ECE 220

Lecture x0003 - 01/23

Slides based on material by: Yuting Chen, Yih-Chun Hu & Ujjal Bhowmik

Recap

- Last lectures, we talked about
  - Keyboard/Display polling and handshaking
  - Subroutines & TRAP mechanism
  - TRAP's RTI uses a different mechanism than RET R7
  - The mechanism is called ***stack*** - an Abstract Data Type

Cover again at start of 01/23

- Reminders:
  - MP1 is due Thursday. Make use of office hours!

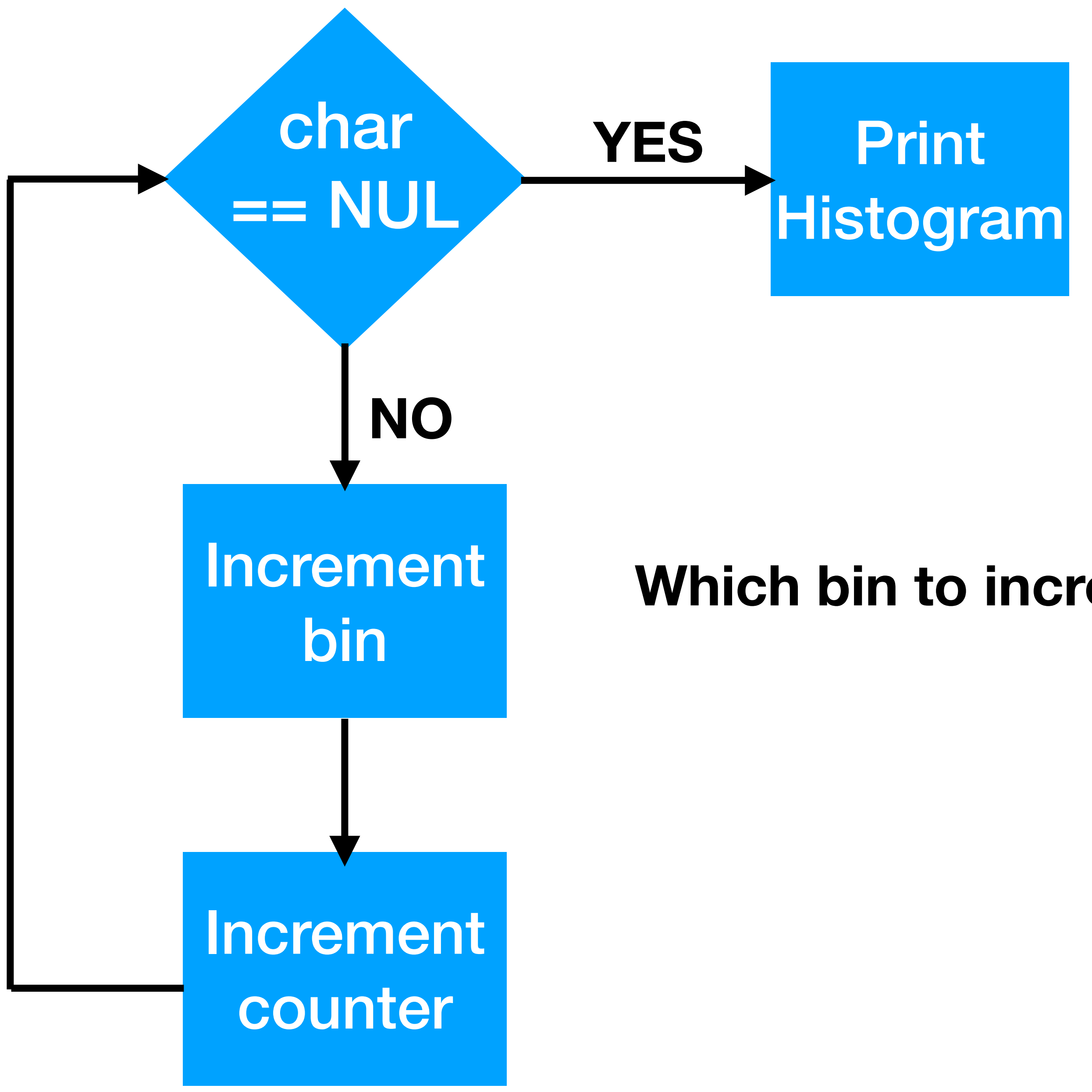
MP 1- Letter frequency decomposition

- Common practice in programming to decompose a task into smaller subtasks
  - What did we learn that can help us do this?
- The task:
  - Given an ASCII string (terminated by NUL)
  - Count the occurrences of each letter (regardless of case), and
  - The number of non-alphabetic characters, and
  - Print out a histogram

MP 1- Letter frequency decomposition

- Divide into two tasks
    - Counting a character
    - Printing histogram

Can only do this after checking entire string.
When is string done? → NUL

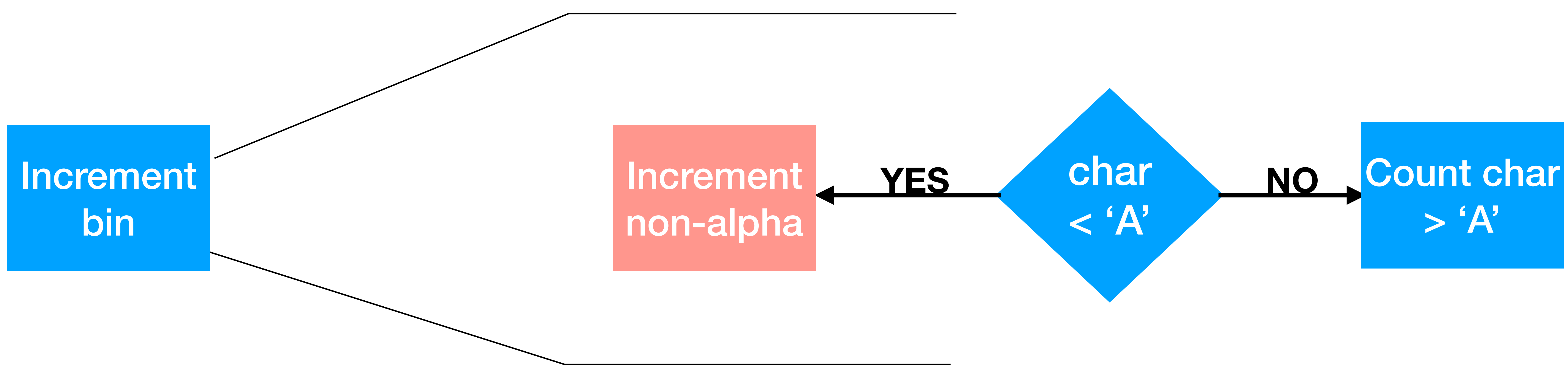


Which bin to increment?

MP 1- Letter frequency decomposition

- Which bin to increment?
  - Need to determine if character is alphabetic or non-alphabetic.

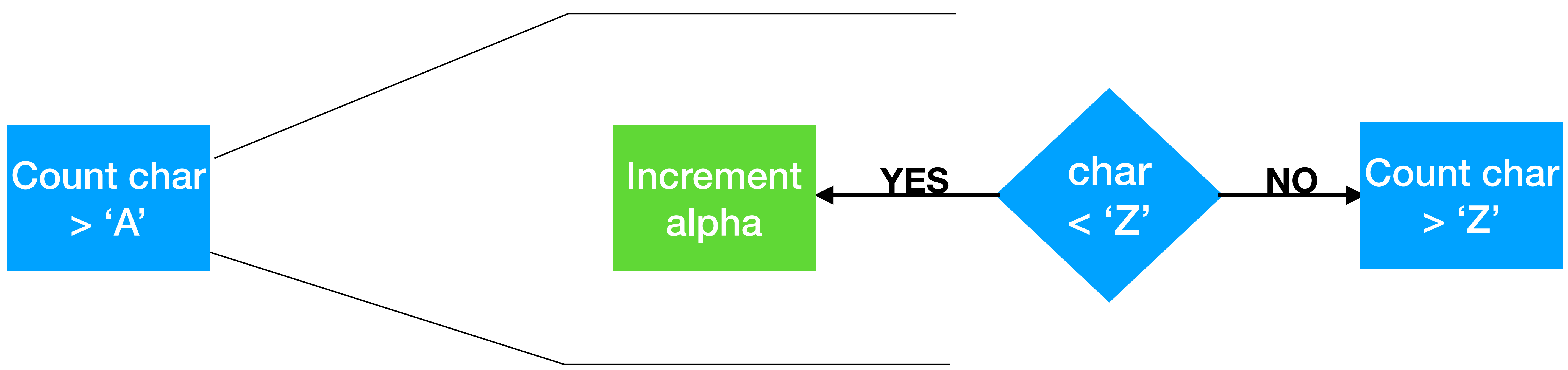
x00		x40	x41		x5A	x5B		x60	x61		x7A	x7B		x7F
NUL		@	A		Z	[		`	a		z	{		DEL



MP 1- Letter frequency decomposition

- Which bin to increment?
  - Need to determine if character is alphabetic or non-alphabetic.

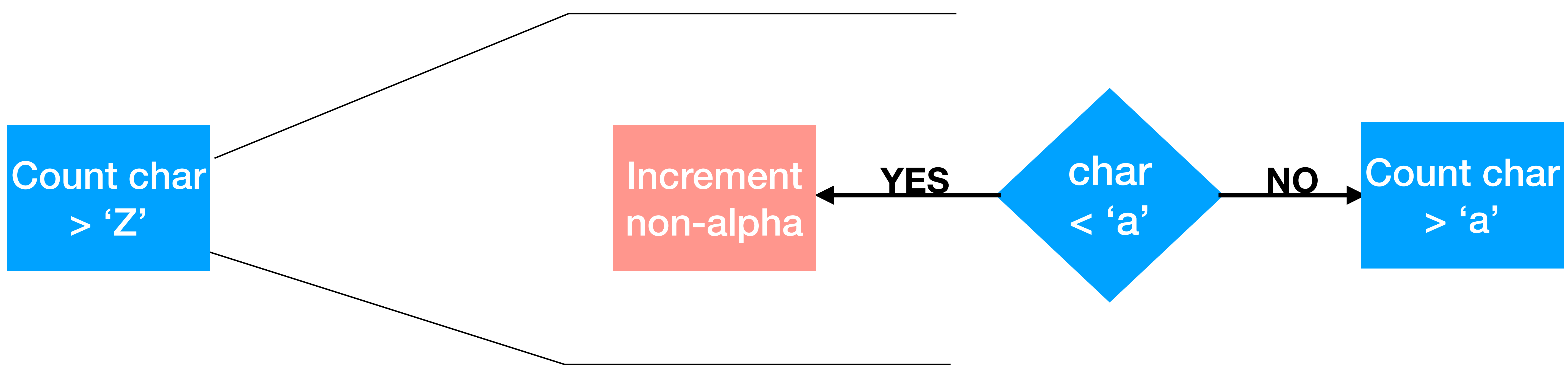
x00		x40	x41		x5A	x5B		x60	x61		x7A	x7B		x7F
NUL		@	A		Z	[		`	a		z	{		DEL



MP 1- Letter frequency decomposition

- Which bin to increment?
  - Need to determine if character is alphabetic or non-alphabetic.

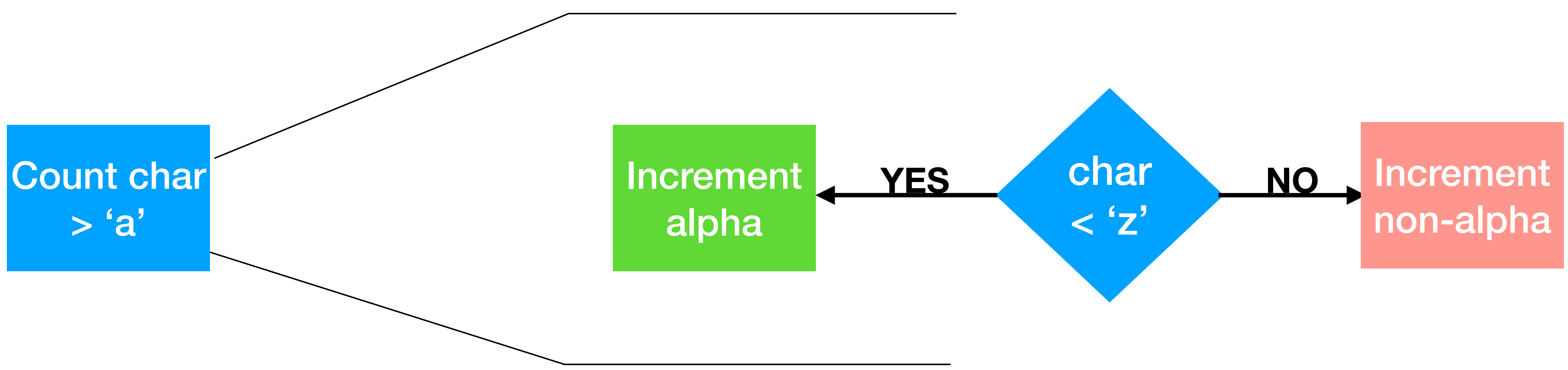
x00		x40	x41		x5A	x5B		x60	x61		x7A	x7B		x7F
NUL		@	A		Z	[		`	a		z	{		DEL



MP 1- Letter frequency decomposition

- Which bin to increment?
  - Need to determine if character is alphabetic or non-alphabetic.

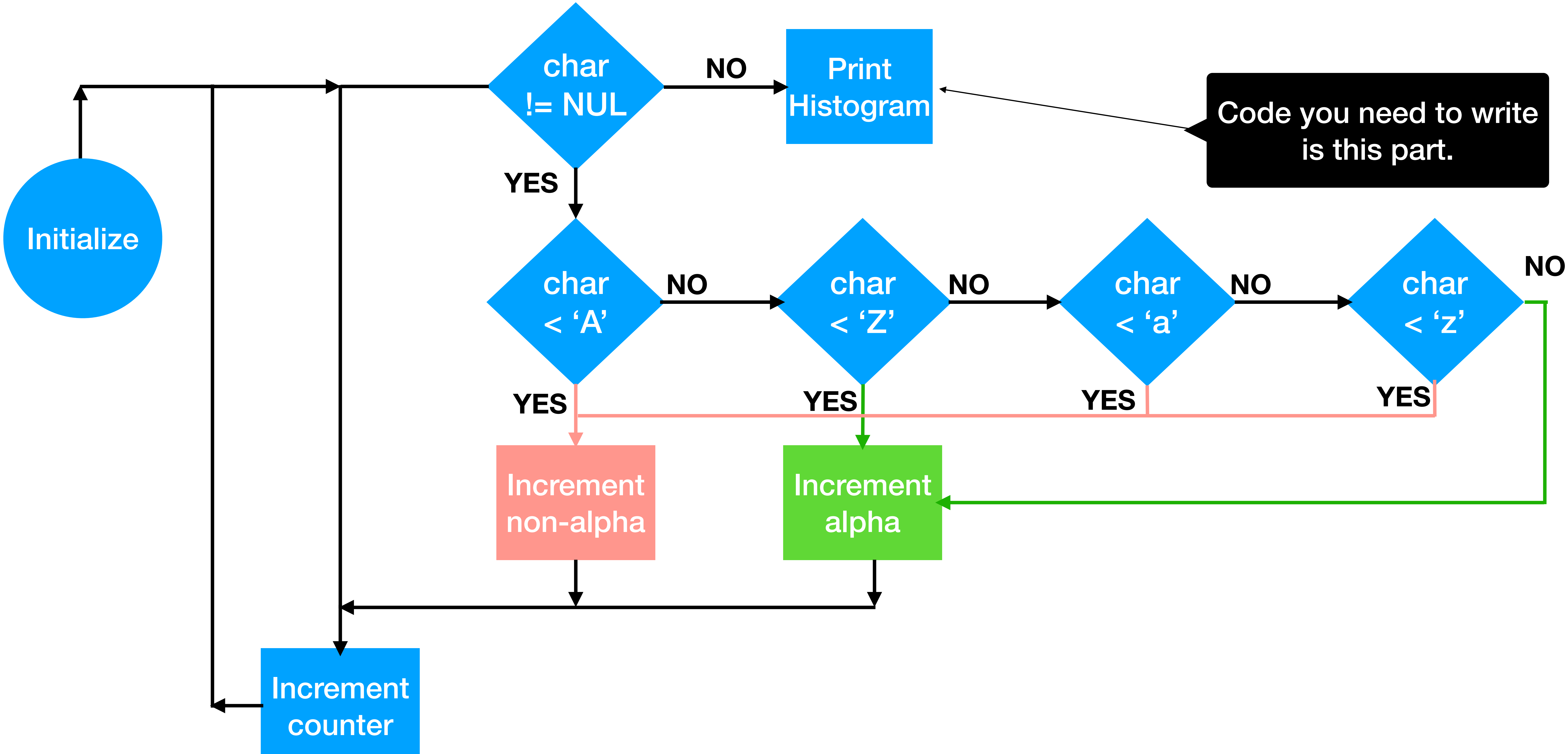
x00		x40	x41		x5A	x5B		x60	x61		x7A	x7B		x7F
NUL		@	A		Z	[		`	a		z	{		DEL



MP 1- Letter frequency decomposition

- What about initialization etc? We need to do three things:
  - fill the histogram with 0s,
  - load any useful values (such as ASCII characters to check the region boundaries)
  - and point to the start of the string
- How to increment alpha → see MP (code already provided)

MP 1- Letter frequency decomposition



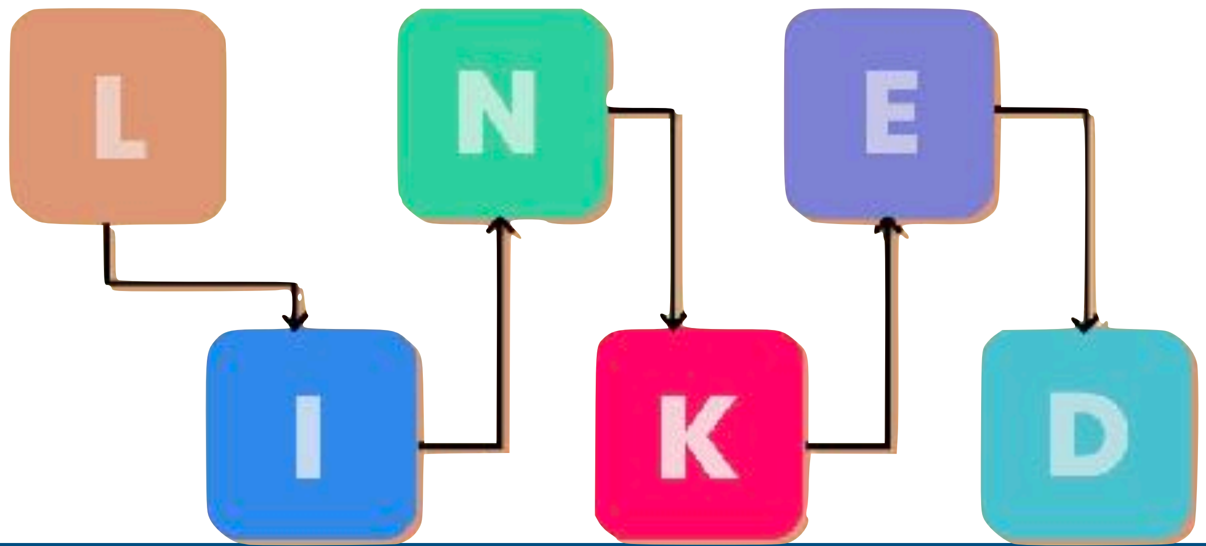
Abstract Data Types

- Abstract Data Type (ADT) refers to a model for a data type that combines the logical description of how data is viewed and the operations that are allowed on it *without* regard to how they will be implemented.

  - *Example: Integers as an ADT are zero, the natural numbers and their additive inverses with the usual operations of addition, multiplication, subtraction, etc.* ***However****, on a computer they may be implemented as 2's complements, IEEE 754, etc.*

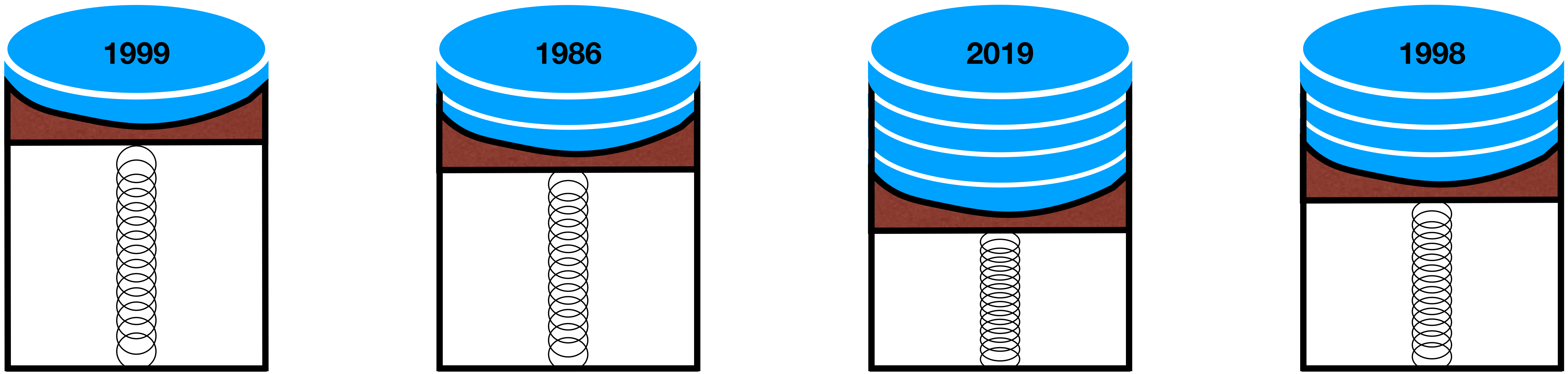
Other ADTs

- Some other Abstract Data Types
  - Queues (example of FIFO: First-In-First-Out)
  - Linked lists
  - Trees
  - Dictionaries



Stack ADT

- Two main operations
  - **PUSH:** add an item to the stack
  - **POP:** remove an item from the stack



A single element **After a PUSH** **After two more PUSHes** **After a POP**

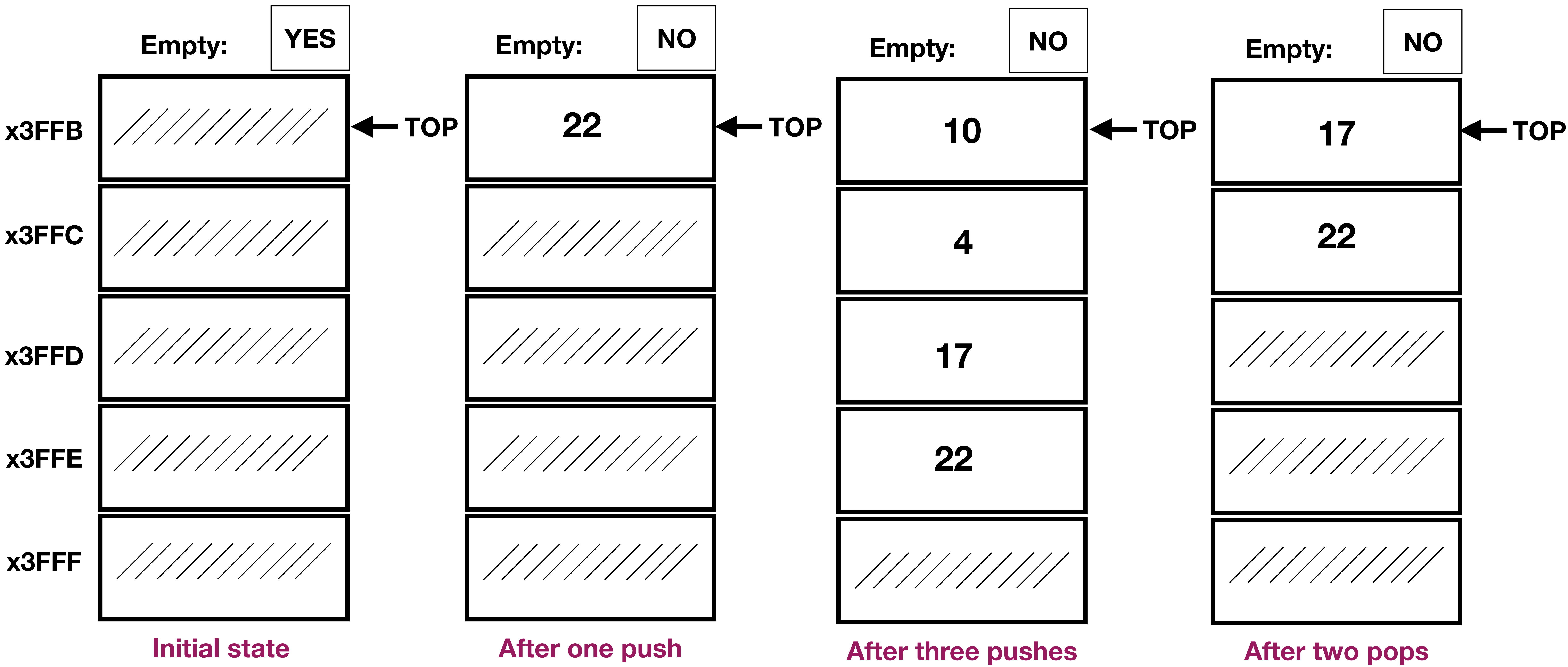
Stack

- It is a **LIFO** (Last-In-First-Out) storage structure
  - The first thing you put in is the last thing you take out
  - The last thing you put in is the first thing you take out

Together called stack protocol

- Main operations are: PUSH/POP
- Most implementations also offer:
  - **PEEK:** view top of the stack without popping an element
  - Methods to check if stack is **ISFULL** or **ISEMPTY**

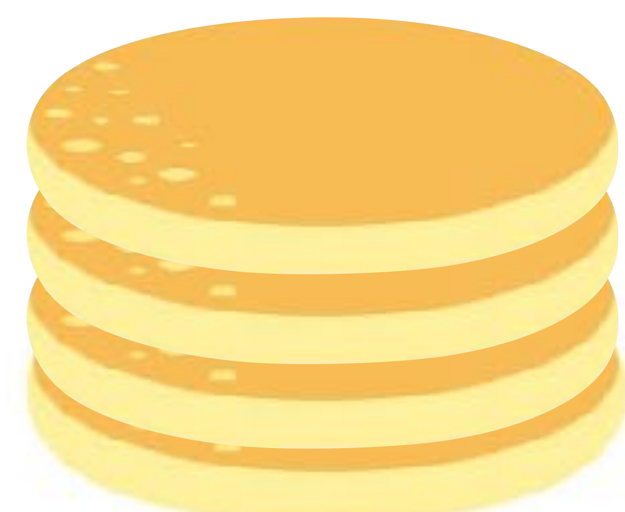
Naive implementation



Another look at a stack

First pancake

After one push
(Second pancake)



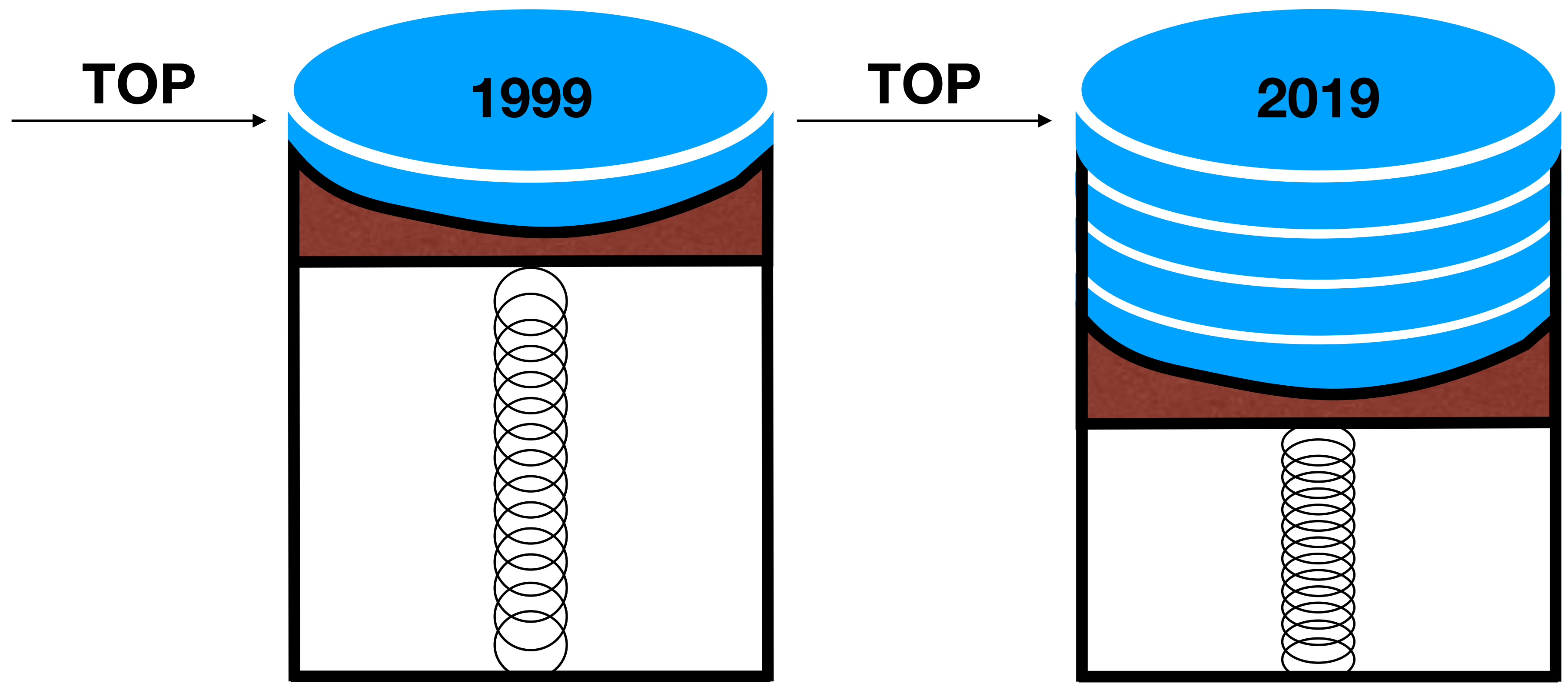
After two more
pushes

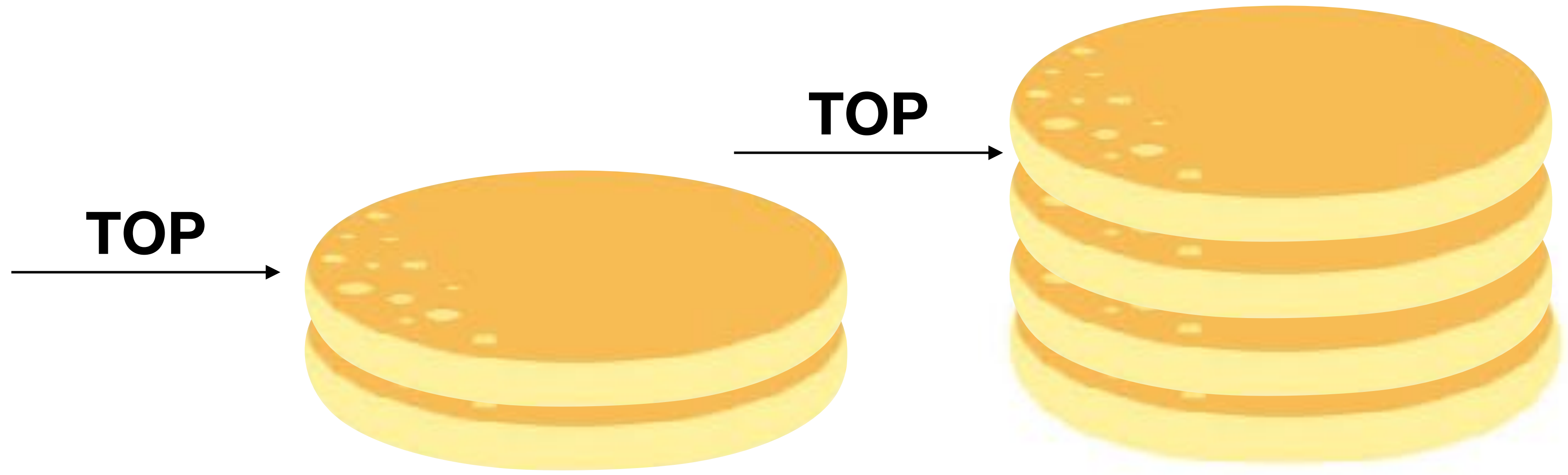
After two
pops

Stack

- What was the difference between the quarter version and the pancake version?

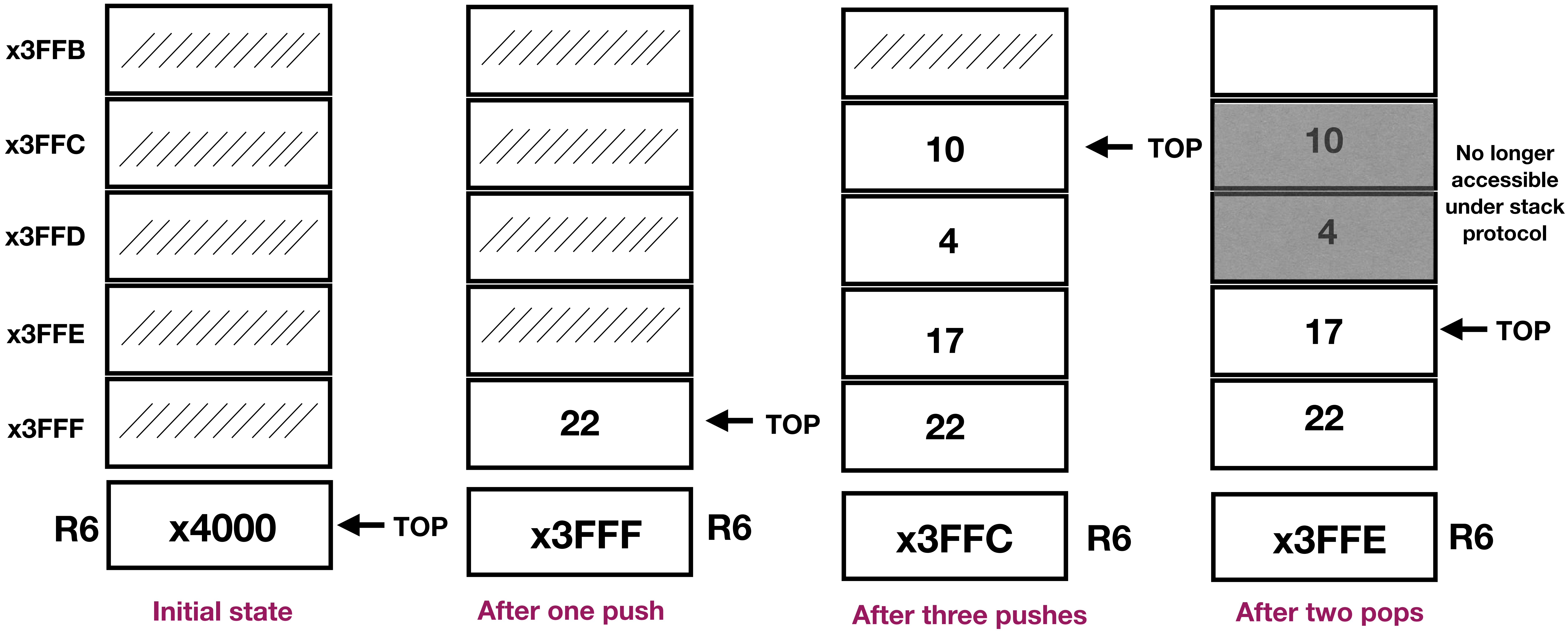
Pointer moved





Data moved

Software implementation



In this implementation, data **do not** move in memory.
By convention, **R6** holds the **top of stack** (TOS) pointer.

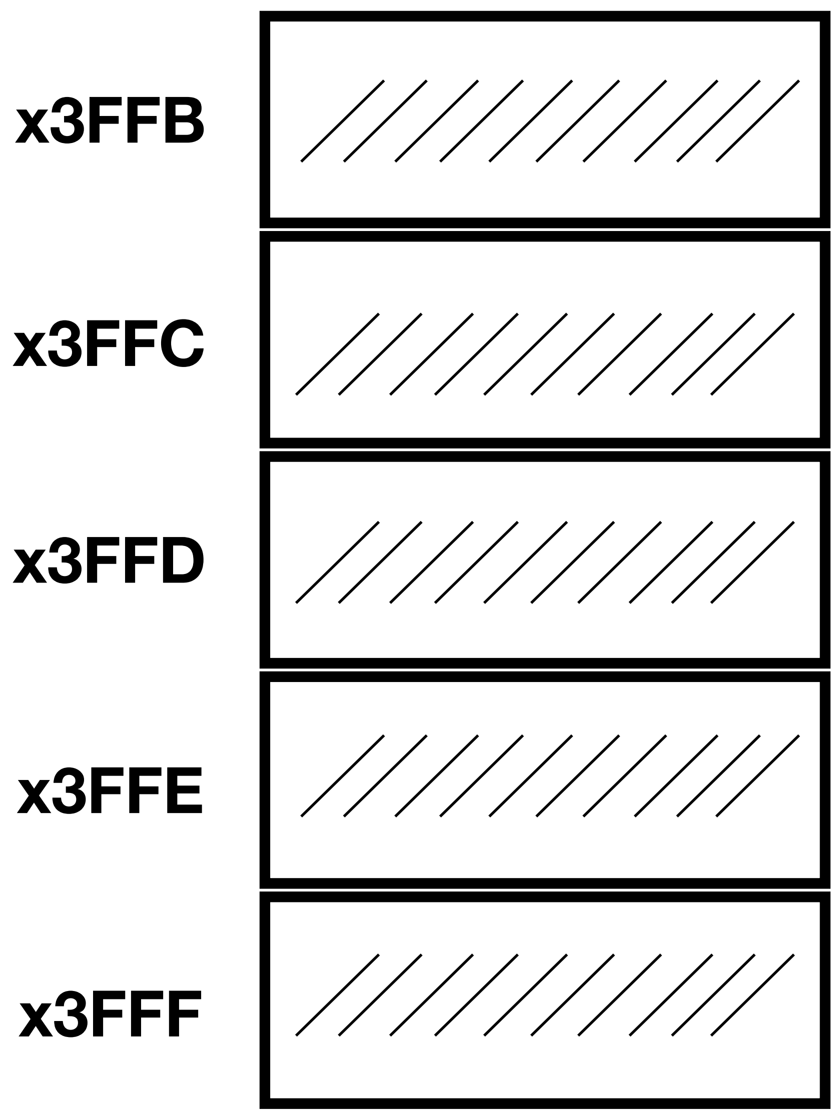
Stacks in LC3

- By convention in LC3, we will use R6 for TOP and R0 for priming pushes and completing pops.
- Basic PUSH code:

```
ADD R6, R6, #-1 ;decrement TOP
STR R0, R6, #0  ;store data
```

- Basic POP code:

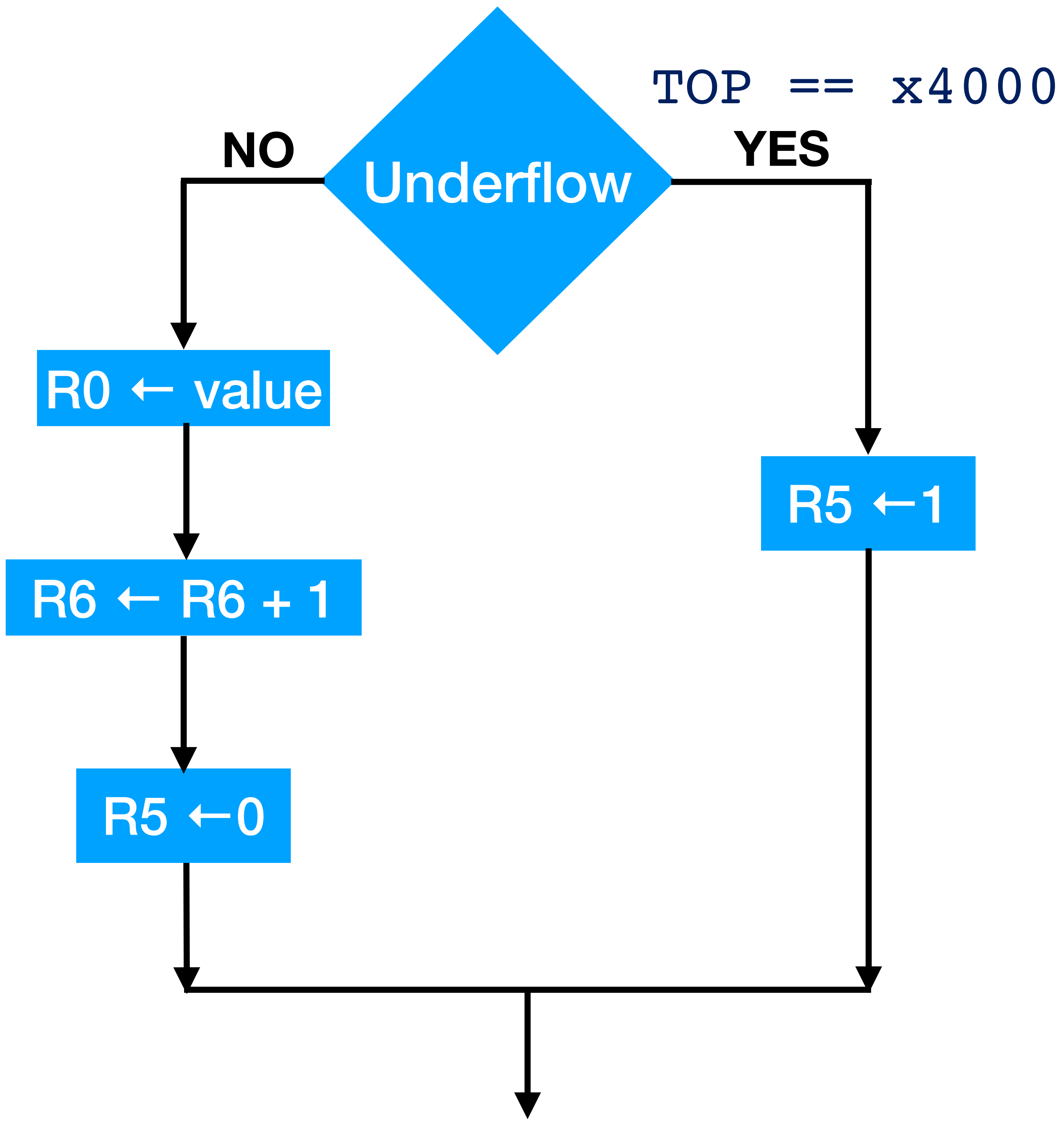
```
LDR R0, R6, #0 ;load data
ADD R6, R6, #1 ;increment TOP
```



Also by convention the stack “grows towards zero”.

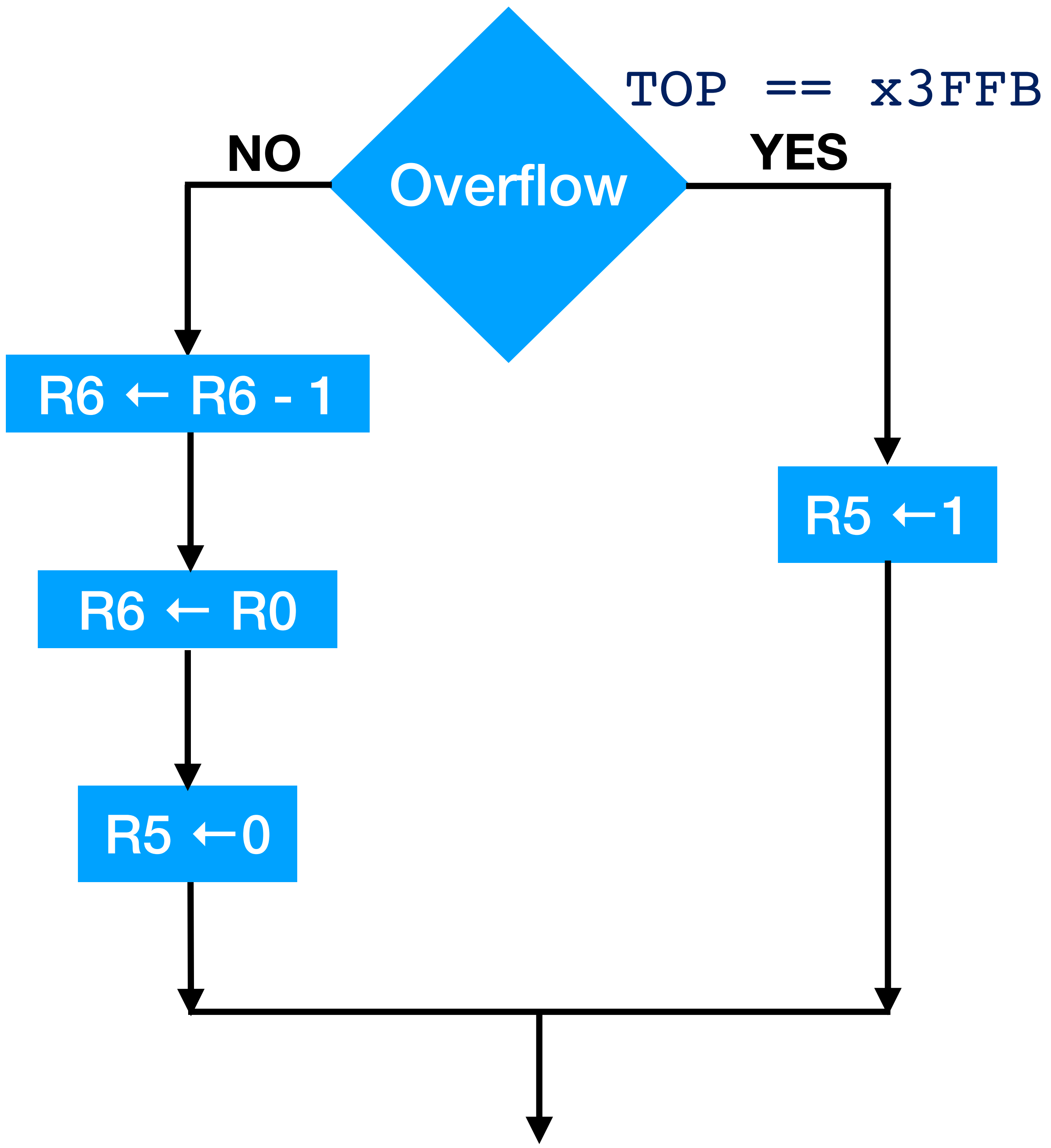
Stacks in LC3 - Pop

- What happens if stack is empty? Or full?
    - Need to detect *overflow* and *underflow.*
    - Use concept of *exit code.*
        - Use `R5` to indicate success (0) or failure (1) of operations.



Stacks in LC3 - Push

- What happens if stack is empty? Or full?
    - Need to detect *overflow* and *underflow.*
    - Use concept of *exit code.*
        - Use R5 to indicate success (0) or failure (1) of operations.



Stacks in LC3

POP Routine

```
POP       AND R5, R5, #0
          LD R1, EMPTY
          ADD R2, R6, R1
          BRz Failure
          LDR R0, R6, #0
          ADD R6, R6, #1
          RET
Failure   ADD R5,R5,#1
          RET
EMPTY     .FILL   xC000
;EMPTY <– -x4000
```

PUSH Routine

```
PUSH      AND R5, R5, #0
          LD R1, MAX
          ADD R2, R6, R1
          BRz Failure
          ADD R6, R6, #-1
          STR R0, R6, #0
          RET
Failure   ADD R5, R5, #1
          RET
MAX       .FILL   xC005
; MAX <-- -x3FFB
```

Exercise: Modify the above routines to save registers we will need.

Example: palindrome check

- Palindromes are numbers or strings that read the same forward as well as backward.
  - madam, refer, racecar, kayak
  - 12/21/33 - 12:21
  - Was it a car or a cat I saw?
  - $12321 = 111^3$
- How to check if a string is a palindrome?

LC3 Exercise/Demo: Palindrome check

An implementation of the stack `PUSH` & `POP` protocols is provided on Git. Use it to fill in the code to check if the 7-letter string starting at `STRSTART` is a palindrome or not.

Example: balanced parentheses

- Consider a string parsing algorithm where protocol where
    - Encounter a $(, [, \{ \mapsto$ push on stack
    - Encounter a $),], \} \mapsto$ pop from stack and compare with popped item
- When are the parenthesis matched?
    - No underflow AND
    - All comparisons ✓ AND
    - Stack empty when finished parsing

Example: RPN arithmetic

- Traditional arithmetic notation is called *infix* notation. Operations are inserted between operands. E.g. $5 + 3$ or 3×4
    - Requires use of parenthesis to indicate order of operations
- An alternative notation is called postfix notation a.k.a Reverse Polish notation (RPN). E.g. $53+$ or $34 \times$
    - Implemented properly, does not require parenthesis/brackets

Practice RPN - MP2 material

- Note: $53- \mapsto 5-3$
- Consider: $34*72-3*+$
    - What does it evaluate to?
    - What is the *infix* version of the above?