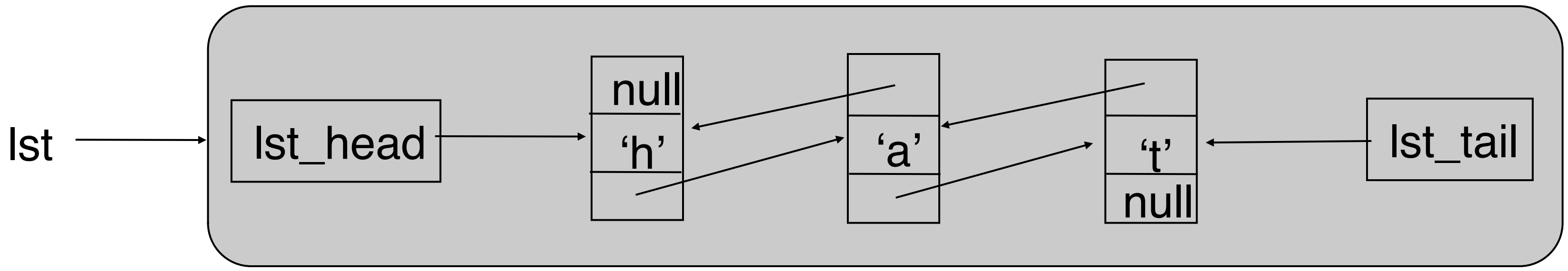
Linked List Operations

EECS 233

Doubly Linked List



- Both next and prev are defined in StringNode
- Why needed?

```
public class LLString {
        private StringNode lst_head;
        private StringNode lst_tail;
        private int theSize;
        ...
```

```
public class StringNode {
        private char ch;
        private StringNode next;
        private StringNode prev;
        ...
}
```

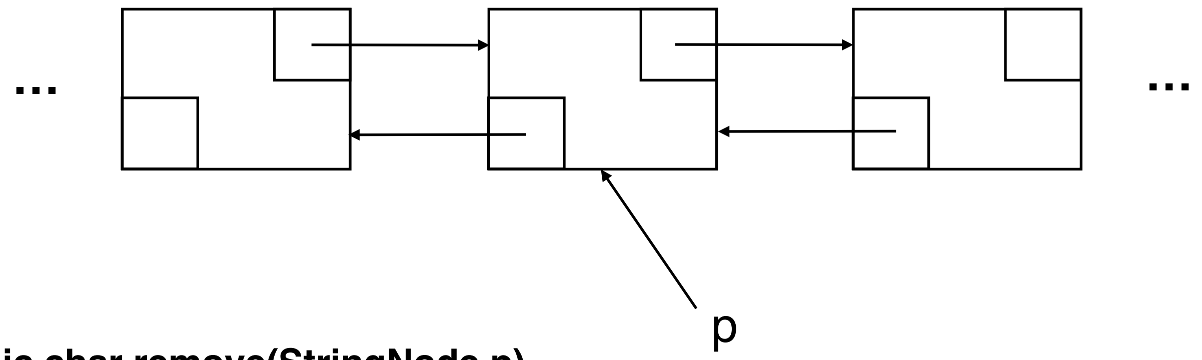
Example: Traversing Linked List

- Access the node at position *i* in a doubly linked list

```
public StringNode getNode(int i) {
        If (i < 0 || i >= theSize) throw an exception
        StringNode ptr;
        If (i < theSize/2) {
                ptr = lst_head;
                for (j = 0; j != i; j++) ptr = ptr.next;
        } else {
                ptr = lst_tail;
                for (j = theSize-1; j != i; j--) ptr = ptr.prev;
        }
        return ptr;
}
```

What is the running time?

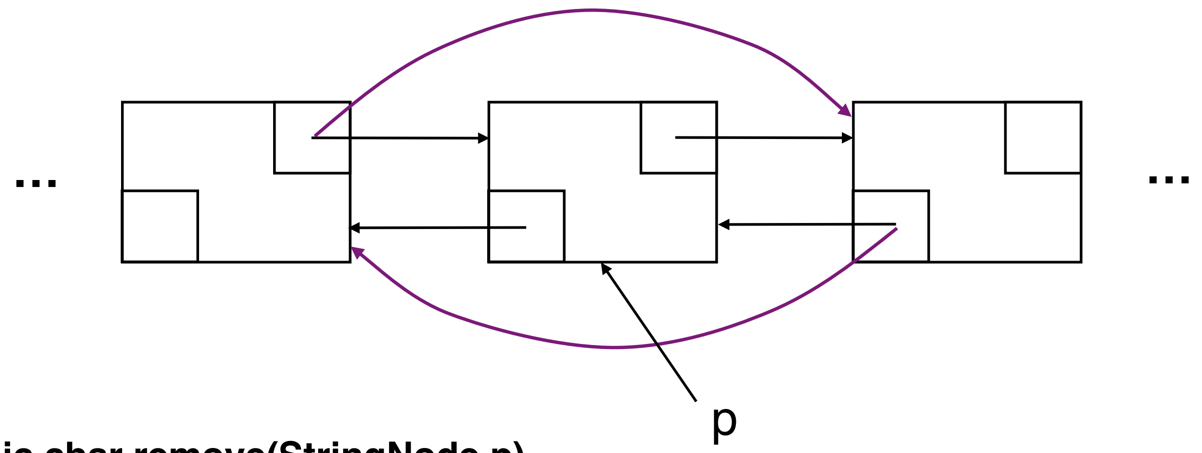
Example: Removing a Node



```
public char remove(StringNode p)
{
        if (p == lst_head || p == lst_tail)

                        ?

        p.next.prev = p.prev;
        p.prev.next = p.next;
        theSize--;

        return p.ch;
  }
```

Do we need to explicitly de-allocate p?

Example: Removing a Node



```
public char remove(StringNode p)
 {
        if (p == lst_head || p == lst_tail)

                        ?

        p.next.prev = p.prev;
        p.prev.next = p.next;
        theSize--;

        return p.ch;
   }
```

Do we need to explicitly de-allocate p?

Example: Inserting a Node

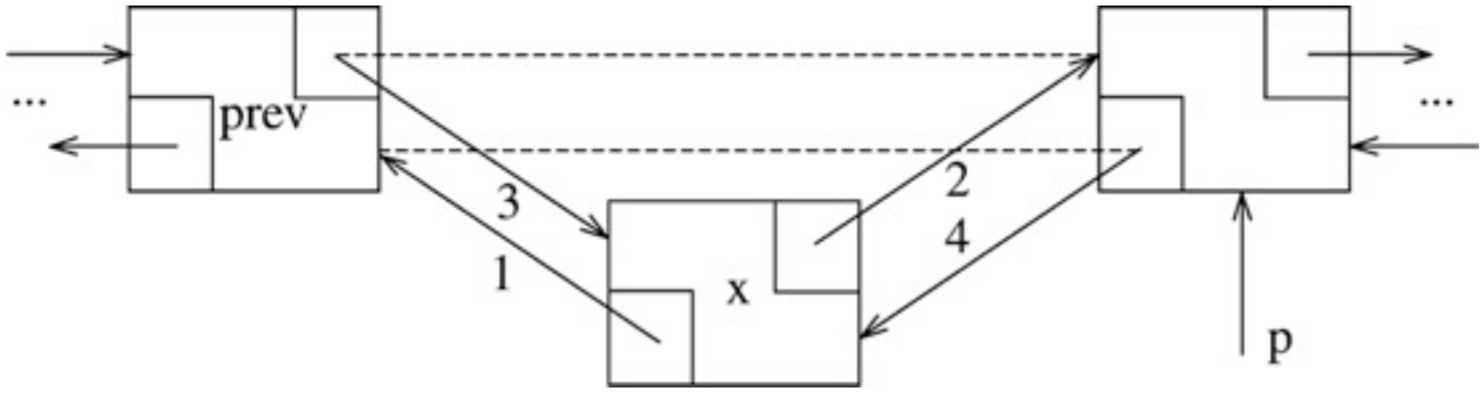
- Insert a new node before p.

newNode.prev = p.prev;

newNode.next = p;

p.prev.next = newNode;

p.prev = newNode;



Example: Inserting a Node

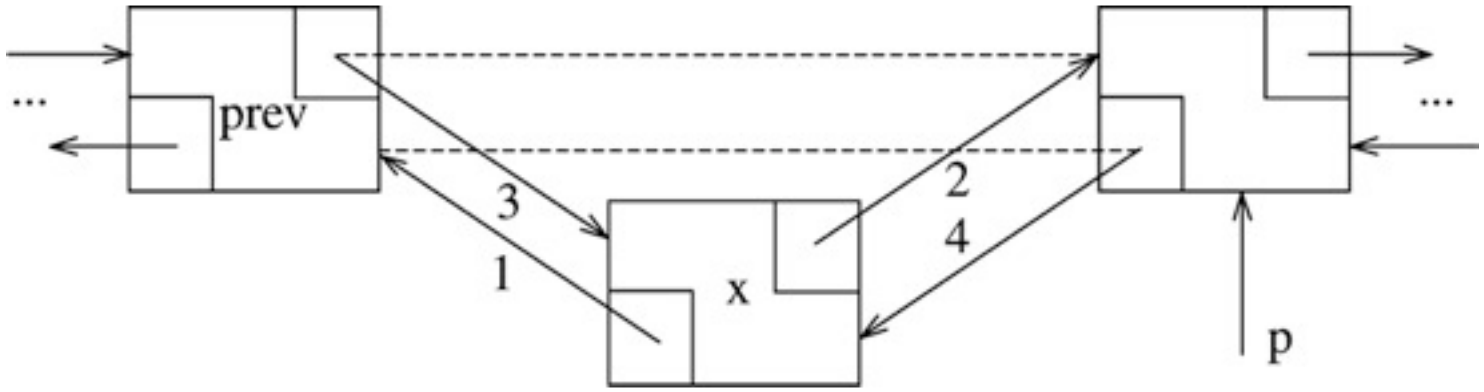
- Insert a new node before p.

```
newNode.prev = p.prev;
newNode.next = p;
p.prev.next = newNode;
p.prev = newNode;
```

What if p is the first element? Last element?

What if p == null?

What if p is not part of a list?



Other Operations

Either simple linked list or doubly linked list

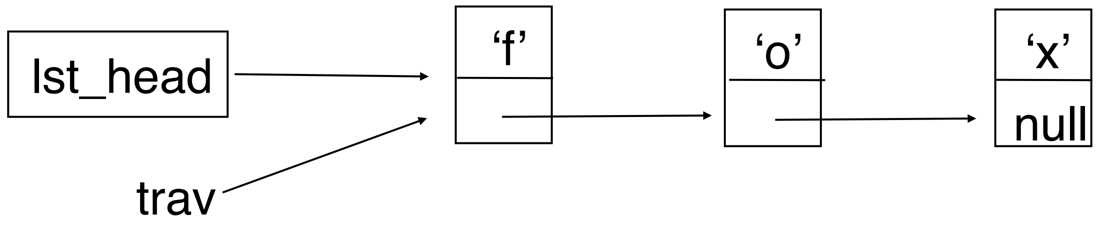
- Count the occurrences of an item in the linked list
- Remove all occurrences of an item
- Reverse a linked list (trivial for doubly linked list)
- Duplicate a linked list

Arrays versus Linked Lists

- Two implementations of list ADT
- Array implementation
  - + Compact
  - + Efficient random access (using index)
  - - Inefficient insert/delete operations
  - - Need to preallocate maximum size
- Link list implementation
  - + Efficient insert/delete
  - + Easy to grow
  - - Random access takes O(N) running time
  - - Uses more space for the links field
- Singly-linked lists versus doubly-linked lists

Traversing A Linked List

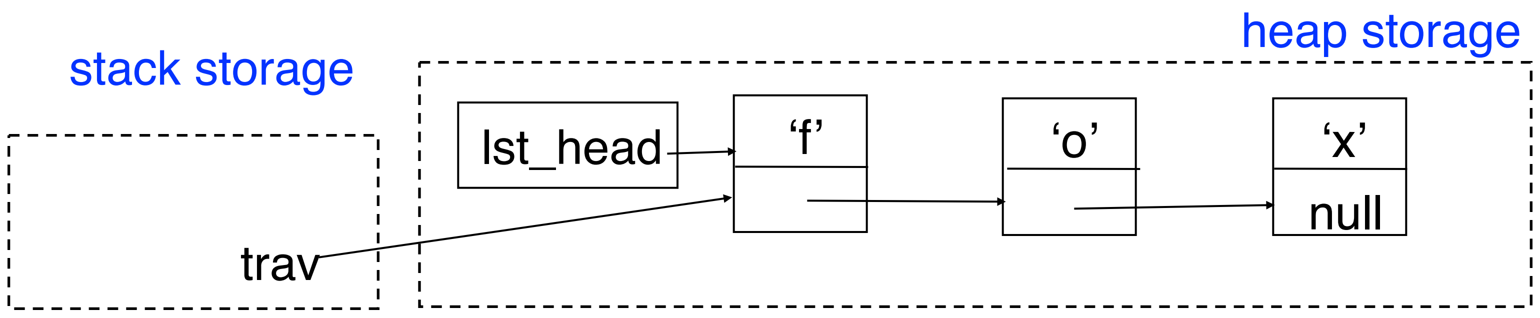
- Common operation for many tasks.
- Can be done using recursion or iteration.
- We make use of a variable (call it trav) that keeps track of where we are in the linked list (a simple linked list here).



- Template for traversing an entire linked list:

```
trav = lst_head.next; // start with the first node (or myList.getFirst() if
                                          // from outside
while (trav != null) {
         … // usually do something here
         trav = trav.next; // move trav down one node
         }
```

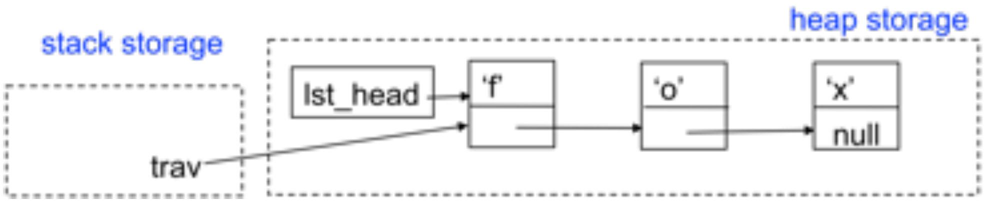
Example: toUpperCase()



- toUpperCase(str): converting str to all upper-case letters

```
public static void toUpperCase(LLString str) {
        StringNode trav;
        if (str.size() > 0) trav = str.getFirst();
        while (trav != null) {
            if (trav.ch >= 'a' && trav.ch <= 'z')
              trav.ch += ('A' – 'a');
            trav = trav.next;
        }
    }
```

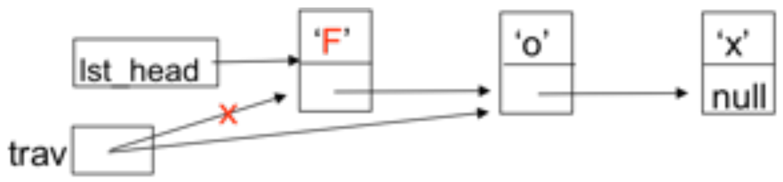
Tracing toUpperCase()



```
public static void toUpperCase(LLString str) {
    StringNode trav;
   if (str.size() > 0) trav = str.getFirst();
         while (trav != null) {
                if (trav.ch >= 'a' && trav.ch <= 'z')
                    trav.ch += ('A' – 'a');
                trav = trav.next;
         }
 }
```

Tracing toUpperCase()

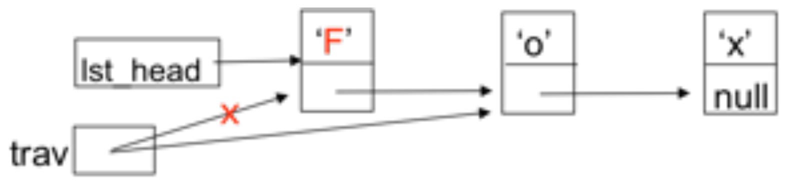
- After the first iteration in the while loop



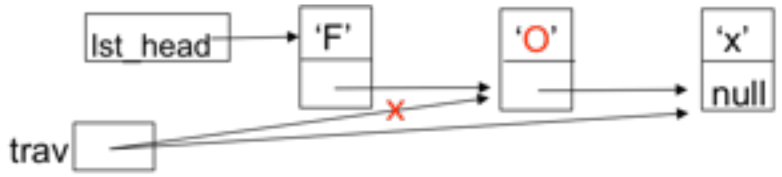
```
public static void toUpperCase(LLString str) {
     StringNode trav;
    if (str.size() > 0) trav = str.getFirst();
            while (trav != null) {
                      if (trav.ch >= 'a' && trav.ch <= 'z')
                           trav.ch += ('A' – 'a');
                      trav = trav.next;
            }
  }
```

Tracing toUpperCase()

- After the first iteration in the while loop



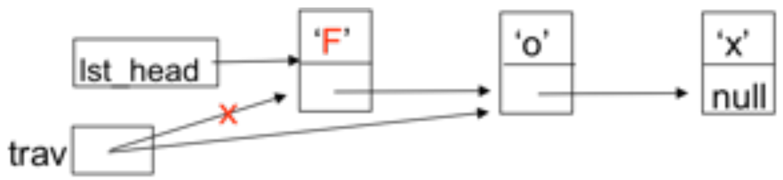
- After the second iteration:



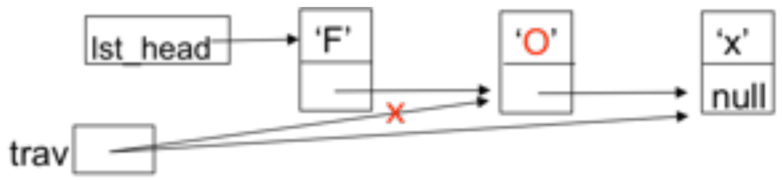
```
public static void toUpperCase(LLString str) {
    StringNode trav;
   if (str.size() > 0) trav = str.getFirst();
          while (trav != null) {
                 if (trav.ch >= 'a' && trav.ch <= 'z')
                     trav.ch += ('A' – 'a');
                 trav = trav.next;
          }
  }
```

Tracing toUpperCase()

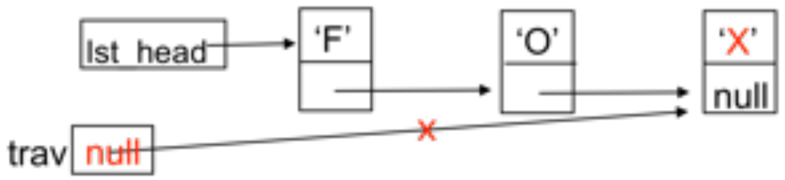
- After the first iteration in the while loop



- After the second iteration:



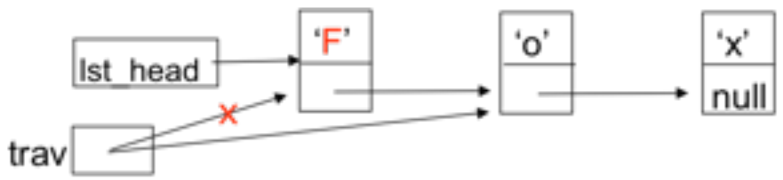
- After the third iteration



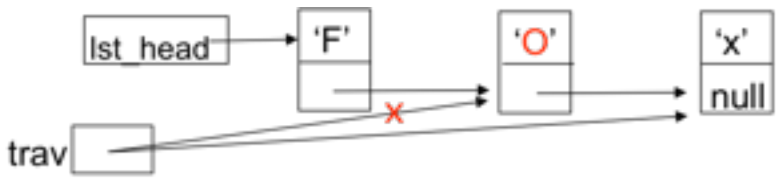
```
public static void toUpperCase(LLString str) {
     StringNode trav;
    if (str.size() > 0) trav = str.getFirst();
           while (trav != null) {
                   if (trav.ch >= 'a' && trav.ch <= 'z')
                       trav.ch += ('A' – 'a');
                   trav = trav.next;
           }
  }
```

Tracing toUpperCase()

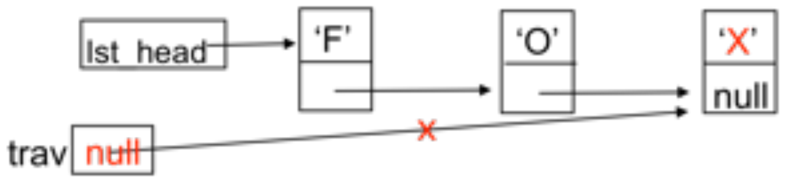
- After the first iteration in the while loop



- After the second iteration:



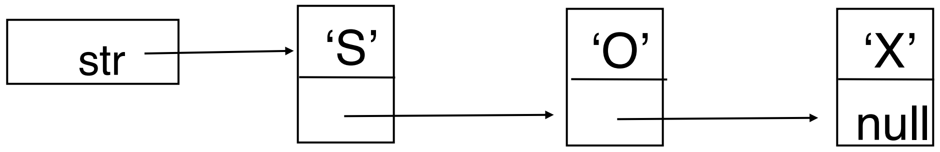
- After the third iteration



```
public static void toUpperCase(LLString str) {
     StringNode trav;
    if (str.size() > 0) trav = str.getFirst();
           while (trav != null) {
                   if (trav.ch >= 'a' && trav.ch <= 'z')
                       trav.ch += ('A' – 'a');
                   trav = trav.next;
           }
  }
```

- Now trav == null, so we break out of the loop and return from toUpperCase(). The changes are already reflected in the linked list.

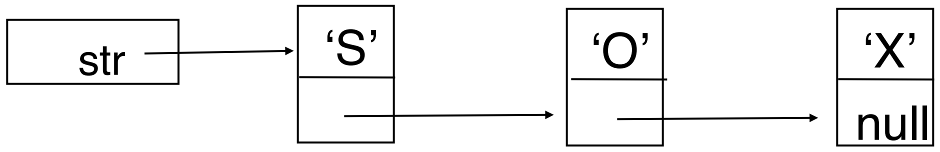
Duplicating A Singly Linked List



- Helper method:
  - Take the starting StringNode
  - Copy all elements to the end
  - Return the first element of the new list
- Recursive method copy(str)
  - Base case: if str is empty, return null
  - Recursion: copy the first character and then make a recursive call to copy the rest
- Preliminaries: StringNode constructor

```
public Class StringNode {
        private char ch;
        private StringNode next;
        public StringNode(char myCh, StringNode nextNode){
            ch = myCh;
            next = nextNode;
        }
        …
```

Duplicating A Simple Linked List



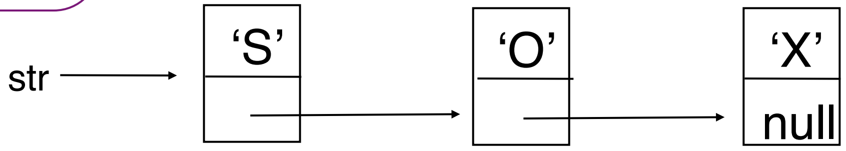
- Recursive method to copy(str)
  - Base case: if str is empty, return null
  - Recursion: copy the first character and then make a recursive call to copy the rest

```
private static StringNode copy(StringNode str) {
        if (str == null) // base case
                return null;
        // create the first node, copying the first character into it
        StringNode copyFirst = new StringNode(str.ch, null);
        // make a recursive call to get a copy of the rest and
        // store the result in the first node's next field
        copyFirst.next = copy(str.next);
        return copyFirst;
}
```

```
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;
 }
```

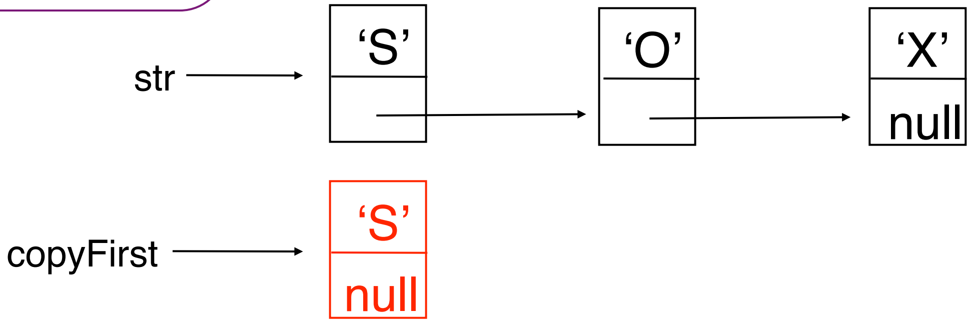
```
public static StringNode copy(StringNode str) {
  if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch, null);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```

In the first call:



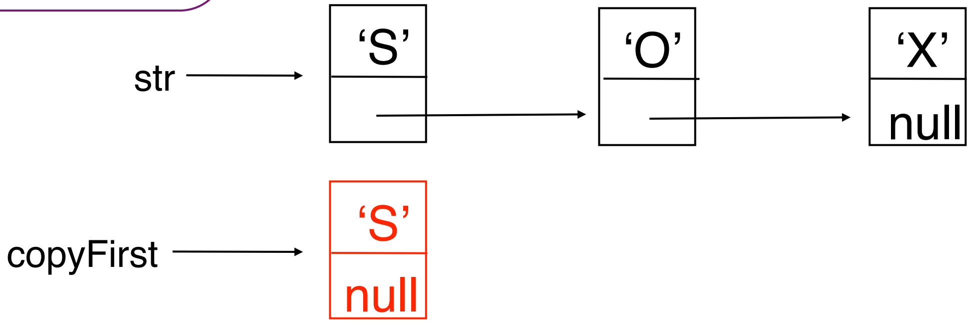
```
public static StringNode copy(StringNode str) {
  if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch, null);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```

In the first call:



```
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;
 }
```

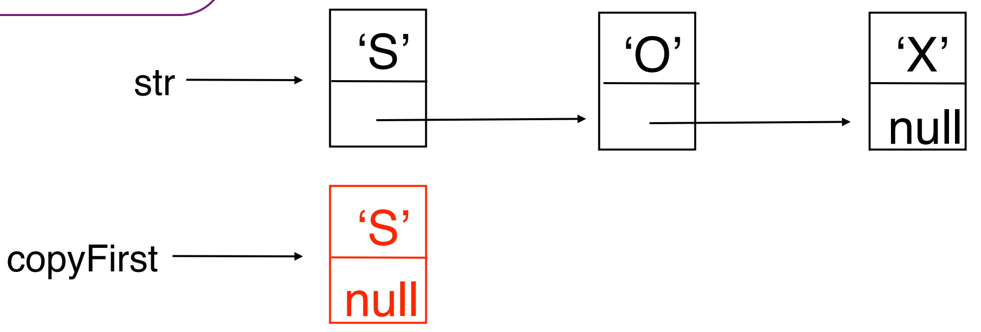
In the first call:



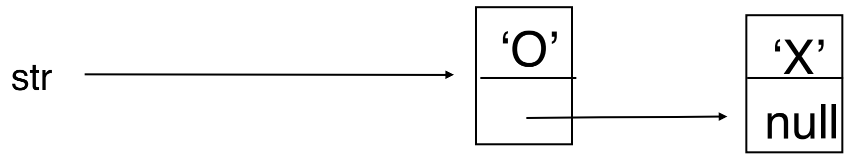
In the second call:

```
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;
 }
```

In the first call:

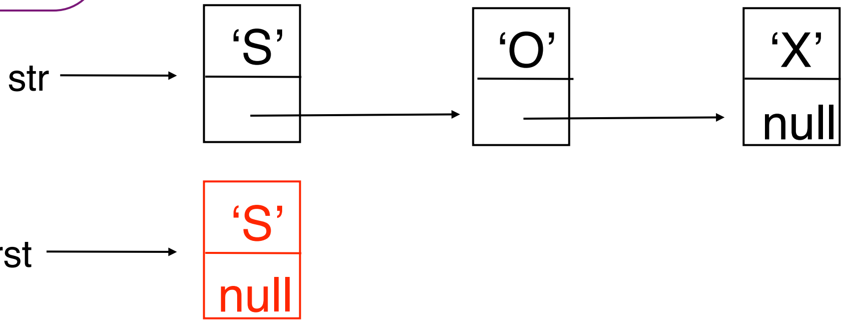


In the second call:

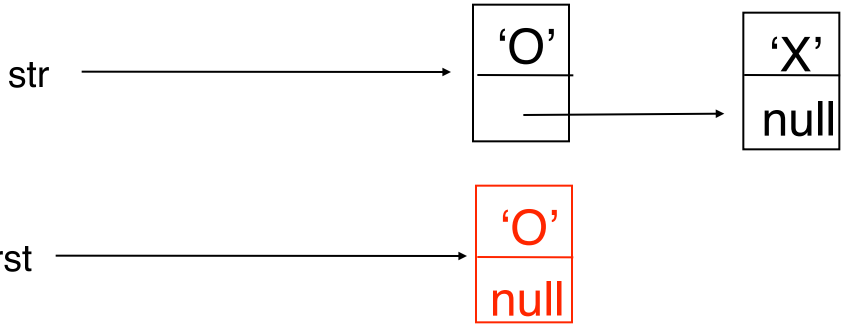


```
public static StringNode copy(StringNode str) {
  if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch, null);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```

In the first call:

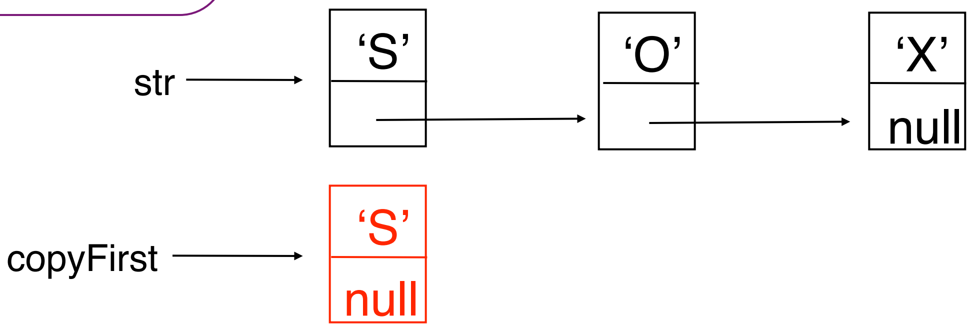


In the second call:

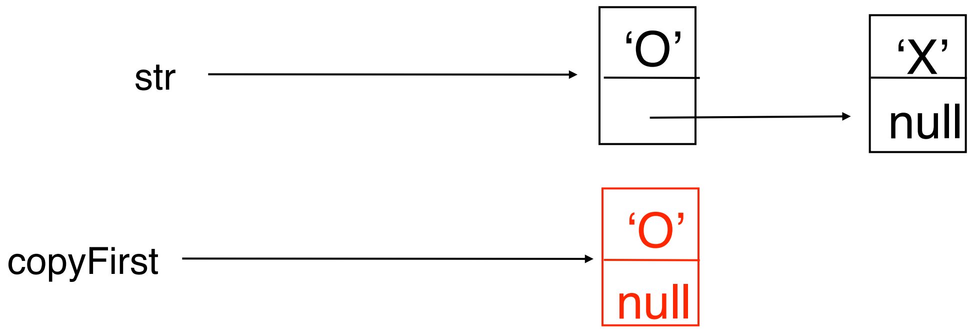


```
public static StringNode copy(StringNode str) {
  if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch, null);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```

In the first call:



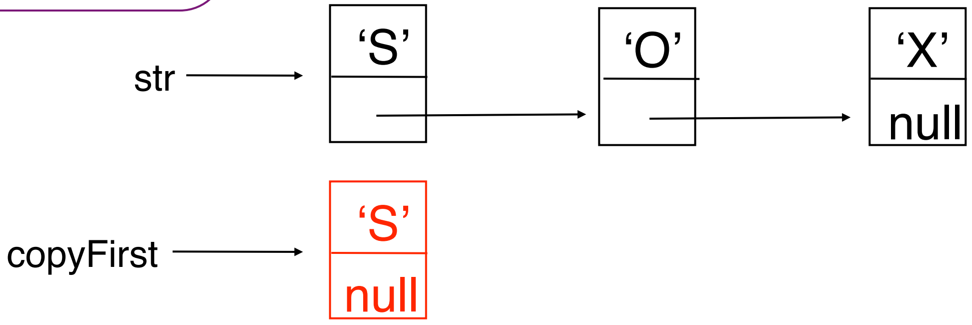
In the second call:



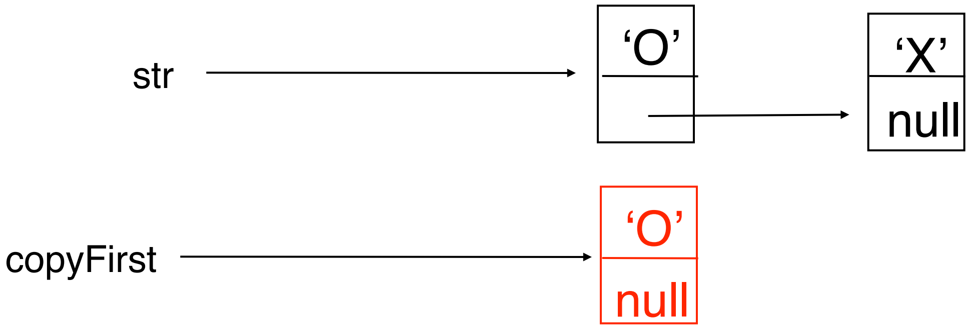
In the third call:

```
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;
 }
```

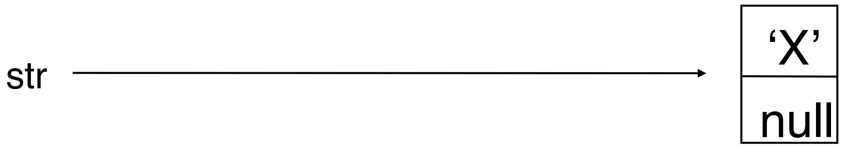
In the first call:



In the second call:

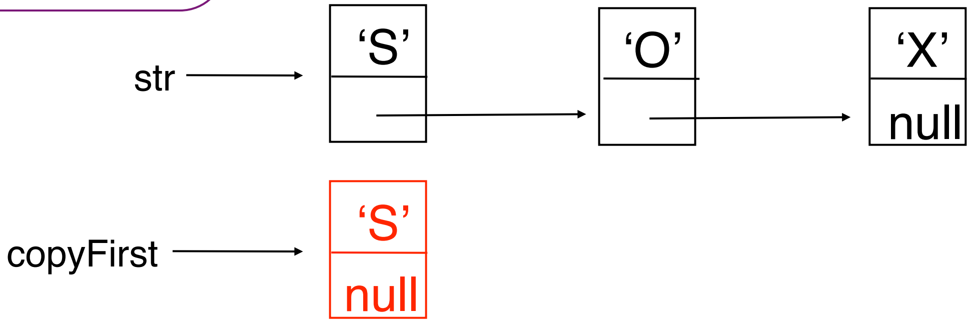


In the third call:

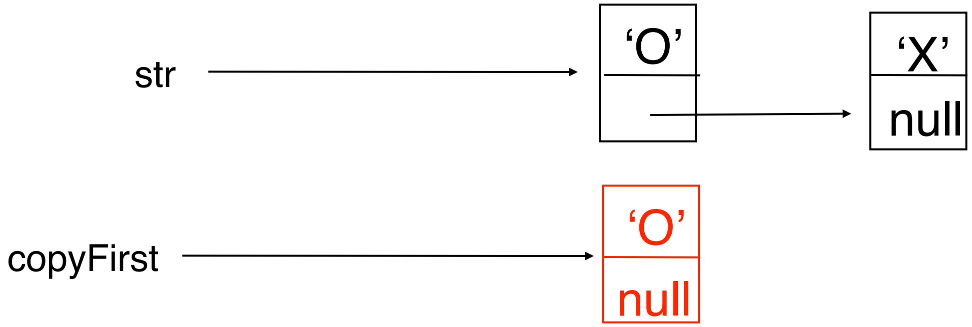


```
public static StringNode copy(StringNode str) {
  if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch, null);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```

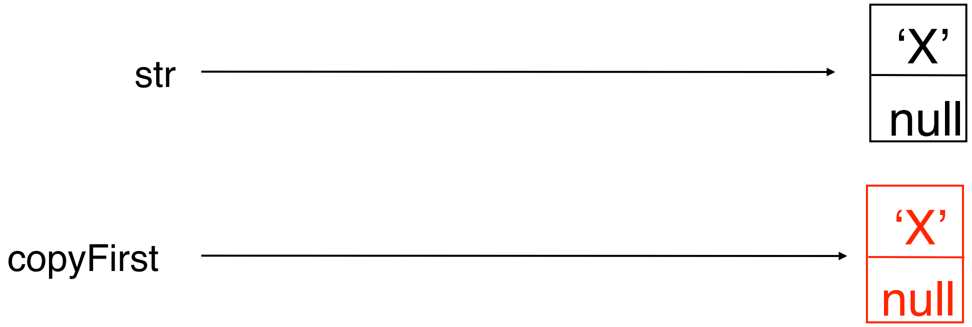
In the first call:



In the second call:



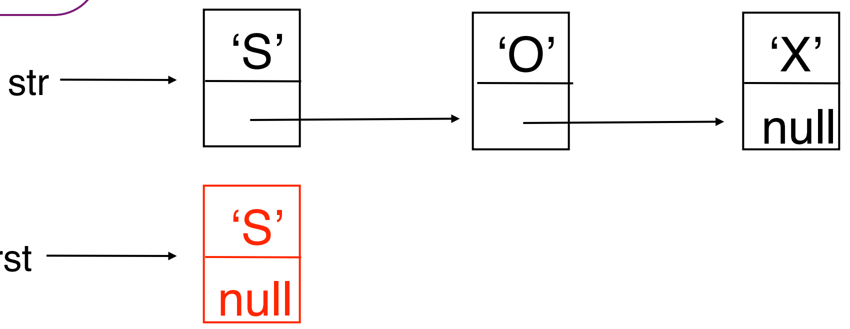
In the third call:



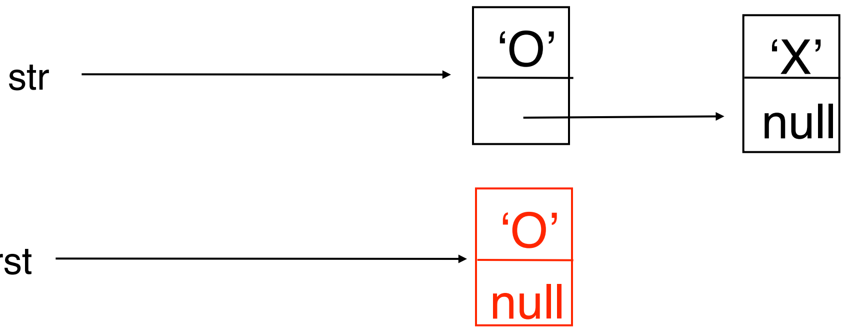
--

```
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;
 }
```

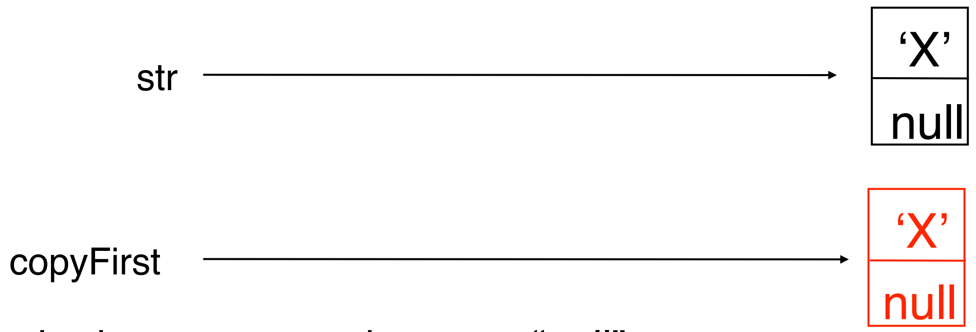
In the first call:



In the second call:



In the third call:

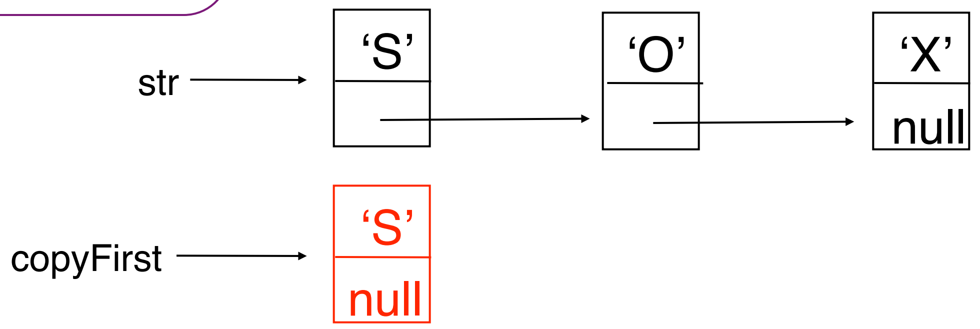


The fourth call reaches the base case and returns "null" :

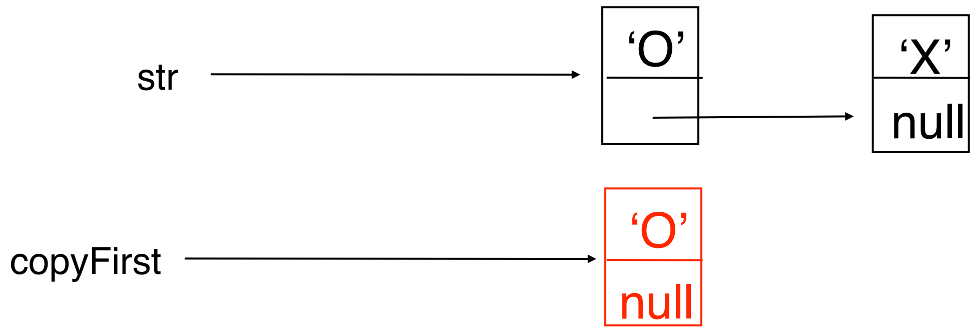
--

```
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;
 }
```

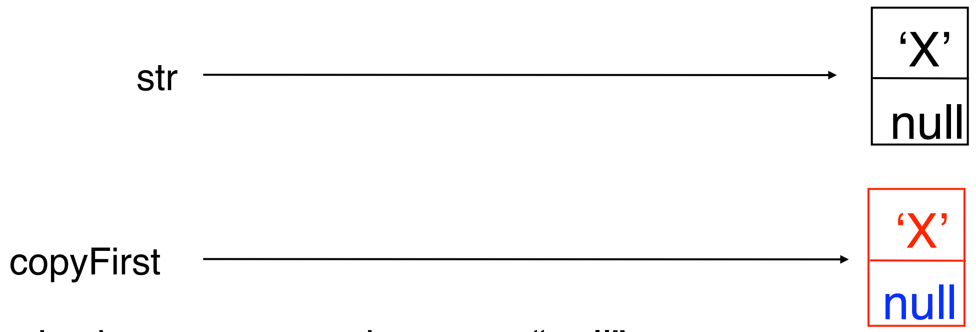
In the first call:



In the second call:



In the third call:



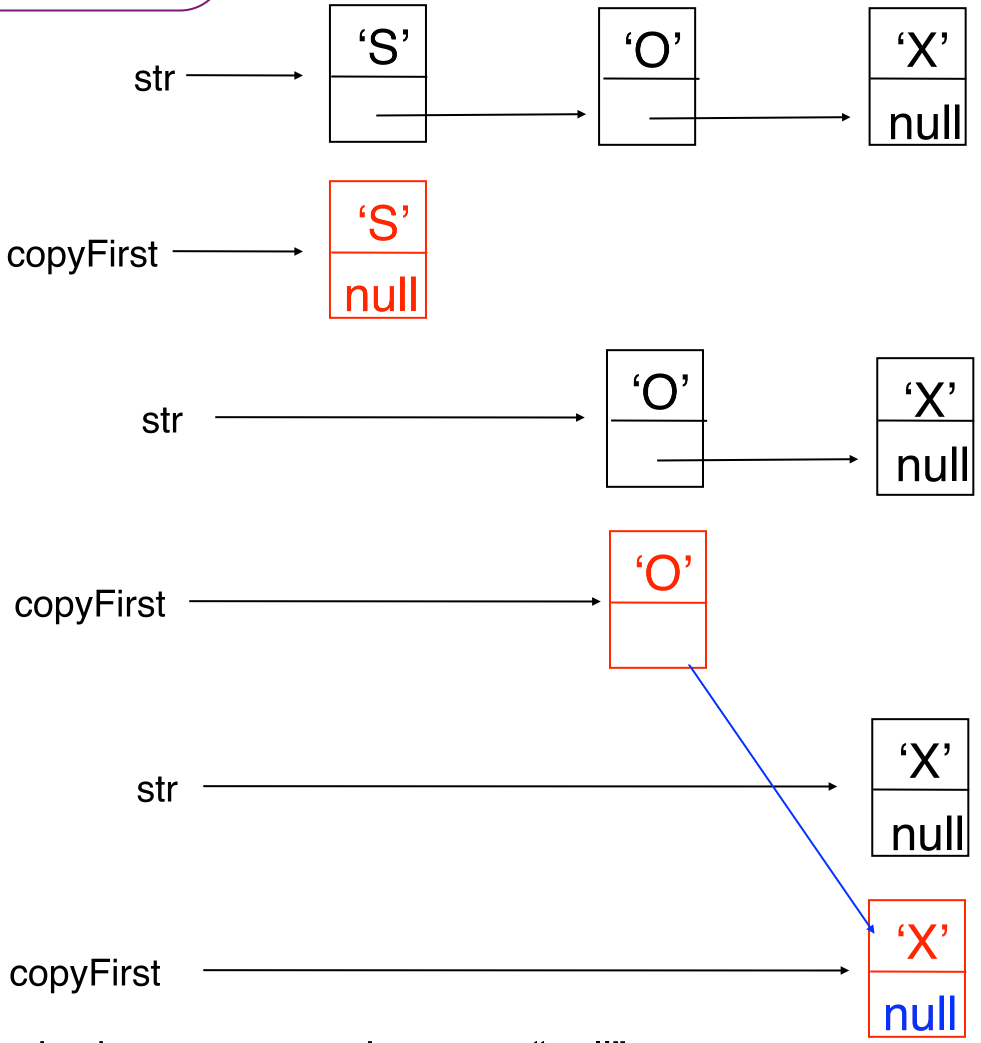
The fourth call reaches the base case and returns "null" :

```
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;
 }
```

In the first call:

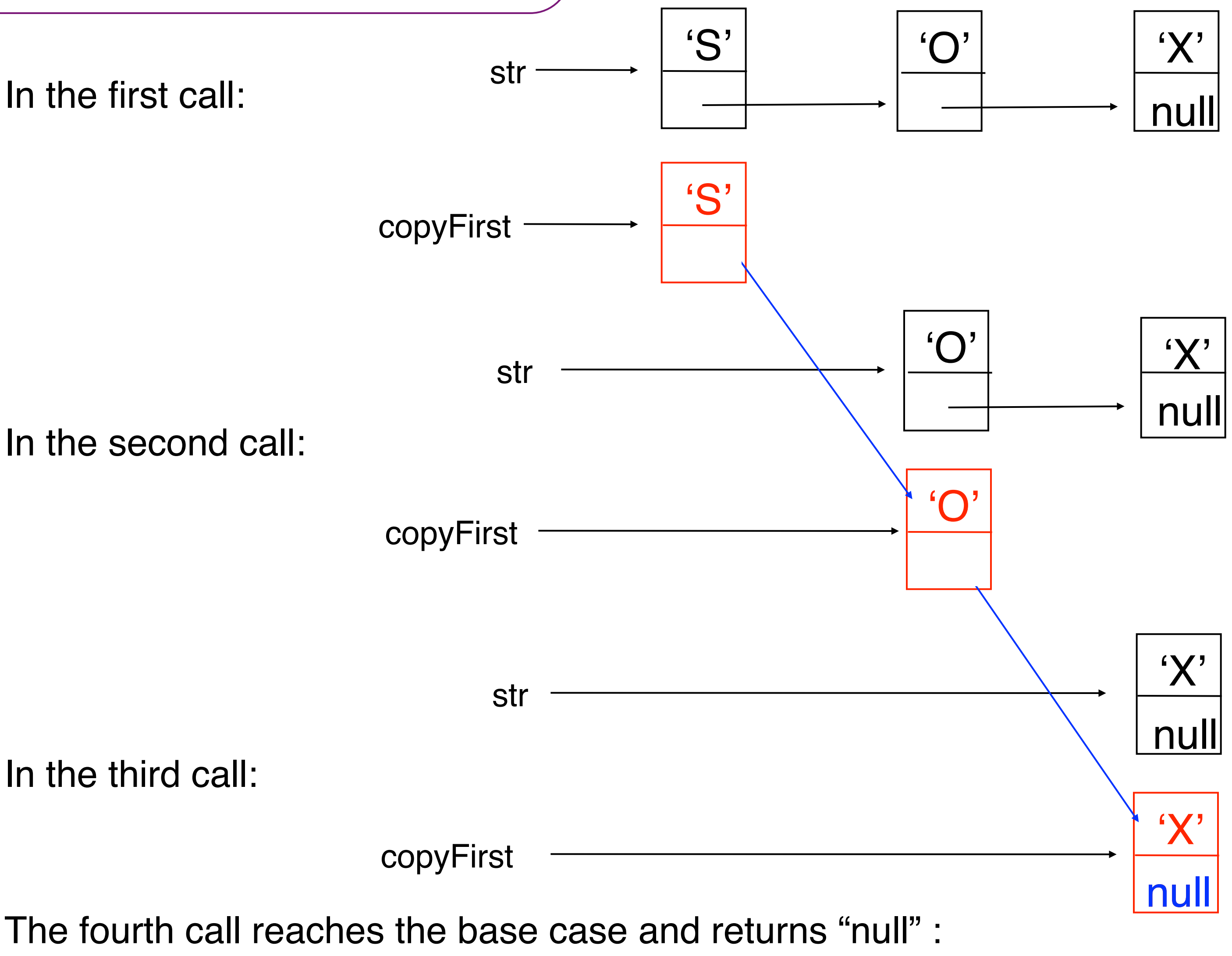
In the second call:

In the third call:



The fourth call reaches the base case and returns “null” :

```
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;
 }
```



The fourth call reaches the base case and returns "null" :

```
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;
 }
```

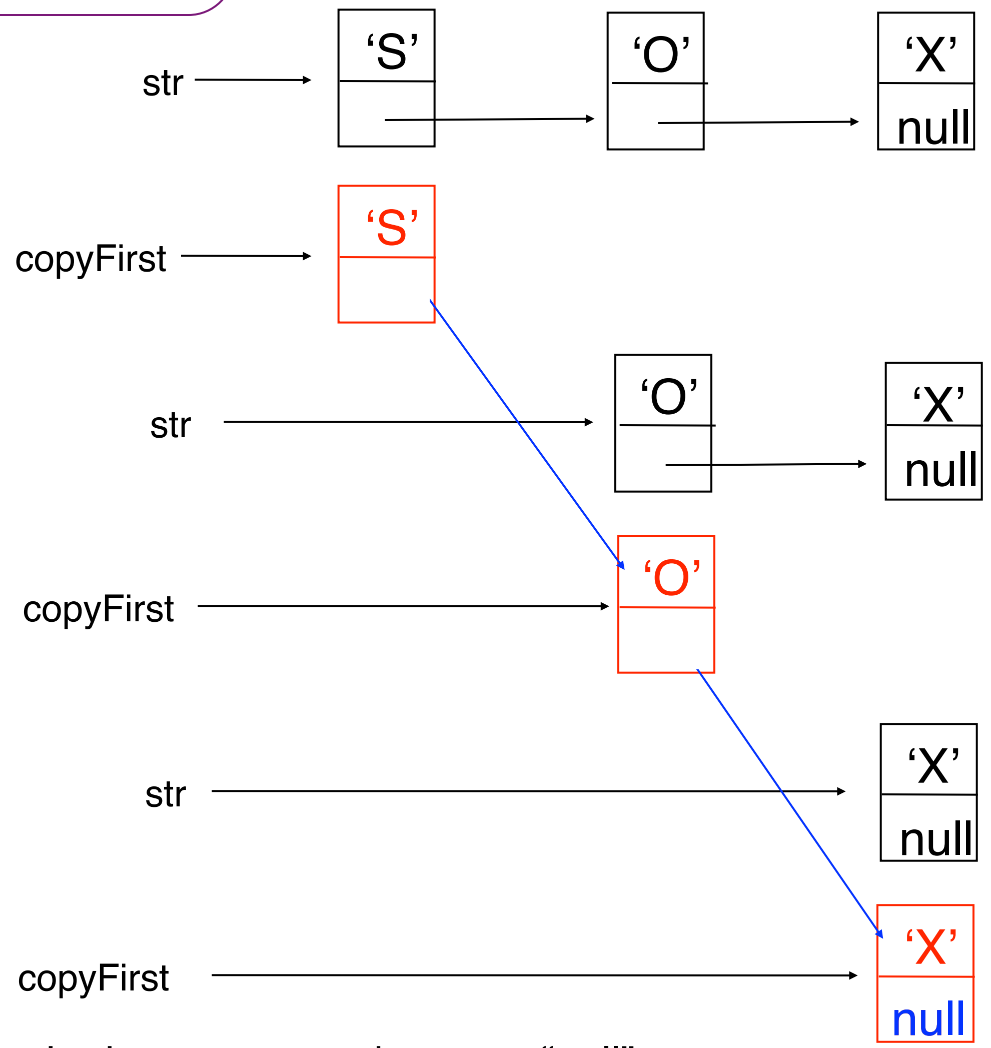
In the first call:

Finally return this

In the second call:

In the third call:

The fourth call reaches the base case and returns "null" :



--

A Slight Modification

```
public static StringNode copy(StringNode str) {
        if (str == null)
                    return null;
        StringNode copyFirst = new StringNode(str.ch, null); // create the first node, copying
                                                               // the first character into it
        copyFirst.next = copy(str.next); // make a recursive call to copy the rest and
                                          // store the result in the first node's next field
        return copyFirst;
}
```

Replace the highlighted lines with

```
return new StringNode(str.ch, copy(str.next));
```

A Slight Modification

```
public static StringNode copy(StringNode str) {
        if (str == null)
                    return null;
        StringNode copyFirst = new StringNode(str.ch, null); // create the first node, copying
                                                             // the first character into it
        copyFirst.next = copy(str.next); // make a recursive call to copy the rest and
                                         // store the result in the first node's next field
        return copyFirst;
}
```

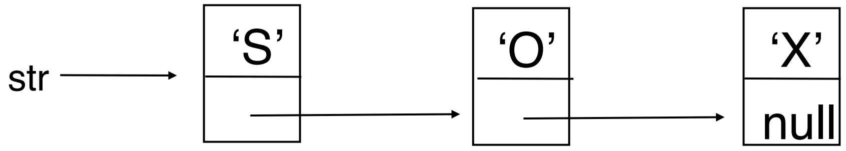
Replace the highlighted lines with

```
return new StringNode(str.ch, copy(str.next));
```

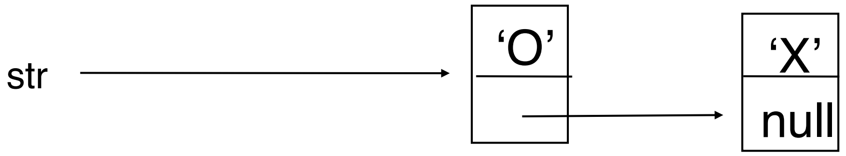
```
public static StringNode copy(StringNode str) {
    if (str == null) return null;
    return new StringNode(str.ch, copy(str.next));
}
```

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

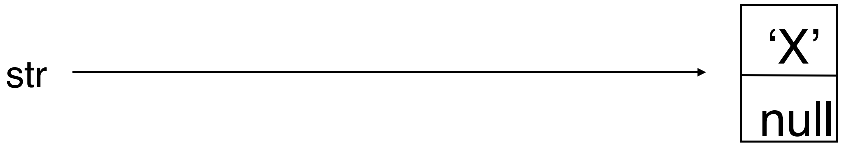
In the first call:



In the second call:



In the third call:



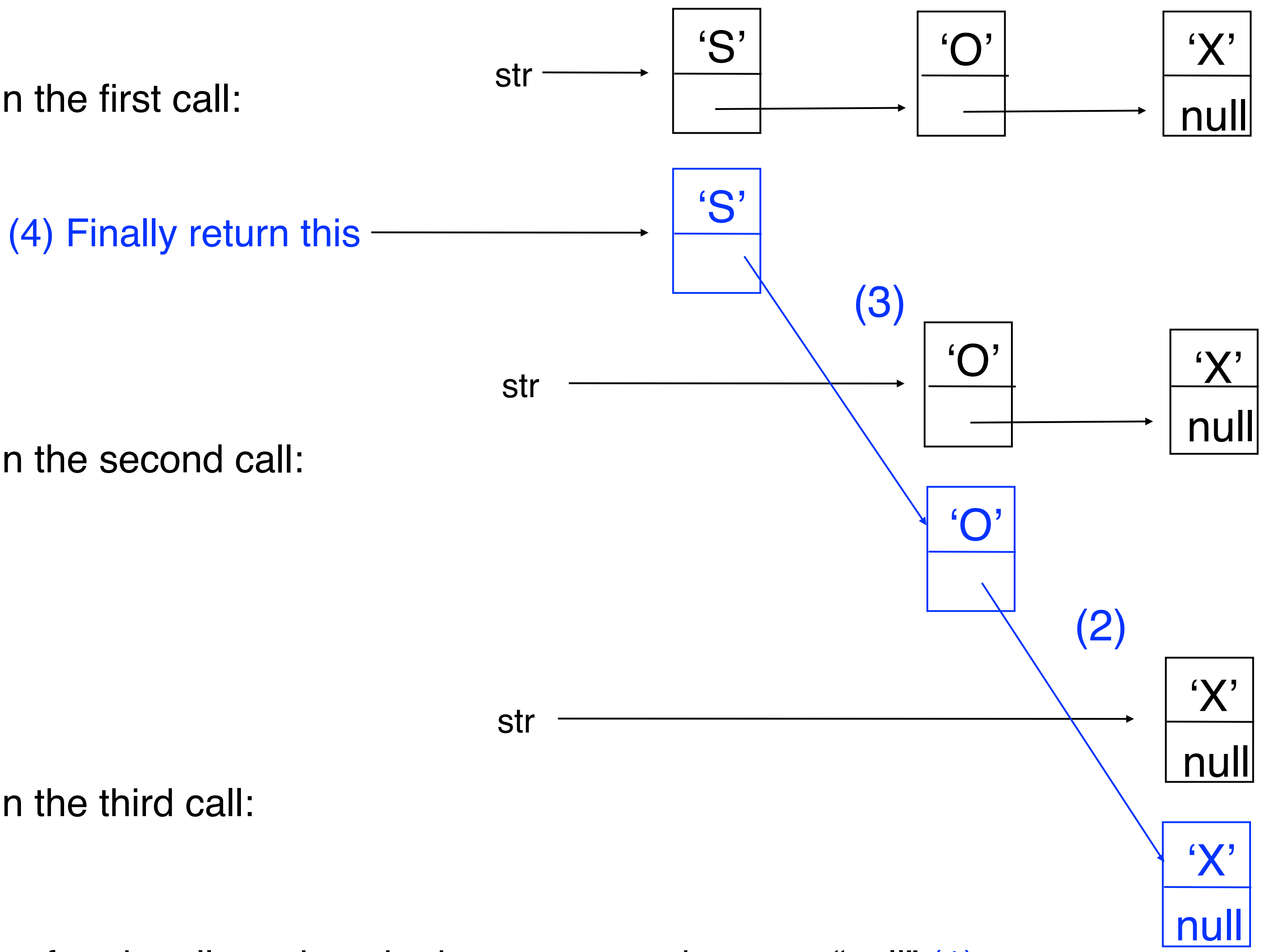
The fourth call reaches the base case and returns “null” :

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

In the first call:

In the second call:

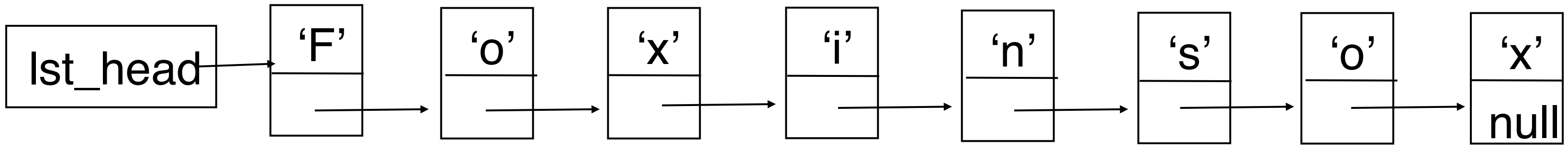
In the third call:



The fourth call reaches the base case and returns “null” (1):

More on Lists: Iterators

- Example: count the number of times that an item ‘o’ appears in a list.



- One possible implementation: a method in another class

```
public class MyClass {
        public static int numOccur(LLString str, char ch) {
                int numOccur = 0;
                for (int i = 0; i < str.length(); i++) {
                    if (ch == str.getChar(i))
                                        numOccur++;
                }
                return numOccur;
        } …
```

- length() and getChar() are defined public methods in LLString
- What is the running time of getChar(), and what is that of numOccur()? O(?)

--

Solution 1: Make numOccur() A LLString Method

```
public class LLString {
          public int numOccur(char ch) {
                    int numOccur = 0;
                    StringNode trav = lst_head;
                    while (trav != null) {
                              if (trav.ch == ch)
                                   numOccur++;
                              trav = trav.next;
                    }
                    return numOccur;
          } …
```

- Number of accesses = ? O(?)

- Problem: we can't anticipate all of the types of operations that users may wish to perform.
- We would like to give users the general ability to iterate over the list.

Solution 2: Give Access to the Internals of the List

- Make StringNode visible
- Provide public “get” methods
  - getNode(i) in LLString
  - getNext() in StringNode

- This would allow us to do the following:

```
public class MyClass {
        public static int numOccur(LLString str, char ch) {
                int numOccur = 0;
                StringNode trav = str. getNode(0);
                while (trav != null) {
                        char c = trav. getChar();
                        if (c == ch)
                            numOccur++;
                        trav = trav. getNext();
                }
                return numOccur;
        } …
```

Solution 2: Give Access to the Internals of the List

- Make StringNode visible
- Provide public “get” methods
  - getNode(i) in LLString
  - getNext() in StringNode

- This would allow us to do the following:

```
public class MyClass {
        public static int numOccur(LLString str, char ch) {
                int numOccur = 0;
                StringNode trav = str. getNode(0);
                while (trav != null) {
                        char c = trav. getChar();
                        if (c == ch)
                            numOccur++;
                        trav = trav. getNext();
                }
                return numOccur;
        } …
```

Makes numOccur dependent on implementation of the list!

Solution 3: Provide an Iterator

- An iterator is an object that provides the ability to iterate over a list *without* violating encapsulation.

- Our Iterator class will have two methods:

  // Are there more items to visit?

  boolean hasNext()

  // Return next item and advance the iterator.

  char next()

- A newly created Iterator object starts out prepared to access the first item in the list, and we use next() to access the items sequentially.

- Example: position of the iterator is shown by the cursor symbol (|)
  - *after the iterator* i *is created:* | “F” “O” “X” …
  - *after calling* i.next()*, which returns* “F”: “F” | “O” “X” …
  - *after calling* i.next()*, which returns* “O”: “F” “O” | “X” …

A List Iterator Class

- Iterator state
  - Keeping cursor position: instance variable “nextNode”
- Any Iterator object is associated with a given LLString object
- Must allow access from Interator to the internals of the associated LLString object
- Multiple iterator objects can be created for the same LLString object

A List-Iterator as Inner Class

- Iterator state
  - Cursor: instance variable “nextNode”
- Any Iterator object is associated with a given LLString object
  - Make Iterator class an inner class of LLString
  - Allows access from Interator to the internals of the associated LLString object
- Multiple iterator objects can be created for the same LLString object

A List-Iterator as Inner Class

- Iterator state
  - Cursor: instance variable “nextNode”
- Any Iterator object is associated with a given LLString object
  - Make Iterator class an inner class of LLString
  - Allows access from Interator to the internals of the associated LLString object
- Multiple iterator objects can be created for the same LLString object

- Iterator as a private inner class.

```
public class LLString {
        private StringNode lst_head;
        private StringNode lst_tail;
        …
        public Iterator iterator(){
            Iterator iter = new Iterator();
            return iter;
        }

}
```

```
private class Iterator {
     private StringNode nextNode;
     private Iterator (){
          nextNode = lst_head;
     }
     …
}
```

- Creation:

```
LLString.Iterator myIter1 = string.iterator();
LLString.Iterator myIter2 = string.iterator();
```

Internals of the Iterator Class

- Two methods are provided in Iterator class:

```
public boolean hasNext() {
        return (nextNode != null);
}
public char next() {
        if (nextNode == null)
                throw exception;
        char ch = nextNode.ch;
        nextNode = nextNode.next;
        return ch;
}
```

- next() does two things:
  - it returns the character stored in the current node
  - it advances the iterator so that it is ready to access the next node

numOccur() Using an Iterator

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
            int numOccur = 0;
            LLString.Iterator iter = str.iterator();
            while (iter.hasNext()) {
                  char ch = iter.next();
                  if (c == ch)
                     numOccur++;
            }
            return numOccur;
      }
      ...
}
```

- The method is outside the LLString class, but it's able to iterate over the characters in the list efficiently without violating encapsulation
  - No usage of StringNode objects
  - Does not depend on LLString internals

numOccur() Using an Iterator

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
            int numOccur = 0;
            LLString.Iterator iter = str.iterator();
            while (iter.hasNext()) {
                char ch = iter.next();
                if (c == ch)
                    numOccur++;
            }
            return numOccur;
    }
    ...
}
```

```
public class MyClass {
    public static int numOccur(List str, char ch) {
        int numOccur = 0;
        List.Iterator iter = str.iterator();
        while (iter.hasNext()) {
            char ch = iter.next();
            if (c == ch)
                numOccur++;
        }
        return numOccur;
    }
    ...
}
```

- The method is outside the LLString class, but it's able to iterate over the characters in the list efficiently without violating encapsulation
  - No usage of StringNode objects
  - Does not depend on LLString internals

Java Support for Iterators

- Java's built-in collection classes all support iterators.
  - Java's Iterator classes are generic
  - the built-in Iterator interface (java.util.Iterator<AnyType> and java.util.ListIterator<AnyType>) specifies the iterator methods
  - they include hasNext() and next() methods like ours (in addition the remove() method)
  - users of an iterator use the interface name as the type of the iterator object