Outline

1. Arrays
2. Linked Lists

```
long arr[] =   new long[5];
```

```
long arr[5];
```

```
arr =   [None] *  5
```

1	5	17	3	25

1	5	17	3	25
8	2	36	5	3

Definition

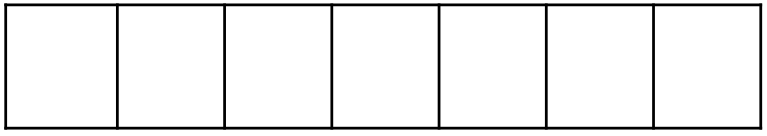
Array:

Contiguous area of memory

Definition

Array:

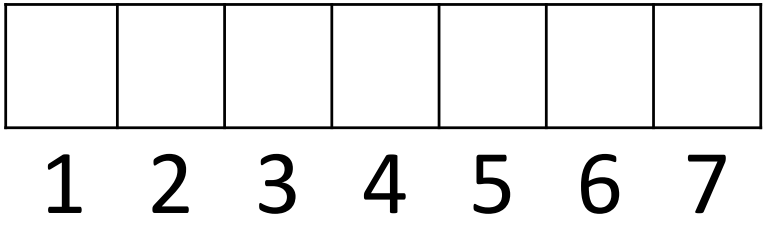
Contiguous area of memory consisting of equal-size elements



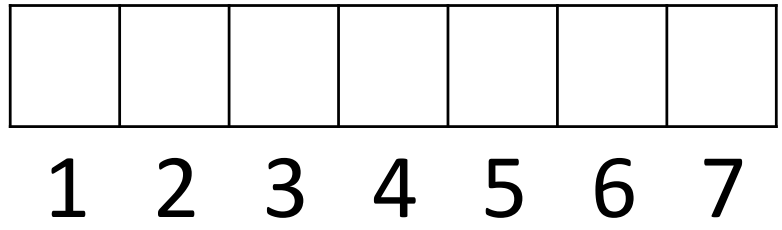
Definition

Array:

Contiguous area of memory consisting of equal-size elements indexed by contiguous integers.

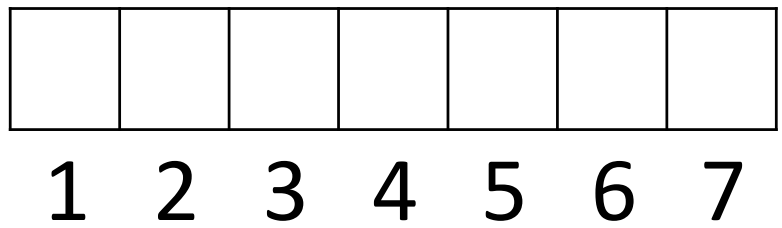


What's Special About Arrays?



What's Special About Arrays?

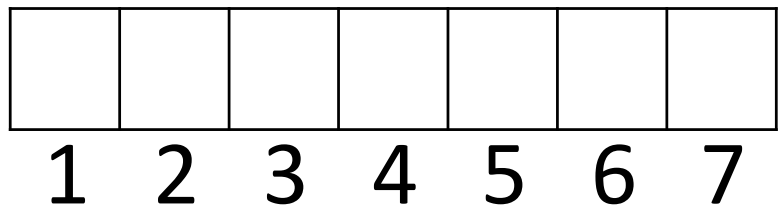
Constant-time access



What's Special About Arrays?

Constant-time access

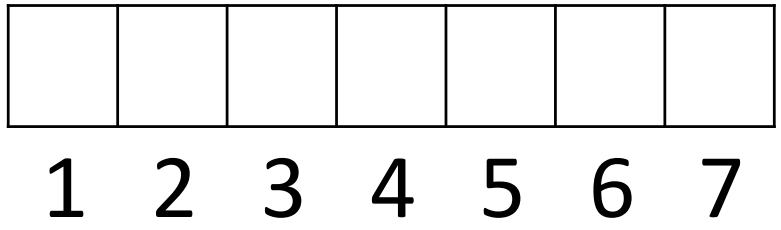
array_addr



What's Special About Arrays?

Constant-time access

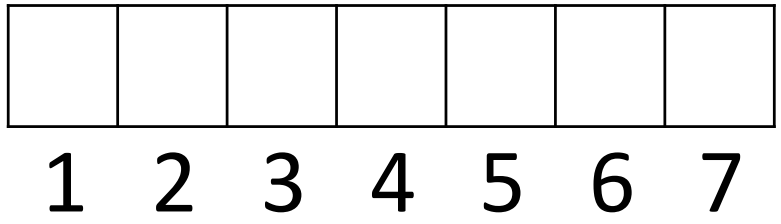
array_addr + elem_size × ( )



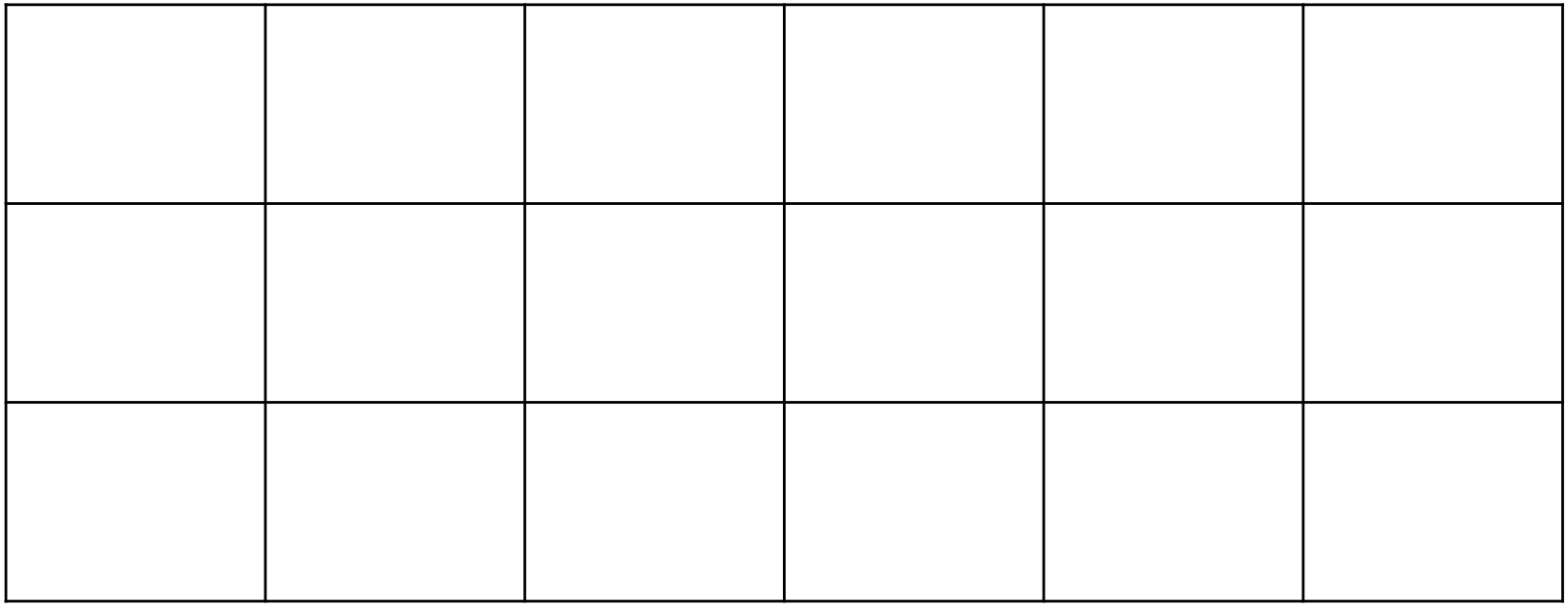
What's Special About Arrays?

Constant-time access

array_addr + elem_size × (i − first_index)



Multi-Dimensional Arrays



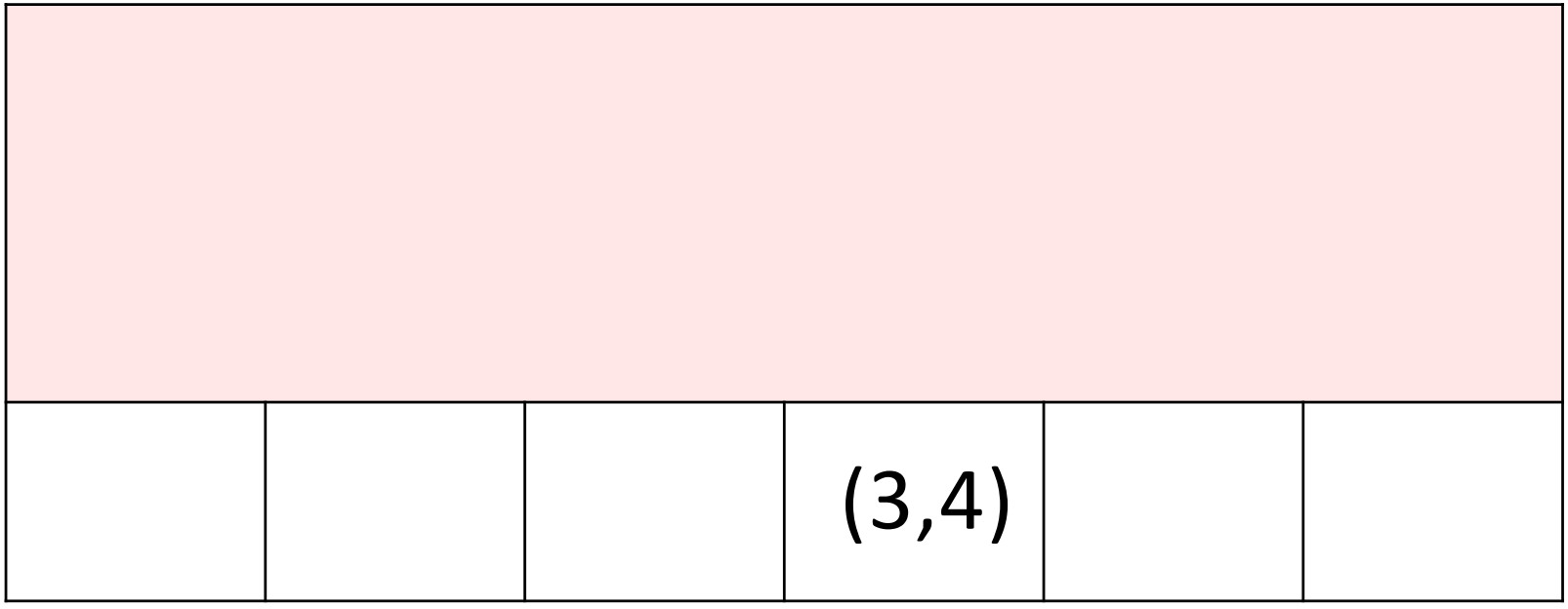
Multi-Dimensional Arrays

(1, 1)					

Multi-Dimensional Arrays

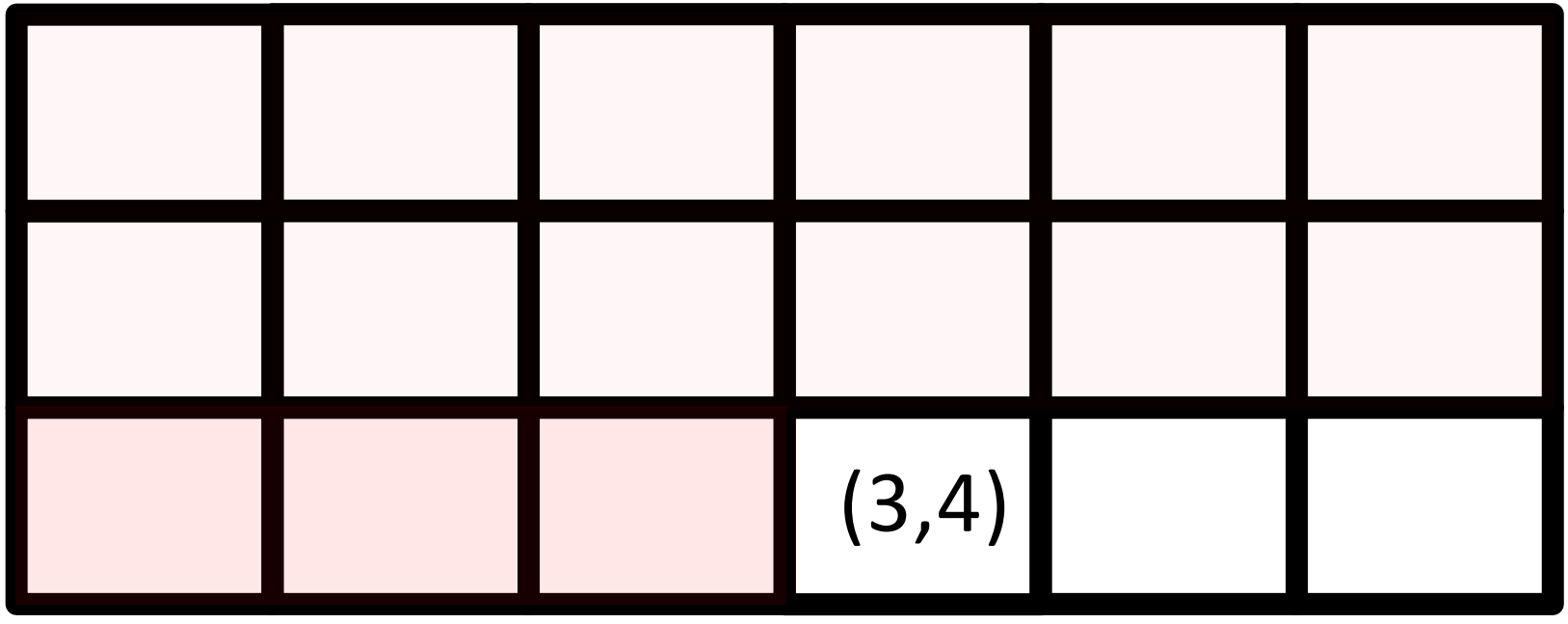
			(3,4)		

Multi-Dimensional Arrays



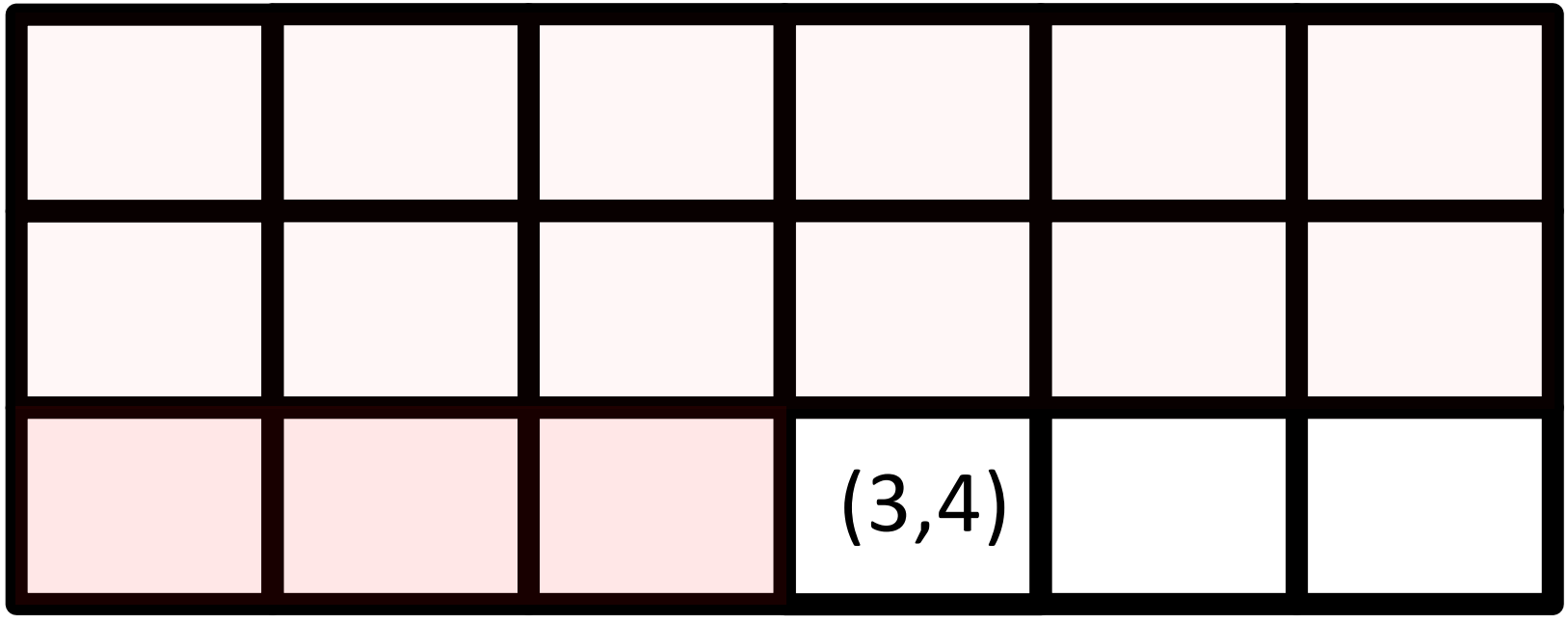
$(3 - 1) \times 6$

Multi-Dimensional Arrays



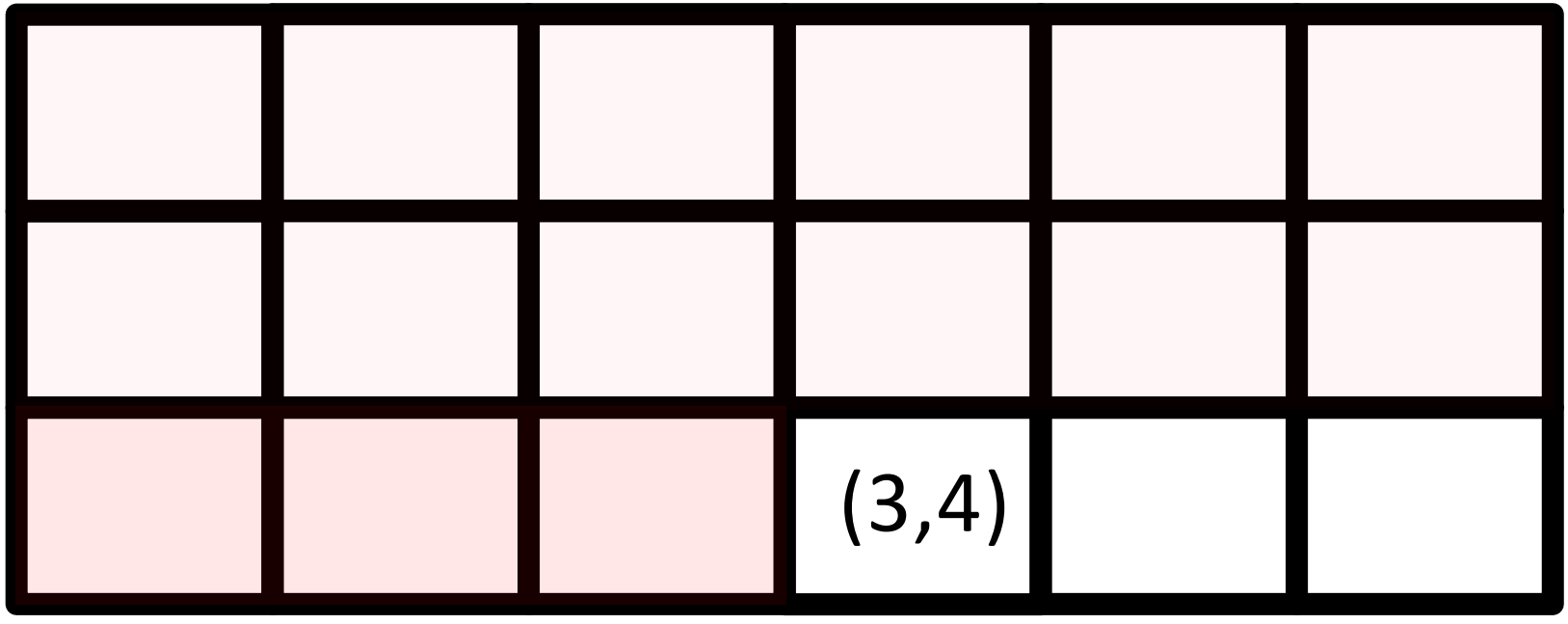
$$(3 - 1) \times 6 + (4 - 1)$$

Multi-Dimensional Arrays



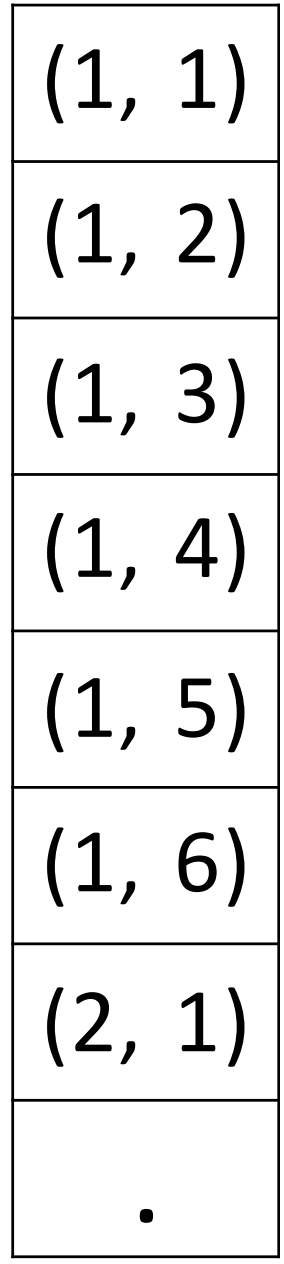
$$\text{elem\_size} \times ((3 - 1) \times 6 + (4 - 1))$$

Multi-Dimensional Arrays

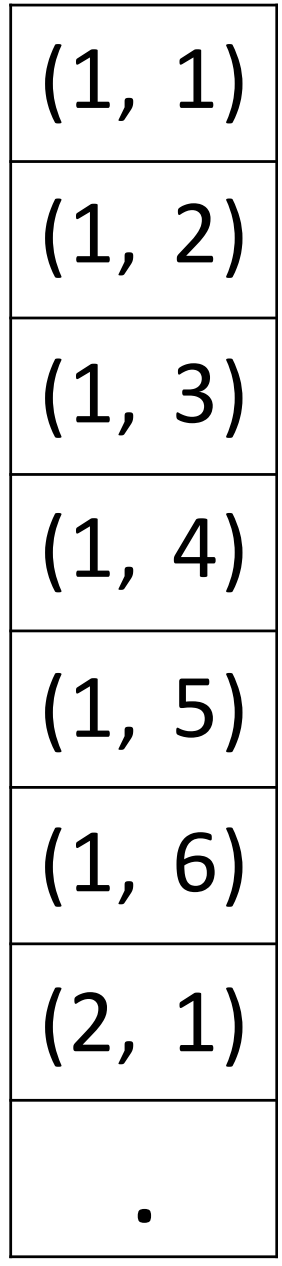


array_addr +

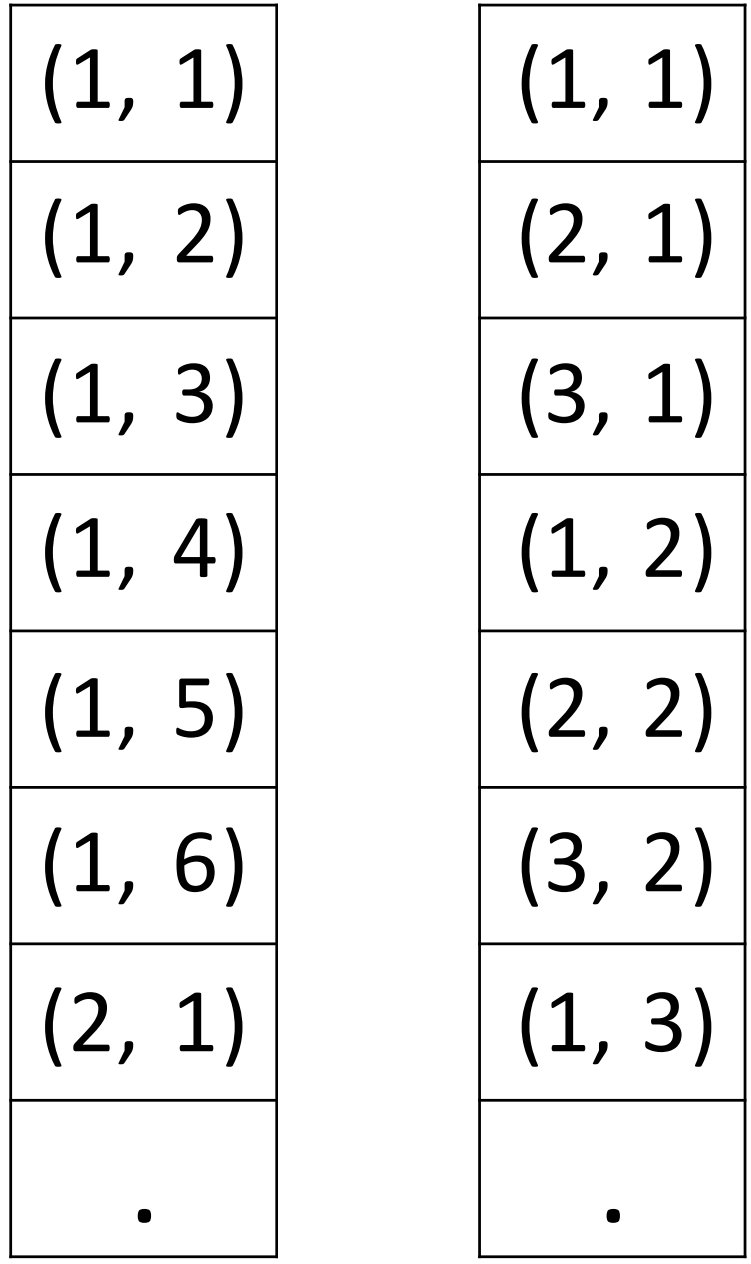
elem_size × ((3 − 1) × 6 + (4 − 1))

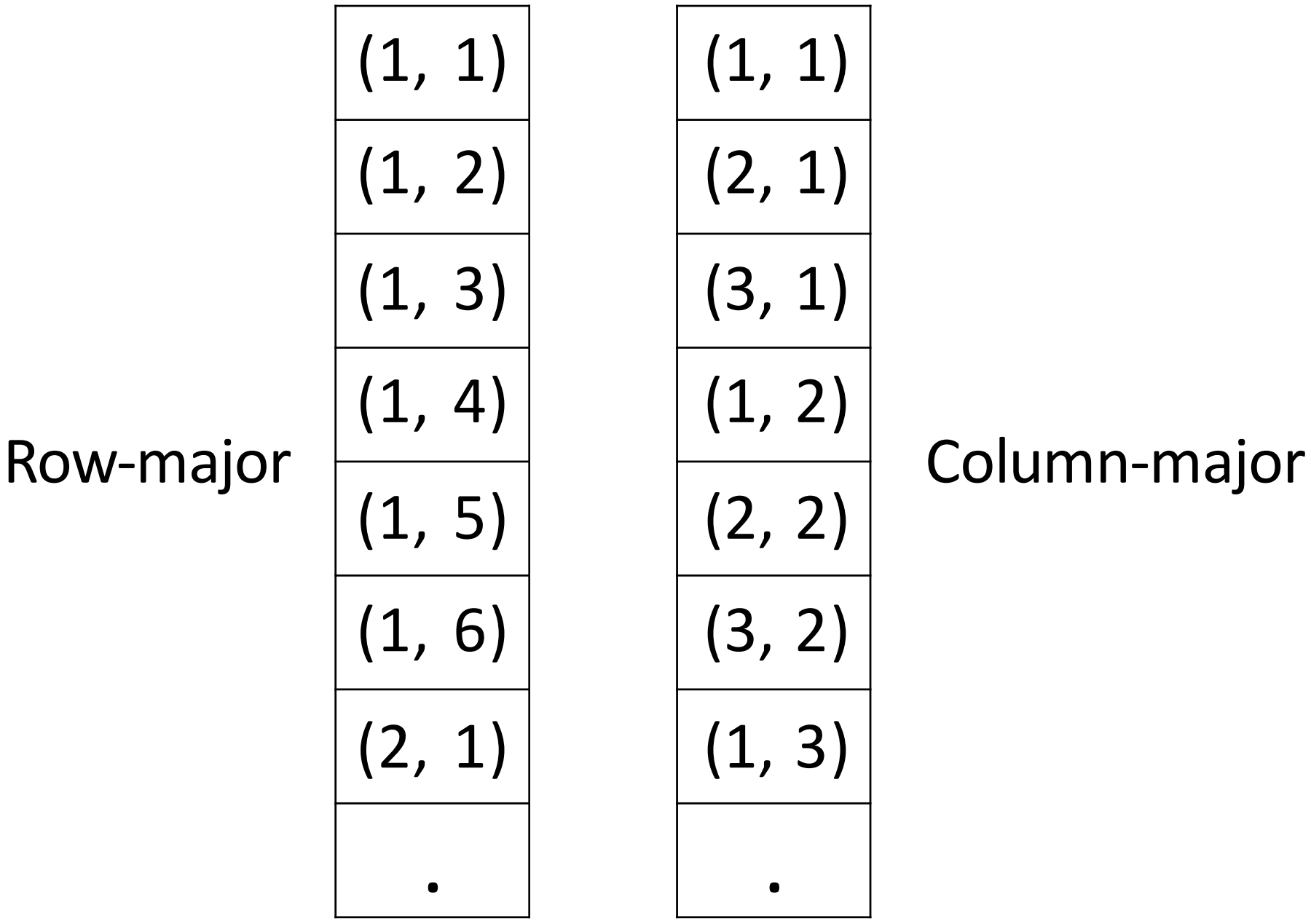

(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 1)
.

Row-major



Row-major



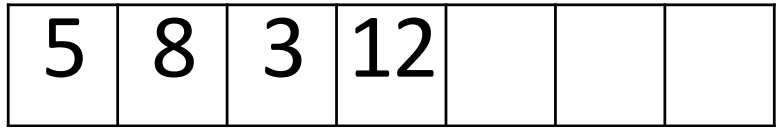

Row-major
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 1)
.
(1, 1)
(2, 1)
(3, 1)
(1, 2)
(2, 2)
(3, 2)
(1, 3)
.
Column-major

Times for Common Operations

	Add	Remove
Beginning		
End		
Middle		

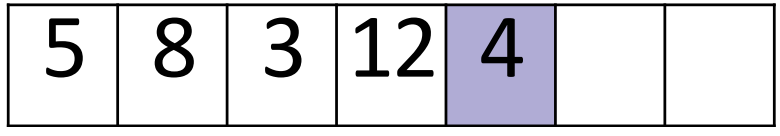
Times for Common Operations

	Add	Remove
Beginning		
End		
Middle		



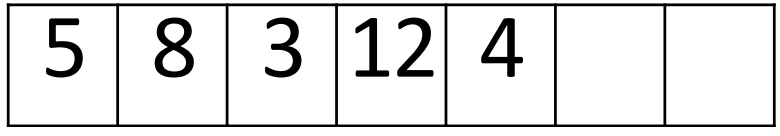
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	
Middle		



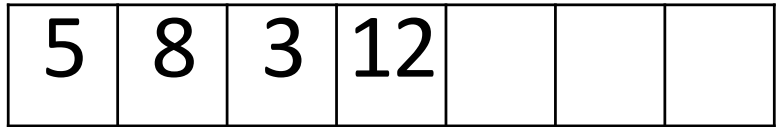
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	
Middle		



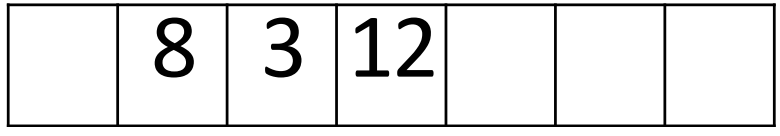
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	$O(1)$
Middle		



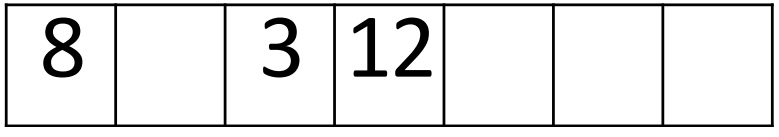
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



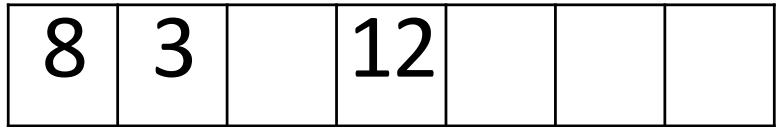
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



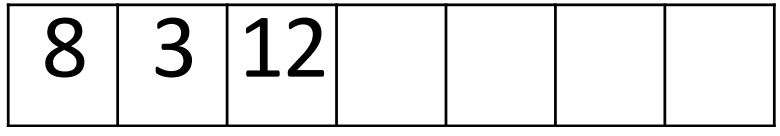
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



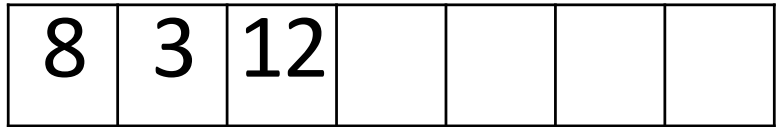
Times for Common Operations

	Add	Remove
Beginning		$O(n)$
End	$O(1)$	$O(1)$
Middle		



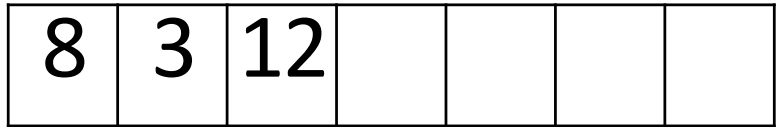
Times for Common Operations

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle		



Times for Common Operations

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$



What we can store in Arrays?

Primitive data types

- Integer
- Double
- Char

- Homogeneous data types

Summary

Summary

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.

Summary

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.

Summary

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.

Summary

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.
- Linear time to add/remove at an arbitrary location.

Mental Coding

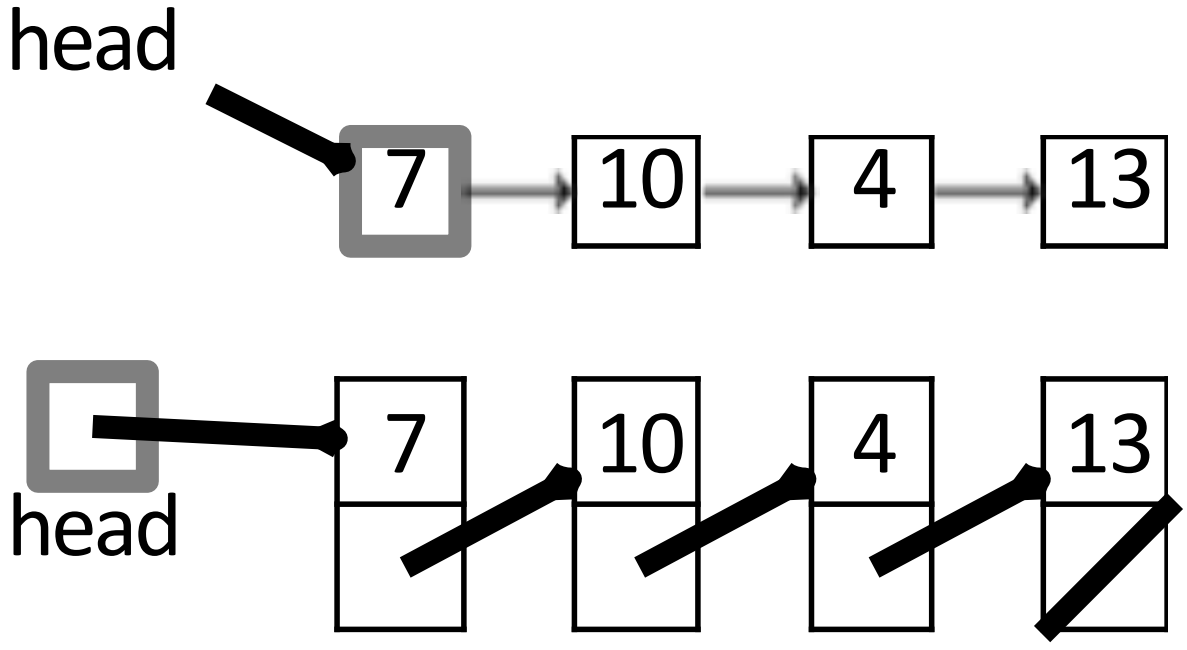
- Check whether a given string a palindrome or not

  “madam”

Outline

1. [illegible]
2. Linked Lists

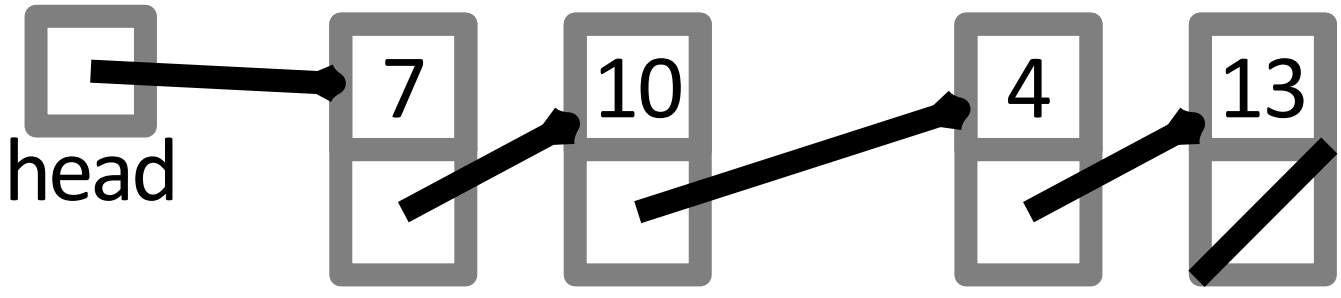
Singly-Linked List



Node contains:

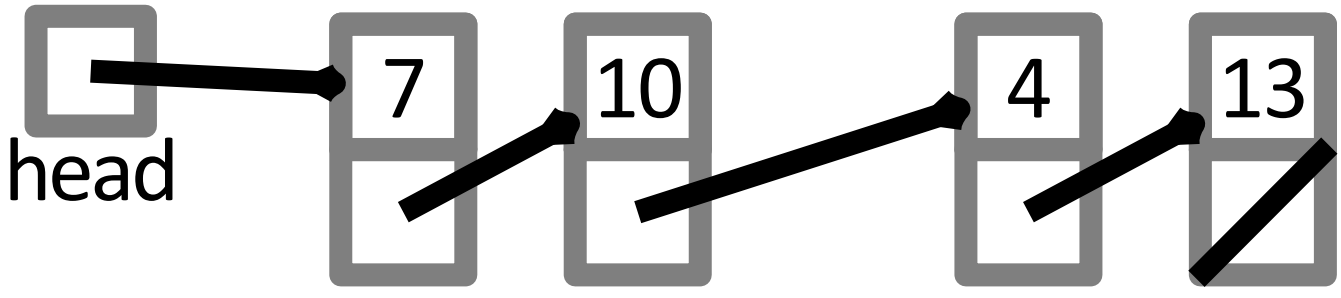
- `key`
- `next` pointer

Times for Some Operations



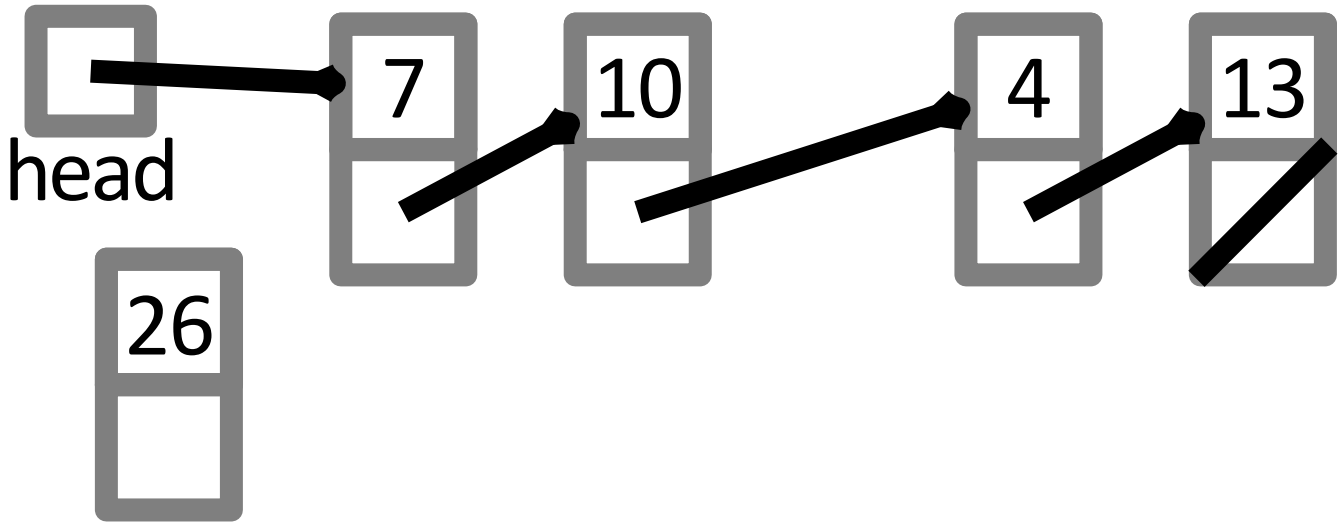
Times for Some Operations

`PushFront`



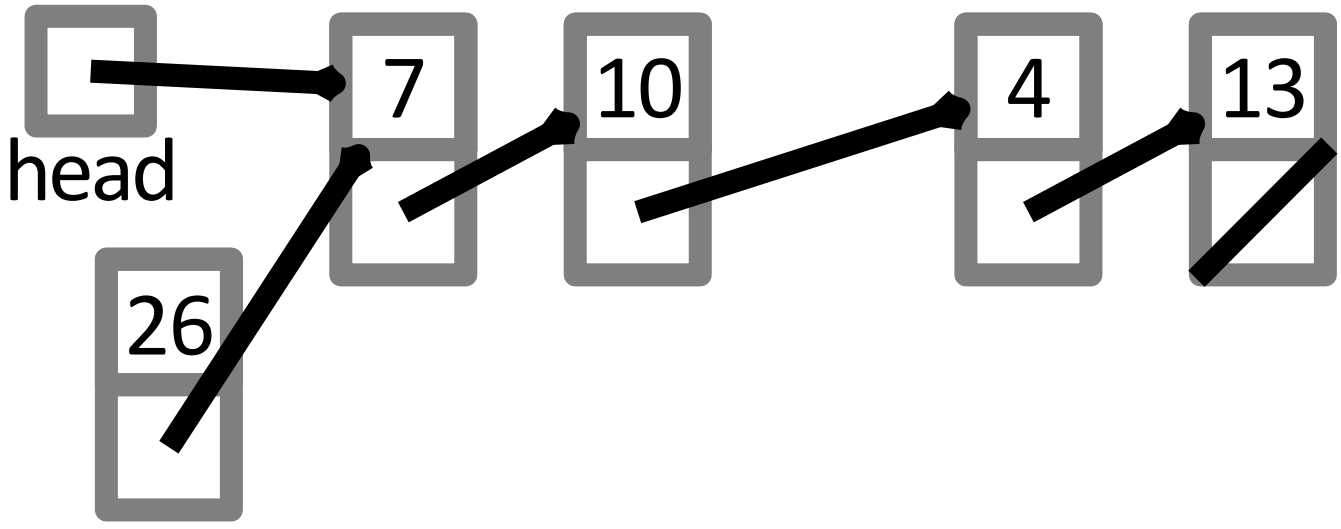
Times for Some Operations

`PushFront`



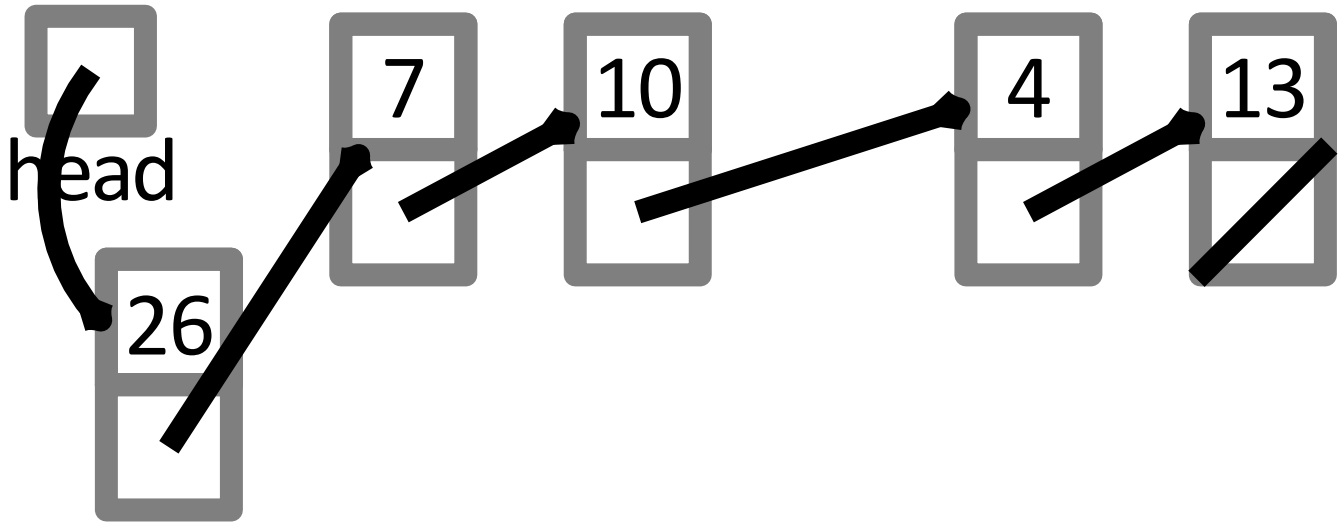
Times for Some Operations

`PushFront`



Times for Some Operations

`PushFront` $O(1)$



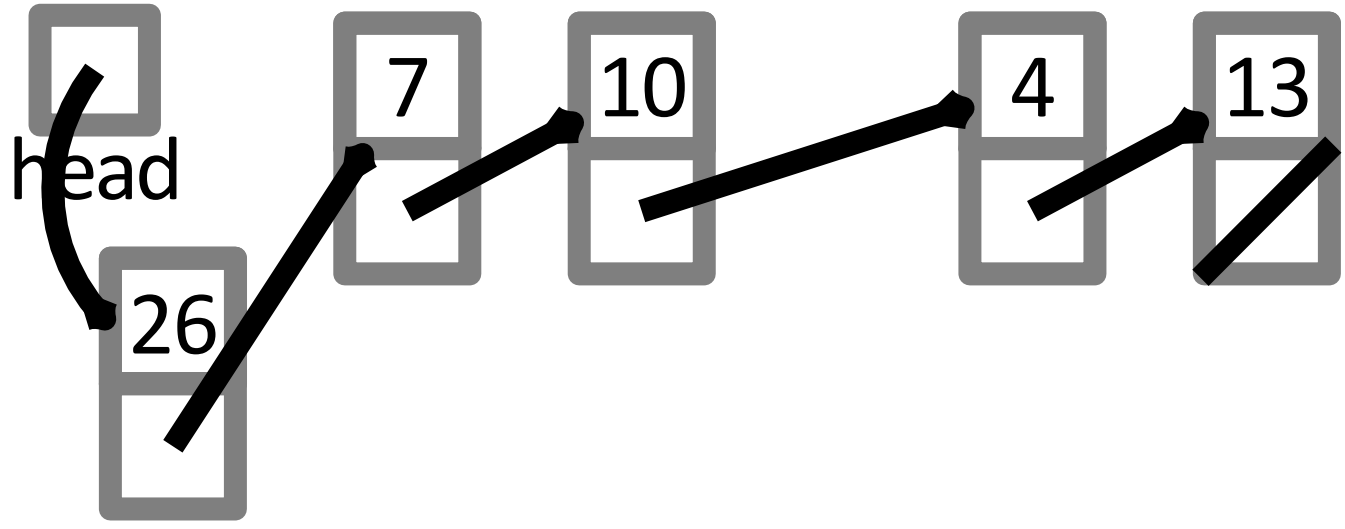
Singly-linked List

PushFront(*key*)

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

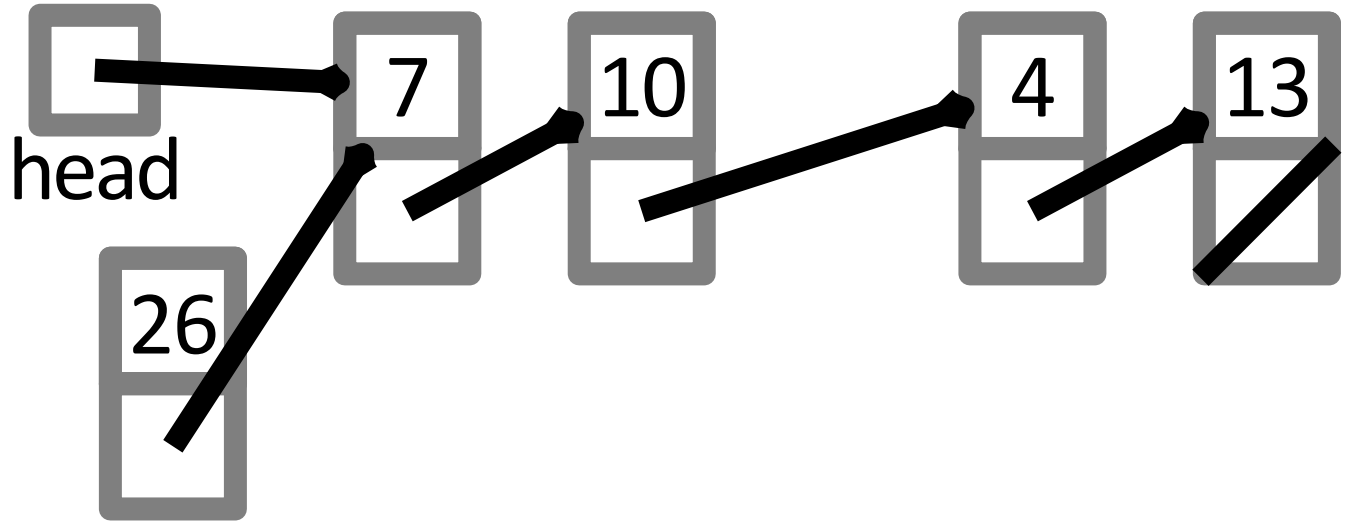
Times for Some Operations

`PopFront`



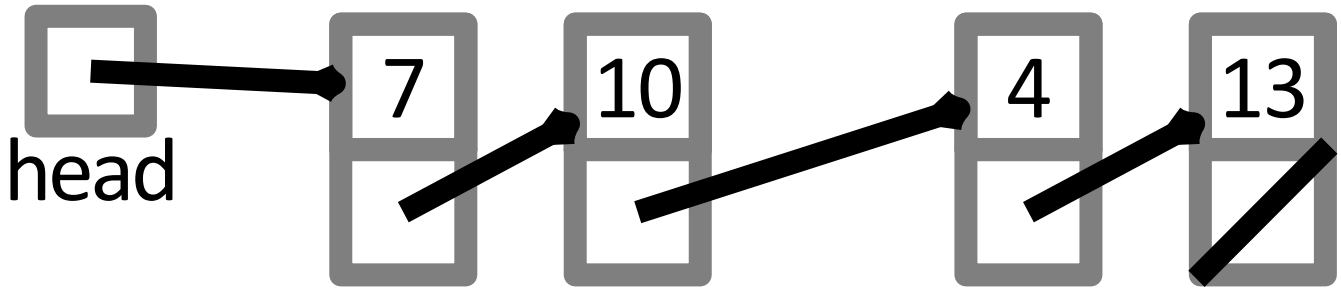
Times for Some Operations

`PopFront`



Times for Some Operations

`PopFront` $O(1)$



Singly-linked List

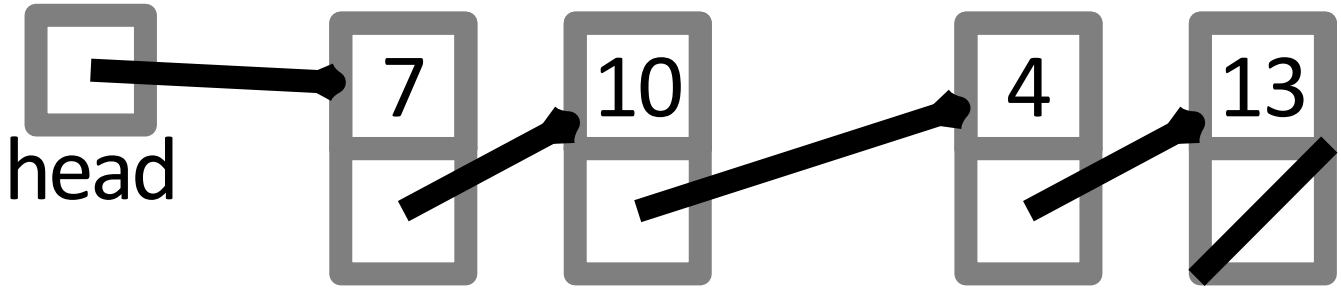
PopFront()

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

Times for Some Operations

`PushBack`

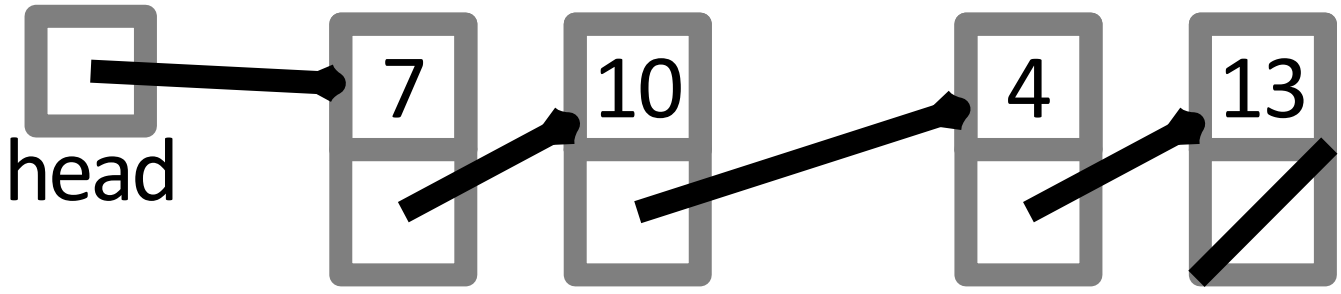
(no tail)



Times for Some Operations

`PushBack` $O(n)$

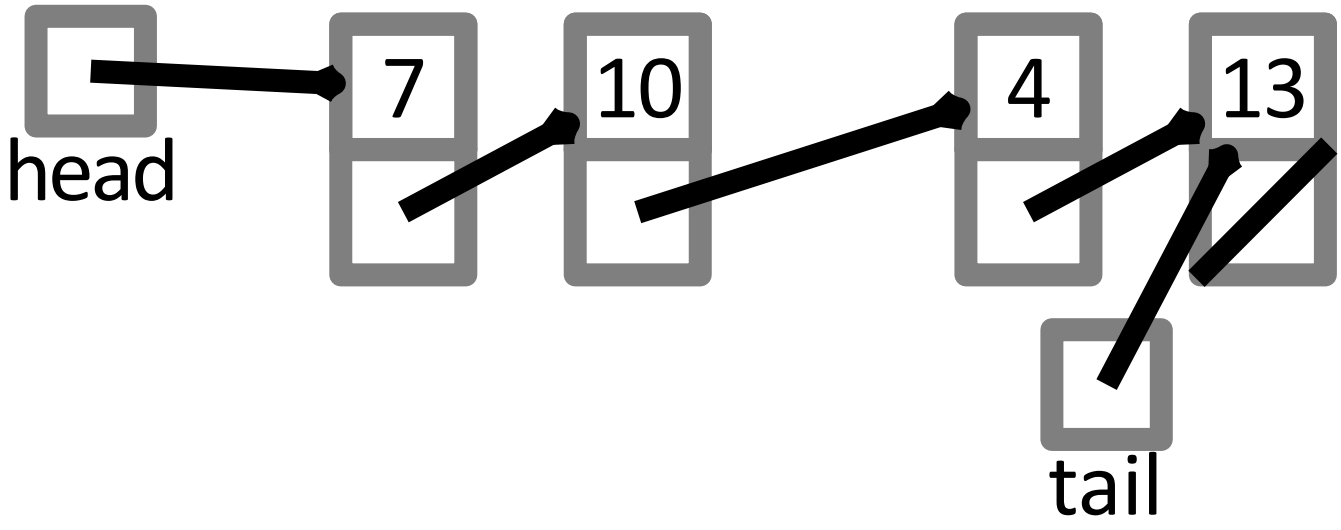
(no tail)



Times for Some Operations

`PushBack`

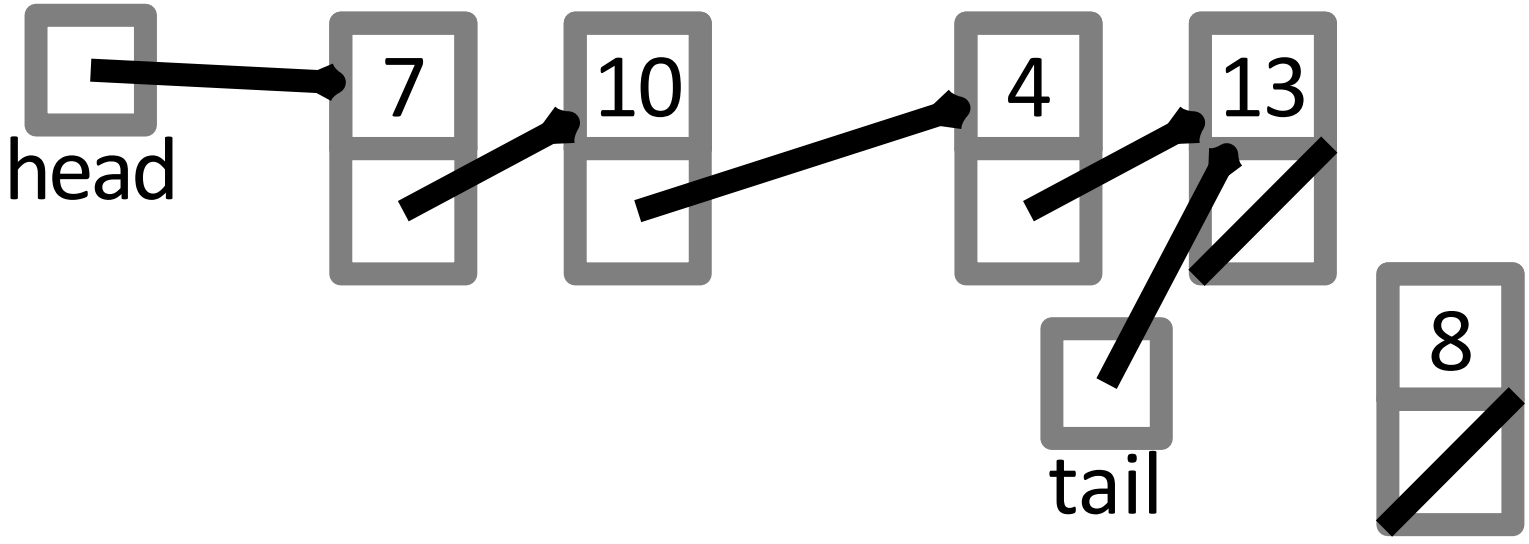
(with tail)



Times for Some Operations

`PushBack`

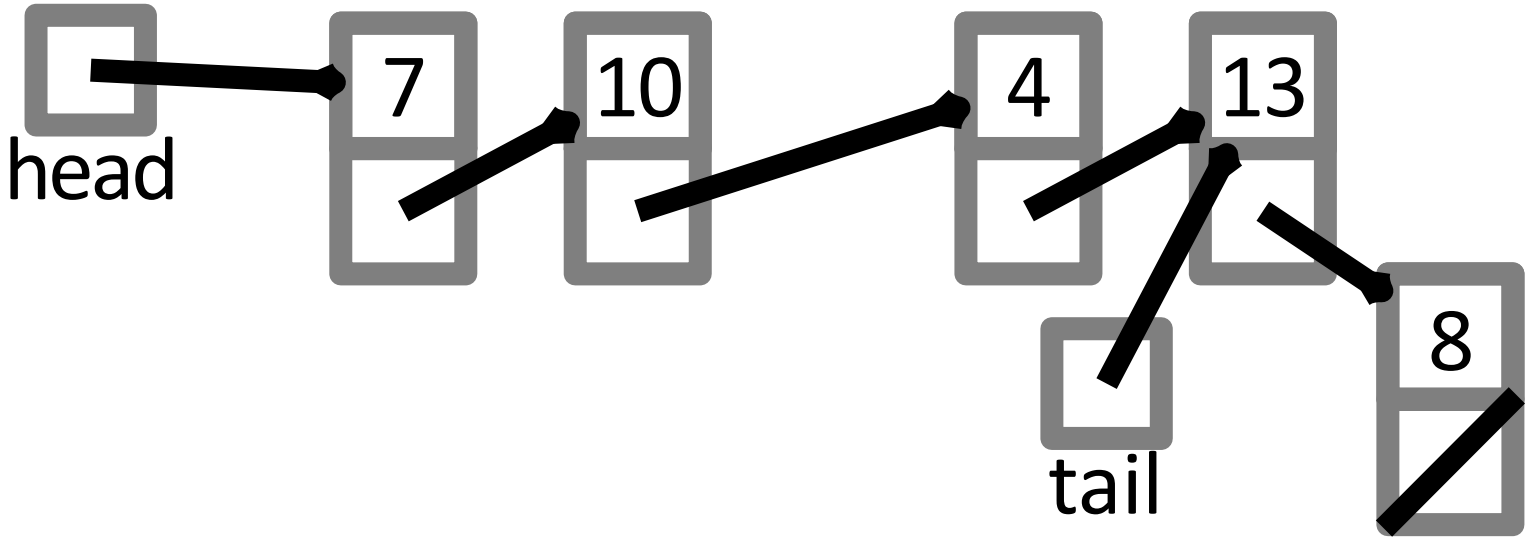
(with tail)



Times for Some Operations

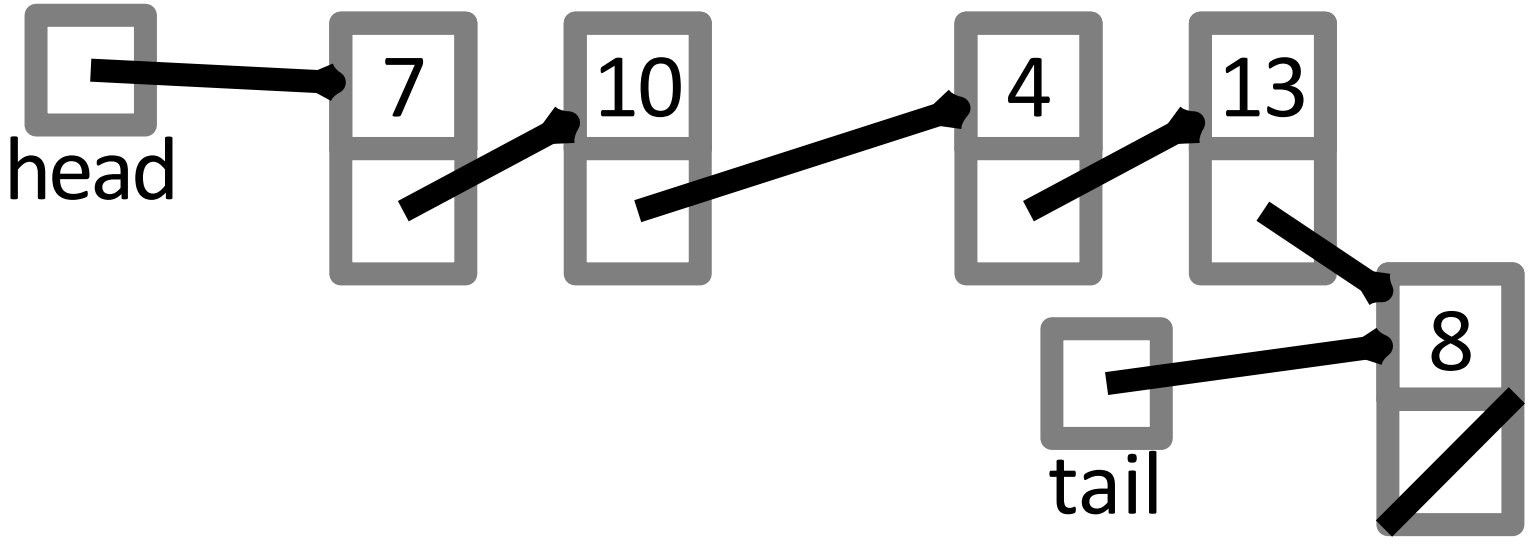
`PushBack`

(with tail)



Times for Some Operations

`PushBack` $O(1)$
(with tail)



Singly-linked List

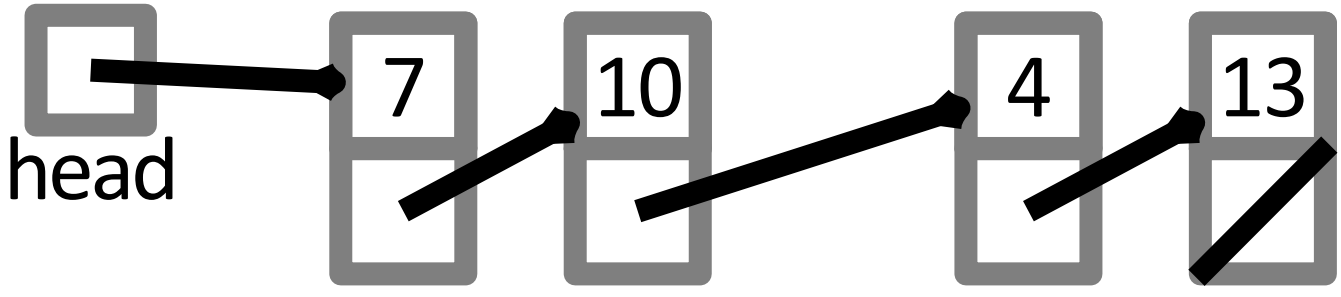
PushBack(*key*)

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

Times for Some Operations

`PopBack`

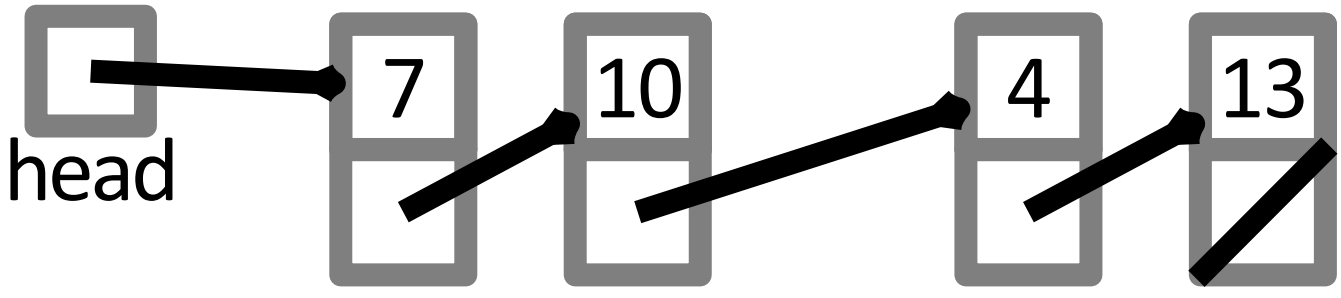
(no tail)



Times for Some Operations

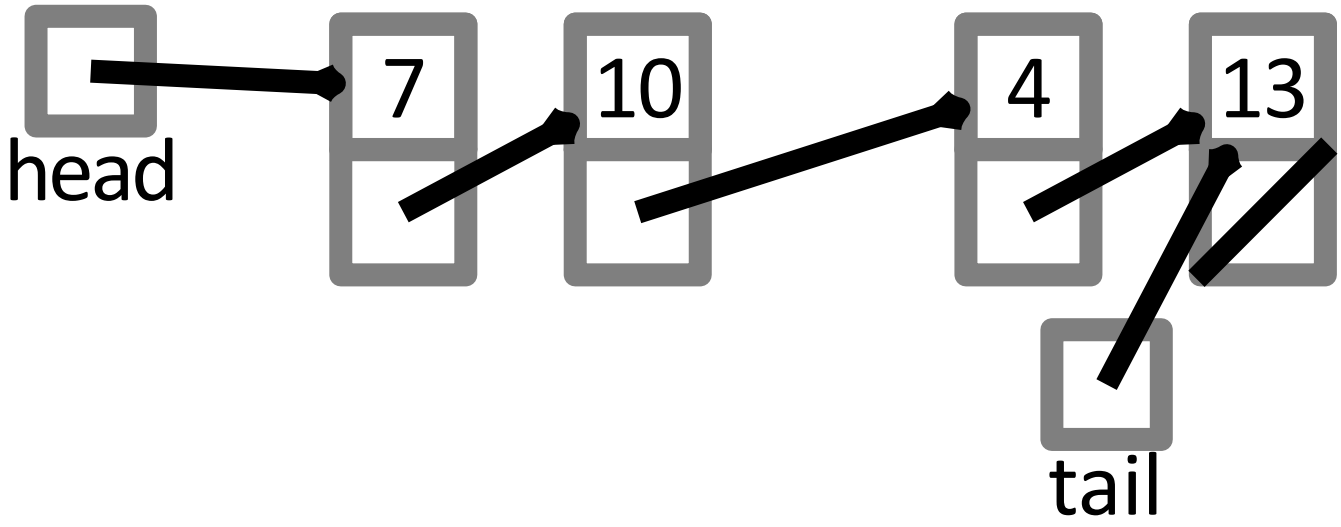
`PopBack` $O(n)$

(no tail)



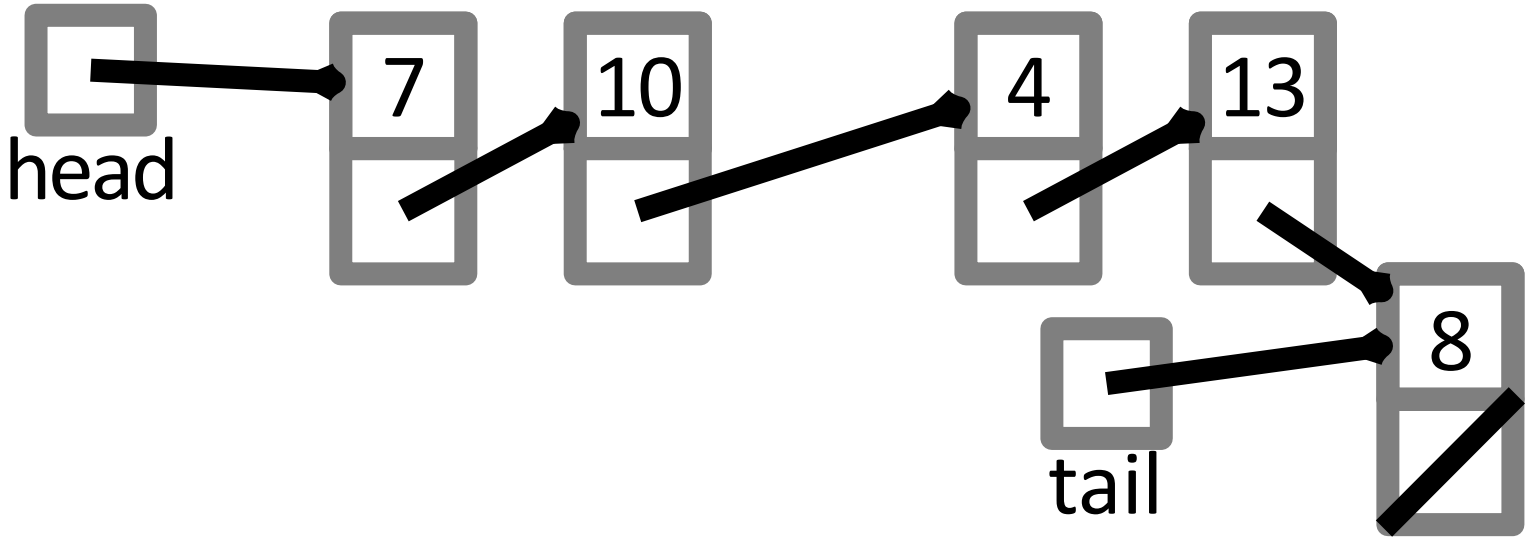
Times for Some Operations

`PopBack`
(with tail)



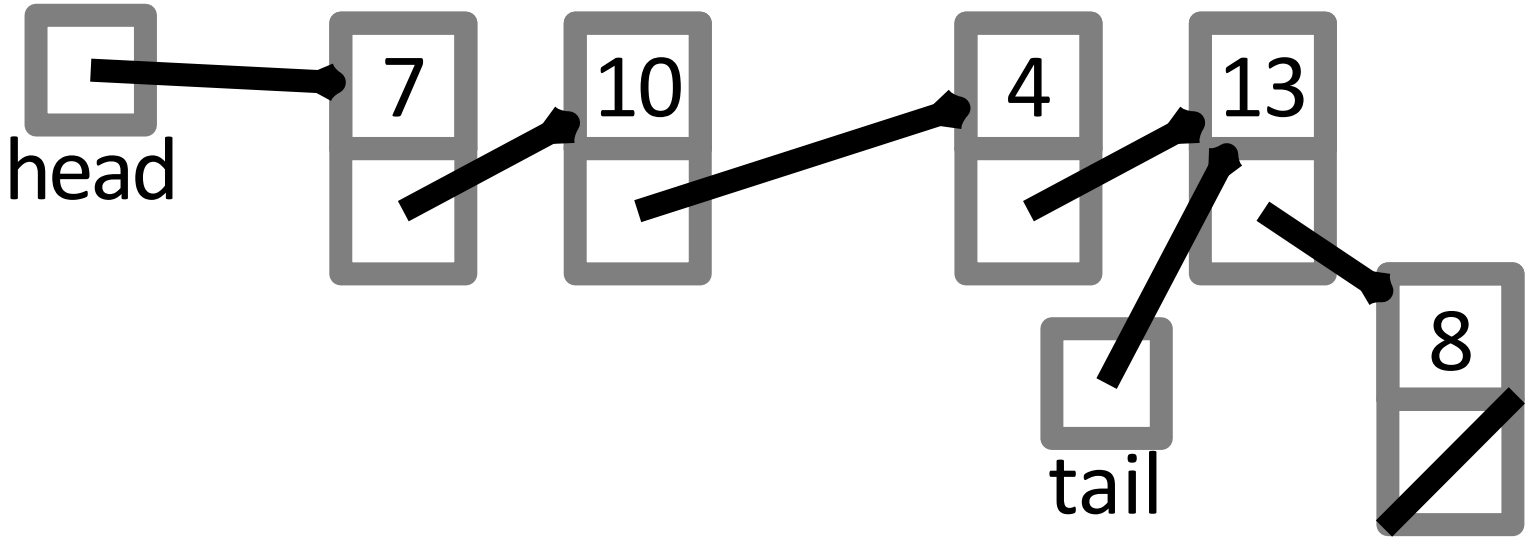
Times for Some Operations

`PopBack`
(with tail)



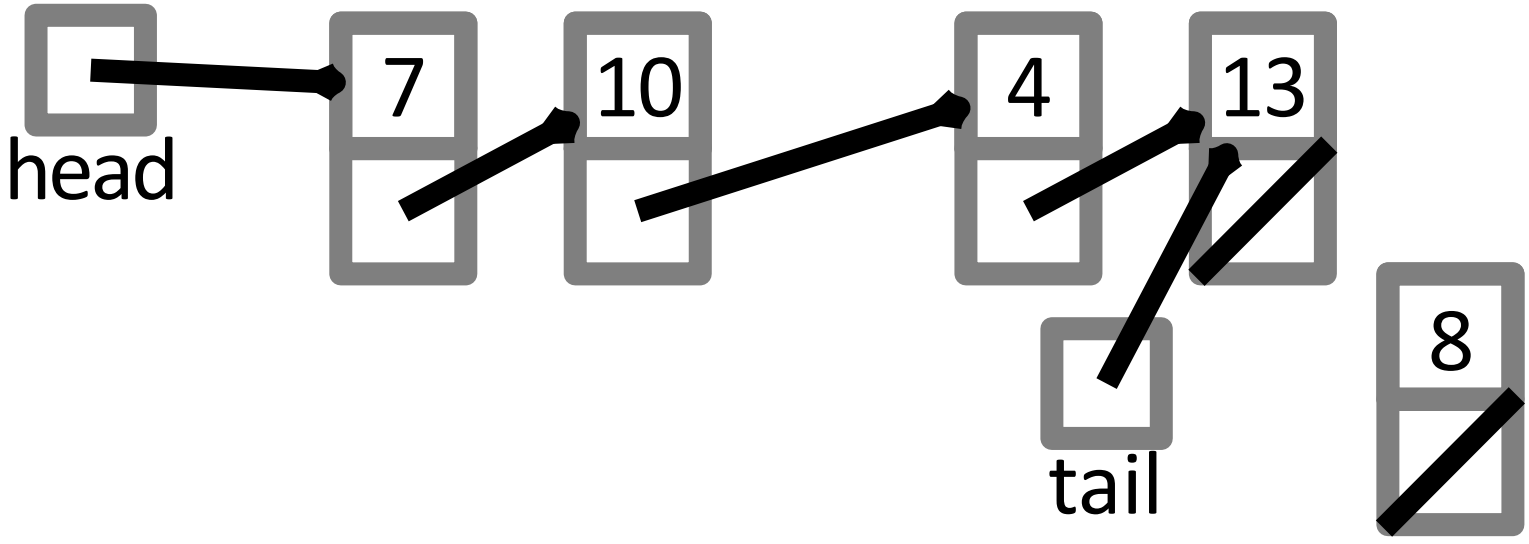
Times for Some Operations

`PopBack`
(with tail)



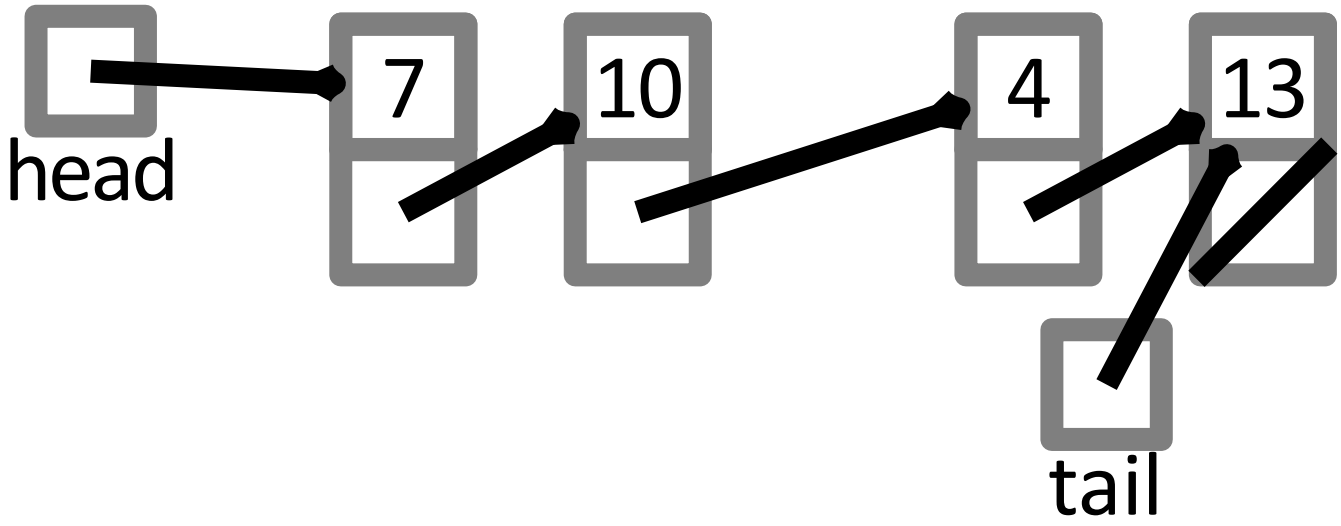
Times for Some Operations

`PopBack`
(with tail)



Times for Some Operations

`PopBack` $O(1)$
(with tail)



Singly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
if head = tail :
   head ← tail ←nil
else:
   p ← head
   while p.next.next != nil:
      p ← p.next
   p.next ← nil;  tail ← p
```

Singly-linked List

AddAfter(*node*, *key*)

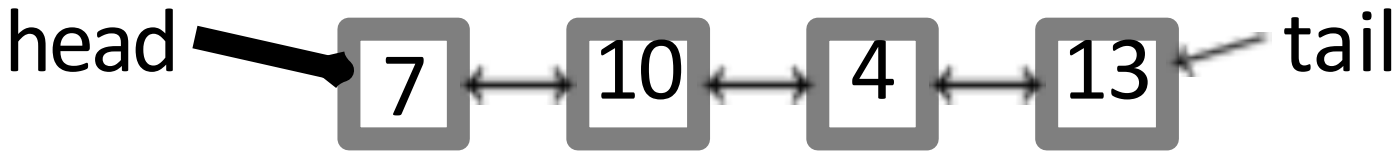
```
node2 ←new node
node2.key ← key
node2.next = node.next
node.next = node2
if tail = node:
   tail ← node2
```

Linked List Operations

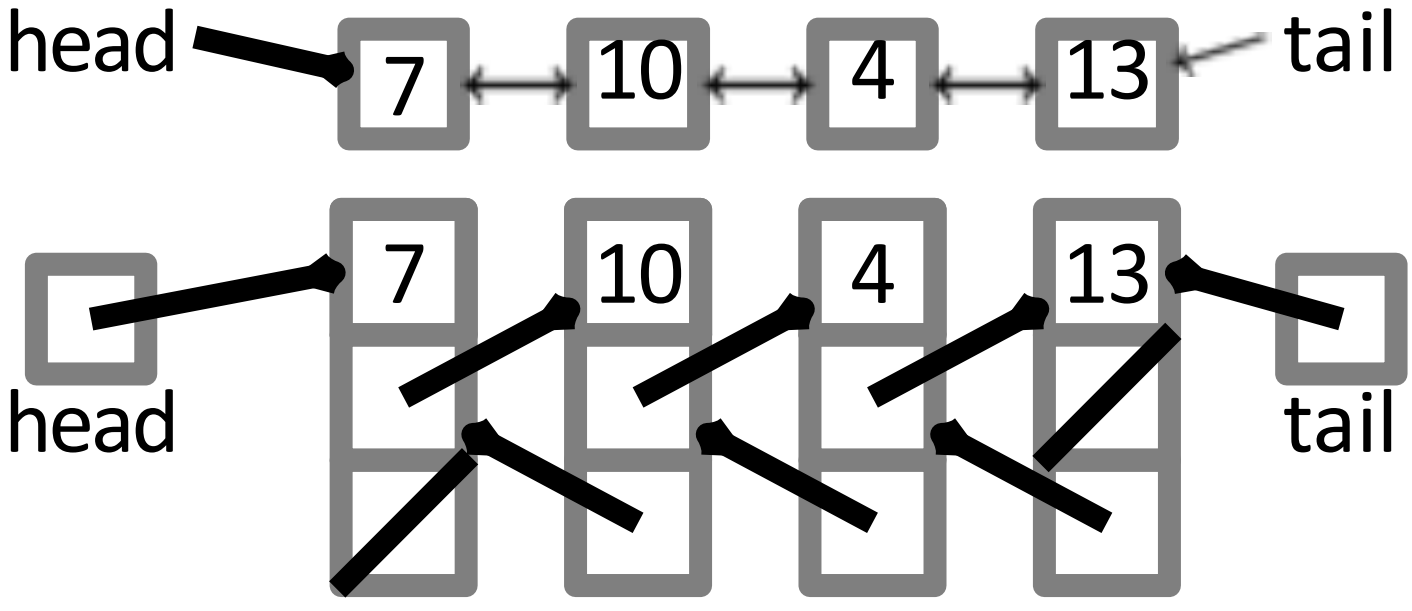
`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`TopBack()`	return back item
`PopBack()`	remove back item
`Find(Key)`	is key in list?
`Erase(Key)`	remove key from list
`Boolean Empty()`	empty list?
`AddBefore(Node, Key)`	adds key before node
`AddAfter(Node, Key)`	adds key after node

Singly-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)	$O(n)$	
AddAfter(Node, Key)	$O(1)$	

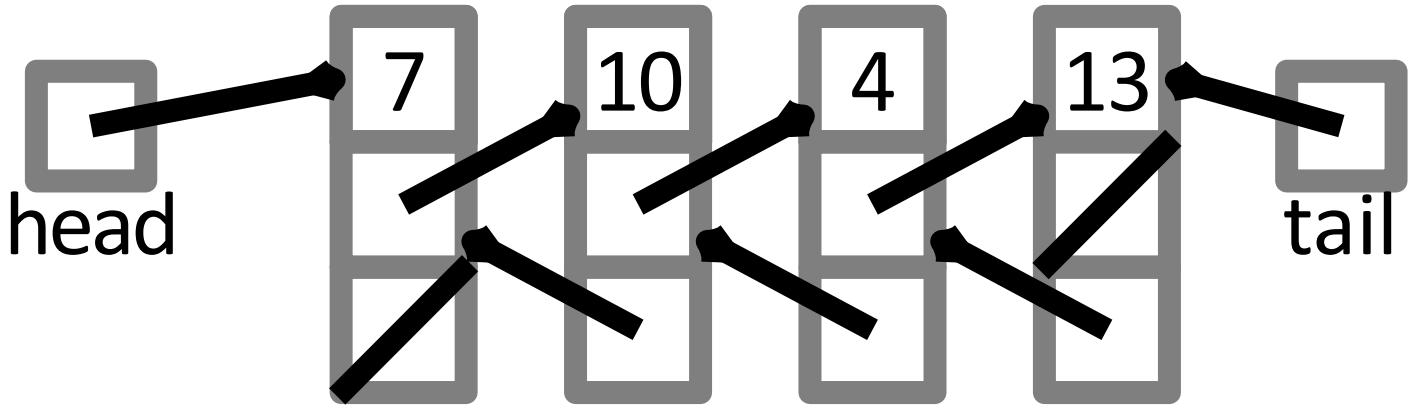
Doubly-Linked List



Doubly-Linked List



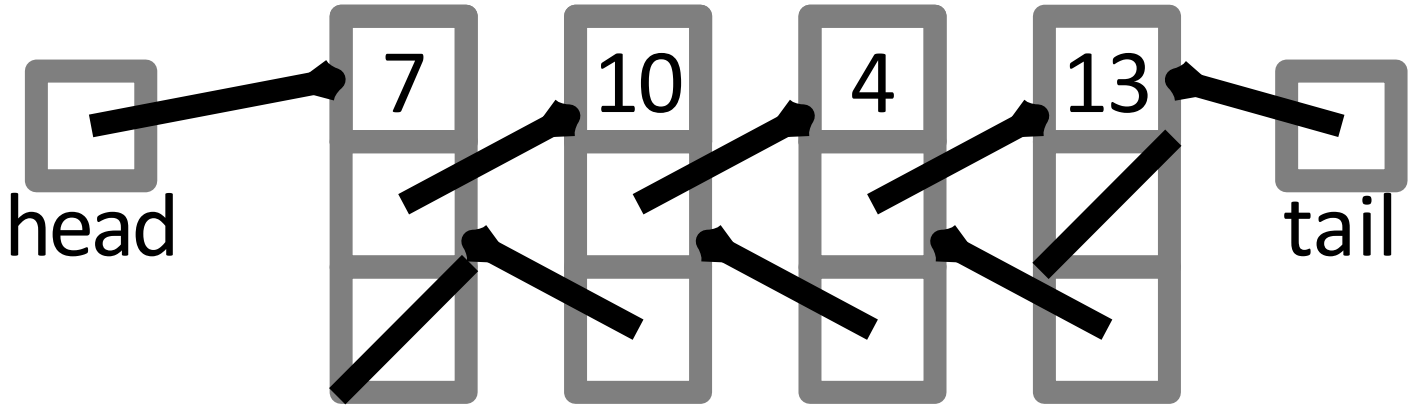
Doubly-Linked List



Node contains:

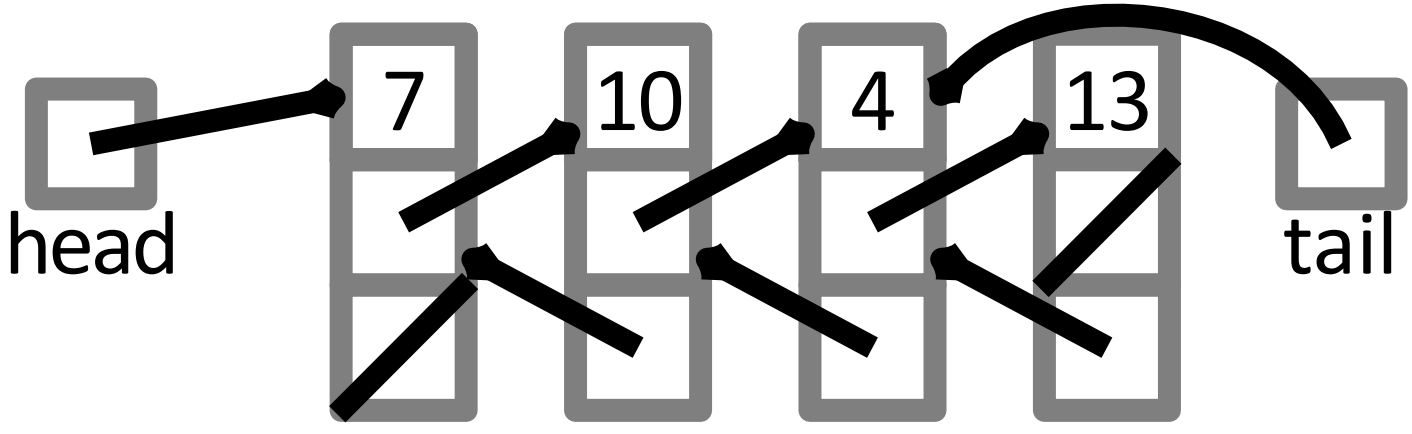
- `key`
- `next` pointer
- `prev` pointer

Doubly-Linked List



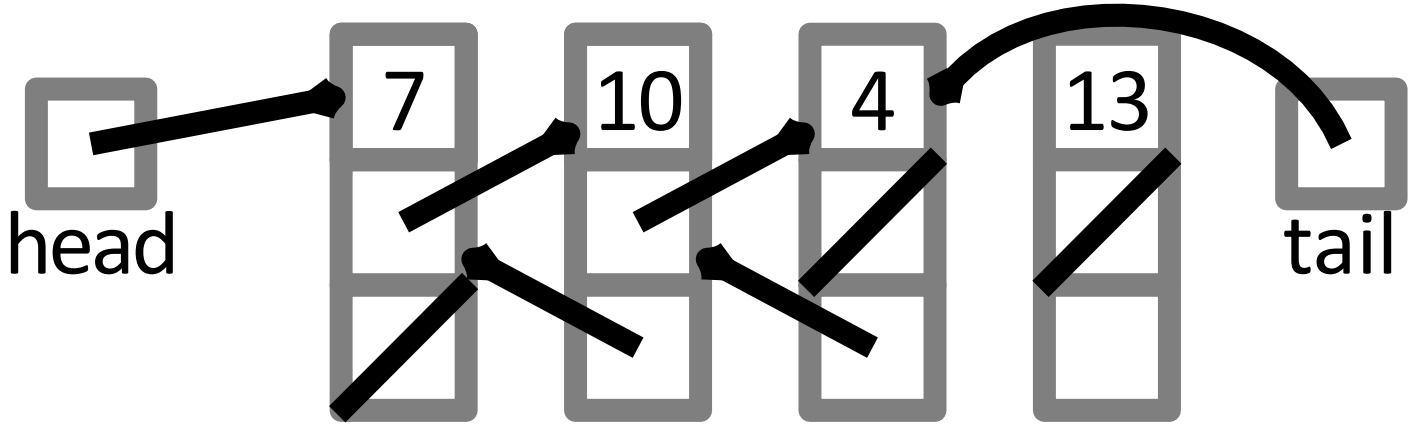
PopBack

Doubly-Linked List



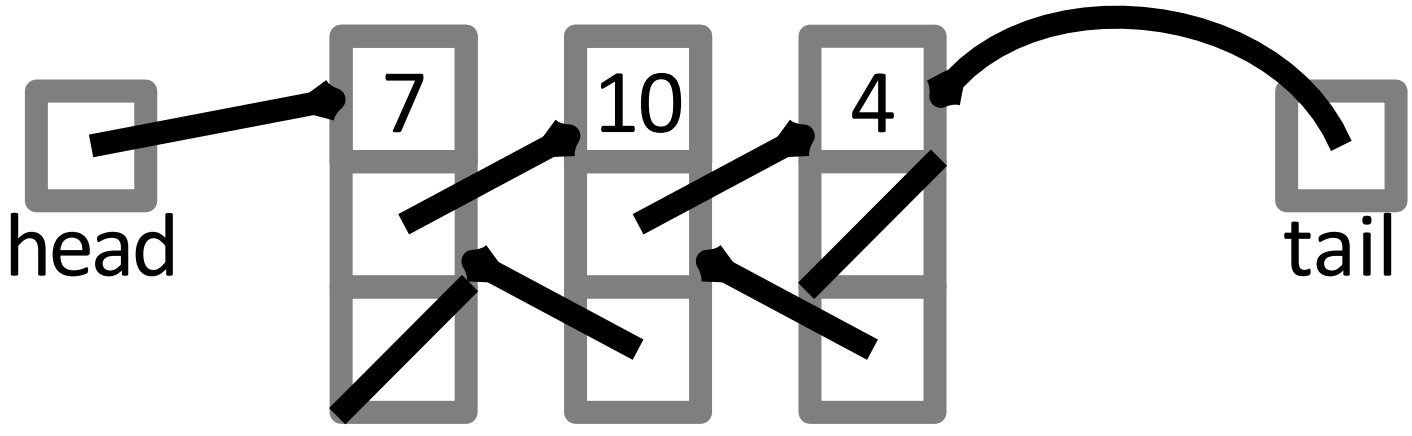
PopBack

Doubly-Linked List



PopBack

Doubly-Linked List



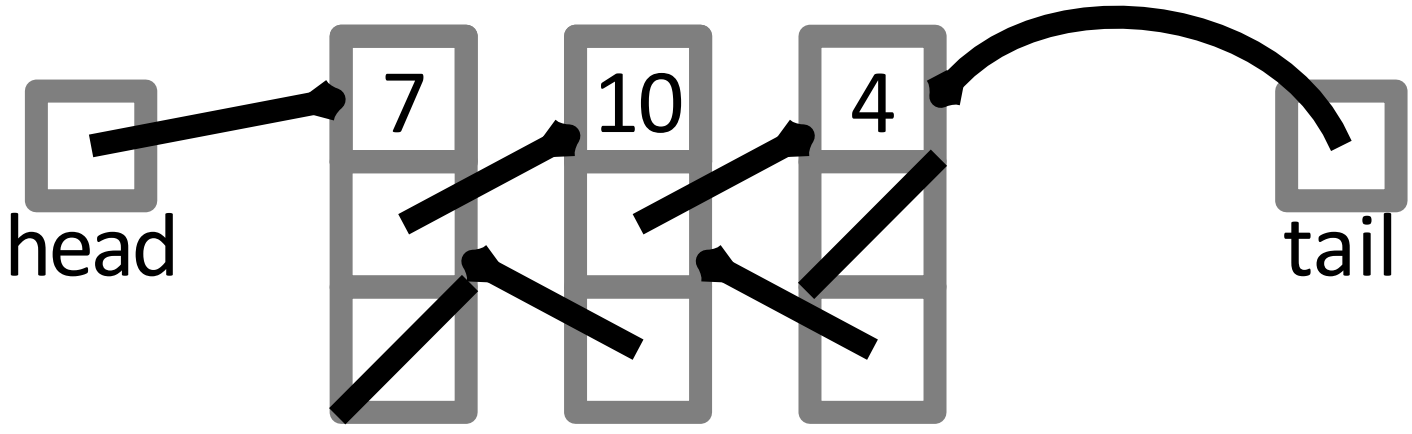
PopBack

Doubly-linked List

PopBack()

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

Doubly-Linked List



`PopBack` $O(1)$

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

AddAfter(*node, key*)

```
node2 ←new node
node2.key ← key
node2.next ← node.next
node2.prev ← node
node.next ← node2
if node2.next !=nil:
   node2.next.prev ← node2
if tail = node:
   tail ← node2
```

Doubly-linked List

AddBefore(*node, key*)

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

Singly-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()	$O(n)$	
Find(Key)	$O(n)$	
Erase(Key)	$O(n)$	
Empty()	$O(1)$	
AddBefore(Node, Key)	$O(n)$	
AddAfter(Node, Key)	$O(1)$	

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)`	$O(1)$	

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.

Assignments

1. Print the Middle of a given linked list
2. Union and Intersection of two Linked Lists
3. Find pairs with given sum in doubly linked list Input : head : 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9
   - x = 7
   - Output: (6, 1), (5,2)

4. Delete the elements in an linked list whose sum is equal to zero

5. You are given a Linked List and a number *K*. You have to reverse it in the groups of K
   - Ex : [1] -> [2] -> [3] -> [4] -> [5] -> null,
   - K = 3
   - output: [3] -> [2] -> [1] -> [5] -> [4] -> null

Assignments

7. Given number *k*, for Single linked list, skip *k* nodes and then reverse k nodes, till the end.
8. Having a List of int [1,1,1,3,1,2,1,1,4,1] Output needed [4,5,6,3,7,2,8,9,4,10] Note: Need not to change value of 3,2,4
9. Reverse a pair of elements in a linked list. abcd – badc
10. Remove duplicate nodes in an unsorted linked list.