

Basic Data Structures: Arrays and Linked Lists

Neil Rhodes

Department of Computer Science and Engineering
University of California, San Diego

Data Structures

<http://bit.ly/algospecialization>

Outline

```
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:

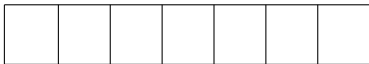
Contiguous area of memory



Definition

Array:

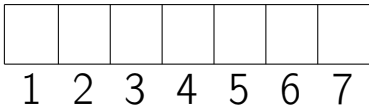
Contiguous area of memory consisting of equal-size elements



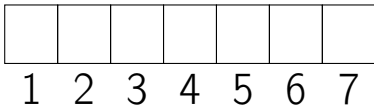
Definition

Array:

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

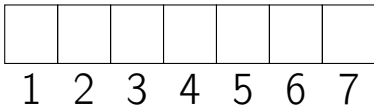


What's Special About Arrays?



What's Special About Arrays?

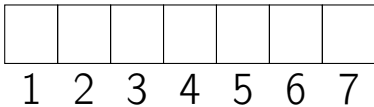
Constant-time access



What's Special About Arrays?

Constant-time access

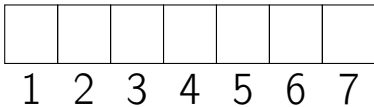
array_addr



What's Special About Arrays?

Constant-time access

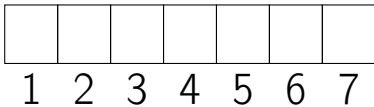
$\text{array_addr} + \text{elem_size} \times (\quad)$



What's Special About Arrays?

Constant-time access

$\text{array_addr} + \text{elem_size} \times (i - \text{first_index})$



Multi-Dimensional Arrays

Multi-Dimensional Arrays

(1, 1)					

Multi-Dimensional Arrays

			(3,4)		

Multi-Dimensional Arrays

			(3,4)		

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

Multi-Dimensional Arrays

			(3,4)		

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

Multi-Dimensional Arrays

			(3,4)		

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

Multi-Dimensional Arrays

			(3,4)		

$$\text{array_addr} + \\ \text{elem_size} \times ((3 - 1) \times 6 + (4 - 1))$$

$(1, 1)$
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(1, 5)$
$(1, 6)$
$(2, 1)$
\vdots

Row-major

$(1, 1)$
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(1, 5)$
$(1, 6)$
$(2, 1)$
\vdots

Row-major

$(1, 1)$
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(1, 5)$
$(1, 6)$
$(2, 1)$
\vdots

$(1, 1)$
$(2, 1)$
$(3, 1)$
$(1, 2)$
$(2, 2)$
$(3, 2)$
$(1, 3)$
\vdots

Row-major

$(1, 1)$
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(1, 5)$
$(1, 6)$
$(2, 1)$
\vdots

Column-major

$(1, 1)$
$(2, 1)$
$(3, 1)$
$(1, 2)$
$(2, 2)$
$(3, 2)$
$(1, 3)$
\vdots

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	$O(1)$	
End		
Middle		

5	8	3	12	4		
---	---	---	----	---	--	--

Times for Common Operations

	Add	Remove
Beginning	$O(1)$	
End		
Middle		

5	8	3	12	4		
---	---	---	----	---	--	--

Times for Common Operations

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

5	8	3	12			
---	---	---	----	--	--	--

Times for Common Operations

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

	8	3	12			
--	---	---	----	--	--	--

Times for Common Operations

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

8		3	12			
---	--	---	----	--	--	--

Times for Common Operations

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

8	3		12			
---	---	--	----	--	--	--

Times for Common Operations

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

8	3	12				
---	---	----	--	--	--	--

Times for Common Operations

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

8	3	12				
---	---	----	--	--	--	--

Times for Common Operations

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

8	3	12				
---	---	----	--	--	--	--

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

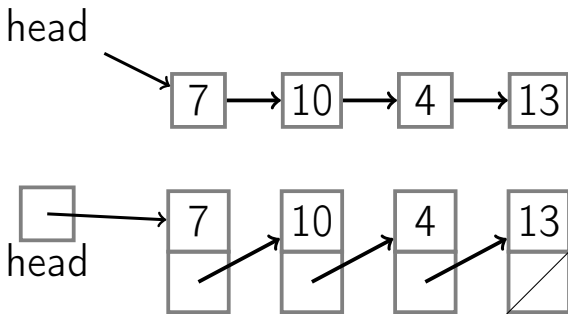
- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.

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?

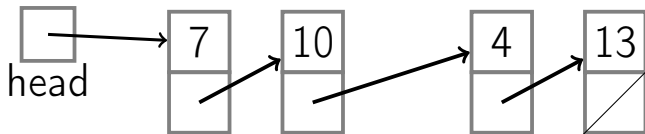
List API

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

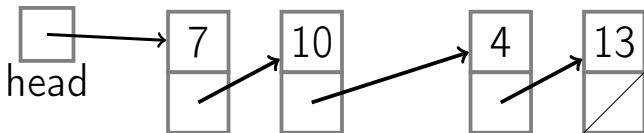
PushFront(Key)	add to front
Key TopFront()	return front item
PopFront()	remove front item
PushBack(Key)	add to back
Key TopBack()	return back item
PopBack()	remove back item
Boolean Find(Key)	is key in list?
Erase(Key)	remove key from list
Boolean Empty()	empty list?
AddBefore(Node, Key)	adds key before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

Times for Some Operations



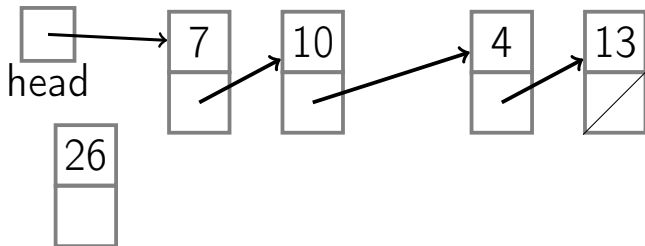
Times for Some Operations

PushFront



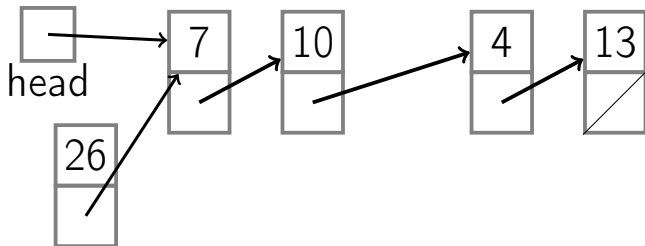
Times for Some Operations

PushFront



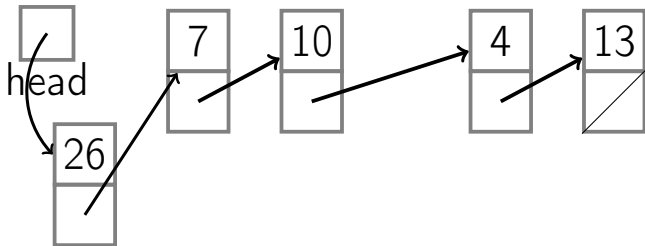
Times for Some Operations

PushFront



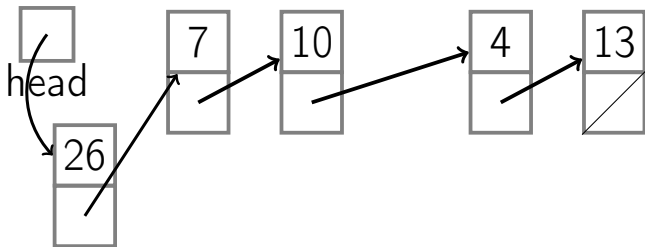
Times for Some Operations

PushFront $O(1)$



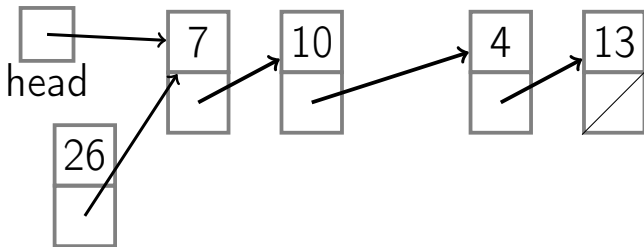
Times for Some Operations

PopFront



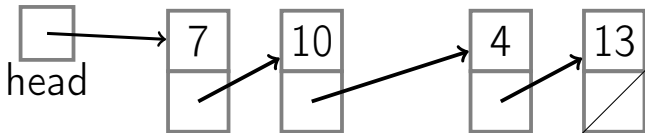
Times for Some Operations

PopFront



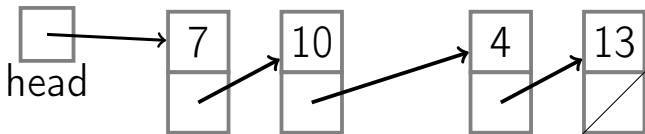
Times for Some Operations

PopFront $O(1)$



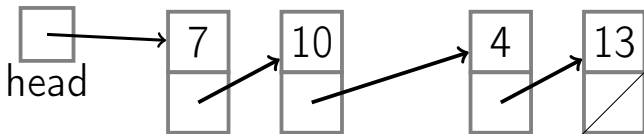
Times for Some Operations

PushBack
(no tail)



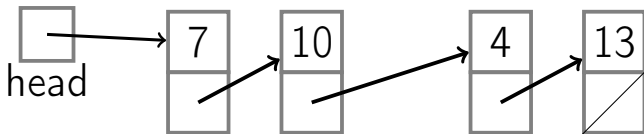
Times for Some Operations

PushBack $O(n)$
(no tail)



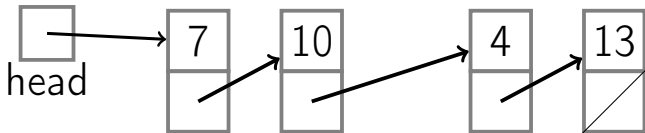
Times for Some Operations

PopBack
(no tail)

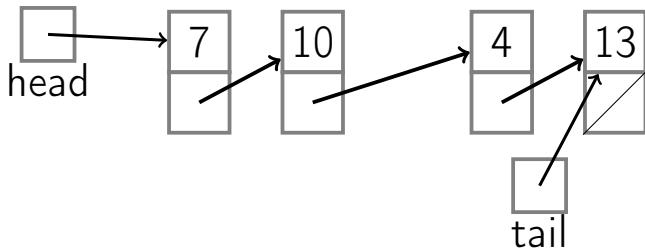


Times for Some Operations

PopBack $O(n)$
(no tail)

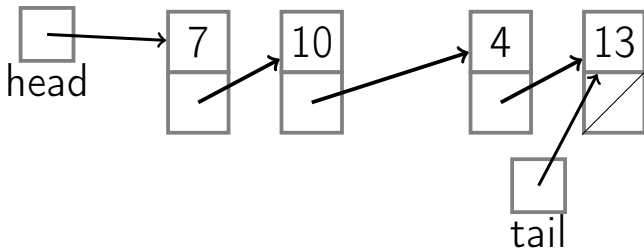


Times for Some Operations



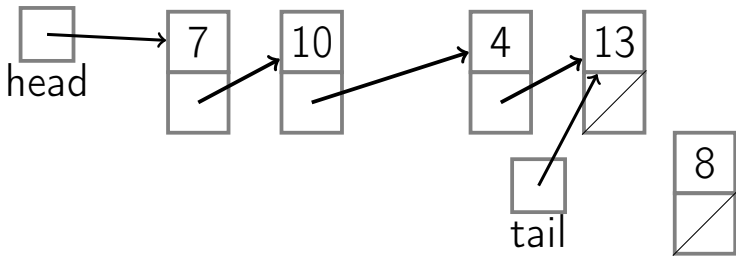
Times for Some Operations

PushBack
(with tail)



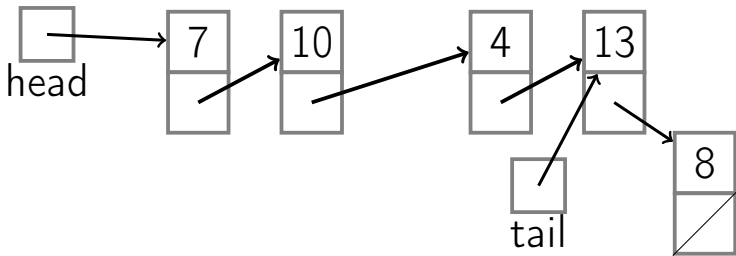
Times for Some Operations

PushBack
(with tail)



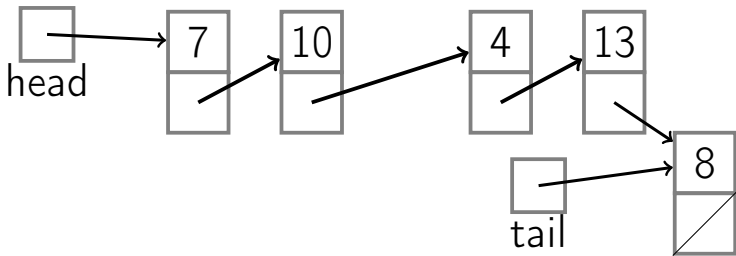
Times for Some Operations

PushBack
(with tail)



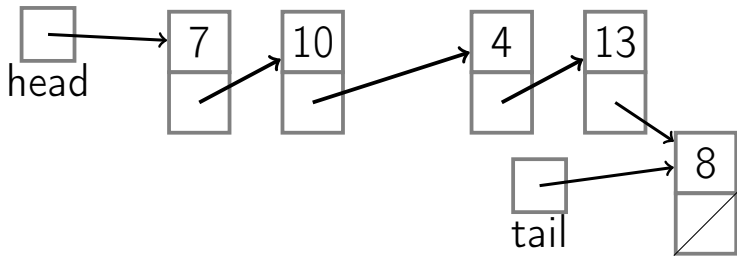
Times for Some Operations

PushBack $O(1)$
(with tail)



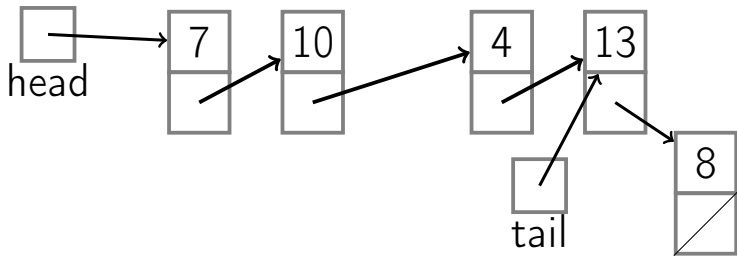
Times for Some Operations

PopBack
(with tail)



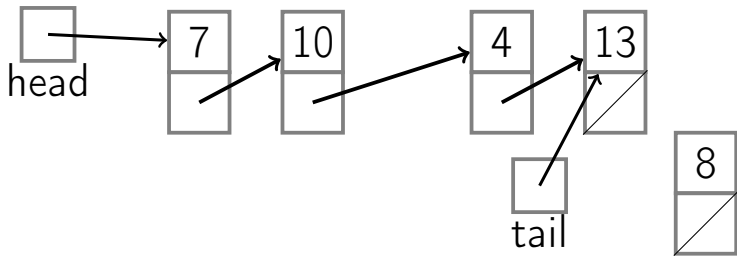
Times for Some Operations

PopBack
(with tail)



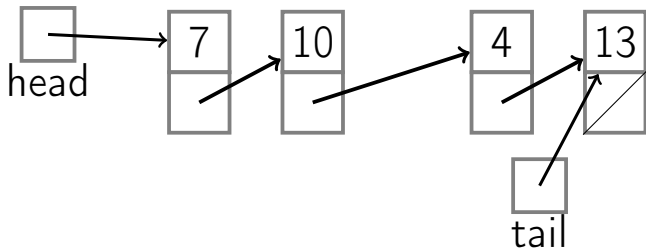
Times for Some Operations

PopBack
(with tail)



Times for Some Operations

PopBack $O(n)$
(with tail)



Singly-linked List

PushFront(*key*)

node \leftarrow new node

node.key \leftarrow *key*

node.next \leftarrow *head*

head \leftarrow *node*

if *tail* = nil:

tail \leftarrow *head*

Singly-linked List

PopFront()

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

Singly-linked List

PushBack(*key*)

node \leftarrow new node

node.key \leftarrow *key*

node.next = nil

Singly-linked List

PushBack(*key*)

node \leftarrow new node

node.key \leftarrow *key*

node.next = nil

if *tail* = nil:

head \leftarrow *tail* \leftarrow *node*

Singly-linked List

PushBack(*key*)

node \leftarrow new node

node.key \leftarrow *key*

node.next = nil

if *tail* = nil:

head \leftarrow *tail* \leftarrow *node*

else:

tail.next \leftarrow *node*

tail \leftarrow *node*

Singly-linked List

PopBack()

Singly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
```

Singly-linked List

PopBack()

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


Singly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
if head = tail:
    head  $\leftarrow$  tail  $\leftarrow$  nil
else:
    p  $\leftarrow$  head
    while p.next.next  $\neq$  nil:
        p  $\leftarrow$  p.next
```

Singly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
if head = tail:
    head  $\leftarrow$  tail  $\leftarrow$  nil
else:
    p  $\leftarrow$  head
    while p.next.next  $\neq$  nil:
        p  $\leftarrow$  p.next
    p.next  $\leftarrow$  nil; tail  $\leftarrow$  p
```

Singly-linked List

AddAfter(*node*, *key*)

node2 \leftarrow new node

node2.key \leftarrow *key*

node2.next = *node.next*

node.next = *node2*

if *tail* = *node*:

tail \leftarrow *node2*

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)	$O(n)$	

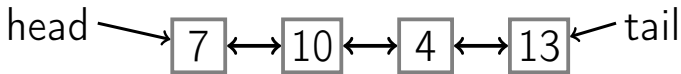
Singly-Linked List	no tail	with tail
PushFront(Key)	$O(1)$	
TopFront()	$O(1)$	
PopFront()	$O(1)$	
PushBack(Key)	$O(n)$	$O(1)$
TopBack()	$O(n)$	$O(1)$
PopBack()	$O(n)$	
Find(Key)	$O(n)$	
Erase(Key)	$O(n)$	

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()	$O(1)$	

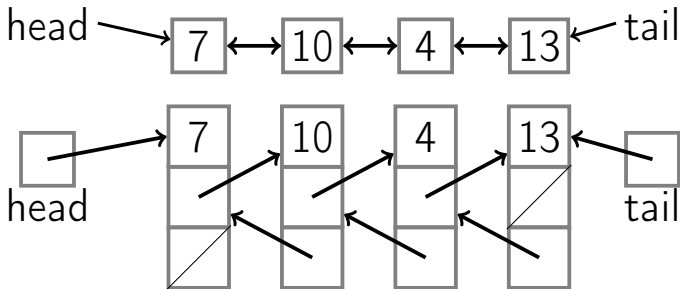
Singly-Linked List	no tail	with tail
PushFront(Key)	$O(1)$	
TopFront()	$O(1)$	
PopFront()	$O(1)$	
PushBack(Key)	$O(n)$	$O(1)$
TopBack()	$O(n)$	$O(1)$
PopBack()	$O(n)$	
Find(Key)	$O(n)$	
Erase(Key)	$O(n)$	
Empty()	$O(1)$	
AddBefore(Node, Key)	$O(n)$	

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)	$O(1)$	

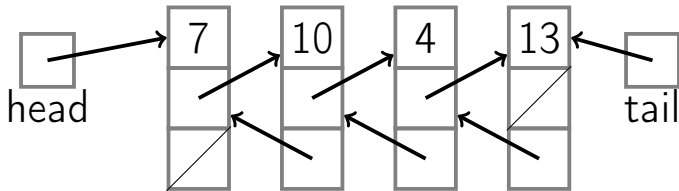
Doubly-Linked List



Doubly-Linked List



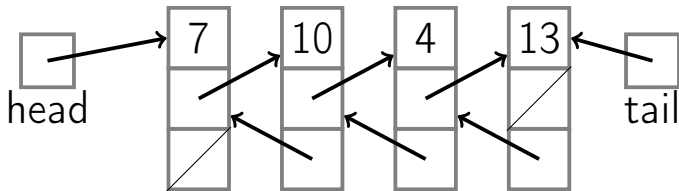
Doubly-Linked List



Node contains:

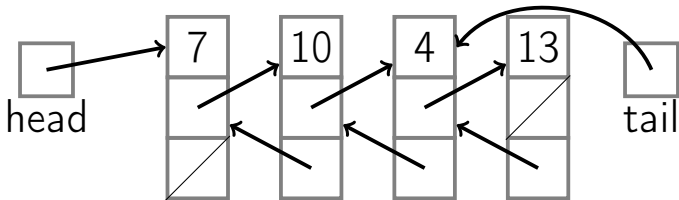
- key
- next pointer
- prev pointer

Doubly-Linked List



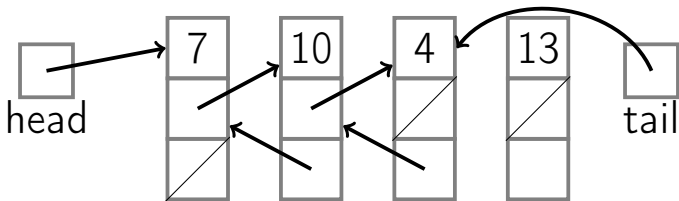
PopBack

Doubly-Linked List



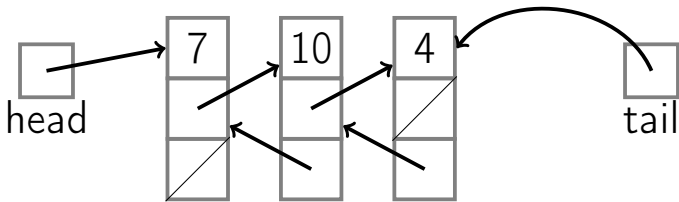
PopBack

Doubly-Linked List



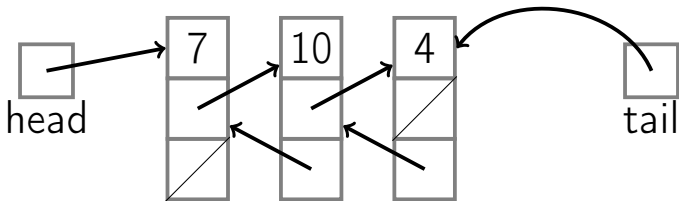
PopBack

Doubly-Linked List



PopBack

Doubly-Linked List



PopBack $O(1)$

Doubly-linked List

PushBack(*key*)

node \leftarrow new node

node.key \leftarrow *key*; *node.next* = nil

Doubly-linked List

PushBack(*key*)

node \leftarrow new node

node.key \leftarrow *key*; *node.next* = nil

if *tail* = nil:

head \leftarrow *tail* \leftarrow *node*

node.prev \leftarrow nil

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

PopBack()

Doubly-linked List

PopBack()

```
if head = nil:  ERROR: empty list
```

Doubly-linked List

PopBack()

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

Doubly-linked List

PopBack()

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

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.prev ← node2  
if node2.prev ≠ 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.

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.