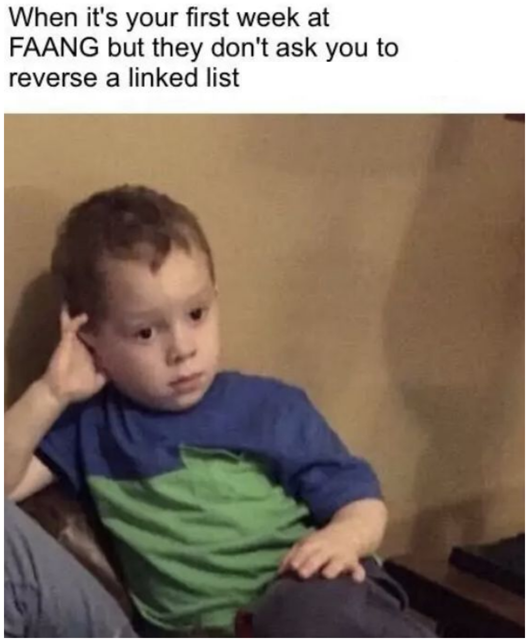


CS 1332R
WEEK 2

ArrayList

Singly LinkedList

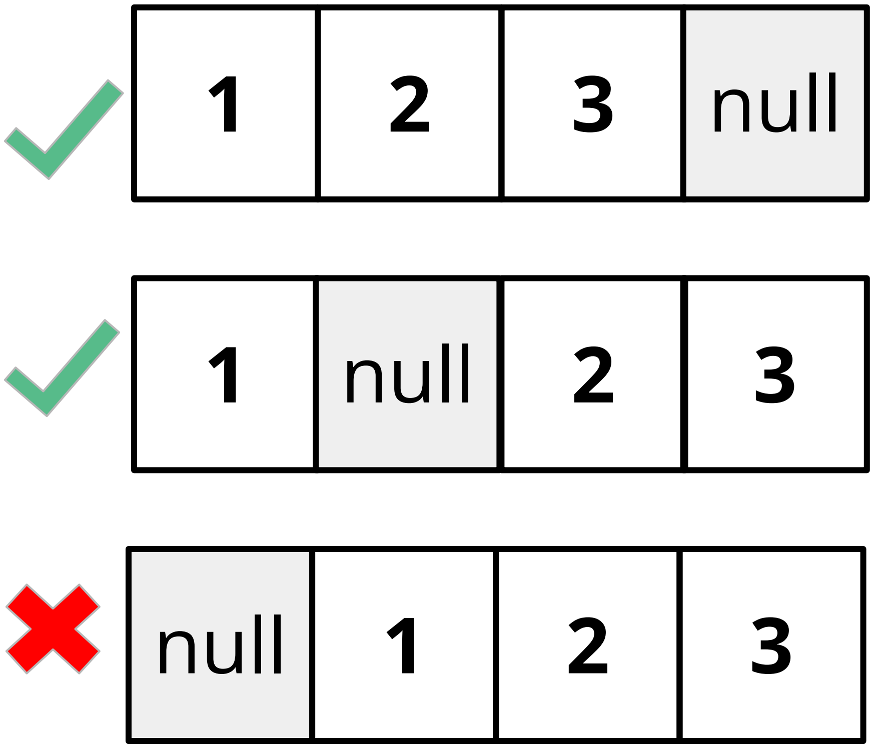
Doubly LinkedList

ANNOUNCEMENTS

- ❑

Array

- Contiguous blocks of memory
- Zero-aligned
- Can have null spots in between indices
- Fixed size
- O(1) add, remove, and get operations



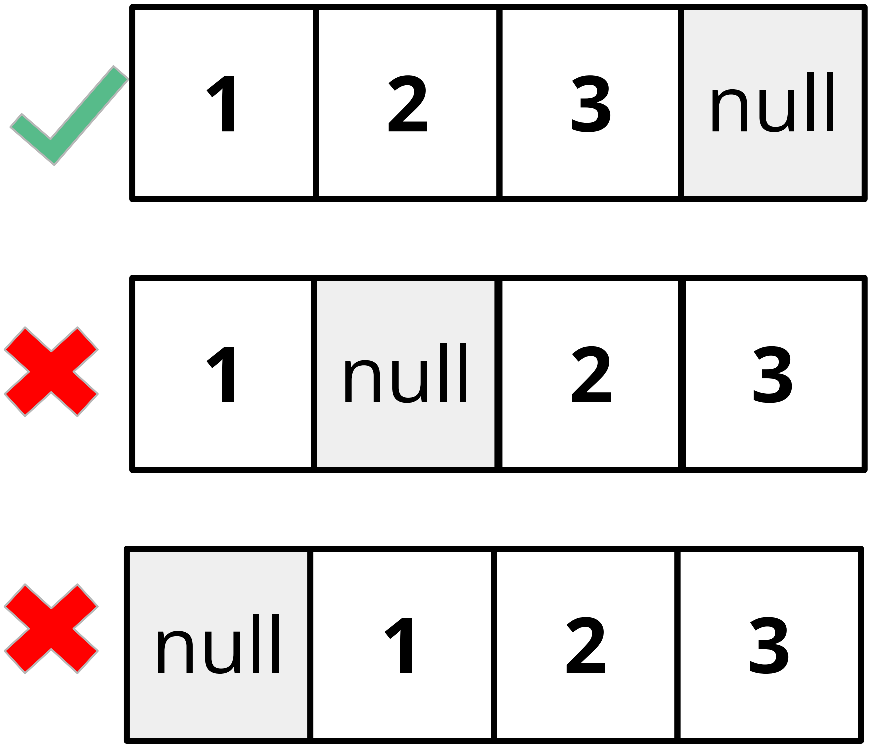
List ADT

- A List is as an ordered, linear, iterable structure of elements.

```
void add(int index, T data)

T get(int index)

boolean contains(T data)

T remove(int index)

int size()
```

ArrayList

- Data structure ***backed by an array*** that implements the List ADT
- Zero-aligned
- Contiguous data
- Can be resized, but the user is unaware of this



ArrayList: Add

0	1	2	3	4	5	6	7	8	9	10
13	**16**	**4**	**7**	**0**	null	null	null	null	null	null

```
backingArray.length = 11
size = 5
```

➔ When we add to a list, there is a specific index we want to insert the new data at.

What are the values of the valid indices we could add at?

0 to size

ArrayList: Add

- When we add to an list, there is a specific index we want to insert the new data at.
    - add to front = add at index **0**
    - add to back = add at index **size**
    - add at index / add to middle = add at some index in **(0, size)**

All of these cases can be implemented in the same way...

```
void add(int index, T data)
```

1. All data at and to the right of `index` is shifted one cell to the right.
2. The new `data` is placed at `index`.
3. Increment `size`.

Would we shift the data starting at `size` or at `index`?

`size` - If we start at `index`, we will override data we need.

ArrayList: Add

`void add(int index, T data)`

1. All data <u>at and to the right</u> of `index` is shifted one cell to the right.
2. The new `data` is placed at `index`.
3. Increment `size`.

EXAMPLE: `add(0, 2)`

```
backingArray.length = 11
size = 5
```

0	1	2	3	4	5	6	7	8	9	10
13	**16**	**4**	**7**	**0**	null	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9	10
13	**13**	**16**	**4**	**7**	**0**	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9	10
2	**13**	**16**	**4**	**7**	**0**	null	null	null	null	null

```
backingArray.length = 11
size = 6
```

ArrayList: Add w/ Resize

0	1	2	3	4	5	6	7	8	9	10
13	**16**	**4**	**7**	**0**	**3**	**4**	**5**	**2**	**11**	**9**

```
backingArray.length = 11
size = 11
```

➔ Remember, an array has a fixed capacity. When the array becomes full, there is no space to add another piece of data. We must resize the backing array.

What condition do we check for to see if we have to resize?

backingArray.length == size

ArrayList: Add w/ Resize

EXAMPLE: `add(2, 2)`

`void add(int index, T data)`

1. Copy all data to the left of `index` into the new array as is.
2. Copy all data at and to the right of `index` one cell to the right in the new array.
3. The new `data` is placed at `index` in the new array.
4. Reassign `backingArray` to the new array.
5. Increment `size`.

0	1	2	3	4
13	16	4	7	0

`backingArray.length = 5`
`size = 5`

0	1	2	3	4	5	6	7	8	9
null	null	null	null	null	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9
13	16	null	null	null	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9
13	16	null	4	7	0	null	null	null	null

0	1	2	3	4	5	6	7	8	9
13	16	2	4	7	0	null	null	null	null

`backingArray.length = 10`
`size = 6`

ArrayList: Remove

0	1	2	3	4	5	6	7	8	9	10
13	**16**	**4**	**7**	**0**	null	null	null	null	null	null

```
backingArray.length = 11
size = 5
```

➔ When we remove from an list, there is a specific index we want to remove data from.

What are the values of the valid indices we could remove from?

0 to size - 1

ArrayList: Remove

`void remove(int index)`

1. Save the data at `index`.
2. All data to the right of `index` is shifted one cell to the left.
3. Set the backing array at `size - 1` to null.
4. Decrement `size`.

No resize case when removing.

EXAMPLE: `remove(0)`

```
backingArray.length = 11
size = 5
```

0	1	2	3	4	5	6	7	8	9	10
13	**16**	**4**	**7**	**0**	null	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9	10
16	**4**	**7**	**0**	**0**	null	null	null	null	null	null

0	1	2	3	4	5	6	7	8	9	10
16	**4**	**7**	**0**	null	null	null	null	null	null	null

```
backingArray.length = 11
size = 4
```

ArrayList: Practice

Perform the following operations on an empty ArrayList of initial capacity 7.

Add 1 to index 0
Add 2 to index 0
Add 3 to index 1
Add 4 to the front
Add 5 to the back
Remove from the front
Remove from the back
Remove from index 1

Add 1 to index 0:	1						
Add 2 to index 0:	2	1					
Add 3 to index 1:	2	3	1				
Add 4 to the front:	4	2	3	1			
Add 5 to the back:	4	2	3	1	5		
Remove from the front:	2	3	1	5			
Remove from the back:	2	3	1				
Remove from index 1:	2	1					

ArrayList: Efficiencies

➔ The time complexity comes from ***shifting***.

	Front	Index	Back
Adding	O(n)	O(n)	O(1)*
Removing	O(n)	O(n)	O(1)
Accessing	O(1)	O(1)	O(1)

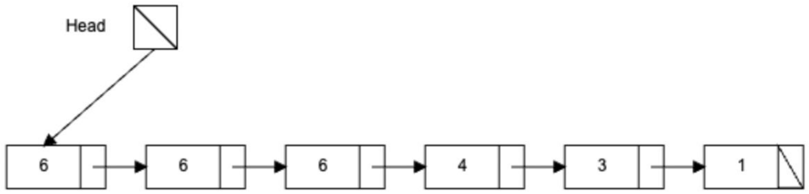
Why do we have an amortized analysis of adding to the back?

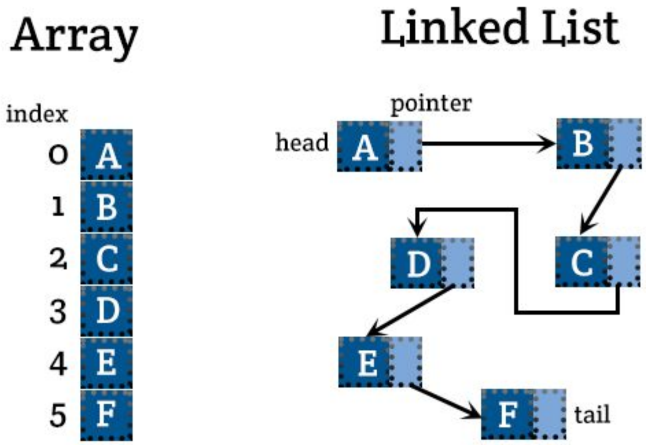
There is no shifting when we have to add to the back. This operation is only O(n) in the resize case.

Primary benefit of ArrayLists: O(1) access

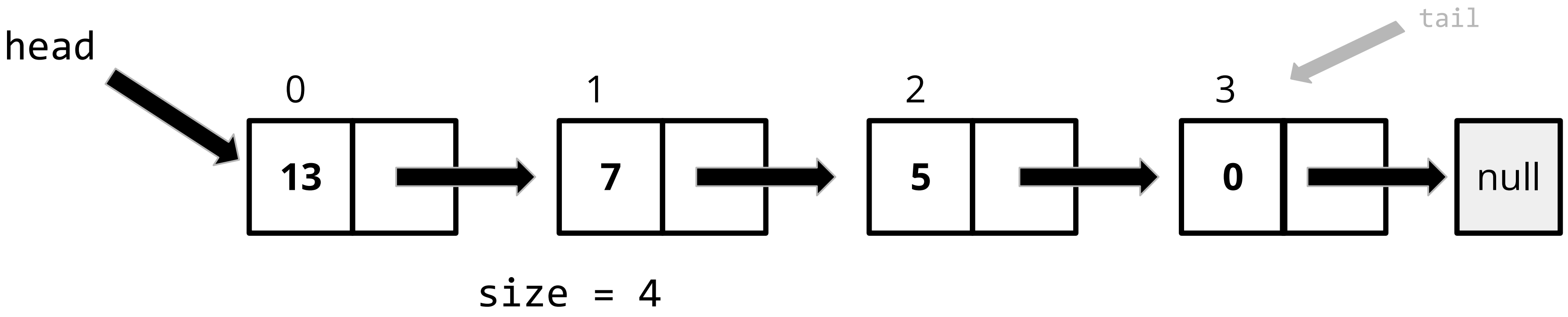
Singly LinkedList

- Data structure ***backed by linked nodes*** that implements the List ADT
- A Singly LinkedList node contains:
  - data
  - a pointer to the next node
- Must have a head pointer, maybe a tail pointer
- Data is no longer contiguous in memory, *but to the user it still is.*





Singly LinkedList: Access




```
Node curr = head;
while (curr != null) {
    // Do things here
    curr = curr.next;
}
```

What should the bounds of for loop be to access a specific index?

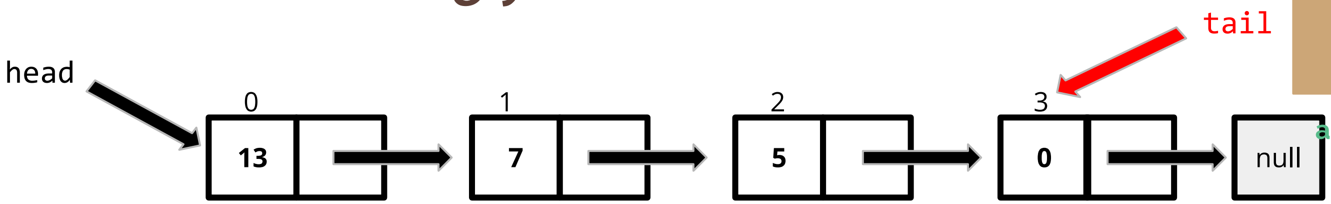
[0, index) - stop value before the index you would like to access

- *O(1) access to the first element*
- *If there is a tail, O(1) access to the last element as well*

```
void add(int index, T data)
```

Singly LinkedList: Add

Could this change any of our procedures?



addToBack becomes O(1)

`size = 4`

ADD TO FRONT

1. Create a new node containing **`data`**.
2. Set the new node's next pointer to **`head`**.
3. Reassign **`head`** to the new node.
4. Increment size.

ADD TO INDEX

1. Create a new node containing `data`.
2. Iterate to the node <u>before</u> `index`.
3. Set the new node's next pointer to the current node's next pointer.
4. Set the current node's next pointer to the new node.
5. Increment size.

ADD TO BACK

Same procedure as add to index where

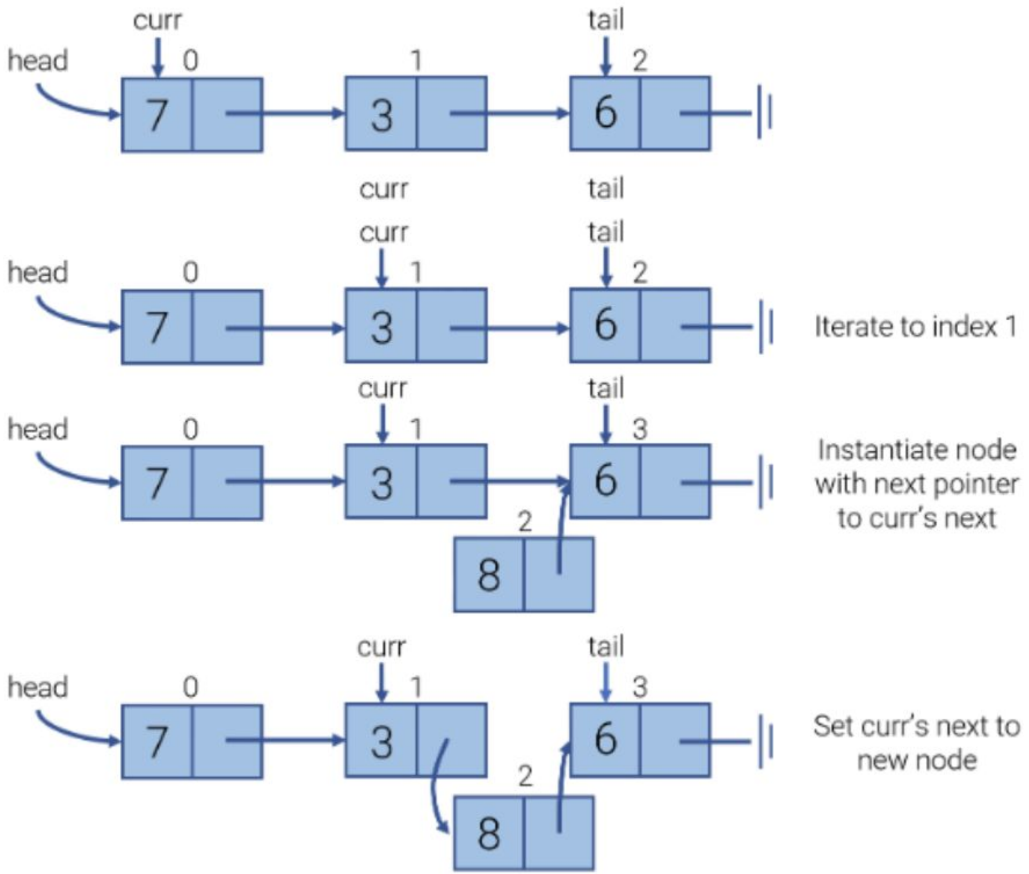
`index == size`

<u>W/ TAIL</u>

1. Create a new node containing `data`.
2. Set `tail`'s next pointer to the new node.
3. Reassign `tail` to the new node.
4. Increment size.

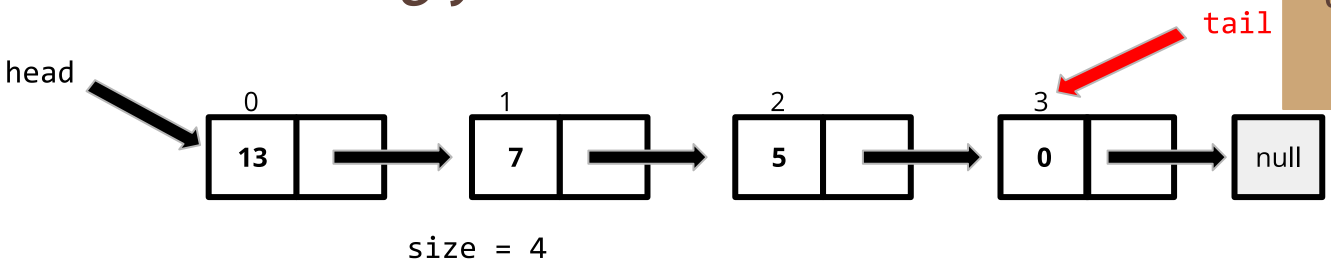
Singly LinkedList: Add

`add(2, 8)`



`void remove(int index)`

Singly LinkedList: Remove



Could this change any of our procedures?

NO, in order to remove we need the node *prior* to the removed node.

REMOVE FROM FRONT

1. Save the data in **`head`**.
2. Reassign **`head`** to **`head.next`**.
3. Decrement size.

REMOVE FROM INDEX

1. Iterate to the node <u>before</u> **`index`**. Save the data in **`curr.next`**.
2. Set the current node's next pointer to **`curr.next.next`**.
3. Decrement size.

REMOVE FROM BACK

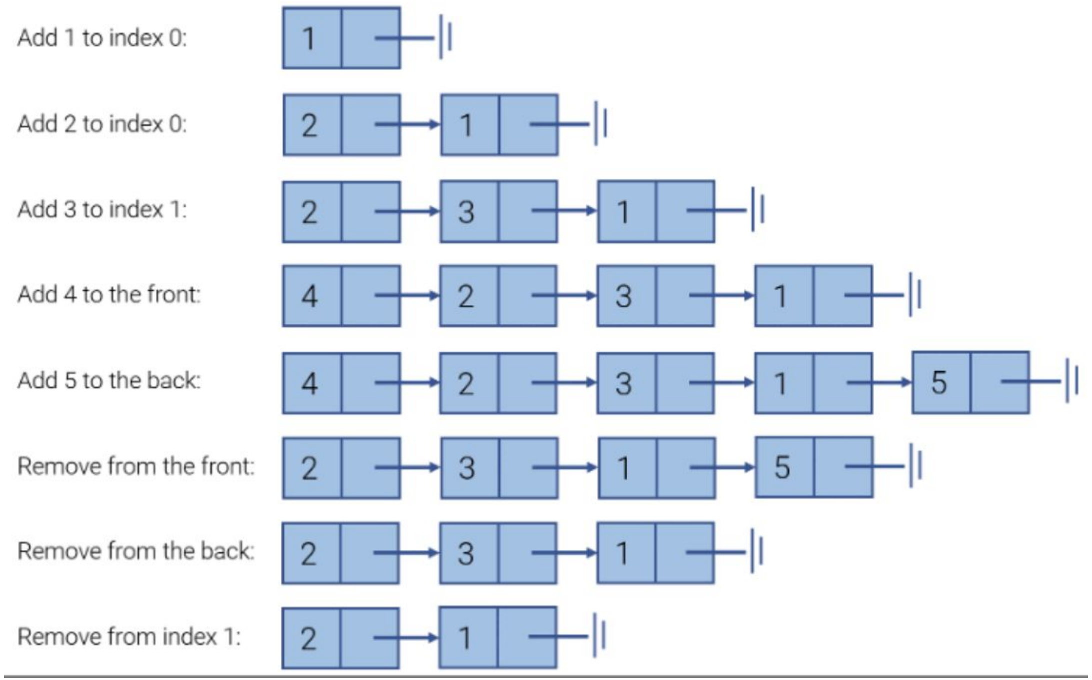
Same procedure as add to index where

`index == size`

Singly LinkedList: Practice

Perform the following operations on an empty SinglyLinkedList.

Add 1 to index 0
Add 2 to index 0
Add 3 to index 1
Add 4 to the front
Add 5 to the back
Remove from the front
Remove from the back
Remove from index 1



Singly LinkedList: Efficiencies

➔ The time complexity comes from ***iterating/accessing.***

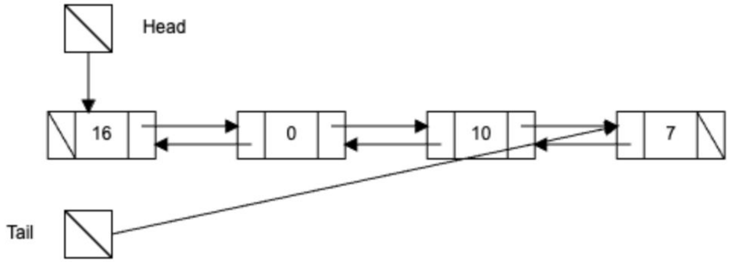
		Front	Middle	Back
With tail	Adding	$O(1)$	$O(n)$	$O(1)$
	Removing	$O(1)$	$O(n)$	$O(n)$
	Accessing	$O(1)$	$O(n)$	$O(1)$
Without tail	Adding	$O(1)$	$O(n)$	$O(n)$
	Removing	$O(1)$	$O(n)$	$O(n)$
	Accessing	$O(1)$	$O(n)$	$O(n)$

What were the time complexity benefits from adding the tail?

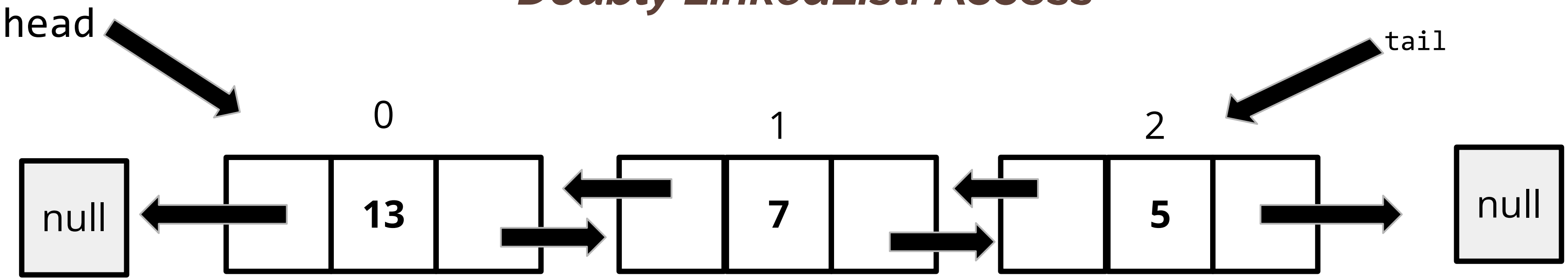
access back, add to back

Doubly LinkedList

- Data structure ***backed by linked nodes*** that implements the List ADT
- A doubly linked list node contains:
  - data
  - a pointer to the next node
  - a pointer to the previous node
- Head pointer and almost always a tail pointer (Java's LinkedList is a doubly linked list with a tail pointer)



Doubly LinkedList: Access




```
size = 3
```

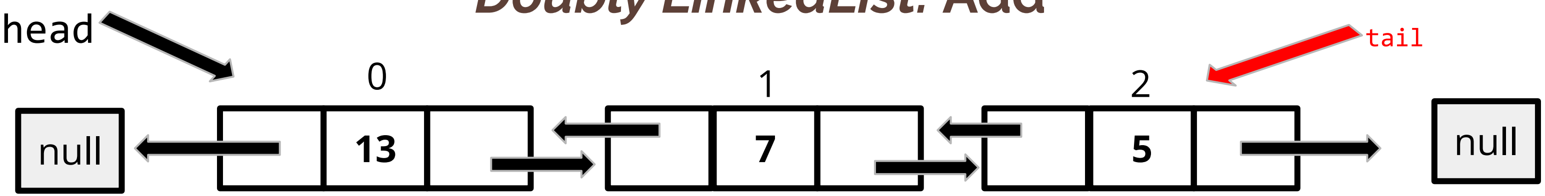
Iterate from the side closest to the index you are looking for.

```
Node curr = head;
while (curr != null) {
    // Do things here
    curr = curr.next;
}
```

```
Node curr = tail;
while (curr != null) {
    // Do things here
    curr = curr.prev;
}
```

`void add(int index, T data)`

Doubly LinkedList: Add



`size = 3`

ADD TO FRONT

1. Create a new node containing **`data`**.
2. Set the new node's next pointer to **`head`**.
3. Set the correct pointers ***to*** the new node. Account for **`head`** and **`tail`** appropriately.
4. Increment size.

ADD TO INDEX

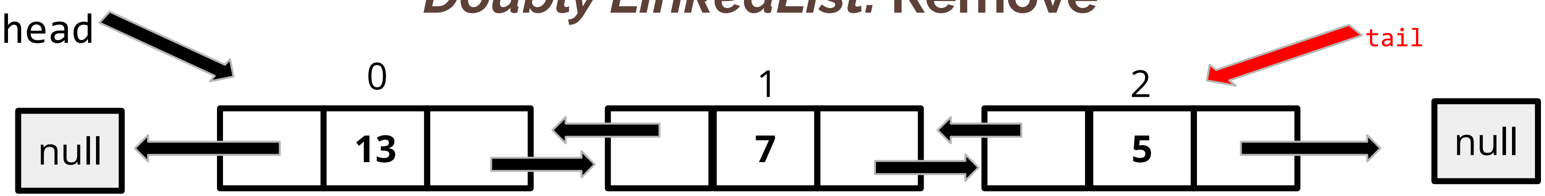
1. Create a new node containing **`data`**.
2. Iterate to the node before **`index`**.
3. Set the new node's next pointer to the current node's next pointer.
4. Set the new node's previous pointer to the current node.
5. Set the correct pointers ***to*** the new node.
6. Increment size.

ADD TO BACK: O(1)

1. Create a new node containing **`data`**.
2. Set the new node's previous pointer to **`tail`**.
3. Set the correct pointers ***to*** the new node. Account for **`head`** and **`tail`** appropriately.
4. Increment size.

`void remove(int index)`

Doubly LinkedList: Remove



`size = 3`

REMOVE FROM FRONT

1. Save the data in **`head`**.
2. Reassign **`head`** to **`head.next`**.
3. Set **`head.prev`** to null.
4. Decrement size.

REMOVE FROM INDEX

1. Iterate to the node before **`index`**. Save the data in **`curr.next`**.
2. Reassign the relevant next and prev pointers.
3. Decrement size.

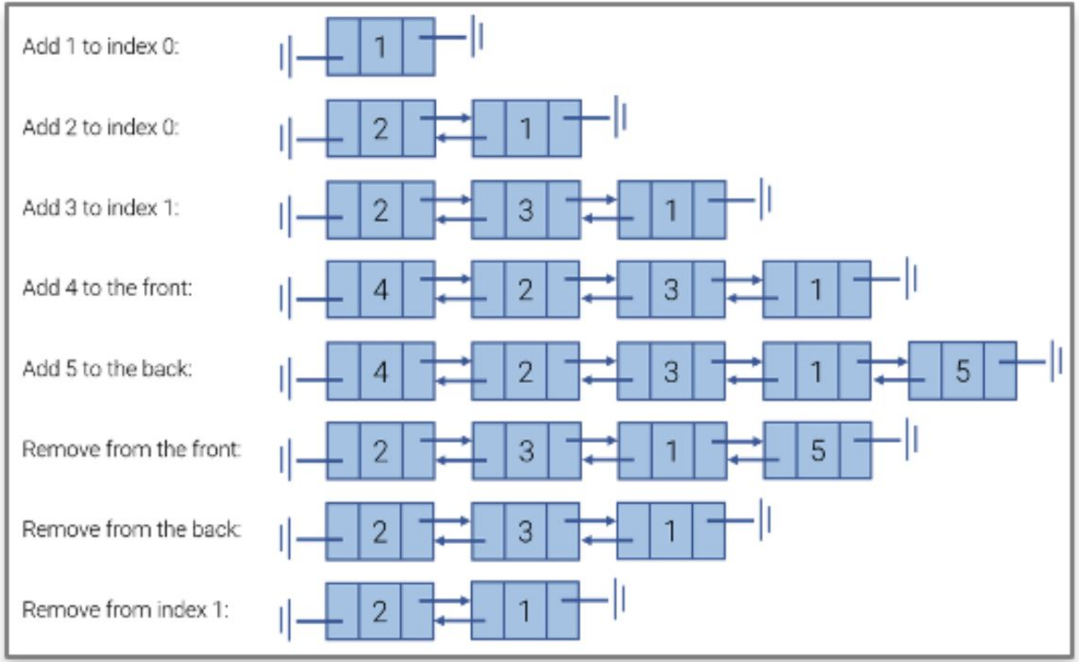
REMOVE FROM BACK: O(1)

1. Save the data in **`tail`**.
2. Reassign **`tail`** to **`tail.prev`**.
3. Set **`tail.next`** to null.
4. Decrement size.

Doubly LinkedList: Practice

Perform the following operations on an empty Doubly LinkedList.

Add 1 to index 0
Add 2 to index 0
Add 3 to index 1
Add 4 to the front
Add 5 to the back
Remove from the front
Remove from the back
Remove from index 1



Doubly LinkedList: Efficiencies

- ➔ The time complexity comes from ***iterating/accessing***.
- ➔ There is slightly higher space usage from the extra pointers.

		Front	Middle	Back
With tail	Adding	$O(1)$	$O(n)$	$O(1)$
	Removing	$O(1)$	$O(n)$	$\mathbf{O(1)}$
	Accessing	$O(1)$	$O(n)$	$O(1)$
Without tail	Adding	$O(1)$	$O(n)$	$O(n)$
	Removing	$O(1)$	$O(n)$	$O(n)$
	Accessing	$O(1)$	$O(n)$	$O(n)$

ArrayLists

- Backed by an array, contiguous in memory
- Resize case → some amortized time complexities
- O(1) access
- O(n) front operations
- O(1) back operations

Both are implementations of the List ADT = ***same public methods***.

LinkedLists

- Backed by nodes that point to each other, scattered in memory
- Usually O(n) access
- O(1) front operations
- O(n) back operations *except in a DLL w/tail*
- Requires extra memory to store pointers

LEETCODE PROBLEMS

[234. Palindrome Linked List]

[206. Reverse Linked List]

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!