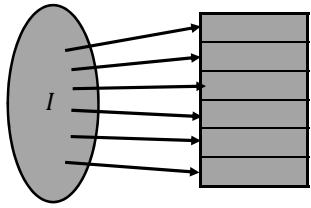


## מערך כמבנה נתונים



**מערך מוגדר ע"י הפעולות הבאות:**  
`create(type, I)` מחזיר מערך  $A$  של איברים מטיפוס  $type$  כאשר האינדקסים הם הקבוצה הסופית  $I$ .  
`get(A, i)` מחזיר את הערך של האיבר עם אינדקס  $i \in I$  בתוך  $A$ .  
`store(A, i, e)` מאחסן במערך  $A$ , תחת אינדקס  $i \in I$ , את ערך הביטוי  $e$ .

לדוגמא, בשפת  $C$  האינדקסים הם הקבוצה  $I = \{0, \dots, n - 1\}$ .  
 אחזר `(get)`:  $A[i]$   
 שמור `(store)`:  $A[i] = e$

**כללים הנשמרים ע"י הפעולות:**

- כל הערכים במערך מאותו טיפוס, והוא נקבע בהכרזה ( $type$ ).
- הערך המאוחזר לפי אינדקס  $i$ , הוא הערך האחרון שנשמר לפי אינדקס  $i$ .
- אחזור לפי אינדקס  $i$  מחזיר ערך בלתי מוגדר אם מעולם לא אוחסן ערך לפי אינדקס זה.

## מערכים, מטריצות דלילות, ורשימות מקושרות

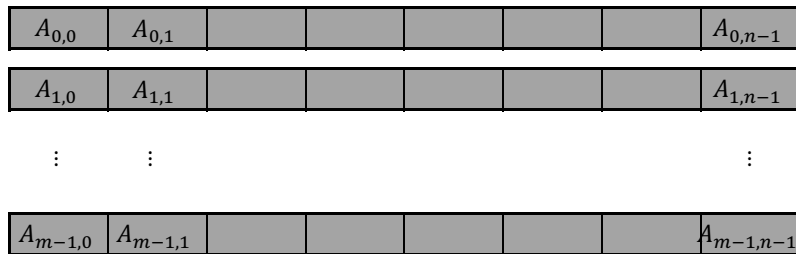
### חומר קריאה לשיעור זה

Chapter 11.2- Linked lists (204 - 213)

## מעריך דו-מימדי

`int A[m][n];`

כל שורה  $A[i]$  היא מערך חד-מימדי באורך  $n$ .



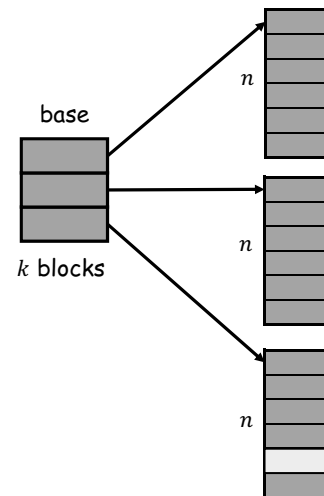
נניח כי  $base$  היא הכתובת של  $A[0][0]$  והמימוש הוא כאזור זיכרון רצוף.

$base + i \cdot n$       הכתובת של שורה  $A[i]$  היא  
 $base + i \cdot n + j$       הכתובת של איבר  $A[i][j]$  היא

## מימושים למערך

1. אזור זיכרון רצוף. כל הפעולות מתבצעות בזמן  $O(1)$ .

2. מספר אזורים בזיכרון של כל אחד גודל קבוע  $n$ :



כדי לממש מערך בגודל  $m$  יש לבחור  $k = \lceil m/n \rceil$ . האיבר  $A[i]$  נמצא בכתובת  $base[\lfloor i/n \rfloor] + i \% n$ .

לדוגמא, כאשר  $n = 6, i = 16$  הכתובת של  $A[16]$  מחושבת ע"י  
 $base[\lfloor 16/6 \rfloor] + 16 \% 6 = base[2] + 4$

זמן חישוב הכתובת  $O(1)$ .

## מערך רב-מימדי (המשך)

הנוסחה לחישוב הכתובת במערך רב ממדי דומה ל**לכלל הורנר** לחישוב ערך פולינום:  

$$p(x_0) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0 = (\dots (a_k x + a_{k-1}) x + \dots + a_1) x + a_0$$

התוכנית הבאה מחשבת את ערך הפולינום בנקודה  $x$ :

```
p=a[k];
for (j=k-1; j>=0; j--)
    p = p*x + a[j]
```

באופן אנלוגי, התוכנית הבאה מחשבת את הכתובת של  $A[i_d] \dots [i_3][i_2][i_1]$  הנתונה ע"י:  

$$base + ((\dots ((i_d n_{d-1} + i_{d-1}) n_{d-2} + i_{d-2}) n_{d-3} + \dots + i_3) n_2 + i_2) n_1 + i_1$$

```
addr=i[d];
for (j=d-1; j>=1; j--)
    addr = addr*n[j] + i[j];
addr= base + addr;
```

סיבוכיות  $O(d)$ .

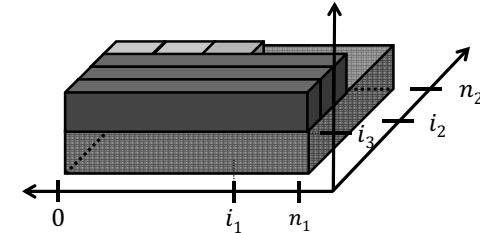
## מערך רב-מימדי

במערך דו-ממדי  $int A[n_2][n_1]$  הכתובת של איבר  $A[i_2][i_1]$  מחושבת, כפי שראינו, ע"י הנוסחה:  

$$base + i_2 n_1 + i_1$$

במערך תלת-ממדי  $int A[n_3][n_2][n_1]$  הכתובת של איבר  $A[i_3][i_2][i_1]$  מחושבת ע"י הנוסחה:  

$$base + (i_3 n_2 + i_2) n_1 + i_1 = base + i_3 \cdot n_2 \cdot n_1 + i_2 \cdot n_1 + i_1$$



**לדוגמה:**  
 התא  $A[1][3][2]$   
 (במרכז)

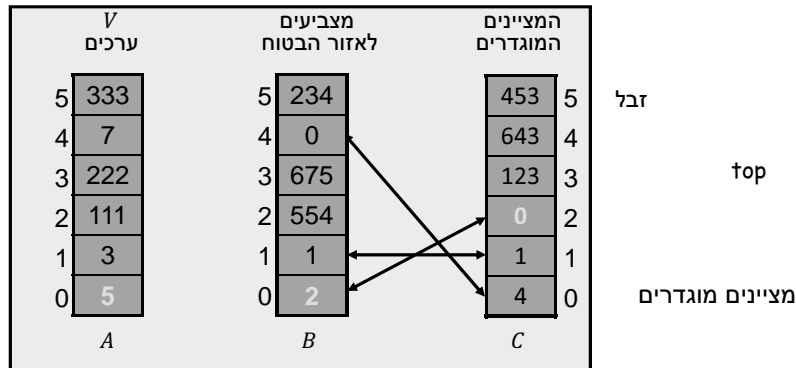
במערך רב-ממדי  $int A[n_d] \dots [n_3][n_2][n_1]$  הכתובת של  $A[i_d] \dots [i_3][i_2][i_1]$  מחושבת ע"י הנוסחה:  

$$base + i_d \cdot n_{d-1} \cdot \dots \cdot n_1 + i_{d-1} \cdot n_{d-2} \cdot \dots \cdot n_1 + \dots + i_1$$

$$= base + ((\dots ((i_d n_{d-1} + i_{d-1}) n_{d-2} + i_{d-2}) n_{d-3} + \dots + i_3) n_2 + i_2) n_1 + i_1$$

## איתחול מערך בזמן $O(1)$

**הבעיה:** בייצוג הרגיל אתחול מערך דורש זמן  $O(n)$ . נראה ייצוג שיאפשר לאתחל מערך בזמן  $O(1)$ . בתמורה נשתמש ביותר זיכרון.



בדוגמא זו:  $V[0]=5, V[1]=3, V[4]=7$



## סריקת מערך רב-מימדי

זמן סריקת כל איברי מערך  $A[n_d] \dots [n_3][n_2][n_1]$  תלוי בסדר הסריקה:

סריקה לפי סדר האחסון מהירה יותר בגלל **לוקליות** של גישה לזיכרון. במימוש שהצגנו, סריקה כאשר האינדקסים הנמוכים נסרקים ראשונים מהירה יותר מסריקה בה האינדקסים הגבוהים נסרקים ראשונים. השיפור הוא בקבוע בלבד ואינו מתבטא בסימון  $O$ .

## הקוד לפעולות המערך

```
top = 0;
constant = const;
```

אתחול  
:init(V,const)

```
if (is_garbage(i))
    return constant;
else
    return A[i];
```

אחזר  
:get(V,i)

```
if (is_garbage(i)) {
    C[top] = i;
    B[i] = top;
    top = top + 1;
}
A[i] = e;
```

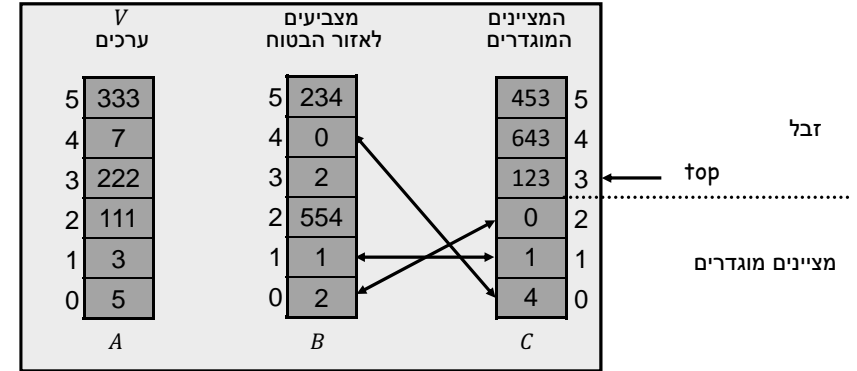
שמור  
:store(V,I,e)

הערה: ברגע שהמערך מתמלא ( $top > n$ ) ניתן להשתמש במערך  $A$  בלבד, כיוון שאין, ויותר לא יהיה, זבל ב- $A$ .

## הקוד לאיתור זבל

```
int is_garbage(int i) {
    return !(( B[i] < top && (B[i] >= 0) && C[B[i]] == i));
}
```

דוגמא לאיבר שהוכנס למבנה:  $C[B[4]] == 4$   
דוגמאות לזבל:  $C[B[3]] = 0$  וכן  $C[B[5]] > top$



## ייצוג מטריצות דלילות

מטריצה דלילה היא מערך דו-ממדי עבורו "רוב מוחלט" של האיברים שווה לאפס (או לקבוע  $c$  כלשהו). מהו רוב מוחלט?

בדוגמא זו מתוך 40 איברים שונים מהקבוע.

	0	1	2	3	4	5	6	7
0			3			1		
1		5						
2				4				
3								
4		2			7	8		

הגדרה אסימפטוטית: אם סדרת מטריצות  $M_1 \dots M_n$  מקיימת שמספר האיברים השונים מקבוע  $c$  הוא מסדר גודל של  $o(m(n))$  (קטן) כאשר  $m(n)$  הוא מספר המקומות המקסימלי במטריצה  $M_n$  אז זוהי סדרה של מטריצות דלילות.

דוגמא: מטריצות אלכסוניות. במטריצות ריבועיות דלילות מגודל  $n \times n$  מספר האיברים השונים מקבוע  $c$  הוא  $o(n^2)$ .

ייצוג: רשימת שלשות  $(i, j, A_{i,j})$  מסודרות בסדר לקסיקוגרפי. דוגמא:  $(0,2,3), (0,5,1), (1,1,5), (2,3,4), (4,1,2), (4,4,7), (4,5,8)$

## ייצוג מטריצות סימטריות

מטריצה סימטרית היא מערך דו-ממדי המקיים  $A[i, j] = A[j, i]$  לכל  $i, j$ .

	0	1	2	...	$n$
0	1				
1	2	3			
2	4	5	6		
:	7	8	9	10	
$n$	...		...		...

הייצוג: אוסף וקטורים המאוחסנים במערך רציף יחיד: 1 2 3 4 5 6 7 8 9 10 ....

מהי הכתובת של איבר  $A[i, j]$ ? נשתמש בנוסחה:  $1 + 2 + \dots + i = i(i + 1)/2$

$$Addr(i, j) = \begin{cases} Base + i(i + 1)/2 + j & \text{if } i \geq j \\ Addr(j, i) & \text{Otherwise} \end{cases}$$

חצי מערך תחתון; חצי מערך עליון;

## תוכנית לחיבור מטריצות $C = A + B$

```
typedef struct NODE {
    int row, col;
    float val;
}

int lex(NODE a, NODE b){
    // lexicographically compares a and b;
    if (a.row,a.col) < (b.row,b.col) return -1;
    if (a.row,a.col) = (b.row,b.col) return 0;
    if (a.row,a.col) > (b.row,b.col) return 1;
    assert(false); return ERROR; // impossible
}

NODE A[mA+1], B[mB+1], C[mA+mB+1];
```

```
void ADD(NODE *A, NODE *B, NODE *C, int *mC){
    int i=0, j=0, *mc=0;
    A[mA]=B[mB]={+∞,+∞, 0};
    while (i < mA || j < mB){
        switch lex(A[i],B[j]){
            case -1: C[*mC++] = A[i++]; break;
            case 1: C[*mC++] = B[j++]; break;
            case 0: C[*mC].row = A[i].row;
                   C[*mC].col = A[i].col;
                   C[*mC].val = A[i++].val + B[j++].val;
                   if (C[*mC].val ≠ c) *mC++;
        }
    }
}
```

## מטריצות דלילות (המשך)

הייצוג: רשימת שלשות  $(i, j, A_{i,j})$  מסודרות בסדר לקסיקוגרפי.

חסרון עקרי: אין גישה אקראית לפי מציין בזמן  $O(1)$ .

יתרונות: חוסך בזיכרון עבור מטריצות דלילות. מאיץ פעולות חיבור וכפל של מטריצות דלילות.

חיבור מטריצות בגודל  $n \times n$  לוקח בייצוג רגיל זמן  $O(n^2)$ .

נניח כעת שיש  $m$  איברים שונים מהקבוע  $c$  במטריצה אחת ו- $k$  בשנייה. חיבור שתי המטריצות נעשה ע"י מיזוג שתי הרשימות המייצגות את המטריצות נ"ל. לזמן המיזוג הוא אורכן הכולל של הרשימות כלומר  $O(m+k)$ . כלומר עבור מטריצות ריבועיות דלילות זמן החיבור הוא  $O(n^2)$ .

תרגיל ממבחן: הראו שכפל מטריצות ריבועיות דלילות בייצוג זה לוקח זמן  $O(n^3)$  במקום  $O(n^3)$ . לשם כך נדרש מיון לפי עמודות של מטריצה  $B$ .

## חיפוש ברשימה מקושרת

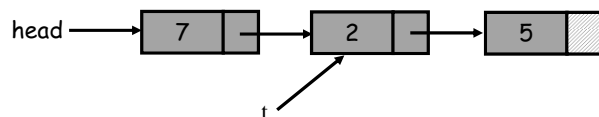
```
void init (NODE *head){
    * head = NULL;
}
```

פעולת איתחול  
:init(head)

```
NODE * find (DATA_TYPE x, NODE *head){
    NODE *t;
    t = head;
    while (t!=NULL && t->info != x)
        t = t->next ;
    return t;
}
```

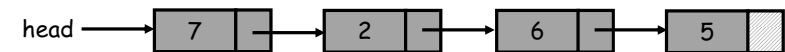
פעולת חיפוש  
:find(x,head)

חיפוש איבר בזמן  $O(n)$  במקרה הגרוע.



## רשימות מקושרות

נזכר כעת כיצד מתבצע חיפוש, הכנסה, והוצאה: find, insert, delete ברשימות מקושרות ונגדיר וריאציות עליהן.



חסרונות בהשוואה למערך:

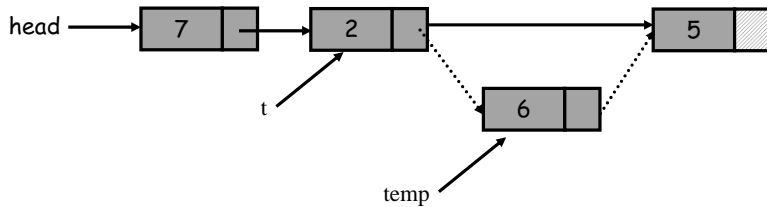
- מאפשר הקצאת זיכרון דינמית.
- אין צורך להקצות מקום זיכרון רצוף.
- ניתן להוציא איבר מתוך רשימה מקושרת בלי להשאיר "חור" כמו במערך ולכן ניתן לסרוק את כל האיברים בזמן ליניארי.

## הוצאת איבר מרשימה מקושרת

```
delete ( NODE *t){
    NODE * temp ;
    temp = t → next; /* מצביע לצומת שמורידים */
    t → next = temp → next ;
    free (temp) ;
}
```

פעולת הסרה  
:delete(t)  
הפרמטר t מצביע לצומת שלפני הצומת שמוציאים.

הוצאת איבר בזמן  $O(1)$  כאשר ידוע מקום ההוצאה.



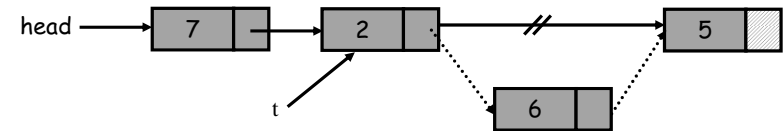
## הכנסת איבר לרשימה מקושרת

```
int insert ( NODE *t, DATA_TYPE x){
    NODE *p;
    if (t == NULL) return 0;
    p = (NODE *) malloc (sizeof (NODE));
    p → info = x;
    p → next = t → next ;
    t → next = p ;
    return 1;
}
```

פעולת הכנסה  
:insert(t,x)

הפרמטר t מצביע לצומת שאחרי מוסיפים צומת חדש.

הכנסת איבר בזמן  $O(1)$  כאשר ידוע מקום ההכנסה.  
דוגמא: insert(t,6)

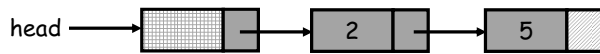


## רשימות עם כותרת

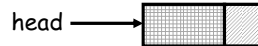
מוסיפים איבר "ריק" בתחילת הרשימה (נקרא לרוב "dummy node" באנגלית). תוספת זו חוסכת:

- טיפול מיוחד ברשימות ריקות.
- טיפול מיוחד בהסרת הצומת הראשון.
- טיפול מיוחד בזמן הכנסה לפני הצומת הראשון.

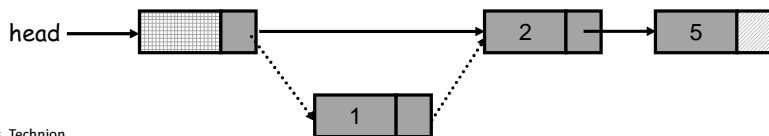
בייצוג זה רשימה בת שתי איברים מיוצגת כך:



ורשימה ריקה מיוצגת כך:



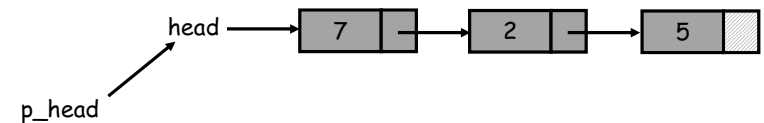
הכנסה לפני איבר ראשון זהה להכנסה לפני איבר כלשהו:



## הוצאת איבר מראש רשימה מקושרת

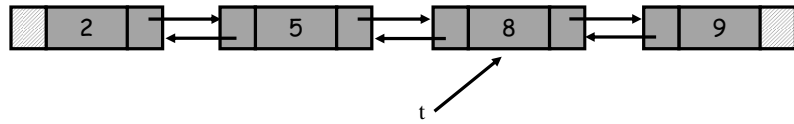
```
int delete_first ( NODE **p_head){
    NODE * temp ;
    if (*p_head == NULL) return 0 ;
    temp = *p_head;
    *p_head = temp → next ;
    free (temp) ;
    return 1; /* success code */
}
```

פעולת מחיקת צומת ראשון  
:delete first(r)



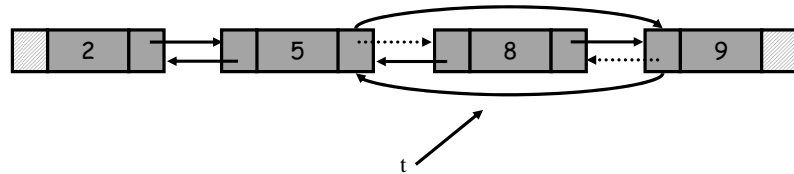
תכנות זה מסורבל. מהו הפתרון לכך ?

## רשימה מקושרת דו-כיוונית

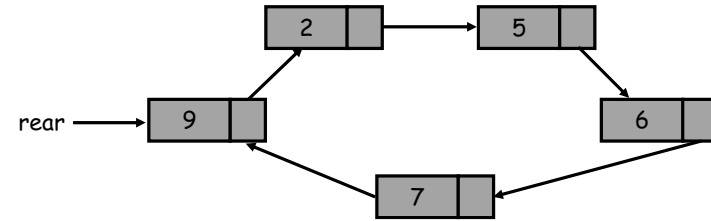


יתרון: מאפשר להוציא איבר בהינתן מצביע t אליו (ולא רק את האיבר שאחריו).

```
t -> next -> prev = t -> prev ;
t -> prev -> next = t -> next ;
```

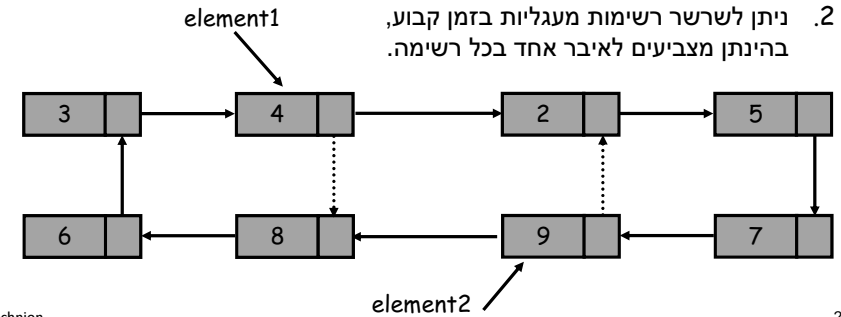


## רשימות מעגליות



יתרונות:

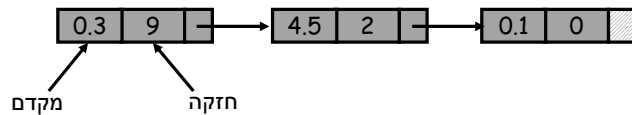
- אפשר להגיע מכל איבר לכל איבר.
- ניתן לשרשר רשימות מעגליות בזמן קבוע, בהינתן מצביעים לאיבר אחד בכל רשימה.



## ייצוג פולינומים

$$0.3x^9 + 4.5x^2 + 0.1$$

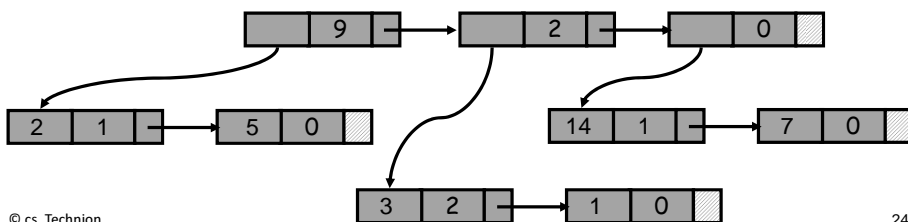
נעלם אחד. דוגמא:



מימוש זה יעיל כאשר הפרש בין החזקה הגבוהה והנמוכה גדול וכן הרבה מקדמים של חזקות הביניים מתאפסים.

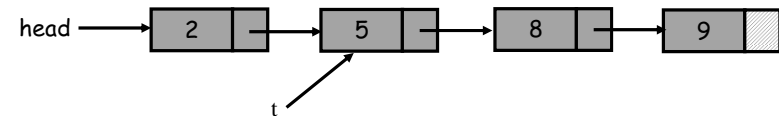
$$(2y + 5)x^9 + (3y^2 + 1)x^2 + (14y + 7)$$

שני נעלמים. דוגמא:

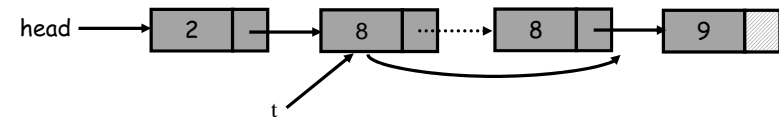


## הוצאה "טריקית"

האם ניתן להוציא איבר אליו מצביע t גם מרשימה חד-כיוונית ?



פשוט נעתיק את האינפורמציה בתא העוקב ל- t לתא ש- t מצביע אליו ונוציא את התא העוקב.



מהן החסרונות של שיטה זו ?

- צומת יכול להכיל הרבה אינפורמציה ולכן העתקה עלולה להיות פעולה ארוכה.
- יתכן וישנם מצביעים נוספים לתא שהוצא ואין אפשרות לעדכן מצביעים אלה.
- לא עובד עבור האיבר האחרון.

## כפל מטריצות דלילות

```

#define N_row 20
#define N_col 20
NODE *row_header[N_row],
      *col_header[N_col];
/* Compute the value of c[i][j] */
float mult_row_col(int i, int j){
    float c = 0;
    NODE *r = row_header[i],
          *k = col_header[j];

    while (r != NULL && k != NULL){
        if (r->column < k->row) r = r->next;
        else if (r->column > k->row) k = k->down;
        else /* r->column = k->row */
            c += r->value * k->value;
            r = r->next; k = k->down;
    }

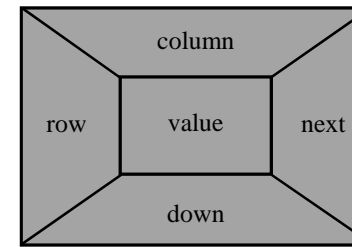
    return c;
}

```

:C = AB מטריצות  
 התוכנית מחשבת את  
 האיבר  $C_{ij}$ .

## ייצוג מטריצות דלילות

מבנה צומת:



```

typedef struct node {
    float value;
    struct node*next, *down;
    int row, column;
} NODE;

```

