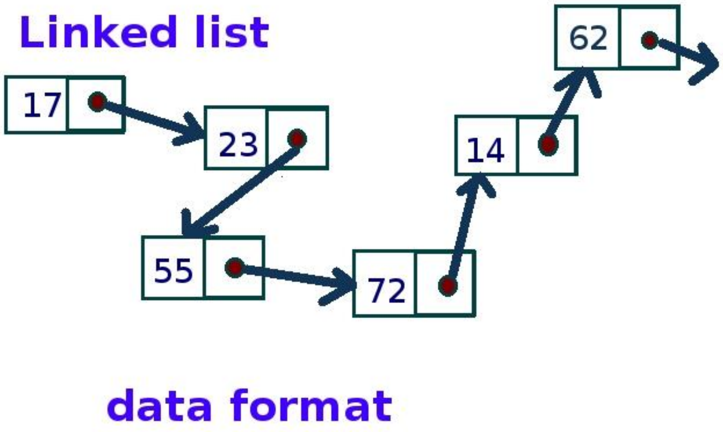


Linked Lists:
An Overview

Computer Science CS112
Boston University

Christine Papadakis-Kanaris

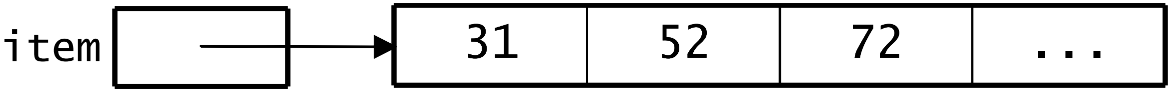
Representing a Sequence of Data

- 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



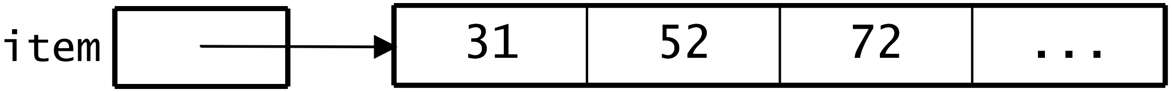
Representing a Sequence of Data

- 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



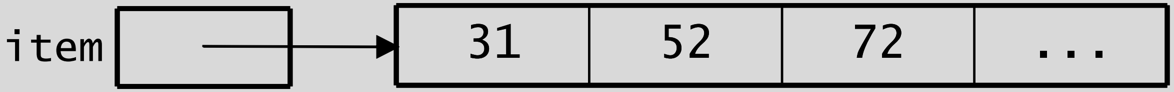
Representing a Sequence of Data

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

- Most common representation = an array

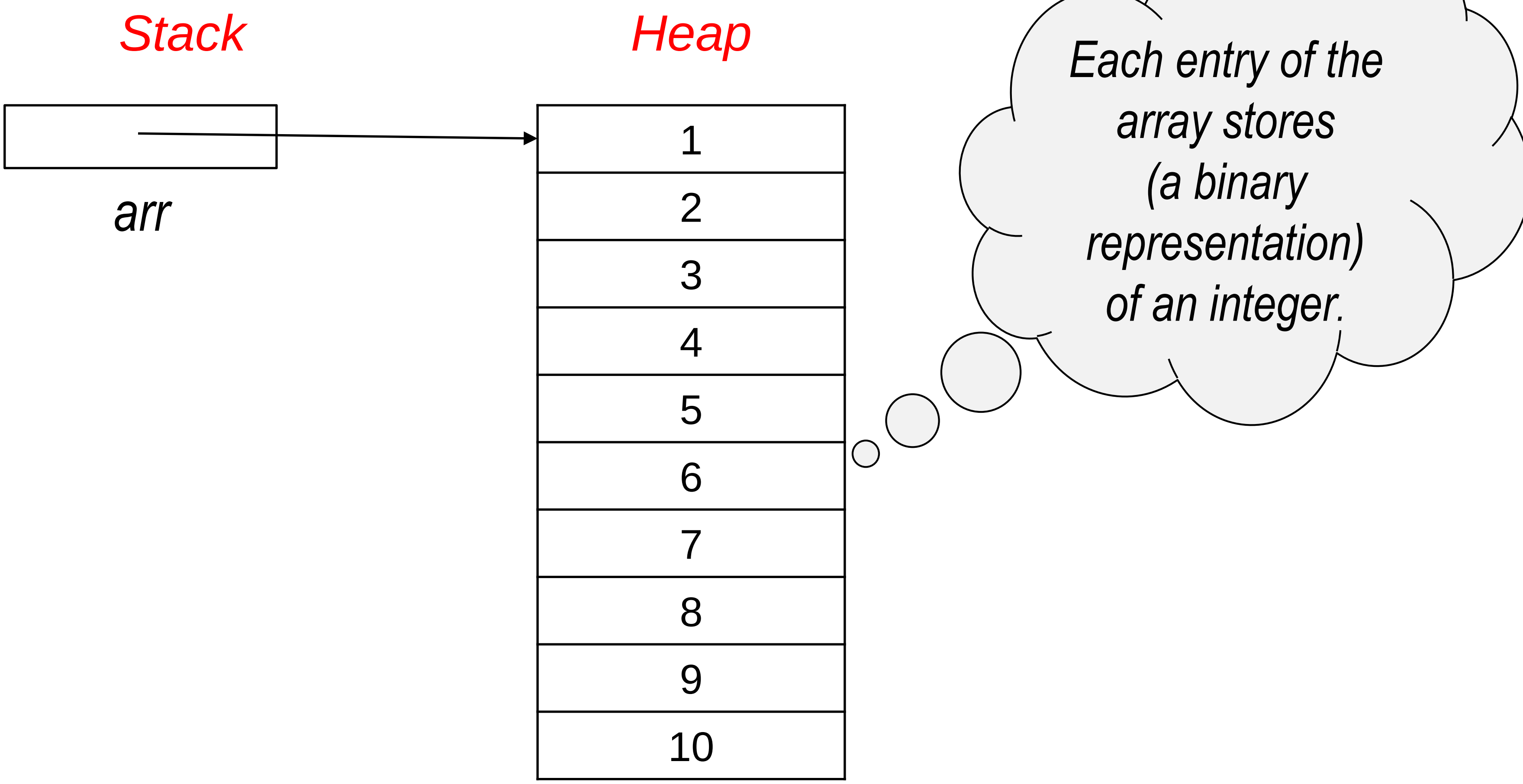
- Advantages of using an array:
  - easy and efficient access to *any* item in the sequence
    - `item[i]` gives you the item at position i in O(1) time
    - known as *random access*
  - very compact (but can waste space if positions are empty)

- 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



Array:
a fixed Data structure

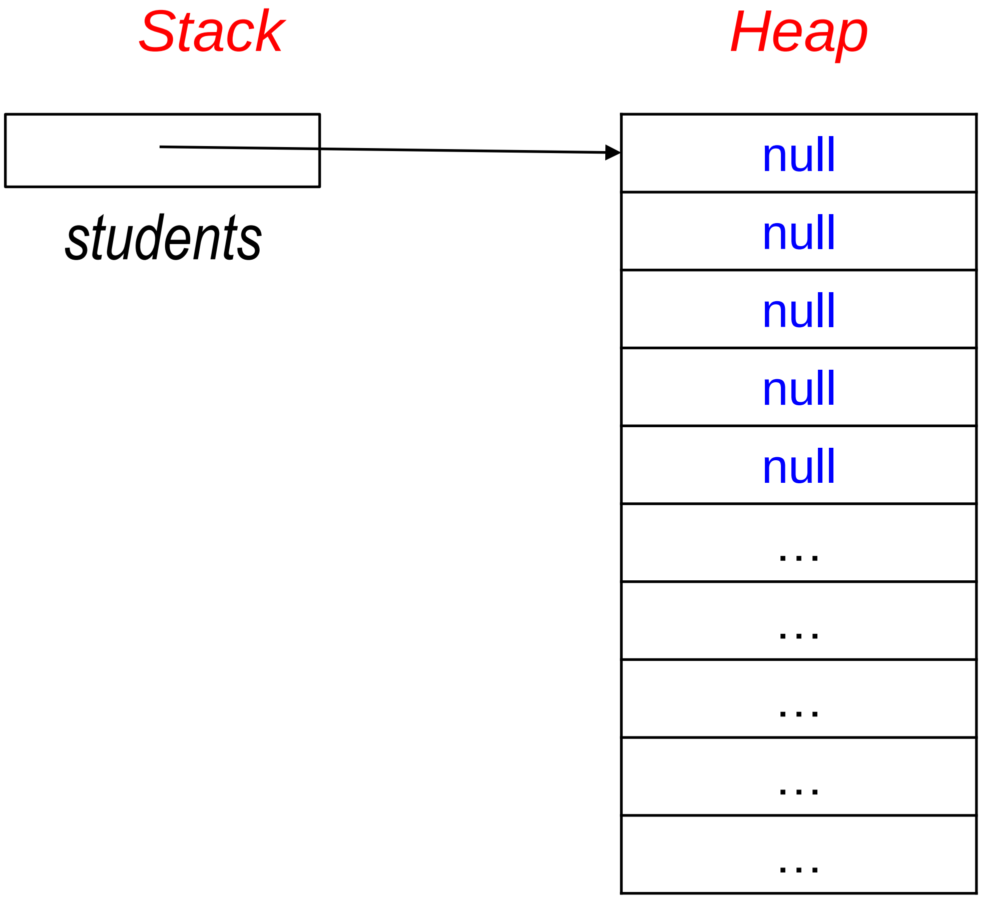
```
int [] arr = {1,2,3,4,5,6,7,8,9,10};
```



Each entry of the array stores (a binary representation) of an integer.

Array:
a fixed Data structure

```
Student [] students = new Student[N];
```



Stack

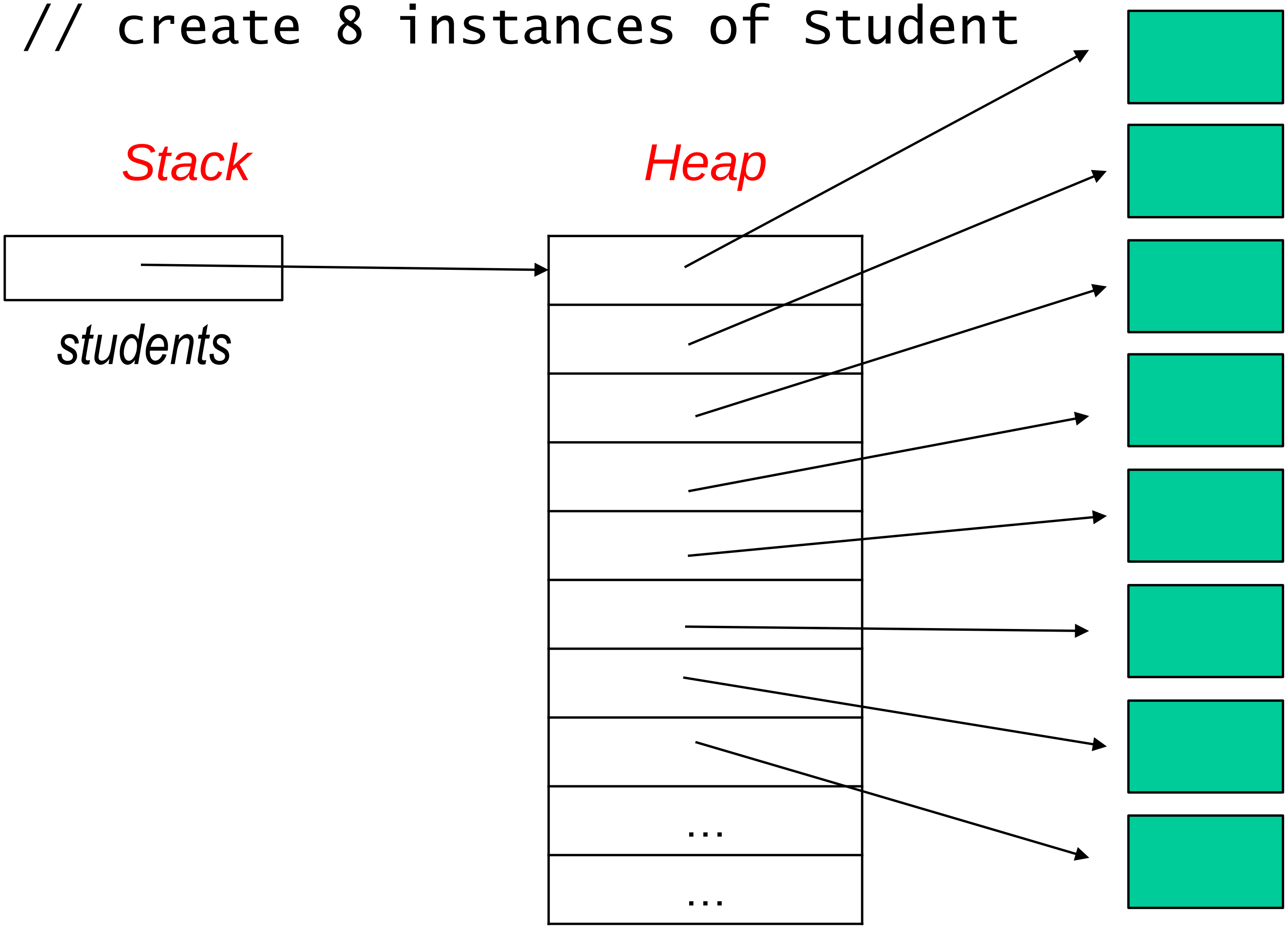
Heap

students

null
null
null
null
null
…
…
…
…
…

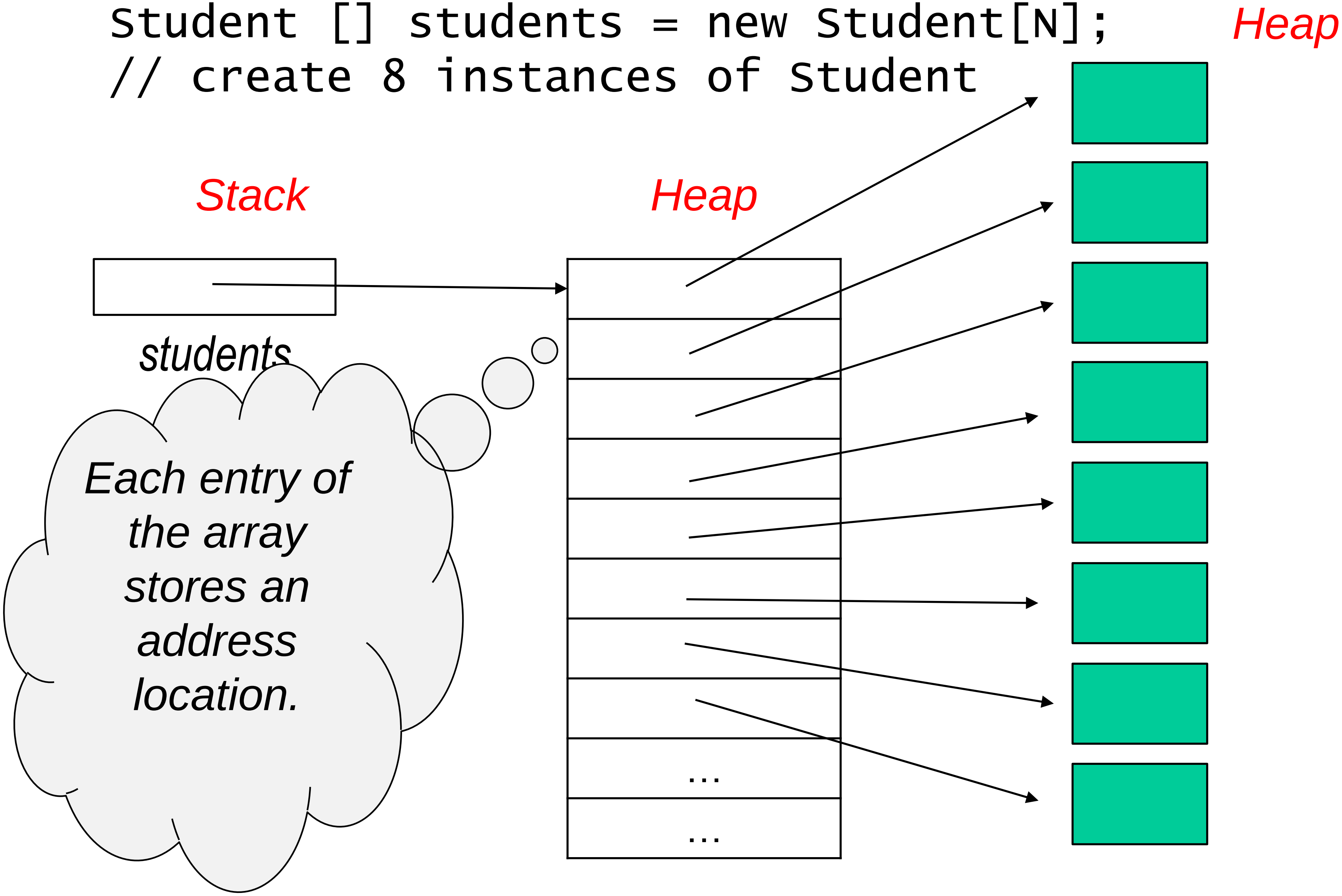
Array:
a fixed Data structure

```
Student [] students = new Student[N];
// create 8 instances of Student
```



Array:
a fixed Data structure

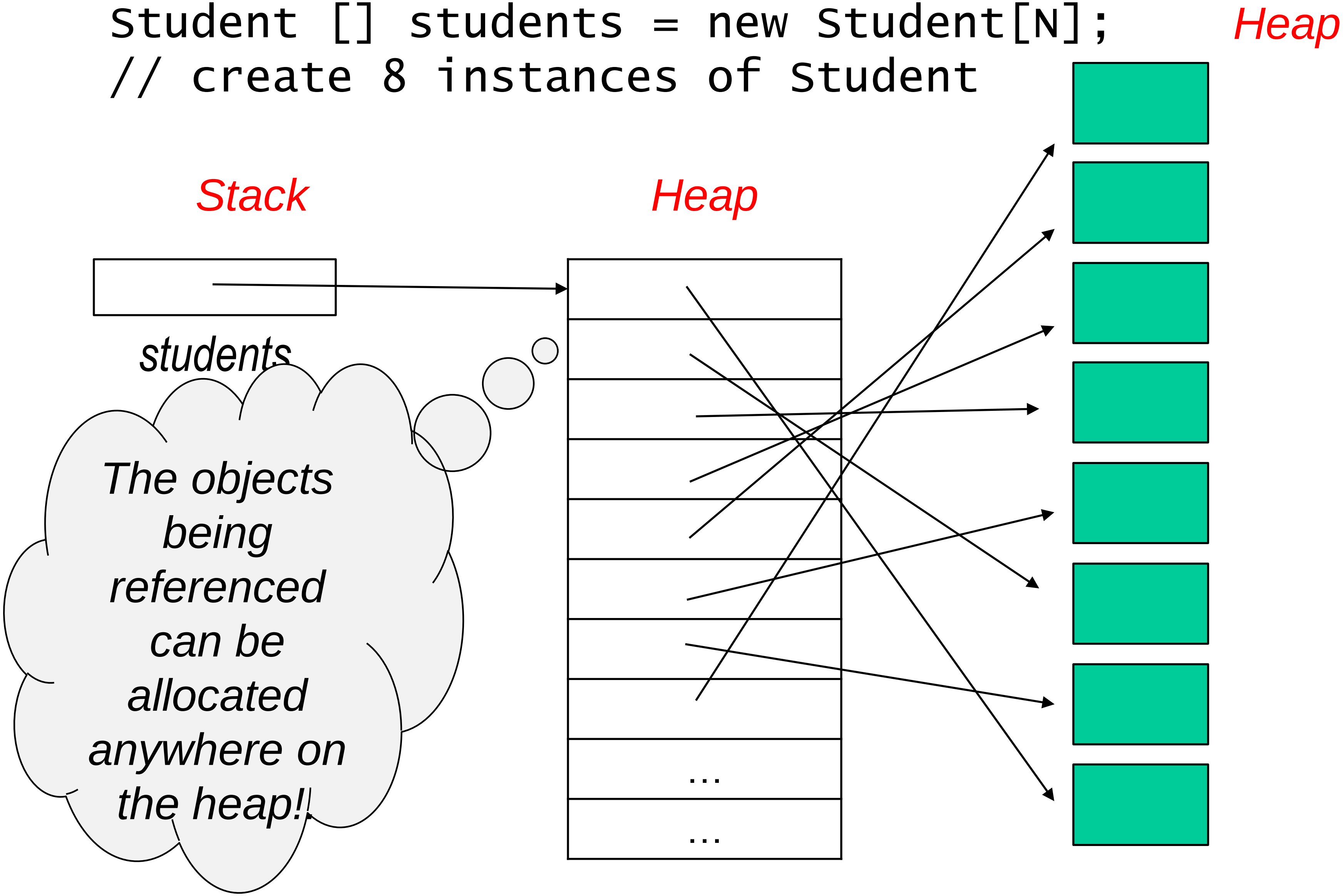
```
Student [] students = new Student[N];
// create 8 instances of Student
```



Each entry of the array stores an address location.

Array:
a fixed Data structure

```
Student [] students = new Student[N];
// create 8 instances of Student
```



Heap

Stack

Heap

students

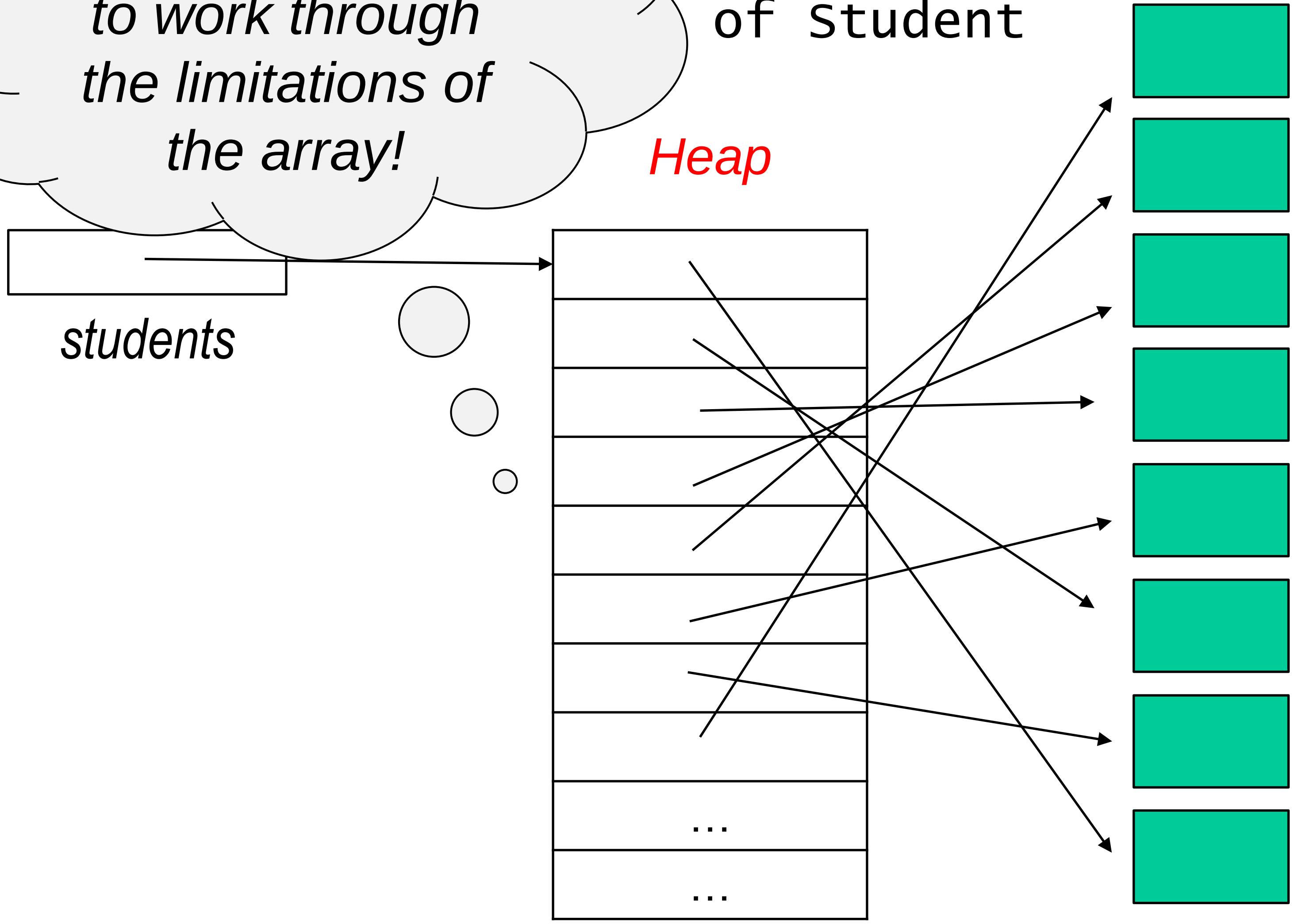
The objects being referenced can be allocated anywhere on the heap!

Array:
a fixed Data structure

```
= new Student[N];
of Student
```

Heap

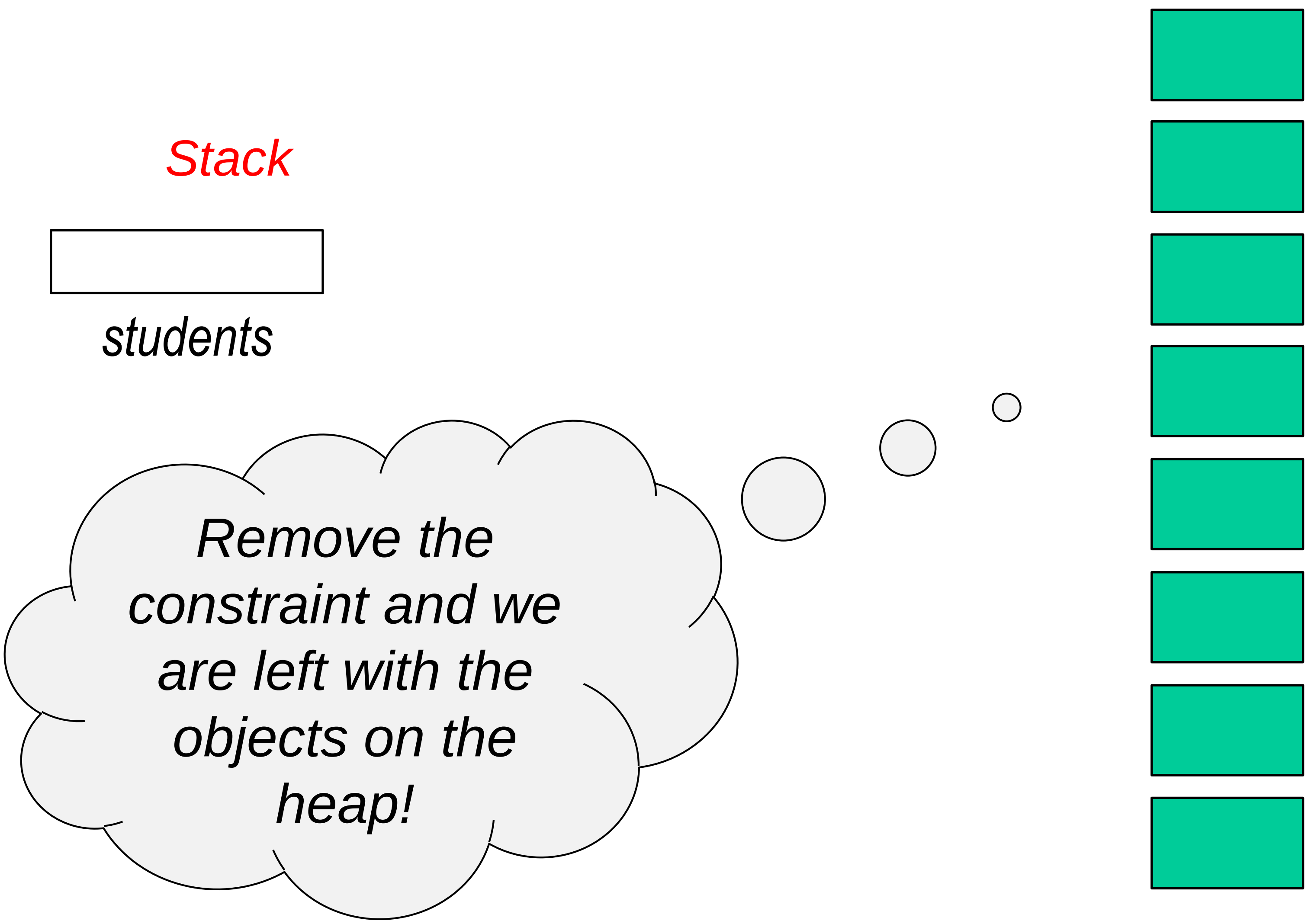
But we still have to work through the limitations of the array!



Linked List:
a dynamic Data structure

```
// create 8 instances of Student
```

Heap



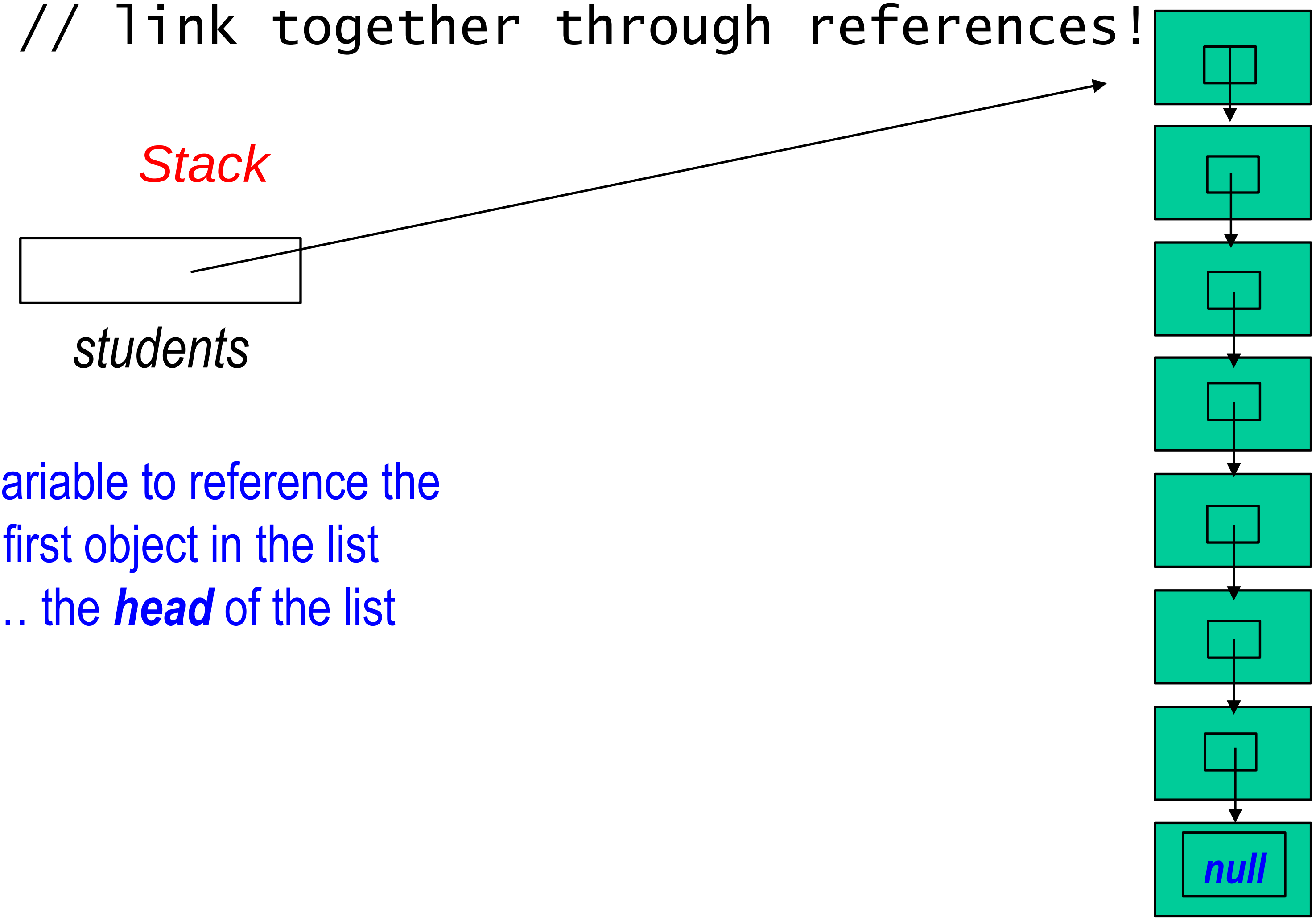
Stack

students

Remove the constraint and we are left with the objects on the heap!

Linked List:
a dynamic Data structure

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



Heap

Stack

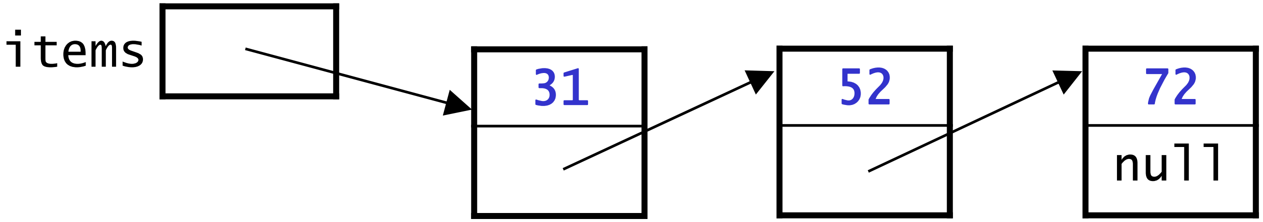
students

null

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

A Linked List of …

- Example:



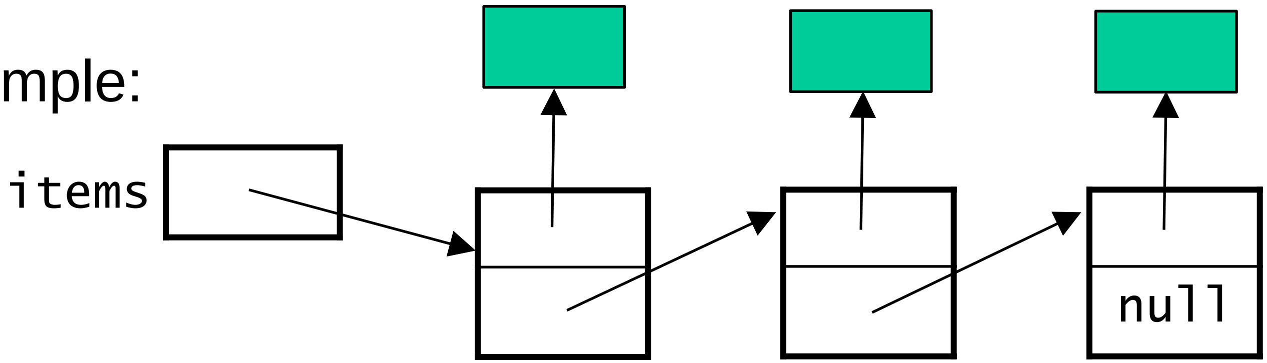
- A linked list stores a sequence of items in separate *nodes*.

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

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

A Linked List of …

- Example:

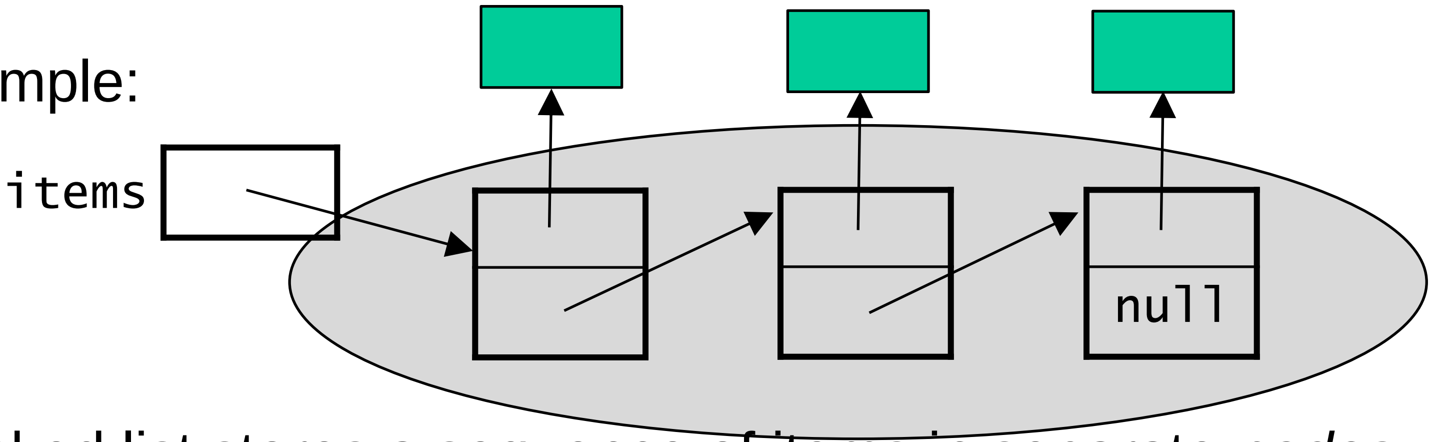


- A linked list stores a sequence of items in separate *nodes*.
- Each node contains:
  - a single *data* item

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

A Linked List of …

- Example:

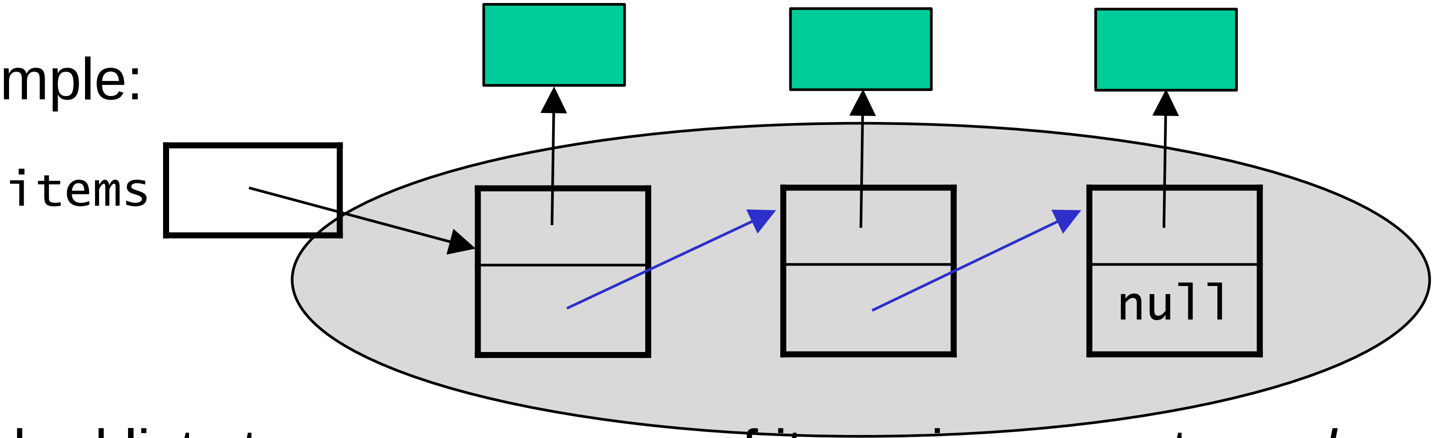


- A linked list stores a sequence of items in separate *nodes*.
- Each node contains:
  - a single *data* item

The nodes of the list form the sequence…

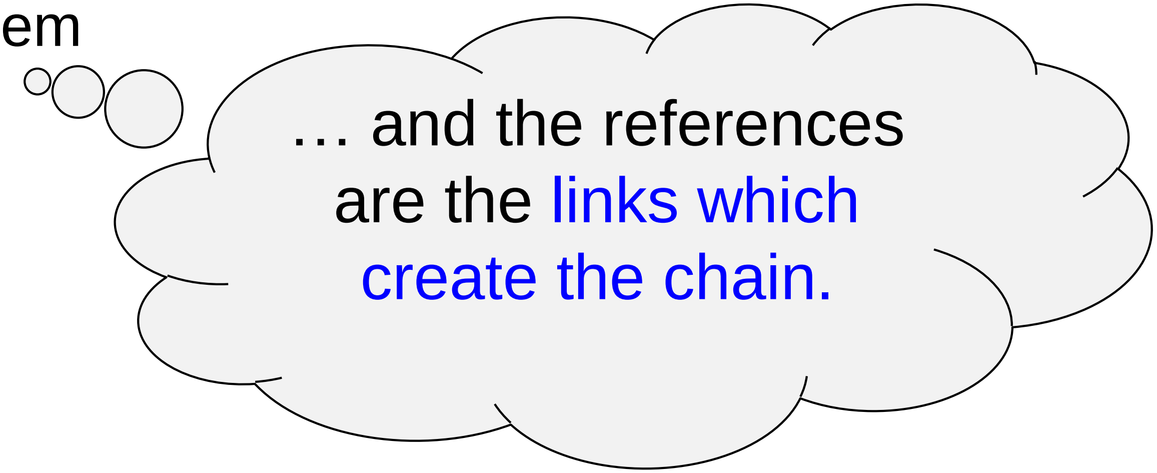
A Linked List of …

- Example:



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

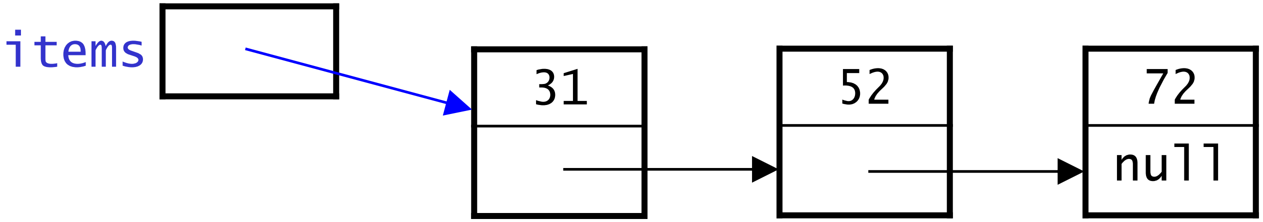
- Each node contains:
  - a single *data* item



… and the references are the links which create the chain.

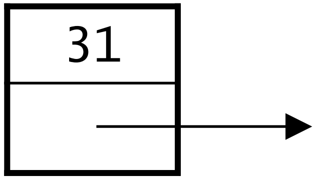
A Linked List

- 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

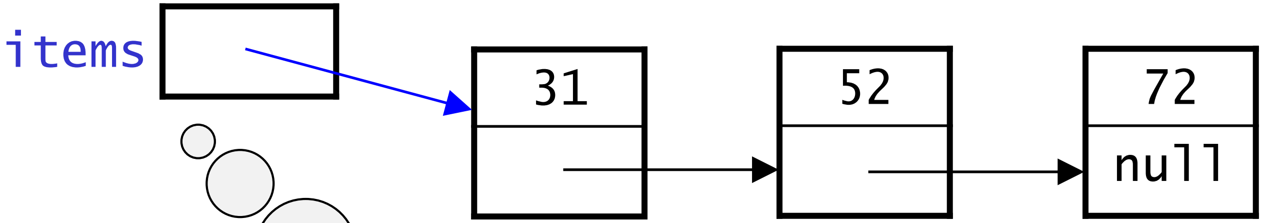
example node:



- 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

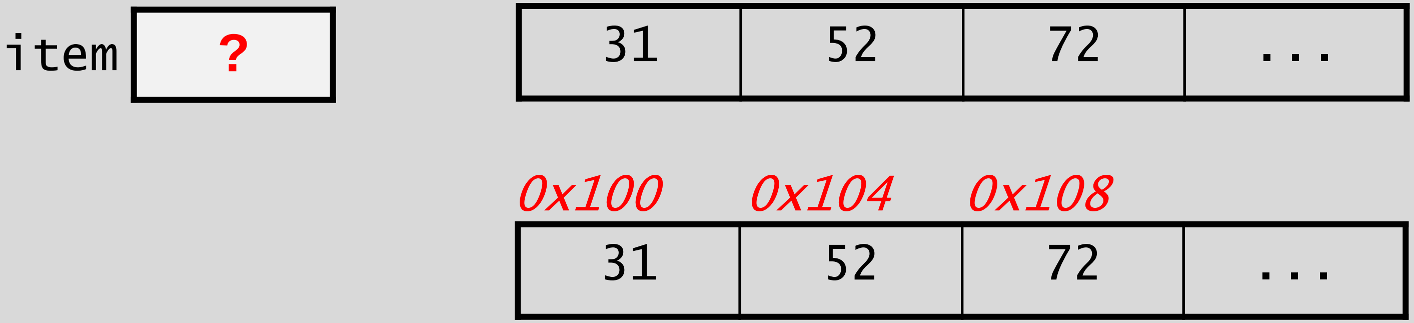
  example no

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:



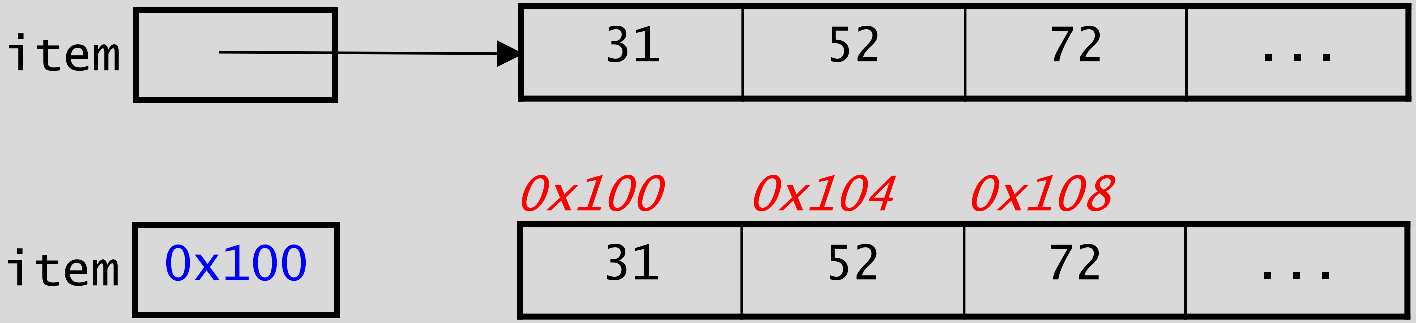
item ?

31	52	72	...

0x100	*0x104*	*0x108*	
31	52	72	...

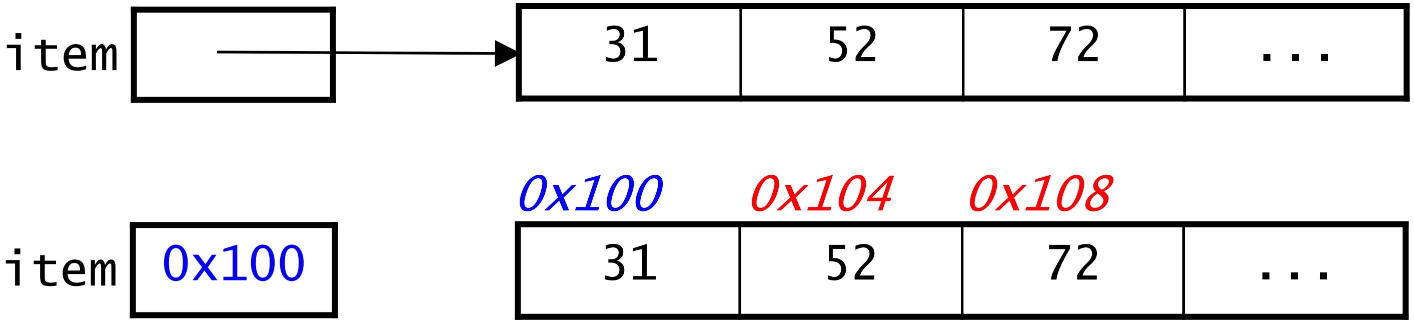
Arrays vs. Linked Lists in Memory

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

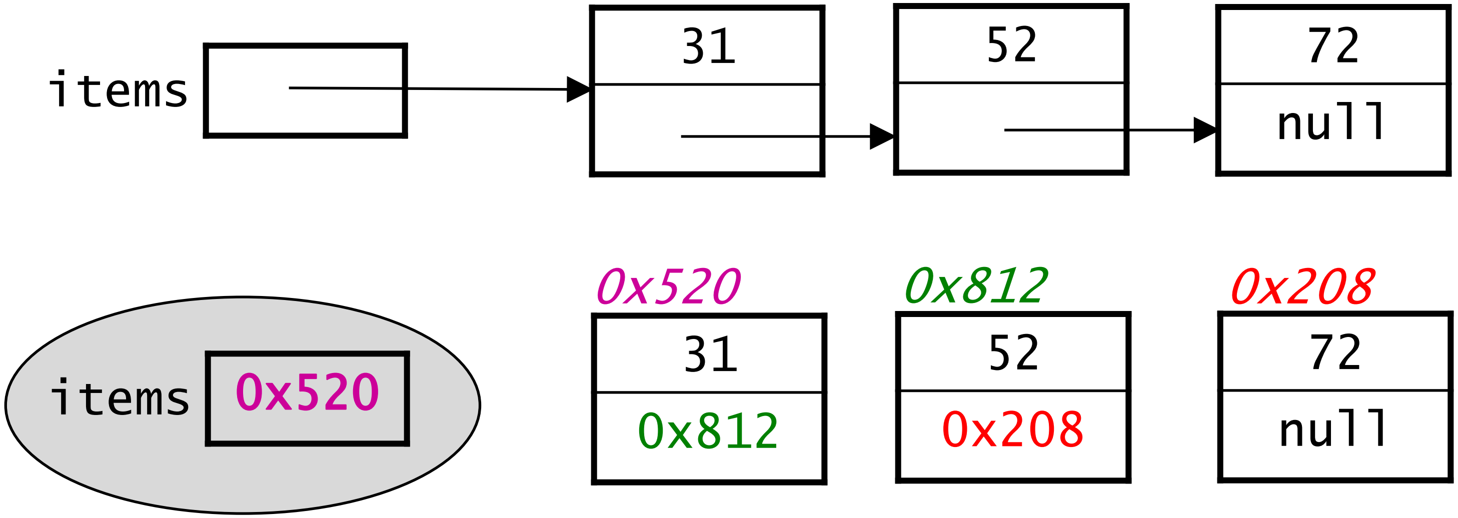


Arrays vs. Linked Lists in Memory

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

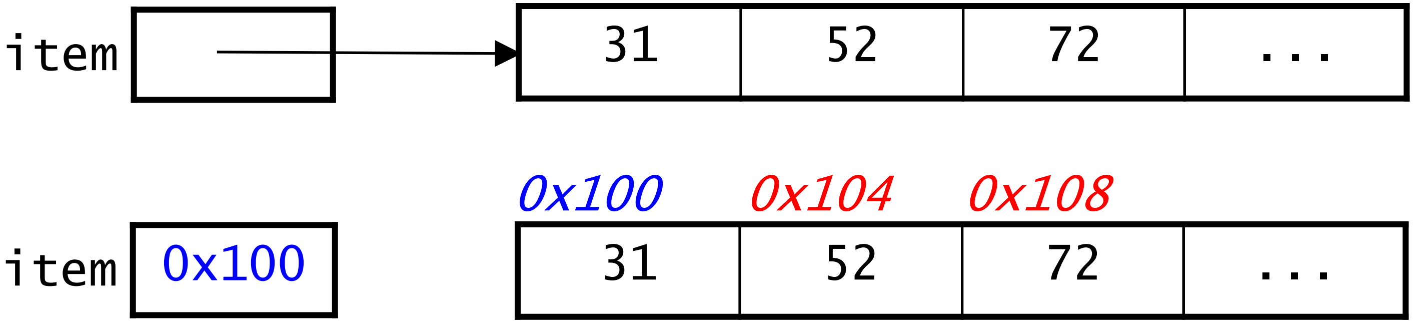

item
31 52 72 ...
item 0x100
0x100 0x104 0x108
31 52 72 ...

- 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.


items
31
52
72 null
items 0x520
0x520 0x812 0x208
31 0x812
52 0x208
72 null

Arrays vs. Linked Lists in Memory

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



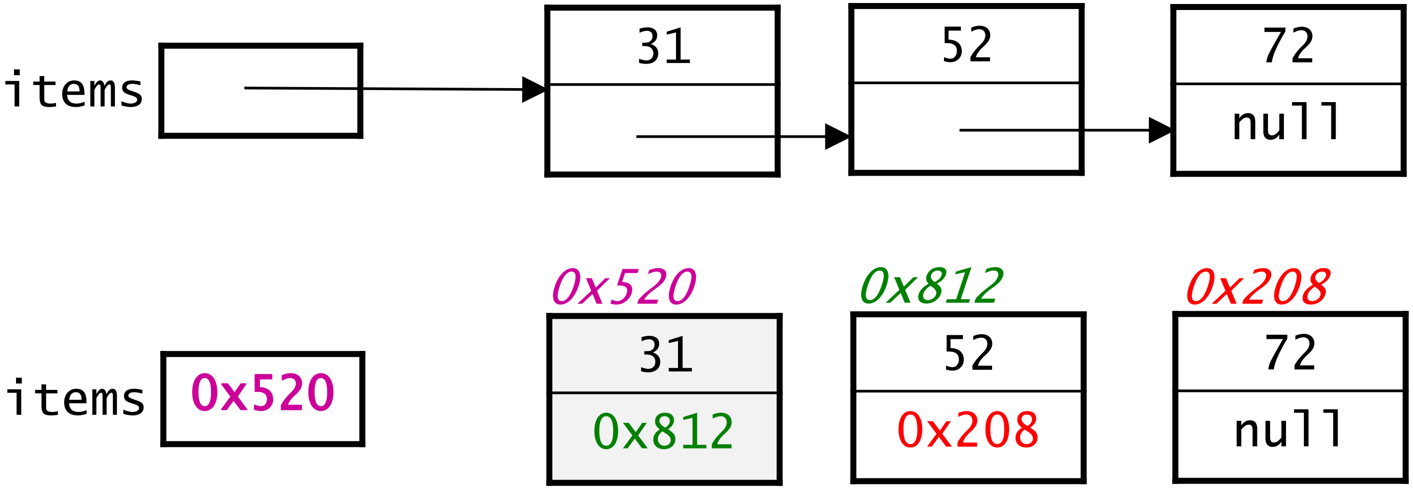
item

31	52	72	...

item 0x100

0x100	0x104	0x108	
31	52	72	...

- 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.



items

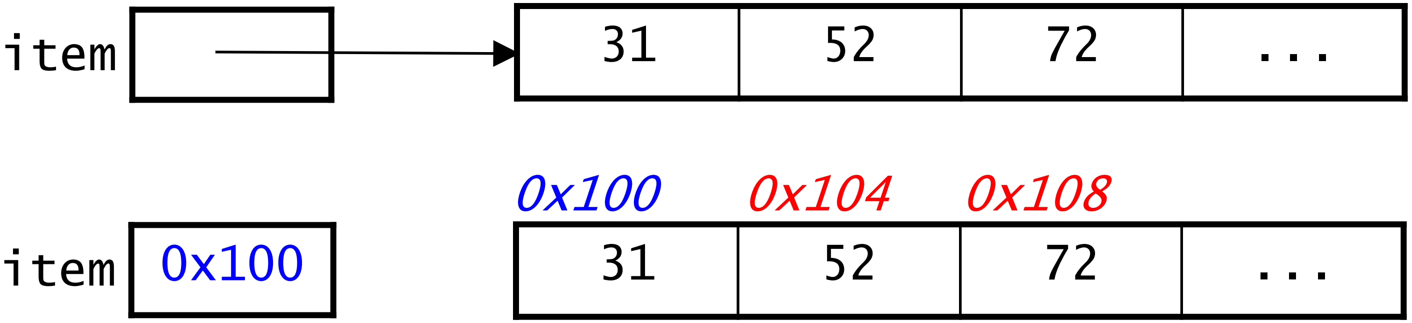
31	52	72
		null

items 0x520

0x520	0x812	0x208
31	52	72
0x812	0x208	null

Arrays vs. Linked Lists in Memory

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



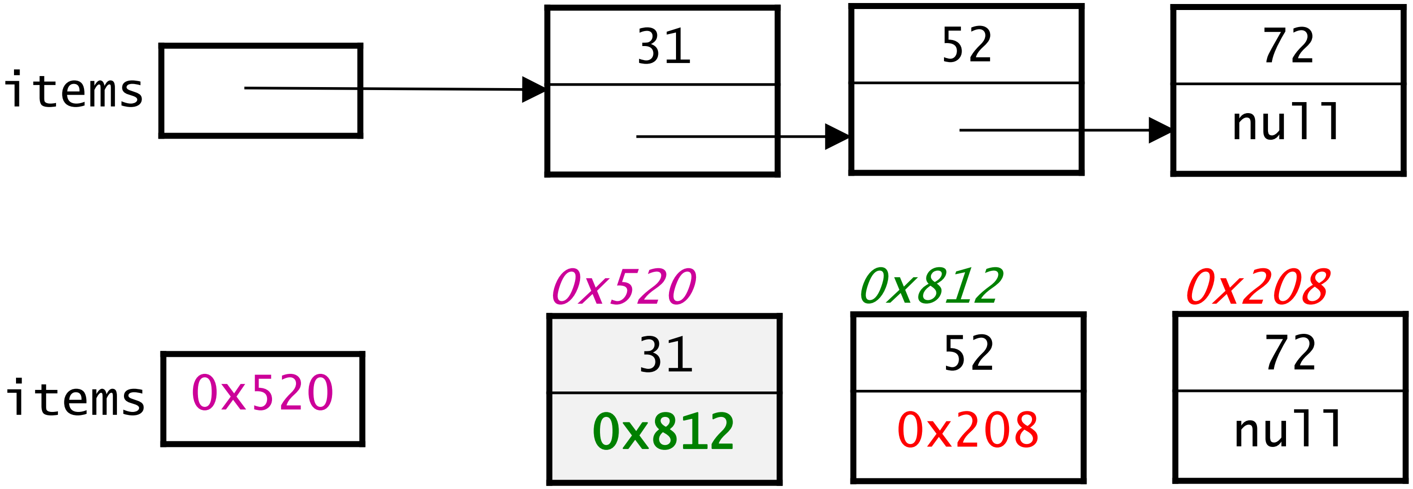
item

31	52	72	...

item 0x100

0x100	0x104	0x108	
31	52	72	...

- 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.



items

31

52

72
null

items 0x520

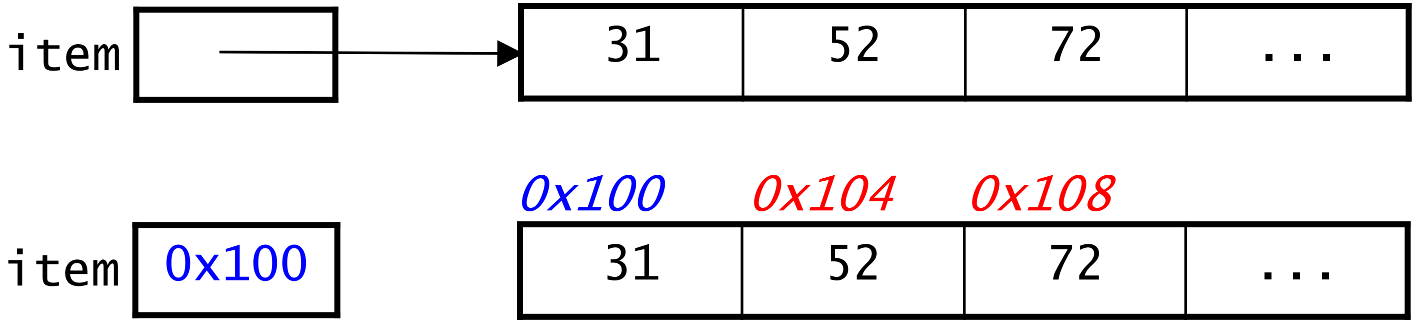
0x520
31
0x812

0x812
52
0x208

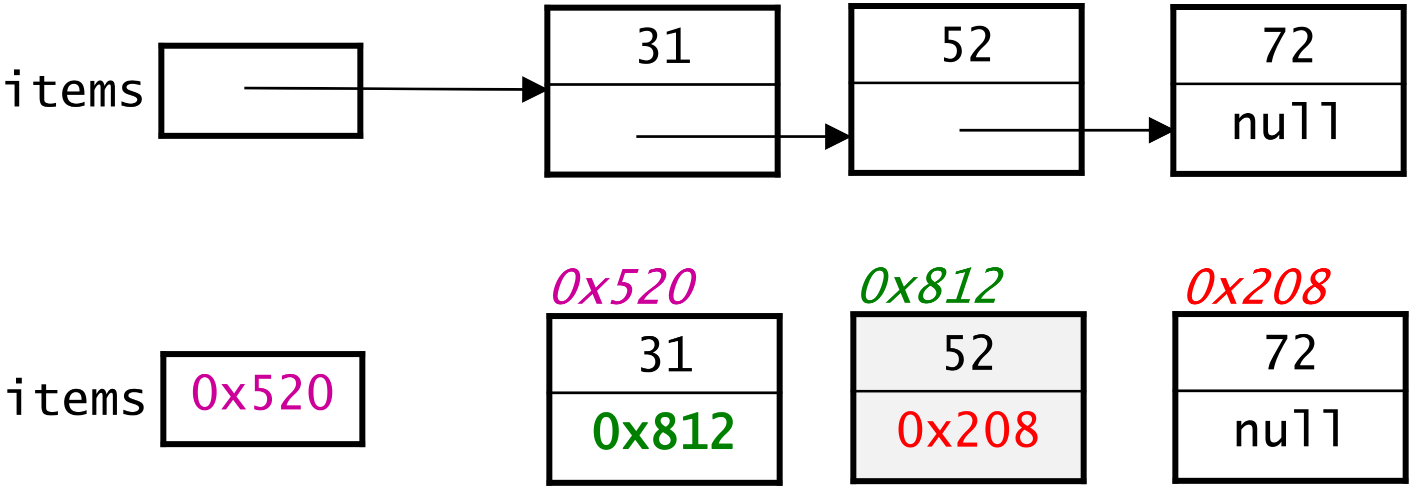
0x208
72
null

Arrays vs. Linked Lists in Memory

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

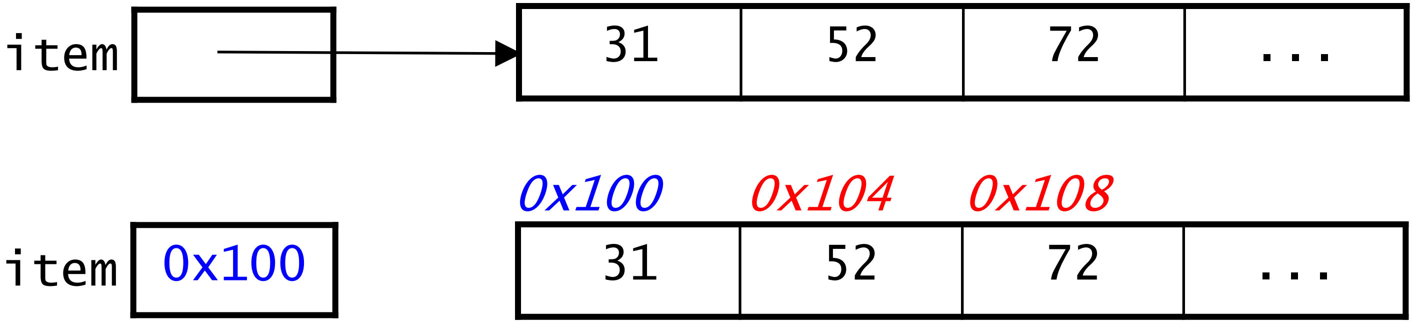


- In a linked list, each node is a *distinct object* on the heap.
  The nodes do *not* have to be next to each other in memory.
  That's why we need the links to get from one node to the next.

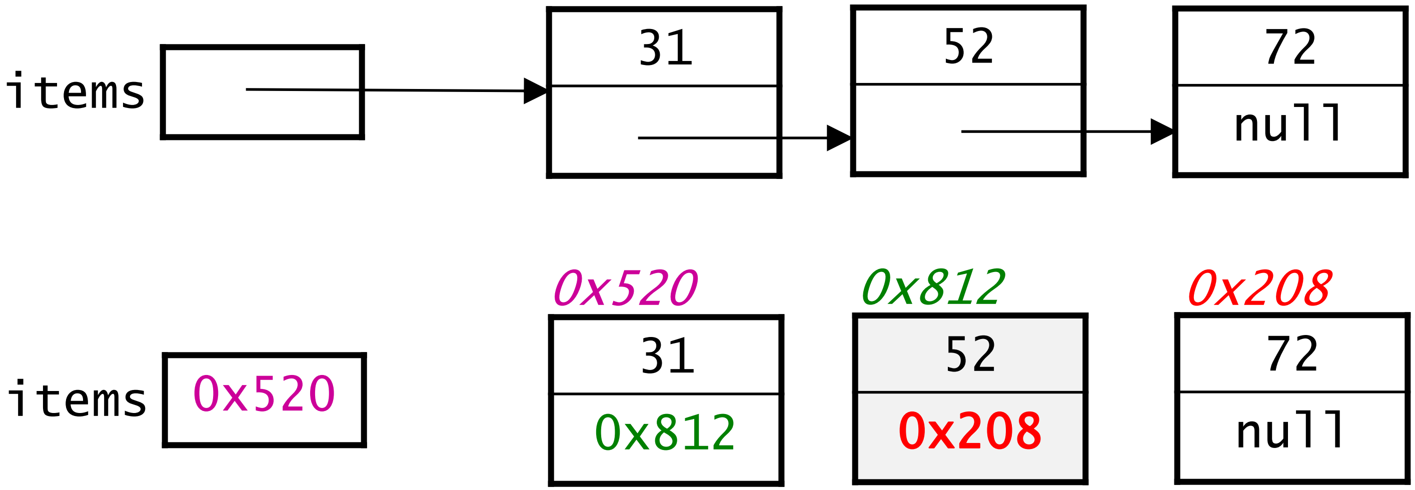


Arrays vs. Linked Lists in Memory

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

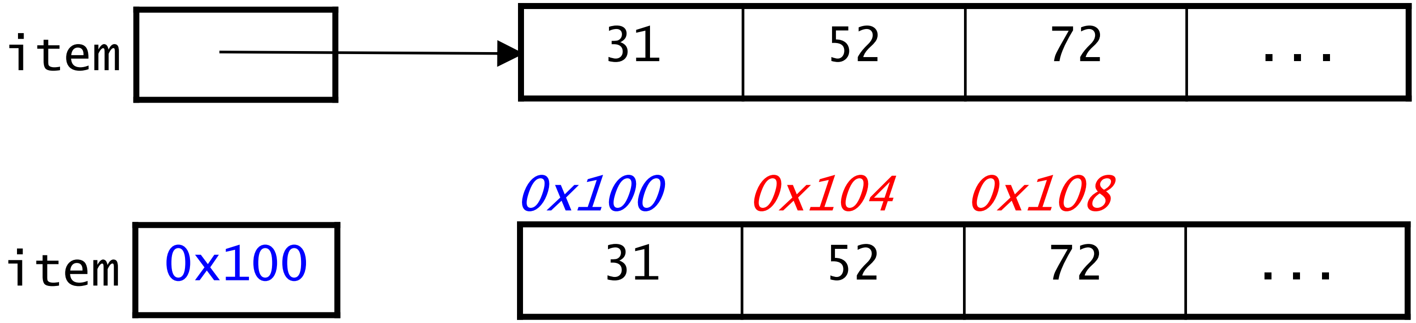


- In a linked list, each node is a *distinct object* on the heap. The nodes do *not* have to be next to each other in memory. That's why we need the links to get from one node to the next.

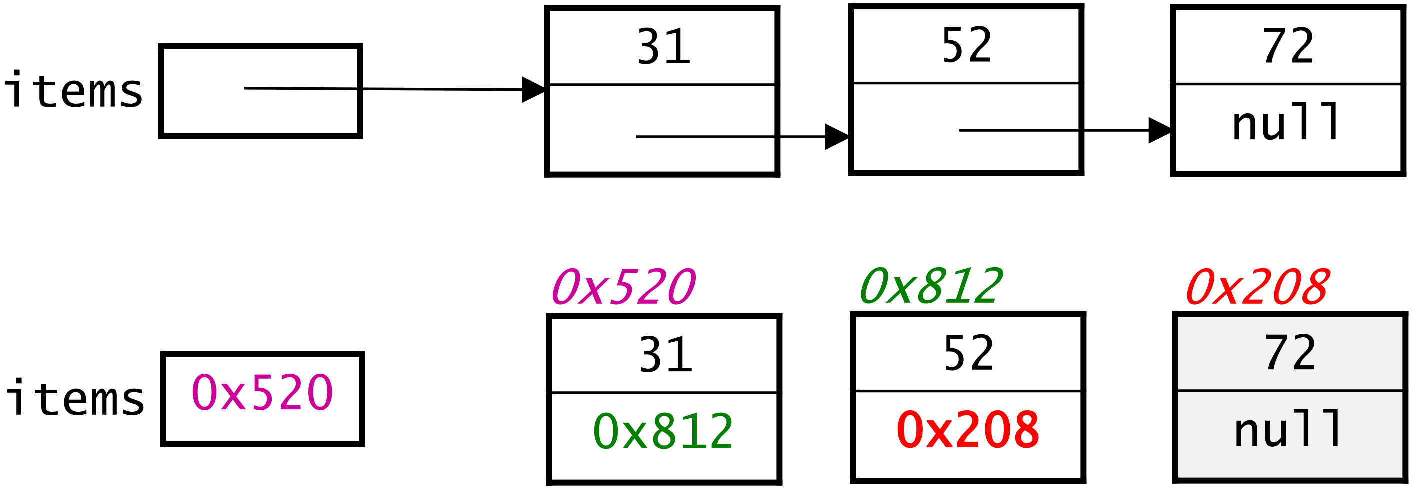


Arrays vs. Linked Lists in Memory

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

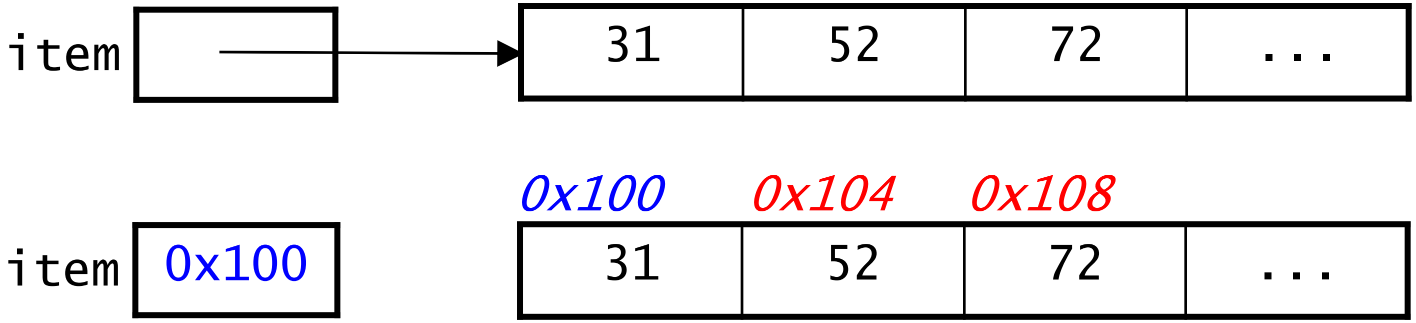


- In a linked list, each node is a *distinct object* on the heap. The nodes do *not* have to be next to each other in memory. That's why we need the links to get from one node to the next.



Arrays vs. Linked Lists in Memory

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



item

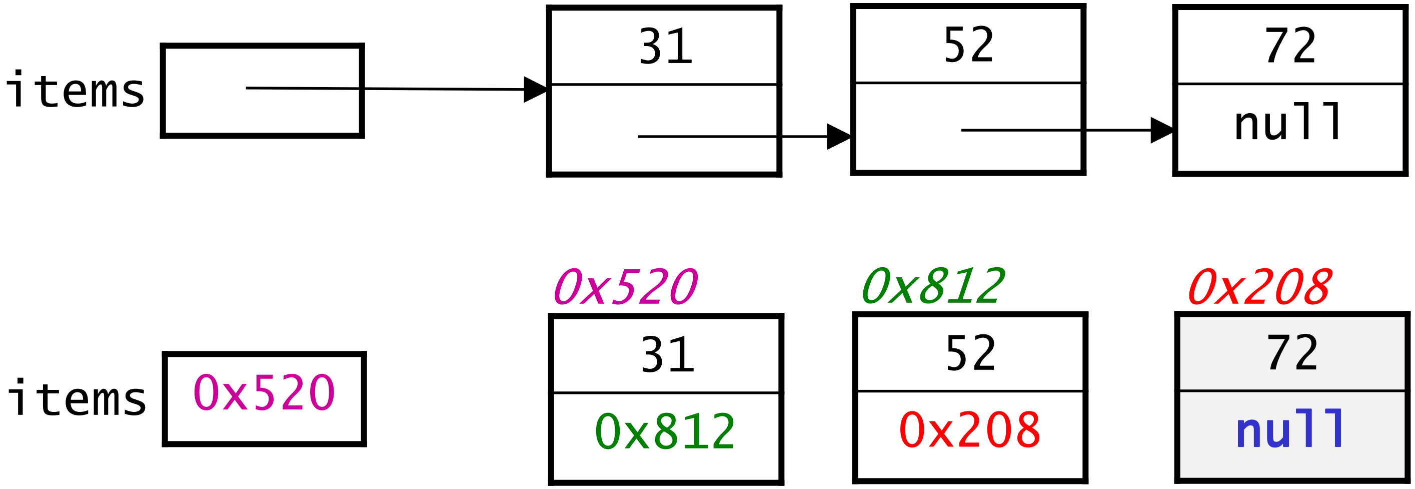
31 52 72 ...

0x100 0x104 0x108

item 0x100

31 52 72 ...

- 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.



items

31 52 72

null

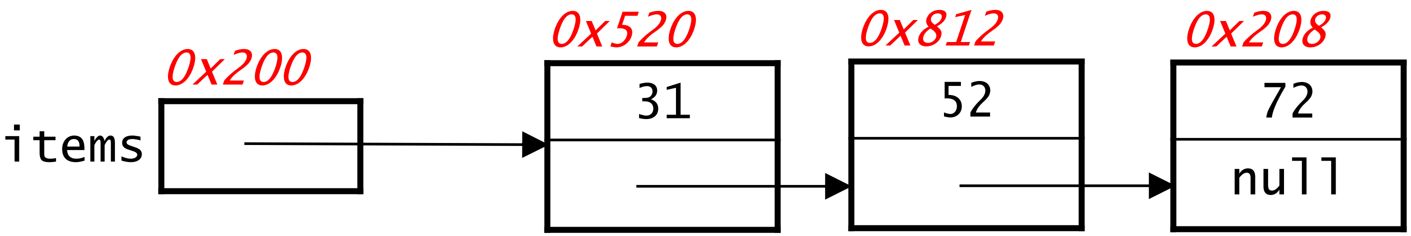
0x520 0x812 0x208

items 0x520

31 52 72

0x812 0x208 null

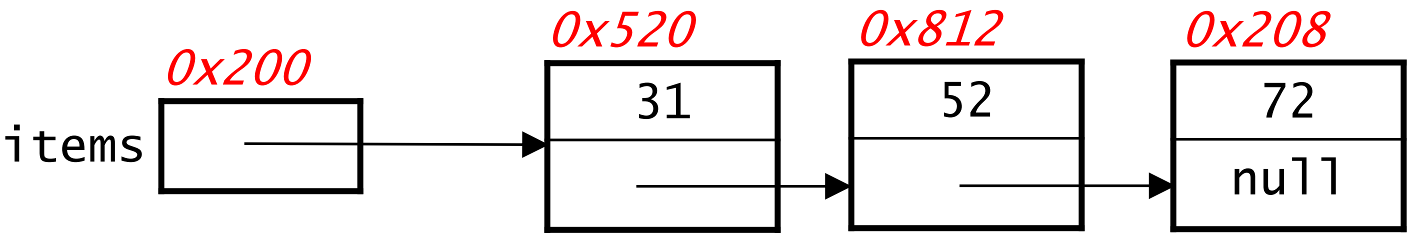
Linked Lists in Memory



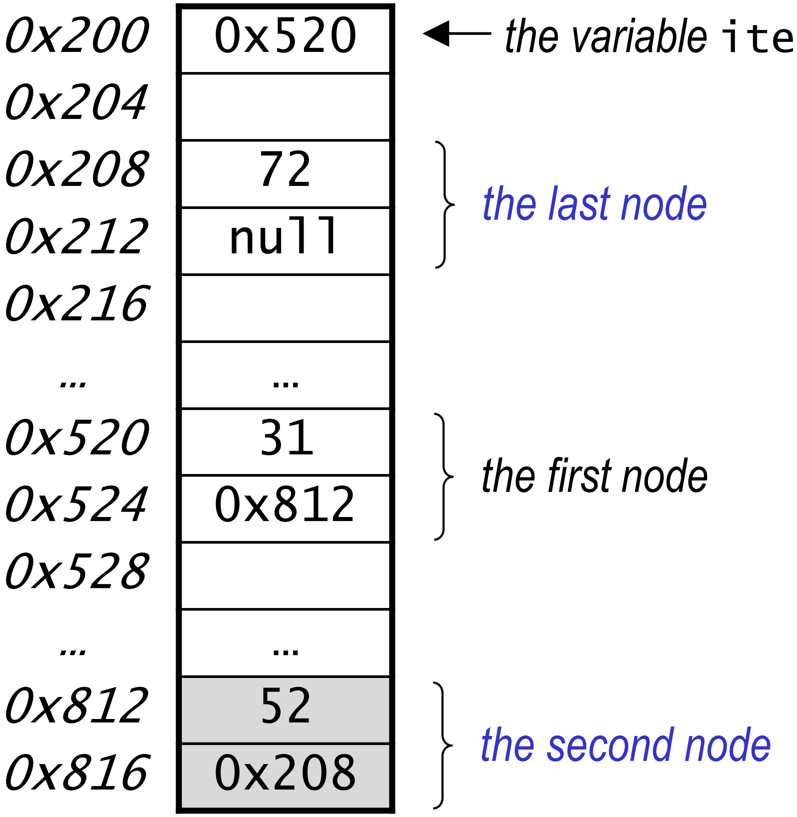
- Here's how the above linked list might actually look in memory:

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

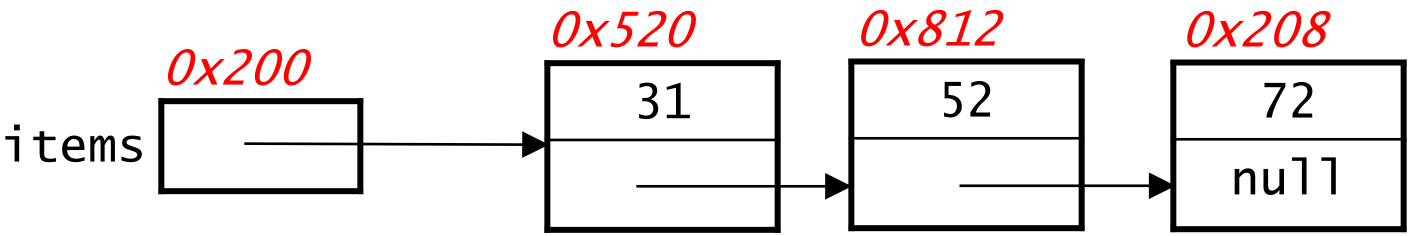


- Here's how the above linked list might actually look in memory:



0x200	0x520	← 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

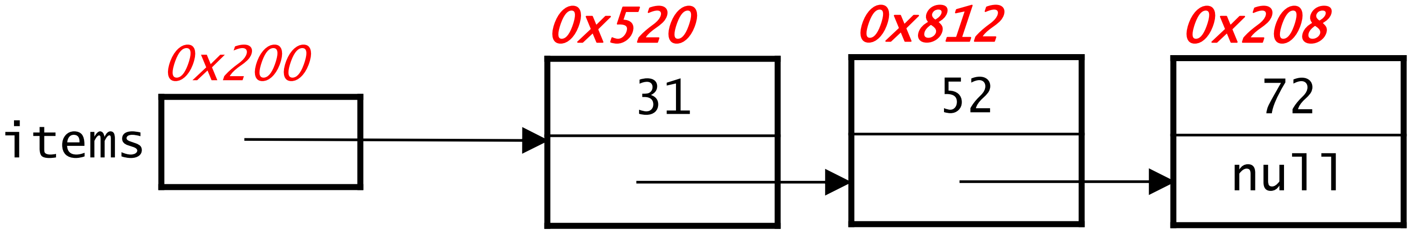


- Here's how the above linked list might actually look in memory:

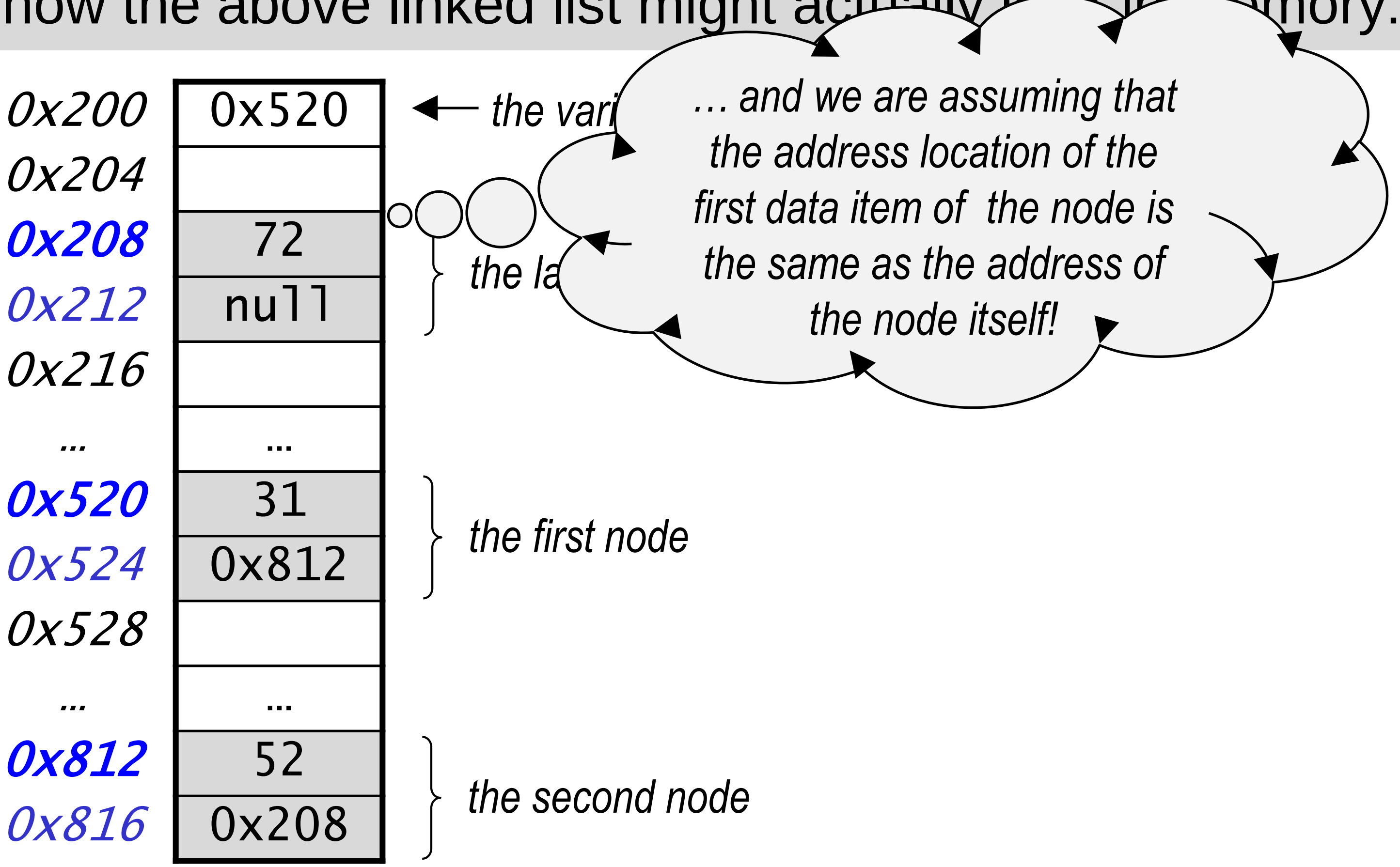
0x200	0x520	← *the vari*
0x204		
0x208	72	*the la*
0x212	null	
0x216		
…	…	
0x520	31	*the first node*
0x524	0x812	
0x528		
…	…	
0x812	52	*the second node*
0x816	0x208	

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

Linked Lists in Memory



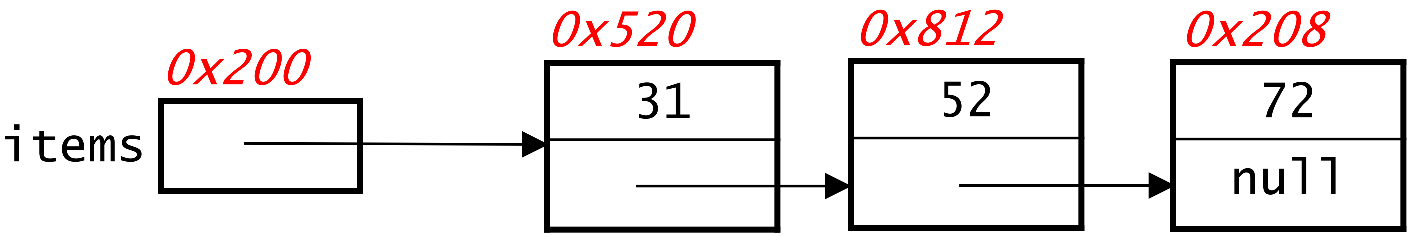
- Here's how the above linked list might actually look in memory:



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

… 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



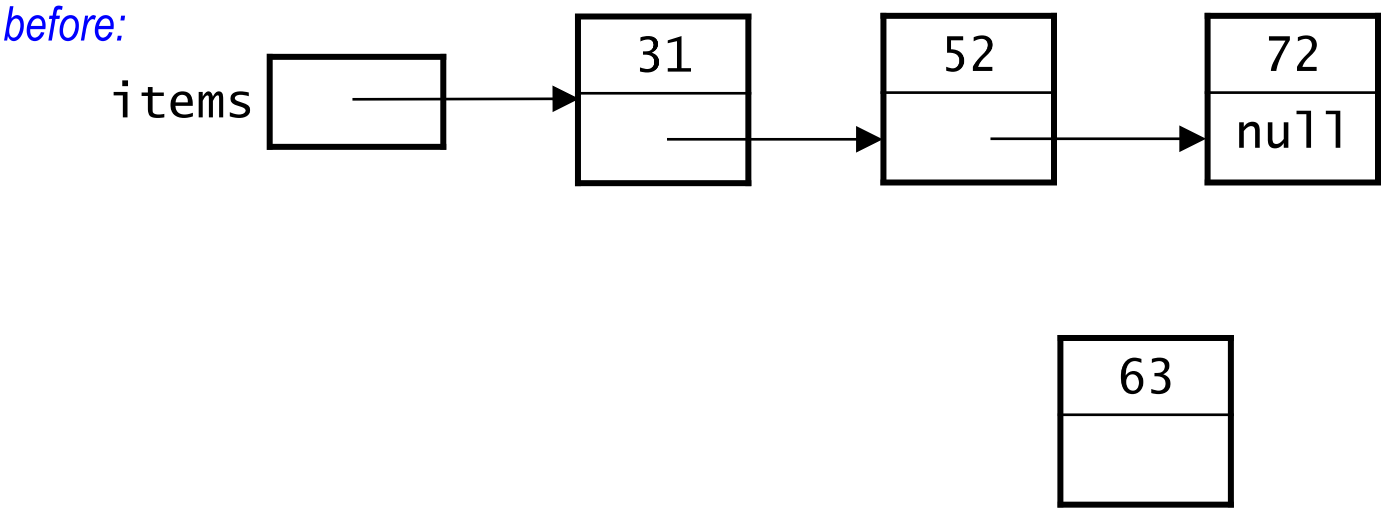
- Here's how the above linked list might actually look in memory:

0x200	0x520	← the vari
0x204		
0x208	72	the la
0x212	null	
0x216		
…	…	
0x520	31	the first node
0x524	0x812	
0x528		
…	…	
0x812	52	the second node
0x816	0x208	

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

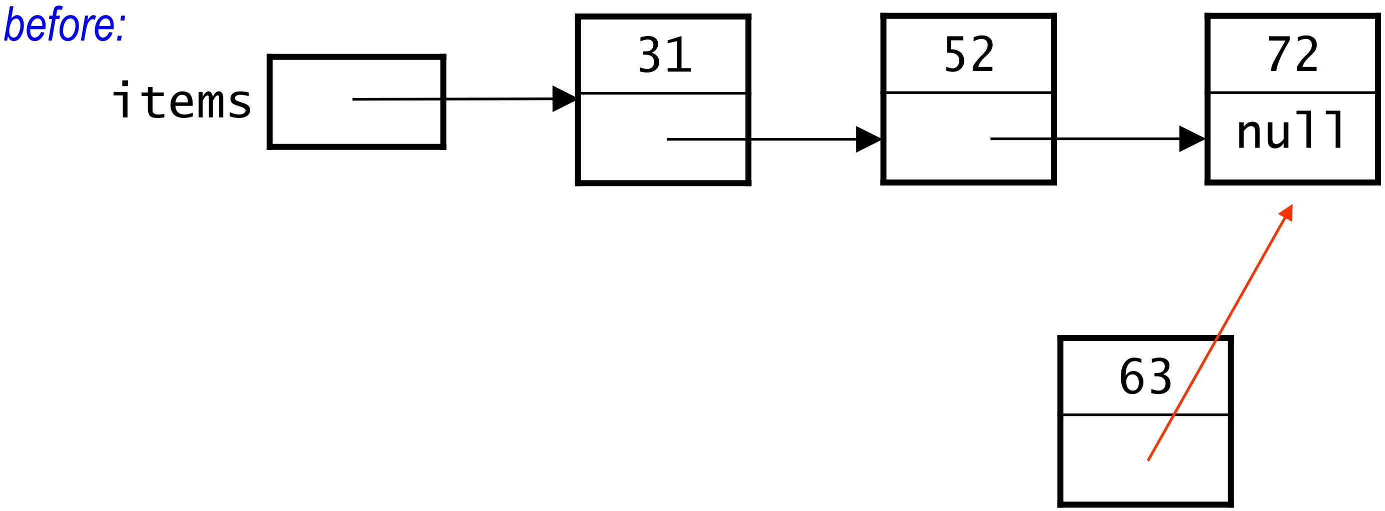
Features of Linked Lists

- 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:



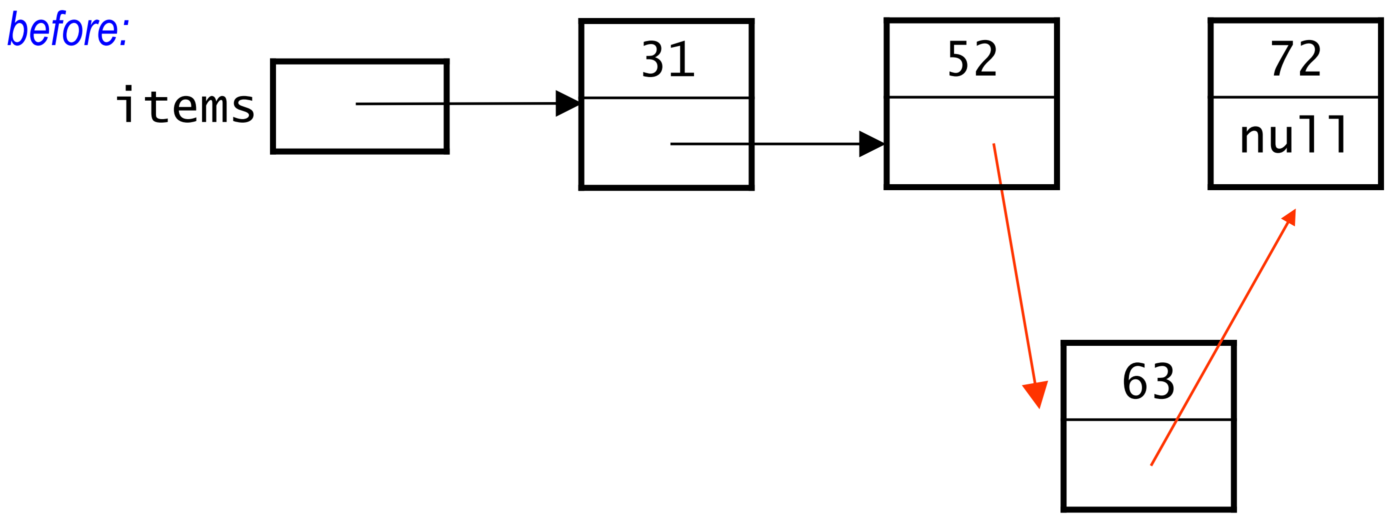
Features of Linked Lists

- 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:



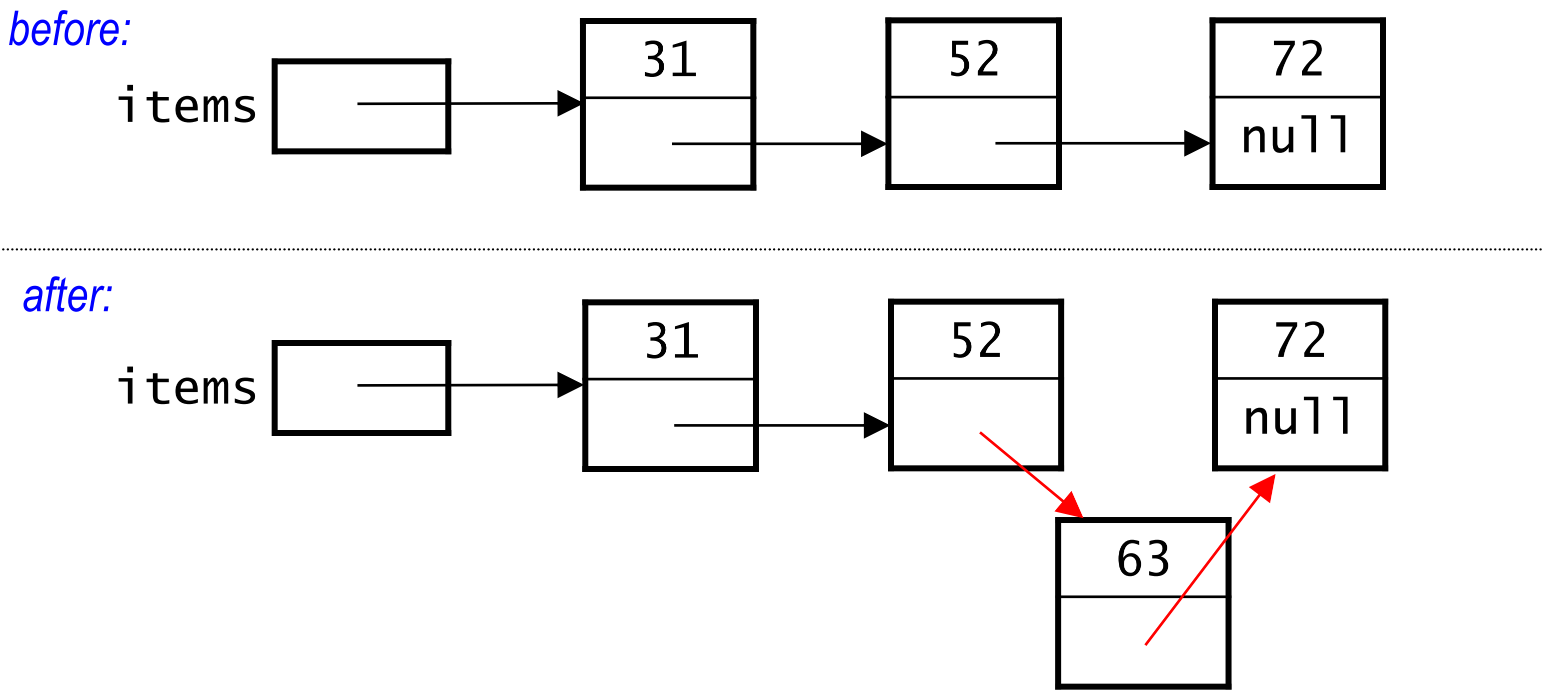
Features of Linked Lists

- 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:



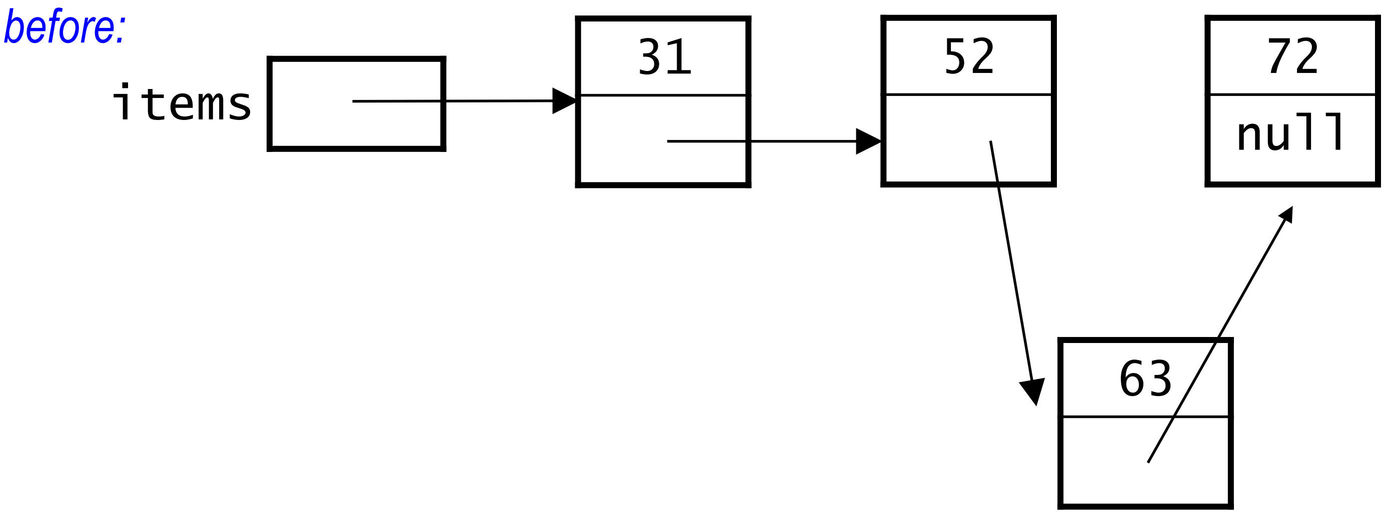
Features of Linked Lists

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



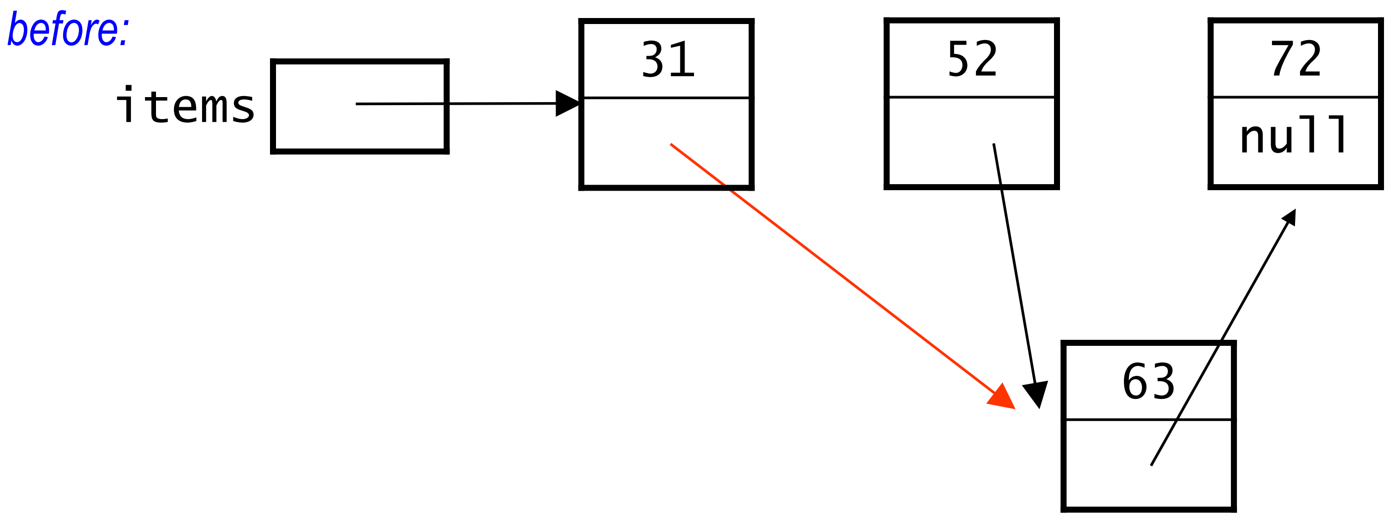
Features of Linked Lists

- 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:



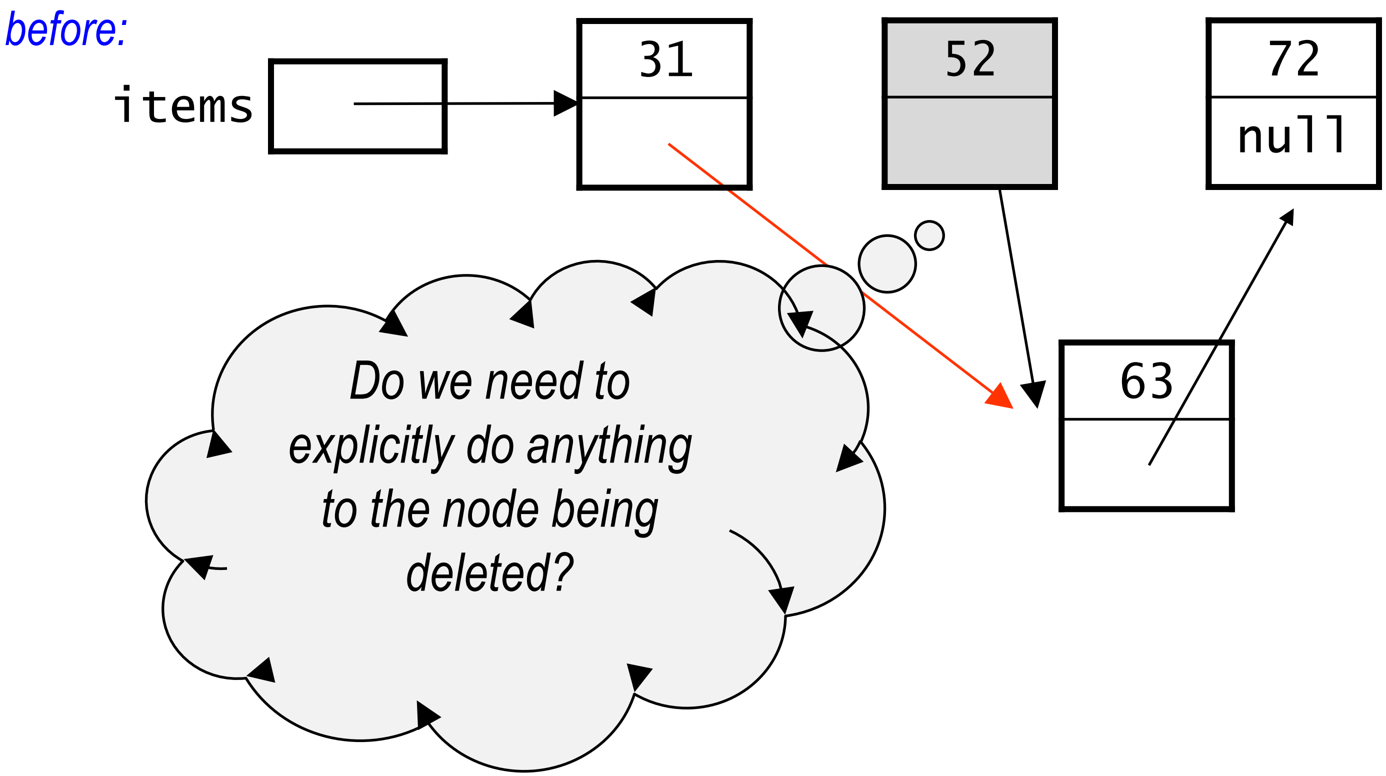
Features of Linked Lists

- 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

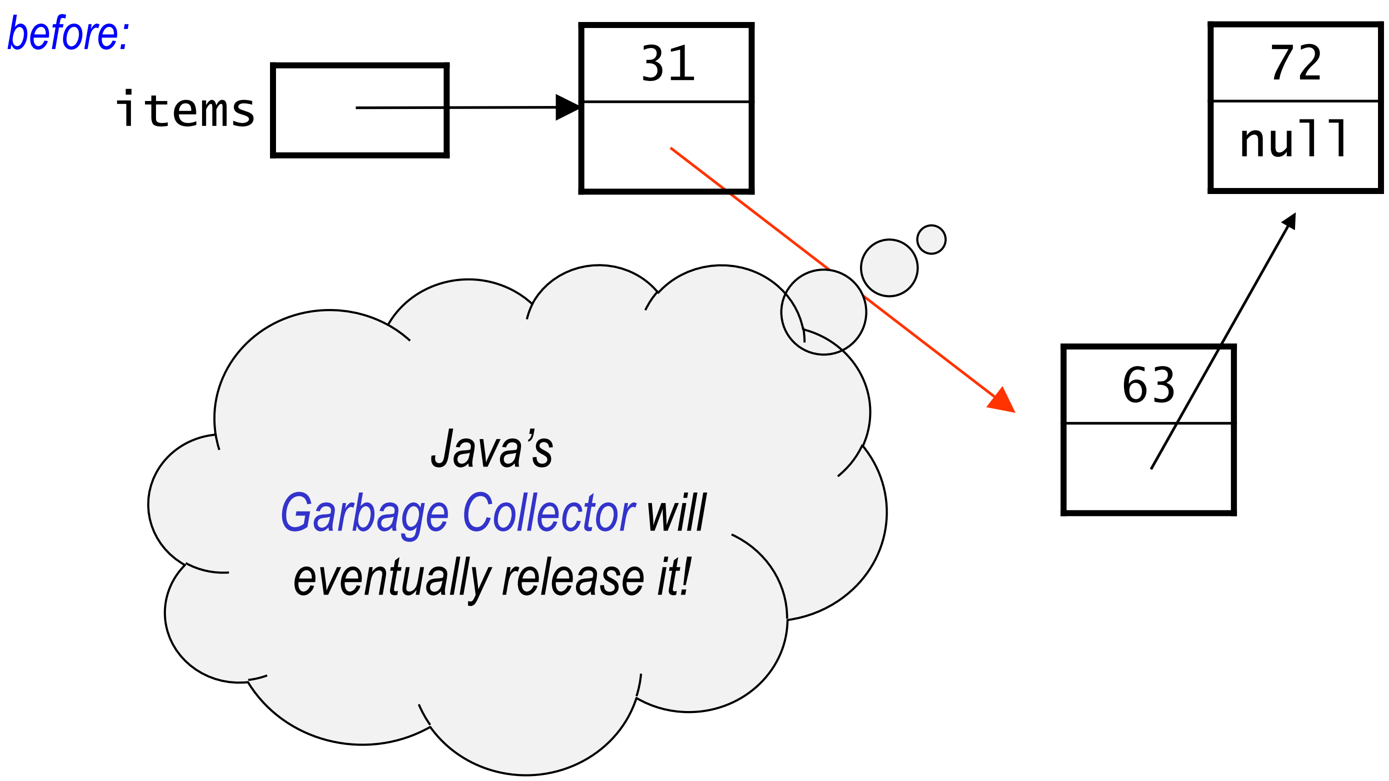
- They can grow without limit (provided there is enough memory).
- To delete an item – also no need to "shift over" other items.
  - for example, to delete node 52:



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:



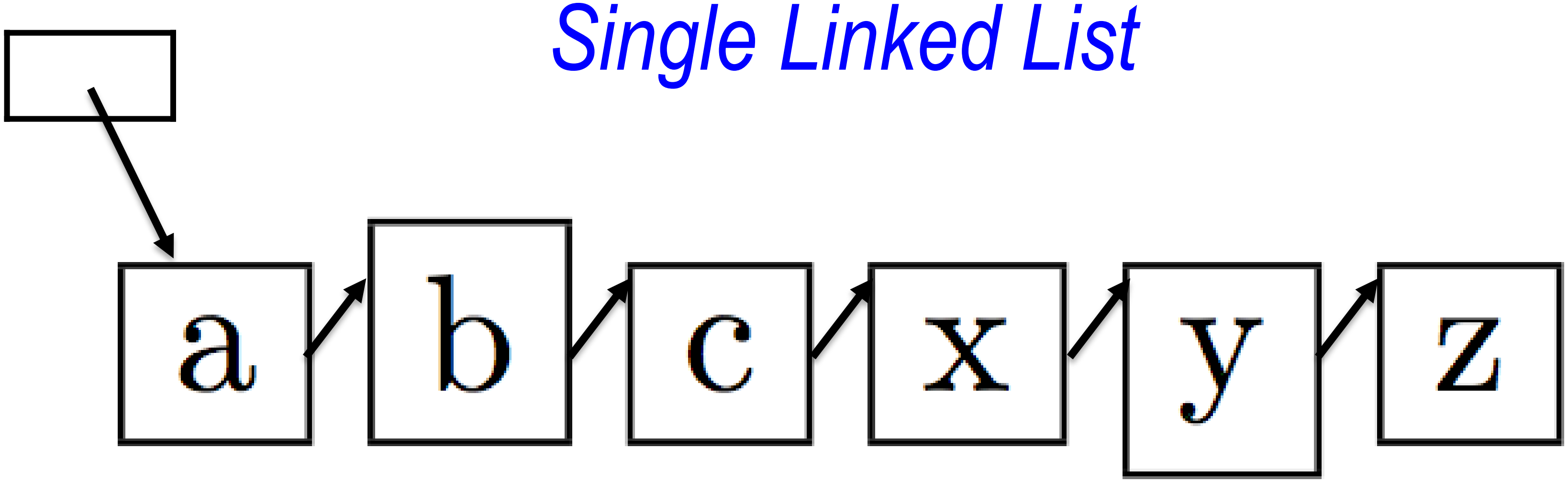
Features of Linked Lists

- 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.

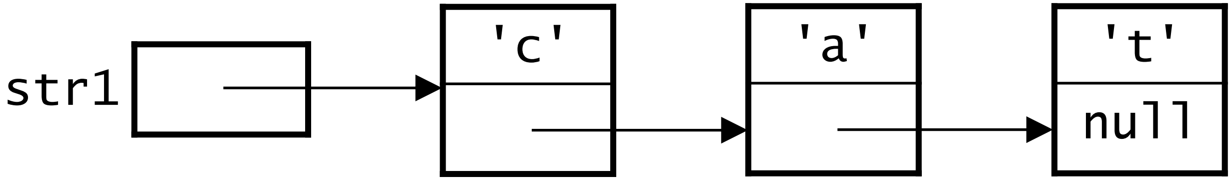
Single Linked List



head of the list

Example:

A String as a Linked List of Characters

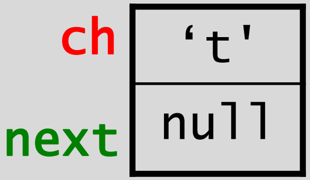


- Each node in the linked list represents one character.

- Java class for this type of node:

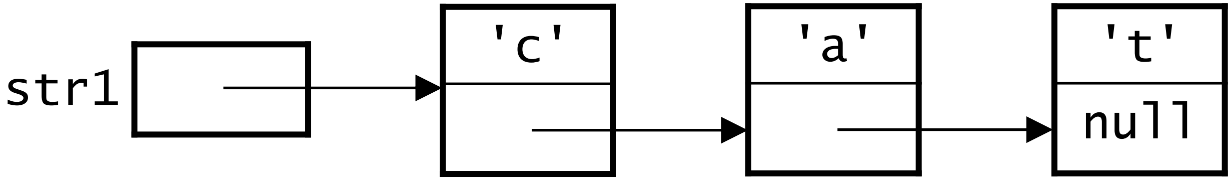
```
public class StringNode {
    private char ch;
    private StringNode next;

    // constructor to initialize the members



    ...
}
```

same type as the node itself!



Example:

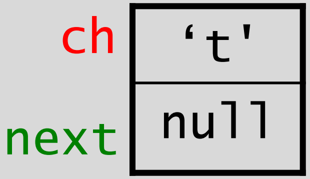
A String as a Linked List of Characters



- Each node in the linked list represents one character.
- Java class for this type of node:

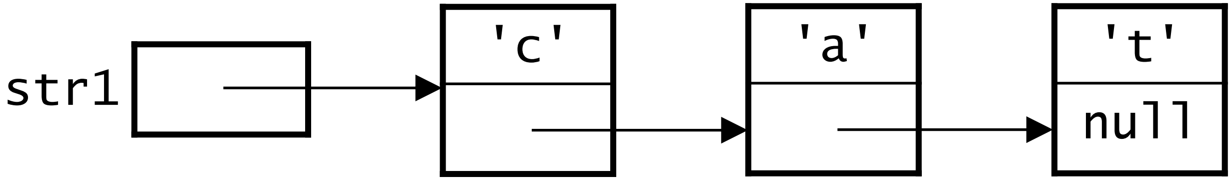
```
public class StringNode {
    private char ch;
    private StringNode next;

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

same type as the node itself!



Example:

A String as a Linked List of Characters

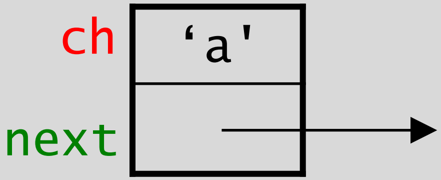


- Each node in the linked list represents one character.

- Java class for this type of node:

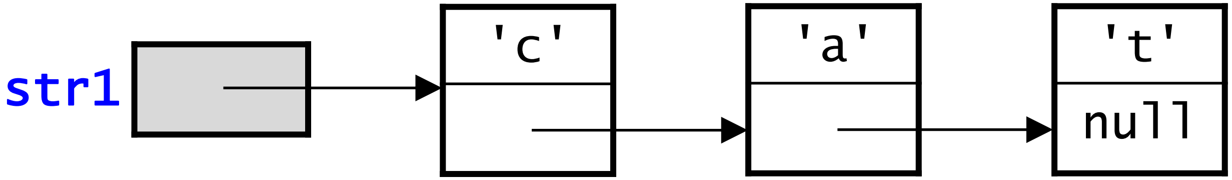
```
public class StringNode {
    private char ch;
    private StringNode next;

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

same type as the node itself!



Example:

A String as a Linked List of Characters

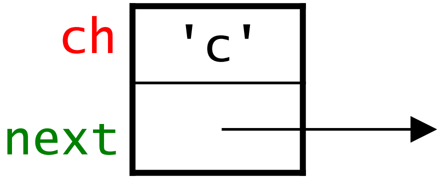


- Each node in the linked list represents one character.

- 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!

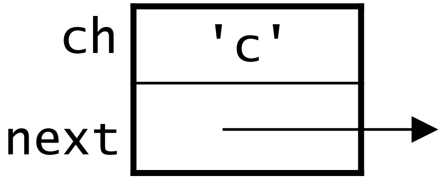


- The string as a whole is represented by a variable that holds a reference to the node for the first character (e.g., `str1` above).

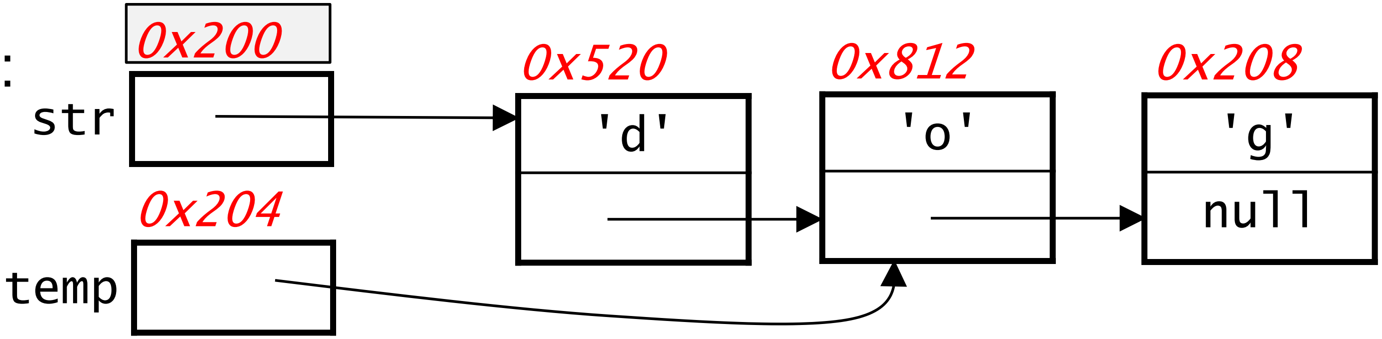
Review of Variables

- 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)

Review of Variables

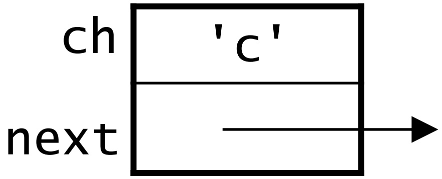


- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:



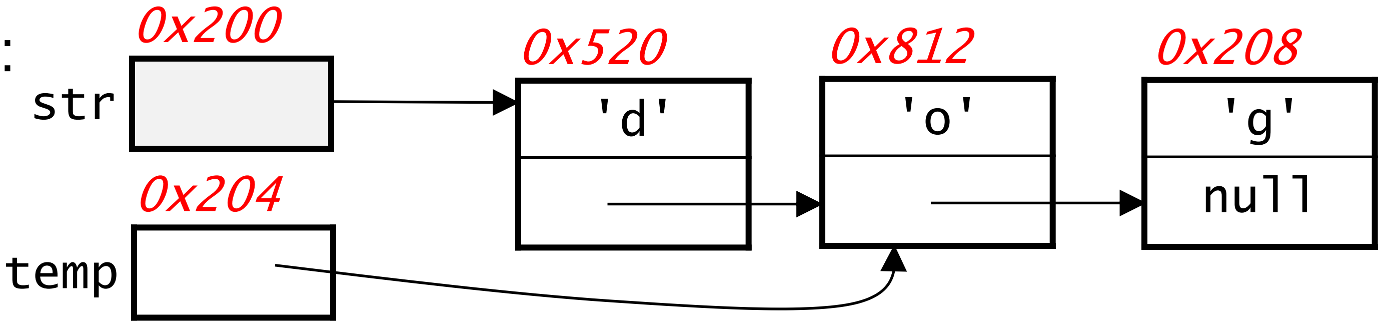
```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

Review of Variables



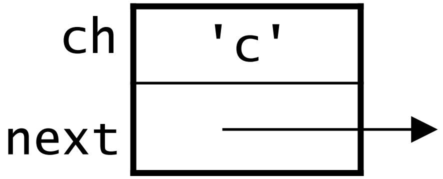
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



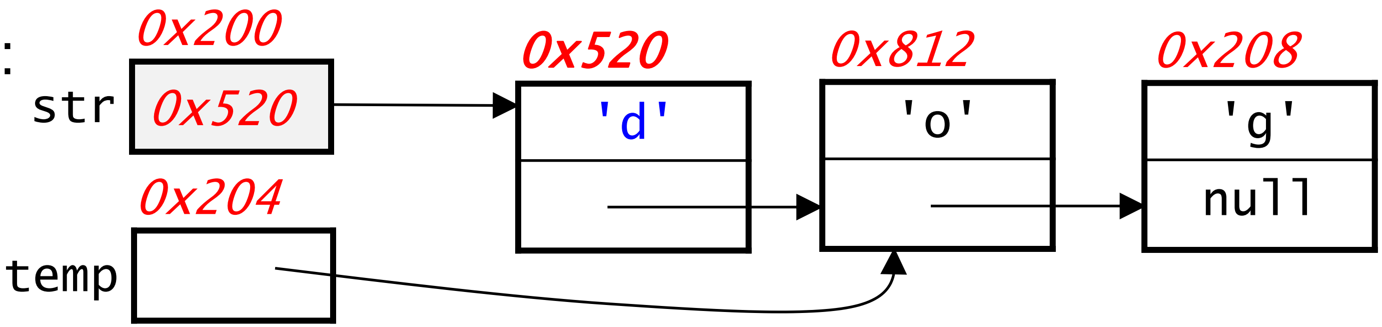
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



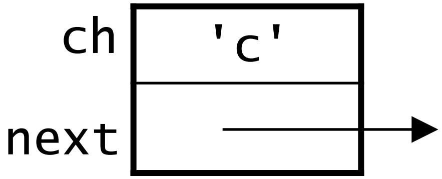
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:

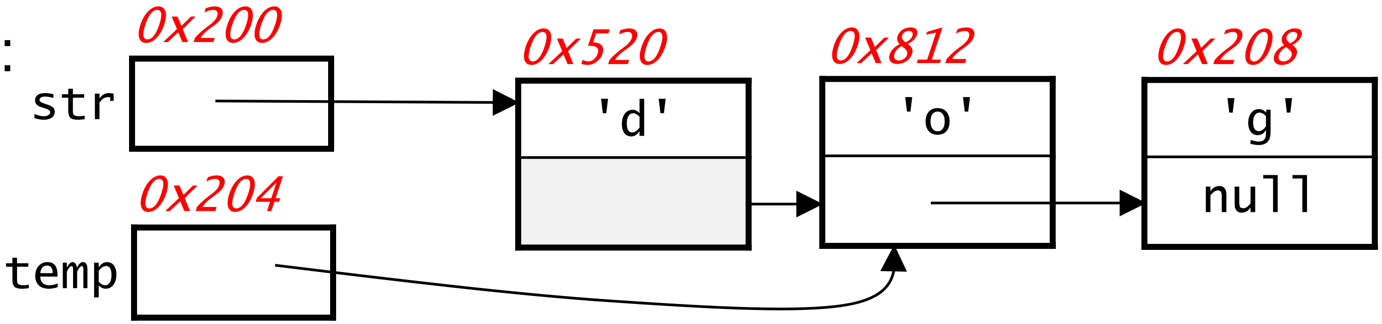


```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables

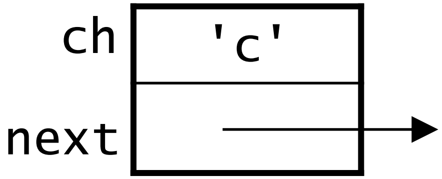


- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:



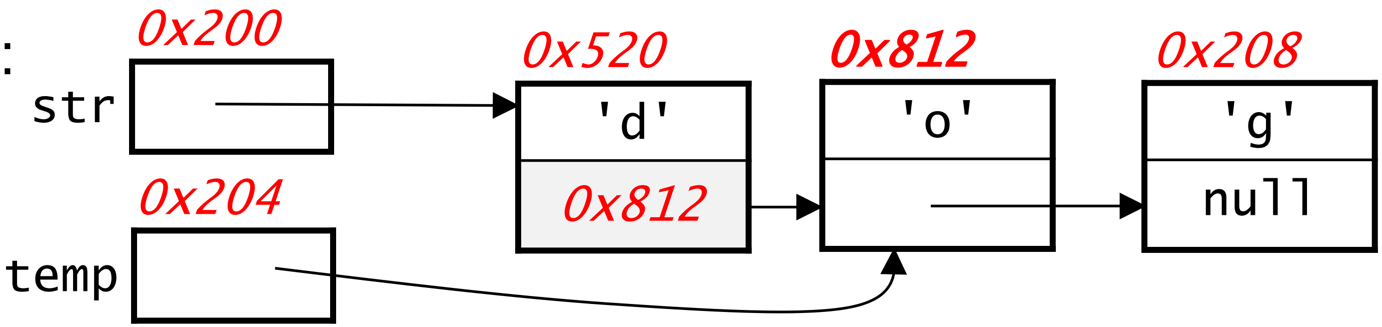
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



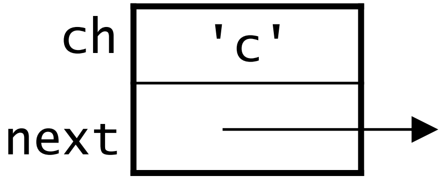
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



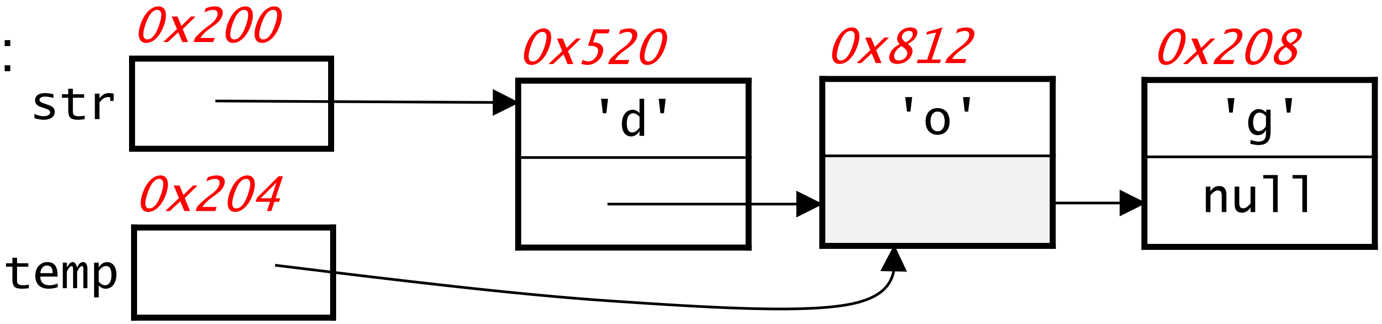
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



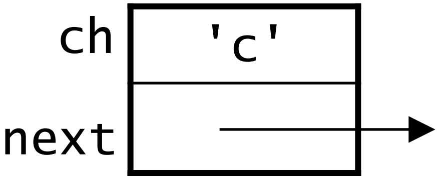
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



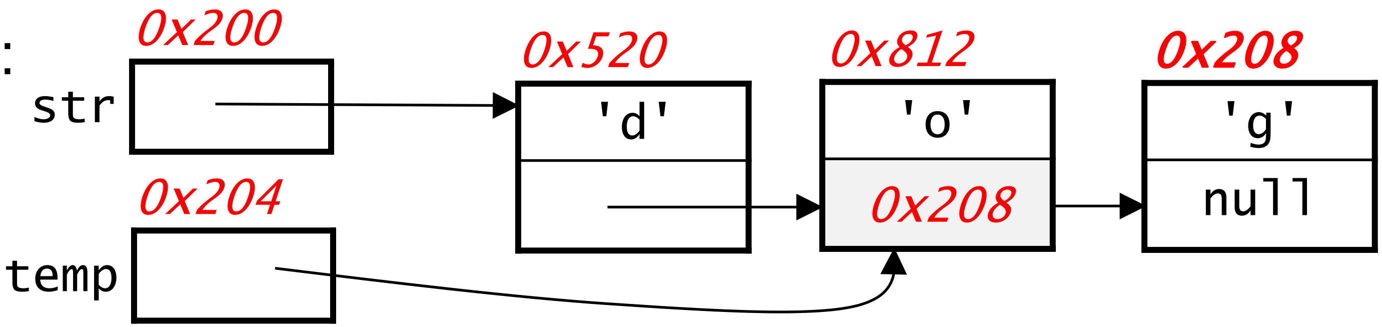
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



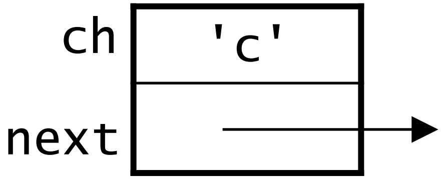
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:

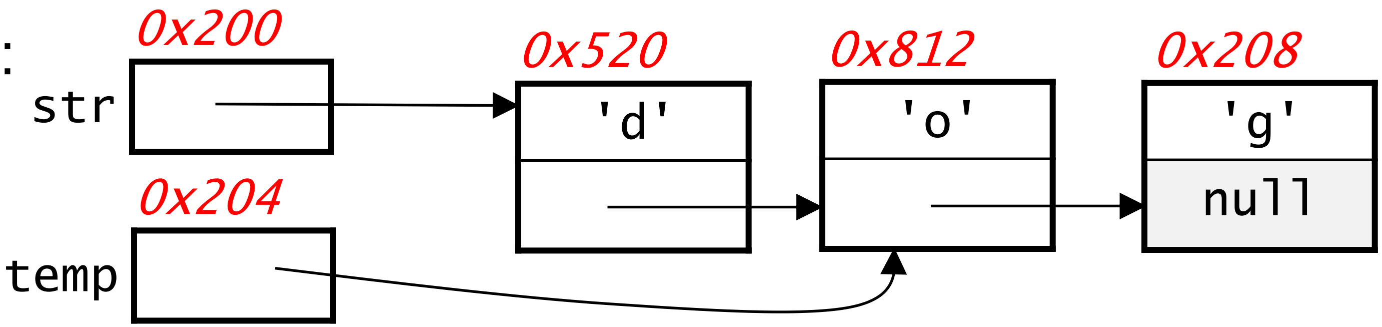


```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables

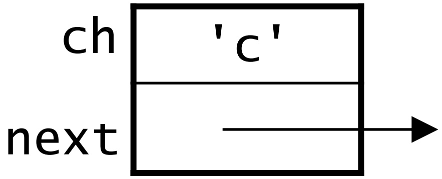


- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:

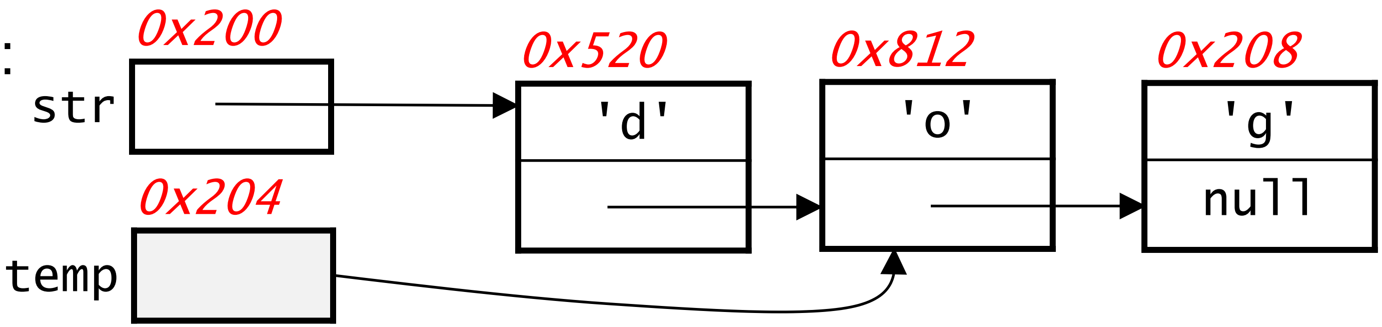


```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables

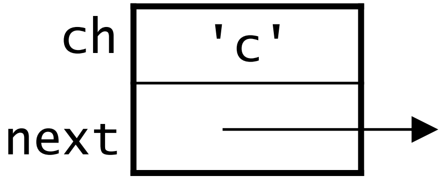


- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)
- Practice:



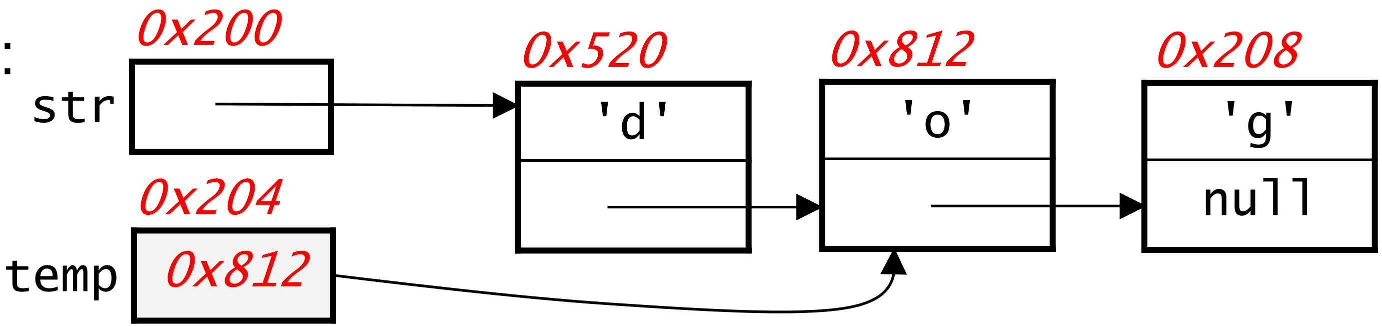
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

Review of Variables



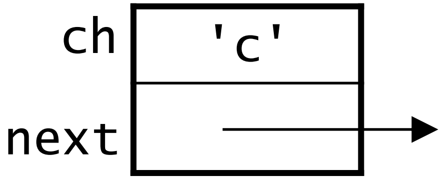
- A variable or variable expression represents both:
  - a "box" or location in memory (the *address* of the variable)
  - the contents of that "box" (the *value* of the variable)

- Practice:



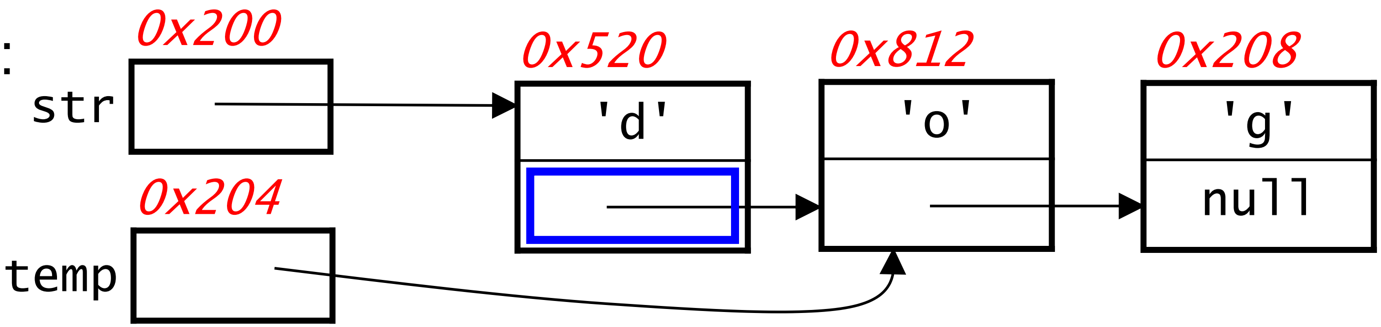
```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```

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)

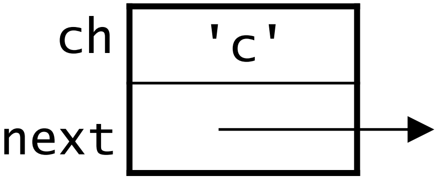
- Practice:



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

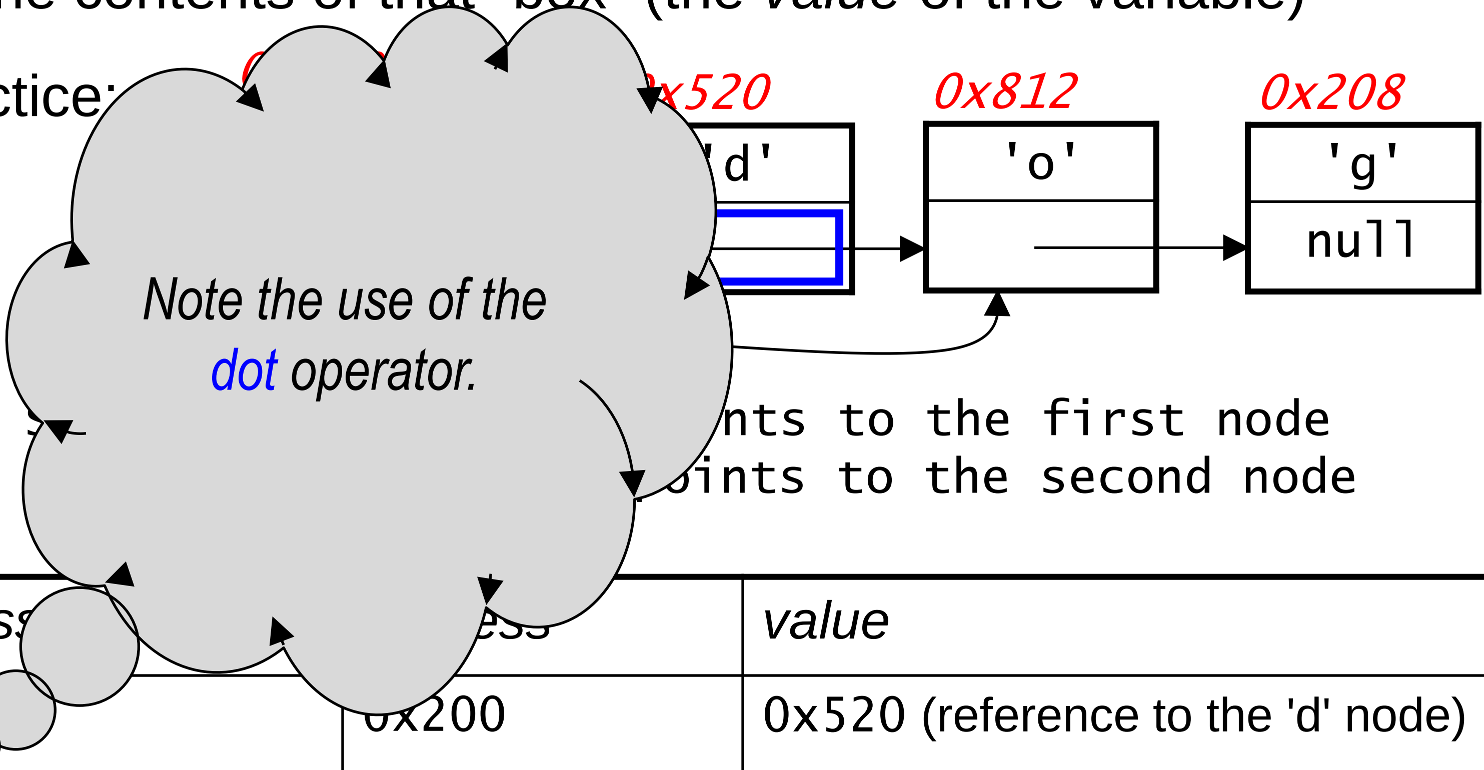
expression	*address*	*value*
`str`	0x200	0x520 (reference to the 'd' node)
`str.ch`	0x520	'd'
`str.next`	**0x522**	**0x812 (reference to the 'o' node)**

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)