עבודת בית **3:**

מגישים:

שם: שי שמאילוב

מ.ס: **305098774**

שם: אופק גוטמן

מ.ס: **203115381**

תא החזרה: **74**

חלק יבש – שאלה **2:**

2.1:

1. נשתמש בקבוצה, מבנה נתונים זה יעזור לנו לאכסן חברים בפייסבוק ללא חשיבות לסדר שלהם. אם למשל נרצה לדעת עבור משתמש מסוים מהו הסדר בין חבריו מיום החברות הראשון שלהם, נשמור את התאריך של היום שנהיו חברים באחד השדות של המבנה. כמו כן בשימוש במבנה נתונים קבוצה אין הגבלה על כמות האיברים ולכן נוכל להוסיף חברים כמה שנרצה. בכדי שנוכל לקיים מצב של שני חברים עם אותו שם נשמור מספר מזהה ייחודי לכל חבר. הגישה לחברי הפייסבוק של המשתמש יעילה בעזרת מבנה הנתונים קבוצה.

2. נשתמש ברשימה מקושרת, בעזרת מבנה נתונים זה נוכל לשמור על הסדר כפי שרצוי. מבנה נתונים קבוצה אינו מתאים כיוון שאין בו סדר. מבנה נתונים מחסנית מאפשר גישה מסורבלת לאיבריו ולכן נבחר ברשימה מקושרת. אמנם בכדי לגשת לאיבר, נצטרך לעבור על כל הרשימה אך מכל המבני נתונים הוא המבנה היעיל ביותר לבעיה הנתונה.

3. נשתמש במחסנית, תנאי הבעיה מדגישים את הצורך בשימוש במחסנית. מובטח שכאשר אדם מגיע, כל האנשים שיגיעו אחריו יצאו לפניו. כלומר, תמיד התקיים שהאורח האחרון שהגיע, הוא הראשון שיצא. מחסנית מבטיחה לנו שמירה על הסדר הנדרש בבעיה ולכן נבחר במבנה נתונים זה.

binaryFind.c

```c
/**generic type*/
typedef void *Element;
/**
 * compare between two elements:
 * if (a>b) return positive number
 * if (a<b) return negative number
 * if (a=b) return NULL
 */
typedef int (*cmp)(Element a,Element b);
/**
 * generic binary search
 * @param arr - array of Elements,the type of each element is void* and the type of the array is void**
 * @param key - type void*, the element we want to search
 * @param len - the amount of elements in the array
 * @param cmpFunc - function to compare between two elements
 * @return the index number of key if exist otherwise -1
 */
int binaryFind(Element *arr,Element key,int len,cmp cmpFunc){
    int low = 0,high = len-1,current_index;
    while(low <= high){
        current_index = (high+low)/2;
        int result = cmpFunc(key,arr[current_index]);

        if(result == 0) return current_index;
        else if(result < 0){
            high = current_index-1;
        }
        else{
            low = current_index+1;
        }
    }
    return -1;

}
```

concatList.c

```c
#include "node.h"

Node concatList(Node head1,Node head2,copyFunc copy,filterFunc filter,Key key);
static Node unitelists(Node head1,Node head2);
static Node listOfIntegers(Node *head,int array[],int lenght);
static Element cpyInteger(Element n);
static bool isOdd(Element element,Key key);
static bool isPrime(Element element,Key key);
static void desstroyList(Node head);
static void printList(Node head,char *a);

int main(){
    int a[]={1,2,3,4};
    int b[]={5,6,7};
    int len_a=4;
    int len_b=3;
    Node head1=NULL;
    listOfIntegers(&head1,a,len_a);
    Node head2=NULL;
    listOfIntegers(&head2,b,len_b);
    int key=2;
    Node head_odd=concatList(head1,head2,cpyInteger,isOdd,&key);
    Node head_prime=concatList(head1,head2,cpyInteger,isPrime,&key);

    printList(head_odd,"Odds List");
    printList(head_prime,"Primes List");
    desstroyList(head_odd);
    desstroyList(head_prime);
    desstroyList(head1);
    desstroyList(head2);

    return 0;


```

```
}

Node concatList(Node head1,Node head2,copyFunc copy,filterFunc filter,Key key){
    if(copy == NULL || filter == NULL || key == NULL){
        return NULL;
    }
    Node head1_new=NULL,currnet_node=NULL;
    while(head1 != NULL){
        Node new_node=NULL;
        NodeResult result;
        if(filter(&(head1->n),key)){
            result=nodeCreate(&new_node);
            if(result != NODE_SUCCESS) return NULL;
            new_node->n=head1->n;
            if(head1_new == NULL){
                head1_new=new_node;
            }
            else{
                currnet_node=head1_new;
                while(currnet_node->next!=NULL){
                    currnet_node=currnet_node->next;
                }
                currnet_node->next=new_node;
            }
        }
        if(head1->next!=NULL){
            head1=head1->next;
        }
        else{
            break;
        }
    }
    Node head2_new=NULL;
    while(head2 != NULL){
        Node new_node=NULL;
        NodeResult result;
        if(filter(&(head2->n),key)){
            result=nodeCreate(&new_node);
            if(result != NODE_SUCCESS) return NULL;
            new_node->n=head2->n;
            if(head2_new == NULL){
                head2_new=new_node;
            }
            else{
                currnet_node=head2_new;
                while(currnet_node->next!=NULL){
                    currnet_node=currnet_node->next;
                }
                currnet_node->next=new_node;
            }
        }
        if(head2->next!=NULL) {
            head2=head2->next;
        }
        else{
            break;
        }
    }
    return unitelists(head1_new,head2_new);
}

static Node listOfIntegers(Node *head,int array[],int lenght) {
    NodeResult result=nodeCreate(head);
    if (result != NODE_SUCCESS) return NULL;
    Node node = *head;
    for(int i = 0;i < lenght;i++){
        node->n=*(array+i);
        if(i < lenght-1){
```

```c
            result=nodeCreate(&(node->next));
            if(result != NODE_SUCCESS) return NULL;
            node = node->next;
        }
    }

    return *head;
}

static Element cpyInteger(Element n){
    if(n == NULL) return NULL;
    int *num=malloc(sizeof(num));
    if(num == NULL) return NULL;
    num=(int *)n;
    return num;
}

static bool isOdd(Element element,Key key){
    assert(element != NULL);
    assert(key != NULL);
    if(*(int *)element%2==0){
        return false;
    }
    return true;
}

static bool isPrime(Element element,Key key){
    assert(element != NULL);
    assert(key != NULL);
    int num = *(int *)element;
    int first_prime = *(int *)key;
    if(num == first_prime) return true;
    if(num < first_prime){
        return false;
    }

    for(int i = first_prime;i < num;i++){
        if(num%i == 0){
            return false;
        }
    }
    return true;
}

static Node unitelists(Node head1,Node head2){
    Node tmp=head1;
    while(tmp->next != NULL){
        tmp = tmp->next;
    }
    tmp->next = head2;
    return head1;
}

static void printList(Node head,char *a){
    printf("%s: " ,a);
    while(head != NULL){
        printf("%d " ,head->n);
        head = head->next;
    }
    printf("\n");
}
static void desstroyList(Node head){
    while(head != NULL){
        Node tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

node.c

```c
#include "node.h"



NodeResult nodeCreate(Node *head){
    Node node = malloc(sizeof(node));
    if(node == NULL) return NODE_OUT_OF_MEMORY;
    node->next=NULL;
    *head=node;
    return NODE_SUCCESS;
}

```

node.h

```c
#ifndef NODE_DRY_NODE_H
#define NODE_DRY_NODE_H
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/**defining of stuct*/
typedef struct node_t{
    int n;
    struct node_t *next;
}*Node;

/**errors to the ADT*/
typedef enum{
    NODE_NULL,
    NODE_OUT_OF_MEMORY,
    NODE_SUCCESS,
}NodeResult;

/**type of generic element*/
typedef void *Element;

/**type for copy function*/
typedef Element(*copyFunc)(Element);

/**type for the filter key*/
typedef void *Key;

/**type for the filter function*/
typedef bool(*filterFunc)(Element , Key);

NodeResult nodeCreate(Node *node);
#endif //NODE_DRY_NODE_H
```