CSE 225: Data Structures and Algorithms

Course Outline

Module 1:	Basic Data Structures
	- Array and Linked List
	- Stacks and Queues
	- Trees
Module 2:	Dynamic Array and Amortized Analysis
Module 3:	Priority Queues and Disjoint Sets
Module 4:	Hash Tables
Module 5:	Binary Search Trees
Module 6:	Graphs

Course Outline

Module 1:	Basic Data Structures
	- Array and Linked List
	- Stacks and Queues
	- Trees
Module 2:	Dynamic Array and Amortized Analysis
Module 3:	Priority Queues and Disjoint Sets
Module 4:	Hash Tables
Module 5:	Binary Search Trees
Module 6:	Graphs

Course Outcomes(COs

Upon Successful completion of this course, students will be able to:

Sl.	CO Description	Weightage (%)
1	Understand the fundamental Data Structures including arrays, linked lists, trees, binary search trees, stacks, queues, priority queues, graphs, and hash tables.	70%
2	Identify appropriate data structures based on algorithmic complexity for solving real-world problems	10%
3	Use programming tools for the implementation of abstract data types (ADT)	20%

Basic Data Structures: Arrays and Linked Lists

Data Structures

```
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

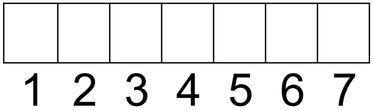
Array:

Contiguous area of memory consisting of equal-size elements

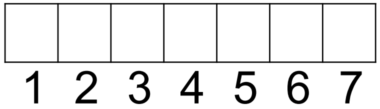
Definition

Array:

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

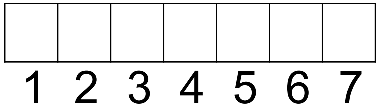


What’s Special About Arrays?



What’s Special About Arrays?

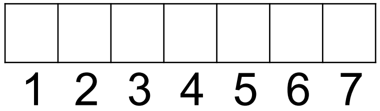
Constant-time access



What’s Special About Arrays?

Constant-time access

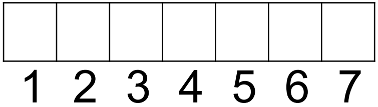
array_addr



What's Special About Arrays?

Constant-time access

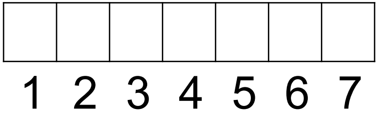
array_addr + elem_size × ( )



What's Special About Arrays?

Constant-time access

array_addr + elem_size × (i − first_index)



Question

Given an array whose:
•address is 1000,
•element size is 8
•first index is 0
What is the address of the element at index 6?

40

48

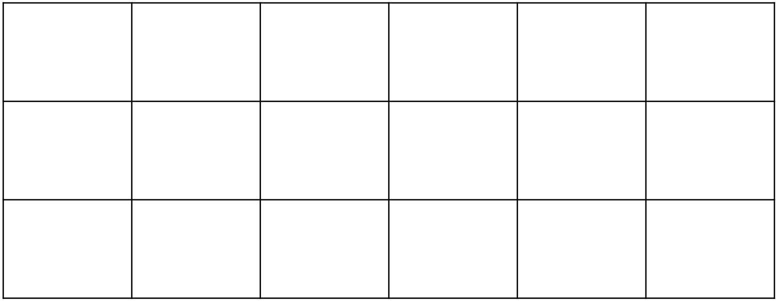
1048

1040

1006

1005

Multi-Dimensional Arrays



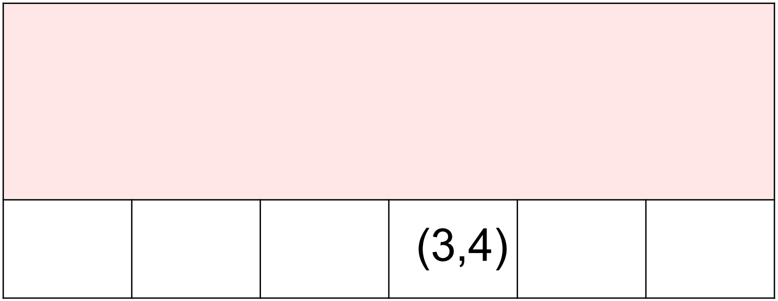
Multi-Dimensional Arrays

(1, 1)					

Multi-Dimensional Arrays

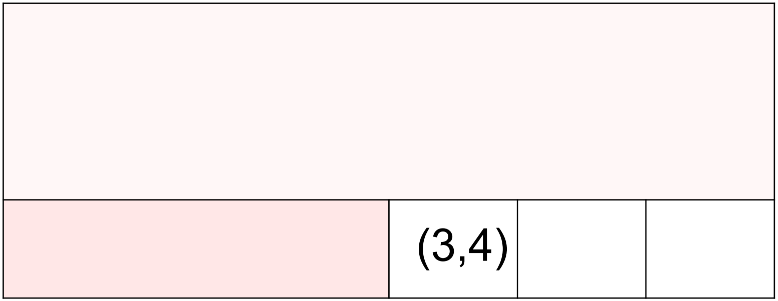
			(3,4)		

Multi-Dimensional Arrays



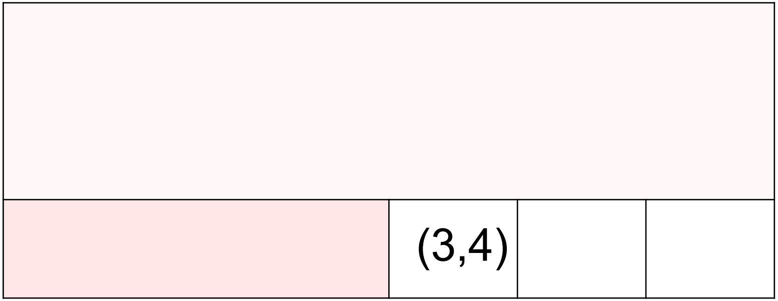
$(3 - 1) \times 6$

Multi-Dimensional Arrays



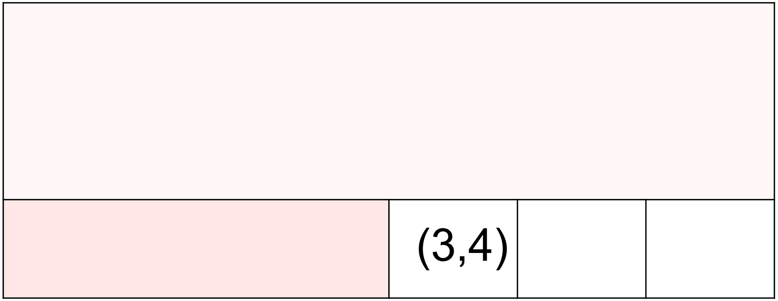
$$(3 - 1) \times 6 + (4 - 1)$$

Multi-Dimensional Arrays



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

Multi-Dimensional Arrays

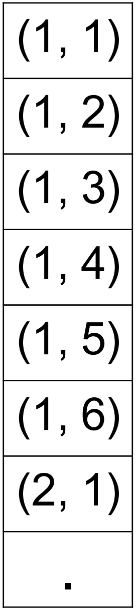


array_addr +

elem_size × ((3 − 1) × 6 + (4 − 1))

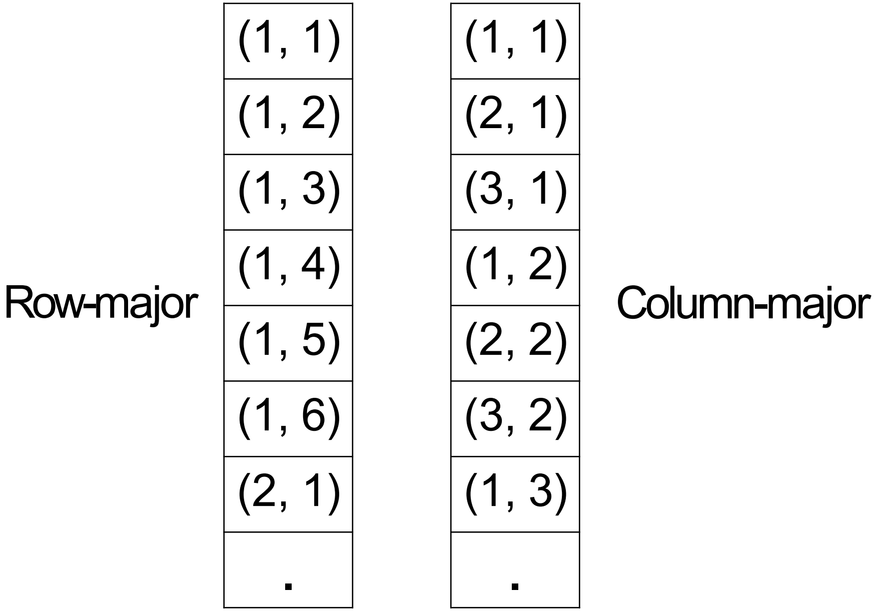
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 1)
.

Row-major



Row-major

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


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

Question

Assume you have a three-dimensional array laid out in column-major order with the first element at indices (1, 1, 1). What are the indices of the next element in memory?

- ○ (1, 2, 1)
- ○ (2, 1, 1)
- ○ (1, 1, 2)

Times for Common Operations

	Add	Remove
Beginning		
End		
Middle		

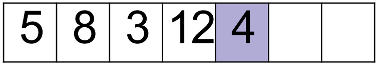
Times for Common Operations

	Add	Remove
Beginning		
End		
Middle		

5	8	3	12			

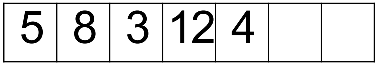
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	
Middle		



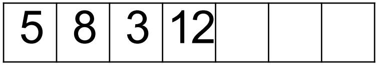
Times for Common Operations

	Add	Remove
Beginning		
End	$O(1)$	
Middle		



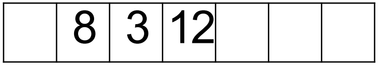
Times for Common Operations

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



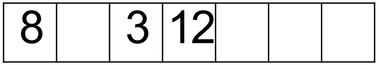
Times for Common Operations

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



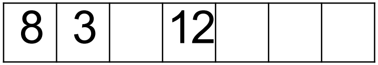
Times for Common Operations

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



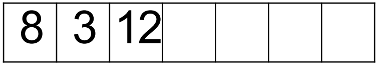
Times for Common Operations

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



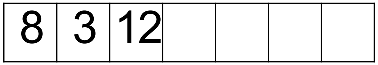
Times for Common Operations

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



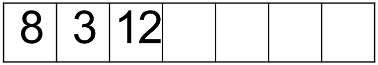
Times for Common Operations

	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	$O(n)$	$O(n)$



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

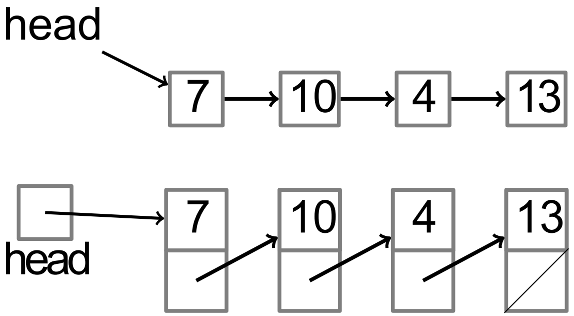
- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.

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.

Outline

Singly-Linked List



Node contains:

- `key`
- `next` pointer

List API

`PushFront(Key)`	add to front

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
	also known as Append

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item
`Boolean Find(Key)`	is key in list?

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item
`Boolean Find(Key)`	is key in list?
`Erase(Key)`	remove key from list

List API

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item
`Boolean Find(Key)`	is key in list?
`Erase(Key)`	remove key from list
`Boolean Empty()`	empty list?

List API

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

List API

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

Question

You have an empty list, and then do the following operations:

```
PushBack(a)
PushFront(b)
PushBack(d)
PushFront(c)
PopBack()
```

What is the contents of the list now?

c, b, a

c, b, a, d

a, b, c

Question

You have an empty list, and then do the following operations:

```
PushBack(a)
PushFront(b)
PushBack(d)
PushFront(c)
PopBack()
```

Here are the list contents after each operation;

PushBack(a) -> a

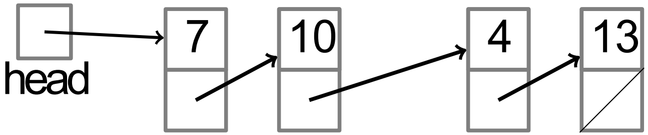
PushFront(b) -> b, a

PushBack(d) -> b, a, d

PushFront(c) -> c, b, a, d

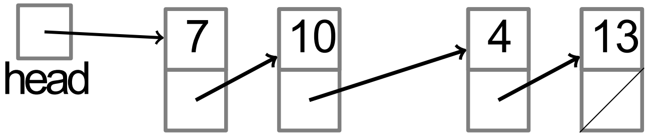
PopBack() -> c, b, a

Times for Some Operations



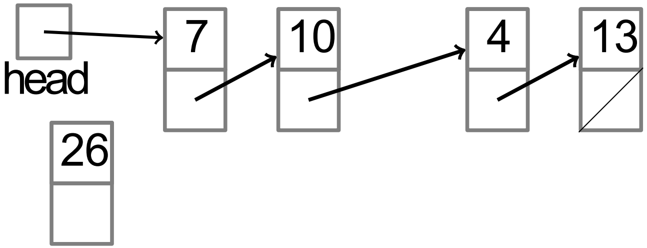
Times for Some Operations

`PushFront`



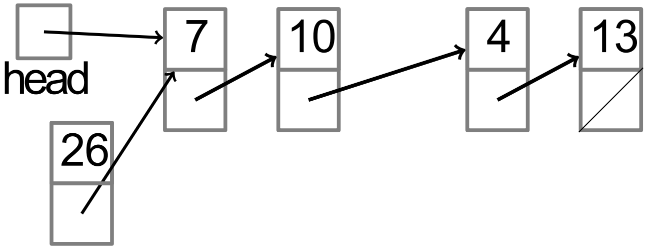
Times for Some Operations

`PushFront`



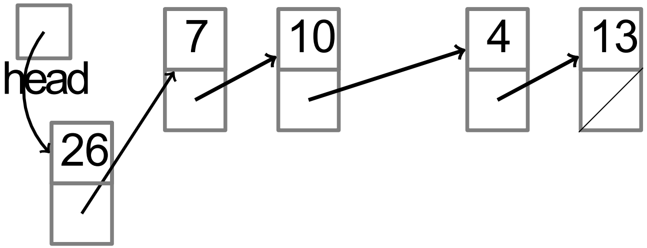
Times for Some Operations

`PushFront`



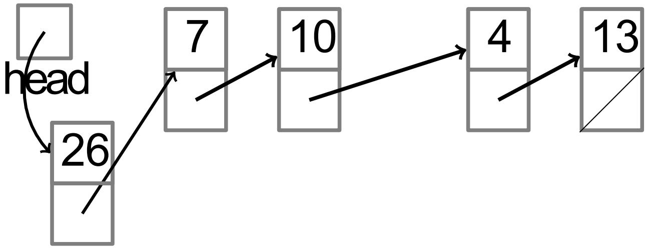
Times for Some Operations

`PushFront` $O(1)$



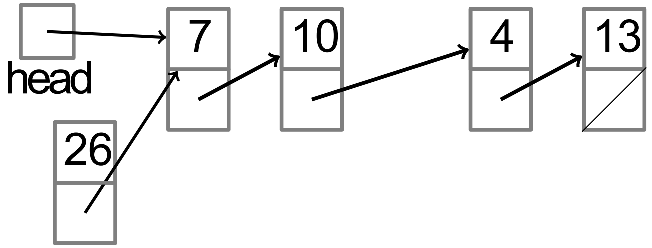
Times for Some Operations

`PopFront`



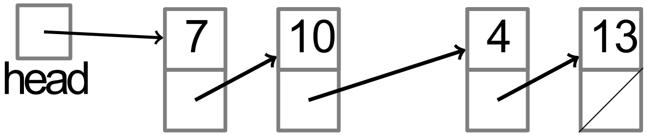
Times for Some Operations

`PopFront`



Times for Some Operations

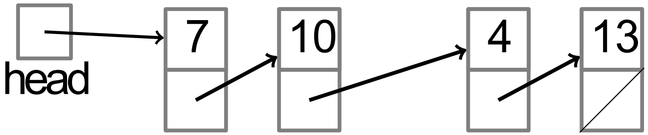
`PopFront` $O(1)$



Times for Some Operations

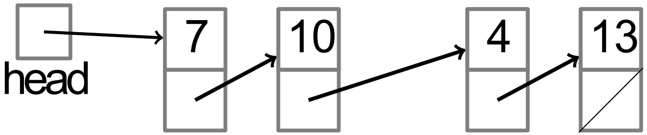
`PushBack`

(no tail)



Times for Some Operations

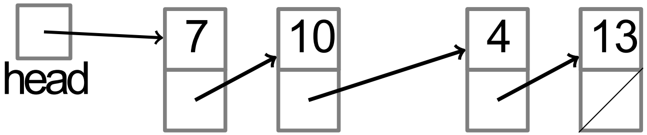
`PushBack` $O(n)$
(no tail)



Times for Some Operations

`PopBack`

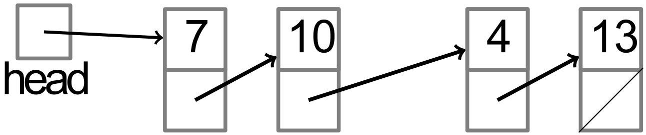
(no tail)



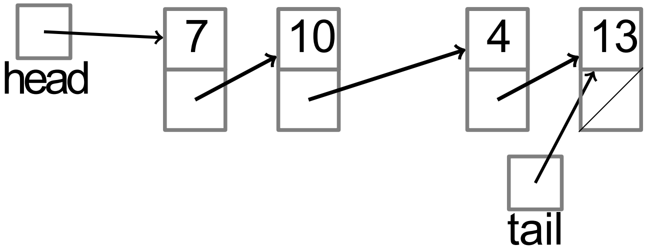
Times for Some Operations

`PopBack` *O*(*n*)

(no tail)



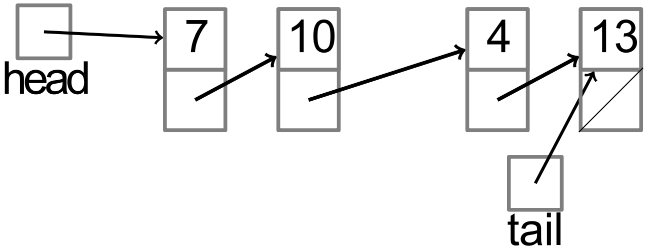
Times for Some Operations



Times for Some Operations

`PushBack`

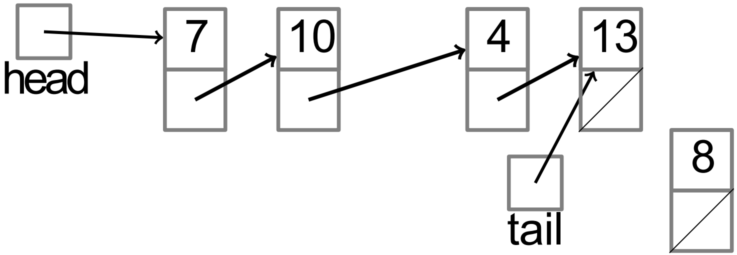
(with tail)



Times for Some Operations

`PushBack`

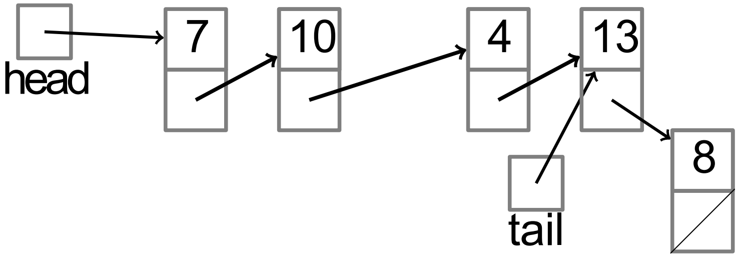
(with tail)



Times for Some Operations

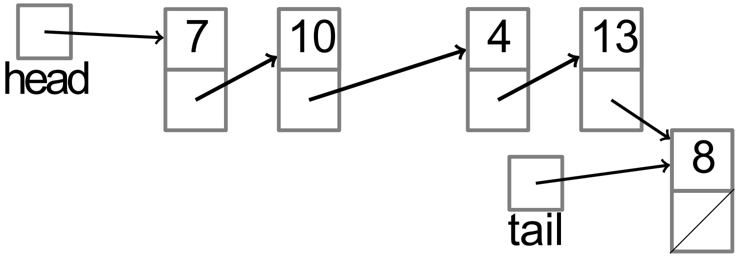
`PushBack`

(with tail)



Times for Some Operations

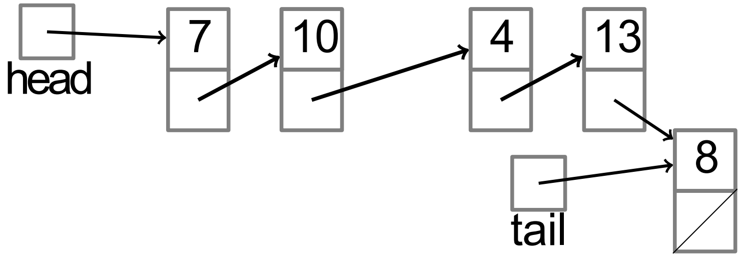
`PushBack` $O(1)$
(with tail)



Times for Some Operations

`PopBack`

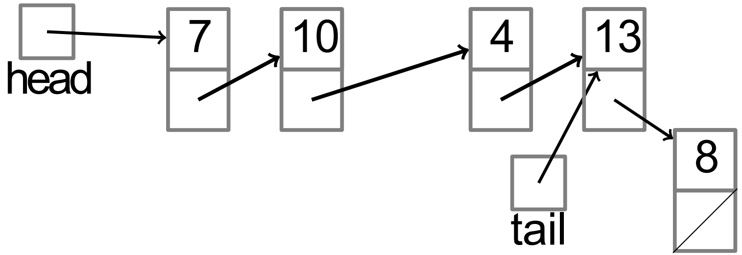
(with tail)



Times for Some Operations

`PopBack`

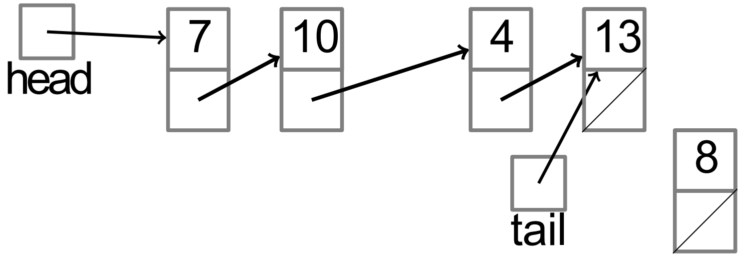
(with tail)



Times for Some Operations

`PopBack`

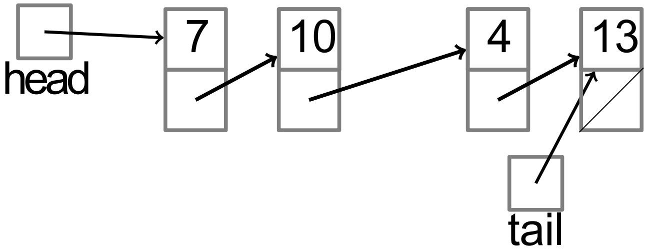
(with tail)



Times for Some Operations

`PopBack` $O(n)$

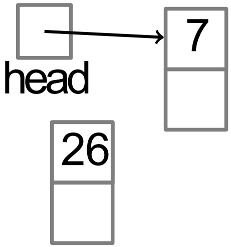
(with tail)



Singly-linked List

PushFront(*key*)

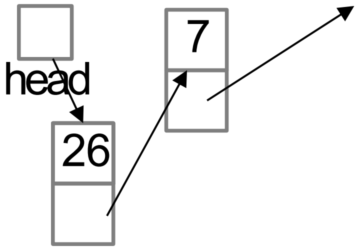
```
node ← new node
node.key ← key
node.next ← head
head ← node
if tail = nil:
    tail ← head
```



Singly-linked List

PushFront(*key*)

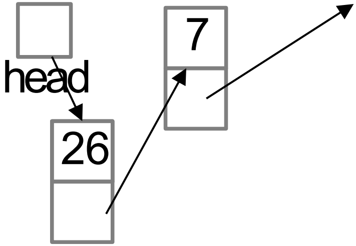
```
node ← new node
node.key ← key
node.next ← head
head ← node
if tail = nil:
    tail ← head
```



Singly-linked List

PopFront()

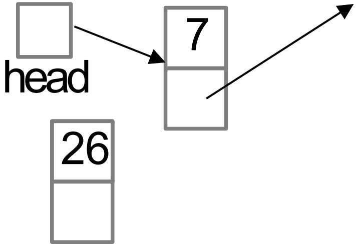
```
if $head$ = nil:
    ERROR: empty list
$head \leftarrow head.next$
if $head$ = nil:
    $tail \leftarrow$ nil
```



Singly-linked List

PopFront()

```
if $head$ = nil:
  ERROR: empty list
$head \leftarrow head.next$
if $head$ = nil:
  $tail \leftarrow$ nil
```



Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$

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()`	$O(n)$	

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)`	$O(n)$	

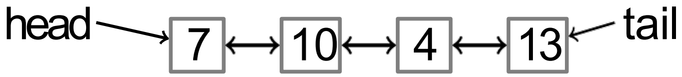
Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	$O(n)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	

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()`	$O(1)$	

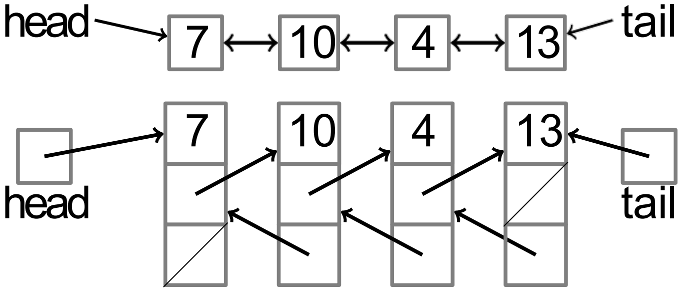
Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	$O(n)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	
`Empty()`	$O(1)$	
`AddBefore(Node, Key)`	$O(n)$	

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)`	$O(1)$	

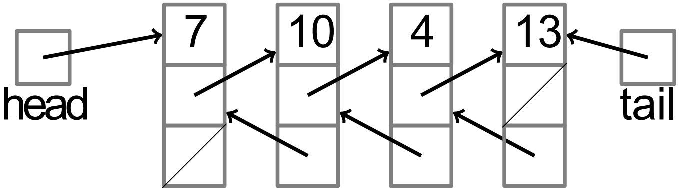
Doubly-Linked List



Doubly-Linked List



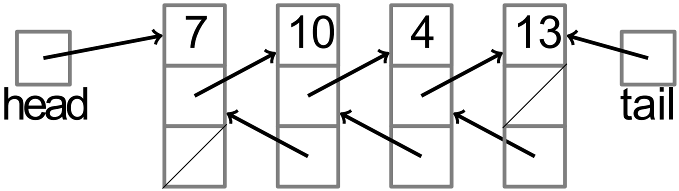
Doubly-Linked List



Node contains:

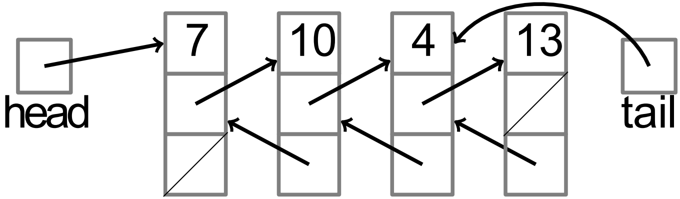
- `key`
- `next` pointer
- `prev` pointer

Doubly-Linked List



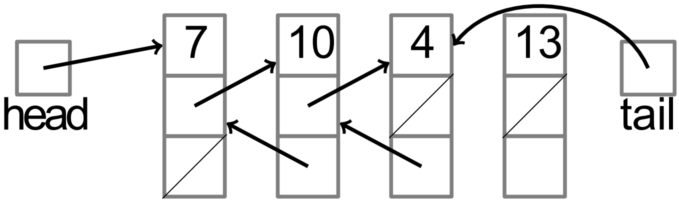
`PopBack`

Doubly-Linked List



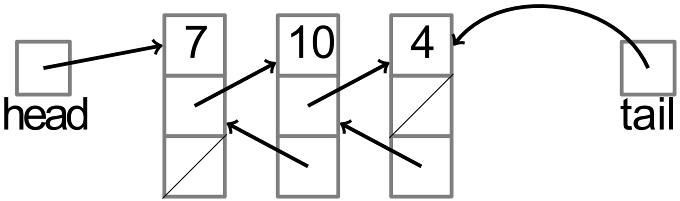
PopBack

Doubly-Linked List



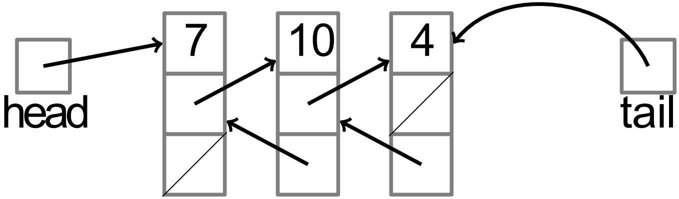
`PopBack`

Doubly-Linked List



`PopBack`

Doubly-Linked List



`PopBack` *O*(1)

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.

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.

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.

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.

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.