INTRODUCTION TO DATA STRUCTURES

Prepared by

Deepthi M Pisharody

Asst Professor

Prajyoti Niketan College, Pudukad

DATA

- Data are simple value or set of values.
- It specifies value of a variable or a constant.
- Eg: name, age, marks

TWO TYPES OF DATA ITEMS

- **Elementary data items-**
- Data items that cannot be further divided
- Eg: first name, last name, age
- **Grouped data items-**
- Data items that be further divided
- Eg:- address- house name, place, pincode

- Field- collection of data items- name, no,class
- Record- collection of related fields (student record)
- File- collection of related records (student file)
- Data → field → record → file

- Entity-
- An entity is something which has some properties called attributes.
- We can assign values to this attributes.
- Entity set
- Entities with similar attributes
- Eg:- employees in an organization
- Information
- Meaningful data or processed data

DATA STUCTURES

A ***data structure*** is basically **a group of data elements** that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used **efficiently.**

- **Logical mathematical model of organisation of data.**
- **Building blocks of program**

- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables

DATA STRUCTURES ARE WIDELY APPLIED IN THE FOLLOWING AREAS:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

CLASSIFICATION OF DATA STRUCTURES

- Data structures are generally categorized into two classes:
- ***primitive* and *non-primitive* data structures.**

PRIMITIVE AND NON-PRIMITIVE DATA STRUCTURES

- Primitive data structures are the fundamental data types which are supported by a programming language.
- Some basic data types are integer, real, character, and boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.
- Non-primitive data structures are those data structures which are created using primitive data structures.
- Examples of such data structures include linked lists, stacks, trees, and graphs.
- Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures

LINEAR AND NON-LINEAR STRUCTURES

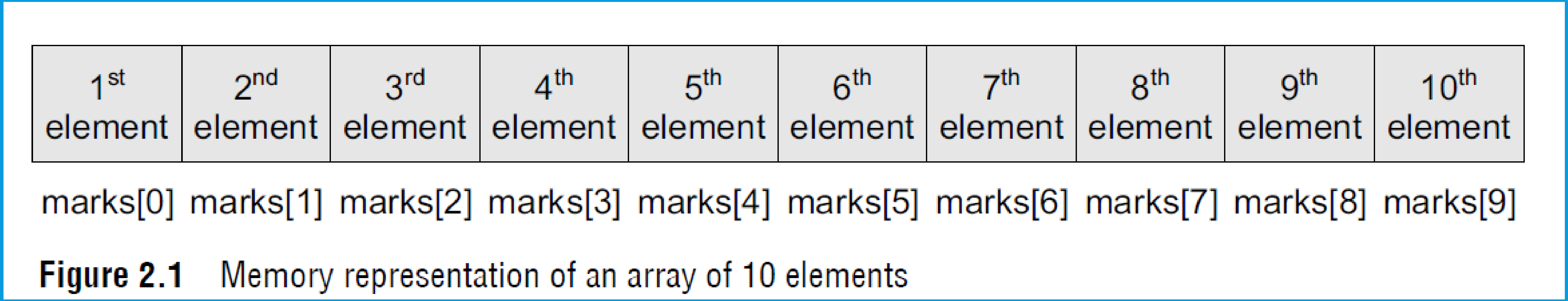
- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
- Examples include arrays, linked lists, stacks, and queues.
- Linear data structures can be represented in memory in two different ways.
- One way is to have to a linear relationship between elements by means of sequential memory locations.
- The other way is to have a linear relationship between elements by means of links.

- However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.
- The relationship of adjacency is not maintained between elements of a non-linear data structure.
- Examples include trees and graphs.

ARRAYS

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).
- In C, arrays are declared using the following syntax:
- `type name[size];`
- For example,
- `int marks[10];`

- The above statement declares an array `marks` that contains 10 elements.
- In C, the array index starts from zero.
- This means that the array marks will contain 10 elements in all.
- The first element will be stored in `marks[0]`, second element in `marks[1]`, so on and so forth.
- Therefore, the last element, that is the 10th element, will be stored in `marks[9]`.



1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Figure 2.1 Memory representation of an array of 10 elements

- Arrays are generally used when we want to store large amount of similar type of data

LIMITATIONS

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

LINKED LISTS

- A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list.
- In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list.
- This feature enables the programmers to write robust programs which require less maintenance.

- In a linked list, each node is allocated space as it is added to the list.
- Every node in the list points to the next node in the list.
- Therefore, in a linked list, every node contains the following two types of data:
- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list

- The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

- Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available.

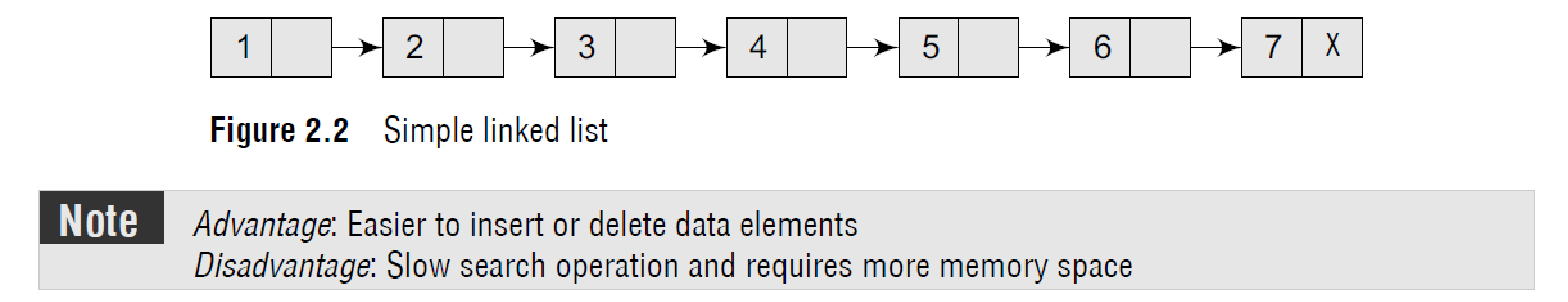


Figure 2.2 Simple linked list

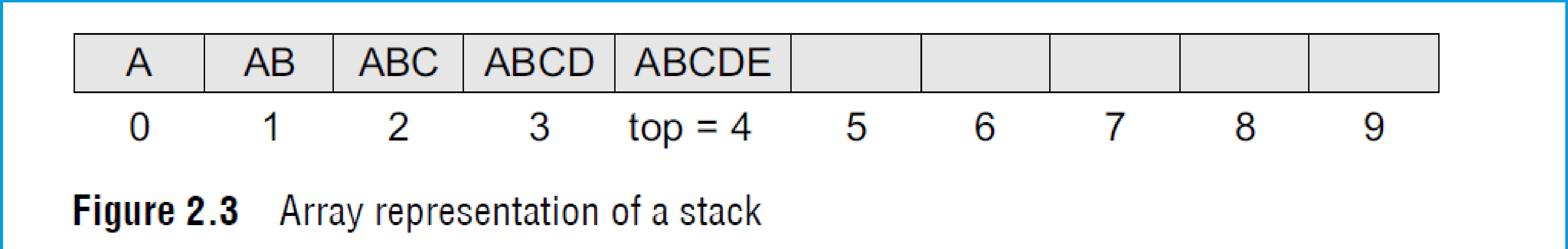
Note

Advantage: Easier to insert or delete data elements
Disadvantage: Slow search operation and requires more memory space

STACKS

- A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the `top` of the stack.
- Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.
- In the computer's memory, stacks can be implemented using arrays or linked lists.

- Every stack has a variable `top` associated with it.
- `Top` is used to store the address of the topmost element of the stack.
- It is this position from where the element will be added or deleted.
- There is another variable `MAX`, which is used to store the maximum number of elements that the stack can store.
- If `top = NULL`, then it indicates that the stack is empty and if `top = MAX–1`, then the stack is full.



A	AB	ABC	ABCD	ABCDE					
0	1	2	3	top = 4	5	6	7	8	9

Figure 2.3 Array representation of a stack

- A stack supports three basic operations: push, pop, and peep.
- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- And the peep operation returns the value of the topmost element of the stack (without deleting it).
- However, before inserting an element in the stack, we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a stack that is already full.
- Similarly, before deleting an element from the stack, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a stack that is already empty.

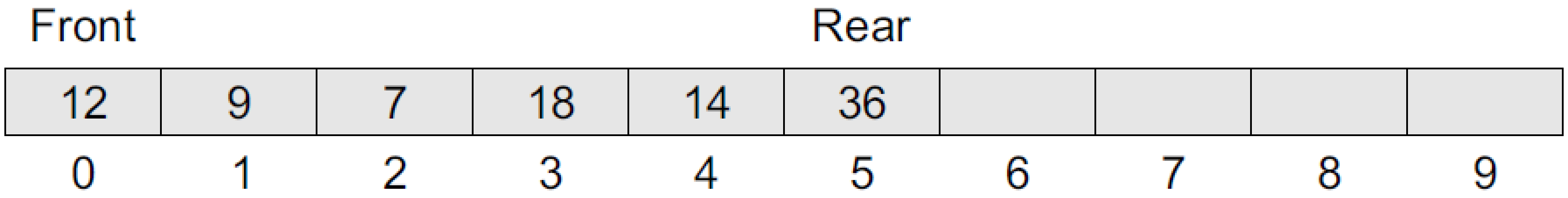
Thank You

INTRODUCTION TO DATA STRUCTURES-PART2

PREPARED BY
DEEPTHI M PISHARODY
ASST PROFESSOR
PRAJYOTI NIKETAN COLLEGE, PUDUKAD

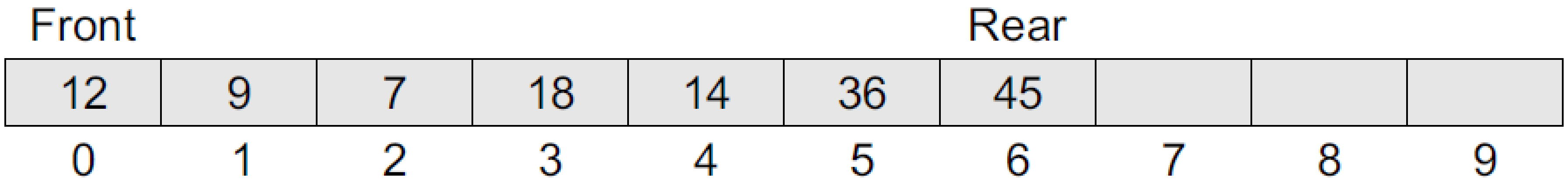
QUEUES

- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the `rear` and removed from the other end called the `front`.
- Like stacks, queues can be implemented by using either arrays or linked lists.
- Every queue has `front` and `rear` variables that point to the position from where deletions and insertions can be done, respectively.



0	1	2	3	4	5	6	7	8	9
12	9	7	18	14	36				

Figure 2.4 Array representation of a queue



0	1	2	3	4	5	6	7	8	9
12	9	7	18	14	36	45			

Figure 2.5 Queue after insertion of a new element

- Here, `front = 0` and `rear = 5`.
- If we want to add one more value to the list, say, if we want to add another element with the value 45, then the `rear` would be incremented by 1 and the value would be stored at the position pointed by the `rear`.
- Here, `front = 0` and `rear = 6`. Every time a new element is to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of `front` will be incremented. Deletions are done only from this end of the queue.

- However, before inserting an element in the queue, we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a queue that is already full.
- A queue is full when `rear = MAX – 1`, where `MAX` is the size of the queue, that is `MAX` specifies the maximum number of elements in the queue.
- Note that we have written `MAX – 1` because the index starts from 0.
- Similarly, before deleting an element from the queue, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a queue that is already empty.
- If `front = NULL` and `rear = NULL`, then there is no element in the queue.

TREES

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.
- The simplest form of a tree is a binary tree.
- A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.
- Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree.
- The root element is the topmost node which is pointed by a ‘root’ pointer.
- If `root = NULL` then the tree is empty.

- *Advantage*: Provides quick search, insert, and delete operations
- *Disadvantage*: Complicated deletion algorithm

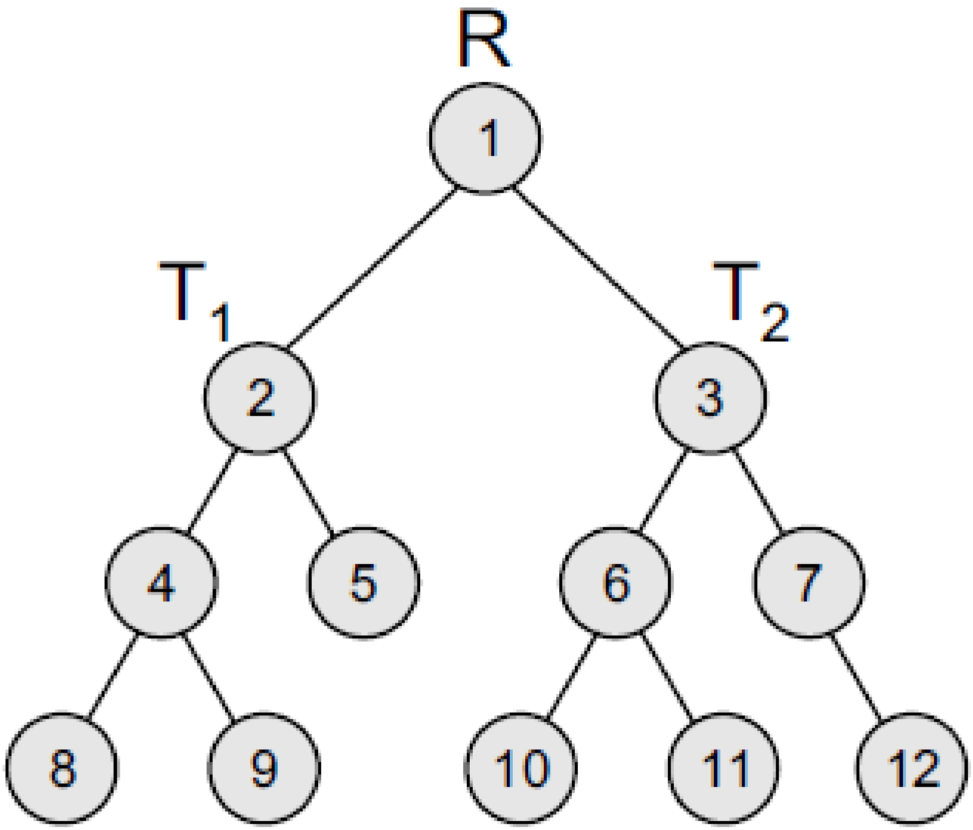


Figure 2.7 Binary tree

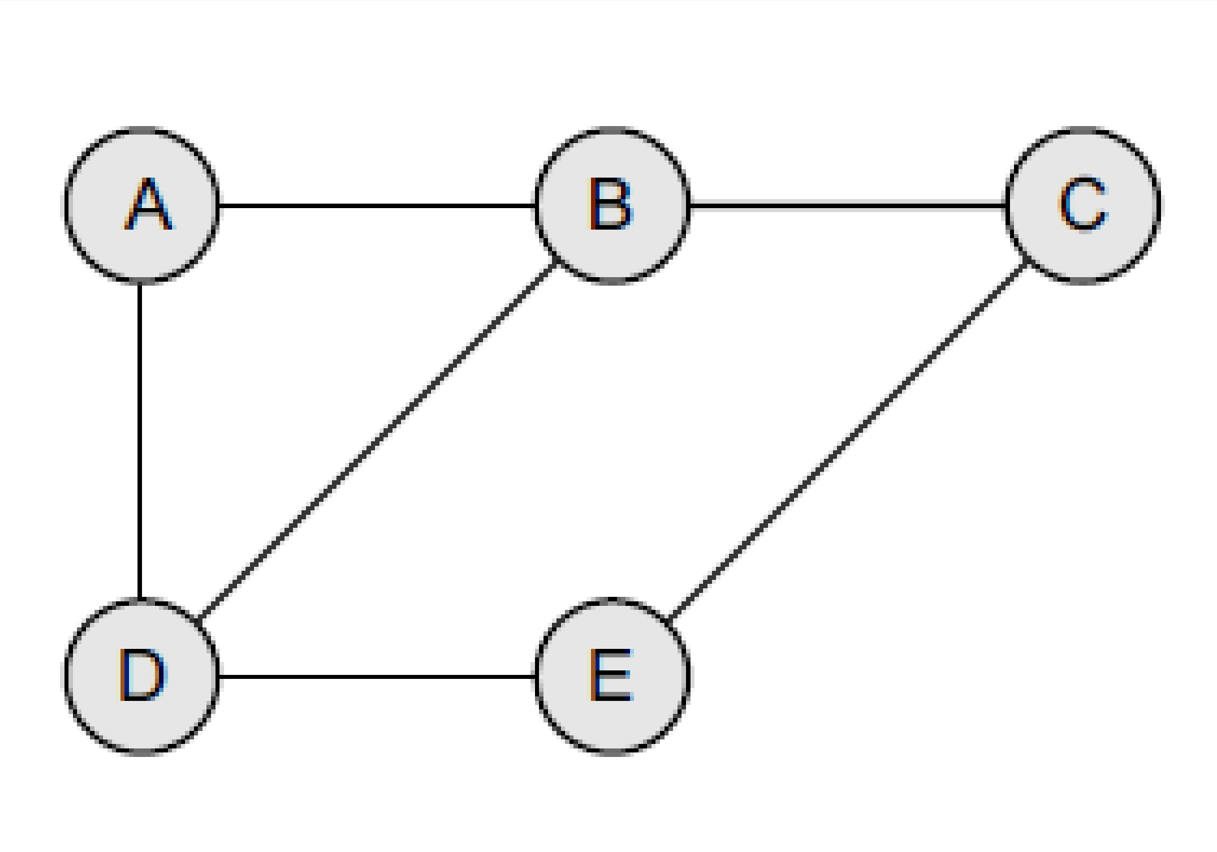
GRAPHS

- A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.
- In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions

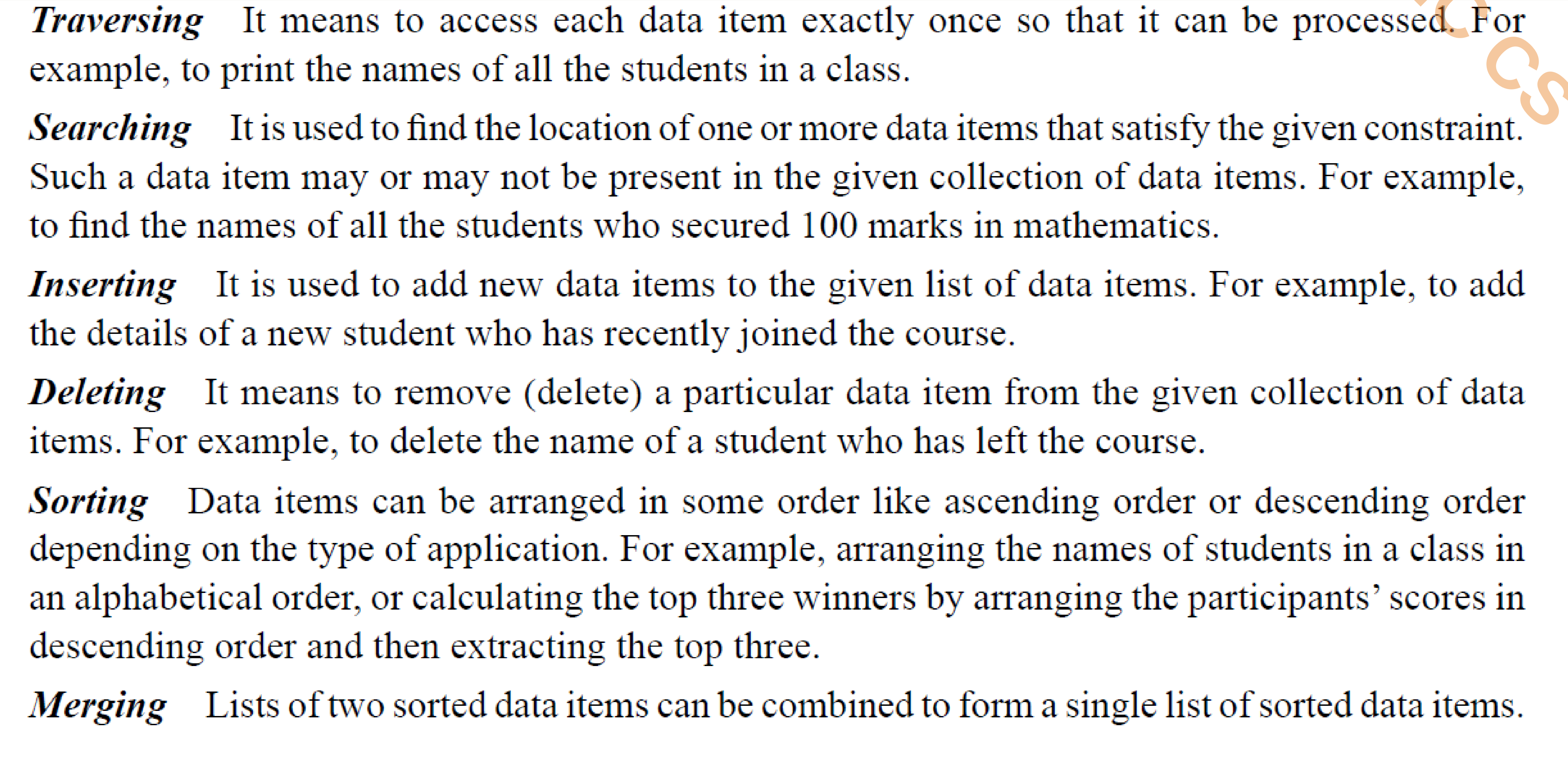
- A node in the graph may represent a city and the edges connecting the nodes can represent roads.
- A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.
- Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

- Note that unlike trees, graphs do not have any root node.
- Rather, every node in the graph can be connected with every another node in the graph.
- When two nodes are connected via an edge, the two nodes are known as *neighbours.*

- *Advantage*: Best models real-world situations
- *Disadvantage*: Some algorithms are slow and very complex


A
B
C
D
E

OPERATIONS ON DATA STRUCTURES



Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

Thank you

INTRODUCTION TO DATA STRUCTURES- PART3

Prepared by
Deepthi M Pisharody
Asst Professor
Prajyoti Niketan College,
Pudukad

ABSTRACT DATA TYPE

- ✓An *abstract data type* (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- ✓ For example, stacks and queues are perfect examples of an ADT.
- ✓We can implement both these ADTs using an array or a linked list.
- ✓This demonstrates the ‘abstract’ nature of stacks and queues.
- ✓To further understand the meaning of an abstract data type, we will break the term into ‘data type’ and ‘abstract’, and then discuss their meanings.

DATA TYPE

- Data type of a variable is the set of values that the variable can take.
- We have already read the basic data types in C include `int`, `char`, `float`, and `double`.
- When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data.
- For example, an `int` variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators `+`, `–`, `*`, and `/`. In other words, the operations that can be performed on a data type are an inseparable part of its identity.
- Therefore, when we declare a variable of an abstract data type (e.g., stack or a queue), we also need to specify the operations that can be performed on it.

ABSTRACT

- The word ‘abstract’ in the context of data structures means *considered apart from the detailed specifications or implementation*.
- In C, an abstract data type can be a structure considered without regard to its implementation.
- It can be thought of as a ‘description’ of the data in the structure with a list of operations that can be performed on the data within that structure.
- The end-user is not concerned about the details of how the methods carry out their tasks.
- They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

- For example, when we use a stack or a queue, the user is concerned only with the type of data and the operations that can be performed on it.
- Therefore, the fundamentals of how the data is stored should be invisible to the user.
- They should not be concerned with how the methods work or what structures are being used to store the data.
- They should just know that to work with stacks, they have `push()` and `pop()` functions available to them.
- Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

ADVANTAGE OF USING ADTS

- In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures.
- For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.
- In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

Thank you

Prepared by

Deepthi M Pisharody

Asst Professor

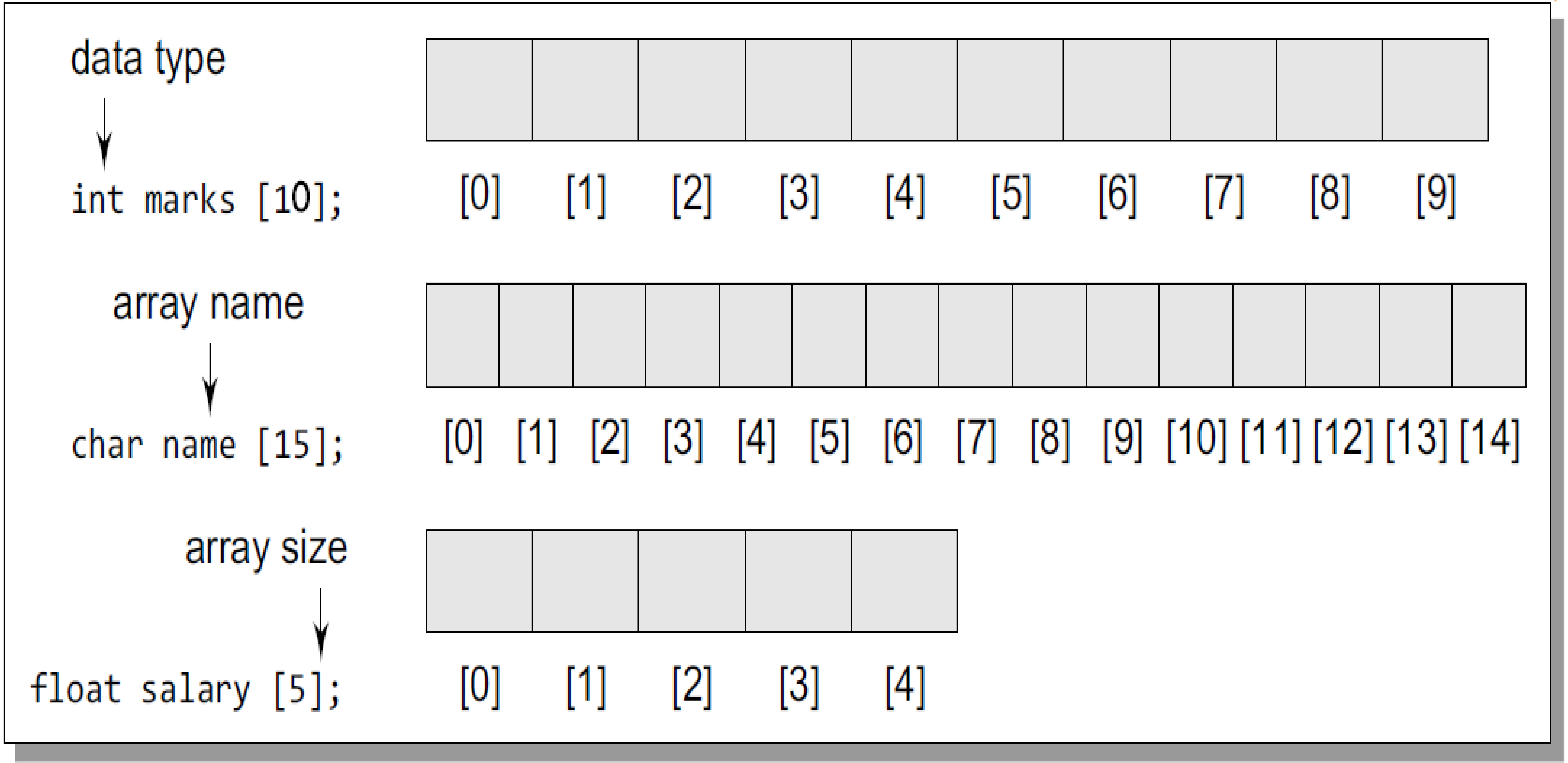
Prajyoti Niketan College, Pudukad

Arrays

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an `index` (also known as the *subscript*).
- The subscript is an ordinal number which is used to identify an element of the array.

DECLARATION OF ARRAYS

- An array must be declared before being used. Declaring an array means specifying the following:
- *Data type*—the kind of values it can store, for example, `int`, `char`, `float`, `double`.
- *Name*—to identify the array.
- *Size*—the maximum number of values that the array can hold.
- Arrays are declared using the following syntax:
- `type name[size];`
- `int marks[10];`



```
int marks [10];
char name [15];
float salary [5];
```

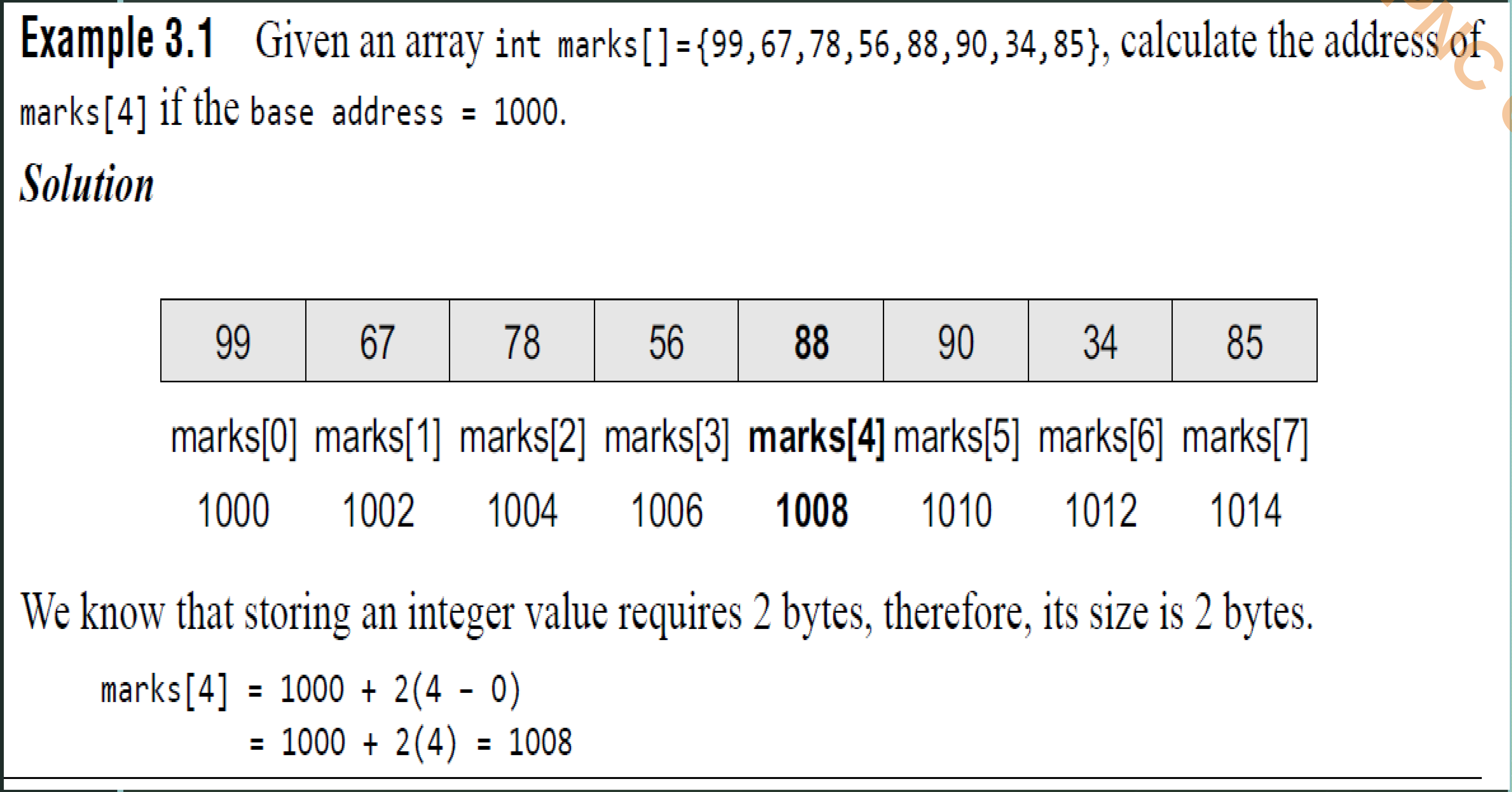
Figure 3.3 Declaring arrays of different data types and sizes

ACCESSING THE ELEMENTS OF AN ARRAY

- To access all the elements, we must use a loop.
- That is, we can access all the elements of an array by varying the value of the subscript into the array.
- There is no single statement that can read, access, or print all the elements of an array.
- To do this, we have to use a loop to execute the same statement with different index values.

Calculating the Address of Array Elements

- `Address of data element, A[k] = BA(A) + w(k – lower_bound)`
- `A` is the array, `k` is the index of the element of which we have to calculate the address, `BA` is the base address of the array `A`, and `w` is the size of one element in memory, for example, size of `int` is 2.



Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

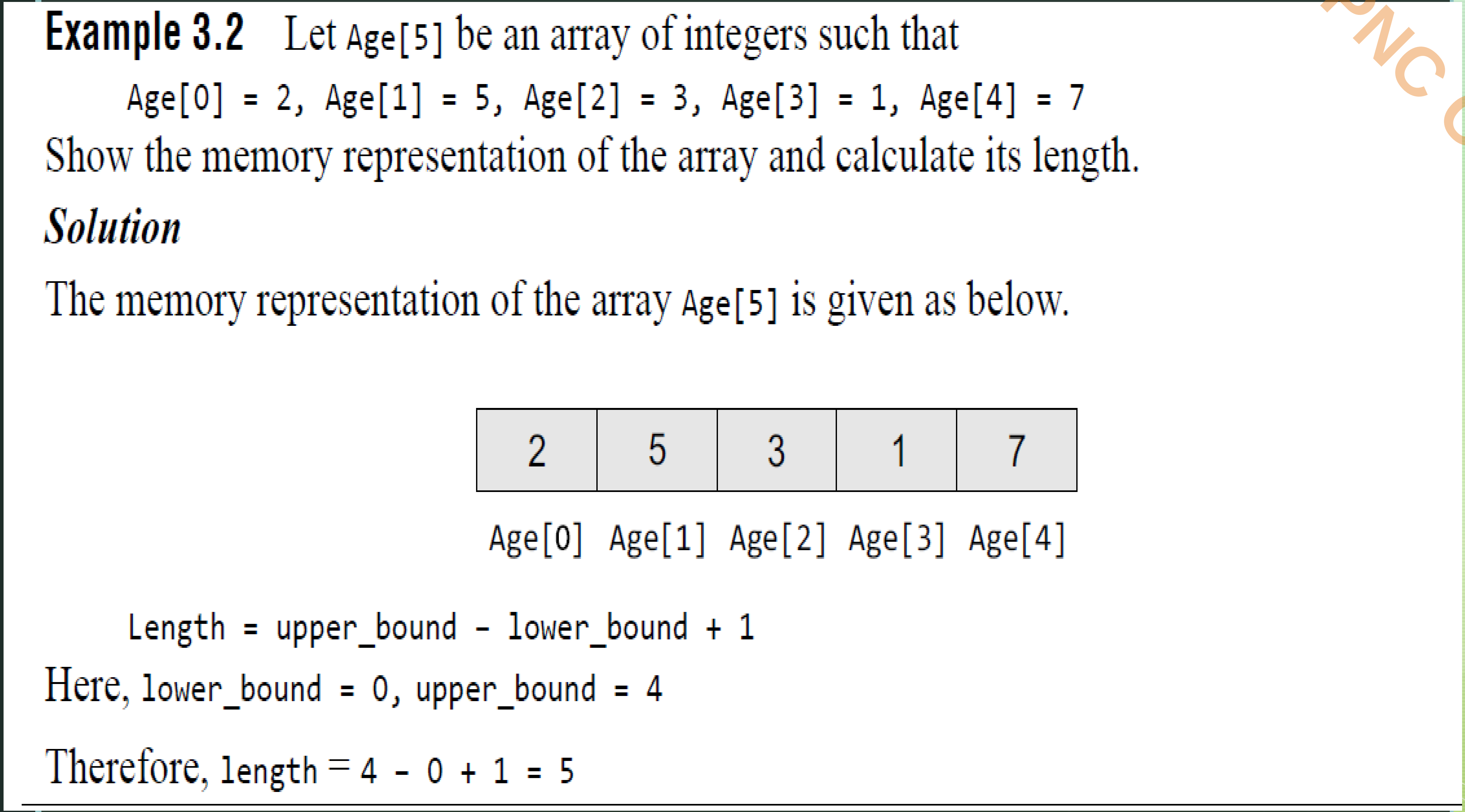
99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
    marks[4] = 1000 + 2(4 - 0)
             = 1000 + 2(4) = 1008
```

Calculating the Length of an Array

- The length of an array is given by the number of elements stored in it.
- The general formula to calculate the length of an array is
- `Length = upper_bound – lower_bound + 1`
- where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.



Example 3.2 Let `Age[5]` be an array of integers such that

```
    Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
```

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

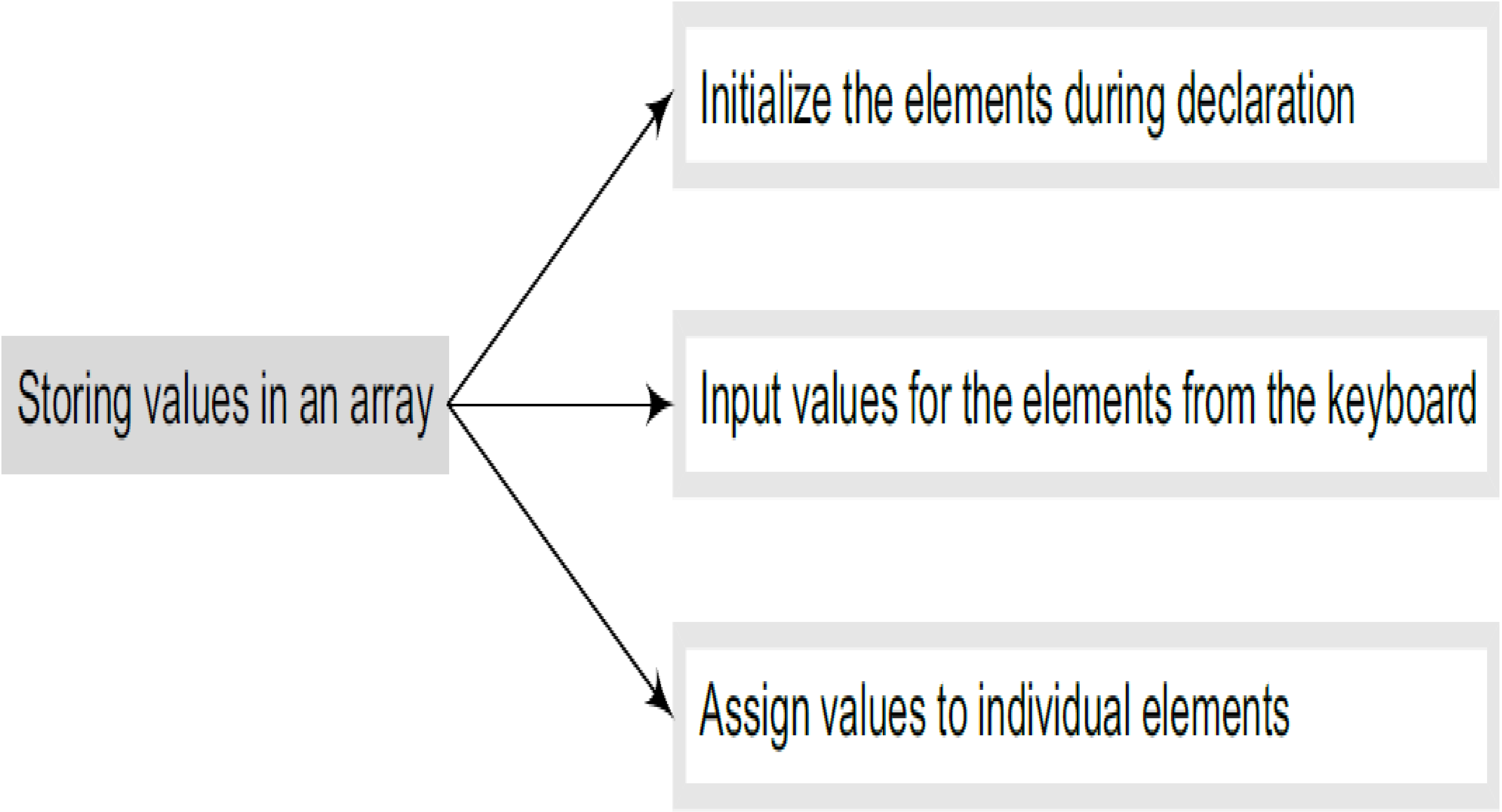
```
    Length = upper_bound - lower_bound + 1
```

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length` = `4 - 0 + 1 = 5`

STORING VALuES IN ARRAYS

- When we declare an array, we are just allocating space for its elements; no values are stored in the array.
- There are three ways to store values in an array.
- First, to initialize the array elements during declaration;
- second, to input values for individual elements from the keyboard;
- third, to assign values to individual elements



Storing values in an array

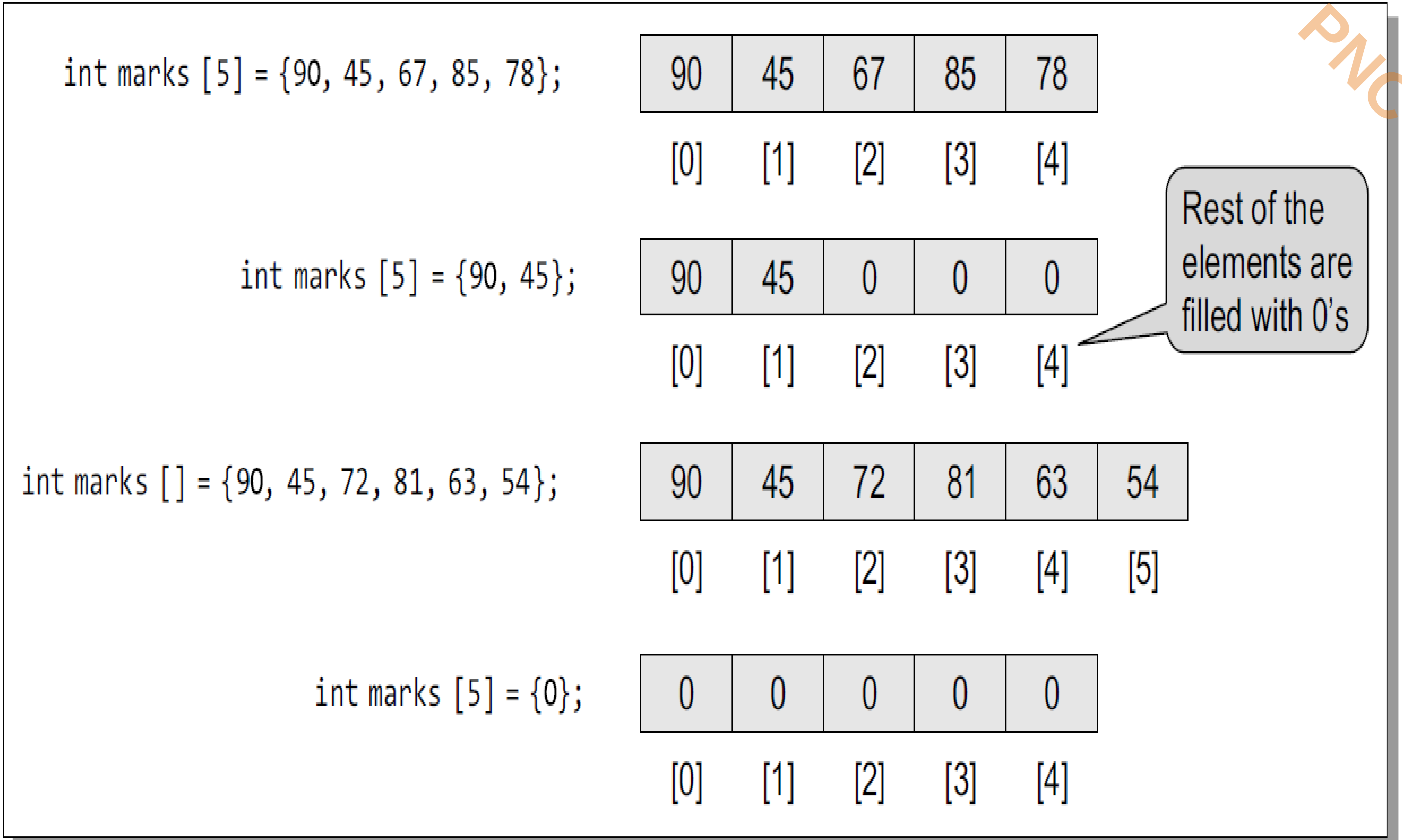
Initialize the elements during declaration

Input values for the elements from the keyboard

Assign values to individual elements

Initializing Arrays during Declaration

- When an array is initialized, we need to provide a value for every element in the array.
- Arrays are initialized by writing,
- `type array_name[size]={list of values};`
- The values are written within curly brackets and every value is separated by a comma.
- It is a compiler error to specify more values than there are elements in the array. When we write,
- `int marks[5]={90, 82, 78, 95, 88};`



```
int marks [5] = {90, 45, 67, 85, 78};
```

90	45	67	85	78
[0]	[1]	[2]	[3]	[4]

```
int marks [5] = {90, 45};
```

90	45	0	0	0
[0]	[1]	[2]	[3]	[4]

Rest of the elements are filled with 0's

```
int marks [] = {90, 45, 72, 81, 63, 54};
```

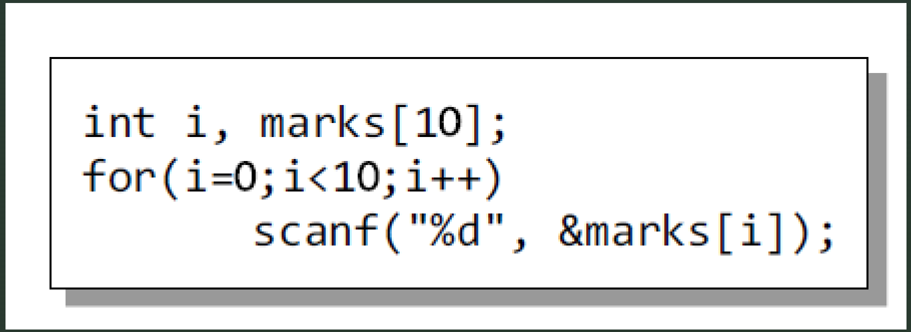
90	45	72	81	63	54
[0]	[1]	[2]	[3]	[4]	[5]

```
int marks [5] = {0};
```

0	0	0	0	0
[0]	[1]	[2]	[3]	[4]

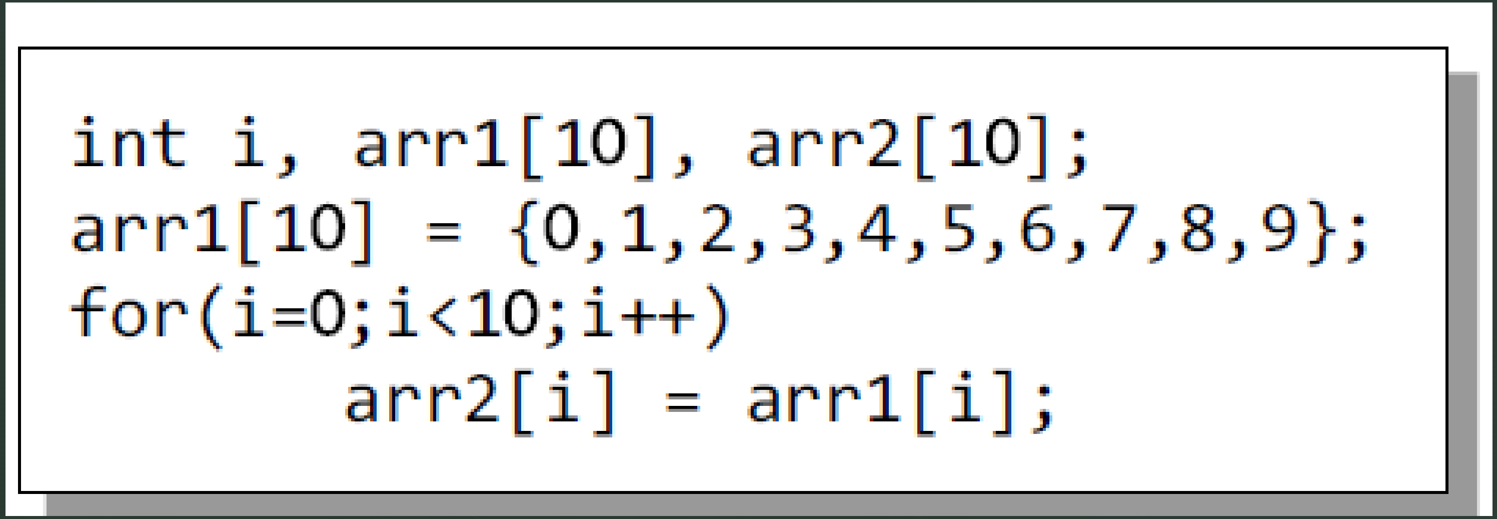
Inputting Values from the Keyboard

- An array can be initialized by inputting values from the keyboard.
- In this method, a `while/do-while` or a `for` loop is executed to input the value for each element of the array.

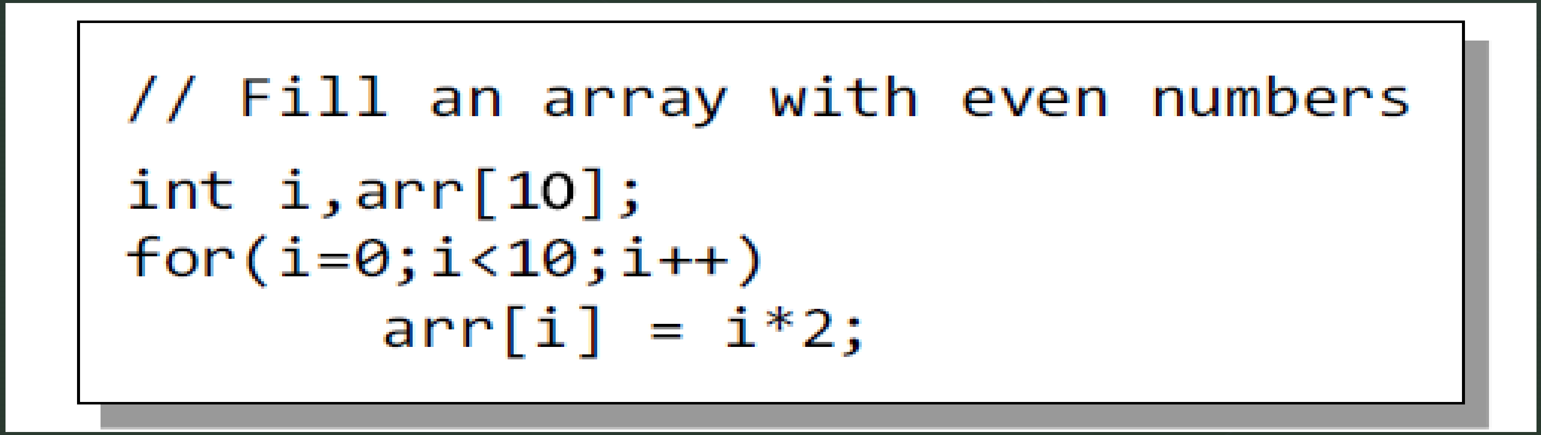


```
int i, marks[10];
for(i=0;i<10;i++)
        scanf("%d", &marks[i]);
```

Assigning Values to Individual Elements



```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
       arr2[i] = arr1[i];
```



```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
       arr[i] = i*2;
```

Thank you

PREPARED BY

DEEPTHI M PISHARODY

ASST PROFESSOR

PRAJYOTI NIKETAN COLLEGE, PUDUKAD

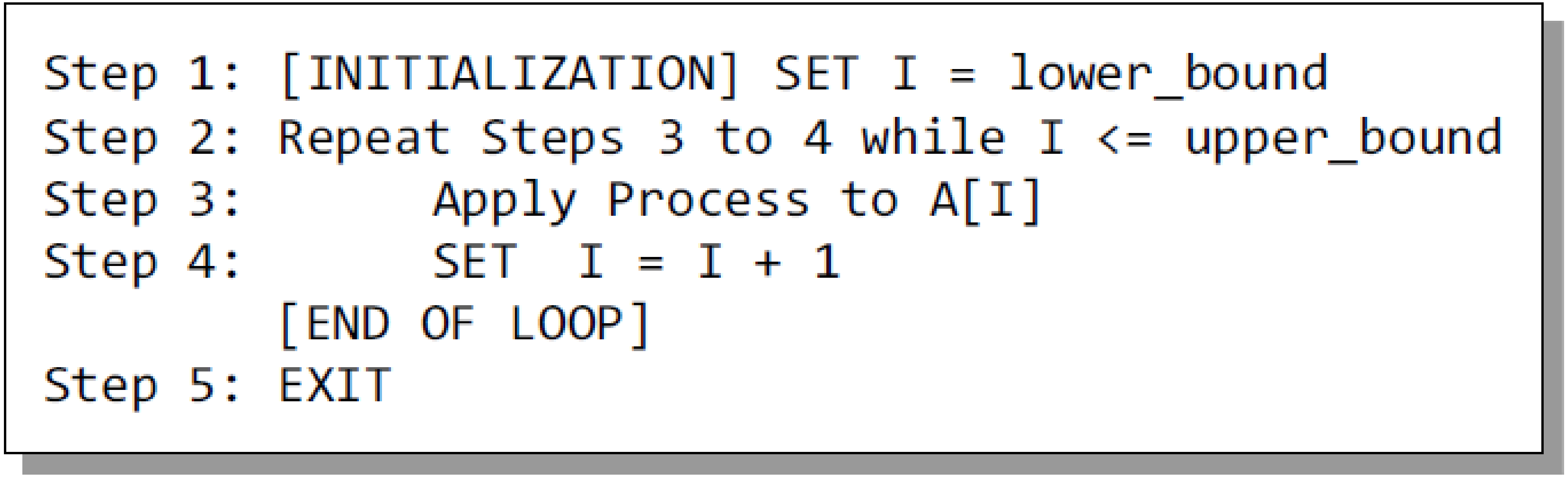
Array Operations

OPERATIONS ON ARRAYS

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Traversing an Array

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements.
- Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward.

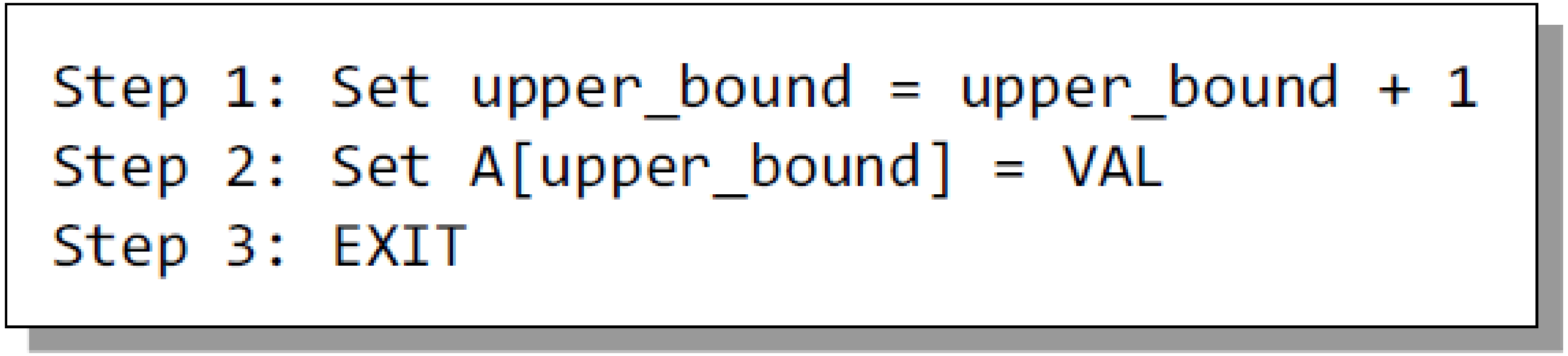


```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:         Apply Process to A[I]
Step 4:         SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

Inserting an Element in an Array

- If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple.
- We just have to add 1 to the `upper_bound` and assign the value. Here, we assume that the memory space allocated for the array is still available.
- For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.



```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

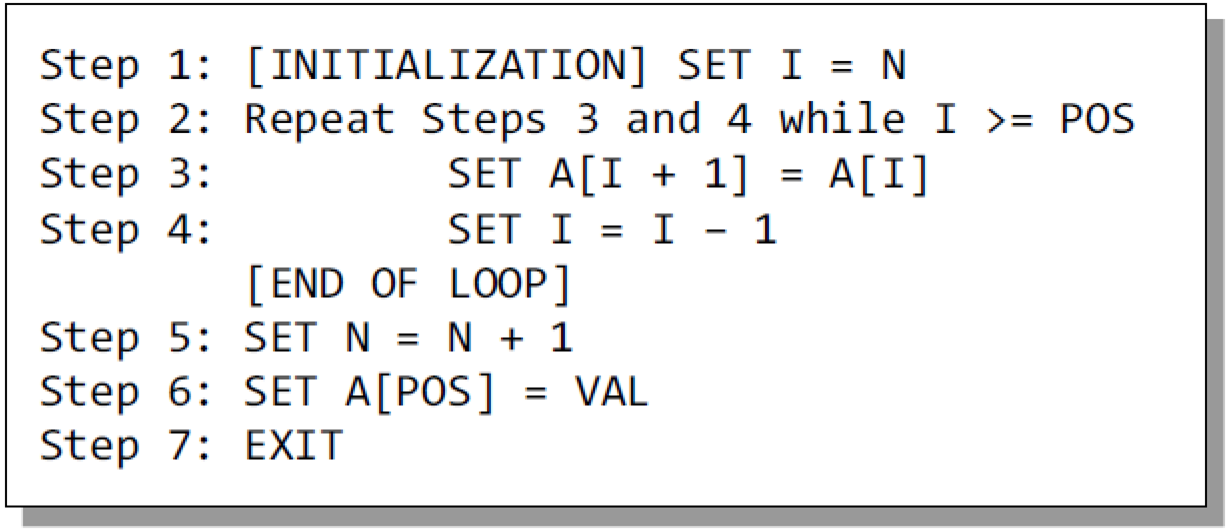
Figure 3.13 Algorithm to append a new element to an existing array

- If we have to insert an element in the middle of the array, then this is not a trivial task.
- On an average, we might have to move as much as half of the elements from their positions in order to accommodate space for the new element.

Algorithm to Insert an Element in the Middle of an Array

The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted
(b) N, the number of elements in the array
(c) POS, the position at which the element has to be inserted
(d) VAL, the value that has to be inserted



```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:         SET A[I + 1] = A[I]
Step 4:         SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

Initial `Data[]` is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling `INSERT(Data, 6, 3, 100)` will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

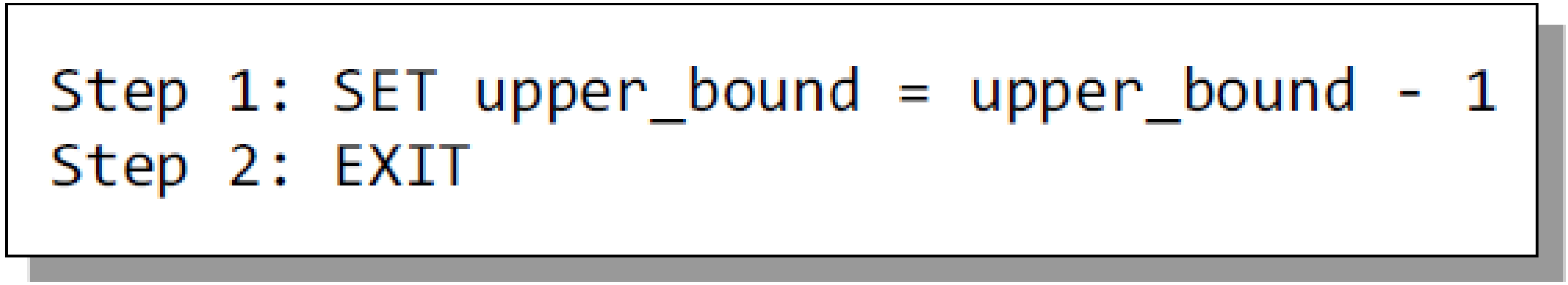
45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

Deleting an Element from an Array

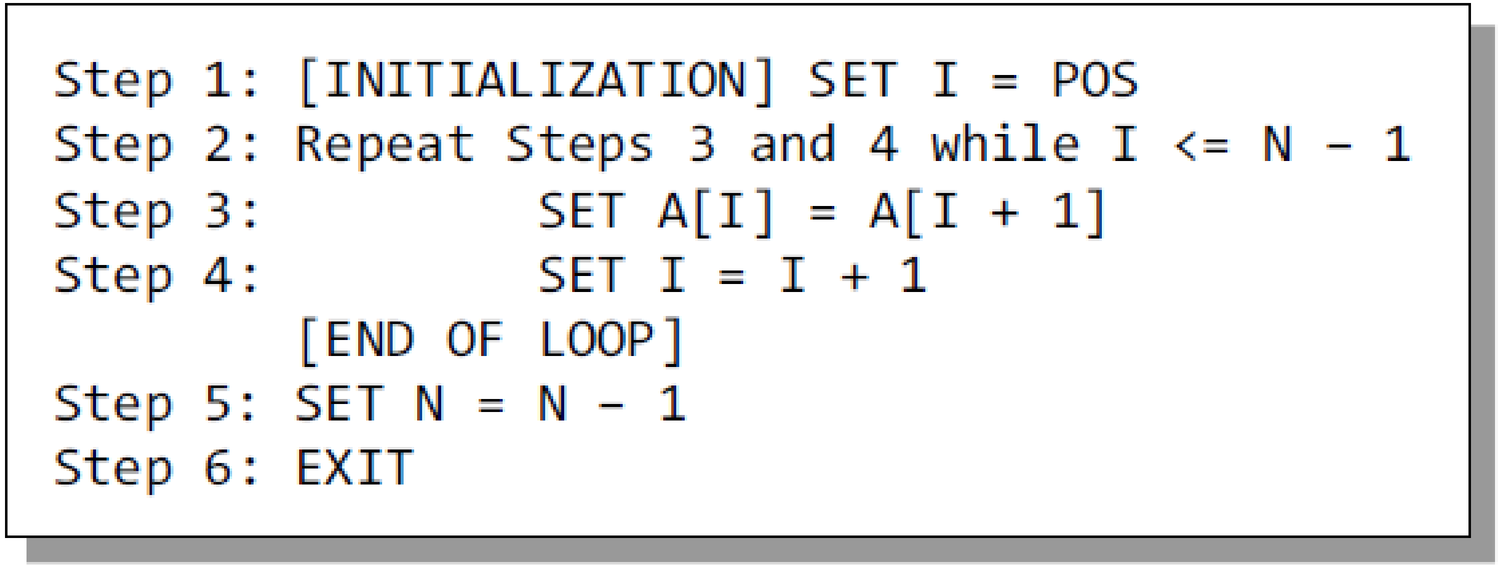
- Deleting an element from an array means removing a data element from an already existing array.
- If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple.
- We just have to subtract 1 from the `upper_bound`.



```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

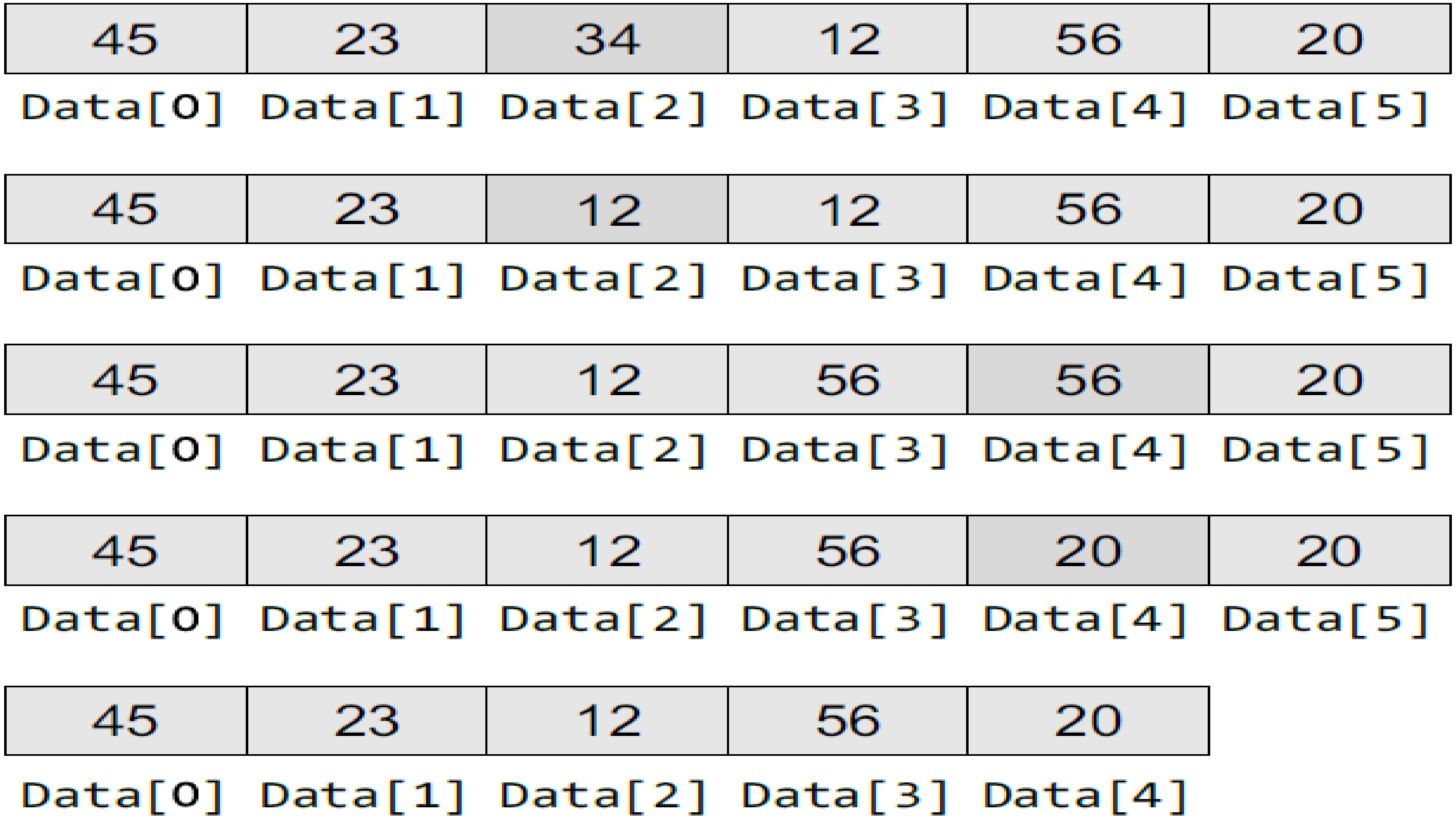
Figure 3.15 Algorithm to delete the last element of an array

- If we have to delete an element from the middle of an array, then it is not a trivial task.
- On an average, we might have to move as much as half of the elements from their positions in order to occupy the space of the deleted element.



```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:         SET A[I] = A[I + 1]
Step 4:         SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Figure 3.16 Algorithm to delete an element from the middle of an array



45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

Figure 3.17 Deleting elements from an array

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array.
- Hence, the merged array contains the contents of the first array followed by the contents of the second array.
- If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another.
- But merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted

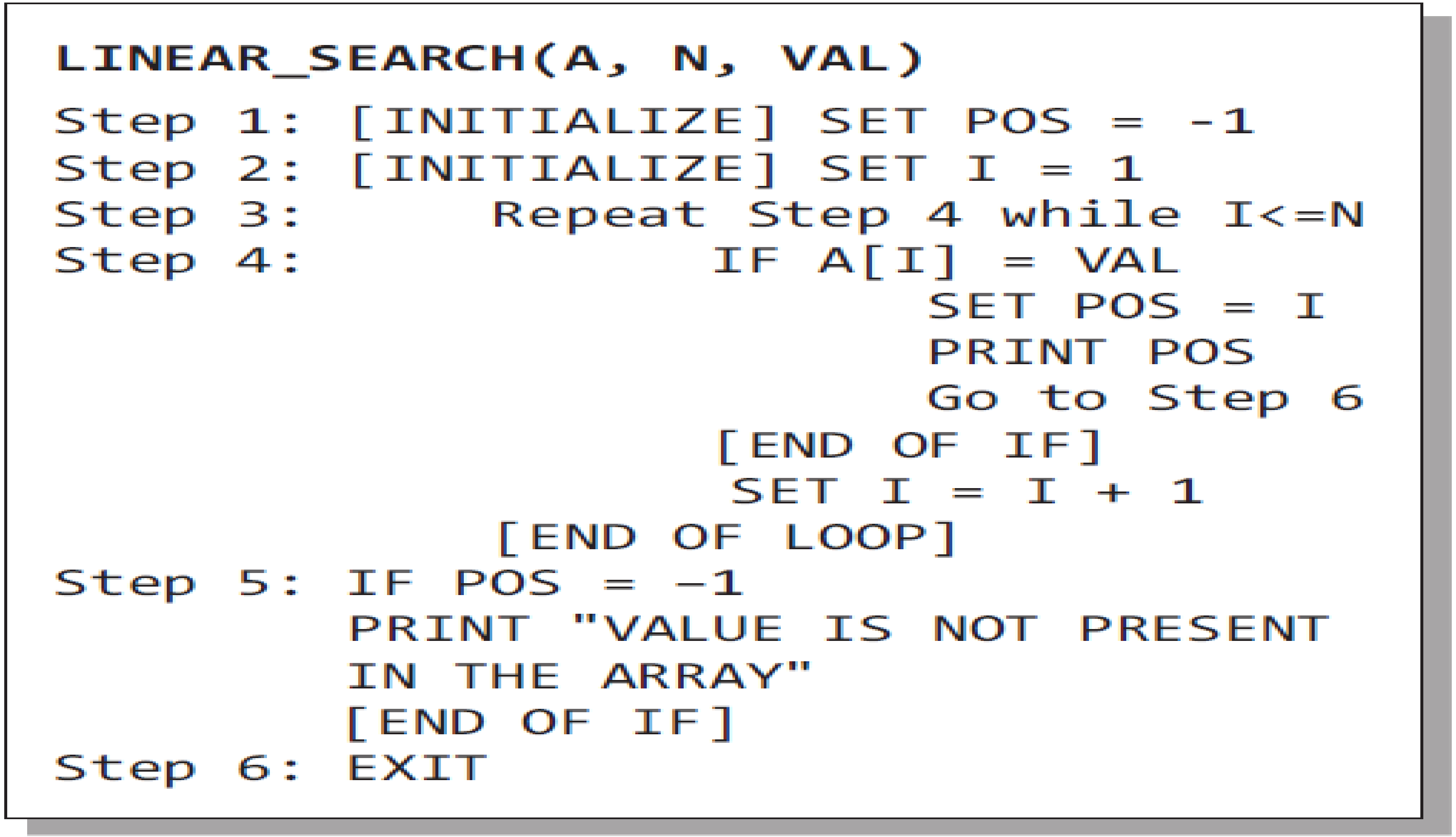
Searching

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- If the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- There are two popular methods for searching the array elements: *linear search* and *binary search.*
- The algorithm that should be used depends entirely on how the values are organized in the array.
- For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

Linear Search

- Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value.
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

- `int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};`
- and the value to be searched is `VAL = 7`, then searching means to find whether the value ‘7’ is present in the array or not.
- If yes, then it returns the position of its occurrence.
- Here, `POS = 3` (index starting from 0).



```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                          SET POS = I
                          PRINT POS
                          Go to Step 6
                    [END OF IF]
                    SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

Complexity of Linear Search Algorithm

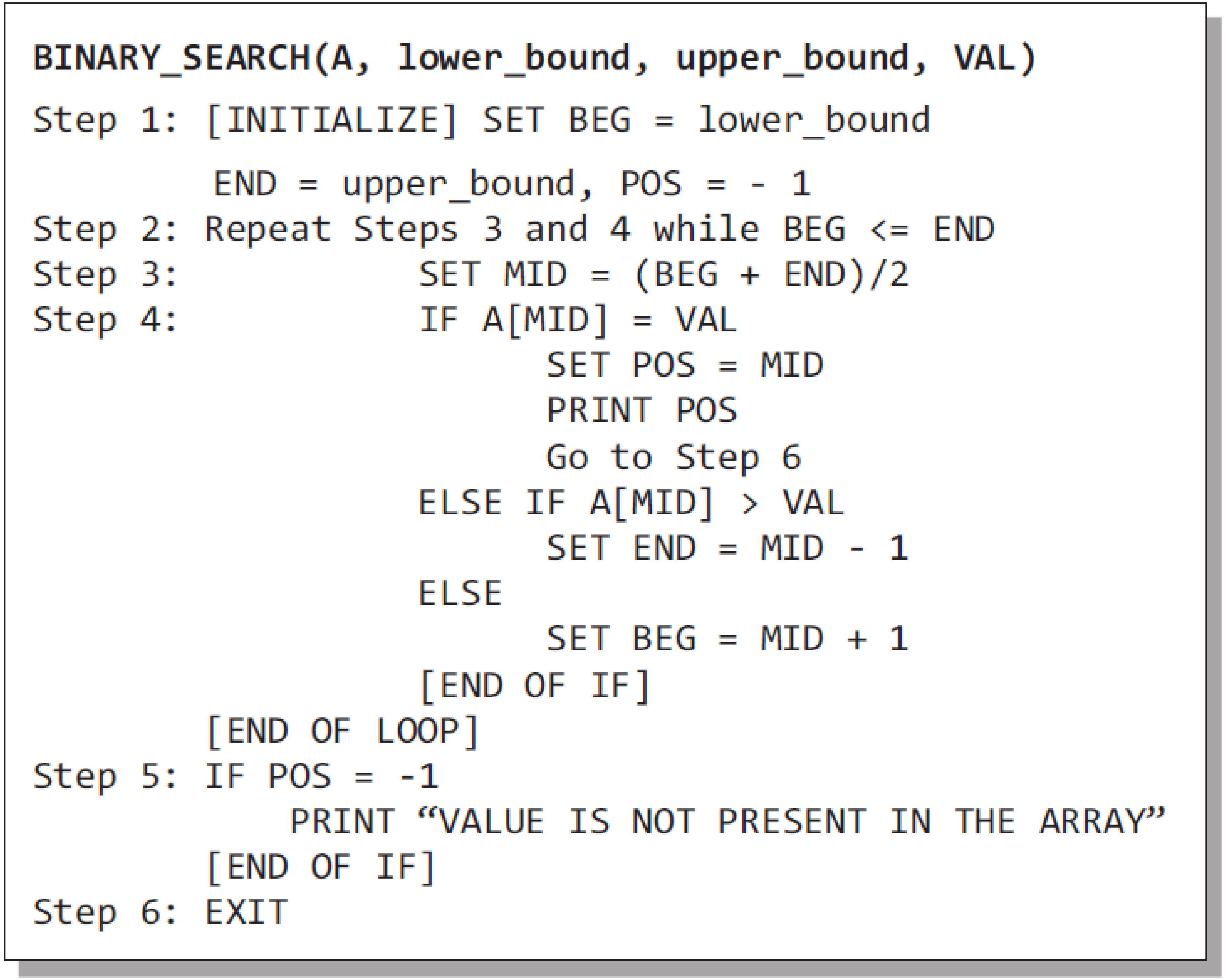
- Linear search executes in `O(n)` time where `n` is the number of elements in the array.
- Obviously, the best case of linear search is when `VAL` is equal to the first element of the array.
- In this case, only one comparison will be made.
- Likewise, the worst case will happen when either `VAL` is not present in the array or it is equal to the last element of the array. In both the cases, `n` comparisons will have to be made.
- However, the performance of the linear search algorithm can be improved by using a sorted array.

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.
- The mechanism of binary search can be better understood by an analogy of a telephone directory.
- When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory.
- Again, we open some page in the middle and the whole process is repeated until we finally find the right name

- We first open the dictionary somewhere in the middle.
- Then, we compare the first word on that page with the desired word whose meaning we are looking for.
- If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half.
- Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word.
- The same mechanism is applied in the binary search.

- Now, let us consider how this mechanism is applied to search for a value in a sorted array.
- Consider an array `A[]` that is declared and initialized as
- `int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`
- and the value to be searched is `VAL = 9.` The algorithm will proceed in the following manner.
- `BEG = 0, END = 10, MID = (0 + 10)/2 = 5`
- Now, `VAL = 9` and `A[MID] = A[5] = 5`
- `A[5]` is less than `VAL`, therefore, we now search for the value in the second half of the array. So,
- we change the values of `BEG` and `MID`.
- Now, `BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8`
- `VAL = 9 and A[MID] = A[8] = 8`
- `A[8]` is less than `VAL`, therefore, we now search for the value in the second half of the segment.
- So, again we change the values of `BEG` and `MID`.
- Now, `BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9`
- Now, `VAL = 9` and `A[MID] = 9`



```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Binary Search Algorithm

- $O(\log_2 n)$

sorting

Thank you

TWO-DIMENSIONAL ARRAYS

Prepared by
Deepthi M Pisharody
Asst Professor
Prajyoti Niketan College, Pudukad

- One-dimensional arrays are organized linearly in only one direction.
- But at times, we need to store data in the form of grids or tables.
- Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures.
- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column.
- The C compiler treats a two-dimensional array as an array of one-dimensional arrays.

Declaring Two-dimensional Arrays

- A two-dimensional array is declared as:
- `data_type array_name[row_size][column_size];`
- Therefore, a two-dimensional `m × n` array is an array that contains `m × n` data elements and each element is accessed using two subscripts, `i` and `j`, where `i <= m` and `j <= n`.
- `int marks[3][5];`

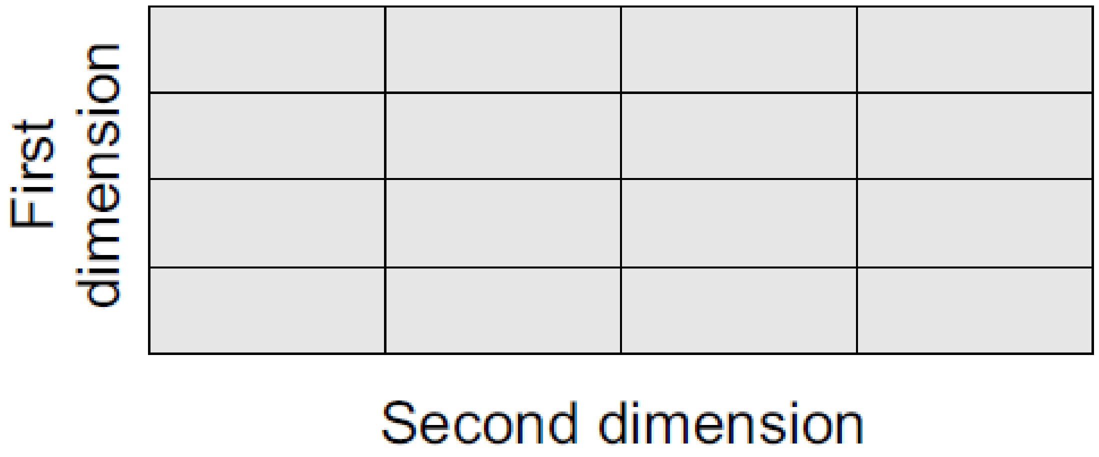
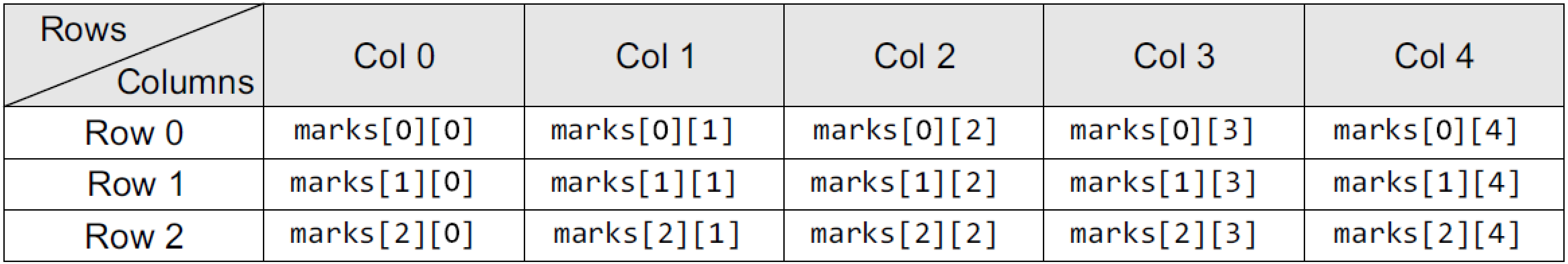


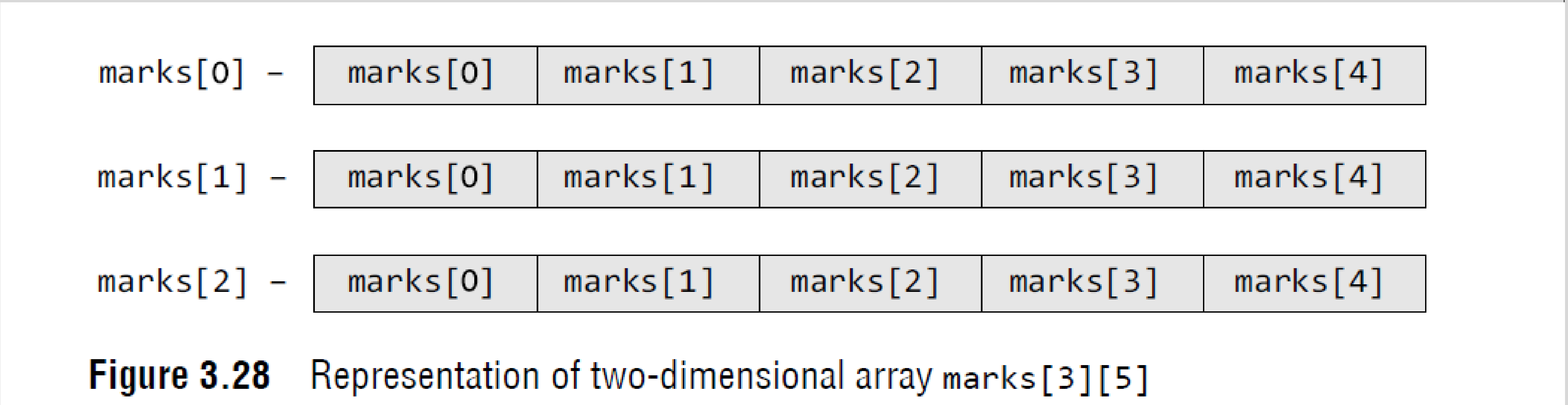
Figure 3.26 Two-dimensional array



Rows Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Figure 3.27 Two-dimensional array

- A 2D array is treated as a collection of 1D arrays.
- Each row of a 2D array corresponds to a 1D array consisting of n elements, where n is the number of columns.



marks[0] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]
marks[1] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]
marks[2] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

Figure 3.28 Representation of two-dimensional array `marks[3][5]`

- Although we have shown a rectangular picture of a two-dimensional array, in the memory, these elements actually will be stored sequentially.
- There are two ways of storing a two-dimensional array in the memory.
- The first way is the *row major order* and the second is the *column major order*.
- Let us see how the elements of a 2D array are stored in a row major order.
- Here, the elements of the first row are stored before the elements of the second and third rows.
- That is, the elements of the array are stored row by row where n elements of the first row will occupy the first n locations.

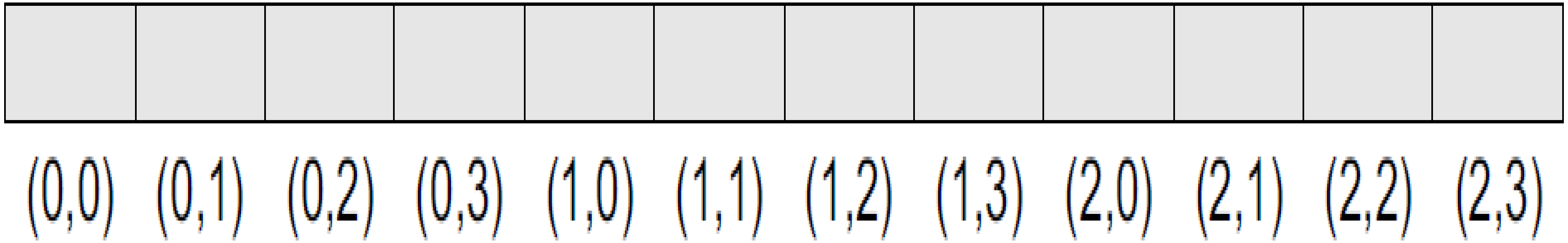


Figure 3.29 Elements of a 3 × 4 2D array in row major order

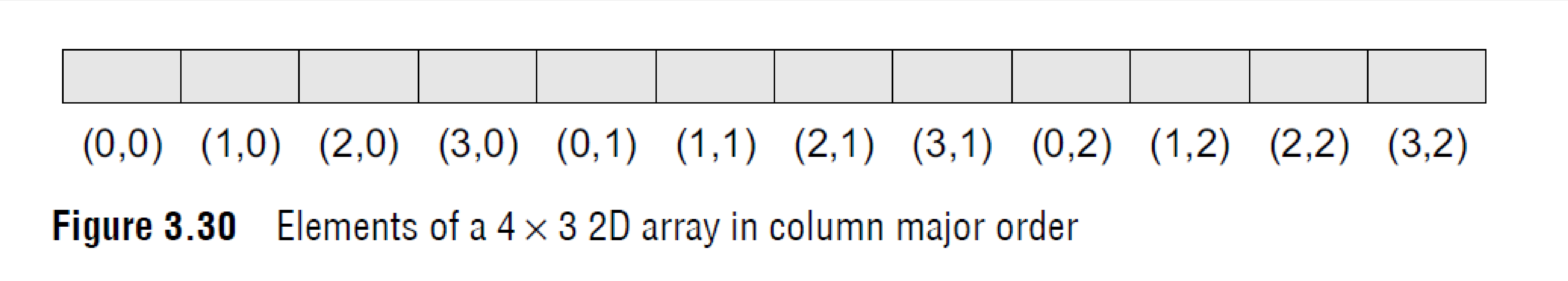
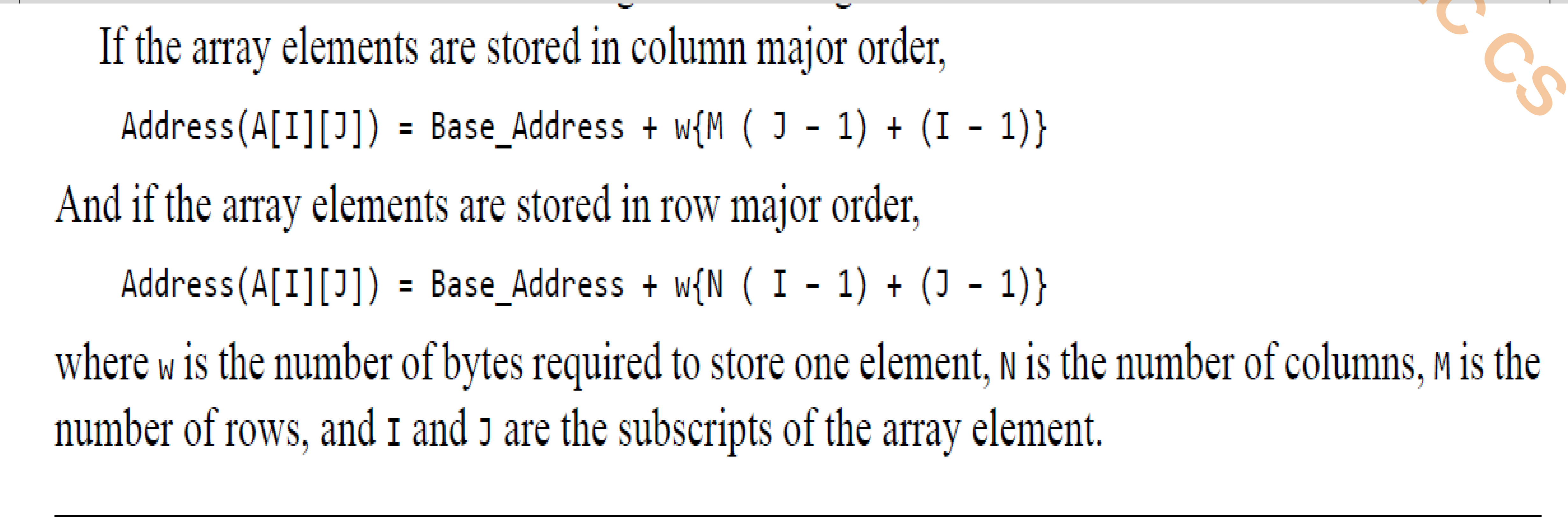


Figure 3.30 Elements of a 4 × 3 2D array in column major order



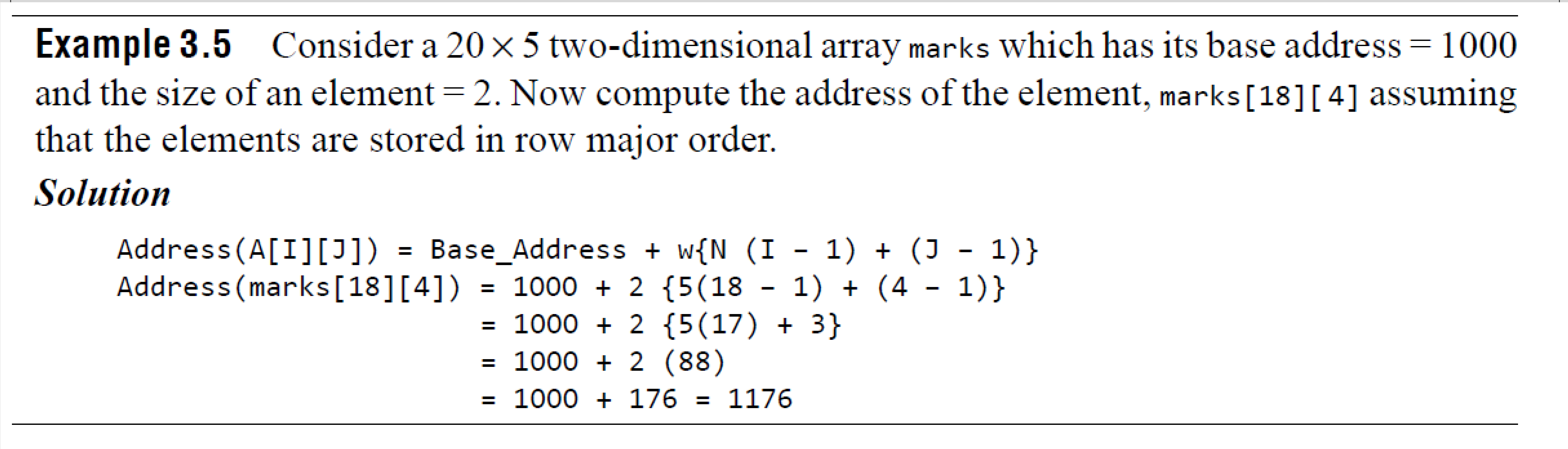
If the array elements are stored in column major order,

```
Address(A[I][J]) = Base_Address + w{M ( J - 1) + (I - 1)}
```

And if the array elements are stored in row major order,

```
Address(A[I][J]) = Base_Address + w{N ( I - 1) + (J - 1)}
```

where w is the number of bytes required to store one element, N is the number of columns, M is the number of rows, and I and J are the subscripts of the array element.



Example 3.5 Consider a 20 × 5 two-dimensional array `marks` which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J]) = Base_Address + w{N (I – 1) + (J – 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 – 1) + (4 – 1)}
                      = 1000 + 2 {5(17) + 3}
                      = 1000 + 2 (88)
                      = 1000 + 176 = 1176
```

Initializing Two-dimensional Arrays

- `int marks[2][3]={90, 87, 78, 68, 62, 71};`
- `int marks[2][3]={{90,87,78},{68, 62, 71}};`

Accessing the Elements of Two-dimensional Arrays

- 2D array are stored in contiguous memory locations.
- In case of one-dimensional arrays, we used a single `for` loop to vary the index `i` in every pass, so that all the elements could be scanned.
- Since the two-dimensional array contains two subscripts, we will use two `for` loops to scan the elements.
- The first `for` loop will scan each row in the 2D array and the second `for` loop will scan individual columns for every row in the array.

OPERATIONS ON TWO-DIMENSIONAL ARRAYS

- Two-dimensional arrays can be used to implement the mathematical concept of matrices.
- In mathematics, a matrix is a grid of numbers, arranged in rows and columns.

Transpose Transpose of an $m \times n$ matrix A is given as a $n \times m$ matrix B, where $B_{i,j} = A_{j,i}$.

Sum Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

Difference Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing:

$$C_{i,j} = A_{i,j} - B_{i,j}$$

Product Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, $m \times n$ matrix A can be multiplied with a $p \times q$ matrix B if $n=p$. The dimension of the product matrix is $m \times q$. The elements of two matrices can be multiplied by writing:

$$C_{i,j} = \Sigma A_{i,k} B_{k,j} \text{ for } k=1 \text{ to } n$$

MULTI-dimensional ARRAYS

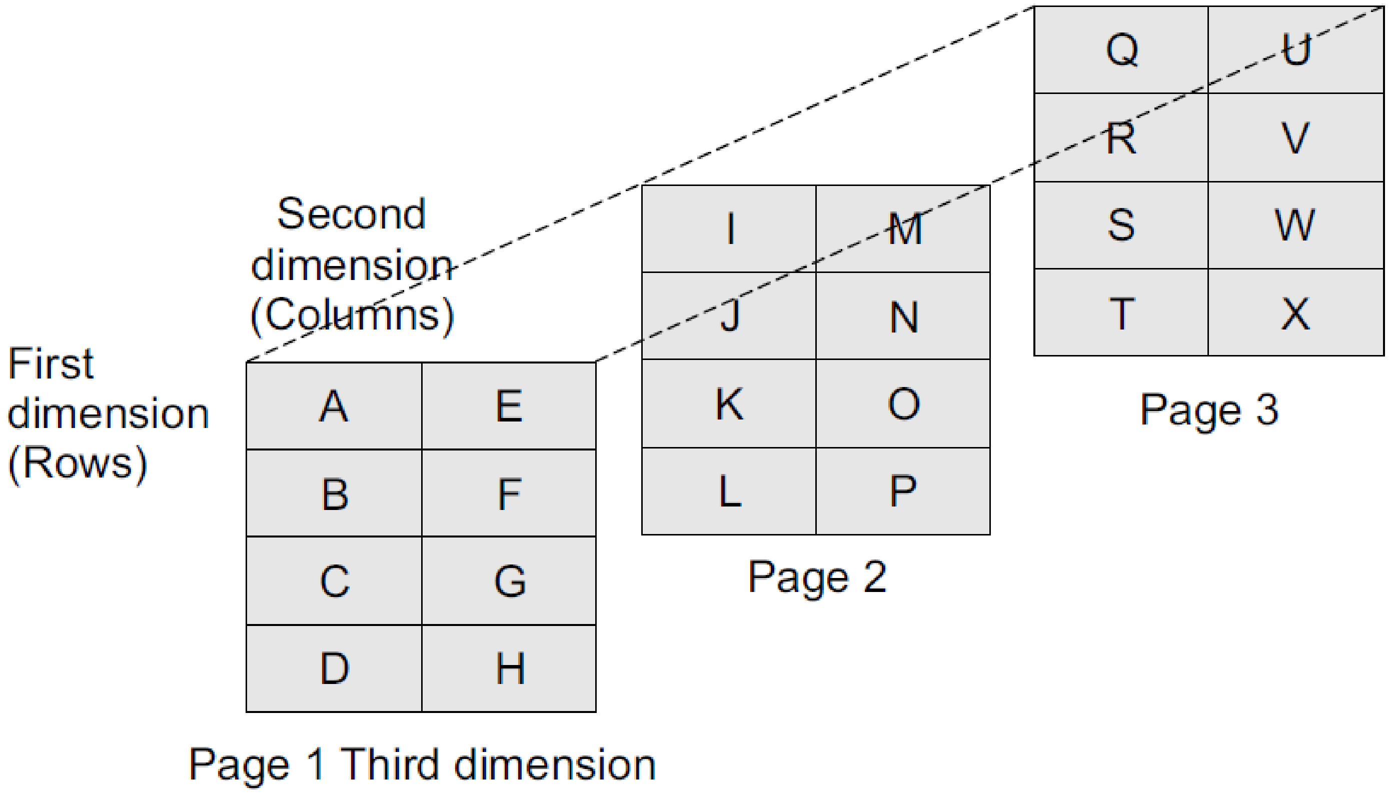


Figure 3.33 Three-dimensional array

- A multi-dimensional array is declared and initialized the same way we declare and initialize one- and two-dimensional arrays.
- A multi-dimensional array can contain as many indices as needed and as the requirement of memory increases with the number of indices used.

Thank you

SPARSE MATRIX

Prepared by

Deepthi M Pisharody

AssAssisstant Professor

Prajyoti Niketan College, Pudukad

- Sparse matrix is a matrix that has large number of elements with a zero value.
- In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used.
- If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory

- Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

There are two types of sparse matrices.

- In the first type of sparse matrix, all elements above the main diagonal have a zero value.
- This type of sparse matrix is also called a (*lower*) *triagonal matrix* because if you see it a lower triangular matrix, $A_{i,j} = 0$ where $i < j$.
- An $n×n$ lower-triangular matrix A has one non-zero element in the first row, two non-zero elements in the second row and likewise n non-zero elements in the *n*th row

$$\begin{bmatrix} 1 & & & & \\ 5 & 3 & & & \\ 2 & 7 & -1 & & \\ 3 & 1 & 4 & 2 & \\ -9 & 2 & -8 & 1 & 7 \end{bmatrix}$$

- To store a lower-triangular matrix efficiently in the memory, we can use a one-dimensional array which stores only non-zero elements.
- The mapping between a two-dimensional matrix and a one-dimensional array can be done
- in any one of the following ways:
- (a) Row-wise mapping—Here the contents of array `A[]` will be {1, 5,
- 3, 2, 7, –1, 3, 1, 4, 2, –9, 2, –8, 1, 7}
- (b) Column-wise mapping—Here the contents of array `A[]` will be
- {1, 5, 2, 3, –9, 3, 7, 1, 2, –1, 4, –8, 2, 1, 7}

upper-triangular matrix

- In an *upper-triangular matrix*, $A_{i,j} = 0$ where $i > j$.
- An $n×n$ upper-triangular matrix A has n non-zero elements in the first row, $n–1$ non-zero elements in the second row and likewise one non-zero element in the *n*th row.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 3 & 6 & 7 & 8 \\ & & -1 & 9 & 1 \\ & & & 9 & 2 \\ & & & & 7 \end{bmatrix}$$

tri-diagonal matrix.

- There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal.
- This type of matrix is also called a *tri-diagonal matrix*.
- Hence in a tridiagonal matrix, $A_{i,j} = 0$, where $|i - j| > 1$.
- In a tridiagonal matrix, if elements are present on
- (a) the main diagonal, it contains non-zero elements for $i=j$. In all, there will be n elements.
- (b) below the main diagonal, it contains non-zero elements for $i=j+1$. In all, there will be $n-1$ elements.
- (c) above the main diagonal, it contains non-zero elements for $i=j-1$. In all, there will be $n-1$ elements.

$$\begin{bmatrix} 4 & 1 & & & & \\ 5 & 1 & 2 & & & \\ & 9 & 3 & 1 & & \\ & & 4 & 2 & 2 & \\ & & & 5 & 1 & 9 \\ & & & & 8 & 7 \end{bmatrix}$$

- To store a tri-diagonal matrix efficiently in the memory,
- we can use a one-dimensional array that stores only non-zero elements.
- The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:
- (a) Row-wise mapping—Here the contents of array `A[ ]` will be
- {4, 1, 5, 1, 2, 9, 3, 1, 4, 2, 2, 5, 1, 9, 8, 7}
- (b) Column-wise mapping—Here the contents of array `A[ ]` will be
- {4, 5, 1, 1, 9, 2, 3, 4, 1, 2, 5, 2, 1, 8, 9, 7}
- (c) Diagonal-wise mapping—Here the contents of array `A[ ]` will be
- {5, 9, 4, 5, 8, 4, 1, 3, 2, 1, 7, 1, 2, 1, 2, 9}

Sparse Matrix Representation

- Array
- Linked List

Single Linear List Example

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

list =

$$\begin{array}{l|cccccc|}
\text{row} & 0 & 0 & 1 & 1 & 3 & 3 \\
\text{column} & 2 & 4 & 2 & 3 & 1 & 2 \\
\text{value} & 3 & 4 & 5 & 7 & 2 & 6
\end{array}$$

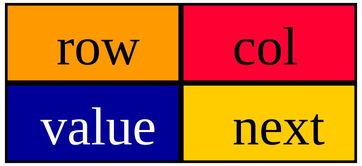
EXAMPLE:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

INDEX	ROW NO.	COL. NO.	VALUES
0	0	2	3
1	0	4	4
2	1	2	5
3	1	3	7
4	3	1	2
5	3	2	6

Chain Representation

Node structure.

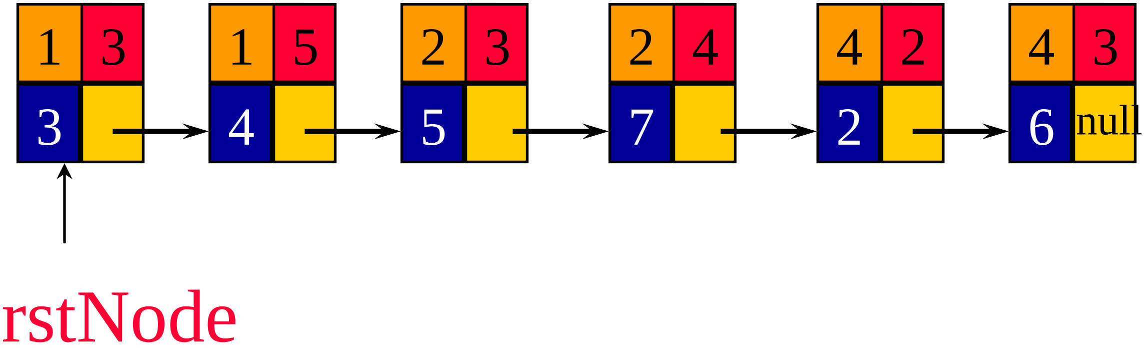


row col

value next

Single Chain

$$\text{list} = \begin{array}{r} \text{row} \\ \text{column} \\ \text{value} \end{array} \left[\begin{array}{cccccc} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{array}\right]$$



One Linear List Per Row

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

row1 = [(3, 3), (5,4)]

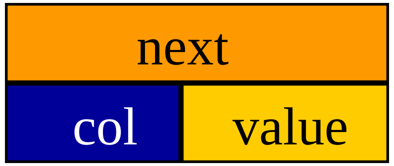
row2 = [(3,5), (4,7)]

row3 = []

row4 = [(2,2), (3,6)]

Array Of Row Chains

Node structure.

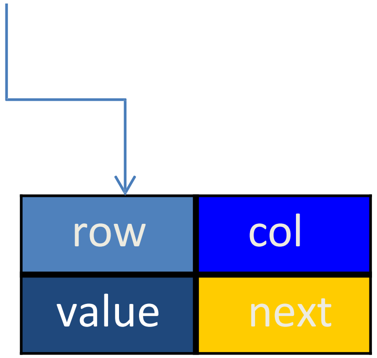


next

col value

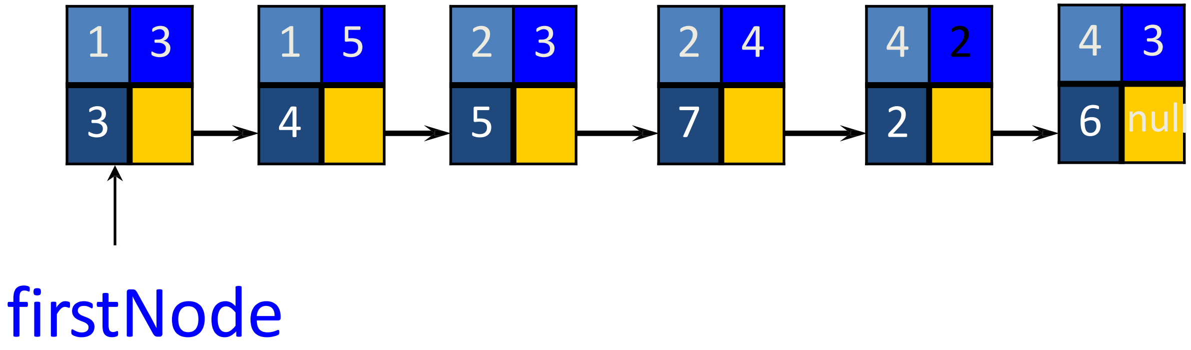
Chain Representation

Node structure:



Single Chain

$$\text{list} = \begin{array}{r|cccccc|} \text{row} & 1 & 1 & 2 & 2 & 4 & 4 \\ \text{column} & 3 & 5 & 3 & 4 & 2 & 3 \\ \text{value} & 3 & 4 & 5 & 7 & 2 & 6 \end{array}$$



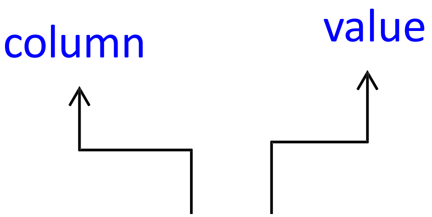
One Linear List Per Row

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



column

value

row1 = [(3, 3), (5,4)]

row2 = [(3,5), (4,7)]

row3 = []

row4 = [(2,2), (3,6)]

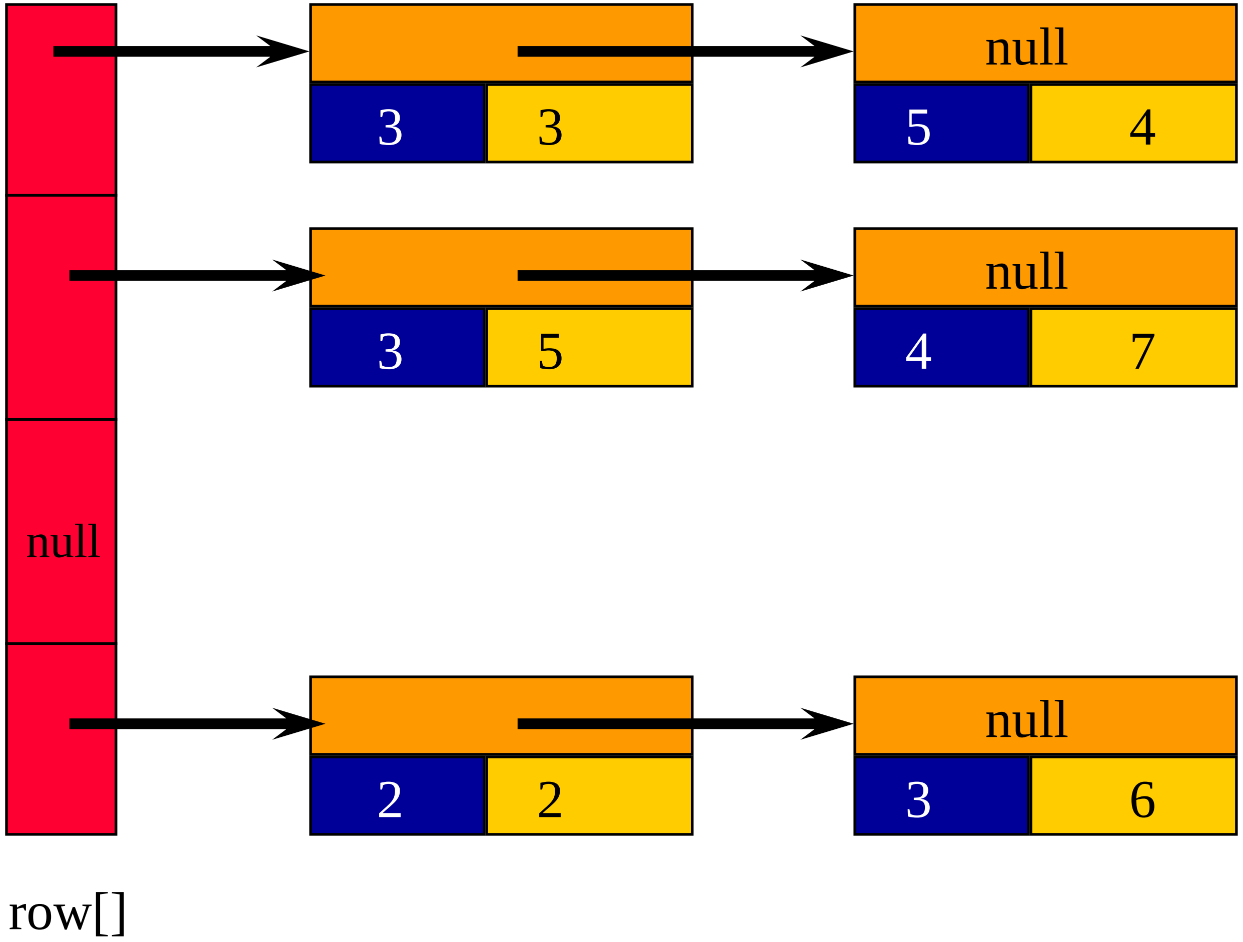
Array Of Row Chains

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



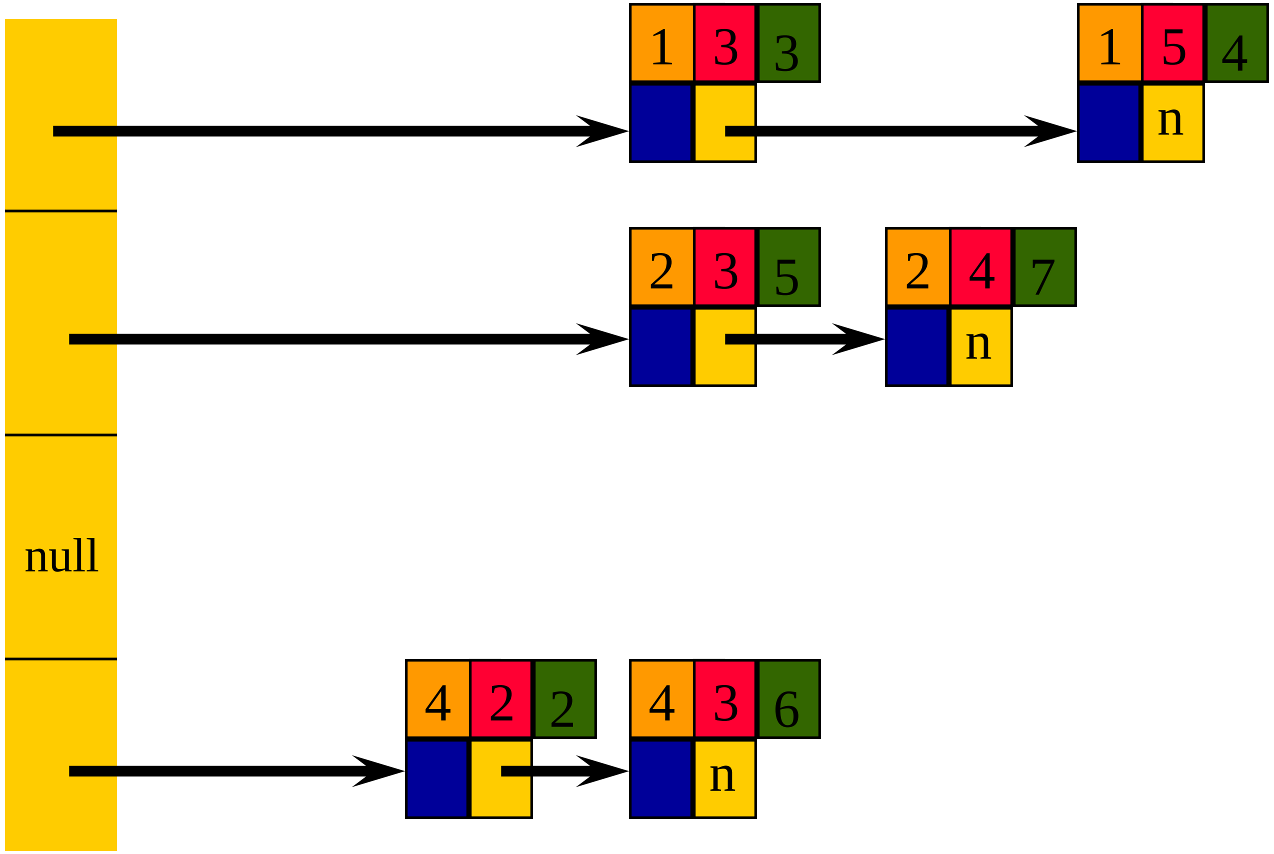
Row Lists

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



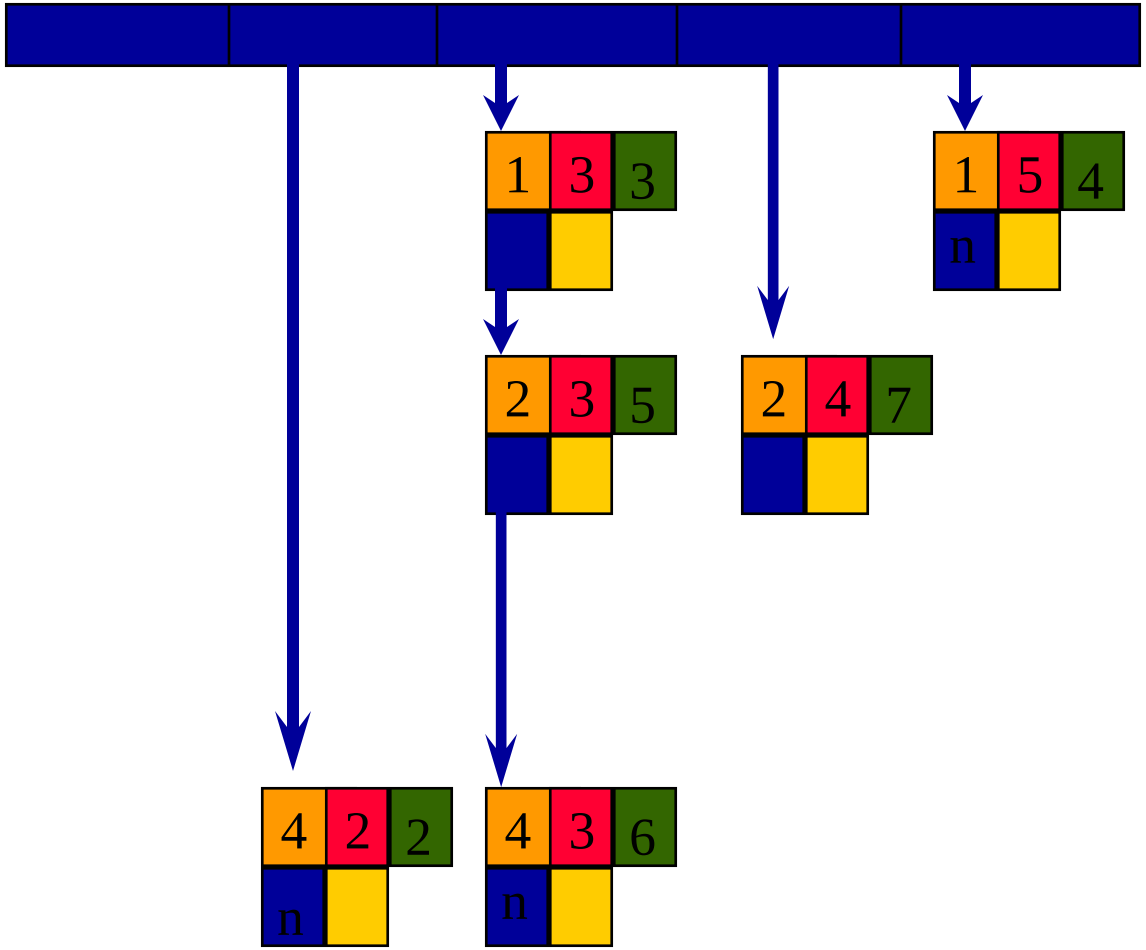
Column Lists

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



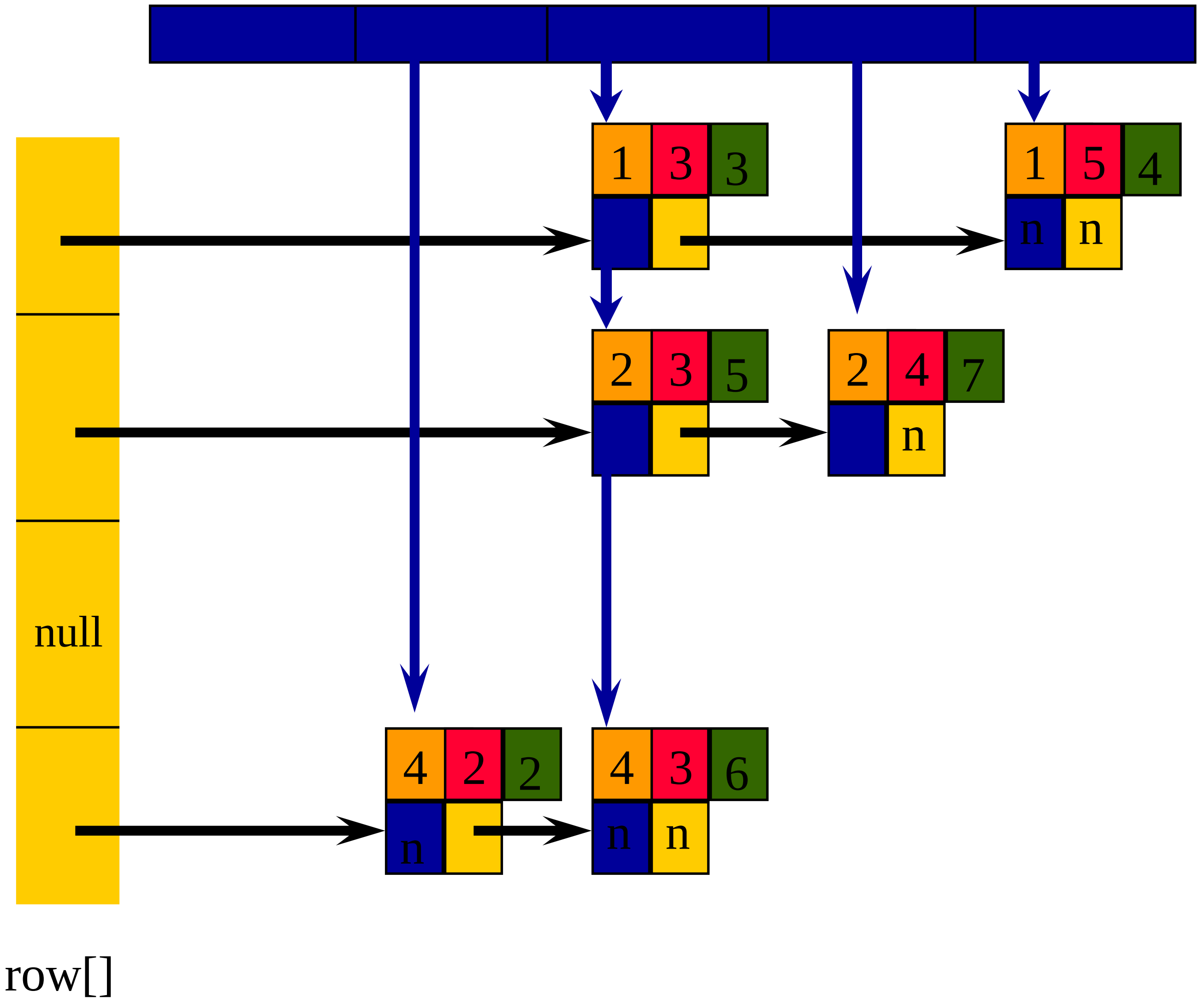
Orthogonal Lists

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



Thank You

LINKED LIST

Prepared by

Deepthi M Pisharody

Asst Professor

Prajyoti Niketan College, Pudukad

INTRODUCTION

- An array is a linear collection of data elements in which the elements are stored in consecutive memory locations.
- While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store.
- For example, if we declare an array as `int marks[10]`, then the array can store a maximum of 10 data elements but not more than that.
- But what if we are not sure of the number of elements in advance?
- Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations.
- So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

- Linked list is a data structure that is free from the aforementioned restrictions.
- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- However, unlike an array, a linked list does not allow random access of data.
- Elements in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.

- A linked list, in simple terms, is a linear collection of data elements.
- These data elements are called *nodes*.
- Linked list is a data structure which in turn can be used to implement other data structures.
- Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

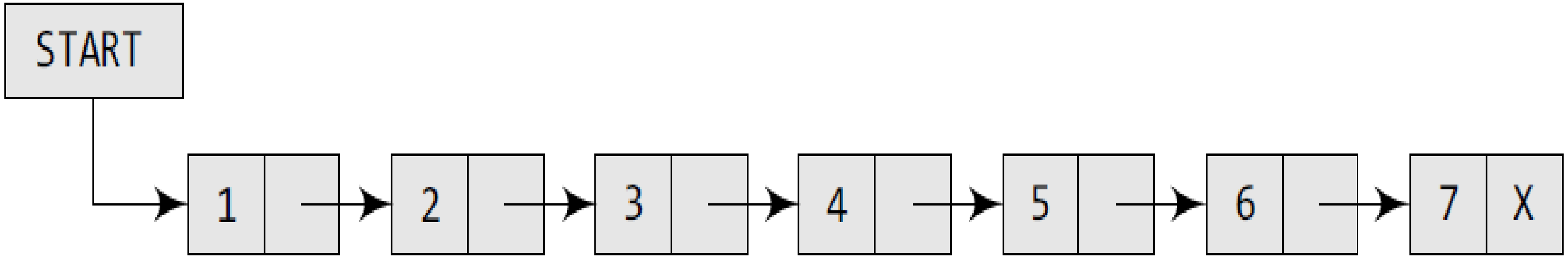


Figure 6.1 Simple linked list

- we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.
- The left part of the node which contains data may include a simple data type, an array, or a structure.
- The right part of the node contains a pointer to the next node (or address of the next node in sequence).
- The last node will have no next node connected to it, so it will store a special value called NULL

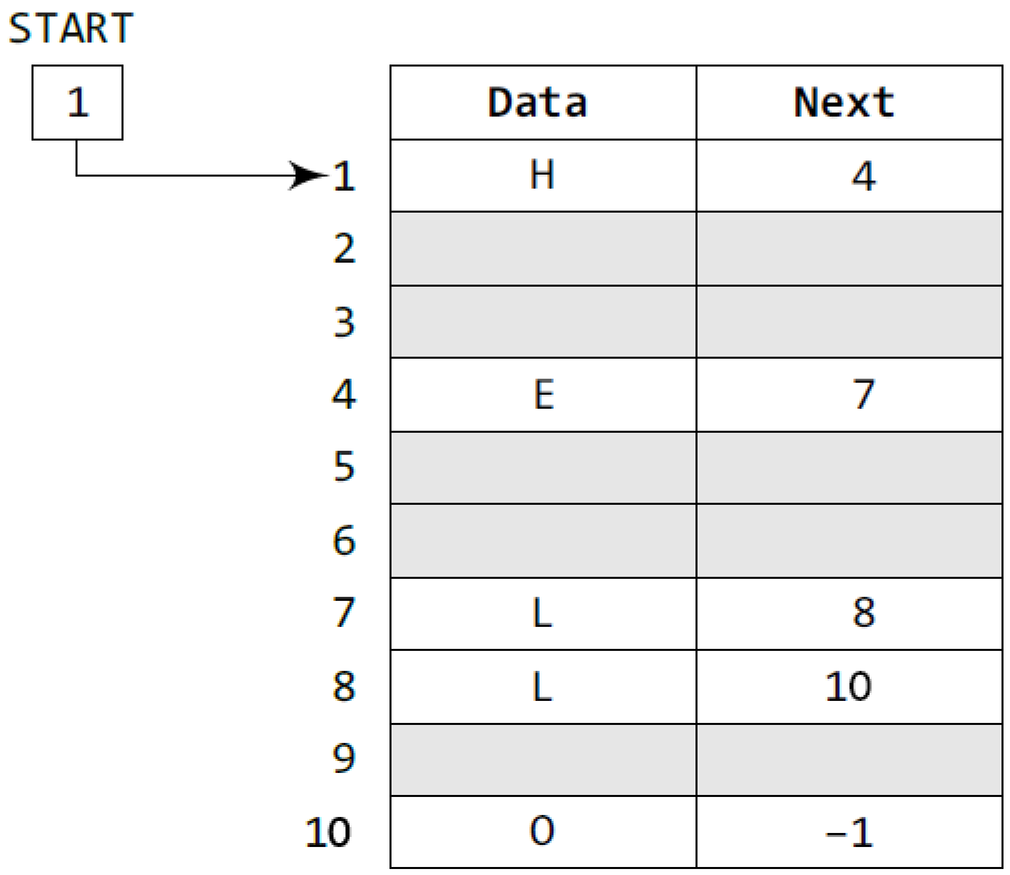
- the NULL pointer is represented by X.
- While programming, we usually define NULL as –1.
- Hence, a NULL pointer denotes the end of the list.
- Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

- Linked lists contain a pointer variable START that stores the address of the first node in the list.
- We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node.
- Using this technique, the individual nodes of the list will form a chain of nodes.
- If START = NULL, then the linked list is empty and contains no nodes.

- In C, we can implement a linked list using the following code:
- `struct node`

```
{
int data;
struct node *next;
};
```

- Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.



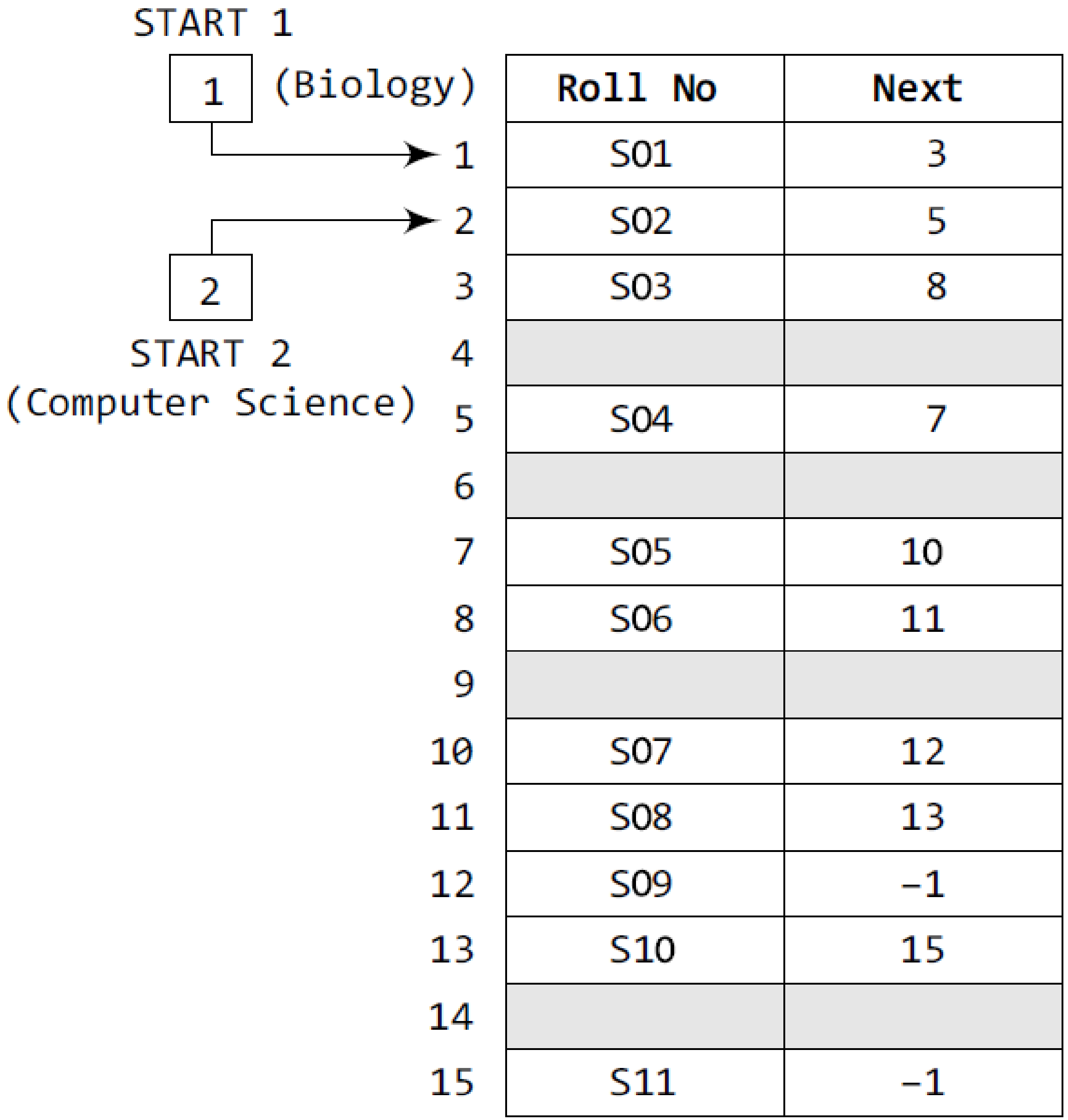
	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	–1

Figure 6.2 START pointing to the first element of the linked list in the memory

- In order to form a linked list, we need a structure called *node* which has two fields, DATA and NEXT.
- DATA will store the information part and NEXT will store the address of the next node in sequence.

- we can see that the variable START is used tostore the address of the first node. Here, in this example, START = 1, so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E.
- Again, we see the corresponding NEXT to go to the next node.
- From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data.
- We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.

- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form theword HELLO.



START 1

1 (Biology)

2

START 2
(Computer Science)

	Roll No	Next
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	−1
13	S10	15
14		
15	S11	−1

Figure 6.3 Two linked lists which are simultaneously maintained in the memory

- There is no ambiguity in traversing through the list because each list maintains a separate `Start` pointer, which gives the address of the first node of their respective linked lists.
- The rest of the nodes are reached by looking at the value stored in the NEXT.

LINKED LIST

PREPARED BY

DEEPTHI M PISHARODY

LINKED LISTS VERSUS ARRAYS

- Both arrays and linked lists are a linear collection of data elements.
- But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data.
- Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.

- Another advantage of a linked list over an array is that we can add any number of elements in the list.
- This is not possible in case of an array.
- For example, if we declare an array as `int marks[20]`, then the array can store a maximum of 20 data elements only.
- There is no such restriction in caseof a linked list.
- Thus, linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes.

	Roll No	Name	Aggregate	Grade	Next
1	S01	Ram	78	Distinction	6
2	S02	Shyam	64	First division	14
3					
4	S03	Mohit	89	Outstanding	17
5					
6	S04	Rohit	77	Distinction	2
7	S05	Varun	86	Outstanding	10
8	S06	Karan	65	First division	12
9					
10	S07	Veena	54	Second division	-1
11	S08	Meera	67	First division	4
12	S09	Krish	45	Third division	13
13	S10	Kusum	91	Outstanding	11
14	S11	Silky	72	First division	7
15					
16					
17	S12	Monica	75	Distinction	1
18	S13	Ashish	63	First division	19
19	S14	Gaurav	61	First division	8

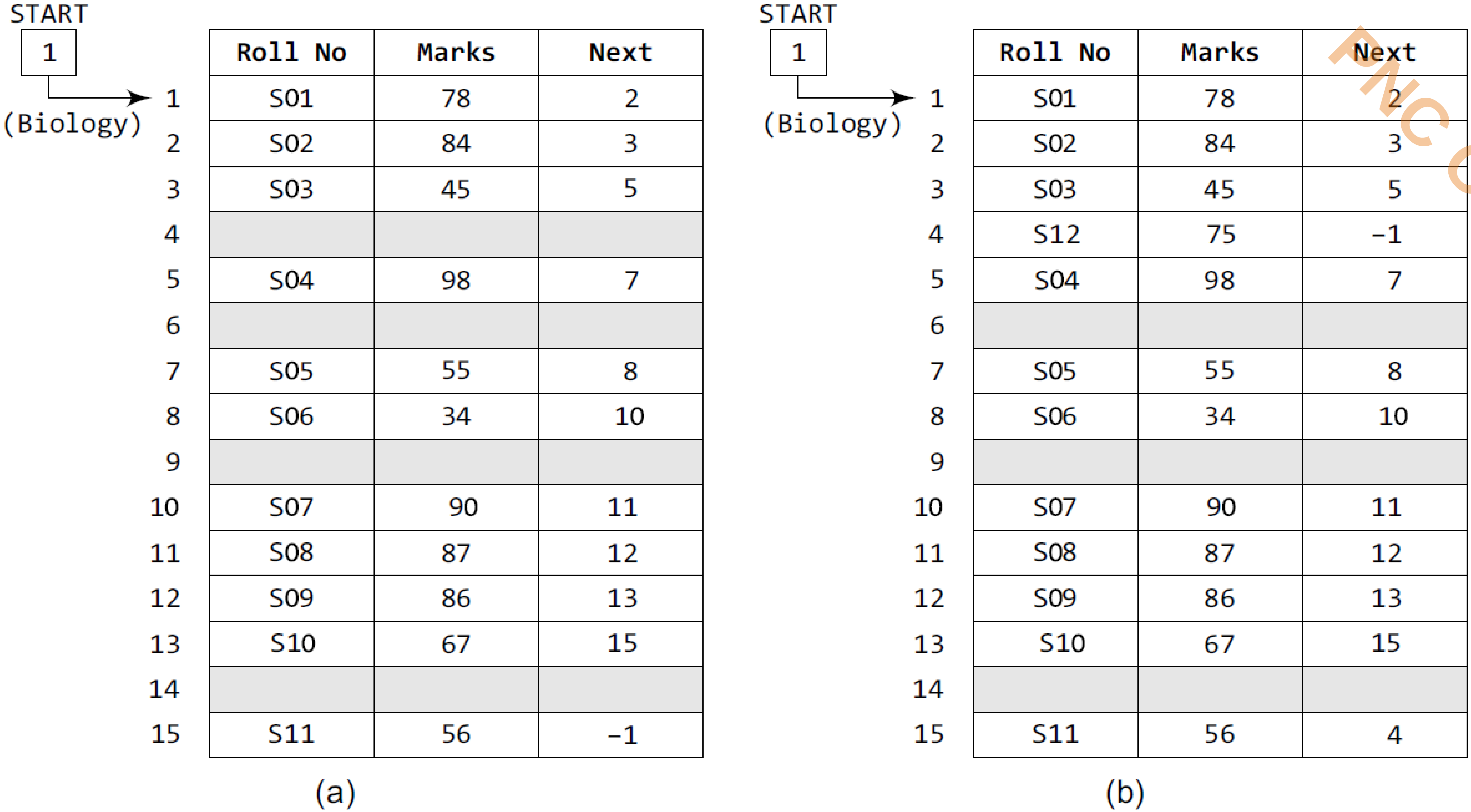
START

18 → 18

MEMORY ALLOCATION AND DE-ALLOCATION FOR A LINKED LIST

- We have seen how a linked list is represented in the memory.
- If we want to add a node to an already existing linked list in the memory, we first find free space in the memory and then use it to store the information.
- The linked list contains the roll number of students, marks obtained by them in Biology, and finally a `NEXT` field which stores the address of the next node in sequence.
- Now, if a new student joins the class and is asked to appear for the same test that the other students had taken, then the new student’s marks should also be recorded in the linked list.
- For this purpose, we find a free space and store the information there.
- In Fig. the grey shaded portion shows free space, and thus we have 4 memory locations available.
- We can use any one of them to store our data.

- Now, the question is which part of the memory is available and which part is occupied?
- When we delete a node from a linked list, then who changes the status of the memory occupied by it
- from occupied to available?
- The answer is the operating system.
- The computer does it on its own without any intervention from the user or the
- programmer.
- As a programmer, you just have to take care of the code to perform insertions and deletions in the list.
- The computer maintains a list of all free memory cells. This list of available space is called the *free pool*.



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	–1

(a)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4	S12	75	–1
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	4

(b)

Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

Thank you

TYPES OF LINKED LIST- LINEAR LINKED LIST

Prepared by
Deepthi M Pisharody
Asst Professor
Prajyoti Niketan College, Pudukad

SINGLY LINKED LISTS

- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence.
- A singly linked list allows traversal of data only in one way

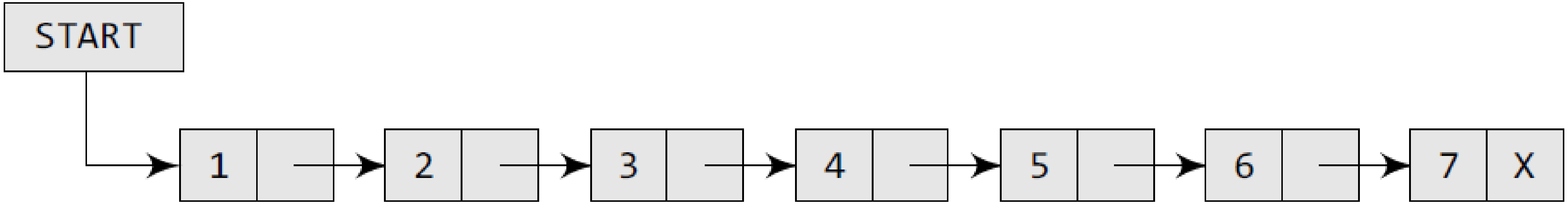
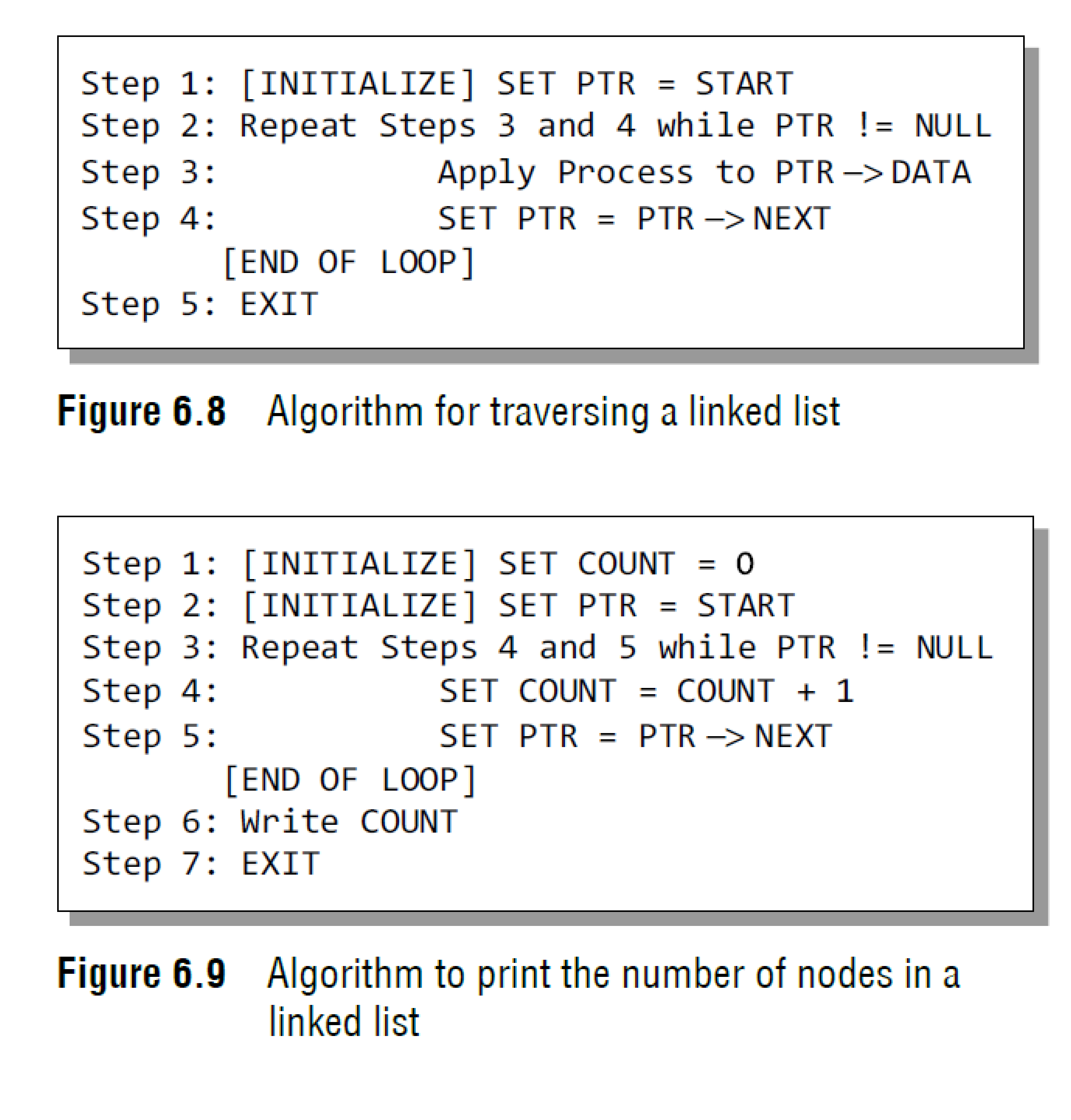


Figure 6.7 Singly linked list

TRAVERSING A LINKED LIST

- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.
- Remember a linked list always contains a pointer variable START which stores the address of the first node of the list.
- End of the list is marked by storing NULL or –1 in the NEXT field of the last node.
- For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:             Apply Process to PTR -> DATA
Step 4:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: EXIT
```

Figure 6.8 Algorithm for traversing a linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:             SET COUNT = COUNT + 1
Step 5:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

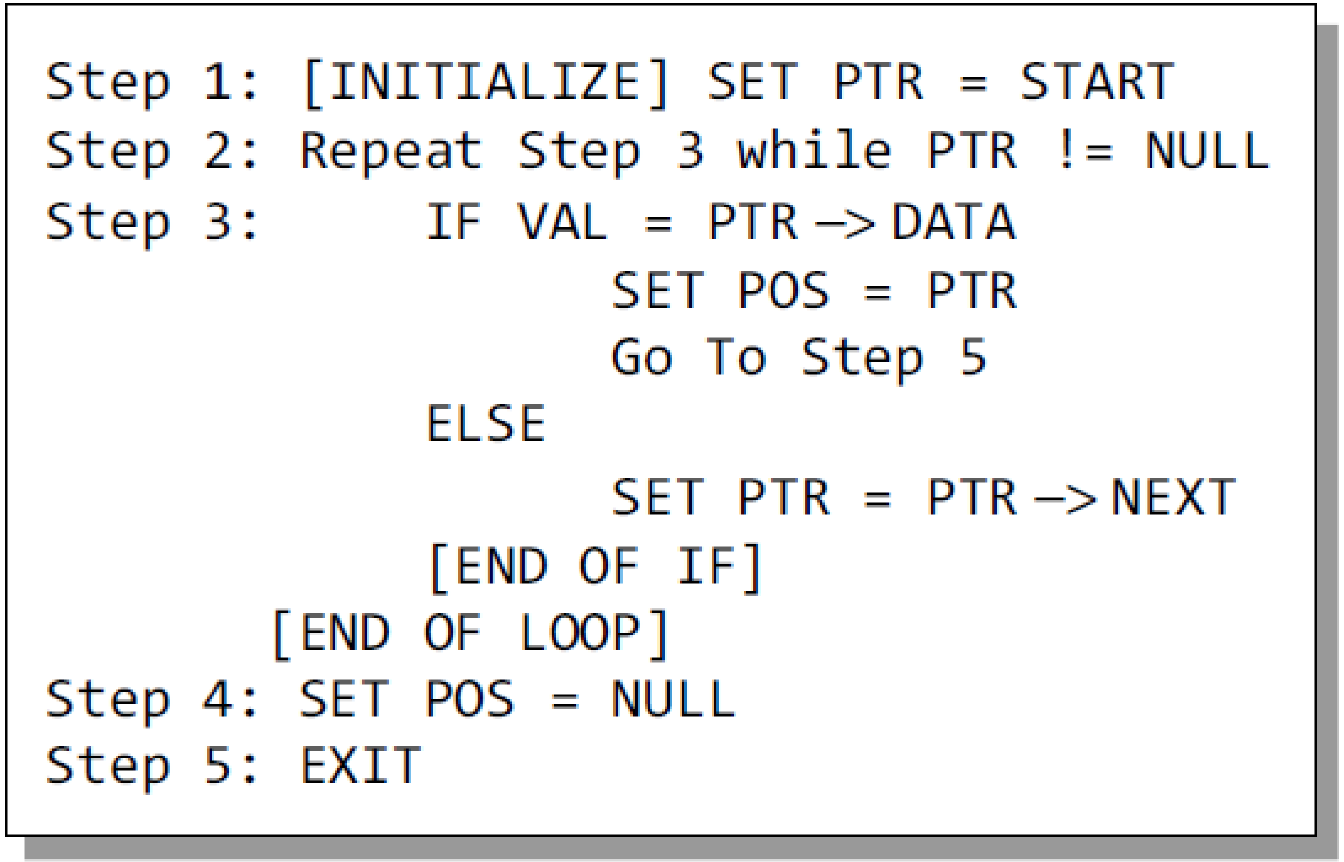
Figure 6.9 Algorithm to print the number of nodes in a linked list

- In this algorithm, we first initialize `PTR` with the address of `START`.
- So now, `PTR` points to the first node of the linked list.
- Then in Step 2, a `while` loop is executed which is repeated till `PTR` processes the last node, that is until it encounters `NULL`.
- In Step 3, we apply the process (e.g., `print`) to the current node, that is, the node pointed by `PTR`.
- In Step 4, we move to the next node by making the `PTR` variable point to the node whose address is stored in the `NEXT` field.

- An algorithm to count the number of nodes in a linked list.
- To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1.
- Once we reach `NULL`,that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed.

SEARCHING FOR A VALUE IN A LINKED LIST

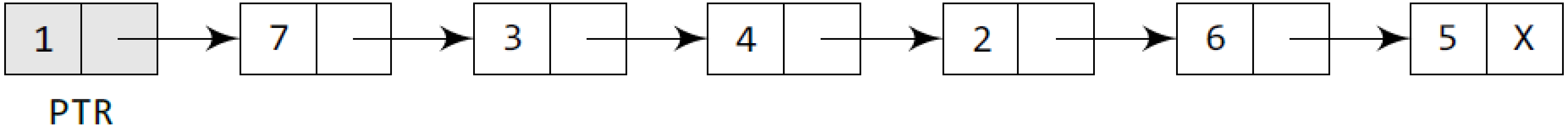
- Searching a linked list means to find a particular element in the linked list.
- A linked list consists of nodes which are divided into two parts, the information part and the next part.
- So searching means finding whether a given value is present in the information part of the node or not.
- If it is present, the algorithm returns the address of the node that contains the value.



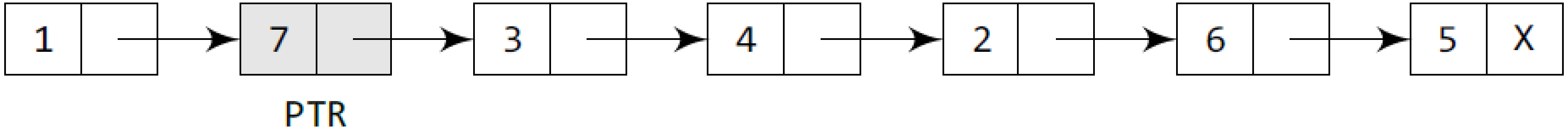
```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                 SET POS = PTR
                 Go To Step 5
            ELSE
                 SET PTR = PTR -> NEXT
            [END OF IF]
       [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

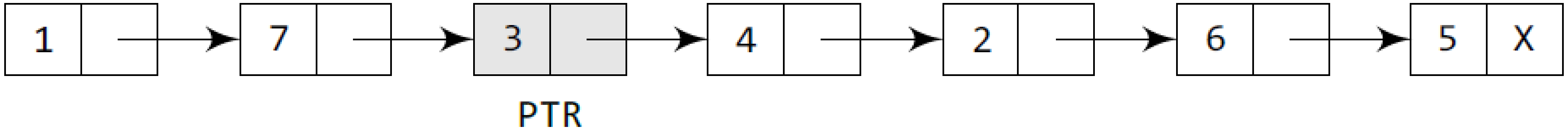
- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made.
- If the search is successful, that is, VAL hasbeen found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.



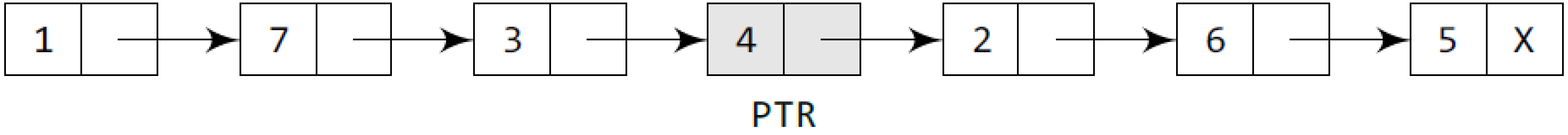
Here PTR –> DATA = 1. Since PTR –> DATA != 4, we move to the next node.



Here PTR –> DATA = 7. Since PTR –> DATA != 4, we move to the next node.



Here PTR –> DATA = 3. Since PTR –> DATA != 4, we move to the next node.

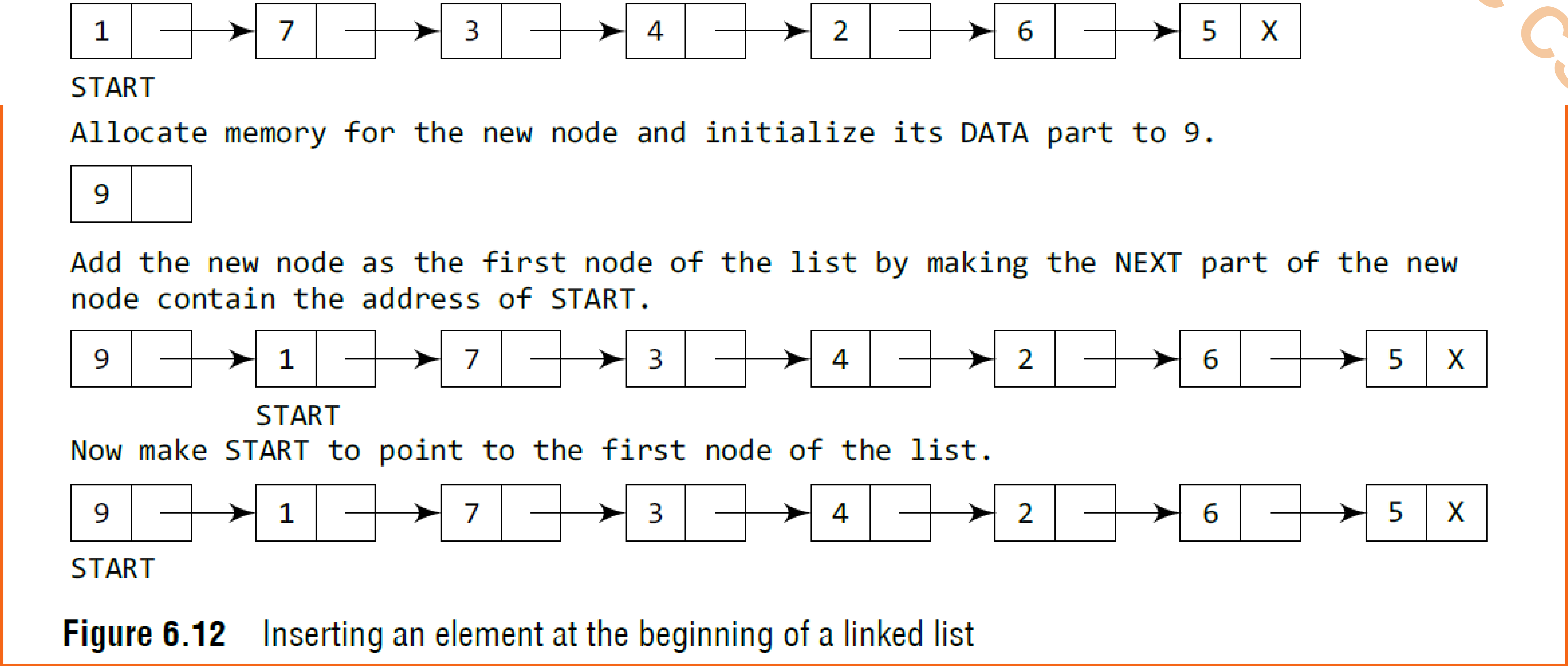


Here PTR –> DATA = 4. Since PTR –> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

INSERTING A NEW NODE IN A LINKED LIST

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.

INSERTING A NODE AT THE BEGINNING OF A LINKED LIST

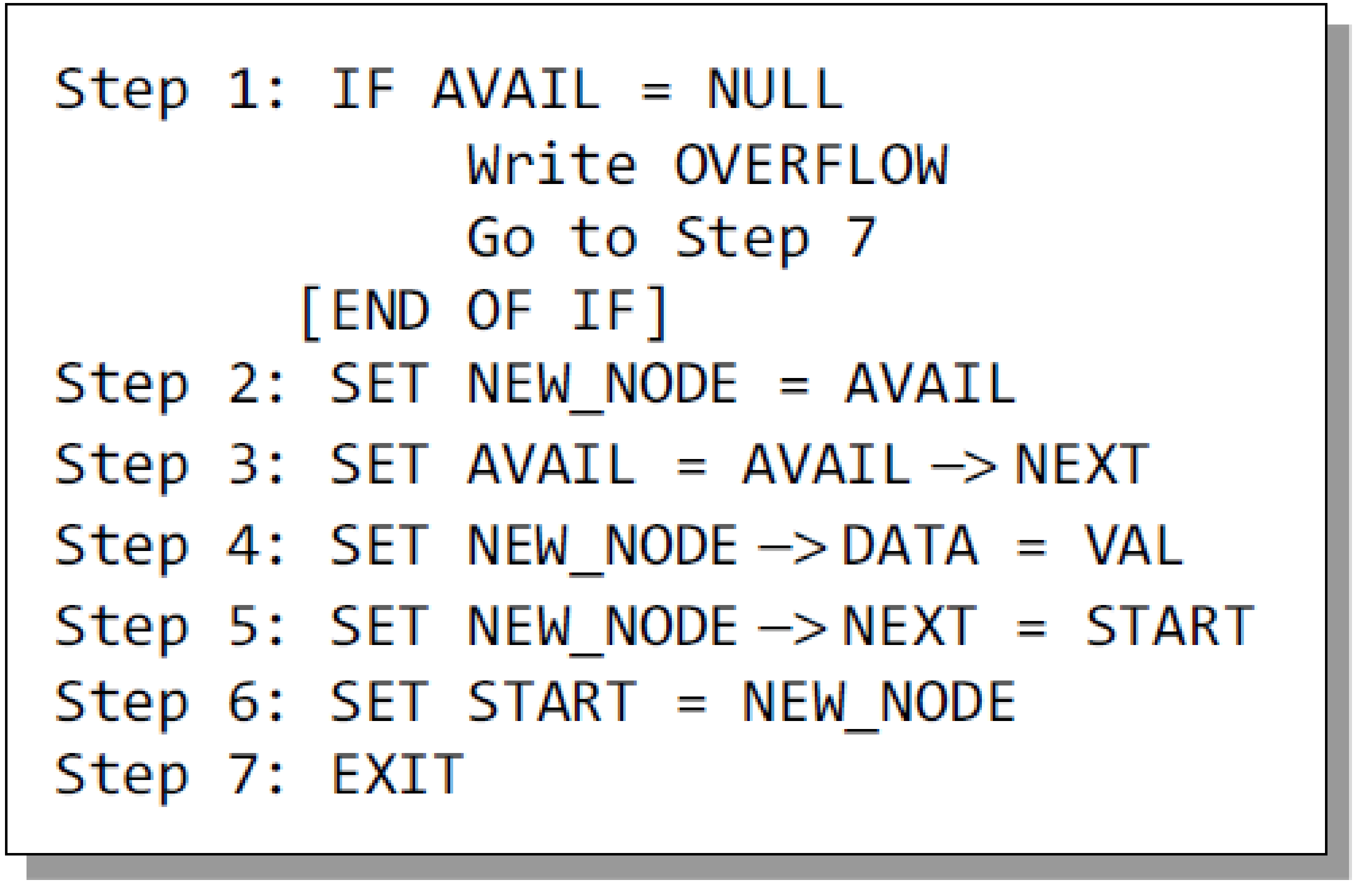


Allocate memory for the new node and initialize its DATA part to 9.

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.

Now make START to point to the first node of the list.

Figure 6.12 Inserting an element at the beginning of a linked list

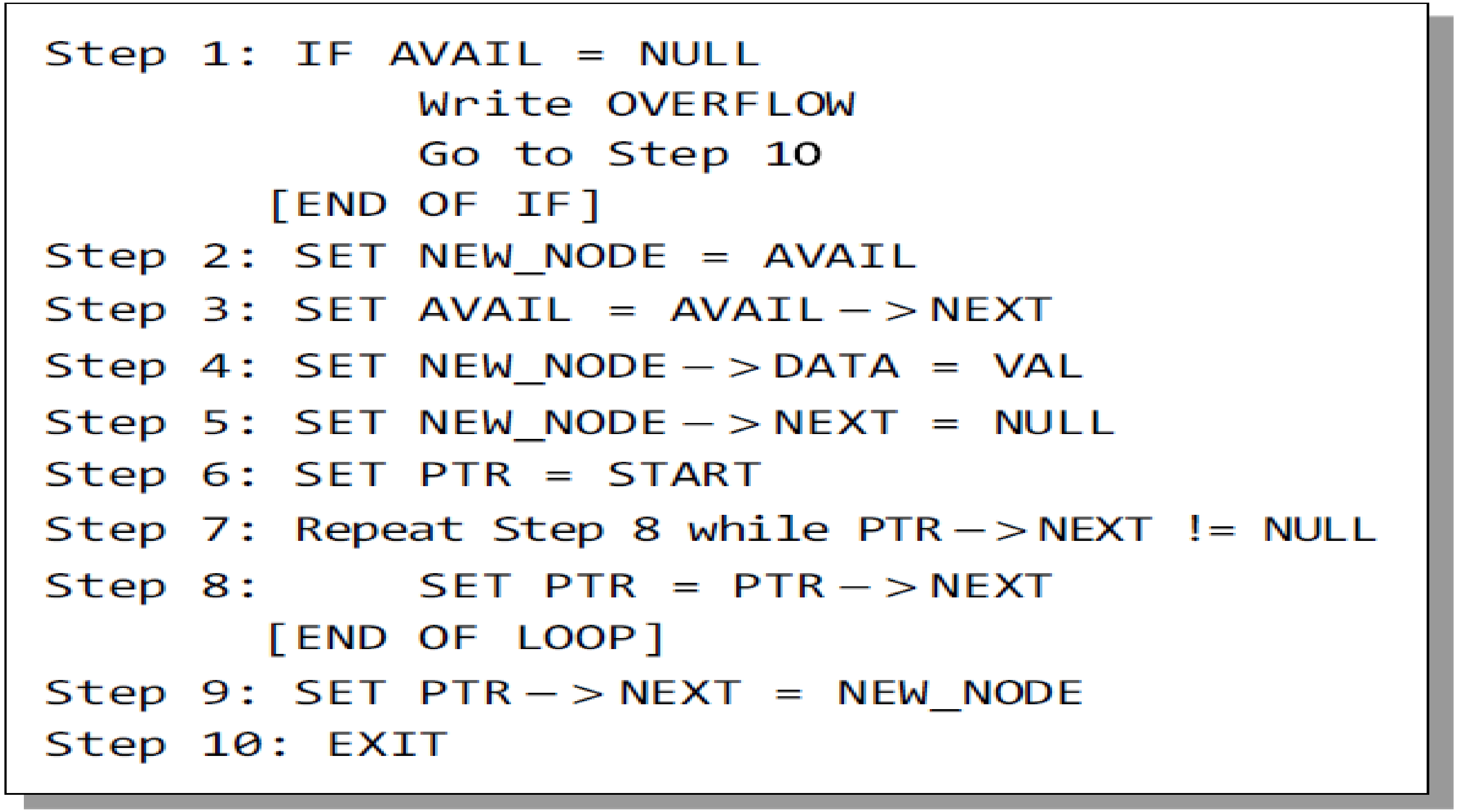


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if a free memory cell is available, then we allocate space for the new node.
- Set its DATA part with thegiven VAL and the next part is initialized with the address of the first node of the list, which is stored in START.

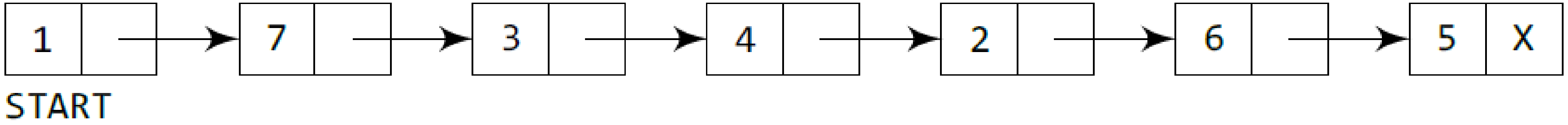
- Now, since the new node is added as the first node of the list, it will now be known as the `START` node, that is, the `START` pointer variable will now hold the address of the `NEW_NODE`. Note the
- following two steps:
- `Step 2: SET NEW_NODE = AVAIL`
- `Step 3: SET AVAIL = AVAIL -> NEXT`
- These steps allocate memory for the new node. In C, there are functions like `malloc()`, `alloc`, and `calloc()` which automatically do the memory allocation on behalf of the user.

INSERTING A NODE AT THE END OF A LINKED LIST

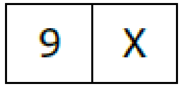


```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
         [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:         SET PTR = PTR->NEXT
         [END OF LOOP]
Step 9: SET PTR->NEXT = NEW_NODE
Step 10: EXIT
```

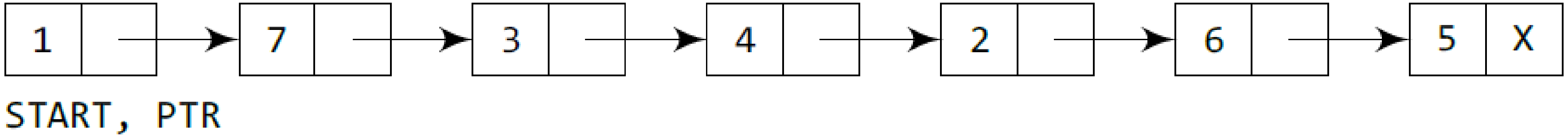
Figure 6.15 Algorithm to insert a new node at the end



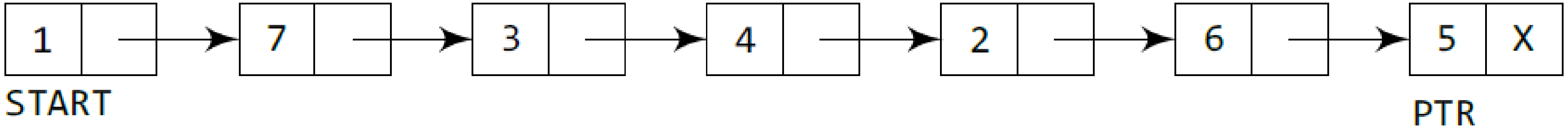
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.



Take a pointer variable PTR which points to START.



Move PTR so that it points to the last node of the list.



Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

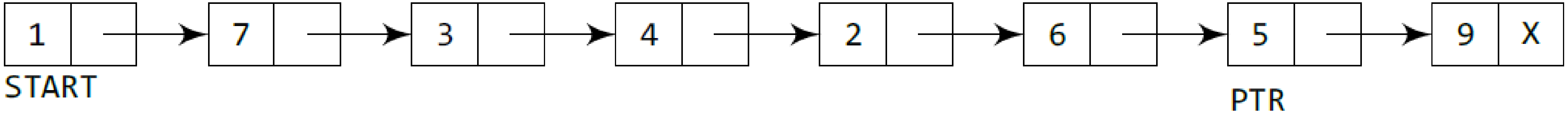
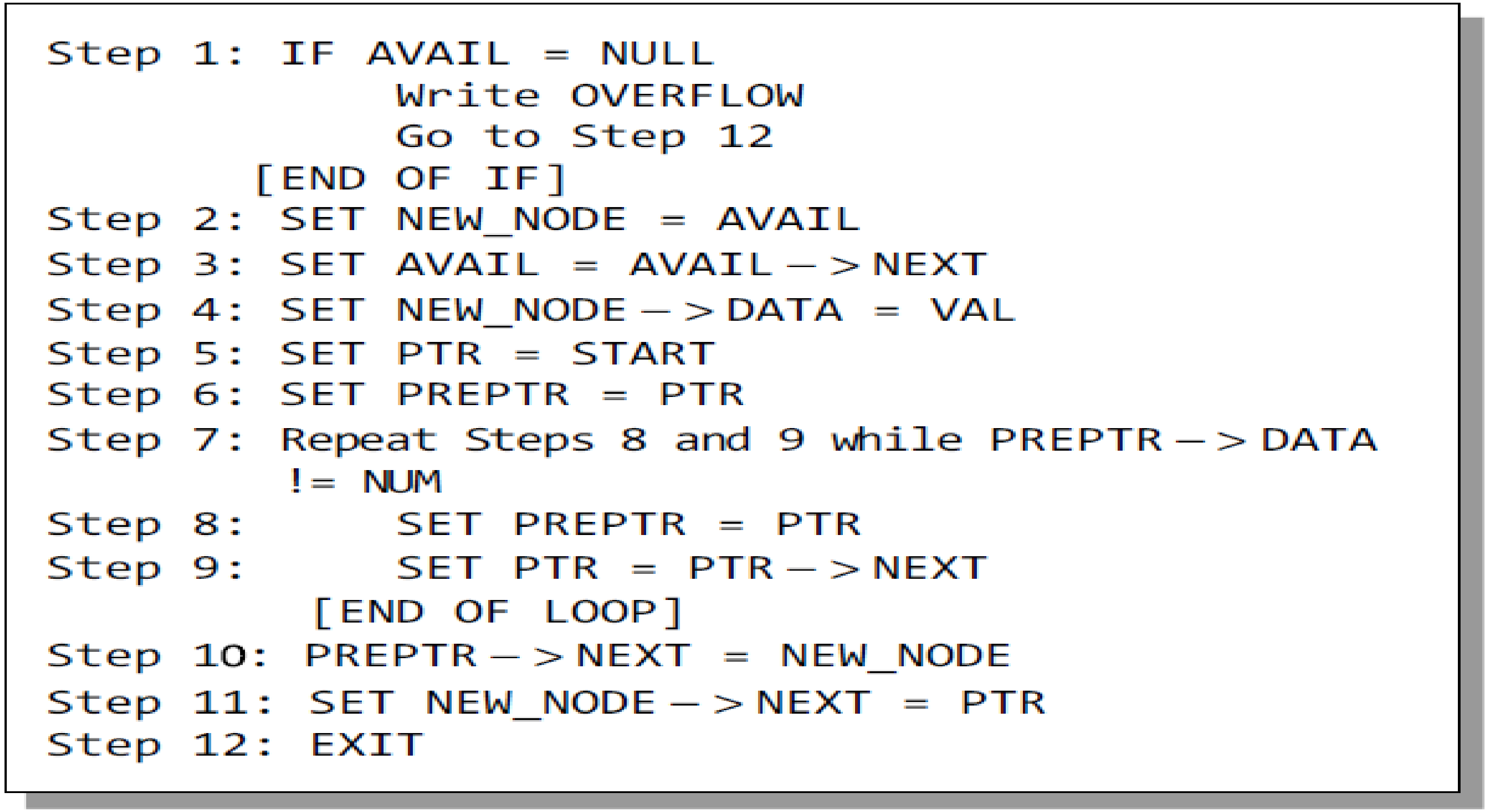


Figure 6.14 Inserting an element at the end of a linked list

INSERTING A NODE AT THE END OF A LINKED LIST

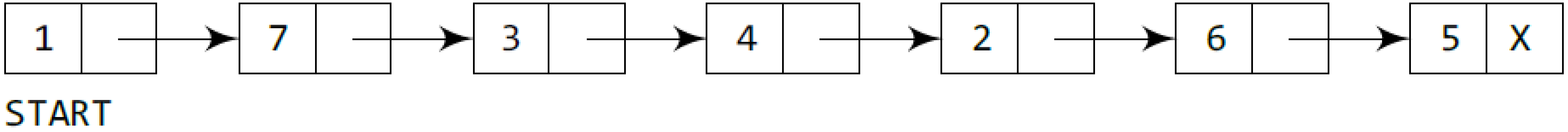
- In Step 6, we take a pointer variable `PTR` and initialize it with `START`.
- That is, `PTR` now points to the first node of the linked list. In the `while` loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the `NEXT` pointer of the last node to store the address of the new node.
- Remember that the `NEXT` field of the new node contains `NULL`, which signifies the end of the linked list.

INSERTING A NODE AFTER A GIVEN NODE IN A LINKED LIST

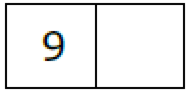


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
        != NUM
Step 8:         SET PREPTR = PTR
Step 9:         SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

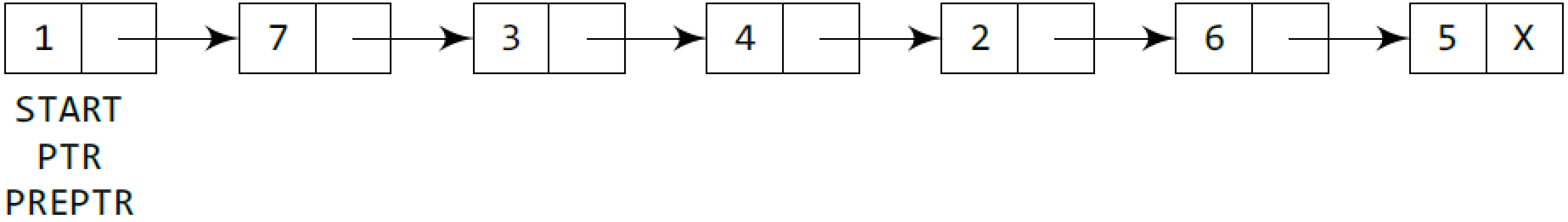
Figure 6.16 Algorithm to insert a new node after a node that has value NUM



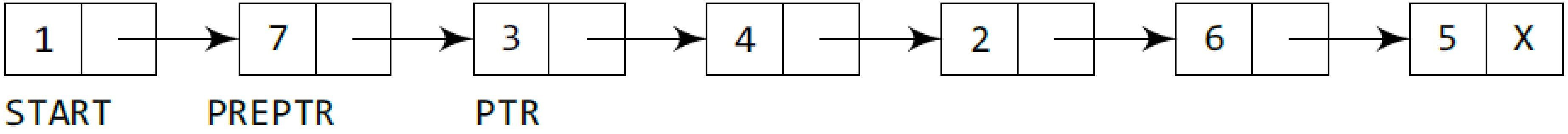
Allocate memory for the new node and initialize its DATA part to 9.



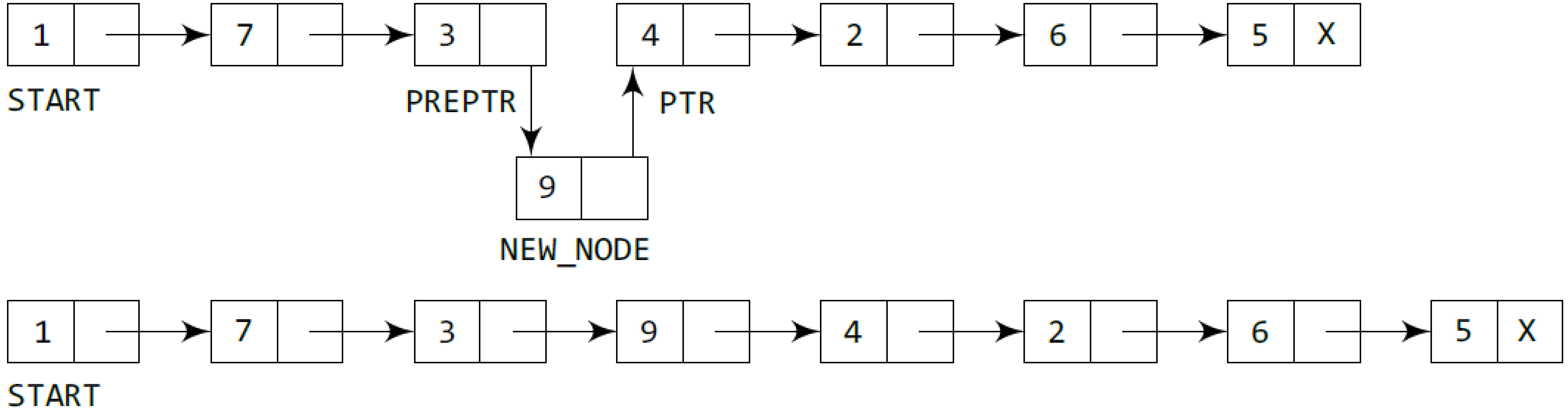
Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.



Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.



Add the new node in between the nodes pointed by PREPTR and PTR.



- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR.
- Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

Thank you

DELETING A NODE FROM A LINKED LISTLINEAR LINKED LIST

Prepared by

Deepthi M Pisharody

Asst Professor

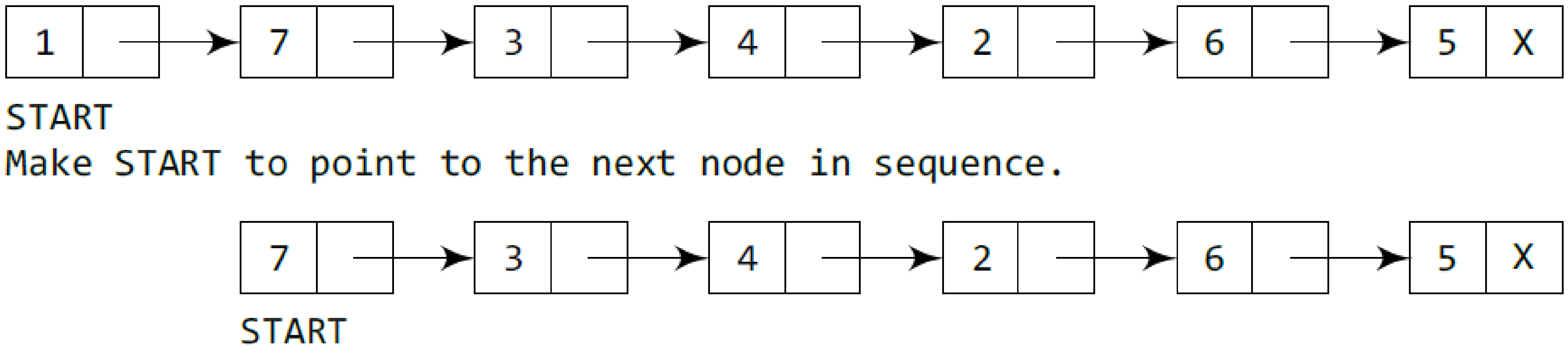
Prajyoti Niketan College, Pudukad

DELETING A NODE FROM A LINKED LIST

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.

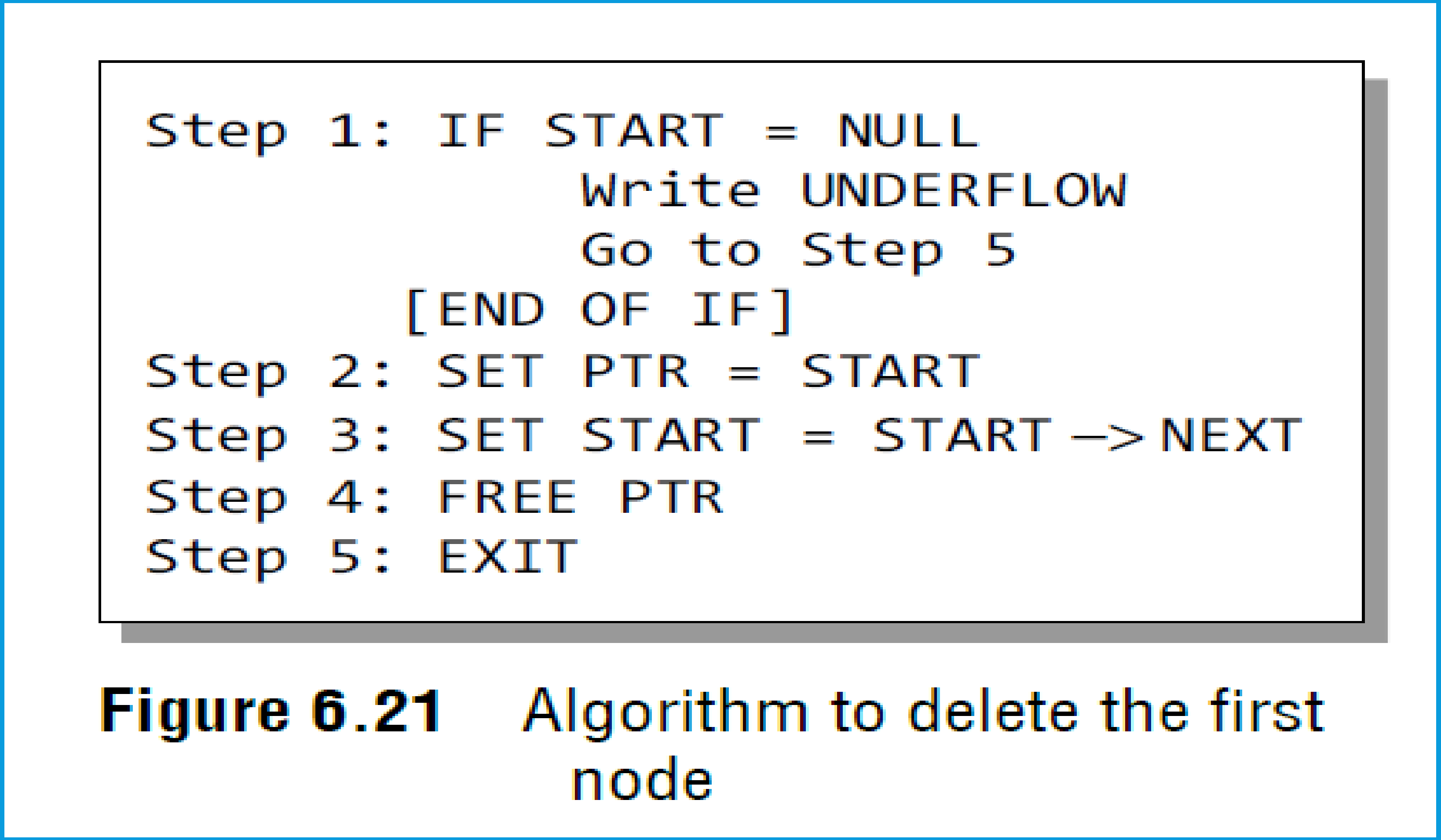
- Underflow is a condition that occurs when we try to delete a node from a linked list that is empty.
- This happens when START = NULL or when there are no more nodes to delete.
- Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node.
- The memory is returned to the free pool so that it can be used to store other programs and data.
- Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated

DELETING THE FIRST NODE FROM A LINKED LIST



Make START to point to the next node in sequence.

Figure 6.20 Deleting the first node of a linked list

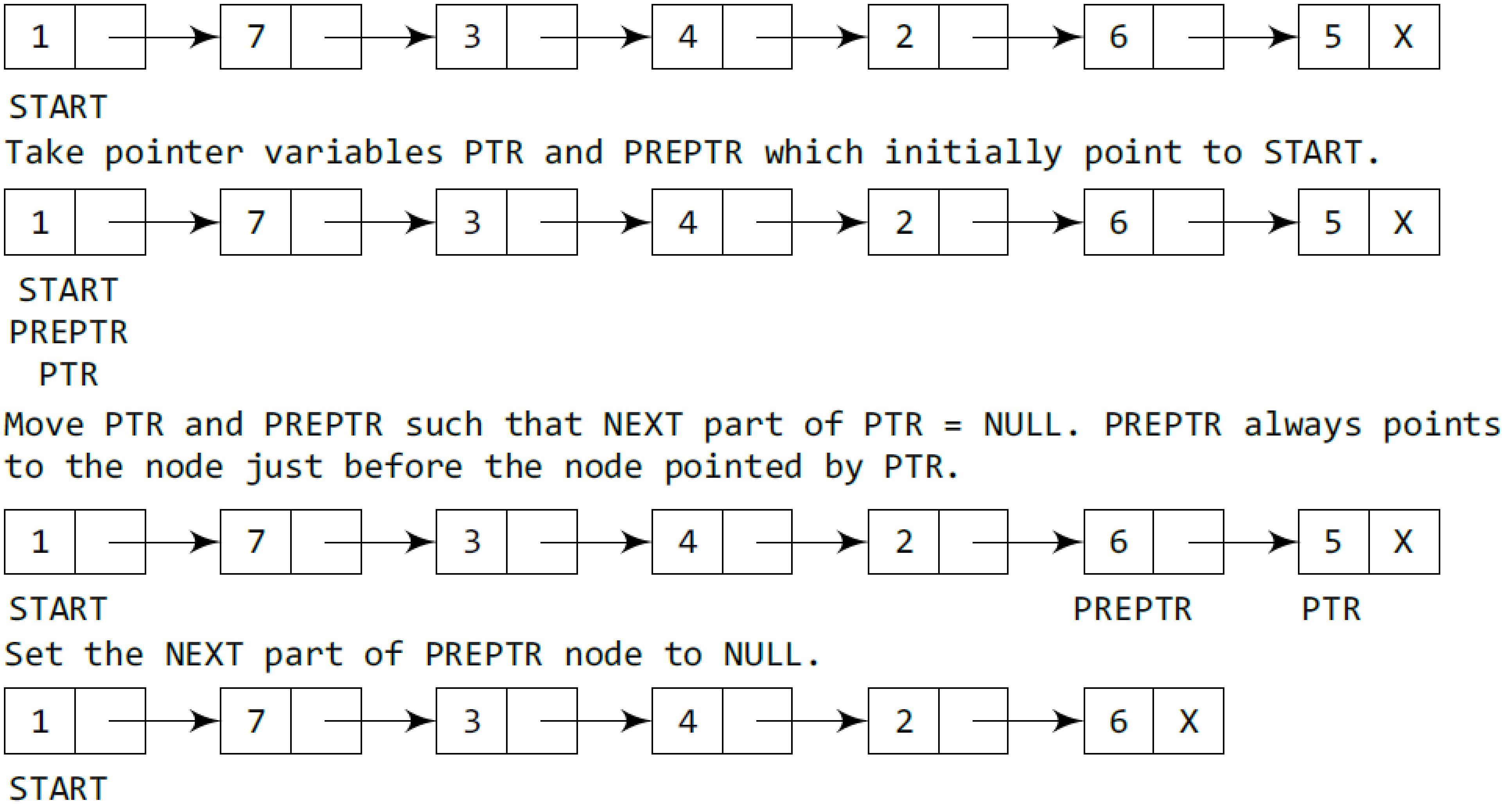


```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 5
           [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

- In Step 1, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

DELETING THE LAST NODE FROM A LINKED LIST

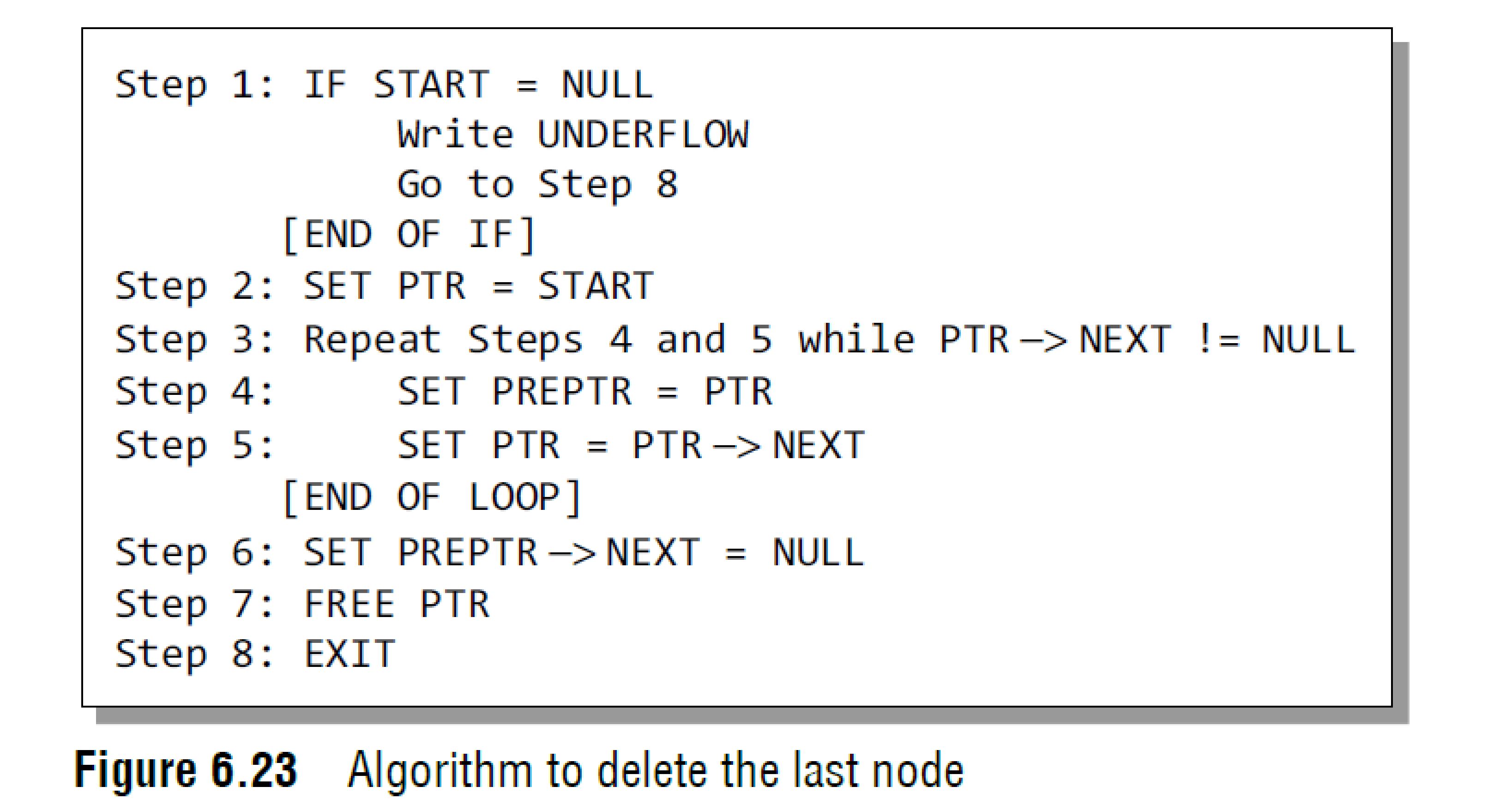


Take pointer variables PTR and PREPTR which initially point to START.

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

Set the NEXT part of PREPTR node to NULL.

Figure 6.22 Deleting the last node of a linked list

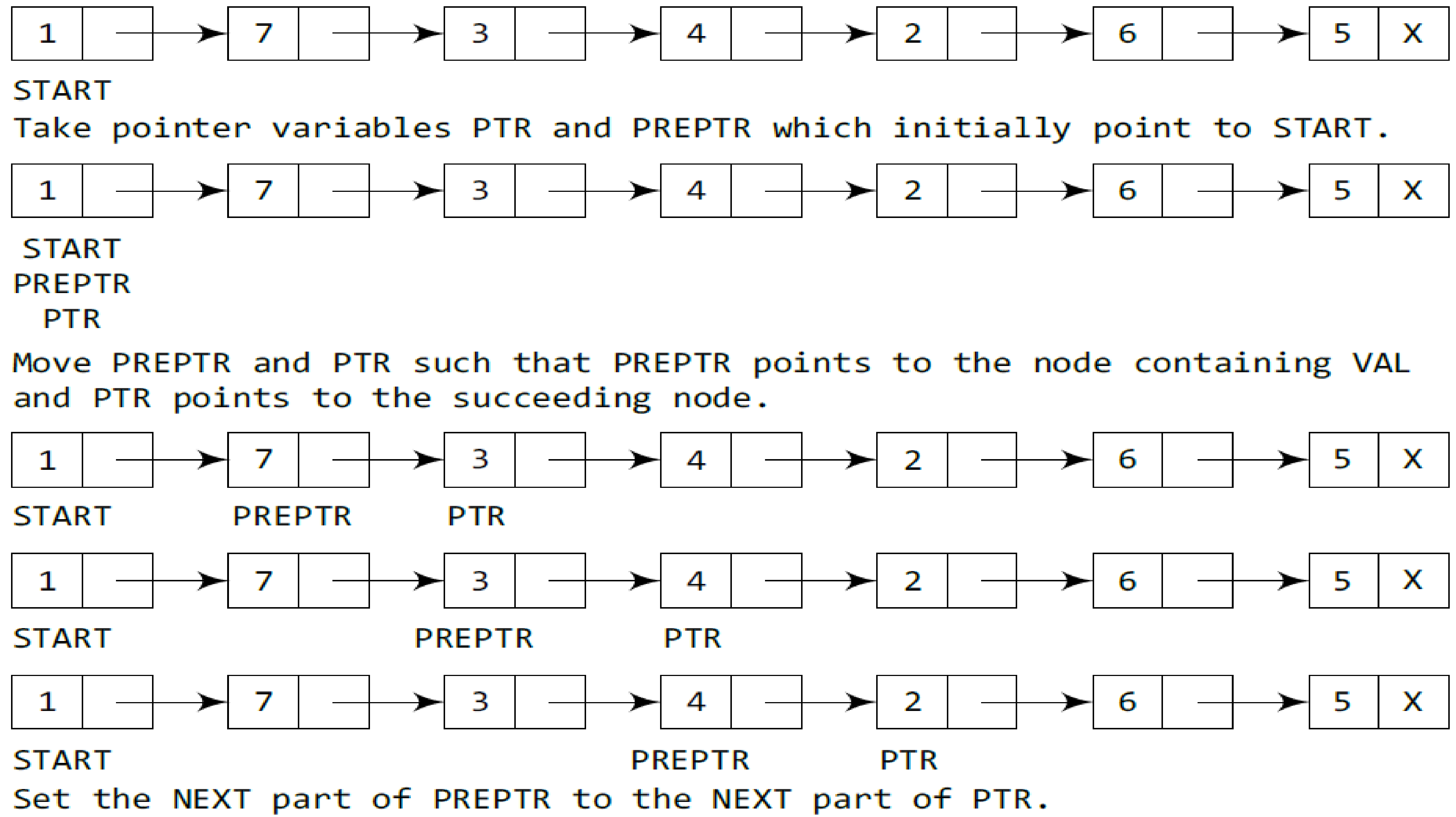


```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
       [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.23 Algorithm to delete the last node

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned back to the free pool.

DELETING THE NODE AFTER A GIVEN NODE IN A LINKED LIST



Take pointer variables PTR and PREPTR which initially point to START.

Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.

Set the NEXT part of PREPTR to the NEXT part of PTR.

Set the NEXT part of PREPTR to the NEXT part of PTR.

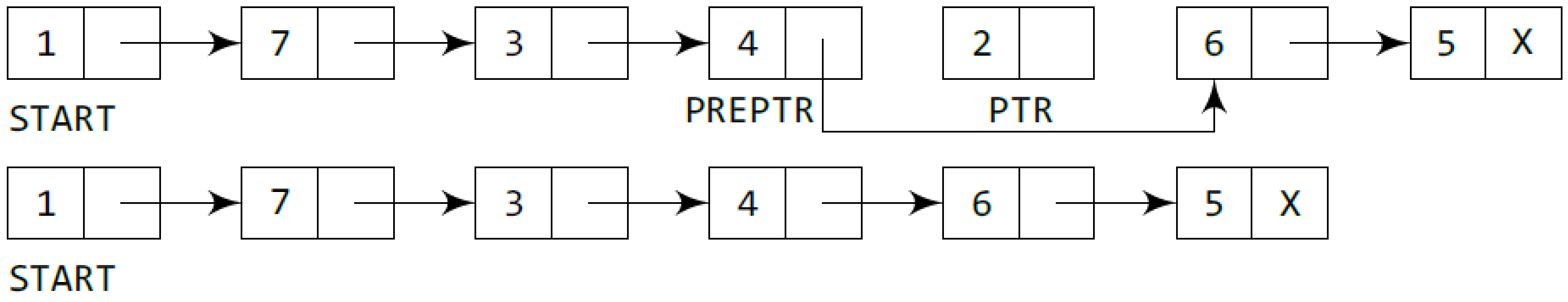
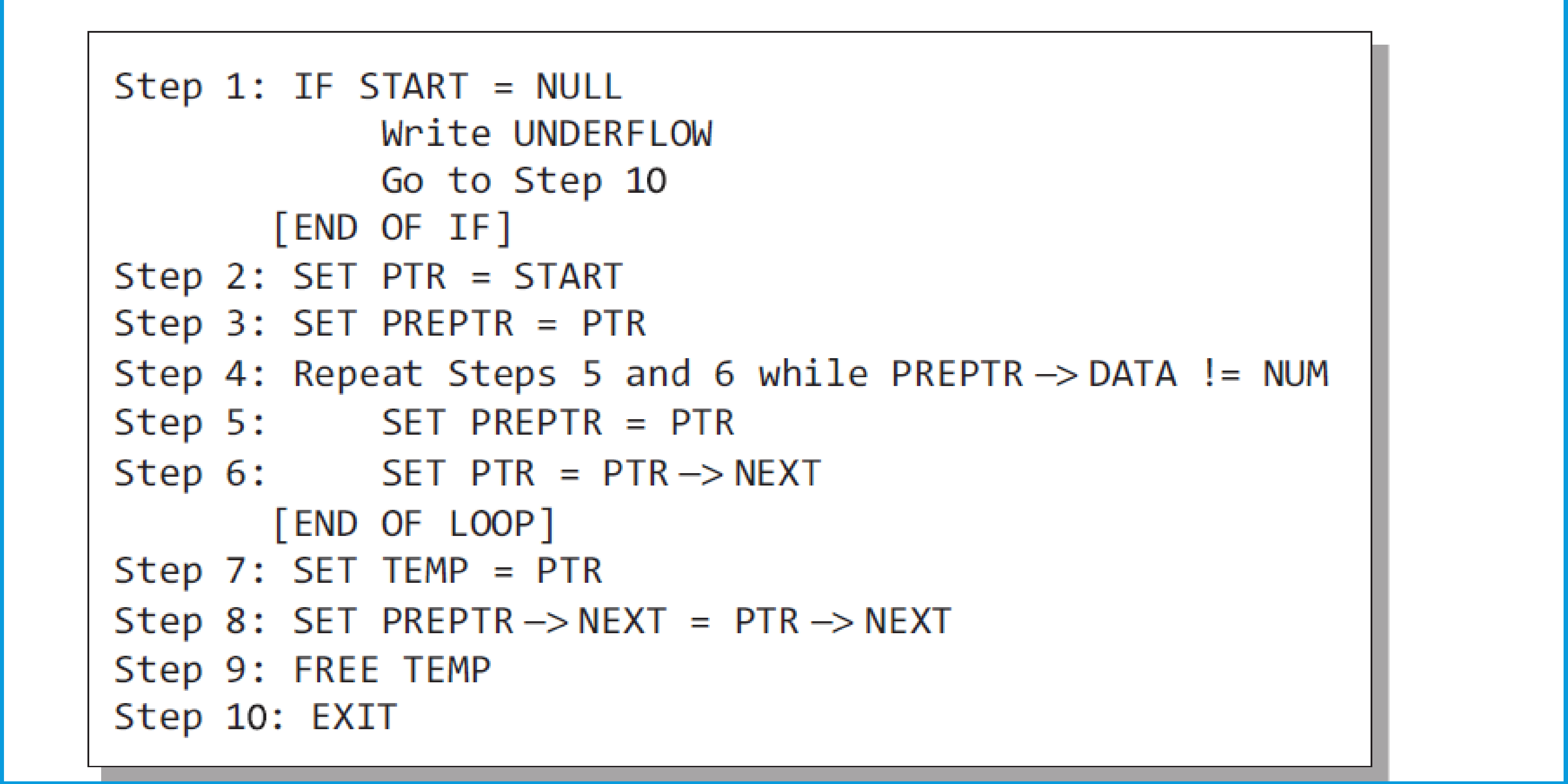


Figure 6.24 Deleting the node after a given node in a linked list



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:       SET PREPTR = PTR
Step 6:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

- In Step 2, we take a pointer variable `PTR` and initialize it with `START`.
- That is, `PTR` now points to the first node of the linked list.
- In the `while` loop, we take another pointer variable `PREPTR` such thatit always points to one node before the `PTR`.
- Once we reach the node containing `VAL` and the node succeeding it, we set the next pointer of the node containing `VAL` to the address contained in next field of the node succeeding it.
- The memory of the node succeeding the given node is freed and returned back to the free pool.

Thank You

Various types of linked list

Prepared by

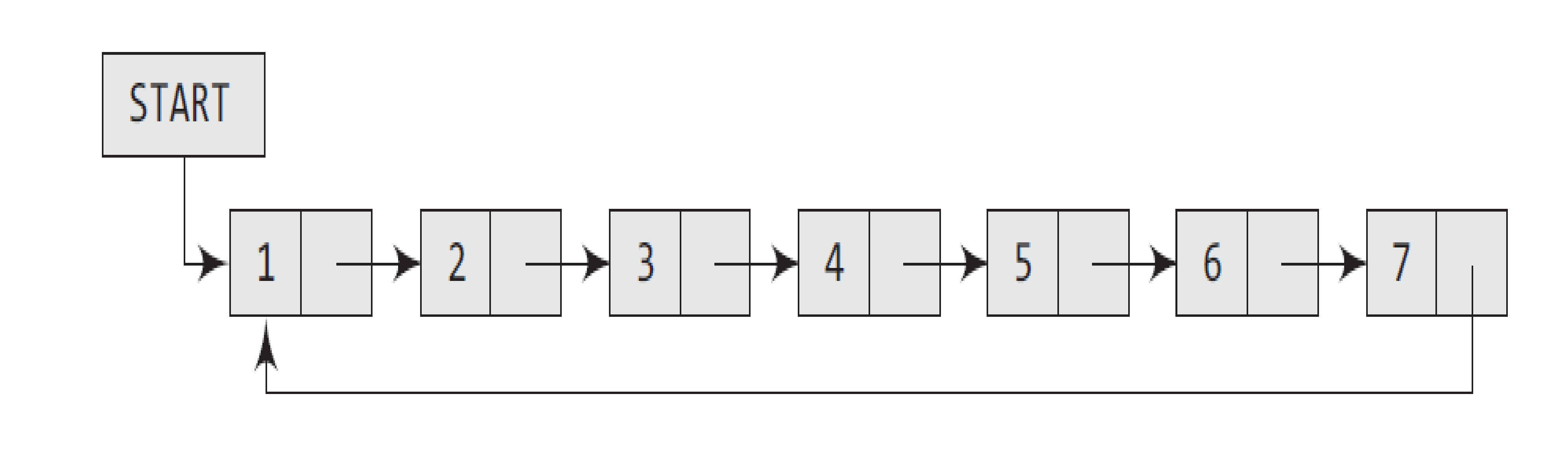
Deepthi M Pisharody

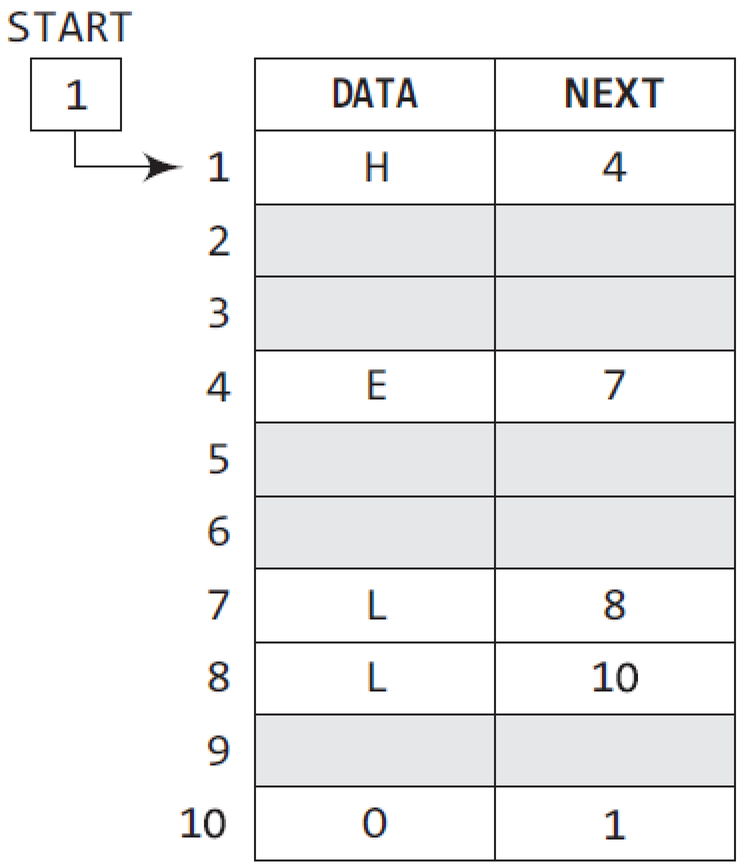
Asst Professor

Prajyoti Niketan College, Pudukad

CIRCULAR LINKED LISTs

- In a circular linked list, the last node contains a pointer to the first node of the list.
- While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.
- Thus, a circular linked list has no beginning and no ending.
- The only downside of a circular linked list is the complexity of iteration.


START
1
2
3
4
5
6
7



START

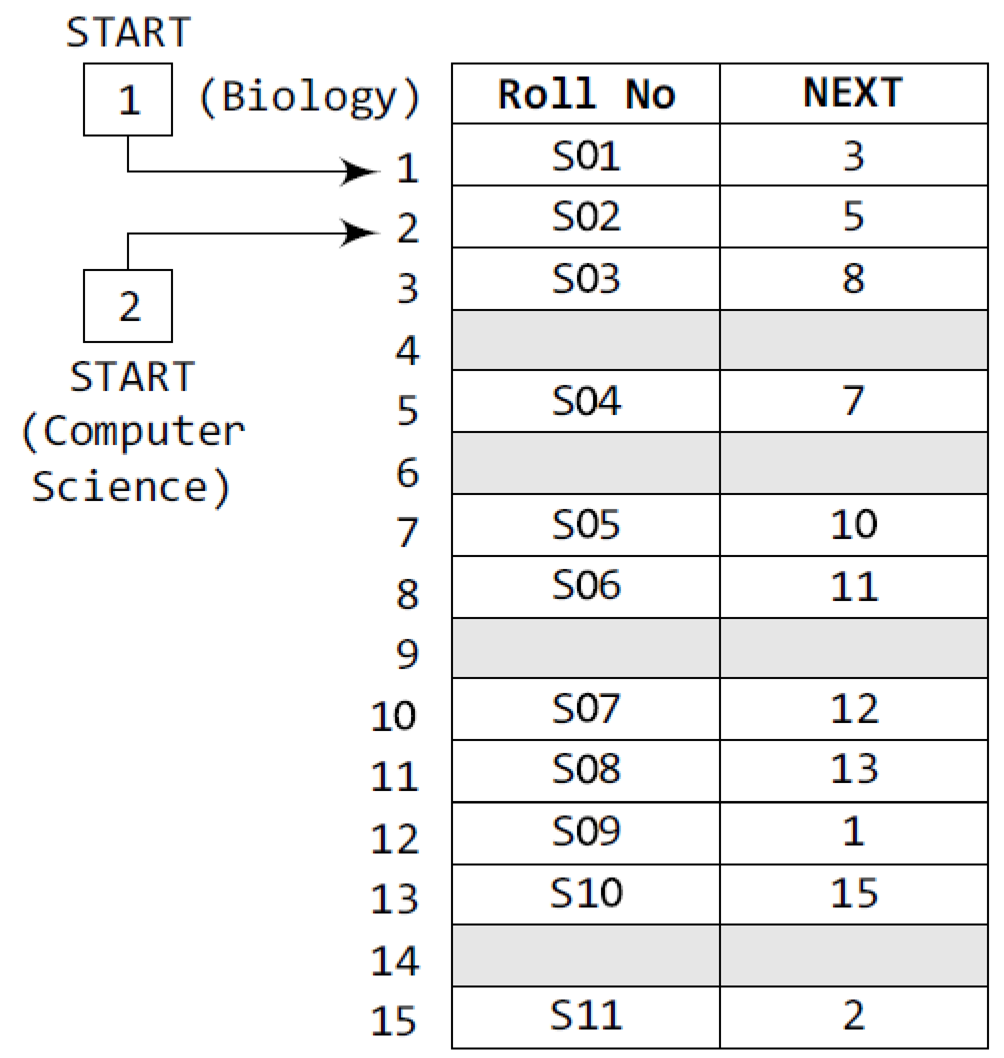
1

	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

Figure 6.27 Memory representation of a circular linked list

- Circular linked lists are widely used in operating systems for task maintenance.
- When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited. How is this done? The answer is simple.
- A circular linked list is used to maintain the sequence of the Web pages visited.
- Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons.
- Actually, this is done using either the circular stack or the circular queue

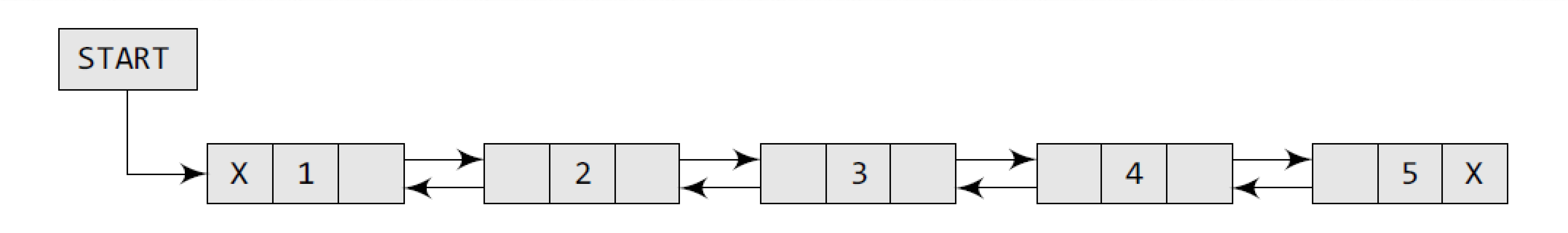
- We can traverse the list until we find the NEXT entry that contains the address of the first node of the list.
- This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

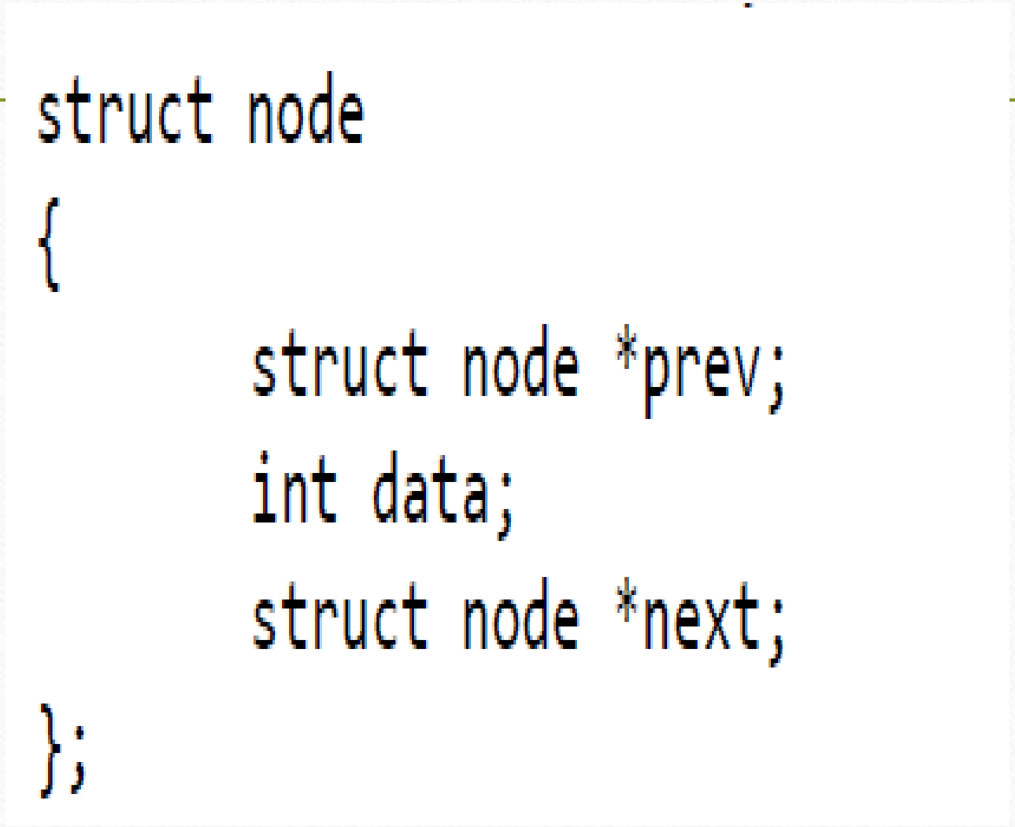


	Roll No	NEXT
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	1
13	S10	15
14		
15	S11	2

DOUBLY LINKED LISTS

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node

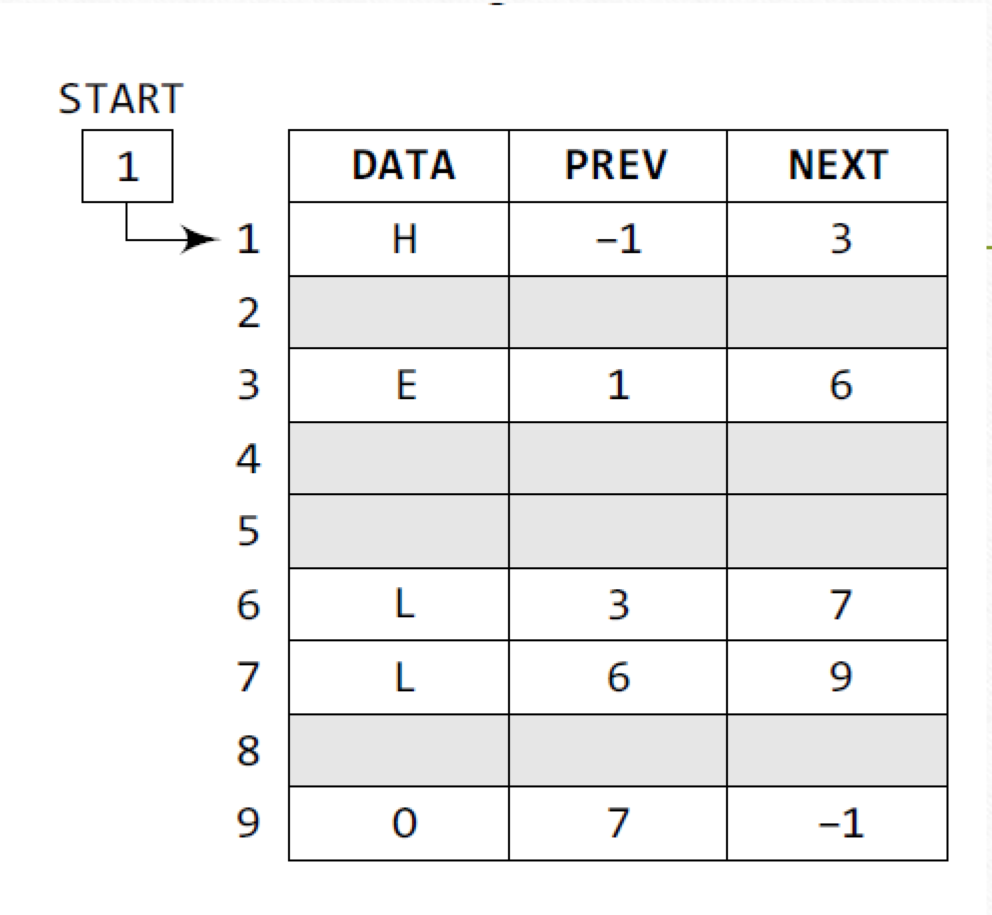

START
X
1
2
3
4
5
X



```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

- The `PREV` field of the first node and the `NEXT` field of the last node will contain `NULL`.
- The `PREV` field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

- A doubly linked list calls for more space per node and more expensive basic operations.
- However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a doubly linked list is that it makes searching twice as efficient.



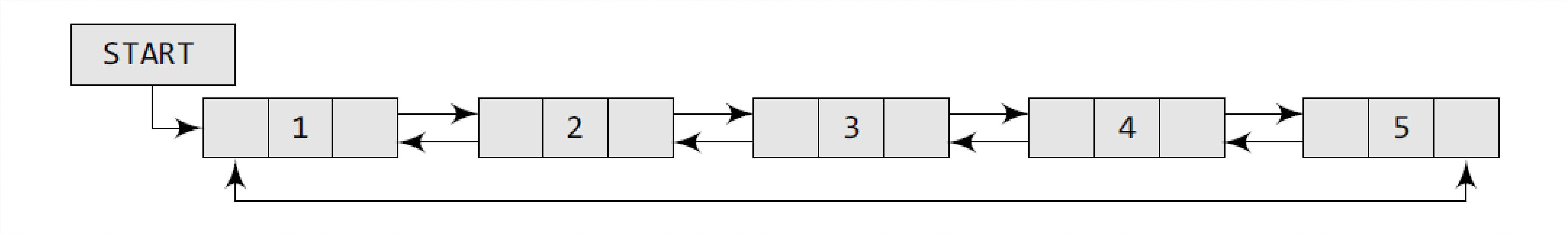
START

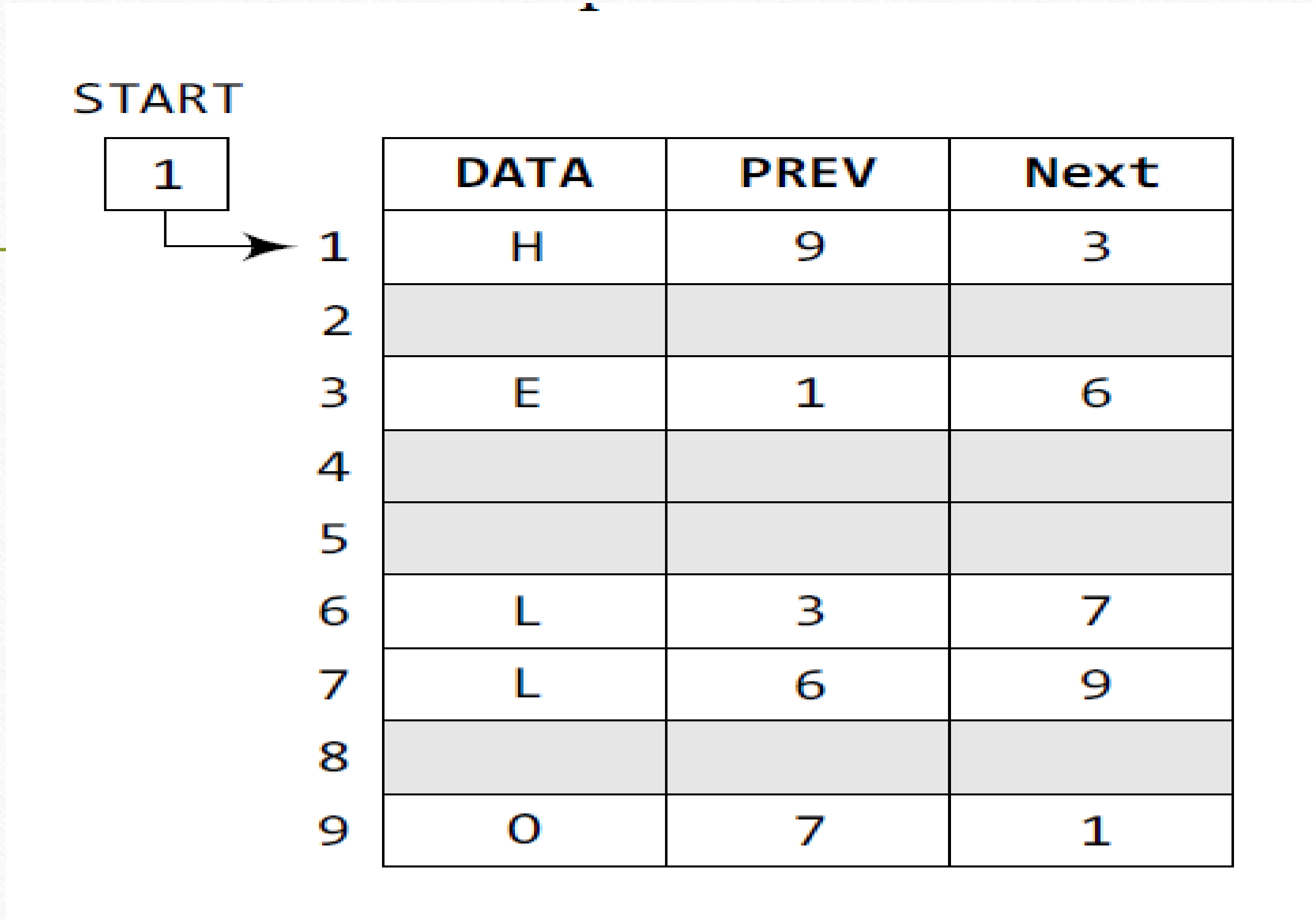
1

	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

CIRCULAR DOUBLY LINKED LISTs

- A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list.
- The circular doubly linked list does not contain `NULL` in the previous field of the first node and the next field of the last node.
- Rather, the next field of the last node stores the address of the first node of the list, i.e., `START`.
- Similarly, the previous field of the first field stores the address of the last node.


START
1
2
3
4
5



START

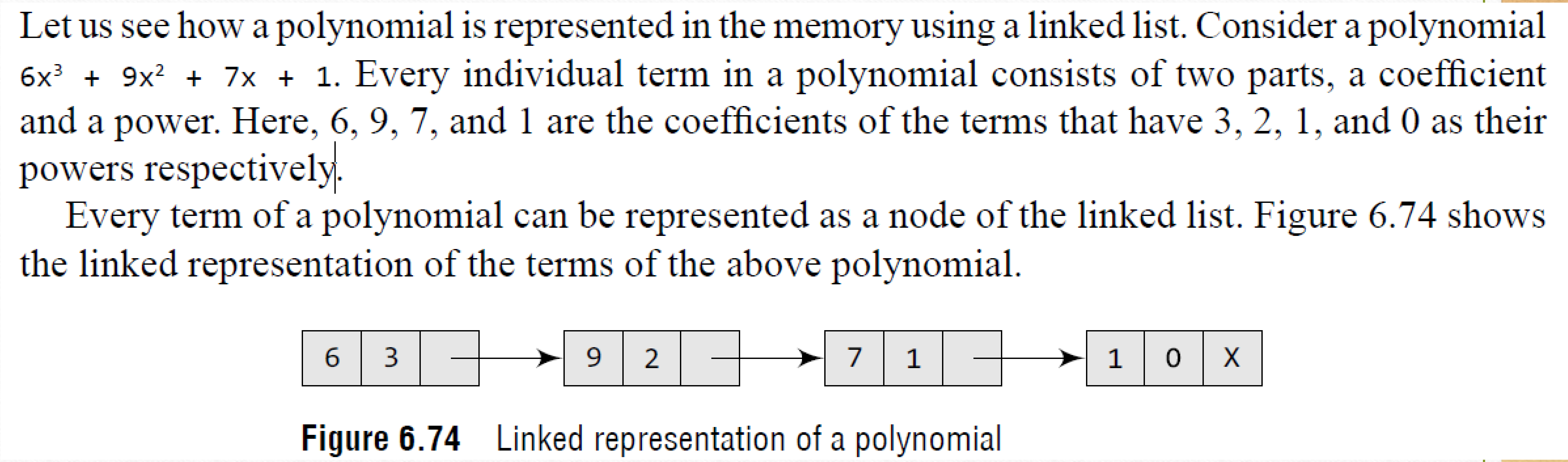
1

	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	1

APPLICATIONS OF LINKED LISTS

- **Polynomial Representation**

Polynomial Representation



Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial $6x^3 + 9x^2 + 7x + 1$. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

Every term of a polynomial can be represented as a node of the linked list. Figure 6.74 shows the linked representation of the terms of the above polynomial.

Figure 6.74 Linked representation of a polynomial

Thank you

STACKS

Prepared by

Deepthi M Pisharody

Asst Professor

Prajyoti Niketan College, Pudukad

- Stack is an important data structure which stores its elements in an ordered manner.
- Eg:- a pile of plates where one plate is placed on top of another.
- When you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., A plate) only at/from one position which is the topmost position.

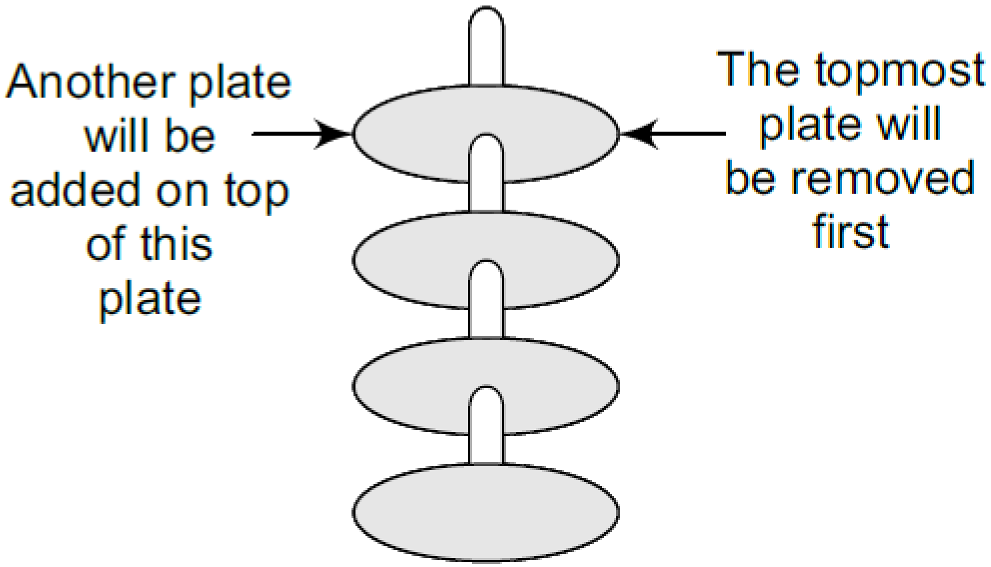
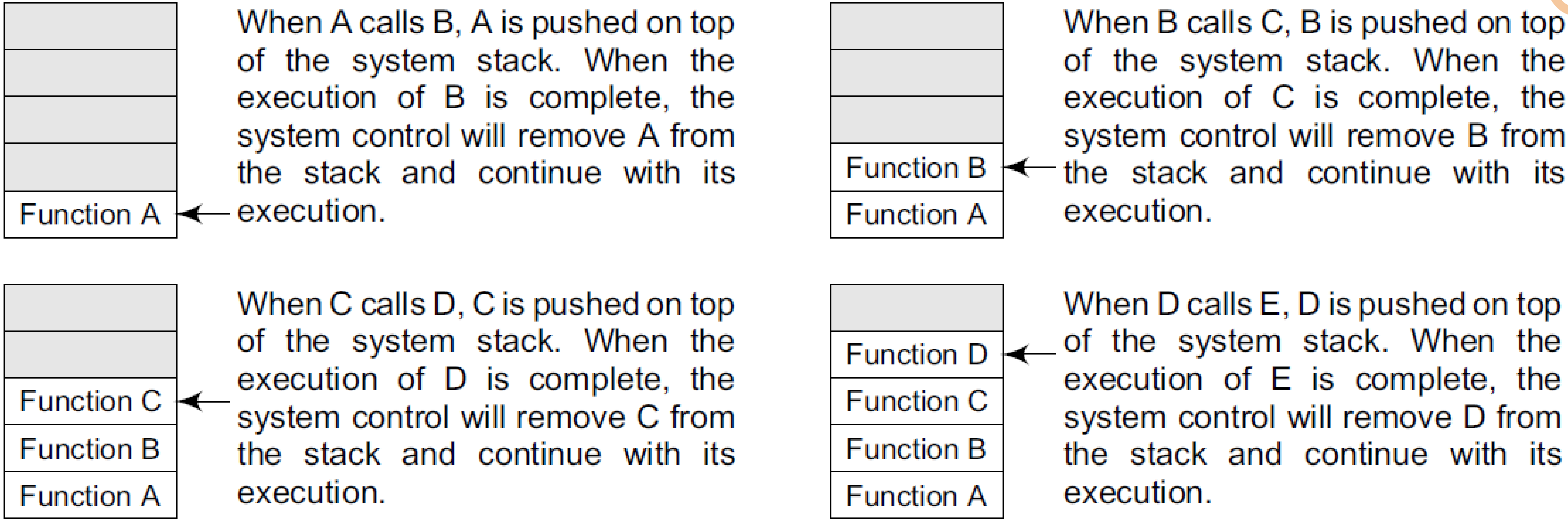

Another plate will be added on top of this plate
The topmost plate will be removed first

Figure 7.1 Stack of plates

- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.
- Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.
- **Stacks can be implemented using either arrays or linked lists**

STACKS IN COMPUTE SCIENCE

- function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.



When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.

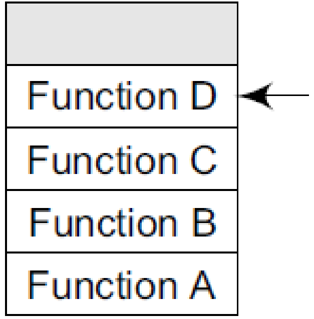
When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.

When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.

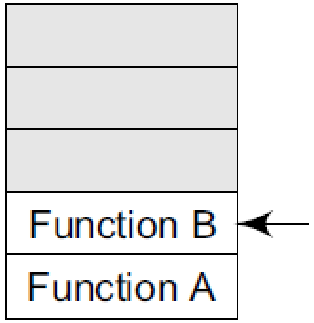
When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Figure 7.2 System stack in the case of function calls

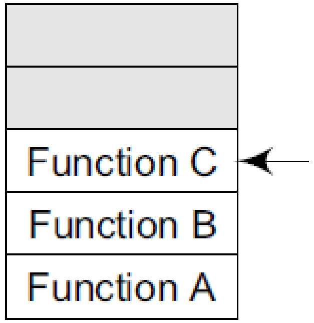
- In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used.
- Whenever a function calls another function, the calling function is pushed onto the top of the stack.
- This is because after the called function gets executed, the control is passed back to the calling function.



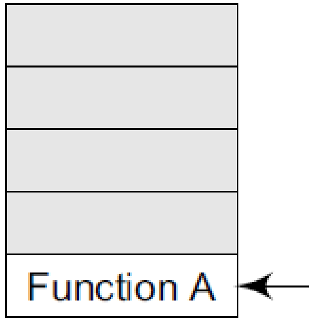
When E has executed, D will be removed for execution.



When C has executed, B will be removed for execution.



When D has executed, C will be removed for execution.



When B has executed, A will be removed for execution.

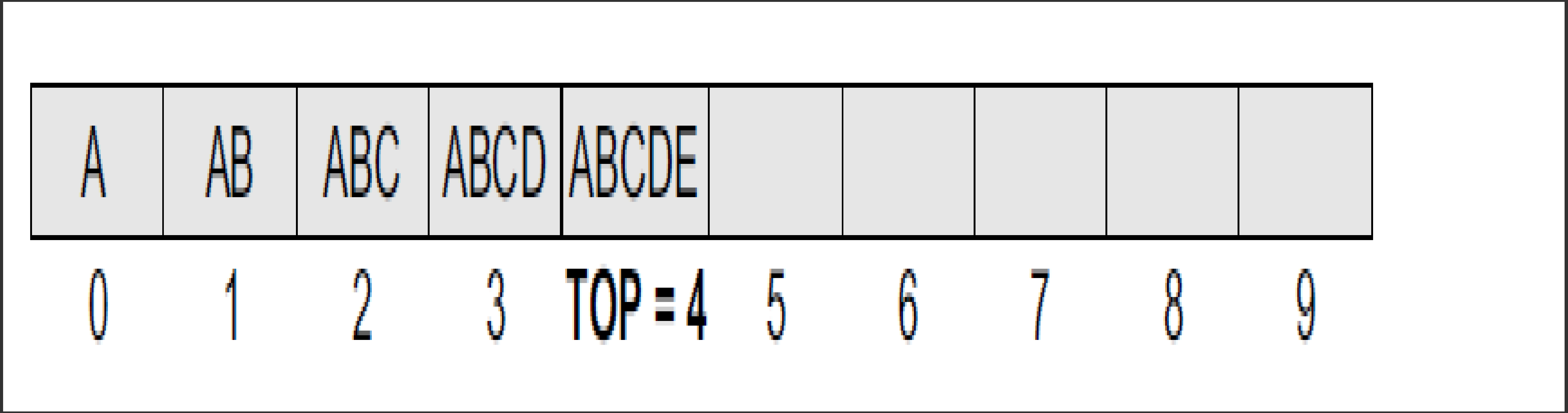
- Now when function E is executed, function D will be removed from the top of the stack and executed.
- Once function D gets completely executed, function C will be removed from the stack for execution.
- The whole procedure will be repeated until all the functions get executed

- The system stack ensures a proper execution order of functions.
- Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled

ARRAY REPRESENTATION OF STACKS

- Stacks can be represented as a linear array.
- Every stack has a variable called `top` associated with it, which is used to store the address of the topmost element of the stack.
- It is this position where the element will be added to or deleted from.
- There is another variable called `max`, which is used to store the maximum number of elements that the stack can hold.

- If `TOP = NULL`, then it indicates that the stack is empty- underflow
- and if `TOP = MAX–1`, then the stack is full- overflow.
- (You must be wondering why we have written `MAX–1`. It is because array indices start from 0.)



A	AB	ABC	ABCD	ABCDE					
0	1	2	3	TOP = 4	5	6	7	8	9

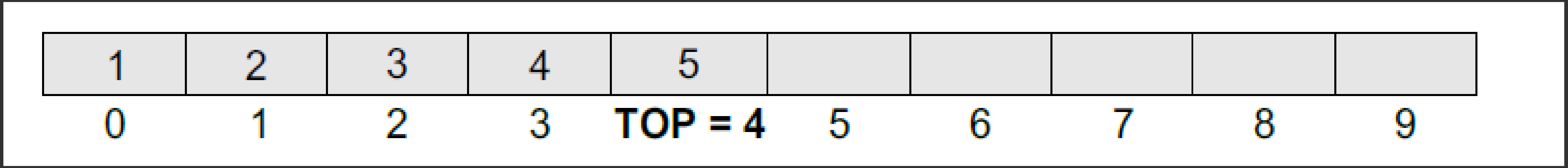
OPERATIONS ON A STACK

- A stack supports three basic operations:
  - push, pop, and peek.
- The push operation adds an element to the top of the stack
- The pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

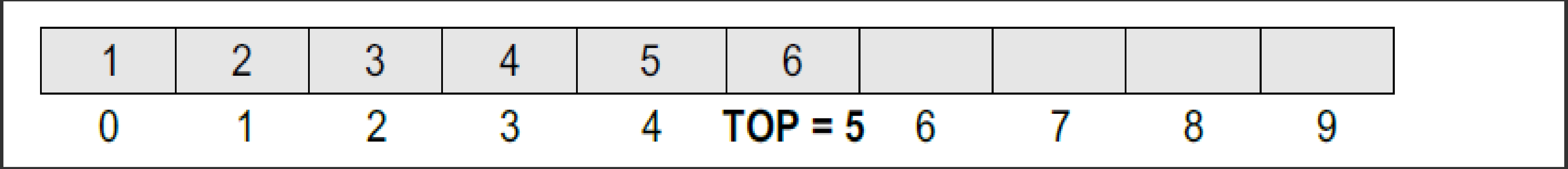
PUSH OPERATION

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- However, before inserting the value, we must first check if `TOP=MAX–1`, because if that is the case, then the stack is full and no more insertions can be done.
- If an attempt is made to insert a value in a stack that is already full, an `OVERFLOW` message is printed.

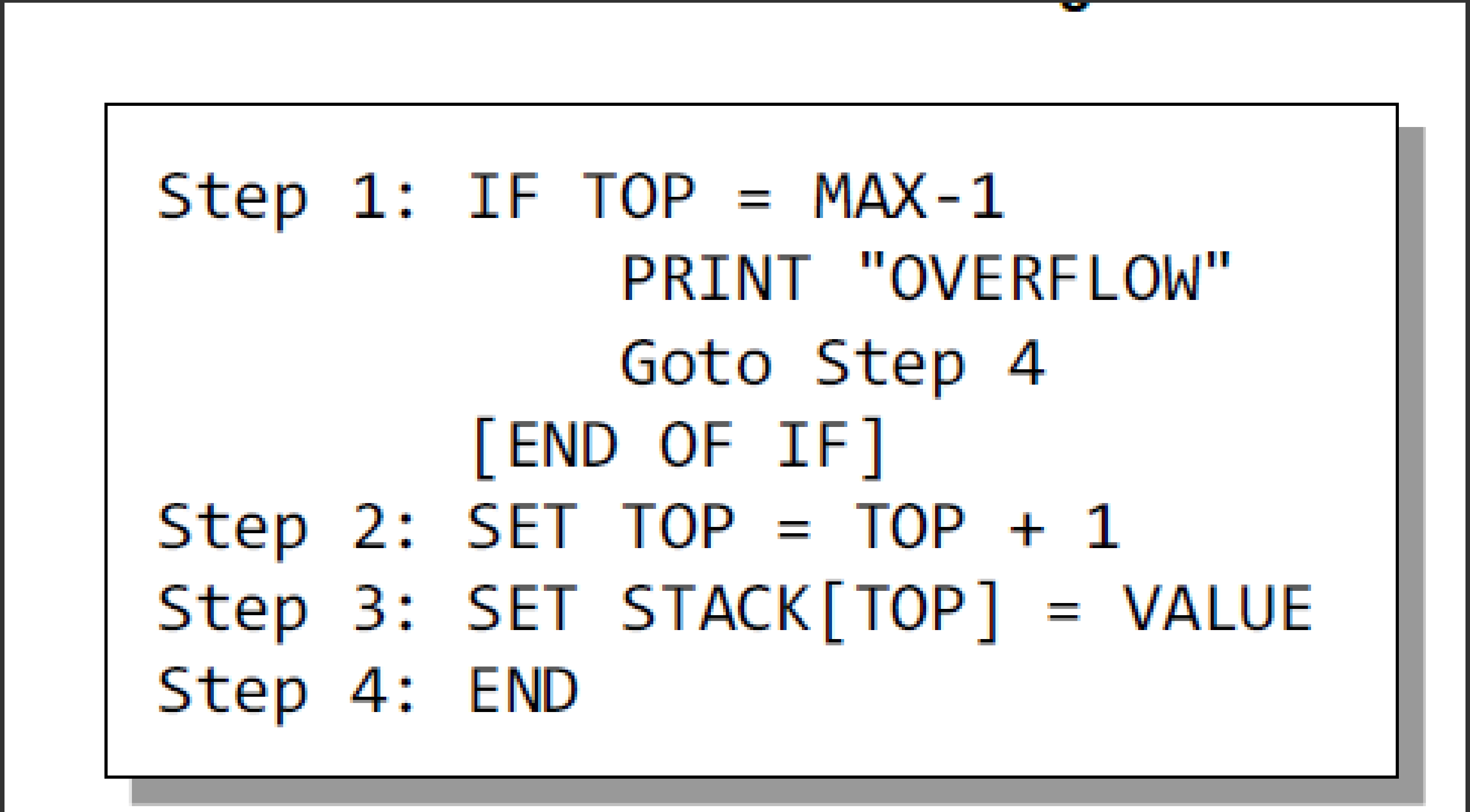
- To insert an element with value 6, we first check if TOP=MAX–1.
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].



1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9



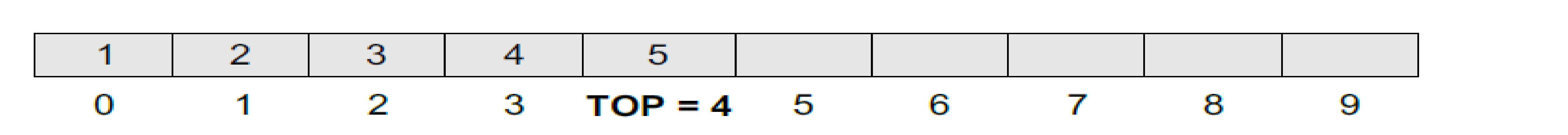
1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9



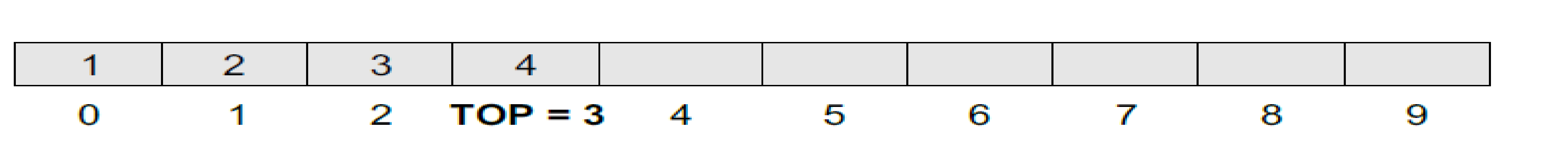
```
Step 1: IF TOP = MAX-1
                PRINT "OVERFLOW"
                Goto Step 4
           [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

POP OPERATION

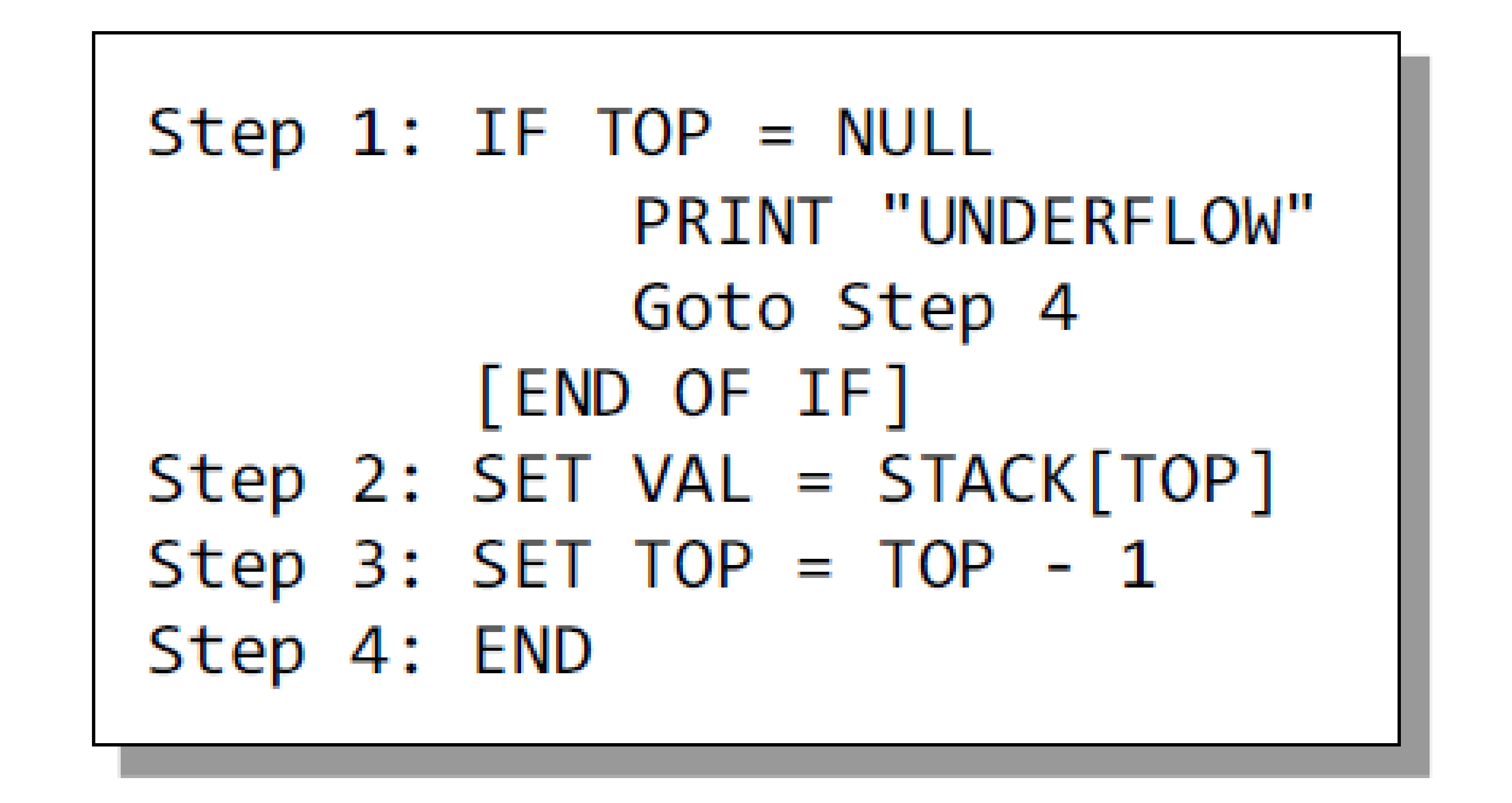
- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.
- To delete the topmost element, we first check if TOP=NULL.
- If the condition is false, then we decrement the value pointed by TOP.



1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9



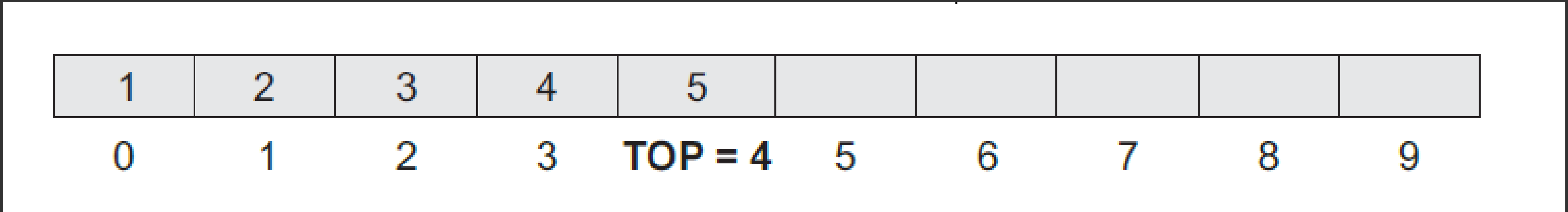
1	2	3	4						
0	1	2	**TOP = 3**	4	5	6	7	8	9



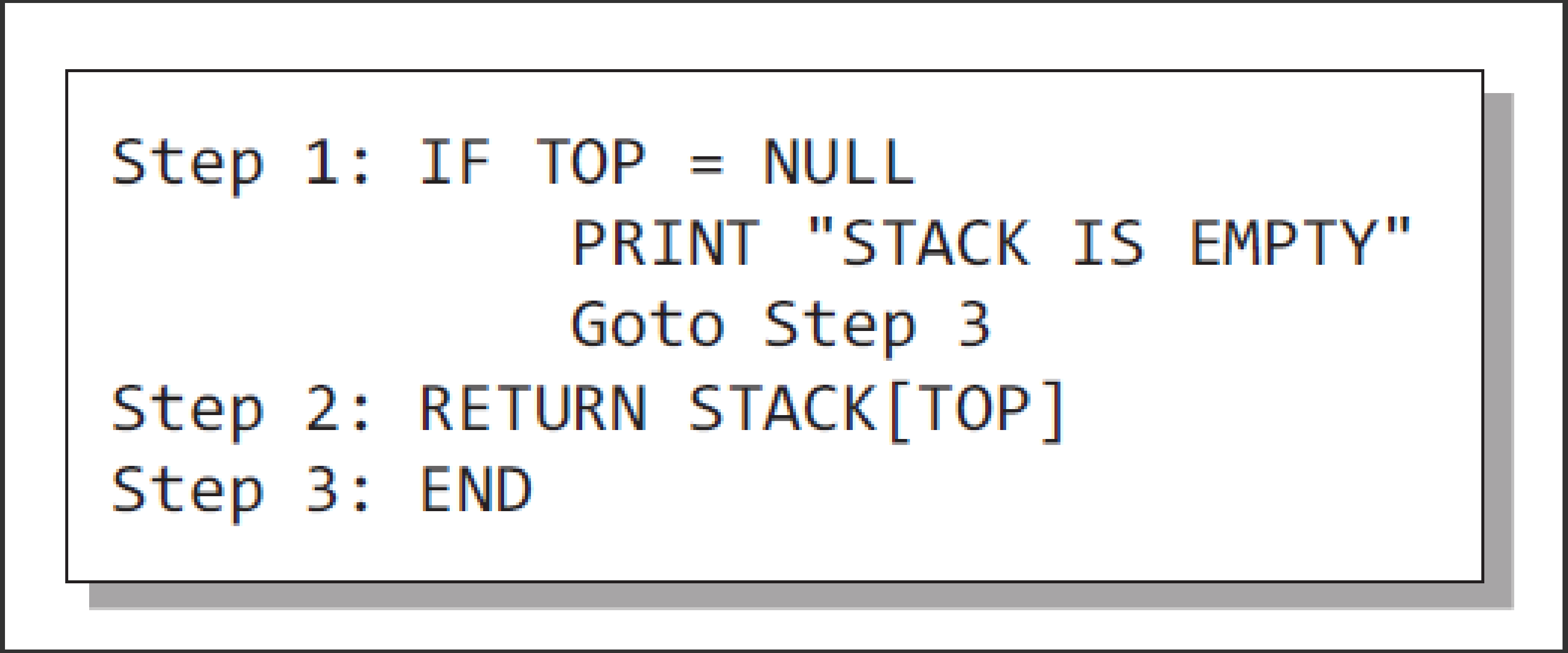
```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

PEEK OPERATION

- `Peek` is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the `Peek` operation first checks if the stack is empty, i.e., if `TOP = NULL`, then an appropriate message is printed, else the value is returned.



1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9



```
Step 1: IF TOP = NULL
              PRINT "STACK IS EMPTY"
              Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Thank You

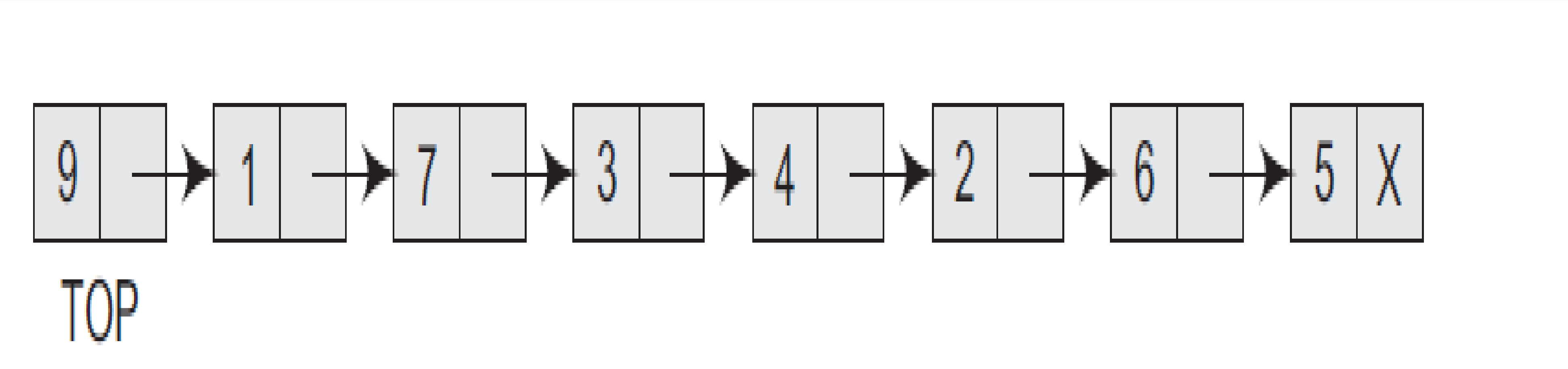
STACKS USING LINKED LIST

PREPARED BY
DEEPTHI M PISHARODY
ASST PROFESSOR
PRAJYOTI NIKETAN COLLEGE, PUDUKAD

LINKED REPRESENTATION OF STACKS

- A stack is created using an array.
- This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.
- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- But if the array size cannot be determined in advance,then the other alternative, i.e., Linked representation, is used.

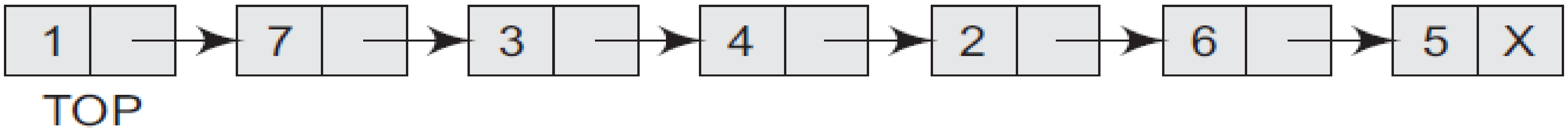
- The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the node pointed by TOP.

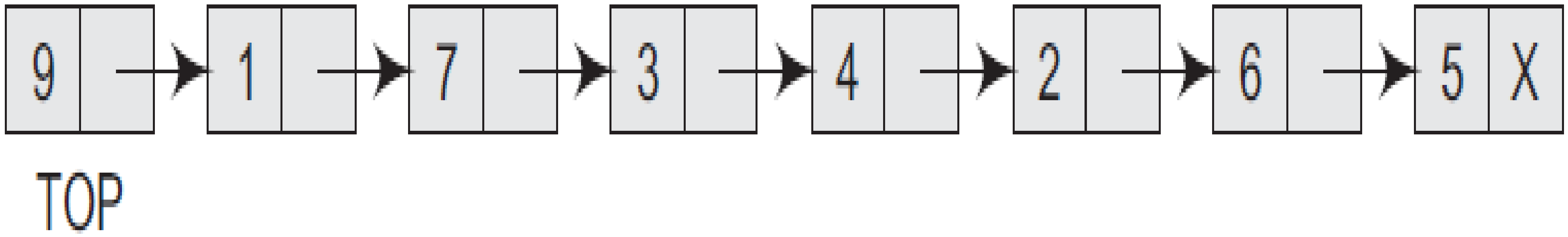

9
1
7
3
4
2
6
5
X
TOP

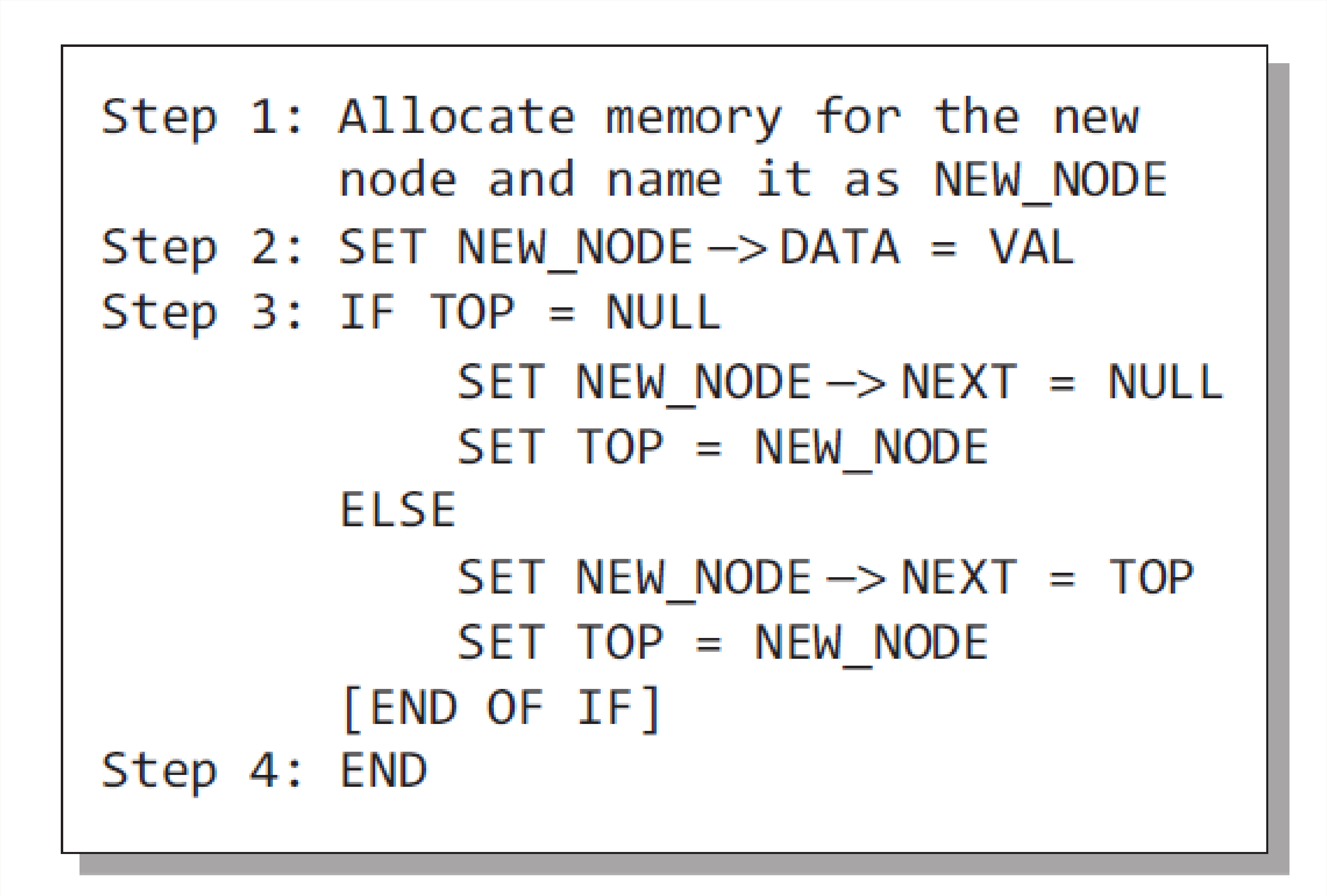
OPERATIONS ON A LINKED STACK

- **Push Operation**
- The `push` operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack

- To insert an element with value 9, we first check if `TOP=NULL`.
- If this is the case, then we allocate memory for a new node, store the value in its `DATA` part and `NULL` in its `NEXT` part.
- The new node will then be called `TOP`.
- However, if `TOP!=NULL`, then we insert the new node at the beginning of the linked stack and name this new node as `TOP`.


1
7
3
4
2
6
5
X
TOP


9
1
7
3
4
2
6
5
X
TOP

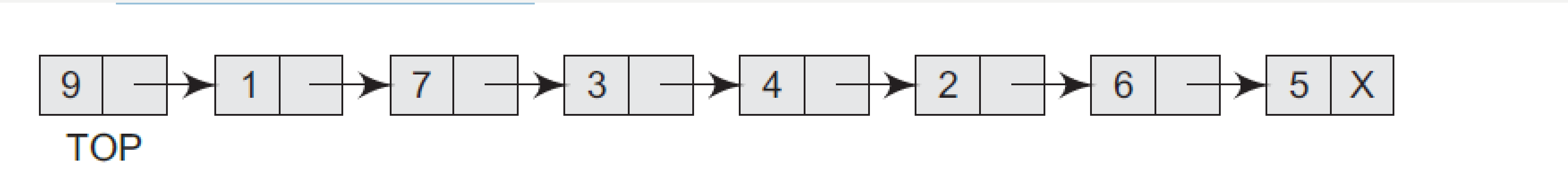


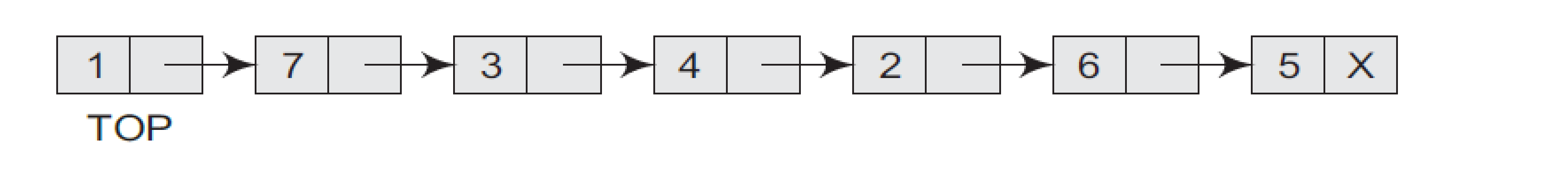
```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

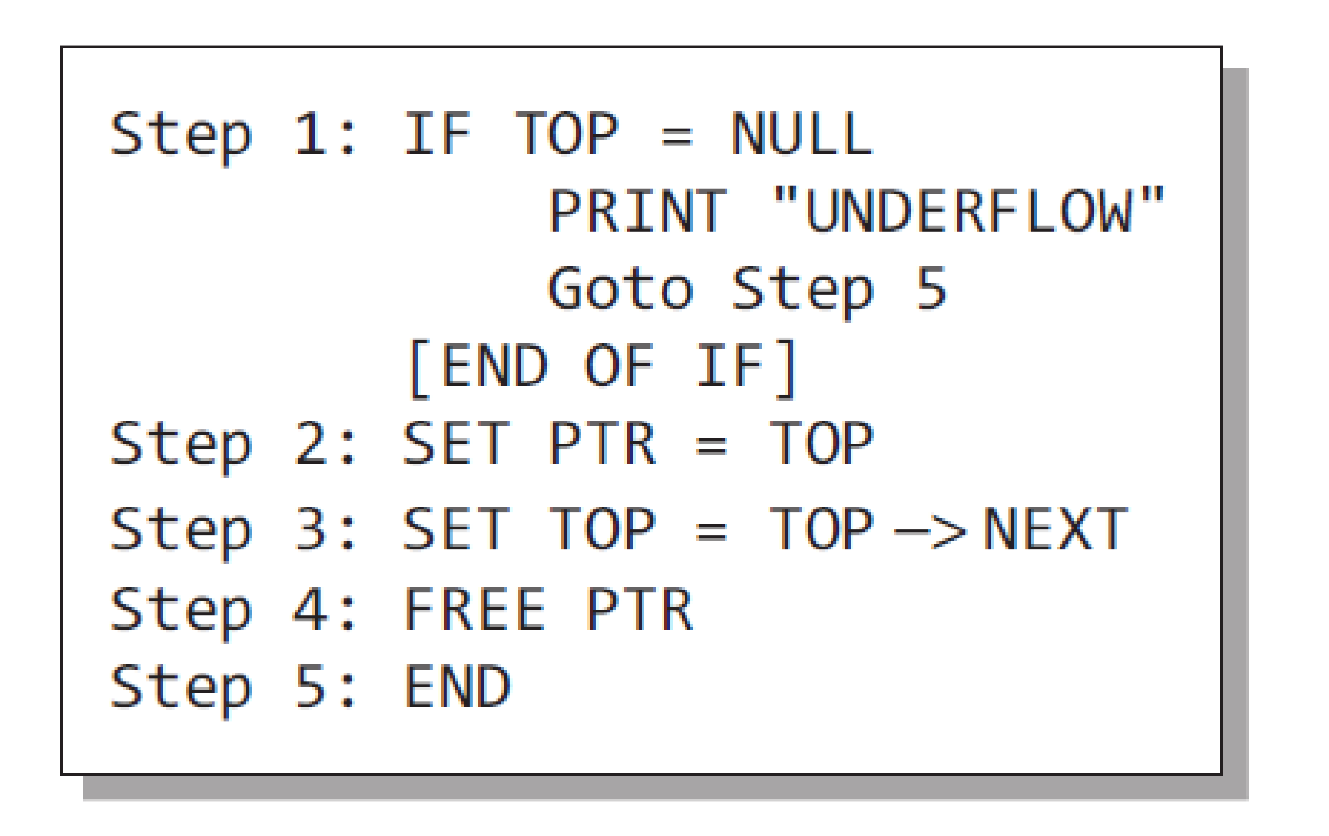
- In Step 1, memory is allocated for the new node.
- In Step 2, the DATA part of the new node is initialized with the value to be stored in the node.
- In Step 3, we check if the new node is the first node of the linked list.
- This is done by checking if TOP = NULL.
- In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP.
- However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

POP OPERATION

- The `pop` operation is used to delete the topmost element from a stack.
- However, before deleting the value, we must first check if `TOP=NULL`, because if this is the case, then it means that the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an `UNDERFLOW` message is printed.


9
1
7
3
4
2
6
5
X
TOP


1
7
3
4
2
6
5
X
TOP



```
Step 1: IF TOP = NULL
                PRINT "UNDERFLOW"
                Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

- In case `TOP!=NULL`, then we will delete the node pointed by `TOP`, and make `TOP` point to the second element of the linked stack.
- In Step 1, we first check for the `UNDERFLOW` condition.
- In Step 2, we use a pointer `PTR` that points to `TOP`.
- In Step 3, `TOP` is made to point to the next node in sequence.
- In Step 4, the memory occupied by `PTR` is given back to the free pool.

Thank you

APPLICATIONS OF STACK

PREPARED BY

MS DEEPTHI M PISHARODY

ASST PROFESSOR

PRAJYOTI NIKETAN COLLEGE, PUDUKAD

APPLICATIONS OF STACK

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

REVERSING A LIST

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack.
- Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

```
int stk[10];

int top=-1;

int pop();

void push(int);
```

```c
void main()
{
int val, n, i,
arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements of the array : ");
for(i=0;i<n;i++)
scanf("%d", &arr[i]);
for(i=0;i<n;i++)
push(arr[i]);
for(i=0;i<n;i++)
{
val = pop();
arr[i] = val;
}
printf("\n The reversed array is : ");
for(i=0;i<n;i++)
printf("\n %d", arr[i]);
getche"();
}
```

```
void push(int val)
{
stk[++top] = val;
}
int pop()
{
return(stk[top--]);
}
```

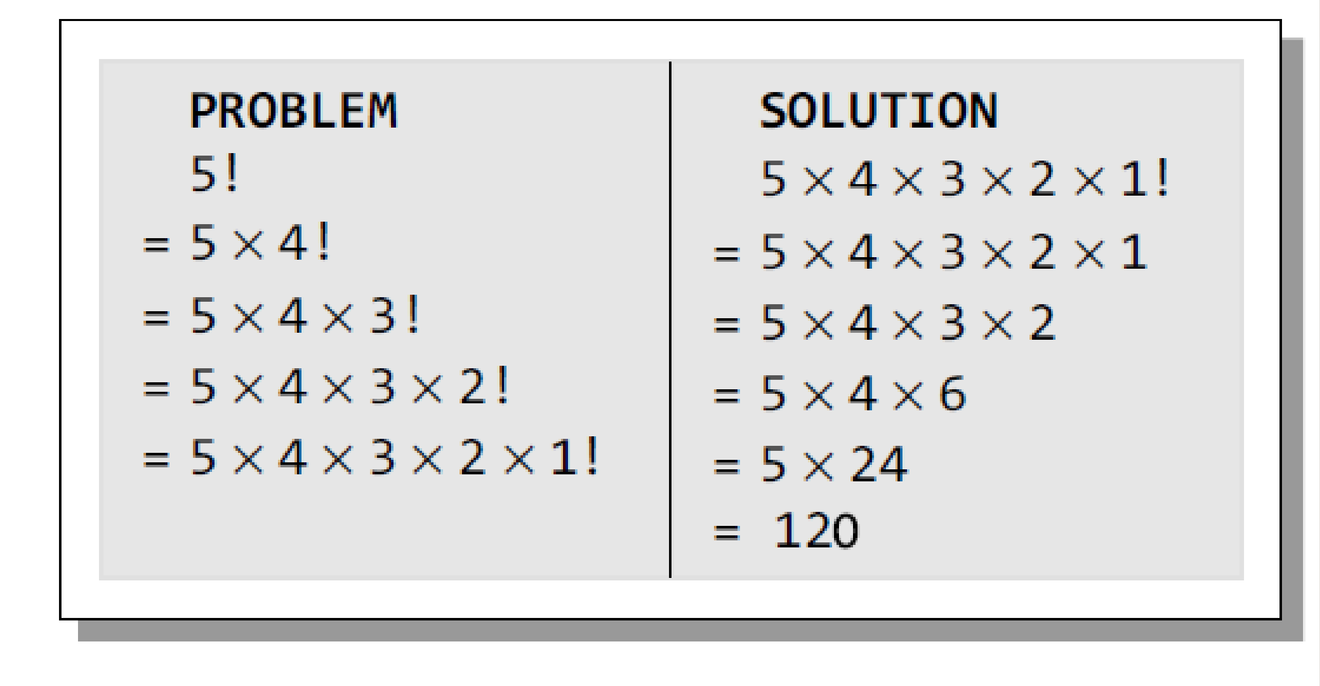
Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 1 2 3 4 5
The reversed array is : 5 4 3 2 1
```

RECURSION

- A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases.

- *Base case*, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- *Recursive case*, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts

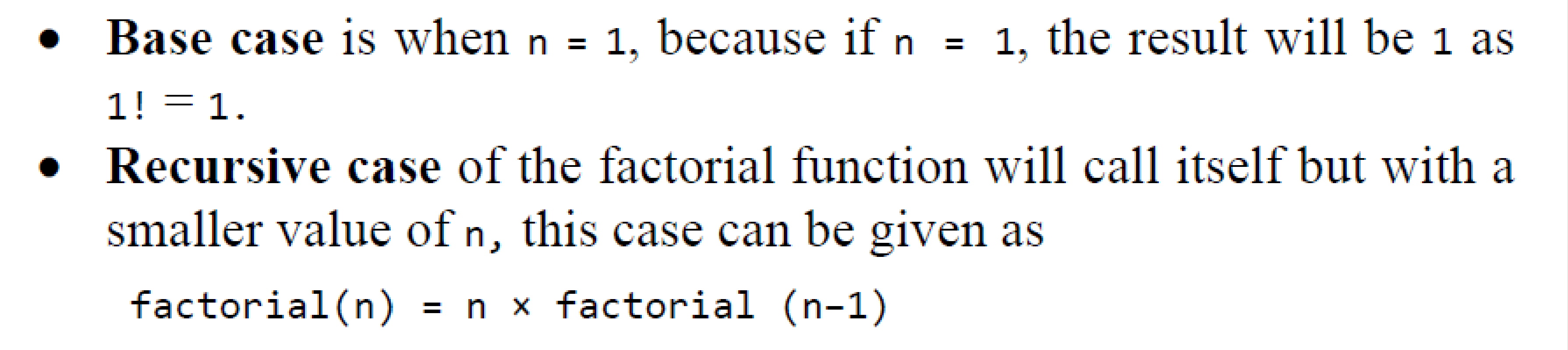


PROBLEM

$$\begin{aligned} & 5! \\ = {} & 5 \times 4! \\ = {} & 5 \times 4 \times 3! \\ = {} & 5 \times 4 \times 3 \times 2! \\ = {} & 5 \times 4 \times 3 \times 2 \times 1! \end{aligned}$$

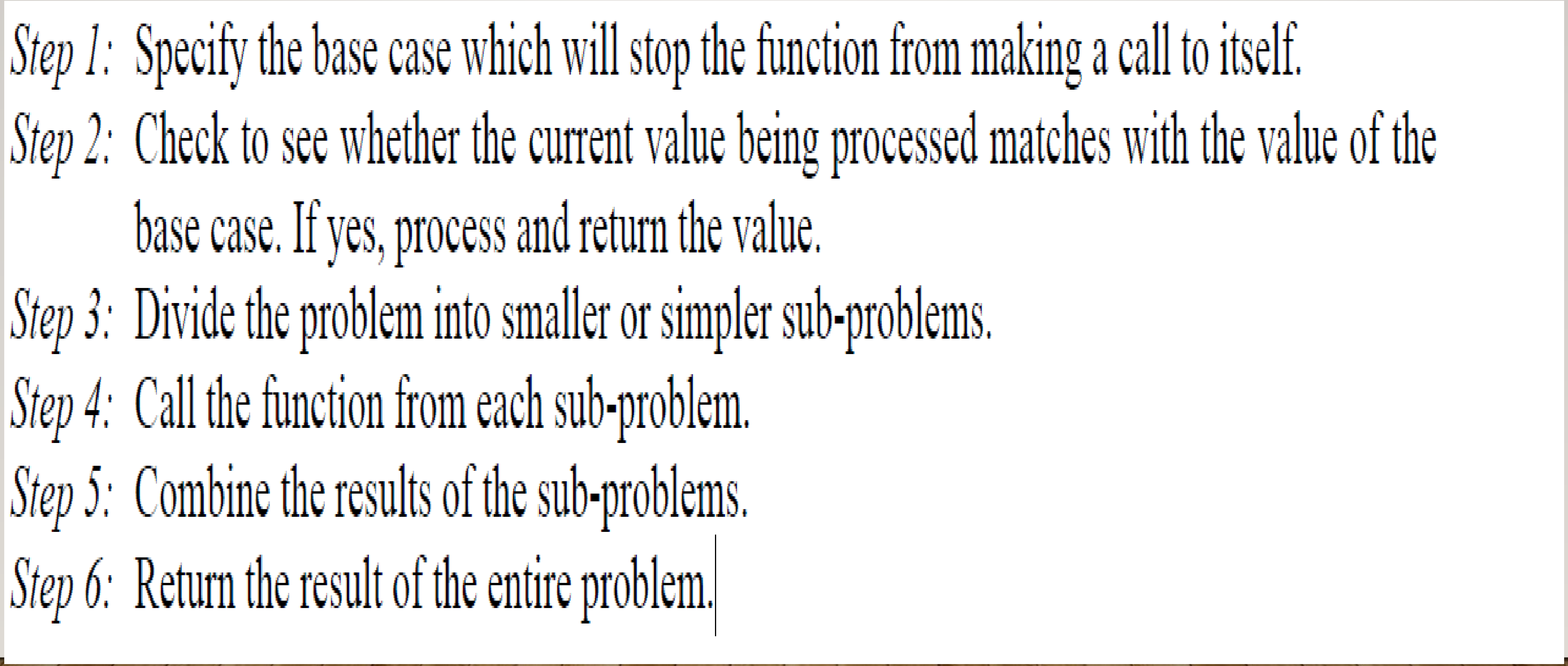
SOLUTION

$$\begin{aligned} & 5 \times 4 \times 3 \times 2 \times 1! \\ = {} & 5 \times 4 \times 3 \times 2 \times 1 \\ = {} & 5 \times 4 \times 3 \times 2 \\ = {} & 5 \times 4 \times 6 \\ = {} & 5 \times 24 \\ = {} & 120 \end{aligned}$$



- **Base case** is when `n = 1`, because if `n = 1`, the result will be `1` as `1! = 1.`
- **Recursive case** of the factorial function will call itself but with a smaller value of `n`, this case can be given as

```
 factorial(n) = n × factorial (n-1)
```



Step 1: Specify the base case which will stop the function from making a call to itself.

Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.

Step 3: Divide the problem into smaller or simpler sub-problems.

Step 4: Call the function from each sub-problem.

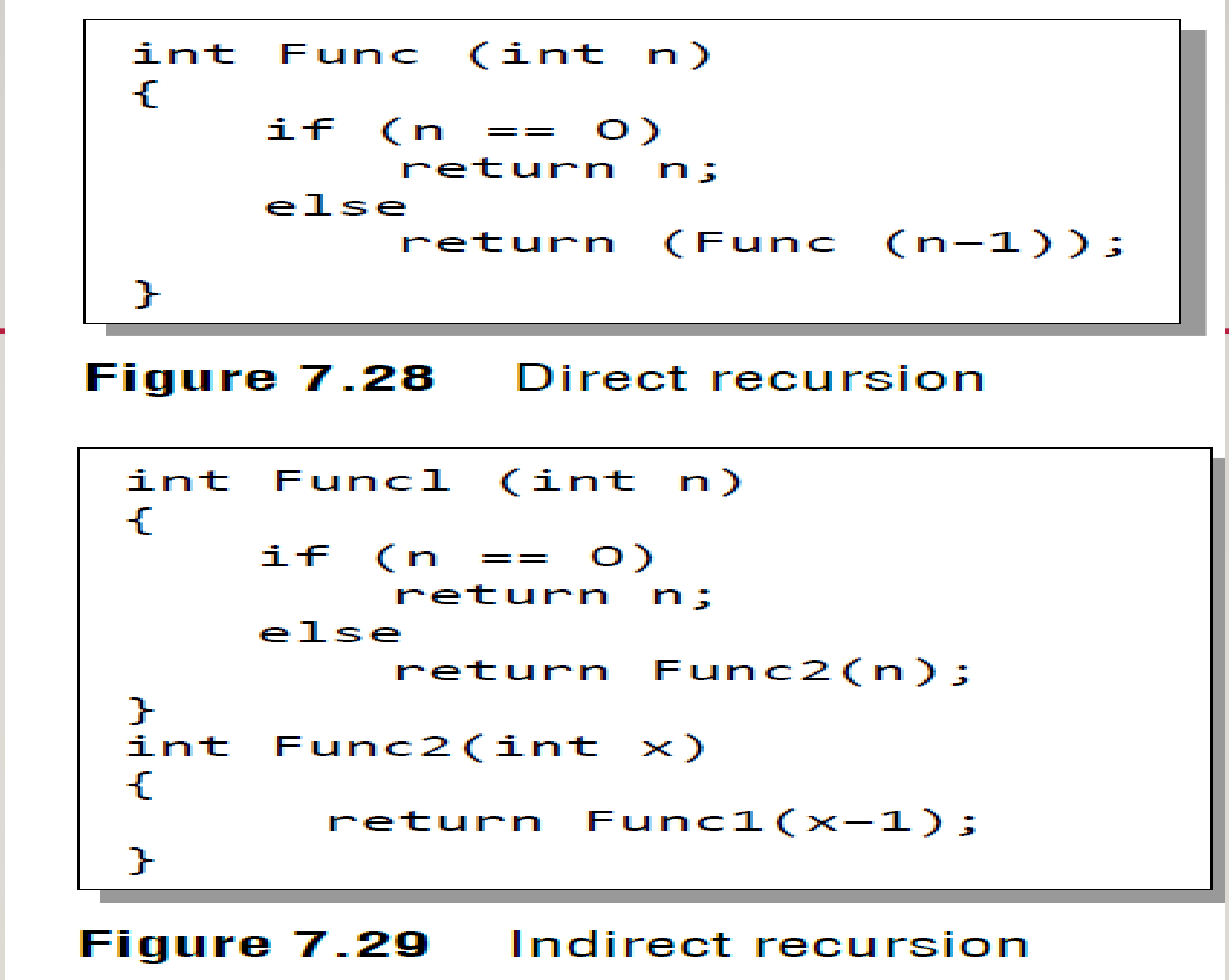
Step 5: Combine the results of the sub-problems.

Step 6: Return the result of the entire problem.

TYPES OF RECURSION

- Any recursive function can be characterized based on:
- whether the function calls itself directly or indirectly (*direct or indirect recursion*),
- whether any operation is pending at each recursive call (*tailrecursive* or not), and
- the structure of the calling pattern (*linear or tree-recursive*).

- *Direct Recursion*
- A function is said to be *directly* recursive if it explicitly calls itself.
- Here, the function `Func()` calls itself for all positive values of `n,` so it is said to be a directly recursive function.
- *Indirect Recursion*
- A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. These two functions are indirectly recursive as they both call each other



```
int Func (int n)
{
      if (n == 0)
            return n;
      else
            return (Func (n-1));
}
```

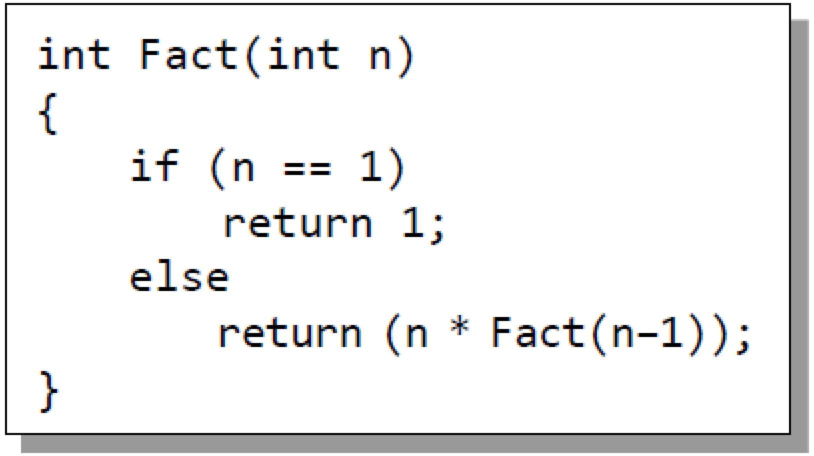
Figure 7.28 Direct recursion

```
int Funcl (int n)
{
      if (n == 0)
            return n;
      else
            return Func2(n);
}
int Func2(int x)
{
         return Func1(x-1);
}
```

Figure 7.29 Indirect recursion

TAIL RECURSION

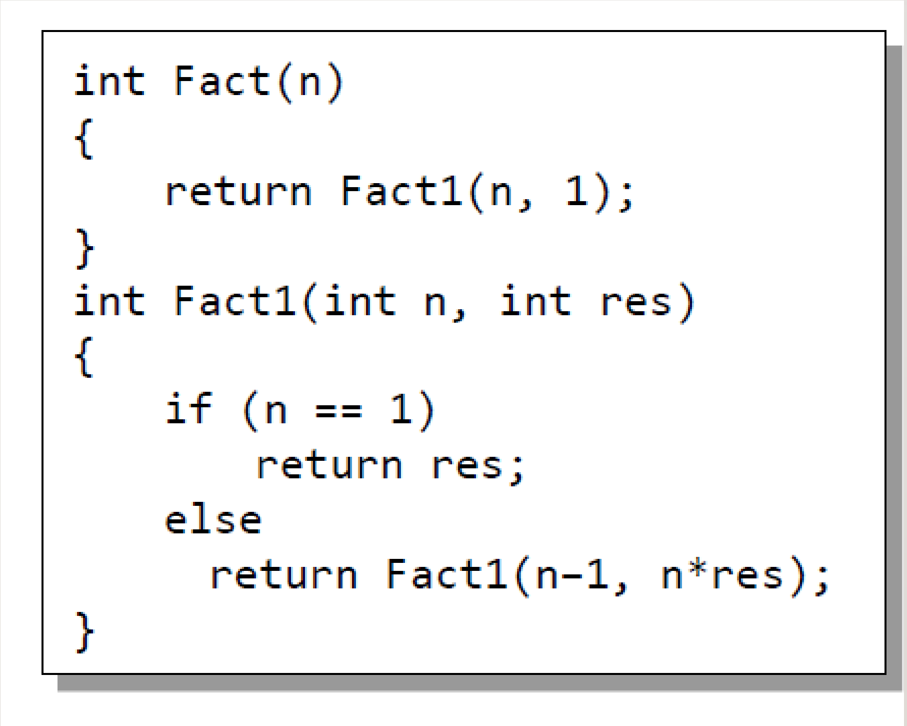
- A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.
- when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.



```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Figure 7.30 Non-tail recursion

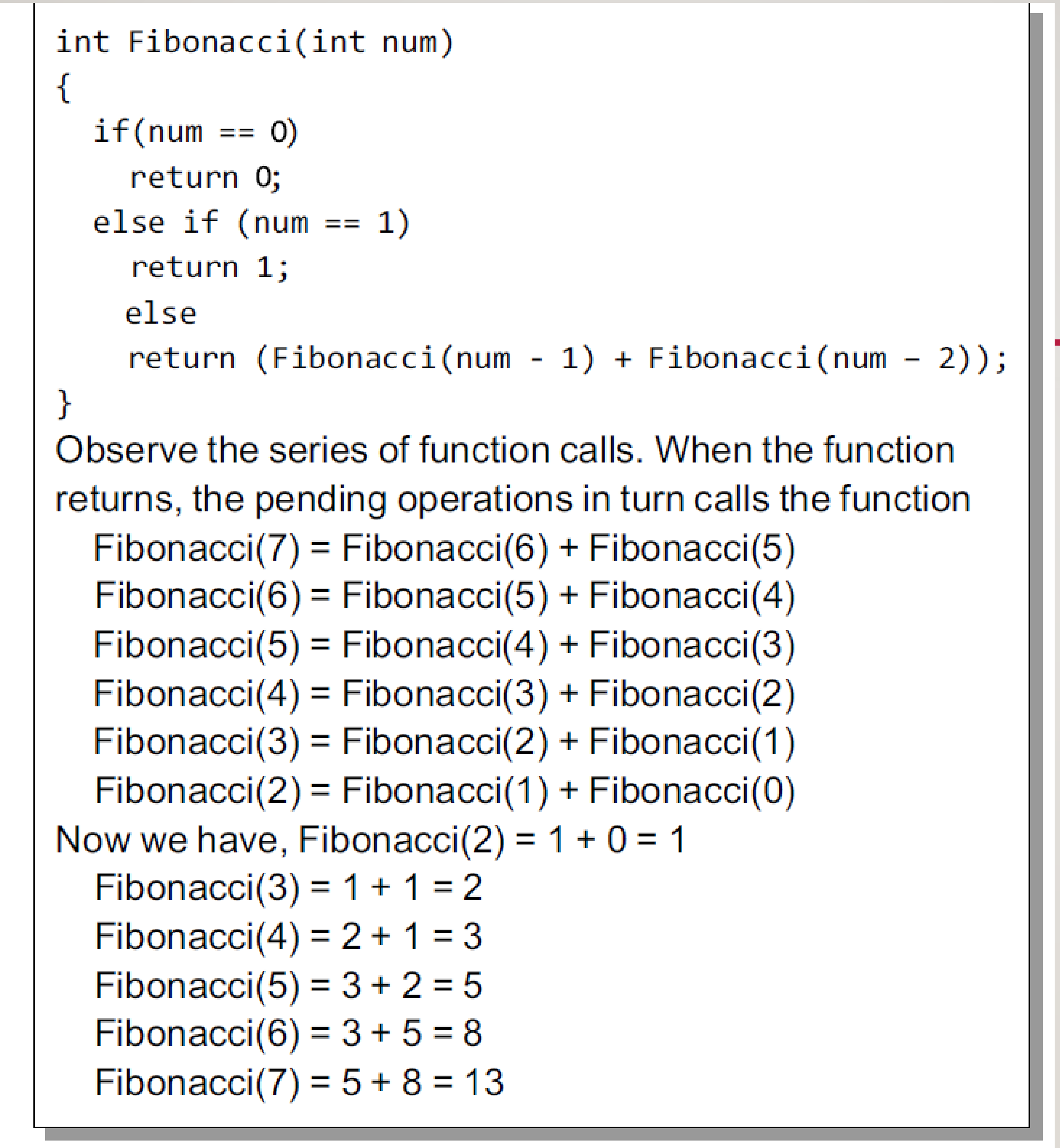
- Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number of calls.



```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

- In simple words, a recursive function is said to be *linearly* recursive when the pending operation (if any) does not make another recursive call to the function.
- For example, observe the last line of recursive factorial function.
- The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to `Fact`.

- On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly*recursive) if the pending operation makes another recursive call to the function.
- For example, the `Fibonacci` function in which the pending operations recursively call the `Fib onacci` function.



```
int Fibonacci(int num)
{
  if(num == 0)
    return 0;
  else if (num == 1)
    return 1;
    else
    return (Fibonacci(num - 1) + Fibonacci(num – 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function

  Fibonacci(7) = Fibonacci(6) + Fibonacci(5)
  Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
  Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
  Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
  Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
  Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

  Fibonacci(3) = 1 + 1 = 2
  Fibonacci(4) = 2 + 1 = 3
  Fibonacci(5) = 3 + 2 = 5
  Fibonacci(6) = 3 + 5 = 8
  Fibonacci(7) = 5 + 8 = 13

Thank you

Arithmetic expressions

Prepared by
Ms Deepthi M Pisharody
Asst Professor
Prajyoti Niketan College, Pudukad

Applications of stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Arithmetic expressions

- An arithmetic expressions consists of operators and operands.
- Operands are numeric values or numeric constants
- Operators are used in arithmetic expression.

Various notations for arithmetic expressions

- Prefix
- Postfix
- Infix

Infix notation

- The operator is placed in between the operands.
- a+b
- a*b
- a+b-c

- Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression.
- Information is needed about operator precedence and associativity rules, and brackets which override these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix notation

- *Postfix notation* was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher.
- His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN

Postfix notation,

- In postfix notation, as the name suggests, the operator is placed after the operands.
- Eg: AB+
- AB*
- AB-
- The order of evaluation of a postfix expression is always from left to right.
- Even brackets cannot alter the order of evaluation

- A postfix operation does not even follow the rules of operator precedence.
- The operator which occurs first in the expression is operated first on the operands.

Prefix notation

- In prefix notation, the operator is placed before the operands.
- +AB
- -AB
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

Conversion of an Infix Expression into a Prefix Expression

- **(A + B) * C**

(+AB)*C

*+ABC

- **(A–B) * (C+D)**

[–AB] * [+CD]

*–AB+CD

Conversion of an Infix Expression into a Postfix Expression

- **(A + B) * C**

(AB+)*C

AB+C*

- **(A–B) * (C+D)**

[AB-] * [CD+]

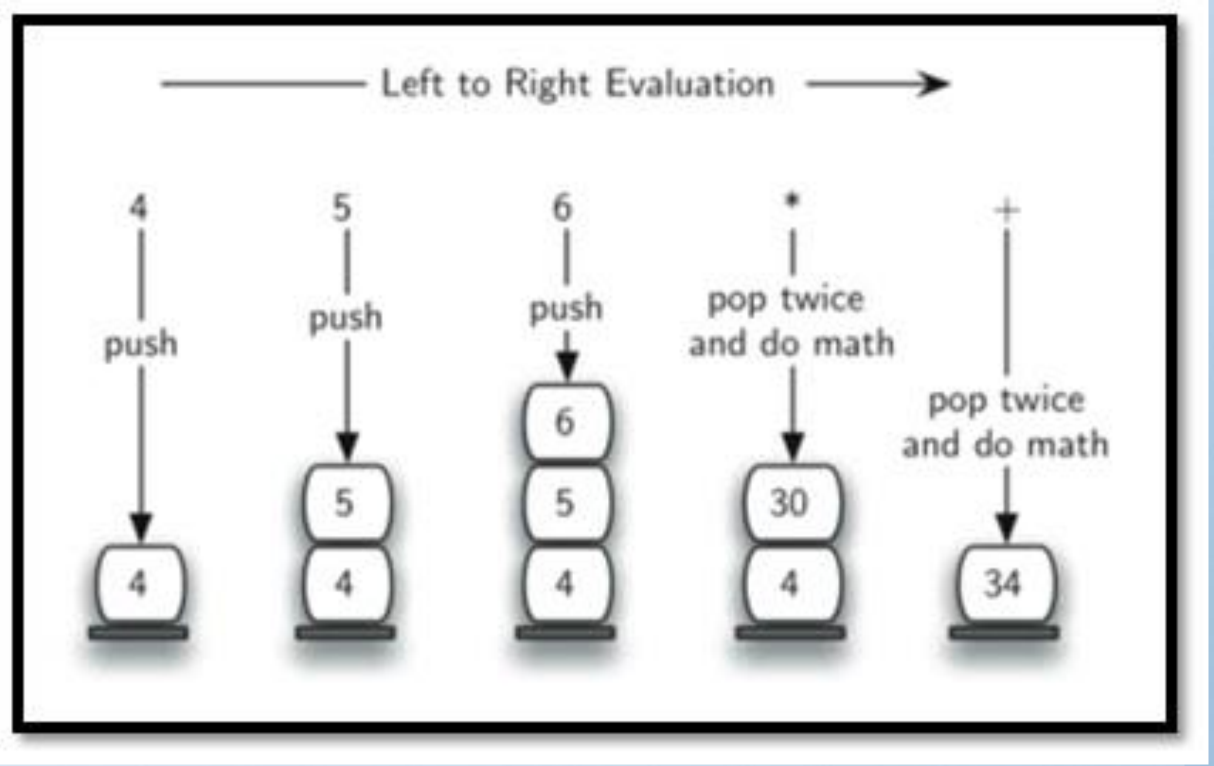
AB-CD+*

Evaluation of a Postfix Expression

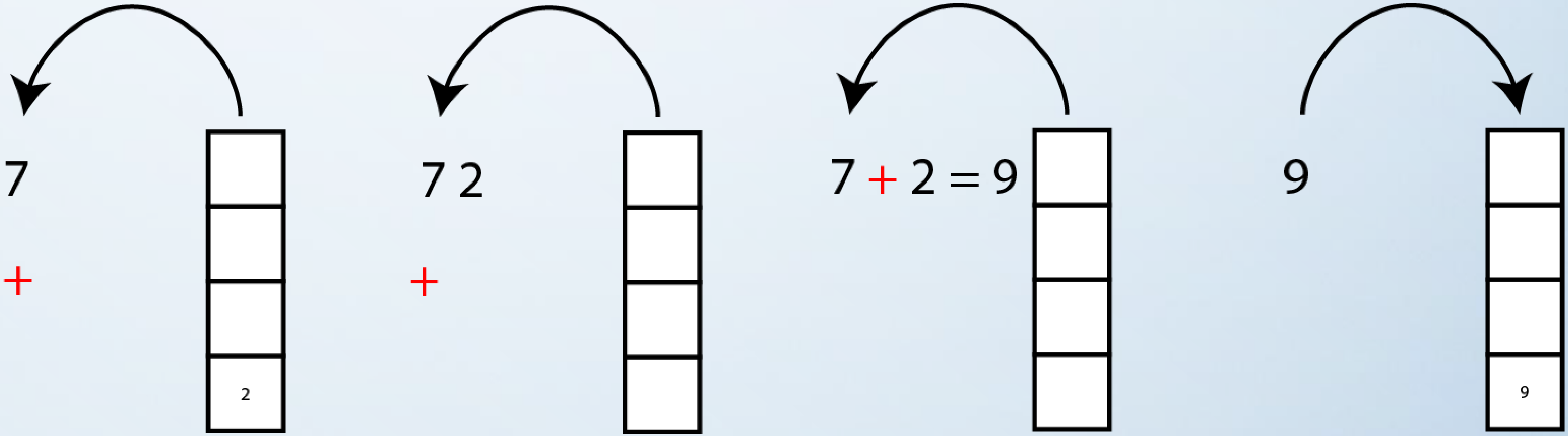
- Using stacks, any postfix expression can be evaluated very easily.
- Every character of the postfix expression is scanned from left to right.
- If the character encountered is an operand, it is pushed on to the stack.
- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values.
- The result is then pushed on to the stack.

Example

- Consider a+b*c
- Its corresponding postfix notation is abc*+
- Consider a=4, b=5 and c=6



72+

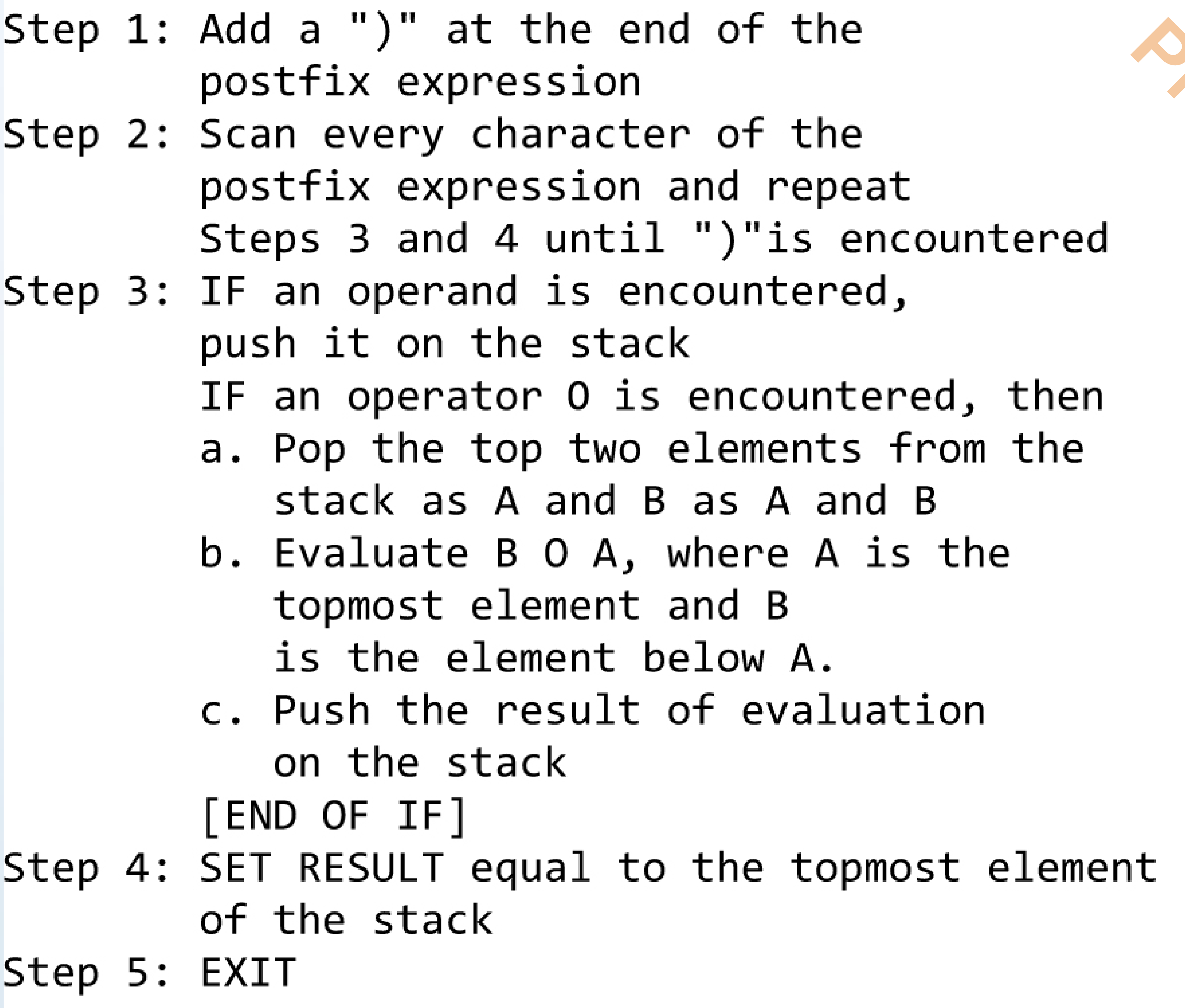


First pop

Second pop

Perform the operator

Push the result onto the stack



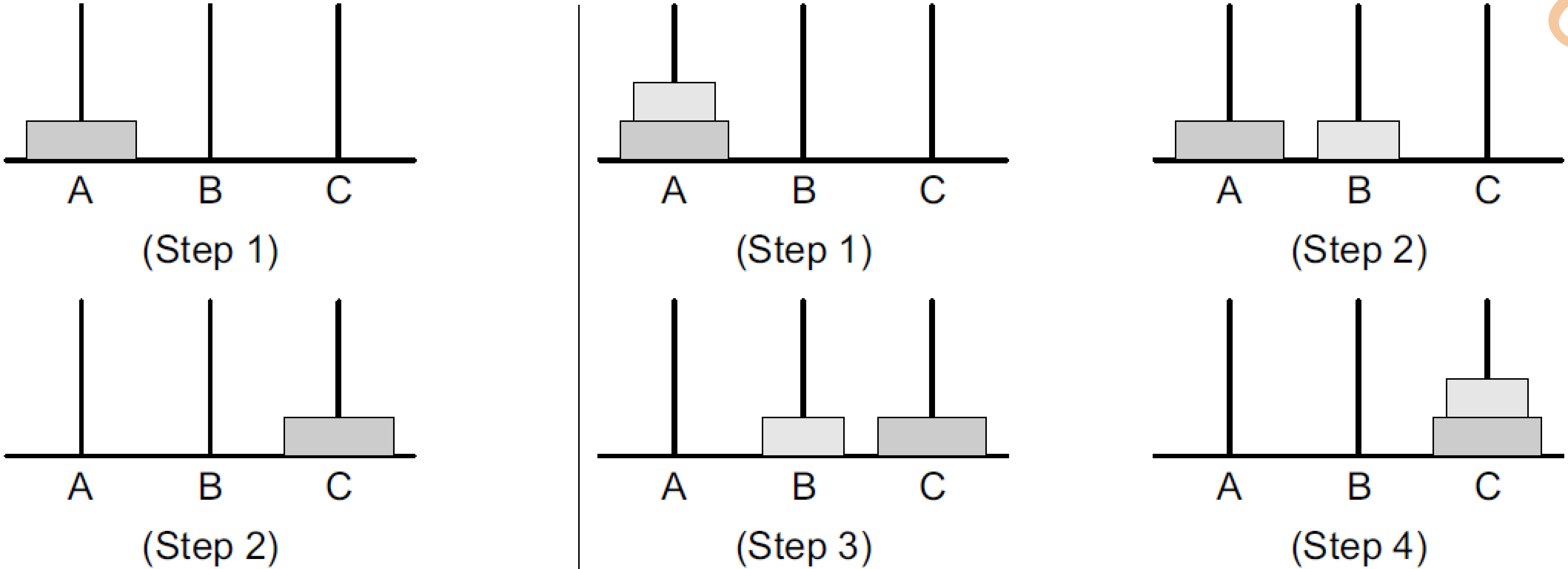
```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

TOWERS OF HANOI

Prepared by
Ms Deepthi M Pisharody
Asst Professor
Prajyoti Niketan College, Pudukad

- The tower of Hanoi is one of the main applications of recursion.
- It says, ‘if you can solve `n-1` cases, then you can easily solve the `nth` case’.

- three rings mounted on pole A.
- The problem is to move all these rings from pole A to pole C while maintaining the same order.
- The main issue is that the smaller disk must always come above the larger disk.



(Step 1)

(Step 1)

(Step 2)

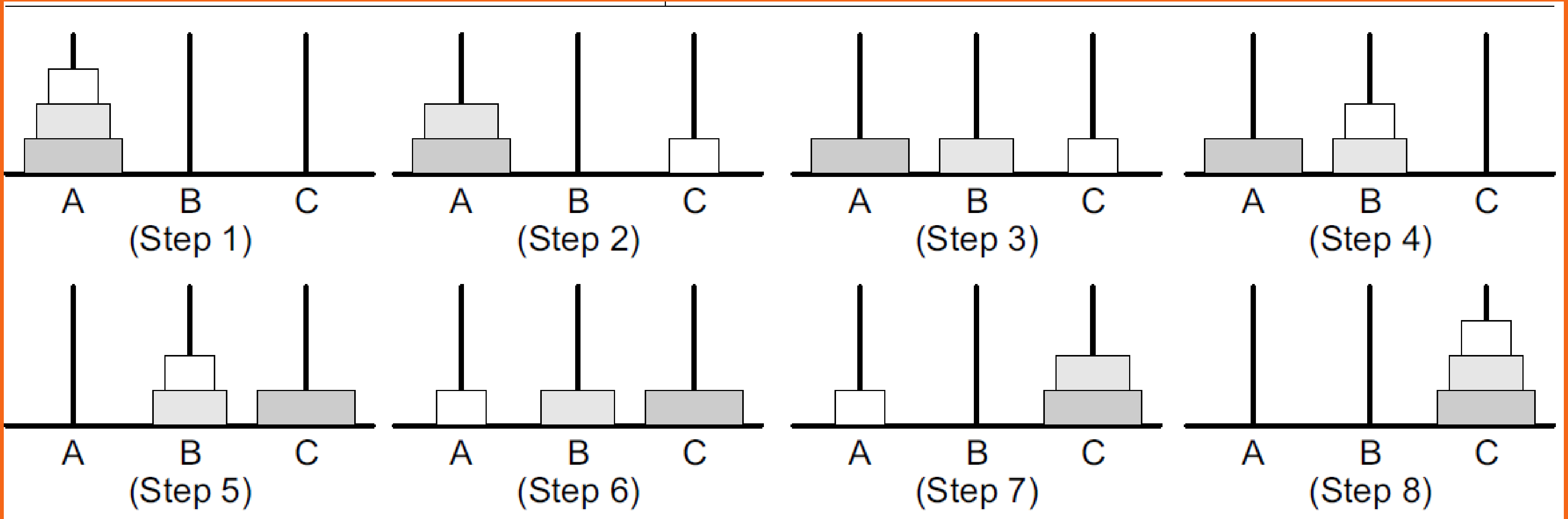
(Step 2)

(Step 3)

(Step 4)

(If there is only one ring, then simply move the ring from source to the destination.)

(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)



(Consider the working with three rings.)

- We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole.
- To transfer all the three rings from A to C, we will first shift the upper two rings `(n-1 rings)` from the source pole to the spare pole.
- We move the first two rings from pole A to B.
- Now that `n-1` rings have been removed from pole A, the `nth` ring can be easily moved from the source pole (A) to the destination pole (C).
- The final step is to move the `n-1` rings from the spare pole (B) to the destination pole (C).

- **Base case: `if n=1`**
- Move the ring from A to C using B as spare
- **Recursive case:**
- Move `n` – `1` rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move `n` – `1` rings from B to C using A as spare

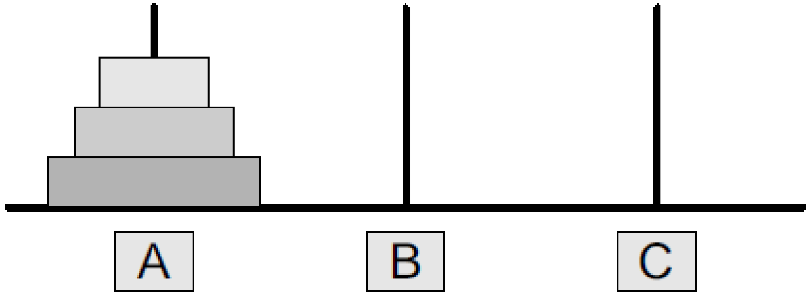


Figure 7.33 Tower of Hanoi

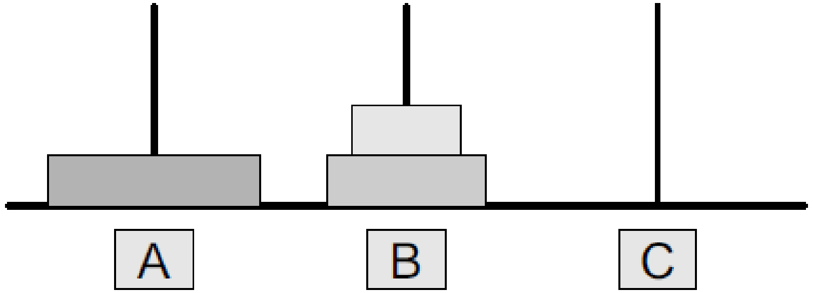


Figure 7.34 Move rings from A to B

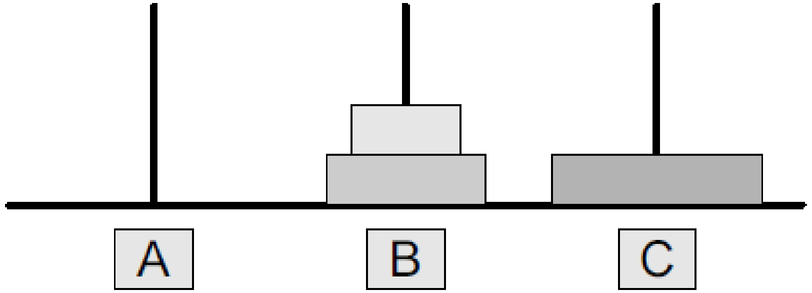


Figure 7.35 Move ring from A to C

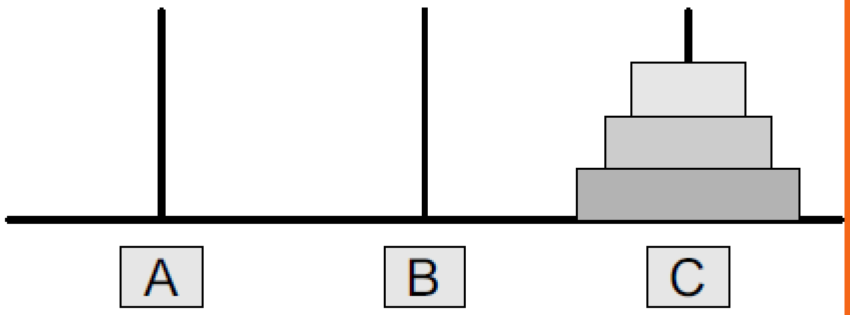


Figure 7.36 Move ring from B to C

```
#include <stdio.h>
int main()
{
int n;
printf("\n Enter the number of rings: ");
scanf("%d", &n);
move(n,'A', 'C', 'B');
return 0;
}
```

```
void move(int n, char source, char dest, char spare)
{
if (n==1)
printf("\n Move from %c to %c",source,dest);
else
{
move(n–1,source,spare,dest);
move(1,source,dest,spare);
move(n–1,spare,dest,source);
}
}
```

Thank you

QUEUES IN DATA STRUCTURES

Prepared by

Deepthi M Pisharody

Asst Professor

Dept of Computer Science

Prajyoti Niketan College, Pudukad

References

Seymour Lipschutz, “Data Structures”, Tata McGraw- Hill Publishing Company Limited, Schaum‟s Outlines

YedidyanLangsam, Moshe J. Augenstein, and Aaron M. Tenenbaum, “Data Structures Using C”, Pearson Education., New Delhi.

Data Structures Using C, Reema Thareja

Real life applications

- **People moving on an escalator.**
- **People waiting for a bus.**
- **People standing outside the ticketing window of a cinema hall.**
- **Luggage kept on conveyor belts.**
- **Cars lined at a toll bridge.**

The element at the first position is served first

Queue Data structure

- A queue is a **FIFO (First-In, First-Out)** data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the **REAR**
- Removed from the other end called the **FRONT**.
- Queues can be implemented by using either **arrays or linked lists.**

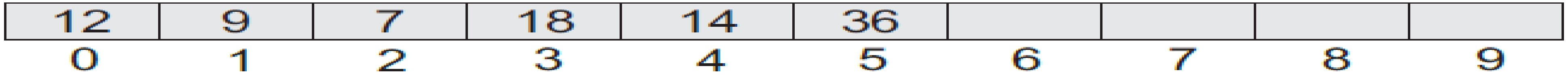
ARRAY REPRESENTATION OF QUEUES

- **Queues can be easily represented using linear arrays.**
- **every queue has `front and rear` variables that point to the position from where deletions and insertions can be done, respectively.**

Operations on Queues

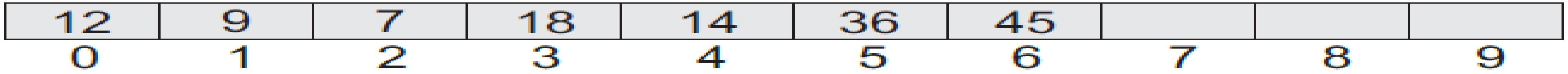
- Every time a new element has to be added, REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.

- If we want to delete an element from the queue, then the value of FRONT will be incremented



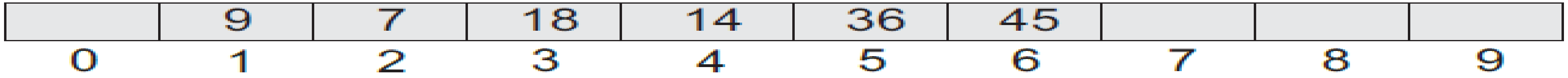
12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Figure 8.1 Queue



12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

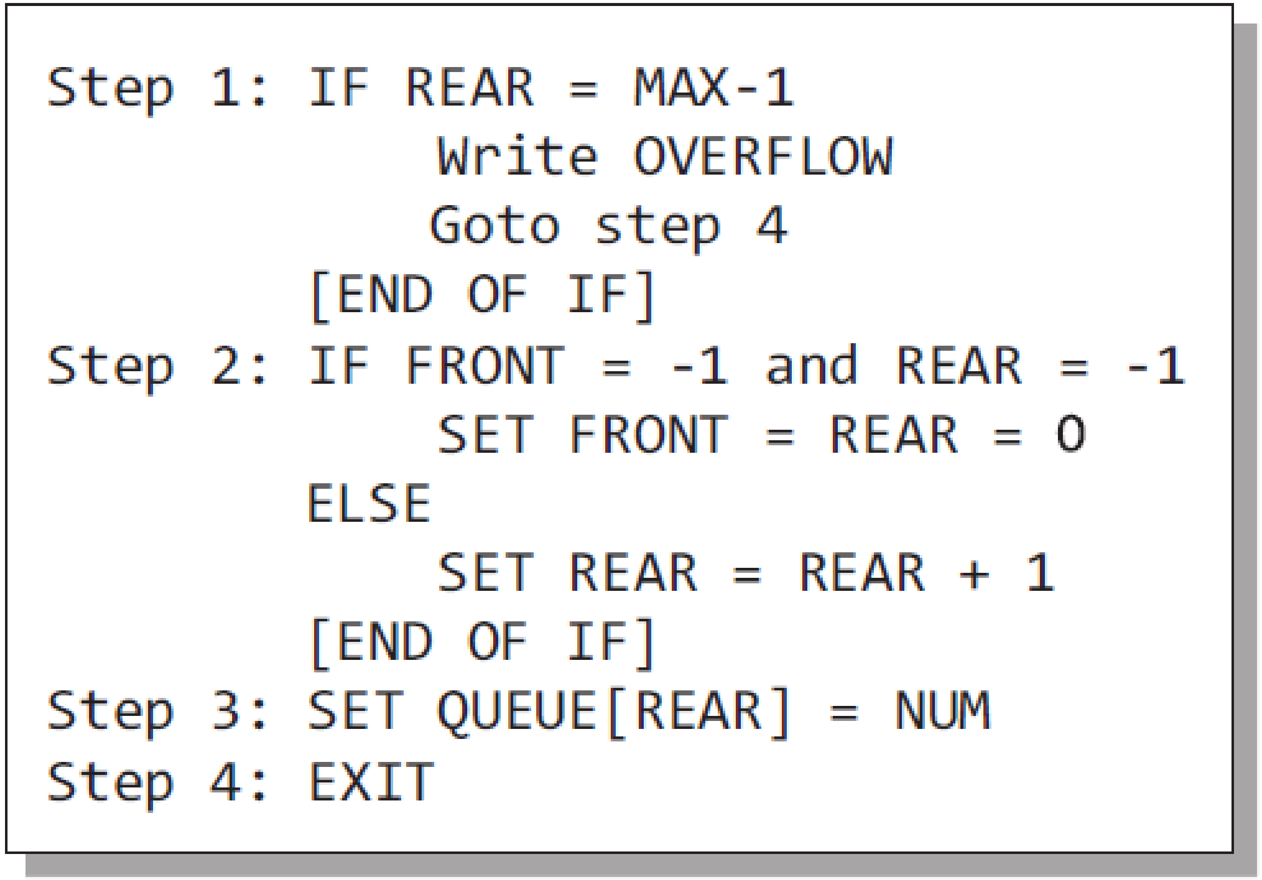
Figure 8.2 Queue after insertion of a new element



	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

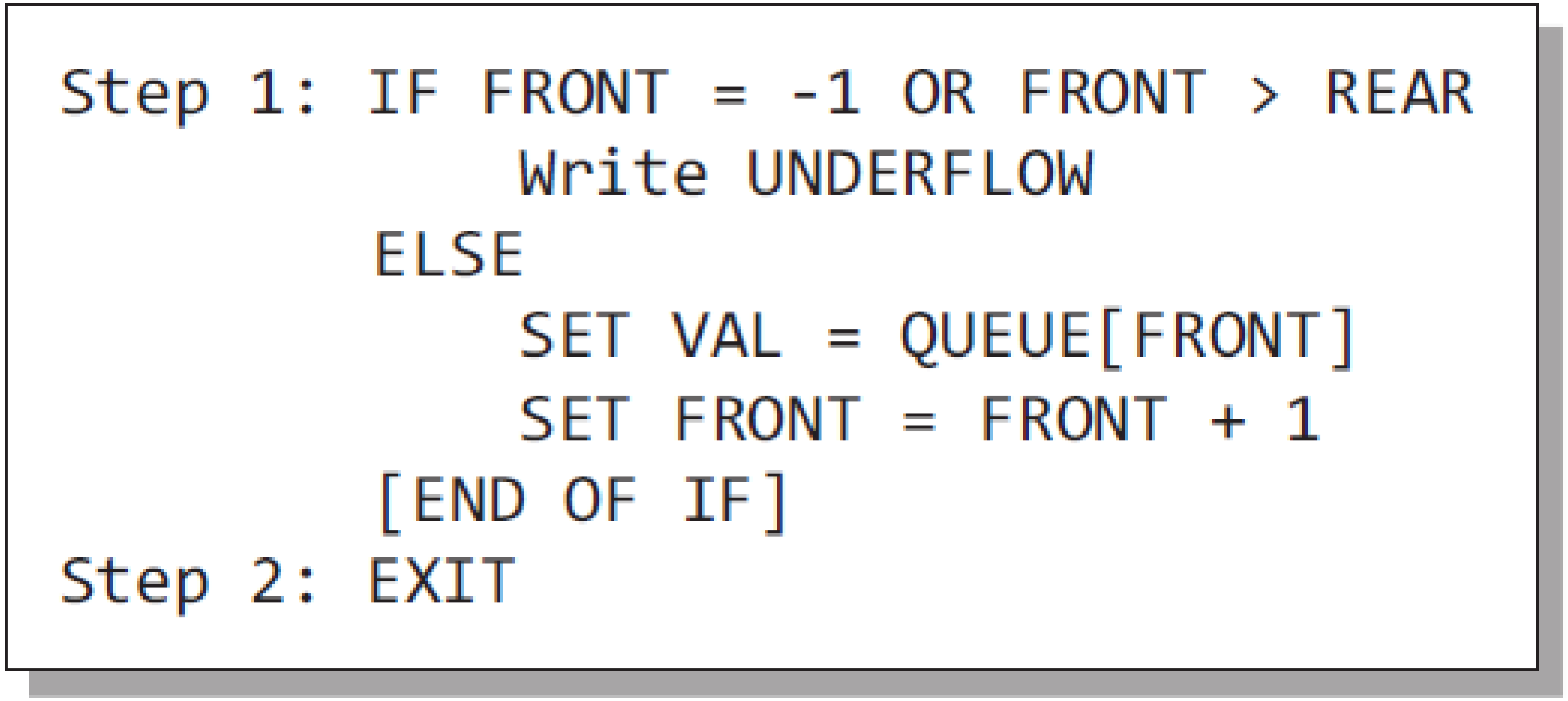
Figure 8.3 Queue after deletion of an element

Algorithm to insert an element in a queue



```
Step 1: IF REAR = MAX-1
              Write OVERFLOW
              Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
              SET FRONT = REAR = 0
        ELSE
              SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from a queue



```
Step 1: IF FRONT = -1 OR FRONT > REAR
              Write UNDERFLOW
          ELSE
              SET VAL = QUEUE[FRONT]
              SET FRONT = FRONT + 1
          [END OF IF]
Step 2: EXIT
```

LINKED REPRESENTATION OF QUEUES

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.
- The `START` pointer of the linked list is used as `FRONT`.
- We will also use another pointer called `REAR`, which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If **`FRONT = REAR = NULL`**, then it indicates that the queue is empty.

Operations on Linked Queues

- The `insert` operation adds an element to the end of the queue, and the `delete` operation removes an element from the front or the start of the queue.

Insert *Operation*

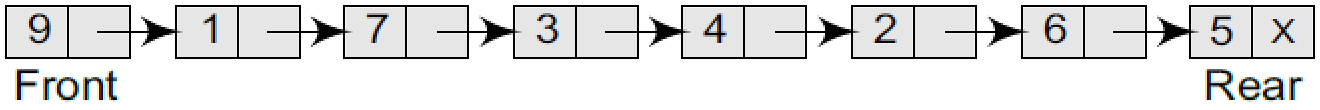


Figure 8.6 Linked queue

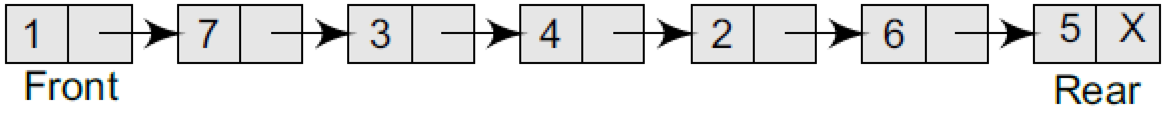


Figure 8.7 Linked queue

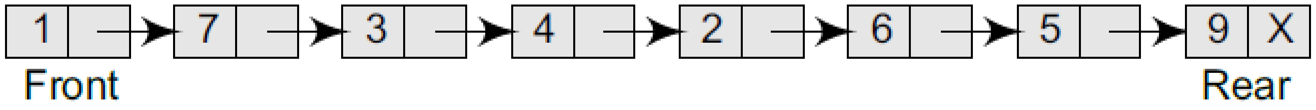
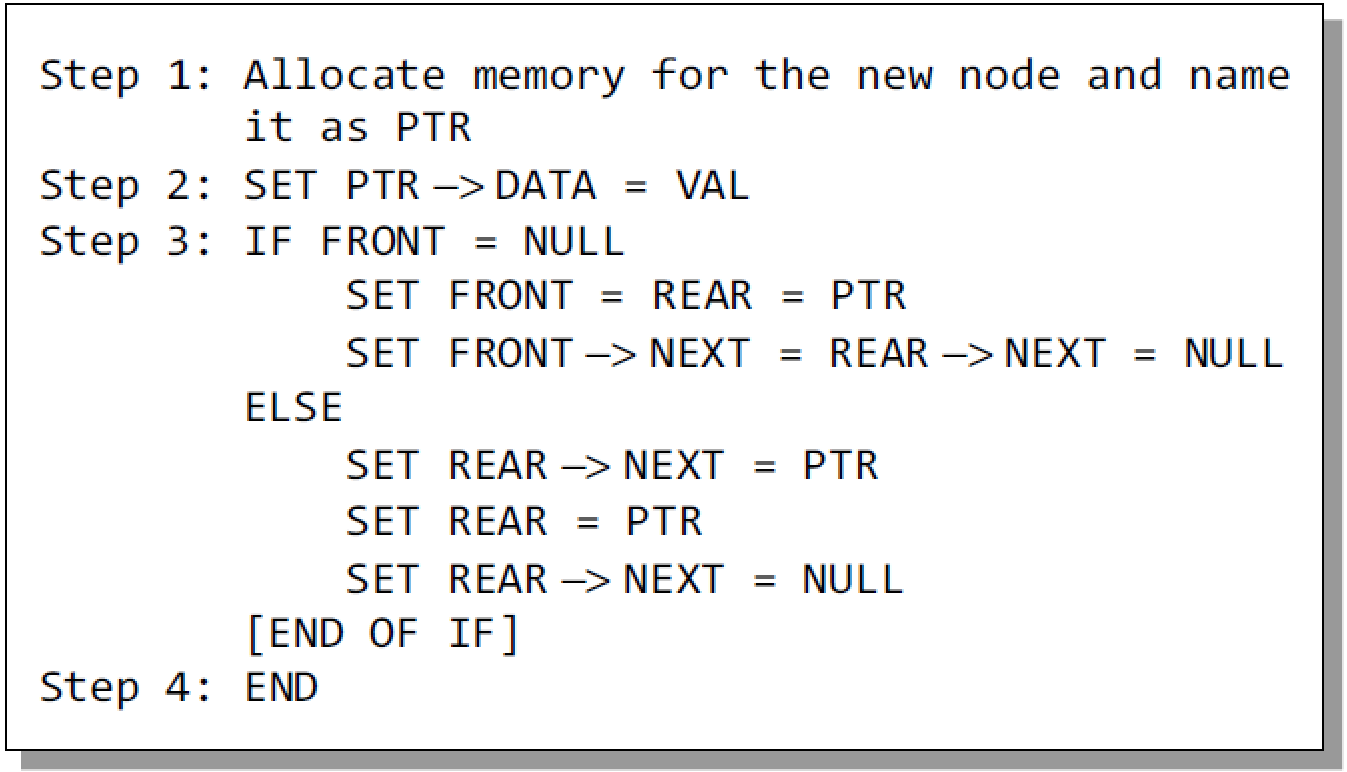


Figure 8.8 Linked queue after inserting a new node

Algorithm to insert an element in a linked queue



```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Structure and pointers

```
struct node
{
   int data;
   struct node *next;
};
struct node *front;
struct node *rear;
```

```
void insert()
{
   struct node *ptr;
   int item;

   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nQUEUE OVERFLOW\n");
      return;
   }
```

```
else
 {
   printf("\nEnter a value to be inserted\n");
   scanf("%d",&item);
   ptr -> data = item;
   if(front == NULL)
   {
     front = ptr;
     rear = ptr;
     front -> next = NULL;
     rear -> next = NULL;
   }
```

```
else
    {
      rear -> next = ptr;
      rear = ptr;
      rear->next = NULL;
    }
  }
}
```

Delete *Operation*

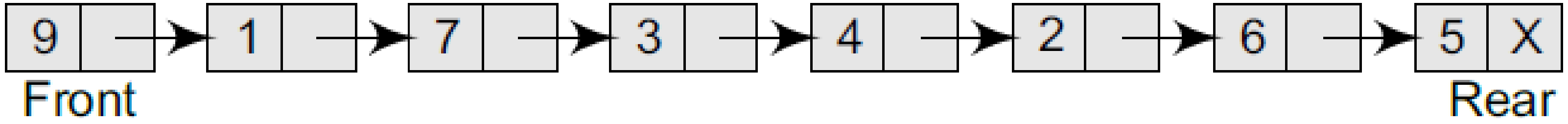


Figure 8.10 Linked queue

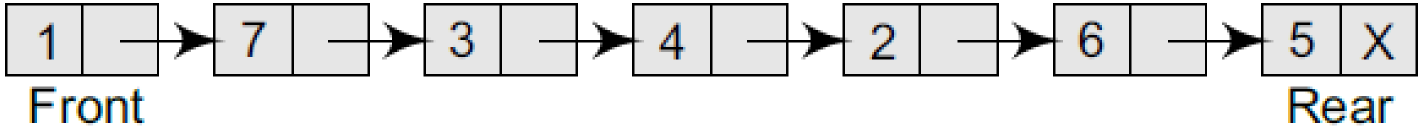
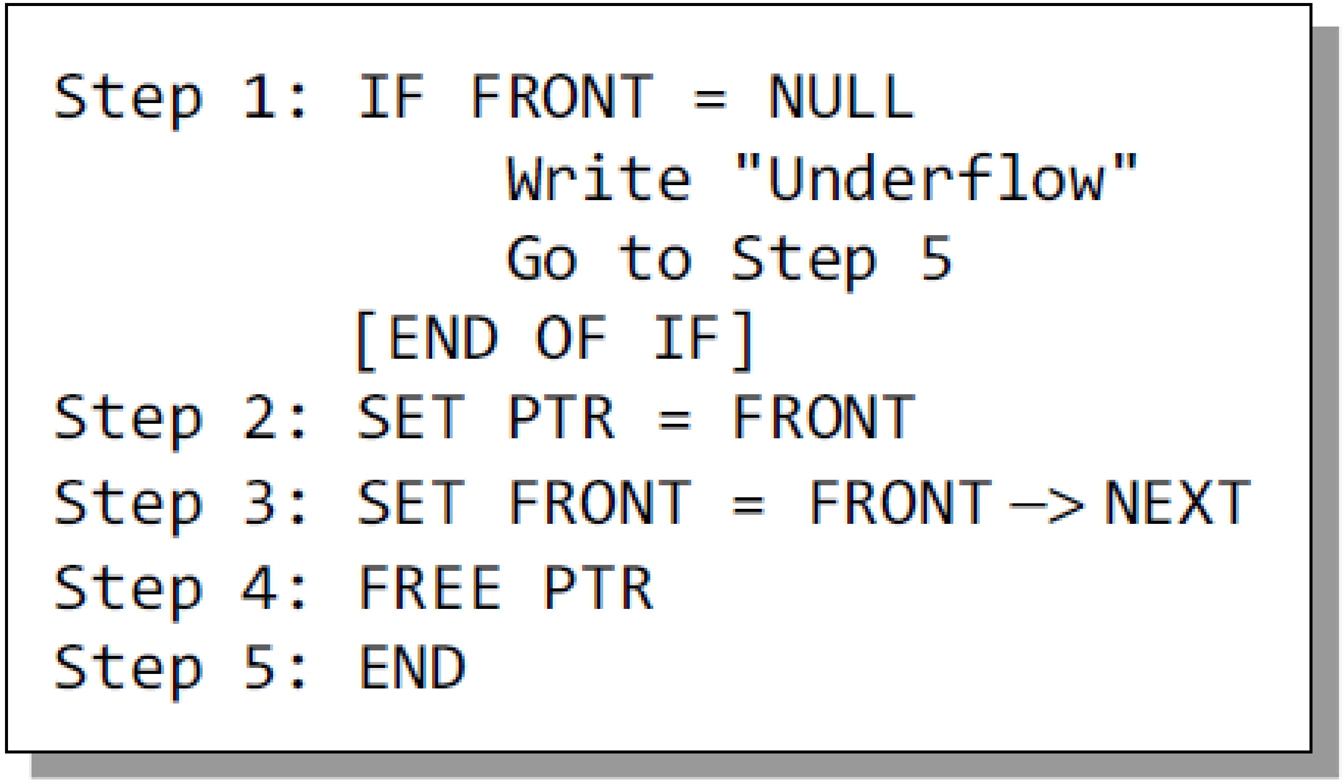


Figure 8.11 Linked queue after deletion of an element

Algorithm to delete an element from a linked queue



```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

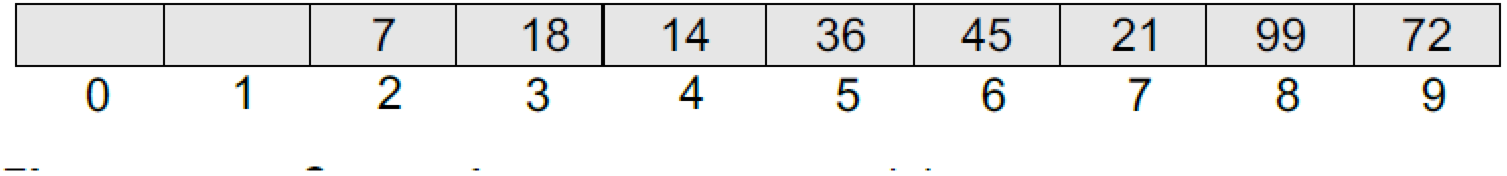
```
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nQUEUE UNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

```
void display()
{
  struct node *ptr;
  ptr = front;
  if(front == NULL)
  {
    printf("\nEmpty queue\n");
  }
else
  {  printf("\nprinting values .....\n");
    while(ptr != NULL)
    {
      printf("\n%d\n",ptr -> data);
      ptr = ptr -> next;
    }
  }
}
```

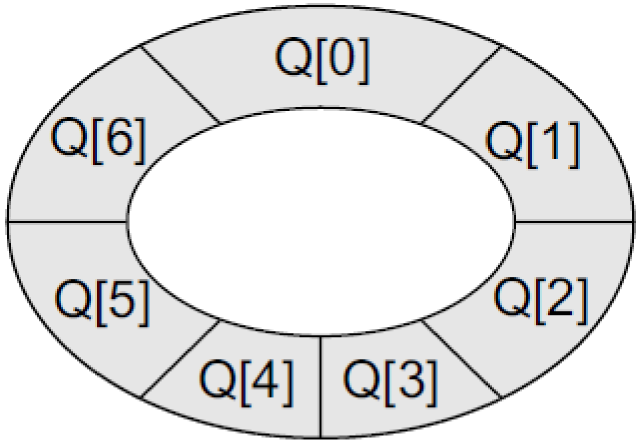
TYPES OF QUEUES

- Circular Queue
- Deque
- Priority Queue
- Multiple Queue

Circular Queues



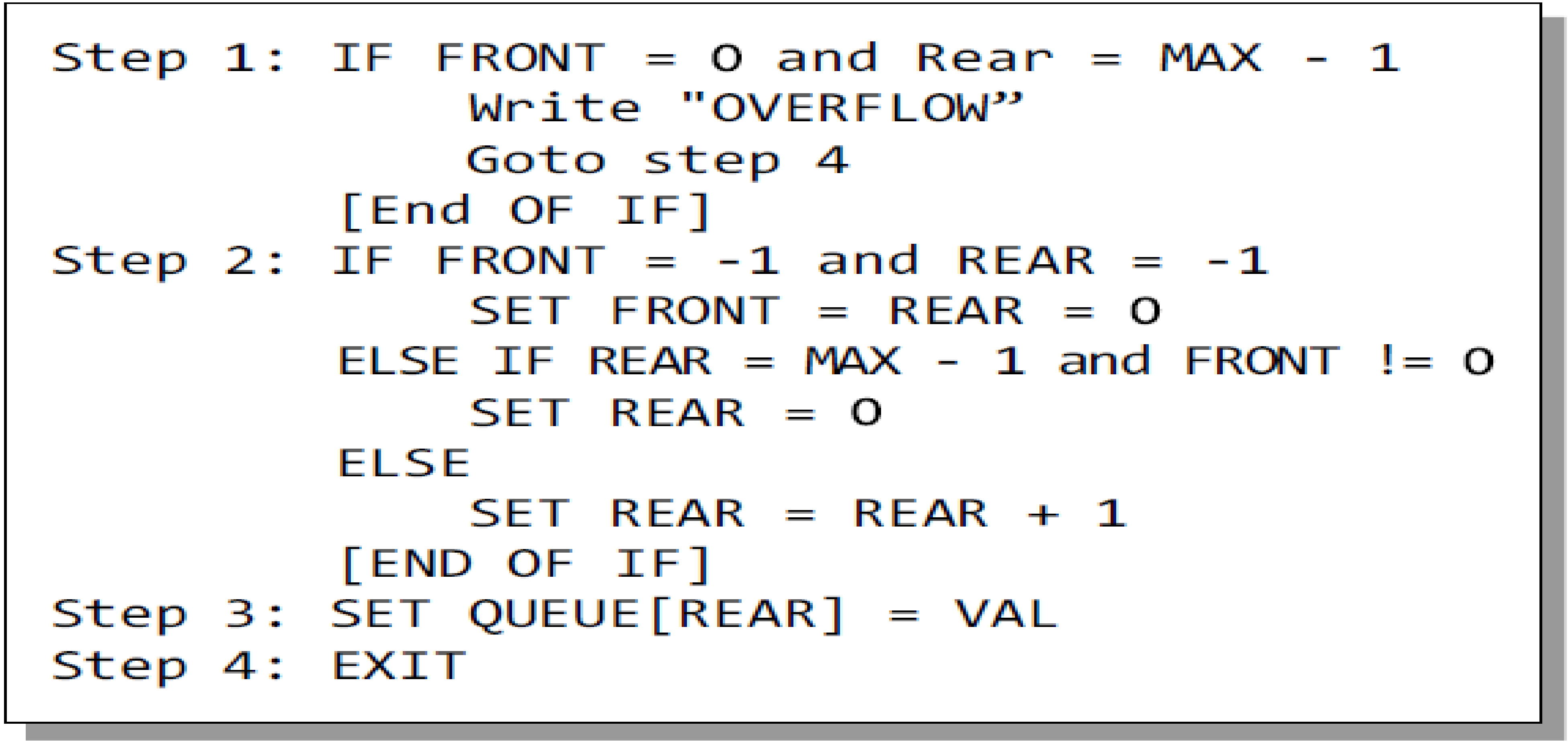
		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9



Inserting an element in a circular queue

- If `front = 0` and `rear = MAX – 1`, then the circular queue is full.
- If `rear != MAX – 1`, then `rear` will be incremented and the value will be inserted
- If `front != 0` and `rear = MAX – 1`, then it means that the queue is not full. So, set `rear = 0` and insert the new element there

Algorithm to insert an element in a circular queue



```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW”
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Deletion from a circular queue

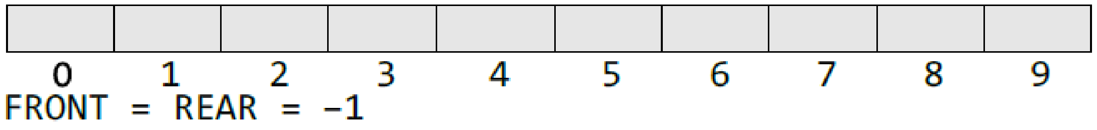
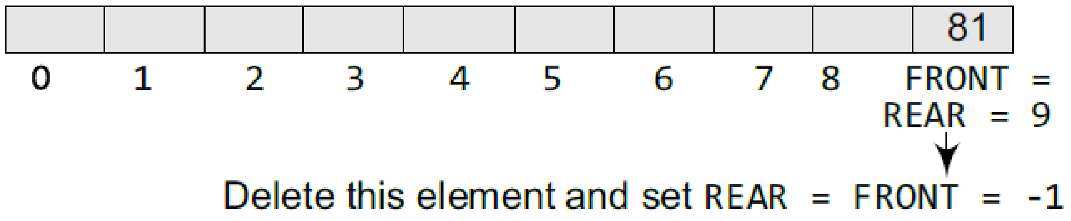
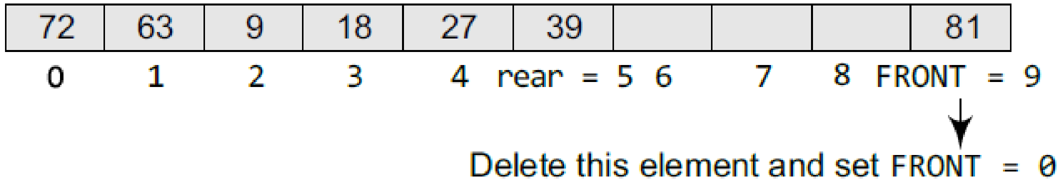


Figure 8.20 Empty queue



Delete this element and set REAR = FRONT = -1

Figure 8.21 Queue with a single element

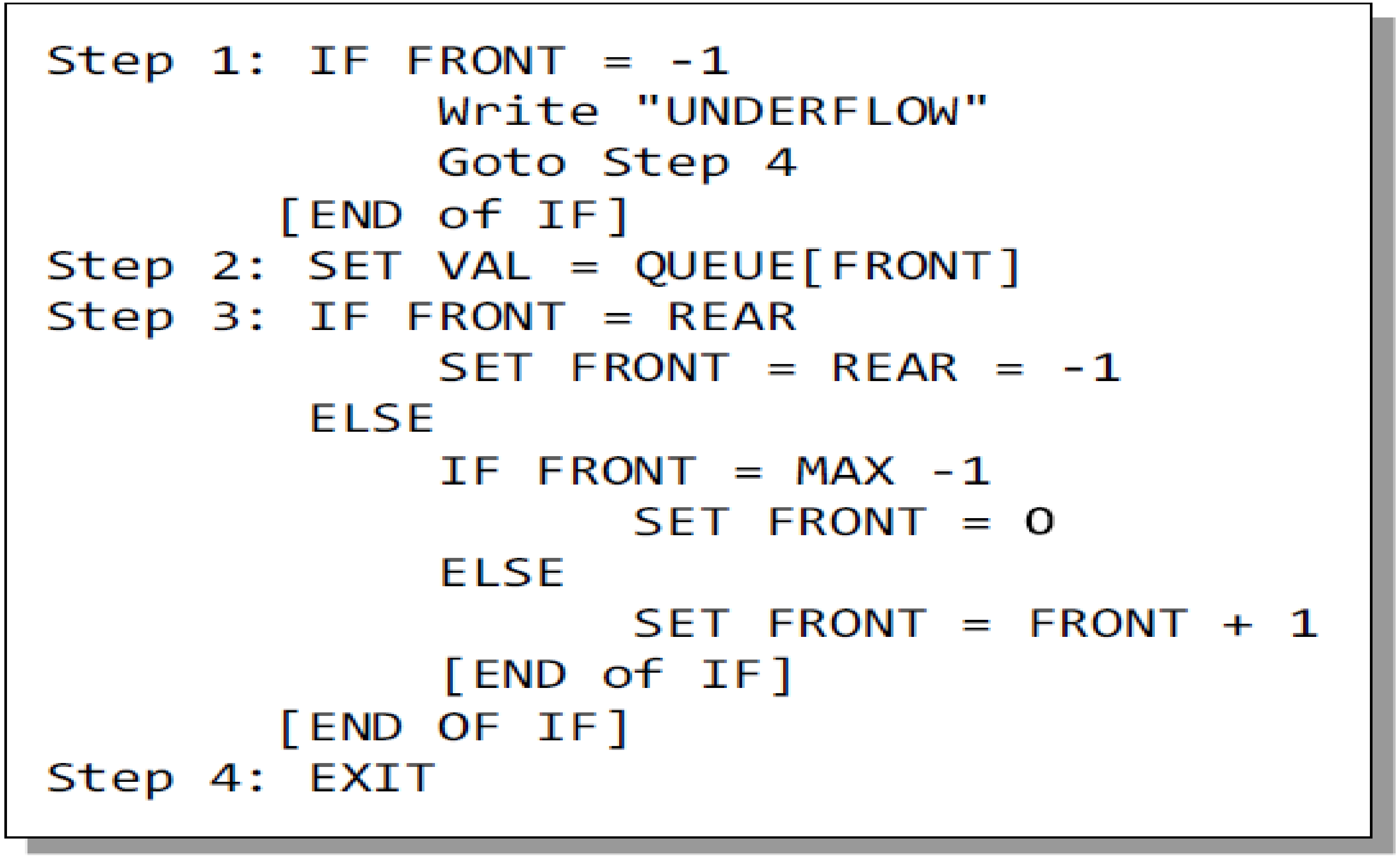


Delete this element and set FRONT = 0

Deletion of an element from circular queue

- If `front = -1`, then there are no elements in the queue. So, an `underflow` condition will be reported.
- If the queue is not empty and `front = rear`, then after deleting the element at the front the queue becomes empty and so `front` and `rear` are set to `-1`
- If the queue is not empty and `front = MAX-1`, then after deleting the element at the front, `front` is set to 0

Algorithm to delete an element from a circular queue



```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
       [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
       [END OF IF]
Step 4: EXIT
```

Deques

- A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end.
- It is also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.
- No element can be added and deleted from the middle.
- In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, `LEFT` and `RIGHT`, which point to either end of the deque.
- The elements in a deque extend from the `LEFT` end to the `RIGHT` end and since it is circular, `Dequeue[N–1]` is followed by `Dequeue[0]`.

Two variants of a double-ended queue.

- *Input restricted deque* In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- *Output restricted deque* In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

			29	37	45	54	63		
0	1	2	LEFT = 3	4	5	6	RIGHT = 7	8	9

42	56						63	27	18
0	RIGHT = 1	2	3	4	5	6	LEFT = 7	8	9

Priority Queues

- A priority queue is a data structure in which each element is assigned a priority.
- The priority of the element will be used to determine the order in which the elements will be processed.
- The general rules of processing the elements of a priority queue are
- **An element with higher priority is processed before an element with a lower priority.**
- **Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.**
- A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first.
- The priority of the element can be set based on various factors.

- Priority queues are widely used in operating systems to execute the highest priority process first.
- The priority of the process may be set based on the CPU time it requires to get executed completely.
- For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.
- However, CPU time is not the only factor that determines the priority, rather it is just one among several factors.

- Another factor is the importance of one process over another.
- In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Implementation of a Priority Queue

- ***Linked Representation of a Priority Queue***
- ***Array Representation of a Priority Queue***

Linked Representation of a Priority Queue

- When a priority queue is implemented using a linked list, then every node of the list will have three parts:
- (a) the information or data part, (b) the priority number of the element, and (c) the address of the next element.
- If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.

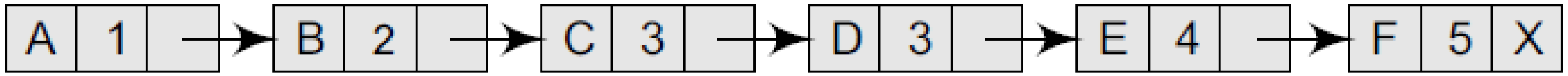
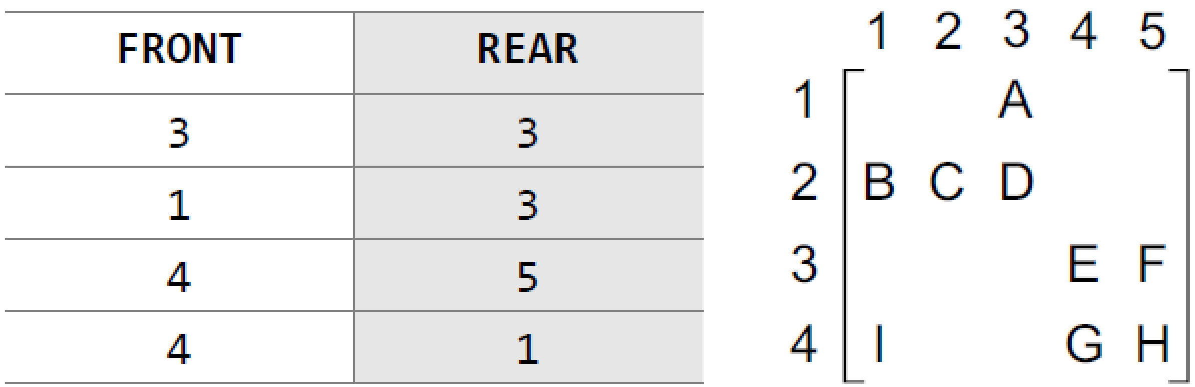


Figure 8.25 Priority queue

Array Representation of a Priority Queue

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues.
- Every individual queue will have its own `FRONT` and `REAR` pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.



FRONT	REAR
3	3
1	3
4	5
4	1

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & & & & E & F \\ 4 & I & & & G & H \end{array}$$

Figure 8.29 Priority queue matrix

Thank You

Algorithms

PREPARED BY

DEEPTHI M PISHARODY

ASST PROFESSOR

PRAJYOTI NIKETAN COLLEGE, PUDUKAD

ALGORITHMS

- The typical definition of algorithm is ‘a formally defined procedure for performing some calculation’.
- If a procedure is formally defined, then it can be implemented using a formal language, and such a language is known as a *programming language*.
- In general terms, an algorithm provides a blueprint to write a program to solve a particular problem.
- It is considered to be an effective procedure for solving a problem in finite number of steps.
- That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

- Algorithms are mainly used to achieve *software reuse.*
- Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java.
- An algorithm is basically a set of instructions that solve a problem.
- It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM

- Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.
- A complex algorithm is often divided into smaller units called modules.
- This process of dividing an algorithm into modules is called modularization.
- The key advantages of modularization are as follows:
  - It makes the complex algorithm simpler to design and implement.
  - Each module can be designed independently.
- While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

- There are two main approaches to design an algorithm—top-down approach and bottom-up approach,

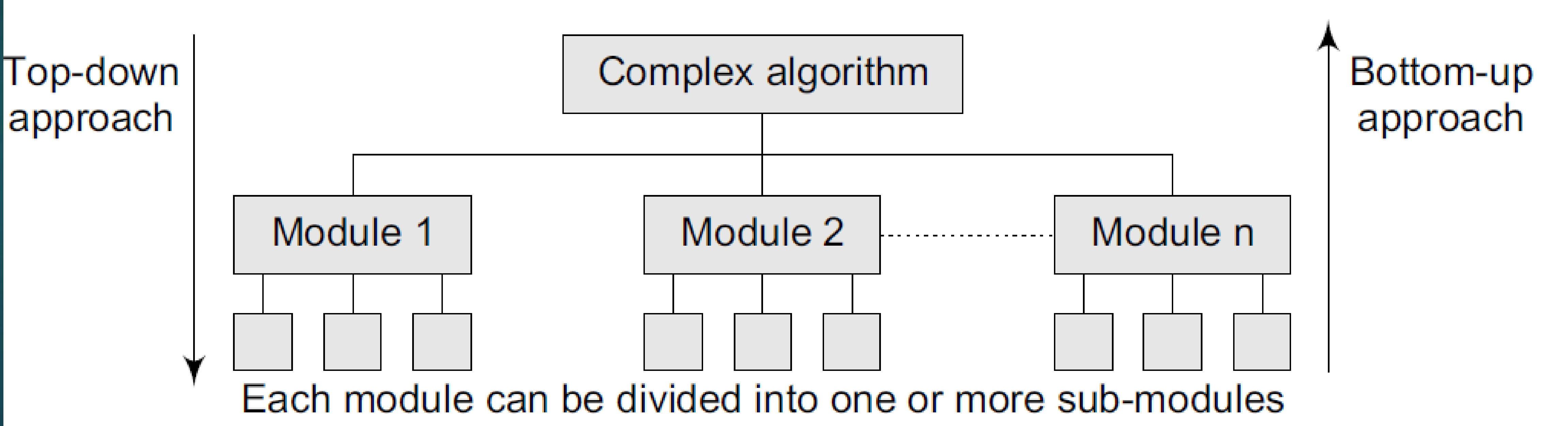


Figure 2.9 Different approaches of designing an algorithm

Top-down approach

- A top-down design approach starts by dividing the complex algorithm into one or more modules.
- These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved.
- Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls.
- Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Bottom-up approach

- A bottom-up approach is just the reverse of top-down approach.
- In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules.
- The higher level modules are implemented by using the operations performed by lower level modules.
- Thus, in this approach sub-modules are grouped together to form a higher level module.
- All the higher level modules are clubbed together to form even higher level modules.
- This process is repeated until the design of the complete algorithm is obtained.

Top-down vs bottom-up approach

- While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.
- Top-down approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.
- Although the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module's boundaries as seen from the clients. But all this is difficult to be done in a strict bottom-up strategy.
- Some top-down activities need to be performed for this. All in all, design of complex algorithms must not be constrained to proceed according to a fixed pattern but should be a blend of top-down and bottom-up approaches.

CONTROL STRUCTURES USED IN ALGORITHMS

- ***Sequence***
- By sequence, we mean that each step of an algorithm is executed in a specified order. Let us write an algorithm to add two numbers.
- ***Decision***
- Decision statements are used when the execution of a process depends on the outcome of some condition. For example, if `x = y`, then print `EQUAL`. So the general form of `IF` construct can be given as:
- `IF` *`condition`* `Then` *`process`*
- A condition in this context is any statement that may evaluate to either a true value or a false value.

- ***Repetition***
- Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as `while`, `do–while`, and `for` loops. These loops execute one or more steps until some condition is true.

Thank you

TIME AND SPACE COMPLEXITY

PREPARED BY

DEEPTHI M PISHARODY

ASST PROFESSOR

PRAJYOTI NIKETAN COLLEGE, PUDUKAD

- Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

- The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. The number of machine instructions which a program executes is called its time complexity.
- Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

- The space needed by a program depends on the following two parts:
- *Fixed part*: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- *Variable part*: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

WORST-CASE, AVERAGE-CASE, BEST-CASE, AND AMORTIZED TIME COMPLEXITY

- ***Worst-case running time*** This denotes the behaviour of an algorithm with respect to the worstpossible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.
- ***Average-case running time*** The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

- ***Best-case running time*** The term ‘best-case performance’ is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.
- ***Amortized running time*** Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

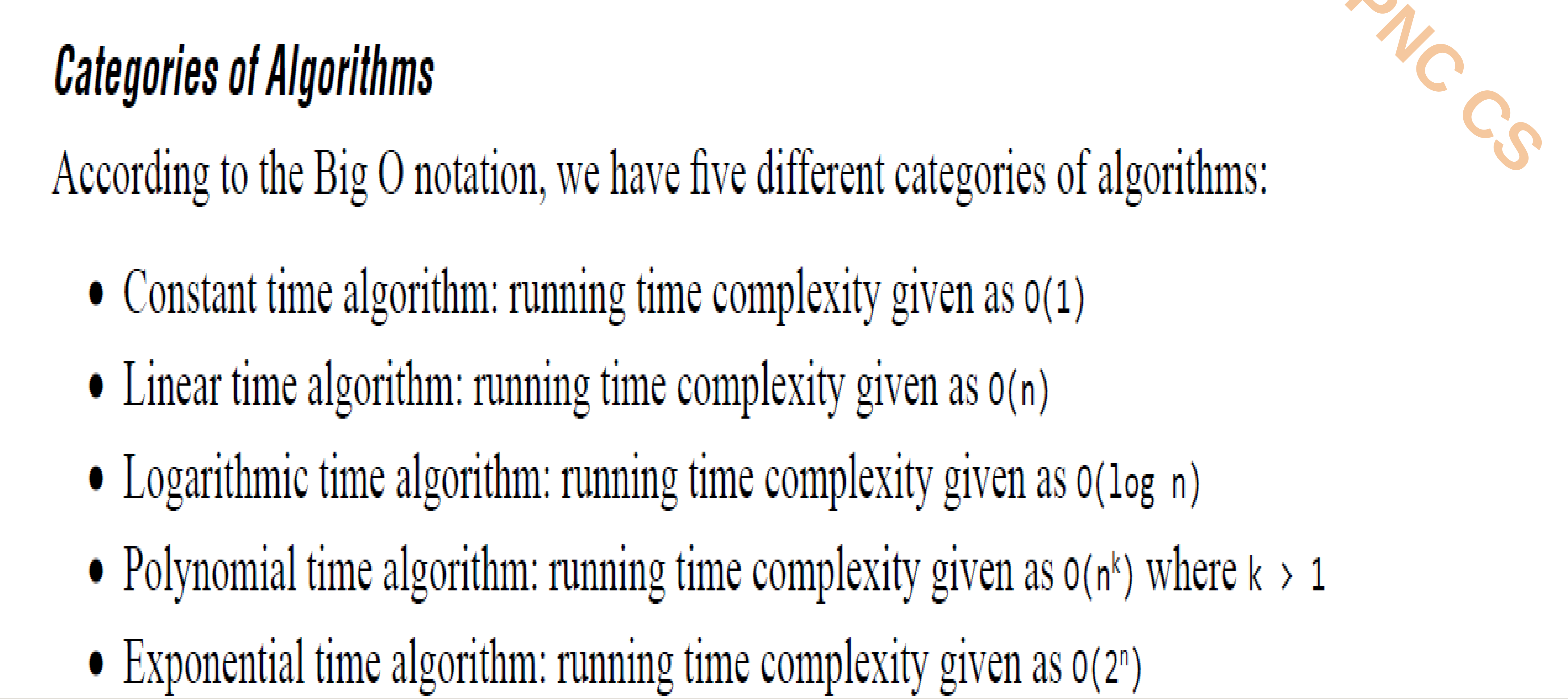
TIME–SPACE TRADE-OFF

- The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution.
- But practically, designing such an ideal algorithm is not a trivial task.
- There can be more than one algorithm to solve a particular problem.
- One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a time–space trade-off among algorithms.

- So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time. On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

EXPRESSING TIME AND SPACE COMPLEXITY

- The time and space complexity can be expressed using a function `f(n)` where `n` is the input size for a given instance of the problem being solved.
- Expressing the complexity is required when
- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.
- The most widely used notation to express this function `f(n)` is the Big O notation.
  - It provides the upper bound for the complexity.

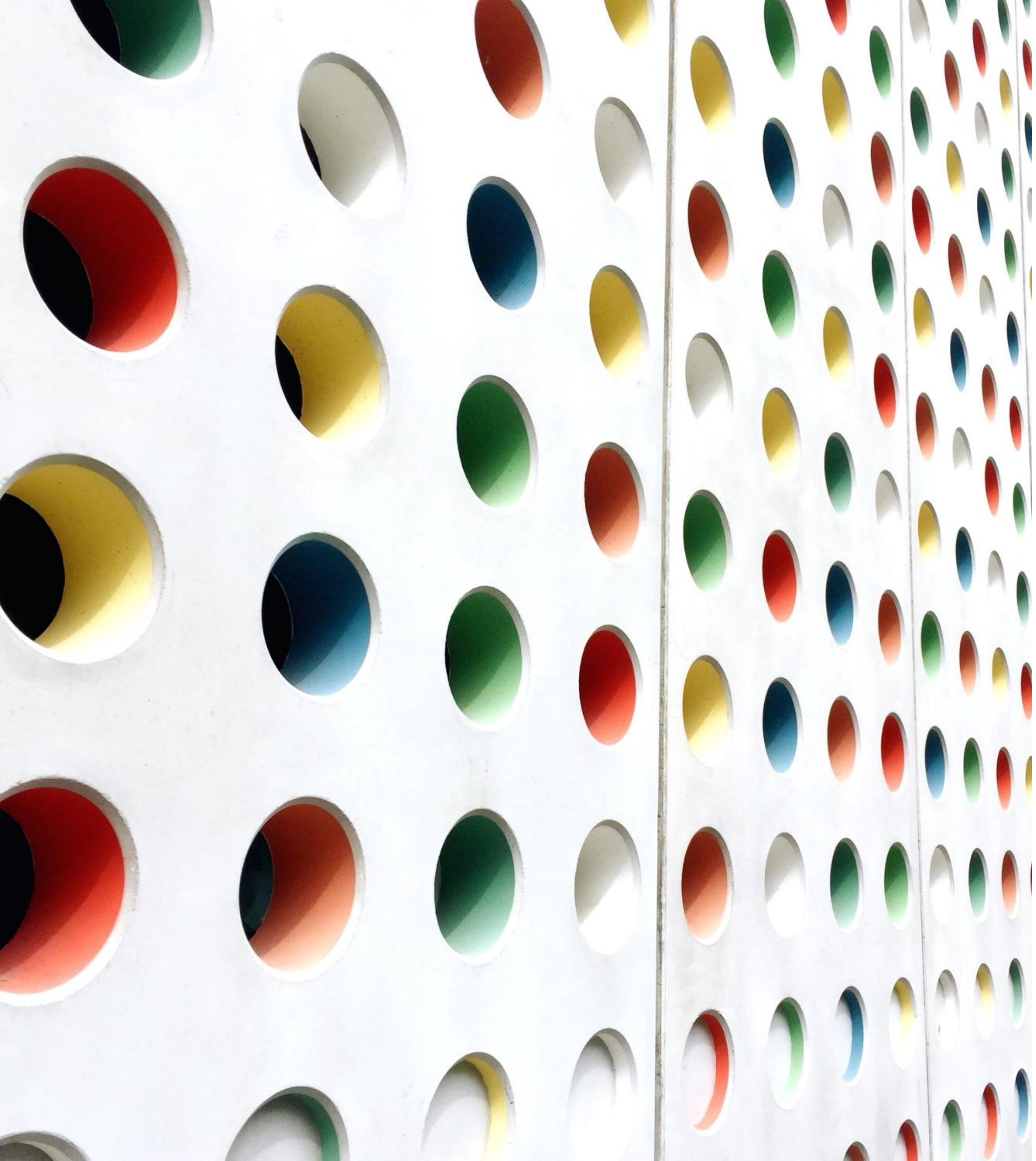


Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as $O(1)$
- Linear time algorithm: running time complexity given as $O(n)$
- Logarithmic time algorithm: running time complexity given as $O(\log n)$
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Thank you



Sorting

PREPARED BY

DEEPTHI M PISHARODY

ASST PROFESSOR

PRAJYOTI NIKETAN COLLEGE, PUDUKAD

Sorting

✓Sorting means arranging the elements of an array so that they are placed in some relevant order

which may be either ascending or descending.

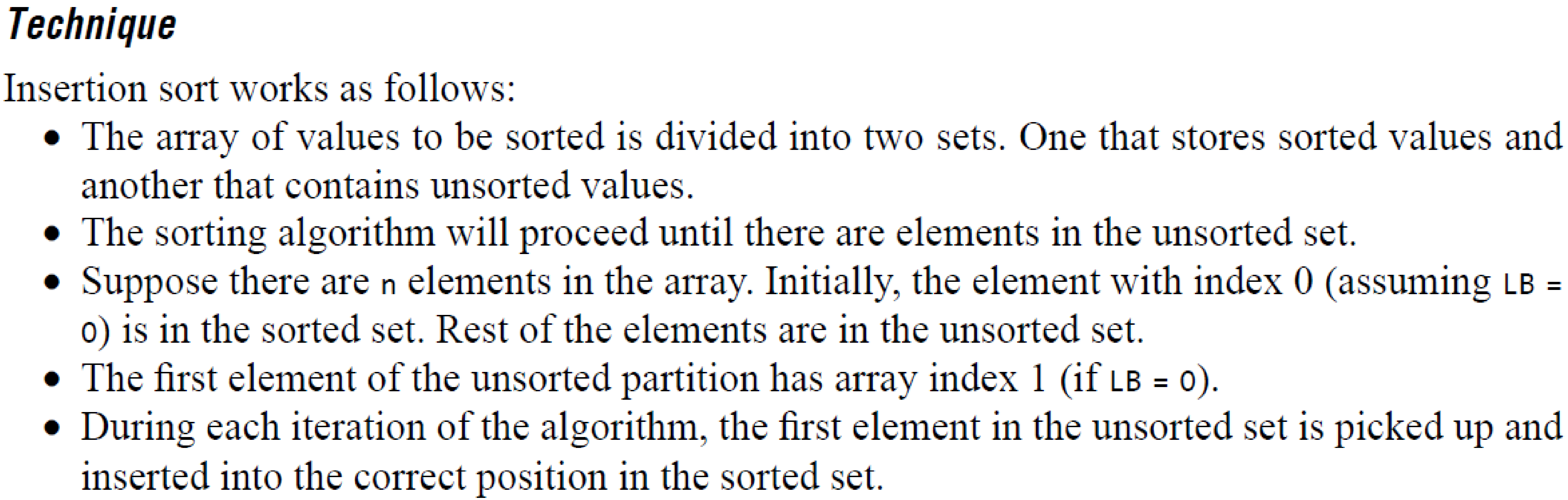
➢***Internal sorting*** which deals with sorting the data stored in the computer’s memory

➢***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

INSERTION SORT

- ✓Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- ✓use it for ordering a deck of cards while playing bridge.
- ✓The main idea behind insertion sort is that it inserts each item into its proper place in the final list.
- ✓To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.
- ✓Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Technique



Technique

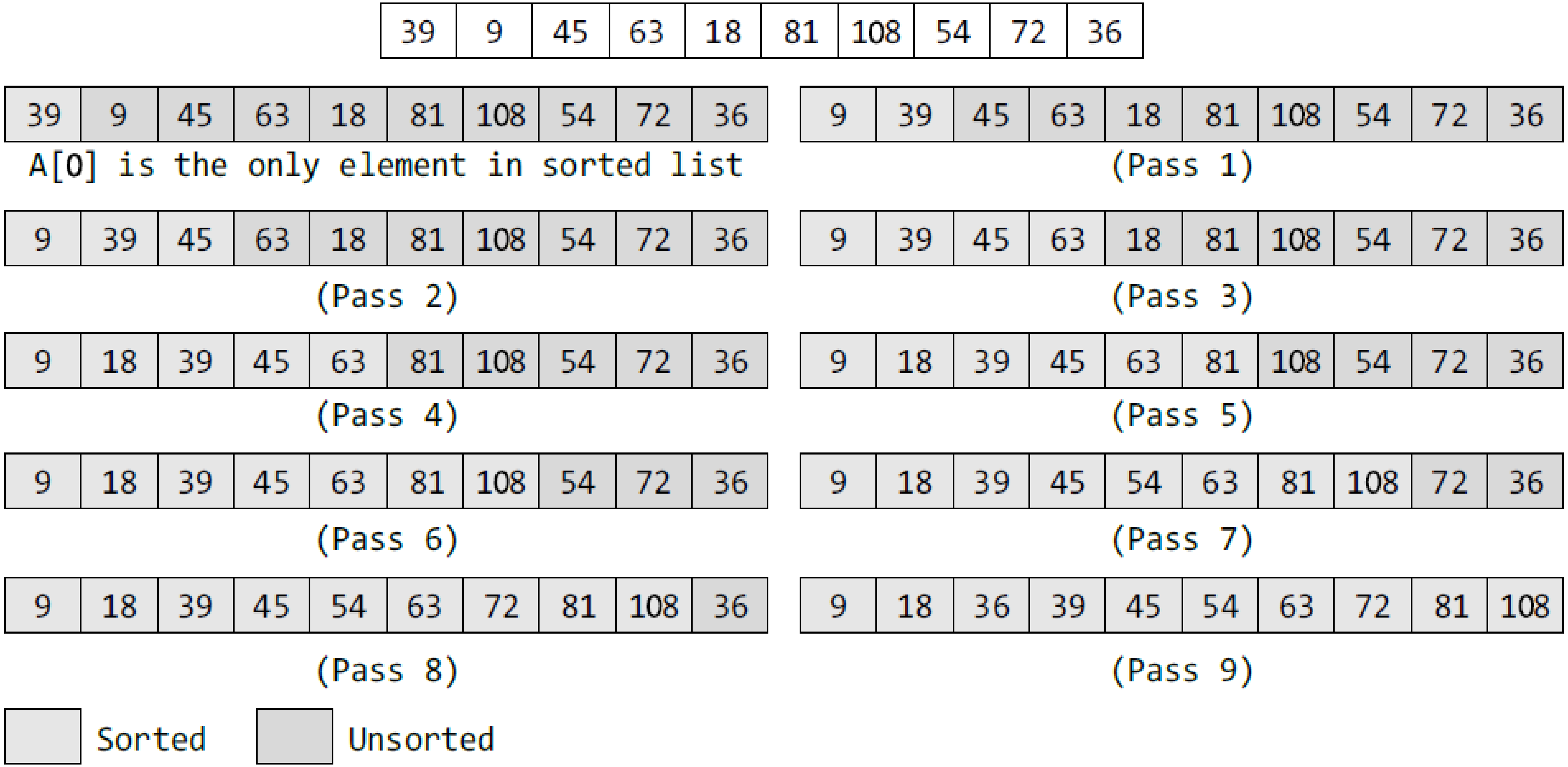
Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are `n` elements in the array. Initially, the element with index 0 (assuming `LB = 0`) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if `LB = 0`).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

- Initially, `A[0]` is the only element in the sorted set. In Pass 1, `A[1]` will be placed either before or after `A[0]`, so that the array `A` is sorted.
- In Pass 2, `A[2]` will be placed either before `A[0]`, in between `A[0]` and `A[1]`, or after `A[1]`.
- In Pass 3, `A[3]` will be placed in its proper place.
- In Pass `N–1`, `A[N–1]` will be placed in its proper place to keep the array sorted.

Example 14.3 Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution



39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36

A[0] is the only element in sorted list

9	39	45	63	18	81	108	54	72	36

(Pass 1)

9	39	45	63	18	81	108	54	72	36

(Pass 2)

9	39	45	63	18	81	108	54	72	36

(Pass 3)

9	18	39	45	63	81	108	54	72	36

(Pass 4)

9	18	39	45	63	81	108	54	72	36

(Pass 5)

9	18	39	45	63	81	108	54	72	36

(Pass 6)

9	18	39	45	54	63	81	108	72	36

(Pass 7)

9	18	39	45	54	63	72	81	108	36

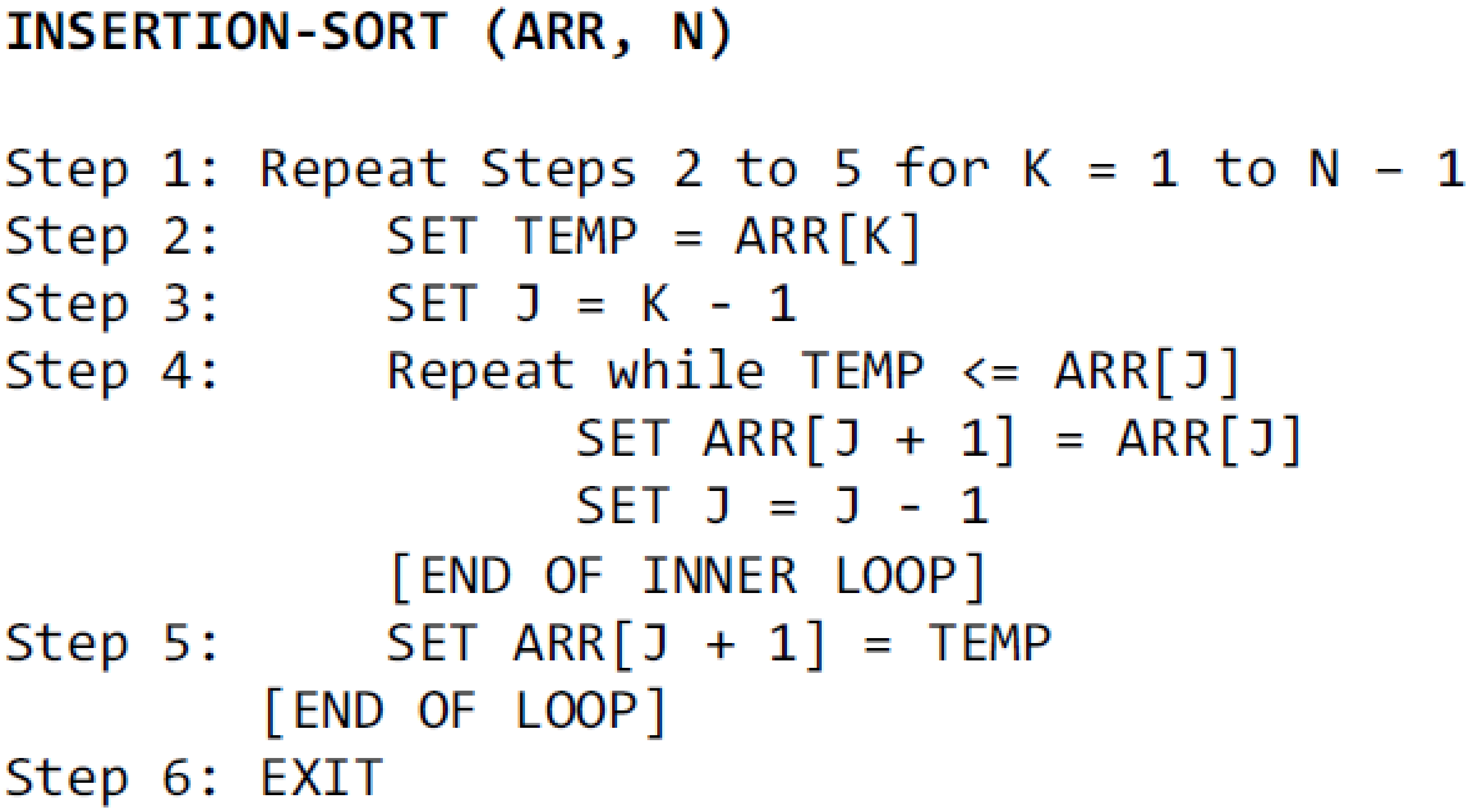
(Pass 8)

9	18	36	39	45	54	63	72	81	108

(Pass 9)

□ Sorted ▨ Unsorted

General Algorithm



```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2:         SET TEMP = ARR[K]
Step 3:         SET J = K - 1
Step 4:         Repeat while TEMP <= ARR[J]
                        SET ARR[J + 1] = ARR[J]
                        SET J = J - 1
                [END OF INNER LOOP]
Step 5:         SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Complexity of Insertion Sort

Best case –O(n)

Worst case-O(n^2)

Advantages of Insertion Sort

✓It is easy to implement and efficient to use on small sets of data.

✓ It can be efficiently implemented on data sets that are already substantially sorted.

✓It performs better than algorithms like selection sort and bubble sort.

✓ Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency.

✓It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.

✓it requires less memory space (only `O(1)` of additional memory space).

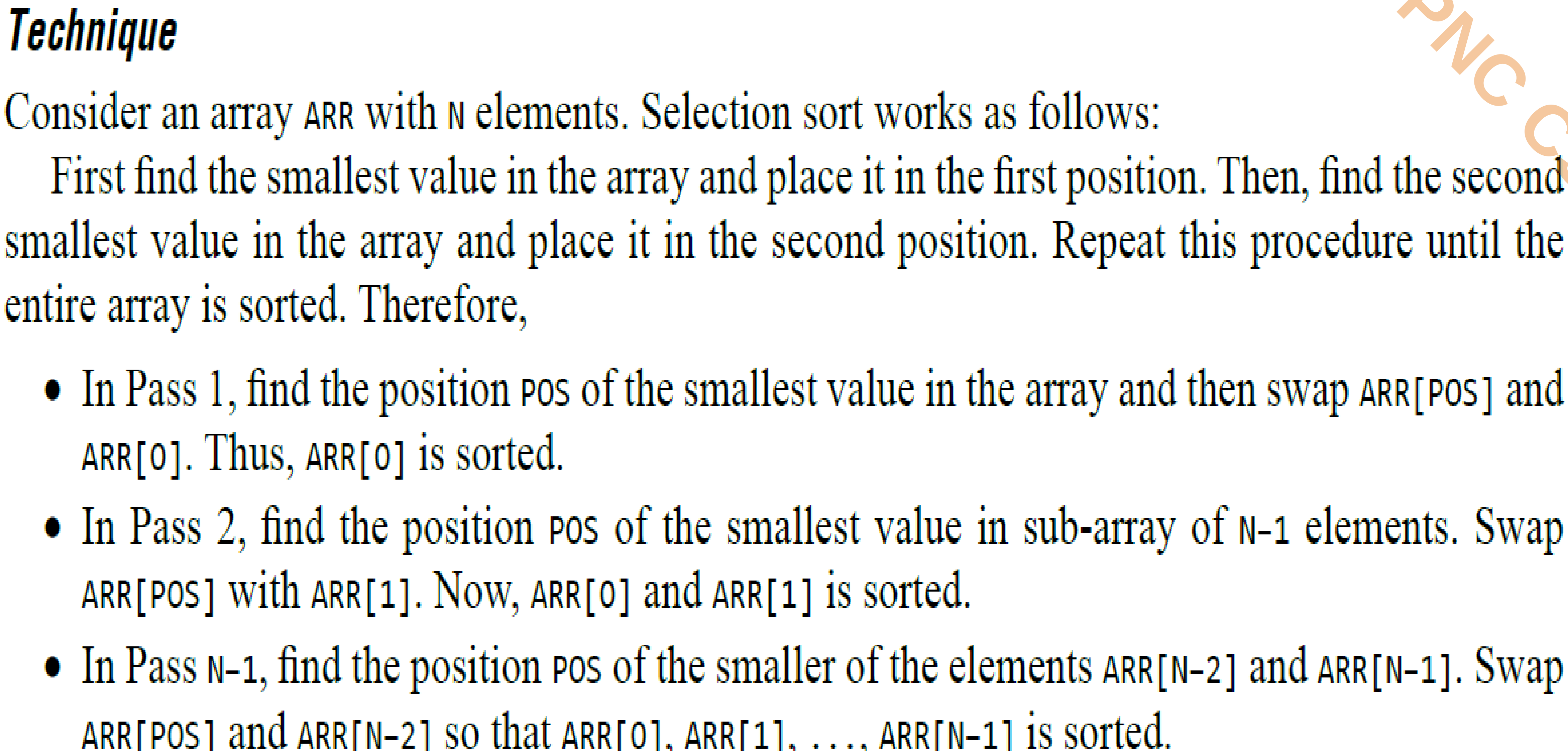
✓ I t is said to be online, as it can sort a list as and when it receives new elements.

```
void insertion_sort(int arr[], int n)
{    int i, j, temp;
 for(i=1;i<n;i++)
 {          temp = arr[i];
    j = i-1;
 while((temp < arr[j]) && (j>=0))
  {
arr[j+1] = arr[j];
    j--;
}
    arr[j+1] = temp;
}}
```

SELECTION SORT

Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages

Selection sort is generally used for sorting files with very large objects (records) and small keys.

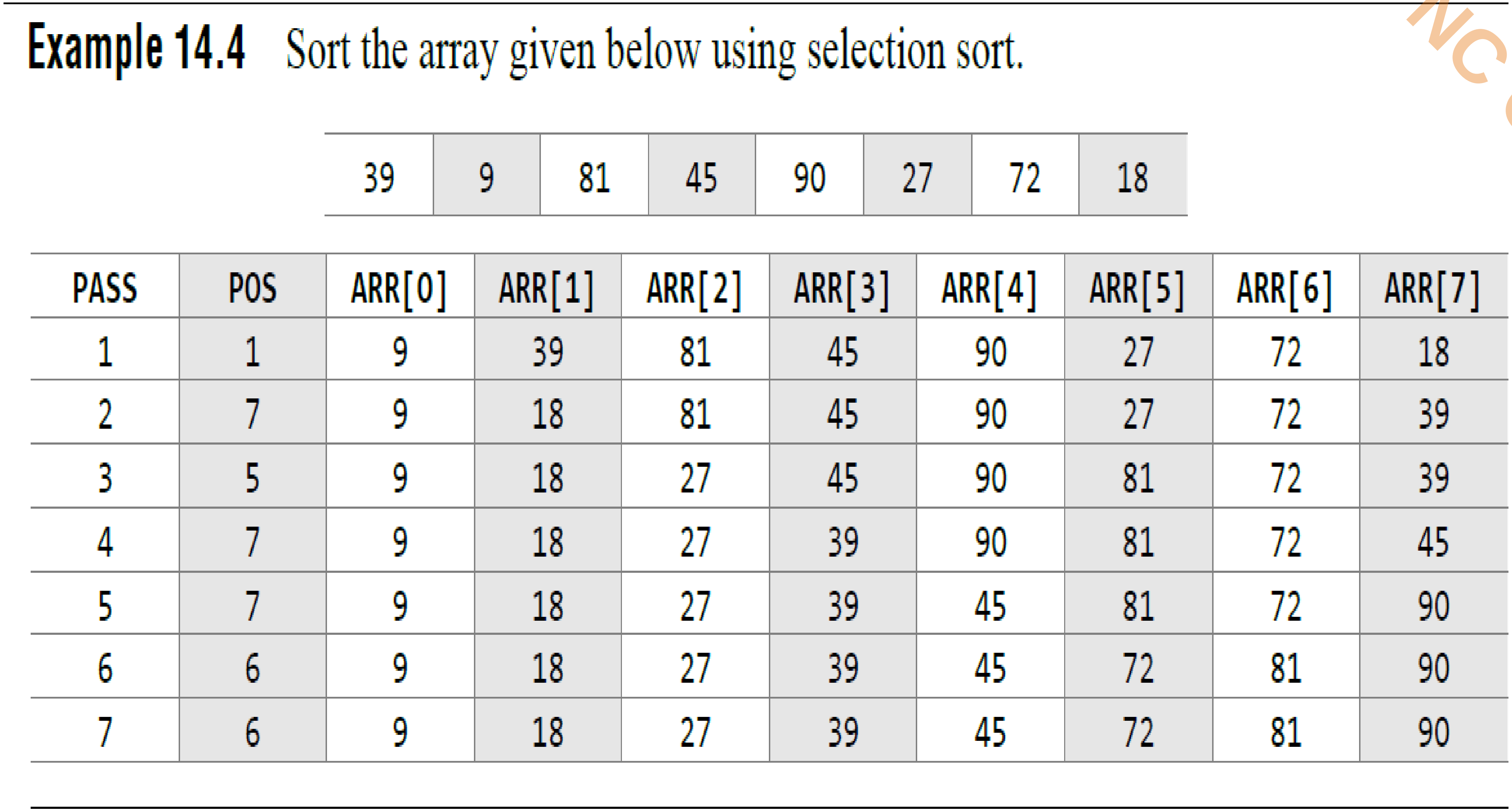


Technique

Consider an array ARR with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1]. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], …, ARR[N-1] is sorted.



Example 14.4 Sort the array given below using selection sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

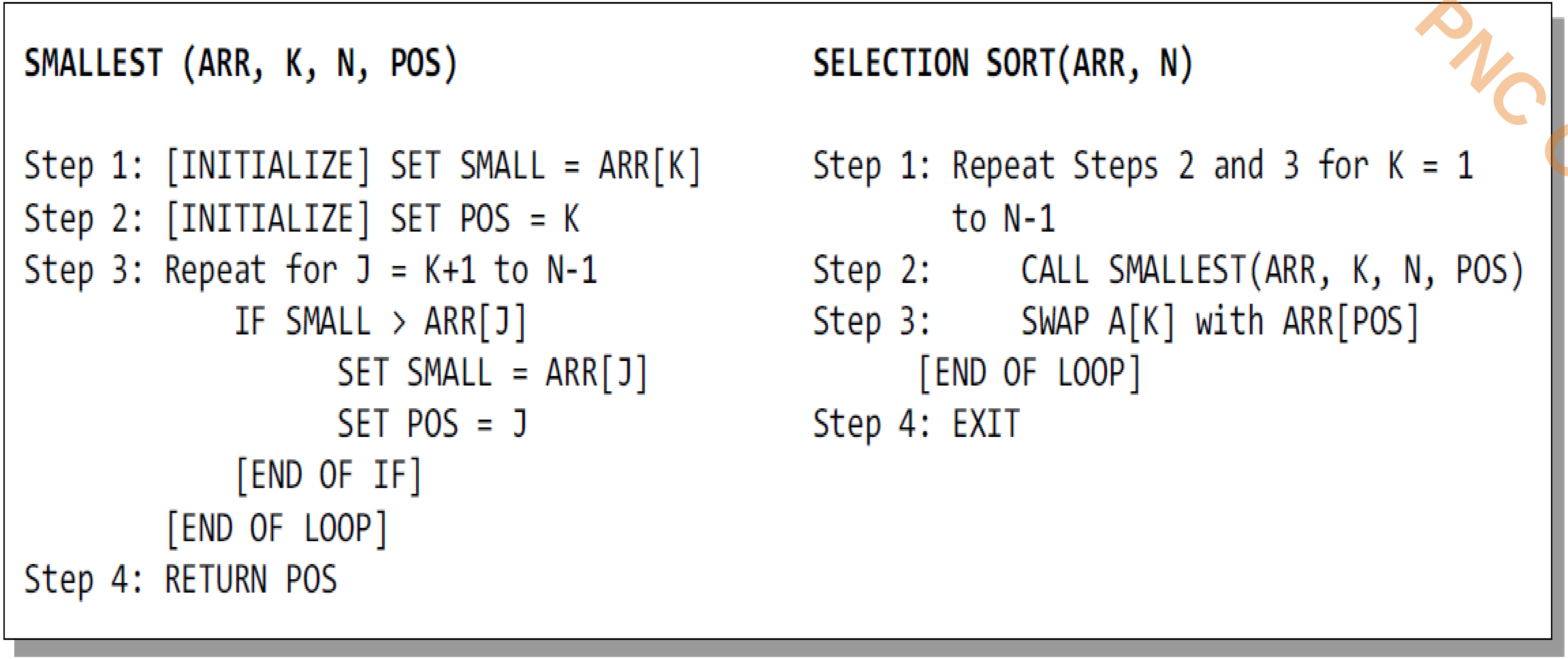
In the algorithm, during the `Kth` pass, we need to find the position `POS` of the smallest elements from `ARR[K]`, `ARR[K+1]`, `...`, `ARR[N]`.

To find the smallest element, we use a variable `SMALL` to hold the smallest value in the sub-array ranging

from `ARR[K]` to `ARR[N]`.

Then, swap `ARR[K]` with `ARR[POS]`.

This procedure is repeated until all the elements in the array are sorted.



```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Figure 14.8 Algorithm for selection sort

Complexity of Selection Sort

$\mathrm{O}(n^2)$

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

QUICK SORT

- Quick sort is a widely used sorting algorithm developed by C. A. R.
- Hoare that makes `O(n log n)` comparisons in the average case to sort an array of `n` elements.
- However, in the worst case, it has a quadratic running time given as $O(n^2)$.
- Basically, the quick sort algorithm is faster than other `O(n log n)` algorithms, because its efficient implementation can minimize the probability of requiring quadratic time.
- Quick sort is also known as partition exchange sort.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.

2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.

3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

- Performs well on large tables
- At each step goal is to place a particular element in its final position within the table.
- In doing so, all elements which precede will have smaller keys, while all that follows have larger values
- This process partition table into two sub tables.
- The above procedure is repeated to each sub table

- Let k be an array with n elements
- Take two variables i,j with initial values i=1 and j=n-1
- Two keys k[0] and k[i] are compared.
- If k[i]< k[0] then I is incremented by 1.
  - The process is repeated

When k[i]>= k[0] we proceed to compare k[j] and k[0].

If k[j]>k[0] j is decremented by 1.

- This is repeated until k[j]<= k[0]

If i<j we exchange k[i] & k[j]

Then keep j fixed and incrementing i continue above method.

When i>=j the desired key k[0] is placed in its final position by exchanging k[0] & k[j]

One pass is over

First sub table lb=0 and =j-1

Second sub table lb=j+1 and ub=n-1

In each sub table perform quick sort

Thank you