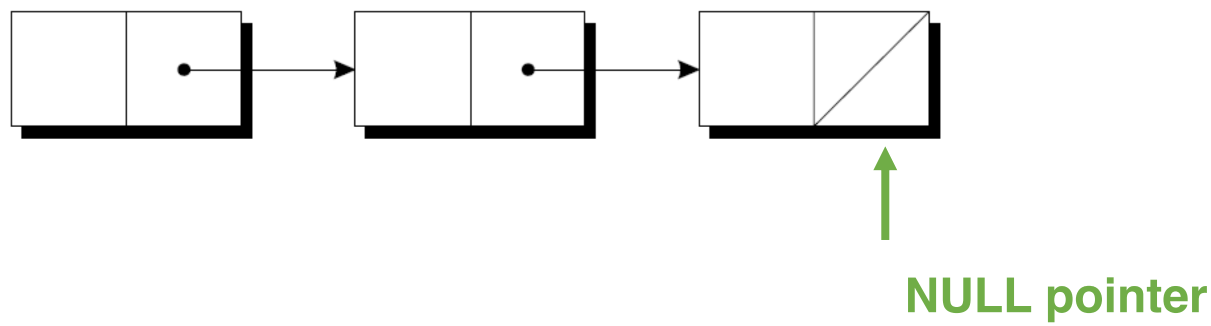
CS265
Advanced Programming Techniques

Linked Lists in C

Linked Lists

- A linked list consists of a chain of structures (called nodes), with each node containing a pointer to the next node in the chain:

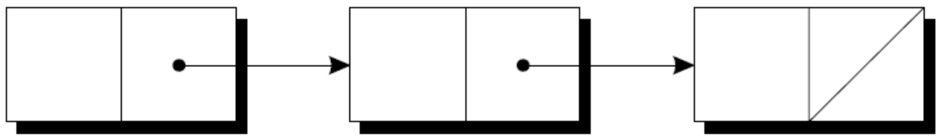


vs arrays

- More flexible
- More dynamic
- Not as fast

Defining a Node Type

- To create a linked list



- First, we need to define a structure that defines a single node

```
struct node {
        int value;           /* data stored in the node  */
        struct node *next;   /* pointer to the next node */
};
```

`node` must be defined to be a `struct` tag not a `typedef` name

Defining a Node Type

- Then, we need to define a variable that points to the first node

```
struct node *first = NULL;
```

initially the linked list is empty

Creating a Node

- As we construct a linked list, we'll create nodes one by one, adding each to the list.
- Steps involved in creating a node:
  1. Allocate memory for the node.
  2. Store data in the node.
  3. Insert the node into the list.
- We'll concentrate on the first two steps for now.

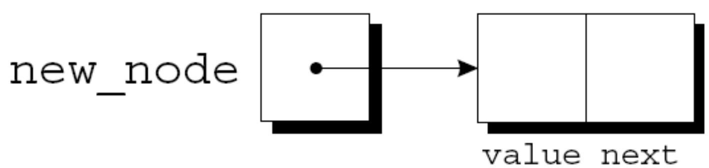
1. Allocate Memory for a Node

- When we create a node, we'll need a variable that can point to the node:

```
struct node *new_node;
```

- We'll use `malloc` to allocate memory for the new node

```
new_node = malloc(sizeof(struct node));
```



2. Store Data in the Node

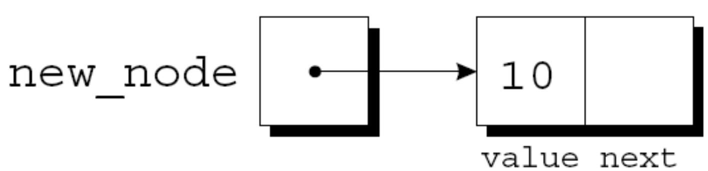
- Next, we'll store data in the `value` member of the new node using either

```
new_node->value = 10;
```

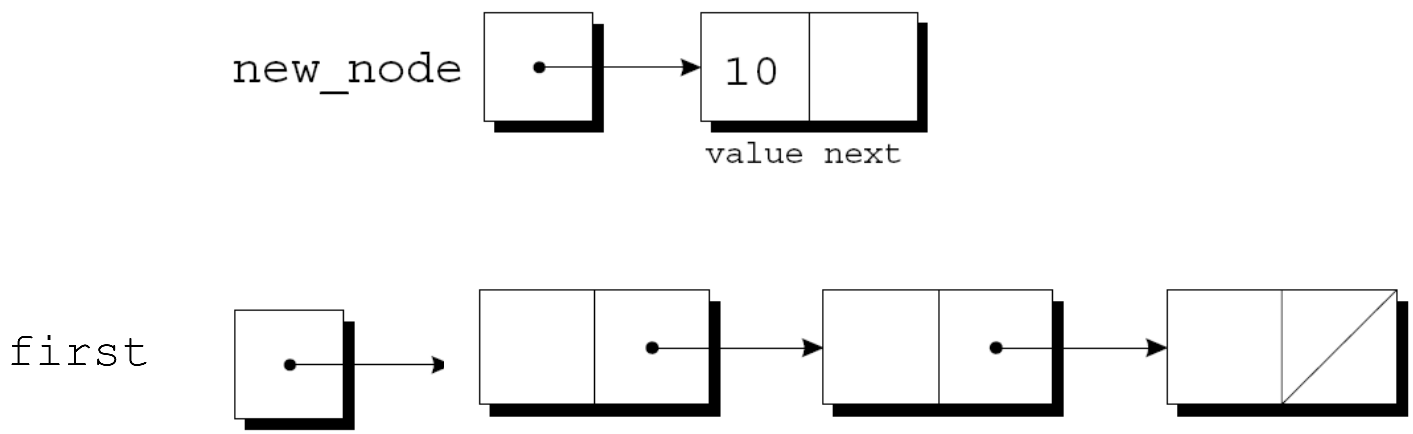
- We can also use

```
(*new_node).value = 10;
```

- Parenthesis are mandatory here because the . operator would otherwise take precedence over the * operator

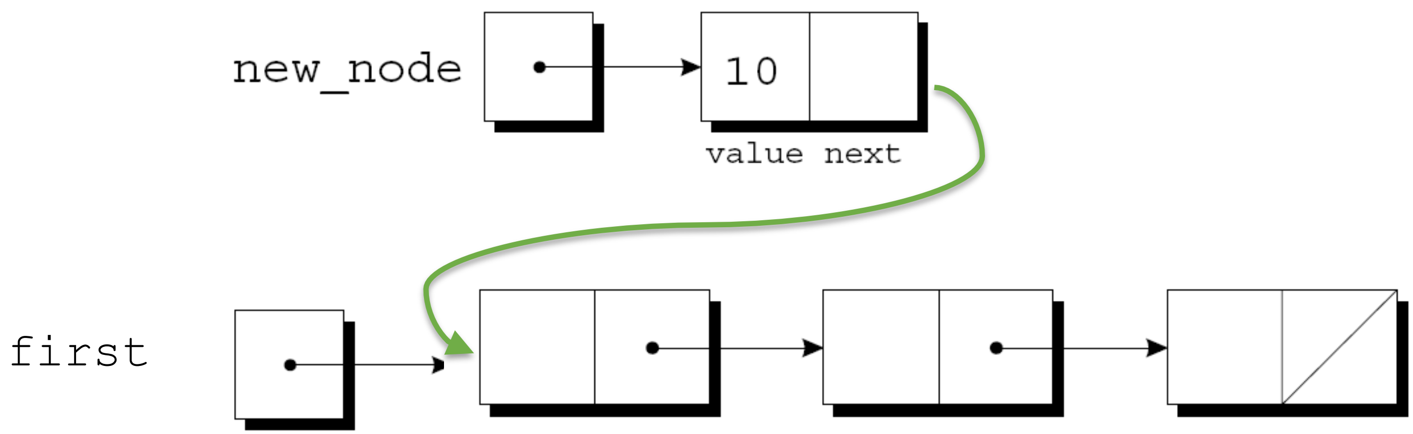


3. Insert a Node at the Beginning of a Linked List



- There are two steps needed to add the `new_node` at the beginning of the list pointed by `first`

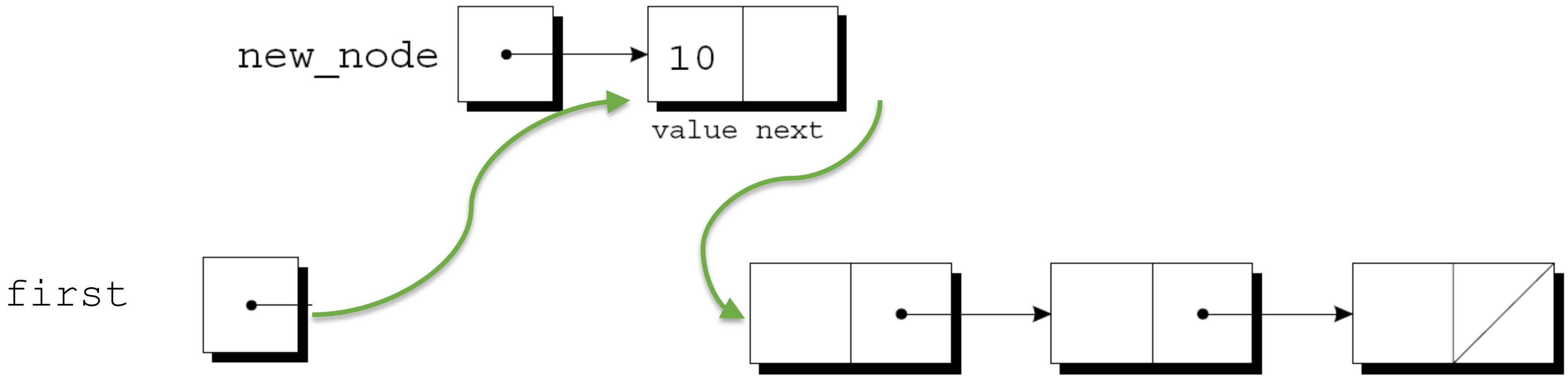
3. Insert a Node at the Beginning of a Linked List



- The first step is to modify the new node's `next` member to point to the node that was previously at the beginning of the list:

```
new_node->next = first;
```

3. Insert a Node at the Beginning of a Linked List



- The second step is to make `first` point to the new node:

```
first = new_node;
```

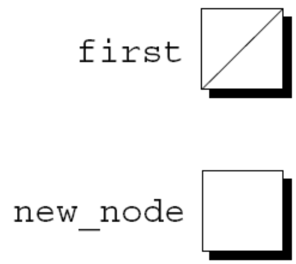
- These statements work even if the list is initially empty

Let’s see this again with an empty list

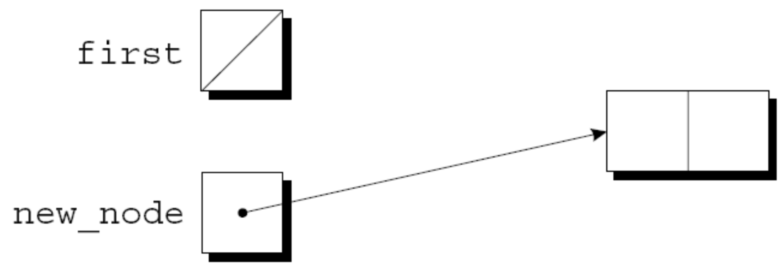
- Let’s trace the process of inserting two nodes into an empty list
- We’ll insert a node containing the number 10 first, followed by a node containing the number 20

Inserting a Node at the Beginning of a Linked List

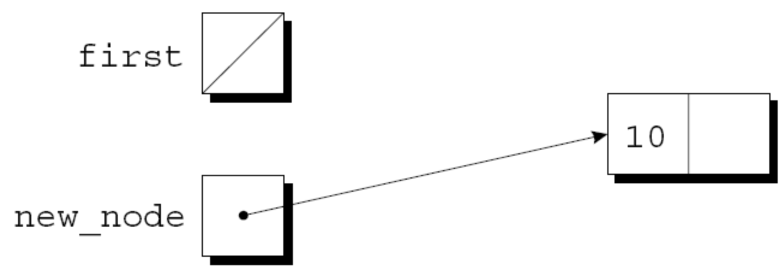
```
first = NULL;
```




```
new_node =
  malloc(sizeof(struct node));
```

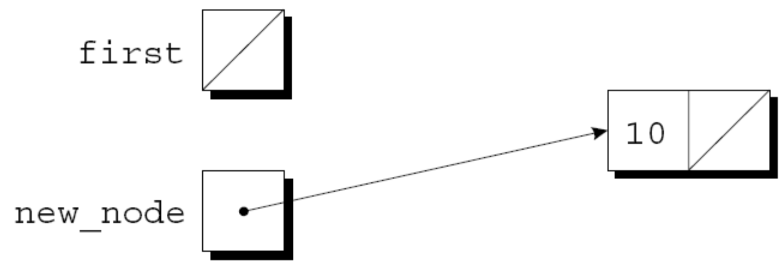



```
new_node->value = 10;
```

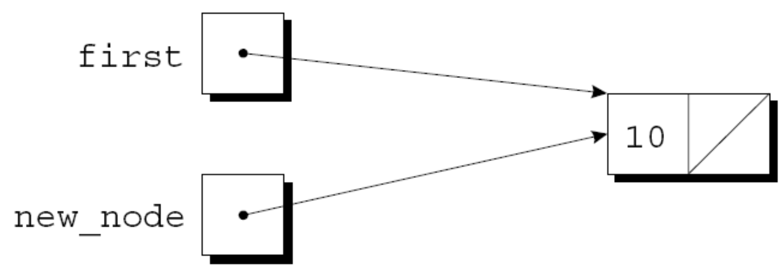


Inserting a Node at the Beginning of a Linked List

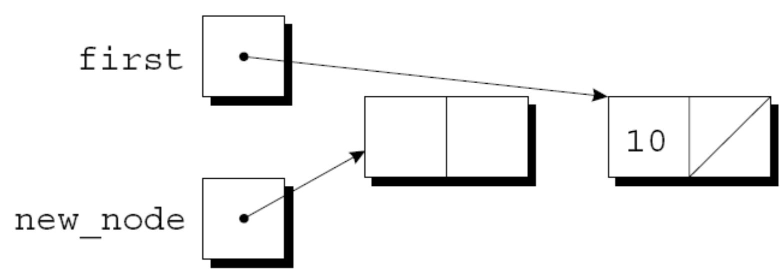
```
new_node->next = first;
```




```
first = new_node;
```

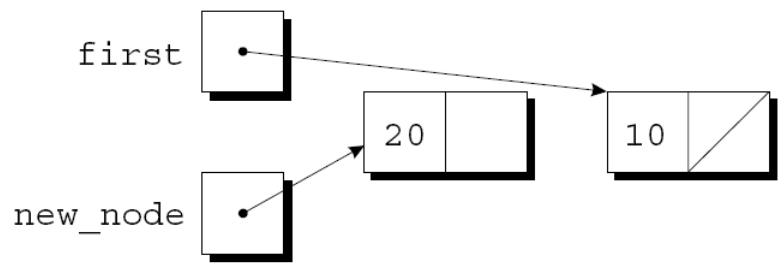



```
new_node =
  malloc(sizeof(struct node));
```

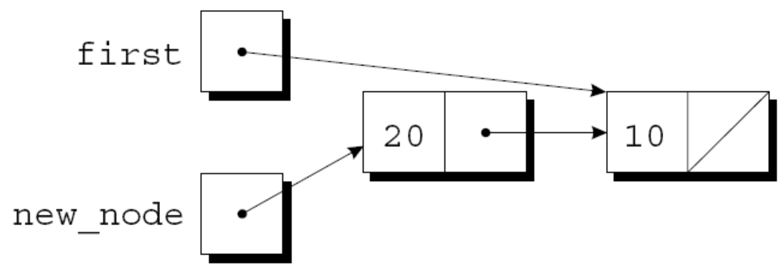


Inserting a Node at the Beginning of a Linked List

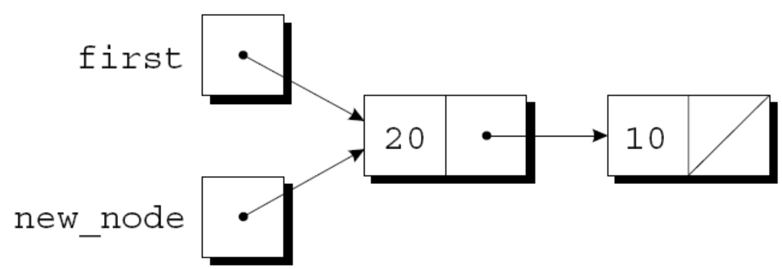
```
new_node->value = 20;
```




```
new_node->next = first;
```




```
first = new_node;
```



Inserting a Node at the Beginning of a Linked List

- A function that inserts a node containing `n` into a linked list, which is pointed to by `list`:

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Searching a Linked List

- Although a `while` loop can be used to search a list, the `for` statement is often superior.
- A loop that visits the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

```
for (p = first; p != NULL; p = p->next)
  …
```

- A loop of this form can be used in a function that searches a list for an integer `n`.

Searching a Linked List

- If it finds `n`, the function will return a pointer to the node containing `n`; otherwise, it will return a null pointer.
- An initial version of the function:

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Searching a Linked List

- There are many other ways to write `search_list`.
- One alternative is to eliminate the `p` variable, instead using `list` itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

- Since `list` is a copy of the original list pointer, there's no harm in changing it within the function.

Searching a Linked List

- Another alternative:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n;
         list = list->next)
    ;
  return list;
}
```

- Since list is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`.

Searching a Linked List

- This version of `search_list` might be a bit clearer if we used a `while` statement:

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

Deleting a Node from a Linked List

- A big advantage of storing data in a linked list is that we can easily delete nodes.
- Deleting a node involves three steps:
  1. Locate the node to be deleted.
  2. Alter the previous node so that it "bypasses" the deleted node.
  3. Call `free` to reclaim the space occupied by the deleted node.
- Step 1 is harder than it looks, because step 2 requires changing the *previous* node.
- There are various solutions to this problem.

Deleting a Node from a Linked List

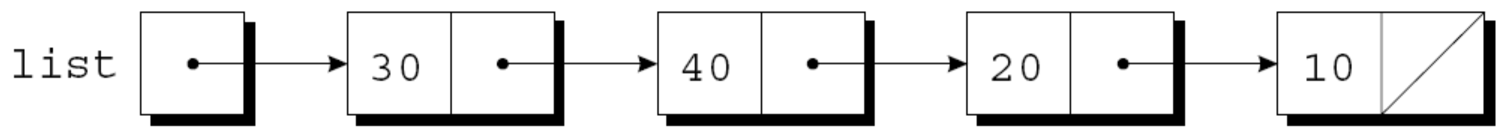
- The “trailing pointer” technique involves keeping a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`).
- Assume that `list` points to the list to be searched and `n` is the integer to be deleted.
- A loop that implements step 1:

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

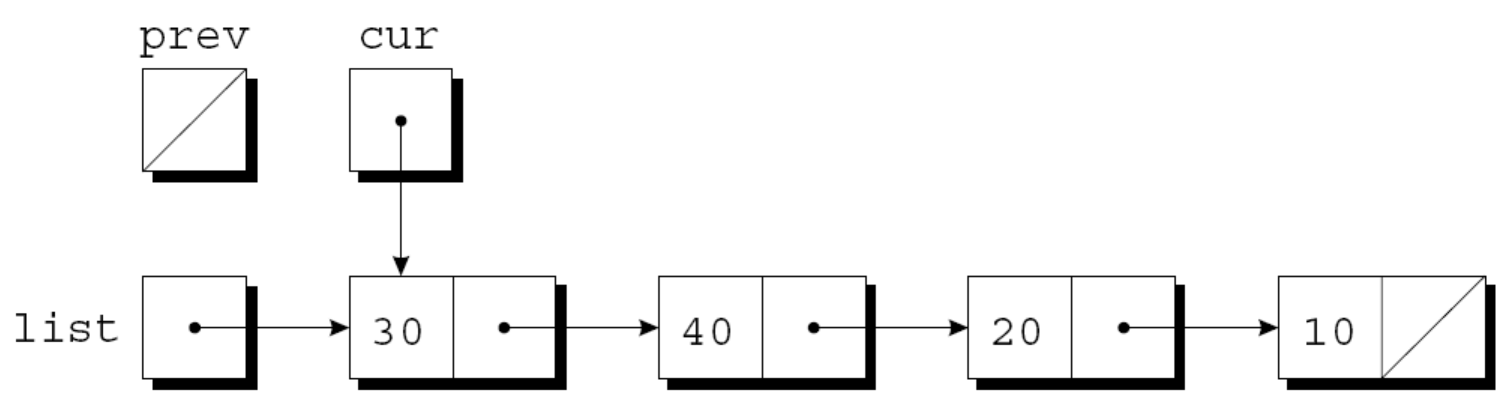
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node.

Deleting a Node from a Linked List

- Assume that `list` has the following appearance and `n` is 20:

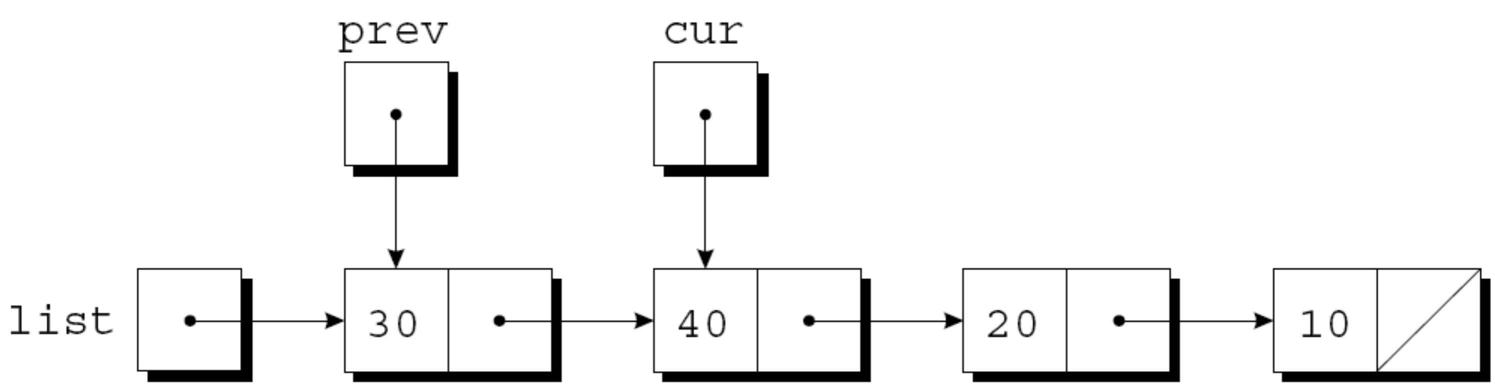


- After `cur = list, prev = NULL` has been executed:



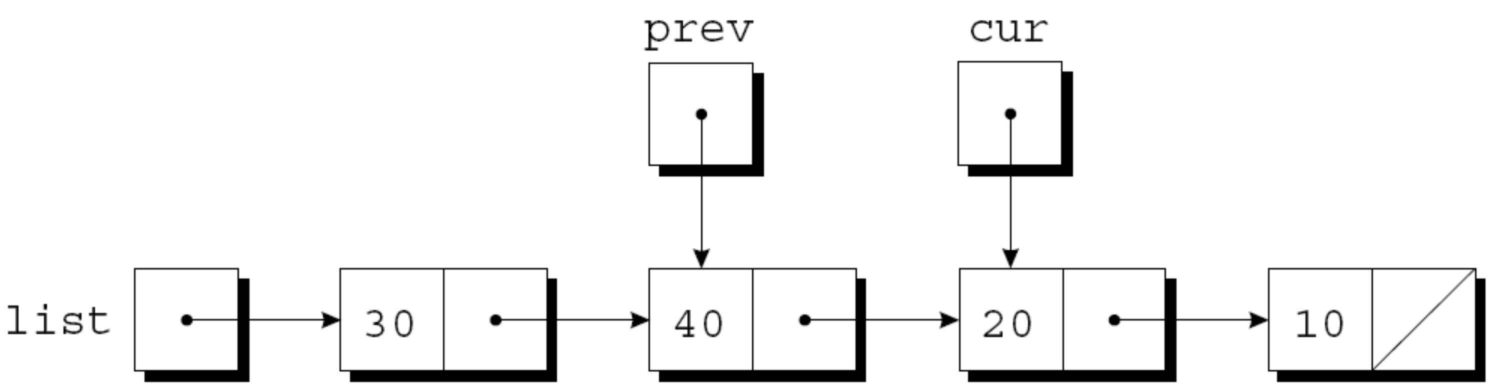
Deleting a Node from a Linked List

- The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20.
- After `prev = cur, cur = cur->next` has been executed:



Deleting a Node from a Linked List

- The test `cur != NULL && cur->value != n` is again true, so `prev = cur`, `cur = cur->next` is executed once more:



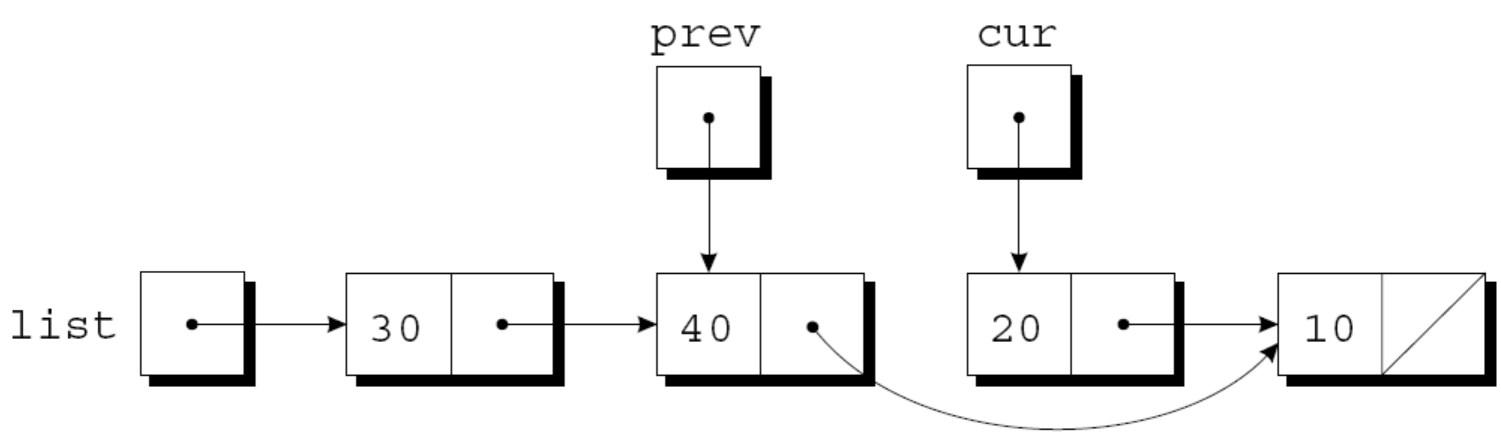
- Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Deleting a Node from a Linked List

- Next, we'll perform the bypass required by step 2.
- The statement

```
prev->next = cur->next;
```

makes the pointer in the previous node point to the node *after* the current node:



Deleting a Node from a Linked List

- Step 3 is to release the memory occupied by the current node:

```
free(cur);
```

Deleting a Node from a Linked List

- The `delete_from_list` function uses the strategy just outlined.
- When given a list and an integer `n`, the function deletes the first node containing `n`.
- If no node contains `n`, `delete_from_list` does nothing.
- In either case, the function returns a pointer to the list.
- Deleting the first node in the list is a special case that requires a different bypass step.

Deleting a Node from a Linked List

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
  if (cur == NULL)
    return list;              /* n was not found */
  if (prev == NULL)
    list = list->next;        /* n is in the first node */
  else
    prev->next = cur->next;   /* n is in some other node */
  free(cur);
  return list;
}
```

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
- Chapter 17