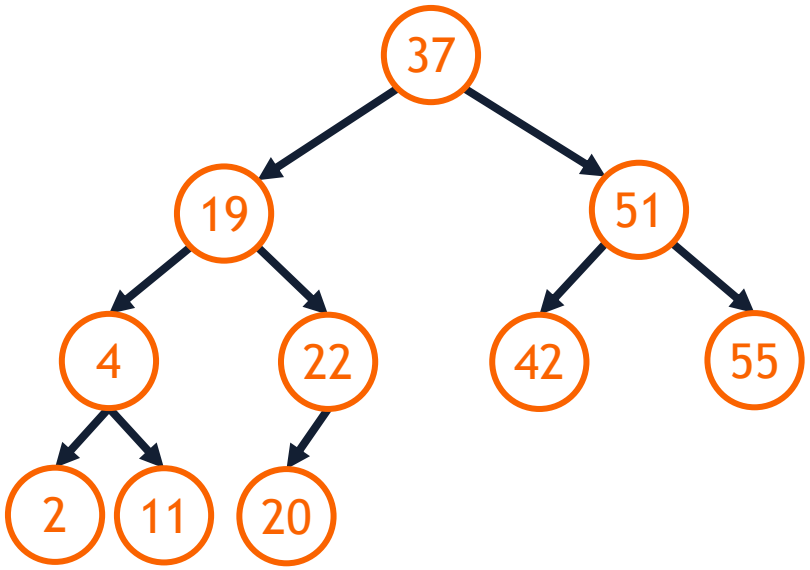
Binary Search Tree (BST)

Prof. Wade Fagen-Ulmschneider

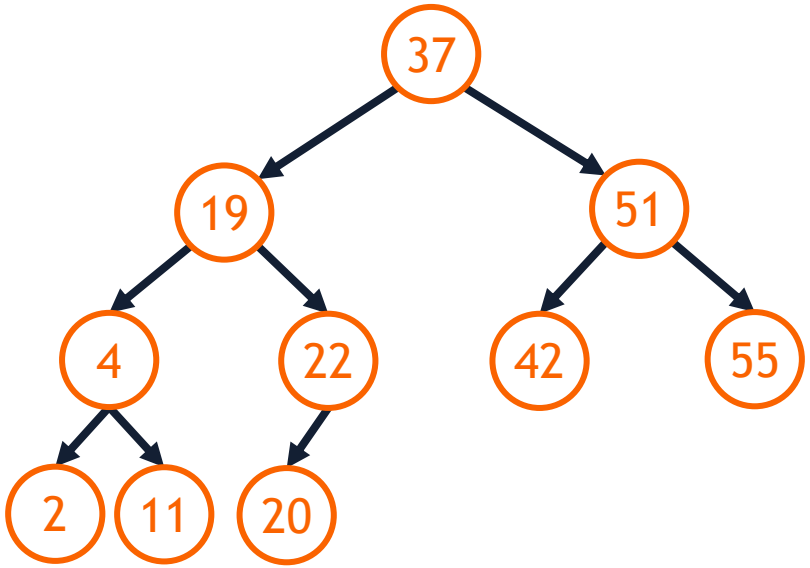
A binary search tree (BST) is an **ordered** binary tree capable of being used as a search structure:

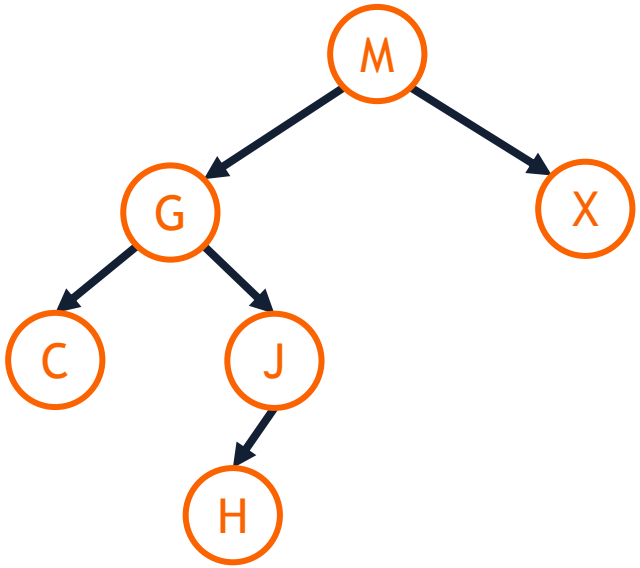


BST Order Property

A binary tree is a BST if **for every node in the tree:**

- Nodes in the **left** subtree are **less** than itself.
- Nodes in the **right** subtree are **greater** than itself.





Dictionary

A dictionary associates a **key** with **data**:

Your login email ➔ Your profile data
Your phone number ➔ Your phone record
A website URL ➔ The webpage
Your street address ➔ Your home

Dictionary ADT

find ➔ Finds the data associated with a key in the dictionary

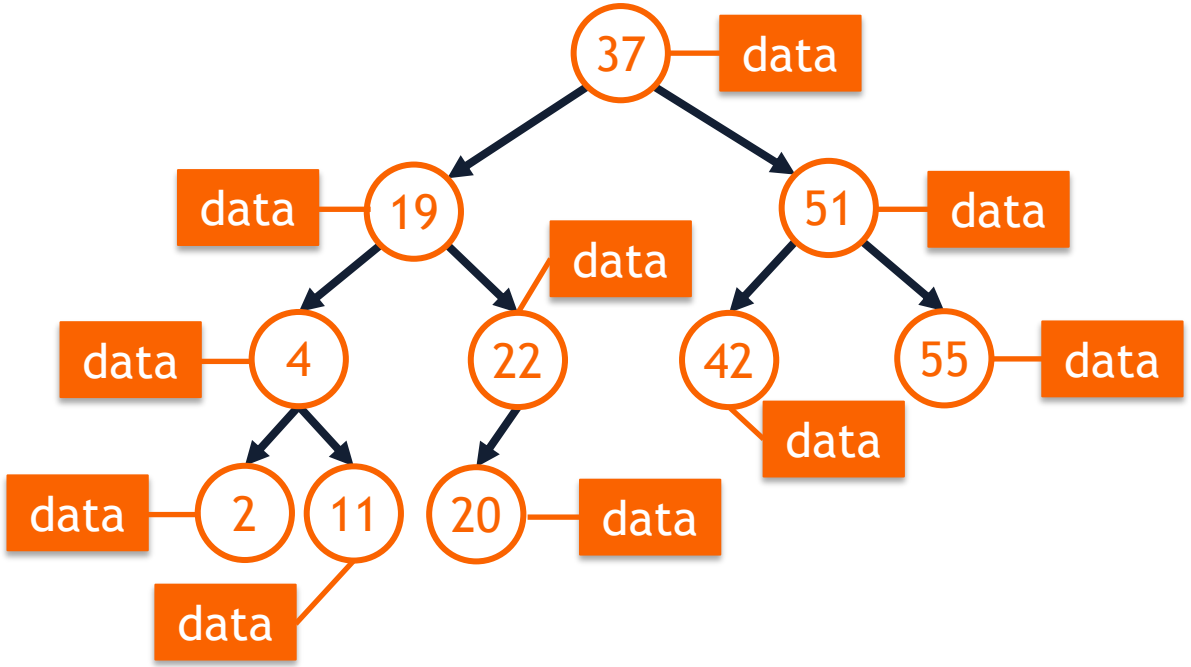
insert ➔ Adds a key/data pair to the dictionary

remove ➔ Removes a key from the dictionary

empty ➔ Returns true if the dictionary is empty

BST-Based Dictionary

A BST used to implement a dictionary will store both a key and data at every node:



bst/Dictionary.h

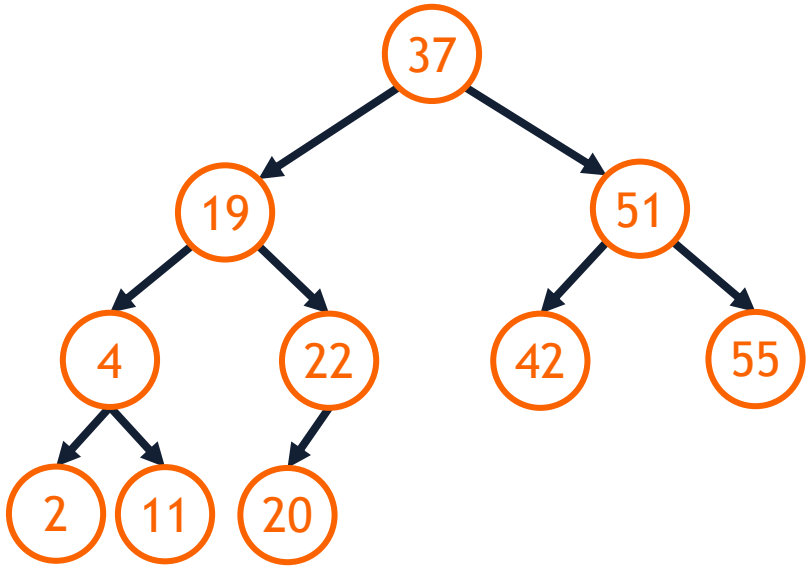
```
10 template <typename K, typename D>
11 class Dictionary {
12   public:
13     Dictionary();
14     const D & find(const K & key);
15     void insert(const K & key, const D & data);
16     const D & remove(const K & key);
17
18   private:
19     class TreeNode {
20       public:
21         const K & key;
22         const D & data;
23         TreeNode *left, *right;
24         TreeNode(const K & key, const D & data)
                   : key(key), data(data), left(nullptr), right(nullptr) { }
25     };
26
27   TreeNode *head_;
…    …
```

Example: Finding in a BST

find(42)

find(11)

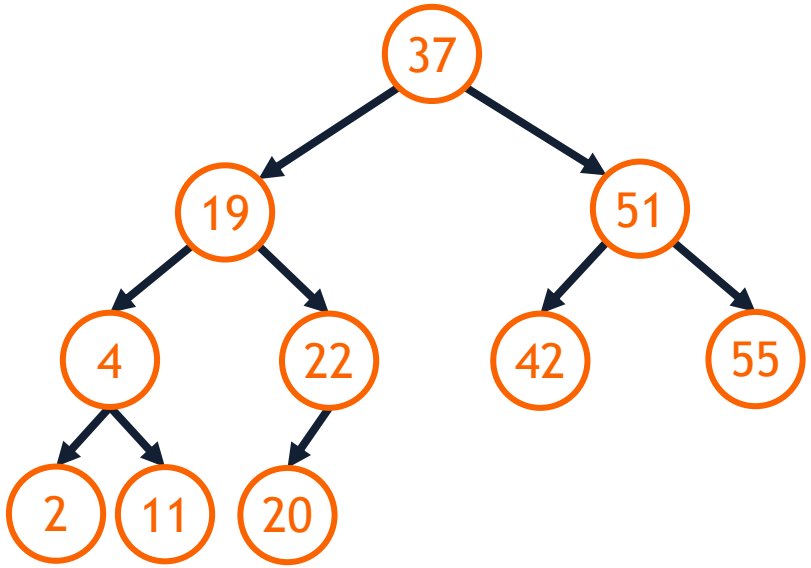
find(17)



Runtime of BST::find

Worst-case outcome of find:

height of tree



bst/Dictionary.cpp

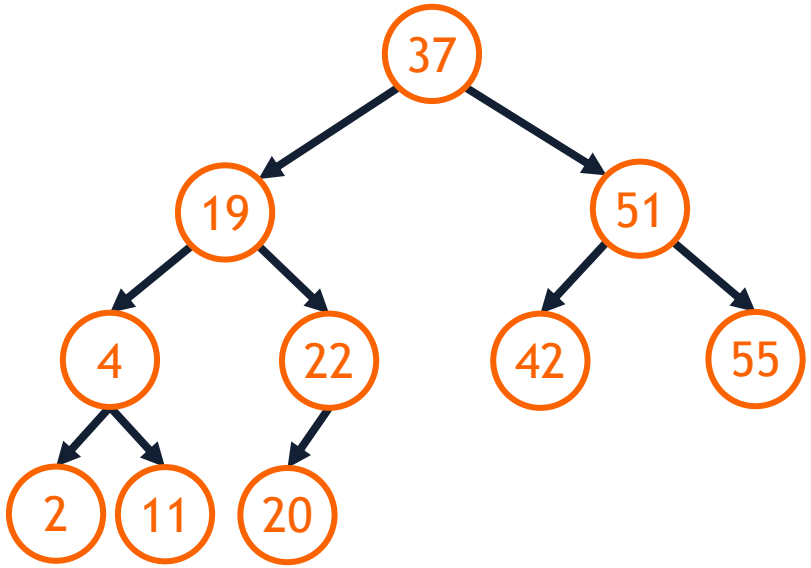
```
18 template <typename K, typename D>
19 const D & Dictionary<K, D>::find(const K & key) {
20   TreeNode *& node = _find(key, head_);
21   if (node == nullptr) { throw std::runtime_error("key not found"); }
22   return node->data;
23 }

…  …

60 template <typename K, typename D>
61 typename Dictionary<K, D>::TreeNode *& Dictionary<K, D>::_find(
   const K & key, TreeNode *& cur) const {
62   if       (cur == nullptr)  { return cur; }
63   else if (key == cur->key) { return cur; }
64   else if (key < cur->key)  { return _find(key, cur->left); }
65   else                      { return _find(key, cur->right); }
66 }
```

Insert into a BST

insert(17)

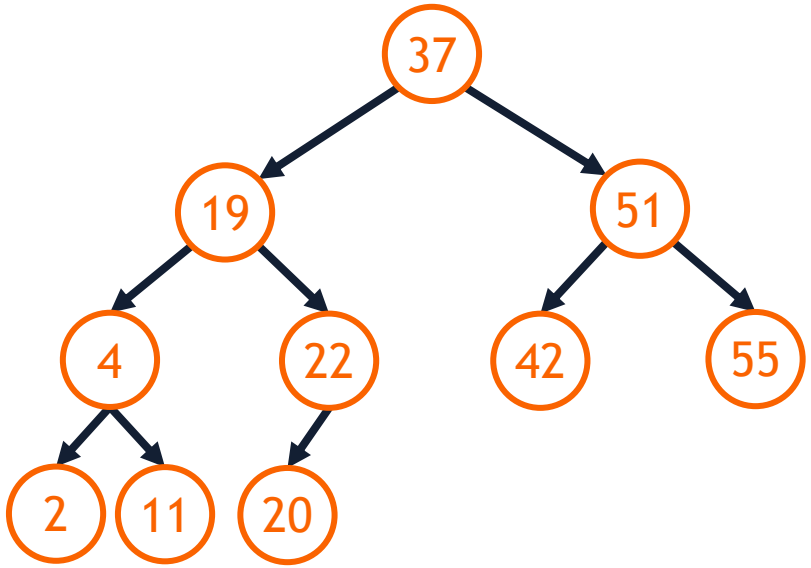


bst/Dictionary.cpp

```
26 /**
27 * insert()
28 * Inserts `key` and associated `data` into the Dictionary.
29 */
30 template <typename K, typename D>
31 void Dictionary<K, D>::insert(const K & key, const D & data) {
32   TreeNode *& node = _find(key, head_);
33   node = new TreeNode(key, data);
34 }
```

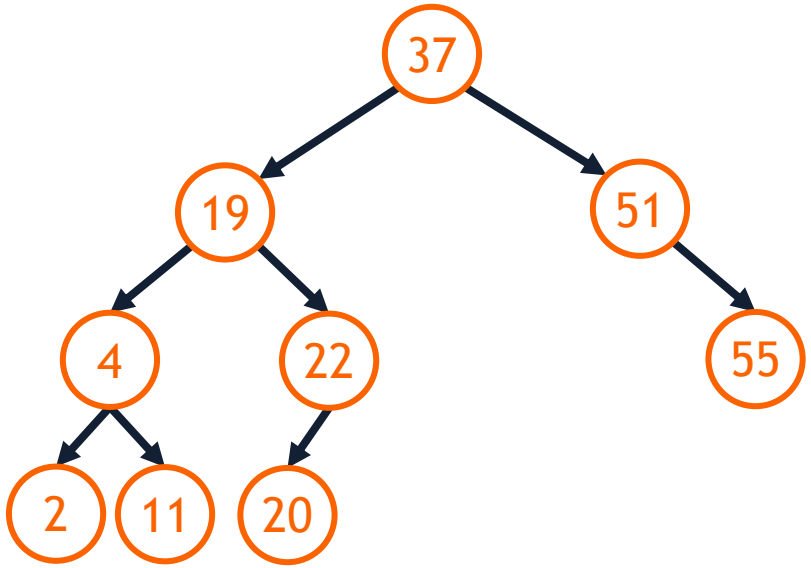
Remove from a BST

remove(42)



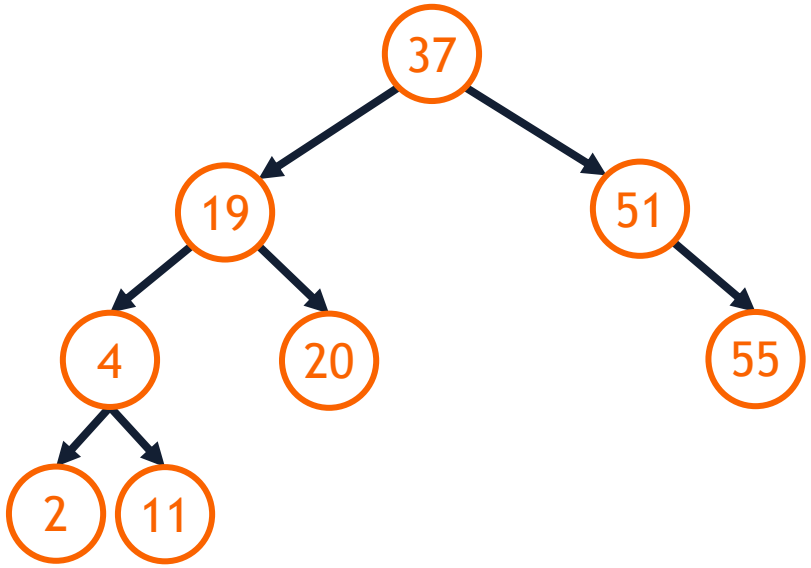
Remove from a BST

remove(22)



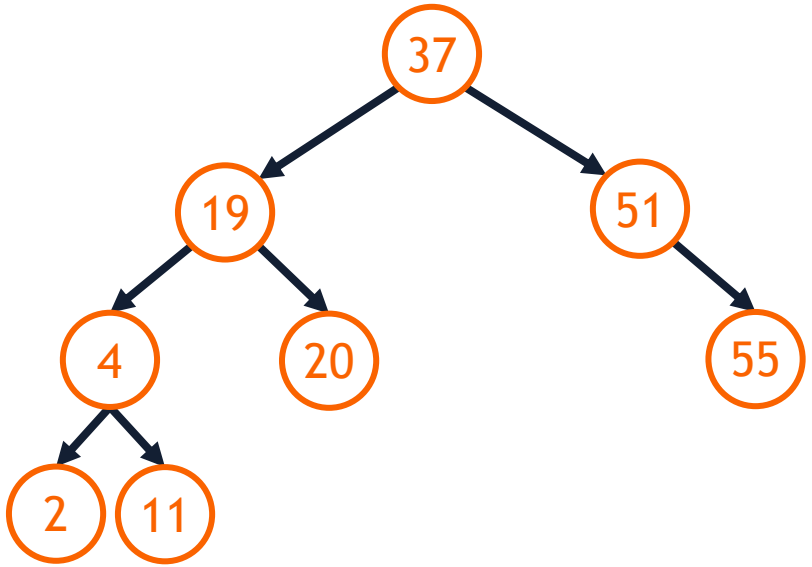
Remove from a BST

remove(37)



In-Order Predecessor (IOP)

The in-order predecessor is the previous node in an in-order traversal of a BST.

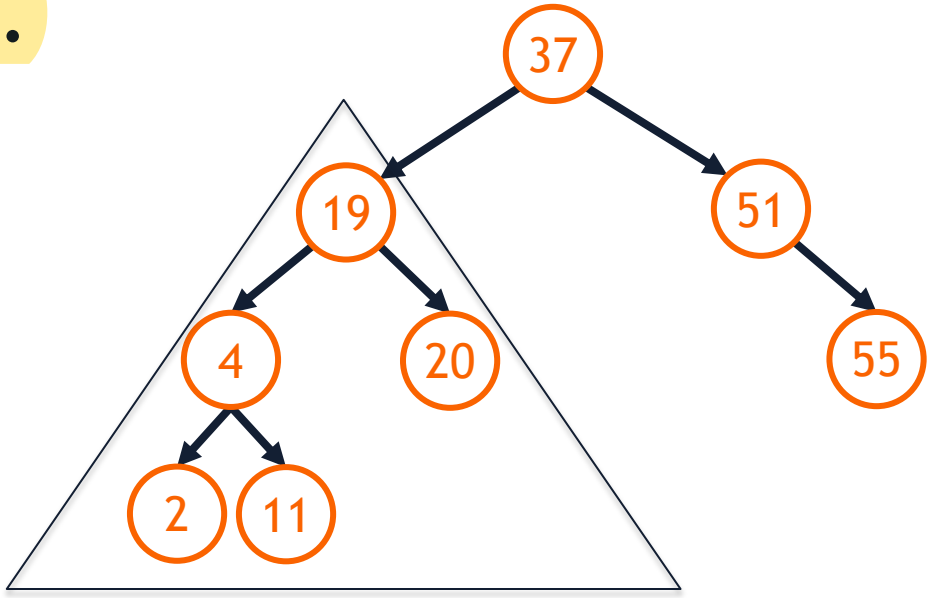


In-Order Traversal: 2 4 11 19 20 37 51 55

In-Order Predecessor (IOP)

The IOP of a node will **always** be the right-most node in the node's left sub-tree.

IOP(37)



BST::remove

Zero children: Simple, delete the node.

One child: Simple, works like a linked list.

Two children:

- Find the IOP of the node to be removed.
- Swap with the IOP.
- Remove the node in it's new position.

bst/Dictionary.cpp

```
37  /**
38  * remove()
39  * Removes `key` from the Dictionary. Returns the associated data.
40  */
41  template <typename K, typename D>
42  const D & Dictionary<K, D>::remove(const K & key) {
43    TreeNode *& node = _find(key, head_);
44    return _remove(node);
45  }

102 template <typename K, typename D>
103 const D & Dictionary<K, D>::_remove(TreeNode *& node) {
104   // Zero child remove:
105   if (node->left == nullptr && node->right == nullptr) {
106     const D & data = node->data;
107     delete(node);
108     node = nullptr;
109     return data;
110   }
      …
```

bst/Dictionary.cpp

```
112     // One-child (left) remove
113     else if (node->left != nullptr && node->right == nullptr) {
114       const D & data = node->data;
115       TreeNode *temp = node;
116       node = node->left;
117       delete(temp);
118       return data;
119     }
120
121     // One-child (right) remove
122     else if (node->left == nullptr && node->right != nullptr) {
123       const D & data = node->data;
124       TreeNode *temp = node;
125       node = node->right;
126       delete(temp);
127       return data;
128     }
```

bst/Dictionary.cpp

```
130     // Two-child remove:
131     else {
132       // Find the IOP
133       TreeNode *& iop = _iop( node->left );
134
135       // Swap the node to remove and the IOP
136       _swap( node, iop );
137
138       // Remove the new IOP node that was swapped
139       return _remove( node );
140     }
141 }
```

bst/main.cpp

```cpp
int main() {
  Dictionary<int, std::string> t;
  t.insert(37, "thirty four");
  t.insert(19, "ninteen");
  t.insert(51, "fifty one");
  t.insert(55, "fifty five");
  t.insert(4, "four");
  t.insert(11, "eleven");
  t.insert(20, "twenty");
  t.insert(2, "two");

  cout << "t.find(51): " << t.find(51) << endl;

  cout << "t.remove(11): " << t.remove(11) << " (zero child remove)" << endl;
  cout << "t.remove(51): " << t.remove(51) << " (one child remove)" << endl;
  cout << "t.remove(19): " << t.remove(19) << " (two child remove)" << endl;

  cout << "t.find(51): " << t.find(51) << endl;
  return 0;
}
```

BST

- A binary search tree (BST) is an ordered binary tree.

- The use of the ordered property allows us to create an efficient dictionary structure.