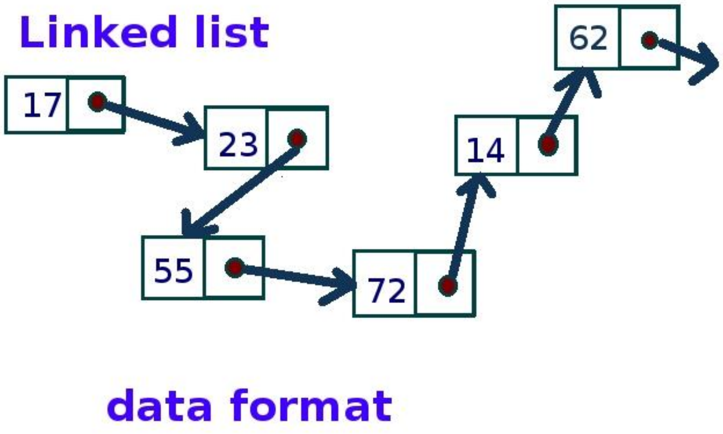


Linked Lists: An Overview

Computer Science CS112
Boston University

Christine Papadakis-Kanaris

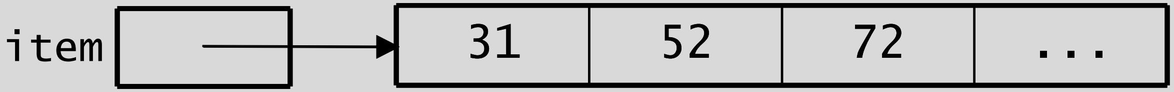
Representing a Sequence of Data

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues

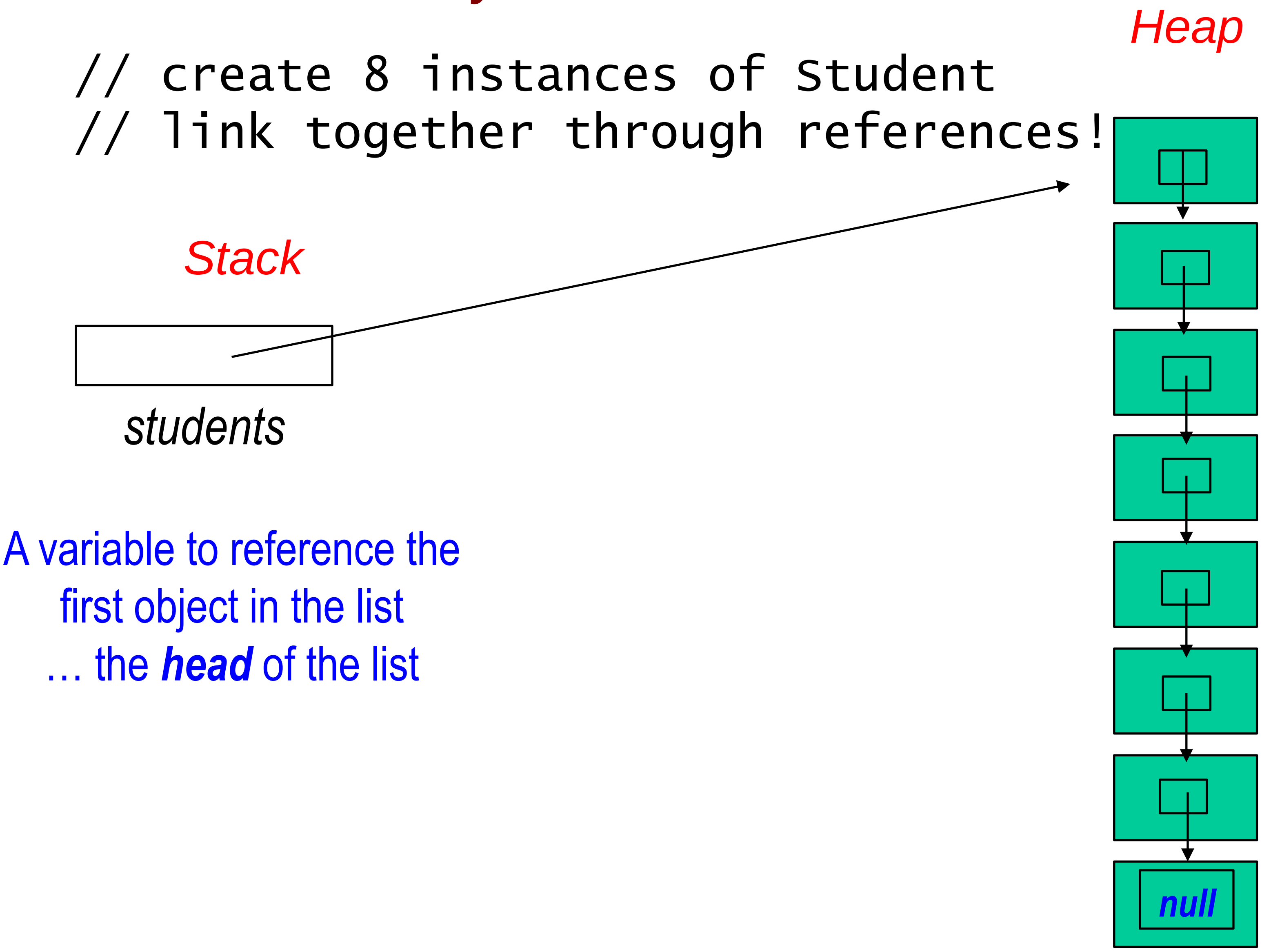
- Most common representation = an array

- Advantages of using an array:
  - easy and efficient access to *any* item in the sequence
    - `item[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



Linked List:
a dynamic Data structure

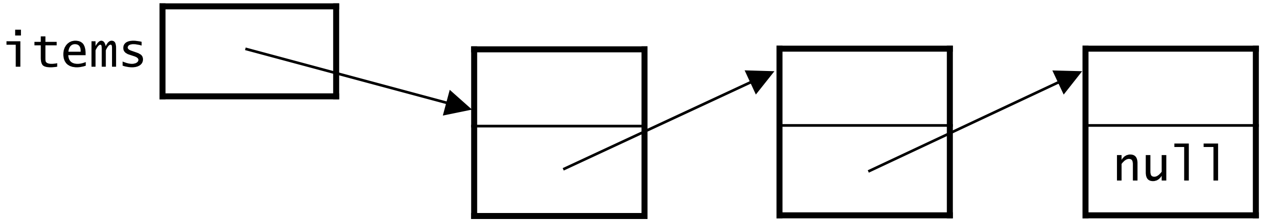


```
// create 8 instances of Student
// link together through references!
```

A variable to reference the first object in the list … the ***head*** of the list

A Linked List

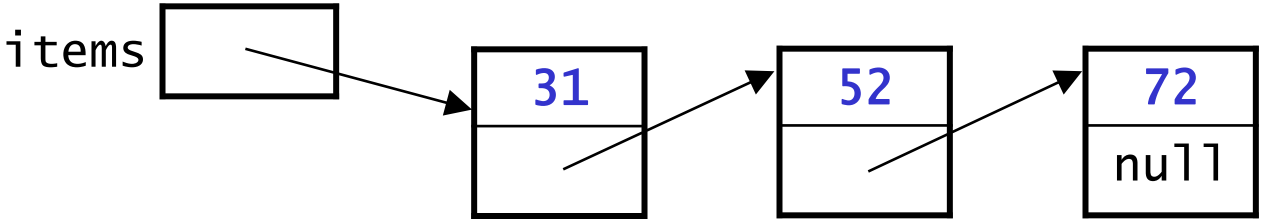
- Example:



- A linked list stores a sequence of items in separate *nodes*.

A Linked List of …

- Example:



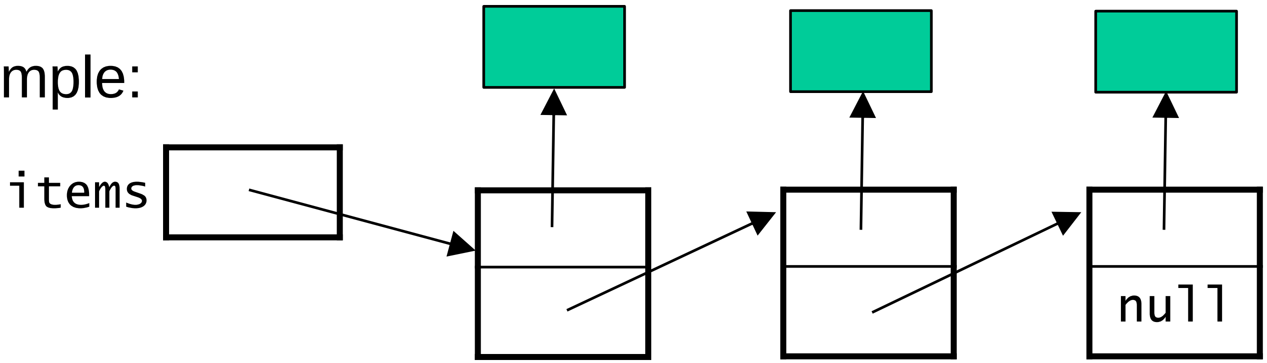
- A linked list stores a sequence of items in separate *nodes*.

- Each node contains:
  - a single *data* item

Note that the item can be a *primitive* variable or a ……………..……

A Linked List of …

- Example:



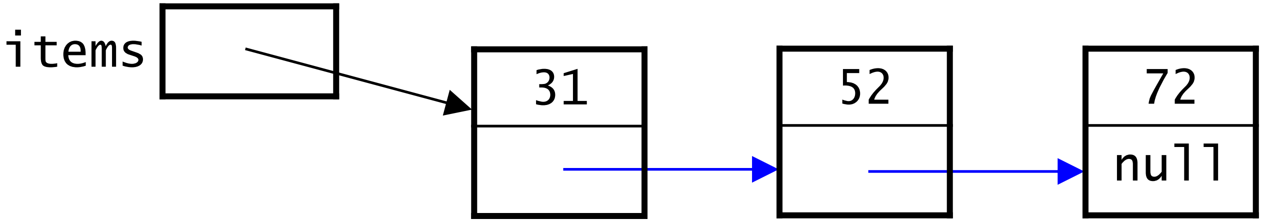
- A linked list stores a sequence of items in separate *nodes*.

- Each node contains:
  - a single *data* item

Note that the item can be a *primitive* variable or a *reference* to an *object*!

A Linked List

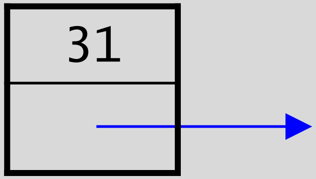
- Example:



- A linked list stores a sequence of items in separate *nodes*.

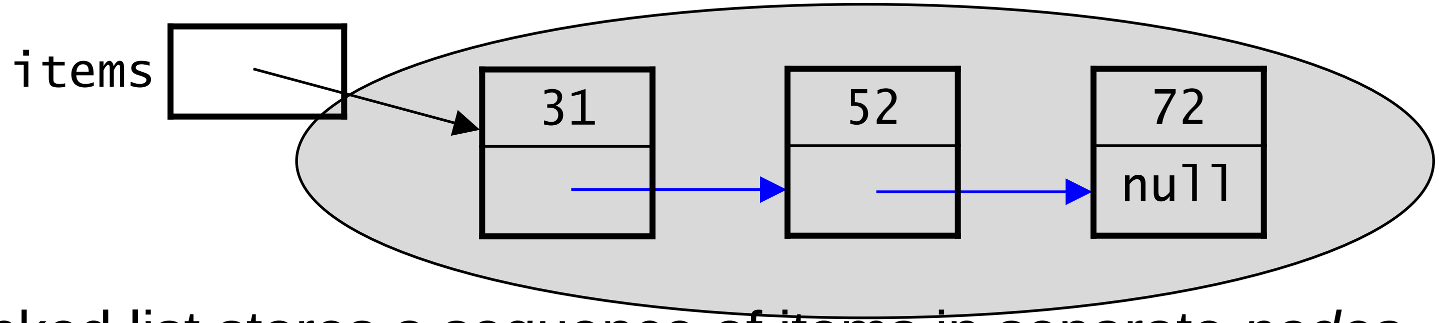
- Each node contains:
  - a single *data* item
  - a "link" (i.e., a reference) to the node containing the next item

example node:



A Linked List

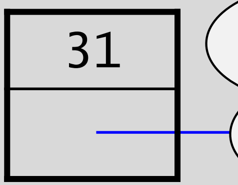
- Example:



- A linked list stores a sequence of items in separate *nodes*.
- Each node contains:
  - a single *data* item
  - a "link" (i.e., a referenc

em

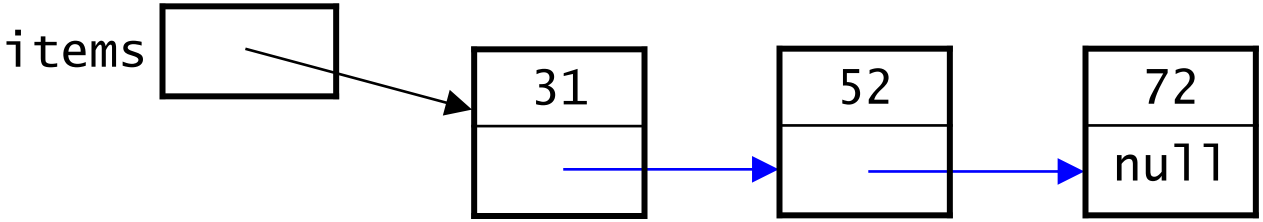
example node:



The *nodes* of the list form the sequence…

A Linked List

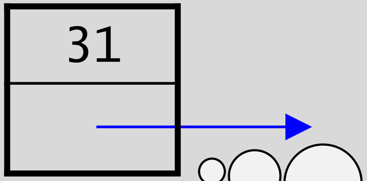
- Example:



- A linked list stores a sequence of items in separate *nodes*.

- Each node contains:
  - a single *data* item
  - a "link" (i.e., a reference) to the node containing the next item

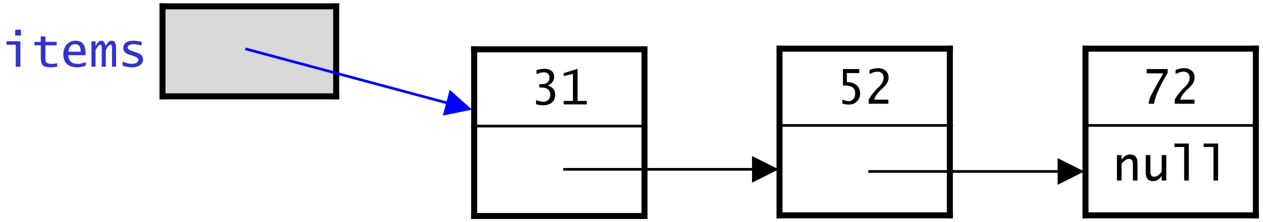
example node:



… and the *references* are the links which form the chain.

A Linked List

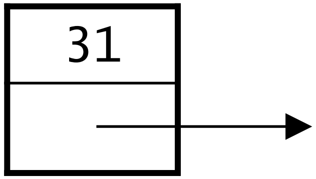
- Example:



- A linked list stores a sequence of items in separate *nodes*.

- Each node contains:
  - a single *data* item
  - a "link" (i.e., a reference) to the node containing the next item

example node:

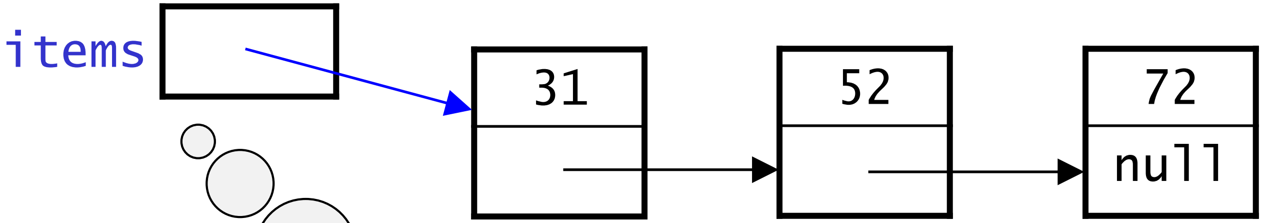


- The last node in the linked list has a link value of `null`.

- 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).

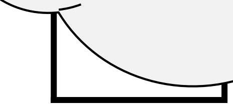
A Linked List

- Example:



- A linked list stor in separate *nodes*.
- Each node c
  - a single
  - a "link" (ntaining the next item

example no

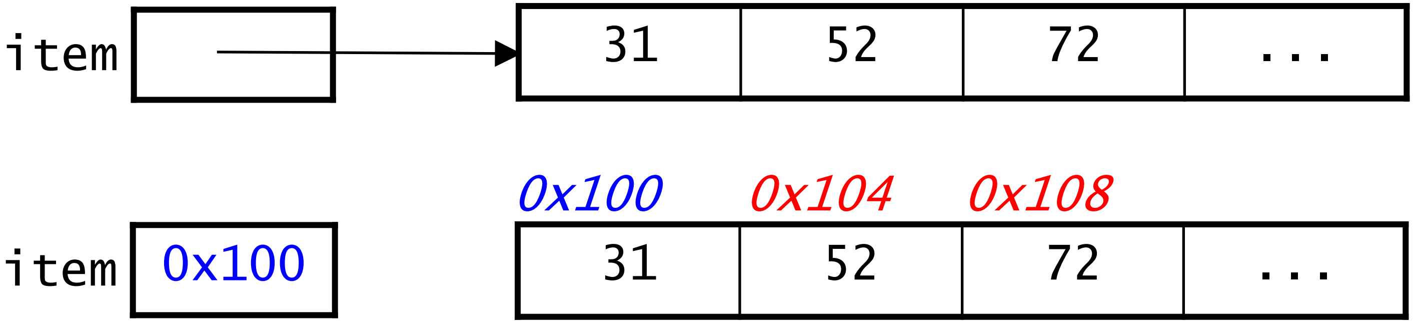


Can be referred to as the *head* or the variable that *references* the “*head of the list*”.

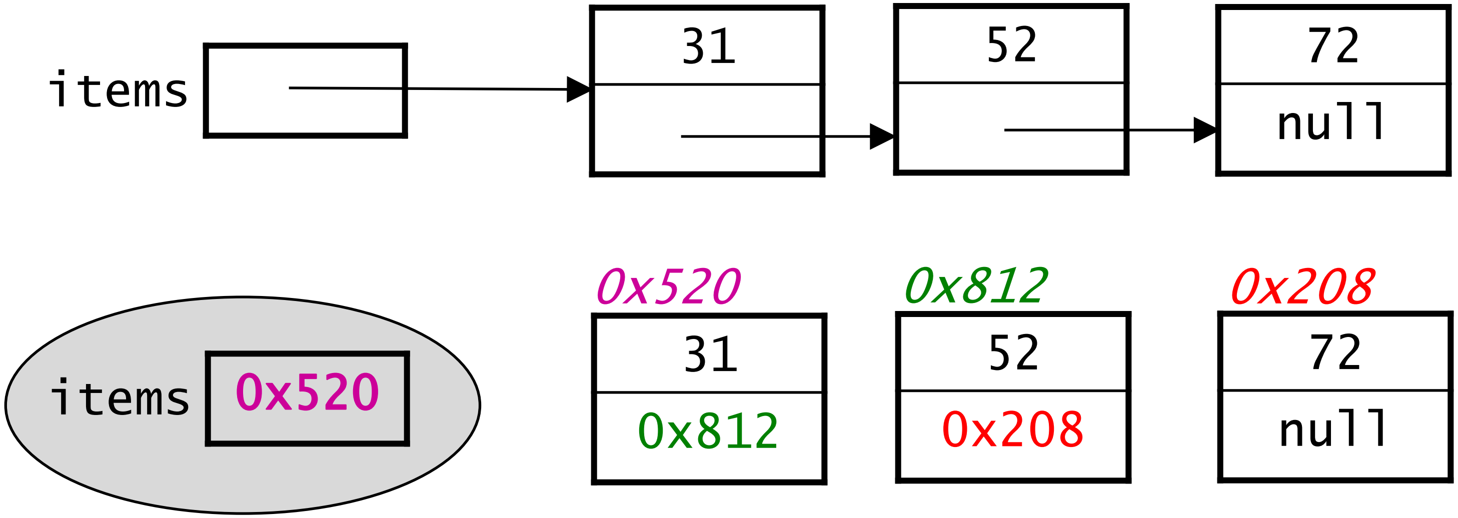
- The last node in the linked list has a link value of `null`.
- The linked list as a whole is represented by a variable that holds a reference to the first node (e.g., `items` in the example above).

Arrays vs. Linked Lists in Memory

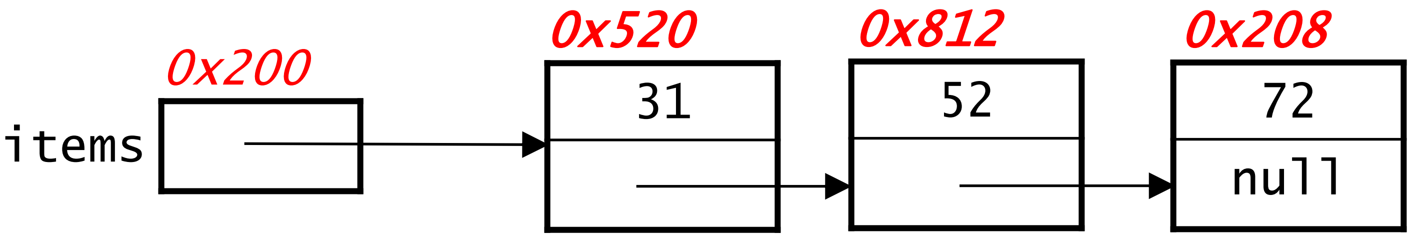
- In an array, the elements occupy consecutive memory locations:



- In a linked list, each node is a *distinct object* on the heap. The nodes 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.



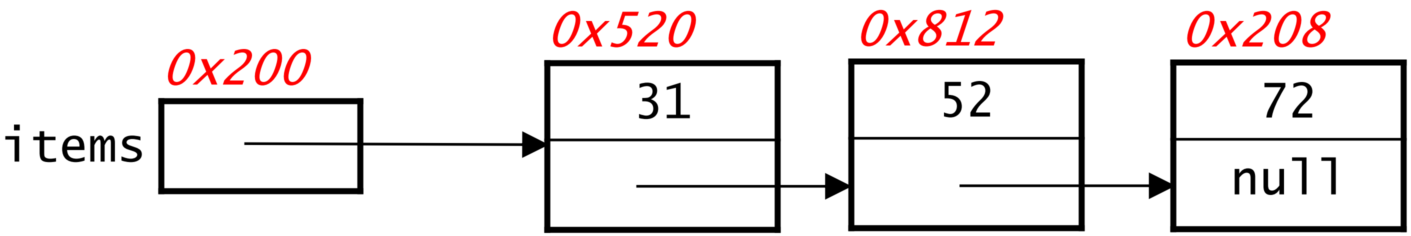
Linked Lists in Memory



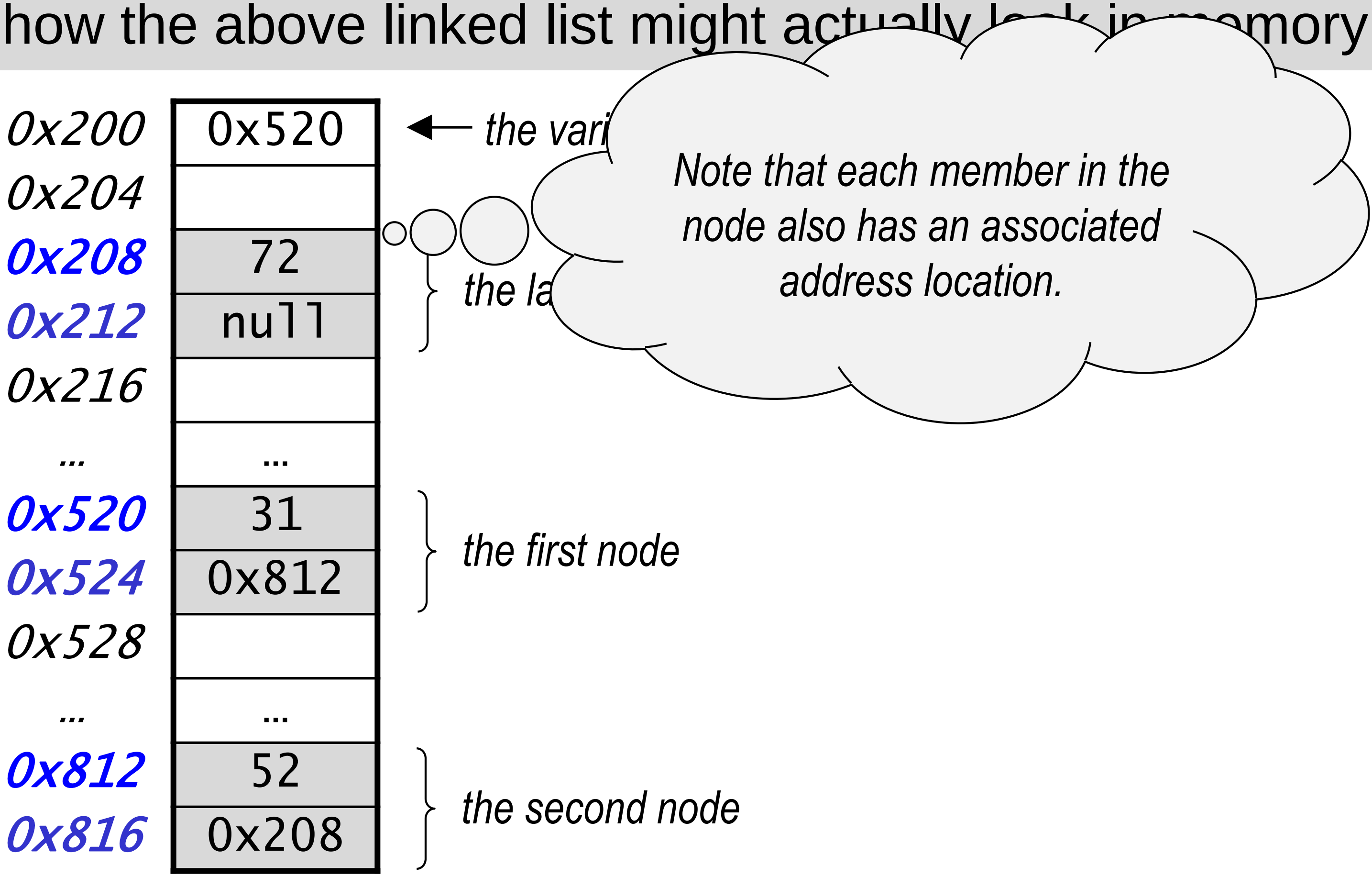
- Here's how the above linked list might actually look in memory:

0x200	0x520	← *the variable* `items`
0x204		
0x208	72	*the last node*
0x212	null	
0x216		
...	...	
0x520	31	*the first node*
0x524	0x812	
0x528		
...	...	
0x812	52	*the second node*
0x816	0x208	

Linked Lists in Memory



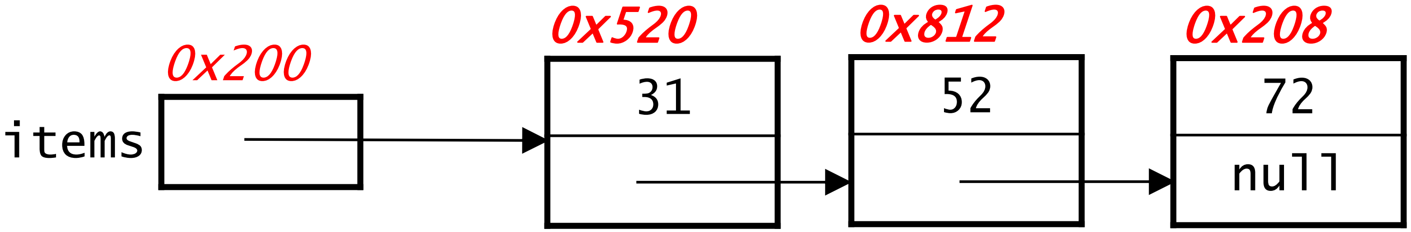
- Here's how the above linked list might actually look in memory:



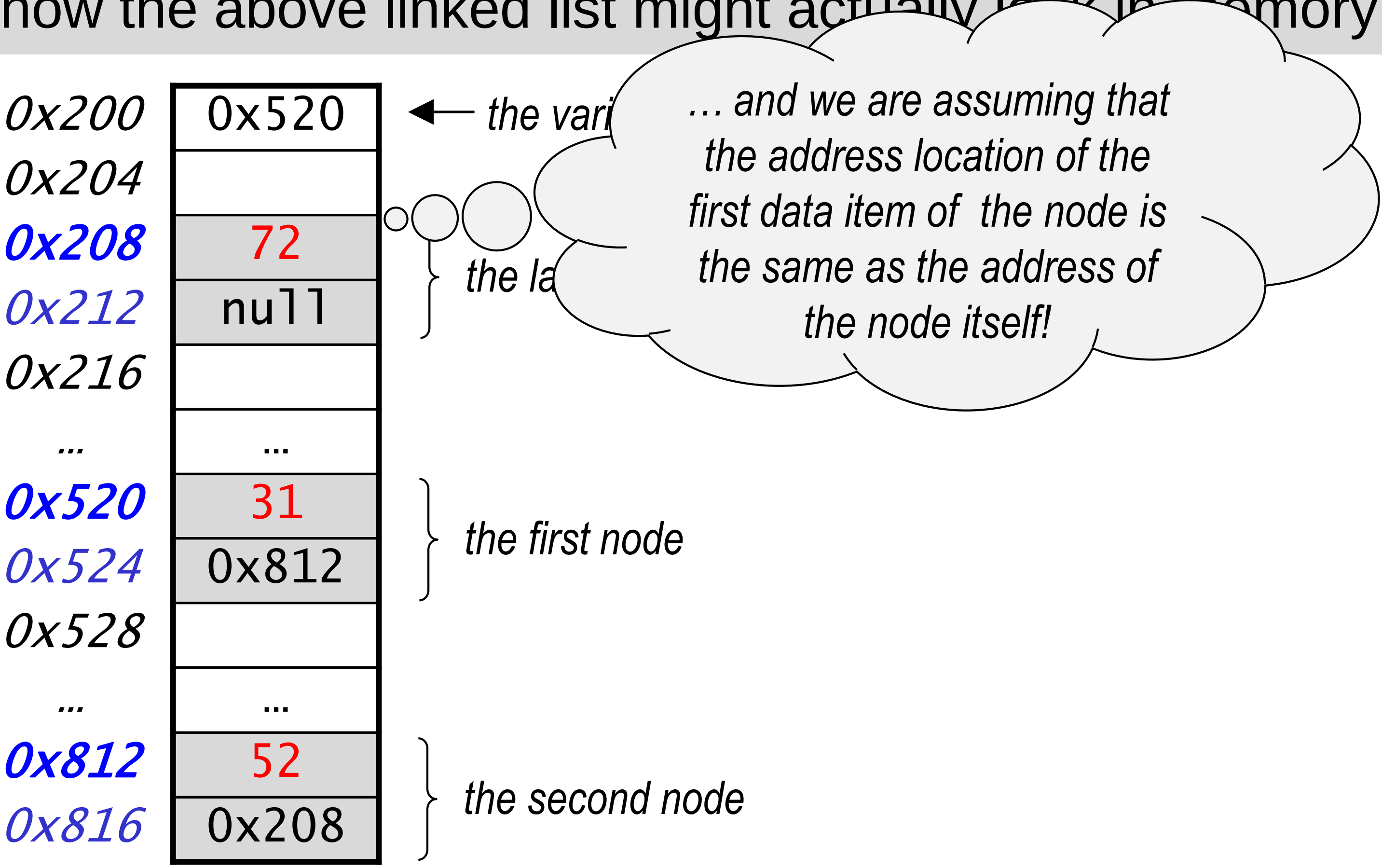
0x200	0x520
0x204	
0x208	72
0x212	null
0x216	
…	…
0x520	31
0x524	0x812
0x528	
…	…
0x812	52
0x816	0x208

Note that each member in the node also has an associated address location.

Linked Lists in Memory



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



0x200	0x520
0x204	
0x208	72
0x212	null
0x216	
…	…
0x520	31
0x524	0x812
0x528	
…	…
0x812	52
0x816	0x208

the vari

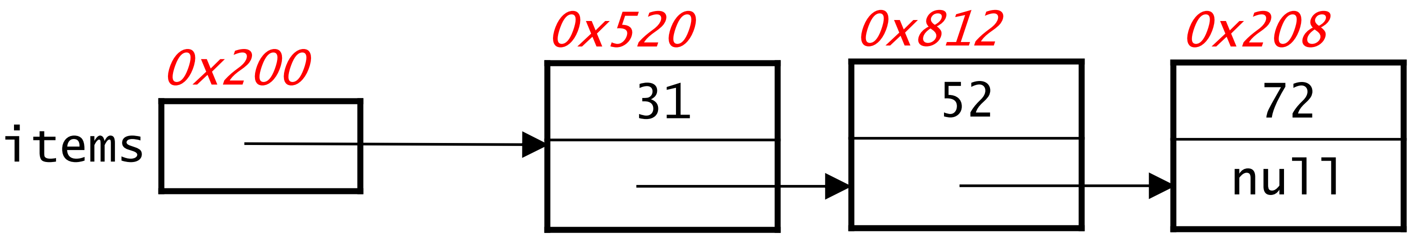
the la

the first node

the second node

… and we are assuming that the address location of the first data item of the node is the same as the address of the node itself!

Linked Lists in Memory



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

0x200	0x520
0x204	
0x208	72
0x212	null
0x216	
…	…
0x520	31
0x524	0x812
0x528	
…	…
0x812	52
0x816	0x208

the vari

the la

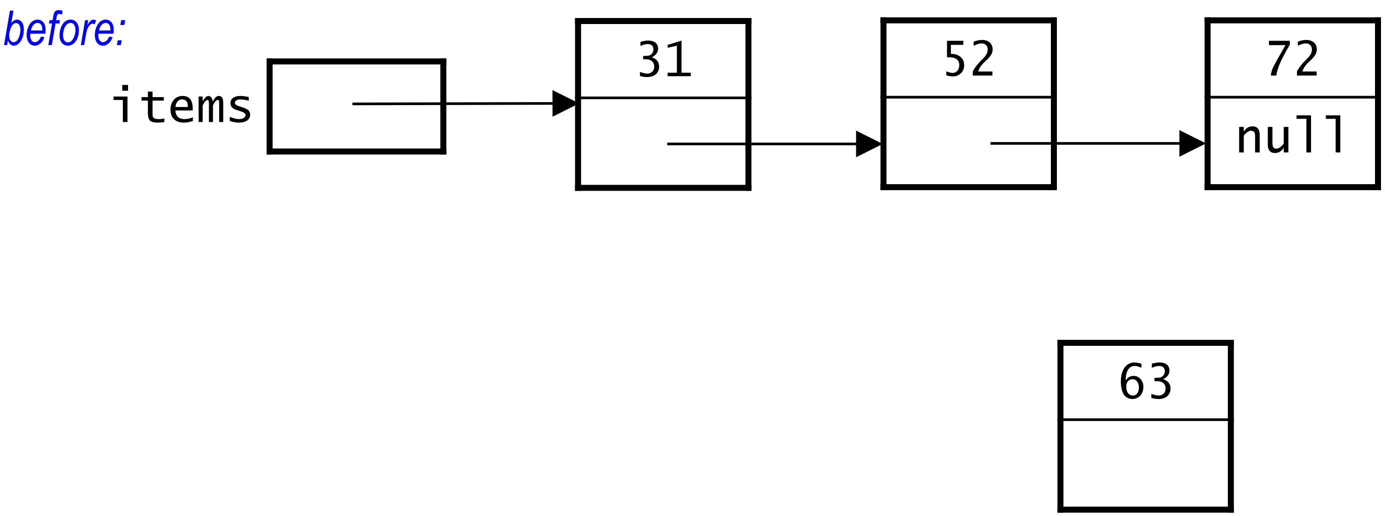
the first node

the second node

Since the data item is an integer, we assume the address location of the next member is 4 bytes away.

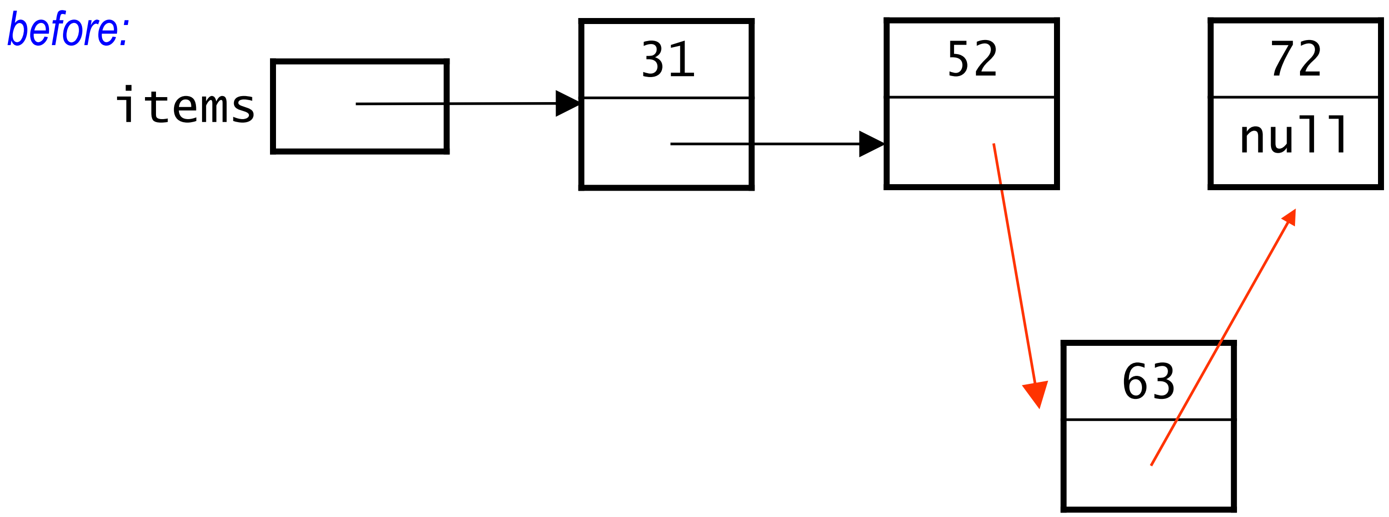
Features of Linked Lists

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



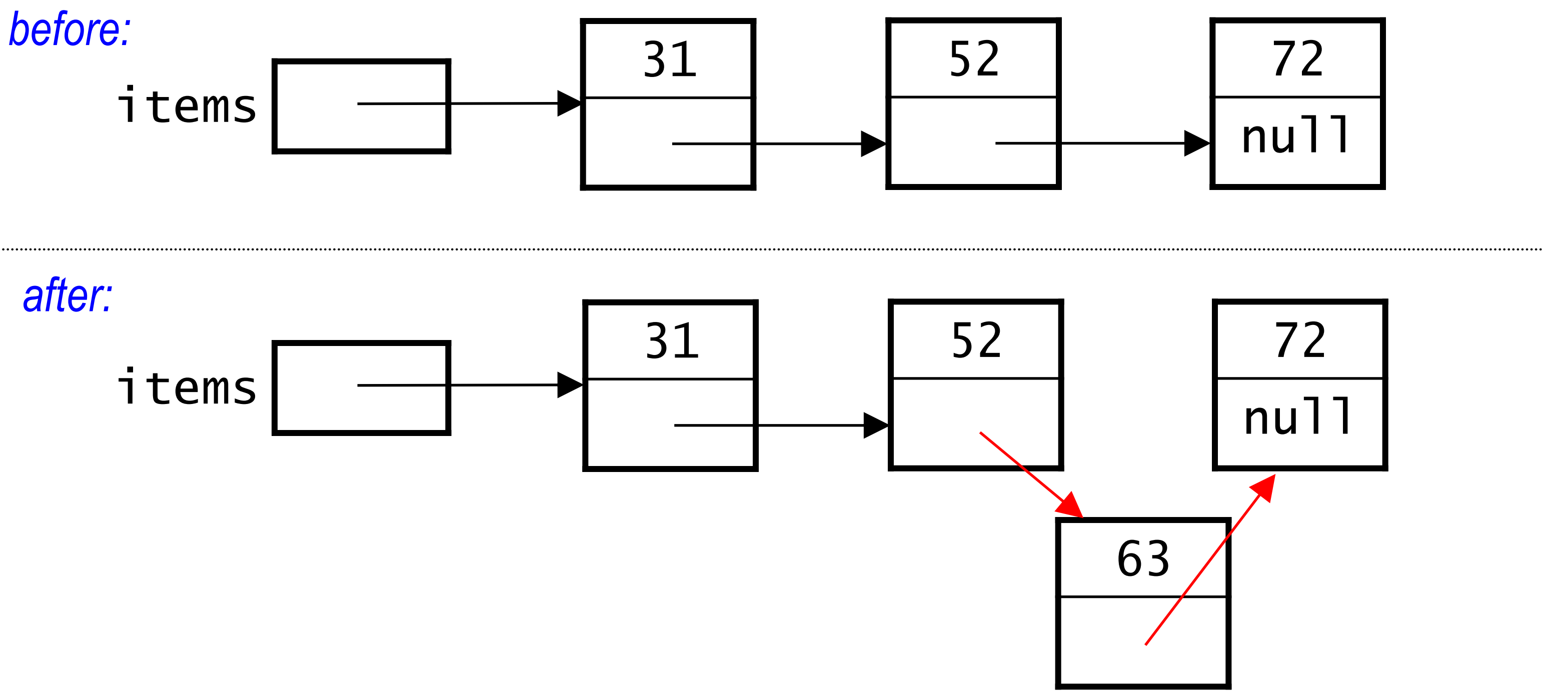
Features of Linked Lists

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



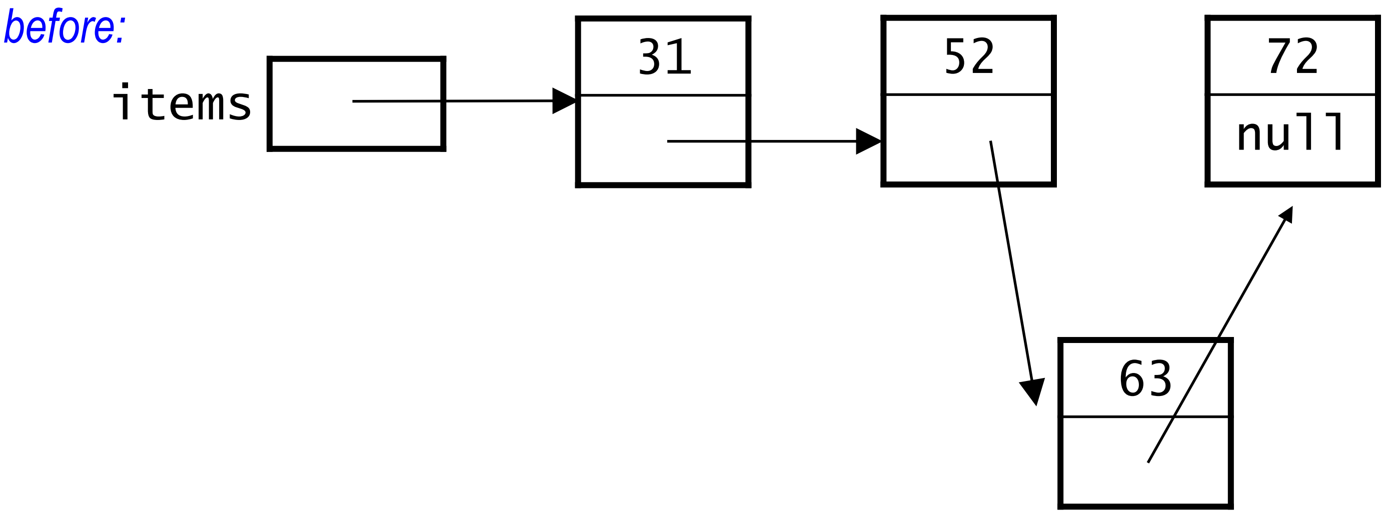
Features of Linked Lists

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



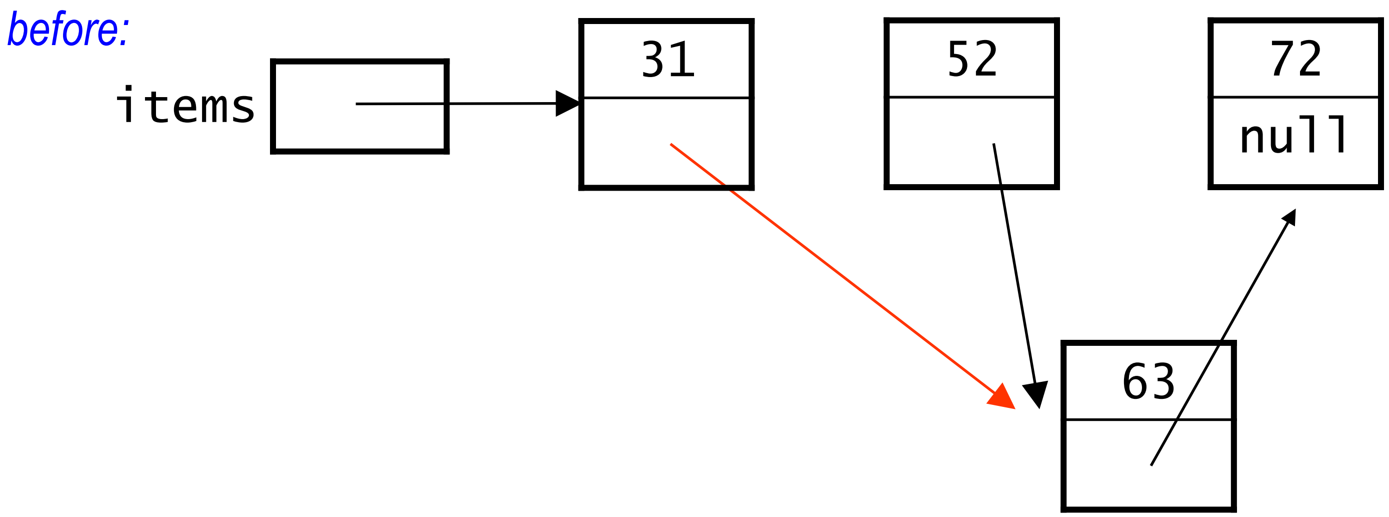
Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- To delete an item – also no need to "shift over" other items.
  - for example, to delete node 52:



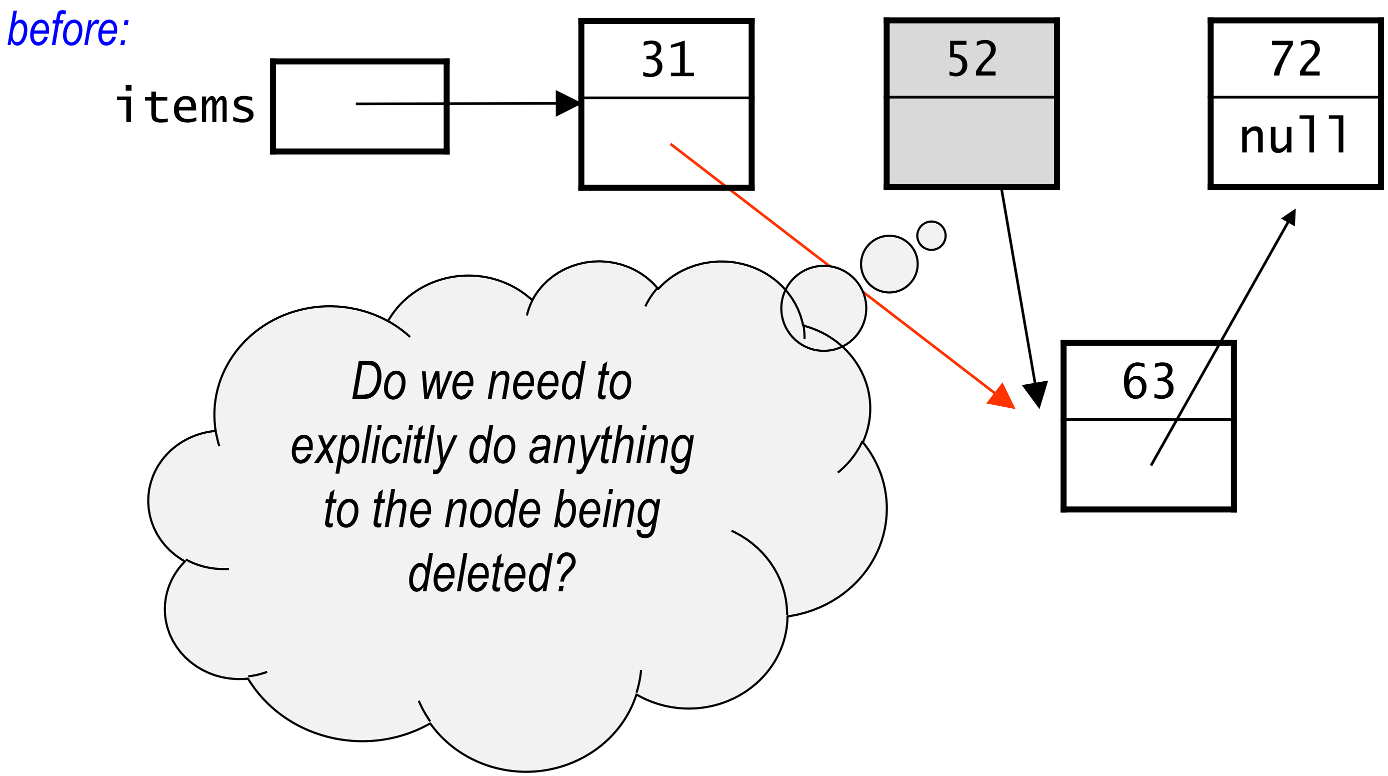
Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- To delete an item – also no need to "shift over" other items.
  - for example, to delete node 52:



Features of Linked Lists

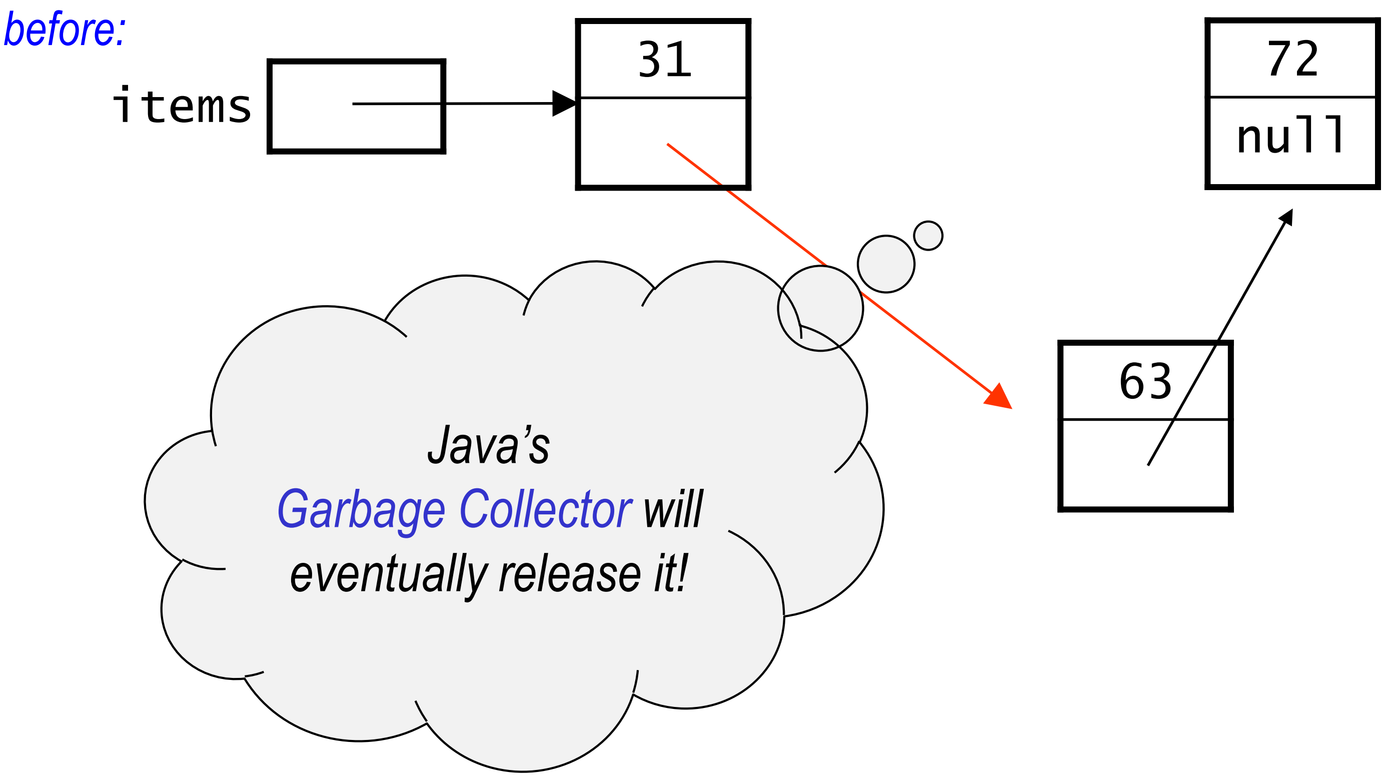
- They can grow without limit (provided there is enough memory).
- To delete an item – also no need to "shift over" other items.
    - for example, to delete node 52:



Do we need to explicitly do anything to the node being deleted?

Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- To delete an item – also no need to "shift over" other items.
  - for example, to delete node 52:



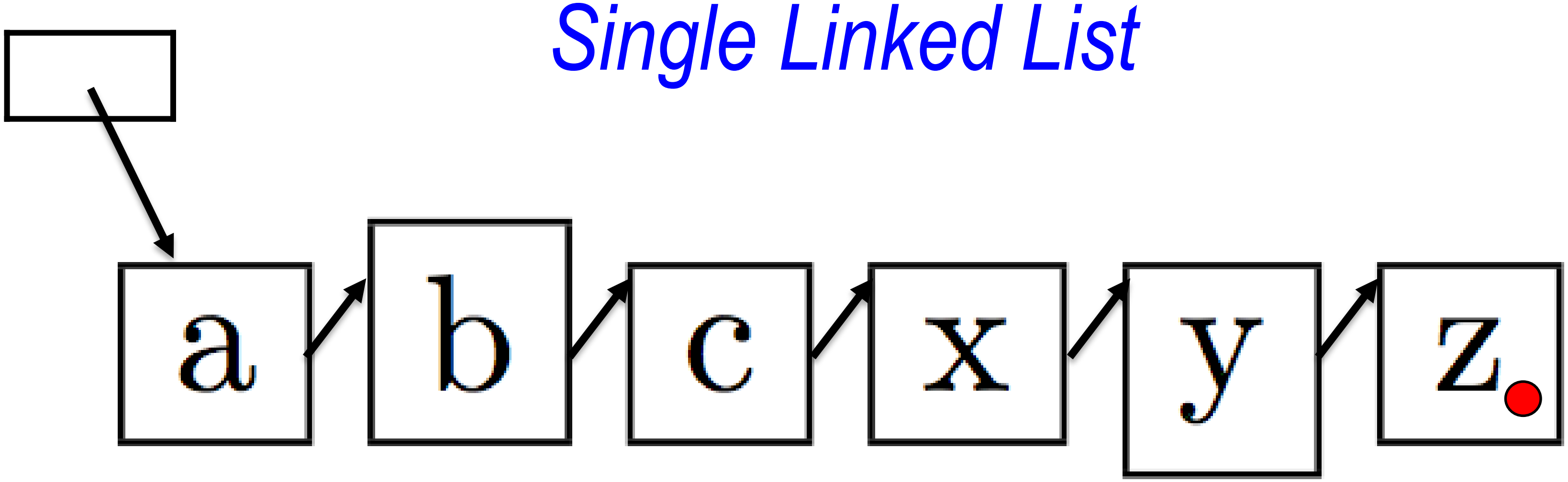
Features of Linked Lists

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

Case Study

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

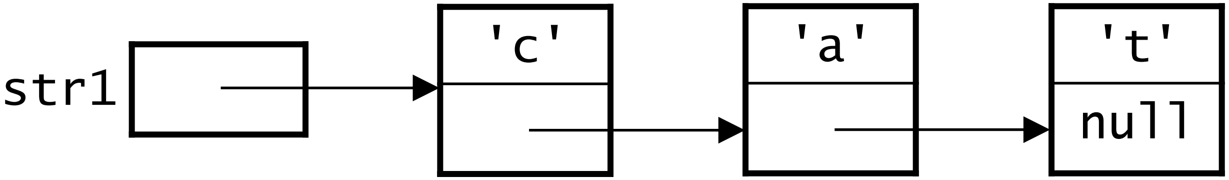
Single Linked List



head of the list

Example:

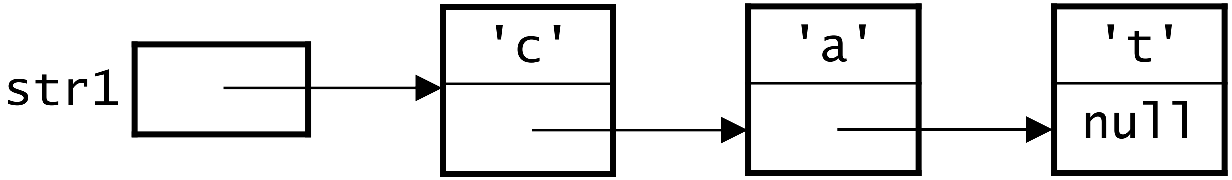
A String as a Linked List of Characters



- Each node in the linked list represents one character.

Example:

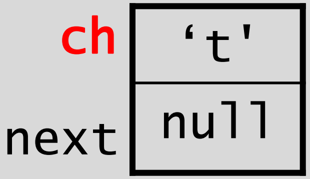
A String as a Linked List of Characters



- Each node in the linked list represents one character.

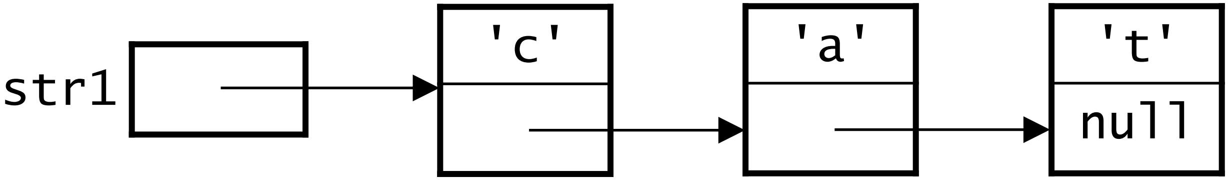
- Java class for this type of node:

```
public class StringNode {
    // data member for ch
    // data member for next


    // constructor to initialize the members



    ...
}
```



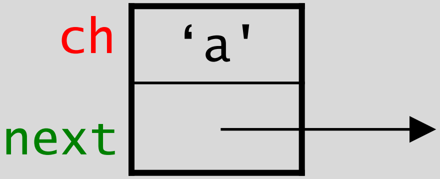
Example:

A String as a Linked List of Characters



- Each node in the linked list represents one character.

- Java class for this type of node:

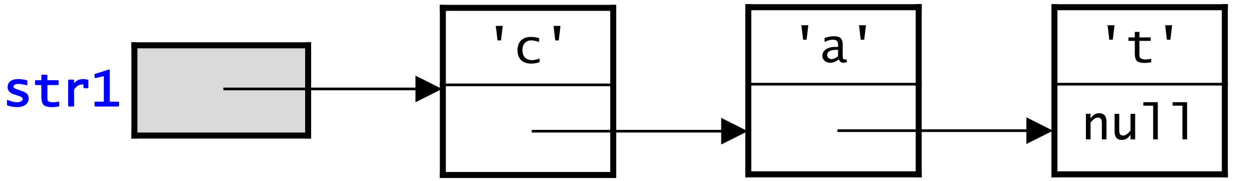


```
public class StringNode {
    private char ch;
    private StringNode next;

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

same type as the node itself!

Example:

A String as a Linked List of Characters

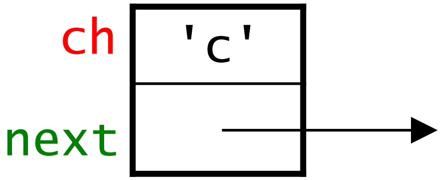


- Each node in the linked list represents one character.

- Java class for this type of node:

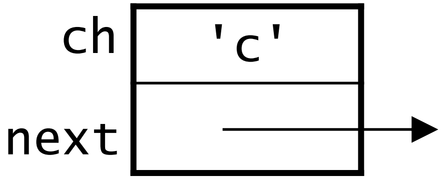
```
public class StringNode {
    private char ch;
    private StringNode next;

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

same type as the node itself!

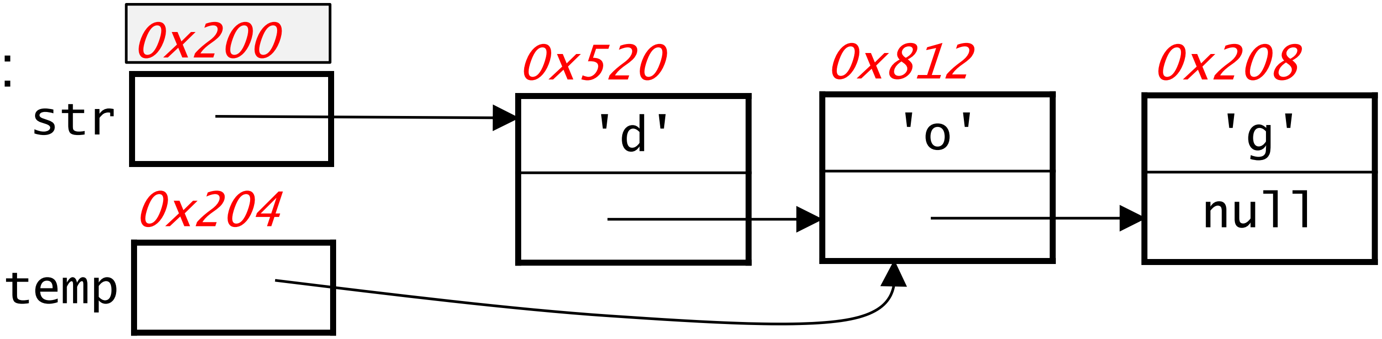


- The string as a whole is represented by a variable that holds a reference to the node for the first character (e.g., `str1` above).

Review of Variables

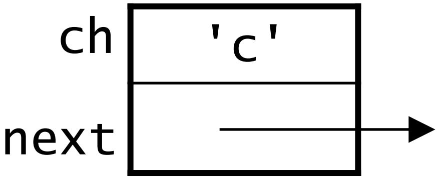


- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:



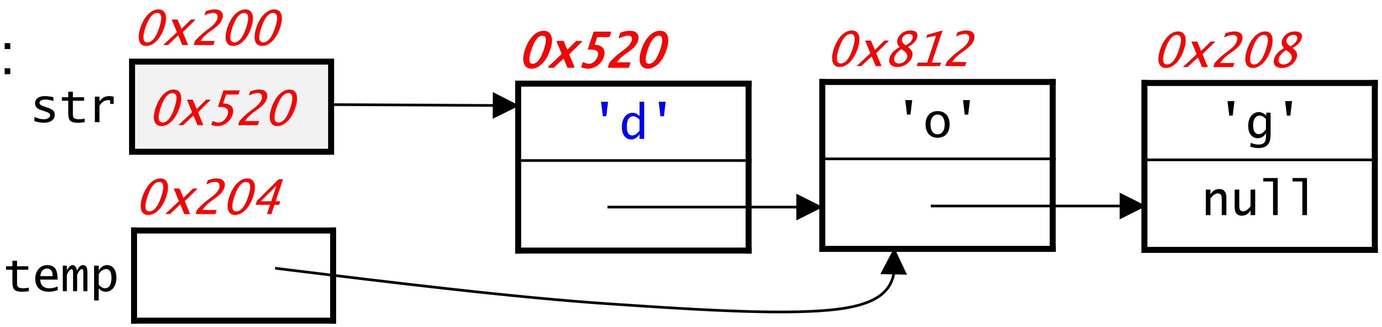
```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

Review of Variables



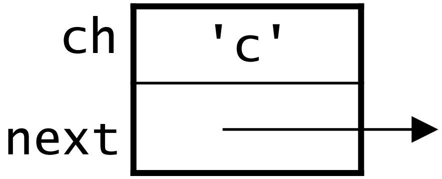
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



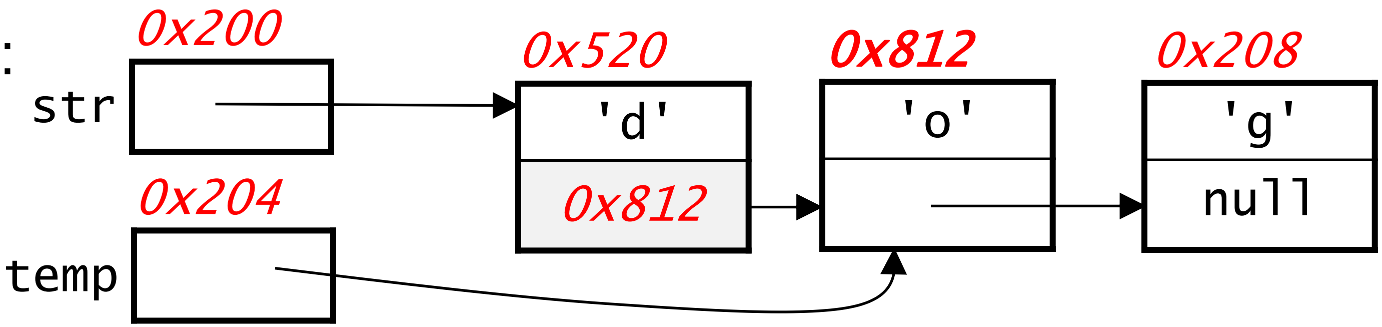
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



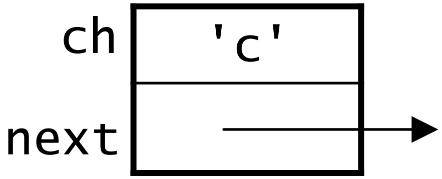
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



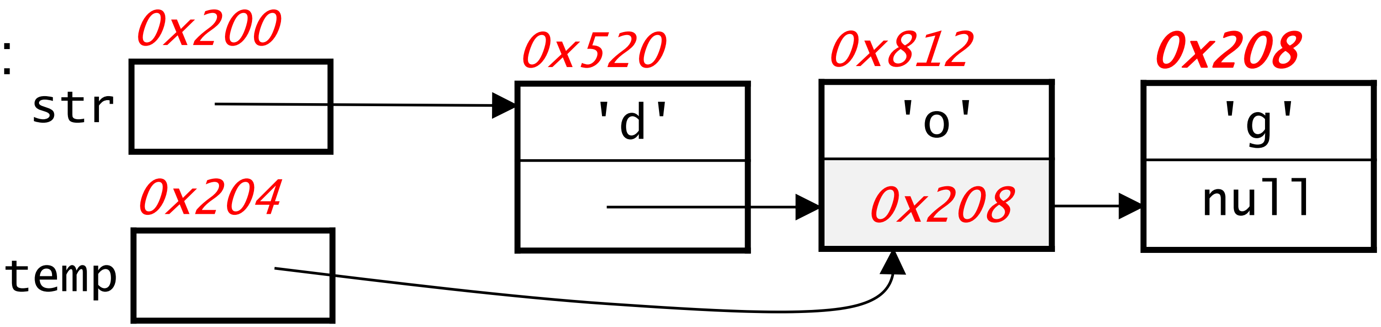
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



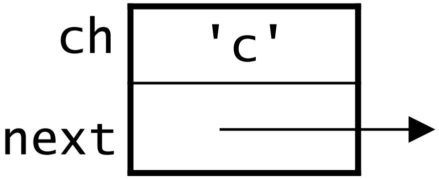
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



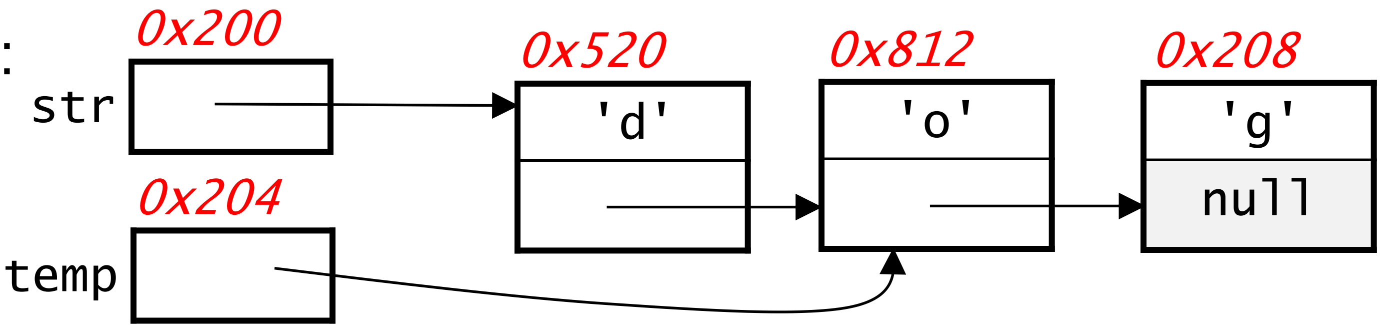
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



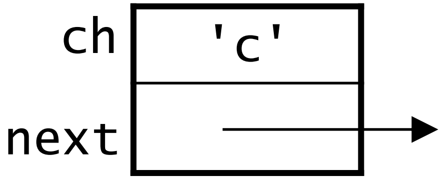
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



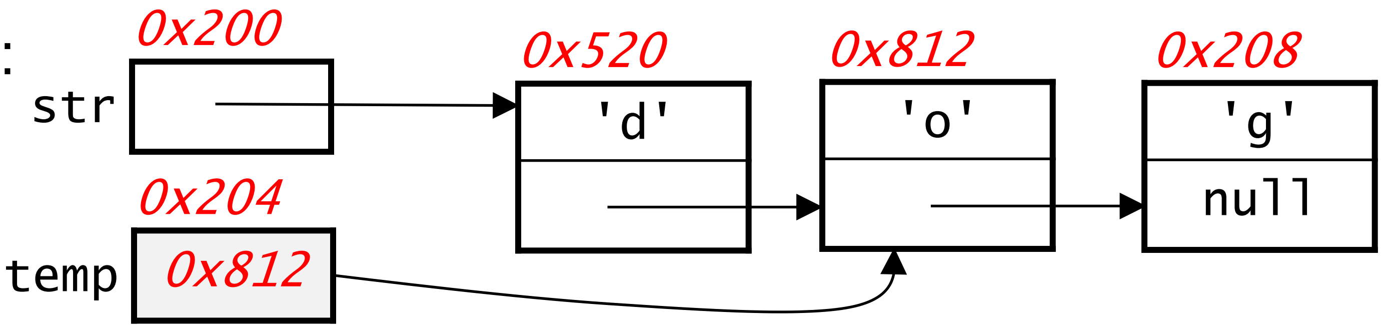
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



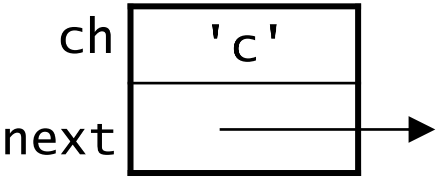
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



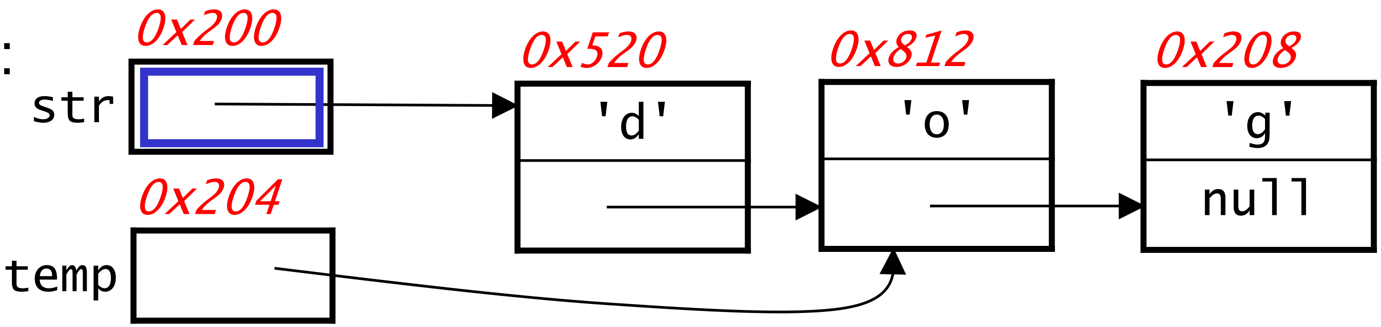
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

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)

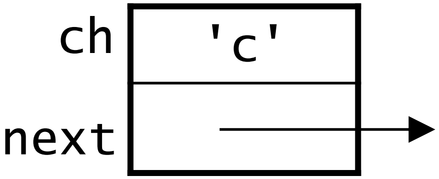
- Practice:



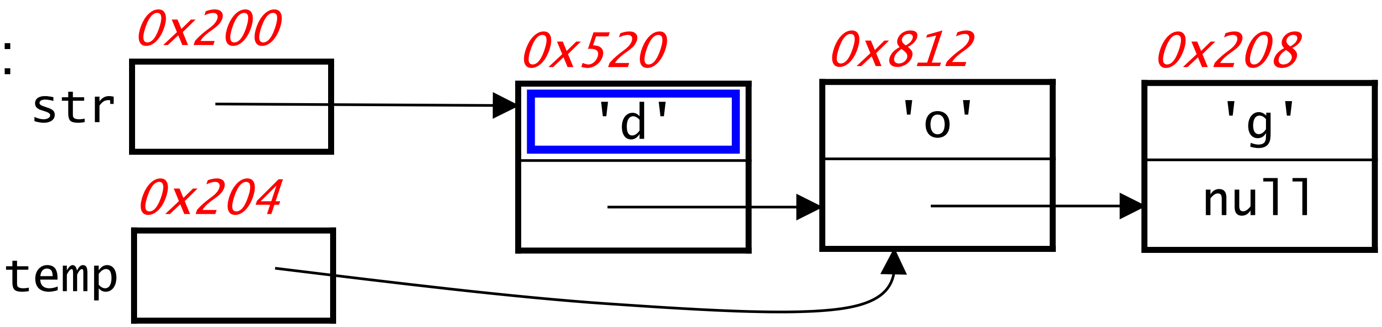
```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

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

Review of Variables



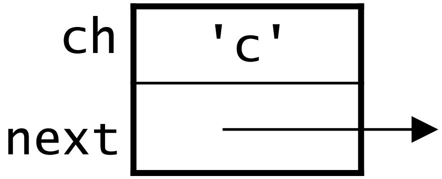
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:



```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

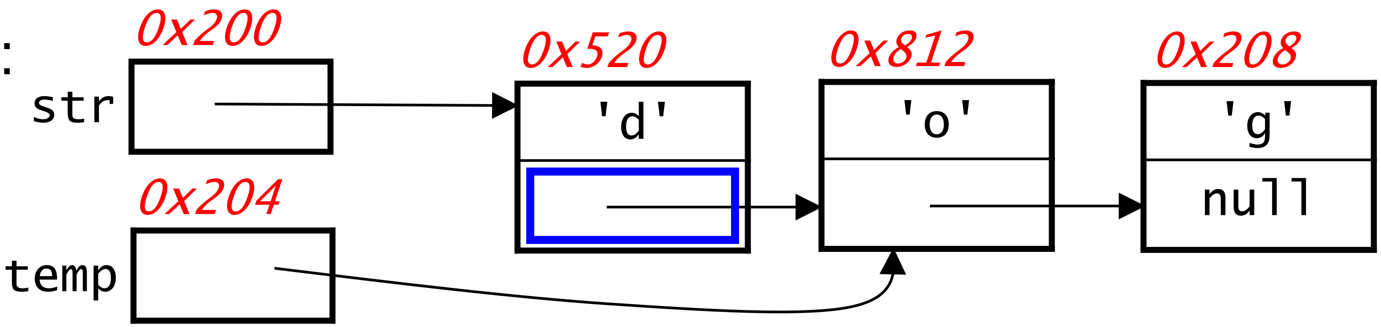
expression	*address*	*value*
`str`	0x200	0x520 (reference to the 'd' node)
`str.ch`	0x520	'd'
`str.next`		

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)

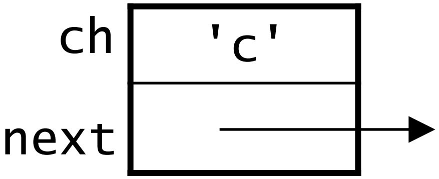
- Practice:



```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

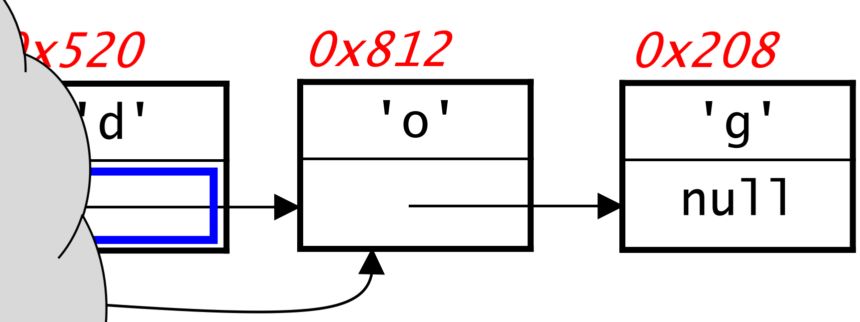
expression	*address*	*value*
`str`	0x200	0x520 (reference to the 'd' node)
`str.ch`	0x520	'd'
`str.next`	**0x522**	**0x812 (reference to the 'o' node)**

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)

- Practice:

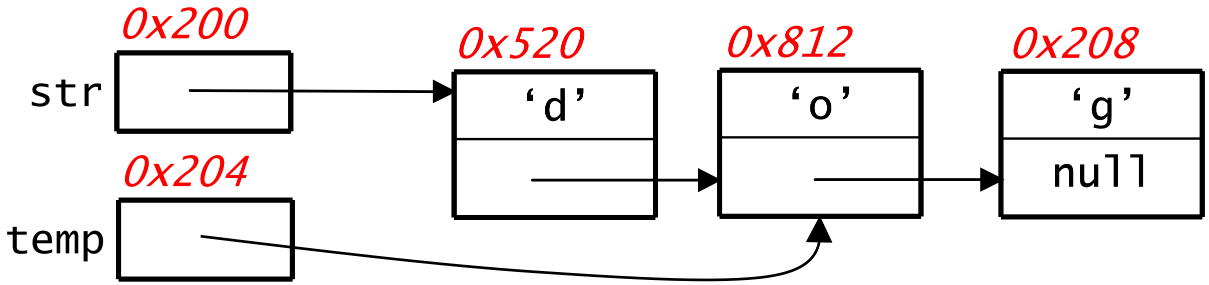


Note the use of the dot operator.

```
nts to the first node
oints to the second node
```

express	ess	*value*
str	0x200	0x520 (reference to the 'd' node)
str.ch	0x520	'd'
str.next	0x522	0x812 (reference to the 'o' node)

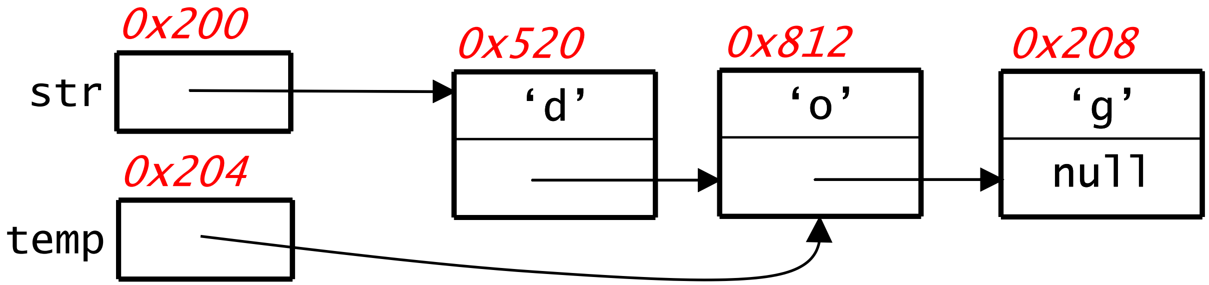
More Complicated Expressions



- Example: `temp.next.ch`

Understanding the dot operator

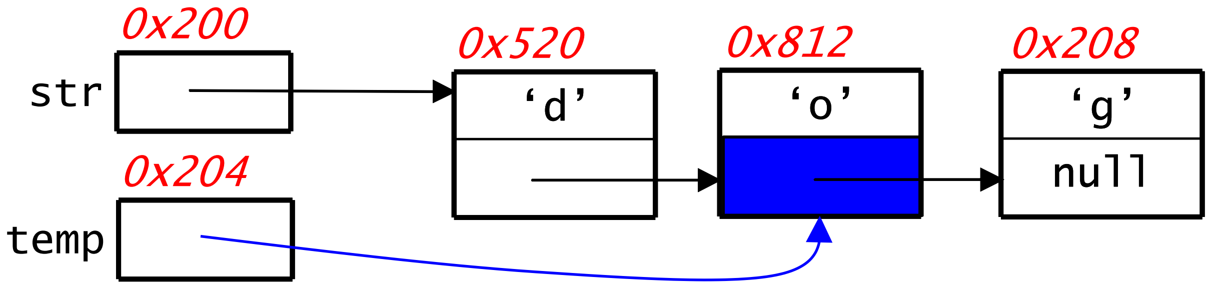
More Complicated Expressions



- Example: `temp.next.ch`

- `System.out.println( temp );` *0x812*

More Complicated Expressions



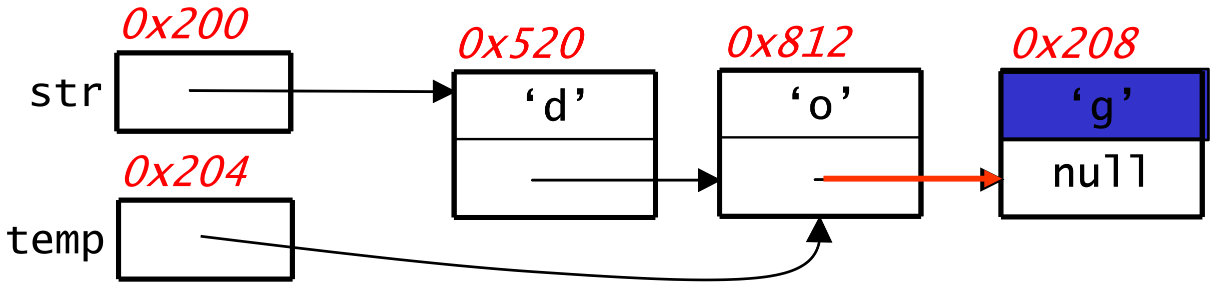
- Example: `temp.next.ch`

- `System.out.println( temp );` *0x812*

- `System.out.println( temp.next );` *0x208*

Says…
follow the reference.
Go to the address location stored in variable temp and access the next field of that object.

More Complicated Expressions



- Example: `temp.next.ch`

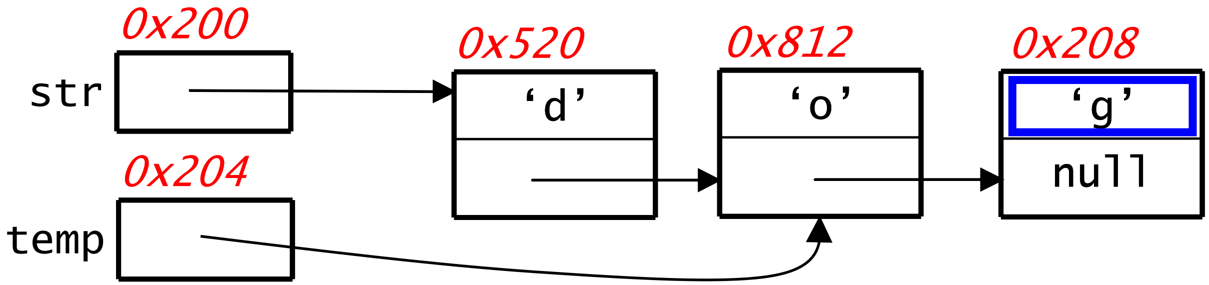
- `System.out.println( temp );` 0x812

- `System.out.println( temp.next );` 0x208

- `System.out.println( temp.next.ch );` 'g'

follow the reference

More Complicated Expressions



- Example: `temp.next.ch`

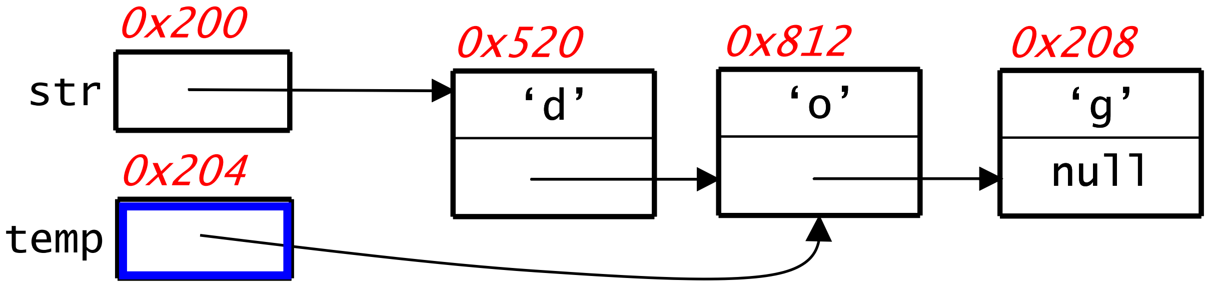
- Start with the beginning of the expression: `temp.next`
  It represents the `next` field of the node to which `temp` refers.
  - address = `0x814`
  - value = `0x208` (reference to the `'g'` node)

- Next, consider `temp.next.ch`
  It represents the `ch` field of the node to which `temp.next` refers.
  - address = **`0x208`**
  - value = **`'g'`**

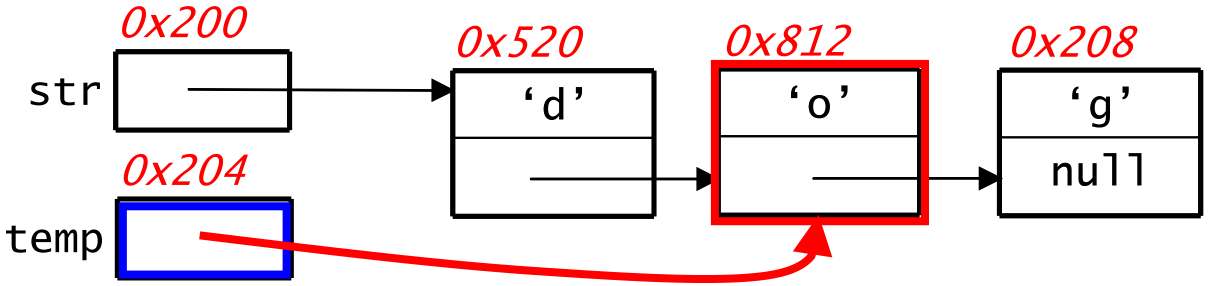
Dereferencing a Reference:

another look

- Each dot causes us to *dereference* the reference represented by the expression preceding the dot.
- Consider again `temp.next.ch`
- Start with temp: `temp.next.ch`



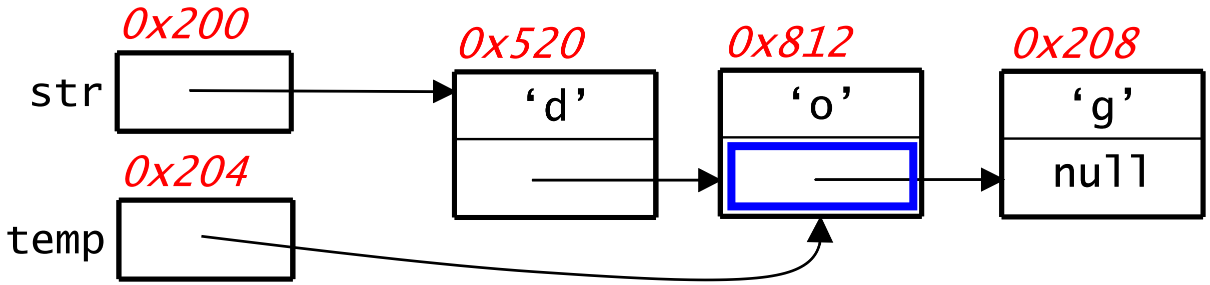
- Dereference: `temp.next.ch`



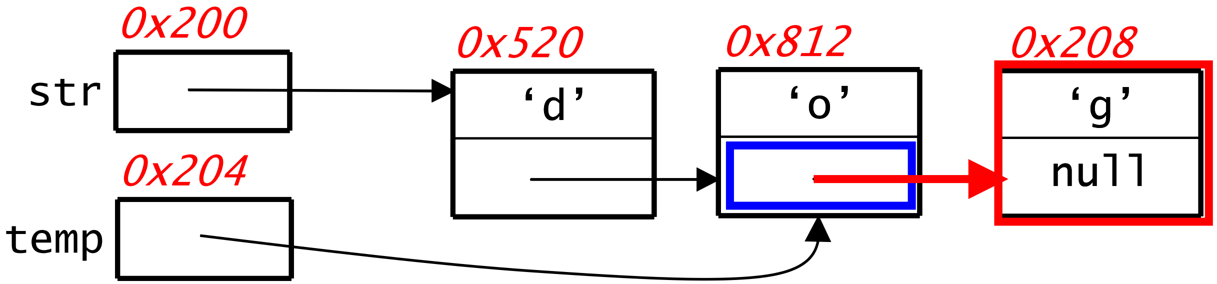
Dereferencing a Reference:

another look

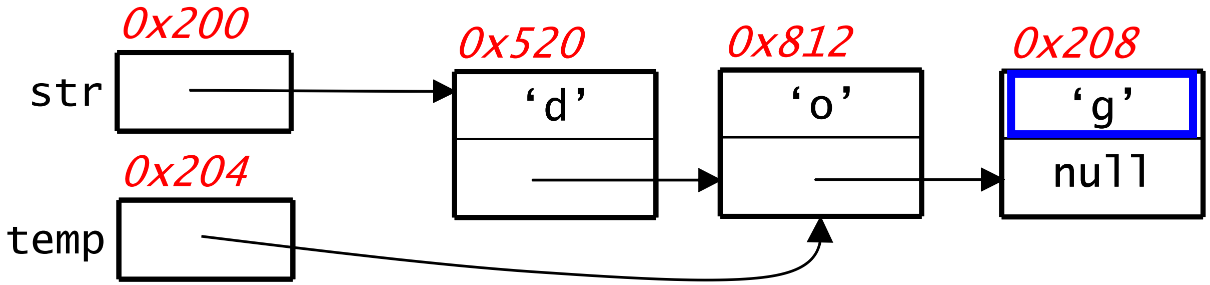
- Get the `next` field: `temp.next.ch`



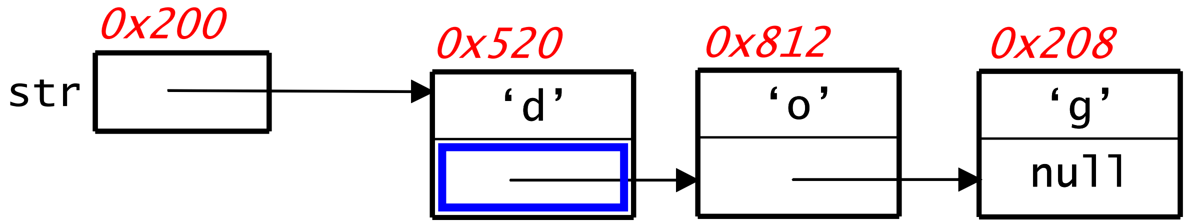
- Dereference: `temp.next.ch`



- Get the `ch` field: `temp.next.ch`



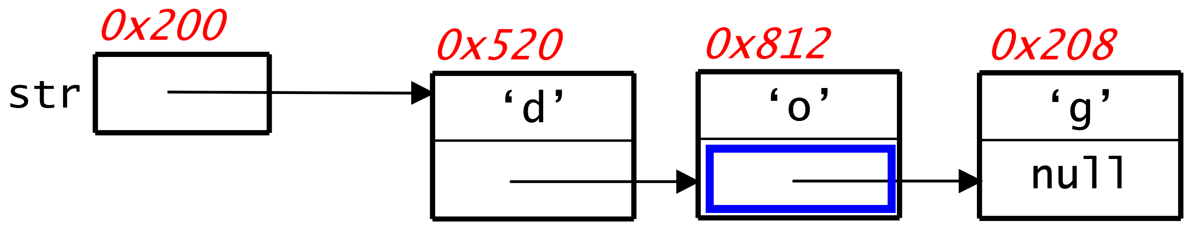
What are the address and value of `str.next.next`?



- `str.next` is the `next` field in the node to which `str` refers
  - it holds a *reference* to the `'o'` node

	address	value
A.	0x522	0x812
B.	0x812	'o'
C.	0x814	0x208
D.	0x208	'g'
E.	0x210	null

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



- `str.next` is the `next` field in the node to which `str` refers
  - it holds a reference to the `'o'` node
- thus, `str.next.next` is the `next` field in the `'o'` node
  - it holds a reference to the `'g'` node

	address	value
A.	`0x522`	`0x812`
B.	`0x812`	`'o'`
C.	**`0x814`**	**`0x208`**
D.	`0x208`	`'g'`
E.	`0x210`	`null`

Review of Assignment Statements

- An assignment of the form

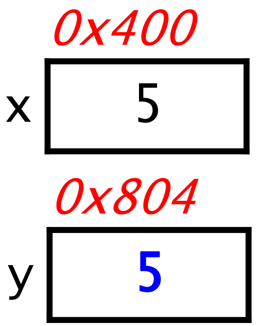
```
var1 = var2;
```

  - takes the *value* of `var2`
  - copies it into the box (*memory cell*) at the *address* of `var1`

In other words, it takes the value in `var2` *and copies it into* `var1`

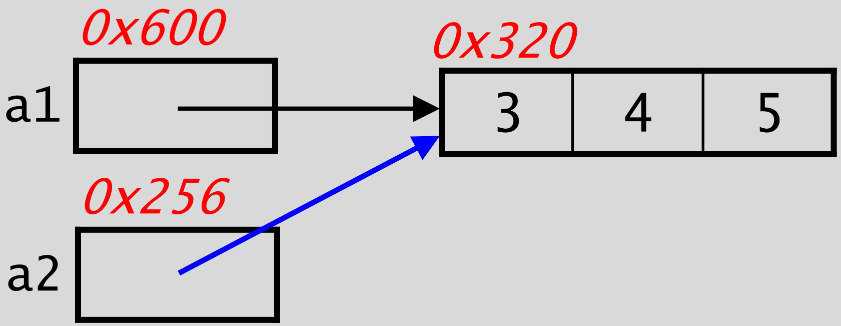
- Example involving integers:

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



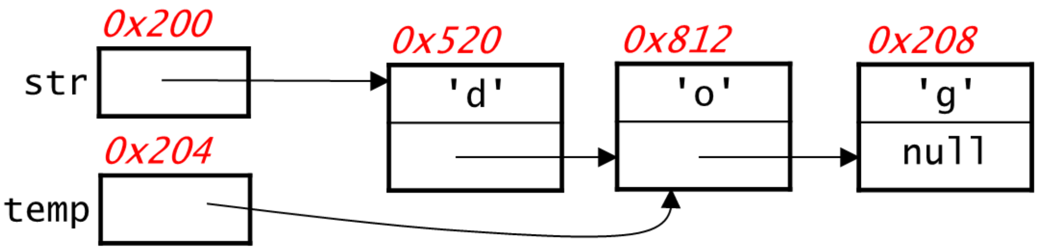
- Example involving references:

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



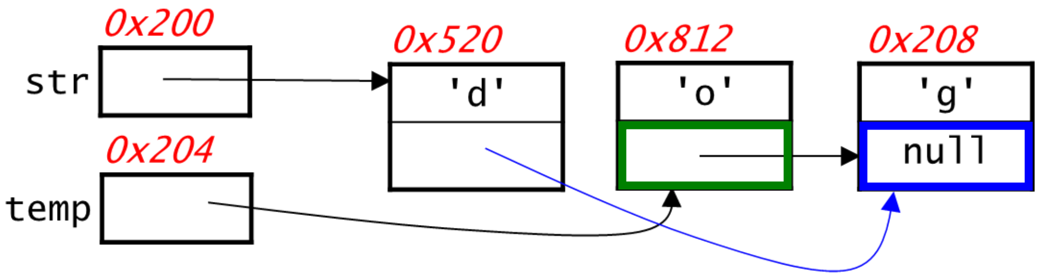
What About These Assignments?

- *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.*



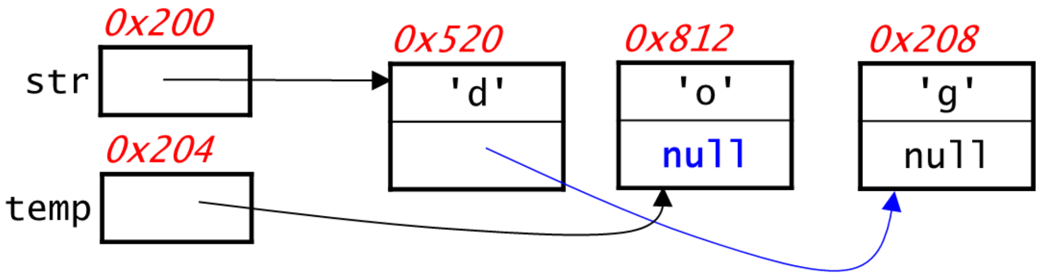
```
1)  str.next = temp.next;
```

0x208 (a reference to the 'g' node)



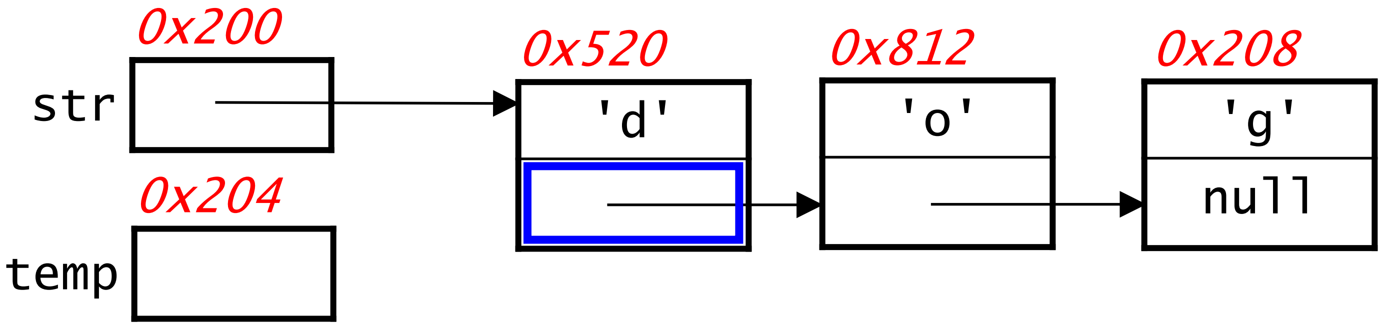
```
2)  temp.next = temp.next.next;
```

null



Writing an Appropriate Assignment

- What assignment is needed to make variable `temp` reference the node containing the character 'o‘ ?



  - find the reference to the `'o'` node:
  - determine the expression to accesses the reference:
    `str.next`
  - write the assignment:
    `temp = str.next;`

Example:
A String as a Linked List of Characters

- 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 will write `length(str1)` instead of `str1.length()`
  - outside the class, call the methods using the class name:

    ```
    StringNode.length(str1)
    ```

- Using static methods allows the methods to handle empty strings.
  - if `str1 == null`:
    - `length(str1)` will work
    - `str1.length()` will throw a `NullPointerException`

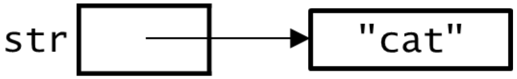
A Linked List Is a Recursive Data Structure!

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

- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

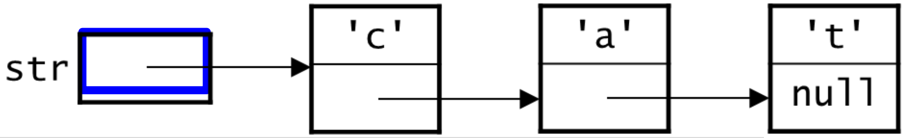
Recursively Finding the Length of a String

- For a 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;
    }
}
```

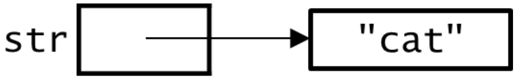
- **For a linked-list string:**



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

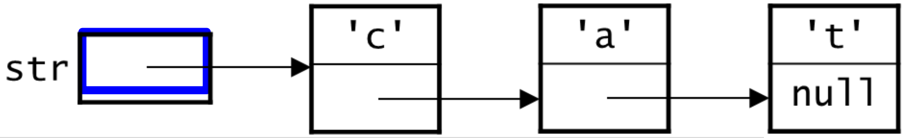
Recursively Finding the Length of a String

- 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;
    }
}
```

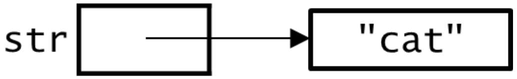
- For a linked-list string:



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

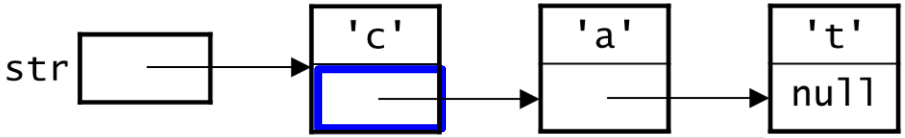
Recursively Finding the Length of a String

- 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;
    }
}
```

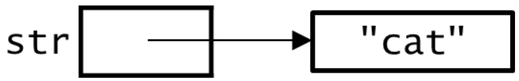
- For a linked-list string:



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

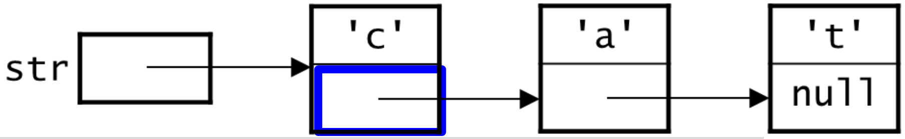
Recursively Finding the Length of a String

- 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;
    }
}
```

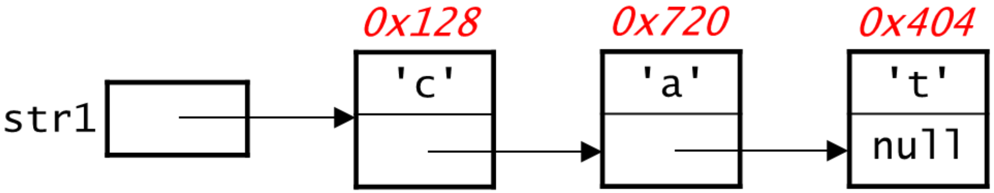
- Version without a variable for the result of the recursive call:

```
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);
    }
}
```



0x128 0x720 0x404

str1

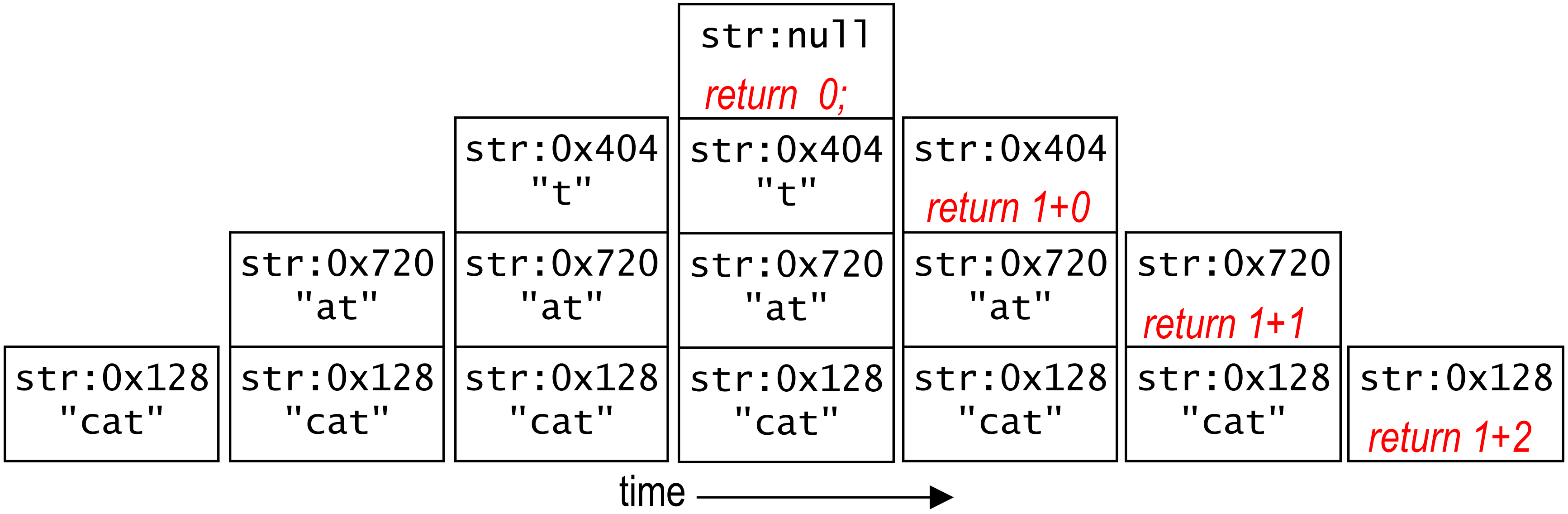
'c'

'a'

't'

null

- Example: `StringNode.length(str1)`



str:0x128
"cat"

str:0x720
"at"

str:0x128
"cat"

str:0x404
"t"

str:0x720
"at"

str:0x128
"cat"

str:null
return 0;

str:0x404
"t"

str:0x720
"at"

str:0x128
"cat"

str:0x404
return 1+0

str:0x720
"at"

str:0x128
"cat"

str:0x720
return 1+1

str:0x128
"cat"

str:0x128
return 1+2

time