Data Structures and Algorithms Course

Stacks & Recursion Review

Course Content

- Big O notation ✓
- Arrays ✓
- Linked Lists ✓
- **Stacks**
- **Recursion**
- Queues
- Hashing
- Sets
- Trees
- AVL Trees
- Heap
- Tries
- Graphs
- Sorting Algorithms
- Searching Algorithms

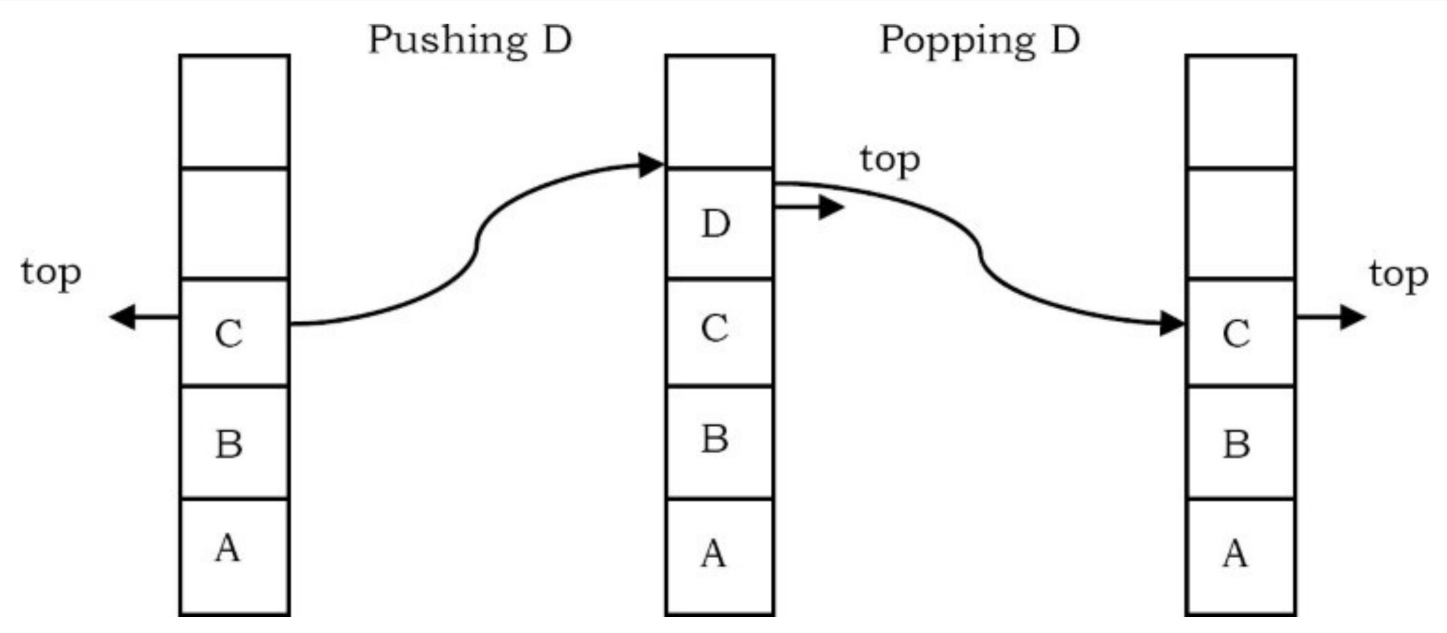
Stacks

"First In" will be the "Last Out"
"FILO"

- You can only pick a new plate from the Top.
- You can only put a plate to the Top.

What is a Stack?

- A stack is an ordered list in which insertion and deletion are done at one end, called top.
- The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.
- When an element is inserted in a stack, the concept is called push,
- When an element is removed from the stack, the concept is called pop.



How does a “Stack” work?

Push each char in word to stack.

String word=“C Y D E O”

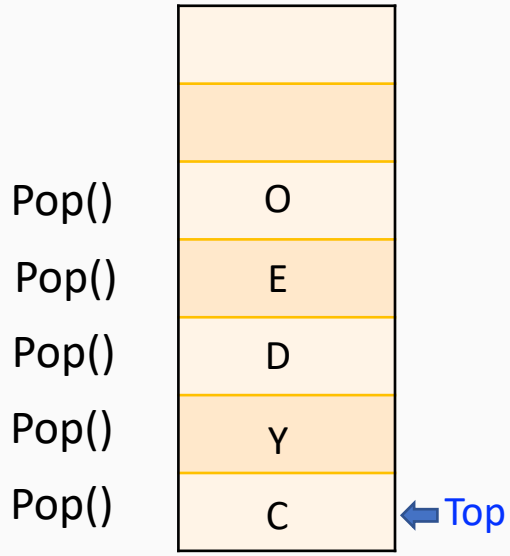
Push “C” to stack

Push “Y” to stack

Push “D” to stack

Push “E” to stack

Push “O” to stack



Stack is empty

Pop the items in and add them to a string

String newWord= “ “ Word is reversed

Stack Operations

- Main stack operations

  - void push(int data): Inserts data onto stack.

  - int pop(): Removes and returns the last inserted element from the stack.

- Auxiliary stack operations

  - Top(): Returns the last inserted element without removing it. (Peek())

  - int Size(): Returns the number of elements stored in the stack.

  - int IsEmpty(): Indicates any elements are stored in the stack or not.

  - int IsFull(): Indicates the stack is full or not.

- Can be implemented using Arrays or Linked Lists (stack is an Abstract Data Type-ADT)

Stack Applications

- Direct applications
  - Balancing of symbols
  - Infix-to-postfix conversion
  - Evaluation of Postfix & Prefix expressions
  - Implementing method calls (including recursion)
  - Page-visited history in a Web browser [Back Buttons]
  - Undo sequence in a text editor
  - Reversing Strings, arrays etc.

Infix, Postfix and Prefix Notations

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

- Infix : (A+B)
- Postfix: AB+
- Prefix: +AB

Stacks Performance

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of push()	O(1)
Time Complexity of pop()	O(1)
Time Complexity of size()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of isFullStack()	O(1)
Time Complexity of deleteStack()	O(1)

Custom Stacks Implementation – With Linked Lists

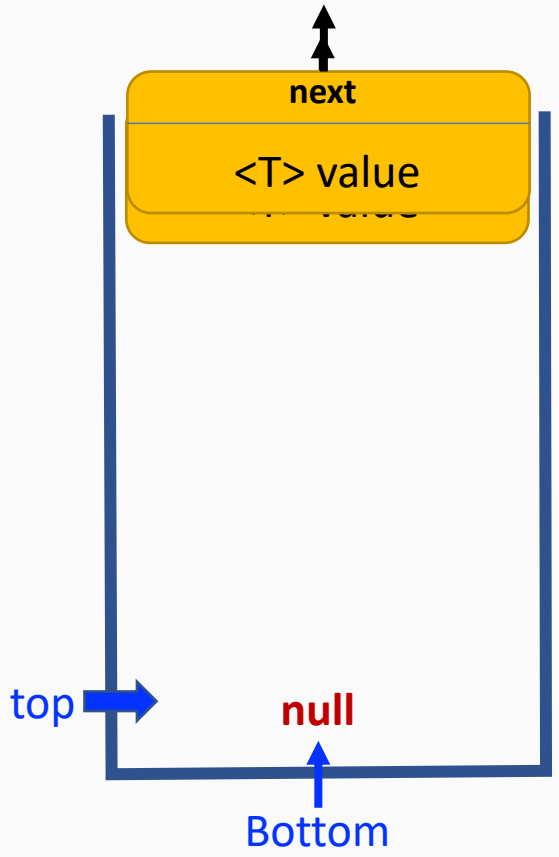
MyStack<T>

Operation:

Push(<T> value)

Push(<T> value)

Pop()



What is the performance of a pop() ?

Custom Stacks Implementation – With Linked Lists

MyStack<T>

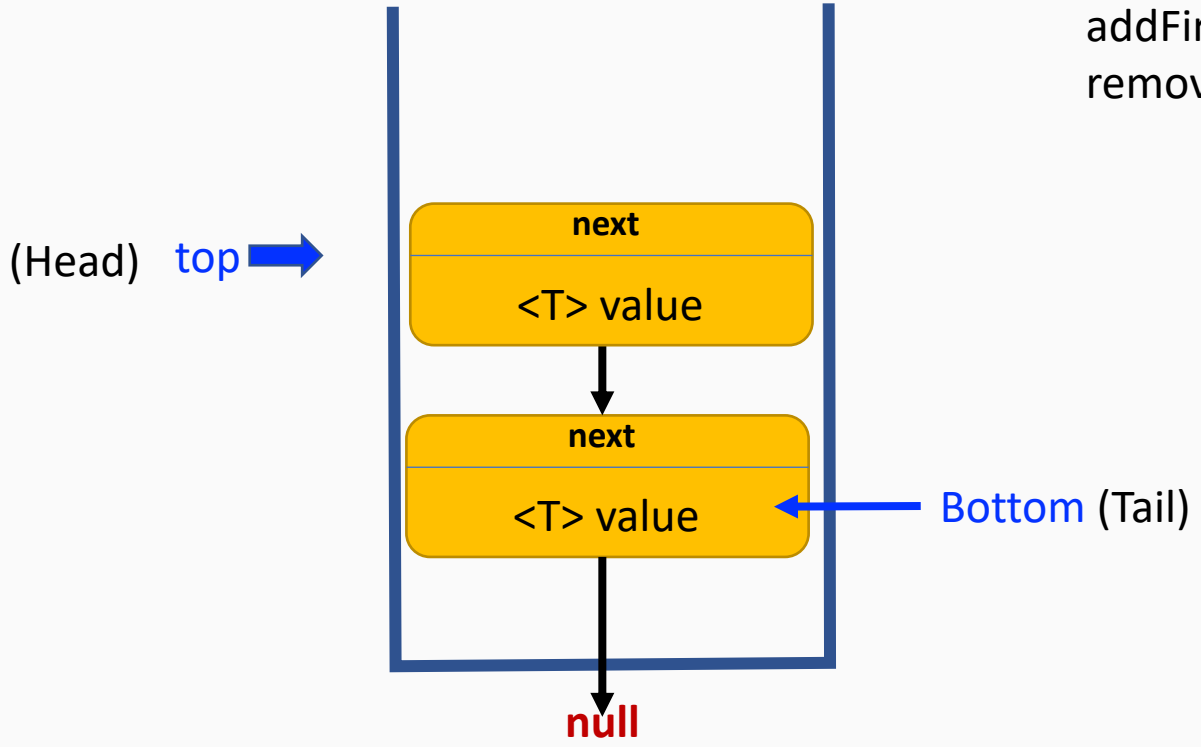
Operation:

Push(<T> value)

Push(<T> value)

Pop()

addFirst()
removeFirst()



Stacks Implementation – With Linked Lists

Lets switch to IntelliJ for creating
a new custom generic ‘Stack’ with O(1) pop()

Stacks Implementation – Collection Framework

Lets switch to IntelliJ for creating a Collections Framework ‘Stack’

Stacks –(Symbol Balancing – Interview Question)

Question : How do we check if the expressions are balanced with the opening and closing delimiters?

Example	Valid	Description
(A+B)+(C-D)	Yes	Symbols are balanced.
((A+B)+(C-D)	No	One Closing brace missing
((A+B)+[C-D])	Yes	Symbols are balanced.
((A+B)+[C-D]}	No	The last closing brace does not match the first opening paranthesis

Algorithm:

1. Create a stack.
2. While (end of input is not reached)
   a. If the character is not a symbol to be balanced ((,),[,],{ or }), ignore it.
   b. If the character is an opening symbol like (, [, {, push it onto the stack
   c. If it is a closing symbol like),],}, and if the stack is empty return false.
      Else pop the stack.
   d. If the symbol popped != corresponding opening symbol, return false.
3. At end of input, if the stack is not empty report an error.

Stacks –(Symbol Balancing – Interview Question)

- Expression = (A+B)- ((C*D)/ [F*(B/2)])
- Algorithm will handle only: A[i]= () (() [()])

Algorithm:

1. Create a stack.
2. while (not end of exp)
   a. If not a symbol to be balanced, ignore it.
   b. If opening symbol like **(, [, {,** push it to stack.
   c. If closing symbol like **),],},**
      - if the stack is empty return false.
      - else pop the stack.
      - If popped is not matching current char, return false.
3. At end of input, if the stack is empty return true.

A[i]	Operation	Stack	Output
(	Push ‘ (‘	(	
)	Pop ‘(’ match with ‘)’-> YES	-	
(	Push ‘ (‘	(	
(	Push ‘ (‘	((	
)	Pop ‘(’ match with ‘)’-> YES	(	
[	Push ‘ [‘	([	
(	Push ‘ (‘	([(	
)	Pop ‘(’ match with ‘)’-> YES	([	
]	Pop ‘[’ match with ‘]’-> YES	(	
)	Pop ‘(’ match with ‘)’-> YES	-	
	Expression end. Is Stack empty? ->YES		**true**

Time Complexity: O(n). Since it is only one pass Space Complexity: O(n) [for stack].

Stacks Key Takeaways

- Last-In-First-Out (LIFO)
- Abstract Data Type (ADT)
- Can be implemented using Arrays/Linked Lists
- Almost all operations run in O(1)

Recursion

```
Method A(n){
Return B(n)+ 1
}
```

```
Method B(n){
Return C (n);
}
```

```
Method C(n){
Return n
}
```

Recursion

- One way to describe repetition within a computer program is the use of loops, such as Java's while-loop and for-loop.
- An entirely different way to achieve repetition is through a process known as '***Recursion'***.
- A recursive function calls itself until a "base condition" is true, and execution stops.
- It is important to ensure that the recursion terminates at some point.
- Any function/method calls itself is recursive.
- Recursive code is generally shorter and easier.
- Requires use of Stacks for function calls.
- Large size of data may result in stack overflow.

Recursion

Factorial Function $$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

Factorial Calculation:

5! = 5 x 4 x 3 x 2 x 1

5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
.
.

n!=n x (n-1)!

Recursion

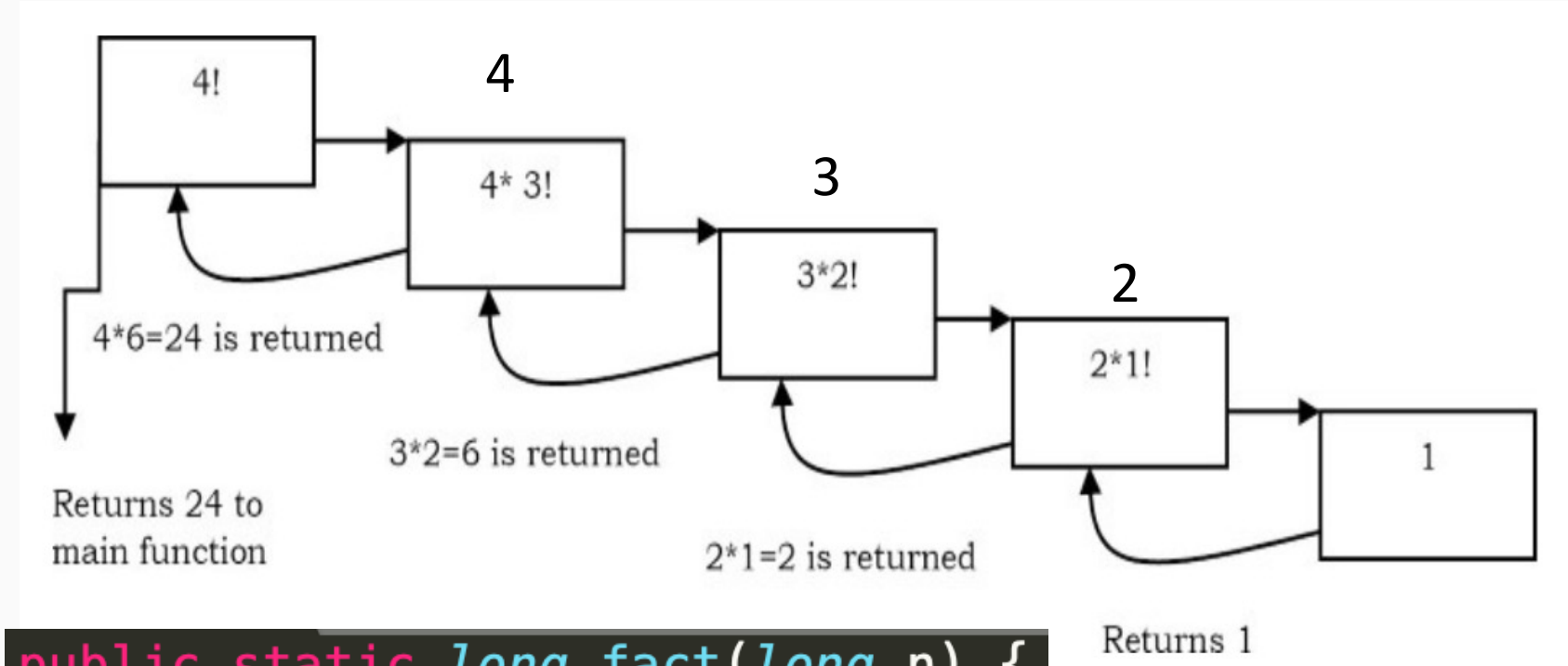
Recursive Factorial Method: Formula : n!=n*(n-1)!

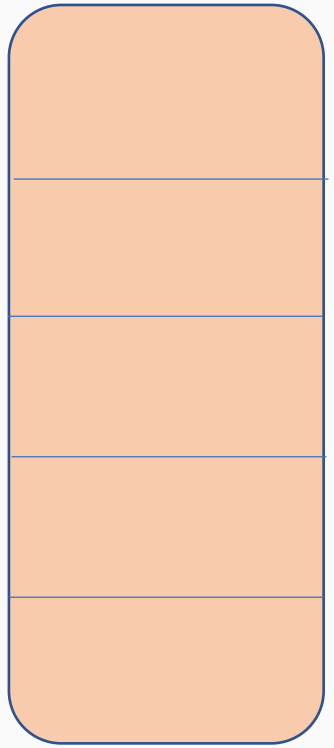
```
public static long fact(long n) {
        if (n <= 1)
            return 1;
        else
            return n * fact(n - 1);
    }
```

Recursion

- Assume we are calculating 4!

$n!=n*(n-1)!$





Stack

```
public static long fact(long n) {
        if (n <= 1)
            return 1;
        else
            return n * fact(n - 1);
    }
```

Result= 1 *2 *3 *4

Recursion-Key Takeaways

- Anything that can be done with recursion can be done iteratively (with loop).
- Recursion is clever, readable and shortly but costly in terms of space complexity.
- Why use recursion over iterative approach?
  - When it comes to problems like sorting and tree traversal we can use recursive solutions to make things simple.
  - Particularly,
    - Merge Sort
    - Quick Sort
    - Tree Traversal
    - Graph Traversal all use recursion.

LeetCode Problem : 1472. Design Browser History

You have a browser of one tab where you start on the homepage and you can visit another url, get back in the history number of steps or move forward in the history number of steps.

Implement the BrowserHistory class:

BrowserHistory(string homepage) Initializes the object with the homepage of the browser.

void visit(string url) Visits url from the current page. It clears up all the forward history.

string back(int steps) Move steps back in history. If you can only return x steps in the history and steps > x, you will return only x steps. Return the current url after moving back in history at most steps.

string forward(int steps) Move steps forward in history. If you can only forward x steps in the history and steps > x, you will forward only x steps. Return the current url after forwarding in history at most steps.

LeetCode Problem : 1472. Design Browser History

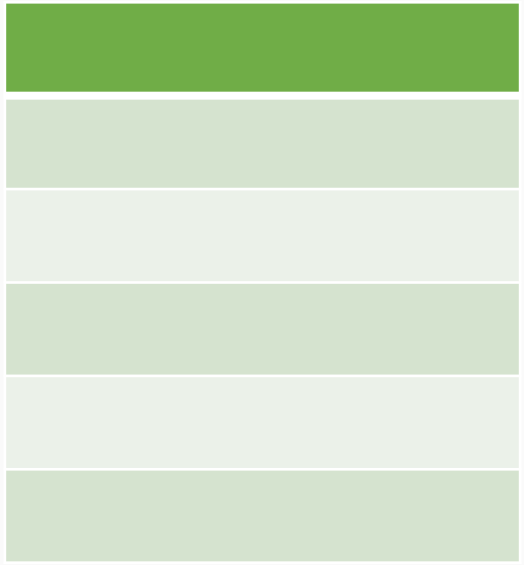
Browser History(“leetcode.com”)

visit(“google.com”)

visit(“facebook.com”)

visit(“youtube.com”)

Back(1)



History stack

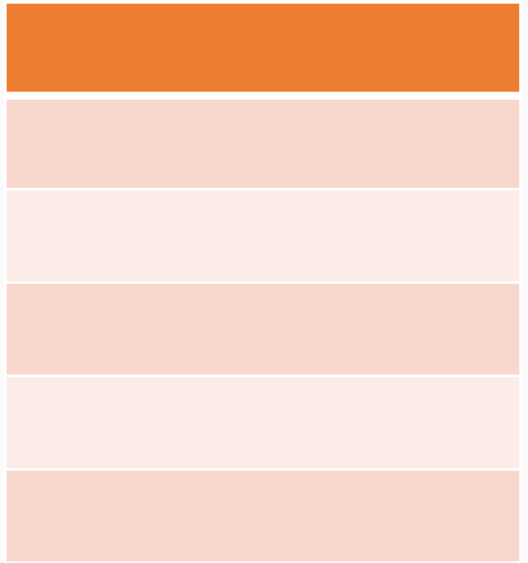
Current Url:

“facebook.com”

(“google.com”)

“leetcode.com”

“youtube.com”



Forward stack

LeetCode Problem : 1472. Design Browser History

Algorithm

1.Initialize variables:

1. Two stacks of strings history and forward, to store the URLs.
2. A string variable current, to store the currently visited URL, which is initialized with the given homepage as it is the first visited URL.

2.Implementing visit(url) method:

1. As we will **visit a new URL** url, we will store current in the history stack, and
2. make the given url as current, and clear the forward stack.

3. Implementing back(int steps) method:

1. We need to **go back by step URLs**.
2. While there are elements in the history stack and we haven't popped step elements from it, we will push current in the forward stack and pop the most recently visited URL from the history stack and mark it as current.
3. At the end, we return current.

4. Implementing forward(steps) method:

1. We need to **go forward by step URLs**.
2. While there are elements in the forward stack and we haven't popped step elements from it, we will push current in the history stack and pop the most recently visited URL from the forward stack and mark it as current.
3. At the end, we return current.

LeetCode Problem : 231. Power of Two

Given an integer n, return *true if it is a power of two. Otherwise, return false*.
An integer n is a power of two, if there exists an integer x such that n == 2^x.

Example 1:
Input: n = 1
Output: true
Explanation: $2^0 = 1$

Example 2:
Input: n = 16
Output: true
Explanation: $2^4 = 16$

LeetCode Problem : 231. Power of Two

Let n=16

16/2 = 8
8 / 2 = 4
4 / 2 = 2
2 / 2 = 1