```cpp
template <typename T> void BinaryTree<T>::inorder(shared_ptr<Node<T>> &node) {
    if (node) {
        if (node->left)
            inorder(node->left);
        std::cout << node->item << " ";
        if (node->right)
            return inorder(node->right);
    }
}

template <typename T> void BinaryTree<T>::preorder(shared_ptr<Node<T>> &node) {
    if (node) {
        std::cout << node->item << " ";
        if (node->left)
            preorder(node->left);
        if (node->right)
            preorder(node->right);
    }
}

template <typename T> void BinaryTree<T>::postorder(shared_ptr<Node<T>> &node) {
    if (node) {
        if (node->left)
            postorder(node->left);
        if (node->right)
            postorder(node->right);
        std::cout << node->item << " ";
    }
}
```

```
template <typename T> void BinaryTree<T>::printTree(shared_ptr<Node<T>> &node, int level) {
    if (node) {
        printTree(node->right, level + 1);
        std::string spaces(level * 2, ' ');
        std::cout << spaces << node->item << "\n";
        printTree(node->left, level + 1);
    }
}
```

```
template <typename T> int BinaryTree<T>::height(shared_ptr<Node<T>> &node) {
    if (node) {
        int heightLeft = height(node->left);
        int heightRight = height(node->right);
        return (1 + std::max(heightLeft, heightRight));
    }
    return 0;
}
```

```
template <typename T> int BinaryTree<T>::sum(shared_ptr<Node<T>> &node) {
    if (node) {
        int sumLeft = sum(node->left);
        int sumRight = sum(node->right);
        return (node->item + sumLeft + sumRight);
    }
    return 0;
}
```

```
template <typename T> int BinaryTree<T>::product(shared_ptr<Node<T>> &node) {
    if (node) {
        int productLeft = product(node->left);
        int productRight = product(node->right);
        return (node->item * productLeft * productRight);
    }
    return 1;
}
```

```
template <typename T> int BinaryTree<T>::nodes(shared_ptr<Node> &root) {
    if (root == nullptr)
        return 0;

    return nodes(root->left) + 1 + nodes(root->right);
}
```

```cpp
template <typename T> bool BinaryTree<T>::isBalanced(shared_ptr<Node<T>> &node) {
    if (node) {
        int heightLeft = height(node->left);
        int heightRight = height(node->right);
        return (std::abs(heightLeft - heightRight) <= 1 &&
                isBalanced(node->left) && isBalanced(node->right));
    }
    return true;
}
```

```
template <typename T> bool BinaryTree<T>::compare(shared_ptr<Node> &a, shared_ptr<Node> &b) {
    if (a == nullptr && b == nullptr)
        return true;
    if (a == nullptr && b != nullptr)
        return false;
    if (a != nullptr && b == nullptr)
        return false;
    if (a->item != b->item)
        return false;

    return compare(a->left, b->left) && compare(a->right, b->right);
}
```

```cpp
template <typename T> void BinaryTree<T>::mirror(shared_ptr<Node> &root) {
    if (root == nullptr)
        return;

    shared_ptr<Node> tmp = root->left;
    root->left = root->right;
    root->right = tmp;

    mirror(root->left);
    mirror(root->right);
}
```

```
template <typename T> bool BSTree<T>::search(shared_ptr<Node<T>> &node, const T &item) {
    if (node) {
        if (node->item < item)
            return search(node->right, item);
        else if (node->item > item)
            return search(node->left, item);
        else // found item
            return true;
    }
    return false;
}
```

```cpp
template <typename T> void BSTree<T>::insert(shared_ptr<Node<T>> &node, const T &newItem) {
    if (node) {
        if (node->item < newItem)
            insert(node->right, newItem);
        else if (node->item > newItem)
            insert(node->left, newItem);
        else
            throw std::invalid_argument("Error: Item " + std::to_string(newItem) +
                    " already exists in the tree.");
    } else {
        node = std::make_shared<Node<T>>(newItem);
    }
}
```

```
template <typename T> bool BSTree<T>::remove(shared_ptr<Node<T>> &node, const T &item) {
   if (node) {
      if (node->item < item) {
         return remove(node->right, item);
      } else if (node->item > item) {
         return remove(node->left, item);
      } else { // found item
         if (!node->left) { // no left subtree
            node = node->right;
         } else if (!node->right) { // no right subtree
            node = node->left;
         } else { // find the rightmost child in the left subtree
            shared_ptr<Node<T>> tmp = node->left, prev = nullptr;
            while (tmp->right) {
               prev = tmp;
               tmp = tmp->right;
            }

            node->item = tmp->item;
            if (prev)
               prev->right = tmp->left;
            else
               node->left = tmp->left;
         }
         return true;
      }
   }
   return false;
}
```