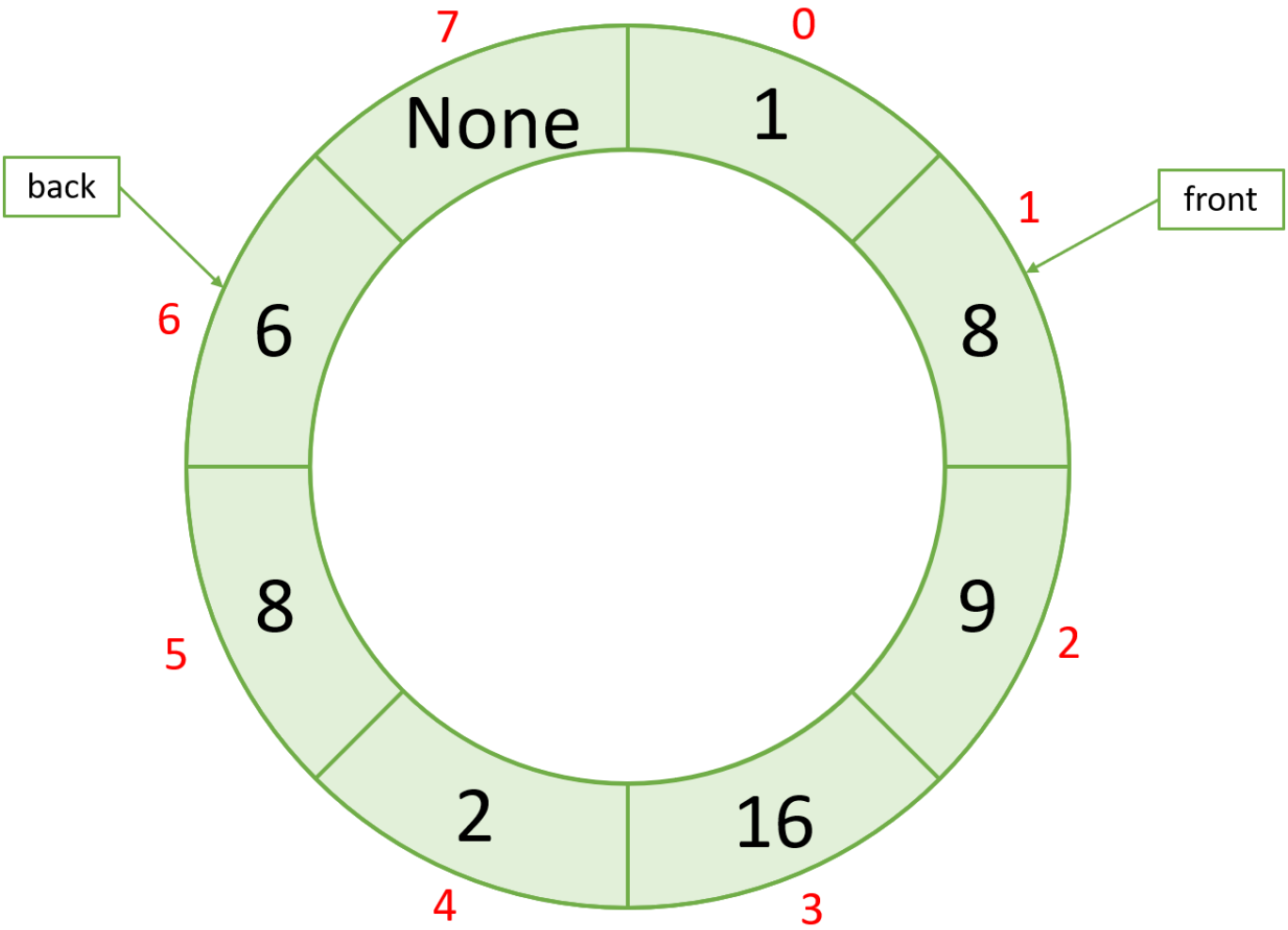
# Project 5: Circular Double-Ended Queues (Deque)

**Due: Thursday, March 3rd @ 10:00pm pm EST**

*This is not a team project, do not copy someone else's work.*




## Assignment Overview

In a typical FIFO (First in First out) queue, elements are added to one end of the underlying structure and removed from the opposite. These are natural for storing sequences of instructions: Imagine that instructions are added to the queue when first processed, and removed when completed. The first instruction processed will also be the first completed - we add it to the front, and remove it from the back.

A deque is a double-ended queue, meaning elements can be added or removed from either end of the queue. This generalizes the behavior described above to account for more complex usage scenarios. The ability to add or remove from both ends of the deque allows the structure to be used as both a FIFO queue and a LIFO stack, simultaneously.

This structure is useful for storing undo operations, where more recent undos are pushed and popped from the top of the deque and old/expired undo are removed from the back of the deque. Trains, consisting of

sequences of cars, can also be thought of as deques: cars can be added or removed from either end, but never the middle.

A circular queue is a queue of fixed size with end-to-end connections. This is a way to save memory as deleted elements in the queue can simply be overwritten. In the picture above at index 0, element 1 has been removed (dequeued) from the queue but the value remains. If two new values are enqueued, then that 1 will be overwritten. After this, the circular queue will have reached capacity, and need to grow.

Circular queues are useful in situations with limited memory. Consider a router in an internet network. A package (a set of bits sent across the network) is sent to this router and it joins the router's processing queue. This router can only hold so many packets before it has to start dropping some. A circular queue would be useful here, as it optimizes memory usage.

A circular deque is a combination of a deque and a circular queue. It sets a max size and can grow and shrink like a circular queue, and it can enqueue/dequeue from both ends.

There are many equivalent implementations of circular deques, each with their own rationale and use cases. For this project you will implement a circular deque using both a circular array and a circular doubly linked list (CDLL) as underlying structures. A function, **plot_speed**, is provided for comparison of the two structures.

## Assignment Notes

TIPS:

- The use of modulo (%) is highly recommended
- Understand what amortized runtime is (also explained below)
- Enqueue and Dequeue both have basic tests which test their functionality in conditions where shrink and grow will not be called. This allows you to test your enqueue and dequeue functions without having to implement grow/shrink.
- Although the API lists enqueue/dequeue first, it is common to implement grow/shrink and then enqueue/dequeue or grow->enqueue then shrink->dequeue. The test cases are designed to allow you to implement these functions independently in the order which best suits you.

RULES:

- The use of Python's Queues library is **NOT ALLOWED** and any use of it will result in a 0 on this project
- The use of `.pop()` is **PROHIBITED.**
  - Any function using .pop() will be deducted all points for test cases and manual grading
  - `.pop(x)` has a runtime of *O(n-x)*, where *n* is the length of the python list `.pop(x)` is called on - in most situations, this will violate time complexity.
- Use of the **nonlocal** keword will result in a 0 on the function is used on
  - You should never need to use this keyword in this project and if you are using it in a function in this class, you're doing something wrong.

## Assignment Specifications

**class CircularDeque:**

*DO NOT MODIFY the following attributes/functions*

- **Attributes**
  - **capacity: int:** the total amount of items that can be placed in your circular deque. This grows and shrinks dynamically, but is never less than 4. Will always be greater than or equal to ;**size**.
  - **size: int:** the number of items currently in your circular deque
  - **queue: list[T]:** The underlying structure holding the data of your circular deque. Many elements may be **None** if your current **size** is less than **capacity**. This grows and shrinks dynamically.
  - **front: int:** an index indicating the location of the first element in the circular deque
  - **back: int:** an index indicating the location of the last element in your circular deque
- **__init__(self, data: list[T], front: int, capacity: int) -> None**
  - Constructs a circular deque
  - **data: list[T]:** a list containing all data to be inserted into the circular deque
  - **front: int:** An index to offset the front pointer to test the circular behavior of the list without growing
  - **capacity: int:** the capacity of the circular deque
  - **Returns:** None
- **__str__(self) -> str** and **__repr__(self) -> str**
  - Represents the circular deque as a string
  - **Returns:** str

*IMPLEMENT the following functions*

- **__len__(self) -> int**
  - Returns the length/size of the circular deque - this is the number of items currently in the circular deque, and will not necessarily be equal to the **capacity**
  - This is a magic method and can be called with **len(object_to_measure)**
  - Time complexity: *O(1)*
  - Space complexity: *O(1)*
  - **Returns:** int representing length of the circular deque
- **is_empty(self) -> bool**
  - Returns a boolean indicating if the circular deque is empty
  - Time complexity: *O(1)*
  - Space complexity: *O(1)*
  - **Returns:** True if empty, False otherwise
- **front_element(self) -> T**
  - Returns the first element in the circular deque
  - Time complexity: *O(1)*
  - Space Complexity: *O(1)*
  - **Returns:** the first element if it exists, otherwise None
- **back_element(self) -> T**
  - Returns the last element in the circular deque
  - Time complexity: *O(1)*
  - Space complexity: *O(1)*
  - **Returns:** the last element if it exists, otherwise None
- **enqueue(self, value: T, front: bool = True) -> None:**
  - Add a value to either the front or back of the circular deque based off the parameter **front**
  - if front is true, add the value to the front. Otherwise, add it to the back
  - Call **grow()** if the size of the list had reached capacity

    - **param value: T:** value to add into the circular deque
    - **param value front:** where to add value T
    - Time complexity: *O(1)**
    - Space complexity: *O(1)**
    - **Returns:** None
- **dequeue(self, front: bool = True) -> T:**
    - Remove an item from the queue
    - Removes the front item by default, remove the back item if False is passed in
    - Calls **shrink()** If the current size is less than or equal to 1/4 the current capacity, and 1/2 the current capacity is greater than or equal to 4, halves the capacity.
    - **param front:** Whether to remove the front or back item from the dequeue
    - Time complexity: *O(1)**
    - Space complexity: *O(1)**
    - **Returns:** removed item, None if empty
- **grow(self) -> None**
    - Doubles the capacity of CD by creating a new underlying python list with double the capacity of the old one and copies the values over from the current list.
    - The new copied list will be 'unrolled' s.t. the front element will be at index 0 and the tail element will be at index [size - 1].
    - Time complexity: *O(n)*
    - Space complexity: *O(n)*
    - **Returns:** None
- **shrink(self) -> None**
    - Cuts the capacity of the queue in half using the same idea as grow. Copy over contents of the old list to a new list with half the capacity.
    - The new copied list will be 'unrolled' s.t. the front element will be at index 0 and the tail element will be at index [size - 1].
    - Will never have a capacity lower than 4, **DO NOT** shrink when shrinking would result in a capacity <= 4
    - Time complexity: *O(n)*
    - Space complexity: *O(n)*
    - **Returns:** None

***Amortized**. *Amortized Time Complexity* means 'the time complexity a majority of the time'. Suppose a function has amortized time complexity *O(f(n))* - this implies that the majority of the time the function falls into the complexity class *O(f(n)),* however, there may exist situations where the complexity exceeds *O(f(n))*. The same logic defines the concept of *Amortized Space Complexity*.

Example: `enqueue(self, value: T, front: bool)` has an amortized time complexity of *O(1)*: In the majority of situations, enqueueing an element occurs through a constant number of operations. However, when the Circular Deque is at capacity, grow(self) is called - this is an *O(n)* operation, therefore in this particular scenario, enqueue exceeds its amortized bound.

# The (not) Application Overview: *Another Circular Deque!!!*

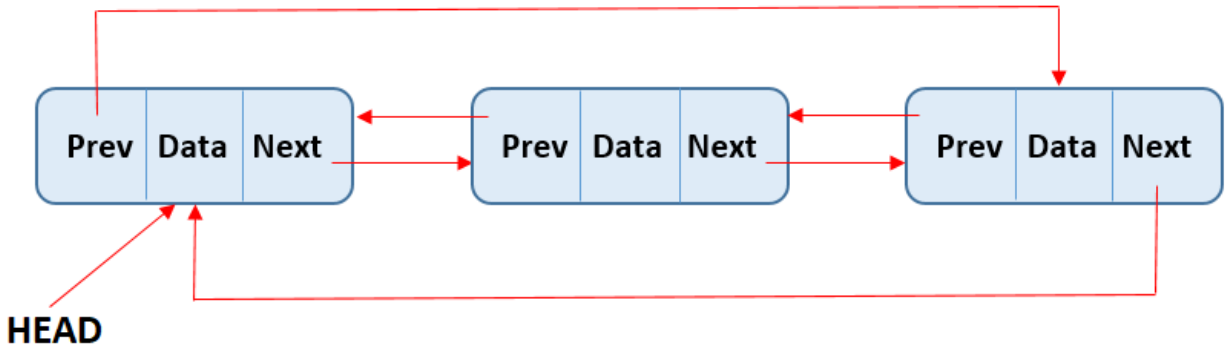
After graduation, you've been hired by an exciting new tech start-up based out of silicon valley. On the first day, your new supervisor, Charles C. "not the MSU Professor" Owen has a quote for you: "there is a word for people who implement circular deques utilizing python lists as the underlying structure: unemployed."

As you mull about the office on your first day, you hear horror stories. This guy hates lists! He flies into an explicable rage whenever he sees one. The threat is clear: if you use a python list for your first assignment, creating a circular deque class for the companies internal use, you'll be fired.

Luckily, Charles "If you use a list, you will face my fists" Owen has provided for your use his own class: the **CDLL (Circular Doubly Linked List)**. It isn't perfect, but it will get the job done.

Throughout the body of this project, you have created an interface via the methods of the **CircularDeque** class for a more complex structure below. Functions such as **grow** and **shrink** would never be called by a user of the class during a typical use case, and the underlying data array never accessed directly - in many languages, such as C++, access to these functions and variables may be restricted entirely to create a simplified forward facing API.

## **Expectations**:




You are given the classes described below. **DO NOT** modify these classes - any modification will result in a zero for this portion of the project.

**class CDLLNode**

*DO NOT MODIFY the following attributes/functions*

- **Attributes:**
  - **val: T:** value stored by the node
  - **next: CDLLNode:** The next node in the **CDLL**
  - **prev: CDLLNode:** The previous node in the **CDLL**
- **__init__(self, val: T, next: CDLLNode = None, prev: CDLLNode = None) -> None**
  - Constructs a **CDLLNode**
  - **param val:** value stored by the node
  - **param next: CDLLNode:** The next node in the **CDLL**
  - **param prev: CDLLNode:** The next node in the **CDLL**
  - **return: None**
- **__eq__(self, other: CDLLNode) -> bool**
  - Compares two **CDLLNode** objects by value
  - **param other: CDLLNode:** The other node
  - **return:** True if the comparison is true, else false
- **__str__(self) -> str**
  - returns a string representation of the **CDLLNode**
  - **return:** a string

**class CDLL:**

*DO NOT MODIFY the following attributes/functions*

- **Attributes:**
  - **head: CDLLNode:** The head of the **CDLL**
  - **size: int:** the number of nodes in the **CDLL**
- **__init__(self) -> None**
  - Creates a **CDLL**
  - **return: None**
- **__eq__(self, other: CDLL) -> bool:**

    - Compares two **CDLL** objects by value
    - **param other: CDLL:** the other list to compare
    - **return:** True if the comparison is true, else false
- **__str__(self) -> str**
    - returns a string representation of the **CDLL**
    - **return:** a string
- **insert(self, val: T, front: bool = True) -> None:**
    - inserts a node with value **val** in the front or back of the **CDLL**
    - **param val: T:** the value to insert
    - **param front: bool = True:** whether to insert in the front of the list, or the back.
    - **return: None**
- **remove(self, front: bool = True) -> None:**
    - removes a node from the **CDLL**
    - **param front: bool = True:** whether to remove from the front of the list, or the back
    - **return: None**

Your mission is to reimplement the functionality of **CircularDeque** with a **CDLL** as the underlying structure.

**class CDLLCD:**

**You are required to implement the following two functions:**

- **enqueue(self, val: T, front: bool = True) -> None**
    - adds a value to the **CDLLCD**
    - **param val: T:** the value to be added
    - **param front: bool = True:** whether to add to the front or the back of the deque
    - Time complexity: *O(1)*
    - Space complexity: *O(1)*
    - **return: None**
- **dequeue(self, front: bool = True) -> T**
    - Removes a value from the deque, returning it
    - **param front: bool = True:** whether to remove from the front or the back of the deque
    - Time complexity: *O(1)*
    - Space complexity: *O(1)*
    - **return: None**

**Three other functions are provided. YOU MAY MODIFY THESE IN ANY WAY YOU PLEASE, INCLUDING TO ADD ATTRIBUTES TO THE CLASS:**

- **__init__(self) -> None**
    - Creates a **CDLLCD** object
    - **return: None**
- **__eq__(self, other: CDLLCD) -> bool**
    - compares two **CDLLCD** objects by value
    - **param other:** the other **CDLLCD**
    - **return:** True if the comparison evaluates to true, else false
- **__str__(self) -> str:**
    - returns a string representation of the **CDLLCD**

    - **return:** a string

## Guarantees/Notes

- You must actually use the underlying **CDLL** in the provided skeleton class. The testcase check for this, but attempting to get around the checks will result in a zero.
- In his list-fuel rage, your boss yelled something about a sentintel node. May beworth checking out...
- Once completed, try running the function **plot_speed** to see how your implementation compares to the main project implementation. They should be pretty close.

## Submission

**Deliverables**

Be sure to upload the following deliverables in a .zip folder to Mimir by 11:59p Eastern Time on Friday, 03/03/22.

Your .zip folder can contain other files (for example, specs.md and tests.py), but must include (at least) the following:

```
|- Project5.zip
    |— Project5/
        |— feedback.xml          (for project feedback)
        |— __init__.py         (for proper Mimir testcase loading)
        |— solution.py    (contains your solution source code)
```

**Grading**

The following 100-point rubric will be used to determine your grade on Project4:

- Tests (70)
    - 00 - Coding Standard: __/3
    - 01 - len(): __/2
    - 02 - is_empty: __/2
    - 03 - front_element: __/2
    - 04 - back_element: __/2
    - 05 - front_enqueue_basic: __/2
    - 06 - back_enqueue_basic: __/2
    - 07 - front_enqueue: __/5
    - 08 - back_enqueue: __/5
    - 09 - front_dequeue_basic: __/2
    - 10 - back_dequeue_basic: __/2
    - 11 - front_dequeue: __/5
    - 12 - back_dequeue: __/5
    - 13 - grow: __/4
    - 14 - shrink: __/4
    - 15 - Circular Deque Comprehensive: __/10

    - 16 - Application: __/10
    - 99 - Feedback.xml: __/3
- Manual (30)
    - M0 - len(): __/1
    - M1 - is_empty: __/1
    - M2 - front_element: __/2
    - M3 - back_element: __/2
    - M4 - front_enqueue: __/3
    - M5 - back_enqueue: __/3
    - M6 - front_dequeue: __/3
    - M7 - back_dequeue: __/3
    - M8 - grow: __/3
    - M9 - shrink: __/3
    - M10 - application: __/6

*This project was created by Jacob Caurdy and Andrew Haas*

*Inspired by previous 331  CircularDeque project created by Angelo Savich and Olivia Mikola*