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;
    }
}
```