Lecture#3

Data Structures

Dr. Abu Nowshed Chy

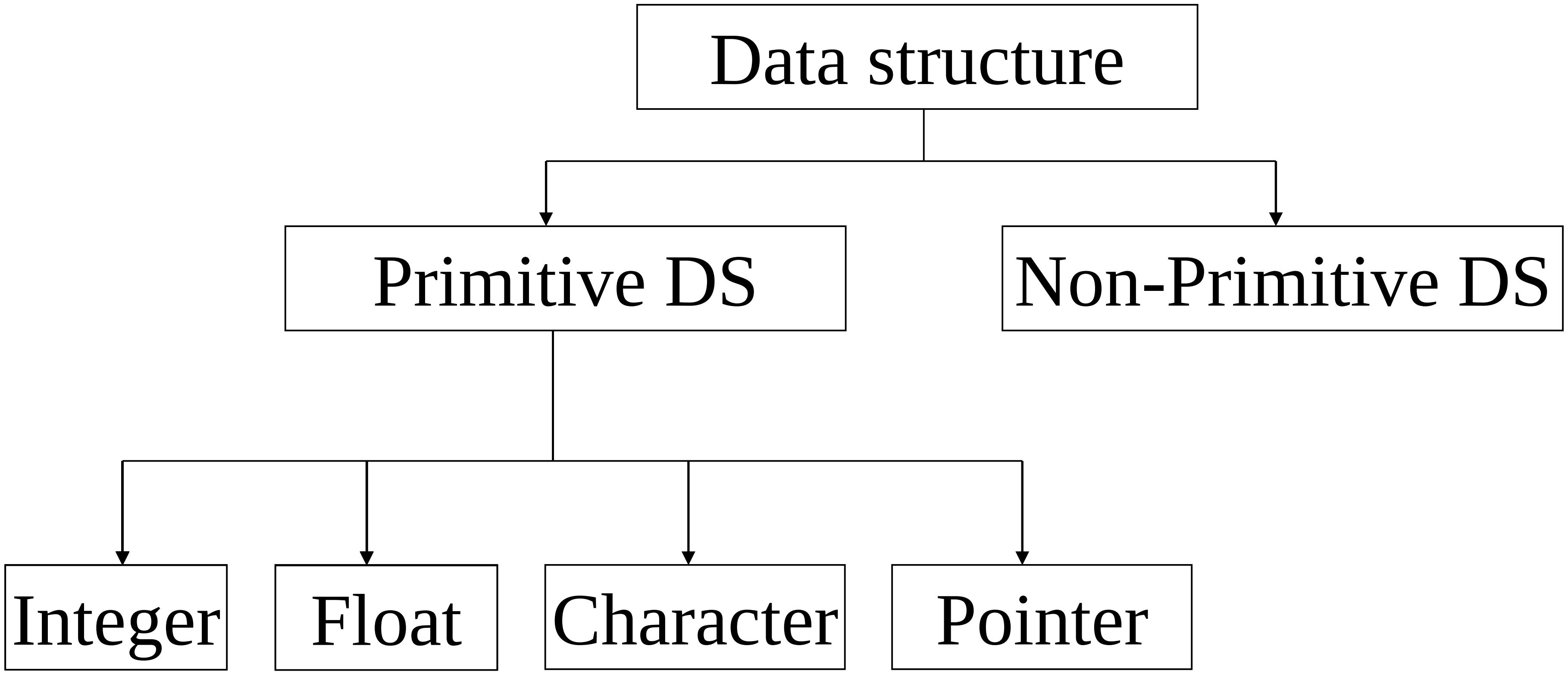
Department of Computer Science and Engineering

University of Chittagong

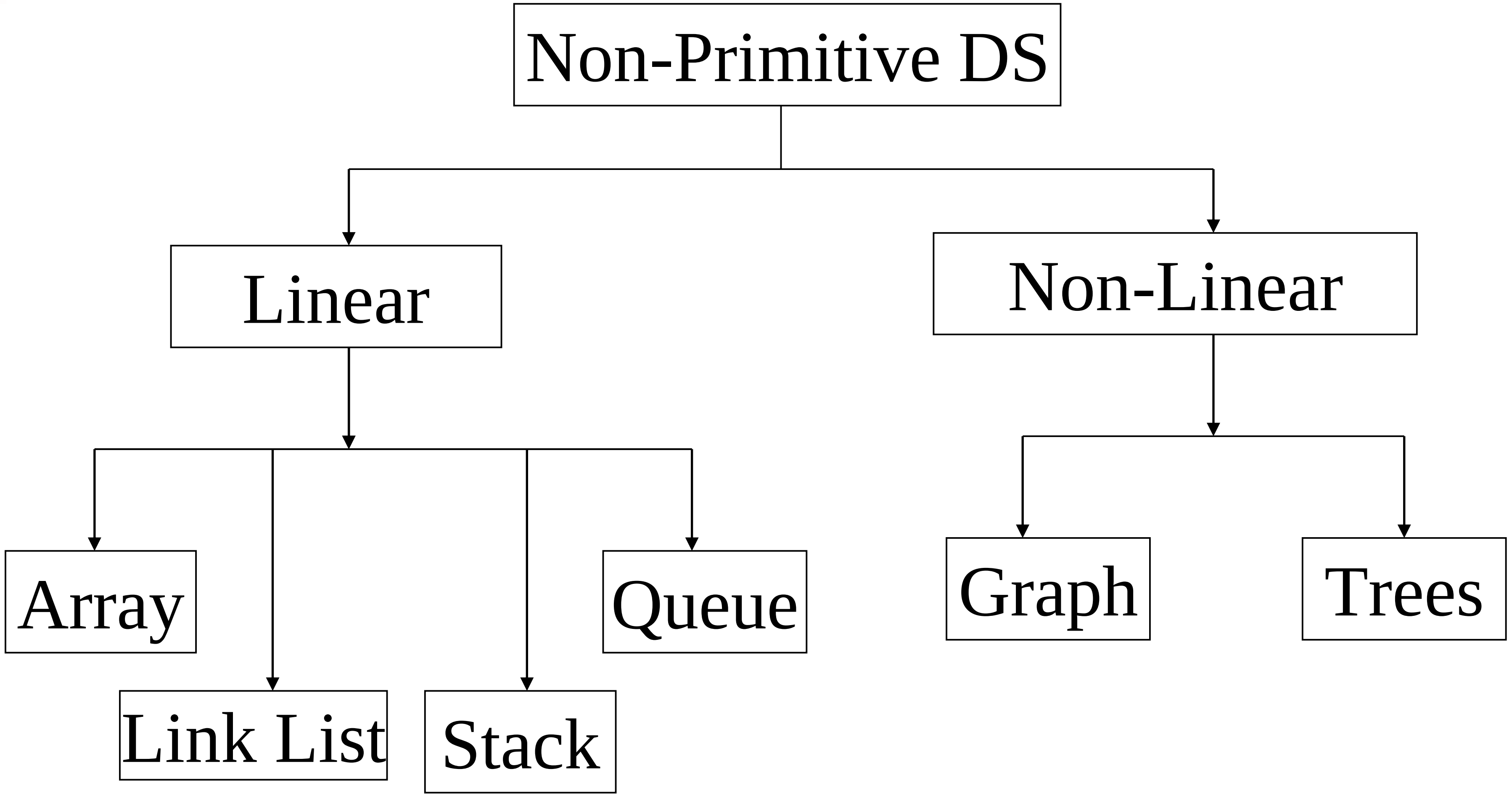
January 12, 2025

Faculty Profile

Types of Data Structure



Types of Data Structure



Data Structure Operations

Operations perform on any linear structure:

- **Traversal:** processing each element in the list
- **Search:** Finding the location of the element with a 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 elements in some type of order
- **Merging:** Combining two lists into a single list

Linear Array

A list of finite number *n of homogeneous* data elements

- The elements of the array are referenced respectively an *index set* consisting of *n* consecutive numbers
- The elements of the array are stored respectively in successive memory locations

The number *n* of element is called the *length* or *size* of the array

Some Array Terminology

Array name

```
temperature[2]
```

Index - also called a *subscript*
- must be an `int`,
- or an expression that evaluates to an `int`

```
temperature[n + 2]
```

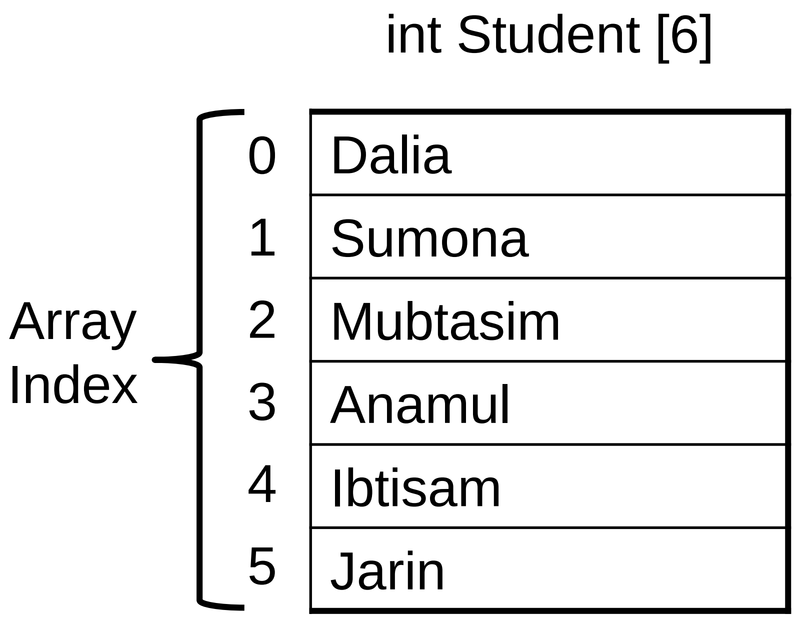
Array *variable* or *Indexed variable*

```
temperature[n+2]
```

Value of the array or indexed variable
- also called an element of the array

```
temperature[n + 2] = 32;
```

Linear Array



Student [0] = Dalia

Student [1] = Sumona

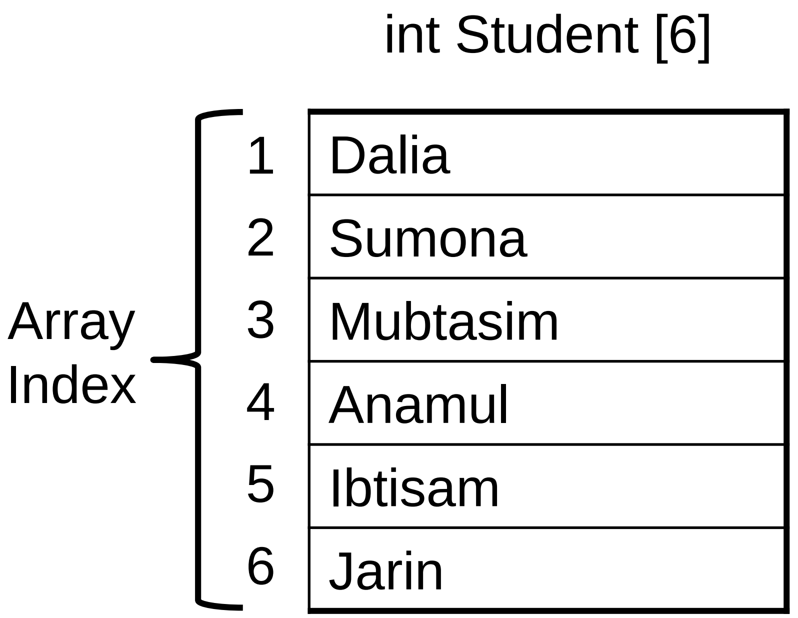
Student [2] = Mubtasim

Student [3] = Anamul

Student [4] = Ibtisam

Student [5] = Jarin

Linear Array



Student [1] = Dalia

Student [2] = Sumona

Student [3] = Mubtasim

Student [4] = Anamul

Student [5] = Ibtisam

Student [6] = Jarin

Representation of Linear Array in Memory

The number n of elements is called the length or size of the array, if not defined we assume index started from 1, 2, 3, 4, …..,n.

In general the length or the number of data elements of array can be obtained from index set by formula:

$$\text{Length} = UB - LB + 1$$

UB is upper bound the largest index, and LB is lower bound the smallest index.

Representation of Linear Array in Memory

Example :

An automobile company uses an array AUTO to record the number of auto mobile sold each year from 1932 through 1984.

LB = 1932

UB = 1984

Length = UB – LB+1 = 1984 – 1932+1 =53

AUTO[k] = Number of auto mobiles sold in the year K

AUTO[1950] = Number of auto mobiles sold in the year 1950

Representation of Linear Array in Memory

- ❖ Let LA be a linear array in the memory of the computer
- ❖ LOC(LA[K]) = address of the element LA[K] of the array LA
- ❖ LOC(LA[K]) = *Base*(LA) + *w**(K - lower bound)

where, *Base*(LA) = the address of the first element of LA

w $\rightarrow$ length of memory location required.

For real number: 4 byte

integer: 2 byte

character: 1 byte

Representation of Linear Array in Memory

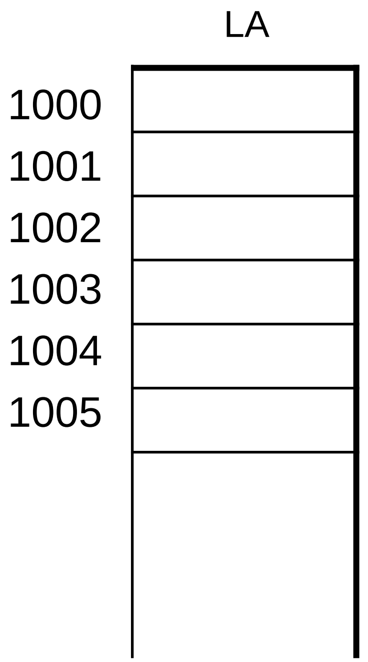
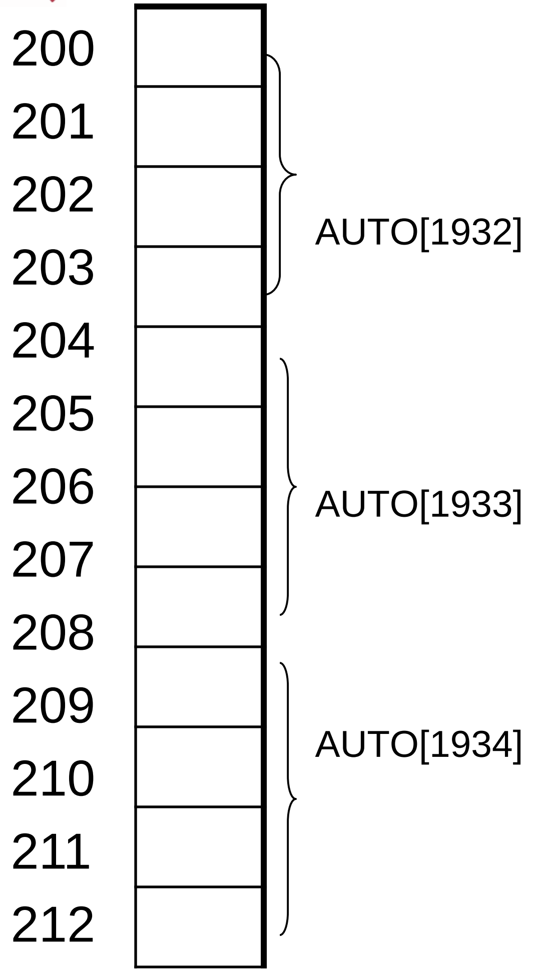


Fig : Computer memory

Representation of Linear Array in Memory



An automobile company uses an array AUTO to record the number of auto mobile sold each year from 1932 through 1984.
Suppose Base(AUTO) = 200, and w = 4.

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) =204
LOC(AUTO[1934]) = 208

The address of the array element for the year K = 1965 can be obtained by using :

LOC(AUTO[1965]) = Base(AUTO) + w*(1965 – lower bound)

=200+4*(1965-1932)=332

Representation of Linear Array in Memory

4.1 Consider the linear arrays AAA(5 : 50), BBB(−5 : 10) and CCC(18).

(a) Find the number of elements in each array.

(b) Suppose *Base*(AAA) = 300 and *w* = 4 words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55].

(a) The number of elements is equal to the length; hence use the formula

$$\text{Length} = \text{UB} - \text{LB} + 1$$

Accordingly,

$$\text{Length(AAA)} = 50 - 5 + 1 = 46$$
$$\text{Length(BBB)} = 10 - (-5) + 1 = 16$$
$$\text{Length(CCC)} = 18 - 1 + 1 = 18$$

Representation of Linear Array in Memory

4.1 Consider the linear arrays AAA(5 : 50), BBB(−5 : 10) and CCC(18).

(a) Find the number of elements in each array.

(b) Suppose *Base*(AAA) = 300 and *w* = 4 words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55].

(b) Use the formula

$$\text{LOC}(\text{AAA}[K]) = \text{Base}(\text{AAA}) + w(K - \text{L}$$

Hence:

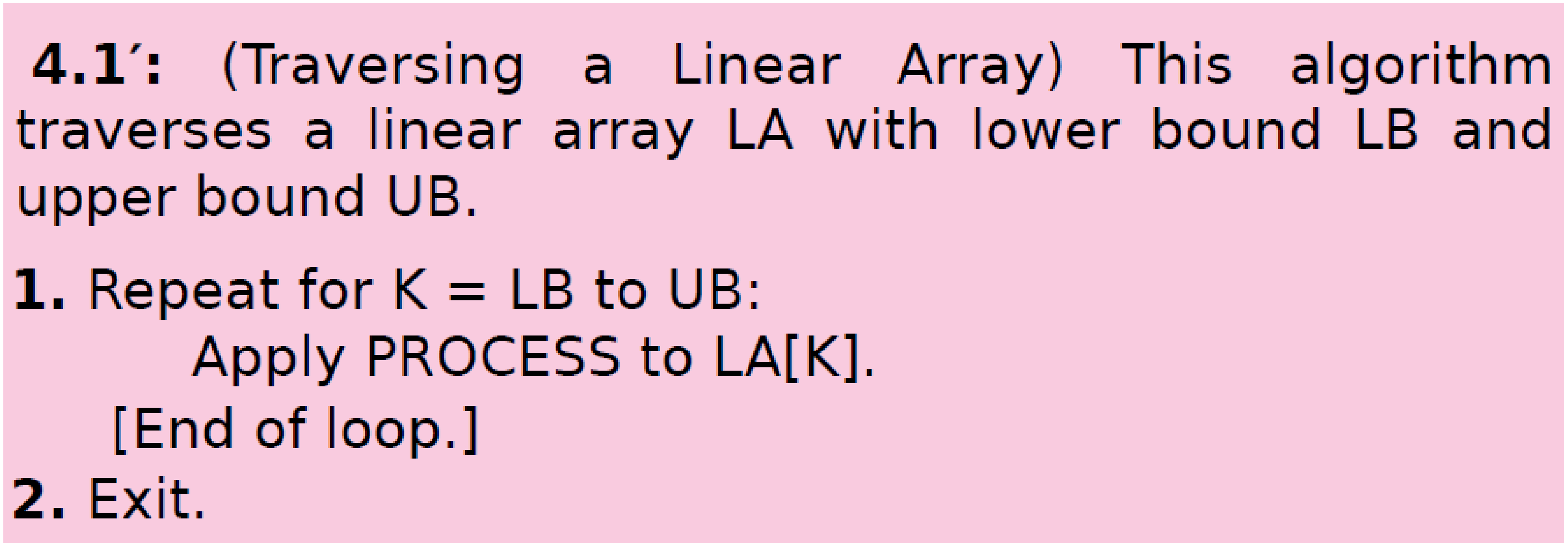
$$\text{LOC}(\text{AAA}[15]) = 300 + 4(15 - 5) = 3$$

$$\text{LOC}(\text{AAA}[35]) = 300 + 4(35 - 5) = 4$$

AAA[55] is not an element of AAA, since 55 exceeds UB = 50.

Traversing a Linear Array

Print the contents of each element of DATA or Count the number of elements of DATA with a given property. This can be accomplished by traversing DATA.



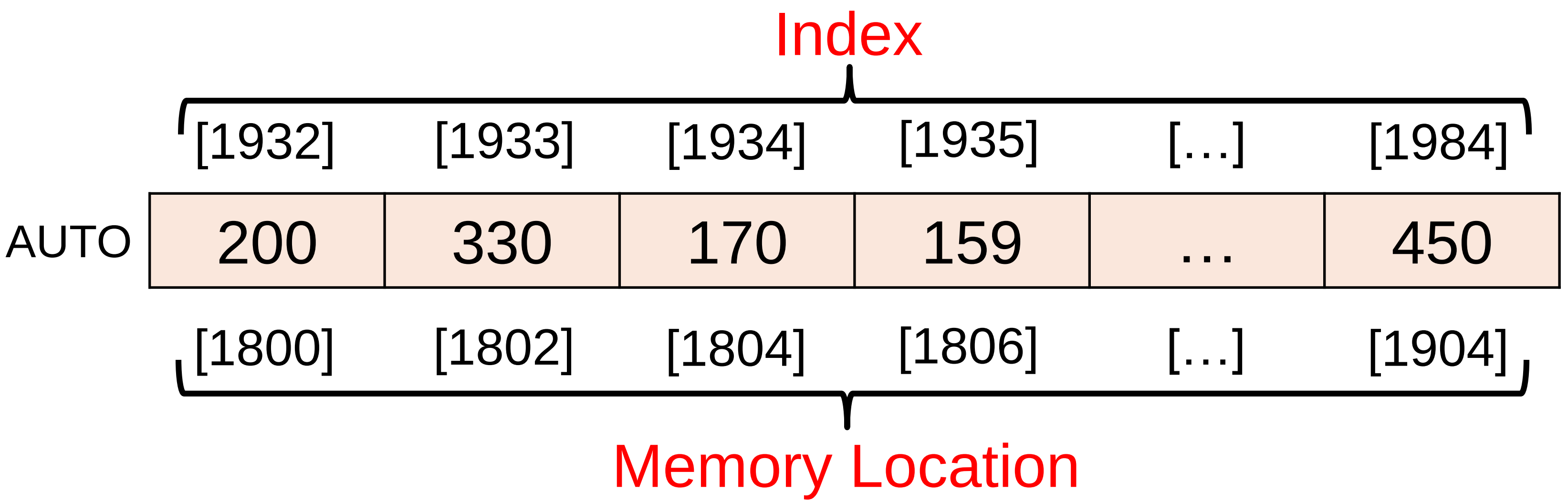
4.1′: (Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

```
1. Repeat for K = LB to UB:
        Apply PROCESS to LA[K].
     [End of loop.]
2. Exit.
```

Traversing a Linear Array

An automobile company uses an array AUTO to record the number of automobile sold each year from 1932 ~ 1984.

a) Print each year and the no. of automobiles sold in that year

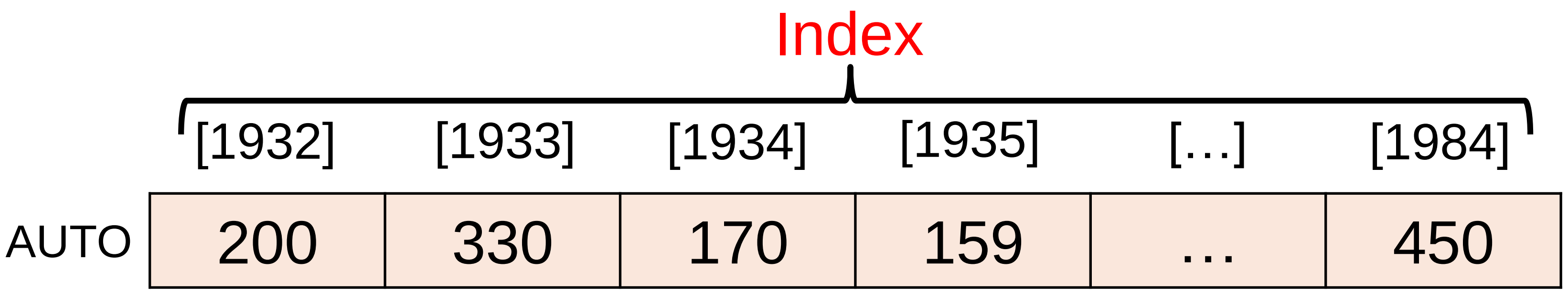


```
1. Repeat for K = 1932 to 1984:
       Print : K, AUTO[K]
2. Exit.
```

Traversing a Linear Array

An automobile company uses an array AUTO to record the number of automobile sold each year from 1932 ~ 1984.

a) Print each year and the no. of automobiles sold in that year



K	AUTO[K]

```
1. Repeat for K = 1932 to 1984:
        Print : K, AUTO[K]
2. Exit.
```

Traversing a Linear Array

An automobile company uses an array AUTO to record the number of automobile sold each year from 1932 ~ 1984.

a) Find the number NUM of years during which more than 300 automobiles were sold.

	[1932]	[1933]	[1934]	[1935]	[…]	[1984]
AUTO	200	330	170	159	…	450

```
1.  Set NUM : = 0.
2.  Repeat for K = 1932 to 1984:
       if AUTO[K]> 300,
          then : set NUM : = NUM+1
3.  Exit.
```

K	AUTO[K]	AUTO[K] > 300	NUM=0

Traversing a Linear Array

An automobile company uses an array AUTO to record the number of auto mobile sold each year from 1932 ~ 1984.

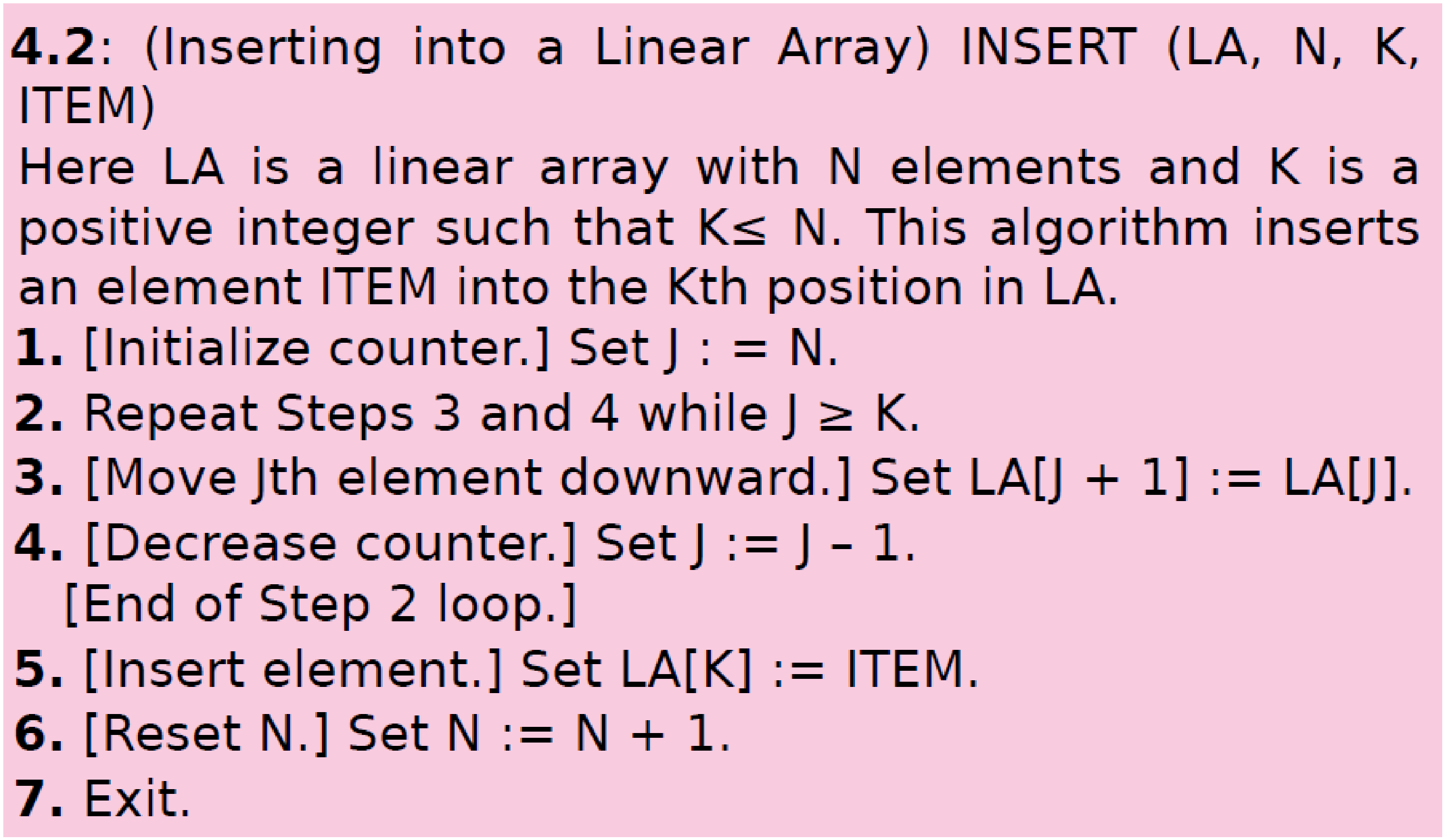
a) Print each year and the no. of automobiles sold in that year

b) Find the number NUM of years during which more than 300 automobiles were sold.

```
1. Repeat for K = 1932 to 1984:
       Print : K, AUTO[K]
2. Exit.
```

```
1. Set NUM : = 0.
2. Repeat for K = 1932 to 1984:
       if AUTO[K]> 300,
           then : set NUM : = NUM+1
3. Exit.
```

Insert into Linear Array



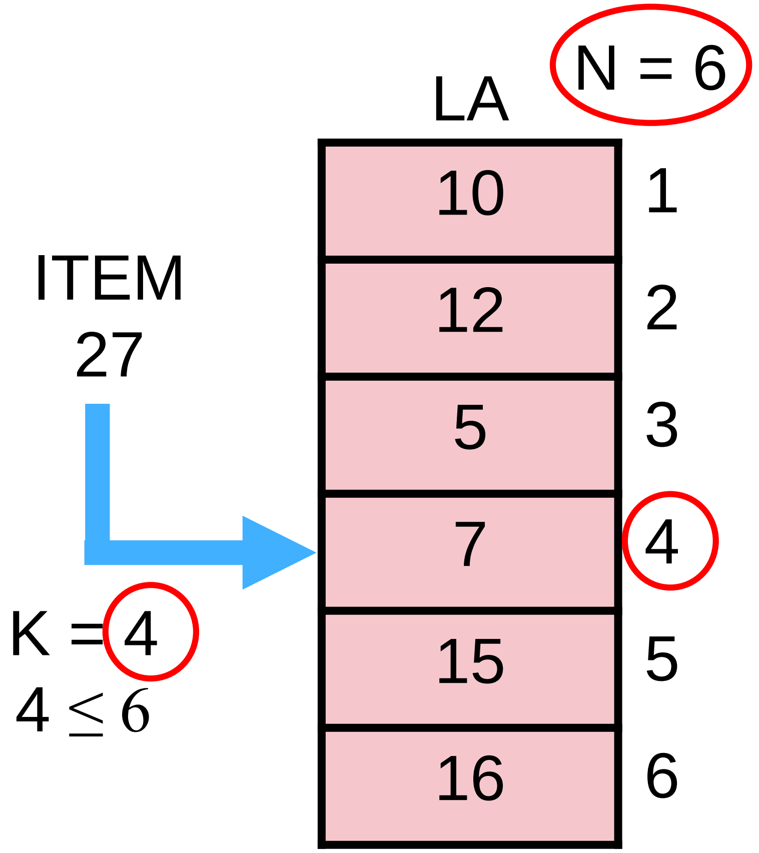
4.2: (Inserting into a Linear Array) INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤ 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 ≥ 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.
```

Insert into Linear Array

Here LA is a linear array with N elements and K is a positive integer such that K≤ N. This algorithm inserts an element ITEM into the Kth position in LA.



Insert into Linear Array

ITEM = 27   K = 4

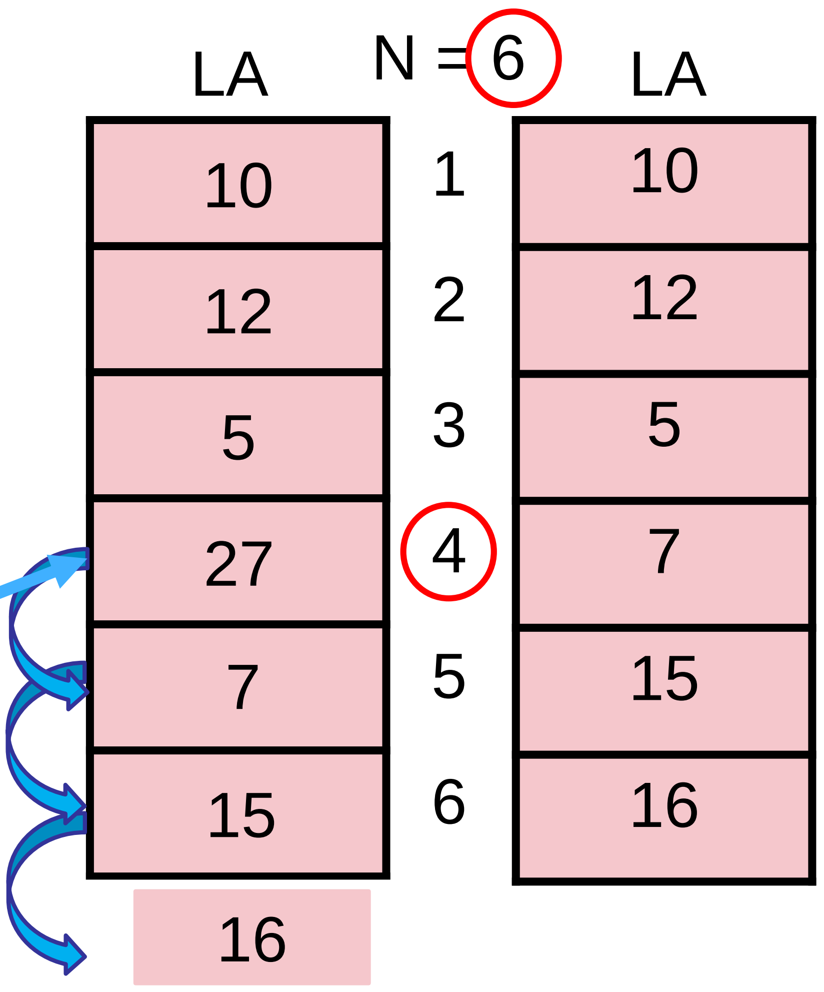
4.2: (Inserting into a Linear Array) INSERT (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that K≤ N. This algorithm inserts an element ITEM into the Kth position in LA.

```
1. [Initialize counter.] Set J := N.
2. Repeat Steps 3 and 4 while J ≥ 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.
```

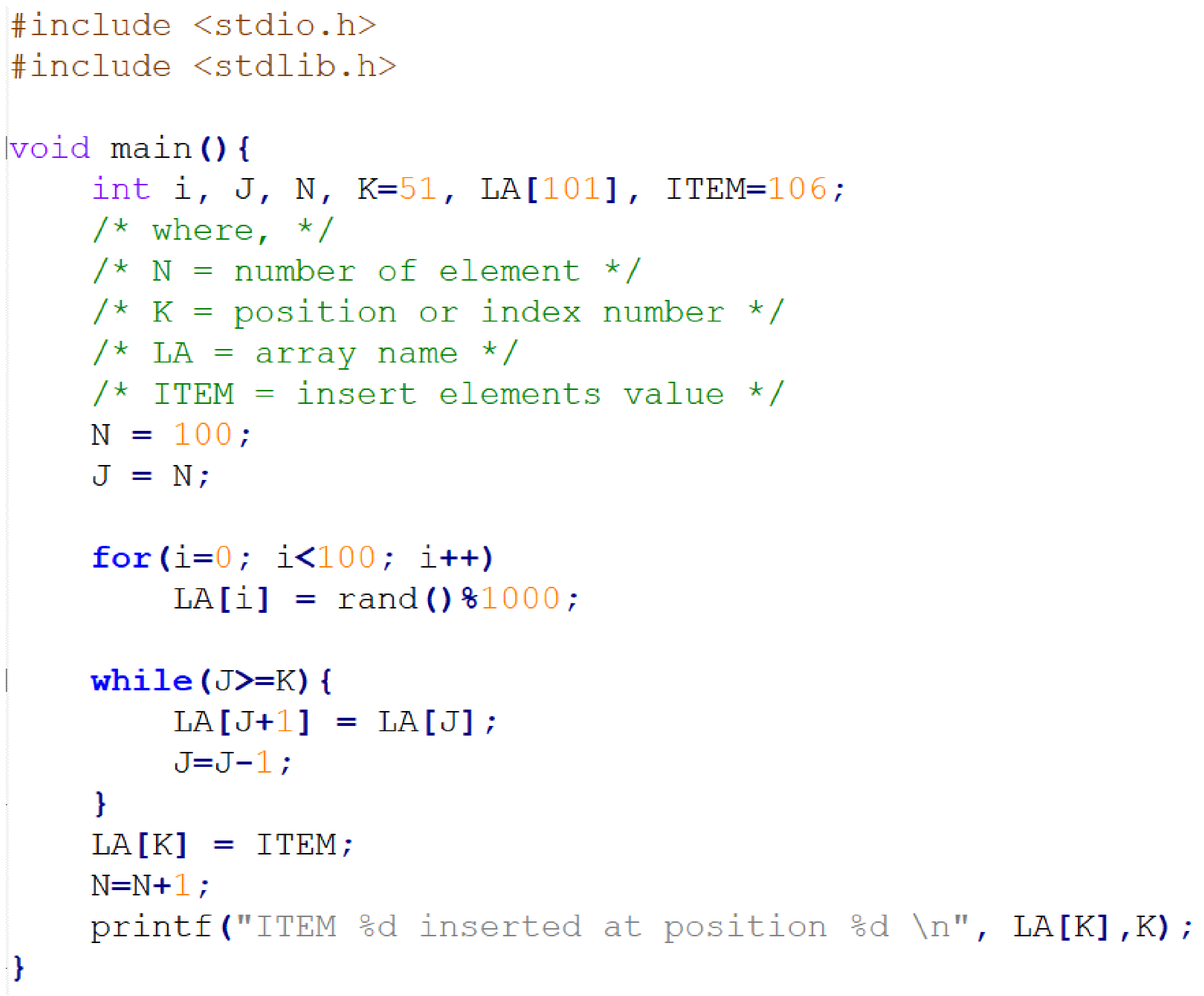
J = 6

LA[4] = 27
N = N+1

```
While(J ≥ K)
{
   LA[J+1]=LA[J];
   J=J-1;
}
```



Insert into Linear Array



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

void main(){
    int i, J, N, K=51, LA[101], ITEM=106;
    /* where, */
    /* N = number of element */
    /* K = position or index number */
    /* LA = array name */
    /* ITEM = insert elements value */
    N = 100;
    J = N;

    for(i=0; i<100; i++)
        LA[i] = rand()%1000;

    while(J>=K){
        LA[J+1] = LA[J];
        J=J-1;
    }
    LA[K] = ITEM;
    N=N+1;
    printf("ITEM %d inserted at position %d \n", LA[K],K);
}
```

Deleting from Linear Array

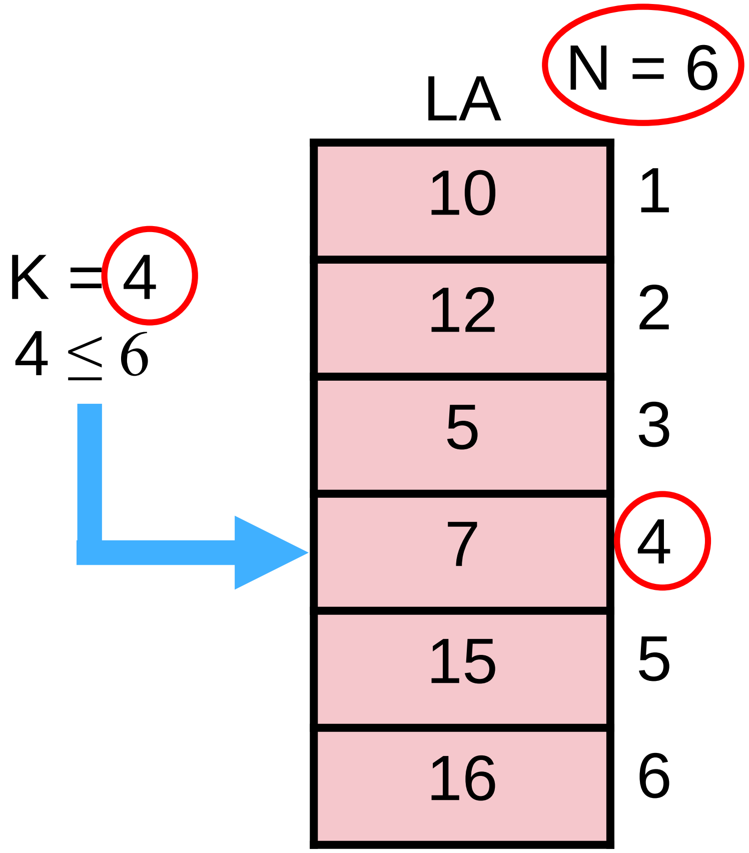
4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)

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 from Linear Array

Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. This algorithm deletes the Kth element from LA.



Deleting from Linear Array

K = 4

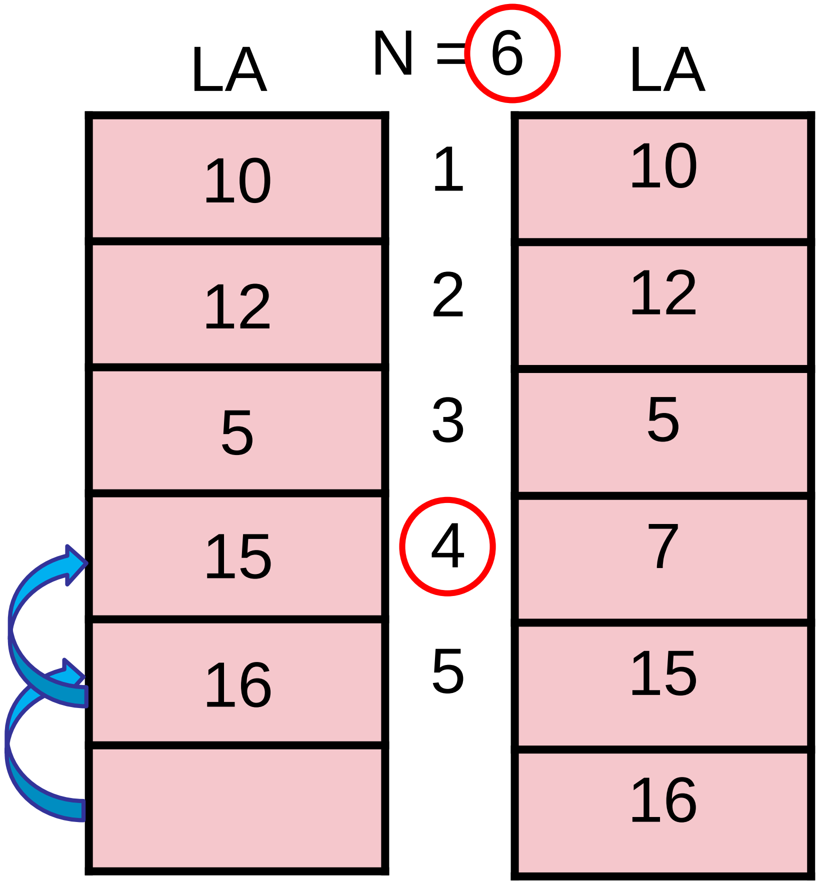
4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. This algorithm deletes the Kth element from LA.

```
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.
```

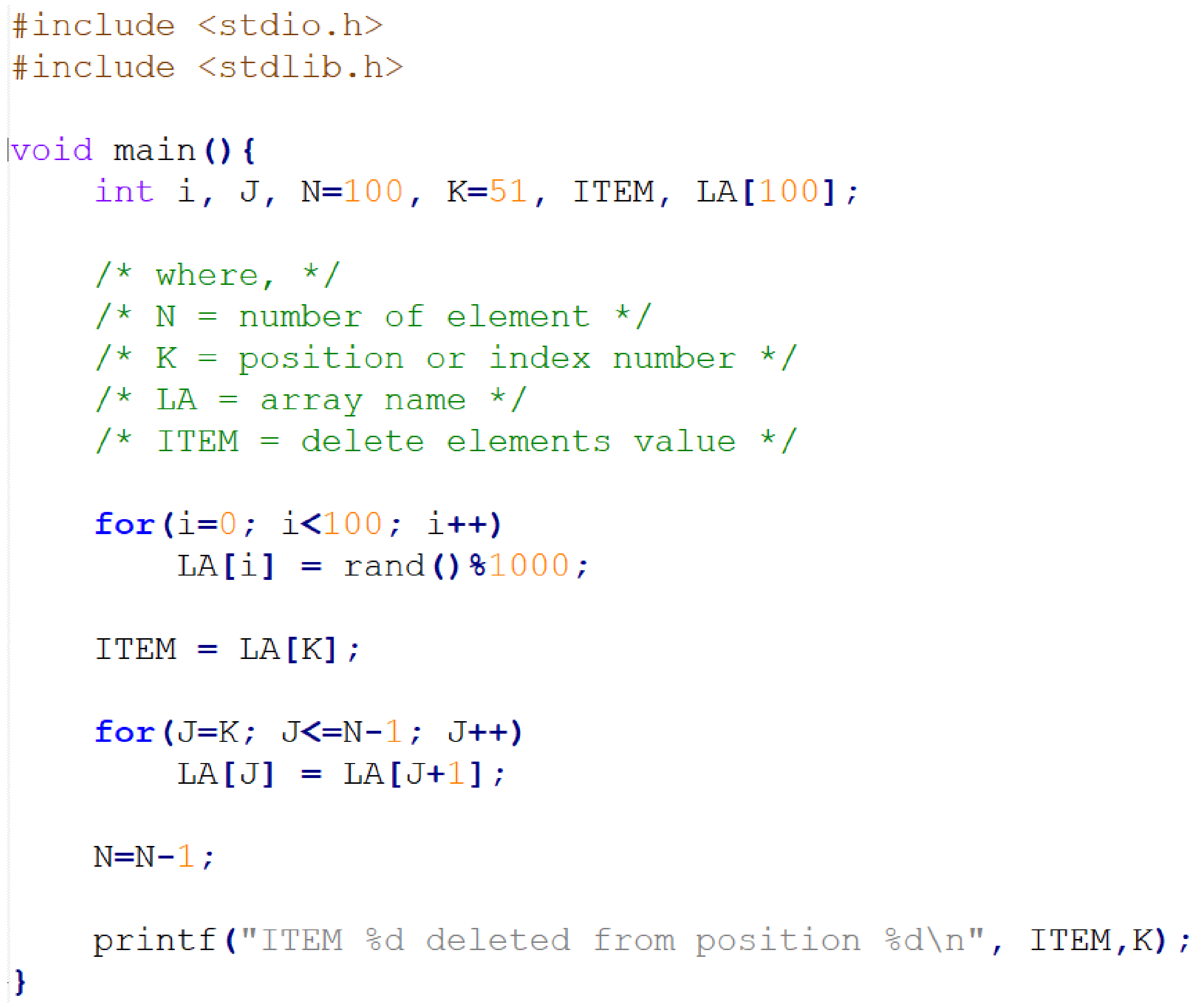
ITEM = LA[4] = 7

N = N-1
i.e. N=6-1 =5

```
for(J=K; J≤N-1; K++)
{
  LA[J]=LA[J+1];
}
```



Deleting from Linear Array



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

void main(){
    int i, J, N=100, K=51, ITEM, LA[100];

    /* where, */
    /* N = number of element */
    /* K = position or index number */
    /* LA = array name */
    /* ITEM = delete elements value */

    for(i=0; i<100; i++)
        LA[i] = rand()%1000;

    ITEM = LA[K];

    for(J=K; J<=N-1; J++)
        LA[J] = LA[J+1];

    N=N-1;

    printf("ITEM %d deleted from position %d\n", ITEM,K);
}
```

Linear Search

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.
```

Linear Search

```c
#include <stdio.h>

void main(){
    int DATA[11]={8,2,5,9,1,4,7,10,3,6},
    i, n=10, ITEM=100, LOC;

    DATA[n+1]=ITEM;
    LOC=0;

    while(DATA[LOC]!=ITEM){
        LOC=LOC+1;
    }

    if (LOC==n+1){
        LOC=-1;
    }

    if (LOC==-1){
        printf("\n ITEM is not in the List\n");
    }
    else{
        printf("\n ITEM found at Position %d \n", LOC);
    }
}
```

Linear Search

```
#include <stdio.h>

void main(){
    int DATA[11]={8,2,5,9,1,4,7,10,3,6},
    i, n=10, ITEM=100, LOC;

    DATA[n+1]=ITEM;
    LOC=0;

    while(DATA[LOC]!=ITEM){
        LOC=LOC+1;
    }

    if (LOC==n+1){
        LOC=-1;
    }

    if (LOC==-1){
        printf("\n ITEM is not in the List\n");
    }
    else{
        printf("\n ITEM found at Position %d \n", LOC);
    }
}
```

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.
```

Binary Search

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.
```

LB = 0
UB = 14
ITEM = 33

BEG = 0
END = 14
MID =7
ITEM = 33

DATA

6	0
13	1
14	2
25	3
33	4
43	5
51	6
53	7
64	8
72	9
84	10
93	11
95	12
96	13
97	14

Binary Search

Ex. Binary search for ITEM 33

BEG = 0

END = 14

MID = INT((0+14)/2) = 7

```
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.
```

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ BEG														↑ END

Binary Search

Ex. Binary search for ITEM 33

BEG = 0

END = 14

MID = INT((0+14)/2) = 7

DATA[MID] = DATA[7] = 53

```
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.
```

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
BEG							MID							END

Binary Search

Ex. Binary search for ITEM 33

BEG = 0

END = 6

MID = INT((0+14)/2) = 7

DATA[MID] = DATA[7] = 53

33 < 53 → END = 7-1 = 6

```
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.
```

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ BEG						↑ END								

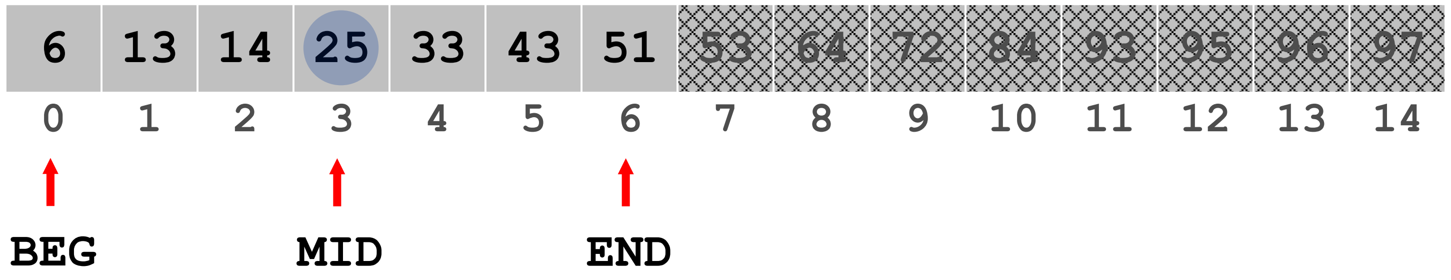
Binary Search

Ex. Binary search for ITEM 33

BEG = 4

END = 6

```
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.
```



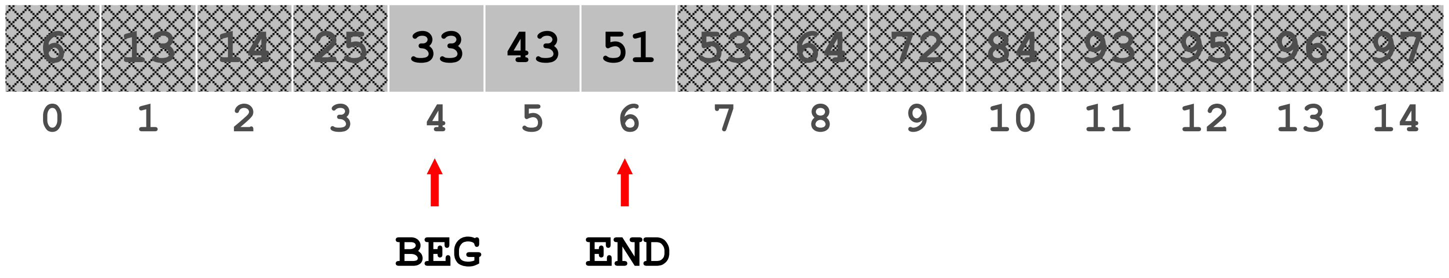
Binary Search

Ex. Binary search for ITEM 33

BEG = 4

END = 6

```
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.
```



Binary Search

Ex. Binary search for ITEM 33

BEG = 4

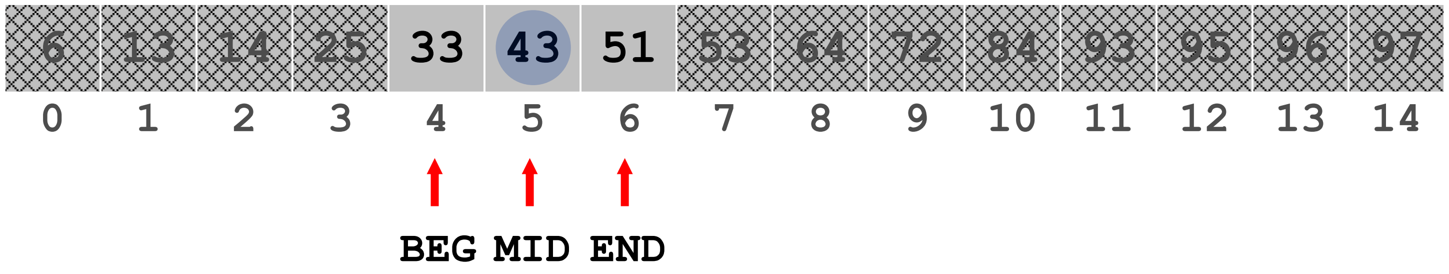
END = 4

MID = INT((4+6)/2) = 5

DATA[MID] = DATA[5] = 43

33 < 43 → END = 5-1 = 4

```
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.
```



Binary Search

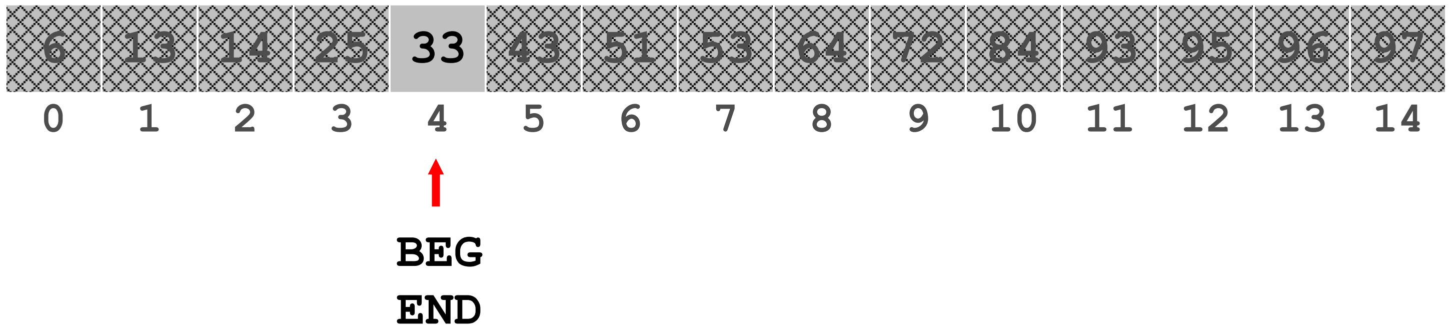
Ex. Binary search for ITEM 33

BEG = 4

END = 4

MID = INT((4+4)/2) = 4

```
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.
```



Binary Search

Ex. Binary search for ITEM 33

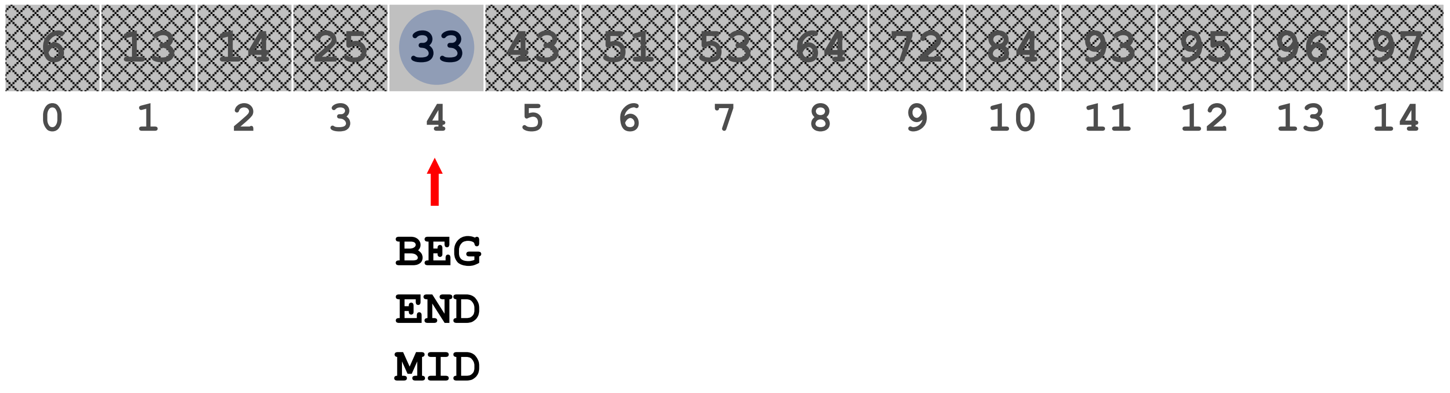
BEG = 4

END = 4

MID = INT((4+4)/2) = 4

DATA[MID] = DATA[4] = 33

```
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.
```



Binary Search

Ex. Binary search for ITEM 33

BEG = 4

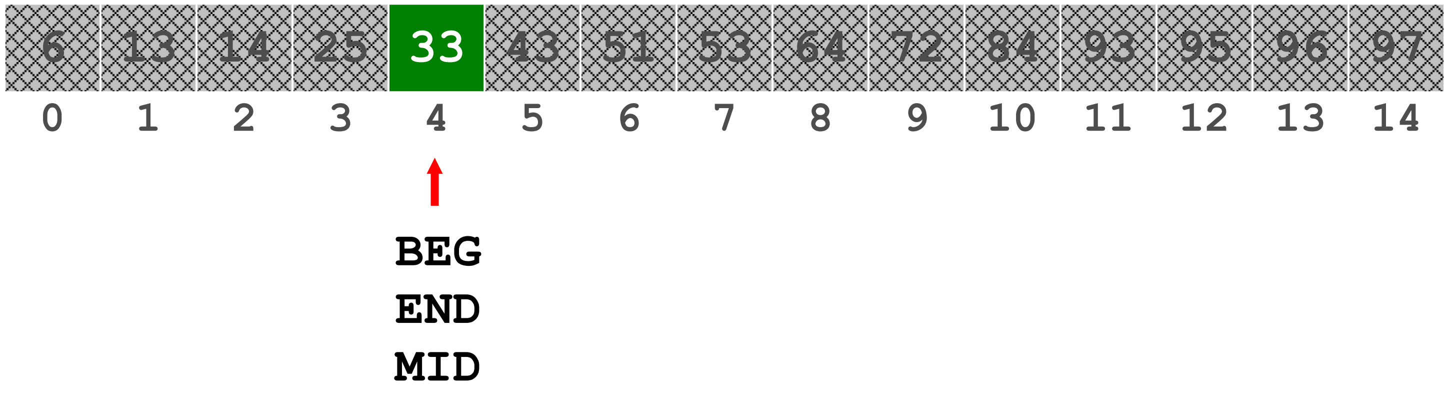
END = 4

MID = INT((4+4)/2) = 4

DATA[MID] = DATA[4] = 33

33 == 33 → ITEM Found

```
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.
```



Binary Search

Simulate binary search algorithm on the array

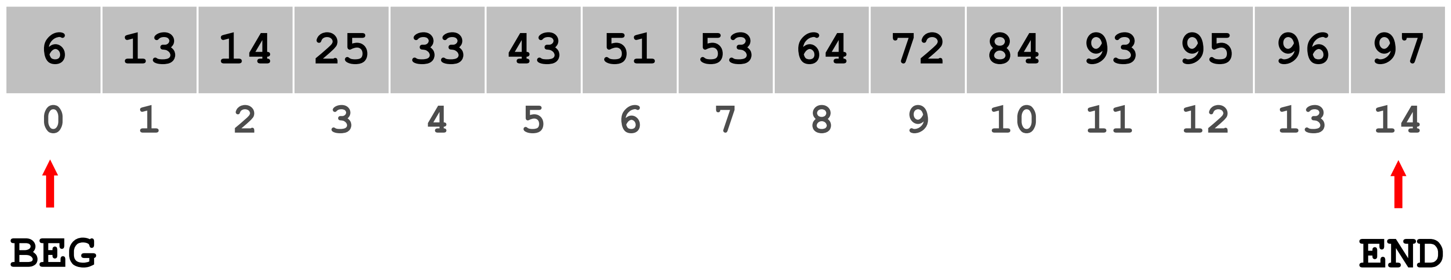
2, 3, 6, 8, 10, 12, 14, 16, 17, 23, 26

when **2, 26,** and **15** are searched showing values of BEG/LOW, END/HIGH, MID, Comparison and Found in a table.

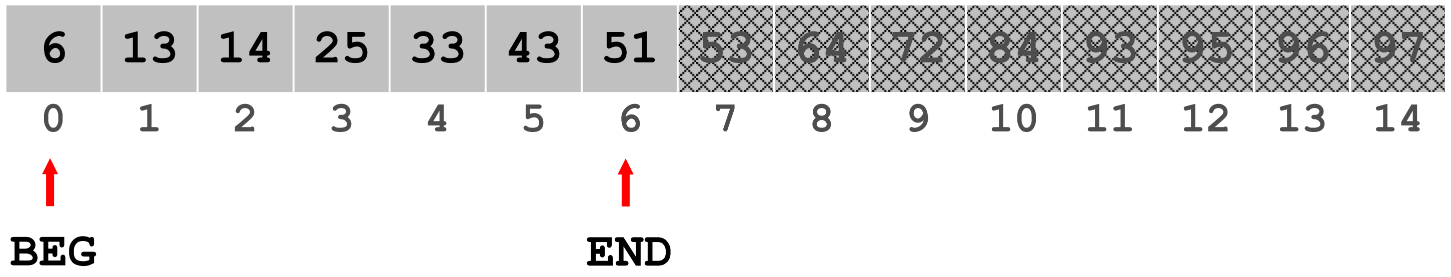
Iteration	BEG	END	MID	Comparison	Found

Binary Search (Complexity Analysis)

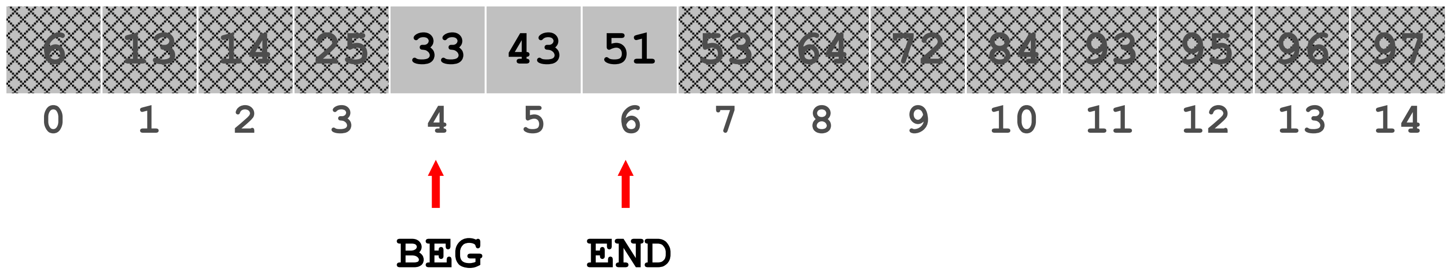
Initial Length of array = n



At Iteration 1: Length of array $= \frac{n}{2}$



At Iteration 2: Length of array $= \frac{n/_2}{2} = \frac{n}{4} = \frac{n}{2^2}$



Binary Search (Complexity Analysis)

At Iteration 3:
Length of array

$$= \frac{n/2}{4} = \frac{n}{8} = \frac{n}{2^3}$$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

After Iteration k:

Length of array $= \frac{n}{2^k}$ $\Longrightarrow$ After k iterations,
the length of the array becomes 1

Therefore, $\frac{n}{2^k} = 1$ $\quad n = 2^k \quad 2^k = n \quad \log_2 2^k = \log_2 n$

$$k \log_2 2 = \log_2 n \qquad k.1 = \log_2 n \qquad k = \log_2 n$$

Binary Search (Complexity Analysis)

Consider an array with 64 elements. Estimate the number of iterations required in worst case scenario while using the Binary search algorithm.

The number of iterations required while considering the worst case in Binary search, $k = \log_2 n$

Here, $n = 64$ Therefore, $k = \log_2 64$

$$\begin{aligned} &= \log_2 2^6 \\ &= 6 \log_2 2 \\ &= 6 . 1 \\ &= 6 \end{aligned}$$

Binary Search

Consider an array with 100 elements. Estimate the number of iterations required in worst case scenario while using the Binary search algorithm.

The number of iterations required while considering the worst case in Binary search, $k = \log_2 n$

Here, $n = 64$ Therefore, $k = \log_2 100$

$$= \log_2 2^{6.65}$$

$$= \log_2 2^{\lceil 6.65 \rceil}$$

$$= \log_2 2^7$$

$$= 7 \log_2 2$$

$$= 7 \,.\, 1$$

$$= 7$$

Binary Search (Complexity Analysis)

```
#include <stdio.h>

void main()
{
    int LB, UB, BEG, END, MID, DATA[7]={2,5,7,9,11,13,15}, ITEM=1;
    LB=0;
    UB=6;
    BEG=LB;
    END=UB;
    MID=(int)((BEG+END)/2);

    while((BEG<=END)&&(DATA[MID]!=ITEM))
    {
        if(ITEM<DATA[MID])
            END=MID-1;
        else
            BEG=MID+1;

    MID=(int)((BEG+END)/2);
    }

    if(DATA[MID]==ITEM)
        printf("LOC = %d", MID);
    else
        printf("ITEM is not in the list.");
}
```

Binary Search (Complexity Analysis)

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.
```

```c
#include <stdio.h>

void main()
{
    int LB, UB, BEG, END, MID, DATA[7]={2,5,7,9,11,13,15}, ITEM=1;
    LB=0;
    UB=6;
    BEG=LB;
    END=UB;
    MID=(int)((BEG+END)/2);

    while((BEG<=END)&&(DATA[MID]!=ITEM))
    {
        if(ITEM<DATA[MID])
            END=MID-1;
        else
            BEG=MID+1;

    MID=(int)((BEG+END)/2);
    }

    if(DATA[MID]==ITEM)
        printf("LOC = %d", MID);
    else
        printf("ITEM is not in the list.");
}
```

Sorting Linear Array (Bubble Sort)

Sorting takes an unordered collection and makes it an ordered one

1	2	3	4	5	6
77	42	35	12	101	5

$$\downarrow$$

1	2	3	4	5	6
5	12	35	42	77	101

Bubble Sort

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.
```

Bubble Sort

"Bubbling Up" the Largest Element

Traverse a collection of elements

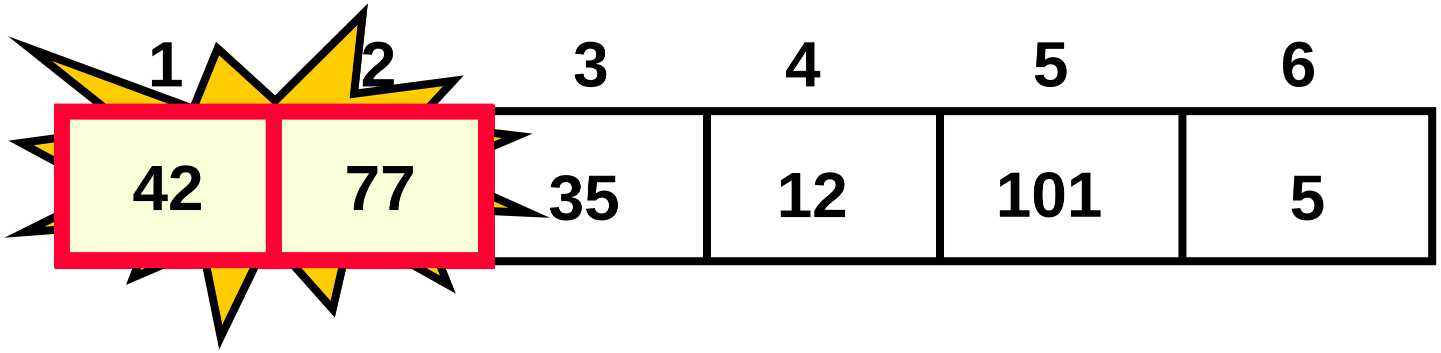
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

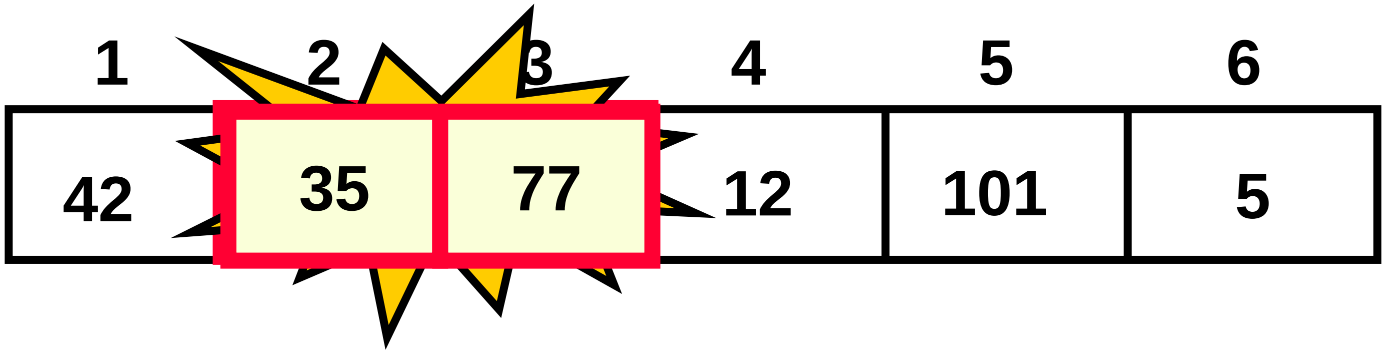


No. of Comparisons: No. of Swap:

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

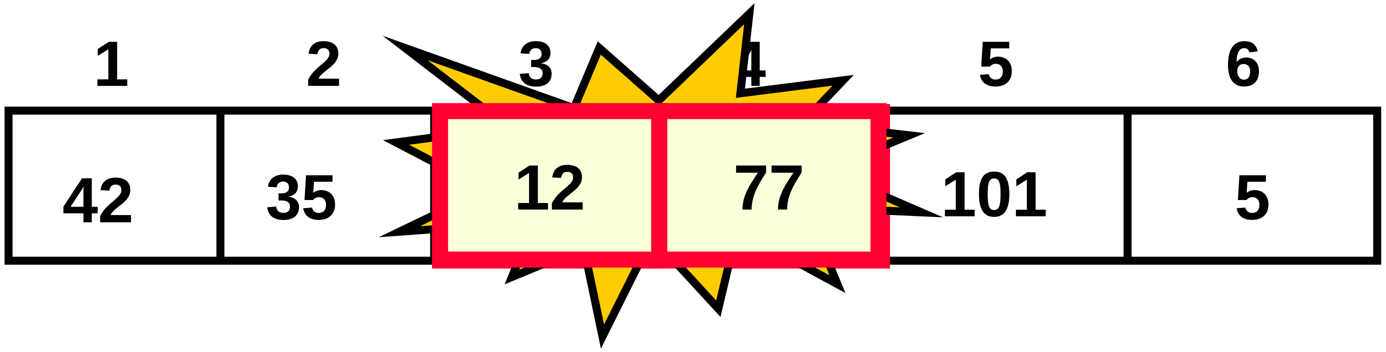


No. of Comparisons: No. of Swap:

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

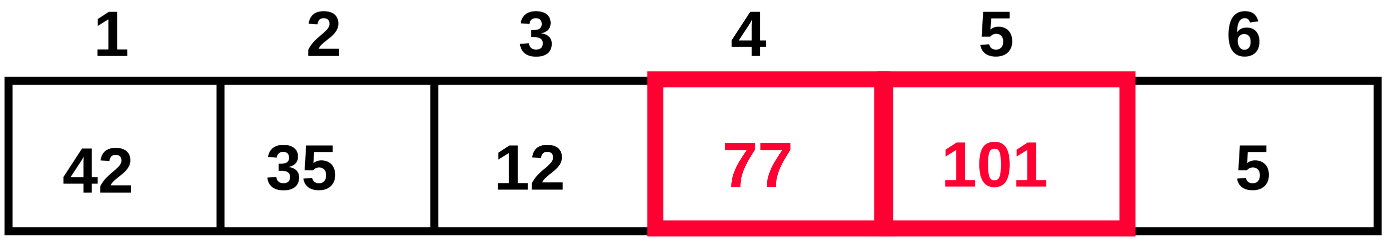


No. of Comparisons: No. of Swap:

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



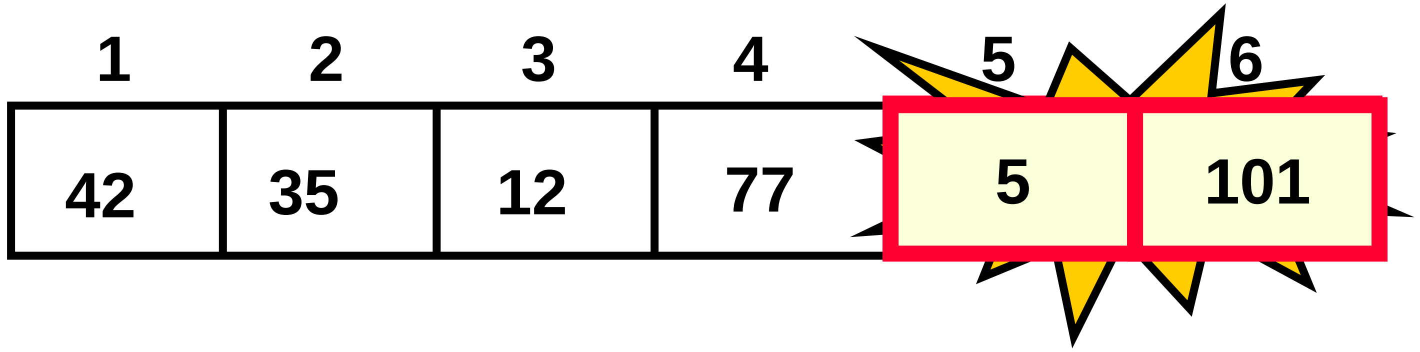
No need to swap

No. of Comparisons: No. of Swap:

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



No. of Comparisons: No. of Swap:

Bubble Sort

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

No. of Comparisons: No. of Swap:

Bubble Sort

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Bubble Sort

- If we have N elements
- And if each time we bubble an element, we place it in its correct location
- Then we repeat the "bubble up" process N – 1 times.
- This guarantees we'll correctly place all N elements.

“Bubbling” All the Elements

N - 1

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

1	2	3	4	5	6
5	12	35	42	77	101

Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

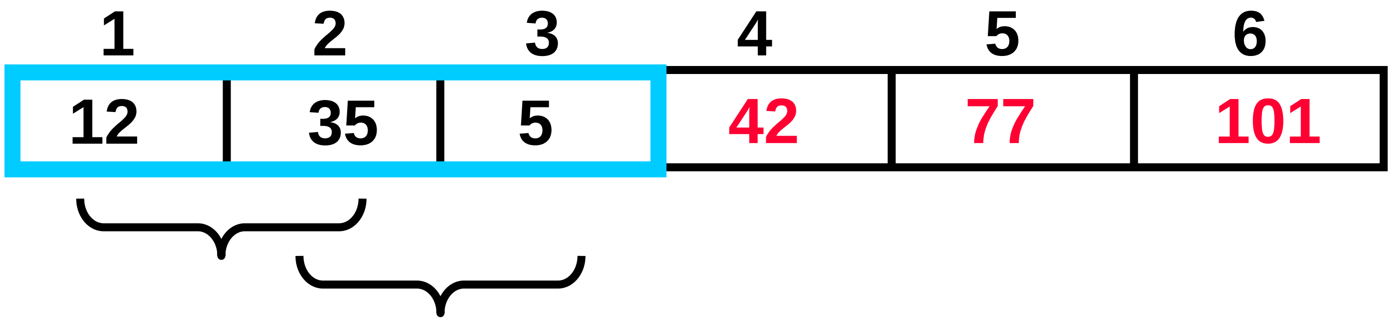
1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

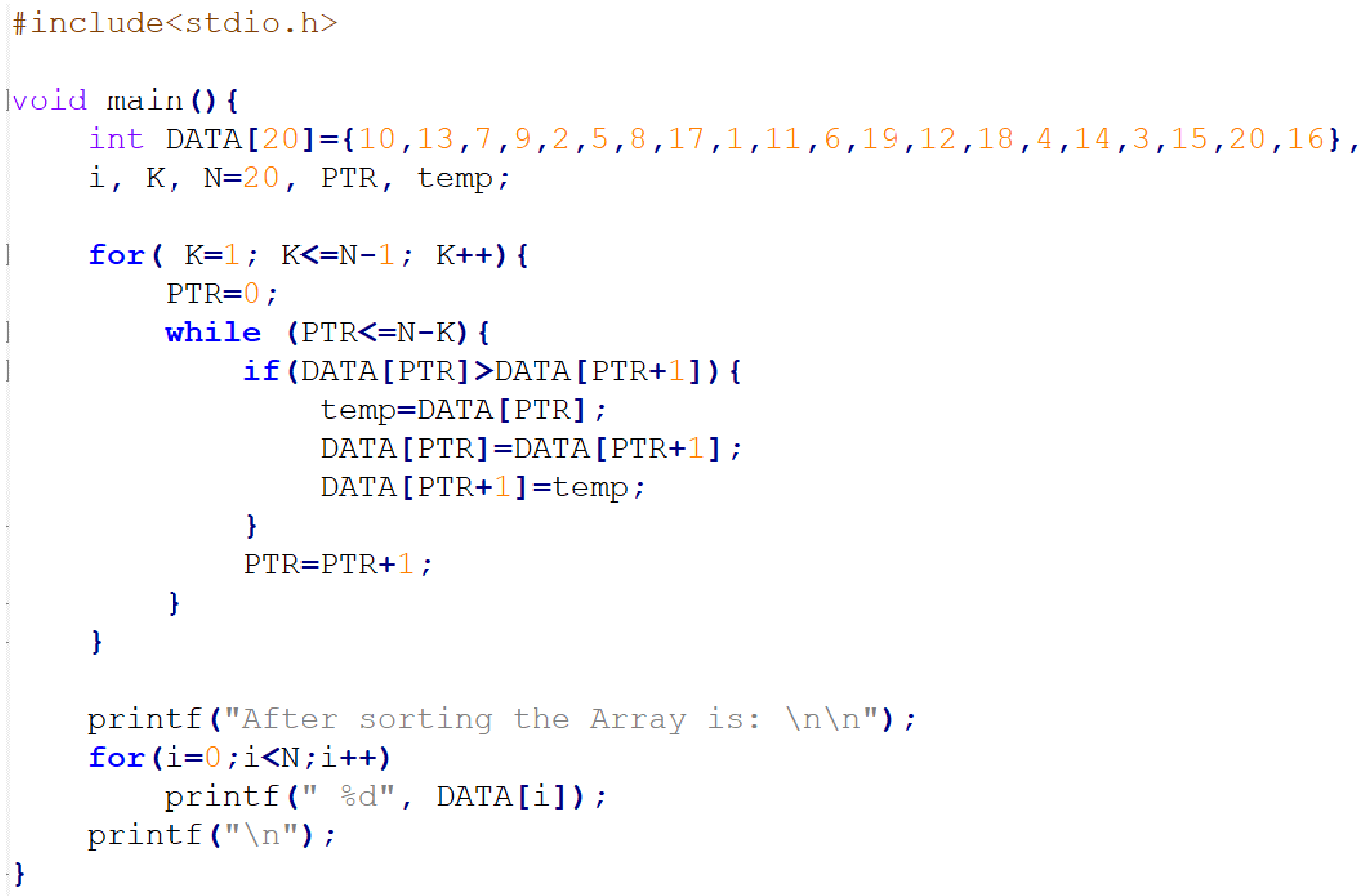
1	2	3	4	5	6
12	5	35	42	77	101

Reducing the Number of Comparisons

- On the Nth "bubble up", we only need to do MAX-N comparisons.
- For example:
  - This is the 4th "bubble up"
  - MAX is 6
  - Thus we have 2 comparisons to do



Bubble Sort Implementation



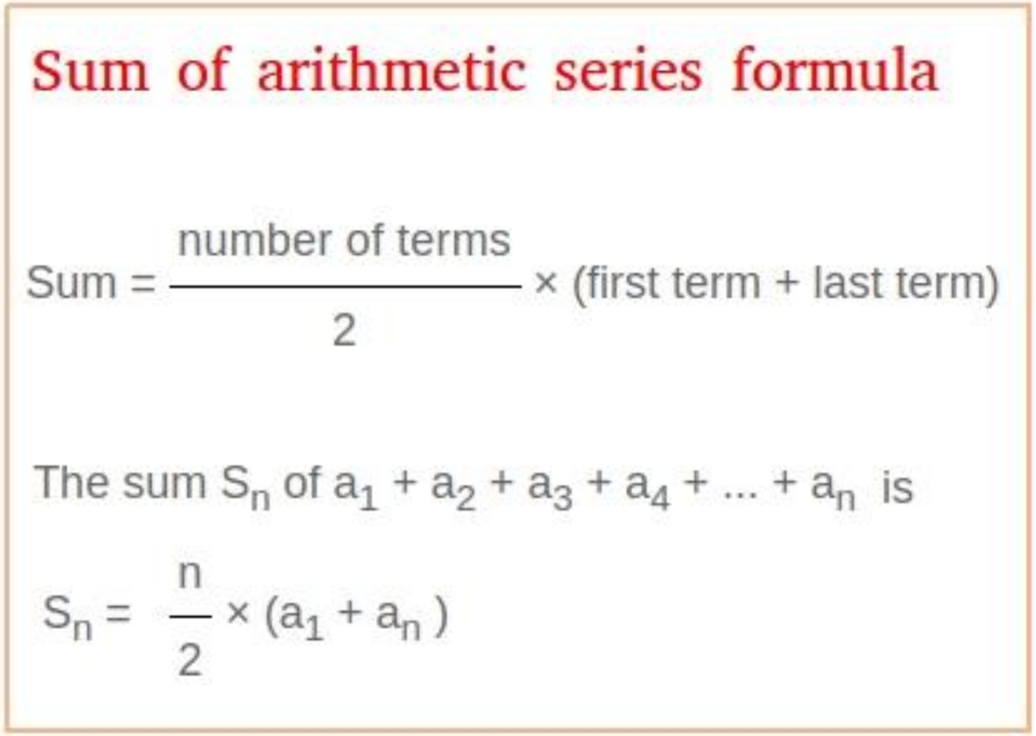
```c
#include<stdio.h>

void main(){
    int DATA[20]={10,13,7,9,2,5,8,17,1,11,6,19,12,18,4,14,3,15,20,16},
    i, K, N=20, PTR, temp;

    for( K=1; K<=N-1; K++){
        PTR=0;
        while (PTR<=N-K){
            if(DATA[PTR]>DATA[PTR+1]){
                temp=DATA[PTR];
                DATA[PTR]=DATA[PTR+1];
                DATA[PTR+1]=temp;
            }
            PTR=PTR+1;
        }
    }

    printf("After sorting the Array is: \n\n");
    for(i=0;i<N;i++)
        printf(" %d", DATA[i]);
    printf("\n");
}
```

Complexity Analysis of Bubble Sort



Sum of arithmetic series formula

$$\text{Sum} = \frac{\text{number of terms}}{2} \times (\text{first term} + \text{last term})$$

The sum S_n of $a_1 + a_2 + a_3 + a_4 + \dots + a_n$ is

$$S_n = \frac{n}{2} \times (a_1 + a_n)$$

Complexity Analysis of Bubble Sort

Traditionally, the time for a sorting algorithm is measured in terms of the number of comparisons. The number $f(n)$ of comparisons in the bubble sort is easily computed. Specifically, there are $n - 1$ comparisons during the first pass, which places the largest element in the last position; there are $n - 2$ comparisons in the second step, which places the second largest element in the next-to-last position; and so on. Thus $\boxed{f(n) = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)}$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2, where n is the number of input items.

Thank You!