Data Structure

Arrays, Records and Pointers

Instructors:

Md Nazrul Islam Mondal

Department of Computer Science & Engineering

Rajshahi University of Engineering & Technology Rajshahi-6204

Outline

- Introduction
- Linear Array
- Representation of Linear Array in Memory
- Traversing Linear Array
- Inserting and Deleting
- Sorting: Bubble Sort
- Searching: Linear Search
- Binary Search

Introduction

Introduction

- Data structure are classified as either **linear** or **nonlinera**.
- A data structure is said to be linear if its elements **form a sequence**, or, in other words, a linear list.
  - The elements represented by means of sequential memory location (**Array**)
  - The elements represented by means of pointers or links (**linked list**)

Introduction

- The operation performs on linear structure –
  - **Traversal** – Processing each element in the list
  - **Search** – Finding the location of the element based on given value or the record with a given key
  - **Insertion** – Adding a new element to the list
  - **Deletion** – Removing an element from the list
  - **Sorting** – arranging the element in some type of order
  - **Merging** – Combining to lists into a single list

Linear Arrays

Linear Arrays

- A linear array is a list of a finite number of homogeneous data elements (i.e. data elements of the same type).
- **Length Calculation:**
    - » Length = UB-LB+1
  - – Example: A[10], LB = 0, UB = 9, Length = 9-0+1 = 10

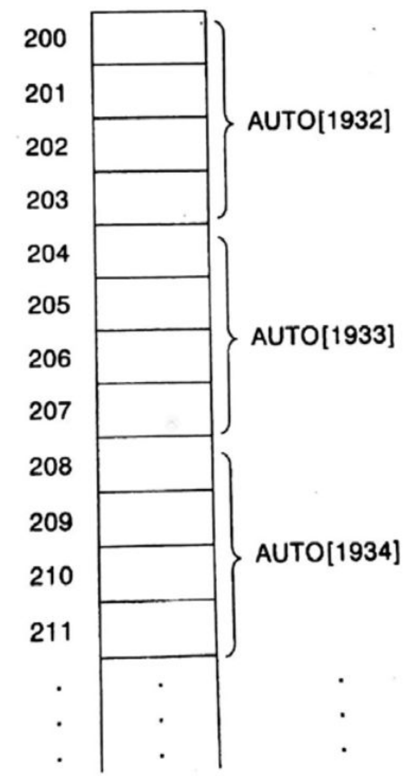
Representation of Linear Arrays in Memory

Representation of Linear Arrays in Memory

- **LOC(LA[k]**])=address of the element LA[K] of the array LA
- **Base(LA**) = Base address of LA
- **Formula**:
  - LOC(LA[K]) = Base(LA)+w(K – lower bound)
    - Where w is the number of words per memory for the array LA

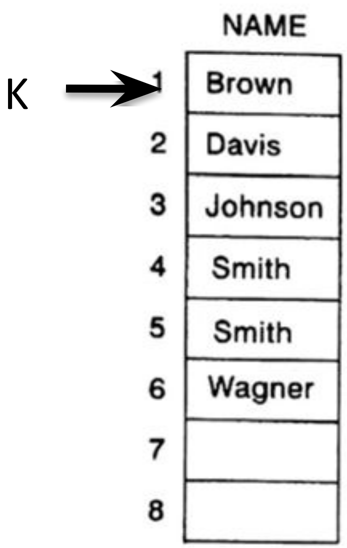
Representation of Linear Arrays in Memory

- **Given,**
  - Base(AUTO)=200
  - LOC(AUTO[1932])=200
  - LOC(AUTO[1933]) = 204
  - LOC(AUTO[1933]) = 208
  - Find the LOC(AUTO[1965]) = ?
- **Solution**:
  - K = 1965. w = 4,
  - LOC(AUTO[1965]) = Base(AUTO)+w(K-Lower Bound)
  - LOC(AUTO[1965]) = 200+4(1965-1932) = 200+4x33 = 200+132 = 332



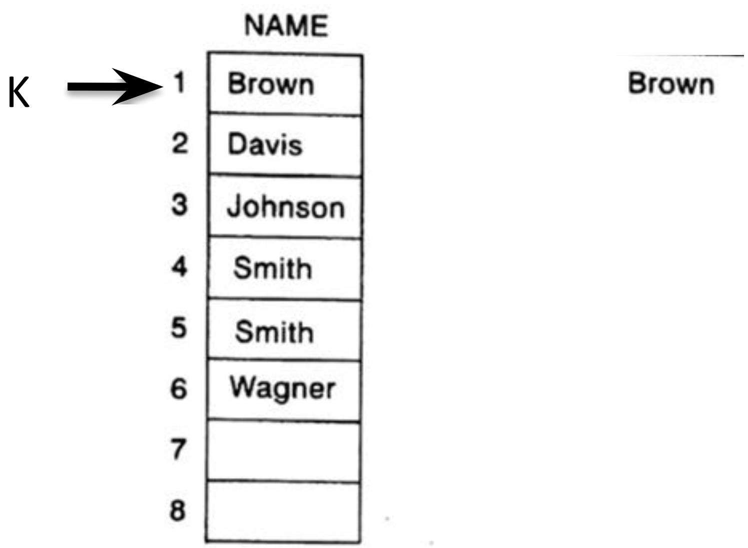
Traversing Linear Arrays

Traversing Linear Arrays



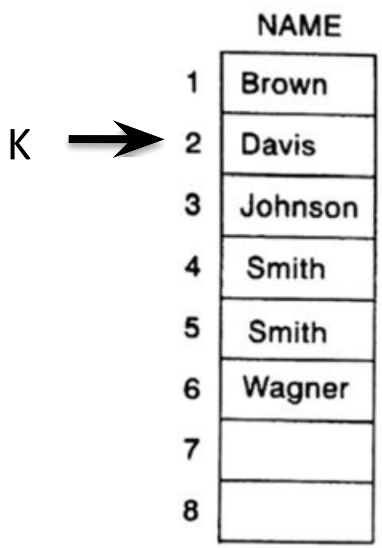
(a)

Traversing Linear Arrays



(a)

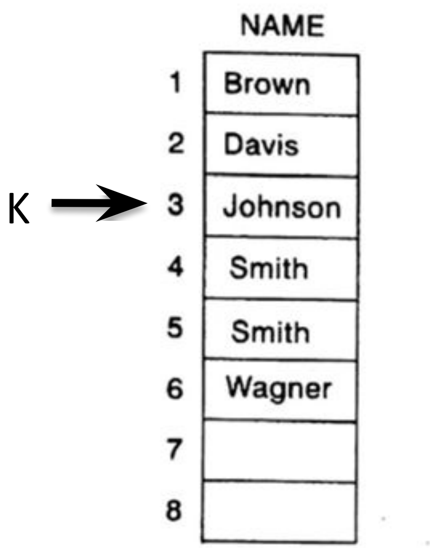
Traversing Linear Arrays



(a)

Brown Davis

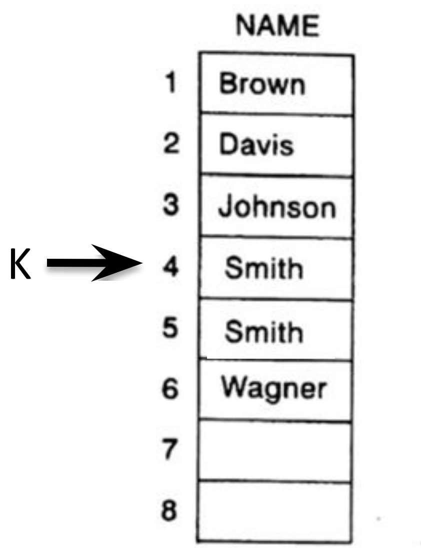
Traversing Linear Arrays



(a)

Brown Davis Johnson

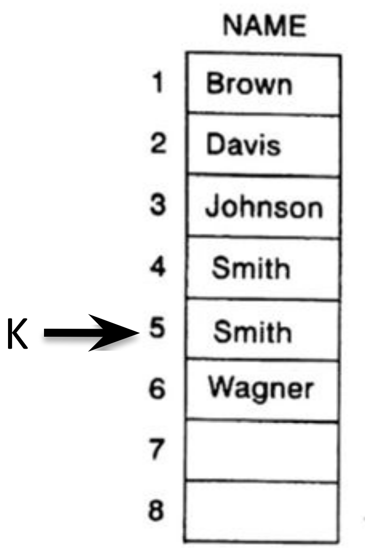
Traversing Linear Arrays



(a)

Brown Davis Johnson Smith

Traversing Linear Arrays



(a)

Brown Davis Johnson Smith Smith

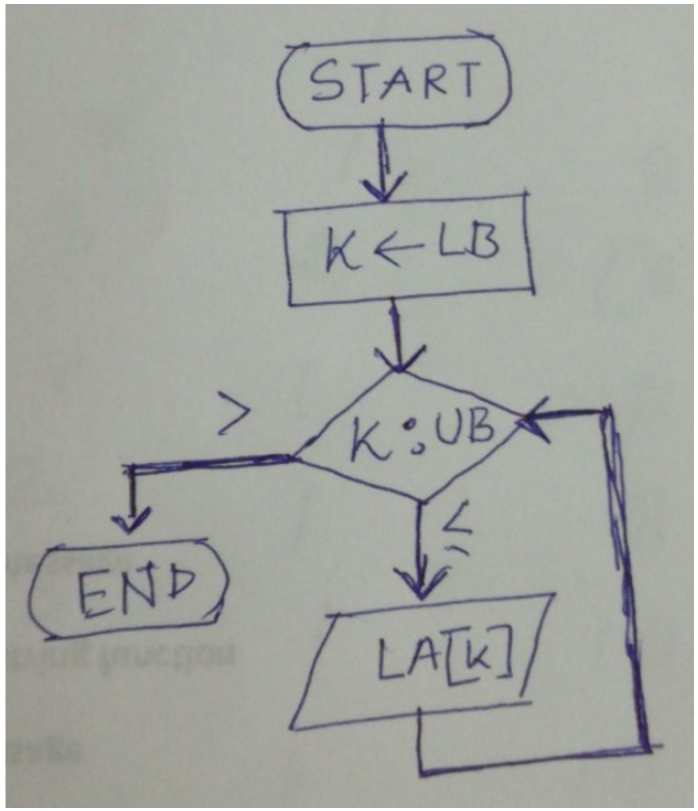
Traversing Linear Arrays

Algorithm 4.1: (Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB.
2. Repeat Steps 3 and 4 while K ≤ UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set K := K + 1.

   [End of Step 2 loop.]
5. Exit.

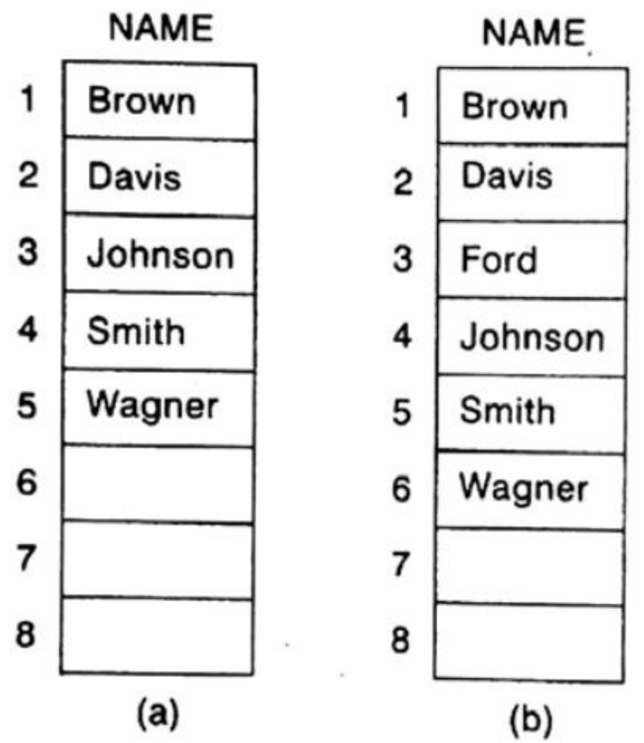
Traversing Linear Arrays



Inserting and Deleting

Inserting

Insert Ford into 3rd Position



Inserting

Insert Ford into 3rd Position

	NAME
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

Ford

Inserting

Insert Ford into 3rd Position

Ford

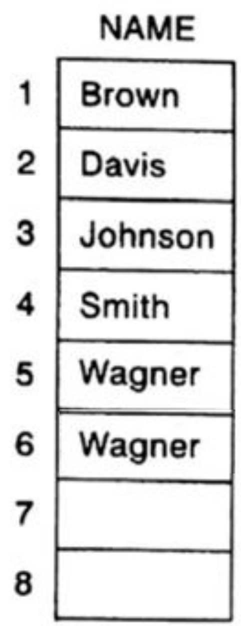
	NAME
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

Inserting

Insert Ford into 3rd Position

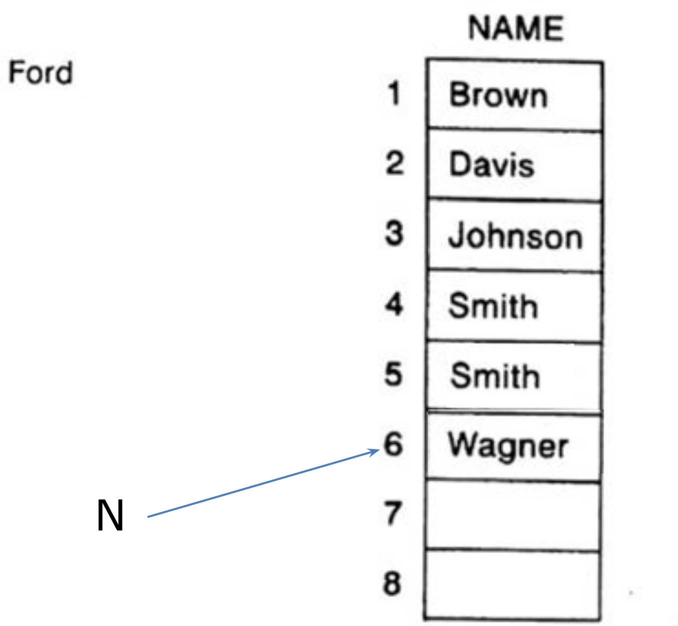
Ford



(a)

Inserting

Insert Ford into 3rd Position



(a)

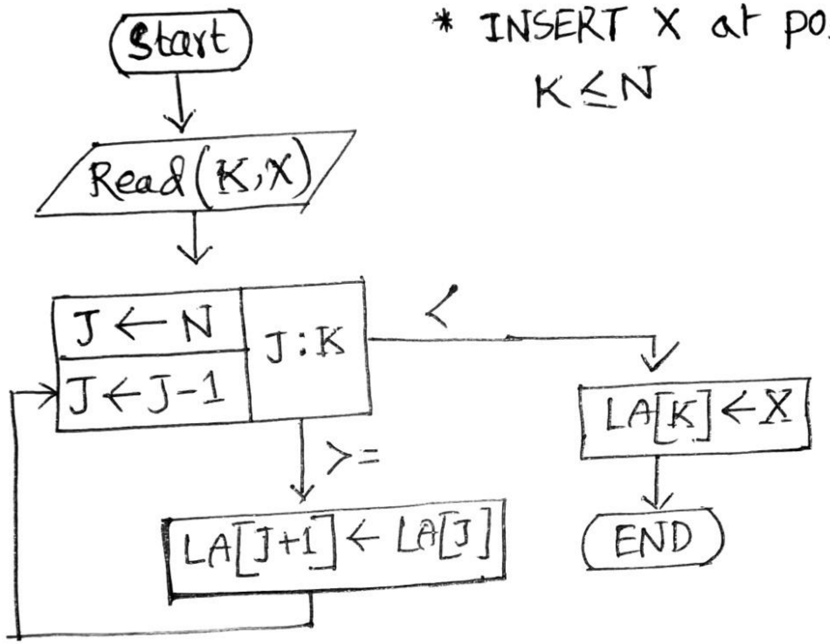
Inserting

Algorithm 4.2: (Inserting into a Linear Array) INSERT(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set $J := N$.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set $LA[J + 1] := LA[J]$.
4. [Decrease counter.] Set $J := J - 1$.

   [End of Step 2 loop.]
5. [Insert element.] Set $LA[K] := ITEM$.
6. [Reset N.] Set $N := N + 1$.
7. Exit.

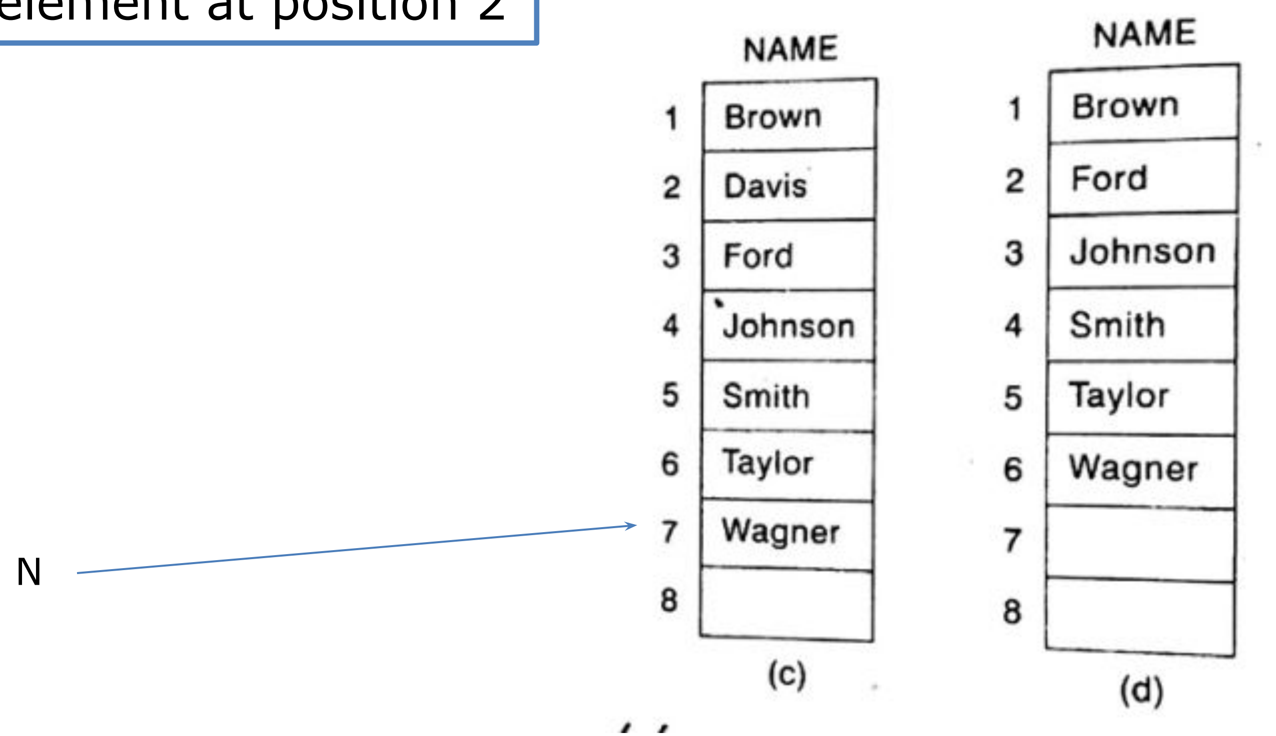
Inserting



* INSERT X at position
K ≤ N

Deleting

Delete element at position 2



Deleting

Delete element at position 2

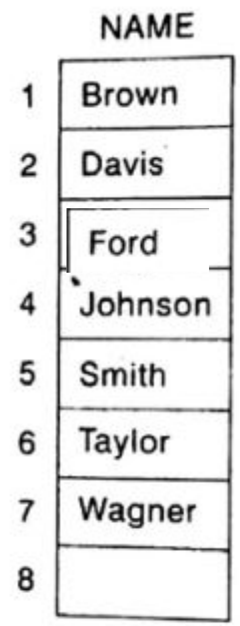
ITEM :=

	NAME
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Taylor
7	Wagner
8	

(c)

Deleting

Delete element at position 2



(c)

Deleting

Delete element at position 2

	NAME
1	Brown
2	Ford
3	Ford
4	Johnson
5	Smith
6	Taylor
7	Wagner
8	

(c)

Deleting

Delete element at position 2

(c)

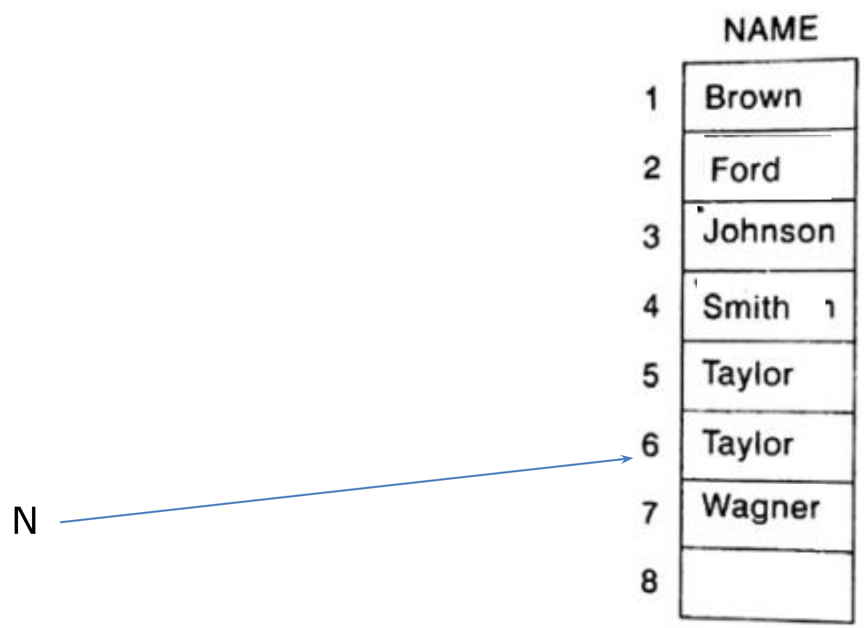
Deleting

Delete element at position 2

(c)

Deleting

Delete element at position 2



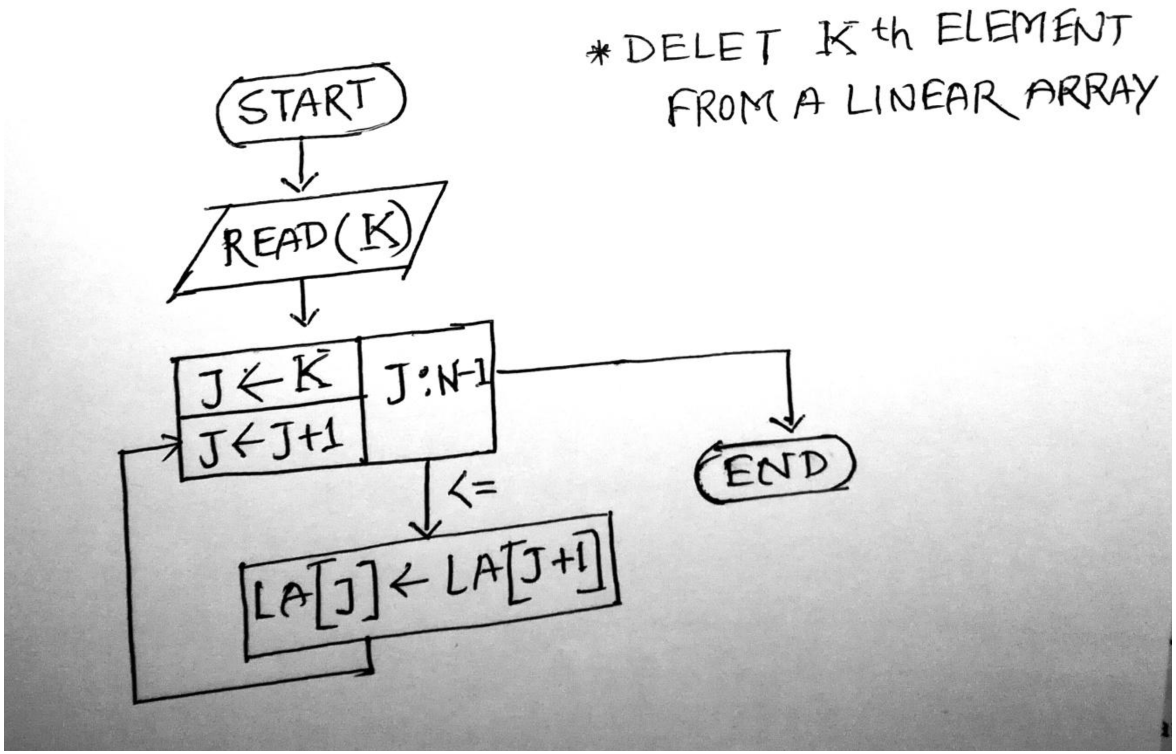
(c)

Deleting

Algorithm 4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N − 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N := N − 1.
4. Exit.

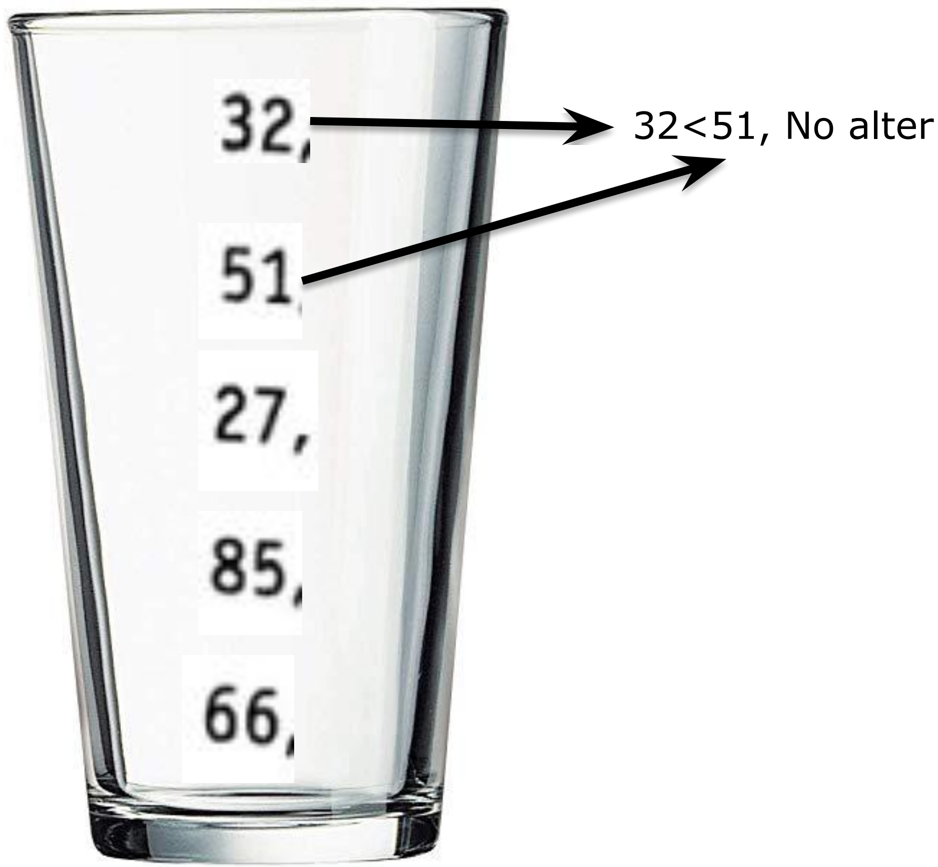
Deleting



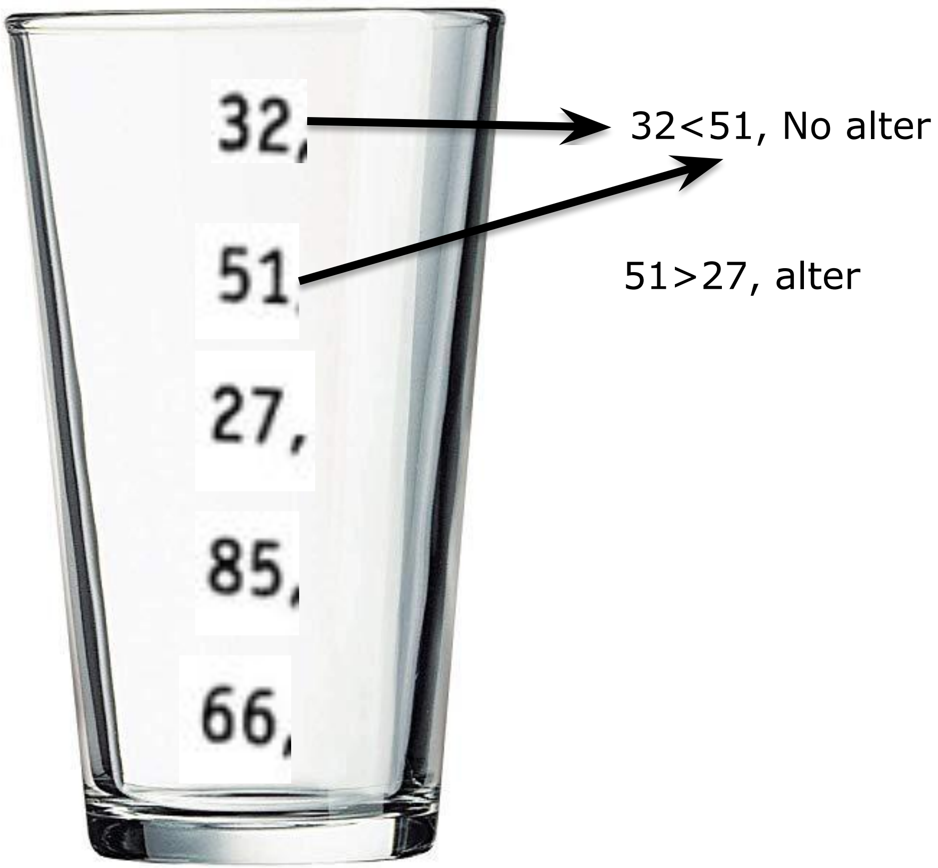
* DELET K th ELEMENT
FROM A LINEAR ARRAY
START
READ(K)
J←K
J←J+1
J:N-1
<=
END
LA[J]←LA[J+1]

Sorting: Bubble Sort

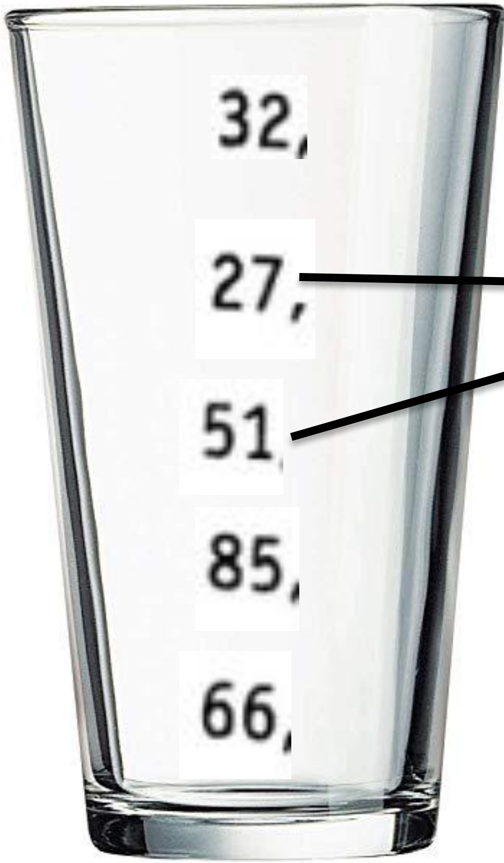
Sorting: Bubble Sort



Sorting: Bubble Sort



Sorting: Bubble Sort

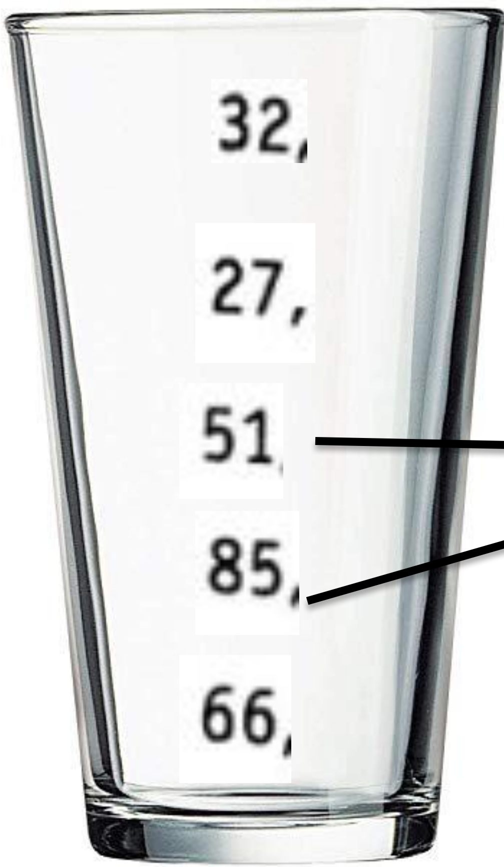


32<51, No alter

51>27, alter

51<85, No alter

Sorting: Bubble Sort



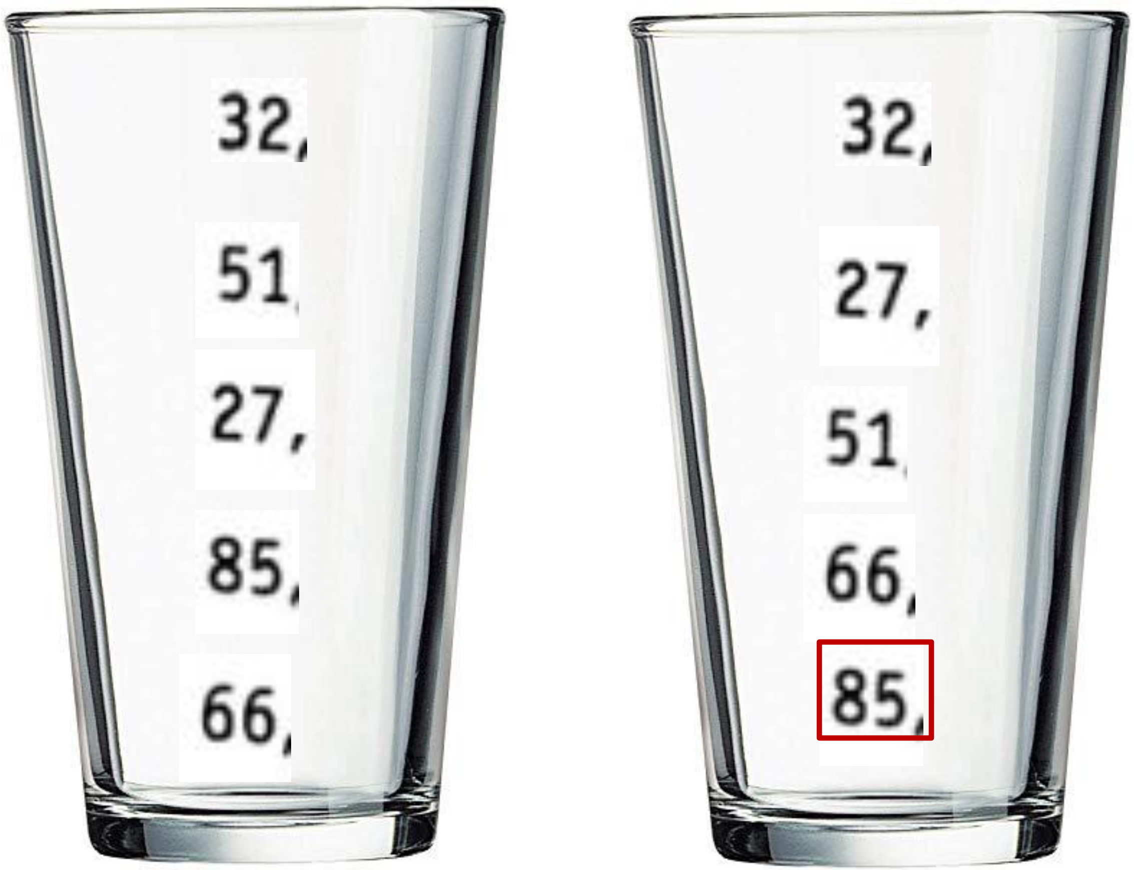
32<51, No alter

51>27, alter

51<85, No alter

85>66, alter

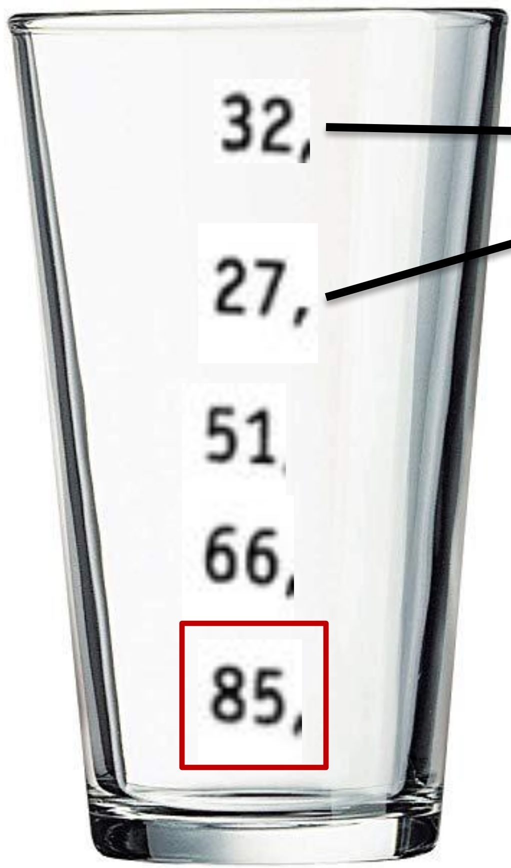
Sorting: Bubble Sort



1^{st} Pass

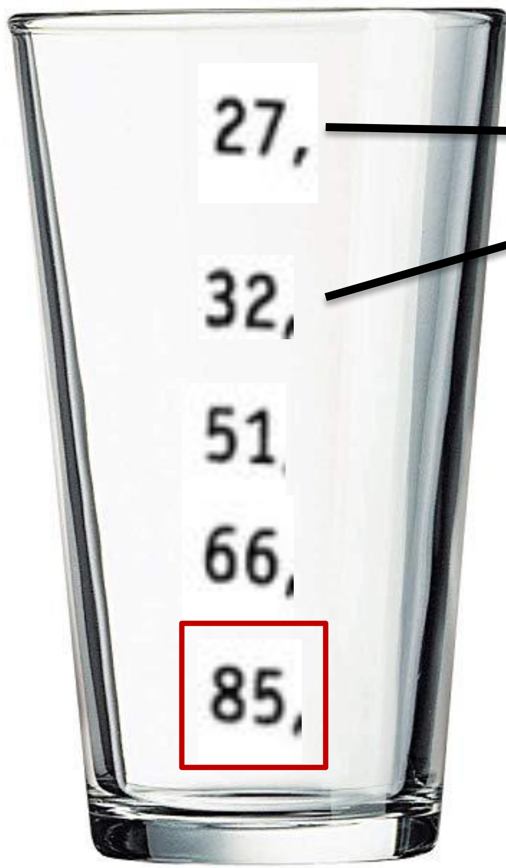
Small values move up direction like bubble

Sorting: Bubble Sort



32>27, alter

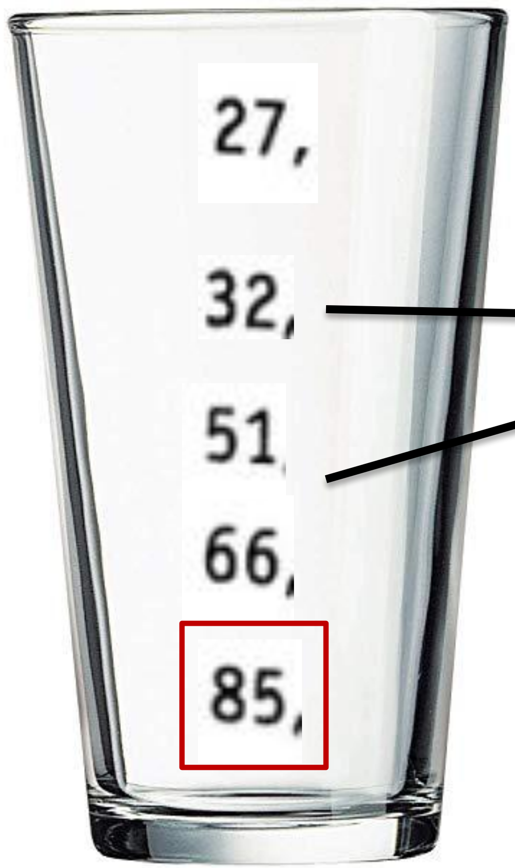
Sorting: Bubble Sort



32>27, alter

32<51, no alter

Sorting: Bubble Sort

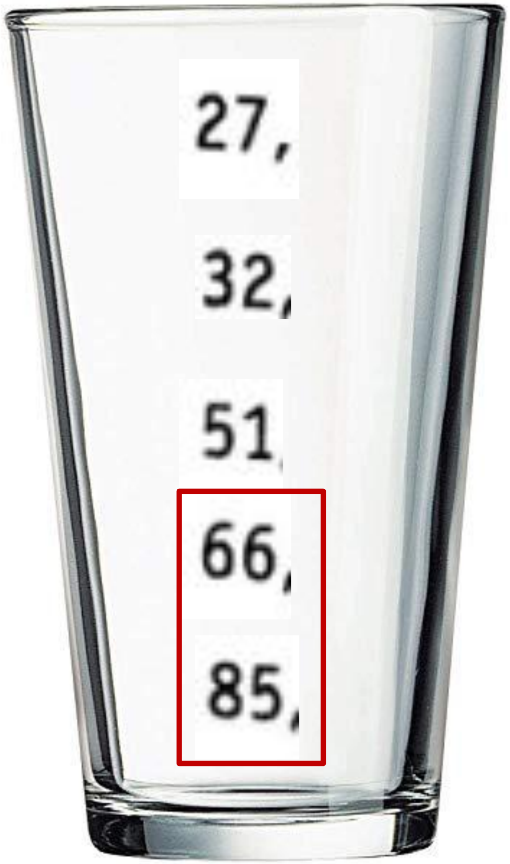


32>27, alter

32<51, no alter

51<66, no alter

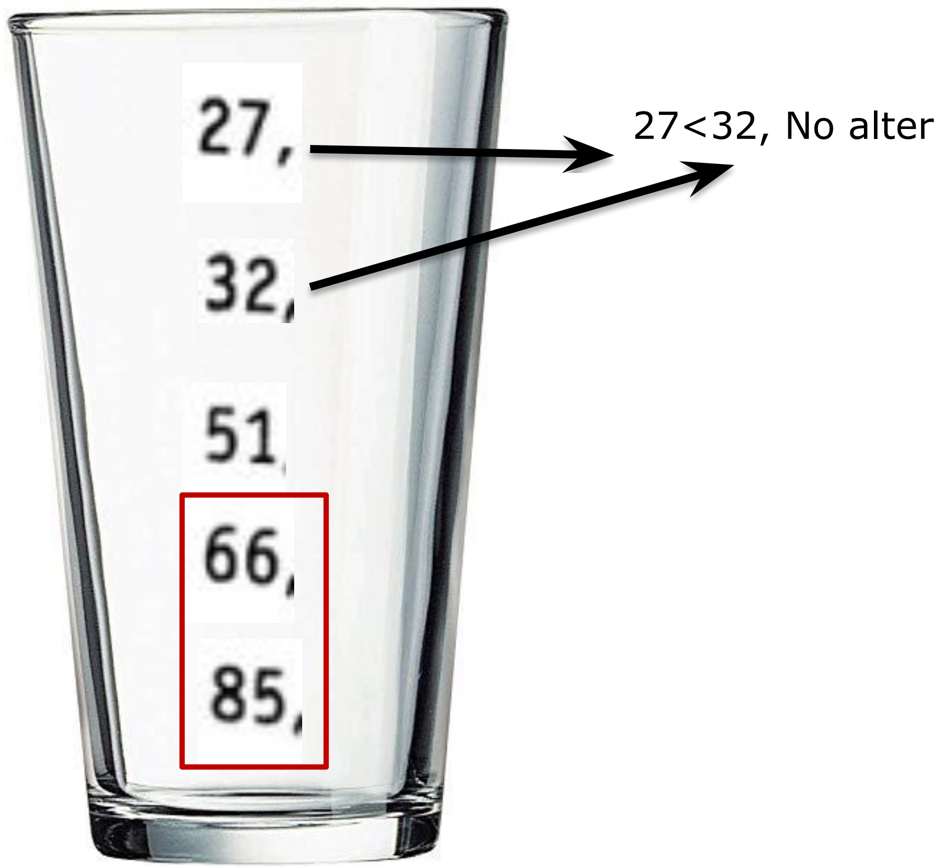
Sorting: Bubble Sort



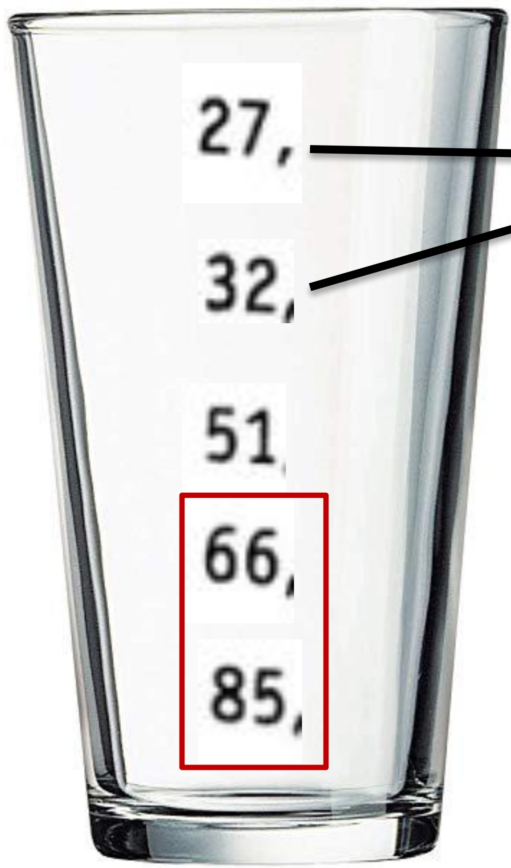
2nd Pass

Small values move up direction like bubble

Sorting: Bubble Sort



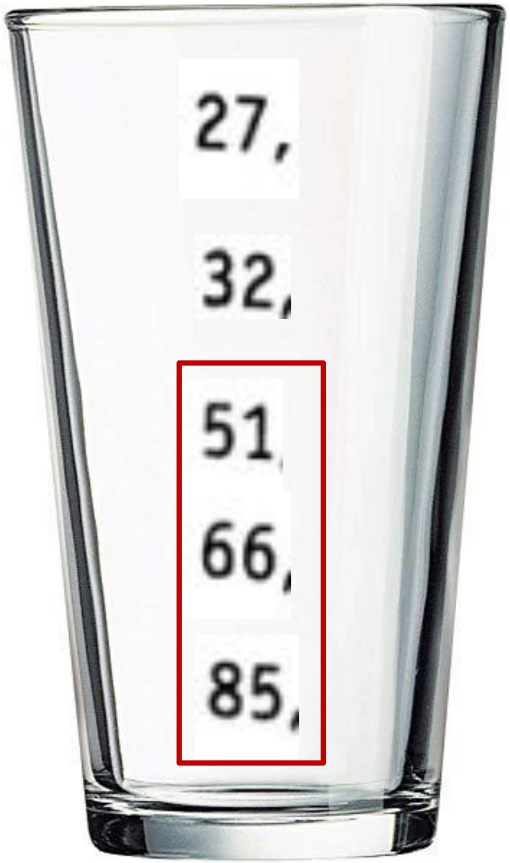
Sorting: Bubble Sort



27<32, No alter

32<51, No alter

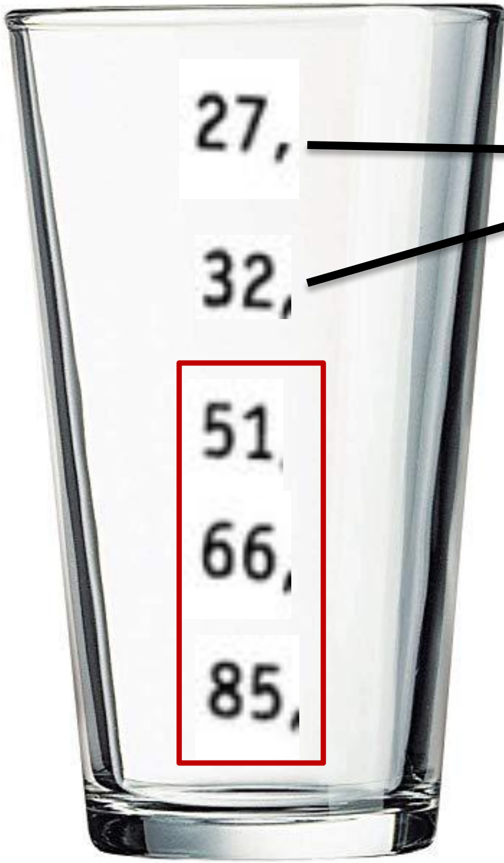
Sorting: Bubble Sort



3rd Pass

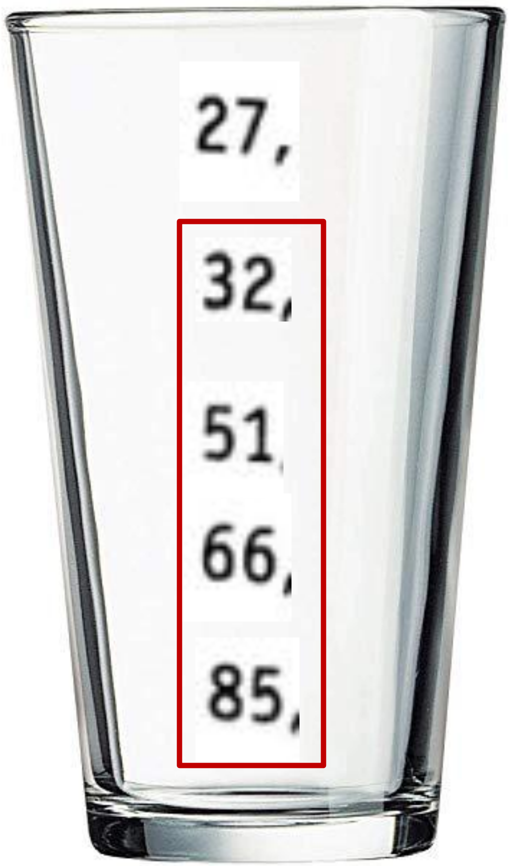
Small values move up direction like bubble

Sorting: Bubble Sort



27<32, No alter

Sorting: Bubble Sort

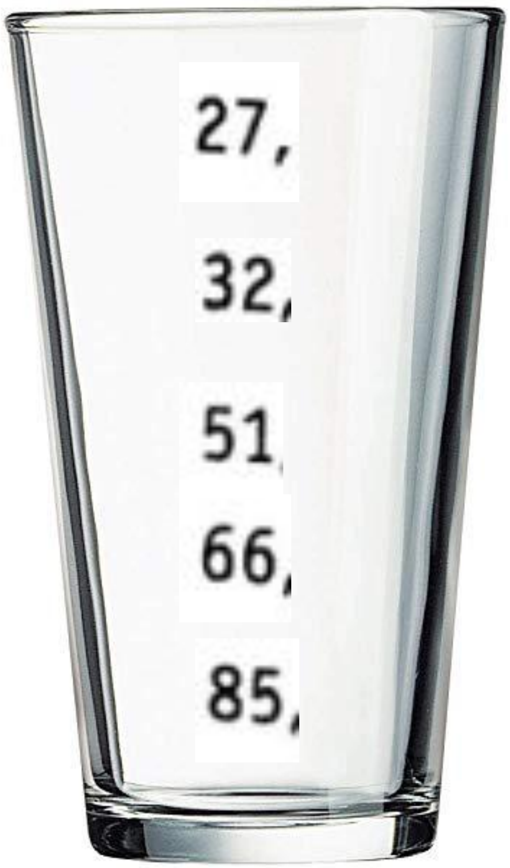


Break

4th Pass

Small values move up direction like bubble

Sorting: Bubble Sort

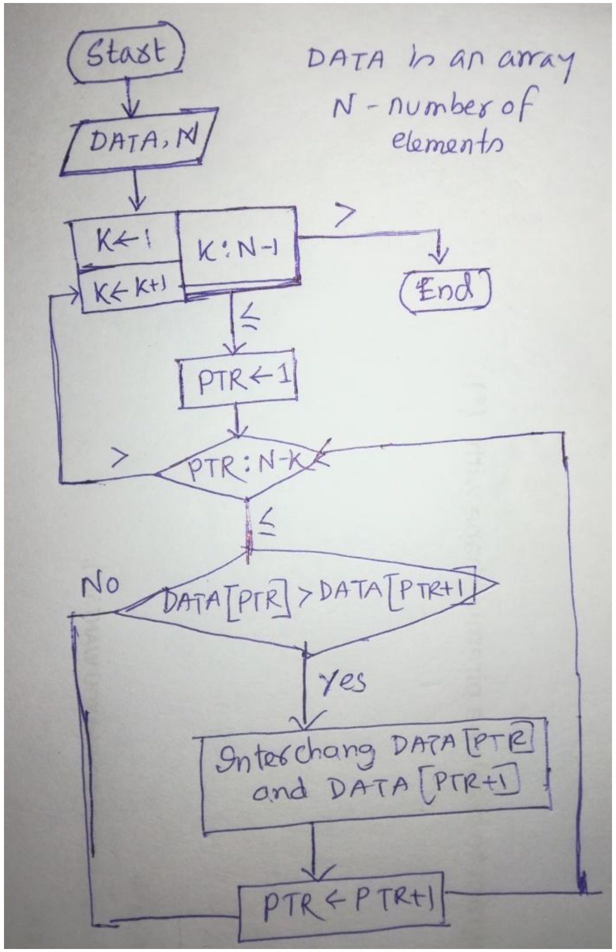


Sorting: Bubble Sort

Algorithm 4.4: (Bubble Sort) BUBBLE(DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N – 1.
2. Set PTR := 1. [Initializes pass pointer PTR.]
3. Repeat while PTR ≤ N – K: [Executes pass.]
(a) If DATA[PTR] > DATA[PTR + 1], then:
Interchange DATA[PTR] and DATA[PTR + 1].
[End of If structure.]
(b) Set PTR := PTR + 1.
[End of inner loop.]
[End of Step 1 outer loop.]
4. Exit.

Sorting: Bubble Sort



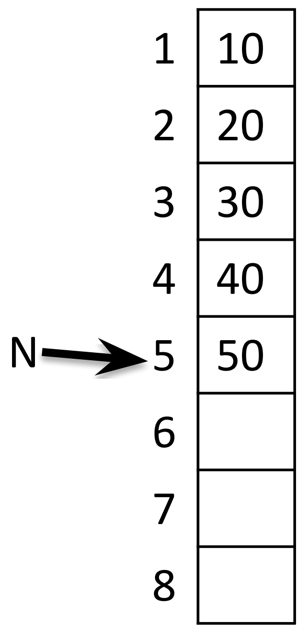
Sorting: Bubble Sort (Complexity)

- **Outer Loop** continues n-1 times
- **Inner Loop** depends on outer loop.
  - 1^{st} time, Inner Loop continue n-1 times
  - 2^{nd} time, Inner Loop continue n-2 times
  - .
  - .
  - n-1 th time, Inner loop continue 1 time
- Then we can write,

$$
\begin{aligned}
f(n) &= (n-1)+(n-2)+...+1 \\
&= 1+2+\ldots+(n-1)+n-n \\
&= \{n(n+1)/2\}-n \\
&= n(n-1)/2 \\
&= O(n^2)
\end{aligned}
$$

Searching

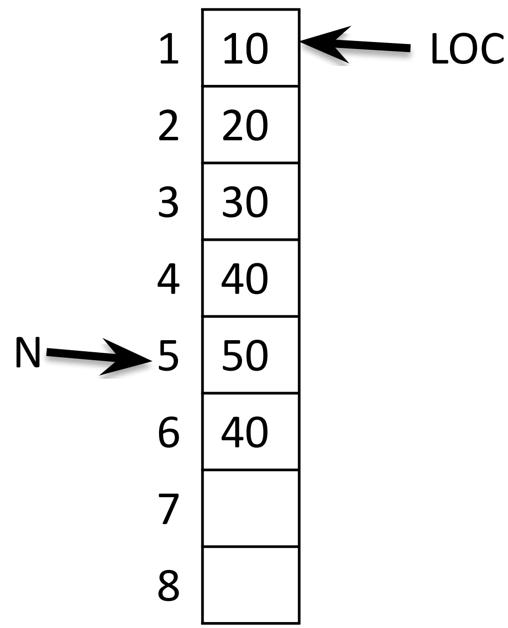
Searching: Linear Search



Search ITEM = **40**

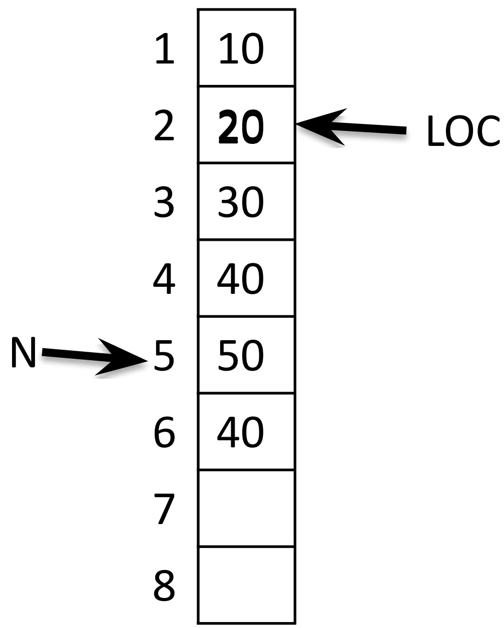
Memory Cost increased but Time complexity reduced

Searching: Linear Search



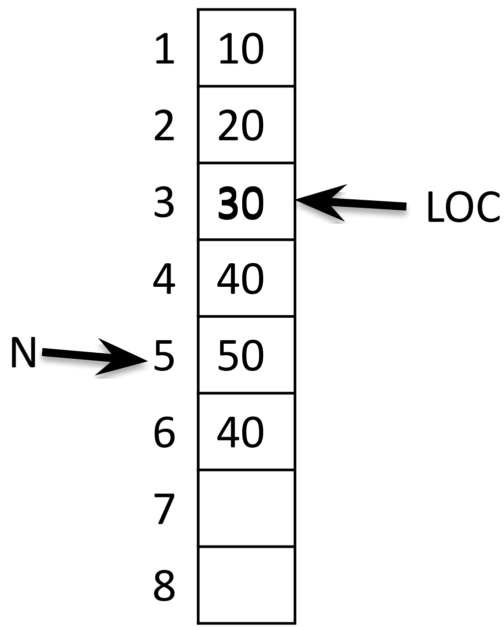
Search ITEM = **40**

Searching: Linear Search



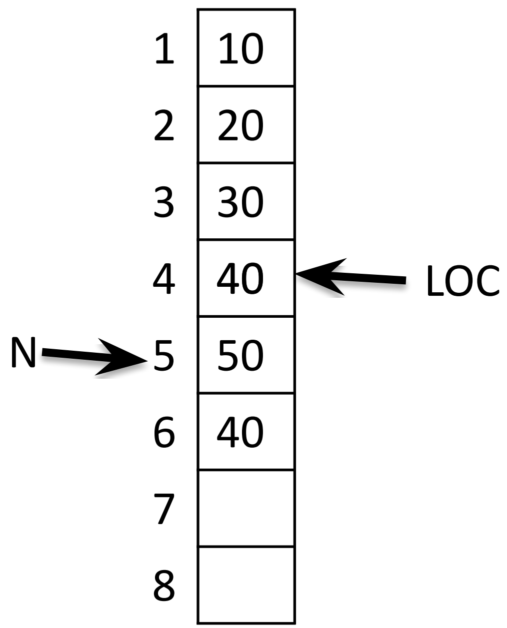
Search ITEM = **40**

Searching: Linear Search



Search ITEM = **40**

Searching: Linear Search



Search ITEM = **40**

=

Final Location, LOC = 4

Searching: Linear Search

Algorithm 4.5: (Linear Search) LINEAR(DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
     Set LOC := LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

Searching: Linear Search (Complexity)

Worst Case Complexity: $f(n) = n+1 = O(n)$

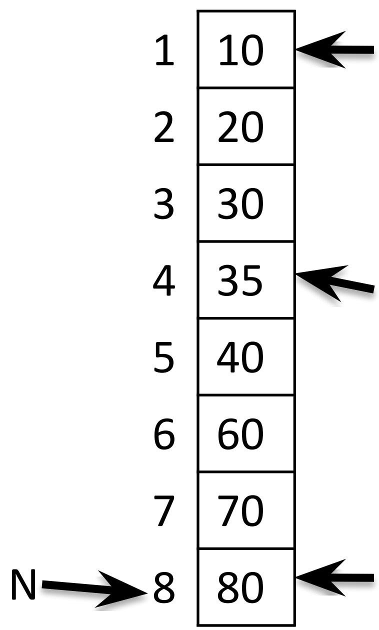
Average Case Complexity: $f(n) = 1.(1/n)+2.(1/n)+ \ldots +n(1/n) + (n+1).\mathbf{0}$

Last Element is inserted which is not in Data list, So probability = 0.

$$\mathbf{f(n) = (1+2+3+ \ldots +n)/n = (n+1)/2}$$

Searching: Binary Search

Consider, DATA is sorted



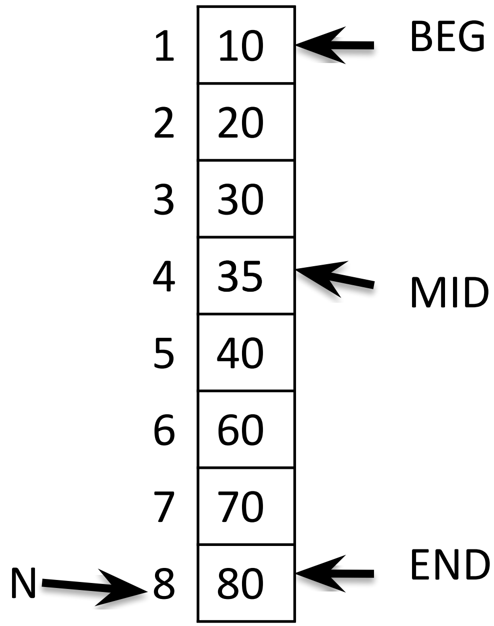
Search ITEM = **40**

LB = 1 and UB = N = 8

BEG = LB and **END** = UB
MID = INT((BEG+END)/2) = 4

Searching: Binary Search

Consider, DATA is sorted



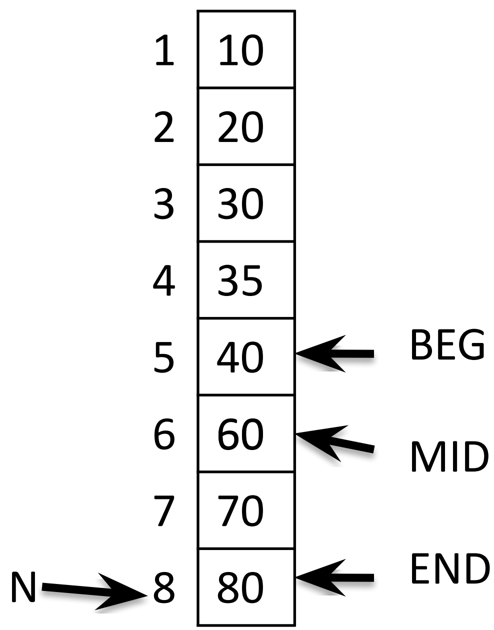
Search ITEM = **40**

<

BEG = MID +1=5
MID = 6

Searching: Binary Search

Consider, DATA is sorted



Search ITEM = **40**

>

END = MID -1=5
MID = 5

Searching: Binary Search

Consider, DATA is sorted

Search ITEM = **40**

	1	10	
	2	20	
	3	30	
	4	35	
	5	**40**	← BEG END MID
	6	60	
	7	70	
N →	8	80	

=

LOC := MID

Searching: Binary Search

Algorithm 4.6: (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3. If ITEM < DATA[MID], then:
       Set END := MID – 1.
   Else:
       Set BEG := MID + 1.
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
       Set LOC := MID.
   Else:
       Set LOC := NULL.
   [End of If structure.]
6. Exit.

Searching: Binary Search (Complexity)

Let, 1^{st} step, data size will be (n/2)
2^{nd} step, data size will be $(n/2)/2 = (n/2^2)$
3^{rd} step, data size will be $= (n/2^3)$

Let After k steps, data size will be 1

That is, $(n/2^k) = 1$
$n = 2^k$
$\log_2 n = \log_2 2^k$
$K = \log_2 n$

The complexity is k i.e. **$\log_2 n$** (worst Case and Average Case)

Searching: Binary Search (Limitation)

- The algorithm requires two conditions –
    1. The list must be sorted
    2. One must have access to the middle element in any sublist.
- This means that one must essentially use **a sorted array** to hold the data
- But keeping data in a sorted array is very expensive when there are many insertions and deletions **(array limitation)**.
- In such situation, one may use a different data structure such as a **linked** list or a **binary search tree**, to store the data.

END