Data Structures

Linked List Basics

Array Disadvantages

- Arrays, although easy to understand have lots of disadvantages
  - Contiguous Memory Allocation
    - Advantage in searching but disadvantage in terms of memory usage
  - Memory Usage is Fixed
    - char name[25] will take 25 spaces even though name you enter is Ali
    - Should be that only 3 bytes be used.

Array Disadvantages… cont

- Now consider

```
Struct person
{
        char name[25];
        int age;
};
Void main (void)
{
        person p[100];
   …
}
```

QUESTIONS:

How many persons you can enter (Maximum)?

What will be the space occupied in memory After person p[100] in main ?

What if you only enter two persons… What Will be the memory size then?

- Even if the data entered is for two persons the amount of memory taken is 100 x (25 + 4)
- Must be a method in C/C++ through which we can allocate memory according to our needs at RUN TIME!
  - Program should ask user… do you want to enter more data… if user selects ‘Yes’ then it should allocate memory for one more person.
  - This way memory will not be ‘wasted’

Need of Dynamic Memory Allocation

- Reason we study pointers is for this step (mostly)
- C gives us function malloc (Memory Allocate) and C++ gives us Keyword New
- We will use Malloc initially
- Why malloc?
  - New keyword is not available in most of the embedded system programming.
  - Writing device drivers??…. They cannot be Object Oriented, hence we must know Malloc

How to Allocate Memory Dynamically

- However, linked lists are not without their drawbacks.
- For example,
- we can perform efficient searches on arrays (e.g., binary search) but this is not practical with a linked list.
- Random accessing is not possible as there is no index.
- Extra overhead with memory allocation and de-allocation, as memory is allocated and de-allocated separately for each item.

Disadvantages…

Memory View Of Linked List

Memory Loc	Value	Name
650	**1000**	Pointer to Linked list first element
…	**…**	…
1000	**Ali**	Person.Name
1026	**25**	Person.Age
1030	**5500**	Next_Person
…	**…**	…
5500	**Zain**	Person.Name
5526	**5**	Person.Age
5530	**NULL**	Next_Person
…	**…**	…

- One of the attributes of a linked list is that there is not a physical relationship between the nodes; that is, they are not stored contiguously in memory (as array elements are).
- To determine the beginning of the list, and each additional element in the list, we need to use pointers.

Linked list data structure

- The pointer to the first node in the list is referred to as the ***head pointer***, because it points to the head node in the list.
- In addition to the head pointer, there are usually other pointers associated with the list. These can include a pointer to the last element in the list (tail pointer) and a pointer that traverses the list to find data (navigator or traversal pointer).

Linked list data structure…

- Oftentimes it is convenient to create a structure that contains the head pointer of a list and also information about the list itself (e.g., number of elements currently in the list, maximum number of elements to be allowed in the list, etc).
- This extra data is referred to as ***metadata***.

Linked list data structure…

Operations

- With a linked list, there are a standard set of functions that operate on the list:
  - Creating the list
    - Initialize pointers to NULL;
  - Inserting nodes
    - Insert at beginning
    - Insert at middle
    - Insert at last
    - Insert in order
  - Deleting nodes
    - Delete from beginning, middle, last
    - Deleting by key
  - Traversing the list
  - Updating the information of an item if exists
  - Reversing
  - Destroying the list etc.

Linked List

- Consists of items that are *linked* to each other
- Each item on the list is associated with a reference (pointer) that indicates where the next item is found
- A *dynamic* data structure: grows and shrinks as necessary at runtime
- There are no unused/empty locations

Singly Linked List

- Simplest form of linked list
- Linked list object holds a reference to the first *node* on the list
- Each node object consists of a reference to the data object being stored, and a reference to the next node in the list

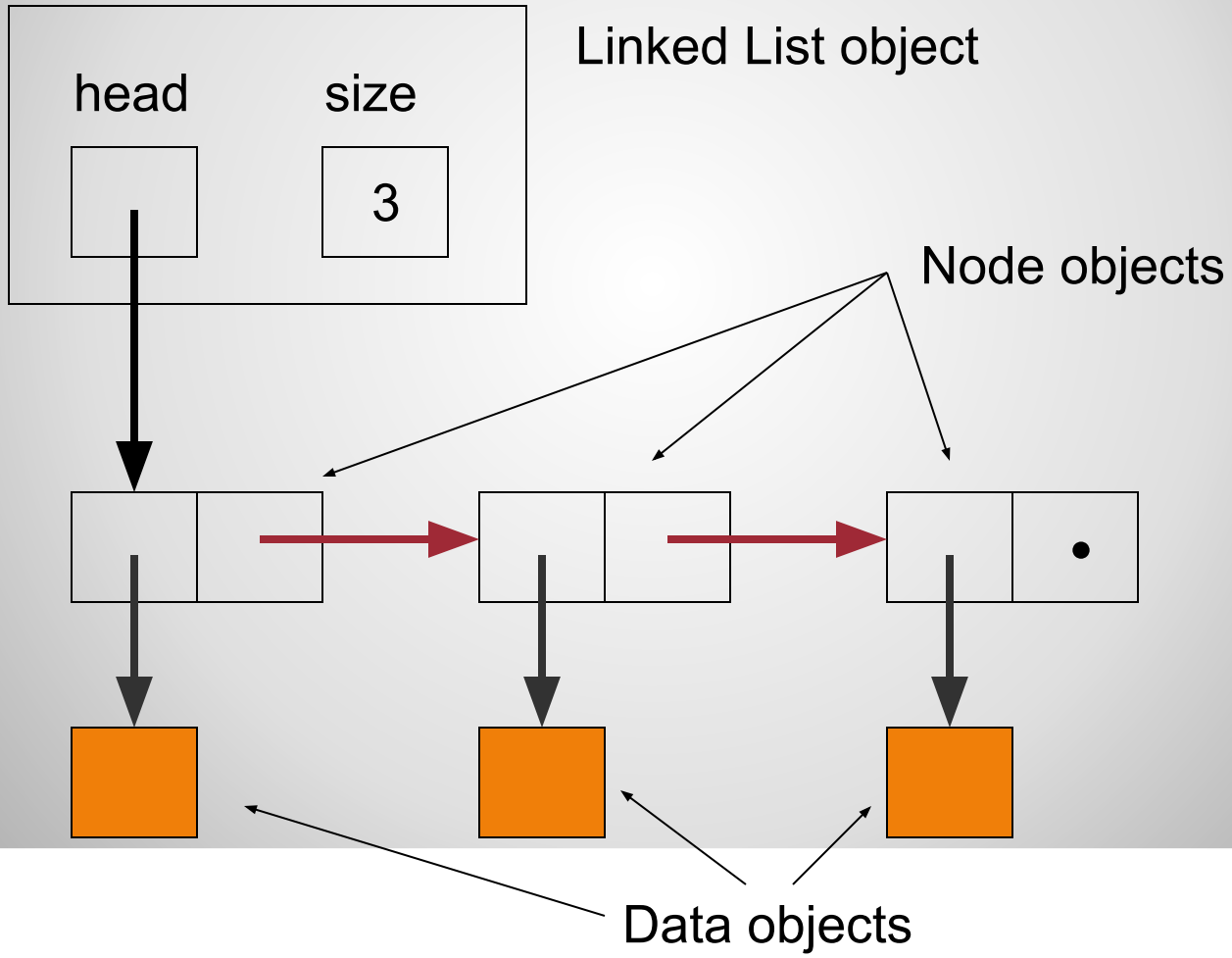
Linked list data structure…

- The pointer to the first node in the list is referred to as the ***head pointer***, because it points to the head node in the list.
- In addition to the head pointer, there are usually other pointers associated with the list. These can include a pointer to the last element in the list (tail pointer) and a pointer that traverses the list to find data (navigator or traversal pointer).

Memory View Of Linked List

Memory Loc	Value	Name
650	**1000**	Pointer to Linked list first element
…	…	…
1000	**Ali**	Person.Name
1026	**25**	Person.Age
1030	**5500**	Next_Person
…	…	…
5500	**Zain**	Person.Name
5526	**5**	Person.Age
5530	**NULL**	Next_Person
…	…	…

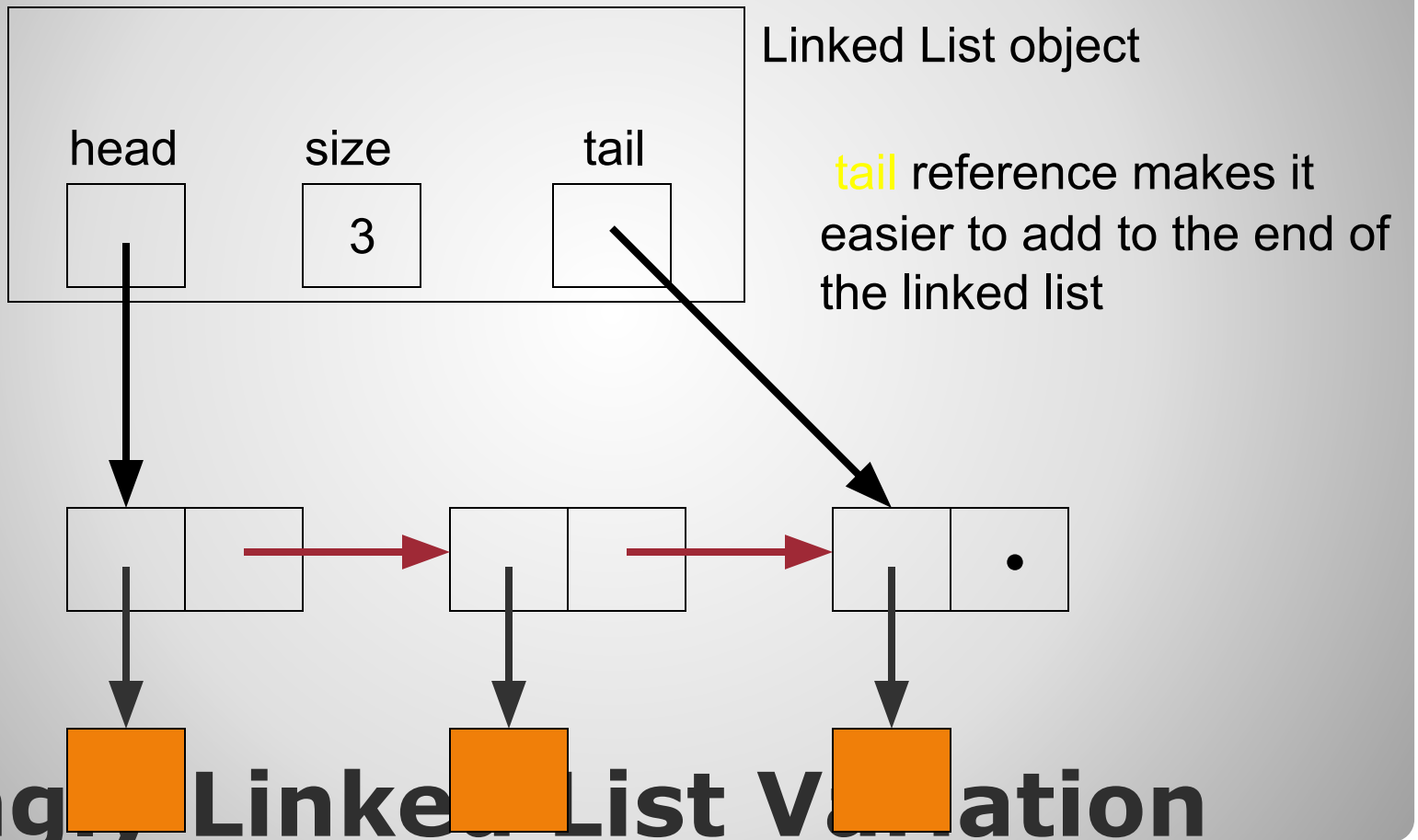
Singly Linked List



Singly Linked List

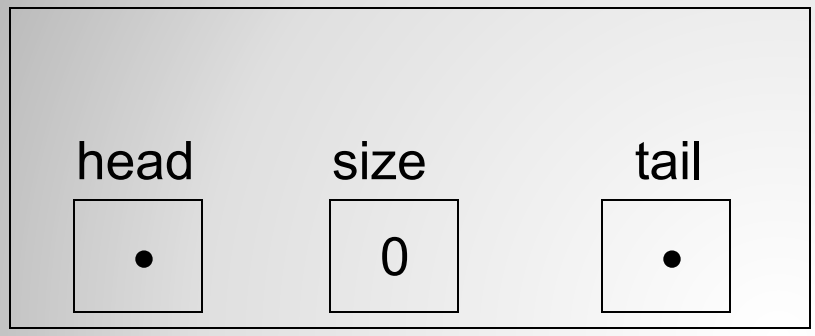
- To find the n^{th} item in a linked list:
  - Follow the head pointer to the first node
  - Find the reference to the next node
  - Follow it to the second node
  - Find the reference to the next node
  - Follow it to the third node
  - Etc, until n^{th} node is reached
  - Data pointer points to the n^{th} item

Singly Linked List Variation

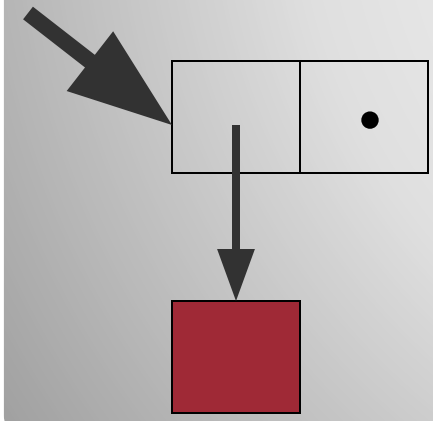


Linked list data structure…

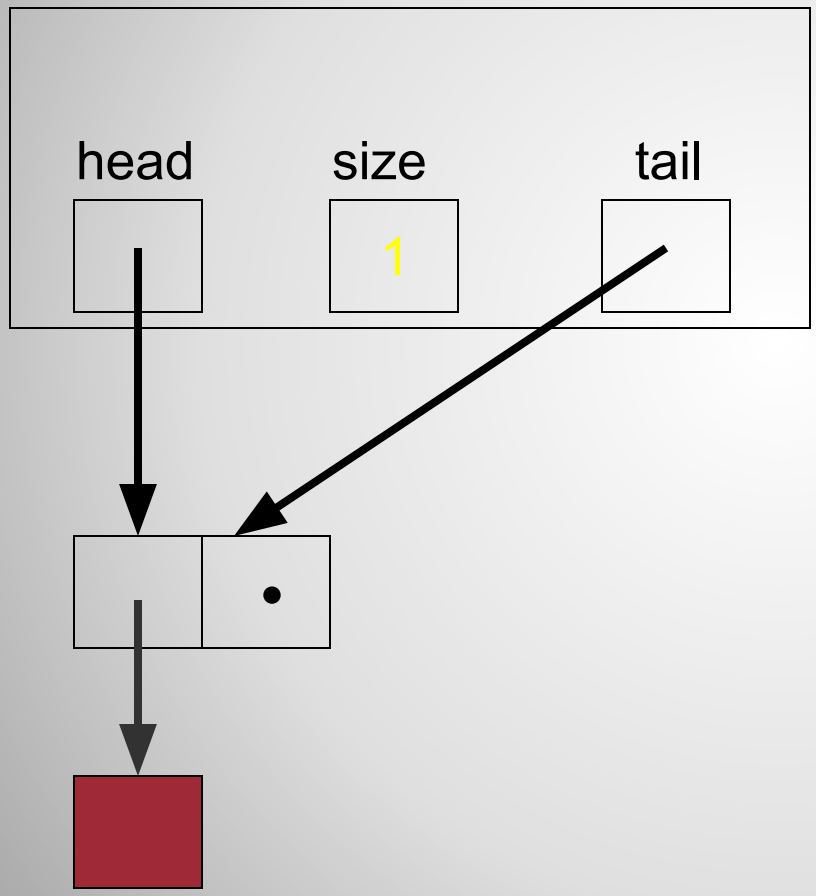
- Oftentimes it is convenient to create a structure that contains the head pointer of a list and also information about the list itself (e.g., number of elements currently in the list, maximum number of elements to be allowed in the list, etc).
- This extra data is referred to as ***metadata***.



Build the new node, and put the reference to the new data item in it



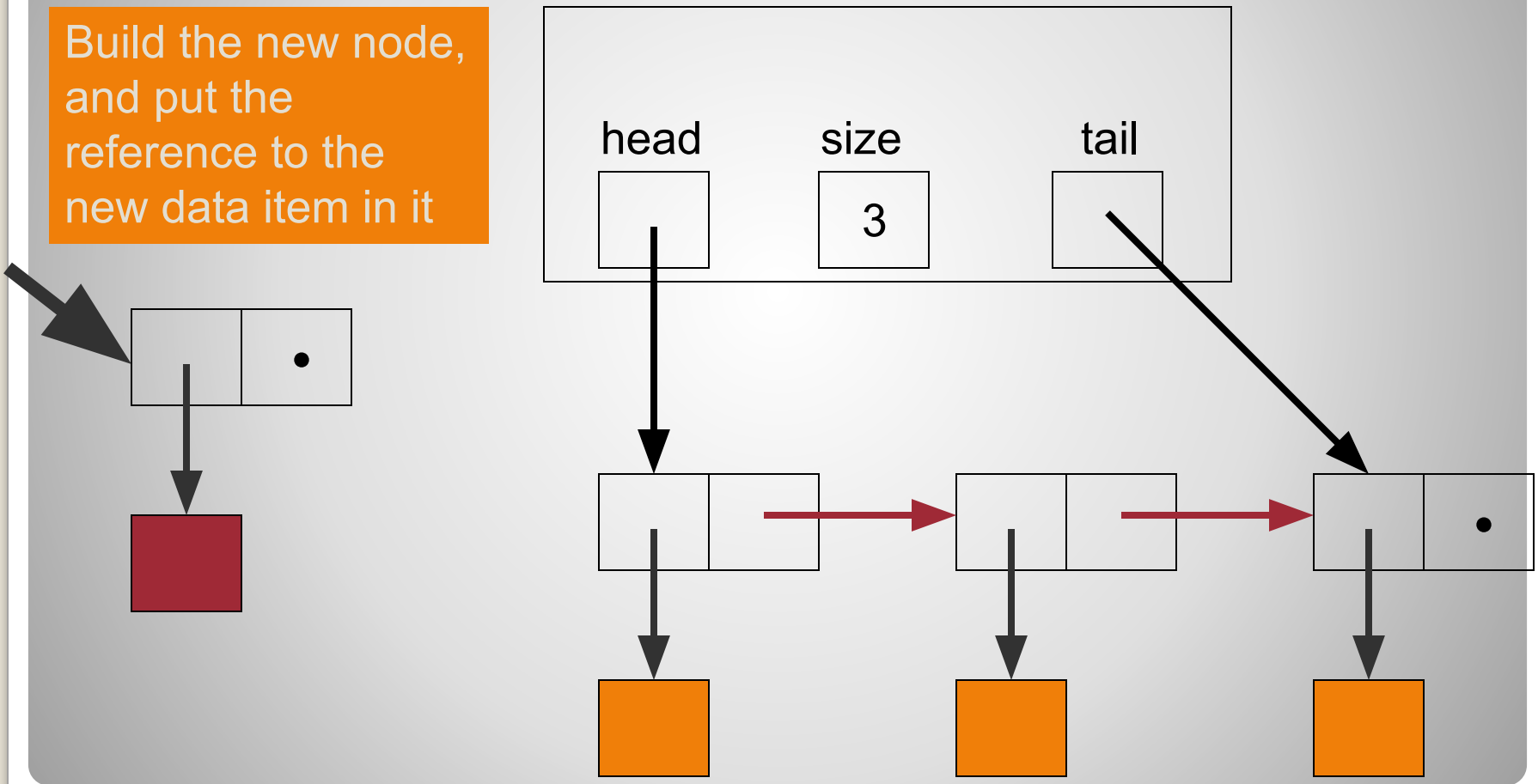
Add Item to an Empty Linked List



Make ***both*** head and tail point at the new node, and increment the list's size

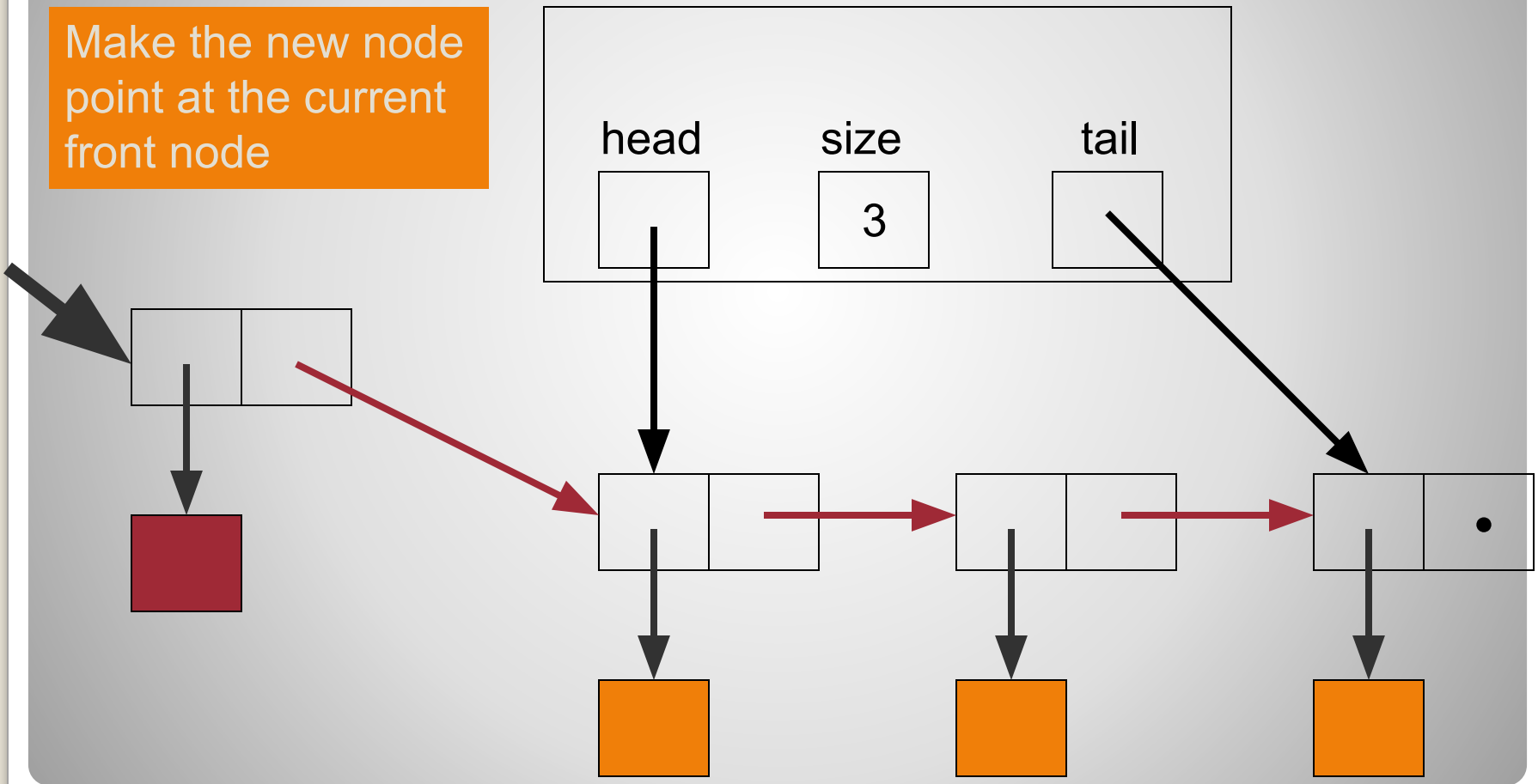
Insert Item at the Front

Build the new node, and put the reference to the new data item in it



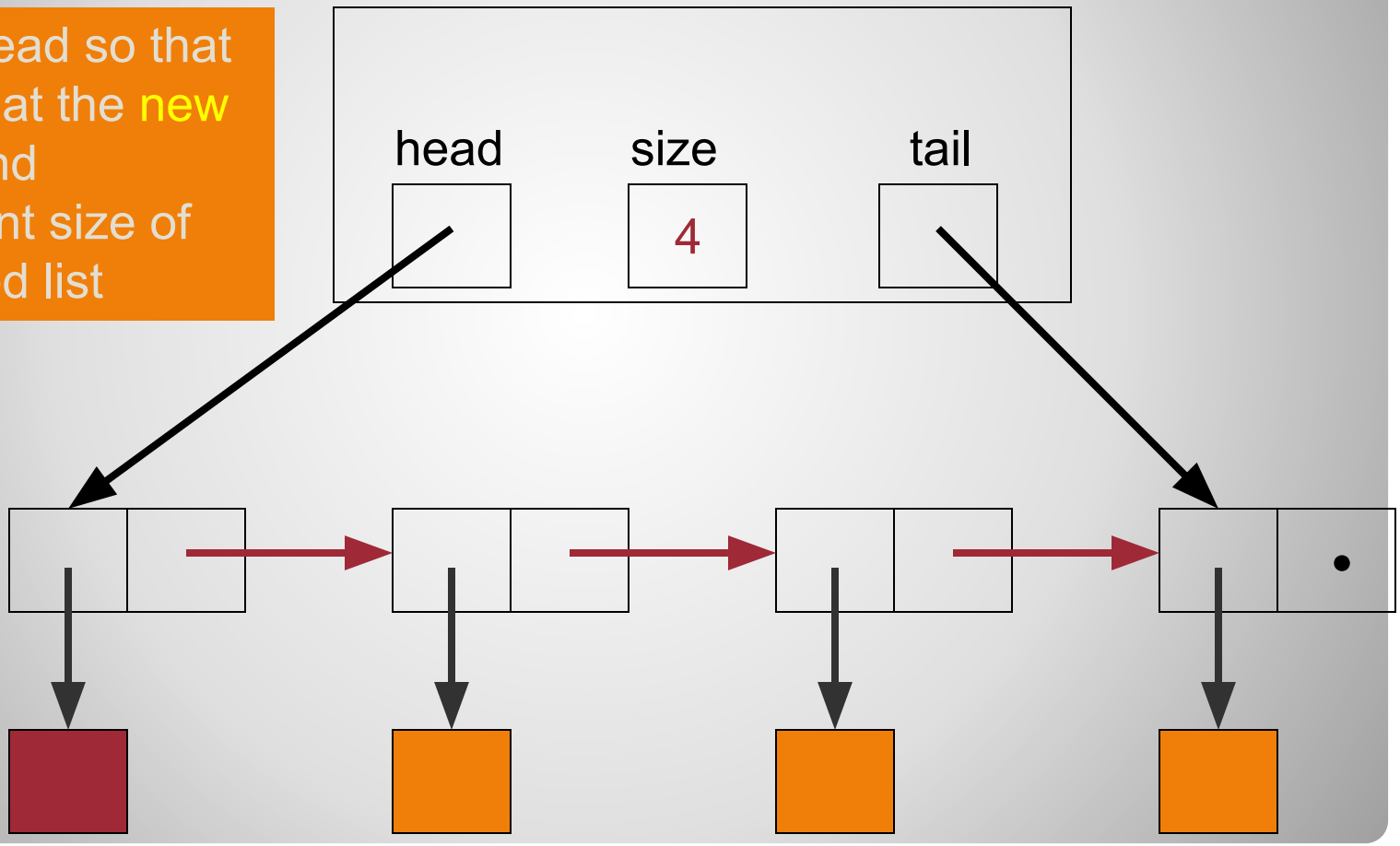
Insert Item at the Front

Make the new node point at the current front node

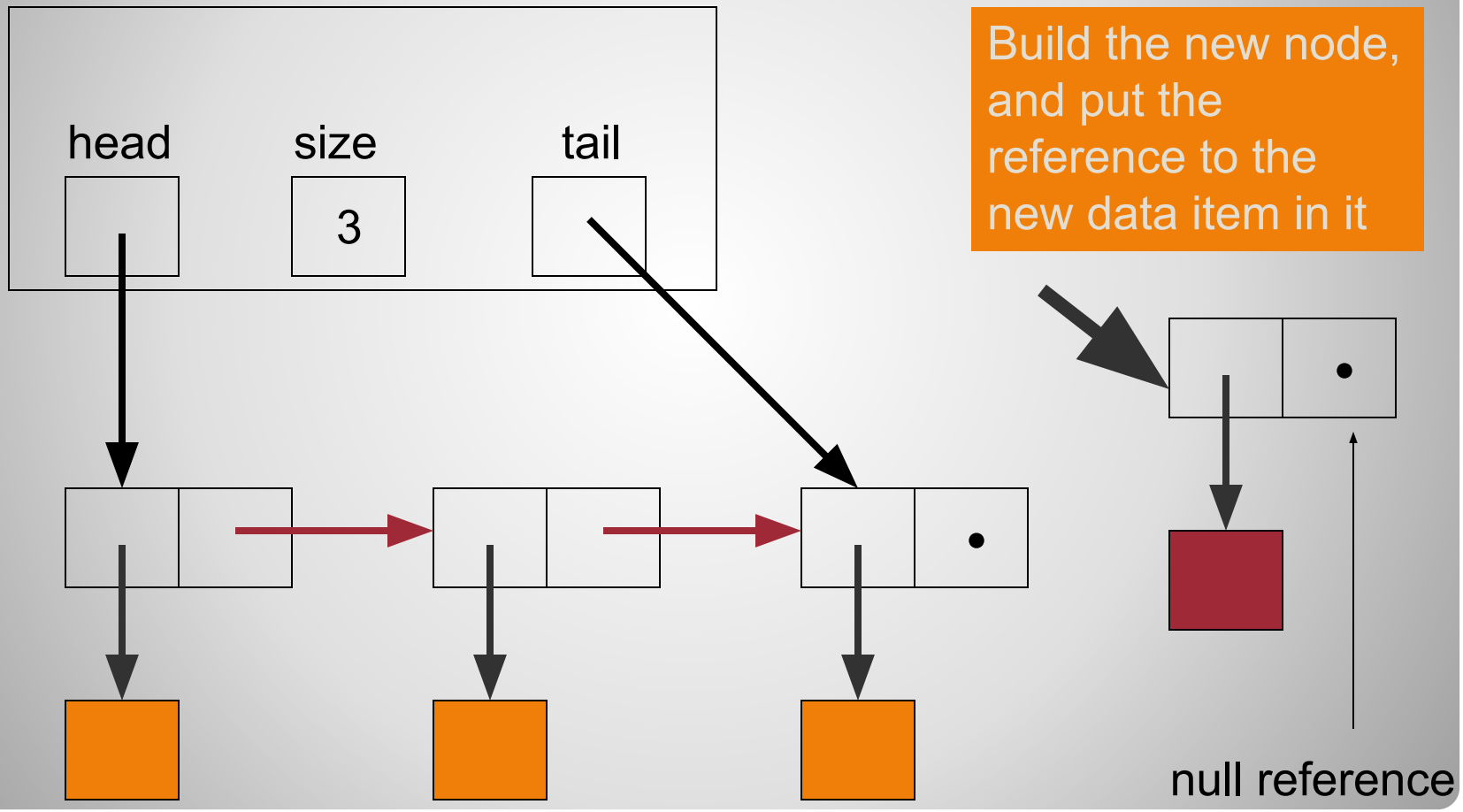


Insert Item at the Front

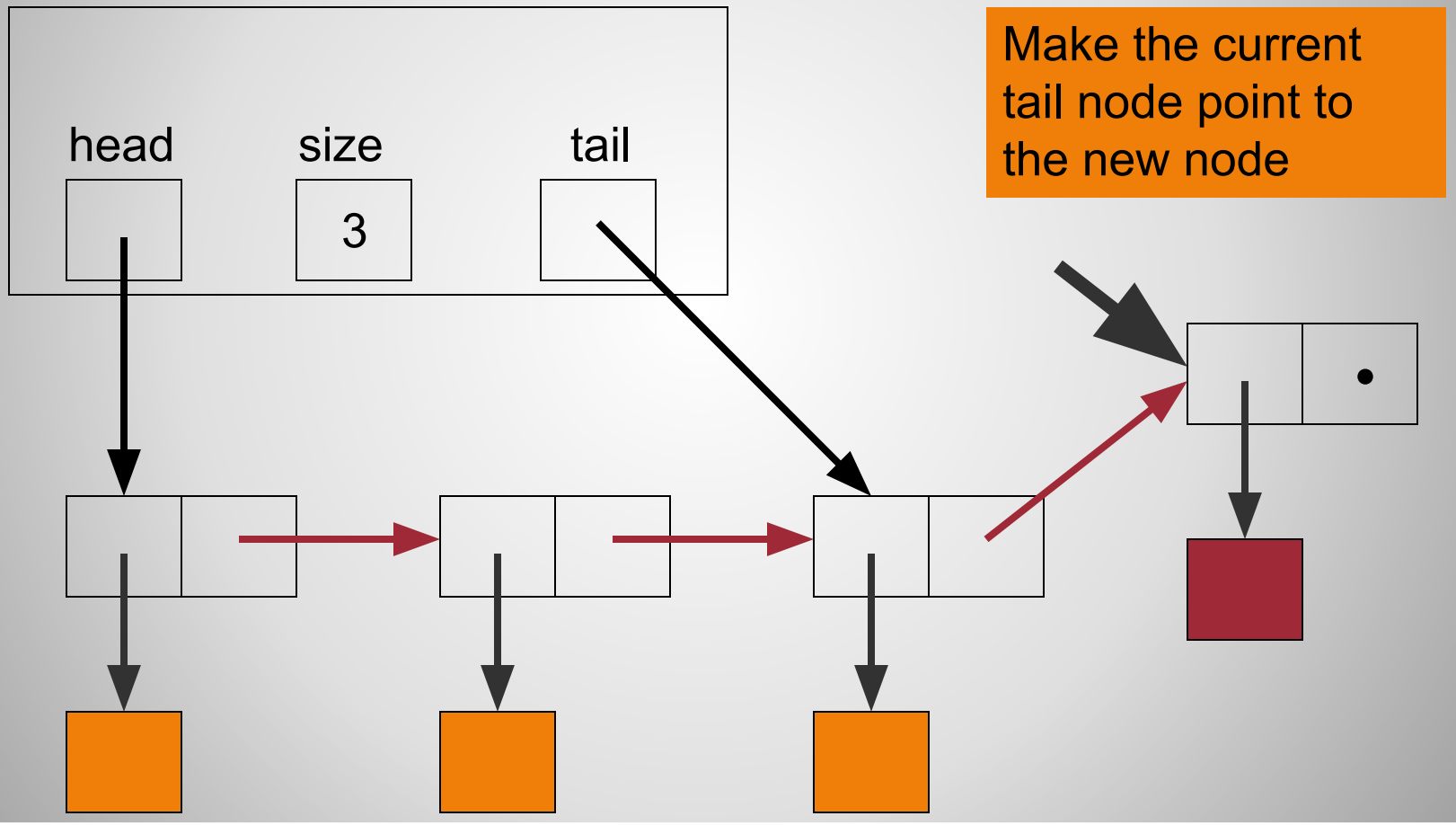
Reset head so that it points at the new node, and increment size of the linked list



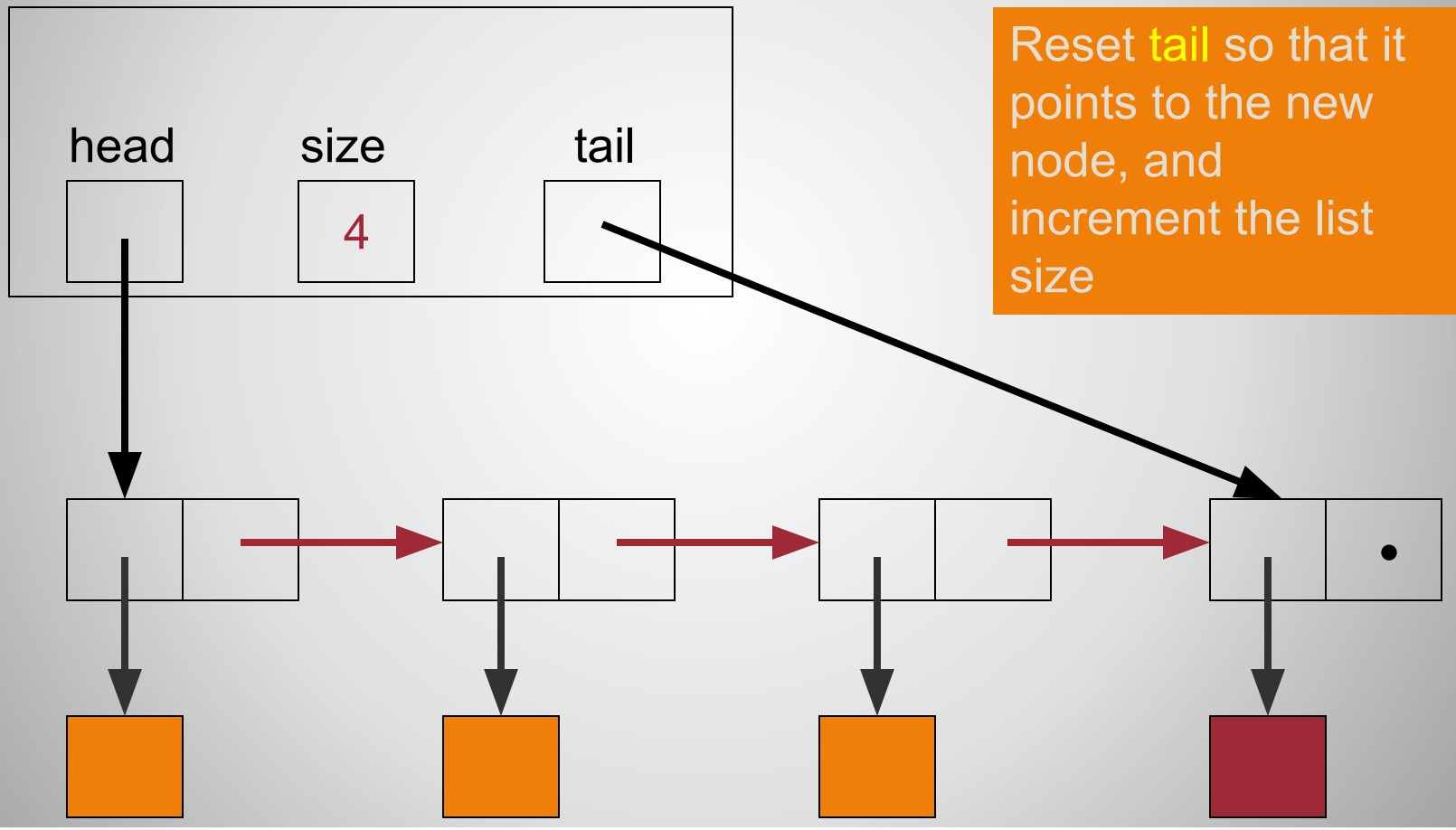
Insert at Rear



Insert at Rear

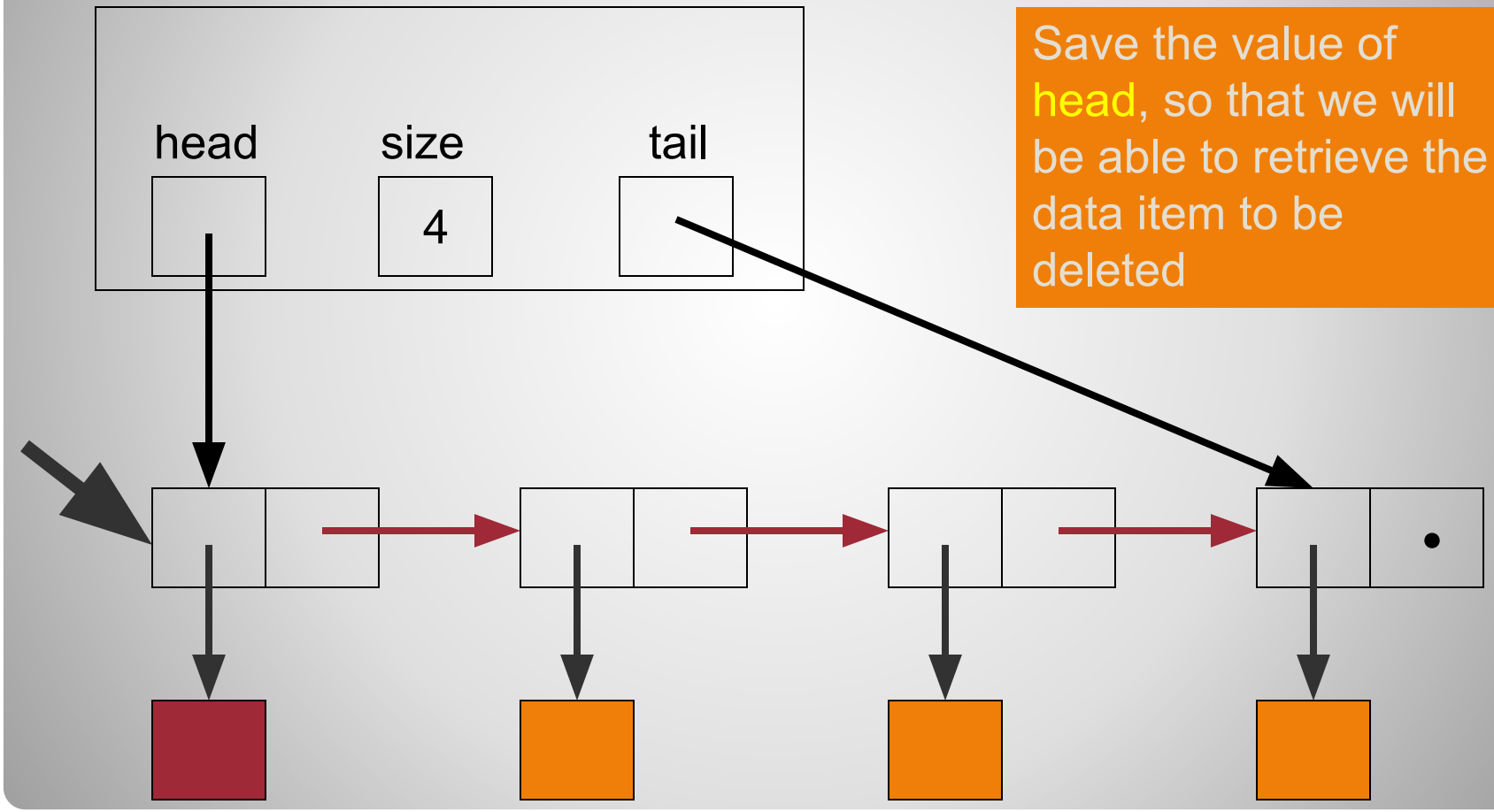


Insert at Rear



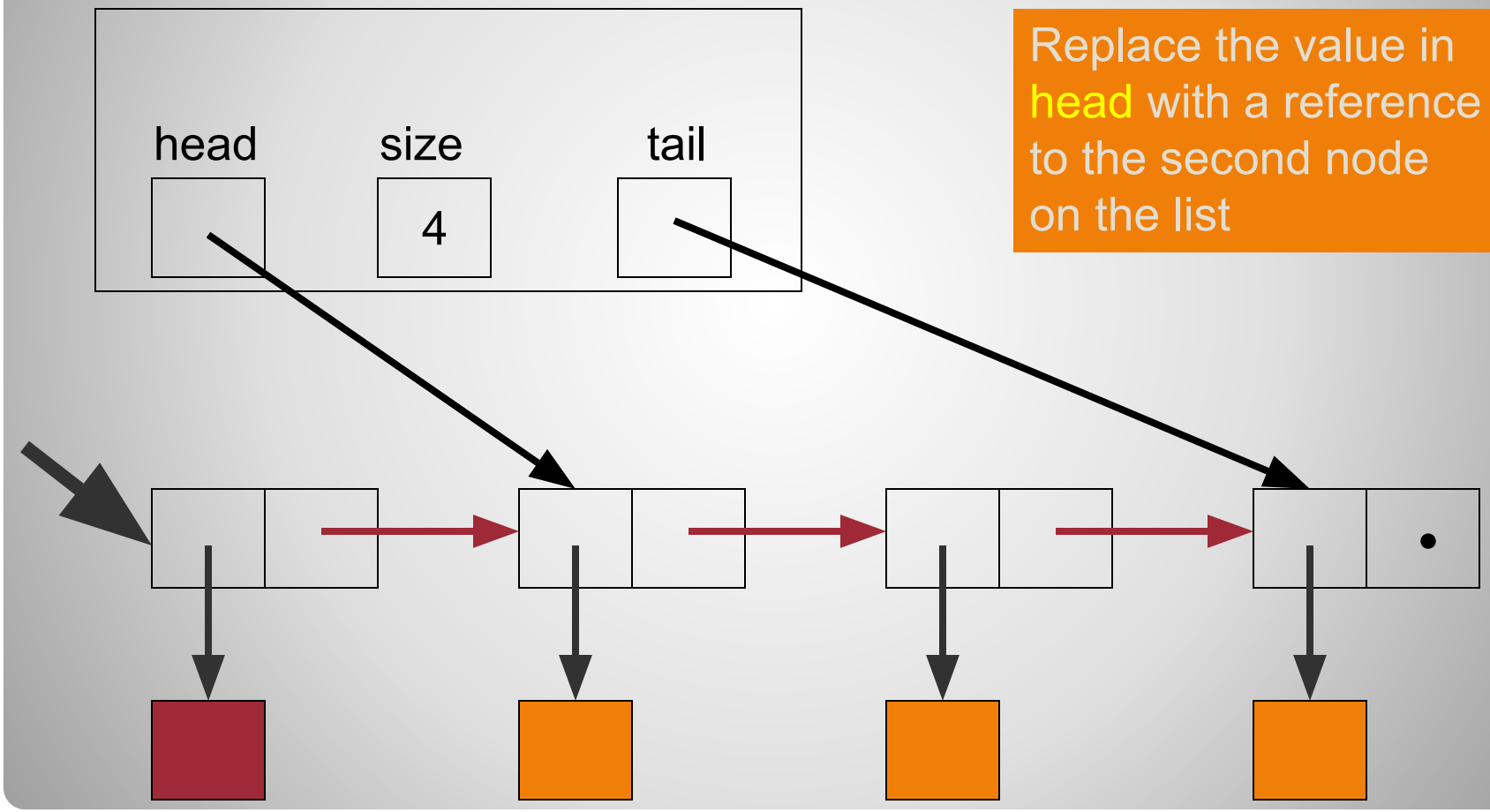
Reset tail so that it points to the new node, and increment the list size

Remove Front Item From a Linked List



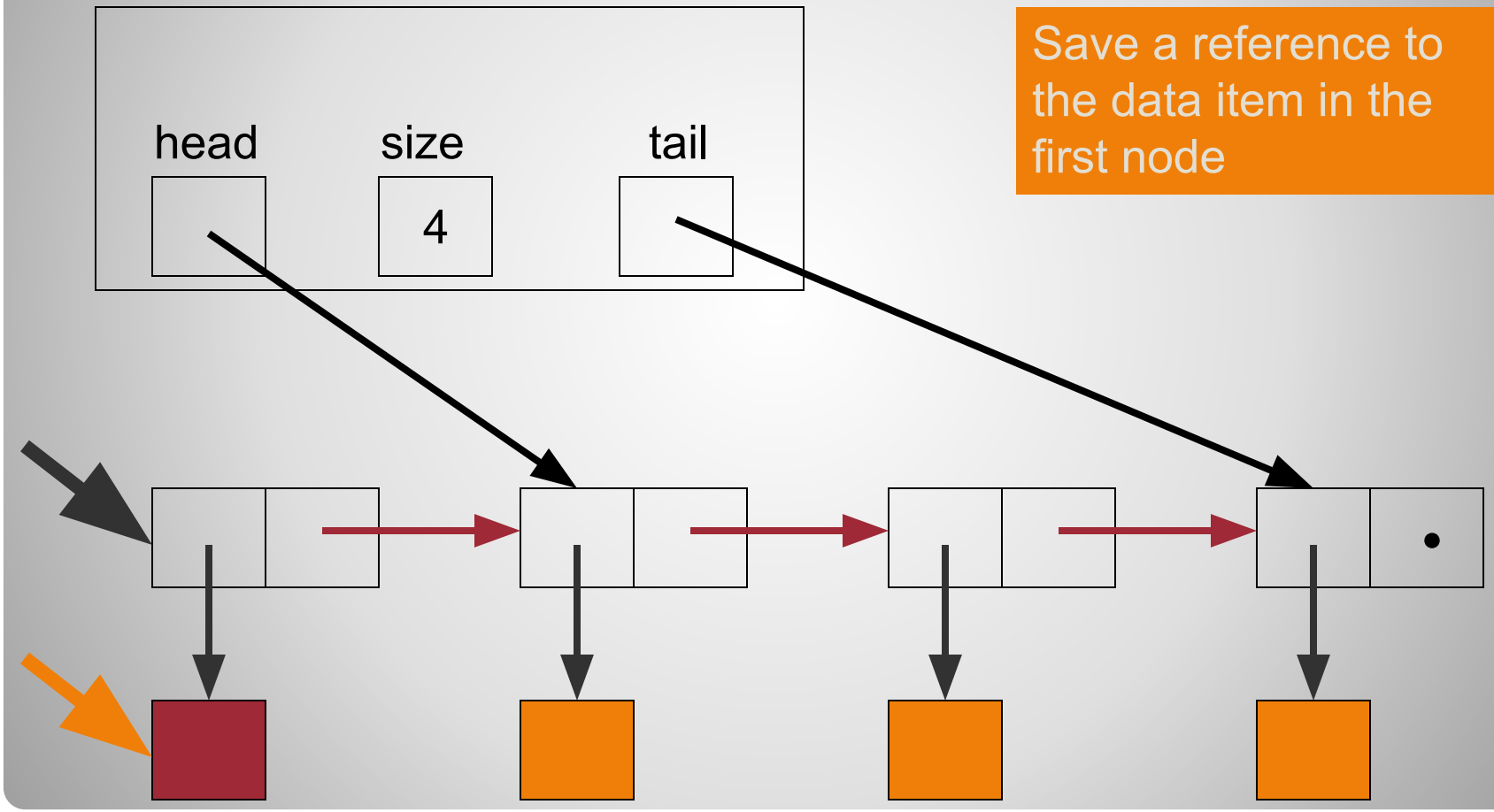
Save the value of head, so that we will be able to retrieve the data item to be deleted

To Remove Front Item From a Linked List

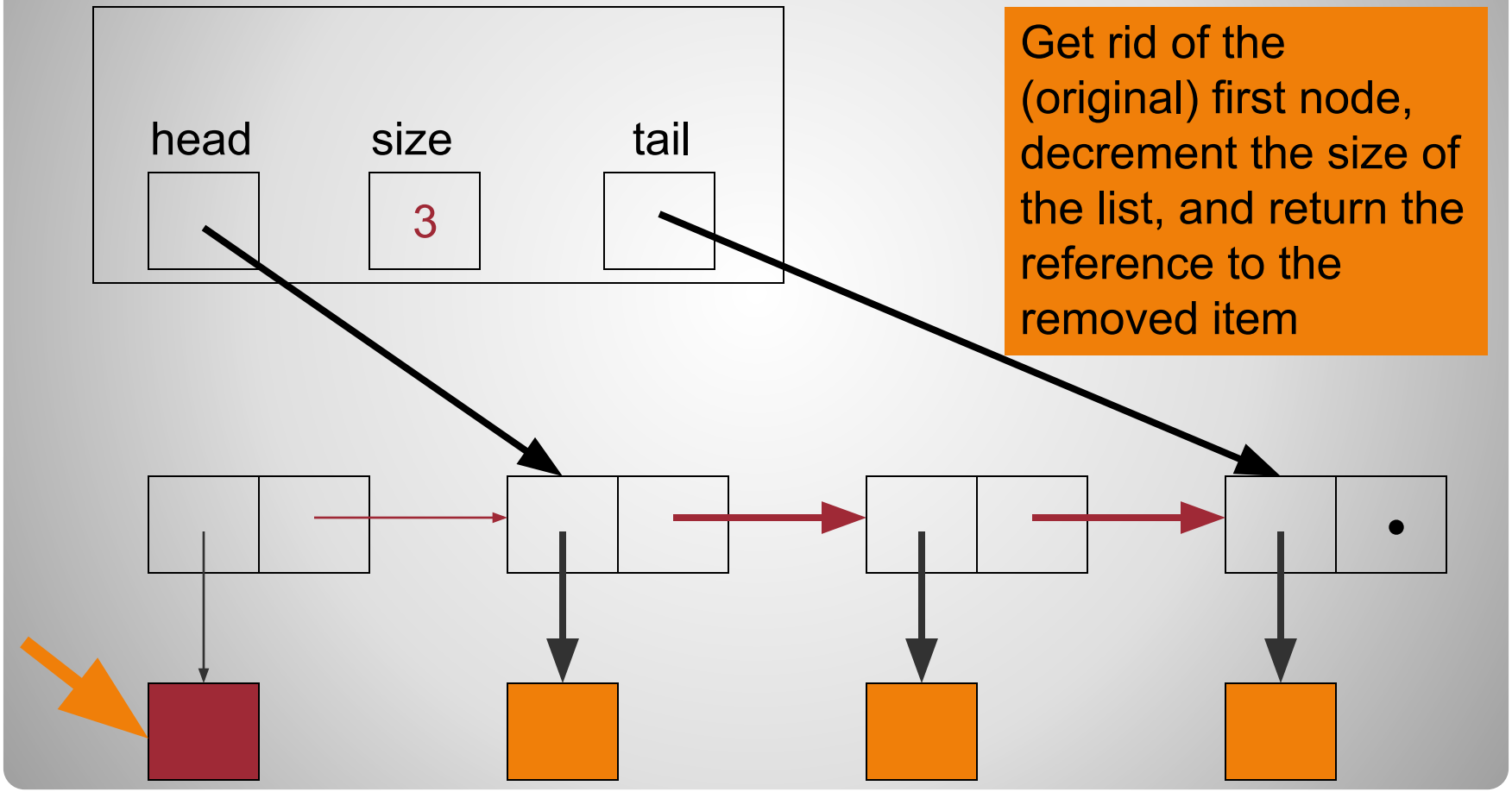


Replace the value in head with a reference to the second node on the list

To Remove Front Item From a Linked List

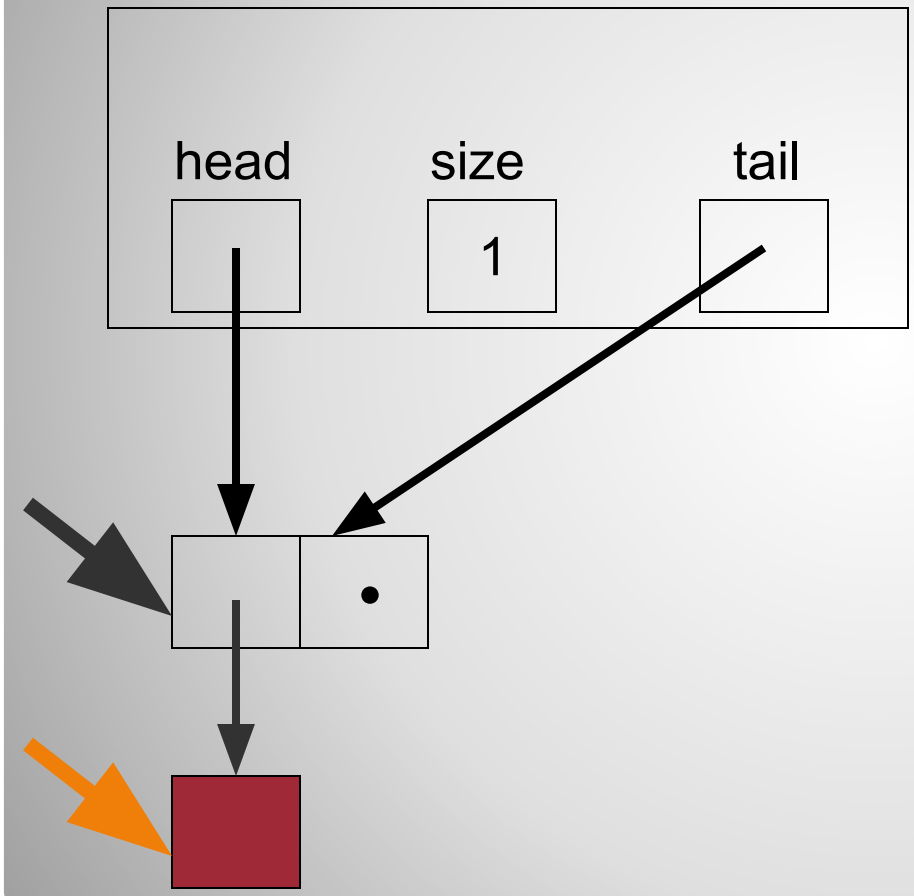


To Remove Front Item From a Linked List



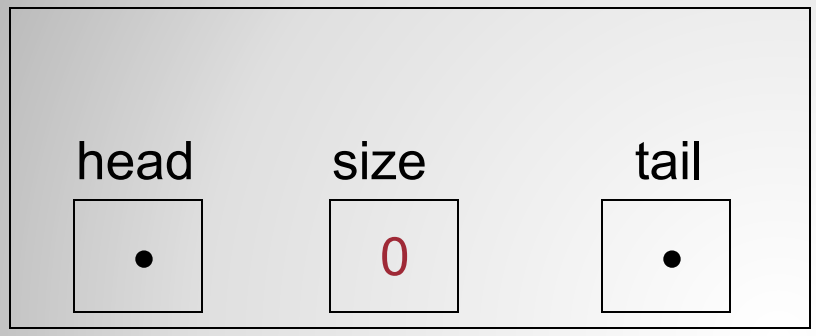
Get rid of the (original) first node, decrement the size of the list, and return the reference to the removed item

To Remove the Only Item From a Linked List

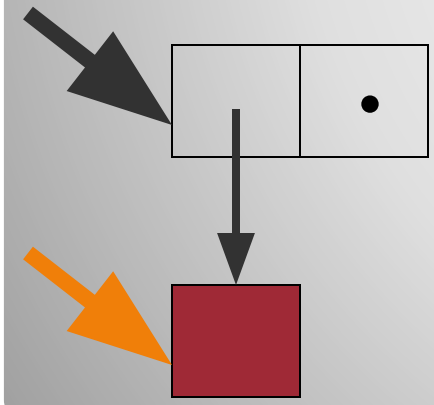


Save a reference to the node, and a reference to the data item found in the first node

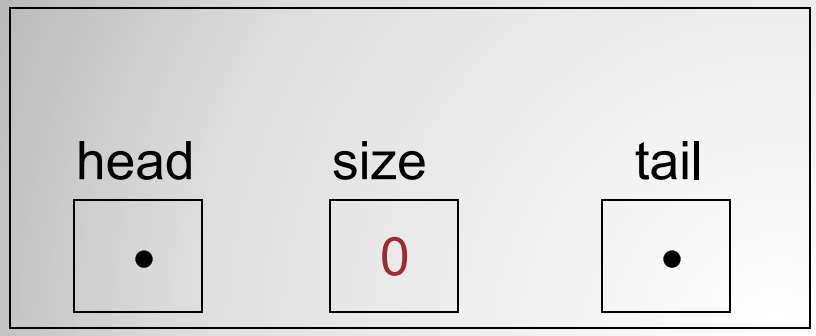
To Remove the Only Item From a Linked List



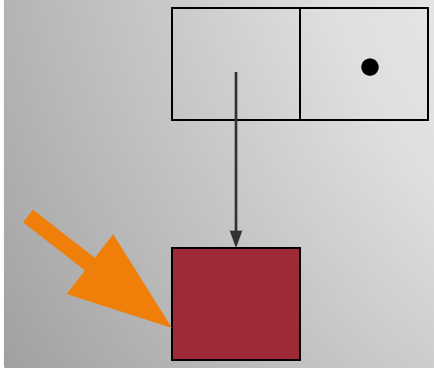
Set head and tail to *NULL*, and decrement the size of the list



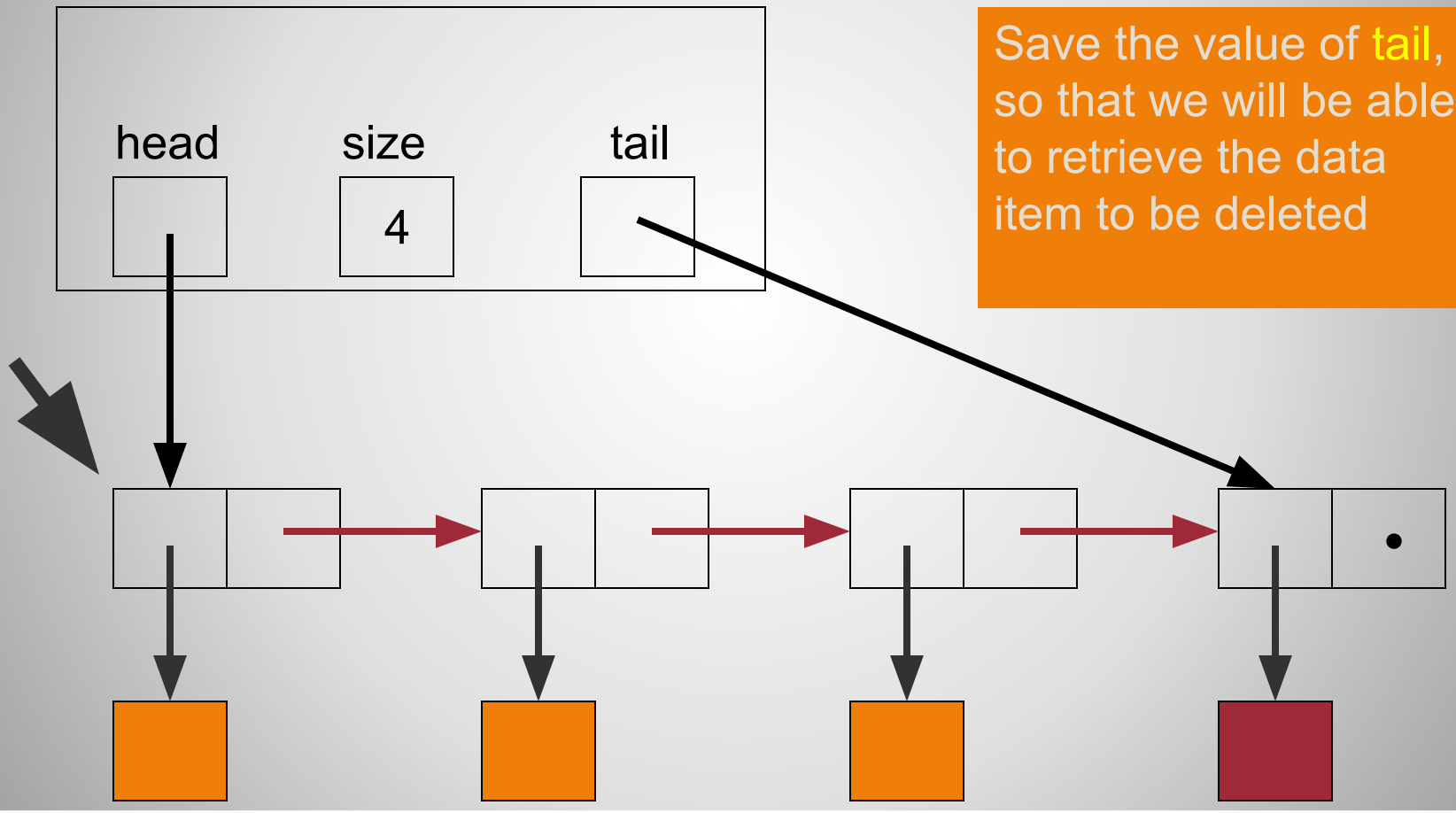
To Remove the Only Item From a Linked List



Get rid of the node, and return the reference to the data item

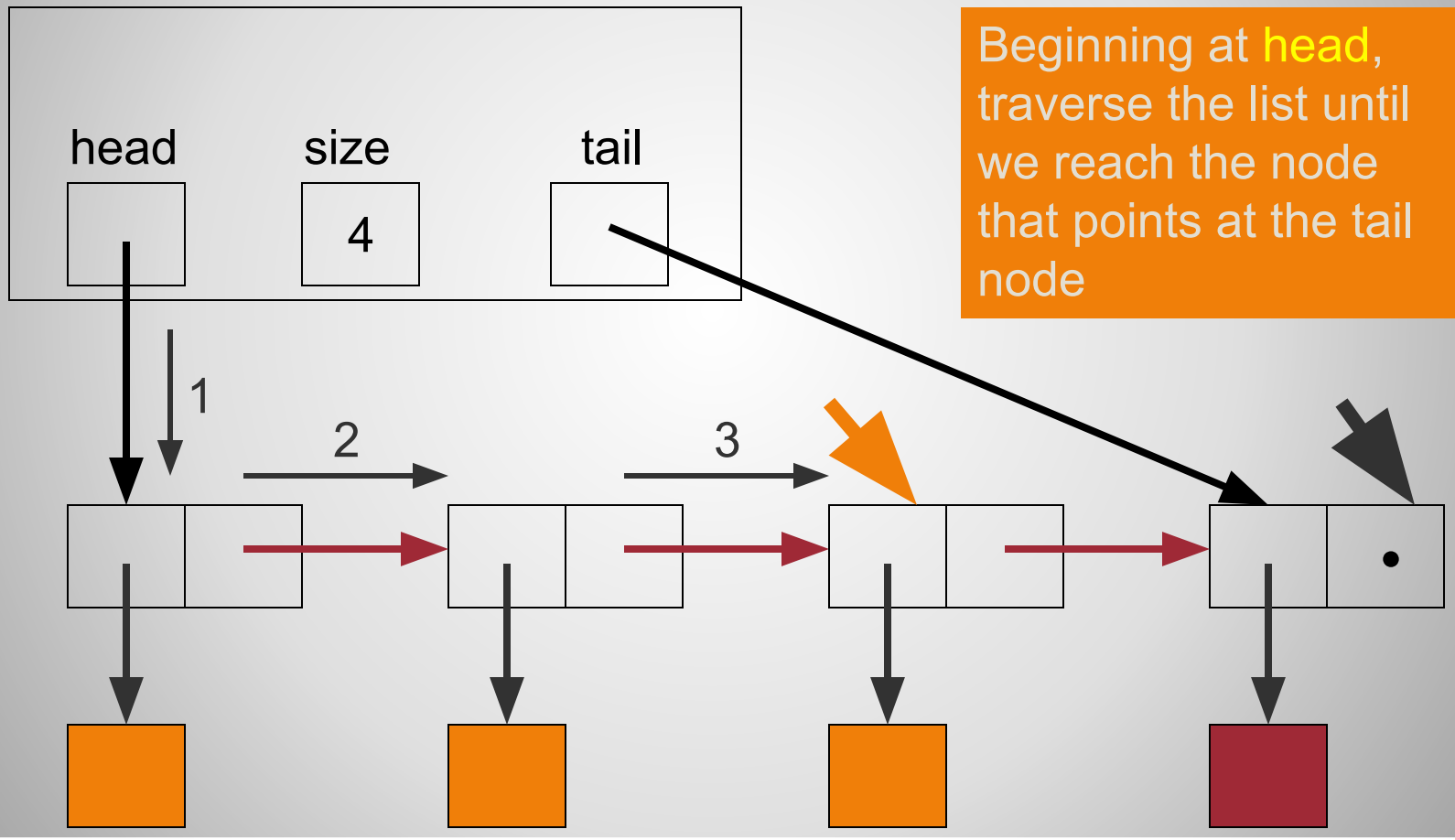


Remove Last Item From a Linked List



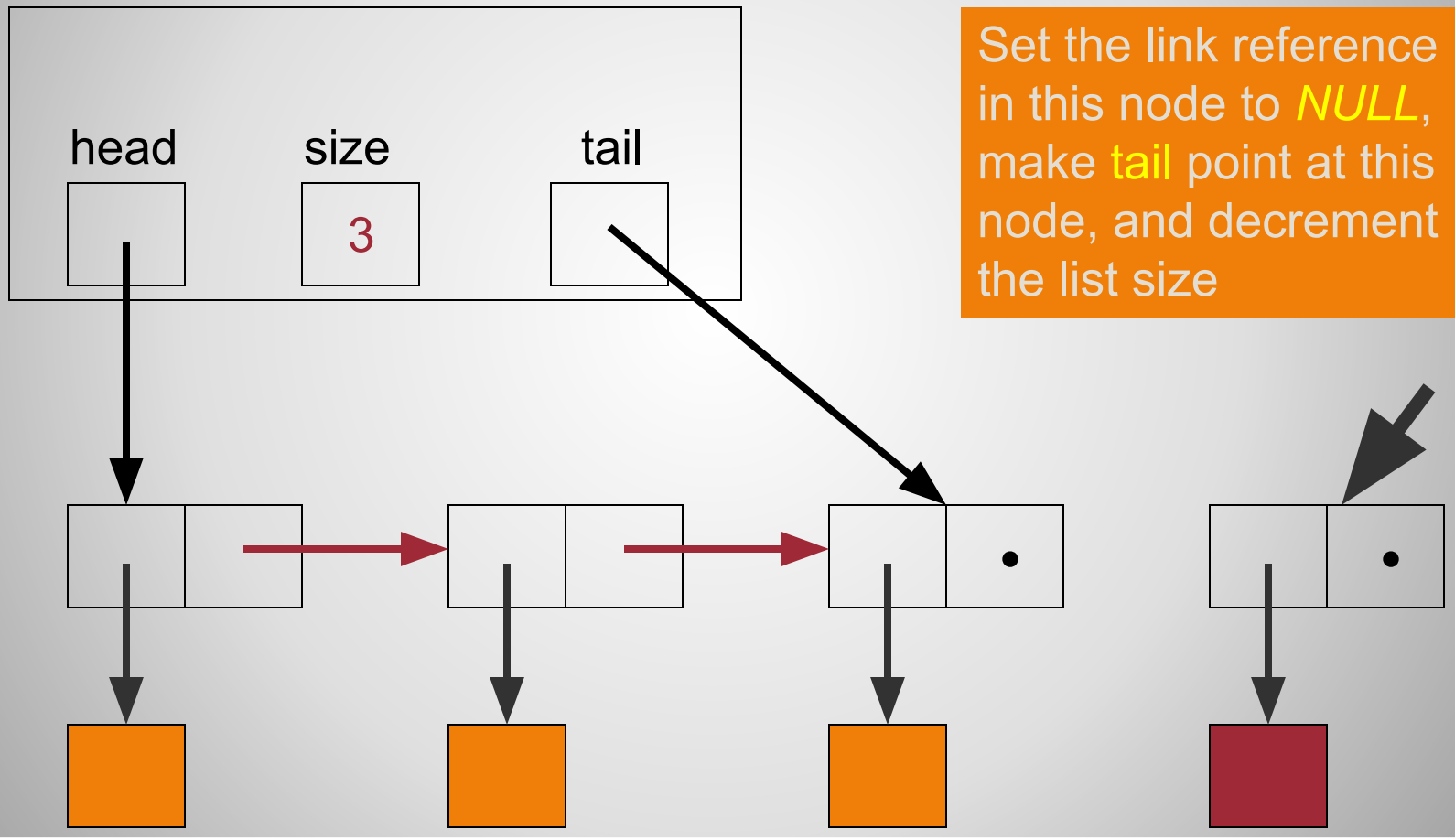
Save the value of tail, so that we will be able to retrieve the data item to be deleted

Remove Last Item From a Linked List



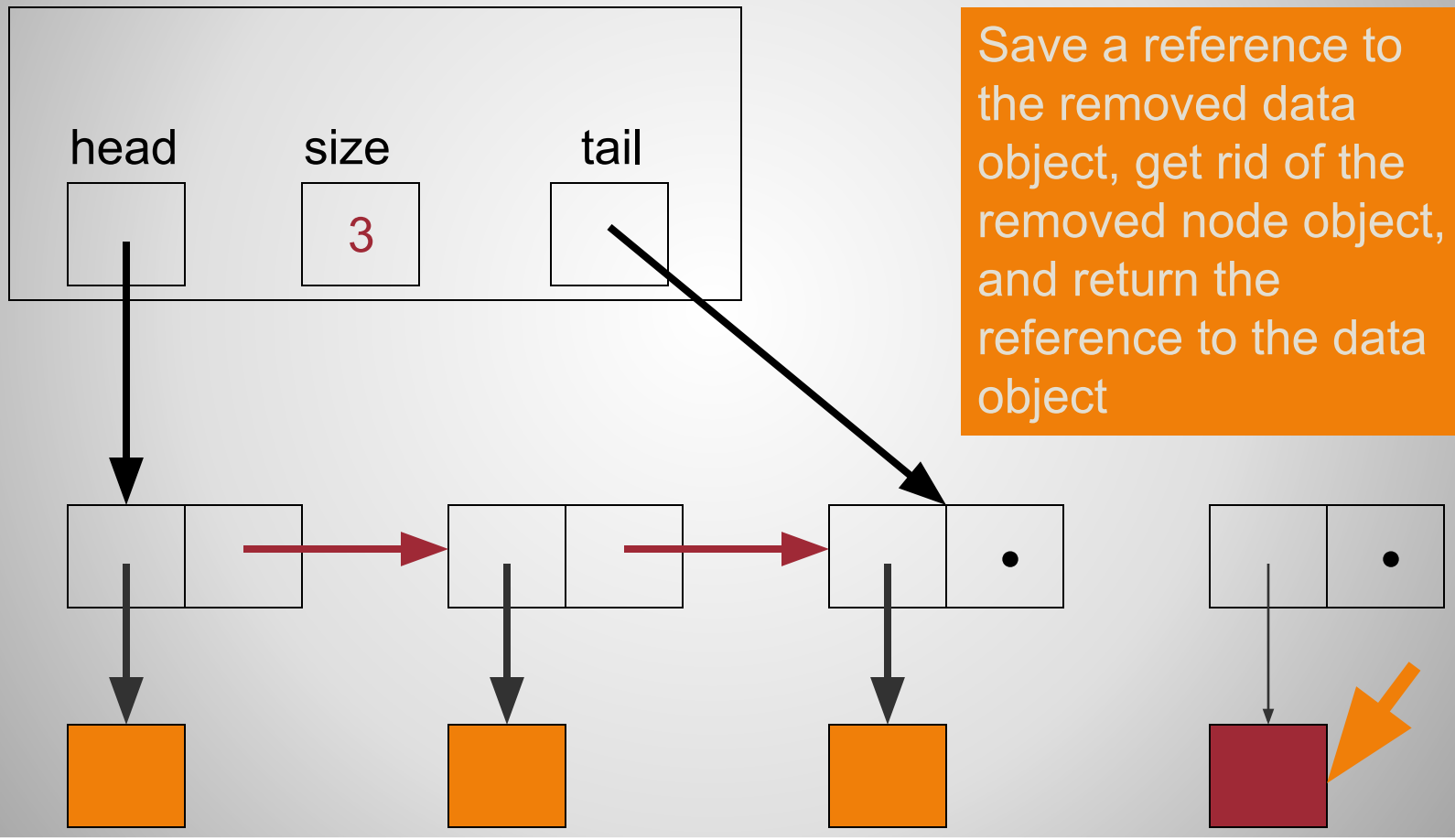
Beginning at head, traverse the list until we reach the node that points at the tail node

Remove Last Item From a Linked List



Set the link reference in this node to *NULL*, make tail point at this node, and decrement the list size

To Remove Last Item From a Linked List



Save a reference to the removed data object, get rid of the removed node object, and return the reference to the data object

Traverse a linked list

- Algorithms that traverse a list start at the first node and examine each node in succession until the last node has been processed.
- Traversal logic is used by several different types of algorithms, such as
  - changing a value in each node,
  - printing the list,
  - summing a field in the list,
  - or calculating the average of a field. Etc etc etc .. (so many requirements)
- Any application that requires that the entire list be processed uses a traversal.

Traversal of linked list

- To traverse the list we need a walking (navigator) pointer.
- This pointer is used to move from node to node as each element is processed.
- Example is our print function….

Search a linked list

- A search list is used by several algorithms to locate data in a list:
- To insert data, we need to know the logical predecessor to the new data.
- To delete data, we need to find the node to be deleted and identify its logical predecessor.
- To retrieve data from a list, we need to search the list and find the data.
- Worst case is that the data to be retrieved is at the last of the list.
- E.g Find the employee who's name is "Salman", if salman is present then the record (node) having salman will be returned else return null node.

Search an Ordered list

at least it will be faster than un-ordered search

- First let us consider searching a list in which the elements are sorted based on a field value.
- Given a target key, the ordered list search attempts to locate the requested node in the linked list.
- If a node in the list matches the target, the search returns true, else we return false.
  - True can also be returning the specific node
- E.g Find the employee who's name is "Salman", It will be faster if the list was in alphabetical order