DS

Doubly-linked Linked Lists

Sorted Linked Lists

List Traversal in One Direction Only

So far, we can start at the head of a linked list and go forward to the end

This means that we can't go from a cell at the end of the list to the one before it

Supporting List Traversal in Two Directions

This feature can be added simply by adding two items to our list:

- tail
- prev

This gives us a doubly-linked list

Tail

tail is a variable very similar to *head*

It is a pointer to a node within the list that happens to point to the end, rather than the start, of the list

It always starts as NULL

e.g. Node *tail = NULL;

Updating tail

As you add items to the end or delete items from the end of the list, you change the value of *tail*

This makes it so you don't have to traverse the list in order to add at the end

Deleting from the tail

If we are deleting the last node in the list, the tail pointer must be changed to point to the node before it

So far, we don't have any way of finding out what that node is except traversing the list and stopping at the node before the tail

Instead, we can keep a pointer to the previous node

Previous Nodes

The pointer to the previous node in the list is typically called *prev*

It is similar in operation to the *next* pointer and is stored in the struct containing the linked list definition (usually right before the *next* field)

The Struct

e.g.

```
struct Node
{
    char name[50];
    char address[50];
    struct Node *prev;
    struct Node *next;
};
```

Doubly-linked List

Because there are now two link pointers within the list, this is called a doubly-linked list

Adding a Node in a Doubly-linked List

When adding a node to the end of a doubly-linked list:

- set the *prev* field to point to the node pointed to by *tail*
- set the *tail* variable to point to the newly-added node

Sorted Lists

So far, these lists are sorted implicitly by order of insertion

Usually, you'd want to sort the list by some other field (e.g. name)

- We will call this the "sorting field" or "sorting key"

This can make retrieval of information based on the sorting field much faster

How to Create a Sorted Linked List

Traverse the list, comparing the sorting field of your new node against the sorting field of each node in your list

- While your node's sorting field value is greater than the sorting field value of the node that you are examining in the list, you know that you haven't reached the insertion point

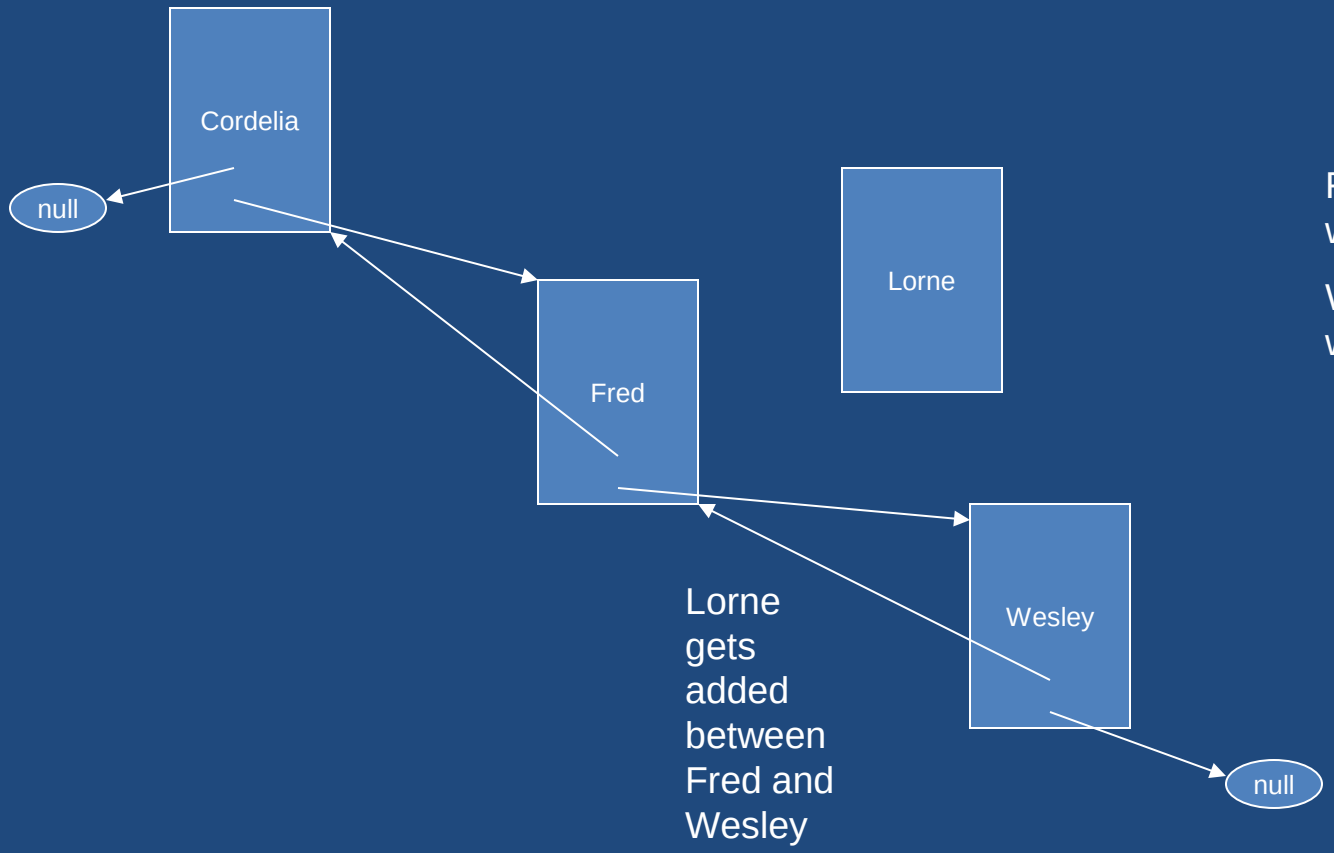
- This is easiest to see in a diagram (still to come in a few slides)

Looping ...

Loop until the sorting field value of the next node in the list is greater than our node (or the end of the list is reached)

- If you're not at the end of the list, you insert your node between the current node and the next one

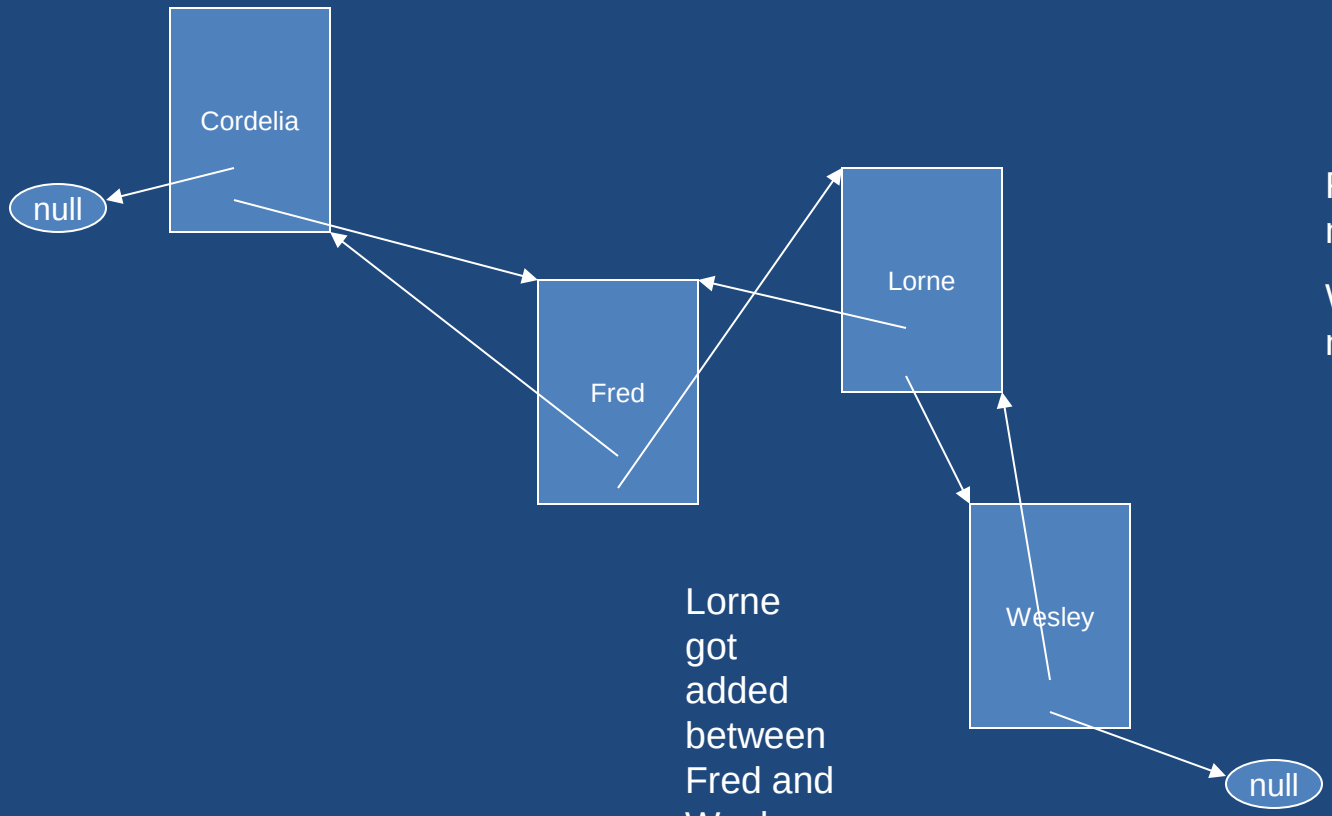
Positioning in a Sorted Doubly-linked List



Fred's next pointer will point to Lorne.

Wesley's prev pointer will point to Lorne.

Positioning in a Sorted Doubly-linked List after Insertion



Lorne got added between Fred and Wesley

Fred's next pointer now points to Lorne.

Wesley's prev pointer now points to Lorne.

Issues About Inserting Into a Doubly-Linked List

You have to watch for:

- inserting at the end of the list
- inserting at the beginning of the list
- inserting into an empty list

- deciding what to do if you have equal sorting fields
- making sure that you handle *next*, *prev*, *head*, and *tail* properly without orphaning
- dereferencing NULL pointers carelessly

Searching in a Sorted List

Searching is faster than in an unsorted list, since you can stop as soon as the node's sorting field is greater than the one you're looking for

- (This only applies if you're searching for a value in the sorting field)

Aside: Issue About Deletion from any List

If you are supposed to delete an item from a list, you either want to:

- delete all occurrences of that item in the list
- or
- get user input to narrow the choice

Deleting only the first occurrence of the item in the list typically is not the right behaviour

Deletion in a Doubly-linked List

Deletion requires more care because we have a *tail* variable and a *prev* pointer that needs to be adjusted

Summary

We can make linked lists more powerful

by adding

sorting

and

bi-directional links with a tail.