

Tree Implementation

Tiziana Ligorio

Today's Plan



Recap

BST Implementation

```

#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



```

#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(shared_ptr<BinaryNode<ItemType>> left_ptr);
    void setRightChildPtr(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);
```

```

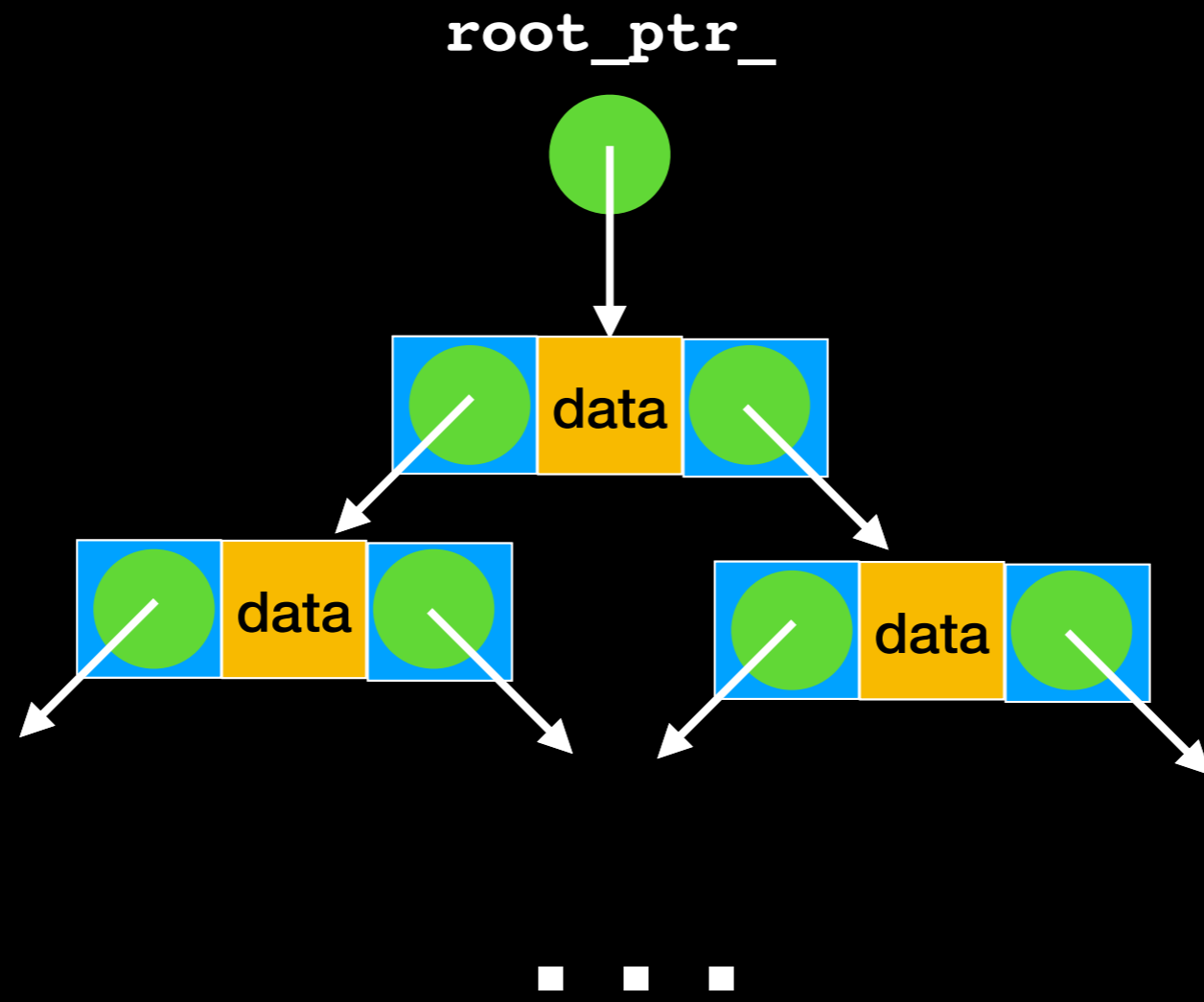
template<typename ItemType>
BinaryNode<ItemType>::BinaryNode(const ItemType& an_item)
                                : item_(an_item){ } //end constructor

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

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

```


BST



```

#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_

```

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

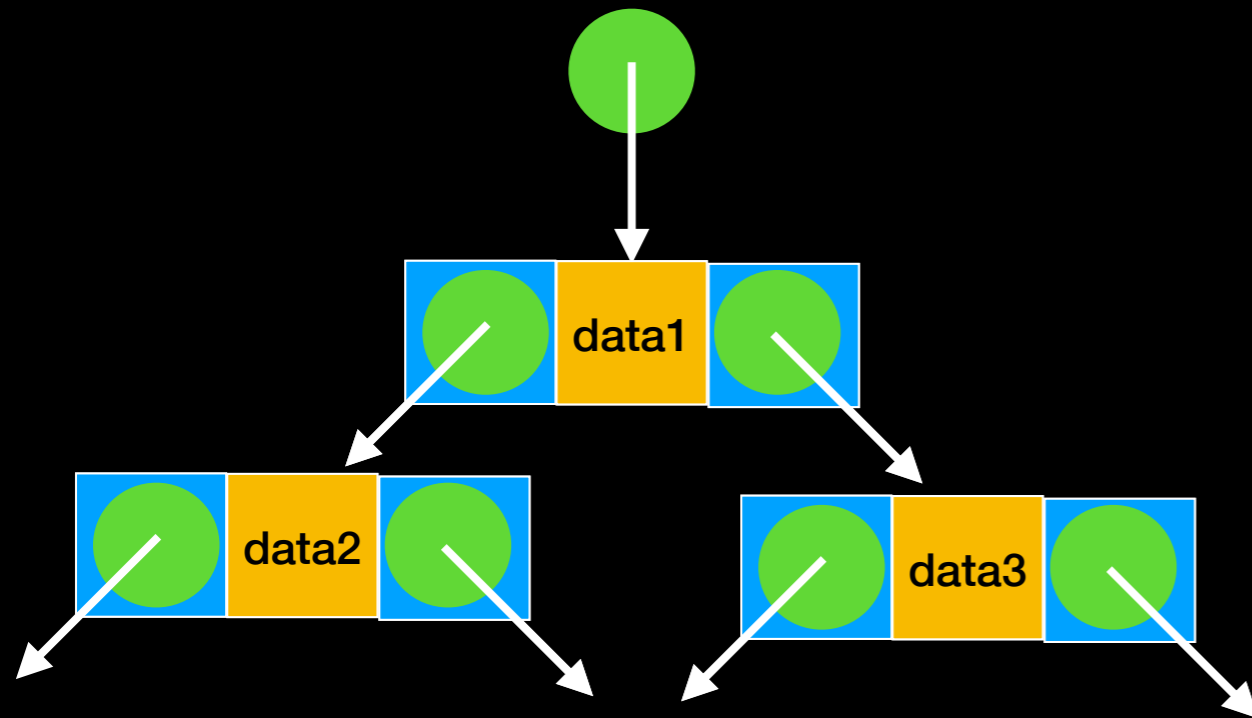
```
template<typename ItemType>
BST<ItemType>::BST(const BST<ItemType>& tree)
{
    root_ptr_ = copyTree(tree.root_ptr_); // Call helper function
} // end copy constructor
```

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)`

`old_tree_root_ptr`



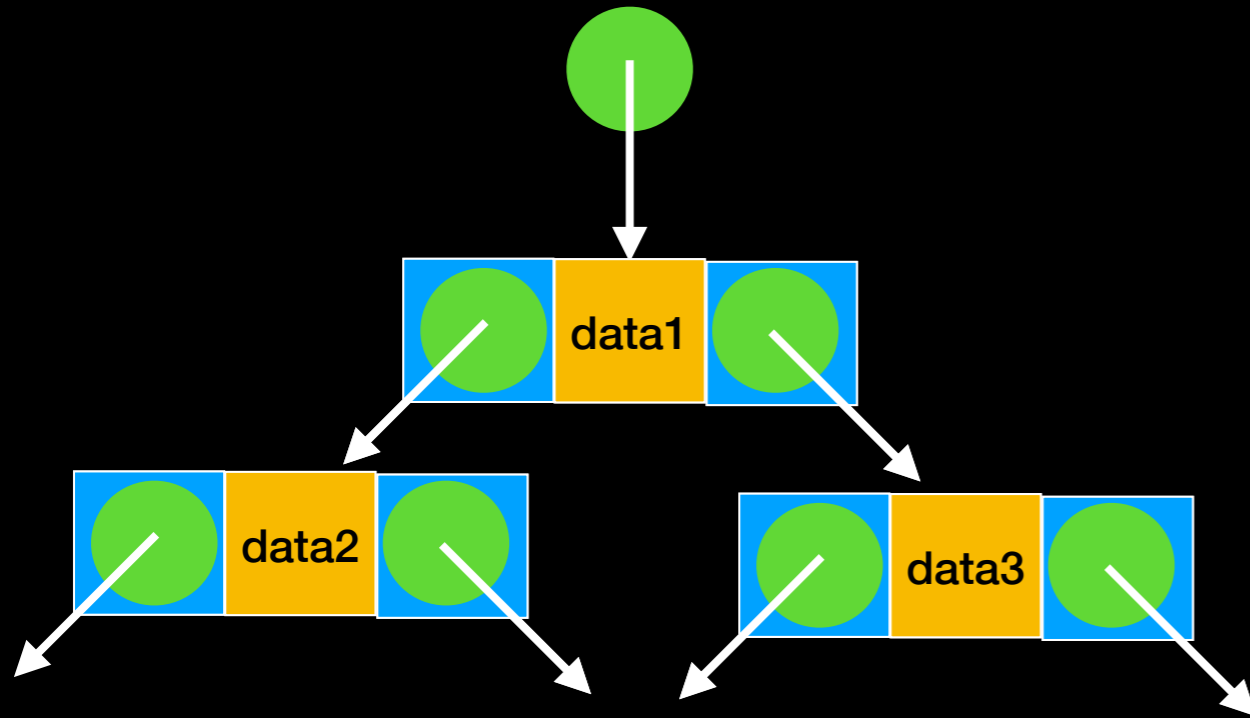
`new_tree_ptr`



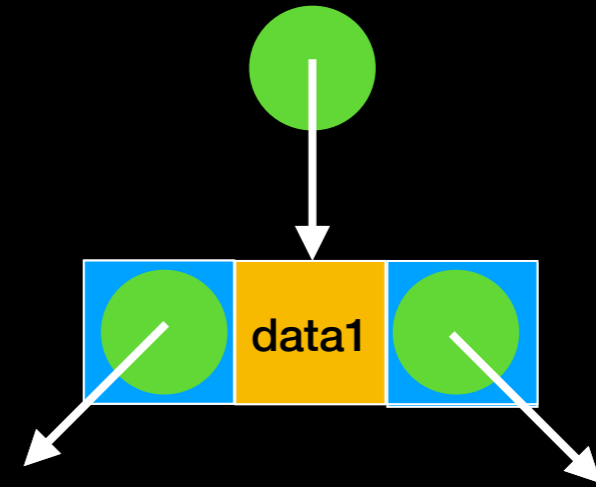
■ ■ ■

`copyTree(old_tree_root_ptr)`

`old_tree_root_ptr`

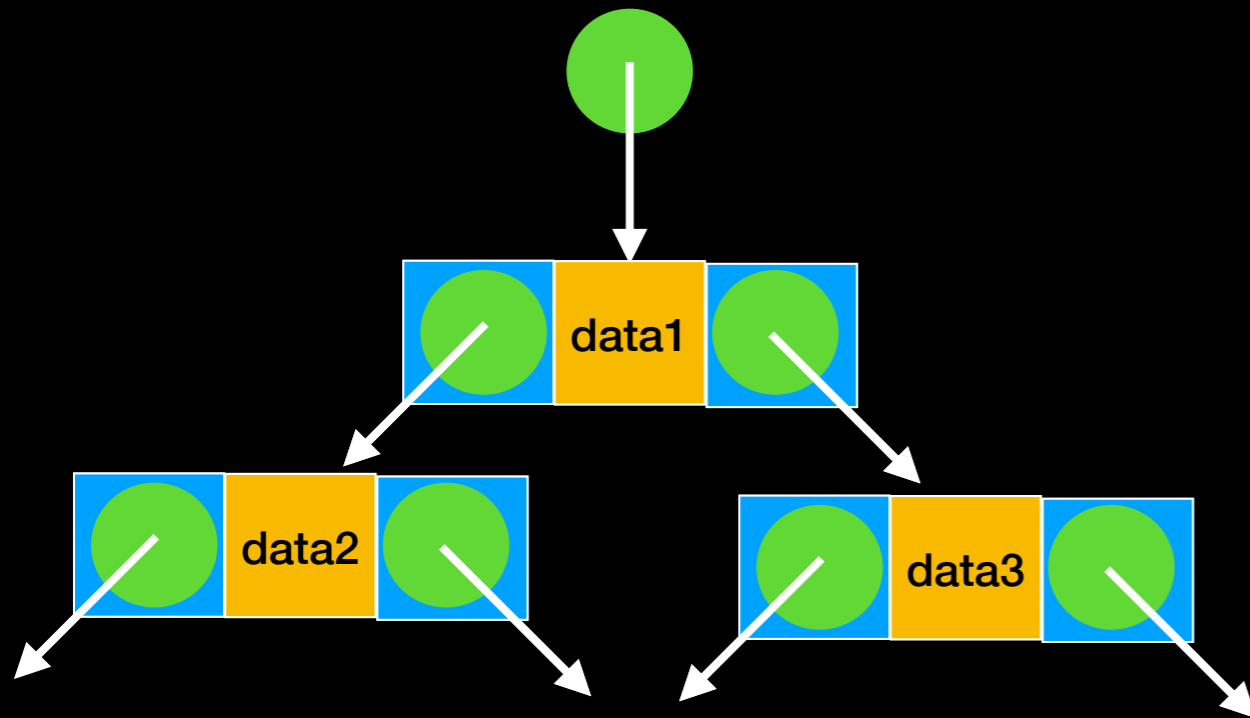


`new_tree_ptr`

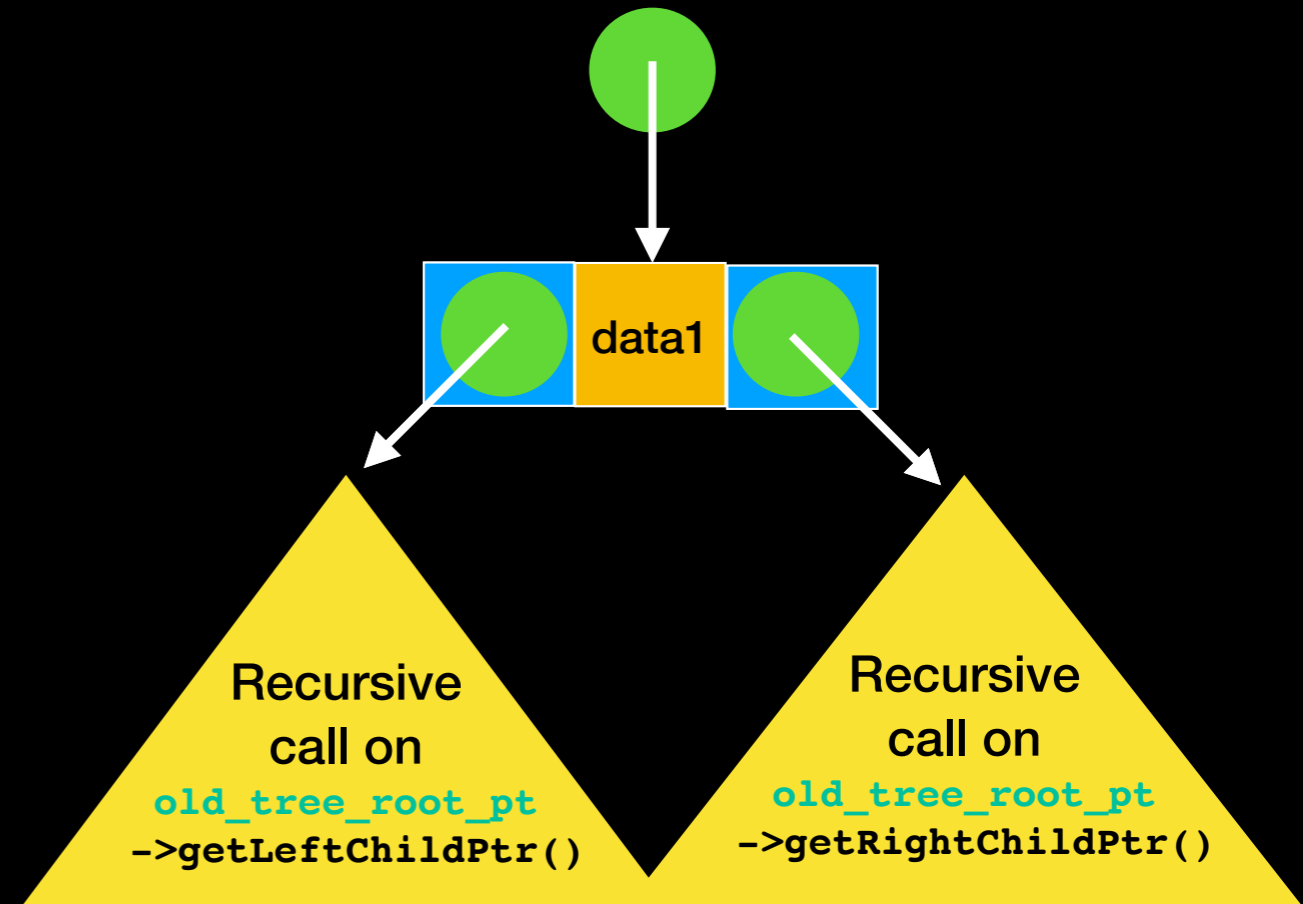


copyTree(old_tree_root_ptr)

old_tree_root_ptr



new_tree_ptr



Copy Constructor Helper Function

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

```
template<typename ItemType>
auto BST<ItemType>::copyTree(const
shared_ptr<BinaryNode<ItemType>> old_tree_root_ptr) const
{
    shared_ptr<BinaryNode<ItemType>> new_tree_ptr;
    // Copy tree nodes during a preorder traversal
    if (old_tree_root_ptr != nullptr)
    {
        // Copy node
        new_tree_ptr = make_shared<BinaryNode<ItemType>>
            (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

Recursive Calls:
Don't want to tie interface
to recursive
implementation:
Use helper function

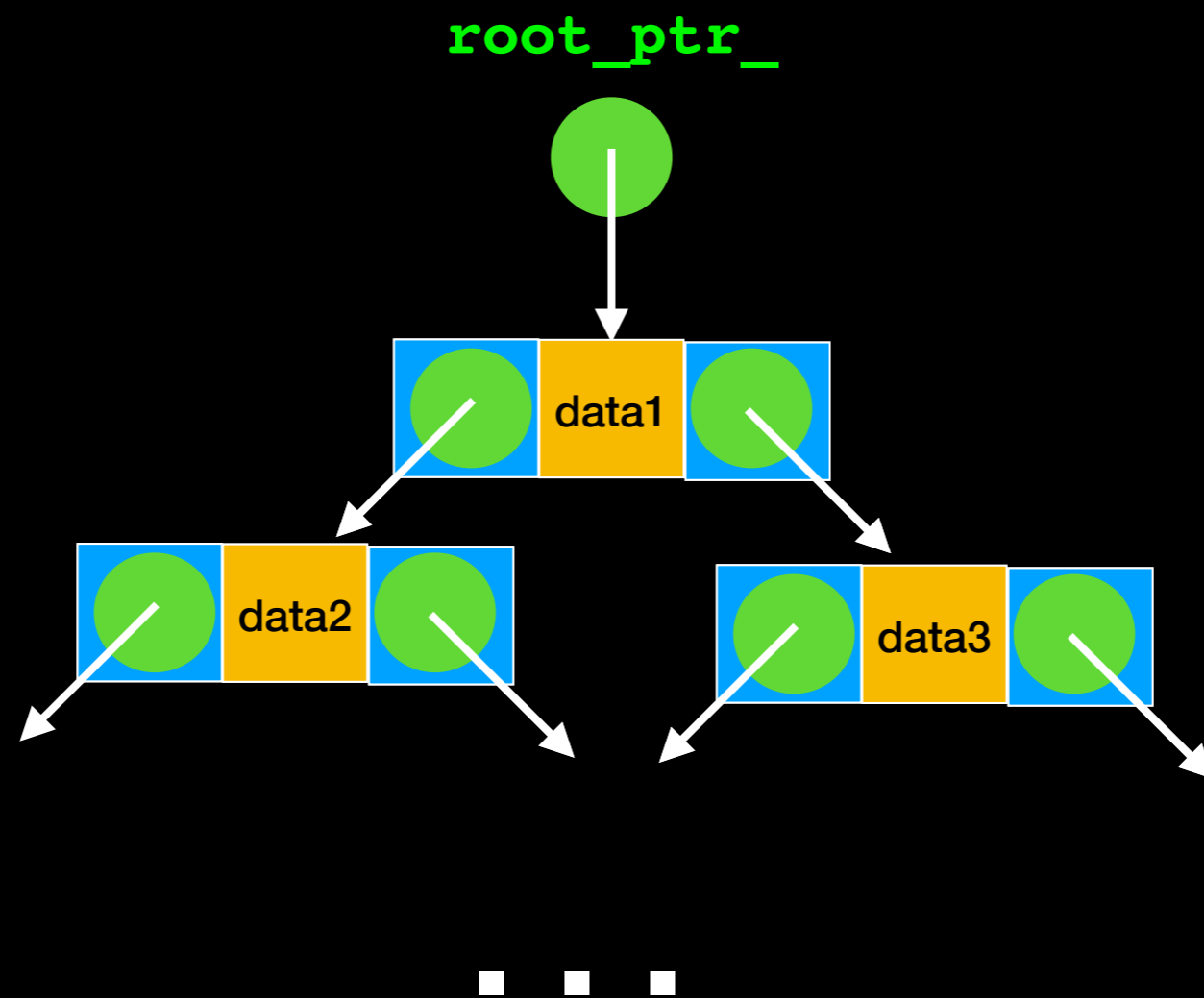
Preorder Traversal
Scheme: copy each node
as soon as it is visited to
make exact copy

Destructor

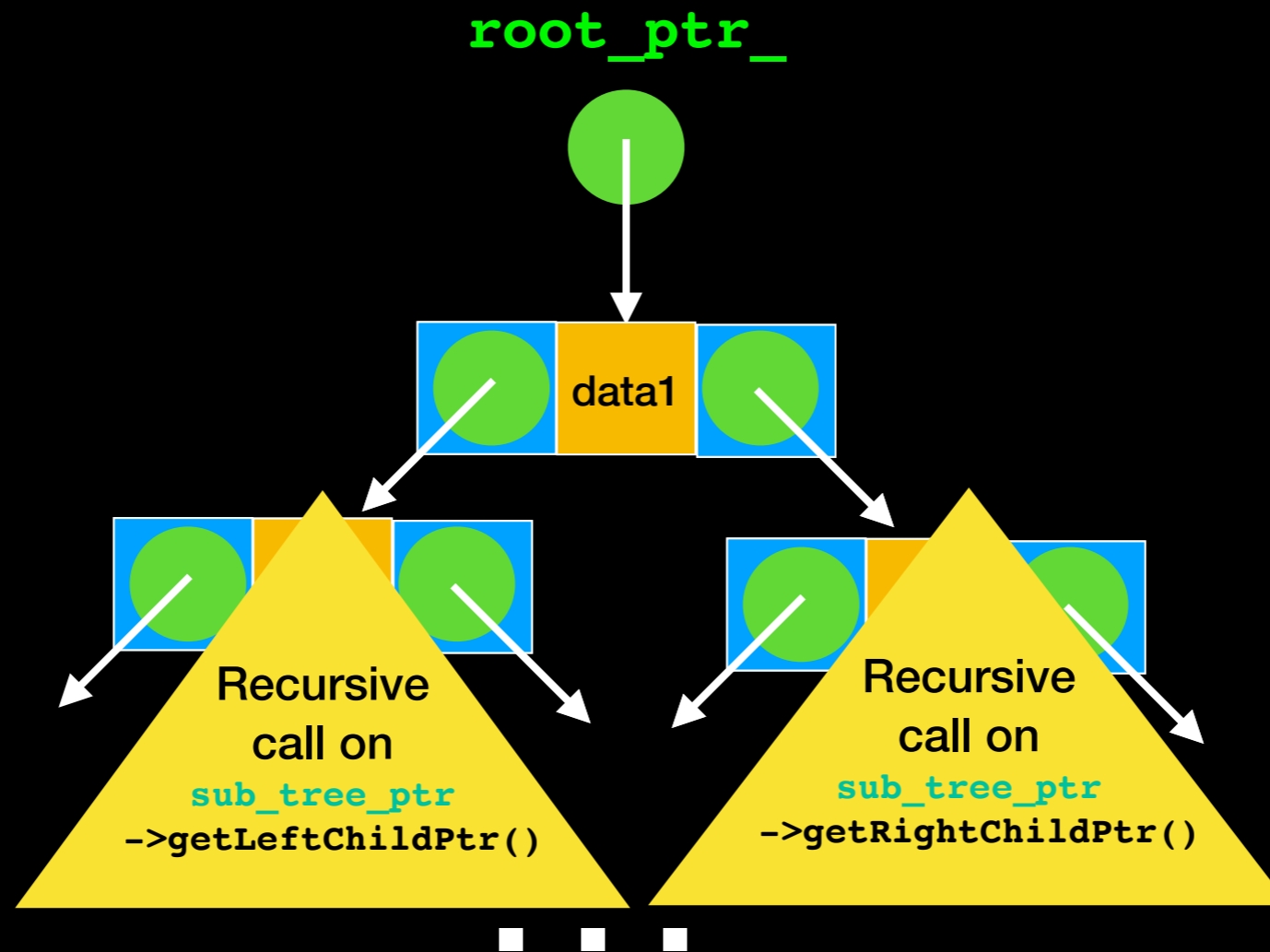
```
template<typename ItemType>
BST<ItemType>::~~BST()
{
    destroyTree(root_ptr_); // Call helper function
} // end destructor
```

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

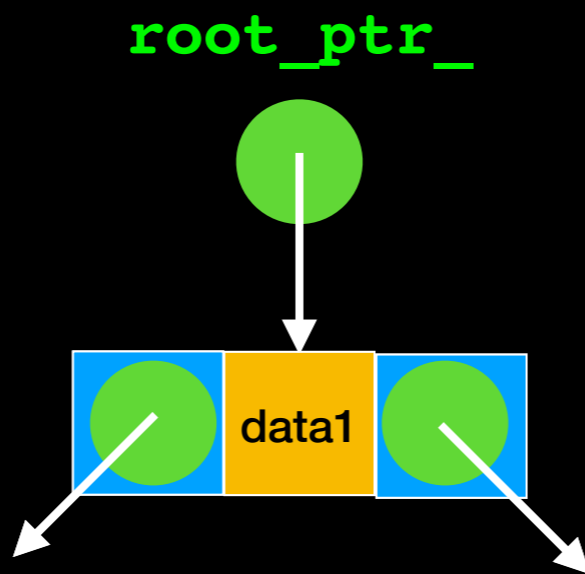
destroyTree(sub_tree_ptr)



destroyTree(sub_tree_ptr)



```
destroyTree(sub_tree_ptr)
```



```
root_ptr_.reset()
```

```
destroyTree(sub_tree_ptr)
```

root_ptr_



Destructor Helper Function

```
template<typename ItemType>
void BST<ItemType>::destroyTree(
    std::shared_ptr<BinaryNode<ItemType>> 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<typename ItemType>
void BST<ItemType>::clear()
{
    destroyTree(root_ptr_); // Call helper method
} // end clear
```

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

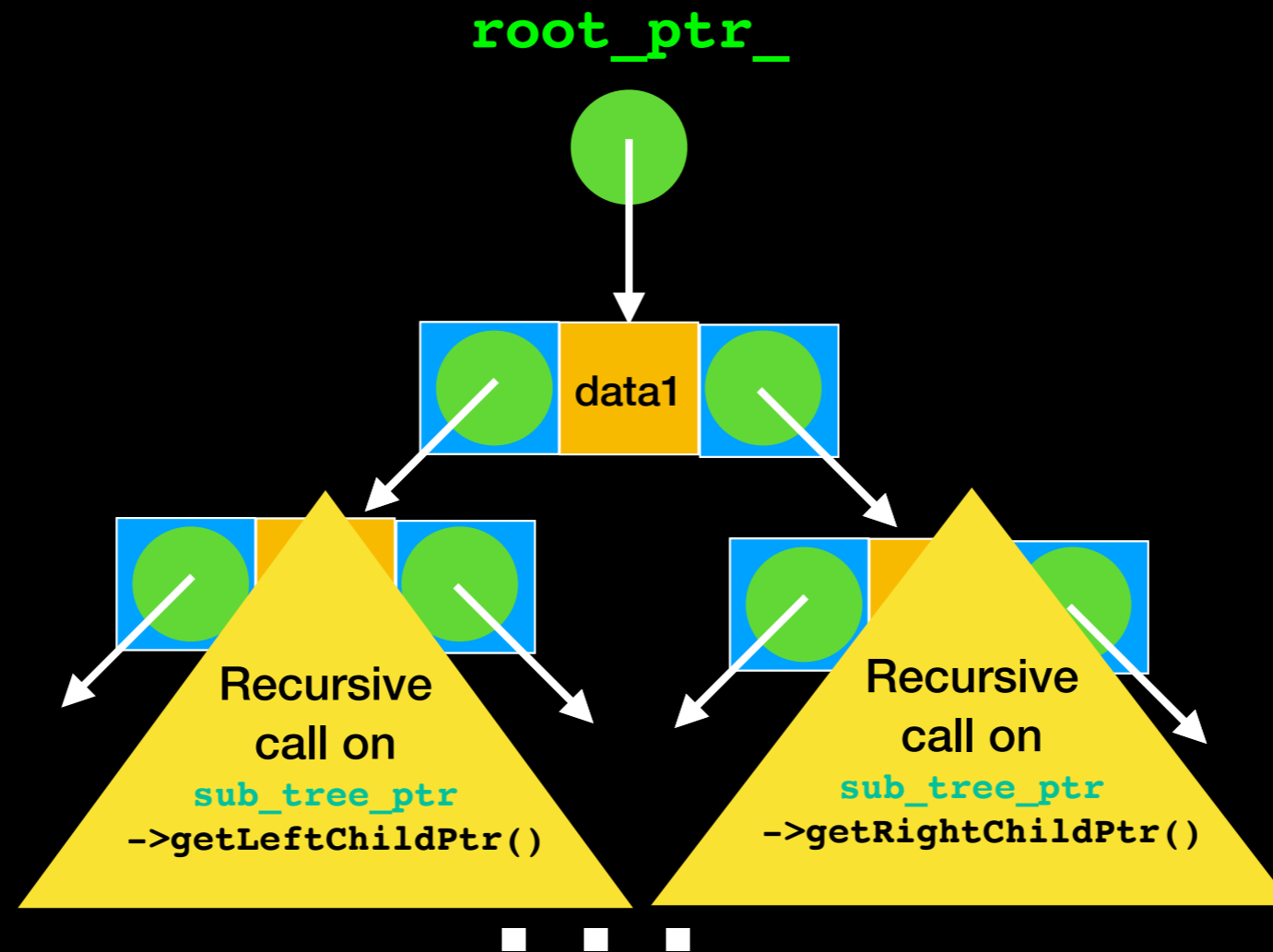
getHeight

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

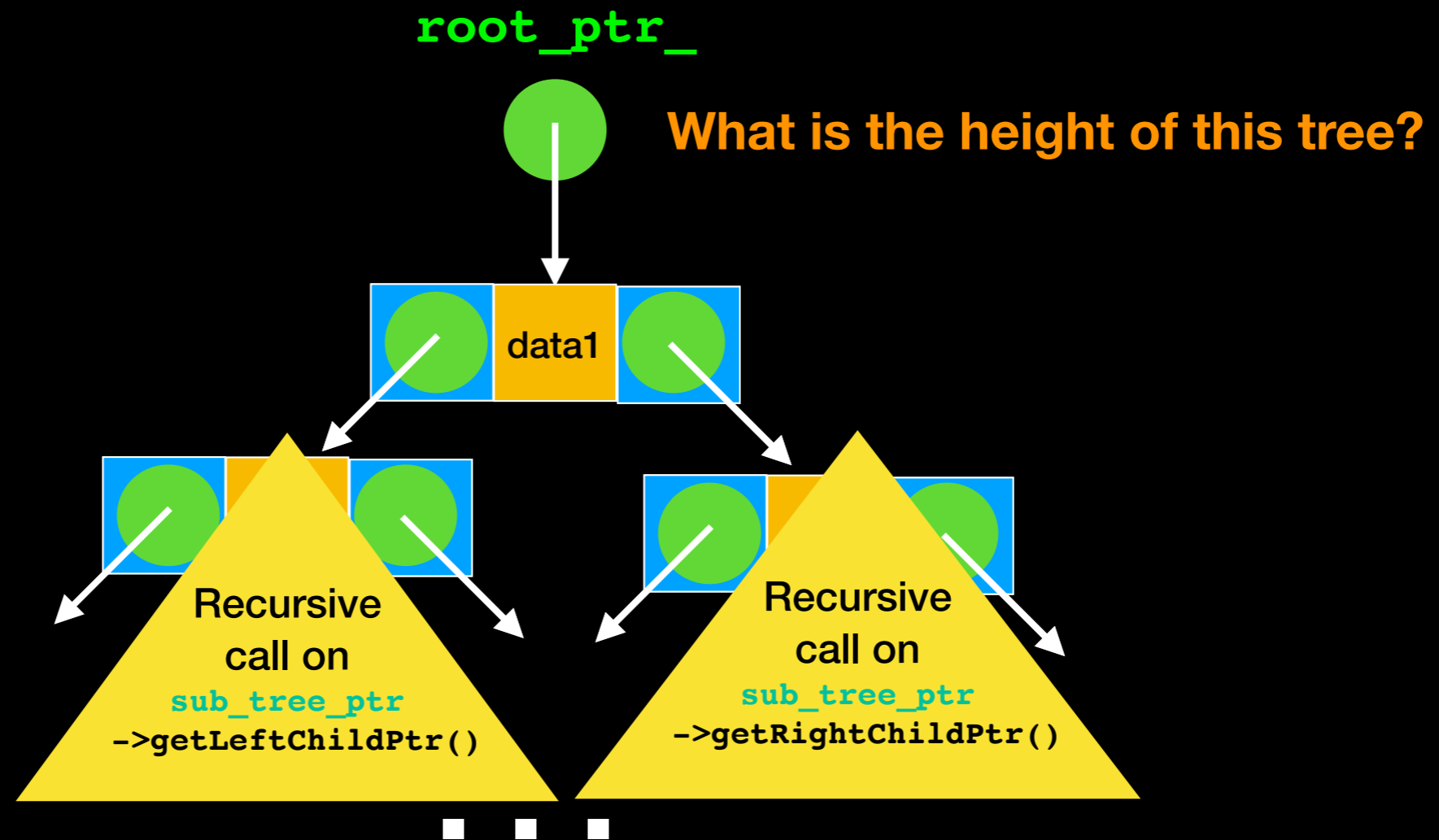
Safe programming: the public method does not take pointer parameter.

Only protected/private methods have access to pointers and may modify tree structure

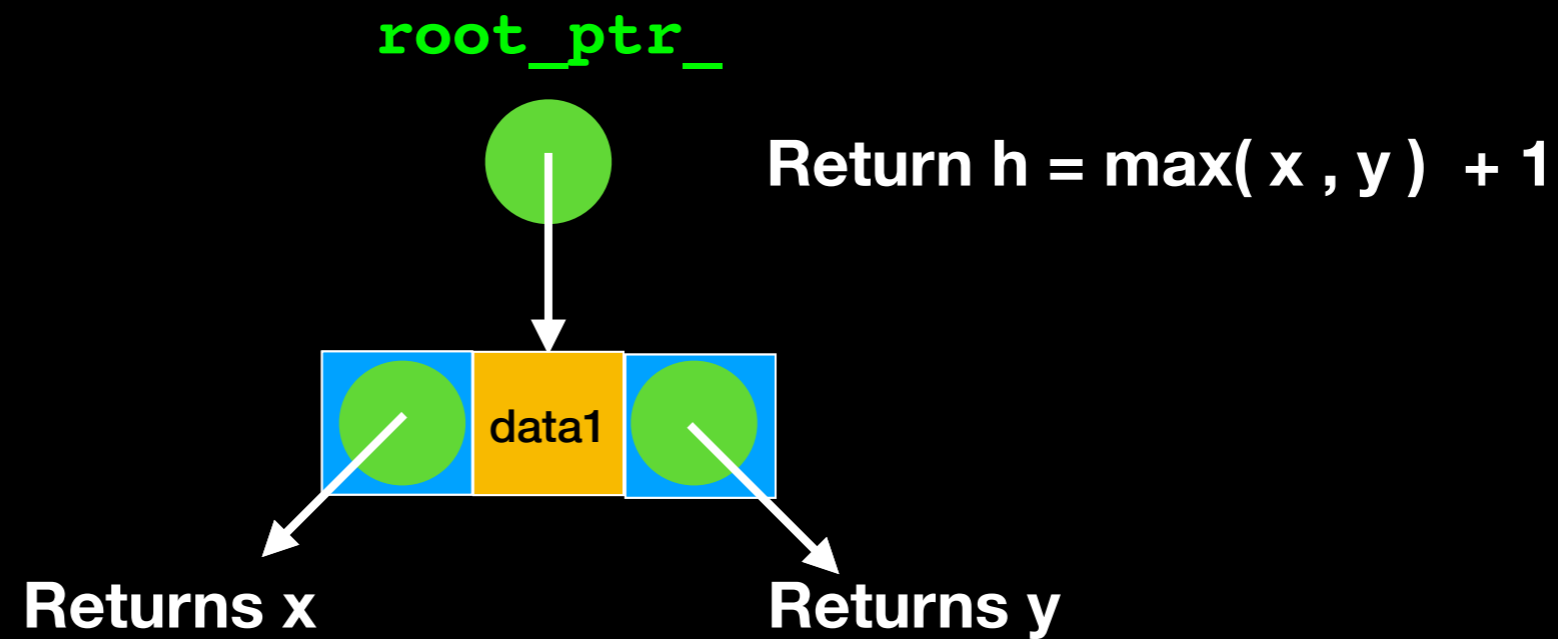
getHeightHelper(**sub_tree_ptr**)



getHeightHelper(**sub_tree_ptr**)



getHeightHelper(**sub_tree_ptr**)



getHeightHelper(sub_tree_ptr)

```
using namespace std;
```

```
template<typename ItemType>
```

```
int BST<ItemType>::getHeightHelper(shared_ptr<BinaryNode<ItemType>>  
                                   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<ItemType>::getNumberOfNodes() const
{
    //try it at home!!!!
}
```

```
int BST<ItemType>::getNumberOfNodesHelper(shared_ptr
                                           <BinaryNode<ItemType>> 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

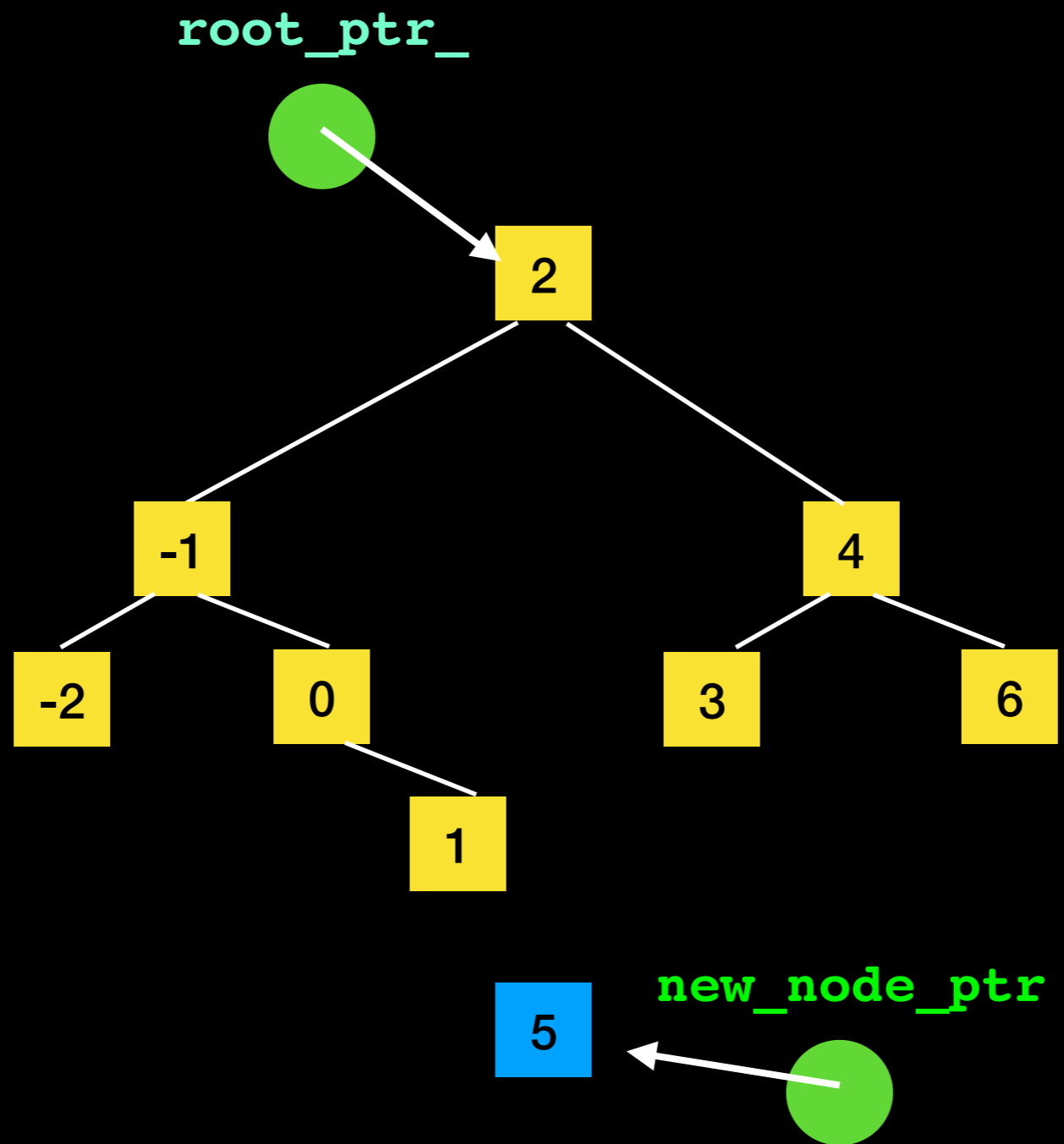
add

```
using namespace std;

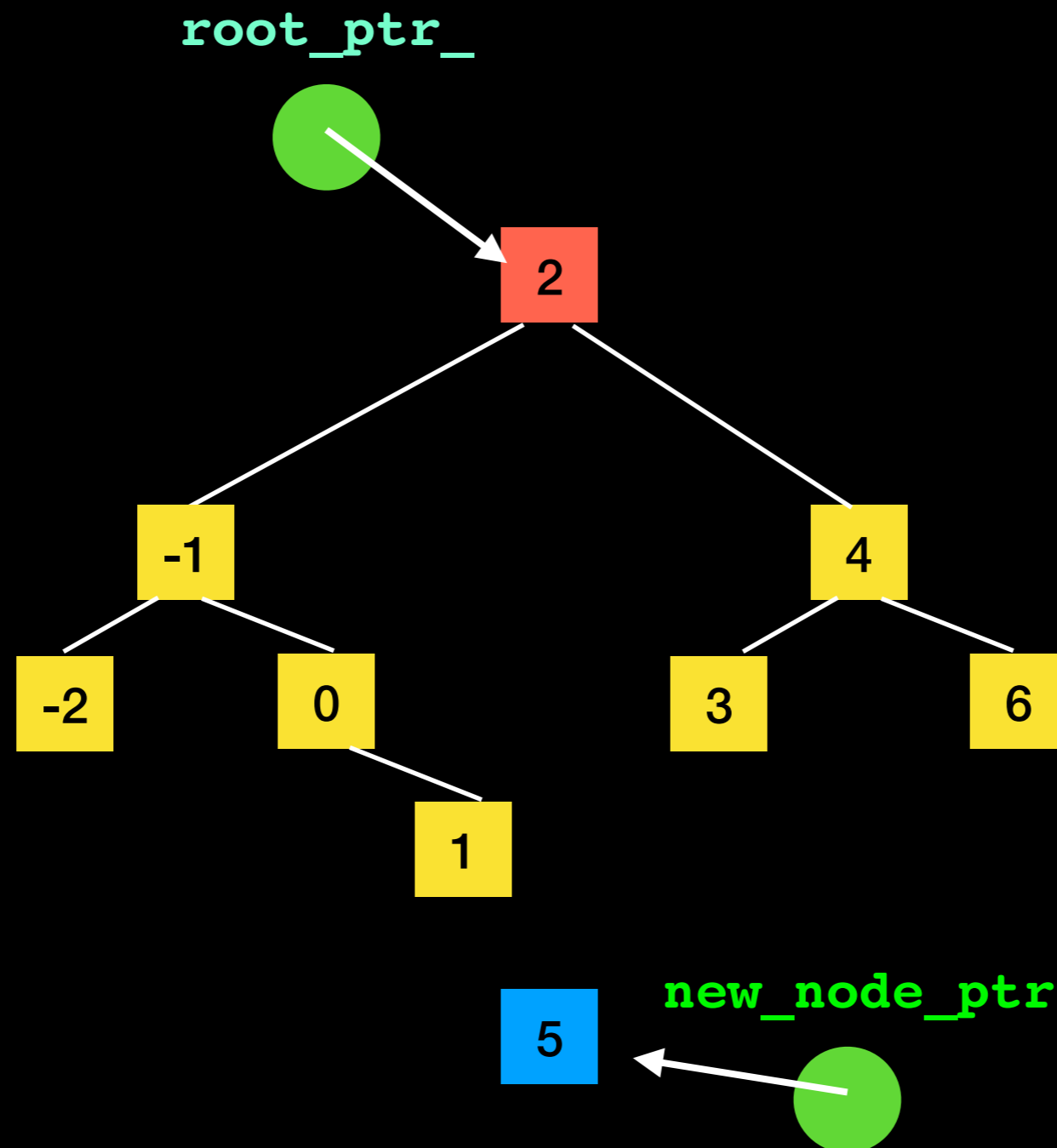
template<typename ItemType>
void BST<ItemType>::add(const ItemType& new_item)
{
    auto new_node_ptr =
        make_shared<BinaryNode<ItemType>>(new_item);
    placeNode(root_ptr_, new_node_ptr);
} // end add
```

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

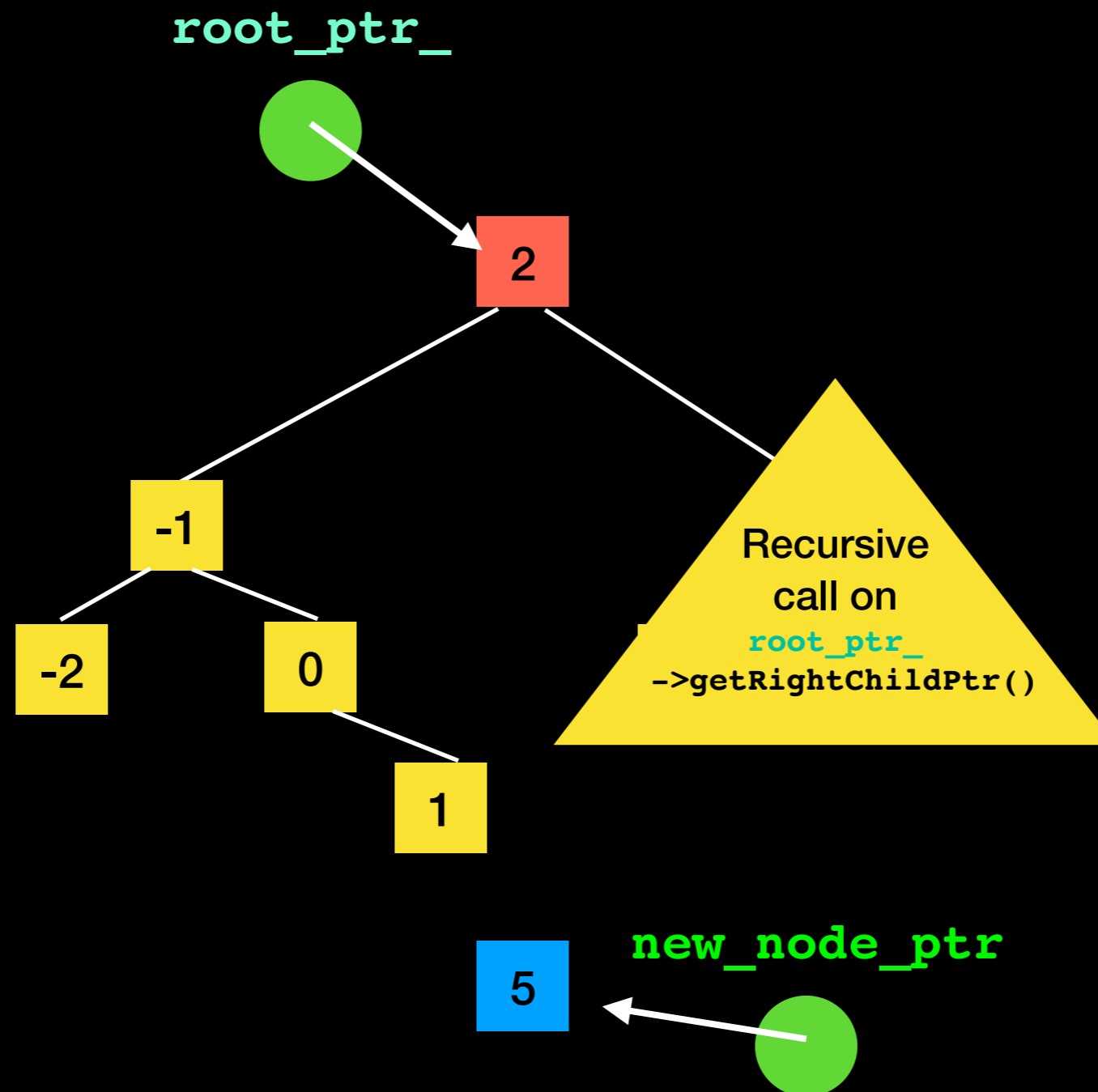
```
placeNode(root_ptr_, new_node_ptr);
```



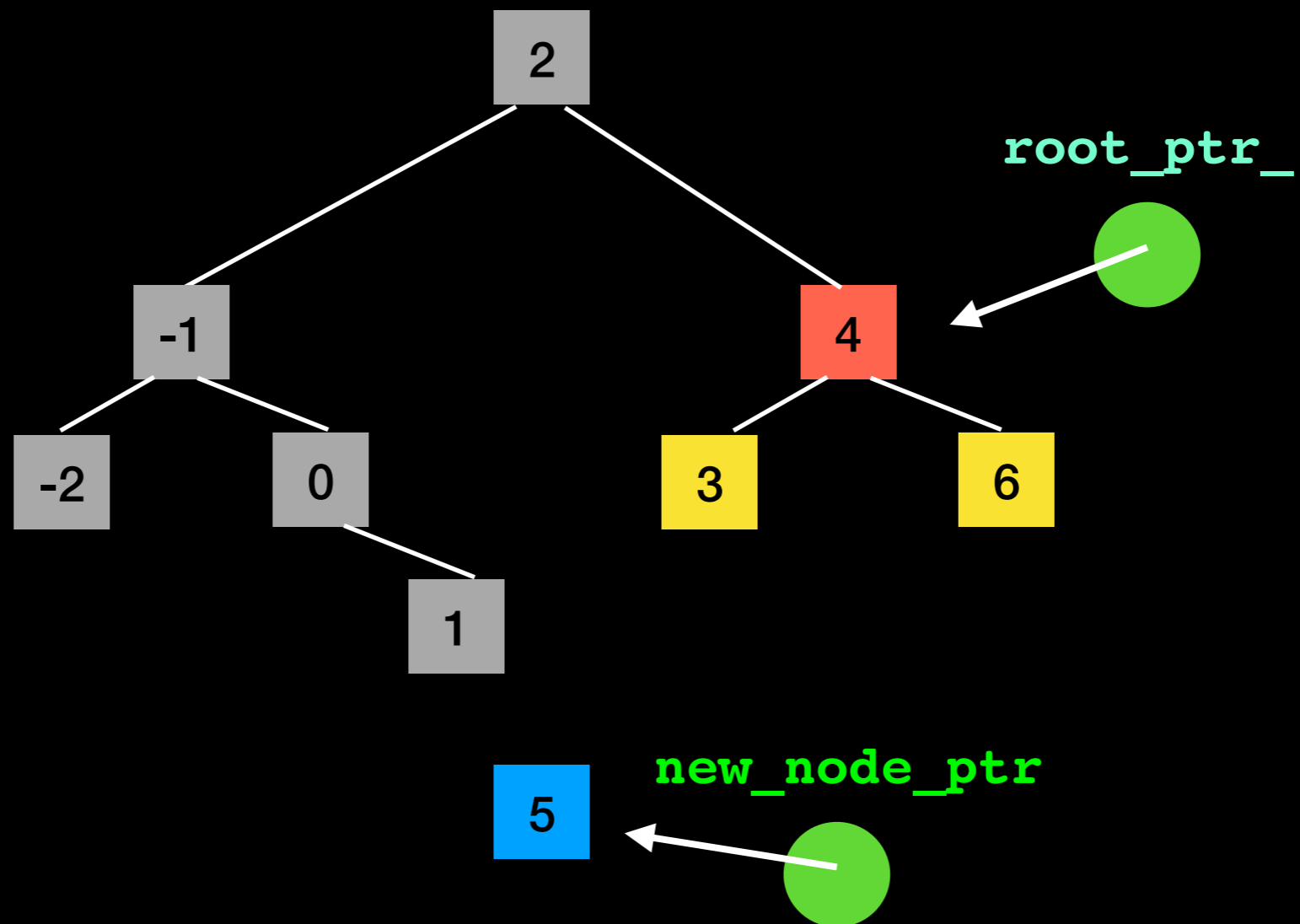
```
placeNode(root_ptr_, new_node_ptr);
```



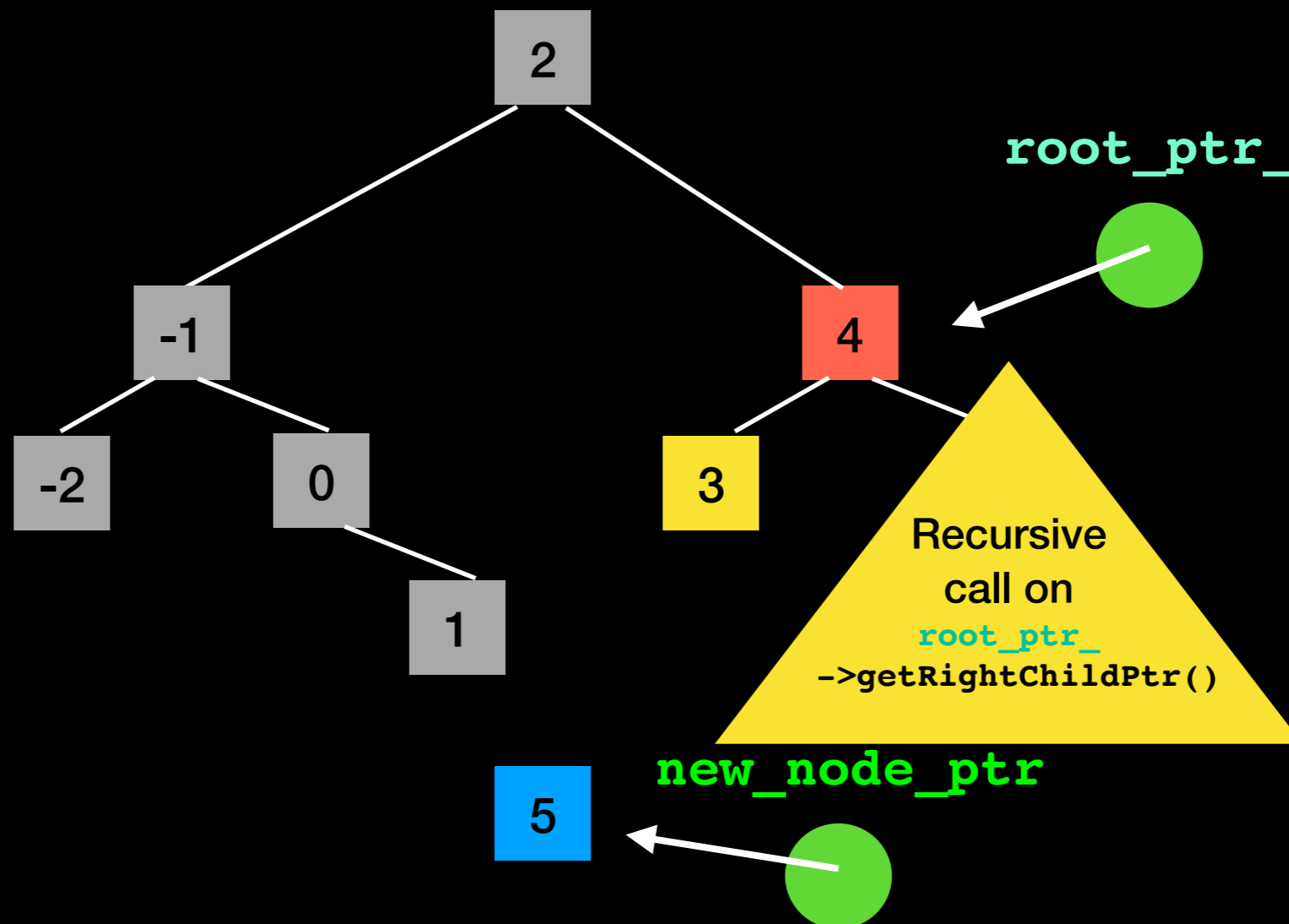

```
placeNode(root_ptr_, new_node_ptr);
```



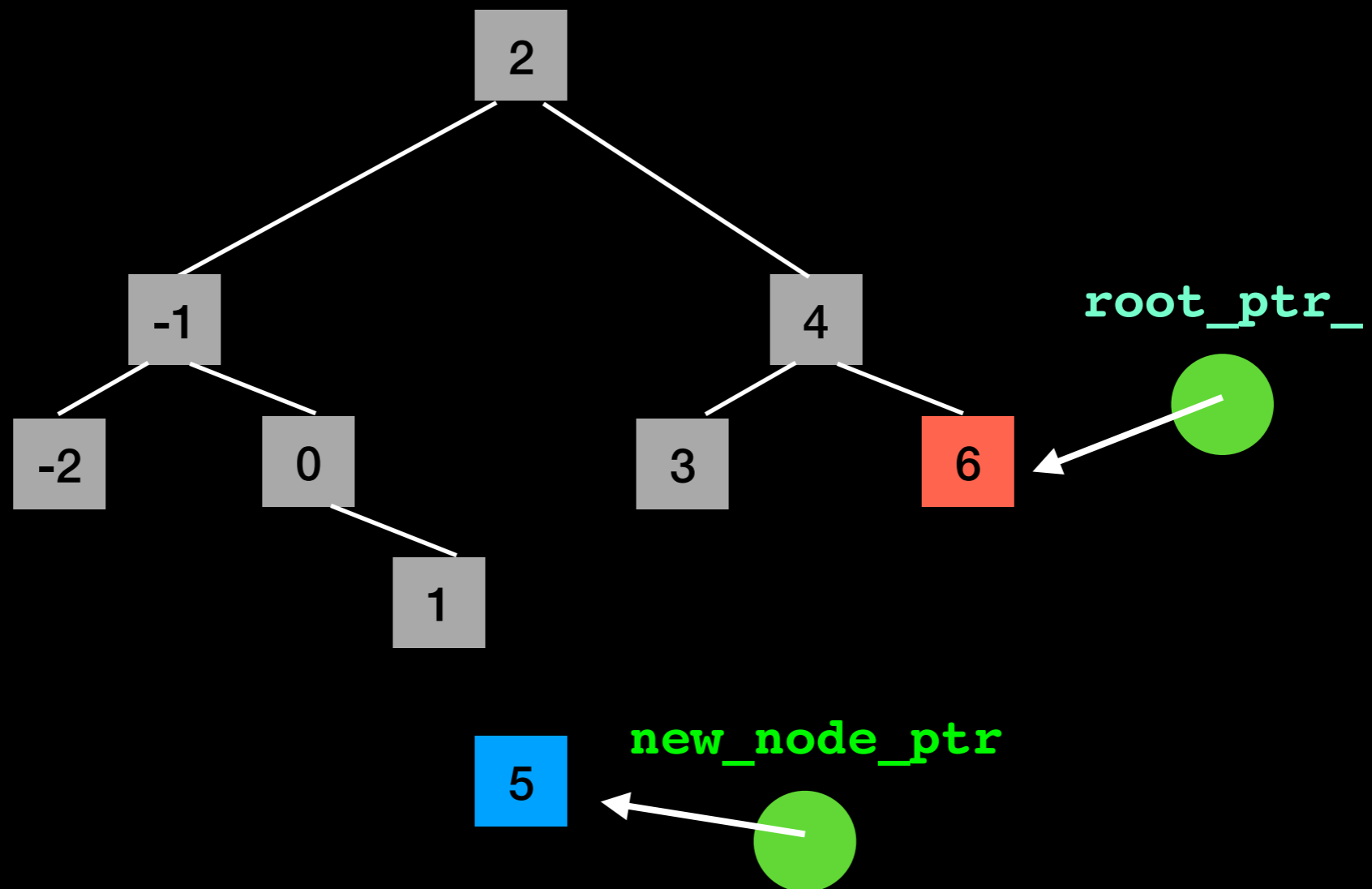
```
placeNode(root_ptr_, new_node_ptr);
```



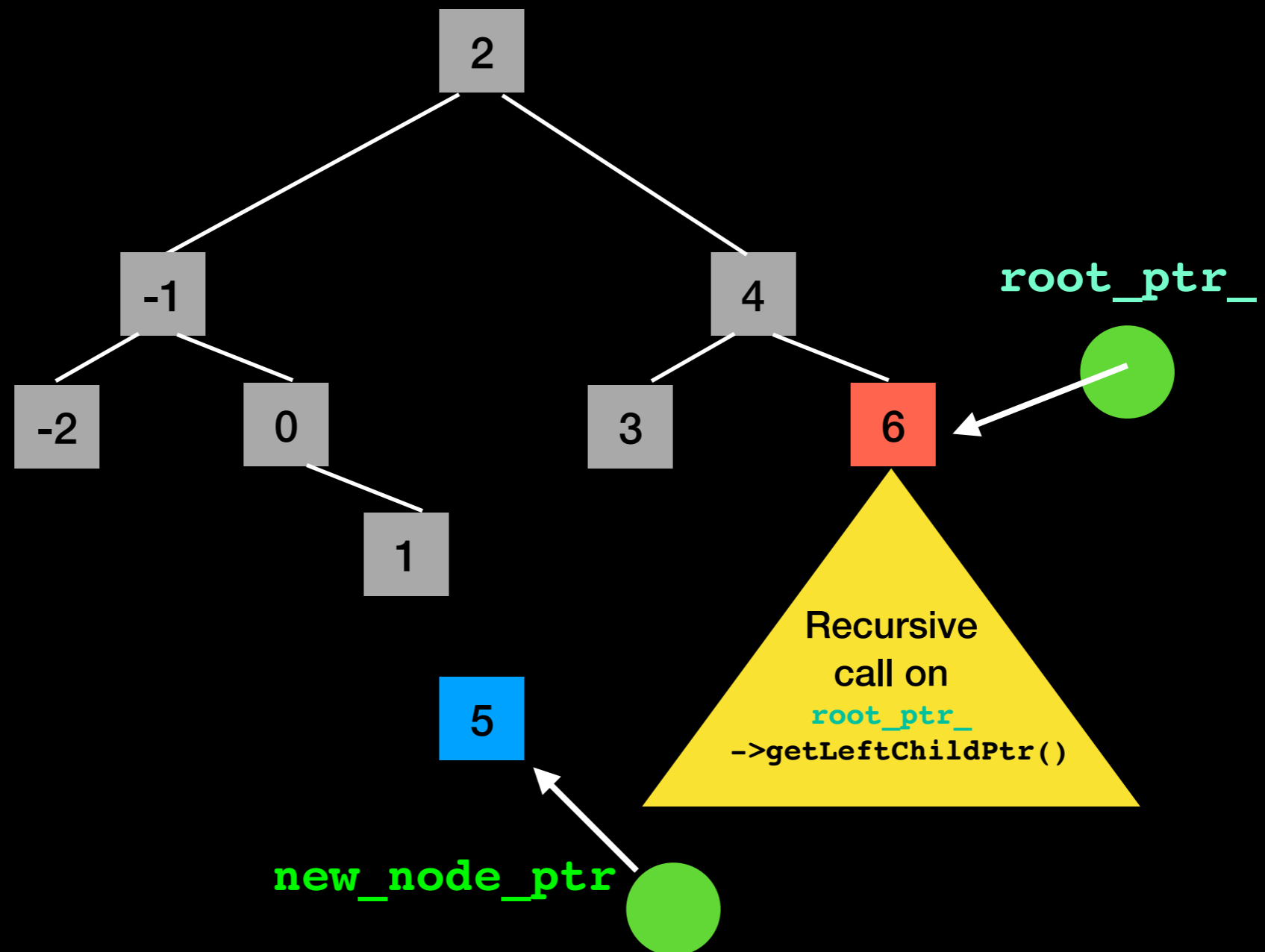
```
placeNode(root_ptr_, new_node_ptr);
```



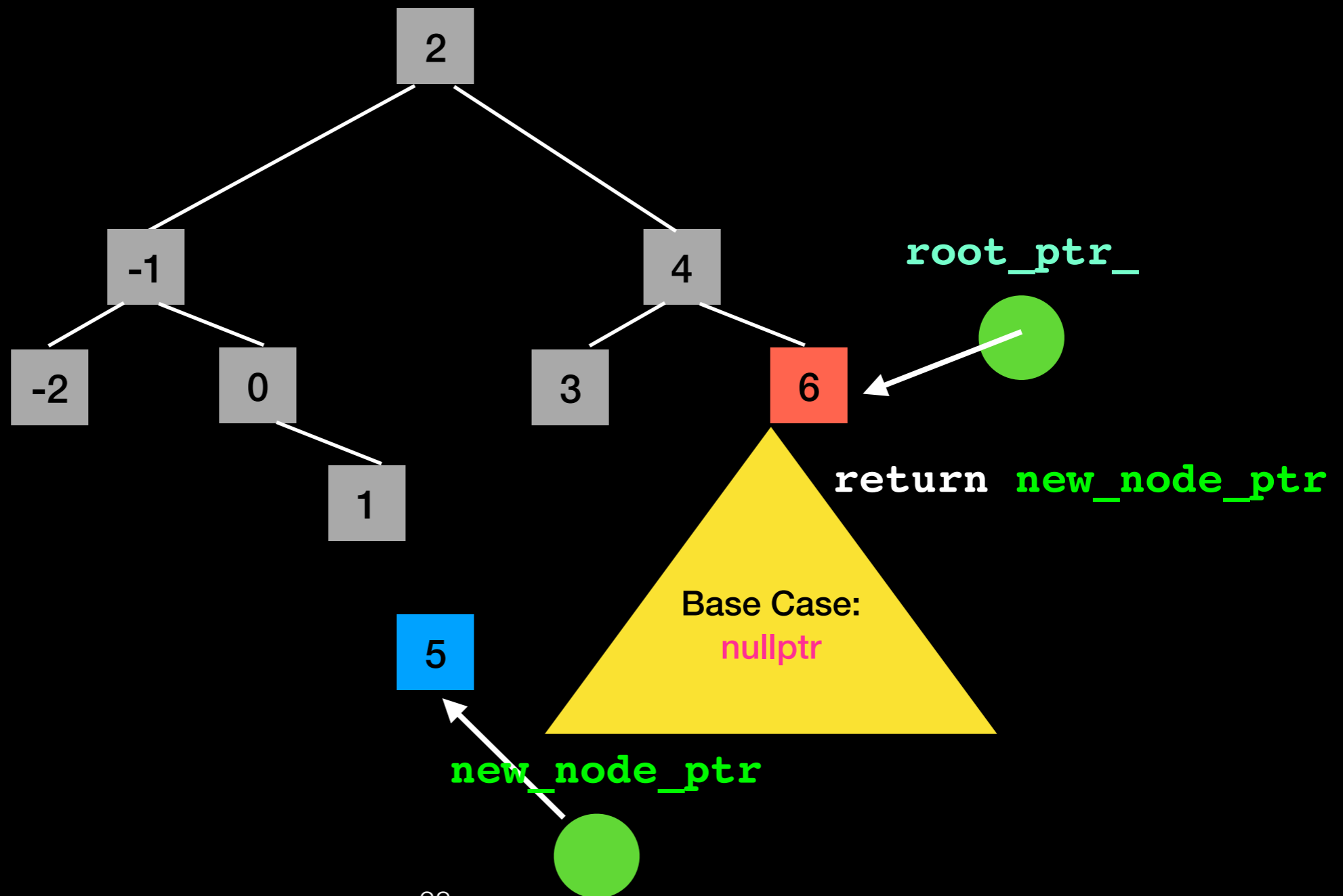
```
placeNode(root_ptr_, new_node_ptr_);
```



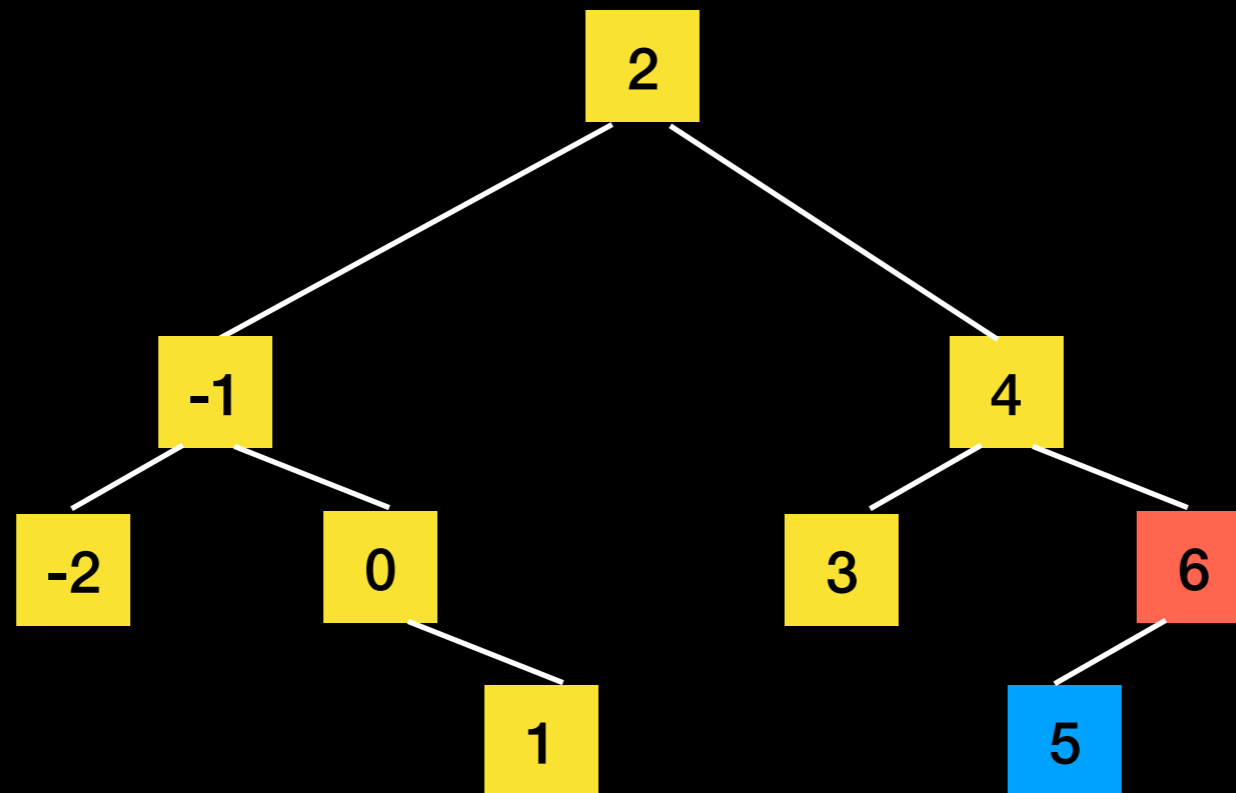
```
placeNode(root_ptr_, new_node_ptr);
```



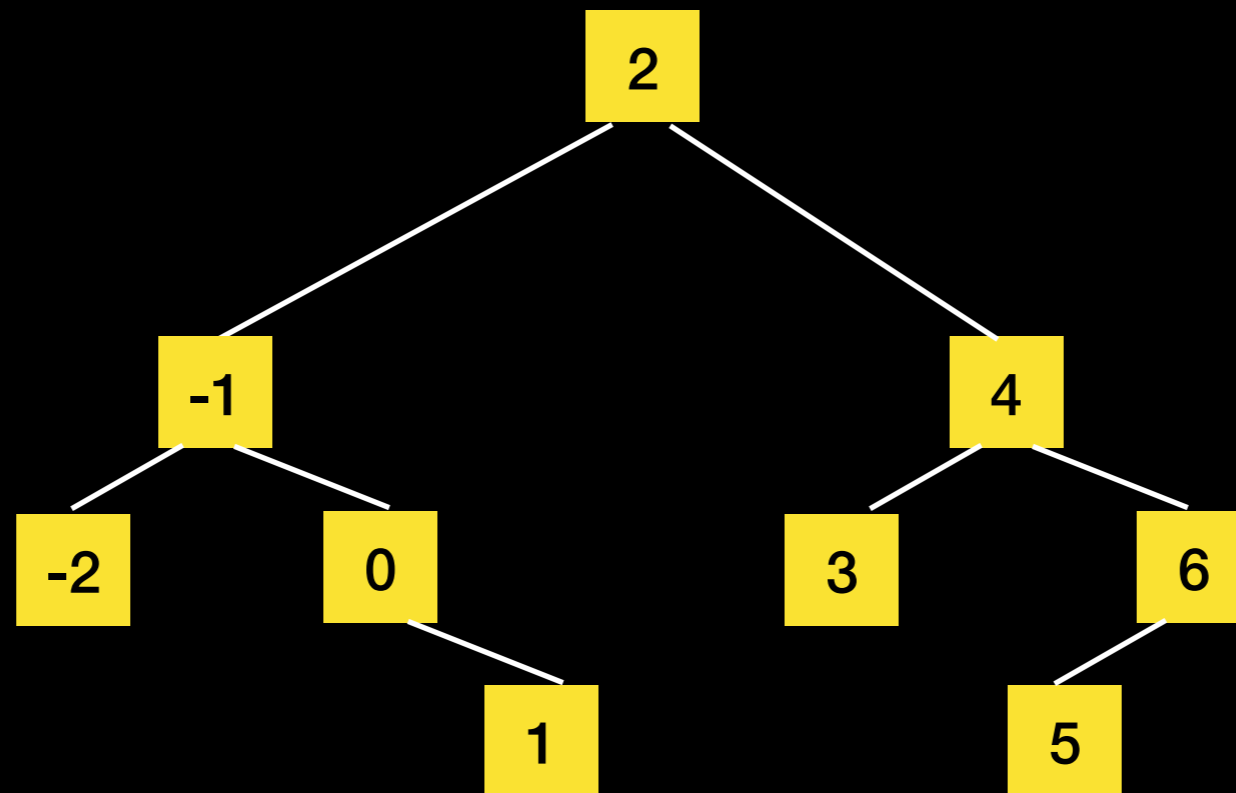
```
placeNode(root_ptr_, new_node_ptr);
```



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



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



add helper function

```
using namespace std;
```

```
template<typename ItemType>
```

```
auto BST<template<typename ItemType>::placeNode(
```

```
    shared_ptr<BinaryNode<template<typename ItemType>>> subtree_ptr,
```

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

Safe programming: 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<typename ItemType>
auto BST<ItemType>::removeValue(std::shared_ptr<BinaryNode<ItemType>>
                               subtree_ptr, const ItemType 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 continued

```
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
```

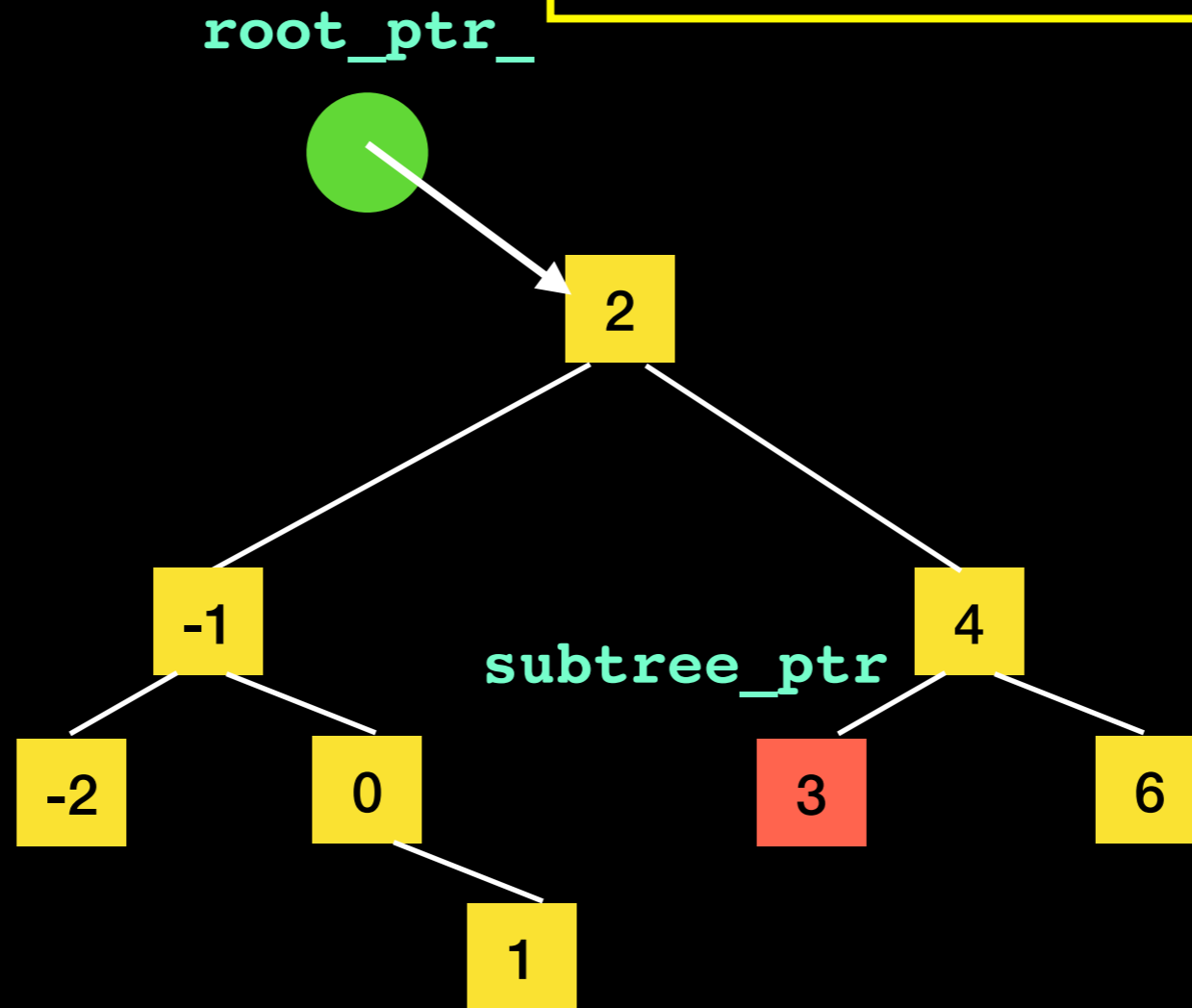
Search for target in left subtree

Search for target in right subtree

```
removeNode(subtree_ptr);
```

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

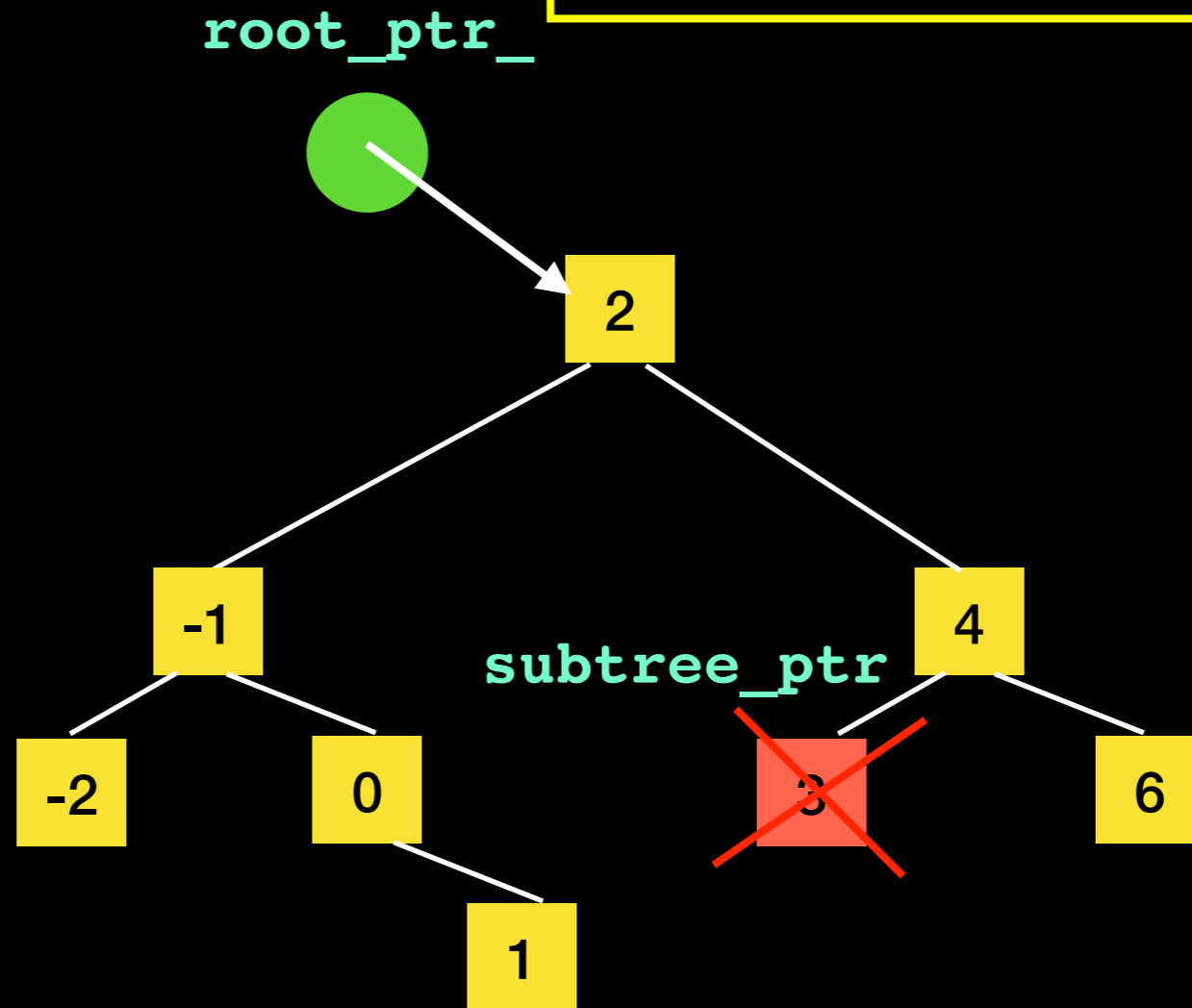
Case 1: target is a leaf



```
removeNode(subtree_ptr);
```

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

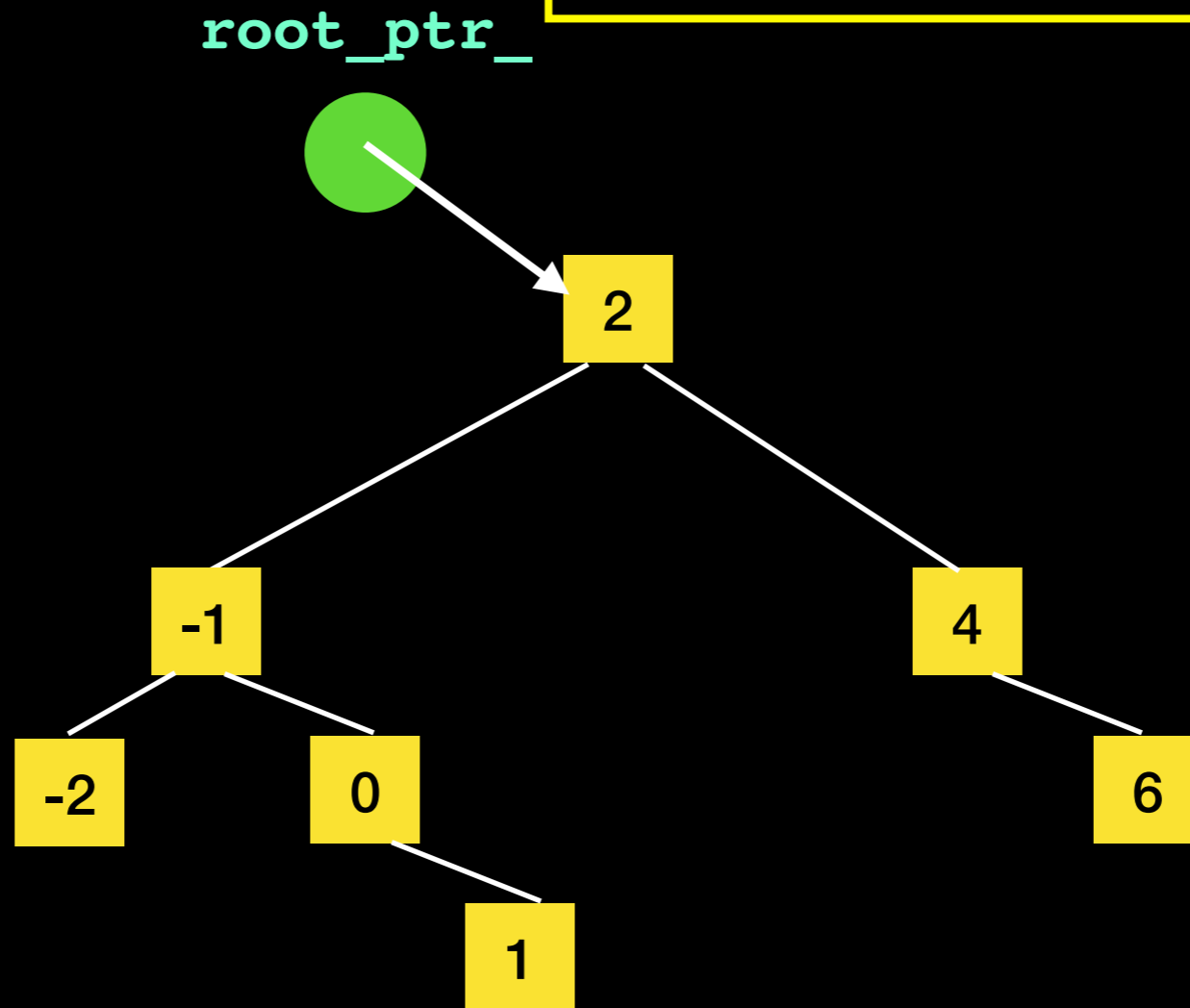
Case 1: target is a leaf



```
removeNode(subtree_ptr);
```

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

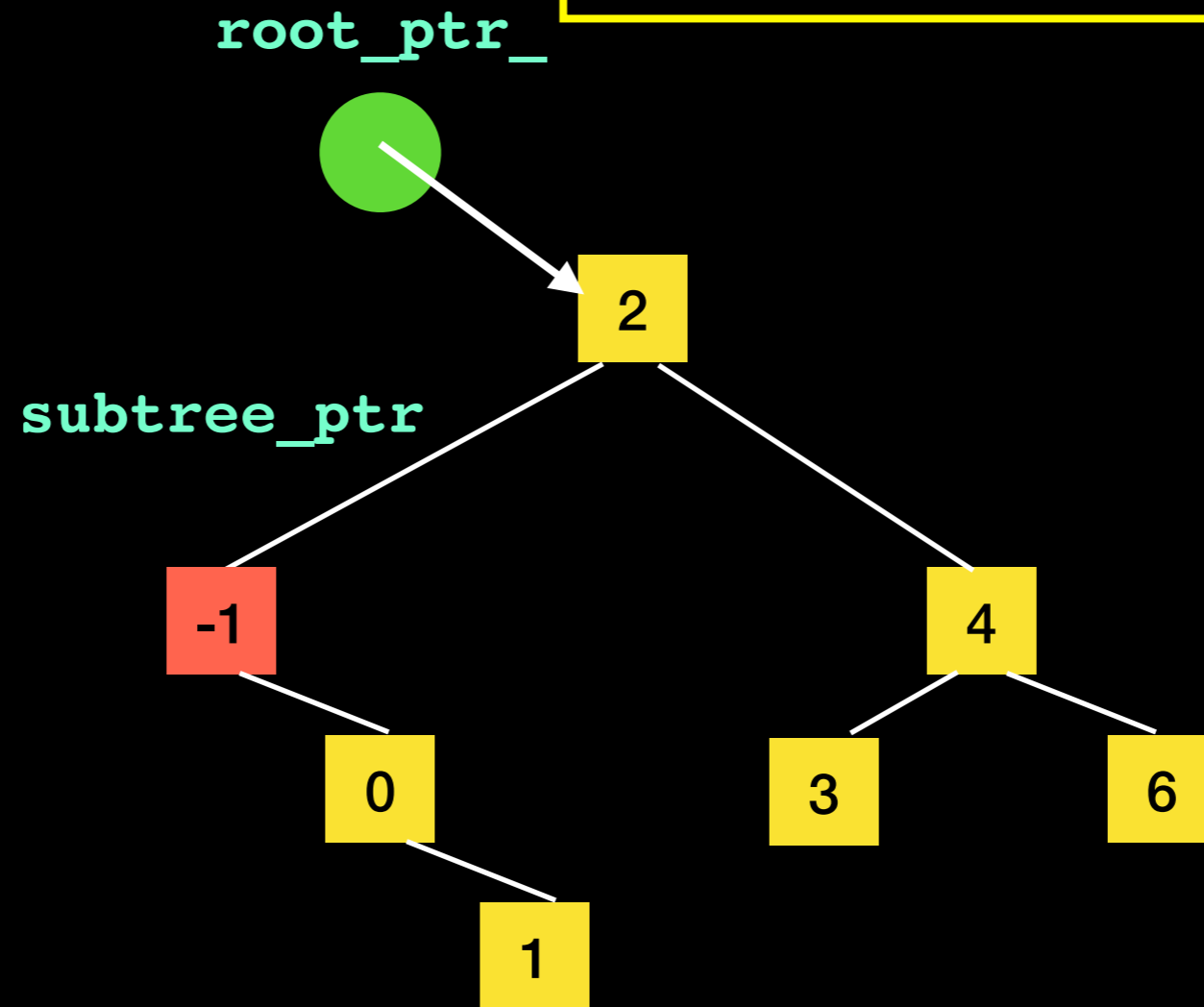
Case 1: target is a leaf



removeNode(subtree_ptr);

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

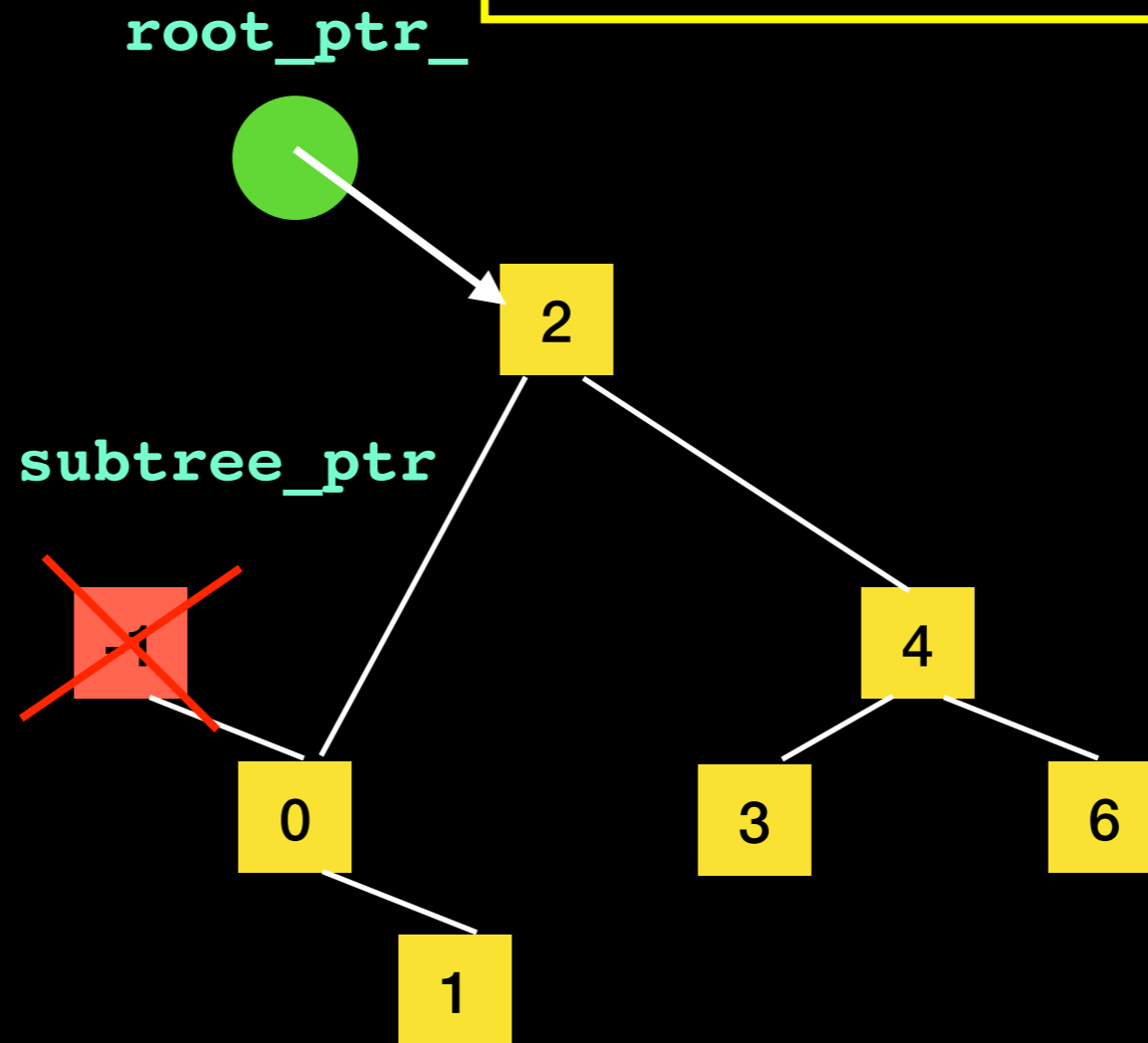
Case 2: target has 1 child
Left and right case are symmetric



removeNode(subtree_ptr);

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

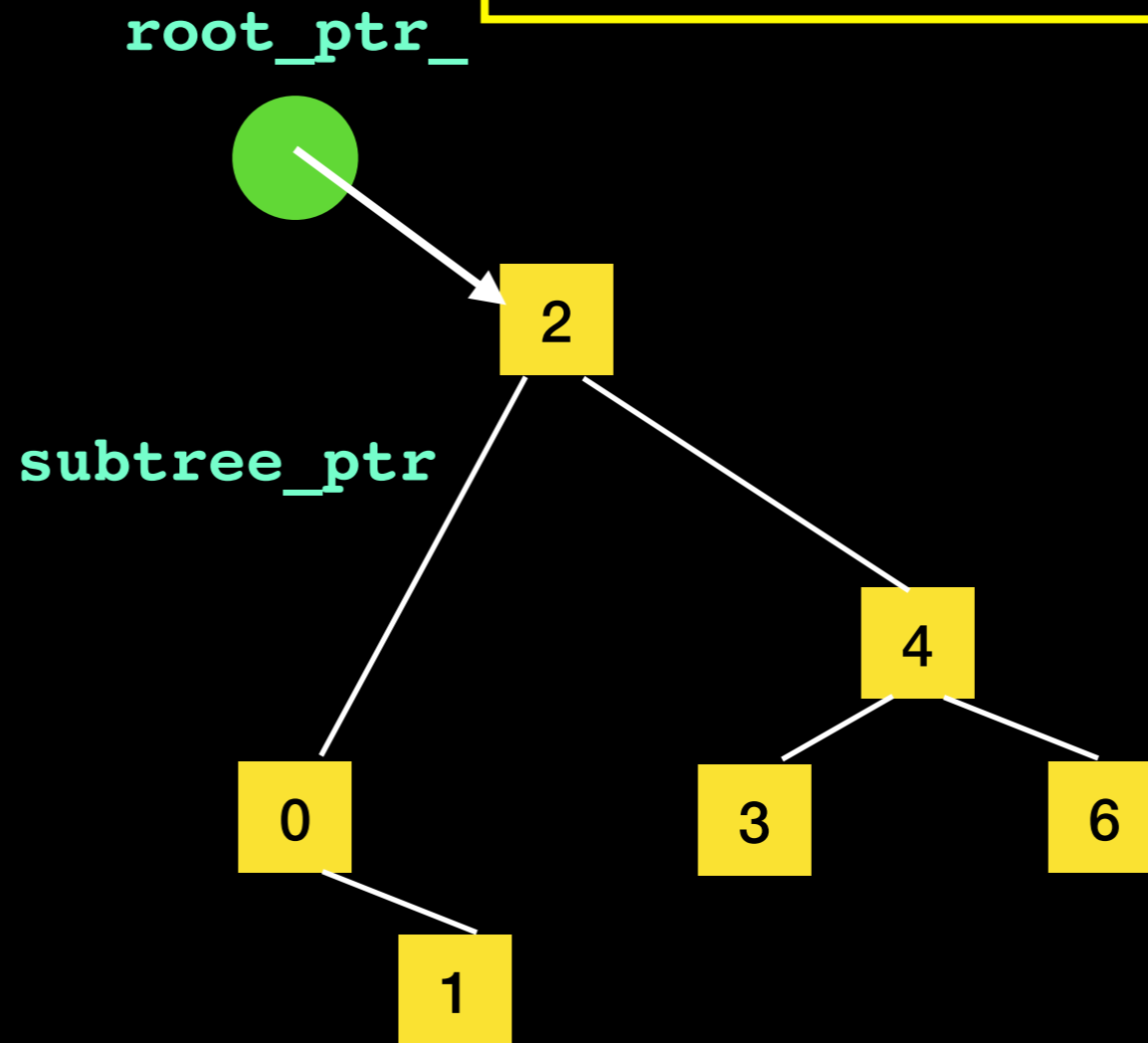
Case 2: target has 1 child
Left and right case are symmetric



removeNode(subtree_ptr);

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

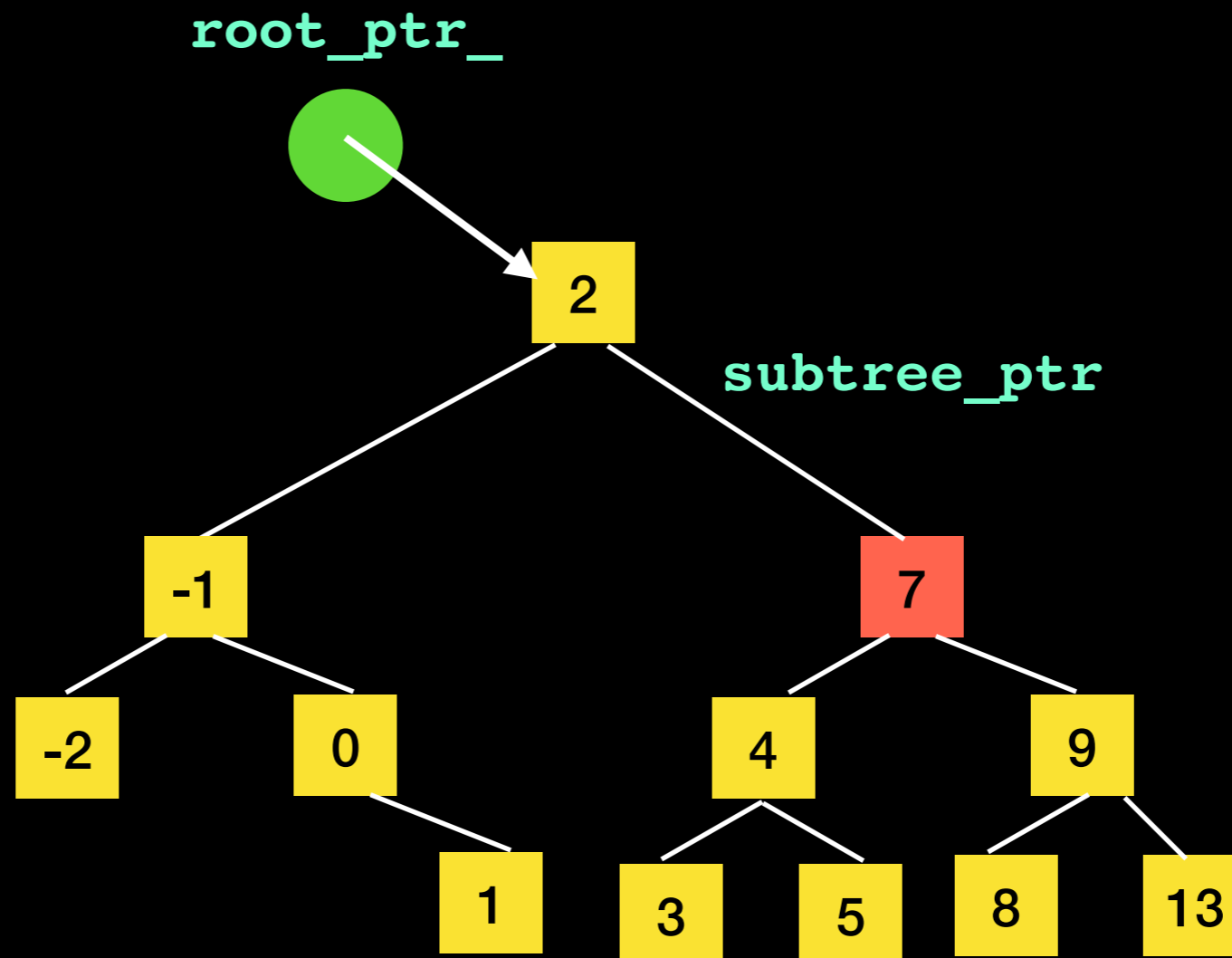
Case 2: target has 1 child
Left and right case are symmetric



Lecture Activity

How would you remove node 7?

Case 3: target has 2 children



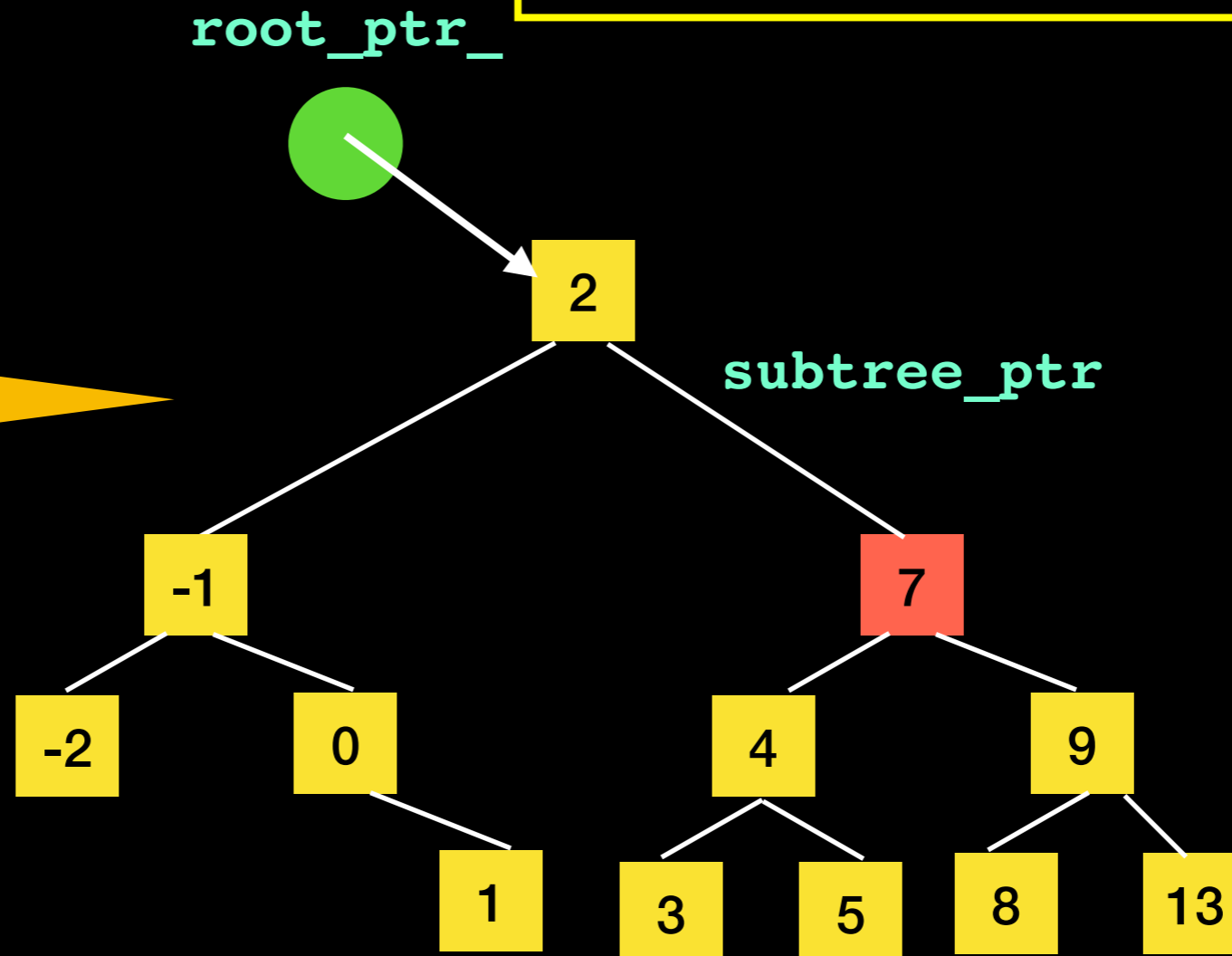
```
removeNode(subtree_ptr);
```

```
if (subtree_ptr->getItem() == target)
{ //Item is 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);

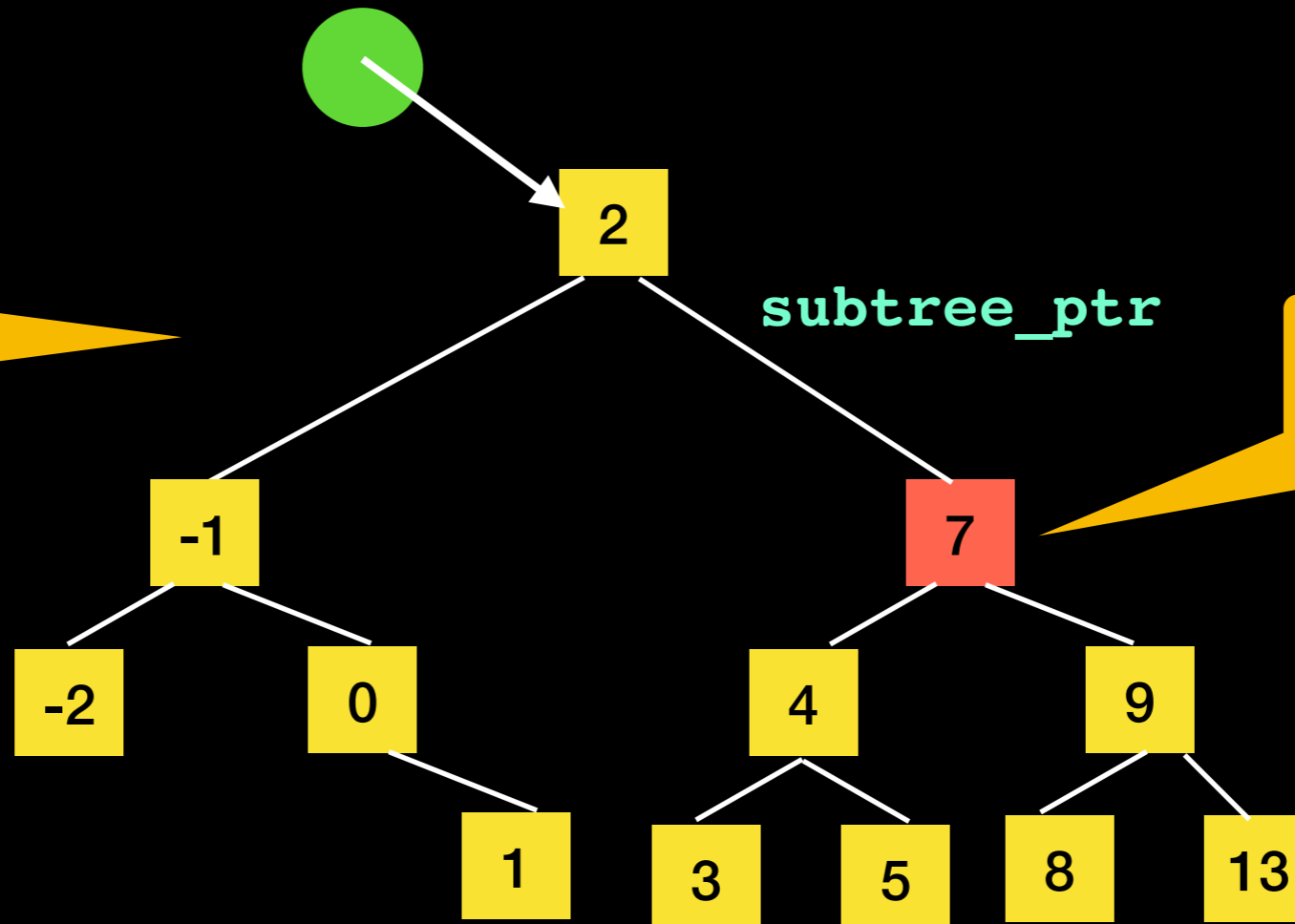
```
if (subtree_ptr->getItem() == target)
{ //Item is the root of this subtree
  subtree_ptr =
    removeNode(subtree_ptr);
  success = true;
  return subtree_ptr;
}
```

Case 3: target has 2 children



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

root_ptr_



What value should we put here?

removeNode(subtree_ptr);

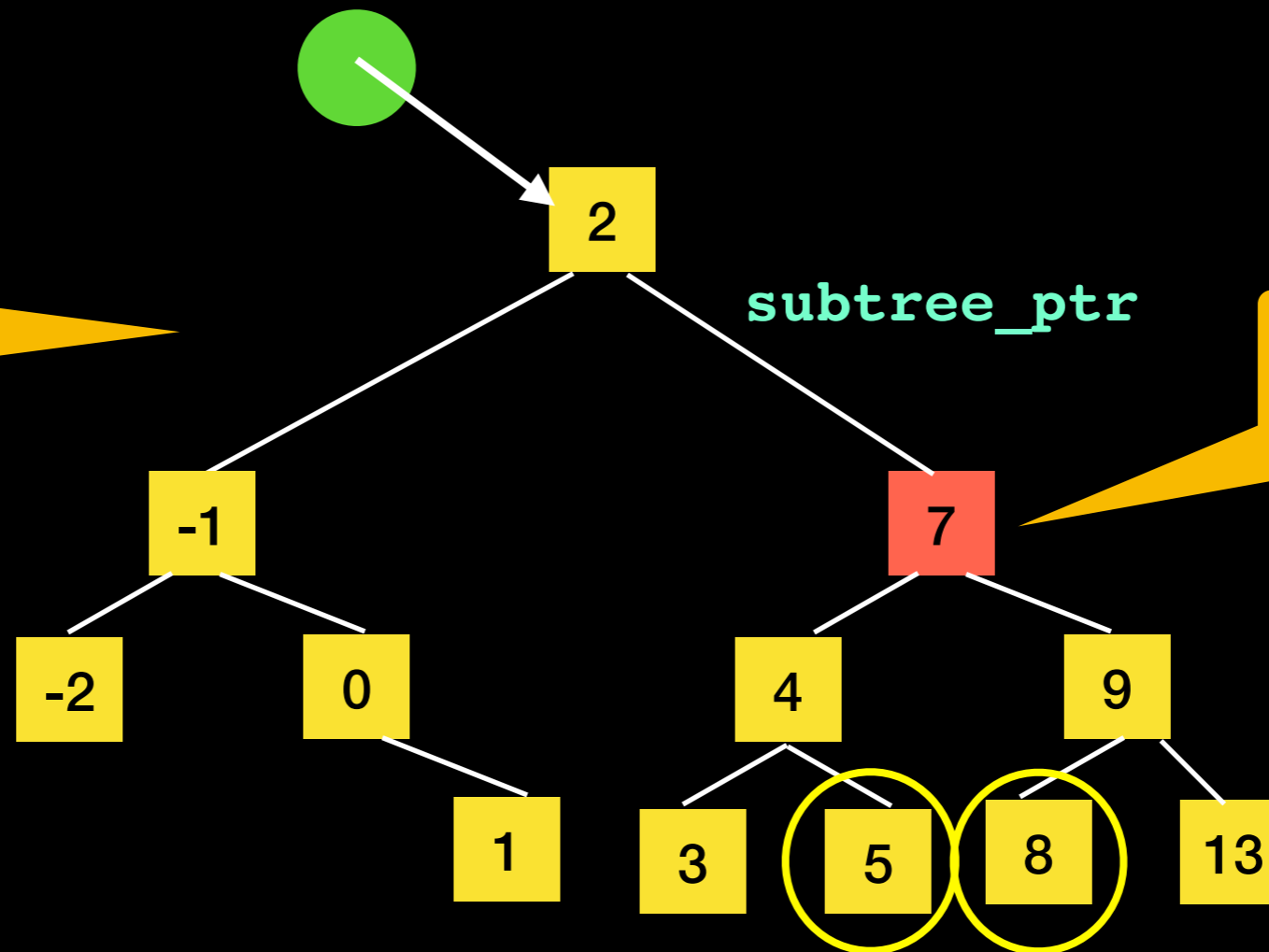
```
if (subtree_ptr->getItem() == target)
{ //Item is the root of this subtree
  subtree_ptr =
    removeNode(subtree_ptr);
  success = true;
  return subtree_ptr;
}
```

Case 3: target has 2 children



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

root_ptr_



What value should we put here?

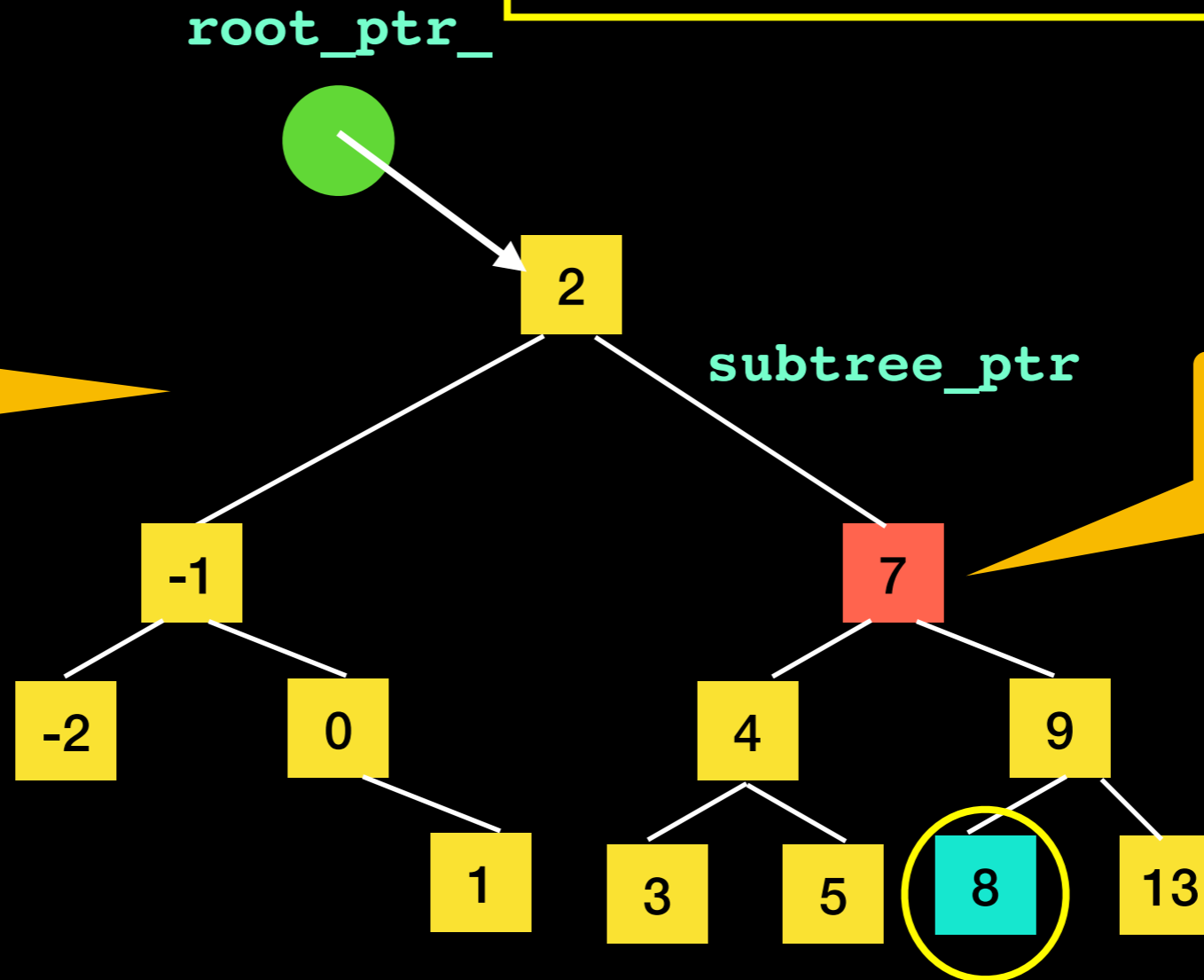
```
removeNode(subtree_ptr);
```

```
if (subtree_ptr->getItem() == target)
{ //Item is 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.

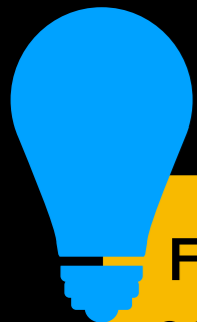


The *inorder successor*

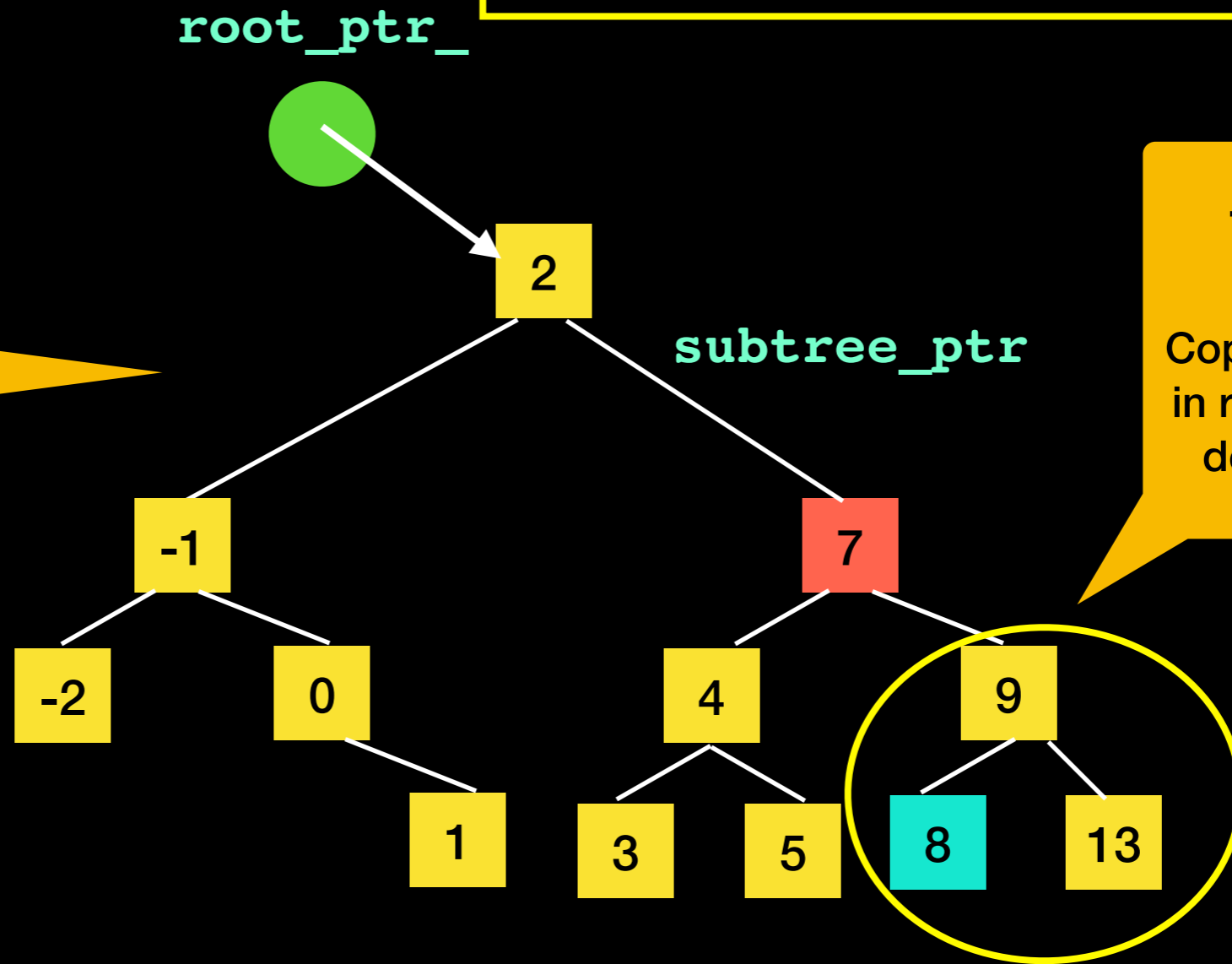
removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
{ //Item is 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.

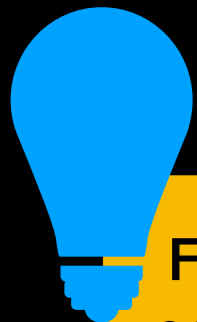


The *inorder* successor:
Copy smallest value in right subtree and delete that node

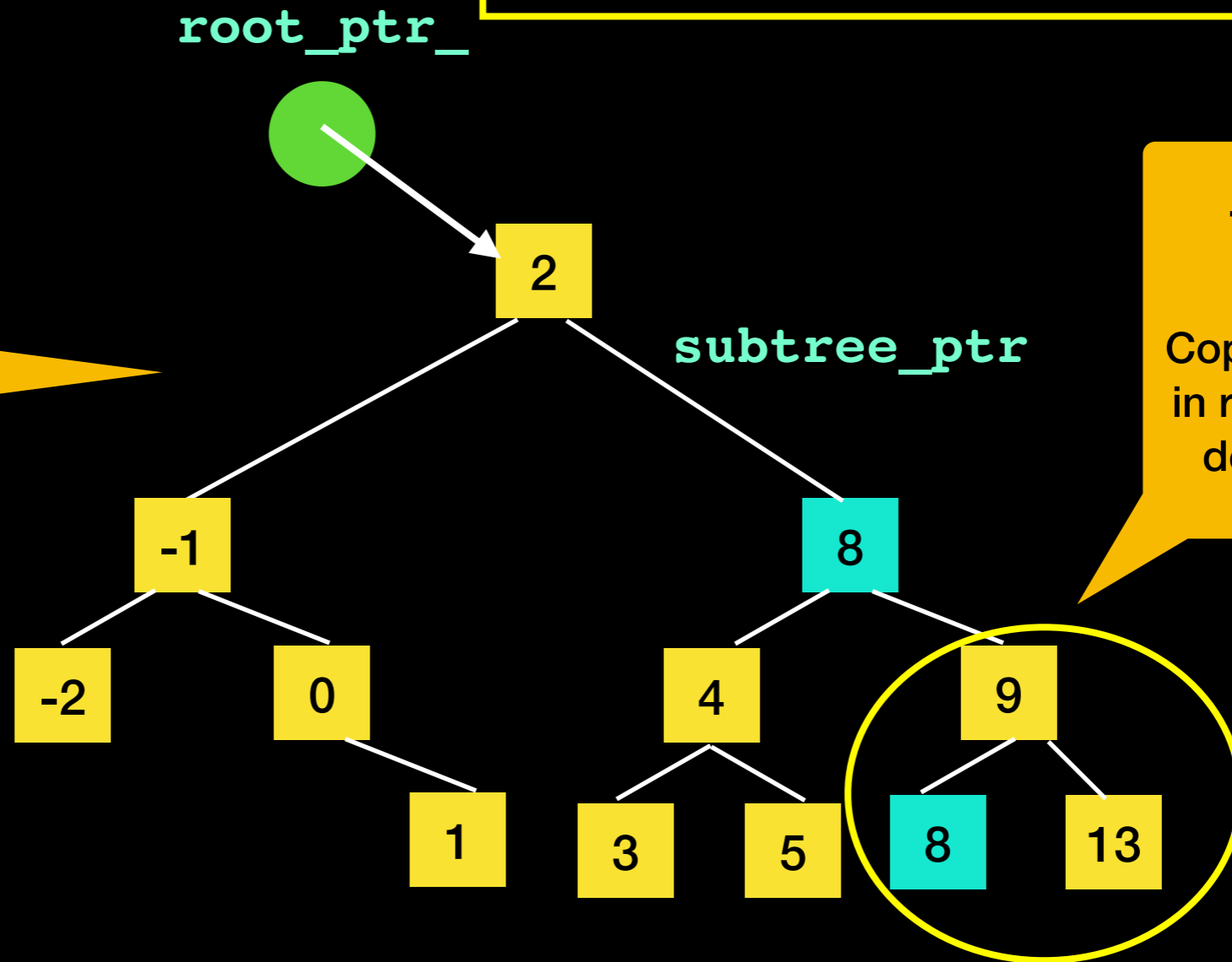
removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
{ //Item is 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

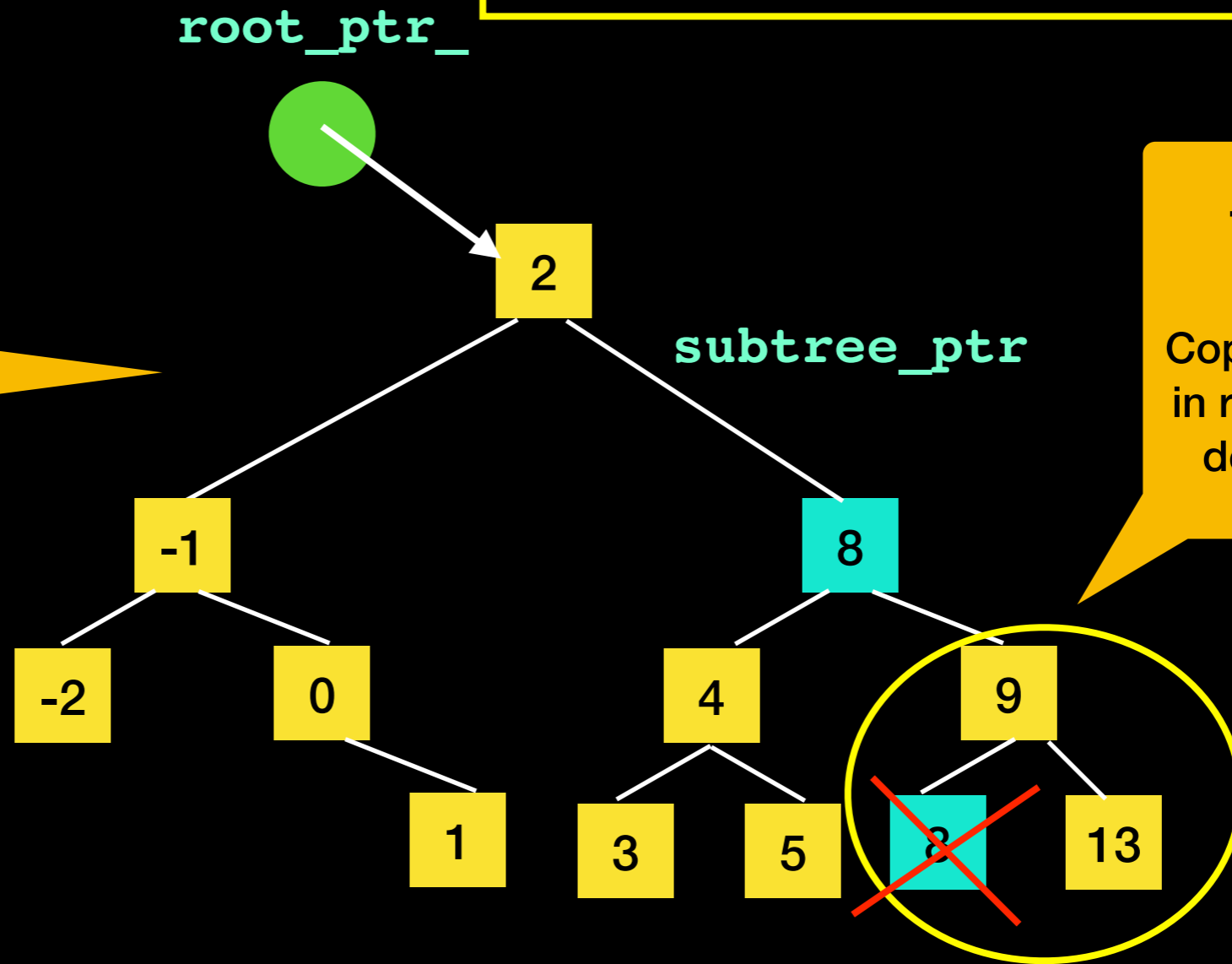
removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
{ //Item is 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);

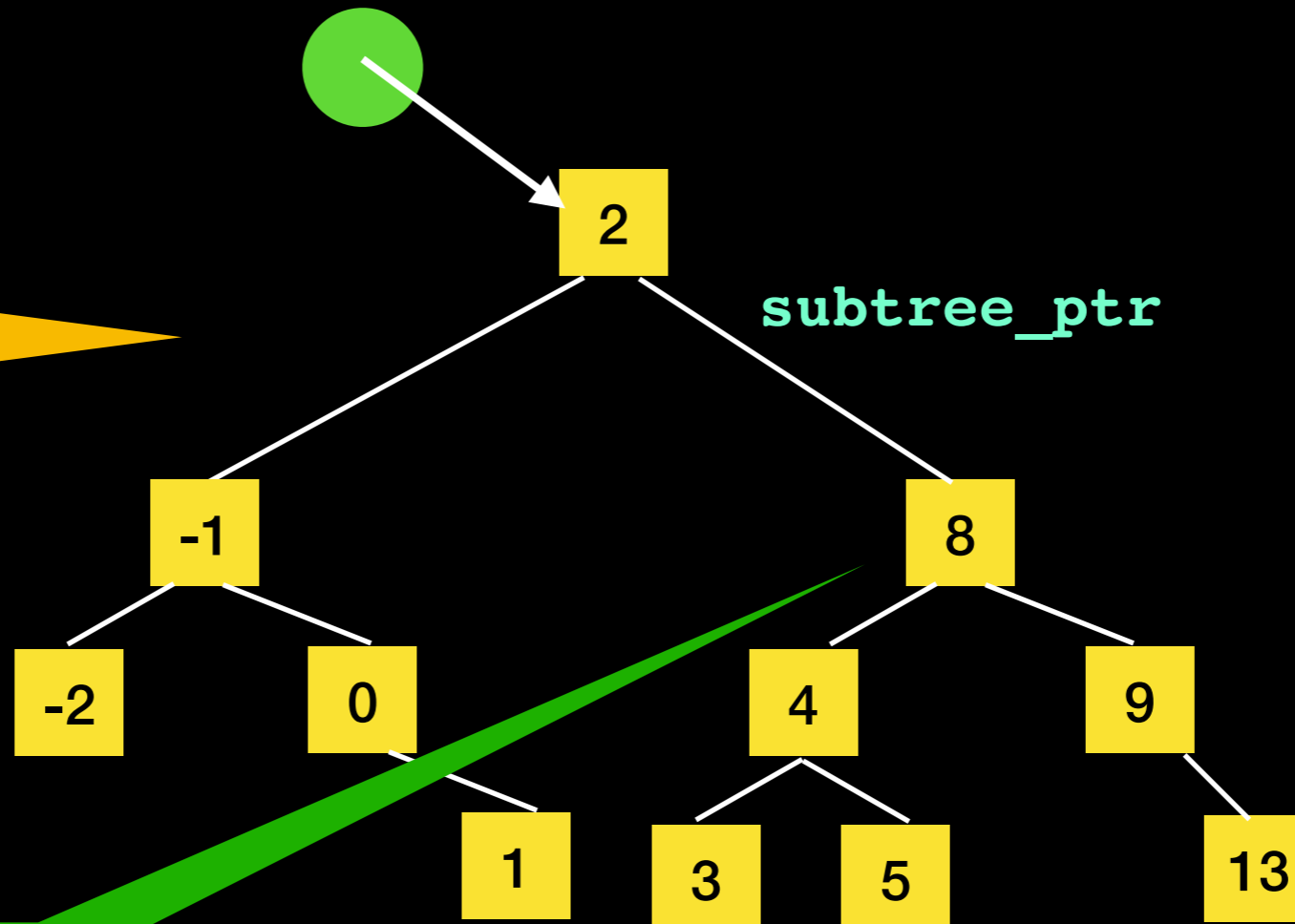
```
if (subtree_ptr->getItem() == target)
{ //Item is the root of this subtree
  subtree_ptr =
    removeNode(subtree_ptr);
  success = true;
  return subtree_ptr;
}
```

Case 3: target has 2 children



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

root_ptr_



subtree_ptr

This operation will actually "reorganize" the tree

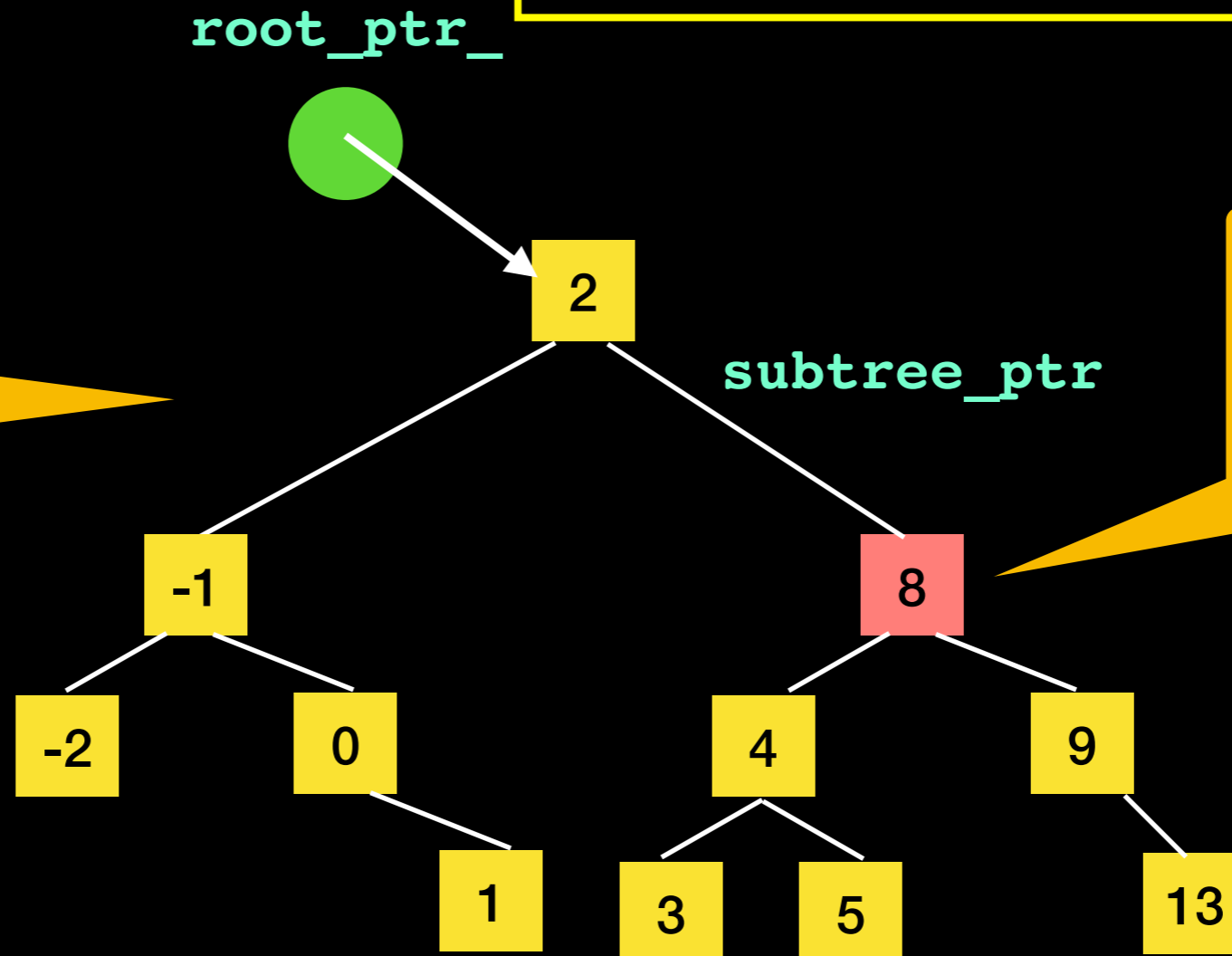
```
removeNode(subtree_ptr);
```

```
if (subtree_ptr->getItem() == target)
{ //Item is 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.

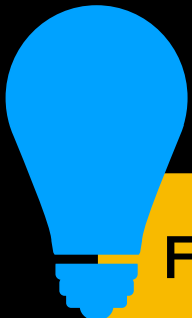


What about removing 8 now? What value should we put here?

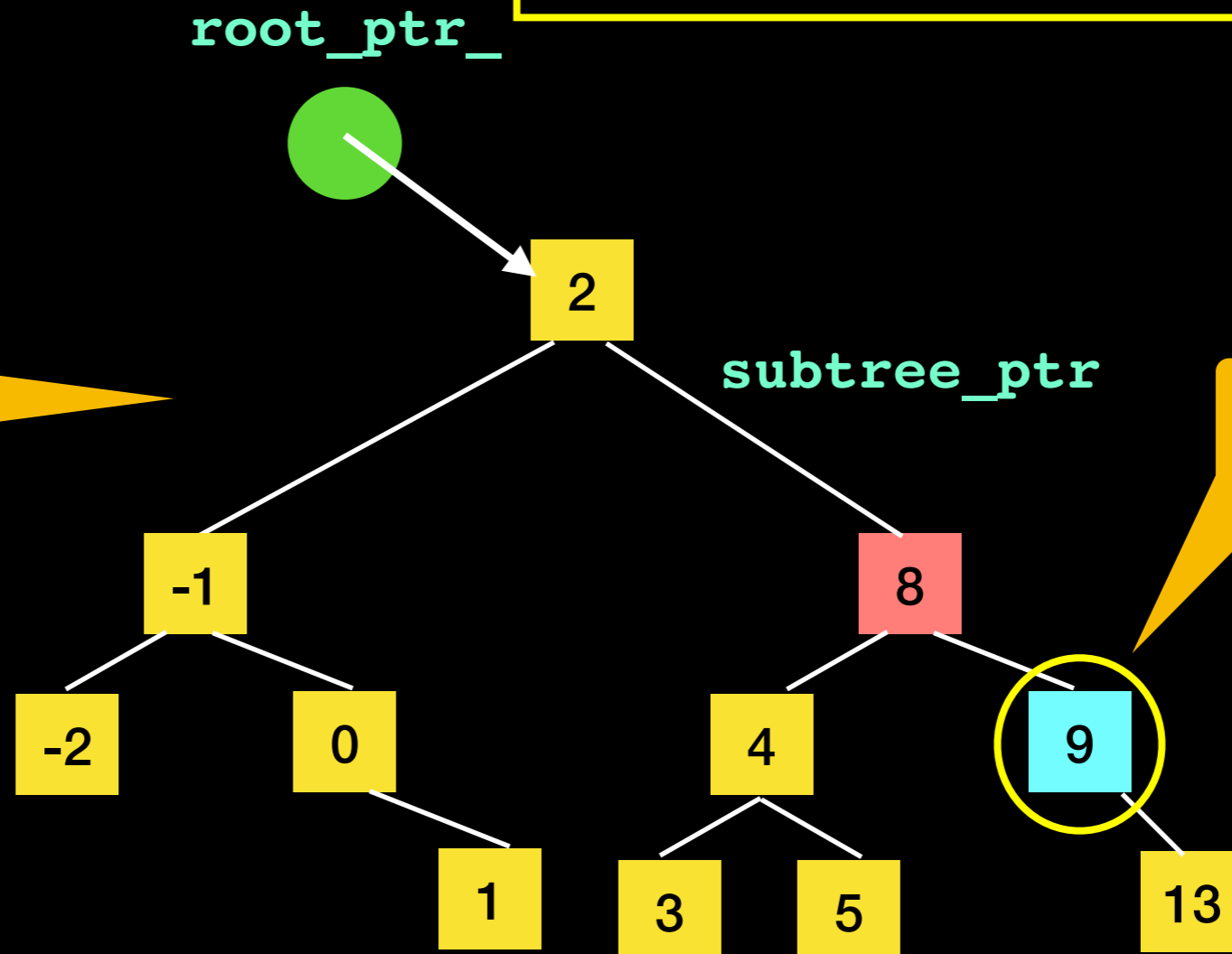
```
removeNode(subtree_ptr);
```

```
if (subtree_ptr->getItem() == target)
{ //Item is 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

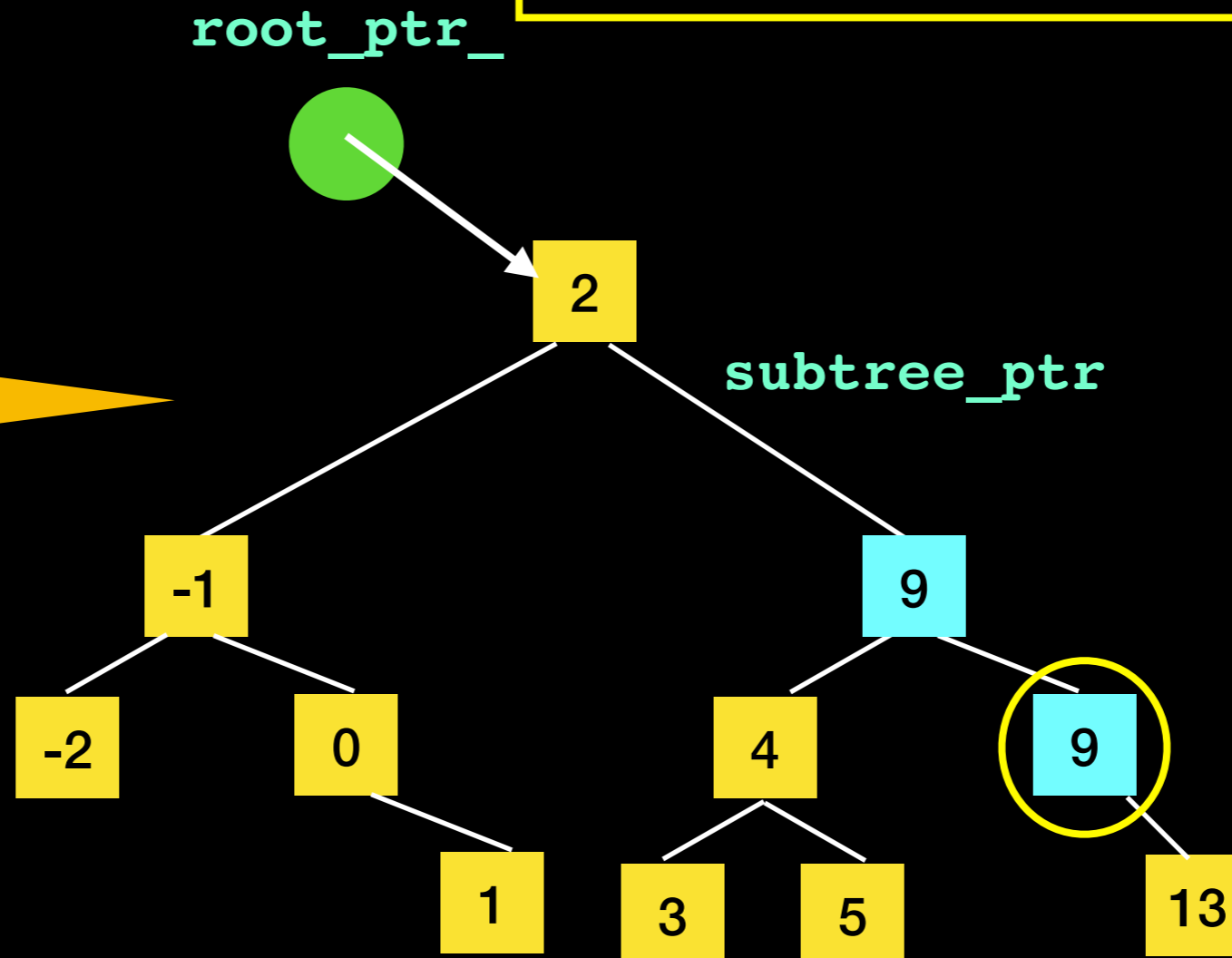
removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
{ //Item is 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.



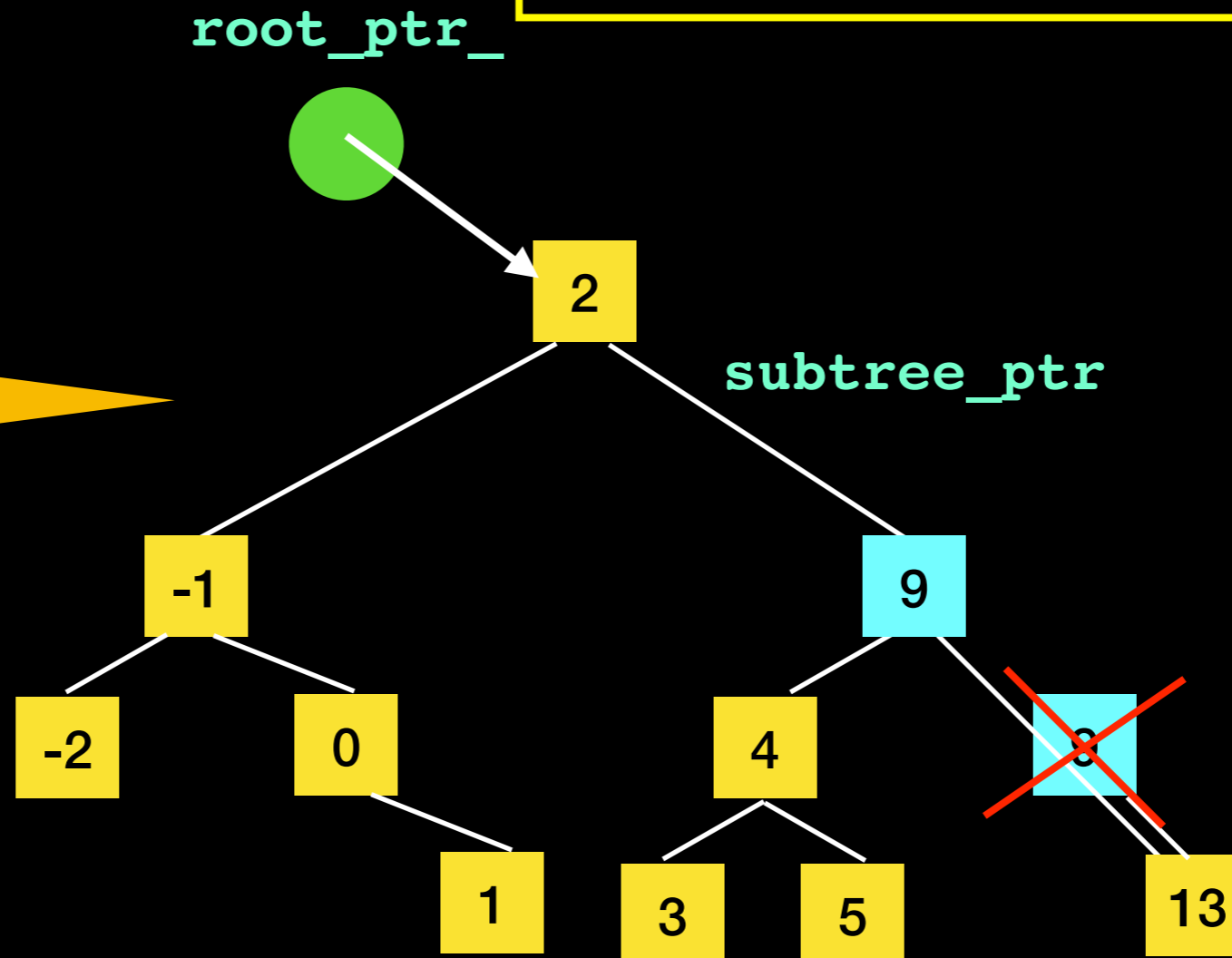
```
removeNode(subtree_ptr);
```

```
if (subtree_ptr->getItem() == target)
{ //Item is 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.



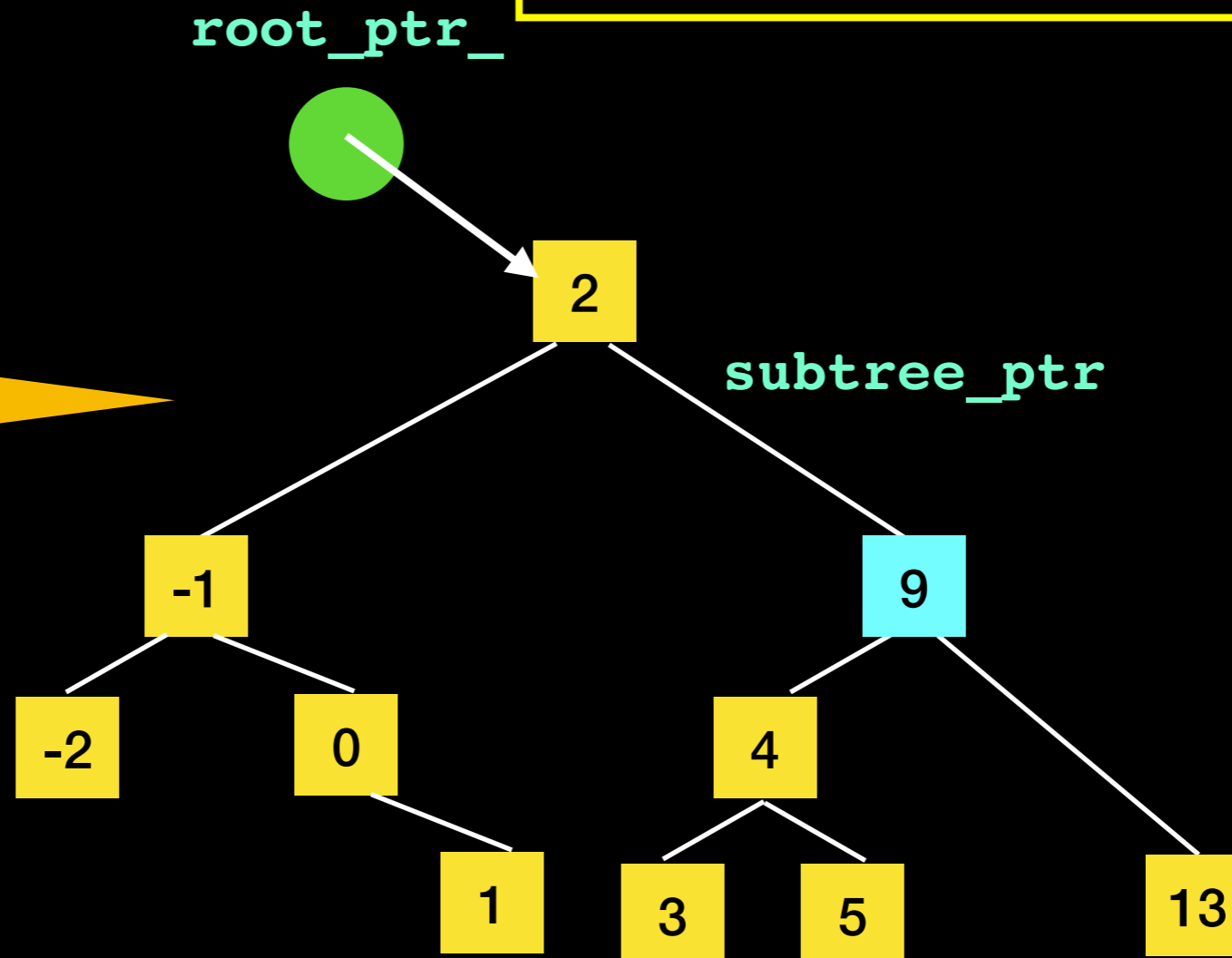
removeNode(subtree_ptr);

```
if (subtree_ptr->getItem() == target)
{ //Item is 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);

```
using namespace std;
template<typename ItemType>
auto BST<ItemType>::removeNode(std::shared_ptr<BinaryNode<ItemType>>
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
  {
    ItemType 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 leaf node

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

removeLeftmostNode

```
using namespace std;

template<typename ItemType>
auto BST<ItemType>::removeLeftmostNode(shared_ptr<BinaryNode<ItemType>>
                                       nodePtr, ItemType& 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<typename ItemType>
void BST<ItemType>::preorderTraverse(Visitor<ItemType>& visit) const
{
    preorder(visit, root_ptr_);
} // end preorderTraverse
```

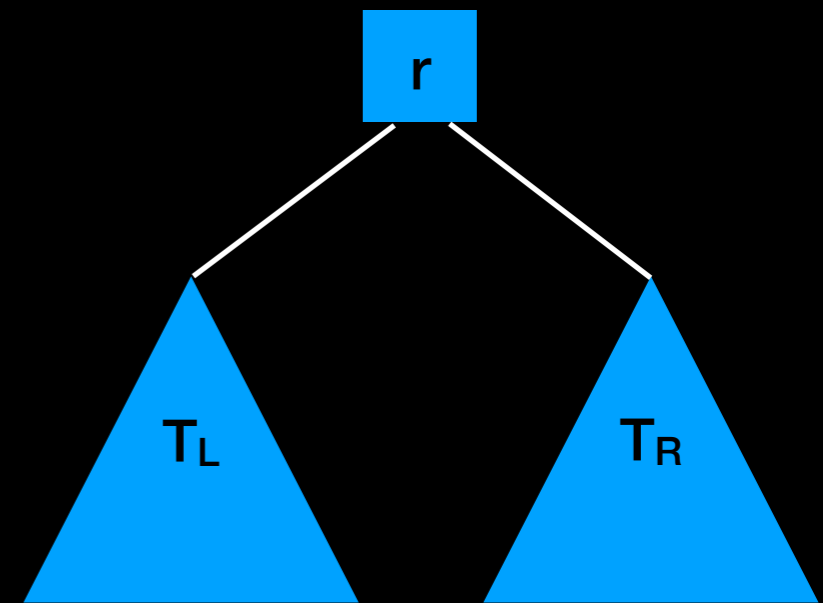
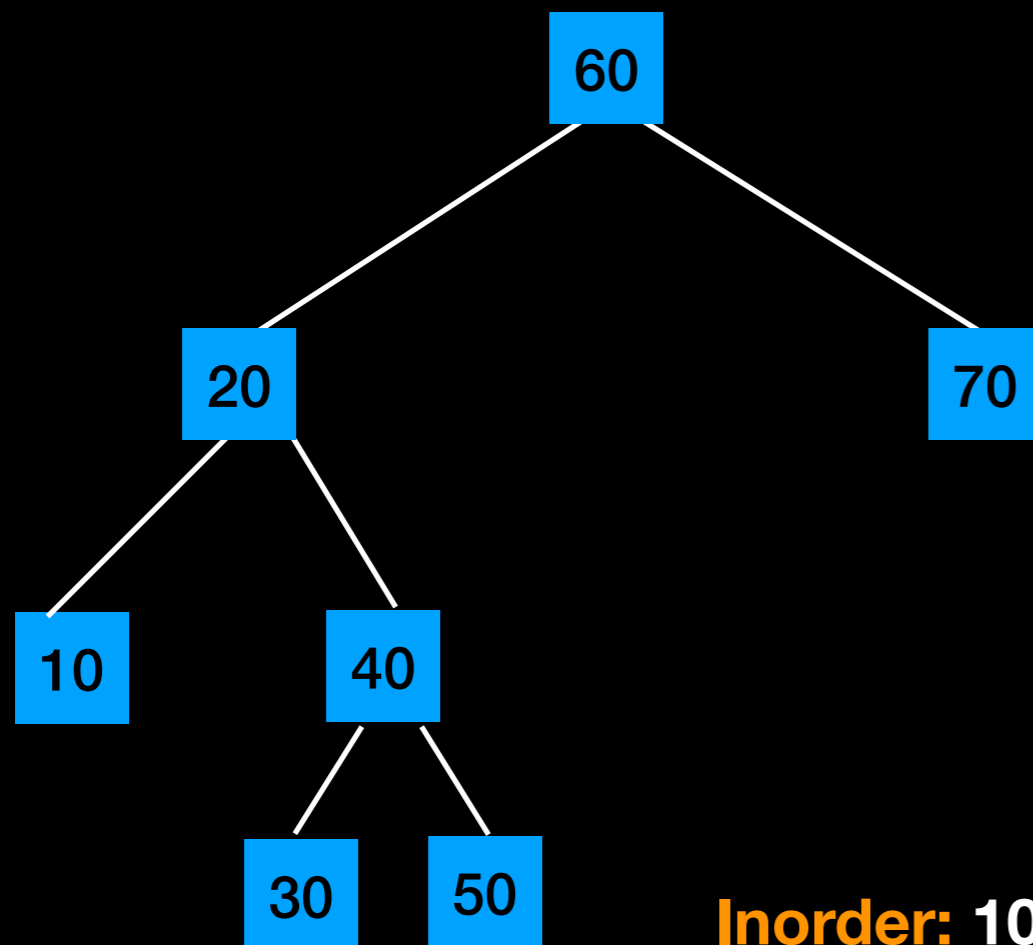
```
template<class ItemType>
void BST<ItemType>::inorderTraverse(Visitor<ItemType>& visit) const
{
    inorder(visit, root_ptr_);
} // end inorderTraverse
```

```
template<class ItemType>
void BST<ItemType>::postorderTraverse(Visitor<ItemType>& 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 TL
  visit the root r
  traverse TR
}
```



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

inorderTraverse Helper Function

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

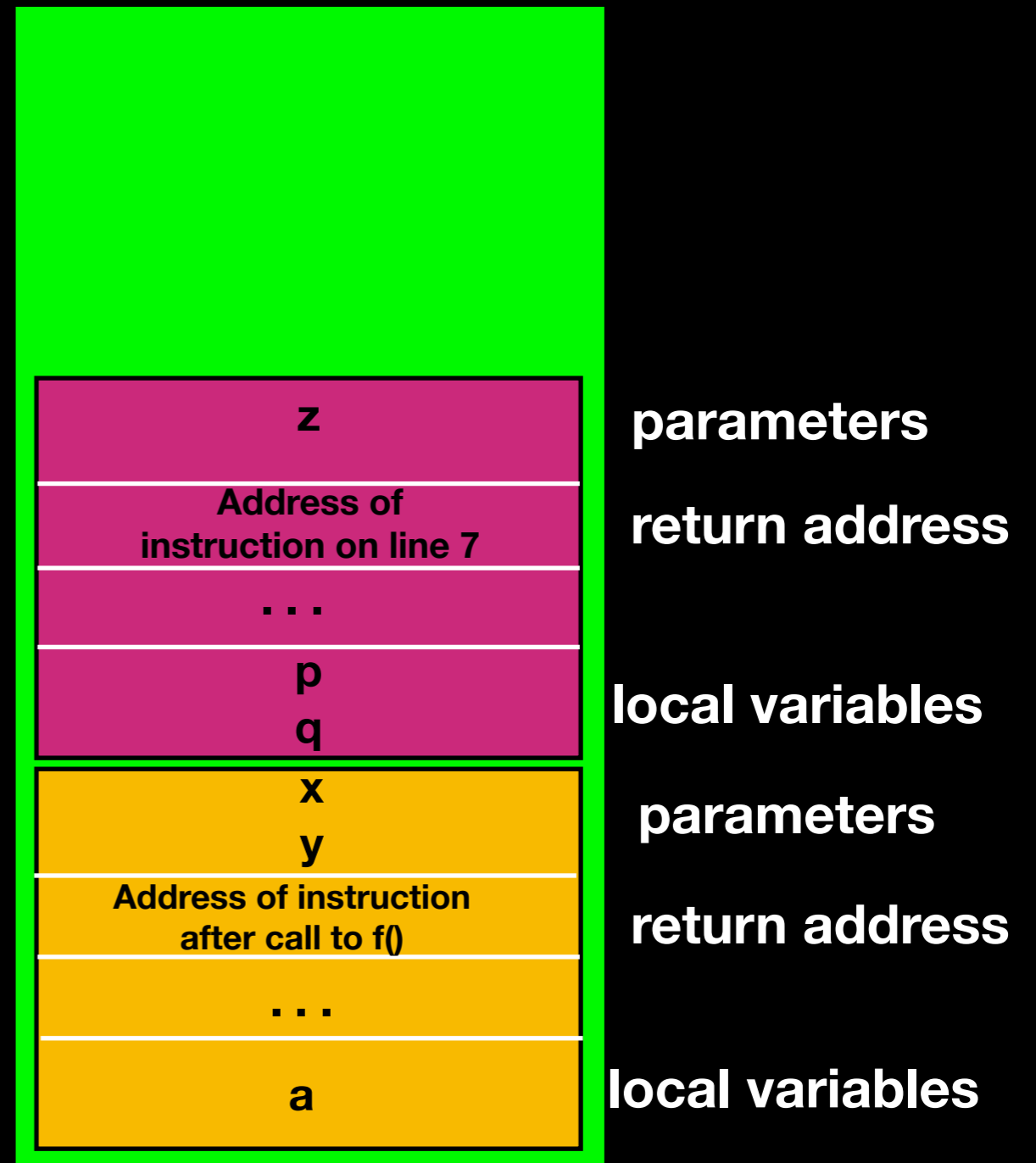
Recall: Program Stack

```
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 }
```

**Stack Frame
for g()**

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

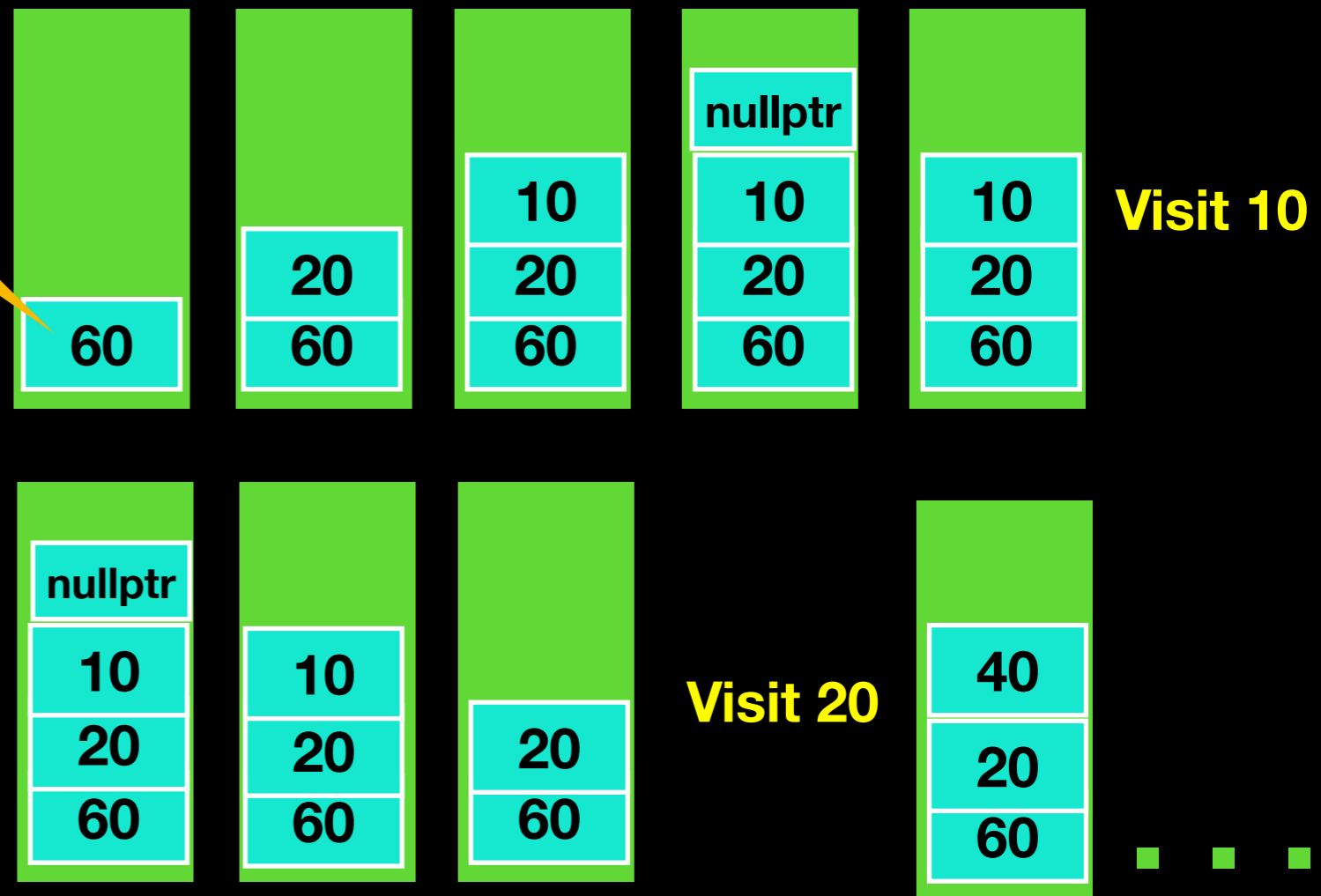
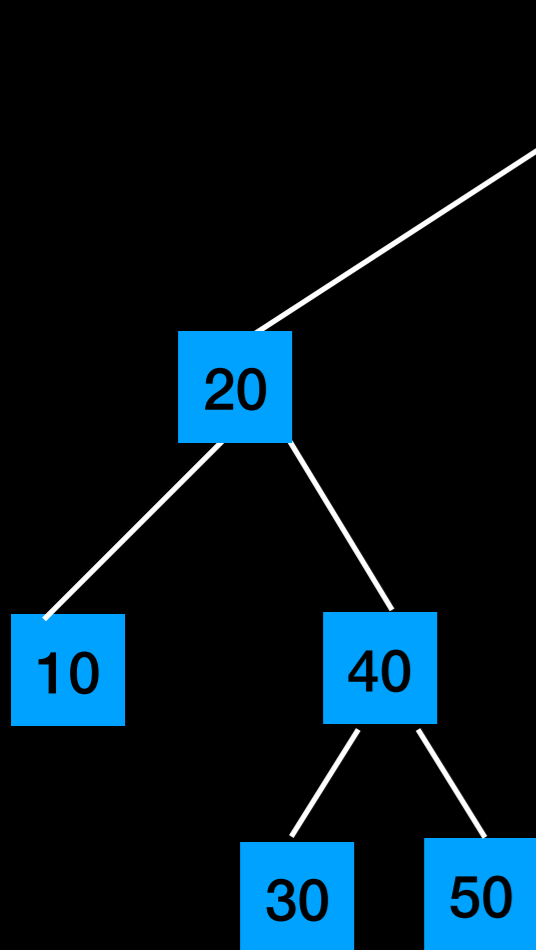
**Stack Frame
for f()**



Recursive Traversal

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

Means function call with tree_ptr pointing to node with data item 60



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

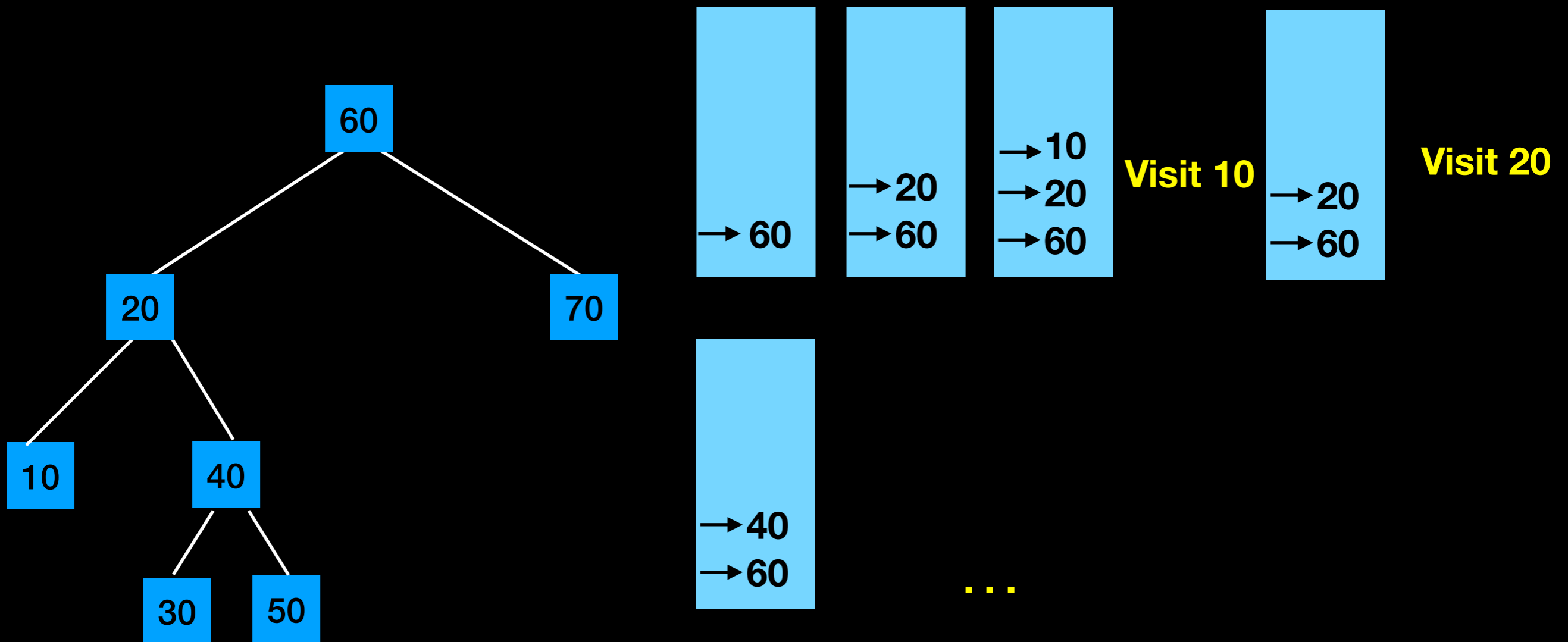
Optimize: Implement iterative approach that maintains an **explicit stack** to keep track of nodes that must be visited

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

```
using namespace std;
template<typename ItemType>
void BST<ItemType>::inorder(Visitor<ItemType>& visit) const
{
    stack<ItemType> node_stack;
    shared_ptr<BinaryNode<ItemType>> 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();
        }
    }
}
```

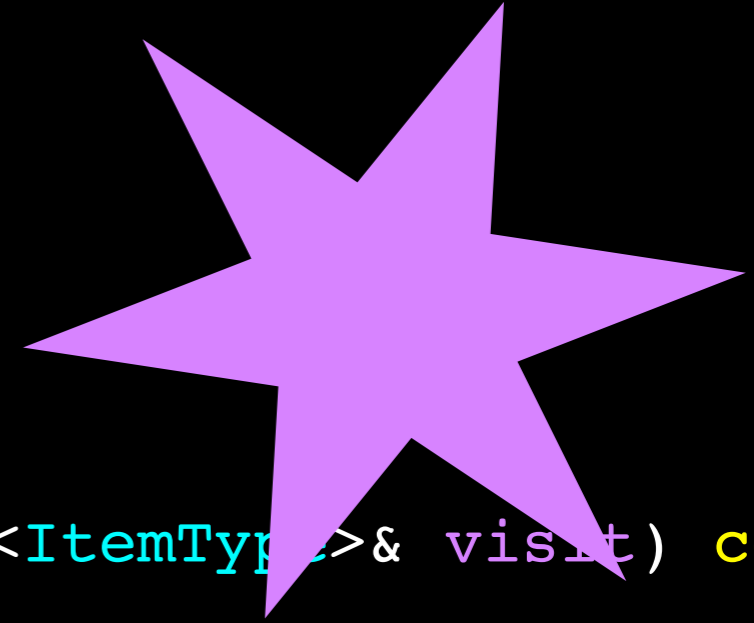
Non-recursive Traversal cont.

```
//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
```

The last cool trick for
you this semester

Traversals



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

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

```
template<class ItemType>
void BST<ItemType>::postorderTraverse(Visitor<ItemType>& 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<typename ItemType>
class Visitor
{
public:

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

};

#endif /* Visitor_hpp */
```

```
#ifndef StringPrinter_hpp
#define StringPrinter_hpp

#include "Visitor.hpp"
#include <iostream>
#include <string>
using namespace std;

class StringPrinter: public Visitor<string>
{
public:
    void operator()(string&) override;
    void operator()(string&, string&) override;

};

#endif /* StringPrinter_hpp */
```



```
#include "StringPrinter.hpp"
using namespace std;

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

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

```

#ifndef Inverter_hpp
#define Inverter_hpp

#include "Visitor.hpp"
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

class Inverter: public Visitor<string>
{
public:

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

};

#endif /* Inverter_hpp */

```

```
#include "Inverter.hpp"

using namespace std;

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

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

Traversal with Functor parameter

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

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

```

using namespace std;
int main() {

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

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

    StringPrinter p;
    Inverter i;

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

    return 0;
}

```

root_ptr_



"a string"

"o string"

```

a string
o string

gnirts a
gnirts o
Program ended with exit code: 0

```

```

using namespace std;
int main() {

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

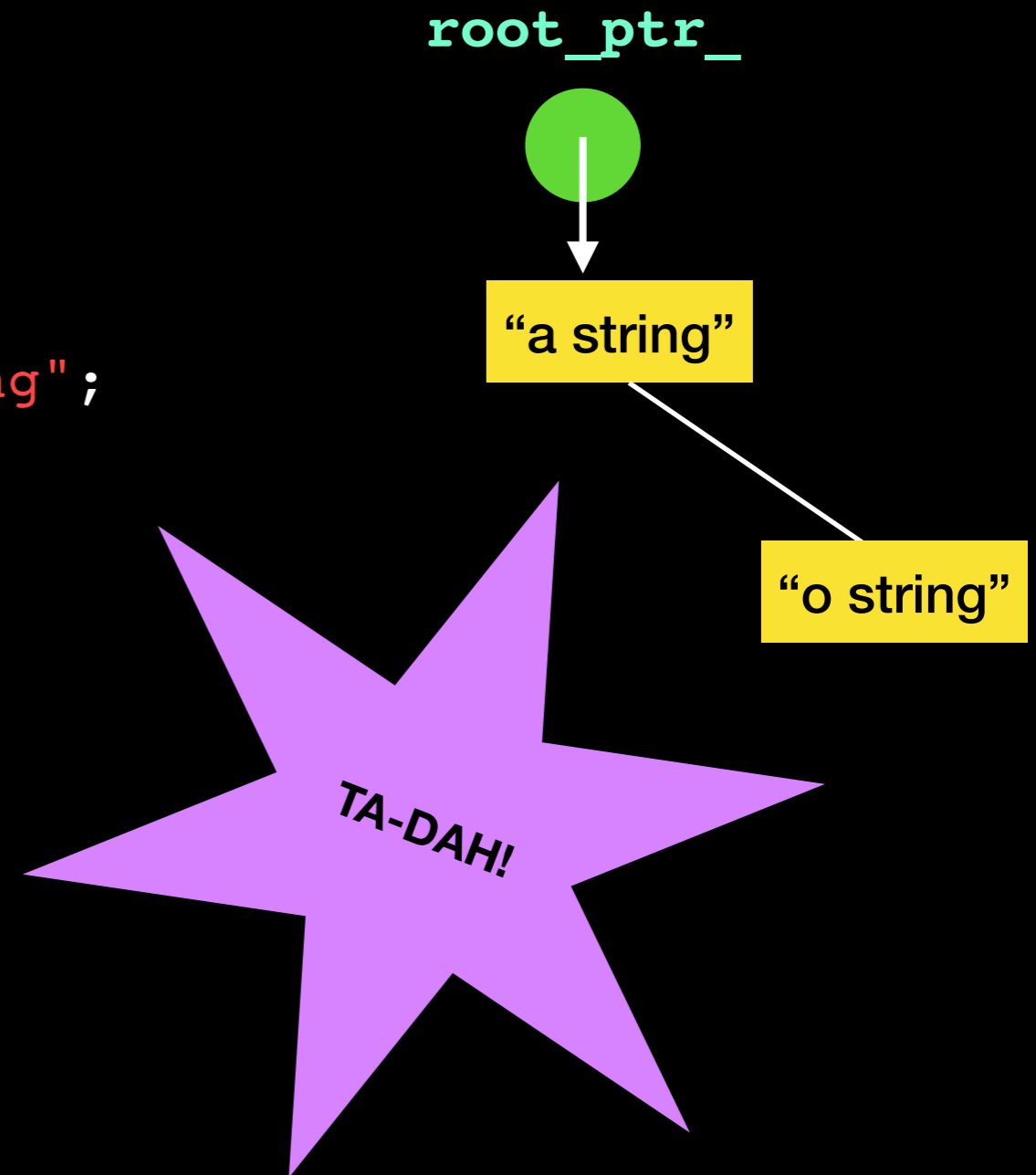
    BST<string> a_tree(a_string);
    a_tree.add(anoter_string);

    StringPrinter p;
    Inverter i;

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

    return 0;
}

```



```

a string
o string

gnirts a
gnirts o
Program ended with exit code: 0

```