CS 1332R
WEEK 3

Circular Singly LinkedList

Stacks

Queues

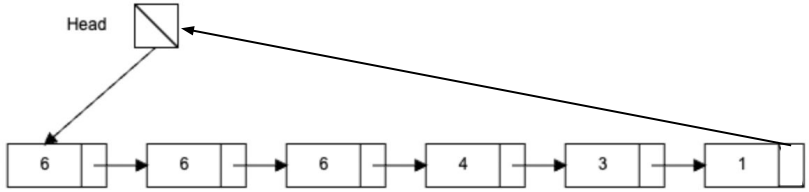
Deques

ANNOUNCEMENTS

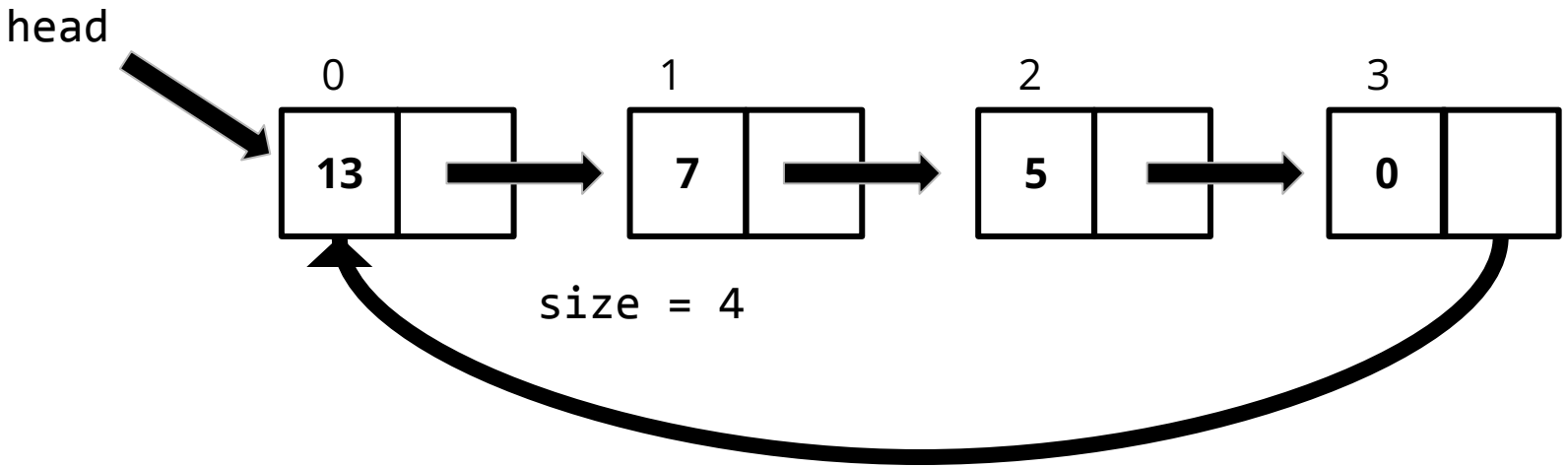
-

Circular Singly LinkedList

- Data structure ***backed by linked nodes*** that implements the List ADT
- A CSLL node contains:
  - data
  - a pointer to the next node
- Head pointer, NEVER a tail pointer (there's no point)
- ***Final node points back to the head node***

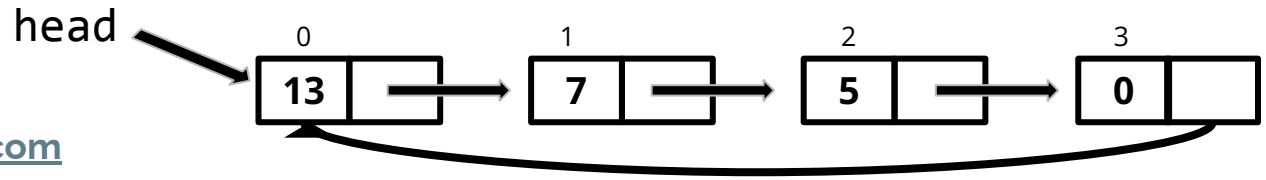


Circular Singly LinkedList: Access




```
Node curr = head;
//handle curr = head case
curr = curr.next;
while (curr != head) {
    // do something
    curr = curr.next;
}
```

Circular Singly LinkedList: Add



csvistool.com

`size = 4`

ADD TO FRONT

1. Create a new node containing `head.data` and pointing to `head.next`.
2. Set the `head`'s next pointer to the new node.
3. Replace `head.data` with `data`.
4. Increment size.

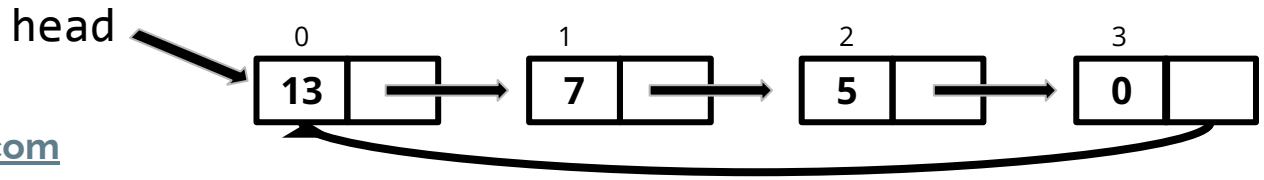
ADD TO INDEX

1. Iterate to the node at `index`.
2. Create a new node containing `curr.data`.
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. Replace `curr.data` with `data`.
6. Increment size.

ADD TO BACK

Same procedure as add to index where `index == size`

Circular Singly LinkedList: Remove



csvistool.com

`size = 4`

REMOVE FROM FRONT

1. Replace `head.data` with `head.next.data`.
2. Set the `head`'s next pointer to `head.next.next`.
3. Decrement size.

REMOVE FROM INDEX

1. Iterate to the node at `index`.
2. Replace `curr.data` with `curr.next.data`.
3. Set `curr`'s next pointer to `curr.next.next`.
4. Decrement size.

REMOVE FROM BACK

Same procedure as remove from index where `index == size - 1`

CircularSinglyLinkedList: 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	$O(1)$	$O(n)$	$O(n)$

****Exact same as a singly linked list.****
We never use a tail with a CSLL in this course.

Stack ADT

- Unordered, not iterable
- Usually backed by an array or an SLL w/o tail - we only need operations at one end of the structure (front or back)
- LIFO = Last In First Out

```
void push(T data)

T pop()

T peek()
```

What are use cases of a stack?

stack of paper, elevator, recursion

Why do none of the methods have an index parameter?

The user does not have a choice of where they add/remove/access from.

Stack Implementations

	Array	SLL w/o tail	DLL w/ tail
Push	Back	Front	Either
Pop	Back	Front	Same as push

Why do we have to choose the back end for the array and the front end for the SLL w/o tail?

Guarantees O(1) pop and push

NOTE: Array-backed stack push is O(1)* due to the resize case.

Queue ADT

- Unordered, not iterable
- Usually backed by a circular array or an SLL w/ tail
- FIFO = First In First Out
- NOTE: Java uses different method names (add, poll/remove)

```
void enqueue(T data)

T dequeue()

T peek()
```

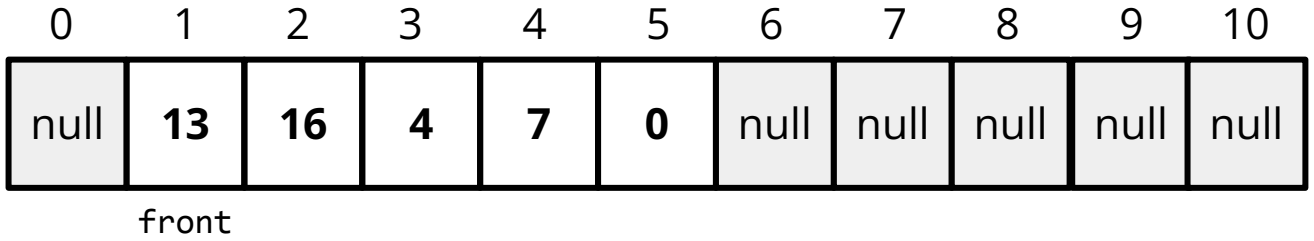
What are use cases of a queue?

waiting in a line, escalator, printer jobs

Why do we have to have a tail in the SLL now? (as opposed to in stacks)

We need to have at least one O(1) operation at the front and back.

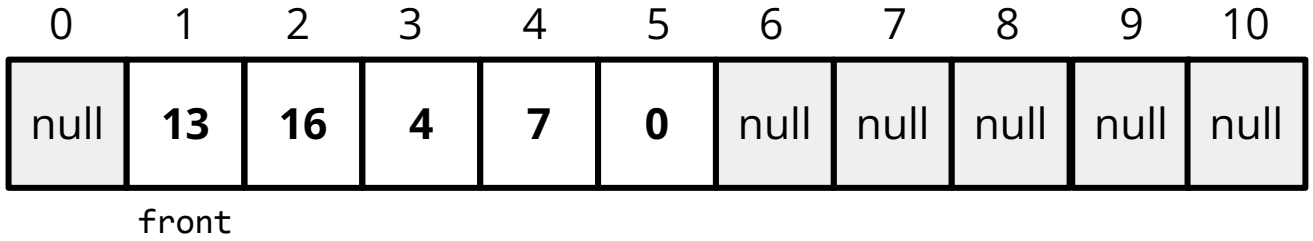
Queue: Circular Array Implementation




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

- Uses the array differently than the ArrayList
- Data does *not* have to be 0-aligned, but it must be contiguous
- We use a `front` variable to keep track of the index of the first element in the queue.
- The back of the queue: **`(front + size - 1) % backingArray.length`**

Queue: Circular Array Implementation




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

ADD TO BACK

What is the index we would add the new data at?

(front + size) % backingArray.length

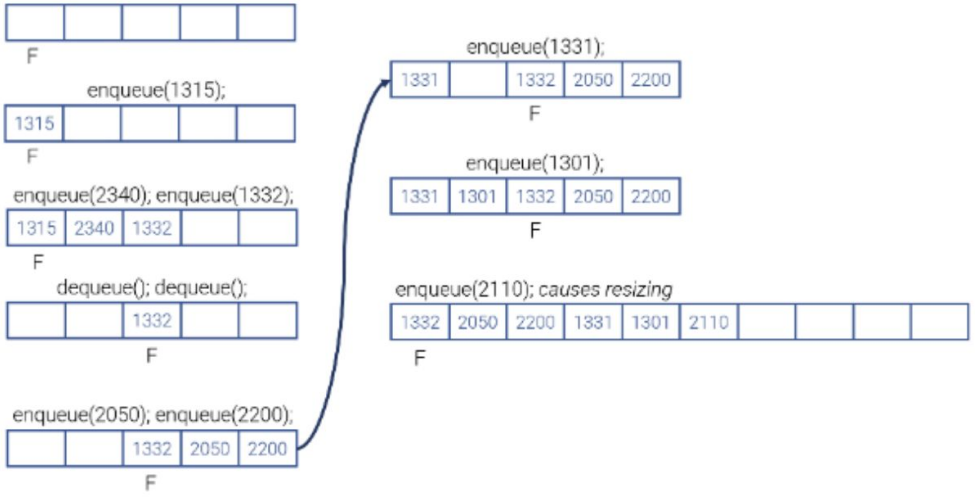
REMOVE FROM FRONT

1. Save the data at index `front`.
2. Set the backing array at `front` to null.
3. Increment `front`, deal with wrap around as necessary.

Queue: Circular Array Practice

Perform the following operations on an empty array-backed queue of initial capacity 5.

enqueue(1315)
enqueue(2340)
enqueue(1332)
dequeue()
dequeue()
enqueue(2050)
enqueue(2200)
enqueue(1331)
enqueue(1301)
enqueue(2110) -> causes resizing



Queue Implementations

	Array	SLL w/ tail	DLL w/ tail
Enqueue	Back	Back	Either
Dequeue	Front	Front	Opposite end as enqueue

O(1) enqueue and dequeue

NOTE: Array-backed queue enqueue is O(1)* due to the resize case.

Deque ADT

- Unordered, not iterable
- Backed by a circular array or a DLL w/ tail
- Basically a queue and a stack combined, add & remove operations at both ends

```
void addToFront(T data)

void addToBack(T data)

T removeFromFront()

T removeFromBack()
```

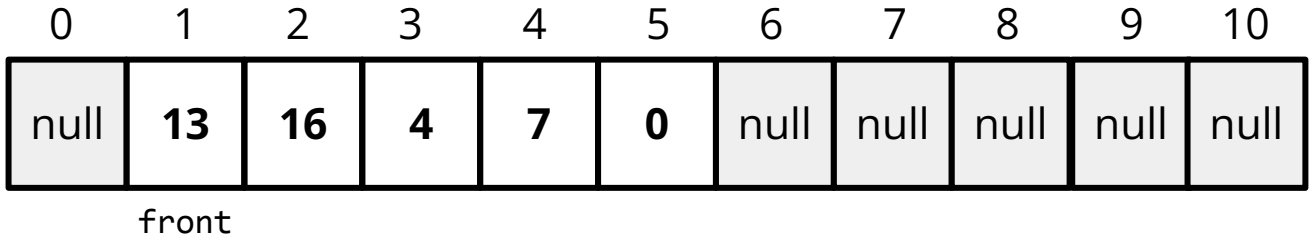
What are use cases of a deque?

deck of cards, undo/redo function

Why do we have to have a DLL w/ tail now? (as opposed to SLL in stacks/queues)

We need to have O(1) add AND remove operations at both ends.

Deque: Circular Array Implementation




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

ADD TO FRONT

1. Decrement `front`, deal with wrap around as necessary.
2. Place the new data at index `front`.

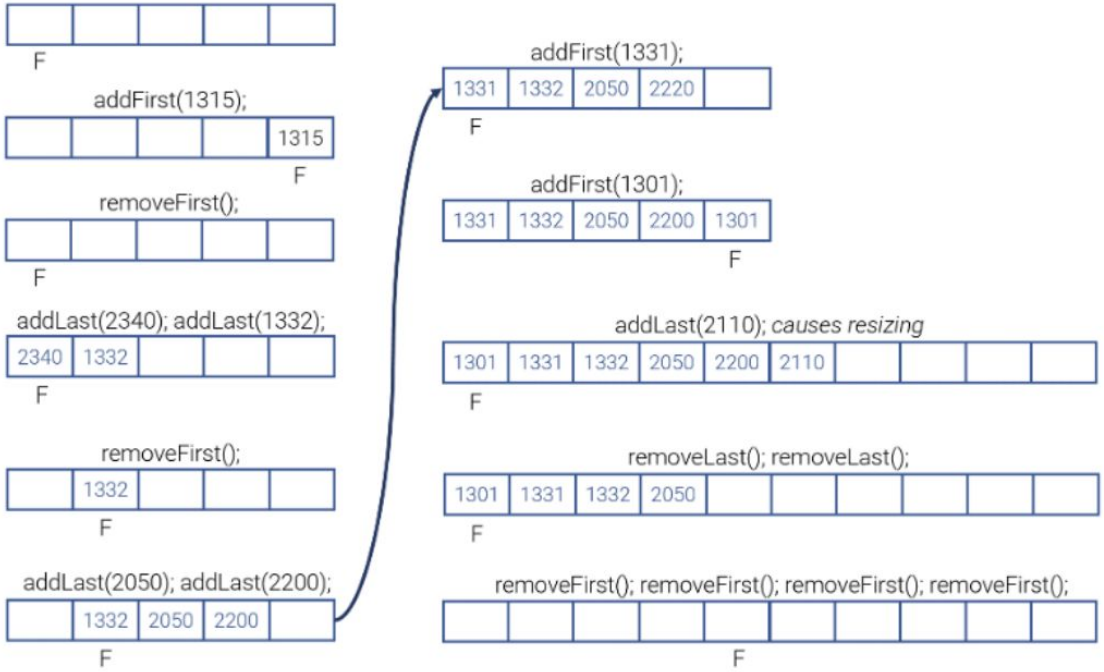
REMOVE FROM FRONT

1. Save the data at index `front`.
2. Set the backing array at `front` to null.
3. Increment `front`, deal with wrap around as necessary.

Deque: Circular Array Practice

Perform the following operations on an empty array-backed deque of initial capacity 5.

addFirst(1315)
removeFirst()
addLast(2340)
addLast(1332)
removeFirst()
addLast(2050)
addLast(2200)
addFirst(1331)
addFirst(1301)
addLast(2110)



Deque Implementations

	Array or DLL w/ tail
addFirst()	Front
addLast()	Back
removeFirst()	Front
removeLast()	Back

Everything is O(1)

NOTE: Array-backed deque add operations are O(1)* due to the resize case.

LEETCODE PROBLEMS

20. Valid Parentheses

71. Simplify Path

141. Linked List Cycle

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

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