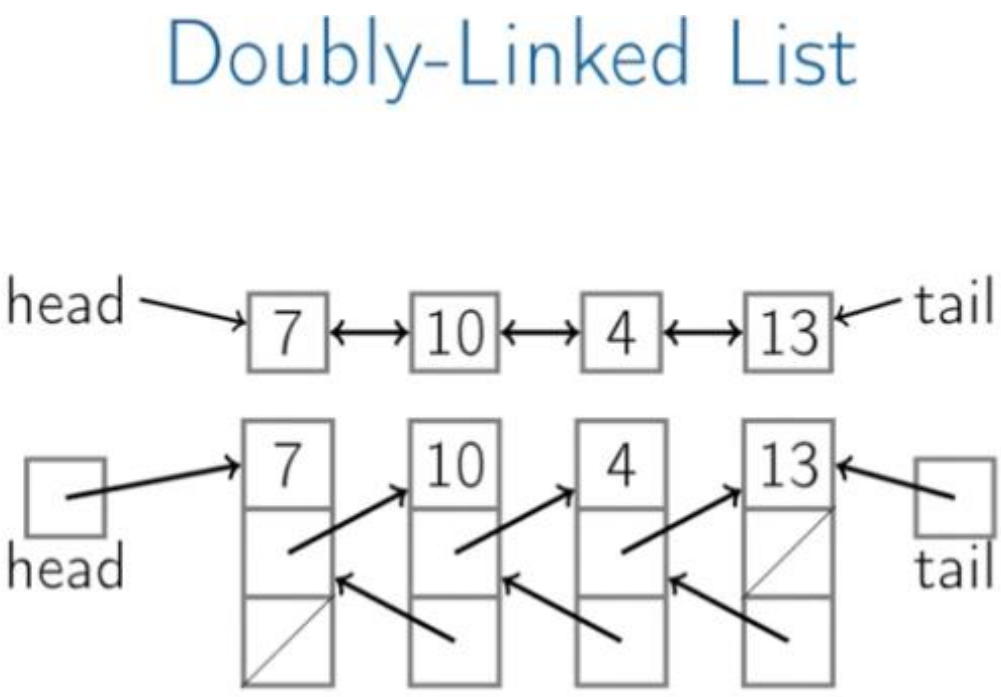
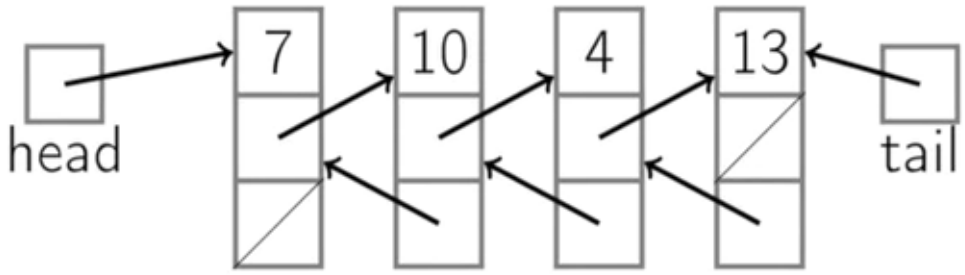
Doubly Linked List



Doubly-Linked List



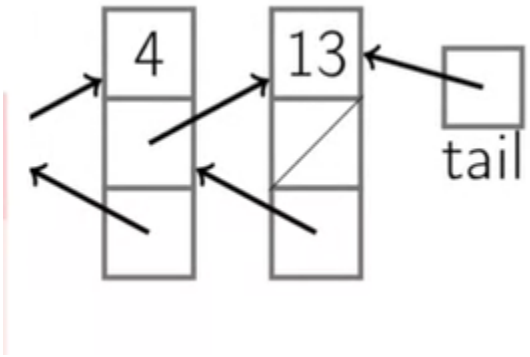
Node contains:

- `key`
- `next` pointer
- `prev` pointer

Doubly-linked List

PushBack(*key*)

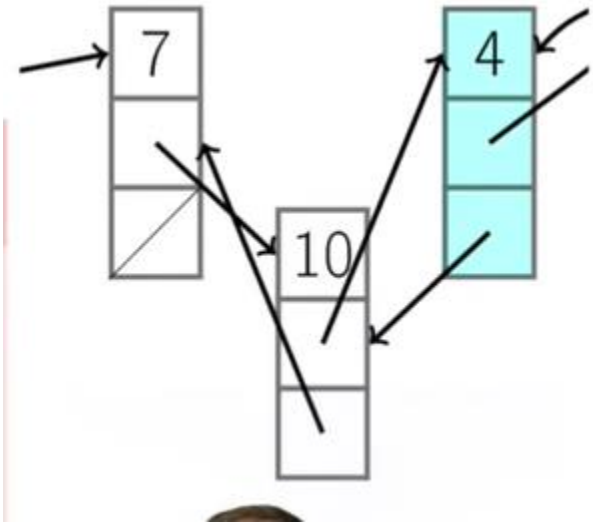
```
node ←new node
node.key ← key ;  node.next =nil
if  tail = nil :
    head ← tail ← node
    node.prev ←nil
else:
    tail.next ← node
    node.prev ← tail
    tail ← node
```



Doubly-linked List

AddBefore(*node*, *key*)

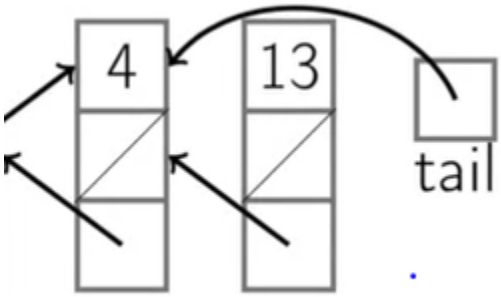
```
node2 ←new node
node2.key ← key
node2.next ← node
node2.prev ← node.prev
node.prev ← node2
if node2.prev ≠nil:
   node2.prev.next ← node2
if head = node:
   head ← node2
```



Doubly-linked List

PopBack()

```
if head = nil:   ERROR: empty list
if head = tail:
   head ← tail ←nil
else:
   tail ← tail.prev
   tail.next ←nil
```



Summary

- Constant time to insert at or remove from the front.
- With tail and doubly-linked, constant time to insert at or remove from the back.
- $O(n)$ time to find arbitrary element.
- List elements need not be contiguous.
- With doubly-linked list, constant time to insert between nodes or remove a node.

Doubly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	~~$O(n)$~~ $O(1)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	
`Empty()`	$O(1)$	
`AddBefore(Node, Key)`	~~$O(n)$~~ $O(1)$	
`AddAfter(Node, Key)`	$O(1)$	