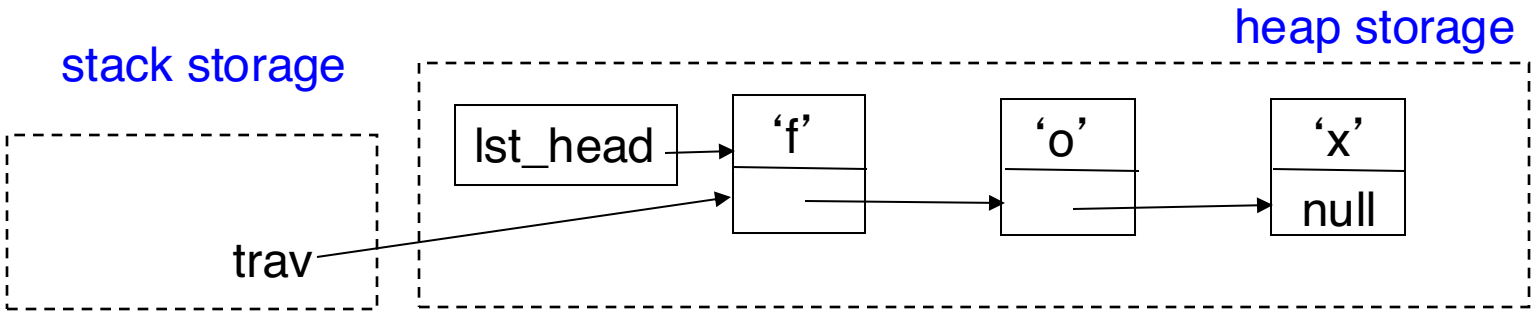
More Linked List Operations

EECS 233

toUpperCase()

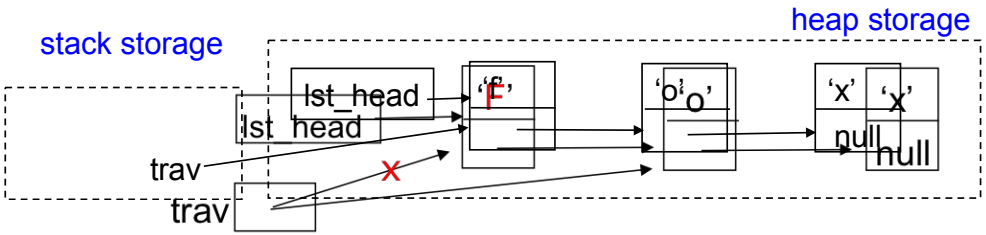


- toUpperCase(str): converting str to all upper-case letters

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

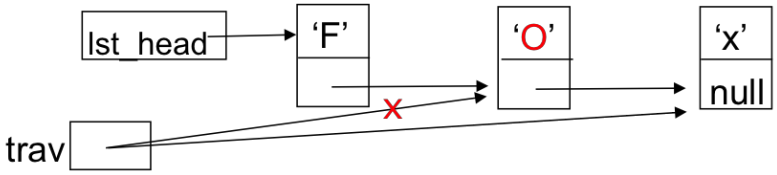
Tracing toUpperCase()

- After the first iteration in the while loop

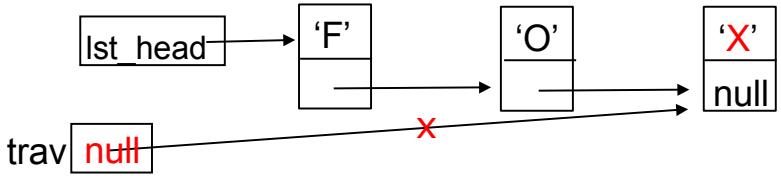



```
private static void toUpperCase(StringNode str) {
        StringNode trav = str;
        while (trav != null) {
            if (trav.ch >= 'a' && trav.ch <= 'z')
                trav.ch += ('A' – 'a');
            trav = trav.next;
        }
}
```

- After the second iteration:



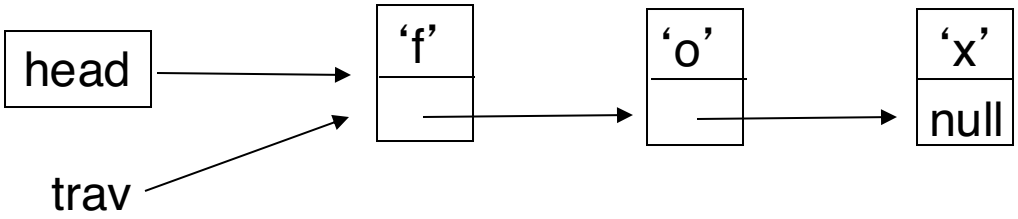
- After the third iteration



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

Traversing A Linked List

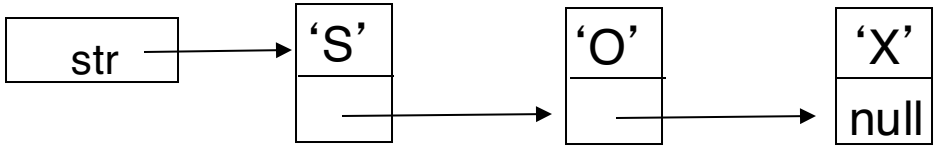
- Common operation for many tasks.
- Can be done using recursion or iteration.
- We make use of a variable (call it trav) that keeps track of where we are in the linked list (a simple linked list here).



- Template for traversing an entire linked list:

```
trav = head; // start with the first node
while (trav != null) {
        … // usually do something here
        trav = trav.next; // move trav down one node
        }
```

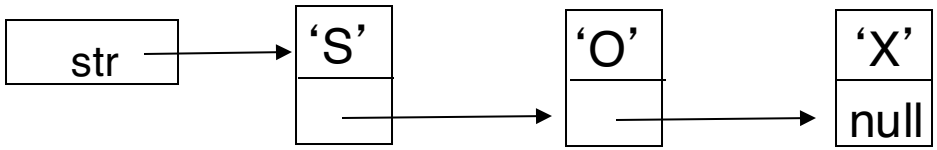
Duplicating A Singly Linked List



- Helper method copy(str):
  - Take the starting StringNode
  - Copy all elements through the end
  - Return the first element of the new list
- Recursive implementation:
  - 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

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

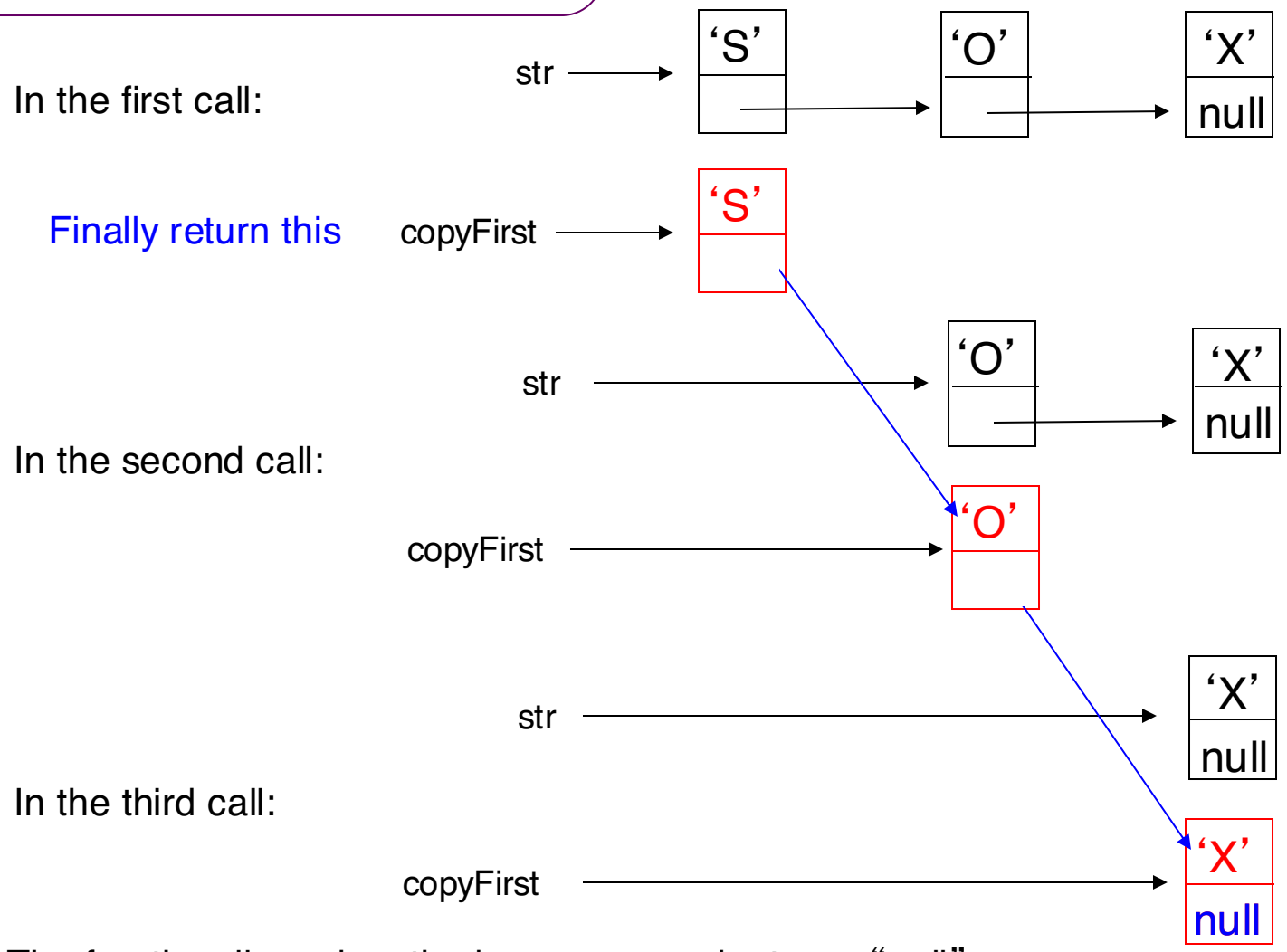
Duplicating A Simple Linked List



- Recursive method to copy(str)

```
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);
        // 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;
}
```

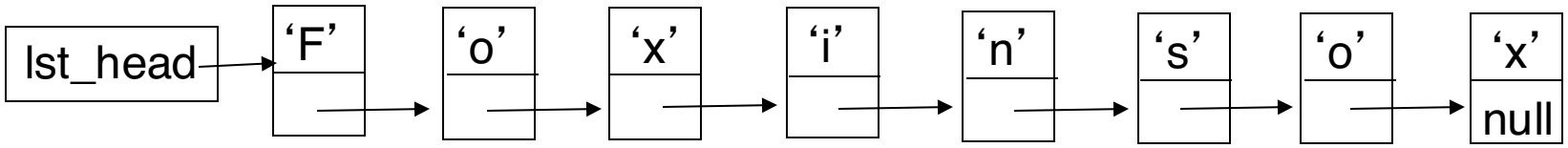
```
private static StringNode copy(StringNode str) {
    if (str == null) return null;
  StringNode copyFirst = new StringNode(str.ch);
  copyFirst.next = copy(str.next);
  return copyFirst;
 }
```



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

General Traversal Support: Iterators

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



- One possible implementation: use public LLString methods

```
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.get(i))
                numOccur++;
        }
        return numOccur;
    } …
```

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

Solution 1: Make numOccur() an 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;
    } …
```

```
public class MyClass {
    public int numOccur(LLString str, char ch) {
        return str.numOccur(ch);
    }
            …
}
```

- 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;
    } …
```

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

- Iterator as an inner class.

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

```
public class Iterator {
    private StringNode nextNode;
    private Iterator (){
        nextNode = head;
    }
    public boolean hasNext() {…}
    public char next() {…}
    …
}
```

- 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 new Exception("Falling off the list end");
        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 c = 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