Binary Search Tree

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;
template<class DT>
class BNode
{
public:
     BNode();
     void setLeftChild(BNode<DT>* n);
     BNode<DT>* getLeftChild();
     void setRightChild(BNode<DT>* n);
     BNode<DT>* getRightChild();
     void setData(DT pdate);
     DT getData();
private:
     DT data;
     BNode* leftchild;
     BNode* rightchild;
};
template<class DT>
class BinarySearchTree
{
public:
     //part1: constructor
     BinarySearchTree();

     //part2: Create and insert a BNode carrying data
     //in the binary search tree. It return true if
     //insertion takes place successfully and false otherwise
     bool insert(const DT data);

     //part3: Search for data in the binary search tree
     // and return the pointer of the node carrying data
     //return null/0 if data doesn't exist
     BNode<DT>* search(const DT data);

     //part4: prints all the data present in the tree
               //sorted in ascending order
     void printSorted();


     //part5: delete the BNode carrying data from the
     //binary search tree. It return true if
     //deletion takes place successfully and false otherwise
     bool Delete(const DT data);


     //part6: destructor, delete all nodes
     ~BinarySearchTree();


private:
     BNode<DT>* root;

};

template<class DT>
BNode<DT>::BNode()
```

```
60. {
61.      leftchild = NULL;
62.      rightchild = NULL;
63. }
64. template<class DT>
65. void BNode<DT>::setLeftChild(BNode<DT>* n)
66. {
67.      leftchild = n;
68. }
69. template<class DT>
70. BNode<DT>* BNode<DT>::getLeftChild()
71. {
72.      return leftchild;
73. }
74. template<class DT>
75. void BNode<DT>::setRightChild(BNode<DT>* n)
76. {
77.      rightchild = n;
78. }
79. template<class DT>
80. BNode<DT>* BNode<DT>::getRightChild()
81. {
82.      return rightchild;
83. }
84. template<class DT>
85. void BNode<DT>::setData(DT pdate)
86. {
87.      data = pdate;
88. }
89. template<class DT>
90. DT BNode<DT>::getData()
91. {
92.      return data;
93. }
94.
95. template<class DT>
96. BinarySearchTree<DT>::BinarySearchTree()
97. {
98.      root = NULL;
99. }
100. template<class DT>
```

```
101. bool BinarySearchTree<DT>::insert(const DT data)
102. {
103.      BNode<DT>* node = new BNode<DT>();
104.      node->setData(data);
105.      BNode<DT>* p = NULL;
106.      BNode<DT>* current = root;
107.      if (root == NULL)
108.      {
109.                root = node;
110.                root->setLeftChild(NULL);
111.                root->setRightChild(NULL);
112.                return true;
113.      }
114.      else
115.      {
116.                current = root;
117.                while (current != NULL)
118.                {
119.                          p = current;
120.                          if (data > current->getData())
121.                          {
122.                                    current = current->getRightChild();
123.                          }
124.                          else if (data < current->getData())
125.                          {
126.                                    current = current->getLeftChild();
127.                          }
128.                          else if (data == current->getData())
129.                          {
130.                                    cout << "Already exists" << endl;
131.                                    return false;
132.                          }
133.                }
134.                if (data < p->getData())
135.                {
136.                          p->setLeftChild(node);
137.                          return true;
138.                }
139.                else if (data > p->getData())
140.                {
141.                          p->setRightChild(node);
142.                          return true;
143.                }
144.      }
145.      return false;
146. }
```

```
147. template<class DT>
148. void BinarySearchTree<DT>::printSorted()
149. {
150.      BNode <DT>* temp = root;
151.      stack<BNode<DT>*>* s = new stack<BNode<DT>*>();
152.      if (temp)
153.      {
154.                while (true)
155.                {
156.                          if (temp != NULL)
157.                          {
158.                                    s->push(temp);
159.                                    temp = temp->getLeftChild();
160.                          }
161.                          else
162.                          {
163.                                    if (!s->empty())
164.                                    {
165.                                              temp = s->top();
166.                                              s->pop();
167.                                              cout << temp->getData() << " ";
168.                                              temp = temp->getRightChild();
169.                                    }
170.                                    else
171.                                    {
172.                                              break;
173.                                    }
174.                          }
175.                }
176.      }
177. }
178. template<class DT>
179. BNode<DT>* BinarySearchTree<DT>::search(const DT data)
180. {
181.      BNode<DT>* node = new BNode<DT>();
182.      node->setData(data);
183.      BNode<DT>* temp = root;
184.
185.      if (data == root->getData())
186.      {
187.                return root;
188.      }
189.      else
190.      {
191.                while (temp != NULL)
192.                {
193.                          if (data > temp->getData())
194.                          {
195.                                    temp = temp->getRightChild();
196.                          }
197.                          else if (data < temp->getData())
198.                          {
199.                                    temp = temp->getLeftChild();
200.                          }
201.                          else if (data == temp->getData())
202.                          {
203.                                    return temp;
204.                          }
205.                          else
206.                          {
207.                                    return 0;
208.                          }
209.                }
210.      }
211. }
```

```
212.
213. template<class DT>
214. bool BinarySearchTree<DT>::Delete(const DT data)
215. {
216.     BNode<DT>* current = root;
217.     BNode<DT>* parent = nullptr;
218.
219.     // Find the node to delete and its parent
220.     while (current != nullptr && current->getData() != data) {
221.             parent = current;
222.             if (data < current->getData()) {
223.                     current = current->getLeftChild();
224.             }
225.             else {
226.                     current = current->getRightChild();
227.             }
228.     }
229.
230.     if (current == nullptr) {
231.             // Node not found
232.             return false;
233.     }
234.
235.     // Case 1: Node to delete is a leaf node
236.     if (current->getLeftChild() == nullptr && current->getRightChild() == nullptr) {
237.             if (current == root) {
238.                     root = nullptr;
239.             }
240.             else if (parent->getLeftChild() == current) {
241.                     parent->setLeftChild(nullptr);
242.             }
243.             else {
244.                     parent->setRightChild(nullptr);
245.             }
246.             delete current;
247.     }
248.     // Case 2: Node to delete has one child
249.     else if (current->getLeftChild() == nullptr) { // Only right child
250.             if (current == root) {
251.                     root = current->getRightChild();
252.             }
253.             else if (parent->getLeftChild() == current) {
254.                     parent->setLeftChild(current->getRightChild());
255.             }
256.             else {
257.                     parent->setRightChild(current->getRightChild());
258.             }
259.             delete current;
260.     }
261.     else if (current->getRightChild() == nullptr) { // Only left child
262.             if (current == root) {
263.                     root = current->getLeftChild();
264.             }
265.             else if (parent->getLeftChild() == current) {
266.                     parent->setLeftChild(current->getLeftChild());
267.             }
268.             else {
269.                     parent->setRightChild(current->getLeftChild());
270.             }
271.             delete current;
272.     }
```

```
273.      // Case 3: Node to delete has two children
274.      else {
275.                BNode<DT>* successor = current->getRightChild();
276.                BNode<DT>* successorParent = current;
277.
278.                // Find the inorder successor (smallest in the right subtree)
279.                while (successor->getLeftChild() != nullptr) {
280.                          successorParent = successor;
281.                          successor = successor->getLeftChild();
282.                }
283.
284.                // Copy the successor's data to the current node
285.                current->setData(successor->getData());
286.
287.                // Delete the successor (which is now a leaf or has only a right child)
288.                if (successorParent->getLeftChild() == successor) {
289.                          successorParent->setLeftChild(successor->getRightChild());
290.                }
291.                else {
292.                          successorParent->setRightChild(successor->getRightChild());
293.                }
294.                delete successor;
295.      }
296.
297.      return true;
298. }
299.
300. int main()
301. {
302.      //creating an object of binary search tree
303.      BinarySearchTree<int>* BST = new BinarySearchTree<int>();
304.
305.      //following insertions should happen successfully as we are inserting unique values
306.      BST->insert(12);
307.      BST->insert(4);
308.      BST->insert(9);
309.      BST->insert(2);
310.      BST->insert(14);
311.      BST->insert(16);
312.      BST->insert(13);
313.      //this insertion should fail as 12 already exists in the Binary Search tree
314.      BST->insert(12);
315.
316.      //prints data carried by the BST in sorted manner
317.      BST->printSorted();
318.
319.      //the first search would be successful and second would fail
320.      BNode<int>* n = BST->search(12);
321.      if (n)
322.      {
323.                cout << "Value exists" << endl;
324.      }
325.      else
326.      {
327.                cout << "Value does not exist" << endl;
328.      }
329.
330.      BNode<int>* w = BST->search(23);
331.      if (w)
332.      {
333.                cout << "Value exists" << endl;
334.      }
335.      else
336.      {
337.                cout << "Value does not exist" << endl;
```

```
338.      }
339.
340.
341.      system("pause");
342.      return 0;
343. }
344.
```

Output