Stacks

CSE 3318 – Algorithms and Data Structures
Alexandra Stefan
University of Texas at Arlington

General Queues

- A queue (as a general concept) is a data type that stores a set of objects.
  - Let **Item** be the type of each object.
- Operations that a queue supports are:
  - ***void insert(Queue *q, Item x) -*** adds object **x** to set **q**. ***Want time complexity Θ(1)***
  - ***Item delete(Queue *q)*** - choose an object **x**, remove that object from **q**, and return it to the calling function. ***Want time complexity Θ(1)***
  - *Item peek(Queue q)* - see what the object that would be deleted next is, but do not remove it from the queue. Return a copy of it or a reference to it.
  - *bool is_empty(Queue q)*
  - *bool is_full(Queue q)*
  - *Queue create()* - may take initial capacity as an argument
  - *void destroy (Queue *q)*
  - *Queue join(Queue q1, Queue q2)* - joins two queues
- **Specialized queues** (based on what item **delete()** removes)
  - Last inserted -> Stack / Pushdown Stack / LIFO (Last In First Out)
  - First inserted -> FIFO Queue (First In First Out)
  - Item with the smallest/largest key (if each item contains a **key**) -> Priority Queue (Heap)
  - Random item

Stack (Pushdown Stack)

Main operations:

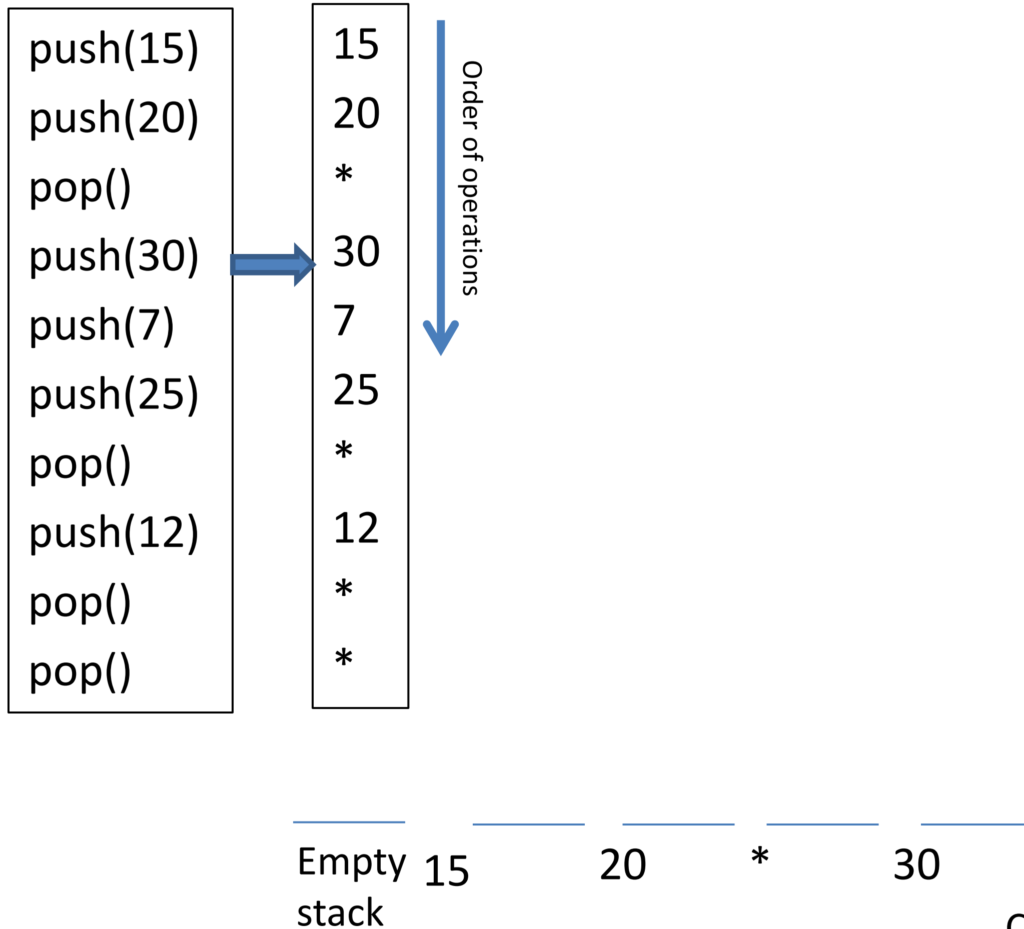
- **push** - Puts an item "on top of the stack".
- **pop** - Removes the item from the top of the stack (the last item added).

Applications

- Function execution in computer programs:
  - when a function is called, it enters the **calling stack**. The function that leaves the calling stack is always the last one that entered (among functions still in the stack).
- Provide a non-recursive implementation for problems that need a recursive solution
- Reverse order (e.g. reverse string)
- Check if balanced parenthesis
- Applications where you need to retrace your steps (and possibly try a different action). E.g. maze exploration, placing queens on a board. (For such problems it is often easier to write a recursive solution than an iterative one.)
- Interpretation and evaluation of symbolic expressions:
  - evaluate expressions like (5+2)*(12-3), or
  - parse C code (as a first step in the compilation process).
- Search methods.
  - traverse or search a graph
- Check leetcode – see what problems require a stack

Push / Pop – Work sheet

Conventions:
- value: Push value on stack
- * : Pop from stack



Empty stack 15 20 * 30

Continue with the other operations.

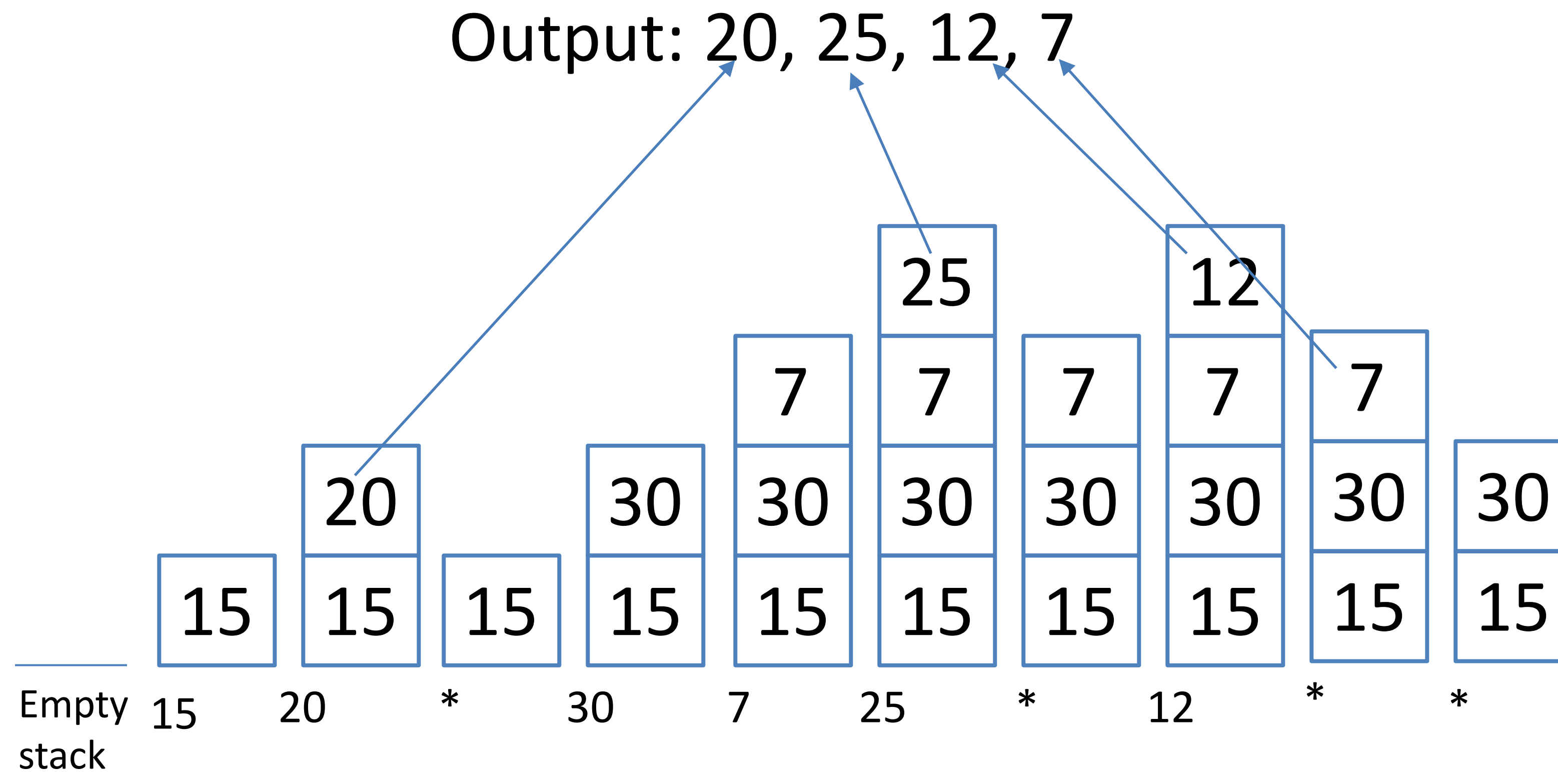
Push / Pop – Answers

Conventions:
- value: Push value on stack
- * : Pop from stack

push(15)
push(20)
pop()
push(30)
push(7)
push(25)
pop()
push(12)
pop()
pop()

15
20
*
30
7
25
*
12
*
*

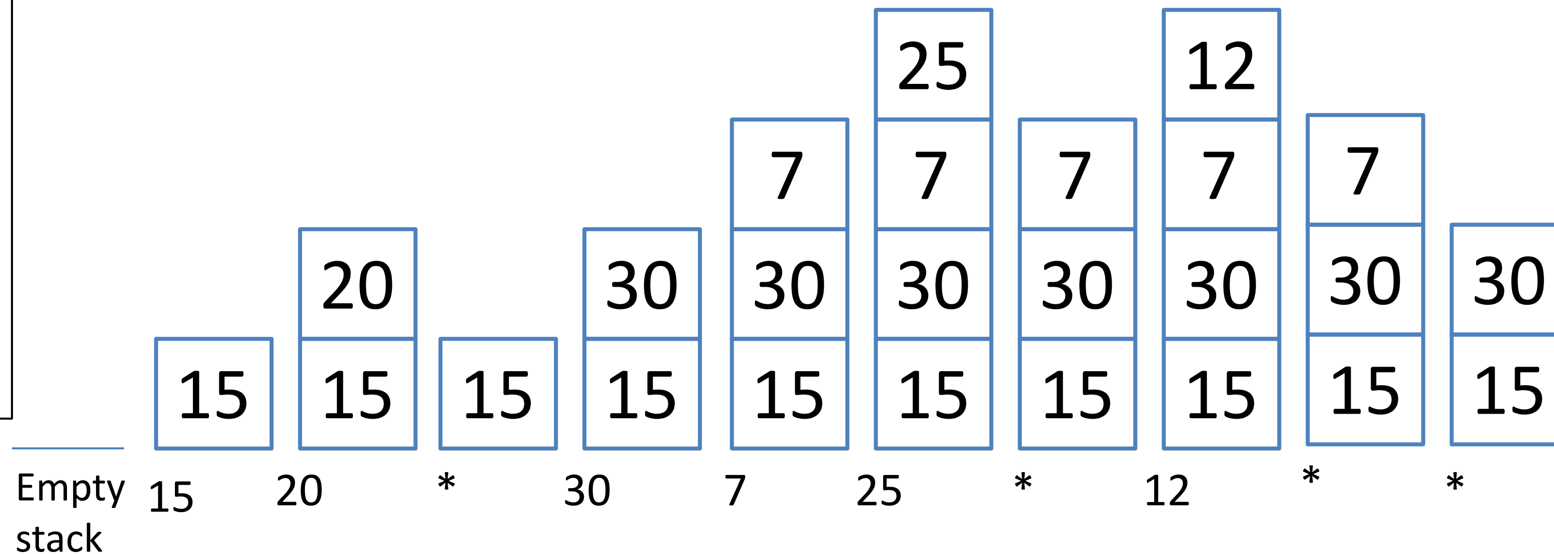
Order of operations



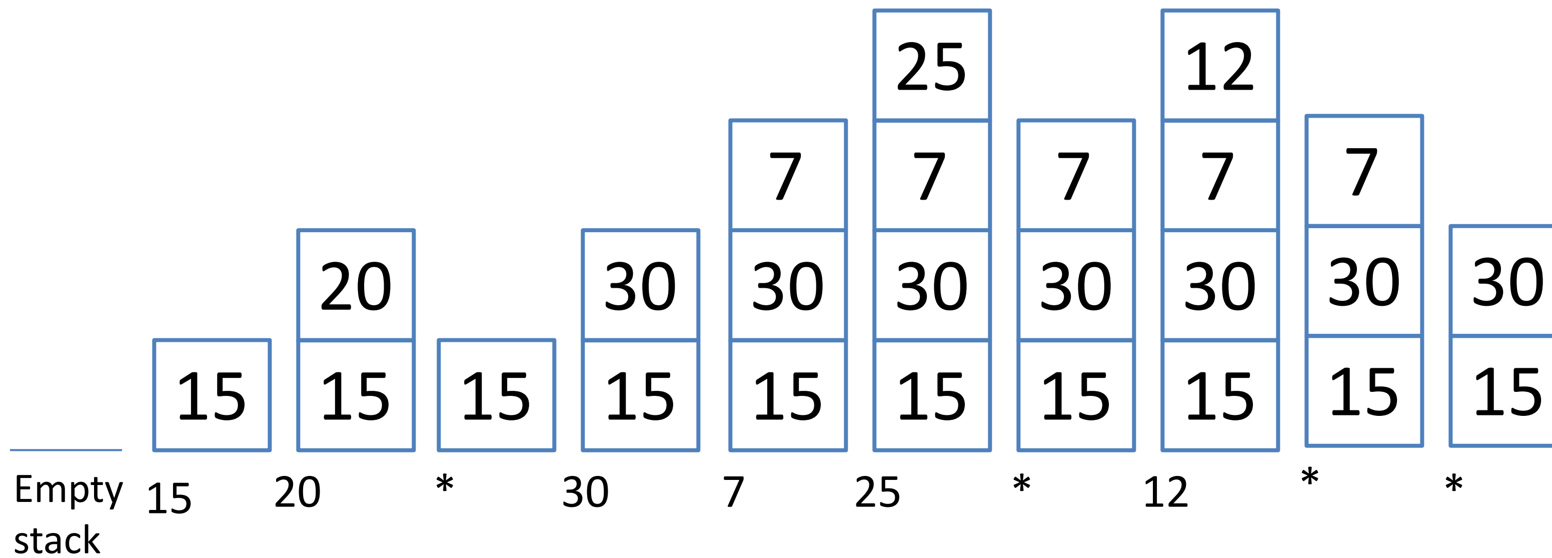
Terminology
Input, Operations, and Output Sequence

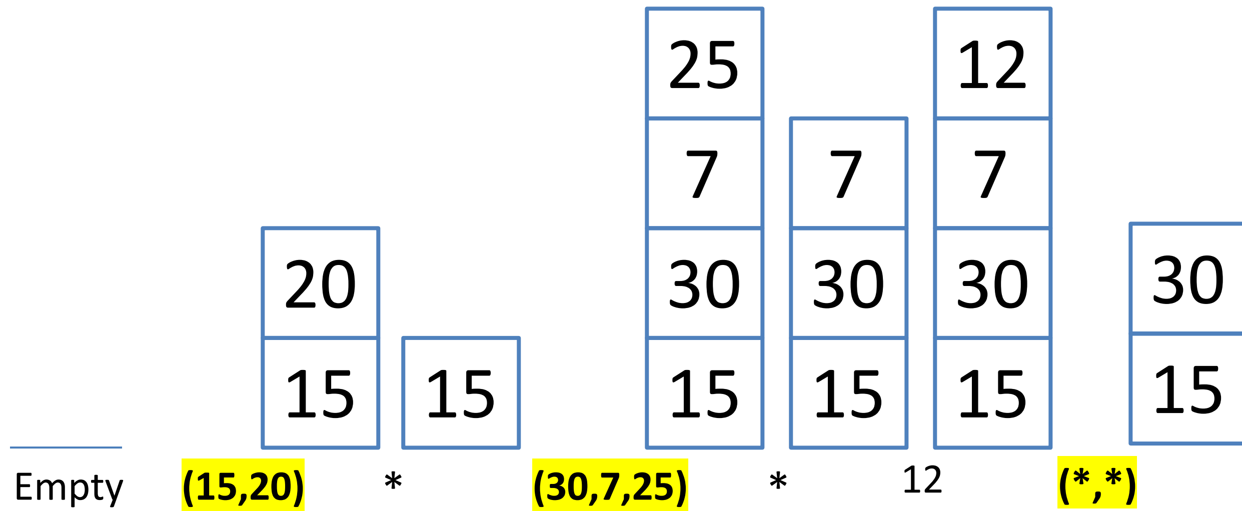
Operations:
15
20
*
30
7
25
*
12
*
*

In this example:
The **input sequence** is: **15, 20, 30, 7, 25, 12**
The **sequence of operations** is: **15, 20, *, 30, 7, 25, *, 12, *, ***
The **output sequence** produced by these operations on a stack is: **20, 25, 12, 7**
The **stack** and its changes is shown below.



Notation conventions for multiple operations and online testing





shows multiple operations of the same type

Notation conventions for online testing (in Canvas):

Operation:*stack after that operation*

E.g.:

:	(empty stack)
15:*15*	(push 15 & *stack)*
20:*15,20*	(push 20 & *stack)*
*:*15*	(pop & *stack)*
7,34:*15,7,34*	*(push 7 then push 34*)
****:*	*(pop,pop,pop)*

Exercises

Conventions:

letter - push(letter)

* - pop()

1. Given sequence of operations, show the stack and output:
   1. ROS**T*E*X*
   2. ROS**T*E***X*   (error)
   3. SOM**E*T**H*I*NGTO*D**O*** (a longer example)
2. Given input and output sequence, show the push and pop operations

   Example: given input: CAT and output ACT, the answer is: CA**T* . (Insert * in the input sequence s.t. that these operations give the desired output.)
   1. Input: INSATE,
      1. Output: SANETI,   Operations Sequence:
      2. Output: ANSITE,   Operations Sequence: (error)

Implementing Stacks

- A stack can be implemented using:
  - Single Linked List – see animation
  - Array - see animation

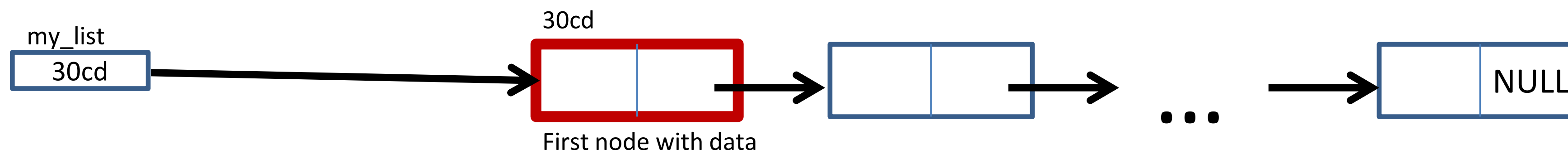
- Both implementations are straightforward.

Single Linked List - Based Stacks

- **Linked List** implementation:
  - What is a stack?
    - A stack is a single linked list.
  - **push(&stack, item)**
    - How? :
    - O(1) – wanted (frequent operation for this data structure)
  - **pop(&stack)**
    - How? :
    - O(1) – wanted (frequent operation for this data structure)
- What type of insert and remove are fast for single linked lists?
  - How many 'ways' can we insert in a list?
- See animation

Single Linked List - Based Stacks

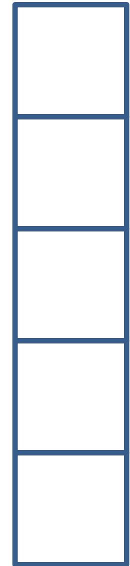
- List-based implementation:
  - What is a stack?
    - A stack is a single linked list.
  - **push(&stack, item)**
    - How? : inserts that item at the beginning of the list.
    - O(1)
  - **pop(&stack)**
    - How? : removes (and returns) the item at the **beginning** of the list.
    - O(1)
- See animation

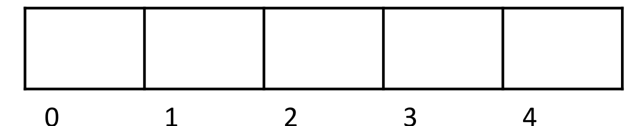


Array-Based Stacks

Array-based Stacks

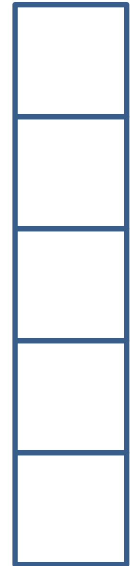
- **Array-based** implementation:
  - What is a stack? What will hold the data of the stack?
  - **push(&stack, item)**
    - How? :
    - Time : O(1) - can we get this?
    - Space: O(1) ?
  - **pop(&stack)**
    - How? :
    - Time: O(1) - can we get this?
    - Space: O(1) ?
  - Perform operations: push(5), push(8), push(20), pop()

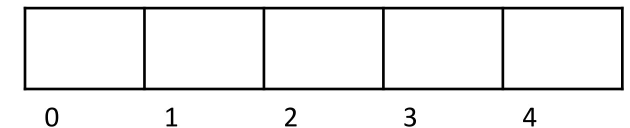




Array-based Stacks

- **Array-based** implementation:
  - What is a stack? What will hold the data of the stack?
    - **An array.**
  - **push(&stack, item)**
    - How? : **'insert' at the end of the array.**
    - Time: O(1) - **Yes**
    - Space: O(1)
  - **pop(&stack)**
    - How? : **'remove' from the end of the array.**
    - Time: O(1) - **Yes**
    - Space: O(1)
- See animation





Defining Stacks Using Arrays

We must choose clear conventions for:

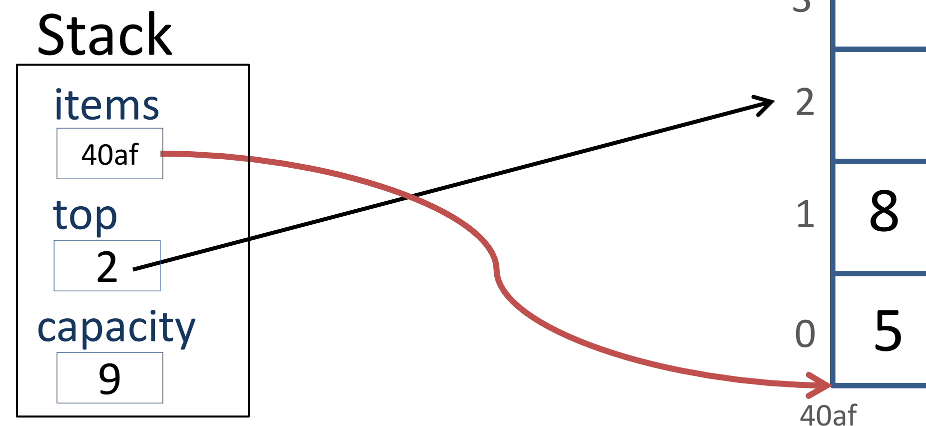
- How to represent the stack
    - What an empty stack looks like (NULL or unused space?)
    - What a full stack looks like
    - What a stack that is partially filled looks like
- What happens at push (how the stack changes)
    - What to do if stack is full: resize or refuse
- What happens at pop (how the stack changes)
    - What to do if stack is empty

Once we chose the below representation (i.e. the data to represent the state of the stack) we must be specific about how it works. For example there are two options for `top`:

- it points to the last element on the stack
- it points to the first available cell on the stack (where the new item pushed onto the stack would be placed)

See animation

```
struct stack_array
{
    int * items;
    int top; //index AFTER last item
    int capacity;
};
```



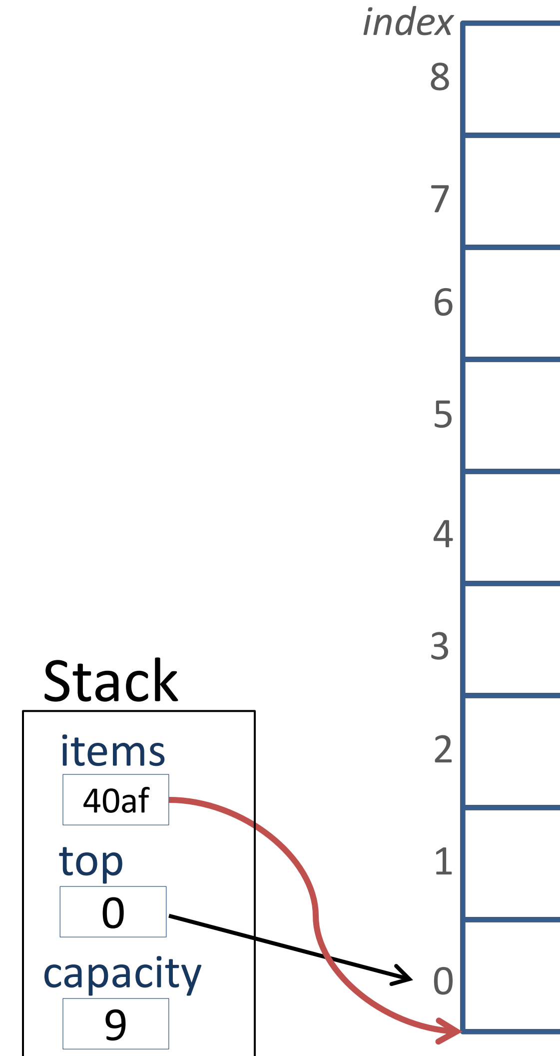
Creating a New Stack

```
struct stack_array{
    int * items;
    int top; //index AFTER last item
    int capacity;
};

// struct stack_array S = newStack(9);  // in main
struct stack_array newStack(int cap){
   struct stack_array res;
//Item temp[cap];  res.items = temp; // BAD
   res.items = malloc(cap*sizeof(int));
   res.capacity = cap;
   res.top = 0;
   return res;
}
```

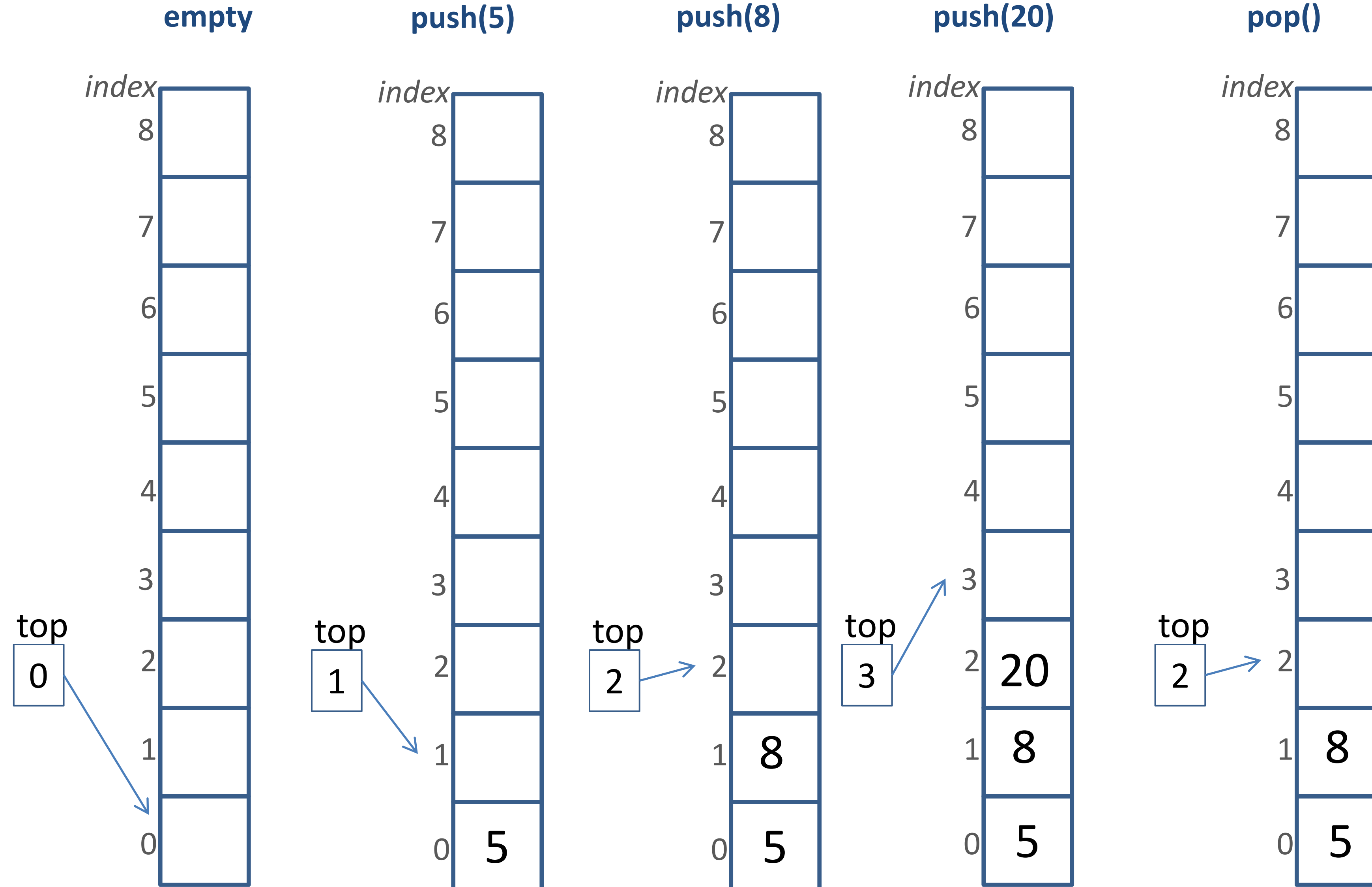
Do not use an array for items (e.g. items[100])!
See the Victim-TAB example showing the difference between Stack and Heap memory.

Why is it ok to return res? (Does it have the TAB pb?)



push(5)
push(8)
push(20)
pop()

Array-based Stacks



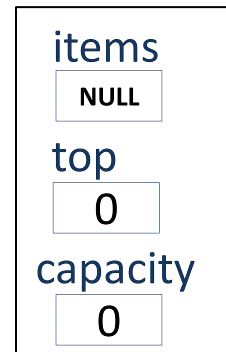
Destroying a stack

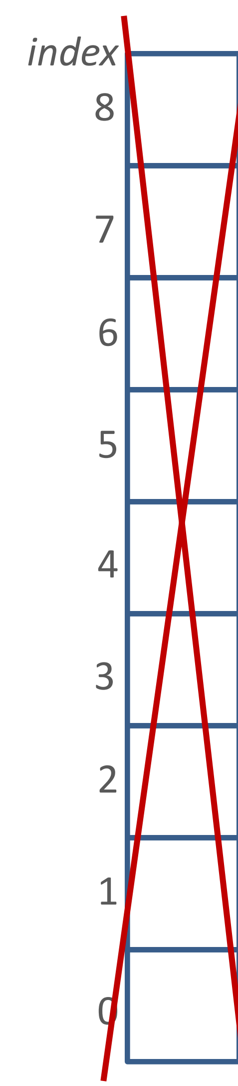
```
struct stack_array{
    int * items;
    int top; //index AFTER last item
    int capacity;
};

void destroy(struct stack_array  * S){
    if (S!=NULL) {
        free(S->items); // what happens if S->items is  NULL?
        S->items = NULL;
        S->capacity = 0; S->top = 0;
    }
}
// main
// struct stack_array S = newStack(9);
// destroy(&S);
```

Do not use an array for items (e.g. items[100])!
See the Victim-TAB example showing the difference between Stack and Heap memory.

Why is it ok to return res? (Does it have the TAB pb?)





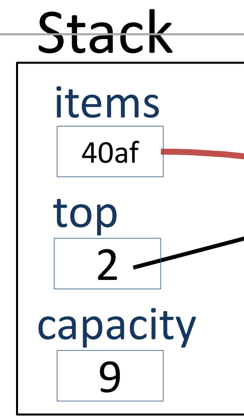
Practice: write push(), pop()

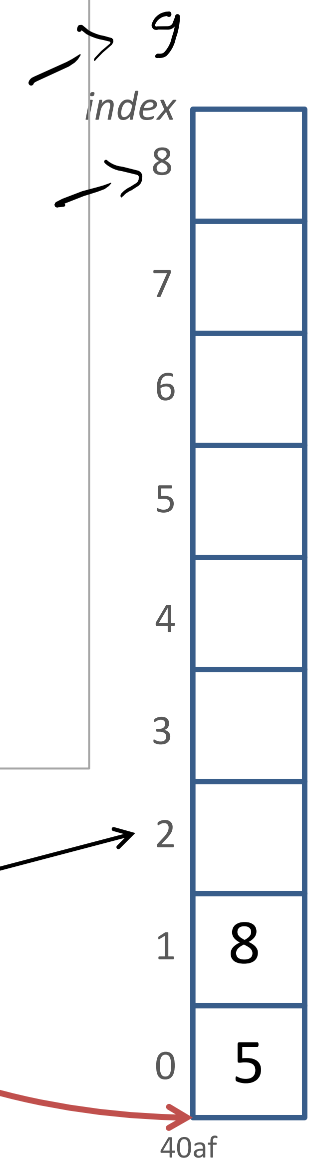
```
___ push(struct stack_array * myStack, int newItem)




____________ pop(struct stack_array * myStack )
```

```
typedef int Item;
struct stack_array
{
    Item * items;
    int top; //index AFTER last item
    int capacity;
};
```





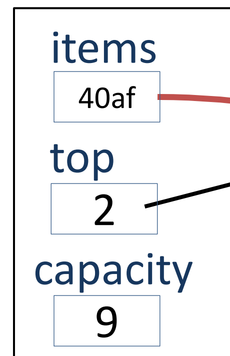
Practice: write push(), pop()

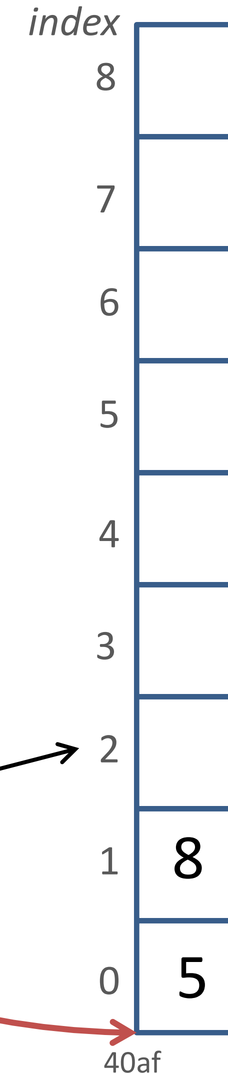
```
void push(struct stack_array * myStack, int newItem) {
    if ((myStack==NULL) || (myStack->top==myStack->capacity))
        printf("Full or myStack is NULL. Exit");
    else {
        myStack->items[myStack->top] = newItem;
        myStack->top++;  // top becomes 3
    }
}

int pop(struct stack_array * myStack )
    int temp = 0;  // or some other default item value, e.g. -1
    if ((myStack==NULL) || (myStack->top==0)) {
        printf("The stack is empty. Exit");
    }
    else {
        myStack->top--;  // top becomes 1
        temp = myStack->items[myStack->top]; // temp has value 8
    }
    return temp;
}
```

```
struct stack_array
{
    int * items;
    int top; //index AFTER last item
    int capacity;
};
```

Stack





Summary and Practice

- Summary
  - Stack
  - Implementation (code)
    - Time complexity
    - Space complexity
  - Application
- Practice from leetcode
  - 20. Valid Parentheses

  https://leetcode.com/problems/valid-parentheses/
  - 1544. Make The String Great

  https://leetcode.com/problems/make-the-string-great/
  - 1190. Reverse Substrings Between Each Pair of Parentheses

  https://leetcode.com/problems/reverse-substrings-between-each-pair-of-parentheses/

Solution for input CAT, output ACT -> give operations

- Input ~~C~~AT: push(C), push(A), push(T) insert pops in between them to get the output ACT
- Output ~~ACT~~

:

C:C

A:CA

*: C

*:

T: T

*:

Solution for input INSATE, output SANETI -> give operations

- ~~INSA~~TE
- Ouput: ~~SAN~~ETI

INS: INS

*: IN   (S)

A:INA

*: IN   (A)

*: I   (N)

TE: ITE

*: IT   (E)

*: I   (T)

*:   (I)