TOBB ETU – 2025 Fall
BIL 214 – Exercises 8

Instructions

- Write the function definitions of the functions defined in "exercises8.h". Do **not** change "exercises8.h".
- Do not include other libraries or define constant/global variables. Do not use recursions.
- From Standard C Library Functions, only use the ones listed in Worksheet_C.
- You must **use dynamic memory allocation** for all array and struct declarations.
- You can assume that parameters are always passed correctly.

exercises8.h

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

float** soru01(int M, int *Ns);
// Allocate a 2D dynamic array with varying column sizes and return its address.
// M is the number of rows and Ns is an int array whose i-th element is the column size of i-th row of A.
// Time complexity of the function must be O(M).

void soru02(float **A, int M);
// Deallocate 2D dynamic array (A) with varying column sizes. M is the number of rows.
// Time complexity of the function must be O(M). Space complexity of the function must be O(1).

void soru03(float **A, int M, int *Ns);
// Set elements of 2D dynamic array (A) with varying column sizes by obtaining their values from the user.
// M is the number of rows and Ns is an int array whose i-th element is the column size of i-th row of A.
// Time complexity of the function must be O(K) where K is the total number of elements of A.
// Space complexity of the function must be O(1).

float* soru04(float **A, int M, int *Ns);
// Save average of each row of 2D dynamic array A to a dynamic array and return its address.
// M is the number of rows and Ns is an int array whose i-th element is the column size of i-th row of A.
// Time complexity of the function must be O(K) where K is the total number of elements of A.
// Space complexity of the function must be O(M).

float*** soru05(int M, int N, int K);
// Allocate a 3D dynamic array and return its address.
// M, N, K are the length of dimensions 0 (most significant), 1, 2 respectively.
// Time complexity of the function must be O(MN).

void soru06(float ***A, int M, int N, int K);
// Deallocate 3D dynamic array A. M, N, K are the length of dimensions 0 (most significant), 1, 2 respectively.
// Time complexity of the function must be O(MN). Space complexity of the function must be O(1).

void soru07(float ***A, int M, int N, int K);
// Set elements of 3D dynamic array A by obtaining their values from the user.
// M, N, K are the length of dimensions 0 (most significant), 1, 2 respectively.
// Time complexity of the function must be O(MNK). Space complexity of the function must be O(1).
```

float** **soru08**(float ***A, int M, int N, int K, int dim);
// Find and return the average of 2D arrays along the given dimension (dim) of 3D dynamic array A.
// M, N, K are the length of dimensions 0 (most significant), 1, 2 respectively.
// Time complexity of the function must be O(MNK). Space complexity of the function must be O(MN+MK+NK).

Note: Use below struct as the node of the Doubly Linked List (DLL) in the following functions.

```
typedef struct Node {
        int key;            // Assume keys are always unique.
        struct Node *next;  // Points to next node
        struct Node *prev;  // Points to previous node
} Node;
```

Node* **soru09**(Node *head, int key);
// Find the node with given key and return its address. If there is no such node, return NULL.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

void **soru10**(Node *head);
// Display the key of each node in DLL.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

int **soru11**(Node **headPtr, int key, int i);
// Insert a new node with *key* to the i-th position (starting from 1) of DLL.
// Assume the user never inserts the same key twice.
// If i is larger than the number of nodes, insert to the end of DLL.
// Return 1 if new node is inserted successfully. Otherwise, return 0.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

int **soru12**(Node **headPtr, int i);
// Delete the node in the i-th position (starting from 1) of DLL.
// If i is larger than the number of nodes, do not delete any node.
// Return 1 if the node is deleted successfully. Otherwise, return 0.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

int **soru13**(Node **headPtr, int key);
// Delete the node with the given key from DLL and return 1. If there is no node with the given key, return 0.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

void **soru14**(Node **headPtr);
// Delete all nodes in DLL and set head to NULL.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

Node* **soru15**(Node *head);
// Make a duplicate (copy) of DLL and return its address. If it fails, return NULL.
// Time and space complexity of the function must be O(N) where N is the total number of nodes in DLL.

void **soru16**(Node **headPtr);
// Reverse the order of the nodes in DLL.
// Time complexity of the function must be O(N) where N is the total number of nodes in DLL.
// Space complexity of the function must be O(1).

Node* **soru17**(Node* head1, Node* head2);
/* Combine the nodes of two sorted doubly linked lists (SDLL) (in ascending order with respect to keys) pointed by "head1" and "head2" in one SDLL and returns its address (of the first node). Return NULL if SDLL is empty. */
// Time complexity of the function must be O(N) where N is the total number of nodes in two SDLLs.
// Space complexity of the function must be O(1).

Note: Use below struct as the node of the Circular Linked List (CLL) in the following function.

```
typedef struct Node_C {
        int key;                // Assume keys are always unique.
        struct Node_C *next;    // Points to next node
} Node_C;
```

void **soru18**(Node_C **headPtr);
// Reverse the order of the nodes in CLL.
// Time complexity of the function must be O(N) where N is the total number of nodes in CLL.
// Space complexity of the function must be O(1).

Note: Use below struct as the node of the Binary Search Tree (BST) in the following functions.

```
typedef struct Node_T {
        int key, depth;             // “depth” stores the depth of the node.
        struct Node_T *left;        // Points to left child
        struct Node_T *right;       // Points to right child
        struct Node_T *parent;      // Points to parent node
} Node_T;
```

Node_T* **soru19**(Node_T *root, int key);
// Find the node with given key and return its address. If there is no such node, return NULL.
// Time complexity of the function must be O(N) where N is the total number of nodes in BST.
// Space complexity of the function must be O(1).

void **soru20**(Node_T *root);
// Display the key and depth of each node in BST with inorder traversal. You can use recursion.
// Time complexity of the function must be O(N) where N is the total number of nodes in BST.
// Space complexity of the function must be O(1).

int **soru21**(Node_T **rootPtr, int key);
// If there is no node with given key, insert a new node with *key* to BST and return 1. Otherwise, return 0.
// New nodes are always inserted at the leaf by maintaining the property of the binary search tree.
// Time complexity of the function must be O(N) where N is the total number of nodes in BST.
// Space complexity of the function must be O(1).