Tree Implementation

Tiziana Ligorio

Today's Plan

Recap

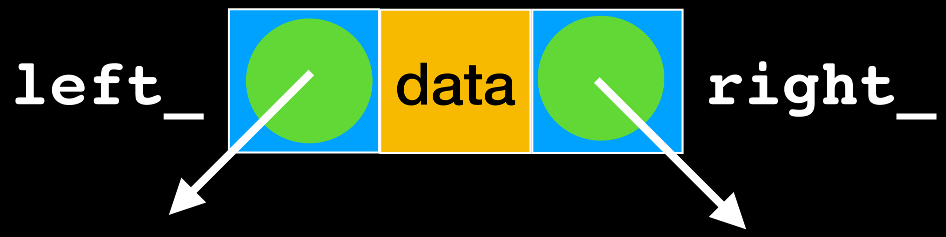
BST Implementation

```cpp
#ifndef BST_H_
#define BST_H_
#include <memory>
using namespace std;
template<typename ItemType>
class BST
{
public:
    BST(); // constructor
    BST(const BST<ItemType>& tree); // copy constructor
    ~ BST(); // destructor
    bool isEmpty() const;
    size_t getHeight() const;
    size_t getNumberOfNodes() const;
    void add(const ItemType& new_item);
    void remove(const ItemType& new_item);
    ItemType find(const ItemType& item) const;
    void clear();
    void preorderTraverse(Visitor<ItemType>& visit) const;
    void inorderTraverse(Visitor<ItemType>& visit) const;
    void postorderTraverse(Visitor<ItemType>& visit) const;
    BST& operator= (const BST<ItemType>& rhs);
private:
    shared_ptr<BinaryNode<ItemType>> root_ptr_;
}; // end BST
#include "BST.cpp"
#endif // BST_H_
```

Let's try something new and use `shared_ptr`: A bit of extra syntax at declaration but then you use them as regular

To implement this as a linked structure what do we need to change in our previous implementation ???

BinaryNode



```cpp
#ifndef BinaryNode_H_
#define BinaryNode_H_
#include <memory>
using namespace std;
template<typename ItemType>
class BinaryNode
{

public:
   BinaryNode();
   BinaryNode(const ItemType& an_item);
   void setItem(const ItemType& an_item);
   ItemType getItem() const;

   bool isLeaf() const;

   auto getLeftChildPtr() const;
   auto getRightChildPtr() const;

   void setLeftChildPtr(std::shared_ptr<BinaryNode<ItemType>> left_ptr);
   void setRightChildPtr(std::shared_ptr<BinaryNode<ItemType>> right_ptr);

private:
   ItemType item_;   // Data portion
   shared_ptr<BinaryNode<ItemType>> left_;    // Pointer to left child
   shared_ptr<BinaryNode<ItemType>> right_;   // Pointer to right child
}; // end BST
#include "BinaryNode.cpp"
#endif // BinaryNode_H_
```

For shared_ptr

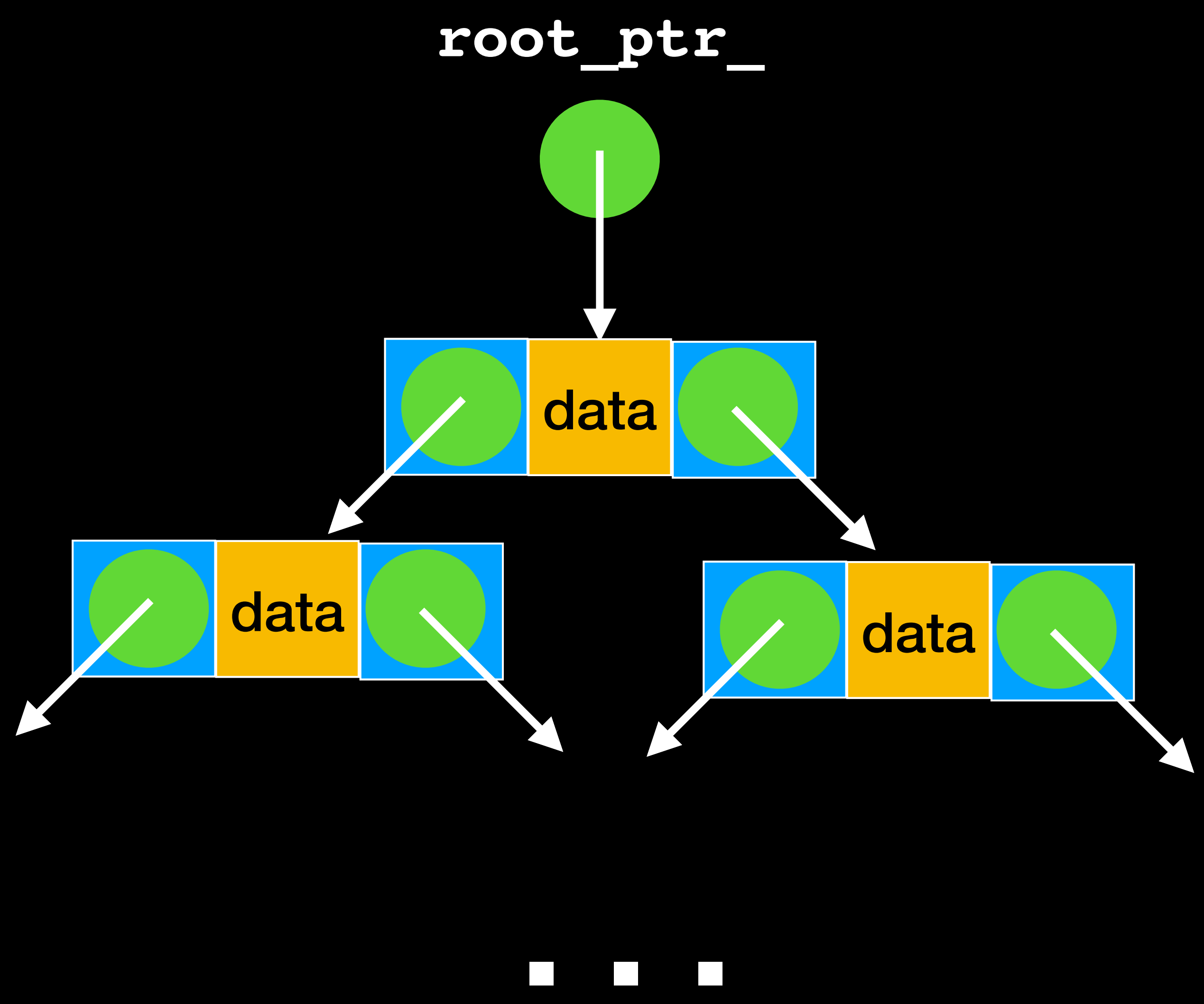
Lecture Activity

Implement:

```
BinaryNode(const ItemType& an_item);

bool isLeaf() const;

void setLeftChildPtr(shared_ptr<BinaryNode<ItemType>> left_ptr);
```

```
template<typename ItemType>
BinaryNode<ItemType>::BinaryNode(const ItemType& an_item)
      : item_(an_item){ }  // end constructor


template<typename ItemType>
bool BinaryNode<ItemType>::isLeaf() const
{
   return ((left_ == nullptr) && (right_ == nullptr));
} // end isLeaf


template<typename ItemType>
void
BinaryNode<ItemType>::setLeftChildPtr(std::shared_ptr<BinaryNode<Ite
mType>> left_ptr)
{
   left_ = left_ptr;
}  // end setLeftChildPtr
```

BST



```
#ifndef BST_H_
#define BST_H_
#include <memory>

template<typename ItemType>
class BST
{

public:
     BST(); // constructor
     BST(const BST<ItemType>& tree); // copy constructor
     ~ BST(); // destructor
     bool isEmpty() const;
     size_t getHeight() const;
     size_t getNumberOfNodes() const;
     void add(const ItemType& new_item);
     void remove(const ItemType& new_item);
     ItemType find(const ItemType& item) const;
     void clear();

     void preorderTraverse(Visitor<ItemType>& visit) const;
     void inorderTraverse(Visitor<ItemType>& visit) const;
     void postorderTraverse(Visitor<ItemType& visit) const;

     BST& operator= (const BST<ItemType>& rhs);

private:
     std::shared_ptr<BinaryNode<ItemType>> root_ptr_;
}; // end BST

#include "BST.cpp"
#endif // BST_H_
```

We want our interface to be generic and not tied to implementation. Many of these will therefore use helper functions, which should be private (or protected if you envision inheritance). I do not include them here in the interface for lack of space.

Copy Constructor

root_ptr of this object

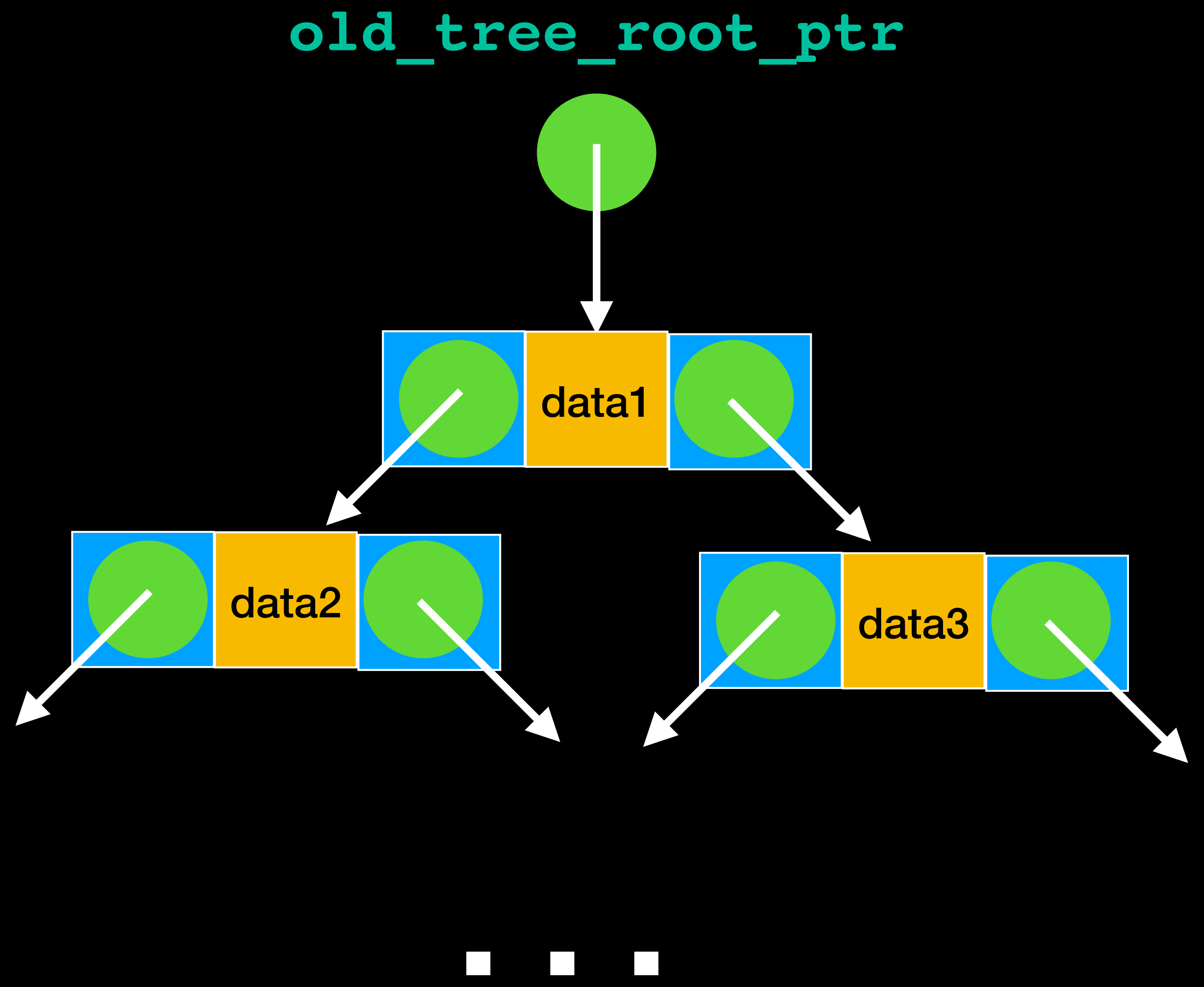
root_ptr of tree: the object I'm going to copy

```
template<class T>
BST<T>::BST(const BST<T>& tree)
{
   root_ptr_ = copyTree(tree.root_ptr_); // Call helper function
}  // end copy constructor
```

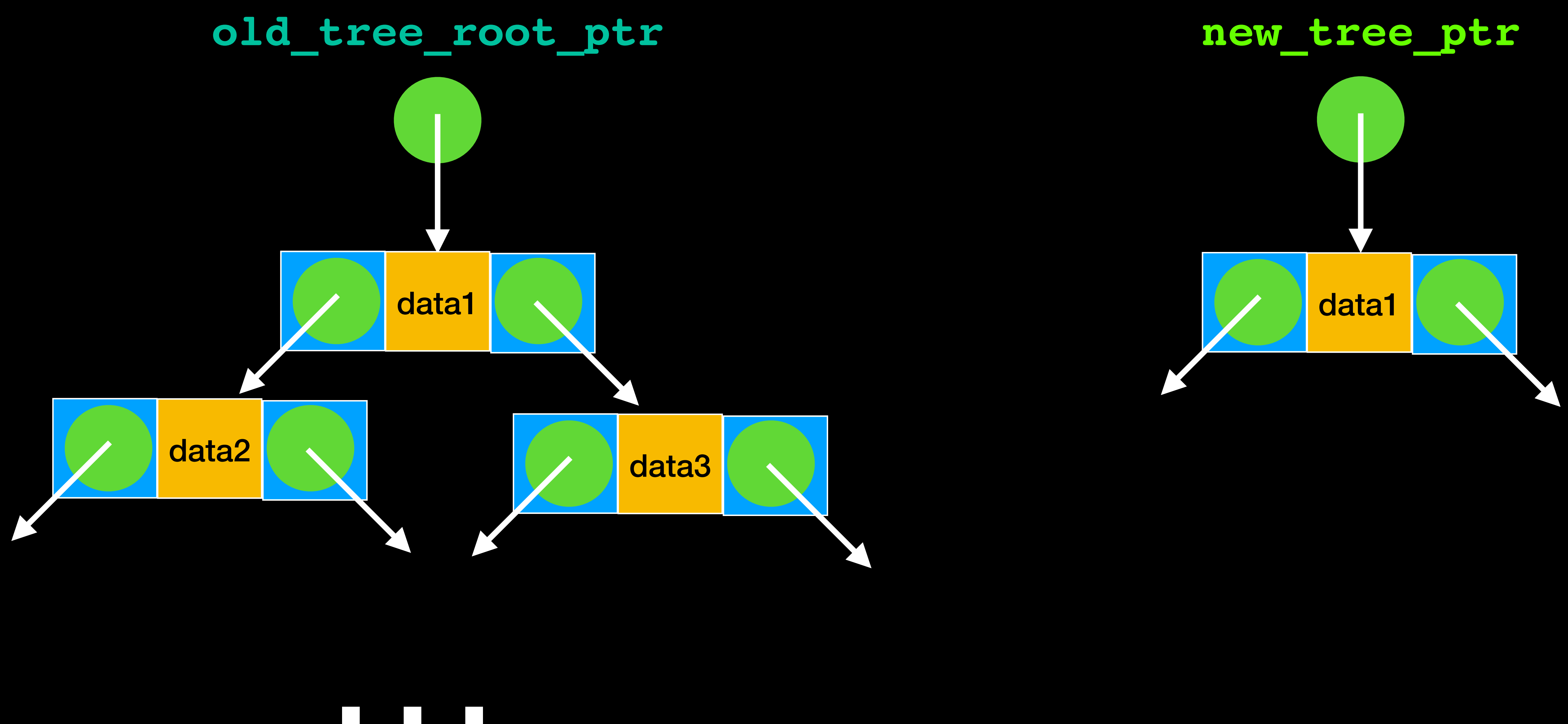
Safe programming: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

I can use the . operator to access a private member variable because it is s within the class definition.

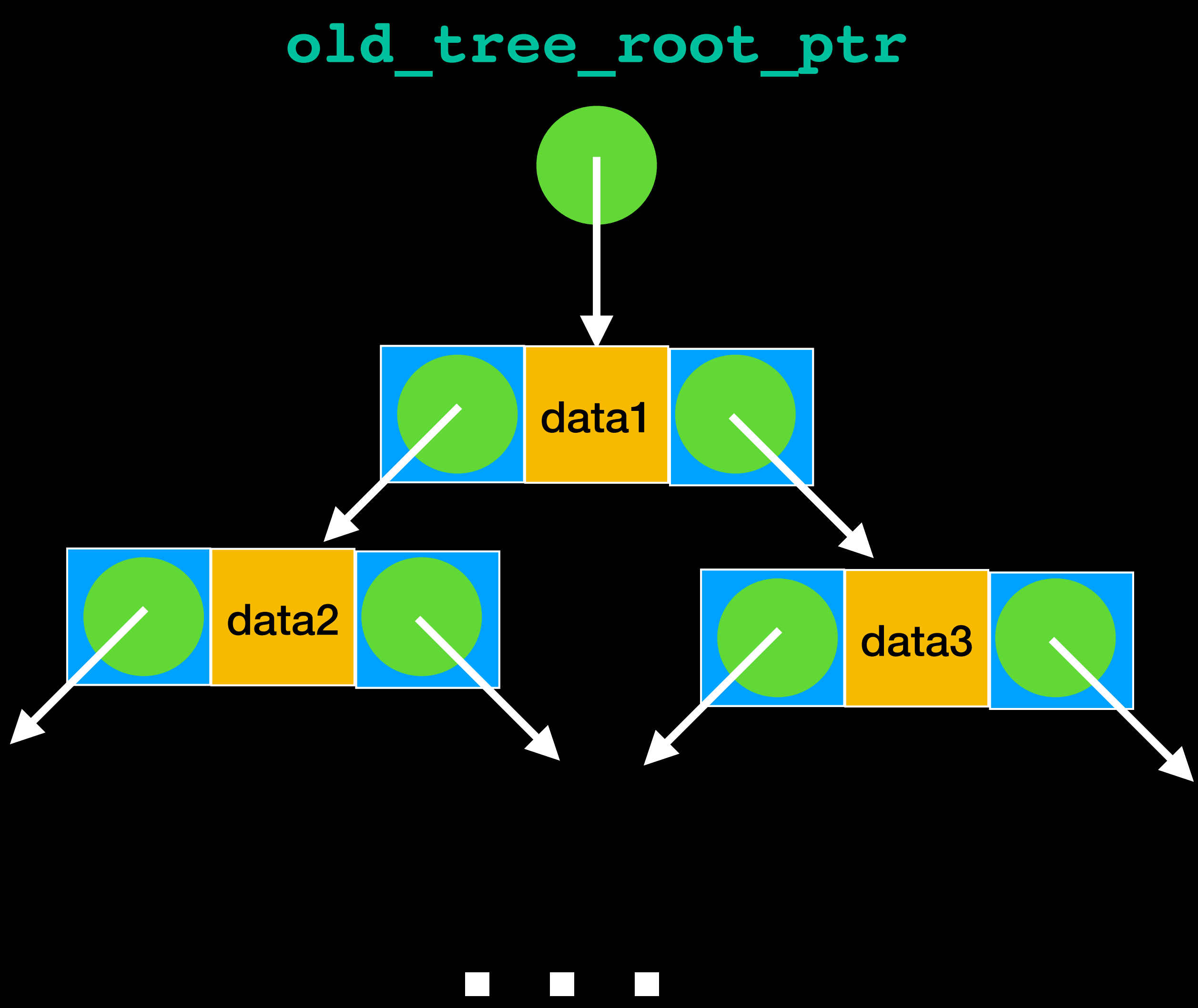
copyTree(old_tree_root_ptr)

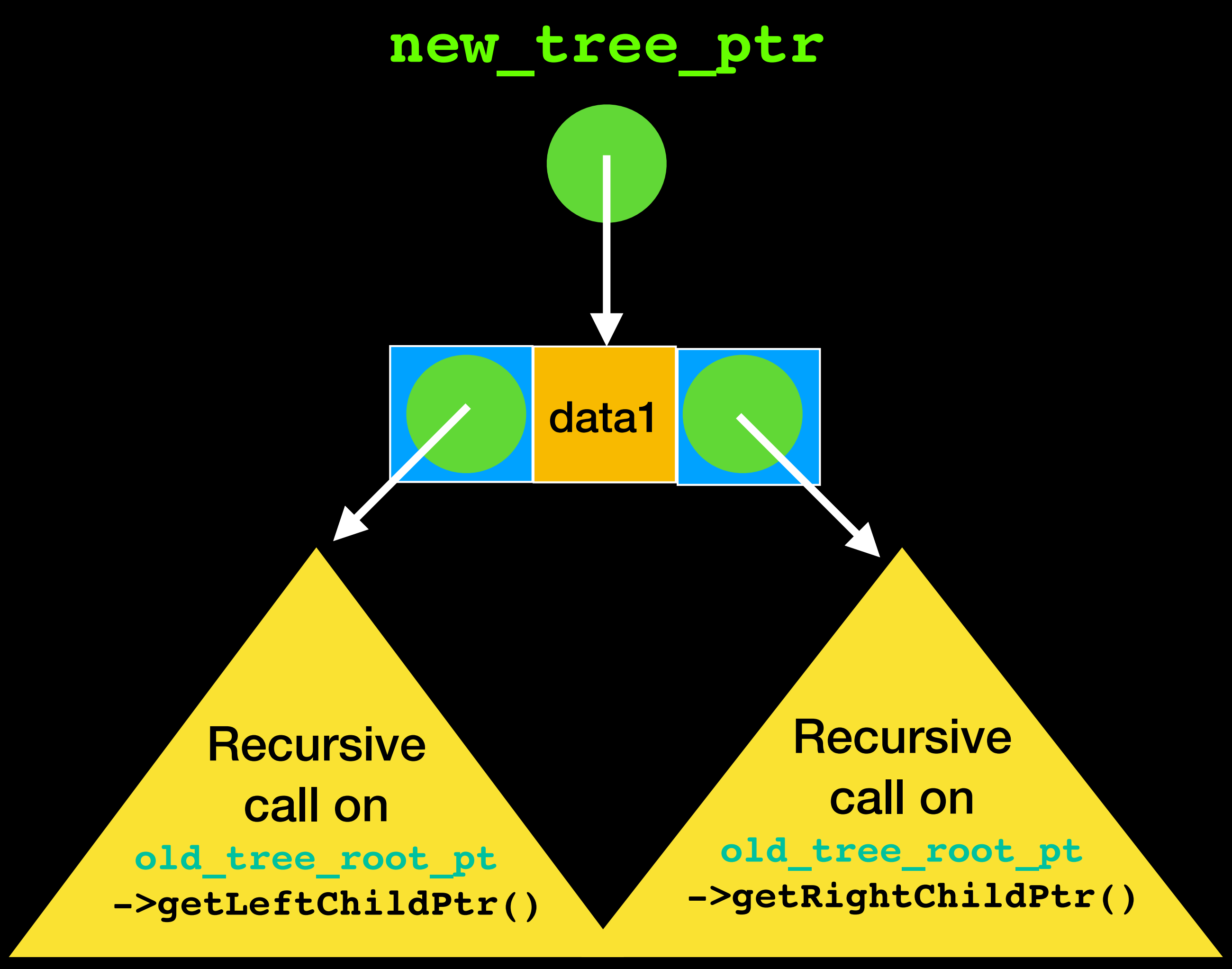


copyTree(old_tree_root_ptr)



copyTree(old_tree_root_ptr)





Copy Constructor Helper Function

Returning `shared_ptr`, cleaner to use auto return type: -std=c++14

```
template<class T>
auto BST<T>::copyTree(const std::shared_ptr<BinaryNode<T>> old_tree_root_ptr) const
{
   std::shared_ptr<BinaryNode<T>> new_tree_ptr;

   // Copy tree nodes during a preorder traversal
   if (old_tree_root_ptr != nullptr)
   {
      // Copy node
      new_tree_ptr = std::make_shared<BinaryNode<T>>(old_tree_root_ptr
                                                  ->getItem(), nullptr, nullptr);
     new_tree_ptr->setLeftChildPtr(copyTree(old_tree_root_ptr->getLeftChildPtr()));
      new_tree_ptr->setRightChildPtr(copyTree(old_tree_root_ptr
                                                           ->getRightChildPtr()));
   }  // end if

   return new_tree_ptr;
}  // end copyTree
```

Recall: this is the syntax for allocating a "**new**" object with `shared_ptr` pointing to it

<u>Recursive Calls:</u>
Don't want to tie interface to recursive implementation: Use helper function

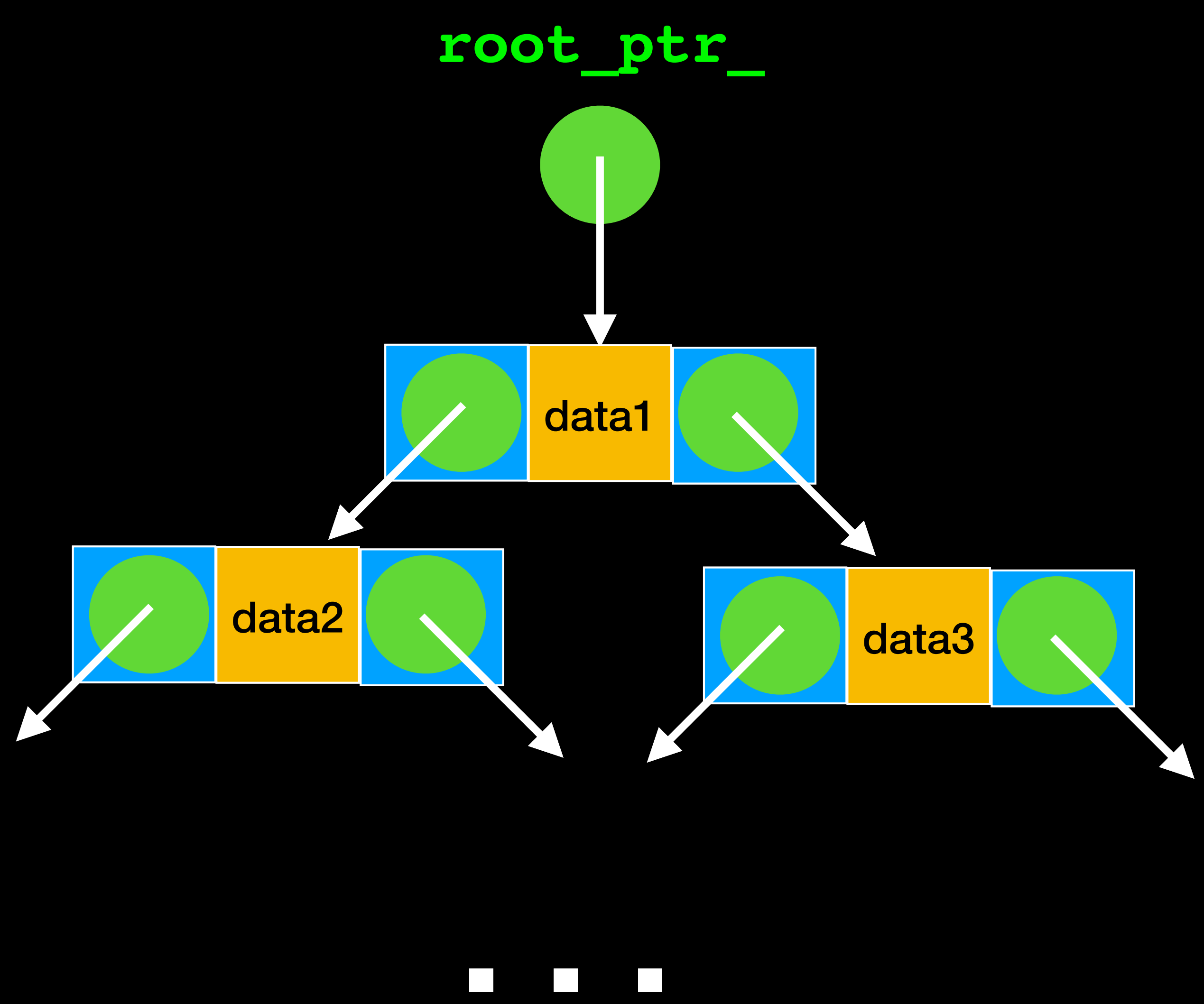
Preorder Traversal Scheme: copy each node as soon as it is visited to make exact copy

Destructor

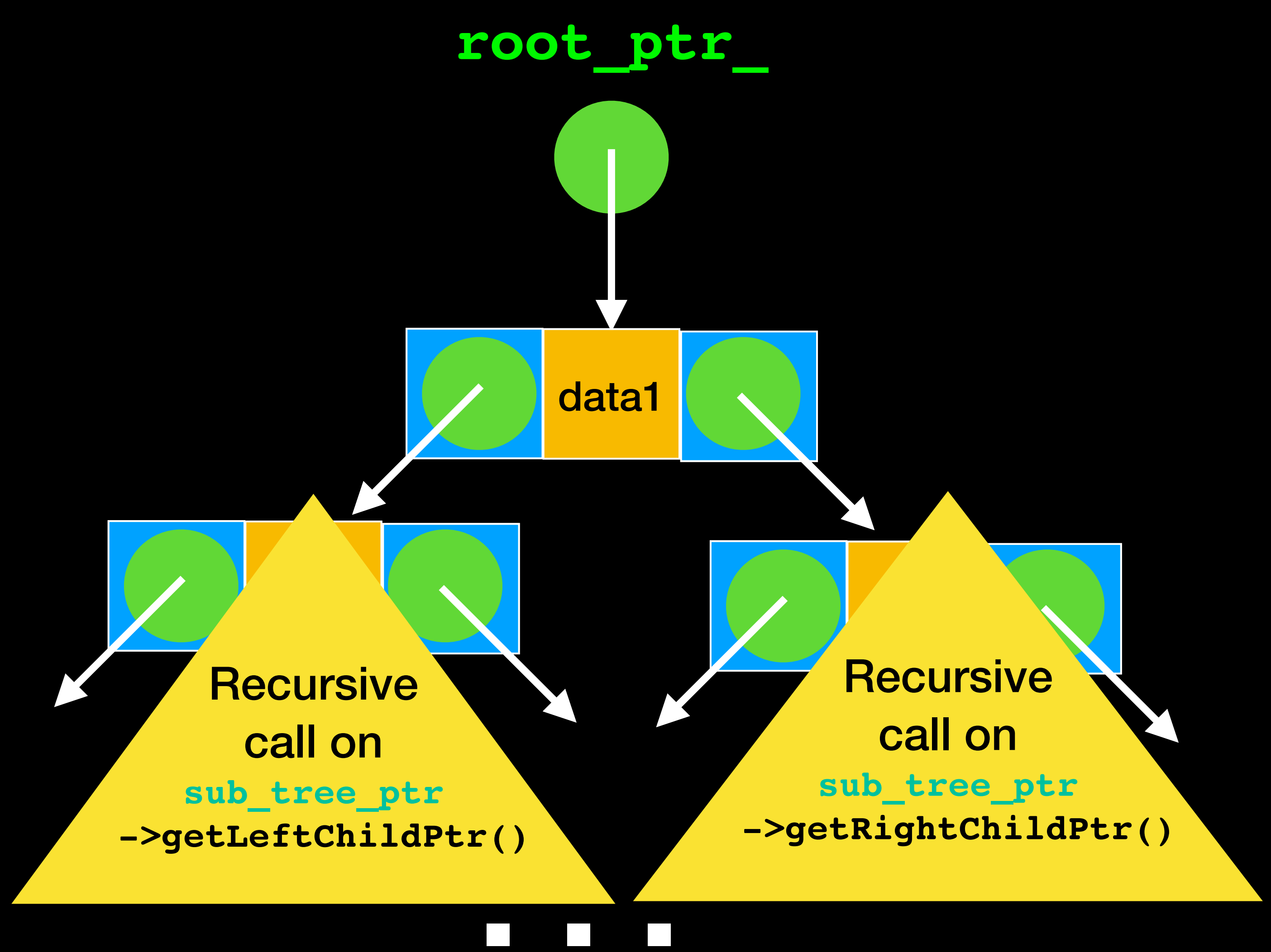
```
template<class T>
BST<T>::~BST()
{
   destroyTree(root_ptr_); // Call helper function
}  // end destructor
```

Safe programming: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

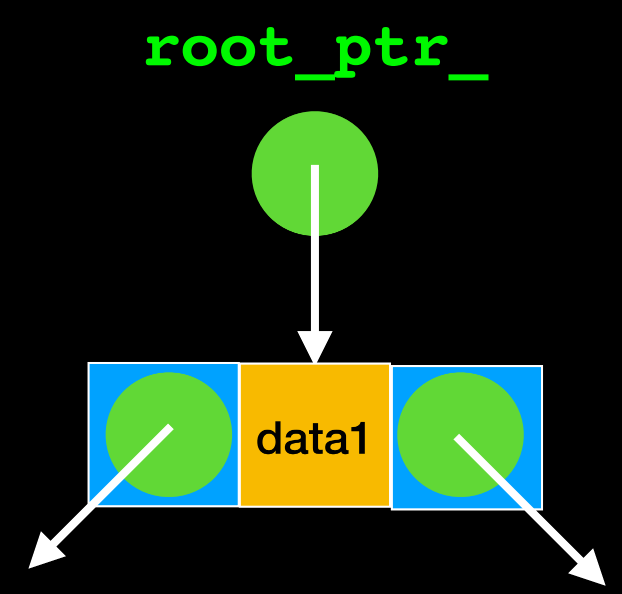
destroyTree(sub_tree_ptr)



destroyTree(sub_tree_ptr)



destroyTree(sub_tree_ptr)



`root_ptr_.reset()`

`destroyTree(sub_tree_ptr)`

`root_ptr_`

Destructor Helper Function

```
template<class T>
void BST<T>::destroyTree(std::shared_ptr<BinaryNode<T>> sub_tree_ptr)
{
   if (sub_tree_ptr != nullptr)
   {
      destroyTree(sub_tree_ptr->getLeftChildPtr());
      destroyTree(sub_tree_ptr->getRightChildPtr());
      sub_tree_ptr.reset(); // same as sub_tree_ptr = nullptr for smart pointers
   }  // end if
}  // end destroyTree
```

Notice: all we have to do is set the `shared_ptr` to `nullptr` with `reset()` and it will take care of deleting the node.

PostOrder Traversal Scheme: Delete node only after deleting both of its subtrees

clear

```
template<class T>
void BST<T>::clear()
{
   destroyTree(root_ptr_); // Call helper method
}  // end clear
```

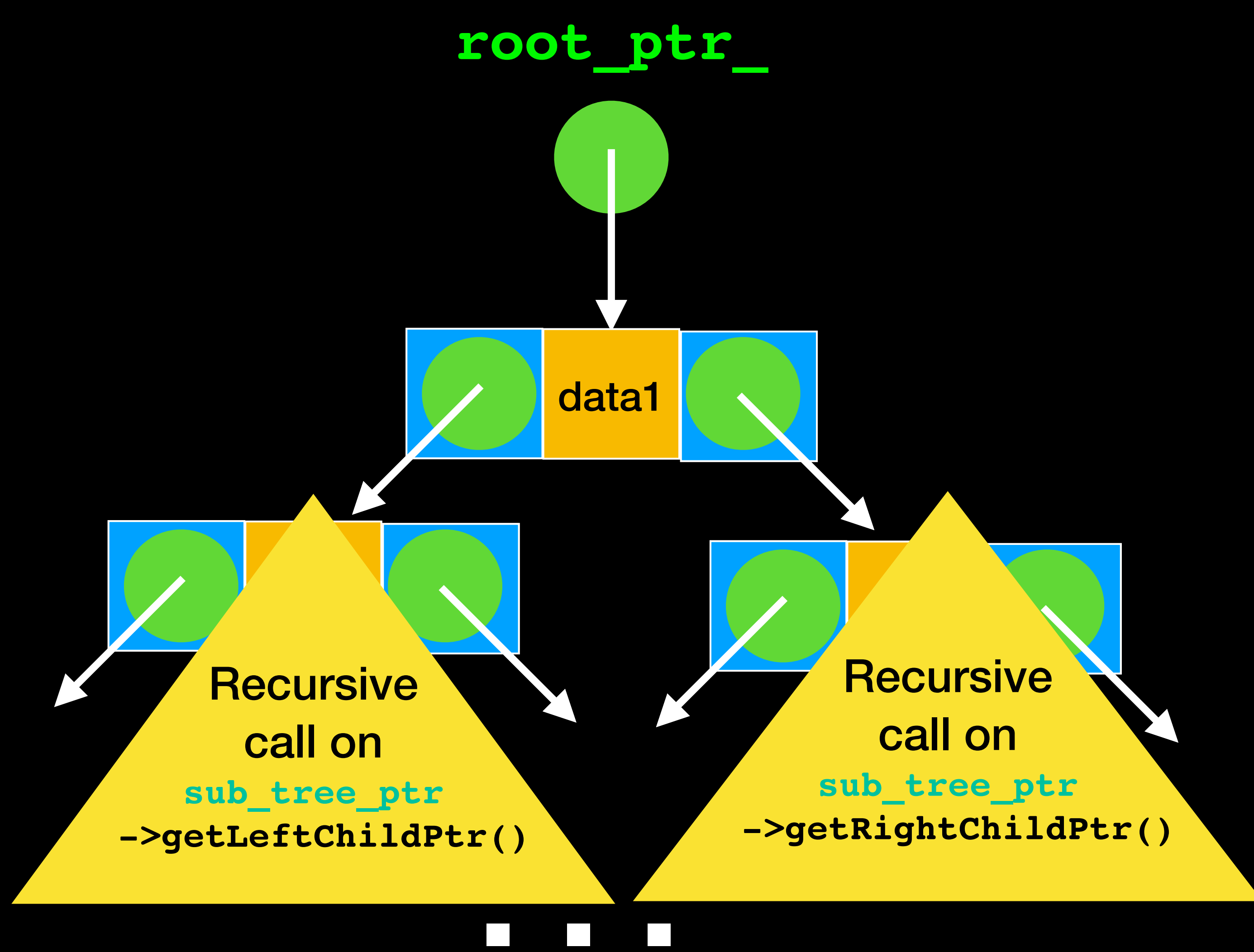
<u>Safe programming</u>: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

getHeight

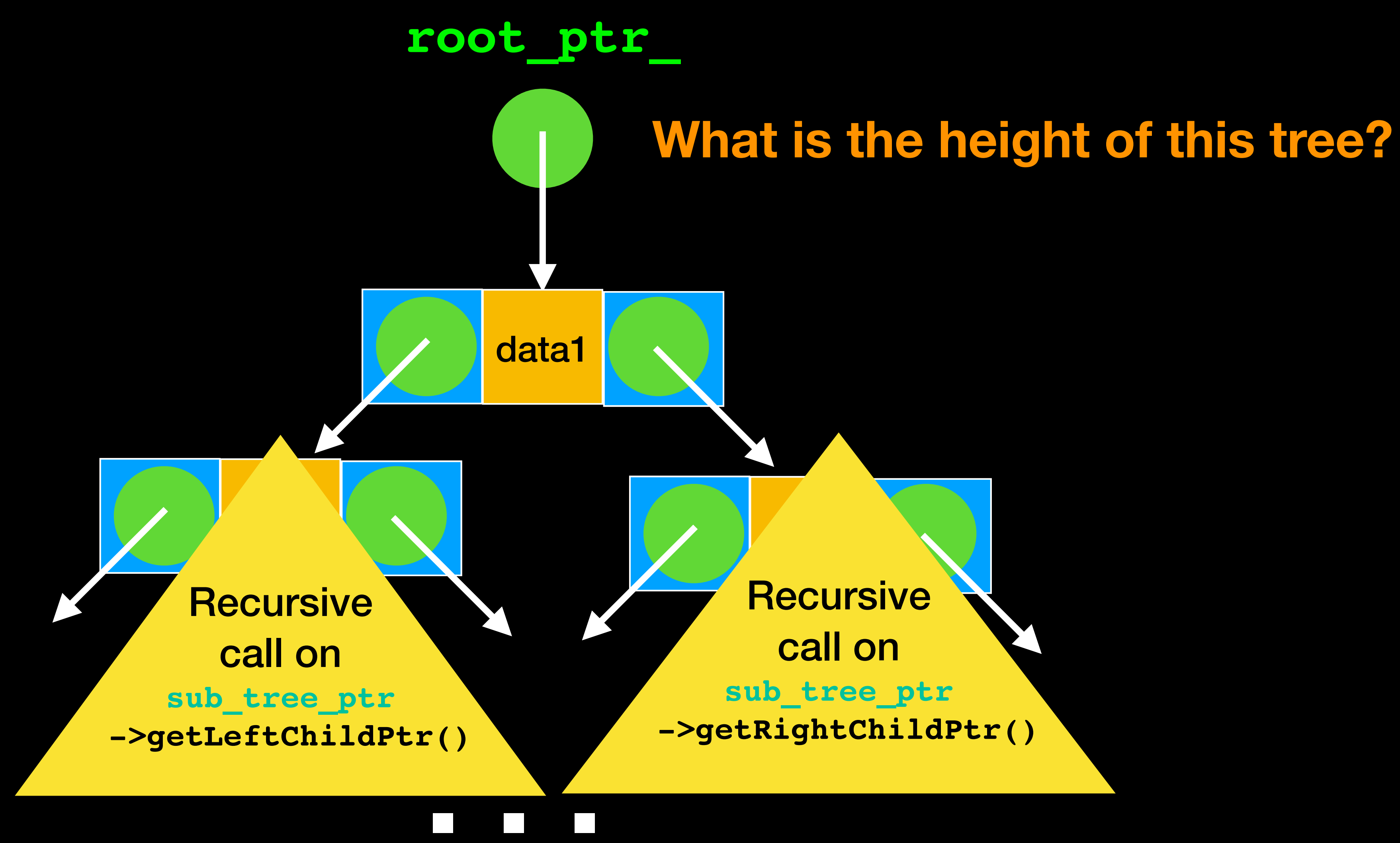
```
template<class T>
int BST<T>::getHeight() const
{
   return getHeightHelper(root_ptr_);
}  // end getHeight
```

Safe programming: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

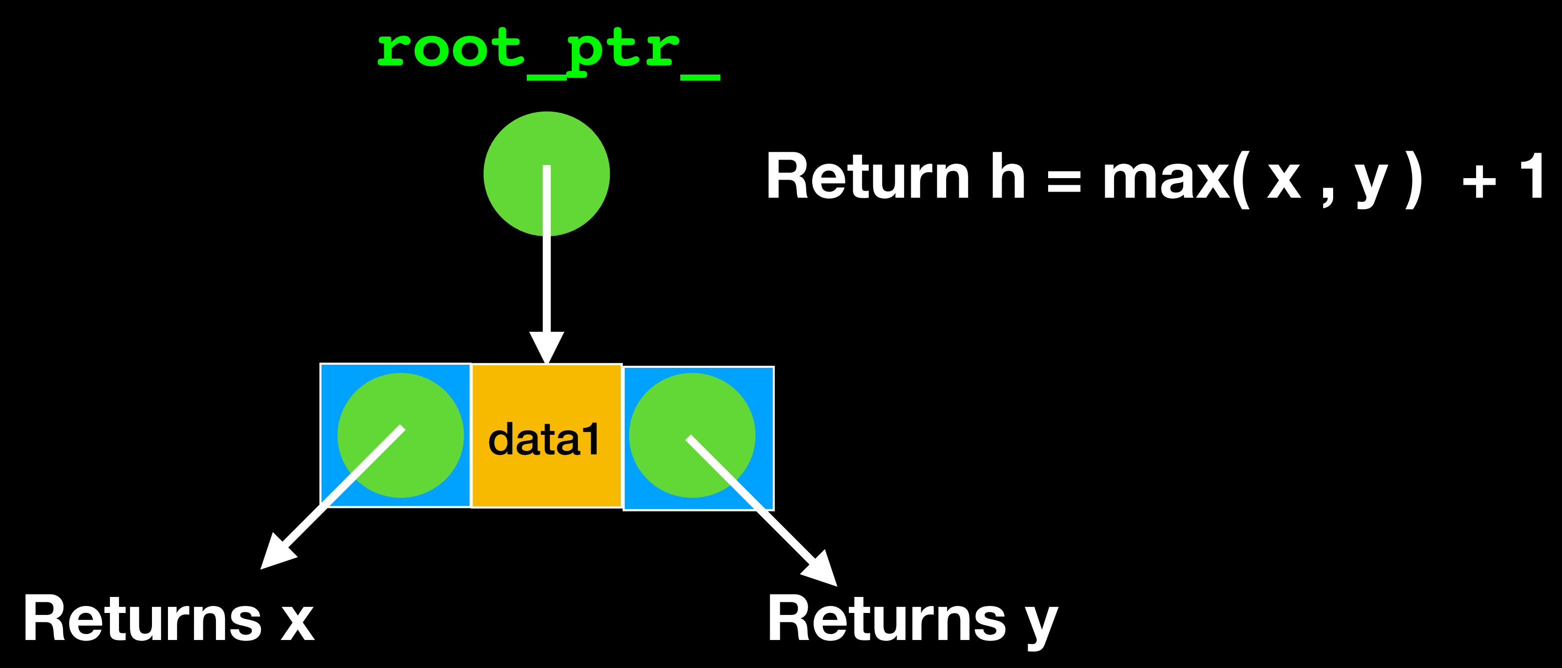
getHeightHelper(sub_tree_ptr)



getHeightHelper(sub_tree_ptr)



getHeightHelper(sub_tree_ptr)



getHeightHelper(sub_tree_ptr)

```
template<class T>
int BST<T>::getHeightHelper(std::shared_ptr<BinaryNode<T>> sub_tree_ptr) const
{
   if (sub_tree_ptr == nullptr)
      return 0;
   else
      return 1 + std::max(getHeightHelper(sub_tree_ptr->getLeftChildPtr()),
                          getHeightHelper(sub_tree_ptr->getRightChildPtr()));
}  // end getHeightHelper
```

Similarly: implement these at home!!!

```
int BST<T>::getNumberOfNodes() const
{ //try it at home!!!!}

int BST<T>::getNumberOfNodesHelper(std::shared_ptr
<BinaryNode<T>> sub_tree_ptr) {//try it at home!!!!}
```

`add` and `remove`

Key methods: determine order of data

Distinguish between different types of Binary Trees

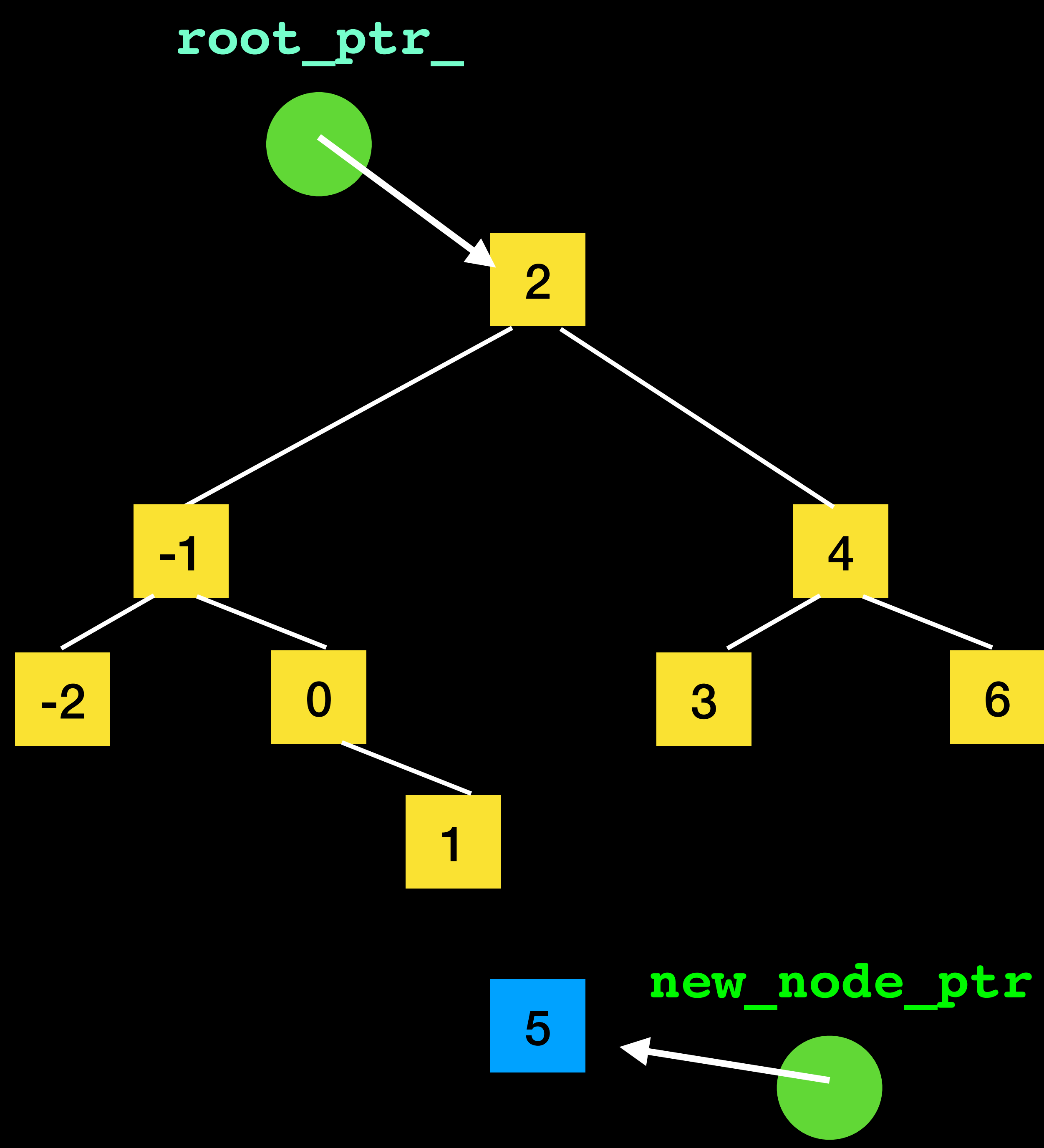
Implement the BST structural property

add

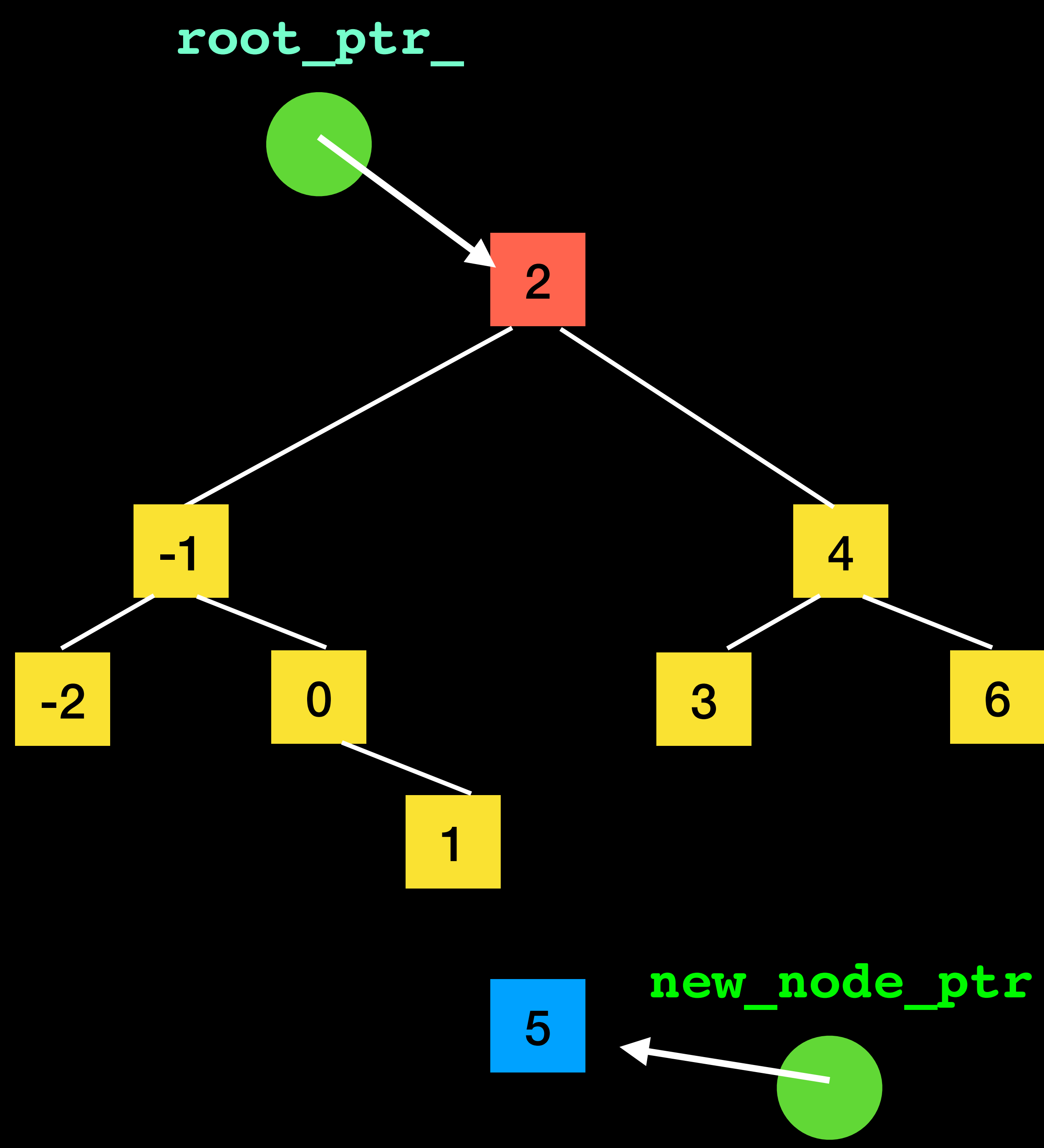
```
template<class T>
void BST<T>::add(const T& new_item)
{
   auto new_node_ptr =
                  std::make_shared<BinaryNode<T>>(new_item);
   placeNode(root_ptr_, new_node_ptr);

}  // end add
```

<u>Safe programming</u>: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

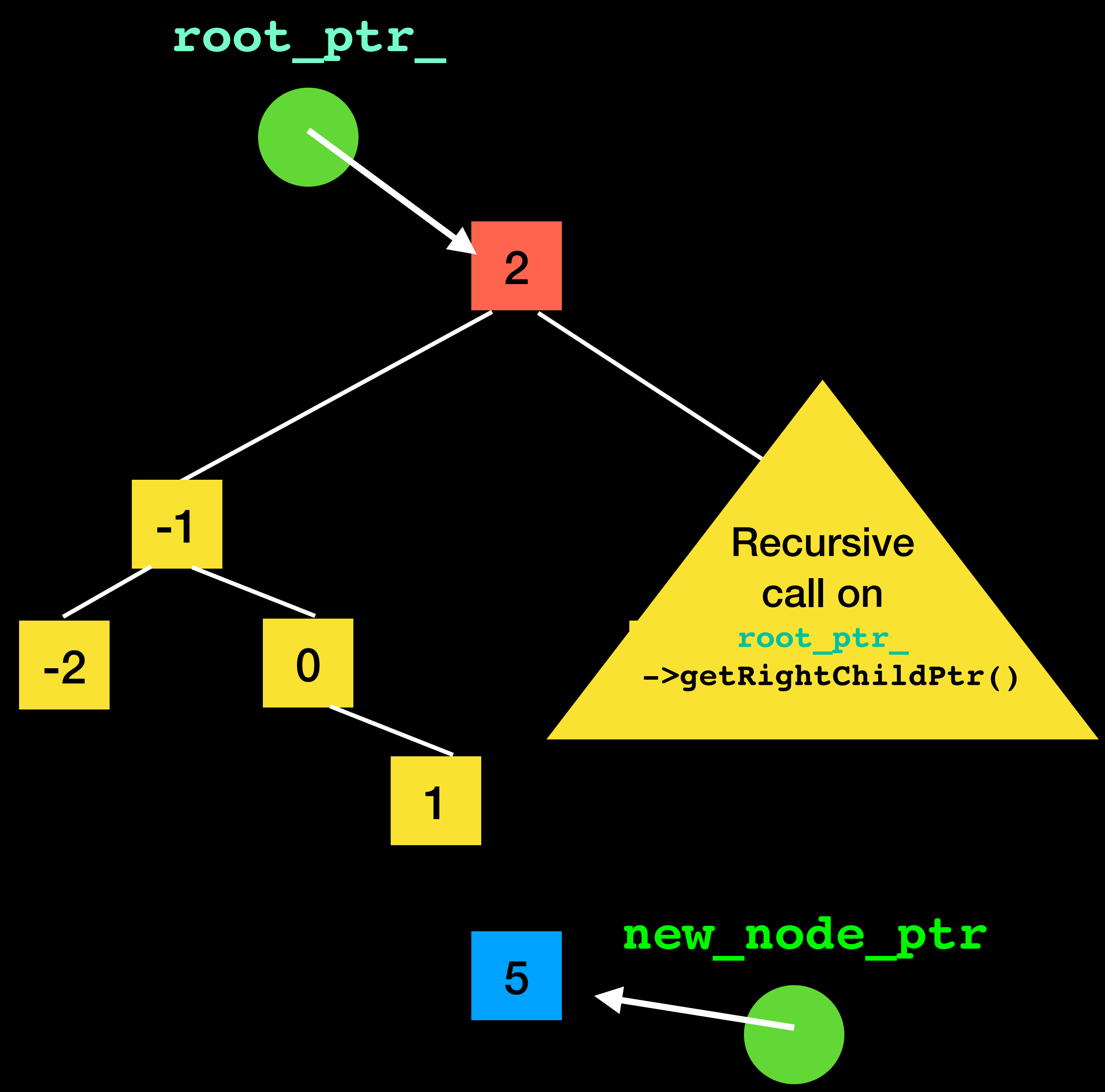
```
placeNode(root_ptr_, new_node_ptr);
```



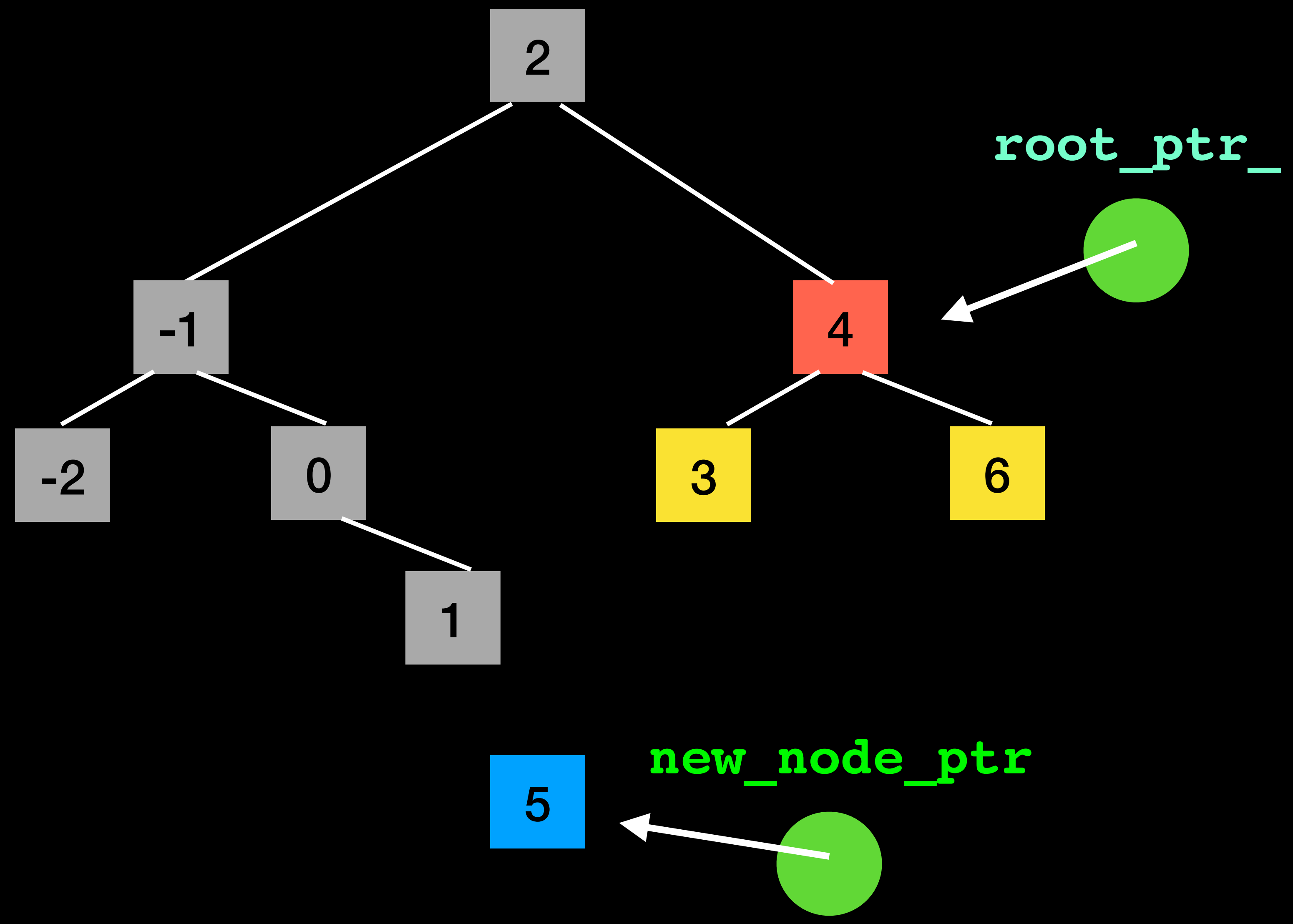
```
placeNode(root_ptr_, new_node_ptr);
```



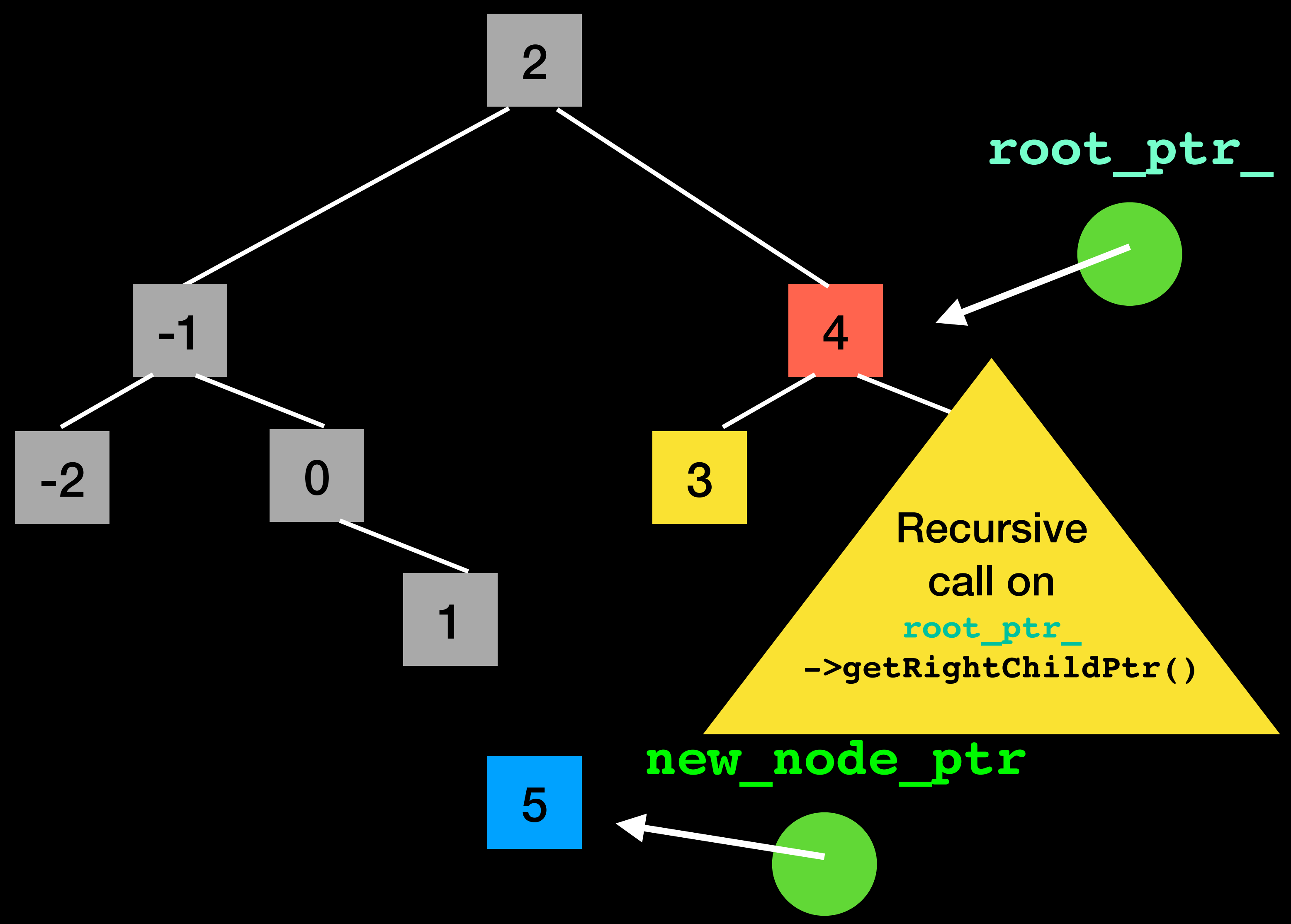
```
placeNode(root_ptr_, new_node_ptr);
```



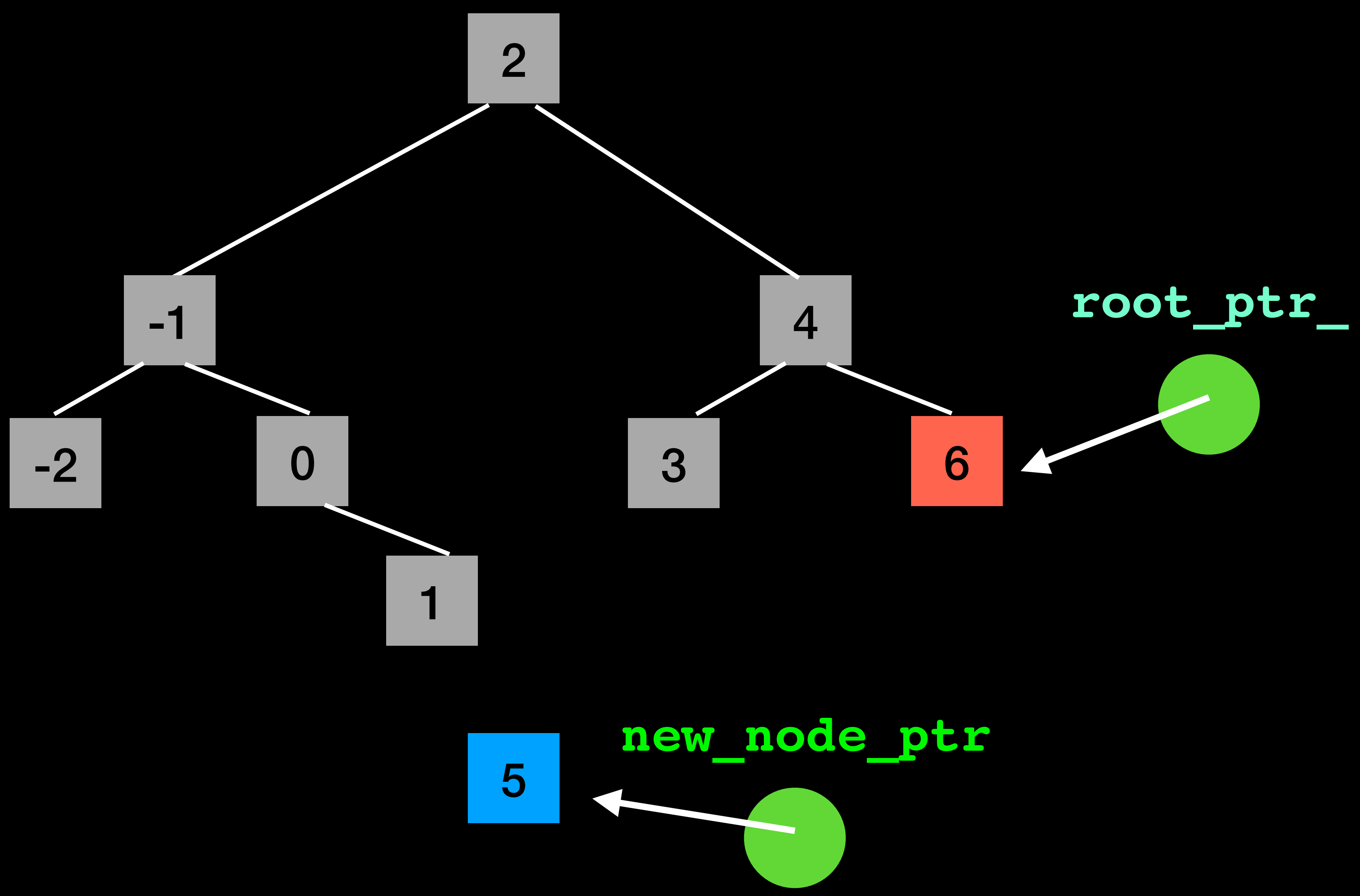
```
placeNode(root_ptr_, new_node_ptr);
```



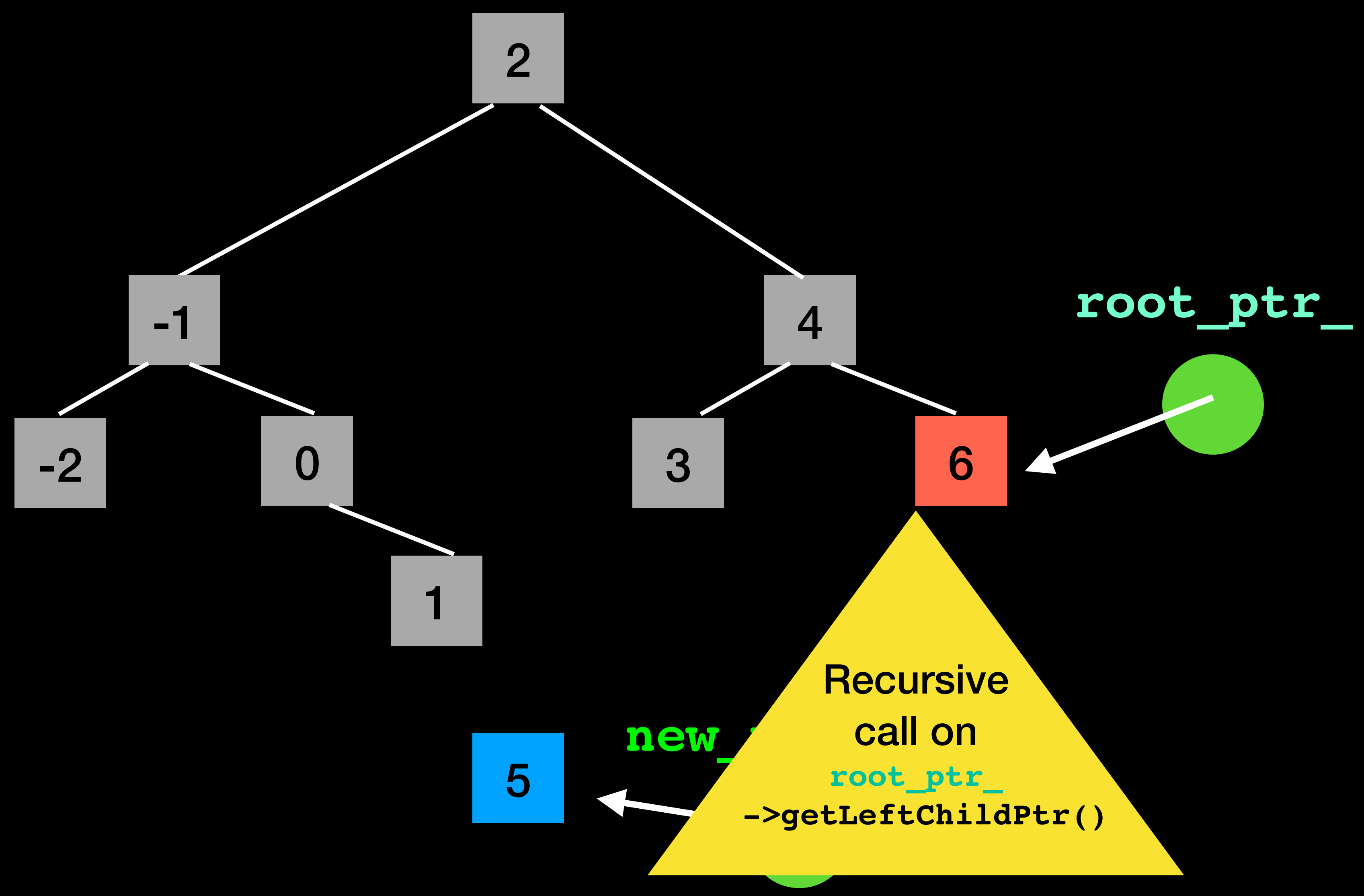
```
placeNode(root_ptr_, new_node_ptr);
```



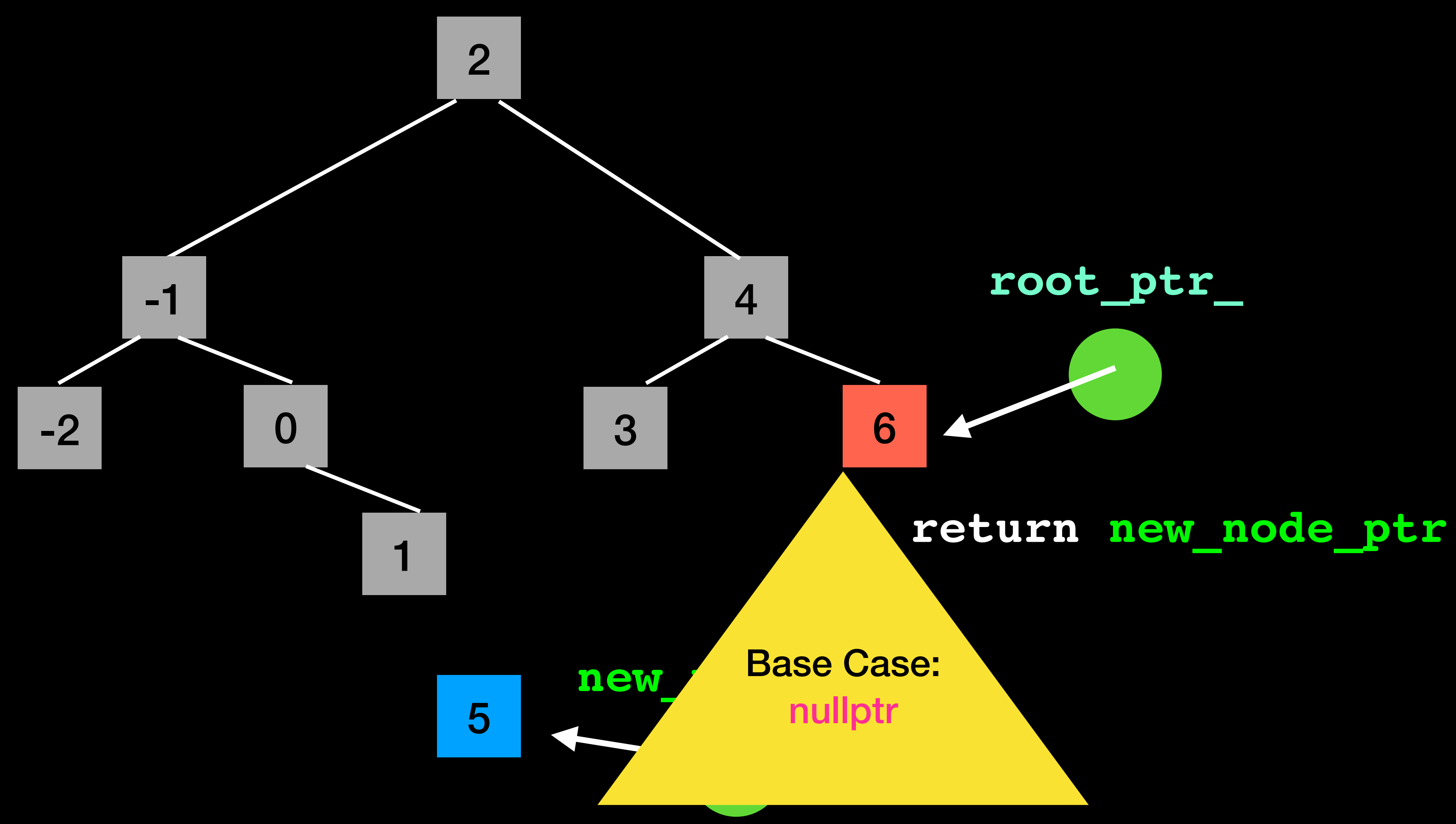
```
placeNode(root_ptr_, new_node_ptr);
```



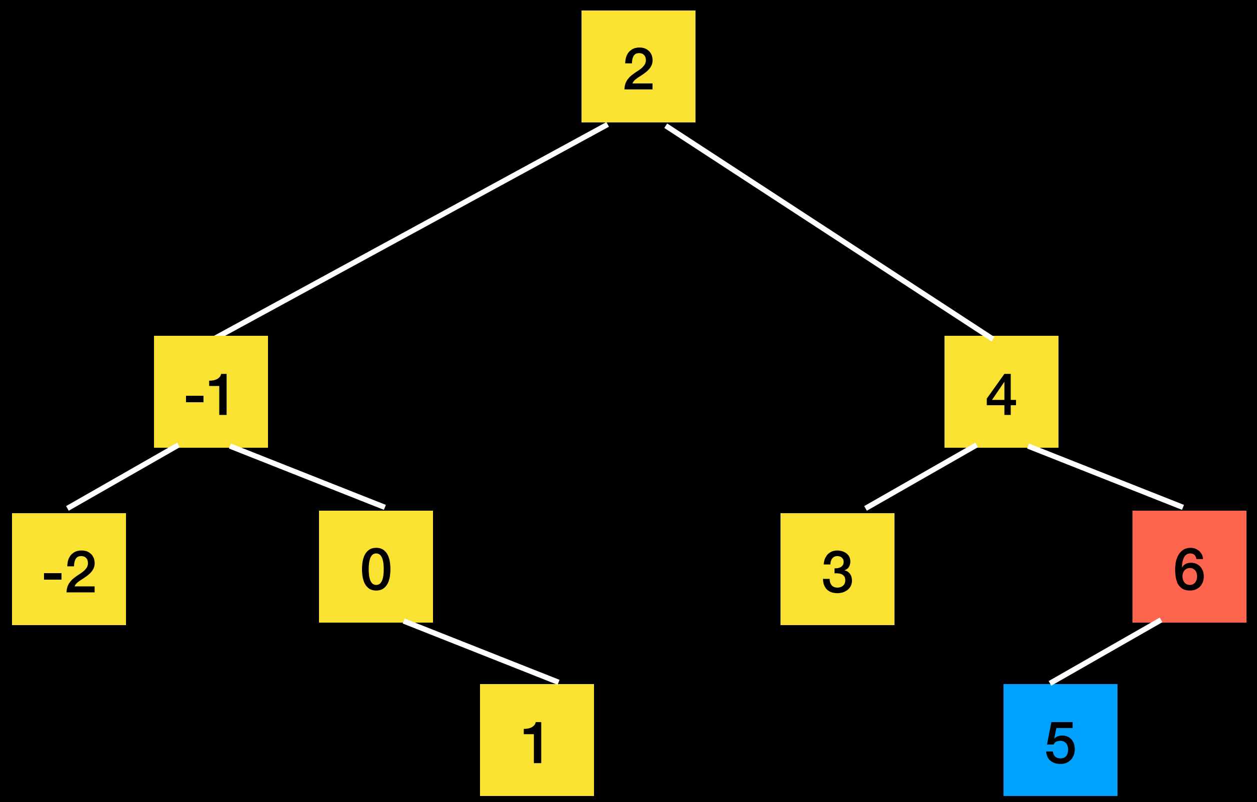
```
placeNode(root_ptr_, new_node_ptr);
```



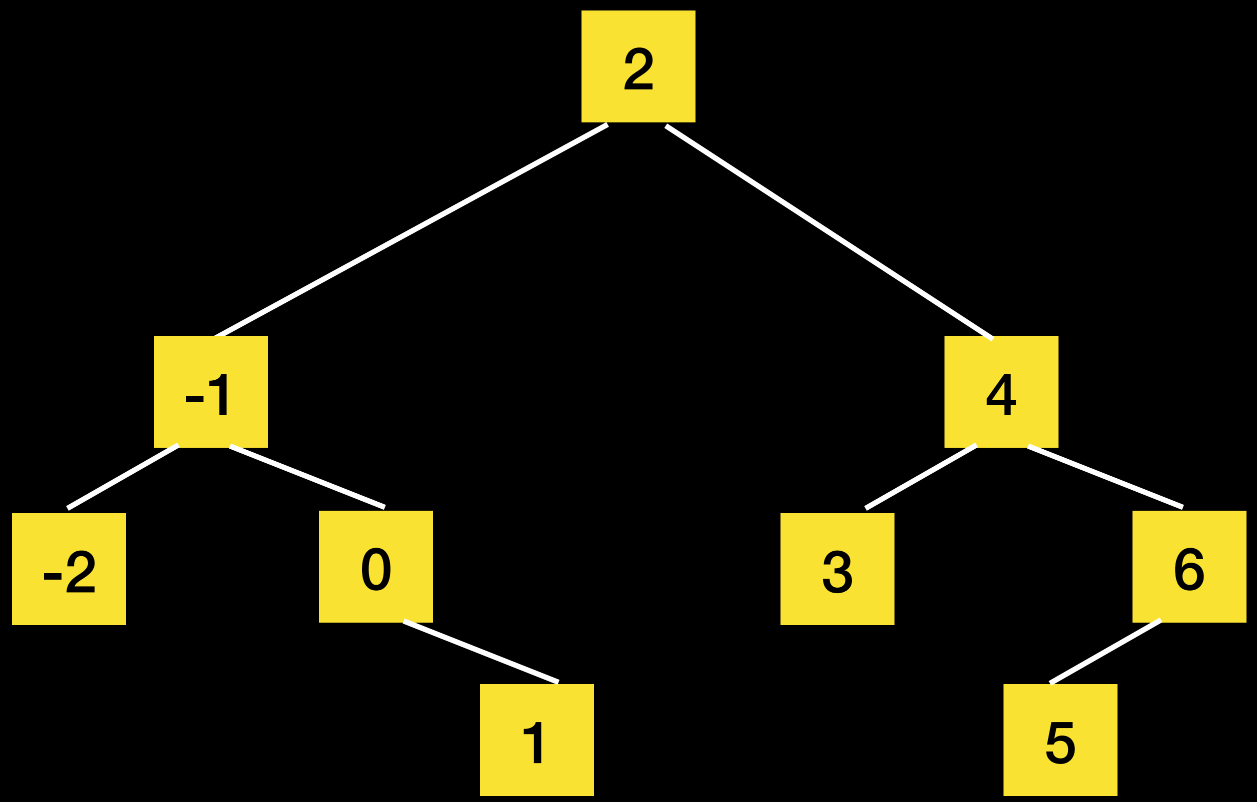
```
placeNode(root_ptr_, new_node_ptr);
```



```
placeNode(root_ptr_, new_node_ptr);
```



```
placeNode(root_ptr_, new_node_ptr);
```



add helper function

```
template<class T>
auto BST<T>::placeNode(std::shared_ptr<BinaryNode<T>> subtree_ptr,

                              std::shared_ptr<BinaryNode<T>> new_node_ptr)
{
   if (subtree_ptr == nullptr)
      return new_node_ptr;  //base case
   else
   {
      if (subtree_ptr->getItem() > new_node_ptr->getItem())
         subtree_ptr->setLeftChildPtr(placeNode(subtree_ptr->getLeftChildPtr(),
                                                            new_node_ptr));
      else
         subtree_ptr->setRightChildPtr(placeNode(subtree_ptr->getRightChildPtr(),
                                                            new_node_ptr));

      return subtree_ptr;

   }  // end if
}  // end placeNode
```

remove

```
template<class T>
bool BST<T>::remove(const T& target)
{
   bool is_successful = false;
   // call may change is_successful
   root_ptr_ = removeValue(root_ptr_, target, is_successful);
   return is_successful;
}  // end remove
```

<u>Safe programming</u>: the public method does not take pointer parameter.
Only protected/private methods have access to pointers and may modify tree structure

`remove` helper function

Looks for the value to remove

```
template<class T>
auto BST<T>::removeValue(std::shared_ptr<BinaryNode<T>>
      subtree_ptr, const T target,  bool& success)
{
   if (subtree_ptr == nullptr)
   {
      // Not found here
      success = false;
      return subtree_ptr;
   }
   if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
   }
```

target not in tree

Found target now remove the node

`remove` helper function cont.ed

```
    else
    {
       if (subtree_ptr->getItem() > target)
       {
          // Search the left subtree
          subtree_ptr->setLeftChildPtr(removeValue(subtree_ptr
                                    ->getLeftChildPtr(), target, success));
       }
       else
       {
          // Search the right subtree
          subtree_ptr->setRightChildPtr(removeValue(subtree_ptr
                                    ->getRightChildPtr(), target, success));
       }
       return subtree_ptr;
    }  // end if
}  // end removeValue
```

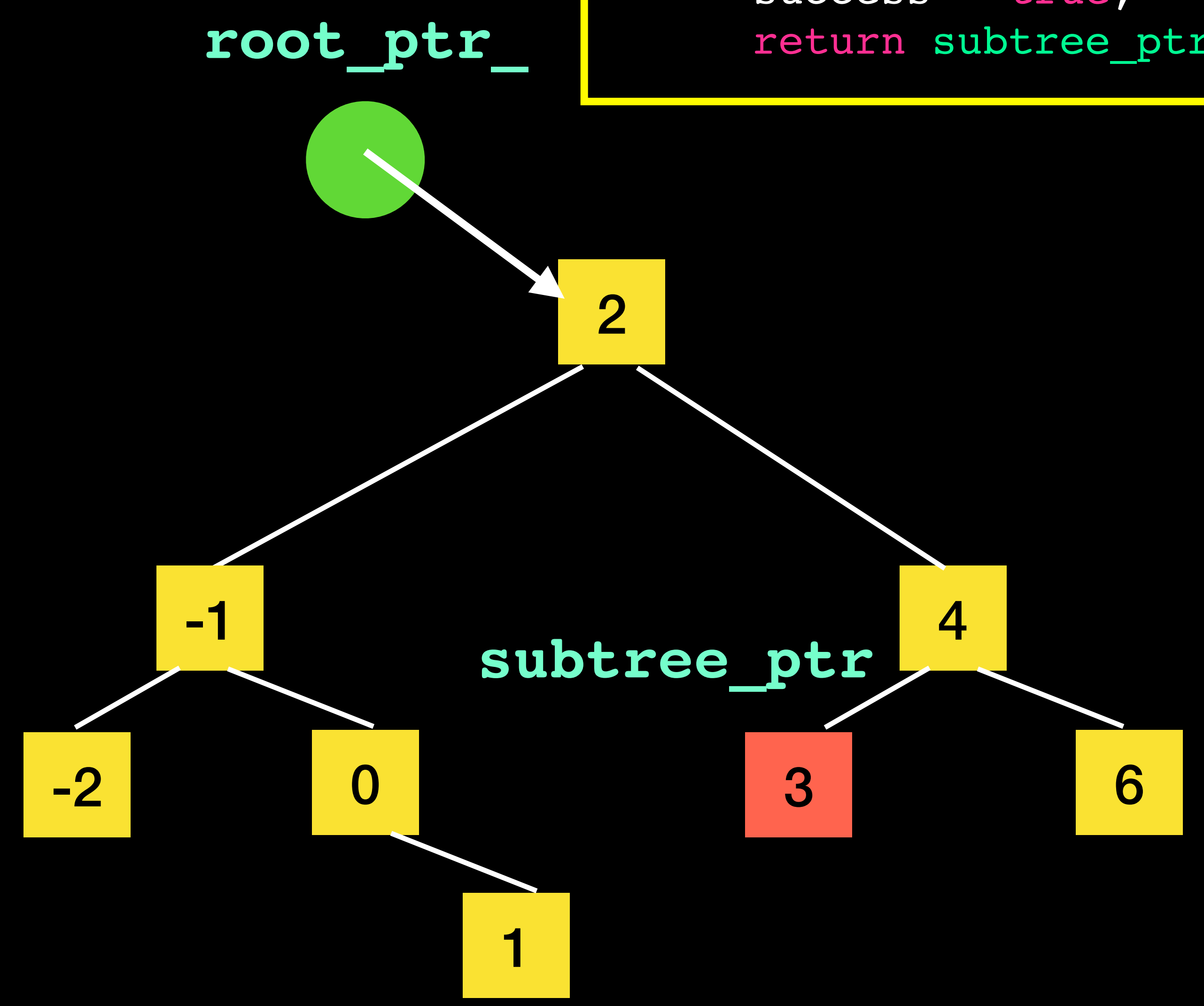
Search for target in left subtree

Search for target in right subtree

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

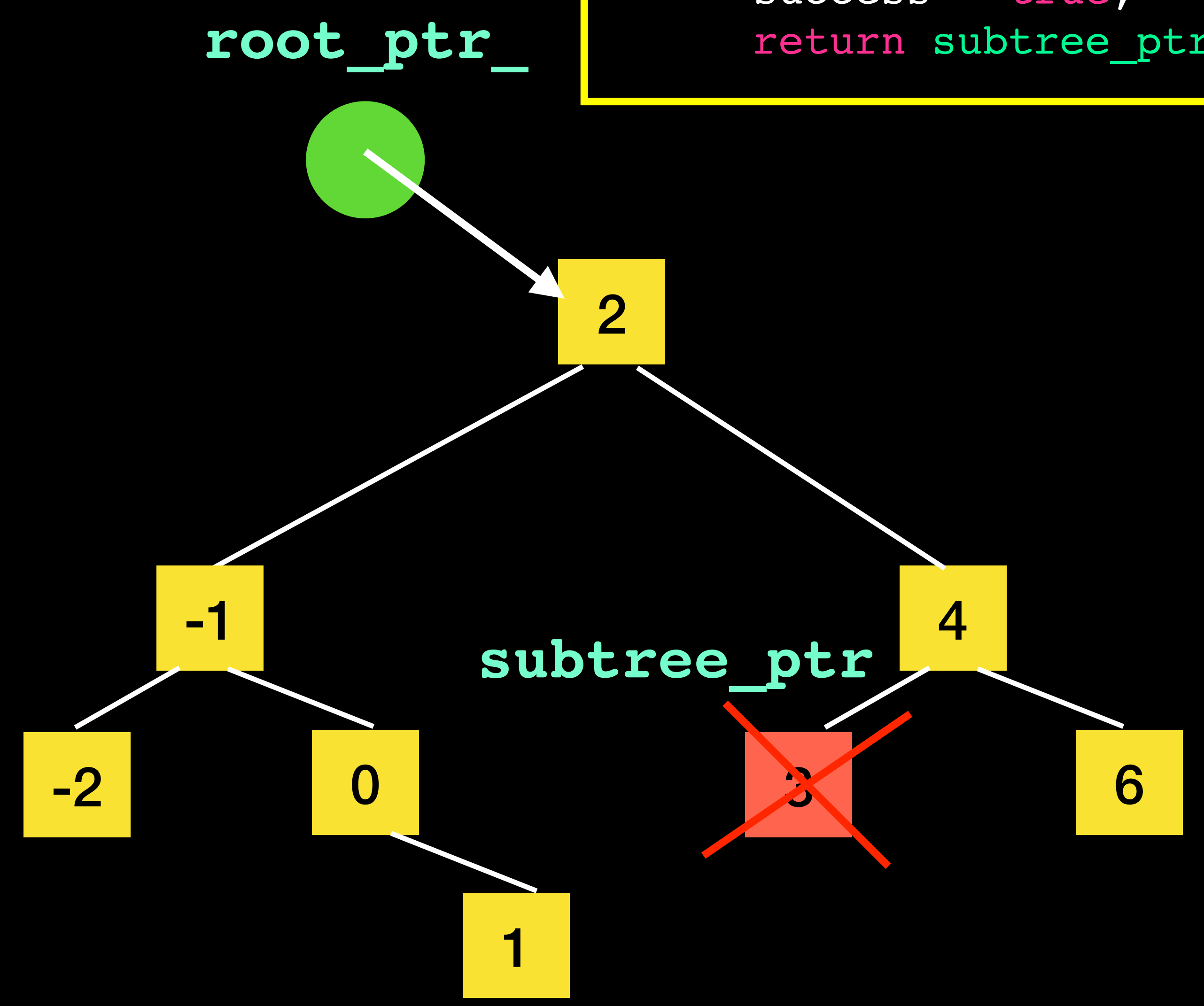
Case 1: target is a leaf



removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

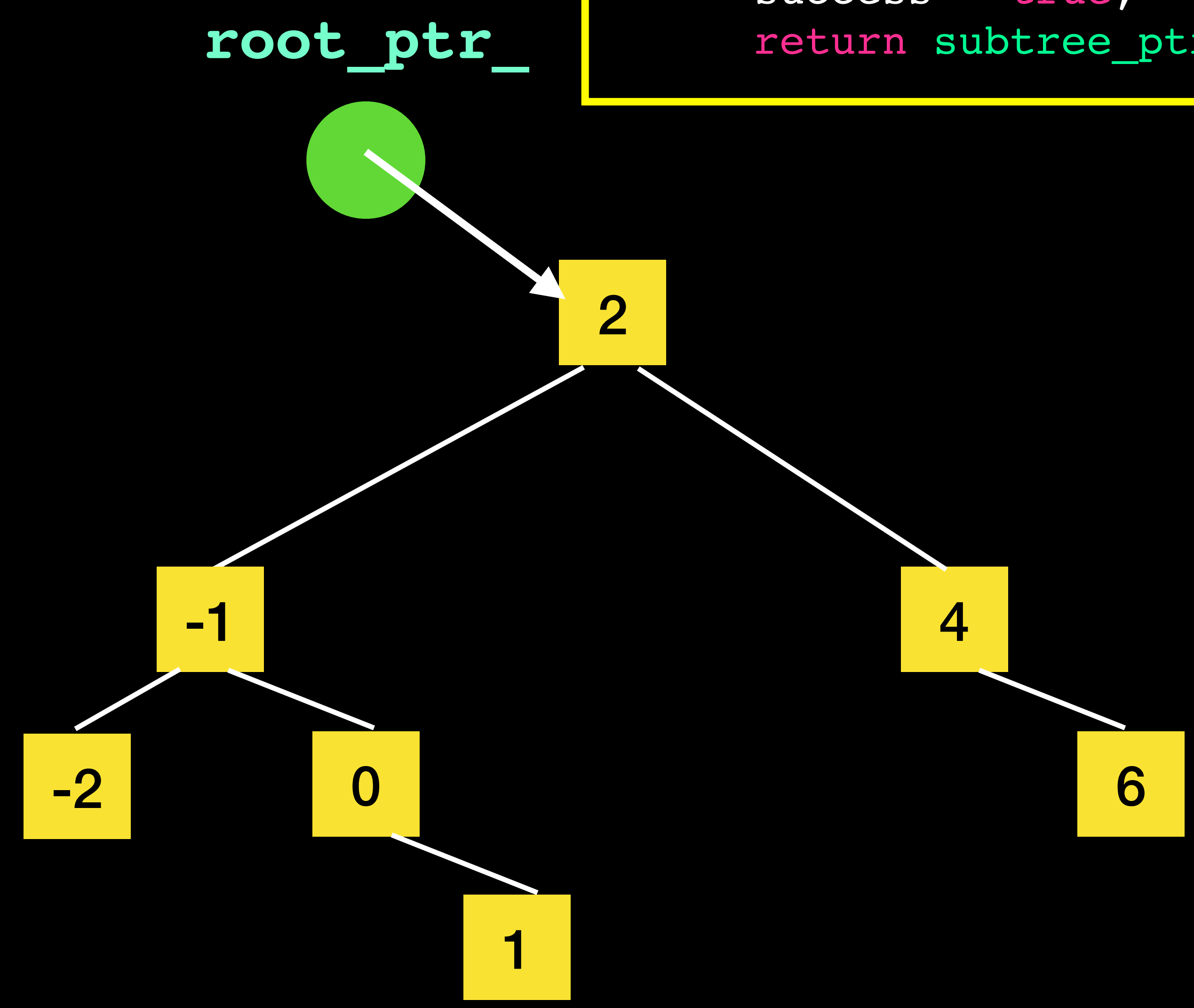
Case 1: target is a leaf



removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
    {
        // Item is in the root of this subtree
        subtree_ptr = removeNode(subtree_ptr);
        success = true;
        return subtree_ptr;
```

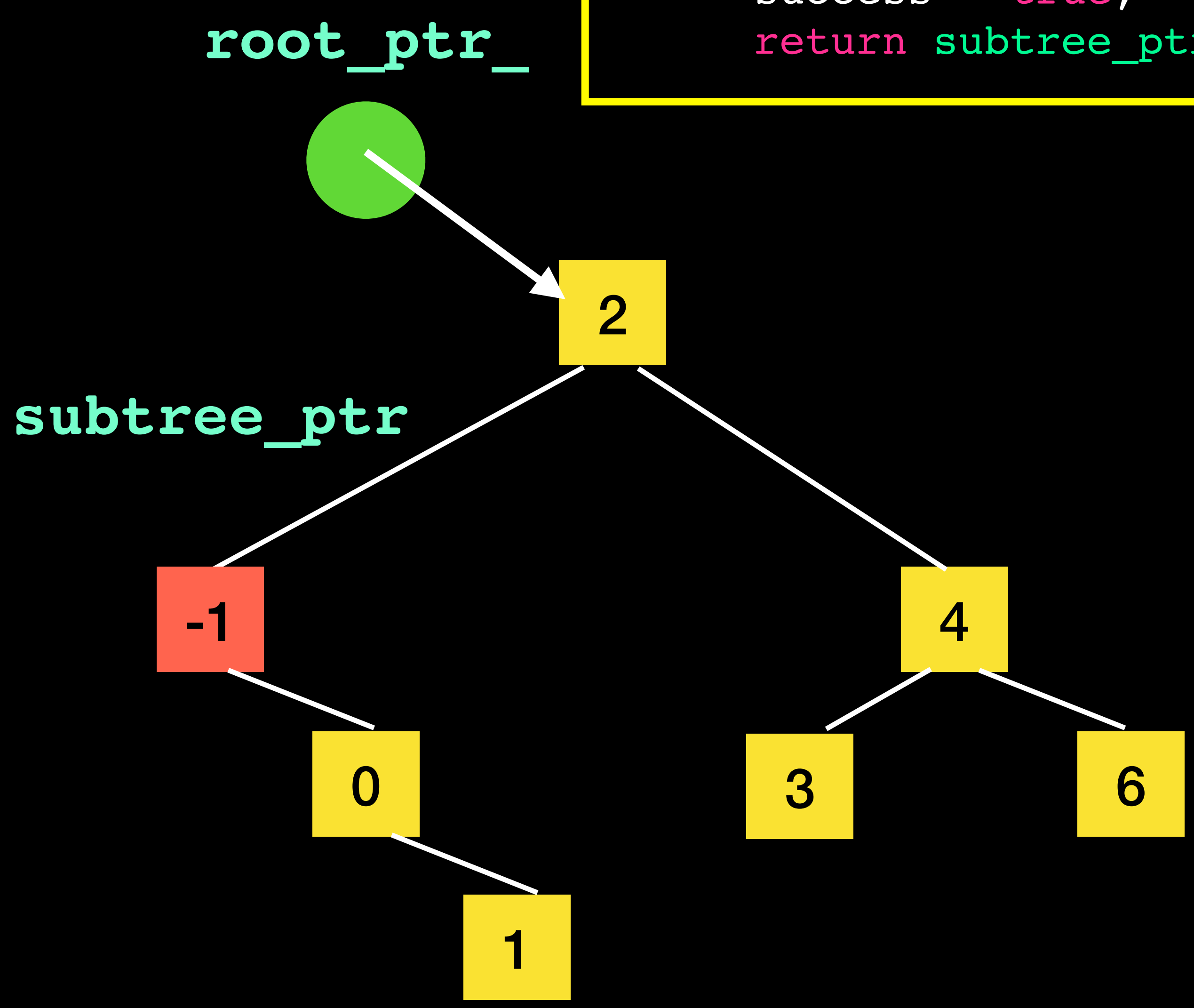
Case 1: target is a leaf



removeNode(subtree_ptr);

Case 2: target has 1 child
Left and right case are symmetric

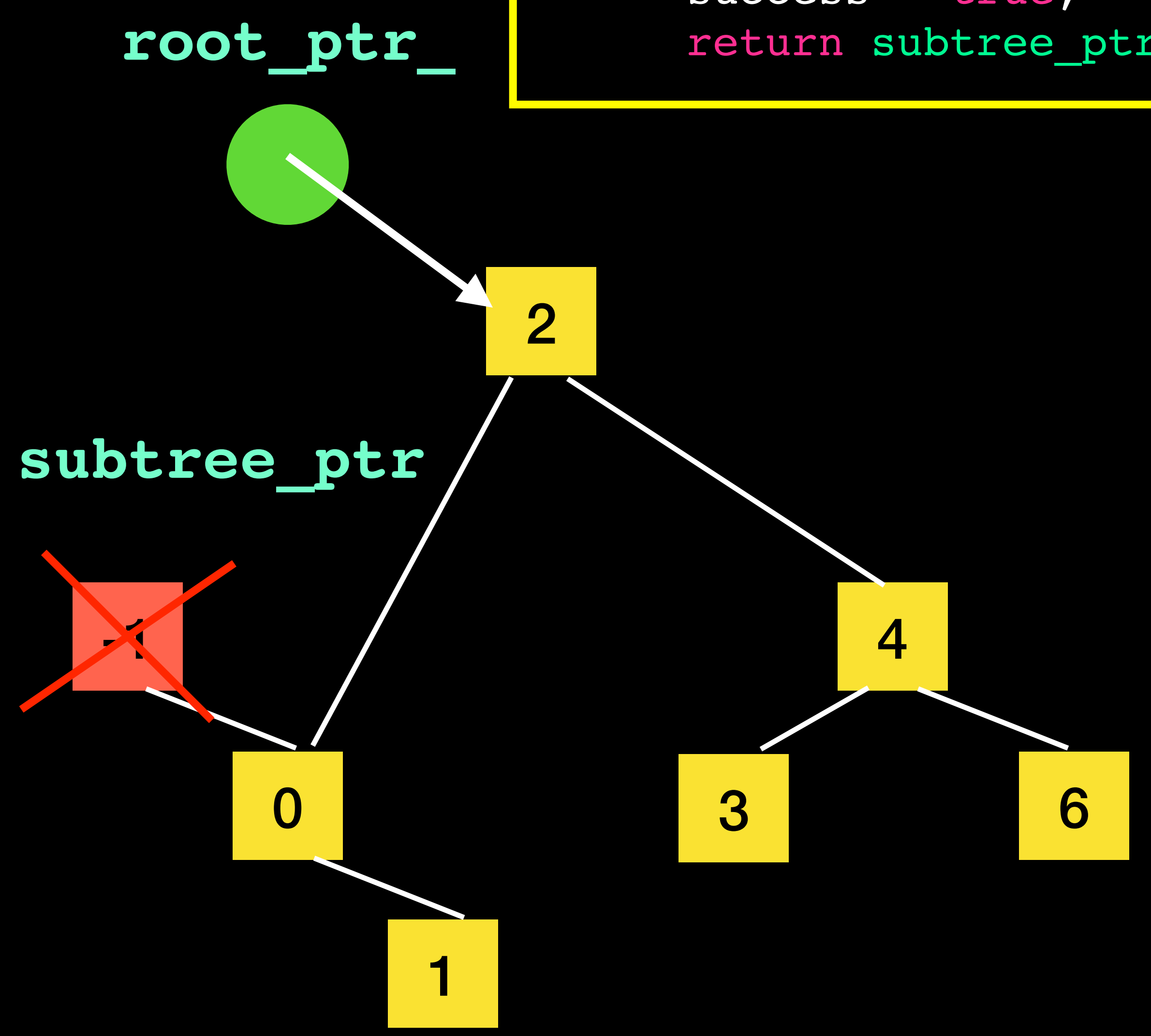
```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```



removeNode(subtree_ptr);

Case 2: target has 1 child
Left and right case are symmetric

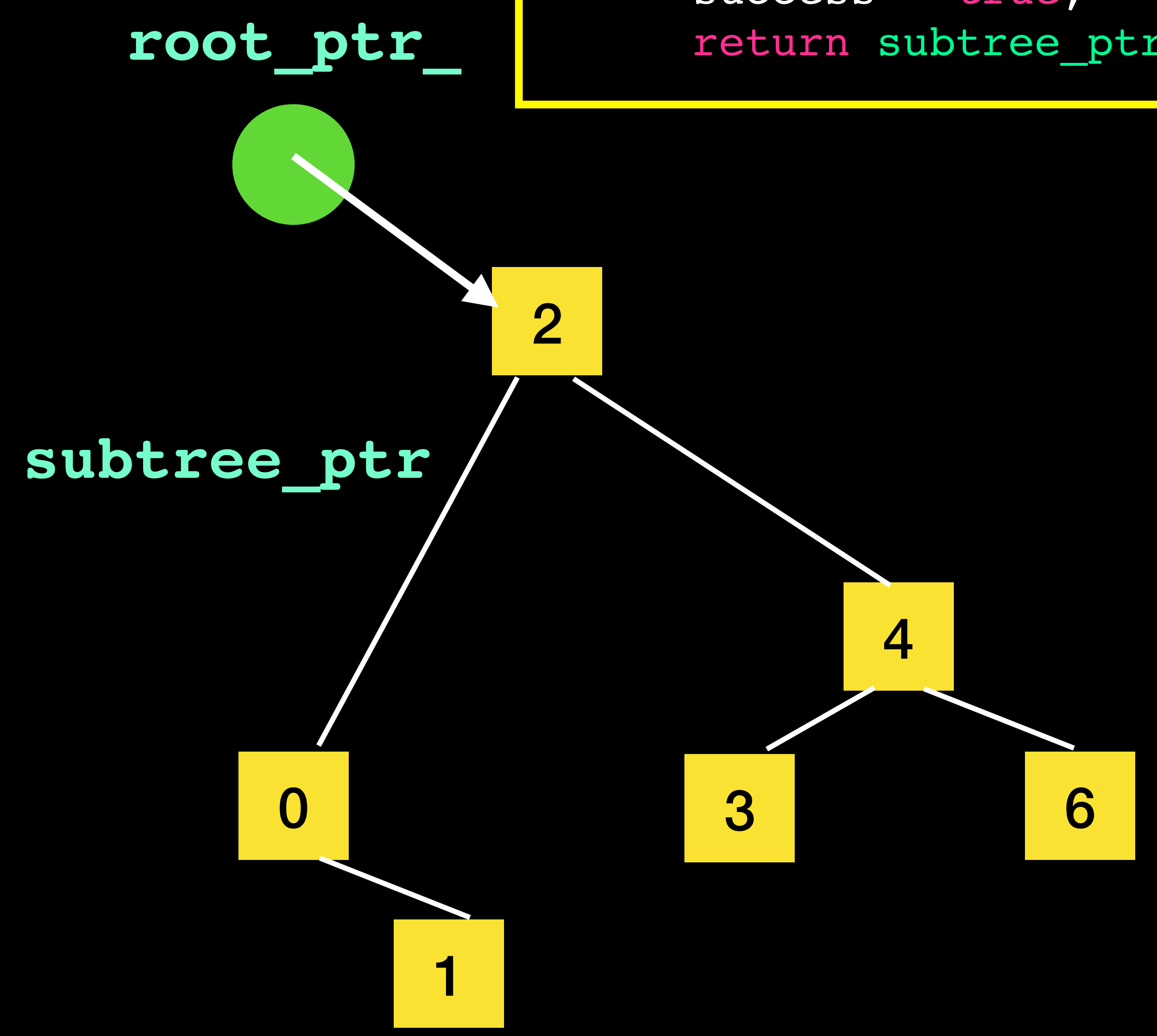
```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```



removeNode(subtree_ptr);

Case 2: target has 1 child
Left and right case are symmetric

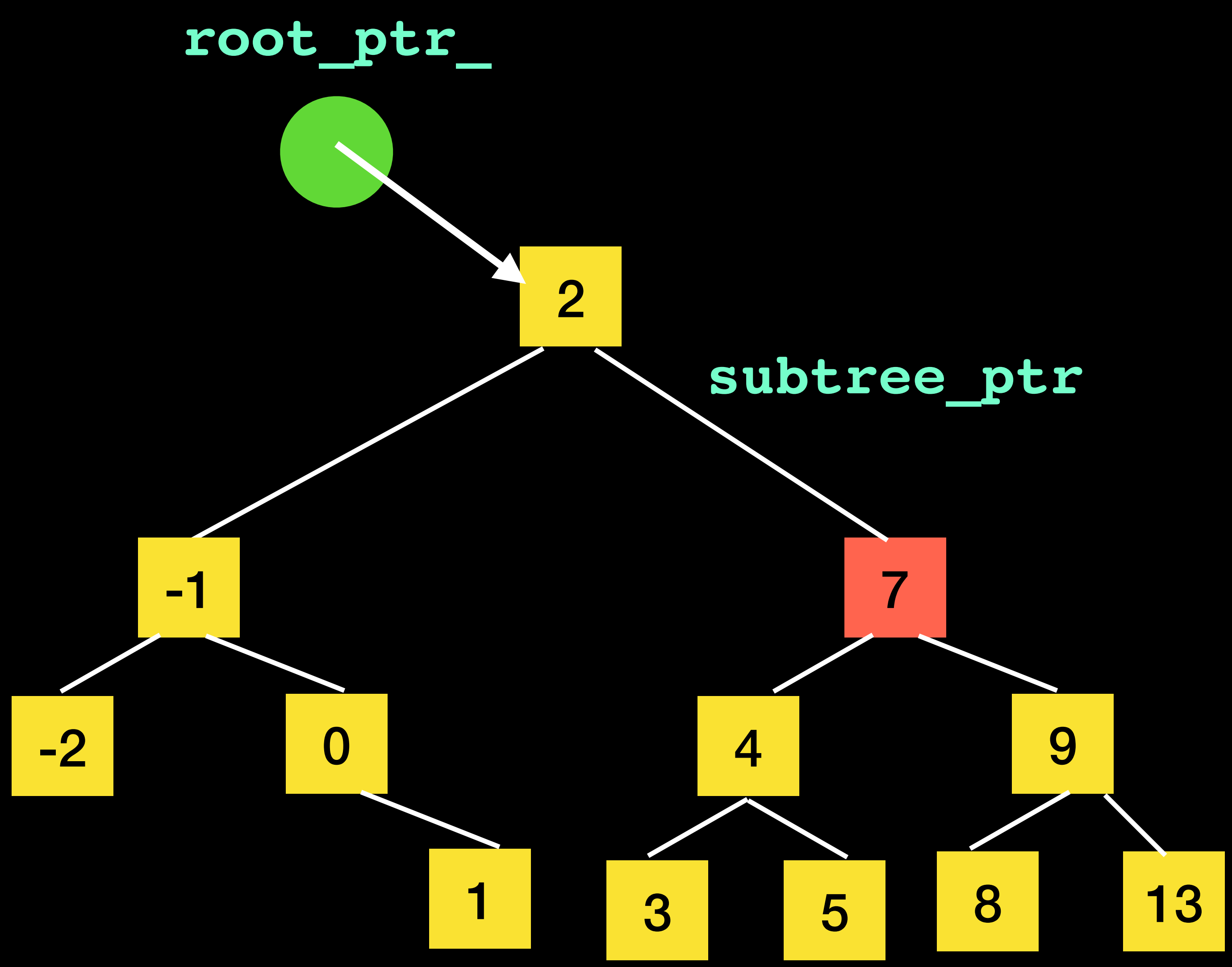
```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```



Lecture Activity

How would you remove node 7?

<u>Case 3</u>: target has 2 children

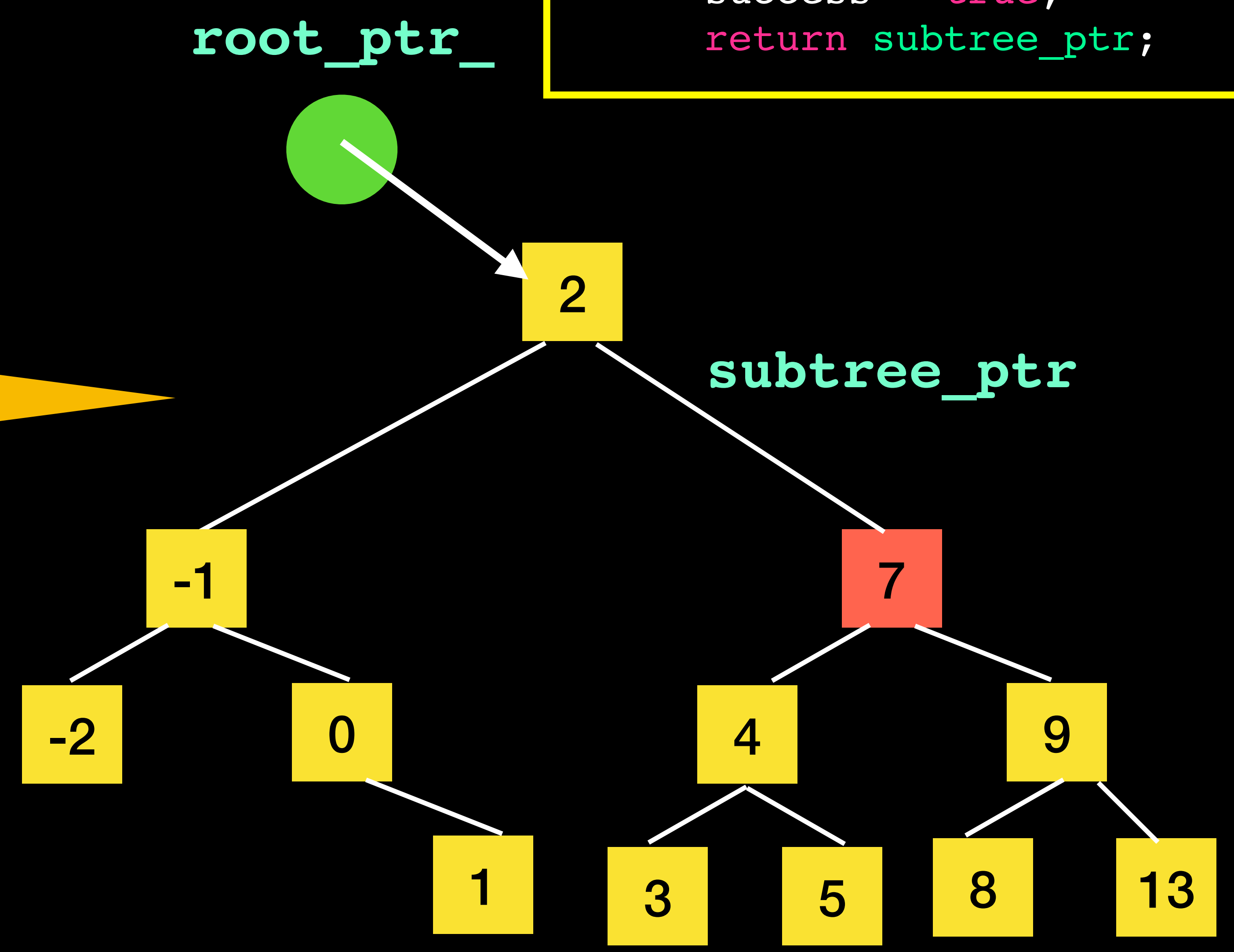


removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.

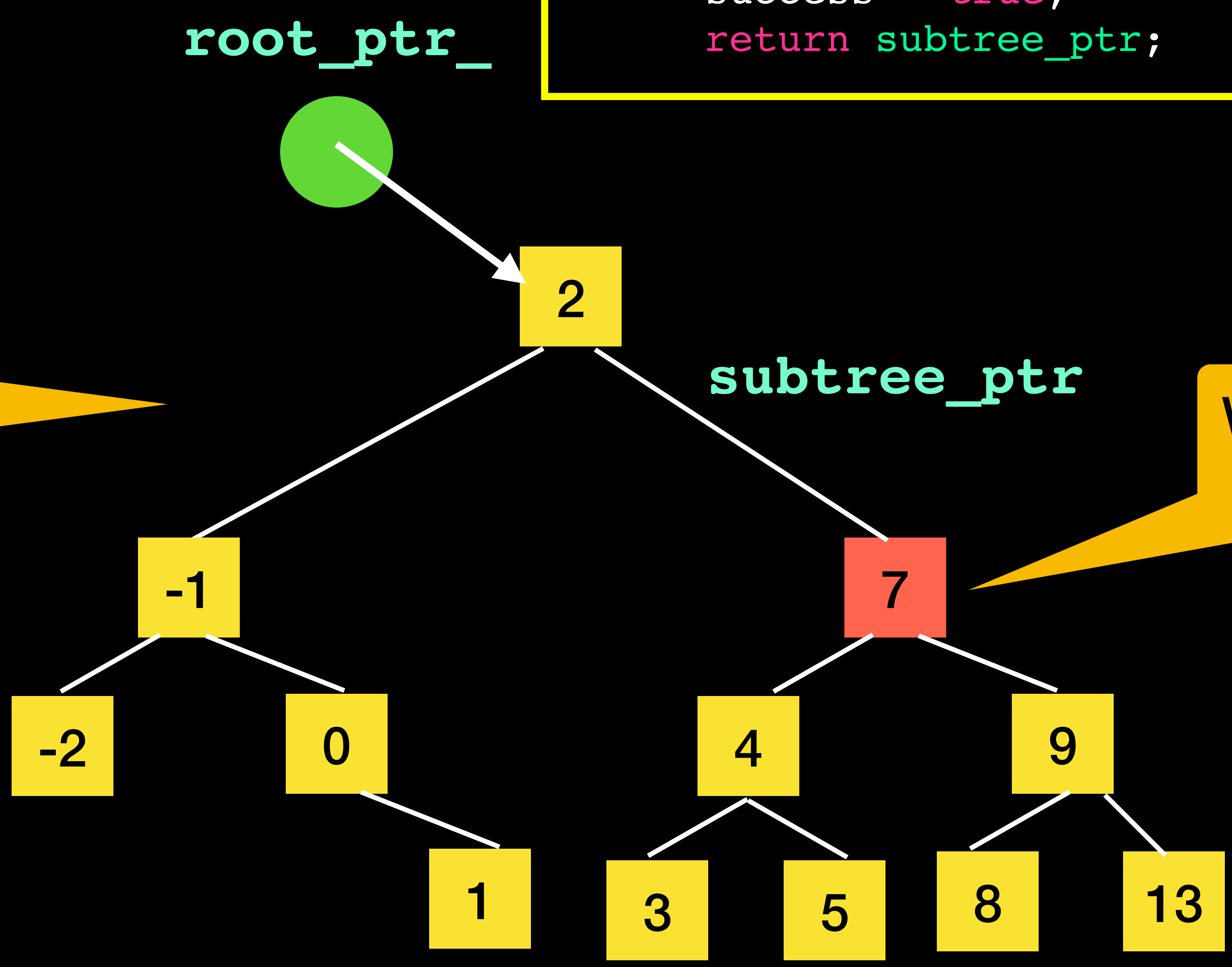


removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.

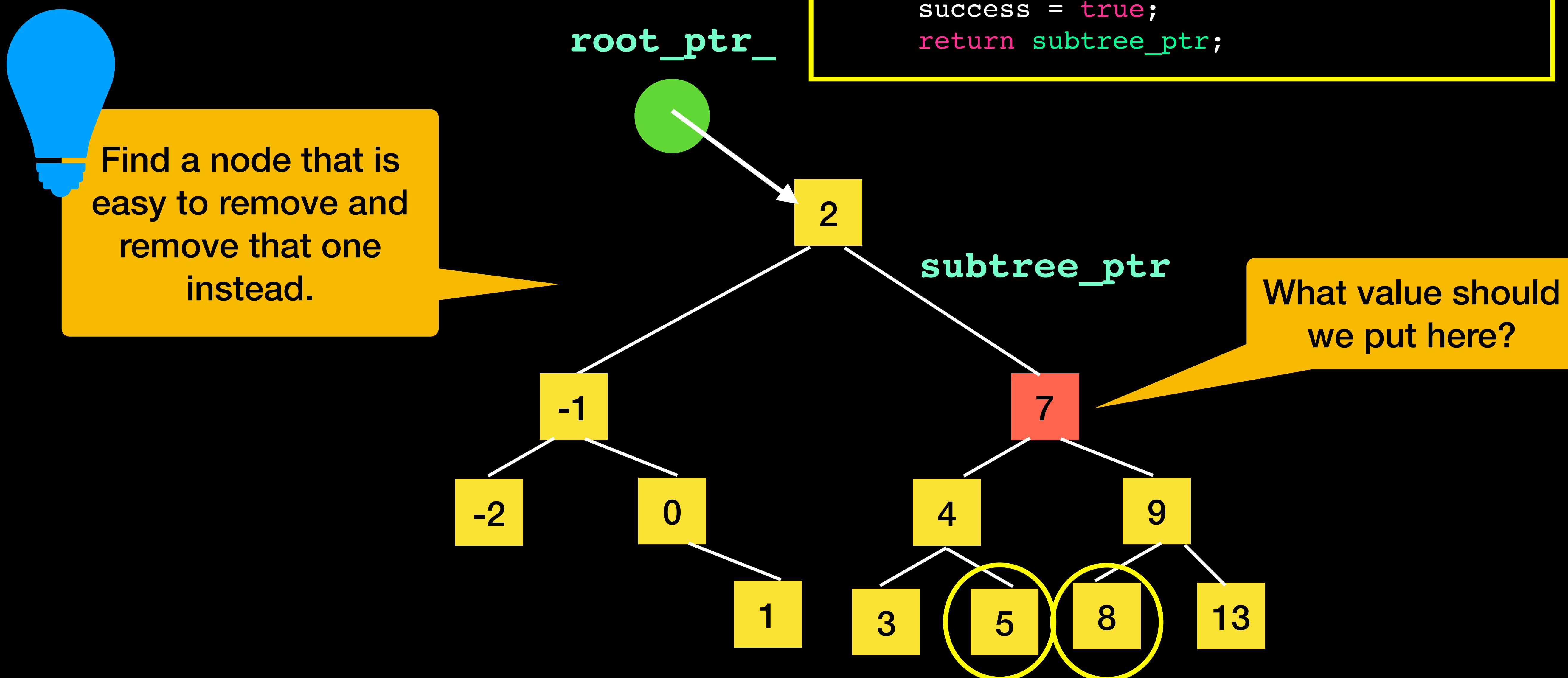


What value should we put here?

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

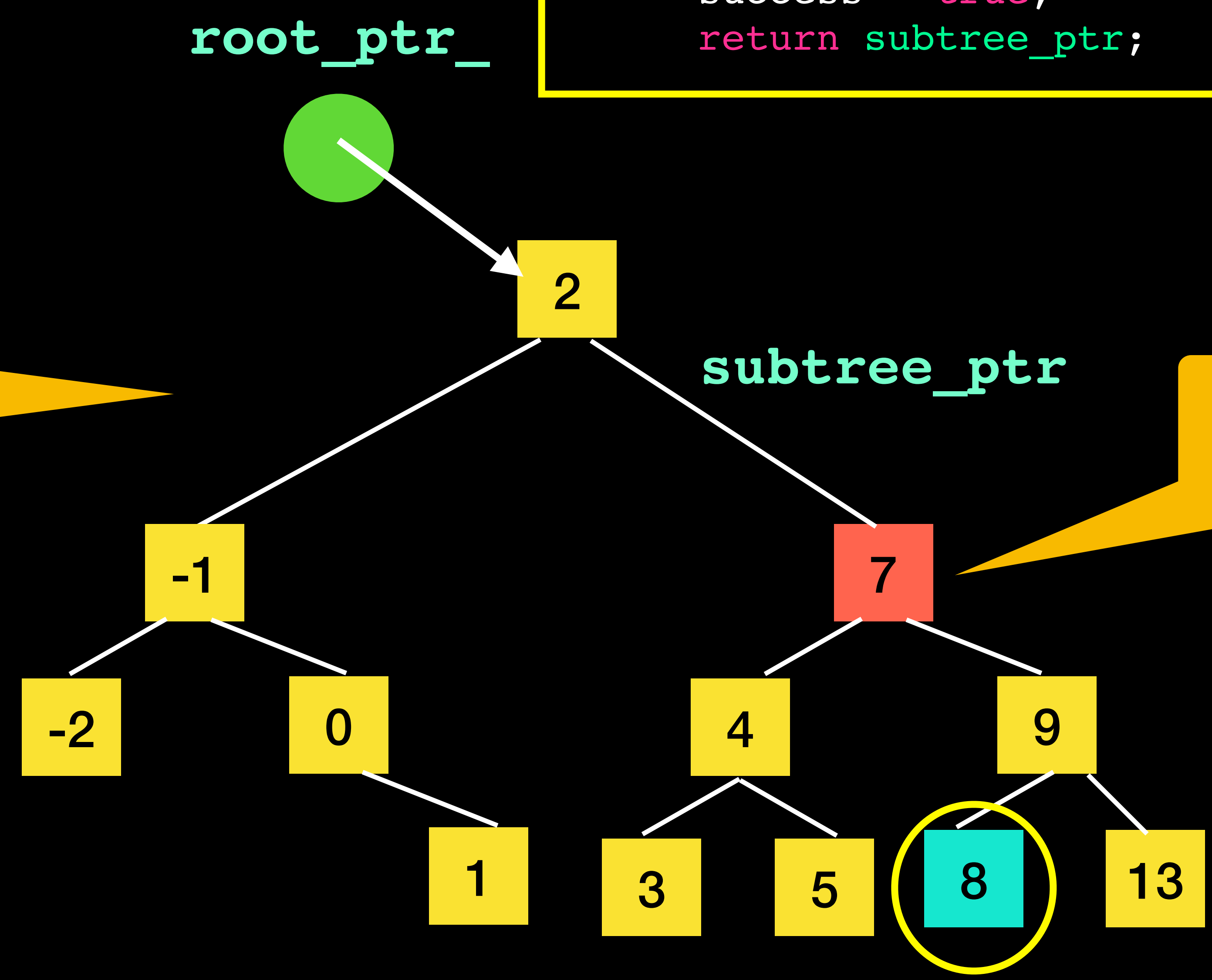


removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



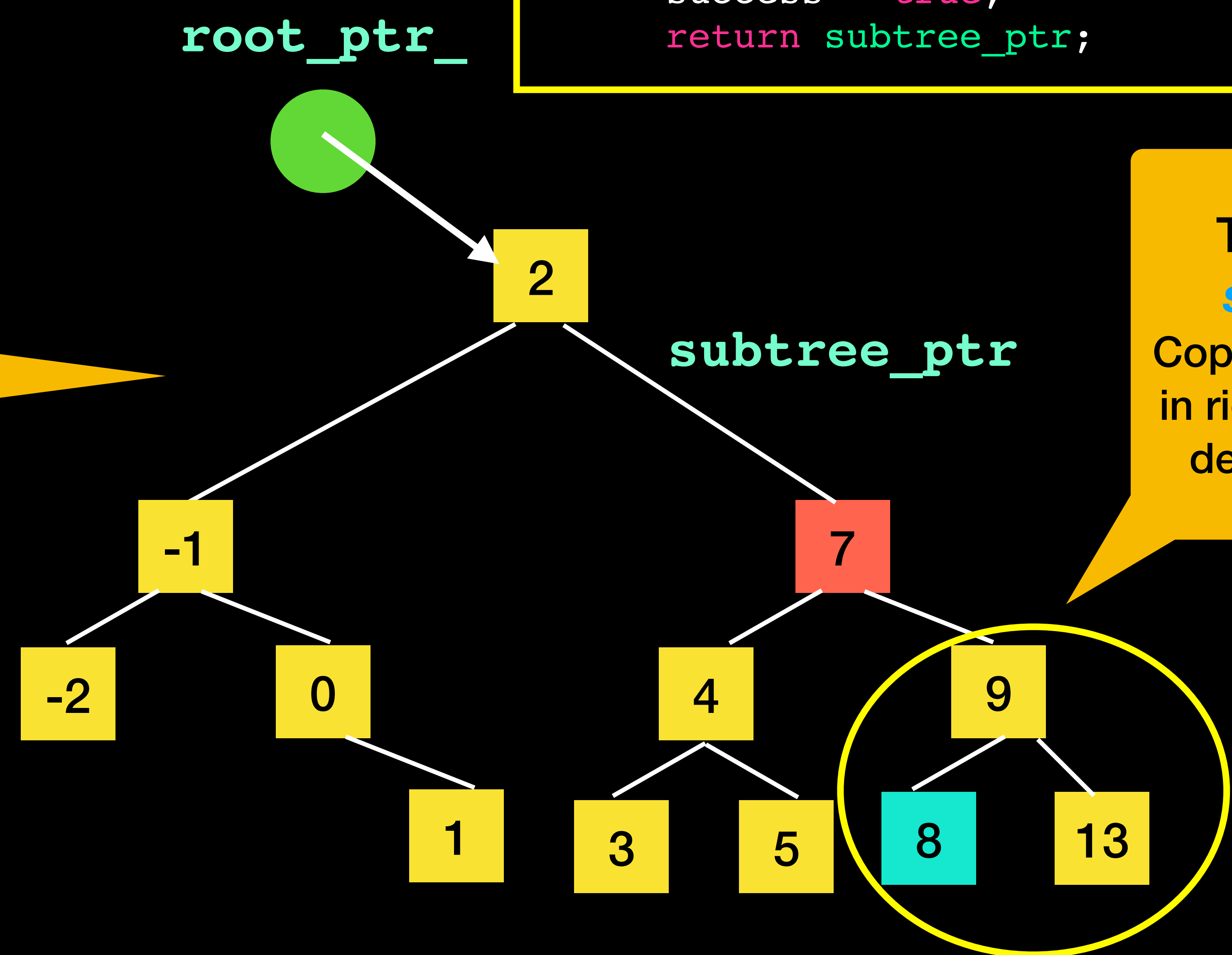
The *inorder successor*

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

<u>Case 3</u>: target has 2 children

Find a node that is easy to remove and remove that one instead.



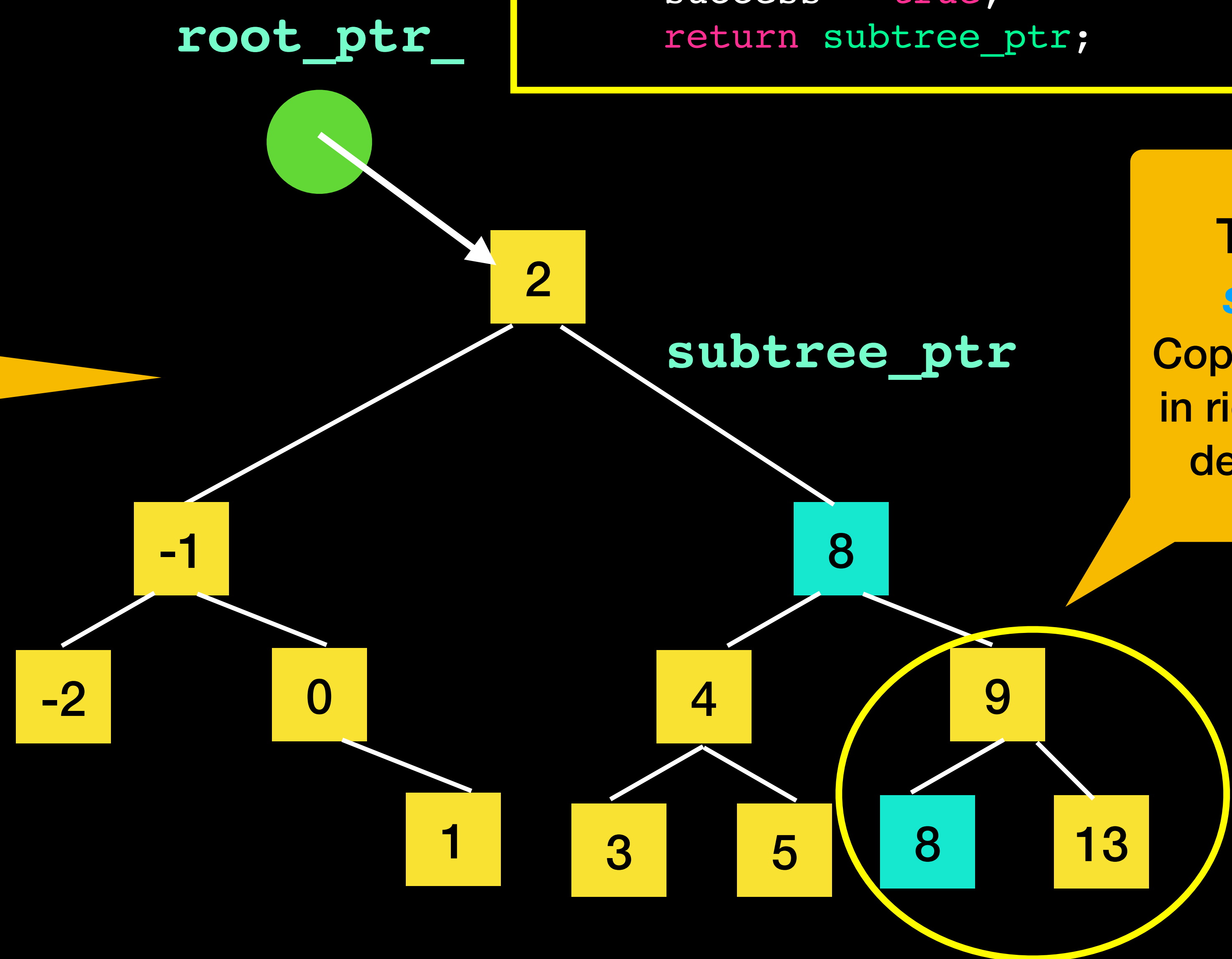
The *inorder successor:*
Copy smallest value in right subtree and delete that node

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



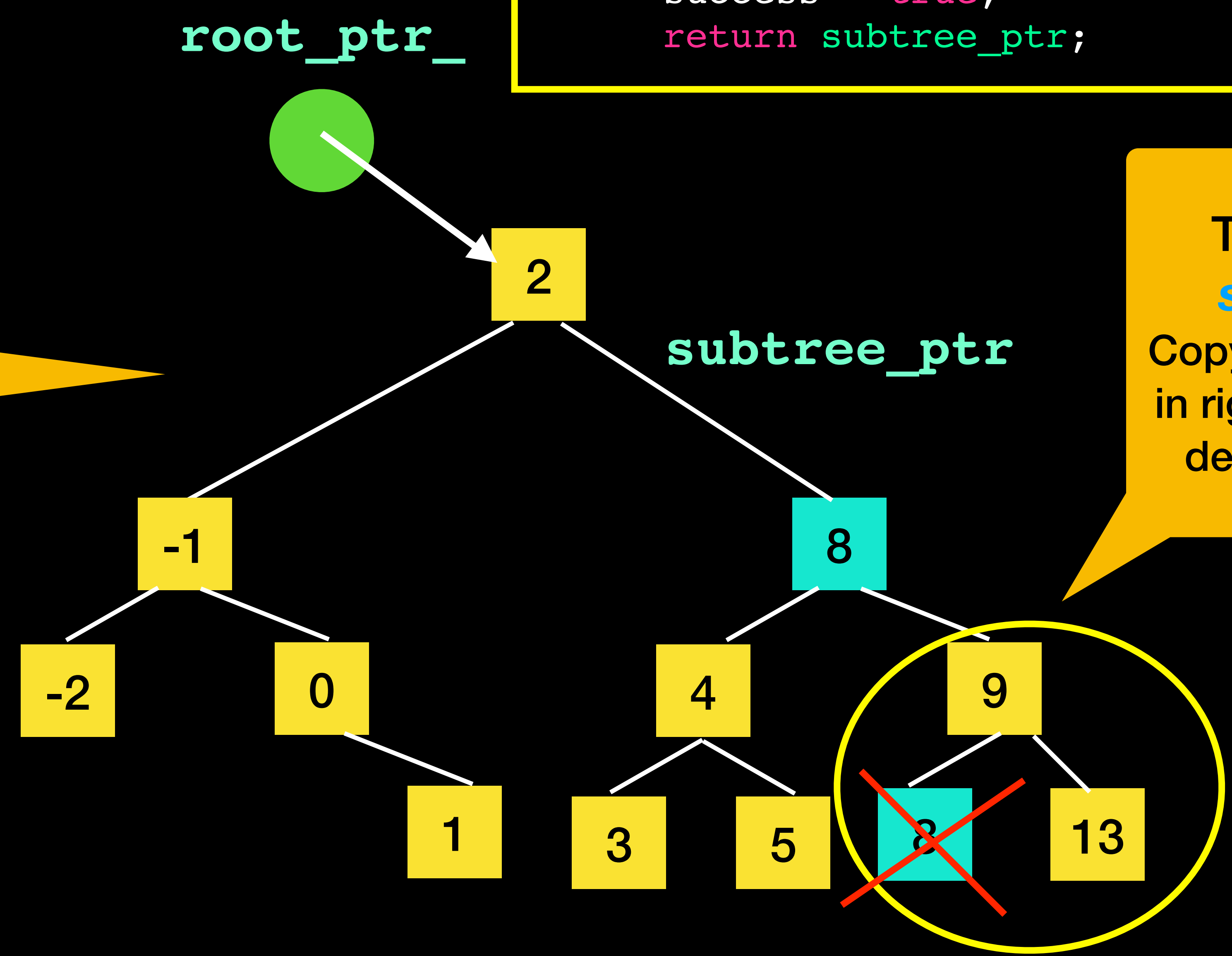
The *inorder successor:* Copy smallest value in right subtree and delete that node

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



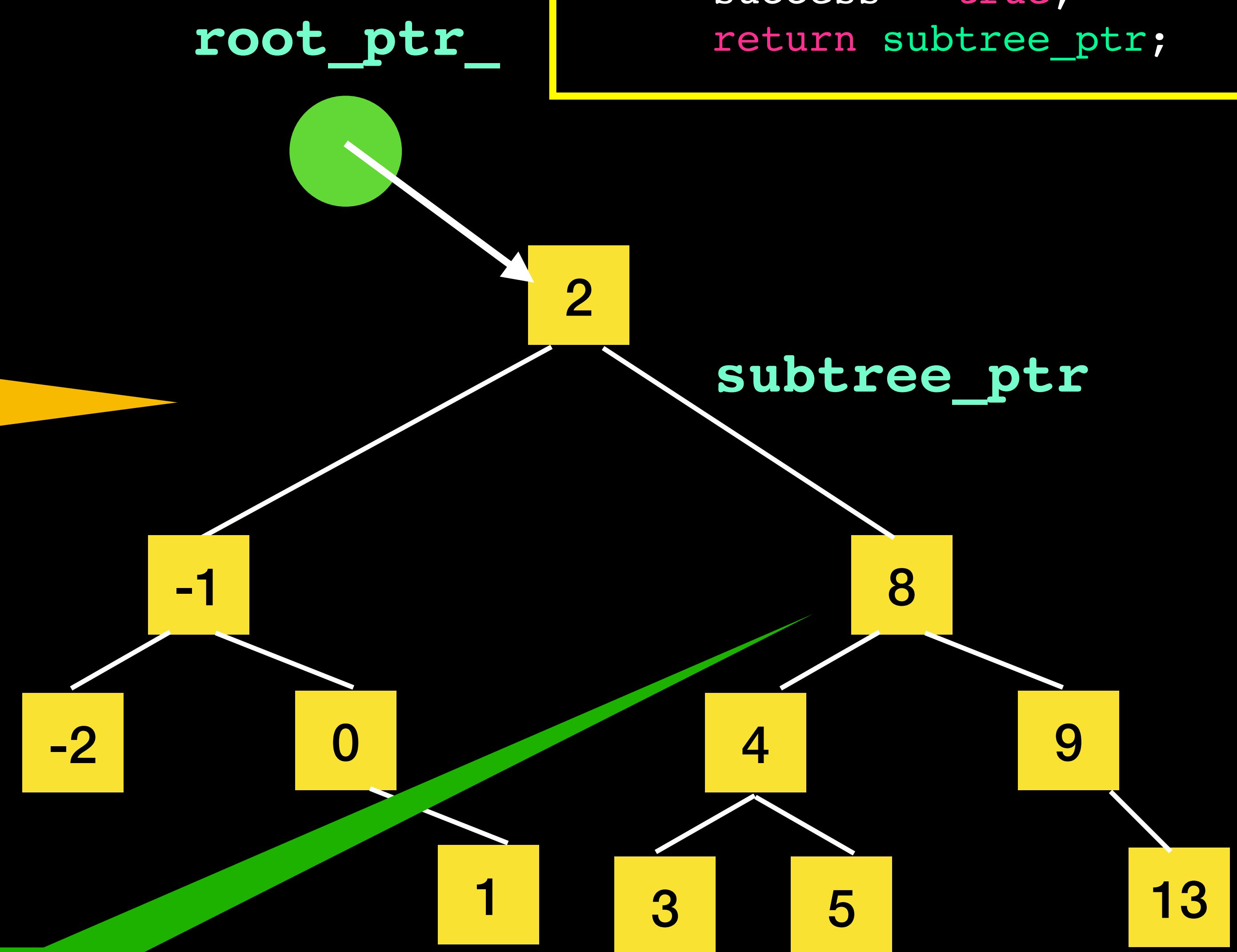
The *inorder successor:* Copy smallest value in right subtree and delete that node

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



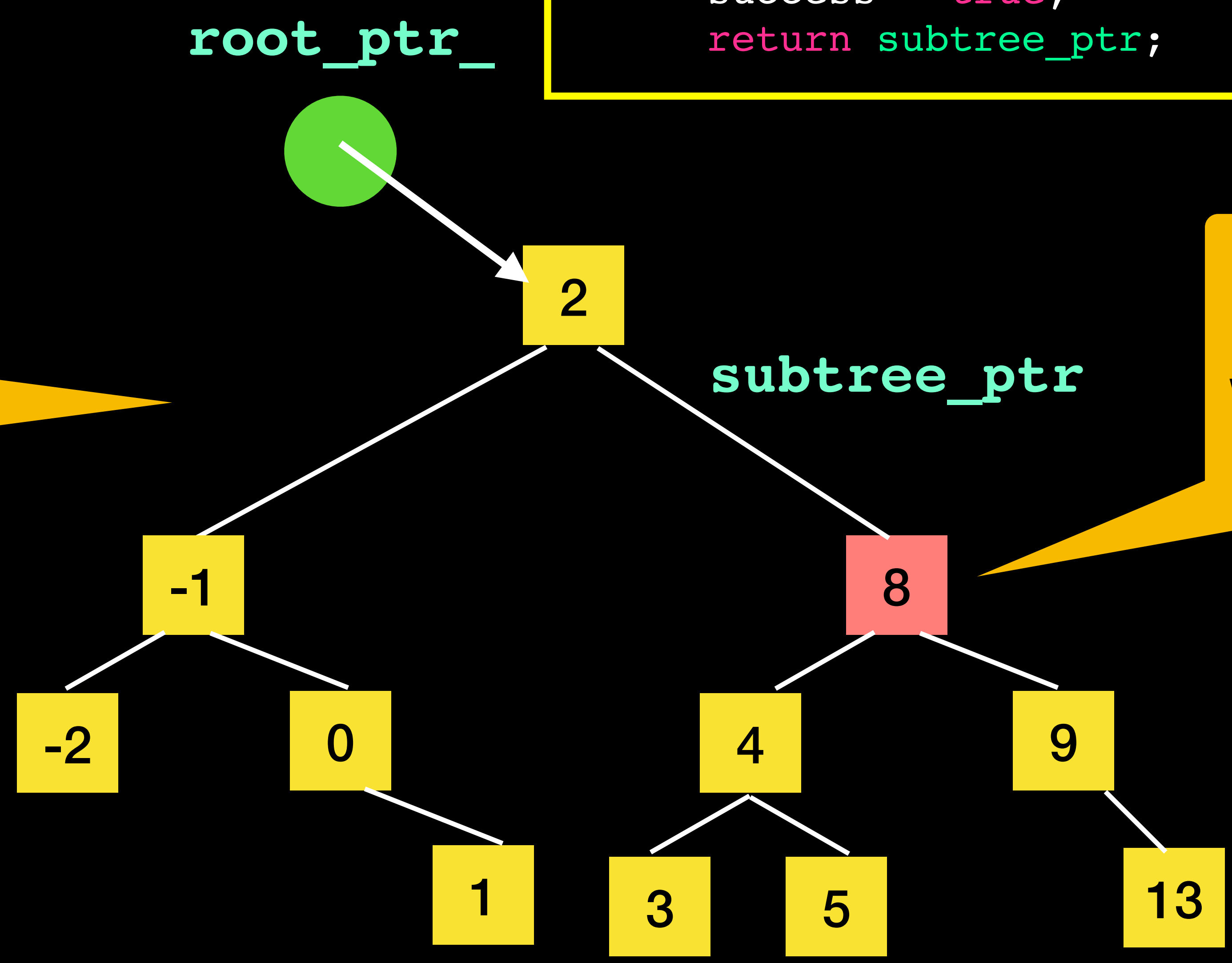
This operation will actually "reorganize" the tree

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
    {
        // Item is in the root of this subtree
        subtree_ptr = removeNode(subtree_ptr);
        success = true;
        return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



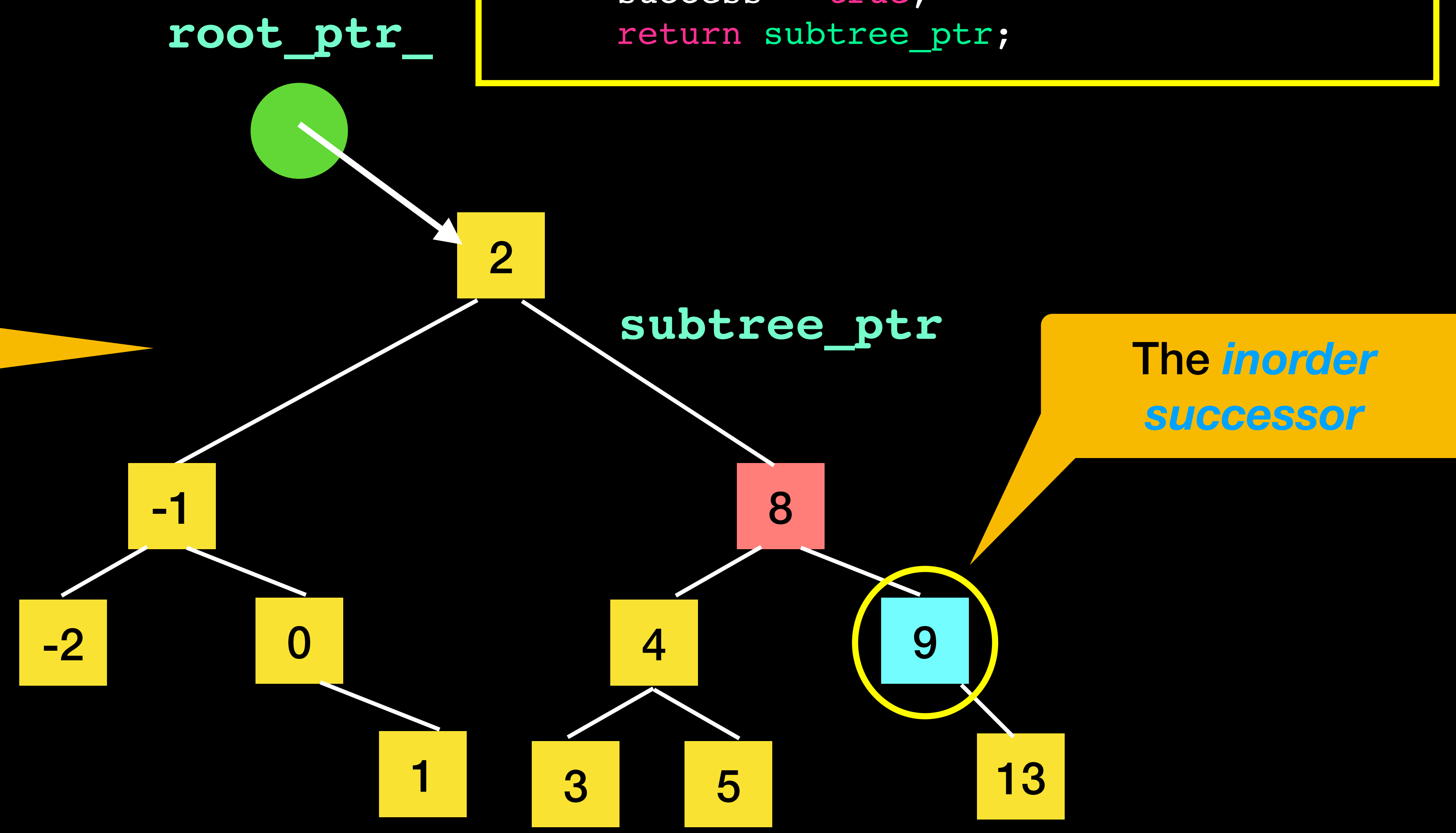
What about removing 8 now? What value should we put here?

removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

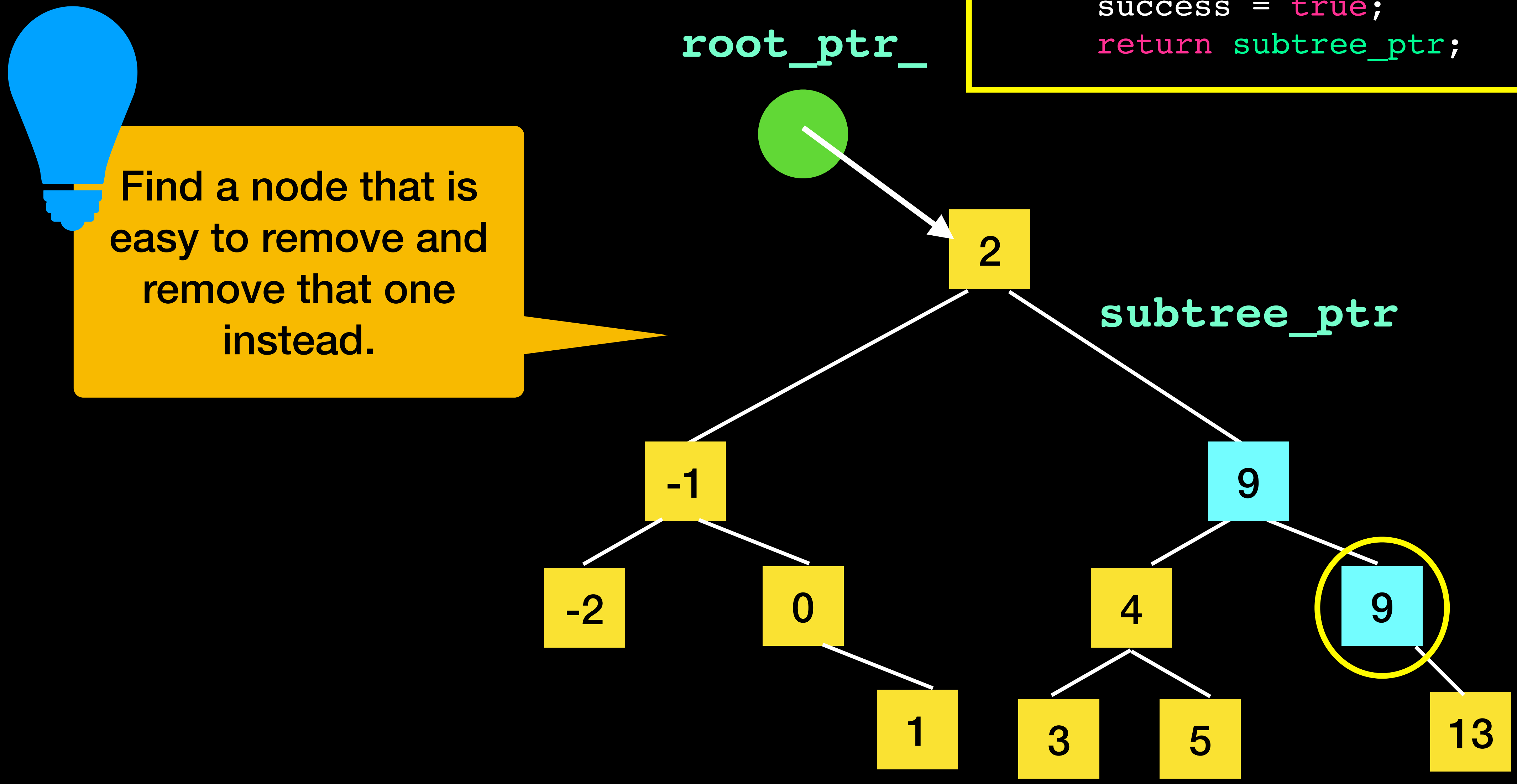
Find a node that is easy to remove and remove that one instead.



removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

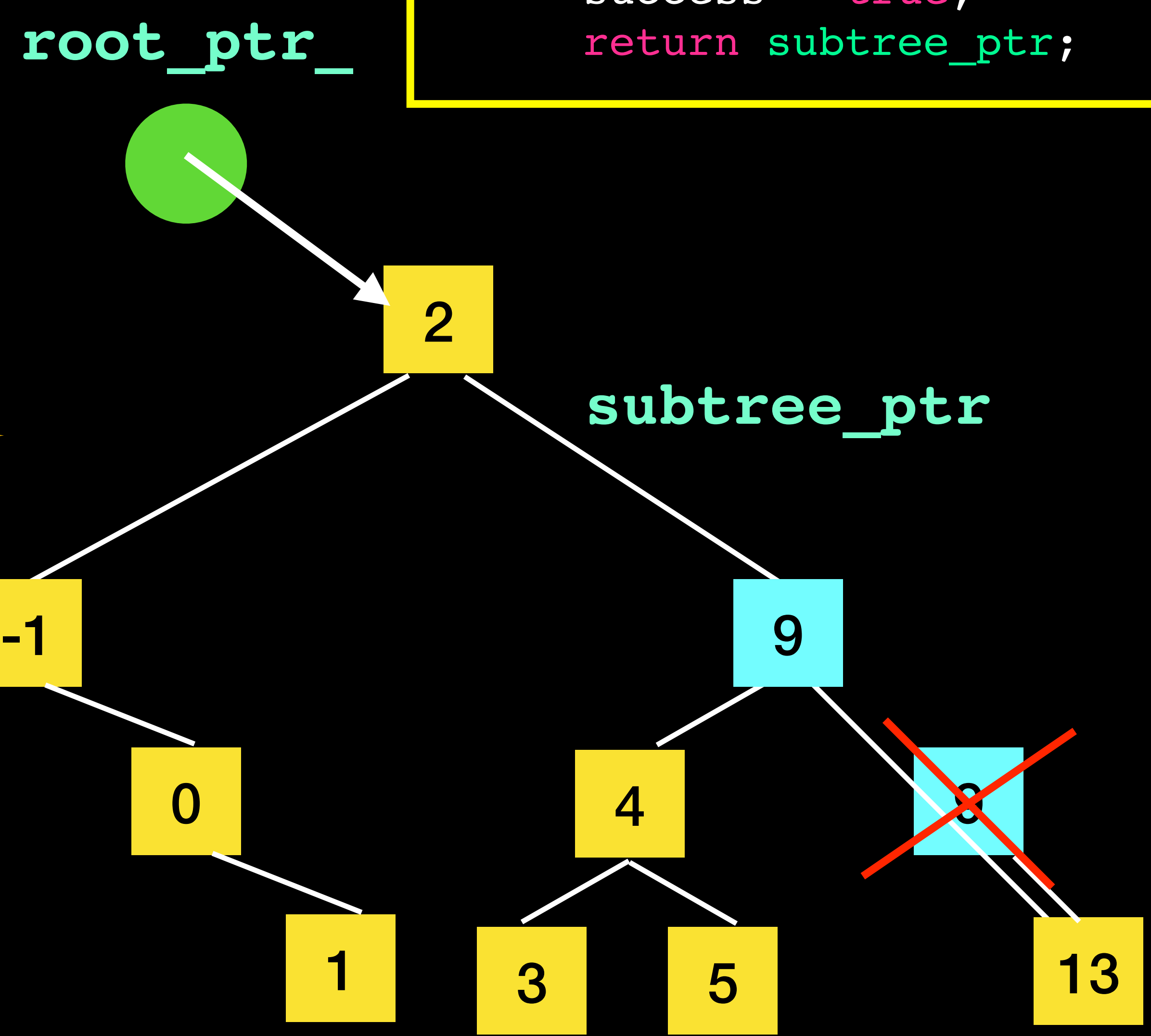


removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.

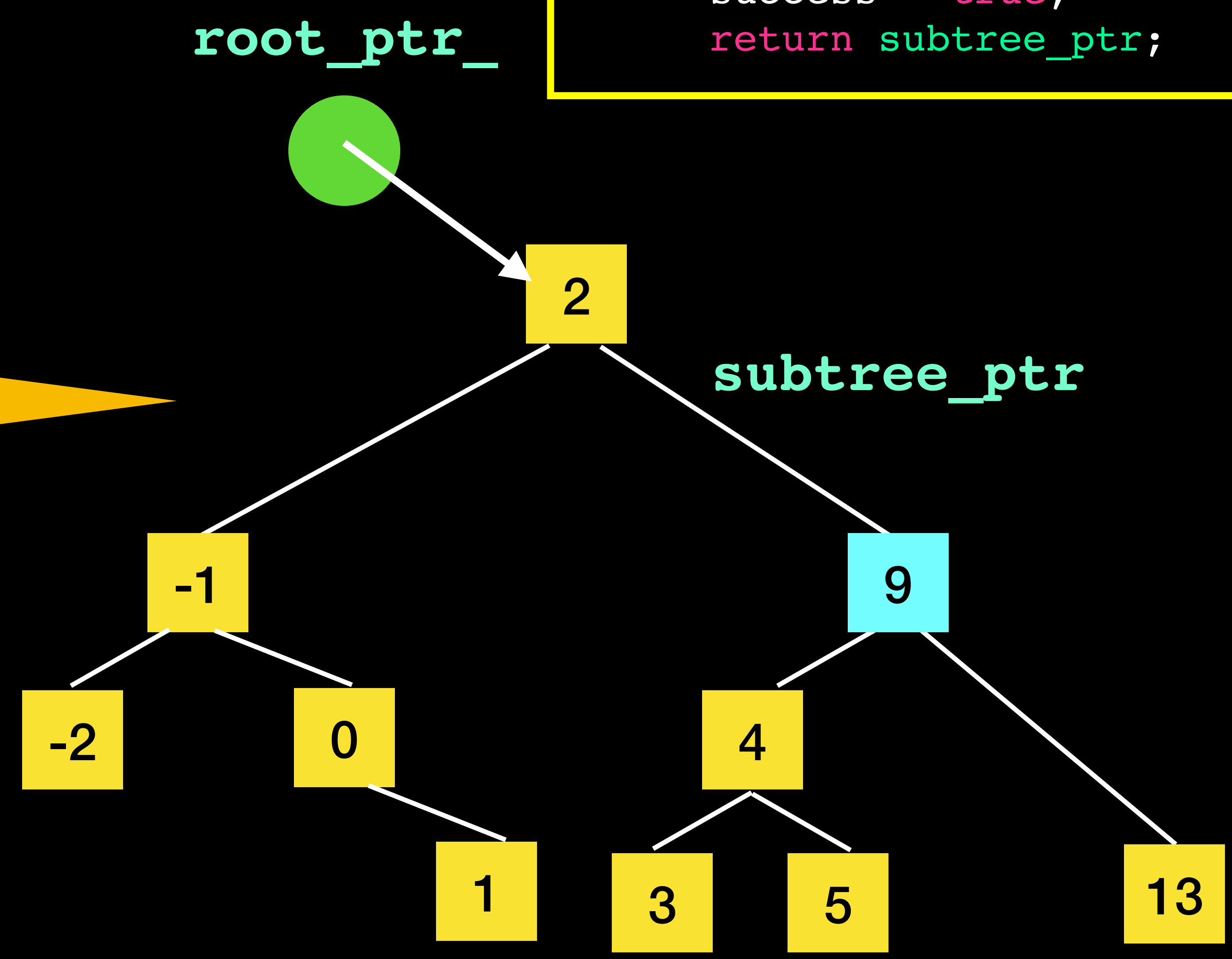


removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
   {
      // Item is in the root of this subtree
      subtree_ptr = removeNode(subtree_ptr);
      success = true;
      return subtree_ptr;
```

Case 3: target has 2 children

Find a node that is easy to remove and remove that one instead.



removeNode(node_ptr);

```
template<class T>
auto BST<T>::removeNode(std::shared_ptr<BinaryNode<T>> node_ptr)
{
   // Case 1) Node is a leaf - it is deleted
   if (node_ptr->isLeaf())
   {
      node_ptr.reset();
      return node_ptr; // delete and return nullptr
   }
   // Case 2) Node has one child - parent adopts child
   else if (node_ptr->getLeftChildPtr() == nullptr)  // Has rightChild only
   {
      return node_ptr->getRightChildPtr();
   }
   else if (node_ptr->getRightChildPtr() == nullptr) // Has left child only
   {
      return node_ptr->getLeftChildPtr();
   }
   // Case 3) Node has two children:
   else
   {
      T new_node_value;
      node_ptr->setRightChildPtr(removeLeftmostNode(node_ptr->getRightChildPtr(),
                                                              new_node_value));

      node_ptr->setItem(new_node_value);
      return node_ptr;

   }  // end if
}  // end removeNode
```

Node is leaf

Node has 1 child

Node has 2 children

Will find leftmost leaf in right subtree, save value in `new_node_value` and delete

Safe Programming:
reference parameter is local to the private calling function

removeLeftmostNode

```
template<class T>
auto BST<T>::removeLeftmostNode(std::shared_ptr<BinaryNode<T>>
                                              nodePtr, T& inorderSuccessor)
{
   if (nodePtr->getLeftChildPtr() == nullptr)
   {
      inorderSuccessor = nodePtr->getItem();
      return removeNode(nodePtr);
   }
   else
   {
      nodePtr->setLeftChildPtr(removeLeftmostNode(nodePtr->getLeftChildPtr(),
                                                    inorderSuccessor));
      return nodePtr;
   }  // end if
}  // end removeLeftmostNode
```

Traversals

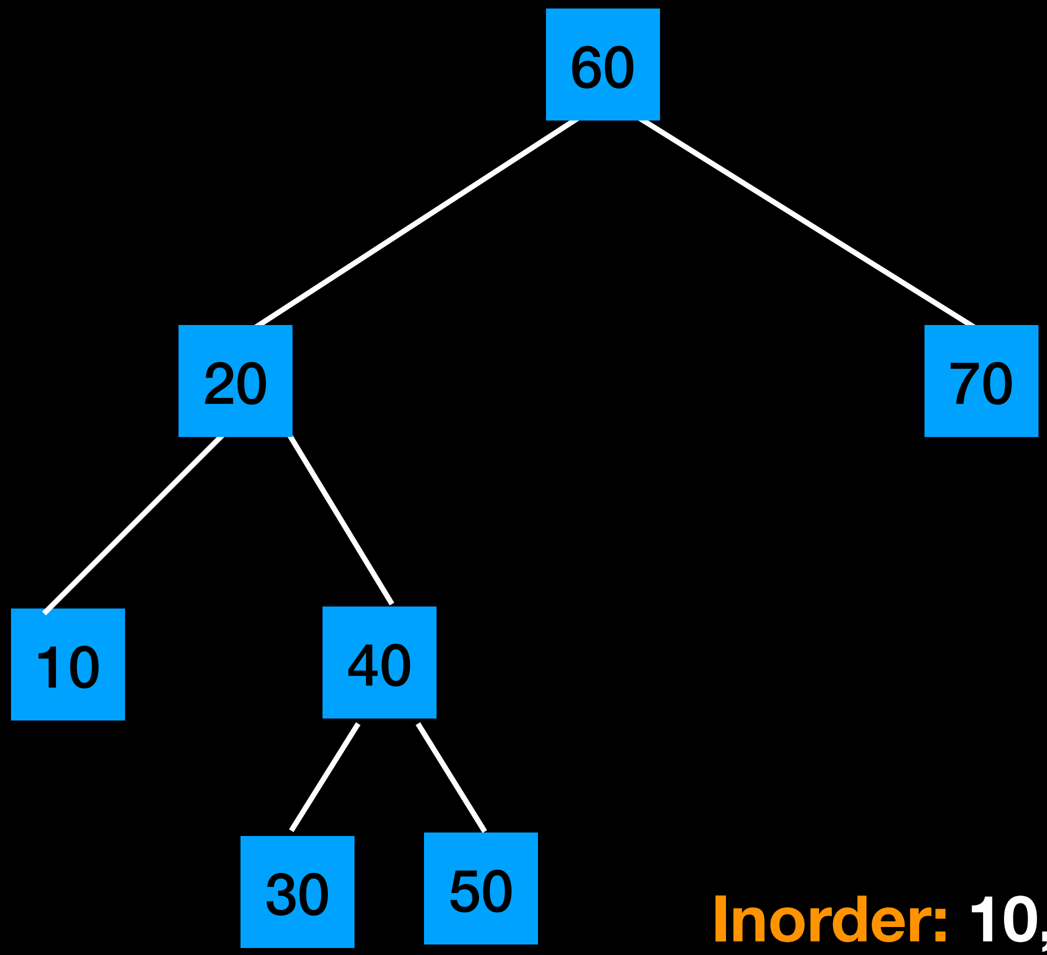
Let’s focus on the traversal for now, we will find out what Visitor does next

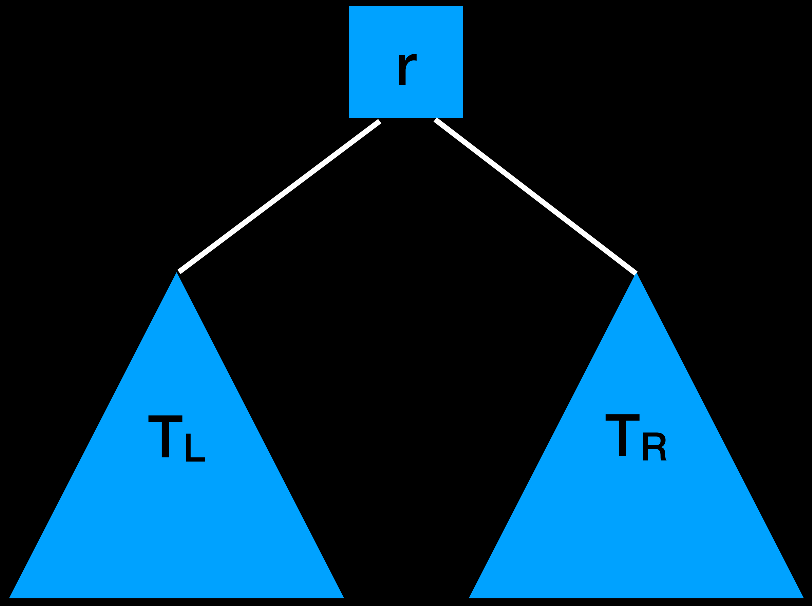
```
template<class T>
void BST<T>::preorderTraverse(Visitor<T>& visit) const
{
   preorder(visit, root_ptr_);
}  // end preorderTraverse

template<class T>
void BST<T>::inorderTraverse(Visitor<T>& visit) const
{
   inorder(visit, root_ptr_);
}  // end inorderTraverse

template<class T>
void BST<T>::postorderTraverse(Visitor<T>& visit) const
{
   postorder(visit, root_ptr_);
}  // end postorderTraverse
```

Visit (retrieve, print, modify …) every node in the tree

Inorder Traversal:

```
if (T is not empty) //implicit base case
{
    traverse T_L
    visit the root r
    traverse T_R
}
```





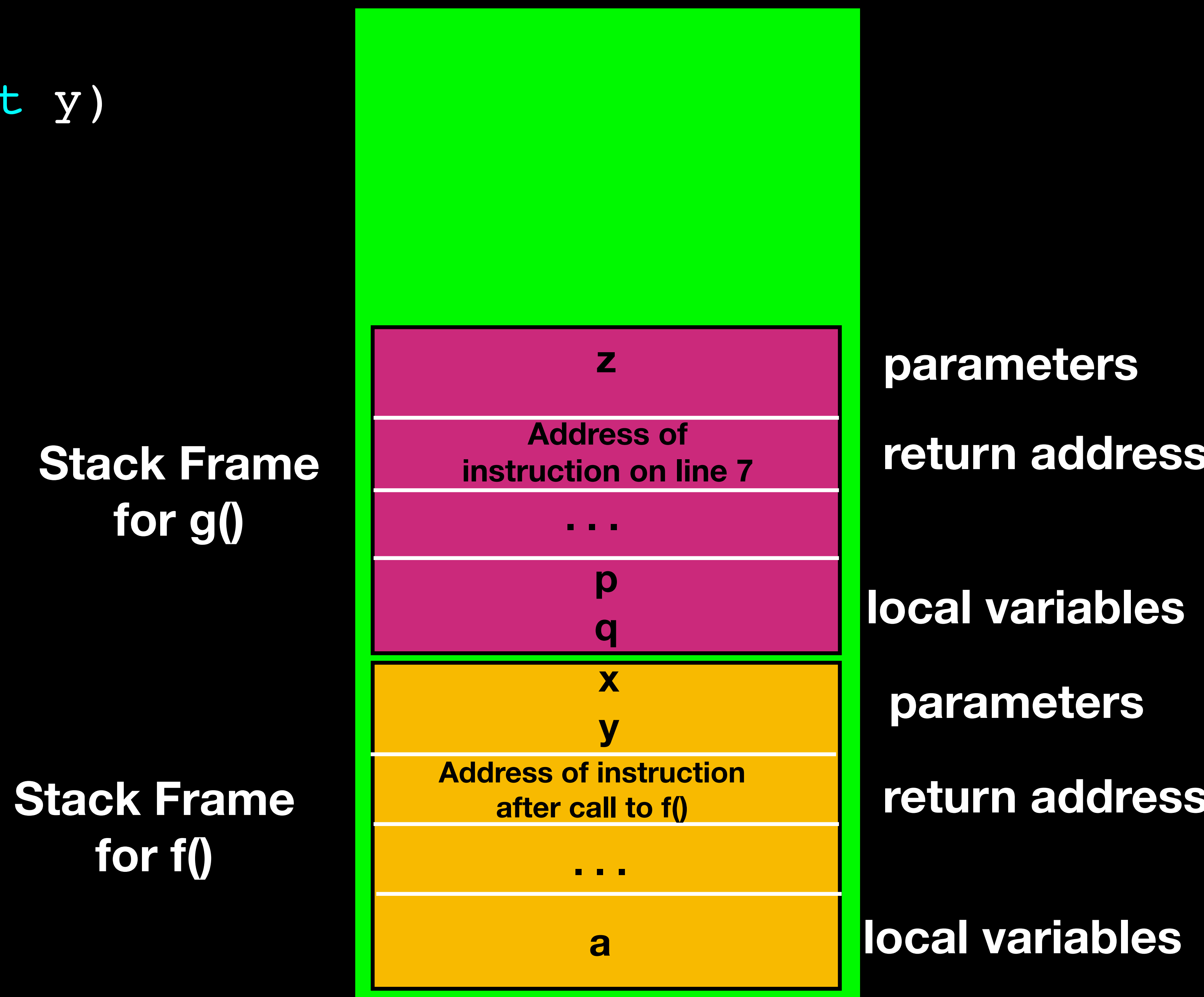
Inorder: 10, 20, 30, 40, 50, 60, 70

inorderTraverse Helper Function

```
template<class T>
void BST<T>::inorder(Visitor<T>& visit,
   std::shared_ptr<BinaryNode<T>> tree_ptr) const
{
    if (tree_ptr != nullptr)
    {
        inorder(visit, tree_ptr->getLeftChildPtr());
        T the_item = tree_ptr->getItem();
        visit(the_item);
        inorder(visit, tree_ptr->getRightChildPtr());
    }  // end if
}  // end inorder
```

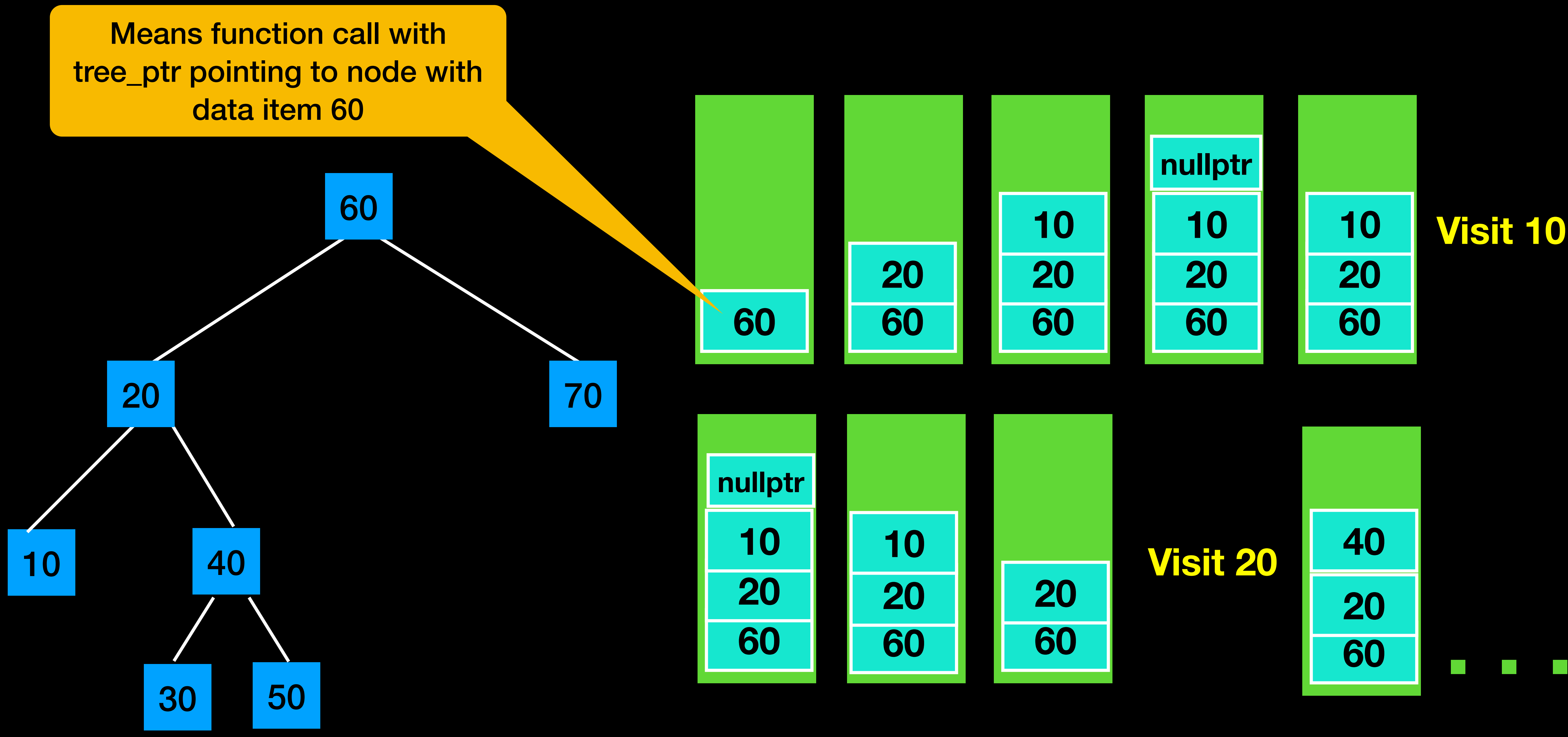
Recall: Program Stack

```
1   void f(int x, int y)
2   {
3     int a;
4     // stuff here
5     if(a<13)
6        a = g(a);
7     // stuff here
8   }

9   int g(int z)
10  {
11    int p ,q;
12    // stuff here
13    return q;
14  }
```



Recursive Traversal

In recursive solution program stack keeps track of what node must be visited next



Recursive Traversal

With recursion:

- program stack implicitly finds node traversal must visit next
- If traversal backs up to node *d* from right subtree it backs up further to *d*'s parent as a consequence of the recursive program execution

Non-recursive Traversal

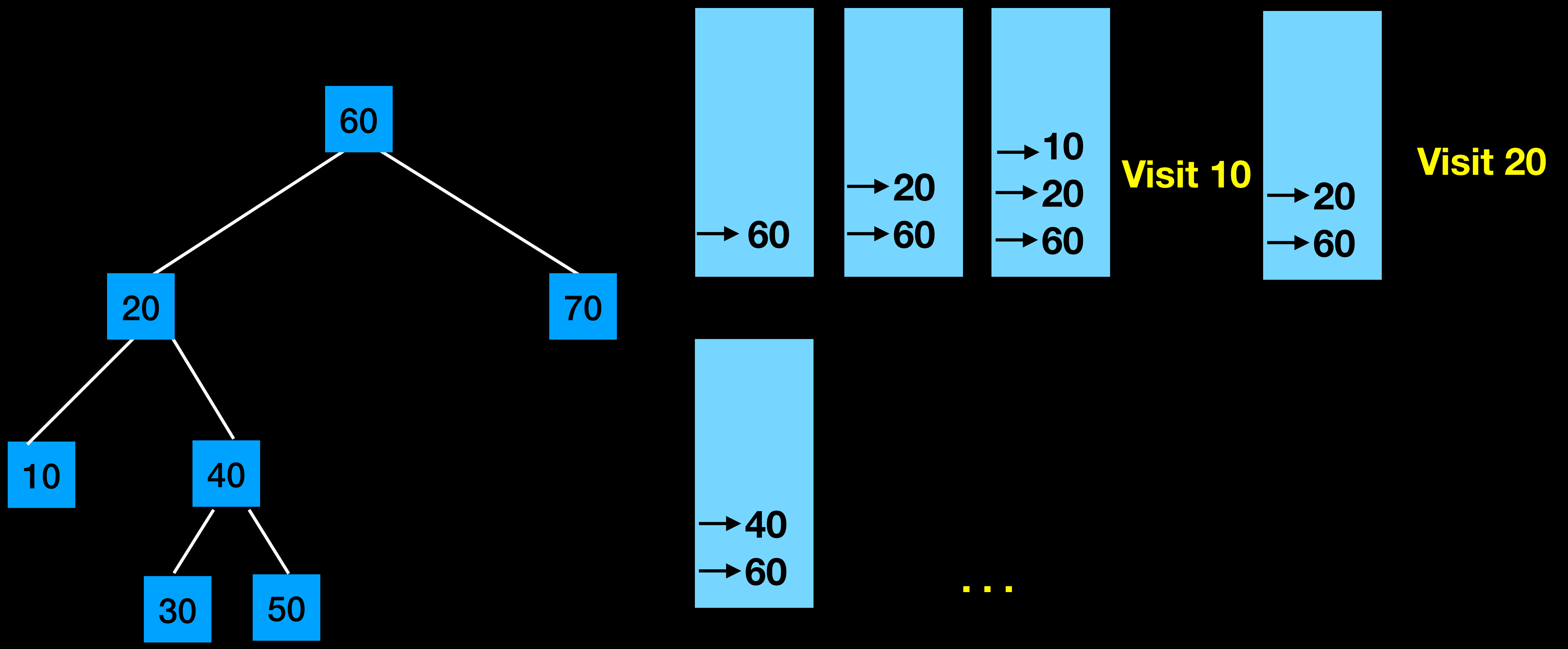
Optimize: Implement iterative approach that maintains an explicit stack to keep track of nodes that must be visited

Place pointer to node on stack **only** before traversing it's left subtree but **NOT** before traversing right subtree

This will also save some "steps" that were unnecessary but implicit in recursive implementation

Non-recursive Traversal

Iterative solution explicitly maintains a stack of pointers to nodes to keep track of what node must be visited next



Non-recursive Traversal

```
template<class T>
void BST<T>::inorder(Visitor<T>& visit) const
{
    std::stack<T> node_stack;
    std::shared_ptr<BinaryNode<T>> current_ptr = root_ptr_;
    bool done = false;

    while(!done)
    {
        if(current_ptr != nullptr)
        {
            node_stack.push(current_ptr);

            //traverse left subtree
            current_ptr = current_ptr->getLeftChildPtr();
        }
```

Non-recursive Traversal cont.

```
        //backtrack from empt subtree and visit the node at top of
        //stack, but if stack is empty traversal is completed
        else{
            done = node_stack.isEmpty();
            if(!done)
            {
                current_ptr = node_stack.top();
                visit(current_ptr->getItem());
                node_stack.pop();

                //traverse right subtree of node just visited
                current_ptr = current_ptr->getRightChildPtr();
            }
        }
    }
}  // end inorder
```

Traversals

The last cool trick for you this semester

Looking for different behavior

```
template<class T>
void BST<T>::preorderTraverse(Visitor<T>& visit) const
{
   preorder(visit, root_ptr_);
}  // end preorderTraverse

template<class T>
void BST<T>::inorderTraverse(Visitor<T>& visit) const
{
   inorder(visit, root_ptr_);
}  // end inorderTraverse

template<class T>
void BST<T>::postorderTraverse(Visitor<T>& visit) const
{
   postorder(visit, root_ptr_);
}  // end postorderTraverse
```

Functors

Objects that by overloading `operator()` can be "called" like a function

POLYMORPHISM!
ABSTRACT CLASS!!!

```
#ifndef Visitor_hpp
#define Visitor_hpp
#include <string>

template<class T>
class Visitor
{
public:

    virtual void operator()(T&) = 0;
    virtual void operator()(T&, T&) = 0;

};

#endif /* Visitor_hpp */
```

```cpp
#ifndef StringPrinter_hpp
#define StringPrinter_hpp

#include "Visitor.hpp"
#include <iostream>
#include <string>


class StringPrinter: public Visitor<std::string>
{
public:

    void operator()(std::string&) override;
    void operator()(std::string&, std::string&) override;

};


#endif /* StringPrinter_hpp */
```

```cpp
#include "StringPrinter.hpp"

void StringPrinter::operator()(std::string& x)
{
    std::cout << x << std::endl;
}

void StringPrinter::operator()(std::string& a, std::string& b)
{
    std::cout << a << b << std::endl;
}
```

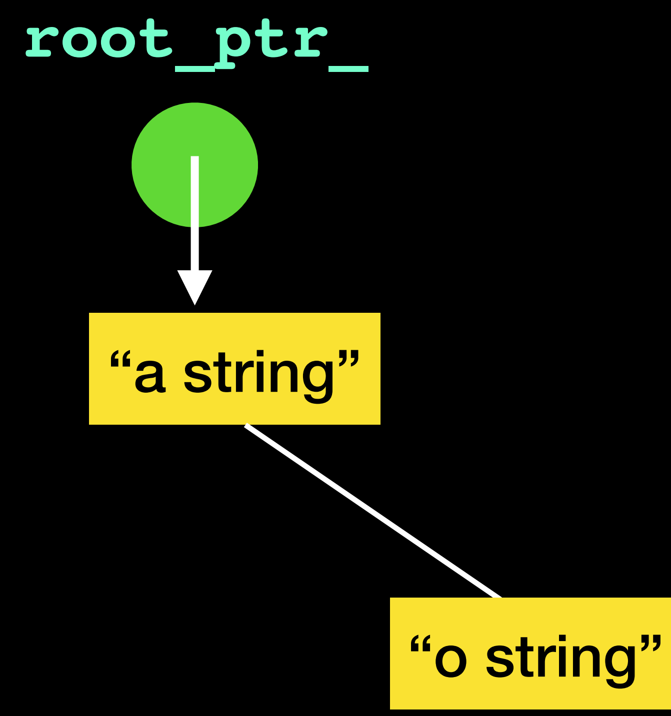
```cpp
#ifndef Inverter_hpp
#define Inverter_hpp

#include "Visitor.hpp"
#include <iostream>
#include <string>
#include <algorithm>


class Inverter: public Visitor<std::string>
{
public:

    void operator()(std::string&) override;
    void operator()(std::string&, std::string&) override;

};

#endif /* Inverter_hpp */
```

```cpp
#include "Inverter.hpp"

void Inverter::operator()(std::string& x)
{
    std::reverse(x.begin(), x.end());
    std::cout << x << std::endl;

}

void Inverter::operator()(std::string& a, std::string& b)
{
    a.swap(b);
    std::cout << a << b << std::endl;
}
```

Traversal with Functor parameter

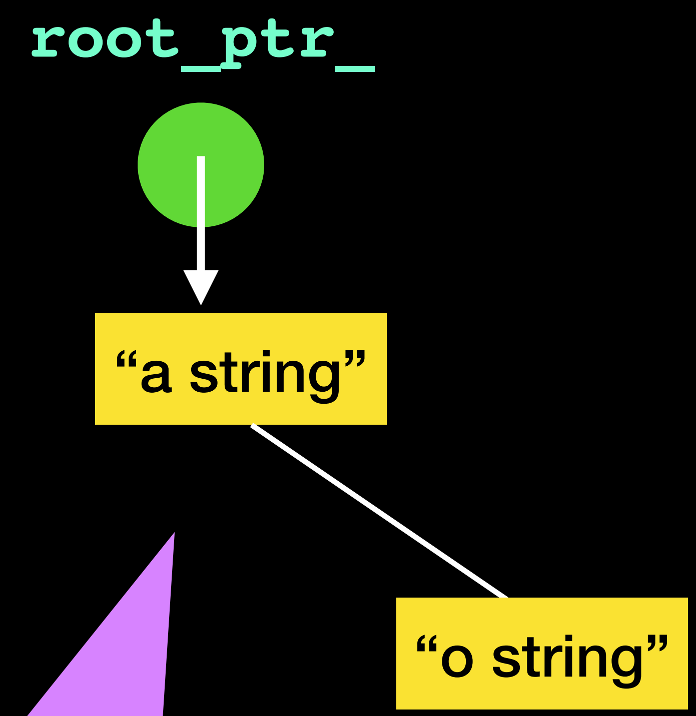
```
template<class T>
void BST<T>::inorder(Visitor<T>& visit,
   std::shared_ptr<BinaryNode<T>> tree_ptr) const
{
    if (tree_ptr != nullptr)
    {
        inorder(visit, tree_ptr->getLeftChildPtr());
        T the_item = tree_ptr->getItem();

        visit(the_item);

        inorder(visit, tree_ptr->getRightChildPtr());
    }  // end if
}  // end inorder
```

```cpp
int main() {

    std::string a_string = "a string";
    std::string anoter_string = "o string";

    BST<std::string> a_tree(a_string);
    a_tree.add(anoter_string);

    StringPrinter p;
    Inverter i;


    a_tree.inorderTraverse(p);
    std::cout << std::endl;
    a_tree.inorderTraverse(i);


    return 0;
}
```




```
a string
o string

gnirts a
gnirts o
Program ended with exit code: 0
```

```
int main() {

    std::string a_string = "a string";
    std::string anoter_string = "o string";

    BST<std::string> a_tree(a_string);
    a_tree.add(anoter_string);

    StringPrinter p;
    Inverter i;


    a_tree.inorderTraverse(p);
    std::cout << std::endl;
    a_tree.inorderTraverse(i);


    return 0;
}
```




```
a string
o string

gnirts a
gnirts o
Program ended with exit code: 0
```