Basic Data Structures III

- ADTs
- Stack ADT
- Queue ADT
- Deque ADT
- Implementations
- Amortized Analysis

Part 1: Abstract Data Types (ADTs)

ADT: Abstract Data Type

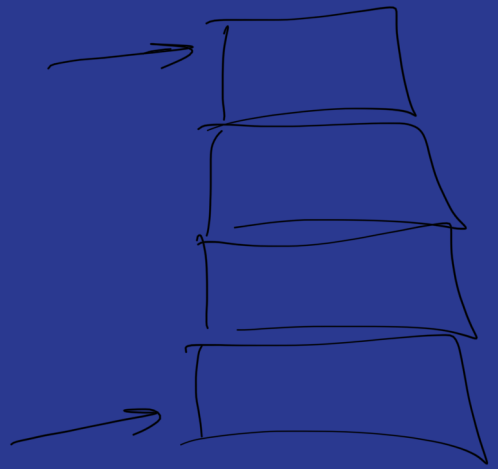
- abstraction of a data structure
- defines operations
- separate from implementation
- may have an “ideal” runtime for operations, but the actual runtimes will be determined by the implementation

The Value of Defining a (narrowly defined) ADT

- HOW an operation is implemented matters.
- HOW an operation is implemented depends on how the data structure is implemented.
- Defining a data structure to do precisely what it needs to do and no more encourages discipline in programming, making code easier to understand.
- It forces a developer to think about which operations are ABSOLUTELY ESSENTIAL and which are not.
- Many algorithms depend on specific ADTs, so understanding the nature of an ADT can help with understanding the algorithm.

Part 2: Stacks

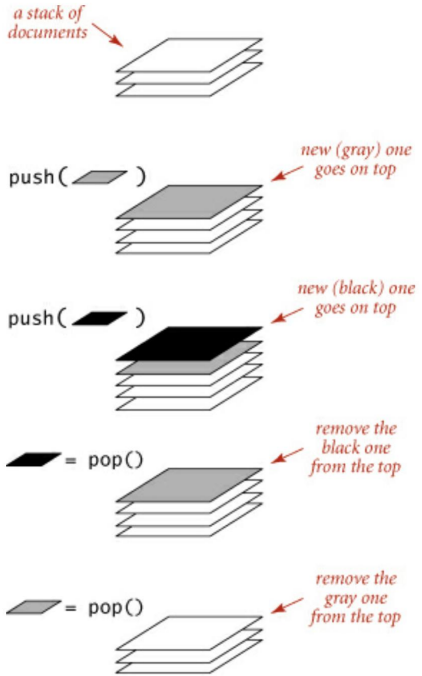
Last in, first out (LIFO)



Stacks


truth
Proud of what's
NOT
in our food.™
NEW
Light
Refreshing
Taste.
86 years late

Typical Stack Operations



Operations on a pushdown stack

- add an item to the top of the stack: push
- remove an item from the top of the stack: pop
- check if the stack is empty: isEmpty
- return the number of elements in the stack: size
- return the top item without removing it: peek / top

Picture from [1]

How would you implement a Stack?

linked list
array

Stacks: What are they good for?

→ recursive algorithm w/ backtracking

→ memory call stack

Stack Application:

Dijkstra's Two-Stack Algorithm for Expression Evaluation

EX:
(8 * ((7 + 3) - ((4+2)*(3 - 1))))

```
let S1 and S2 be empty stacks
for each character c in the expression do:
    if c is an operand
    then push it onto S1
    else if c is an operator
    then push it onto S2
    else if c is a right parenthesis
    then pop an operator o from S2
        pop the requisite number of operands from S1
        calculate the result of applying o to the operands and
        push the result onto S1

end for
return the last value on S1
```

Stack Application:

Memory Management

```
void foo(char *ptr) {
      char buf[16];
      strcpy(buf, ptr);
}

int main(int argh, char **argv) {
      foo(argv[1]);
      return 0;
}
```

Stack Application:

Forth

- A Stack-based programming language.
- Everything revolves around a single stack.
- Example:

```
5 4 +
8 -
2 *
```

```
variable f
: factorial 1 f ! 1 + 1 do i f
@ * f ! loop f ? ;
5 factorial
```

If a program uses a Stack, what does that tell us?

- We are dealing with data that needs to be stored and processed, and…
- we want to process it in LIFO order.

Part 3: Queues

First in, first out (FIFO)....

Typical Queue Operations

Picture from https://pngtree.com/freepng/people-png-elements_2835125.html

- add an item to the back of the queue: enqueue
- remove an item from the front of the queue: dequeue
- return the front item without removing it: peek
- check if the queue is empty: isEmpty
- return the number of elements in the queue: size

How would you implement a Queue?

Queues: What are they good for?

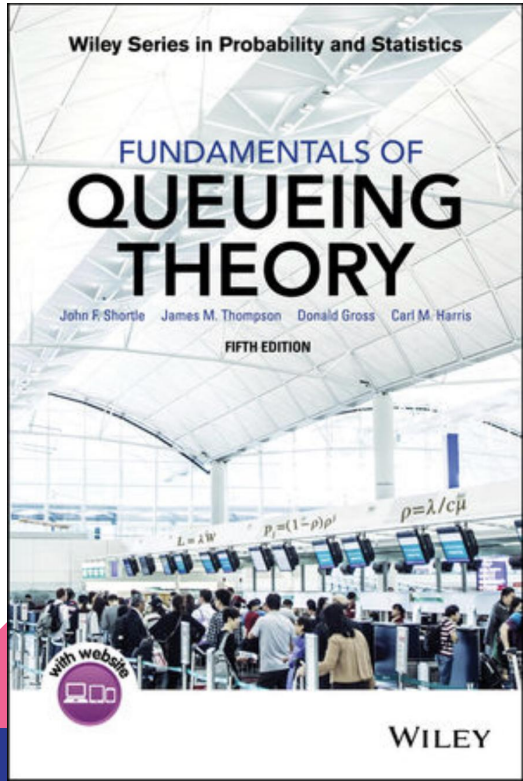
user → requests
networks

BFS

Queue Application:

Networks and Data Communication

- Sending
- Routing
- Receiving
- Processing



Queue Application:

Breadth-First-Search (BFS)

- A breadth-first search on a tree or graph seeks to search shorter paths before longer paths.
- To do that, we use a queue and put new items on the queue to be visited so that immediate neighbors are visited before things that are further away.

If a program uses a Queue, what does that tell us?

- We are dealing with data that needs to be stored and processed, and…
- we want to process it in FIFO order.

Part 4: Double-ended Queues (Deques)

Combines the functionality of a Stack and a Queue.

Typical Deque Operations

- add to the front
- add to the back
- remove from the front
- remove from the back
- return the front item without removing it
- return the back item without removing it
- check if the deque is empty
- get the number of elements in the deque

Part 5: Implementations

- Linked List Implementations
- Array-based Stack
- Array-based Queue
- Amortized Analysis

1. Ideally, how much space should a Stack/Queue take if we have ***N*** elements?

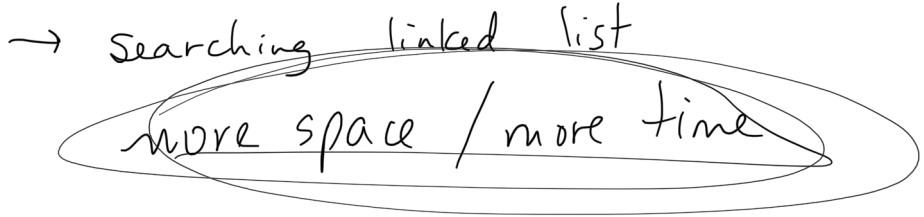
O(N)

2. Ideally, how much time should *push/pop/enqueue/dequeue* take if we have ***N*** elements?

O(1)

Linked List Implementation of Stacks and Queues

- As we discussed before, it takes ___O(1)___ time to add or remove elements from the front or the back of a linked list.
- This means that implementing a Stack or a Queue with a linked list is straightforward and will guarantee ___O(1)___ time for the basic operations.
- However, we should also consider the disadvantages of linked lists…



Arrays

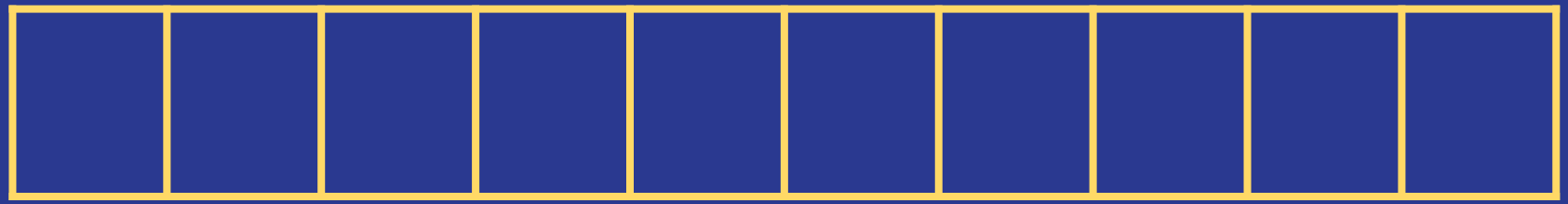
- Built in to Java (and many other languages)
- Size is fixed and specified when array is created
- So what can be done if you end up with more elements than you expected?

Implementing a Stack with an array

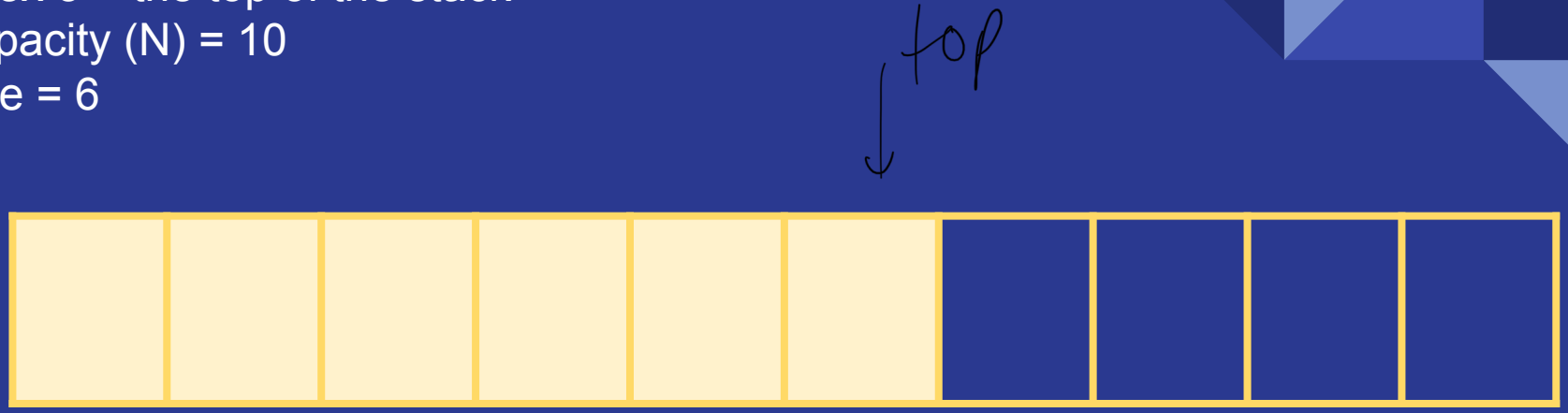
Dynamic Resizing

- Start with an array of size X.
- When the stack is full, make a new array of size 2X and copy the items over.
- If the size of the stack falls below a threshold (around ¼ of X), make a new array of size X/2 and copy the items over (keeps the size of the stack between ½ full and full.)
- This provides a nice balance for runtime and space efficiency (given the results of an *amortized analysis* of the runtime, which we will see later.)

Empty Stack
index 0 = the bottom of the stack
Capacity (N) = 10
Size = 0

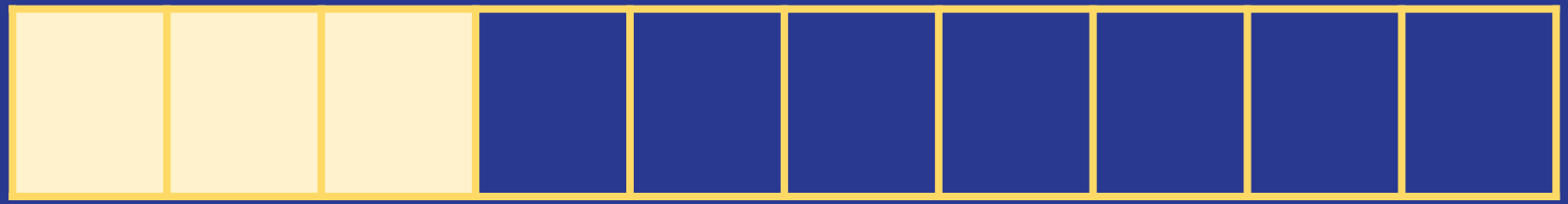


index 0 = the bottom of the stack
index 5 = the top of the stack
Capacity (N) = 10
Size = 6



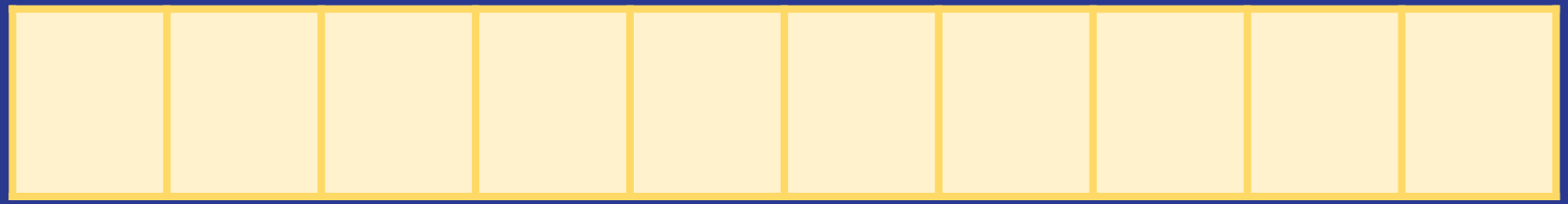
after *push*ing 6 items

index 0 = the bottom of the stack
index 2 = the top of the stack
Capacity (N) = 10
Size = 3



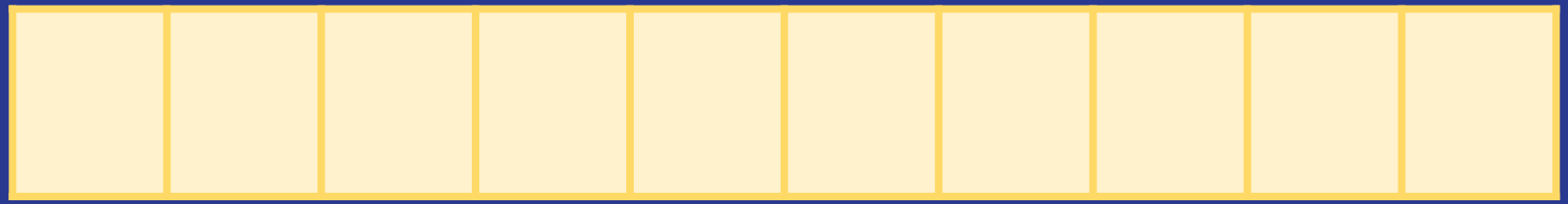
after *pop*ping 3 items

index 0 = the bottom of the stack
index 9 = the top of the stack
Capacity (N) = 10
Size = 10



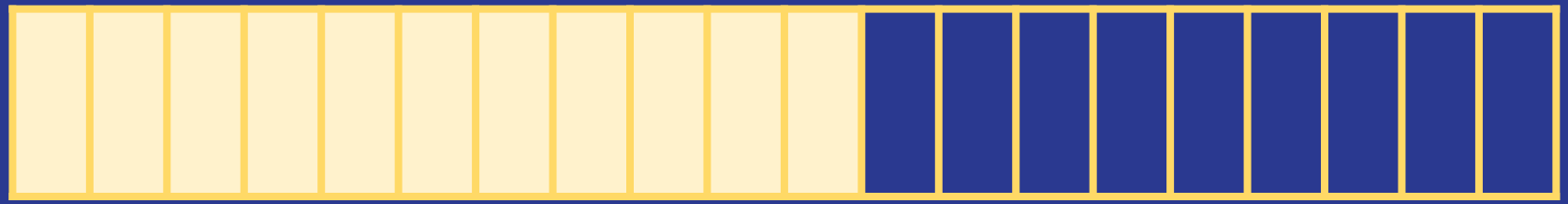
after pushing 7 items

index 0 = the bottom of the stack
index 9 = the top of the stack
Capacity (N) = 10
Size = 10



What do we do if we want to *push* one more item?

index 0 = the bottom of the stack
index 10 = the top of the stack
Capacity (N) = 20
Size = 11

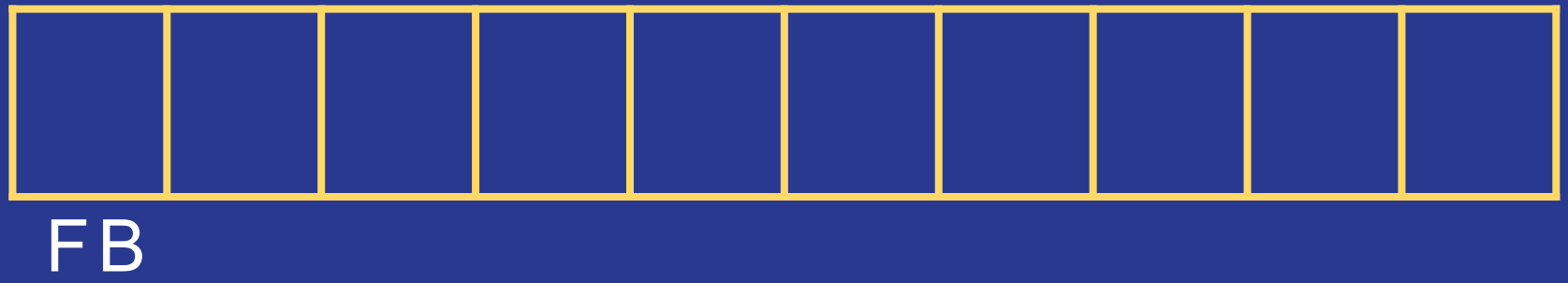


Create a new array that is twice as large and copy the items over...but how much time does that take?

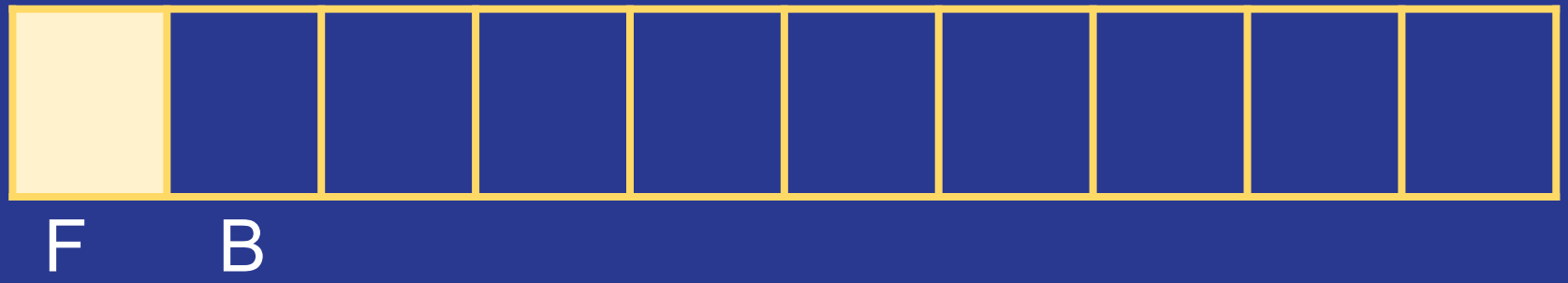
Queue implementation with array

- Dynamically resize the same as the Stack
- Differences with Stack
  - Names of operations
  - Operations and indexes
    - Need to keep track of front and back
    - Need to allow for adding to back and removing from front
    - How to keep track of size
    - Wrap around to utilize full capacity

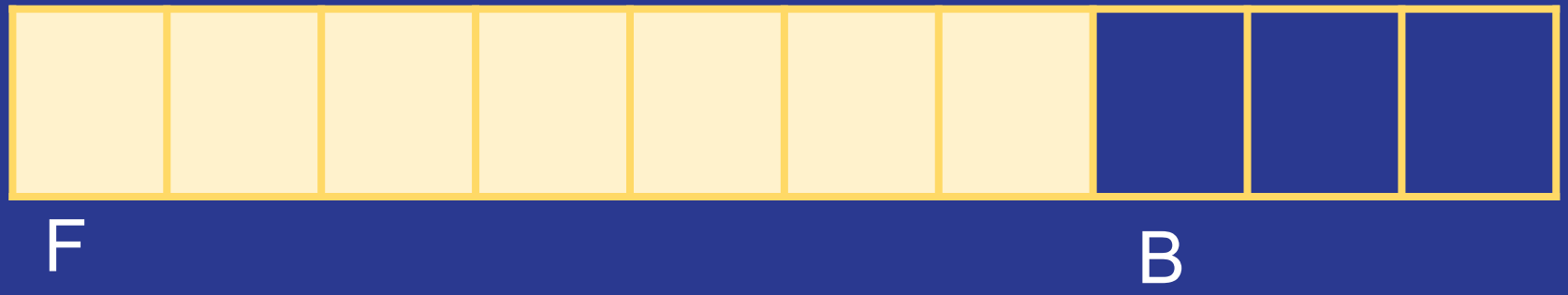
Empty Queue
F = index of the front
B = index of the back
Capacity (N) = 10
Size = 0



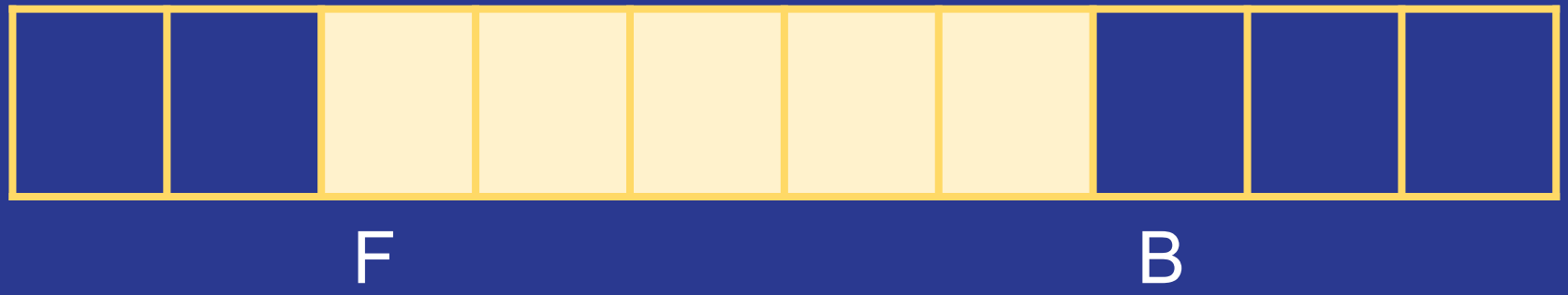
Enqueue 1 item
Size = 1



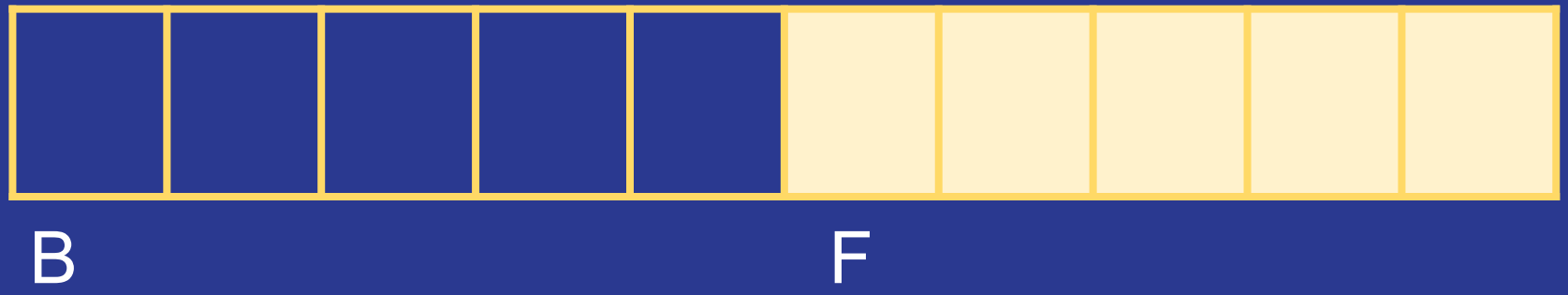
Enqueue 6 more items
Size = 7



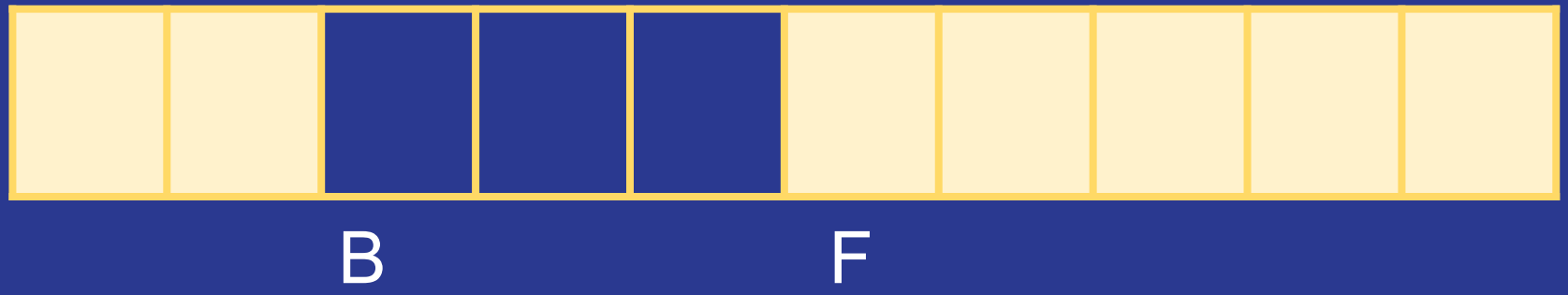
Dequeue 2 items
Size = 5



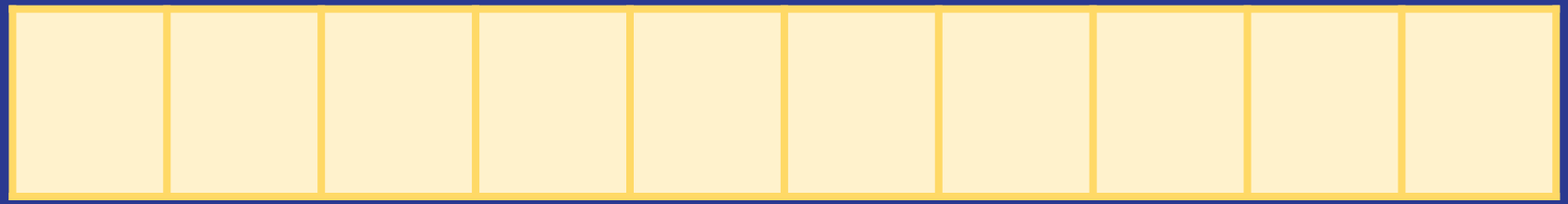
Dequeue 3 items and Enqueue 3 items
Size = 5
Index for Back wraps around



Enqueue 2 more items
Size = 7

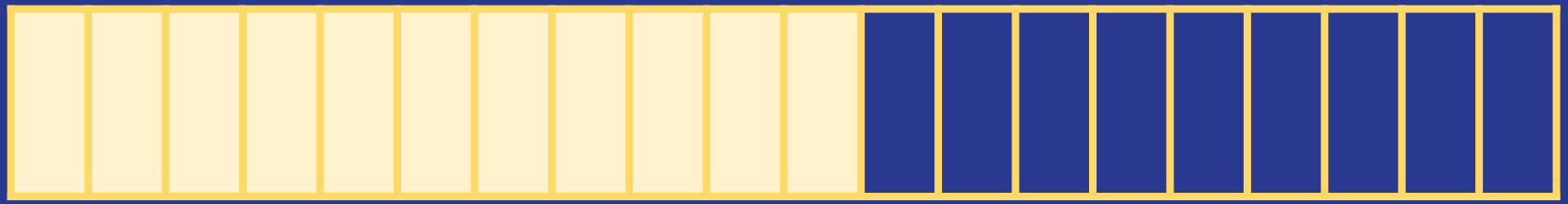


Enqueue 3 more items
Size = 10



B F

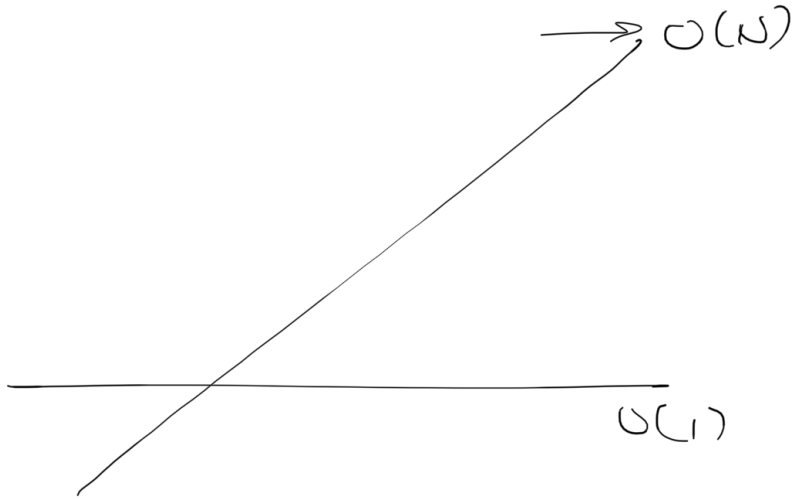
Enqueue 1 more item
Size = 11



Double the size of the array, copy the items over, and reset the pointers for front and back.

Part 6: Analysis

Analysis of Operations



Array-based Stack/Queue with dynamic size

- isEmpty: O(1)
- size: O(1)
- push/enqueue: best-O(1); worst-O(N)
- pop/dequeue: best-O(1); worst-O(N)

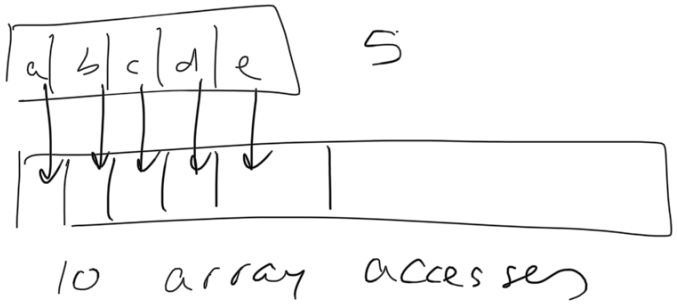
The two things we need to make sure of in order to guarantee good *amortized* time are:

- the worst-case runtime should only happen when the array is resized, which is guaranteed by the wrap-around method
- the resizing should be done by doubling (or something similar to that)

Amortized Runtime Analysis

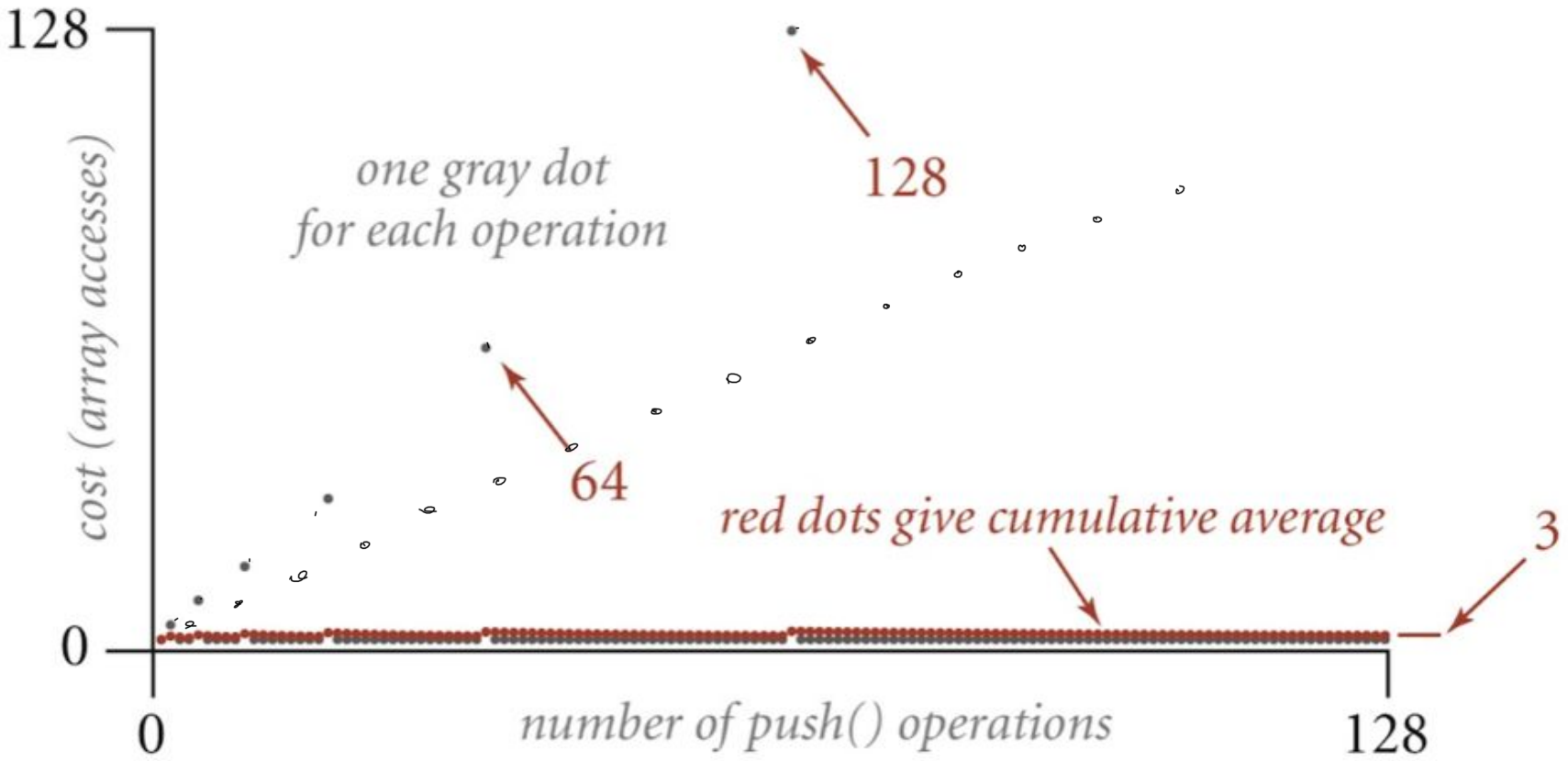
- used when analyzing the runtime of a single operation on a data structure
- may be useful when the best-case and worst-case runtimes vary quite a bit *and* the worst-case doesn't happen very often
- calculated by:
  - considering a worst-case string of *N* operations
  - adding up the total cost
  - dividing by *N*

Amortized Analysis for *push*



- A good cost model will be to count the *array accesses* (i.e. the number of times the array is accessed.)
- A worst-case scenario: N pushes (no pops)

- Each one of these *pushes* will access the array 1 time. (N array accesses)
- Plus, some of them will require resizing which will require *2M* array accesses, where *M* is the number of elements in the array when the array is resized.
- We will add up this total and then divide it by *N* to get the average cost of each *push*.



Picture from [1]

Amortized cost of push where the cost model is the number of array accesses and we resize by doubling.

total # of array accesses $= N + 2(1 + 2 + 4 + \dots + N/2)$

$$= N + 2\sum_{k=0}^{\log_2(N/2)} 2^k = N + 2\left[2^{\log N/2 + 1} - 1\right]$$

$$= N + 2(N - 1)$$

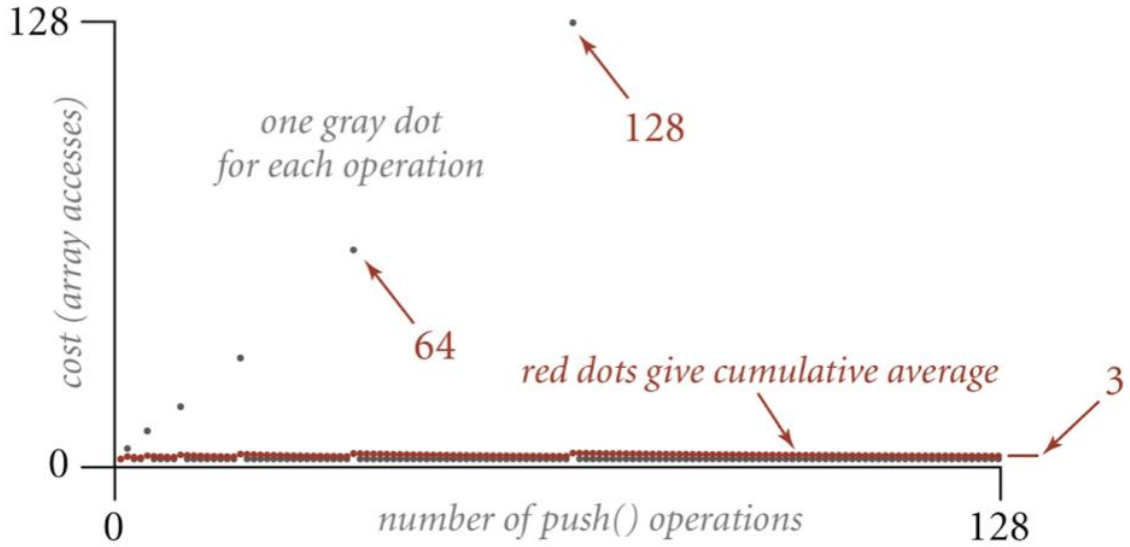
$$= N + 2N - 2$$

$$= 3N - 2$$

if $T(N) \approx 3N$, then $\frac{T(N)}{N} \approx 3 \rightarrow O(1)$

Amortized Analysis for *push:* why the resizing strategy matters

- Notice how the dots for when the array is resized get further apart as N gets larger.
- This would not be the case if we resized by adding a constant amount to the array size.



Amortized cost of push where the cost model is the number of array accesses and we resize by adding 100 to the array capacity.

total # of array accesses =

$$N + 2(1 + 101 + 201 + \dots + N - 100)$$

$$= N + 2\sum_{k=0}^{\frac{N-101}{100}} (100k + 1) = N + 200\left[\frac{\left(\frac{N-101}{100}\right)\left(\frac{N-101}{100} + 1\right)}{2}\right] + 2\left[\frac{N-101}{100} + 1\right]$$

$\Theta(N^2)$

If $T(N)$ is $\Theta(N^2)$,

then $\frac{T(N)}{N}$ is $\Theta(N)$.

Summary of Analysis for array-based Stacks and Queues

Operation	Best Case	Worst Case	Amortized
push/enqueue	O(1)	O(N)	O(1)
pop/dequeue	O(1)	O(N)	O(1)
isEmpty	O(1)	O(1)	O(1)
peek	O(1)	O(1)	O(1)
size	O(1)	O(1)	O(1)

What's better: implementing a Stack with an **array** or a **linked list**?

References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)