```
#include "btNode.h"

void dumpToArrayInOrder(btNode* bst_root, int* dumpArray)
{
   if (bst_root == 0) return;
   int dumpIndex = 0;
   dumpToArrayInOrderAux(bst_root, dumpArray, dumpIndex);
}

void dumpToArrayInOrderAux(btNode* bst_root, int* dumpArray, int& dumpIndex)
{
   if (bst_root == 0) return;
   dumpToArrayInOrderAux(bst_root->left, dumpArray, dumpIndex);
   dumpArray[dumpIndex++] = bst_root->data;
   dumpToArrayInOrderAux(bst_root->right, dumpArray, dumpIndex);
}

void tree_clear(btNode*& root)
{
   if (root == 0) return;
   tree_clear(root->left);
   tree_clear(root->right);
   delete root;
   root = 0;
}

int bst_size(btNode* bst_root)
{
   if (bst_root == 0) return 0;
   return 1 + bst_size(bst_root->left) + bst_size(bst_root->right);
}

// write definition for bst_insert here
void bst_insert(btNode*& bst_root, int insInt){

   if(bst_root == 0) // If Tree is empty, add insInt to root
   {
      btNode* temp_root = new btNode;
      temp_root->data = insInt;
      temp_root->left = temp_root->right = 0;
      bst_root = temp_root;
      return;
   }

   btNode* cursor = bst_root; // Creating cursor for traversing tree

   while (cursor != 0) // If tree not empty
   {
      if (cursor->data > insInt) // If insInt smaller, insert left
      {
         if(cursor->left == 0) // If left empty, insert insInt
         {
            cursor->left = new btNode;
            cursor->left->data = insInt;
            cursor->left->left = cursor->left->right = 0;
            return;
         }
         else // Continue to next left node
         {
            cursor = cursor->left;
         }
      }
      else if(cursor->data < insInt) // If insInt smaller, insert left
      {
         if(cursor->right == 0) // If left empty, insert insInt
         {
            cursor->right = new btNode;
            cursor->right->data = insInt;
            cursor->right->left = cursor->right->right = 0;
```

```
70              return;
71           }
72           else // Continue to next right node
73           {
74              cursor = cursor->right;
75           }
76        }
77        else // Current data matches insert data, replacing anyway
78        {
79           cursor->data = insInt;
80           return;
81        }
82     }
83
84  }
85
86  // write definition for bst_remove here
87  bool bst_remove(btNode*& bst_root, int remInt)
88  {
89     if(bst_root == 0) // If tree is empty, return false
90     {
91        return false;
92     }
93
94     if(bst_root->data > remInt) // Target to remove is less than current
95     {
96        bst_remove(bst_root->left, remInt);
97
98     }
99     else if(bst_root->data < remInt) // Target to remove is greater than current
100    {
101       bst_remove(bst_root->right, remInt);
102    }
103    else                              // Target has been found
104    {
105       if(bst_root->left != 0 && bst_root->right != 0) // Current root has both
106       {                                               // left and right children
107          bst_remove_max(bst_root->left, bst_root->data);
108       }
109       else                           // Current root has only one child or none
110       {
111          btNode* old_bst_root = bst_root;
112          if(bst_root->left == 0 && bst_root->right != 0) // Right child present
113          {
114             bst_root = bst_root->right;
115          }
116          else if (bst_root-> left != 0 && bst_root->right == 0) // Left present
117          {
118             bst_root = bst_root->left;
119          }
120          else                        // No children
121          {
122             bst_root = 0;
123          }
124          delete old_bst_root;       // Free old root
125       }
126       return true;
127    }
128 }
129
130 // write definition for bst_remove_max here
131 void bst_remove_max(btNode*& bst_root, int& data)
132 {
133    if(bst_root == 0) // If current root is empty, return
134    {
135       return;
136    }
137
138    if(bst_root->right == 0) // If no right child, root is largest item
```

```
139      {
140         btNode* removeNode = bst_root;
141         data = bst_root->data;
142         bst_root = bst_root->left;
143         delete removeNode;
144      }
145      else                  // Right child found, root not largest item
146      {
147         bst_remove_max(bst_root->right, data);
148      }
149   }
150
```