SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220502<A|D>

Replace <A|D> with this section's letter

Recursion III

—

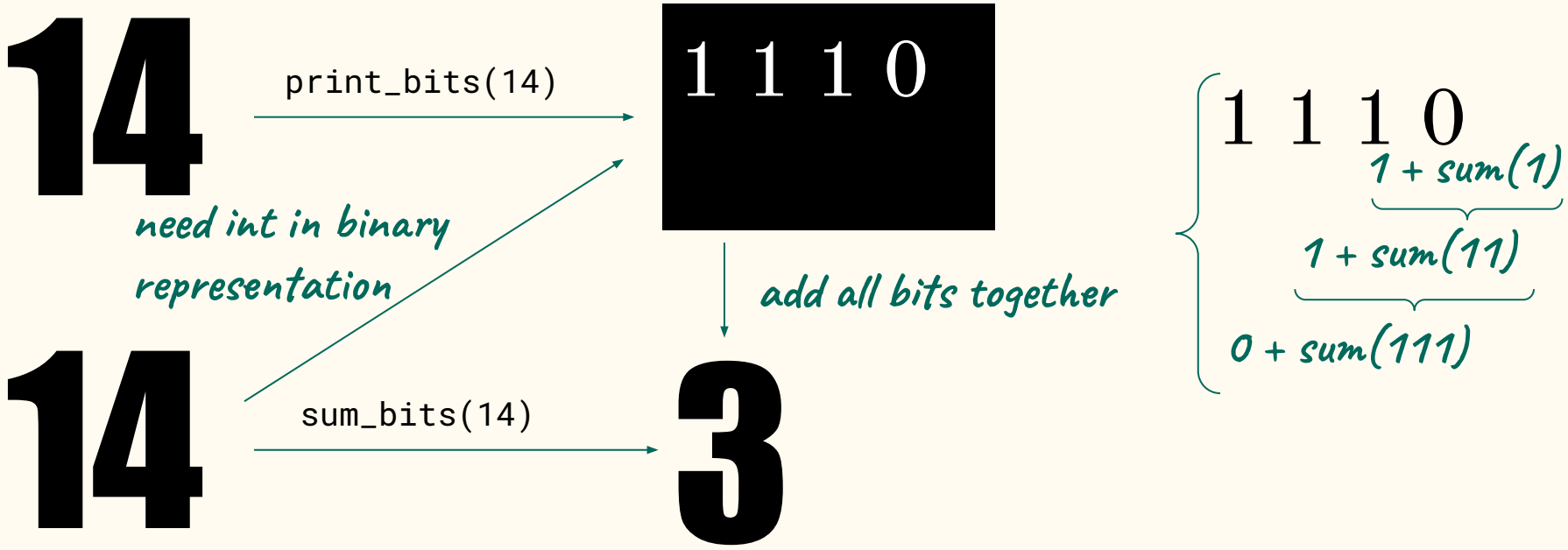
CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Recursive strategy (continued)
- Binary tree recursion
- Memoization

Recursive strategy

Summing bits



Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

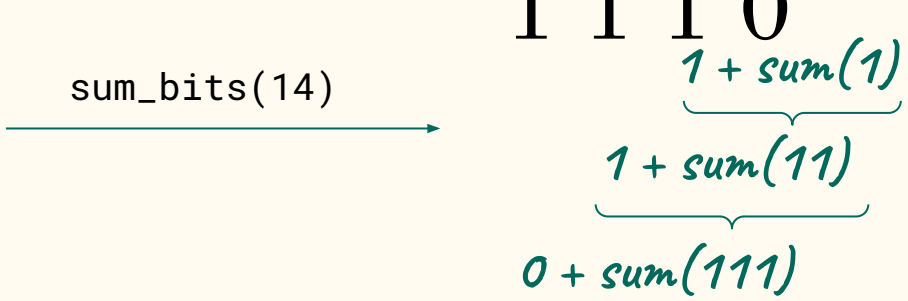
Notice: return type

```
int sum_bits(int num) { }
```

Summing bits

14

sum_bits(14)




```
int sum_bits(int num) {
    // base case

    // recursive case
}
```

Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    // base case

    ---

    // recursive case
}
```

Summing bits

14

sum_bits(14)

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    // base case
    ---

    // recursive case
}
```

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220502<A|D>

Replace <A|D> with this section's letter

What range of values for num represent the base case of this problem? Which condition indicates there are no bits left to sum?

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    // base case
    ___

    // recursive case
}
```

Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    // base case
    if (num < 2) ___;

    // recursive case
}
```

Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    // base case
    if (num < 2) _5_;

    // recursive case
}
```

When the base case is reached, what needs to happen?
Which statement replaces blank #5?

14

sum_bits(14) →

1 1 1 0

1 + sum(1)

1 + sum(11)

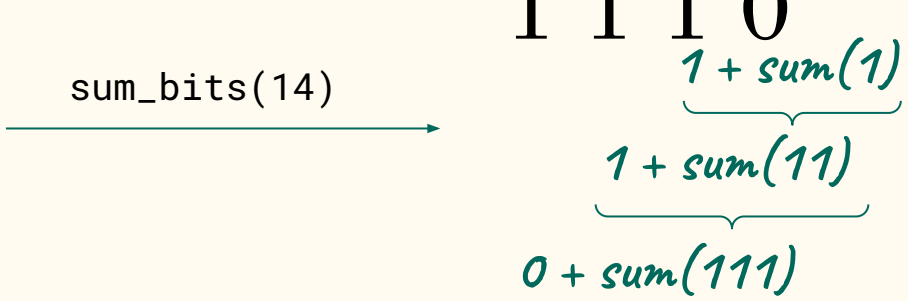
0 + sum(111)

```
int sum_bits(int num) {
    // base case
    if (num < 2) _5_;

    // recursive case
}
```

Summing bits

14

sum_bits(14)




```
int sum_bits(int num) {
    // base case
    if (num < 2) return num;

    // recursive case
}
```

Summing bits

14

sum_bits(14)

1 1 1 0

1 + sum(1)

1 + sum(11)

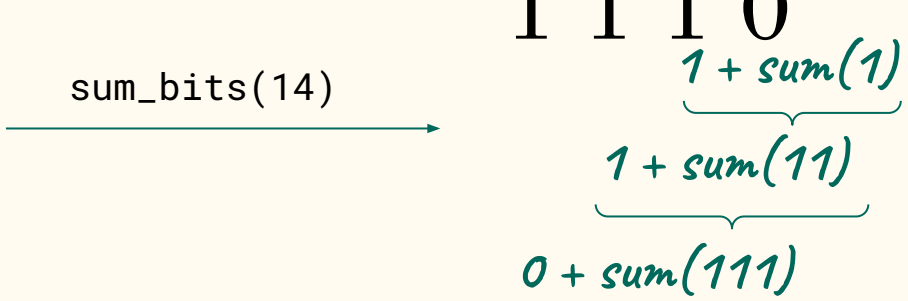
0 + sum(111)

```
int sum_bits(int num) {
    // base case
    if (num < 2) {
        return num;
    } else {
        // recursive case
    }
}
```

Summing bits

14

sum_bits(14)




```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        ---
    }
}
```

Summing bits

14

sum_bits(14)

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return ___ + ___;
    }
}
```

Which expression (replacing blank #6) evaluates to the value of the bit to be added to the sum of the remaining bits?

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return _6_ + ___;
    }
}
```

Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return num % 2 + ___;
    }
}
```

Summing bits

14

sum_bits(14)

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return num % 2 + _7_;
    }
}
```

Which recursive function call evaluates to the sum of the remaining bits (replacing blank #7)?

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return num % 2 + _7_;
    }
}
```

Summing bits

14

sum_bits(14)

1 1 1 0

1 + sum(1)

1 + sum(11)

0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        // recursive case
        return num % 2 + sum_bits(num / 2);
    }
}
```

Summing bits

14

`sum_bits(14)` →

1 1 1 0

1 + sum(1)

1 + sum(11)

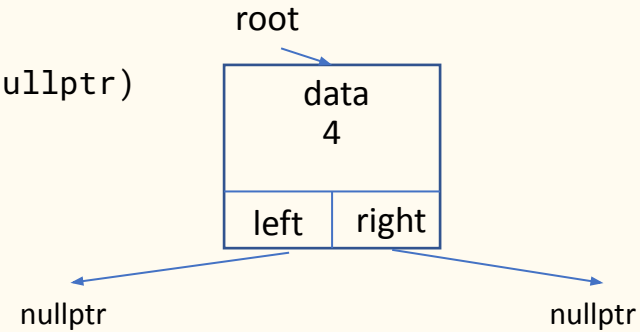
0 + sum(111)

```
int sum_bits(int num) {
    if (num < 2) {
        return num;
    } else {
        return num % 2 + sum_bits(num / 2);
    }
}
```

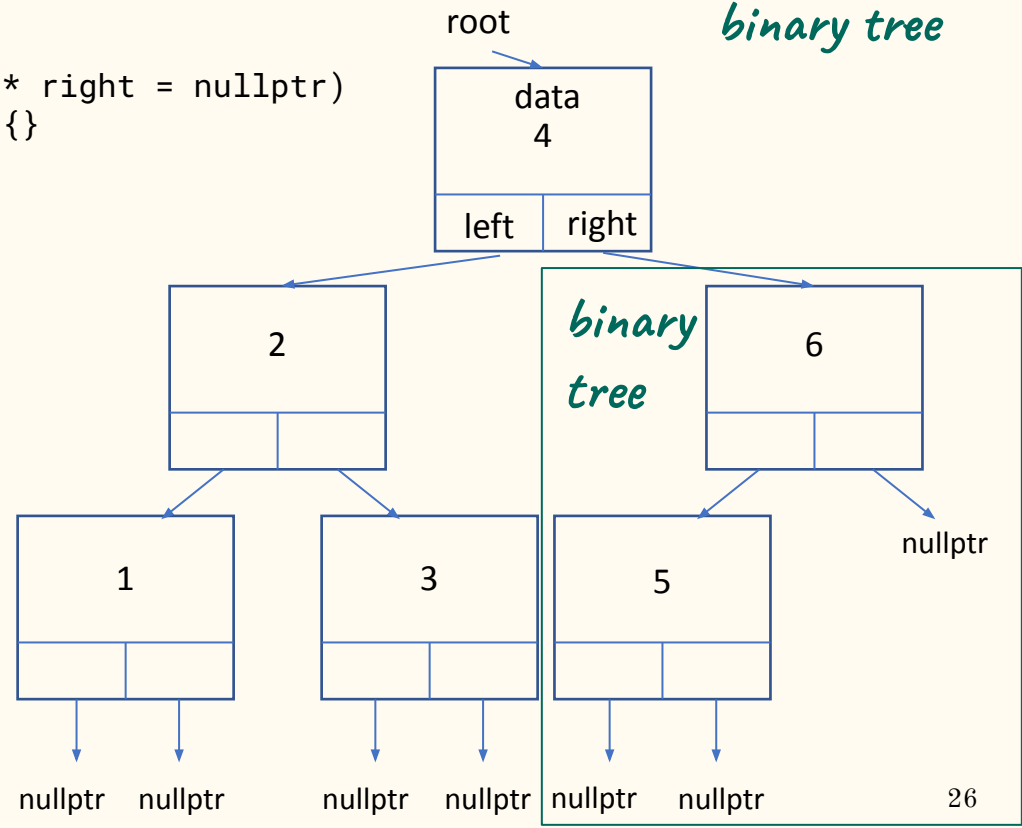
Binary tree recursion

Binary tree structure

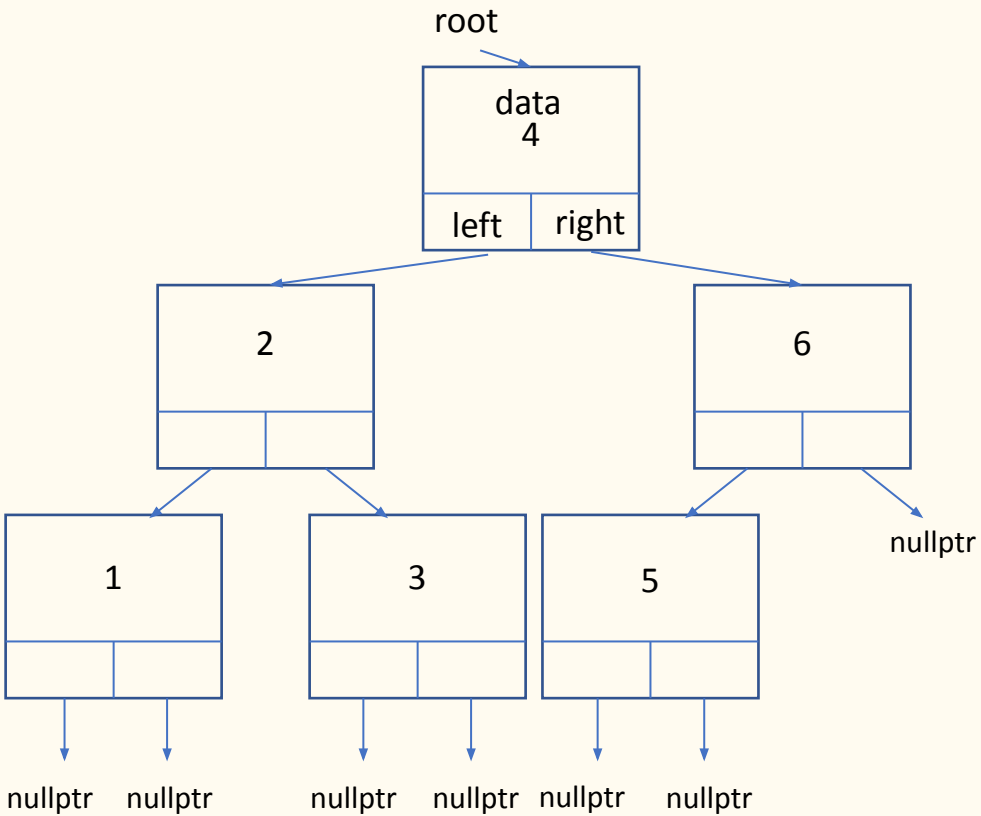
```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


int main() {
   TNode* root = new TNode(4);
}
```



```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

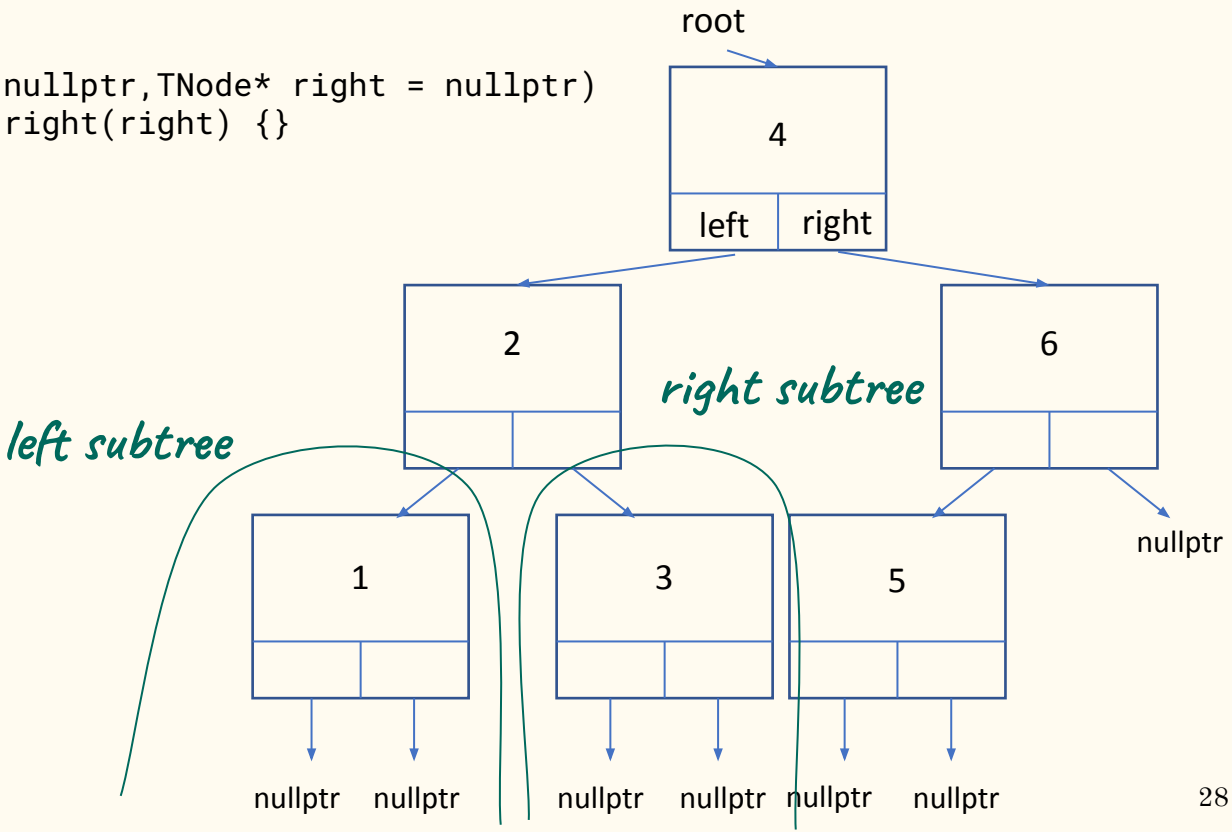


How many (non-empty) binary trees exist in this structure?



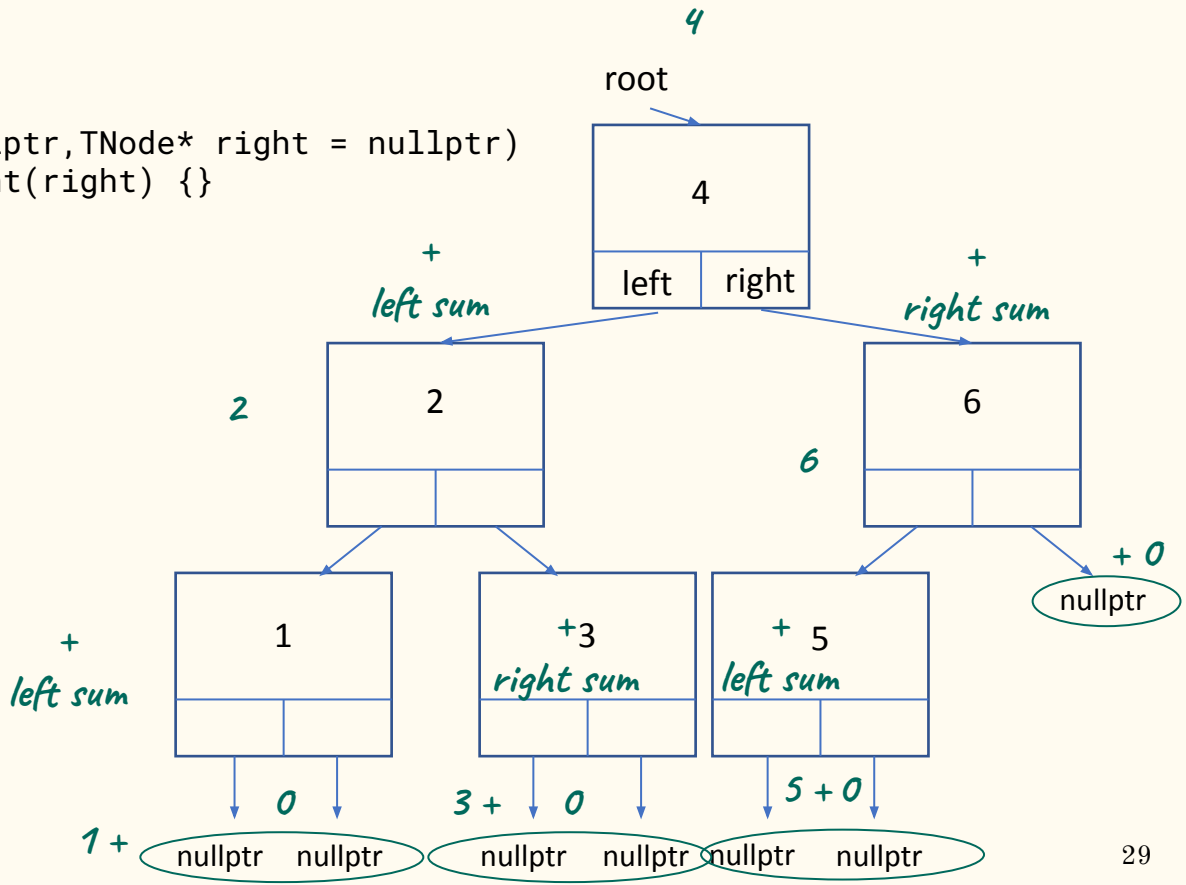
Binary tree structure

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```



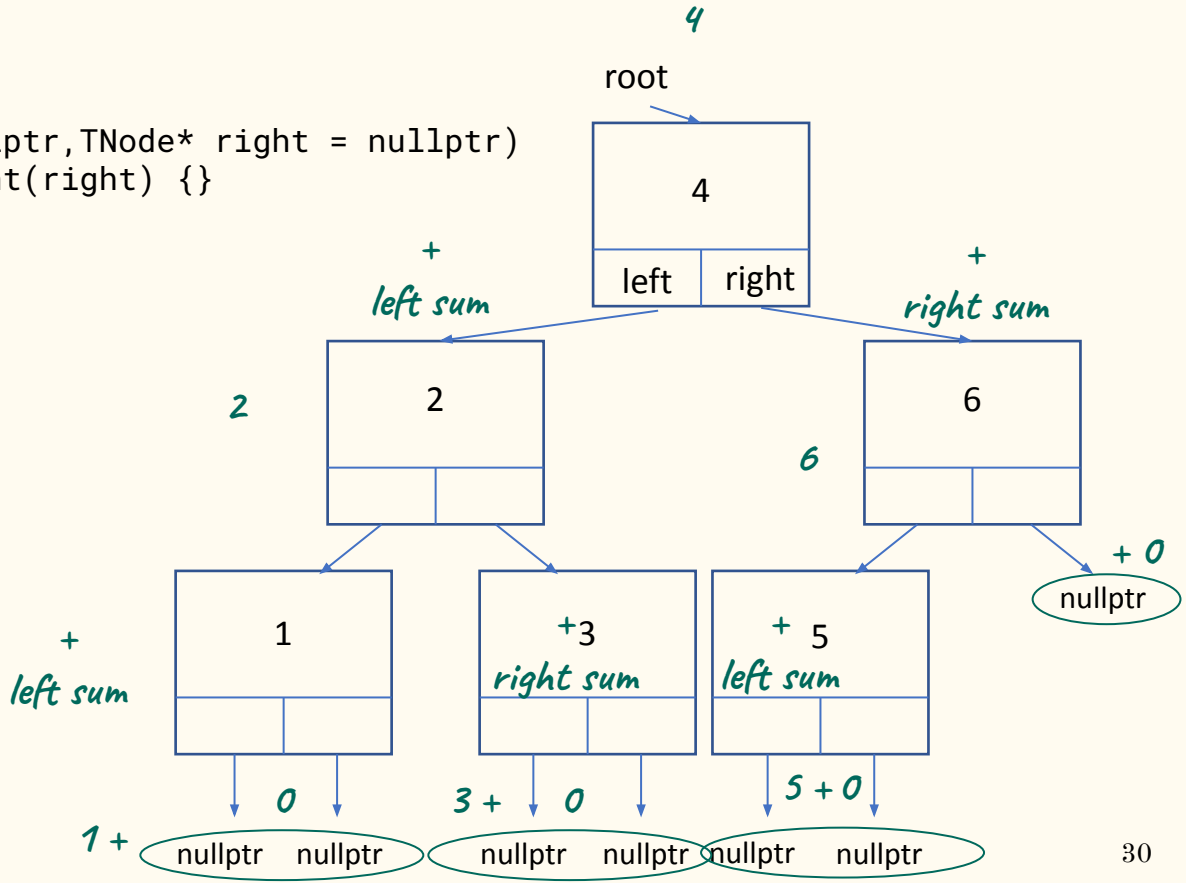
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) { }
```



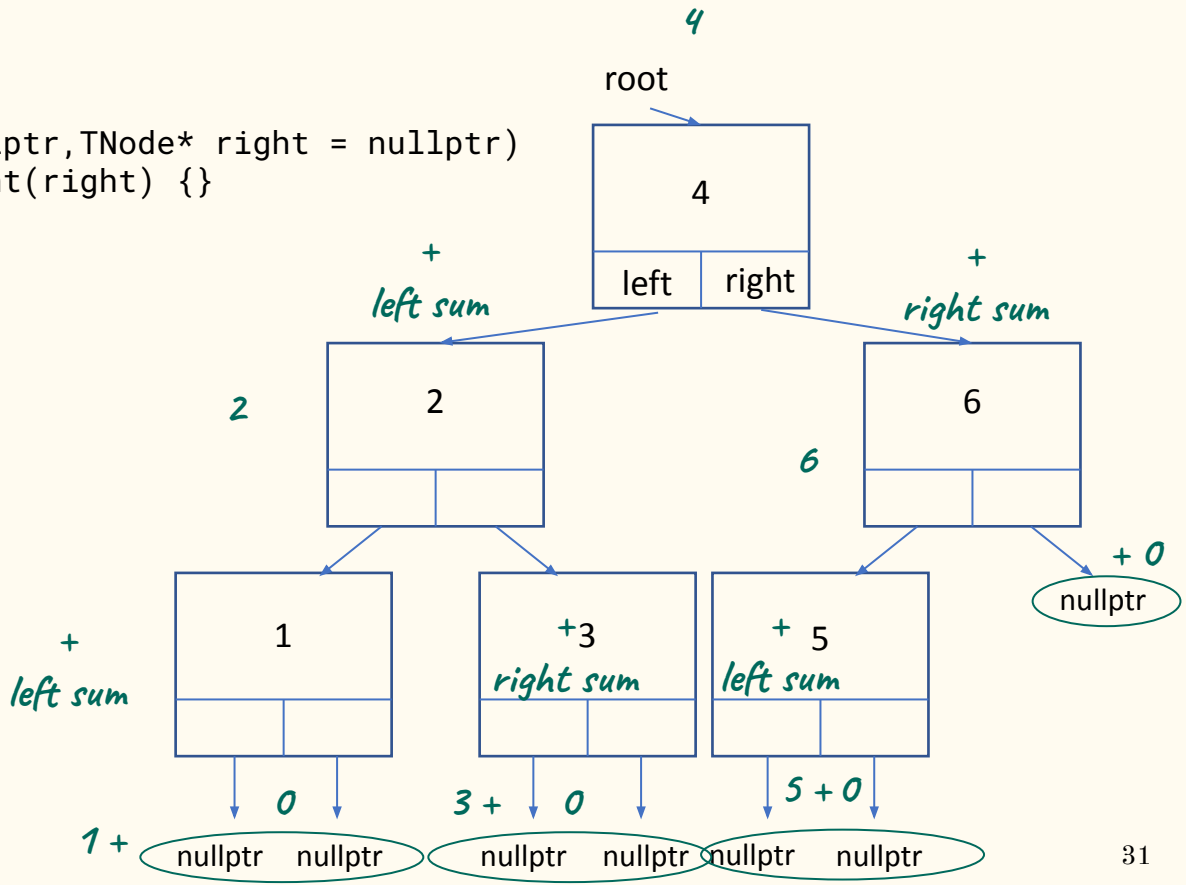
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case

     // recursive case

 }
```



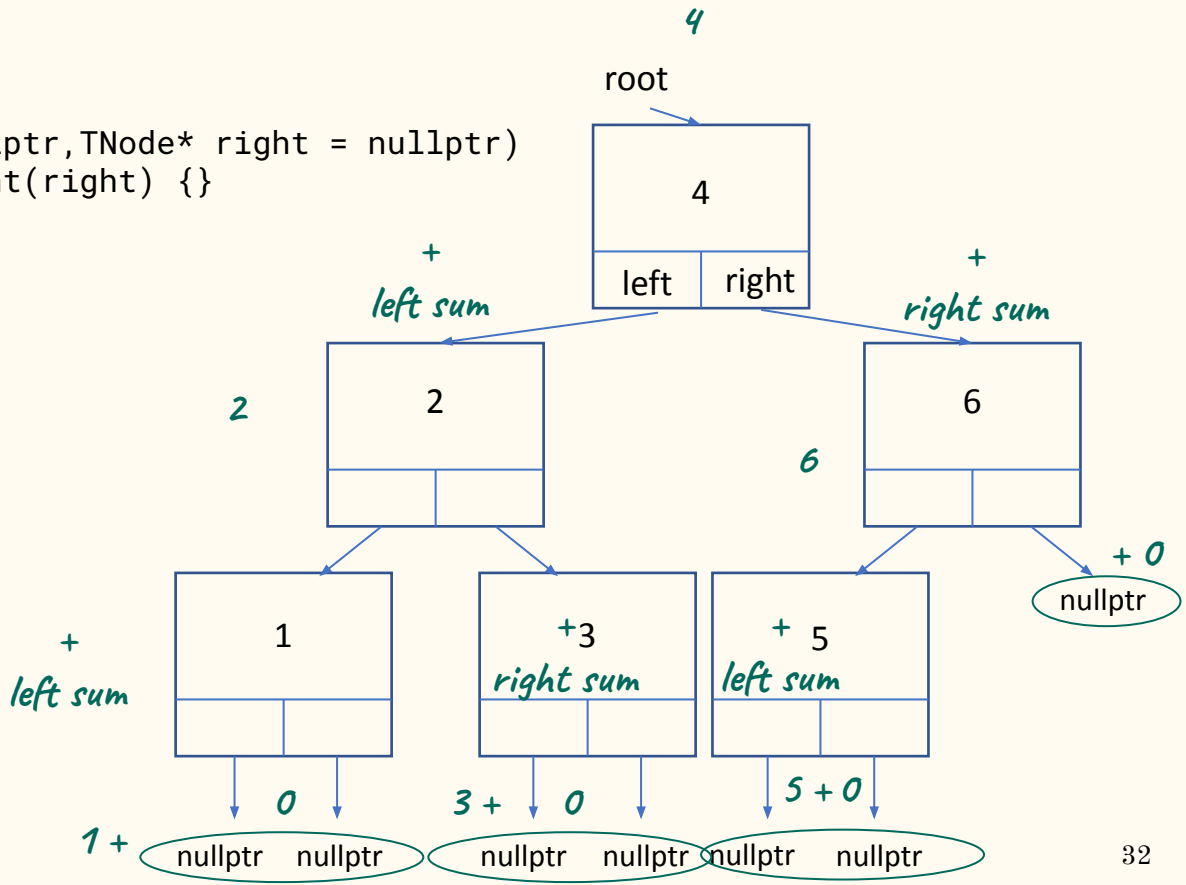
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case

     // recursive case
 }
```



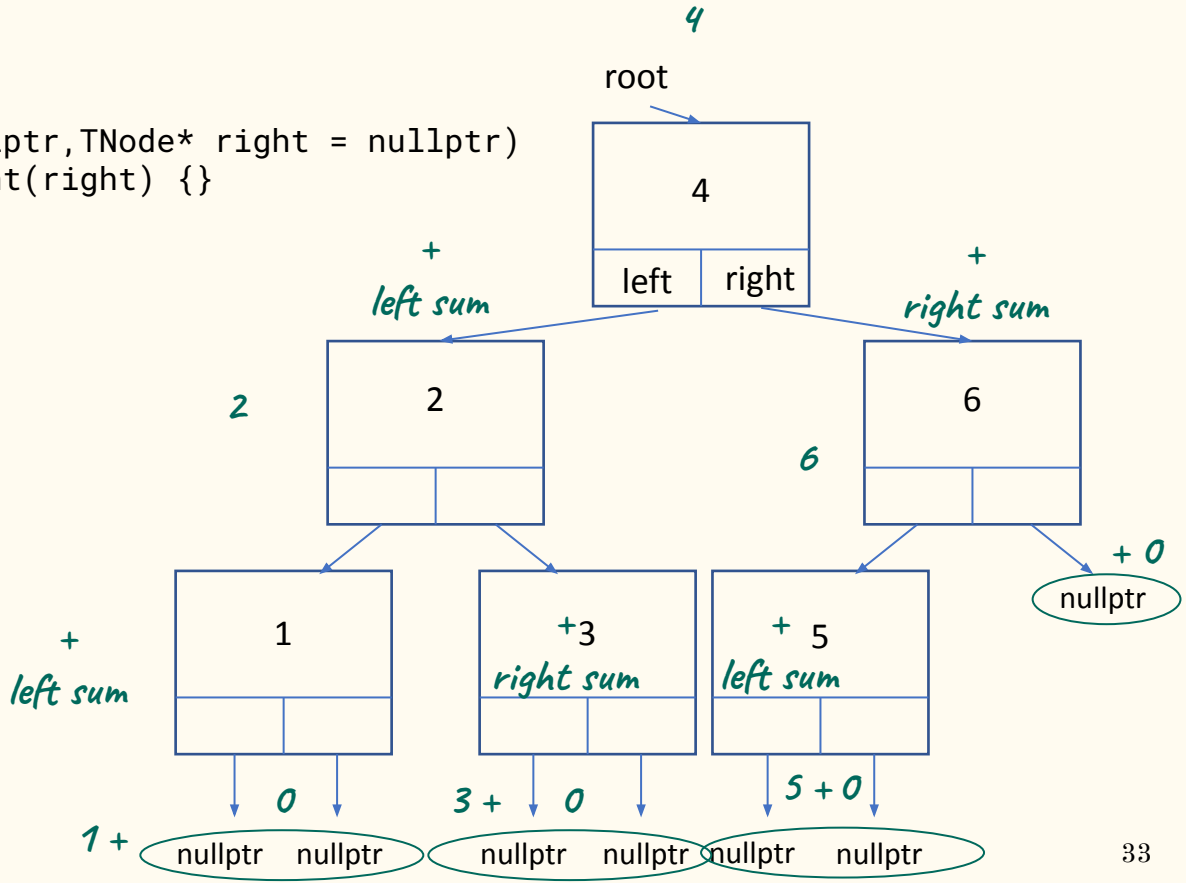
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     ---

     // recursive case
 }
```



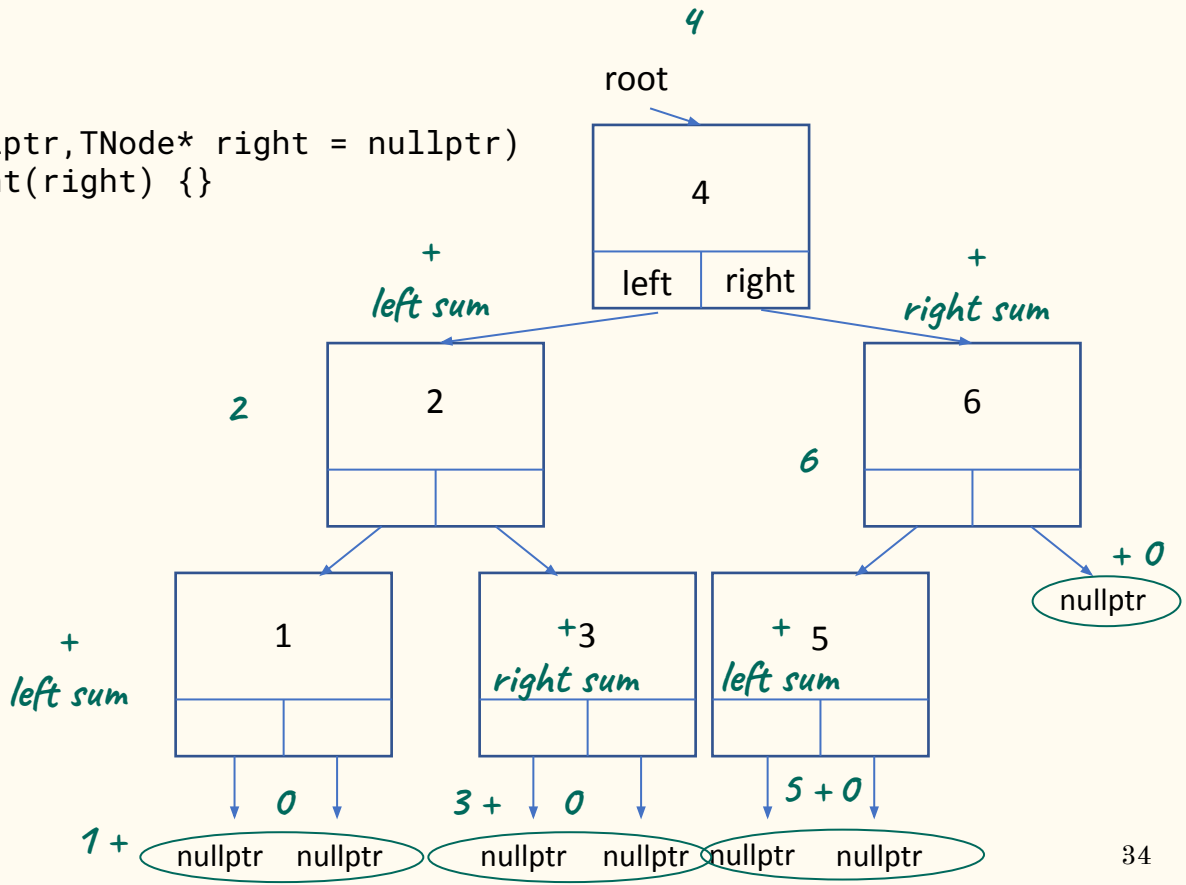
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (___) ___;

     // recursive case
 }
```



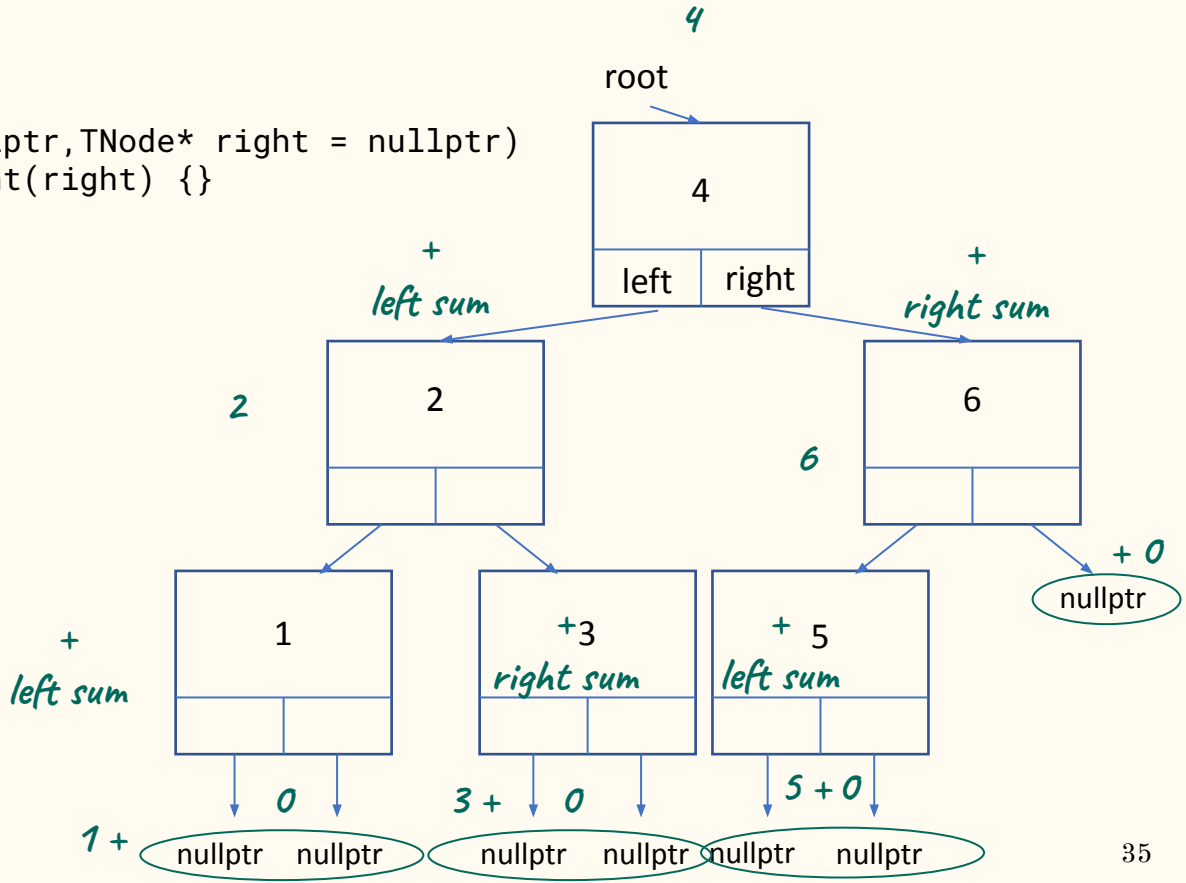
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == ___) ___;

     // recursive case
 }
```



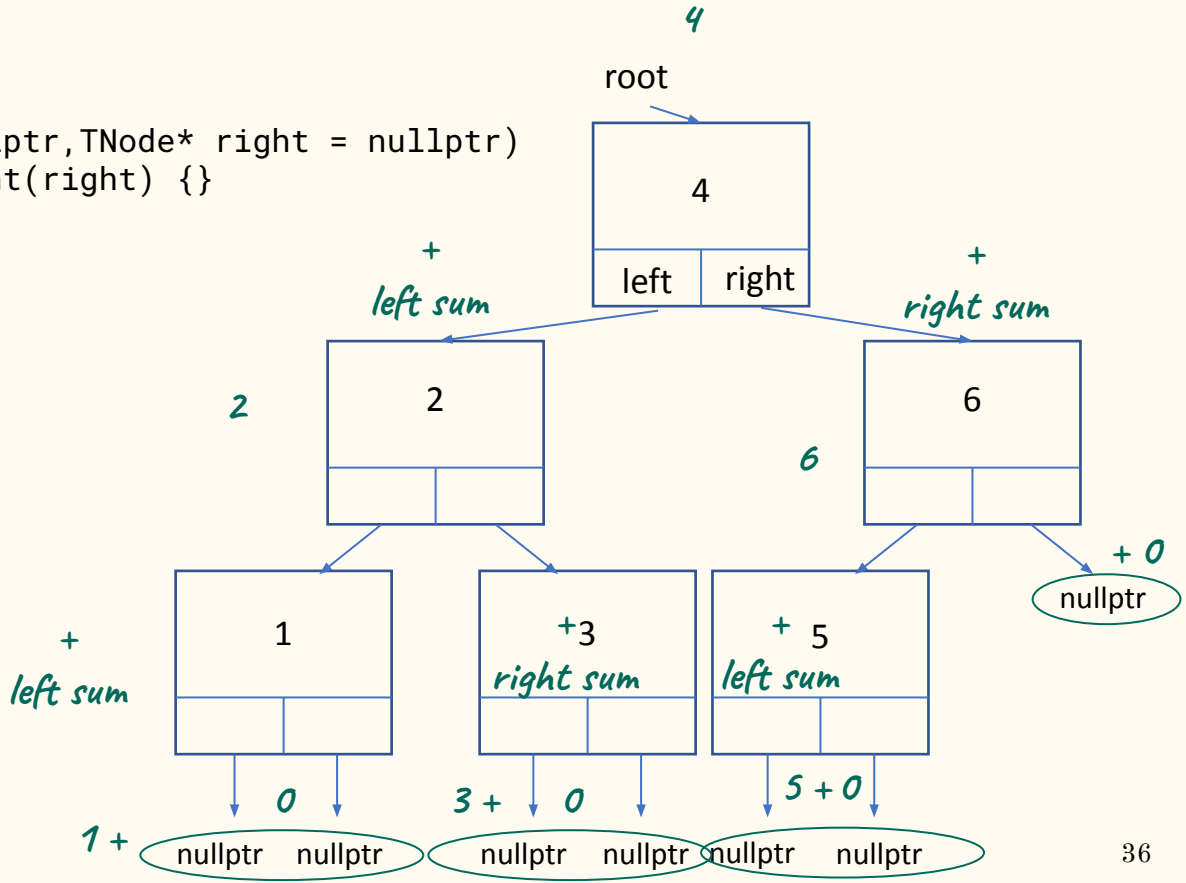
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == _8_) ___;

     // recursive case
 }
```



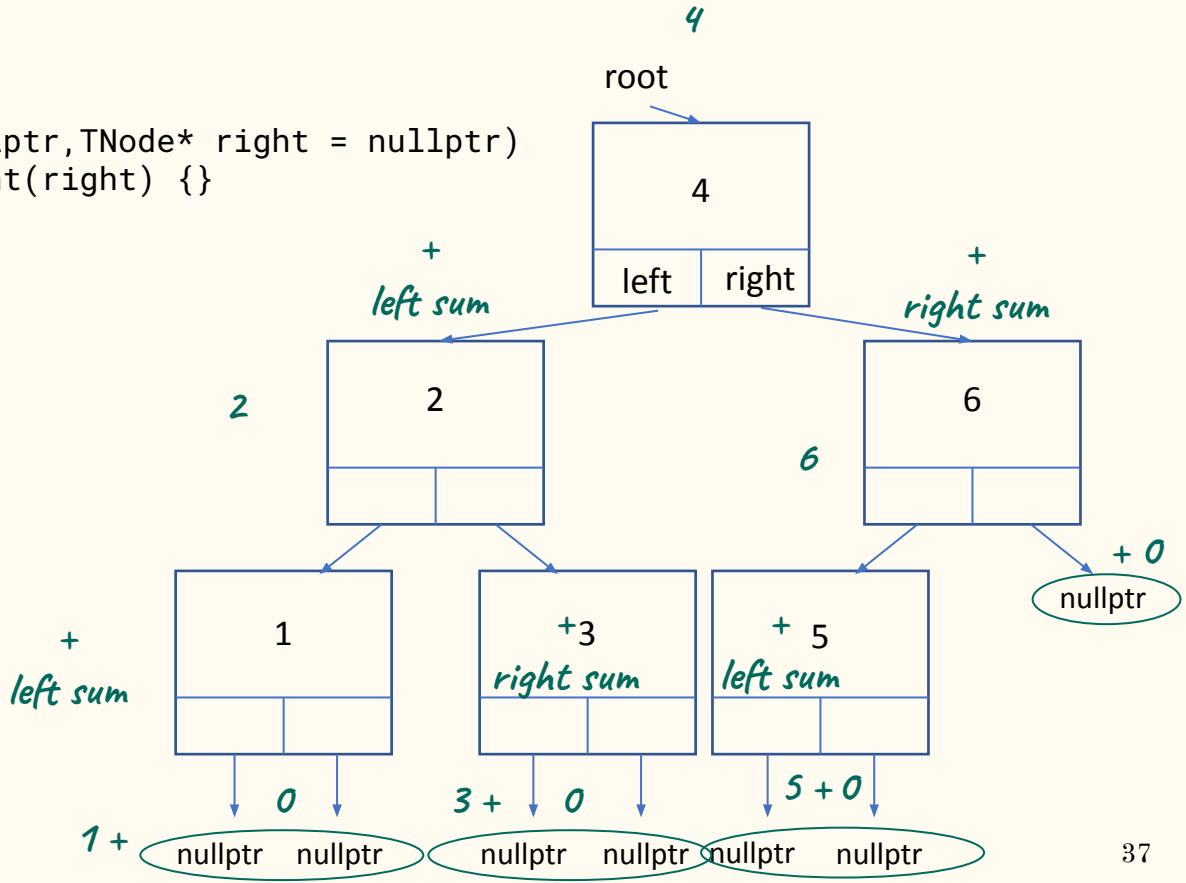
Which value replaces blank #8 as the condition for the base case?

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == _8_) ___;

     // recursive case
 }
```



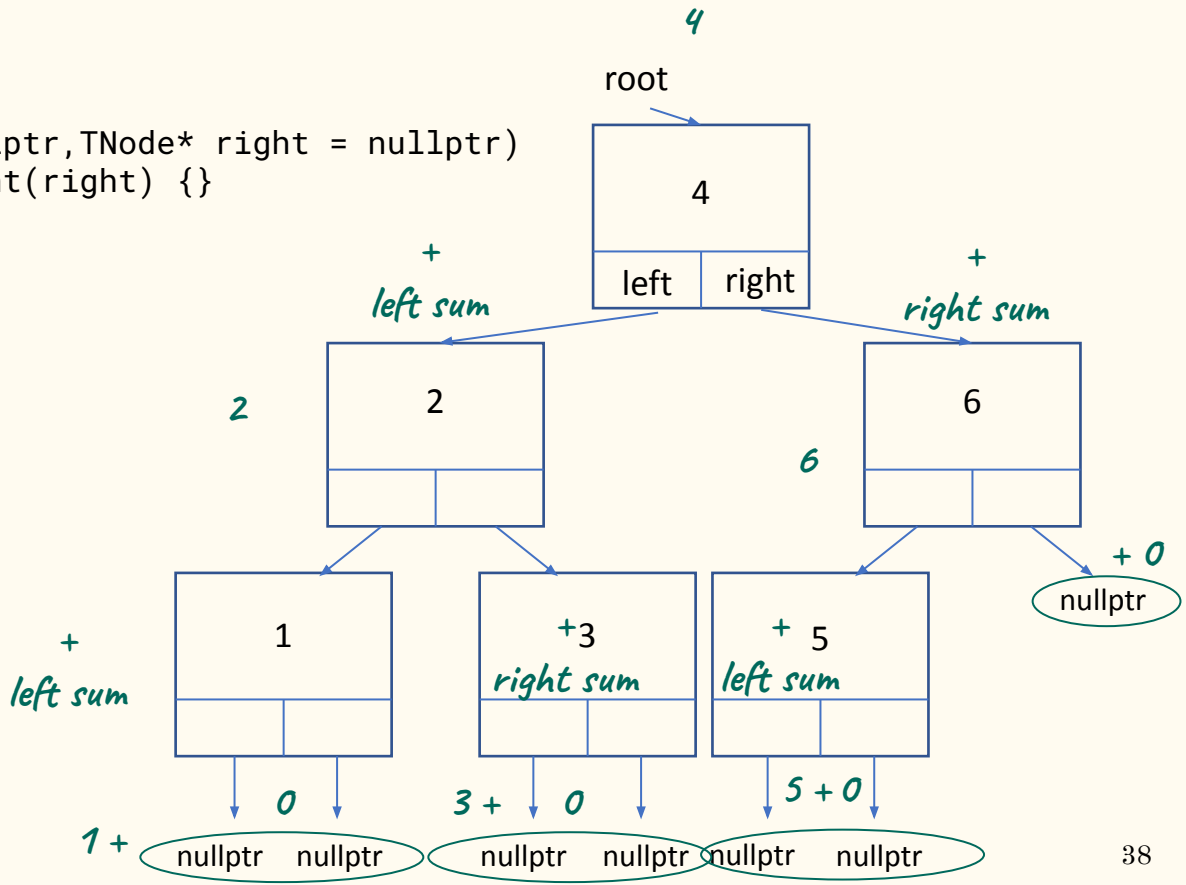
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == nullptr) ___;

     // recursive case
 }
```



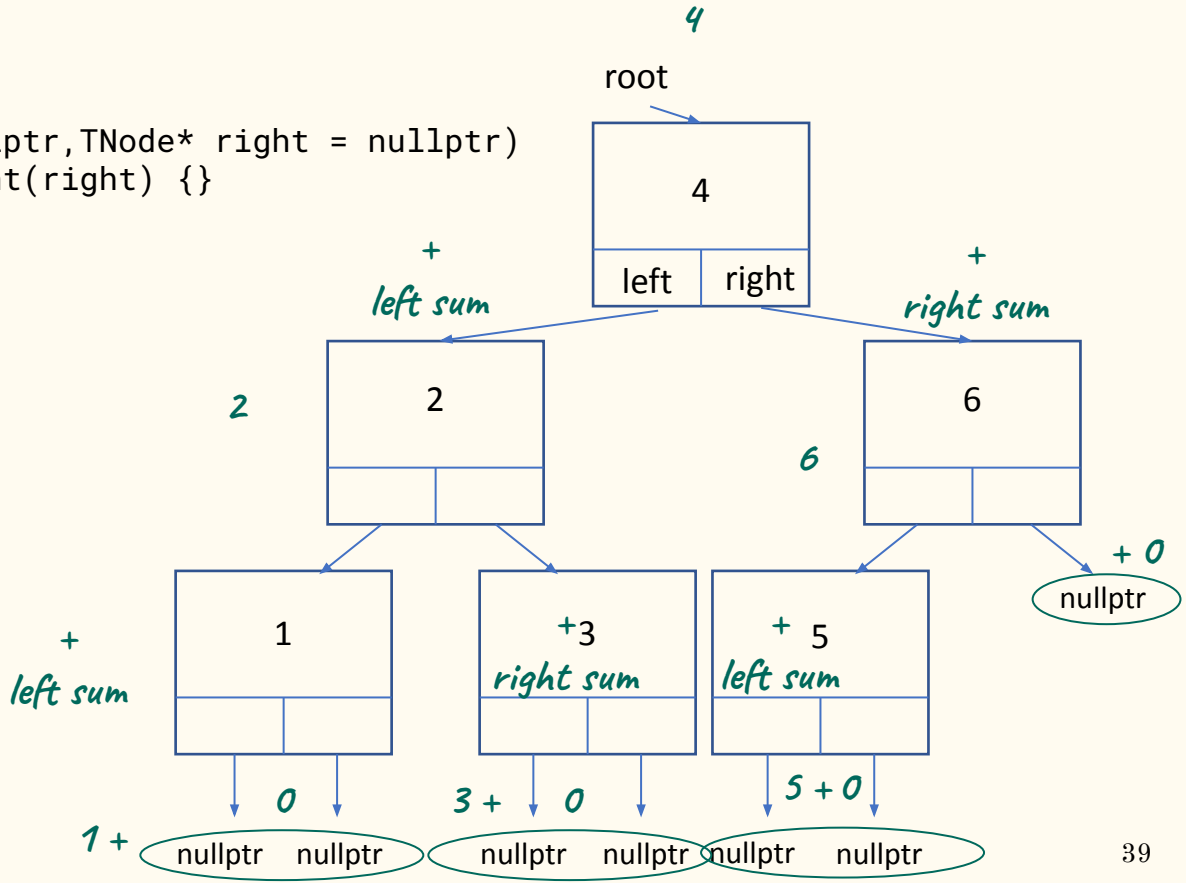
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == nullptr) return ___;

     // recursive case
 }
```



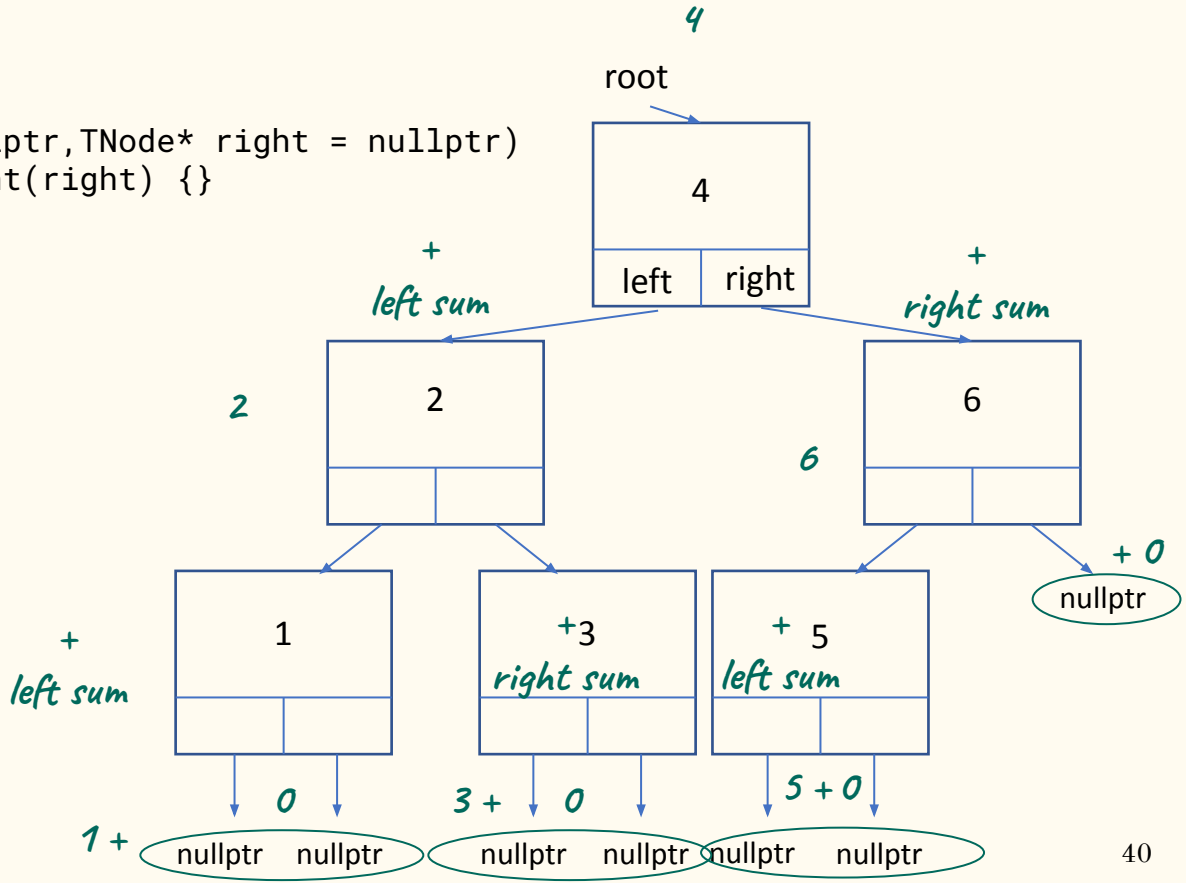
Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == nullptr) return _9_;

     // recursive case
 }
```



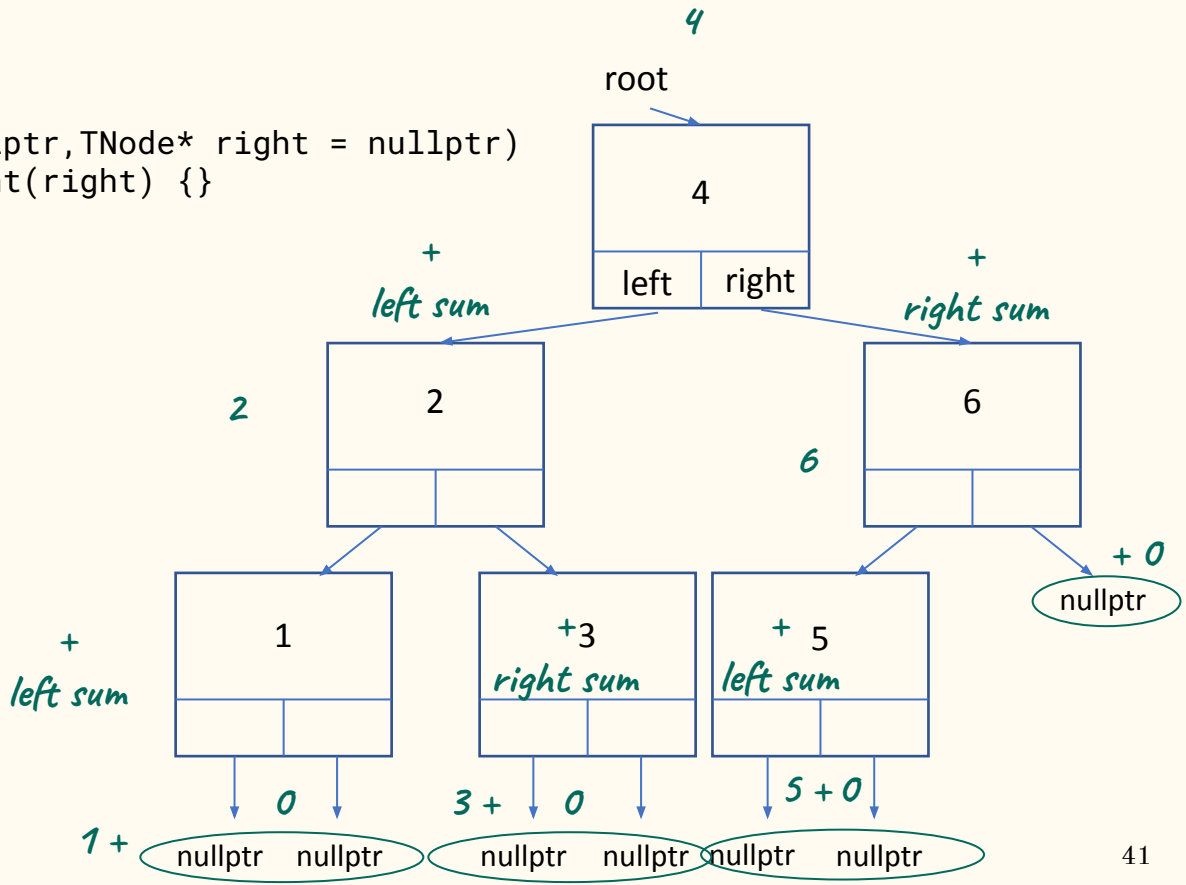
Which statement replaces blank #9 to return the correct value when the base case is reached?

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == nullptr) return _9_;

     // recursive case
 }
```



Summing tree nodes

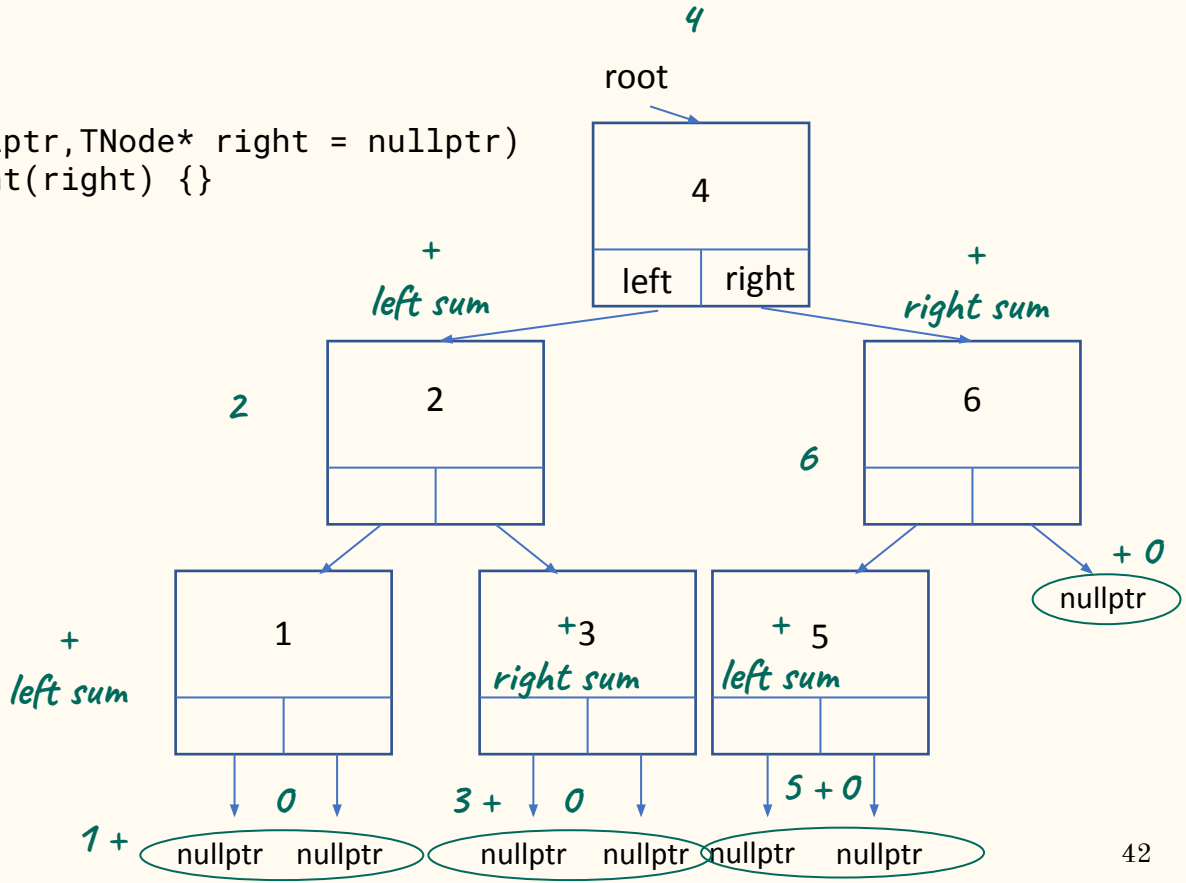
```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     // base case
     if (root == nullptr) return 0;

     // recursive case
 }
```



Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

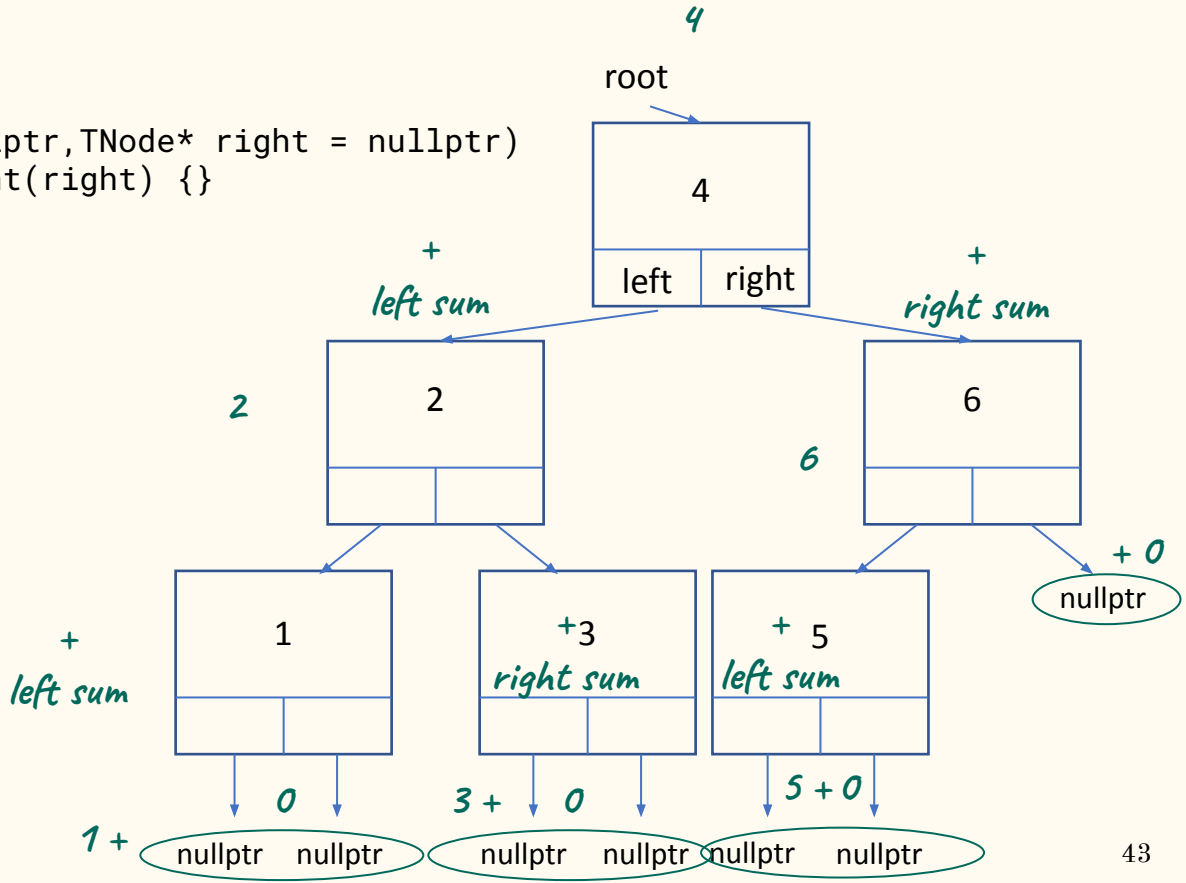
```
int tree_sum(TNode* root) {
    // base case
    if (root == nullptr) {
        return 0;
    } else {
        // recursive case
    }
}
```



Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

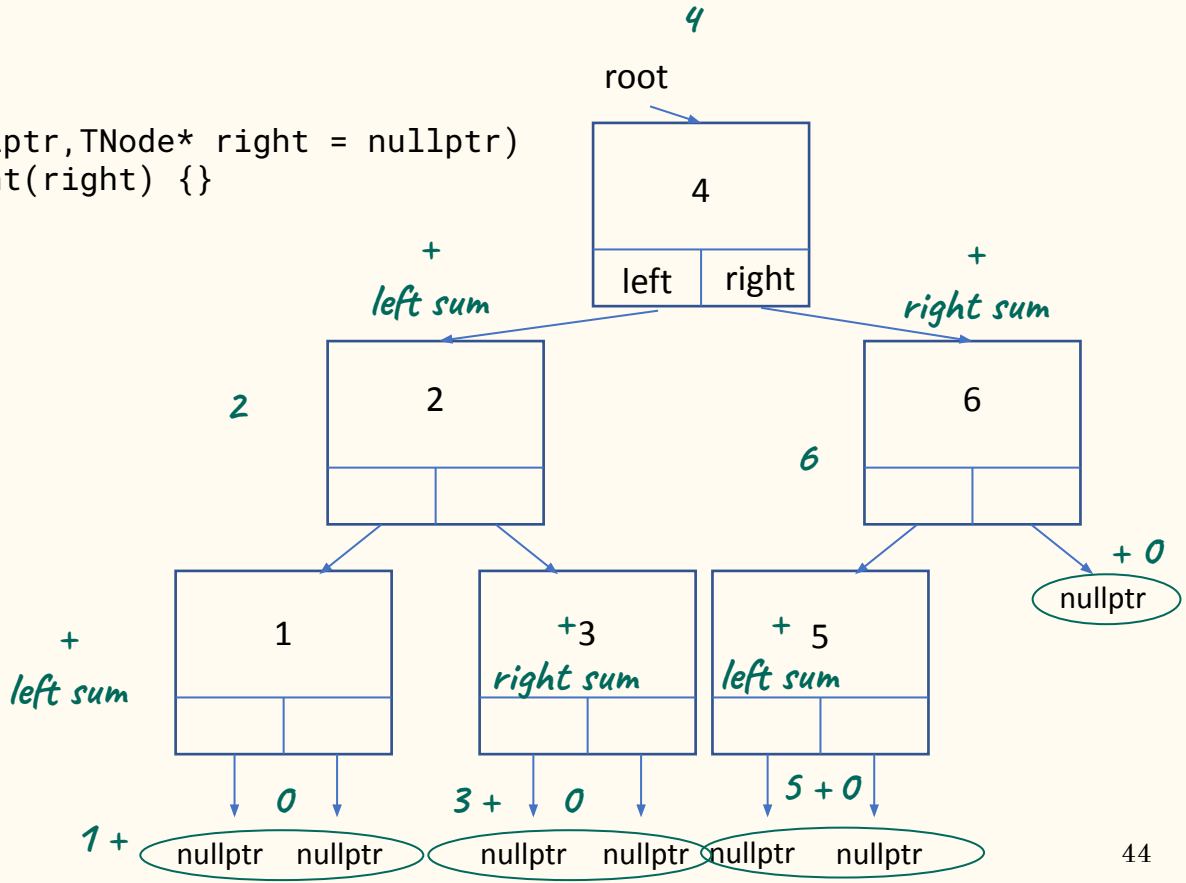
```
int tree_sum(TNode* root) {
    if (root == nullptr) {
        return 0;
    } else {
        // recursive case
    }
}
```



Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

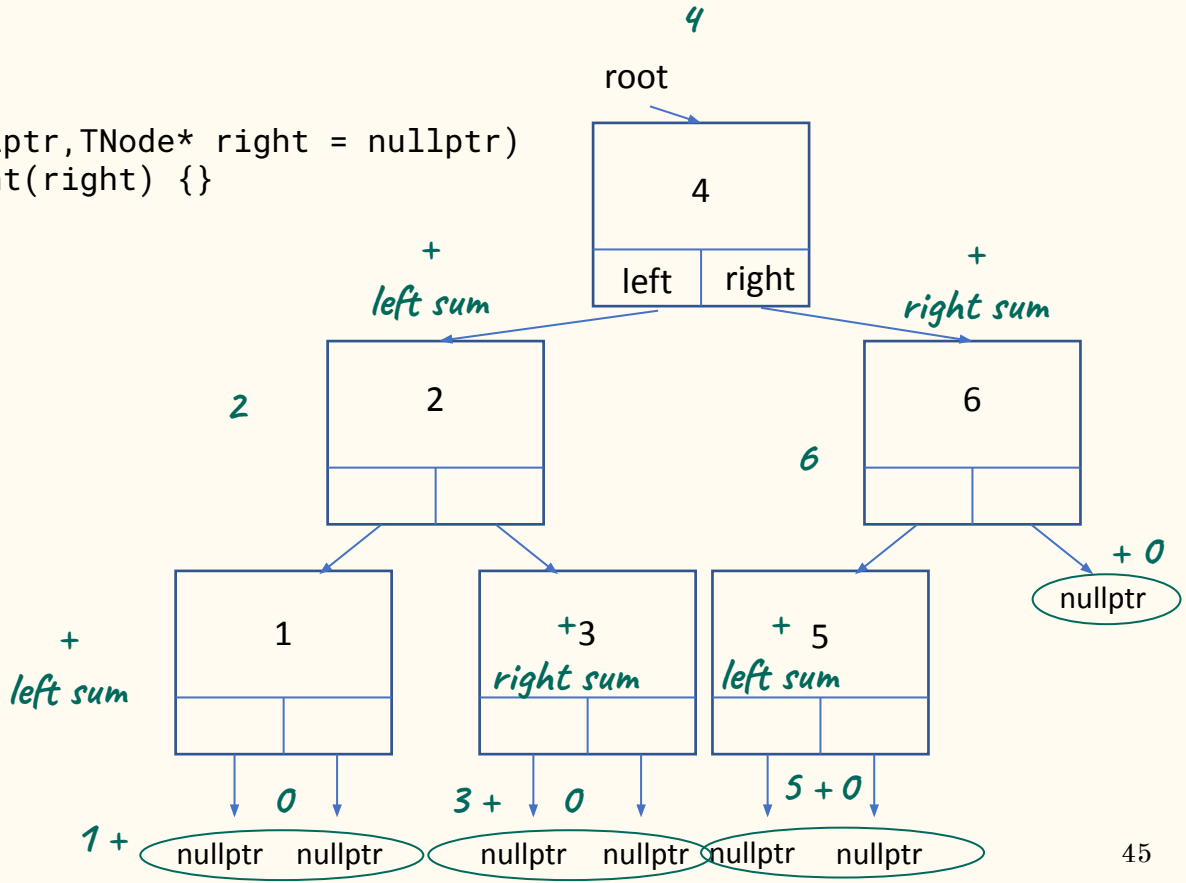
```
int tree_sum(TNode* root) {
    if (root == nullptr) {
        return 0;
    } else {
        // recursive case
        ---
    }
}
```



Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

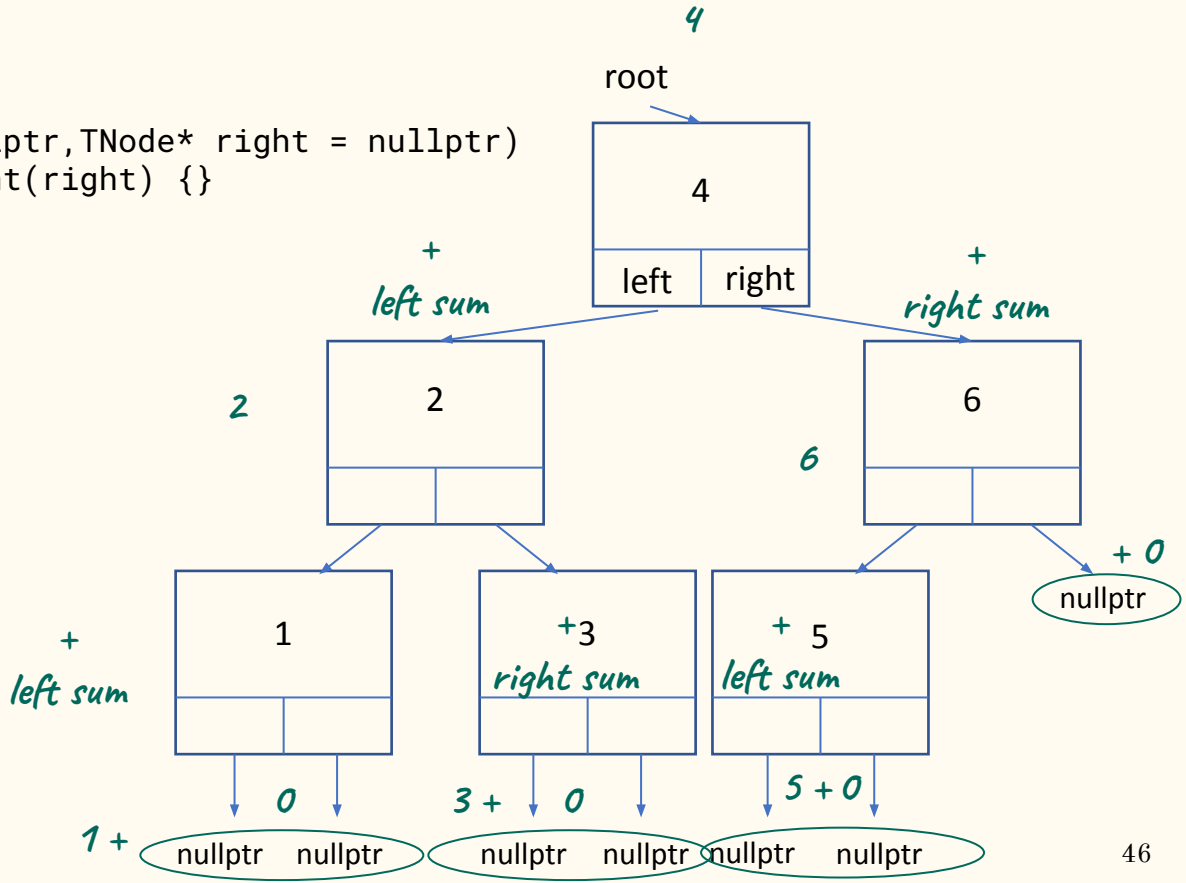
```
int tree_sum(TNode* root) {
    if (root == nullptr) {
        return 0;
    } else {
        // recursive case
        return ___ + ___ + ___;
    }
}
```



Summing tree nodes

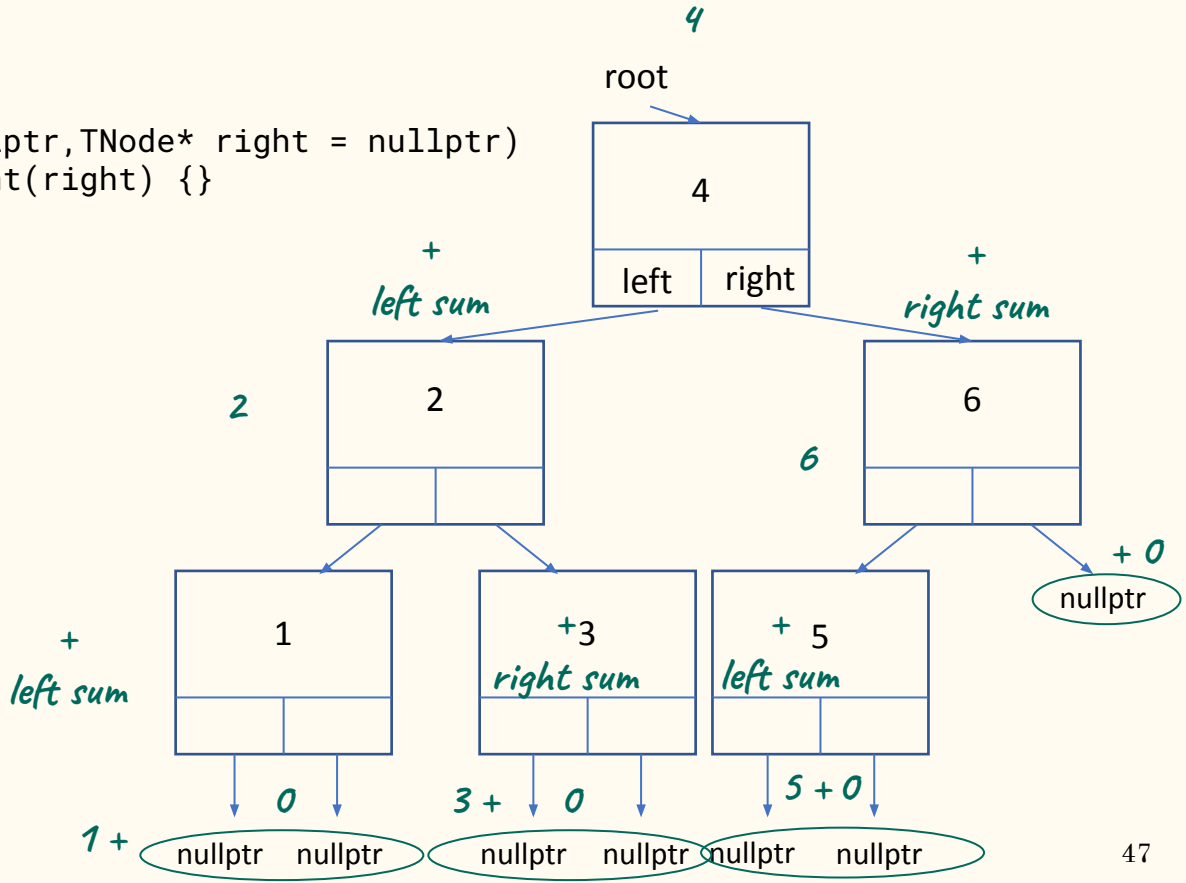
```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};
```

```
int tree_sum(TNode* root) {
    if (root == nullptr) {
        return 0;
    } else {
        // recursive case
        return _10_ + ___ + ___;
    }
}
```



Which expression replaces blank #10 to add the value at the current `TNode` to the sum?

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return _10_ + ___ + ___;
     }
 }
```



Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + ___ + ___;
     }
 }
```

Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + _11_ + ___;
     }
 }
```

Which recursive function call replaces blank #11 to include the sum of the left subtree in the full sum?

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + _11_ + ___;
     }
 }
```

Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + tree_sum(root->left) + ___;
     }
 }
```

Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + tree_sum(root->left) + _12_;
     }
 }
```

Which recursive function call replaces blank #12 to include the sum of the right subtree in the full sum?

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + tree_sum(root->left) + _12_;
     }
 }
```

Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         // recursive case
         return root->data + tree_sum(root->left) + tree_sum(root->right);
     }
 }
```

Summing tree nodes

```
struct TNode {
    TNode(int val, TNode* left = nullptr,TNode* right = nullptr)
        : data(val), left(left), right(right) {}
    int data;
    TNode* left;
    TNode* right;
};


 int tree_sum(TNode* root) {
     if (root == nullptr) {
         return 0;
     } else {
         return root->data + tree_sum(root->left) + tree_sum(root->right);
     }
 }
```

Memoization

Fibonacci numbers

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

for $N > 1$

recursive definition

$F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8, F_9, F_{10}, \ldots$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

fibonacci sequence

Fibonacci numbers

numbers can be large

```
long long fib(int n) {
    long long curr = 0, next = 1;
    for (int i = 1; i <= n; ++i) {
        long long temp = curr + next;
        curr = next;
        next = temp;
    }
    return curr;
}
```

fib(0) = 0
fib(1) = 1
fib(2) = 1 (fib(0) + fib(1))
fib(3) = 2 (fib(1) + fib(2))
fib(4) = 3 (fib(2) + fib(3))
fib(5) = 5 (fib(3) + fib(4))
fib(6) = 8 (fib(4) + fib(5))
fib(7) = 13 (fib(5) + fib(6))
fib(8) = 21 (fib(6) + fib(7))
fib(9) = 34 (fib(7) + fib(8))
fib(10) = 55 (fib(8) + fib(9))

Fibonacci numbers

```
long long fib(int n) {
    long long curr = 0, next = 1;
    for (int i = 1; i <= n; ++i) {
        long long temp = curr + next;
        curr = next;
        next = temp;
    }
    return curr;
}
```

$$F_0 = 0$$

$$F_1 = 1$$

$$F_N = F_{N-1} + F_{N-2}$$

can be implemented with recursive function

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case

    // recursive case
}
```

Which values of `n` represent the base case for the `fib_recurse()` function?

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case
    ---

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case
    if (n < 2) ___;

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case
    if (n < 2) _13_;

    // recursive case
}
```

Which statement is executed when the base case of the `fib_recurse()` function is reached (replacing blank #13)?

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case
    if (n < 2) _13_;

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    // base case
    if (n < 2) return n;

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    // recursive case
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    // recursive case
    ---
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    // recursive case
    return _14_;
}
```

Which expression computes the Nth Fibonacci number for n ≥ 2 (replacing blank #14)?

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    // recursive case
    return _14_;
}
```

A recursive Fibonacci computation

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    // recursive case
    return fib_recurse(n - 1) + fib_recurse(n - 2);
}
```

A recursive Fibonacci computation

$F_0 = 0$

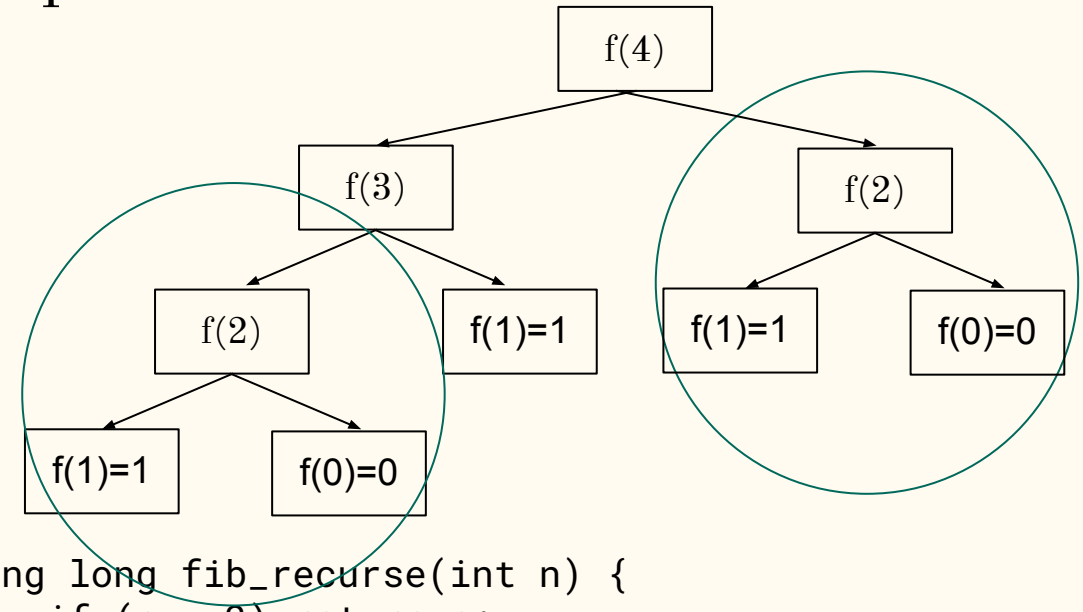
$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$

```
long long fib(int n) {
    long long curr = 0, next = 1;
    for (int i = 1; i <= n; ++i) {
        long long temp = curr + next;
        curr = next;
        next = temp;
    }
    return curr;
}
```

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    return fib_recurse(n - 1) + fib_recurse(n - 2);
}
```

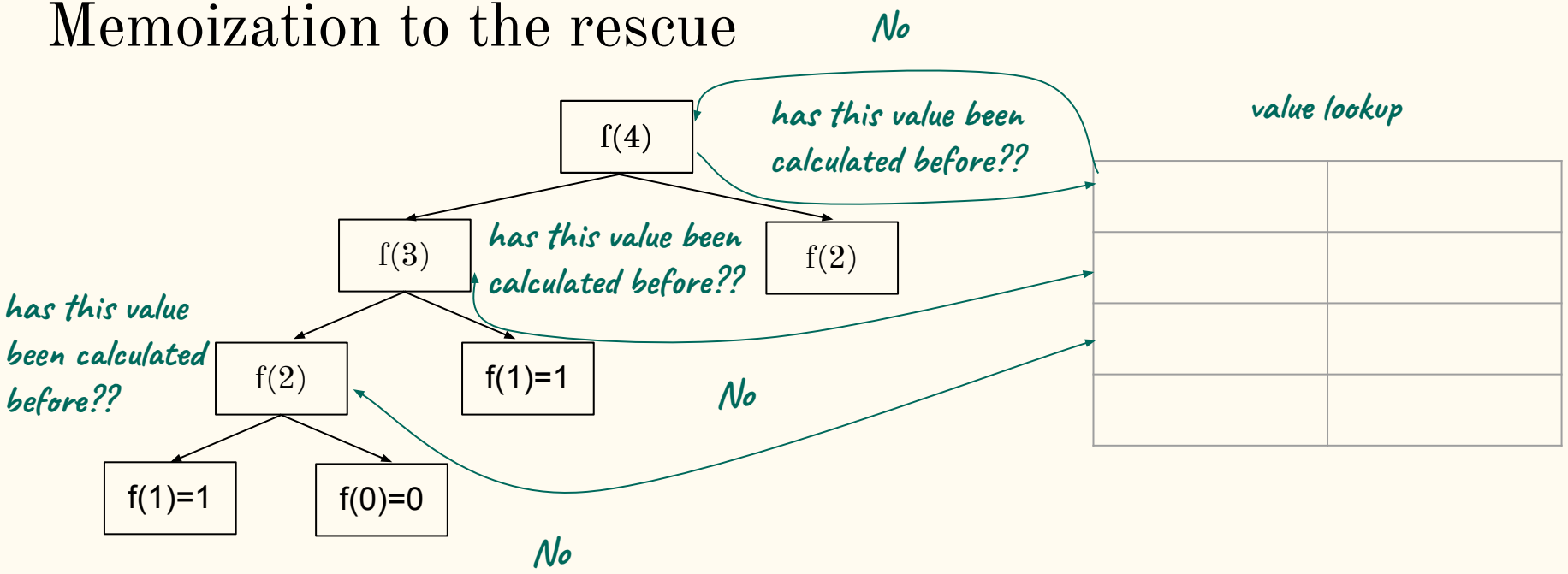
Repeated recursion calls



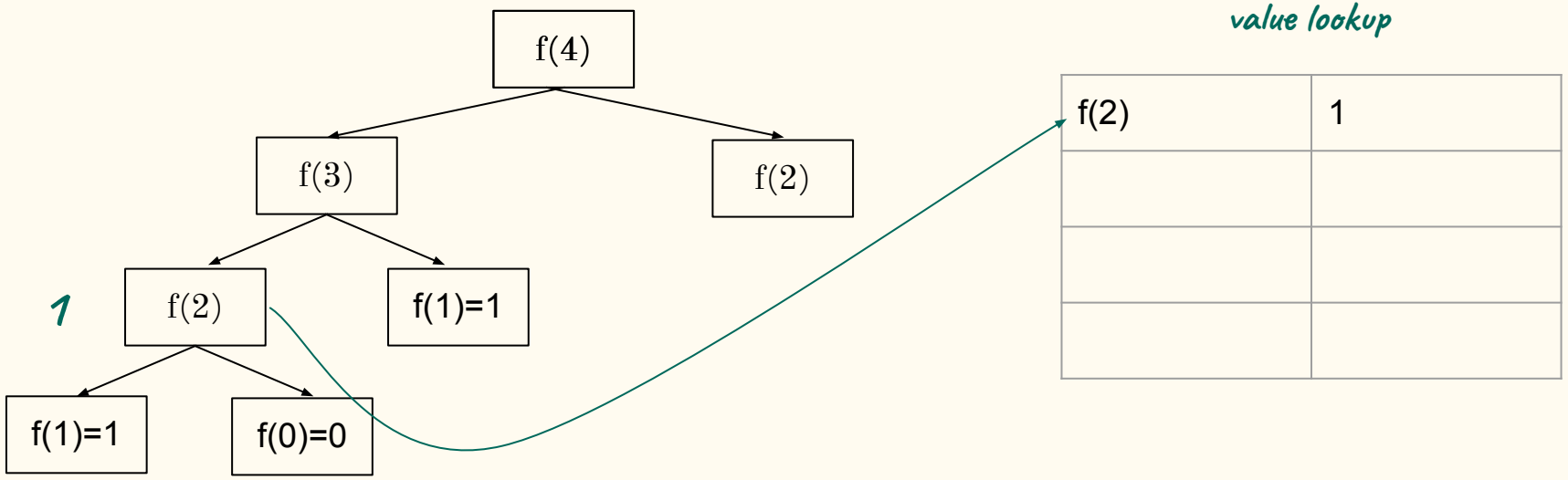
f => fib_recurse

```
long long fib_recurse(int n) {
    if (n < 2) return n;

    return fib_recurse(n - 1) + fib_recurse(n - 2);
}
```

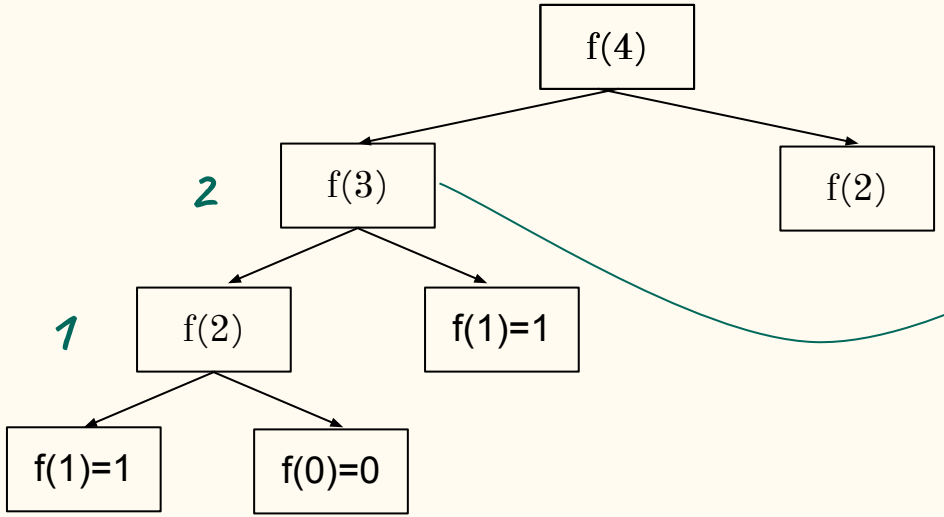
Memoization to the rescue



Memoization to the rescue



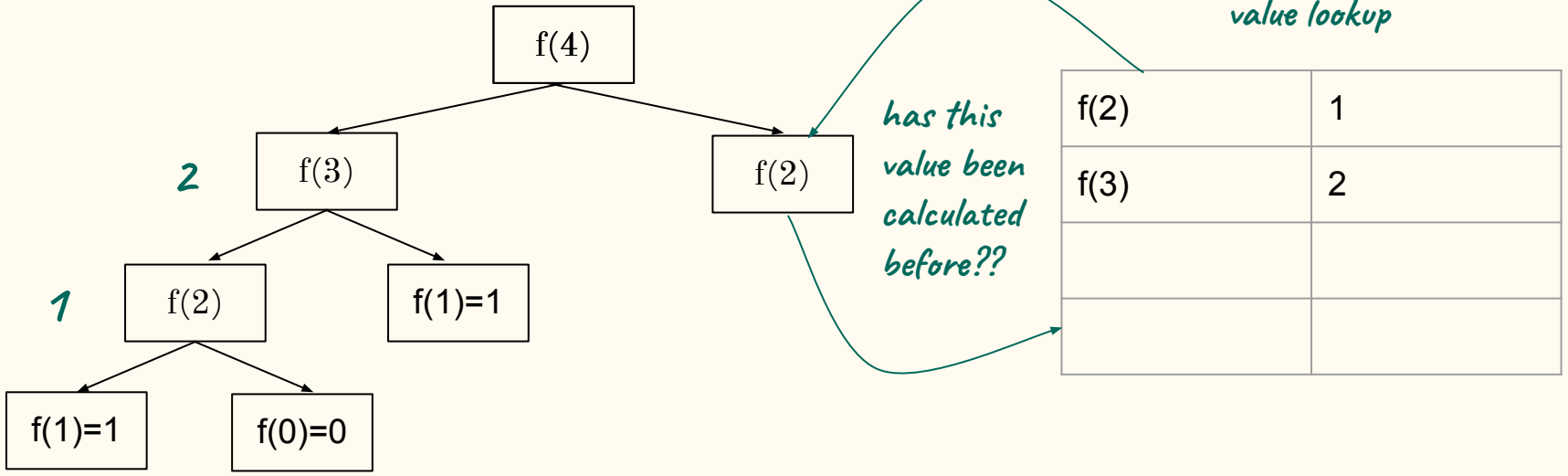
Memoization to the rescue



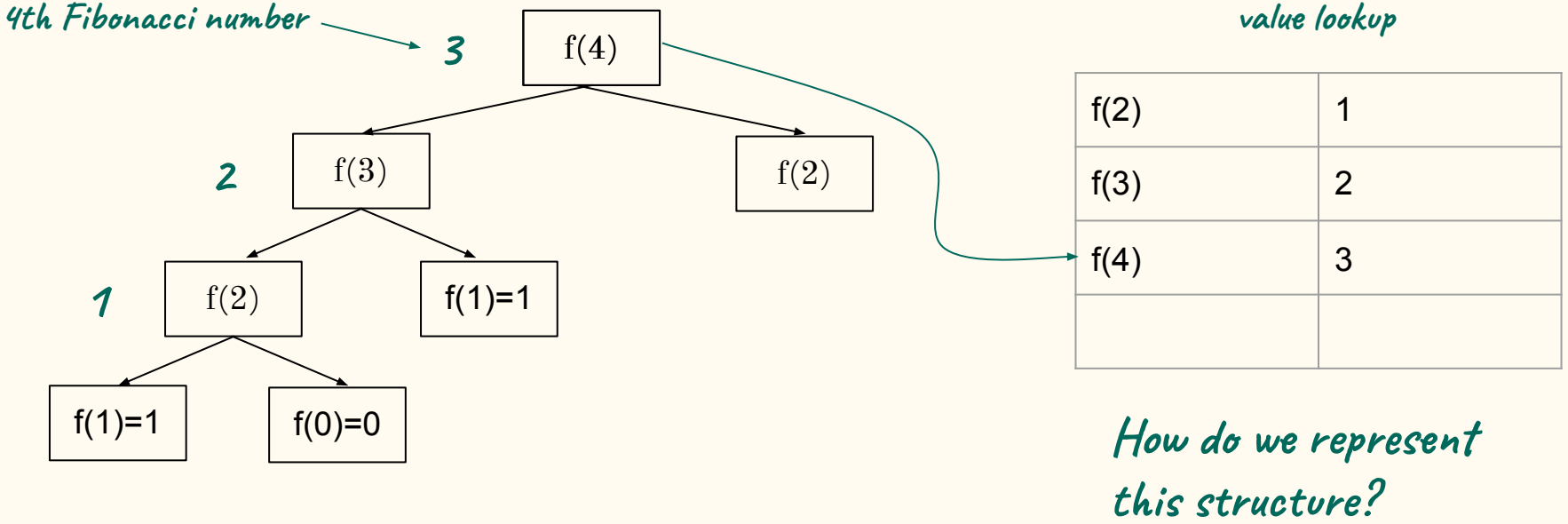
value lookup

f(2)	1
f(3)	2

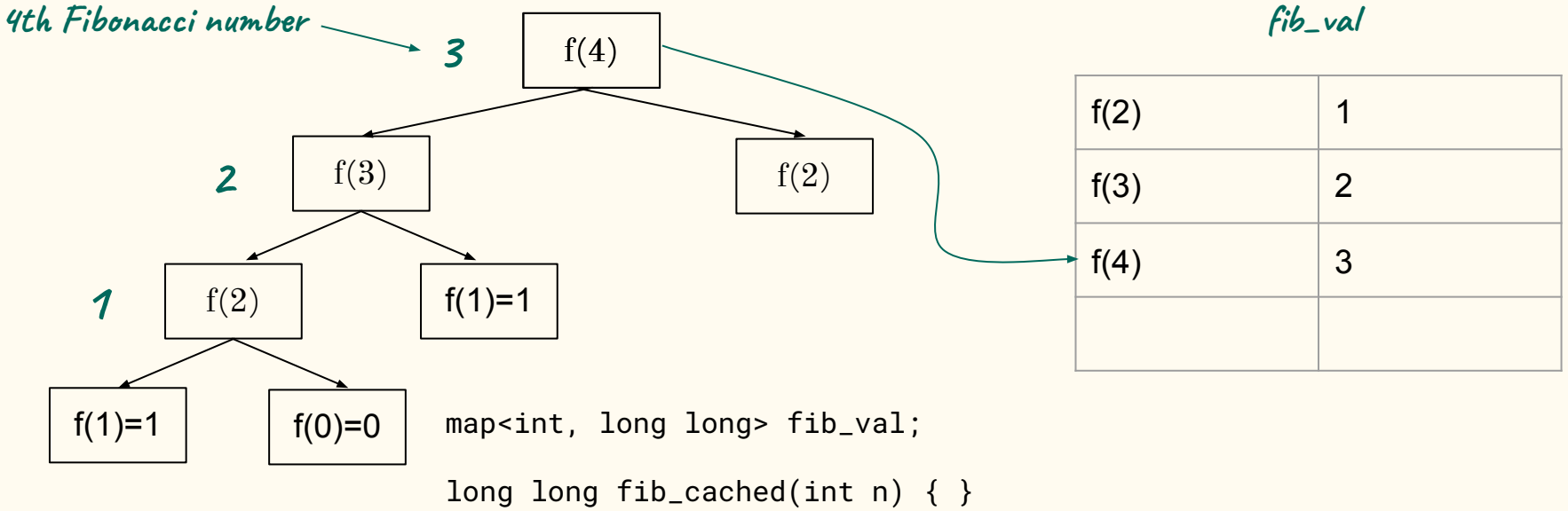
Memoization to the rescue



Memoization to the rescue



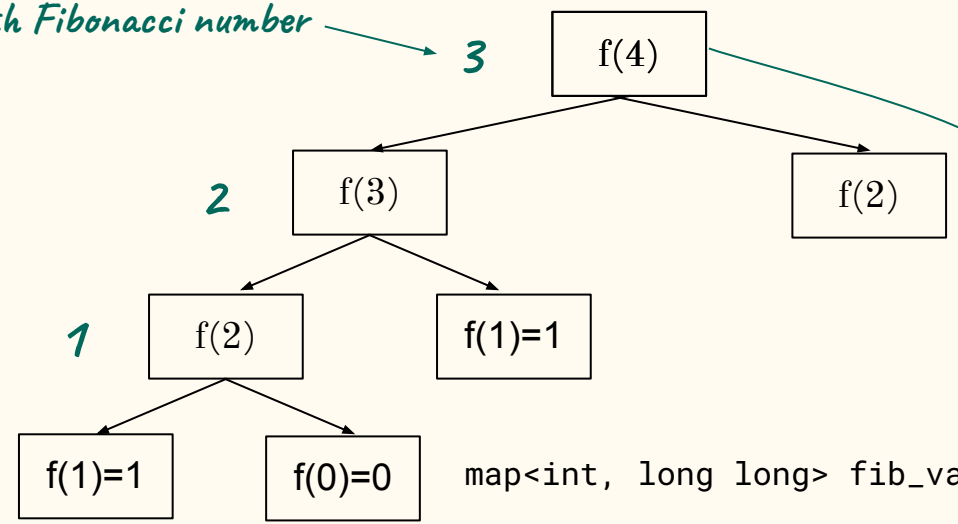
Memoization with a global object




```
map<int, long long> fib_val;

long long fib_cached(int n) { }
```

Memoization with a global object

4th Fibonacci number → 3



fib_val

f(2)	1
f(3)	2
f(4)	3

```
map<int, long long> fib_val;

long long fib_cached(int n) {
    if (n < 2) return n;
```

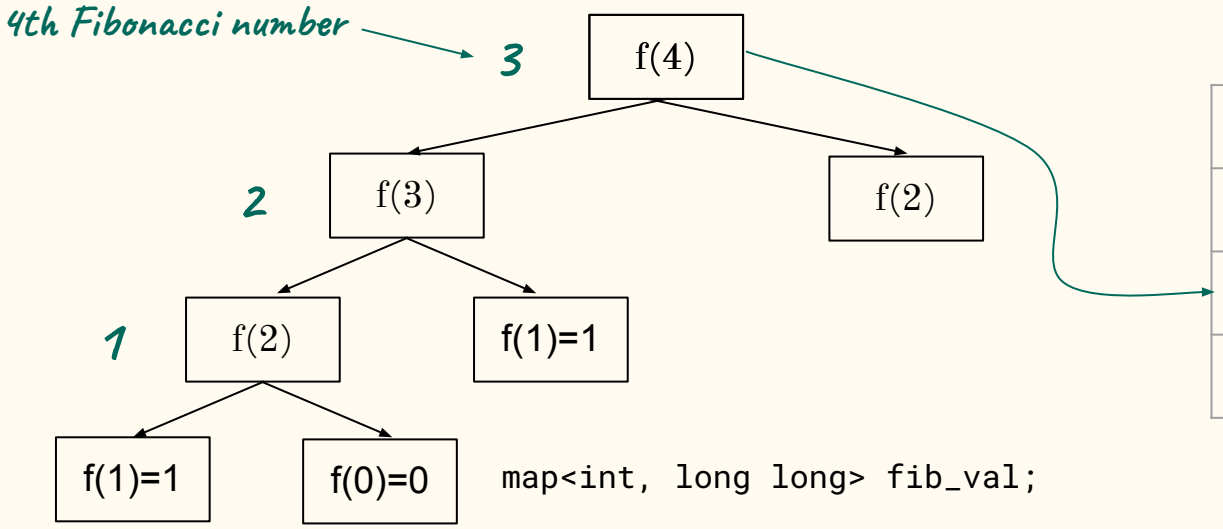
base case

nth Fibonacci number added to fib_val

```
    return fib_val[n] = fib_cached(n - 1) + fib_cached(n - 2);
}
```

recursive case

Memoization with a global object



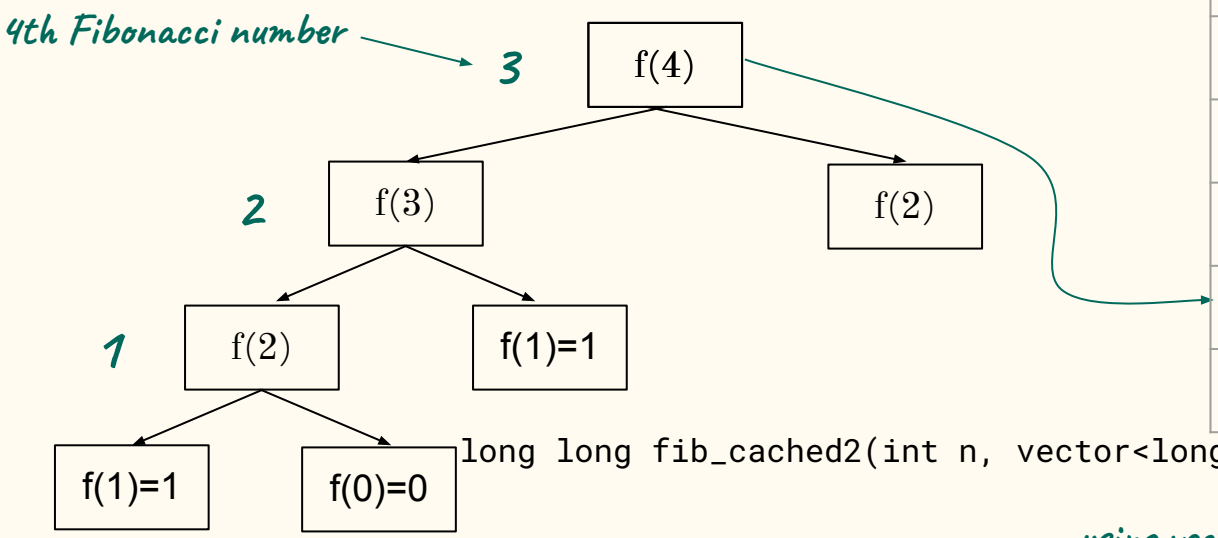
fib_val

f(2)	1
f(3)	2
f(4)	3

```
map<int, long long> fib_val;

long long fib_cached(int n) {
    if (n < 2) return n;
    map<int,long long>::iterator found = fib_val.find(n);
    if (found != fib_val.end()) return found->second;

    return fib_val[n] = fib_cached(n - 1) + fib_cached(n - 2);
}
```

save unnecessary recursive calls

Memoization with function parameters



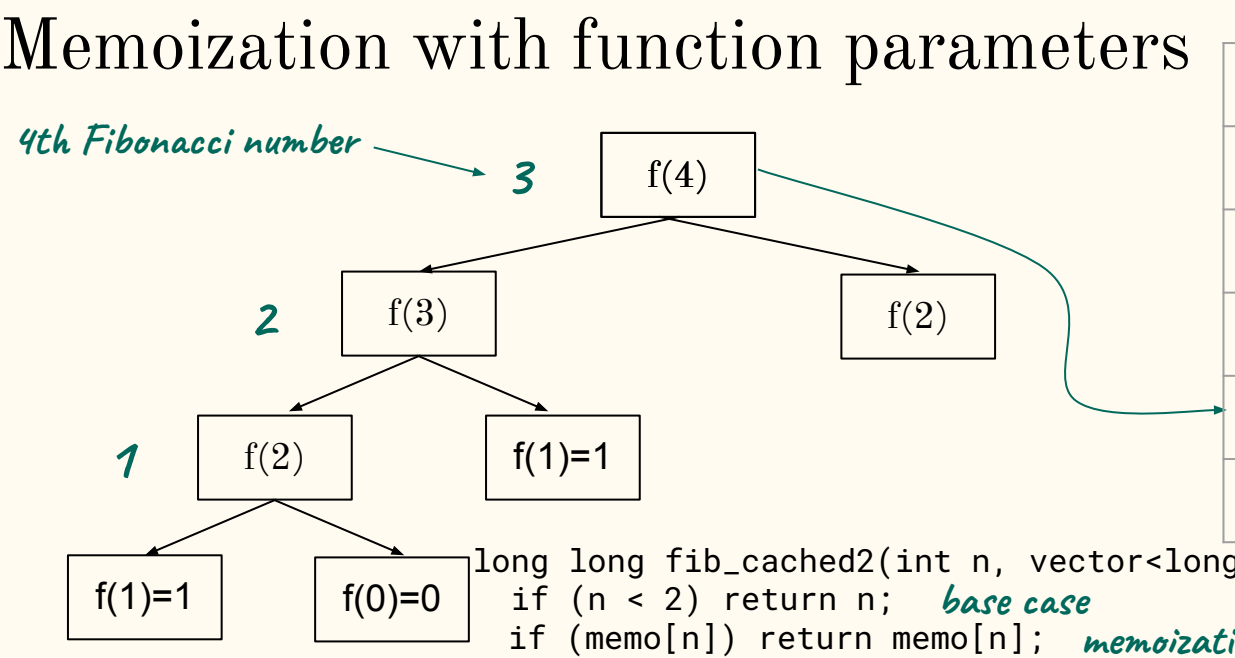
memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) { }
```

using vector& parameter

Memoization with function parameters



memo

0	
1	
2	1
3	2
4	3

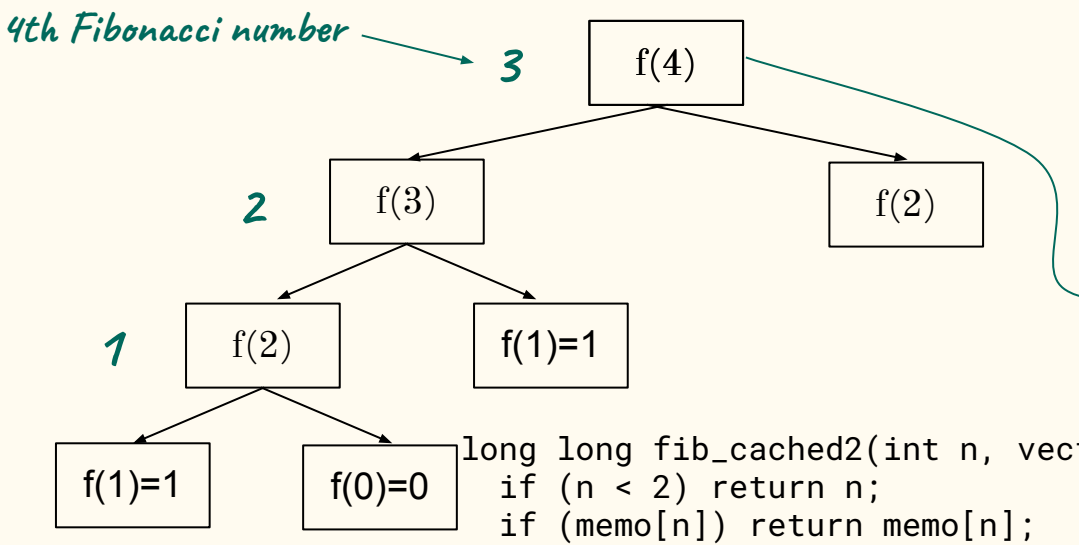
```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

// need to instantiate the vector
```

base case

memoization

recursive case

Memoization with function parameters

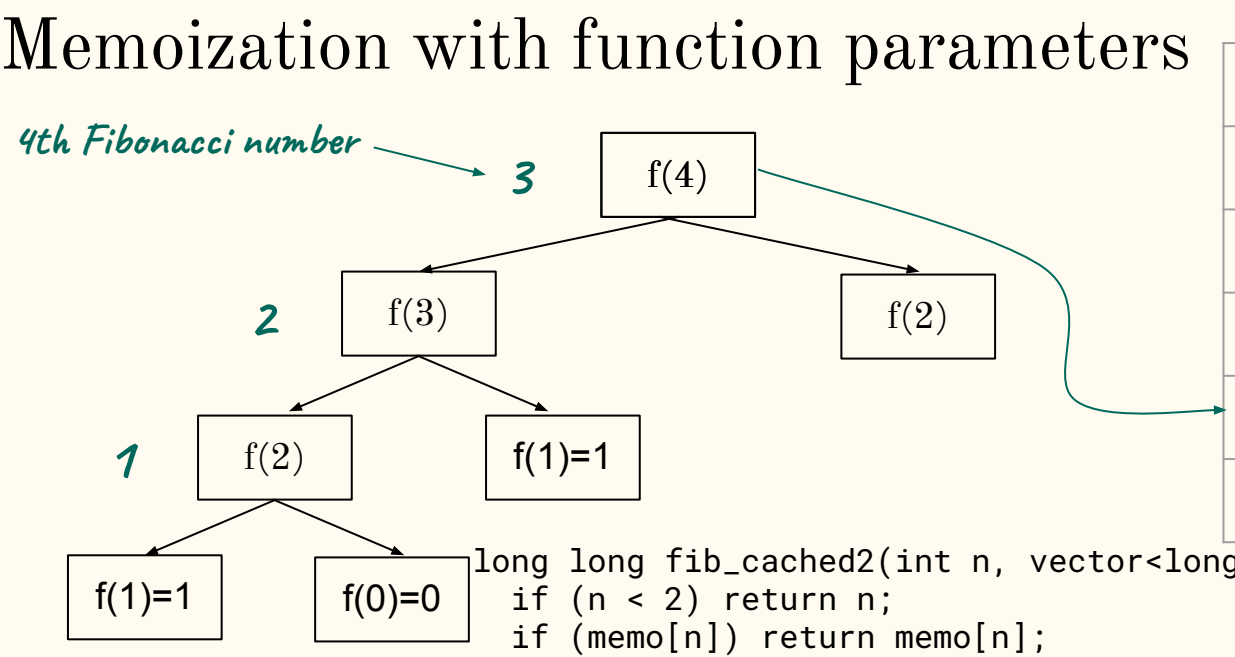


memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) { }
```

Memoization with function parameters

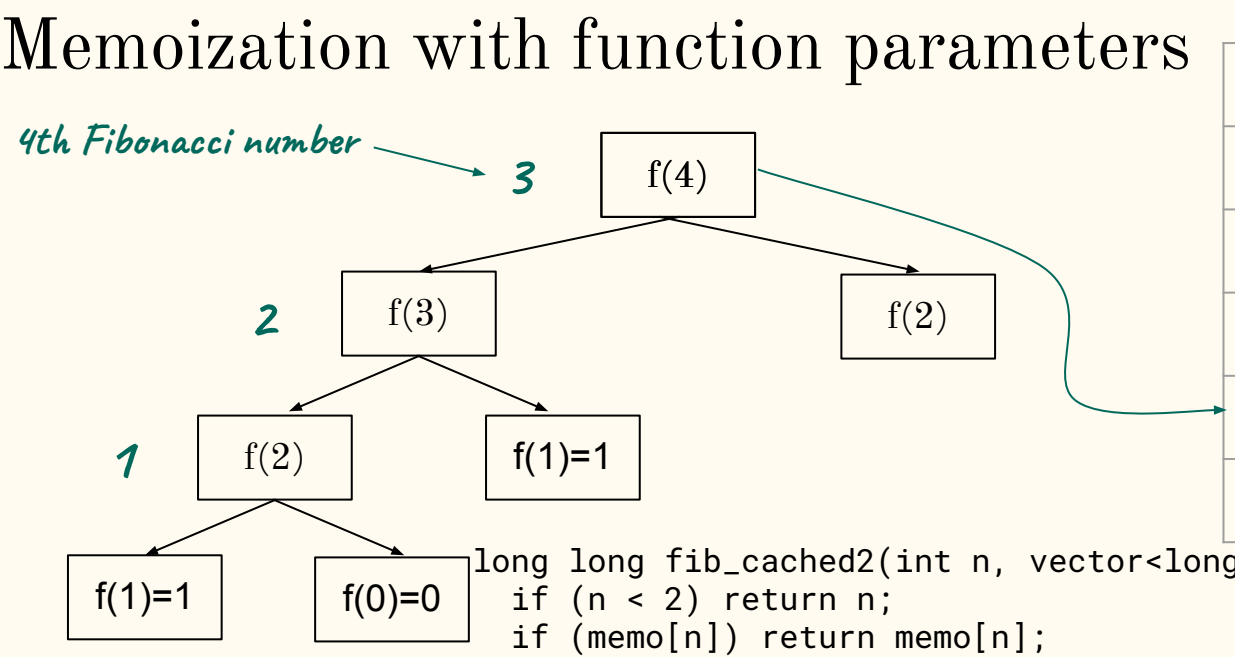


memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  // return the nth Fibonacci number
  ---
}
```

Memoization with function parameters

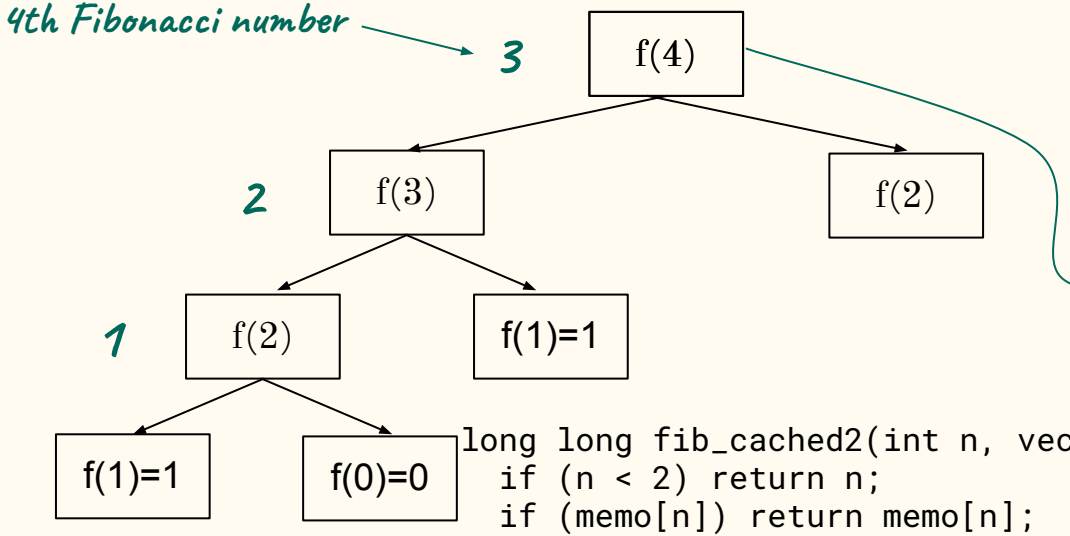


memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  // return the nth Fibonacci number
  return _15_
}
```

Which function call will evaluate to the nth Fibonacci number (replacing blank #15)?

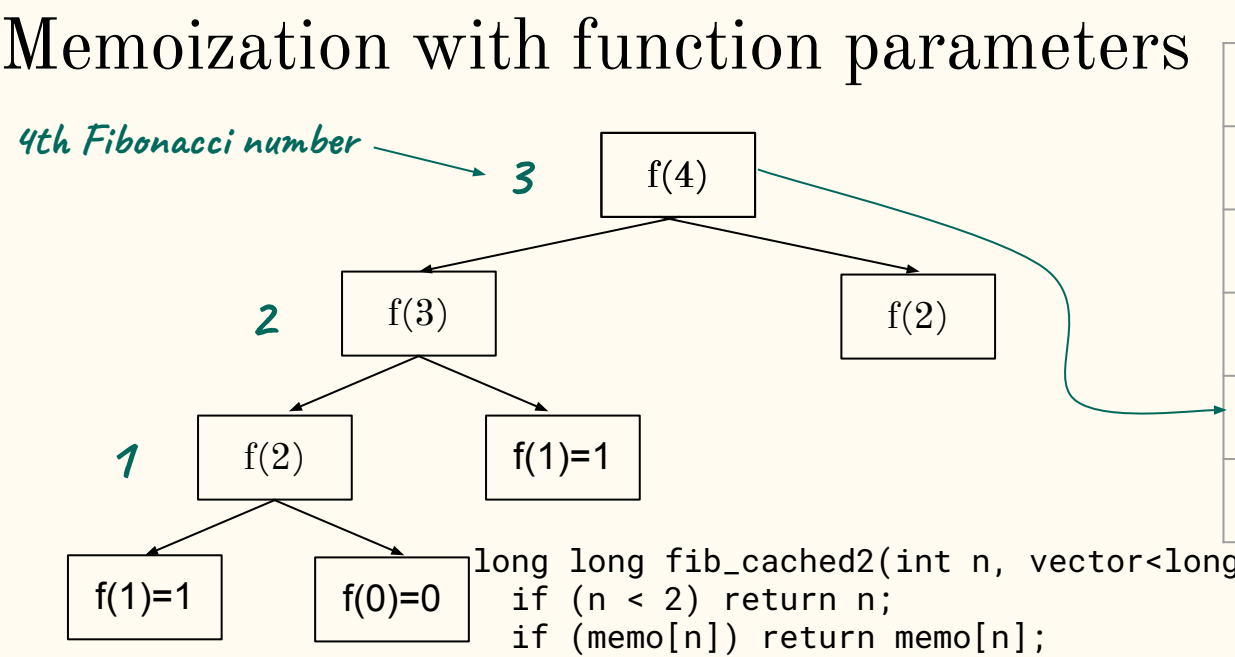


memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  // return the nth Fibonacci number
  return _15_;
}
```

Memoization with function parameters

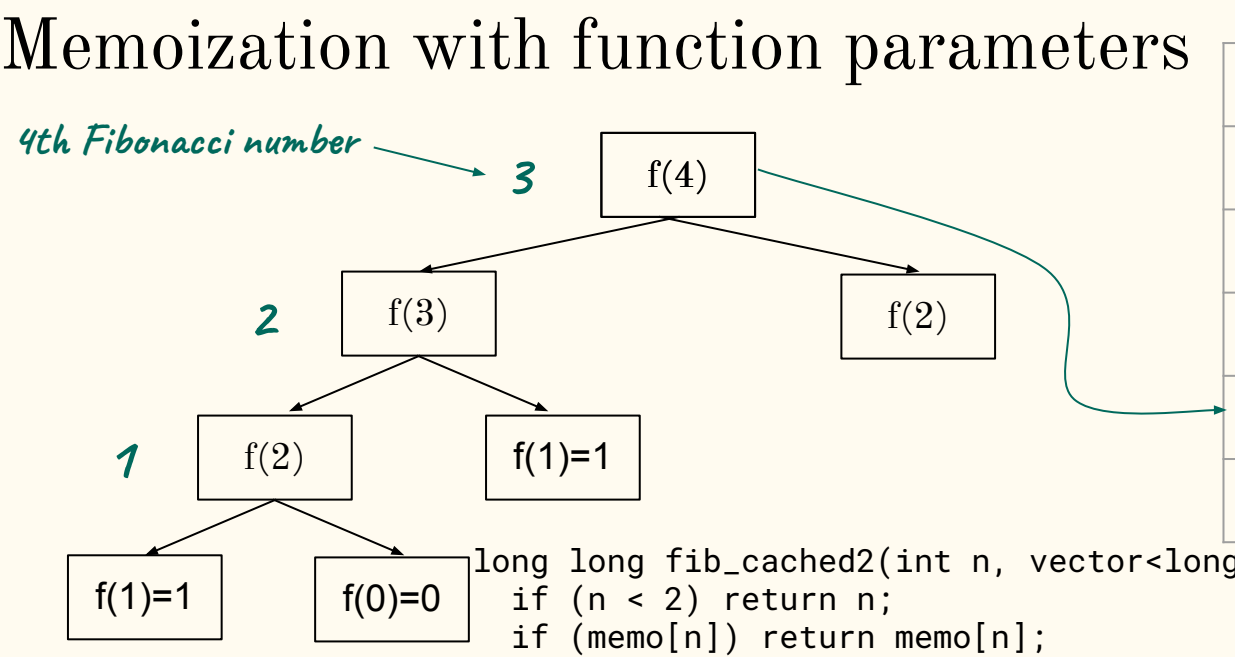


memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  // return the nth Fibonacci number
  return fib_cached2(n, memo);
}
```

Memoization with function parameters



memo

0	
1	
2	1
3	2
4	3

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}
```

Memoization with function parameters

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}

int main() {
    int fib_57 = ___;


}
```

Memoization with function parameters

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}

int main() {
    int fib_57 = _16_;



}
```

Memoization with function parameters

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}

int main() {
    int fib_57 = _16_;



}
```

Which function call evaluates to the 57th Fibonacci number (replacing blank #16)?

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}

int main() {
    int fib_57 = _16_;



}
```

Memoization with function parameters

```
long long fib_cached2(int n, vector<long long>& memo) {
  if (n < 2) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fib_cached2(n - 1, memo) + fib_cached2(n - 2, memo);
}

long long fib_memoized(int n) {
  vector<long long> memo(n + 1, 0);
  return fib_cached2(n, memo);
}

int main() {
    int fib_57 = fib_memoized(57);



}
```