

cs280s21-b.sg

[Dashboard](#) / [My courses](#) / [cs280s21-b.sg](#) / [General](#) / [Assignment 3: AVL Trees](#)

- [Description](#)
- [Submission](#)
- [Edit](#)
- Submission view

Submitted on Sunday, March 14, 2021, 1:00 AM ([Download](#))

Automatic evaluation[-]

Proposed grade: 100 / 100

Compilation[-]

Running code
Executable successfully created

Comments[-]

[+][Summary of tests](#)

AVLTree.h

```
1  /*****  
2  /*!  
3  \file:      AVLTree.h  
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119  
5  \par email: weizhe.goh@digipen.edu  
6  \date:      March 12, 2021  
7  \brief      This file contains the declarations needed to implement the simple  
8  .....      API for AVL Trees with recursive algorithms.  
9  
10 Copyright (C) 2021 DigiPen Institute of Technology.  
11 Reproduction or disclosure of this file or its contents without the  
12 prior written consent of DigiPen Institute of Technology is prohibited.  
13 */  
14 /*****  
15  
16 //-----  
17 #ifndef AVLTREE_H  
18 #define AVLTREE_H  
19 //-----  
20 #include <stack>  
21 #include "BSTree.h"  
22  
23 /*!  
24 Definition for the AVL Tree  
25 */  
26 template <typename T>  
27 class AVLTree : public BSTree<T>  
28 {  
29 public:  
30     AVLTree(ObjectAllocator *OA = 0, bool ShareOA = false);  
31     virtual ~AVLTree() = default; // DO NOT IMPLEMENT  
32     virtual void insert(const T& value) override;  
33     virtual void remove(const T& value) override;  
34  
35     ..... // Returns true if efficiency implemented  
36     static bool ImplementedBalanceFactor(void);  
37  
38 private:  
39     // private stuff  
40  
41     unsigned int node_count(typename BSTree<T>::BinTree& tree) const;  
42  
43     void insert_start(typename BSTree<T>::BinTree& tree, const T& value);  
44     void insert_node(typename BSTree<T>::BinTree& node, const T& value,  
45     ..... std::stack<typename BSTree<T>::BinTree*>& nodes);  
46  
47     void remove_start(typename BSTree<T>::BinTree& tree, const T& value);  
48     void remove_node(typename BSTree<T>::BinTree &tree, const T& value,  
49     ..... std::stack<typename BSTree<T>::BinTree*>& nodes);  
50  
51  
52     void RotateLeft(typename BSTree<T>::BinTree& tree);  
53     void RotateRight(typename BSTree<T>::BinTree& tree);  
54  
55     void BalanceAVLTree(std::stack<typename BSTree<T>::BinTree*>& nodes);  
56 };  
57  
58 #include "AVLTree.cpp"  
59  
60 #endif  
61 //-----  
62
```

AVLTree.cpp

```

1  /*****
2  /*!
3  \file:      AVLTree.cpp
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119
5  \par email: weizhe.goh@digipen.edu
6  \date:      March 12, 2021
7  \brief      This file contains the definitions needed to implement the simple
8               API for AVL Trees with recursive algorithms.
9
10 Copyright (C) 2021 DigiPen Institute of Technology.
11 Reproduction or disclosure of this file or its contents without the
12 prior written consent of DigiPen Institute of Technology is prohibited.
13 */
14 /*****
15
16 #include "AVLTree.h"
17
18 /*****
19 /*!
20 \fn      template<typename T>
21           AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA):
22           BSTree<T>(OA, ShareOA){}
23
24 \brief    Constructor of AVLTree
25
26 \param    OA - Object allocator for the BSTree nodes
27
28 \param    ShareOA - boolean for sharing object allocator
29 */
30 /*****
31 template<typename T>
32 AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA):BSTree<T>(OA, ShareOA){}
33
34 /*****
35 /*!
36 \fn      template<typename T>
37           void AVLTree<T>::insert(const T& value)
38
39 \brief    Insert a node into the tree
40
41 \param    value - The value of the node to be inserted
42 */
43 /*****
44 template<typename T>
45 void AVLTree<T>::insert(const T& value)
46 {
47     insert_start(BSTree<T>::get_root(), value);
48     node_count(BSTree<T>::get_root());
49 }
50
51 /*****
52 /*!
53 \fn      template <typename T>
54           void AVLTree<T>::insert_start(typename BSTree<T>::BinTree& tree,
55           const T& value)
56
57 \brief    Function to start the recursion to insert a node into the tree
58
59 \param    tree - The tree that the node to be inserted
60
61 \param    value - The value of the node to be inserted
62 */
63 /*****
64 template <typename T>
65 void AVLTree<T>::insert_start(typename BSTree<T>::BinTree& tree, const T& value)
66 {
67     std::stack<typename BSTree<T>::BinTree*> stack_;
68
69     insert_node(tree, value, stack_);
70 }
71
72 /*****
73 /*!
74 \fn      template <typename T>
75           void AVLTree<T>::insert_node(typename BSTree<T>::BinTree & node,
76           const T& value, std::stack<typename BSTree<T>::BinTree*> & nodes)
77
78 \brief    Helper function to insert a node into the tree by recursion
79
80 \param    node - The tree or subtree for node to be inserted
81
82 \param    value - The value of the node to be inserted
83
84 \param    nodes - The stack to push or pop the nodes
85 */
86 /*****
87 template <typename T>
88 void AVLTree<T>::insert_node(typename BSTree<T>::BinTree & node, const T& value,
89 std::stack<typename BSTree<T>::BinTree*> & nodes)
90 {
91     if(node == 0)
92     {
93         node = BSTree<T>::make_node(value);
94         BalanceAVLTree(nodes);
95     }
96     else if (value < node->data)
97     {
98         nodes.push(&node);
99         insert_node(node->left, value, nodes);
100    }
101    else if (value > node->data)
102    {
103        nodes.push(&node);
104        insert_node(node->right, value, nodes);
105    }
106    else
107        std::cout << "Error, duplicate item" << std::endl;

```

```

108 }
109
110 /*****
111  */
112 \fn      template <typename T>
113           unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree& tree)
114           const
115
116 \brief   Count the number of nodes in a tree or subtree
117
118 \param  tree - The tree to count the number of total nodes in that tree.
119
120 \return Return the total node count of a tree
121 */
122 /*****
123  */
124 template <typename T>
125 unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree& tree) const
126 {
127     if(tree == NULL)
128         return 0;
129
130     tree->count = 1 + node_count(tree->left) + node_count(tree->right);
131
132     return tree->count;
133 }
134 /*****
135  */
136 \fn      template <typename T>
137           void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree& tree)
138
139 \brief   Rotate the tree around the node to the left
140
141 \param  tree - The node to rotate about
142 */
143 /*****
144  */
145 template <typename T>
146 void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree& tree)
147 {
148     typename BSTree<T>::BinTree temp = tree;
149     tree = tree->right;
150     temp->right = tree->left;
151     tree->left = temp;
152
153     tree->count = temp->count;
154
155     //recount node count
156     unsigned leftCount = (temp->left) ? temp->left->count : 0;
157     unsigned rightCount = (temp->right) ? temp->right->count : 0;
158
159     temp->count = leftCount + rightCount + 1;
160 }
161 /*****
162  */
163 \fn      template <typename T>
164           void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree& tree)
165
166 \brief   Rotate the tree around the node to the right
167
168 \param  tree - The node to rotate about
169 */
170 /*****
171  */
172 template <typename T>
173 void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree& tree)
174 {
175     typename BSTree<T>::BinTree temp = tree;
176     tree = tree->left;
177     temp->left = tree->right;
178     tree->right = temp;
179
180     tree->count = temp->count;
181
182     //recount node count
183     unsigned leftCount = (temp->left) ? temp->left->count : 0;
184     unsigned rightCount = (temp->right) ? temp->right->count : 0;
185
186     temp->count = leftCount + rightCount + 1;
187 }
188 /*****
189  */
190 \fn      template <typename T>
191           void AVLTree<T>::BalanceAVLTree
192           (std::stack<typename BSTree<T>::BinTree*>& nodes)
193
194 \brief   Balance the AVL tree
195
196 \param  nodes - The stack of nodes used to balance the tree
197 */
198 /*****
199  */
200 template <typename T>
201 void AVLTree<T>::BalanceAVLTree(std::stack<typename BSTree<T>::BinTree*>& nodes)
202 {
203     while(!nodes.empty())
204     {
205         typename BSTree<T>::BinTree* topnode = nodes.top();
206         nodes.pop();
207
208         typename BSTree<T>::BinTree& y = *topnode;
209
210         int RH = BSTree<T>::tree_height(y->right);
211         int LH = BSTree<T>::tree_height(y->left);
212
213         if(abs(LH - RH) > 1)
214         {
215             if(RH > (LH + 1))
216             {

```

```

215         //promote twice
216         if(BSTree<T>::tree_height(y->right->left) >
217            BSTree<T>::tree_height(y->right->right))
218         {
219             RotateRight(y->right);
220             RotateLeft(y);
221         }
222         else
223         {
224             //promote once
225             RotateLeft(y);
226         }
227     }
228     else if ((RH + 1) < LH)
229     {
230         //promote once
231         if(BSTree<T>::tree_height(y->left->left) >
232            BSTree<T>::tree_height(y->left->right))
233         {
234             RotateRight(y);
235         }
236         else
237         {
238             //promote twice
239             RotateLeft(y->left);
240             RotateRight(y);
241         }
242     }
243 }
244 }
245 }
246 /*****
247  *!
248  \fn      template <typename T>
249            void AVLTree<T>::remove(const T& value)
250
251  \brief   Remove a node in the tree
252
253  \param   value - The value of the node to be removed
254  */
255 /*****
256  template <typename T>
257  void AVLTree<T>::remove(const T& value)
258  {
259      remove_start(BSTree<T>::get_root(), value);
260      node_count(BSTree<T>::get_root());
261  }
262
263 /*****
264  *!
265  \fn      template <typename T>
266            void AVLTree<T>::remove_start(typename BSTree<T>::BinTree& tree,
267            const T& value)
268
269  \brief   Function to begin the recursion to remove node from the tree
270
271  \param   tree - The tree to remove the node from
272
273  \param   value - The value of the node to be removed
274  */
275 /*****
276  template <typename T>
277  void AVLTree<T>::remove_start(typename BSTree<T>::BinTree& tree, const T& value)
278  {
279      std::stack<typename BSTree<T>::BinTree*> stack_;
280      remove_node(tree, value, stack_);
281  }
282
283 /*****
284  *!
285  \fn      template<typename T>
286            void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree,
287            const T& value, std::stack<typename BSTree<T>::BinTree*>& nodes)
288
289  \brief   Helper function to remove the node to delete by recursion
290
291  \param   tree - The tree to remove the node from
292
293  \param   value - The value of the node to be removed
294
295  \param   nodes - The stack to pop out the nodes from
296  */
297 /*****
298  template<typename T>
299  void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree, const T& value,
300      std::stack<typename BSTree<T>::BinTree*>& nodes)
301  {
302      if(tree == 0)
303          return ;
304      else if (value < tree->data)
305      {
306          nodes.push(&tree);
307          remove_node(tree->left, value, nodes);
308      }
309      else if (value > tree->data)
310      {
311          nodes.push(&tree);
312          remove_node(tree->right, value, nodes);
313      }
314      else
315      {
316          if(tree->left == 0)
317          {
318              typename BSTree<T>::BinTree temp = tree;
319              tree = tree->right;
320              BSTree<T>::free_node(temp);
321              BalanceAVLTree(nodes);

```

```
322     }
323     else if (tree->right == 0)
324     {
325         typename BSTree<T>::BinTree temp = tree;
326         tree = tree->left;
327         BSTree<T>::free_node(temp);
328         BalanceAVLTree(nodes);
329     }
330     else
331     {
332         //two child
333         typename BSTree<T>::BinTree pred = 0;
334         BSTree<T>::find_predecessor(tree, pred);
335         tree->data = pred->data;
336         nodes.push(&tree);
337         remove_node(tree->left, tree->data, nodes);
338     }
339 }
340 }
341
342 /*****
343  *!
344  \fn      template<typename T>
345           bool AVLTree<T>::ImplementedBalanceFactor(void)
346
347  \brief   Function for efficient balancing
348
349  \return  Returns true if implemented efficient balancing, else return false
350  */
351 /*****
352  template<typename T>
353  bool AVLTree<T>::ImplementedBalanceFactor(void)
```

BSTree.h

```

1  /*****
2  /*!
3  \file:      BSTree.h
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119
5  \par email: weizhe.goh@digipen.edu
6  \date:      March 12, 2021
7  \brief      This file contains the declarations needed to implement the simple
8               API for Binary Search Tree with recursive algorithms.
9
10 Copyright (C) 2021 DigiPen Institute of Technology.
11 Reproduction or disclosure of this file or its contents without the
12 prior written consent of DigiPen Institute of Technology is prohibited.
13 */
14 /*****
15
16 //-----
17 #ifndef BSTREE_H
18 #define BSTREE_H
19 //-----
20 #include <string>      // std::string
21 #include <stdexcept>  // std::exception
22
23 #include "ObjectAllocator.h"
24
25 /*!
26 The exception class for the AVL/BST classes
27 */
28 class BSTException : public std::exception
29 {
30 public:
31     /*!
32     Non-default constructor
33
34     \param ErrCode
35     The kind of exception (only one currently)
36
37     \param Message
38     The human-readable reason for the exception.
39     */
40     BSTException(int ErrCode, const std::string& Message) :
41         error_code_(ErrCode), message_(Message) {
42     };
43
44     /*!
45     Retrieve the exception code.
46
47     \return
48     E_NO_MEMORY
49     */
50     virtual int code() const {
51         return error_code_;
52     }
53
54     /*!
55     Retrieve the message string
56
57     \return
58     The human-readable message.
59     */
60     virtual const char *what() const throw() {
61         return message_.c_str();
62     }
63
64     ///! Destructor
65     virtual ~BSTException() {}
66
67     ///! The kinds of exceptions (only one currently)
68     enum BST_EXCEPTION{E_NO_MEMORY};
69
70 private:
71     int error_code_;      ///!< The code of the exception
72     std::string message_; ///!< Readable message text
73 };
74
75 /*!
76 The definition of the BST
77 */
78 template <typename T>
79 class BSTree
80 {
81 public:
82     ///! The node structure
83     struct BinTreeNode
84     {
85         BinTreeNode *left;  ///!< The left child
86         BinTreeNode *right; ///!< The right child
87         T data;             ///!< The data
88         int balance_factor; ///!< optional for efficient balancing
89         unsigned count;     ///!< nodes in this subtree for efficient indexing
90         ///! Default constructor
91         BinTreeNode() : left(0), right(0), data(0), balance_factor(0), count(1){};
92
93         ///! Conversion constructor
94         BinTreeNode(const T& value) :
95             left(0), right(0), data(value), balance_factor(0), count(1) {};
96     };
97
98     ///! shorthand
99     typedef BinTreeNode* BinTree;
100
101     BSTree(ObjectAllocator *OA = 0, bool ShareOA = false);
102     BSTree(const BSTree& rhs);
103     virtual ~BSTree();
104     BSTree& operator=(const BSTree& rhs);
105     const BinTreeNode* operator[](int index) const; //for r-values (Extra Credit)
106     virtual void insert(const T& value);
107     virtual void remove(const T& value);

```

```
108     void clear();
109     bool find(const T& value, unsigned &compares) const;
110     bool empty() const;
111     unsigned int size() const;
112     int height() const;
113     BinTree root() const;
114
115 protected:
116     BinTree& get_root();
117     BinTree make_node(const T& value) const;
118     void free_node(BinTree node);
119     int tree_height(BinTree tree) const;
120     void find_predecessor(BinTree tree, BinTree &predecessor) const;
121
122 private:
123     // private stuff...
124
125     ObjectAllocator* oa;
126
127     bool Custom_OA;
128     bool share;
129     BinTree root_;
130
131     BinTree copy_tree(BinTree& destination, const BinTree& source);
132     void free_tree(BinTree& root);
133     void delete_node(BinTree& tree, const T& value);
134     const BinTreeNode* sub_node(BinTree tree, int compares) const;
135     void insert_node(BinTree& tree, const T& value);
136     bool find_node (BinTree tree, const T& value, unsigned& compares) const;
137 };
138
139 #include "BSTree.cpp"
140
141 #endif
142 //-----
```

BSTree.cpp


```

1  /*****
2  /*!
3  \file:      BSTree.cpp
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119
5  \par email: weizhe.goh@digipen.edu
6  \date:      March 12, 2021
7  \brief      This file contains the definitions needed to implement the simple
8               API for Binary Search Tree with recursive algorithms.
9
10 Copyright (C) 2021 DigiPen Institute of Technology.
11 Reproduction or disclosure of this file or its contents without the
12 prior written consent of DigiPen Institute of Technology is prohibited.
13 */
14 /*****/
15
16 #include "BSTree.h"
17
18 /*****/
19 /*!
20 \fn      template<typename T>
21           BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA)
22
23 \brief    Constructor of BSTree
24
25 \param    OA - Object allocator for the BSTree nodes
26
27 \param    ShareOA - boolean for sharing object allocator
28 */
29 /*****/
30 template<typename T>
31 BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA): oa{OA}, share{ShareOA},
32 root_{0}
33 {
34     if(OA)
35         Custom_OA = false;
36     else
37     {
38         OAConfig config(true);
39         oa = new ObjectAllocator(sizeof(BinTreeNode), config);
40         Custom_OA = true;
41     }
42 }
43
44 /*****/
45 /*!
46 \fn      template<typename T>
47           BSTree<T>::BSTree(const BSTree& rhs)
48
49 \brief    Copy constructor of BSTree
50
51 \param    rhs - The BSTree to be copied from
52 */
53 /*****/
54 template<typename T>
55 BSTree<T>::BSTree(const BSTree& rhs)
56 {
57     if(rhs.share)
58     {
59         oa = rhs.oa;
60         Custom_OA = false;
61         share = true;
62     }
63     else
64     {
65         OAConfig config(true);
66         oa = new ObjectAllocator(sizeof(BinTreeNode), config);
67         Custom_OA = true;
68         share = false;
69     }
70
71     copy_tree(root_, rhs.root_);
72
73     root_>count = rhs.root()->count;
74 }
75
76 /*****/
77 /*!
78 \fn      template<typename T>
79           BSTree<T>::~~BSTree()
80
81 \brief    Destructor of BSTree
82 */
83 /*****/
84 template<typename T>
85 BSTree<T>::~~BSTree()
86 {
87     clear();
88
89     //false
90     if(!share)
91         delete oa;
92 }
93
94 /*****/
95 /*!
96 \fn      template<typename T>
97           BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)
98
99 \brief    Assignment operator of BSTree
100
101 \param    rhs - The BSTree to be copied from
102
103 \return Returns the assigned tree itself
104 */
105 /*****/
106 template<typename T>
107 BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)

```



```

108 {
109
110     if(this == &rhs)
111         return *this;
112
113     if(rhs.share)
114     {
115         oa = rhs.oa;
116         Custom_OA = false;
117         share = true;
118     }
119     else
120     {
121         OACconfig config(true);
122         delete oa;
123         oa = new ObjectAllocator(sizeof(BinTreeNode), config);
124         Custom_OA = true;
125         share = false;
126     }
127
128     if(rhs.root())
129     {
130         clear();
131         copy_tree(root_, rhs.root_);
132     }
133
134     return *this;
135 }
136
137 /*****
138  *!
139  \fn      template <typename T>
140           typename BSTree<T>::BinTreeNode BSTree<T>::copy_tree(BinTreeNode& destination,
141           const BinTreeNode& source)
142
143  \brief   Helper function to construct copy tree by recursion
144
145  \param   destination - copied BSTree
146
147  \param   source - BSTree to be copied from
148  */
149 /*****
150  template <typename T>
151  typename BSTree<T>::BinTreeNode BSTree<T>::copy_tree(BinTreeNode& destination,
152  const BinTreeNode& source)
153  {
154
155      if(!source)
156          return nullptr;
157
158      destination = make_node(source->data);
159
160      destination->balance_factor = source->balance_factor;
161
162      destination->count = source->count;
163
164      destination->left = copy_tree(destination->left, source->left);
165      destination->right = copy_tree(destination->right, source->right);
166
167      return destination;
168  }
169
170 /*****
171  *!
172  \fn      template<typename T>
173           const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index)
174           const
175
176  \brief   subscript operator to search in the tree
177
178  \param   index - the index to match
179
180  \return  Returns the matching node
181  */
182 /*****
183  template<typename T>
184  const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index) const
185  {
186      return sub_node(root_, index);
187  }
188
189 /*****
190  *!
191  \fn      template <typename T>
192           const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTreeNode tree,
193           int compares) const
194
195  \brief   Helper function to find the correct node by recursion
196
197  \param   tree - The tree to find the match
198
199  \param   compares - the value to match
200
201  \return  Returns the matching node
202  */
203 /*****
204  template <typename T>
205  const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTreeNode tree,
206  int compares) const
207  {
208      if(!tree)
209          return NULL;
210
211      unsigned temp = (tree->left) ? tree->left->count : 0;
212
213      if(temp > static_cast<unsigned>(compares))
214          return sub_node(tree->left, compares);

```

```

215         else if (temp < static_cast<unsigned>(compares))
216             return sub_node(tree->right, compares - temp - 1);
217         else
218             return tree;
219     }
220
221     /*****
222     /*!
223     \fn      template <typename T>
224             void BSTree<T>::insert(const T& value)
225
226     \brief   Insert a node into the tree
227
228     \param   value - the value of node to be inserted
229     */
230     /*****/
231     template <typename T>
232     void BSTree<T>::insert(const T& value)
233     {
234         insert_node(root_, value);
235     }
236
237     /*****
238     /*!
239     \fn      template<typename T>
240             void BSTree<T>::insert_node(BinTree& tree, const T& value)
241
242     \brief   Helper function to insert a node into tree by recursion
243
244     \param   tree - The tree for node to be inserted
245
246     \param   value - the value of node to be inserted
247     */
248     /*****/
249     template<typename T>
250     void BSTree<T>::insert_node(BinTree& tree, const T& value)
251     {
252         try
253         {
254             if(!tree)
255             {
256                 tree = make_node(value);
257             }
258             else if (value < tree->data)
259             {
260                 ++tree->count;
261                 insert_node(tree->left, value);
262             }
263             else if (value > tree->data)
264             {
265                 ++tree->count;
266                 insert_node(tree->right, value);
267             }
268             else
269             {
270                 std::cout << "Error, duplicated item" << std::endl;
271             }
272         }
273         catch(const OException& e)
274         {
275             throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
276         }
277     }
278
279     /*****
280     /*!
281     \fn      template <typename T>
282             void BSTree<T>::remove(const T& value)
283
284     \brief   Remove node based on input value
285
286     \param   value - The value of node to be removed
287     */
288     /*****/
289     template <typename T>
290     void BSTree<T>::remove(const T& value)
291     {
292         delete_node(root_, value);
293     }
294
295     /*****
296     /*!
297     \fn      template<typename T>
298             void BSTree<T>::clear()
299
300     \brief   Clear the tree's nodes
301     */
302     /*****/
303     template<typename T>
304     void BSTree<T>::clear()
305     {
306         free_tree(root_);
307     }
308
309     /*****
310     /*!
311     \fn      template<typename T>
312             void BSTree<T>::free_tree(BinTree& tree)
313
314     \brief   Helper function to clear tree nodes by recursion
315
316     \param   tree - The tree to be freed
317     */
318     /*****/
319     template<typename T>
320     void BSTree<T>::free_tree(BinTree& tree)
321     {

```

```

322     if(!tree)
323     {
324         return;
325     }
326     free_tree(tree->left);
327     free_tree(tree->right);
328     delete_node(tree, tree->data);
329 }
330 /*****
331  *!
332  \fn      template<typename T>
333           void BSTree<T>::delete_node(BinTree& tree, const T& value)
334
335  \brief   Helper function to delete node
336
337  \param   tree - the tree to remove node from
338
339  \param   value - The value of node to be removed
340  */
341 /*****
342  template<typename T>
343  void BSTree<T>::delete_node(BinTree& tree, const T& value)
344  {
345      if(tree == NULL)
346      {
347          return;
348      }
349      else if (value < tree->data)
350      {
351          --tree->count;
352          delete_node(tree->left, value);
353      }
354      else if (value > tree->data)
355      {
356          --tree->count;
357          delete_node(tree->right, value);
358      }
359      else
360      {
361          --tree->count;
362
363          if(tree->left == 0)
364          {
365              BinTree temp = tree;
366              tree = tree->right;
367              free_node(temp);
368          }
369          else if (tree->right == 0)
370          {
371              BinTree temp = tree;
372              tree = tree->left;
373              free_node(temp);
374          }
375          else
376          {
377              BinTree pred = 0;
378              find_predecessor(tree, pred);
379              tree->data = pred->data;
380              delete_node(tree->left, tree->data);
381          }
382      }
383  }
384 }
385 /*****
386  *!
387  \fn      template<typename T>
388           bool BSTree<T>::find(const T& value, unsigned& compares) const
389
390  \brief   Find the node in the tree with the matching value
391
392  \param   value - the value of node to be found
393
394  \param   compares - the number of function calls used to find the matching node
395
396  \return  Return true if node with matching value exist, else return false
397  */
398 /*****
399  template<typename T>
400  bool BSTree<T>::find(const T& value, unsigned& compares) const
401  {
402      return find_node(root_, value, compares);
403  }
404 }
405 /*****
406  *!
407  \fn      template<typename T>
408           bool BSTree<T>::find_node(BinTree tree, const T& value,
409                                     unsigned& compares) const
410
411  \brief   Helper function to find a node in the tree with a matching value by
412           recursion
413
414  \param   tree - the tree to be searched
415
416  \param   value - the value of node to be found
417
418  \param   compares - the number of function calls used to find the matching node
419
420  \return  Return true if node with matching value exist, else return false
421  */
422 /*****
423  template<typename T>
424  bool BSTree<T>::find_node(BinTree tree, const T& value, unsigned& compares) const
425  {
426      ++compares;
427
428      if(tree == 0)
429      {
430          return false;
431      }
432      else if(value == tree->data)
433      {
434          return true;
435      }
436      else if(value < tree->data)
437      {
438          return find_node(tree->left, value, compares);
439      }
440      else if(value > tree->data)
441      {
442          return find_node(tree->right, value, compares);
443      }
444  }
445 }

```

```

429     else if(value < tree->data)
430         return find_node(tree->left, value, compares);
431     else
432         return find_node(tree->right, value, compares);
433 }
434
435 /*****
436  *!
437  \fn      template<typename T>
438           bool BSTree<T>::empty() const
439
440  \brief   Check if the tree is empty
441
442  \return  Return true if tree is empty, else return false
443  */
444 /*****
445  template<typename T>
446  bool BSTree<T>::empty() const
447  {
448      return (root_ == 0);
449  }
450
451 /*****
452  *!
453  \fn      template<typename T>
454           unsigned int BSTree<T>::size() const
455
456  \brief   Counts the number of nodes in the tree
457
458  \return  Returns the number of nodes in the tree
459  */
460 /*****
461  template<typename T>
462  unsigned int BSTree<T>::size() const
463  {
464      return (root_) ? root_->count : 0;
465  }
466
467 /*****
468  *!
469  \fn      template<typename T>
470           int BSTree<T>::height() const
471
472  \brief   Counts the height of a tree
473
474  \return  Returns the height of a tree
475  */
476 /*****
477  template<typename T>
478  int BSTree<T>::height() const
479  {
480      return tree_height(root_);
481  }
482
483 /*****
484  *!
485  \fn      template<typename T>
486           typename BSTree<T>::BinTree BSTree<T>::root() const
487
488  \brief   Get the root of the tree
489
490  \return  Returns the root of the tree
491  */
492 /*****
493  template<typename T>
494  typename BSTree<T>::BinTree BSTree<T>::root() const
495  {
496      return root_;
497  }
498
499 /*****
500  *!
501  \fn      template <typename T>
502           typename BSTree<T>::BinTree& BSTree<T>::get_root()
503
504  \brief   Get the root of the tree
505
506  \return  Returns the root of the tree
507  */
508 /*****
509  template <typename T>
510  typename BSTree<T>::BinTree& BSTree<T>::get_root()
511  {
512      return root_;
513  }
514
515 /*****
516  *!
517  \fn      template <typename T>
518           typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) const
519
520  \brief   Make a new node for the tree
521
522  \param   value - the value of the new node
523
524  \return  Returns the new node
525  */
526 /*****
527  template <typename T>
528  typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) const
529  {
530      try
531      {
532          BinTree memory = reinterpret_cast<BinTreeNode*>(oa->Allocate());
533          BinTree new_node = new (memory) BinTreeNode(value);
534          return new_node;
535      }

```

```
536         catch(const OException &e)
537     {
538         throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
539     }
540 }
541
542 /*****
543  *!
544  \fn      template <typename T>
545           void BSTree<T>::free_node(BinTree node)
546
547  \brief   Free a node from the tree
548
549  \param   node - The node to be freed
550  */
551 /*****
552  template <typename T>
553  void BSTree<T>::free_node(BinTree node)
554  {
555      node->~BinTreeNode();
556      oa->Free(node);
557  }
558
559 /*****
560  *!
561  \fn      template <typename T>
562           int BSTree<T>::tree_height(BinTree tree) const
563
564  \brief   Helper function to find the height of a tree by recursion
565
566  \param   tree - The tree to be counted
567
568  \return  Returns the height of the tree
569  */
570 /*****
571  template <typename T>
572  int BSTree<T>::tree_height(BinTree tree) const
573  {
574      if(tree == 0)
575          return -1;
576
577      int L = tree_height(tree->left);
578      int R = tree_height(tree->right);
579
580      if(L > R)
581          return L + 1;
582      else
583          return R + 1;
584  }
585
586 /*****
587  *!
588  \fn      template <typename T>
589           void BSTree<T>::find_predecessor(BinTree tree, BinTree& predecessor)
590           const
591
592  \brief   Finds the parent of a node
593
594  \param   tree - the node to be searched
595
596  \param   predecessor - the node to fill as predecessor
597  */
598 /*****
599  template <typename T>
600  void BSTree<T>::find_predecessor(BinTree tree, BinTree& predecessor) const
601  {
```