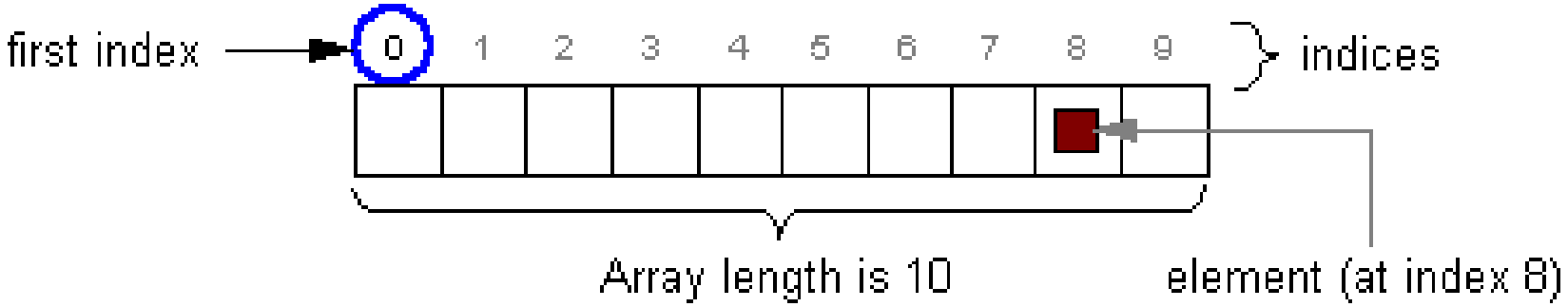
# מבני נתונים

סופיה קירמברג

# מערך חד-ממדי

מערך הינו מבנה זיכרון המכיל מספר ערכים מאותו טיפוס, גודל המערך נקבע כאשר הוא נוצר (בזמן ריצה) ולאחר מכן הוא קבוע

```
int[] arr = new int[10];
```




```
arr[8] = 3;

System.out.println(arr[8]);
```

# הגדרה פונקציונאלית של מערך

אתחול (סוג ,אורך)

כתיבה (מיקום ,ערך)

- מצליחה למיקום בין 0 לאורך ולערך השייך לסוג
- נכשלת למיקום שלילי או מהאורך ומעלה
- נכשלת לערך שלא שייך לסוג

קריאה (מיקום)

- מחזירה את הערך האחרון שנכתב למיקום ,למיקום בין 0 לאורך
- נכשלת למיקום שלילי או מהאורך ומעלה
- נכשלת למיקום שלא הייתה אליו כתיבה

# יעילות הפעולות

אתחול

- ממופה להקצאת זיכרון על-ידי מערכת ההפעלה
- העלות לא יותר מ-O(n) כאשר n מספר האיברים

כתיבה וקריאה של איבר כלשהו בין 0 ל- n-1

- ממופות לפעולות load ו-store של המעבד
- העלות חסומה על-ידי זמן הקריאה המקסימאלי (קריאה מהדיסק)
  - בדרך כלל, ניתן לבצע במחזורי שעון בודדים – מיליארדיות שניה
  - גישה ישירה ב-O(1) לתא הזיכרון לפי החישוב של

    base address + index X size

# מערך חד-מימדי

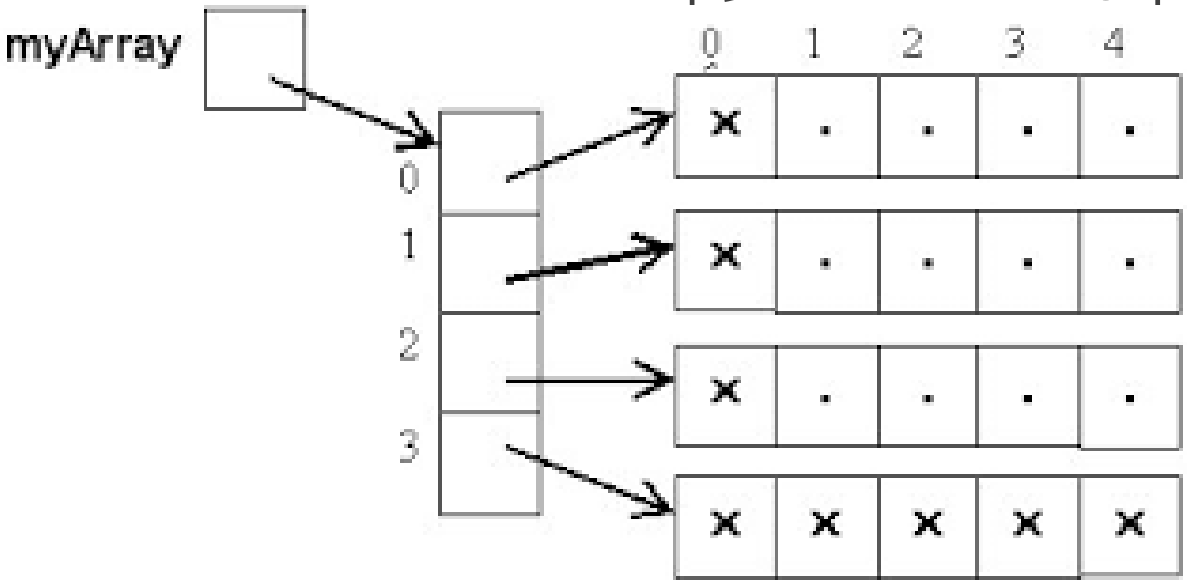
אתחול מערך בזמן ההגדרה (Java)

```
int[] arr = {2,3,5,7,11};
```

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | 2 | 3 | 5 | 7 | 11 |

# מערכים רב-ממדיים

מערך דו-ממדי מורכב ממערך שכל איבר בו הוא מערך חד ממדי

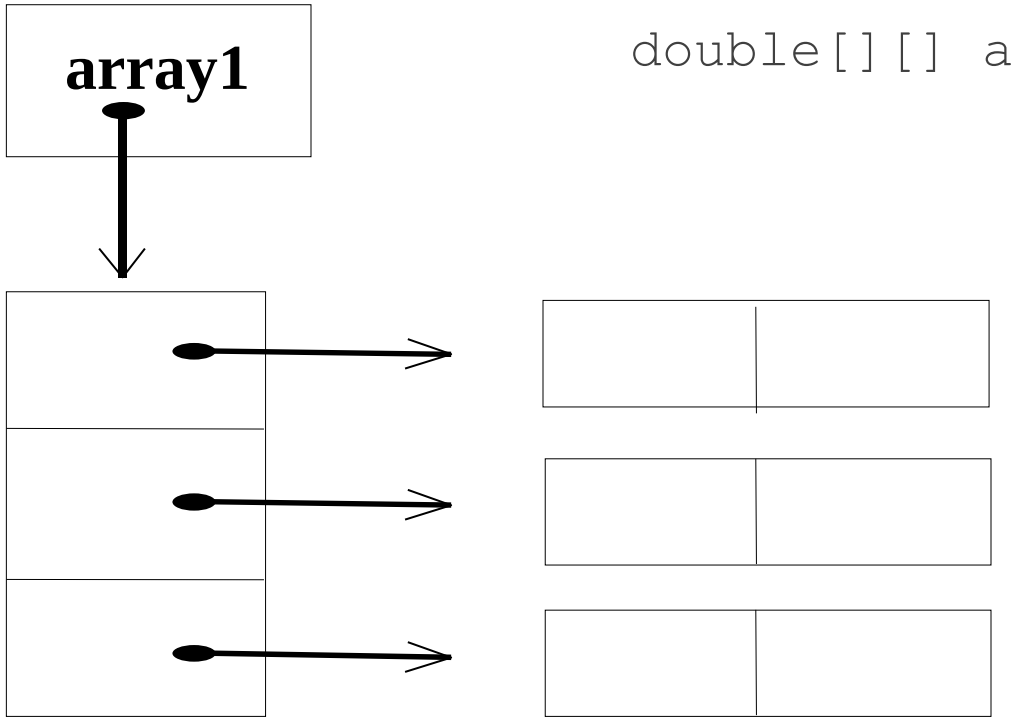


# מערכים רב-ממדיים

שני אופנים להגדרת מערך דו-ממדי:

- מטריצות מלבניות

```
double[][] array1 = new int[3][2];
```

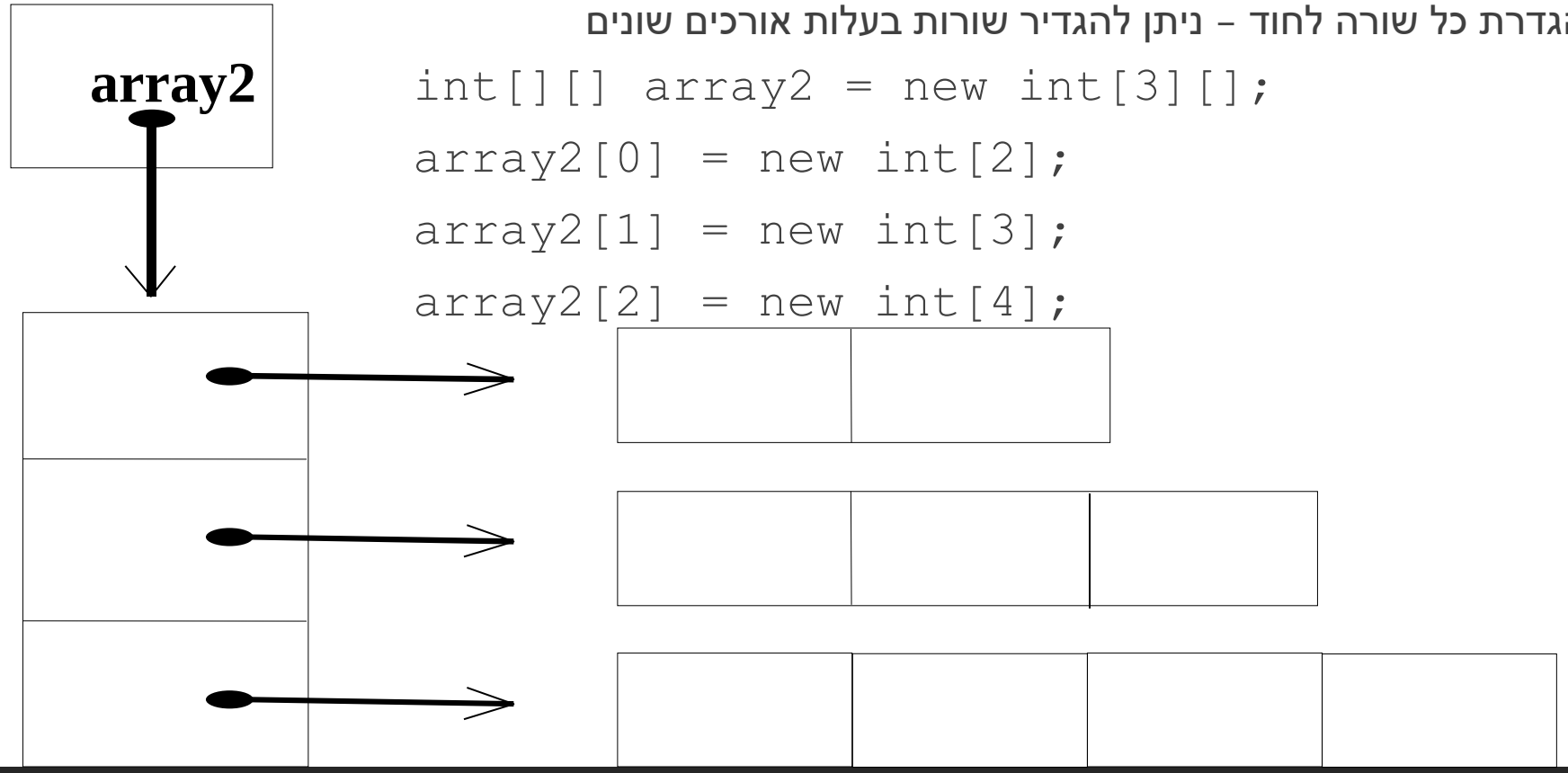


# מערכים רב-ממדיים

שני אופנים להגדרת מערך דו-ממדי:

- הגדרת כל שורה לחוד – ניתן להגדיר שורות בעלות אורכים שונים

```
int[][] array2 = new int[3][];
array2[0] = new int[2];
array2[1] = new int[3];
array2[2] = new int[4];
```



# מערכים רב-ממדיים

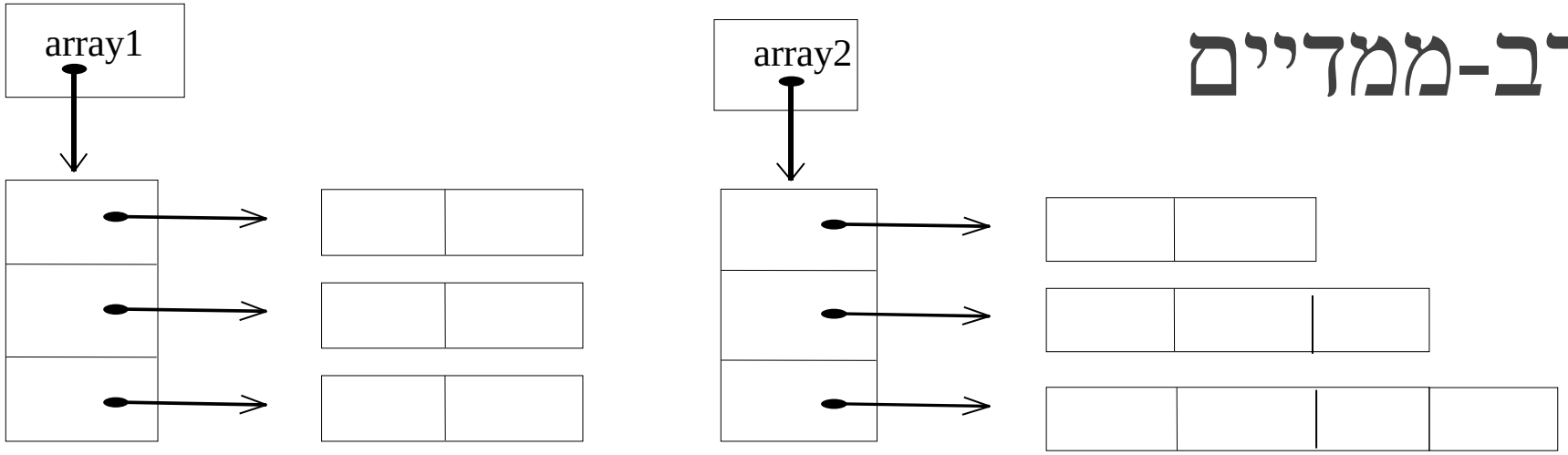
אתחול בזמן ההגדרה

```
int[][] array1 = {
                  { 1,  1},
                  {-1,  1},
                  {-1, -1},
        };
```

אפשר גם

```
int[][] array2 ={ {1,3},
                  {5,6,7},
                  {4}
                };
```

# מערכים רב-ממדיים




array1.length → 3

array1[0].length → 2

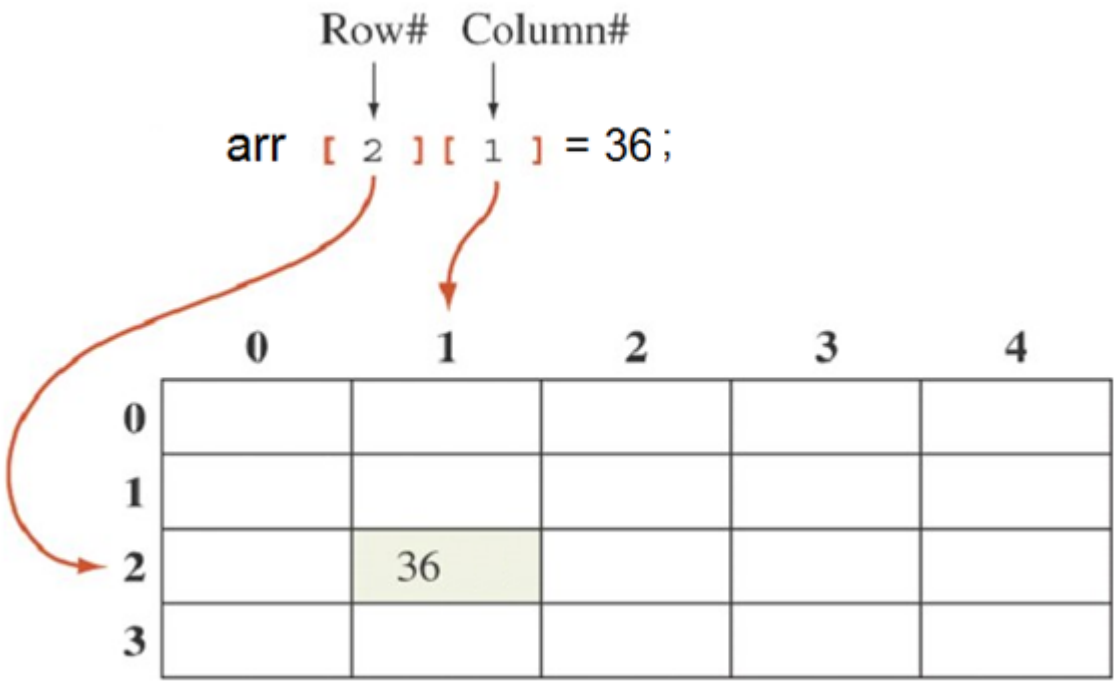
array2.length → 3

array2[2].length → 4

array2[1].length → 3

# מערכים רב-ממדיים

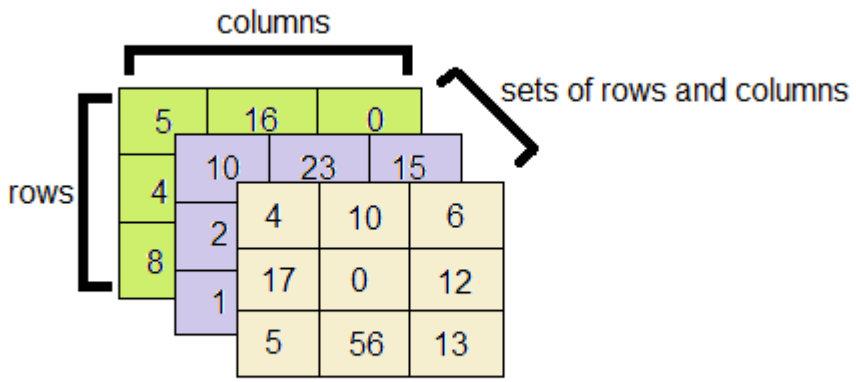
גישה לתא במערך הדו-ממדי



# מערכים רב-ממדיים

מערכים רב-ממדיים

```
double[][][] array3 = new double[3][3][3];
```



# מערכים - סיכום

למערכים יש יתרונות וחסרונות

- גישה לאיבר במקום i מתבצעת ב-O(1)
- מאחר ואחסון תאי המערך הינו רציף בזיכרון עלולה להיווצר בעיה כאשר רוצים לאחסן הרבה נתונים
- הקצאת המקום מראש עשויה להיות בעייתית כאשר לא ידוע מראש כמה תאים נצטרך
  - הקצאה של מערך גדול עלולה להוביל לניצול לא יעיל של הזיכרון
  - הקצאה של מערך קטן עלולה לא להספיק
- במערך ממוין פעולות של עדכון הוספה ומחיקה עלולים להתבצע ב-O(n) בגלל הצורך "להזיז" ערכים כדי לפנות מקום לערכים חדשים ולצמצם תאים שהערכים בהם נמחקו

# מערכים דינמיים

בשפות תכנות רבות ישנם מערכים המתיימרים להיות דינמיים – כלומר שגודלם אינו קבוע וניתן לשינוי.

```
// create an array list
ArrayList al = new ArrayList();

// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
...
```

האם המערך אכן דינמי?

# מערכים דינמיים

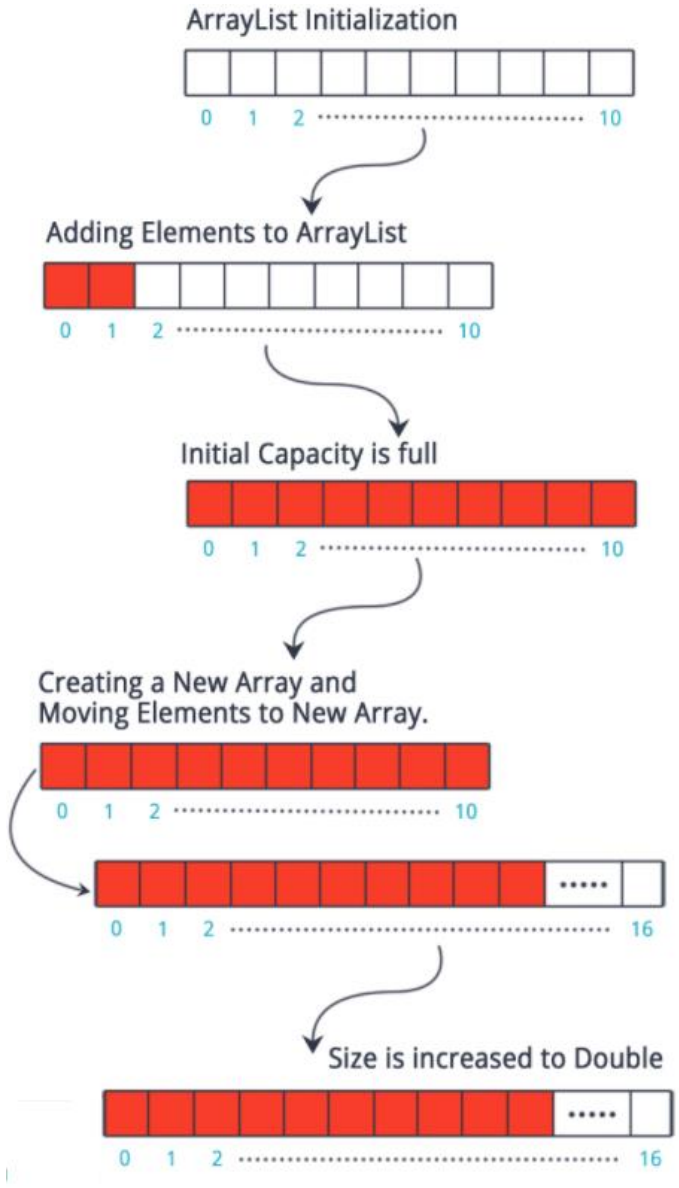
המערך נוצר בגודל התחלתי קטן

כל עוד יש מקום המערך מתמלא

ברגע שנגמר המקום ויש דרישה לאחסן נתון נוסף – יוצרים מערך חדש גדול יותר, מעתיקים אליו את כל הנתונים מהמערך הקודם ויש מקום גם לנתון החדש

אחוז הגידול הוא לרוב בין 50%-100%

- מדוע נדרשת רציפות הזיכרון?
- האם עדיין גישה לתא זיכרון לצורך אחסון היא O(1)?



# מערכים דינמיים

רציפות זיכרון נדרשת לצורך ביצוע פעולת get() ב-O(1) – על בסיס כתובת ההתחלה + ההיסט

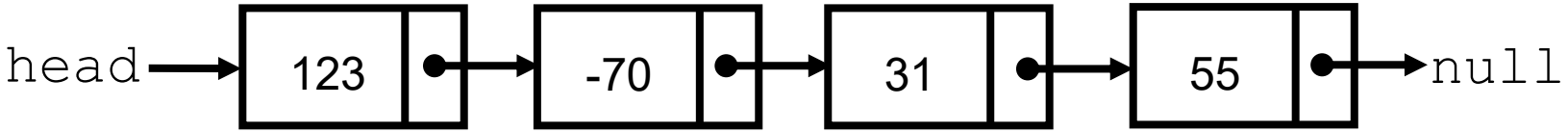
base address + index X size

בעוד שפעולת add() לסוף המערך מתבצעת במערך "רגיל" ב-O(1) הרי שבמערך דינמי יכול להיות שנגמר המקום וצריך "להגדיל" את המערך. לכן זמן החישוב יתבצע באופן הבא:

חישוב זה נקרא ניתוח לשיעורין (amortized). לרוב הפעולה "קלה" ויעילותה רבה אך לעתים רחוקות היא עלולה להיות "יקרה" ויעילותה פחותה, כמו במקרה זה

(O(1) add + O(1) add + ... + O(n) array copy) / n operations = O(1)

# רשימה מקושרת




מבנה נתונים אחר בו משתמשים לאחסון רשימת ערכים הוא רשימה מקושרת (Linked List)

רשימה מקושרת היא מבנה נתונים המורכב מאוסף של חוליות (links) המשורשרות ביניהן באמצעות מצביעים

כל חוליה מכילה, בדרך כלל, שדה של מידע (או אובייקט, או גם וגם), ושדה נוסף המכיל מצביע לחוליה הבאה

שדה המצביע של החוליה האחרונה ברשימה המקושרת מכיל את הערך null, המציין את סוף הרשימה

# רשימה מקושרת

מבנה החוליה מורכב מהנתון data (בדוגמה זו נשמור ערך מסוג מספר שלם), ומצביע המחזיק את כתובתה של החוליה הבאה ברשימה (next)

הרשימה דורשת להחזיק מצביע לחוליה הראשונה בלבד (head)

- לשאר החוליות מגיעים דרך סיור ברשימה החל מהחוליה הראשונה

```
class Node
{
    int data;
    Node next;
}

class LinkedList
{
    Node head;
}
```

# רשימה מקושרת

```
public class Node {
      int data;
      Node next;

    public Node(int data) {
       this.data = data;
       this.next = null;
    }
}
```

```
public class LinkedList {
    Node head;

    public LinkedList() {
        head = null;
    }

    public boolean isEmpty(){
        return head==null;
    }
}
```

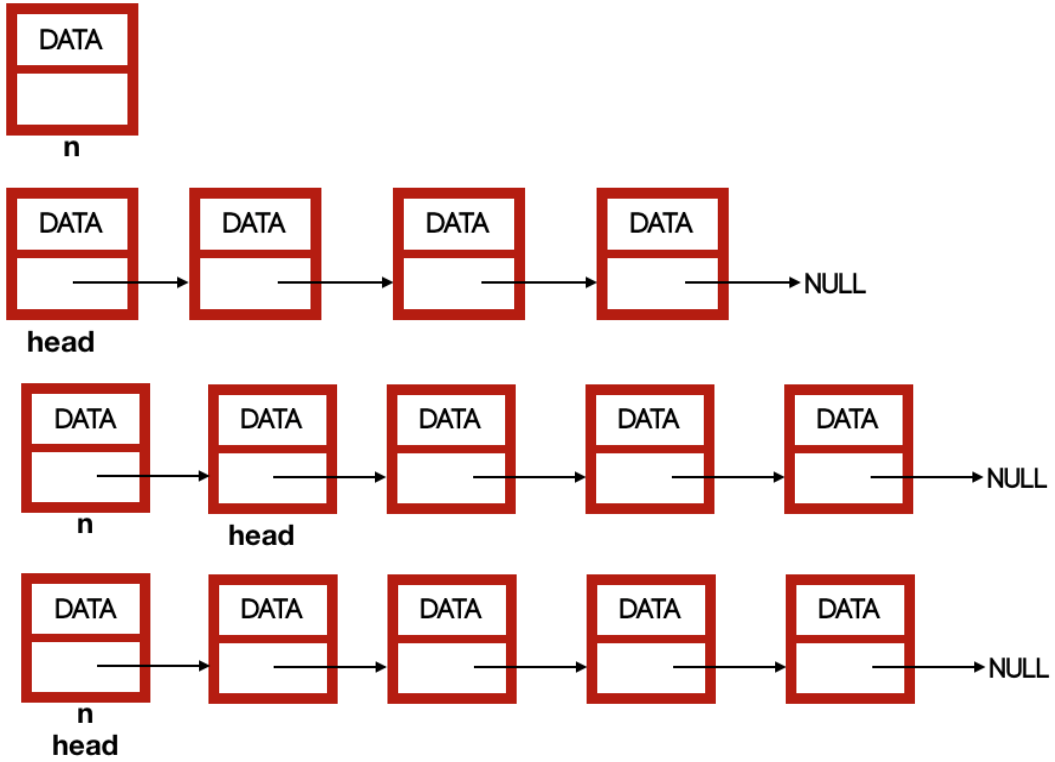
# רשימה מקושרת

פעולות ברשימה מקושרת

- הכנסת ערך data
  - בראש הרשימה (לא ממוין) אם אין חשיבות לסדר ההכנסה
  - בסוף הרשימה אם יש חשיבות לסדר ההכנסה
  - במיקום i (אם i חוקי)
  - אם רוצים שהרשימה תהיה ממוינת צריך קודם לחפש את המקום אליו יש להוסיף את החוליה החדשה
- חיפוש ערך
  - מתחילים מהראש וממשיכים עד סוף הרשימה או עד מציאת הערך המבוקש data
- מחיקת ערך data
  - מחפשים את data, ומוחקים אותו ע"י שרשור הקודם לו לבא אחריו

# הכנסת אלמנט חדש לראש הרשימה

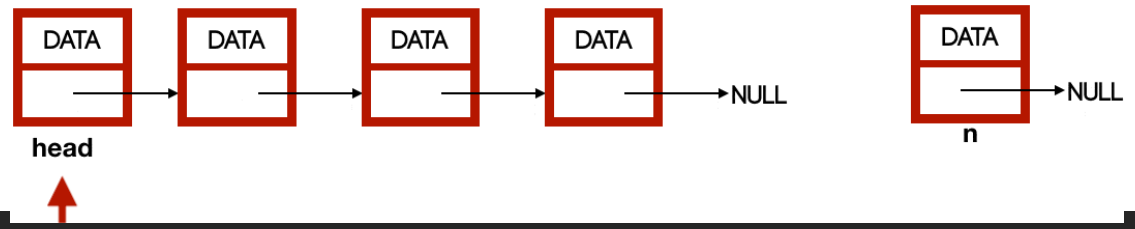
```
public void insertFirst(int data) {
      Node newNode = new Node(data);
       newNode.next = head;
       head = newNode ;
}
```



# הוספת אלמנט חדש לסוף הרשימה

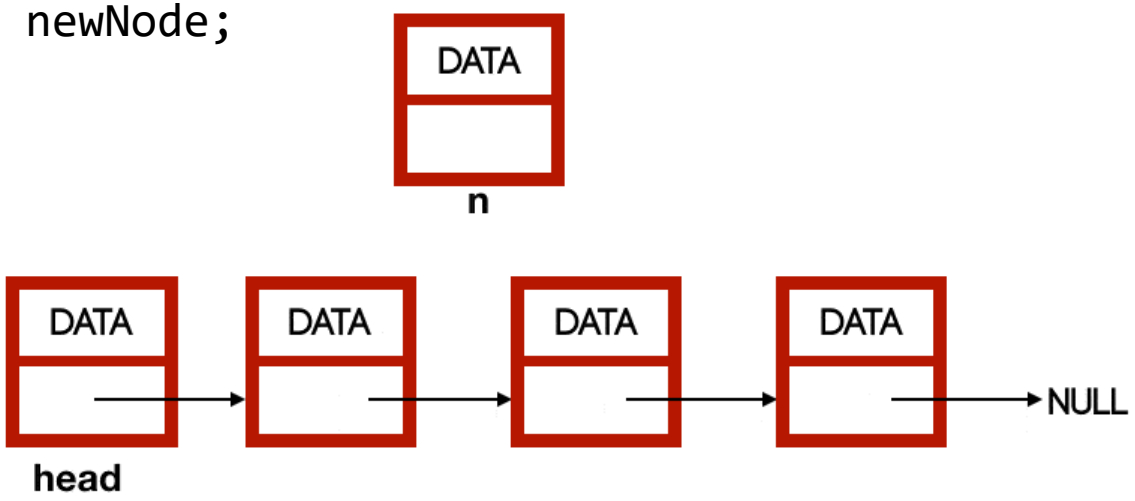
```
public void insertLast(int data) {
        Node newNode = new Node(data);
        if (head == null)
                head = newNode;
        else {
                Node tmp = head;
                while (tmp.next != null )
                        tmp = tmp.next;
                tmp.next = newNode;
        }
}
```

- אם פעולת ההוספה שכיחה, כדאי להוסיף לרשימה תכונה נוספת tail אשר תצביע תמיד על החוליה האחרונה ברשימה
- כך ייחסך חיפוש זנב הרשימה בכל פעם



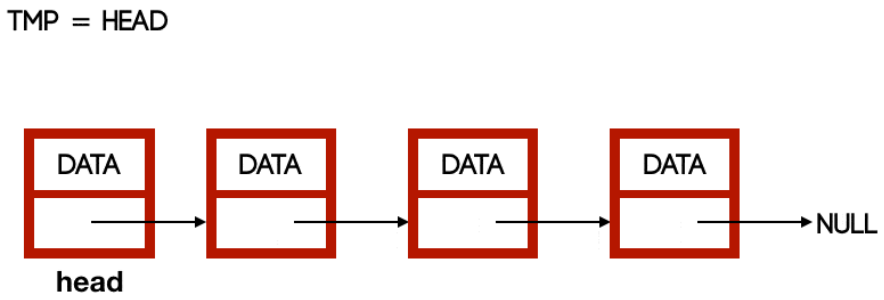
# הוספת אלמנט חדש לרשימה אחרי חוליה

```
public void insertAfter(int data, Node after) {
        Node newNode = new Node (data);
        newNode.next = after.next;
        after.next = newNode;
}
```



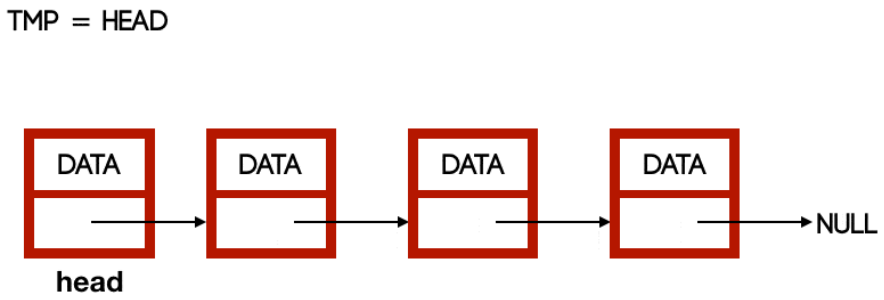
# איתור מיקום אלמנט בעל ערך data

```
public int indexOf(int data) {
        int counter = 0;
        Node tmp = head;
        while (tmp != null && tmp.data != data) {
                counter = counter + 1;
                 tmp = tmp.next;
        }
        if (tmp == null) // reached end
                return -1; // -1 means not found
        else // data found
                return counter;
}
```



# איתור חוליה בעלת ערך data

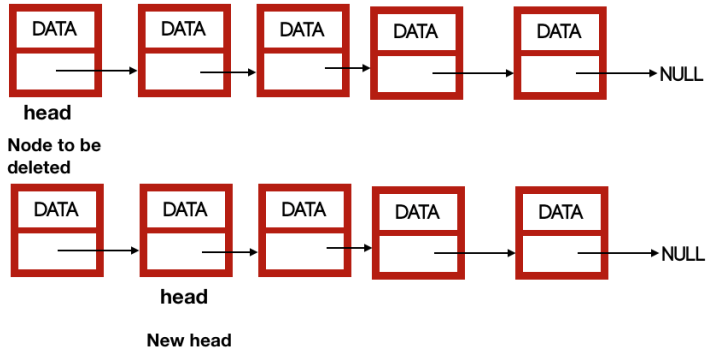
```
public Node find(int data) {
        Node tmp = head;
        while (tmp != null) {
                if (tmp.data == data)
                        return tmp;
                 tmp = tmp.next;
        }
        return null; // data not found
}
```

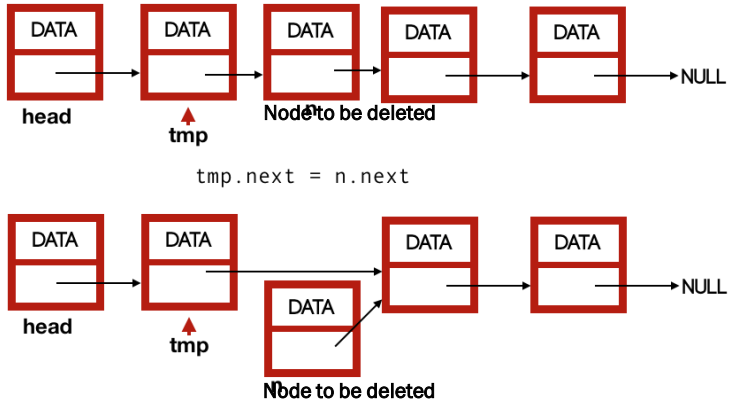


# רשימה מקושרת

מחיקת איבר בעל ערך נתון data

```
public void delete (int data) {
        Node nodeToDelete = find (data);
        if (nodeToDelete == null)
                return;
        if (head == nodeToDelete)
                head = head.next;
        else {
                Node tmp = head;
                while (tmp != null) {
                        if (tmp.next == nodeToDelete) {
                                tmp.next = nodeToDelete.next;
                                break;
                        }
                        tmp = tmp.next;
                }
        }
}
```



# רשימה מקושרת

כיצד נחשב את size()

- אפשר לרוץ על הרשימה החל מ-head ועד לסופה ולספור - O(n)
- עדיף להוסיף שדה counter לרשימה, ולעדכן את ערכו בכל הוספה/מחיקה של חוליה. המתודה size() תחזיר את ערכו – O(1)

```
public class LinkedList {
  Node head;
  int counter;

  public LinkedList() {
      head = null;
      counter = 0;
    }
```

```
  public void insert()

        ...
        counter++;
    }

  public void delete() {
        ...
        counter--;
    }
```

```
  public int size() {
     return counter;
    }
}
```