# מת"מ ת"ב 3 – חלק יבש

## 2.1:

1. **גרף.** לכל מסלול טיסה יש נקודת מוצא(NODE) ונקודת סיום(NODE). המסלול שמקשר ביניהם הוא הקשת המחברת את ה- NODES. לכן בבעיה זו צריך גרף ששומר קבוצת צמתים(NODE) וקבוצת קשתות המחברת ביניהן. לכן המבנה הנתונים הרצוי הוא גרף, כי הוא מתאים לבעיות הדורשות אבסטרקציה של רשתות כגון רשת כבישים, רשת מחשבים וכו'.

2. **מחסנית.** כדי לבטל את השינוי האחרון(UNDO) שנעשה תוך כדי העיצוב צריך מבנה נתונים המאפשר להוציא את ה ELEMENT האחרון שנכנס למבנה הנתונים. המחסנית היא בחירה טובה של מבנה נתונים מכיוון שמאפשרת הכנסה, גישה והוצאה( **של האיבר העליון בלבד**) תוך שמירה על סדר ההכנסה ואיפשור כפילויות.

3. **רשימה.** זה הוא מבנה הנתונים המתאים ביותר לבעיה מכיוון שניתן בקלות להוסיף ולהוריד NODES מאמצע הרשימה. בנוסף, ייתכן ואותה המנה תוקלד פעמיים(ככה שקבוצה לא מתאימה) – ברשימה אין זו בעיה. ה element, מה שנראה למשתמש, של כל node יהיה שם המנה. אבל בתוך node נאגר מידע נוסף, כמו סטטוס ההזמנה. ניתן לתכנן את המערכת בצורה יפה ויעילה ככה שהרשימה תכיל שלושה "חלקים" – בהמתנה, בהכנה ומוכן. כל מנה שתשנה את הסטטוס תשובץ למקום המתאים ברשימה.

4. **קבוצה(SET).** לא ניתן להוסיף את אותו התחביב פעמיים כאשר אין חשיבות לסדר בין התחביבים(אין יתרון בשמירת הסדר). SET - קבוצה: מאפשרת הכנסת איבר פעם אחת בלבד ואינה שומרת על הסדר בין איברי הקבוצה, לכן זה הוא מבנה הנתונים המתאים.

## :2.2

```
//this function executes the quicksort algorithm, and sorts “arr” in ascending order

//param1 – void* arr: the array to be sorted(can be of any type)

//param2 – int low: the starting index from which to sort the array(should be 0 if the whole array should
//be sorted

//param3 – int high: the last index in the array(if the arrays length is n then high should equal (n-1))

//param4 – size_t size: the size in bytes of the array type(can be sent using “sizeof(type)”)

//param5 - int (*cmp)(void*,void*): a function pointer to the compare function that can compare
//between the elements in the array. if the left parameter is bigger then the right parameter the this
//function returns 1, else it returns 0.

void quickSort(void* arr, int low, int high, size_t size, int (*cmp)(void*,void*))

{

   if (low < high)

   {

      /* pi is partitioning index, arr[p] is now

         at right place */

      int pi = partition(arr, low, high, size, cmp);


      // Separately sort elements before

      // partition and after partition

      quickSort(arr, low, pi - 1, size, cmp);

      quickSort(arr, pi + 1, high, size, cmp);

   }

}
```

```
//this function places the pivot(arr[high]) in its appropriate place in the sorted array

//param1 – void* arr: the array to be sorted(can be of any type)

//param2 – int low: the starting index from which to sort the array

//param3 – int high: the last index until which to sort the array

//param3 – int high: the last index in the array(if the arrays length is n then high should equal (n-1))

//param4 – size_t size: the size in bytes of the array type(can be sent using "sizeof(type)")

//param5 - int (*cmp)(void*,void*): a function pointer to the compare function that can compare
//between the elements in the array. if the left parameter is bigger then the right parameter the this
//function returns 1, else it returns 0.

int partition (void* arr, int low, int high, size_t size, int (*cmp)(void*,void*))

{

   void* pivot = arr + size*high;    // pivot

   int i = (low - 1);  // Index of smaller element


   for (int j = low; j <= high- 1; j++)

   {

      // If pivot is bigger then the current element

      if (cmp(pivot, arr + size*j))

      {

         i++;    // increment index of smaller element

         swap(arr + size*i, arr + size*j, size);

      }

   }

   swap(arr + size*(i+1), arr + size*high, size);

   return (i + 1);

}
```

```
//this function swaps between the value in x and the value in y(by value)
//param1 – void *x: the address of the element that will be swapped with y
//param2 – void *y: the address of the element that will be swapped with x
//param4 – size_t size: the size in bytes of the type of x and y(can be sent using “sizeof(type)”)
static void swap(void *x, void *y, size_t l)
{
    char *a = x, *b = y, c;
    while(l--) //swap each byte one by one
    {
        c = *a;
        *a++ = *b;
        *b++ = c;
    }
}
```

## 2.3

```
//suumary: dynamically allocate a new node and copy the data into it

//param1 - int n: the data to be inserted into the new node

//return: the new node

Node createNode(int n) {

   Node ptr = malloc(sizeof(*ptr)); //create node

   if(!ptr) { //if the allocation failed then return NULL

      return NULL;

   }

   ptr->n = n; //set the data of the new node

   ptr->next = NULL; //set the next node to NULL for safety

   return ptr; //return the new node

}



void destroyList(Node ptr) {

   while(ptr) {

      Node toDelete = ptr;
```

```
        ptr = ptr->next;
        free(toDelete);
    }
}


//deep copies a list into a newly created list in the same order
//param1 - Node list: the list to be copied
//return: a list that is a deep copy of the parameter "list"
Node listCopy(Node list) {
    if(list == NULL) { //if the list is empty the return an empty list(NULL)
        return NULL; //return an empty list
    }
    else {
        Node return_node = createNode(list->n); //the head of the list
        Node next_new_node = return_node; //iterator for the new nodes
        Node next_old_node = list; //the iterator for the old nodes
        while(next_old_node->next != NULL) { //if end of the list was reached
            next_old_node = next_old_node->next; //go to next node
            next_new_node->next = createNode(next_old_node->n); //copy old node
```

```
            if(next_new_node->next == NULL) { //if memory allocation failed
                destroyList(return_node); //destroy all new nodes
                return NULL; //return an empty list
            }
            next_new_node = next_new_node->next; //go to next node


        }
        return return_node; //return the new list
    }
}


//pushes the "lastNode", to the end of the "list"
//param1 - Node lastNode: the node to be pushed to the end of "list"
//param2 - Node list: the list that the "lastNode" will be pushed into
//return: the head of the new list with the last_node at the end of the list
Node pushNodeToTheEndOfList(Node last_node, Node list) {
    if(list == NULL) { //if the list is empty then return the "last_node"
        return last_node;
    }
```

```
    Node temp_node = list;

    while(temp_node->next != NULL) { //loop until the end of the list

        temp_node = temp_node->next; //iterate to the next element in the list

    }

    //if last_node is NULL then the list will stay the same

    temp_node->next = last_node; //push the lastNode to the end of the list

    return list;

}



//deep copies a list into a newly created list in reversed order

//param1 - Node list: the list to be copied

//return: a list that is a reversed deep copy of the parameter "list"

Node listCopyReversed(Node list) {

    if(list == NULL) { //if the list is empty the return an empty list(NULL)

        return NULL; //return an empty list

    }

    else if(list->next == NULL) { //if the end of the original list was reached

        //make the last node in list the new first node in the list

        Node last_node = createNode(list->n);
```

```
    if(last_node == NULL) { //if memory allocation failed
        destroyList(last_node); //destroy new node
        return NULL; //return an empty list
    }
    return last_node; //return the head of the list
}
else {
    Node return_node = listCopyReversed(list->next);
    if(return_node == NULL) { //memory allocation failed
        return NULL; //return an empty list
    }
    Node last_node = createNode(list->n);
    if(last_node == NULL) { //if memory allocation failed
        destroyList(last_node); //destroy new node
        destroyList(return_node); //destroy all new nodes
        return NULL; //return an empty list
    }
    return_node = pushNodeToTheEndOfList(last_node, return_node);
    return return_node; //return the head of the list
```

```
    }
}


//summary: this function receives an array of lists and concatenates them
//one after the other, while lists in even indexes in the array will be
//concatenated in reversed order whill lists in odd indexes in the array will
//be concatenated in regular order
//NOTE: the lists will be deep copied
//param1 - Node array_of_lists[]: an array of lists(lists will be concatenated)
//param2 - int n: size of the array "array_of_lists"
//return: the new concatenated list
Node listJoinAlternating(Node array_of_lists[], int n) {
    Node return_list = NULL; //start with an empty list
    for(int i = 0; i < n; ++i) { //loop over the array
        //no need to concatenate if the list is empty
        if(array_of_lists[i] != NULL) {
            if(i % 2 == 0) { //if i is even(odd if you start counting from 1)
                //retrieve the reversed list "array_of_lists[i]"
                Node temp_node = listCopyReversed(array_of_lists[i]);
                if(temp_node == NULL) { //if memory allocation failed
```

```
            destroyList(return_list); //destroy all new nodes
            return NULL; //return an empty list
          }
          //concatenate the list to the end
          return_list = pushNodeToTheEndOfList(temp_node, return_list);
        }
        else {
          //retrieve a copy of the list "array_of_lists[i]"
          Node temp_node = listCopy(array_of_lists[i]);
          if(temp_node == NULL) { //if memory allocation failed
            destroyList(return_list); //destroy all new nodes
            return NULL; //return an empty list
          }
          //concatenate the list to the end
          return_list = pushNodeToTheEndOfList(temp_node, return_list);
        }
      }
    }
    return return_list; //return the head of the list
}
```