VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

Programming for Engineers

Lecture 13: Abstract Data – Queues and Stacks

Course ID: EE057IU

Outline

- Queues
- Stacks
- Hash Table

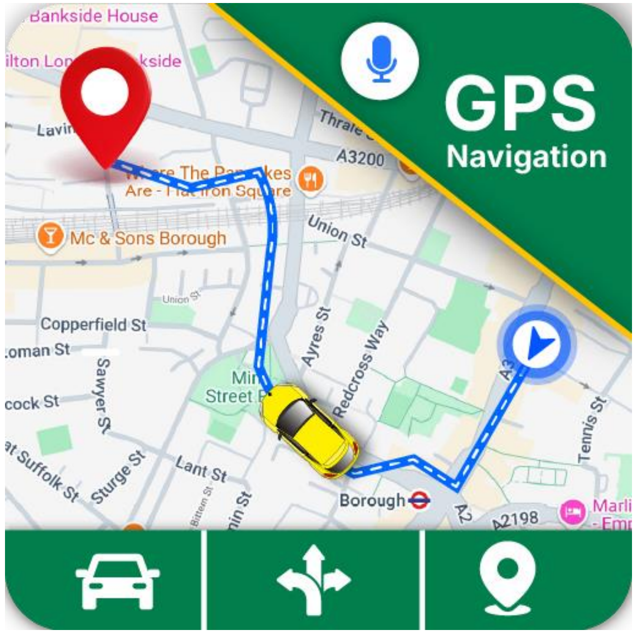
Abstract Data Type (ADT)

What is ADT?

The general ideal: an ADT is like a contract or a user manual. It tells you what the data does, not how it work inside.

An analogy: The GPS on Google Map

- **User-view (ADT):** You see your position, your current moving direction. You know if you press “The car” icon/button, you get your current latitude and longitude. You do not car how the GPS signal works.
- **Inside view (implementation):** The signals, hardwares, softwares inside your smart phone.



ADT vs Data Structure

ADT

- Tell us **WHAT** operations we can do
- Example: “I need a List that I can add items to”

Data Structure

- Tell us **HOW** the computure stores the data
- Example: “I will build that list using an Array, some pointers or just a Linked List”

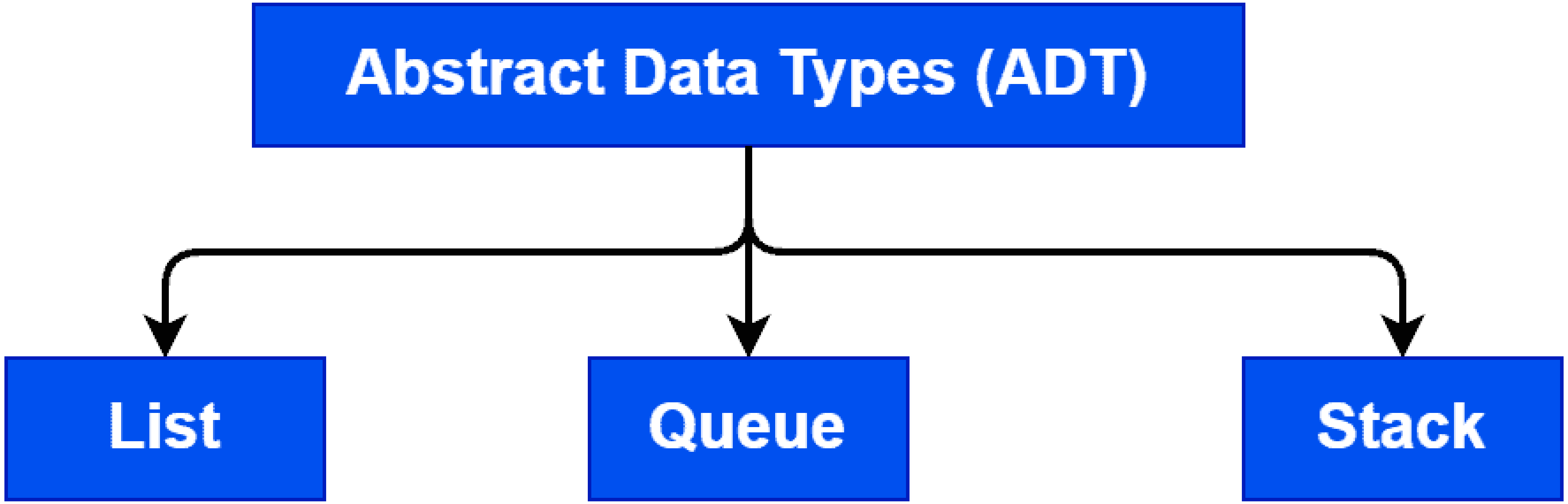
Why do we need ADTs

- **Modularity**
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Can be defined by:
    - **specification** (*Locality* or *Modifiability*)
    - **Parameterization**
- **Procedure abstraction**
  - Separates the purpose and use of a module from its implementation
  - A module's specifications should
    - Detail how the module behaves
    - Identify details that can be hidden within the module
- **Information hiding**
  - Hides certain implementation details within a module
  - Makes these details inaccessible from outside the module

Example ADTs: The List

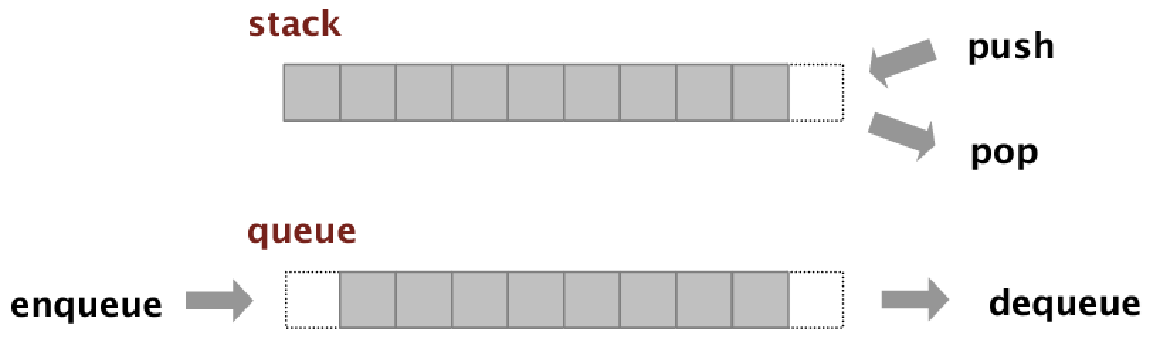
- ADT
  - Data: a collection of item (10,20,30…)
  - **Operations:**
    - `insert(item):` to add something
    - `delete(item):` to remove something
    - `isEmpty():` check if this List empty?
- The implementation
  - Option A: Use a fixed array (fast access but fixed size)
  - Option B: Linked List (dynamic size but slower to access)

Types of ADTs



Stacks and queues

- Fundamental Abtract Data Type
  - Operations: insert, remove, iterate, test if empty.
  - Intent is clear when we insert.
  - Which item do we remove?



Stack. Examine the item most recently added. ⟵ LIFO = "last in first out"

Queue. Examine the item least recently added. ⟵ FIFO = "first in first out"

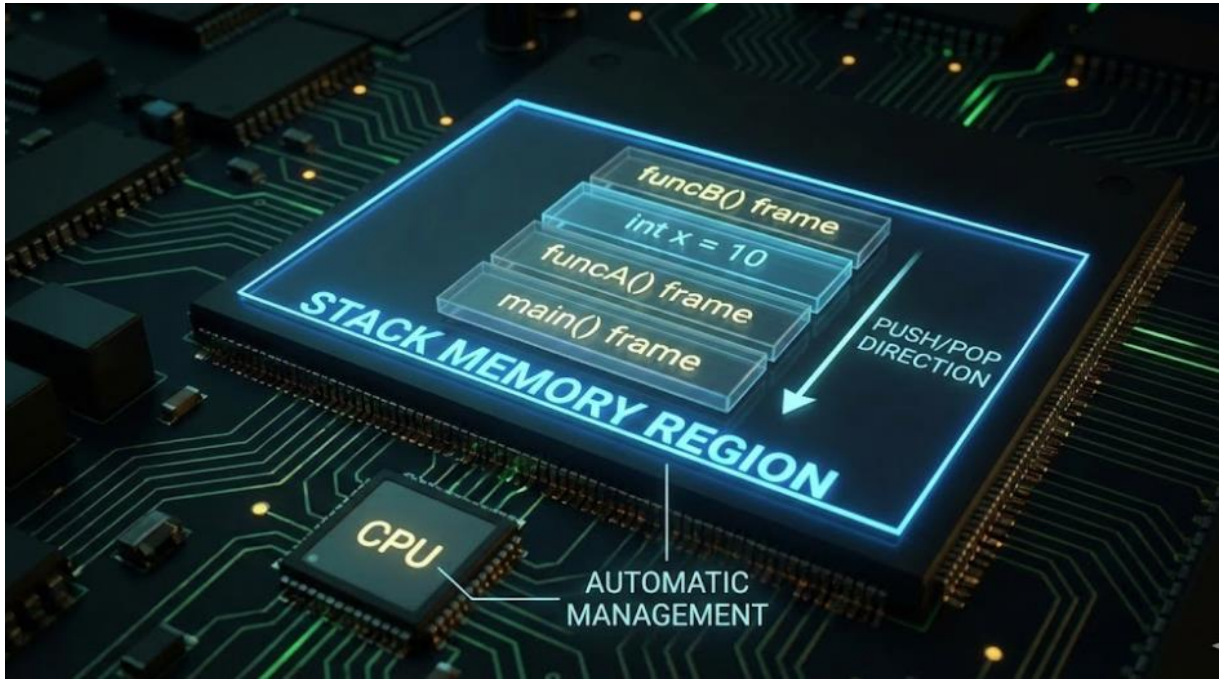
Stacks and queues

STACK:

- is a specific region of your computer's RAM reserved for temporary variable storage.
- Managed automatically.
- It can grow (from high to low memory address).
- Has a limit. If you exceed it, you get Stack Overflow

STACK Frame:

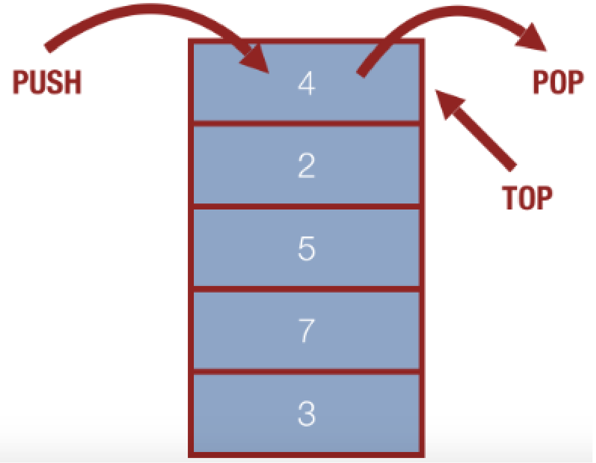
- is a block of memory.
- When a function is called, a block of memory is pushed on to the stack.
- Typically contains: local variables, arguments, return address



Stacks

- A *stack* is an abstract data type with the following behaviors / functions:
  - **push(value)** - add an element onto the top of the stack
  - **pop()** - remove the element from the top of the stack and return it
  - **peek()** - look at the element at the top of the stack, but don’t remove it
  - **isEmpty()** - a boolean value, true if the stack is empty, false if it has at least one element. (note: a runtime error occurs if a pop() or peek() operation is attempted on an empty stack.)

Why do we call it a stack? Because we model it using a stack of things:



The **push**, **pop**, and **peek** operations are the only element operations allowed by the stack ADT, and as such, only the top element is accessible. Therefore, a stack is a Last-In-First-Out (LIFO) structure: the last item in is the first one out of a stack.

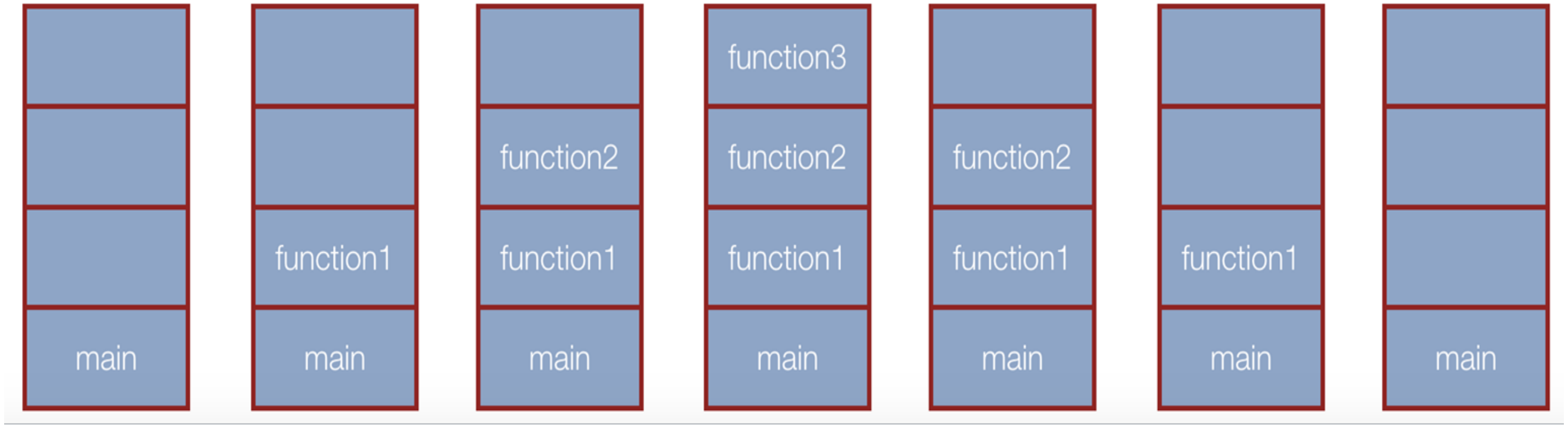
Stacks

- Despite the stack's limitations, the stack is a very frequently used ADT
- Stack operations are so useful that there is a stack built into every program running on your computer

```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

```
function2() {
    function3();
    return;
}
```

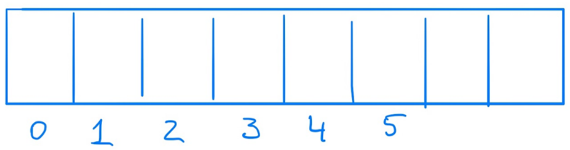


This is a LIFO pattern!

Stack Implementation Using an Array

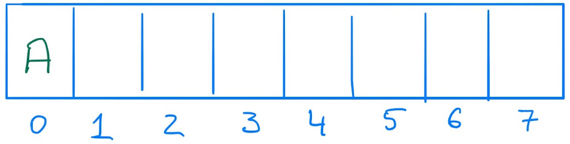
- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

capacity = 8
size = 0



Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

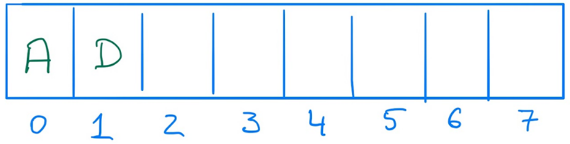


push (A)

capacity = 8
size = 1

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.



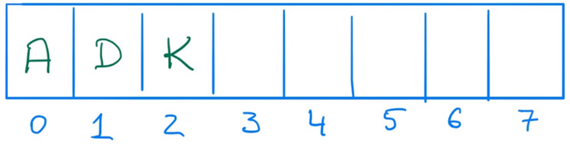
push (A)
push (D)

capacity = 8
size = 2

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

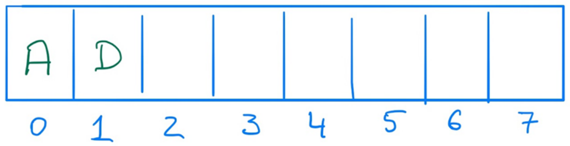
capacity = 8
size = 3



push (A)
push (D)
push (K)

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.



push (A)
push (D)
push (K)
pop ( )

capacity = 8
size = 2

The top element is at index 2, so that is the element removed when *pop* is called.

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

A	D	P	L	M			
0	1	2	3	4	5	6	7

push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)

capacity = 8
size = 5

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

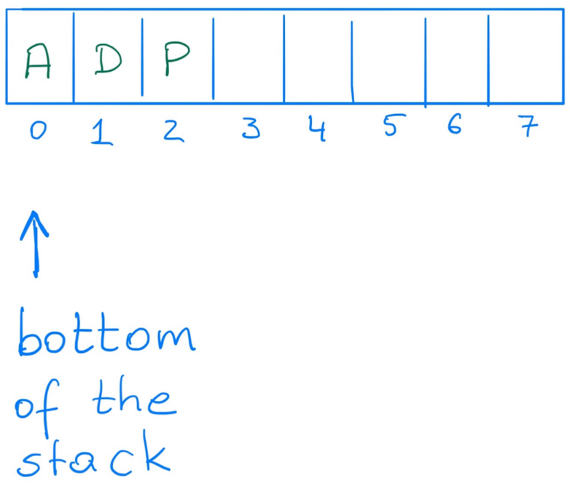
A	D	P					
0	1	2	3	4	5	6	7

push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ()
pop()

capacity = 8
size = 3

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

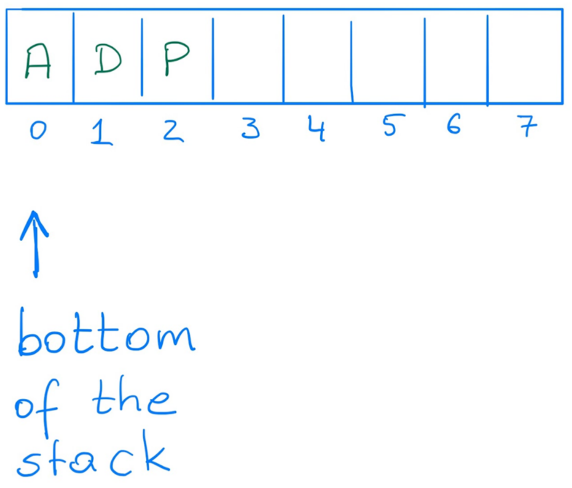


push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ( )
pop ( )

- **bottom of the stack** is always at index 0

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

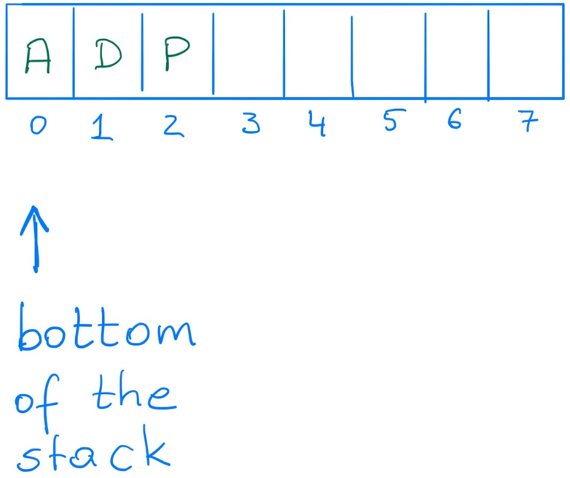


push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ()
pop()

- **bottom of the stack** is always at index 0
- **top of the stack** moves as we push and pop elements

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

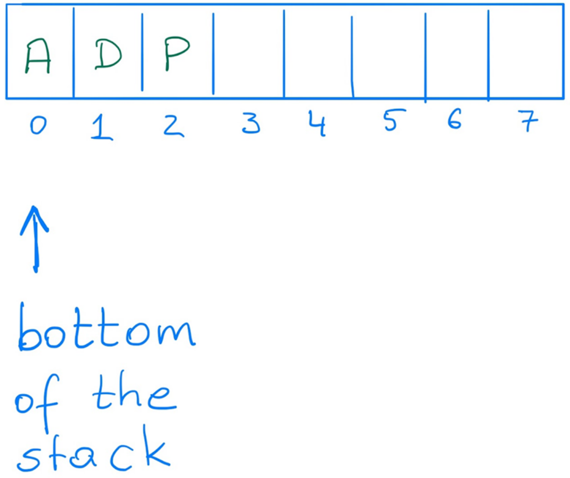


push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ()
pop()

- **bottom of the stack** is always at index 0
- **top of the stack** moves as we push and pop elements
- how do we know the index at which the next element should be pushed/added?

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.



push(A)
push(D)
push(K)
pop()
push(P)
push(L)
push(M)
pop()
pop()

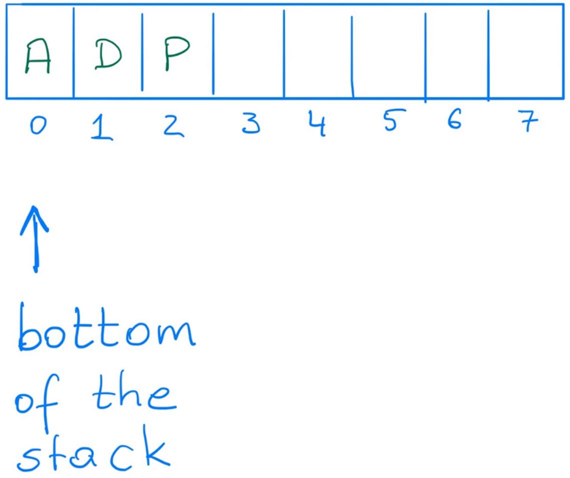
- **bottom of the stack** is always at index 0
- **top of the stack** moves as we push and pop elements
- how do we know the index at which the next element should be pushed/added? **the index of the first *empty space* is `size`**

$$\text{Index of next empty space} = \text{Current Number of Elements}$$

the index of the *top* is `size - 1`

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.

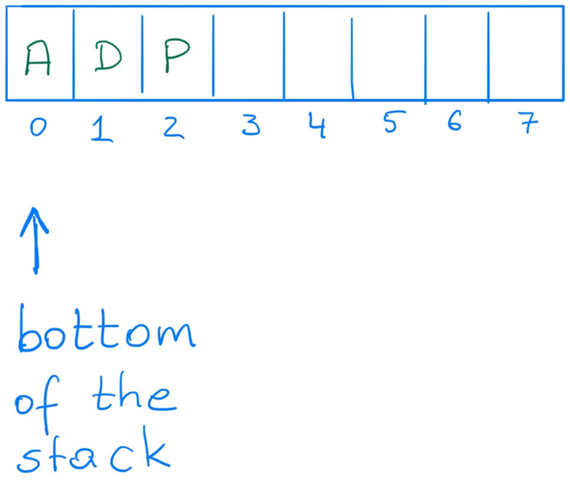


push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ()
pop()

- **bottom of the stack** is always at index 0
- **top of the stack** moves as we push and pop elements
- how do we know the index at which the next element should be pushed/added? **the index of the first *empty space* is `size`**
- how do we know the index from which the element should be popped/removed?

Stack Implementation Using an Array

- One way to implement a stack is to use the array as the underlying storage.
- The decision that needs to be made is **where should the top and bottom of the stack be**.



push (A)
push (D)
push (K)
pop ( )
push (P)
push (L)
push (M)
pop ()
pop()

- **bottom of the stack** is always at index 0
- **top of the stack** moves as we push and pop elements
- how do we know the index at which the next element should be pushed/added? **the index of the first *empty space* is `size`**
- how do we know the index from which the element should be popped/removed? **the index of the *top* is `size - 1`**

Queues

➢ The next ADT we are going to talk about is a *queue.* A queue is similar to a stack, except that (much like a real queue/line), it follows a "First-In-First-Out" (FIFO) model:

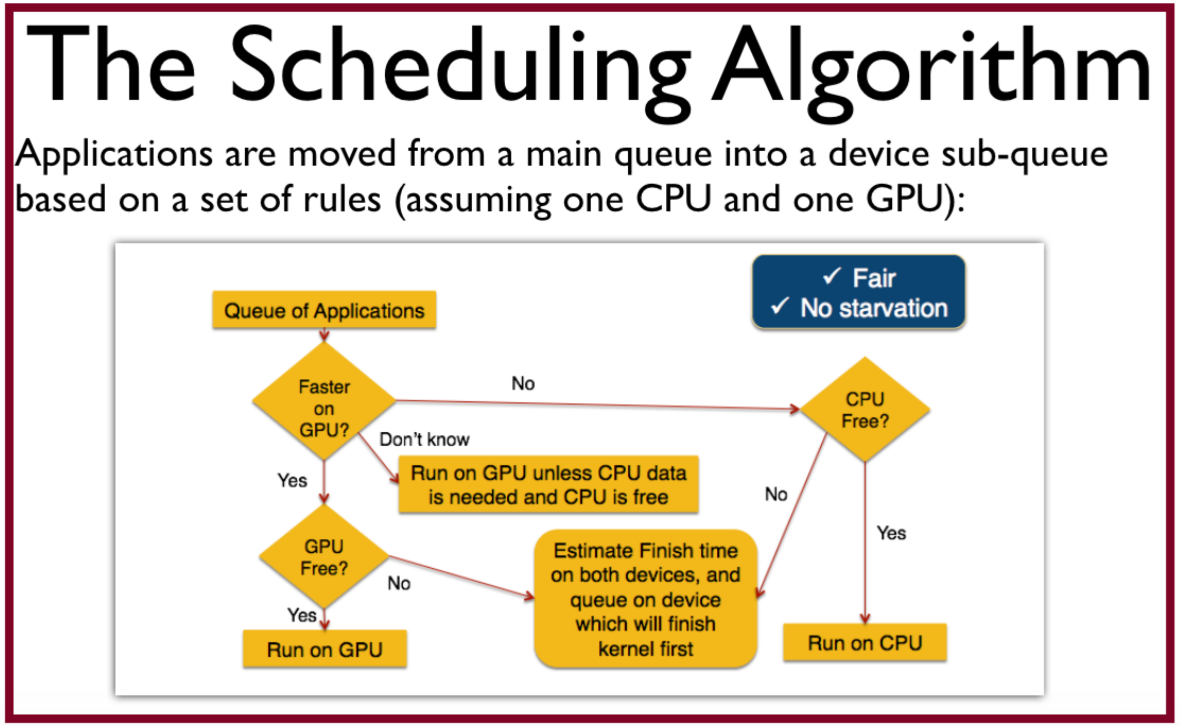
- The first person in line is the first person served.
- The last person in line is the last person served.
- Insertion into a queue **enqueue()** is done at the *back* of the queue, and removal from a queue **dequeue()** is done at the *front* of the queue.

Queues

- Like the stack, the queue Abstract Data Type can be implemented in many ways (we will talk about some later!). A queue must implement at least the following functions:
  - **enqueue(value)** - add an element onto the back of the queue
  - **dequeue()** - remove the element from the front of the queue and return it
  - **peek()** - look at the element at the front of the queue, but don’t remove it
  - **isEmpty()** - a boolean value, true if the queue is empty, false if it has at least one element. (note: a runtime error occurs if a dequeue() or peek() operation is attempted on an empty queue).

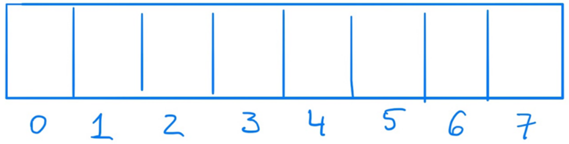
Queues Examples

- **There are many real world problems that are modeled well with a queue:**
  - Jobs submitted to a printer go into a queue (although they can be deleted, so it breaks the model a bit)
  - Ticket counters, supermarkets, etc.
  - File server - files are doled out on a first-come-first served basis
  - Call centers (“your call will be handled by the next available agent”)
  - Scheduling work between a CPU and a GPU is queue based.



Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.

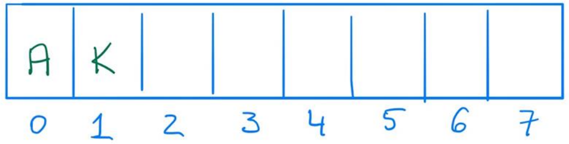


capacity = 8
size = 0
front = 0
back = 0

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.

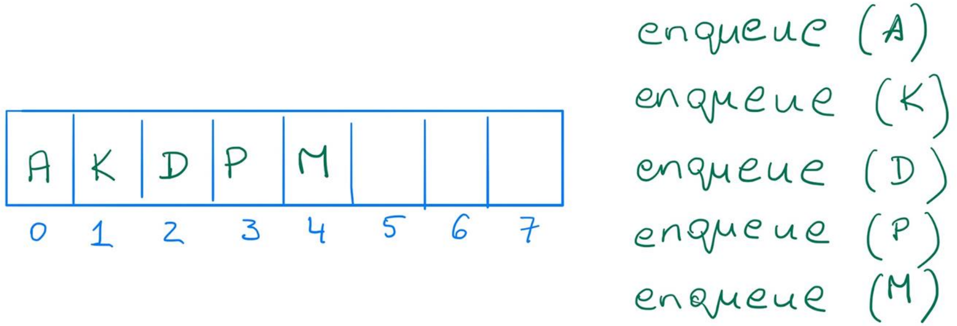
enqueue (A)
enqueue (K)



capacity = 8
size = 2
front = 0
back = 2

Queue Implementation Using an Array

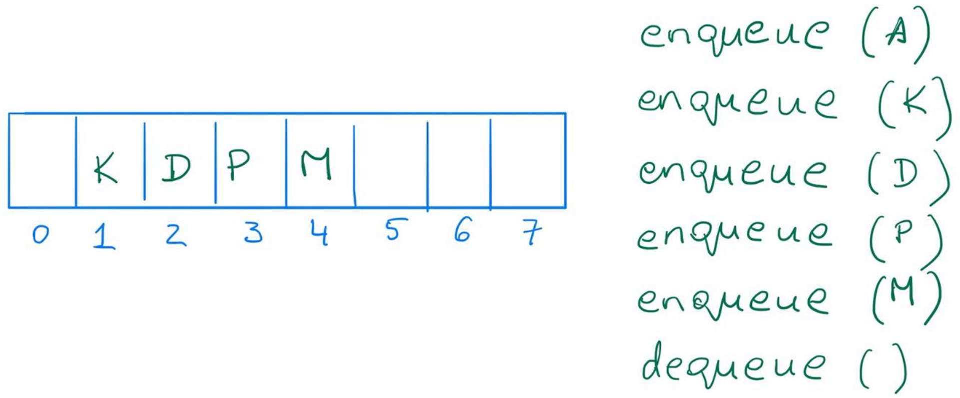
- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



capacity = 8
size = 5
front = 0
back = 5

Queue Implementation Using an Array

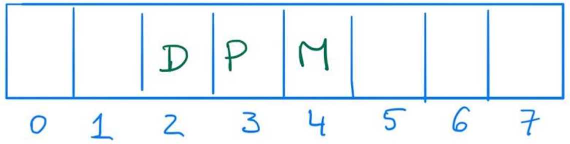
- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



capacity = 8
size = 4
front = 1
back = 5

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.

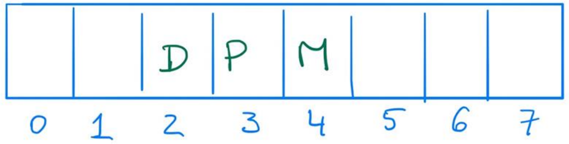


enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )

capacity = 8
size = 3
front = 2
back = 5

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



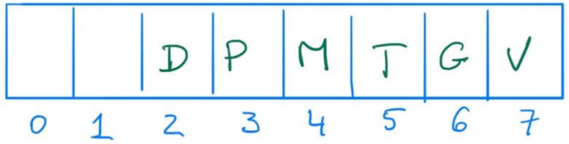
enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )

capacity = 8
size = 3
front = 2
back = 5

front - index of the first element (size > 0)
back - index at which the next element should be added
size = back - front

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



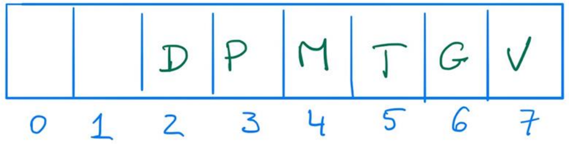
enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )
enqueue (T)
enqueue (G)
enqueue (V)

capacity = 8
size = 6
front = 2
back = ??? 8??? -> Pointer has "drifted" off the end of the array

front - index of the first element (size > 0)
back - index at which the next element should be added
size = back - front

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.

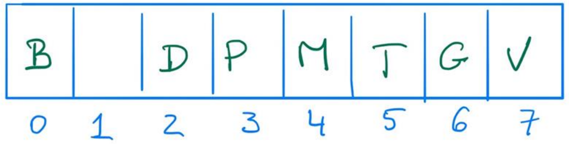


enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )
enqueue (T)
enqueue (G)
enqueue (V)
? enqueue (B)

capacity = 8
size = 6+1
front = 2
back = ???

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



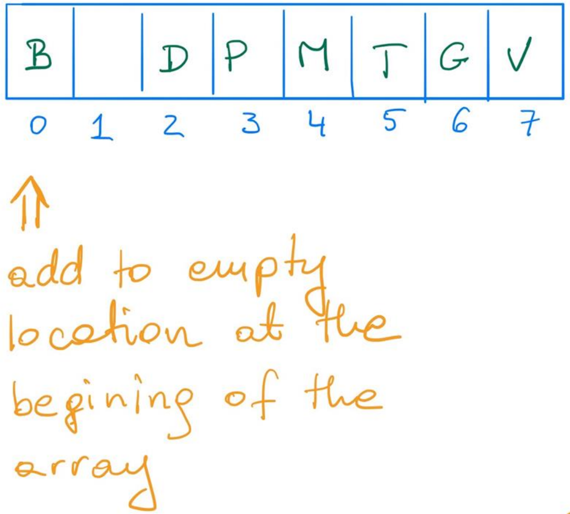
add to empty location at the begining of the array

enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )
enqueue (T)
enqueue (G)
enqueue (V)
? enqueue (B)

capacity = 8
size = 7
front = 2
back = 1

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



enqueue (A)
enqueue (K)
enqueue (D)
enqueue (P)
enqueue (M)
dequeue ( )
dequeue ( )
enqueue (T)
enqueue (G)
enqueue (V)
? enqueue (B)

capacity = 8
size = 7
front = 2
back = 1

size = (back - front) % capacity
(when size < capacity)

We can use the array as a **circular array**: where there are empty locations available in the front of the array, we wrap the values back to the start of the array instead of allocating a new one.

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.

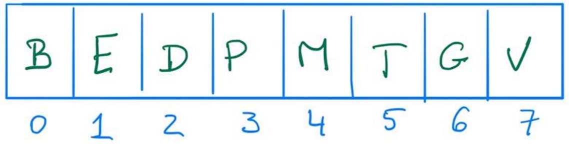
B	E	D	P	M	T	G	V
0	1	2	3	4	5	6	7

enqueue (A)
. . .
enqueue (B)
enqueue (E)

capacity = 8
size = 8
front = 2
back = 2

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



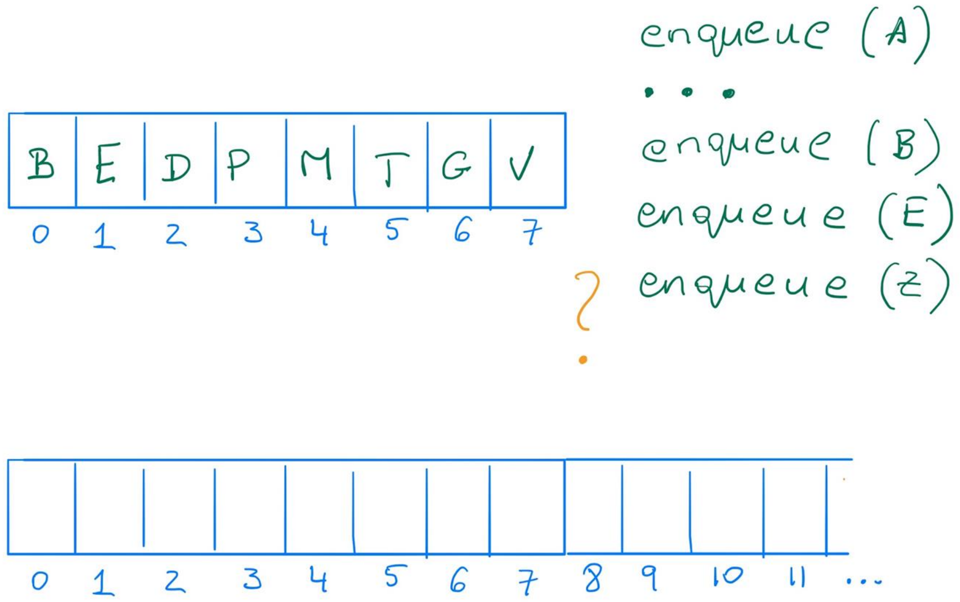
enqueue (A)
. . .
enqueue (B)
enqueue (E)
? enqueue (Z)

capacity = 8
size = 8
front = 2
back = 2

size == capacity
so we need a larger array

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



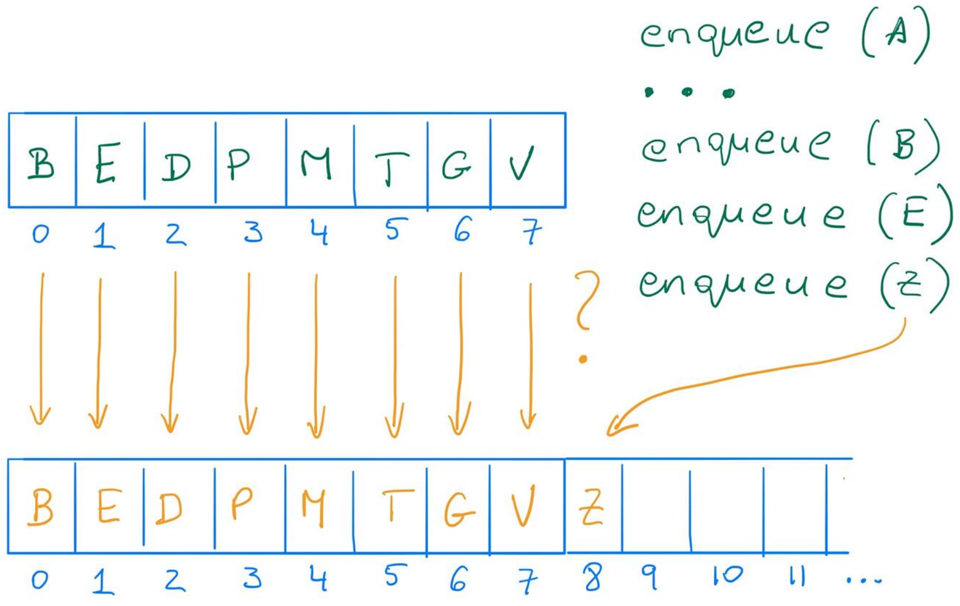
capacity = 8
size = 8
front = 2
back = 2

size == capacity
so we need a larger array

- create a new larger array
- copy of the values from the old array to the new one
- place the new value at the end

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



Will this work?

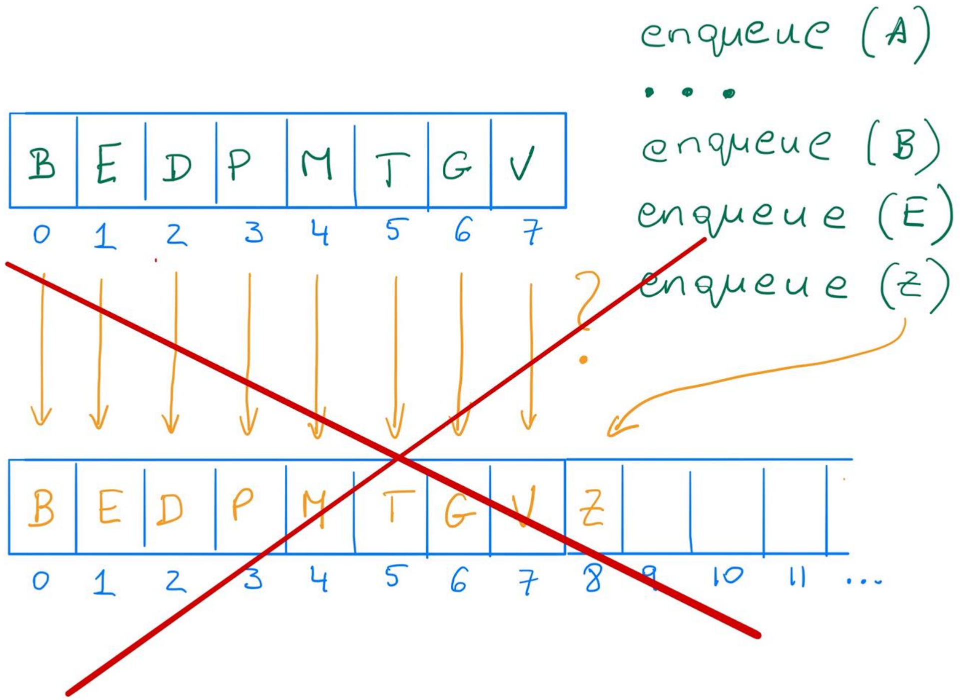
capacity = 8
size = 8
front = 2
back = 2

size == capacity
so we need a larger array

- create a new larger array
- copy of the values from the old array to the new one
- place the new value at the end

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



Will this work?
NO

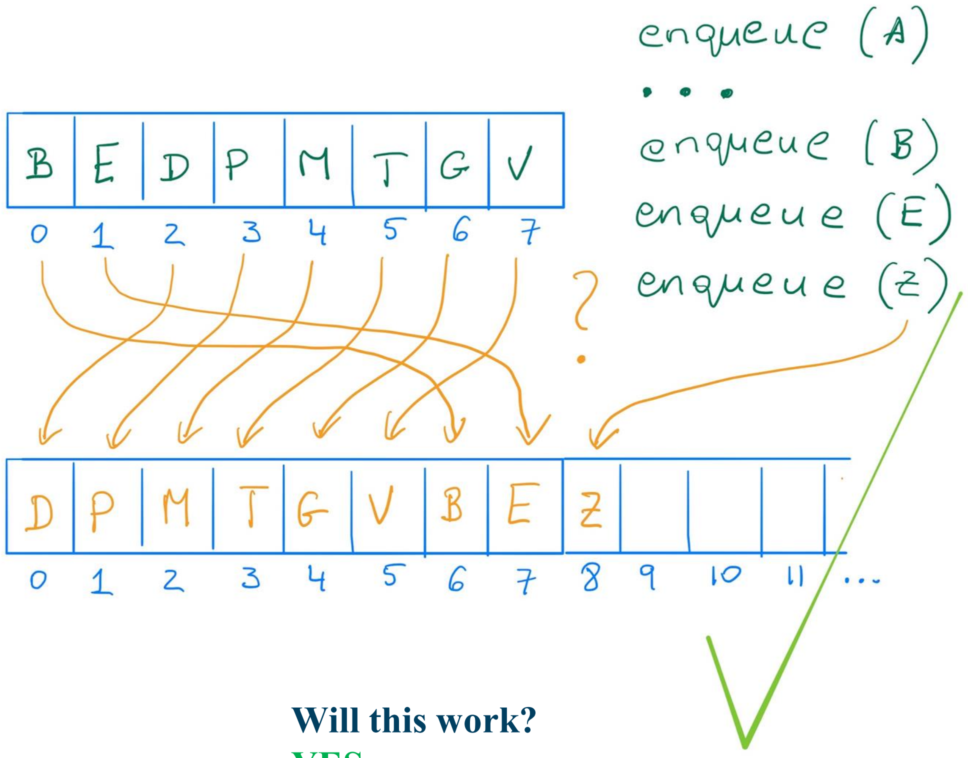
capacity = 8
size = 8
front = 2
back = 2

size == capacity
so we need a larger array

- create a new larger array
- copy of the values from the old array to the new one
- place the new value at the end

Queue Implementation Using an Array

- One way to implement a queue is to use the array as the underlying storage.
- The decision that needs to be made is **where should the front and the end of the queue be**.



Will this work?
YES

capacity = 8
size = 8
front = 2
back = 2

- create a new larger array
- Copy the elements starting from *front* (index 2) to the end of the old array (D through V)
- Copy the wrapped-around elemens (B and E) and place them after V
- Reset *front* to 0

Queue example

```c
#include <stdio.h>
#include <stdbool.h>
#define CAPACITY 5  // Maximum size of our queue
// Global variables for simplicity
int queue[CAPACITY];
int front = 0;      // Points to the first item
int back = 0;       // Points to the NEXT empty spot
int size = 0;       // Keeps track of how many items we have
```

```c
void enqueue(int value) {
    if (size == CAPACITY) {
        printf("ERROR: Queue is Full! Cannot add %d\n", value);
        return;
    }

    queue[back] = value;                // 1. Put value in the empty spot
    back = (back + 1) % CAPACITY;       // 2. Move 'back' (Wrap around if needed)
    size++;                             // 3. Increase size

    printf("Enqueued %d \t(Front: %d, Back: %d)\n", value, front, back);
}
```

```c
void dequeue() {
    if (size == 0) {
        printf("ERROR: Queue is Empty!\n");
        return;
    }

    int removed = queue[front];       // 1. Grab the item at the front
    queue[front] = 0;                 // (Optional) Clear it for visualization
    front = (front + 1) % CAPACITY;   // 2. Move 'front' (Wrap around if needed)
    size--;                           // 3. Decrease size

    printf("Dequeued %d \t(Front: %d, Back: %d)\n", removed, front, back);
}
```

Queue example

```c
int main() {
    printf("--- 1. Fill the Queue ---\n");
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50); // Queue is now FULL (Size 5)
    printArray();

    printf("--- 2. Try to add one more (Should fail) ---\n");
    enqueue(60);
    printf("\n");
```

Enqueue 2 items until the queue is full. Try to enqueue one more!

```c
    printf("--- 3. Dequeue 2 items ---\n");
    dequeue(); // Removes 10 (from index 0)
    dequeue(); // Removes 20 (from index 1)
    printArray();
```

Dequeue 2 items making index 0 and 1 empty.

```c
    printf("--- 4. Enqueue new items (WRAPS AROUND to index 0) ---\n");
    enqueue(88); // Should go to index 0!
    enqueue(99); // Should go to index 1!
    printArray();
```