Problem Solving (C66)

هذا البرنامج التدريبي مُصاغ بعناية لتمكين المتدربين من تطوير قدراتهم الفكرية على غرار المبرمجين المحترفين، والتعاون بكفاءة ضمن فريق محترف في شركة "جمال تك" أو أي مؤسسة متعددة الجنسيات أخرى. نظرًا لأهمية اللغة الإنجليزية في بيئة العمل العالمية، يتم تقديم المحتوى التدريبي بالإنجليزية. لا يشترط إتقان اللغة بشكل كامل، لكن من الضروري امتلاك القدرة الكافية لفهم المتطلبات وتنفيذها بشكل فعّال. يُمكن للمتدربين استخدام مترجم جوجل أو الاستعانة بـ "شات جي بي تي" للتغلب على أية عقبات لغوية، المهم هو الفهم الدقيق للمطلوب وتحقيقه بنجاح.

لتعظيم الاستفادة من التدريب، يُنصح بمحاولة حل التمارين بشكل مستقل لمدة ساعة واحدة على الأقل قبل الرجوع إلى الحل المرفق في نهاية الملف.

قد يتضمن الحل كودًا برمجيًا غير مفسر بعد، والغرض من ذلك هو تشجيعك على محاولة فهم الأكواد البرمجية الجديدة التي لم تتعرض لها من قبل. هذه المهارة ضرورية في سوق العمل، حيث تتطور لغات البرمجة باستمرار ويظهر كل يوم لغات جديدة. ستواجه دائمًا أكوادًا لم تدرسها من قبل، ومن المهم أن تكون قادرًا على فهمها بنفسك دون الحاجة إلى دراسة مسبقة. يمكنك الاستعانة بمحرك البحث جوجل، أو استخدام ChatGPT، أو حتى اللجوء لأصدقائك للمساعدة. الهدف الأساسي هو أن تصل إلى فهم معنى كل كود بأي طريقة ممكنة لتتمكن من إيجاد موقعك في سوق العمل.

إن وجود كود برمجي غير مفسر يشكل تحديًا يتوجب عليك إيجاد حل له. هذا النوع من التدريبات يعد جزءًا أساسيًا من تدريبات 'Problem Solving'، التي تهدف إلى تمكينك من أداء عملك بفاعلية بغض النظر عن التحديات والعقبات. هذه القدرة على حل المشكلات هي ما يتمتع به العاملون في 'جمال تك'، ومن الضروري أن تطور في نفسك هذه المهارة لتصبح عضوًا فعالًا في فريق عمل 'جمال تك'.

Gammal Tech's Innovative Cache System

Background

Gammal Tech, a trailblazer in the software development industry, is working on a cutting-edge technology to enhance computer architecture. They are developing an innovative cache system aimed at optimizing data access speed. This system is designed to revolutionize how data is stored and retrieved in computing devices, further cementing Gammal Tech's status as a leader in innovative technology solutions.

Problem Description

Your task is to help Gammal Tech design a basic simulation of their new cache system. The cache system operates on a simple principle: it temporarily stores frequently accessed data to reduce access time. The system has a limited size and follows a Least Recently Used (LRU) eviction policy, where the least recently accessed item is removed when the cache is full.

You need to implement a program that simulates the operation of this cache system. The program should be able to process two types of operations:

Access a data item with a given identifier.
Query the cache to check if a data item is present.

Your program should output the status of the cache after each operation.

Input Format

- The first line contains two integers, `N` and `Q`, representing the size of the cache and the number of operations, respectively.
- The next `Q` lines describe the operations. Each line starts with a character (`A` for Access or `Q` for Query), followed by an integer representing the data item's identifier.

Output Format

- After each `Access` operation, output the contents of the cache in the order of most recently to least recently accessed.
- After each `Query` operation, output `Yes` if the item is in the cache, otherwise `No`.

Constraints

- $1 \leq N \leq 100$
- $1 \leq Q \leq 1000$
- Data item identifiers are integers in the range
- $1 \leq \text{identifier} \leq 1000$

Sample Input:

```
3 6
A 5
A 12
Q 5
A 6
Q 12
A 7
```

Sample Output:

```
5
12 5
Yes
6 12 5
No
7 6 12
```

Explanation

- After accessing `5` and `12`, the cache contains these items.
- Querying for `5` returns `Yes` as it is present in the cache.
- After accessing `6`, the cache contains `6, 12, 5`.
- Querying for `12` returns `No` as it is no longer in the cache after accessing `7`, which causes `12` to be evicted following the LRU policy.

لتعظيم الاستفادة من التدريب، يُنصح بمحاولة حل التمرين بشكل مستقل لمدة ساعة واحدة على الأقل قبل الرجوع إلى الحل المرفق

C Programming Solution:

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

typedef struct {
    int key;
    struct Node* prev;
    struct Node* next;
} Node;

typedef struct {
    int capacity, count;
    Node* head, *tail;
    Node* array[1001]; // Assuming the maximum identifier value is 1000
} LRUCache;

LRUCache* lRUCacheCreate(int capacity) {
    LRUCache* cache = (LRUCache*)malloc(sizeof(LRUCache));
    cache->capacity = capacity;
    cache->count = 0;
    cache->head = NULL;
    cache->tail = NULL;
    for (int i = 0; i < 1001; i++) {
        cache->array[i] = NULL;
    }
    return cache;
}

void moveToHead(LRUCache* cache, Node* node) {
    if (cache->head == node) return;
    if (cache->tail == node) {
        cache->tail = node->prev;
        cache->tail->next = NULL;
    } else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    node->next = cache->head;
    node->prev = NULL;
    cache->head->prev = node;
    cache->head = node;
}
```

```
void addNodeToHead(LRUCache* cache, Node* node) {
    if (cache->count == 0) {
        cache->head = node;
        cache->tail = node;
    } else {
        node->next = cache->head;
        cache->head->prev = node;
        cache->head = node;
    }
    cache->count++;
}

void removeNode(LRUCache* cache, Node* node) {
    if (cache->head == node && cache->tail == node) {
        cache->head = NULL;
        cache->tail = NULL;
    } else if (cache->head == node) {
        cache->head = node->next;
        cache->head->prev = NULL;
    } else if (cache->tail == node) {
        cache->tail = node->prev;
        cache->tail->next = NULL;
    } else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    cache->count--;
}

void lRUCacheAccess(LRUCache* cache, int key) {
    Node* node = cache->array[key];
    if (node == NULL) {
        node = (Node*)malloc(sizeof(Node));
        node->key = key;
        if (cache->count == cache->capacity) {
            Node* tail = cache->tail;
            removeNode(cache, tail);
            cache->array[tail->key] = NULL;
            free(tail);
        }
        addNodeToHead(cache, node);
        cache->array[key] = node;
    } else {
        moveToHead(cache, node);
    }
}

bool lRUCacheQuery(LRUCache* cache, int key) {
    Node* node = cache->array[key];
    return node != NULL;
}
```

```c
void lRUCachePrint(LRUCache* cache) {
    Node* node = cache->head;
    while (node != NULL) {
        printf("%d ", node->key);
        node = node->next;
    }
    printf("\n");
}

void lRUCacheFree(LRUCache* cache) {
    Node* node = cache->head;
    while (node != NULL) {
        Node* temp = node;
        node = node->next;
        free(temp);
    }
    free(cache);
}

int main() {
    int N, Q;
    scanf("%d %d", &N, &Q);
    LRUCache* cache = lRUCacheCreate(N);

    for (int i = 0; i < Q; i++) {
        char op;
        int key;
        scanf(" %c %d", &op, &key);
        if (op == 'A') {
            lRUCacheAccess(cache, key);
            lRUCachePrint(cache);
        } else if (op == 'Q') {
            printf("%s\n", lRUCacheQuery(cache, key) ? "Yes" : "No");
        }
    }

    lRUCacheFree(cache);
    return 0;
}
```