
103         else
104         {
105             if (node->left == nullptr)
106             {
107                 node->left = new Node<Type>(newValue, node, nullptr, nullptr);
108             }
109             else
110             {
111                 insertRecursion(newValue, node->left);
112             }
113         }
114     }
115
```

```
116     void destroyRecursion(Node<Type>* node)
117     {
118         if (node != nullptr)
119         {
120             destroyRecursion(node->left);
121             destroyRecursion(node->right);
122             delete node;
123         }
124     }
125 
126     Type* searchRecursion(const Type& searchValue, const Node<Type>* node) const
127     {
128         if (searchValue == node->value)
129         {
130             return node;
131         }
132         else
133         {
134             if (searchValue > node->value)
135             {
136                 if (searchValue == node->right->value)
137                 {
138                     return node->right;
139                 }
140                 else
141                 {
142                     searchRecursion(searchValue, node->right);
143                 }
144             }
145             else
146             {
147                 if (searchValue == node->left->value)
148                 {
149                     return node->left;
150                 }
151                 else
152                 {
153                     searchRecursion(searchValue, node->left);
154                 }
155             }
156         }
157     }
158 
159     void InOrder_Recursion(const Node<Type>* node, std::ostream& output) const
160     {
161         if (root == nullptr)
162         {
163             throw(Except_runtime_error_class("Tree is empty - Nothing to print", EMPTY,
    5));
164         }
165         else if(node != nullptr)
166         {
167 
168             InOrder_Recursion(node->left, output);
169 
170             output << node->value << "   ";
171 
172             InOrder_Recursion(node->right, output);
173 
```

```
174         }
175     }
176 
177     void PostOrder_Recursion(const Node<Type>* node, std::ostream& output) const
178     {
179         if (root == nullptr)
180         {
181             throw(Except_runtime_error_class("Tree is empty - Nothing to print", EMPTY,
    5));
182         }
183         if (node != nullptr)
184         {
185             PostOrder_Recursion(node->left, output);
186 
187             PostOrder_Recursion(node->right, output);
188 
189             output << node->value << "   ";
190         }
191 
192     }
193 
194     void PreOrder_Recursion(const Node<Type>* node, std::ostream& output) const
195     {
196         if (root == nullptr)
197         {
198             throw(Except_runtime_error_class("Tree is empty - Nothing to print", EMPTY,
    5));
199         }
200         if (node != nullptr)
201         {
202             output << node->value << "   ";
203 
204             PreOrder_Recursion(node->left, output);
205 
206             PreOrder_Recursion(node->right, output);
207 
208         }
209     }
210 
211     void BreadthFirst_Recursion(std::queue<Node<Type>*>& queue, const Node<Type>* node,
    std::ostream& output) const
212     {
213         if (root == nullptr)
214         {
215             throw(Except_runtime_error_class("Tree is empty - Nothing to print", EMPTY,
    5));
216         }
217         if (node != nullptr)
218         {
219             output << node->value << "   ";
220 
221             if (node->left != nullptr)
222             {
223                 queue.push(node->left);
224             }
225 
226             if (node->right != nullptr)
227             {
228                 queue.push(node->right);
```

```
            }

            queue.pop();

            if (!queue.empty())
            {
                BreadthFirst_Recursion(queue, queue.front(), output);
            }
        }
    }



    void Print2DUtil(const Node<Type>* branch, int space, std::ostream& output) const
    {
        // Base case
        if (branch == NULL)
            return;

        // Increase distance between levels
        space += 10;

        // Process right child first
        Print2DUtil(branch->right, space, output);

        // Print current node after space
        // count
        output << '\n';
        for (int i = 10; i < space; i++)
        {
            output << " ";
        }

        output << branch->value << "\n";

        // Process left child
        Print2DUtil(branch->left, space, output);
    }

    // Helper function to print branches of the binary tree
    void showTrunks(Trunk* p, std::ostream& output)
    {
        if (p == nullptr)
            return;

        showTrunks(p->prev, output);

        output << p->str;
    }

    // Recursive function to print binary tree
    // It uses inorder traversal
    void printTreeHelper(Node<Type>* node, Trunk* prev, bool isLeft, std::ostream& output)
    {
        if (node == nullptr)
            return;

        std::string prev_str = "    ";
```

```
287         Trunk* trunk = new Trunk(prev, prev_str);
288 
289         printTreeHelper(node->right, trunk, true, output);
290 
291         if (!prev)
292             trunk->str = "---";
293         else if (isLeft)
294         {
295             trunk->str = ".---";
296             prev_str = "    |";
297         }
298         else
299         {
300             trunk->str = "`---";
301             prev->str = prev_str;
302         }
303 
304         showTrunks(trunk, output);
305         output << node->value << '\n';
306 
307         if (prev)
308             prev->str = prev_str;
309         trunk->str = "    |";
310 
311         printTreeHelper(node->left, trunk, false, output);
312     }
313 
314 
315 public:
316 
317     LinkedBinaryTree<Type>()
318     {
319         root = nullptr;
320         currentSize = 0;
321     }
322 
323     //LinkedBinaryTree<Type>(const LinkedBinaryTree<Type>& otherTree);
324 
325     ~LinkedBinaryTree()
326     {
327         destroy();
328     }
329 
330     void destroy()
331     {
332         destroyRecursion(root);
333         root = nullptr;
334         currentSize = 0;
335     }
336 
337     //bool empty() const;
338     //bool full() const;
339     //int size() const;
340 
341     void insert(const Type& newValue)
342     {
343         if (root == nullptr)
344         {
```

```
345             root = new Node<Type>(newValue, nullptr, nullptr, nullptr);
346         }
347         else
348         {
349             insertRecursion(newValue, root);
350         }
351 
352         currentSize++;
353     }
354 
355     Type* search(const Type& searchValue) const
356     {
357         search(searchValue, root);
358     }
359 
360     void printAll(std::ostream& output) const;
361 
362     void Traversal_InOrder(std::ostream& output) const
363     {
364         output << "In Order Traversal:" << '\n';
365         InOrder_Recursion(root, output);
366         output << "\n\n";
367     }
368 
369 
370     void Traversal_PostOrder(std::ostream& output) const
371     {
372         output << "Post Order Traversal:" << '\n';
373         PostOrder_Recursion(root, output);
374         output << "\n\n";
375     }
376 
377     void Traversal_PreOrder(std::ostream& output) const
378     {
379         output << "Pre Order Traversal:" << '\n';
380         PreOrder_Recursion(root, output);
381         output << "\n\n";
382     }
383 
384     void Traversal_BreadthFirst(std::ostream& output) const
385     {
386         std::queue<Node<Type>*> queue;
387 
388         queue.push(root);
389 
390         output << "Breadth First Traversal:" << '\n';
391         BreadthFirst_Recursion(queue, root, output);
392         output << "\n\n";
393     }
394 
395     void Print2D(std::ostream& output) const
396     {
397         Print2DUtil(root, 0, output);
398     }
399 
400 
401 
402     void printTree(std::ostream& output) const
```

```
403     {
404         printTreeHelper(root, nullptr, false, output);
405         output << "\n\n";
406     }
407 
408     void PrintLevelByLevel(std::ostream& output) const
409     {
410         if (root == nullptr)
411         {
412             throw(Except_runtime_error_class("Tree is empty - Nothing to print", EMPTY,
    5));
413         }
414 
415         output << "Level By Level:" << '\n';
416 
417         std::queue<Node<Type>*> queue;
418 
419         int levelNodes = 0;
420 
421         queue.push(root);
422 
423         while (!queue.empty())
424 {
425 
426             levelNodes = queue.size();
427 
428             while (levelNodes > 0)
429             {
430                 Node<Type>* temp = queue.front();
431 
432                 output << temp->value << "   ";
433 
434                 if (temp->left != nullptr)
435                 {
436                     queue.push(temp->left);
437                 }
438 
439                 if (temp->right != nullptr)
440                 {
441                     queue.push(temp->right);
442                 }
443 
444                 queue.pop();
445 
446                 levelNodes--;
447             }
448             output << '\n';
449         }
450 
451         output << "\n";
452     }
453 
454 
455 
456 };
457 
458 
459 #endif // !_LINKEDBINARYTREE_H_
460 
```