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()
```

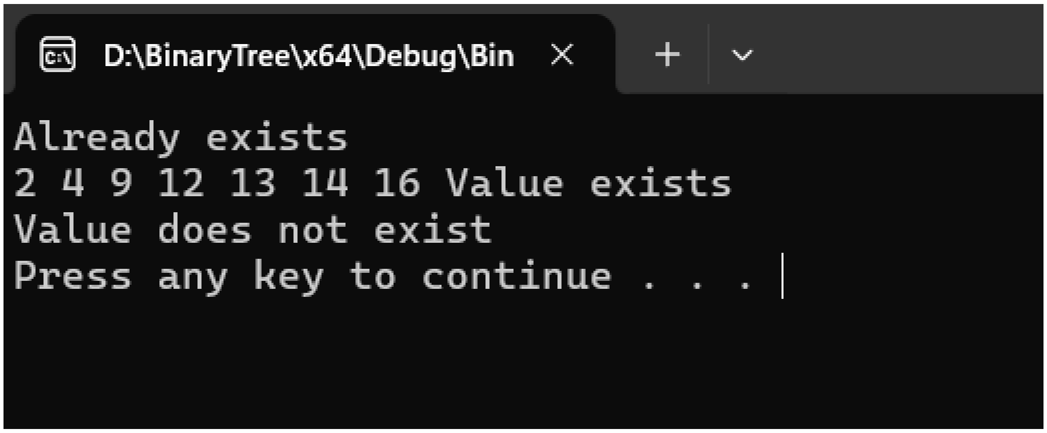
```cpp
{
    leftchild = NULL;
    rightchild = NULL;
}
template<class DT>
void BNode<DT>::setLeftChild(BNode<DT>* n)
{
    leftchild = n;
}
template<class DT>
BNode<DT>* BNode<DT>::getLeftChild()
{
    return leftchild;
}
template<class DT>
void BNode<DT>::setRightChild(BNode<DT>* n)
{
    rightchild = n;
}
template<class DT>
BNode<DT>* BNode<DT>::getRightChild()
{
    return rightchild;
}
template<class DT>
void BNode<DT>::setData(DT pdate)
{
    data = pdate;
}
template<class DT>
DT BNode<DT>::getData()
{
    return data;
}

template<class DT>
BinarySearchTree<DT>::BinarySearchTree()
{
    root = NULL;
}
template<class DT>
```

```cpp
bool BinarySearchTree<DT>::insert(const DT data)
{
    BNode<DT>* node = new BNode<DT>();
    node->setData(data);
    BNode<DT>* p = NULL;
    BNode<DT>* current = root;
    if (root == NULL)
    {
            root = node;
            root->setLeftChild(NULL);
            root->setRightChild(NULL);
            return true;
    }
    else
    {
            current = root;
            while (current != NULL)
            {
                    p = current;
                    if (data > current->getData())
                    {
                            current = current->getRightChild();
                    }
                    else if (data < current->getData())
                    {
                            current = current->getLeftChild();
                    }
                    else if (data == current->getData())
                    {
                            cout << "Already exists" << endl;
                            return false;
                    }
            }
            if (data < p->getData())
            {
                    p->setLeftChild(node);
                    return true;
            }
            else if (data > p->getData())
            {
                    p->setRightChild(node);
                    return true;
            }
    }
    return false;
}
```

```
template<class DT>
void BinarySearchTree<DT>::printSorted()
{
    BNode <DT>* temp = root;
    stack<BNode<DT>*>* s = new stack<BNode<DT>*>();
    if (temp)
    {
        while (true)
        {
            if (temp != NULL)
            {
                s->push(temp);
                temp = temp->getLeftChild();
            }
            else
            {
                if (!s->empty())
                {
                    temp = s->top();
                    s->pop();
                    cout << temp->getData() << " ";
                    temp = temp->getRightChild();
                }
                else
                {
                    break;
                }
            }
        }
    }
}
template<class DT>
BNode<DT>* BinarySearchTree<DT>::search(const DT data)
{
    BNode<DT>* node = new BNode<DT>();
    node->setData(data);
    BNode<DT>* temp = root;

    if (data == root->getData())
    {
        return root;
    }
    else
    {
        while (temp != NULL)
        {
            if (data > temp->getData())
            {
                temp = temp->getRightChild();
            }
            else if (data < temp->getData())
            {
                temp = temp->getLeftChild();
            }
            else if (data == temp->getData())
            {
                return temp;
            }
            else
            {
                return 0;
            }
        }
    }
}
```

```cpp

template<class DT>
bool BinarySearchTree<DT>::Delete(const DT data)
{
    BNode<DT>* current = root;
    BNode<DT>* parent = nullptr;

    // Find the node to delete and its parent
    while (current != nullptr && current->getData() != data) {
        parent = current;
        if (data < current->getData()) {
            current = current->getLeftChild();
        }
        else {
            current = current->getRightChild();
        }
    }

    if (current == nullptr) {
        // Node not found
        return false;
    }

    // Case 1: Node to delete is a leaf node
    if (current->getLeftChild() == nullptr && current->getRightChild() == nullptr) {
        if (current == root) {
            root = nullptr;
        }
        else if (parent->getLeftChild() == current) {
            parent->setLeftChild(nullptr);
        }
        else {
            parent->setRightChild(nullptr);
        }
        delete current;
    }

    // Case 2: Node to delete has one child
    else if (current->getLeftChild() == nullptr) { // Only right child
        if (current == root) {
            root = current->getRightChild();
        }
        else if (parent->getLeftChild() == current) {
            parent->setLeftChild(current->getRightChild());
        }
        else {
            parent->setRightChild(current->getRightChild());
        }
        delete current;
    }
    else if (current->getRightChild() == nullptr) { // Only left child
        if (current == root) {
            root = current->getLeftChild();
        }
        else if (parent->getLeftChild() == current) {
            parent->setLeftChild(current->getLeftChild());
        }
        else {
            parent->setRightChild(current->getLeftChild());
        }
        delete current;
    }
    // Case 3: Node to delete has two children
    else {
            BNode<DT>* successor = current->getRightChild();
            BNode<DT>* successorParent = current;

            // Find the inorder successor (smallest in the right subtree)
            while (successor->getLeftChild() != nullptr) {
                    successorParent = successor;
                    successor = successor->getLeftChild();
            }

            // Copy the successor's data to the current node
            current->setData(successor->getData());

            // Delete the successor (which is now a leaf or has only a right child)
            if (successorParent->getLeftChild() == successor) {
                    successorParent->setLeftChild(successor->getRightChild());
            }
            else {
                    successorParent->setRightChild(successor->getRightChild());
            }
            delete successor;
    }

    return true;
}

int main()
{
    //creating an object of binary search tree
    BinarySearchTree<int>* BST = new BinarySearchTree<int>();

    //following insertions should happen successfully as we are inserting unique values
    BST->insert(12);
    BST->insert(4);
    BST->insert(9);
    BST->insert(2);
    BST->insert(14);
    BST->insert(16);
    BST->insert(13);
    //this insertion should fail as 12 already exists in the Binary Search tree
    BST->insert(12);

    //prints data carried by the BST in sorted manner
    BST->printSorted();

    //the first search would be successful and second would fail
    BNode<int>* n = BST->search(12);
    if (n)
    {
            cout << "Value exists" << endl;
    }
    else
    {
            cout << "Value does not exist" << endl;
    }

    BNode<int>* w = BST->search(23);
    if (w)
    {
            cout << "Value exists" << endl;
    }
    else
    {
            cout << "Value does not exist" << endl;
    }


    system("pause");
    return 0;
}

```

Output



```
Already exists
2 4 9 12 13 14 16 Value exists
Value does not exist
Press any key to continue . . .
```