Slide 19 add-front

What should happen

What actually happens

node->next = lst->front; // link node to 10
lst->front = node; // link wrapper to node.

Does this alternative version work? No.

lst->front = node; // link wrapper to node
node->next = lst->front; // link node to 10.
    not the old front node anymore.

Summary: always link cur node to the next node first, then link prev node to the cur node.
Slide 21: Traverse a list

```plaintext
node = node -> next;
```

Slide 23-25: Destroy a List

For every node, we need to:

1. Free the node
2. Go to the next node

```
node
```

1. Free the node first, then cannot get to the next node

```
node
```

2. Go to the next node first, then cannot free cur node.

```
node
```

Notes
Slides 23 destroy list (iterative) v1

save the next node
free cur node
go to next node

Slide 24 destroy list (iterative) v2

save the cur node
go to next node
free cur node

Slide 25 destroy list (recursive)

reverse order

void free_nodes (struct llnode *node) {
  if (node) {
    struct llnode *nextnode = node->next;  // ∆
    free (node);
    free_nodes (nextnode);
  }

  ∆ save the next node for the recursive call first
}
Slide 28: duplicate a list (Iterative)

First try:

old list

old node

new list

Actual code:

old list

old node

prev node

new list

△ Need to remember prev node to link it to new node.
Insert to the front is special when?
- `curr -> next = lst -> front;`  
- `i < slst -> front -> item` or `slst` is empty.

Insert into any other position.

1. Find the correct position.

   \[
   \begin{align*}
   & 10 \rightarrow 20 \rightarrow 50 \rightarrow 100 \\
   & \text{\textless 30 \quad \textless 30 \quad \textgreater 30.}
   \end{align*}
   \]

   - Go through the list until an element \( \geq i \).
   - Want to insert before \( x \).
   - Need a pointer to the element right before \( x \).

   \[
   \begin{align*}
   & 10 \rightarrow 20 \rightarrow 50 \rightarrow 100 \\
   & \text{prevnode} \rightarrow \text{check: is prevnode -> next's value > i ?}
   \end{align*}
   \]

2. Insert node at the correct position.
   - Create new node
   - Link new node to prevnode -> next
   - Link prev node to new node.
Slides 29-30: Insert

* insert 200

![Diagram of list insertion]

Slides 31-32: Removing nodes

Slide 31: Remove the front node

1. Save current node
2. Link front to next node
3. Free current node

![Diagram of list removal]

next node

backup

struct llnode *next_node = (lst->front)->next;
free(lst->front);
lst->front = next_node;
Slide 32 remove node from an arbitrary position. Given value i of node, remove first node with value i.

1. Find the correct position.
   - If we've reached the end, return false.
   - Else, remove node.

   1. Remove front.
   2. Remove from middle.
   4. Given value is in the first node, remove front.

   Edge cases:
   - List is empty.

3. Item not found.

4. List is empty. Item not found.
Slide 11. Naïve translation of insert into C.

\[
\begin{align*}
&\text{insert } 30 \ '(10 \ 20 \ 50 \ 100) \rightarrow \\
&\text{cons } 10 \ \text{(insert } 30 \ '(20 \ 50 \ 100)) \rightarrow \\
&\text{cons } 20 \ \text{(insert } 30 \ '(50 \ 100)) \rightarrow \\
&\text{cons } 30 \ '(50 \ 100) \rightarrow \\
&\text{part of } a.
\end{align*}
\]

Rest does not make a new list, it produces a pointer to the second node.

Two lists share nodes.

Perfectly fine with functional programming:
- no mutation
- garbage collector

Problematic with imperative programming:
- mutation
- free.

Guidelines:
1. Lists do NOT share nodes
2. New nodes are only created when necessary.