```
1  /*****************************************************************************/
2  /*!
3  \fn template <typename T> AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA) : BSTree<T>(OA
4  \brief
5  Constructor of the AVLTree
6
7  \param OA
8  object allocator for the tree's nodes
9
10 \param ShareOA
11 boolean for sharing object allcoator among trees
12
13 \return
14 none
15
16 */
17 /*****************************************************************************/
18
19 template <typename T>
20 AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA) : BSTree<T>(OA, ShareOA) {}
21
22 /*****************************************************************************/
23 /*!
24 \fn template <typename T> AVLTree<T>::~AVLTree()
25 \brief
26 Destructor of AVLTree

27 \return
28 none
29 */
30 /*****************************************************************************/
```

```
31
32  template <typename T>
33  AVLTree<T>::~AVLTree() {}
34
35  /*************************************************************************/
36  /*!
37  \fn template <typename T> void AVLTree<T>::insert(const T& value) throw(BSTException)
38  \brief
39  insert a node into the tree
40
41  \param value
42  the value of node to be inserted
43
44  \return
45  none
46  */
47  /*************************************************************************/
48
49  template <typename T>
50  void AVLTree<T>::insert(const T& value) throw(BSTException) {
51
52          insert_begin(BSTree<T>::get_root(), value);
53          node_count(BSTree<T>::get_root());
54  }
55
56  /*************************************************************************/
57  /*!
58  \fn template <typename T> void AVLTree<T>::insert_begin(typename BSTree<T>::BinTree &tree, cons
59  \brief
60  the function to begin the recursion process to insert node
61
62  \param tree
63  the value of node to be inserted
64
65  \param value
66  the value of node to be inserted
67
68  \return
69  none
70  */
71  /*************************************************************************/
72
73  template <typename T>
74  void AVLTree<T>::insert_begin(typename BSTree<T>::BinTree &tree, const T& value) {
75          std::stack<typename BSTree<T>::BinTree*> stack_;
76          insert_node(tree, value, stack_);
77
78  }
79
80  /*************************************************************************/
81  /*!
82  \brief
```

```
83  helper function insert a node into the tree by recursion
84
85  \param node
86  the tree/subtree for node to be inserted
87
88  \param value
89  the value of node to be inserted
90
91  \param nodes
92  the stack to push/pop nodes
93
94  \return
95  none
96  */
97  /*************************************************************************/
98
99  template <typename T>
100 void AVLTree<T>::insert_node(typename BSTree<T>::BinTree & node, const T& value, std::stack<ty
101         if (node == 0) {
102                 node = BSTree<T>::make_node(value);
103                 BalanceAVLTree(nodes);
104         }
105         else if (value < node->data) {
106                 nodes.push(&node);
107                 insert_node(node->left, value, nodes);
108         }
109         else if (value > node->data) {
110                 nodes.push(&node);
111                 insert_node(node->right, value, nodes);
112         }
113         else
114                 std::cout << "Error, duplicate item" << std::endl;
115 }
116
117 /*************************************************************************/
118 /*!
119 \fn template <typename T> void AVLTree<T>::remove(const T& value)
120 \brief
121 remove a node in the tree
122
123 \param value
124 the value of node to be removed
125
126 \return
127 none
128 */
129 /*************************************************************************/
130
131 template <typename T>
132 void AVLTree<T>::remove(const T& value) {
133         remove_begin(BSTree<T>::get_root(), value);
134         node_count(BSTree<T>::get_root());
```

```
135 }
136
137 /*************************************************************************/
138 /*!
139 \fn template <typename T> void AVLTree<T>::remove_begin(typename BSTree<T>::BinTree &tree, cons
140 \brief
141 the function to begin the recursion process to remove node
142
143 \param tree
144 the value of node to be removed
145
146 \param value
147 the value of node to be removed
148
149 \return
150 none
151 */
152 /*************************************************************************/
153
154 template <typename T>
155 void AVLTree<T>::remove_begin(typename BSTree<T>::BinTree &tree, const T& value) {
156         std::stack<typename BSTree<T>::BinTree *> stack_;
157         remove_node(tree, value, stack_);
158 }
159
160 /*************************************************************************/
161 /*!
162 \fn template<typename T> void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree, const
163 \brief
164 helper function to remove for the correct node to delete by recursion.
165
166 \param tree
167 the tree to remove node from
168
169 \param value
170 the value to match for removal
171
172 \param nodes
173 the stack deal with
174
175 \return
176 none
177 */
178 /*************************************************************************/
179
180 template<typename T>
181 void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree, const T& value,
182         std::stack<typename BSTree<T>::BinTree*>& nodes) {
183
184         if (tree == 0)
185                 return;
186
```

```
187         else if (value < tree->data){
188                 nodes.push(&tree);
189                 remove_node(tree->left, value, nodes);
190         }
191         else if (value > tree->data){
192                 nodes.push(&tree);
193                 remove_node(tree->right, value, nodes);
194         }
195         else{ // match fit
196 
197                 if (tree->left == 0){
198                         typename BSTree<T>::BinTree temp = tree;
199                         tree = tree->right;
200                         BSTree<T>::free_node(temp);
201                 }
202                 else if (tree->right == 0){
203                         typename BSTree<T>::BinTree temp = tree;
204                         tree = tree->left;
205                         BSTree<T>::free_node(temp);
206                 }
207                 else{ // if node has 2 children
208                         typename BSTree<T>::BinTree pred = 0;
209                         BSTree<T>::find_predecessor(tree, pred);
210                         tree->data = pred->data;
211                         remove_node(tree->left, tree->data, nodes);
212                         BalanceAVLTree(nodes);
213                 }
214         }
215 }
216 
217 /*************************************************************************/
218 /*!
219 \fn template <typename T> void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree & node)
220 \brief
221 rotate the tree around the node to the left
222 
223 \param node
224 the node to rotate about
225 
226 \return
227 none
228 */
229 /*************************************************************************/
230 
231 template <typename T>
232 void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree & node) {
233 
234         typename BSTree<T>::BinTree temp = node;
235         node = node->right;
236         temp->right = node->left;
237         node->left = temp;
238 
```

```
        node->count = temp->count;
        //recount node count
        unsigned leftCount = (temp->left) ? temp->left->count : 0;
        unsigned rightCount = (temp->right) ? temp->right->count : 0;

        temp->count = leftCount + rightCount + 1;
}

/*****************************************************************************/
/*!
\fn template <typename T> void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree & node)
\brief
rotate the tree around the node to the right

\param node
the node to rotate about

\return
none
*/
/*****************************************************************************/

template <typename T>
void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree & node) {

                typename BSTree<T>::BinTree temp = node;
                node = node->left;
                temp->left = node->right;
                node->right = temp;

                node->count = temp->count;
                //recount node count
                unsigned leftCount = (temp->left) ? temp->left->count : 0;
                unsigned rightCount = (temp->right) ? temp->right->count : 0;

                temp->count = leftCount + rightCount + 1;
}

/*****************************************************************************/
/*!
\fn template <typename T> void AVLTree<T>::BalanceAVLTree(std::stack<typename BSTree<T>::BinTre
\brief
balances the avl tree

\param nodes
the stack of nodes used to balance the tree

\return
none
*/
/*****************************************************************************/

```

```
template <typename T>
void AVLTree<T>::BalanceAVLTree(std::stack<typename BSTree<T>::BinTree*> & nodes) {
        while (!nodes.empty()) {
                typename BSTree<T>::BinTree * topnode = nodes.top();
                nodes.pop();

                typename BSTree<T>::BinTree &node = *topnode;

                int RH = BSTree<T>::tree_height(node->right);
                int LH = BSTree<T>::tree_height(node->left);

                if (RH > (LH + 1)) {
                        //promote twice
                        if (BSTree<T>::tree_height(node->right->left) > BSTree<T>::tree_height(
                                RotateRight(node->right);
                                RotateLeft(node);
                        }
                        else
                                RotateLeft(node); //promote once
                }
                else
                if ((RH + 1)< LH ) {
                        //promote once
                        if (BSTree<T>::tree_height(node->left->left) > BSTree<T>::tree_height(n
                                RotateRight(node);
                                node_count(BSTree<T>::get_root());
                        }
                        else { //promote twice
                                RotateLeft(node->left);
                                RotateRight(node);
                        }
                }

        }
}

/*************************************************************************/
/*!
\fn template <typename T> unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree & tre
\brief
extra credit for efficient balancing

\return
total node count of a tree
*/
/*************************************************************************/

template <typename T>
unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree & tree) const {
        if (tree == NULL)
                return 0;

```

```
343         tree->count = 1 + node_count(tree->left) + node_count(tree->right);
344 
345         return tree->count;
346 }
347 
348 /*****************************************************************************/
349 /*!
350 \fn template <typename T> bool AVLTree<T>::ImplementedBalanceFactor(void)
351 \brief
352 extra credit for efficient balancing
353 
354 \return
355 none
356 */
357 /*****************************************************************************/
358 
359 template <typename T>
360 bool AVLTree<T>::ImplementedBalanceFactor(void) { return false; }
```