2.1

1. נשתמש בתור עדיפויות

כל פעם שמתרגל עצלן ירצה לאכסן תרגיל במבנה הוא יכניס אותו לתחילת התור ב-  וכל פעם שירצה לשלוף את התרגיל הישן ביותר הוא יוכל לשלוף מראש התור ב-  מה שיאפשר לאותו מתרגל מינימום השקעה באכסון ושליפה שהן רוב הפעולות.

1. נשתמש בגרף

זהו המבנה המתאים ביותר מכיוון שניתן למצוא בקלות את התלויות בניהם ע"י מציאת רכיבי קשירות ע"י אחת מהאלגוריתמים המפורסמים לשם כך, לדוגמא  ומציאת מעגל בגרף תעשה ע"י מציאת קשת אחורית ע"י הרצת  או מציאת קשת  ע"י הרצת  ולכן נקבל את הסיבוכיות הלינארית של  כאשר  מייצג את מספר הצמתים בגרף ו-  את מספר הקשתות בגרף

1. נשתמש במילון

זהו המבנה ההגיוני ביותר כאשר המפתחות יהיו שמות האנשים וה- data יהיה מספר הטלפון

מימוש קלאסי הוא ע"י Hash-table שייתן לנו סיבוכיות הכנסה וחיפוש ב-  במקרה הממוצע

1. נשתמש ברשימה מקושרת כשאר הכנסת איברים תעשה לסוף הרשימה (ולא להתחלה כמו רשימה סטנדרטית). כך שהנכסים שתקועים אצלו הכי הרבה זמן יהיו בראש הרשימה אך הוא עדיין יוכל לעבור על כולם (להבדיל מתור עדיפויות) ובנוסף הוצאת נכס מהמבנה לאחר מכירה תהיה דיי פשוטה מבחינת המימוש.

סיבוכיות הכנסה והוצאה יהיו ב- 

2.2

1. אלגוריתם
2.  - מערך של אלמנטים 

 - אינדקס לאיבר הראשון של תת המערך הנוכחי ברקורסיה

 - אינדקס לאיבר האחרון של תת המערך הנוכחי ברקורסיה

* בריצה הראשונה הפונקציה מקבלת את הערכים :

**genericMergeSort**((**Element**\*)arr,0,array\_size - 1,**strCmp**);

או שניתן גם להפעיל אותה ע"י פונקציית מעטפת מתאימה**.**

 - פונקציה המקבלת 2 אלמנטים ומחזירה:

* מספר חיובי כלשהו אם 
* 0 אם 
* מספר שלילי כלשהו אם 

**2.2 - a**

*//===================================================================================*

*//type definition*

**typedef** **void**\* **Element**;

**typedef** **int** (\***CmpFunction**)(**Element**,**Element**);

*//-----------------------------------------------------------------------------------*

*//when given 2 sorted sub-arrays of an array , the function sort the total array*

**void** **merge**(**Element**\* arr,**int** L,**int** M,**int** R,**CmpFunction** cmpFunction) {

**int** i = L , j = M + 1 , index = 0;

**Element** temp\_arr[R - L + 1];

**while** (i <= M && j <= R) {

**if** ( cmpFunction(arr[i],arr[j]) < 0) {

temp\_arr[index++] = arr[i++];

}**else**{

temp\_arr[index++] = arr[j++];

}

assert(temp\_arr[index - 1] != NULL);

}

**while** ( i <= M ) {

temp\_arr[index++] = arr[i++];

assert(temp\_arr[index - 1] != NULL);

}

**while** ( j <= R ) {

temp\_arr[index++] = arr[j++];

assert(temp\_arr[index - 1] != NULL);

}

**for** (**int** i = L , k = 0 ; i <= R ; i++) {

arr[i] = temp\_arr[k++];

assert(temp\_arr[index - 1] != NULL);

}

}

*//-----------------------------------------------------------------------------------*

*//by divide and conccer we devide the array recursivley to 2 sub-arrays and merge them*

**void** **genericMergeSort**(**Element**\* arr,**int** L,**int** R,**CmpFunction** cmpFunction) {

**if** (L >= R) **return**;

**int** M = (L + R) / 2;

**genericMergeSort**(arr,L,M,cmpFunction);

**genericMergeSort**(arr,M + 1,R,cmpFunction);

**merge**(arr,L,M,R,cmpFunction);

}

*//===================================================================================*

**2.3 - a**

*//===================================================================================*

*//declaring type Operatore*

**typedef** **int** (\***Operator**)(**int**,**int**);

*//-----------------------------------------------------------------------------------*

*/\* create a new list , initiallize all the data to 0*

*\* ----Errors:------*

*\* assumes that malloc don't failes*

*\*/*

**Node** **listCreate**(**int** size) {

**Node** list = NULL;

**for** (**int** i = 0 ; i < size ; i++) {

**Node** node = **malloc**( **sizeof**(\*node) );

assert(node != NULL);

node->*n* = 0;

node->*next* = list;

list = node;

}

**return** list;

}

*//-----------------------------------------------------------------------------------*

*//returning the size of a given list*

**int** **getListSize**(**const** **Node** **const** list) {

**Node** list\_running = list;

**int** counter = 0 ;

**while** (list\_running != NULL) {

counter++;

list\_running = list\_running->*next*;

}

**return** counter;

}

*//-----------------------------------------------------------------------------------*

*//return "true" if 2 list have the same size or "false" otherwise*

bool **areEqualSize**(**const** **Node** **const** list1,**const** **Node** **const** list2) {

**if** (**getListSize**(list1) == **getListSize**(list2)) {

**return** true;

}**else**{

**return** false;

}

}

*//-----------------------------------------------------------------------------------*

*//main function - workes as described in exercise*

**Node** **listOperator**(**const** **Node** **const** list1,**const** **Node** **const** list2,**Operator** operator) {

**if** (!**areEqualSize**(list1,list2) || !(list1 && list2 && operator) ) **return** NULL;

**Node** list = **listCreate**( (**getListSize**(list1)) );

**Node** list1\_running = list1 , list2\_running = list2 , list\_running = list;

**while** (list1\_running != NULL && list2\_running != NULL) {

list\_running->*n* = operator(list1\_running->*n*,list2\_running->*n*);

list1\_running = list1\_running->*next*;

list2\_running = list2\_running->*next*;

list\_running = list\_running->*next*;

}

**return** list;

}

*//===================================================================================*

**2.3 - b**

*//===================================================================================*

*// release all the memory allocated for list*

**void** **listDestroy** (**Node** list) {

**while** (list != NULL) {

**Node** temp = list->*next*;

**free**(list);

list = temp;

}

}

*//-----------------------------------------------------------------------------------*

*//a binary operator that returns max(a,b)*

**int** **maxOperator**(**int** a , **int** b) {

**if** (a > b)

**return** a;

**else**

**return** b;

}

*//-----------------------------------------------------------------------------------*

*/\* the function workes as described in exercise*

*\* giving R and L is equvivalent to array size*

*\*/*

**Node** **arrayOfListsOperators**(**Node**\* arr,**int** L,**int** R) {

**if** (L >= R) **return** arr[L];

**int** M = (L + R) / 2;

**Node** left\_list = **arrayOfListsOperators**(arr,L,M);

**Node** right\_list = **arrayOfListsOperators**(arr,M + 1,R);

**Node** result\_list = **listOperator**(left\_list,right\_list,**maxOperator**);

**listDestroy**(left\_list);

**listDestroy**(right\_list);

**return** result\_list;

}

*//===================================================================================*