LINKED LISTS

OBJECTIVES

To learn what linked lists are

To insert and delete nodes in LL

What is a LINKED LIST?

A data structure that consists of dynamic variables linked together to form a chain-like structure.

Used as an

ALTERNATIVE

to arrays

During execution, the linked list can either…

GROW *or* **SHRINK**

Used when the

DATA SIZE VARIES

during execution

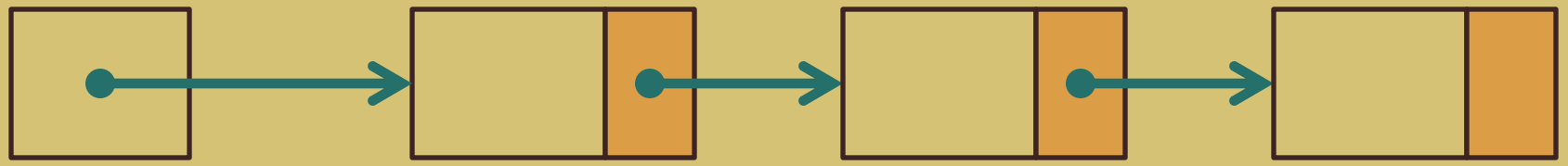
Unlike arrays, linked lists

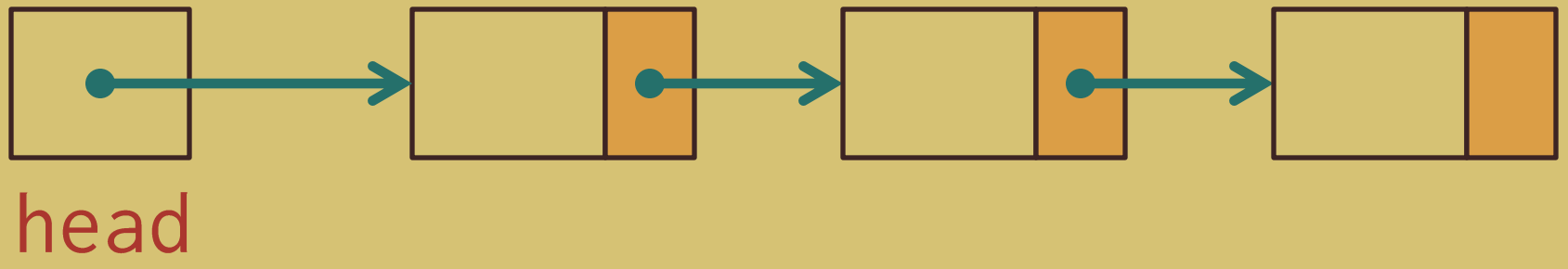
SAVE MEMORY

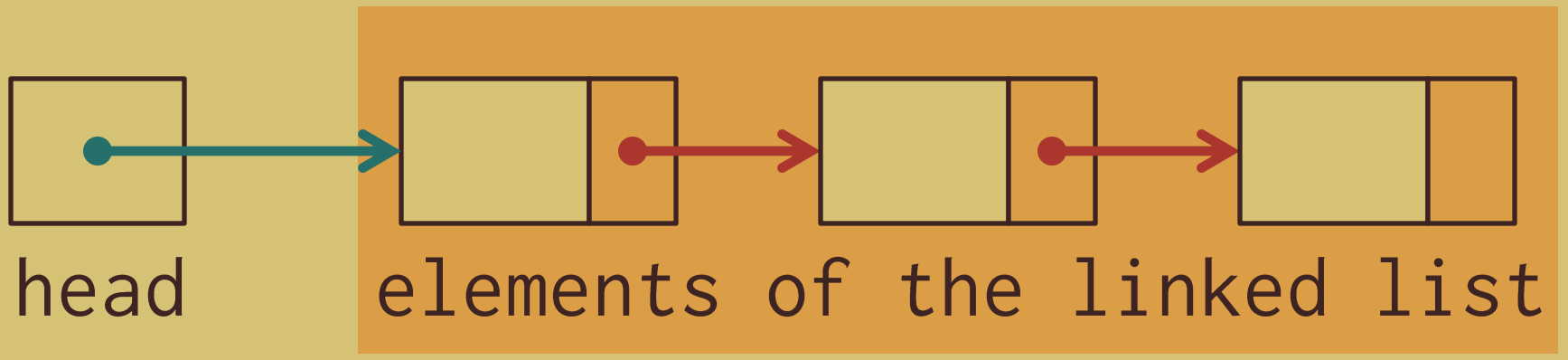
Allocated memory will never exceed what is needed by the program.

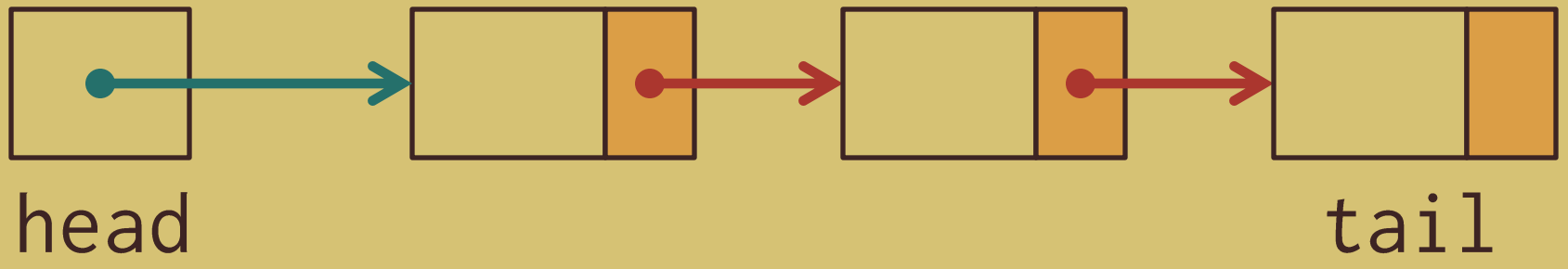
Dynamic arrays can handle the change in the maximum size of data, …

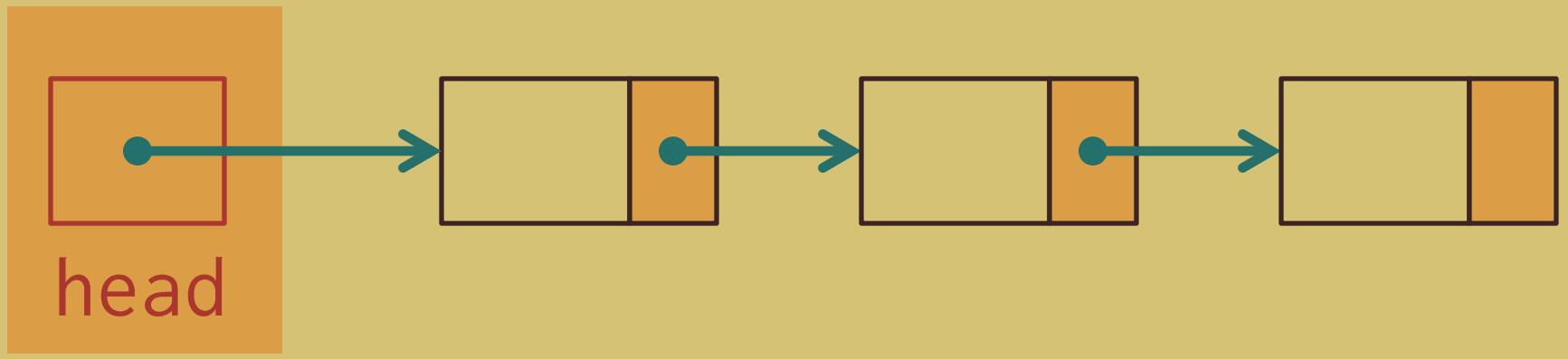
… but it is possible that the allocated memory will not be used.



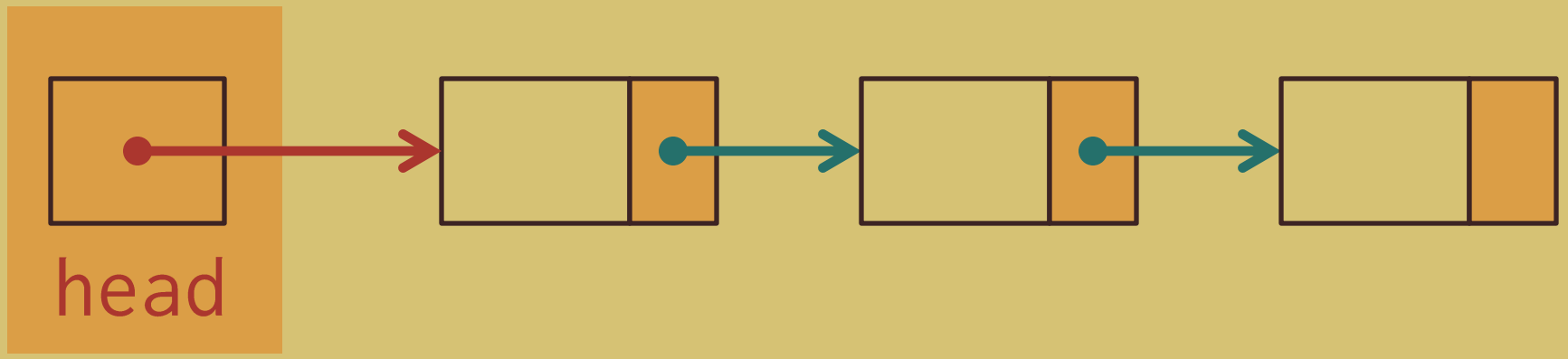

head


head
elements of the linked list

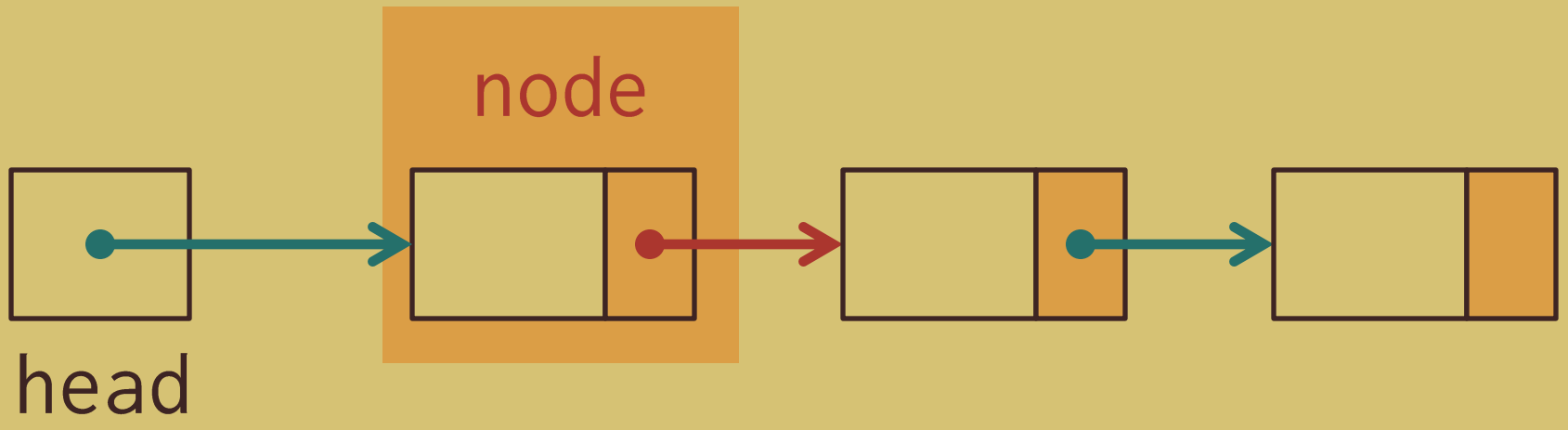

head
tail

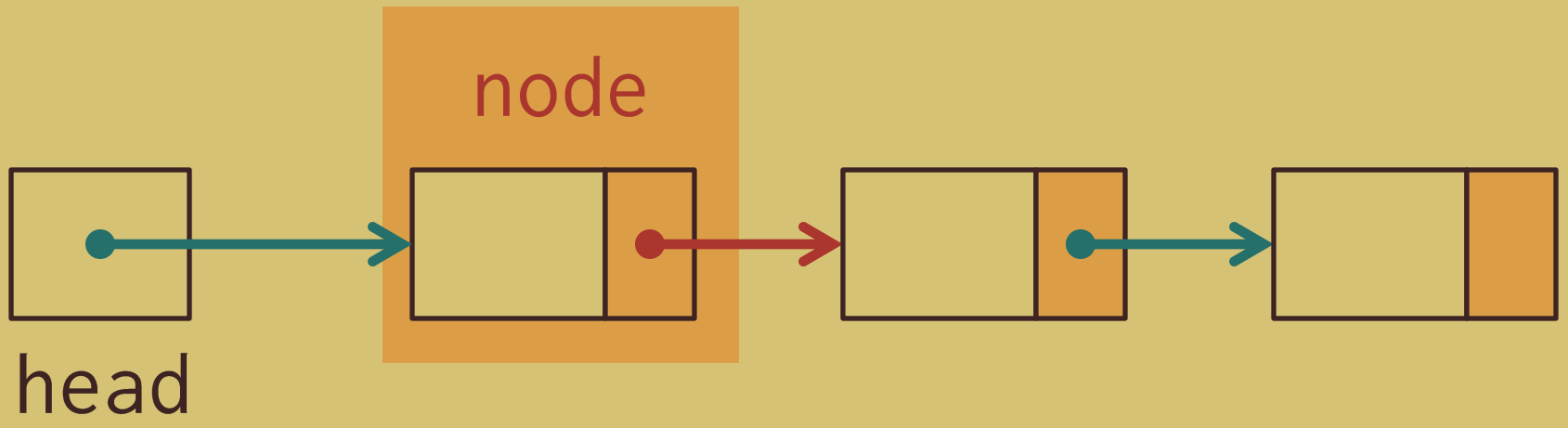


The head is a pointer (to a structure).

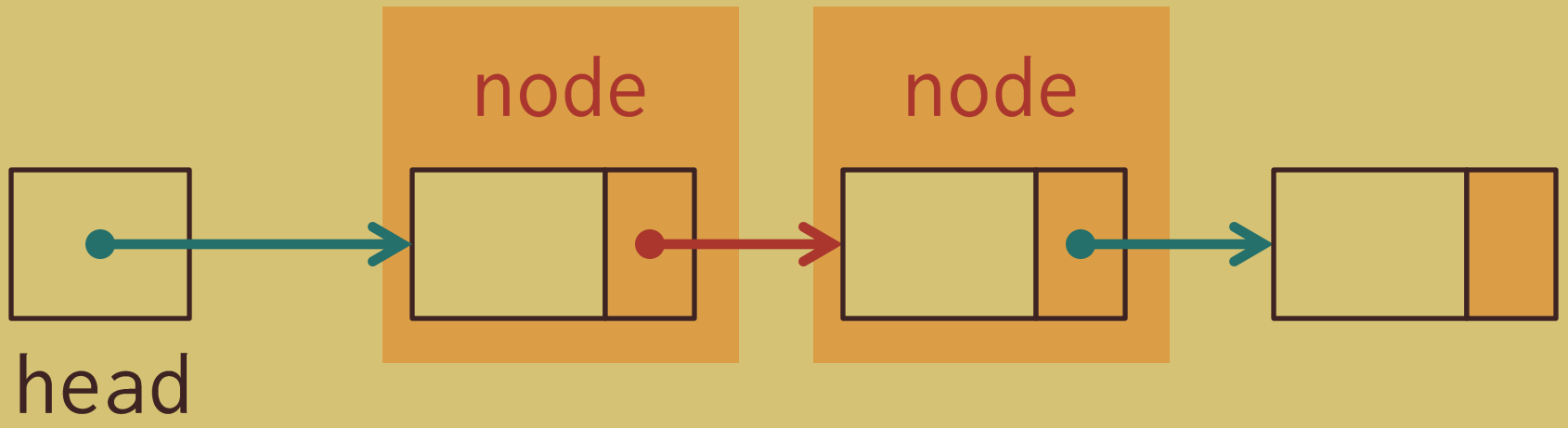


It holds the address of the first element of the linked list.


node
head

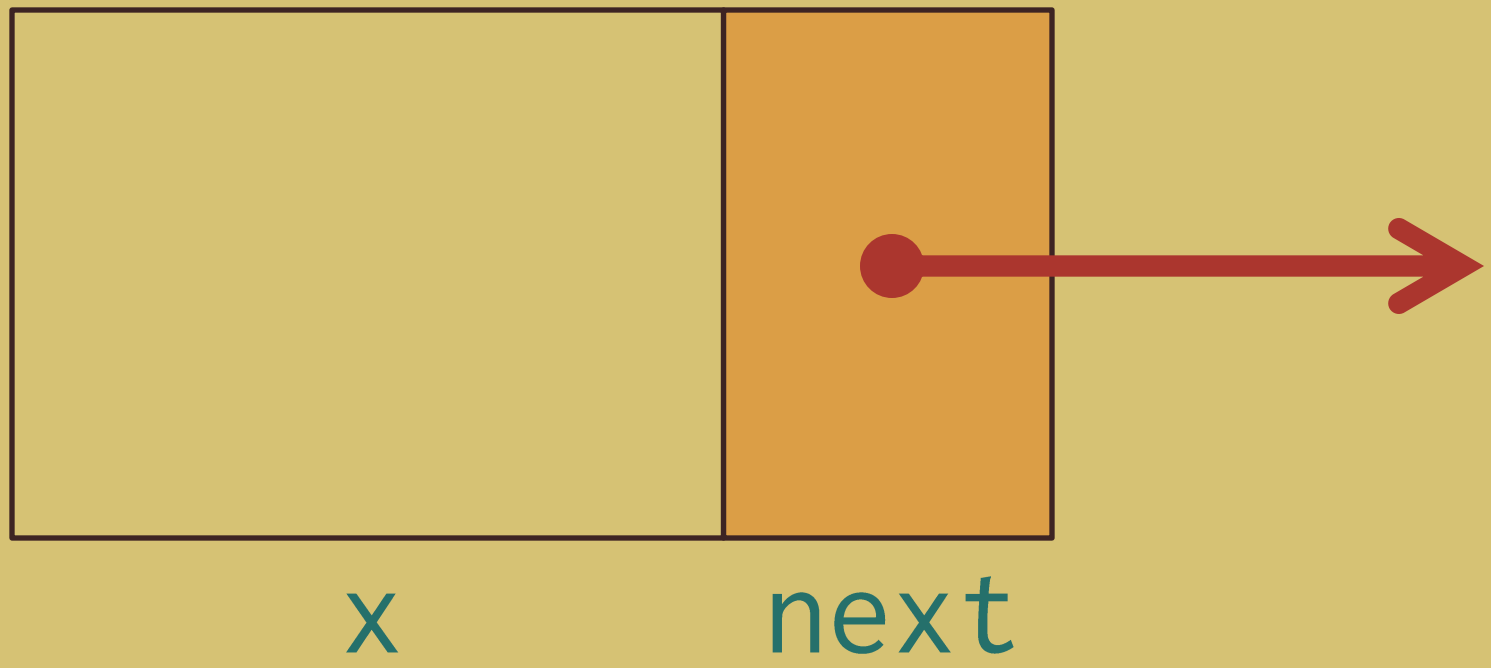


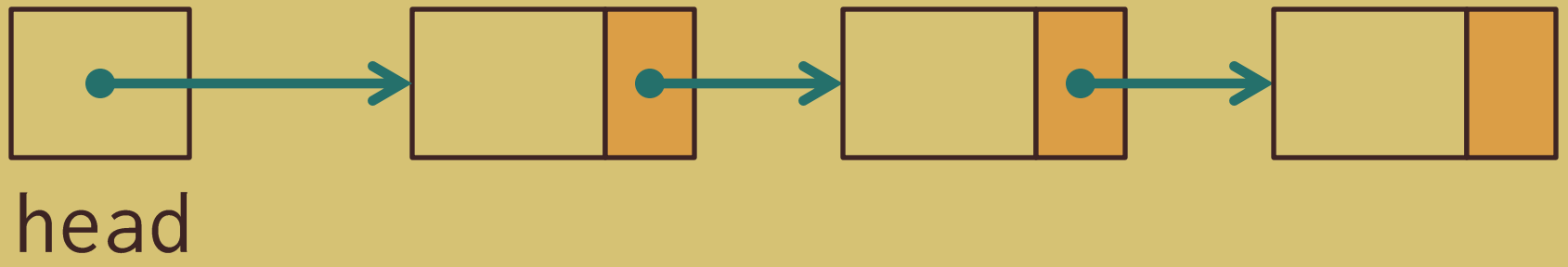
A node is a self-referential structure*.



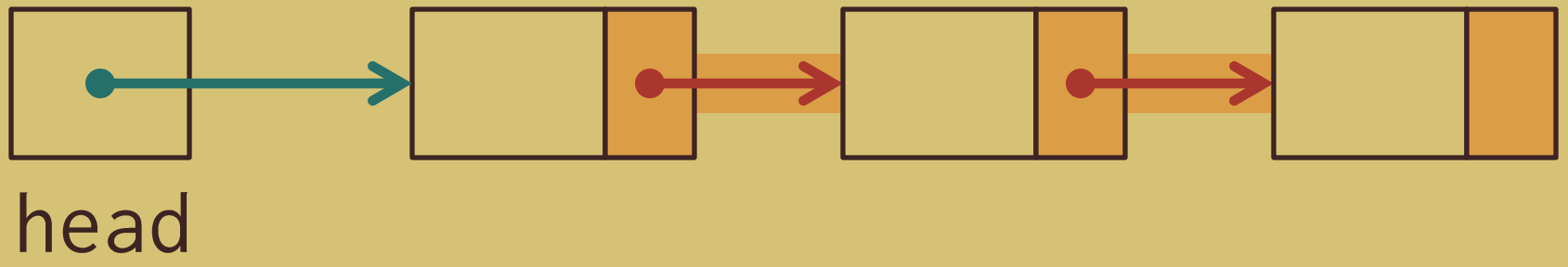
*A structure that has a pointer to an instance of itself as a field.

```
struct node {
   int x;
   //pointer to an instance of
   //struct node
   struct node *next;
}
```

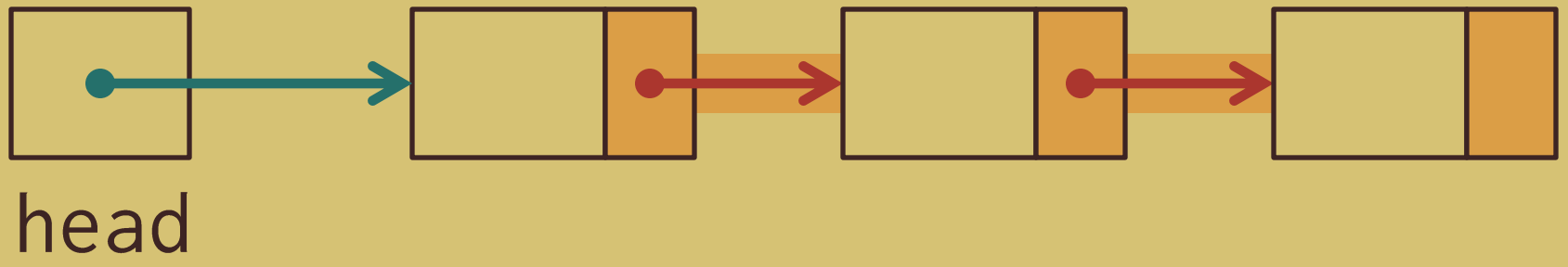

x
next



`malloc()` and `free()` are used to dynamically grow and shrink the linked list.



The pointer in each node will point to the next node in the list.



May have a dummy node.

Kinds of

LINKED LISTS:

SINGLY LINKED LISTS

SINGLY LINKED LISTS

DOUBLY LINKED LISTS

CIRCULAR SINGLY LINKED LISTS

CIRCULAR SINGLY LINKED LISTS

CIRCULAR DOUBLY LINKED LISTS

A dummy node is a node in the linked list that does not contain data.

It is used to simplify some linked lists operations.

The 4 basic

OPERATIONS ON LINKED LISTS:

INSERT

DELETE

SEARCH

VIEW

INSERT DELETE

INSERT

Deals with inserting values to the linked list.

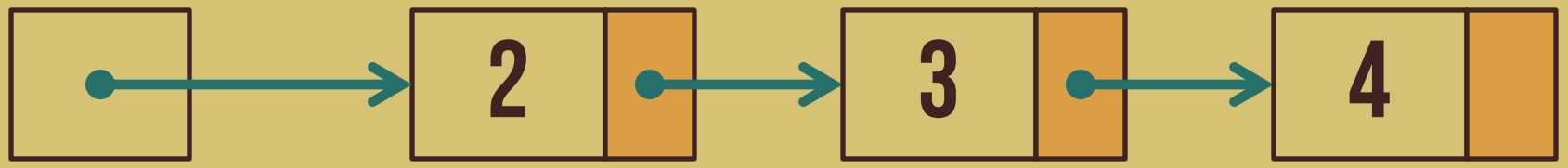
Different CASES of INSERT:

INSERT AT HEAD

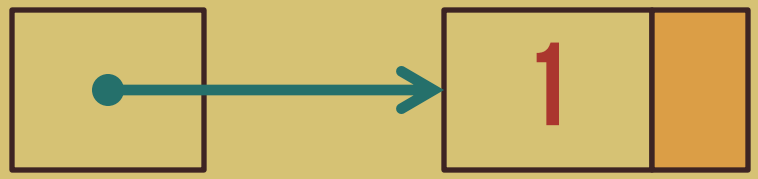
INSERT AT HEAD

You have a new node that you want to insert at the beginning.

INSERT AT HEAD

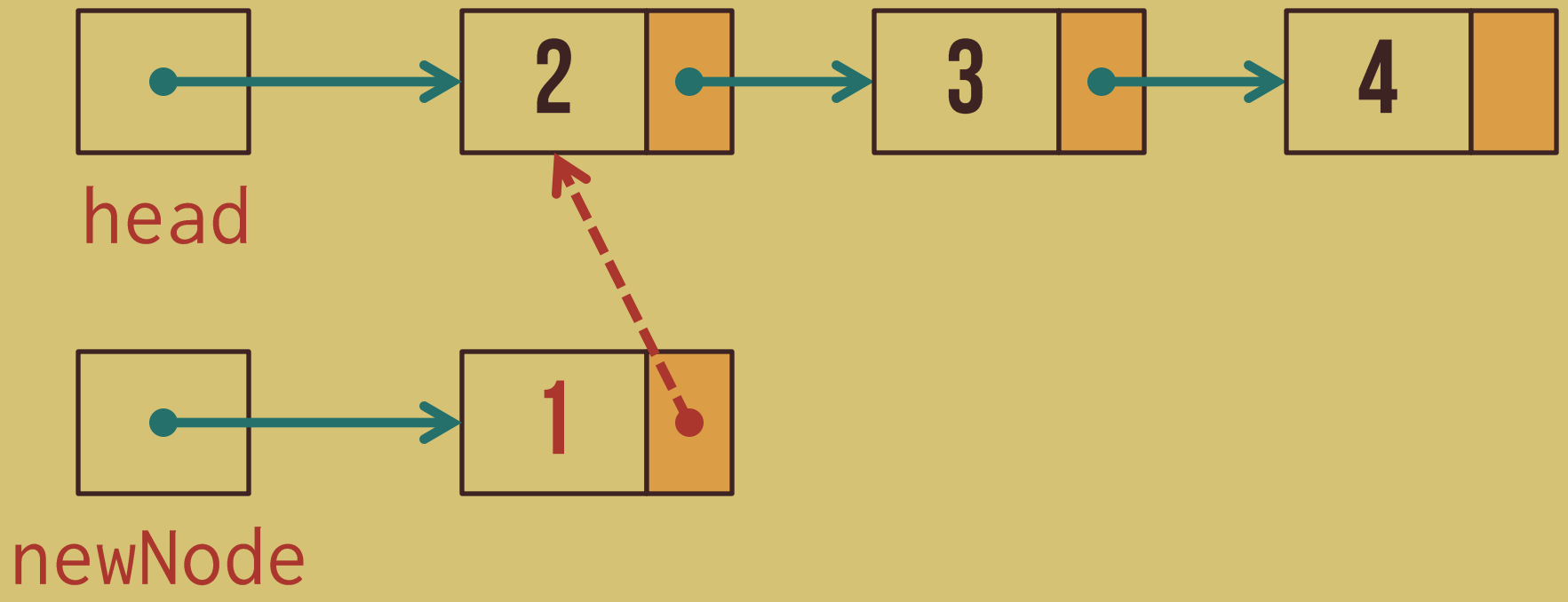


head



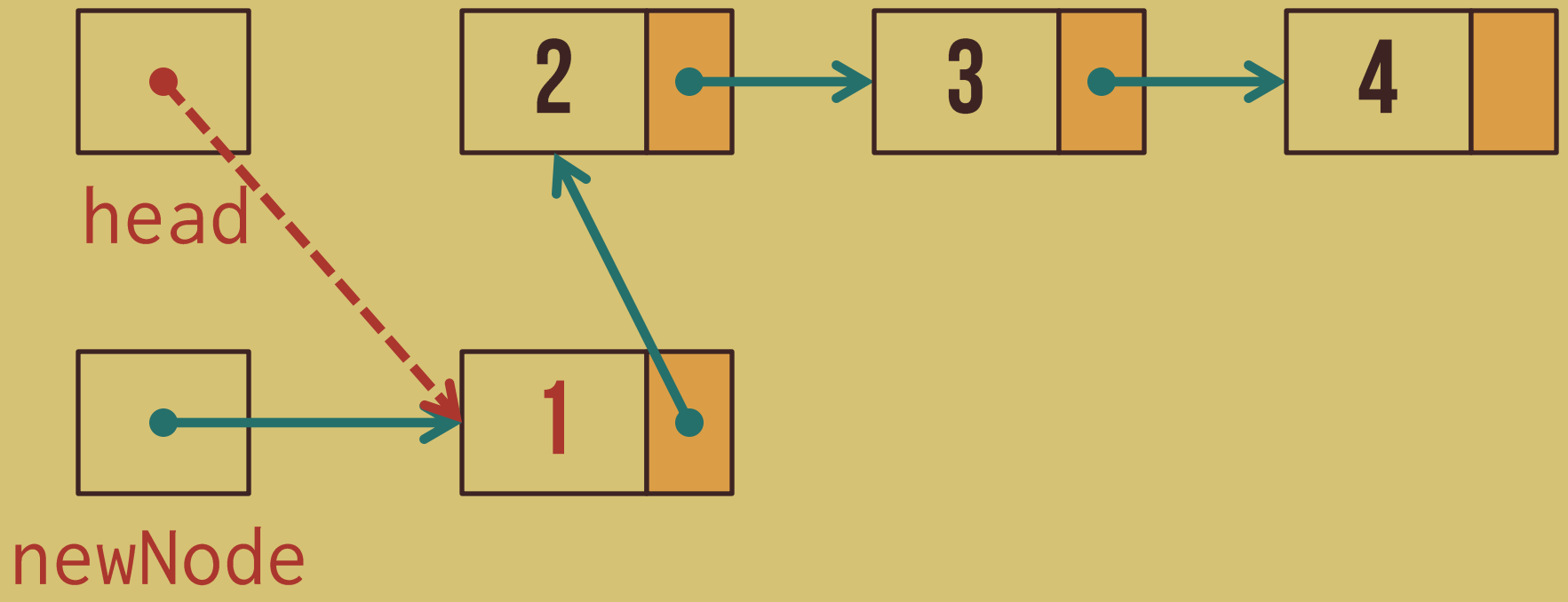
newNode

INSERT AT HEAD



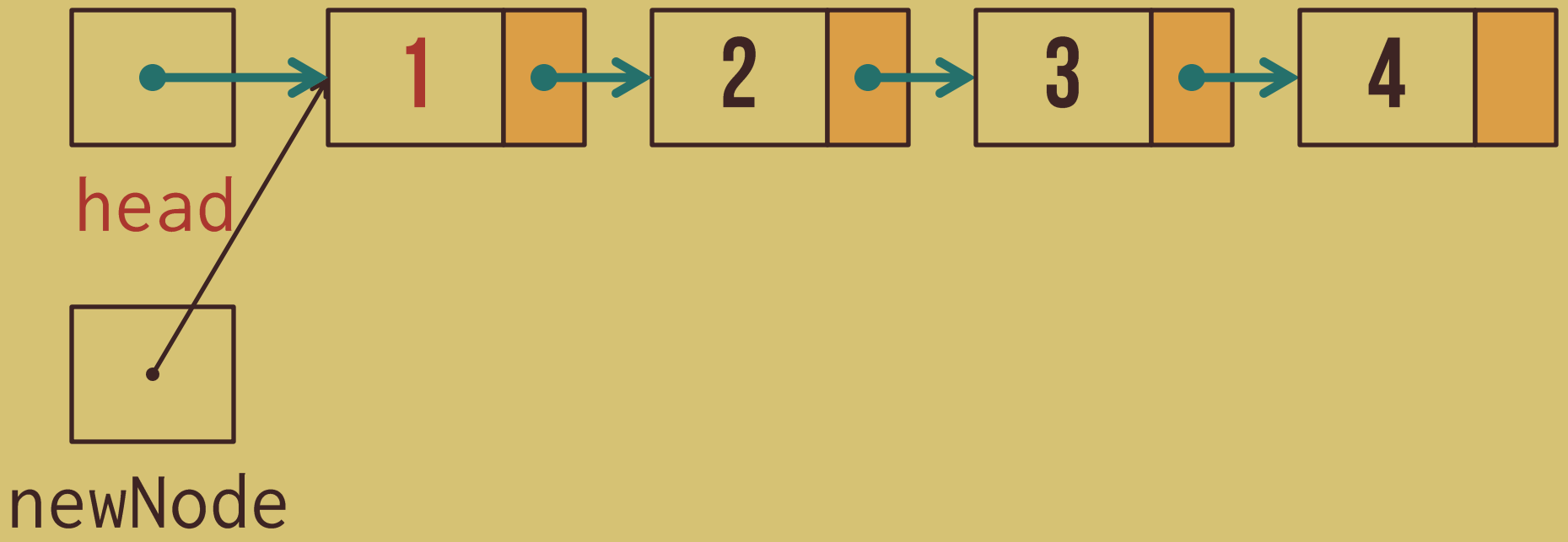
1. Make the next pointer of the new node (1) point to the node pointed to by head (2).

INSERT AT HEAD



2. Make the head pointer point to the new node (1) .

INSERT AT HEAD

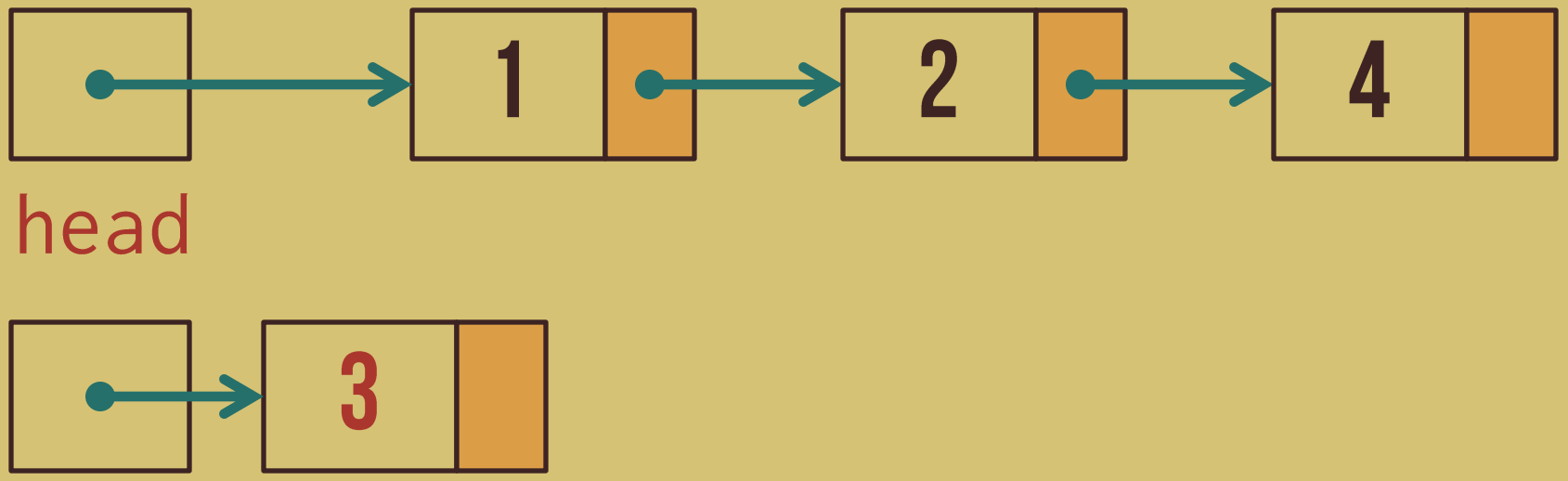


INSERT AT MIDDLE

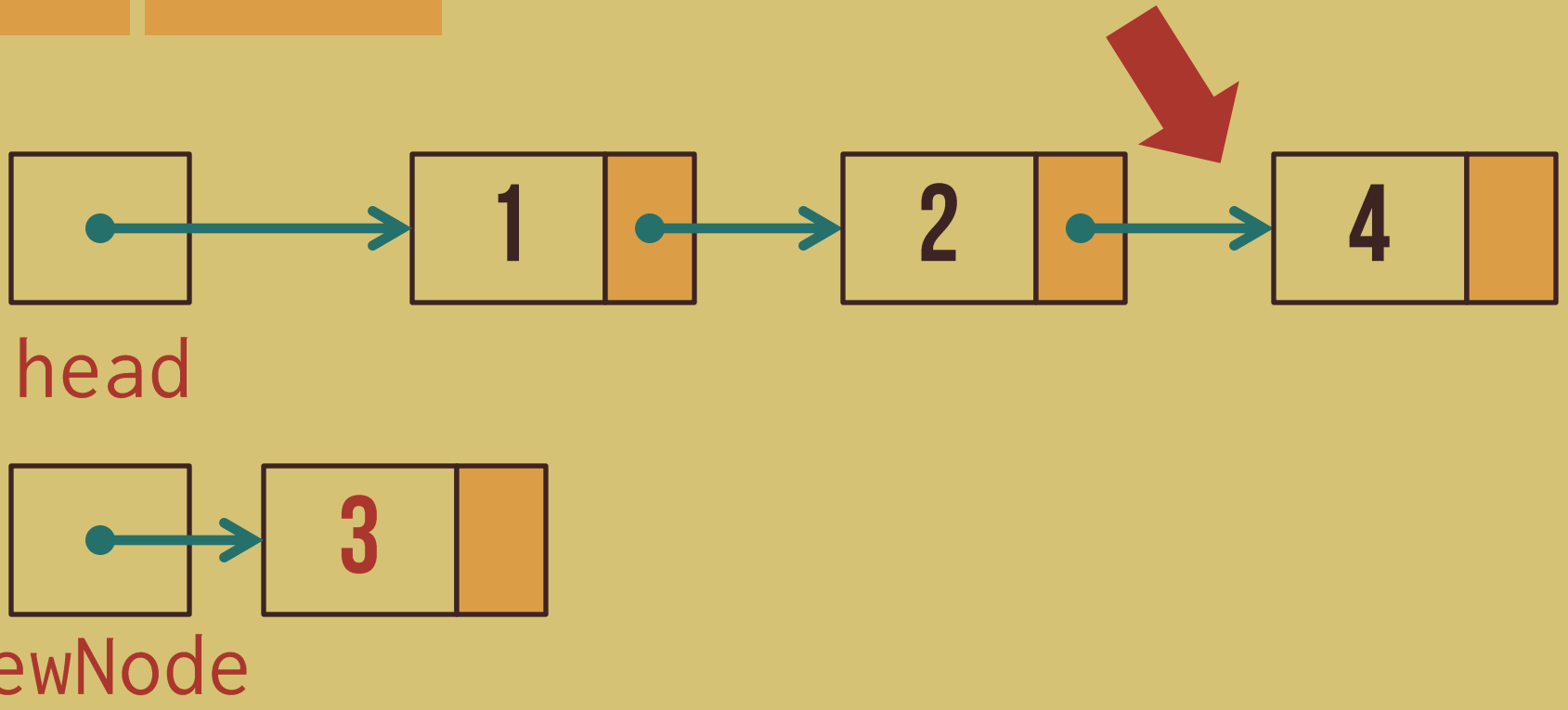
INSERT AT MIDDLE

You have a new node that you want to insert somewhere in between the linked list.

INSERT AT MIDDLE

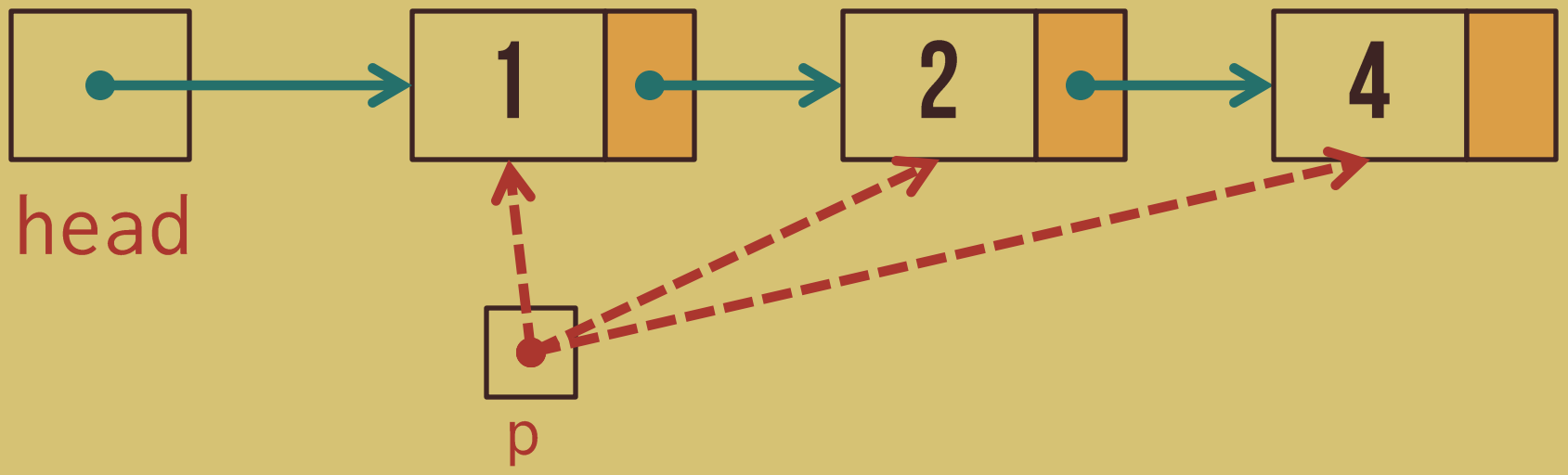


INSERT AT MIDDLE



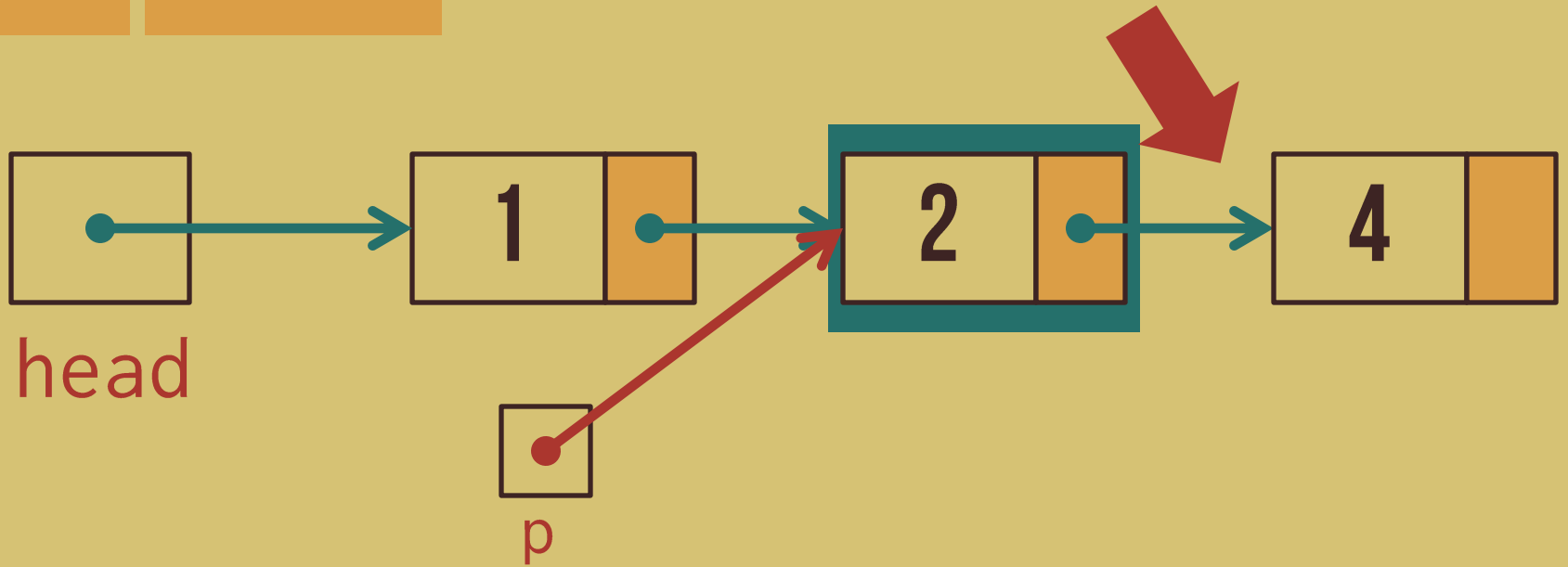
1a. Find the position where the node is to be inserted.

INSERT AT MIDDLE



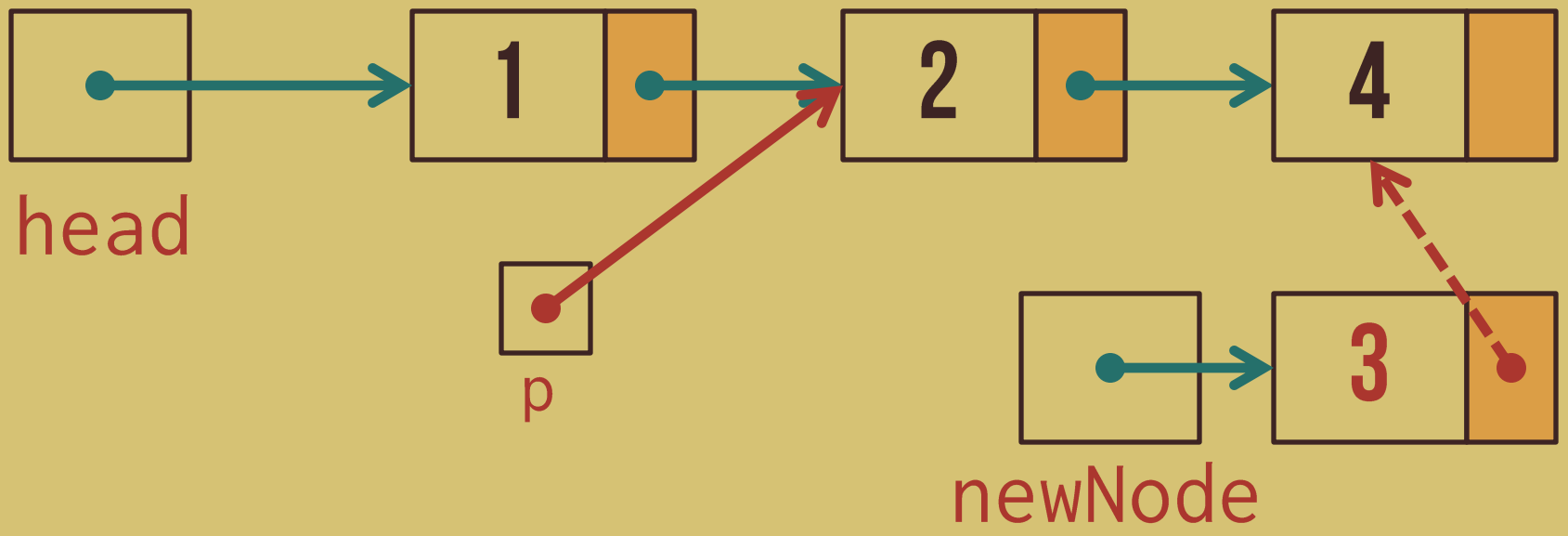
We usually do this by traversing using a pointer (p).

INSERT AT MIDDLE



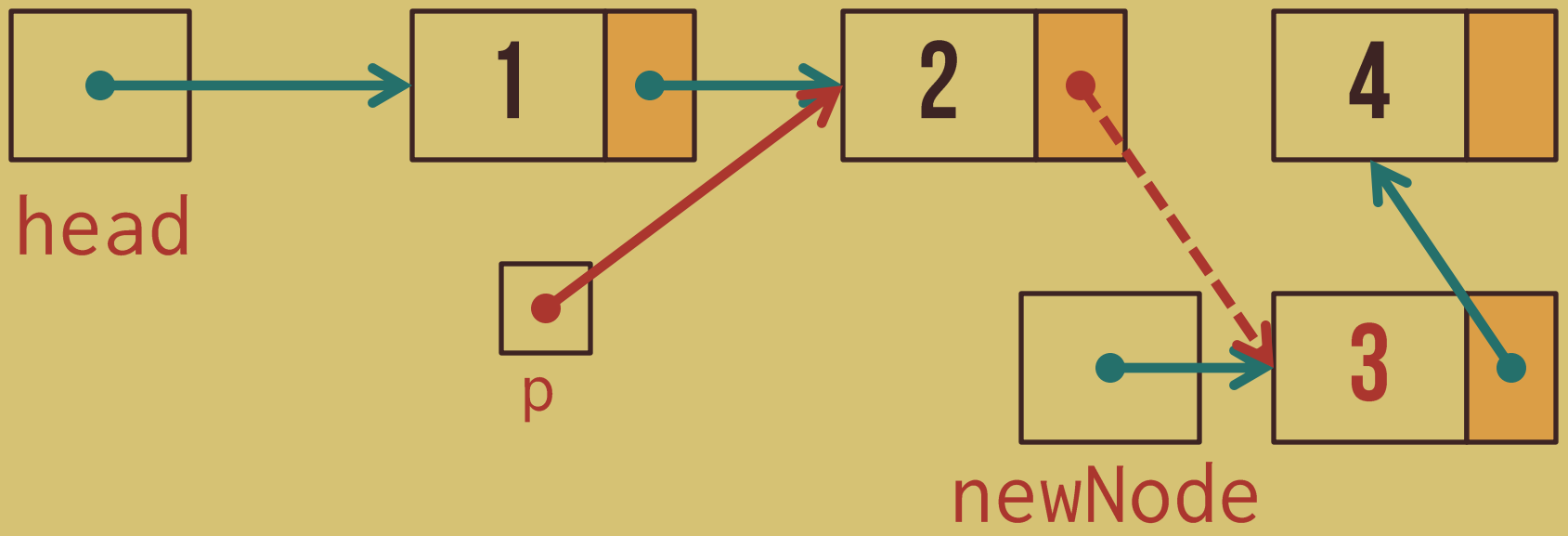
1b. Let's remember/select the node *before* the position we want to insert to.

INSERT AT MIDDLE



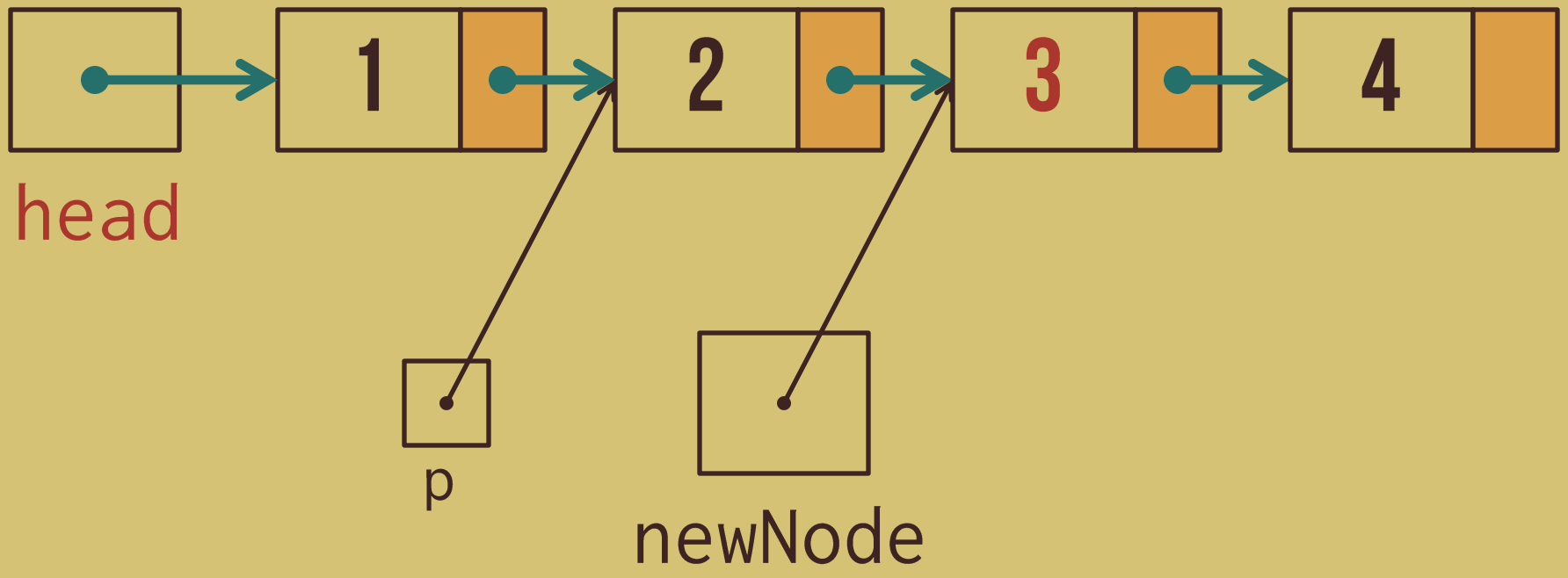
2. Make the next pointer of the new node (3) point to the node next to the node we selected in step 1b (2).

INSERT AT MIDDLE



3. Make the next pointer of the node selected in step 1b (2) point to the new node (3).

INSERT AT MIDDLE



Usually used in conjunction with insert at head.

INSERT AT TAIL

INSERT AT TAIL

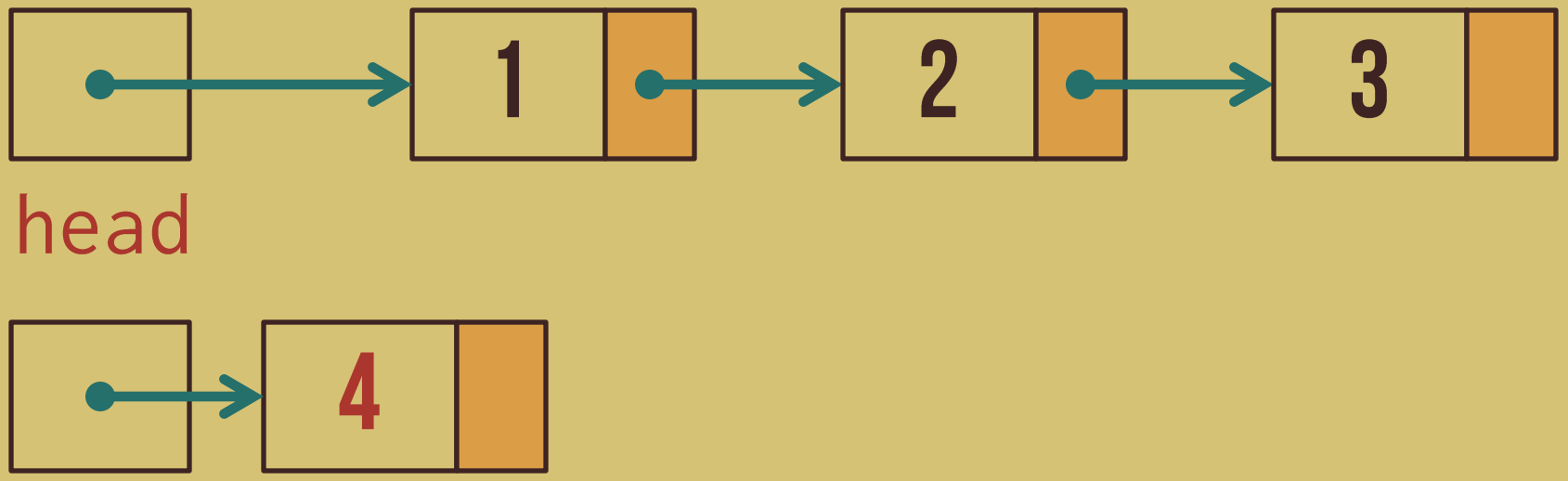
Used when the position of the new value is at the end of the linked list.

INSERT AT TAIL

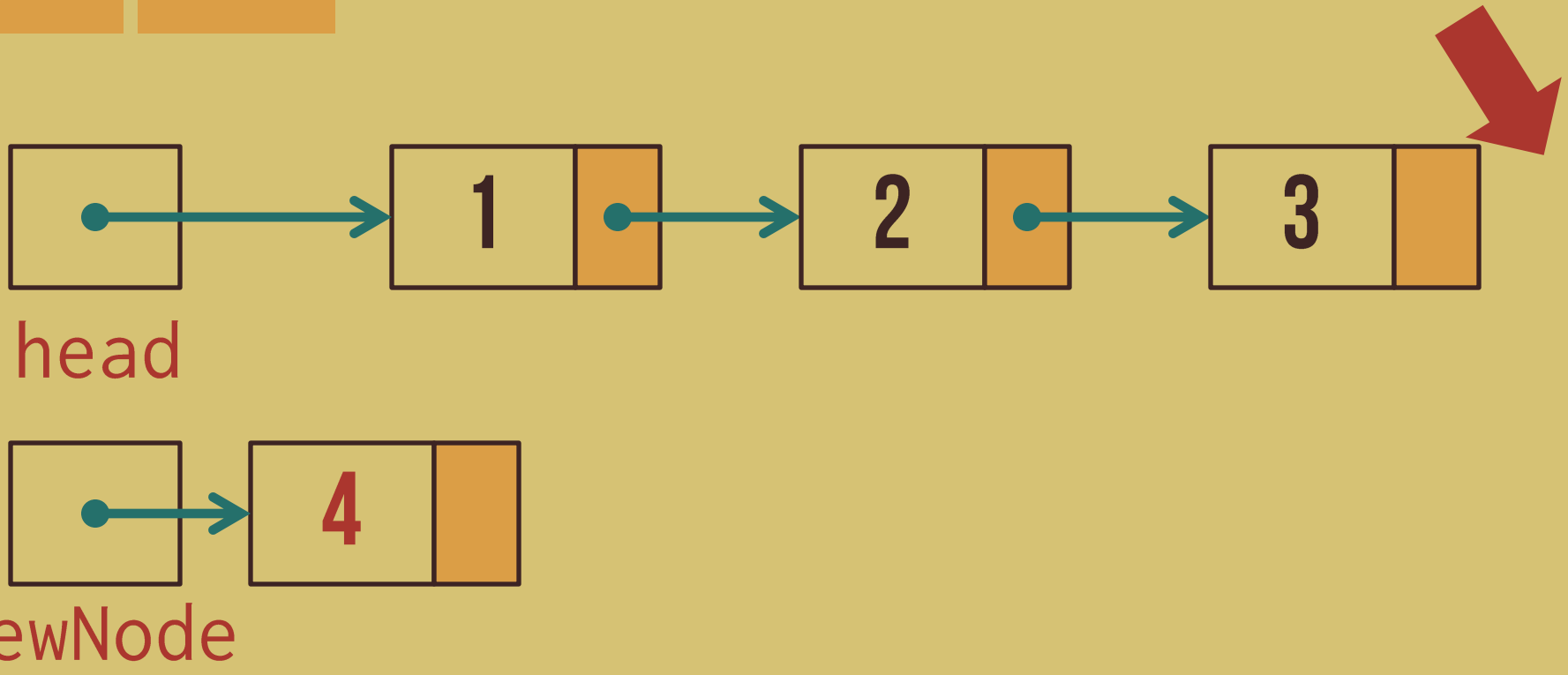
Can be treated as a special case of inserting at the middle*.

*There surely will be implementation differences.

INSERT AT TAIL

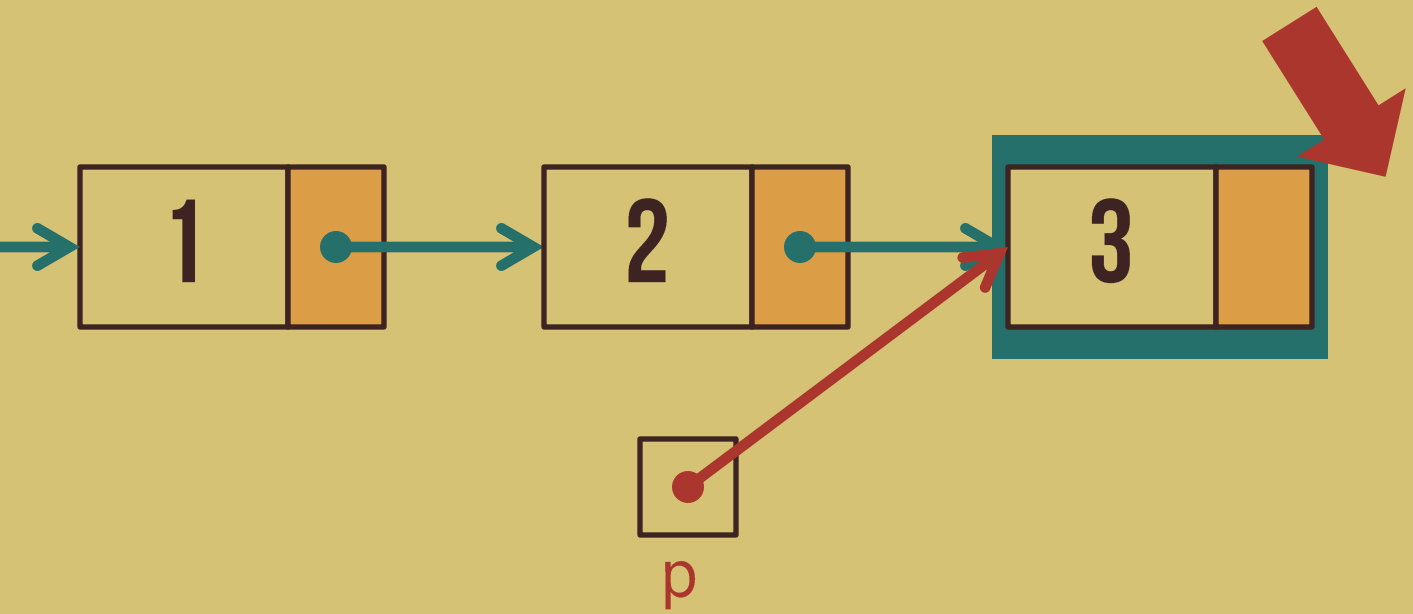


INSERT AT TAIL



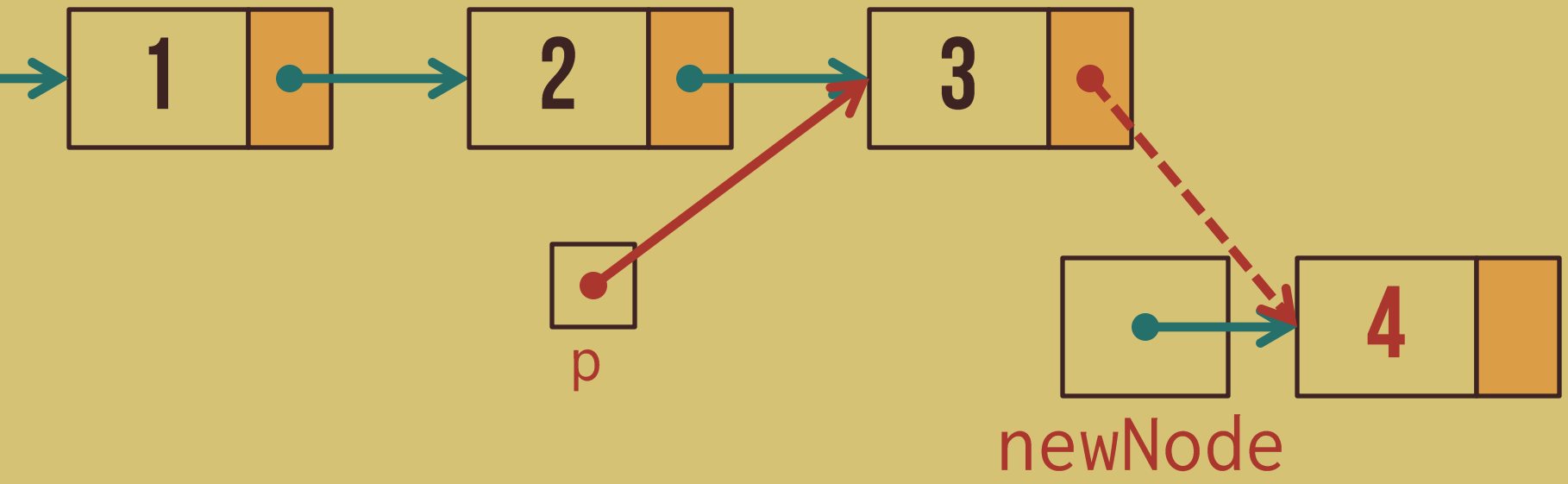
1a. Find the position where the node is to be inserted.

INSERT AT TAIL



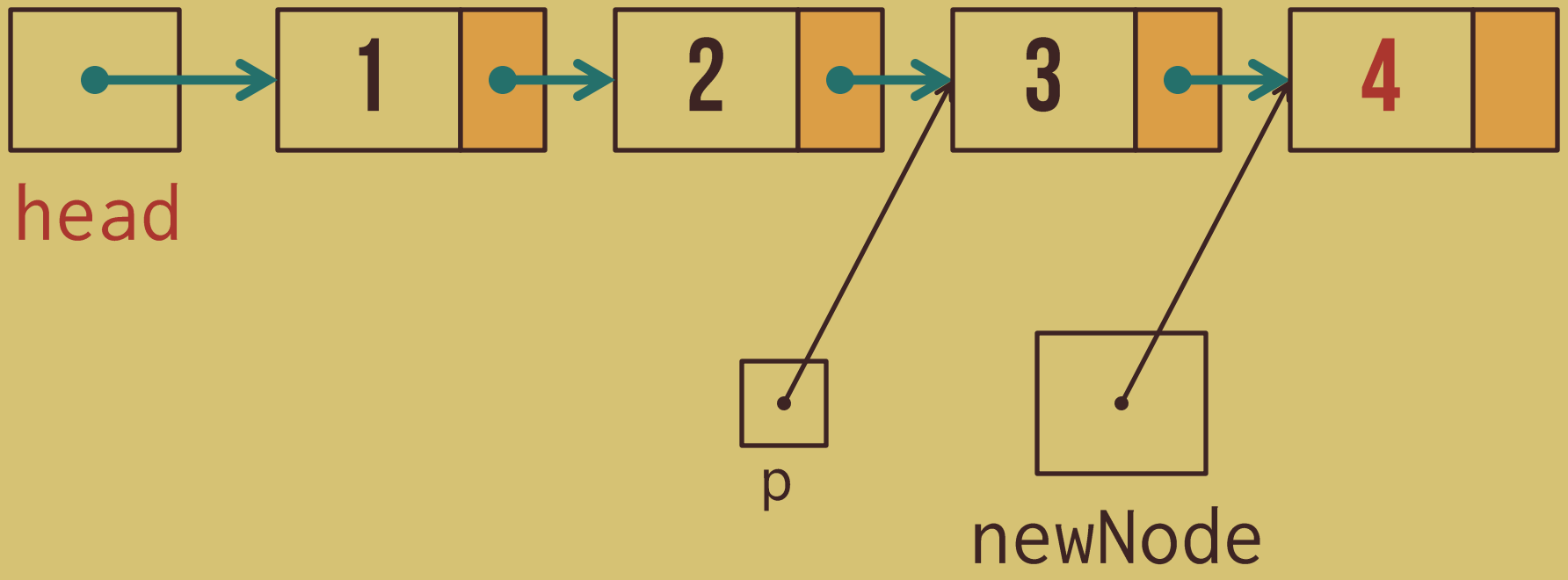
1b. Let's remember/select the node *before* the position we want to insert to.

INSERT AT TAIL



2. Make the next pointer of the node selected in step 1b (3) point to the new node (4).

INSERT AT TAIL



INSERT

SOME NOTES

To mark the end of the list, the next pointer of the last node is given the value NULL.

INSERT

NULL is a const value defined in `stdlib.h`.

INSERT

In pointers, it’s used to symbolize that a pointer is not pointing anywhere.

In our visualizations, we assume that a pointer field that does not have an outgoing arrow has a NULL value.

INSERT

To prevent pointers having garbage values, ALWAYS initialize your pointers to NULL.

INSERT DELETE

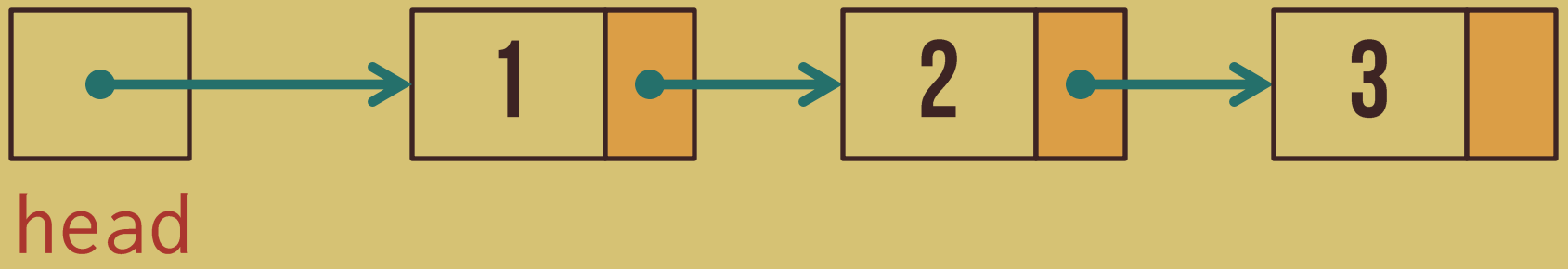
DELETE

Used to delete elements from the linked list.

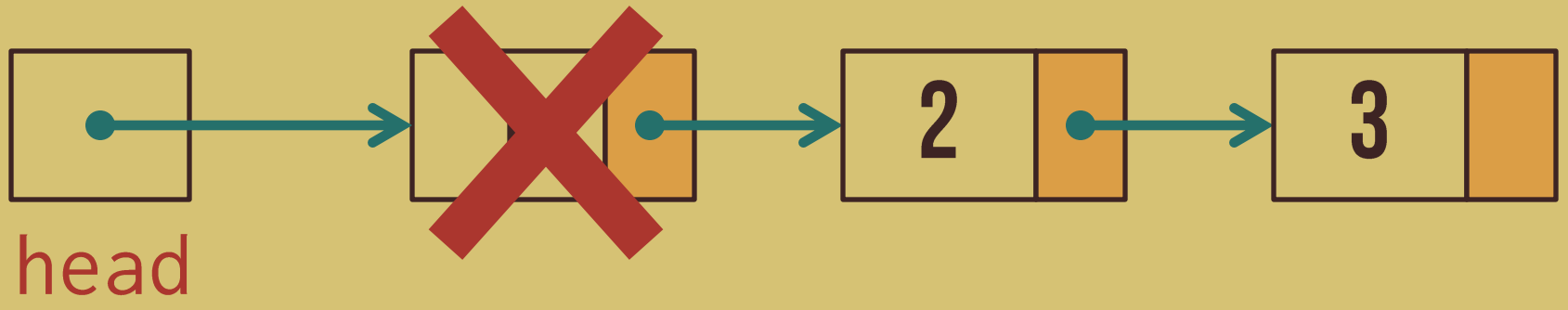
Different CASES of DELETE:

DELETE AT HEAD

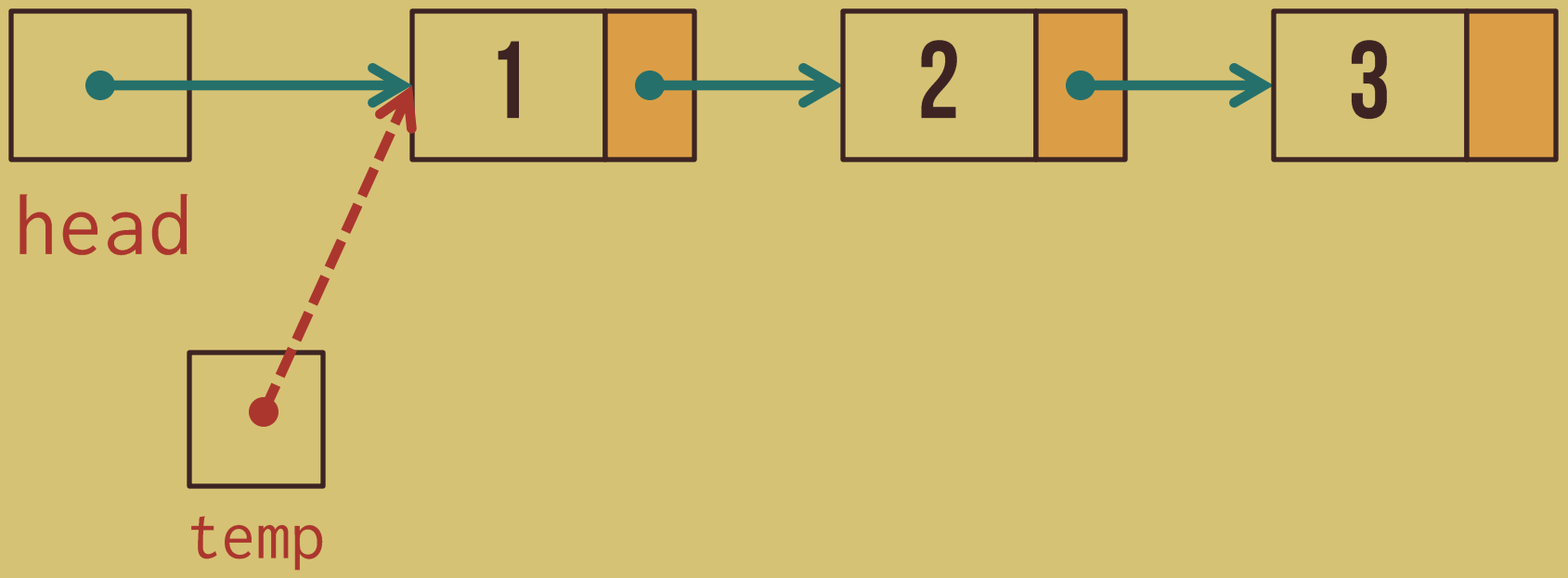
DELETE AT HEAD



DELETE AT HEAD

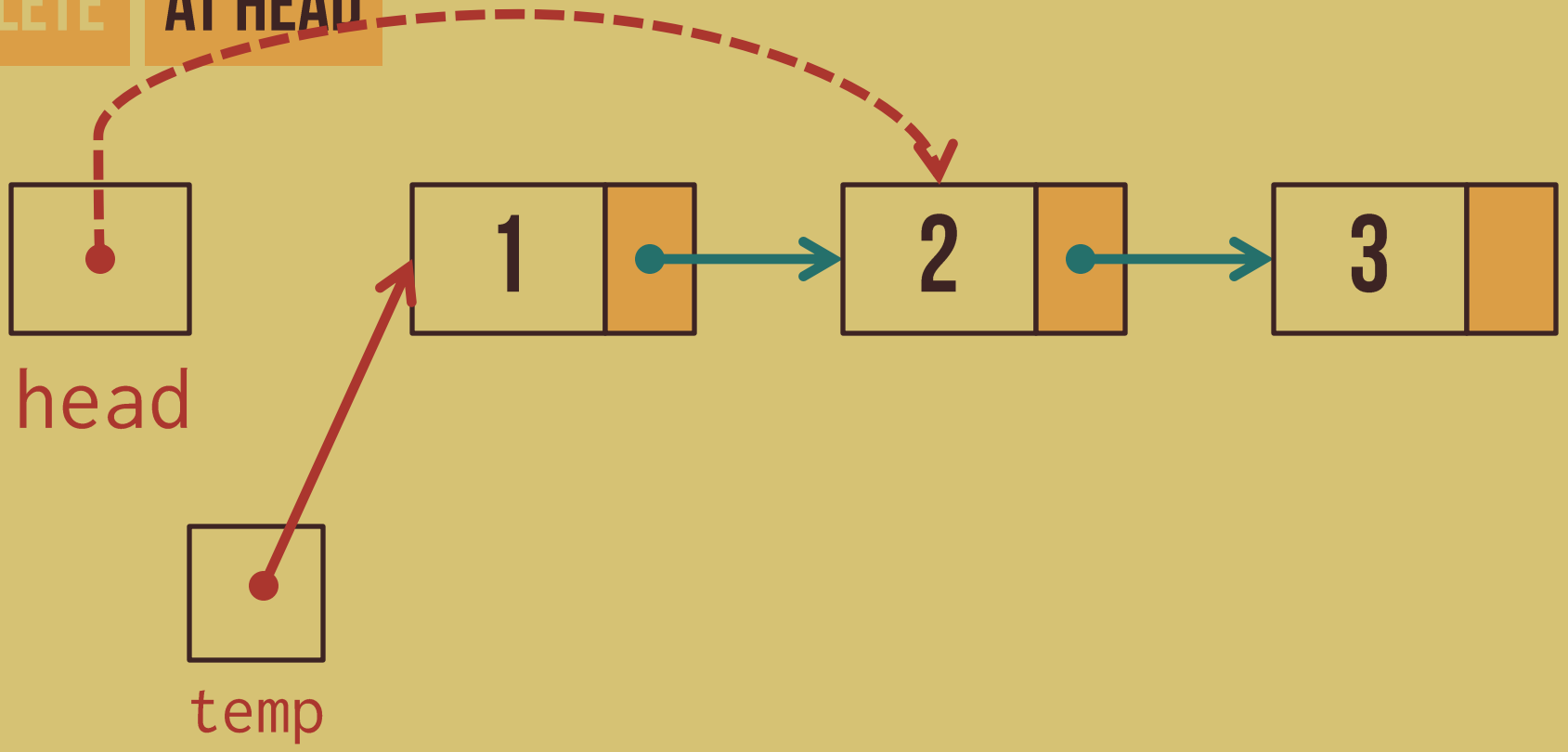


DELETE AT HEAD



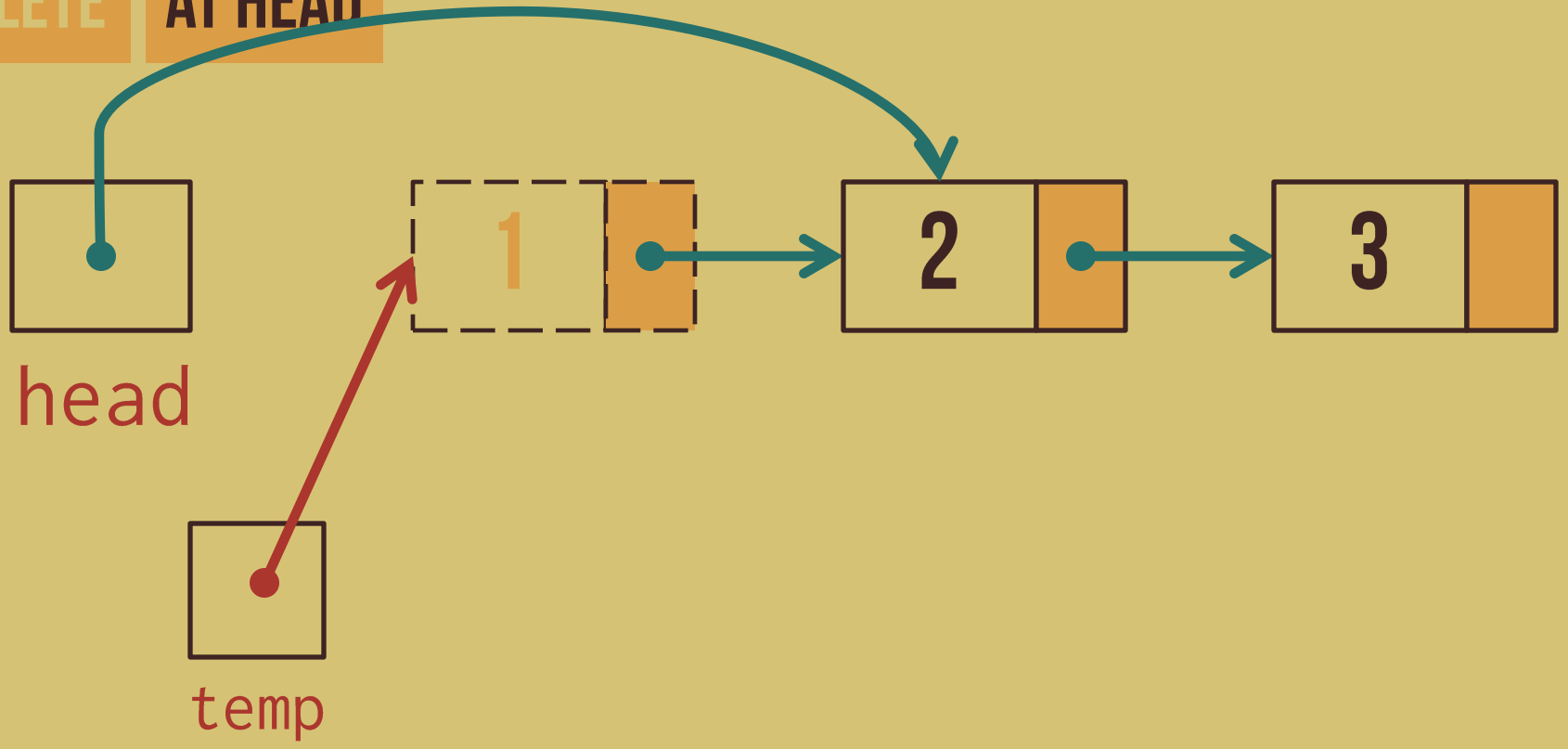
1. Have another pointer (temp) point to the first node.

DELETE AT HEAD



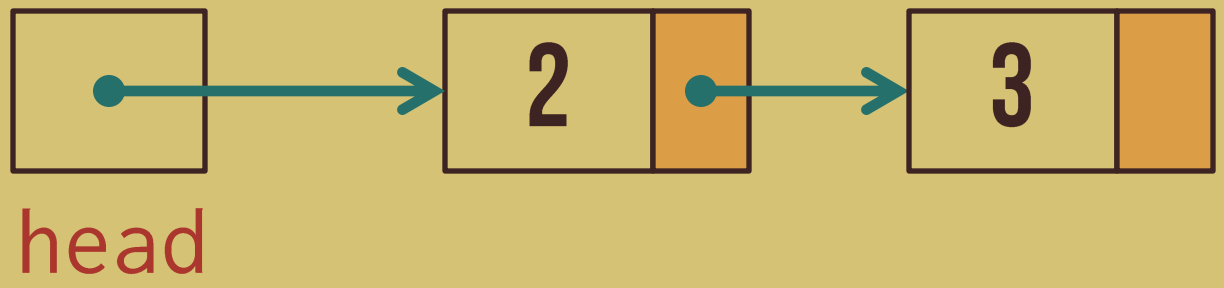
2. Make head refer to the second node.

DELETE AT HEAD



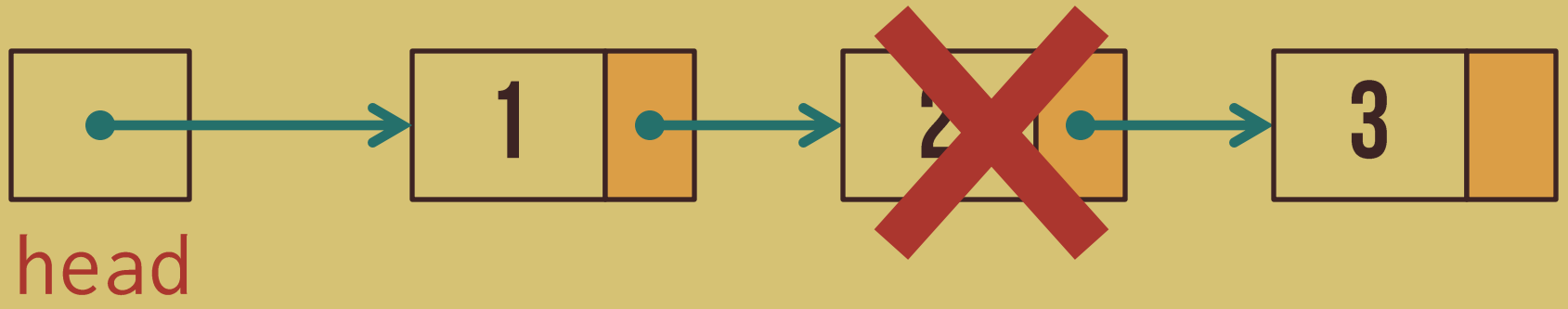
3. Free the first node using temp.

DELETE AT HEAD

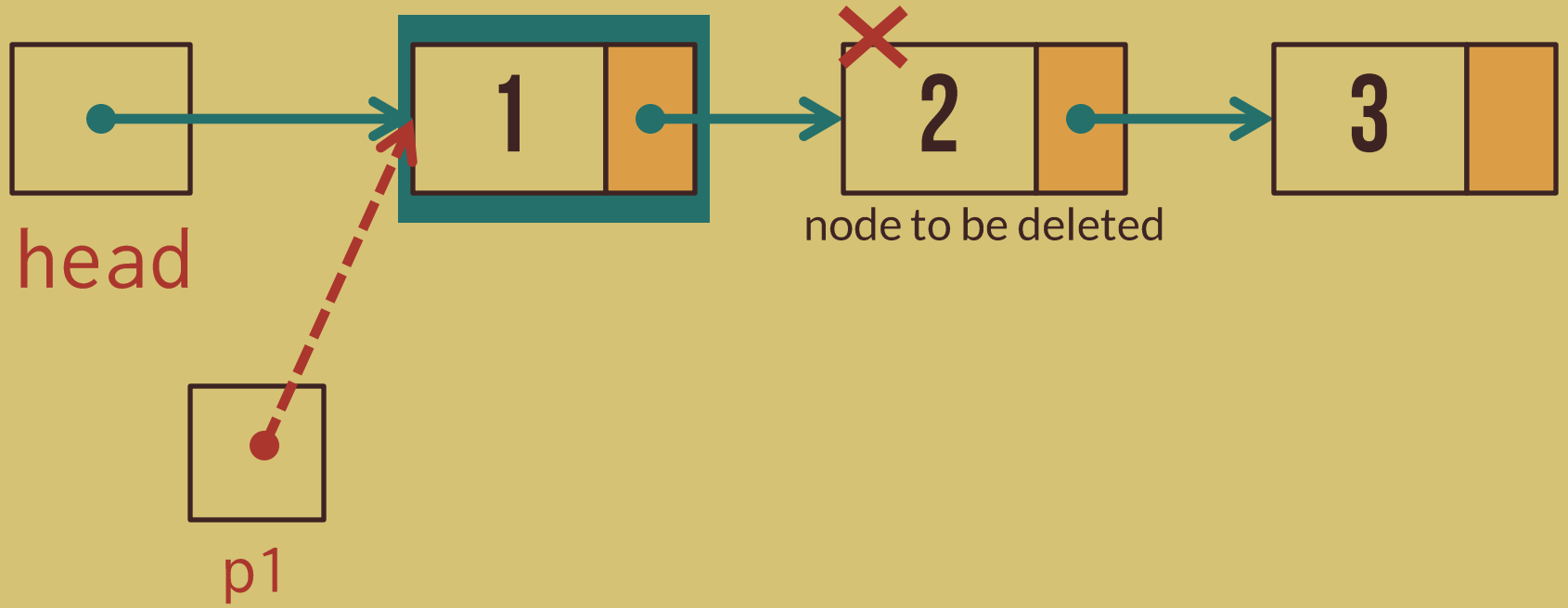


DELETE AT MIDDLE

DELETE AT MIDDLE

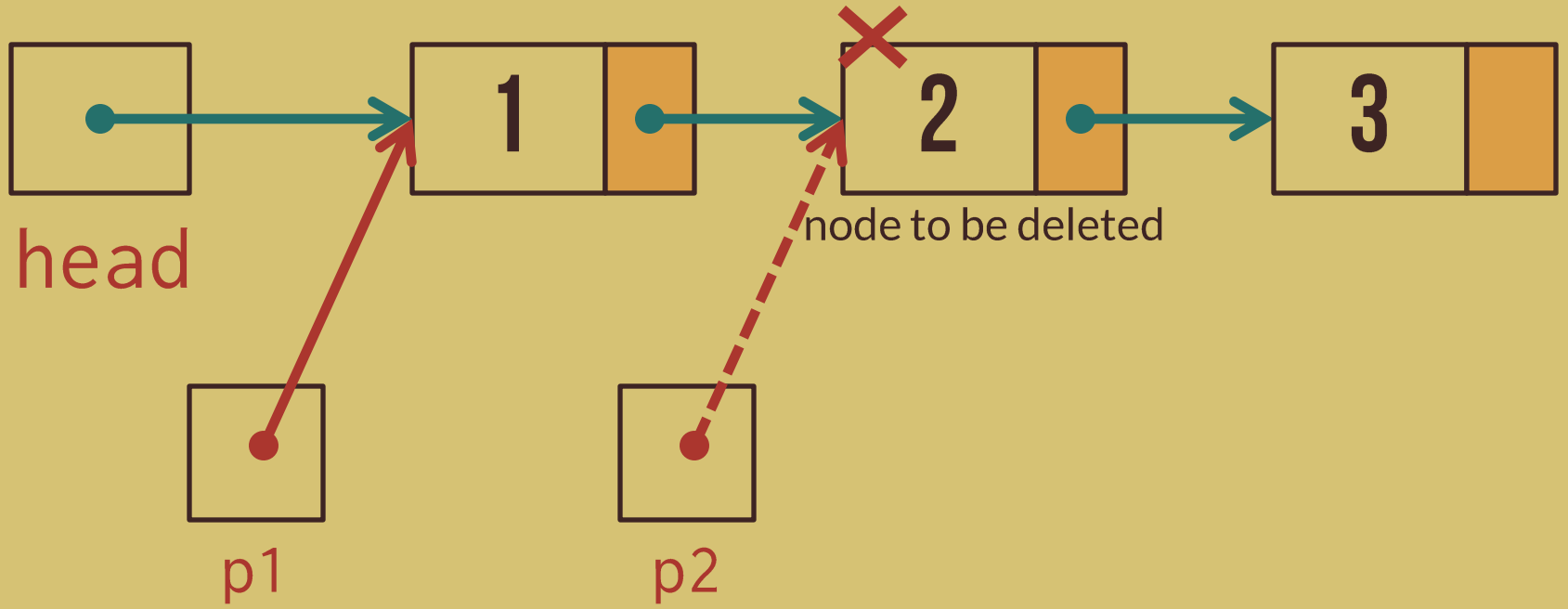


DELETE AT MIDDLE



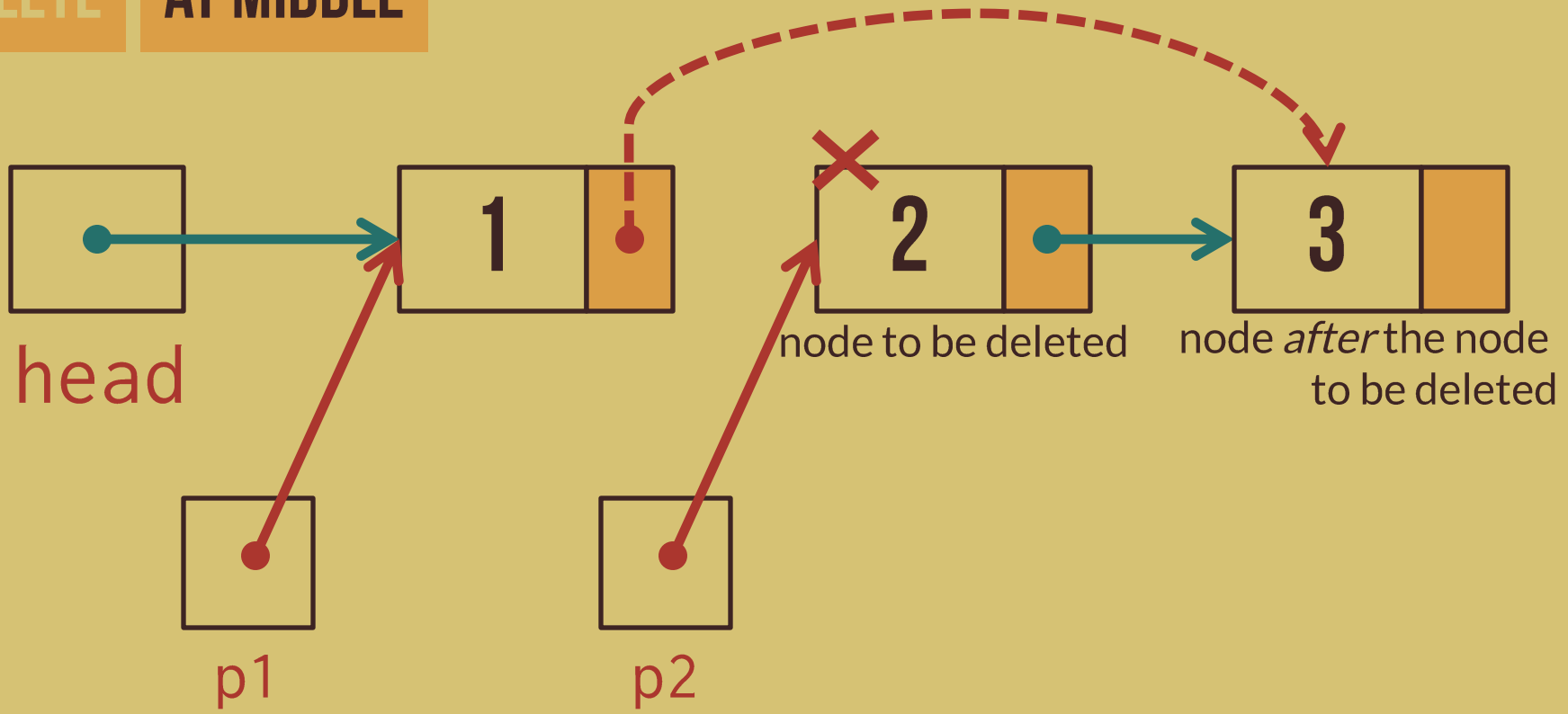
1. Find the node *before* the node to be deleted and let a pointer (p1) point it.

DELETE AT MIDDLE



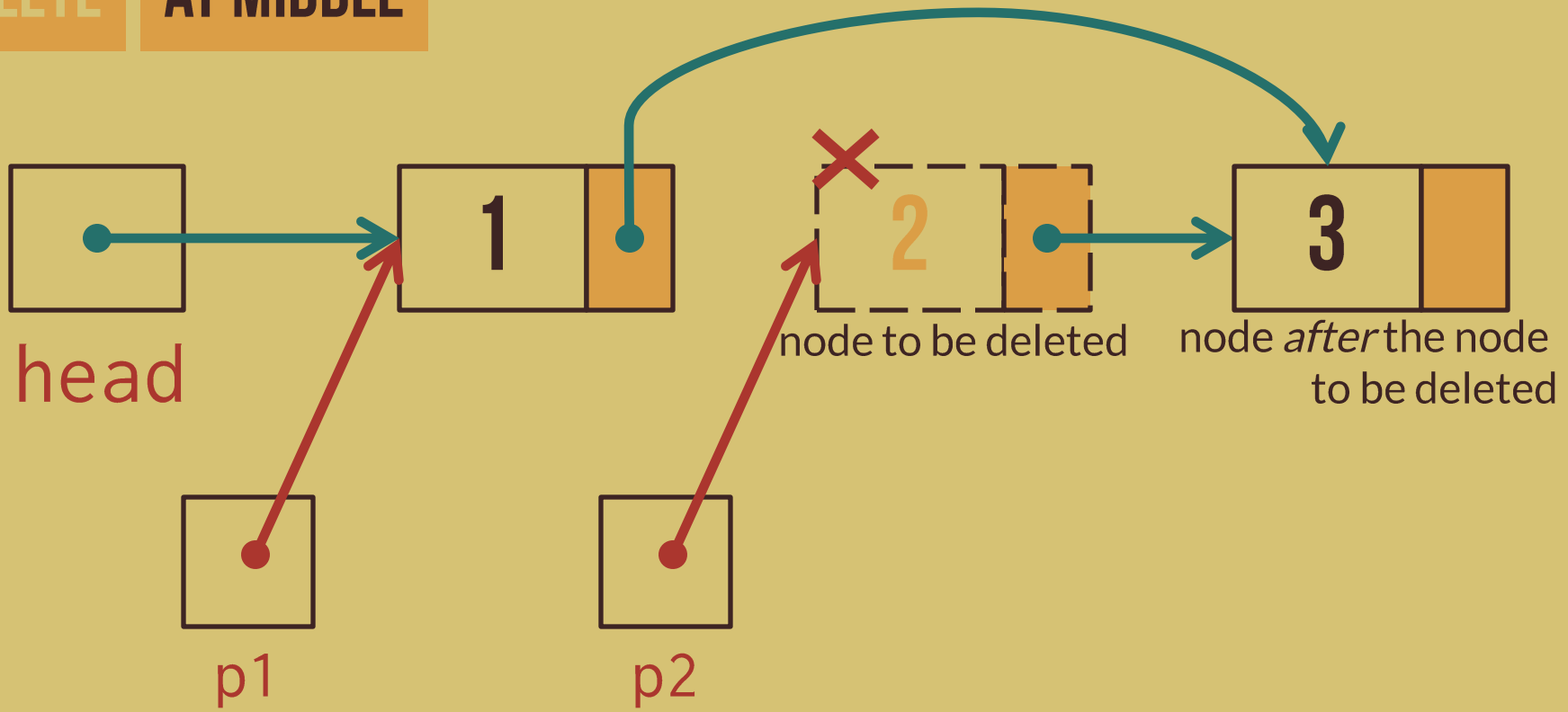
2. Have another pointer (p2) refer to the node to be deleted.

DELETE AT MIDDLE



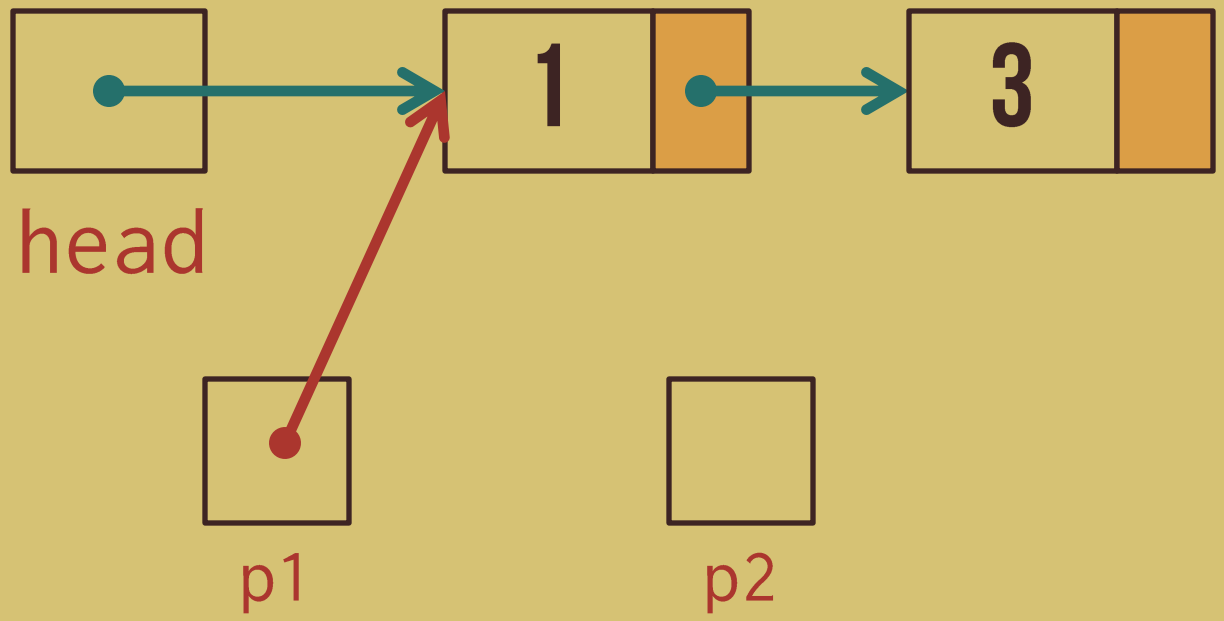
3. Make the next pointer of the node pointed by p1 refer to the node *after* the node to be deleted (using p2).

DELETE AT MIDDLE



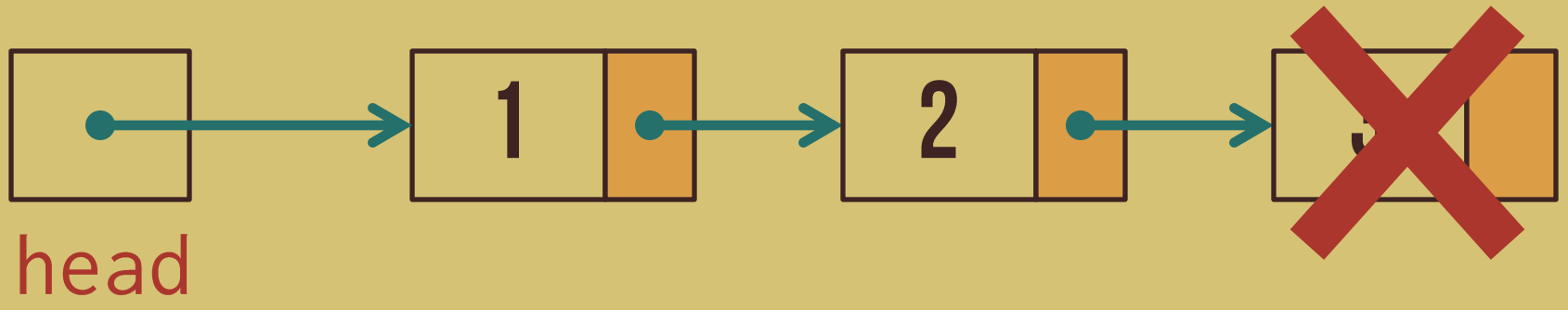
4. Free the node to be deleted using p2.

DELETE AT MIDDLE

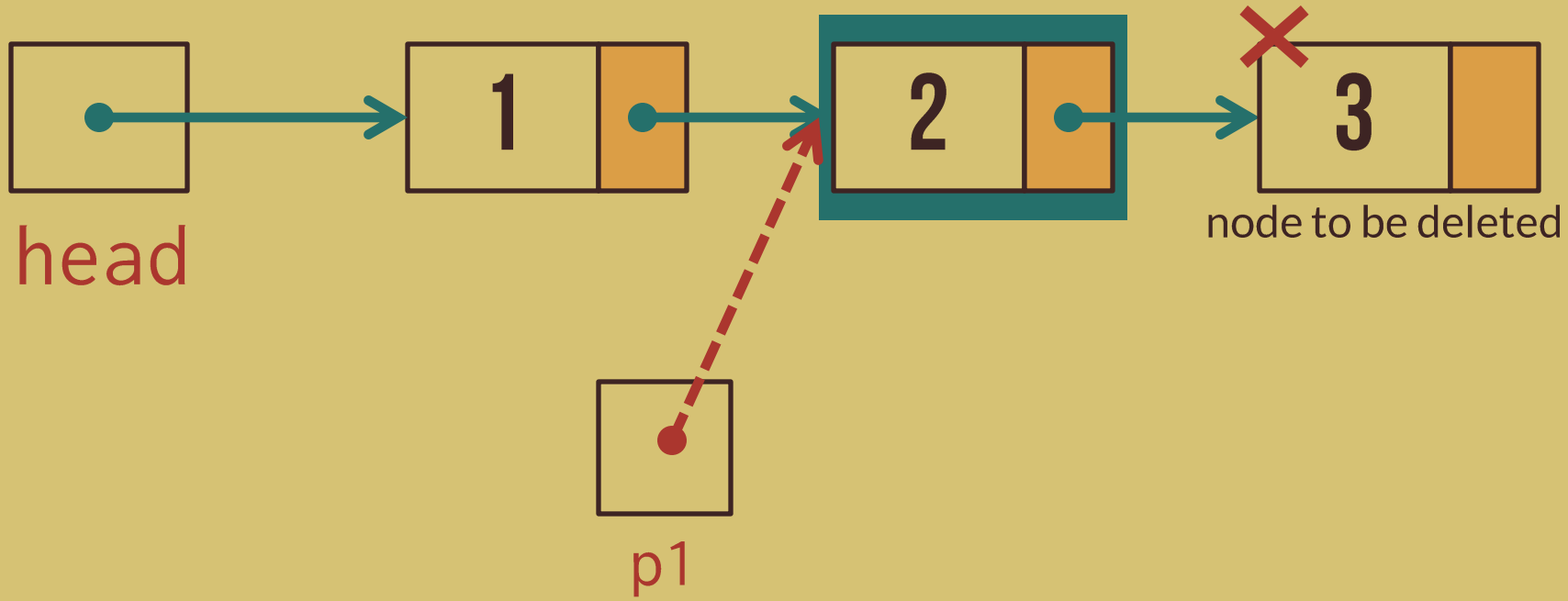


DELETE AT TAIL

DELETE AT TAIL

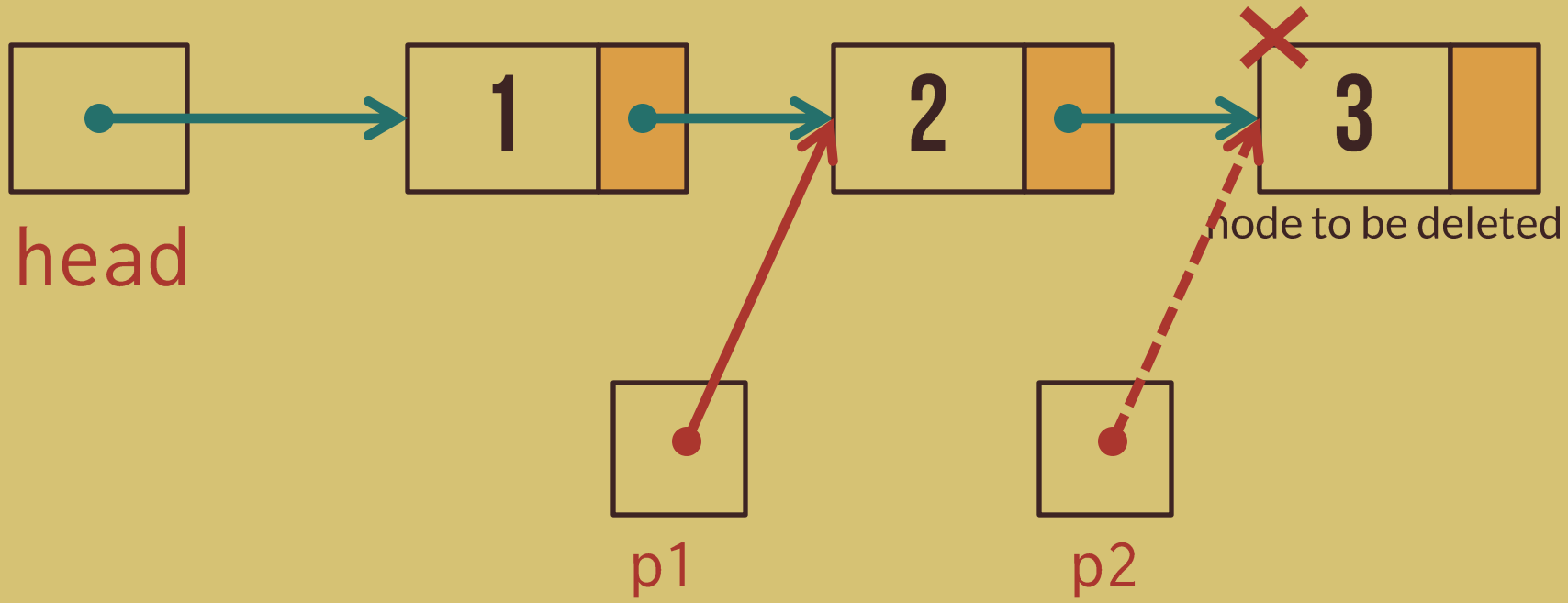


DELETE AT TAIL



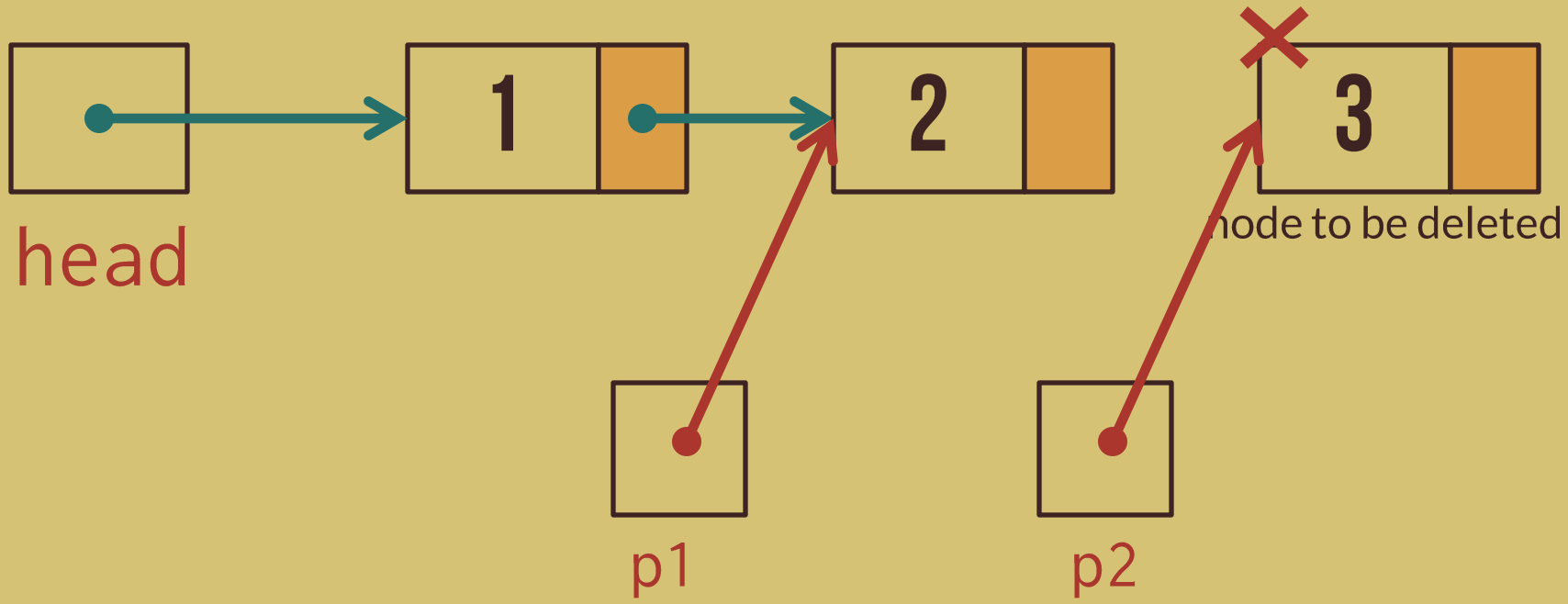
1. Find the node *before* the tail using a pointer (p1).

DELETE AT TAIL



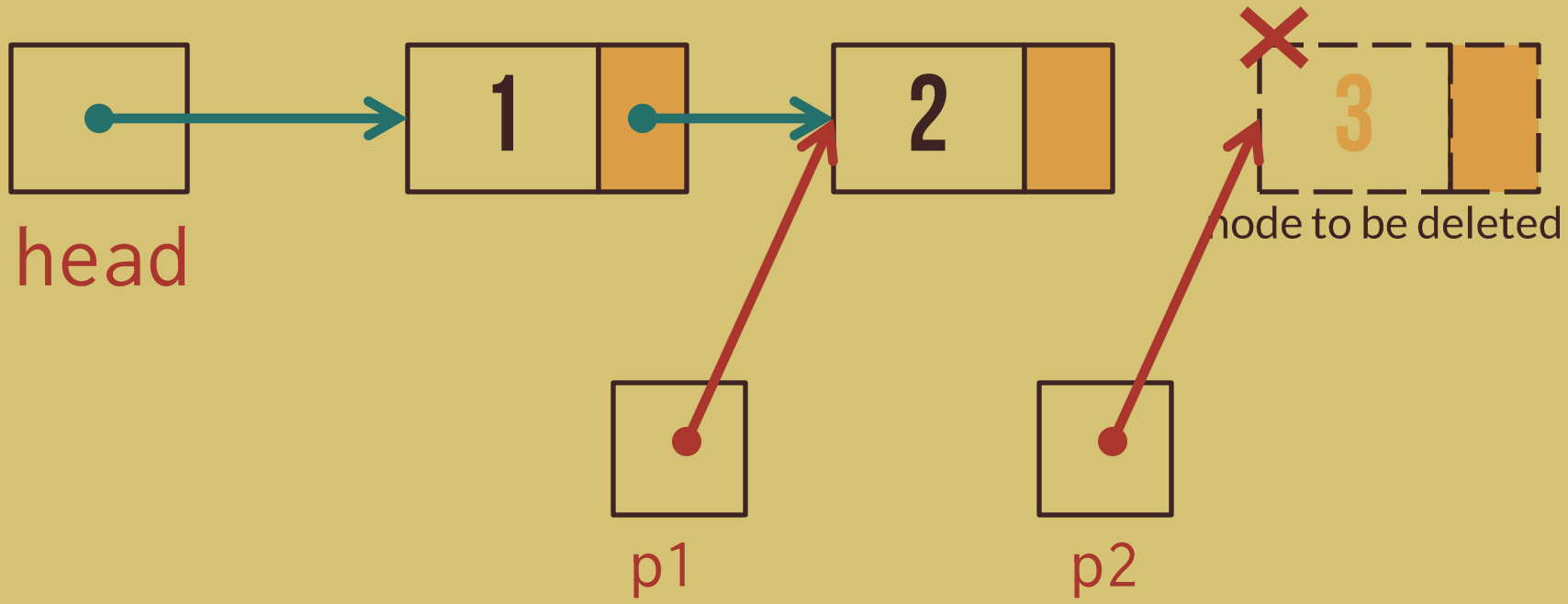
2. Have another pointer (p2) refer to the last node.

DELETE AT TAIL



3. Make the next pointer of the node referred to by p1 to NULL.

DELETE AT TAIL



4. Free the node pointed by p2.

DELETE AT TAIL