DATA STRUCTURE

Chapter 3 — STACKS

IMPORTANT PROBLEMS & SOLUTIONS

Set 5 (Problems 40–48)

Branching Recursion, Tree Traversals, String & Queue Problems, and a Dynamic-Stack / Advanced Application

(Diploma / B.Tech — Computer Science & Engineering)

About This Set

This fifth set moves from linear recursion (as seen with factorial in Set 1) to branching recursion, connects stacks with binary tree traversal, and closes with one advanced application problem often asked in higher-level examinations. Problem numbering continues from Sets 1–4. This set covers:

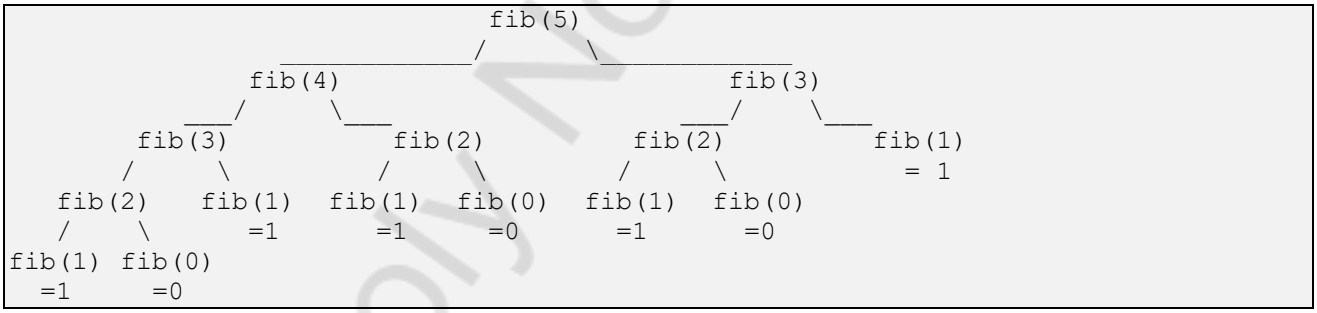
- Branching recursion — the Fibonacci call tree and the Euclidean GCD algorithm
- Removing adjacent duplicate characters from a string using a stack
- Finding the minimum removals needed to make a parenthesis string valid
- Non-recursive (stack-based) inorder and preorder traversal of a binary tree
- Reversing the first K elements of a queue using a stack
- Overcoming the fixed-size limitation of an array-based stack through dynamic resizing
- Largest Rectangle in a Histogram — an advanced stack application

Problem 40

Statement: Using the recursive definition fib(n) = fib(n-1) + fib(n-2), with fib(0) = 0 and fib(1) = 1, draw the call tree for fib(5) and determine (a) the total number of function calls made, and (b) the maximum depth of the call stack at any time.

Solution:

Unlike factorial, where each call makes only one further call (linear recursion), fib(n) makes two further calls, so the call stack now branches into a tree. Every node below is one activation record that is pushed onto the stack when the call is made and popped when it returns:



(a) Counting every node in the tree above (every call, whether it returns directly from a base case or recurses further) gives a total of 15 function calls for fib(5).

(b) At any given moment, only the calls lying on a single root-to-leaf path are simultaneously present on the call stack (since a call waits for its first recursive call to finish before making its second). The longest such path is fib(5) → fib(4) → fib(3) → fib(2) → fib(1), which is 5 calls deep — so the maximum stack depth is 5, even though 15 calls are made in total.

Answer: Total calls = 15; Maximum call-stack depth = 5; fib(5) = 5.

Problem 41

Statement: Using the Euclidean algorithm gcd(a, b) = a if b = 0, else gcd(b, a mod b), trace the activation records placed on the stack for the call gcd(48, 18), and find the result.

Solution:

Calling phase — one activation record is pushed for each call, until the base case (b = 0) is reached:

Call (pushed, top → bottom order at deepest point)	Condition Checked	Next Action
gcd(6, 0) (TOP — base case)	b = 0	returns 6 directly
gcd(12, 6)	b = 6 ≠ 0	calls gcd(6, 12 mod 6) = gcd(6, 0)
gcd(18, 12)	b = 12 ≠ 0	calls gcd(12, 18 mod 12) = gcd(12, 6)
gcd(48, 18) (bottom — first call made)	b = 18 ≠ 0	calls gcd(18, 48 mod 18) = gcd(18, 12)

Returning phase — each activation record is popped from the top downward, passing its result straight back to the call that is waiting for it (the Euclidean algorithm does no further computation on the way back, unlike factorial):

1. gcd(6, 0) returns 6 → popped.
2. gcd(12, 6) simply returns the value 6 it received → popped.
3. gcd(18, 12) returns 6 → popped.
4. gcd(48, 18) returns 6 → popped (call stack now empty).

Answer: gcd(48, 18) = 6.

Problem 42

Statement: Repeatedly remove pairs of adjacent, identical characters from the string "abccba" using a stack, until no more such pairs remain.

Solution:

```
Algorithm REMOVE_ADJACENT_DUPLICATES(STR)
Step 1: Create an empty stack S
Step 2: For each character CH in STR, left to right:
           If S is not empty AND S.top() = CH, then
                Pop(S)            // cancels the duplicate pair
           Else
                Push(CH)
Step 3: The characters remaining in S, read bottom to top, form the result
```

Character Scanned	Action	Stack (bottom → top)
a	Push (stack empty)	a
b	Top='a' ≠ 'b' → Push	a, b
c	Top='b' ≠ 'c' → Push	a, b, c
c	Top='c' = 'c' → Pop (cancel)	a, b
b	Top='b' = 'b' → Pop (cancel)	a
a	Top='a' = 'a' → Pop (cancel)	(empty)

Notice how removing the innermost "cc" pair exposes a new "bb" pair, and removing that in turn exposes an "aa" pair — the stack naturally handles this cascading cancellation without any extra logic.

Answer: The entire string cancels out — the result is the empty string.

Problem 43

Statement: Find the minimum number of characters that must be removed from the string ()())(to make it a validly balanced parenthesis string, using a stack.

Solution:

Rule: push the index of every '(' encountered. For every ')', if the stack is non-empty, pop it (a valid match); if the stack is empty, this ')' is unmatched and must be removed. At the end, every index still left in the stack is an unmatched '(' that must also be removed.

Index	Character	Action	Stack (indices, bottom → top)
0	(	Push index 0	0
1	)	Stack non-empty → pop 0 (matched)	(empty)
2	(	Push index 2	2
3	)	Stack non-empty → pop 2 (matched)	(empty)
4	)	Stack empty → this ')' is UNMATCHED (mark for removal)	(empty)
5	(	Push index 5	5

At the end of the scan, one unmatched ')' was found (at index 4), and one '(' is still left on the stack unmatched (at index 5). Removing the characters at indices 4 and 5 from "()())(" leaves "()()", which is validly balanced.

Answer: Minimum removals = 2 (the ')' at index 4 and the '(' at index 5), giving the valid string "()()".

Problem 44

Statement: For the binary tree shown below, perform an inorder traversal (Left, Root, Right) without recursion, using an explicit stack.

Solution:

```
Algorithm ITERATIVE_INORDER(root)
Step 1: stack = empty ; curr = root
Step 2: While curr ≠ NULL OR stack is not empty:
           While curr ≠ NULL:
                Push(curr)
                curr = curr.left
           curr = Pop()
           Visit(curr)
           curr = curr.right
```

Step	Action	Stack (bottom → top)	Output so far
1	Push A, B, D (following left children)	A, B, D	
2	curr = NULL → Pop D → Visit D → curr = D.right (NULL)	A, B	D
3	curr = NULL → Pop B → Visit B → curr = B.right = E	A	D, B
4	Push E (E.left is NULL, so loop ends immediately)	A, E	D, B
5	curr = NULL → Pop E → Visit E → curr = E.right (NULL)	A	D, B, E

6	curr = NULL → Pop A → Visit A → curr = A.right = C	(empty)	D, B, E, A
7	Push C (C.left is NULL); Pop C → Visit C → curr = NULL	(empty)	D, B, E, A, C

Answer: Inorder traversal: D, B, E, A, C.

Problem 45

Statement: For the same binary tree as Problem 44, perform a preorder traversal (Root, Left, Right) without recursion, using an explicit stack.

Solution:

```
Algorithm ITERATIVE_PREORDER(root)
Step 1: If root = NULL, Return
Step 2: Push(root)
Step 3: While stack is not empty:
           node = Pop()
           Visit(node)
           If node.right ≠ NULL, Push(node.right)
           If node.left ≠ NULL, Push(node.left)
```

Pushing the right child before the left child ensures the left child is popped (and therefore visited) first, which is what preorder requires.

Step	Action	Stack (bottom → top)	Output so far
1	Push A	A	
2	Pop A → Visit A → push C (right), push B (left)	C, B	A
3	Pop B → Visit B → push E (right), push D (left)	C, E, D	A, B
4	Pop D → Visit D → no children	C, E	A, B, D
5	Pop E → Visit E → no children	C	A, B, D, E
6	Pop C → Visit C → no children	(empty)	A, B, D, E, C

Answer: Preorder traversal: A, B, D, E, C.

Problem 46

Statement: Reverse the first K = 3 elements of the queue (front → back) 1, 2, 3, 4, 5, keeping the remaining elements in their original order, using a stack.

Solution:

```
Algorithm REVERSE_FIRST_K(QUEUE, K)
Step 1: Dequeue the first K elements from QUEUE and Push each onto a stack S
Step 2: While S is not empty:
           Enqueue(Pop(S)) back into QUEUE
Step 3: Repeat (Size(QUEUE) - K) times:
           Dequeue an element from QUEUE and Enqueue it again
           (this moves the untouched elements from the front to the back)
```

Step	Action	Queue (front → back)	Stack S (bottom → top)
Start	—	1, 2, 3, 4, 5	(empty)
Step 1	Dequeue 1, 2, 3 and push onto S	4, 5	1, 2, 3
Step 2	Pop 3, 2, 1 and enqueue each in turn	4, 5, 3, 2, 1	(empty)

Step 3 (round 1)	Dequeue 4, enqueue 4	5, 3, 2, 1, 4	—
Step 3 (round 2)	Dequeue 5, enqueue 5	3, 2, 1, 4, 5	—

Answer: Final queue (front → back): 3, 2, 1, 4, 5 — the first 3 elements are reversed; 4 and 5 keep their original order.

Problem 47

Statement: Explain how the fixed-size limitation of an array-based stack (Section 3.4) can be overcome using a dynamic resizing (doubling) strategy, and trace it for a stack that starts with capacity 2, on the sequence: Push(10), Push(20), Push(30), Push(40), Push(50).

Solution:

Instead of declaring a fixed MAXSIZE and reporting overflow when it is reached, a dynamic stack starts with a small array and, whenever a PUSH is attempted on a completely full array, allocates a brand-new array of double the current capacity, copies every existing element into it, and only then completes the push. This trades an occasional O(n) copying cost for the guarantee that overflow (in the traditional sense) never happens as long as memory is available; averaged over many pushes, the extra cost works out to O(1) per push.

```
Algorithm PUSH_DYNAMIC(ITEM)
Step 1: If SIZE = CAPACITY, then
           Create a NEW array of size 2 × CAPACITY
           Copy all SIZE elements from the old array into NEW
           Replace the old array with NEW ; CAPACITY = 2 × CAPACITY
Step 2: ARRAY[SIZE] = ITEM ; SIZE = SIZE + 1
```

Operation	Size Before	Capacity Before	Resize Triggered?	Size After	Capacity After
Push(10)	0	2	No	1	2
Push(20)	1	2	No	2	2
Push(30)	2	2	Yes → capacity doubles to 4	3	4
Push(40)	3	4	No	4	4
Push(50)	4	4	Yes → capacity doubles to 8	5	8

Answer: Final stack: 10, 20, 30, 40, 50 (bottom to top), size = 5, capacity = 8 — no overflow occurred despite starting from a capacity of just 2.

Problem 48

Statement: For the histogram with bar heights [2, 1, 5, 6, 2, 3], find the area of the largest rectangle that can be formed, using a stack.

Solution:

Rule: maintain a stack of bar indices with strictly increasing heights. Scan the bars left to right (with an imaginary bar of height 0 appended at the end to flush the stack). Whenever the current bar is shorter than the bar at the stack's top, pop the top — that popped bar's height H is the height of a rectangle whose width extends from just after the new stack top to just before the current index; compute its area and update the maximum.

i	height[i]	Pop & Compute	Stack after step (indices)	Max Area So Far

0	2	— (stack empty)	0	0
1	1	Pop 0: h=2, width=1 (stack empty) → area=2	1	2
2	5	— (5 > height[1]=1)	1, 2	2
3	6	— (6 > height[2]=5)	1, 2, 3	2
4	2	Pop 3: h=6, width=4-2-1=1 → area=6. Pop 2: h=5, width=4-1-1=2 → area=10	1, 4	10
5	3	— (3 > height[4]=2)	1, 4, 5	10
6 (sentinel, h=0)	0	Pop 5: h=3,w=1→3. Pop 4: h=2,w=4→8. Pop 1: h=1,w=6→6	(empty)	10

The maximum area of 10 is achieved by the rectangle formed using bars at indices 2 and 3 (heights 5 and 6), taking the smaller height 5 across a width of 2.

Answer: Largest rectangle area = 10.

Practice Tip: For branching-recursion questions, always separate "total number of calls" from "maximum stack depth" in your answer — examiners frequently ask for both, and they are usually different numbers.