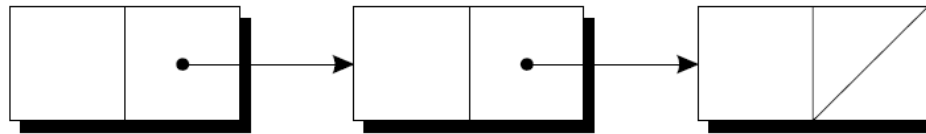


Linked Lists

Linked Lists

- Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures.
- A **linked list** consists of a chain of structures (called **nodes**), with each node containing a pointer to the next node in the chain:



- The last node in the list contains a null pointer.

Linked Lists

- A linked list is more flexible than an array: we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed.
- On the other hand, we lose the “random access” capability of an array:
 - Any element of an array can be accessed in the same amount of time.
 - Accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it’s near the end.

Declaring a Node Type

- To set up a linked list, we'll need a structure that represents a single node.
- A node structure will contain data (an integer in this example) plus a pointer to the next node in the list:

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

- `node` must be a tag, not a `typedef` name, or there would be no way to declare the type of `next`.

Declaring a Node Type

- Next, we'll need a variable that always points to the first node in the list:

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

- Setting `first` to `NULL` indicates that the list is initially 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.

Creating a Node

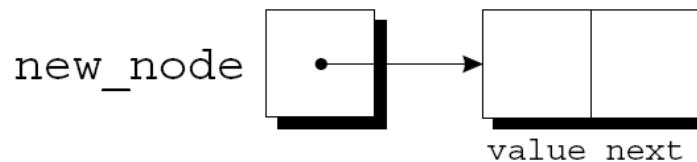
- When we create a node, we'll need a variable that can point to the node temporarily:

```
struct node *new_node;
```

- We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

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

- `new_node` now points to a block of memory just large enough to hold a `node` structure:

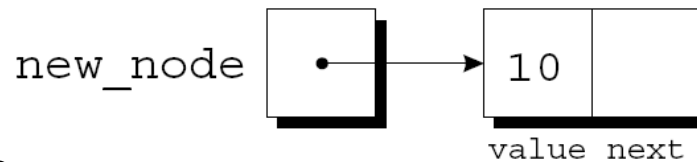


Creating a Node

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

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

- The resulting picture:



- The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

The -> Operator

- Accessing a member of a structure using a pointer is so common that C provides a special operator for this purpose.
- This operator, known as ***right arrow selection***, is a minus sign followed by >.
- Using the -> operator, we can write

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

instead of

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

The -> Operator

- The -> operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed.
- A `scanf` example:

```
scanf("%d", &new_node->value);
```
- The `&` operator is still required, even though `new_node` is a pointer.

Inserting a Node at the Beginning of a Linked List

- One of the advantages of a linked list is that nodes can be added at any point in the list.
- However, the beginning of a list is the easiest place to insert a node.
- Suppose that `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list.

Inserting a Node at the Beginning of a Linked List

- It takes two statements to insert the node into the 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;
```

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

```
first = new_node;
```

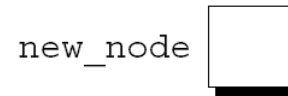
- These statements work even if the list is empty.

Inserting a Node at the Beginning of a Linked List

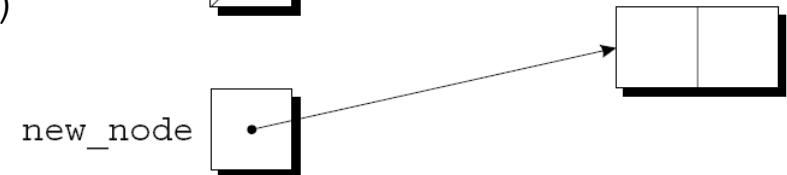
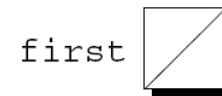
- Let's trace the process of inserting two nodes into an empty list.
- We'll insert a node containing the number 10 first, followed by a node containing 20.

Inserting a Node at the Beginning of a Linked List

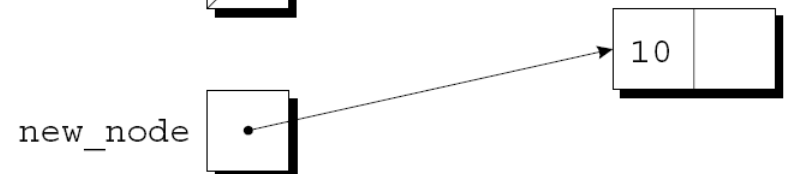
```
first = NULL;
```



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

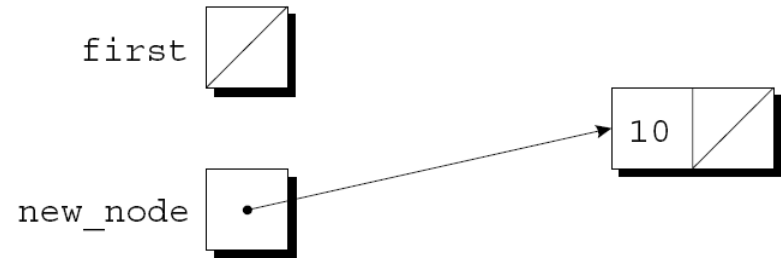


```
new_node->value = 10;
```

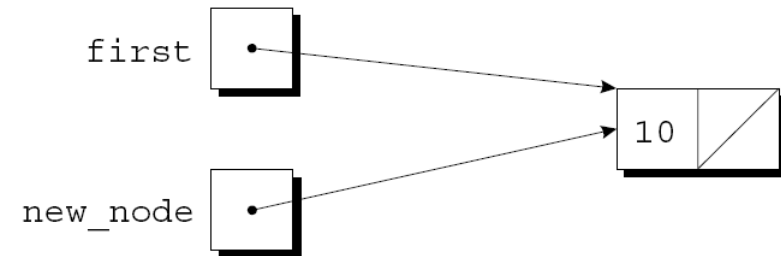


Inserting a Node at the Beginning of a Linked List

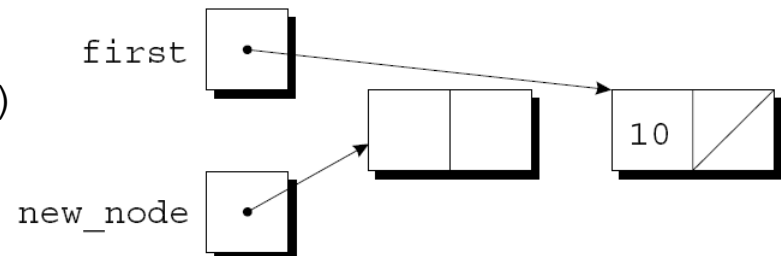
```
new_node->next = first;
```



```
first = new_node;
```

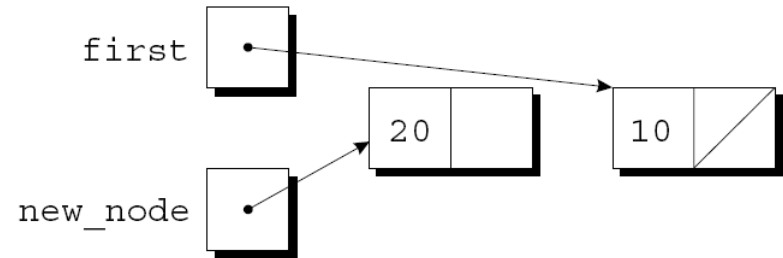


```
new_node =  
    malloc(sizeof(struct node))
```

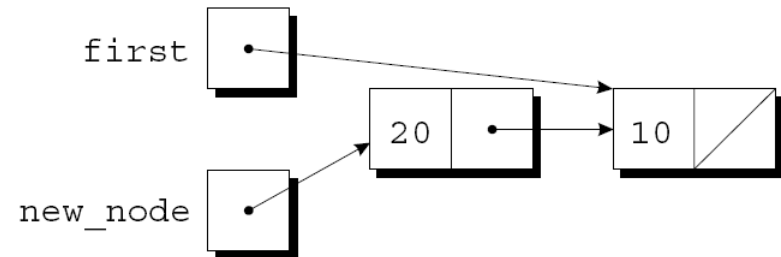


Inserting a Node at the Beginning of a Linked List

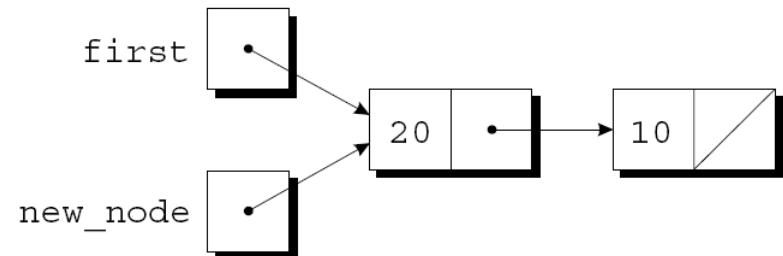
```
new_node->value = 20;
```



```
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 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;
}
```

Inserting a Node at the Beginning of a Linked List

- Note that `add_to_list` returns a pointer to the newly created node (now at the beginning of the list).
- When we call `add_to_list`, we'll need to store its return value into `first`:

```
first = add_to_list(first, 10);  
first = add_to_list(first, 20);
```

- Getting `add_to_list` to update `first` directly, rather than return a new value for `first`, turns out to be tricky.

Inserting a Node at the Beginning of a Linked List

- A function that uses `add_to_list` to create a linked list containing numbers entered by the user:

```
struct node *read_numbers(void)
{
    struct node *first = NULL;
    int n;

    printf("Enter a series of integers (0 to terminate): ");
    for (;;) {
        scanf("%d", &n);
        if (n == 0)
            return first;
        first = add_to_list(first, n);
    }
}
```

- The numbers will be in reverse order within the list.

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.

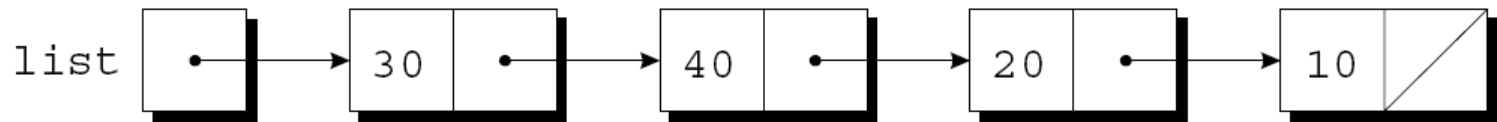
Deleting a Node from a Linked List

- 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:

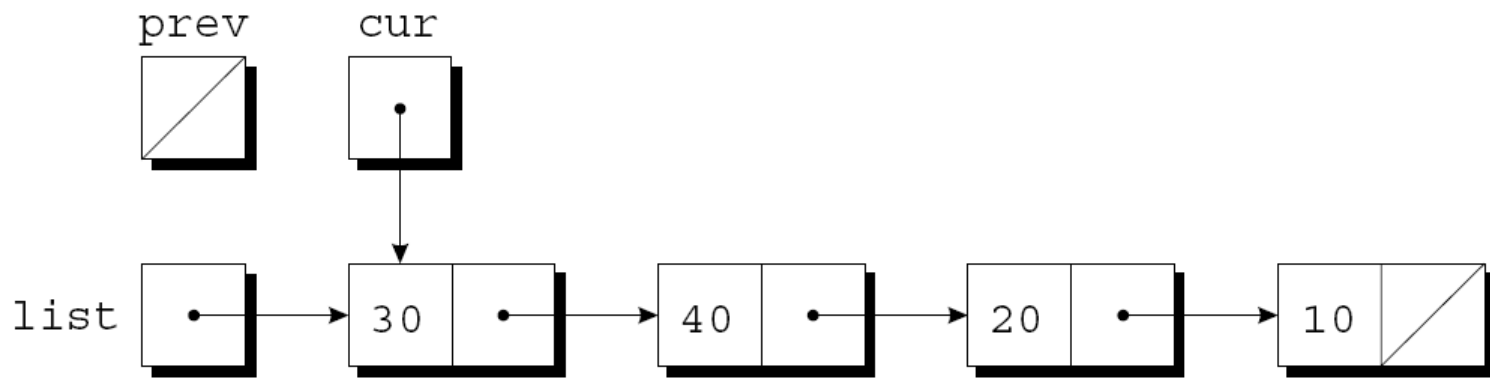
```
for (cur = list, prev = NULL;  
    cur != NULL && cur->value != n;  
    prev = cur, cur = cur->next)  
    ;
```
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node.

Deleting a Node from a Linked List

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

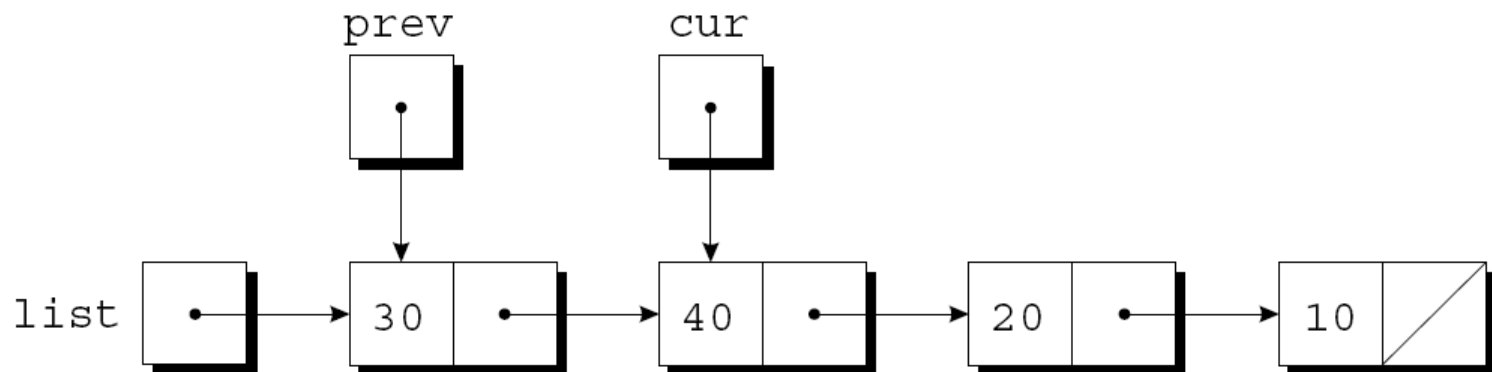


- After `cur = list`, `prev = NULL` has been executed:



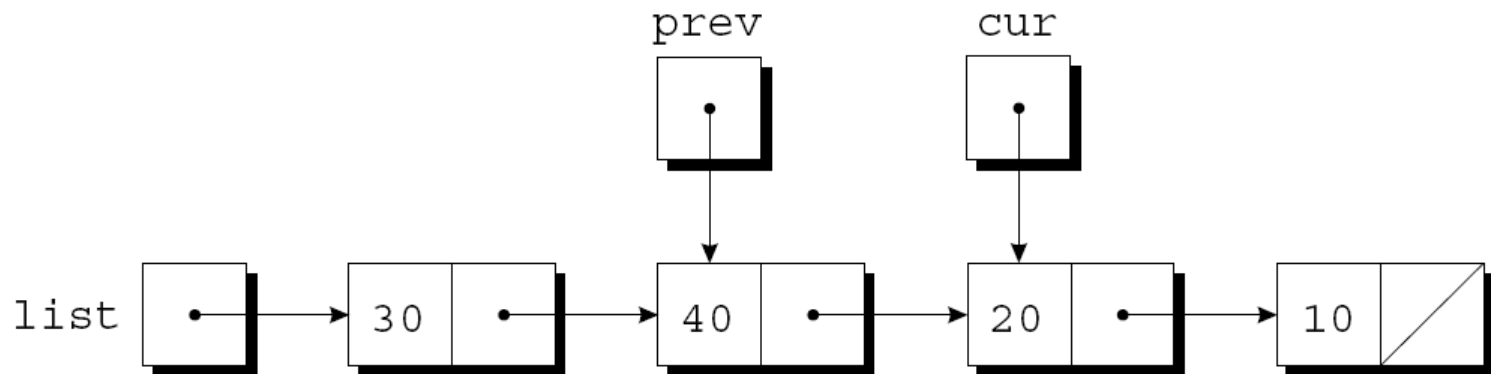
Deleting a Node from a Linked List

- 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:



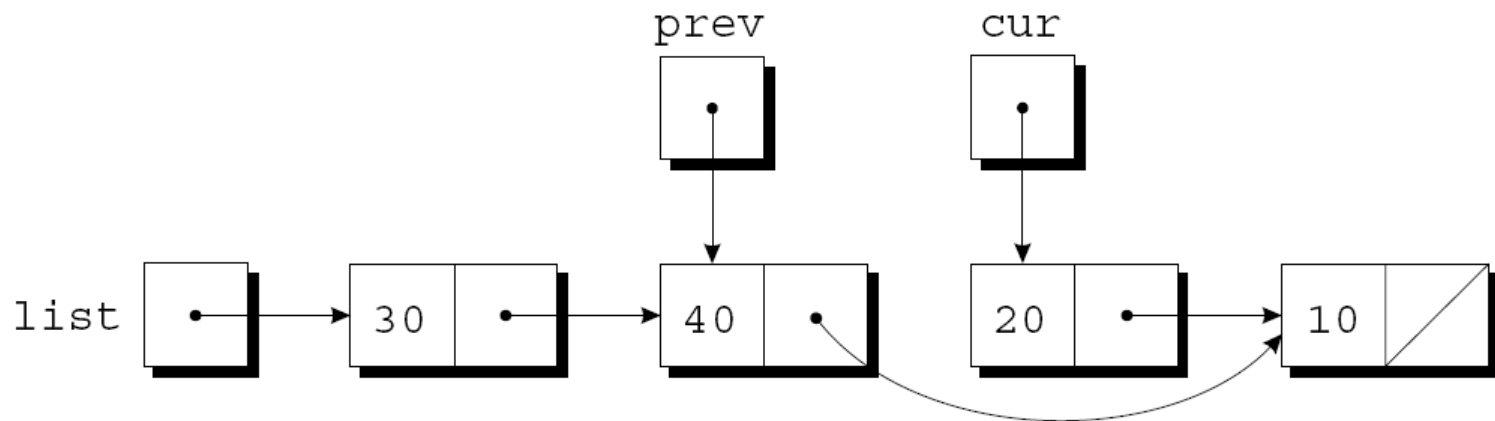
- Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

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;
}
```

Ordered Lists

- When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ***ordered***.
- Inserting a node into an ordered list is more difficult, because the node won't always be put at the beginning of the list.
- However, searching is faster: we can stop looking after reaching the point at which the desired node would have been located.