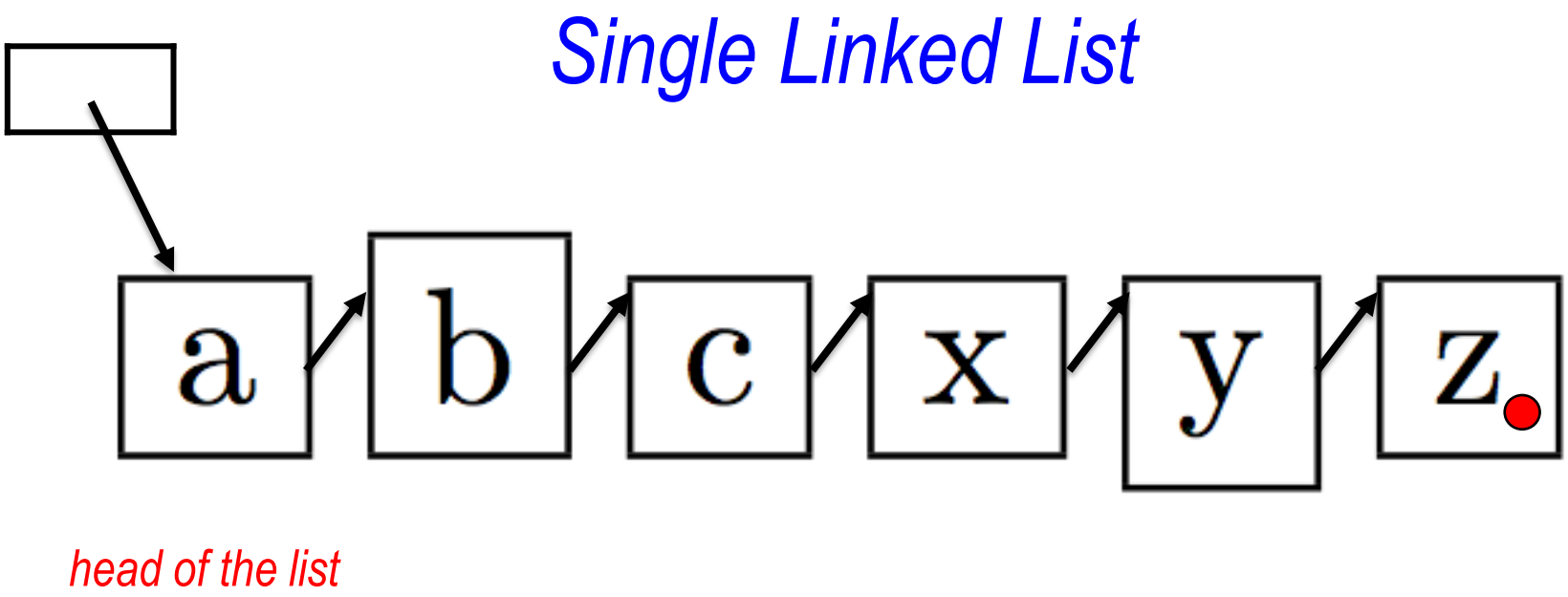
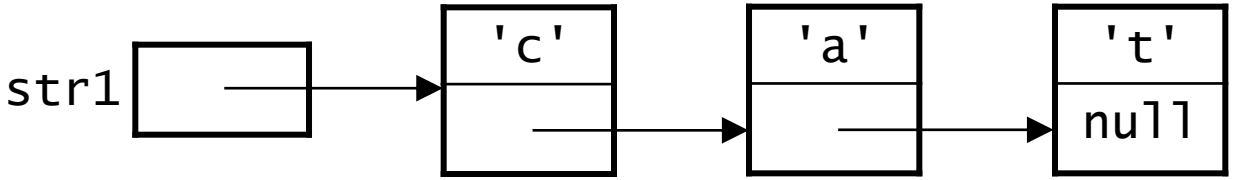
Case Study

- A linked list class to represent a string as a linked list of characters.



Example:

A String as a Linked List of Characters

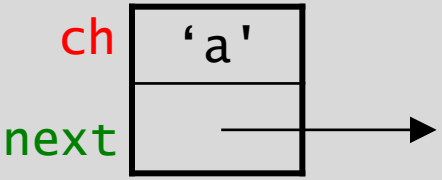


- Each node in the linked list represents one character.

- Java class for this type of node:

```
public class StringNode {
    private char ch;
    private StringNode next;
```

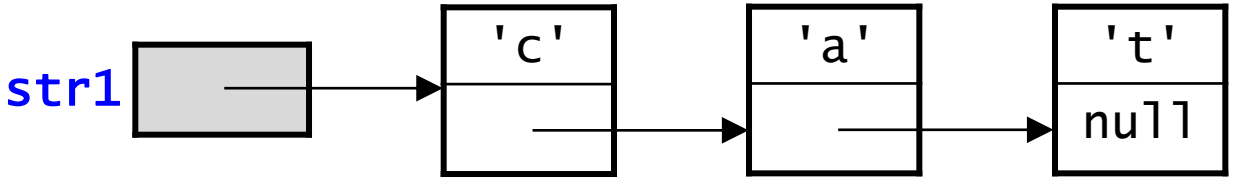
same type as the node itself!




```
    public StringNode(char c, StringNode n) {
        this.ch = c;
        this.next = n;
    }
    ...
}
```

Example:

A String as a Linked List of Characters

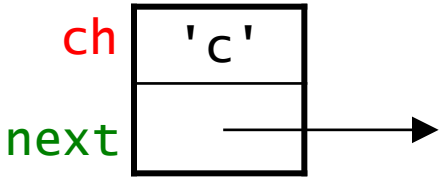


- Each node in the linked list represents one character.

- Java class for this type of node:

```
public class StringNode {
    private char ch;
    private StringNode next;

    public StringNode(char c, StringNode n) {
        this.ch = c;
        this.next = n;
    }
    ...
}
```

same type as the node itself!



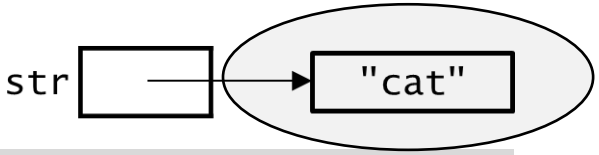
- The string as a whole is represented by a variable that holds a reference to the node for the first character (e.g., `str1` above).

A Linked List Is a Recursive Data Structure!

- Recursive definition of a linked list: a linked list is either
  a) empty or
  b) a single node, followed by a linked list

Recall: Recursively Finding the Length of a String

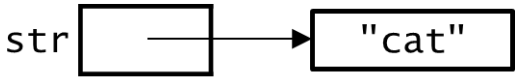
- For a built-in Java `String` object:




```
public static int length(String str) {
    if (str == null || str.equals("")) {
        return 0;
    } else {
        int lenRest = length(str.substring(1));
        return 1 + lenRest;
    }
}
```

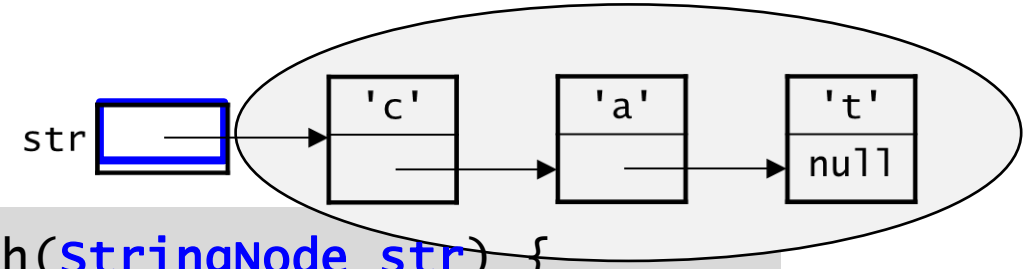
Recursively Finding the Length of a String

- For a built-in Java `String` object:




```
public static int length(String str) {
    if (str == null || str.equals("")) {
        return 0;
    } else {
        int lenRest = length(str.substring(1));
        return 1 + lenRest;
    }
}
```

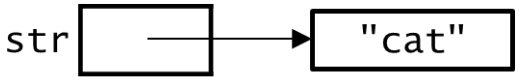
- **For a linked-list string:**




```
public static int length(StringNode str) {
    if (???) {
        return 0;
    } else {
        int lenRest = length(???);
        return 1 + lenRest;
    }
}
```

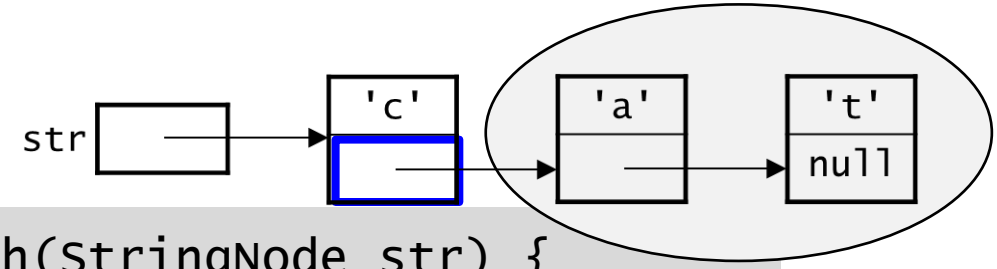
Recursively Finding the Length of a String

- For a built-in Java `String` object:




```
public static int length(String str) {
    if (str == null || str.equals("")) {
        return 0;
    } else {
        int lenRest = length(str.substring(1));
        return 1 + lenRest;
    }
}
```

- For a linked-list string:




```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

Recursively Finding the Length of a String

An Alternative Version of the Method

- Original version:

```
public static int length(StringNode str) {
    if (str == null || str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

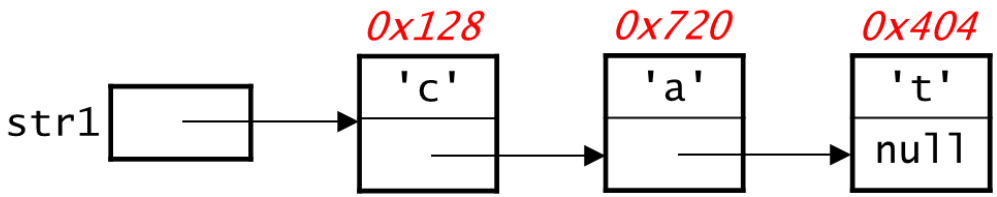
- Version without a variable for the result of the recursive call:

```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```

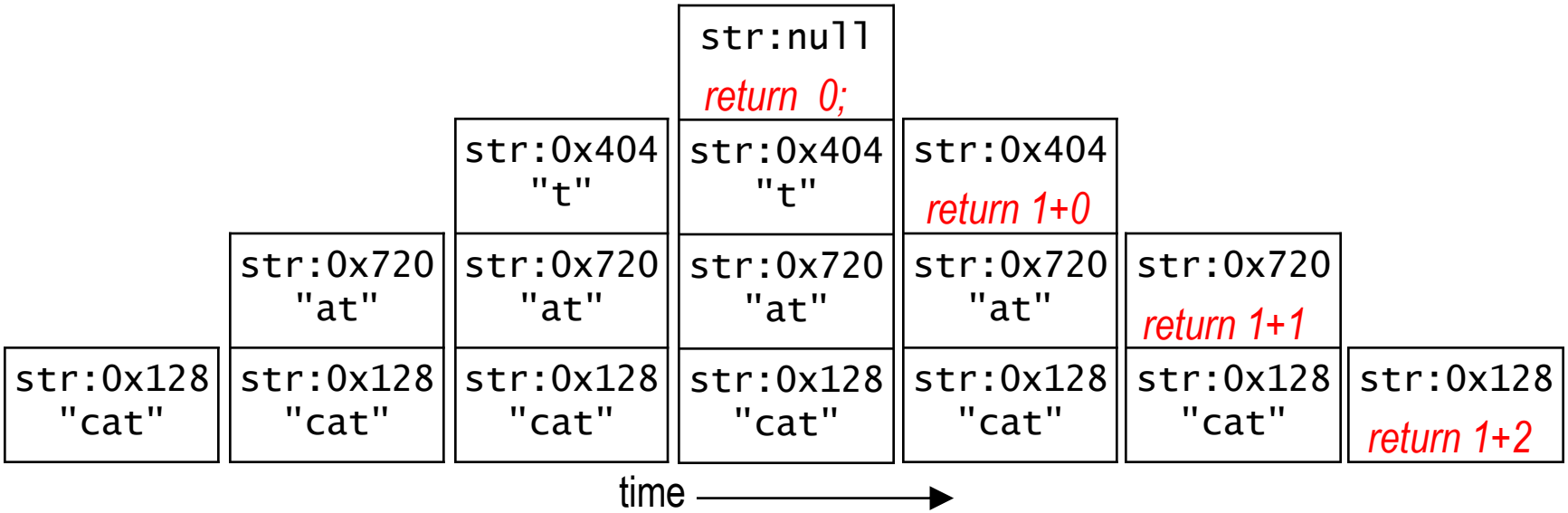
Tracing `length()`:

the recursive method

```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```



- Example: `StringNode.length(str1)`



Get the Node at Position i in a Linked List:

a recursive method

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Recursive approach:

```
private static StringNode getNode(StringNode str, int i) {

    if (i < 0 || str == null) { // base case 1: not found
        return null;
    } else if (i == 0) {        // base case 2: just found
        return str;
    } else {                    // recursive case
        return getNode(str.next, i - 1);
    }

}
```

Get the Node at Position i in a Linked List:

*a recursive method, an **alternative** implementation*

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Recursive approach:

```
private static StringNode getNode(StringNode str, int i) {
    String rets = null;

    // base case 1: implied
    if (str != null && i >= 0) {
        if (i == 0)          // base case 2: just found
            rets = str;
        else                 // recursive case
            rets = getNode(str.next, i-1);
    }

    return( rets );
}
```

Get the Node at Position i in a Linked List:

an iterative method

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Iterative approach:

```
private static StringNode getNode(StringNode str, int i) {
    StringNode node = null;

    if ( i >= 0 ) {
       int count = 0;

       while (str != null) {
          if (count++ == i){      // found the node
             node = str;          // assign to return
             break;
          }
          str = str.next;         // reference the next node
       }
    }
    return(node);
}
```

Get the Node at Position i in a Linked List:

an iterative method

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Iterative approach:

```
private static StringNode getNode(StringNode str, int i) {
    StringNode node = null;

    if ( i >= 0 ) {
       int count = 0;

       while (str != null) {
          if (count++ == i){      // found the node
             node = str;          // assign to return
             break;
          }
          str = str.next;         // reference the next node
       }
    }
    return(node);
}
```

Get the Node at Position i in a Linked List:

an iterative method

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Iterative approach:

```
private static StringNode getNode(StringNode str, int i) {
    StringNode node = null;

    if ( i >= 0 ) {
       int count = 0;

       while (str != null) {
          if (count++ == i){      // found the node
             node = str;          // assign to return
             break;
          }
          str = str.next;         // reference the next node
       }
    }
    return(node);
}
```

Get the Node at Position i in a Linked List:

an iterative method

- `getNode(str, i)` – a private *helper* method that returns a reference to the ith node in the linked list (i == 0 for the first node)

- Iterative approach:

```
private static StringN                                    {
    StringNode node

     if ( i >= 0 )
        int count = 0

        while (str !=
            if (count++ ==                    e node
                node =                  //        to return
                breal
            }
            str = str.next;             // reference the next node
        }
    }
     return(node);
}
```

Note that the variable `str` is being reassigned within this method. Will the variable that was passed to this method be impacted?

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Creating a Copy of a Linked List:

a recursive approach

- `copy(str)` – create a copy of `str` and return a reference to it

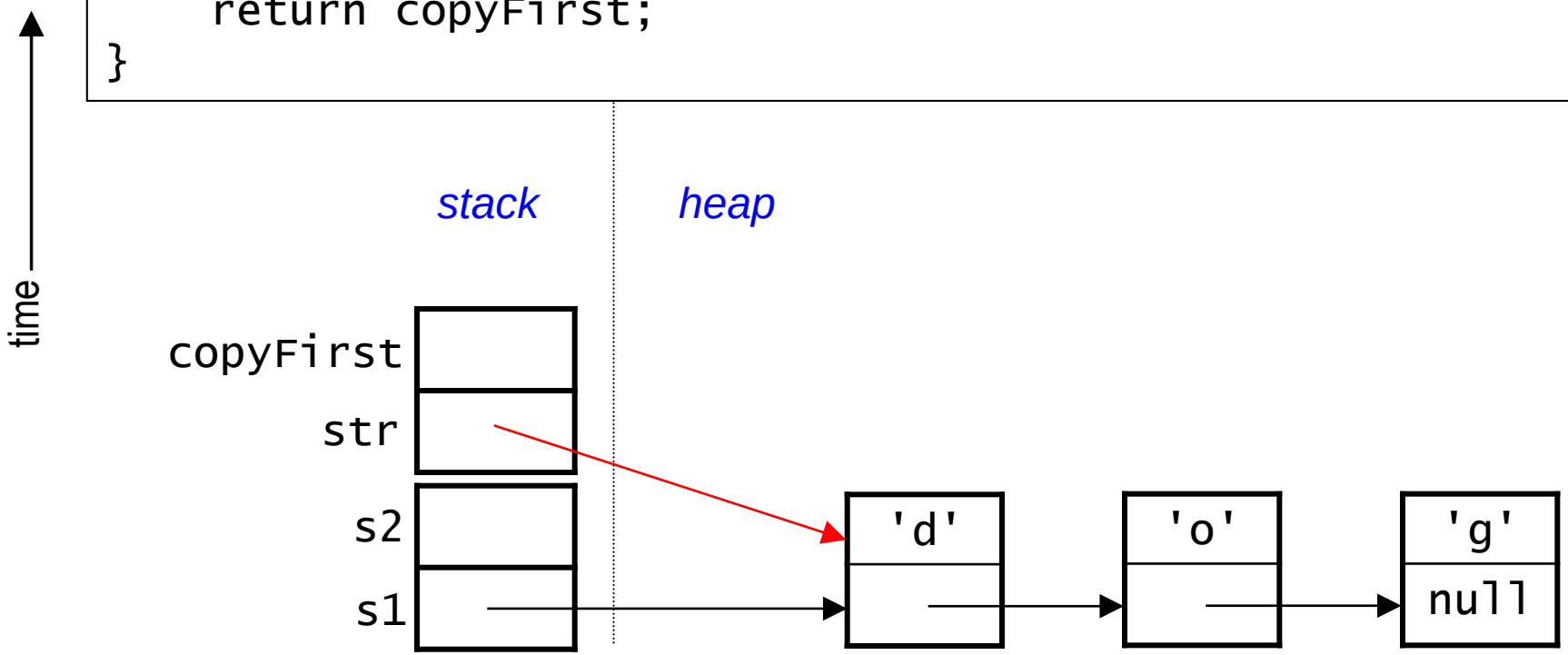
- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // copy the first node (the one to which str refers)
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to copy the rest, and
    // store the result in copyFirst's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Tracing copy(): the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

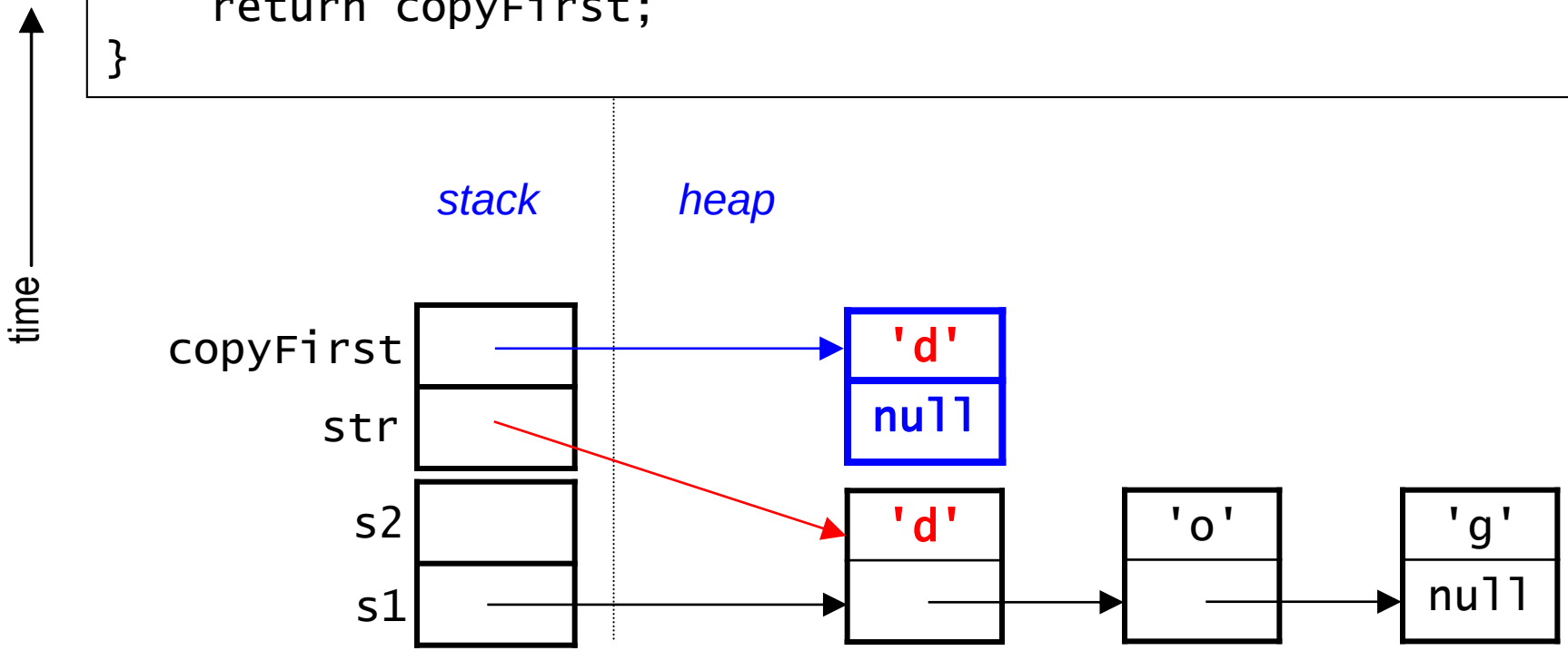
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

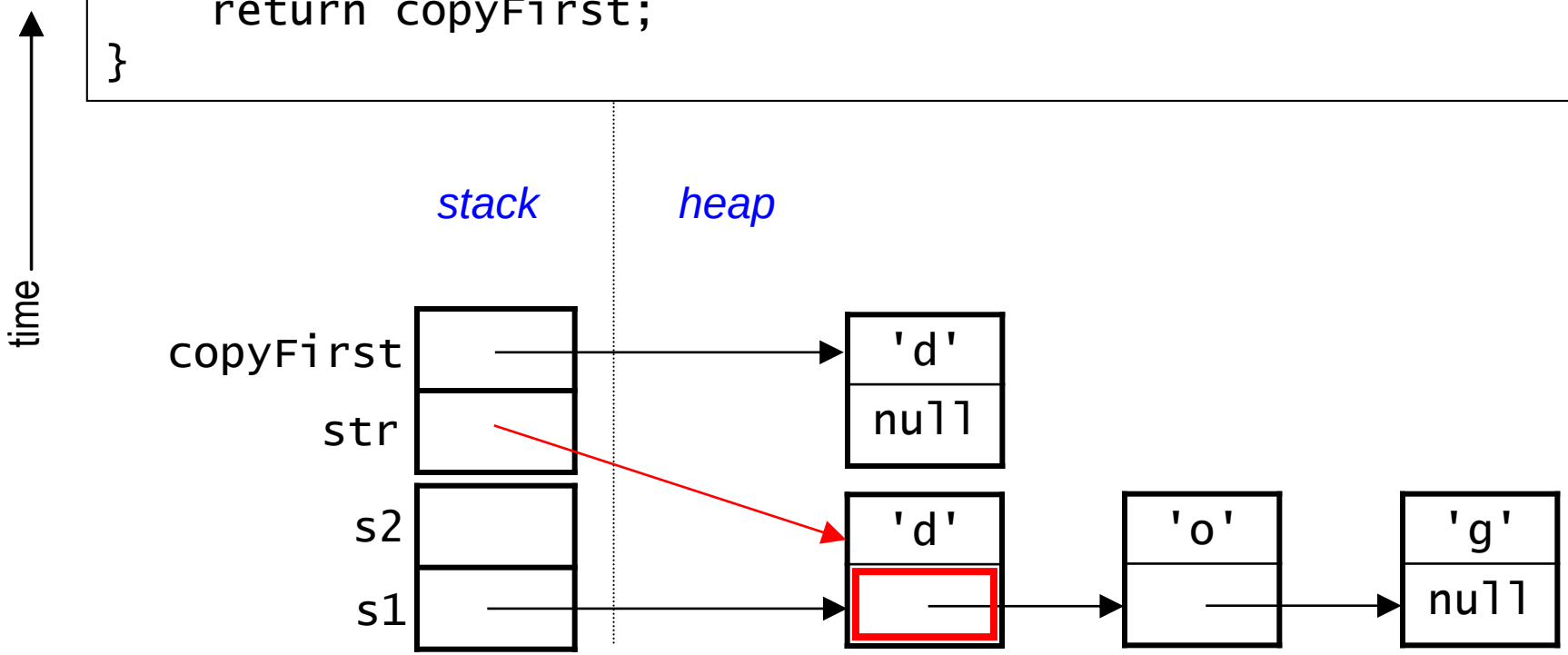
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

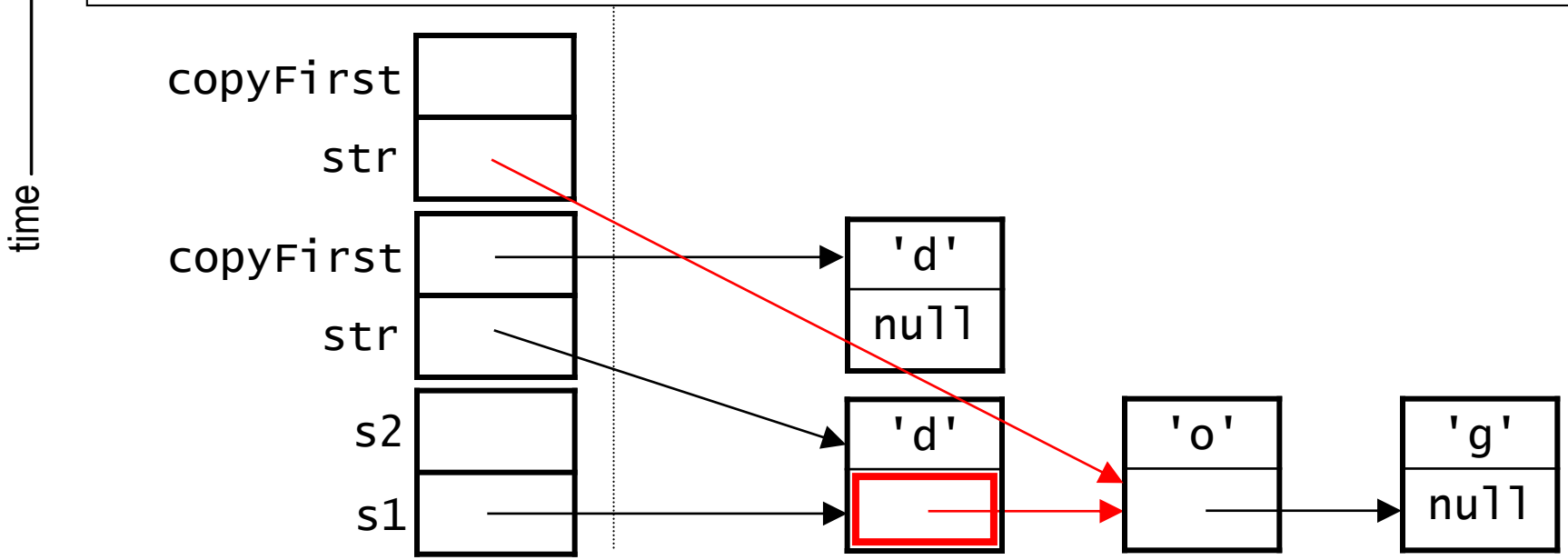
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

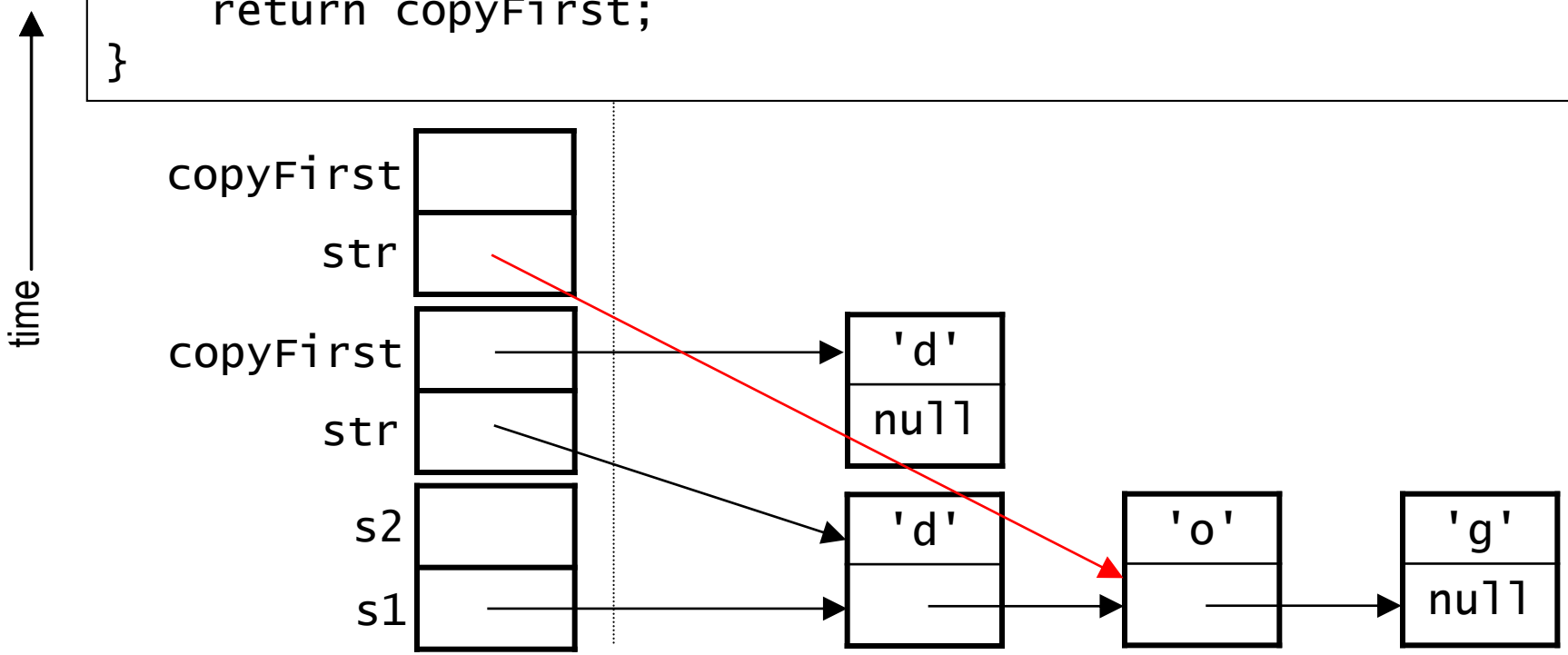
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the recursive calls

- From a client: `StringNode s2 = StringNode.copy(s1);`

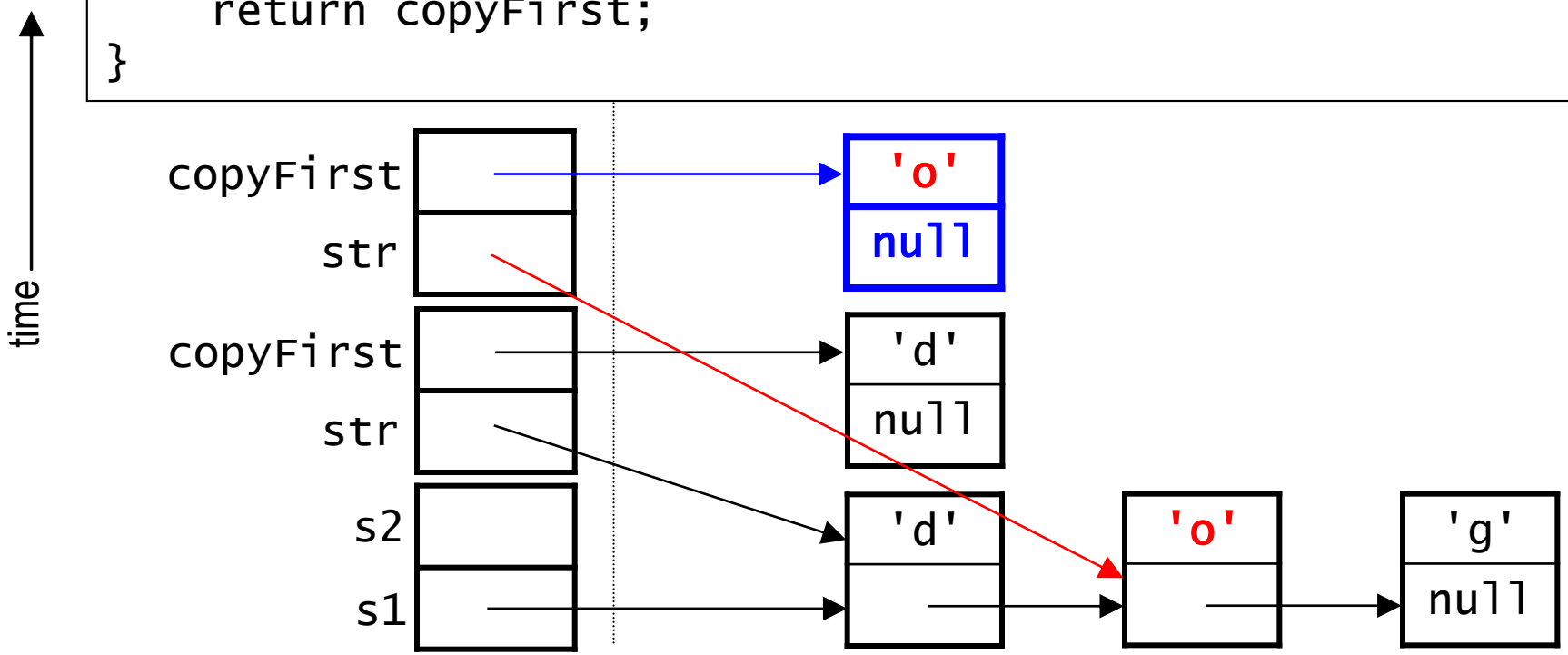
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the recursive calls

- From a client: `StringNode s2 = StringNode.copy(s1);`

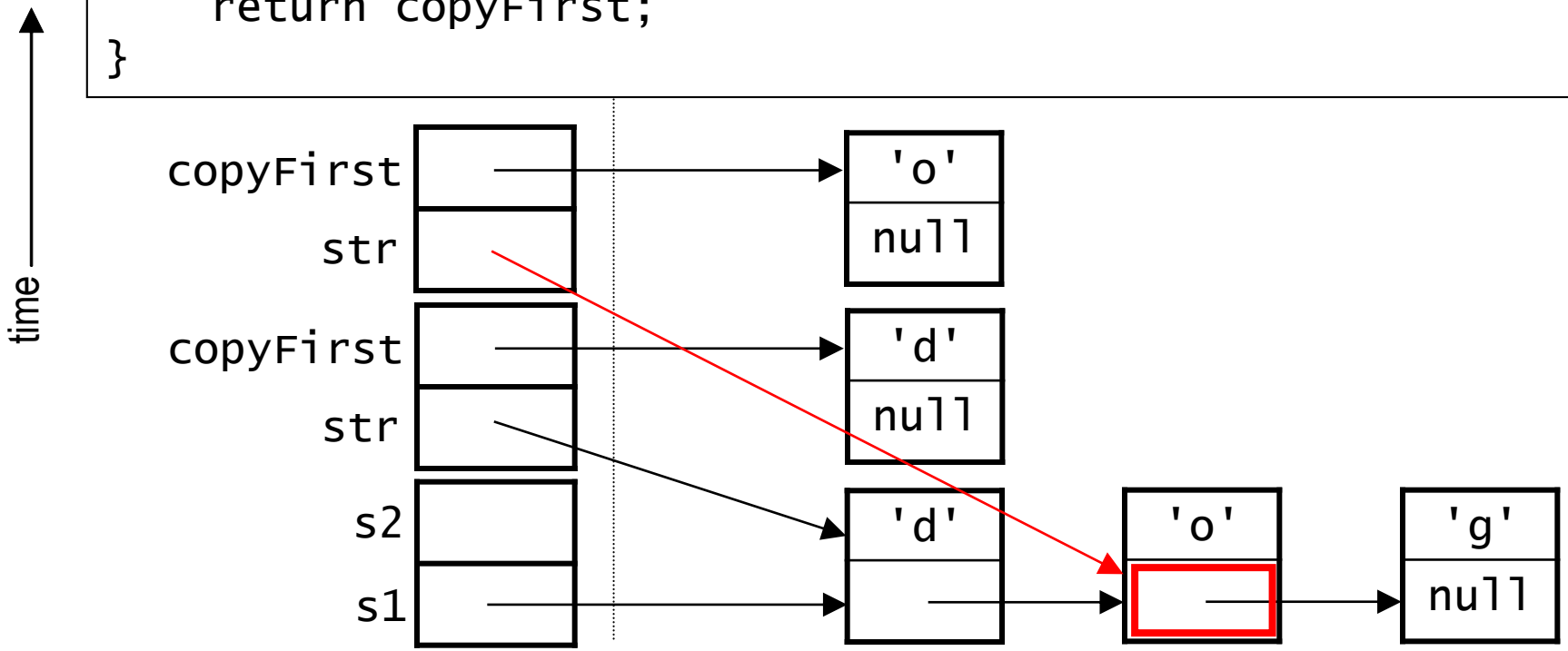
```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



Tracing copy(): the recursive calls

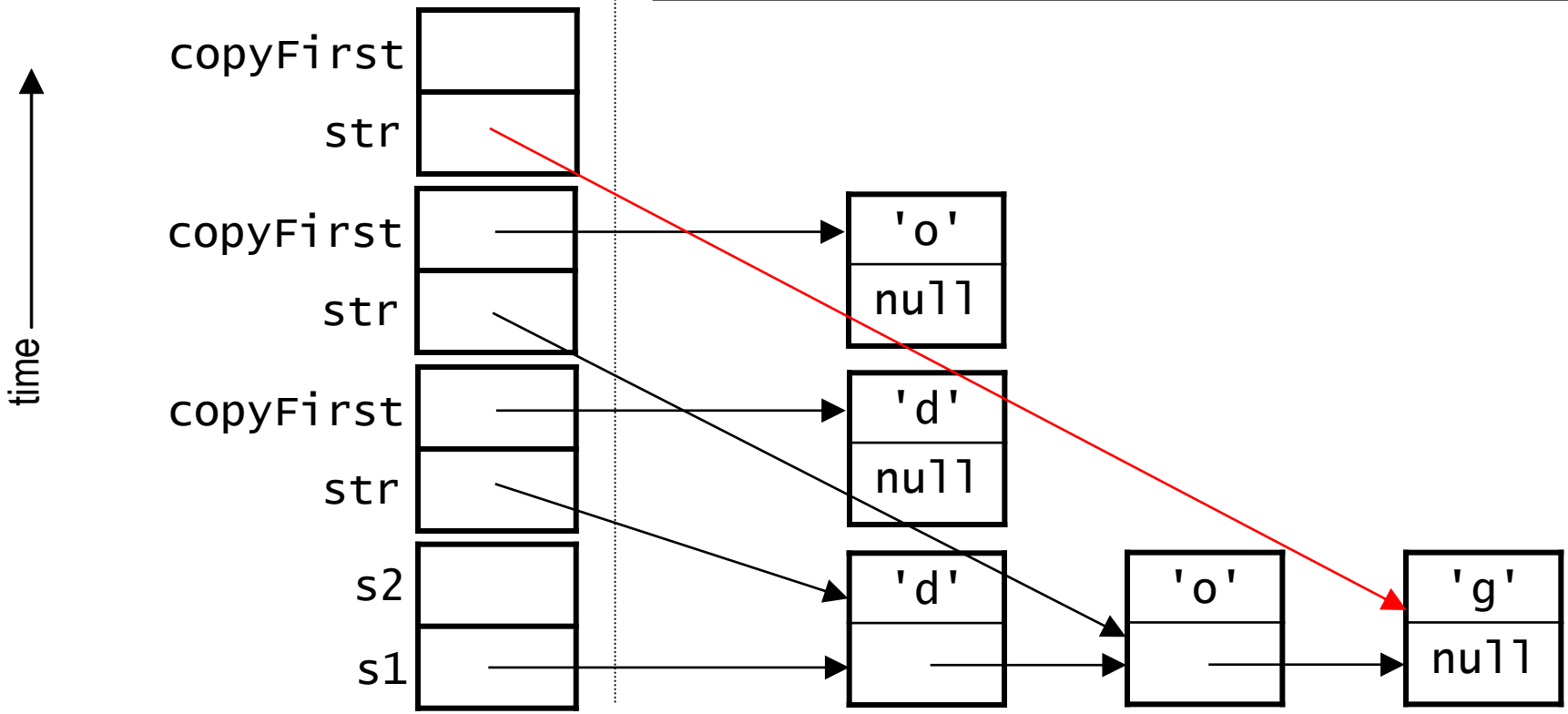
- From a client: `StringNode s2 = StringNode.copy(s1);`

```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new StringNode(str.ch, null);
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



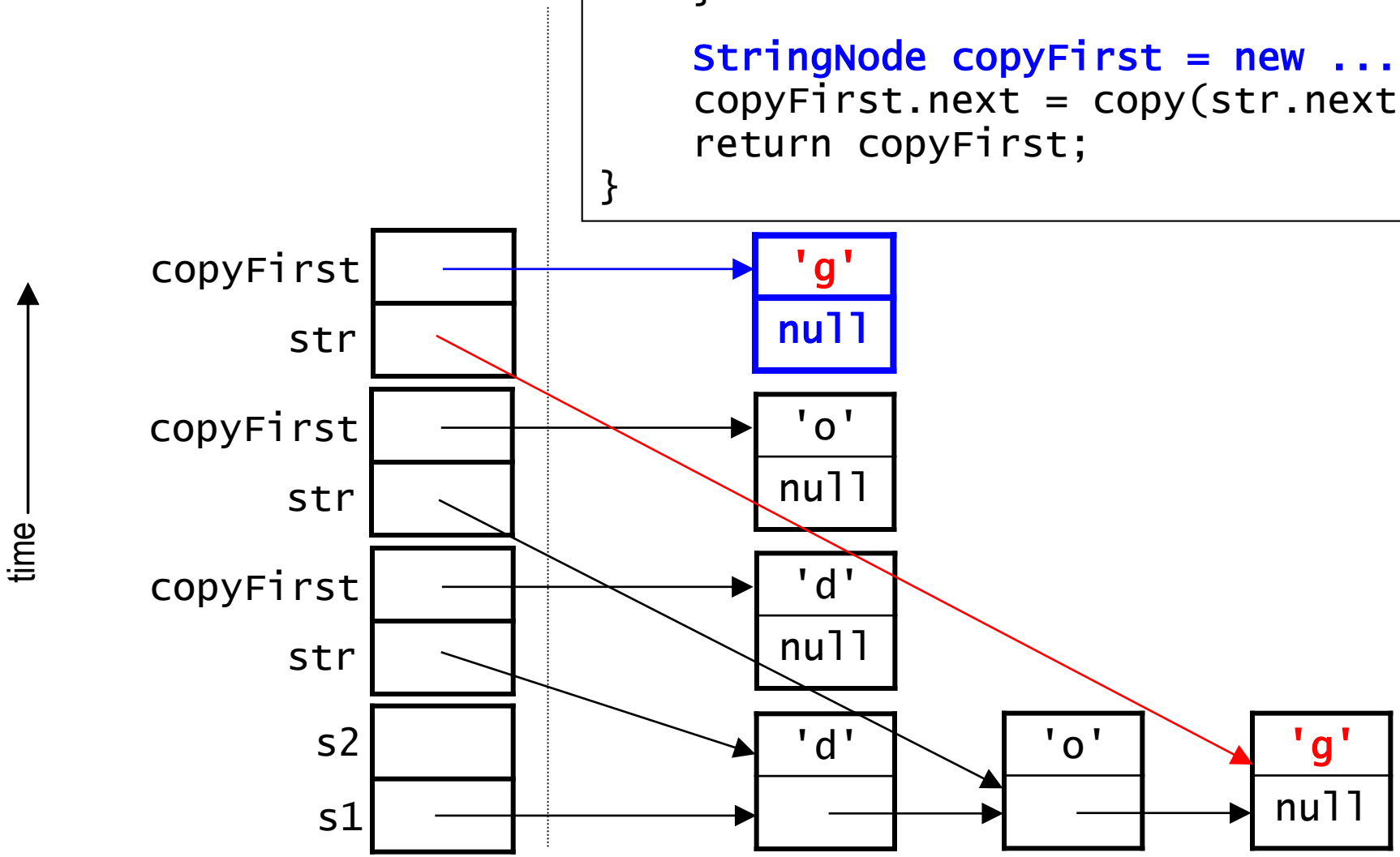
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



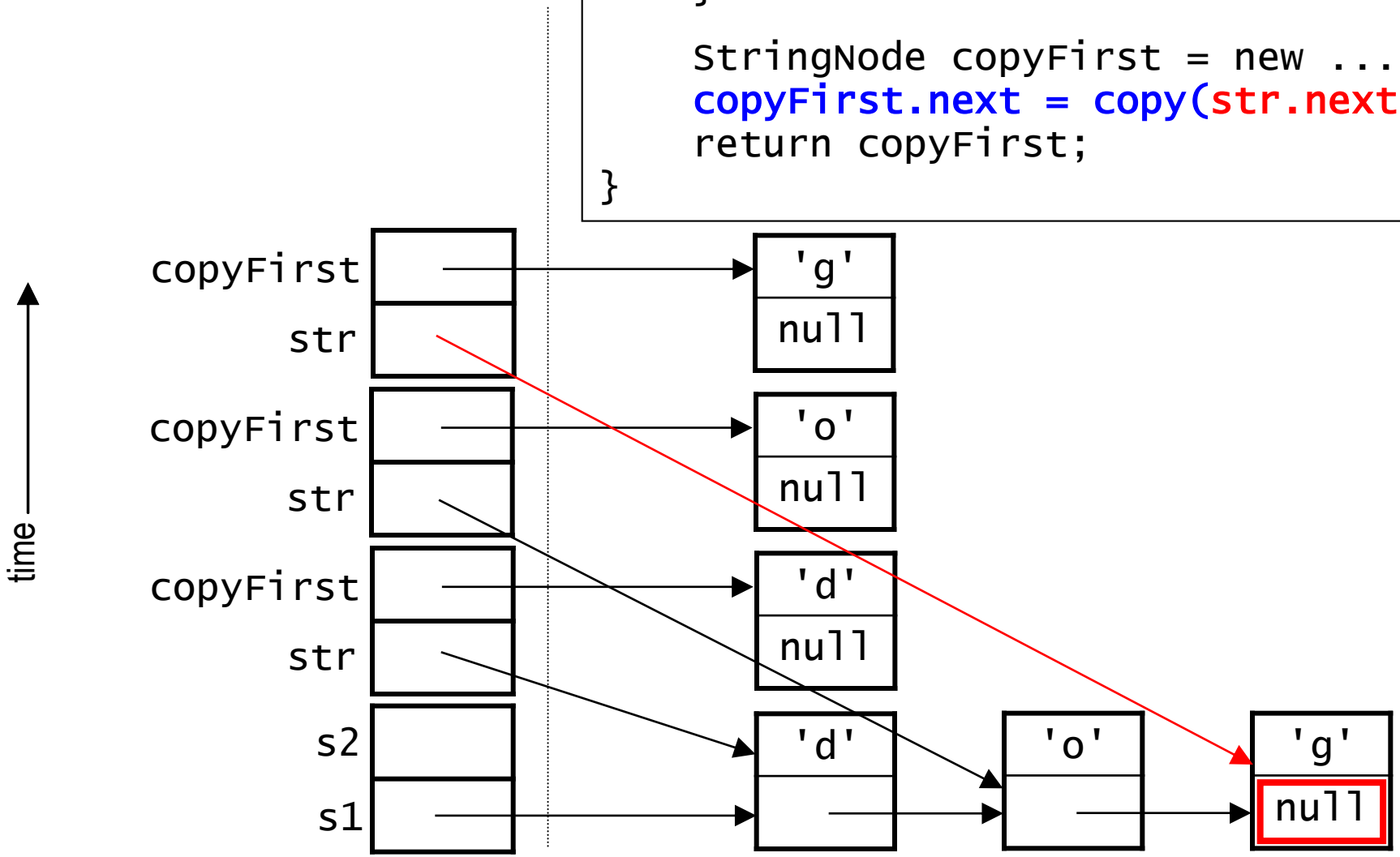
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



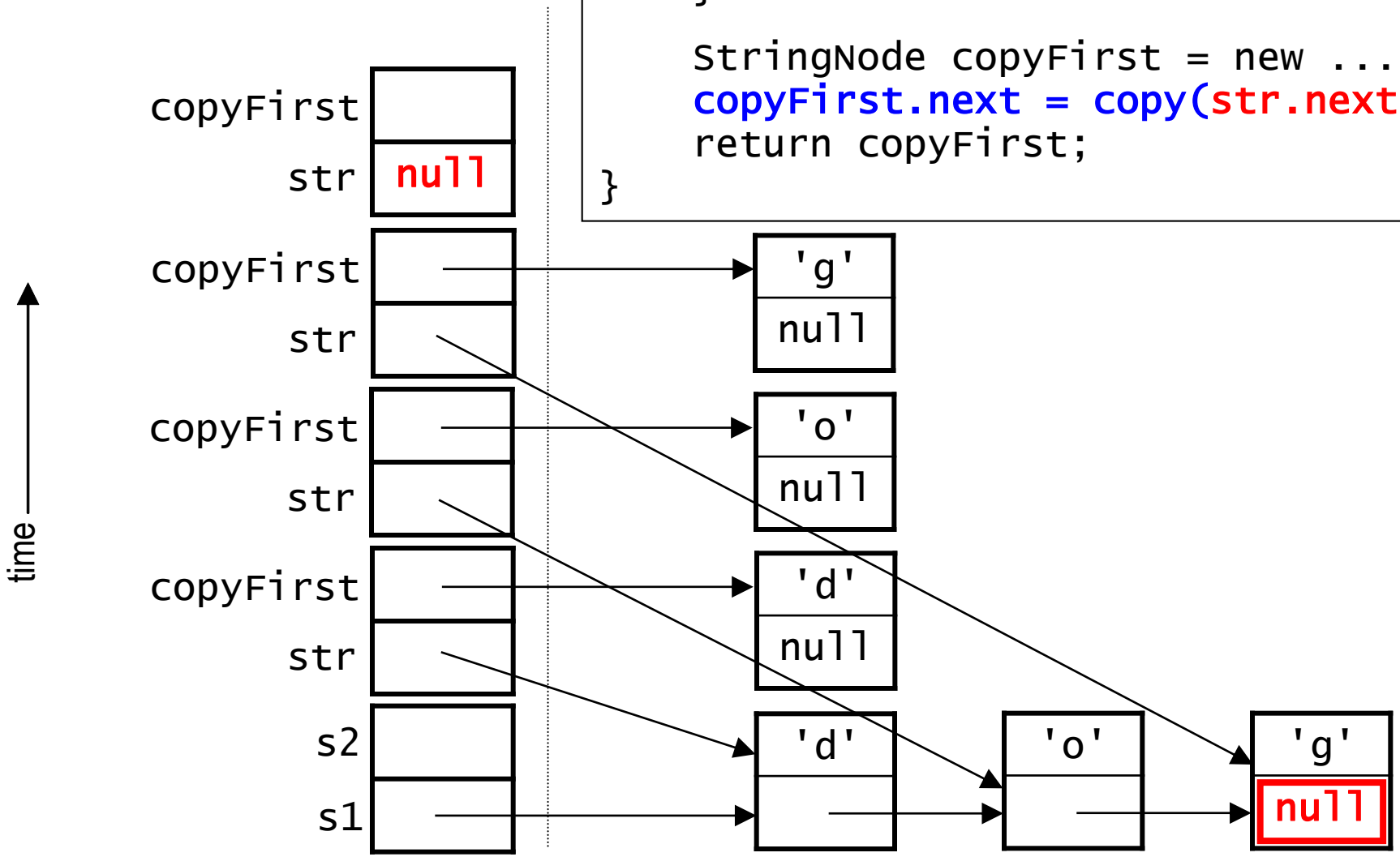
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



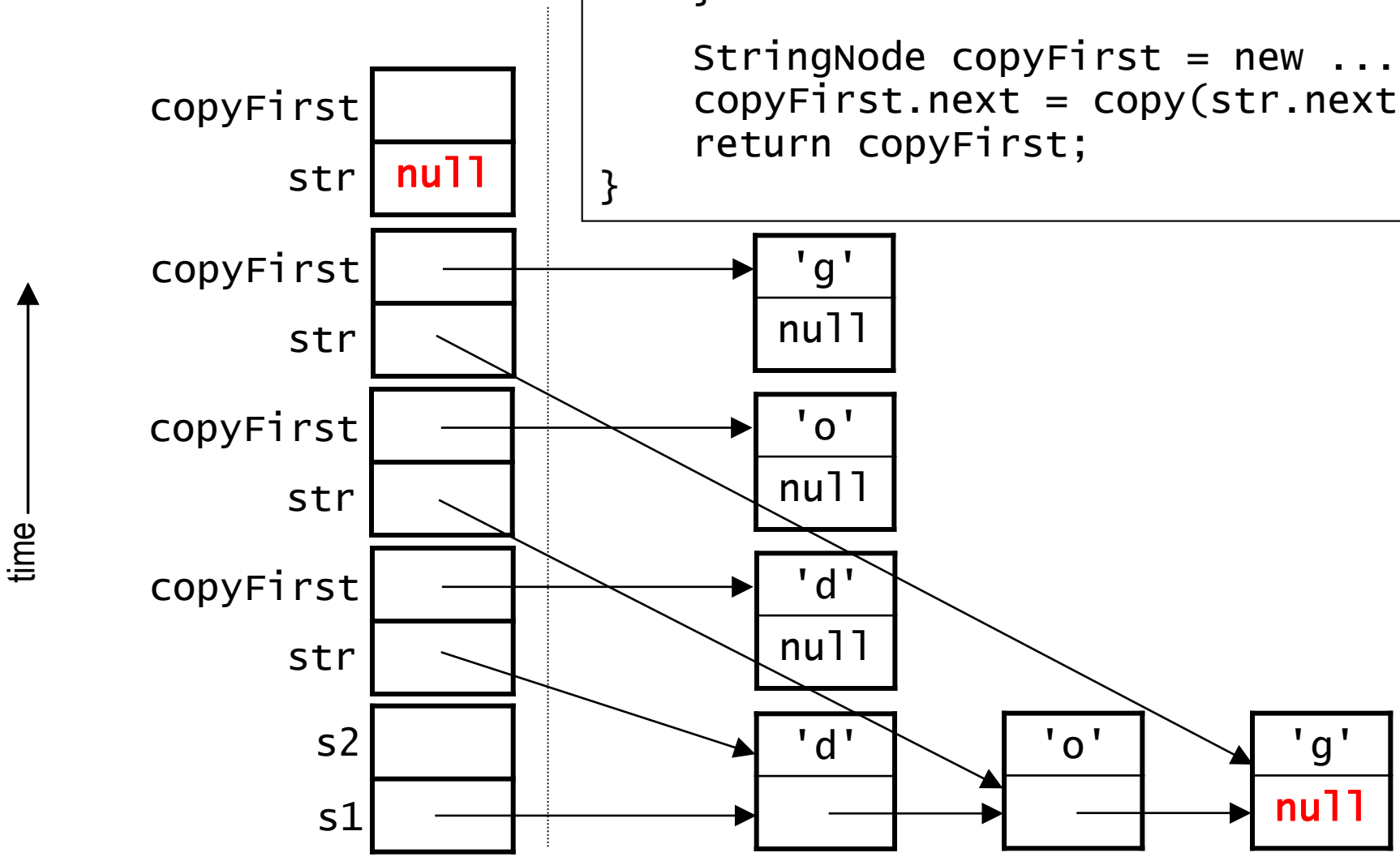
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



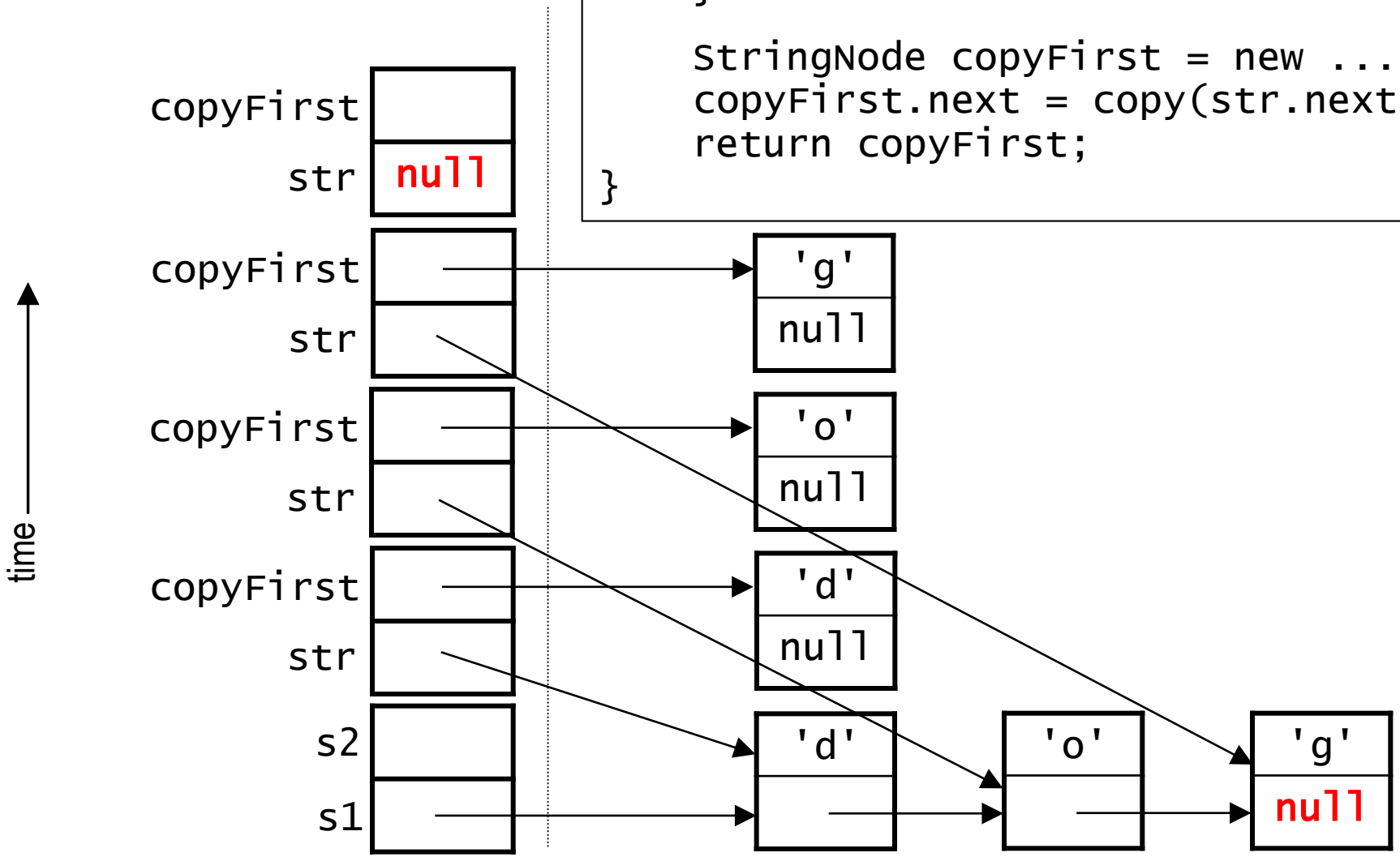
Tracing copy(): the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



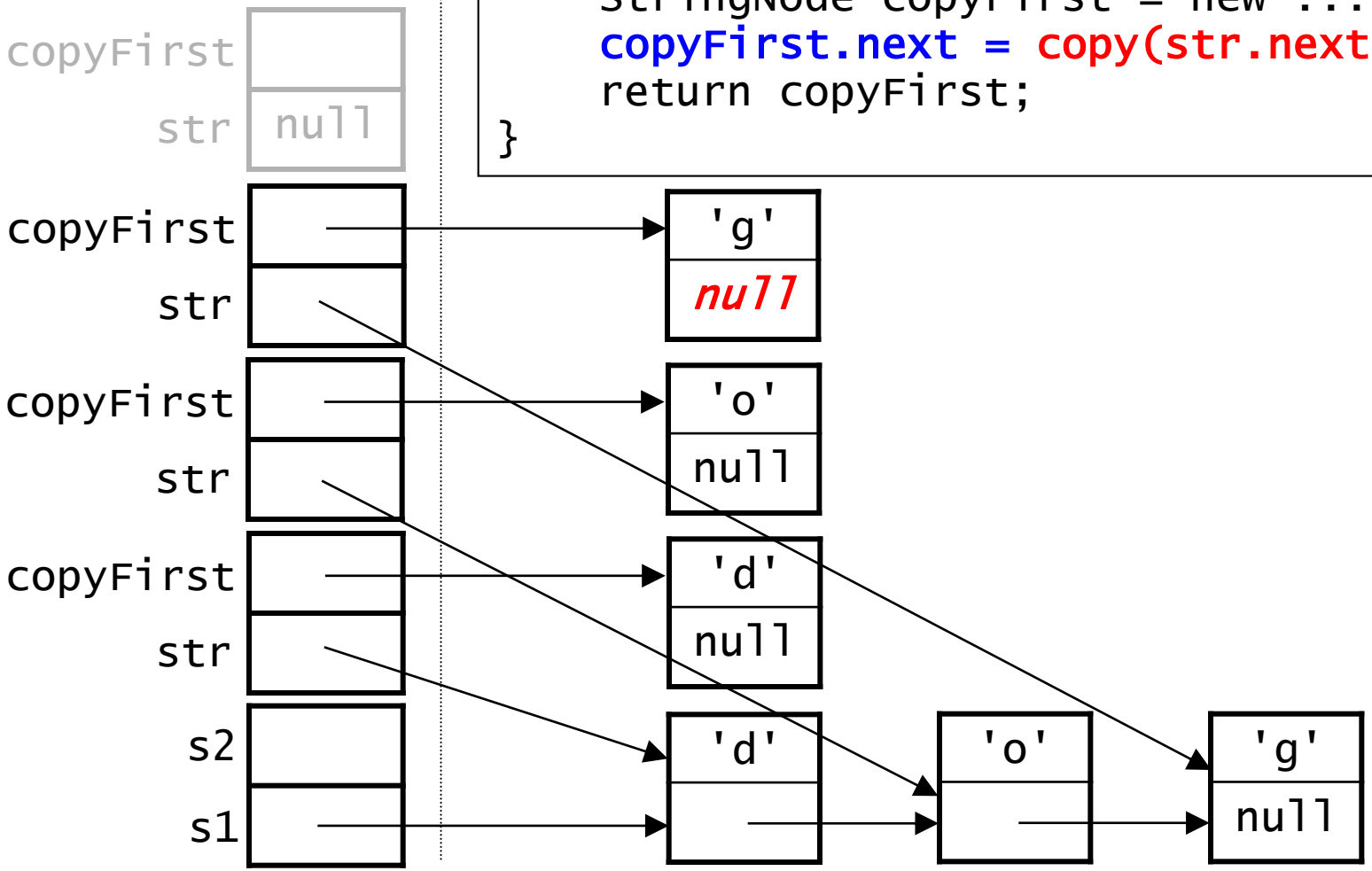
Tracing copy(): the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



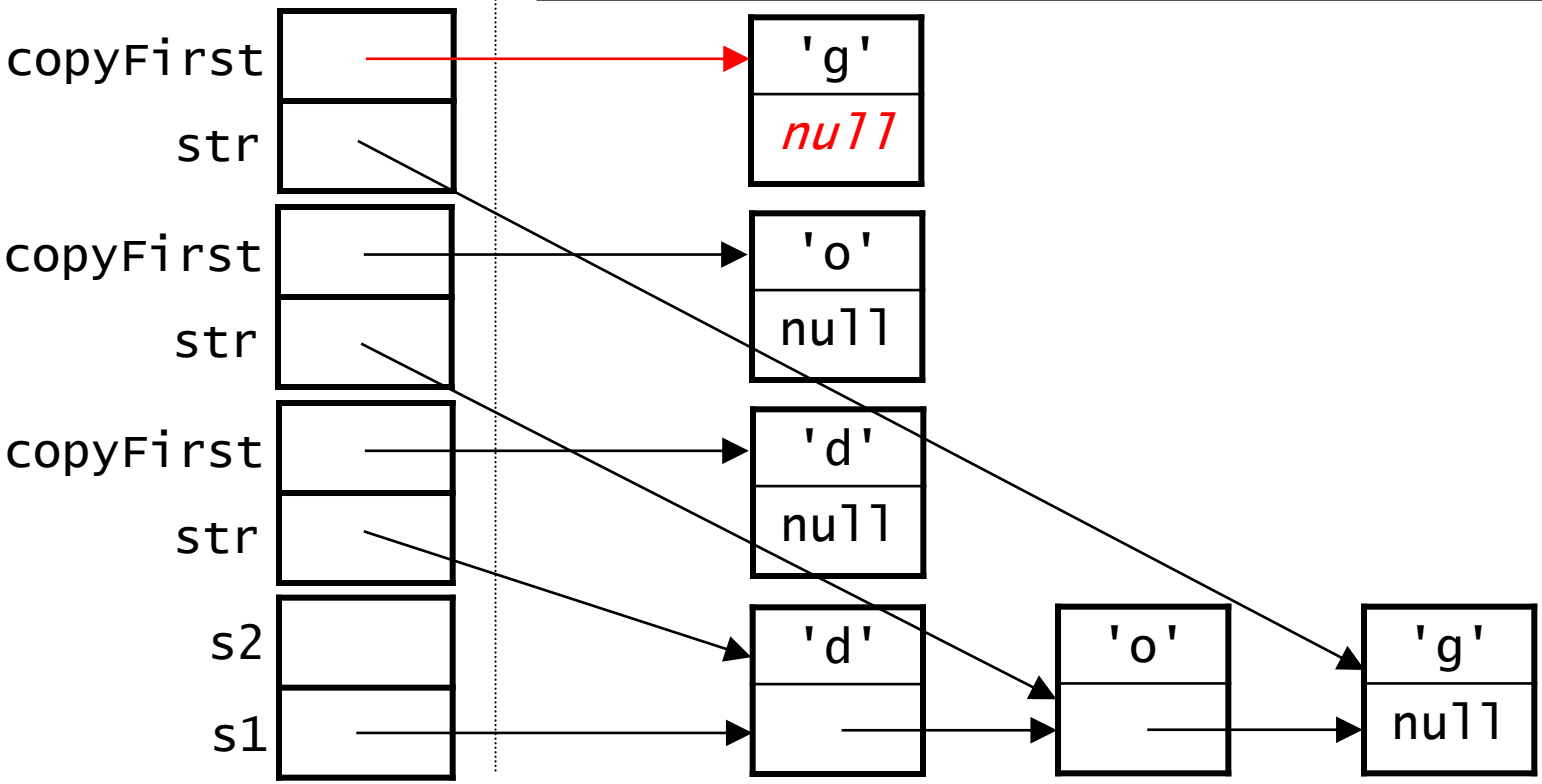
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



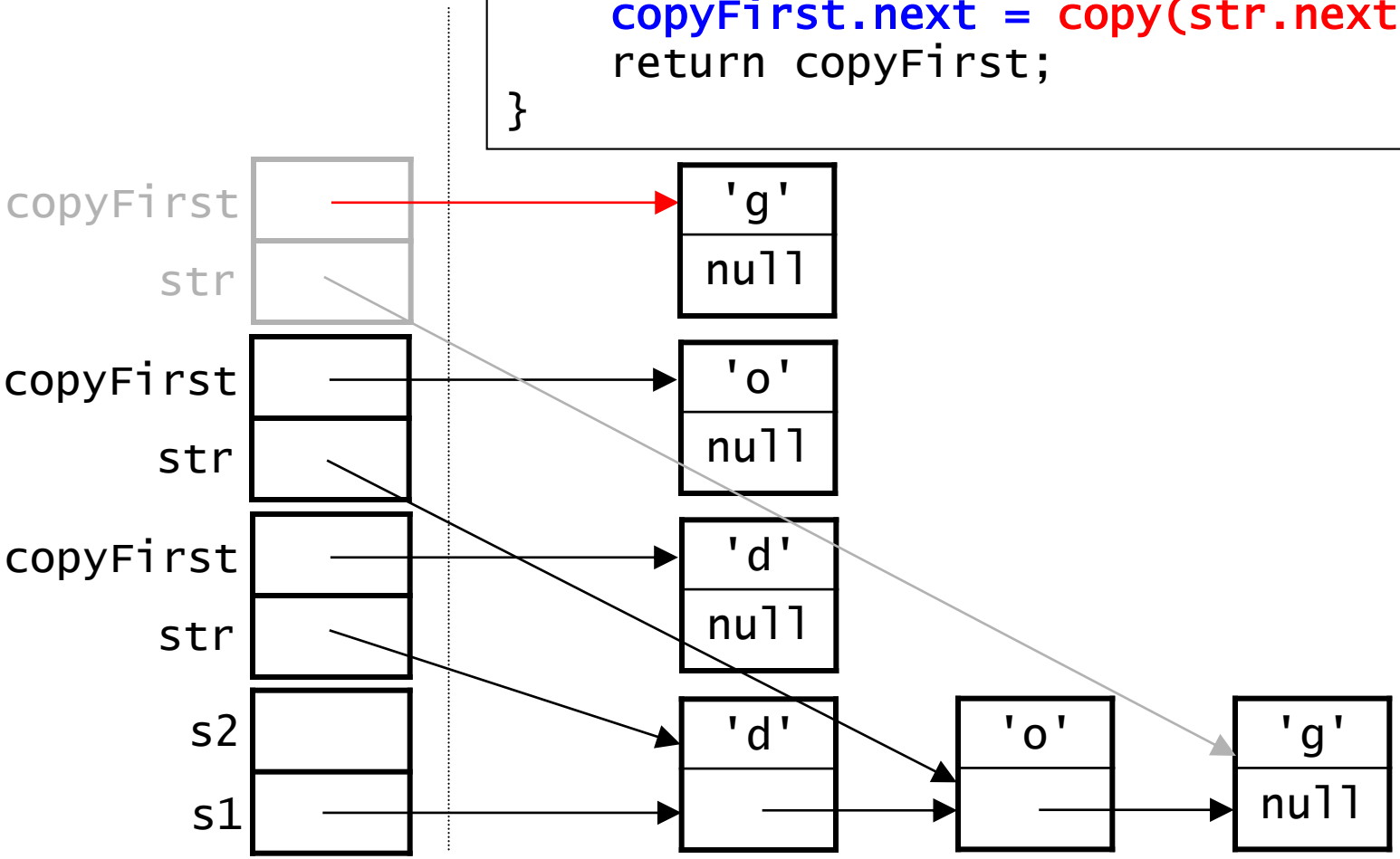
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



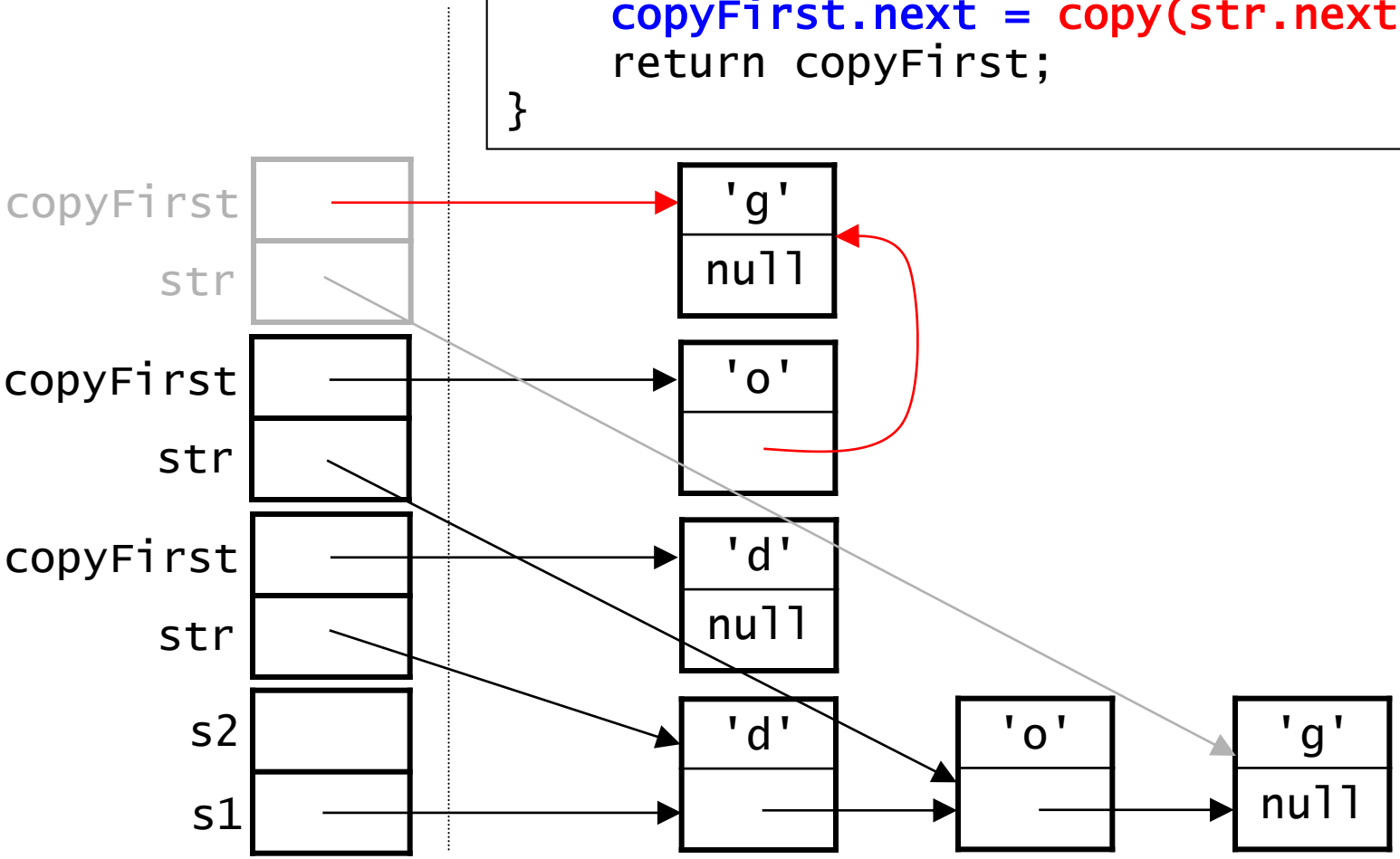
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



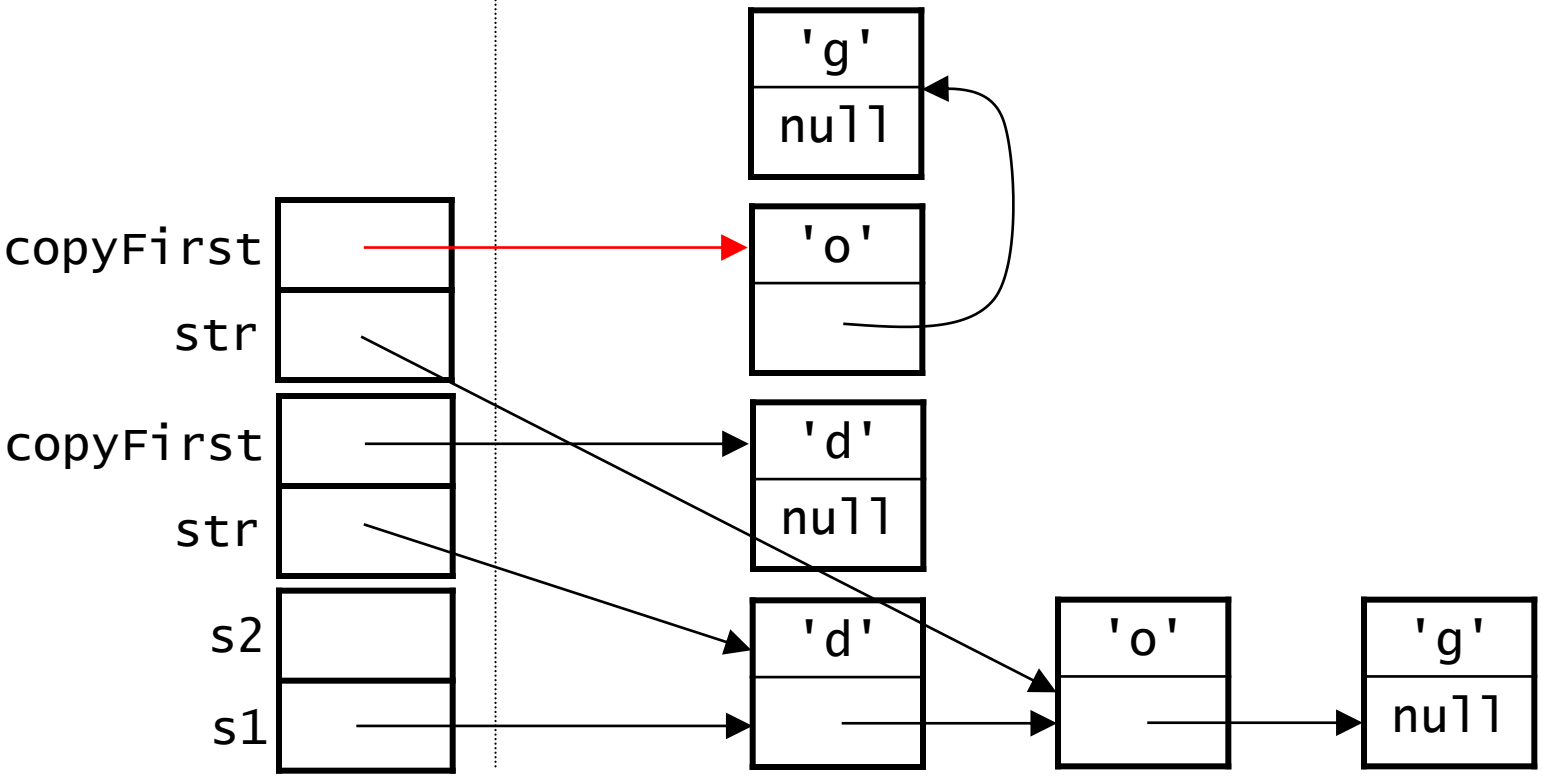
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



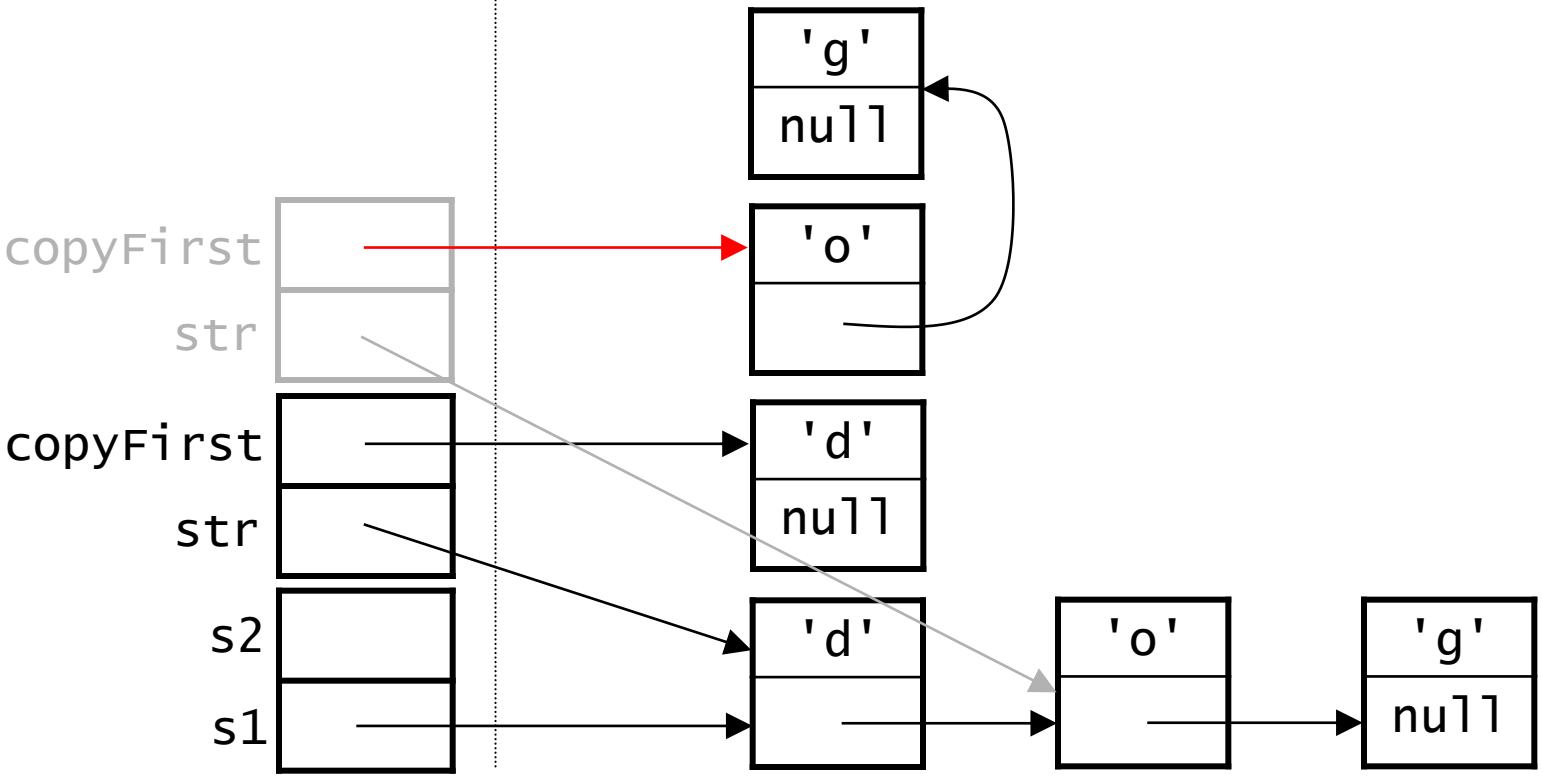
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



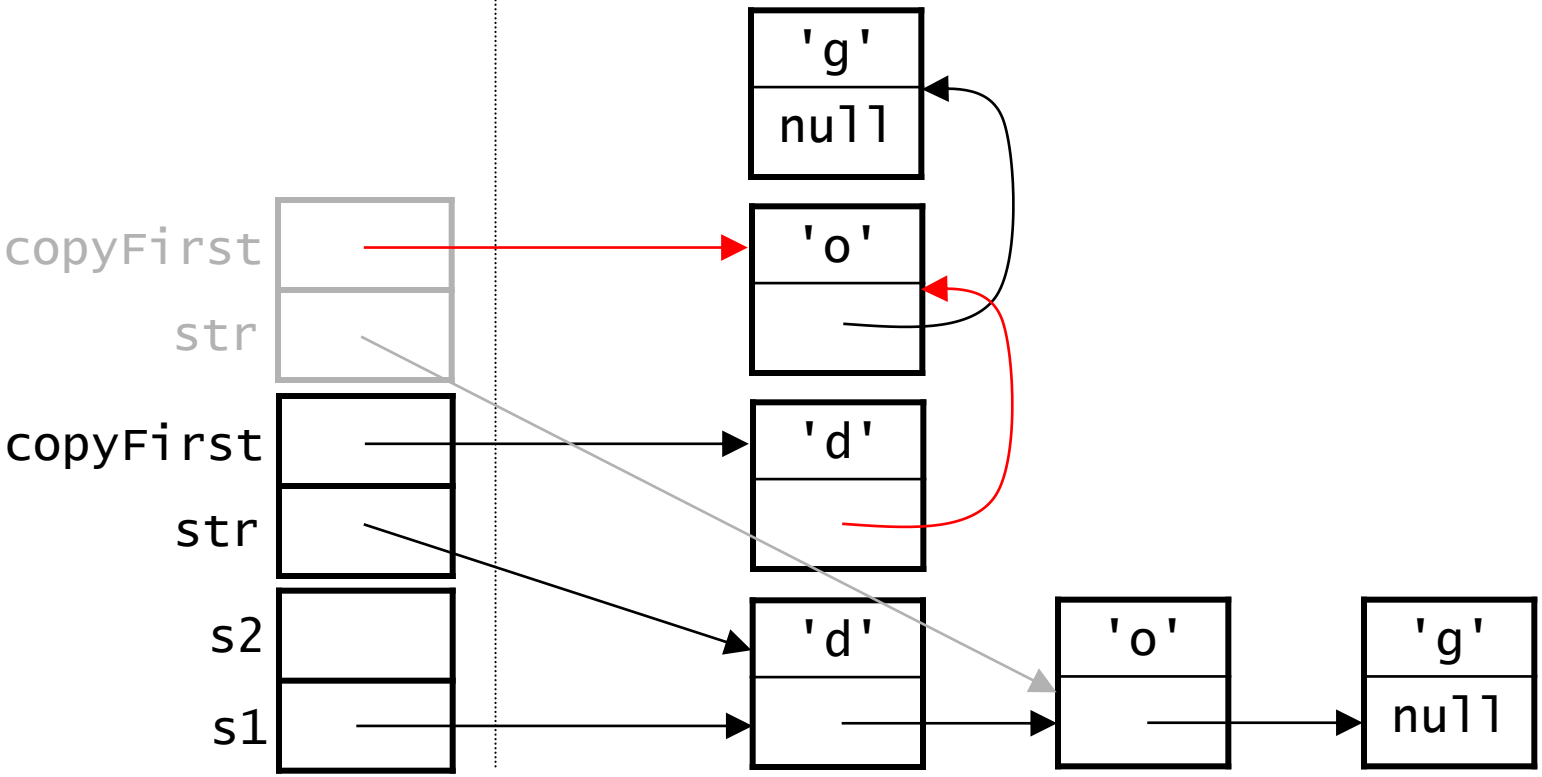
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



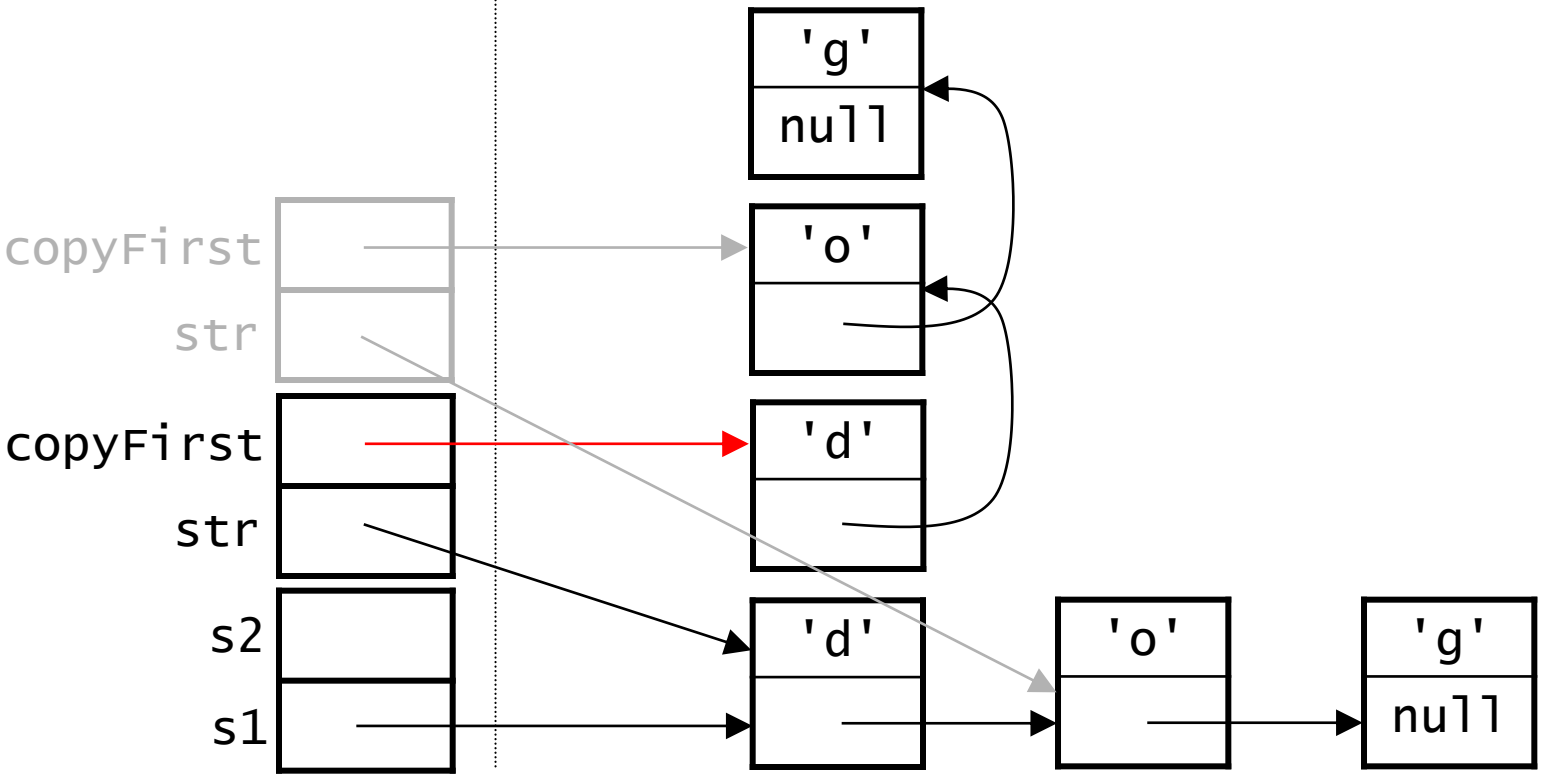
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



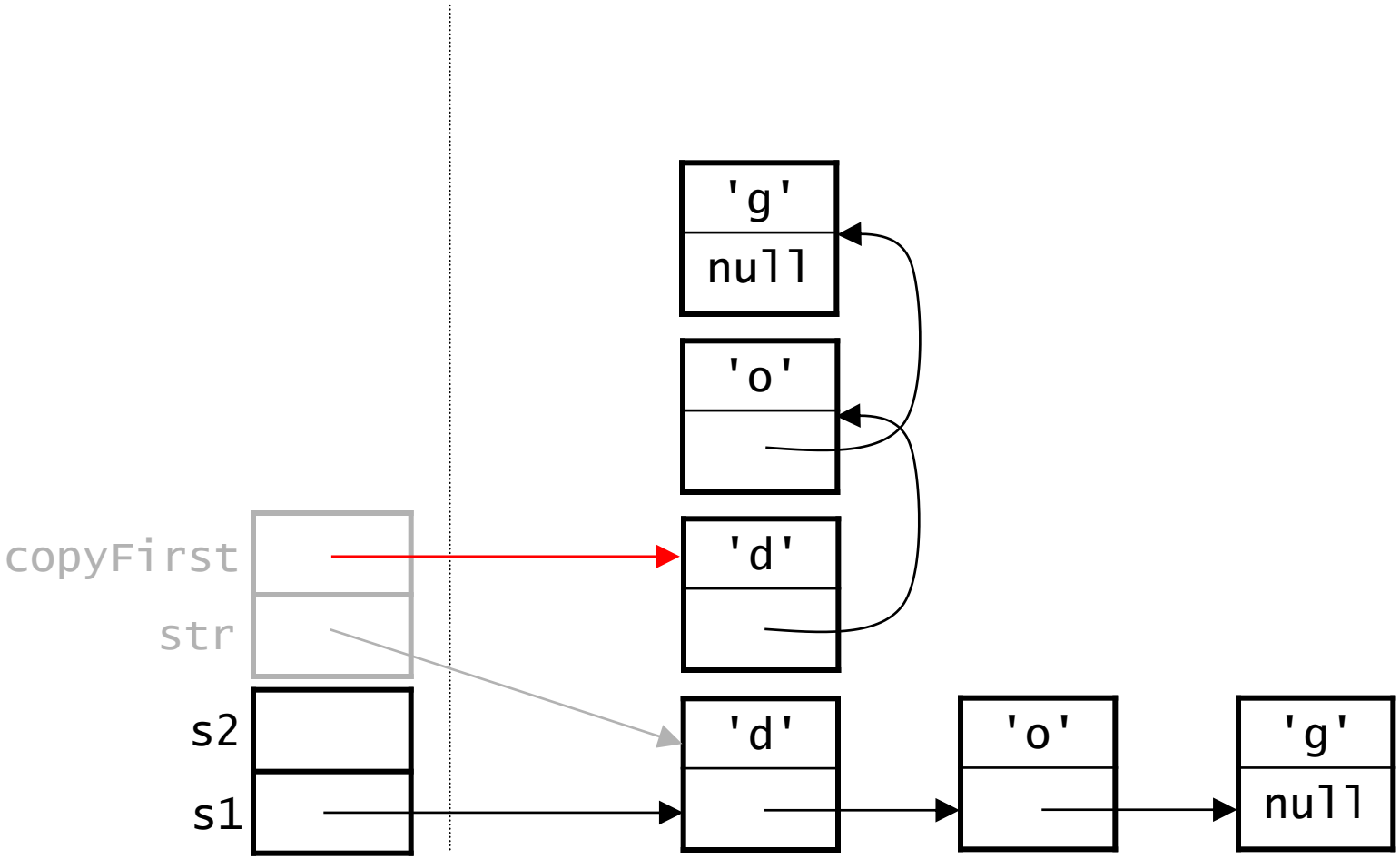
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyFirst = new ...
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```



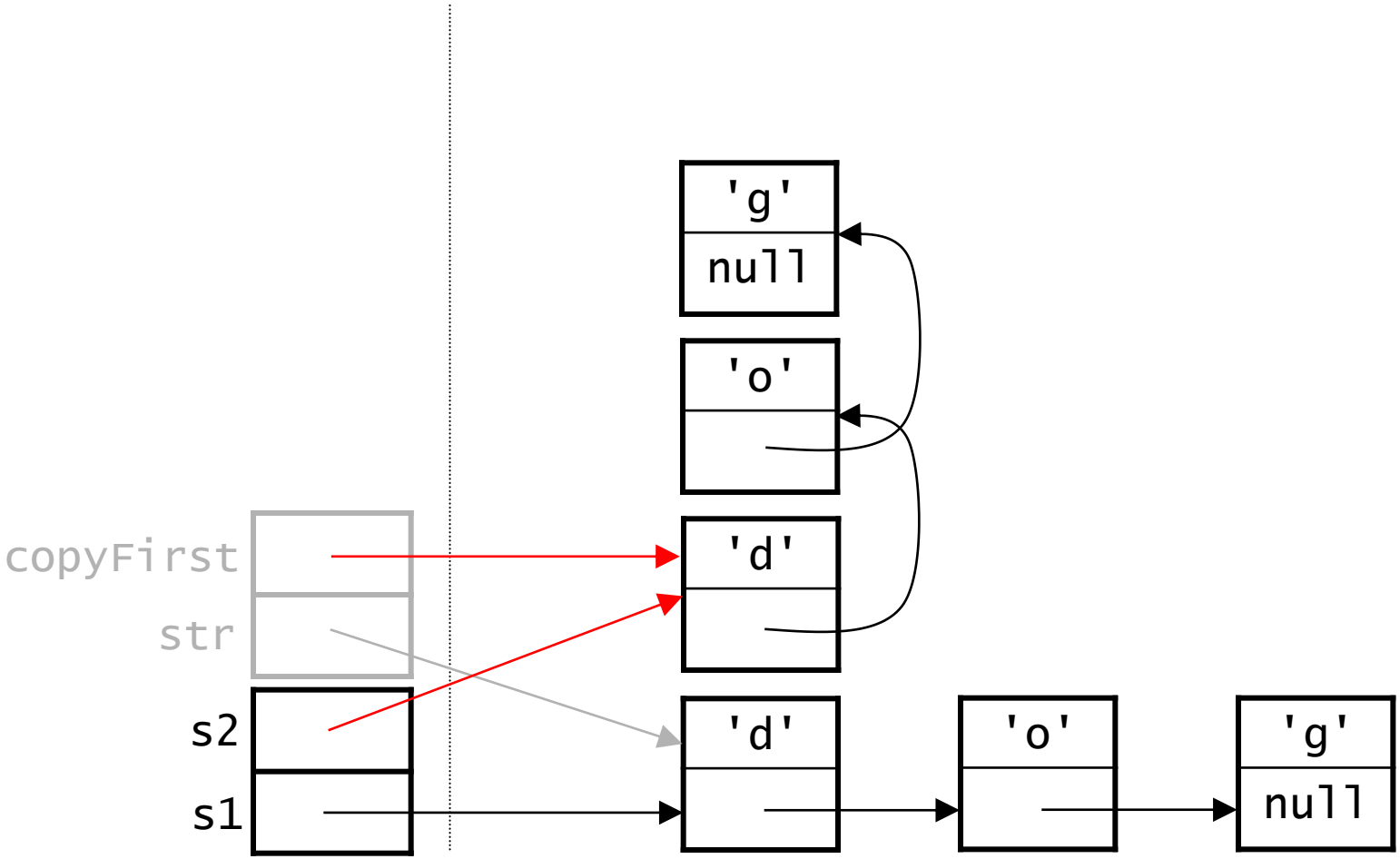
Tracing `copy()`: returning from the base case

- From a client: `StringNode s2 = StringNode.copy(s1);`



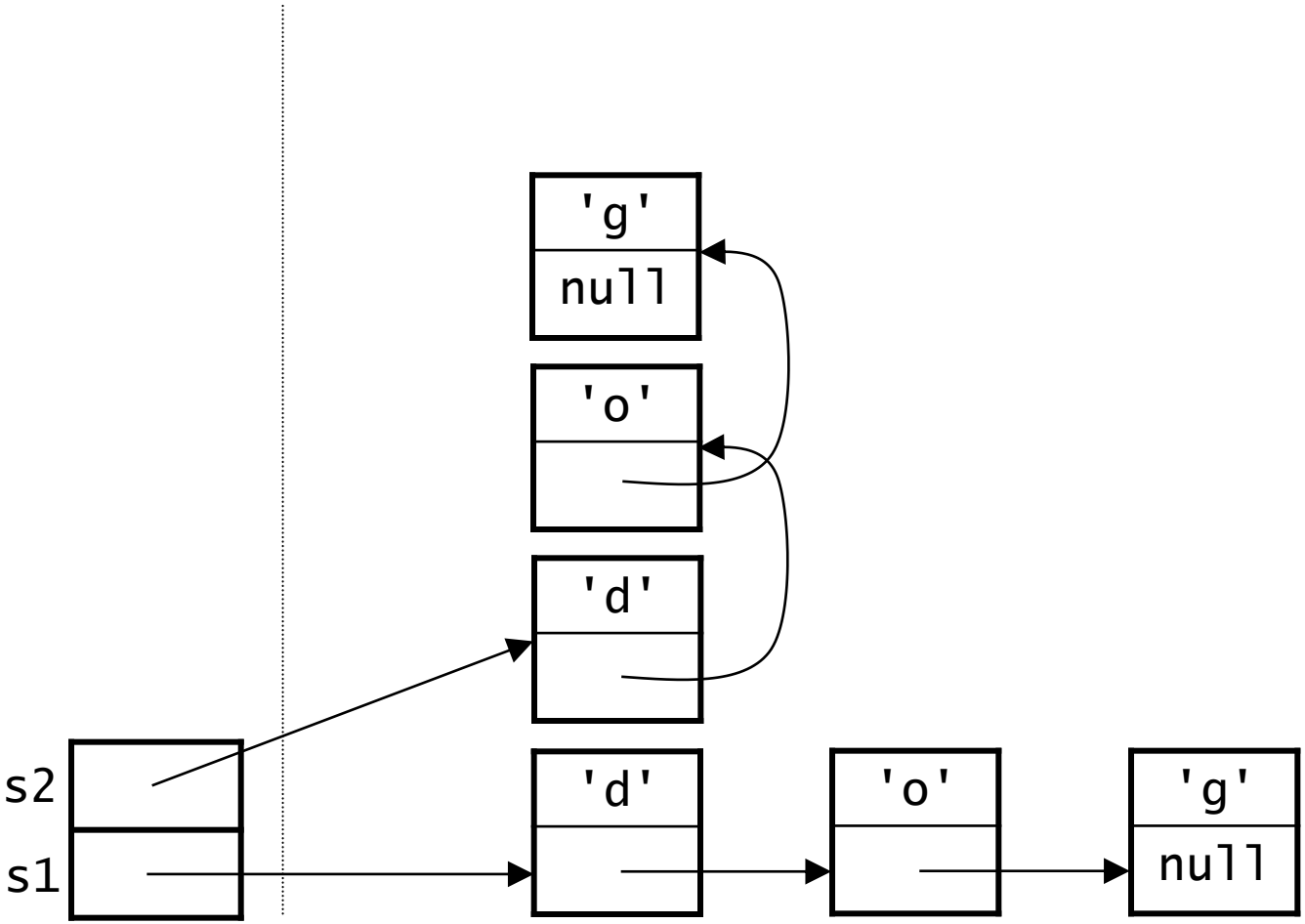
Tracing `copy()`: returning from the base case

- From a client: `StringNode s2 = StringNode.copy(s1);`



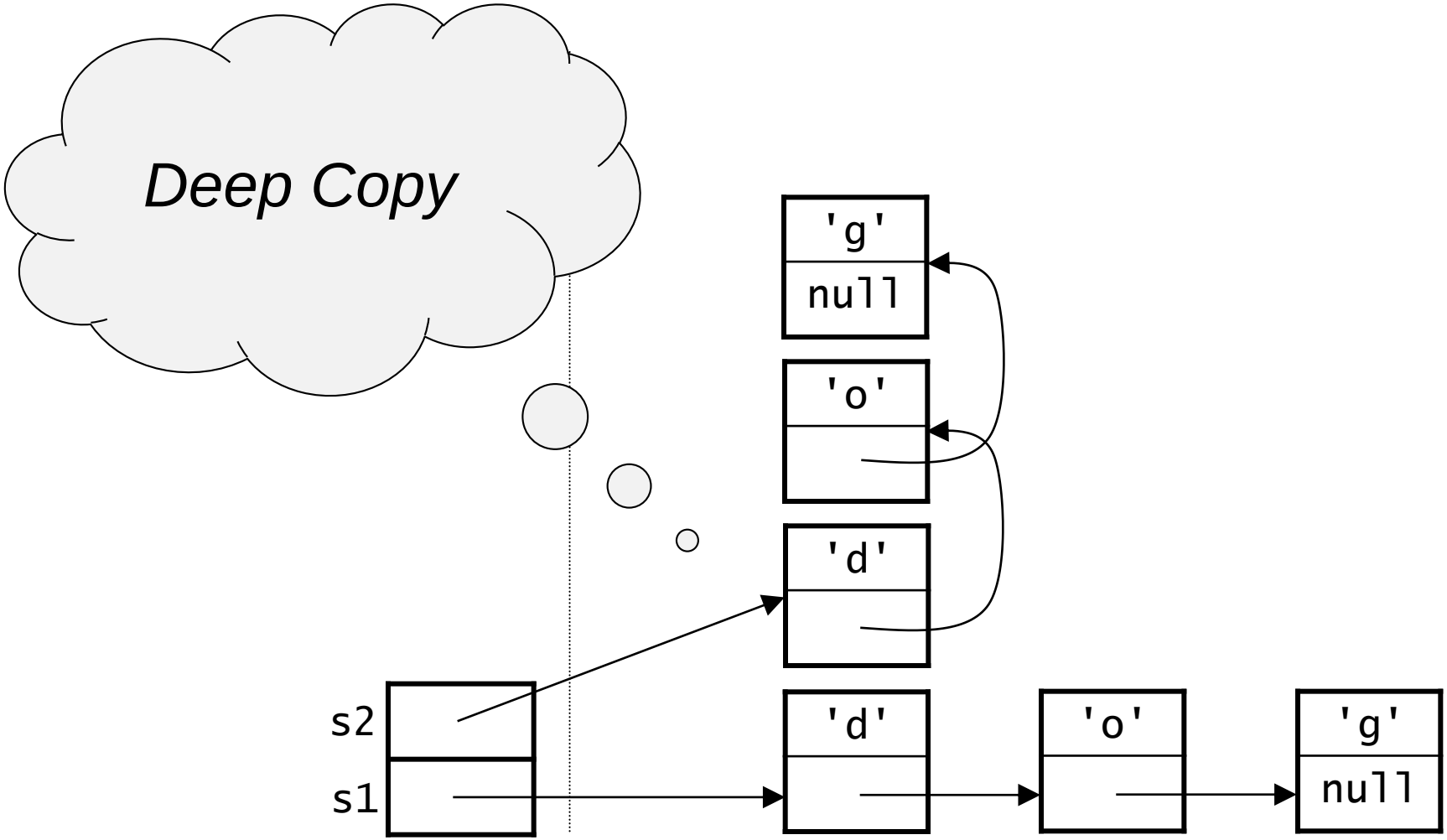
Tracing `copy()`: Final Result

- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.

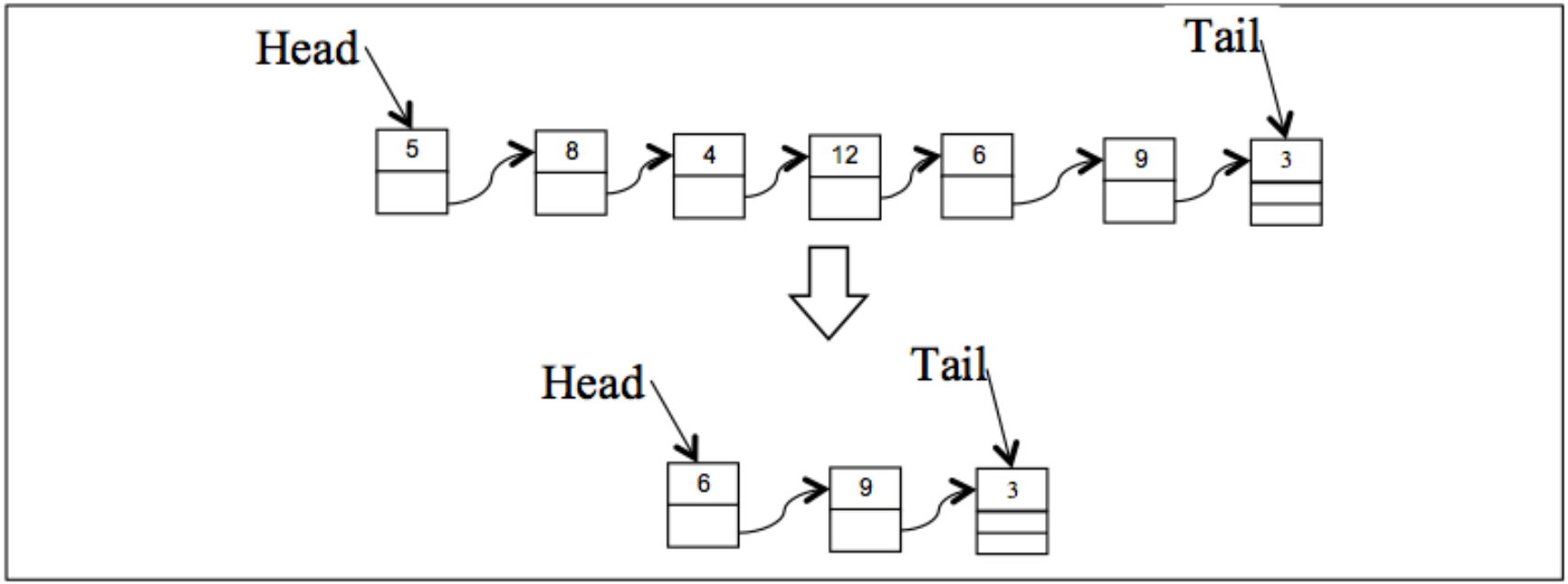


Tracing `copy()`: Final Result

- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.

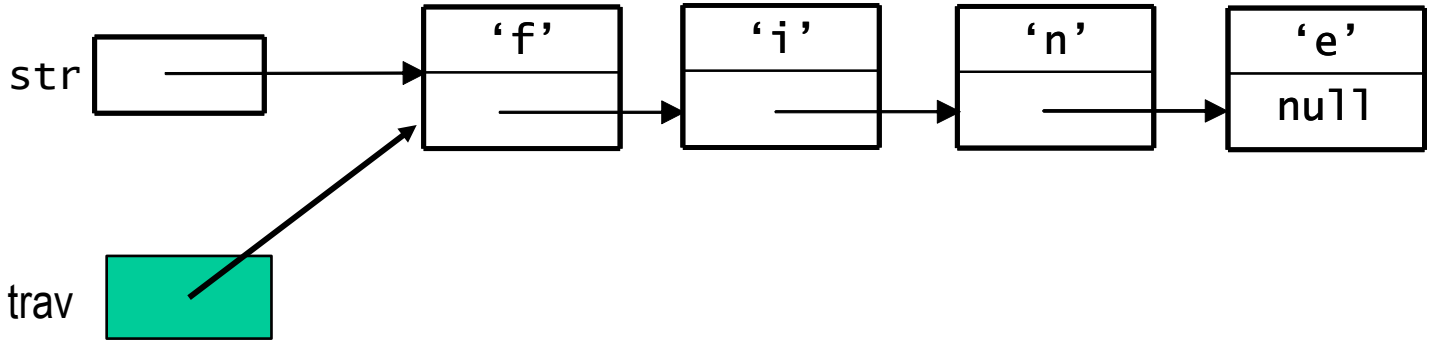


Traversing a List *iteratively*



Using Iteration to Traverse a Linked List

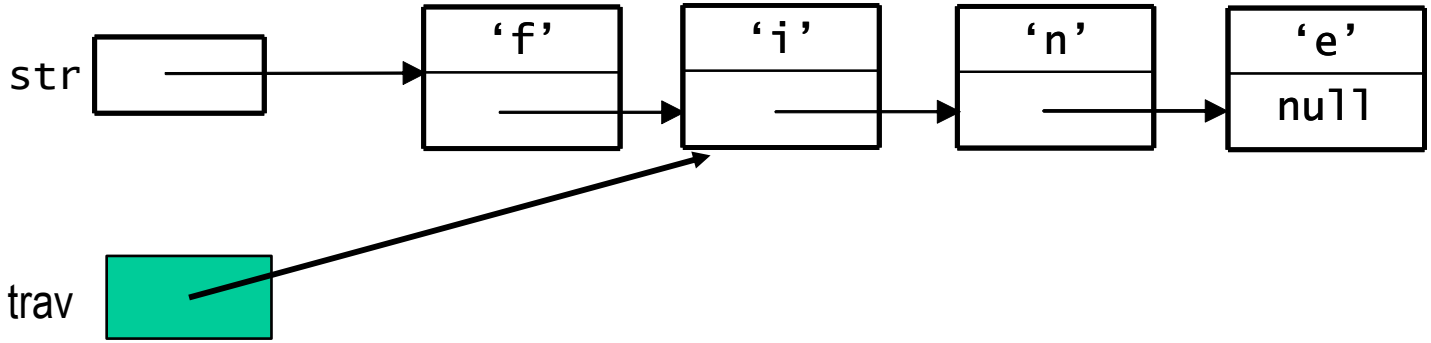
- Many tasks require us to traverse or "walk down" a linked list.
- We've already seen methods that use recursion to do this.
- Traversing a linked list is often done iteratively using a loop.



- Assign a new pointer to reference the first node in the list.

Using Iteration to Traverse a Linked List

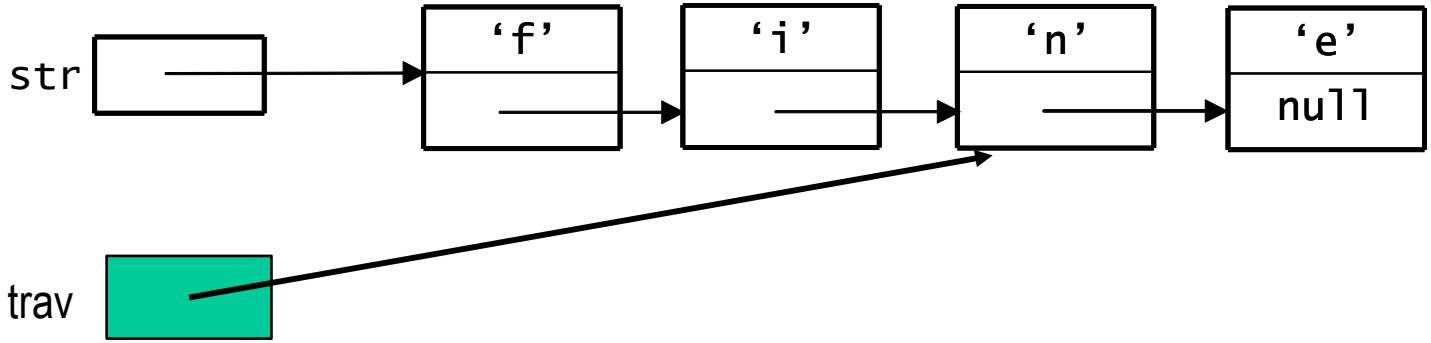
- Many tasks require us to traverse or "walk down" a linked list.
- We've already seen methods that use recursion to do this.
- Traversing a linked list is often done iteratively using a loop.



- Update the *trav* pointer to reference the next node n the list with each iteration of the loop!

Using Iteration to Traverse a Linked List

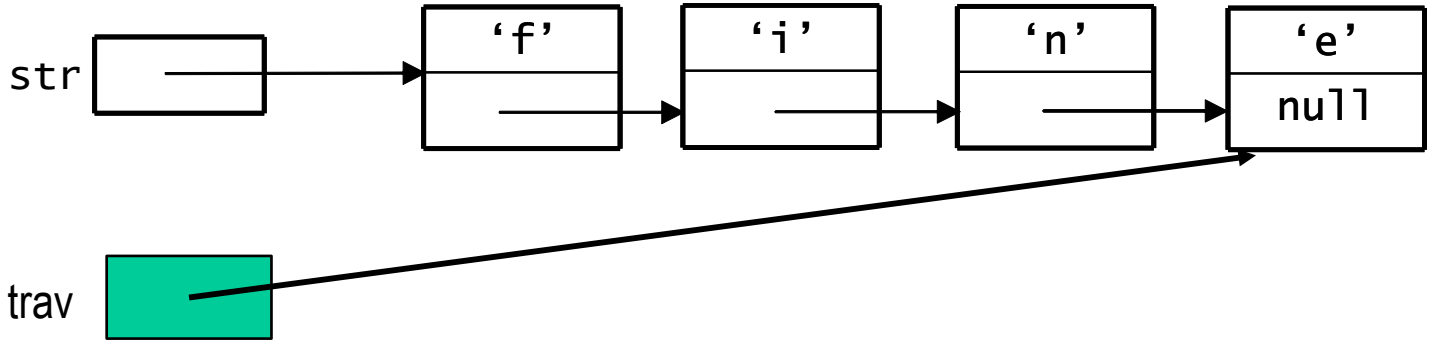
- Many tasks require us to traverse or "walk down" a linked list.
- We've already seen methods that use recursion to do this.
- Traversing a linked list is often done iteratively using a loop.



- Update the *trav* pointer to reference the next node n the list with each iteration of the loop!

Using Iteration to Traverse a Linked List

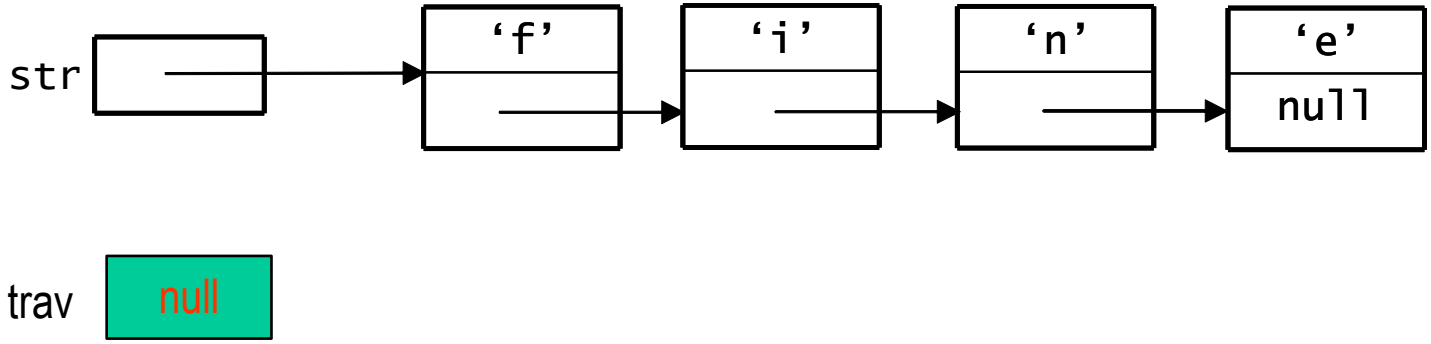
- Many tasks require us to traverse or "walk down" a linked list.
- We've already seen methods that use recursion to do this.
- Traversing a linked list is often done iteratively using a loop.



- Update the *trav* pointer to reference the next node n the list with each iteration of the loop!

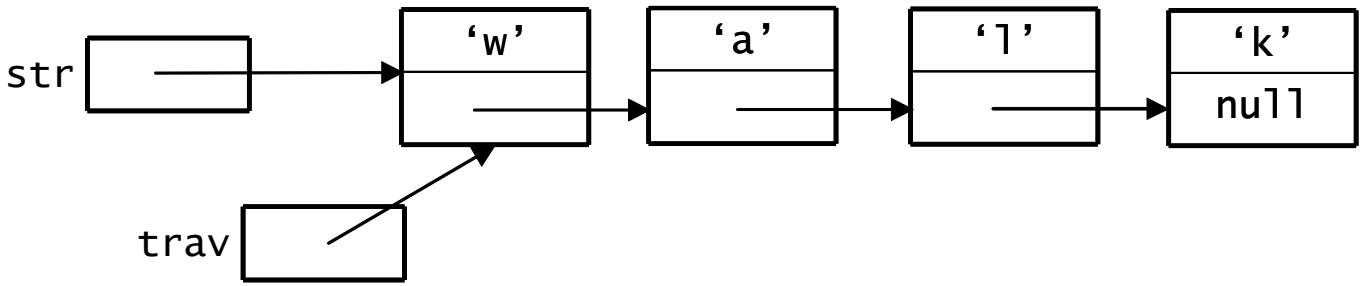
Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or "walk down" a linked list.
- We've already seen methods that use recursion to do this.
- Traversing a linked list is often done iteratively using a loop.



- *Depending on the objective of the traversal*, the loop will stop when the value of *trav* is null, or when it is referencing the node you are looking for!

Using Iteration to Traverse a Linked List

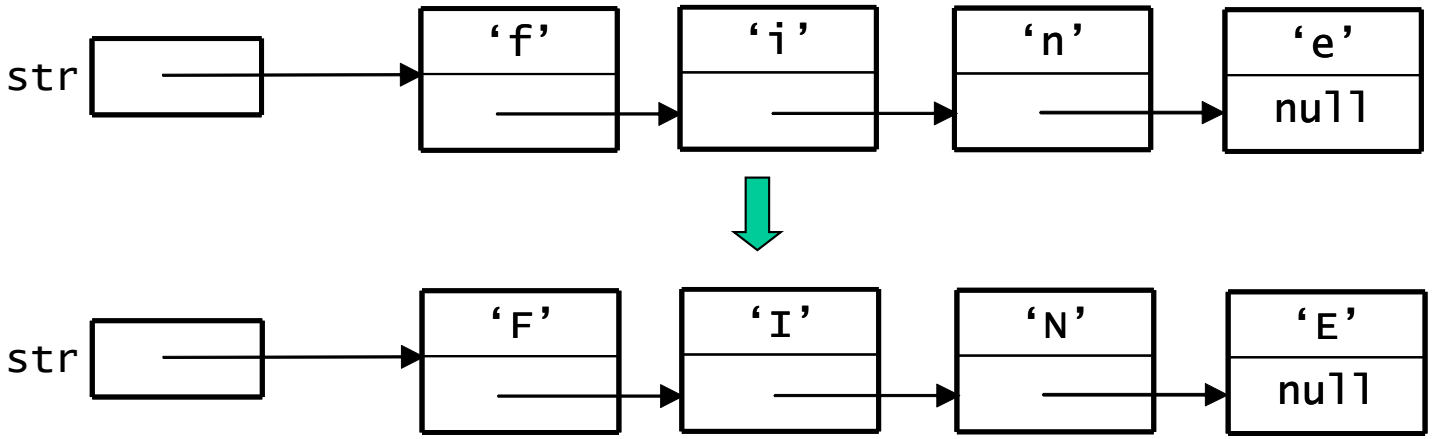


- General template for traversing **all** the nodes in a linked list:

```
// assign to some traversal variable (e.g trav) the
// reference to the head of the list
StringNode trav = str;   // use a temporary reference

while (trav != null) {

   // usually do something here

   trav = trav.next;   // move trav down one node
}
```

Example of Iterative Traversal

- `toUpperCase(str)`: convert `str` to all upper-case letters

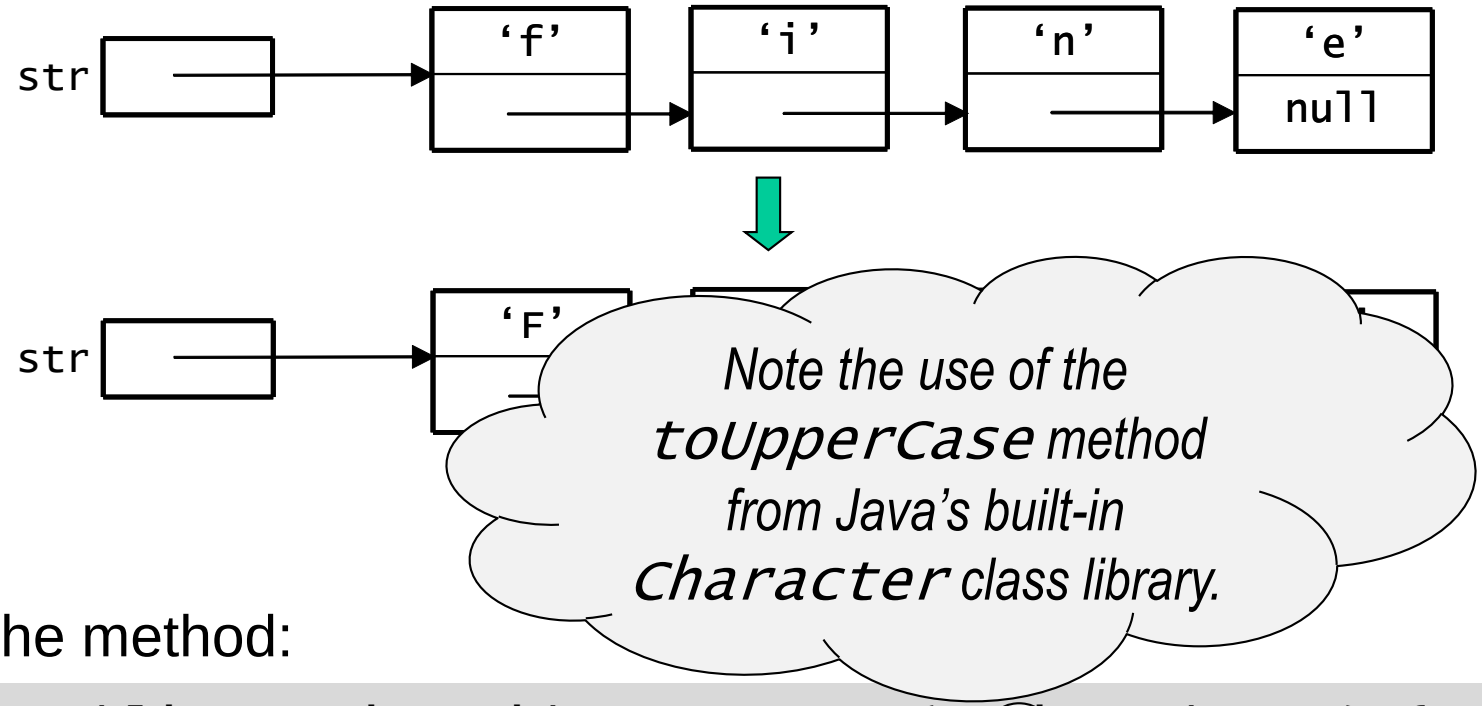


- The method:

```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

Example of Iterative Traversal

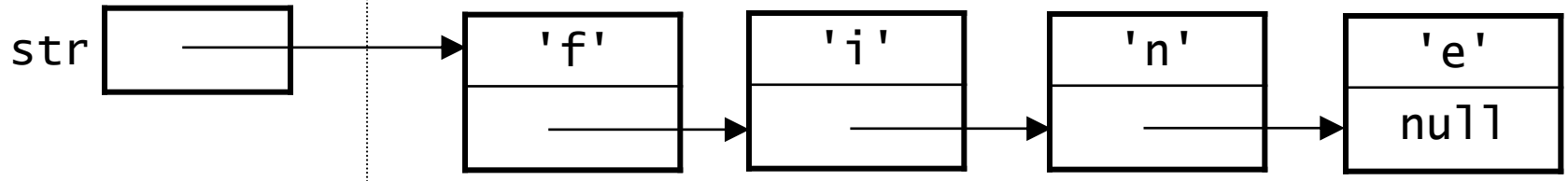
- `toUpperCase(str)`: convert `str` to all upper-case letters



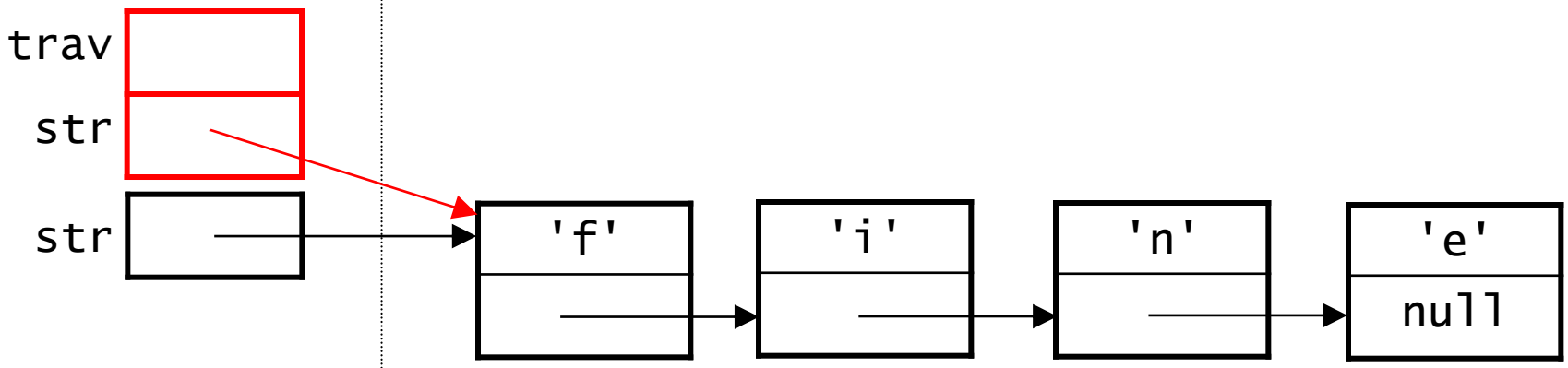
- The method:

```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

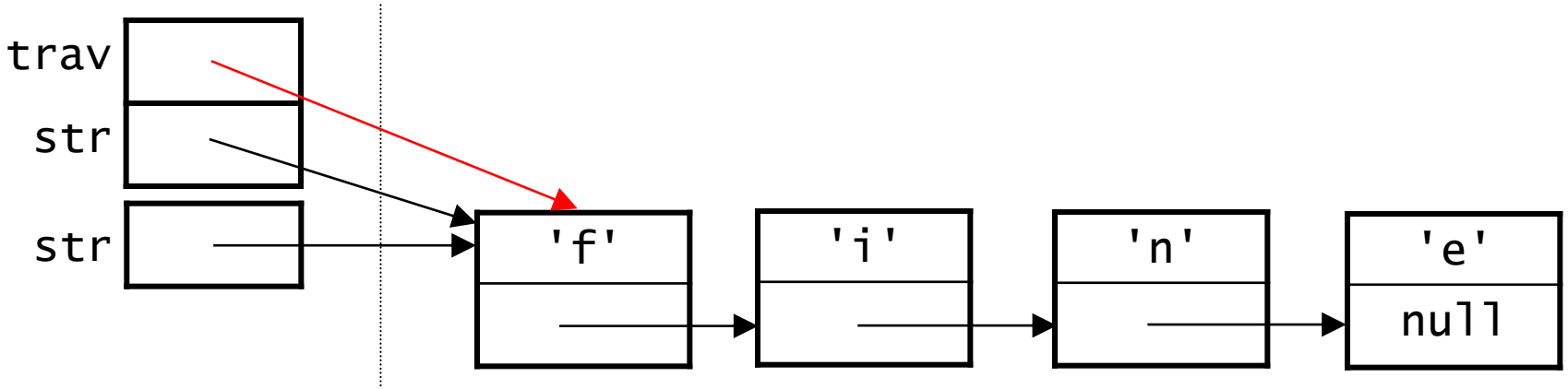
Tracing `toUpperCase()`



Calling `StringNode.toUpperCase(str)` adds a stack frame to the stack:

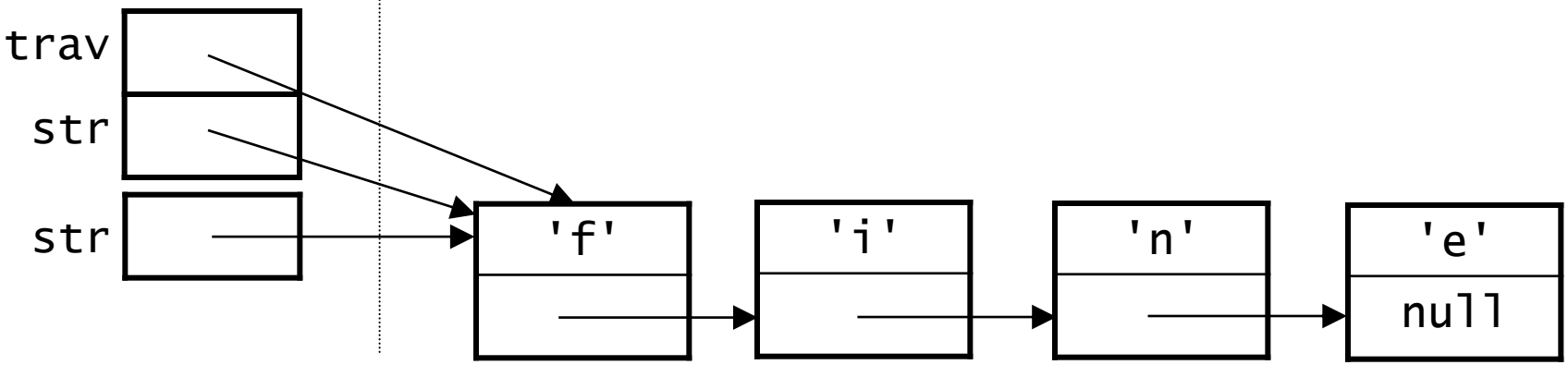



```
StringNode trav = str;
```



Tracing `toUpperCase()`

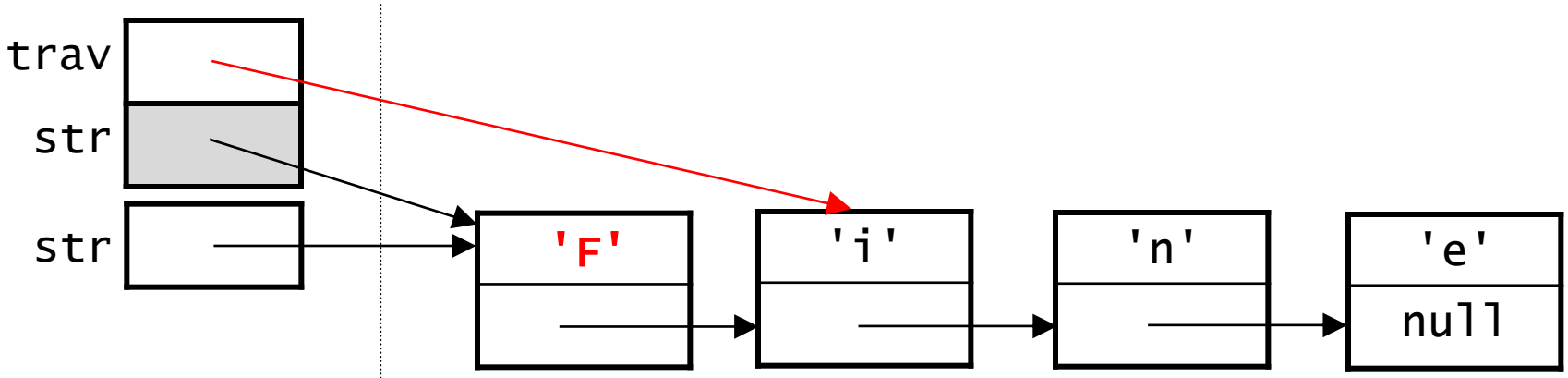
from the previous page:



we enter the `while` loop:

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

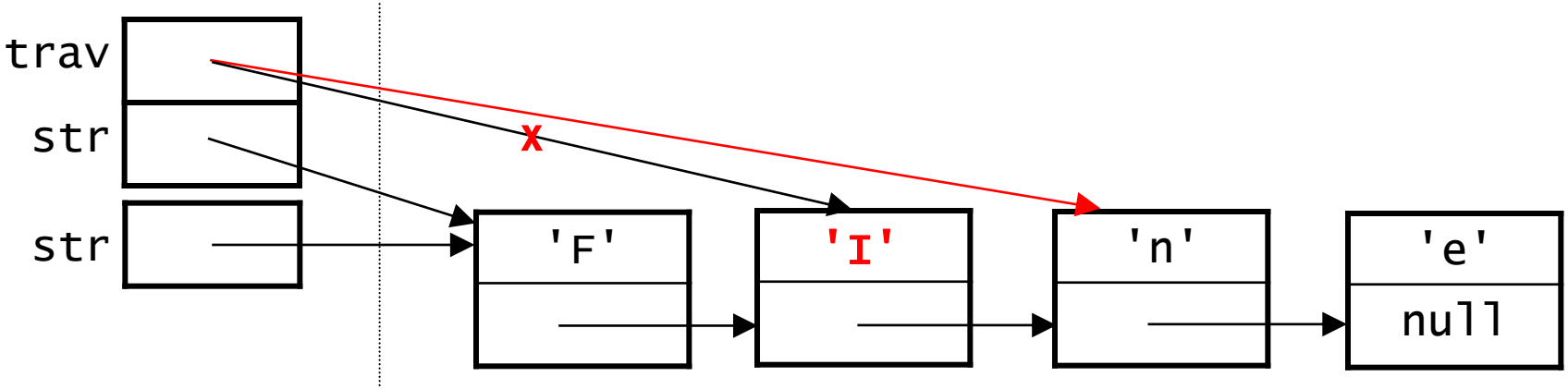
results of the first pass through the loop:



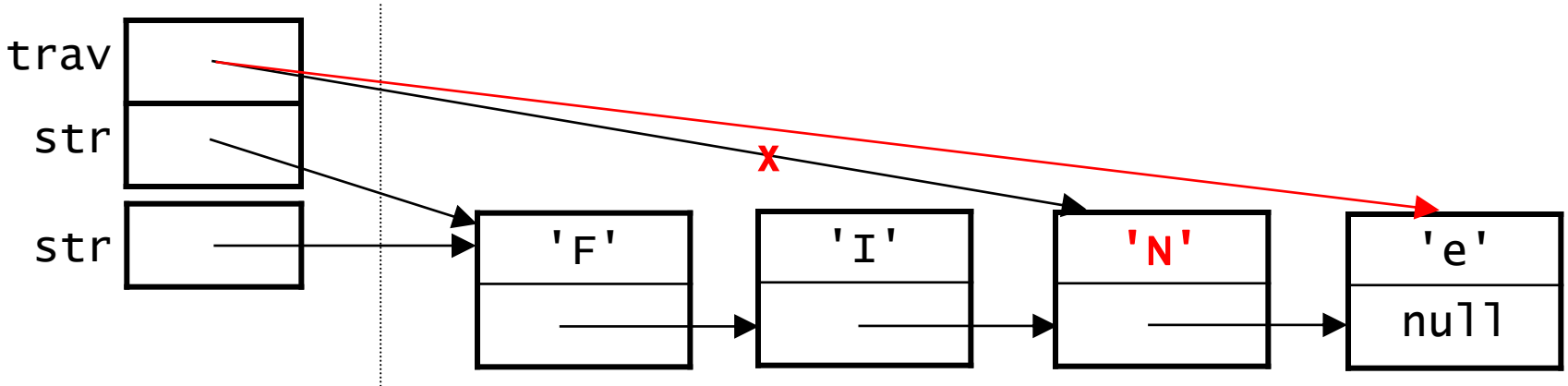
Tracing `toUpperCase()`

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the second pass through the loop:



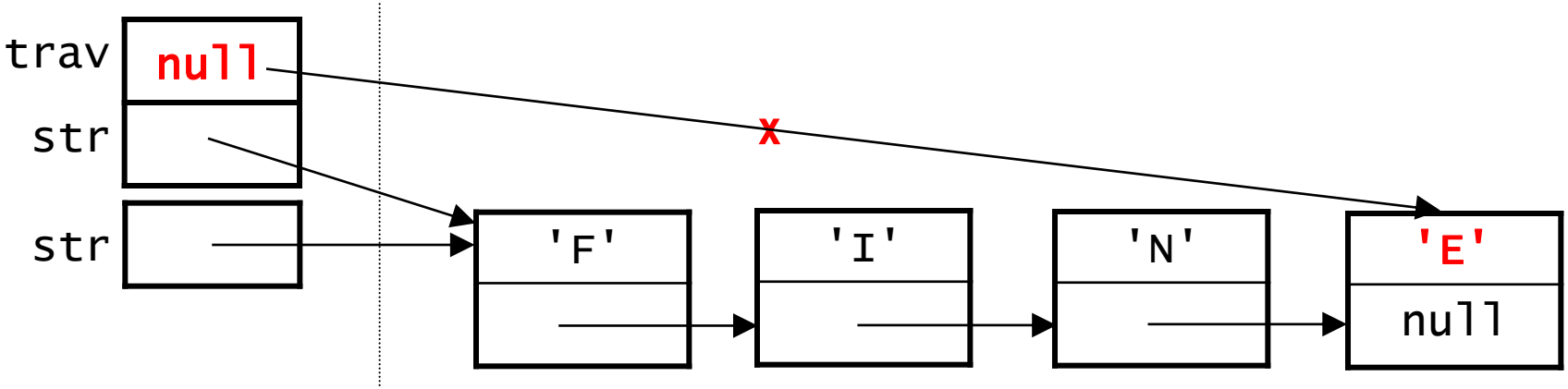
results of the third pass:



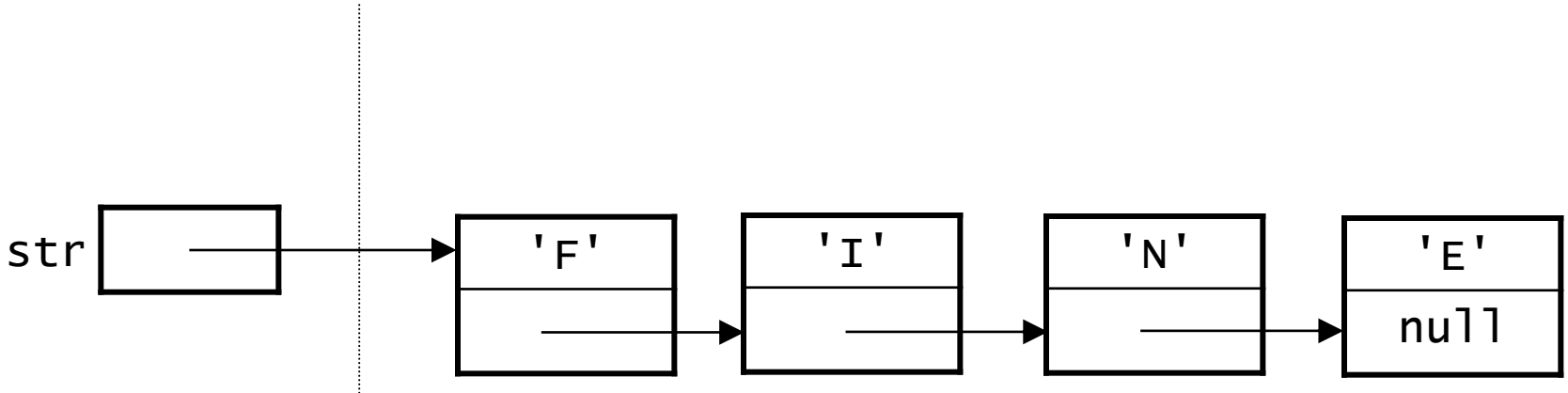
Tracing `toUpperCase()`

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the fourth pass through the loop:



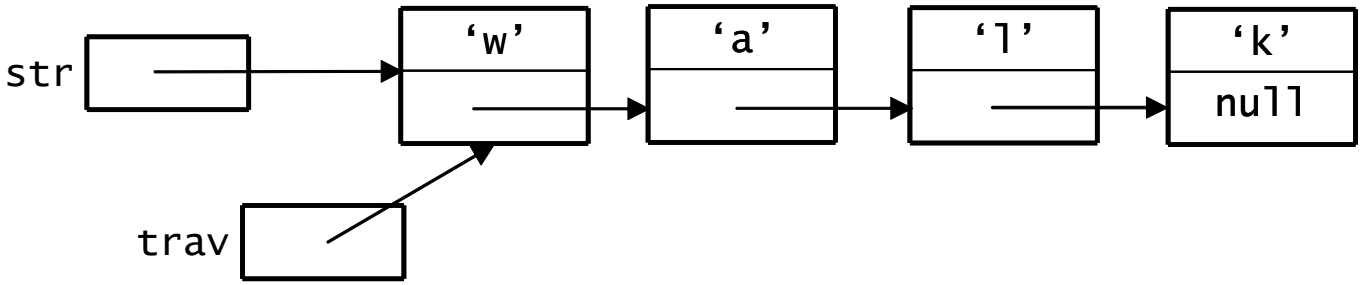
and now `trav == null`, so we break out of the loop and return:



Using Iteration to Traverse a Linked List:

summary

- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.



- Template for traversing an entire linked list:

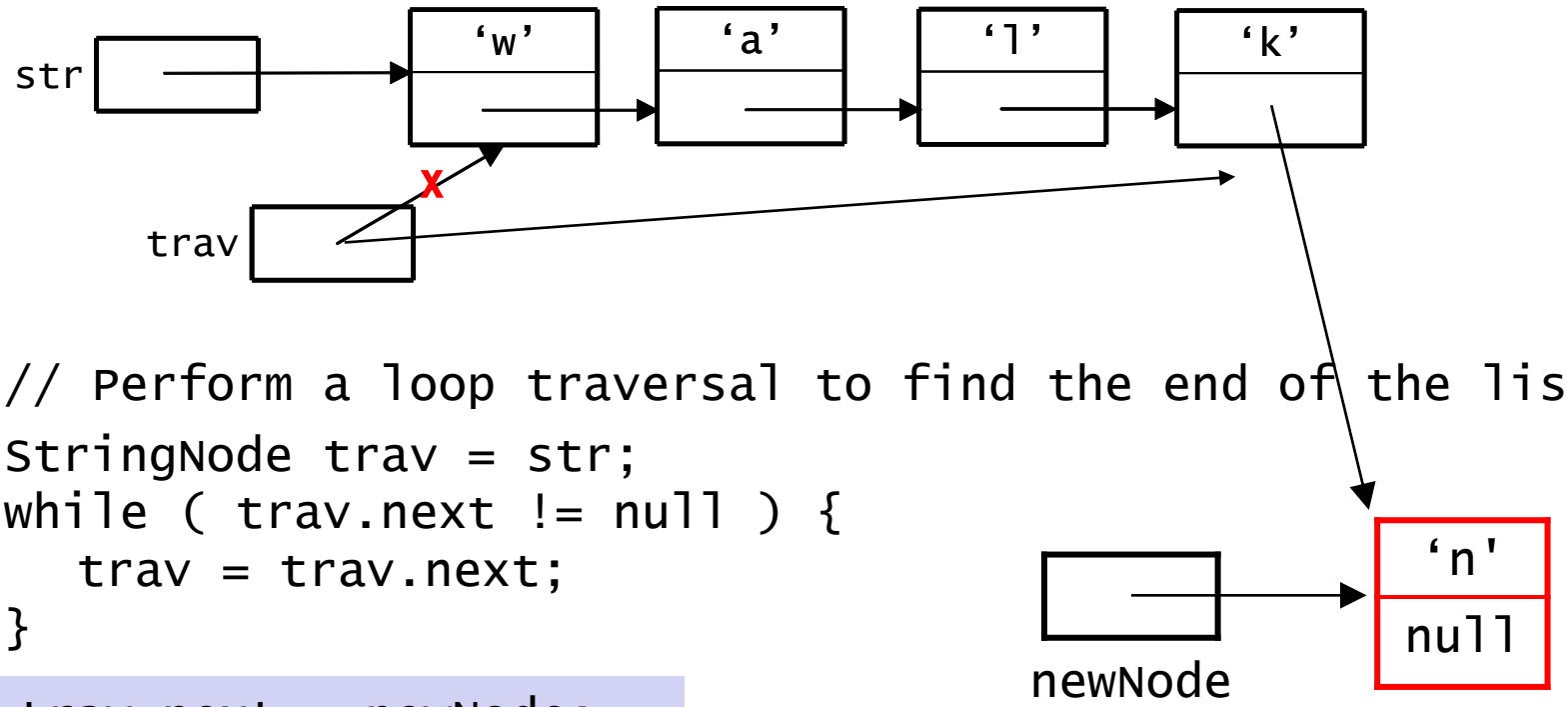
```
StringNode trav = str;
while (trav != null) {
    // do something here
    trav = trav.next;
}
```

Method to **append** a newNode to a single linked list

- `append(str, newNode)` – a private method that appends the newNode to the list referenced by str.

```
private static StringNode append(StringNode str,
  StringNode newNode)
{

}
```

Append *newNode* to the end of a single linked list…




```
// Perform a loop traversal to find the end of the list
StringNode trav = str;
while ( trav.next != null ) {
   trav = trav.next;
}
```

```
trav.next = newNode;
```

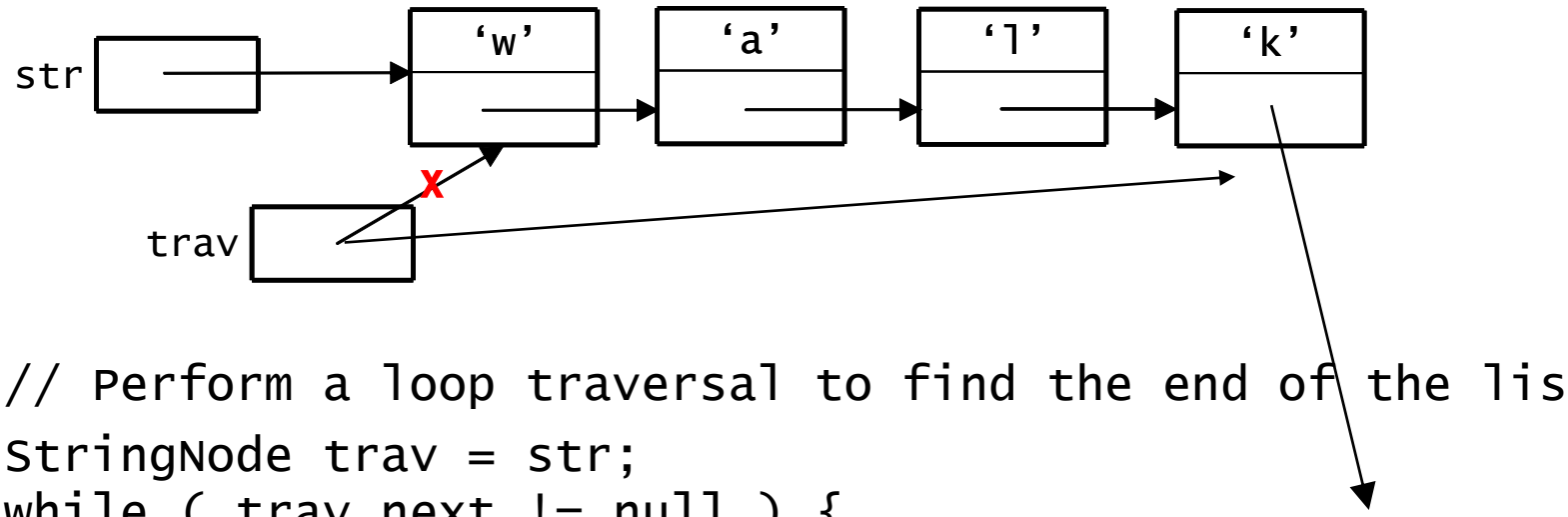
A. trav != null

B. trav.next != null

C. str != trav

D. none of the above

Append *newNode* to the end of a single linked list…




```
// Perform a loop traversal to find the end of the list
StringNode trav = str;
while ( trav.next != null ) {
   trav = trav.next;
}
```

```
trav.next = newNode;
```

What would happen if the list was empty?

A. trav != null

B. trav.next != null

C. str != trav

D. none of the above

Method to **append** a newNode to a single linked list

- `append(str, newNode)` – a private method that appends the newNode to the list referenced by str.

```
private static StringNode append(StringNode str,
  StringNode newNode)
{
    StringNode trav = str;    // assign traversal reference

    if ( trav ) {
        while (trav.next != null ) // stop at last node
            trav = str.next;

        trav.next = newNode;
    } else
        str = newNode;       // assign it as first node

    return( str );    // return reference to first node
}
```

Finding the Length of a String (list):

an iterative method

- `length(str)` – a private method that returns the length of the list.
- Iterative approach:

```
private static int length(StringNode str) {

}
```

Finding the Length of a String (list):

an iterative method

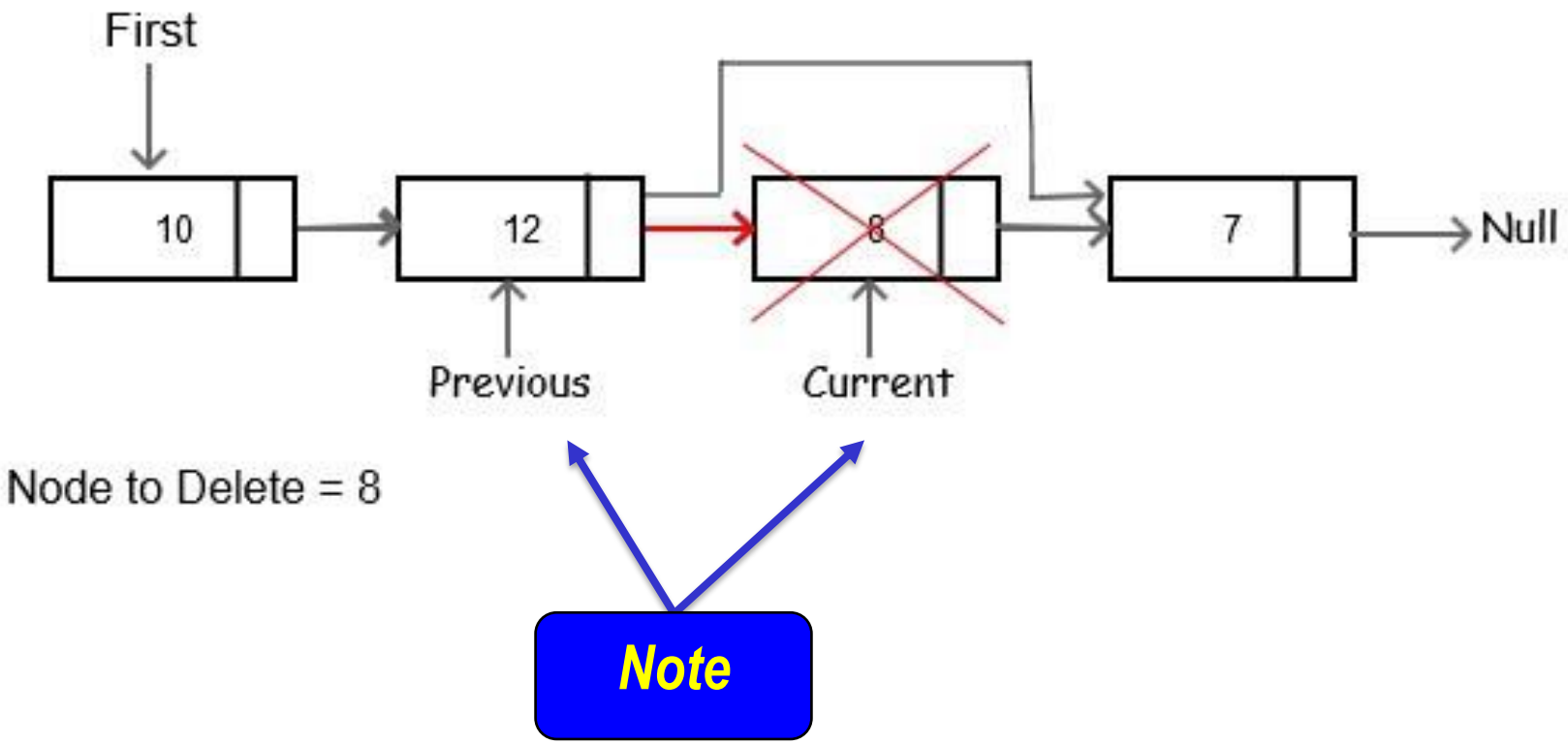
- `length(str)` – a private method that returns the length of the list.
- Iterative approach:

```
private static int length(StringNode str) {

    int counter = 0;

    while ( str != null ) {
        counter++;
        str = str.next
    }

    return( counter );

}
```

Note the while loop!

Inserting and Deleting

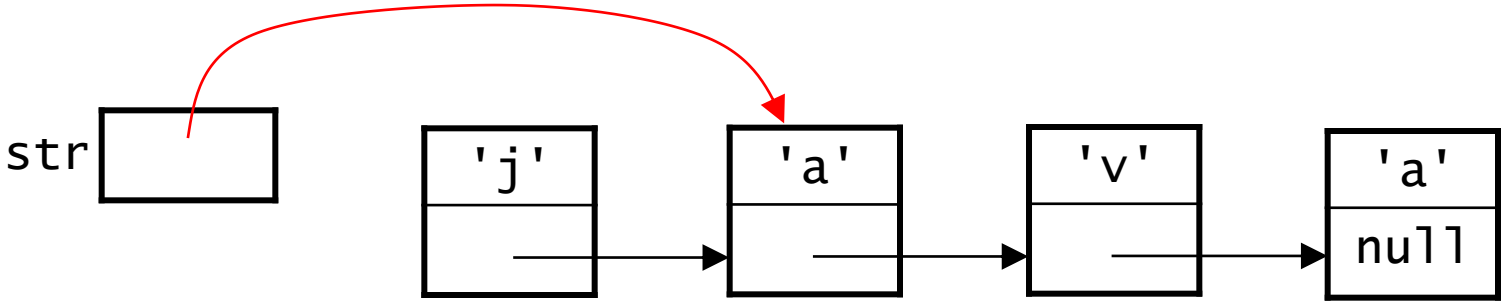
by list traversal



Deleting the Item at Position i

- <u>Special case:</u> `i == 0` (deleting the first item)
- Update our reference to the first node:

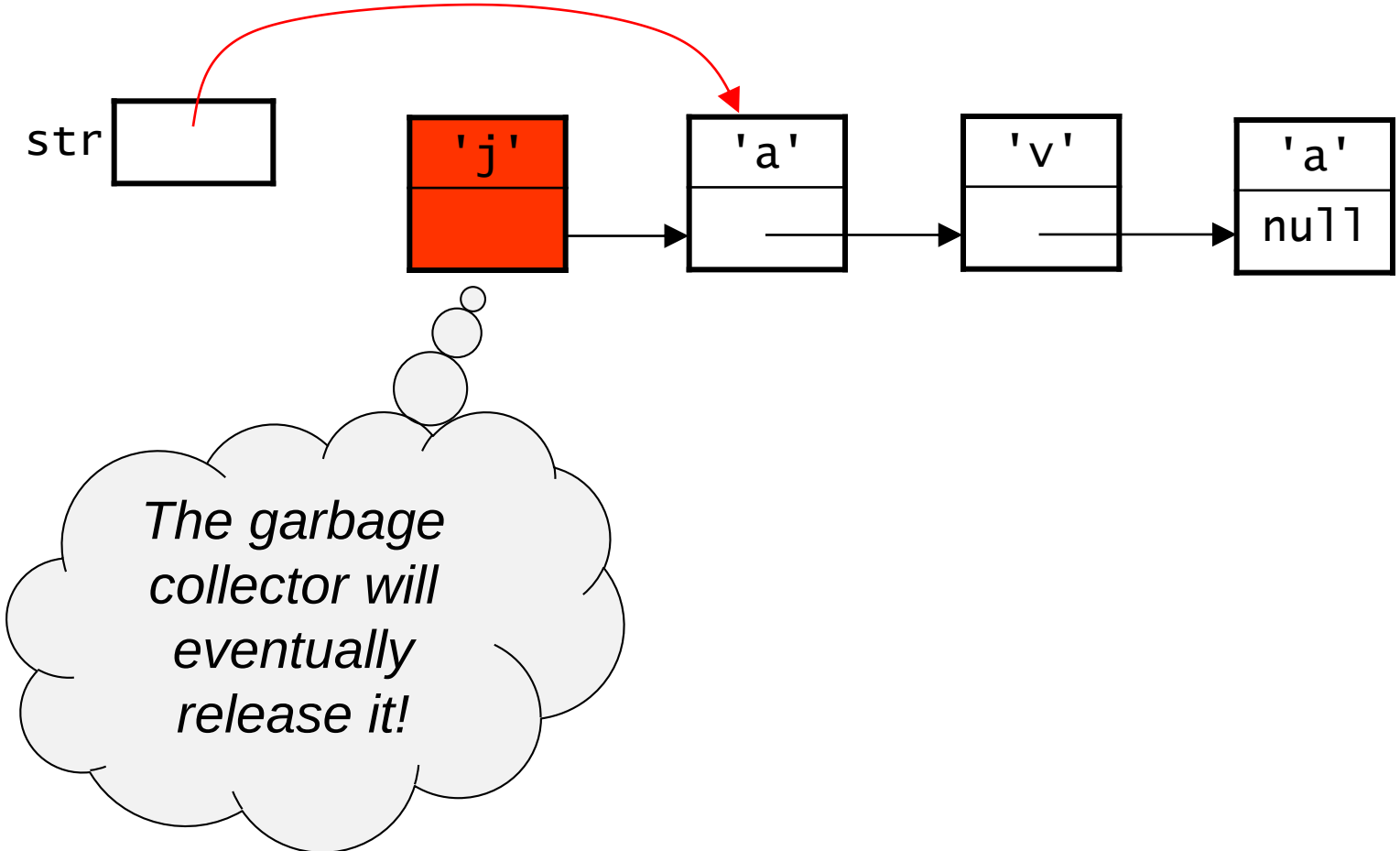
```
str = str.next;
```



Deleting the Item at Position i

- <u>Special case:</u> `i == 0` (deleting the first item)
- Update our reference to the first node:

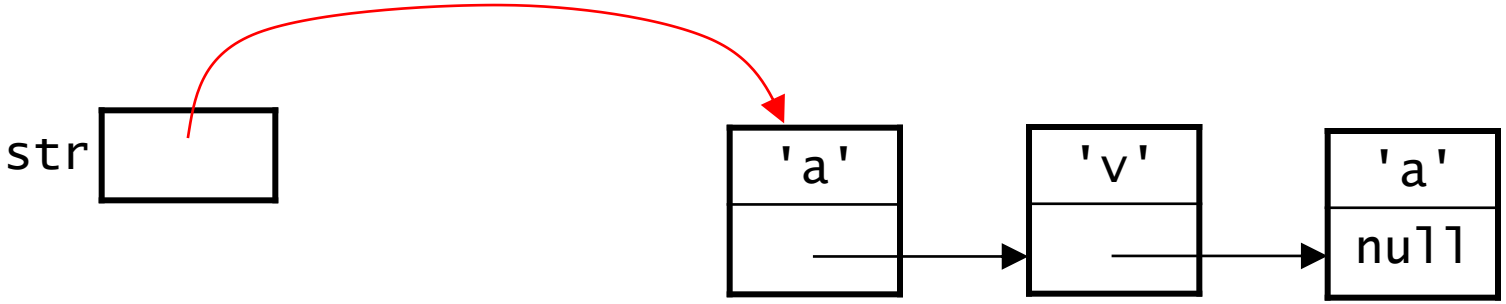
```
str = str.next;
```



Deleting the Item at Position i

- <u>Special case:</u> `i == 0` (deleting the first item)
- Update our reference to the first node:

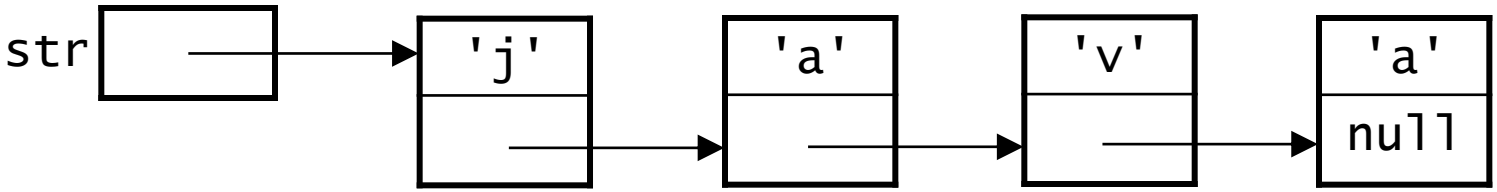
```
str = str.next;
```



Deleting the Item at Position i (cont.)

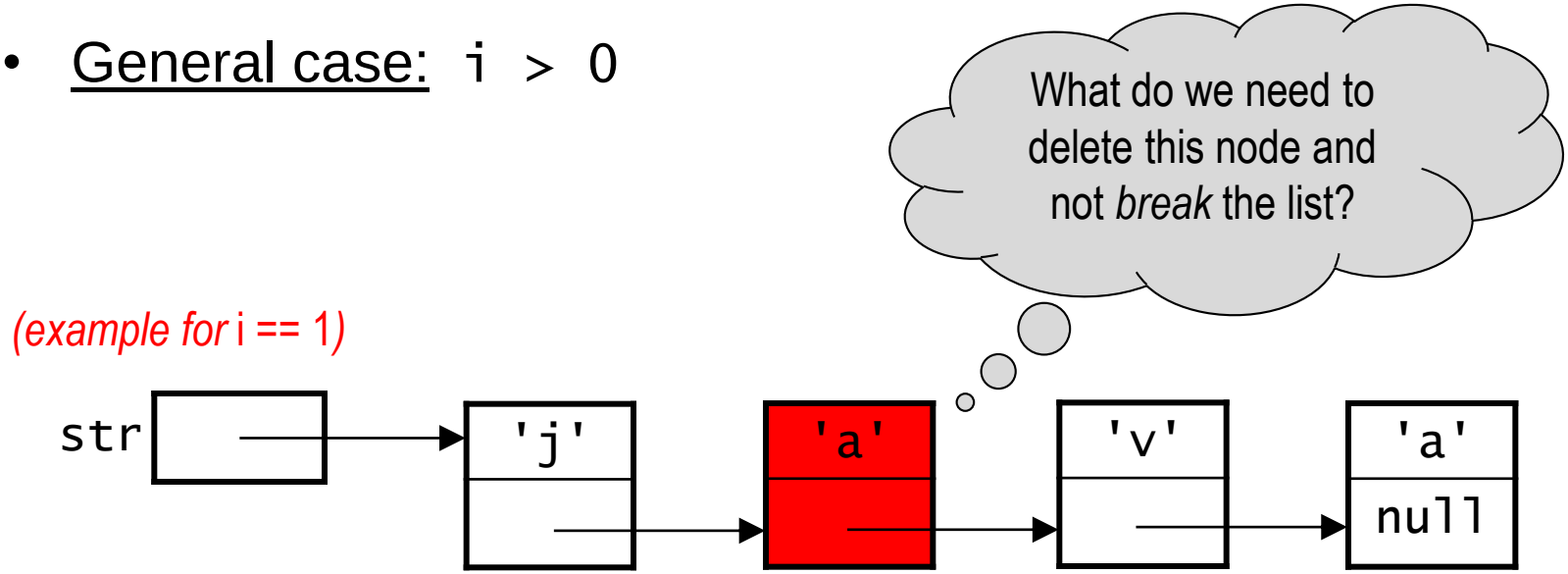
- <u>General case:</u> `i > 0`

(example for i == 1*)*



Deleting the Item at Position i (cont.)

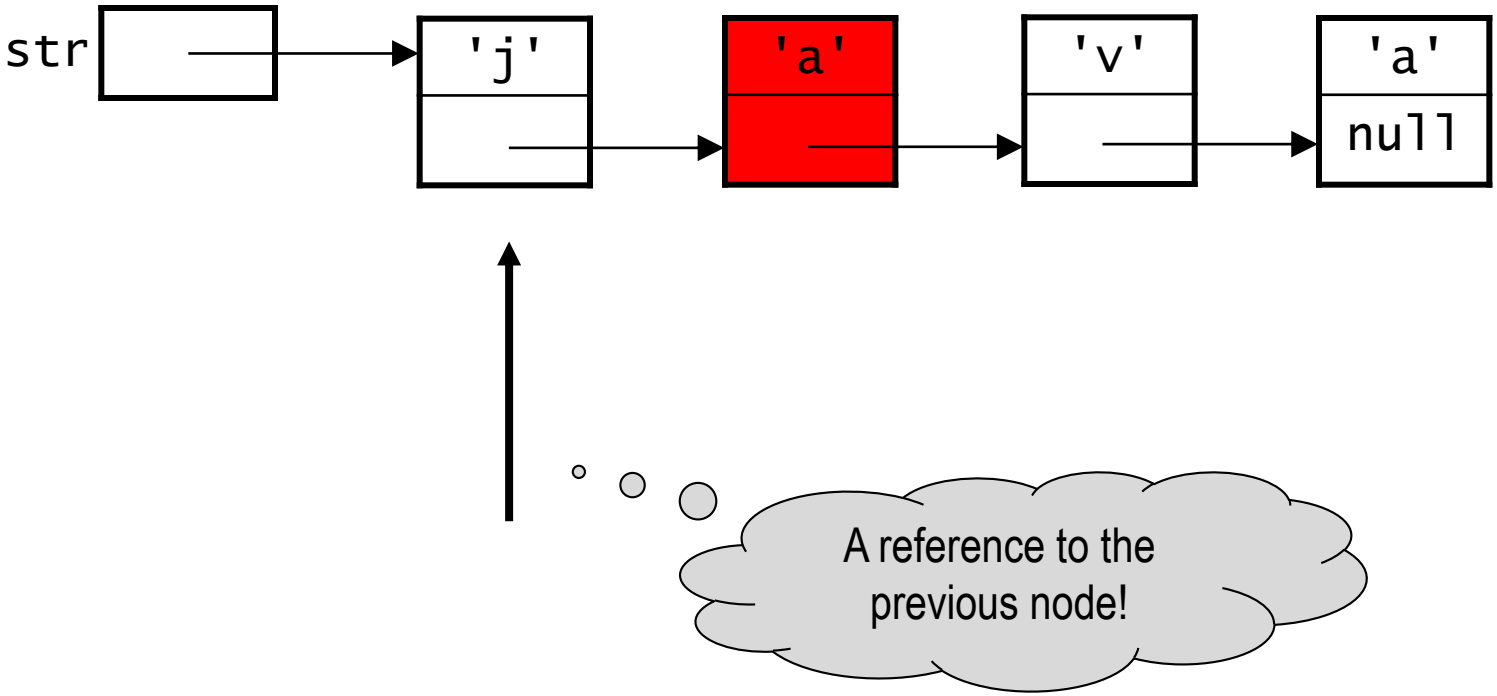
- <u>General case:</u> `i > 0`



Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`

(example for i == 1*)*

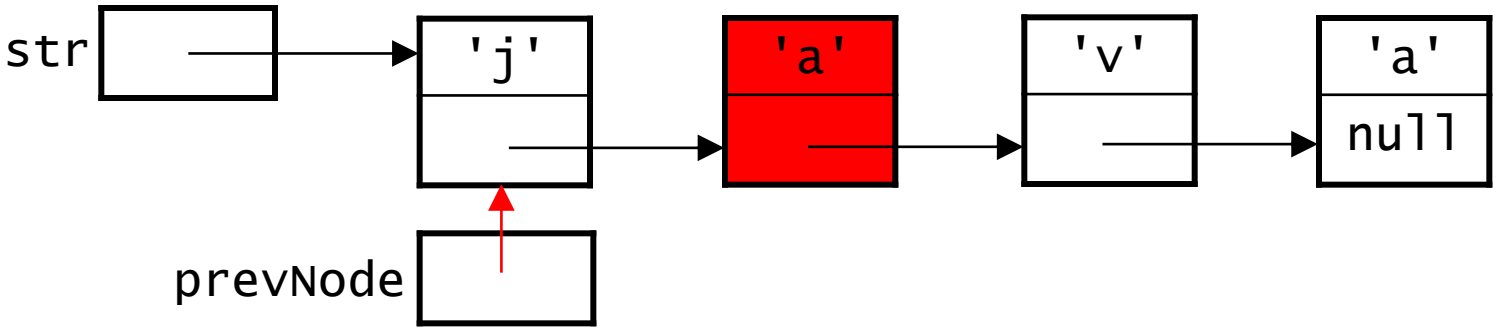


Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`
- First obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i - 1);
```

(example for i == 1*)*

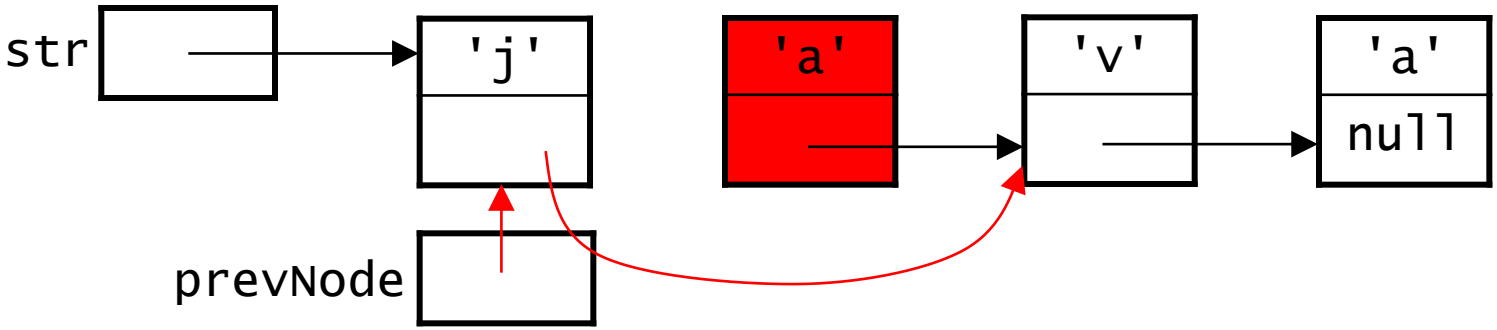


Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`
- First obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i - 1);
```

(example for i == 1*)*



- `prevNode.next = prevNode.next.next;`

An alternative to using multiple forward references is…

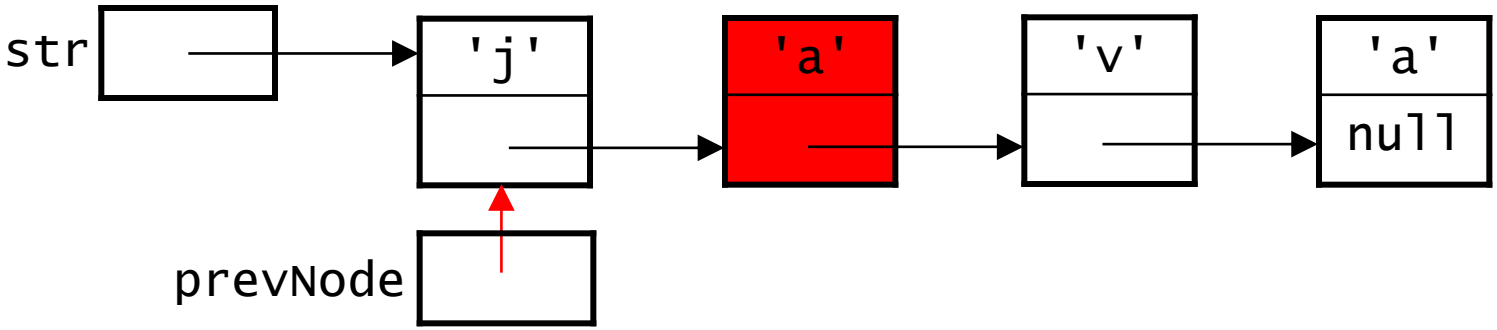
Deleting the Item at Position i

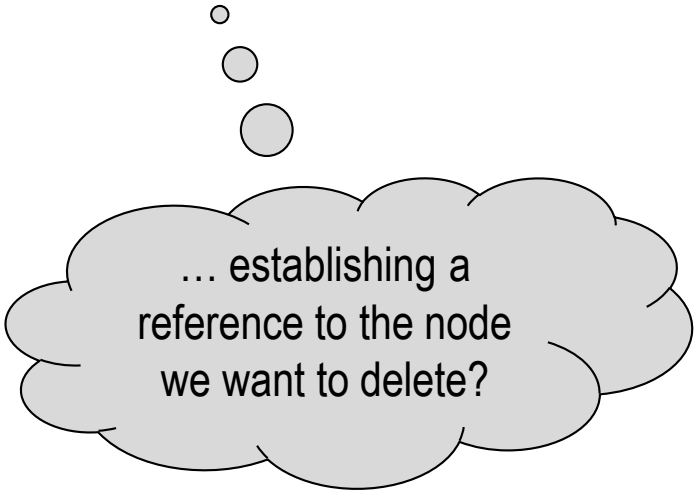
(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*





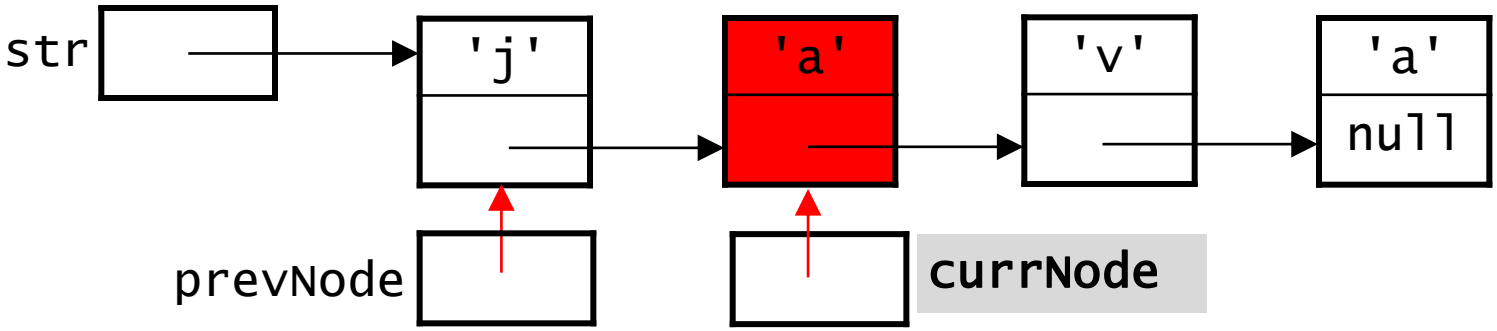
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



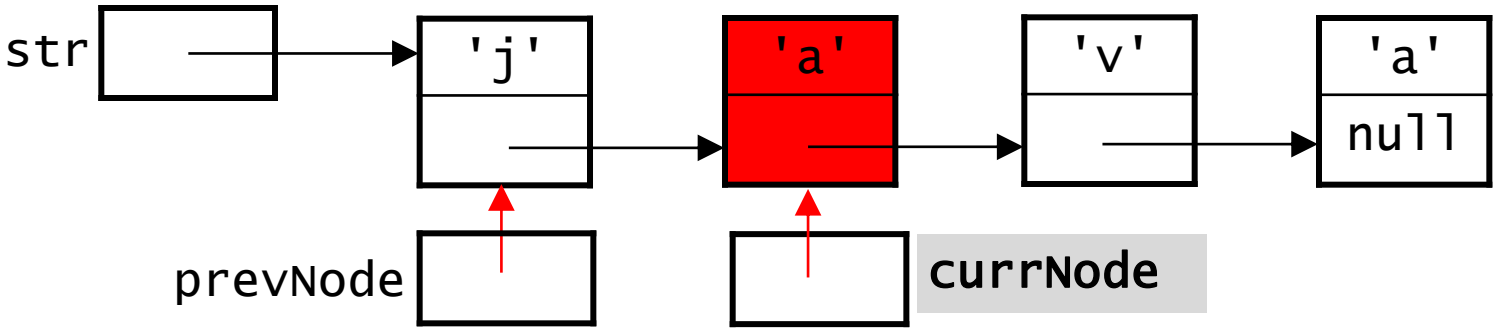
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



- `What assignment statement can be used now?`

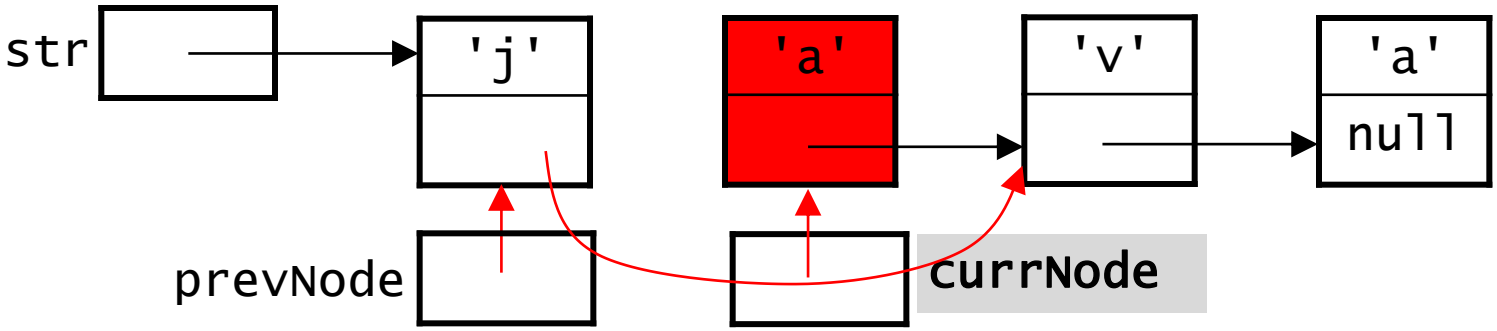
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



- `prevNode.next = currNode.next;`

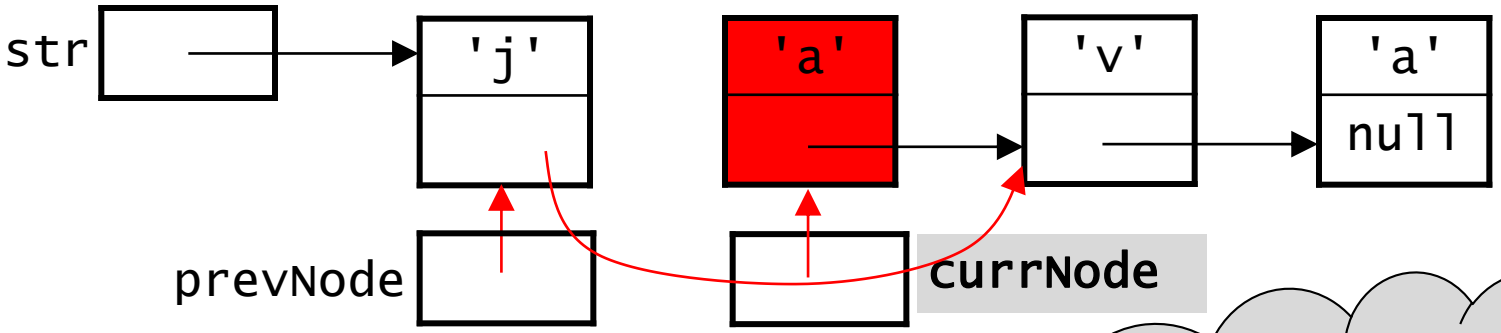
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



- `prevNode.next = currNode.next;`

Note that to establish the two references, prevNode and currNode we called the method, `getNode` twice:

`getNode(i-1)`
`getNode(i)`

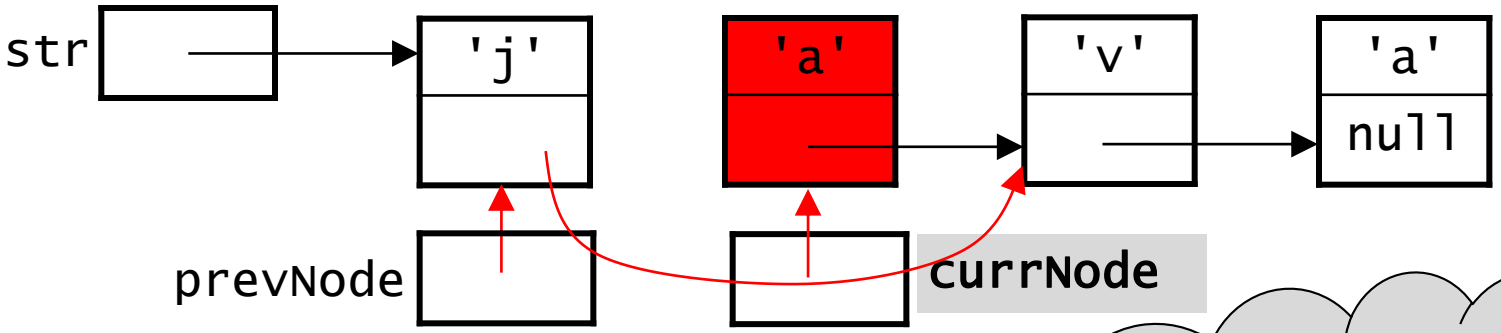
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*

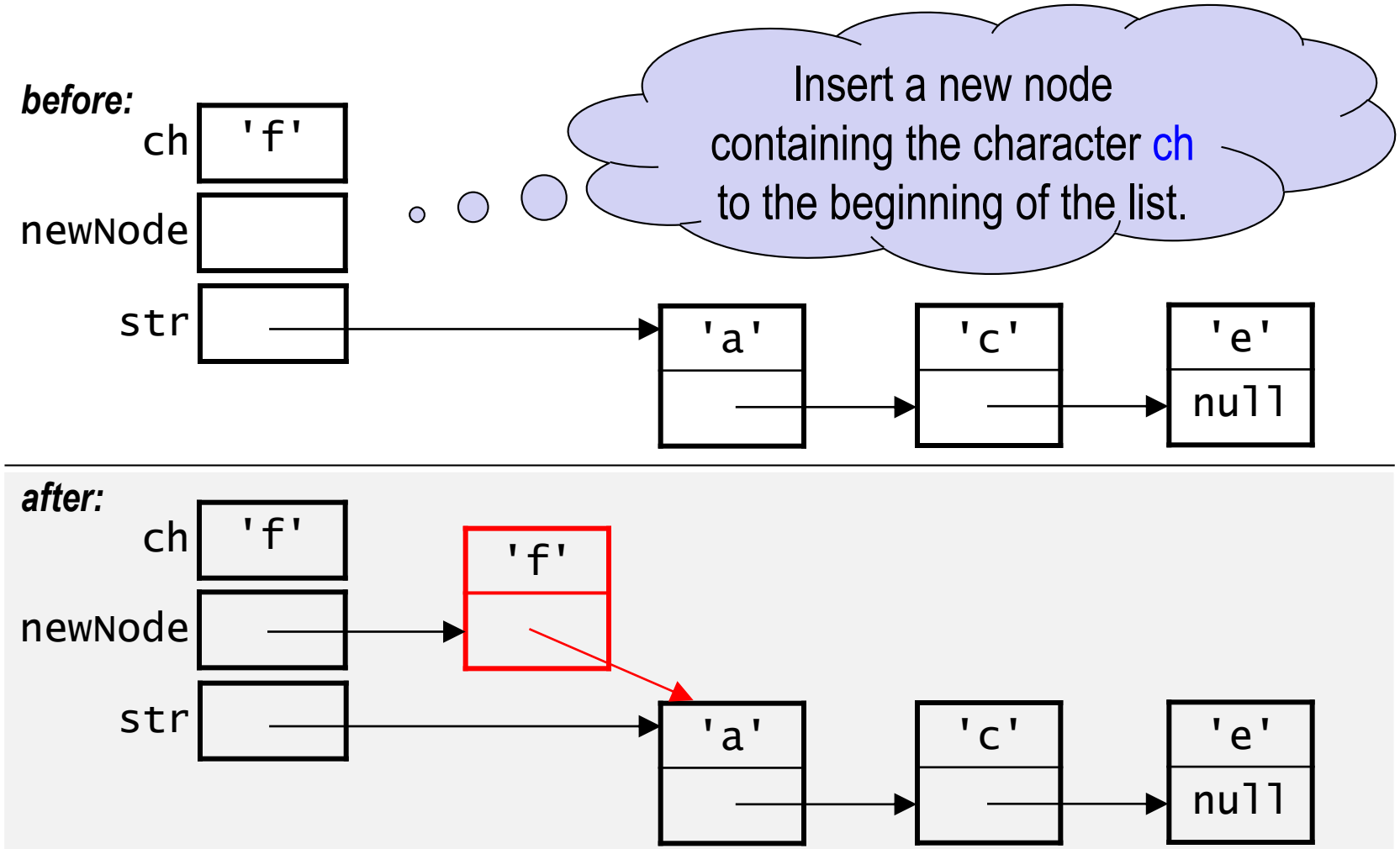


- `prevNode.next = currNode.next;`

To more efficiently establish both references, we can *set the references within the same traversal!* More on this to come…

Inserting an Item at Position i

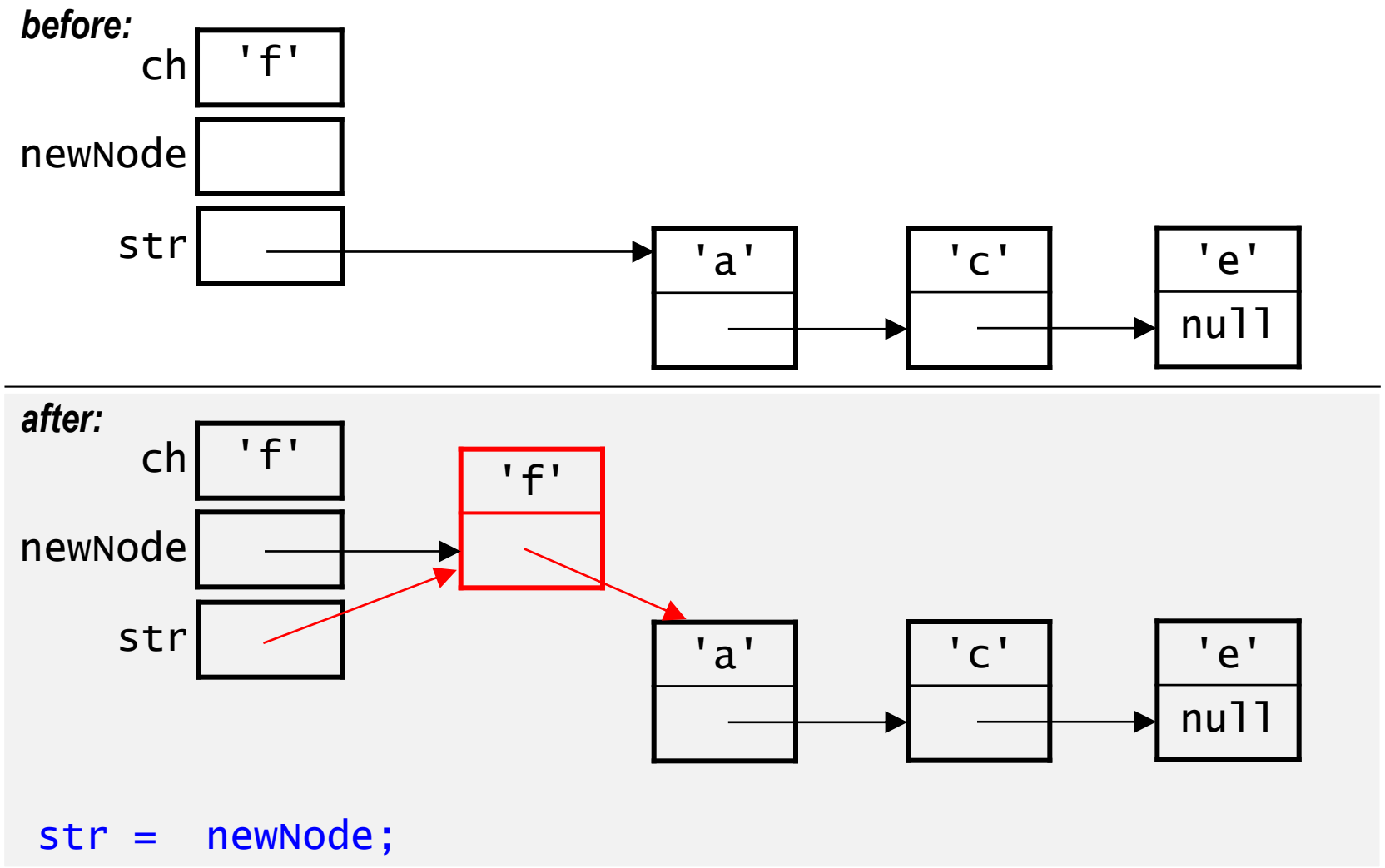
- <u>Special case:</u> `i == 0` (insertion at the front of the list)




```
StringNode newNode = new StringNode( ch, str );
```

Inserting an Item at Position i

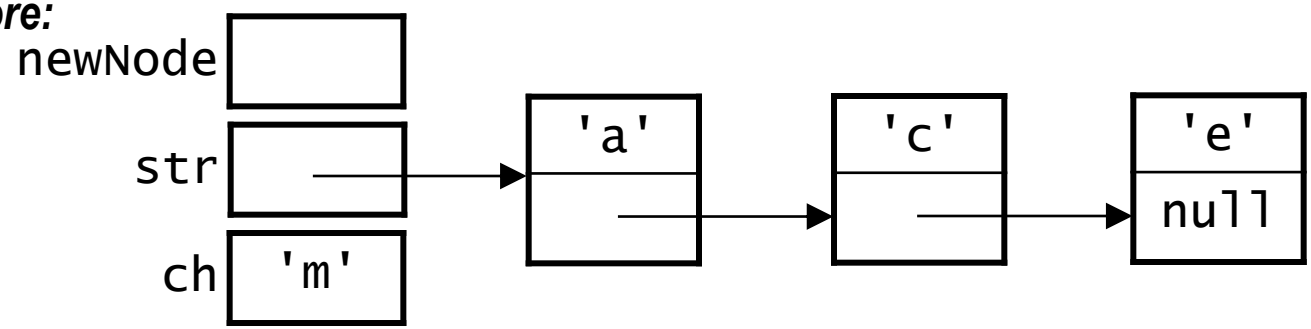
- <u>Special case:</u> `i == 0` (insertion at the front of the list)



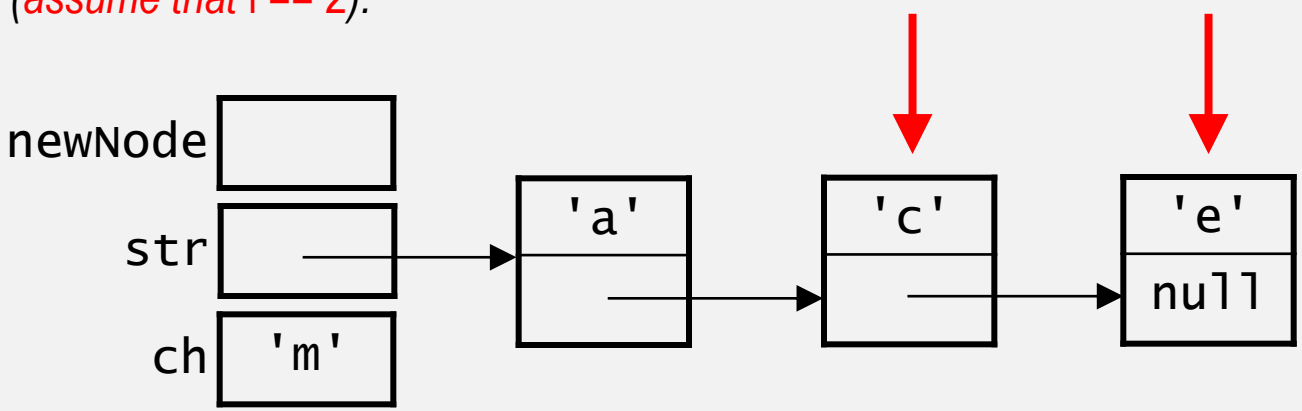
Inserting an Item at Position i (cont.)

- <u>General case:</u> `i > 0` (insert *before* the item currently in posn `i`)

before:



after *(assume that* i == 2*):*

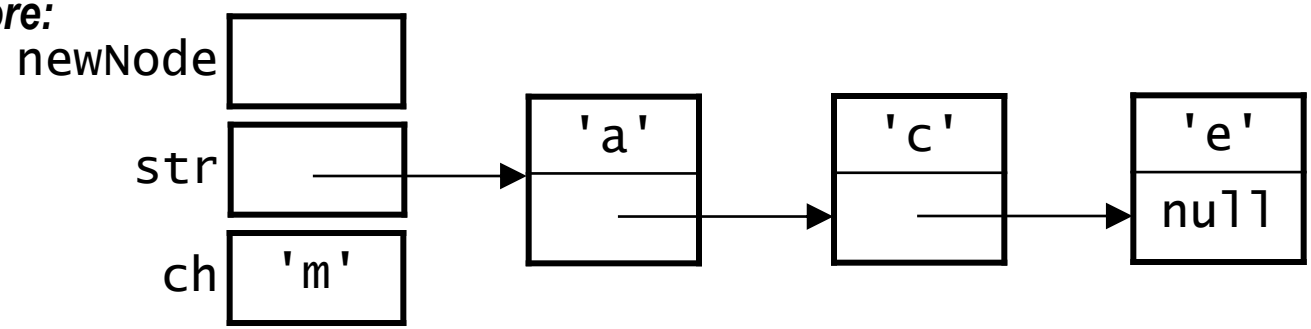



```
Statement to find the node at position i-1?
Statement to create the new node?
Assignment statement that inserts the node in the list?
```

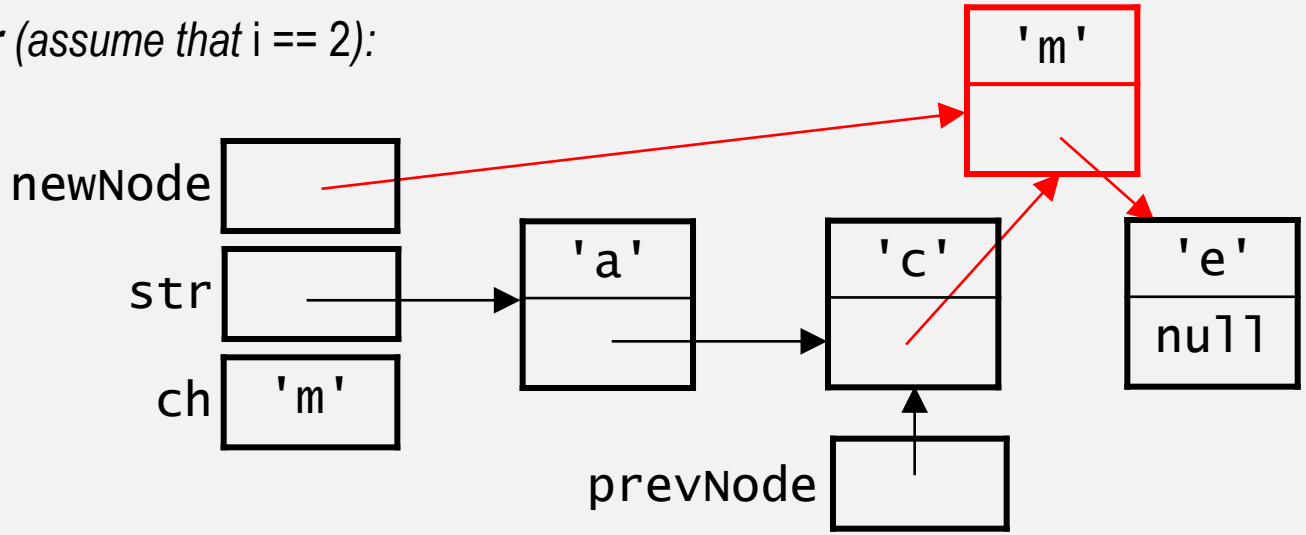
Inserting an Item at Position i (cont.)

- <u>General case:</u> `i > 0` (insert *before* the item currently in posn `i`)

before:



after *(assume that* i == 2*):*




```
StringNode prevNode = getNode(i - 1);
StringNode newNode = new StringNode(ch, prevNode.next);
prevNode.next = newNode;
```