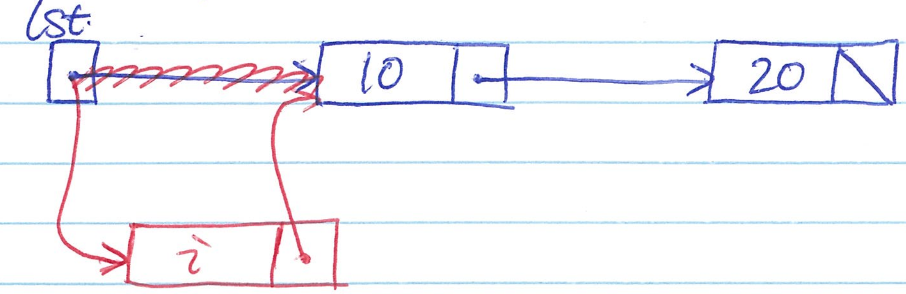
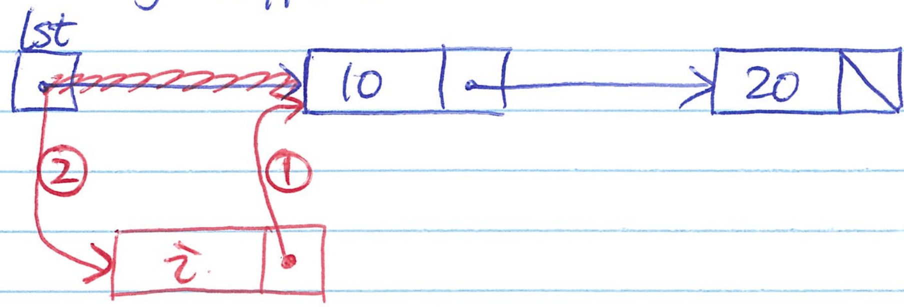
Slide 19 add_front

What should happen



What actually happens.

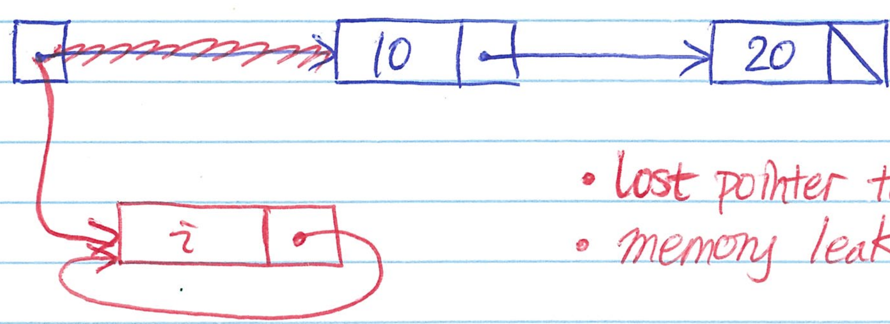



```
node -> next = lst -> front; // link node to 10
lst -> front = node; // link wrapper to node.
```

Does this alternative version work? NO.

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

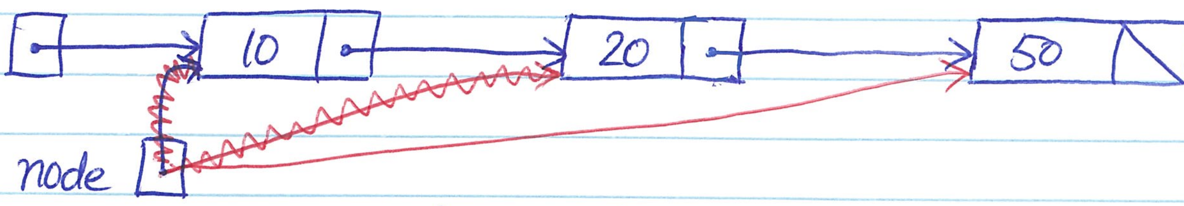
not the old front node anymore.



- lost pointer to 10.
- memory leak.

Summary: always link cur node to the next node first.
then link prev node to the cur node.

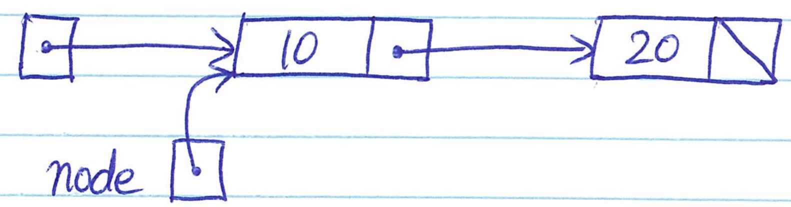
Slide 21: traverse a list



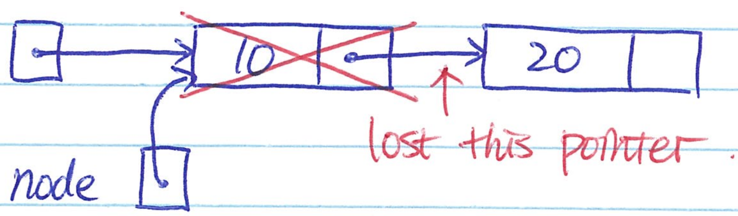
node = node → next ;

Slide 23-25: destroy a list.

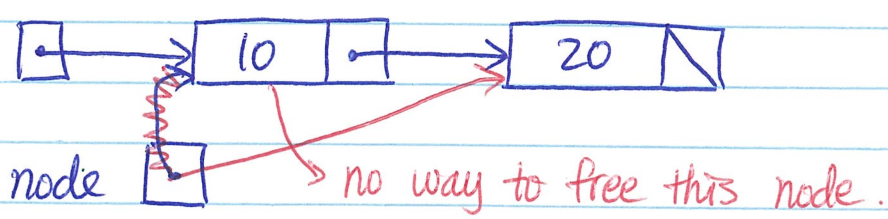
For every node, we need to
① free the node
② go to the next node.
} cannot do both with one pointer.



① free the node first, then cannot get to the next node

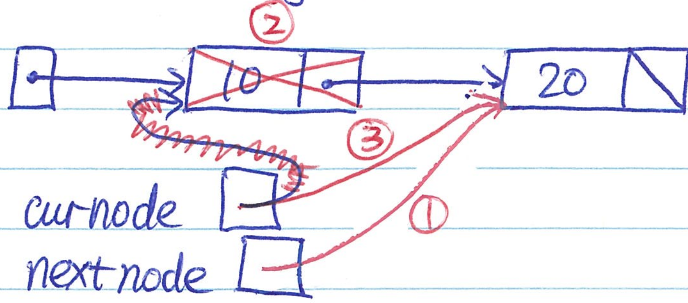


② go to the next node first, then cannot free cur node.



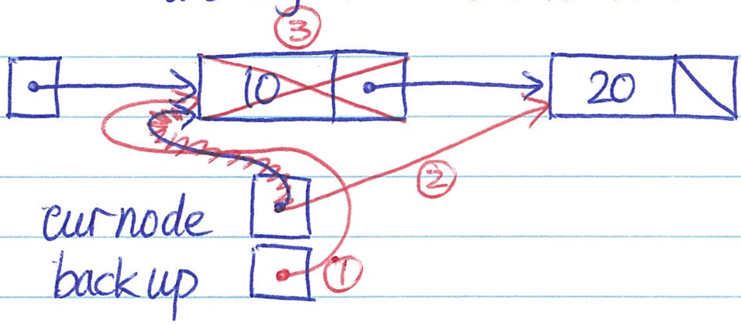
notes

Slides 23 destroy list (iterative). v1



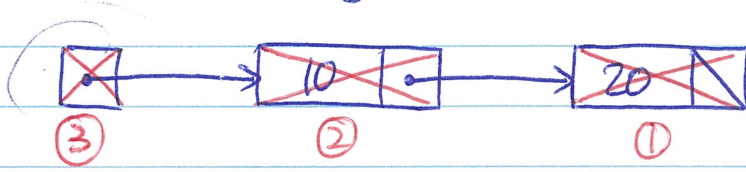
save the next node.
free cur node
go to next node.

Slide 24 destroy list (iterative) v2



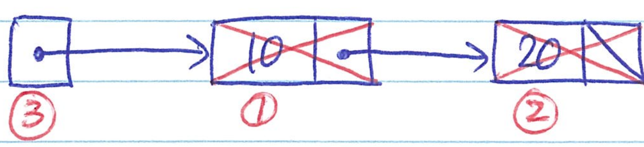
save the cur node
go to next node
free cur node.

Slide 25 destroy list (recursive)



reverse order

in order

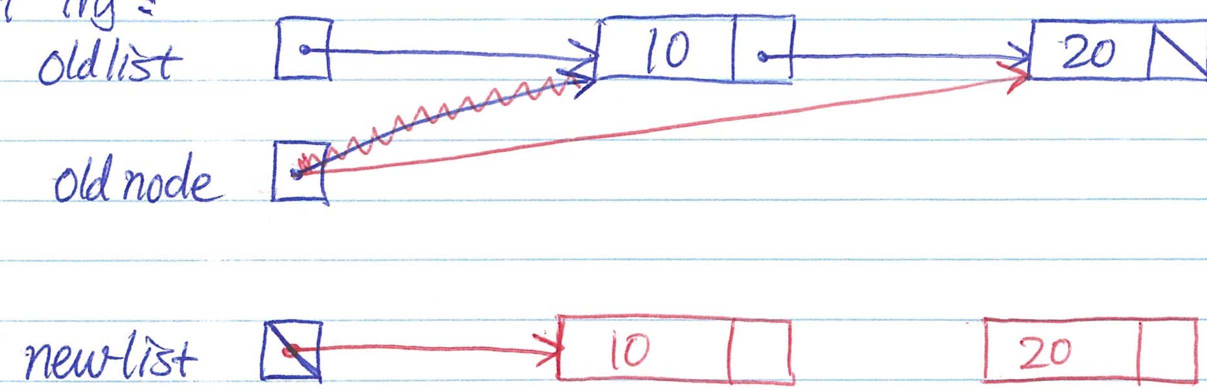



```
void free_nodes(struct llnode *node) {
    if (node) {
        struct llnode *nextnode = node->next; Δ
        free(node);
        free_nodes(nextnode);
    }
}
```

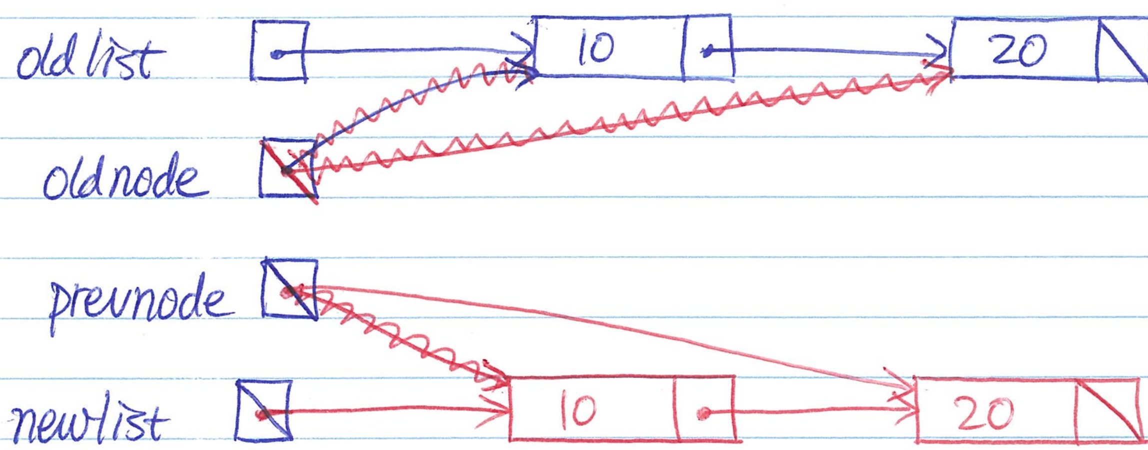
Δ save the next node for the recursive call first

Slide 28 : duplicate a list (iterative)

First try :

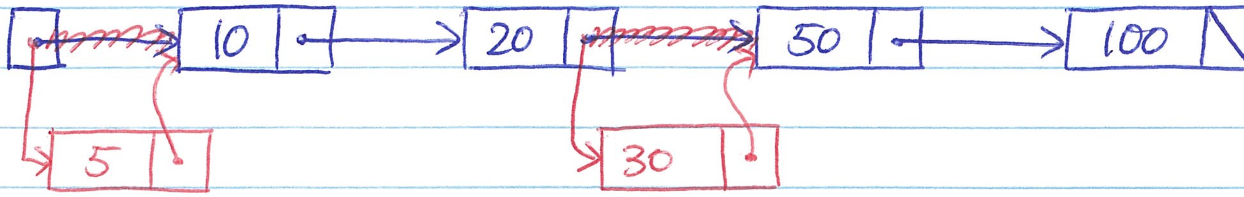


Actual code :



Δ Need to remember prev node to link it to new node.

Slides 29-30 Insert.



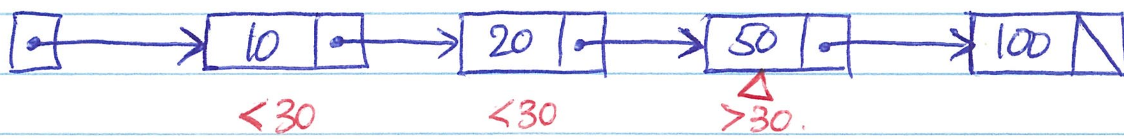
Insert to the front is special.

curr -> next = lst -> front;
lst -> front = curr;

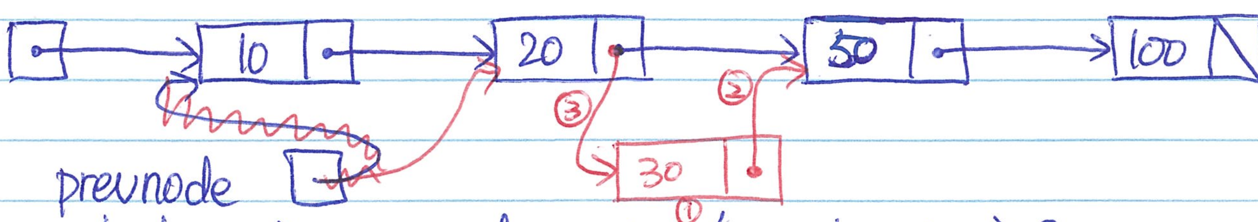
when?
$i <$ slst -> front -> item.
or slst is empty.

Insert into any other position.

① find the correct position.



- go through the list until $\underbrace{\text{an element}}_{x} > i$.
- want to insert before x.
- need a pointer to the element right before x.

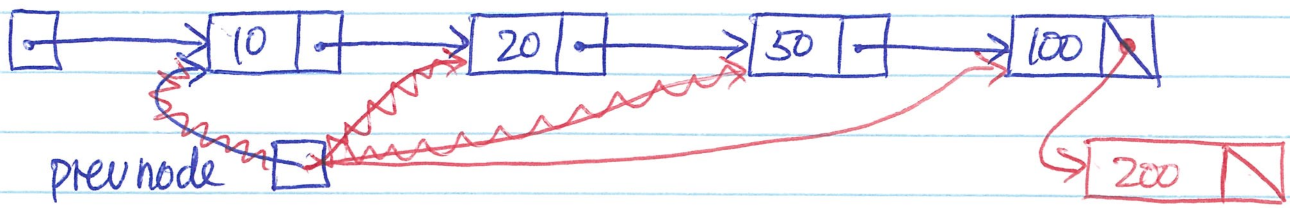


check: is prevnode -> next's value > i ?

② insert node at the correct position.
- create new node
- link new node to prevnode -> next
- link prev node to new node.

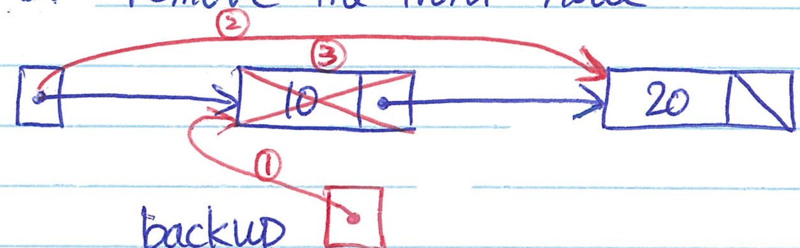
Slides 29–30 Insert.

Insert 200.

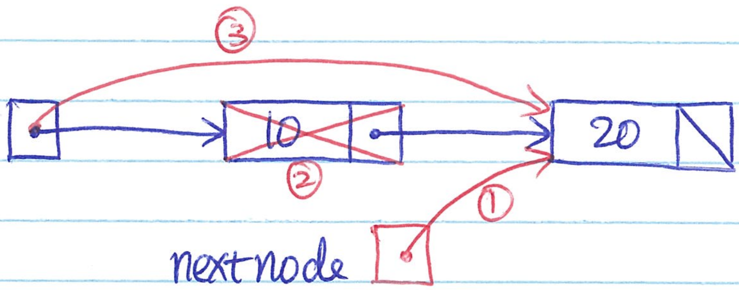


Slides 31–32 removing nodes.

Slide 31 remove the front node



- save cur node.
- link front to next node
- free cur node



- Save the next node
- free cur node
- link front to next node

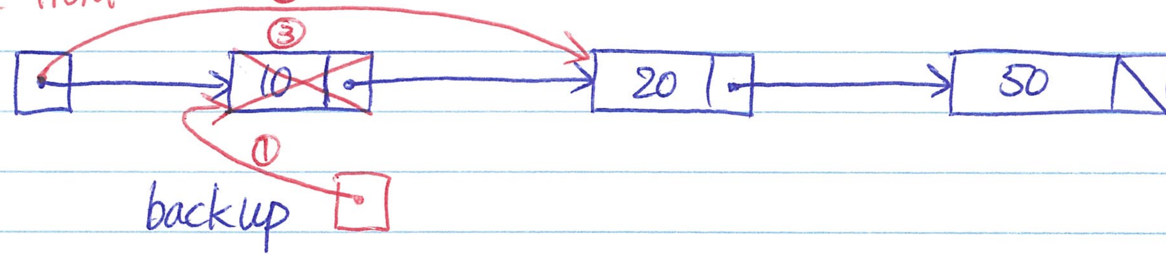
```
struct llnode *next node = lst->front->next;
 free(lst->front);
lst->front = next node;
```

Slide 32 remove node from an arbitrary position.
given value i of node, remove first node with value i.

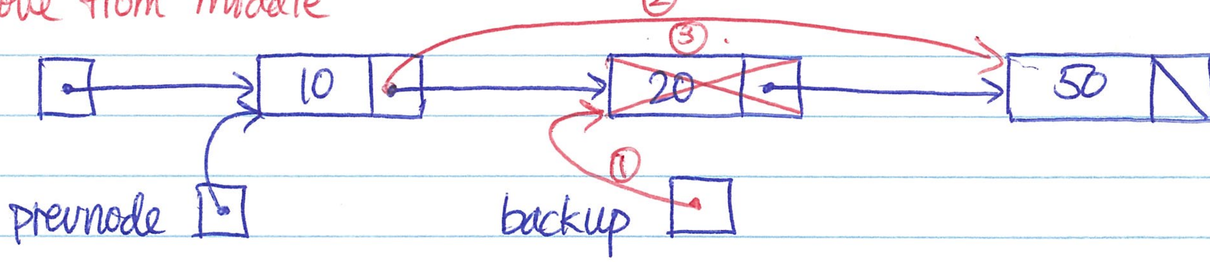
- find the correct position.
  - if we've reached the end, return false ③
  - else, remove node. ②
    - back up the current node
    - link prev node to the next node.
    - free current node through backup.

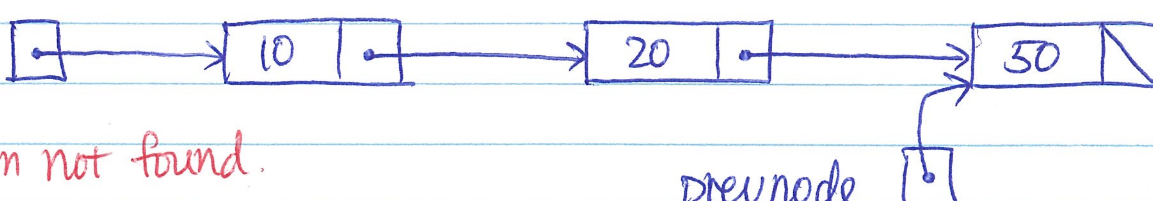
edge cases:
- list is empty. ④
- given value is in the first node. remove_front ①

①
remove front



②
remove from middle





③
item not found.

④  list is empty item not found.

Slide 11. naive translation of insert into C.

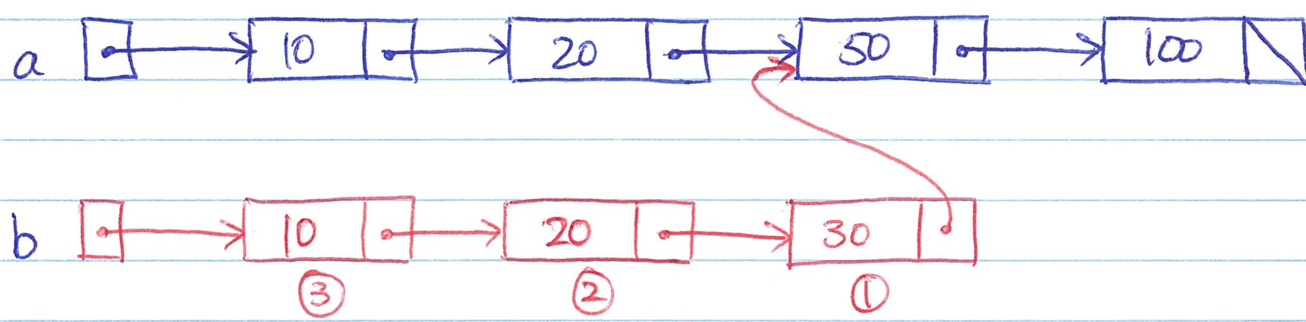
```
(insert 30 '(10 20 50 100))
(cons 10 (insert 30 '(20 50 100)))

(cons 20 (insert 30 '(50 100)))

(cons 30 '(50 100))
```

$\underbrace{\texttt{'(50 100))}}_{\text{part of a.}}$

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.