C code for designing a Stacks that supports getMin() function with O(1) time and O(1) extra space.

To design a stack that supports the getMin() function with O(1) time complexity and O(1) extra space, we need to use a trick to keep track of the minimum element in the stacks at all times without using extra space storing minimum values.

Key idea:

Instead of maintaining a separate "minimum stack", we can manipulate the elements inside the stacks itself. We can modify the way we push and pop elements to always track the minimum element efficiently. This method ensures that we can always get the minimum element in constant time.

Approach:

We will utilize the following idea while performing push and pop operations:

1. Push Operation:

=> When a new element x is pushed onto the stack, we check whether it is smaller than or equal to the current minimum.

=> We can adjust the value of the element being pushed by storing the difference between the current minimum and the new element, ensuring that we always know the minimum element of the stack without using extra space.

2. Pop Operation:

=> When popping an element, if the element is less than or equal to the current minimum, we update the minimum value. The popped element will provide us with enough information to restore the correct minimum.

3. getMin Operation:

=> The getMin() function can simply return the current minimum element, which is always available at the top of the stack.

Key Points:

=> The idea is to adjust the stack values while performing the operations so that the minimum element can be inferred from the stack itself.

=> The stack will only store the adjusted values, and the minimum is updated as part of the normal operations.

=> We do not use extra space for a separate minimum stack, so the space complexity remains O(1).

Code Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
// Define the stack structure
struct Stack {
    int *arr; // Array to store stack elements
    int top; // Tracks the top of the stack.
```

```
    int capacity; // Maximum capacity of the stack
};
// Function to create a stack of given capacity.
struct Stack* createStack(int capacity) {
    struct Stack* stack = (struct Stack*) malloc(sizeof(
    struct Stack));
stack->capacity = capacity;
stack->top = -1;
stack->arr = (int*) malloc(capacity * sizeof
    (int));
    return stack;
}
// Function to check if the stack is full
int isFull(struct Stack* stack) {
    return stack->top == stack->capacity - 1;
}
// Function to check if the stack is empty.
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
// Function to push an element onto the stack
void push(struct Stack* stack, int x) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
```

```
        return;
    }

    // If stack is empty, push the element
    if (isEmpty(stack)) {
        stack->arr[++stack->top] = x;
    } else {
        int currentMin = getMin(stack);
        // If new element x is less than or equal to
        // current minimum, modify the element
        if (x <= currentMin) {
            stack->arr[++stack->top] = 2 * x -
                currentMin;
        } else {
            stack->arr[++stack->top] = x;
        }
    }
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    int topElement = stack->arr[stack->top--];
    // If the popped element is less than or equal to current
    // minimum, it was a modified value
```

```
    if (topElement <= getMin(stack)) {
        int currentMin = getMin(stack);
        // Restore the actual element before modification
        return currentMin;
    }
    return topElement;
}
// Function to get the minimum element in the stack
int getMin(struct Stack* stack) {
    if (isEmpty(stack)) {
        return INT_MAX; // Return a large number for
                        an empty stack.
    }
    int topElement = stack->arr[stack->top];
    int currentMin = (topElement <= getMin(stack)) ?
                     topElement : getMin(stack);
    return currentMin;
}
// Driver code to test the stack with getMin
int main() {
    struct Stack* stack = createStack(10);
    push(stack, 5);
    printf("Minimum element: %d\n", getMin(stack));
    // Output: 5
    push(stack, 3);
    printf("Minimum element: %d\n", getMin(stack));
    // Output: 3
```

```
push(stack, 7);
printf("Minimum element: %d\n", getMin(stack));
                                    // Output: 3

push(stack, 2);
printf("Minimum element: %d\n", getMin(stack));
                                    // Output: 2

pop(stack);
printf("Minimum element after pop: %d\n", getMin
        (stack)); // Output: 3.

pop(stack);
printf("Minimum element after pop: %d\n", getMin
        (stack)); // Output: 3

    return 0;

}
```

Explanation of Code:

1. push():

=> When pushing a new element x, if x is smaller than or equal to the current minimum, we store 2*x - min. This modification ensures that when we pop the element later, we can recover the actual value and keep track of the minimum efficiently.

=> If x is larger than the current minimum, we just push x directly onto the stack.

current minimum, it was a modified value (a special value to track the minimum).

→ To restore the current minimum, we return the current minimum and then compute the new minimum.

3. getMin():

→ This function returns the current minimum value from the stack, which is always at the top.

→ If the top value is modified (less than or equal to the current minimum), we update the minimum accordingly.

Example Walkthrough:

Let's walk through an example:

1. Push 5 → Stack: [5] → Min: 5.

2. Push 3 → Stack: [5, 3] → Min: 3 (3 is smaller than 5, so the value at the top is 2*3-5 = 1).

3. Push 7 → Stack: [5, 3, 7] → Min: 3 (7 is larger than 3, so we just push 7).

4. Push 2 → Stack: [5, 3, 7, 2] → Min: 2 (2 is smaller than 3, so the value at the top is 2*2-3 = 1).

Now, popping:

→ Pop the top element (2 → stack top) → Min remains 3.

→ Pop again → The element is 1 (a modified value), restore to 3.

Least Common Ancestor (LCA) In a Binary Tree

=> The least common Ancestor (LCA) of two nodes in a binary tree is defined as the deepest node that is an ancestor of both nodes.

=> An ancestor of a node is a node that is on the path from the root to that node. The LCA of two nodes n1 and n2 is the deepest node that is an ancestor of both n1 and n2. If either n1 or n2 is the ancestor of the other, then that node is considered as the LCA.

Key Concepts:

1. **Binary Tree:** A tree data structure where each node has at most two children (left and right).

2. **Ancestor:** A node A is an ancestor of node B if A is present on the path from the root to node B.

3. **Deepest Node:** The node that is farthest from the root in the tree. The LCA is the deepest node common to the paths to n1 and n2.

4. **Recursive Approach:** The LCA of two nodes can be found by traversing the tree recursively and checking for the following:

=> If the current node is one of the nodes (either n1 or n2), return it.

=> Recursive into the left and right subtrees.

=> If one node is found in the left subtree and the other is found in the right subtree, the current node is LCA.

=> If both nodes are present in same subtree, propagate

that node up the tree.

C code to Find LCA in a Binary Tree:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for the tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*) malloc(sizeof(
        struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to find the LCA of two nodes n1 and n2
struct Node* findLCA(struct Node* root, int n1, int n2)
{
    // Base case: If the root is NULL or matches one of the
    // nodes
    if (root == NULL || root->data == n1 || root->data == n2)
        return root;
```

```
// Recurse for the left and right subtrees
struct Node* leftLCA = findLCA(root->left, n1, n2);
struct Node* rightLCA = findLCA(root->right, n1, n2);

// If both left and right LCA are not NULL, the current node
// is the LCA
if (leftLCA && rightLCA)
    return root;

// Otherwise, return the non-null child
return (leftLCA != NULL) ? leftLCA : rightLCA;
}

// Main function to test the findLCA function
int main() {
    // Construct a simple binary tree
    struct Node* root = newNode(3);
    root->left = newNode(5);
    root->right = newNode(1);
    root->left->left = newNode(6);
    root->left->right = newNode(2);
    root->left->right->left = newNode(7);
    root->left->right->right = newNode(4);
    root->right->left = newNode(0);
    root->right->right = newNode(8);

    int n1 = 5, n2 = 1;
    struct Node* lca = findLCA(root, n1, n2);
    if (lca != NULL) {
        printf("LCA of %d and %d is %d\n", n1, n2,
```

```
lca->data);
} else {
    printf("LCA not found.\n");
}
return 0;
}
```

Explanation of Code:

1. Tree Node Structure:

=> The Node structure is defined with three fields: data (the value stored in the node), left (pointer to the left child), and right (pointer to the right child).

2. newNode Function:

=> This function creates a new node and returns a pointer to it. The data of the node is passed as an argument, and the left and right children are initialized to NULL.

3. findLCA function:

=> This function finds the LCA of two nodes n1 and n2 in the binary tree rooted at root.

=> Base Case: If the root is NULL or the root's data matches either n1 or n2, return the root.

=> Recursive calls: We recursively search for the LCA in the left and right subtrees. If the LCA is found in both subtrees, then the current root is the LCA.

=> Return the non-NULL child: If one of the recursive calls returns NULL, it means that node wasn't found

is on that subtree. Hence, we return the non-NULL child, which indicates that the LCA is on the other subtree.

4. Main Function:

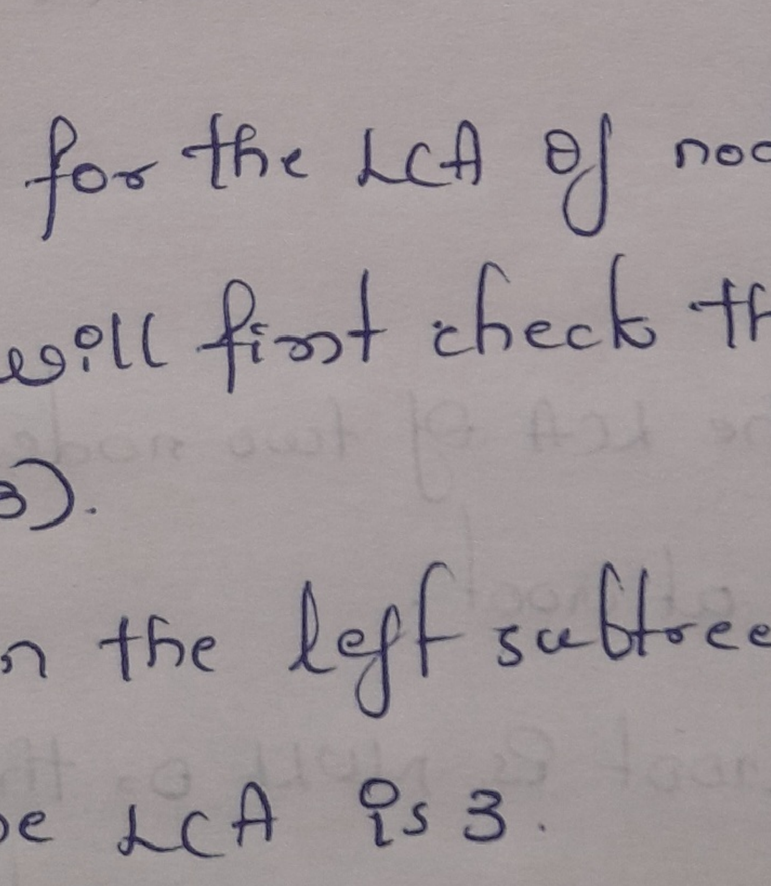
=> A simple binary tree is constructed.

-> The findLCA function is called with the root of the tree and two nodes (n1 = 5 and n2 = 1).

=> The result is printed.

Example:

For the tree structure:



If we are looking for the LCA of nodes 5 and 1:

=> The function will first check the left and right subtrees of the root (3).

-> It finds 5 in the left subtree and 1 in the right subtree, so the LCA is 3.

Output:

LCA of 5 and 1 is 3.