CS 106X, Lecture 20
Advanced Trees

reading:

Programming Abstractions in C++, Chapters 16.1-16.4

Plan For Today

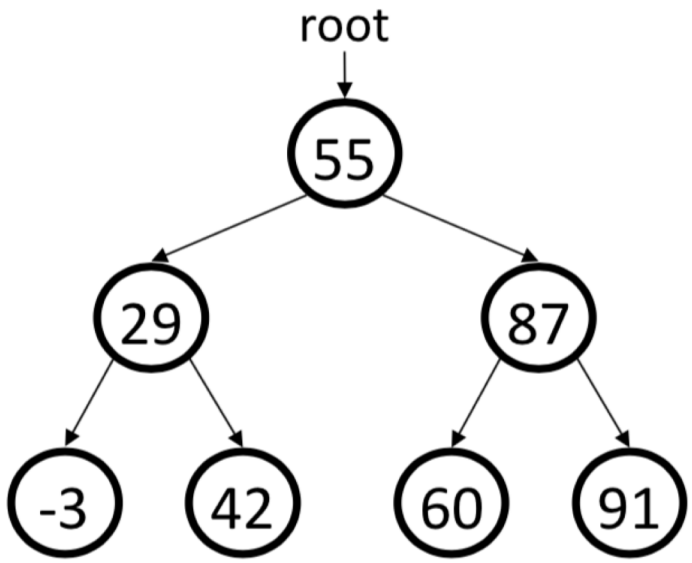
- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

Plan For Today

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

Binary Search Trees

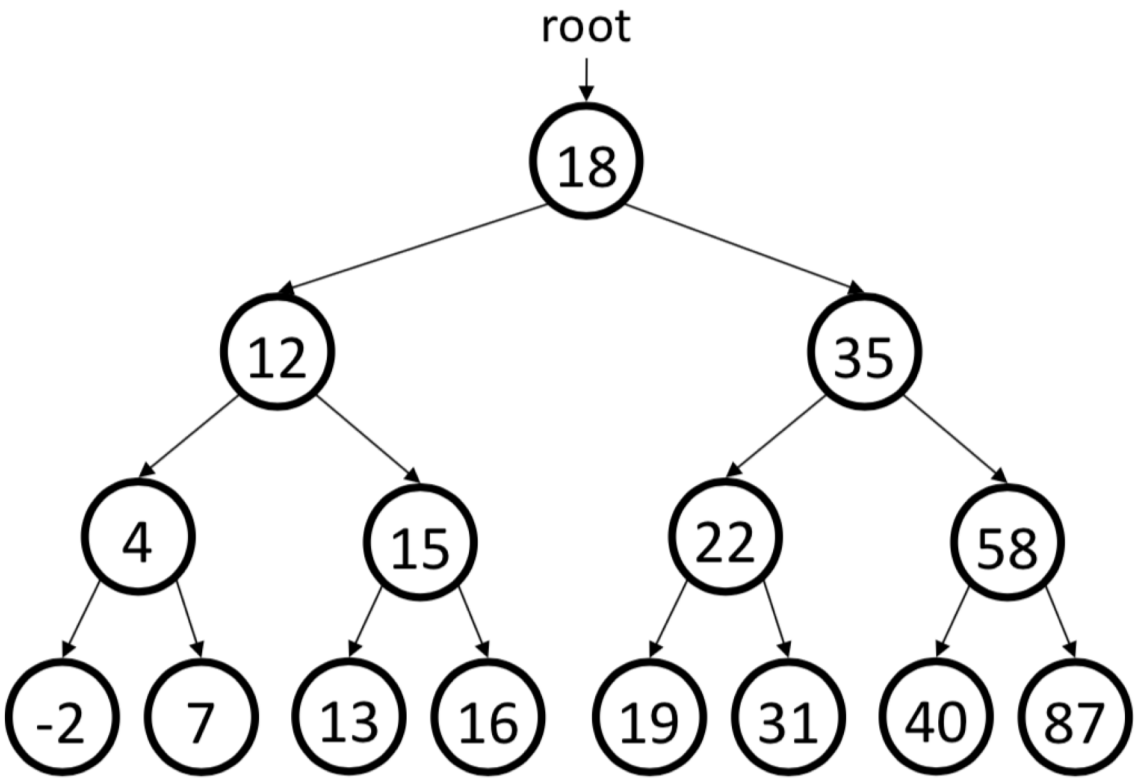
- **binary search tree** ("BST"): a binary tree where each non-empty node *R* has the following properties:
  - every element of *R*'s left subtree contains data less than *R*'s data,
  - every element of *R*'s right subtree contains data greater than *R*'s,
  - *R*'s left and right subtrees are also binary search trees.



Searching a BST

- Describe an algorithm for searching a binary search tree.
  - Try searching for the value 31, then 6.

- What is the maximum number of nodes you would need to examine to perform any search?



Searching a BST

- Each time we go left or right in a *balanced tree*, we discard *half* of the nodes left.
- **Question:** how many times can I divide by 2 until I get 1 node?

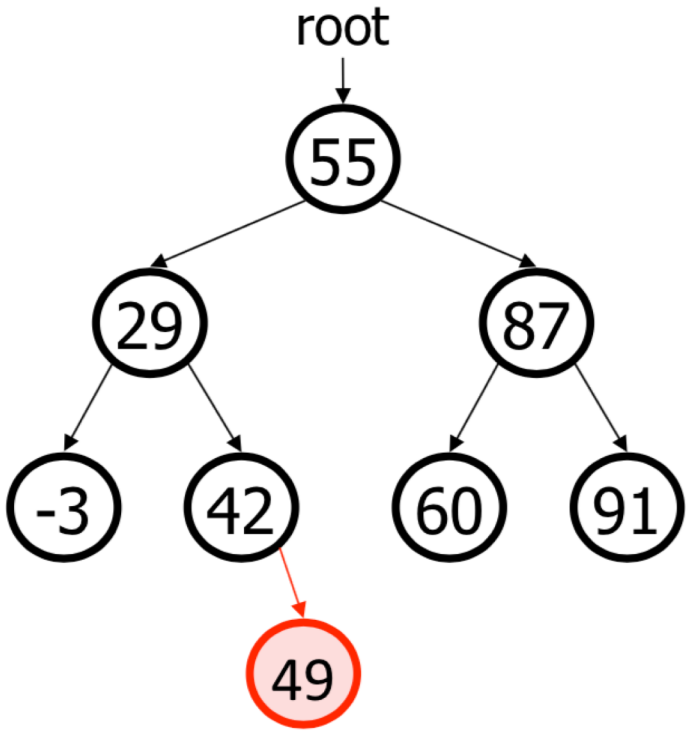
$$\boldsymbol{O}(\log_{\mathbf{2}} \boldsymbol{N})$$

Plan For Today

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

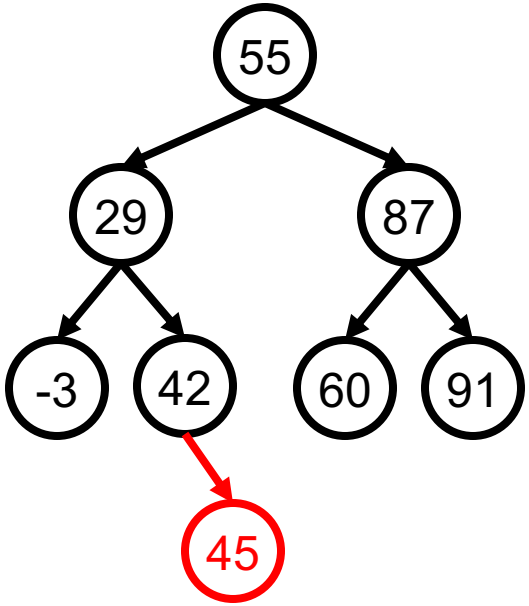
Exercise: add

- Write a function **add** that adds a given integer value to the BST.
  - Add the new value in the proper place to maintain BST ordering.
    - `tree.add(root, 49);`



Exercise: add

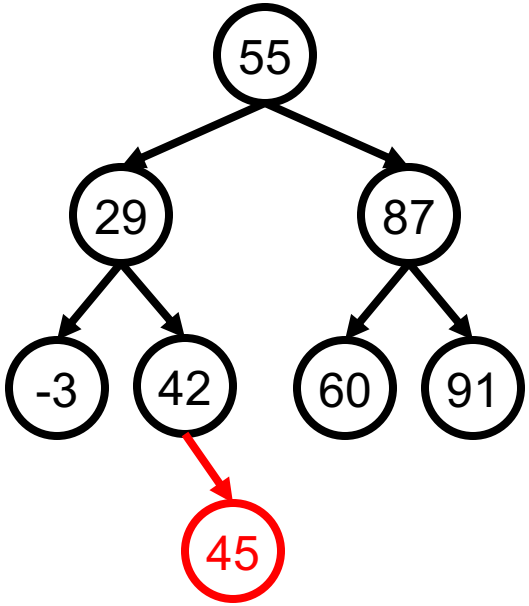
```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



Must pass the current node *by reference* for changes to be seen. Why?

Exercise: add

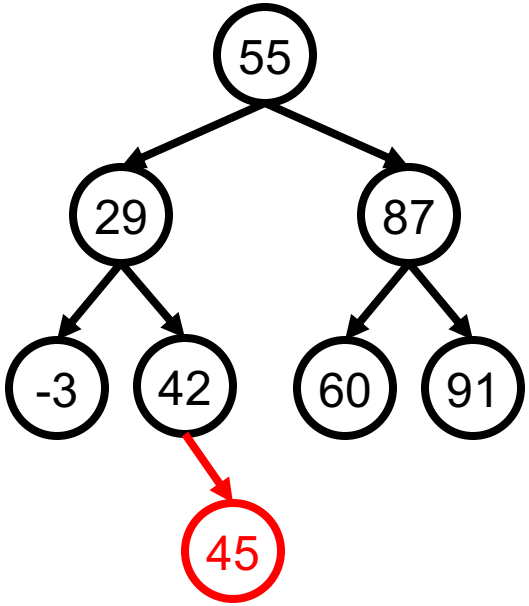
```
void add(TreeNode* node, int value) {
    if (node == nullptr) {
        // Add value at this place in tree
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



Must pass the current node *by reference* for changes to be seen. Why?

Exercise: add

```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



Must pass the current node *by reference* for changes to be seen. Why?

Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

heap

stack

main

root: **nullptr**

Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

heap

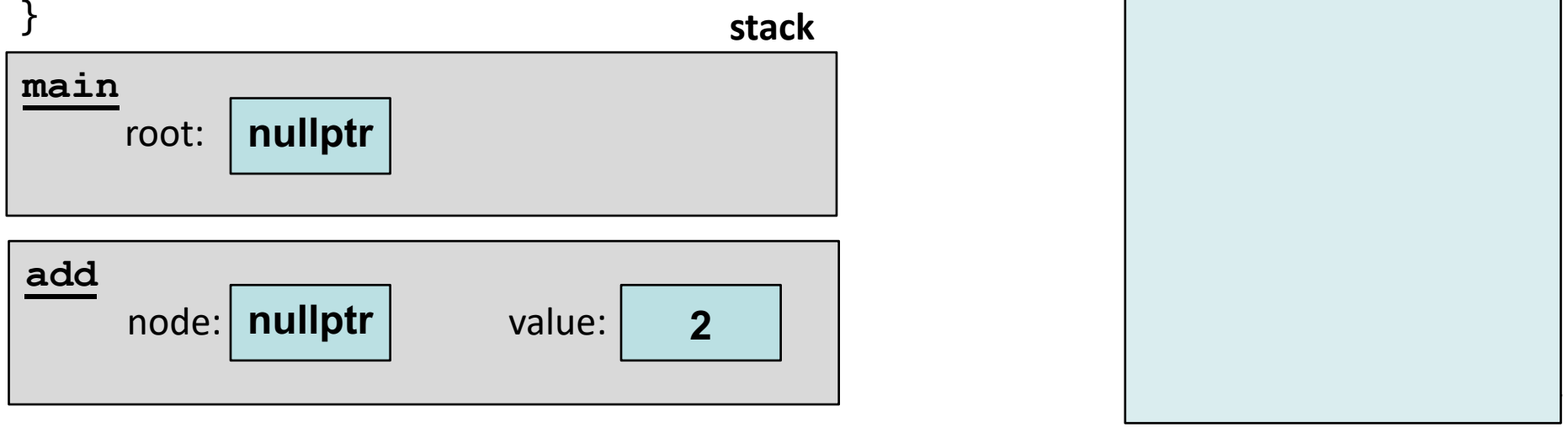
stack

main

root: **nullptr**

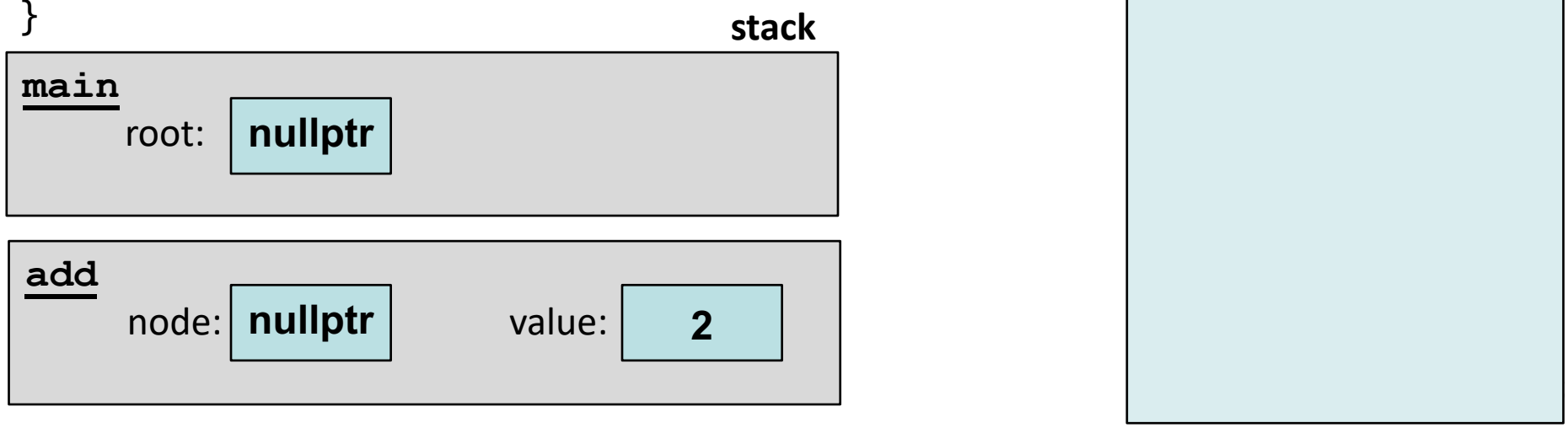
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



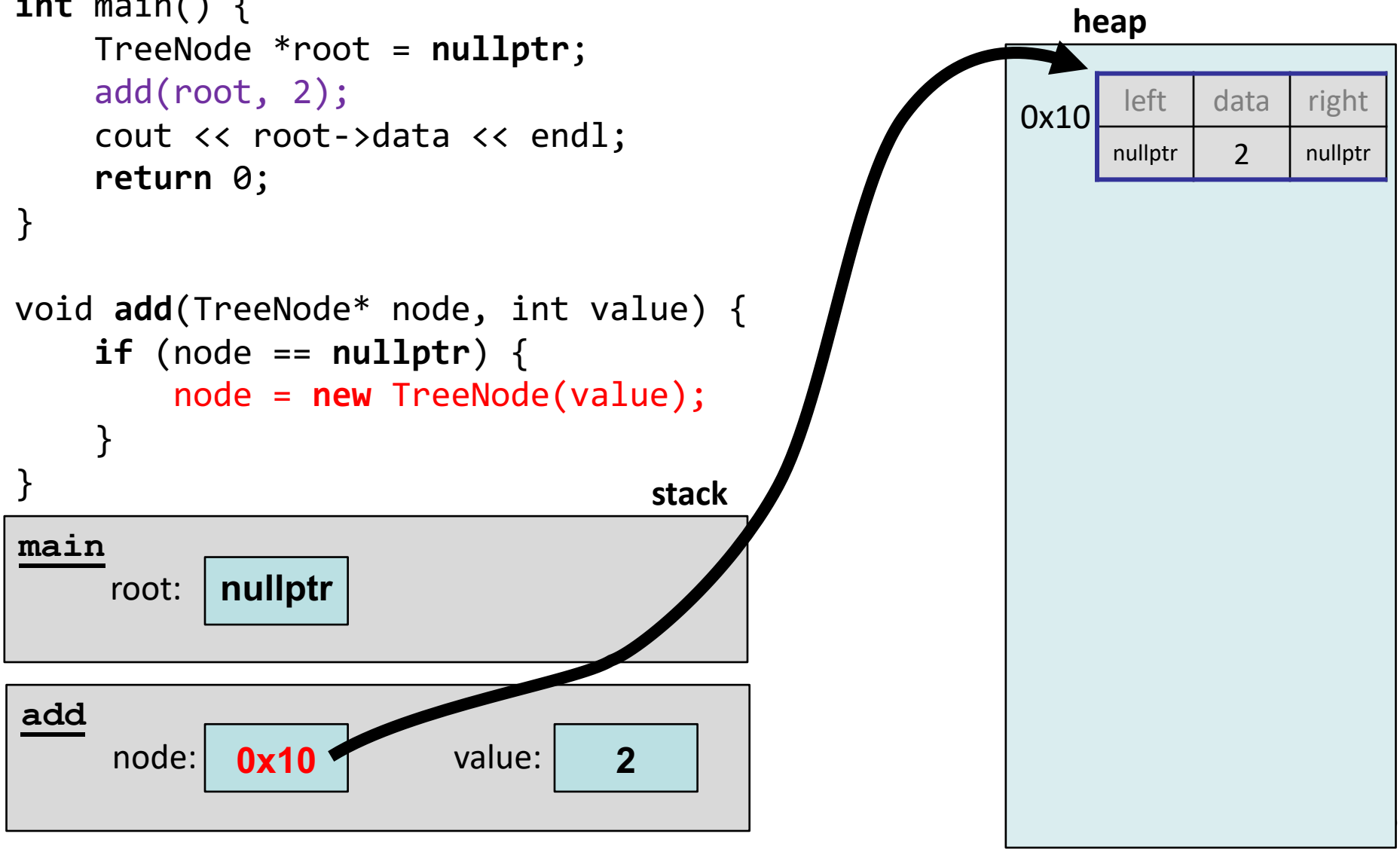
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



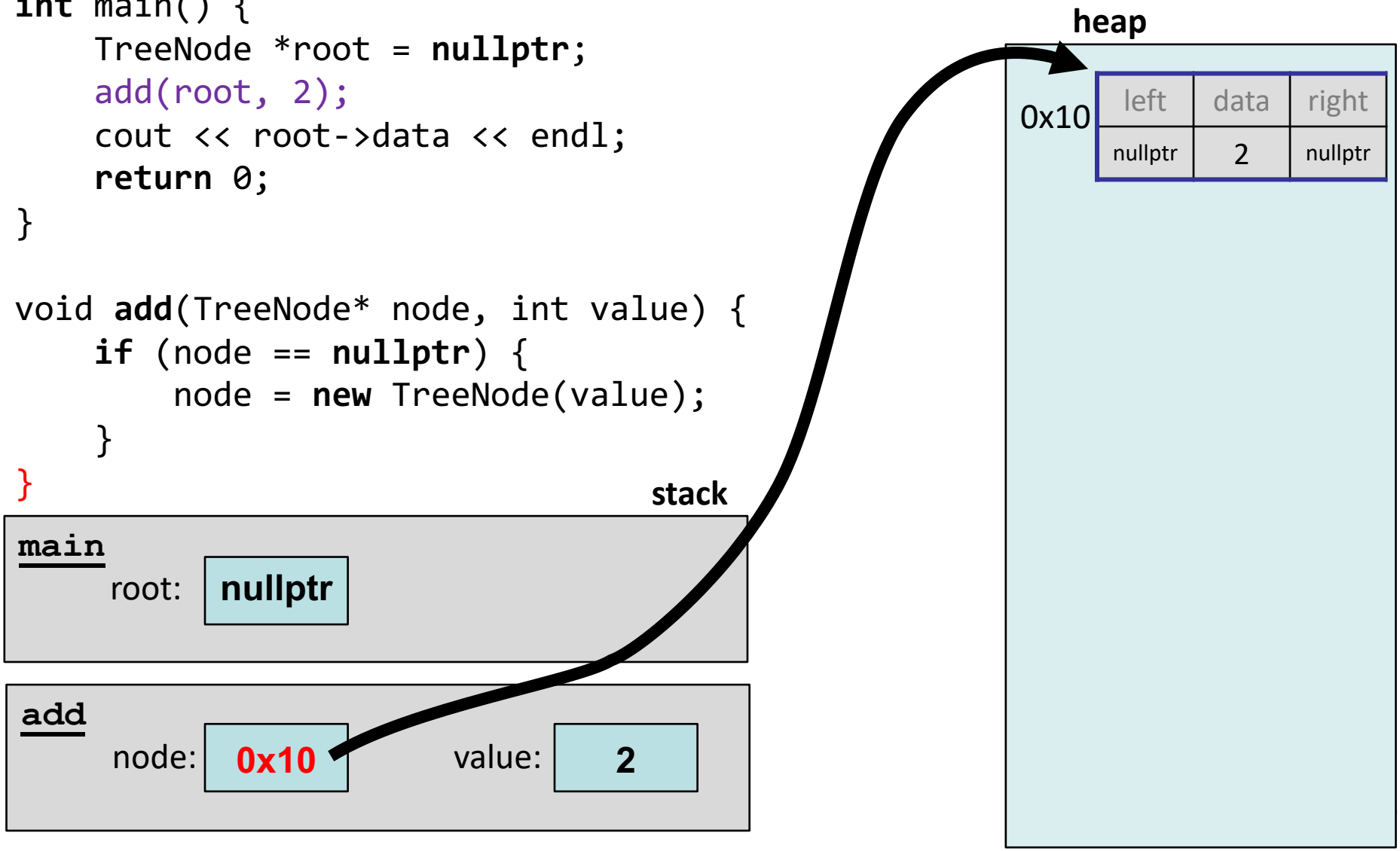
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

stack

main

root: **nullptr**

heap

0x10

left	data	right
nullptr	2	nullptr

Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

stack

main

root: **nullptr**

heap

0x10

left	data	right
nullptr	2	nullptr

Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode* node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

stack

main

root: **nullptr**

heap

0x10	left	data	right
	nullptr	2	nullptr

Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

heap

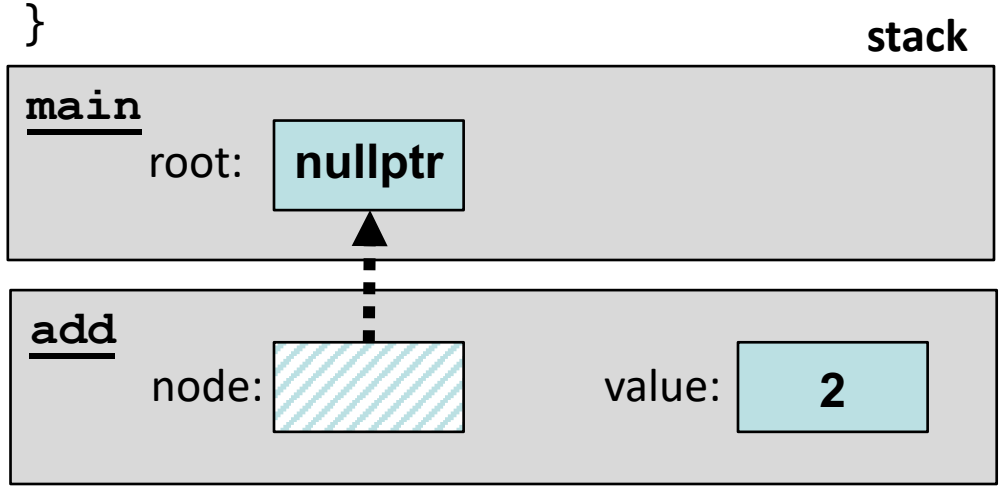
stack

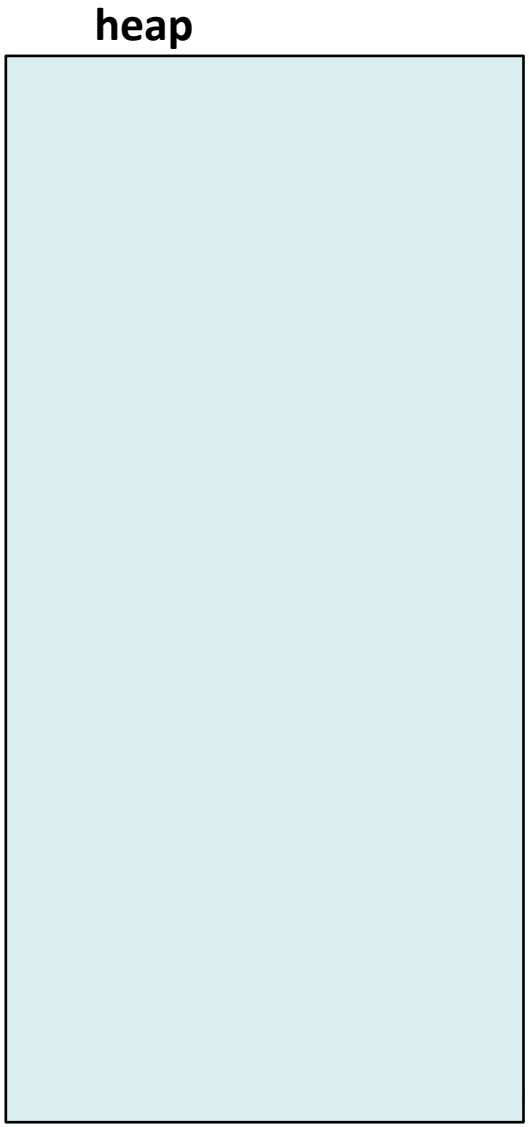
main

root: **nullptr**

Creating a List

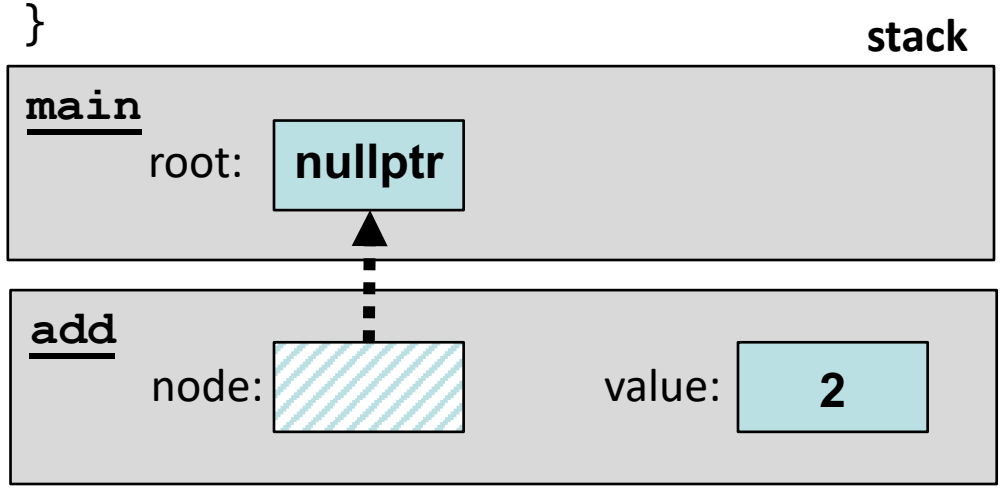
```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```

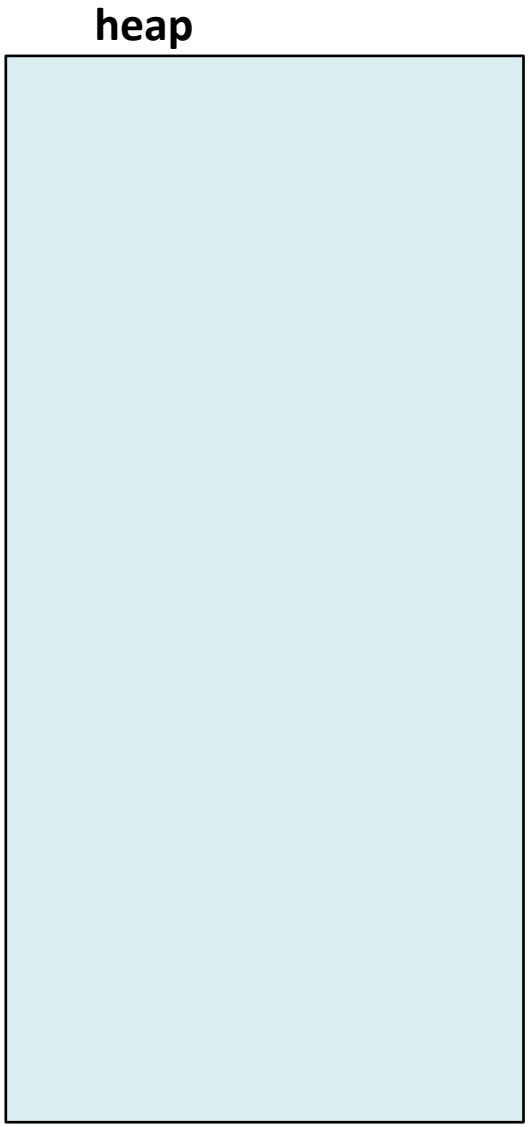




Creating a List

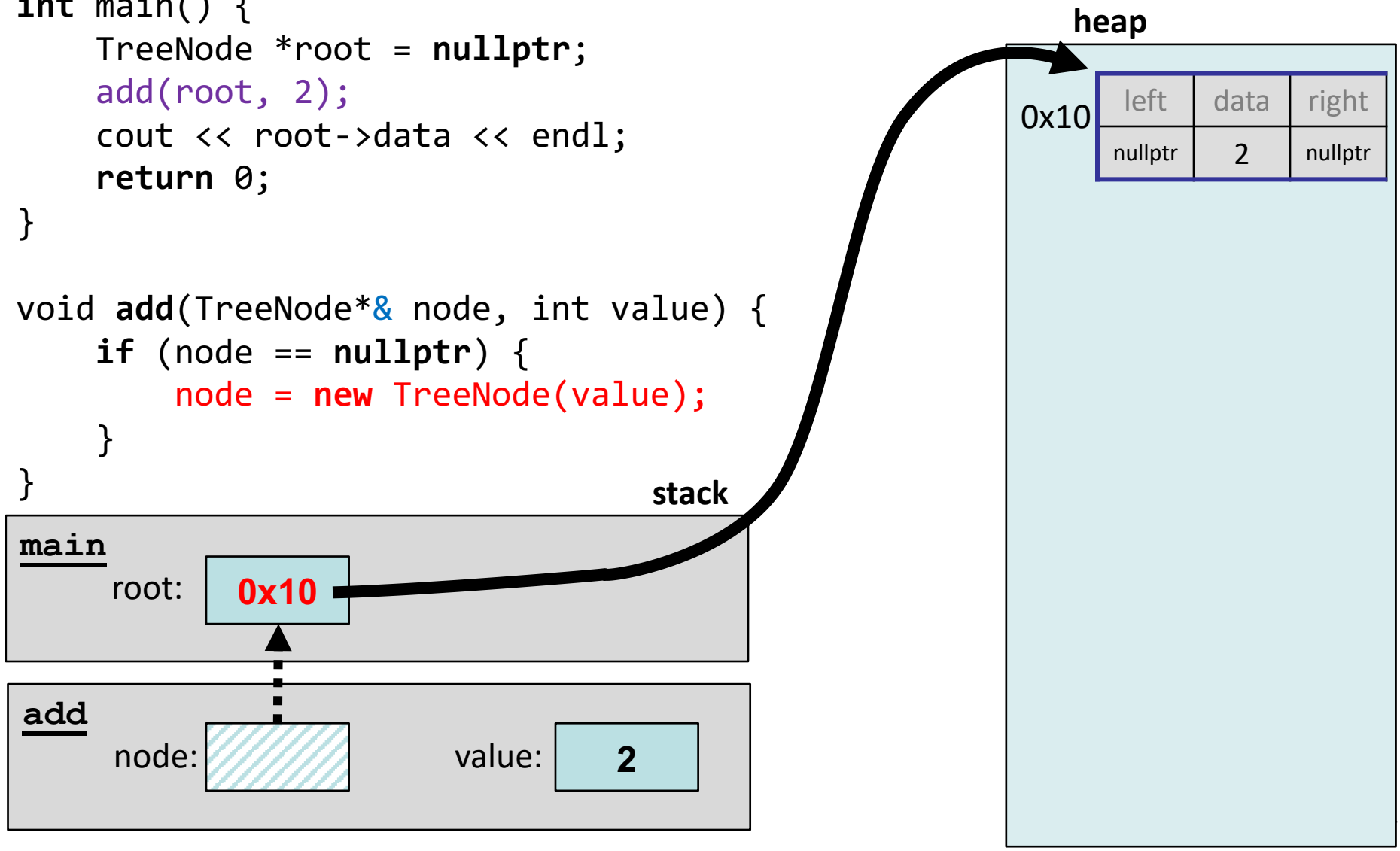
```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```





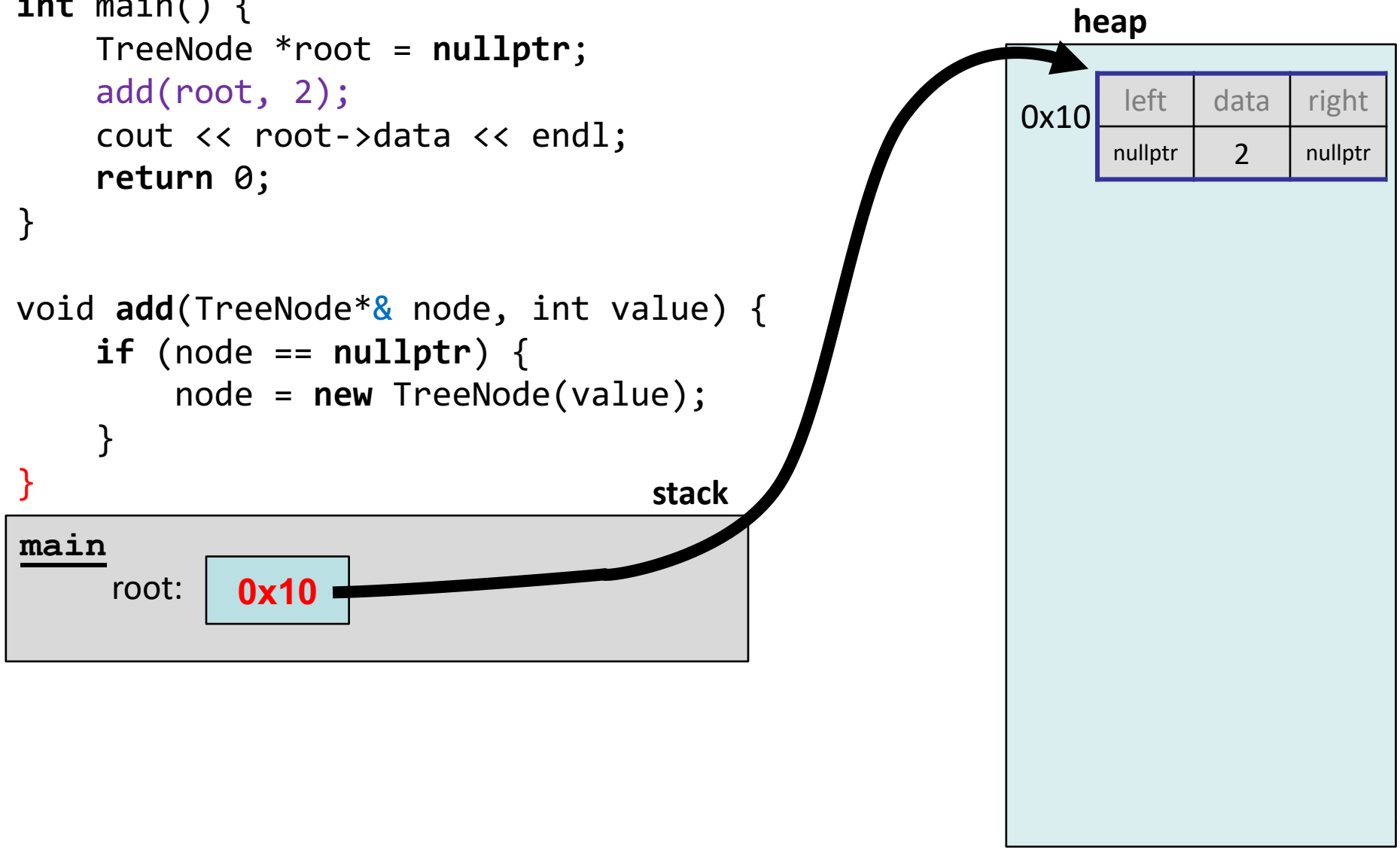
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



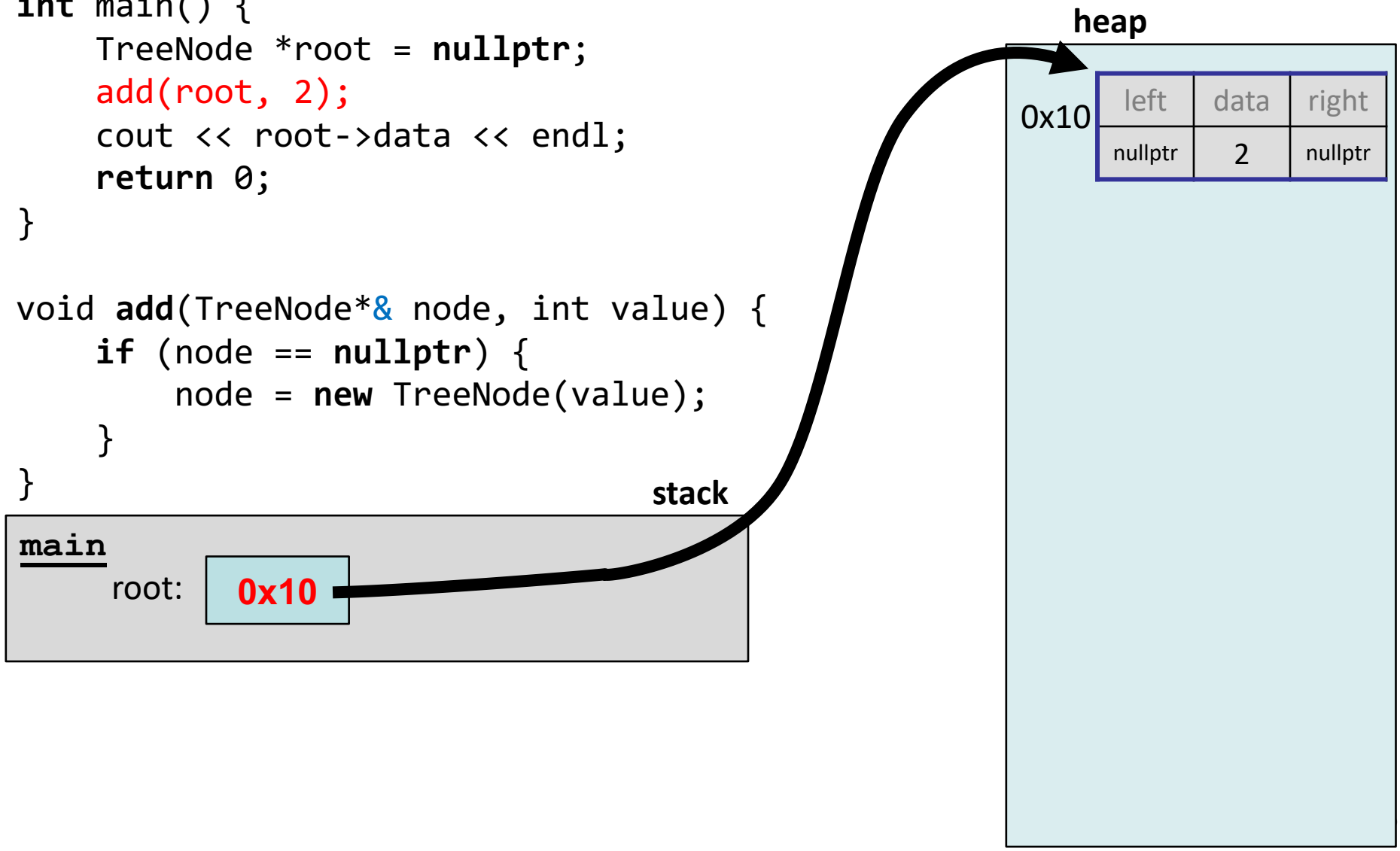
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



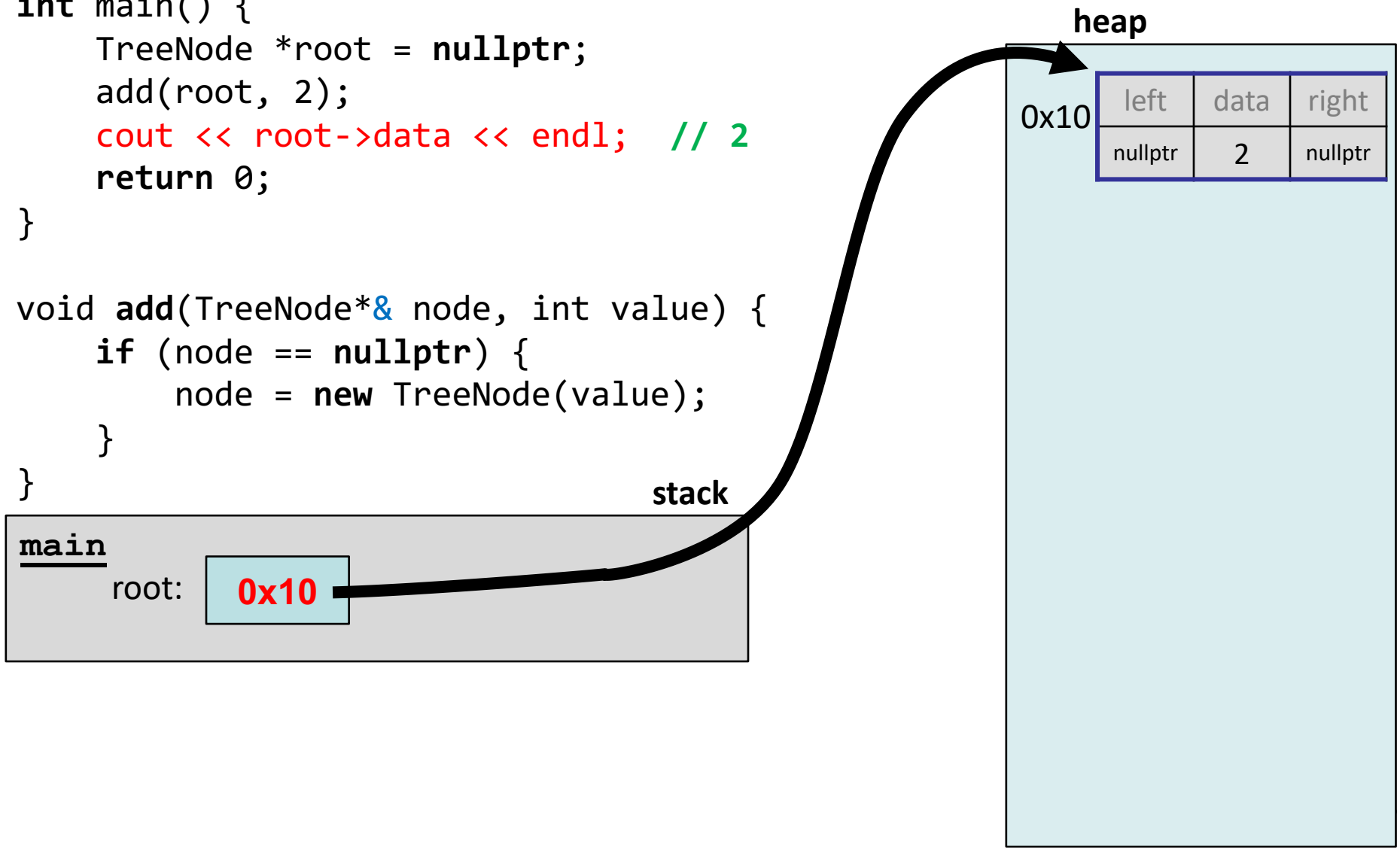
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



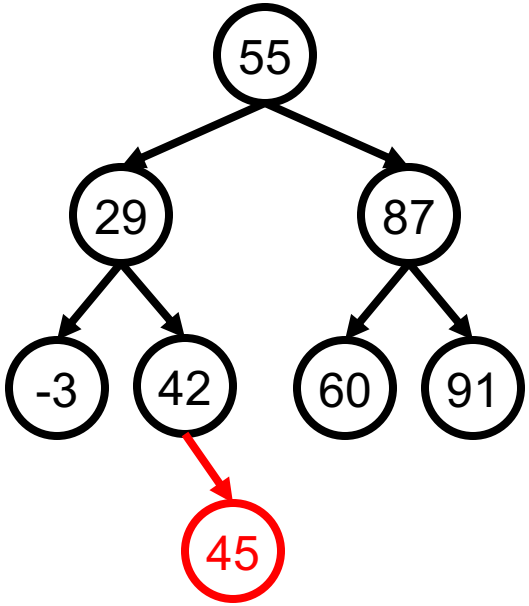
Creating a List

```
int main() {
    TreeNode *root = nullptr;
    add(root, 2);
    cout << root->data << endl;  // 2
    return 0;
}

void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    }
}
```



Exercise: add

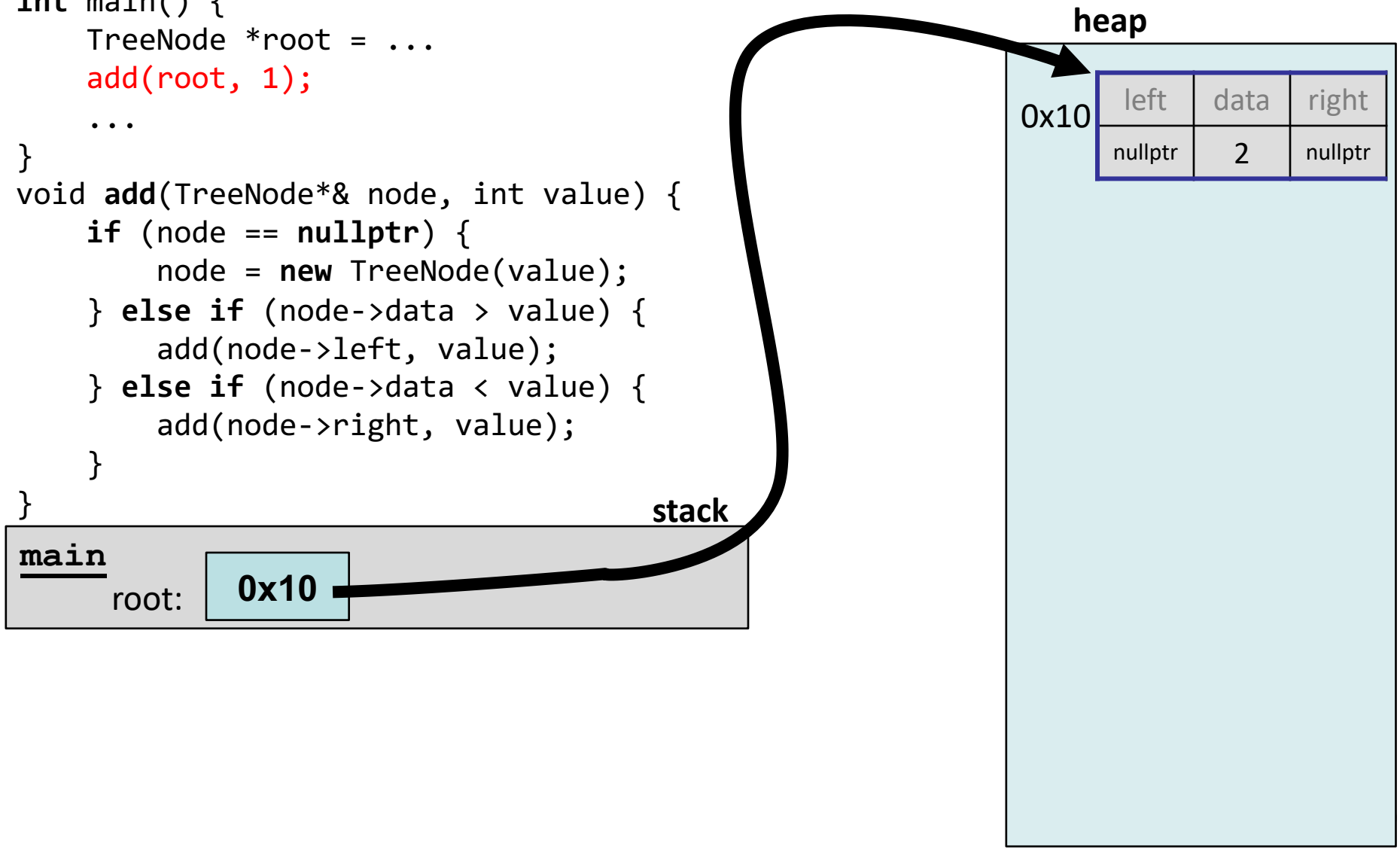
```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



Must pass the current node *by reference* for changes to be seen. Why?

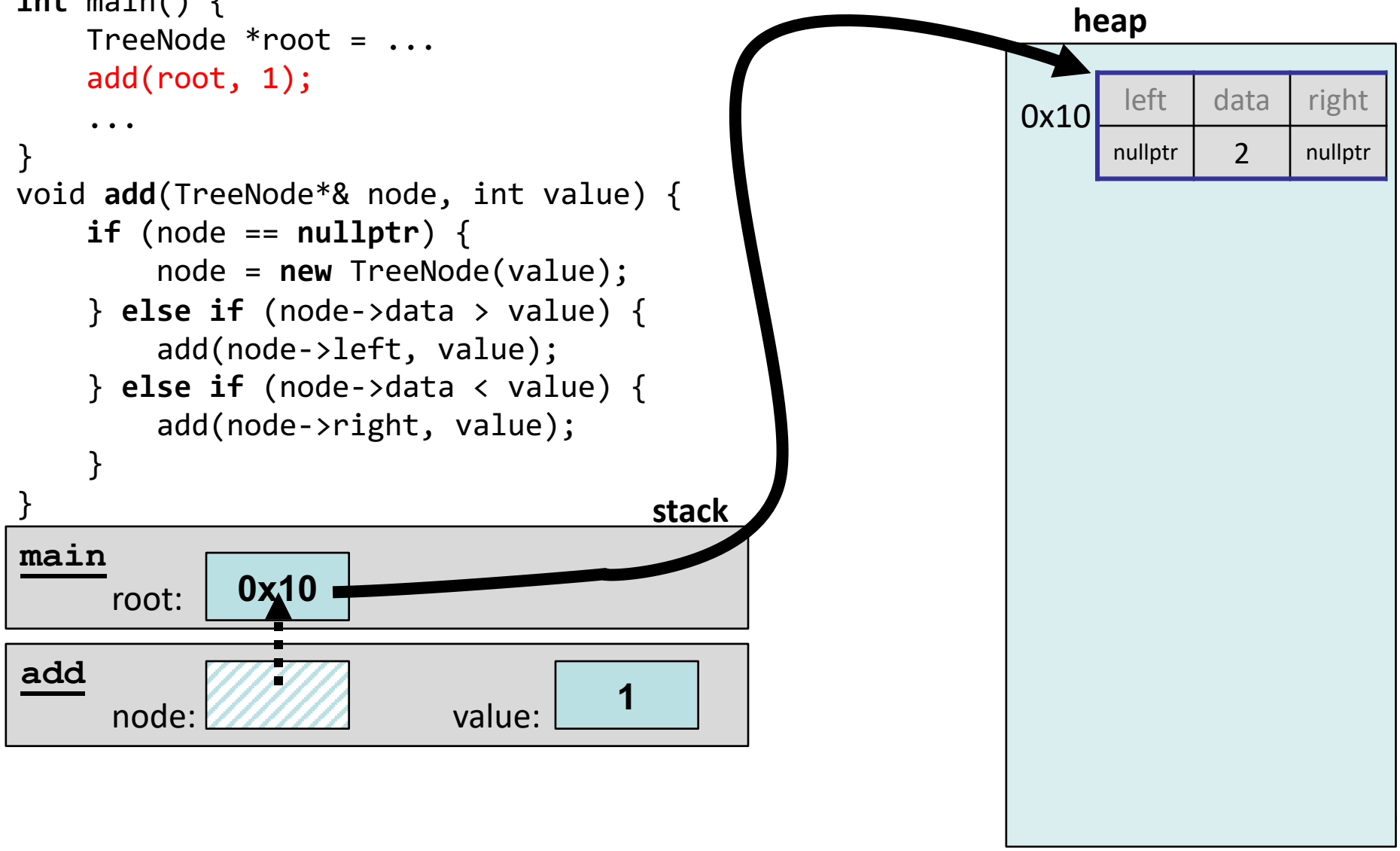
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



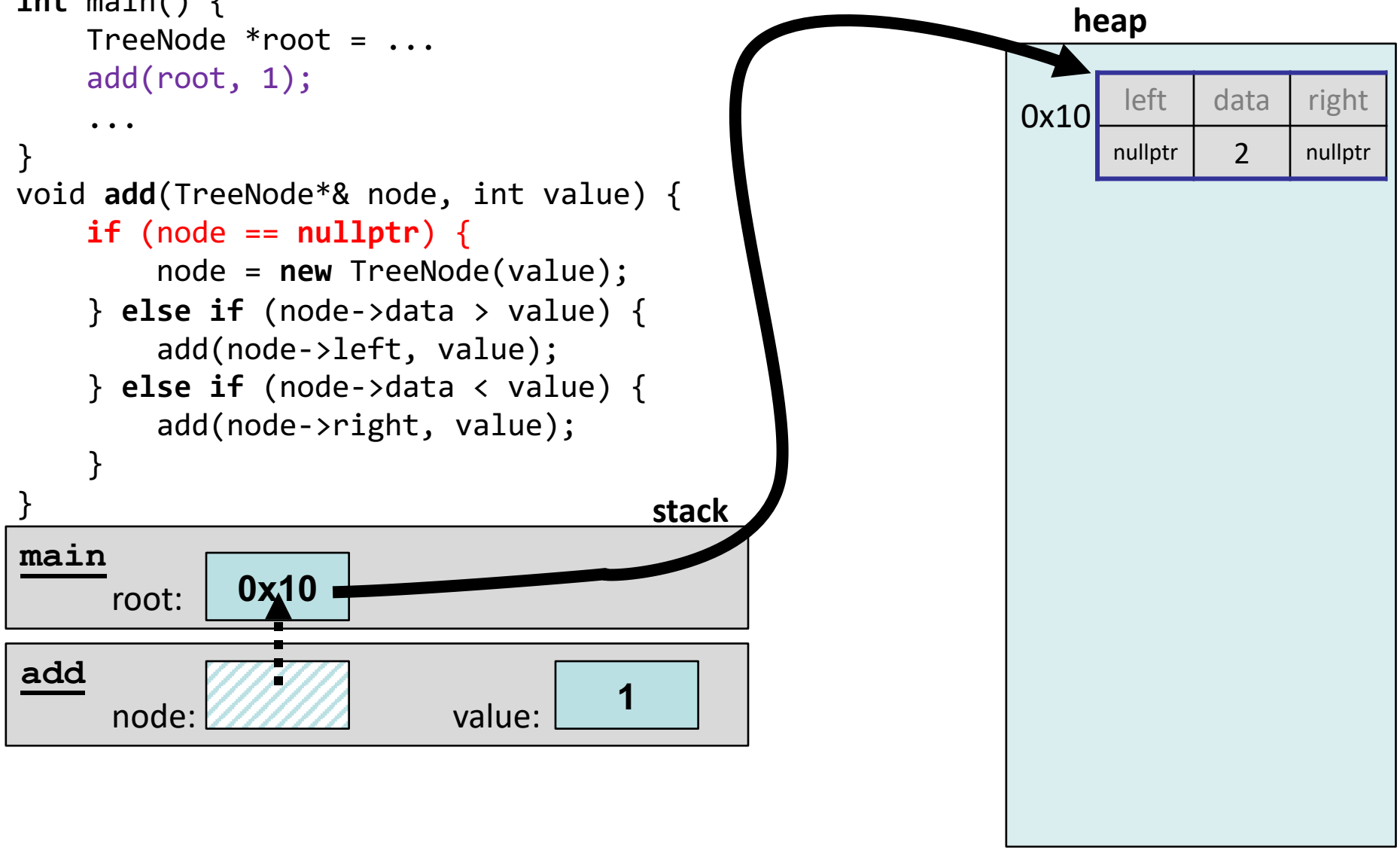
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



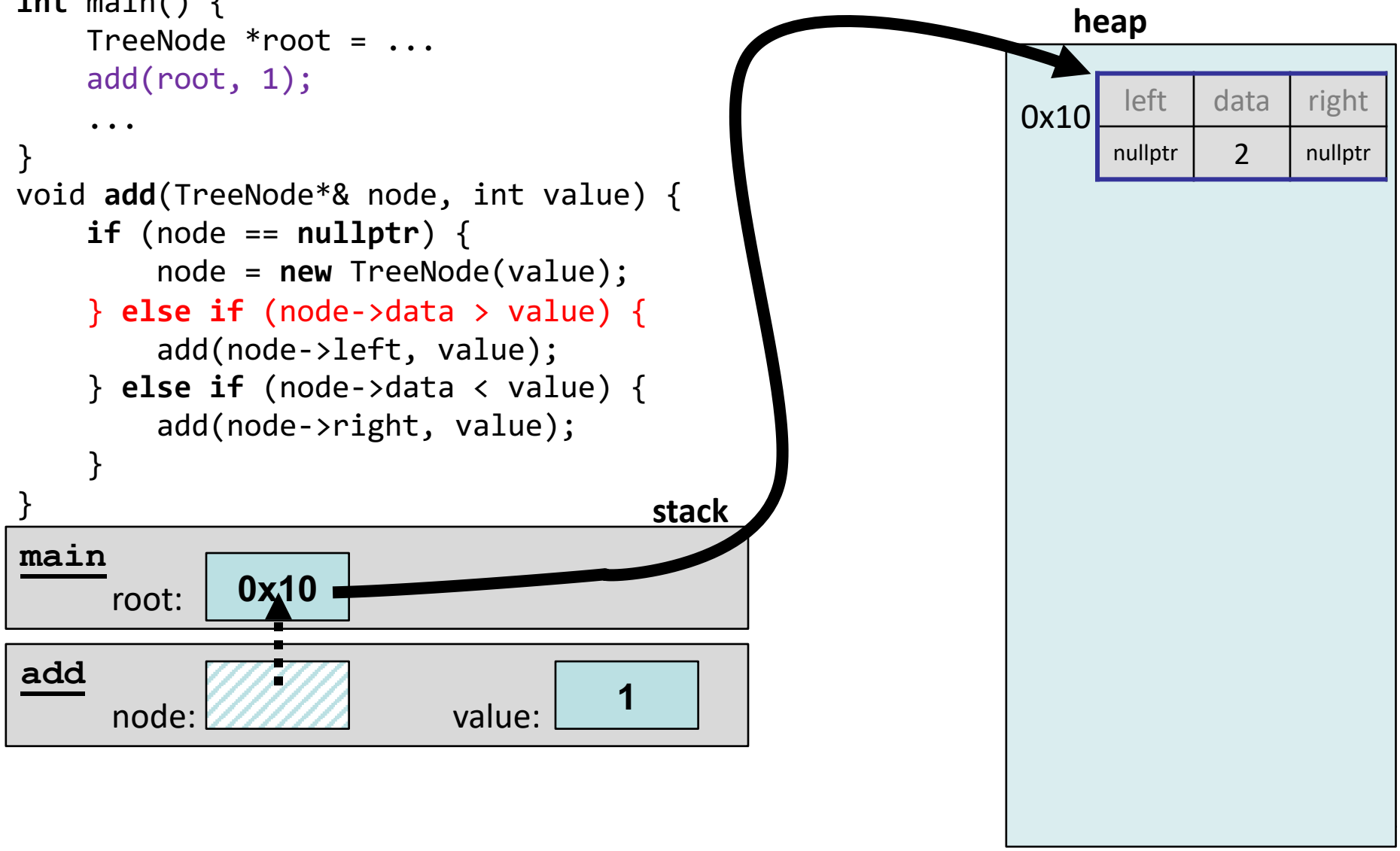
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



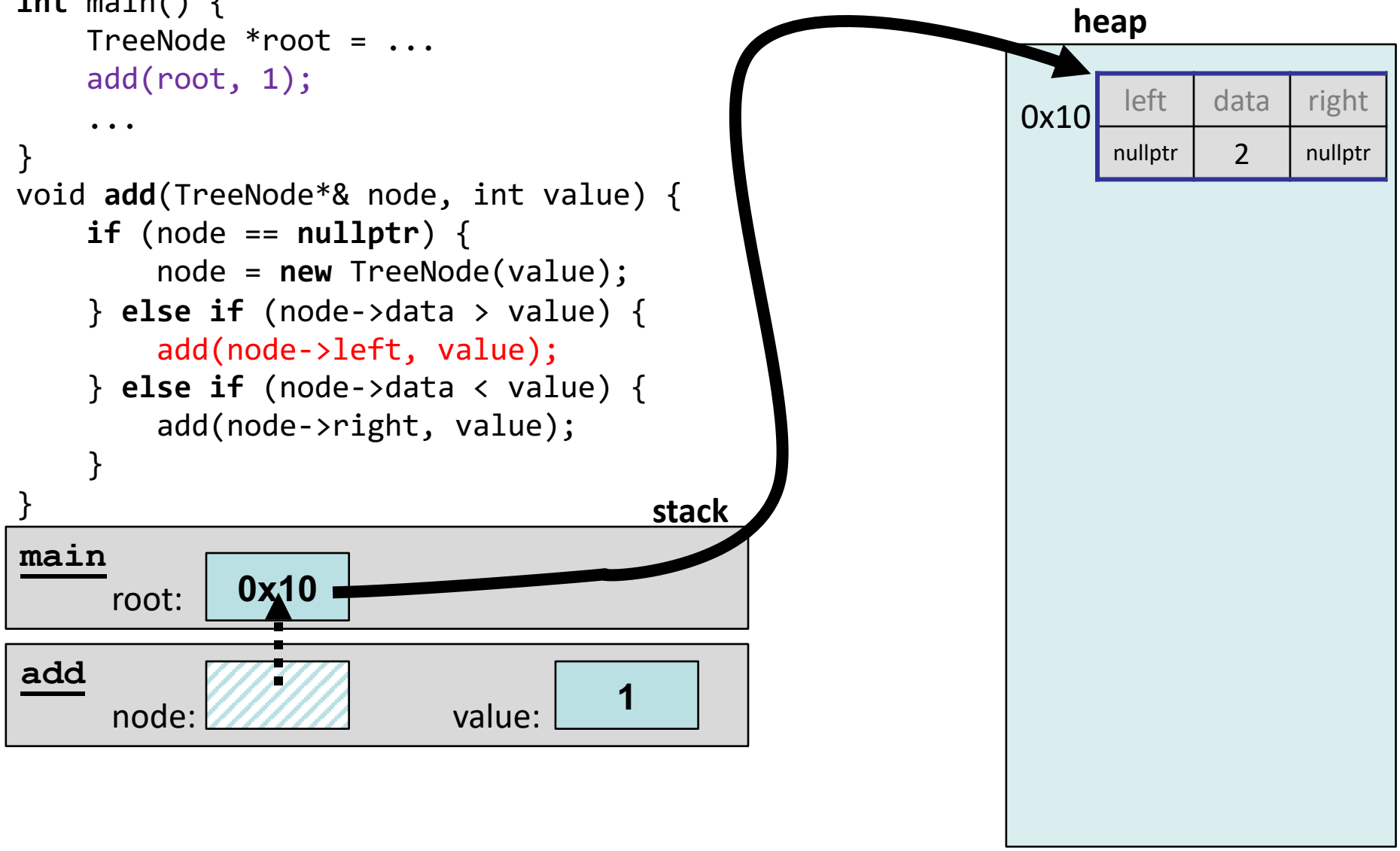
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



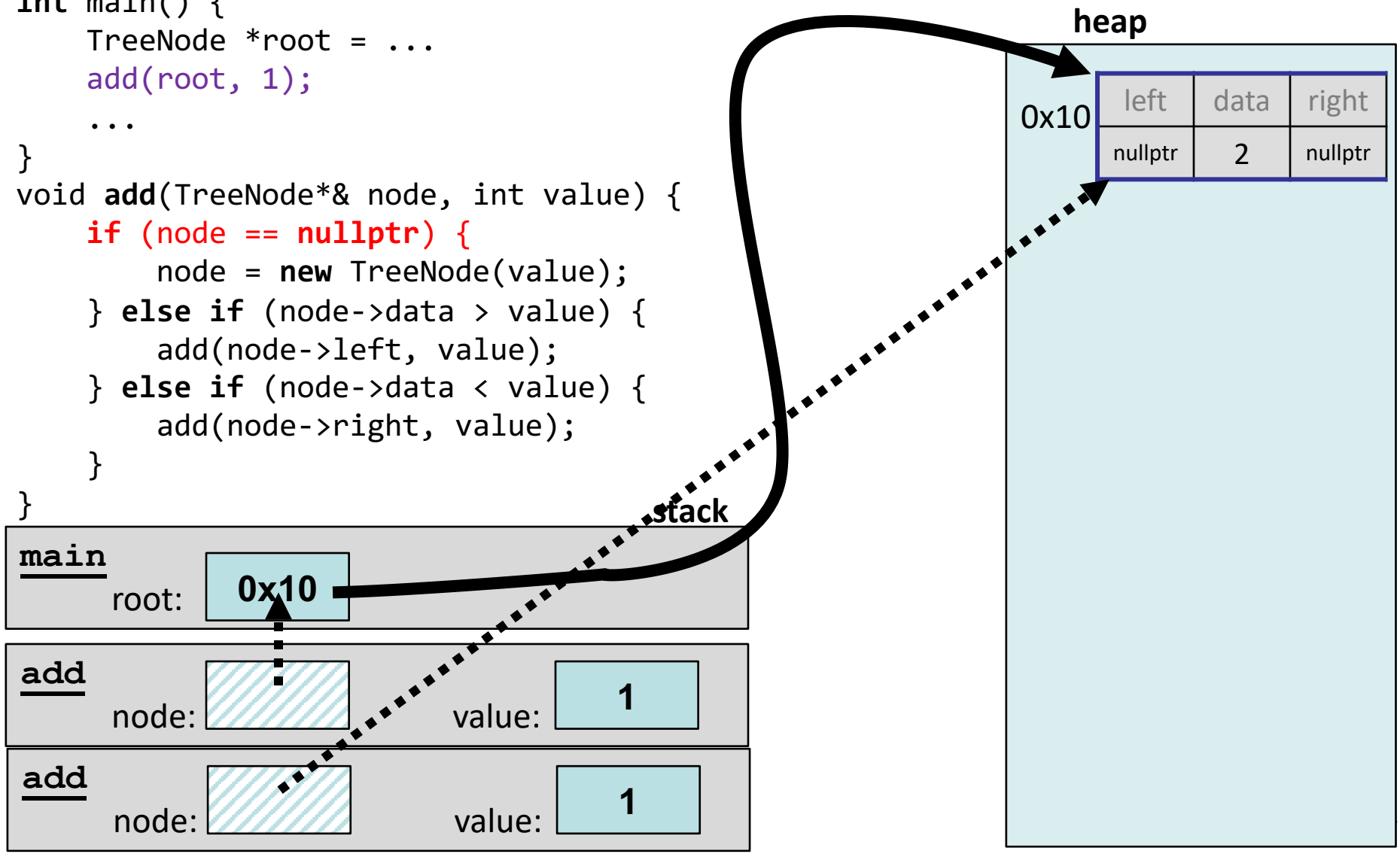
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



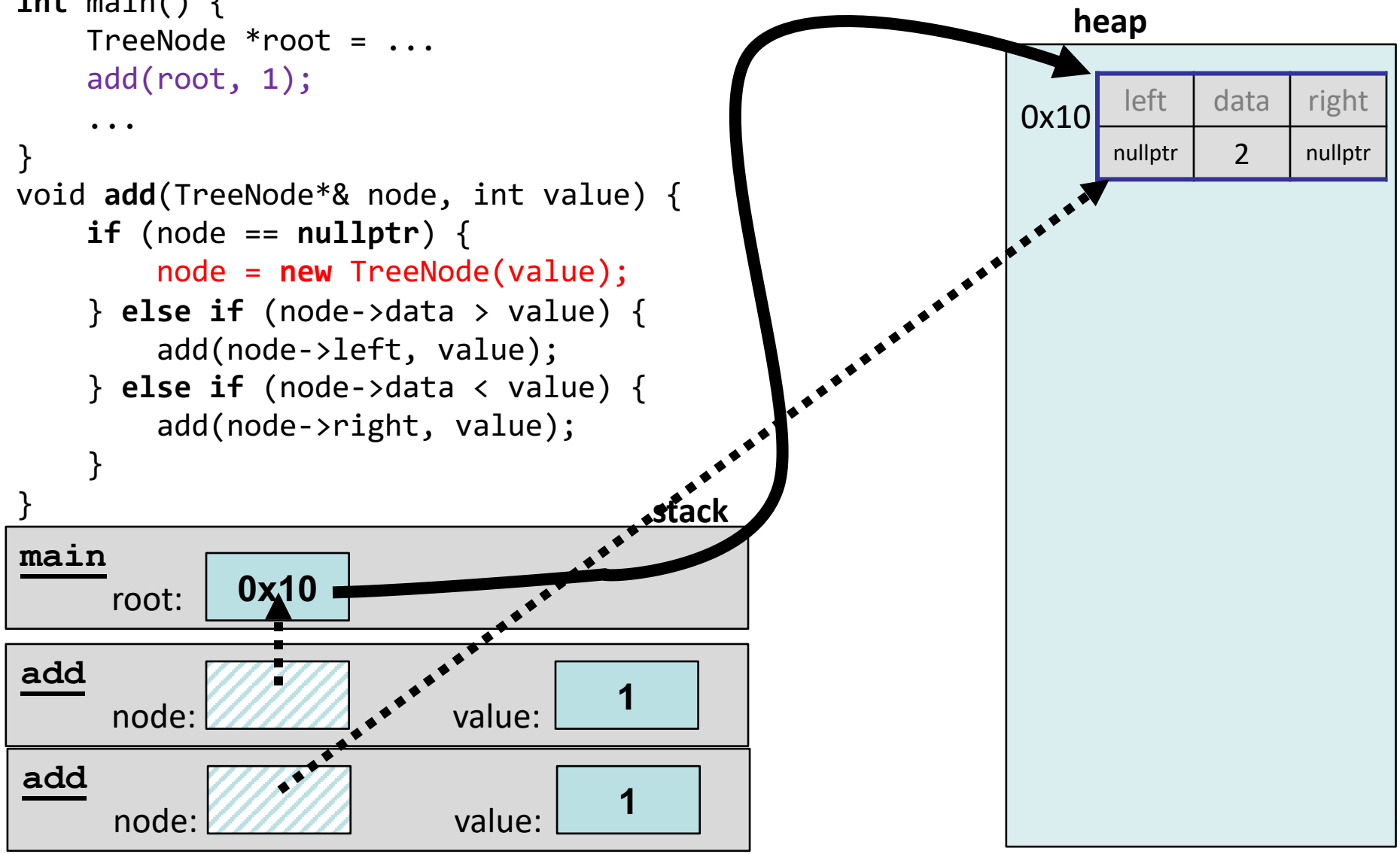
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



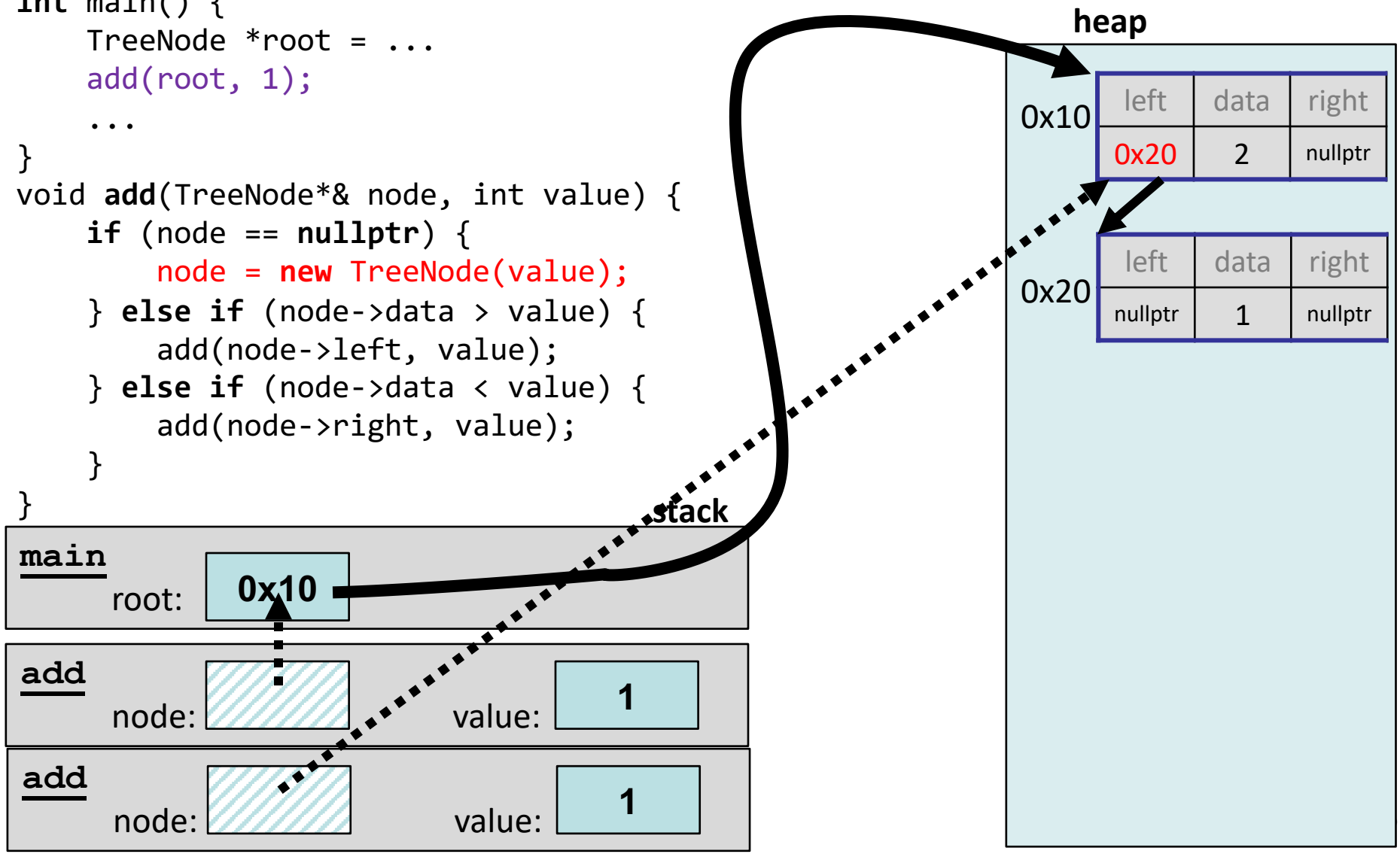
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



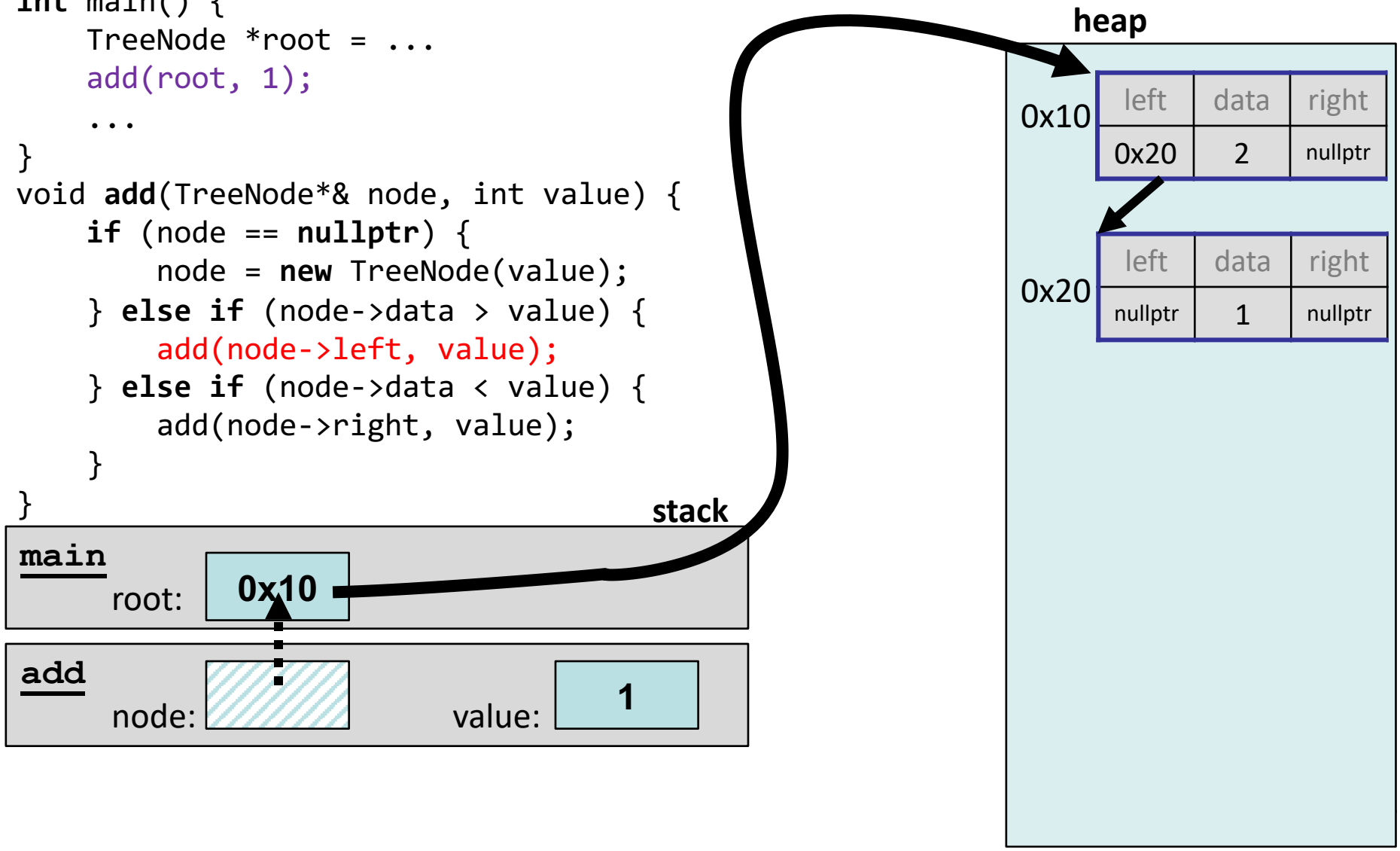
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



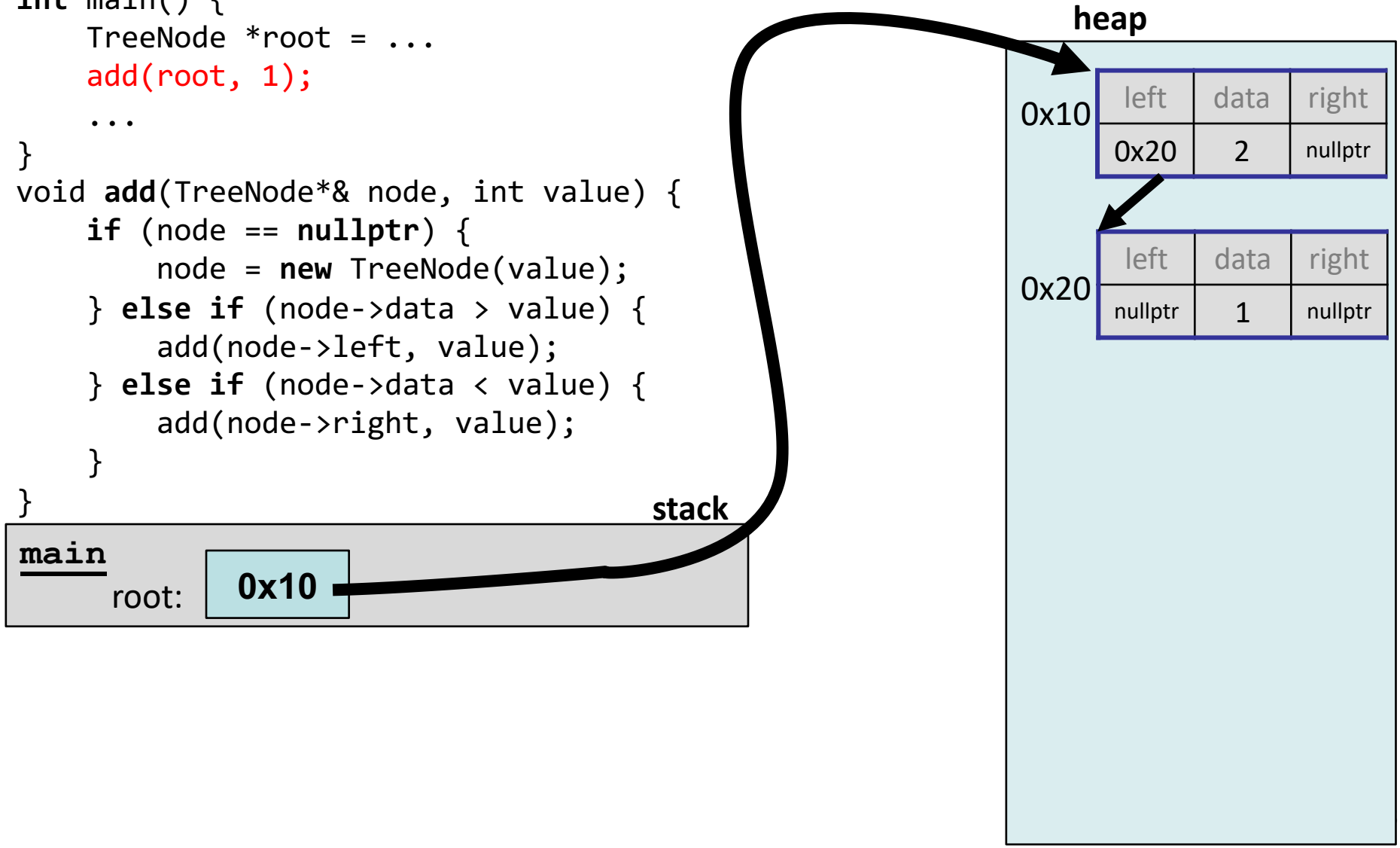
Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



Creating a List

```
int main() {
    TreeNode *root = ...
    add(root, 1);
    ...
}
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```

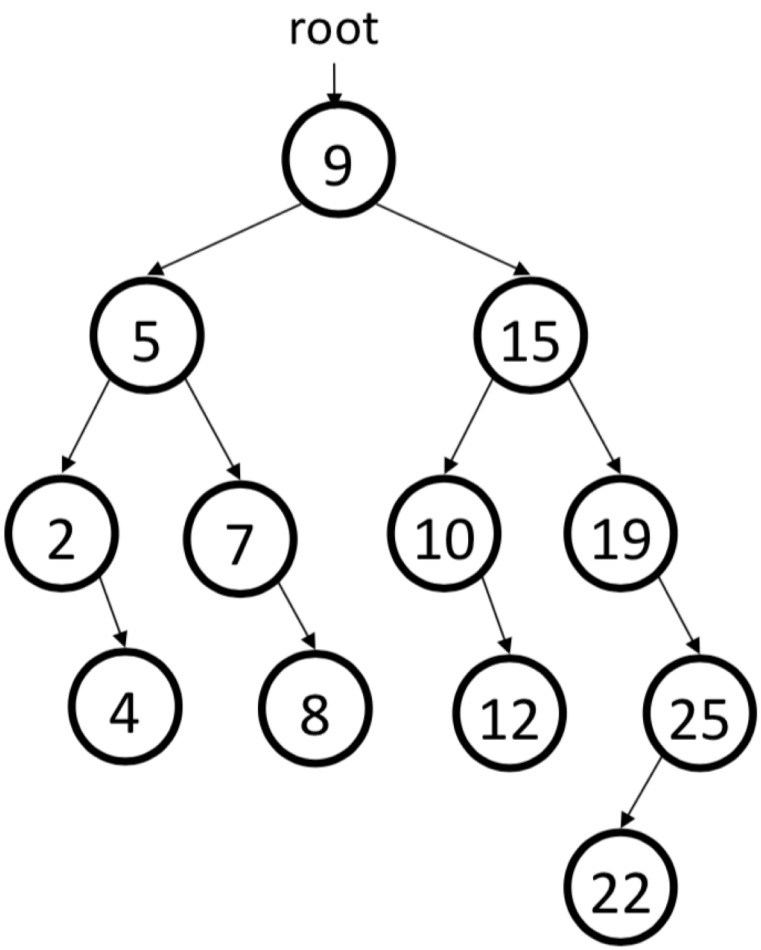


Plan For Today

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

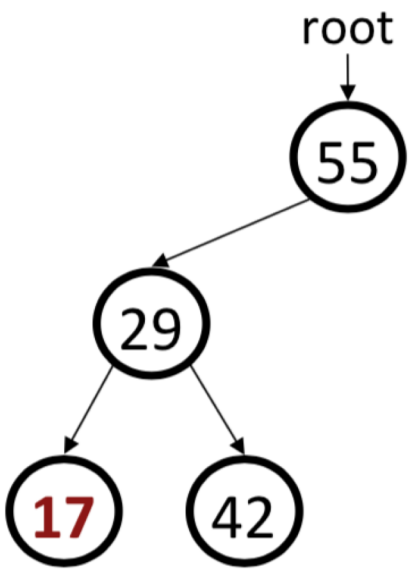
Removing from a BST

- Suppose we want to **remove** values from the BST below.
  - Removing a leaf like 4 or 22 is easy.
  - What about removing 2? 19?
  - How can you remove a node with two large subtrees under it, such as 15 or 9?
- What is the general algorithm?

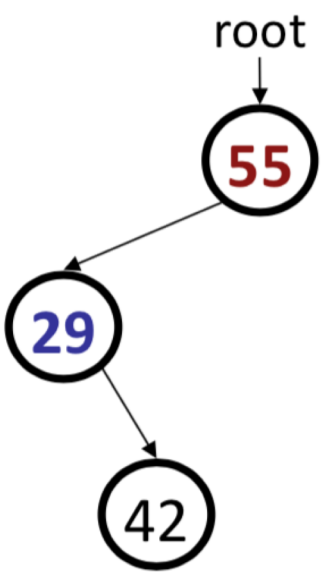


Removing from a BST

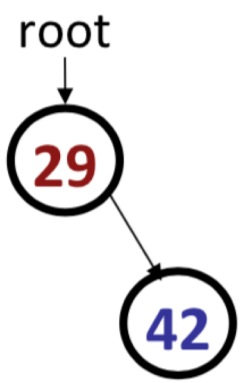
1. a **leaf**: Replace with `nullptr`
2. a node with a **left child only**: Replace with left child
3. a node with a **right child only**: Replace with right child



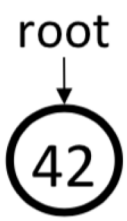
`remove(root, 17);`



`remove(root, 55);`

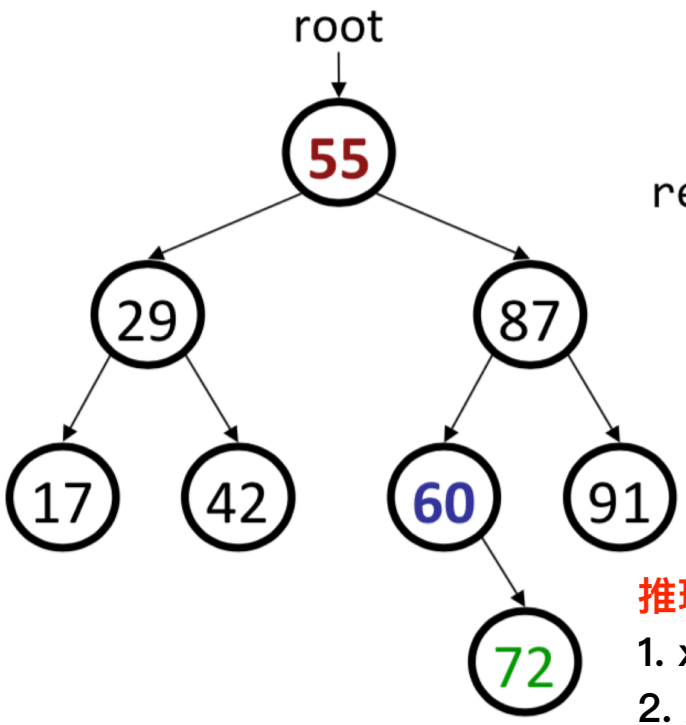


`remove(root, 29);`

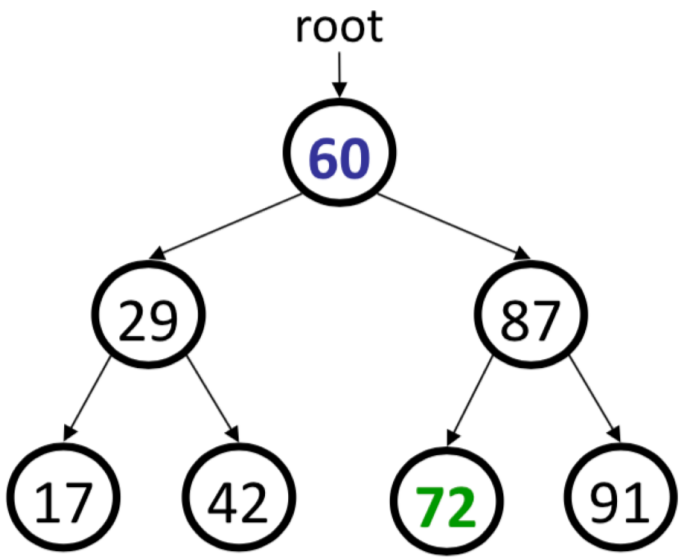


Removing from a BST

4. a node with **both** children: replace with **min from right**
(replacing with **max from left** would also work)



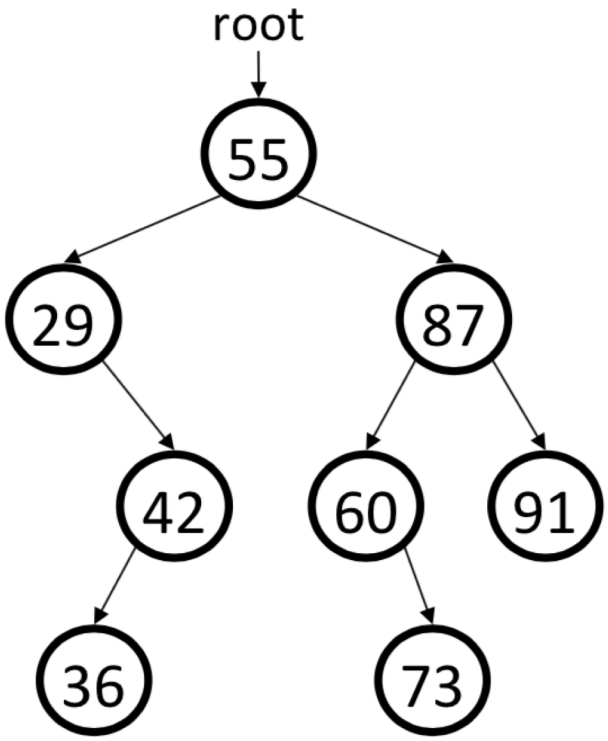

```
remove(root, 55);
```



推理：找1个值x替换55 st
1. x > 左边所有值
2. x < 右边所有值
因为右边subtree的值一定大于左边，可从右边的subtree里面找–>又要小于右边所有的值，所以找右边的min

Exercise: remove

- Add a function **`remove`** that accepts a root pointer and removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.
  - `remove(root, 73);`
  - `remove(root, 29);`
  - `remove(root, 87);`
  - `remove(root, 55);`



Plan For Today

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

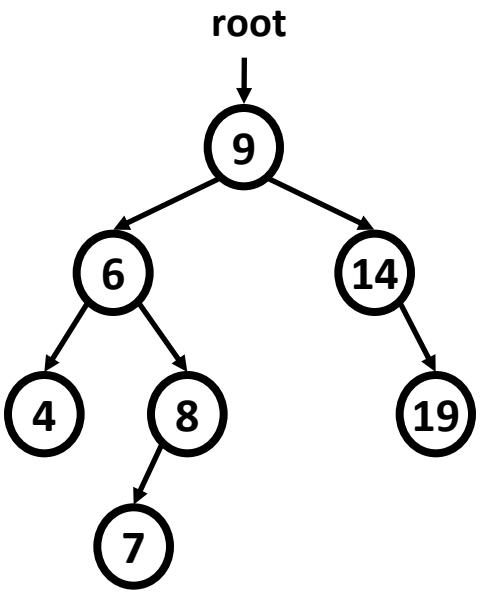
Announcements

- **HW6** (MiniBrowser 2) will be out later today – it is all about trees!
  - More efficient LineManager
  - Autocomplete
  - Compression

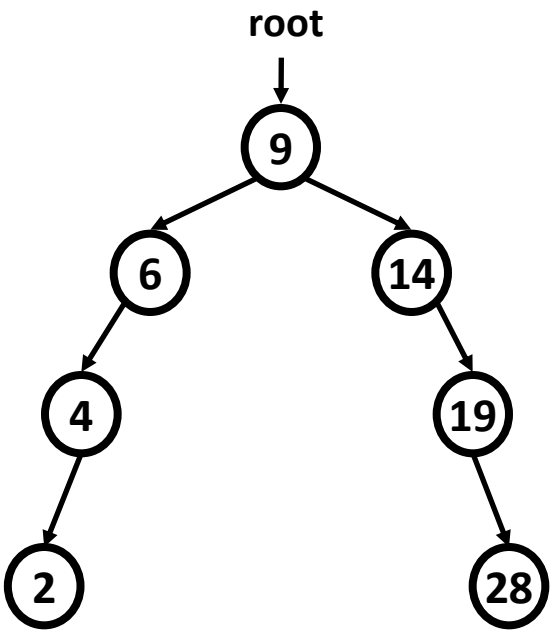
Plan For Today

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

Balanced Trees

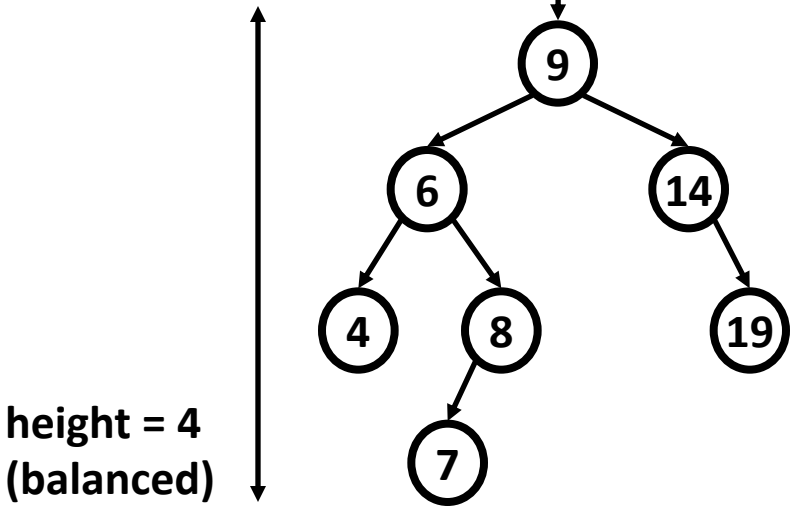


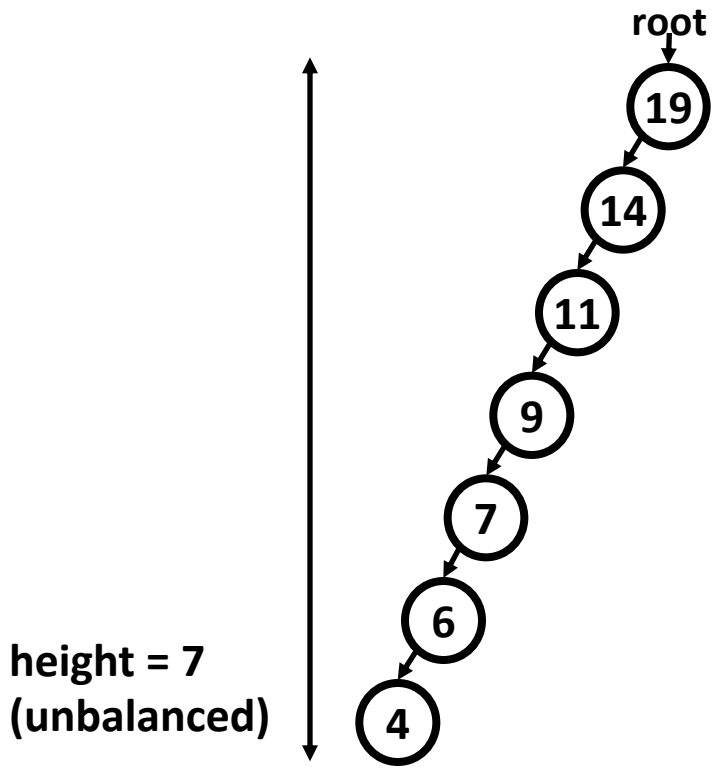
Balanced Trees



Trees and balance

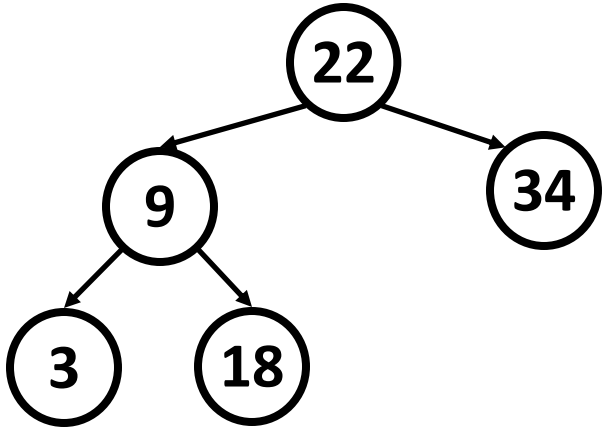
- **balanced tree**: One where for every node R, the height of R's subtrees differ by at most 1, and R's subtrees are also balanced.
  - Balanced tree's height is roughly $\log_2 N$. Unbalanced is closer to N.





BST balance question

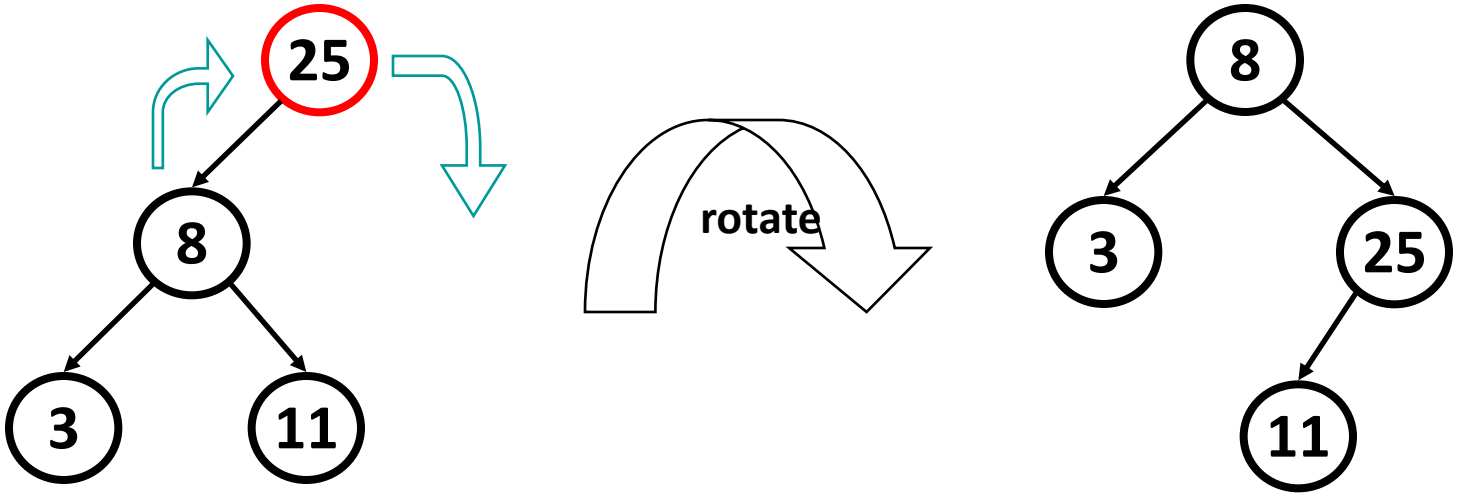
- Adding the following nodes to an empty BST in the following order produces the tree at right: 22, 9, 34, 18, 3.



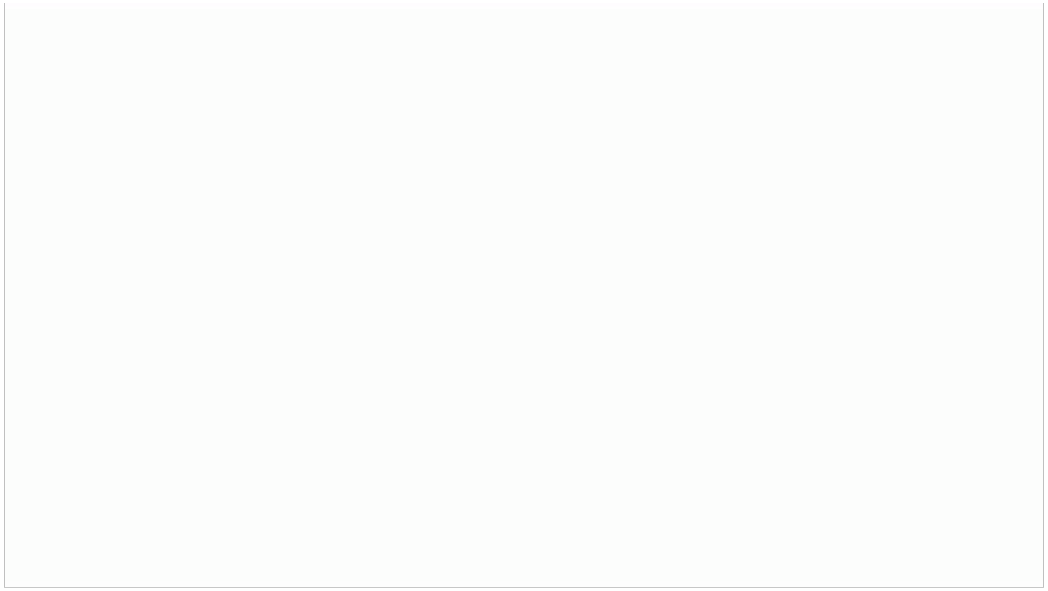
- **Q:** What is an order in which we could have added the nodes to produce an unbalanced tree?
  - **A.** 18, 9, 34, 3, 22
  - **B.** 9, 18, 3, 34, 22
  - **C.** 9, 22, 3, 18, 34
  - **D.** none of the above

AVL Trees

- **AVL tree**: A binary search tree that uses modified add and remove operations to stay balanced as its elements change.
  - *basic idea:* When nodes are added/removed, repair tree shape until balance is restored.
    - rebalancing is O(1); overall tree maintains an O(log *N*) height

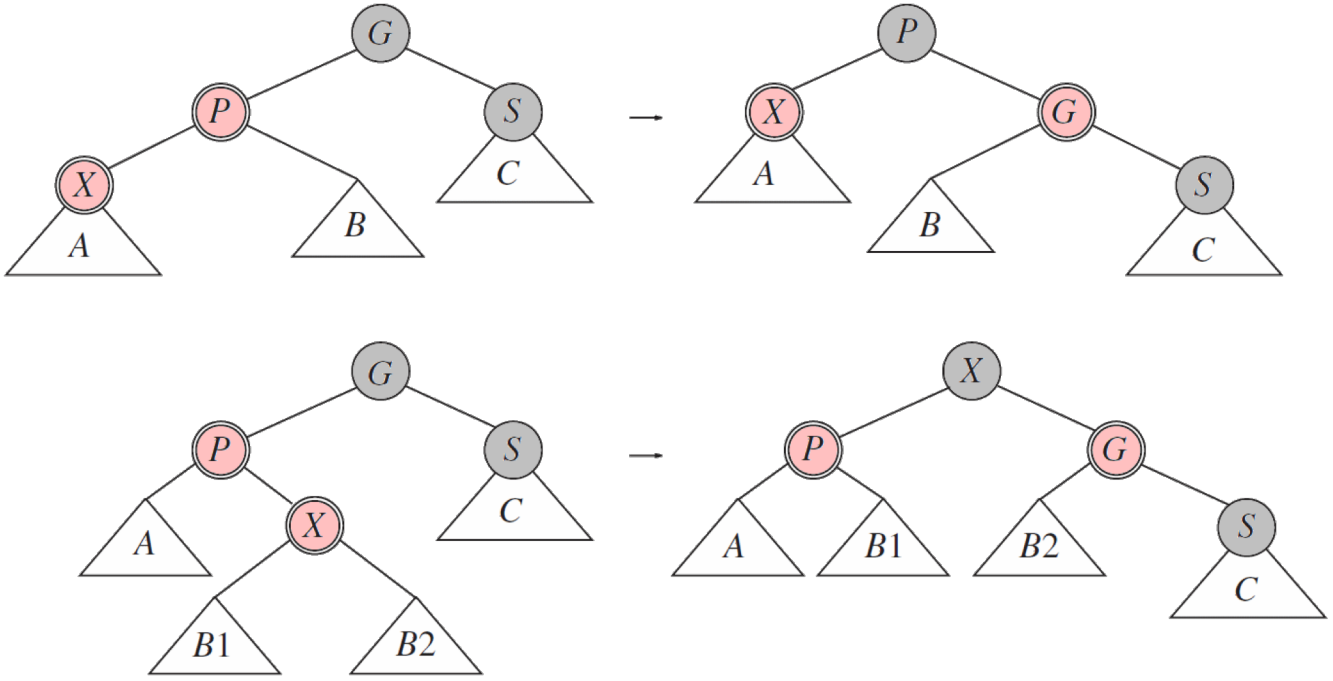


AVL Trees



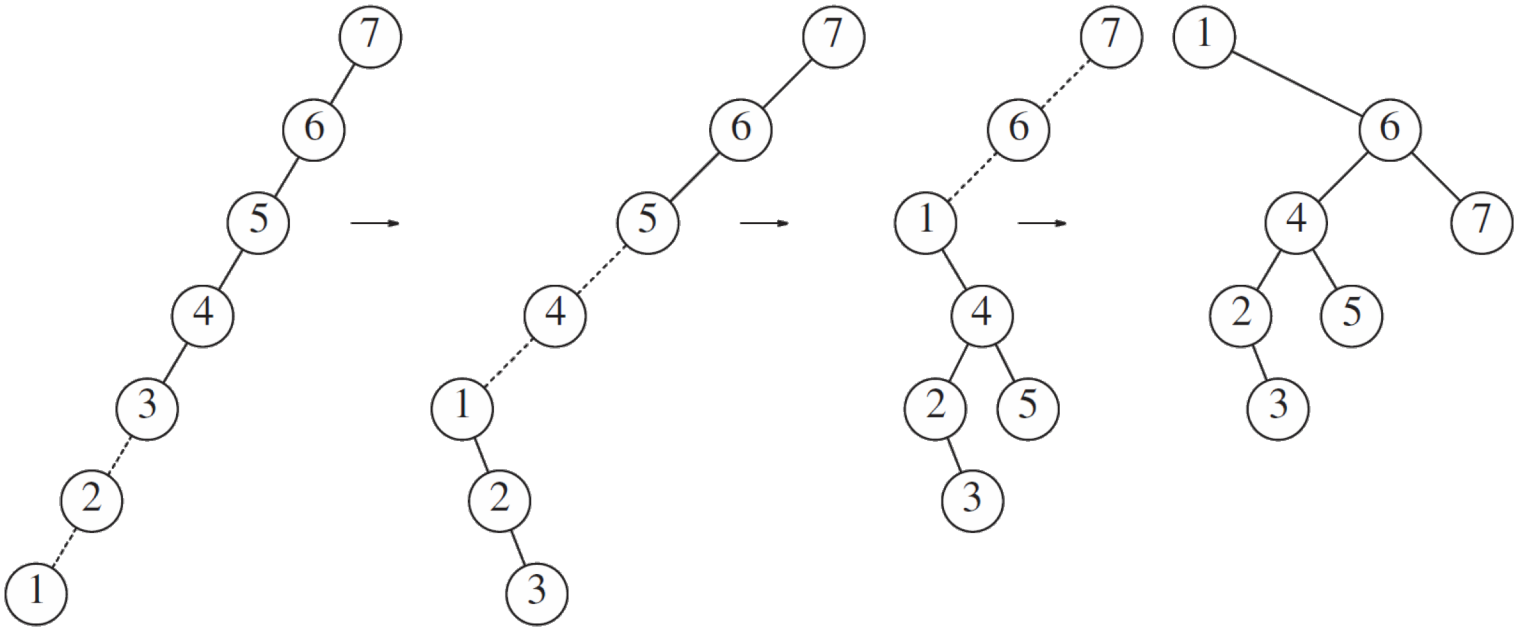
Red-Black Trees

- **red-black tree:** Gives each node a "color" of red or black. (video)
  - Root is black. Root's direct children are red. All leaves are black.
  - If a node is red, its children must all be black.
  - Every path downward from a node to the bottom must contain the same number of "black" nodes.



Splay Trees

- **splay tree:** Rotates each element you access to the top/root
  - very efficient when that element is accessed again (happens a lot)
  - easy to implement and does not need height field in each node



Balanced Trees

- AVL Trees
- Red-Black Trees
- Splay Trees

Take CS161 and CS166 for more!

Recap

- Binary Search Trees
  - Adding
  - Removing
- Announcements
- Balanced Trees

Next time: Tries and Graphs