רשימות מקושרות

רשימה מקושרת – (Linked List) היא אחד ממבני נתונים הבסיסיים ביותר במדעי המחשב, כשמטרתה אחסון נתונים בצורה יעילה. הרשימה המקושרת היא אוסף מבנים המפוזרים בזכרון ולאו דווקא נמצאים ברצף (כמו איברי המערך), בכל איבר מאוחסן מידע (אחד מאותם נתונים אותם רצינו לאחסן) וכן מצביע לאיבר הבא ברשימה. היא מבנה נתונים דינאמי וגמיש : ניתן להוסיף או להוציא איברים עפ"י הצורך ללא צורך בהכרזה מראש על מספר איברים מקסימאלי.

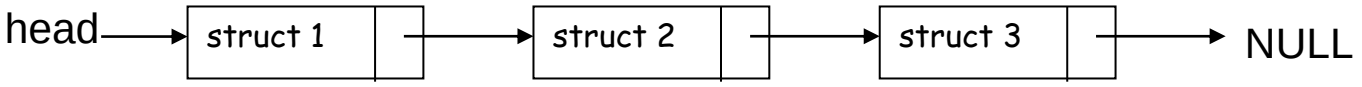
<u>יתרונות לרשימה</u>

1. מאפשר הקצאת זיכרון דינמית בעת הצורך.
2. אין צורך להקצות מקום זיכרון רצוף.
3. הוצאת איבר מאמצע רשימה לא מותירה "חור".

<u>יתרונות למערך</u>

1. גישה לאיבר לפי אינדקס בזמן $O(1)$.

מבנה רשימה מקושרת :



רשימות מקושרות

- כל ***צומת*** ברשימה מקושרת זה משתנה מסוג מבנה שמכיל מצביע למשתנה מאותו סוג.

הגדרת מבנה עבור צומת ברשימה:

```
typedef struct Node
{
    ........
    struct Node* next;
}Node;
```

שדות מידע

מצביע לצומת הבא

- עבור כל רשימה יש להגדיר מצביע לראש הרשימה – מצביע לצומת הראשון ברשימה. **חובה לאפס אותו!!**

Node * head = NULL;

- מצביע לראש הרשימה מכיל NULL כאשר הרשימה לא קיימת – עדיין לא נבנתה או כבר שוחררה.

- המבנה האחרון ברשימה חייב להצביע ל-NULL, עלינו לדאוג לזה.
- לפעמים יהיה נוח לשמור גם:

מצביע לסוף הרשימה,משתנה עבור כמות האיברים ברשימה וכו'

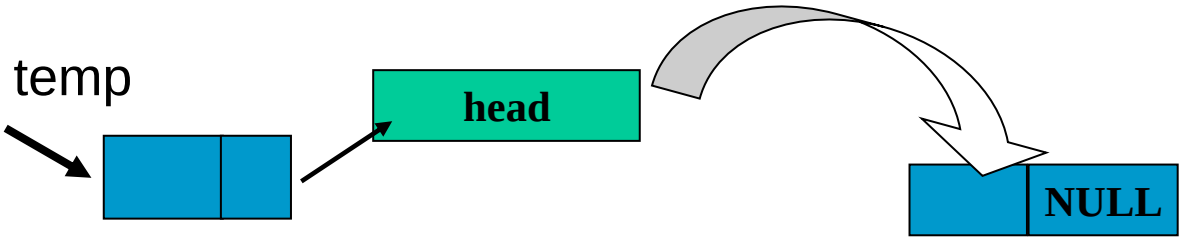
רשימה – הוספת צומת (בצורה סכמאטית)

- **יצירת צומת temp (תמיד הקצאה דינאמית!!!!)**

```
temp = (struct Node *)malloc(sizeof(struct Node));
```

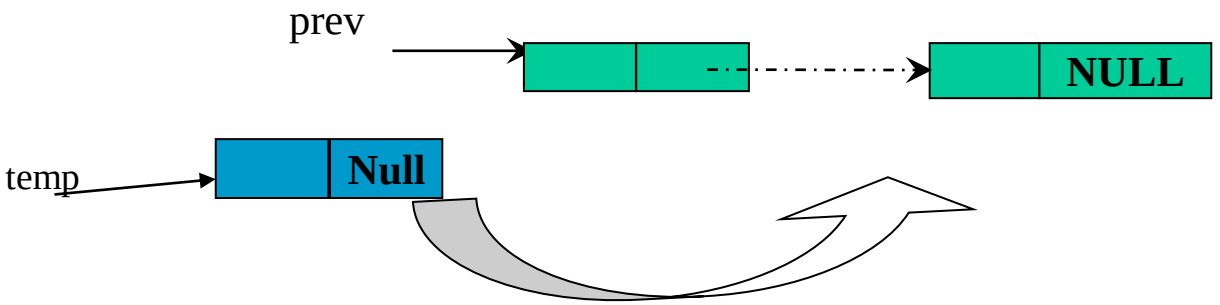
- הכנסת צומת **temp** לראש הרשימה
(גם לרשימה ריקה וגם ללא ריקה)

```
temp->next = head;
head = temp;
```



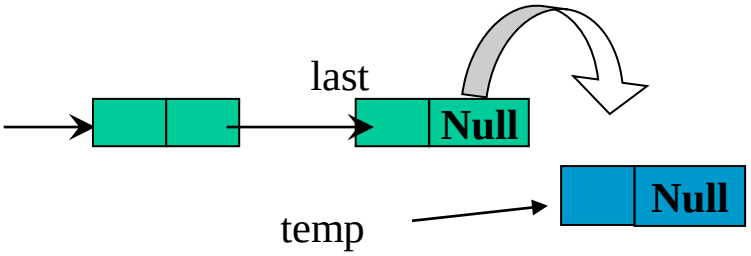
רשימה – הוספת צומת (בצורה סכמאטית)

- הוספת צומת temp אחרי צומת prev




```
temp->next = prev->next;
prev->next = temp;
```

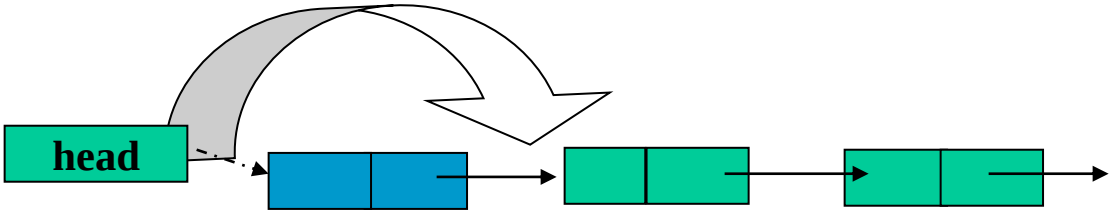
- הוספת צומת temp לסוף הרשימה.**בדרך כלל,זה אינו מקרה מיוחד אלא זה כלול במקרה כללי**




```
last->next = temp;
```

רשימה – מחיקת צומת (בצורה סכמאטית)

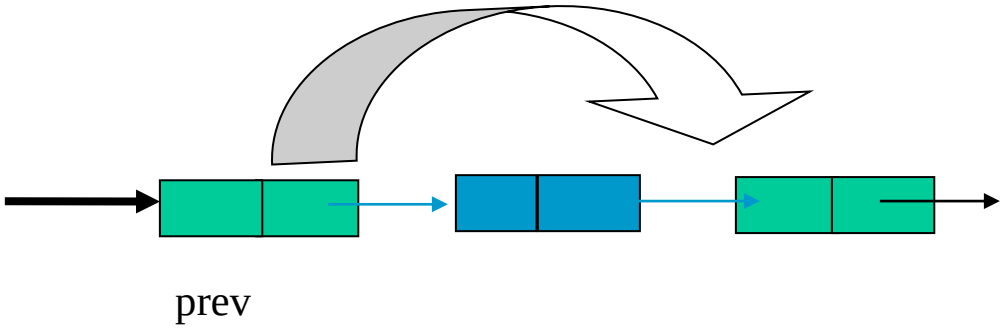
- מחיקת צומת ראשון ברשימה




```
temp = head;
head = head->next;
free (temp);
```

- מחיקת צומת temp שבא אחרי prev

```
temp = prev->next;
```




```
prev->next = temp->next;
free (temp);
```

עידכון מצביע לראש הרשימה - חשוב!

כפי שלמדנו, אם רוצים שפונקציה תשנה את התוכן של משתנה שהוגדר בפונקציה אחרת, ישנן שתי שיטות –

1. מעבירים לפונקציה את הכתובת של המשתנה לתוך מצביע.
2. משתמשים באופרטור return כדי להחזיר את המשתנה.

נראה את שתי השיטות עבור עידכון מצביע לראש הרשימה.

<u>מעבירים לפונקציה את הכתובת של המשתנה :</u>

```
int main()
{
    struct Node* head = NULL;
   ……
   func(&head);
   ……
}

void func(struct Node **p)
{
  *p = ….
    …….
}
```

עידכון מצביע לראש הרשימה - חשוב!

משתמשים באופרטור return כדי להחזיר את המשתנה.

```
int main()
{
    struct Node* head = NULL;
   .........
   head = func(head);
   .........
}

struct Node * func(struct Node *p)
{
  .......
  return p;
}
```

בשקף הבא דוגמא 1 ברשימות מקושרות. בדוגמא הזאת נבנית רשימה מהנתונים שונים זה מזה. נא לשים לב שעידכון המצביע נעשה בעזרת שיטה המופיעה בשקף הזה.

דוגמא 1

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

/* declaration of node type */
typedef struct Node
{
    int val;
    struct  Node *next;
} Node;

/* declaration of functions */

/* creating a new node with a given value */
Node* createNode (int value);

/* adding a value to an unsorted list */
Node* insertList (Node *head, int value);

/* adding a value to a sorted list */
Node* insertSortedList (Node *head, int value);

/* deleting a value from an unsorted list */
Node* deleteFromList (Node *head, int value);
```

דוגמא 1

```
/* printing the elements in a list */
void printList (Node *head);

/* freeing the memory of the list nodes */
void freeList (Node *head);


int main()
{
 int array[10] = {23, 2, 78, 901, 1001, 3, 80, 23452, 5, 67};
 Node *unsorted_head = NULL, *sorted_head = NULL;
 int i;

    /* initializing the lists with the elements in the array */
 for (i = 0; i < 10; i++)
 {
    unsorted_head = insertList (unsorted_head, array[i]);
    sorted_head = insertSortedList (sorted_head, array[i]);
}
```

בגלל שפונקציות מחזירות ערך, המצביעים האלה מתעדכנים והשינוי שנעשה בפונקציות למצביעים, נשמר.

דוגמא 1

```
/* printing */
    printList (unsorted_head);
    printf ("\n");
    printList (sorted_head);
    printf ("\n");


/* deleting some elements from unsorted list */
 for (i = 0; i < 10; i++)
   if( (i%3) == 0)
      unsorted_head =deleteFromList (unsorted_head,
       array[i]);
   /* printing */
  printList (unsorted_head);
 printf ("\n");
 printList (sorted_head);
 printf ("\n");

/* freeing the lists' dynamically allocated memory */
freeList (unsorted_head);
freeList (sorted_head);

return 0;
}
```

דוגמא 1

```
/* creating a new node with a given value */
Node* createNode (int value)
{
        /* allocating the memory */
        Node *ptr = (Node*) malloc (sizeof(Node));

        if(ptr!=NULL)
        {           /* initializing */
            ptr->val = value;
            ptr->next = NULL;
        }
        return ptr;
}
```

דוגמא 1

```
/* adding a value to an unsorted list */
Node* insertList (Node *head, int value)
{
    /* creating a new node */
    Node *ptr = createNode (value);

    if(ptr == NULL)
    {
      freeList (head);
      printf("Allocation problem");
      exit(1);
    }
     /* adding the new node to the head of the list */
    ptr->next = head;
    head = ptr;

return head;
}
```

דוגמא 1

```
/* adding a value to a sorted list */
Node* insertSortedList (Node *head, int value)
{
  Node *ptr,*pCurrent,*pPrev;
  /* creating a new node */
  pCurrent = pPrev = head;

  ptr = createNode (value);
  if(ptr == NULL)
  {
     freeList (head);
     printf("Allocation problem");
     exit(1);
  }
```

המשך בשקף הבא...

דוגמא 1

```
  /* adding the new node to the correct place in the list */
  while ( pCurrent != NULL )
  {
     if(pCurrent->val > ptr->val)
        break;
    pPrev = pCurrent;          /*moving two pointers /*
    pCurrent = pCurrent->next;
  }
  ptr->next = pCurrent;
  if(pCurrent == head)  /*first element or empty list*/
     head = ptr;
  else
     pPrev->next = ptr;

 return head;
}
```

דוגמא 1

```
/* deleting a value from an unsorted list */
Node* deleteFromList (Node *head, int value)
{
    Node *pCurrent ,*pPrev,*pNext;

    pCurrent = head;
    pPrev = NULL;
    /* finding the right node to delete */
    while ( pCurrent != NULL )
    {
      if(pCurrent->val == value)
      {
         pNext = pCurrent->next;
         free(pCurrent);                    /* delete node */
         if (pPrev == NULL)      /*It was a first element*/
            head = pNext;
         else
            pPrev->next = pNext;
        break;
      }
      else {
         pPrev = pCurrent;
         pCurrent = pCurrent->next;
      }
    }
      return head;  }
```

דוגמא 1

```
/* printing the elements in a list */
void printList (Node *head)
{
    /* going over the element and printing each one */
    while (head != NULL)
    {
        printf ("%d ", head->val);
        head = head->next;
    }
}

/* freeing the memory of the list nodes */
void freeList (Node *head)
{
    Node *temp;
    while (head != NULL)
    {
        temp = head;
        head = head->next;
        free (temp);

    }
}
```

רשימות מקושרות – סיכום של נקודות חשובות

- כאשר מחליטים לעבוד עם רשימה מקושרת, יש להגדיר מצביע לתחילת הרשימה ולאפסו ל-NULL.
- יצירת איבר חדש ברשימה - באמצעות הקצאה דינאמית
- הוספה לרשימה / מחיקה מרשימה - לשים לב למקרי קצה! בעיקר - יש לשים לב האם יש צורך לעדכן את המצביע לראש הרשימה או לא. כמו כן, יש לבדוק שהרשימה לא ריקה. תשימו לב שניסיון לפנות לשדות של מבנה מהמצביע שהוא שווה ל-NULL גורם לשגיאת הרצה!
- יש לעבוד בצורה בטיחותית! לא לנתק צמתים מהרשימה לפני שבדקתם ששאר הצמתים לא הולכים לאיבוד! ניתן ומומלץ להשתמש במצביעי עזר.
- יש לקחת בחשבון שאם הגענו לצומת כלשהו, לא ניתן להגיע לצומת שלפני (רשימה חד- כיוונית). יש לתכנן את ההתקדמות בהתאם.

רשימות מקושרות – סיכום של נקודות חשובות

- כאשר עוברים על הרשימה בצורה הבאה:

```
while(head != NULL) {}
```

מתקיימים דברים הבאים:

1. מטפלים בכל הצמתים ברשימה
2. בסיום, head = NULL

- כאשר עוברים על הרשימה בצורה הבאה:

```
while(head->next != NULL) {}
```

מתקיימים דברים הבאים:

1. לא מטפלים במבנה האחרון ברשימה.
2. בסיום, head מצביע על המבנה האחרון ברשימה.