1. Review of Array-based List ADT & Q/A
2. Time complexity of ArrayList operations
3. Special use-cases: stacks & queues
4. Array-based Stack ADT
5. Stack applications
6. Array-based (circular) Queue ADT
7. Queue applications
8. Priority queue -- aka. Heap ADT

* * *

# 1. Review

# 2. Time complexity of ArrayList operations

for n elements:

index-based read/update = O(1)
append = O(1)
del from end = O(1)
del (worst-case) = O(n)
insert (worst-case) = O(n)
extend m-element list by n elements = O(n)
search (worst-case) = O(n) --- due in part to unordered contents of list
count = O(n)

lot of O(n) ops! all having to do with working with items in the "middle" of the list ...
important for general purpose API, but can be eliminated for many special purpose
applications

# 3. Stacks & Queues

Stack: last-in, first-out (LIFO) ADT --- only allow access to data at one end of the
sequence: the "top" of the stack
- clear analogy: a stack of books
- operations: *push* onto the stack, *pop* off the stack (and maybe *peek* at the top
item)

Queue: first-in, first-out (FIFO) ADT --- items are added at one end and taken off the
other
- useful for modeling, say, fair scheduling models (e.g., tracking multiple tasks for a single worker)
- operations: *enqueue* at the tail of the queue; *dequeue* off the head of the queue

# 4. Array-based Stack

(Trivial!)

* * *

class Stack:
    def __init__(self):
        self.data = []

    def push(self, x):
        self.data.append(x)

    def pop(self):
        lastidx = len(self.data)-1
        val = self.data[lastidx]
        del self.data[lastidx]
        return val

    def peek(self):
        return self.data[len(self.data)-1]

    def __repr__(self):
        return repr(self.data)

    def __len__(self):
        return len(self.data)

    def __bool__(self):
        return len(self.data) > 0

# 5. Stack Applications

Conceptually, anything that requires simulating LIFO semantics --- i.e., most recent item/order/operation must be satisfied before earlier ones. E.g.,

- balancing parentheses (consider examples)
- backtracking (e.g., exploring a maze or implementing a multi-level search)
- returning to preceding function calls
def check_parens(expr):
    s = Stack()
    for c in expr:
        if c == '(':
            s.push(c)
        elif c == ')':
            if not s:
                return False
            else:
                s.pop()
    return len(s) == 0

def fib_rec(n):
    if n <= 1:
        return 1
    else:
        return fib_rec(n-1) + fib_rec(n-2)

def fib_stack(n):
    s = Stack()
    s.push(n)
    fib = 0
    while s:
        n = s.pop()
        if n <= 1:
            fib += 1
        else:
            s.push(n-1)
            s.push(n-2)
    return fib

# 6. Array-based Queue implementation

Naive implementation:

class Queue:
    def __init__(self):
        self.data = []

    def enqueue(self, x):
self.data.append(x)

def dequeue(self):
    val = self.data[0]
    del self.data[0]
    return val

Problem: dequeue is a O(n) operation!
- unnecessary problem ... the user doesn't care about indexes

* * *

Alternative: use a fixed-length queue, and allow indexes to "wrap around" in the implementation

class QueueCircular:
    def __init__(self, size=10):
        self.data = [None] * size
        self.count = 0
        self.head  = self.tail = 0

    def enqueue(self, x):
        if self.count < len(self.data):
            self.data[self.tail] = x
            self.tail = (self.tail + 1) % len(self.data)
            self.count += 1

    def dequeue(self):
        if self.count > 0:
            val = self.data[self.head]
            self.head = (self.head + 1) % len(self.data)
            self.count -= 1
            return val

    def __iter__(self):
        for offset in range(self.count):
            yield self.data[(self.head + offset) % len(self.data)]

    def __repr__(self):
        return repr(list(self))

* * *
Another (smart) alternative: Queue from two Stacks!

class QueueFromStacks:
    def __init__(self):
        self.inbox = stack.Stack()
        self.outbox = stack.Stack()

    def enqueue(self, x):
        self.inbox.push(x)

    def dequeue(self):
        if self.outbox:
            return self.outbox.pop()
        if self.inbox:
            while self.inbox:
                self.outbox.push(self.inbox.pop())
            return self.outbox.pop()

Performance?

# 7. Queue applications

- simulating any sort of FIFO system (e.g., helpdesk, packet router, web server, OS scheduler)
- handling an in-application "work" queue
- exploring a graph "breadth-first"

# 8. Priority Queue

Motivation: sometimes, instead of a strictly fair queueing policy, we want to be able to pull items out of the queue by way of some arbitrary priority; i.e., enqueue in arbitrary order, but dequeue in predefined order (e.g., ascending)

Naive implementation would require maintaining a fully sorted array of items ...

How to do this? Big-O time complexity of enqueue operation? (Discussion)

* * *