```cpp
// BST.cpp
#include "BST.h"


/*********************************************************************
clear()
Parameters: Node*&
Complexity: O(n)

Private method to recursively delete every node from the tree.
Complexity is O(n) because it has to remove every
*********************************************************************/
void BST::clear(Node*& r){
    if (r != nullptr) { //If the node exists
        clear(r->left); //clear the left subtree
        clear(r->right);//clear the right subtree
        delete r;       //delete the current node
        r = nullptr;    //remove dangling points
    }
}

/*********************************************************************
insertAux()
Parameters: Node*&, int
Complexity: O(n)

Private method that uses recursion to insert an int into the tree.
It's passed a pointer to a node and an int to be inserted.
Complexity is O(n) because the tree may not be balanced.
*********************************************************************/
void BST::insertAux(Node*& r, int v) {
    if (r == nullptr)   //If the passed node doesn't exist
        r = new Node(v); //make a new one from the heap containing the
                         //passed int

    else {               // If the passed node does exist
        r->size++;       // increment the size value of this node

        if (v < r->val)             // If the passed int is less than the
            insertAux(r->left, v);  // value of this node, go to the left
                                    //subtree

        else                        // if the passed int is greater than or
            insertAux(r->right, v); // equal to the value of this node, go
    }                               // to the left subtre
}

/*********************************************************************
removeAux()
Parameters: Node*&, int
Complexity: O(n)

```

```
53  Private method that uses recursion to remove a node from the tree.
54  It's passed a pointer to a node and an int to be removed.
55  Complexity is O(n) because the tree may not be balanced.
56  ***************************************************************/
57  void BST::removeAux(Node*& r, int v) {
58      if (r == nullptr)   // If the node is not found
59          return;
60
61      else{
62          if (v < r->val) {           // If v is less than the value of the node,
63              r->size--;              // decrment the size of nodes along the
64              removeAux(r->left, v);  // path and go left.
65          }
66          else
67              if (v > r->val) {           // If v is less than the value of the
68                  r->size--;              // node, decrement size and go right.
69                  removeAux(r->right, v);
70
71              }
72              else //The node to be removed has been found and r points to it.
73
74          // Two children
75                  if (r->left != nullptr && r->right != nullptr) {
76                     // Find the successor by going right and all the way left
77                      Node* temp = r->right;  // temp points to node to the right
78                      Node* back = r->right;  // back points to node to the right
79
80                      while (temp->left != nullptr) {   // go all the way left
81                          back = temp;         // back points to previous node
82                          temp = temp->left; // temp points to node to the left
83                      }
84                      //when left is nullptr, temp points to successor node, copy
85                      //the successor node's value and size to it's replacement
86                      r->val = temp->val;
87                      r->size = temp->size;
88
89                      // If we did not go left at all just splice temp node out
90                      if (r->right == temp) {
91                          r->right = temp->right;
92                          delete temp;
93                      }
94                      else {//we did go left so splice out temp here
95                          back->left = temp->right;
96                          delete temp;
97                      }
98                  }
99          //  Node has less than two children.
100                 else {
101                     if (r->left != nullptr) {
102                         Node* t = r;// new r is sent back thru ref parameter
103                         r = r->left;
104                         delete t;
```

```
105                     }
106                     else {// r->right is not null or it is.
107                         Node* t = r; //t points to node to be deleted
108                         r = r->right;   // r either points to right subnode
109                                         // or nullptr
110                         // new r is sent back thru ref parameter
111                         delete t;
112                     }
113                 }
114 
115     }
116     return;
117 }
118 
119 /*********************************************************************
120 inOrderPrintAux()
121 Parameters: Node*&, ostream&
122 Complexity: O(n)
123 
124 Private method that uses recursion to print the value of every node int the
125 tree in ascending order.
126 It's passed a pointer to a node and an ostream type that determines how it
127 outputs the information.
128 Complexity is O(n) because it has to go to and print every node in the
129 tree.
130 *********************************************************************/
131 void BST::inOrderPrintAux(Node* r, ostream& os)const {
132     if (r != nullptr) {
133         inOrderPrintAux(r->left, os);// print left sub tree
134         os << r->val << " ";
135         inOrderPrintAux(r->right, os);//print right sub tree
136     }
137 
138 }
139 
140 /*********************************************************************
141 numNodesAux()
142 Parameters: Node*&
143 Complexity: O(n)
144 
145 Private method that uses recursion to return the total number of nodes in the
146 tree.
147 It's passed a pointer to a node.
148 Complexity is O(n) because the method has to go to and count every node in
149 the tree.
150 *********************************************************************/
151 int BST::numNodesAux(Node*& r)const {
152     if (r == nullptr)
153         return 0;
154     else
155         return 1 + numNodesAux(r->left) + numNodesAux(r->right);
156 }
```

```
157
158
159 /*****************************************************************
160 searchAux()
161 Parameters: Node*&, int
162 Complexity: O(n)
163
164 Private method that uses recursion to search for a node in the tree and return
165 true or false depending on whether it's found.
166 It's passed a pointer to a node and an int to be searched for.
167 Complexity is O(n) because the tree may not be balanced.
168 *****************************************************************/
169 bool BST::searchAux(Node*& r, int v)const {
170     if (r == nullptr) {             //v is not found
171         return false;
172     }
173
174     if (v < r->val)
175         searchAux(r->left, v);      //go left
176
177     else if (v > r->val)
178         searchAux(r->right, v);     //go right
179     else                            // v is found
180         return true;
181
182 }
183
184 /*****************************************************************
185 rankAux()
186 Parameters: Node*&, two ints
187 Complexity: O(n)
188
189 Private method that uses recursion to return an integer's rank in the tree.
190 It's passed a pointer to a node and two ints. Int v is the int who's rank
191 will be determined. Int rank keeps track of the rank of v through the
192 recursive calls.
193 The rank is its position in the sorted tree. For example, the smallest
194 int in a tree would be rank 1. The largest would be equal to the number
195 of nodes in the tree. If the integer is not found in the tree, it
196 returns 0.
197 Complexity is O(n) because the method may have to go through every node in
198 the tree.
199 *****************************************************************/
200 int BST::rankAux(Node*& r, int v, int rank) {
201     if (r == nullptr)   //If v is not found, it's rank is 0
202         return 0;
203
204     if (v < r->val)
205         rankAux(r->left, v, rank);  // go left
206     else if (v > r->val) {
207         if (r->left == nullptr)     //if the left subnode doesn't exist,
208             rank++;                 //increment size.
```

```
209         else
210             rank += (1 + r->left->size);//add the size of the left subtree +1
211         rankAux(r->right, v, rank);     // to rank and go right
212     }
213     else {                          //v is found. If v has a left subnode, add
214         if (r->left != nullptr)     //it's size to rank.
215             rank += r->left->size;
216         return rank;
217     }
218 }
219
220 /*********************************************************************
221 rangeAux()
222 Parameters: Node*&, two ints
223 Complexity: O(n)
224
225 Private method that uses recursion to return the amount of nodes in a certain
226 range. The range is the number of nodes in the tree that are greater than or
227 equal to the first argument (i), and less than the second (j).
228 Complexity is O(n) because the method may have to go through every node in
229 the tree.
230 *********************************************************************/
231 int BST::rangeAux(Node*& r, int i, int j) {
232 // if the arguments create an impossible range or a node is nullptr, return 0
233     if (i >= j || r == nullptr)
234         return 0;
235
236     if (r->val >= i && r->val < j)  //the node is in the range between i and j.
237                                     //Start counting until it's not in range.
238         return 1 + rangeAux(r->left, i, j) + rangeAux(r->right, i, j);
239     else if (r->val >= i)
240         return rangeAux(r->left, i, j); // go left until r->val is < j
241     else if (r->val < j)
242         return rangeAux(r->right, i, j); // go right until r->val is >= i
243     else
244         return 0;
245 }
246
247
248
249
250
```