CIS 190: C/C++ Programming

Linked Lists

Why Use Linked Lists?

- solve many of the problems arrays have
- like…

Problems with Arrays

- arrays have a fixed size
  - may be too large, or too small
- arrays must be held in contiguous space
  - may have the room, but not contiguously
  - can cause “dead” space in memory
- arrays are difficult to “break” or edit
  - add one element in the middle
  - remove one element from the middle w/o a gap
  - break into multiple arrays

Solutions through Linked Lists

- arrays have a fixed size
  - linked lists can change size constantly
- arrays must be held in contiguous space
  - may have the room, but not contiguously
  - can cause “dead” space in memory
- arrays are difficult to “break” or edit
  - add one element in the middle
  - remove one element from the middle w/o a gap
  - break into multiple arrays

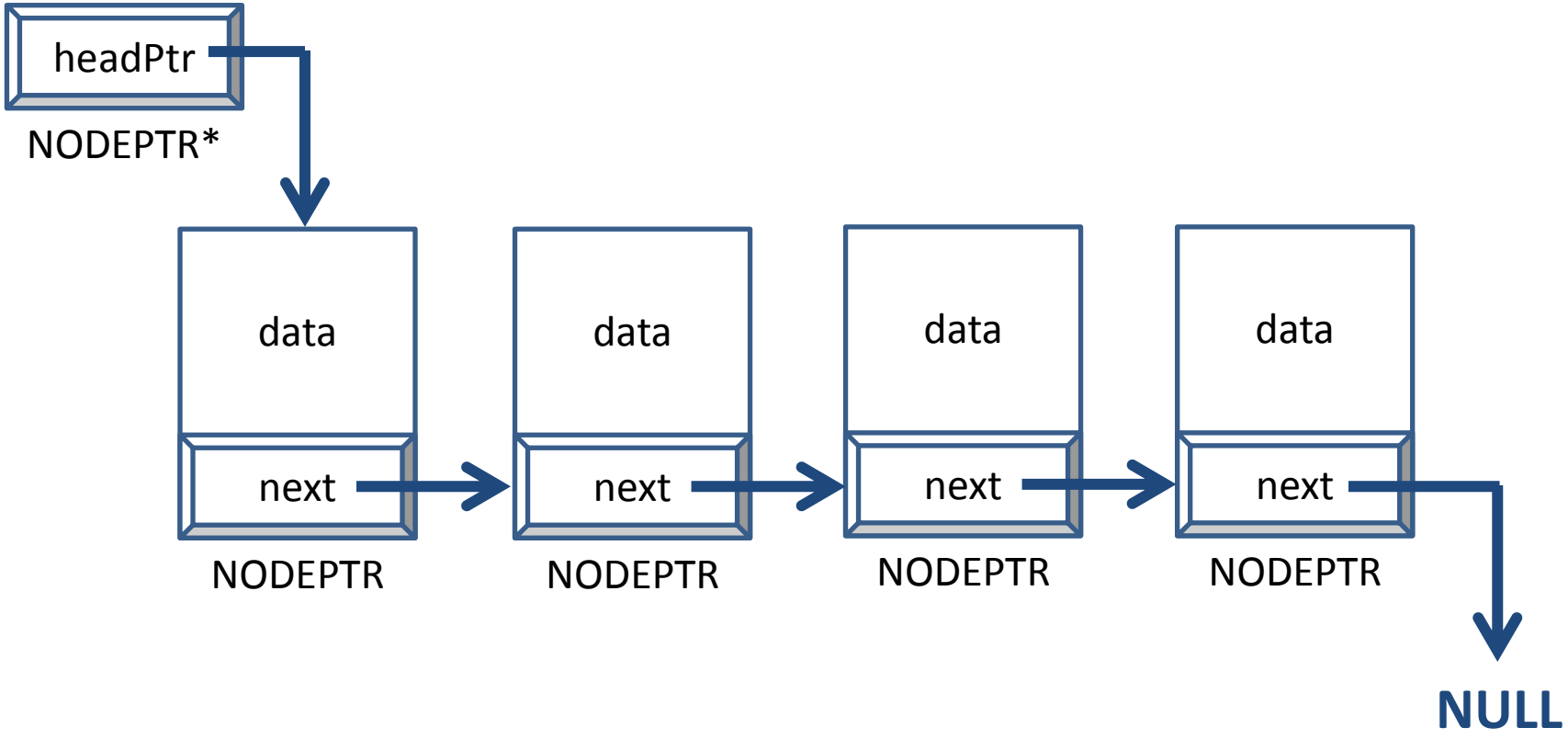
Solutions through Linked Lists

- arrays have a fixed size
  - linked lists can change size constantly
- arrays must be held in contiguous space
  - only one **node** must be held in contiguous space
  - linked list may be stored in many disparate places
- arrays are difficult to “break” or edit
  - add one element in the middle
  - remove one element from the middle w/o a gap
  - break into multiple arrays

Solutions through Linked Lists

- arrays have a fixed size
  - linked lists can change size constantly
- arrays must be held in contiguous space
  - only one **node** must be held in contiguous space
  - linked list may be stored in many disparate places
- arrays are difficult to “break” or edit
  - can add nodes anywhere in a linked list
  - remove elements with no gaps at all
  - concatenation and separation are feasible

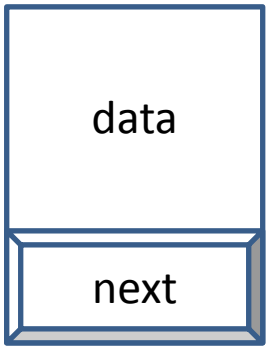
Linked Lists



Nodes

- a “node” is one element of a linked list

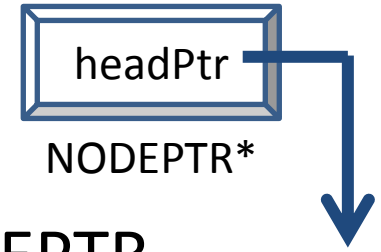
- nodes consist of two parts:
  - data stored in node
  - pointer to next node in list



- typically represented as structs

headPtr

- headPtr is not the first node in the list



- headPtr is of type NODEPTR*
  - it is a pointer to a variable of type NODEPTR
- headPtr being NULL means the list is empty

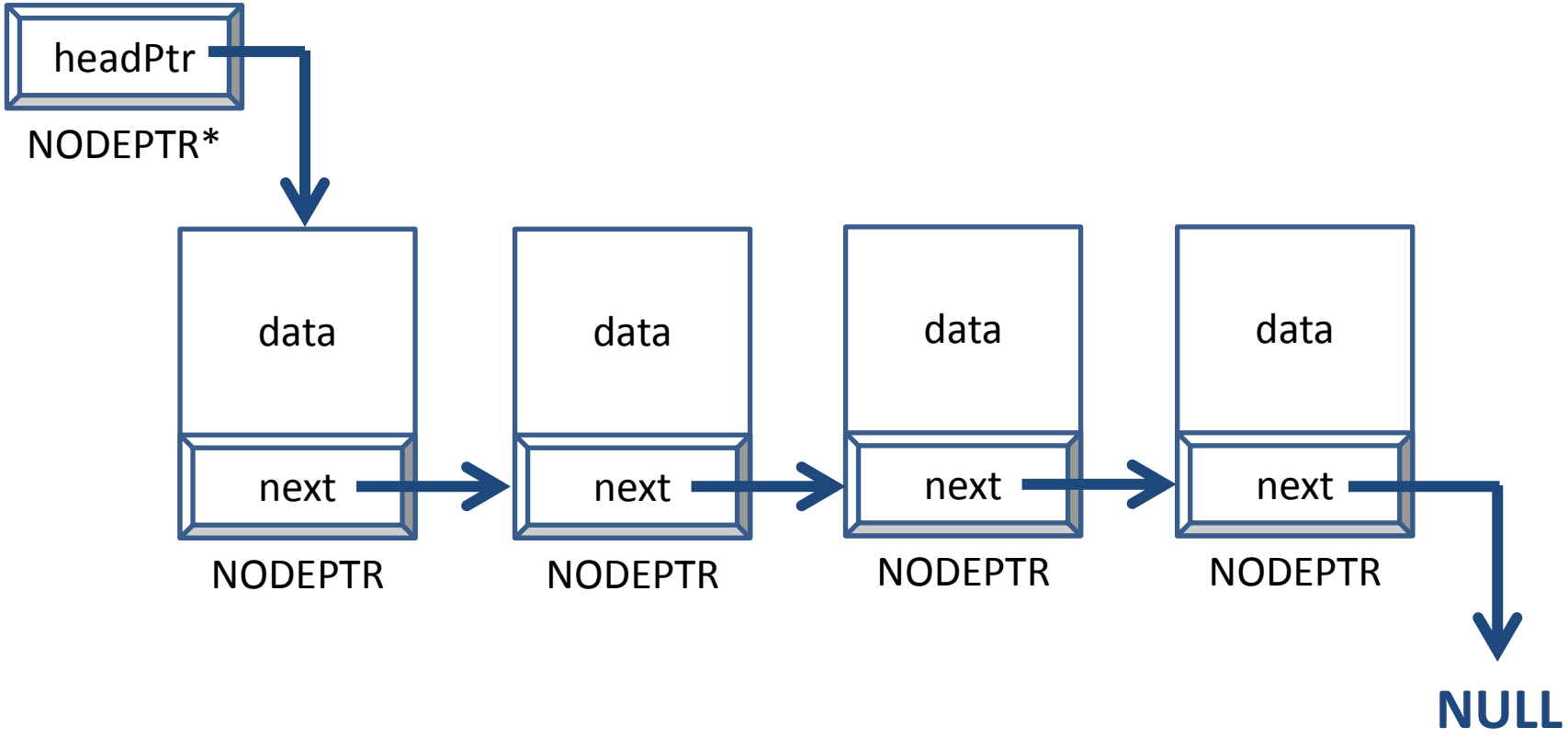
- convention inside functions:
  - **`NODEPTR* headPtr`** = pointer to a NODEPTR
  - **`NODEPTR head`** = address of first node

Node Definition

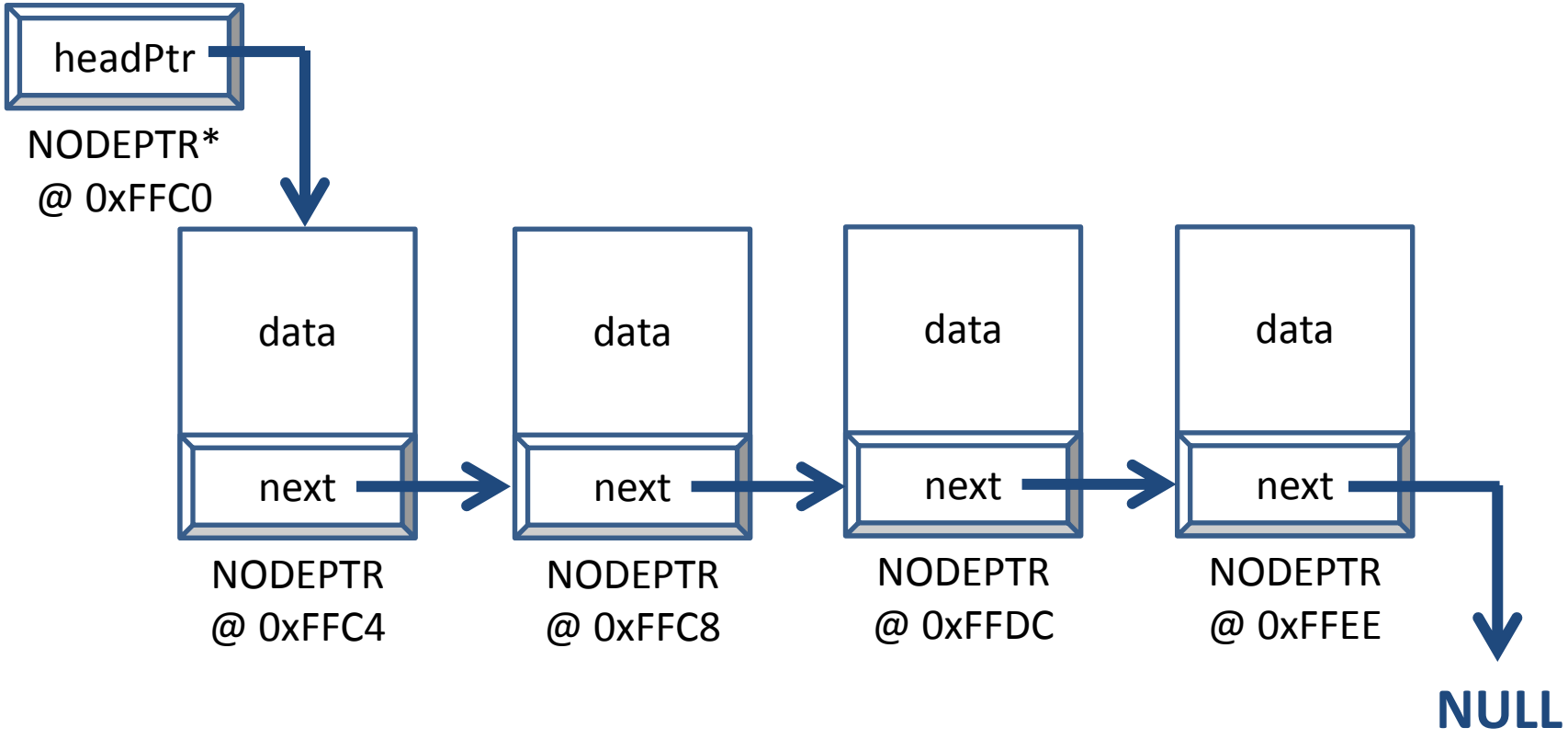
```
typedef struct node * NODEPTR;

typedef struct node {
    int     data;
    NODEPTR next;
} NODE;
```

- typedef NODEPTR beforehand so that it can be used in the definition of the NODE structure

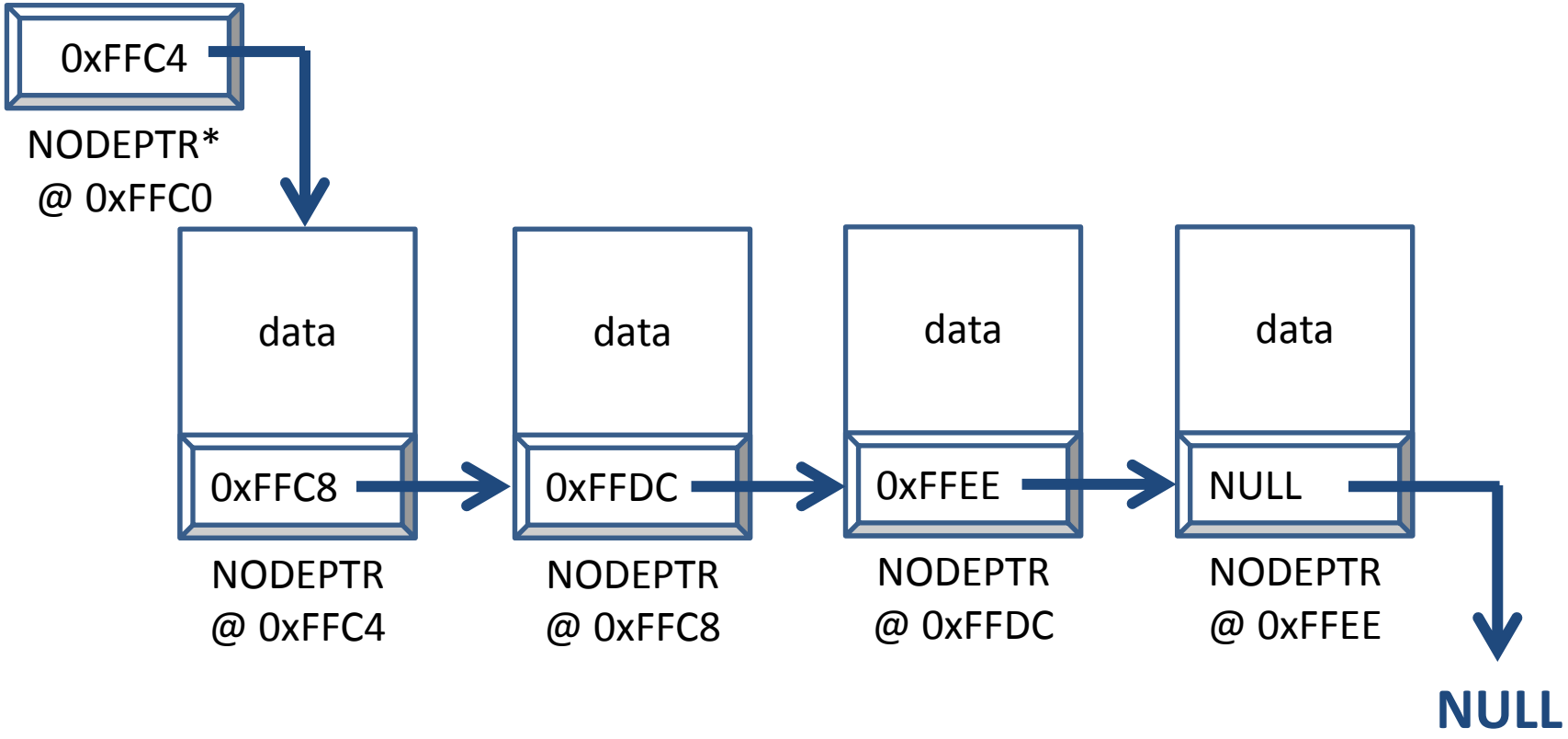
Linked Lists



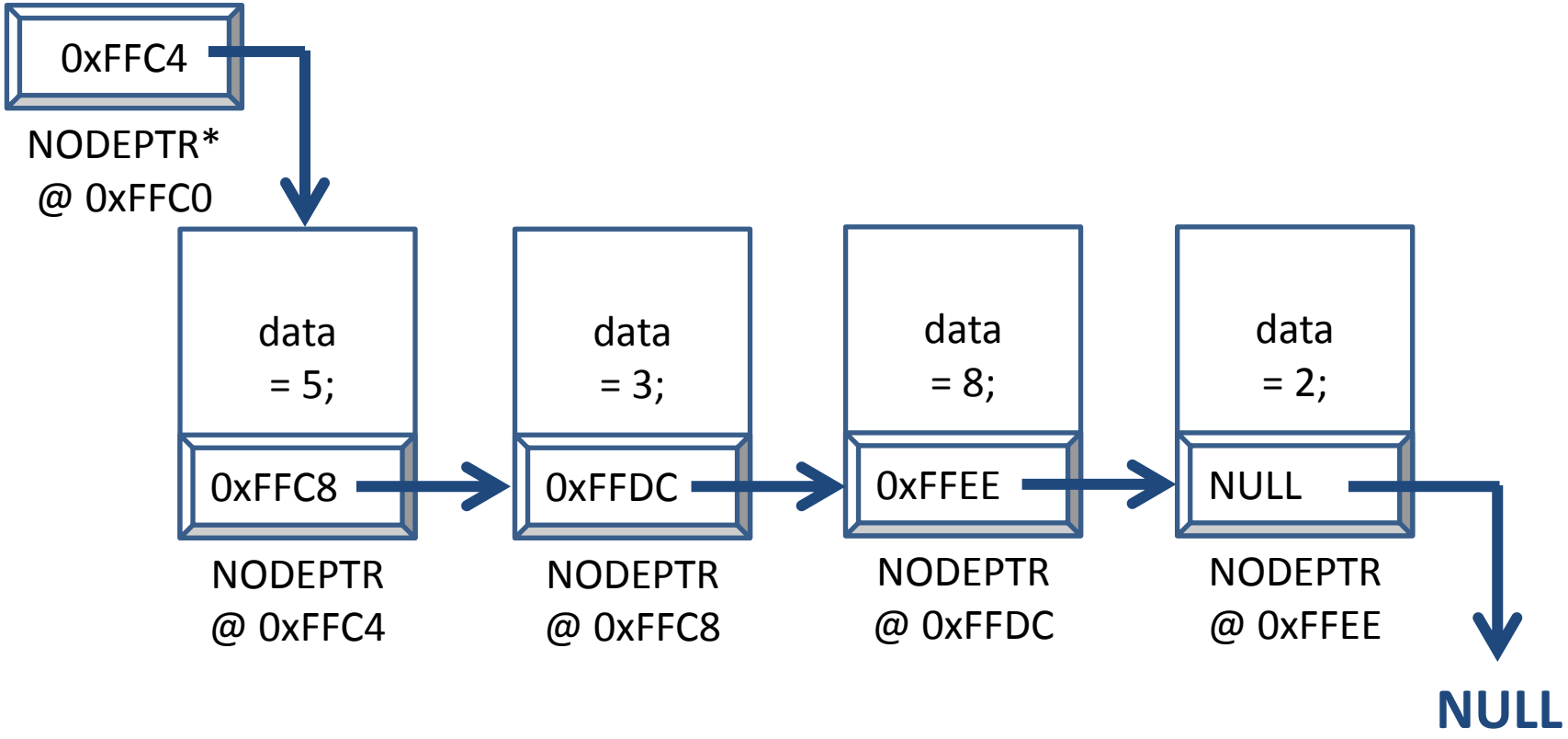
Linked Lists



Linked Lists



Linked Lists



Linked List Operations

- create a new node
- assign values to the data in a node
- print the entire linked list
  - in a readable format
- insert a node
  - at the end of the list
  - somewhere else: middle of list, beginning, etc.
- delete a node

Creating a Node

```
NODEPTR CreateNode (void)
```

1. create and allocate memory for a node

```
newNode = (NODEPTR) malloc (sizeof(NODE));
```

2. ensure that memory was allocated
3. initialize data

Creating a Node

```
NODEPTR CreateNode (void)
```

1. create and allocate memory for a node

```
newNode = (NODEPTR) malloc (sizeof(NODE));
```

– cast as NODEPTR, but space for NODE – why?

2. ensure that memory was allocated
3. initialize data

Setting a Node's Data

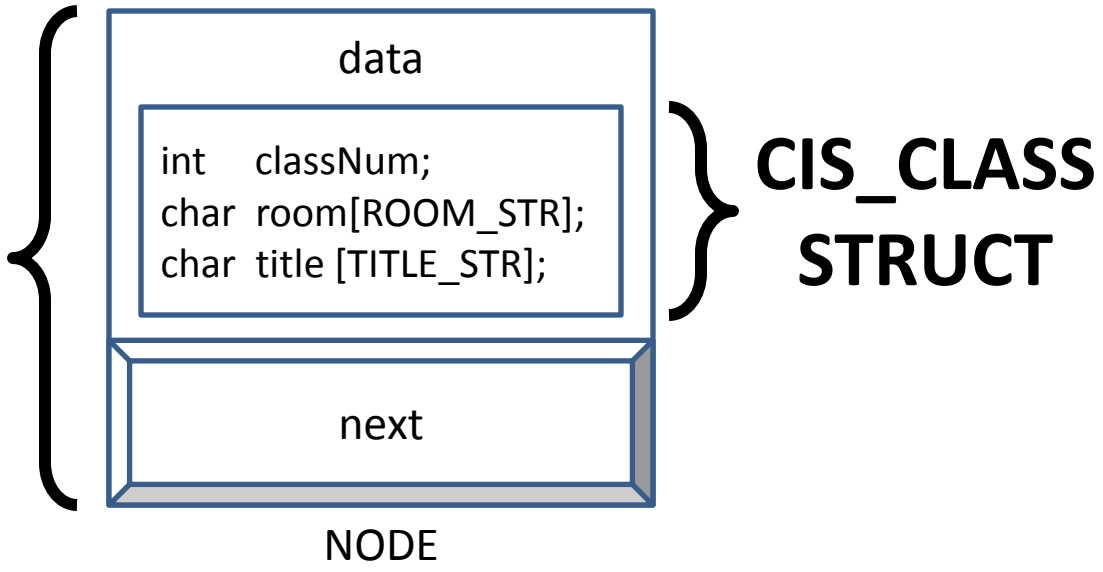
```
void SetData (NODEPTR temp,
              int data)
```

- NODEPTR is a pointer, but it points to a struct
  - use arrow notation to access elements
    - or dot star notation

```
temp->data = data;
(*temp).data = data;
```

When “data” is a Struct

```
typedef struct node {
     CIS_CLASS class;
     NODEPTR   next;
} NODE;
```



Setting Data when “data” is a Struct

```
void SetData (NODEPTR temp, int classNum,
              char room  [ROOM_STR],
              char title [TITLE_STR])

temp->class.classNum = classNum;
strcpy(temp->class.room,  room);
strcpy(temp->class.title, title);
```

- `class` struct is not a pointer, so we simply use dot notation

Traversing a Linked List

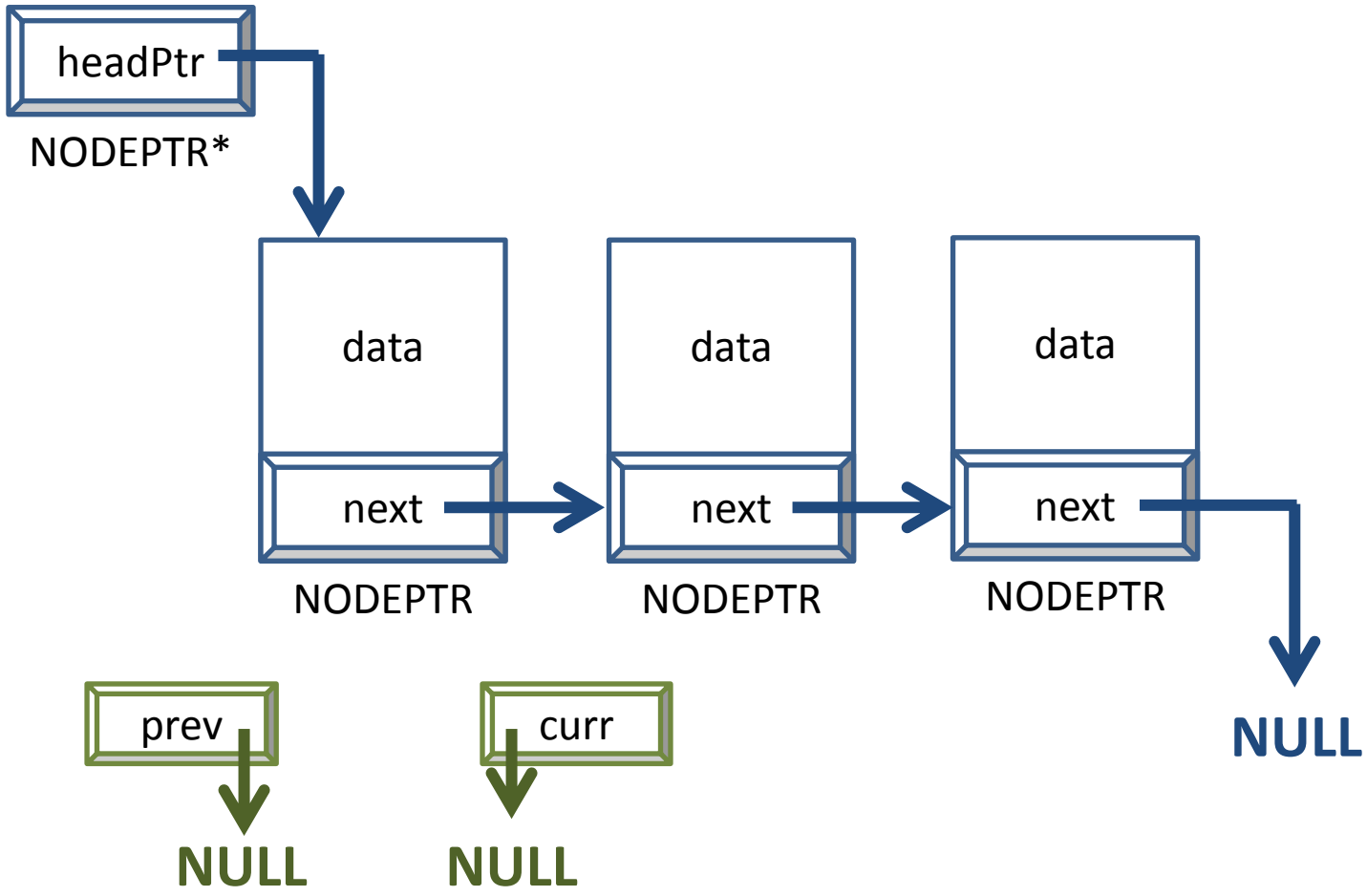
- used for many linked list operations
- check to see if list is empty
- use two temporary pointers to keep track of current and previous nodes (**`prev`** and **`curr`**)
- move through list, setting **`prev`** to **`curr`** and **`curr`** to the next element of the list
  - continue until you hit the end (or conditions met)

Special Cases with Linked Lists

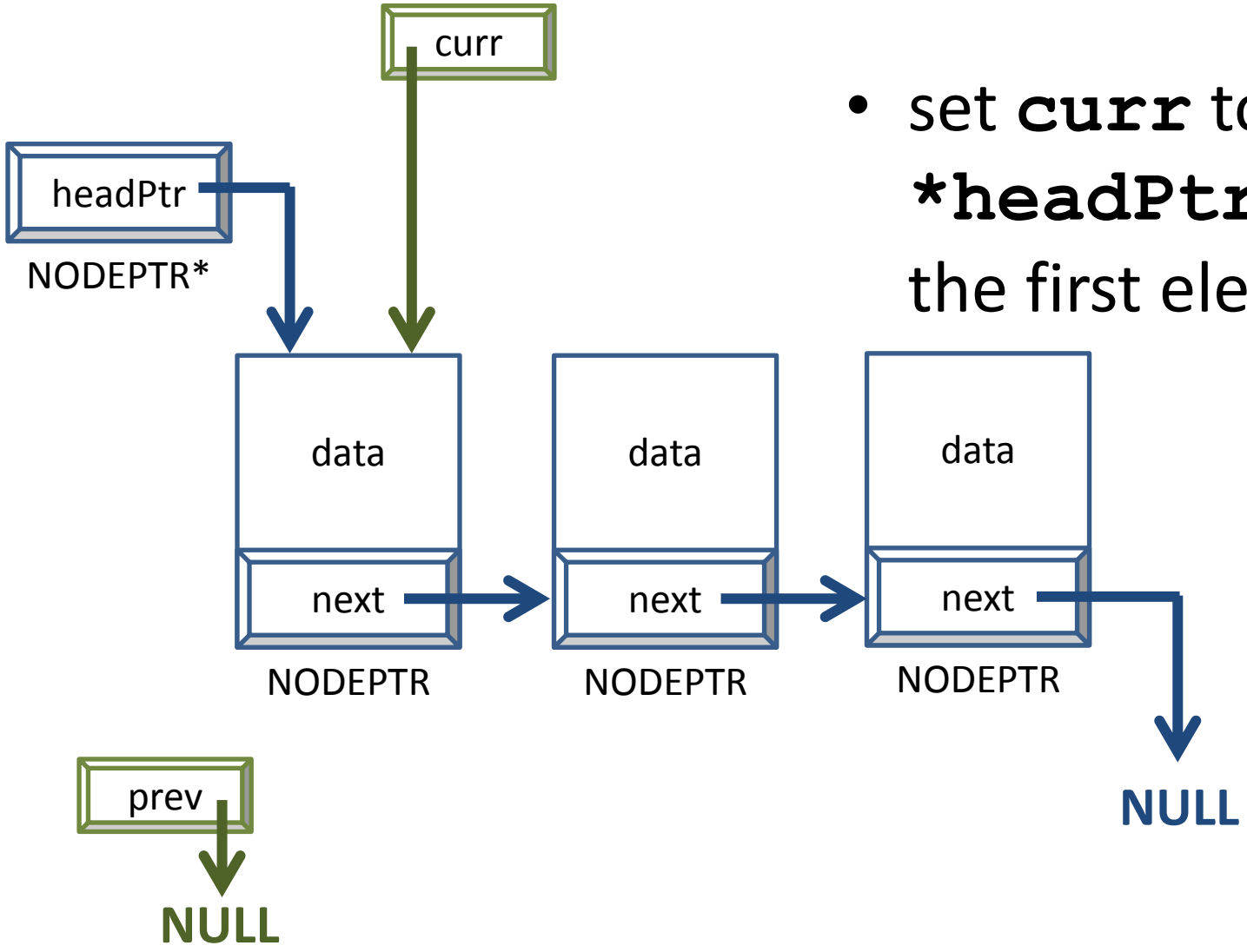
- always a separate rule when dealing with the first element in the list (where headPtr points)
  - and a separate rule for when the list is empty

- laid out in the code available online, but keep it in mind whenever working with linked lists
  - make sure you understand the code before you start using it in your programs

Traversing a Linked List – Step by Step

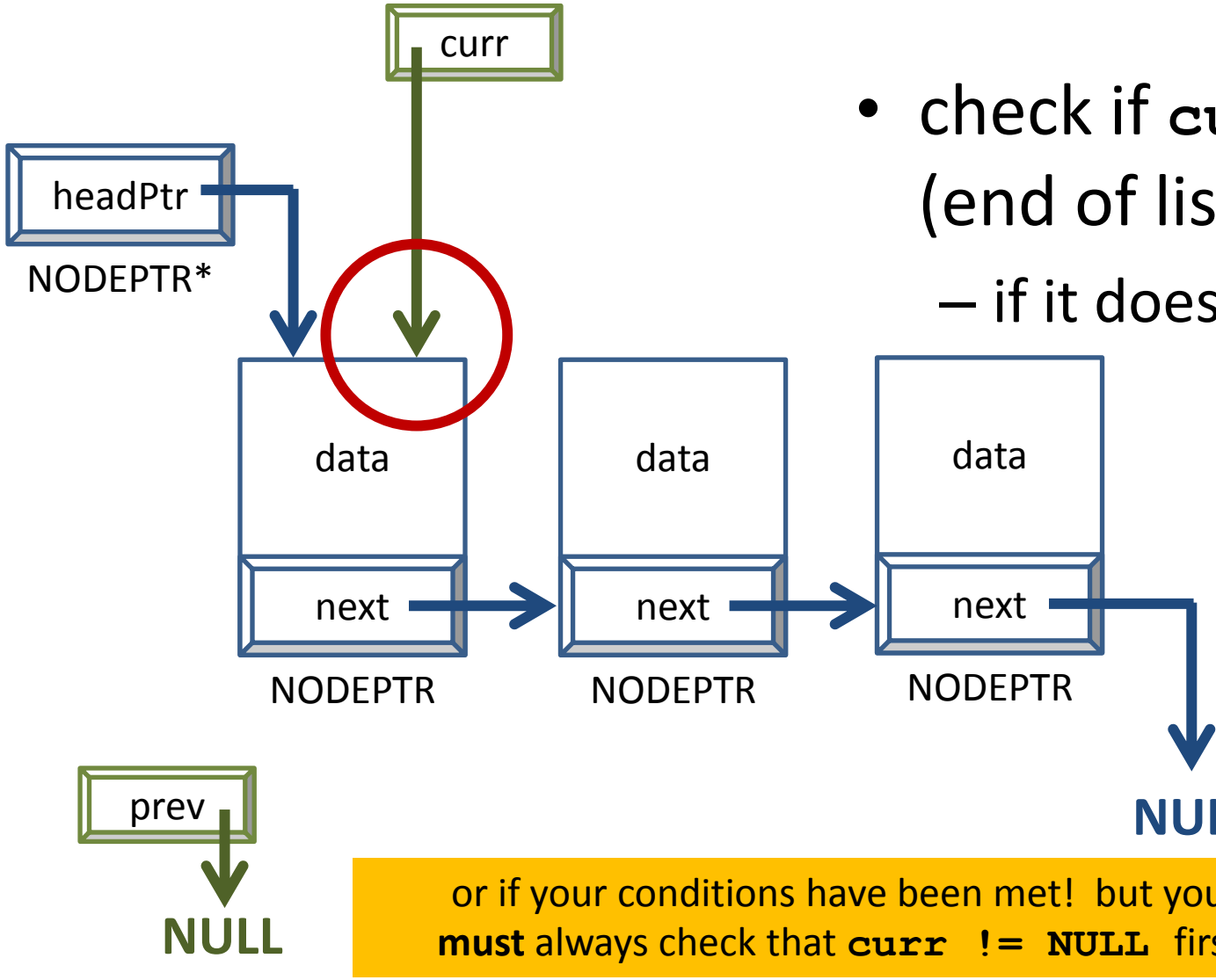


Traversing a Linked List – Step 1



- set **`curr`** to **`*headPtr`**, the first element

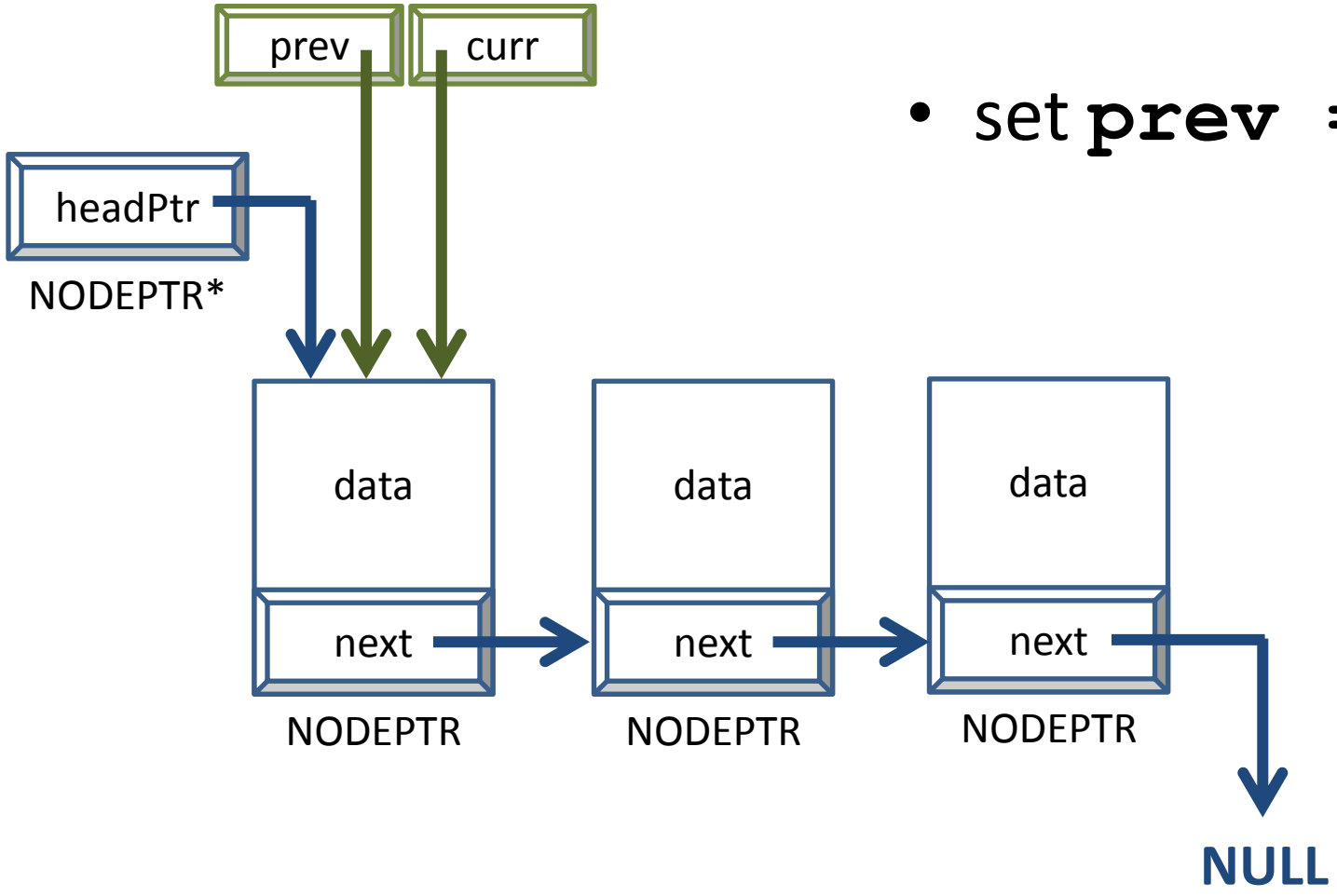
Traversing a Linked List – Step 2



- check if **`curr == NULL`** (end of list)
  - if it doesn't, continue

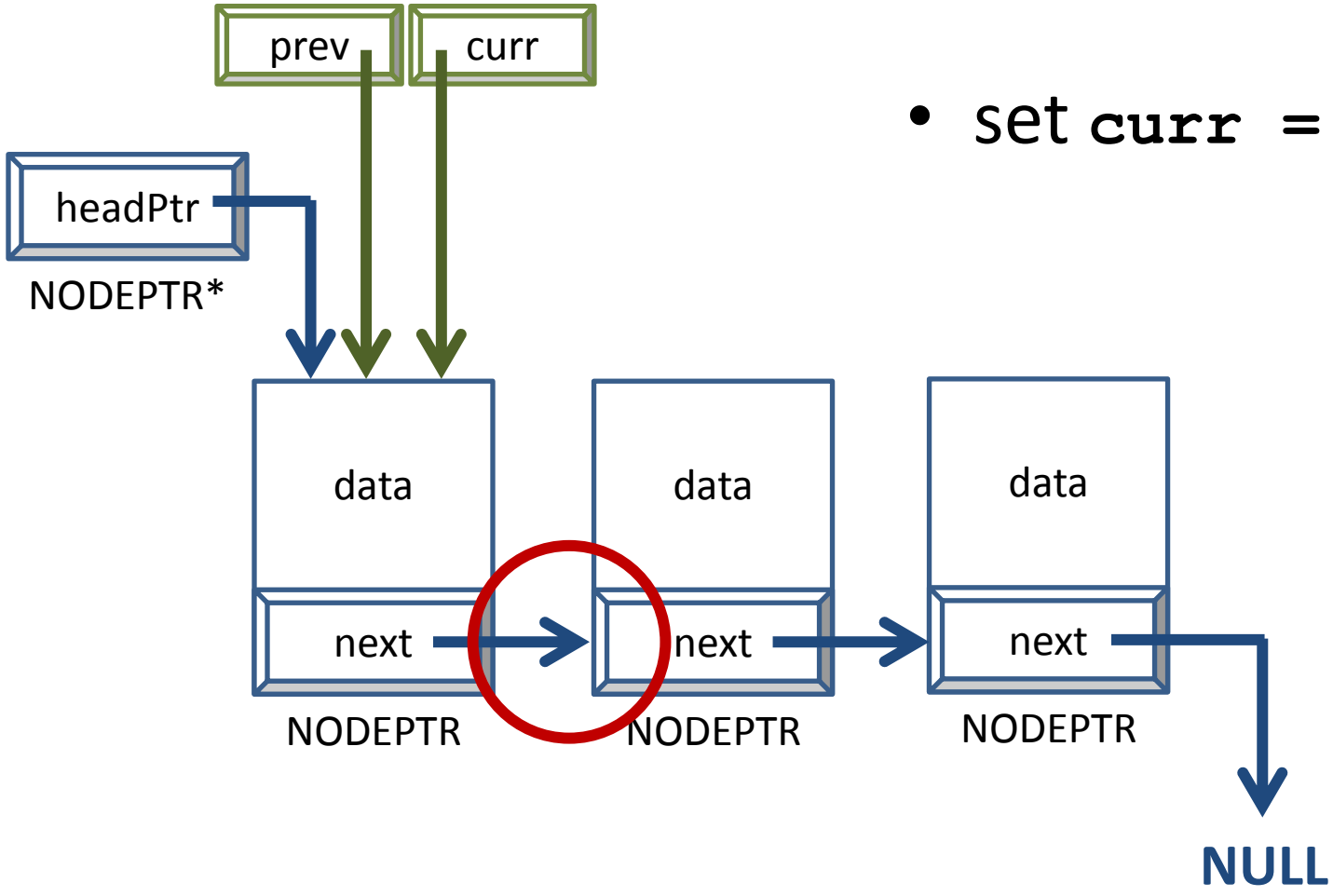
or if your conditions have been met! but you **must** always check that **`curr != NULL`** first

Traversing a Linked List – Step 3



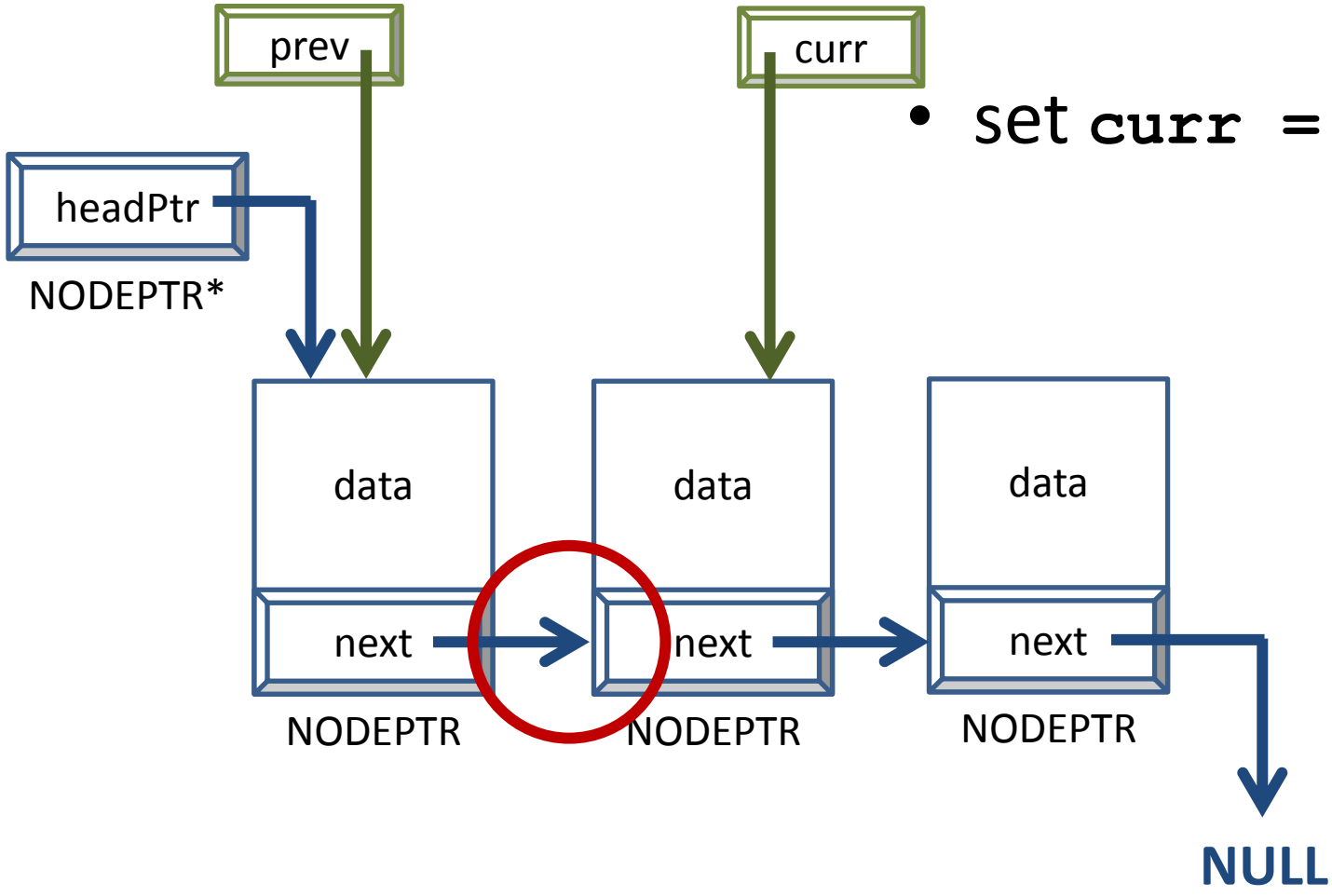
- set **`prev = curr`**

Traversing a Linked List – Step 4



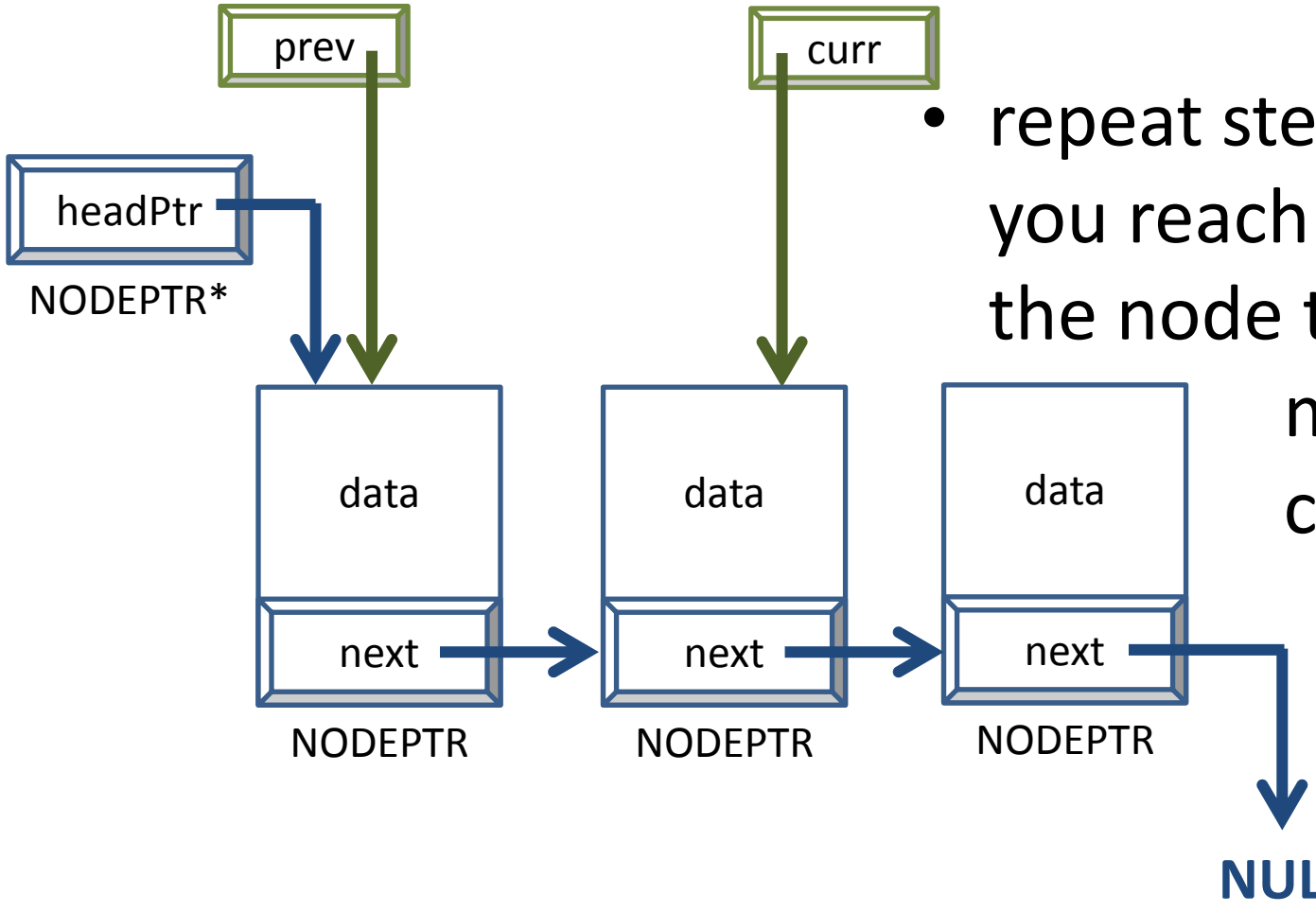
- set `curr = curr->next`

Traversing a Linked List – Step 4



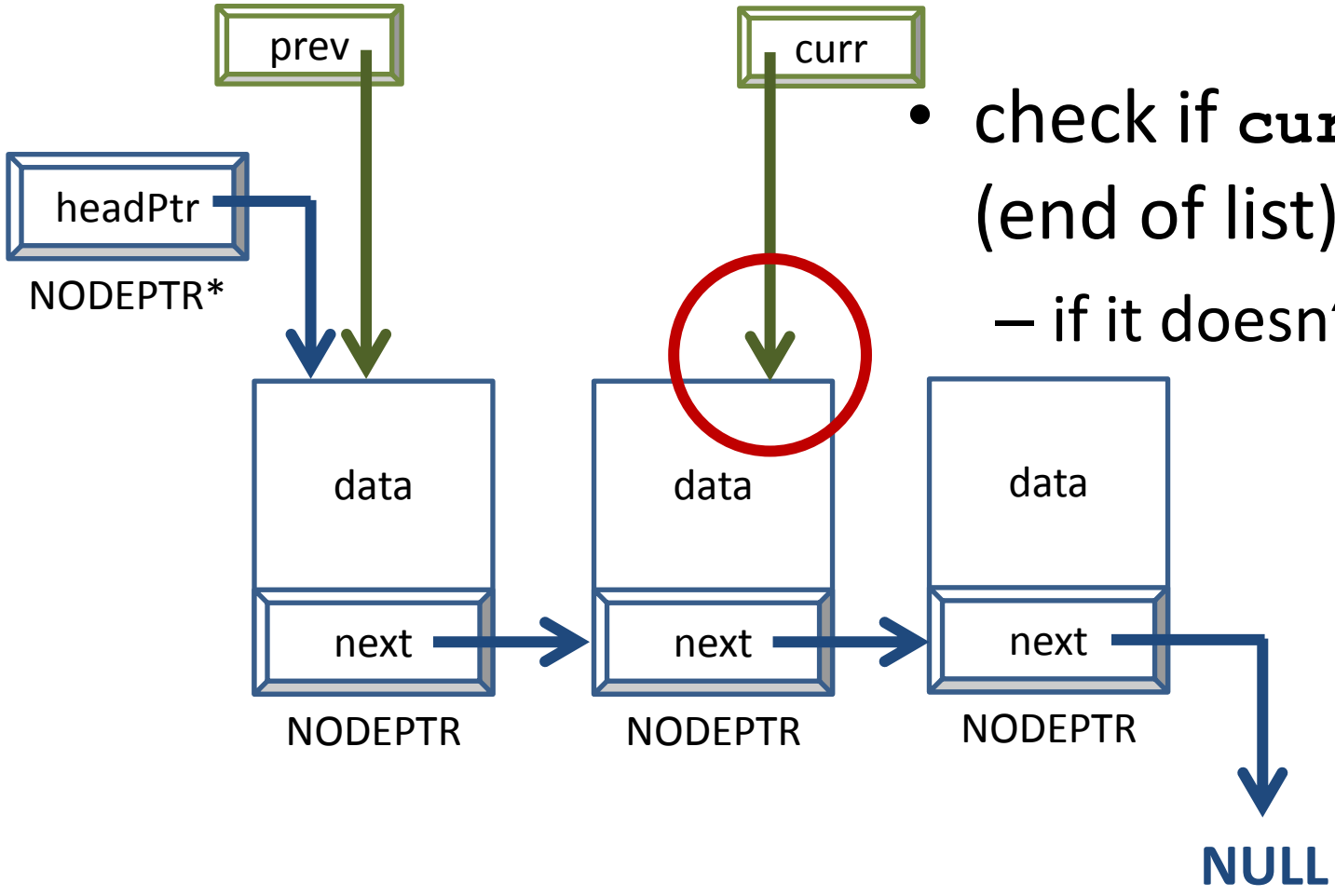
- set `curr = curr->next`

Traversing a Linked List – Step 5



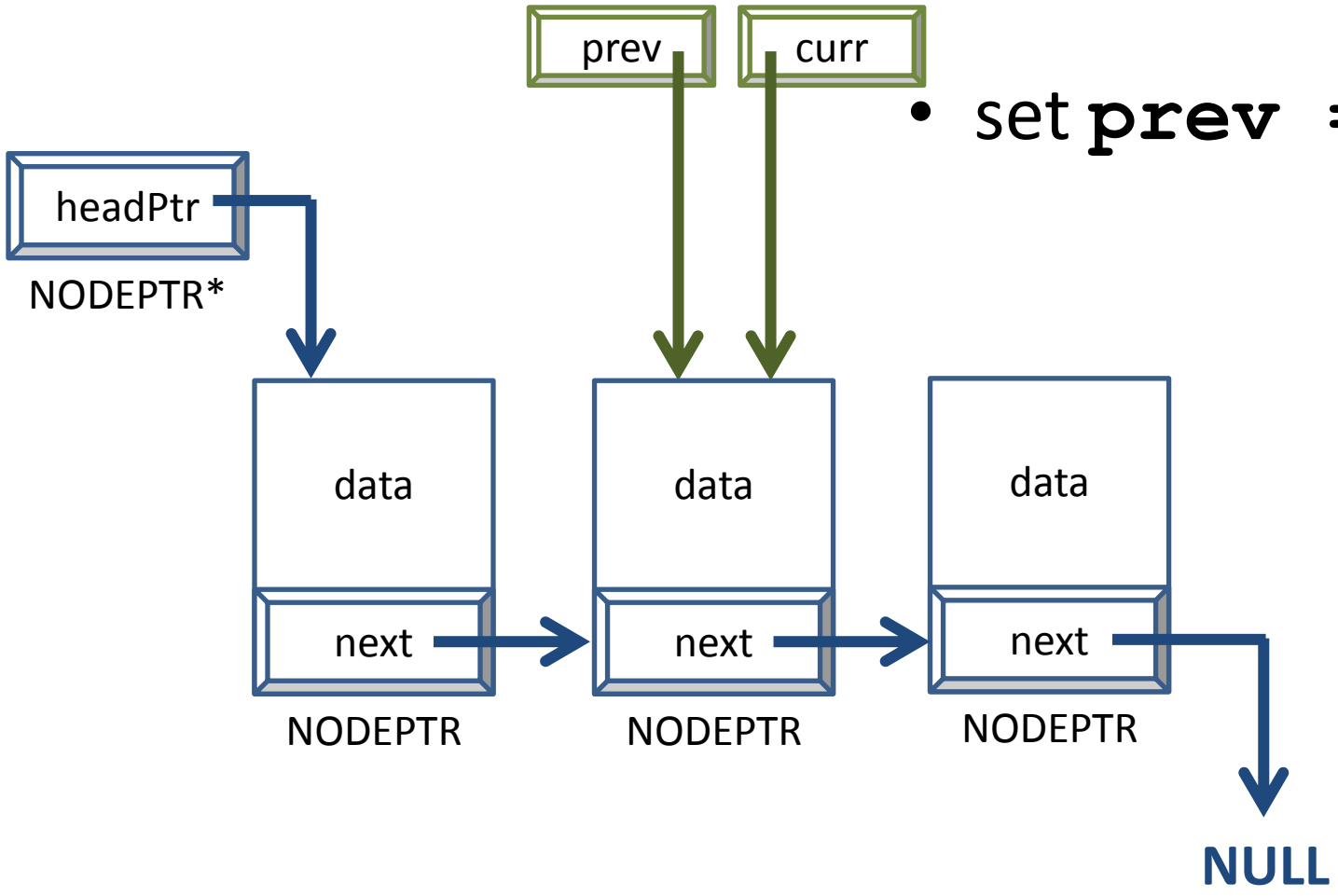
- repeat steps 2-4 until you reach the end (or the node that matches the conditions you are looking for)

Traversing a Linked List – Step 5…



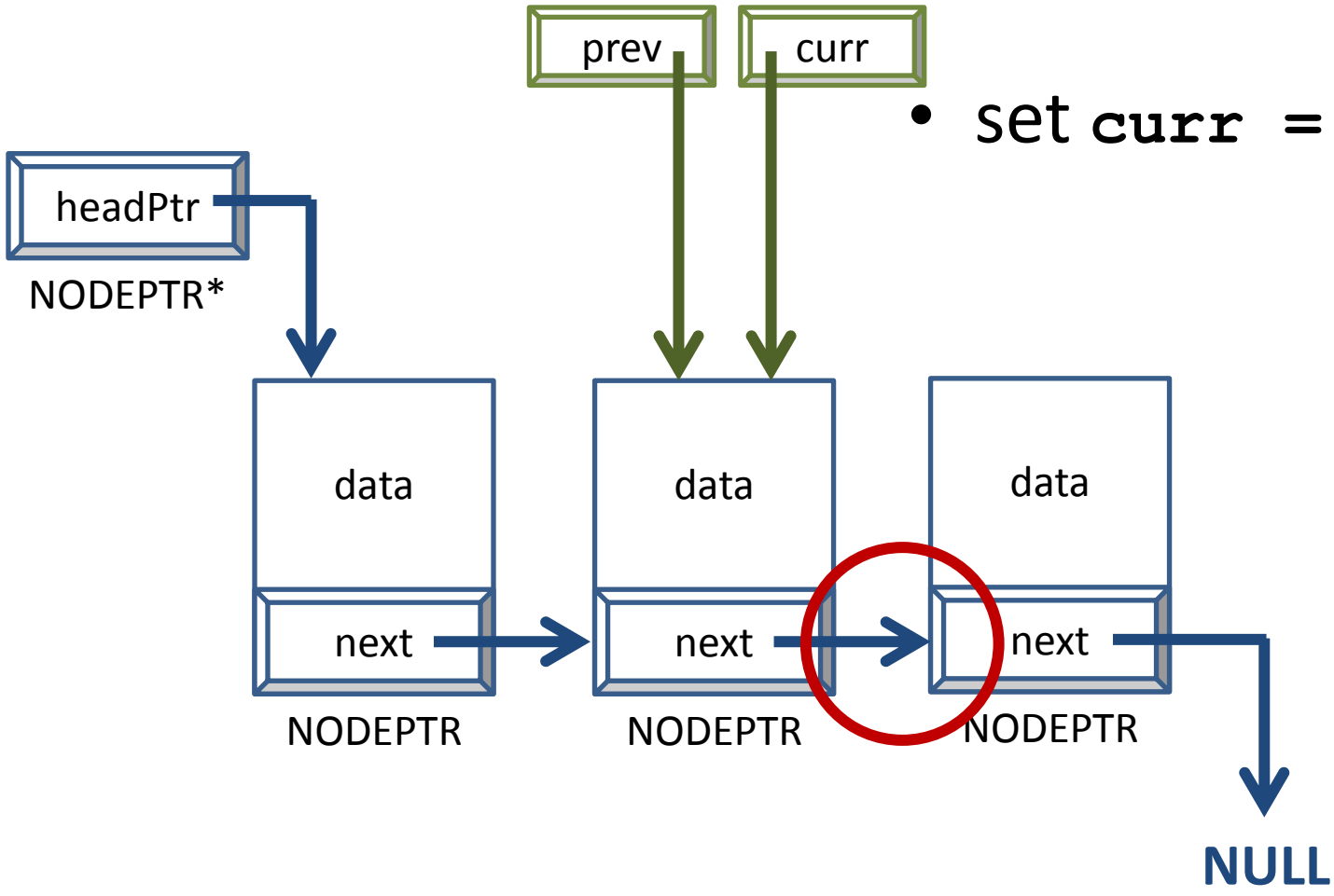
- check if **`curr == NULL`** (end of list)
  - if it doesn't, continue

Traversing a Linked List – Step 5…



- set **`prev = curr`**

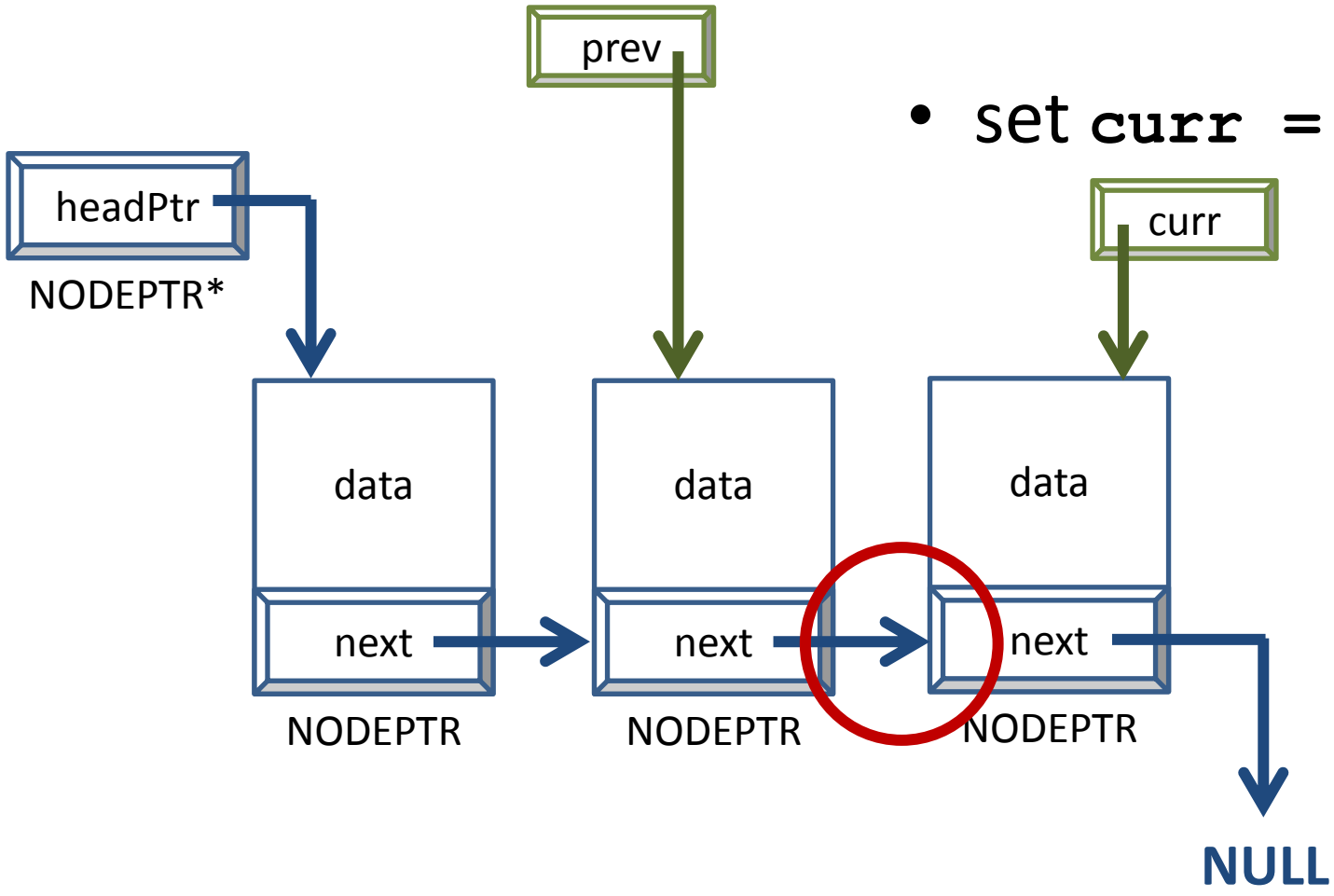
Traversing a Linked List – Step 5…



- set `curr = curr->next`

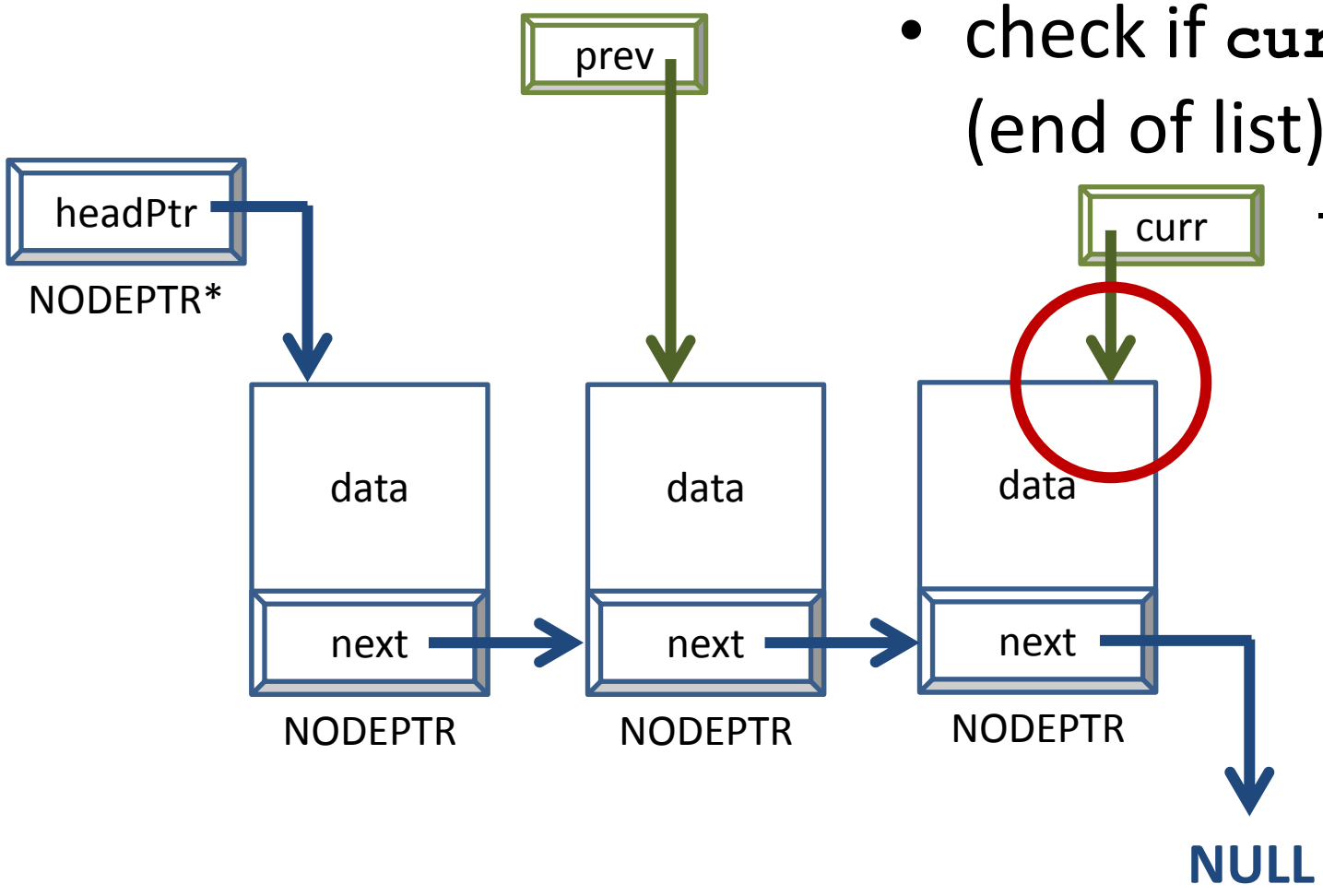
Traversing a Linked List – Step 5…

- set `curr = curr->next`



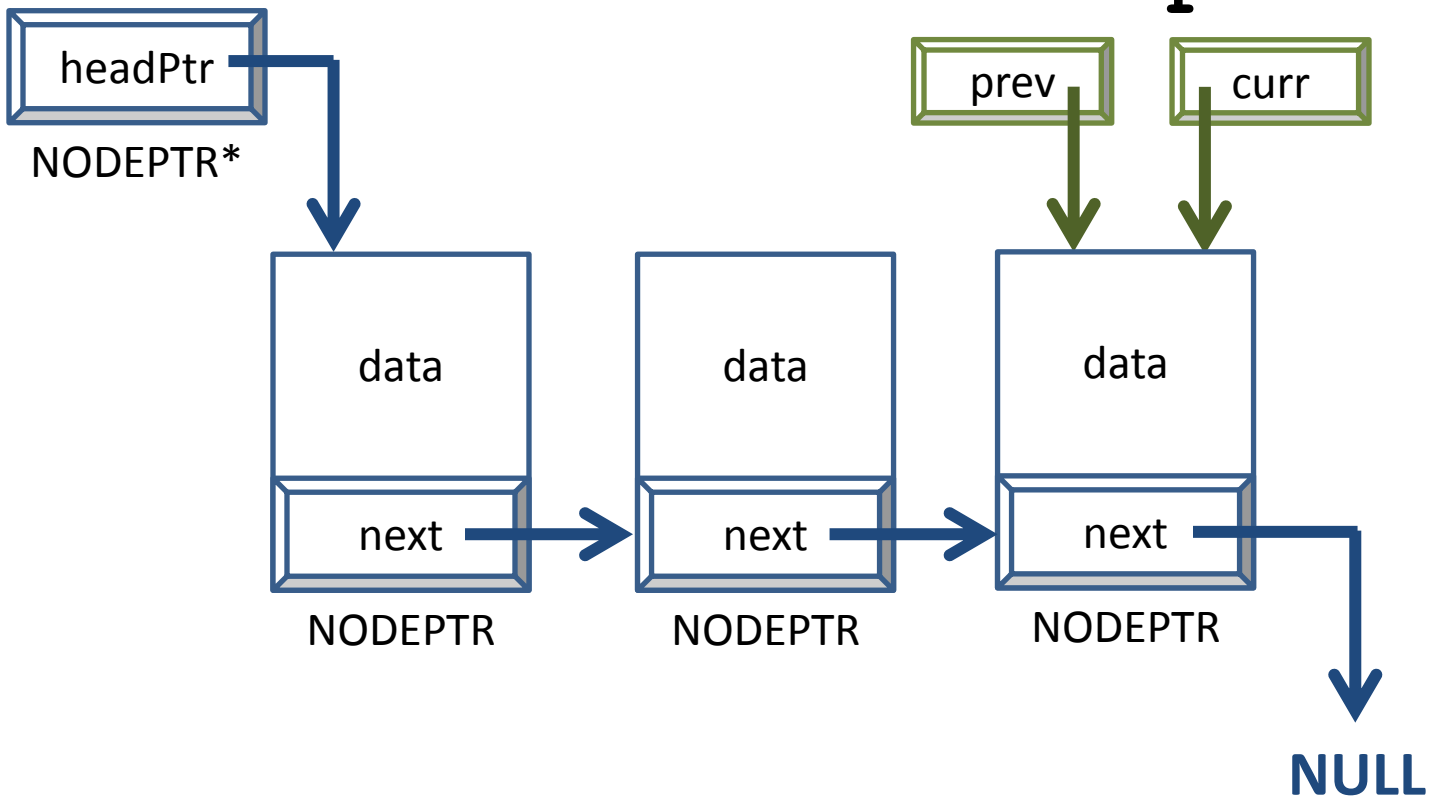
Traversing a Linked List – Step 5…

- check if **`curr == NULL`** (end of list)
  - if it doesn't, continue



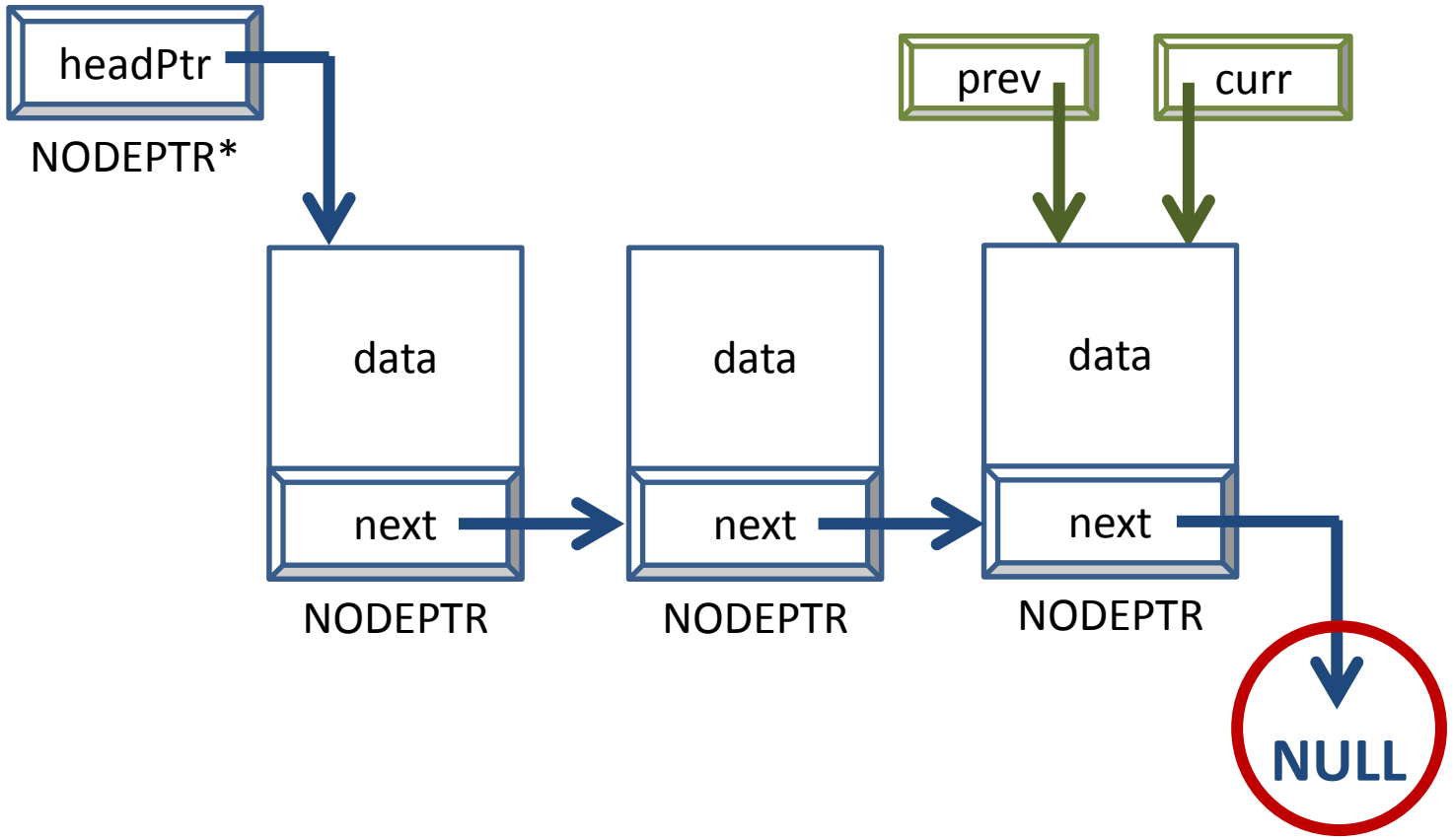
Traversing a Linked List – Step 5…

- set **`prev = curr`**



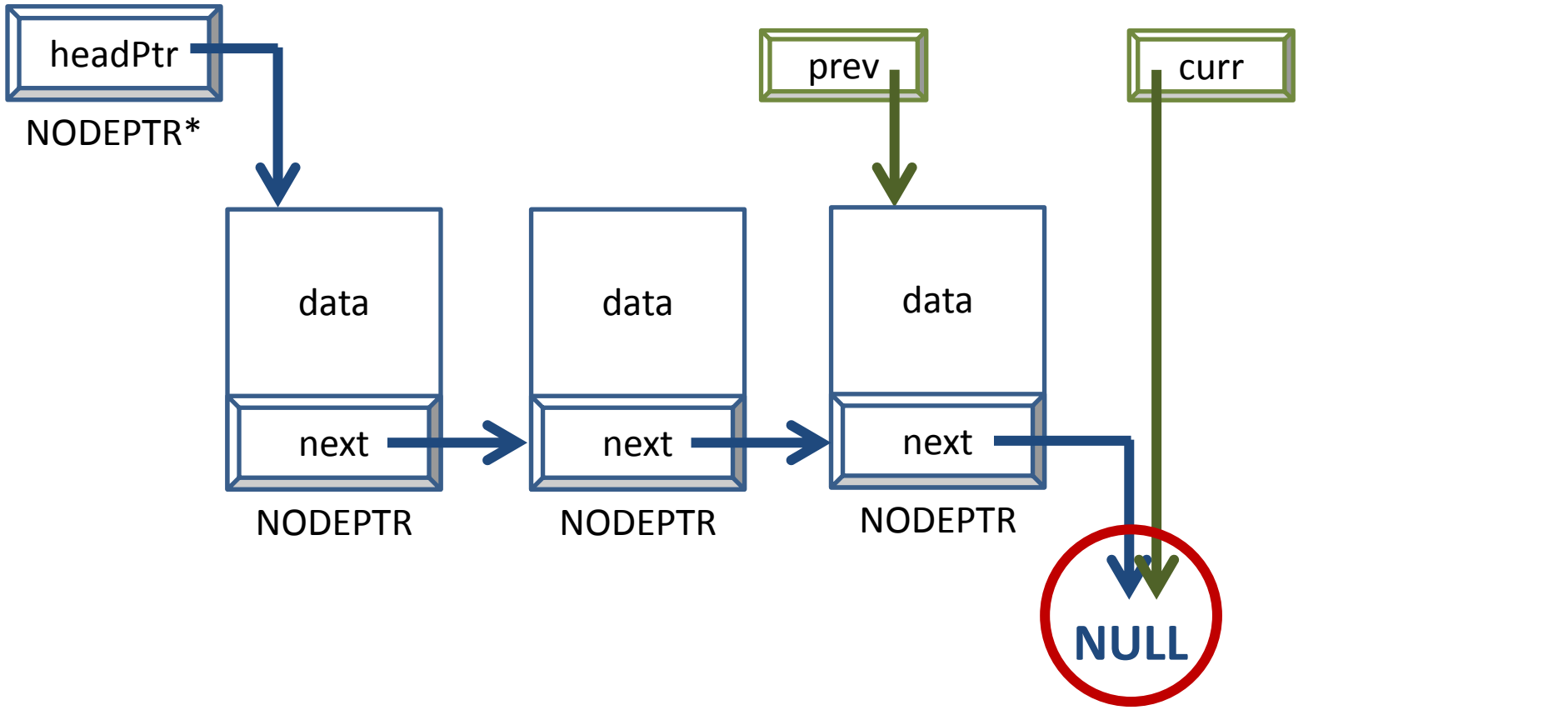
Traversing a Linked List – Step 5…

- set `curr = curr->next`



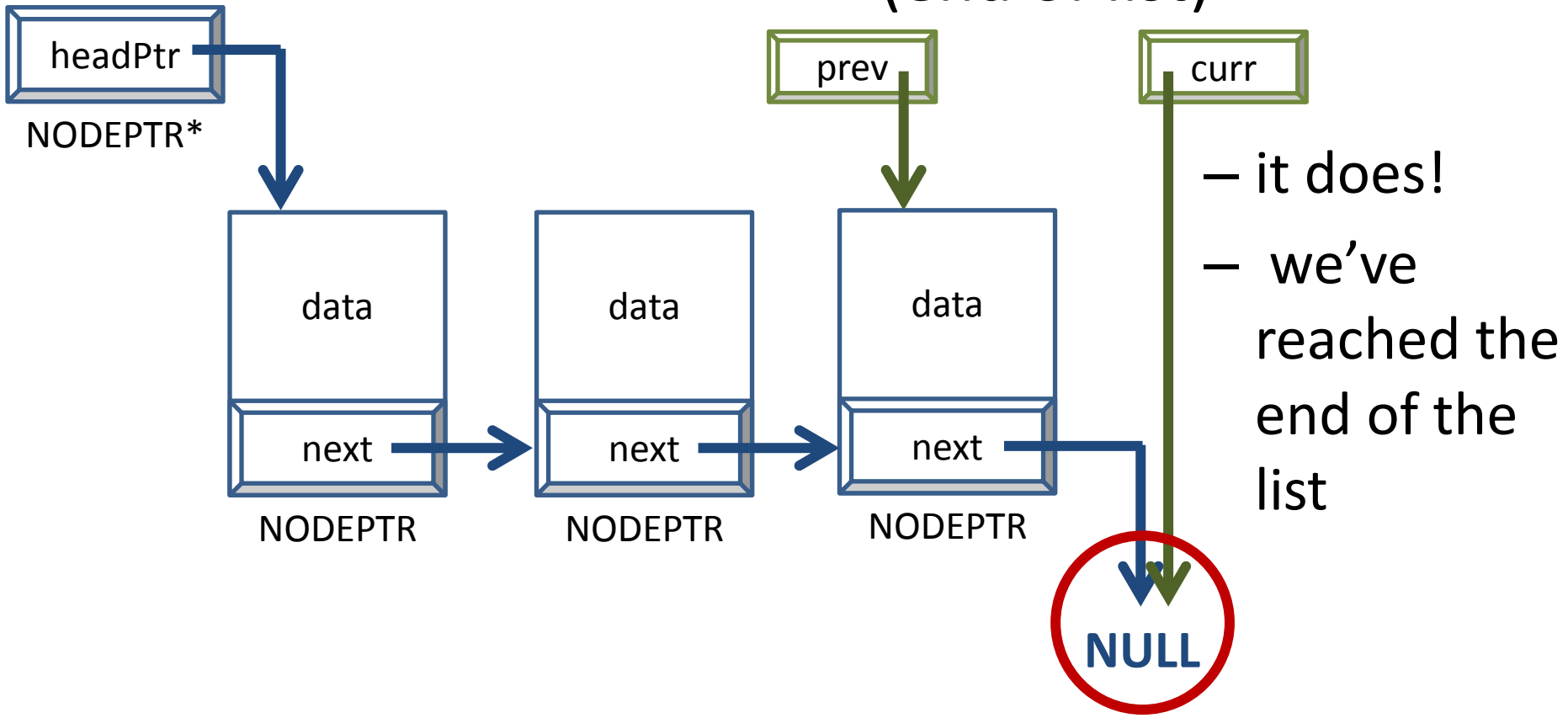
Traversing a Linked List – Step 5…

- set `curr = curr->next`



Traversing a Linked List – Step 5…

- check if **`curr == NULL`** (end of list)



  - it does!
  - we've reached the end of the list

Printing the Entire Linked List

```
void PrintList (NODEPTR head)
```

- check to see if list is empty
  - if so, print out a message
- if not, traverse the linked list
  - print out the data of each node

  - NODEPTR head is pointer to first node

Inserting a Node

```
void Insert (NODEPTR *headPtr,
             NODEPTR  temp)
```

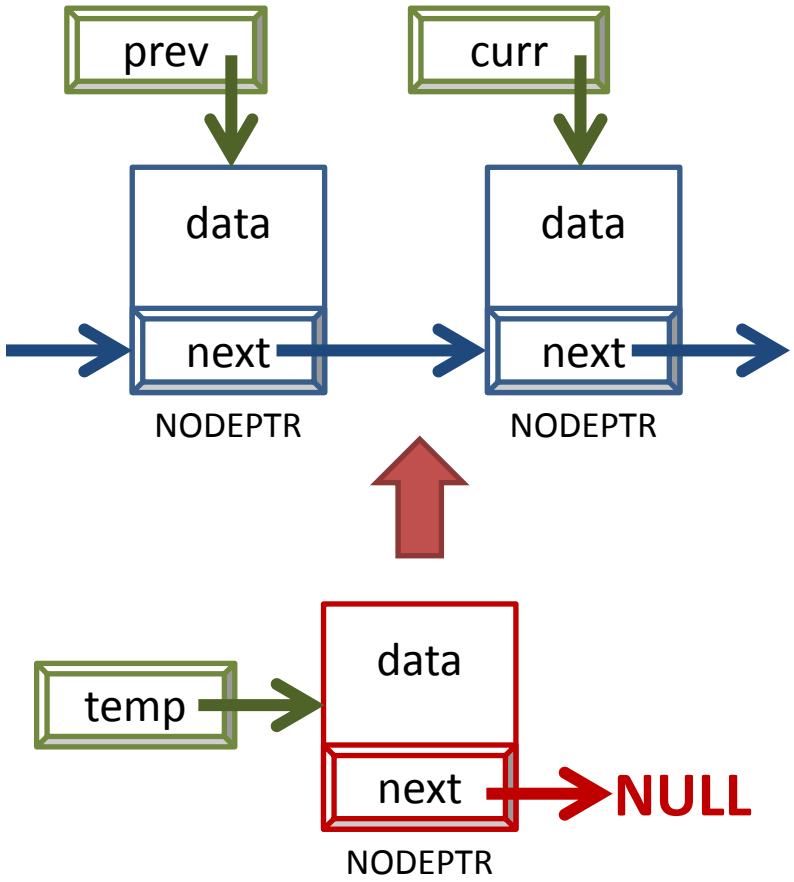
- check if list is empty
  - if so, temp becomes the first node
- if list is not empty
  - traverse the list and insert temp at the end

Inserting a Node in Middle

```
int Insert (NODEPTR *headPtr,
        NODEPTR  temp, int where)
```

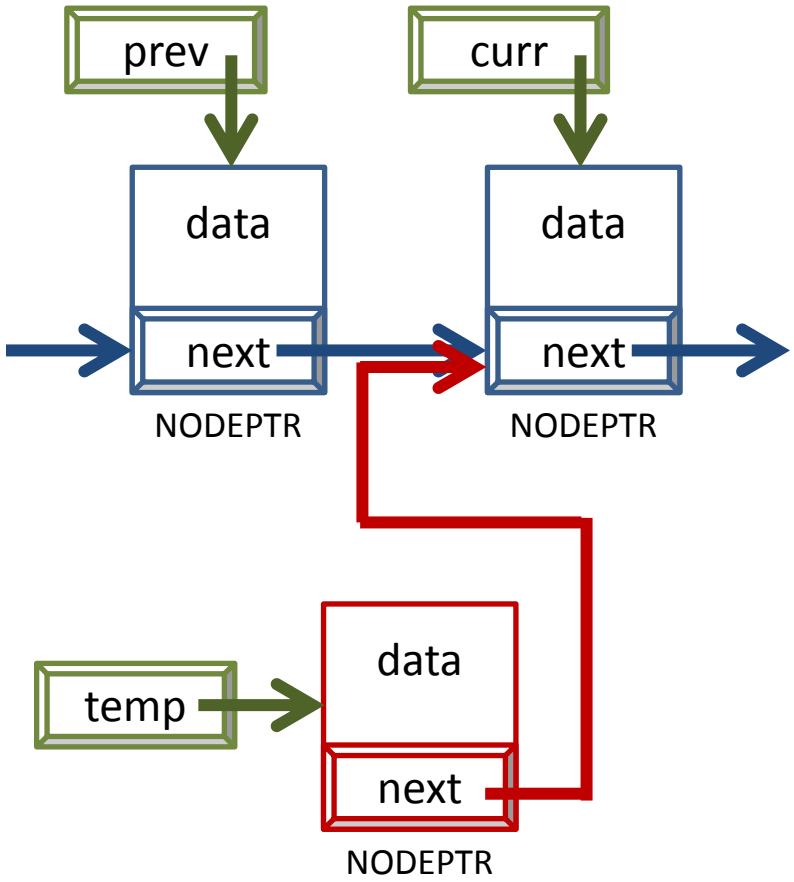
- traverse list until you come to point to insert
  - CAUTION: don't go past the end
- insert temp at appropriate spot
  - CAUTION: don't "lose" any pointers
- return an integer to convey success/failure

Inserting a Node – Step 1



- traverse the list until you find where you want to insert temp

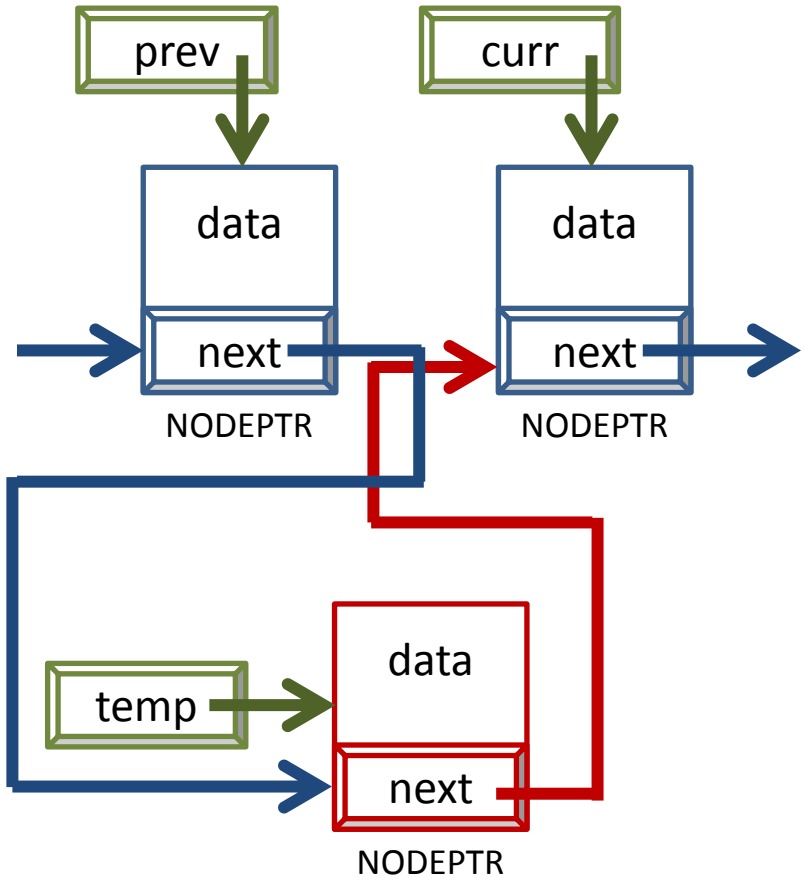
Inserting a Node – Step 2



- first have **`temp`** point to the next node in the list (**`curr`**)

```
temp->next = curr;
```

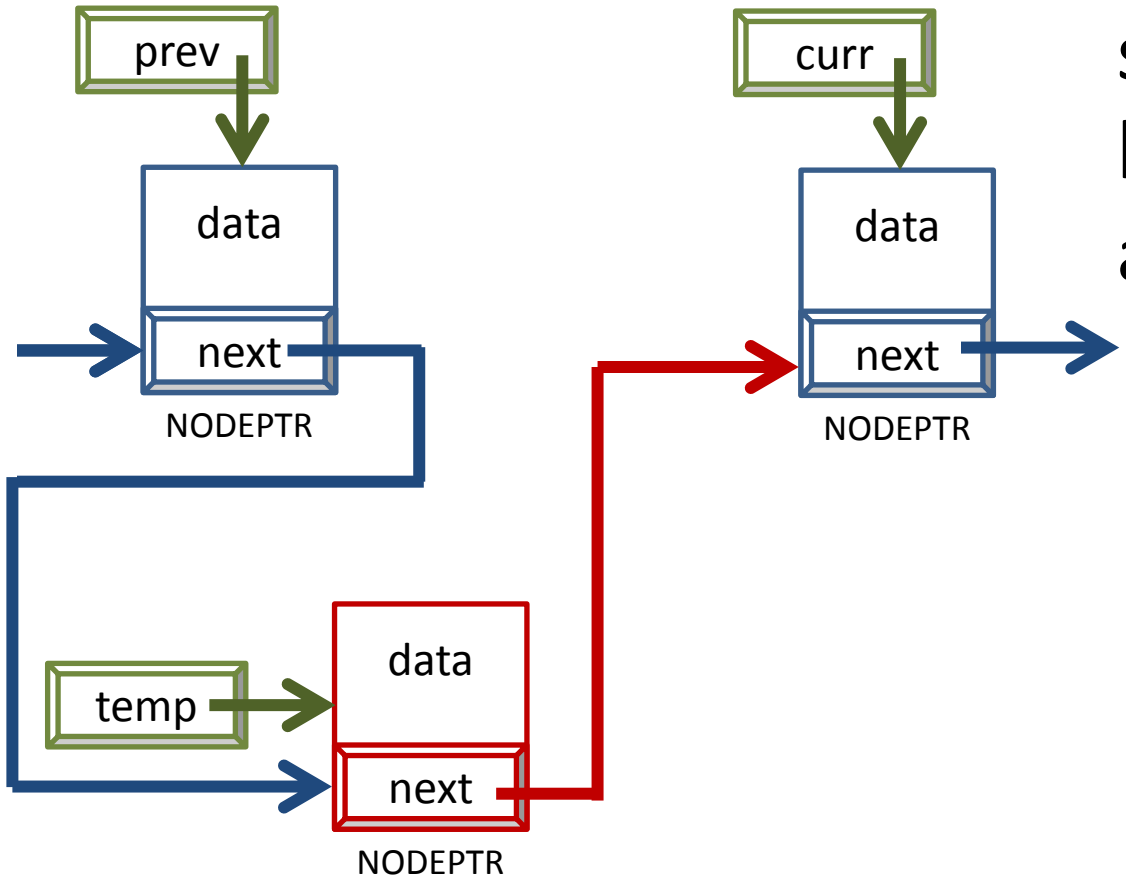
Inserting a Node – Step 3



- then you can have **`prev`** point to **`temp`** as the new next node in the list

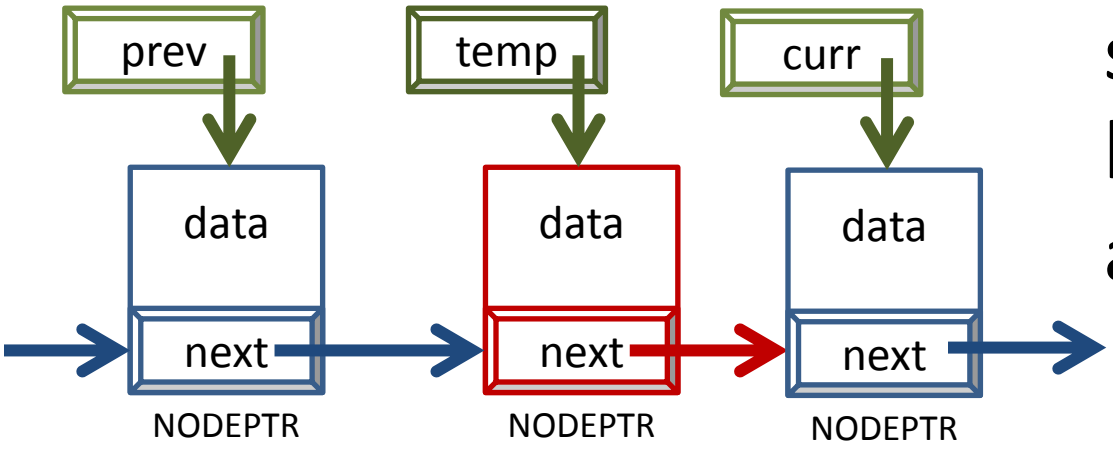
```
temp->next = curr;
```

Inserting a Node – Done



- **`temp`** is now stored in the list between **`prev`** and **`curr`**

Inserting a Node – Done



- **`temp`** is now stored in the list between **`prev`** and **`curr`**
  - return a successful code (insert worked)

Deleting a Node

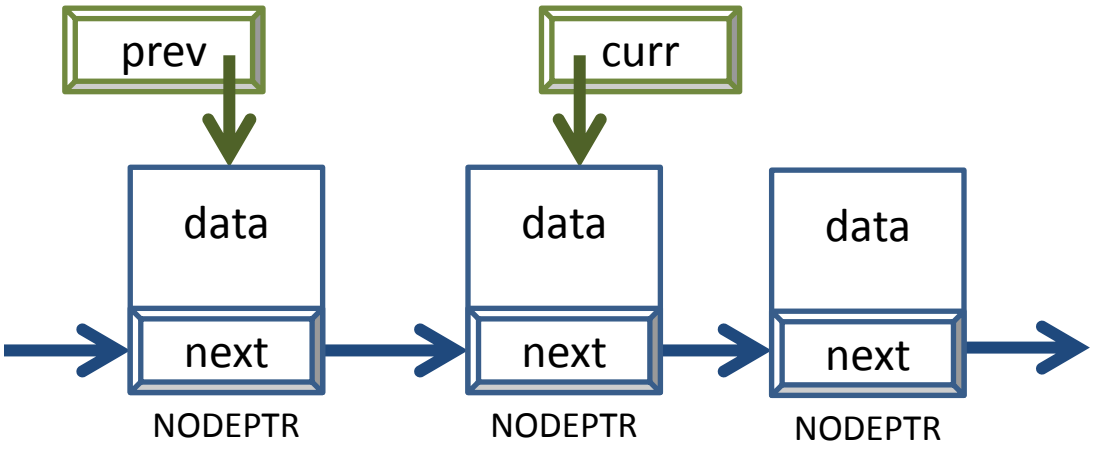
```
int Delete (NODEPTR *headPtr,
            int target)
```

- similar to insert
- pass in a way to find node to delete
  - traverse list until you find the correct node:

```
curr->data == target
```

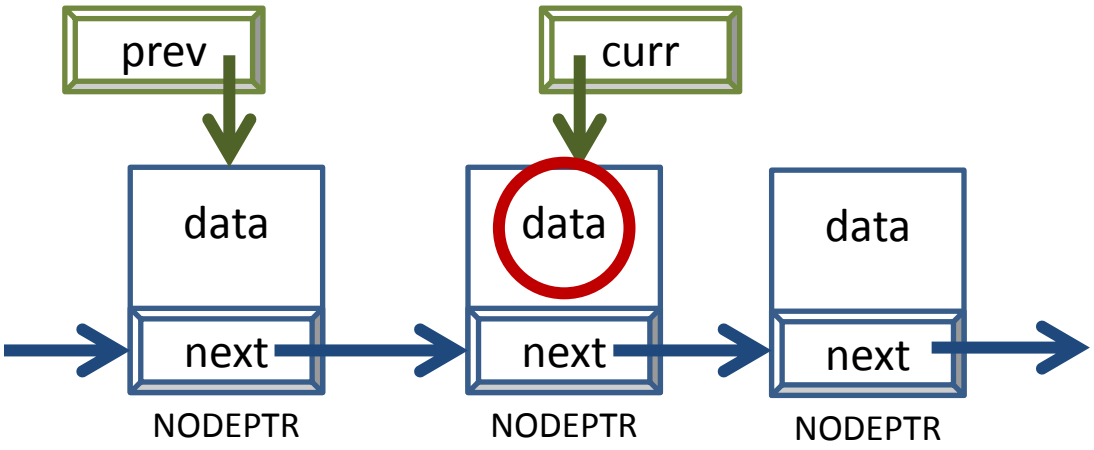
- return an integer to convey success/failure

Deleting a Node – Step 1



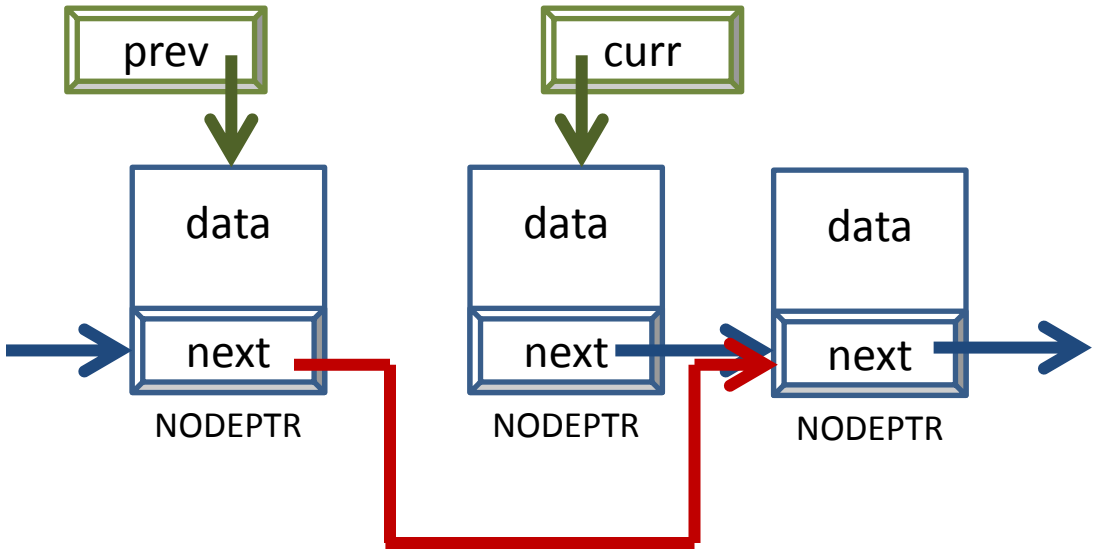
- traverse the list, searching until **`curr->data`** matches **`target`**

Deleting a Node – Step 1



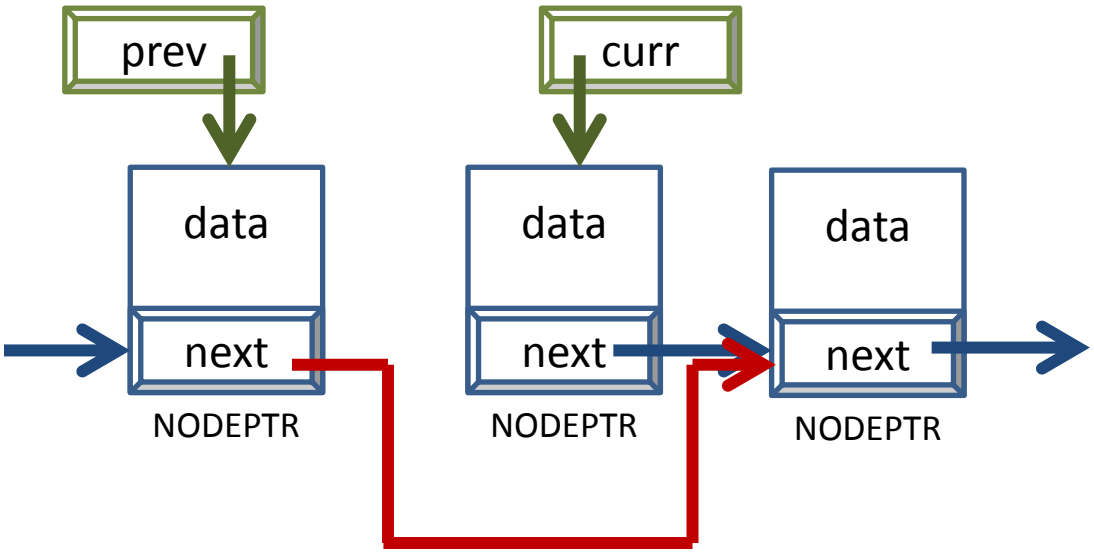
- traverse the list, searching until **`curr->data`** matches **`target`**

Deleting a Node – Step 2



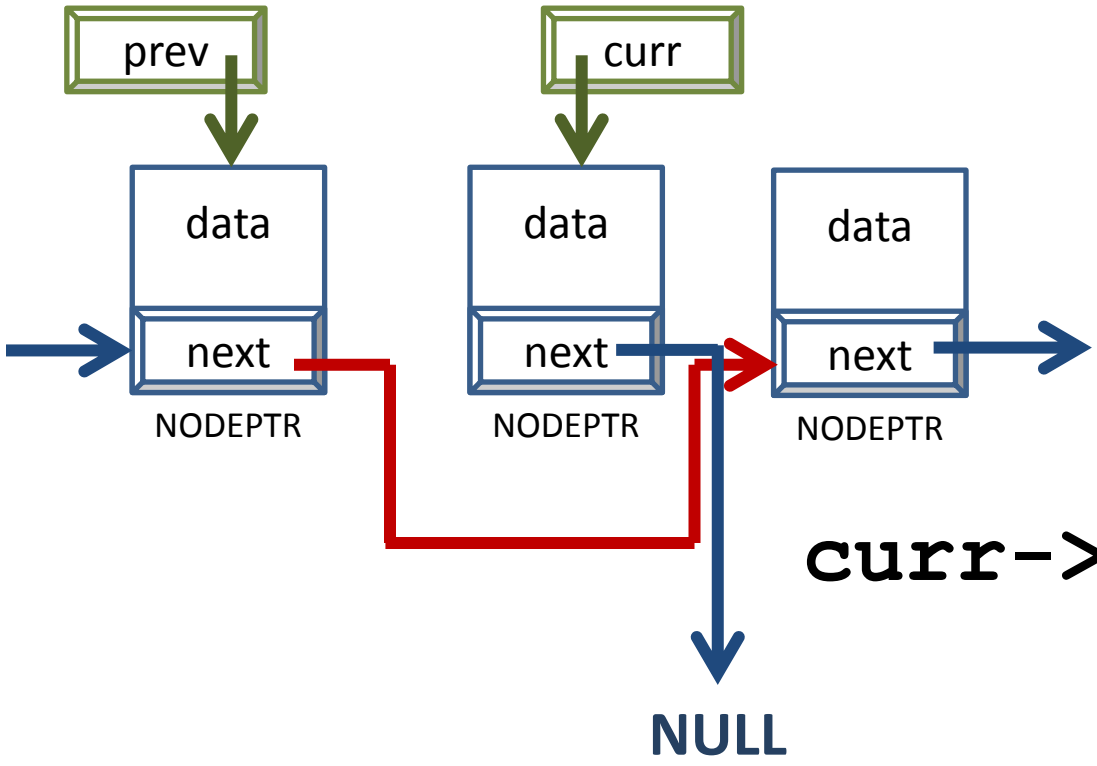
- "remove" **`curr`** from the list by changing **`prev->next`** to **`curr->next`**

Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

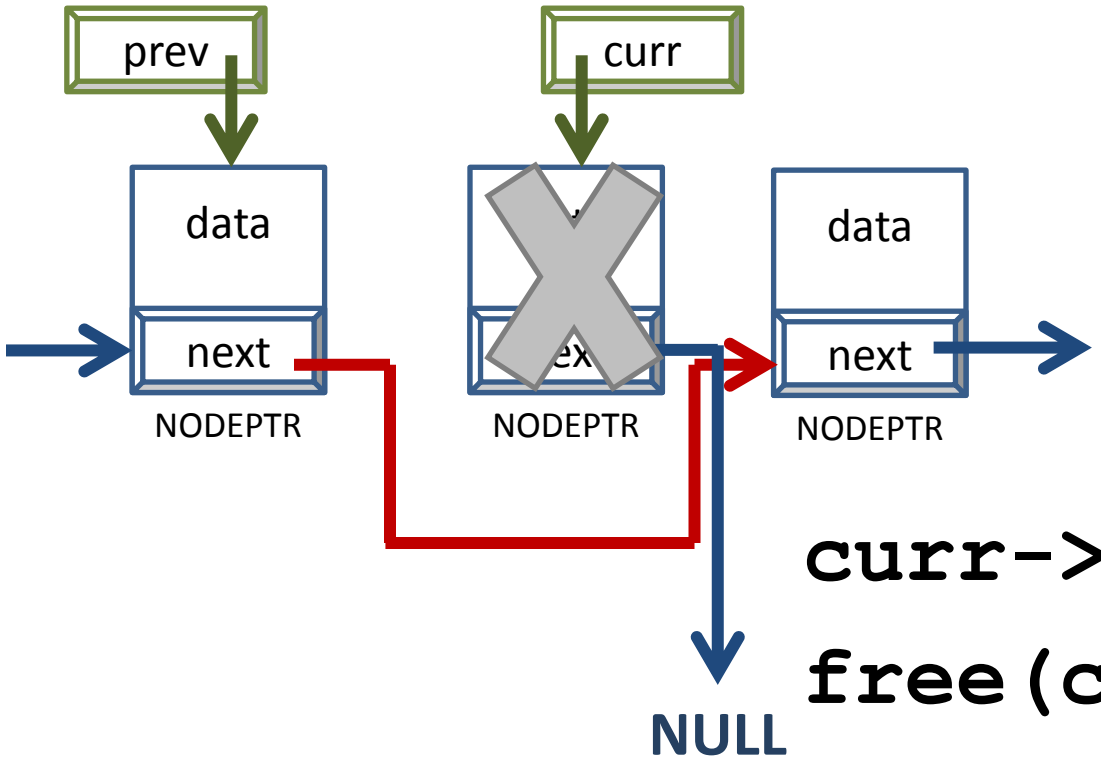
Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

```
curr->next = NULL;
```

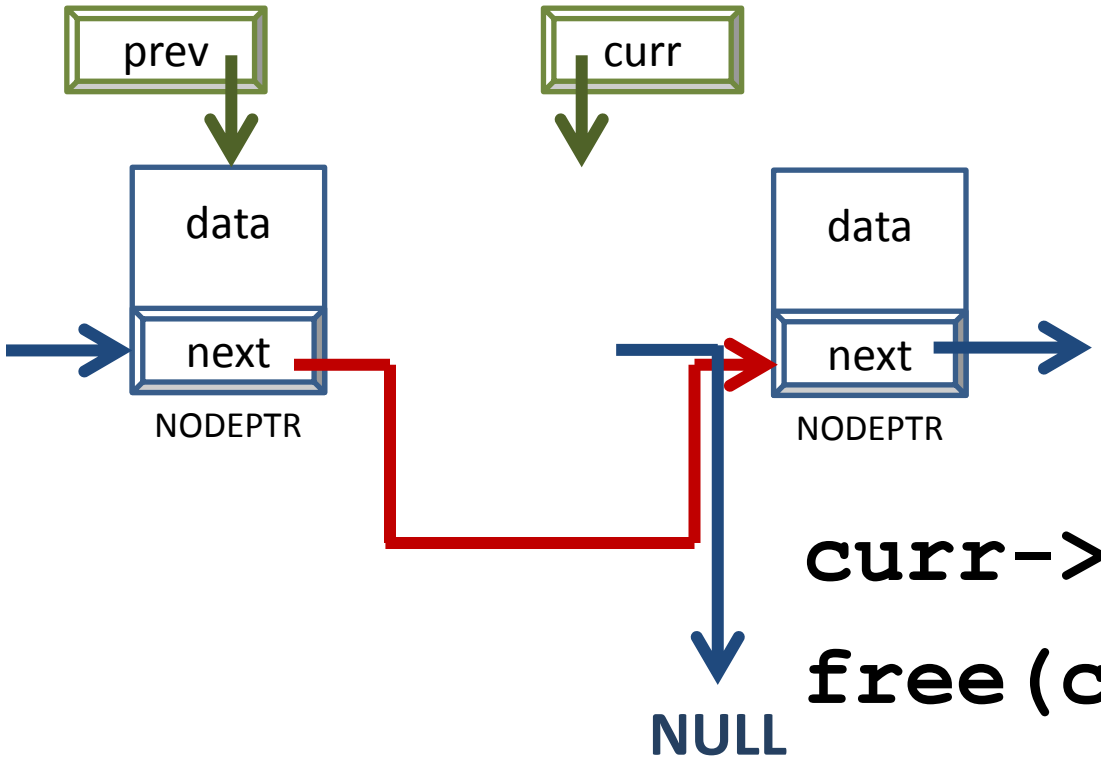
Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

```
curr->next = NULL;
free(curr);
```

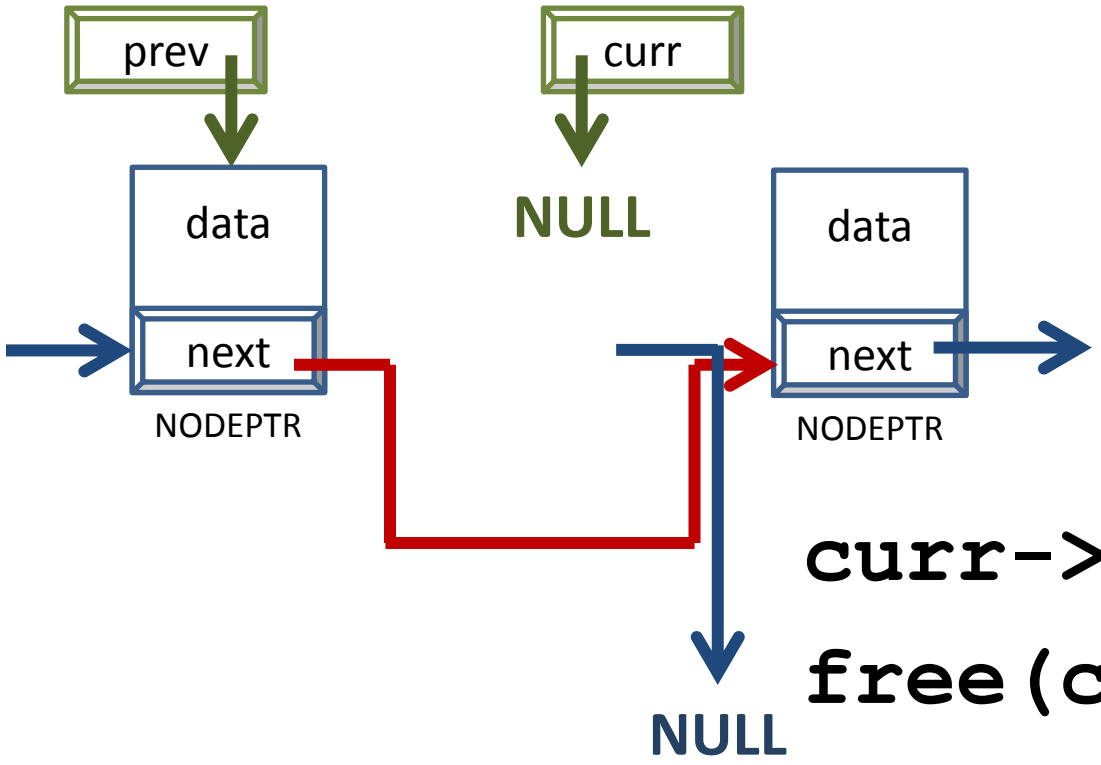
Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

```
curr->next = NULL;
free(curr);
```

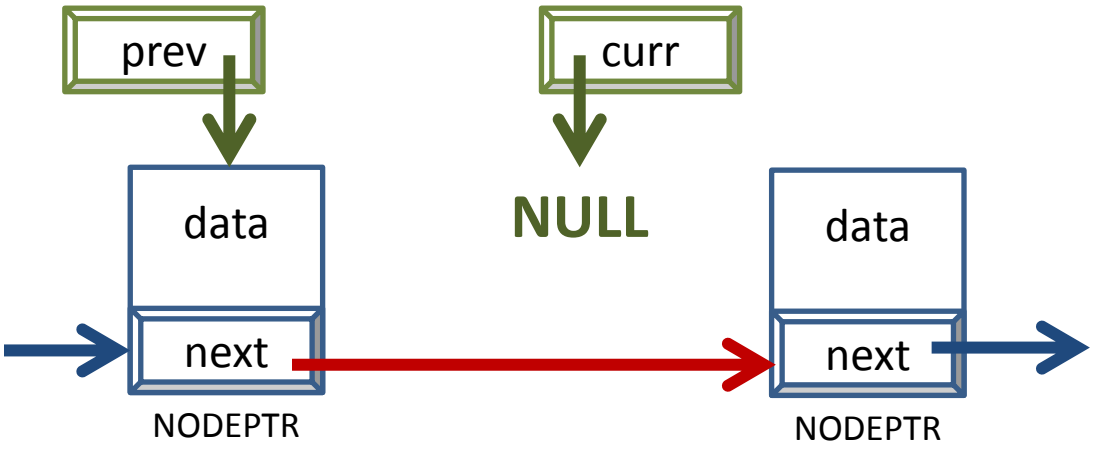
Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

```
curr->next = NULL;
free(curr);
curr = NULL;
```

Deleting a Node – Step 3



- free the memory used by **`curr`** and set pointers to NULL

```
curr->next = NULL;
free(curr);
curr = NULL;
```

Linked List Code

- code for all of these functions available on the syllabus page
- comments explain each step
- you can use this code in your Homework 4B, or as the basis for similar functions

Homework 4B

- Karaoke

- heavy on pointers and memory management
- think before you attack

- start early
- test often
- use a debugger when needed

Moving a Node Between Lists

- will need to write a Move() function to perform this task for Homework 4B

Moving a Node Between Lists

- will need to write a Move() function to perform this task for Homework 4B

- traverse list until you come to node to move
  - CAUTION: don't go past the end
- remove node from one list, add to other
  - CAUTION: don't "lose" any pointers
- return an integer to convey success/failure