

1.IMPLEMENT STACK USING QUEUES

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

typedef struct StackItemBase {
    int value;
    struct StackItemBase* nextItemPtr;
} StackItem;

typedef struct {
    StackItem* head;
} MyStack;

MyStack* myStackCreate() {
    MyStack* newStack = (MyStack*)malloc(sizeof(MyStack));
    newStack->head = NULL;
    return newStack;
}

void myStackPush(MyStack* obj, int x) {
    StackItem* newHead = (StackItem*)malloc(sizeof(StackItem));
    newHead->value = x;
    newHead->nextItemPtr = obj->head;
    obj->head = newHead;
}

int myStackPop(MyStack* obj) {
    int result = obj->head->value;
    StackItem* newHead = obj->head->nextItemPtr;
    free(obj->head);
    obj->head = newHead;
    return result;
}

int myStackTop(MyStack* obj) {
    return obj->head->value;
}

bool myStackEmpty(MyStack* obj) {
    return obj->head == NULL;
}

void myStackFree(MyStack* obj) {
    while (obj->head != NULL) {
        StackItem* newHead = obj->head->nextItemPtr;
        free(obj->head);
        obj->head = newHead;
    }
    free(obj);
}
```

Problem List

Run

Submit

Premium

Description

Editorial

Solutions

Submissions

225. Implement Stack using Queues

Solved

Easy

Topics

Companies

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element `x` to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a queue, which means that only `push` to back, `peek/pop` from front, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

5.9K 46

</> Code

C Auto

Ln 1, Col 1

```
1 #include <stdbool.h>
2 #include <stdlib.h>
3
4 typedef struct StackItemBase {
5     int value;
6     struct StackItemBase* nextItemPtr;
7 } StackItem;
8
9 typedef struct {
10     StackItem* head;
11 } MyStack;
12
13 MyStack* myStackCreate() {
14     MyStack* newStack = (MyStack*)malloc(sizeof(MyStack));
15     newStack->head = NULL;
16 }
```

Saved to local

Testcase

Test Result

Accepted

Runtime: 4 ms

Case 1

Input

["MyStack", "push", "push", "top", "pop", "empty"]

2.IMPLEMENT QUEUE USING STACKS

```
#include<stdio.h>
typedef struct {
    int s1[100];
    int s2[100];
    int cnt1;
    int cnt2;
} MyQueue;

MyQueue* myQueueCreate()
{
    MyQueue* que = malloc(sizeof(MyQueue));
    que->cnt1 = 0;
    que->cnt2 = 0;
    return que;
}

void myQueuePush(MyQueue* obj, int x)
{
    obj->s1[obj->cnt1++] = x;
}

int myQueuePop(MyQueue* obj)
{
    if (obj->cnt1 == 0 && obj->cnt2 == 0) return -1;
    if (obj->cnt2 == 0)
    {
        while (obj->cnt1 > 0)
        {
            obj->s2[obj->cnt2++] = obj->s1[--obj->cnt1];
        }
    }
    int a = obj->s2[--obj->cnt2];
    return a;
}

int myQueuePeek(MyQueue* obj)
{
    if (obj->cnt2 == 0) return obj->s1[0];
    return obj->s2[obj->cnt2 - 1];
}

bool myQueueEmpty(MyQueue* obj)
{
    if (obj->cnt1 == 0 && obj->cnt2 == 0) return true;
    return false;
}

void myQueueFree(MyQueue* obj)
{
    free(obj);
}
```

Problem List

Run

Submit

0

Premium

Description

Editorial

Solutions

Submissions

232. Implement Queue using Stacks

EasyTopicsCompanies

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element `x` to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a stack, which means only `push` to `top`, `peek/pop` from `top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

7.4K89

</> Code

C Auto

```
1 #include<stdio.h>
2 typedef struct {
3     int s1[100];
4     int s2[100];
5     int cnt1;
6     int cnt2;
7 } MyQueue;
8
9 MyQueue* myQueueCreate()
10 {
11     MyQueue* que = malloc(sizeof(MyQueue));
12     que->cnt1 = 0;
13     que->cnt2 = 0;
14     return que;
15 }
```

Saved to localLn 1, Col 1

Testcase

Test Result

AcceptedRuntime: 0 ms

Case 1

Input

["MyQueue", "push", "push", "peek", "pop", "empty"]

3.DESIGN CIRCULAR QUEUE

```
#INCLUDE<STDIO.H>
typedef struct
{
    int* queue;
    int front;
    int rear;
    int size;
} MyCircularQueue;

bool myCircularQueueIsFull(MyCircularQueue* obj);
bool myCircularQueueIsEmpty(MyCircularQueue* obj);

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* ans = malloc(sizeof(MyCircularQueue));
    ans->queue = (int*)malloc((k+1) * sizeof(int));
    ans->front = 0;
    ans->rear = 0;
    ans->size = k+1;
    return ans;
}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value)
{
    if(myCircularQueueIsFull(obj))
        return false;
    else{
        obj->queue[obj->rear] = value;
        obj->rear = (obj->rear+1) % (obj->size);
        return true;
    }
}

bool myCircularQueueDeQueue(MyCircularQueue* obj)
{
    if(myCircularQueueIsEmpty(obj))
        return false;
    else{
        obj->front = (obj->front + 1)% obj->size;
        return true;
    }
}

int myCircularQueueFront(MyCircularQueue* obj)
{
    if(myCircularQueueIsEmpty(obj))
        return -1;
    else
        return obj->queue[obj->front];
}

int myCircularQueueRear(MyCircularQueue* obj)
{
    if(myCircularQueueIsEmpty(obj))
        return -1;
    else
        return obj->queue[(obj->rear-1+obj->size) % obj->size];
}

bool myCircularQueueIsEmpty(MyCircularQueue* obj)
{
    if((obj->front) == (obj->rear))
        return true;
    else
        return false;
}

bool myCircularQueueIsFull(MyCircularQueue* obj)
```

```

{

    return ( (obj->rear + 1) % obj->size == obj->front);

}

void myCircularQueueFree(MyCircularQueue* obj)
{
    free(obj->queue);
    free(obj);
}

```

Problem List

Description
Editorial
Solutions
Submissions

622. Design Circular Queue

Medium Topics Companies

Solved

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the `MyCircularQueue` class:

- `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`.
- `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`.
- `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`.
- `boolean enqueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful.
- `boolean dequeue()` Deletes an element from the circular queue. Return `true` if the operation is successful.
- `boolean isEmpty()` Checks whether the circular queue is empty or not.
- `boolean isFull()` Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

3.5K 20

Code

```

1 #include <stdio.h>
2 typedef struct {
3     int* queue;
4     int front;
5     int rear;
6     int size;
7 } MyCircularQueue;
8
9 bool myCircularQueueIsFull(MyCircularQueue* obj);
10 bool myCircularQueueIsEmpty(MyCircularQueue* obj);
11
12 MyCircularQueue* myCircularQueueCreate(int k) {
13     MyCircularQueue* ans = malloc(sizeof(MyCircularQueue));
14     ans->queue = (int*)malloc((k + 1) * sizeof(int));
15     ans->front = 0;

```

Saved to local Ln 1, Col 1

Testcase

Test Result

Accepted Runtime: 0 ms

Case 1

Input:

```

["MyCircularQueue","enqueue","enqueue","enqueue","enqueue","Rear","isFull","dequeue","enqueue","Rear"]

```

4. BUILD ARRAY USING STACK OPERATIONS

```
#include <stdlib.h>

#include <string.h>

char** buildArray(int* target, int targetSize, int n, int* returnSize) {
    char** targetArray = (char**)malloc(2 * n * sizeof(char*));

    size_t current = -1;
    size_t internalCounter = 1;
    for (size_t i = 0; i < targetSize; i++) {
        while (internalCounter < target[i]) {
            current++;
            targetArray[current] = (char*)calloc(5, sizeof(char));
            strncpy(targetArray[current], "Push", 4);

            current++;
            targetArray[current] = (char*)calloc(5, sizeof(char));
            strncpy(targetArray[current], "Pop", 4);

            internalCounter++;
        }

        current++;
        targetArray[current] = (char*)calloc(5, sizeof(char));
        strncpy(targetArray[current], "Push", 4);

        if (target[targetSize - 1] == internalCounter) {
            break;
        }

        internalCounter++;
    }

    *returnSize = current + 1;
    return targetArray;
}
```

The screenshot displays a coding platform interface. On the left, the problem description for "1441. Build an Array With Stack Operations" is visible. It states that given an integer array `target` and an integer `n`, the goal is to build an array of stack operations ("Push" and "Pop") such that the stack (from bottom to top) equals `target`. The rules specify that "Push" adds the next integer from the stream (1 to n) to the stack, and "Pop" removes the top element. The solution must return the sequence of operations, or any valid sequence if multiple exist.

On the right, a code editor shows the implementation of the `buildArray` function. The code uses a loop to iterate through the `target` array. For each element, it pushes elements from the stream until the stack matches the target element, then pops them. Finally, it pushes the target element itself. The function returns the constructed array of operations and updates the `returnSize`.

Below the code editor, the "Test Result" section shows that the solution is "Accepted" with a runtime of 3 ms. The input for the test case is `target = [1, 3]`.

5.SPLIT LINKED LIST IN PARTS

```
#include<stdio.h>
int count_nodes(struct ListNode *root)
{
    struct ListNode * trav_ptr = root;
    int count = 0;
    while(trav_ptr)
    {
        count++;
        trav_ptr = trav_ptr->next;
    }
    return count;
}

struct ListNode** splitListToParts(struct ListNode* root, int k, int* returnSize)
{
    int no_of_nodes = count_nodes(root);

    int no_of_elements_in_each_half;
    int rem;

    if( k <= no_of_nodes)
    {
        no_of_elements_in_each_half = (no_of_nodes/k);
        rem = (no_of_nodes%k);
    }
    else
    {
        no_of_elements_in_each_half = 1;
        rem = 0;
    }

    struct ListNode ** list_ptr = (struct ListNode **)malloc(sizeof(struct ListNode *)*k);

    int i=0;
    for(i=0; i<k; i++)
    {
        memset((list_ptr+i), NULL, sizeof(struct ListNode *));
    }

    int list_ptr_index = 0;

    struct ListNode * trav_ptr = root;
    struct ListNode * next_ptr = root;
    int count = 1;

    while(trav_ptr)
    {
        if(count == no_of_elements_in_each_half)
        {
            if(rem != 0)
            {
                rem--;
                count = 1;
                trav_ptr = trav_ptr->next;
                next_ptr = trav_ptr->next;
                trav_ptr->next = NULL;
                *(list_ptr + list_ptr_index) = root;
                list_ptr_index++;

                root = next_ptr;
                trav_ptr = next_ptr;
            }
            else
            {

```



```

        count = 1;
        next_ptr = trav_ptr->next;
        trav_ptr->next = NULL;

        *(list_ptr + list_ptr_index) = root;
        list_ptr_index++;

        root = next_ptr;
        trav_ptr = next_ptr;
    }
}
else
{
    count++;
    trav_ptr = trav_ptr->next;
}
}

*returnSize = k;

return list_ptr;

}

```

Problem List

Run
Submit

0
Premium

Description
Editorial
Solutions
Submissions

725. Split Linked List in Parts

Medium

Topics: Companies: Hint

Given the `head` of a singly linked list and an integer `k`, split the linked list into `k` consecutive linked list parts.

The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.

The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.

Return an array of the `k` parts.

Example 1:

```

graph LR
    1((1)) --> 2((2))
    2 --> 3((3))

```

Input: head = [1,2,3], k = 5
Output: [[1], [2], [3], [1], [1]]

3.9K 49

Code

```

1 #include<stdio.h>
2 int count_nodes(struct ListNode *root)
3 {
4     struct ListNode * trav_ptr = root;
5     int count = 0;
6     while(trav_ptr)
7     {
8         count++;
9         trav_ptr = trav_ptr->next;
10    }
11    return count;
12 }
13
14 struct ListNode** splitListToParts(struct ListNode* root, int k, int* returnSize)
15 {

```

Saved to local Ln 89, Col 2

Testcase Test Result

Accepted Runtime: 5 ms

Case 1 Case 2

Input

```

head =
[1,2,3]

```

6. REVERSE LINKED LIST

```
#include<stdio.h>
struct ListNode* reverseList(struct ListNode* head)
{
    struct ListNode* temp=NULL;
    struct ListNode* previous=NULL;

    while(head!=NULL)
    {
        temp=head->next;
        head->next=previous;
        previous=head;
        head=temp;
    }
    return previous;
}
```

206. Reverse Linked List

Given the `head` of a singly linked list, reverse the list, and return the reversed list.

Example 1:

Diagram illustrating the reversal of a linked list. The original list is 1 → 2 → 3 → 4 → 5. The reversed list is 5 → 4 → 3 → 2 → 1.

Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]

Example 2:

Code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ListNode* reverseList(struct ListNode* head) {
5     struct ListNode* curr = head;
6     struct ListNode* prev = NULL;
7     struct ListNode* next;
8
9     if(!head || !head->next) {
10         return head;
11     }
12
13     while (curr) {
14         next = curr->next; // Save the next node
15         curr->next = prev; // Reverse the link
```

Test Result: Accepted Runtime: 6 ms

Case 1 Case 2 Case 3

Input: head = [1,2,3,4,5]

7. NUMBER OF RECENT CALLS

```
#include<stdio.h>
typedef struct
{
    int before;
    int now;
    int ping[10000];
} RecentCounter;

RecentCounter *recentCounterCreate()
{
    RecentCounter *obj = malloc(sizeof(RecentCounter));
    memset(&obj->ping, 0, 10000);
    obj->before = 0;
    obj->now = 0;
    return obj;
}

int recentCounterPing(RecentCounter *obj, int t)
{
    obj->ping[obj->now++] = t;
    while (obj->ping[obj->before] < t - 3000) obj->before++;
    return obj->now - obj->before;
}

void recentCounterFree(RecentCounter *obj)
{
    free(obj);
}
```

The screenshot shows the LeetCode interface for problem 933, "Number of Recent Calls". The problem description states that a `RecentCounter` class counts requests within a 3000ms window. The provided C++ solution implements this with a sliding window array. The test results show the solution is "Accepted" with a runtime of 0 ms.

933. Number of Recent Calls Solved ✓

Easy Topics Companies

You have a `RecentCounter` class which counts the number of recent requests within a certain time frame.

Implement the `RecentCounter` class:

- `RecentCounter()` Initializes the counter with zero recent requests.
- `int ping(int t)` Adds a new request at time `t`, where `t` represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range `[t - 3000, t]`.

It is **guaranteed** that every call to `ping` uses a strictly larger value of `t` than the previous call.

Example 1:

Input
["RecentCounter", "ping", "ping", "ping", "ping"]
[[], [1], [100], [3001], [3002]]

Output
[null, 1, 2, 3, 3]

Explanation
`RecentCounter recentCounter = new RecentCounter();`

```
1 #include <stdio.h>
2 typedef struct {
3     int before;
4     int now;
5     int ping[10000];
6 } RecentCounter;
7
8 RecentCounter* recentCounterCreate() {
9     RecentCounter* obj = malloc(sizeof(RecentCounter));
10    memset(&obj->ping, 0, 10000);
11    obj->before = 0;
12    obj->now = 0;
13    return obj;
14 }
15
```

Accepted Runtime: 0 ms

Case 1

Input
["RecentCounter", "ping", "ping", "ping", "ping"]

8. RECTANGLE AREA

```
#include<stdio.h>

int computeArea(int ax1, int ay1, int ax2, int ay2, int bx1, int by1, int bx2, int by2){
    int cx1=0, cx2=0, cy1=0, cy2=0;

    if(ax2>bx1 && ax1<bx2){
        cx1 = ax1>bx1?ax1:bx1;
        cx2 = ax2<bx2?ax2:bx2;
    }
    if(ay2>by1 && ay1<by2){
        cy1 = ay1>by1?ay1:by1;
        cy2 = ay2<by2?ay2:by2;
    }

    return ((ax2-ax1)*(ay2-ay1))
        + ((bx2-bx1)*(by2-by1))
        - ((cx2-cx1)*(cy2-cy1));
}
```

The screenshot displays a coding platform interface. On the left, the problem description for "223. Rectangle Area" is shown, including a diagram of two overlapping rectangles on a 2D plane. The first rectangle is purple with its bottom-left corner at $(ax1, ay1) : (-3, 0)$ and its top-right corner at $(ax2, ay2) : (3, 4)$. The second rectangle is red with its bottom-left corner at $(bx1, by1) : (0, 0)$ and its top-right corner at $(bx2, by2) : (9, 2)$. The diagram shows the intersection of the two rectangles. On the right, the code editor shows the C code for the `computeArea` function, which calculates the total area covered by the two rectangles by summing their individual areas and subtracting the overlapping area. The code is saved to local storage. Below the code editor, the test result section shows "Accepted" with a runtime of 2 ms, and the input for Case 1 is `ax1 = -3`.

223. Rectangle Area

Given the coordinates of two **rectilinear** rectangles in a 2D plane, return the total area covered by the two rectangles.

The first rectangle is defined by its **bottom-left** corner $(ax1, ay1)$ and its **top-right** corner $(ax2, ay2)$.

The second rectangle is defined by its **bottom-left** corner $(bx1, by1)$ and its **top-right** corner $(bx2, by2)$.

Example 1:

Accepted Runtime: 2 ms

Case 1

Input

ax1 = -3

9.CIRCULAR SENTENCE

```
#include<stdio.h>
bool isCircularSentence(char * sentence)
{
    if (sentence[0] != sentence[strlen(sentence) - 1])
        return false;
    for (int i = 0; i <= strlen(sentence); i++)
    {
        if (sentence[i] == ' ' && sentence[i - 1] != sentence[i + 1])
            return false;
    }
    return true;
}
```

The screenshot displays the LeetCode interface for problem 2490, "Circular Sentence". The left pane contains the problem description, which defines a circular sentence as one where the last character of a word equals the first character of the next word, and the last character of the last word equals the first character of the first word. It provides examples of valid and invalid sentences. The right pane shows a C++ solution for the problem, which checks for spaces and compares the first and last characters of the sentence. Below the code, the test results are shown as "Accepted" with a runtime of 0 ms. The input for the test case is "leetcode exercises sound delightful".

2490. Circular Sentence

Easy Topics Companies Hint

A **sentence** is a list of words that are separated by a **single** space with no leading or trailing spaces.

- For example, "Hello World", "HELLO", "hello world hello world" are all sentences.

Words consist of **only** uppercase and lowercase English letters. Uppercase and lowercase English letters are considered different.

A sentence is **circular** if:

- The last character of a word is equal to the first character of the next word.
- The last character of the last word is equal to the first character of the first word.

For example, "leetcode exercises sound delightful", "eetcode", "leetcode eats soul" are all circular sentences. However, "Leetcode is cool", "happy Leetcode", "Leetcode" and "I like Leetcode" are **not** circular sentences.

Given a string `sentence`, return `true` if it is circular. Otherwise, return `false`.

Example 1:

Input: `sentence = "leetcode exercises sound delightful"`
Output: `true`

```
1 #include<stdio.h>
2 bool isCircularSentence(char * sentence)
3 {
4     if (sentence[0] != sentence[strlen(sentence) - 1])
5         return false;
6     for (int i = 0; i <= strlen(sentence); i++)
7     {
8         if (sentence[i] == ' ' && sentence[i - 1] != sentence[i + 1])
9             return false;
10    }
11    return true;
12 }
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

sentence =

"leetcode exercises sound delightful"

10.BINARY TREE PATHS

```
#include<stdio.h>

int resIdx;
void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res);
char ** binaryTreePaths(struct TreeNode* root, int* returnSize)
{
    if (root == NULL)
    {
        *returnSize = 0;
        return NULL;
    }
    resIdx = 0;
    int i;
    char ** res = (char **)malloc(sizeof(char *) * 40);
    int ls[150];
    findpath(root,&ls[0],0,res);
    *returnSize = resIdx;
    return &res[0];
}

void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res)
{
    char temp[100];
    int l=0,i=0;
    if (root->left == NULL && root->right == NULL)
    {
        ls[ls_idx] = root->val;
        ls_idx+=1;
        res[resIdx] = (char *)malloc(sizeof(char) * 100);
        while (i < ls_idx)
        {
            if (i==0)
            {
                l = l + sprintf(&temp[l], "%d", ls[i]);
            }
            else
            {
                l = l + sprintf(&temp[l], "->%d", ls[i]);
            }
            i++;
        }
        strcpy(res[resIdx],temp);
        resIdx++;
        return;
    }
    ls[ls_idx] = root->val;
    if (root->left != NULL)
    {
        findpath(root->left,ls,ls_idx+1,res);
    }

    if (root->right != NULL)
    {
        findpath(root->right,ls,ls_idx+1,res);
    }
    return;
}
```

Problem List

Run

Submit

Premium

Description

Editorial

Solutions

Submissions

257. Binary Tree Paths

EasyTopicsCompanies

Given the `root` of a binary tree, return *all root-to-leaf paths in **any order***.

A **leaf** is a node with no children.

Example 1:

```
graph TD; 1((1)) --- 2((2)); 1 --- 3((3)); 2 --- 5((5));
```

6.4K28

Solved

</> Code

C Auto

```
1 #include<stdio.h>
2 int resIdx;
3 void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res);
4 char ** binaryTreePaths(struct TreeNode* root, int* returnSize)
5 {
6
7     if (root == NULL)
8     {
9         *returnSize = 0;
10        return NULL;
11    }
12    resIdx = 0;
13    int i;
14    char ** res = (char **)malloc(sizeof(char *) * 40);
15    int ls[150];
```

Saved to localLn 1, Col 1

Testcase

Test Result

Accepted

Runtime: 2 ms

Case 1Case 2

Input

root =

[1,2,3,null,5]

PROBLEM: **DESIGN A PARKING SYSTEM**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct {
    int big;
    int medium;
    int small;
} ParkingSystem;

// to initiate the parking sytems struct
ParkingSystem* parkingSystemCreate(int big, int medium, int small) {

    ParkingSystem *park = calloc(1, sizeof(ParkingSystem));
    if (park == NULL){
        perror("Malloc park");
        return NULL;
    }

    // assigning the slot size for each parking lot type
    park->big = big;
    park->medium = medium;
    park->small = small;

    return park;
}

// Add the car to the parking lot. Return true if its able, false otherwise.
bool parkingSystemAddCar(ParkingSystem* obj, int carType) {

    // If the carType is big, reduce the big slot
    if (carType == 1 && obj->big != 0){
        obj->big -= 1;
        return true;
    }

    // If the carType is medium, reduce the medium slot
    if (carType == 2 && obj->medium != 0){
        obj->medium -= 1;
        return true;
    }

    // If the carType is small, reduce the small slot
    if (carType == 3 && obj->small != 0){
        obj->small -= 1;
        return true;
    }

    return false;
}

// Free the malloc
void parkingSystemFree(ParkingSystem* obj) {
```



```

    free(obj);
}

/**
 * Your ParkingSystem struct will be instantiated and called as such:
 * ParkingSystem* obj = parkingSystemCreate(big, medium, small);
 * bool param_1 = parkingSystemAddCar(obj, carType);
 *
 * parkingSystemFree(obj);
 */

```

1603. Design Parking System

Design a parking system for a parking lot. The parking lot has three kinds of parking spaces: big, medium, and small, with a fixed number of slots for each size.

Implement the `ParkingSystem` class:

- `ParkingSystem(int big, int medium, int small)` Initializes object of the `ParkingSystem` class. The number of slots for each parking space are given as part of the constructor.
- `bool addCar(int carType)` Checks whether there is a parking space of `carType` for the car that wants to get into the parking lot. `carType` can be of three kinds: big, medium, or small, which are represented by 1, 2, and 3 respectively. **A car can only park in a parking space of its `carType`.** If there is no space available, return `false`, else park the car in that size space and return `true`.

Example 1:

Input
 ["ParkingSystem", "addCar", "addCar", "addCar", "addCar"]
 [[1, 1, 0], [1], [2], [3], [1]]

Output
 [null, true, true, false, true]

```

1 #include<stdio.h>
2 #include<stdlib.h>
3 >typedef struct { ...
4 } ParkingSystem;
5
6 // to initiate the parking systems struct
7 ParkingSystem* parkingSystemCreate(int big, int medium, int small) {
8
9     ParkingSystem *park = calloc(1, sizeof(ParkingSystem));
10    if (park == NULL){
11        perror("Malloc park");
12        return NULL;
13    }
14 }

```

Accepted Runtime: 0 ms

Case 1

Input

["ParkingSystem","addCar","addCar","addCar","addCar"]

A "real-time problem on queues" could mean a few things, but in a general sense, it could refer to a problem that is designed to mimic a real-world scenario where a queue data structure might be used. Queues are commonly used in real-world applications for tasks like job scheduling, handling tasks in a first-come-first-served (FIFO) manner, or managing resources like printers in an office.

Here is a simple example of a "real-time problem on queues"

Problem: **Design a Parking System**

Design a parking system for a parking lot that has three kinds of parking spaces: **big**, **medium**, and **small**, with respective capacities of **5**, **10**, and **20** parking spaces. Implement the ParkingSystem class:

- `ParkingSystem(int big, int medium, int small)` Initializes object of the ParkingSystem class. The number of each type of space is given as **big**, **medium**, and **small**, respectively.
- **`bool addCar(int carType)`** Checks whether there is a parking space of carType for the car that wants to get into the parking lot. carType can be of three kinds: **big**, **medium**, or **small**. The parking lot has at least one empty parking space for each car type.

You need to return true if there is a parking space for the car, otherwise, return false.

Input:

```
["ParkingSystem", "addCar", "addCar", "addCar", "addCar"] [[1, 1, 0], [1], [2], [3], [1]]
```

Output:

```
[null, true, true, false, false]
```

Explanation:

```
ParkingSystem parkingSystem = new ParkingSystem(1, 1, 0); parkingSystem.addCar(1); // return true because there is 1 available slot for a big car parkingSystem.addCar(2); // return true because there is 1 available slot for a medium car parkingSystem.addCar(3); // return false because there is no available slot for a small car parkingSystem.addCar(1); // return false because there is no available slot for a big car. It is already occupied.
```

Note:

- The number of spaces is in the range [0, 1000]
- carType is 1, 2, or 3
- Each carType may be either a big car, medium car, or a small car

This problem simulates a real-world scenario where a queue data structure might be used to manage parking space availability. It requires implementing a queue to track the available parking spaces for each car type and to check if there is an available parking space for a given car type when a car arrives at the parking lot.