Trees

Slide Set: 9

Learning Objectives

- Follow and explain tree-based algorithms
- Design and implement classes for **binary tree** nodes and nodes for **general trees**
- **Tree traversal** algorithms
- The rules for implementing a **binary search tree**
- **Implementation of a data structure using a binary search tree**

Introduction to Trees

Binary Tree

- A **binary tree** is a finite set of nodes
- The set might be empty (the empty tree)
- If the set is not empty, **it follows these rules:**
  - There is one special node, called the **root**
  - Each node may be associated with up to two other different nodes, called its **left child** and its **right child**
  - If a node c is the child of another node p, then we say that “p is c’s **parent**”
  - Each node, except the root, has exactly one parent; the root has no parent
  - If you start at a node and move to the node’s parent, then move again to that node’s parent, and keep moving upward to each node’s parent, you will eventually reach the root

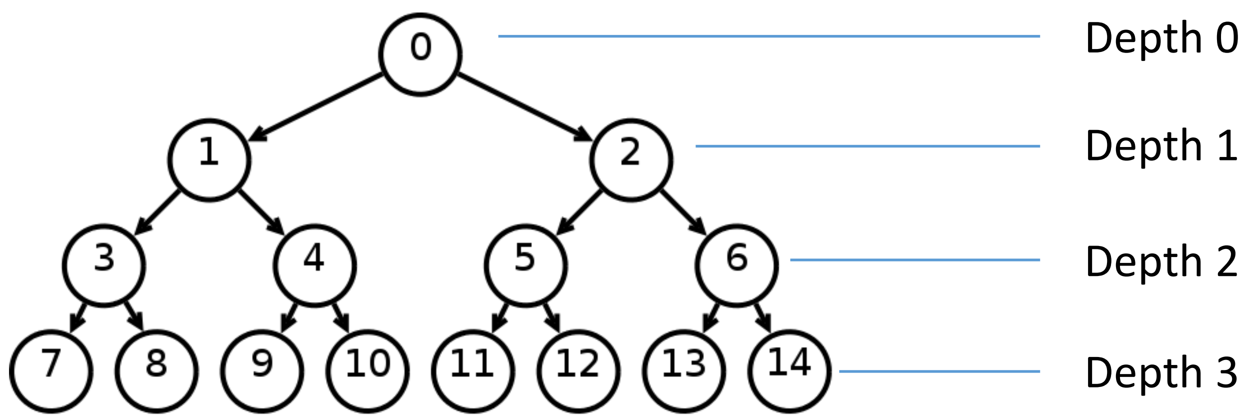
Binary Tree

- **Parent**: The parent of a node is the node linked above it
- **Sibling:** Two nodes are siblings if they have the same parent
- **Ancestor:** A node's parent is its first ancestor. The parent of the parent is the next ancestor. The parent of the parent of the parent is the next ancestor . . . and so forth, until you reach the root
- **Descendant**: A node's children are its first descendants. The children's children are its next descendants,…
- **Subtree**: Any node in a tree also can be viewed as the root of a new, smaller tree
  - This view enables us to implement recursive tree algorithms
- **Left and right subtrees of a node**: For a node in a binary tree, the nodes beginning with its left child and below are its left subtree; The nodes beginning with its right child and below are its right subtree

Introduction

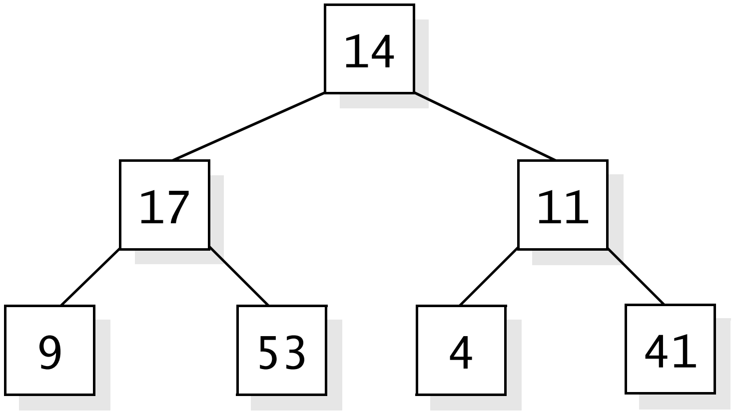
Binary Tree

- In the following tree, for example:
  - 3 is **parent** of 7
  - 7 and 8 are **siblings**
  - 3, 1, and 0 are the **ancestors** of 7
  - 7 is a **descendant** of 3
  - **Depth of node** 7 is three
    - (Note: Depth of a tree with only root is 0, depth of an empty tree is -1)
  - **Depth of the tree** is three



Full Binary Tree:

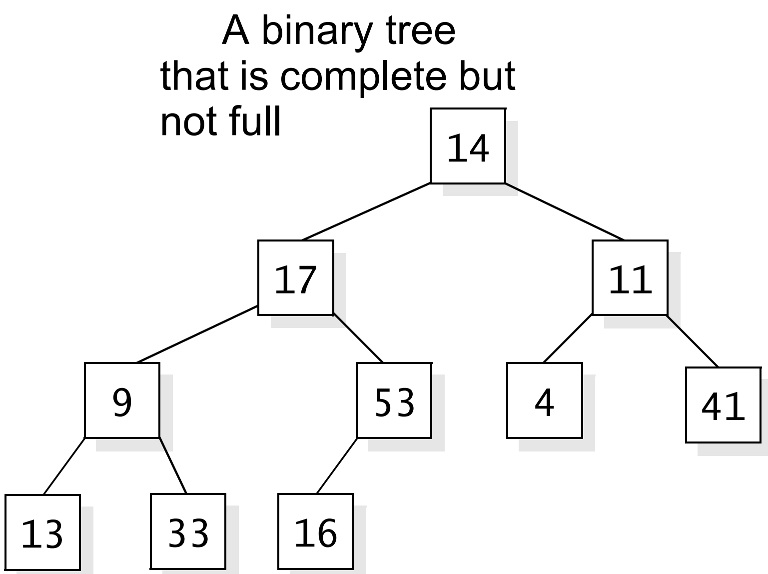
- A full binary tree (sometimes **proper binary tree** or **2-tree**) is a tree in which every node other than the leaves has two children

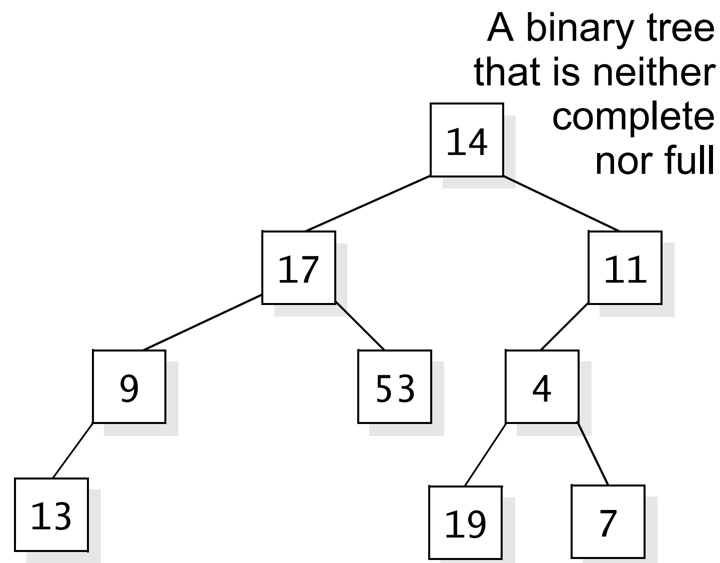


Binary Tree

Complete Binary Tree:

- A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

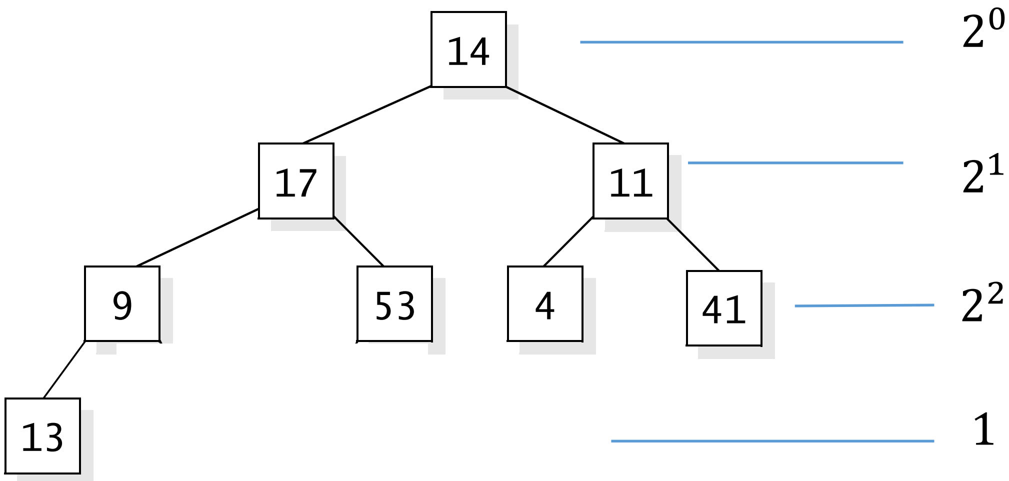




Introduction

Binary Tree

The minimum number of entries in a complete binary tree with depth *n* is 2^n

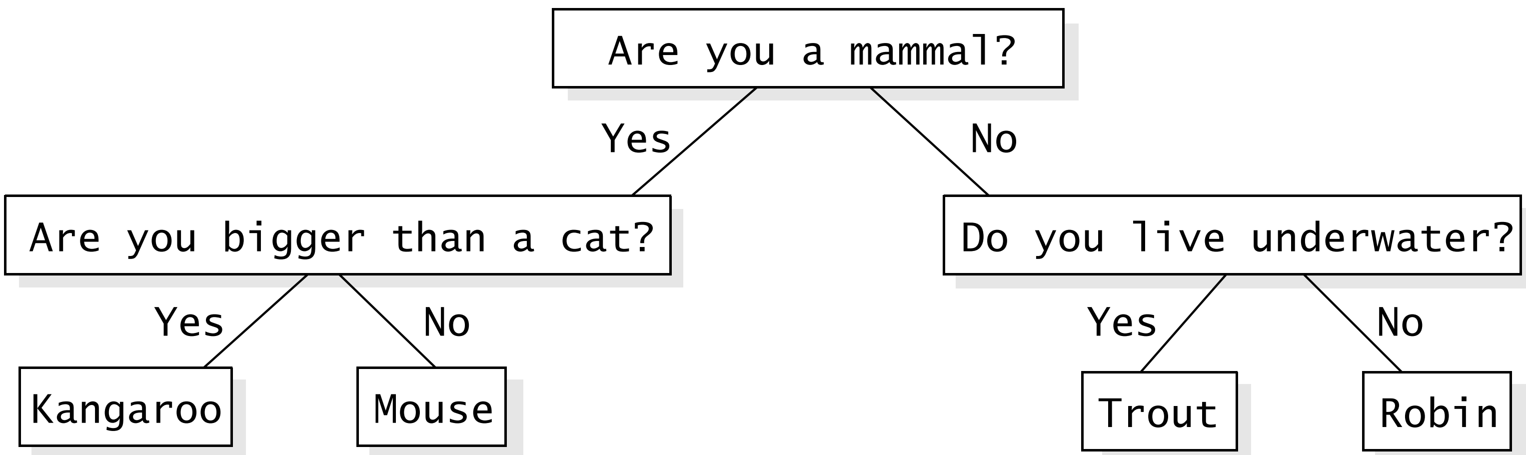


$$2^0 + 2^1 + 2^2 + 1 = (2^3 - 1) + 1 = 2^3$$

Binary Tree

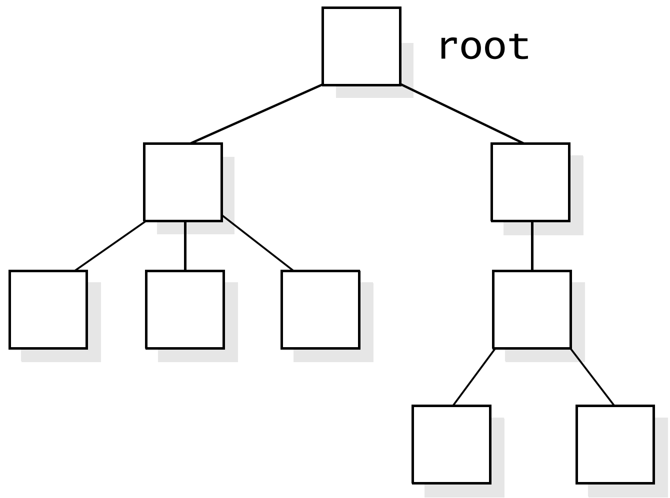
Binary Taxonomy Tree:

- You start at the root, and ask the question that is written there
- If the answer is 'yes', you move to the left child
- If the answer is 'no', you move to the right child



General Trees

General Trees:
- There is one special node called the root
- Each node may be associated with one or more nodes, called its children
- Each node, except the root, has exactly one parent
- Moving towards a node’s parent, and the parent of the parent, and…, will eventually reach the root



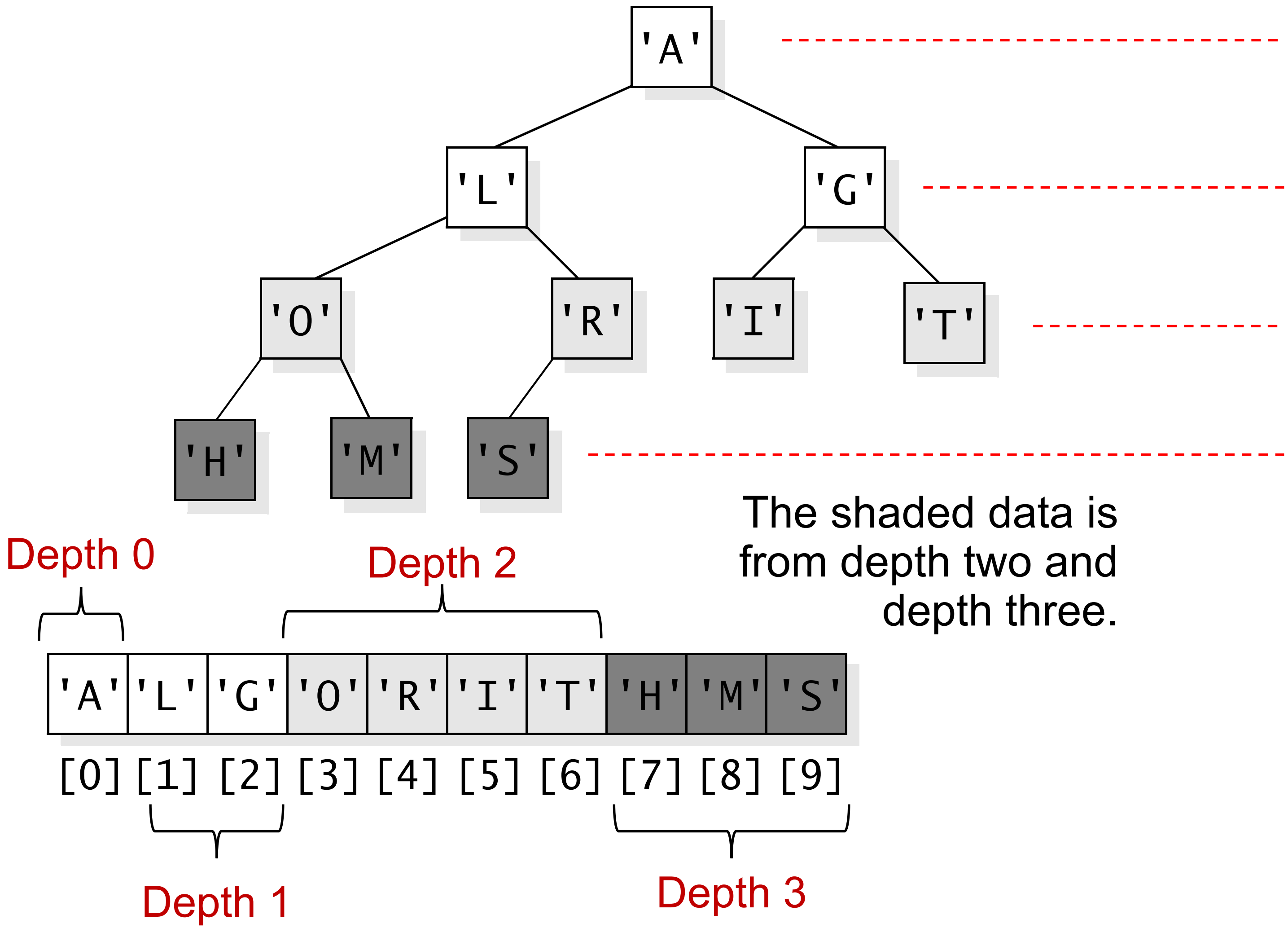
Tree Representation

Array Representation of Complete Binary Trees

- In a **complete binary tree**:
  - All of the depths are full, except perhaps for the deepest
  - At the deepest depth, the nodes are as far left as possible

- A simple representation using **arrays**
- The representation can use:
  - A **fixed-sized array** which means that the size of the data structure is fixed during compilation, and during execution it does not grow larger or smaller
  - A **dynamic array** allowing the representation to grow and shrink as needed during the execution of a program

Array Representation of Complete Binary Trees



The shaded data is from depth two and depth three.

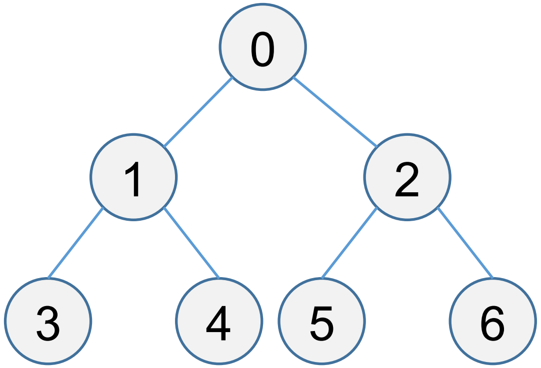
Array Representation of Complete Binary Trees

Reasons why the array representation is convenient:

- The data from the root always appears in the [0] component of the array
- Suppose that the data for a non-root node appears in component [i] of the array:
  - The data for its parent is always at location [(i-1)/2] (using integer division)
- Suppose that the data for a node appears in component [i] of the array, then its children (if they exist) always have their data at these locations:
  - Left child at component [2i+1]
  - Right child at component [2i+2]

Array Representation of Complete Binary Trees

- Parent $(i) = \lfloor (i-1)/2 \rfloor$
- Left child $(i) = 2i + 1$
- Right child $(i) = 2i + 2$
- Left sibling $(i) = i - 1$ if i is even
- Right sibling $(i) = i + 1$ if i is odd and $i+1 <= n$

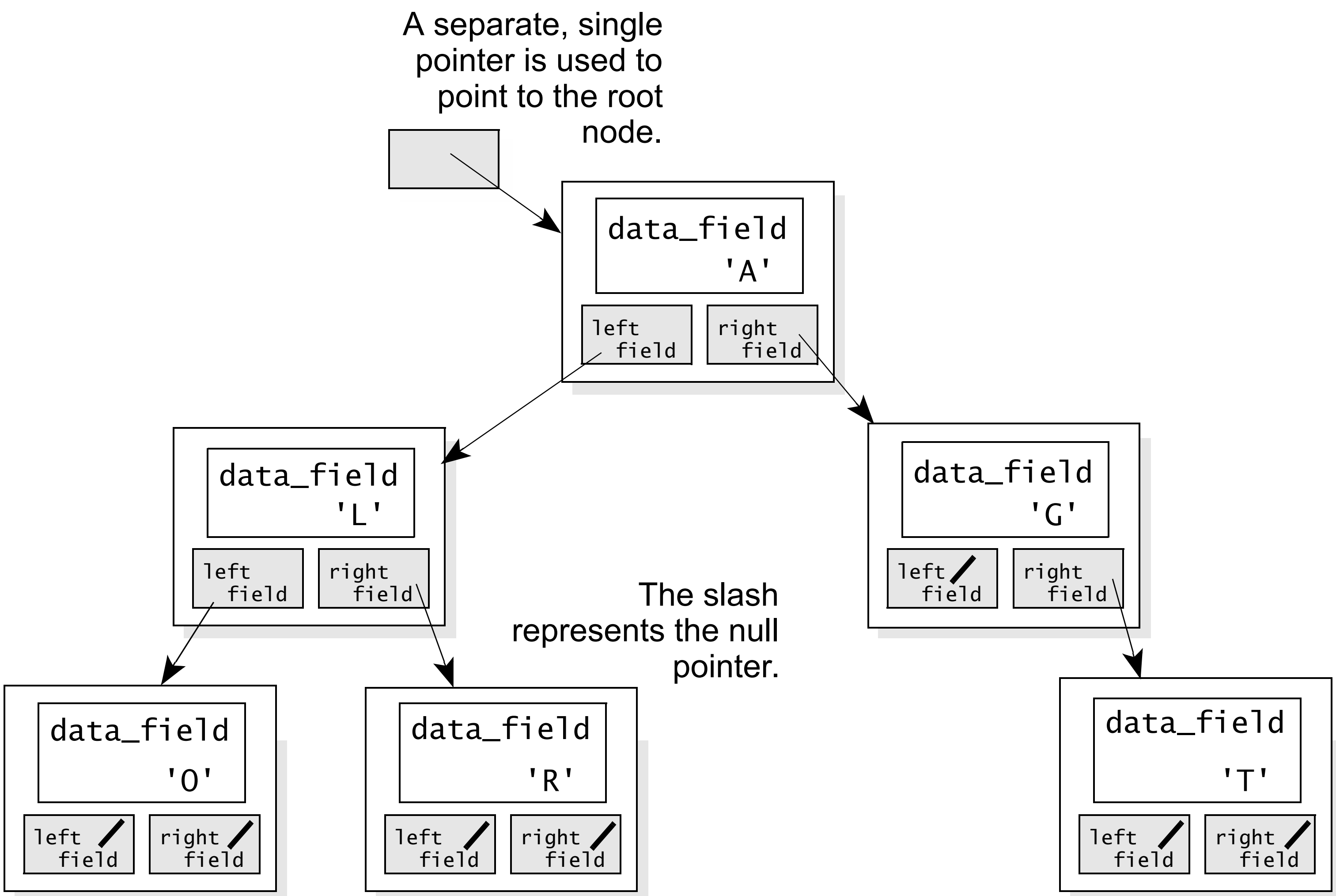


Representing a Binary Tree with a Class for Nodes

- **Each node is stored in an object of a new `binary_tree_node`**
- Each node contains pointers that link it to other nodes
- An entire tree is represented as a pointer to the root node

```
template <class Item>
class binary_tree_node
{

    private:
            Item data_field;
            binary_tree_node *left_field;
            binary_tree_node *right_field;
};
```

Representing a Binary Tree with a Class for Nodes



Representing a Binary Tree with a Class for Nodes

Header file for the `binary_tree_node`

```
#ifndef BINTREE_H
#define BINTREE_H
#include <cstdlib>    // Provides NULL and size_t

namespace scu_coen79_10
{
    template <class Item>
    class binary_tree_node
    {
     public:

       // TYPEDEF
       typedef Item value_type;

       // CONSTRUCTOR
       binary_tree_node( const Item& init_data = Item( ),
                              binary_tree_node* init_left = NULL,
                              binary_tree_node* init_right = NULL   )
```

Representing a Binary Tree with a Class for Nodes

```
// MODIFICATION MEMBER FUNCTIONS
Item& data( ) { return data_field; }
binary_tree_node* left( ) { return left_field; }
binary_tree_node* right( ) { return right_field; }


void set_data(const Item& new_data) { data_field = new_data; }
void set_left(binary_tree_node* new_left) { left_field = new_left;}
void set_right(binary_tree_node* new_right)
                                   { right_field = new_right; }

// CONST MEMBER FUNCTIONS
const Item& data( ) const { return data_field; }
const binary_tree_node* left( ) const { return left_field; }
const binary_tree_node* right( ) const { return right_field; }


bool is_leaf( ) const
    { return (left_field == NULL) && (right_field == NULL); }
```

Representing a Binary Tree with a Class for Nodes

```
private:
    Item data_field;
    binary_tree_node *left_field;
    binary_tree_node *right_field;
};


// NON-MEMBER FUNCTIONS for the binary_tree_node<Item>:
template <class Process, class BTNode>
void inorder(Process f, BTNode* node_ptr);


template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr);


template <class Process, class BTNode>
void postorder(Process f, BTNode* node_ptr);


template <class Item, class SizeType>
void print(binary_tree_node<Item>* node_ptr, SizeType depth);
```

Traversing a tree (Implementation will be explained later)

Representing a Binary Tree with a Class for Nodes

Returning nodes to the heap
(Implementation will be explained)

```
template <class Item>
void tree_clear(binary_tree_node<Item>*& root_ptr);
```

Copying a tree
(Implementation will be explained)

```
template <class Item>
binary_tree_node<Item>* tree_copy(
                                    const binary_tree_node<Item>* root_ptr);


template <class Item>
std::size_t tree_size(const binary_tree_node<Item>* node_ptr);
}

#include "bintree.template"
#endif
```

Representing a Binary Tree with a Class for Nodes: Returning Nodes to the Heap

```
template <class Item>
void tree_clear(binary_tree_node<Item>*& root_ptr);
// Precondition: root_ptr is the root pointer of a binary tree (which
// may be NULL for the empty tree).
// Postcondition: All nodes at the root or below have been returned
// to the heap, and root_ptr has been set to NULL.
```

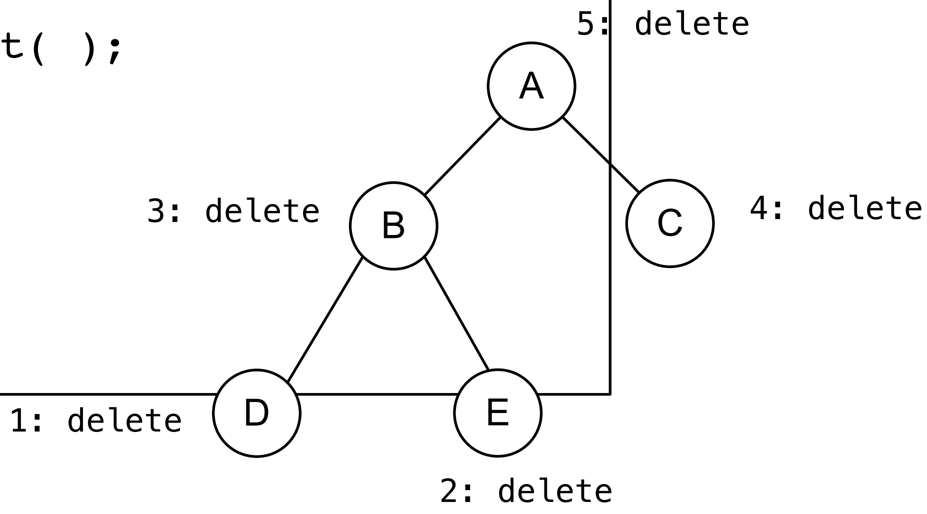
It is a non-member function, why?

Implementation through a recursive algorithm:

1. Clear the left subtree
2. Clear the right subtree
3. Return the root node to the heap
4. Set the root pointer to NULL

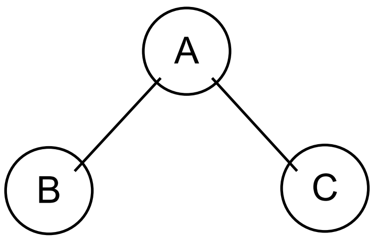
Representing a Binary Tree with a Class for Nodes: Returning Nodes to the Heap

```
template <class Item>
void tree_clear(binary_tree_node<Item>*& root_ptr)
// Library facilities used: cstdlib
{
    binary_tree_node<Item>* child;
    if (root_ptr != NULL)
    {
        child = root_ptr->left( );
        tree_clear( child );

        child = root_ptr->right( );
        tree_clear( child );

        delete root_ptr;
        root_ptr = NULL;
    }
}
```



Representing a Binary Tree with a Class for Nodes: Returning Nodes to the Heap

- **Exercise:**
How many times the `tree_clear` function is invoked when we delete the following tree?



Tree Representation

Representing a Binary Tree with a Class for Nodes: Copying a Tree

```
template <class Item>
binary_tree_node<Item>* tree_copy(
                                const binary_tree_node<Item>* root_ptr);
// Precondition: root_ptr is the root pointer of a binary tree (which
// may be NULL for the empty tree).
// Postcondition: A copy of the binary tree has been made, and the
// return value is a pointer to the root of this copy.
```

Through a recursive algorithm:

1. Make `l_ptr` point to a copy pf the left subtree
2. Make `r_ptr` point to a copy pf the right subtree
3. `return new binary_tree_node (root_ptr->data(), l_ptr, r_ptr)`

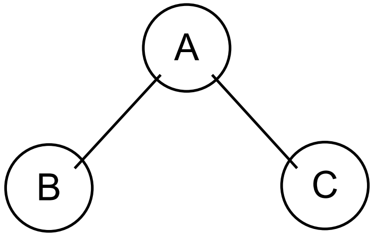
Representing a Binary Tree with a Class for Nodes: Copying a Tree

```
template <class Item>
binary_tree_node<Item>* tree_copy(
                      const binary_tree_node<Item>* root_ptr)
{
        binary_tree_node<Item> *l_ptr;
        binary_tree_node<Item> *r_ptr;

        if (root_ptr == NULL)
            return NULL;
        else
        {
            l_ptr = tree_copy( root_ptr->left( ) );
            r_ptr = tree_copy( root_ptr->right( ) );
            return new binary_tree_node<Item>(
                                 root_ptr->data( ), l_ptr, r_ptr);
        }

}
```

Representing a Binary Tree with a Class for Nodes: Copying a Tree

- **Exercise:**

How many times the `tree_copy` function is invoked when we make a copy of the following tree?



Tree Traversals

Tree Traversals

- Tree traversal: Processing all the nodes in a tree
- For a binary tree, there are three common ways of traversal:
  - **pre-order traversal**
  - **in-order traversal**
  - **post-order traversal**

Pre-Order Traversal

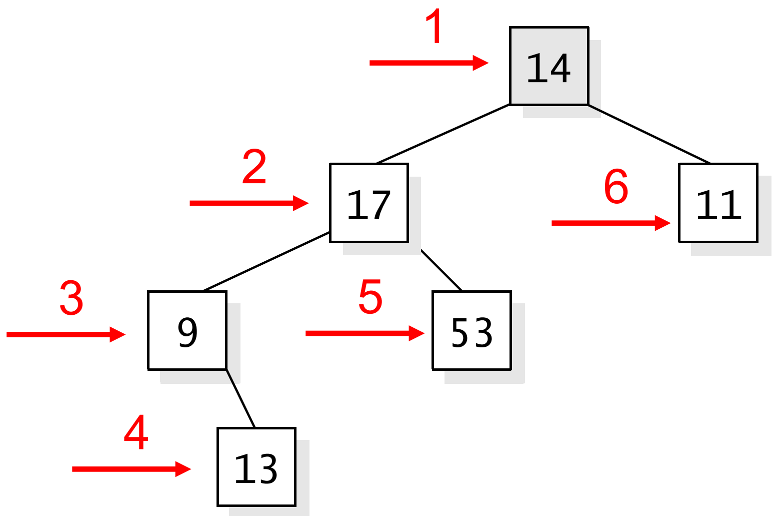
Pre-Order Traversal

1. Process the root
2. Process the nodes in the left subtree with a recursive call
3. Process the nodes in the right subtree with a recursive call

```
template <class Item>
void preorder_print(const binary_tree_node<Item>* node_ptr)
// Precondition: node_ptr is a pointer to a node in a binary tree
// (or node_ptr may be NULL to indicate the empty tree).
// Postcondition: If node_ptr is non-NULL, then the data of
// *node_ptr and all its descendants have been written to cout with
// the << operator, using a pre-order traversal.
{
    if (node_ptr != NULL)
    {
        std::cout << node_ptr->data( ) << std::endl;
        preorder_print( node_ptr->left( ) );
        preorder_print( node_ptr->right( ) );
    }
}
```

Tree Traversals

Pre-Order Traversal



14 17 9 13 53 11

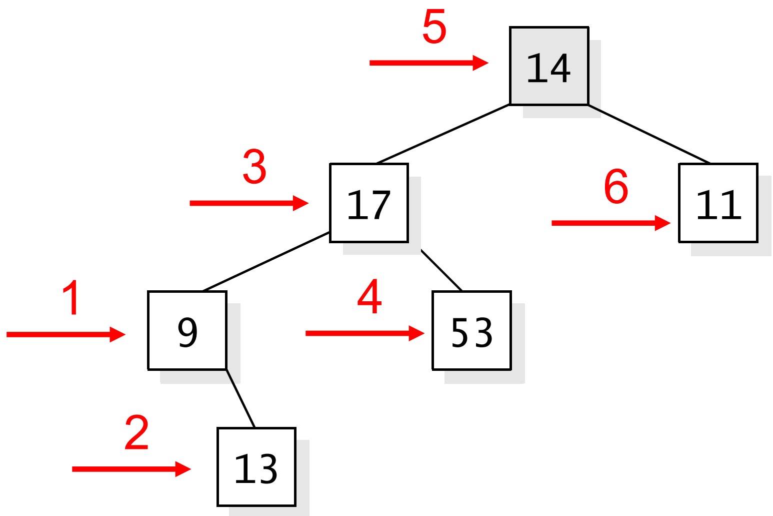
In-Order Traversal

In-Order Traversal
1. Process the nodes in the left subtree with a recursive call
2. Process the root
3. Process the nodes in the right subtree with a recursive call

```
template <class Item>
void inorder_print(const binary_tree_node<Item>* node_ptr)
{
    if (node_ptr != NULL)
    {
        inorder_print ( node_ptr->left( ) );
        std::cout << node_ptr->data( ) << std::endl;
        inorder_print ( node_ptr->right( ) );
    }
}
```

Note: Another type of in-order traversal:
Backward In-order Traversal
1. Process the nodes in the right subtree with a recursive call
2. Process the root
3. Process the nodes in the left subtree with a recursive call

In-Order Traversal



9 13 17 53 14 11

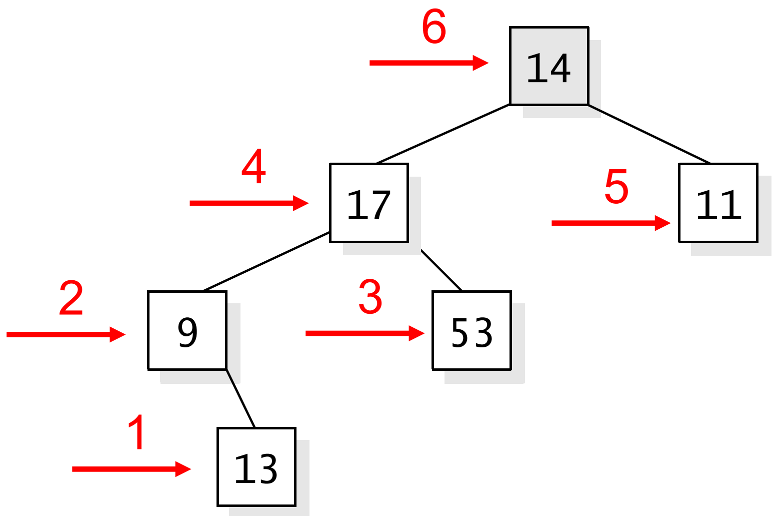
Post-Order Traversal

Post-Order Traversal

1. Process the nodes in the left subtree with a recursive call
2. Process the nodes in the right subtree with a recursive call
3. Process the root

```
template <class Item>
void postorder_print(const binary_tree_node<Item>* node_ptr)
{
    if (node_ptr != NULL)
    {
        postorder_print ( node_ptr->left( ) );
        postorder_print ( node_ptr->right( ) );
        std::cout << node_ptr->data( ) << std::endl;
    }
}
```

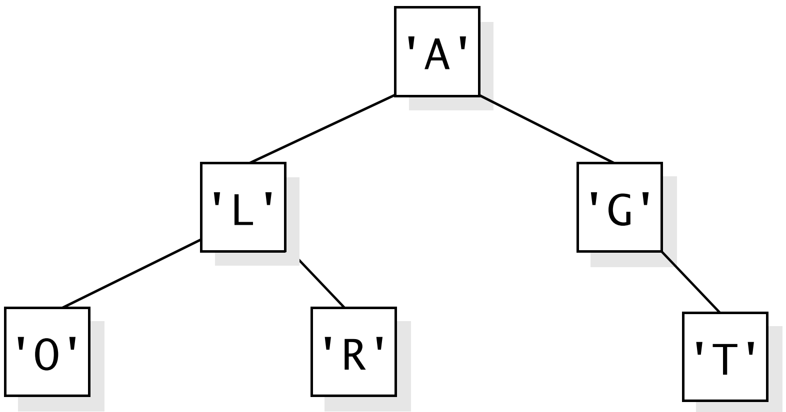
Post-Order Traversal



13 9 53 17 11 14

Tree Traversals

❑ Example



Pre-order: A L O R G T
In-order: O L R A G T
Post-order: O R L T G A
Backward in-order traversal: T G A R L O

Tree Traversals

Parameters can be a Function

- In general, we would like to be able to do any kind of processing during tree traversal — not just printing
- **We can just replace the `cout` statement in the traversal function with some other form of processing**
  - This is very inefficient: We need to develop a new function for each type of processing
- **It is possible to write just one function that is capable of doing a tree traversal and carrying out virtually any kind of processing at the nodes**

Parameters can be a Function

- Example: A function called `apply`, with three arguments:
  - A *void* function `f`
  - An array of integers called `data`
  - A `size_t` value called n, indicating the number of components in the array

```
void apply(void f(int&), int data[ ], size_t n);
```

The power of the `apply` function comes from the fact that **its first argument can be any *void* function with a single integer reference parameter**

Parameters can be a Function

```
void apply(void f(int&), int data[ ], size_t n)
{
    size_t i;
    for (i = 0; i < n; ++i)
        f(data[i]);
}
```

- Example

```
apply(seven_up,...
```

```
void seven_up(int& i)
// Postcondition: i has had 7 added to its value.
{
    i += 7;
}
```

```
apply(triple,...
```

```
void triple(int& i);
// Postcondition: i has been increased by a factor of 3.
```

Tree Traversals

Parameters can be a Function

Obtaining more generality: The component type of the array is specified by the template parameter

```
template <class Item, class SizeType>
void apply(void f(Item&), Item data[ ], SizeType n)
{
    size_t i;
    for (i = 0; i < n; ++i)
        f(data[i]);
}
```

```
void convert_to_upper(char& c);
//  Postcondition: If c was a lowercase letter, then it has been converted to
//  the corresponding uppercase letter; otherwise c is unchanged.
```

```
apply(convert_to_upper,...
```

Tree Traversals

Parameters can be a Function

- Suppose that `name` is an array of 10 characters
- We can convert all these characters to uppercase with a single call to the `apply` template function:

```
apply(convert_to_upper, name, 10);
```

```
void convert_to_upper(char& c);
//  Postcondition: If c was a lowercase letter, then it has been converted to
//  the corresponding uppercase letter; otherwise c is unchanged.
```

Parameters can be a Function

- Currently, the first argument to the apply function must have the form:
  ```
  void f(Item&);
  ```
  - The return type is void, and the parameter type is a reference to Item
- Precludes many functions that we might want to use
- For example, `f` cannot have a value parameter (it must have a *reference* parameter)

Obtaining more generality:

- **We add the third template parameter**
- **The component type of the first argument is specified by a template parameter**

Tree Traversals

Parameters can be a Function

```
template <class Process, class Item, class SizeType>
void apply(Process f, Item data[ ], SizeType n)
{
    size_t i;
    for (i = 0; i < n; ++i)
        f(data[i]);
}
```

- Sample functions that can be used with the `apply` function:

```
void triple(int& i); // Postcondition: i has been multiplied by three.
void print(int i);   // Postcondition: i has been printed to cout.
void print(const string& s); // Postcondition: s has printed to cout.
```

A sample code that passes function `printValue` to the apply function

```
int main()
{
    int array1[] = {0, 1, 2, 3, 4};
    apply(printValue, array2, 4);
    return 0;
}
```

Template Functions for Tree Traversals

This template function will apply a function f to all the items in a binary tree, using a pre-order traversal:

```
template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr)
// Precondition: node_ptr is a pointer to a node in a binary tree (or
// node_ptr may be NULL to indicate the empty tree).
// Postcondition: If node_ptr is non-NULL, then the function f has been
// applied to the contents of *node_ptr and all of its descendants, using a
// pre-order traversal.
// Note: BTNode may be a binary_tree_node or a const binary tree node.
// Process is the type of a function f that may be called with a single
// Item argument (using the Item type from the node).
{
    if (node_ptr != NULL) {
         f( node_ptr->data( ) );
         preorder(f, node_ptr->left( ) );
         preorder(f, node_ptr->right( ) );
    }
}
```

We don't even need to know exactly what f does

Binary Search Trees

Properties of Binary Search Trees

- Binary trees offer an improved way of **implementing the bag class**
- **This implementation requires that the bag's entries can be compared with the usual comparison operators <, >, ==, and so on**
- These operators must form a strict weak ordering
- **Take advantage of the order to store the items in the nodes of a binary tree, using a strategy that will make it easy to find items**

Reminder:
A Strict Weak Ordering has to behave the way that "less than" behaves: if 'a' is less than 'b' then 'b' is not less than 'a', if 'a' is less than 'b' and 'b' is less than 'c' then 'a' is less than 'c', …

Properties of Binary Search Trees

- **Binary Search Tree (BST) Storage Rules**
  - The entry in node n is never less than an entry in its left subtree (though it may be equal to one of these entries)
  - The entry in node n is less than every entry in its right subtree

- BSTs also can store a collection of strings, or real numbers, or **anything that can be compared using some sort of less-than comparison**

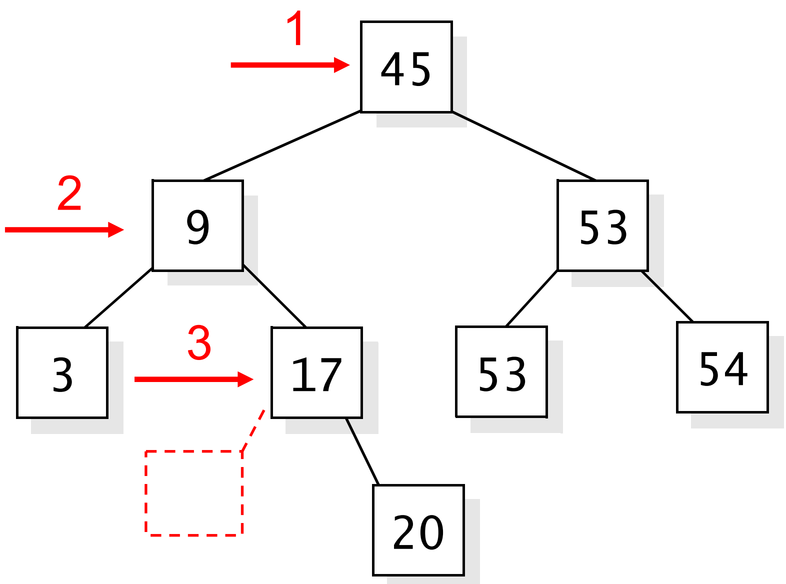
- This provides higher search efficiency ($O(\log(n))$) compared to the implementations using array or linked-list ($O(n)$)

- The higher efficiency of searching in a BST motivates us to implement the bag class with a BST

Properties of Binary Search Trees

- With a binary search tree, searching for an entry is often much quicker

Assume we are looking for number 16



To find an item, in the worst case we need to examine *d* entries

- ***d* = the depth of the tree + 1**

Implementing the Bag Class with a Binary Search Tree

- **Invariant for the Sixth Bag:**
  - The items in the bag are stored in a binary search tree
  - The root pointer of the binary search tree is stored in the member variable `root_ptr` (which may be NULL for an empty tree)

```
template <class Item>
class bag
{

 public:
    || Prototypes of public member functions go here.

 private:
    binary_tree_node<Item> *root_ptr; // Root pointer
};
```

Binary Search Trees

Implementing the Bag Class with a Binary Search Tree

```
#ifndef BAG6_H
#define BAG6_H
#include <cstdlib>    // Provides NULL and size_t
#include "bintree.h" // Provides binary_tree_node and related functions

namespace scu_coen79_10
{
    template <class Item>
    class bag
    {
    public:
        // TYPEDEF
        typedef std::size_t size_type;
        typedef Item value_type;

        // CONSTRUCTORS and DESTRUCTOR
        bag( );
        bag(const bag& source);
        ~bag( );
```

Implementing the Bag Class with a Binary Search Tree

```
    // MODIFICATION functions
    size_type erase(const Item& target);
    bool erase_one(const Item& target);
    void insert(const Item& entry);
    void operator +=(const bag& addend);
    void operator =(const bag& source);

    // CONSTANT functions
    size_type size( ) const;
    size_type count(const Item& target) const;

private:
    binary_tree_node<Item> *root_ptr;
    void insert_all(binary_tree_node<Item>* addroot_ptr);
};
```

This function is used by the `operator +=`

Implementing the Bag Class with a Binary Search Tree

```
    // NONMEMBER functions for the bag<Item> template class
    template <class Item>
    bag<Item> operator +(const bag<Item>& b1, const bag<Item>& b2);
}

#include "bag6.template" // Include the implementation.
#endif
```

```
Header Definition: end
```

- **Full implementation is a programming project**
- **The next few slides provide you with some hints for the implementation**

Implementing the Bag Class with a Binary Search Tree

Constructors:

- The default constructor sets `root_ptr = NULL`
- The copy constructor needs to make a new copy of the source's tree, and point `root_ptr` to the root of this copy
  - Use the `tree_copy` function to do the copying

The destructor:

- Return all nodes to the heap
- Use an appropriate function from `bintree.h`

Implementing the Bag Class with a Binary Search Tree

The assignment operator:

- The bag uses dynamic memory, therefore, we must overload the assignment operator
- The assignment operator works like the copy constructor with two preliminary steps:
  1. First check if it is a self-assignment by comparing (`this == &source`): If yes, then return
  2. If there is no self-assignment, then before we copy the source tree we must release all memory used by the nodes of the current tree
     - Use `tree_clear` to release memory

The `size` member function:

- Simply returns the answer from `tree_size(root_ptr)` using the `tree_size` function from `bintree.h`

Implementing the Bag Class with a Binary Search Tree

The `count` member function: Counts the number of occurrences of an item called `target`

```
template <class Item>
typename bag<Item>::size_type  bag<Item>::count(const Item& target)
                                                            const
{
    size_type answer = 0;
    binary_tree_node<Item> *cursor;

    cursor = root_ptr;

    TODO: Use a loop to move the cursor down through the tree, always
    moving along the path where the target might occur


    return answer;
}
```

Implementing the Bag Class with a Binary Search Tree

The `count` member function (Cont'd)

At each point in the tree we have four possibilities:

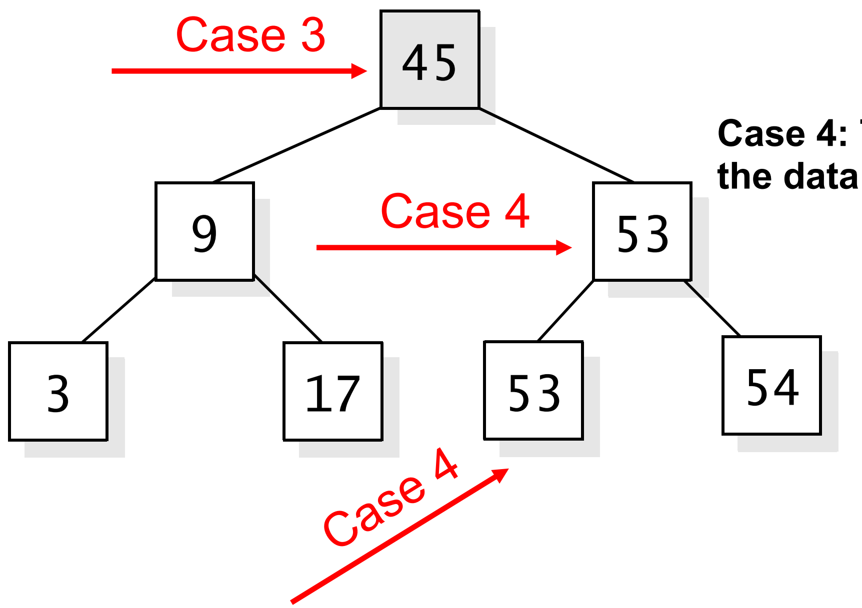
1. **The `cursor` can become NULL:** End the loop and return
2. **The data at the `cursor` node might be larger than the target**: The target can appear only in the left subtree
   - `cursor = cursor->left( );`
3. **The data at the `cursor` node might be smaller than the target**
   - `cursor = cursor->right( );`
4. **The target might equal the data at the `cursor` node**
   - Add one to `answer`
   - Continue the search to the left (**since items to the left are less than or equal to the item at the cursor node**)

Implementing the Bag Class with a Binary Search Tree

- ❑ **Assume we want to count number of occurrences of 53**

Case 3: The data at the cursor node is smaller than the target

Case 4: The target is equal to the data at the cursor node



Implementing the Bag Class with a Binary Search Tree

- **The `insert` member function:**

- Adds a new item to a binary search tree

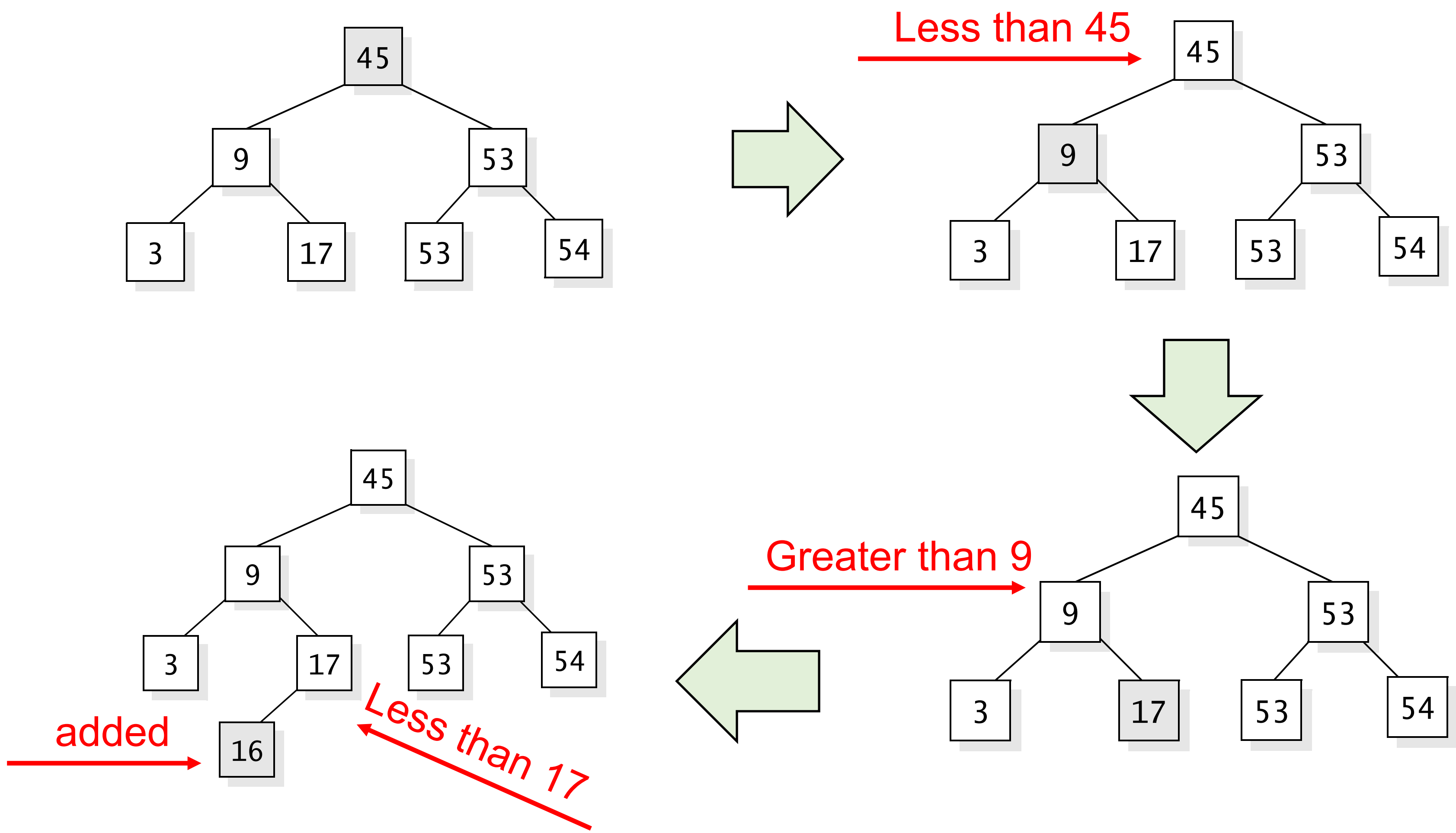
```
void insert(const Item& entry);
```

- **Case 1**: First handle this special case: When **the first entry is inserted**, simply call `root_ptr = new binary_tree_node<Item>(entry)`
- **Case 2:** There are already some other entries in the tree:
  - We pretend to search for the exact entry that we are trying to insert
  - We stop the search just before the cursor falls off the bottom of the tree, and we insert the new entry at the spot where the cursor was about to fall off

- Use a boolean variable called `done`, which is initialized to false
- Implement a loop that continues until `done` becomes true

Implementing the Bag Class with a Binary Search Tree

- Consider the task of inserting 16



Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
void bag<Item>::insert(const Item& entry)
// Header file used: bintree.h
{
    binary_tree_node<Item> *cursor;


    if (root_ptr == NULL)
    {   // Add the first node of the binary search tree:
        root_ptr = new binary_tree_node<Item>(entry);
        return;
    }


    else
    {   // Move down the tree and add a new leaf:
        cursor = root_ptr;

    }
}
```

When the tree is empty

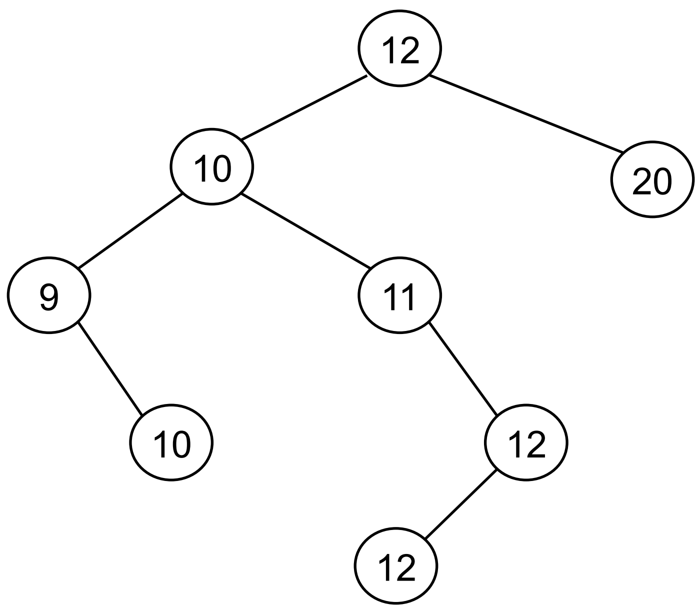
When the tree is not empty

TODO: Find the position to add the new entry, then add it

Implementing the Bag Class with a Binary Search Tree

- How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

- A **Simple Solution** is to allow same keys on left side (we could also choose right side)
- For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree



Implementing the Bag Class with a Binary Search Tree

- **The `erase_one` member function:**

The `erase_one` member function: Removes a specified item from a binary search tree

- Prototype: `bool erase_one(const Item& target);`

- We implement the `erase_one` function with two auxiliary functions to reduce the complexity of implementation

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
bool bag<Item>::erase_one(const Item& target)
{
     return bst_remove(root_ptr, target);
}
```

uses `bst_remove`
(explained later)

```
template <class Item>
bool bst_remove(binary_tree_node<Item>*& root_ptr,
                                         const Item& target)
```

uses `bst_remove_max`
(explained later)

```
template <class Item>
void bst_remove_max(binary_tree_node<Item>*& root_ptr,
                                         Item& removed)
```

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
bool bst_remove(binary_tree_node<Item>*& root_ptr, const Item& target);
```

It is a reference to a pointer since the function will affect the passed pointer

It is a const reference as target is only used to find the node to be deleted

```
// Precondition: root_ptr is a root pointer of a binary search tree
// (or may be NULL for the empty tree).

// Postcondition: If target was in the tree, then one copy of target
// has been removed, root_ptr now points to the root of the new
// (smaller) binary search tree, and the function returns true.
// Otherwise, if target was not in the tree, then the tree is
// unchanged, and the function returns false.
```

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
bool bst_remove(binary_tree_node<Item>*& root_ptr, const Item& target)
{
 binary_tree_node<Item> *oldroot_ptr;
 if (root_ptr == NULL) { return false; }

 if (target < root_ptr->data( )) {
    return bst_remove(root_ptr->left( ), target);   }

 if (target > root_ptr->data( )) {
    return bst_remove(root_ptr->right( ), target); }

 if (root_ptr->left( ) == NULL)
 {                  .........
       return true;    }

 bst_remove_max(root_ptr->left( ),
                      root_ptr->data( ));

 return true;
}
```

target not yet found

Without left tree, and w/ or w/o right tree

With left tree; and w/ or w/o right tree

target found

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
bool bst_remove(binary_tree_node<Item>*& root_ptr, const Item& target)
```

- Employs a recursive implementation to remove the target

Handles these cases:

- **Case 1: Empty tree:** return
- **Case 2: The target less than the root entry:** make a recursive call to delete the target from the left subtree
- **Case 3: The target greater than the root entry:** make a recursive call to delete the target from the right subtree
- **Case 4: The target equal to the root entry**
  - **Case 4a:** The root node has no left child
  - **Case 4b:** The root node does have a left child

`target` not yet found

`target` found

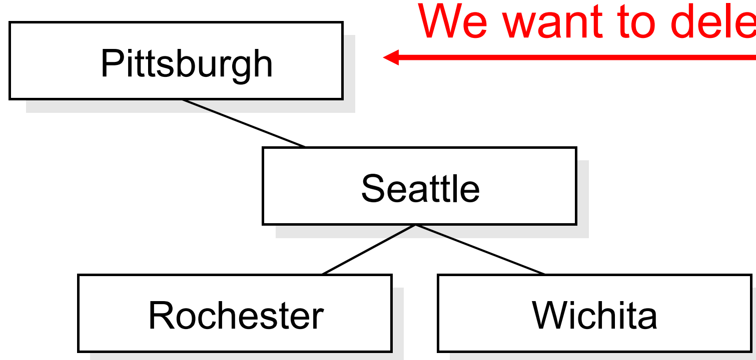
Implementing the Bag Class with a Binary Search Tree

- Case 4: The target equal to the root entry
  - **Case 4a: The root node has no left child**

❑ Example

The left subtree is empty.

We want to delete



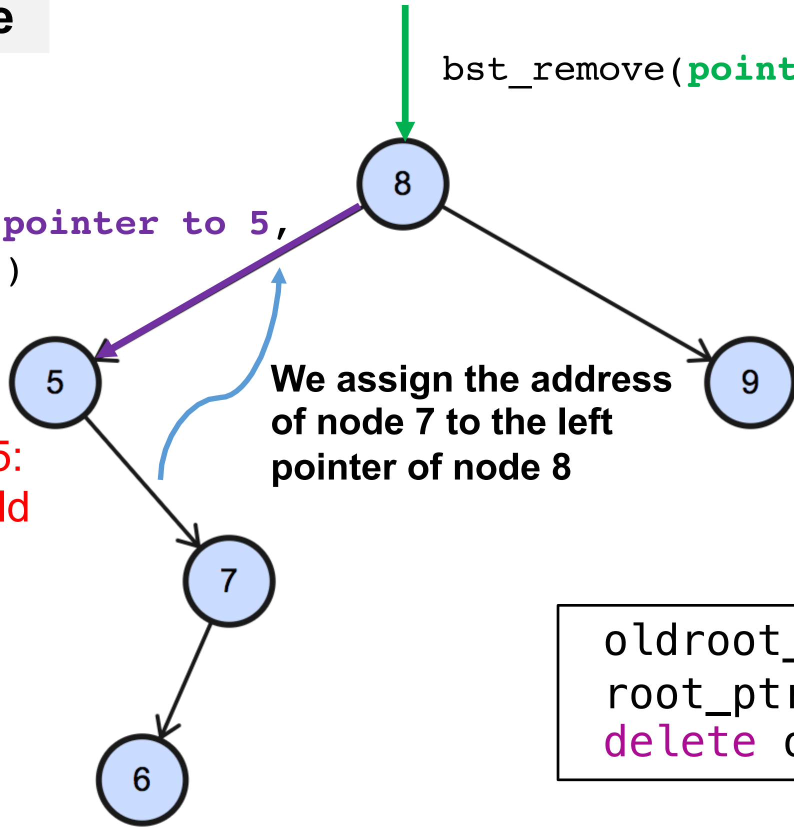
- **We can delete the root entry and make the right child (Seattle) the new root node**

```
oldroot_ptr = root_ptr;
root_ptr = root_ptr->right( );
delete oldroot_ptr;
```

- **This scheme also works properly if there is no right child**

Implementing the Bag Class with a Binary Search Tree

❑ Example

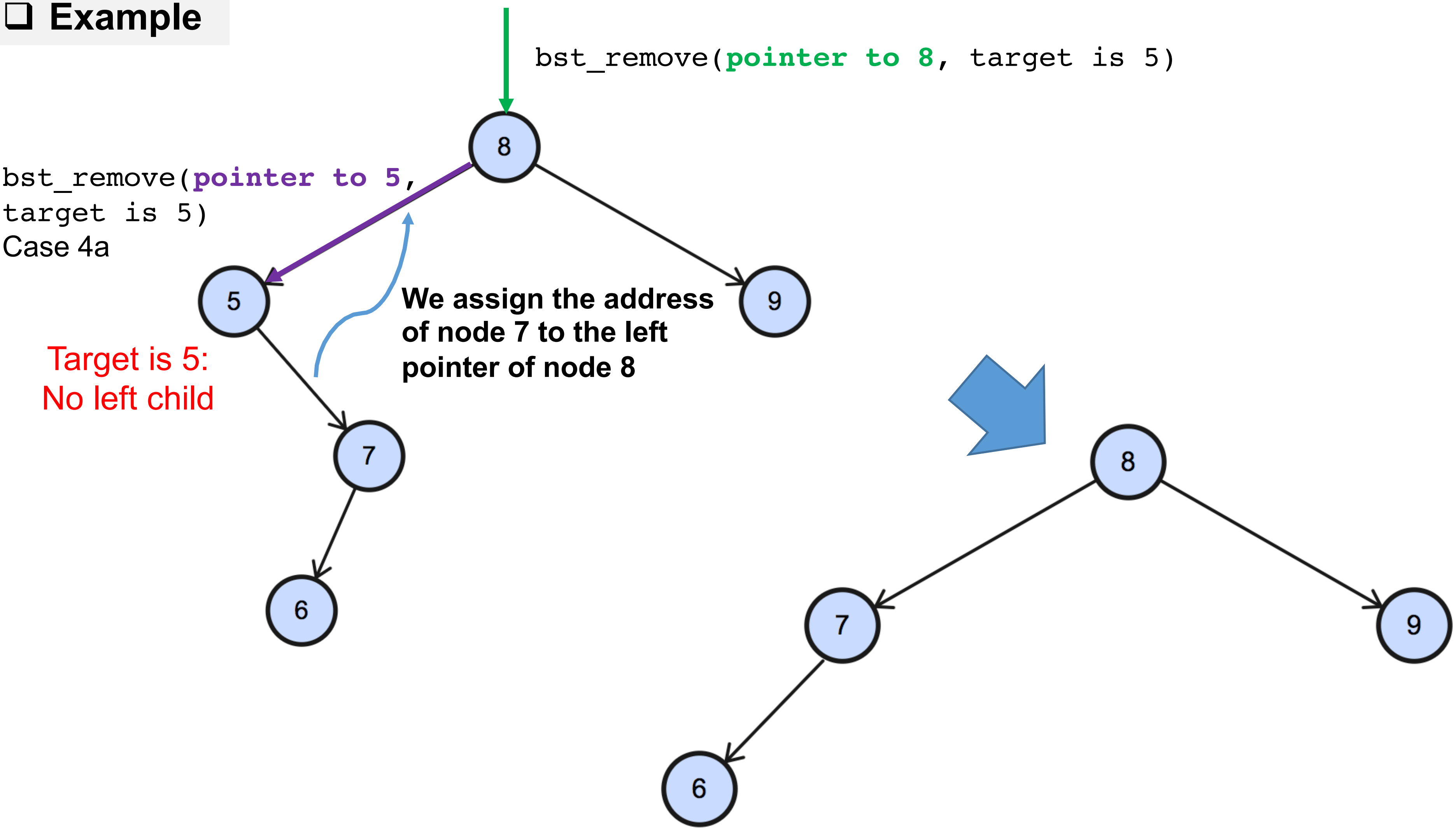



```
oldroot_ptr = root_ptr;
root_ptr = root_ptr->right( );
delete oldroot_ptr;
```

```
oldroot_ptr = pointer to 5;
root_ptr = pointer to 7;
delete node 5;
```

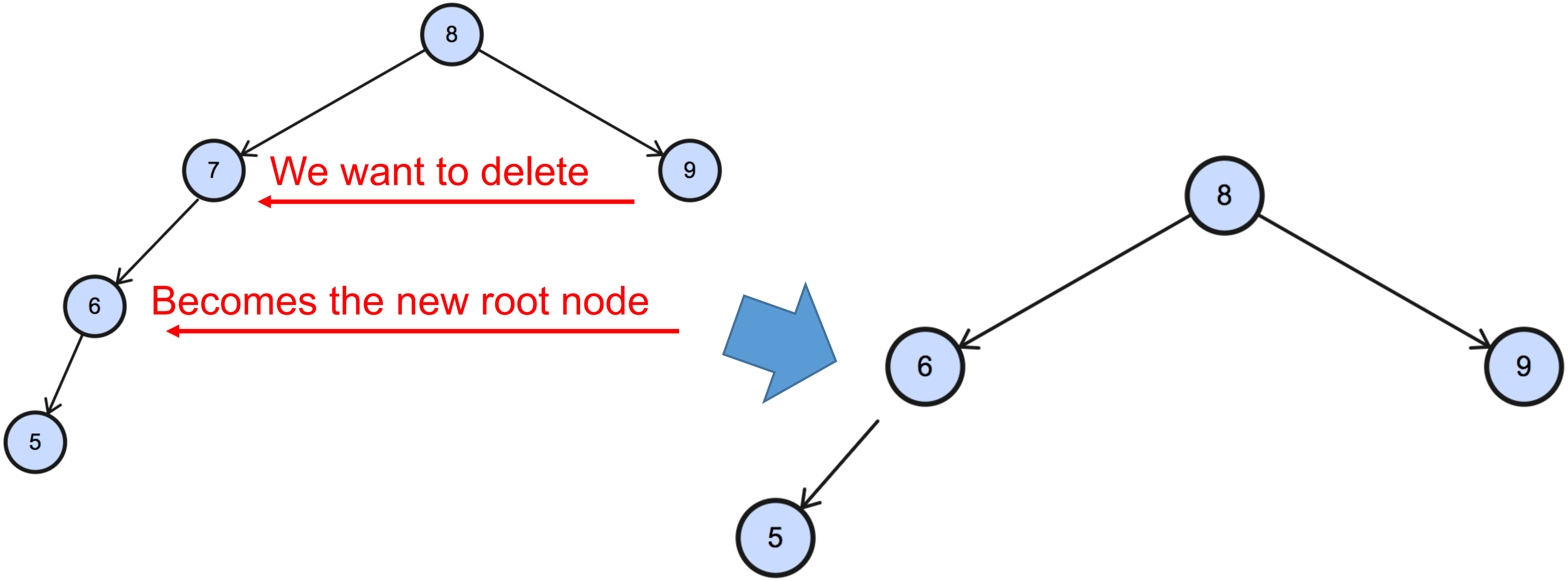
Implementing the Bag Class with a Binary Search Tree

❑ Example



Implementing the Bag Class with a Binary Search Tree

- Case 4: The target equal to the root entry
    - **Case 4b: The root node does have a left child**
- **If there is no right child, then the left child can become the new root**



– What if it has a right child? We need a better solution…

Implementing the Bag Class with a Binary Search Tree

We want to design a more general solution

- **To find some entry in the non-empty left subtree, and move this entry up to the root**
- **This case works even when the node to be deleted has a right child**

- Question: How to find this entry?

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
void bst_remove_max( binary_tree_node<Item>*& root_ptr, Item& removed );

// Precondition: root_ptr is a root pointer of a non-empty binary
// search tree.

// Postcondition: The largest item in the binary search tree has been
// removed, and root_ptr now points to the root of the new (smaller)
// binary search tree. The reference parameter, removed, has been set
// to a copy of the removed item.
```

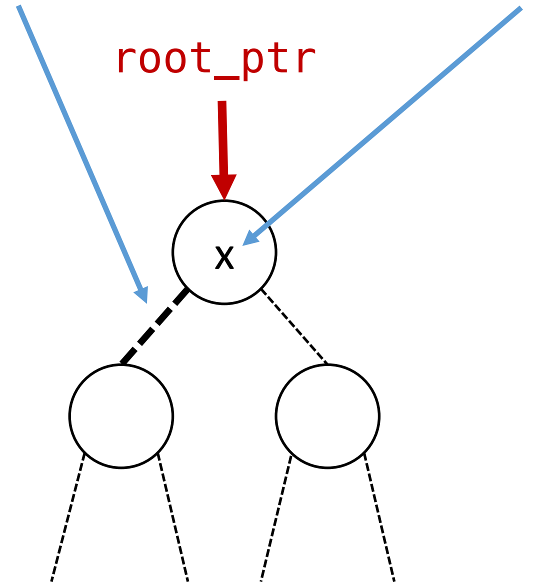
Implementing the Bag Class with a Binary Search Tree

- How do we delete the largest item from the left subtree, and place this same item at the root?

Removes the largest item from the tree pointed to by this pointer

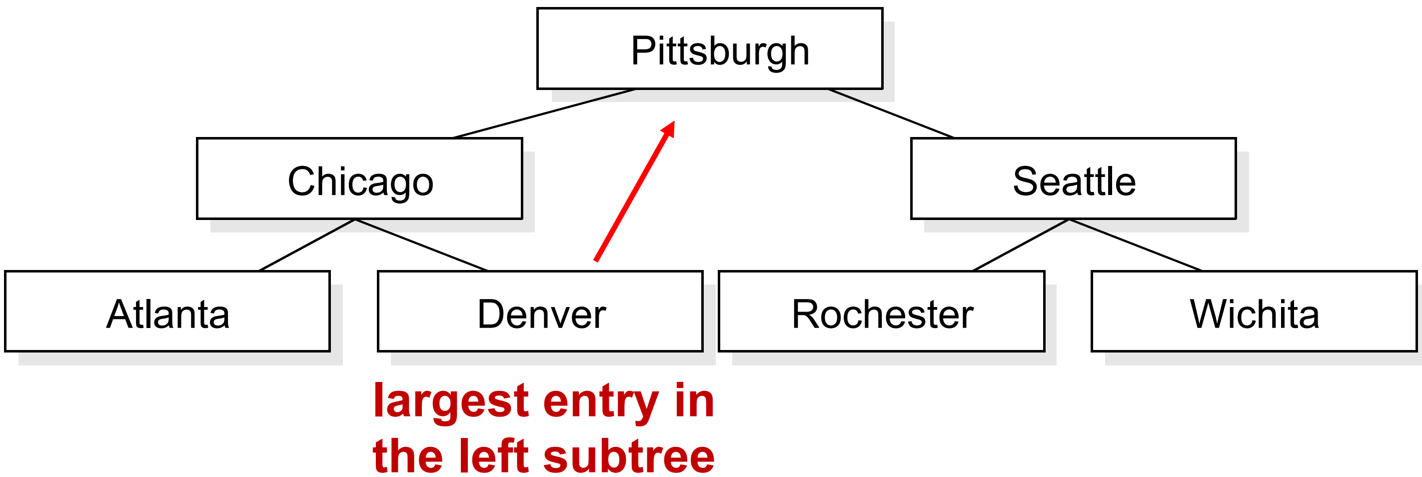
Places the largest item in the tree pointed to by `root_ptr->left( )` in `root_ptr->data( ));`

```
bst_remove_max(root_ptr->left( ), root_ptr->data( ));
```



Implementing the Bag Class with a Binary Search Tree

- ❑ **Assume we want to delete “Pittsburgh”**

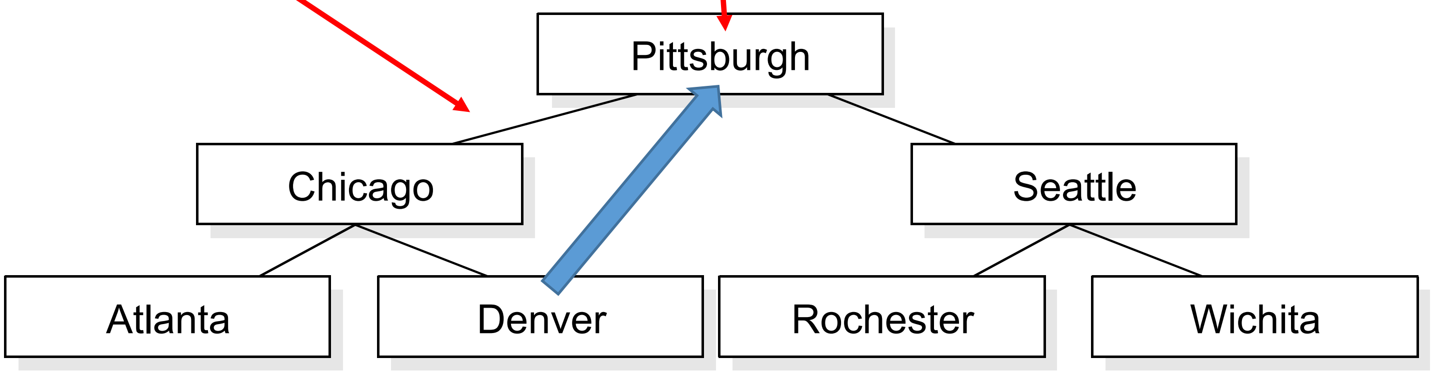


- **We need to find the largest item in the left sub-tree**

Implementing the Bag Class with a Binary Search Tree

- Assume we want to delete “Pittsburgh”

```
bst_remove_max(root_ptr->left( ), root_ptr->data( ));
```



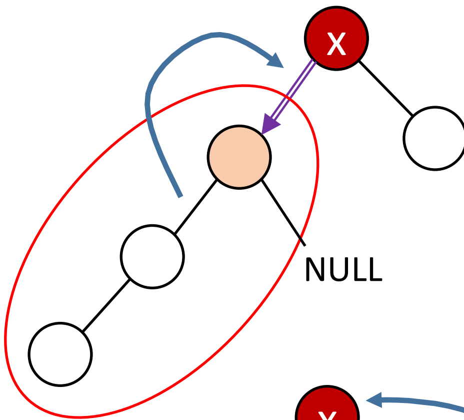
Binary Search Trees

Implementing the Bag Class with a Binary Search Tree

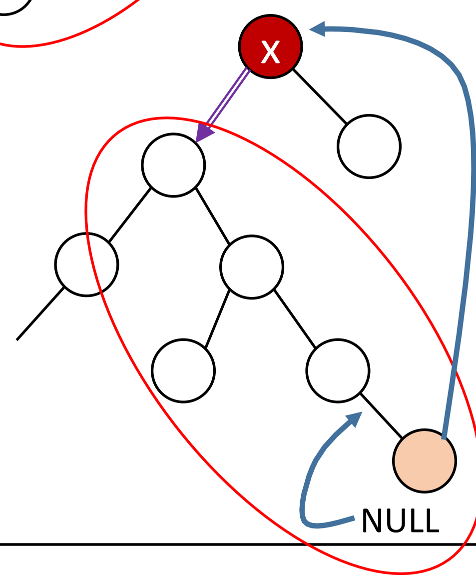
- Implementing the `bst_remove_max` function

```
template <class Item>
void bst_remove_max(binary_tree_node<Item>*& root_ptr, Item& removed)
{
```

Case 1: No right child: The largest item is at the root, so you can set `removed` equal to the data from the root, move the root pointer down to the left, and delete the root node



Case 2: There is a right child: There are larger items in the right subtree. In this case, make a recursive call to delete the largest item from the right subtree




```
}
```

Implementing the Bag Class with a Binary Search Tree

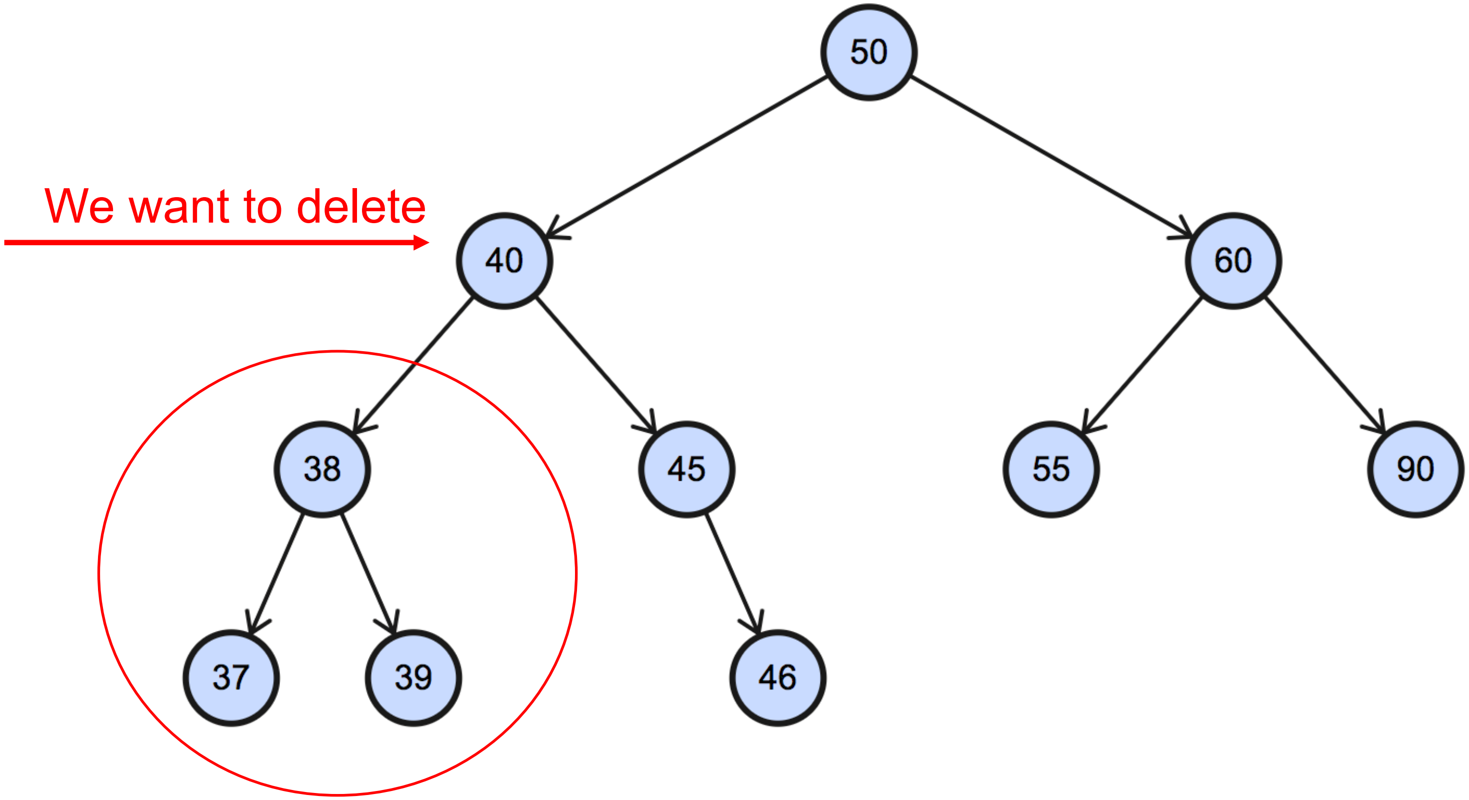
```
template <class Item>
void bst_remove_max(binary_tree_node<Item>*& root_ptr, Item& removed)
// Precondition: root_ptr is a root pointer of a non-empty binary
// search tree.
// Postcondition: The largest item in the binary search tree has been
// removed, and root_ptr now points to the root of the new (smaller)
// binary search tree. The reference parameter, removed, has been set
// to a copy of the removed item.
{
    binary_tree_node<Item> *oldroot_ptr;
    assert(root_ptr != NULL);

    if (root_ptr->right( ) == NULL)
     {
        removed = root_ptr->data( );
        oldroot_ptr = root_ptr;
        root_ptr = root_ptr->left( );
        delete oldroot_ptr;
      }

    else
      {  bst_remove_max(root_ptr->right( ), removed); }
}
```

No right child: The largest item is at the root

There is a right child: There are larger items in the right subtree

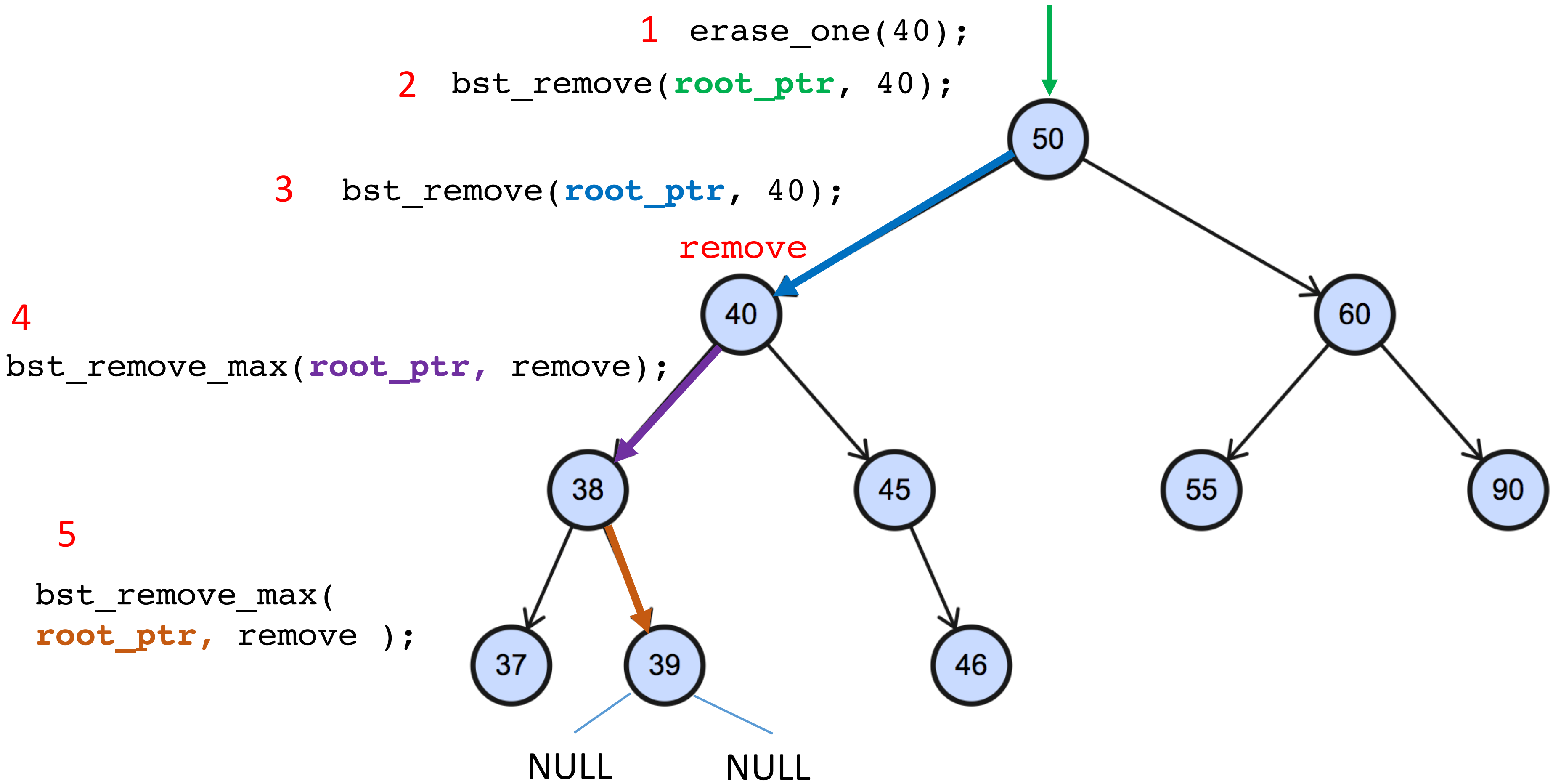
Implementing the Bag Class with a Binary Search Tree

- Assume we want to delete 40



We need to find the largest entry in the left subtree and remove it

Implementing the Bag Class with a Binary Search Tree




```
removed = root_ptr->data( );
oldroot_ptr = root_ptr;
root_ptr = root_ptr->left( );
delete oldroot_ptr;
```

copy 39 to the value of node 40;
oldroot_ptr = pointer to 39;
right pointer of 38 = NULL;
delete pointer to 39;

Implementing the Bag Class with a Binary Search Tree

- **The `erase` member function:**

The `erase` member function: Removes all the occurrences of an item from a binary search tree

- Prototype: `bool erase(const Item& target);`

Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
bool bag<Item>::erase(const Item& target)
{
    return bst_remove_all(root_ptr, target);
}
```

uses `bst_remove_all`

```
template <class Item>
bool bst_remove_all(binary_tree_node<Item>*& root_ptr,
                                                      const Item& target)
```

uses `bst_remove_max`

```
template <class Item>
void bst_remove_max(binary_tree_node<Item>*& root_ptr,
                                                      Item& removed)
```

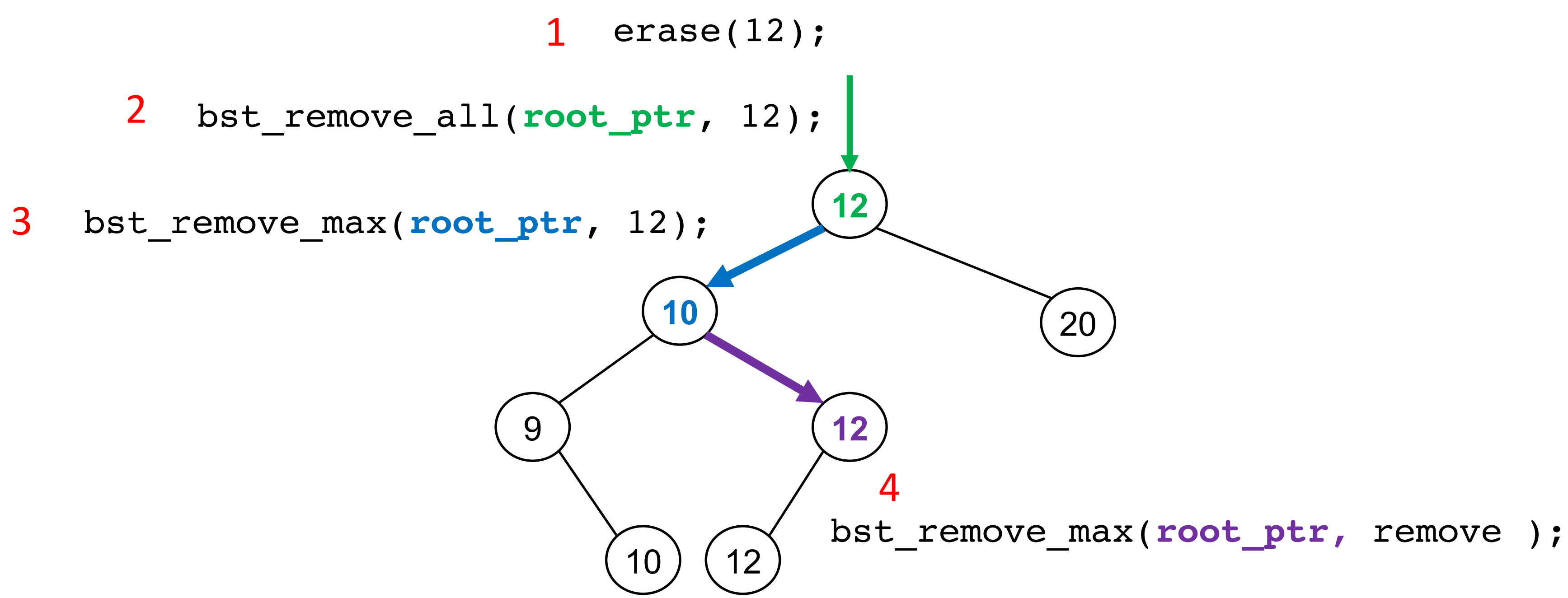
Implementing the Bag Class with a Binary Search Tree

```
template <class Item>
typename bag<Item>::size_type bst_remove_all
                (binary_tree_node<Item>*& root_ptr, const Item& target)
{
  binary_tree_node<Item> *oldroot_ptr;
  if (root_ptr == NULL) { return 0; }

  if (target < root_ptr->data( )) {
    return bst_remove_all(root_ptr->left( ), target);  }

  if (target > root_ptr->data( )) {
    return bst_remove_all(root_ptr->right( ), target); }

  if (root_ptr->left( ) == NULL)
  {   .........
      .........
      return 1;
  }
  bst_remove_max(root_ptr->left( ), root_ptr->data( ));

  return 1 + bst_remove_all(root_ptr, target);
}
```

Continue searching for more copies of the target to remove
This continued search must start at the current root (since the maximum element that we moved up from our left subtree might also be a copy of the target)

Implementing the Bag Class with a Binary Search Tree

- Assume we want to erase all instances of 12



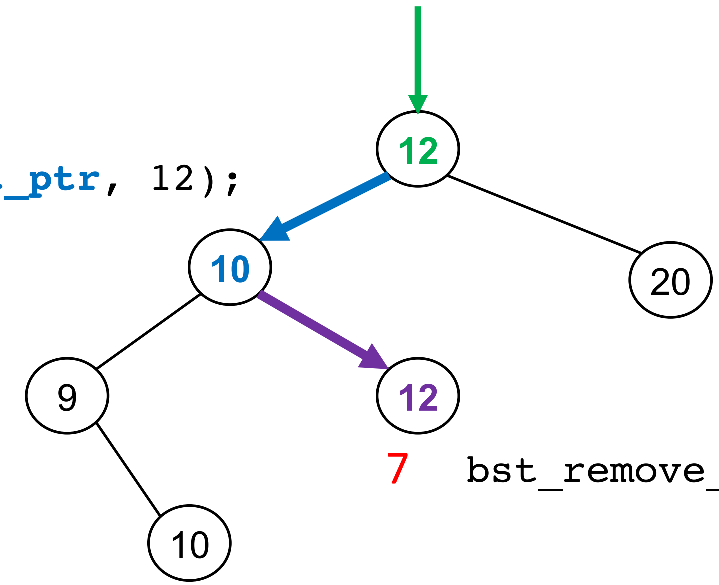

```
removed = root_ptr->data( );
oldroot_ptr = root_ptr;
root_ptr = root_ptr->left( );
delete oldroot_ptr;
```

copy 12 to the value of node 12;
oldroot_ptr = pointer to 12;
right pointer of 10 = left pointer of node 12;
delete node 12;

Implementing the Bag Class with a Binary Search Tree

5 `1 + bst_remove_all(root_ptr, 12);`

6 `bst_remove_max(root_ptr, 12);`



7 `bst_remove_max(root_ptr, 12);`

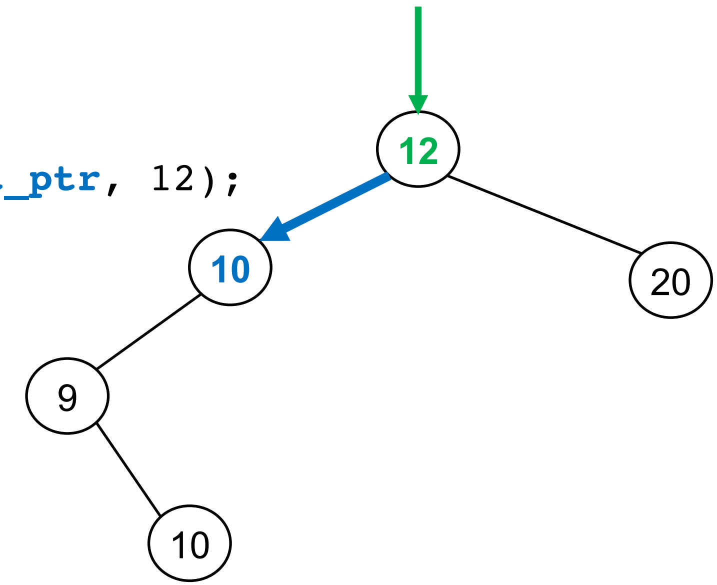
```
removed = root_ptr->data( );
oldroot_ptr = root_ptr;
root_ptr = root_ptr->left( );
delete oldroot_ptr;
```

copy 12 to the value of node 12;
oldroot_ptr = pointer to 12;
right pointer of 10 = left pointer of node 12;
delete node 12;

Implementing the Bag Class with a Binary Search Tree

8 `1 + bst_remove_all(root_ptr, 12);`

9 `bst_remove_max(root_ptr, 12);`




```
removed = root_ptr->data( );
oldroot_ptr = root_ptr;
root_ptr = root_ptr->left( );
delete oldroot_ptr;
```

copy 10 to the value of node 12;
oldroot_ptr = pointer to 10;
left pointer of 12 = right pointer of node 10;
delete node 10;

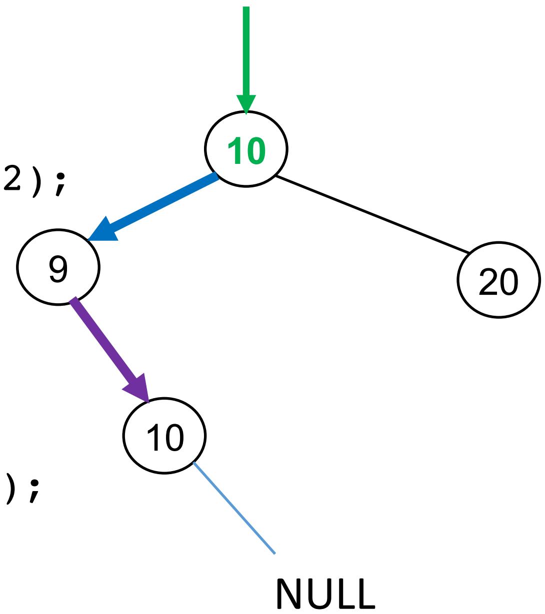
Implementing the Bag Class with a Binary Search Tree

10 `1 + bst_remove_all(root_ptr, 12);`

11 `bst_remove_all(root_ptr, 12);`

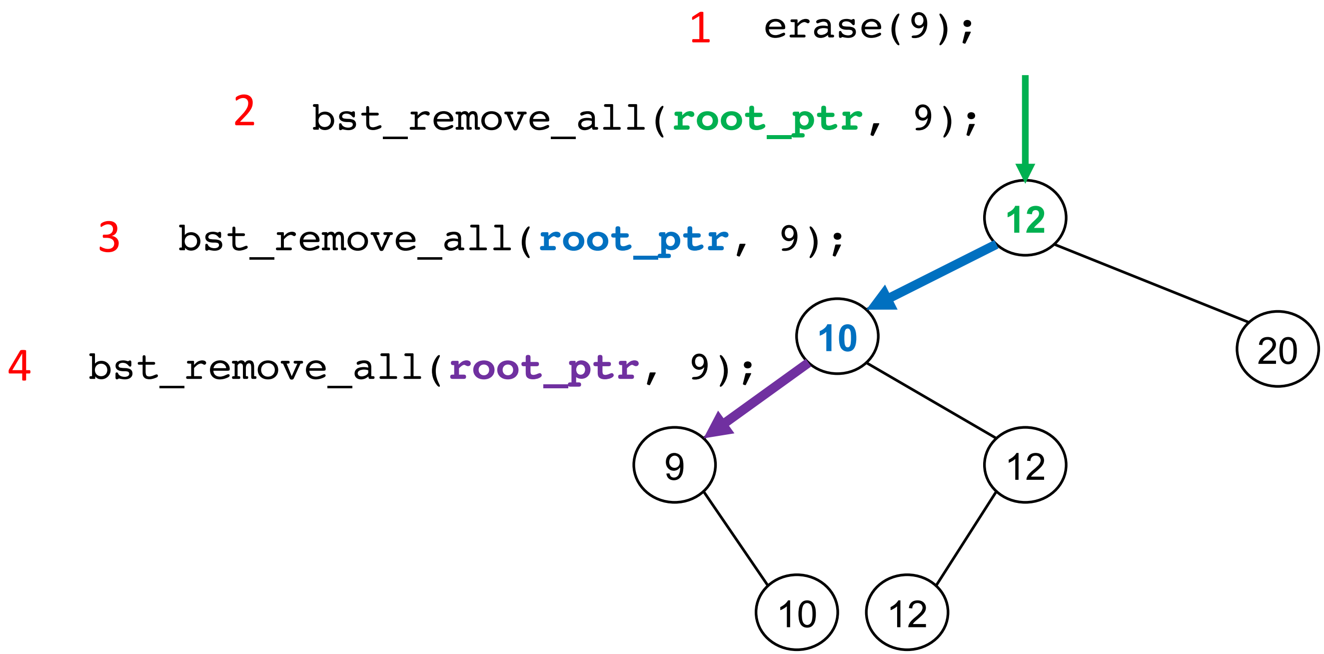
12 `bst_remove_all(root_ptr, 12);`

13 `bst_remove_all(NULL, 12);`



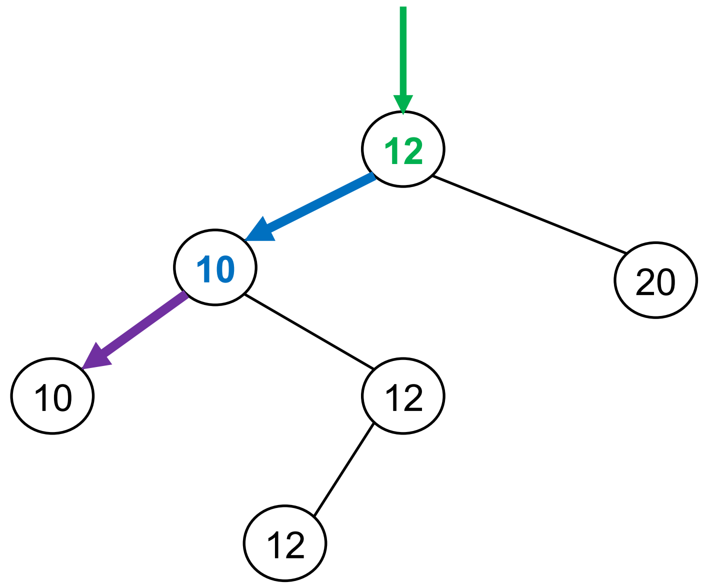
Implementing the Bag Class with a Binary Search Tree

- Assume we want to erase all instances of 9




```
if (root_ptr->left( ) == NULL)
{ // Target was found and there is no left subtree, so we can
  // remove this node, making the right child be the new root.
  oldroot_ptr = root_ptr;
  root_ptr = root_ptr->right( );
  delete oldroot_ptr;
  return 1;
}
```

Implementing the Bag Class with a Binary Search Tree



Implementing the Bag Class with a Binary Search Tree

- **The += member function:**

```
template <class Item>
void bag<Item>::operator +=(const bag<Item>& addend)
{
    if (root_ptr == addend.root_ptr)
      {
          bag<Item> copy = addend;
          insert_all(copy.root_ptr);
      }

    else
        insert_all(addend.root_ptr);
}
```

b += b

b += c

- **Benefits from an auxiliary function `insert_all`**
- **`insert_all` is actually another bag member function**

Implementing the Bag Class with a Binary Search Tree

- **The `insert_all` member function:**

```
template <class Item>
void bag<Item>::insert_all(binary_tree_node<Item>* addroot_ptr)
// Precondition: addroot_ptr is the root pointer of a binary search
// tree that is separate for the binary search tree of the bag that
// activated this method.
// Postcondition: All the items from the addroot_ptr's binary search
// tree have been added to the binary search tree of the bag that
// activated this method.

{
    if (addroot_ptr != NULL)
    {
       insert(addroot_ptr->data( ));
       insert_all(addroot_ptr->left( ));
       insert_all(addroot_ptr->right( ));
    }
}
```

Explicitly uses the pre-order traversal of the tree

Implementing the Bag Class with a Binary Search Tree

- We could also use the post-order traversal
- **Avoid in-order because:**
    - The nodes of the addend tree will be processed in order from smallest to largest
    - These nodes will be inserted into the other bag from smallest to largest
    - **The resulting tree ends up with a single long, narrow path, with only right children**
    - **Searching and other algorithms are inefficient when the trees lose their branching structure**

Summary

Summary

- Trees are a nonlinear structure
- Applications such as:
  - Organizing information (such as taxonomy trees)
  - Implementing an efficient version of the bag class (using binary search trees)
- Trees may be implemented with:
  - Fixed-sized arrays: Appropriate for complete binary trees
  - Dynamic data structures
- A tree traversal consists of processing a tree by applying some action to each node
  - Using parameters that are functions, we can write extremely flexible tree traversals

Summary

- Binary search trees are one common application of trees
  - Permit us to store a bag of ordered items in a manner where adding, deleting, and searching for entries is potentially much faster than with a linear structure
- Operations on trees are good candidates for recursive thinking
  - Because many tree operations include a step to process one or more sub- trees, and this step is “a smaller version of the same problem.”

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition
2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition
3) C++: Classes and Data Structures, by Jeffrey Childs
4) http://en.cppreference.com
5) http://www.cplusplus.com
6) https://isocpp.org