

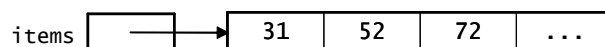
Linked Lists

Computer Science E-22
Harvard University

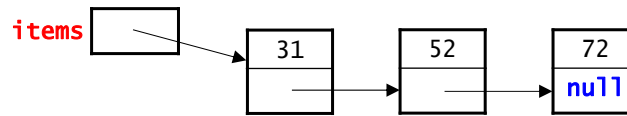
David G. Sullivan, Ph.D.

Representing a Sequence of Data

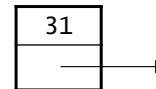
- Sequence – an ordered collection of items (position matters)
 - we will look at several types: lists, stacks, and queues
- Most common representation = an array
- Advantages of using an array:
 - easy and efficient access to *any* item in the sequence
 - `items[i]` gives you the item at position *i* in $O(1)$ time
 - known as *random access*
 - very compact (but can waste space if positions are empty)
- Disadvantages of using an array:
 - have to specify an initial array size and resize it as needed
 - inserting/deleting items can require shifting other items
 - ex: insert 63 between 52 and 72



Alternative Representation: A Linked List

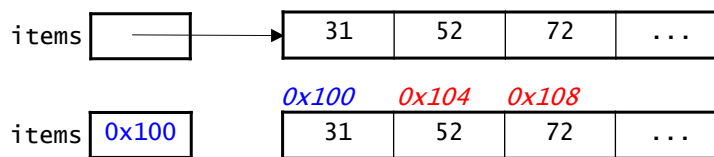


- A linked list stores a sequence of items in separate *nodes*.
- Each node is an object that contains:
 - a single item
 - a "link" (i.e., a reference) to the node containing the next item
- The last node in the linked list has a link value of null.
- The linked list as a whole is represented by a variable that holds a reference to the first node.
 - e.g., `items` in the example above

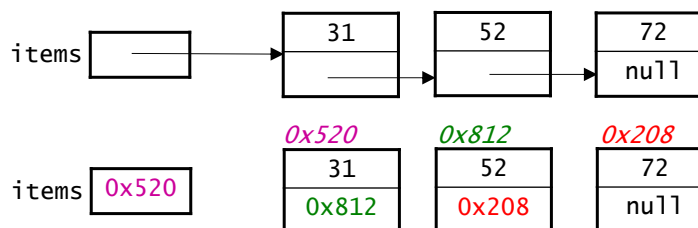


Arrays vs. Linked Lists in Memory

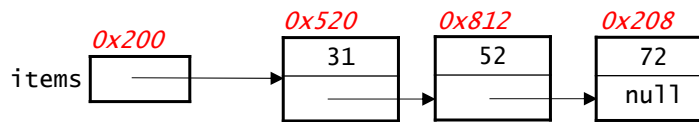
- In an array, the elements occupy consecutive memory locations:



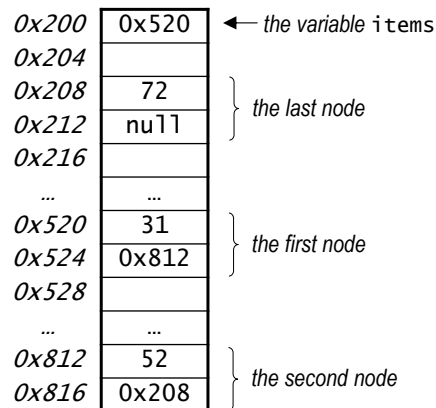
- In a linked list, the nodes are distinct objects.
 - do *not* have to be next to each other in memory
 - that's why we need the links to get from one node to the next!



Linked Lists in Memory

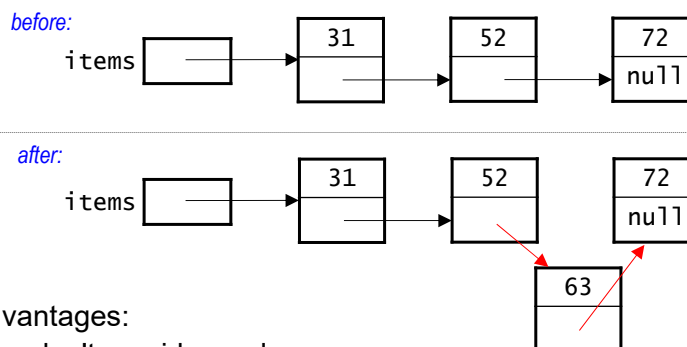


- Here's how the above linked list might actually look in memory:



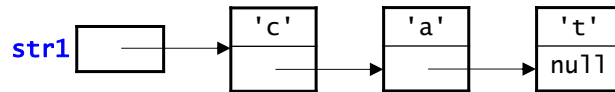
Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- Easy to insert/delete an item – no need to "shift over" other items.
 - for example, to insert 63 between 52 and 72:



- Disadvantages:
 - they don't provide random access
 - need to "walk down" the list to access an item
 - the links take up additional memory

A String as a Linked List of Characters



- Each node 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!

```
graph LR; Node["ch: 'c'<br/>next: StringNode"] --> Next["StringNode"];
```

- 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 String as a Linked List (cont.)

- An empty string will be represented by a null value.

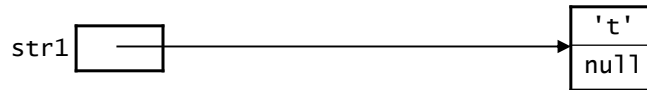
example:

```
StringNode str2 = null;
```

- We will use *static* methods that take the string as a parameter.
 - e.g., we'll write `length(str1)` instead of `str1.length()`
 - outside the class, call the methods using the class name:
`StringNode.length(str1)`
- This approach allows the methods to handle empty strings.
 - if `str1 == null`:
 - `length(str1)` will work
 - `str1.length()` will throw a `NullPointerException`

Building a Linked List of Characters I

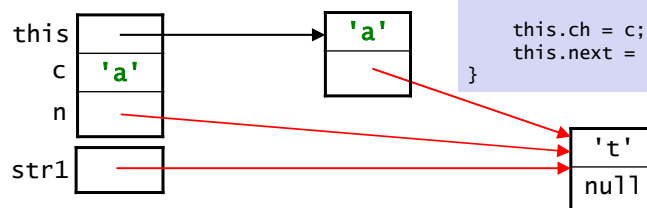
```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```



- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:
`StringNode str1 = new StringNode('t', null);`

Building a Linked List of Characters II

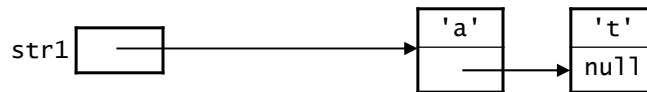
```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```



- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:
`StringNode str1 = new StringNode('t', null);`
`str1 = new StringNode('a', str1);`

Building a Linked List of Characters III

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

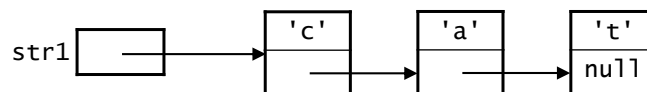


- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
```

Building a Linked List of Characters IV

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



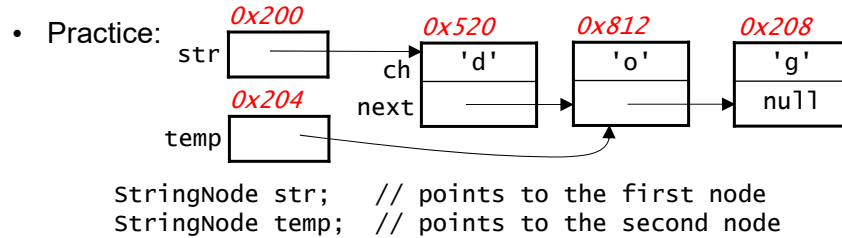
- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
str1 = new StringNode('c', str1);
```

- Later, we'll see methods that can be used to build a linked list and add nodes to it.

Review of Variables

- A variable or variable expression represents both:
 - a "box" or location in memory (the *address* of the variable)
 - the contents of that "box" (the *value* of the variable)

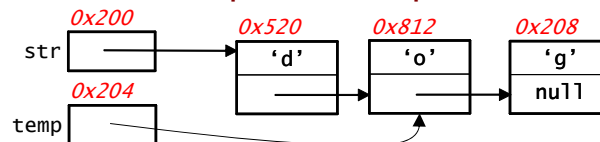


expression	address	value
str	0x200	0x520 (ref to the 'd' node)
str.ch		
str.next		

Assumptions:

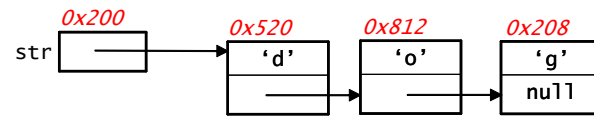
- ch field has the same memory address as the node itself.
- next field comes 2 bytes after the start of the node.

More Complicated Expressions



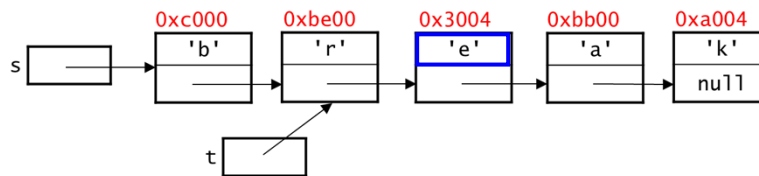
- Example: `temp.next.ch`
- Start with the beginning of the expression: `temp.next`
It represents the next field of the node to which `temp` refers.
 - address =
 - value =
- Next, consider `temp.next.ch`
It represents the `ch` field of the node to which `temp.next` refers.
 - address =
 - value =

What are the address and value of `str.next.next`?

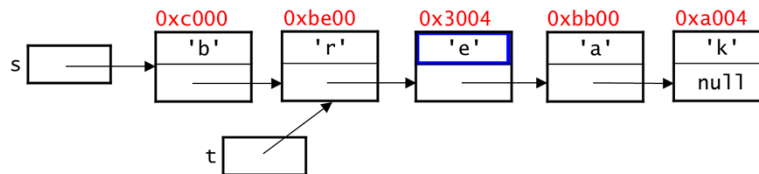


- `str.next` is...
- thus, `str.next.next` is...

What expression using `t` would give us 'e'?



What expression using t would give us 'e'?



Working backwards...

- I know that I need the ch field in the 'e' node
- Where do I have a reference to the 'e' node?
- What expression can I use for the box containing that reference?

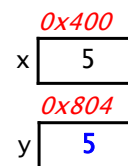
Review of Assignment Statements

- An assignment of the form
`var1 = var2;`
 - takes the value inside var2
 - copies it into var1

- Example involving integers:

```

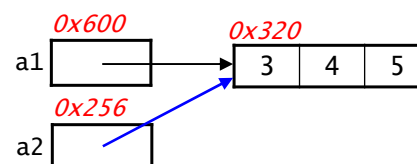
int x = 5;
int y = x;
      5
  
```



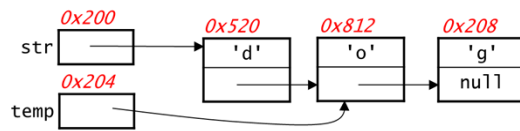
- Example involving references:

```

int[] a1 = {3, 4, 5};
int[] a2 = a1;
          0x320
  
```



What About These Assignments?



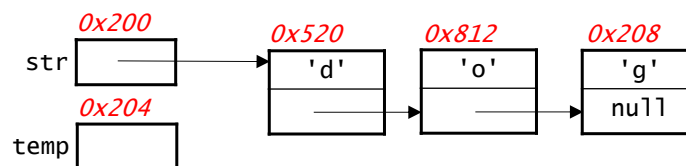
- Identify the two boxes.
- Determine the value in the box specified by the right-hand side.
- Copy that value into the box specified by the left-hand side.

1) `str.next = temp.next;`

2) `temp.next = temp.next.next;`

Writing an Appropriate Assignment

- If `temp` didn't already refer to the 'o' node, what assignment would be needed to make it refer to that node?



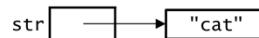
- start by asking: where do I currently have a reference to the 'o' node?
- then ask: what expression can I use for that box?
- then write the assignment:

A Linked List Is a Recursive Data Structure!

- Recursive definition: a linked list is either
 - a) empty or
 - b) a single node, followed by a linked list
- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

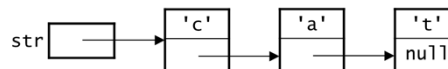
Recursively Finding the Length of a String

- For a Java String object:



```
public static int length(String str) {  
    if (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;  
    }  
}
```

An Alternative Version of the Method

- Original version:

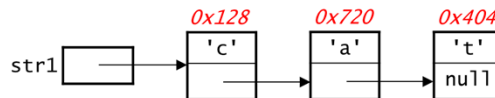
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

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

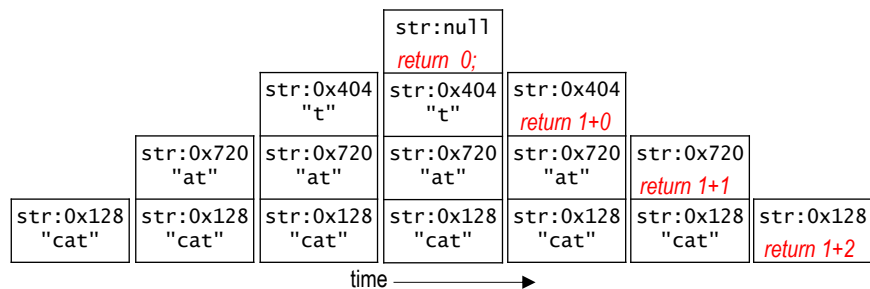
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```

Tracing length()

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

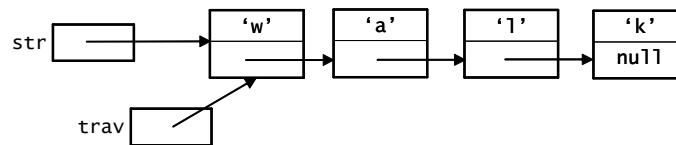


- Example: stringNode.length(str1)



Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or "walk down" a linked list.
- We just saw a method that used recursion to do this.
- It can also be done using iteration (for loops, while loops, etc.).
- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.

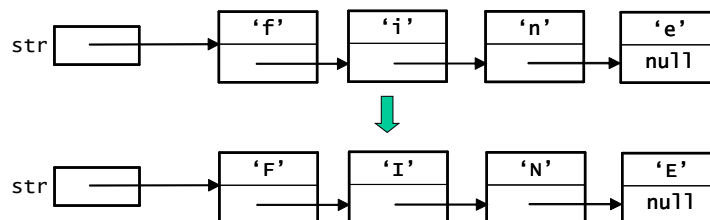


- Template for traversing an entire linked list:

```
StringNode trav = str;    // start with first node
while (trav != null) {
    // process the current node here
    trav = trav.next;    // move trav to next node
}
```

Example of Iterative Traversal

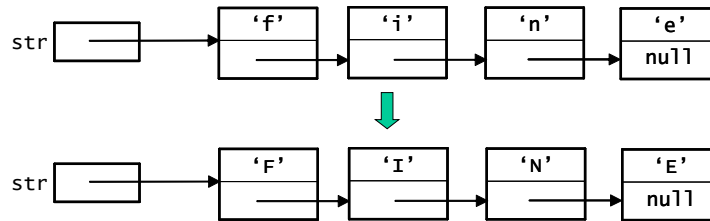
- `toUpperCase(str)`: converting `str` to all upper-case letters



- Similar to the built-in method for Java `String` objects.
- This method processes linked-list strings:
 - uses a loop to process one `StringNode` at a time
 - modifies the internals of the string (unlike the built-in version)
 - thus, it doesn't need to return anything

Example of Iterative Traversal (cont.)

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

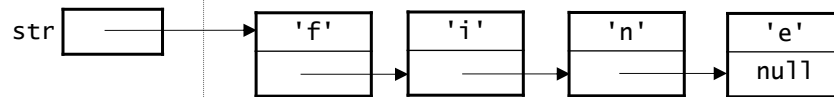


- Here's the method:

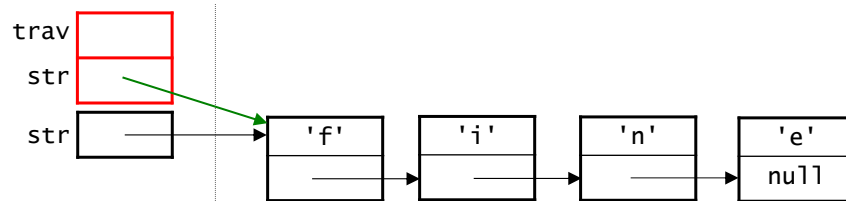
```
public static void toUpperCase(StringNode str) {  
    StringNode trav = str;  
    while (trav != null) {  
        trav.ch = Character.toUpperCase(trav.ch);  
        trav = trav.next;  
    }  
}
```

- uses a built-in static method from the Character class to convert a single char to upper case

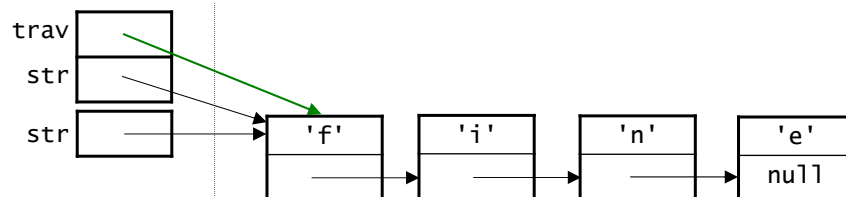
Tracing toUpperCase(): Before the Loop



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



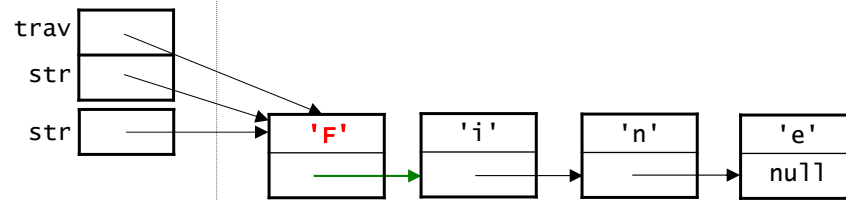
StringNode trav = str;



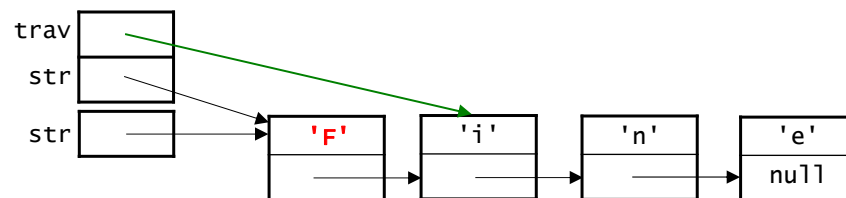
Tracing toUpperCase(): First Iteration of Loop

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

after updating trav.ch:



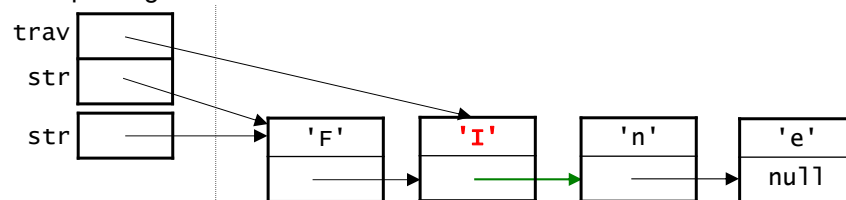
after updating trav:



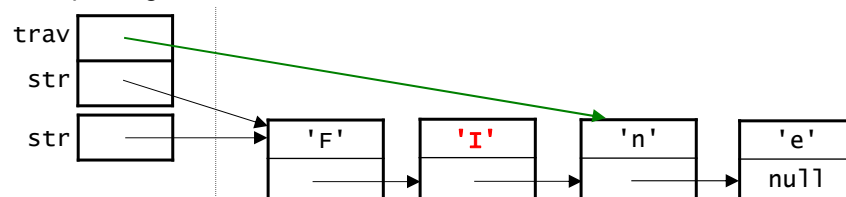
Tracing toUpperCase(): Second Iteration

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

after updating trav.ch:



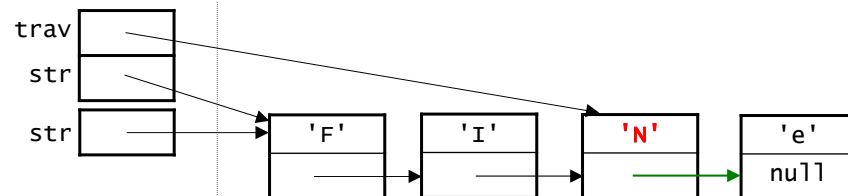
after updating trav:



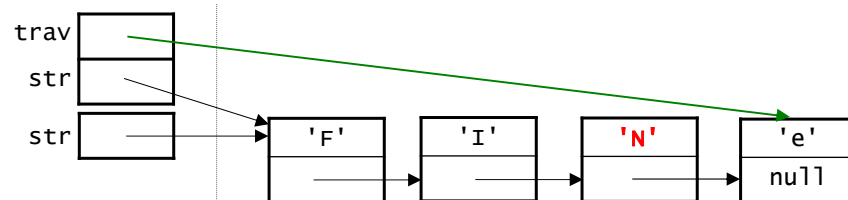
Tracing toUpperCase(): Third Iteration

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

after updating trav.ch:



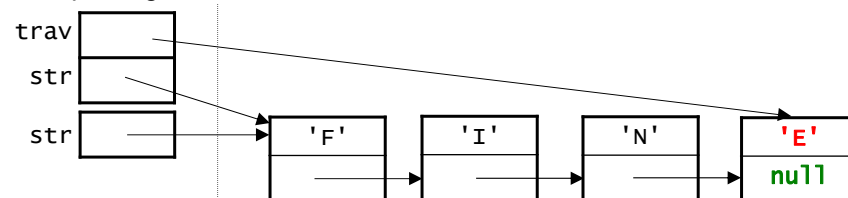
after updating trav:



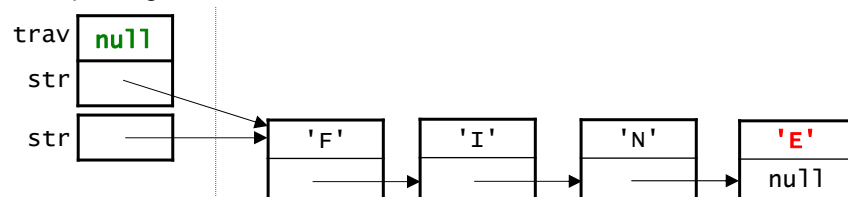
Tracing toUpperCase(): Fourth Iteration

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

after updating trav.ch:



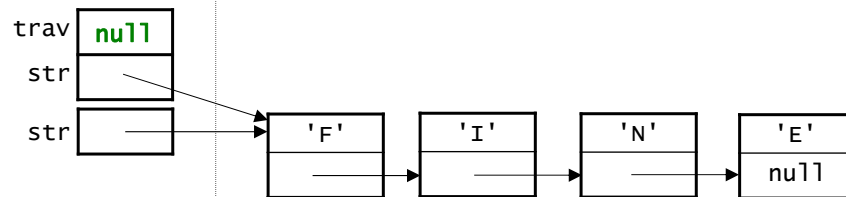
after updating trav:



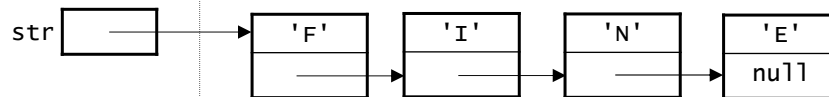
Tracing toupperCase(): Finishing Up

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

results of the final iteration:

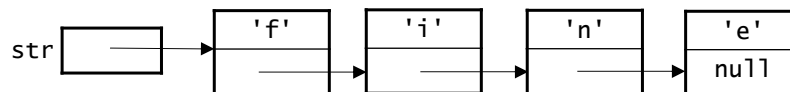


and now `trav == null`, so we end the loop and return:



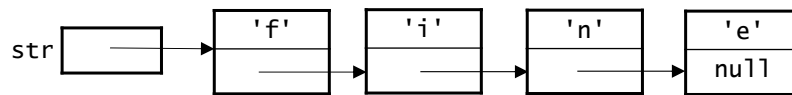
Getting the Node at Position *i* in a Linked List

- `getNode(str, i)` – should return a reference to the *i*th node in the linked list to which `str` refers



- Examples:
 - `getNode(str, 0)` should return a ref. to the 'f' node
 - `getNode(str, 3)` should return a ref. to the 'e' node
 - `getNode(str.next, 2)` should return a ref. to...?
- More generally, when $0 < i < \text{length of list}$,
`getNode(str, i)` is equivalent to `getNode(str.next, i-1)`

Getting the Node at Position i in a Linked List



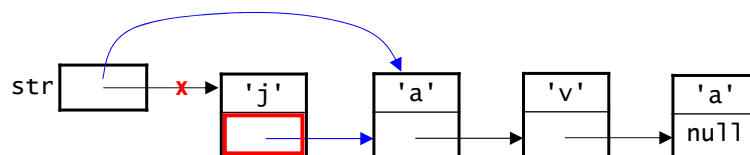
- Recursive approach to getNode(str, i):
 - if $i == 0$, return str (base case)
 - else call getNode(str.next, i-1) and return what it returns!
 - other base case?

- Here's the method:

```
private static StringNode getNode(StringNode str, int i) {  
    if (i < 0 || str == null) { // base case 1: no node i  
        return null;  
    } else if (i == 0) { // base case 2: just found  
        return str;  
    } else {  
        return getNode(str.next, i-1);  
    }  
}
```

Deleting the Item at Position i

- Special case: $i == 0$ (deleting the first item)
- Update our reference to the first node by doing:
 str = str.next;



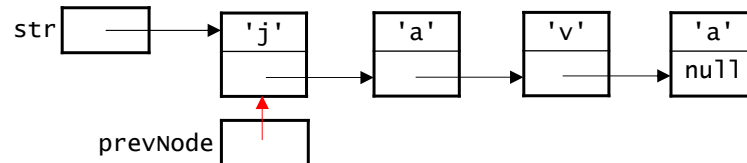
Deleting the Item at Position i (cont.)

- General case: $i > 0$

1. Obtain a reference to the *previous* node:

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

(example for $i == 1$)



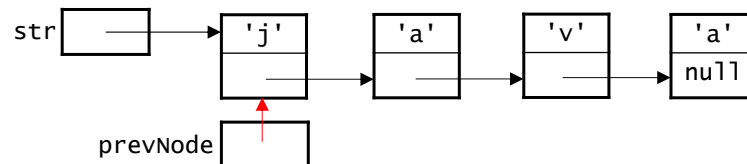
Deleting the Item at Position i (cont.)

- General case: $i > 0$

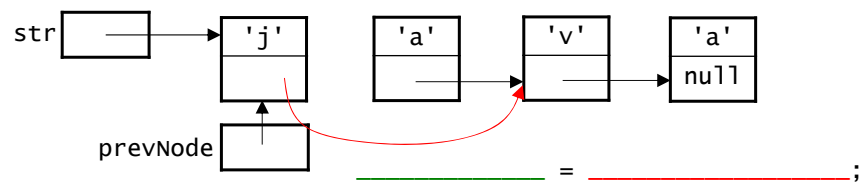
2. Update the references to remove the node

(example for $i == 1$)

before:



after:



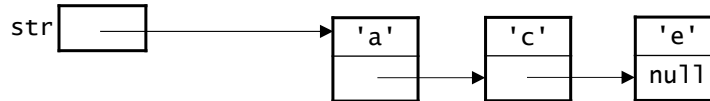
Inserting an Item at Position i

- Special case: $i == 0$ (insertion at the front of the list)
- Step 1: *Create the new node. Fill in the blanks!*

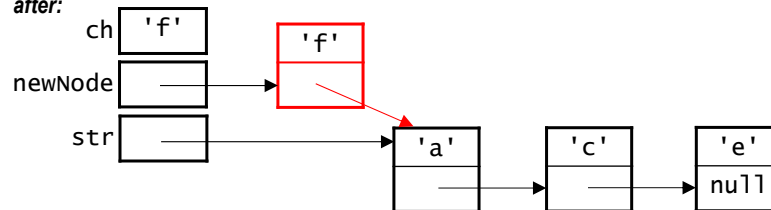
before:

ch

'f'



after:

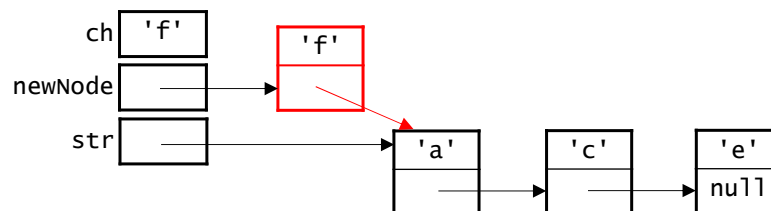


```
StringNode newNode = new StringNode(_____, _____);
```

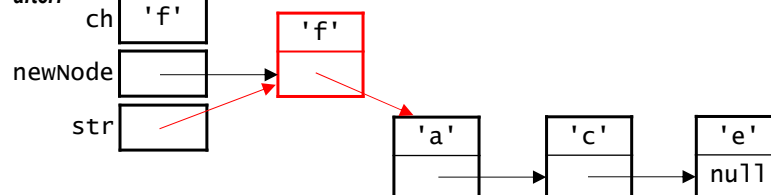
Inserting an Item at Position i (cont.)

- Special case: $i == 0$ (continued)
- Step 2: *Insert the new node. Write the assignment!*

before (result of previous slide):

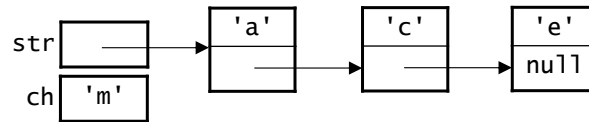


after:

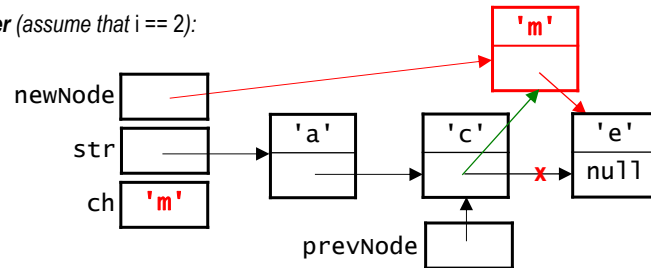


Inserting an Item at Position i (cont.)

- General case: $i > 0$ (insert *before* the item currently in posn i)
before:



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



```

StringNode prevNode = getNode(i - 1);
StringNode newNode = new StringNode(ch, _____);
_____ // one more line
  
```

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```

public static StringNode deleteChar(StringNode str, int i) {
    ...
    if (i == 0) {                // special case
        str = str.next;
    } else {                    // general case
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null) {
            prevNode.next = prevNode.next.next;
            ...
        }
        return str;
    }
}
  
```

- Clients should call them as part of an assignment:

```

s1 = StringNode.deleteChar(s1, 0);
s2 = StringNode.insertChar(s2, 0, 'h');
  
```

- If the first node changes, the client's variable will be updated to point to the new first node.

Creating a Copy of a Linked List

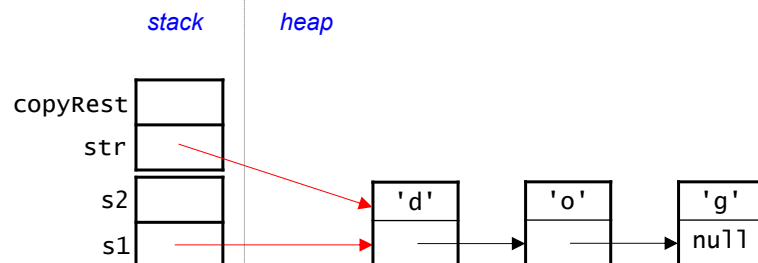
- `copy(str)` – create a copy of *the entire list* to which `str` refers
- Recursive approach:
 - base case: if `str` is empty, return `null`
 - else: – make a recursive call to copy the rest of the linked list
 - create and return a copy of the first node,
with its next field pointing to the copy of the rest

```
public static StringNode copy(StringNode str) {  
    if (str == null) {          // base case  
        return null;  
    }  
    // make a recursive call to copy the rest of the list  
    StringNode copyRest = copy(str.next);  
  
    // create and return a copy of the first node,  
    // with its next field pointing to the copy of the rest  
    return new StringNode(str.ch, copyRest);  
}
```

Tracing `copy()`: the initial call

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

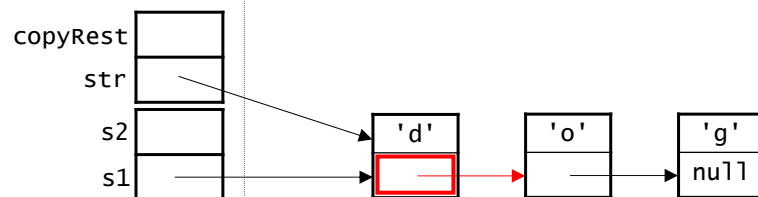
```
public static StringNode copy(StringNode str) {  
    if (str == null) {  
        return null;  
    }  
    StringNode copyRest = copy(str.next);  
    return new StringNode(str.ch, copyRest);  
}
```



Tracing copy(): the initial call

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

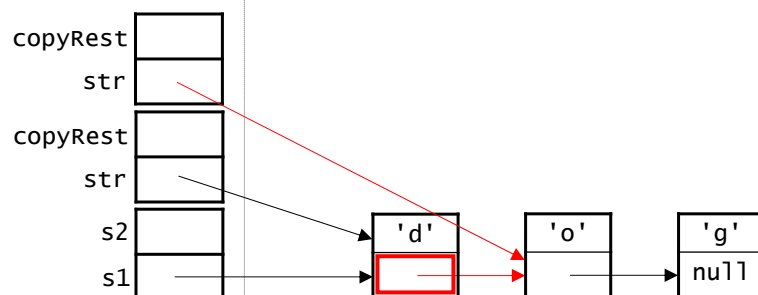
```
public static StringNode copy(StringNode str) {  
    if (str == null) {  
        return null;  
    }  
    StringNode copyRest = copy(str.next);  
    return new StringNode(str.ch, copyRest);  
}
```



Tracing copy(): the initial call

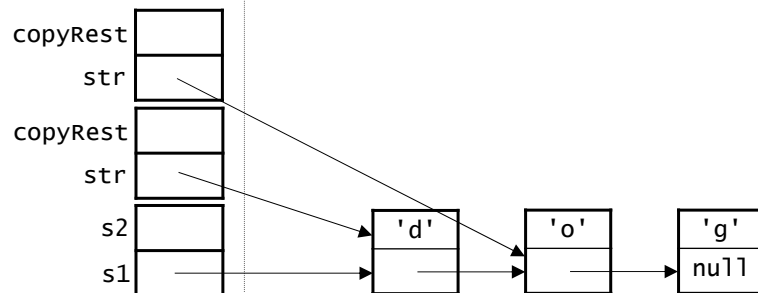
- From a client: `StringNode s2 = StringNode.copy(s1);`

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



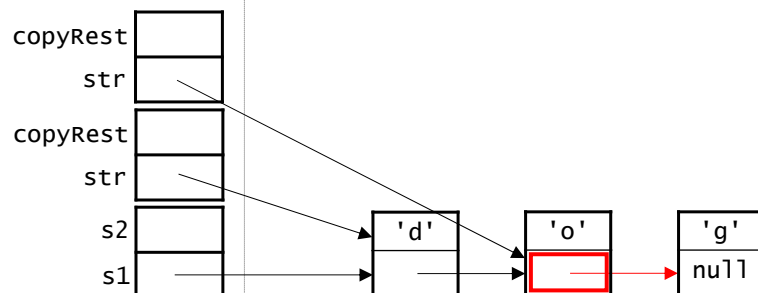
Tracing copy(): the recursive calls

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



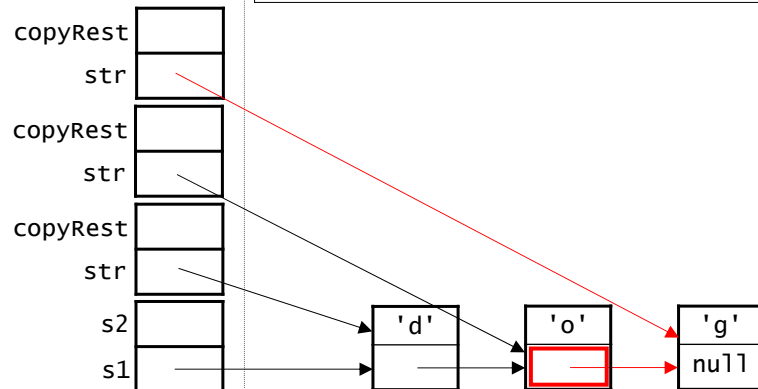
Tracing copy(): the recursive calls

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



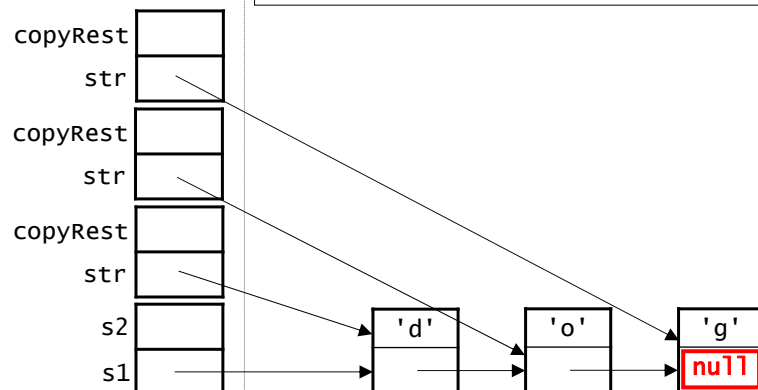
Tracing copy(): the recursive calls

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



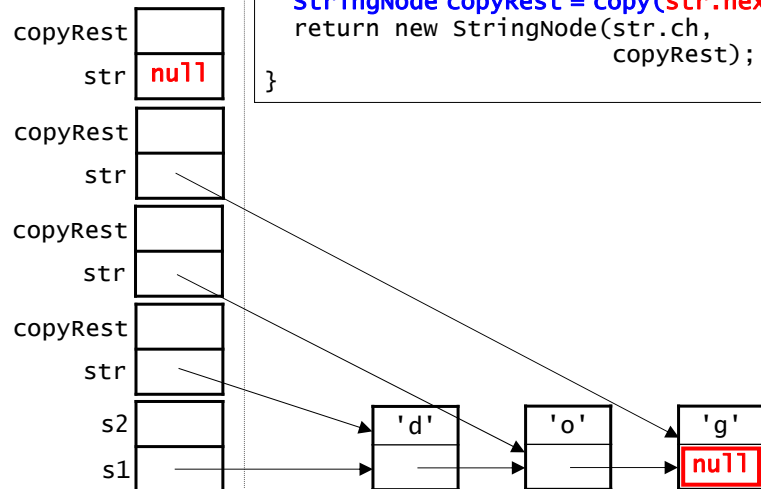
Tracing copy(): the recursive calls

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



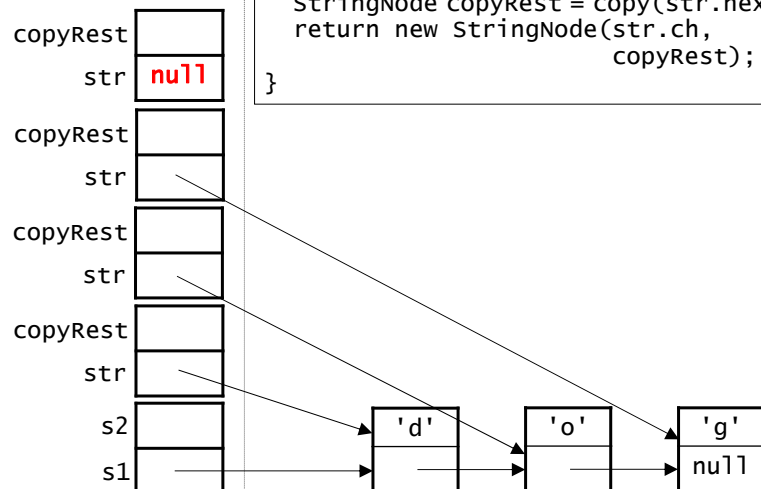
Tracing copy(): the recursive calls

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



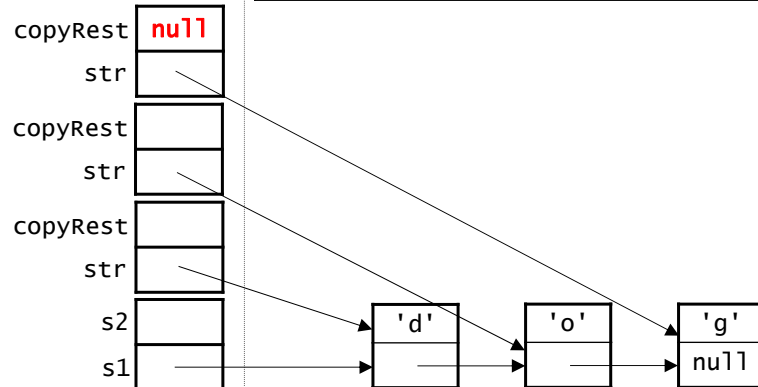
Tracing copy(): the base case

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



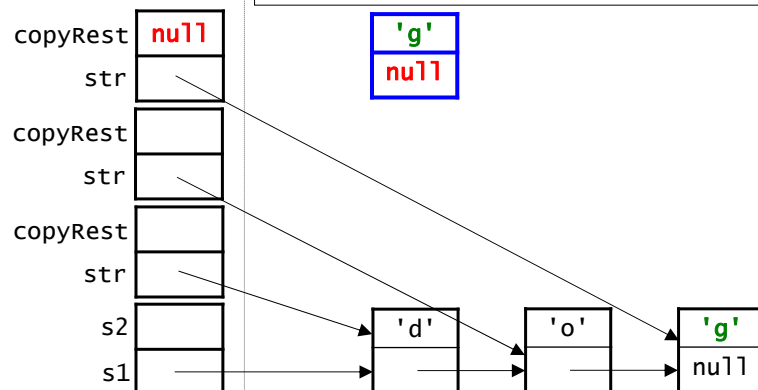
Tracing copy(): returning from the base case

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



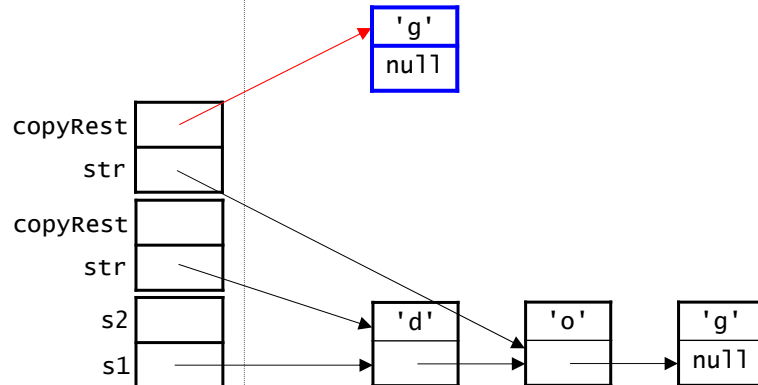
Tracing copy(): returning from the base case

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



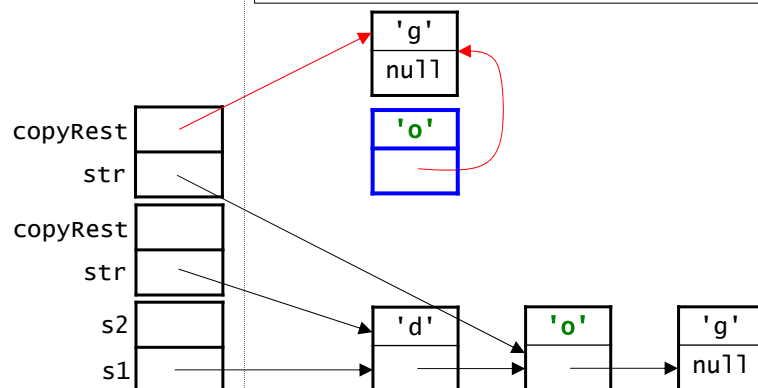
Tracing copy(): returning from the base case

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



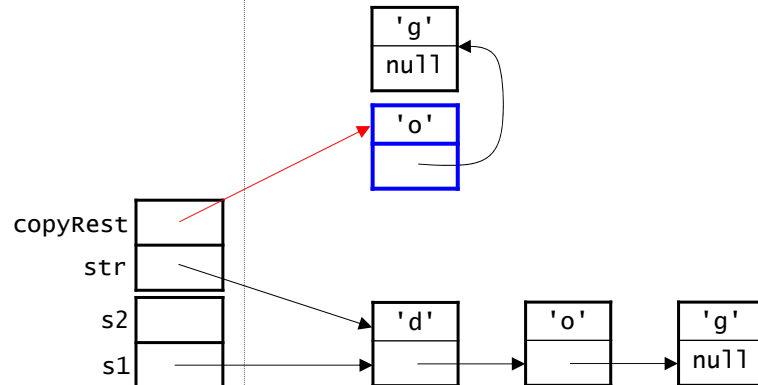
Tracing copy(): returning from the base case

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



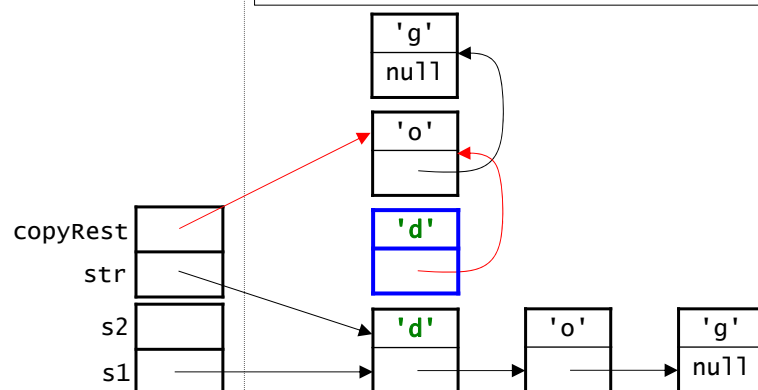
Tracing copy(): returning from the base case

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



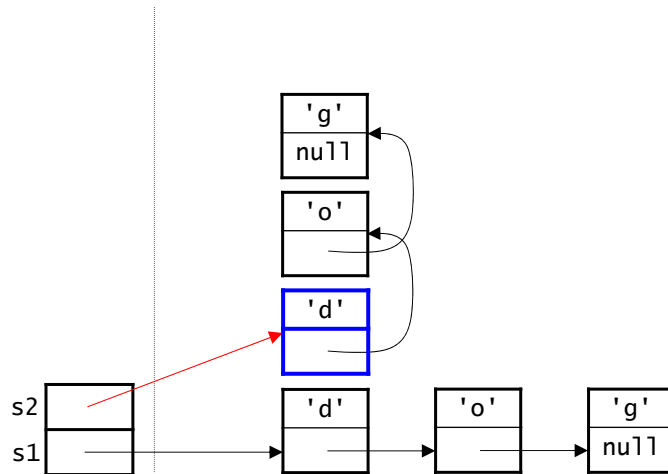
Tracing copy(): returning from the base case

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



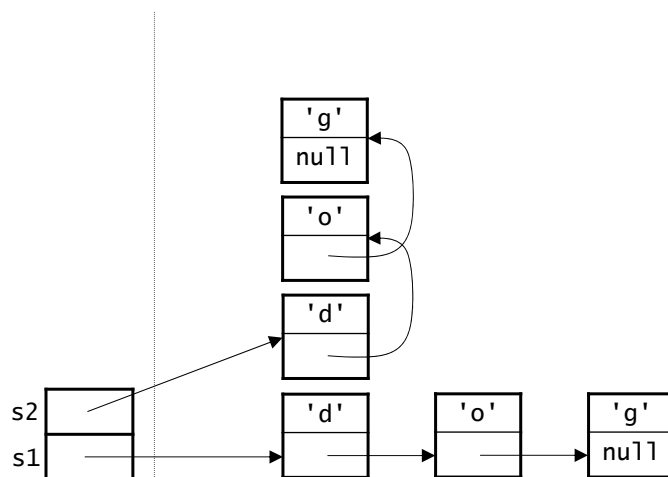
Tracing copy(): returning from the base case

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



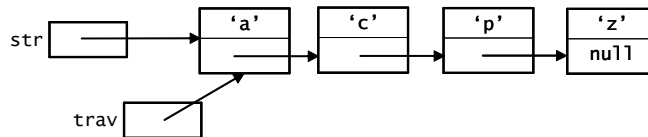
Tracing copy(): Final Result

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



Using a "Trailing Reference" During Traversal

- When traversing a linked list, one trav may not be enough.
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



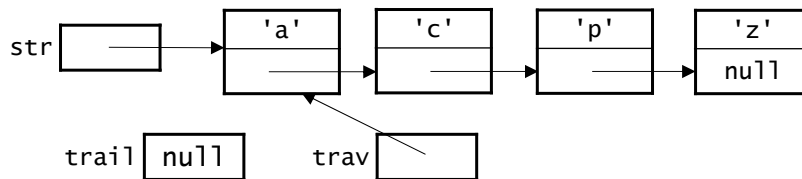
- Traverse the list to find the right position:

```
StringNode trav = str;
while (trav != null && trav.ch < ch) {
    trav = trav.next;
}
```
- When we exit the loop, where will `trav` point? Can we insert 'n'?
- The following changed version doesn't work either. Why not?

```
while (trav != null && trav.next.ch < ch) {
    trav = trav.next;
}
```

Using a "Trailing Reference" (cont.)

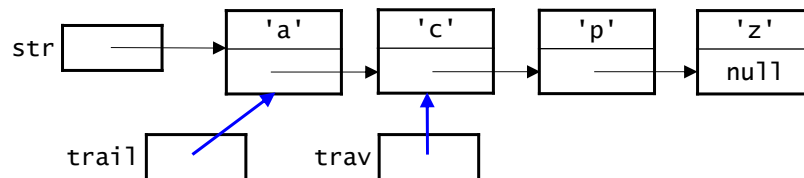
- To get around the problem seen on the previous page, we traverse the list using two different references:
 - `trav`, which we use as before
 - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

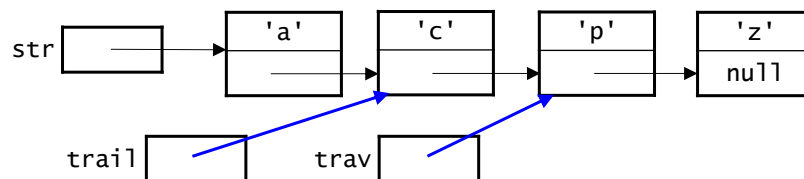
- To get around the problem seen on the previous page, we traverse the list using two different references:
 - trav, which we use as before
 - trail, which stays one node behind trav



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

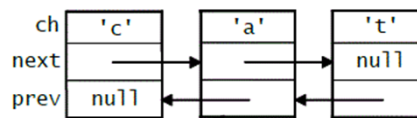
Using a "Trailing Reference" (cont.)

- To get around the problem seen on the previous page, we traverse the list using two different references:
 - trav, which we use as before
 - trail, which stays one node behind trav



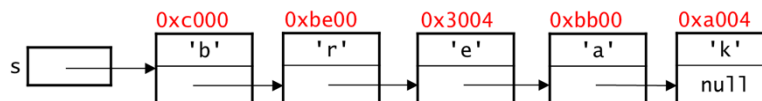
```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Doubly Linked Lists



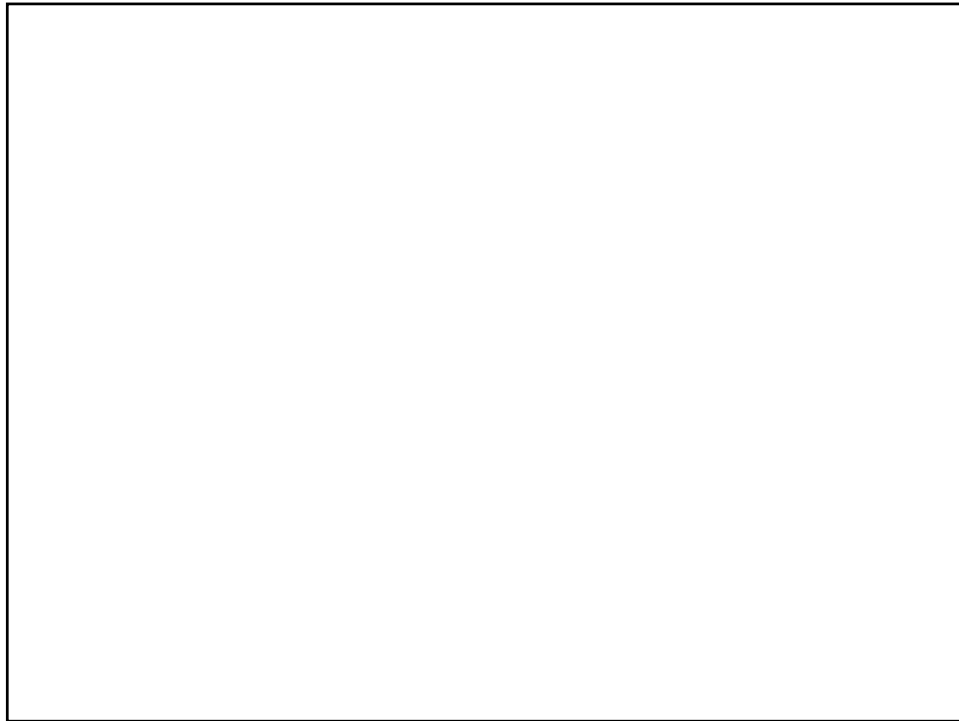
- In a doubly linked list, every node stores *two* references:
 - next, which works the same as before
 - prev, which holds a reference to the previous node
 - in the first node, prev has a value of null
- The prev references allow us to "back up" as needed.
 - remove the need for a trailing reference during traversal!
- Insertion and deletion must update both types of references.

Find the address and value of `s.next.next.ch`

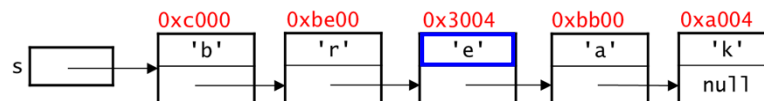


Extra practice!

- | | <u>address</u> | <u>value</u> |
|----|----------------|--------------|
| A. | 0xbe00 | 'r' |
| B. | 0x3004 | 'e' |
| C. | 0xbb00 | 'a' |
| D. | none of these | |



Find the address and value of `s.next.next.ch`



- `s.next` is the next field in the node to which `s` refers
 - it holds a reference to the 'r' node
- thus, `s.next.next` is the next field in the 'r' node
 - it holds a reference to the 'e' node
- thus, `s.next.next.ch` is the ch field in the 'e' node
 - it holds the 'e' !

	<u>address</u>	<u>value</u>
A.	0xbe00	'r'
B.	0x3004	'e'
C.	0xbb00	'a'
D.	none of these	