

```

1  #include "btNode.h"
2
3  void dumpToArrayInOrder(btNode* bst_root, int* dumpArray)
4  {
5      if (bst_root == 0) return;
6      int dumpIndex = 0;
7      dumpToArrayInOrderAux(bst_root, dumpArray, dumpIndex);
8  }
9
10 void dumpToArrayInOrderAux(btNode* bst_root, int* dumpArray, int& dumpIndex)
11 {
12     if (bst_root == 0) return;
13     dumpToArrayInOrderAux(bst_root->left, dumpArray, dumpIndex);
14     dumpArray[dumpIndex++] = bst_root->data;
15     dumpToArrayInOrderAux(bst_root->right, dumpArray, dumpIndex);
16 }
17
18 void tree_clear(btNode*& root)
19 {
20     if (root == 0) return;
21     tree_clear(root->left);
22     tree_clear(root->right);
23     delete root;
24     root = 0;
25 }
26
27 int bst_size(btNode* bst_root)
28 {
29     if (bst_root == 0) return 0;
30     return 1 + bst_size(bst_root->left) + bst_size(bst_root->right);
31 }
32
33 // write definition for bst_insert here
34 void bst_insert(btNode*& bst_root, int insInt){
35
36     if(bst_root == 0) // If Tree is empty, add insInt to root
37     {
38         btNode* temp_root = new btNode;
39         temp_root->data = insInt;
40         temp_root->left = temp_root->right = 0;
41         bst_root = temp_root;
42         return;
43     }
44
45     btNode* cursor = bst_root; // Creating cursor for traversing tree
46
47     while (cursor != 0) // If tree not empty
48     {
49         if (cursor->data > insInt) // If insInt smaller, insert left
50         {
51             if(cursor->left == 0) // If left empty, insert insInt
52             {
53                 cursor->left = new btNode;
54                 cursor->left->data = insInt;
55                 cursor->left->left = cursor->left->right = 0;
56                 return;
57             }
58             else // Continue to next left node
59             {
60                 cursor = cursor->left;
61             }
62         }
63         else if(cursor->data < insInt) // If insInt smaller, insert left
64         {
65             if(cursor->right == 0) // If left empty, insert insInt
66             {
67                 cursor->right = new btNode;
68                 cursor->right->data = insInt;
69                 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) // Only right child
113            { // present not left
114                bst_root = bst_root->right;
115            }
116            else if (bst_root->left != 0 && bst_root->right == 0) // Only left
117            { // child present
118                bst_root = bst_root->left; // not right
119            }
120            else // No children
121            {
122                bst_root = 0;
123            }
124            delete old_bst_root; // Free up 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
```