



DOUBLY LINKED LISTS

CS A250 – C++ Programming II

REVIEW

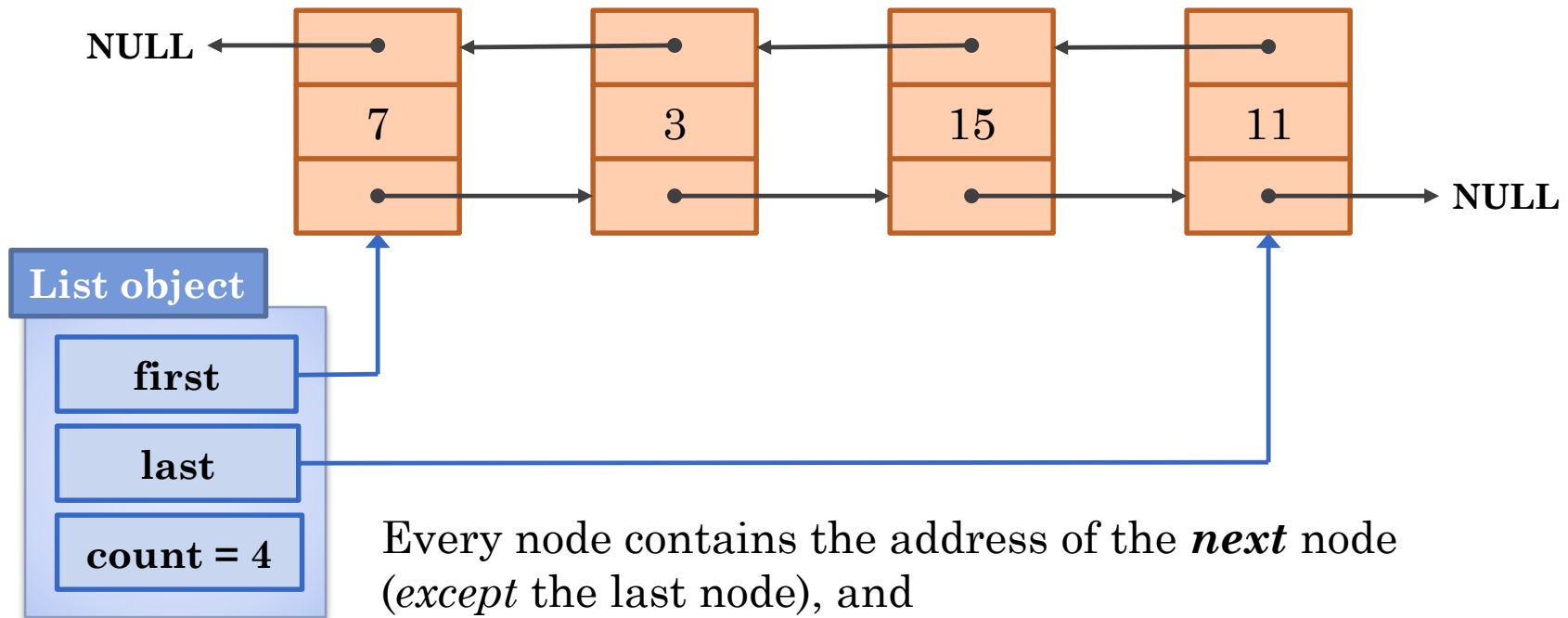
- Singly-linked list
 - Constructed using **pointers**
 - **Grows** and **shrinks** during *runtime*
 - **Doubly-linked lists:**
 - A variation with **pointers** in *both directions*
- **Pointers** are the backbone of such structures
 - Use *dynamic* variables
- **Standard Template Library**
 - Has predefined versions of some structures

DOUBLY-LINKED LISTS

○ A **doubly-linked list**

- Links to **next** node *and* to **previous** node
- Can follow link in either direction
- Can make some operations easier
- **NULL** signifies the **beginning** *and* the **end** of the list

DOUBLY-LINKED LISTS (CONT.)



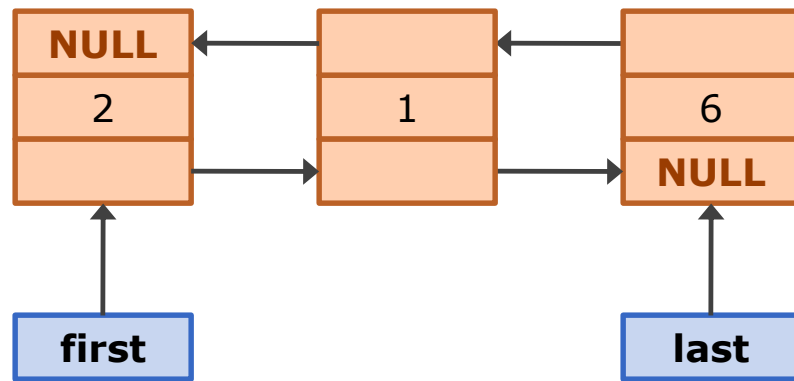
DOUBLY-LINKED NODE DEFINITION

```
class Node
{
public:
    Node () : data(0), nextLink(NULL), previousLink(NULL) {}
    Node (int theData, Node *previous, Node *next)
        : data(theData), nextLink(next), previousLink(previous) {}
    Node *getNextLink( ) const { return nextLink; }
    Node *getPreviousLink( ) const { return previousLink; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setNextLink(Node *pointer) { nextLink = pointer; }
    void setPreviousLink(Node *pointer) { previousLink = pointer; }
    ~Node()

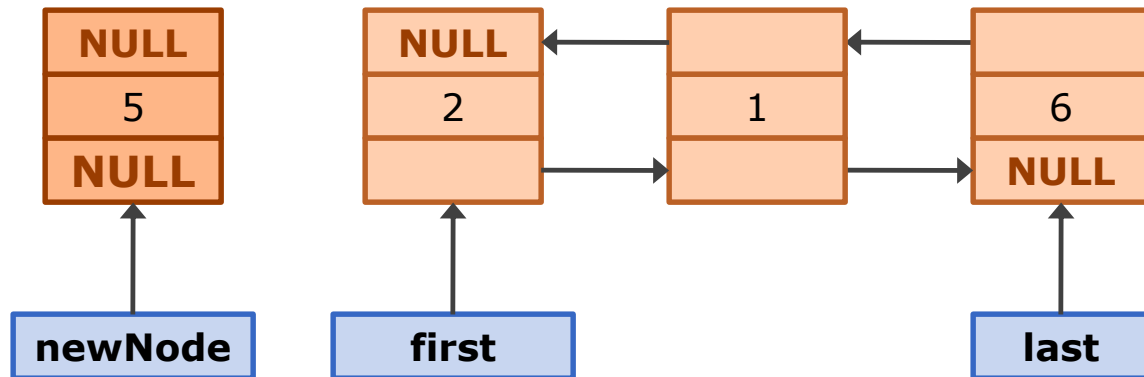
private:
    int data;    //to simplify, we are using only one piece of data
    Node *nextLink;
    Node *previousLink;
};
```

ADDING A NODE TO THE FRONT (1 OF 2)

Existing list *before* adding new node.

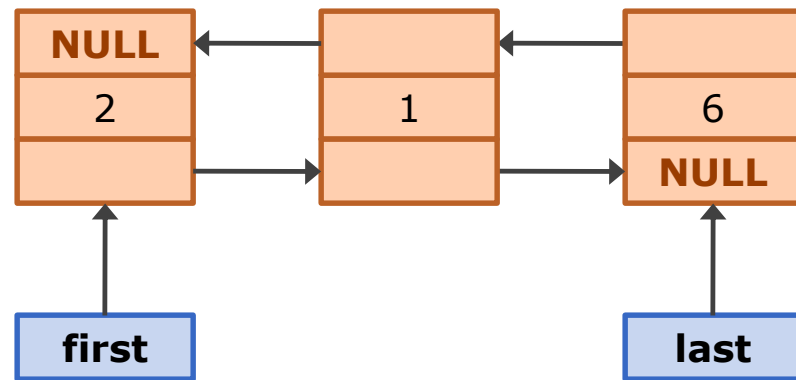


Create a **pointer newNode**, then create a **new node** and point the **pointer newNode** to the **newly created node**.

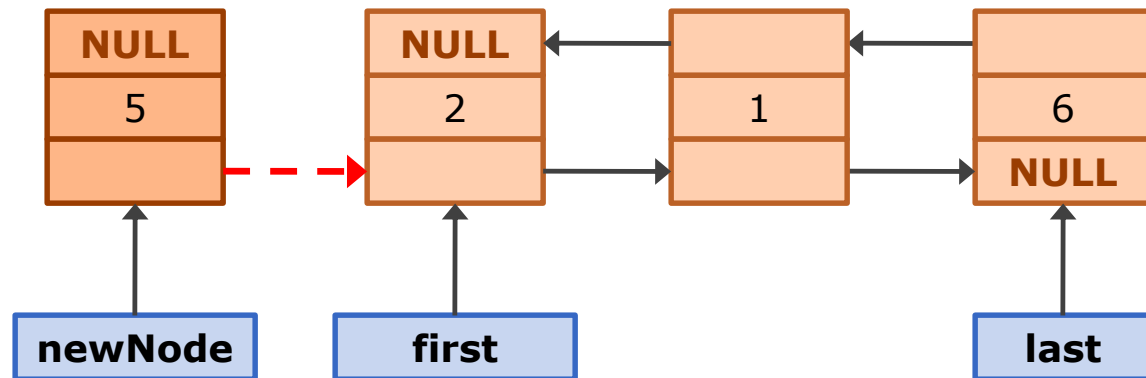


ADDING A NODE TO THE FRONT (1 OF 2)

Existing list *before* adding new node.

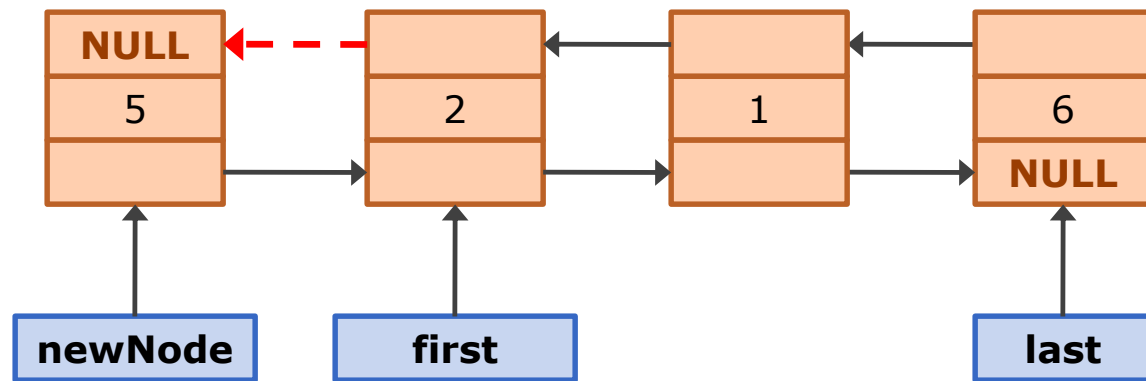


(1) Set pointer **nextLink** of the **new node** to point to the **first node**.

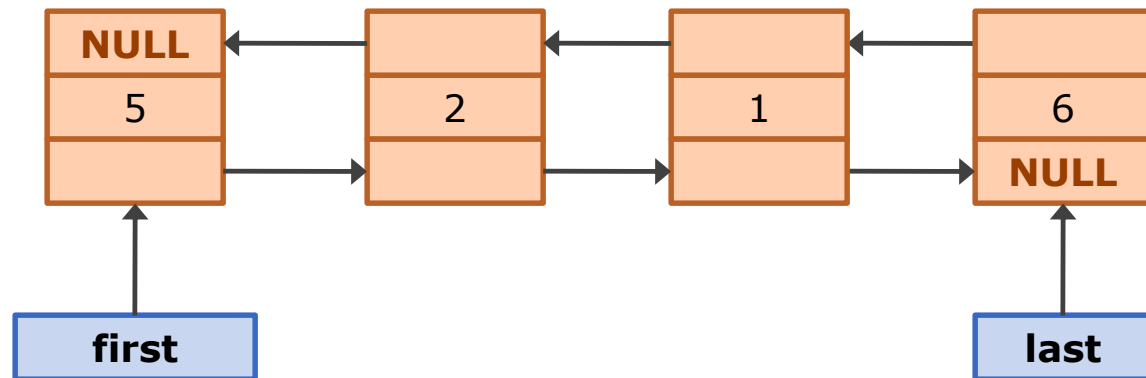


ADDING A NODE TO THE FRONT (2 OF 2)

(2) Set pointer **previousLink** of the **first** node to point to the **new node**.



(3) Set pointer **first** to point to the **new node**.

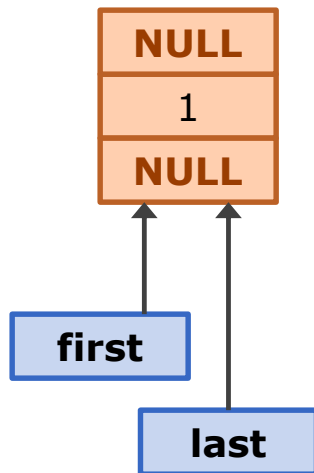


DELETING A NODE FROM THE LIST

- To **delete** a specific node from the list, we need to first find out whether that node exists by searching the list.
- The delete operation has several cases:
 - The *list is empty*
 - The item to be deleted *is in the first node* of the list, which would require us to change the value of pointer first
 - The *first node* is the only node in the list
 - The item to be deleted is *somewhere in the list*
 - The *last node* needs to be deleted
 - The item to be deleted is *not in the list*

DELETING A NODE FROM THE LIST (CONT.)

- **Case:** List contains only one node.
- **Delete:** Node 1

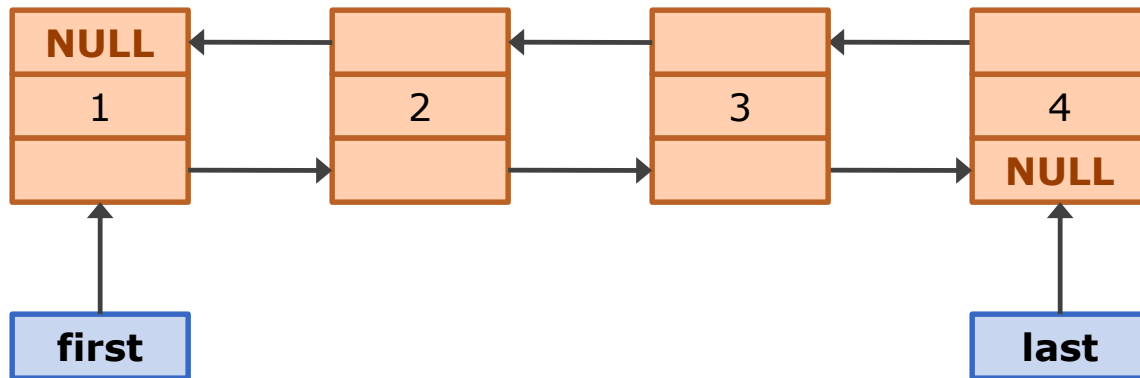


If there is only one node in the list, both pointers **first** and **last** will be pointing to it.

No need to create a pointer, **BUT** you need to re-set both pointers **first** and **last** to **NULL**.

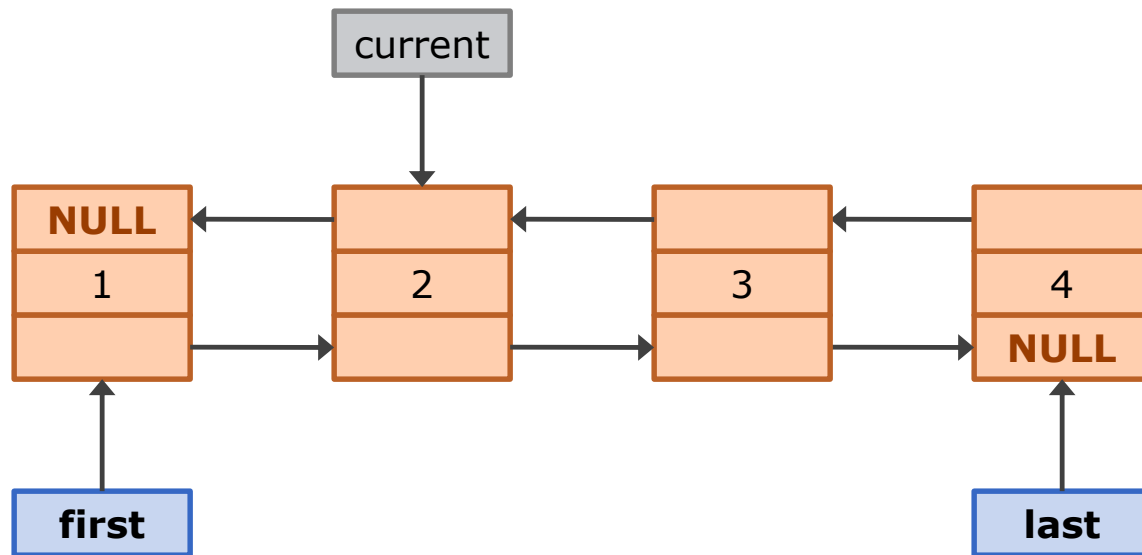
DELETING A NODE FROM THE LIST (CONT.)

- **Case:** List contains more than one node.
- **Delete:** Node 3



DELETING A NODE FROM THE LIST (CONT.)

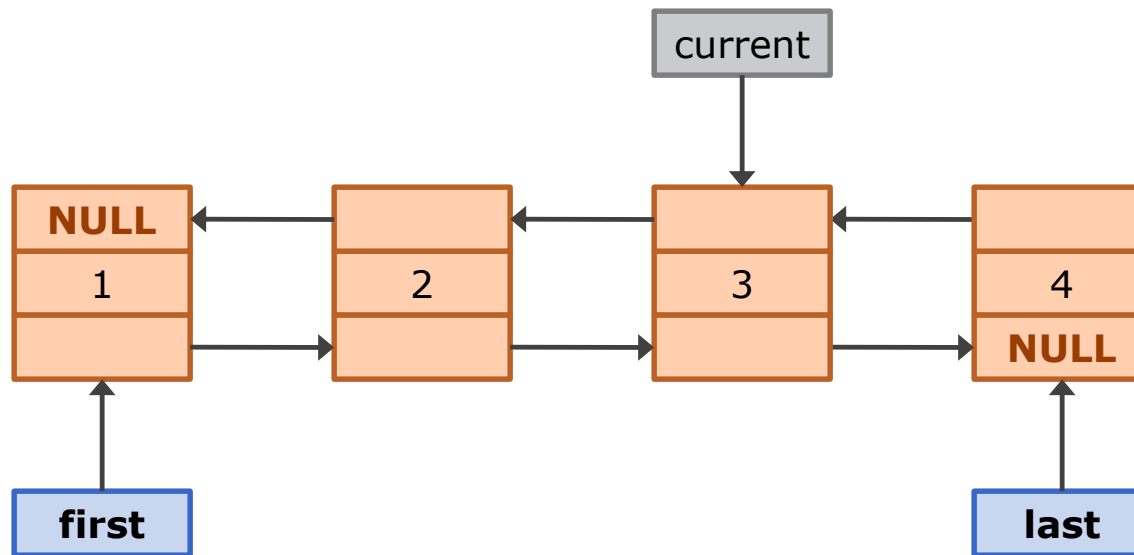
- **Case:** List contains more than one node.
- **Delete:** Node 3



Create a new pointer, **current**, and make it point to the **second** node (you have already checked the first node).

DELETING A NODE FROM THE LIST (CONT.)

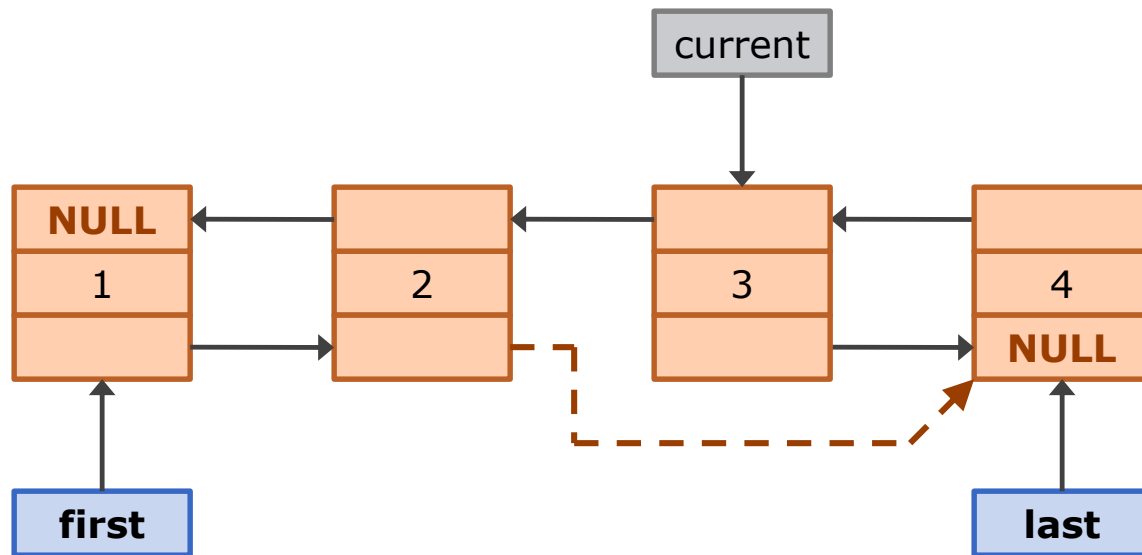
- **Case:** List contains more than one node.
- **Delete:** Node 3



Move pointer **current** forward until you find the node that contains the data to delete.

DELETING A NODE FROM THE LIST (CONT.)

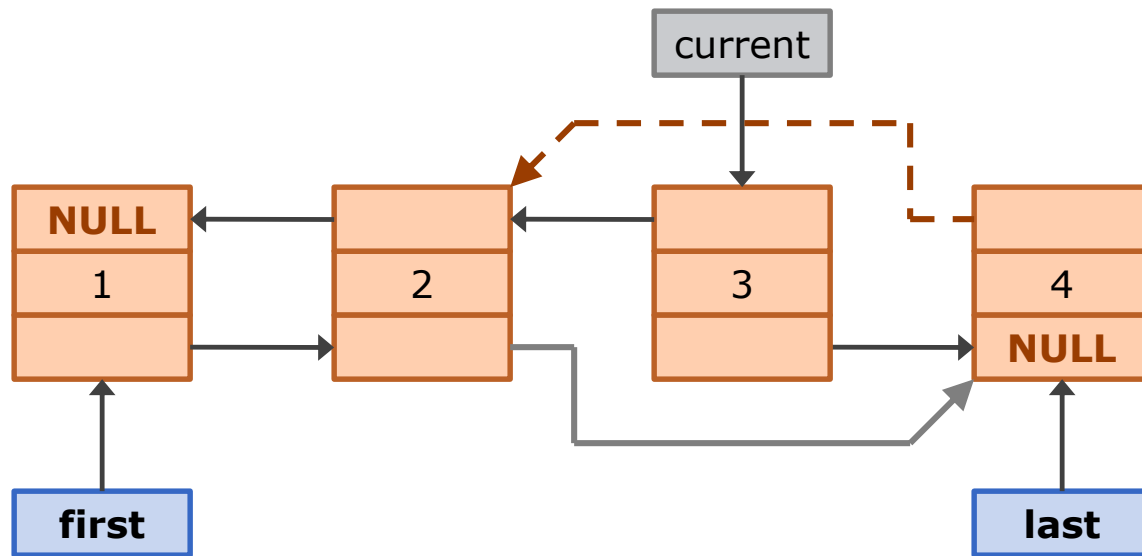
- **Case:** List contains more than one node.
- **Delete:** Node 3



Make node 2 point to node 4.

DELETING A NODE FROM THE LIST (CONT.)

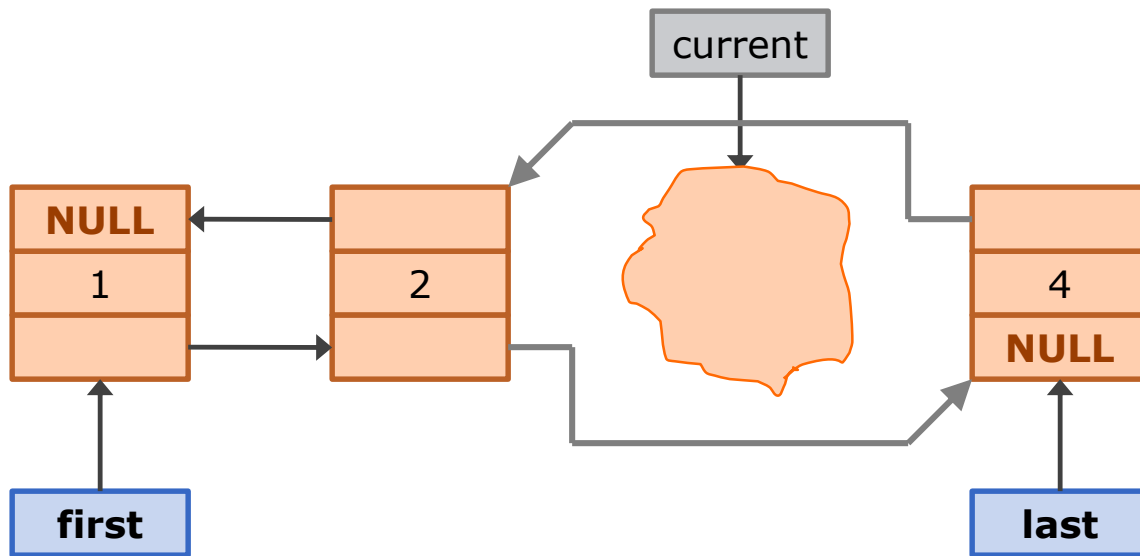
- **Case:** List contains more than one node.
- **Delete:** Node 3



Make node 4 point to node 2.

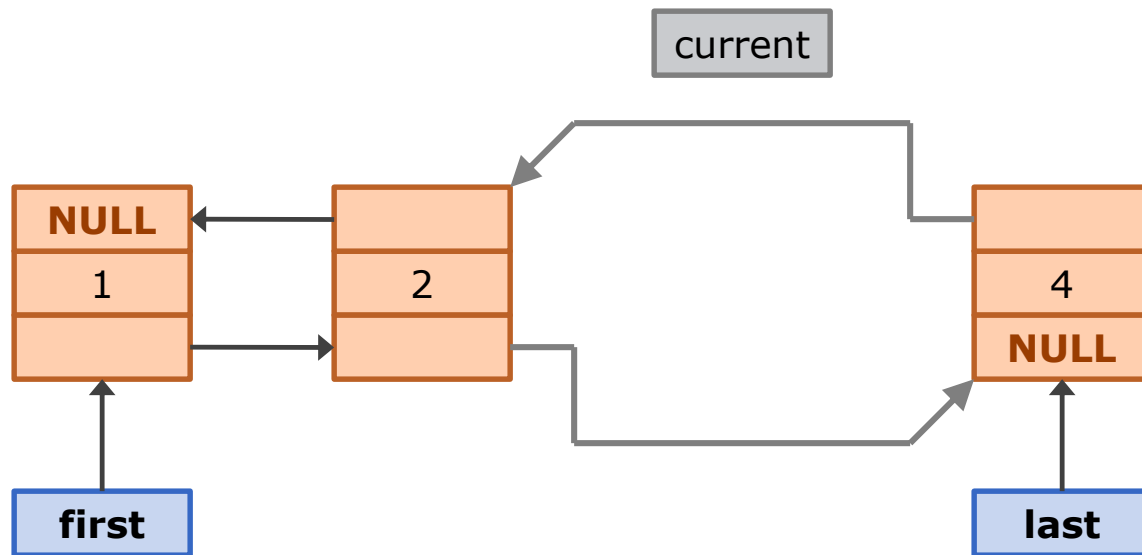
DELETING A NODE FROM THE LIST (CONT.)

- **Case:** List contains more than one node.
- **Delete:** Node 3



DELETING A NODE FROM THE LIST (CONT.)

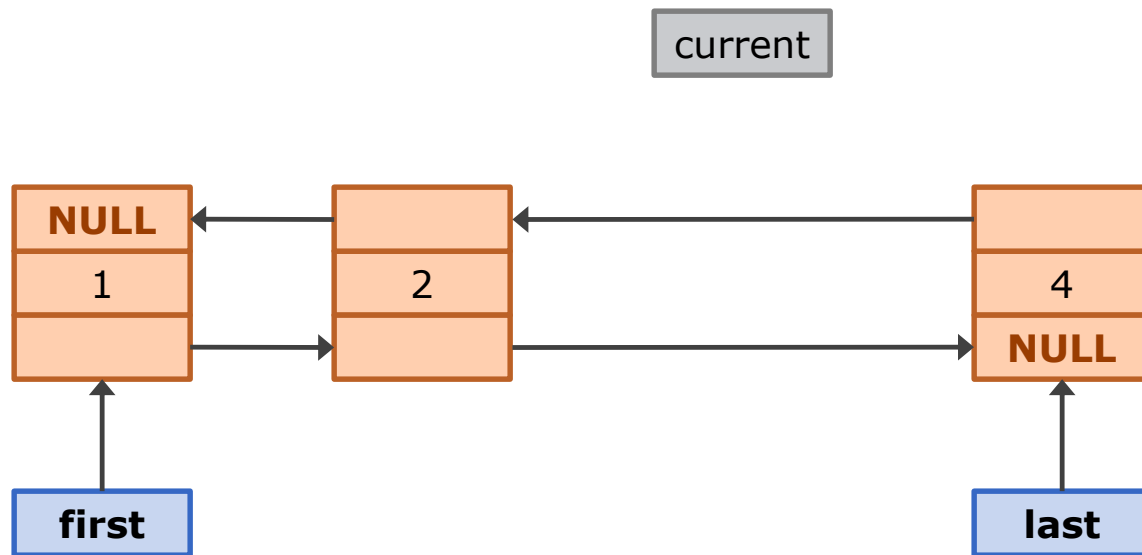
- **Case:** List contains more than one node.
- **Delete:** Node 3



Set **current** to NULL.

DELETING A NODE FROM THE LIST (CONT.)

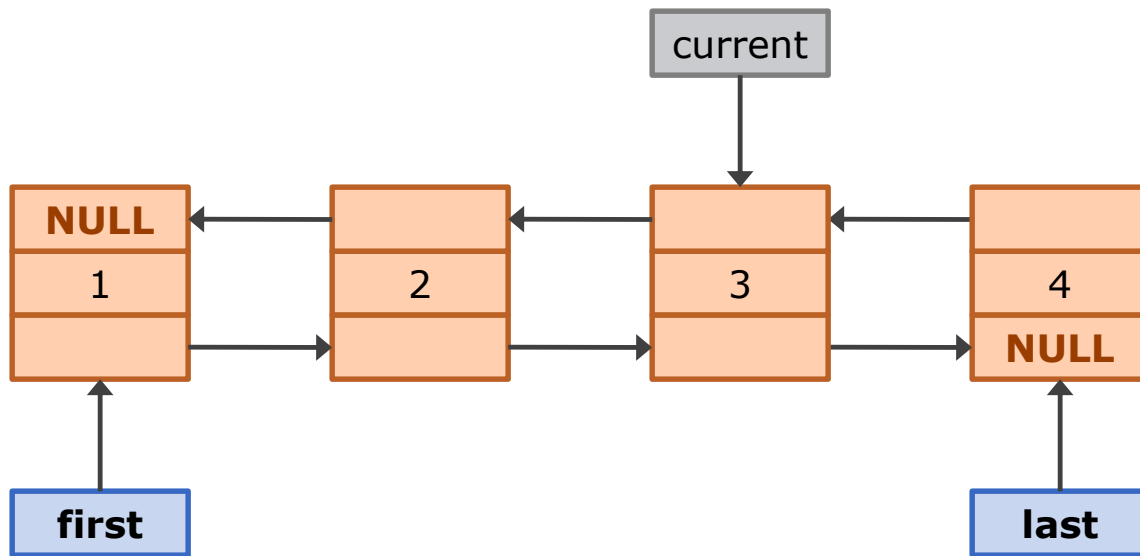
- **Case:** List contains more than one node.
- **Delete:** Node 3



Node 3 has been deleted.

DELETING A NODE FROM THE LIST (CONT.)

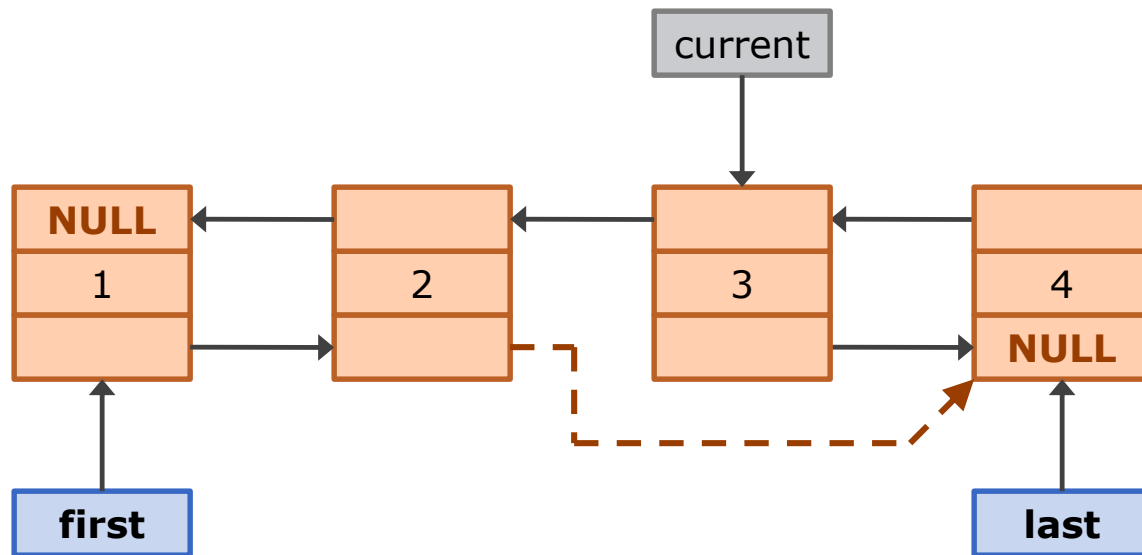
- **Case:** List contains more than one node.
- **Delete:** Node 3



Let's *go back* and look at the syntax needed when using only one pointer **current**.

DELETING A NODE FROM THE LIST (CONT.)

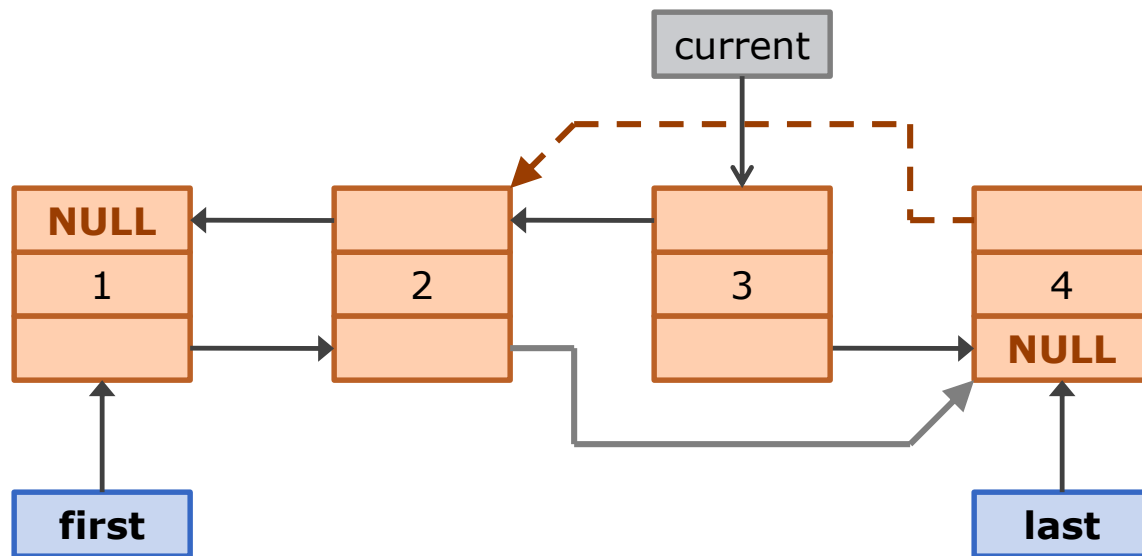
- **Case:** List contains more than one node.
- **Delete:** Node 3



```
current->getPrev() -> setNext(current->getNext());  
      ( node 2 )                ( node 4 )
```

DELETING A NODE FROM THE LIST (CONT.)

- **Case:** List contains more than one node.
- **Delete:** Node 3



```
current->getNext()->setPrev(current->getPrev());  
      ( node 4 )                ( node 2 )
```

A LIST AS AN ADT

- A list as an **Abstract Data Type (ADT)** is a *generic* definition of a list
 - It has basic operations to manipulate the list
- Implementation is not relevant
 - User needs to know only basic operations
 - A **List ADT** can be implemented as
 - An array
 - A linked list (singly, doubly)
 - A vector
- Can be **sorted** or **unsorted**

BASIC OPERATIONS OF A LIST ADT

- Whether you are implementing the list as an **array** or a **linked list**, basic operations are necessary
 - **Default constructor**
 - Initialize the list to an empty state
 - **Empty the list**
 - Re-initializes a list to an empty state
 - **Insert**
 - Inserts an element in the list
 - Can be in a particular order
 - **Get number of elements**
 - Returns the number of elements in the list

BASIC OPERATIONS OF A LIST ADT (CONT.)

- **Get first element**
 - Returns the first element in the list
- **Get last element**
 - Returns the last element in the list
- **Search list**
 - Searches the whole list for a given element
 - Returns a **boolean** value
- **Delete element**
 - Need to consider cases:
 - List is empty
 - The element is not in the list
- **Copy list**
 - Makes an identical copy of a list
- **Destructor**
 - If list is **dynamic**, deallocates list from memory

EXAMPLE

- Project: Doubly-linked List



DOUBLY-LINKED LIST (END)