Units 6 & 7: Linked Lists

Pointer Recap

- Computers store data in memory cells
- Each cell has an *unique address*

Addr	Content	Addr	Content	Addr	Content	Addr	Content
0	x: 2	1	y: 6	2	k: 8	3	m: 9
4	a[0]: ‘a’	5	a[1]: ‘b’	6	a[2]: ‘\0’	7	3
8	2	9	3	10	2	11	10
12	…	13	…	14	…	15	…

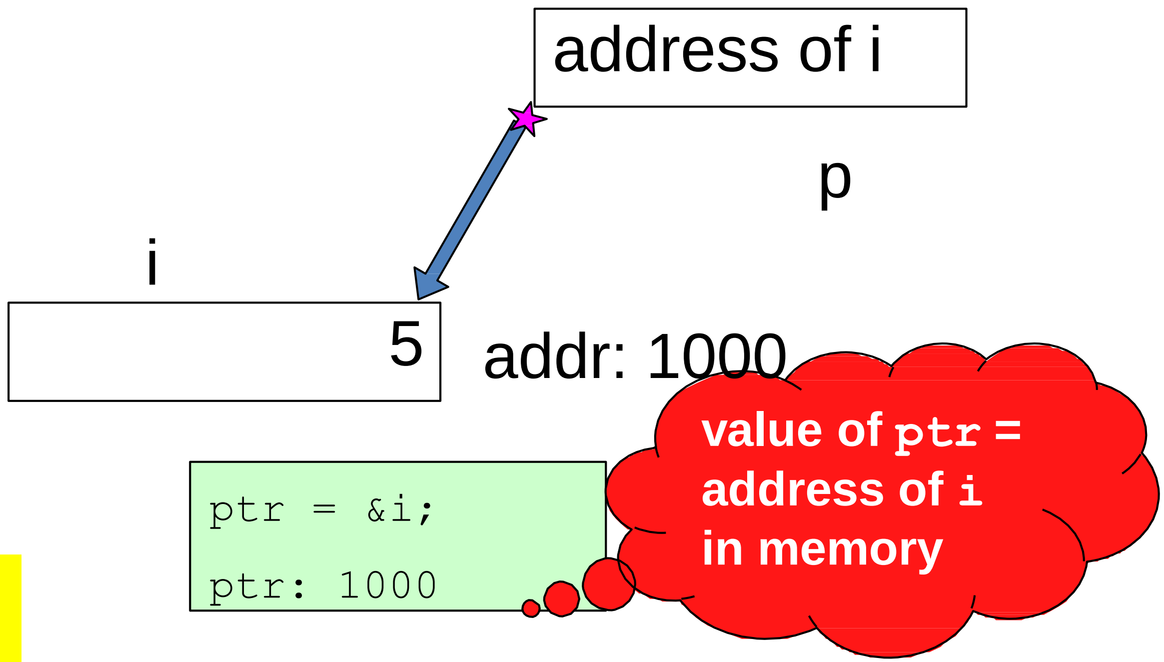
Addressing Concept

- Pointer stores the **address** of an entity (variable)
- It **refers** to a memory location

```
int i = 5;
int *p; /* declare a pointer variable */
p = &i; /* store address-of i to ptr */
```

What actually *ptr* is?

- **`p`** is a variable storing **an address**
- **`p`** is **NOT** storing the actual value of i



scanf("%d", &i);

Twin Operators

- &: Address-of operator
  - Get the *address* of an entity
    - e.g.
      - `int *ptr;`
      - `ptr = &j;`
  - Don't mix it with && -- AND operator

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 33	1002	k: 58	1003	m: 74
1004	ptr: ??	1005		1006		1007	

Twin Operators

- *: De-reference operator
  - Refer to the *content* of the referee
    - e.g. `*ptr = 33;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 33	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

Linked Lists

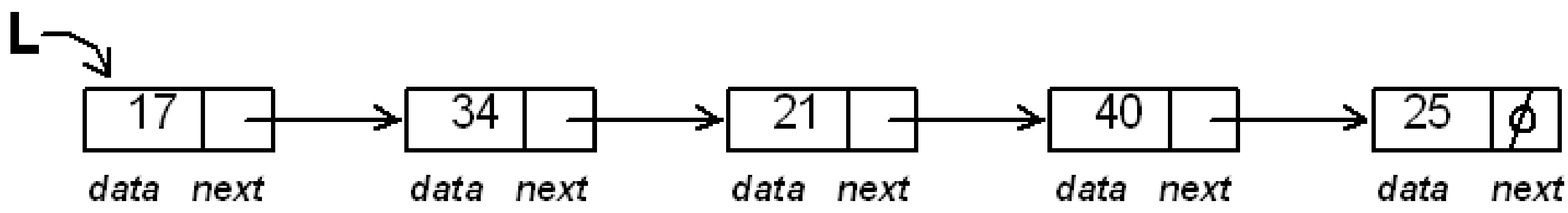
Linked Lists

- A *linked list* is a series of connected *nodes (cells)*
- Here is a linked list of integers:

  - Data: an integer field
  - Pointer: stores the address of the next node

- The end of the list is marked by a **NULL** symbol, shown by ϕ in the pointer field.
  - Conceptually different from the NULL byte of a string

Scanning Linked Lists



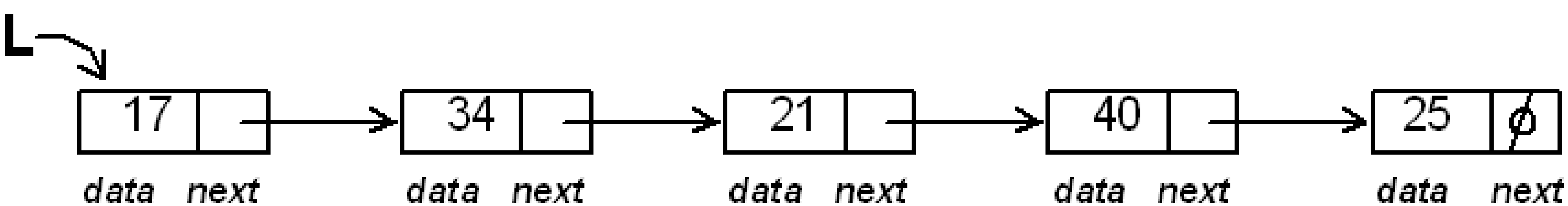
- To access a linked list, we need a pointer to its first node.
  - L is a pointer-type variable
  - It holds an access pointer (the curved arrow) to the first node of the list
  - From the first node , we can move to the second and so on.

Scanning Linked Lists

– Scanning must stop when `NULL` is reached.

– With a linked list, there is no random (or direct) access

- to reach the 4th node, we must start from the access pointer, and scan along.
- This is different from arrays (which have random access)



Linked Lists in C

- We declare a structure to store a node
  - e.g., in C:

```
typedef  struct _node{
      int data;
      struct _node* next;
} node;
```

  - The declaration `struct _node* next` is recursive, or self-referring – i.e.,
    - Attribute `next` is the same type as that being defined.
    - This allows one `node` to point to another `node` – and so on!

A note on notation

```
typedef  struct _node{
   int data;
   struct _node* next;
} node;
```

- In C, if we have a struct `node`, with two fields: `data` and `next`

```
node x; // x is an instance of node
node *y; // y is a pointer to a node
```

- To access (e.g.) the data field we say

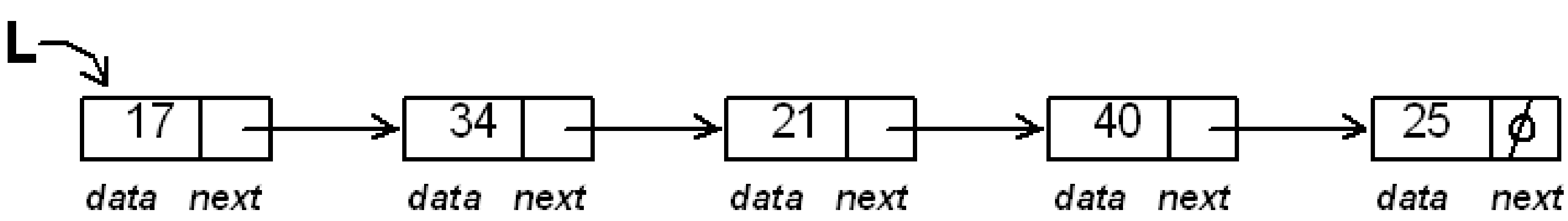
```
x.data = 12; // dot notation
y = malloc(sizeof(node));
y->data = 12; // arrow notation
```

Scanning

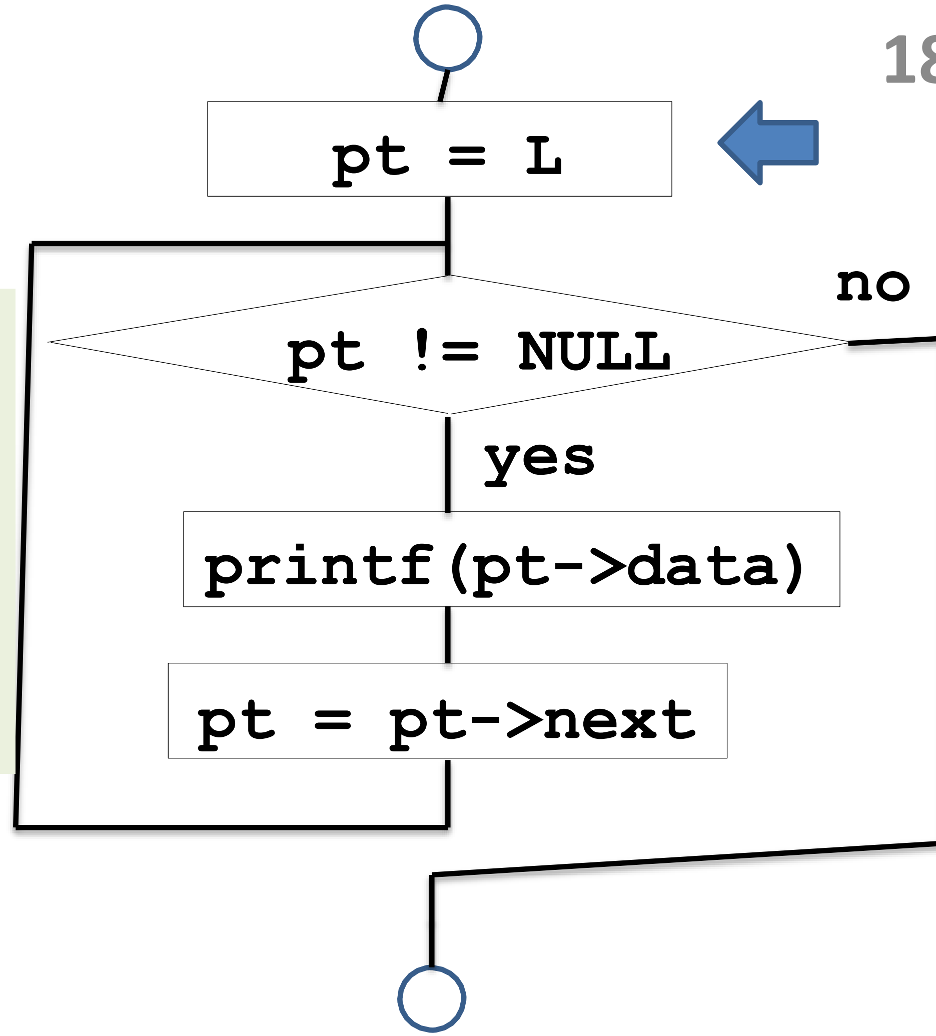
- **Scanning must stop when NULL is reached.**
  - `pt` starts at the first node, set by `pt = L`;
  - If `pt!= NULL`, we haven't reach the last node
  - So we move to the next node, using

    `pt = pt->next;`

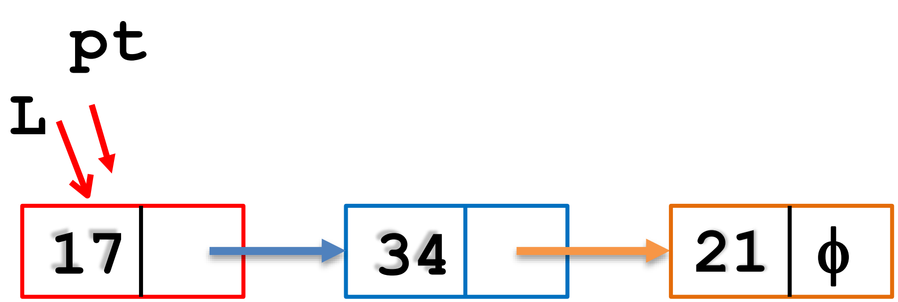
```
void traverse(node *L) {
    node *pt = L;
    while (pt != NULL) {
          printf("%d\n", pt-
          >data);
          pt = pt->next;
    }
 }
```

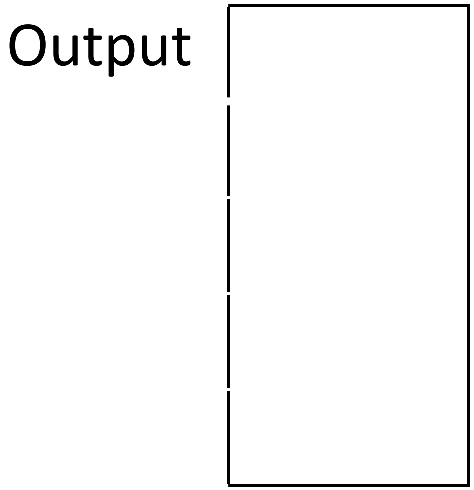


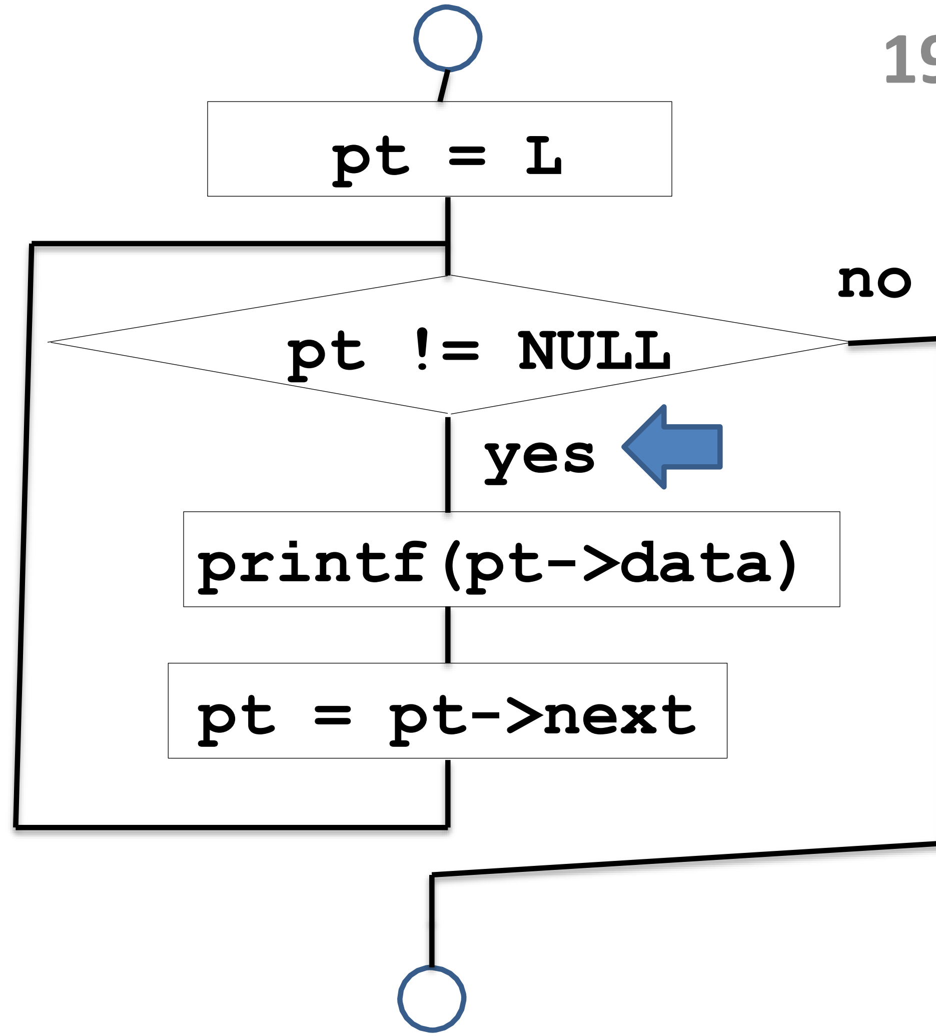
```
void traverse(node *L)
 {
        node *pt = L;
        while (pt != NULL)
        {
          printf("%d\n", pt->data);
          pt = pt->next;
        }
 }
```

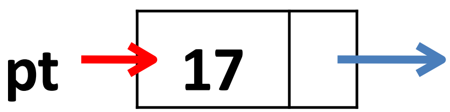


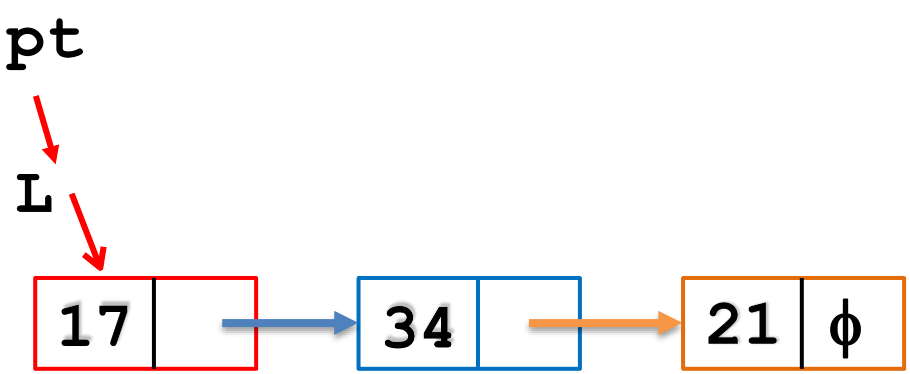
Initialise scan pointer …

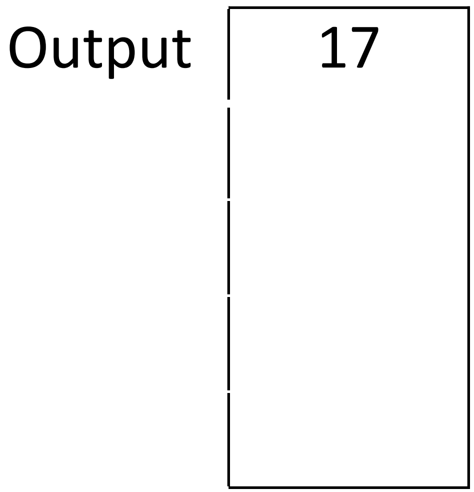


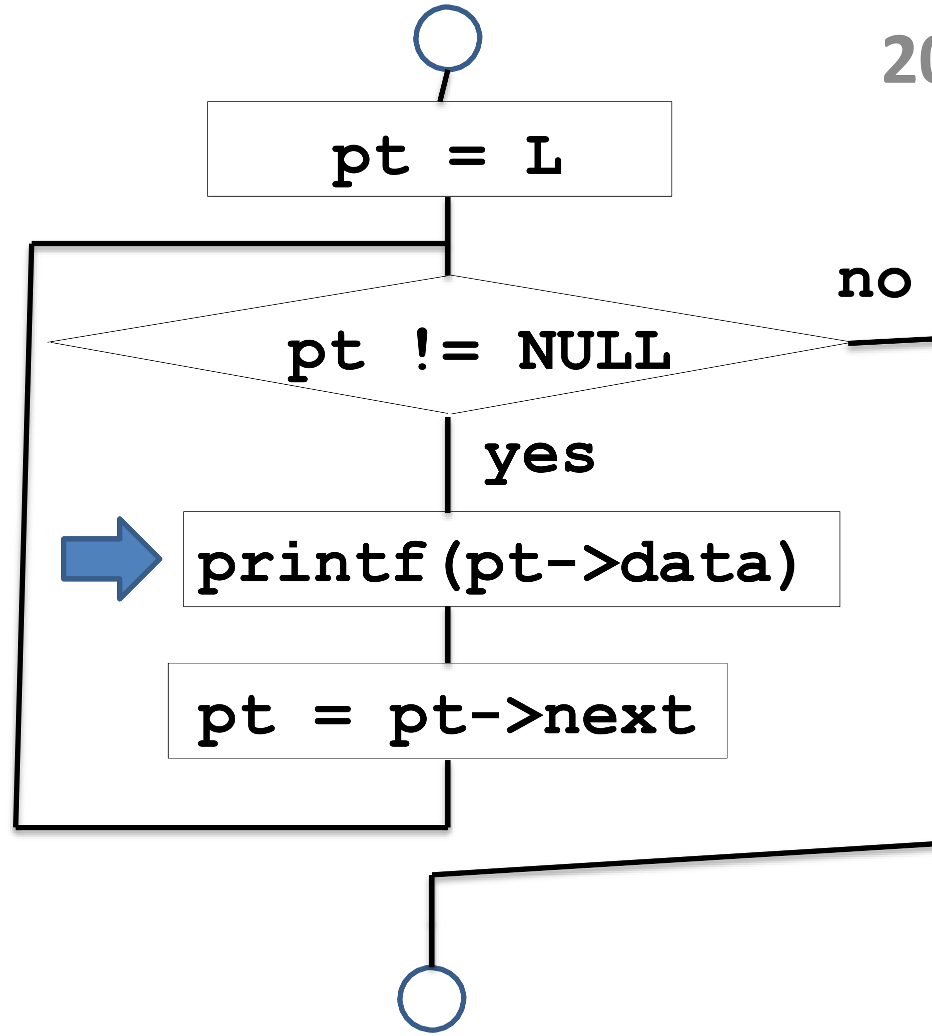

Output

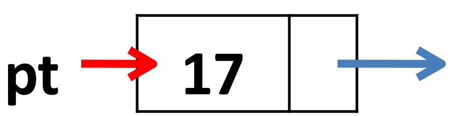

pt = L
pt != NULL
no
yes
printf(pt->data)
pt = pt->next

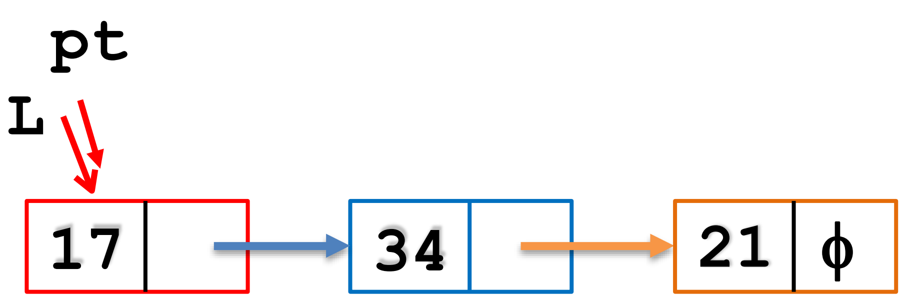

pt
17

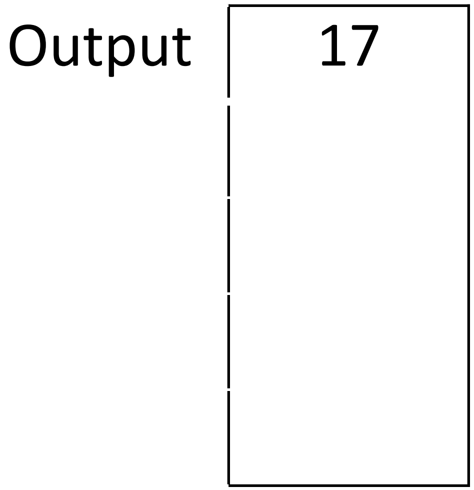

pt
L
17
34
21
ϕ

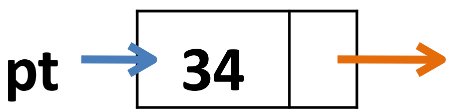

Output
17

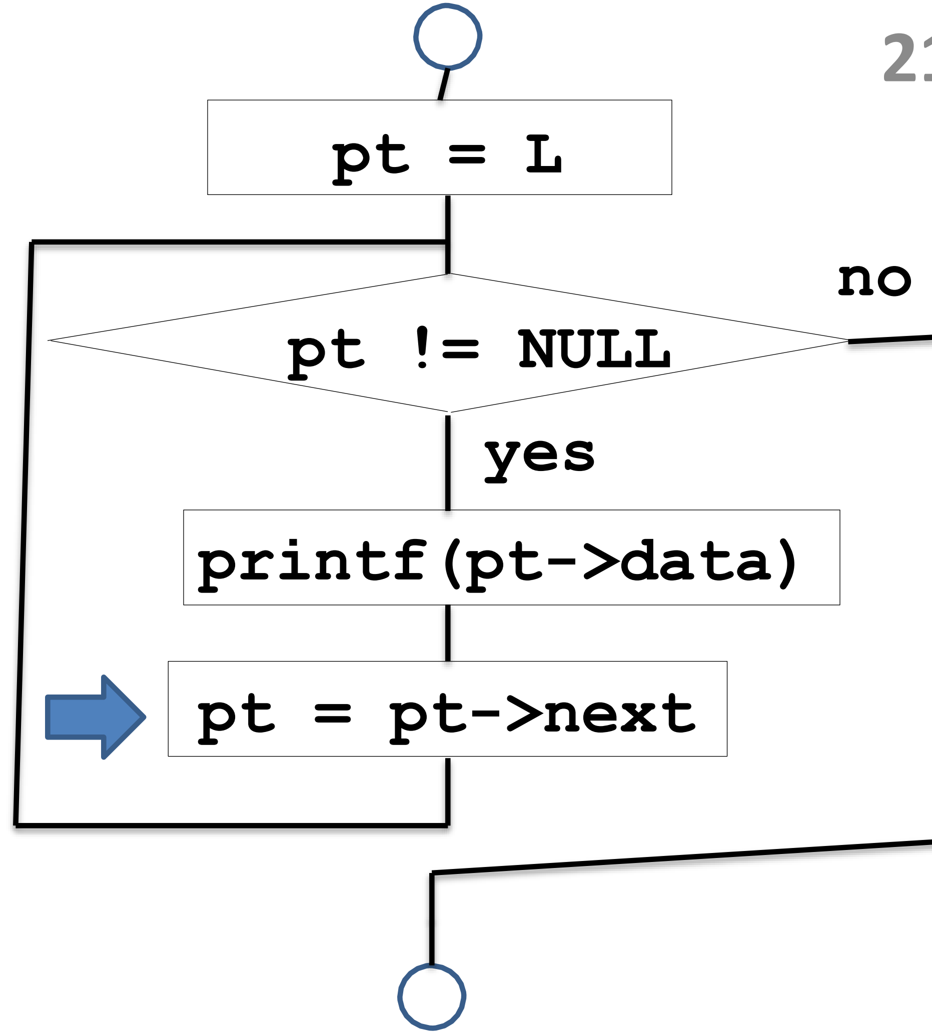

pt = L
pt != NULL
no
yes
printf(pt->data)
pt = pt->next

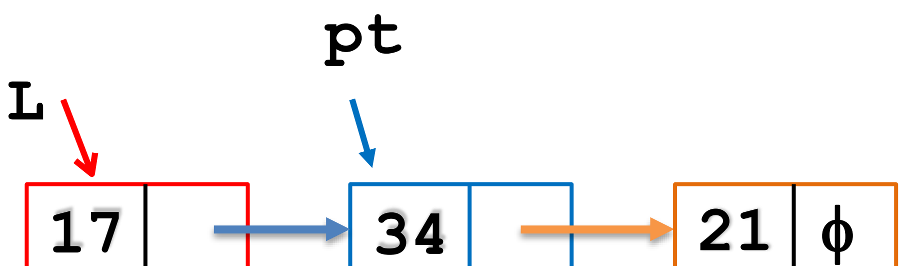

pt
17


pt
L
17
34
21
ϕ

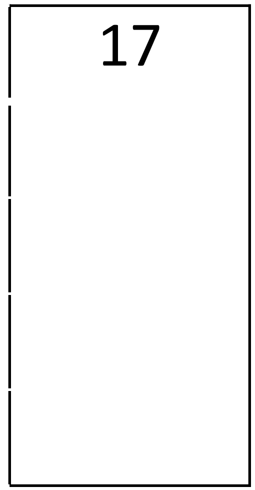

Output
17

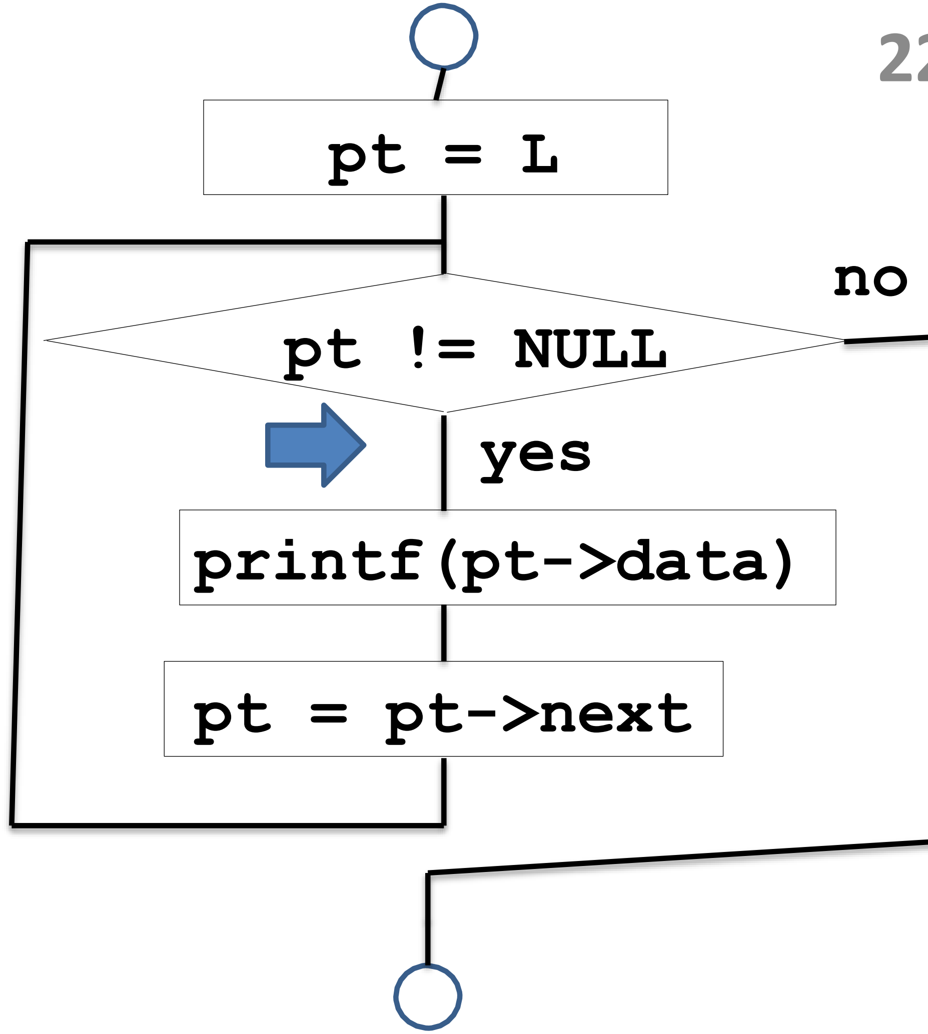

pt
34

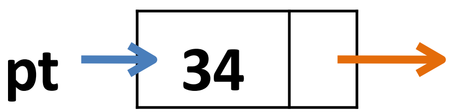

pt = L
no
pt != NULL
yes
printf(pt->data)
pt = pt->next

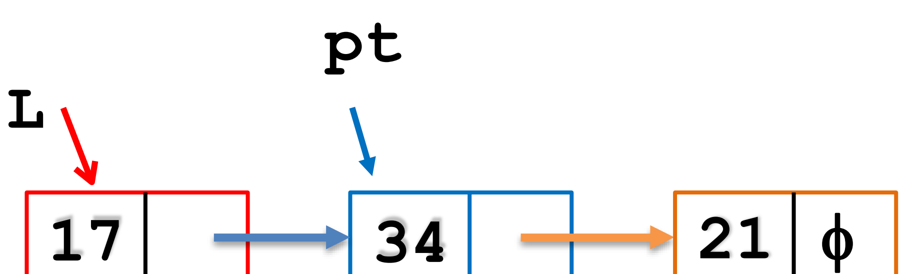

pt
L
17
34
21
ϕ

Output


17


pt = L
pt != NULL
no
yes
printf(pt->data)
pt = pt->next

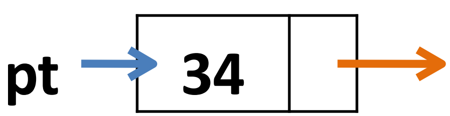

pt
34

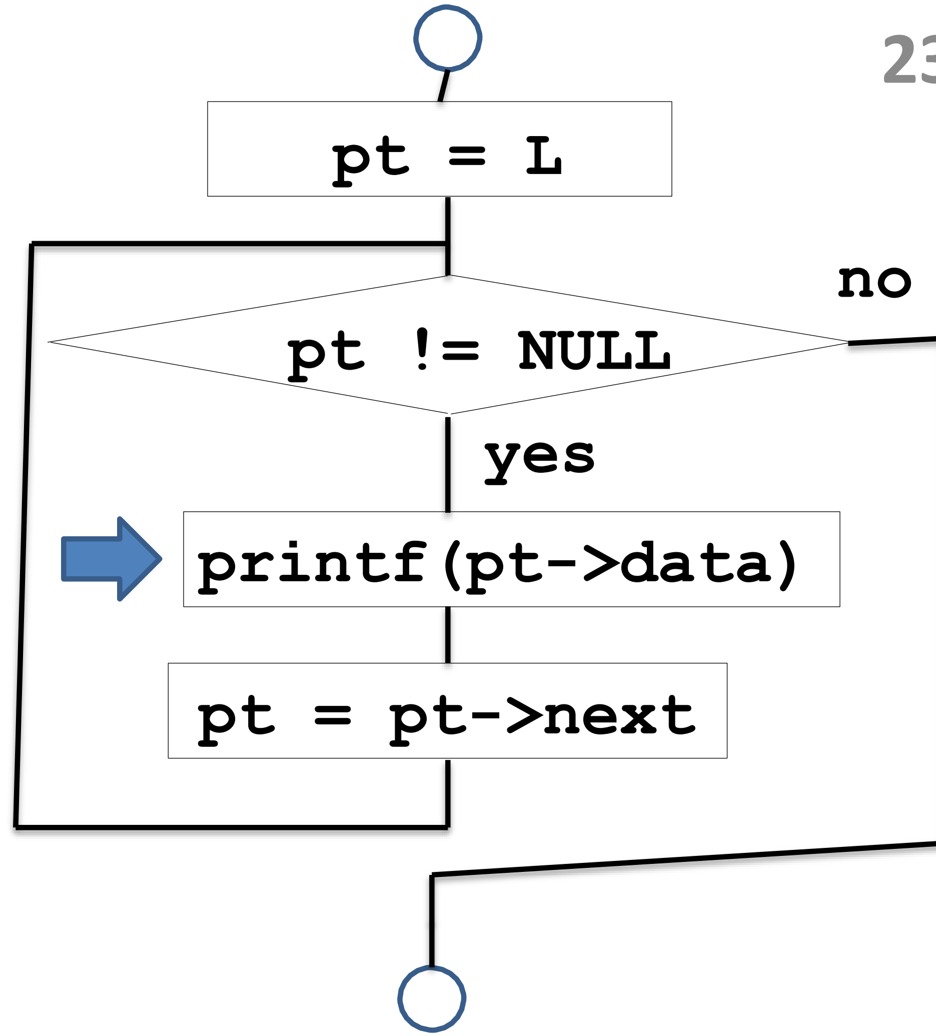

pt
L
17
34
21
ϕ

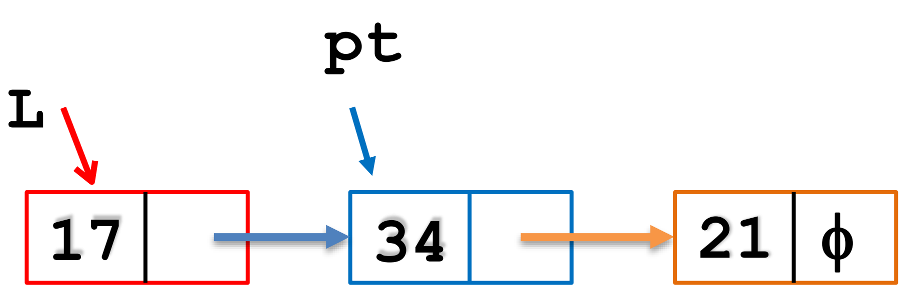
Output

17

34

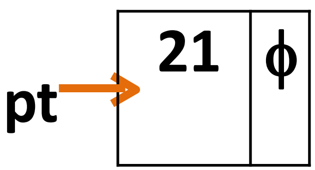


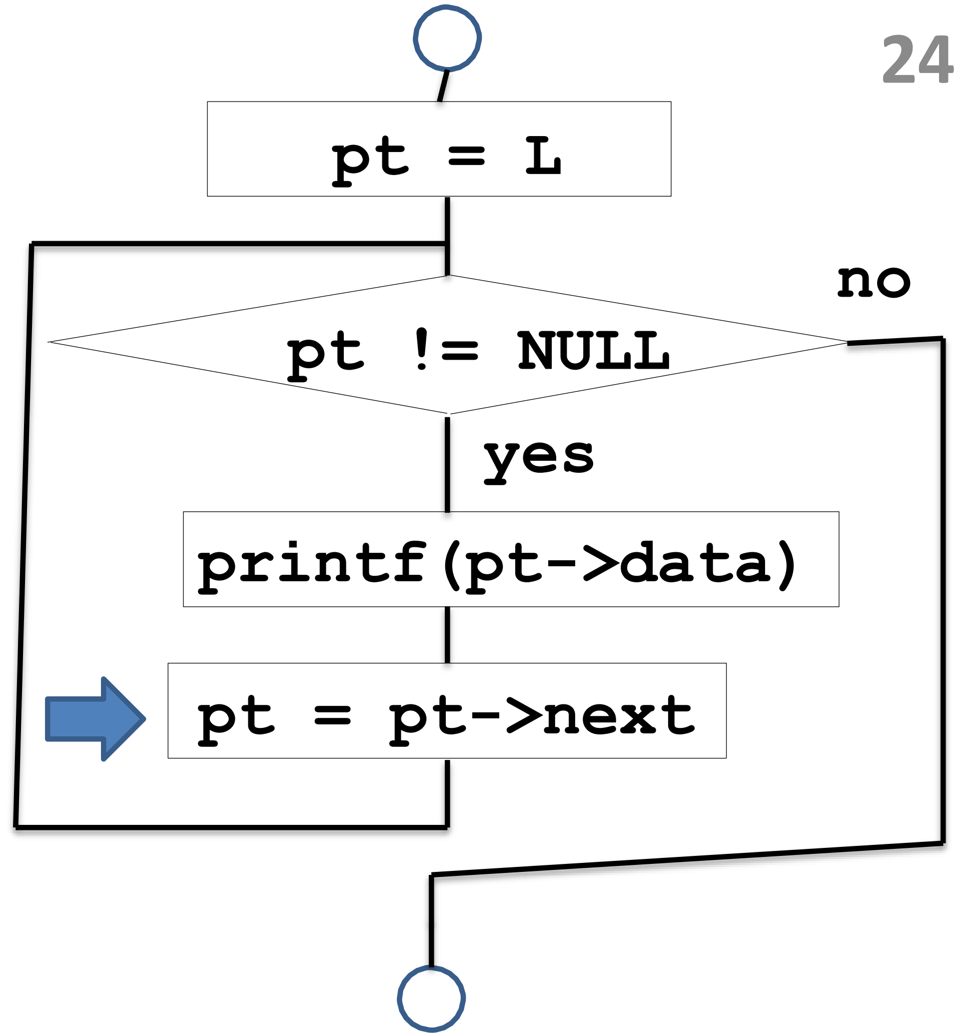


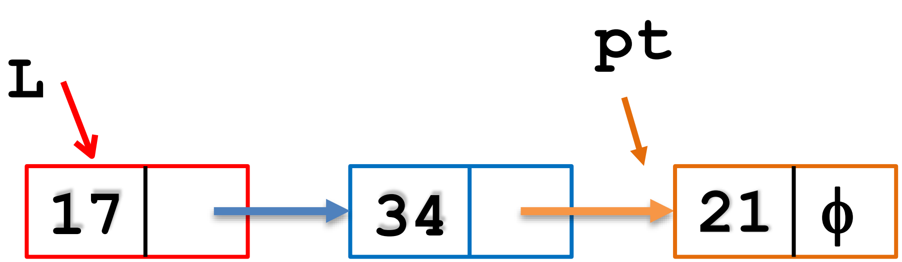


Output

17
34

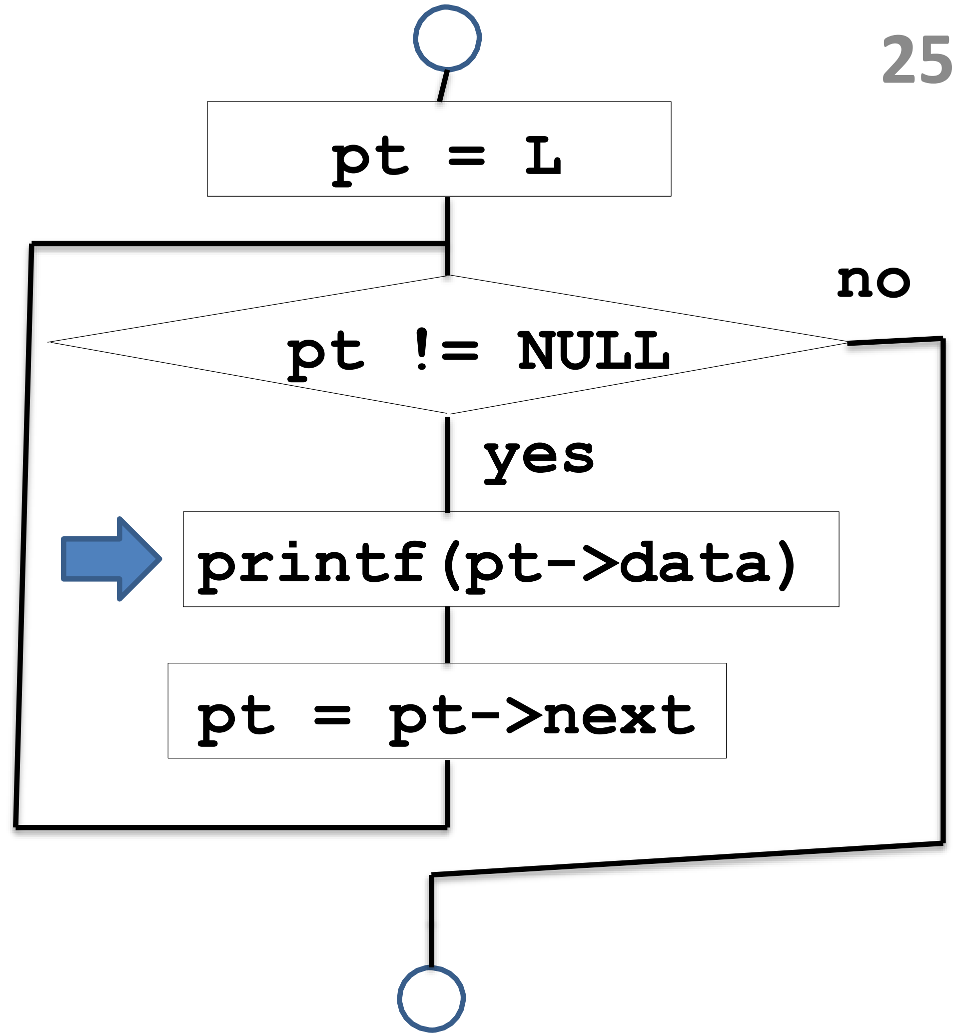


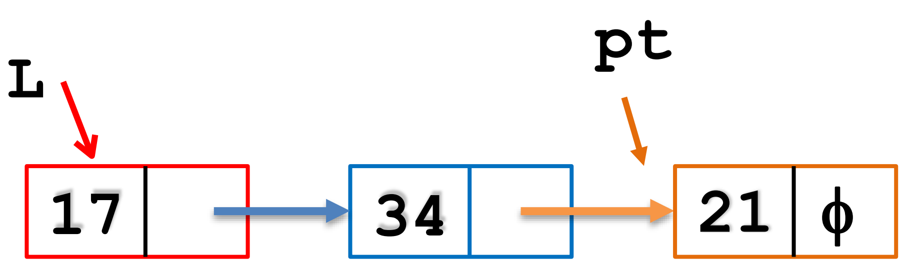




Output

17
34
21

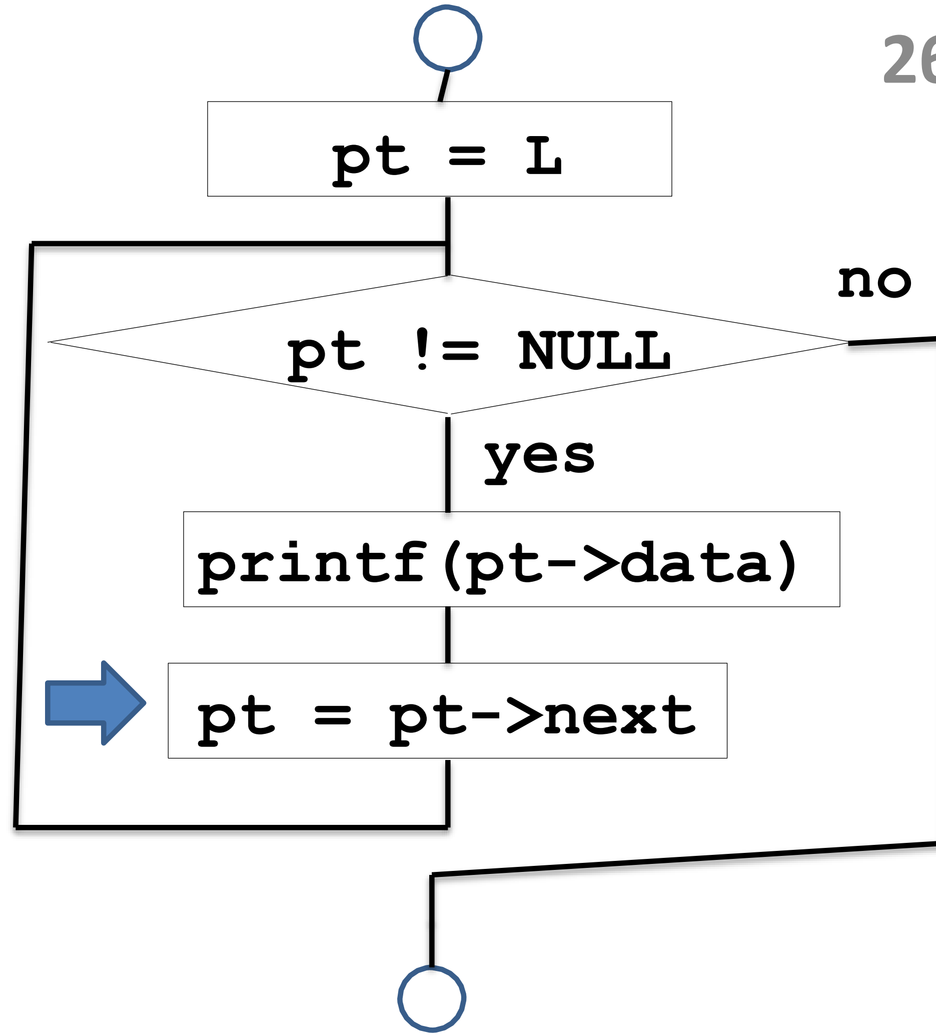


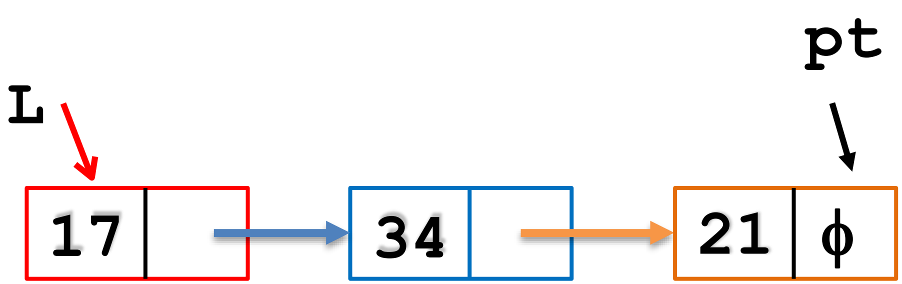


Output

17
34
21

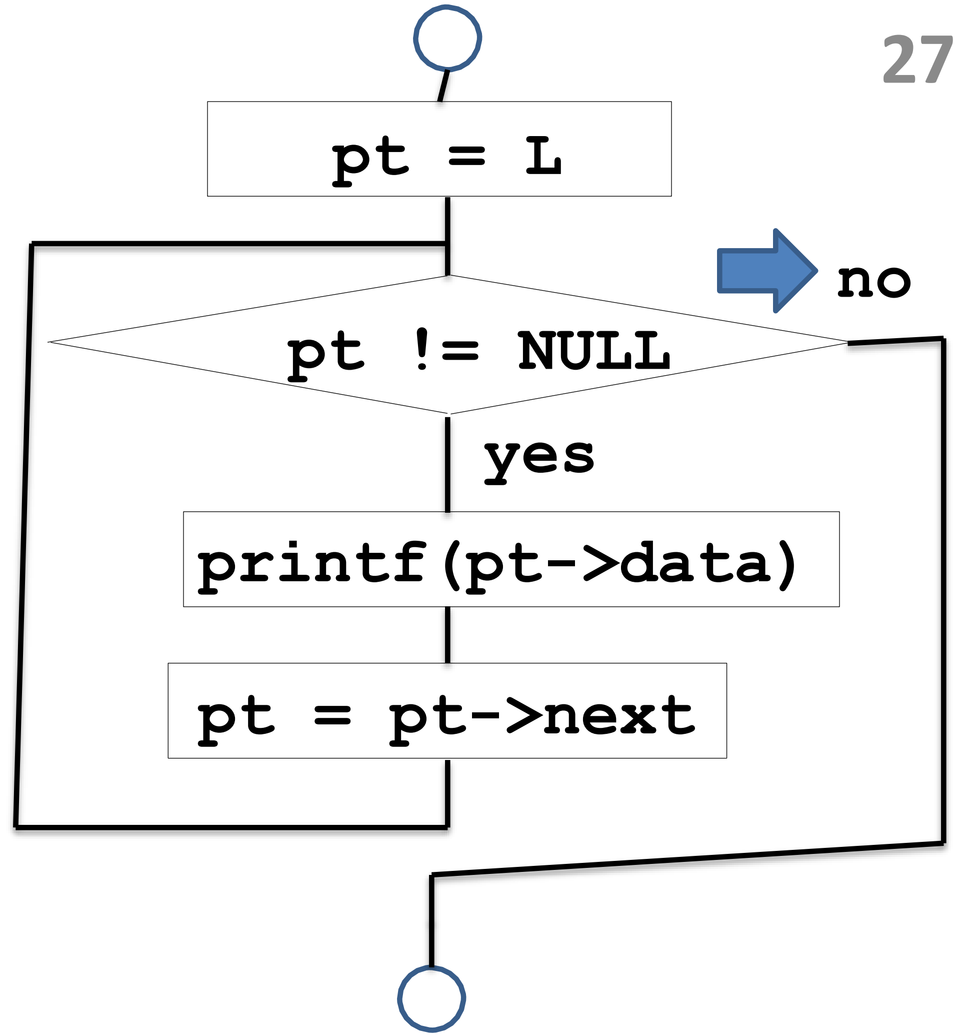
pt → ϕ

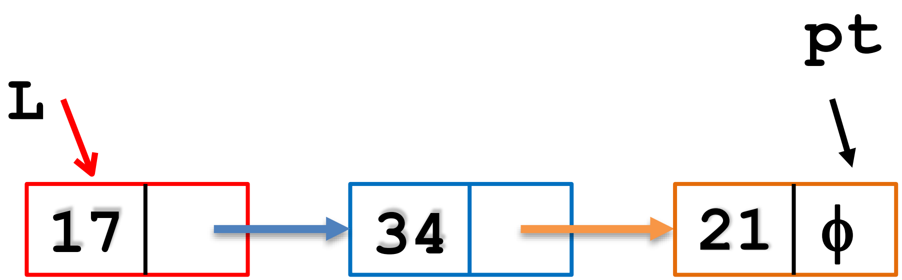




Output
17
34
21

pt → ϕ

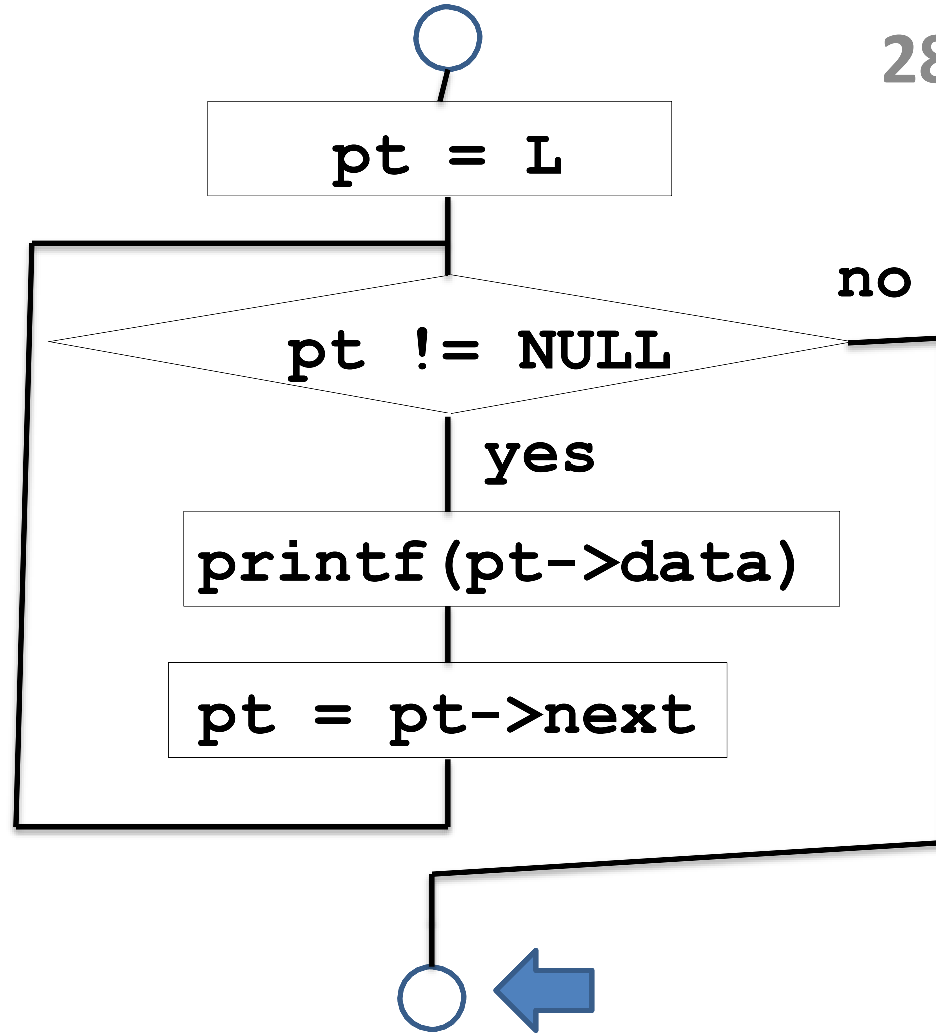


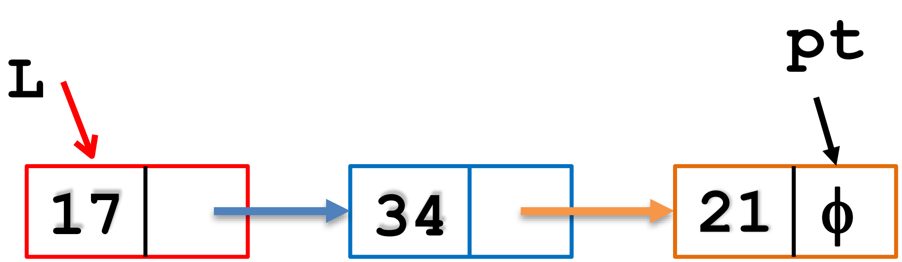


Output

17
34
21

pt → ϕ





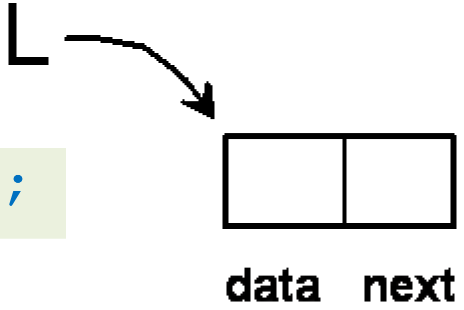
THE DYNAMIC NATURE OF LINK LISTS

Advantages of Linked Lists

- Linked Lists are *dynamic*
  - *Dynamic* means they can grow or shrink as required
- Linked lists are non-contiguous
  - Different from arrays
  - It is could be hard to find a large contiguous memory space

Building a List

- To create memory space for a pointer to point to, in C we use **`malloc`** and **`sizeof`**.
  - **`sizeof`** tells us how many bytes a node structure occupies.
  - **`malloc`** allocates that many bytes from the memory and returns the address of the allocated memory space.



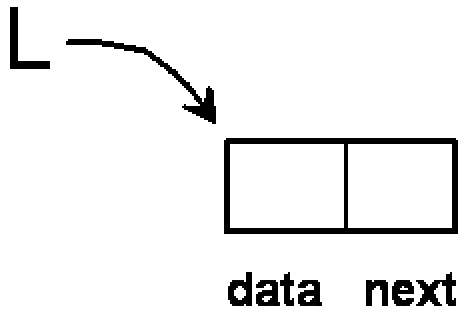

```
node *L = malloc(sizeof(node));
```

– variable **`L`** points to a newly created **`node`** element

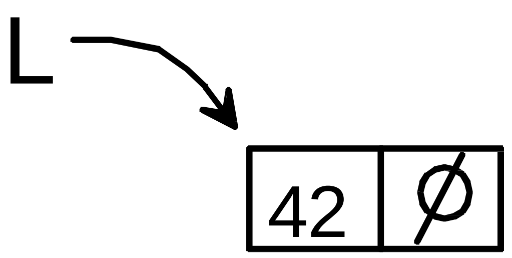
Building a List (cont)

- C uses arrow notation to assign values, where L is a pointer.

```
L->data = 42;
L->next = NULL;
```



- Now, L points to a one-element linked list:
- Not very interesting!
  – How do we grow the list?

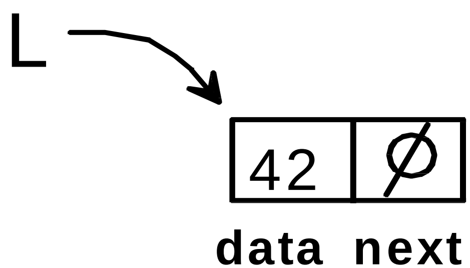


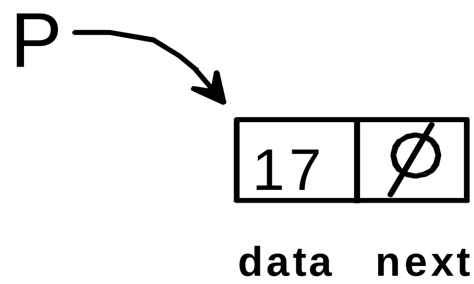
Creating a new node

```
node* new(int _data, node*_next)
{
      node *res = malloc(sizeof(node));
      res->data = _data;
      res->next = _next;
      return res;
}
```

Building a List

- Let’s create two node elements:
  - `node *L = new ( 42, NULL);`
    `node *P = new ( 17, NULL);`

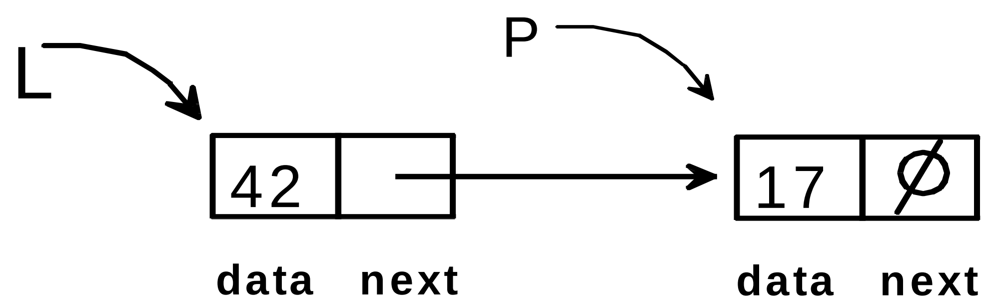




- Now let’s just do this ....

  `L->next = P;`

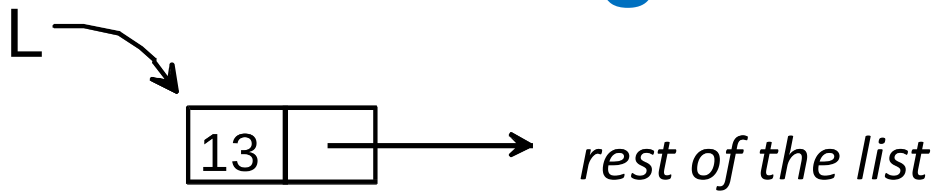
- .... and we get a 2-node list



GROWING THE LIST: ADDING AT THE FRONT

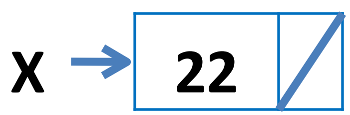
Adding at the Front

Existing list:



- Given any **non-empty** list L:
  - Let's add 22 – first make a node `X` to hold it:

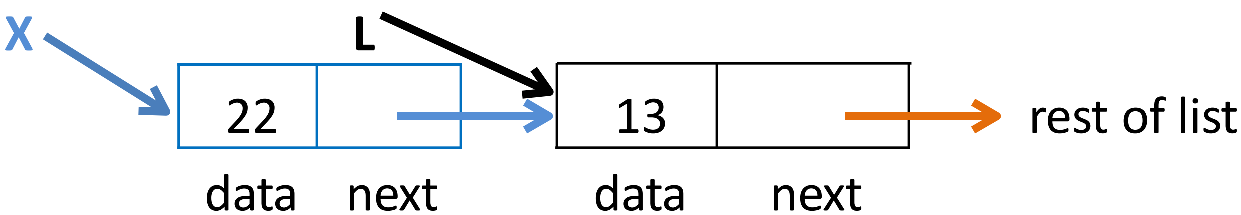
```
node *X = new (22, NULL);
```



Adding at the Front

- Now make the next point to the '13' node, L:

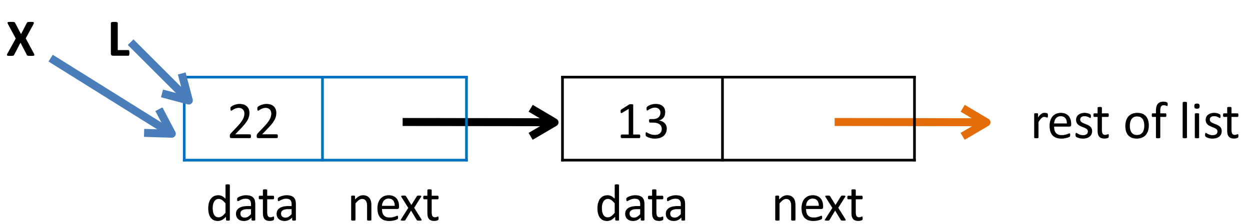
```
X->next = L;
```



- Finally, make L point to the new node, X:

```
L = X;
```

(note that X is no longer needed)

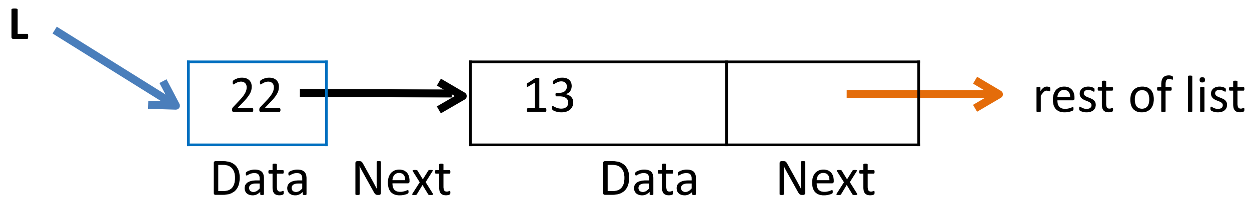


Adding at the Front

- Using these three steps:

```
node *X = new ( 22, NULL);
X->next = L;
L = X;
```

- we finish up with:



More linked list operations

ADDING A NODE AT THE END

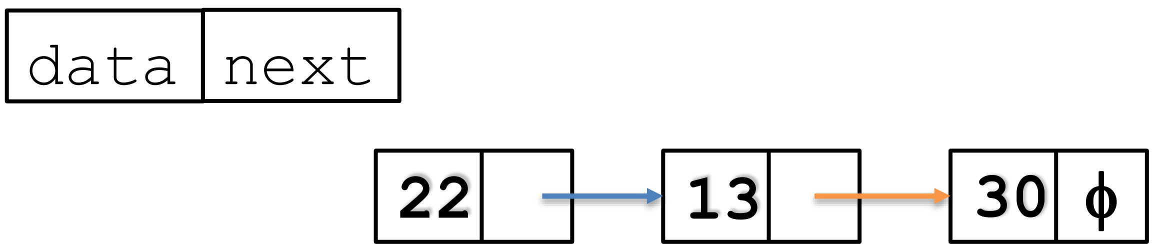
Linked Lists – Adding at the Back

- *Steps* :
  - `L` points to the *first* node of the list
  - Find the *last* node of the list, `z`
  - Create a new node
  - Make node `z->next` points to the new node

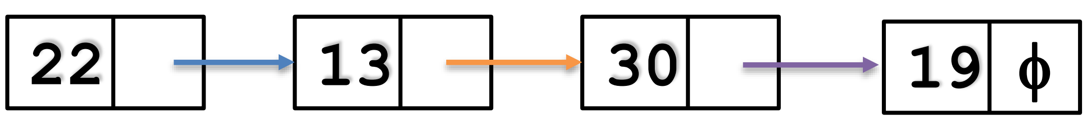
- To find the last node, we must *scan* the list

Adding a node at the end

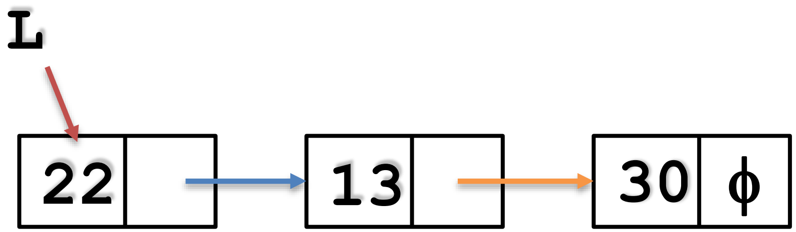
Here is our list at the beginning.

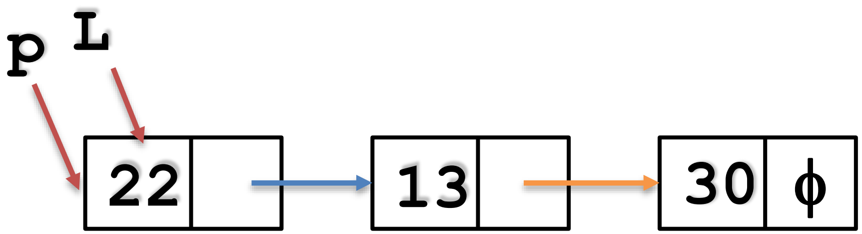


Here is our list after adding a new node containing **the value 19**

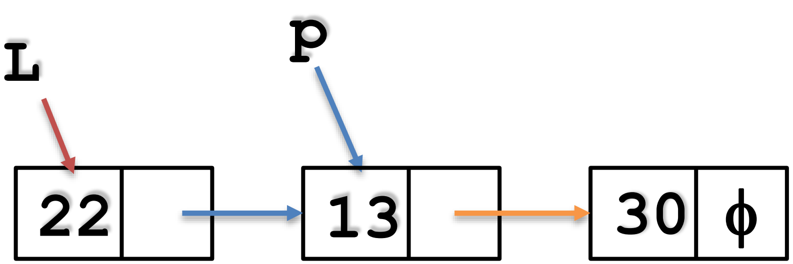


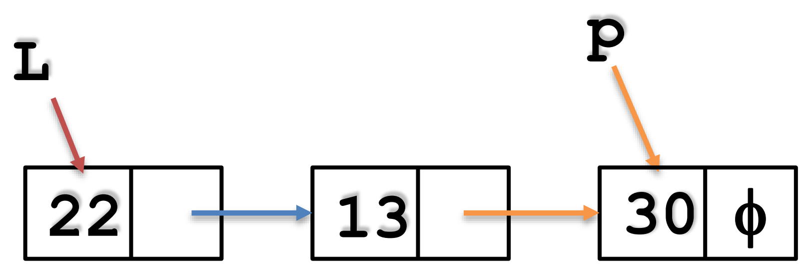
Adding a node at the end





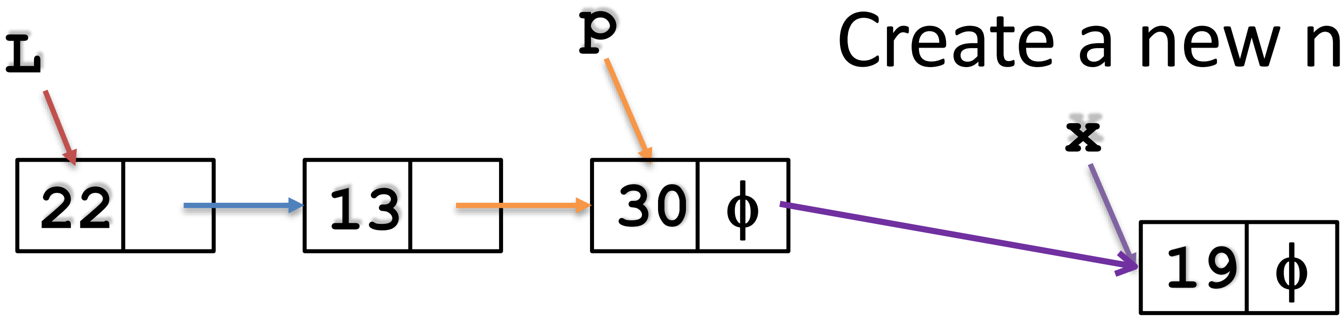
We need to move our "current element pointer" **p** to the last element in the list.



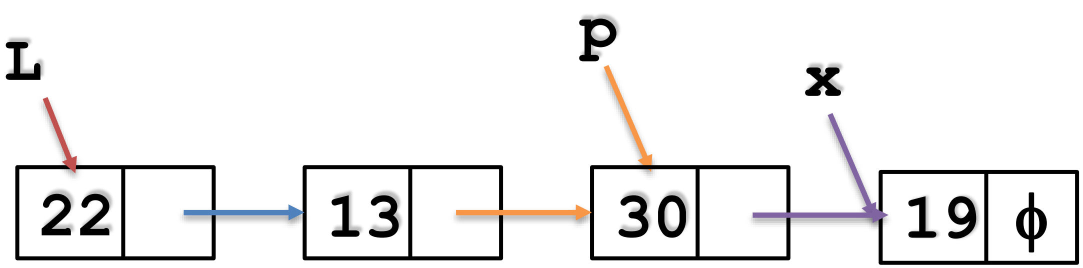


Adding a the node at end

Create a new node, **x**.



Make **p->next** point to **x**.

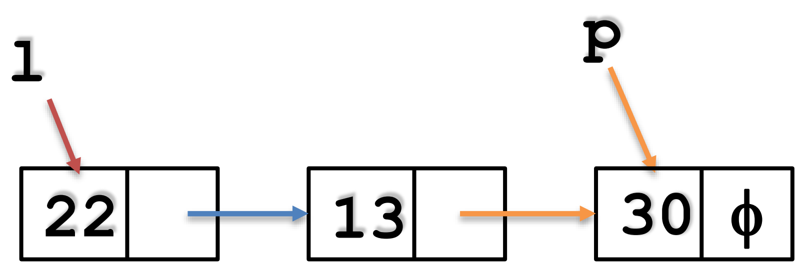


Scanning to the end of the list

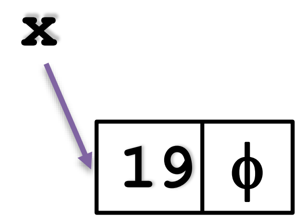
- Scan to the end of the list…

```
p = L; //initialise scan-pointer p
while ( p is not the last node ) //find last node
    p = p->next;
```

- Stops with **p** on last node of L.



Adding a node at the end



- To add 19 at the end
  - create new node:

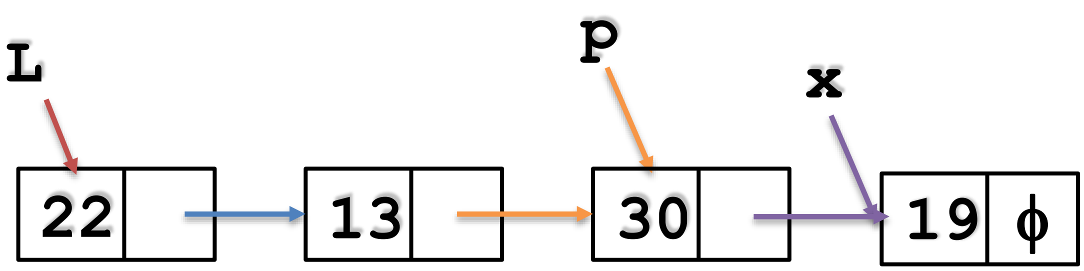
```
node *x = new(19, NULL);
```

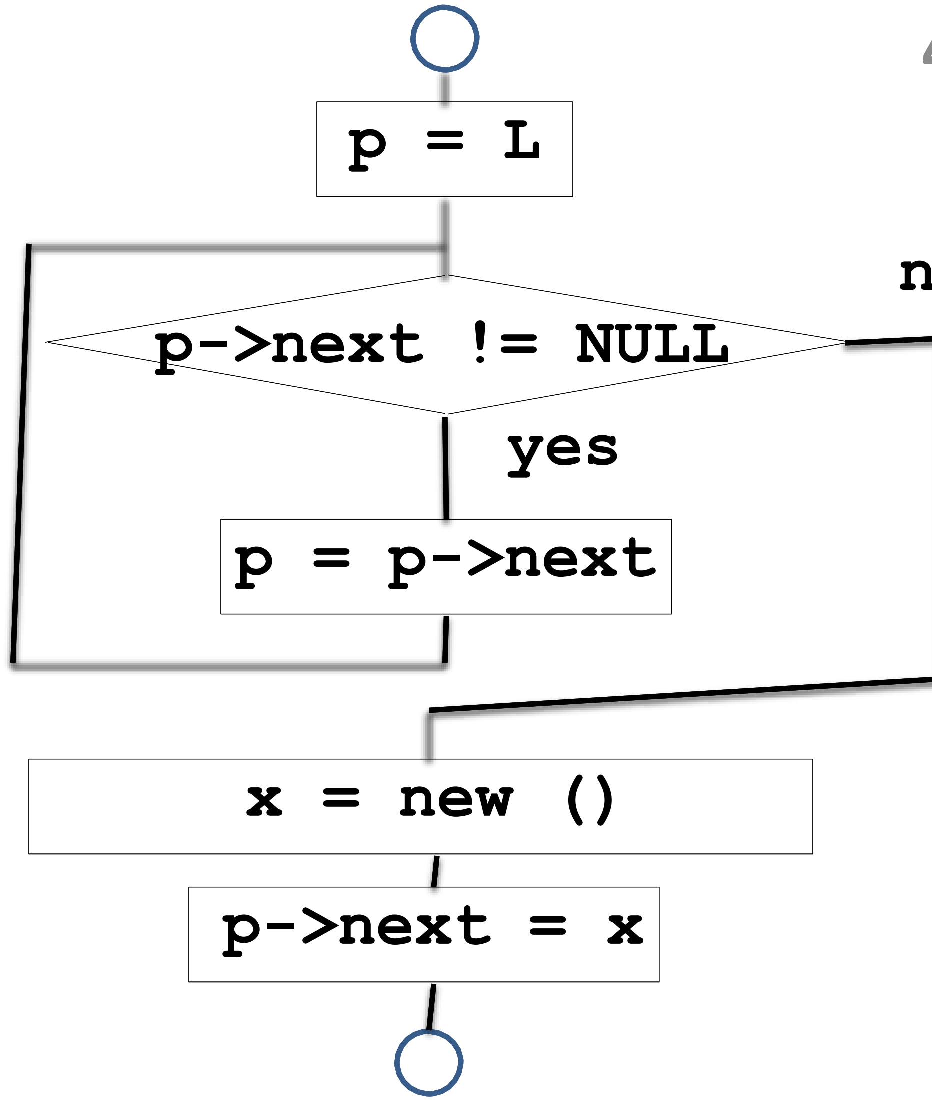
  - and attach it to **p**:

```
p->next = x;
```

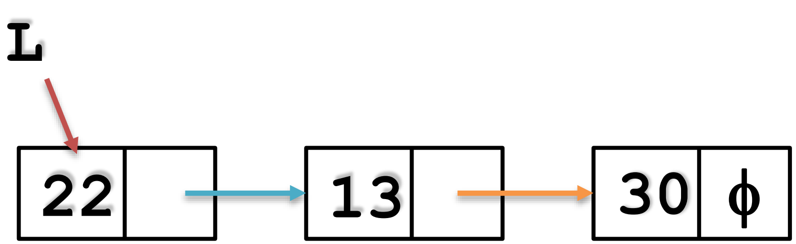
- or, we combine the last two operations:

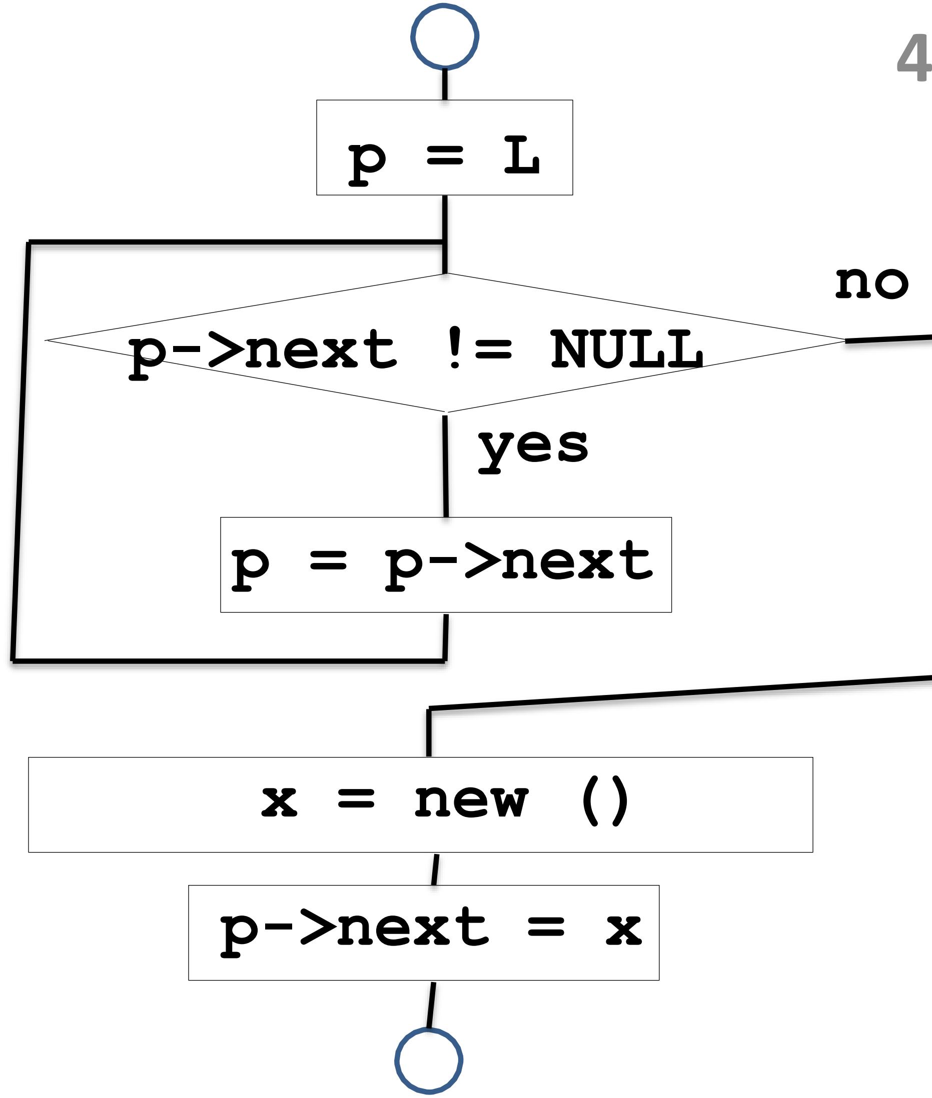
```
p->next = new (19, NULL);
```



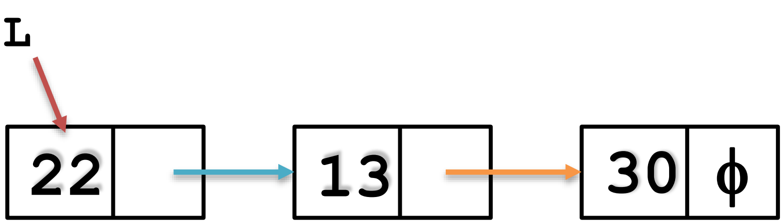



```
p = L;
while (p->next!= NULL)
      p = p->next;
node *x = new (19, NULL);
p->next = x;
```

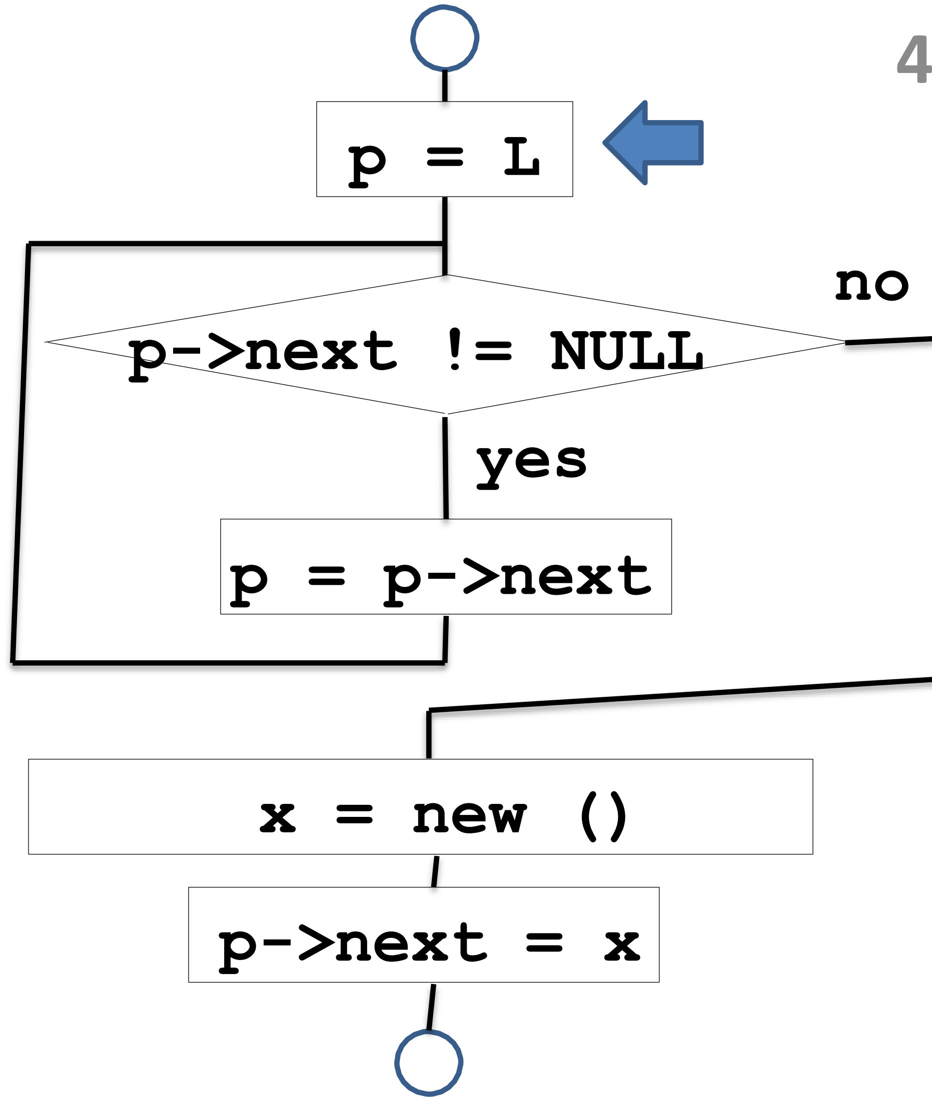


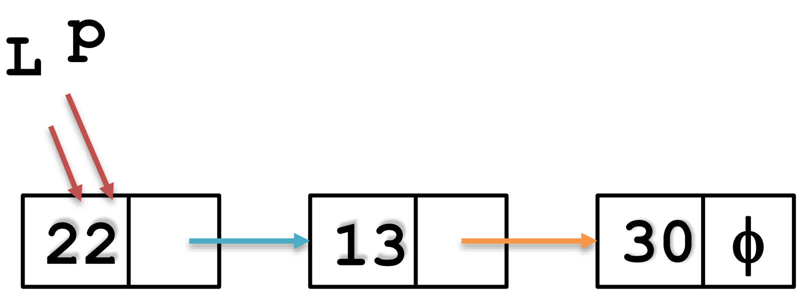


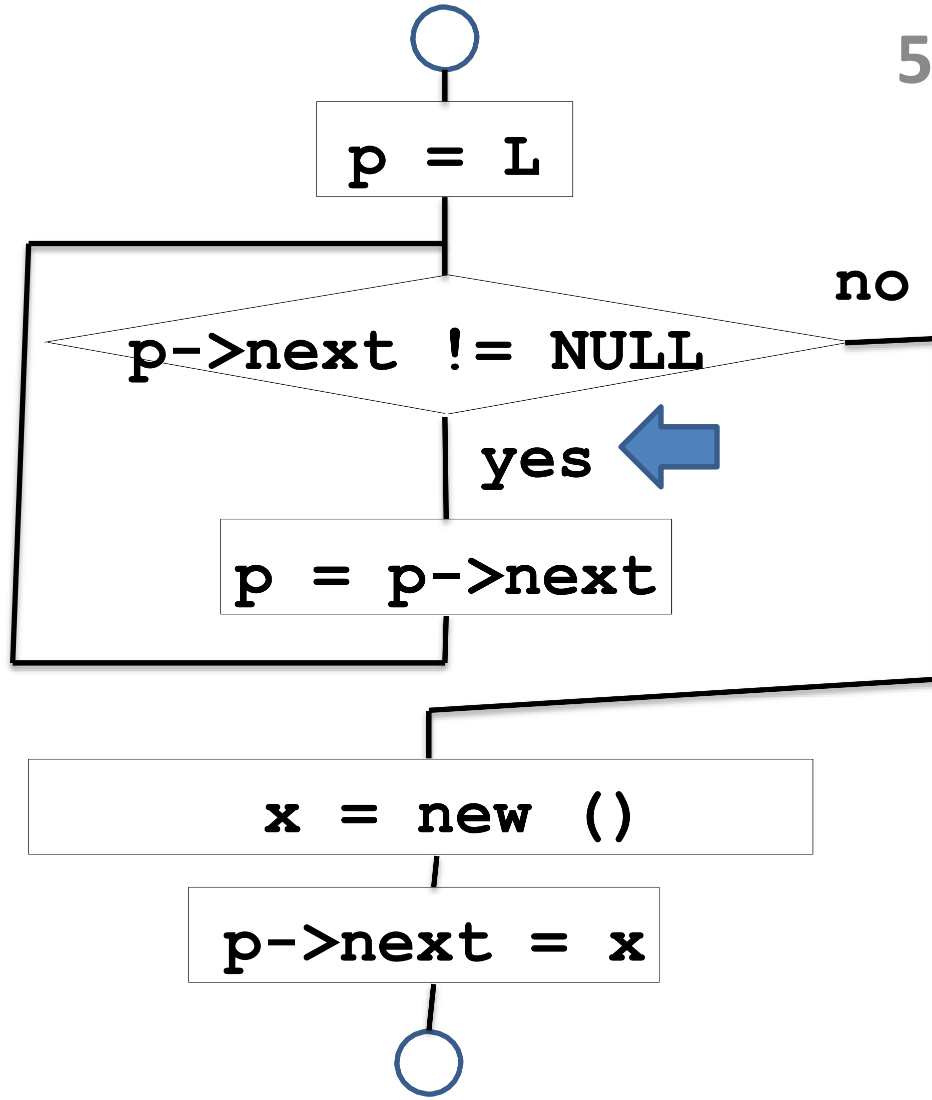
p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

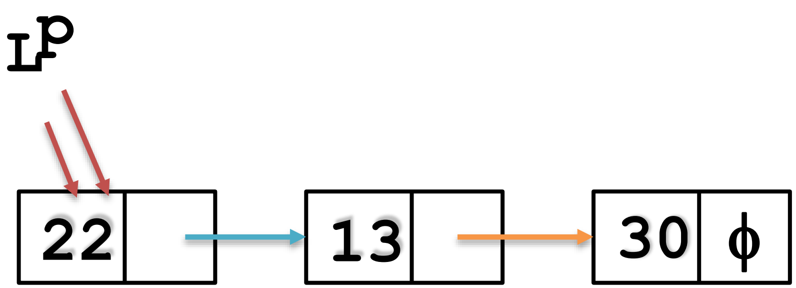


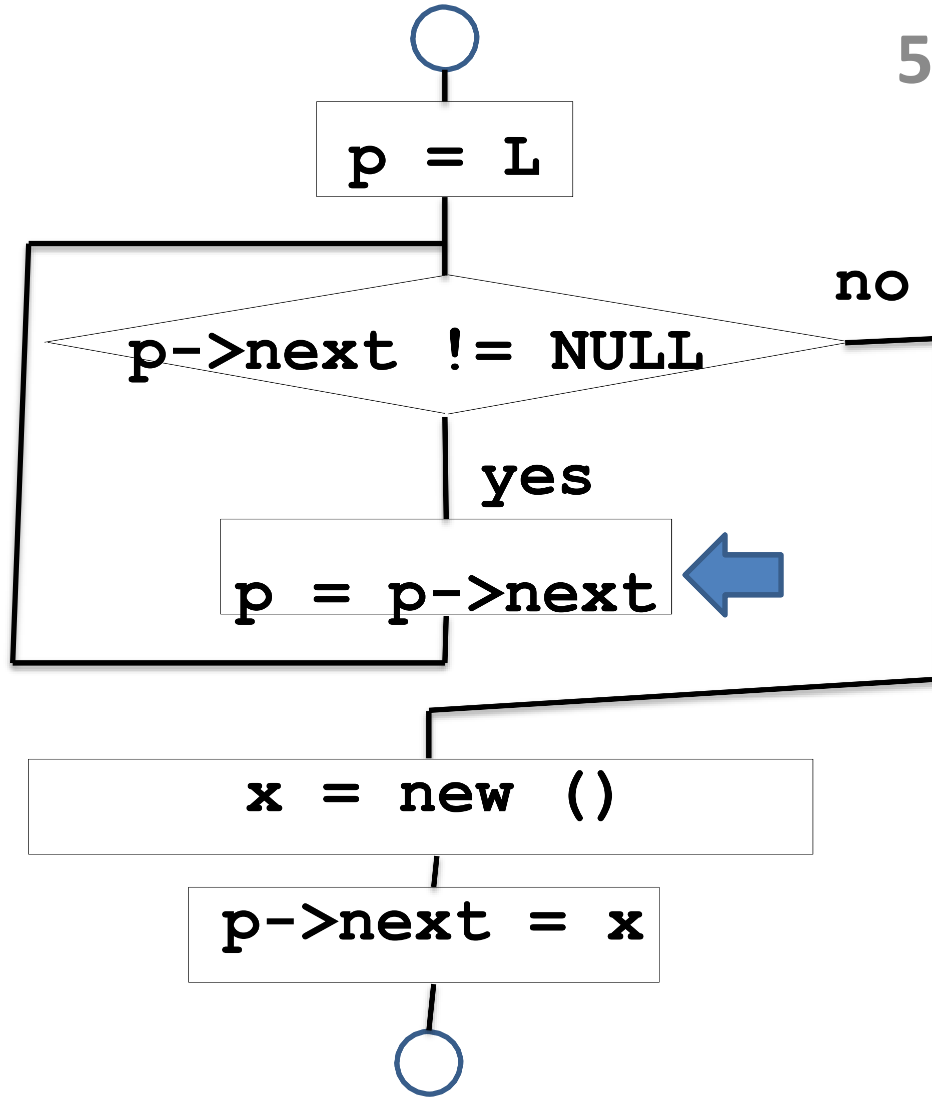
L
22
13
30
ϕ

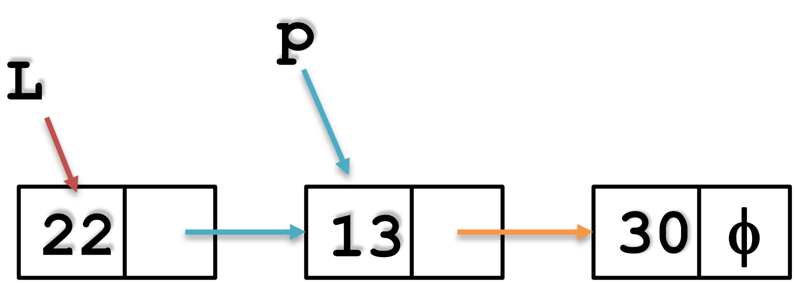

p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

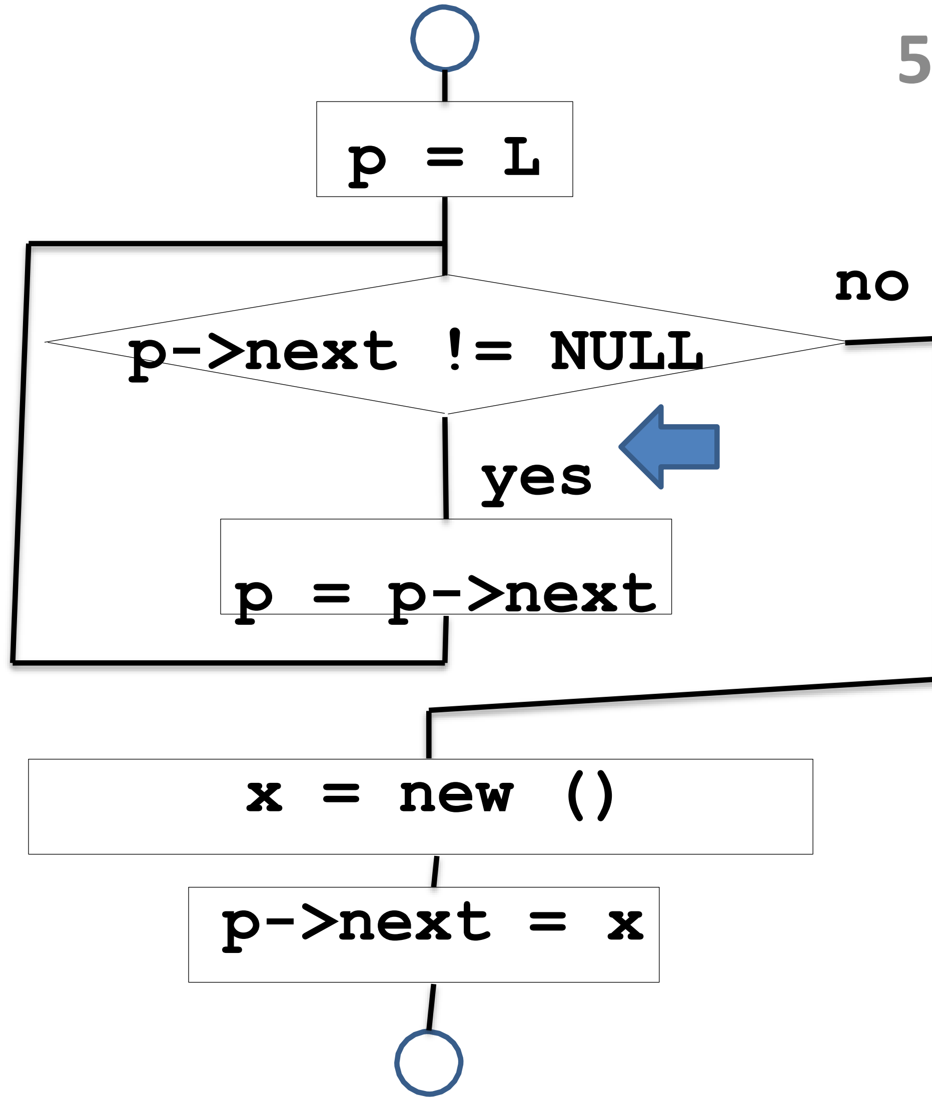

L
p
22
13
30
ϕ

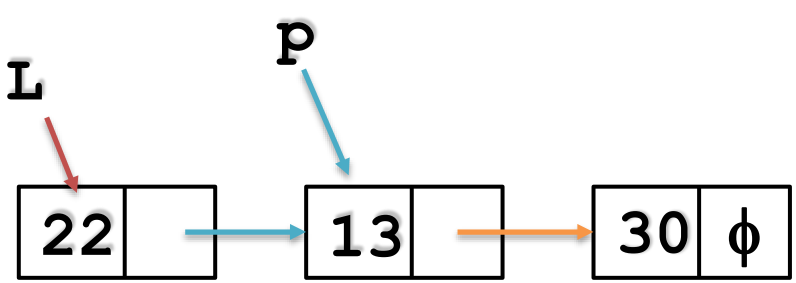

p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

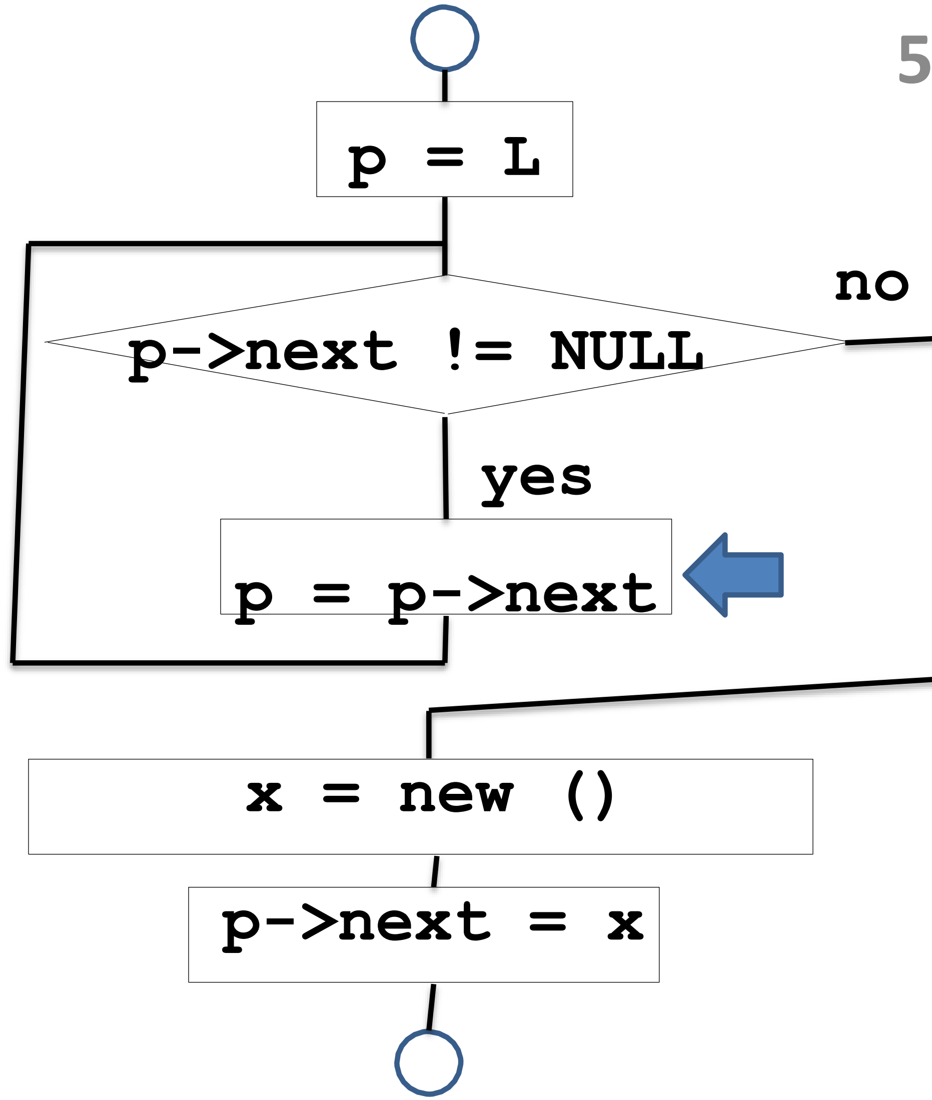

L
p
22
13
30
ϕ

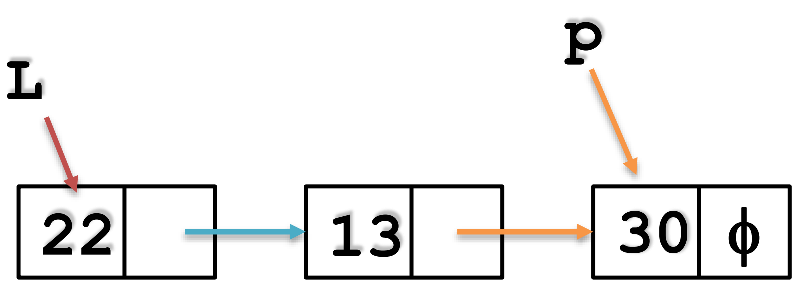

p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

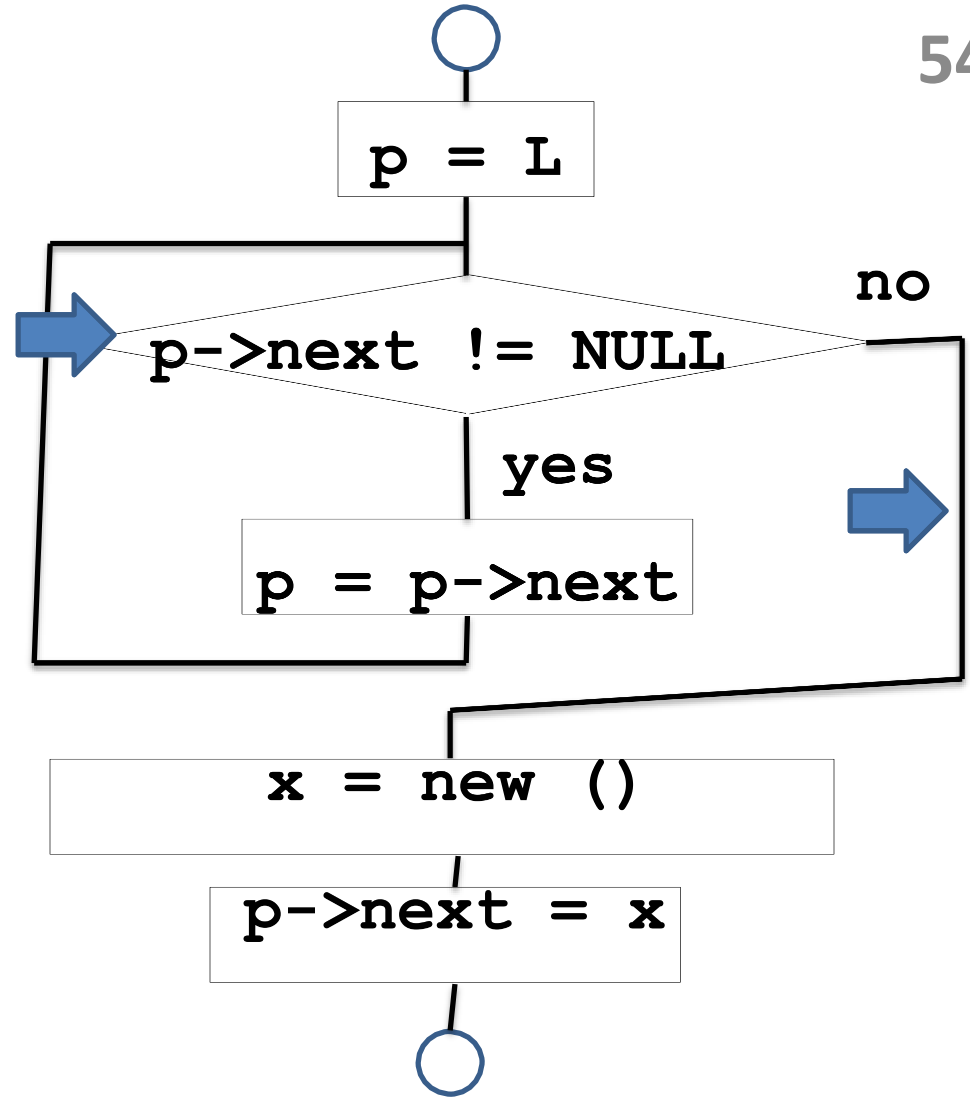

p
L
22
13
30
ϕ

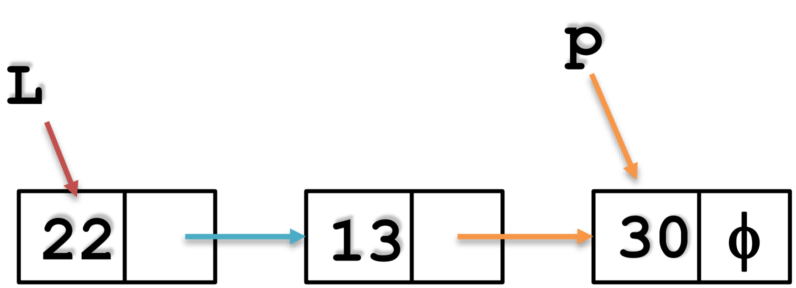

p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

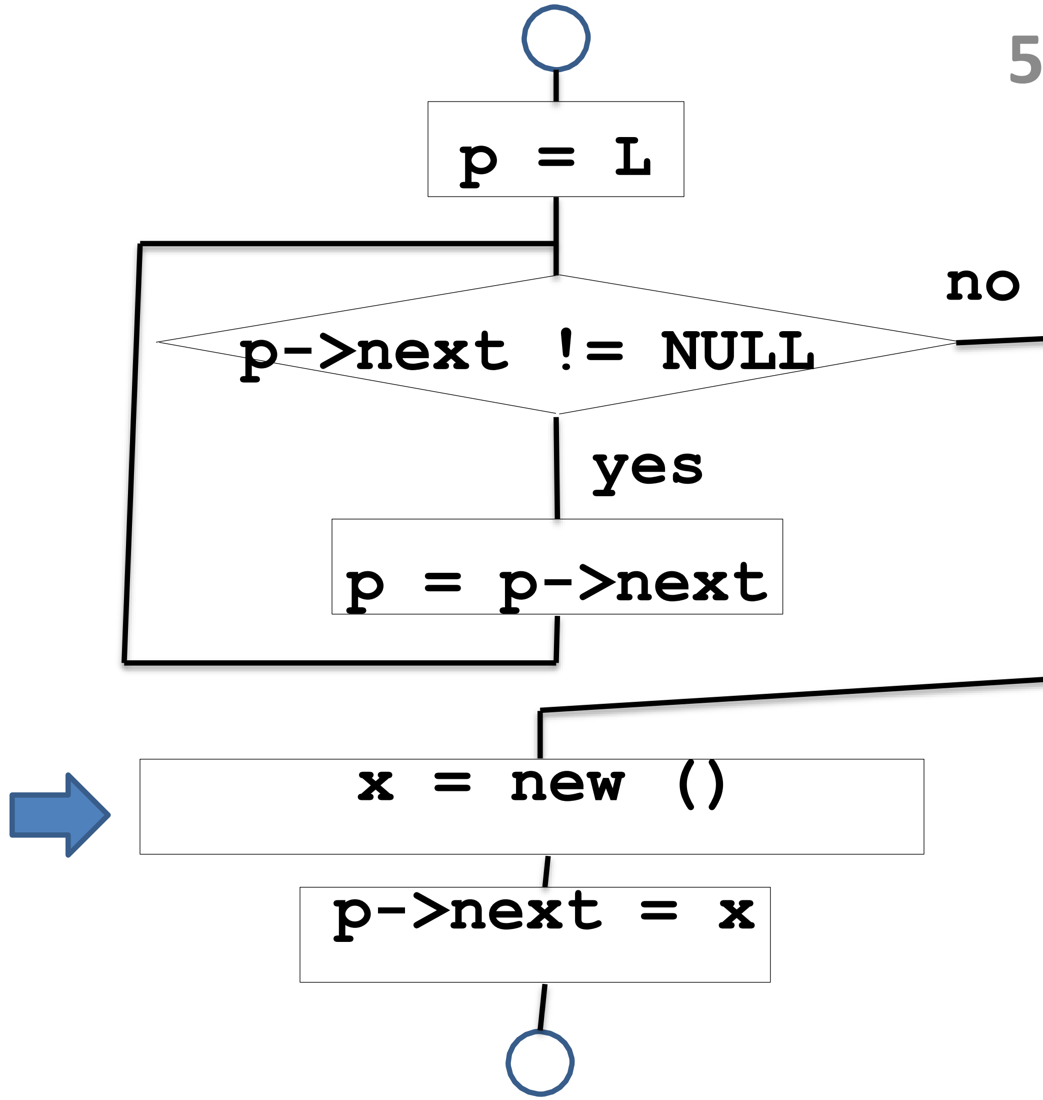

L
p
22
13
30
ϕ


p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

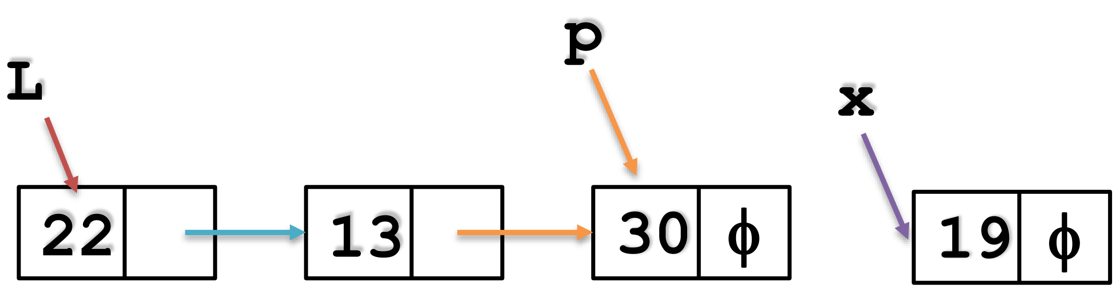

L
p
22
13
30
ϕ


p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x

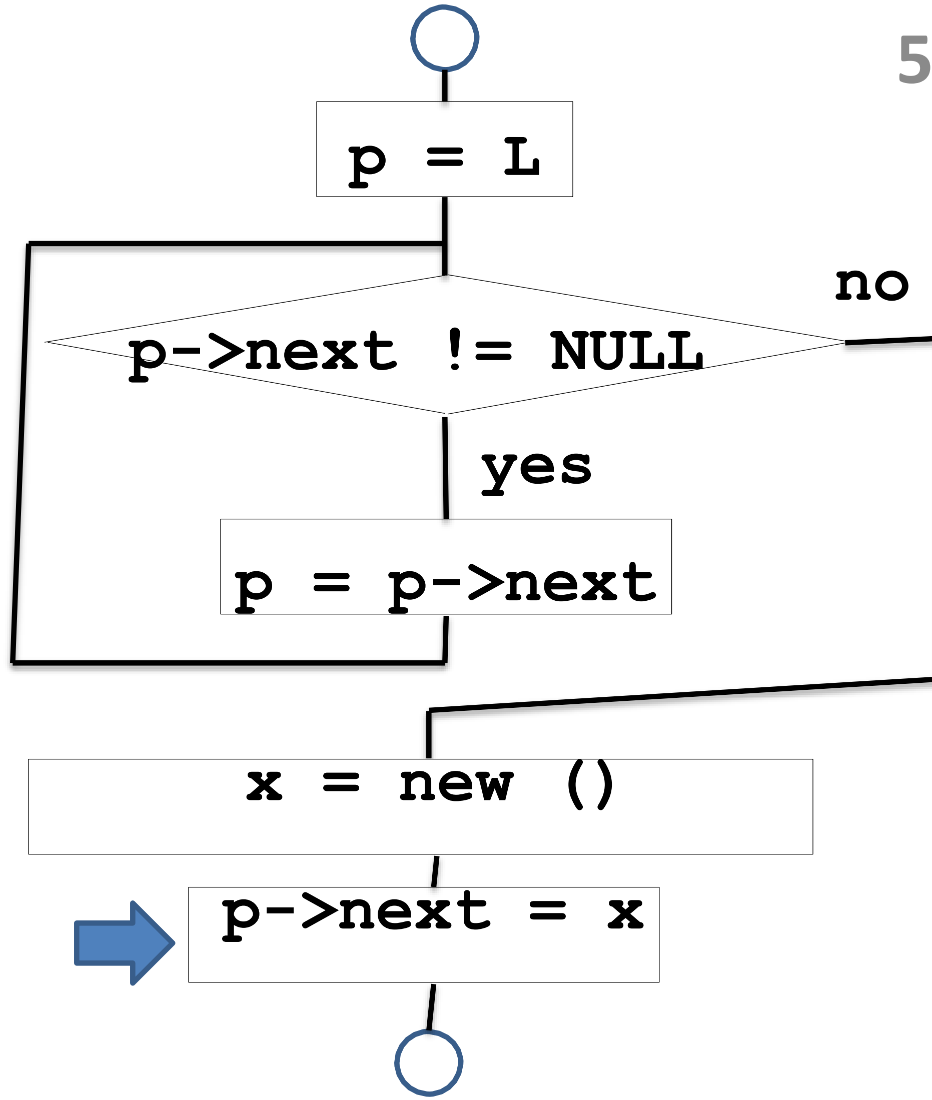

L
p
22
13
30
ϕ

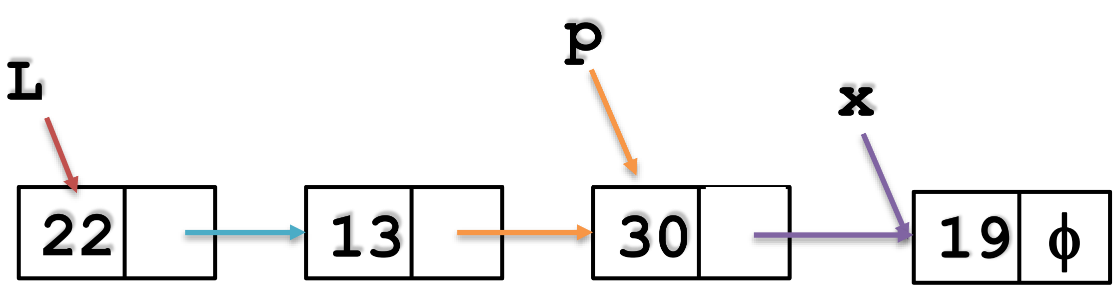


p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x



L
p
x
22
13
30
ϕ
19
ϕ


p = L
p->next != NULL
no
yes
p = p->next
x = new ()
p->next = x


L
p
x
22
13
30
19
ϕ

Adding node at End : The Whole Business

```
p =L; //initialise scan-pointer    p
while (p->next != NULL)//find last cell
   p = p->next;
p->next = new ( 19, NULL);
```

Quiz:

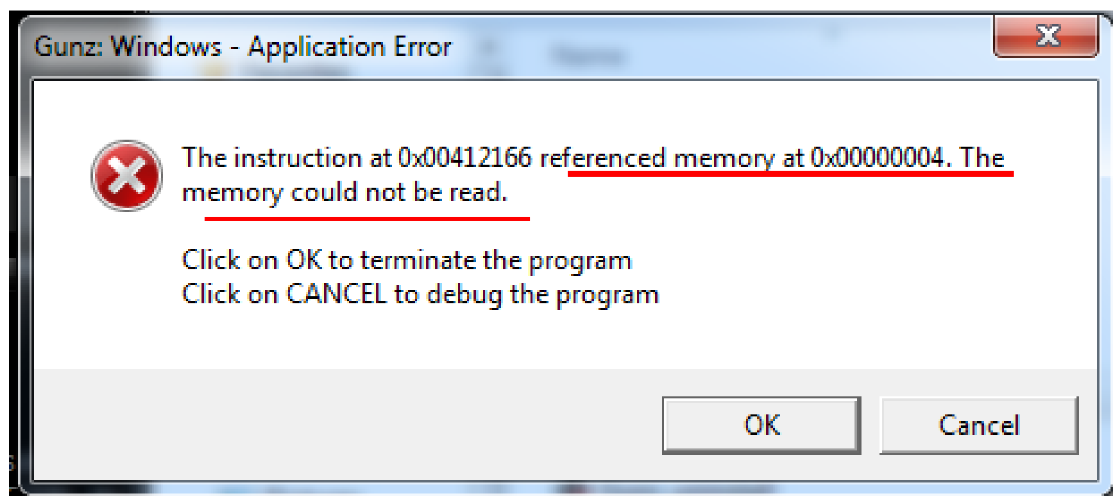
What would happen if **`L`** was NULL?

How should we adapt the code to cope?

Adding Cell at End : The Whole Business

- If **L** is NULL ….
- This generates a run-time error
  - Because there is no `p->next`

```
p = L;  //  initialise scan-pointer p
while (p->next != NULL) //  find last cell
   p = p->next;
p->next = …;
```



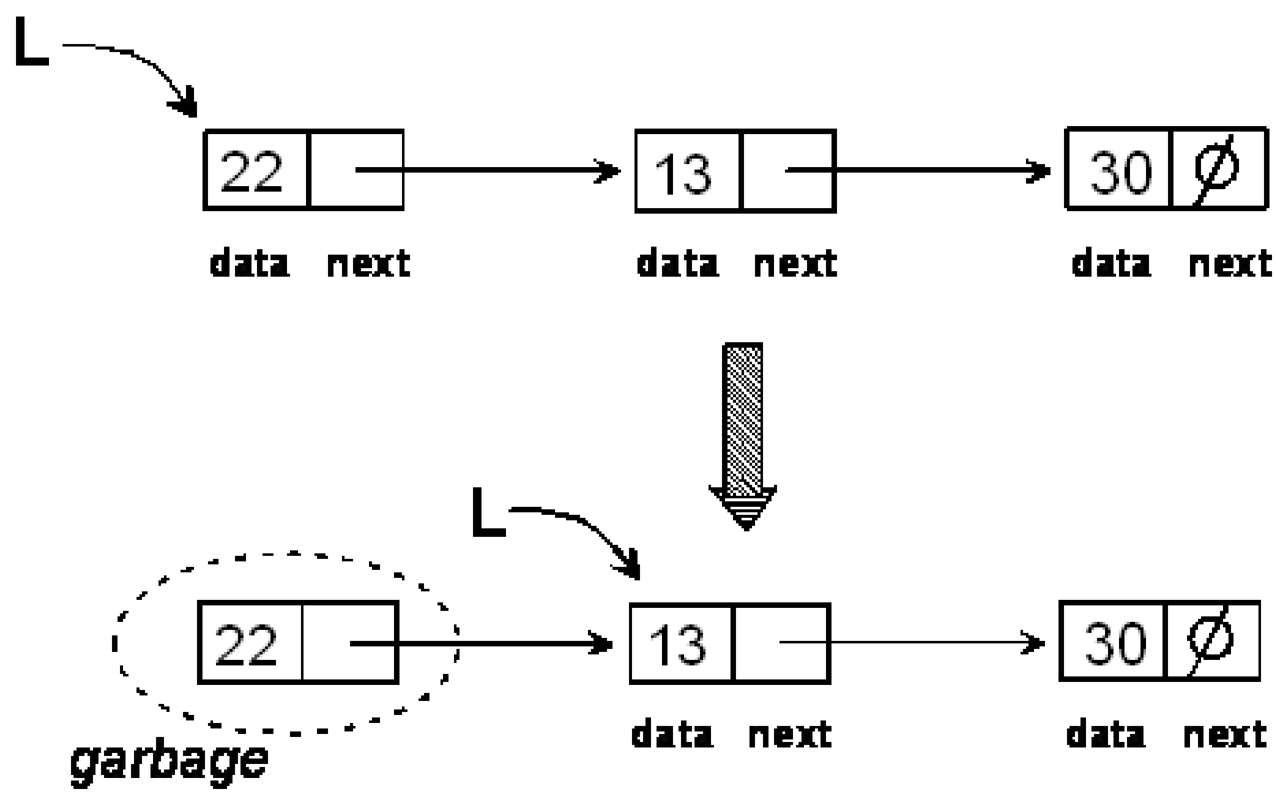
Adding Node at End : The Whole Business

```
if (L != NULL)
{
      p = L;
      while (p->next != NULL)
            p = p->next;
      p->next = new ( 19, NULL );
}
else {
      L = new ( 19, NULL);
}
```

REMOVING THE FIRST NODE

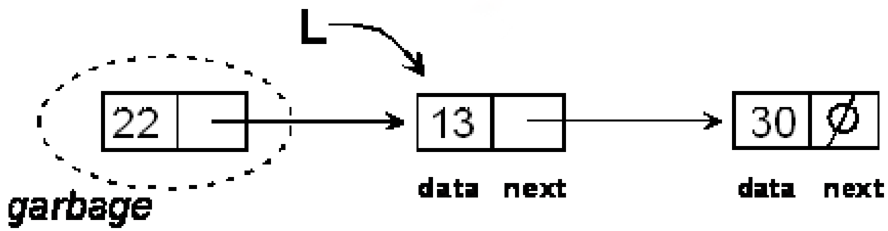
Removing the First Item

```
int x = L->data; // keep data value
L = L->next;     // delete first cell
return x;        // return result
```

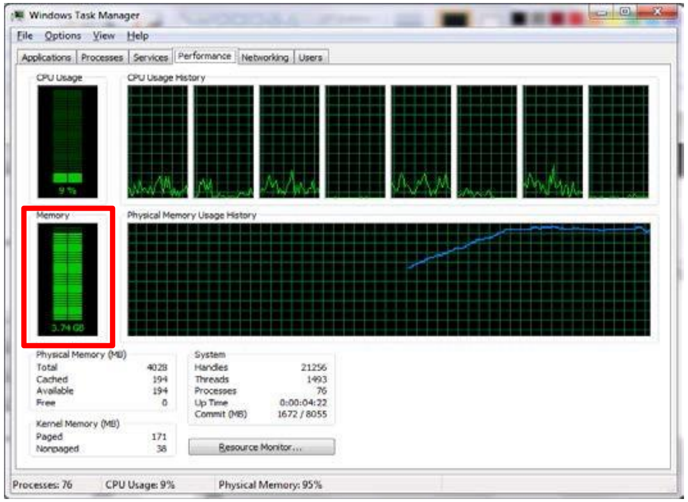


Afterwards, X holds the value 22.
The “garbage” cell can be
re-used later.

Why is it "garbage"?



- It's garbage because
  - Nothing is pointing at it anymore, and therefore
  - it can no longer be accessed (or used) by anyone
- It's just taking up space …
  - that can be reused later …

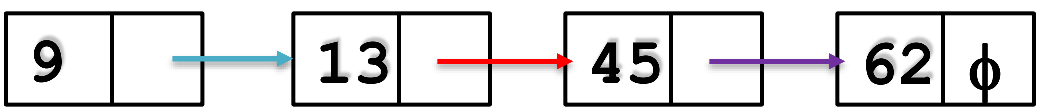


DELETE THE MIDDLE NODE FROM A LIST

Deleting node in middle

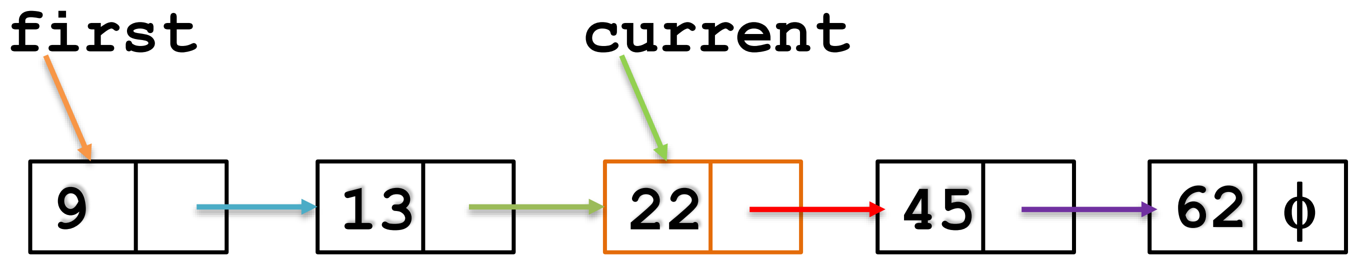
Here is our list at the beginning.

Here is our list after deletion.

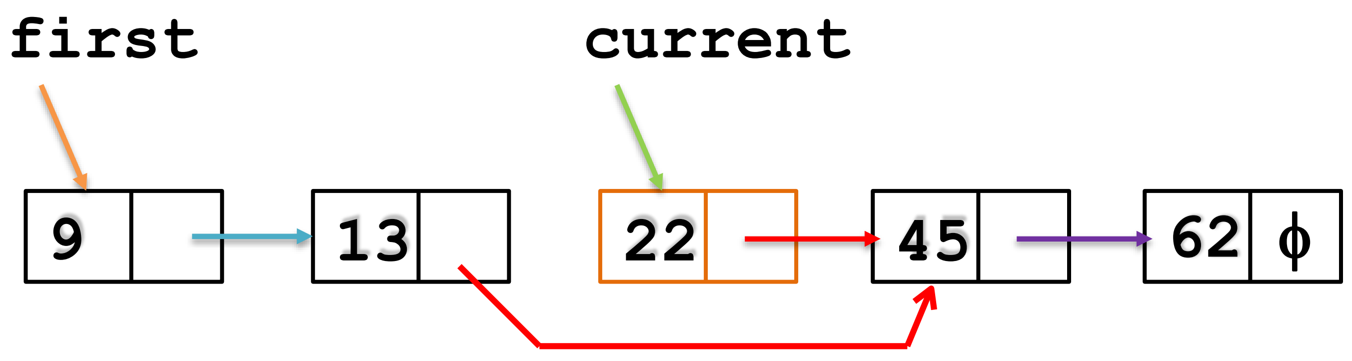


Deleting node in middle

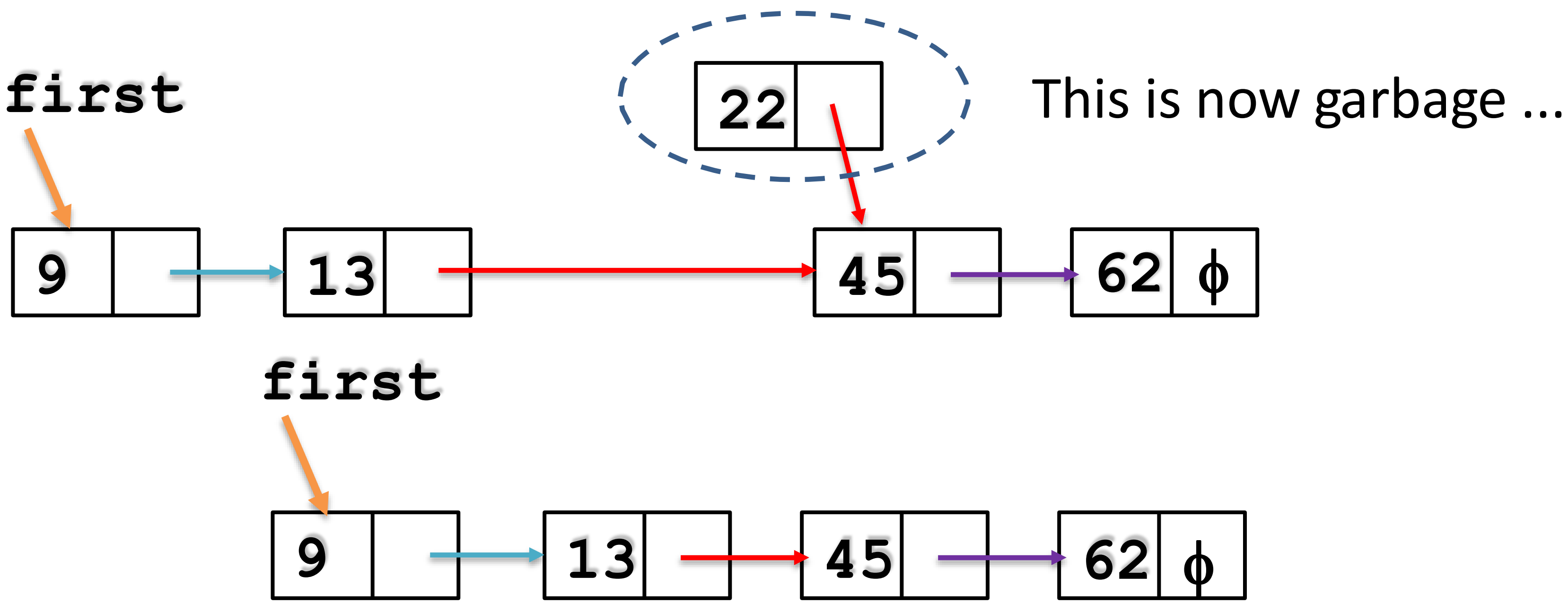
We need to find the node to be deleted.



We need to make the next pointer of the previous-to-current node point to the same thing as the **`current->next`** pointer.

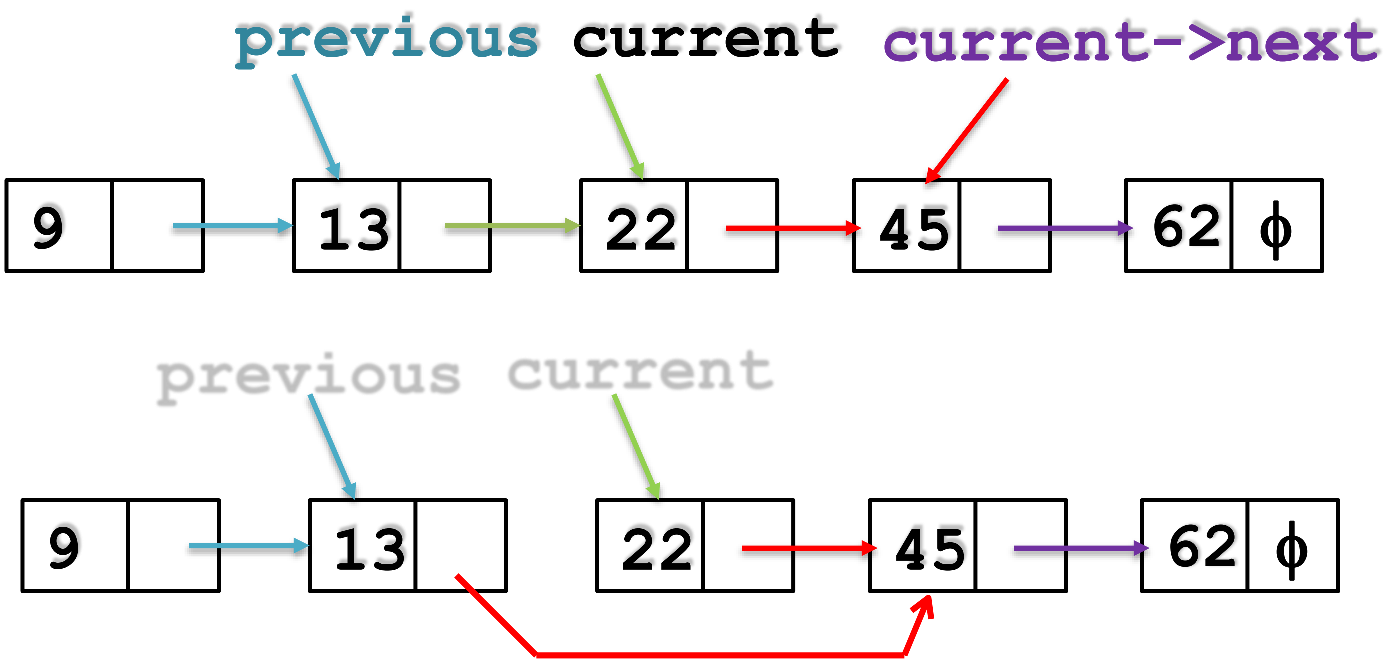


Deleting node in middle



Algorithm

- For our algorithm, not only do we need a **`current`** pointer, we need a "**`previous`**-to-current" pointer.
- **`previous->next = current->next;`**
  `current = NULL;`



Delete middle node from List


```
bool found = false;
node *current = first;
node *previous = NULL;

while (current != NULL)
{
        if (current->value == x) {
                found = true;
                break;
        }
        else {
              previous = current;
              current = current->next;
        }
}
if (found) {
        if (previous != NULL)
                previous->next = current->next;
        else
                first = current->next;
}
```

DOUBLY LINKED LISTS

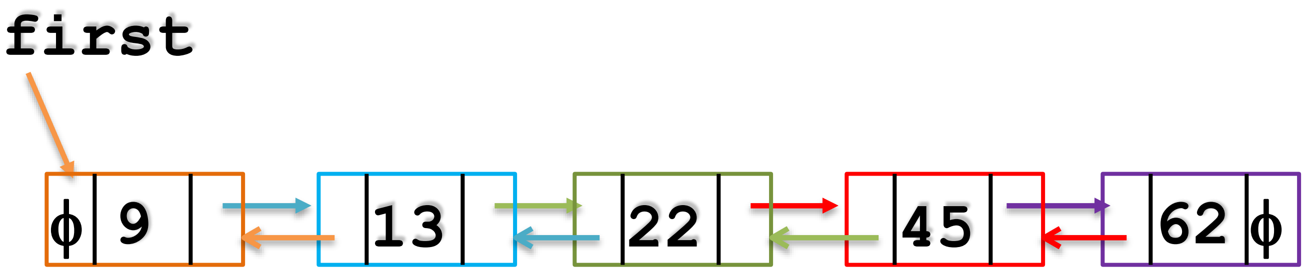
Multiple pointers

- If we can have one pointer (going forward) why can’t we have another pointer (going backward)?

```
typedef struct _node {
    struct _node *previous;
    int data;
    struct _node *next;
} doubleLL;
```

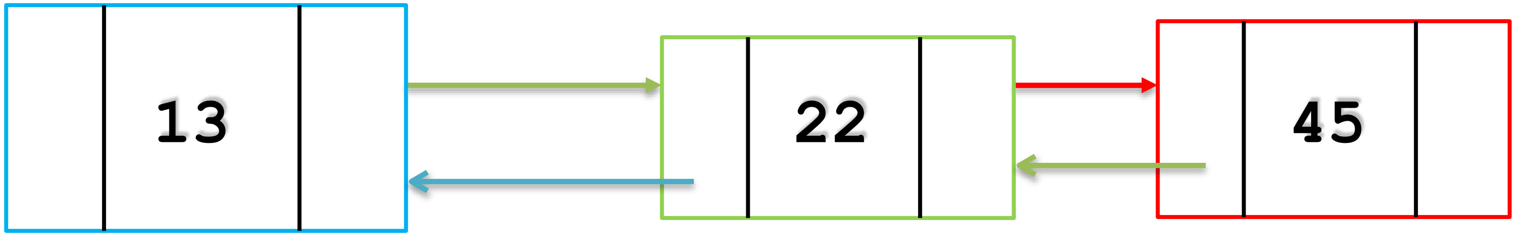
The structure of a doubly-linked node

`previous`	`value`	`next`



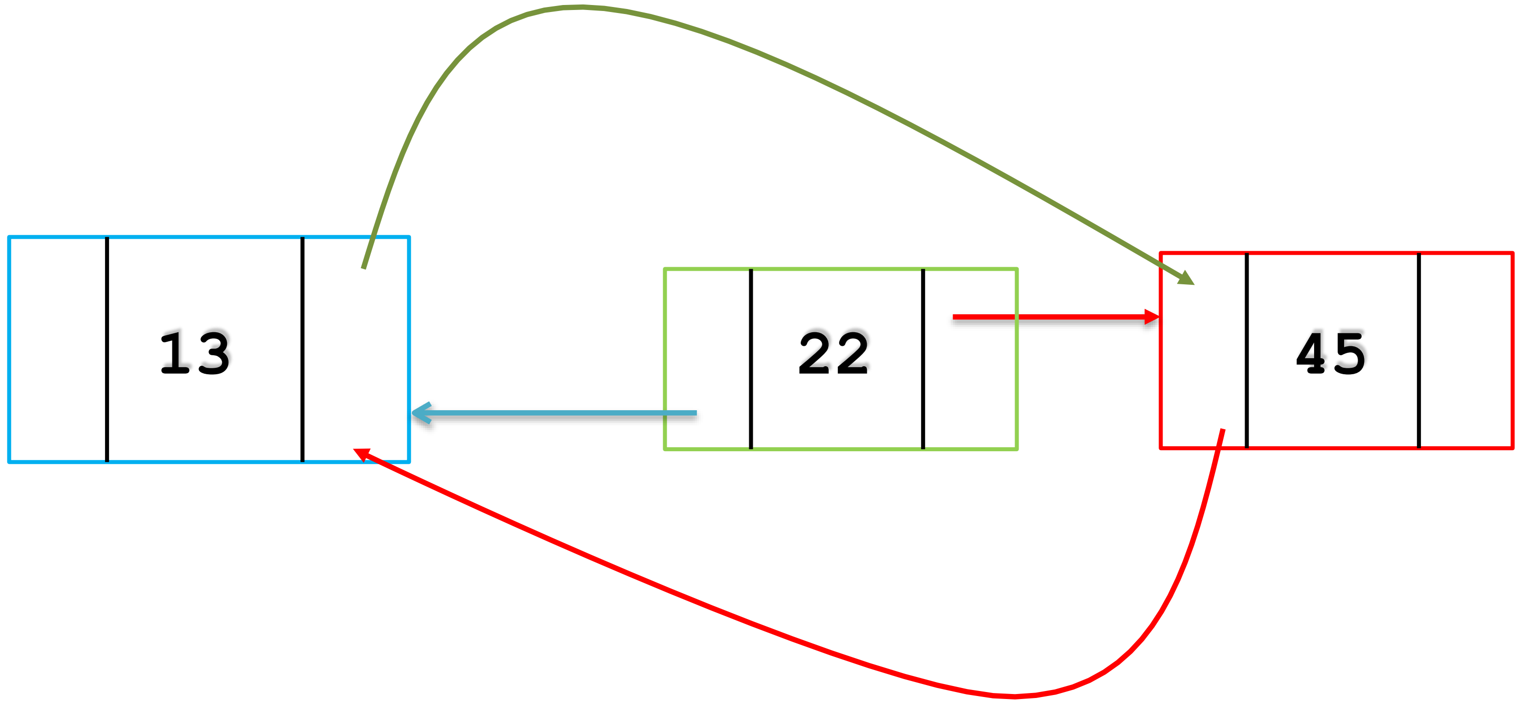
An example of a doubly linked list

Deleting a node from the middle



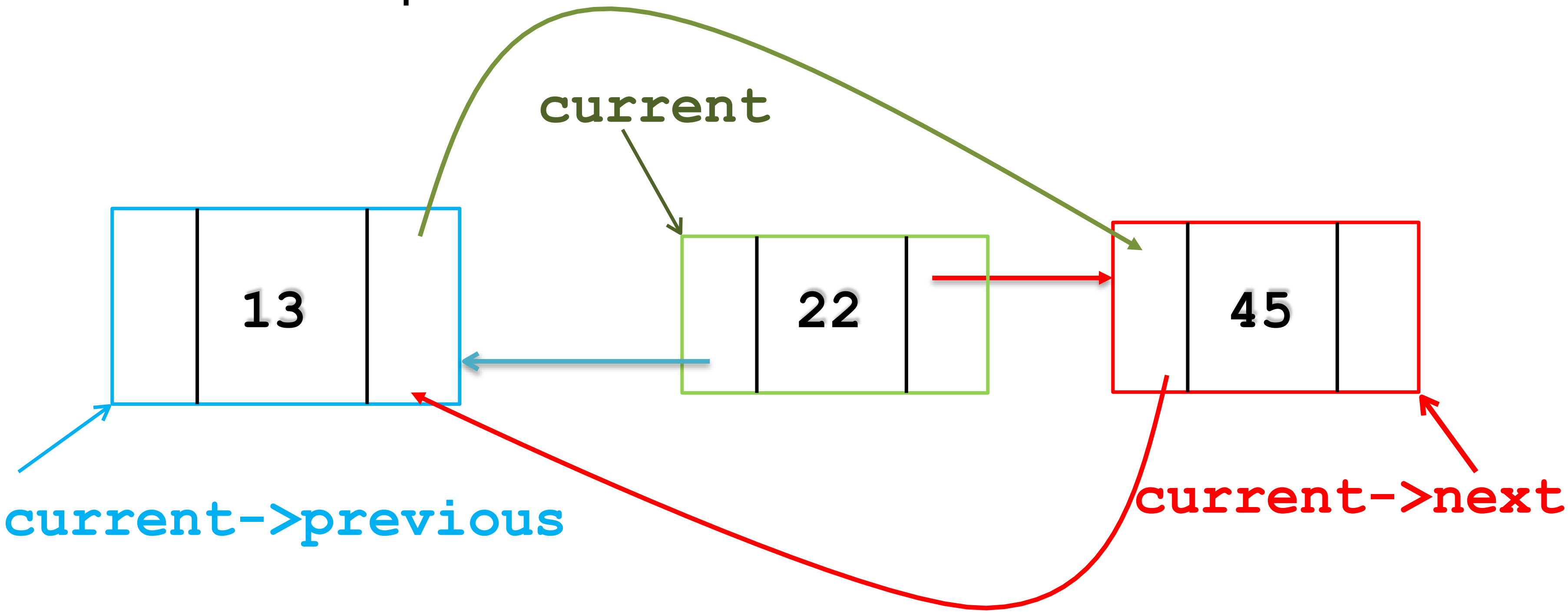
We want to delete 22 from the doubly linked list

Deleting a node from the middle



Deleting a node from the middle

Let `current` points to the node we want to delete




```
1. (current->previous)->next = current->next;

2. (current->next)->previous = current->previous;
```

```
doubleChainLL* deleteMiddle
              (node *first, int x)
{
    bool found = false;
    node *current = first;

    while (current!=NULL){
      if (current->data == x) {
              found = true;
              break;
      }
      else current = current->next;
    }

    if (found){
       current->previous->next = current->next;
       current->next->previous = current->previous;
       free(current);
    }
    return first;
}
```

Locating a node that contains value x

Update the doubly linked list

Exercise

a) Work through the example of the previous slide.
b) Implement them in C.

The only way to master linked list and pointers

c) In our "`deleteMiddle`" function, we haven't taken account of the special cases.
   i) what if the list is empty?
   ii) what if the node to be deleted is at the start of the list?
   iii) what if the node to be deleted is at the end of the list?

Add code to address these issues in C.

THE END