

# RAMAIAH INSTITUTE OF TECHNOLOGY

MSRIT NAGAR, BENGALURU, 560054



A Report on

## LEET CODE PROGRAMS

*Submitted in partial fulfilment of the OTHER COMPONENT requirements as a part of the Data Structures Lab subject with code ISL36 for the III Semester of degree of **Bachelor of Engineering in Information Science and Engineering***

Submitted by

**SAYEED KHAN**

**(1MS23IS402-T)**

Under the Guidance of

Faculty In-charge

**MR. Shivananda S.**

Assistant Professor

Dept. of ISE

**Department of Information Science and Engineering**

**Ramaiah Institute of Technology**

2023 - 2024

SLno	Program name
1	STACK EASY
2	FiIND THE MINIMUM NUMBER OF FIBONACCI NUMBERS WHOSE SUM IS K MEDIUM
3	CIRCULAR QUEUE MEDIUM
4	IMPLEMENT QUEUE USING STACK EASY
5	LINKED LIST EASY
6	LINKED LIST MEDIUM
7	BINARY TREE EASY
8	GRAPH MEDIUM
9	ADD BINARY TREE EASY
10	DESIGN HASHMAP MEDIUM
11	<u>REAL WORLD PROBLEM</u>

# 1 STACK EASY

## 1381. Design a Stack With Increment Operation

Medium

Topics

Companies

Hint

Design a stack that supports increment operations on its elements.

Implement the `CustomStack` class:

- `CustomStack(int maxSize)` Initializes the object with `maxSize` which is the maximum number of elements in the stack.
- `void push(int x)` Adds `x` to the top of the stack if the stack has not reached the `maxSize`.
- `int pop()` Pops and returns the top of the stack or `-1` if the stack is empty.
- `void inc(int k, int val)` Increments the bottom `k` elements of the stack by `val`. If there are less than `k` elements in the stack, increment all the elements in the stack.

### Example 1:

#### Input

```
["CustomStack","push","push","pop","push","push","push","increment","increment","pop","pop","pop","pop"]
```

```
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[[],[[,[[
```

#### Output

```
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
```

The screenshot displays the LeetCode problem page for 'Design a Stack With Increment Operation'. The problem is marked as 'Medium' and has been solved by the user 'sayeedkhan' on February 19, 2024. The solution is implemented in C, using a struct to represent the stack with fields for 'top', 'arr', and 'max'. The 'push' method checks if the stack is full before adding an element, and the 'inc' method increments the bottom 'k' elements. The 'pop' method returns the top element or -1 if the stack is empty. The runtime is 28 ms, and the memory usage is 13.53 MB, both of which beat 72.73% of other submissions. A bar chart shows the performance distribution. The test results section shows that the solution passed all test cases, including the provided example input.

```
typedef struct {
    int top;
    int* arr;
    int max;
} CustomStack;

CustomStack* customStackCreate(int maxSize) {
    CustomStack* stack = (CustomStack*)malloc(sizeof(CustomStack));
    stack->arr = (int*)malloc(sizeof(int)*maxSize);
    stack->top = -1;
    stack->max = maxSize;
    return stack;
}

void customStackPush(CustomStack* obj, int x) {
    if (obj->top != obj->max-1) {
        obj->arr[++obj->top] = x;
    }
}

int customStackPop(CustomStack* obj) {
    if (obj->top > -1) {
        return obj->arr[obj->top--];
    }
    return -1;
}

void customStackInc(CustomStack* obj, int k, int val) {
    for (int i = 0; i < k; i++) {
        obj->arr[i] += val;
    }
}
```

Accepted Runtime: 0 ms

Case 1

Input

```
["CustomStack","push","push","pop","push","push","push","increment","increment","pop","pop","pop","pop"]
```

# STACK EASY CODE

```
typedef struct {
    int top;
    int* arr;
    int max;
} CustomStack;

CustomStack* customStackCreate(int maxSize) {
    CustomStack* stack=(CustomStack*)malloc(sizeof(CustomStack));
    stack->arr=(int*)malloc(sizeof(int)*maxSize);
    stack->top=-1;
    stack->max=maxSize;
    return stack;
}

void customStackPush(CustomStack* obj, int x) {
    if(obj->top!=obj->max-1){
        obj->arr[++obj->top]=x;}
}

int customStackPop(CustomStack* obj) {
    if(obj->top!=-1){
        return -1;
    }
    return obj->arr[obj->top--];
}

void customStackIncrement(CustomStack* obj, int k, int val) {
    for(int i =0;i<k;i++){
        if(i<obj->max){
            obj->arr[i]=obj->arr[i]+val;
        }
    }
}

void customStackFree(CustomStack* obj) {
    free(obj);
}
```

## 2 Find the Minimum Number of Fibonacci Numbers Whose Sum Is K Medium

### 1414. Find the Minimum Number of Fibonacci Numbers Whose Sum Is K

Solved 

Medium

Topics

Companies

Hint

Given an integer  $k$ , return the minimum number of Fibonacci numbers whose sum is equal to  $k$ . The same Fibonacci number can be used multiple times.

The Fibonacci numbers are defined as:

- $F_1 = 1$
- $F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$  for  $n > 2$ .

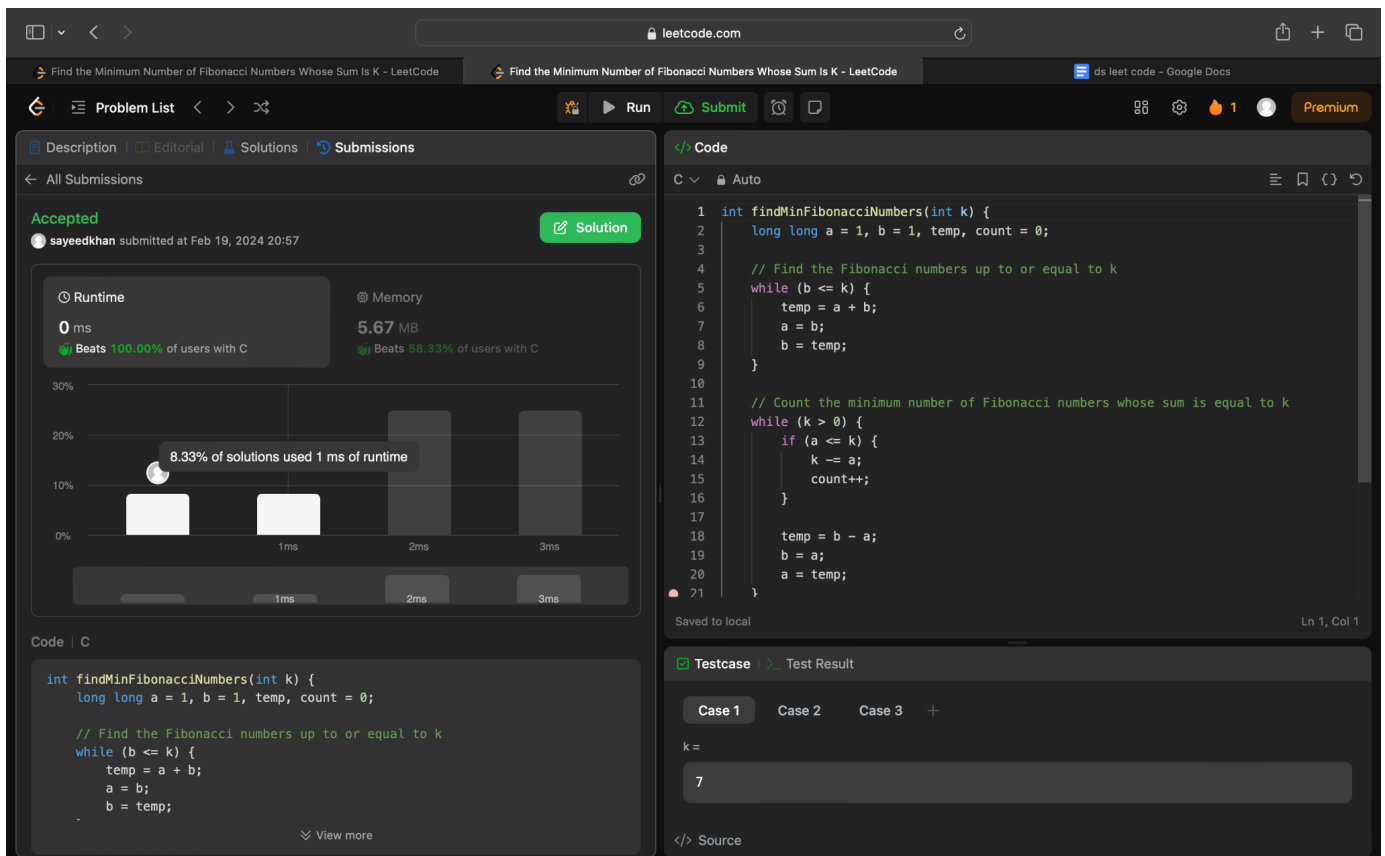
It is guaranteed that for the given constraints we can always find such Fibonacci numbers that sum up to  $k$ .

Example 1:

**Input:**  $k = 7$

**Output:** 2

**Explanation:** The Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, ...  
For  $k = 7$  we can use  $2 + 5 = 7$ .



The screenshot shows the LeetCode interface for the problem 'Find the Minimum Number of Fibonacci Numbers Whose Sum Is K'. The problem is marked as 'Solved' and 'Medium'. The description states: 'Given an integer  $k$ , return the minimum number of Fibonacci numbers whose sum is equal to  $k$ . The same Fibonacci number can be used multiple times. The Fibonacci numbers are defined as:  $F_1 = 1$ ,  $F_2 = 1$ , and  $F_n = F_{n-1} + F_{n-2}$  for  $n > 2$ . It is guaranteed that for the given constraints we can always find such Fibonacci numbers that sum up to  $k$ .'

**Example 1:**  
**Input:**  $k = 7$   
**Output:** 2  
**Explanation:** The Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, ...  
For  $k = 7$  we can use  $2 + 5 = 7$ .

The solution is implemented in C++:

```
int findMinFibonacciNumbers(int k) {  
    long long a = 1, b = 1, temp, count = 0;  
  
    // Find the Fibonacci numbers up to or equal to k  
    while (b <= k) {  
        temp = a + b;  
        a = b;  
        b = temp;  
    }  
  
    // Count the minimum number of Fibonacci numbers whose sum is equal to k  
    while (k > 0) {  
        if (a <= k) {  
            k -= a;  
            count++;  
        }  
  
        temp = b - a;  
        b = a;  
        a = temp;  
    }  
}
```

The submission is accepted, with a runtime of 0 ms (beats 100.00% of users with C) and memory usage of 5.67 MB (beats 58.33% of users with C). A bar chart shows that 8.33% of solutions used 1 ms of runtime.

The test case input is  $k = 7$ , and the output is 2.

# Fibonacci Numbers Whose Sum Is K **CODE**

```
int findMinFibonacciNumbers(int k) {
    long long a = 1, b = 1, temp, count = 0;

    // Find the Fibonacci numbers up to or equal to k
    while (b <= k) {
        temp = a + b;
        a = b;
        b = temp;
    }

    // Count the minimum number of Fibonacci numbers whose sum is equal to k
    while (k > 0) {
        if (a <= k) {
            k -= a;
            count++;
        }

        temp = b - a;
        b = a;
        a = temp;
    }

    return count;
}
```

## 3 CIRCULAR QUEUE MEDIUM

### 622. Design Circular Queue

Medium

Topics

Companies

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.

The screenshot shows the LeetCode interface for the problem 'Design Circular Queue'. The left sidebar displays the problem description, a submission status of 'Accepted' for user 'sayeedkhan', and a performance graph showing a runtime of 30 ms, which beats 30.03% of users with C. The main area shows the C code implementation of the `MyCircularQueue` class. The code includes a `typedef struct` for the queue, with `front`, `rear`, and `obj` pointers, and a `n` integer for the size. The implementation includes methods for `Front`, `Rear`, `enqueue`, `dequeue`, `isEmpty`, and `isFull`. The right sidebar shows the test case results, indicating 'Accepted' with a runtime of 2 ms.

```
typedef struct
{
    int front;
    int rear;
    int *obj;
    int n;
} MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* obj = (MyCircularQueue*) malloc(sizeof(MyCircularQueue));
    obj->n = k;
    obj->front = -1;
    obj->rear = -1;
    obj->obj = (int*) malloc(k * sizeof(int));
    return obj;
}

bool myCircularQueueIsEmpty(MyCircularQueue* q)
{
    if(q->front == -1)
        return true;
    return false;
}

bool myCircularQueueIsFull(MyCircularQueue* q)
{
    if((q->rear + 1) % q->n == q->front)
        return true;
    return false;
}

void myCircularQueueFree(MyCircularQueue* obj)
{
    free(obj);
}
```

# CIRCULAR QUEUE MEDIUM CODE

```
typedef struct
{
    int front;
    int rear;
    int *a;
    int n;
} MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* q = malloc(sizeof *q);
    q->a = (int*) malloc(sizeof(int) * k);
    q->front = -1;
    q->rear = -1;
    q->n = k;
    return q;
}

bool myCircularQueueEnQueue(MyCircularQueue* q, int value)
{
    if((q->rear + 1) % q->n == q->front)
        return false;
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear = (q->rear + 1) % q->n;
        q->a[q->rear] = value;
        return true;
    }
}

bool myCircularQueueDeQueue(MyCircularQueue* q)
{
    if(q->front == -1)
        return false;
    else
    {
        if (q->front == q->rear)
        {
            q->front = -1;
        }
    }
}
```



```
q->rear = -1;
}
else
q->front = (q->front + 1) % q->n;
return true;
}
}

int myCircularQueueFront(MyCircularQueue* q)
{
if(q->front==-1)
return -1;
return q->a[q->front];
}

int myCircularQueueRear(MyCircularQueue* q)
{
if(q->front==-1)
return -1;
return (q->a[q->rear]);
}

bool myCircularQueueIsEmpty(MyCircularQueue* q)
{
if(q->front==-1)
return true;
return false;
}

bool myCircularQueueIsFull(MyCircularQueue* q)
{
if((q->rear + 1) % q->n == q->front)
return true;
return false;
}

void myCircularQueueFree(MyCircularQueue* obj)
{
free(obj);
}
```

# 4 Implement Queue using Stacks **EASY**

## 232. Implement Queue using Stacks

Easy

Topics

Companies

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.

The screenshot shows a LeetCode submission interface for the problem "Implement Queue using Stacks". The submission is in C and has been accepted. The left sidebar shows the problem description, editorial, solutions, and submissions. The main area displays the submission details, including runtime (0 ms), memory (5.52 MB), and a performance graph. The right sidebar shows the code editor with the C implementation of the `MyQueue` class. The code includes functions for `push`, `pop`, `peek`, `empty`, and `free`. The bottom right shows the test result for "Case 1", which is "Accepted" with a runtime of 2 ms. The input for the test case is `["MyQueue", "push", "push", "peek", "pop", "empty"]`.

```
typedef struct {
    int ar[100];
    int head;
    int tail;
    int cnt;
} MyQueue;

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

int myQueuePop(MyQueue* obj) {
    return obj->ar[obj->head];
}

int myQueuePeek(MyQueue* obj) {
    if(obj == NULL) return NULL;
    return obj->ar[obj->head];
}

bool myQueueEmpty(MyQueue* obj) {
    if(obj == NULL) return false;
    return (obj->cnt == 0);
}

void myQueueFree(MyQueue* obj) {
    if(obj == NULL) return;
    free(obj);
}
```

# Implement Queue using Stacks **EASY** CODE

```
typedef struct {
int ar[100];
int head;
int tail;
int cnt;
} MyQueue;

MyQueue* myQueueCreate() {
MyQueue* obj = malloc(sizeof(MyQueue));
obj->head = 0;
obj->tail = 0;
obj->cnt = 0;
return obj;
}

void myQueuePush(MyQueue* obj, int x) {
if(obj == NULL) return;
obj->cnt++;
obj->ar[obj->tail] = x;
obj->tail = (obj->tail + 1)%100;
}

int myQueuePop(MyQueue* obj) {
if(obj == NULL) return NULL;
obj->cnt--;
obj->head = (obj->head + 1)%100;
return (obj->ar[(obj->head-1)%100]);
}

int myQueuePeek(MyQueue* obj) {
if(obj == NULL) return NULL;
return obj->ar[obj->head];
}

bool myQueueEmpty(MyQueue* obj) {
if(obj == NULL) return false;
return (obj->cnt?false:true);
}

void myQueueFree(MyQueue* obj) {
if(obj == NULL) return;
free(obj);
}
```

# 5 Linked list **easy**

## 141. Linked List Cycle

Solved 

Easy

Topics

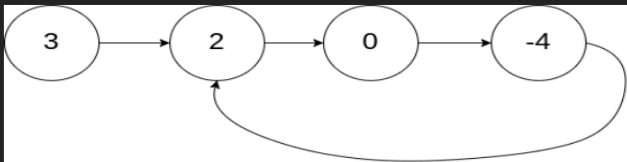
Companies

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

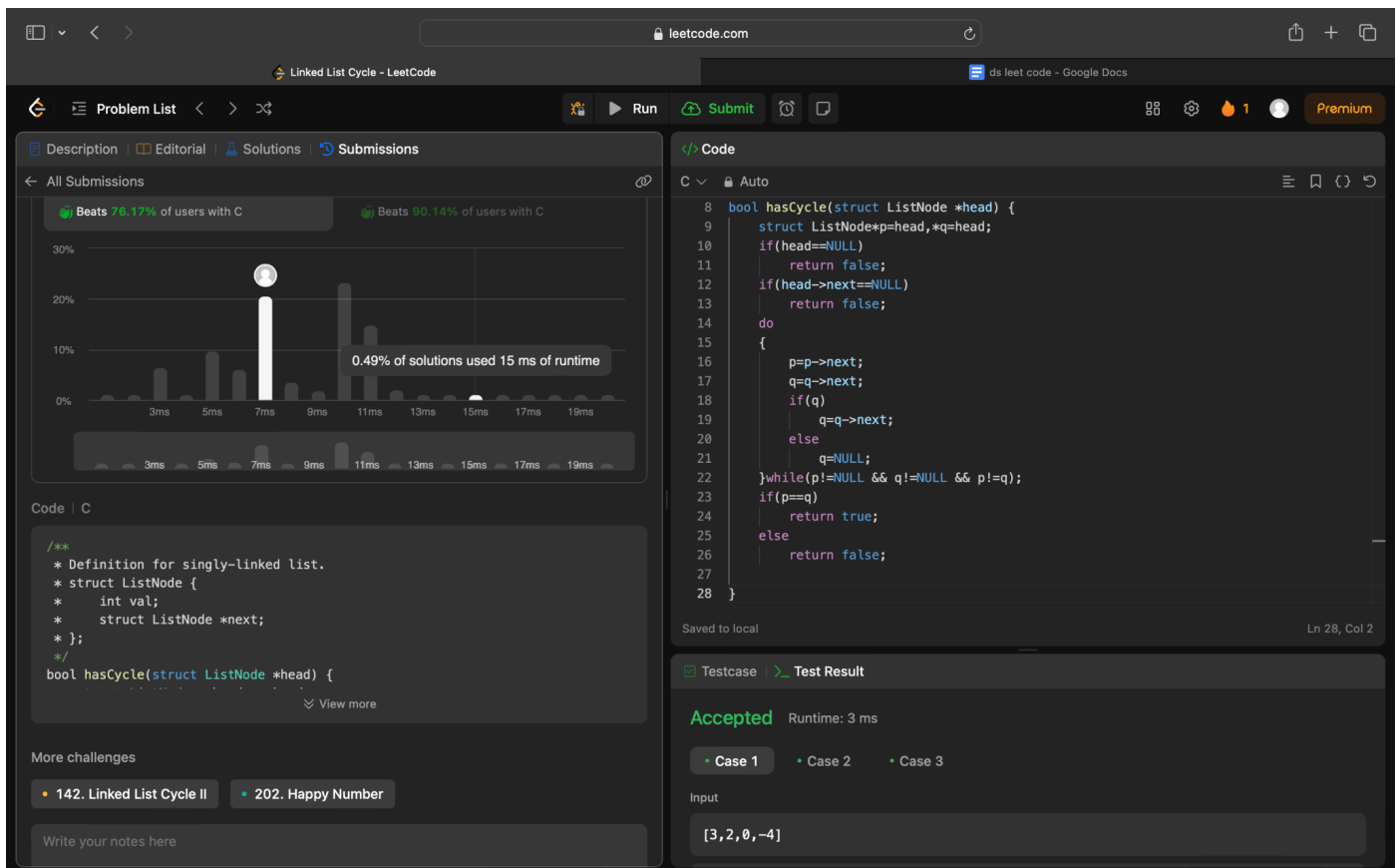
**Example 1:**



**Input:** `head = [3,2,0,-4]`, `pos = 1`

**Output:** `true`

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).



The screenshot shows the LeetCode interface for problem 141. Linked List Cycle. The interface includes a problem description, a performance graph, the C code solution, and the test results.

**Problem Description:** Given `head`, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.** Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

**Example 1:** `head = [3,2,0,-4]`, `pos = 1`. `Output: true`. `Explanation:` There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

**Performance Graph:** The graph shows the distribution of runtime times for solutions. The x-axis represents runtime in milliseconds (3ms to 19ms), and the y-axis represents the percentage of solutions (0% to 30%). The current solution is highlighted with a white bar at 7ms, indicating it beats 76.17% of users with C. A tooltip indicates that 0.49% of solutions used 15 ms of runtime.

**Code:** The code is written in C and implements a Floyd's Cycle-Finding algorithm (tortoise and hare). It uses two pointers, `p` and `q`, both starting at `head`. `p` moves one step at a time, while `q` moves two steps at a time. If there is a cycle, the two pointers will eventually meet. If `head` is `NULL` or `head->next` is `NULL`, it returns `false`. If the pointers meet, it returns `true`. Otherwise, it returns `false`.

```
bool hasCycle(struct ListNode *head) {
    struct ListNode *p=head, *q=head;
    if(head==NULL)
        return false;
    if(head->next==NULL)
        return false;
    do
    {
        p=p->next;
        q=q->next;
        if(q)
            q=q->next;
        else
            q=NULL;
    }while(p!=NULL && q!=NULL && p!=q);
    if(p==q)
        return true;
    else
        return false;
}
```

**Test Results:** The test results show that the solution is **Accepted** with a runtime of 3 ms. The input for the test case is `[3,2,0,-4]`.

# Linked list **easy** code

```
bool hasCycle(struct ListNode *head) {
    struct ListNode*p=head,*q=head;
    if(head==NULL)
        return false;
    if(head->next==NULL)
        return false;
    do
    {
        p=p->next;
        q=q->next;
        if(q)
            q=q->next;
        else
            q=NULL;
    }while(p!=NULL && q!=NULL && p!=q);
    if(p==q)
        return true;
    else
        return false;
}
```

## 6 Linked list medium

### 817. Linked List Components

Medium

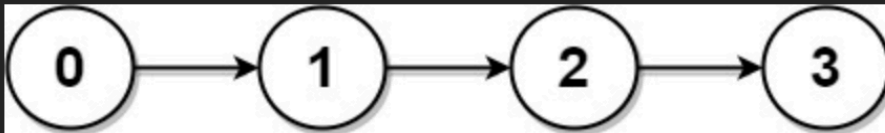
Topics

Companies

You are given the `head` of a linked list containing unique integer values and an integer array `nums` that is a subset of the linked list values.

Return the number of connected components in `nums` where two values are connected if they appear **consecutively** in the linked list.

Example 1:



**Input:** `head = [0,1,2,3]`, `nums = [0,1,3]`

**Output:** 2

**Explanation:** 0 and 1 are connected, so [0, 1] and [3] are the two connected components.

The screenshot displays the LeetCode problem page for '817. Linked List Components'. The interface includes a problem description, a code editor, and submission statistics.

**Problem Description:** You are given the `head` of a linked list containing unique integer values and an integer array `nums` that is a subset of the linked list values. Return the number of connected components in `nums` where two values are connected if they appear **consecutively** in the linked list.

**Example 1:** `head = [0,1,2,3]`, `nums = [0,1,3]`. **Output:** 2. **Explanation:** 0 and 1 are connected, so [0, 1] and [3] are the two connected components.

**Code Editor:** The code is written in C and implements the solution by traversing the linked list and checking for consecutive values in the `nums` array.

```
7  /*
8  int numComponents(struct ListNode* head, int* nums, int numsSize){
9
10     int total = 0, stop = 0;
11     struct ListNode *tmp = NULL, *ptr = NULL;
12     int visited[numsSize], last = 0;
13     memset(visited, 0, sizeof(visited));
14     tmp = head;
15
16     while(tmp != NULL)
17     {
18         for(int i = 0; i < numsSize; i++)
19         {
20             if((tmp->val == nums[i]) && visited[i] != 1)
21             {
22                 if((ptr != NULL) && (ptr->val == nums[last]))
23                 {
24                     total = total;
25                 }
26                 else
27                 {
```

**Submission Statistics:** The solution was accepted by sayeedkhan on Feb 19, 2024, at 20:04. The runtime is 303 ms, which beats 40.00% of users with C. The memory usage is 9.34 MB, which beats 65.71% of users with C.

**Testcase Results:** The solution passed all test cases, including Case 1 and Case 2. The runtime for the test cases is 5 ms.

## Linked list **medium** Code

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *   int val;
 *   struct ListNode *next;
 * };
 */

int numComponents(struct ListNode* head, int* nums, int numsSize){

    int total = 0, stop = 0;
    struct ListNode *tmp = NULL, *ptr = NULL;
    int visited[numsSize], last = 0;
    memset(visited, 0, sizeof(visited));
    tmp = head;

    while(tmp != NULL)
    {
        for(int i = 0; i < numsSize; i++)
        {
            if((tmp->val == nums[i]) && visited[i] != 1)
            {
                if((ptr != NULL) && (ptr->val == nums[last]))
                {
                    total = total;
                }
                else
                {
                    total += 1;
                }
                visited[i] = 1;
                last = i;
                break;
            }
        }
        ptr = tmp;
        tmp = tmp->next;
    }

    return total;
}
```

# 7 BINARY TREE EASY

## 94. Binary Tree Inorder Traversal

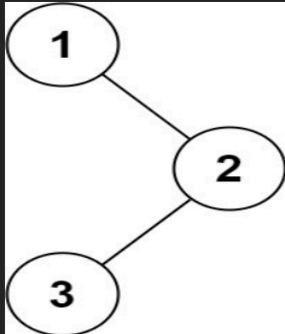
Easy

Topics

Companies

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

**Example 1:**



**Input:** `root = [1,null,2,3]`

**Output:** `[1,3,2]`

Screenshot of the LeetCode interface for the problem "Binary Tree Inorder Traversal".

**Problem Description:** Binary Tree Inorder Traversal - LeetCode

**Accepted Solution:** sayeedkhan submitted at Feb 19, 2024 20:17

**Runtime:** 0 ms (Beats 100.00% of users with C)

**Memory:** 5.80 MB (Beats 83.08% of users with C)

**Code (C):**

```
int i=0;
int arr[101]={0};
void inorder(struct TreeNode* s)
{
    if(s!=NULL)
    {
        inorder(s->left);
        arr[i++]=s->val;
        inorder(s->right);
    }
}

int* inorderTraversal(struct TreeNode* root, int* returnSize){
    inorder(root);
    int* ans=malloc(i*sizeof(int));
    for(int j=0;j<i;j++) ans[j]=arr[j];
    *(returnSize)=i;
    i=0;
    return ans;
}
```

**Testcase Results:** Accepted (Runtime: 0 ms)

**Case 1:** Input: `root = [1,null,2,3]`



# BINARY TREE EASY CODE

```
int i=0;
int arr[101]={0};
void inorder(struct TreeNode* s)
{
    if(s!=NULL)
    {
        inorder(s->left);
        arr[i++]=s->val;
        inorder(s->right);
    }
}

int* inorderTraversal(struct TreeNode* root, int* returnSize){
    inorder(root);
    int* ans=malloc(i*sizeof(int));
    for(int j=0;j<i;j++) ans[j]=arr[j];
    *(returnSize)=i;
    i=0;
    return ans;
}
```

## 8 GRAPH MEDIUM

### 133. Clone Graph

Solved 

Medium

Topics

Companies

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

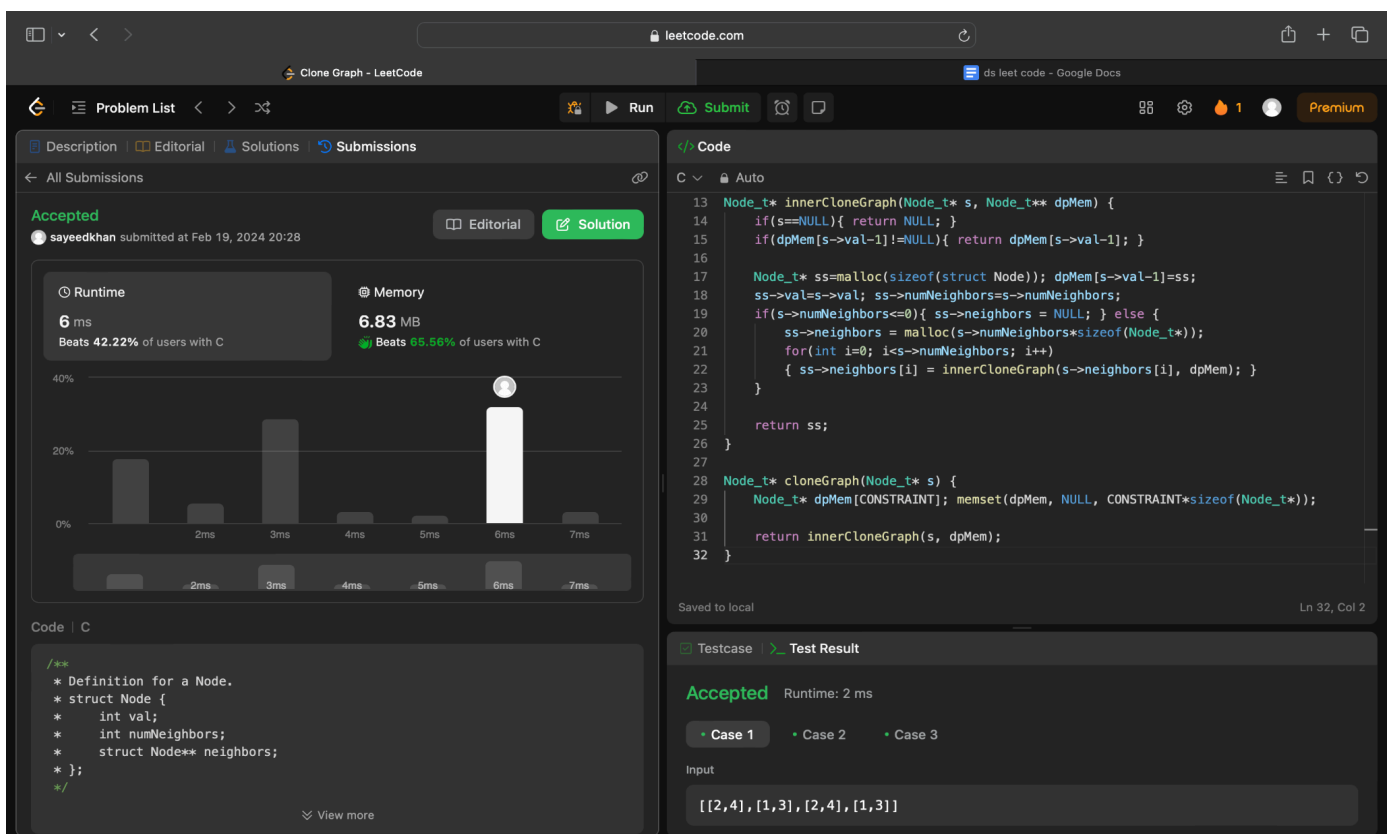
```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

#### Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An **adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.



The screenshot displays the LeetCode problem page for "133. Clone Graph". The problem description is on the left, and the C solution is on the right. The solution is a recursive function that clones the graph by creating new nodes and linking their neighbors. The test results show that the solution is "Accepted" with a runtime of 2 ms.

**Problem Description:**

Given a reference of a node in a **connected** undirected graph. Return a **deep copy** (clone) of the graph. Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

**Test case format:**

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An **adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

**C Solution:**

```
Node_t* innerCloneGraph(Node_t* s, Node_t** dpMem) {
    if(s==NULL){ return NULL; }
    if(dpMem[s->val-1]!=NULL){ return dpMem[s->val-1]; }

    Node_t* ss=malloc(sizeof(struct Node)); dpMem[s->val-1]=ss;
    ss->val=s->val; ss->numNeighbors=s->numNeighbors;
    if(s->numNeighbors<=0){ ss->neighbors = NULL; } else {
        ss->neighbors = malloc(s->numNeighbors*sizeof(Node_t*));
        for(int i=0; i<s->numNeighbors; i++)
            { ss->neighbors[i] = innerCloneGraph(s->neighbors[i], dpMem); }
    }

    return ss;
}

Node_t* cloneGraph(Node_t* s) {
    Node_t* dpMem[CONSTRANT]; memset(dpMem, NULL, CONSTRANT*sizeof(Node_t*));
    return innerCloneGraph(s, dpMem);
}
```

**Test Results:**

Accepted Runtime: 2 ms

Case 1 Case 2 Case 3

Input: `[[2,4],[1,3],[2,4],[1,3]]`

# GRAPH MEDIUM CODE

```
#define CONSTRAINT 100

typedef struct Node Node_t;

Node_t* innerCloneGraph(Node_t* s, Node_t** dpMem) {
    if(s==NULL){ return NULL; }
    if(dpMem[s->val-1]!=NULL){ return dpMem[s->val-1]; }

    Node_t* ss=malloc(sizeof(struct Node)); dpMem[s->val-1]=ss;
    ss->val=s->val; ss->numNeighbors=s->numNeighbors;
    if(s->numNeighbors<=0){ ss->neighbors = NULL; } else {
        ss->neighbors = malloc(s->numNeighbors*sizeof(Node_t*));
        for(int i=0; i<s->numNeighbors; i++)
        { ss->neighbors[i] = innerCloneGraph(s->neighbors[i], dpMem); }
    }

    return ss;
}

Node_t* cloneGraph(Node_t* s) {
    Node_t* dpMem[CONSTRAINT]; memset(dpMem, NULL, CONSTRAINT*sizeof(Node_t*));

    return innerCloneGraph(s, dpMem);
}
```

# 9 ADD BINARY TREE EASY

## 67. Add Binary

Easy

Topics

Companies

Given two binary strings `a` and `b`, return *their sum as a binary string*.

**Example 1:**

**Input:** `a = "11", b = "1"`

**Output:** `"100"`

**Example 2:**

**Input:** `a = "1010", b = "1011"`

**Output:** `"10101"`

**Constraints:**

- `1 <= a.length, b.length <= 104`
- `a` and `b` consist only of `'0'` or `'1'` characters.
- Each string does not contain leading zeros except for the zero itself.

The screenshot shows the LeetCode interface for problem 67. Add Binary. The problem is marked as Easy. The constraints are: `1 <= a.length, b.length <= 104`, `a` and `b` consist only of `'0'` or `'1'` characters, and each string does not contain leading zeros except for the zero itself. The solution is in C and is accepted. The runtime is 2ms, which beats 45.17% of users with C. The memory usage is 5.88 MB, which beats 68.84% of users with C. The code is as follows:

```
char * addBinary(char * a, char * b){
    int sizeA = strlen(a);
    int sizeB = strlen(b);
    int sizeOutput = (sizeA > sizeB ? sizeA : sizeB) + 1;
    char * output = (char *)malloc(sizeOutput + 1);
    int sum = 0;

    output[sizeOutput] = '\0';

    while(sizeA > 0 || sizeB > 0 || sum > 0) {
        if(sizeA > 0) {
            sum += a[--sizeA] - '0';
        }
        if(sizeB > 0) {
            sum += b[--sizeB] - '0';
        }
        output[--sizeOutput] = sum % 2 + '0';
        sum /= 2;
    }

    return output + sizeOutput;
}
```

# ADD BINARY TREE **EASY** CODE

```
char * addBinary(char * a, char * b){
int sizeA = strlen(a);
int sizeB = strlen(b);
int sizeOutput = (sizeA > sizeB ? sizeA : sizeB) + 1;
char * output = (char *)malloc(sizeOutput + 1);
int sum = 0;
output[sizeOutput] = '\0';
while(sizeA > 0 || sizeB > 0 || sum > 0) {
if(sizeA > 0) {
sum += a[--sizeA] - '0';
}
if(sizeB > 0) {
sum += b[--sizeB] - '0';
}
output[--sizeOutput] = sum % 2 + '0';
sum /= 2;
}
return output + sizeOutput;
}
```

# 10 DESIGN HASHMAP EASY

## 706. Design HashMap

Easy

Topics

Companies

Design a HashMap without using any built-in hash table libraries.

Implement the `MyHashMap` class:

- `MyHashMap()` initializes the object with an empty map.
- `void put(int key, int value)` inserts a `(key, value)` pair into the HashMap. If the `key` already exists in the map, update the corresponding `value`.
- `int get(int key)` returns the `value` to which the specified `key` is mapped, or `-1` if this map contains no mapping for the `key`.
- `void remove(key)` removes the `key` and its corresponding `value` if the map contains the mapping for the `key`.

Example 1:

**Input**

```
["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]  
[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]
```

**Output**

```
[null, null, null, 1, -1, null, 1, null, -1]
```

The screenshot shows a LeetCode submission interface for the problem "Design HashMap". The submission is in C and has been accepted. The runtime is 111 ms, which beats 54.55% of users with C. The memory usage is 31.32 MB, which beats 97.98% of users with C. The submission was made by "sayeedkhan" on Feb 19, 2024 at 22:18.

The code defines a `MyHashMap` struct with a `size` and a `Table` of `HashNode` pointers. The `HashNode` struct contains `key`, `value`, `next`, and `prev` pointers. The `put` method inserts a new key-value pair or updates an existing one. The `get` method returns the value for a given key or -1 if not found. The `remove` method removes a key-value pair.

```
#define TABLE_SIZE 10

struct HashNode{
    int key;
    int value;
    struct HashNode *next;
    struct HashNode *prev;
};

struct MyHashMap
{
    int size;
    struct HashNode **Table;
};

typedef struct MyHashMap MyHashMap;

int hash(struct MyHashMap *ht,int key)
{
    return key%(ht->size);
}
```

The test case shows the following sequence of operations: `["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]` with corresponding inputs: `[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]`. The expected output is `[null, null, null, 1, -1, null, 1, null, -1]`.

# DESIGN HASHMAP EASY CODE

```
#define TABLE_SIZE 10

struct HashNode{
int key;
int value;
struct HashNode *next;
struct HashNode *prev;
};

struct MyHashMap
{
int size;
struct HashNode **Table;
};

typedef struct MyHashMap MyHashMap;

int hash(struct MyHashMap *ht,int key)
{
return key%(ht->size);
}

MyHashMap* myHashMapCreate() {
struct MyHashMap *ht=NULL;
ht=(struct MyHashMap *)malloc(sizeof(MyHashMap));
(ht)->size=TABLE_SIZE;
(ht)->Table=(struct HashNode **)calloc(TABLE_SIZE,sizeof(struct HashNode *));
for(int iCnt=0;iCnt<(ht->size);iCnt++)
{
(ht)->Table[iCnt]=NULL;
}
return ht;
}

void myHashMapPut(struct MyHashMap *ht, int key, int value) {
int index=hash(ht,key);
struct HashNode *temp=ht->Table[index];
while(temp!=NULL)
{
if(temp->key==key)
{
temp->value=value;
return;
}
temp=temp->next;
}
struct HashNode *newn=(struct HashNode *)malloc(sizeof(struct HashNode));
newn->key=key;
newn->value=value;
newn->next=NULL;
struct HashNode *first=ht->Table[index];

if(ht->Table[index]==NULL)
{
ht->Table[index]=newn;
}
else //Insert_First
{
newn->next=first;
ht->Table[index]=newn;
}
}
```

```

}

int myHashMapGet(struct MyHashMap *ht, int key) {
int find=-1;
int index=hash(ht,key);

struct HashNode *temp=ht->Table[index];

while(temp!=NULL)
{
if(temp->key==key)
{
return temp->value;
}
temp=temp->next;
}

return -1;
}

void myHashMapRemove(struct MyHashMap *ht, int key) {
int index=hash(ht,key);

struct HashNode *first=ht->Table[index];
struct HashNode *temp=first;

struct HashNode *hold=NULL;

while(temp!=NULL)
{
if(temp->key==key)
{
if(hold==NULL)
{
ht->Table[index]=temp->next;
}
else
{
hold->next=temp->next;
}
free(temp);
Return;
}
hold=temp;
temp=temp->next;
}
}

void myHashMapFree( struct MyHashMap *ht) {
for(int i=0;i<TABLE_SIZE;i++)
{
free(ht->Table[i]);
}
free(ht);
}

/**
* Your MyHashMap struct will be instantiated and called as such:
* MyHashMap* obj = myHashMapCreate();
* myHashMapPut(obj, key, value);
* int param_2 = myHashMapGet(obj, key);
* myHashMapRemove(obj, key);
* myHashMapFree(obj);
*/

```



# REAL WORLD PROBLEM

- This program simulates a task management system with tasks having priorities. The priority queue will be used to efficiently manage tasks based on their priority, and a hash table will be used to quickly access and update the tasks.
- The program uses a priority queue (min-heap) to maintain tasks based on their priority and a hash table to quickly access and update tasks. The priority queue ensures that tasks are processed in order of priority, and the hash table provides quick access to tasks for updates based on their names.

## CODE

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

// Define structures for priority queue (min-heap) node and hash table entry
typedef struct PriorityQueueNode {
    char task[50];
    int priority;
} PriorityQueueNode;

typedef struct HashTableEntry {
    char task[50];
    int priority;
    struct HashTableEntry* next;
} HashTableEntry;

// Define structure for priority queue
typedef struct {
    PriorityQueueNode* heap;
    int size;
    int capacity;
} PriorityQueue;

// Define structure for hash table
typedef struct {
    HashTableEntry* table[10]; // Using a simple hash table with 10 slots for demonstration
```

```

} HashTable;

// Function prototypes
PriorityQueue* initializePriorityQueue(int capacity);
void enqueue(PriorityQueue* priorityQueue, char task[], int priority);
void heapifyUp(PriorityQueue* priorityQueue, int index);
void dequeue(PriorityQueue* priorityQueue);
HashTable* initializeHashTable();
void addTaskToHashTable(HashTable* hashTable, char task[], int priority);
void updatePriorityInHashTable(HashTable* hashTable, char task[], int newPriority);
void displayTasksHashTable(HashTable* hashTable);

int main() {
    PriorityQueue* taskPriorityQueue = initializePriorityQueue(10);
    HashTable* taskHashTable = initializeHashTable();

    int choice;
    char task[50];
    int priority, newPriority;

    do {
        printf("\nTask Management System Menu\n");
        printf("1. Add a task\n");
        printf("2. Update task priority\n");
        printf("3. Display tasks (hash table)\n");
        printf("0. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the task: ");
                scanf(" %[^\n]", task);
                printf("Enter the priority of the task: ");
                scanf("%d", &priority);

                enqueue(taskPriorityQueue, task, priority);
                addTaskToHashTable(taskHashTable, task, priority);
                break;

            case 2:
                printf("Enter the task to update priority: ");
                scanf(" %[^\n]", task);
                printf("Enter the new priority of the task: ");

```

```

scanf("%d", &newPriority);

updatePriorityInHashTable(taskHashTable, task, newPriority);
break;

case 3:
printf("Tasks in hash table:\n");
displayTasksHashTable(taskHashTable);
break;

case 0:
printf("Exiting the program. Goodbye!\n");
break;

default:
printf("Invalid choice. Please try again.\n");
}

} while (choice != 0);

free(taskPriorityQueue->heap);
free(taskPriorityQueue);
return 0;
}

PriorityQueue* initializePriorityQueue(int capacity) {
PriorityQueue* priorityQueue = (PriorityQueue*)malloc(sizeof(PriorityQueue));
priorityQueue->heap = (PriorityQueueNode*)malloc(capacity * sizeof(PriorityQueueNode));
priorityQueue->size = 0;
priorityQueue->capacity = capacity;
return priorityQueue;
}

void enqueue(PriorityQueue* priorityQueue, char task[], int priority) {
if (priorityQueue->size < priorityQueue->capacity) {
priorityQueue->heap[priorityQueue->size].priority = priority;
strcpy(priorityQueue->heap[priorityQueue->size].task, task);
heapifyUp(priorityQueue, priorityQueue->size++);
printf("Task added to priority queue.\n");
} else {
printf("Priority queue is full. Cannot add more tasks.\n");
}
}

```

```

void heapifyUp(PriorityQueue* priorityQueue, int index) {
    int parent = (index - 1) / 2;

    while (index > 0 && priorityQueue->heap[index].priority <
priorityQueue->heap[parent].priority) {
        // Swap nodes if child has higher priority than the parent
        PriorityQueueNode temp = priorityQueue->heap[index];
        priorityQueue->heap[index] = priorityQueue->heap[parent];
        priorityQueue->heap[parent] = temp;

        index = parent;
        parent = (index - 1) / 2;
    }
}

void dequeue(PriorityQueue* priorityQueue) {
    if (priorityQueue->size > 0) {
        // Swap the root (highest priority) with the last element
        PriorityQueueNode temp = priorityQueue->heap[0];
        priorityQueue->heap[0] = priorityQueue->heap[--priorityQueue->size];
        priorityQueue->heap[priorityQueue->size] = temp;

        // Re-heapify to maintain the min-heap property
        heapifyDown(priorityQueue, 0);
        printf("Task with the highest priority removed from priority queue.\n");
    } else {
        printf("Priority queue is empty. Cannot dequeue.\n");
    }
}

void heapifyDown(PriorityQueue* priorityQueue, int index) {
    int leftChild = 2 * index + 1;
    int rightChild = 2 * index + 2;
    int smallest = index;

    if (leftChild < priorityQueue->size && priorityQueue->heap[leftChild].priority <
priorityQueue->heap[smallest].priority) {
        smallest = leftChild;
    }

    if (rightChild < priorityQueue->size && priorityQueue->heap[rightChild].priority <
priorityQueue->heap[smallest].priority) {
        smallest = rightChild;
    }
}

```

```

if (smallest != index) {
    // Swap nodes if a child has higher priority
    PriorityQueueNode temp = priorityQueue->heap[index];
    priorityQueue->heap[index] = priorityQueue->heap[smallest];
    priorityQueue->heap[smallest] = temp;

    // Recursively heapify the affected subtree
    heapifyDown(priorityQueue, smallest);
}
}

HashTable* initializeHashTable() {
    HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));

    for (int i = 0; i < 10; i++) {
        hashTable->table[i] = NULL;
    }

    return hashTable;
}

void addTaskToHashTable(HashTable* hashTable, char task[], int priority) {
    unsigned int index = priority % 10; // Using priority as a hash index for demonstration
    HashTableEntry* newEntry = (HashTableEntry*)malloc(sizeof(HashTableEntry));
    strcpy(newEntry->task, task);
    newEntry->priority = priority;
    newEntry->next = hashTable->table[index];
    hashTable->table[index] = newEntry;
}

void updatePriorityInHashTable(HashTable* hashTable, char task[], int newPriority) {
    unsigned int index = newPriority % 10; // Using new priority as a hash index for demonstration
    HashTableEntry* entry = hashTable->table[index];

    while (entry != NULL) {
        if (strcmp(task, entry->task) == 0) {
            entry->priority = newPriority;
            printf("Task priority updated in hash table.\n");
            return;
        }
        entry = entry->next;
    }
}

```

```

}

printf("Task not found in hash table.\n");
}

void displayTasksHashTable(HashTable* hashTable) {
for (int i = 0; i < 10; i++) {
HashTableEntry* entry = hashTable->table[i];

while (entry != NULL) {
printf("Priority: %d, Task: %s\n", entry->priority, entry->task);
entry = entry->next;
}
}
}

```

## OUTPUT :

```

Task Management System Menu
1. Add a task
2. Update task priority
3. Display tasks (hash table)
0. Exit
Enter your choice: |

```

==>

```

Task Management System Menu
1. Add a task
2. Update task priority
3. Display tasks (hash table)
0. Exit
Enter your choice: 1
Enter the task: chrome
Enter the priority of the task: 1
Task added to priority queue.

```