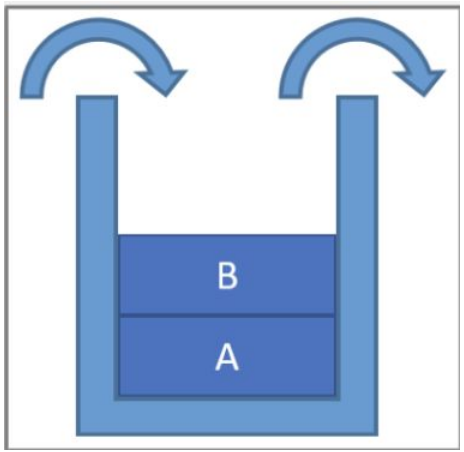
Stacks, Queues, Deques

1332 Recitation: Week of May 18th

The Stack

- A last-in, first-out (LIFO) abstract data type (ADT)
  - In other words, adds and removes occur on the *same side* of the structure
- OPERATIONS:
  - push(data) - add the data to the “top” of the stack
  - pop() - remove the data at the top of the stack
  - peek() - optional, returns the data at the top without removing
- Possible backing structures, chosen for efficiency reasons:
  - Array or ArrayList
  - Singly Linked List
  - Doubly Linked List



The SLL-backed Stack

- Does *not* require a tail pointer
- The “top” of the stack is actually the front of the linked list
- Very simple, uses only SLL operations we have already discussed
  - push(data) -> addToFront(data)
  - pop() -> removeFromFront()

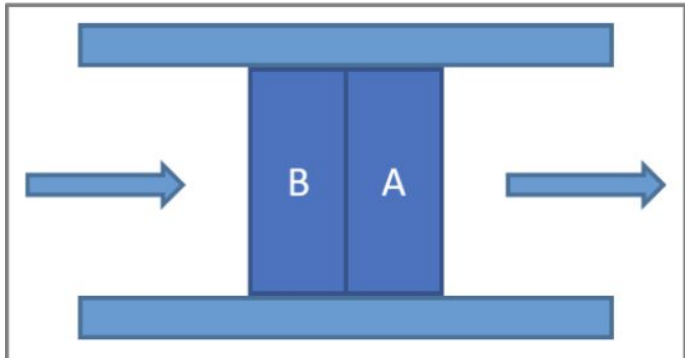
The array-backed Stack

- Requires a size variable in addition to the array
- The “top” of the stack is the back of the array
  - arr[size] is the next empty index, so this is where we push
  - arr[size - 1] is the element at the “top,” so this is where we pop

- Note about ArrayList
  - Since all data always starts at index 0 and is contiguous (no empty indices between elements), you could also use an ArrayList to back a stack

The Queue

- A first-in, first-out (FIFO) abstract data type
  - i.e., adds and removes occur at *opposite* ends of the structure
- OPERATIONS:
  - enqueue(data) - add data to the “back” of the queue
  - dequeue() - remove data from the “front” of the queue
  - peek() - optional, look at the data at the “front”
- Possible backing structures:
  - Array (with circular behavior)
  - Singly Linked List w/ tail
  - Doubly Linked List w/ tail



The SLL-backed Queue

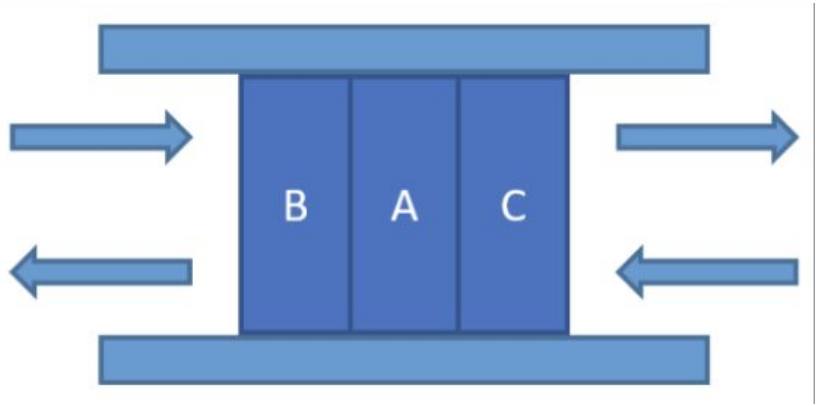
- *Requires* a tail pointer to get O(1) operations
- The “front” of the queue is the front of the list
- The “back” of the queue is the back of the list
- Operations map directly to O(1) linked list operations:
    - enqueue(data) -> addToBack(data)
    - dequeue() -> removeFromFront()

The array-backed Queue

- Requires a size variable, a front variable, and an array
- The array behaves circularly
  - i.e. when there is not space at the end of the array, we wrap the data around to the front
- arr[front] will *always* be the “front” of the queue
- arr[(front + size) % arr.length] will always be the first empty index at the “back”
- For enqueue():
  - Put the element at arr[(front + size) % arr.length], then increment the size
- For dequeue():
  - Remove the element at arr[front], then increment front and decrement size
    - front = (front + 1) % arr.length so that front doesn’t ever go out of bounds

The Deque

- Short for “double-ended queue”
  - We can add and remove to either side of the structure
- OPERATIONS:
  - addFirst(data)
  - addLast(data)
  - removeFirst()
  - removeLast()
- Possible backing structures:
  - Array (very similar to the array-backed queue)
  - Doubly Linked List w/ tail



The DLL-backed Deque

- *Requires a tail*
- addFirst(data) -> addToFront(data)
- addLast(data) -> addToBack(data)
- removeFirst() -> removeFromFront()
- removeLast() -> removeFromBack()

The array-backed Deque

- Keep a front variable and a size variable
- Most important thing is the **indices**
  - arr[(front - 1) % capacity] is where we addFirst()
  - arr[front] is where we removeFirst()
  - arr[(front + size) % capacity] is where we addLast()
  - arr[(front + size - 1) % capacity] is where we removeLast()
- When doing an operation to the front, you *must* update the front variable
  - After removeFirst(), front = (front + 1) % cap
  - After addFirst(), front = (front - 1) % cap
- After doing any operation, you *must* update the size

Comparable

- Interface Comparable<T>
- Allows us to define a “natural ordering” in the class so that we can compare objects to each other
- Only one method -- compareTo(other)
  - a.compareTo(b) returns a **negative** int if a < b
  - a.compareTo(b) returns **0** if a and b are equal
  - a.compareTo(b) returns a **positive** int if a > b
  - Think of it as returning the sign of (a - b)

Comparator

- Also an interface
- Allows us to compare objects based on our own definition
  - compareTo() is written in the class of the objects we want to compare, but we can define our own comparator in any class we want
- Only one method: compare(a, b)
  - comparator.compare(a, b) functions the same way as a.compareTo(b)