The mechanics of recursion

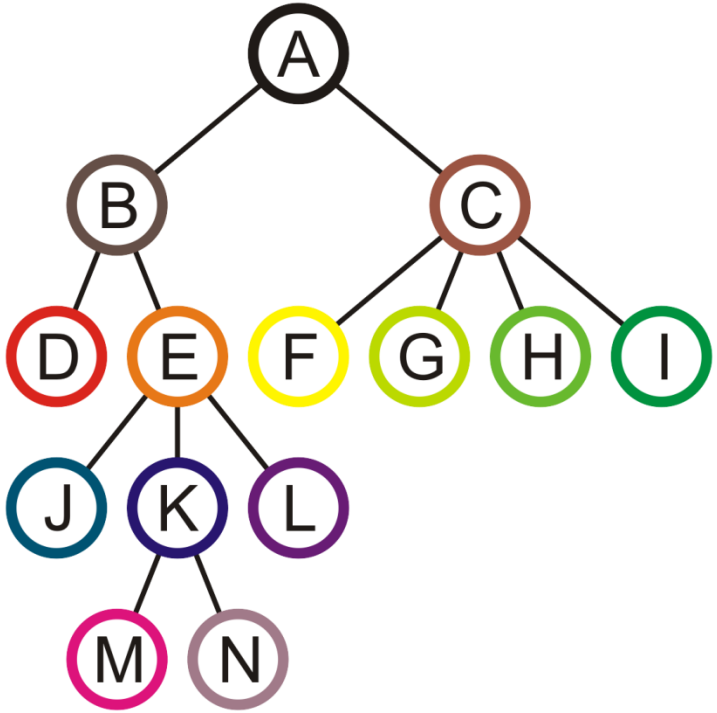
Outline

This topic will step through a recursive function call of the `Simple_tree` class

- We will recursively calculate `size()` on a tree
- We will be looking at the memory stack to see how local variables are saved
- In the end, the recursion will give us the number of nodes in the tree

Recursion

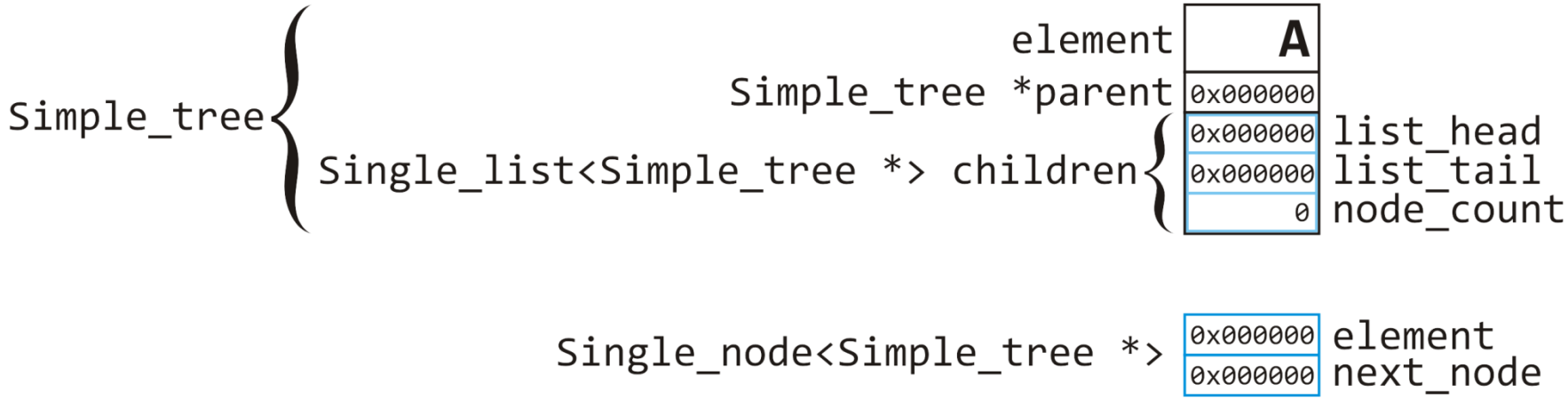
We will use the following `Simple_tree` implementation to demonstrate how recursive functions work



Recursion

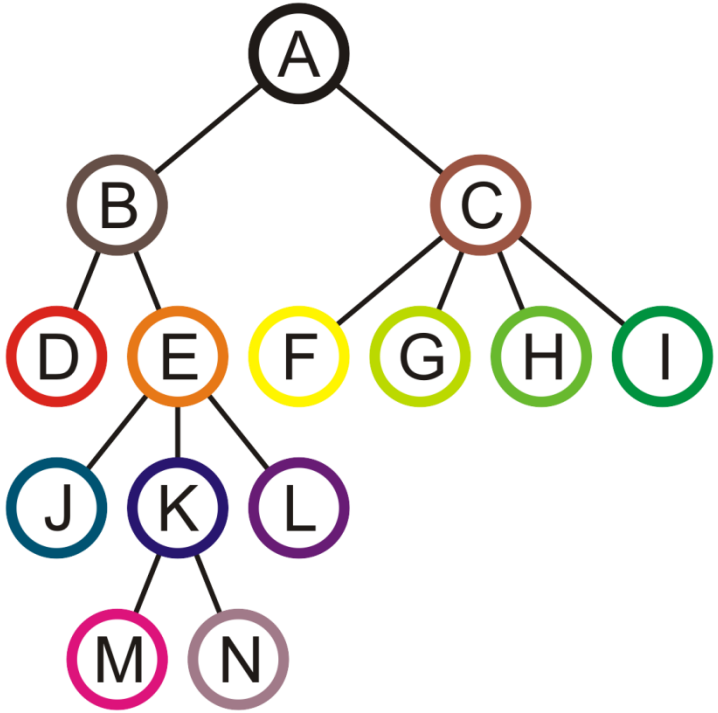
We will use the following `Simple_tree` implementation to demonstrate how recursive functions work

```
template <typename Type>
class Simple_tree {
    private:
        Type node_value;
        Simple_tree *parent;
        Single_list<Simple_tree *> children;
};
```

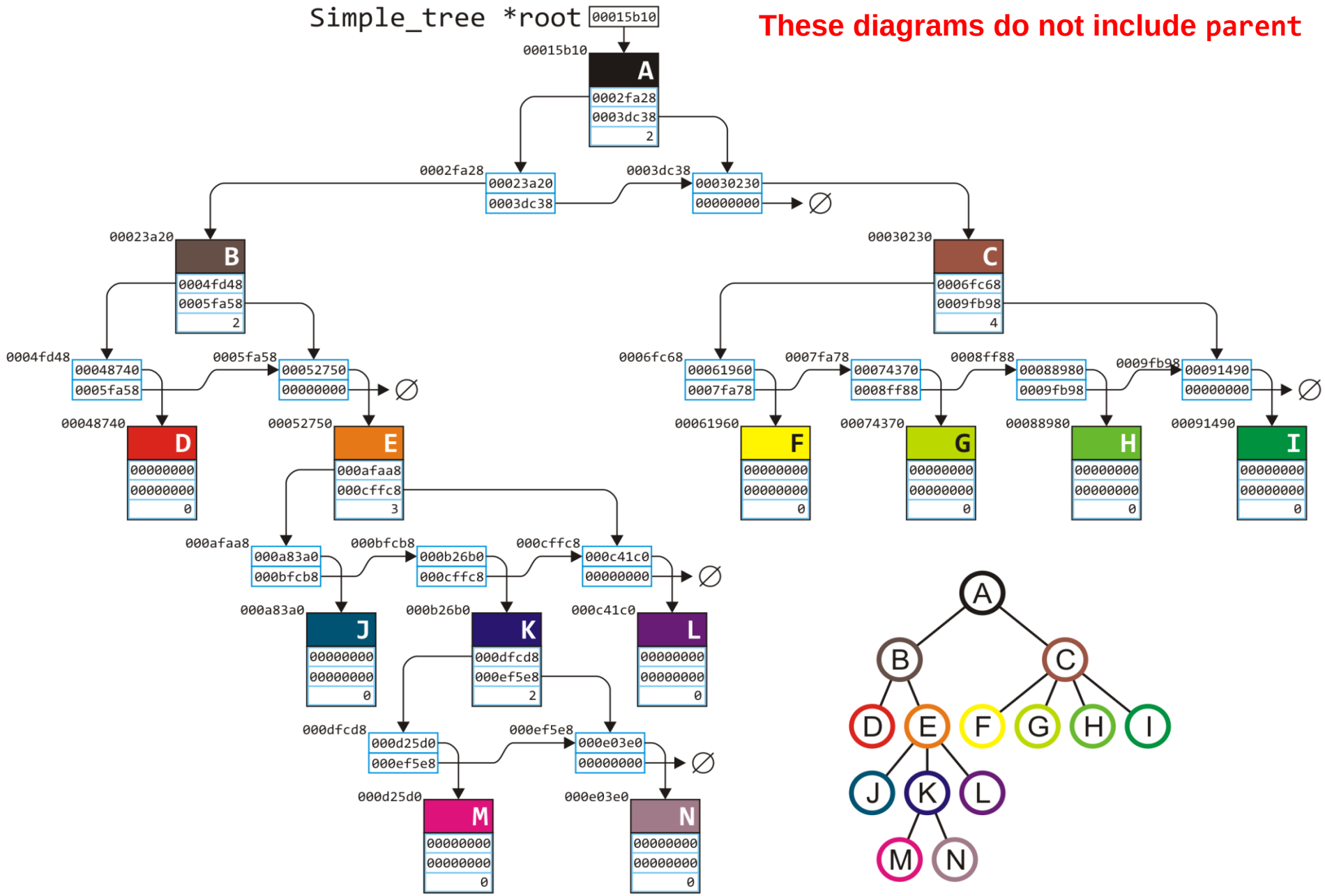


Recursion

The tree containing fourteen nodes



is represented by a reasonably complex object—assume the root node is stored in a pointer `Simple_tree *root;`


Simple_tree *root
00015b10
00015b10
A
0002fa28
0003dc38
2
0002fa28
00023a20
0003dc38
0003dc38
00030230
00000000
00023a20
B
0004fd48
0005fa58
2
00030230
C
0006fc68
0009fb98
4
0004fd48
00048740
0005fa58
0005fa58
00052750
00000000
0006fc68
00061960
0007fa78
0007fa78
00074370
0008ff88
0008ff88
00088980
0009fb98
0009fb98
00091490
00000000
00048740
D
00000000
00000000
0
00052750
E
000afaa8
000cffc8
3
00061960
F
00000000
00000000
0
00074370
G
00000000
00000000
0
00088980
H
00000000
00000000
0
00091490
I
00000000
00000000
0
000afaa8
000a83a0
000bfcb8
000bfcb8
000b26b0
000cffc8
000cffc8
000c41c0
00000000
000a83a0
J
00000000
00000000
0
000b26b0
K
000dfcd8
000ef5e8
2
000c41c0
L
00000000
00000000
0
000dfcd8
000d25d0
000ef5e8
000ef5e8
000e03e0
00000000
000d25d0
M
00000000
00000000
0
000e03e0
N
00000000
00000000
0
These diagrams do not include parent
A
B
C
D
E
F
G
H
I
J
K
L
M
N

Recursion

We will go through, step-by-step, what happens in memory when a recursive call to `int size()` is made:

```
template <typename Type>
int Simple_tree<Type>::size() const {
    int s = 1;

    for (
        Single_node<Simple_tree *> *ptr = children.head();
        ptr != 0; ptr = ptr->next()
    ) {
        s += ptr->value()->size();
    }

    return s;
}
```

Recursion

This function has

- two explicit local variables, and
- one implied local variable

These are

```
int s;
Single_node<Simple_tree *> *ptr
Simple_tree const *this
```

Recall that every time a member function of `Simple_tree` is called, the local variable `this` is assigned the address of the object on which the call is made

Recursion

you know that memory for local variables is assigned on a stack

- Each time a function is called, memory for that function is pushed on top of the stack

Example

Suppose the local variable root is assigned the address of the root of our tree: `00015b10`

The first thing to occur is that some function must call

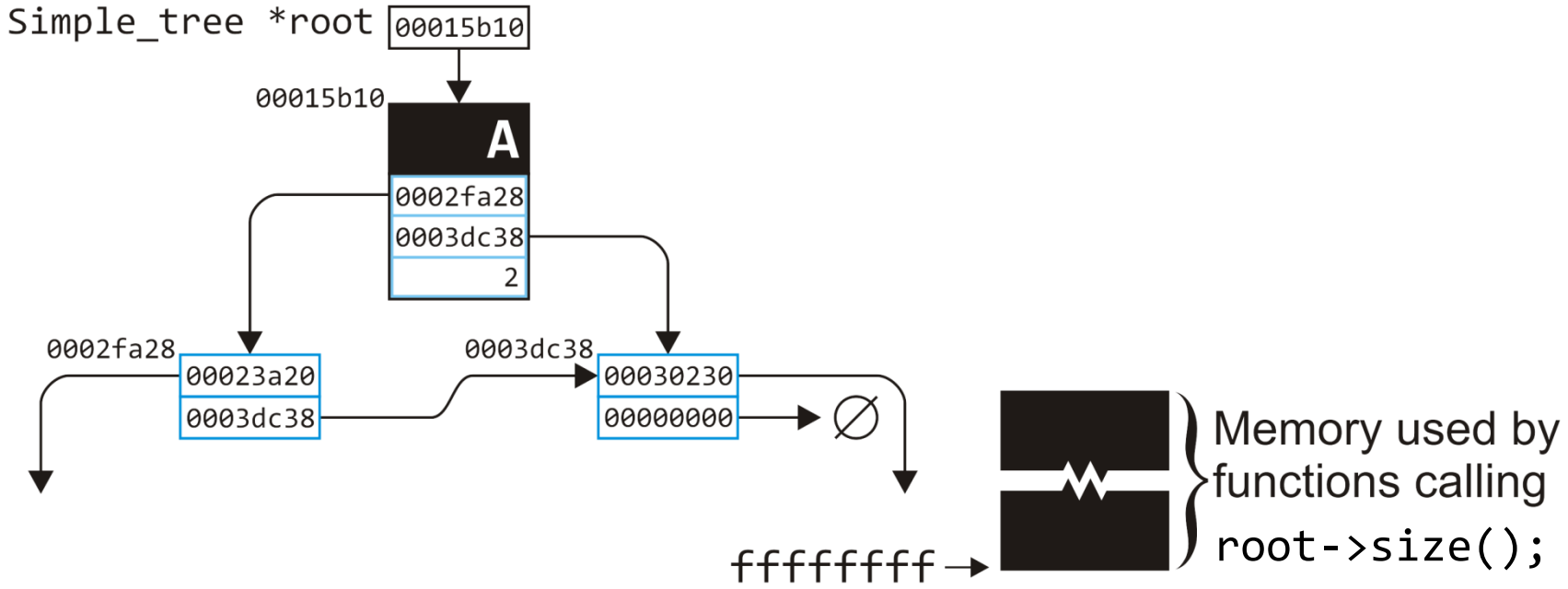
```
int h = root->size();
```

Example

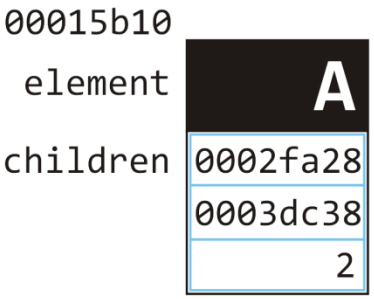
Suppose the local variable root is assigned the address of the root of our tree: `00015b10`

The first thing to occur is that some function must call

```
int h = root->size();
```



Example

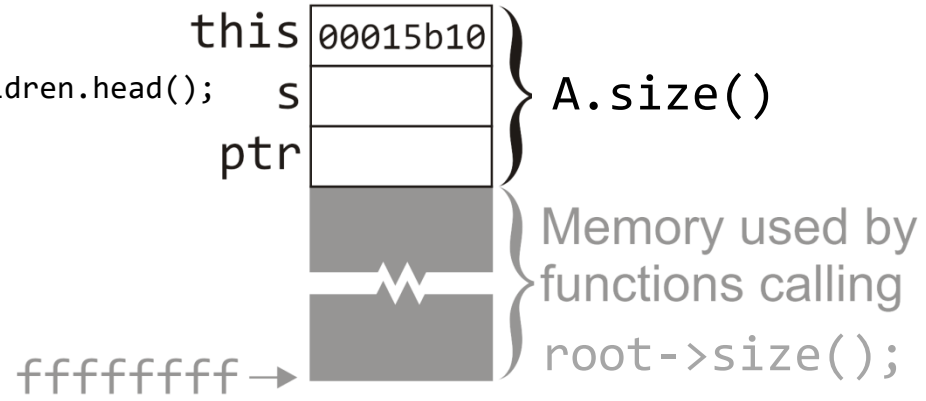


Memory is allocated on the stack for the call to `size` call with memory for `this`, `s`, and `ptr`

The implicit local variable `this` is assigned the address of the node

`Simple_tree const *this`

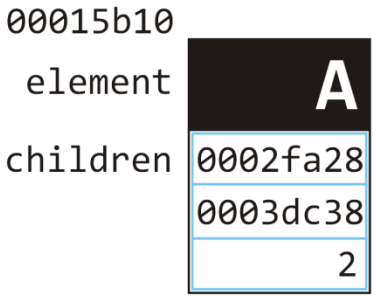
```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



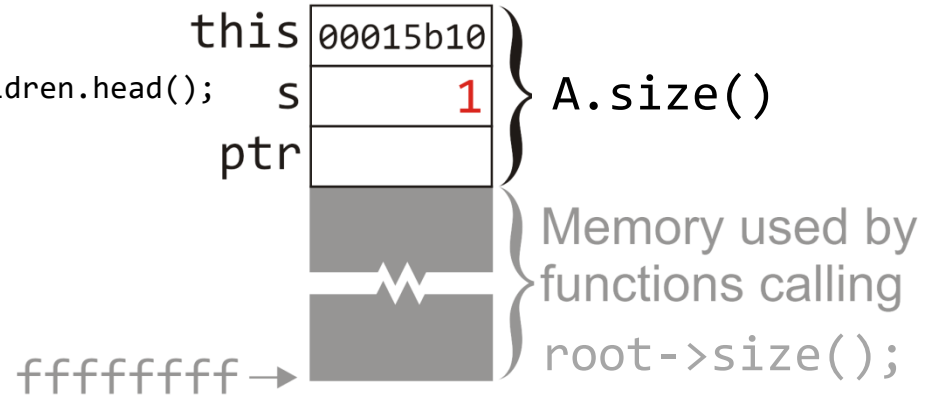
Example

First, the local variable `s` is set to 1

- This sets the corresponding location on the stack



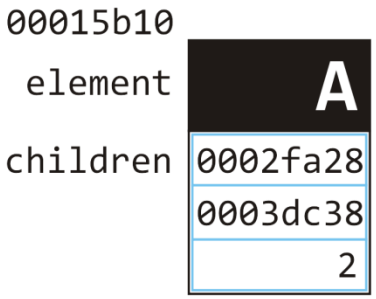

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

First, the local variable `s` is set to 1

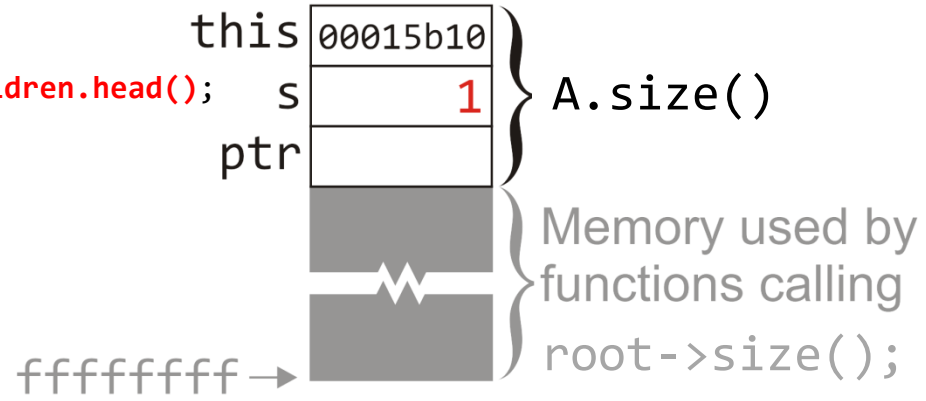
- This sets the corresponding location on the stack



The initialization of the for loop calls `children.head()`

- The variable `this` tells us where to find this linked list

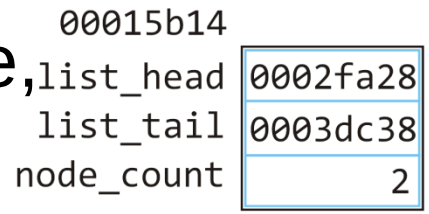
```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



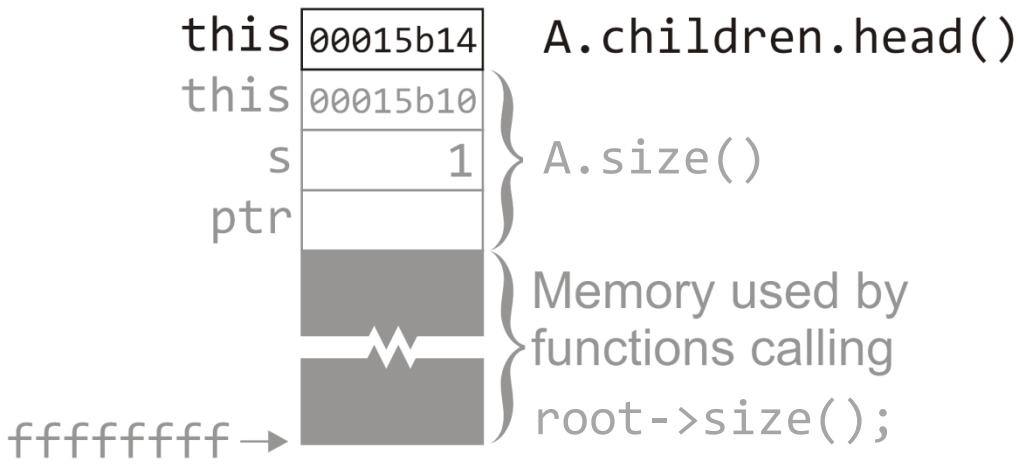
Example

For the call to `children.head()`, we must allocate new memory on the stack—in this case, it only stores the implicit local variable `this`

- Note that the address is `&A + 4`

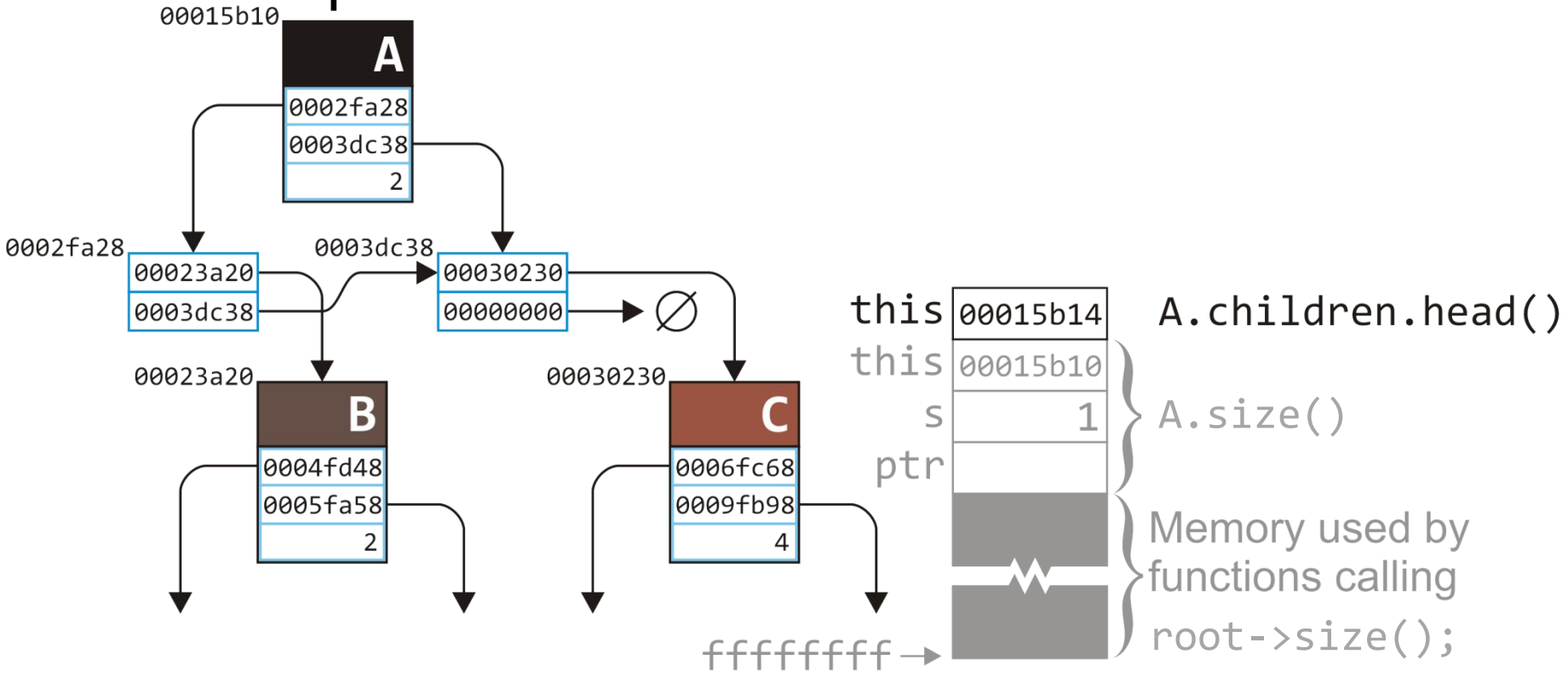



```
Simple_node *Single_list::head() const {
    return list_head;
}
```



Example

For the call to `children.head()`, we must allocate new memory on the stack—in this case, it only stores the implicit local variable `this`

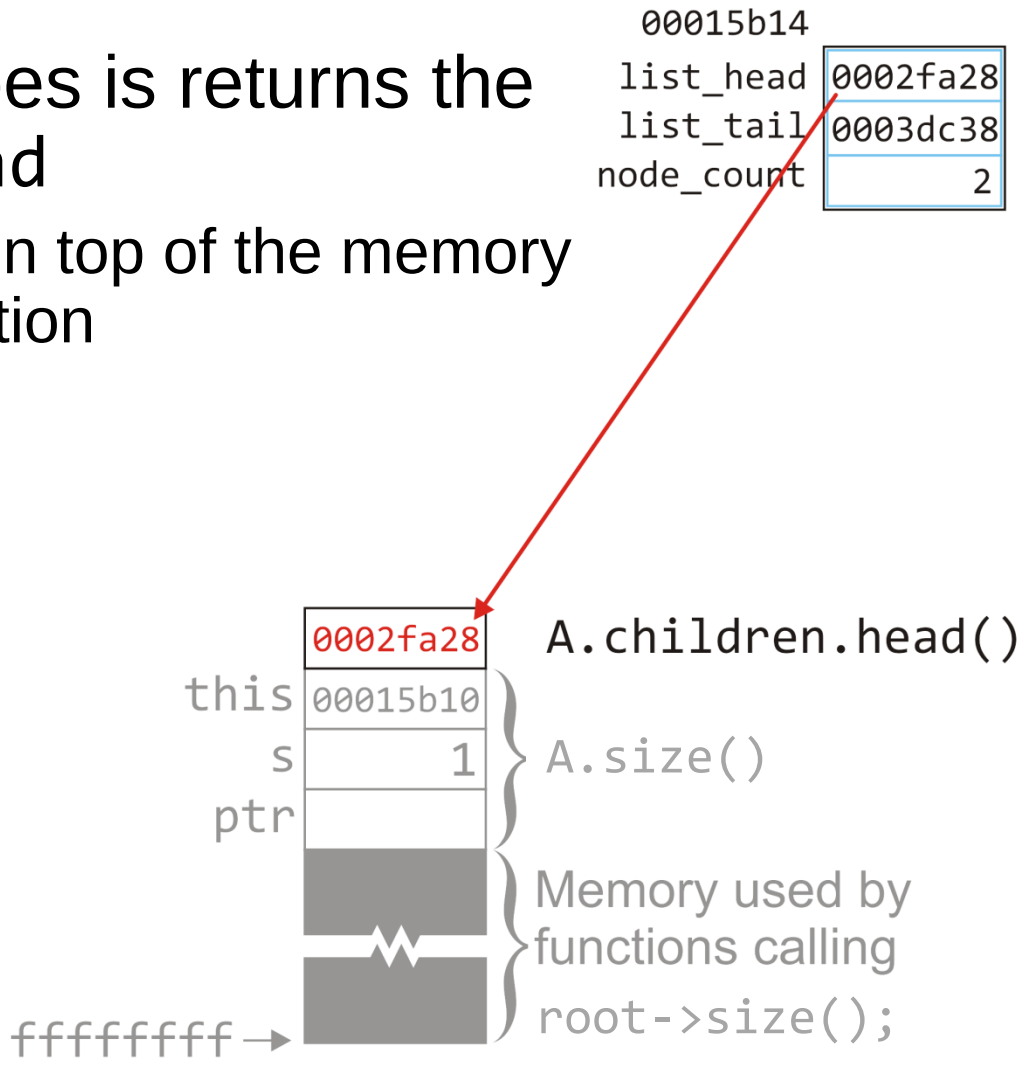


Example

All this member function does is returns the member variable `list_head`

- This is copied immediately on top of the memory allocated for the calling function

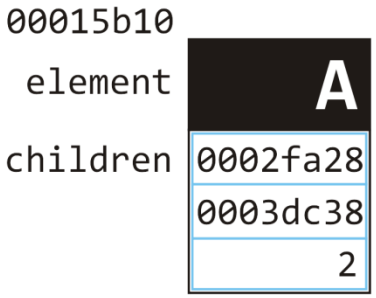
```
Simple_node *Single_list::head() const {
    return list_head;
}
```



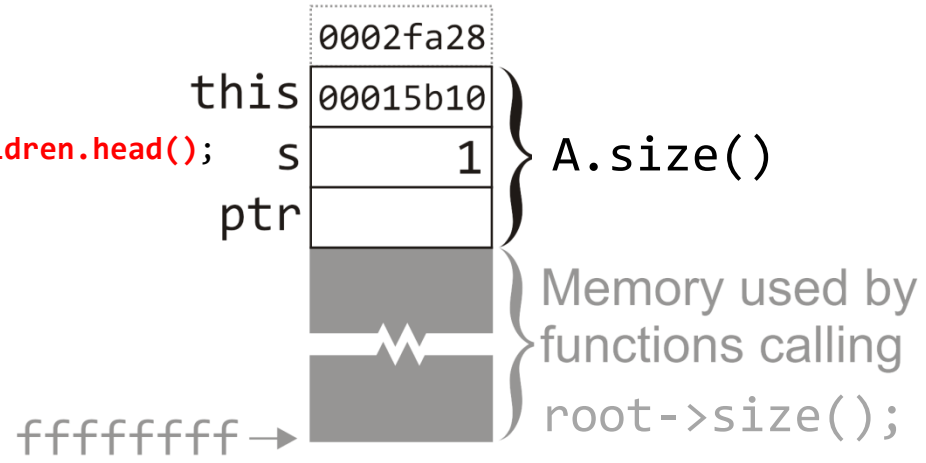
Example

We now return to the calling function

- We take the return variable and assign it to `ptr`

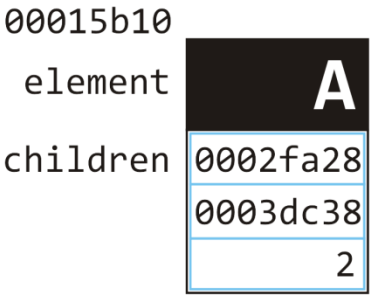



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

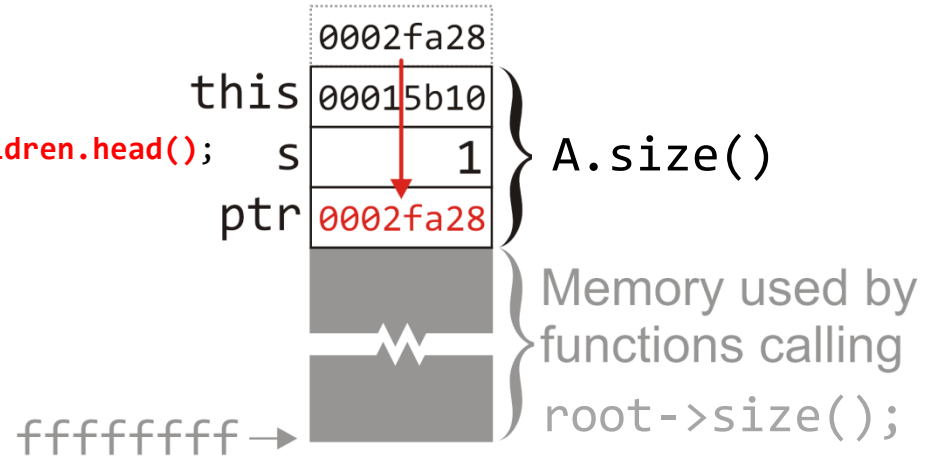


Example

The value stored immediately above the memory allocated for the current function call is copied to the memory allocated for `ptr`

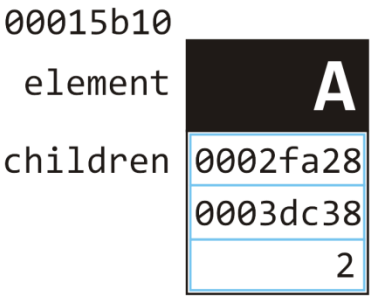



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

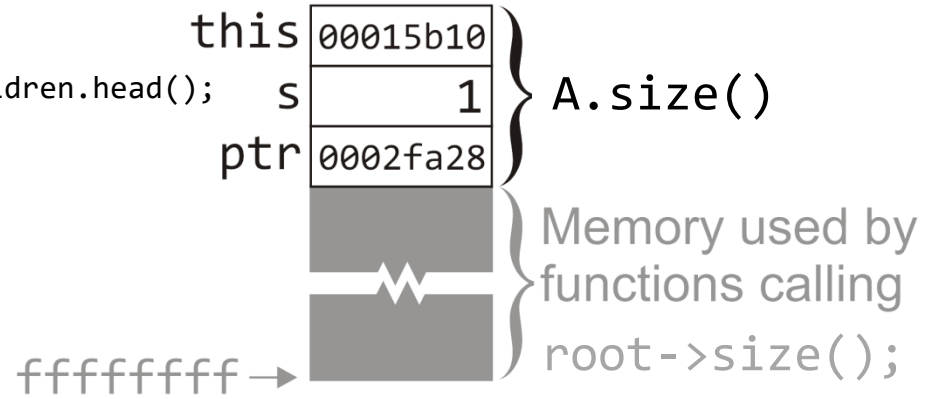


Example

We evaluate `ptr != 0`, determine this returns true, and we therefore proceed into the loop

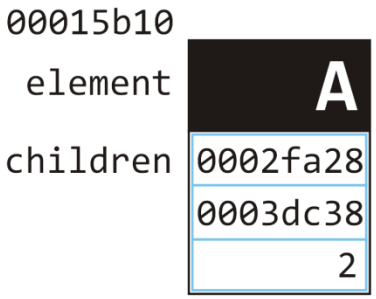



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



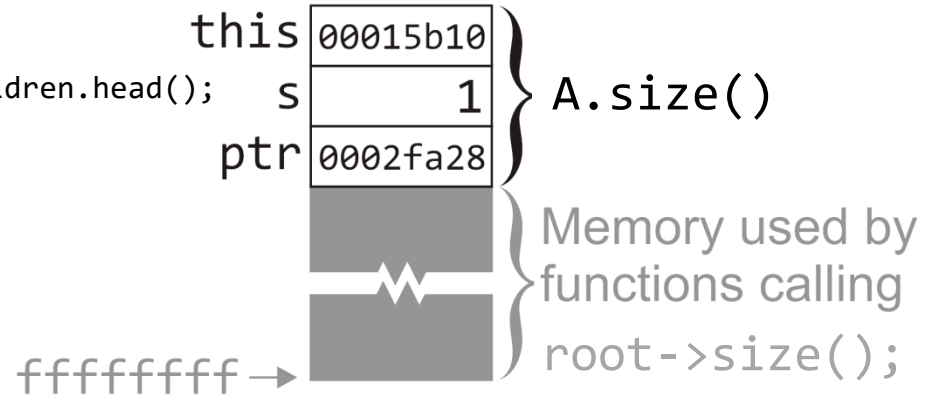
Example

We evaluate `ptr != 0`, determine this returns true, and we therefore proceed into the loop



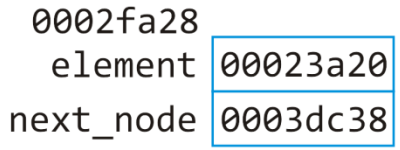
The first thing we do in the loop is call `ptr->value()`

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

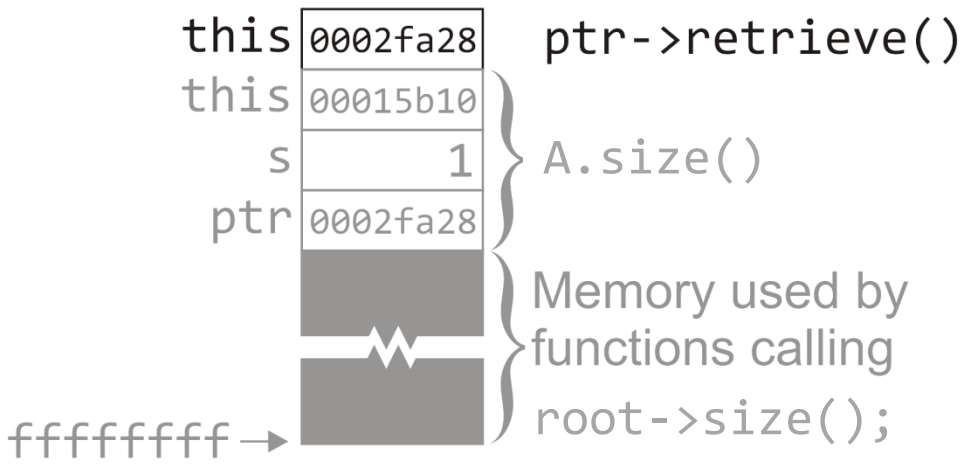


Example

In this call to `value()`, `this` is assigned the address of the node in question

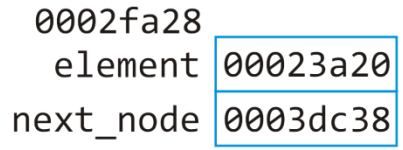


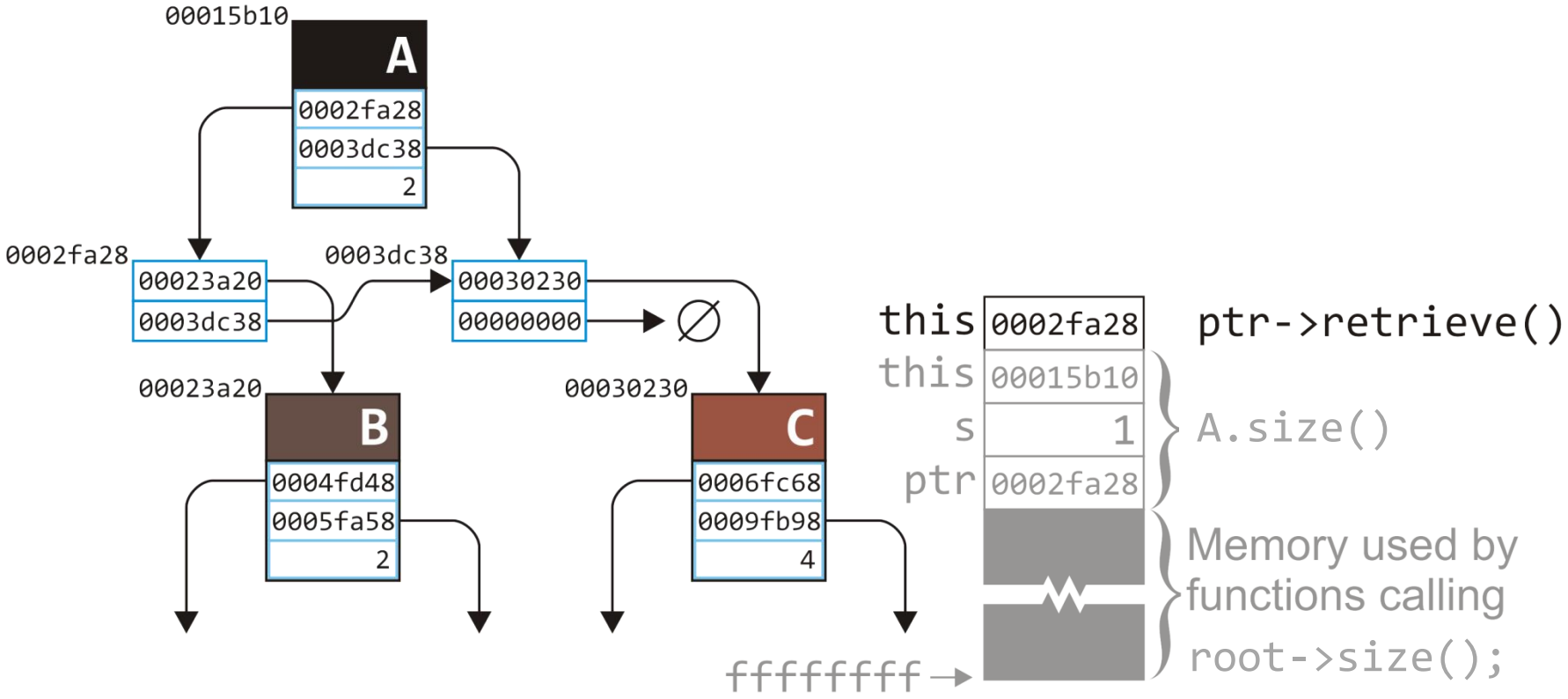

```
Simple_tree *Single_node::value() const {
   return node_value;
}
```



Example

In this call to `value()`, `this` is assigned the address of the node in question

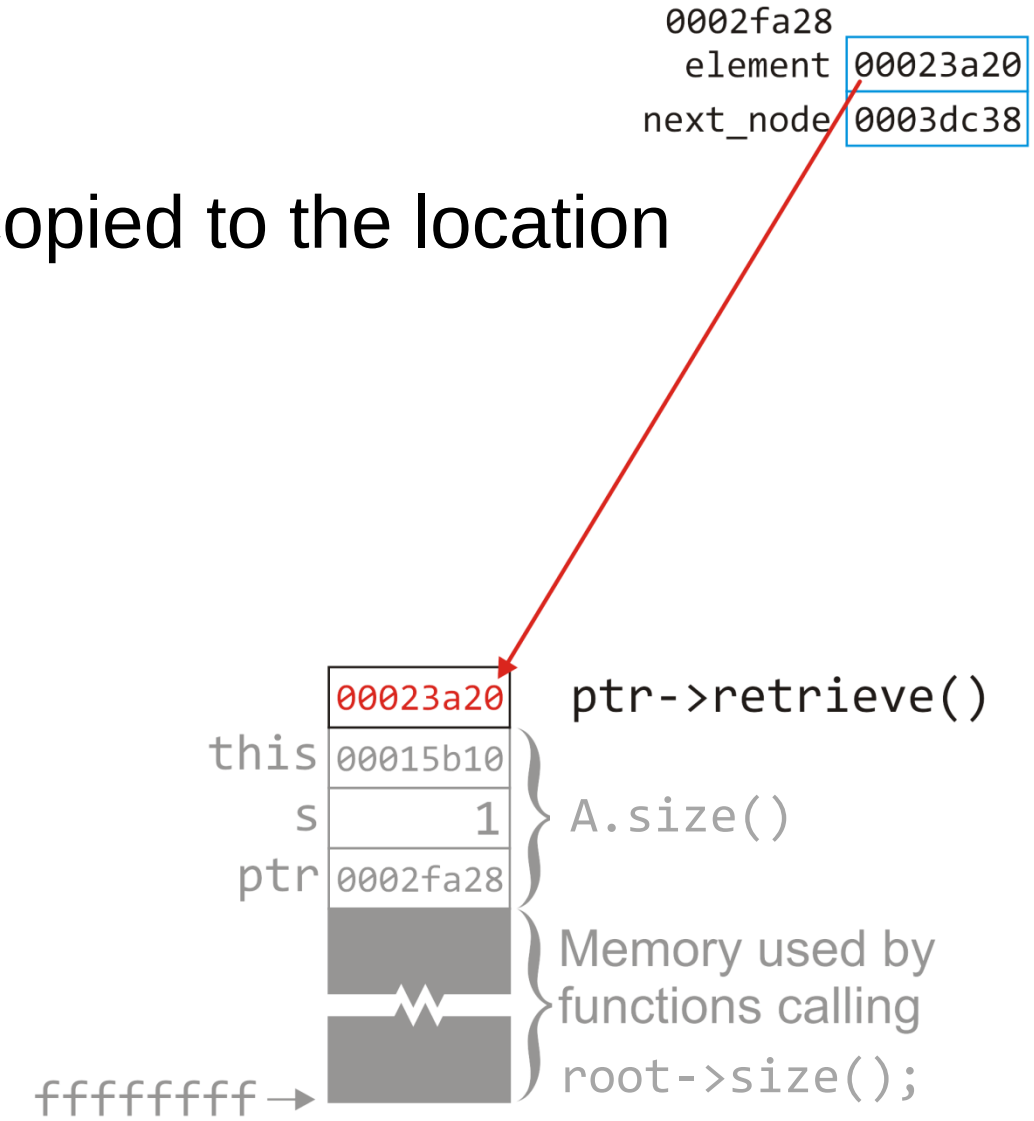




Example

The entry under `node_value` is copied to the location of the return value

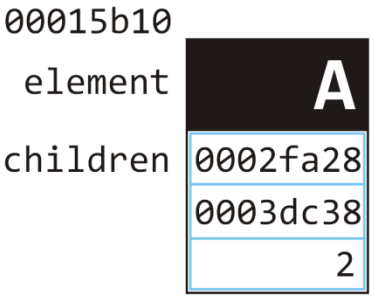
```
Simple_tree *Single_node::value() const {
   return node_value;
}
```



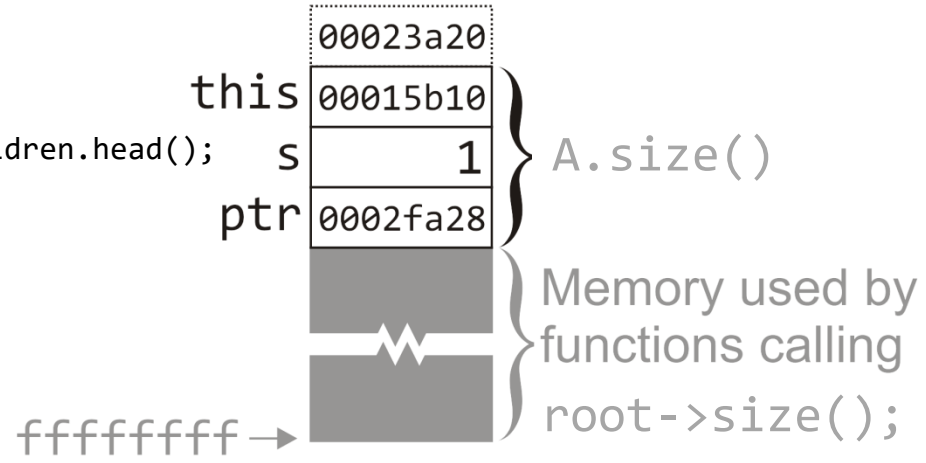
Example

We now return to the calling function

- We don't store this value; instead, we immediately recursively call `size()` on the node stored at this location



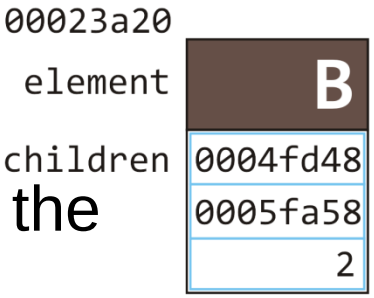

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

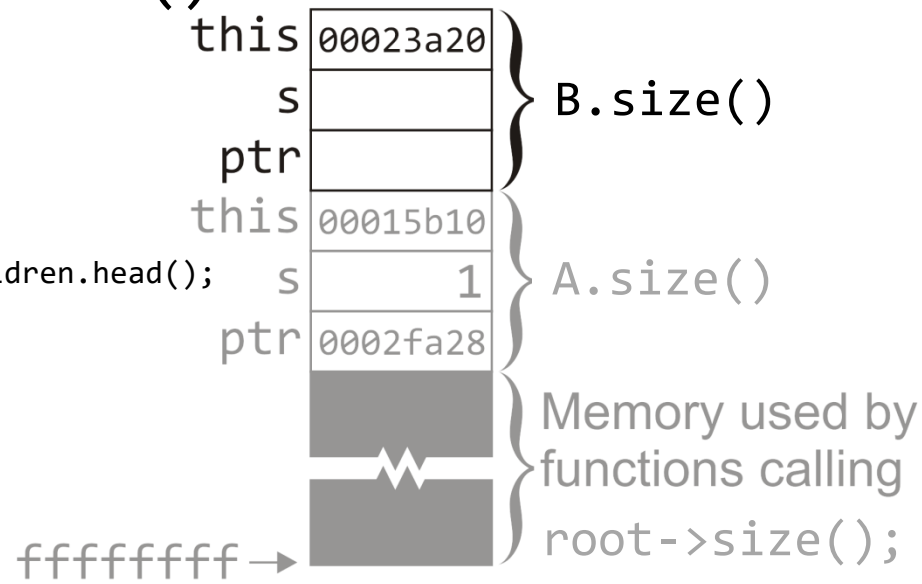
So now we have a second call to `size()`

- We must allocate memory for the three local variables `this`, `s`, and `ptr` and `this` is assigned the address of the object we are calling `size()` on—in this case, node B



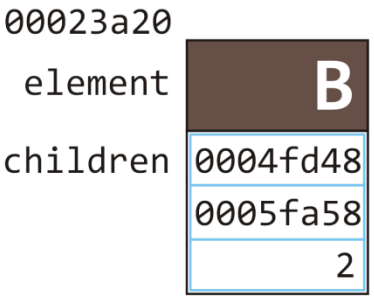
- Note that in this function call, we have no access to any of the local variables of the initiating call to `size()` on node A

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

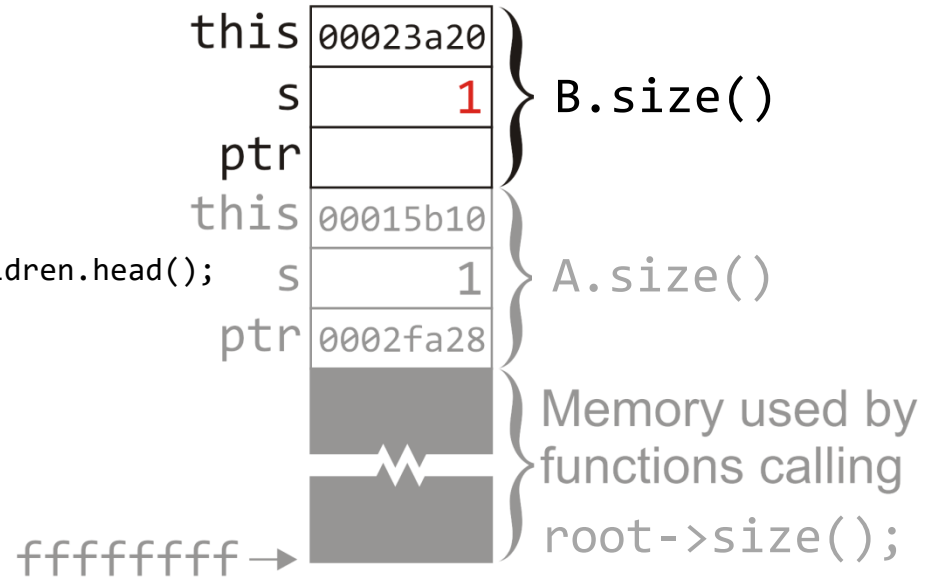


Example

Like before, we step through the function and begin by initializing s

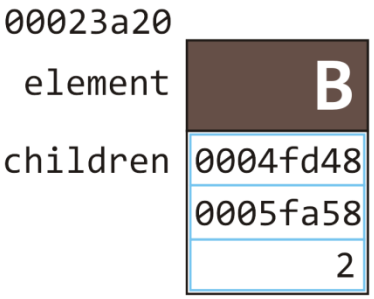



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



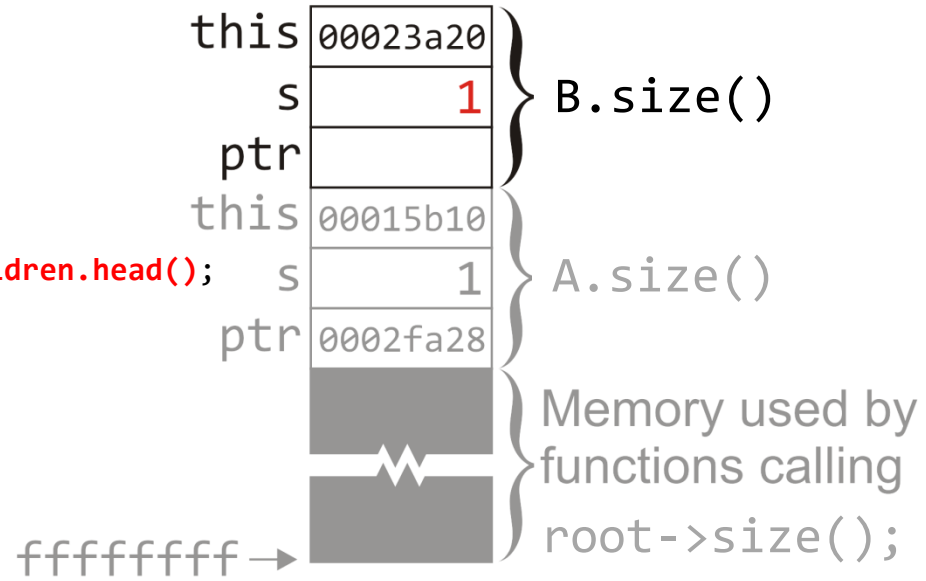
Example

Like before, we step through the function and begin by initializing `s`



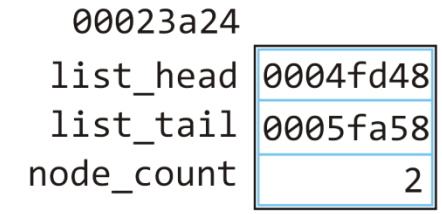
Again, we initialize the loop, but to the liked list associated with this node

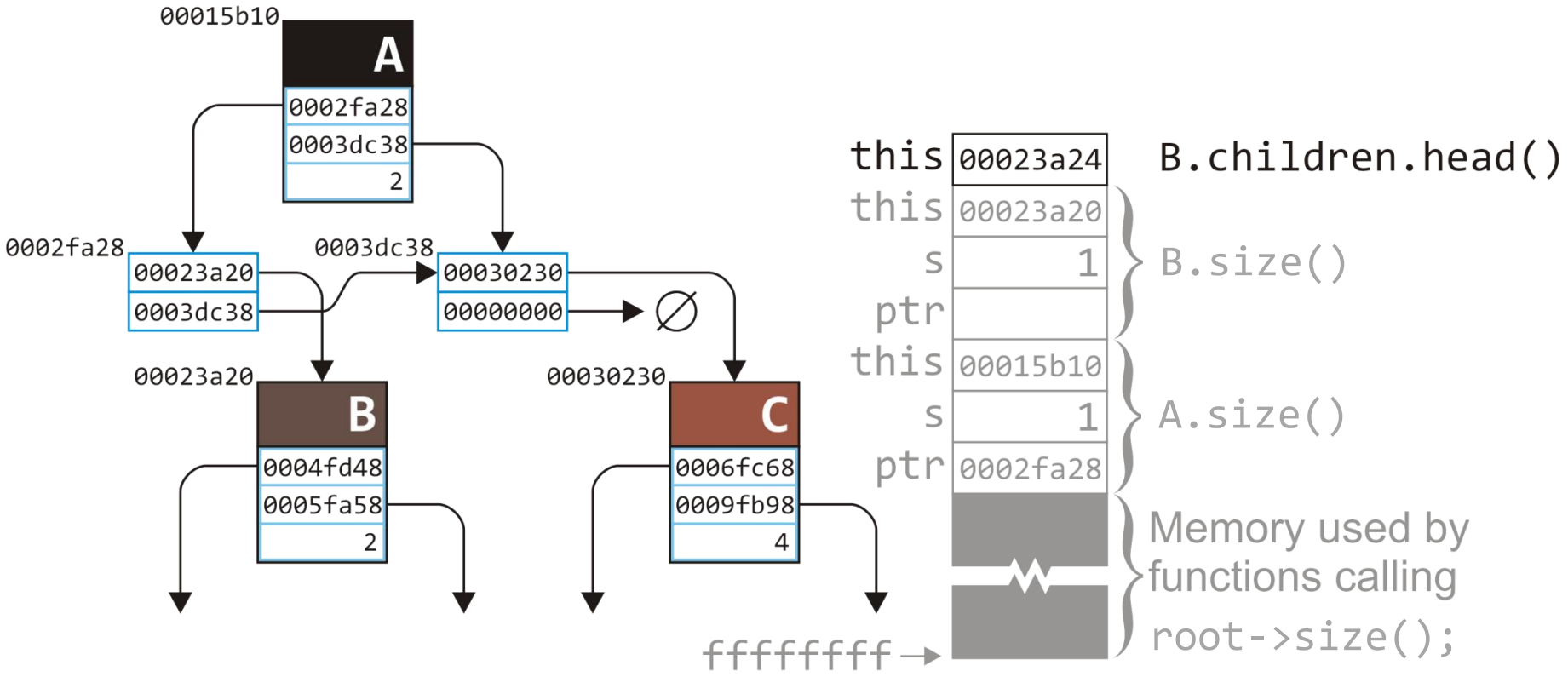
```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

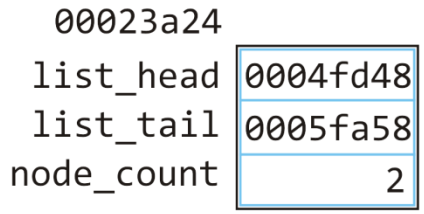
The call to `children.head()` will return the `list_head` of this singly linked list





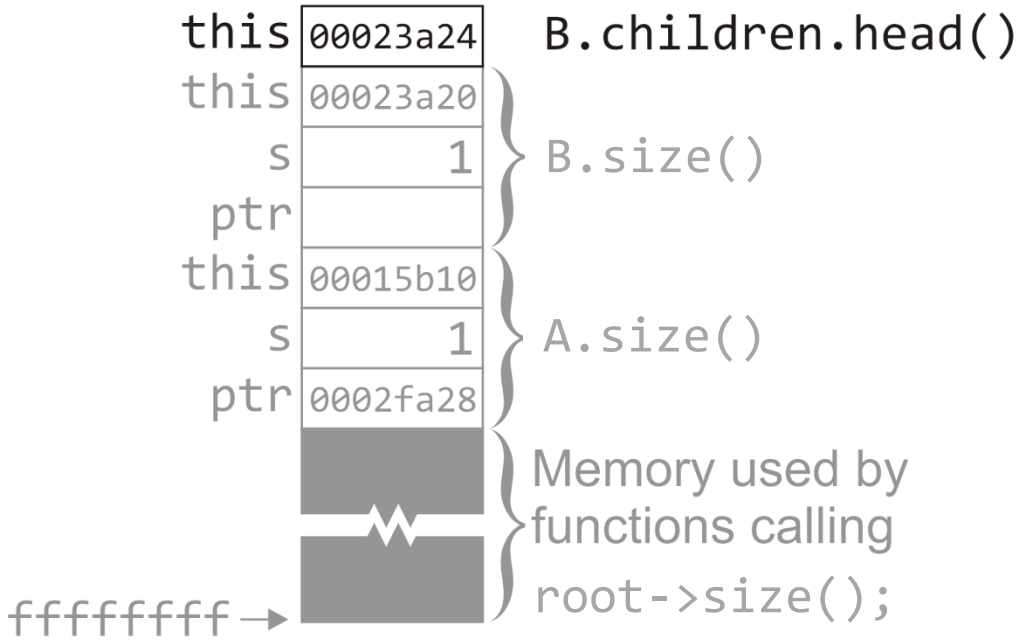
Example

The call to `children.head()` will return the `list_head` of this singly linked list



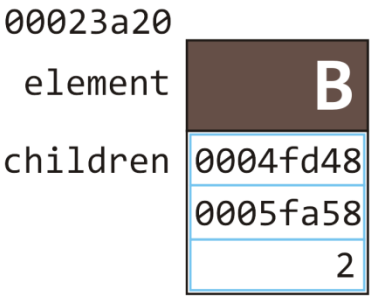
The return value is `0004fd48` and this will be placed on top of the stack

```
Simple_node *Single_list::head() const {
    return list_head;
}
```

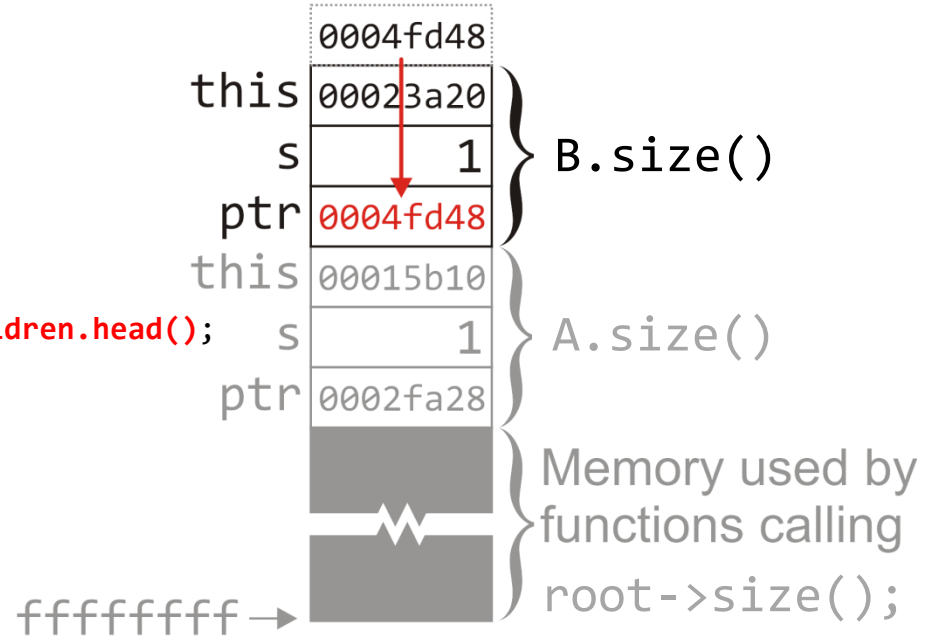


Example

The return value is assigned to the `ptr` associated with this function call



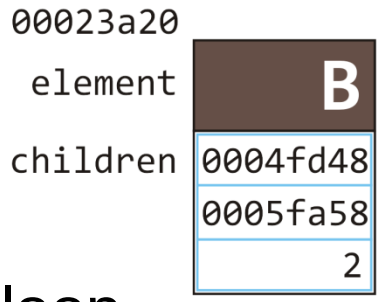

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



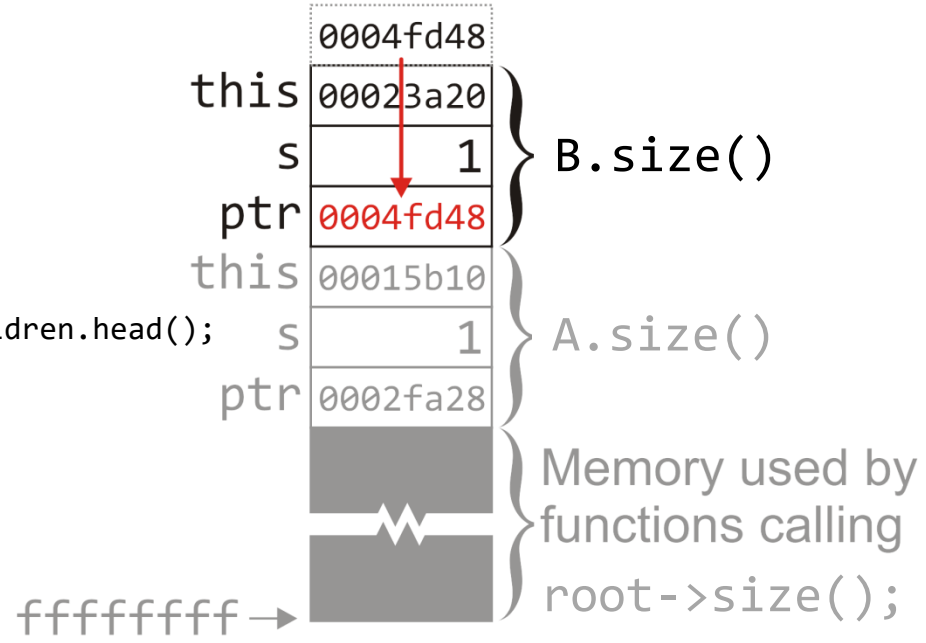
Example

The return value is assigned to the `ptr` associated with this function call

- We check that **`ptr != 0`** and proceed to the body of the loop




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

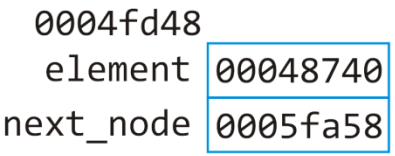


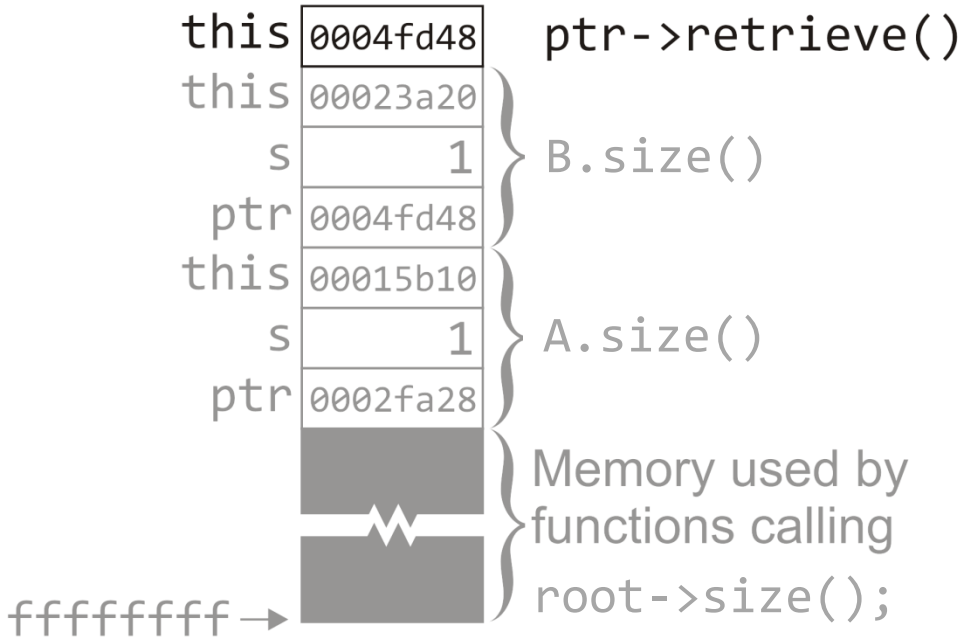
Example

The call to `value()` returns a pointer to the first child of node B

- `children` is a linked list of pointers

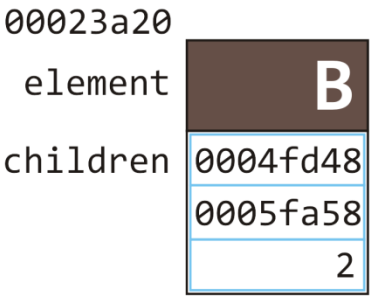
```
Simple_tree *Single_node::value() const {
   return node_value;
}
```



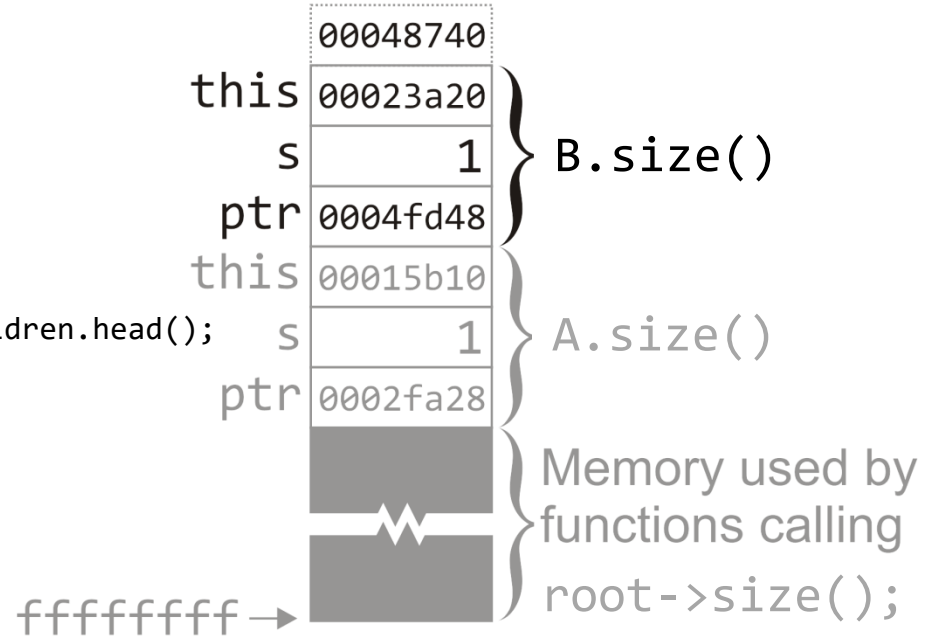


Example

The return value to `value()` is again, left on top of the stack and we will recursively call size on the node at that address



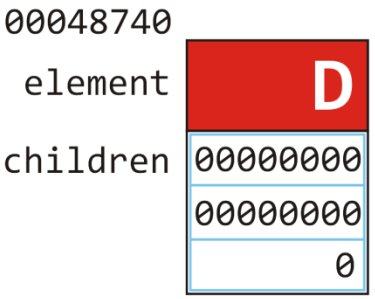

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



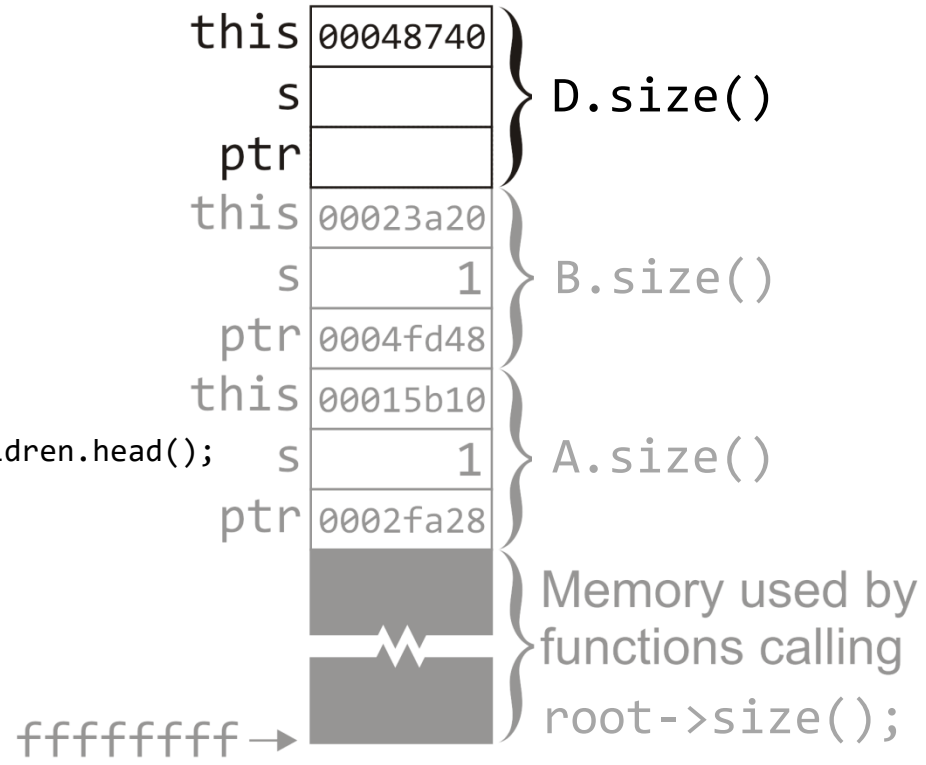
Example

The first child of node B is node D

- As before, we set up three entries on the stack for local variables

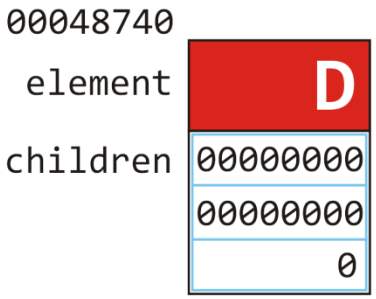


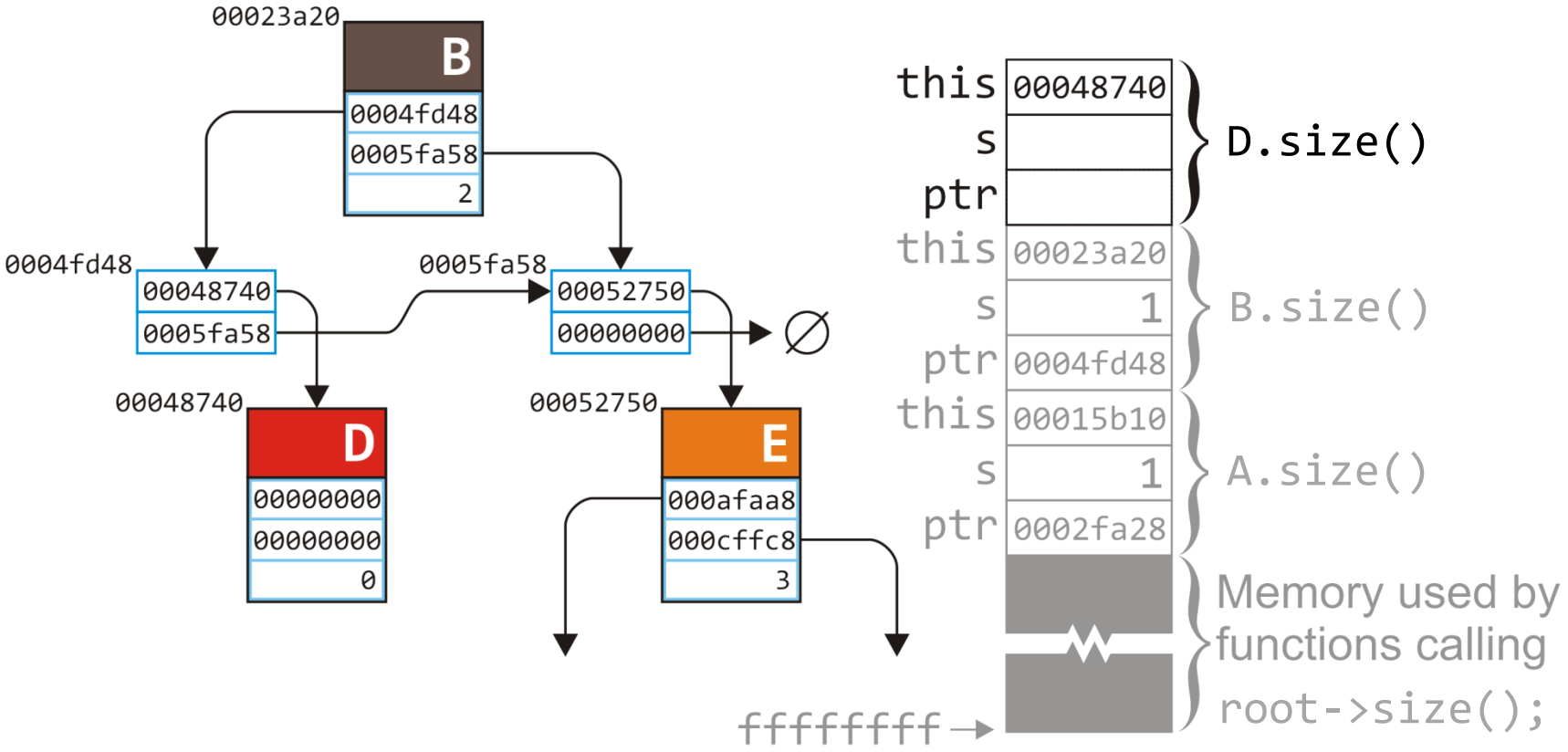

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

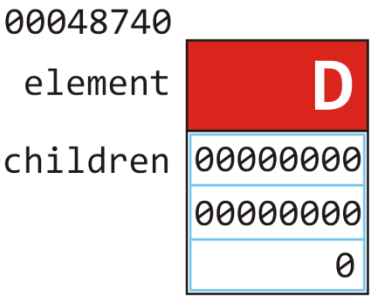
Looking at where we are in the tree



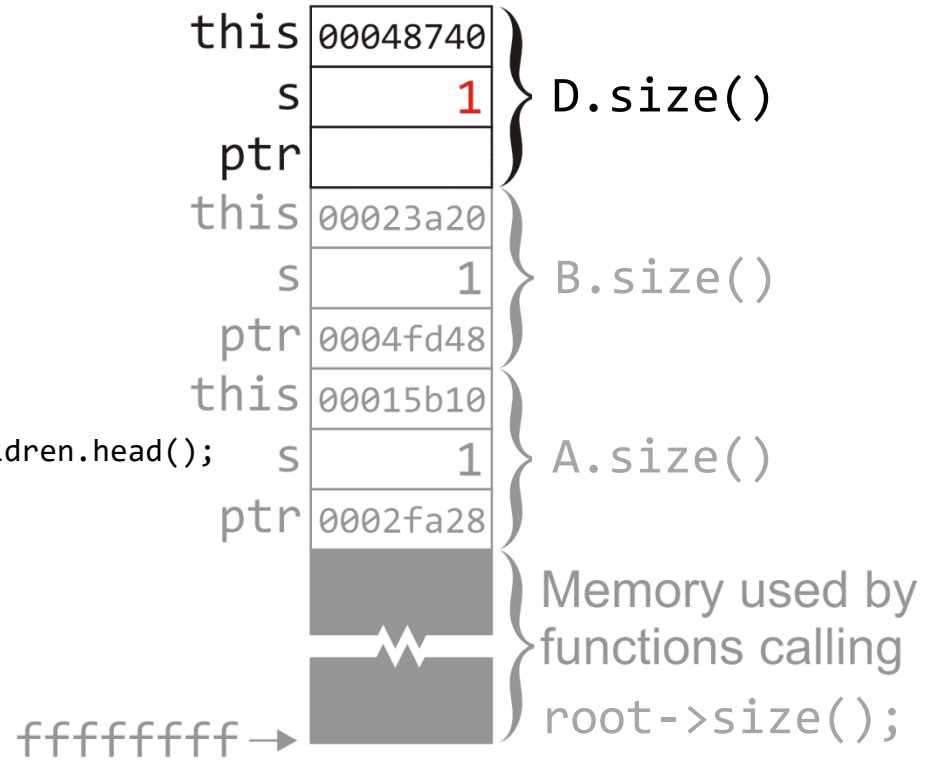


Example

We initialize the local variable `s` for this function call




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

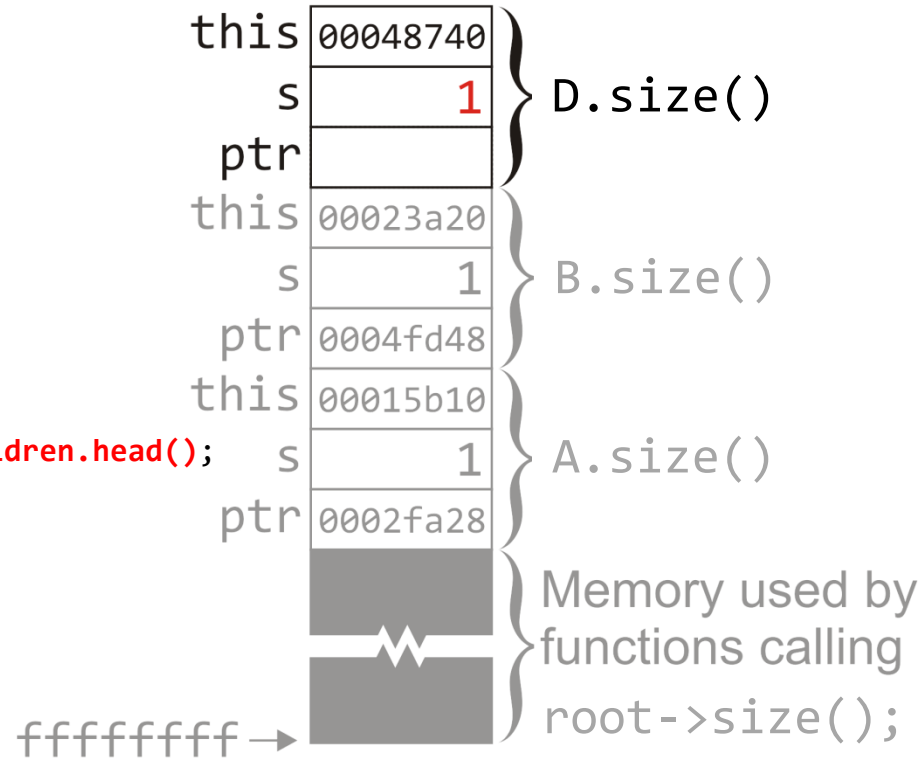


Example

We initialize the local variable `s` for this function call

Again, we must also access the head of this linked list

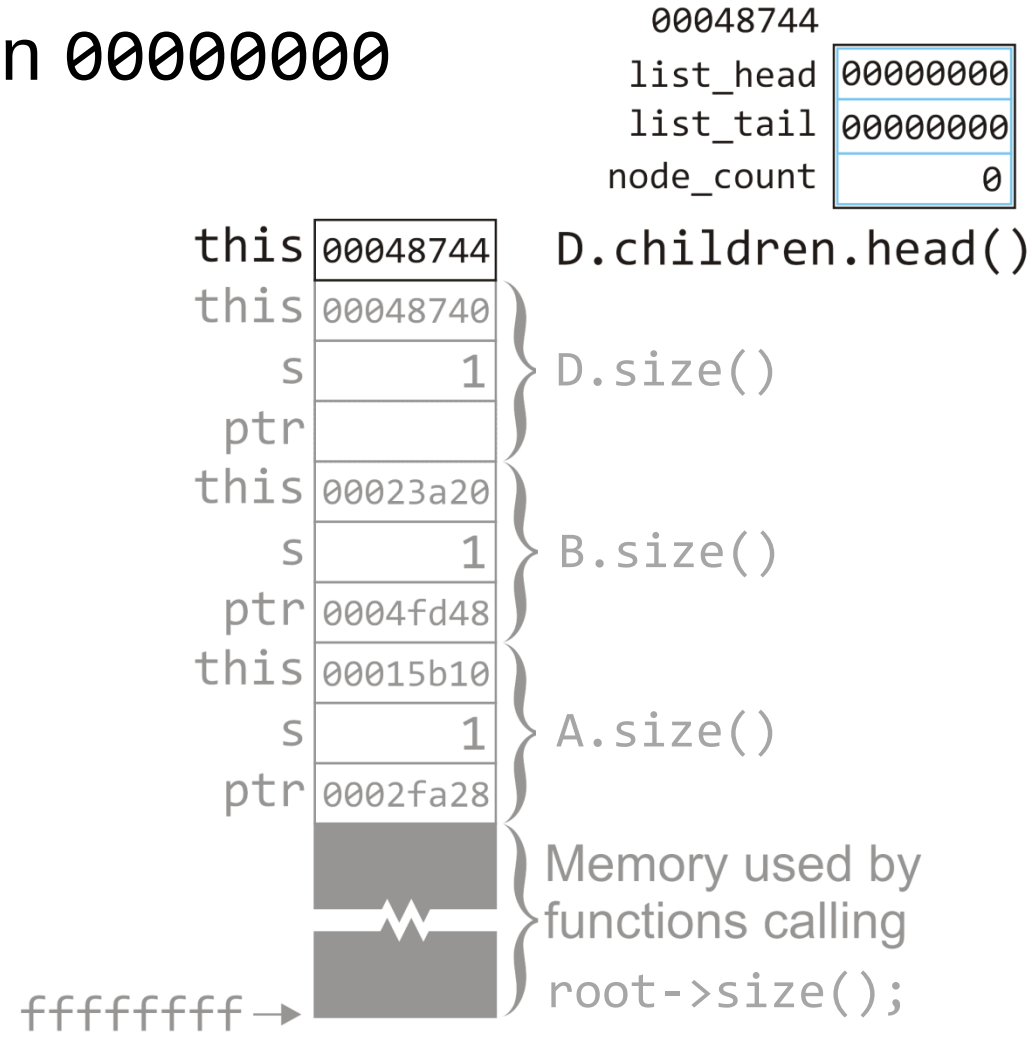
```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

With this call, we now return `00000000`

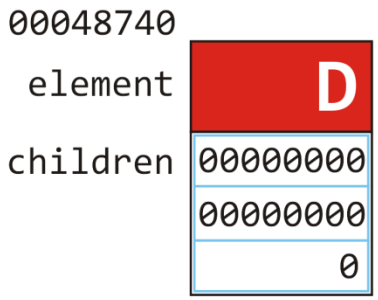
```
Simple_node *Single_list::head() const {
    return list_head;
}
```



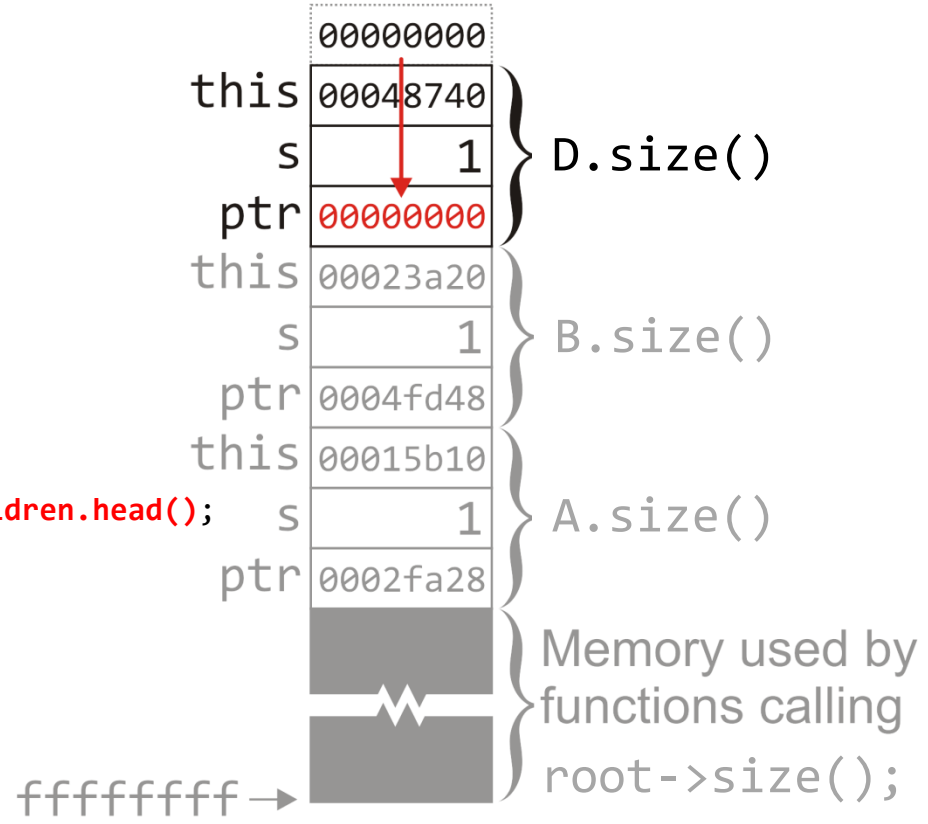
Example

The return value is copied to the top of the stack and we return to `size()` called on node D

- The return value is copied to the location for `ptr`



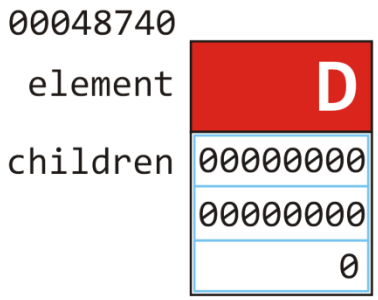

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

The return value is copied to the top of the stack and we return to `size()` called on node D

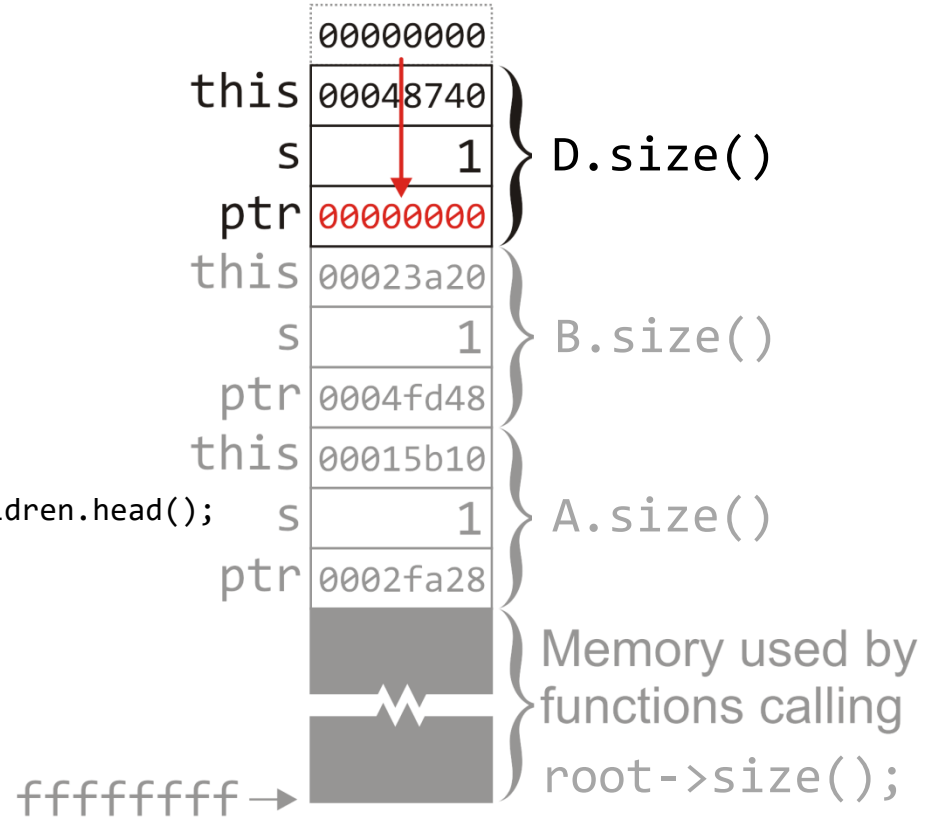
- The return value is copied to the location for `ptr`



In this case, `ptr != 0` returns `false`

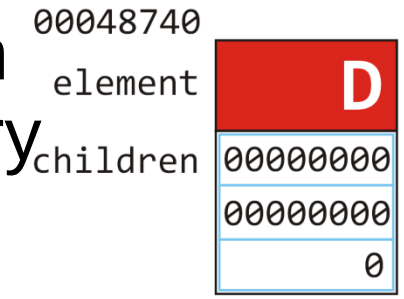
- We jump to the return statement

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

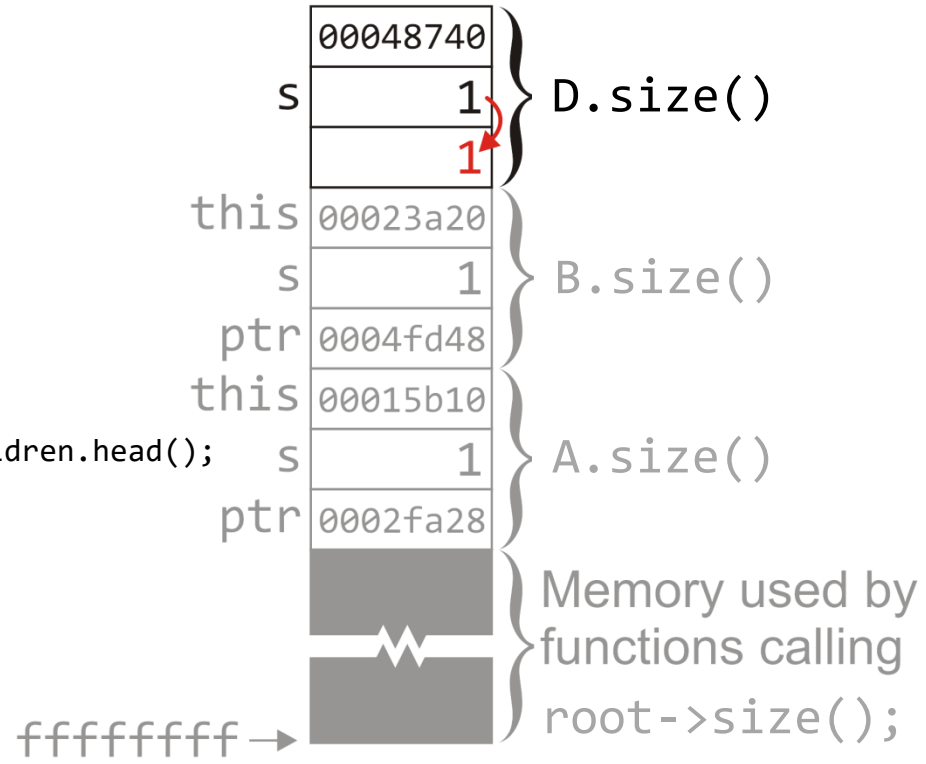


Example

Because we are returning, we must copy the return value to the location immediately above the memory allocated for calling function




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

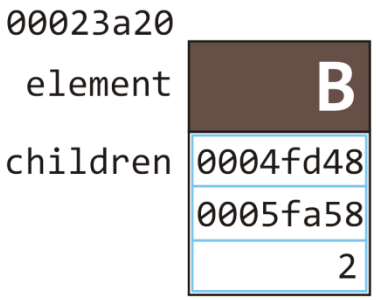


Example

We are just finished the call to

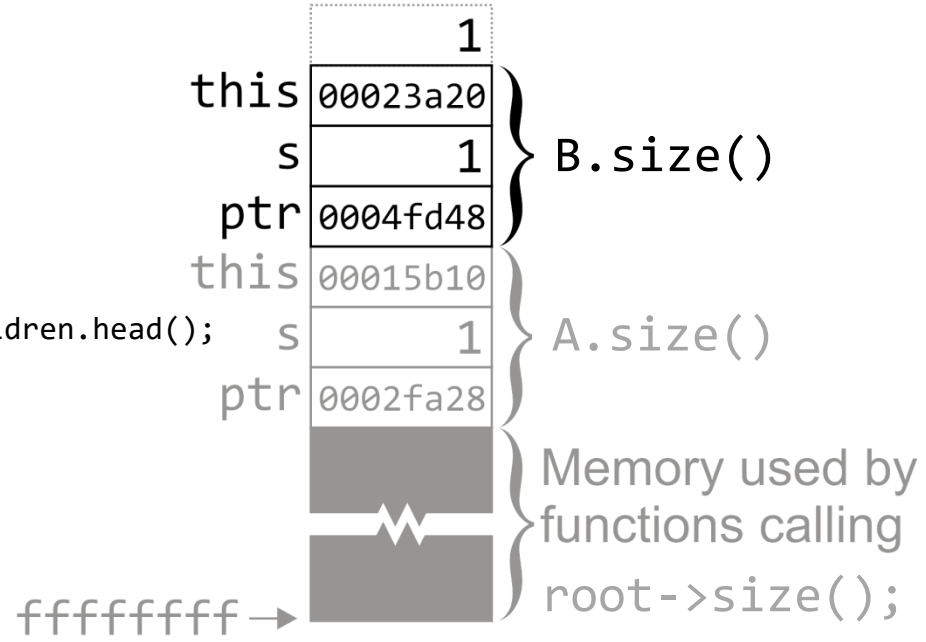
- `ptr->value()->size()`

where `ptr` had the address of node D



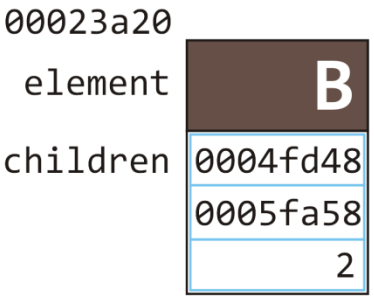
We must now deal with this return value

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

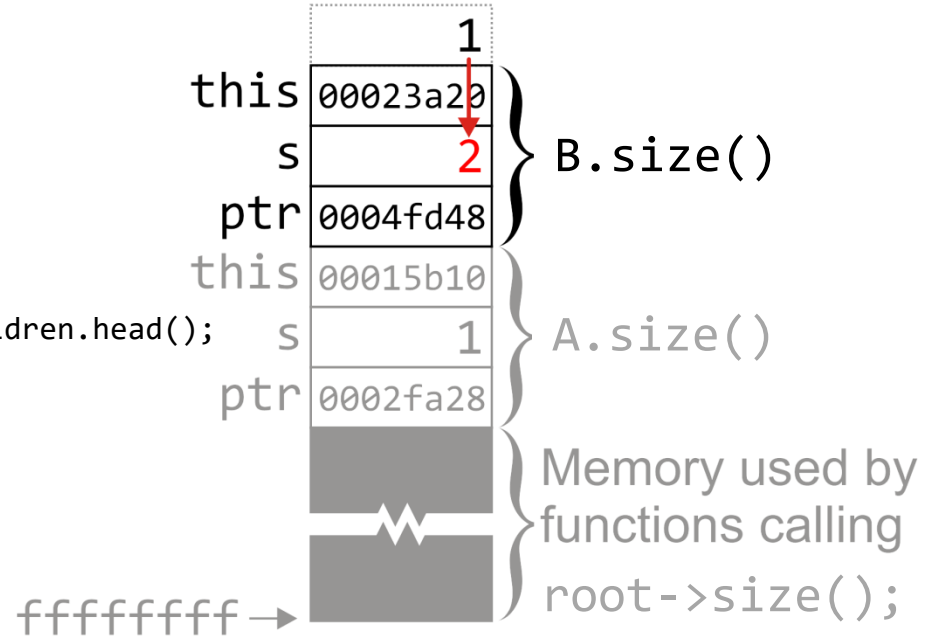


Example

In this case, we add it to the local variable **s**

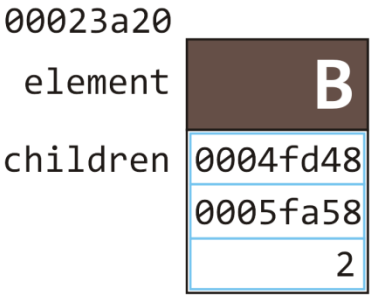



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

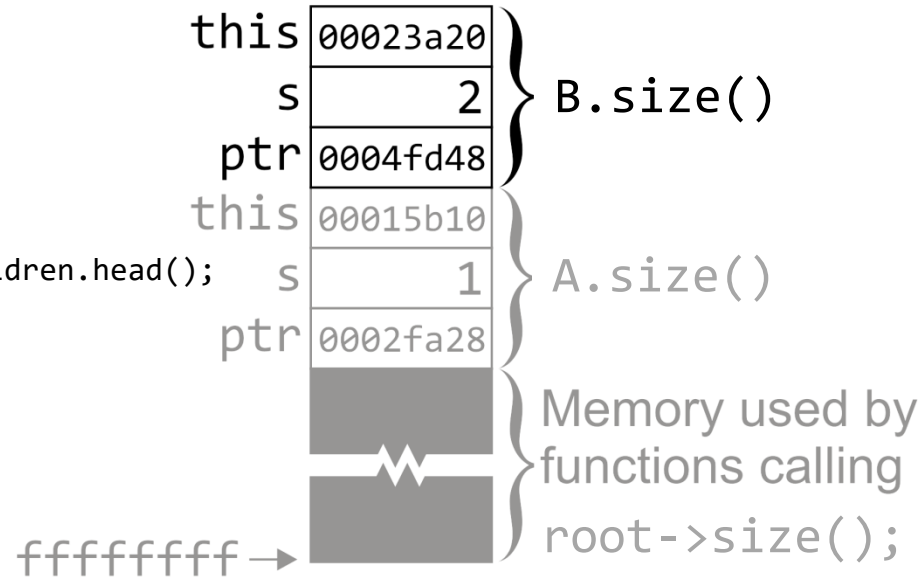


Example

At this point, we have finished executing the body of this loop—we next proceed to the *increment statement* : `ptr = ptr->next()`

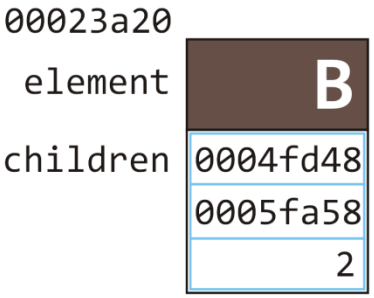


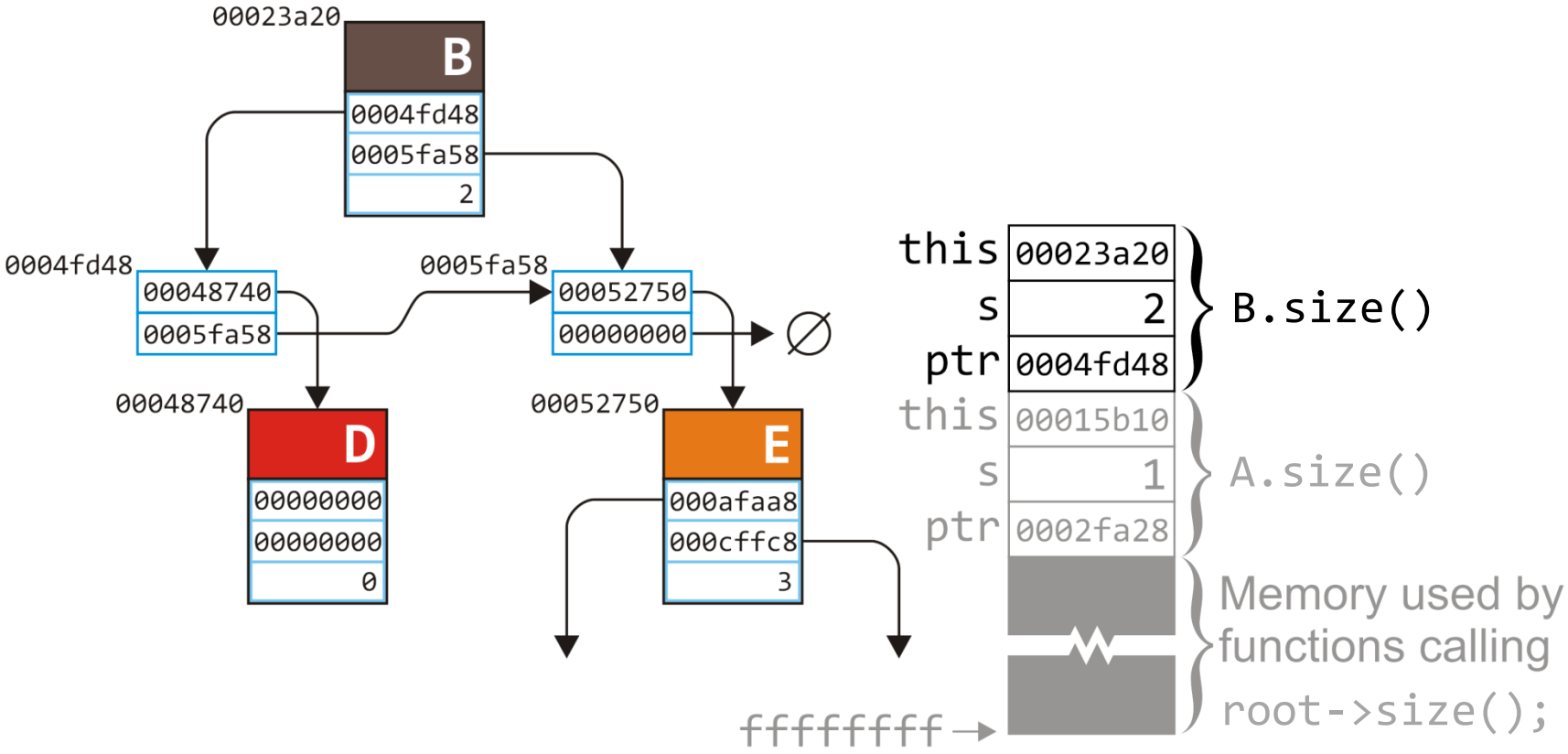

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

At this point, we have finished executing the body of this loop—we next proceed to the *increment statement* : `ptr = ptr->next()`





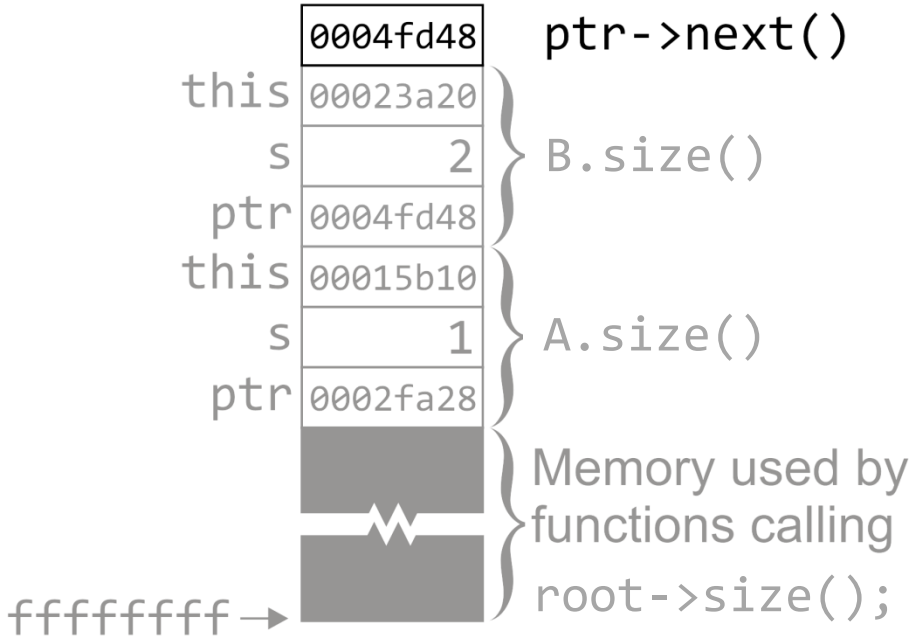
Example

0004fd48	
element	00048740
next_node	0005fa58

Again, we start with a function call on the node stored at `0004fd48`—this requires information placed onto the stack

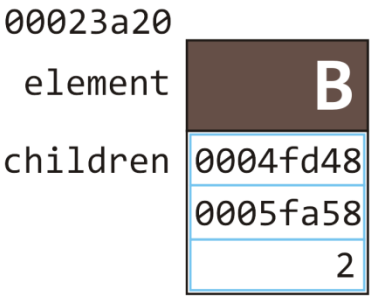
- The address `0005fa58` is the return value

```
Simple_tree *Single_node::next() const {
   return next_node;
}
```

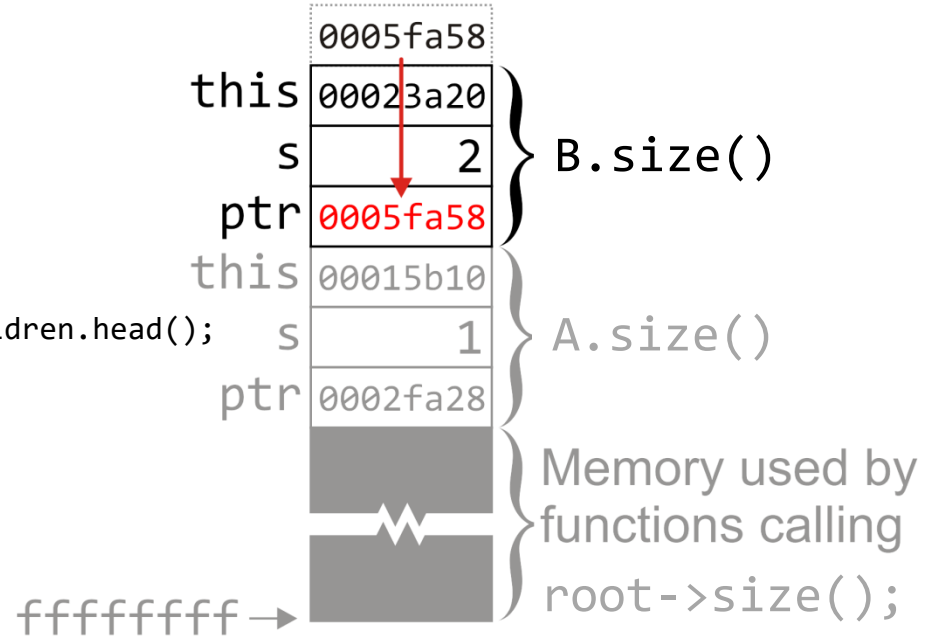


Example

The return value is copied to the memory location for `ptr`



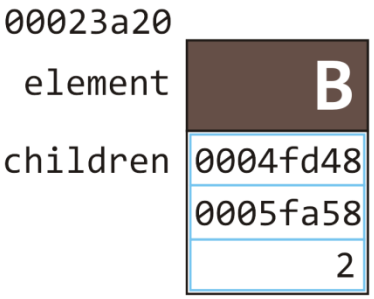

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



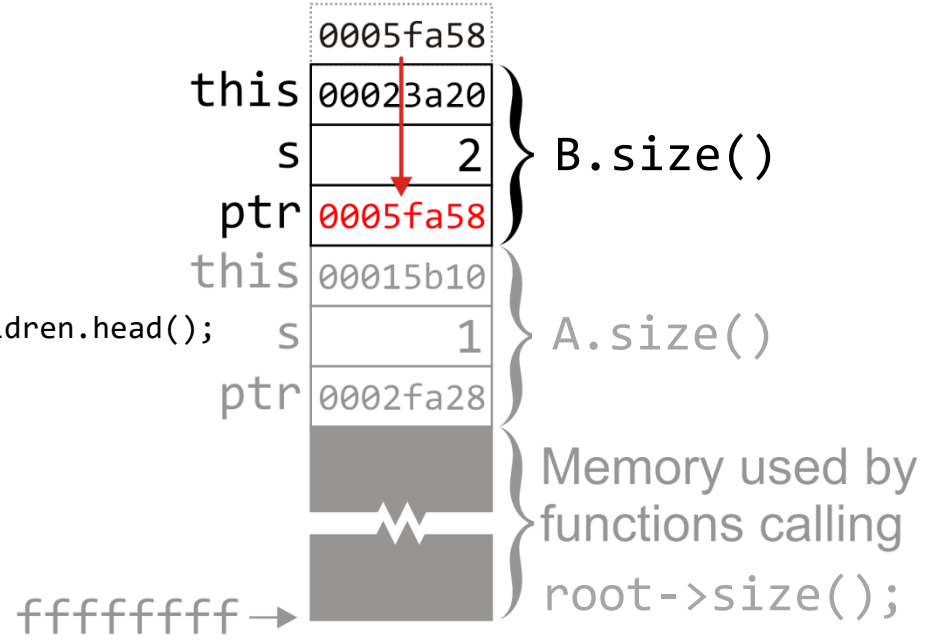
Example

The return value is copied to the memory location for `ptr`

- The test `ptr != 0` evaluates to `true`, so we continue into the body



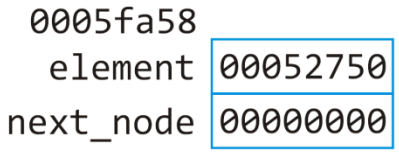

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



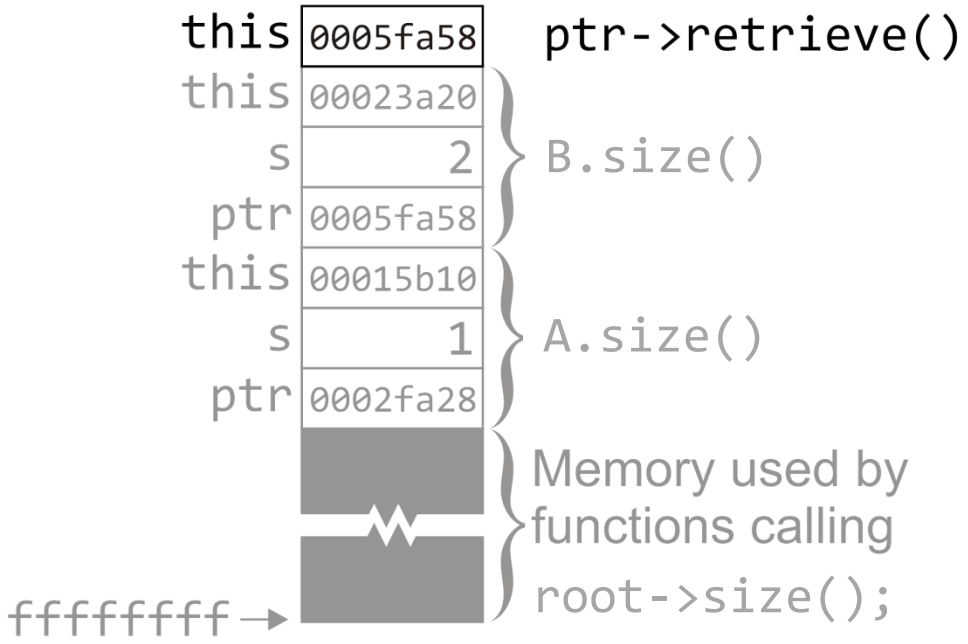
Example

We call `ptr->value()` on this node in the body of the for loop

- This will return the address `00052750`



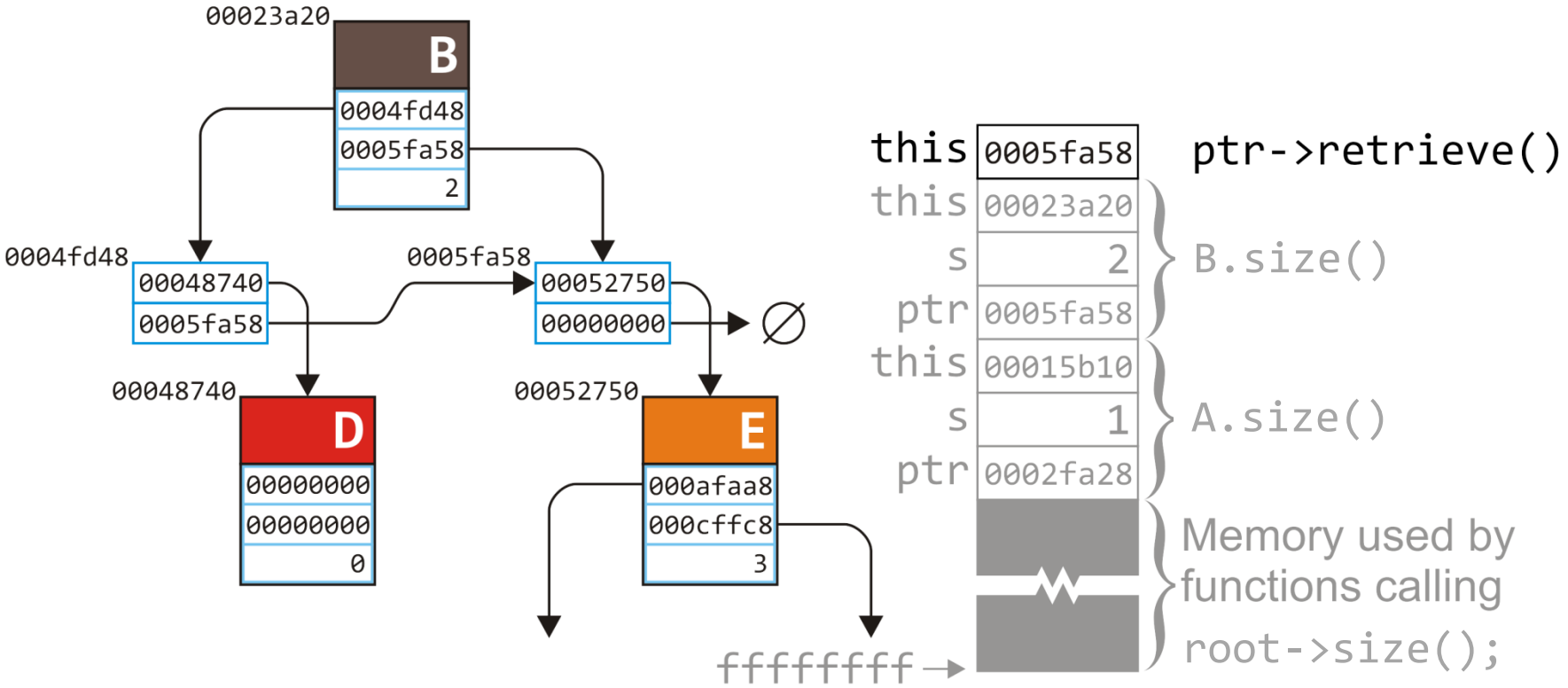

```
Simple_tree *Single_node::value() const {
   return node_value;
}
```



Example

We call `ptr->value()` on this node in the body of the for loop

- This will return the address `00052750`
- We are currently at the second node in the linked list



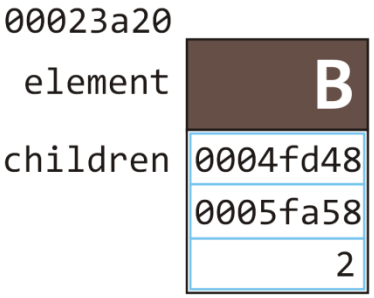
0005fa58

element	00052750
next_node	00000000

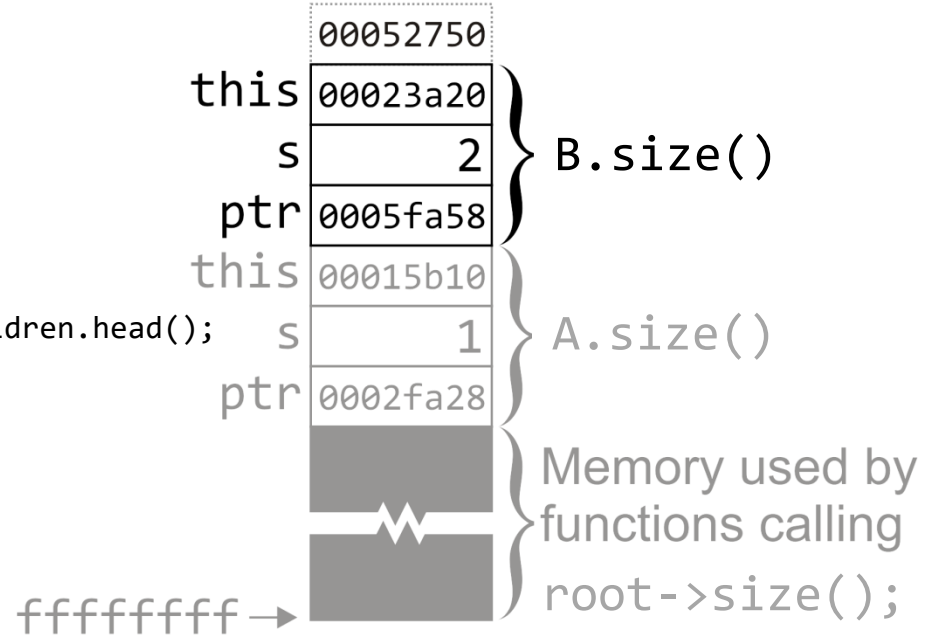
Example

This value is placed on top of the stack

- We will proceed to call `size()` on the tree node at this address



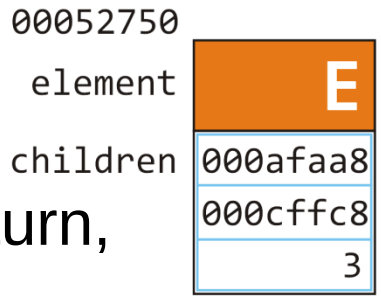

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

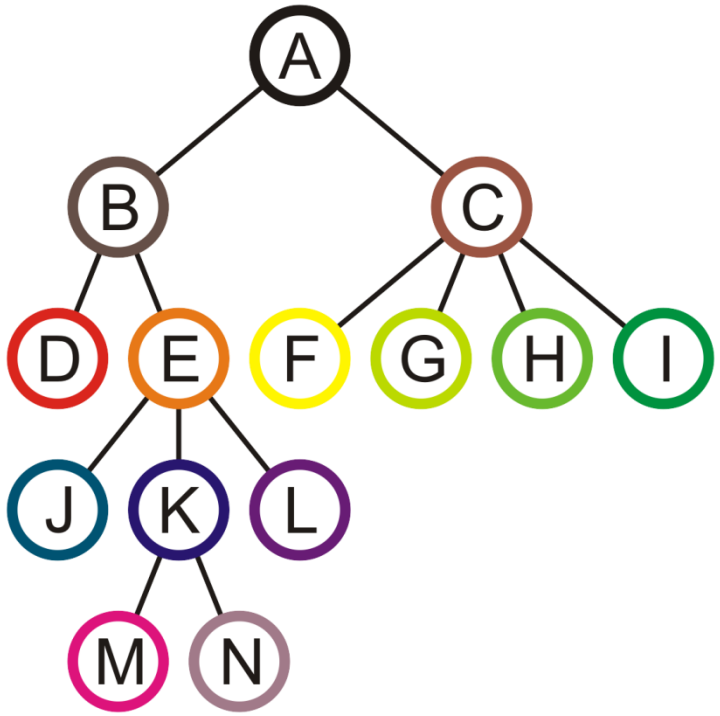


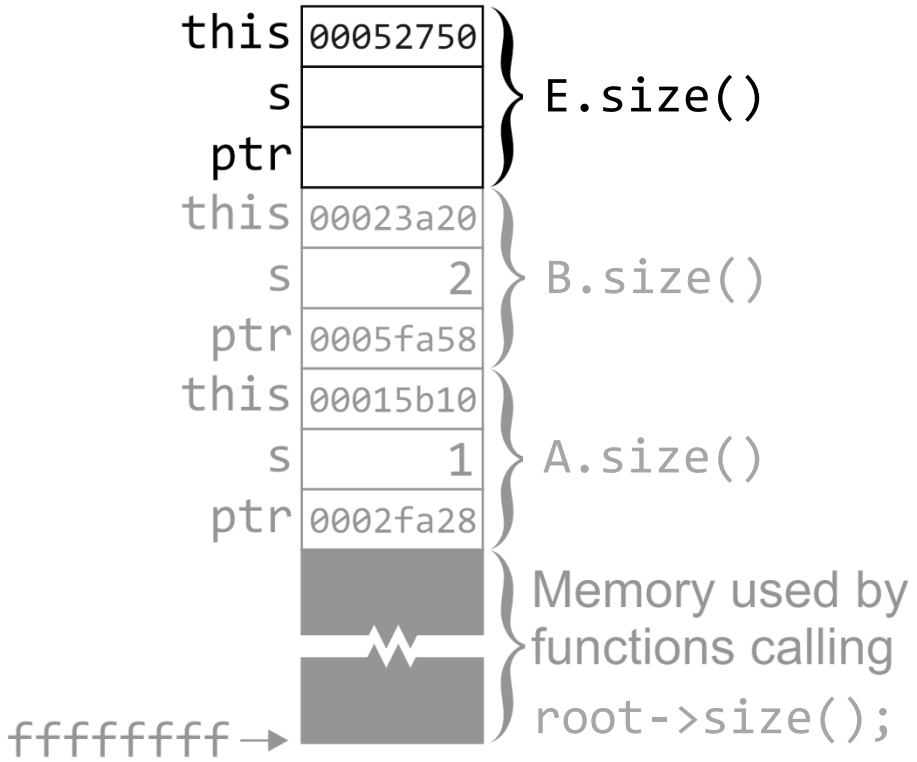
Example

As before, we allocate memory for a call to `size()` on a tree node

- We will repeat the process on tree node E and, in turn, on all its descendants



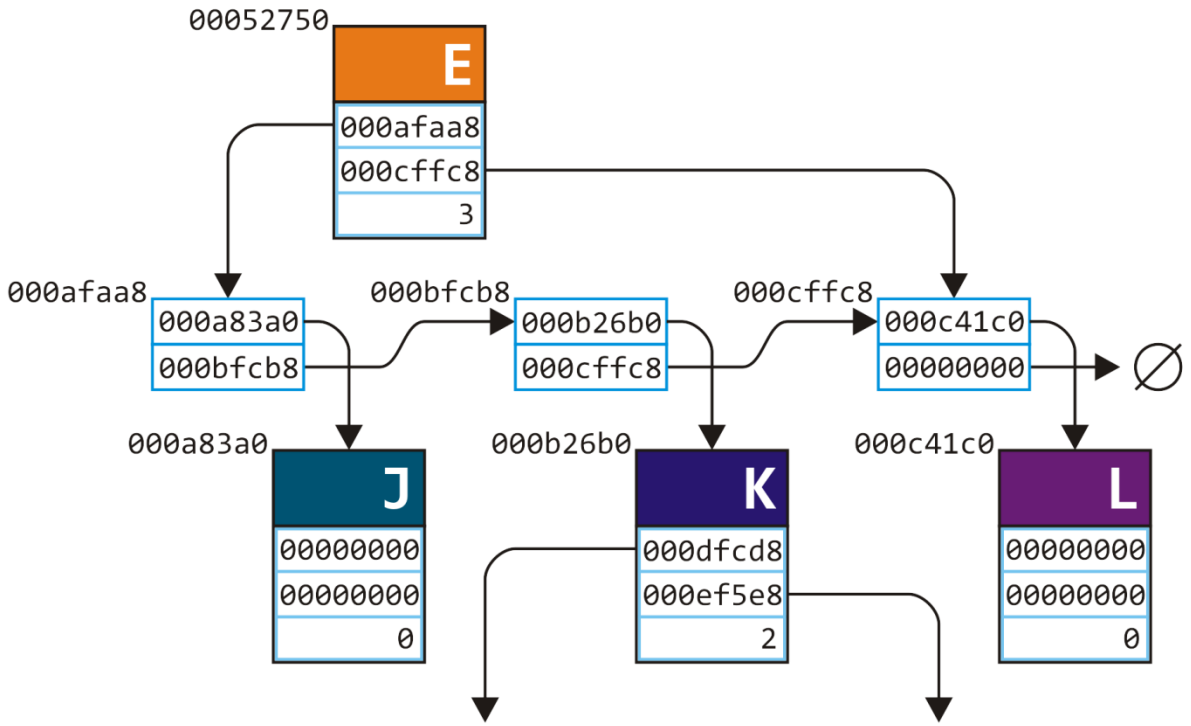




Example

This is the node we are now recursively calling `size()` on

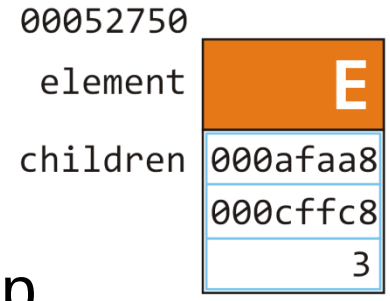
- Node K has two children, each of which are leaf nodes



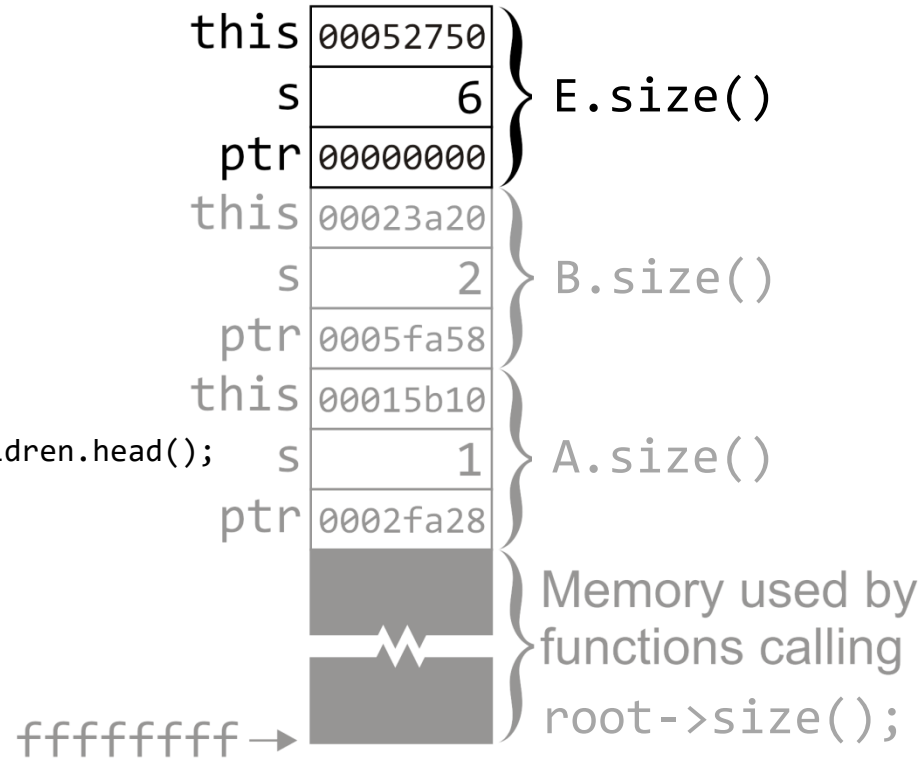
Example

It should be obvious that, at the end, s will be assigned `6` and `ptr` is assigned `0`

- We are at the end of this tree node's linked list
- When `ptr != 0` returns `false`, we end the for loop

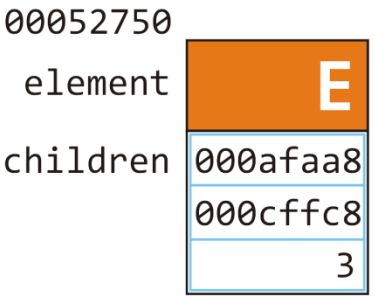



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

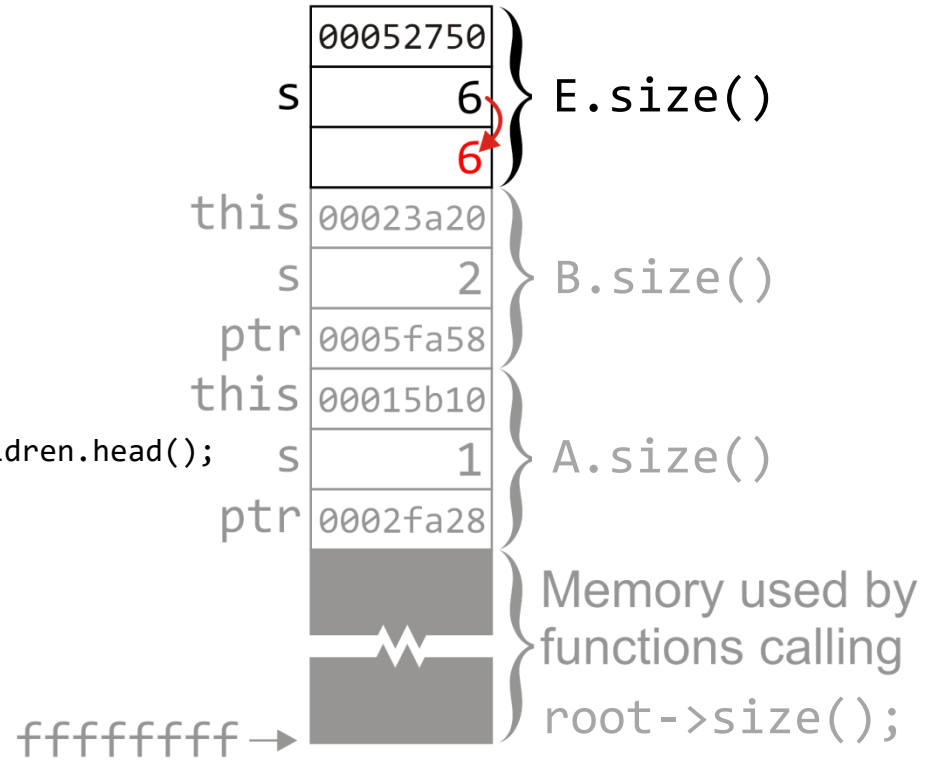


Example

In preparation to return, we copy the value of s to the appropriate location

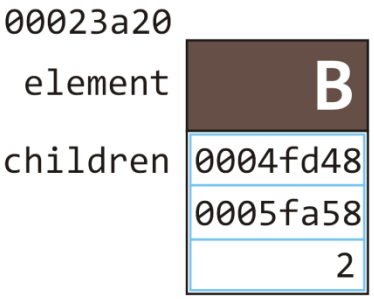



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

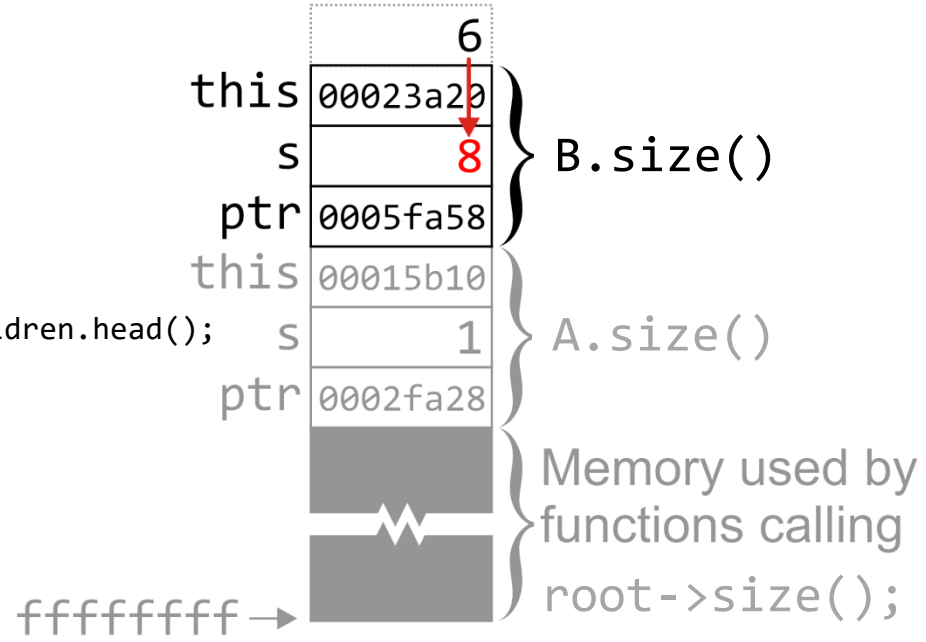


Example

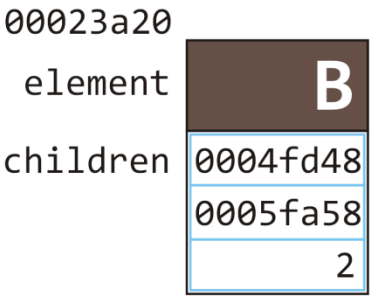
We are now back in tree node B and we add the returned value onto the local variable `s` for the current function call




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



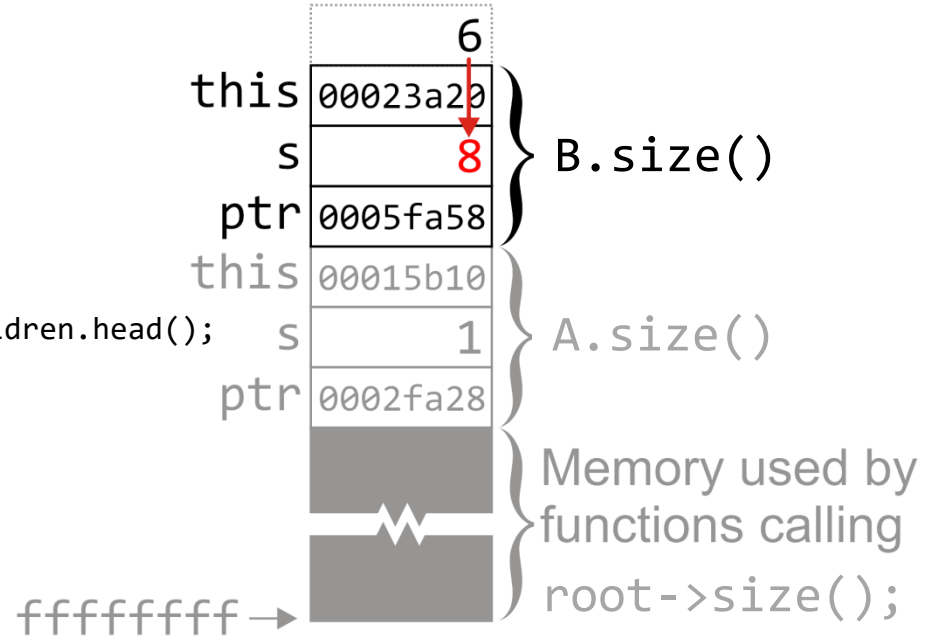
Example



We are now back in tree node B and we add the returned value onto the local variable `s` for the current function call

- We are finished executing the body of the loop, so we proceed to `ptr = ptr->next()`

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



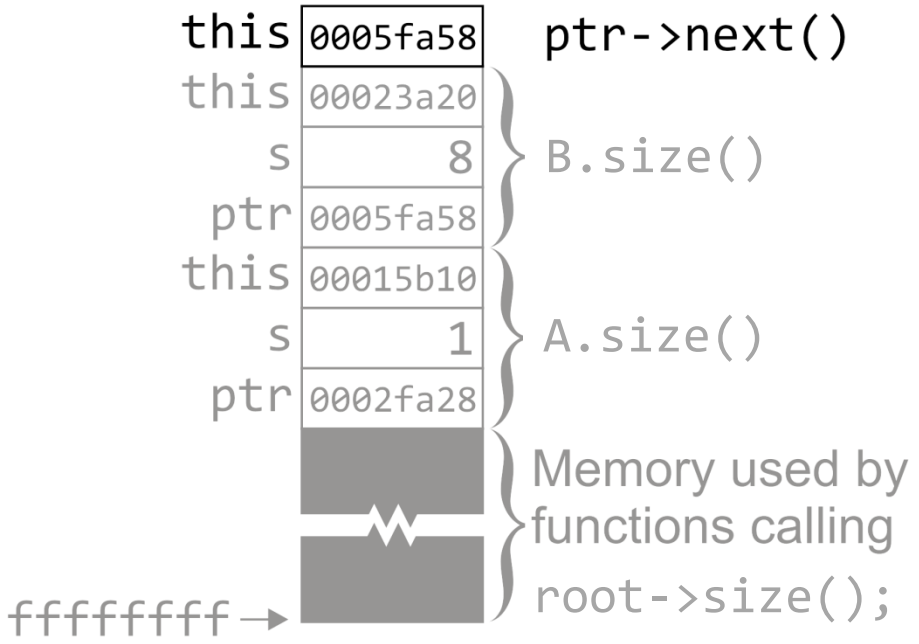
Example

We call `next()` on the node at the address `ptr`

- In this case, it is `00000000`

0005fa58	
element	00052750
next_node	00000000

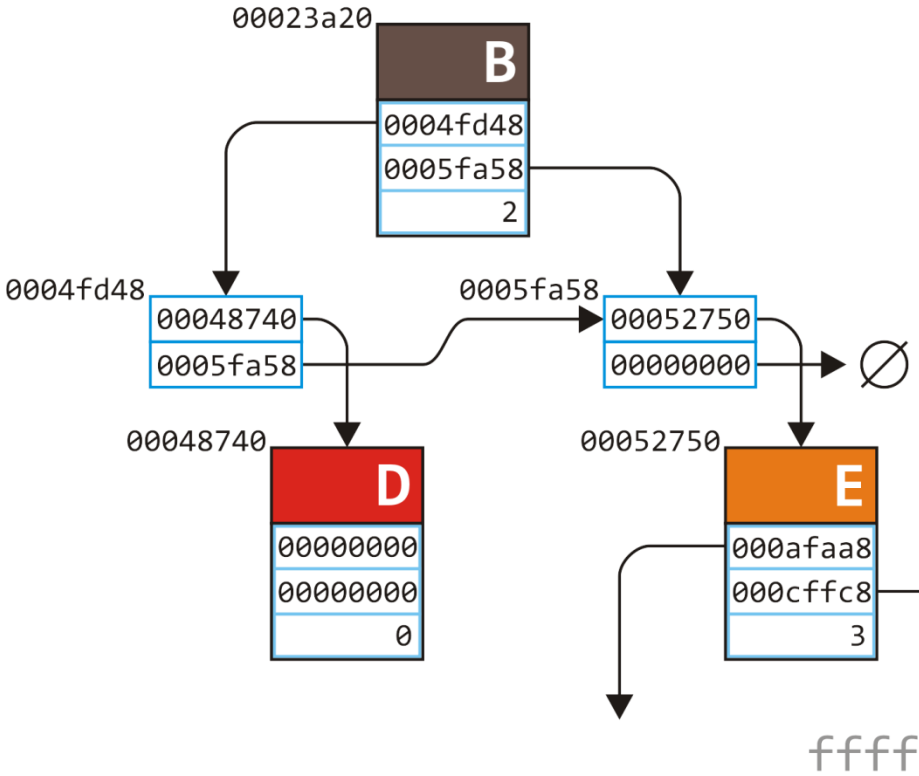
```
Simple_tree *Single_node::next() const {
   return next_node;
}
```

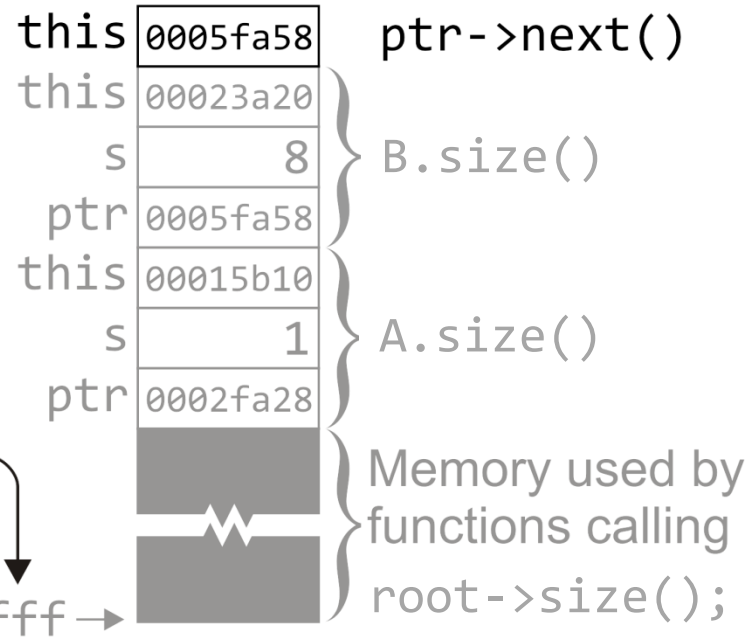


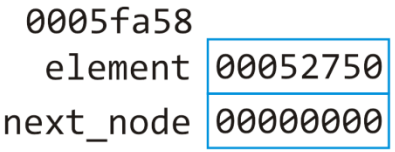
Example

We call `next()` on the node at the address `ptr`

- In this case, it is `00000000`
- We are at the end of this linked list

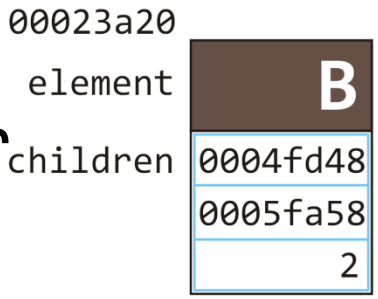




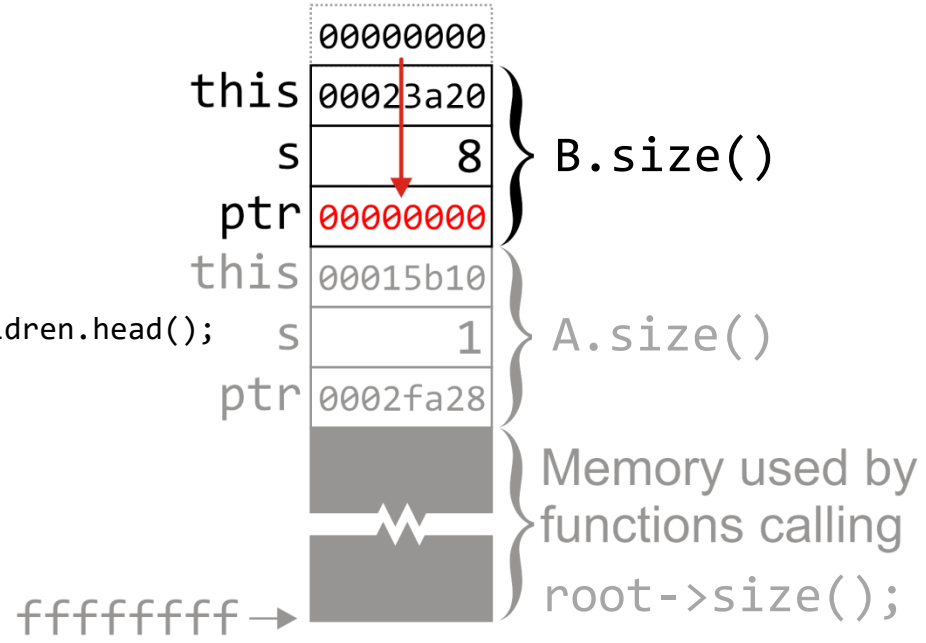


Example

This value is copied to the memory location of `ptr` for the function call on tree node B



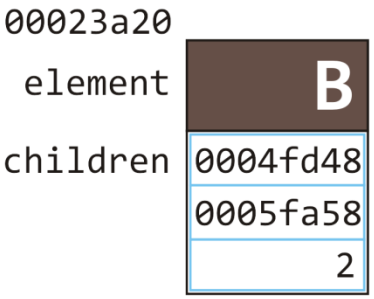

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



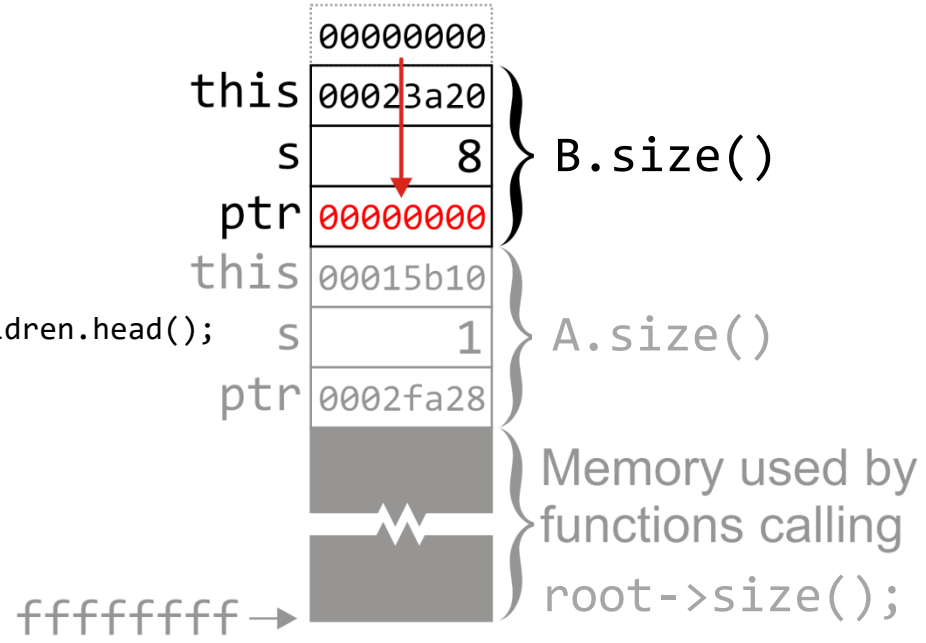
Example

This value is copied to the memory location of `ptr` for the function call on tree node B

- `ptr != 0` evaluates to `false`, so we end the loop

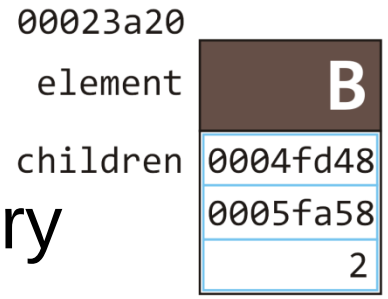



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

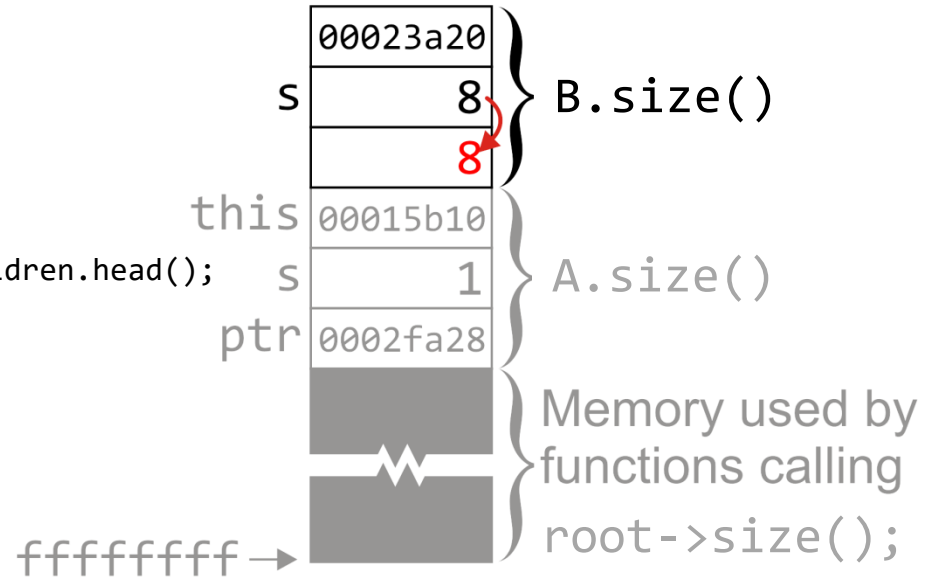


Example

We prepare to return the value of `s` by copying its value to the location immediately above the memory for the previous function call

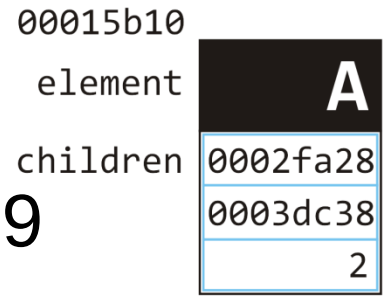



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

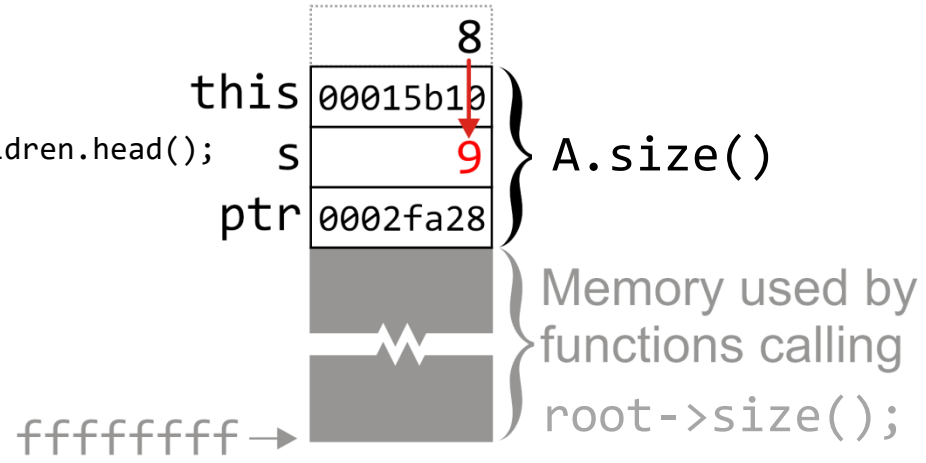


Example

Now that we're back in the function call on tree node A, we add the return value onto s: 1 + 8 = 9

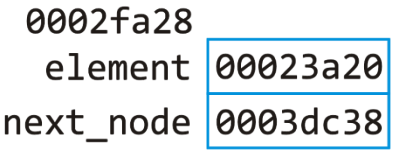



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

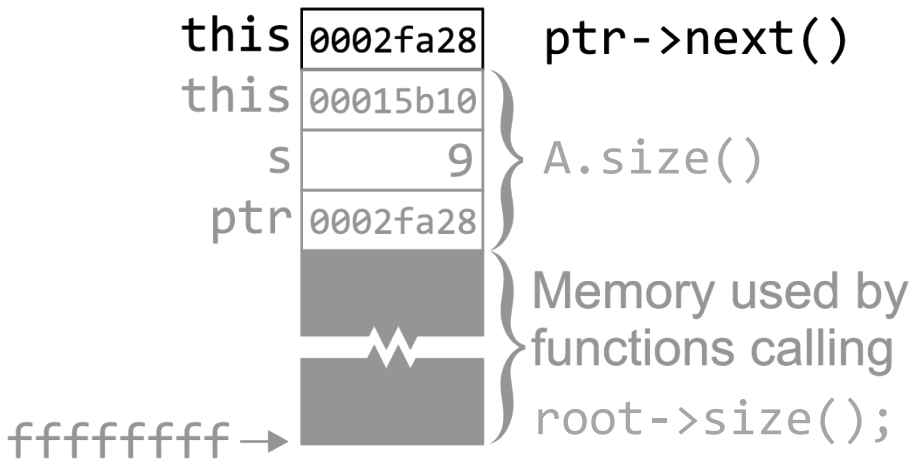


Example

The body of the loop is over; we call `next()` on the node at the address stored in `ptr`

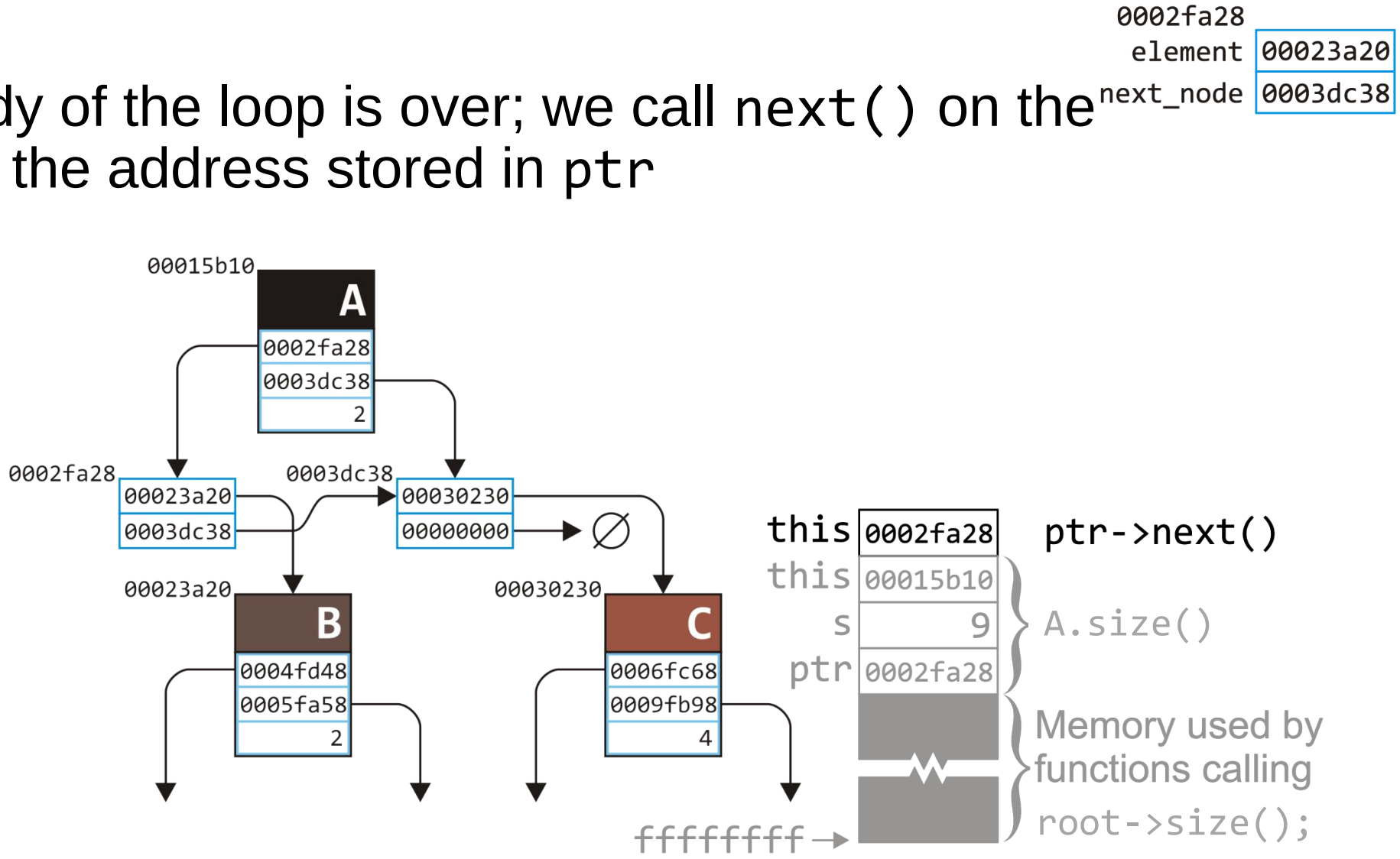



```
Simple_tree *Single_node::next() const {
   return next_node;
}
```



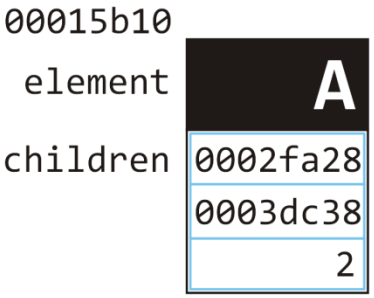
Example

The body of the loop is over; we call `next()` on the node at the address stored in `ptr`

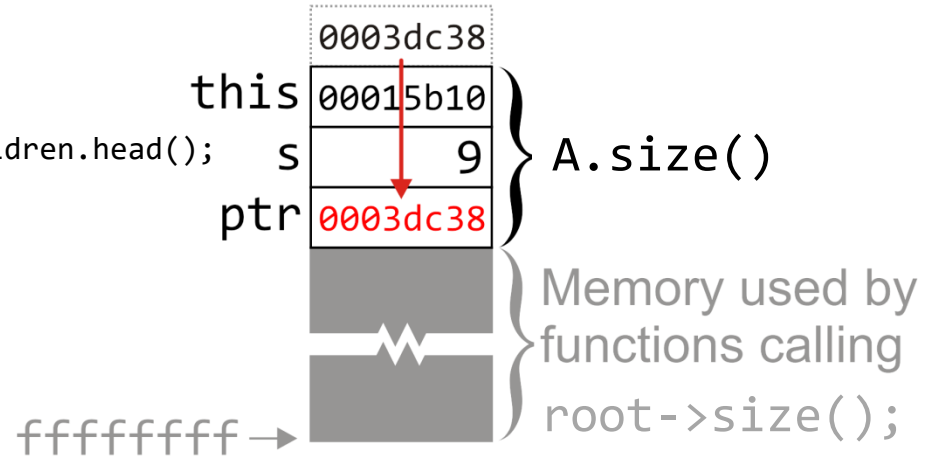


Example

The returned value is assigned to `ptr`



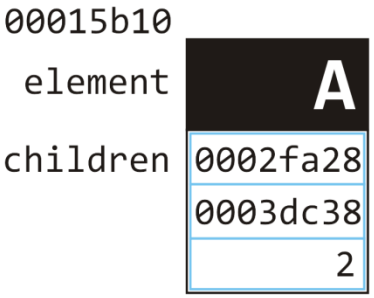

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



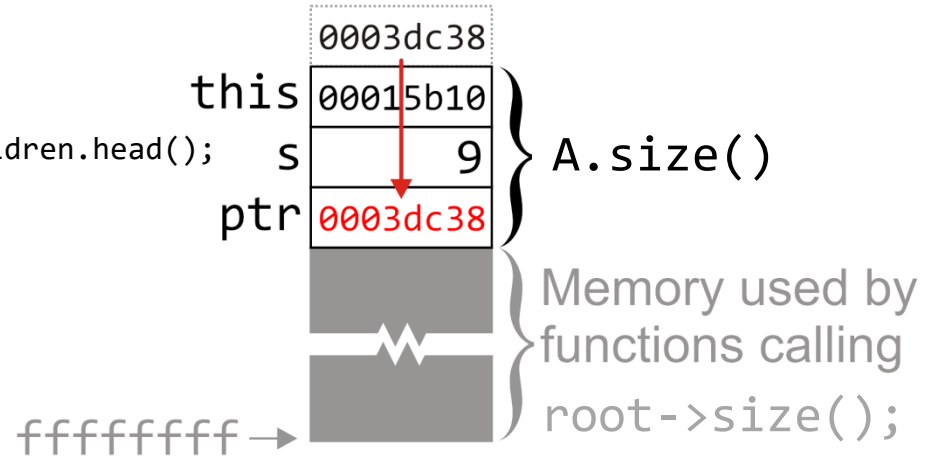
Example

The returned value is assigned to `ptr`

- We note that it is `ptr != 0`



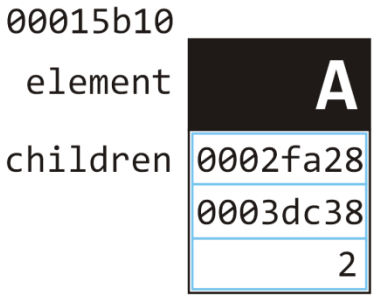

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



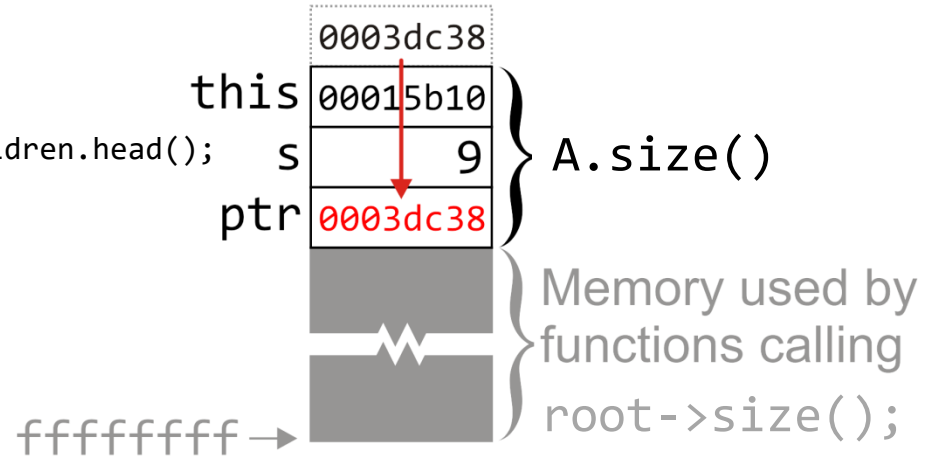
Example

The returned value is assigned to `ptr`

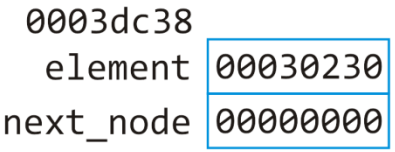
- We note that it is `ptr != 0`
- We call `ptr->value()`




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



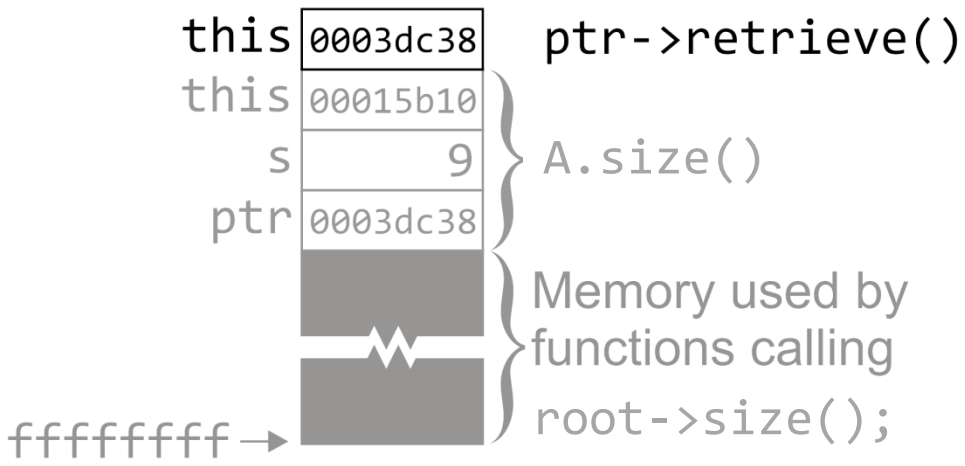
Example



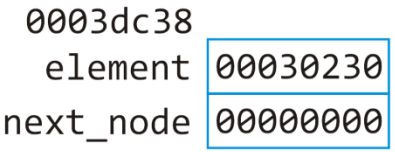
This node stores the address of the tree node at `0030230`

- This is the value returned

```
Simple_tree *Single_node::value() const {
   return node_value;
}
```

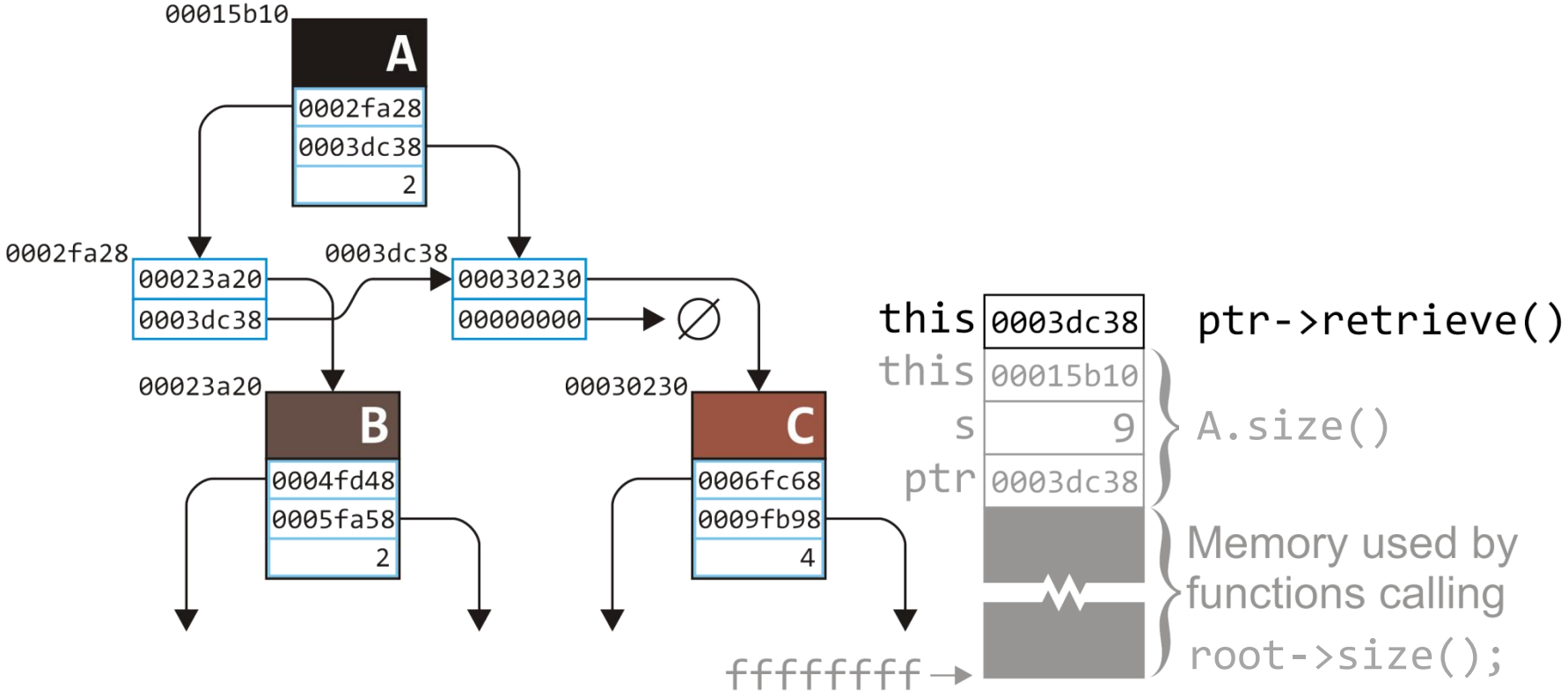


Example



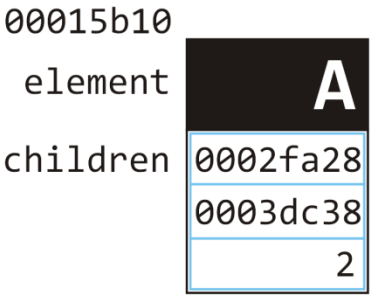
This node stores the address of the tree node at `0030230`

- This is the value returned

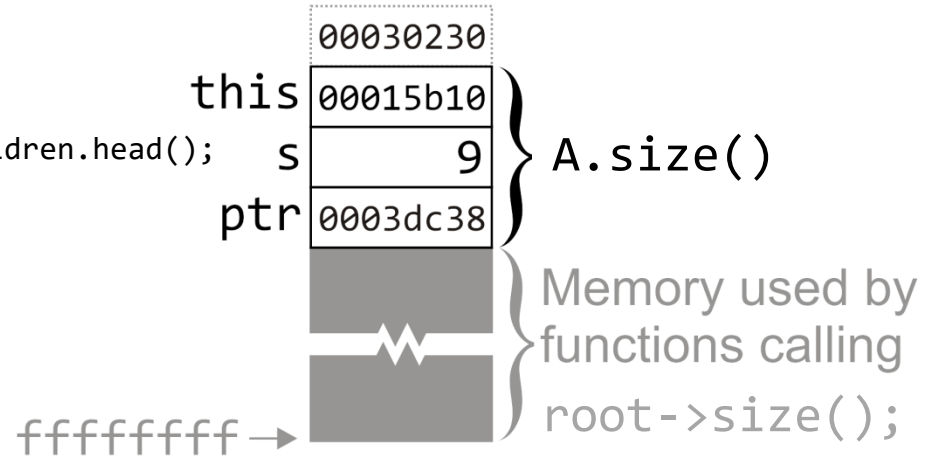


Example

We call `size()` on this tree node



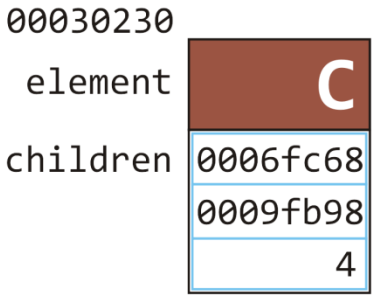

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

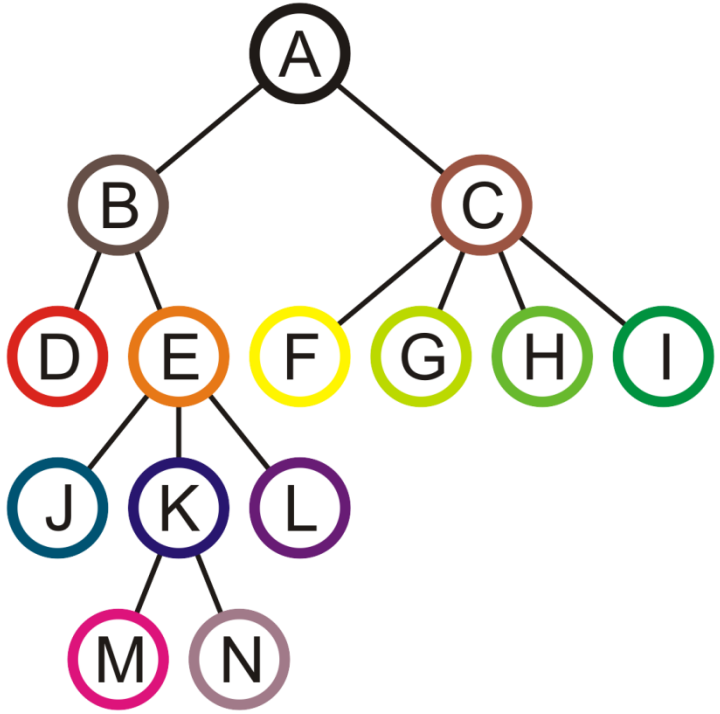


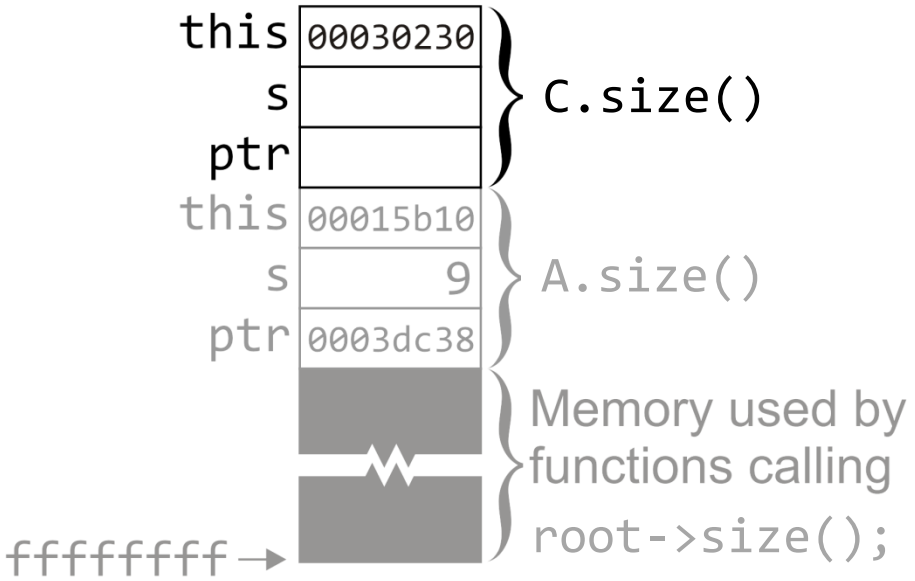
Example

Again, we won't go through the whole process

- After walking through the linked list, we will note that the number of nodes in this tree is 5

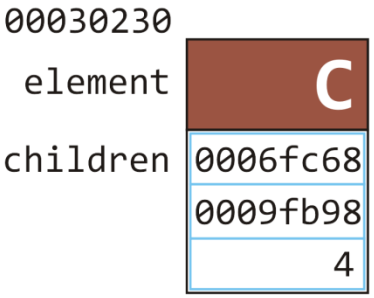




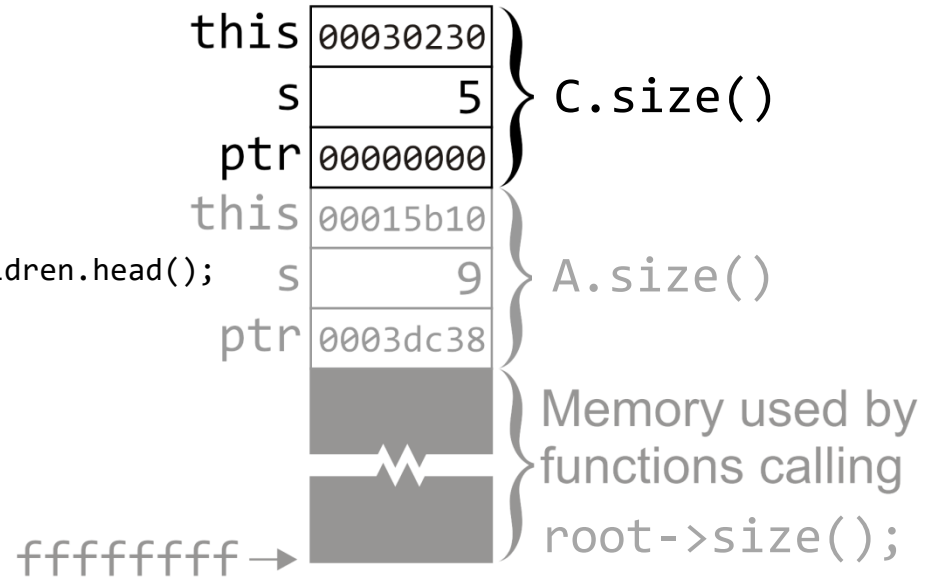


Example

By the end, `s` is assigned `5` and `ptr` is `00000000`

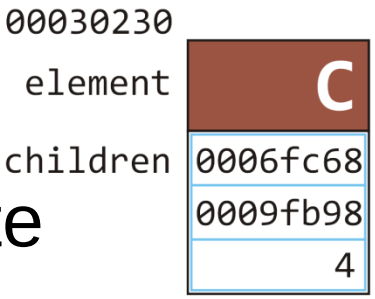



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

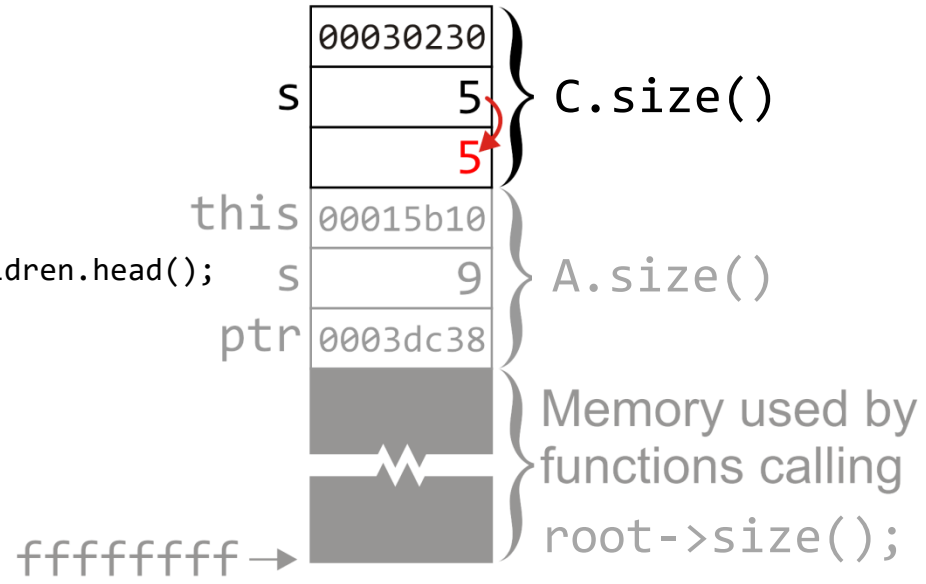


Example

The condition `ptr != 0` is false, so we prepare to return by copying the value of `s` into the appropriate location



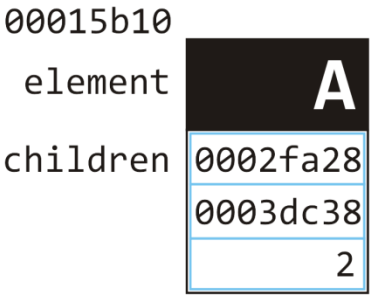

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



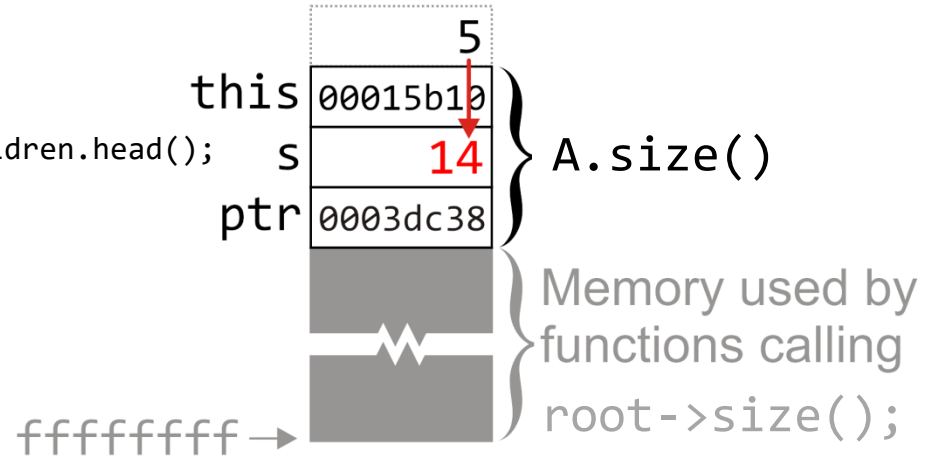
Example

The calling function adds the return value to so

- The new value is 9 + 5 = 14

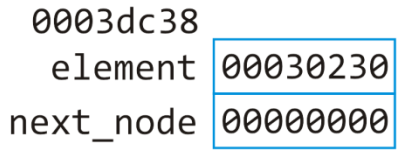



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

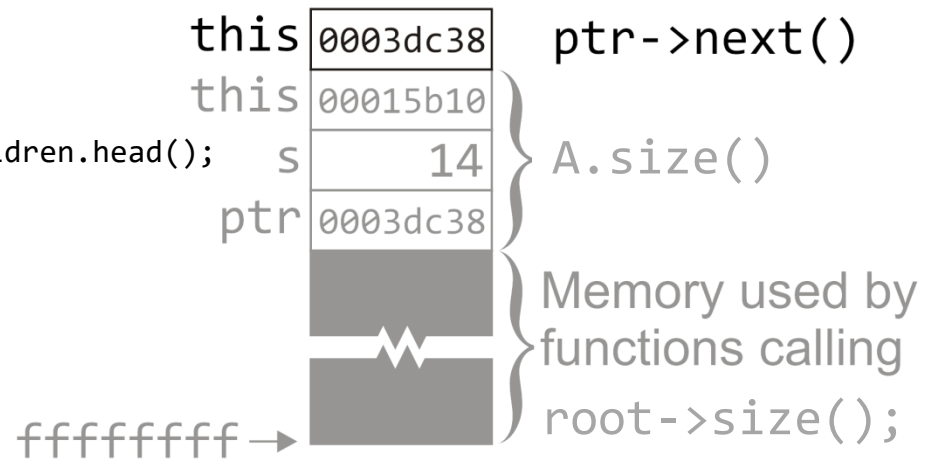


Example

The loop is over, so we call `ptr->next()`

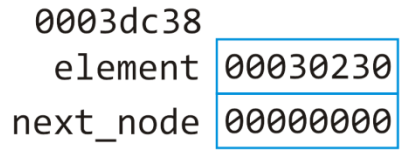


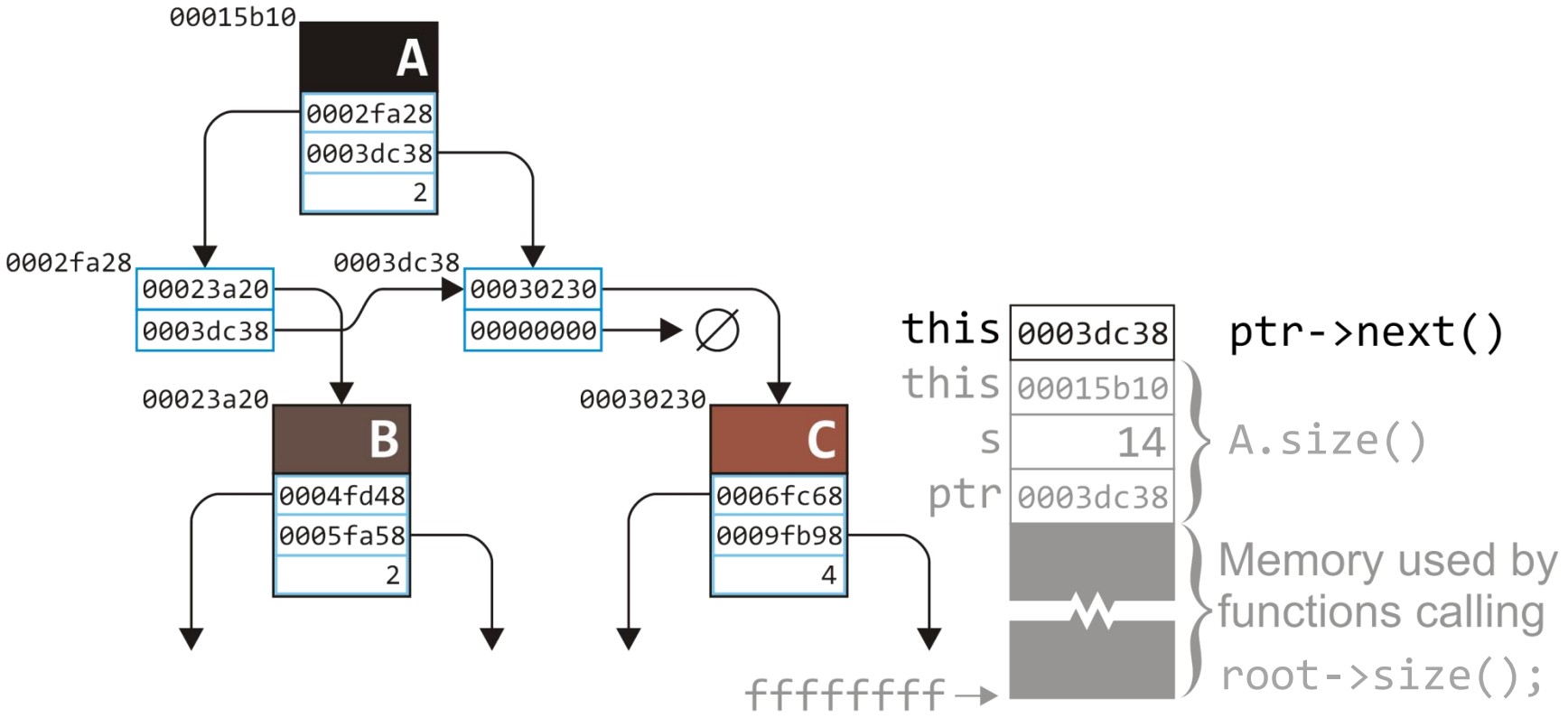

```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

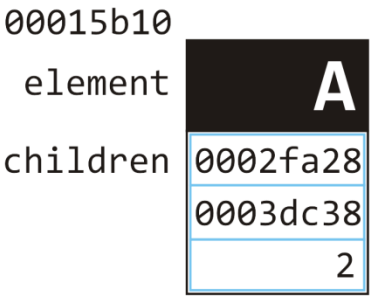
The loop is over, so we call `ptr->next()`



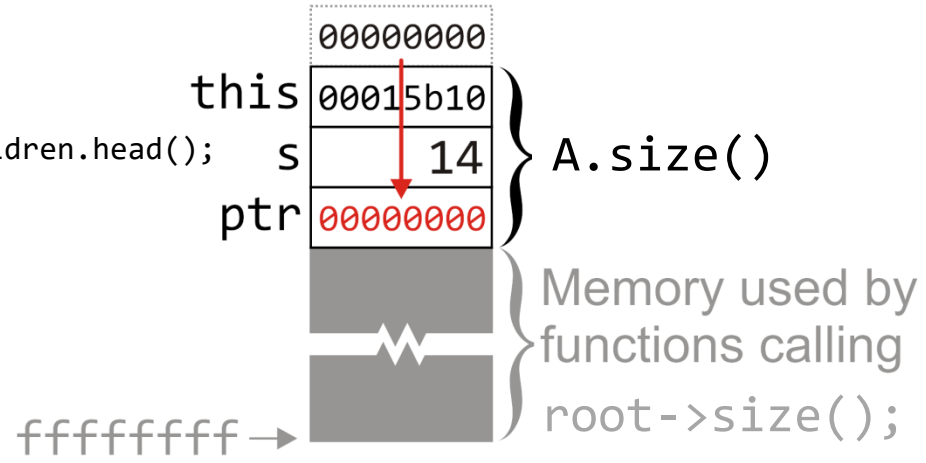


Example

The returned value of `ptr->next()` is copied to the appropriate location for the variable `ptr`

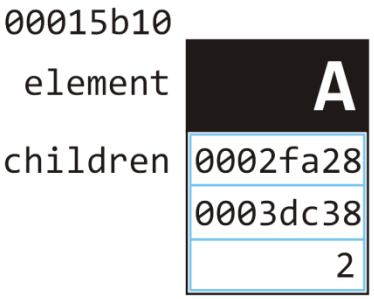



```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```

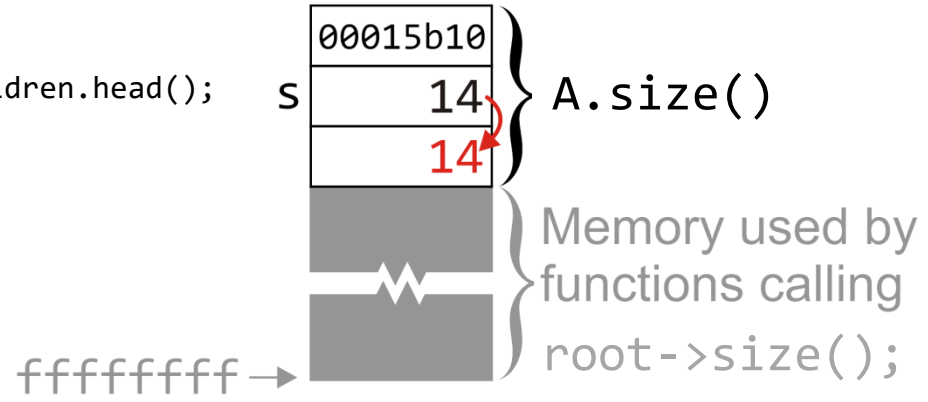


Example

At this point, `ptr != 0` returns false, so we prepare to return by copying the local variable `s` to the appropriate location




```
int Simple_tree::size() const {
   int s = 1;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      s += ptr->value()->size();
   }

   return s;
}
```



Example

We have returned to whatever function originally called

```
root->size();
```

That function can access the value returned by the call, either assigning it, using it as a further function call, or possibly discarding it