University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Pointer-Based Data Structures

Can We Speed Up Deletion from Linked Lists?

Can we speed up deletion from a linked list?

To delete **`p`**, we need to find **`p`**'s predecessor.

Any ideas?

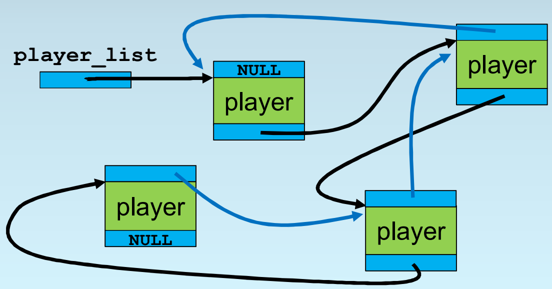
Why not **add a second `player_t*`**?

We can **call it `prev`**.

Doing so gives us **a doubly-linked list**.

There are Many Ways to Doubly-Link a List

Drawn somewhat sloppily…



Use a Sentinel and a Cyclic List to Simply the Code

One good way,

- where "good" means that
- both insertion and deletion are simple,
- is to **use a sentinel**:

```
static player_t player_list;
```

Notice that **`player_list`** is not a pointer.

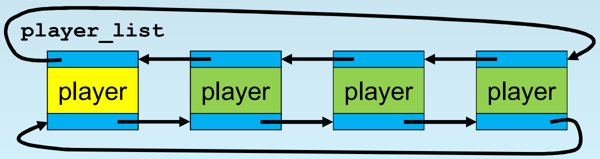
It's a fake player for use as a sentinel.

To avoid **`NULL`**, **the list is** then **cyclic**.

A Cyclic, Doubly-Linked List with a Sentinel

Drawn below is a **cyclic, doubly-linked list with a sentinel**.

All pointers point to start of target structures (not the middle).



Easy to Walk Over the List of Players

Let's see how it's used.

How do we do something for all players?

```
player_t* p;
for (
                ) {
    // do something for all players
}
```

Start with an iteration variable **p** and a **for** loop.

Easy to Walk Over the List of Players

Let's see how it's used.

How do we do something for all players?

```
player_t* p;
for (p = player_list.next;

                ) {
    // do something for all players
}
```

Where is the first player?

Easy to Walk Over the List of Players

Let's see how it's used.

How do we do something for all players?

```
player_t* p;
for (p = player_list.next;
     &player_list != p;
                ) {
    // do something for all players
}
```

What is the end of the list?
(Hint: not **NULL**.)

Easy to Walk Over the List of Players

Let's see how it's used.

How do we do something for all players?

```
player_t* p;
for (p = player_list.next;
      &player_list != p;
      p = p->next) {
    // do something for all players
}
```

And how do we advance to the next player?

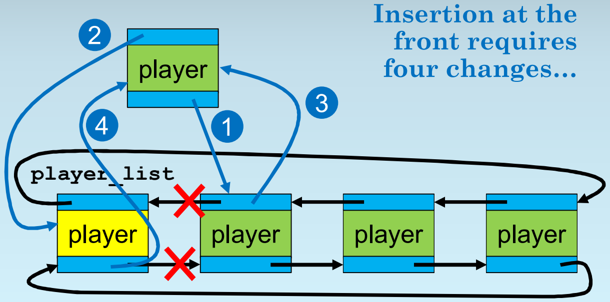
The Opposite Direction is Equally Easy

But…what if we want the other direction?

Just change next to prev in both cases!

```
player_t* p;
for (p = player_list.prev;
      &player_list != p;
      p = p->prev) {
    // do something for all players
}
```


Insertion Requires Four Changes In Correct Order



Insertion at the front requires four changes…

Insertion at Either End of the List is Easy

Given a new player_t* p, we have…

```
p->next = player_list.next;
p->prev = &player_list;
player_list.next->prev = p;
player_list.next = p;
```

Or, at the end,

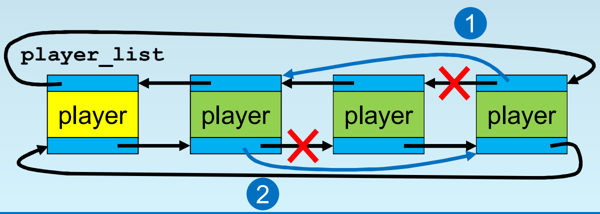
```
p->prev = player_list.prev;
p->next = &player_list;
player_list.prev->next = p;
player_list.prev = p;
```


Deletion Requires Only Two Changes

The order doesn't matter.

What about deletion?

How can we delete the middle real player?



Deletion is Quite Simple

Given a `player_t* p` to be deleted…

```
p->next->prev = p->prev;
p->prev->next = p->next;
```

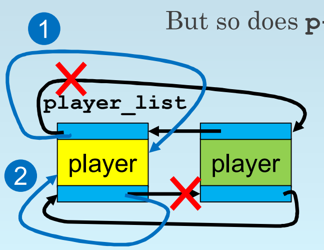
That's it! No loop required!

Sentinel Links to Itself When the List is Empty

What happens if we delete the last player?

`p->next` points to `player_list`

But so does `p->prev`…

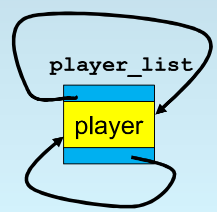


`player_list` now points to itself in both directions!

(That's an empty list.)

Sentinel Links to Itself When the List is Empty

(an empty cyclic, doubly-linked list with a sentinel)



Pointers Can Serve Many Purposes

In general, we can
- **add an arbitrary number of pointers**
- to any structure.

Pointers can be **used to organize groups of structures** in different ways.
- orderings
- relationships
- properties

Example: Use Linked List to Maintain Ordering

For example, say that we want to sort players
- by name,
- by age, and
- by number of games played.

We **can maintain all three orderings**
- **using** three separate "next" fields (**`player_t*`**) in the player structure.
- Each field corresponds to a single ordering.

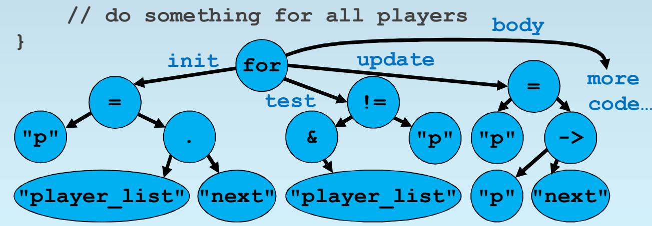
Example: Abstract Syntax Trees (ASTs)

Another example:
- **abstract syntax trees (ASTs)**
- used as an intermediate representation (IR) of a program for compilation

Nodes represent operators or statements,
- **operands** are a relation to operators, and
- **initialization, tests, and updates** are a relation to statements (if, for, while, do).
- **All make use of pointers** to other nodes.

Illustration of an AST Construction

```
for (p = player_list.next;
     &player_list != p;
     p = p->next) {
    // do something for all players
}
```



switch Statement Cases Cannot be Linked Directly

What about `switch` statements?

What's the problem?

Number of cases is effectively unbounded.

How can we add fields to point to an unknown number of cases?

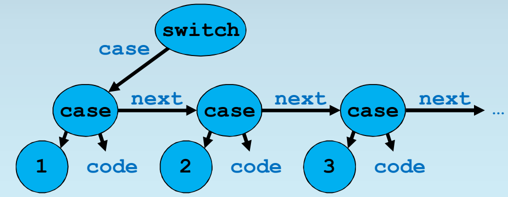
Answer: we can't.

So … don't allow `switch` statements?

Didn't we already solve this problem?

Illustration of a **switch** Statement Construction

Solution: use two pointers…



You'll see something similar in MPs 10 and 11.

