DATA STRUCTURES
Mechanical Engineering

III SEMESTER

Prepared by:

Mrs. K Laxminarayanamma, Assistant Professor

MODULE -I

Basic concepts: Introduction to data structures, classification of data structures, operations on data structures; Searching techniques: Linear search and Binary search; Sorting techniques: Bubble sort, selection sort, insertion sort and comparison of sorting algorithms.

What is Data?

Data(Dictionary Definition):

the quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

Example: $C = A + B$

When Data Becomes Information?

Data	imxal si eman yM
Information	My name is laxmi

When data arranged in a systematic way then it gets a structure and become meaningful

This meaningful or processed data is called information

It is not difficult to understand that data needs to be managed in such a way so that it can produce some meaningful information

To provide an appropriate way to structure the data, we need to know about

Data Structures

Introduction to Data Structures

What is Data structure?

- A **data structure** is a **data** organization, management, and storage format that enables efficient access and modification. More precisely, a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data.**

  In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later.

- A data structure should be seen as a logical concept that must address two fundamental concerns.

  I. **First, how the data will be stored, and**

  II. **Second, what operations will be performed on it.**

Real-time application of Data Structures

- To store the contacts on our phone, then the software will simply place all our contacts in an array.
- Arrangement of leader-board of a game can be done simply through arrays
- Images are linked with each other. So, an image viewer software uses a linked list to view the previous and the next images using the previous and next buttons.
- Web pages can be accessed using the previous and the next URL links which are linked using linked list.

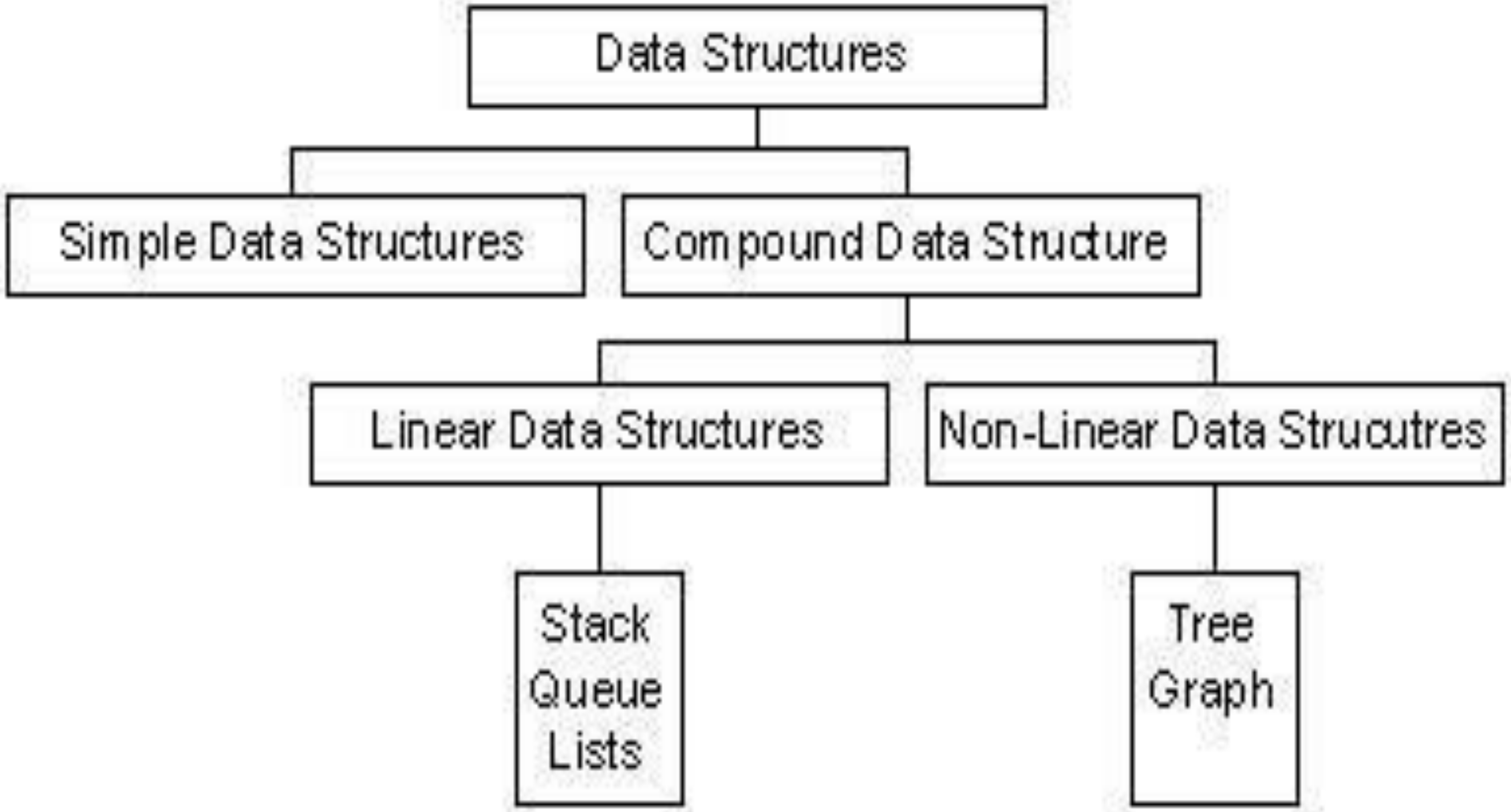
Real-time application of Data Structures

- Converting infix to postfix expressions
- Undo operation is also carried out through stack.
- Operating System uses queue for job scheduling.
- To handle congestion in networking queue can be used.
- Facebook’s Graph API uses the structure of Graphs.
- GPS navigation system also uses shortest path APIs.
- Databases also uses tree data structures for indexing
- Domain Name Server(DNS) also uses tree structures.

Real-time application of Data Structures

- Every time we type something to be searched in google chrome or other browsers, it generates the desired output based on the principle of hashing.

Classification of Data Structures



Simple and Compound Data Structures

- Simple Data Structure: Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

- Compound Data structure: Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

Linear and Non-linear Data Structures

- **Linear Data Structure:**

  Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

- **Non-Linear Data Structure:**

  Non-linear data structures can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

Operations on Data Structures

i. Add an element

ii. Delete an element

iii. Traverse

iv. Sort the list of elements

v. Search for a data element

Algorithm Definition

- An Algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

- The word algorithm originates from the Arabic word Algorism which is linked to the name of the Arabic Mathematician AI Khwarizmi.

- AI Khwarizmi is considered to be the first algorithm designer for adding numbers.

Structure of an Algorithm

- **An algorithm has the following structure:**
  - Input Step
  - Assignment Step
  - Decision Step
  - Repetitive Step
  - Output Step

Properties of an Algorithm

- Finiteness:- An algorithm must terminate after finite number of steps.
- Definiteness:-The steps of the algorithm must be precisely defined.
- Generality:- An algorithm must be generic enough to solve all problems of a particular class.
- Effectiveness:- The operations of the algorithm must be basic enough to be put down on pencil and paper.
- Input-Output:- The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps

Algorithm Analysis and Complexity

- The performances of algorithms can be measured on the scales of Time and Space.

- The Time Complexity of an algorithm or a program is a function of the running time of the algorithm or a program.

- The Space Complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run to completion.

Time Complexity

Complexity	Notation	Description
Constant	O(1)	Constant number of operations, not depending on the input data size.
Logarithmic	O(logn)	Number of operations proportional of log(n) where n is the size of the input data.
Linear	O(n)	Number of operations proportional to the input data size.
Quadratic	O(n^2)	Number of operations proportional to the square of the size of the input data.
Cubic	O(n^3)	Number of operations proportional to the cube of the
		size of the input data.
Exponential	O(2^n)	Exponential number of operations, fast growing.
	O(k^n)	

Searching Methods

- **Search:**
  Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

- There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

  1. Linear or Sequential Search
  2. Binary Search

Linear Search

- Begins search at first item in list, continues searching sequentially(item by item) through list, until desired item(key) is found, or until end of list is reached.

Also called sequential or serial search.

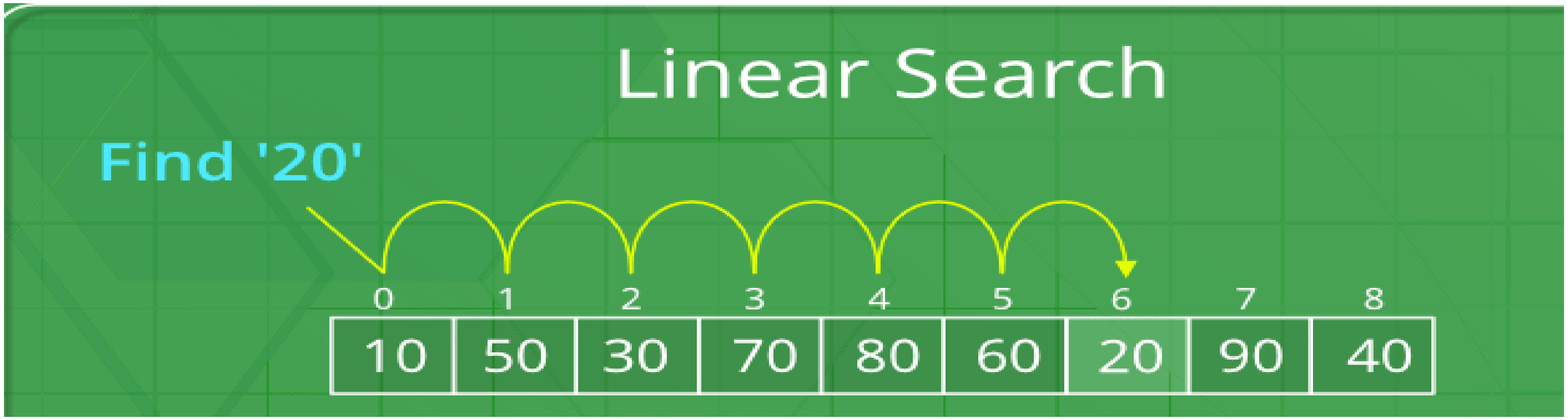
- Obviously not an efficient method for searching ordered lists like phone directory(which is ordered alphabetically).

- **Advantages**
  1. Algorithm is simple.
  2. List need not be ordered in any particular way.

- Time Complexity of Linear Search is O(n).

Linear Search

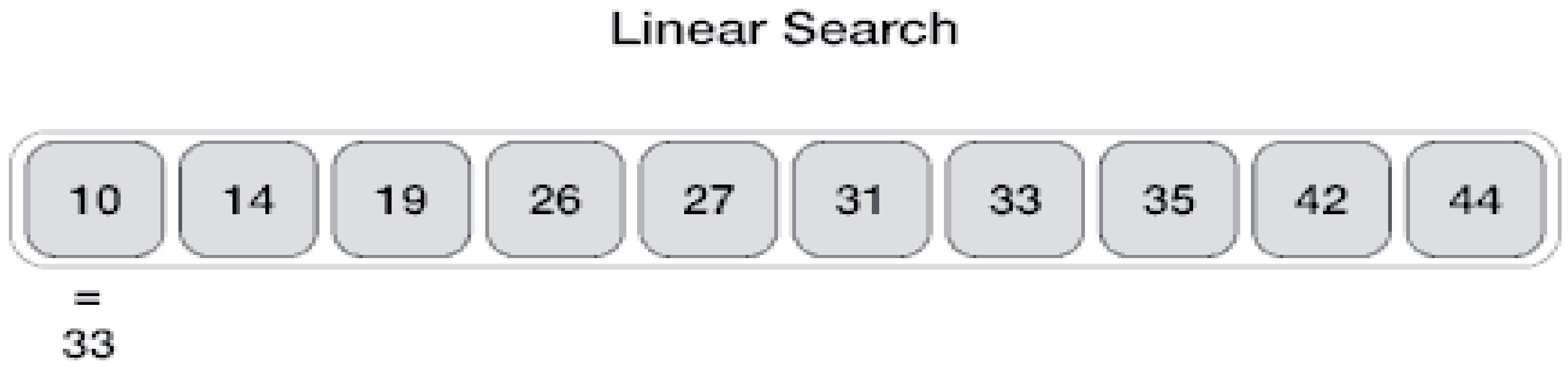
Example:



Linear Search

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one.

- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Solution:
A simple approach is to do **linear search**, i.e
Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
If x matches with an element, return the index.
If x doesn't match with any of elements, return -1.

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Linear Search

Algorithm:

Linear Search (Array A, Value x) :

```
Step 1: Set i to 0
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
 Step 8: Exit
```

Linear Search

```
#LINEAR SEARCH
items=[]
n= int(input("Enter  how many elements u want "))
print("enter %d elements"%n)
for i in range(0,n):
    element=int(input())
    items.append(element)
print(items)
x=int(input("enter element to search:"))
i=flag=0
while(i<len(items)):
    if((items[i])==x):
        flag=1
        break
    i=i+1
if(flag==1):
    print("location number statrts from 0 and found at location ",i)
else:
      flag=-1
      print("not found")
```

Output1:

```
Enter  how many elements u want 5
enter 5 elements
12
34
67
89
90
[12, 34, 67, 89, 90]
enter element to search:89
location number statrts from 0 and found at location  3
```

Output2:

```
Enter  how many elements u want 3
enter 3 elements
12
89
67
[12, 89, 67]
enter element to search:100
not found
```

Binary Search

- An algorithm to solve this task looks at the middle of the array or array segment first

- If the value looked for is smaller than the value in the middle of the array

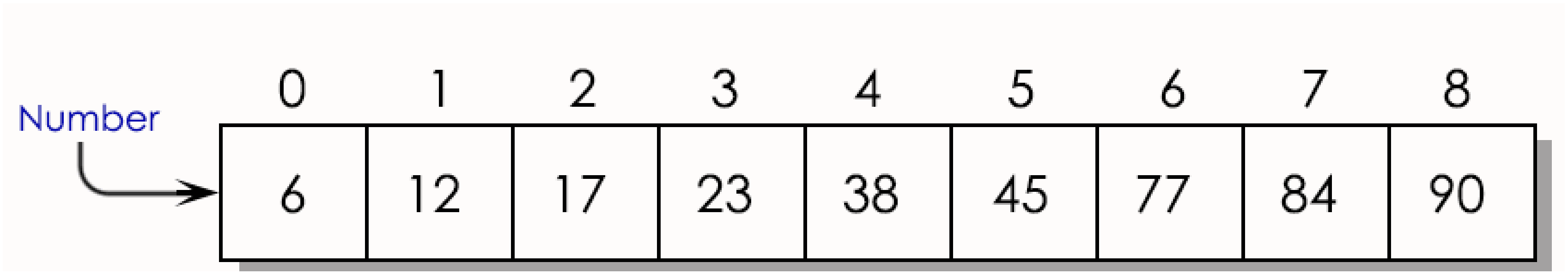
- Then the second half of the array or array segment can be ignored

- This strategy is then applied to the first half of the array or array segment

Binary Search

Binary Search : In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomies divide-and-conquer search algorithm and executes in logarithmic time.

Step by step example:

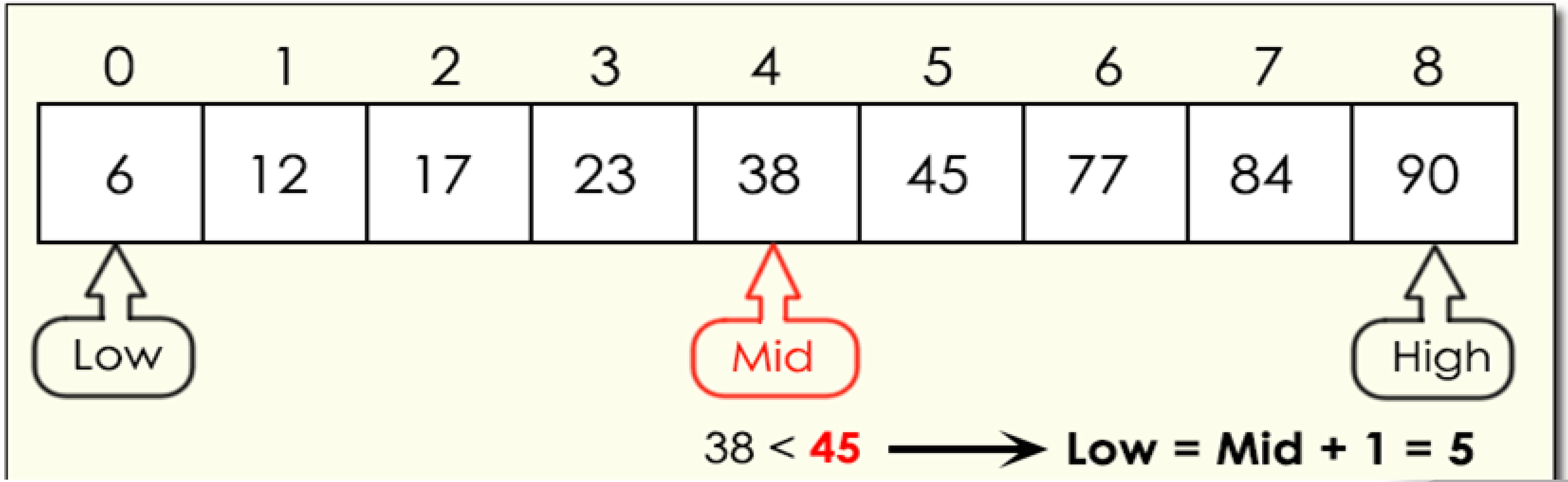


Binary Search

	Low	High	Mid
#1	0	8	4

Search (45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



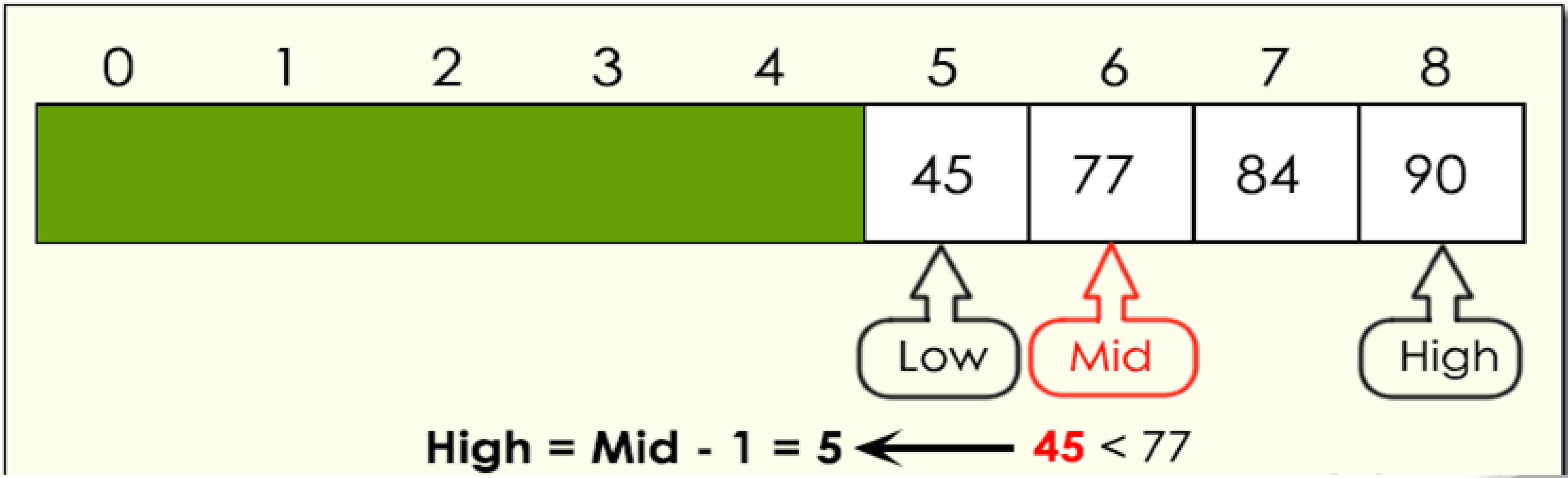
38 < **45** $\longrightarrow$ **Low = Mid + 1 = 5**

Binary Search

	Low	High	Mid
#1	0	8	4
#2	5	8	6

Search (45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



High = Mid - 1 = 5 ← 45 < 77

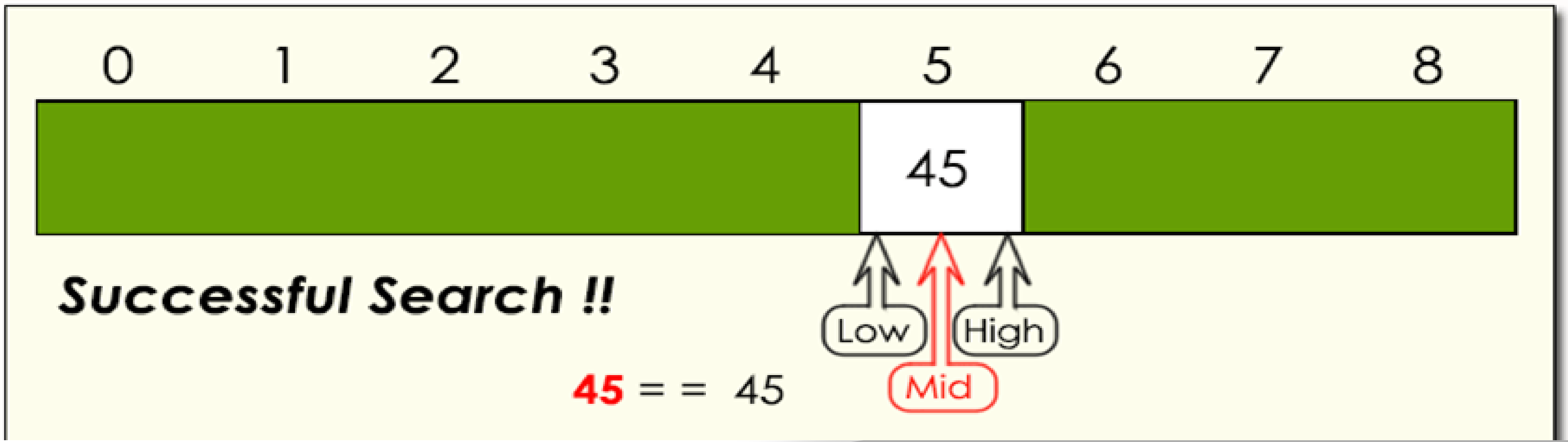
Binary Search

Example1:

	Low	High	Mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

Search (45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

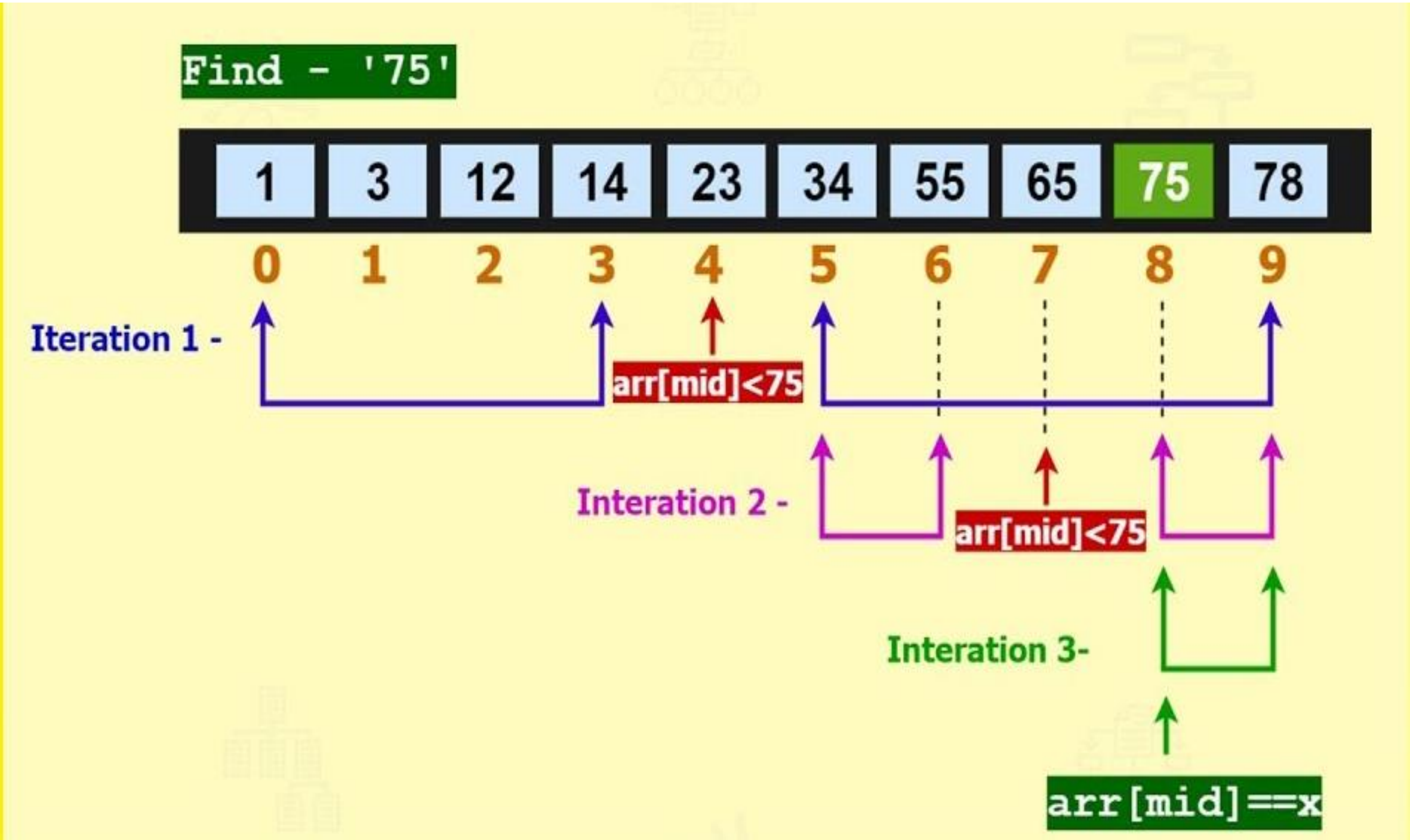


Successful Search !!

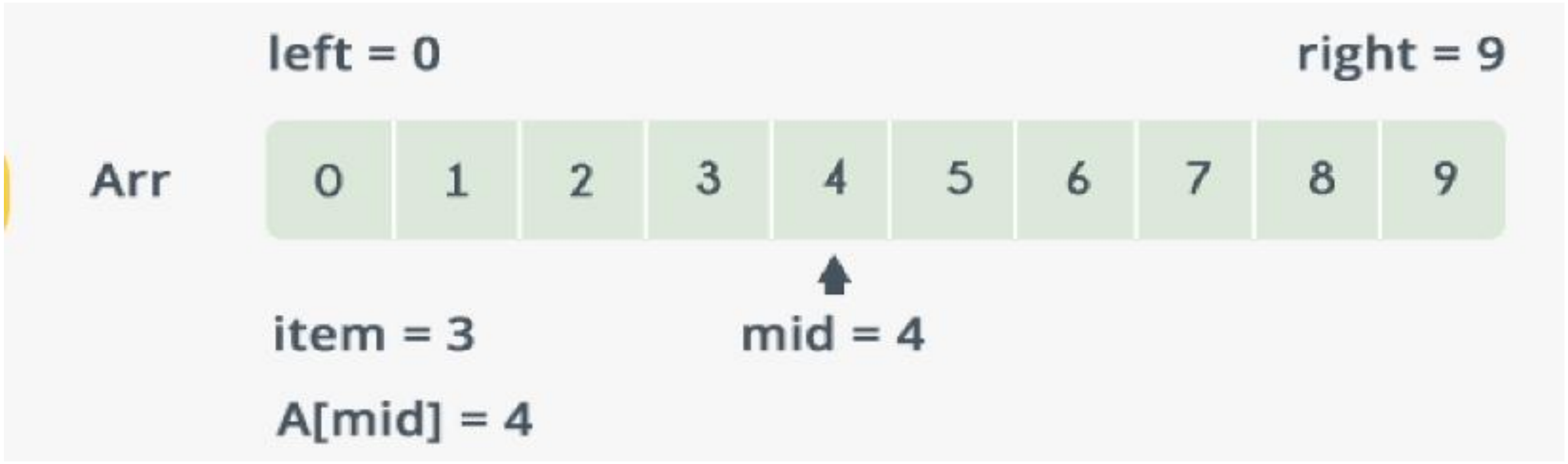
45 = = 45

Binary Search

Example2:



Example3:



Binary Search

Algorithm Binary Search

Implement binary search following the below steps:

1.Start with the middle element of the given list:
   - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
   - Otherwise, compare the middle element with the target value,
      - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
      - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.

2.If a match is found, return the index of the element matched.

3.Otherwise, return -1

## Binary Search

#Binary Search

```
item_list=[]
n=int(input("enter how many elements u want?"))
print("enter %d elements in sorted order"%n)
for i in range(0,n):
    element=int(input())
    item_list.append(element)
#item_list.sort()
print(item_list)
item=int(input("enter element to be searched"))
```

Binary Search

```
first = 0
last = len(item_list)-1
found = False
while( first<=last and not found):
    mid = (first + last)//2
    if item_list[mid] == item :
        found = True
    else:
        if item < item_list[mid]:
            last = mid-1
        else:
            first = mid + 1
if(found==True):
    print("element is found at location ",mid)
else:
    print("element is not found")
```

Binary Search

Output:

```
enter how many elements u want?5
enter 5 elements in sorted order
12
67
89
90
100
[12, 67, 89, 90, 100]
enter element to be searched89
element is found at location  2
```

Sorting techniques

Sorting:
is any process of arranging items systematically, and has two common, yet distinct meanings:
ordering: arranging items in a sequence ordered by some criterion;

categorizing: grouping items with similar properties.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

Sorting techniques: Bubble sort, selection sort, insertion sort

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Bubble Sort

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

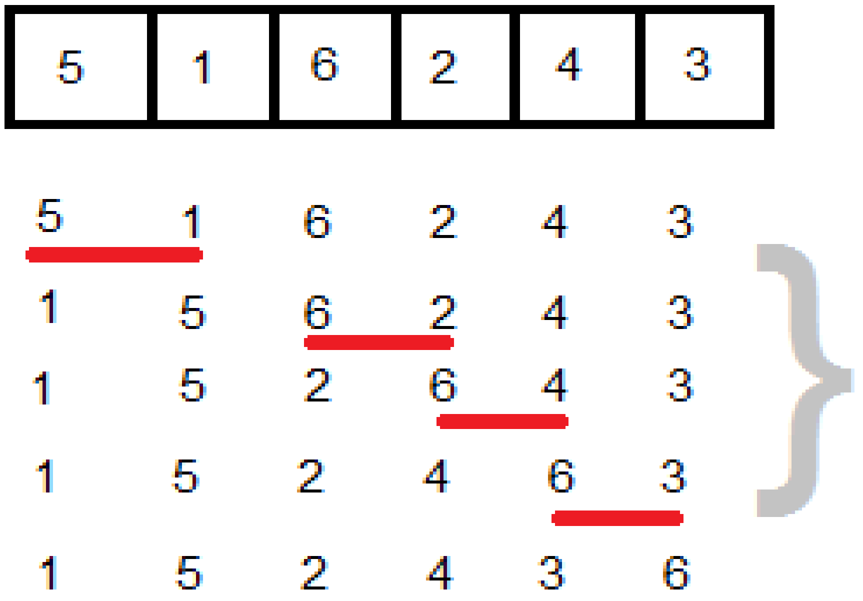
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Bubble Sort

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubble Sort Example



Lets take this Array.

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

Bubble Sort

Python program for implementation of Bubble Sort

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Bubble Sort

```
# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Output:

Sorted array is: 11 12 22 25 34 64 90

Selection Sort

Selection Sort:
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

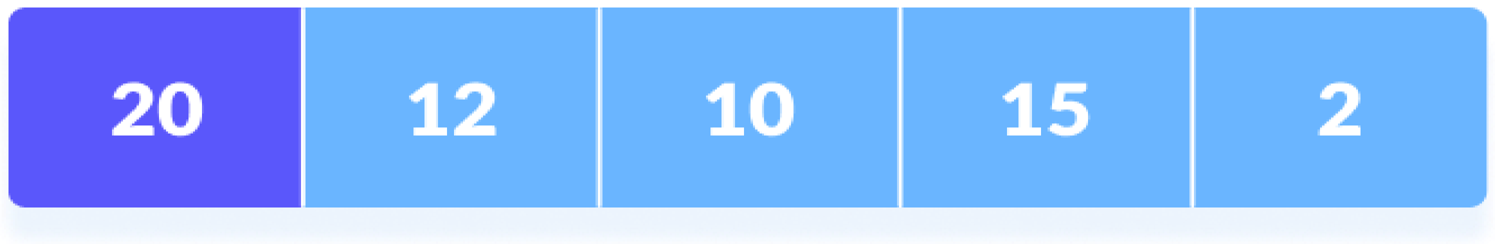
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Selection Sort Example

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

How Selection Sort Works?

1.Set the first element as minimum



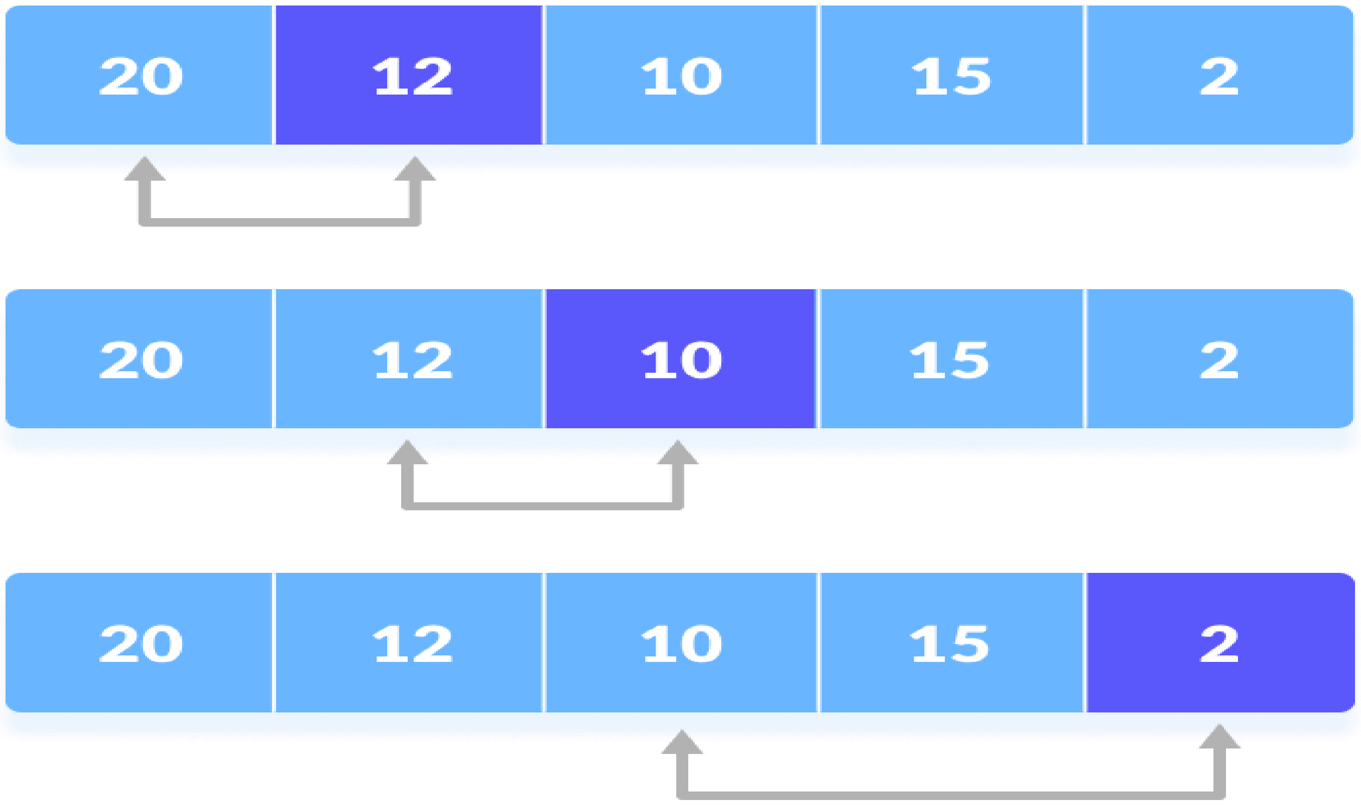
Select first element as minimum

Selection Sort Example

2.Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum

Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

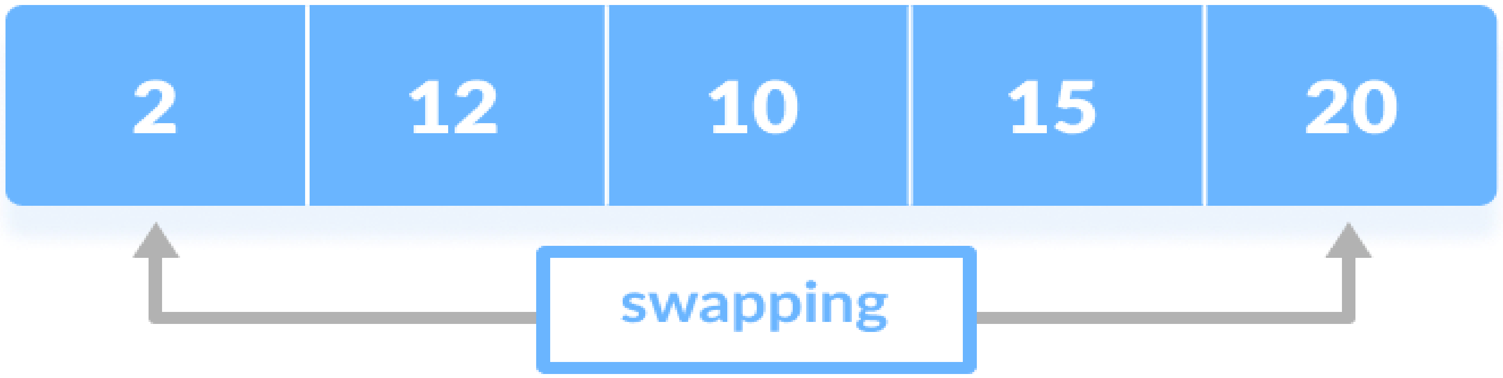
Selection Sort Example Cont..



Compare minimum with the remaining elements

Selection Sort Example Cont..

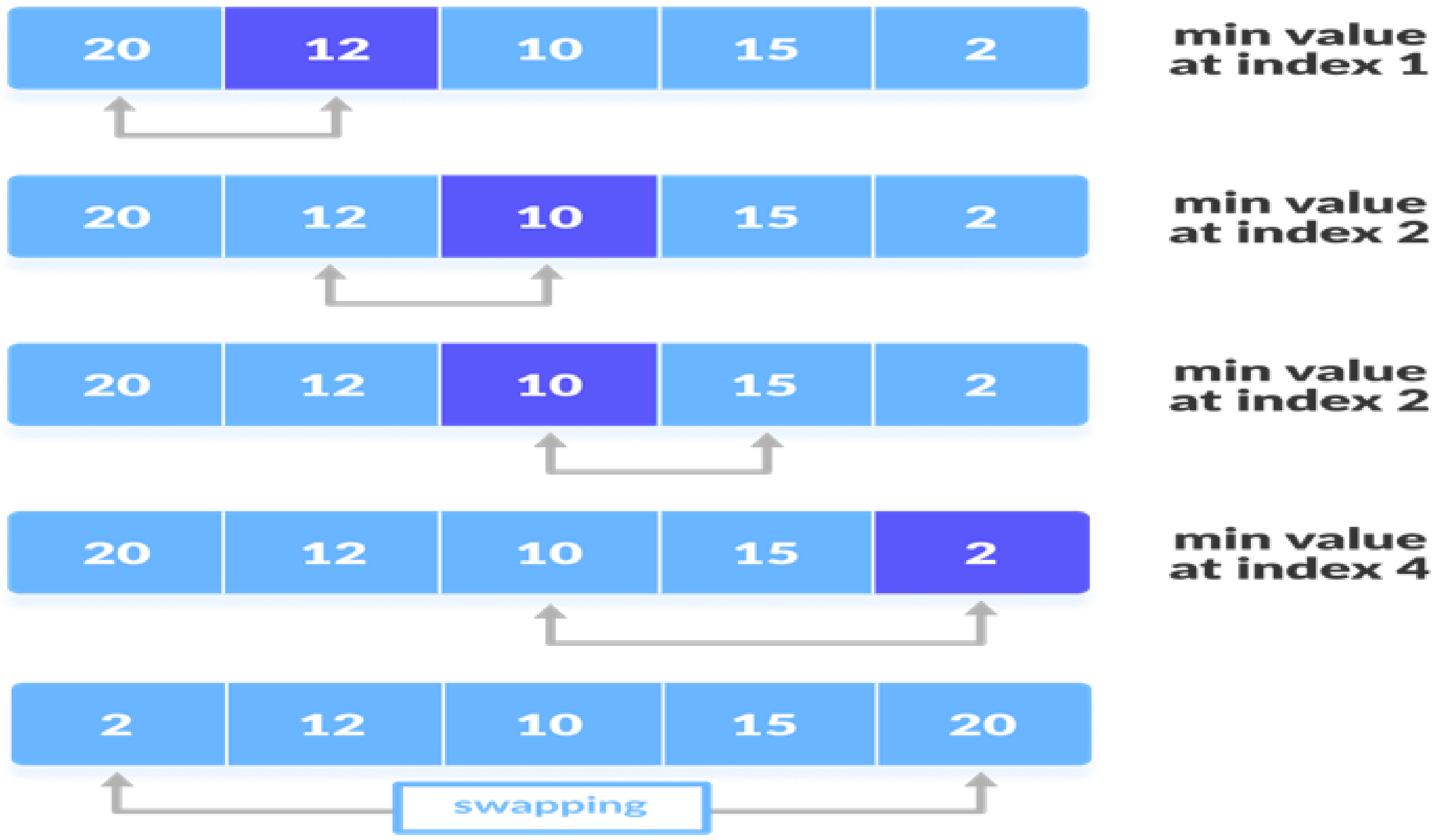
3.After each iteration, minimum is placed in the front of the unsorted list



Swap the first with minimum

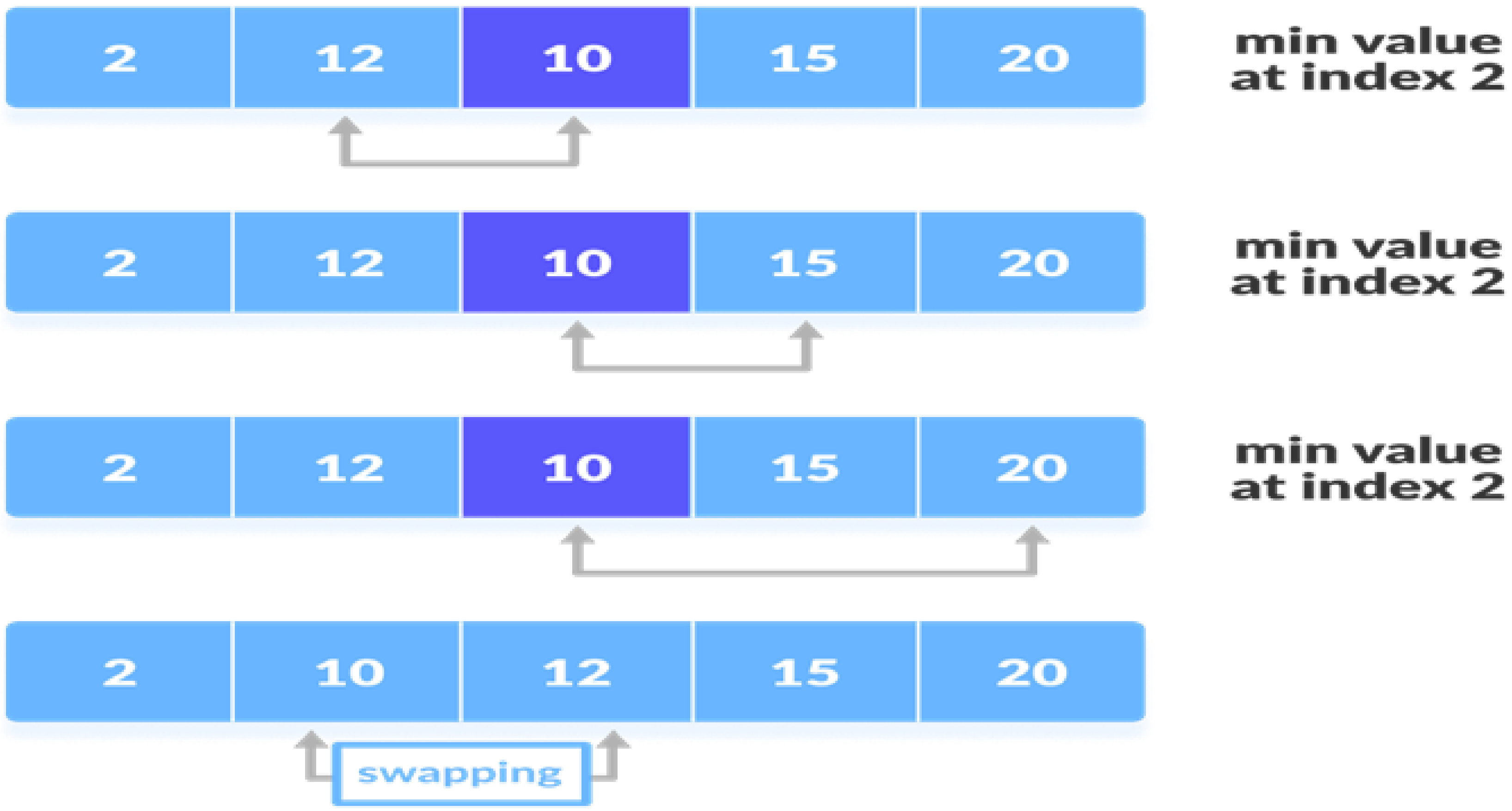
4.For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

Selection Sort Example Cont..



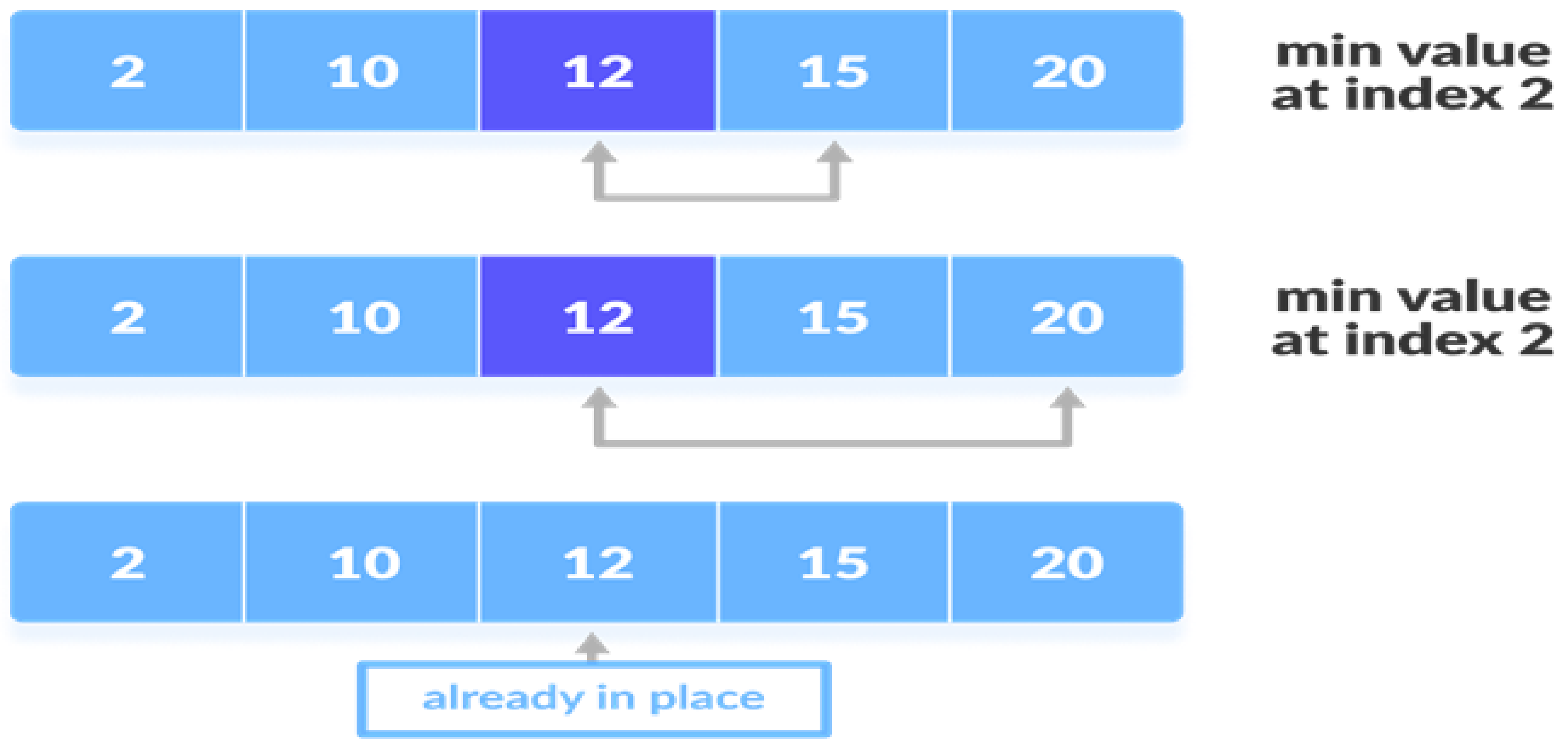
The first iteration

Selection Sort Example Cont..



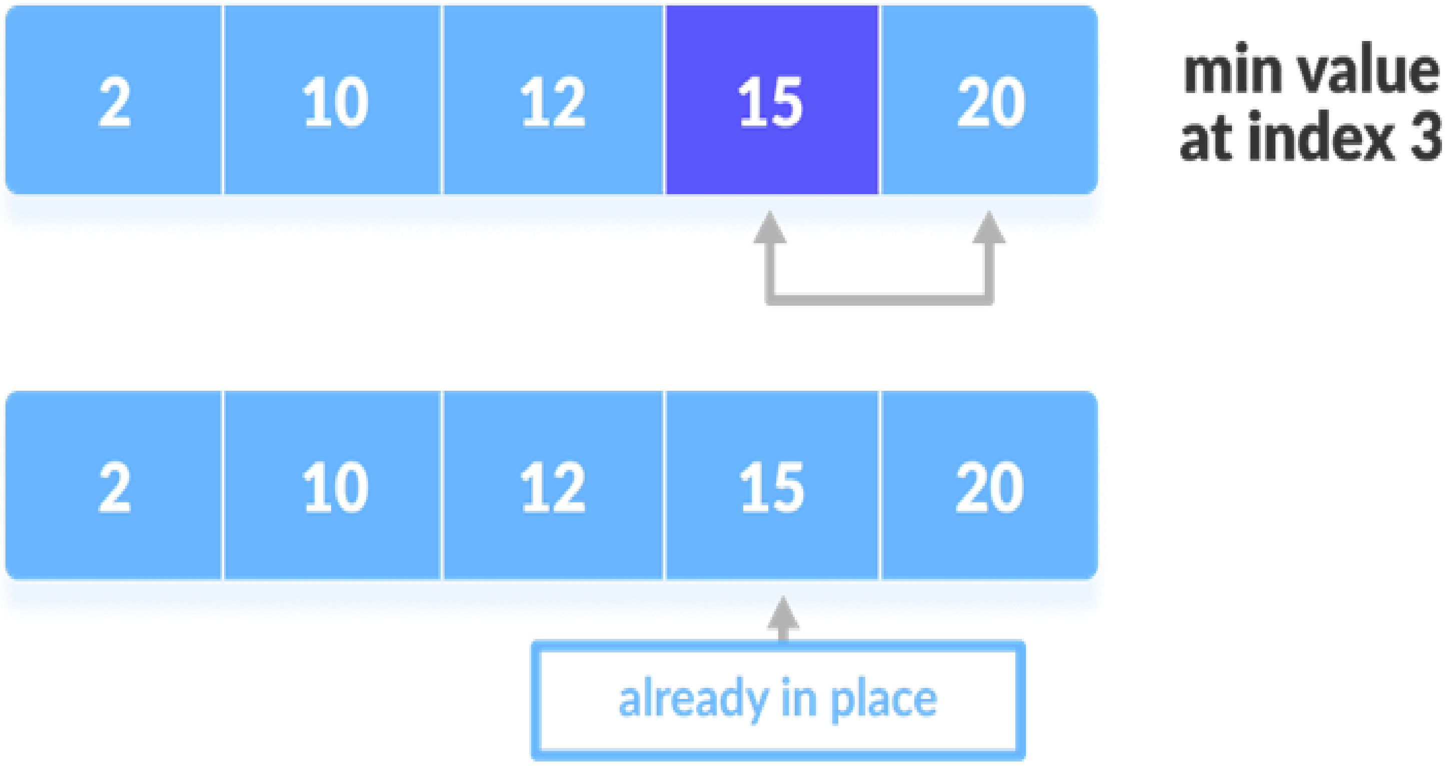
The second iteration

Selection Sort Example Cont..



The third iteration

Selection Sort Example Cont..



The fourth iteration

Selection Sort Example

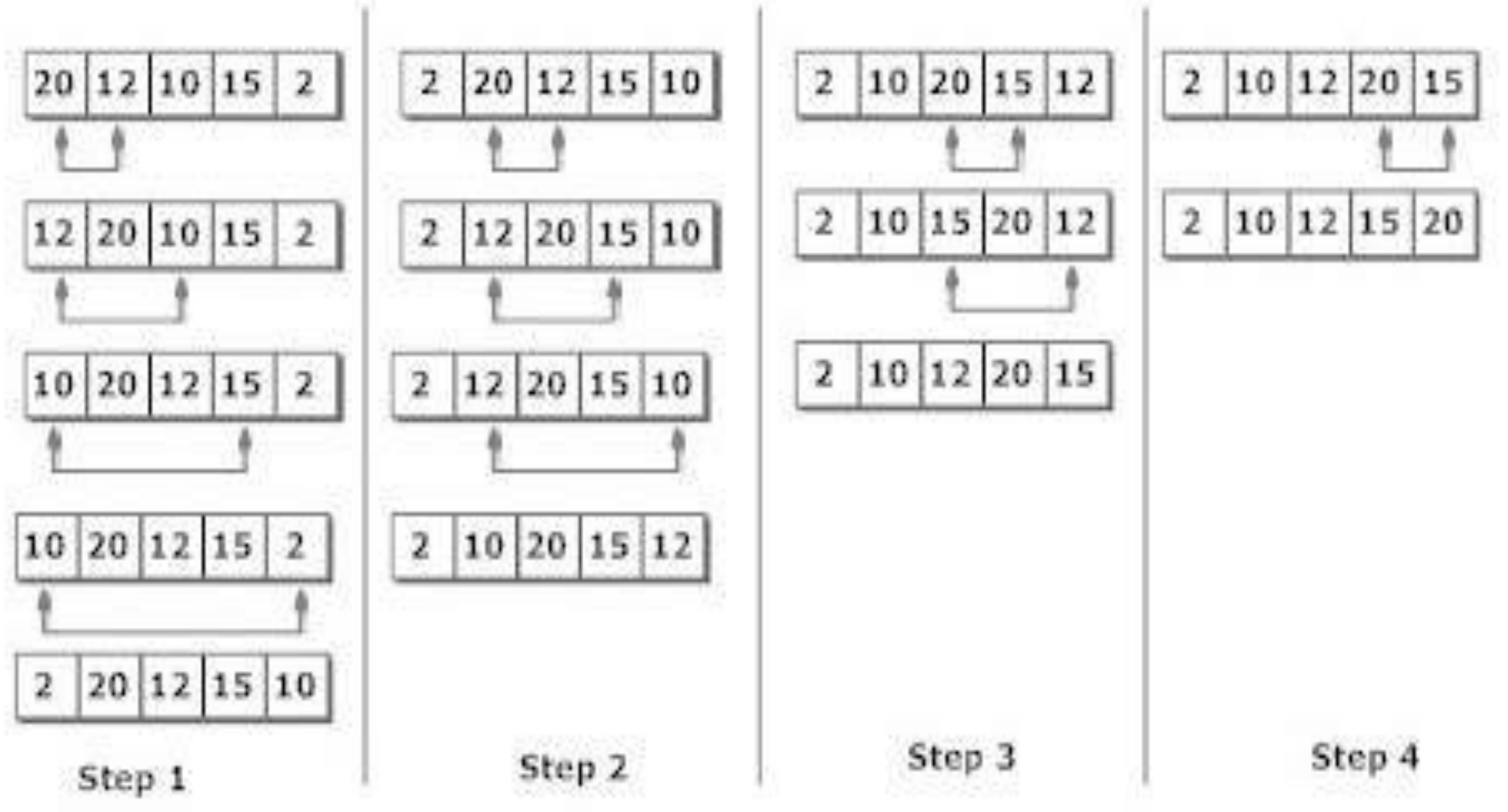
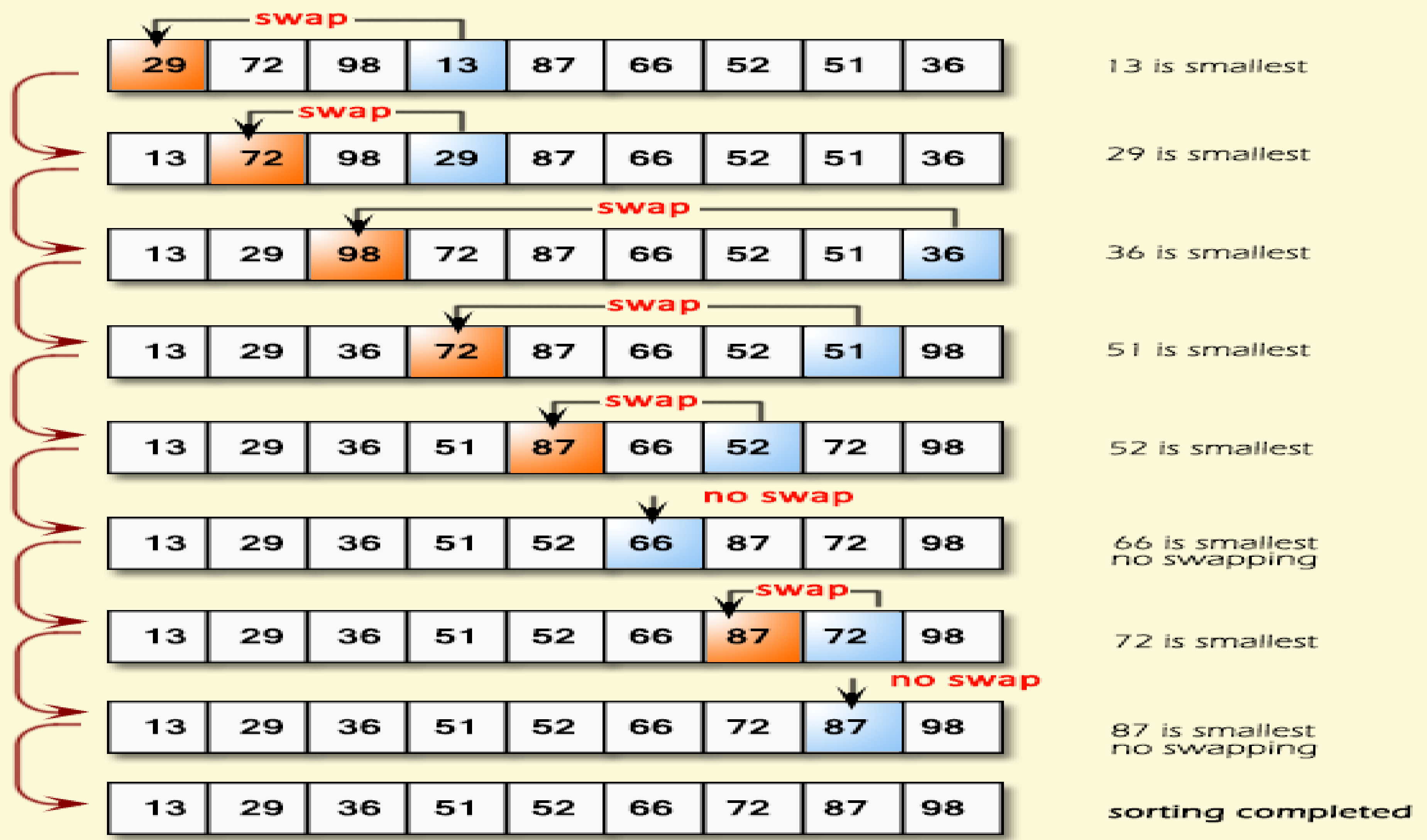
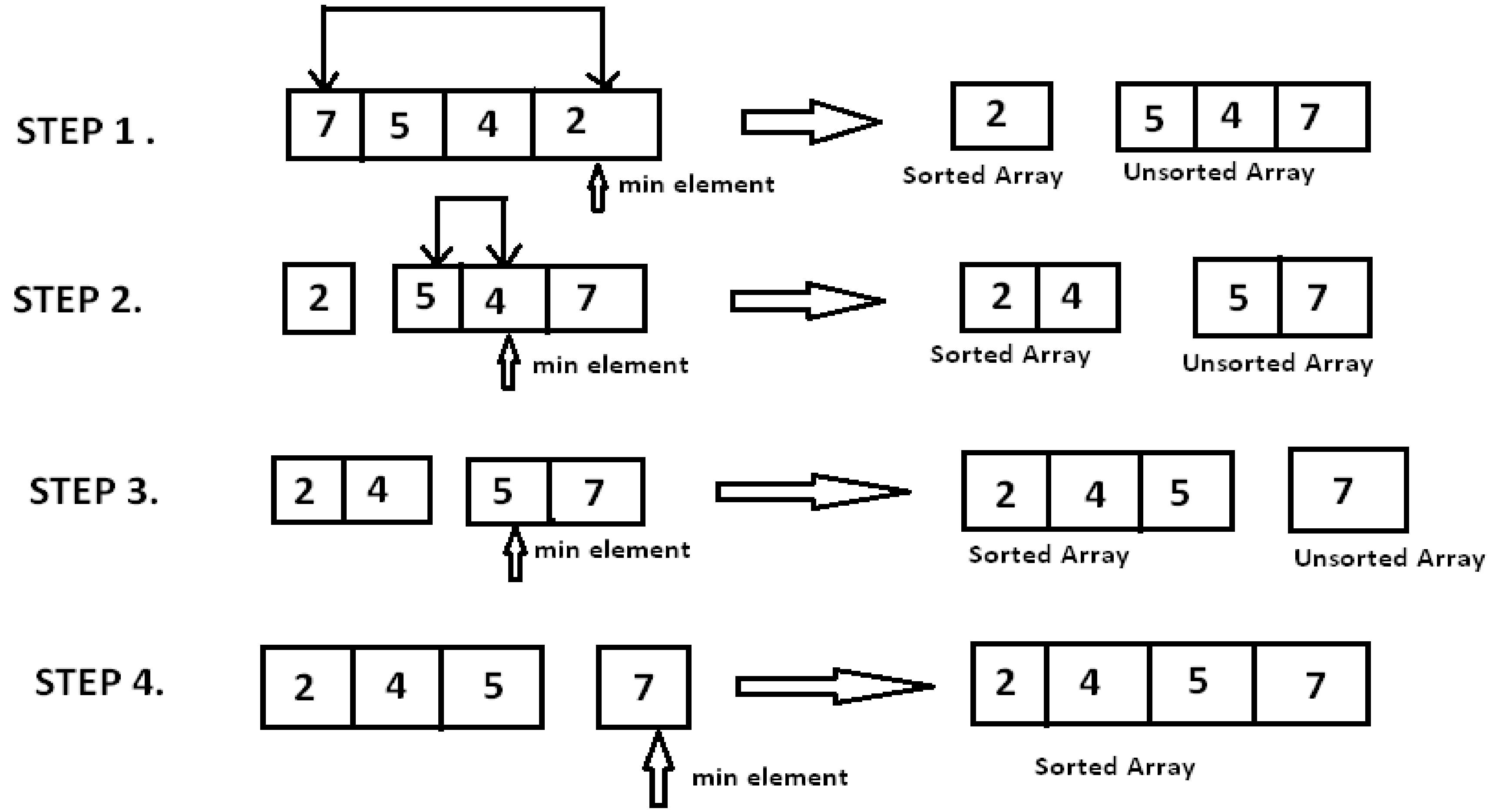


Figure: Selection Sort

Selection Sort Example



Selection Sort Example



Selection Sort

```
# Python program for implementation of Selection
# Sort
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]
```

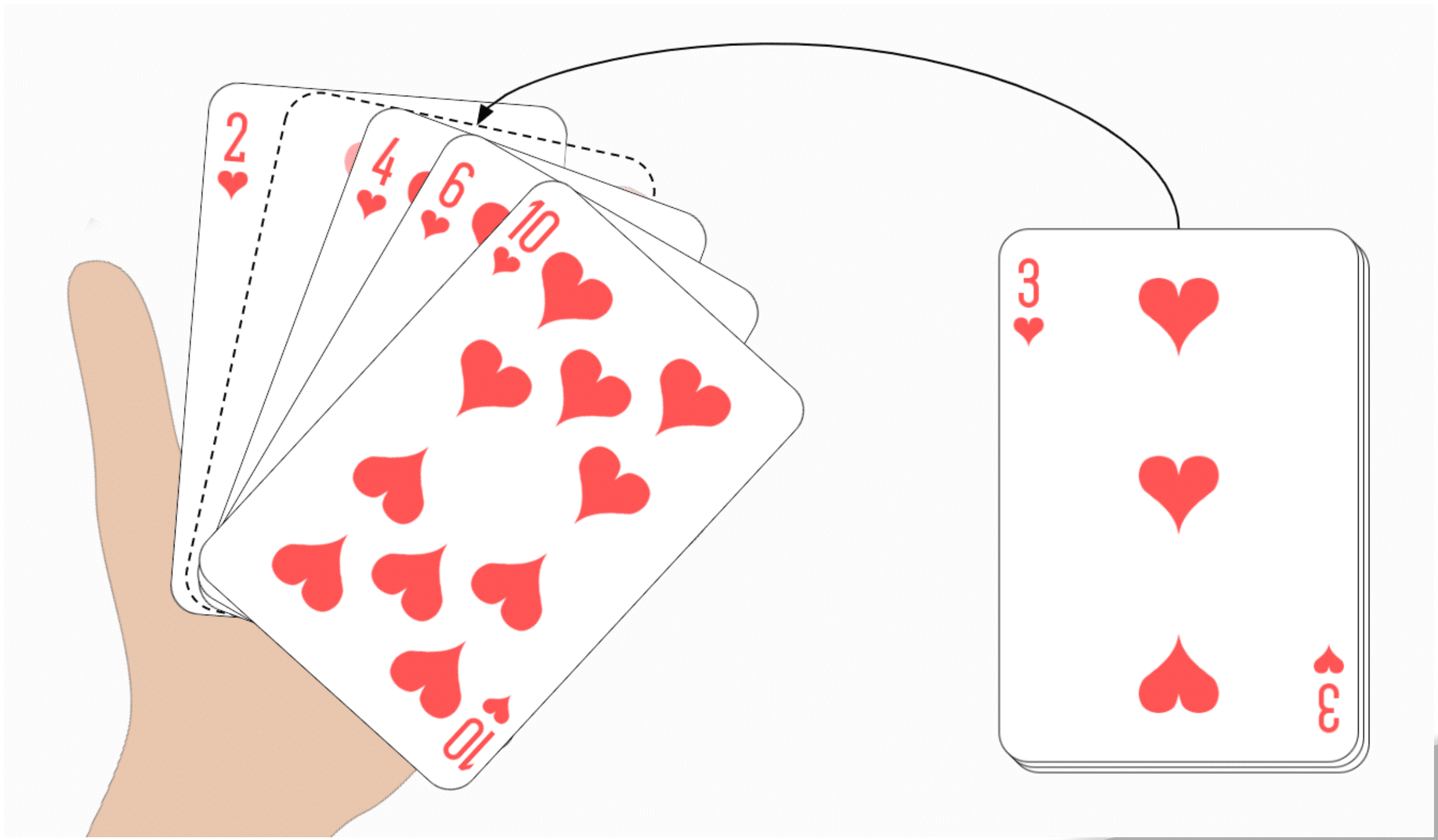
```
# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

Output:
Sorted array: 11 12 22 25 64

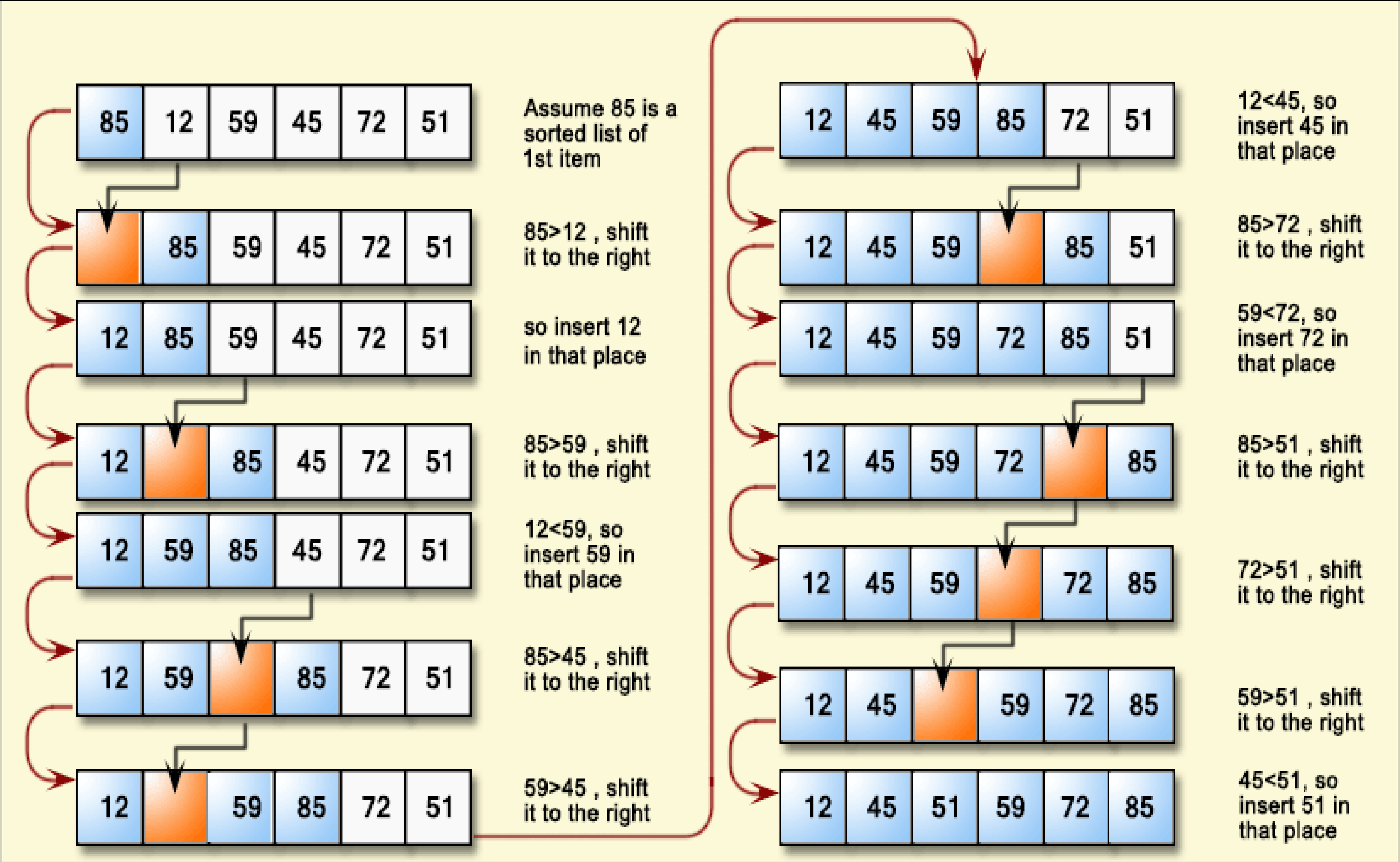
Insertion Sort

Insertion Sort:
Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion Sort

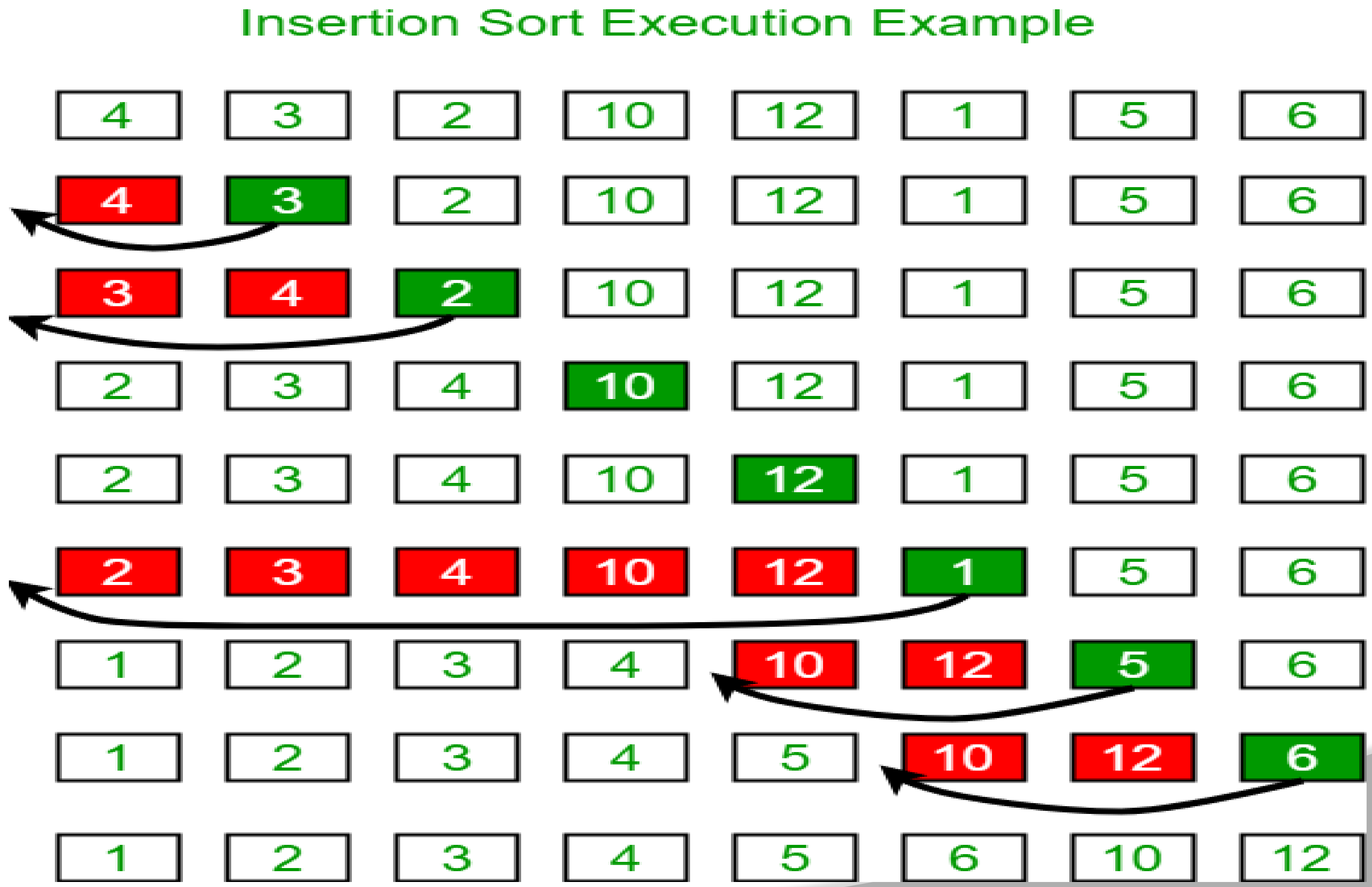


Insertion Sort



85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
_ 85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 _ 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 _ 85 72 51
85>45 , shift it to the right
12 _ 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 _ 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 _ 85
85>51 , shift it to the right
12 45 59 _ 72 85
72>51 , shift it to the right
12 45 _ 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place

Insertion Sort



Insertion Sort

6 5 3 1 8 7 2 4

Insertion sort

```
# Python program for implementation of Insertion Sort

arr=[]
n=int(input("enter how many elements u want?"))
print("enter %d elements"%n)
for i in range(0,n):
    element=int(input())
    arr.append(element)
for i in range(1, len(arr)):
    key = arr[i]
```

Insertion sort

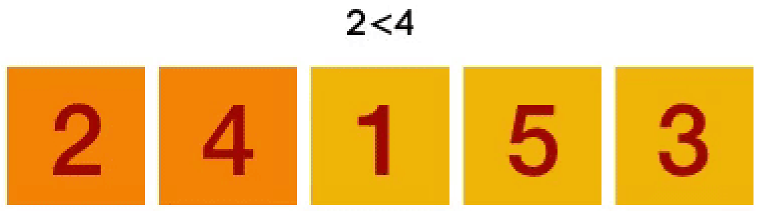
```
 # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
    j = i-1
    while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
    arr[j+1] = key
print ("Sorted array is:")
print (arr)
```

comparison of sorting algorithms

Bubble Sort

6 5 3 1 8 7 2 4

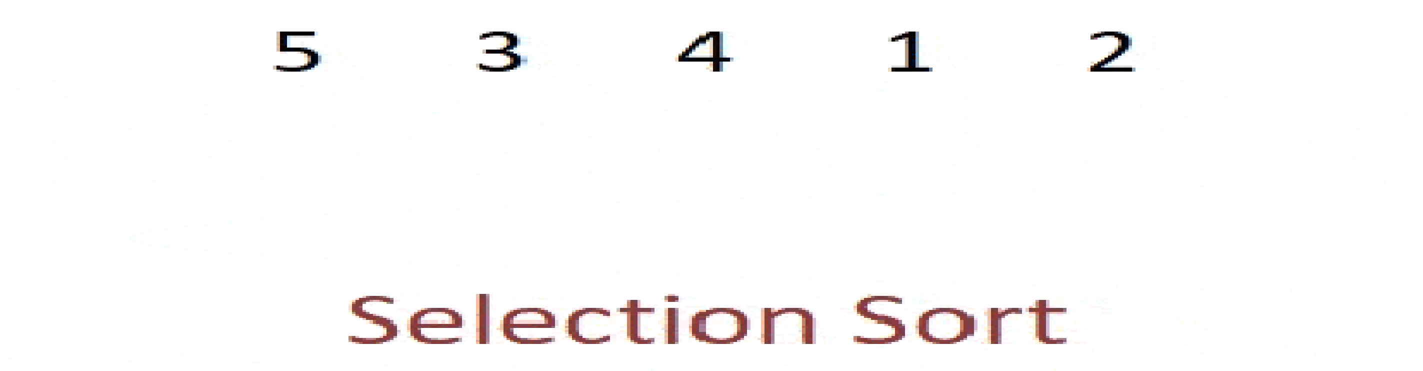
Bubble Sort



Bubble Sort

- The **time complexity of Bubble Sort** is O(n^2).
- The main advantage of **Bubble Sort** is the simplicity of the algorithm.
- The space **complexity** for **Bubble Sort** is O(1)

Selection Sort



Selection sort

In computer science, selection sort is an in-place comparison **sorting algorithm**. It has an O(n^2) time complexity, which makes it inefficient on large lists, and generally performs worse than the similar **insertion sort**.

Advantages:

1.It is simple and easy to implement

2.It can be used for small data sets

3.It is 60 percent more efficient than bubble sort

Disadvantage:

In case of large data sets, the efficiency of selection sort drops as compared to insertion sort

Insertion sort

The average case often has the same **complexity** as the worst case. So **insertion sort**, on average, takes) O(n2) **time**. **Insertion sort** has a fast best-case running **time** and is a good **sorting** algorithm to use if the input list is already mostly **sorted**.

Advantages:

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

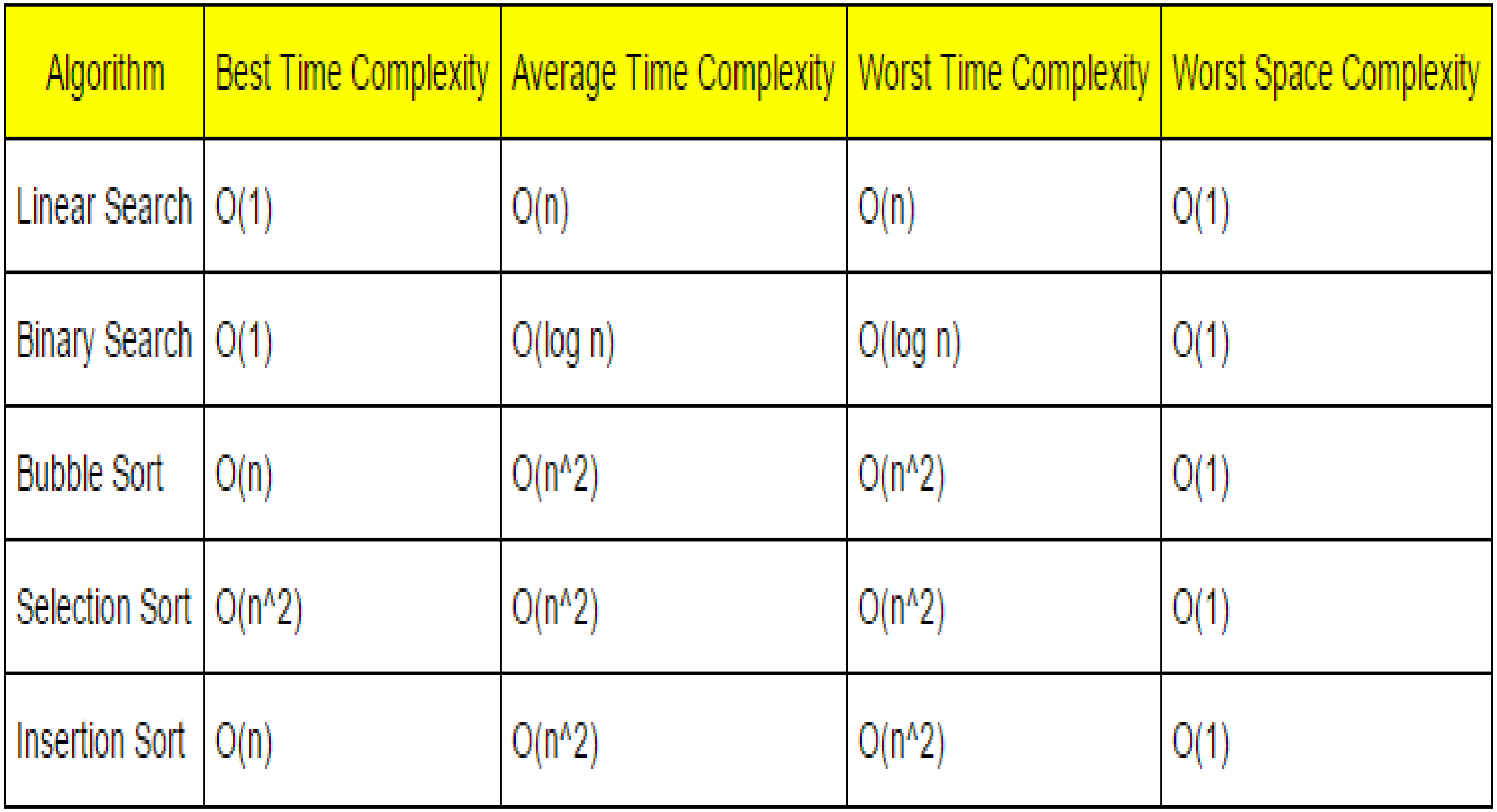
2.It can be efficiently implemented on data sets that are already substantially sorted

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

4.It is 40 percent faster than the selection sort

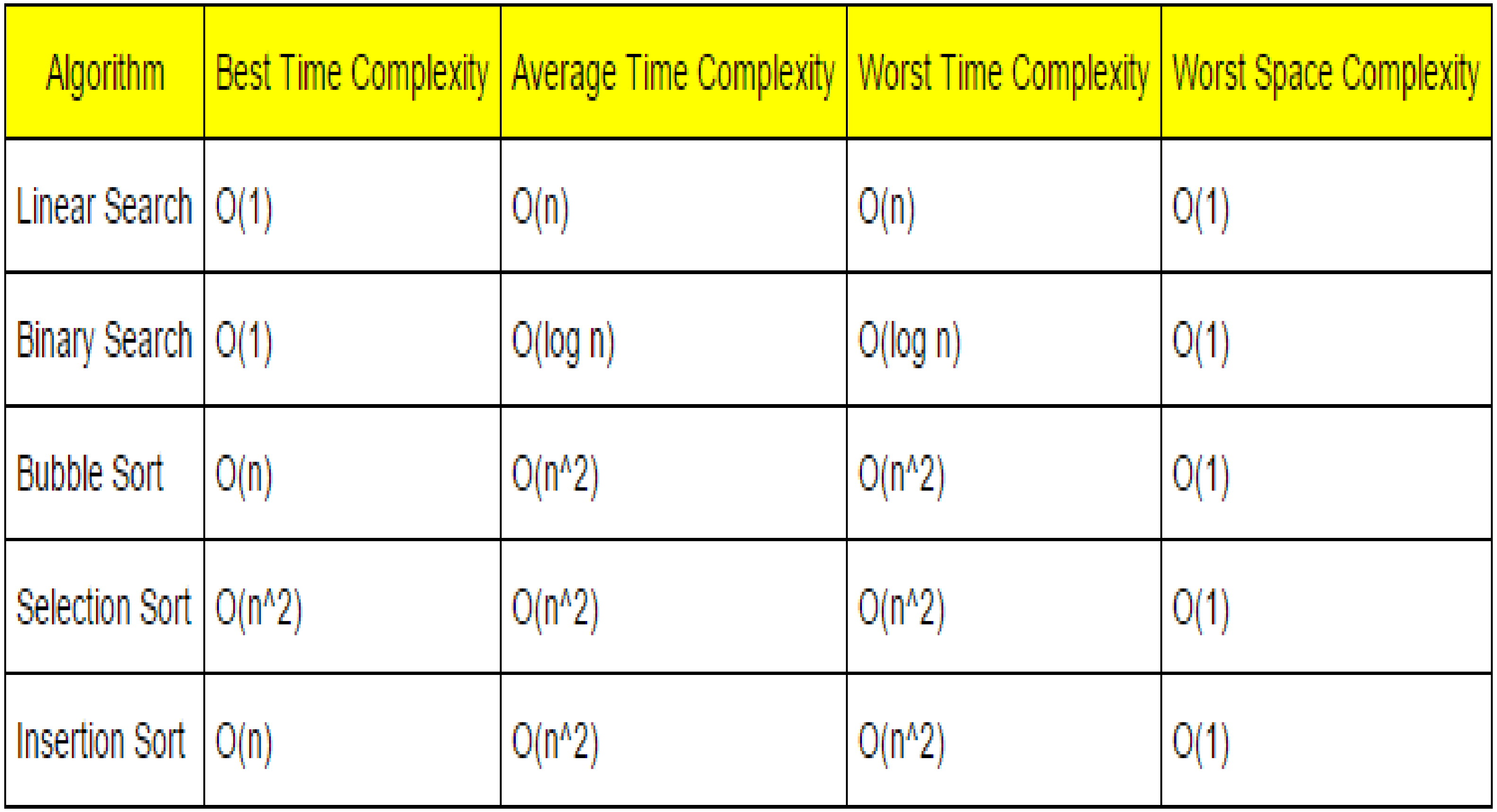
5.It is said to be online, as it can sort a list as and when it receives new element

Sorting And Searching Algorithms – Time Complexities



Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)

Sorting And Searching Algorithms - Time Complexities



Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)

MODULE -II

Stacks: Primitive operations, implementation of stacks using arrays, applications of stacks arithmetic expression conversion and evaluation; Queues: Primitive operations; Implementation of queues using Arrays, applications of linear queue, circular queue and double ended queue (deque).

Stack

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.

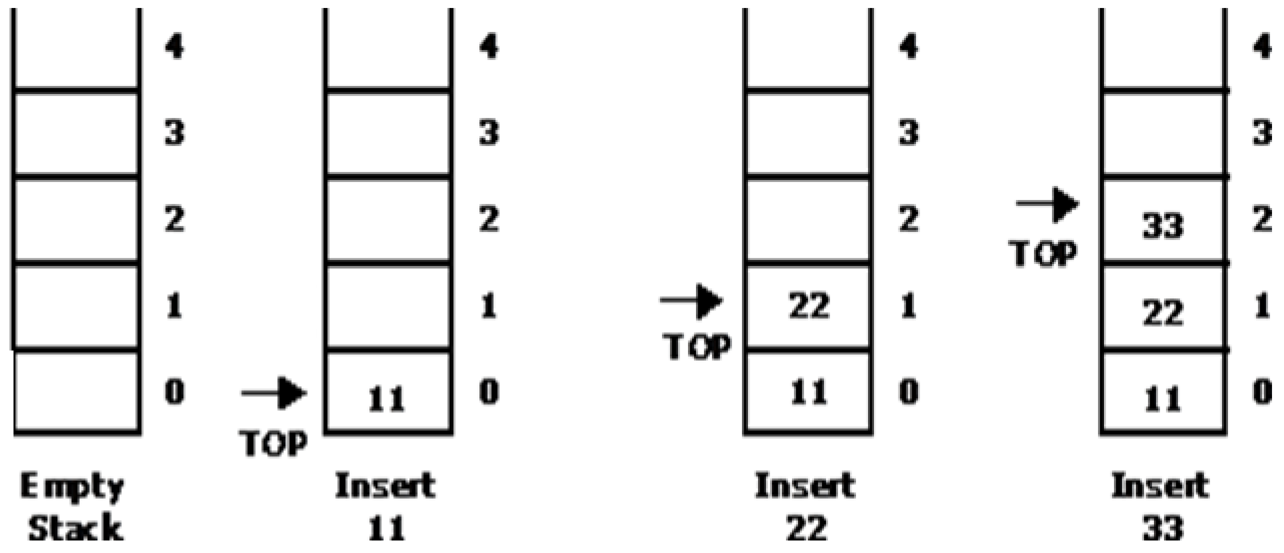
- The elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

- Stack is also known as a LIFO (Last in Fast out) list or Push down list.

Basic Stack Operations

- **PUSH:** It is the term used to insert an element into a stack.

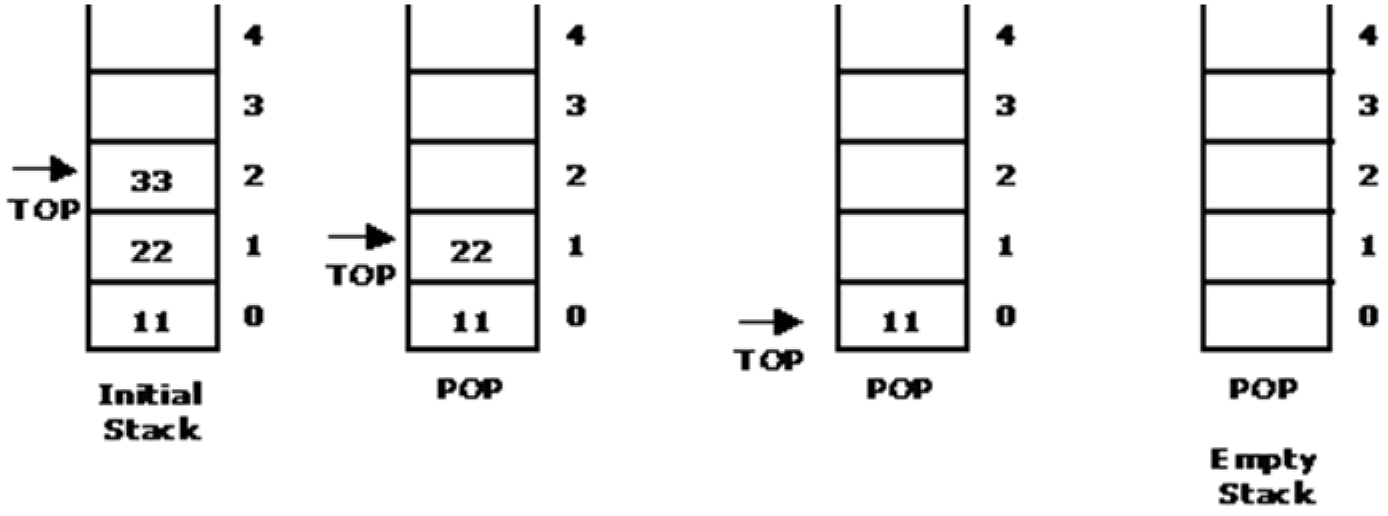
Top=-1



PUSH operations on stack

Basic Stack Operations

POP: It is the term used to delete an element from a stack.



Top=-1

POP operation on a stack

Standard Error Messages in Stack

- Two standard error messages of stack are
  - **Stack Overflow:** If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition.
  - **Stack Underflow:** If we attempt to remove elements beyond the base of the stack, we will encounter a stack underflow condition.

Stack Operations

- **PUSH** (STACK, TOP, MAXSTR, ITEM): This procedure pushes an ITEM onto a stack

```
1. If TOP = MAXSIZE-1, then Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1 [Increases TOP by 1]
3. Set STACK [TOP] := ITEM. [Insert ITEM in TOP position]
4. Return
```

- **POP** (STACK, TOP, ITEM): This procedure deletes the top element of STACK and assign it to the variable ITEM

```
1. If TOP = -1, then Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]
3. Set TOP := TOP - 1 [Decreases TOP by 1]
4. Return
```

Python code to demonstrate Implementing stack

```
# Python code to demonstrate Implementing
# stack using list
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack)
```

Output :

```
['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']
 Iqbal
['Amar', 'Akbar', 'Anthony', 'Ram']
 Ram
 ['Amar', 'Akbar', 'Anthony']
```

Applications of Stack

- Converting algebraic expressions from one form to another. E.g. Infix to Postfix, Infix to Prefix, Prefix to Infix, Prefix to Postfix, Postfix to Infix and Postfix to prefix.
- Evaluation of Postfix expression.
- Parenthesis Balancing in Compilers.
- Depth First Search Traversal of Graph.
- Recursive Applications.

Arithmetic Expression

- **Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. E.g.: (A + B) * (C - D)

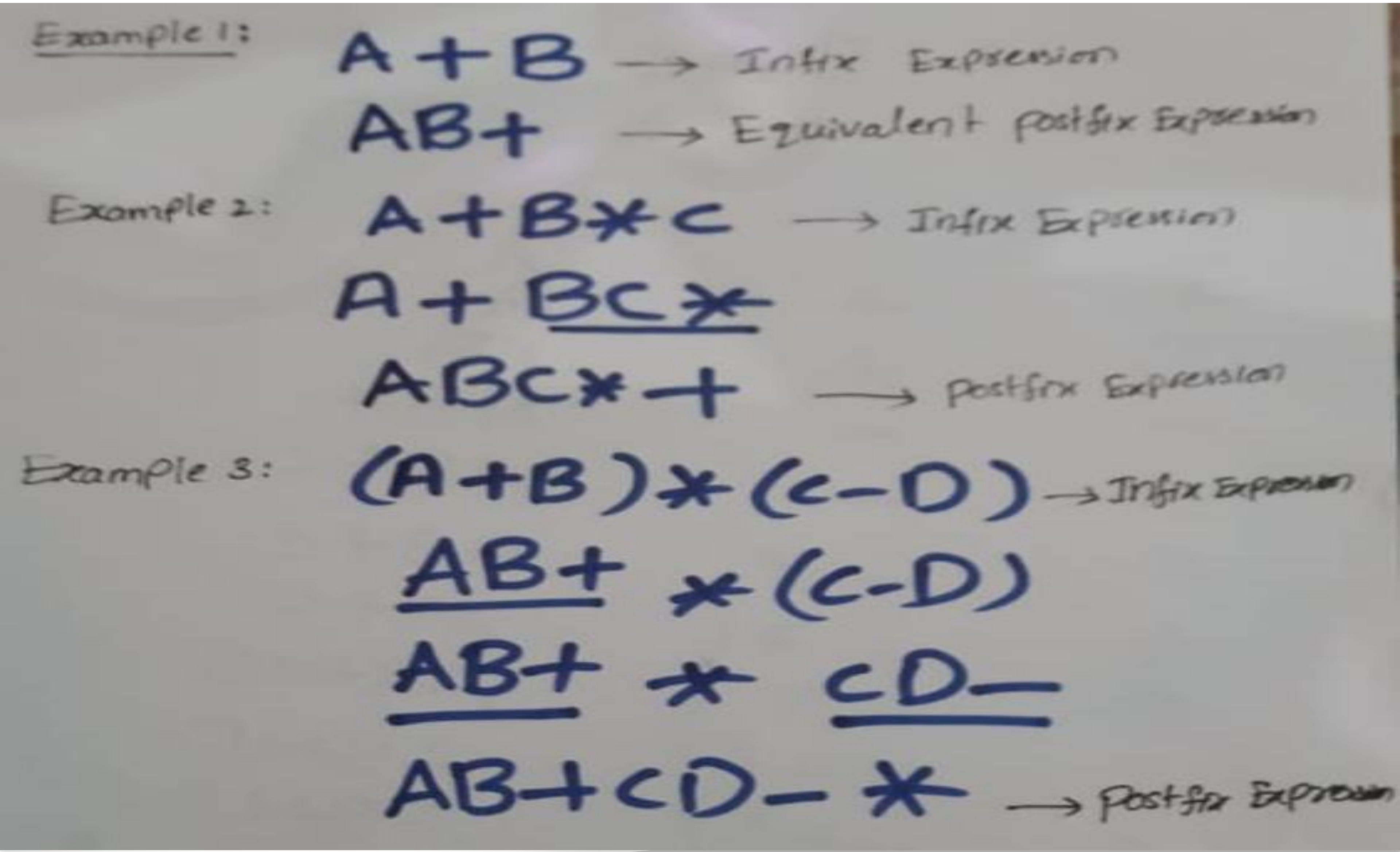
- **Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands.

  The prefix notation is called as polish notation. E.g.: * + A B – C D

- **Postfix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands.

  The postfix notation is called as suffix notation and is also referred to reverse polish notation. E.g: A B + C D - *

Conversion of Infix to Postfix



Example 1: A + B → Infix Expression

AB+ → Equivalent postfix Expression

Example 2: A + B * C → Infix Expression

$A + \underline{BC*}$

ABC*+ → Postfix Expression

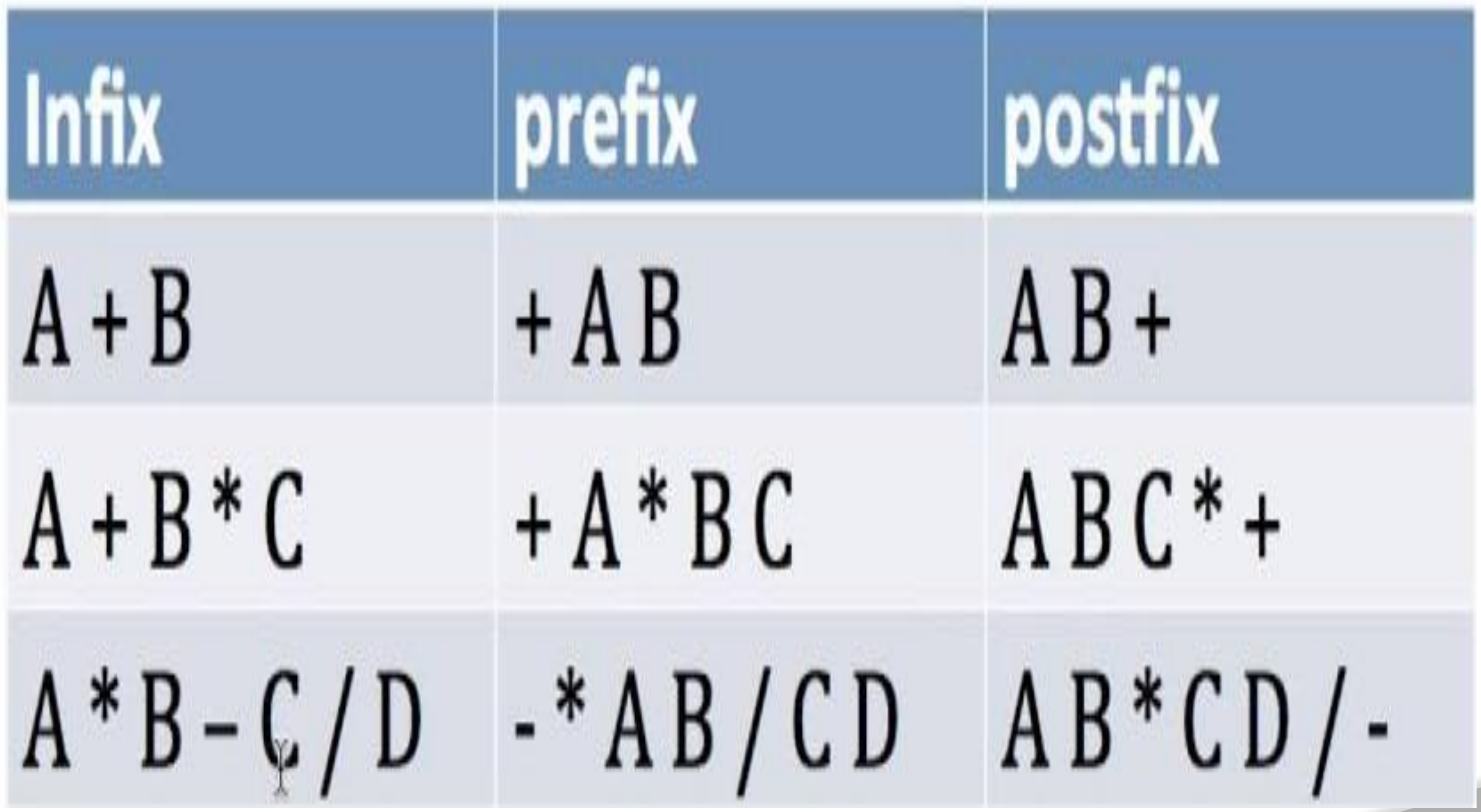
Example 3: (A+B)*(C−D) → Infix Expression

$\underline{AB+} * (C-D)$

$\underline{AB+} * \underline{CD-}$

AB+CD−* → Postfix Expression

Conversion of Infix to Postfix



Infix	prefix	postfix
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
A * B - C / D	- * A B / C D	A B * C D / -

Algorithm to Convert Infix to Postfix Using Stack

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘), push it.

…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Algorithm to Convert Infix to Postfix Using Stack

4. If the scanned character is an ‘(‘, push it to the stack.

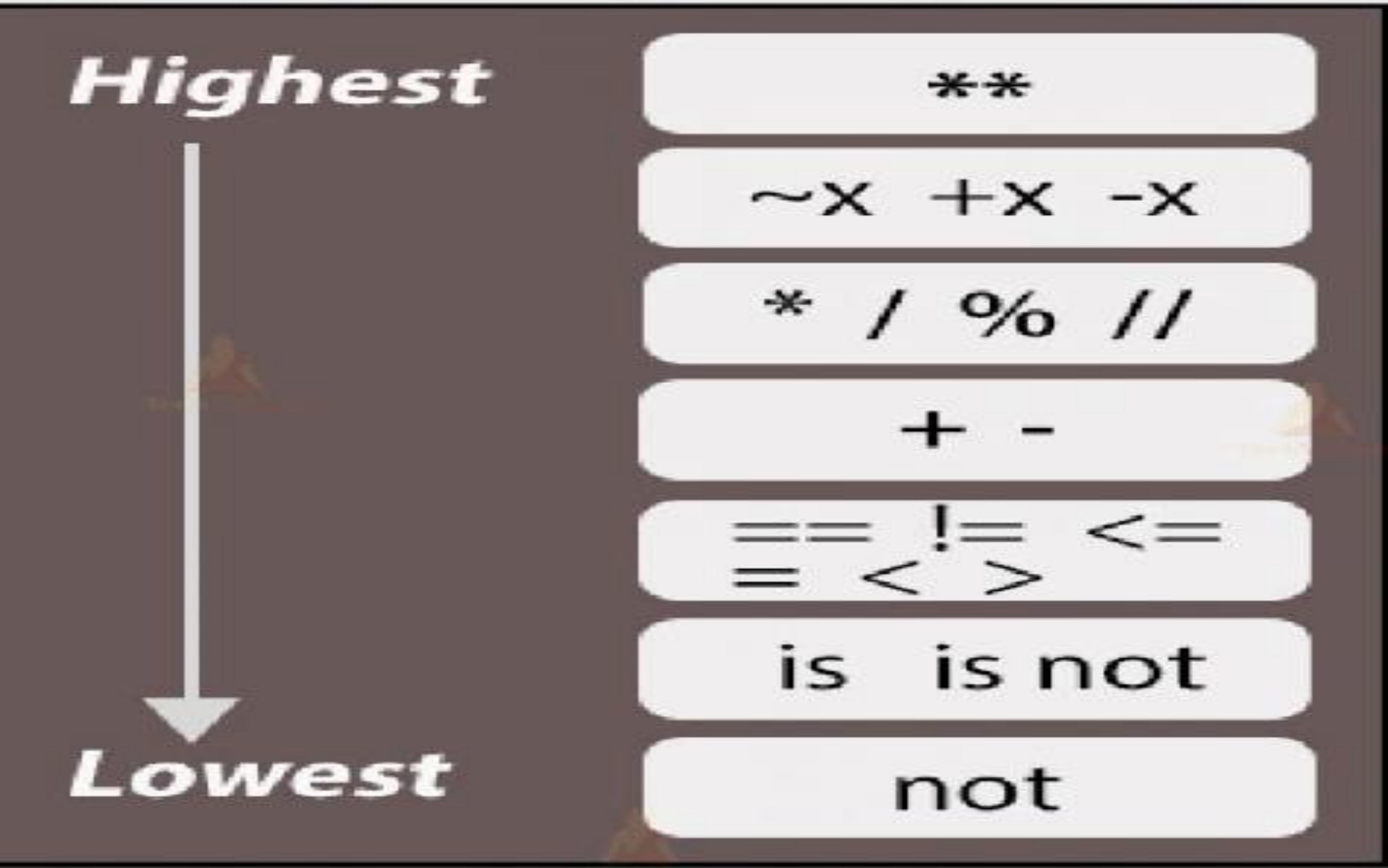
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and add to output from the stack until it is empty

8. Print the output

Operators Precedence



Operators Precedence

Python Operators Precedence Rule – PEMDAS
You might have heard about the BODMAS rule in your school’s mathematics class.
Python also uses a similar type of rule known as PEMDAS.
P – Parentheses
E – Exponentiation
M – Multiplication
D – Division
A – Addition
S – Subtraction

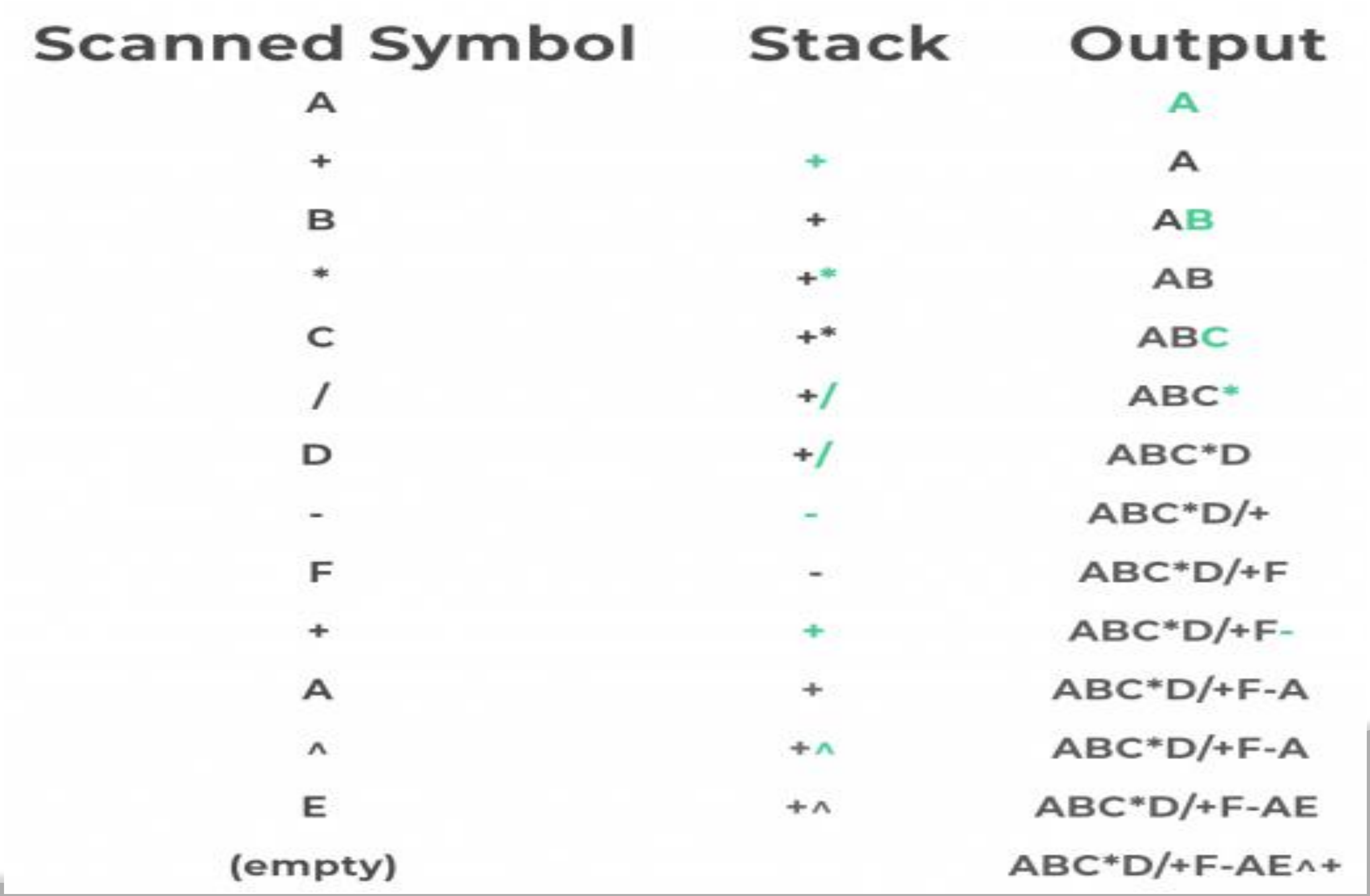
Conversion of Infix to Postfix Using Stack

a + (b*c)

Read character	Stack	Output
a	Empty	a
+	+	a
(	+(	a
b	+(	ab
*	+(*	ab
c	+(*	abc
)	+	abc*
		abc*+

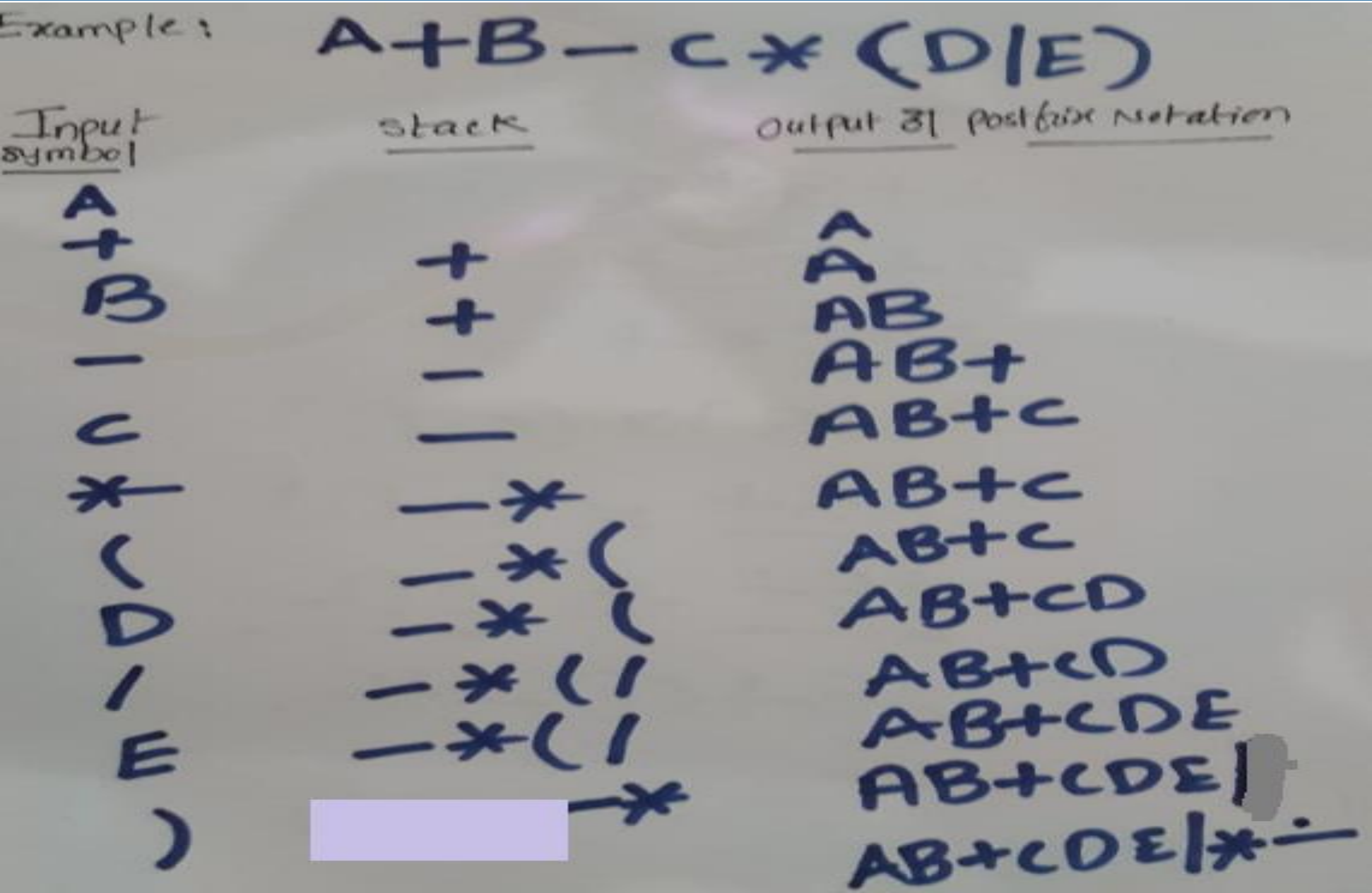
Conversion of Infix to Postfix Using Stack

Expression = A + B * C / D - F + A ^ E



Scanned Symbol	Stack	Output
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
/	+/	ABC*
D	+/	ABC*D
-	-	ABC*D/+
F	-	ABC*D/+F
+	+	ABC*D/+F-
A	+	ABC*D/+F-A
^	+^	ABC*D/+F-A
E	+^	ABC*D/+F-AE
(empty)		ABC*D/+F-AE^+

Conversion of Infix to Postfix Using Stack

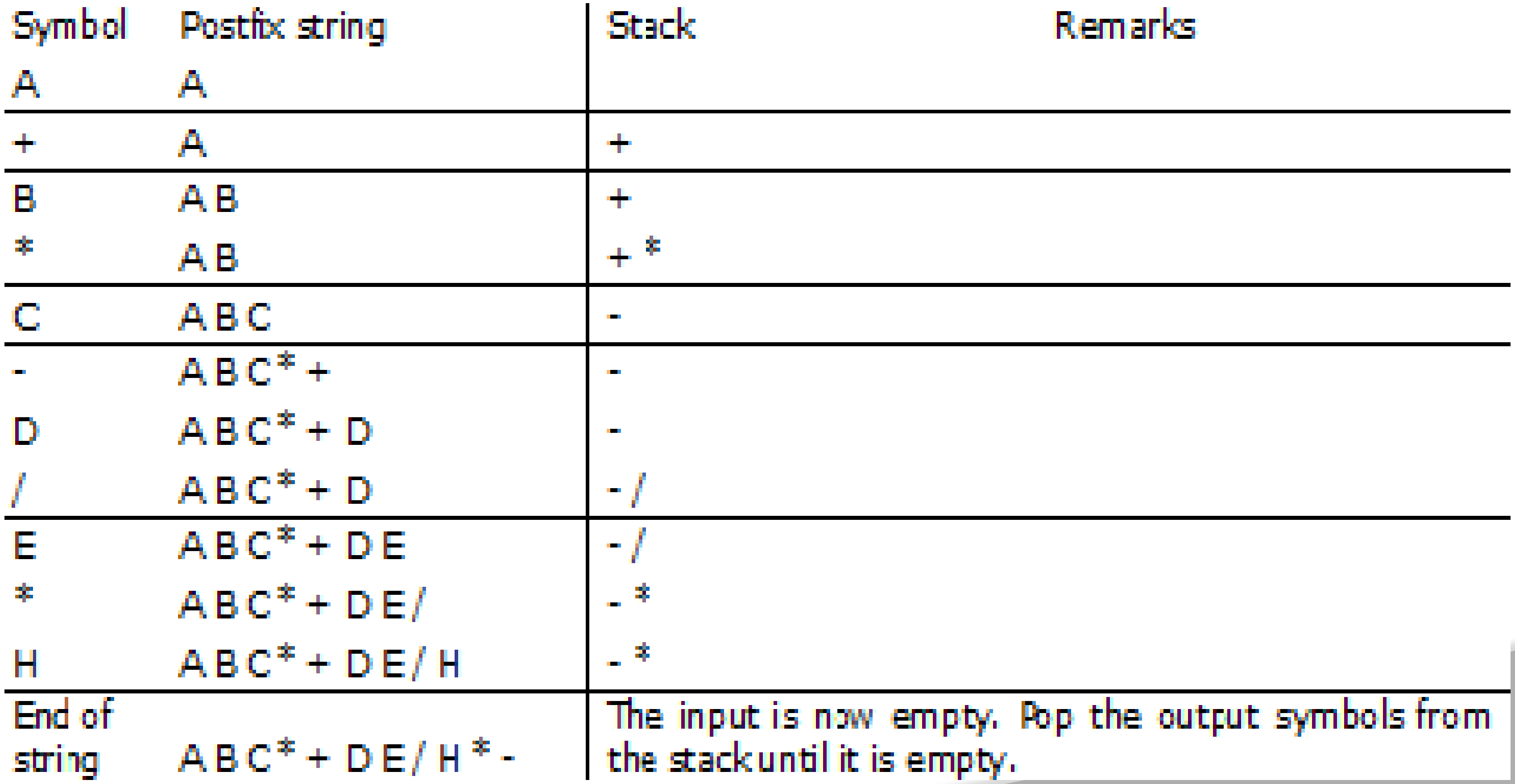


Example: A+B−C*(D/E)

Input symbol	Stack	Output or Postfix Notation
A		A
+	+	A
B	+	AB
−	−	AB+
C	−	AB+C
*	−*	AB+C
(	−*(	AB+C
D	−*(	AB+CD
/	−*(/	AB+CD
E	−*(/	AB+CDE
)	[redacted] −*	AB+CDE/
		AB+CDE/*−

Algorithm to Convert Infix to Postfix Using Stack

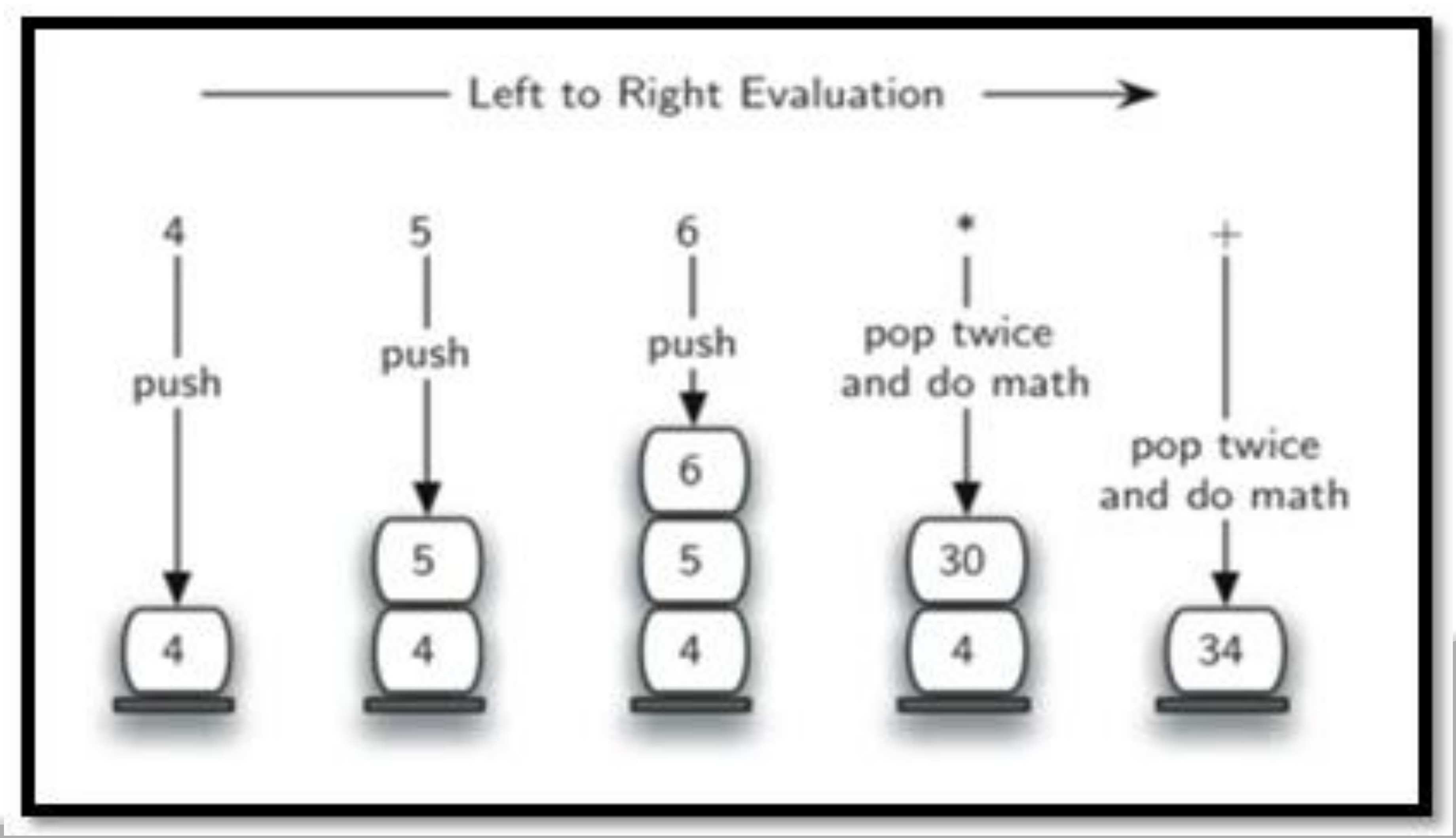
Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.



Symbol	Postfix string	Stack	Remarks
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	-	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

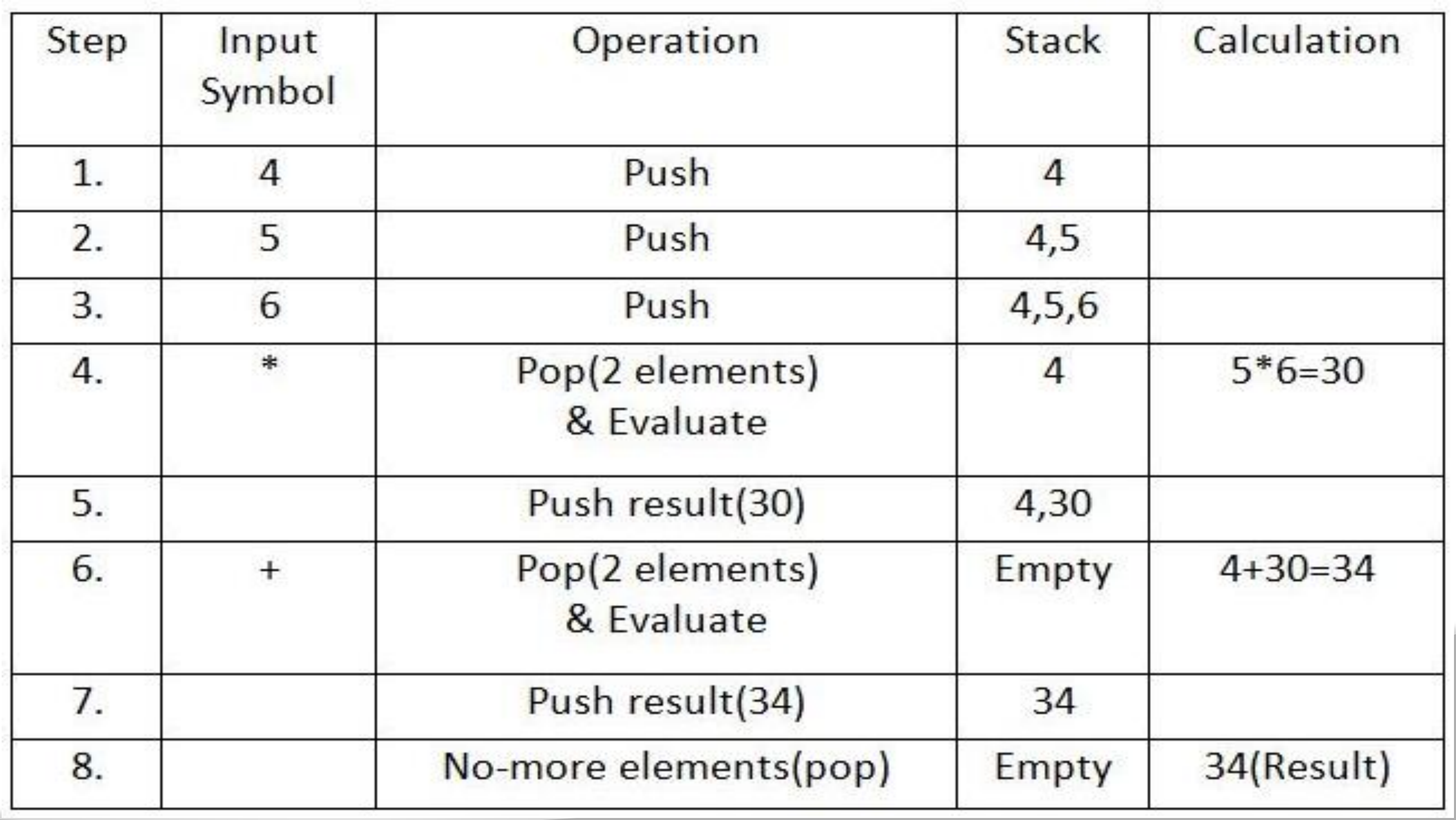
Evaluation of Postfix Expression

Expression: 456*+



Evaluation of Postfix Expression

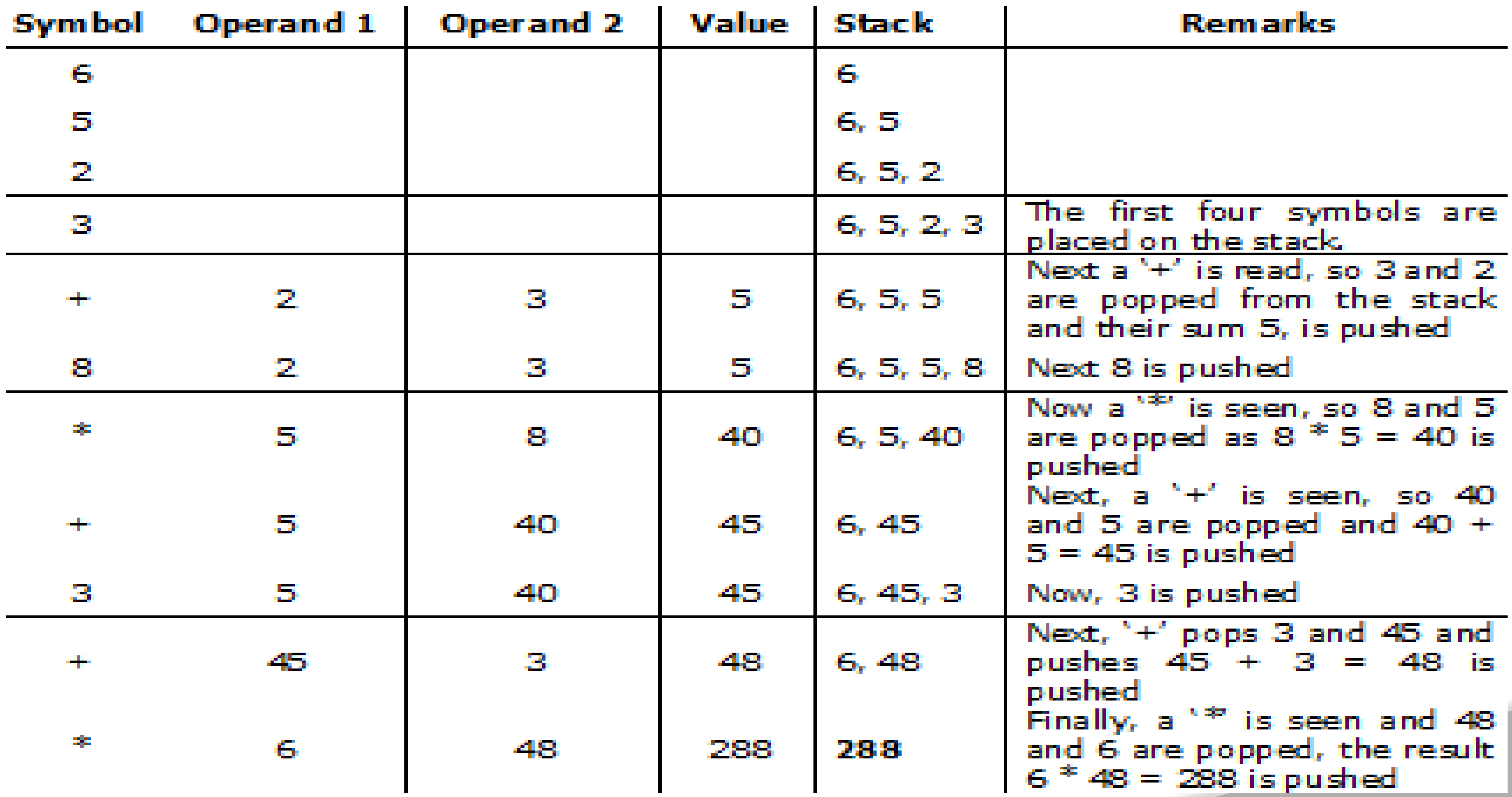
Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Evaluation of Postfix Expression

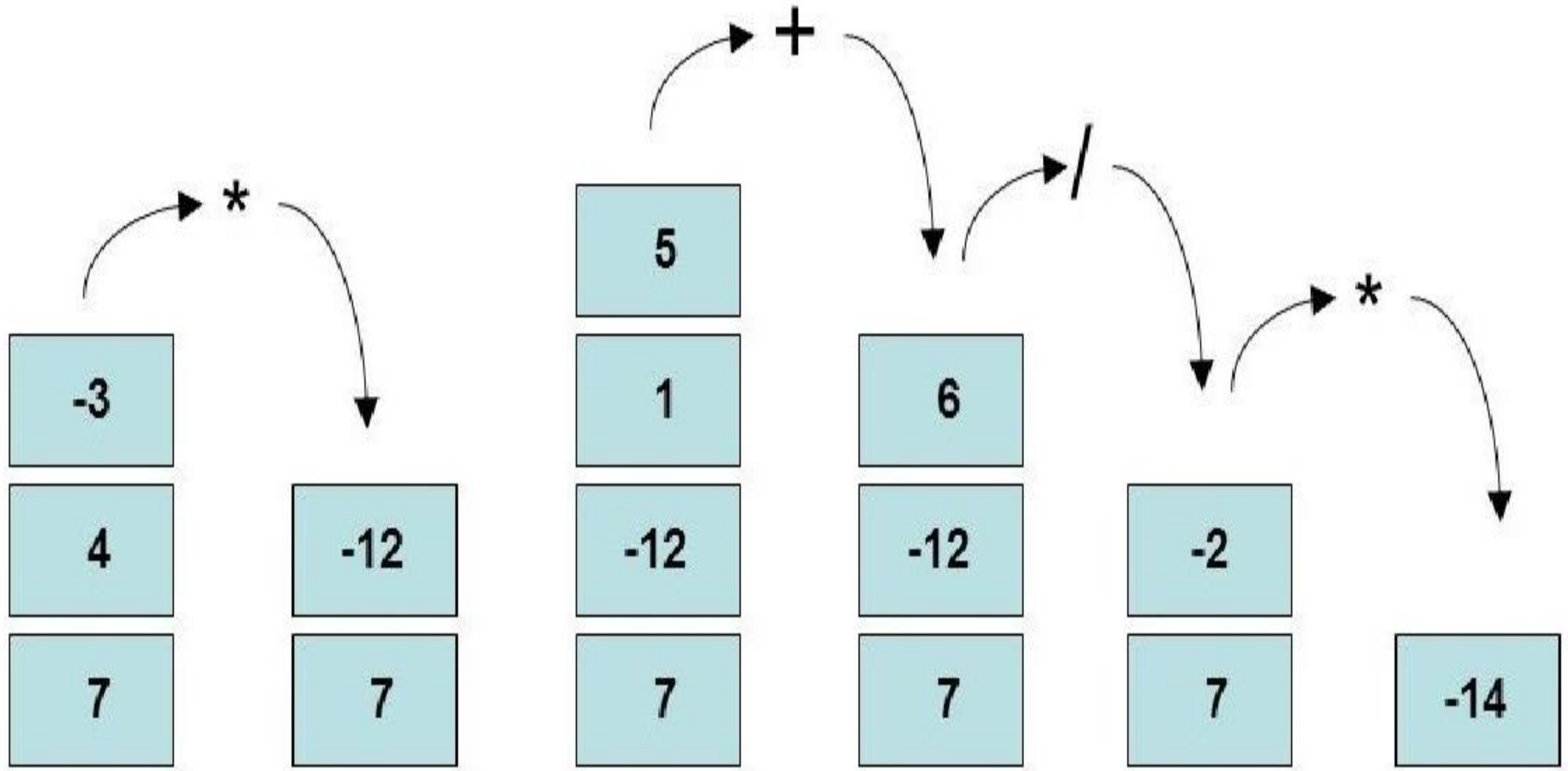
Postfix expression: 6 5 2 3 + 8 * + 3 + *



Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

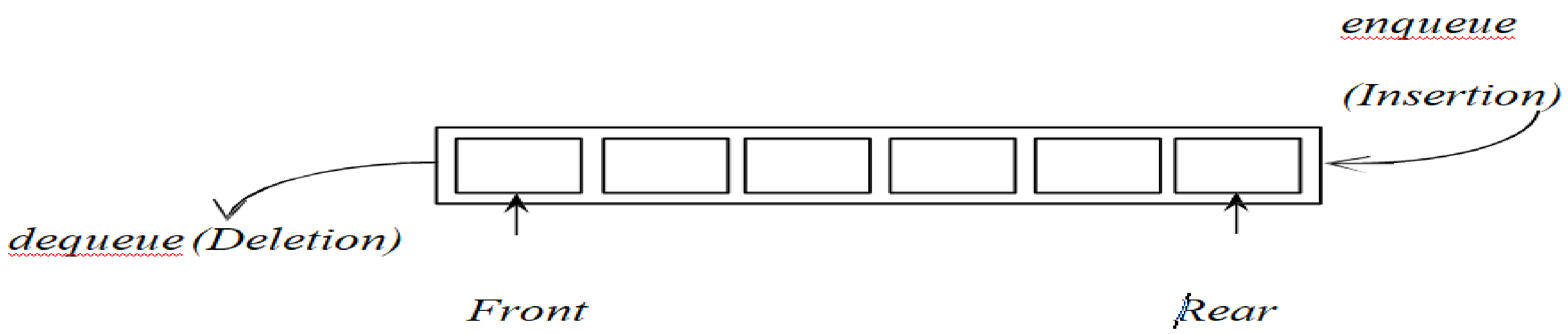
Evaluation of Postfix Expression

- Expression = 7 4 -3 * 1 5 + / *



Queue

"A queue is an ordered list in which all insertions done at one end called REAR and deletions are made at another end called FRONT". Queues are referred to as First In First Out (FIFO)



Example

1.T he people waiting in line at a bank cash counter form a queue.
2. In computer, the jobs waiting in line to use the processor for execution. This queue is called

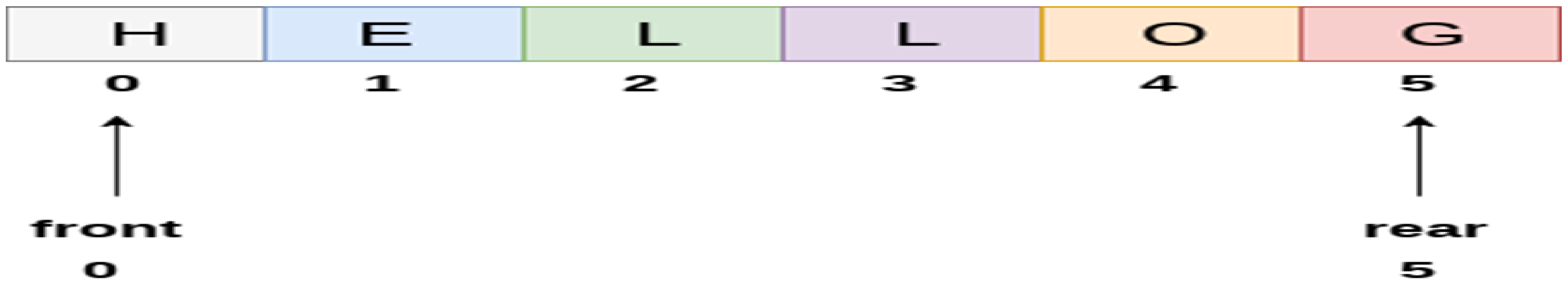
Array Representation of Queue

➢Queue can be easily represented by using linear arrays.

➢ There are two variables i.e. **front** and **rear**, that are implemented in the case of every queue.

➢Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and rear is -1 which represents an empty queue.

Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue Operations using Array

- Various operations of Queue are:

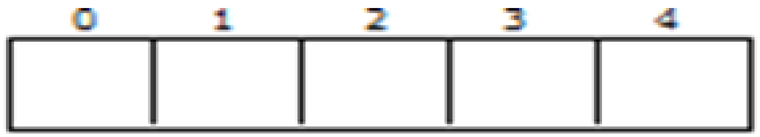
➢ **insertQ():** inserts an element at the end of queue Q.

➢ **deleteQ():** deletes the first element of Q.

➢ **displayQ():** displays the elements in the queue.

Insertion Operation

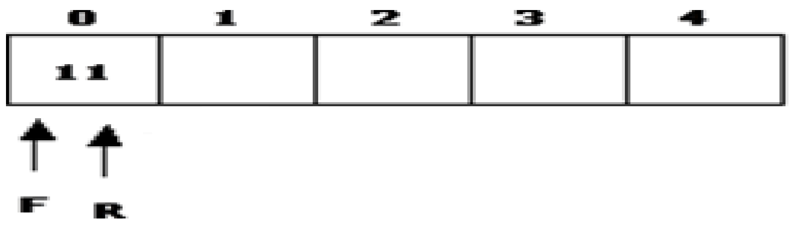
Initially the queue is empty.



Queue Empty

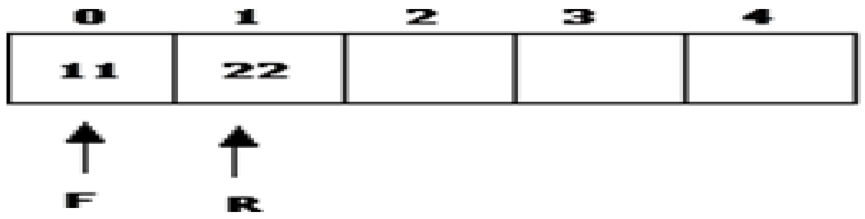
F=R= -1

Now, insert 11 to the queue. Then queue status will be:



F=F+1= -1+1=0
R=R+1= -1+1=0

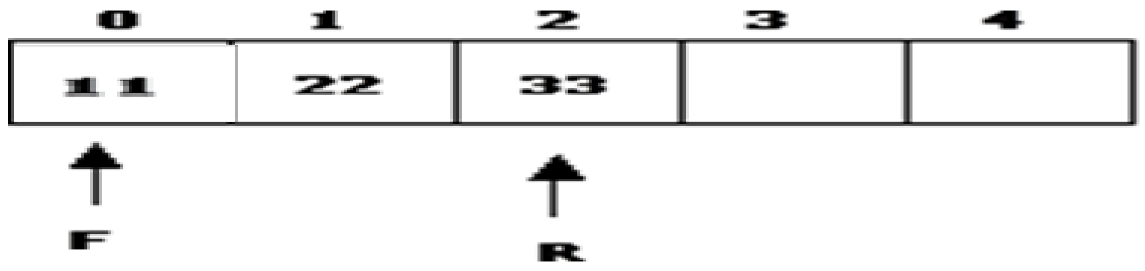
Next, insert 22 to the queue. Then the queue status is:



R=R+1=0+1=1
F=0

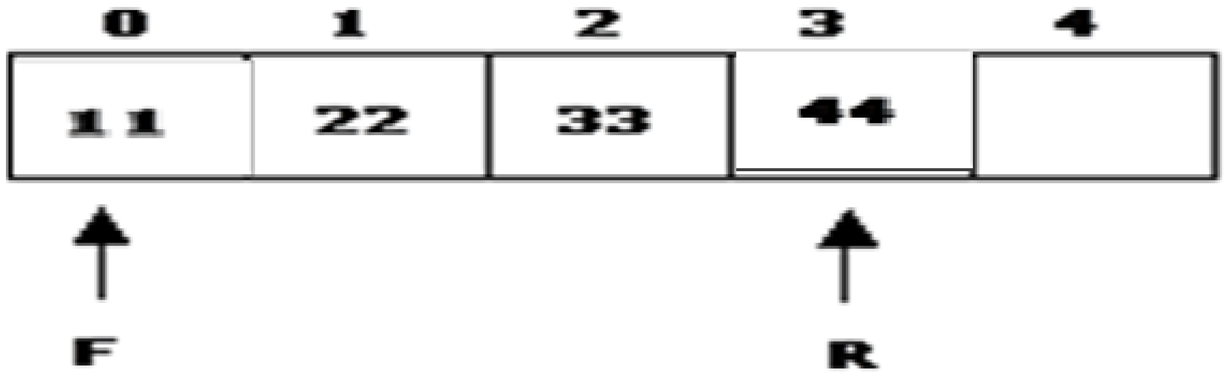
Insertion Operation

Next, insert 33 to the queue. Then the queue status is:



R=R+1=1+1=2
F=0

Next, insert 44 to the queue. Then the queue status is:

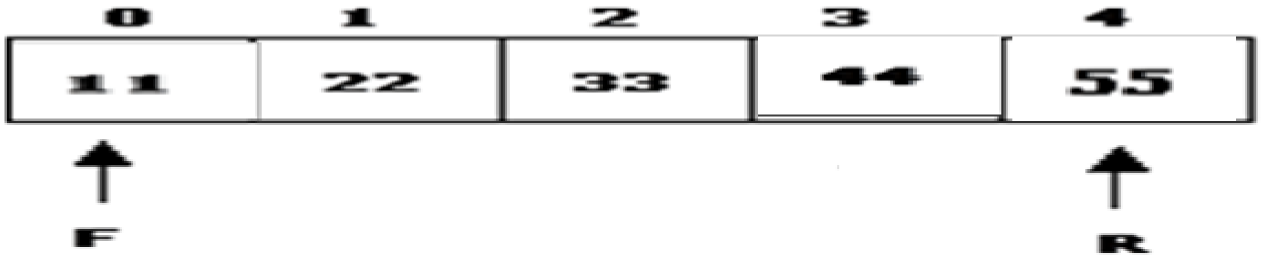


R=R+1=2+1=3
F=0

Insertion Operation

Next, insert 55 to the queue. Then the queue status is:

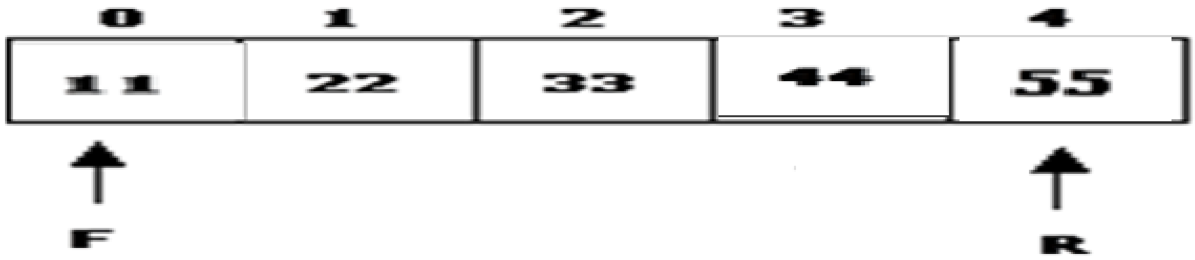
We cannot insert 66 to the queue as it signals queue is full. The queue status is as follows:



R=R+1=3+1=4
F=0

Next insert another element, say 66 to the queue.

We cannot insert 66 to the queue as it signals queue is full. The queue status is as follows:



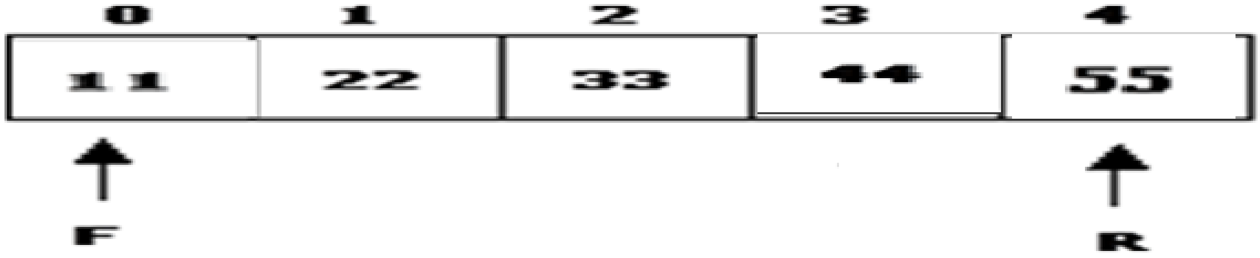
R=R+1=4+1=5
F=0

Algorithm to insert an element in a queue

Algorithm:

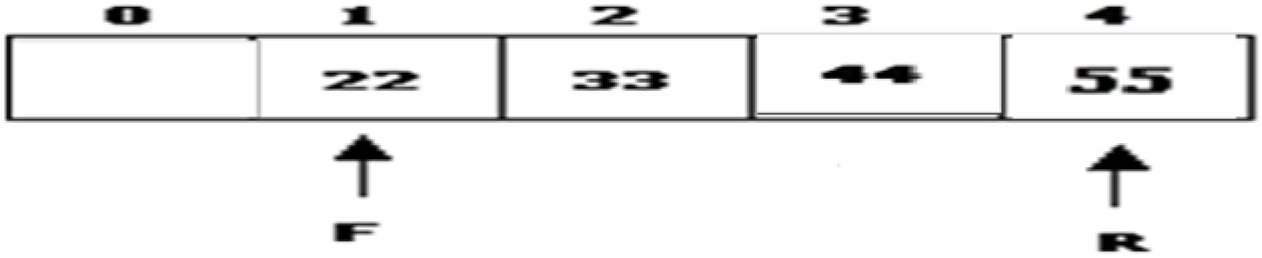
```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step 4
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

Deletion Operation



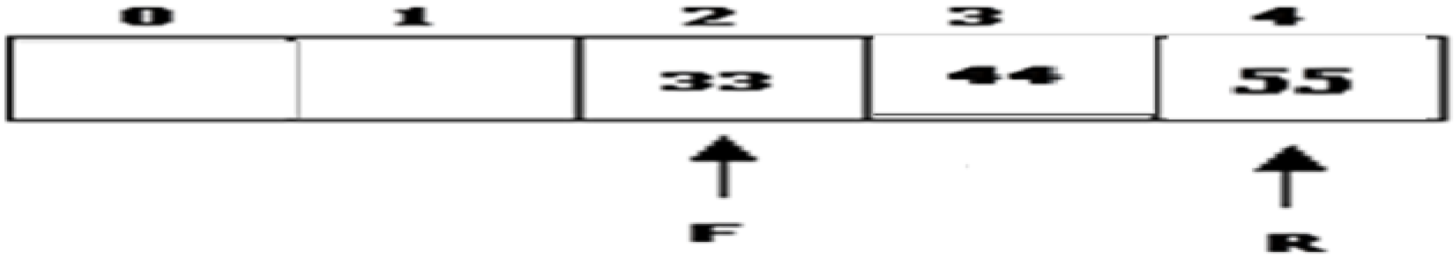
R=R+1=3+1=4
F=0

Now, delete an element



R=4
F=F+1=0+1=0

Now, delete one more element



R=4
F=F+1=1+1=2

Algorithm to delete an element from the queue

Algorithm:

```
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
```

Applications of Queue

- It is used to schedule the jobs to be processed by the CPU.

- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

- Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue

- A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.

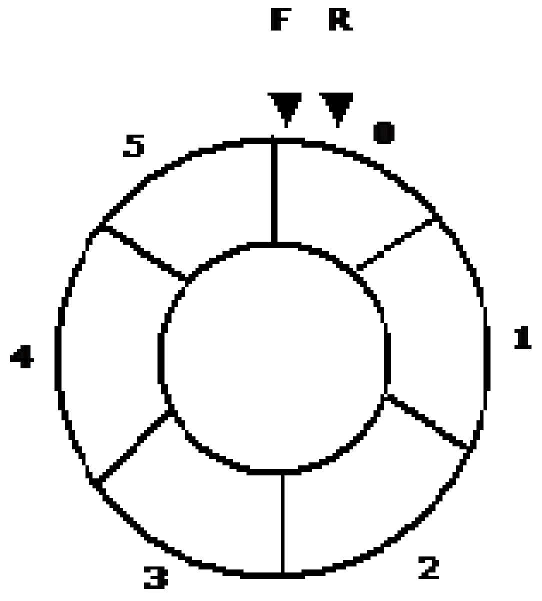
- Suppose if we have a Queue of n elements then after adding the element at the last index i.e. (n-1)th , as queue is starting with 0 index, the next element will be inserted at the very first location of the queue which was not possible in the simple linear queue.

Circular Queue operations

- The Basic Operations of a circular queue are
  - ➢ InsertionCQ: Inserting an element into a circular queue results in Rear = (Rear + 1) % MAX, where MAX is the maximum size of the array.
  - ➢ DeletionCQ : Deleting an element from a circular queue results in Front = (Front + 1) % MAX, where MAX is the maximum size of the array.
  - ➢ TraversCQ: Displaying the elements of a circular Queue.
- Circular Queue Empty: Front=Rear=0

Circular Queue Representation Using Arrays

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.

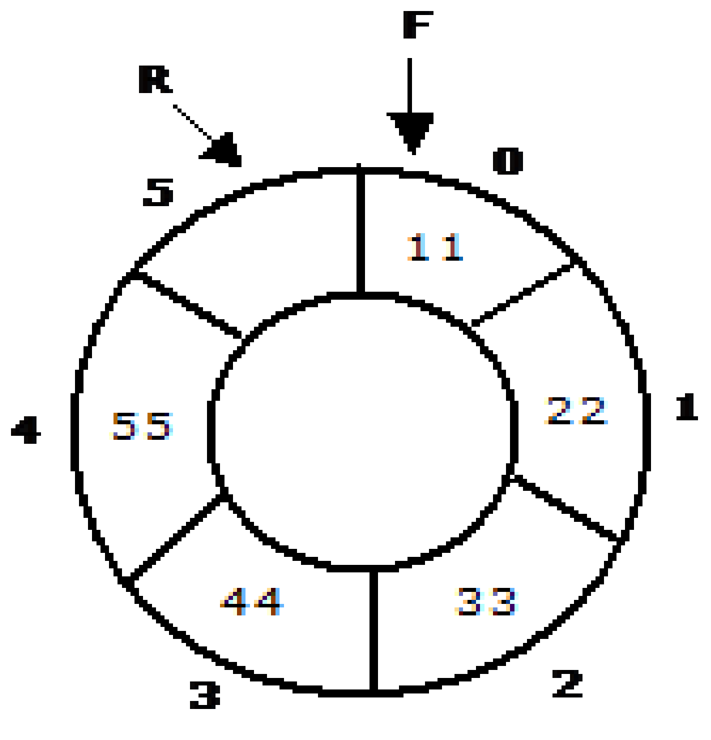


Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Insertion and Deletion operations on a Circular Queue

Insert new elements 11, 22, 33, 44 and 55 into the circular queue. The circular queue status is:

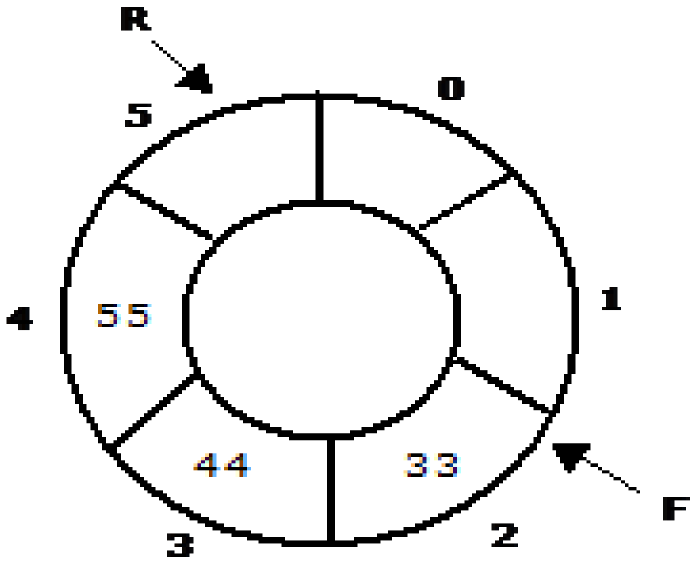


Circular Queue

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

Insertion and Deletion operations on a Circular Queue

- Now, delete two elements 11, 22 from the circular queue. The circular queue status is as follows:

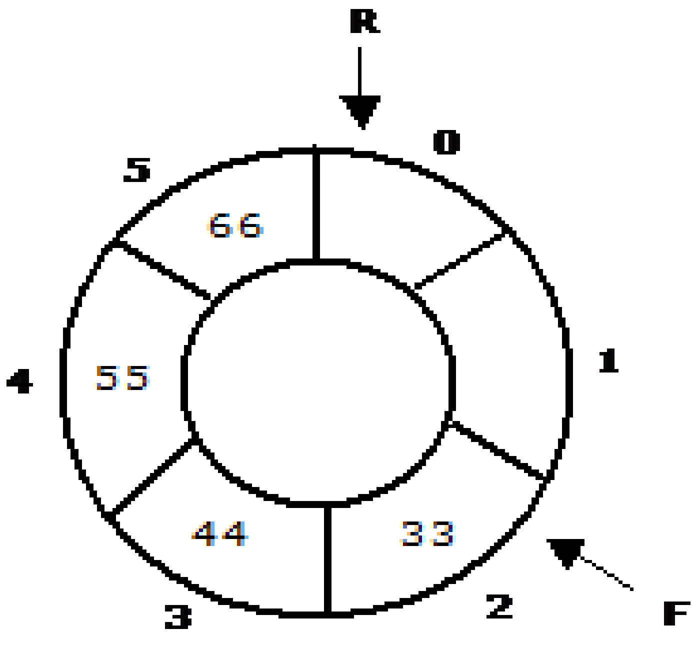


Circular Queue

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

Insertion and Deletion operations on a Circular Queue

Again, insert another element 66 to the circular queue. The status of the circular queue is:

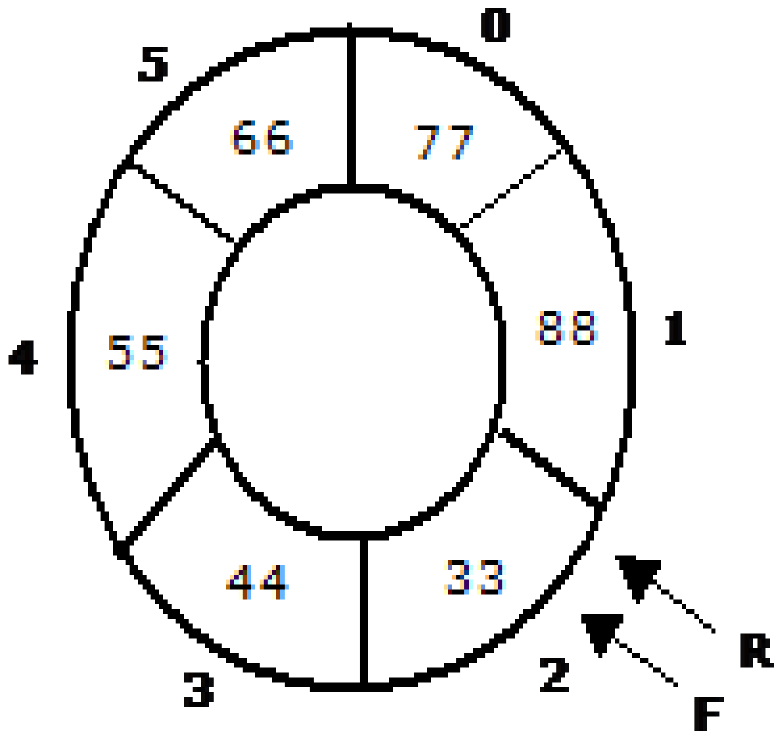


Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

Insertion and Deletion operations on a Circular Queue

Again, insert 77 and 88 to the circular queue. The status of the Circular queue is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

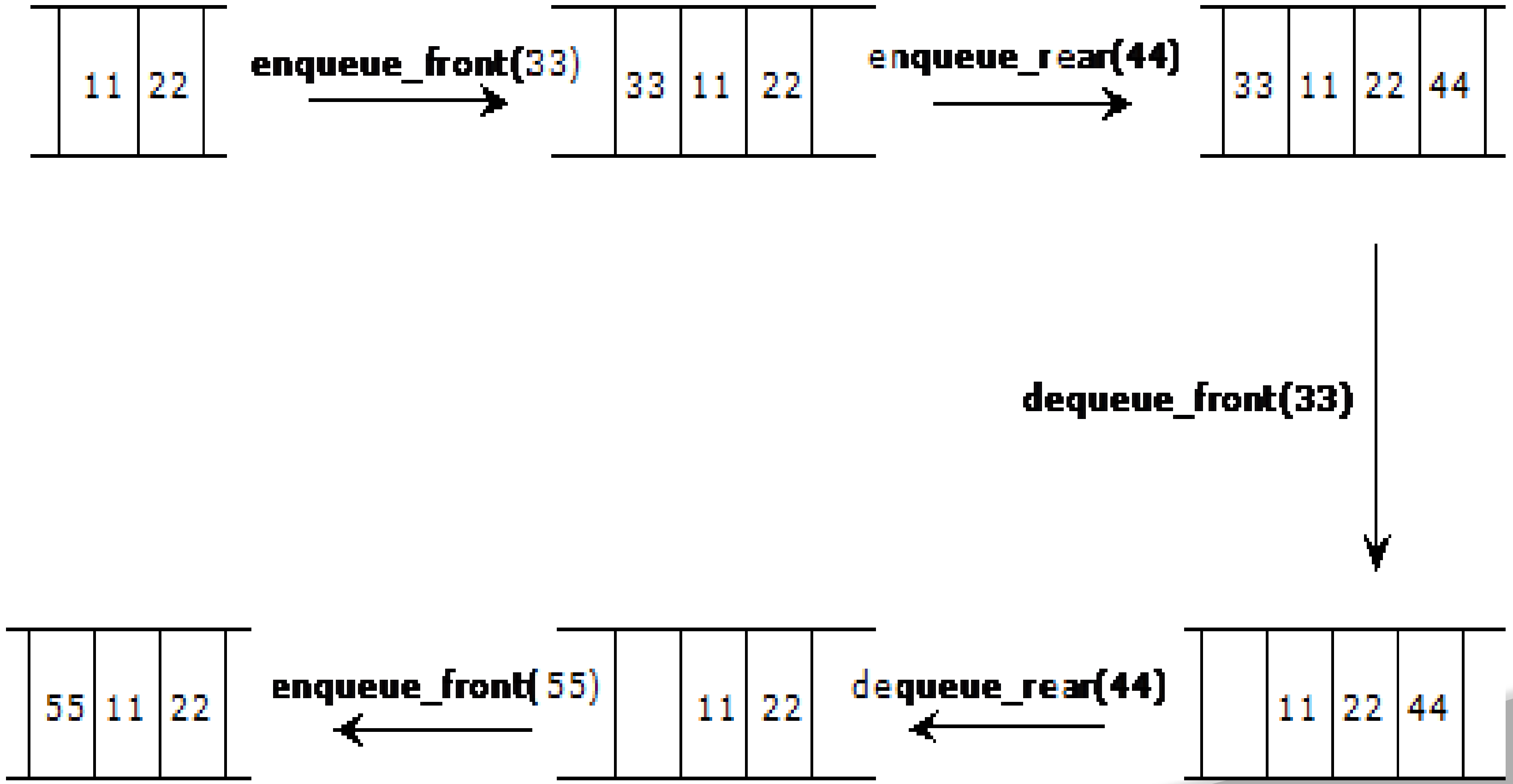
Computer controlled Traffic Signal System uses circular queue.

CPU scheduling and Memory management.

Double Ended Queue (DEQUE)

- It is a special queue like data structure that supports insertion and deletion at both the ends.
- Such an extension of a queue is called a double-ended queue, or deque, which is usually pronounced "deck" to avoid confusion with the dequeue method of the regular queue, which is pronounced like the abbreviation "D.Q."
- It is also often called a head-tail linked list.

DEQUE Representation using arrays



Types of DEQUE

- There are two variations of deque. They are:
  - Input restricted deque (IRD)
  - Output restricted deque (ORD)
- An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.
- An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

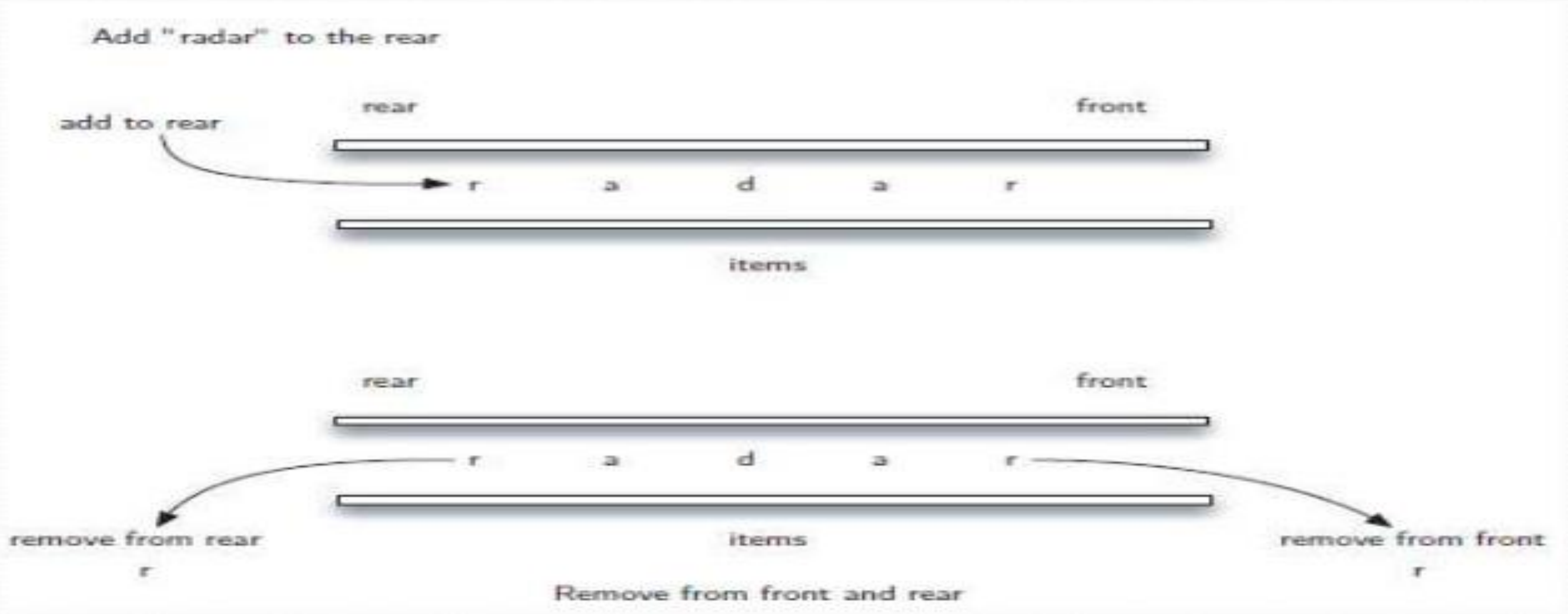
Applications of Deque

- Since Deque supports both stack and queue operations, it can be used as both.
- The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications

- Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque

APPLICATIONS OF DEQUE

Palindrome-checker



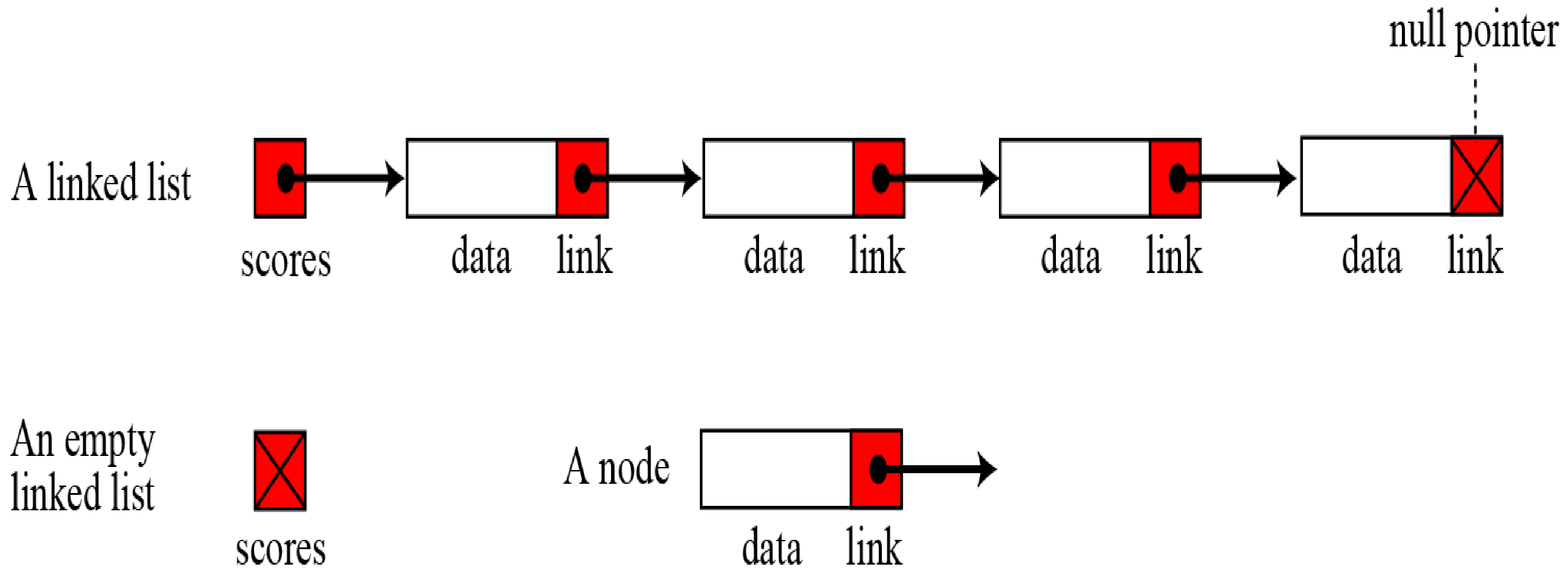
MODULE -III

Linked lists: Introduction, singly linked list, representation of a linked list in memory, operations on a single linked list; Applications of linked lists: Polynomial representation and sparse matrix manipulation.

Types of linked lists: Circular linked lists, doubly linked lists; Linked list representation and operations of Stack and Queue.

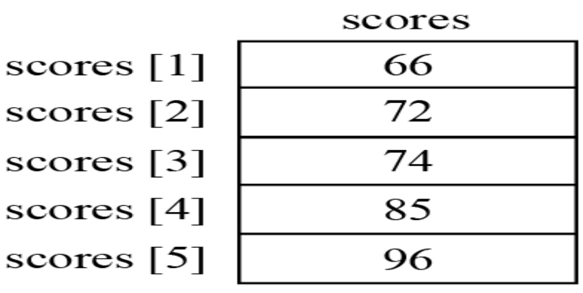
Introduction to Linked List

A linked list is a collection of data in which each element contains the location of the next element—that is, each element contains two parts: data and link.

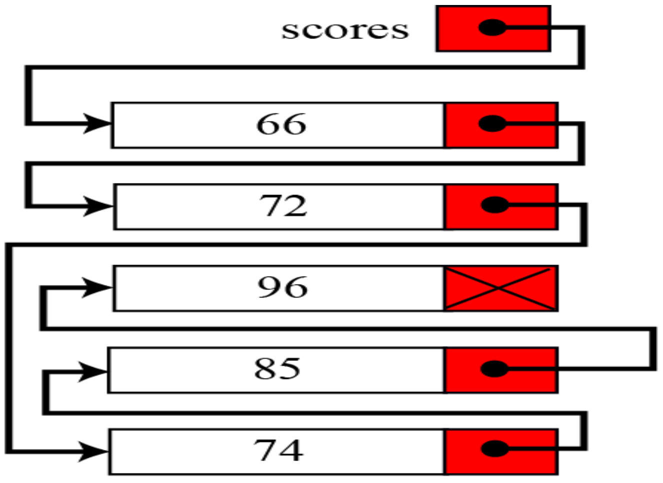


Arrays versus Linked Lists

- Both an array and a linked list are representations of a list of items in memory. The only difference is the way in which the items are linked together. The Figure below compares the two representations for a list of five integers.



a. Array representation



b. Linked list representation

Linked List: A Dynamic Data Structure

- A data structure that can shrink or grow during program execution.
- The size of a dynamic data structure is not necessarily known at compilation time, in most programming languages.
- Efficient insertion and deletion of elements.
- The data in a dynamic data structure can be stored in non-contiguous (arbitrary) locations.
- Linked list is an example of a dynamic data structure.

Advantages of linked list

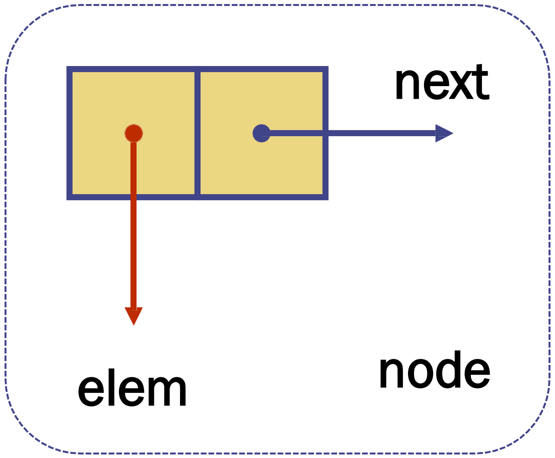
- Unused locations in array is often a wastage of space
- Linked lists offer an efficient use of memory
  - Create nodes when they are required
  - Delete nodes when they are not required anymore
  - We don‘t have to know in advance how long the list should be

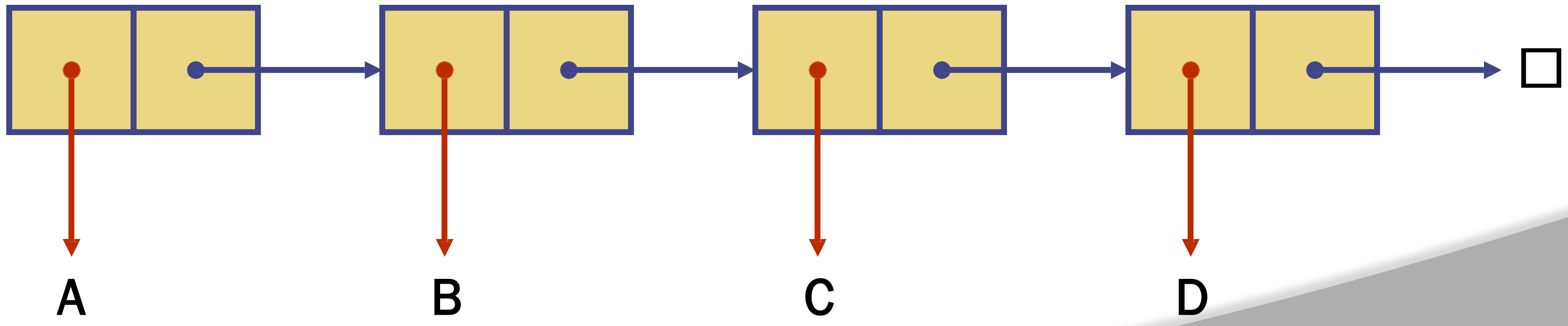
Types of linked lists

- There are four types of Linked lists:
  - Single linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - Circular single linked list
    - Pointer in the last node points back to the first node
  - Doubly linked list
    - Two —start pointers– first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - Circular double linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

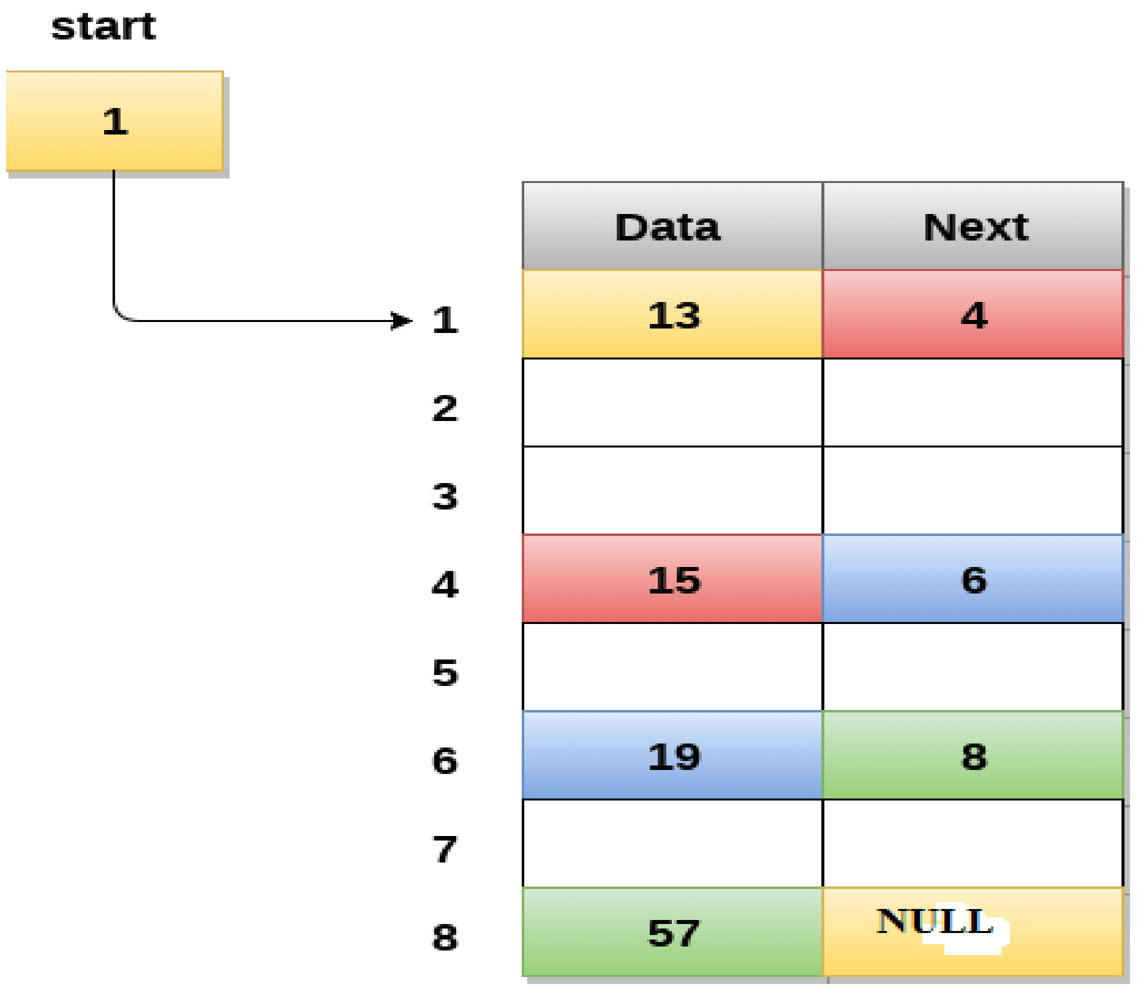
Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - **Element**
  - **link to the next node**





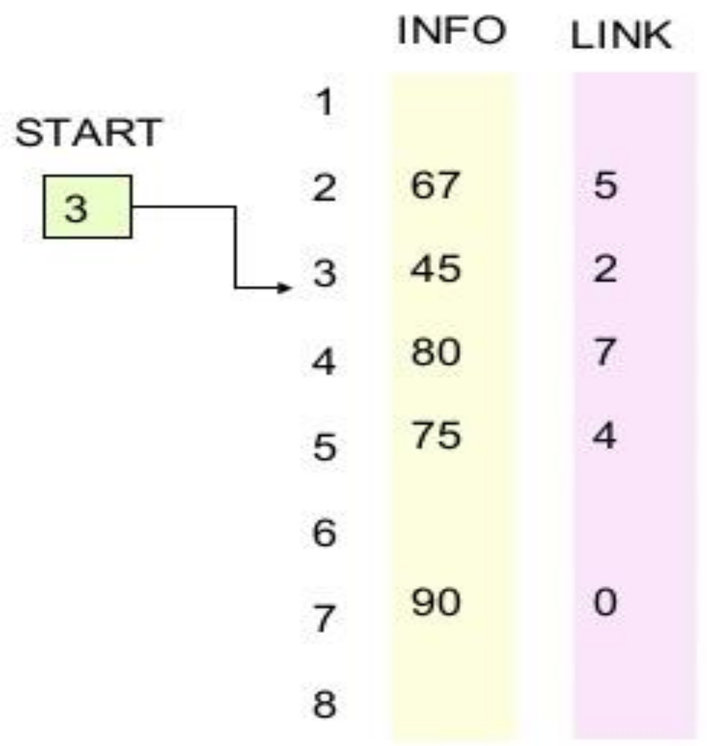
Representation of a linked list in memory



	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	NULL

Memory Representation of a linked list

Representation of Linked lists in memory



	INFO	LINK
1		
2	67	5
3	45	2
4	80	7
5	75	4
6		
7	90	0
8		

START=3, INFO[3]=45

LINK[3]=2, INFO[2]=67

LINK[2]=5, INFO[5]=75

LINK[5]=4, INFO[4]=80

LINK[4]=7, INFO[7]=90

LINK[7]=0, NULL value, So the list has ended

Operations on Linked Lists

- The basic operations of a single linked list are
  - **Creation**
  - **Insertion**
  - **Deletion**
  - **Traversing**

Applications of linked list

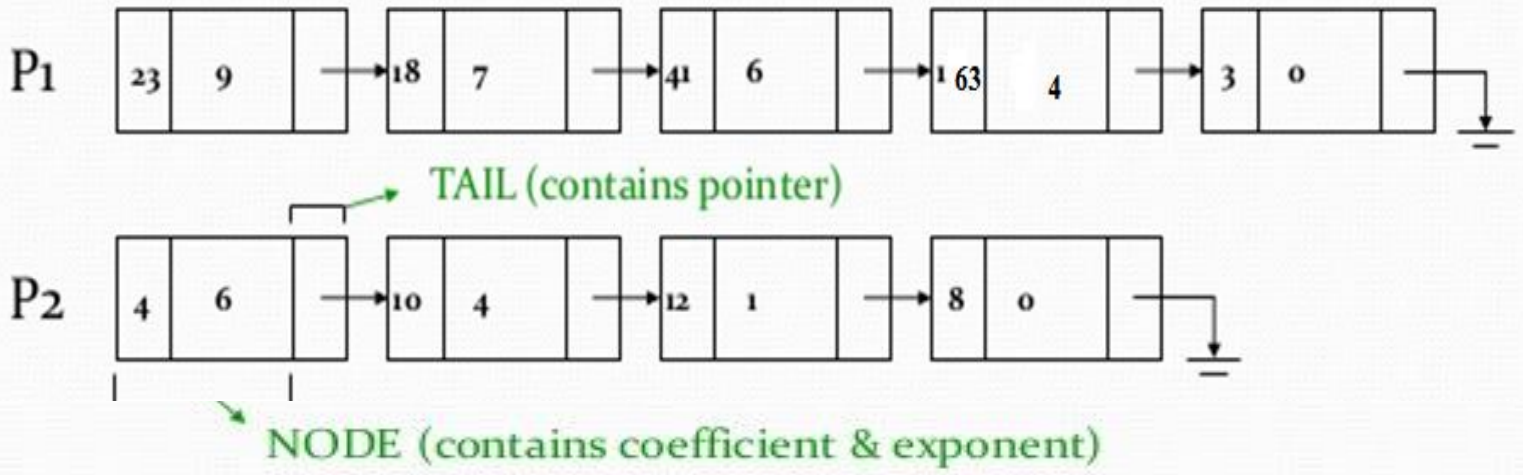
- Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

  $P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

- Represent very large numbers and operations of the large number such as addition, multiplication and division.
- Linked lists are to implement stack, queue, trees and graphs.
- Implement the symbol table in compiler construction.

Polynomial Representation

- Linked list Implementation:
  - $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
  - $p2(x) = 4x^6 + 10x^4 + 12x + 8$



Sparse Matrices

Matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

sparse … many elements are zero
dense … few elements are zero

```
Example:
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
```

Example Of Sparse Matrices

- Diagonal
- Tridiagonal
- Lower triangular

Representation Of Sparse Matrices

Single linear list in row-major order:

- scan the nonzero elements of the sparse matrix in row-major order
- each nonzero element is represented by a triple (row, column, value)
- the list of triples may be an array list or a linked list (chain)

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 =

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

Array Linear List Representation

Example:

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

list =

row	**1**	**1**	**2**	**2**	**4**	**4**
column	**3**	**5**	**3**	**4**	**2**	**3**
value	**3**	**4**	**5**	**7**	**2**	**6**

Chain Representation

Node structure:

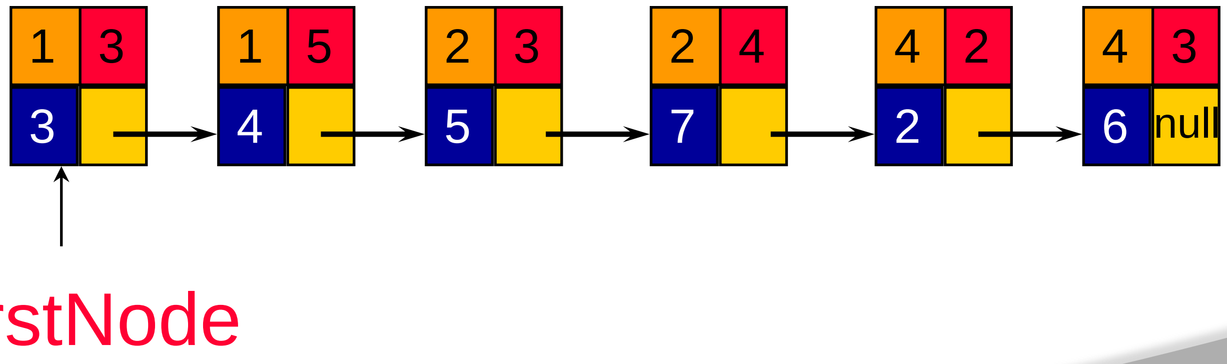
row	col
value	next

Single Chain

Example:

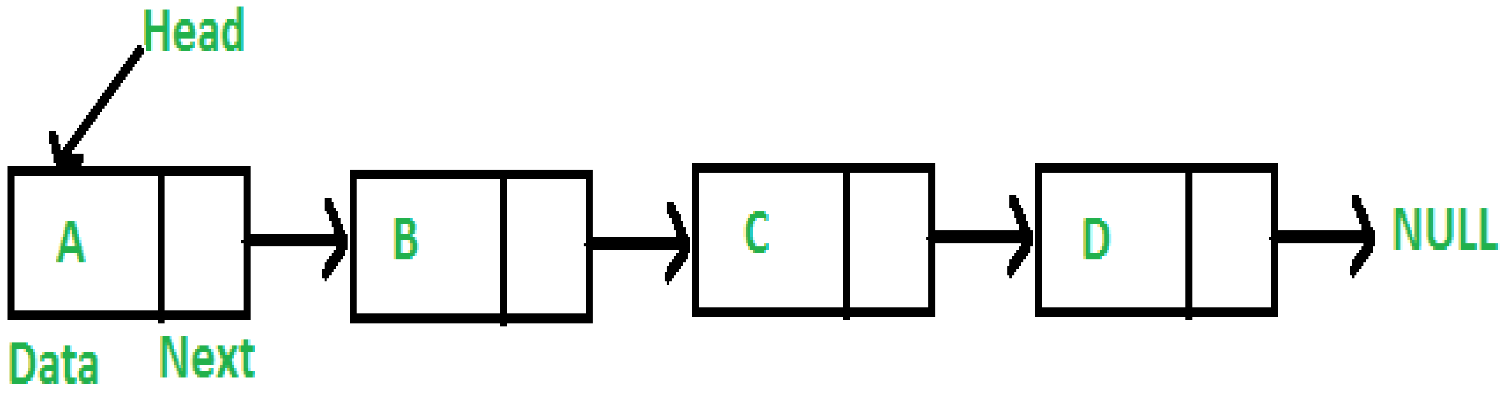
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

		row	1	1	2	2	4	4
list	=	column	3	5	3	4	2	3
		value	3	4	5	7	2	6



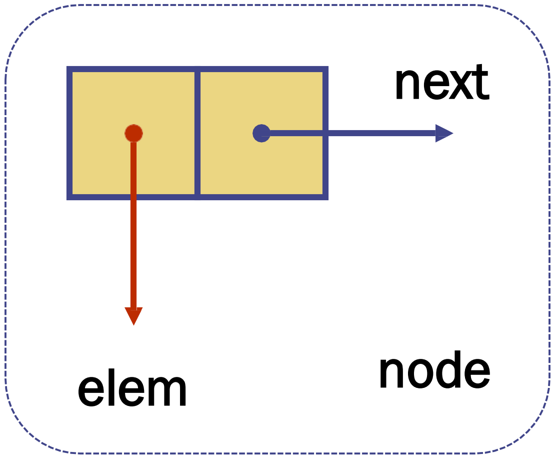
Single Linked List

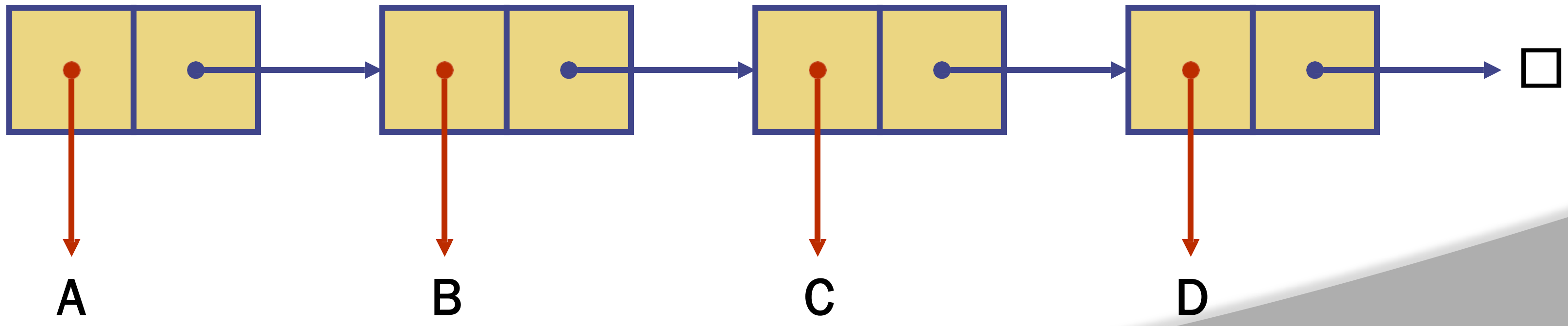
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



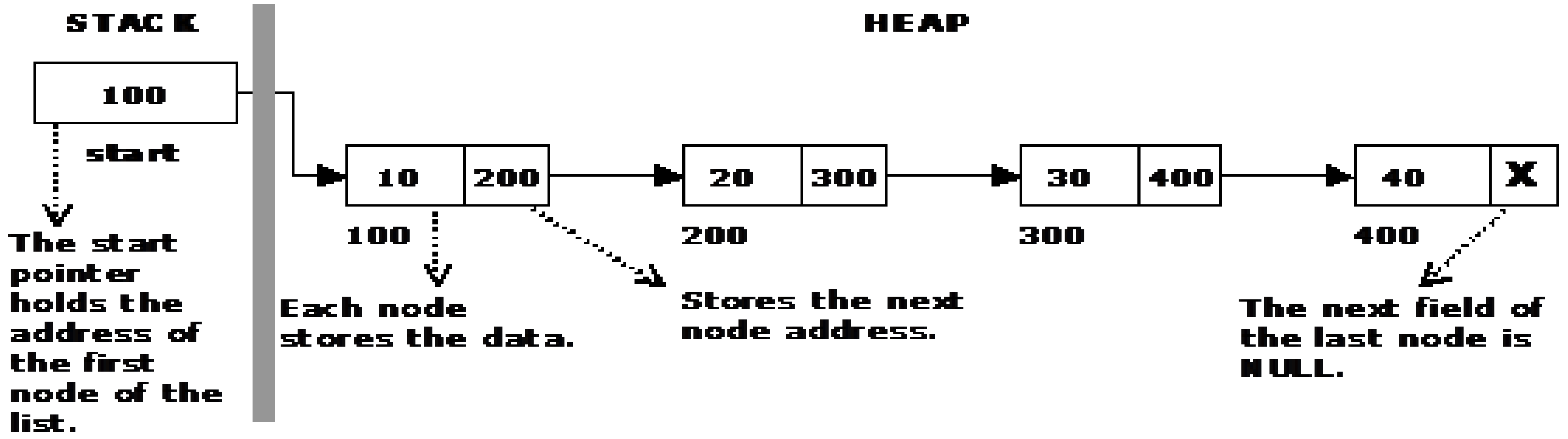
Single Linked List

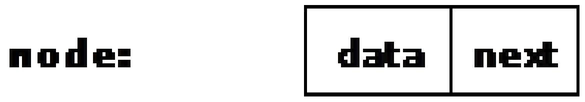
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - **Element**
  - **link to the next node**

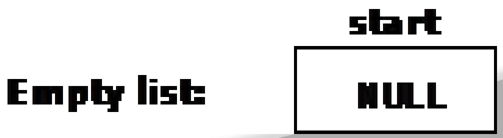




Single Linked List


STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
10
200
20
300
30
400
40
X
100
200
300
400
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.





Operations on Single Linked Lists

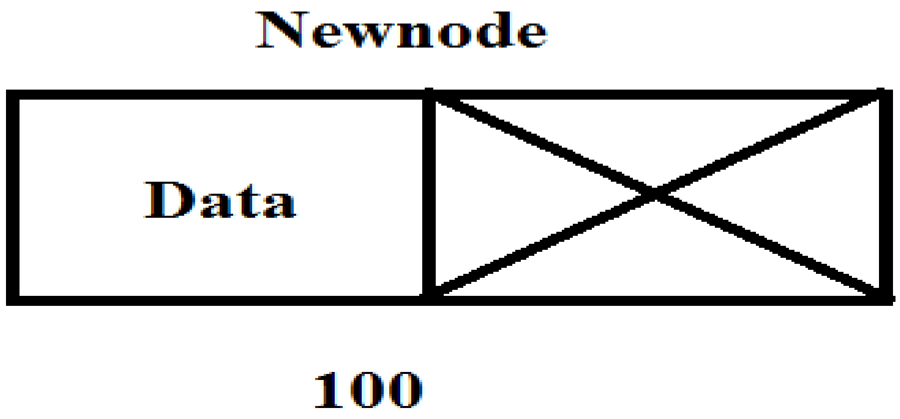
- The basic operations of a single linked list are
  - **Creation**
  - **Insertion**
  - **Deletion**
  - **Traversing**

Creation of a Single Linked List

- A linked list allocates space for each element separately in its own block of memory called a "node".
- Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node.
- Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free().
- The front of the list is a pointer to the —start node

Creating a node for Single Linked List

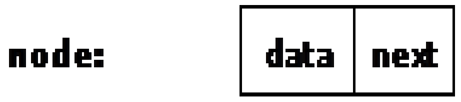
- Sufficient memory has to be allocated for creating a node.
- The information is stored in the memory, allocated by using the malloc() function.,after allocating memory for the structure of type node,
- the information for the item (i.e., data) has to be read from the user,
- set next field to NULL and finally returns the address of the node.

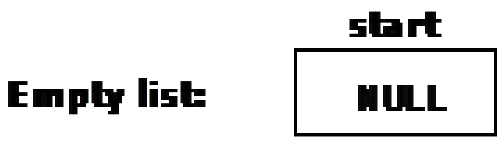


Creation of a Single Linked List

LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

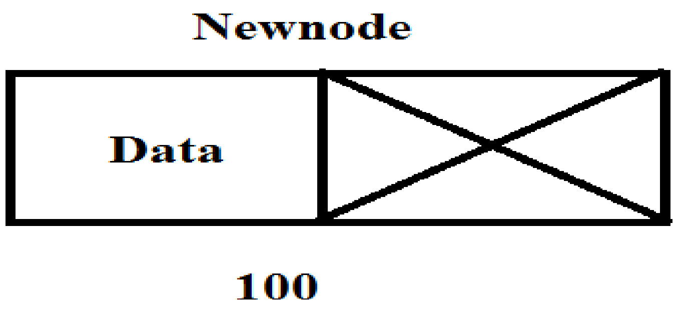
```
# Node class
class Node:

  # Function to initialize the node object
          def __init__(self, data):
                    self.data = data  # Assign data
                    self.next = None  # Initialize   # next as null

  class SLL:
     def __init__(self):
        self.start=None
```

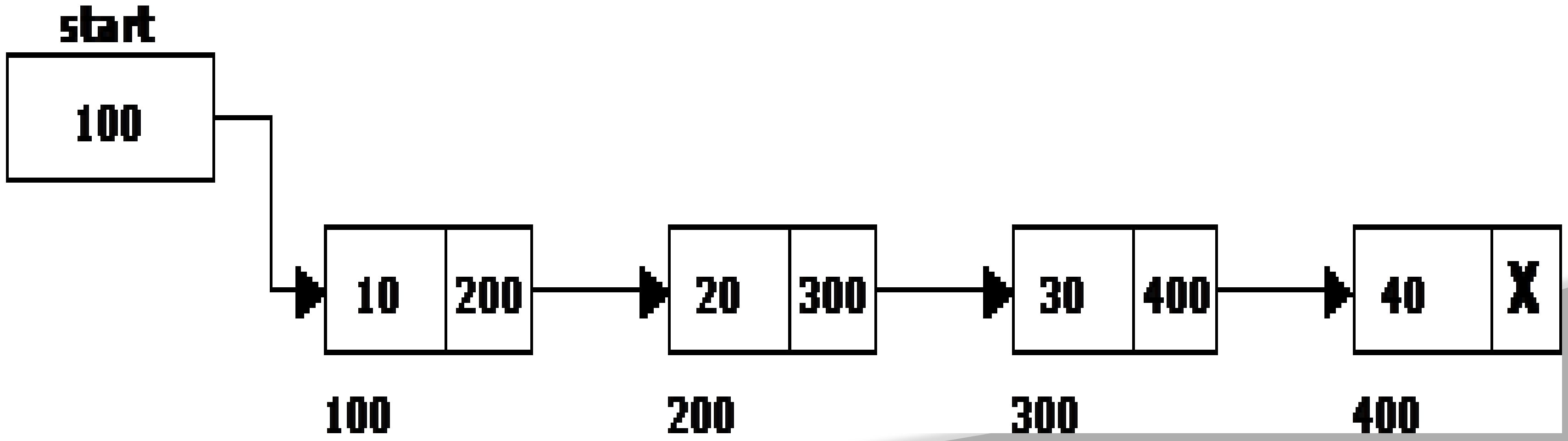




Creating a single linked list with N nodes

```
1.Create a newnode
2.Set newnode.data=data
3.Set newnode.next=NULL
4.Set temp=start
5.Repeat step 6 whiletemp.next!=NULL
6.          set temp=temp.next
   [end of loop]
7.Set temp.next=newnode
8.Exit
```



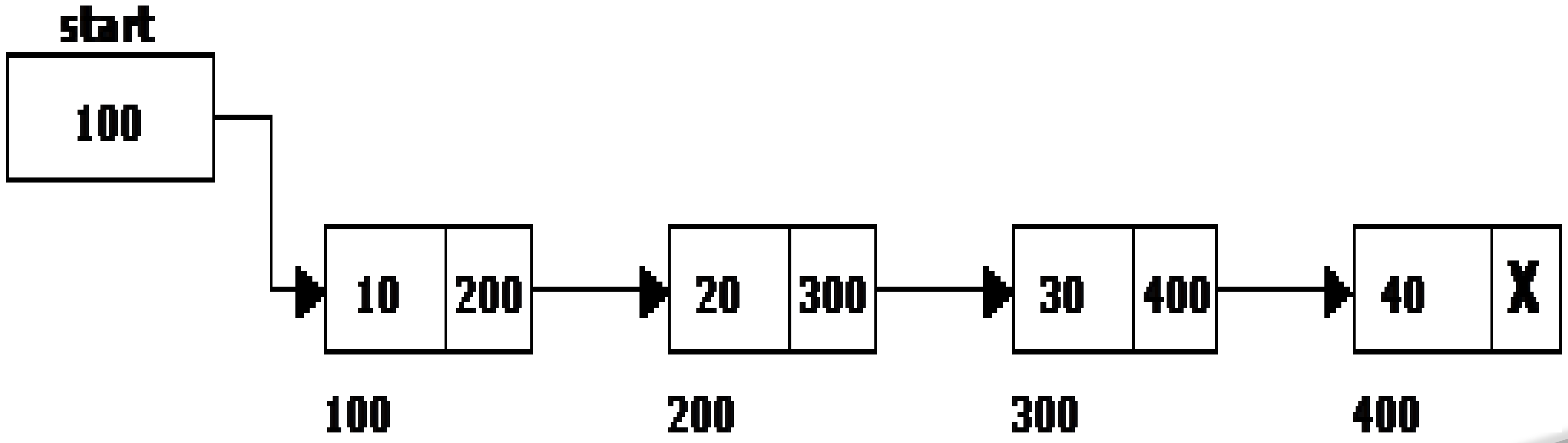


Creating a single linked list with N nodes

```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
class SLL:
    def __init__(self):
        self.start=None
    def createlist(self):
        n=int(input("enter no of nodes"))
        for i in range(n):
            data=int(input("enter value"))
            newnode=Node(data)
            if(self.start==None):
                self.start=newnode
            else:
                temp=self.start
                while temp.next!=None:
                    temp=temp.next
                temp.next=newnode
```

Traversing a Single Linked List

```
1.Set temp=start
2.Repeat step 3 and 4 While temp!=NULL
3.   Apply process to temp.data
4.   Set temp=temp.next
     [Enf of loop]
5.Exit
```



Print Each node of a Single Linked List

```
1.Set temp=start
2.Write temp.data
3.Repeat step 4 and 5 While temp.next!=NULL
4.   Write  temp.data
5.   Set temp=temp.next
     [Enf of loop]
5.Exit
```

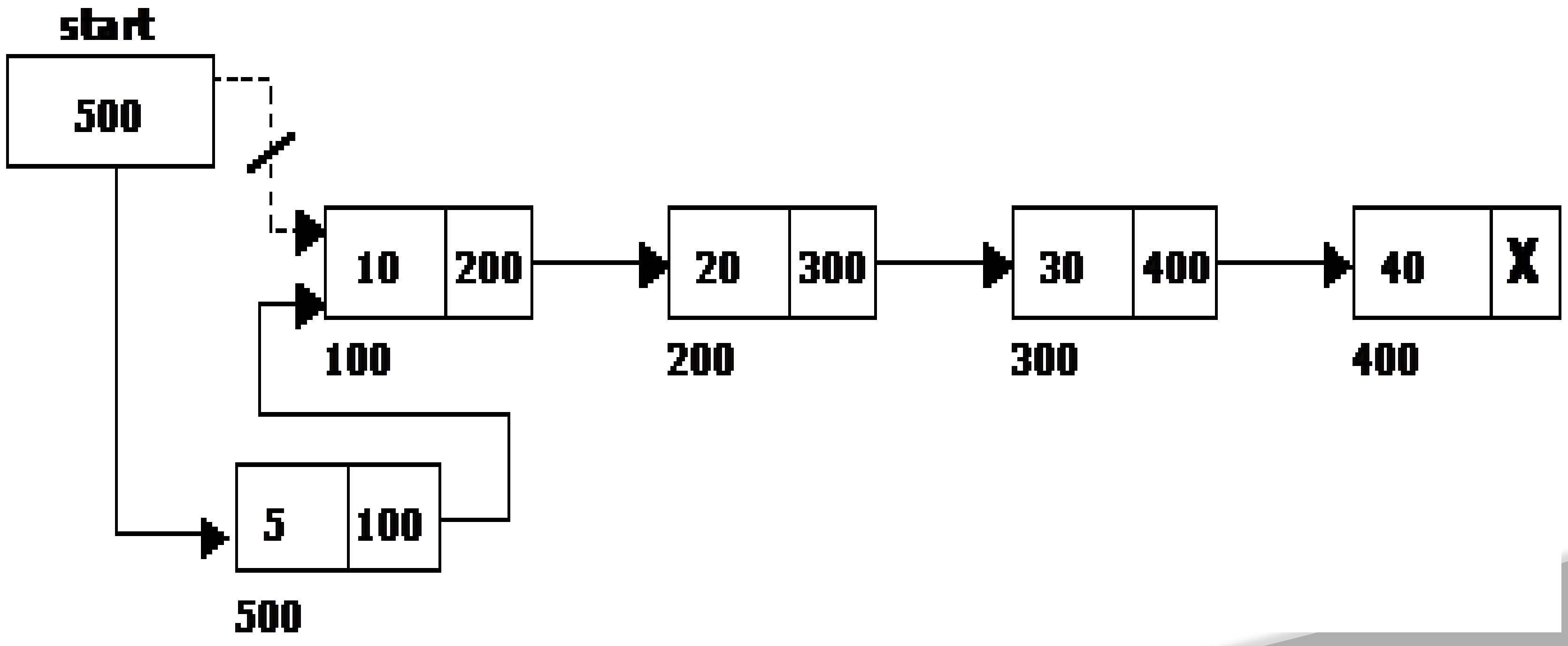
```
#function to Display each node of a linked list
def display(self):
        print("element in single linked list are:")
        if self.start==None:
            print("Empty")
        else:
            temp=self.start
            print(temp.data)
            while temp.next!=None:
                temp=temp.next
                print(temp.data)
```

Inserting a node

- Inserting a node into a single linked list can be done at
  - Insertion at the beginning of the list.
  - Insertion at the end of the list.
  - Insertion in the middle of the list.

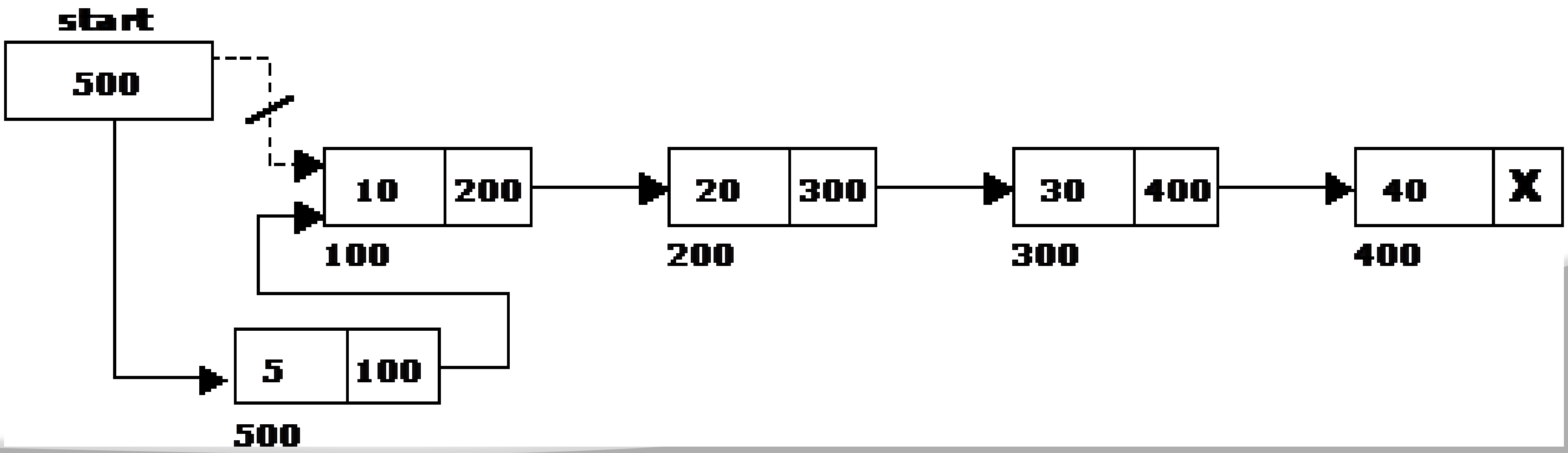
Inserting a node at the beginning

1.Create a nwenode
2.Set newnode.data=data
3.Set newnode.next=start
4.Set start=newnode
5.Exit



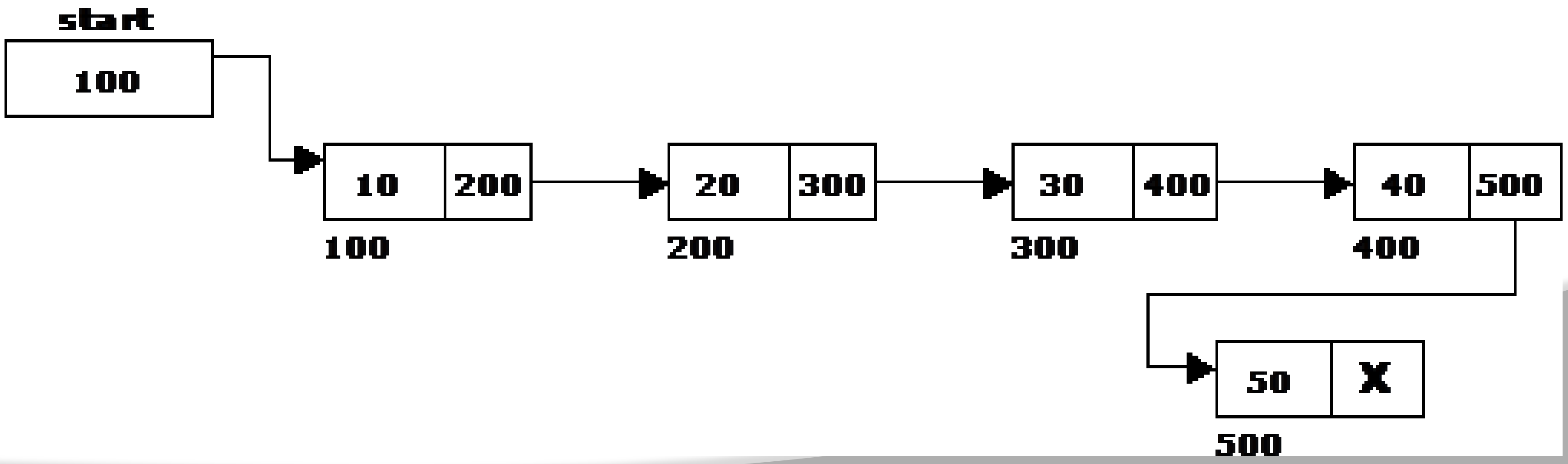
Inserting a node at the beginning

```
def insertbegin(self):
        data=int(input("enter value"))
        newnode=Node(data)
        if self.start==None:
            self.start=newnode
        else:
            temp=self.start
            newnode.next=temp
            self.start=newnode
```



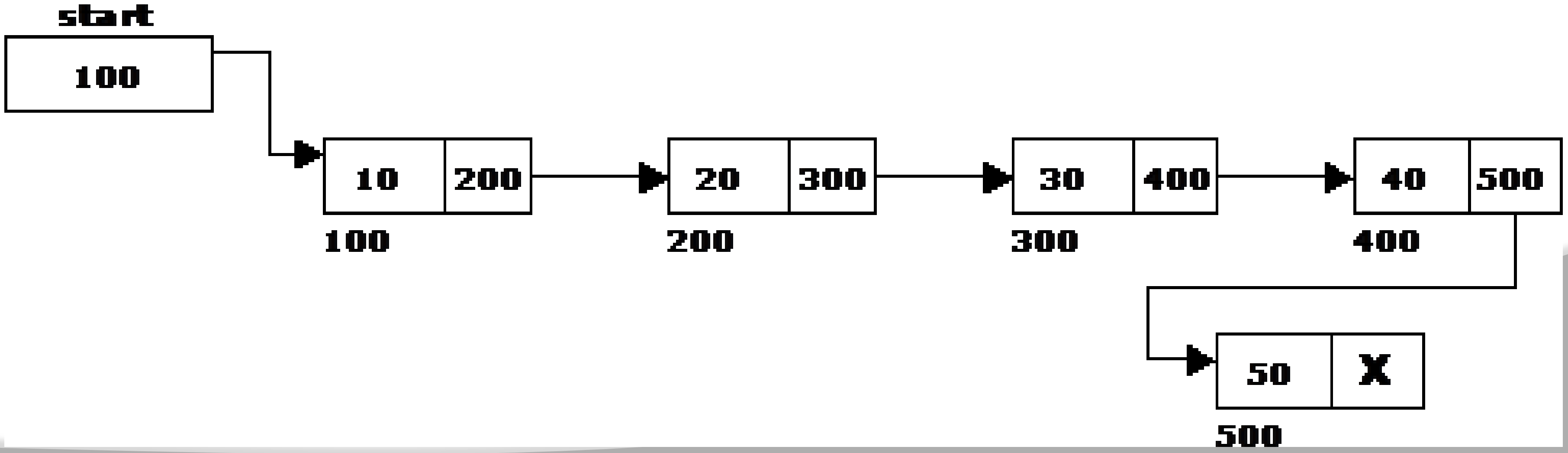
Inserting a node at the end

```
1.Create a newnode
2.Set newnode.data=data
3.Set newnode.next=NULL
4.Set temp=start
5.Repeat step 6 whiletemp.next!=NULL
6.          set temp=temp.next
   [end of loop]
7.Set temp.next=newnode
8.Exit
```

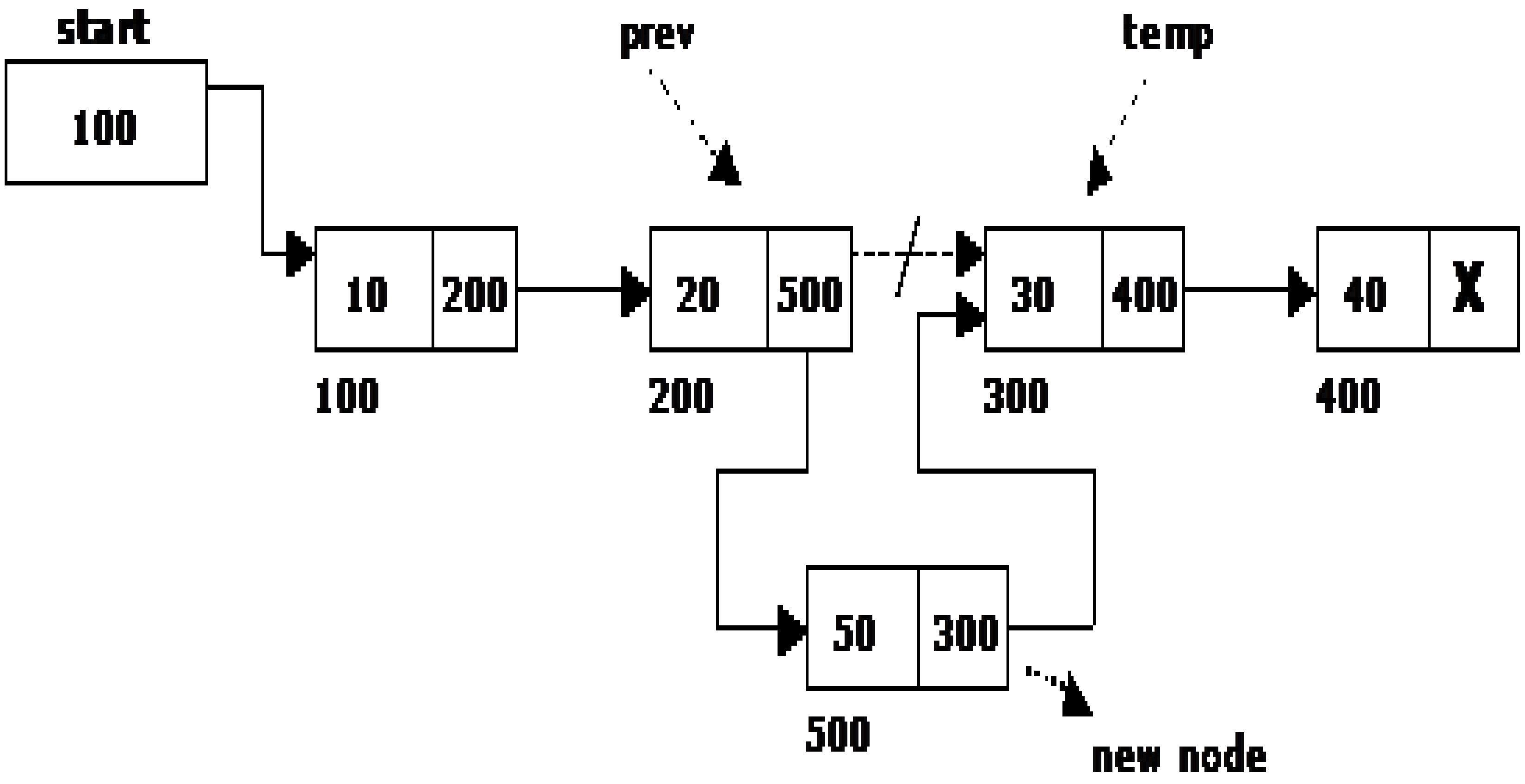


Inserting a node at the end

```
def insertend(self):
        n=int(input("enter value"))
        newnode=Node(n)
        if(self.start==None):
            self.start=newnode
        else:
            temp=self.start
            while temp.next!=None:
                temp=temp.next
            temp.next=newnode
```



Inserting a node at intermediate position



Inserting a node at intermediate position

```
def insertmid(self):
    n=int(input("enter value"))
    newnode=Node(n)
    pos=int(input("enter position"))
    c=self.count()
    if(self.start==None):
        self.start=newnode
    else:
        if pos>1 and pos<=c:
            temp=self.start
            prev=temp
            i=1
            while i<pos:
                prev=temp
                temp=temp.next
                i+=1
            newnode.next=temp
            prev.next=newnode
```

Python Classes and Objects

Python is an object oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class
To create a class, use the keyword class:

Example:
Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

use the class named MyClass to create objects:

Example

```
class MyClass:
        x = 5
p1 = MyClass()
print(p1.x)
```

The __init__() Function

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Example

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Output:

```
Hello my name is John
```

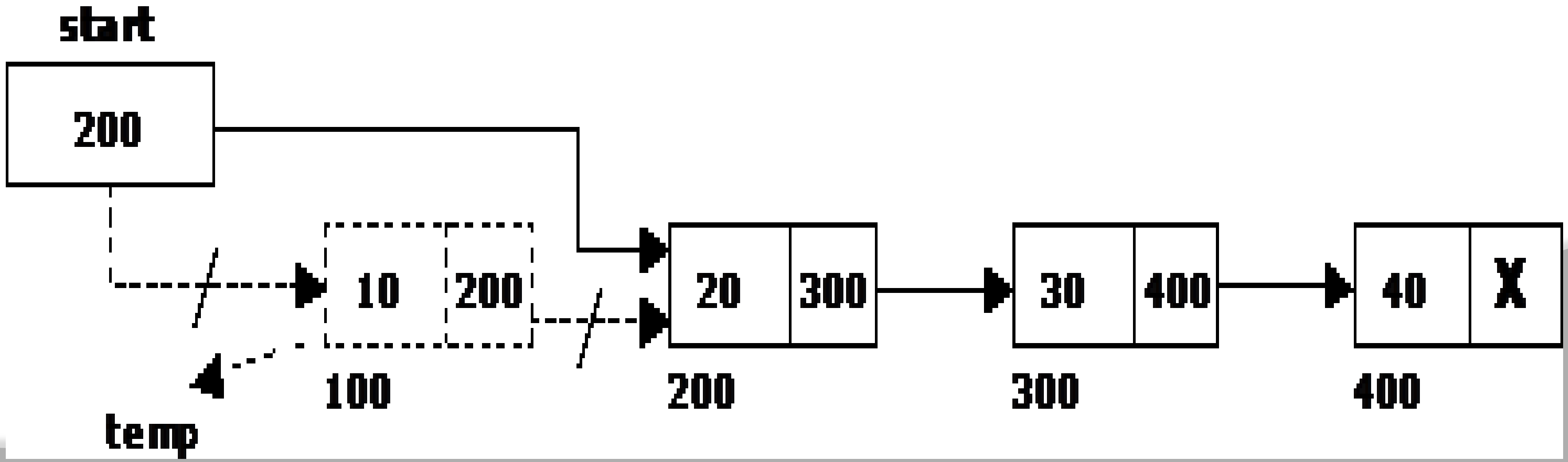
Deletion of a node from a single linked list

- Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.
  - Deleting a node at the beginning.
  - Deleting a node at the end.
  - Deleting a node at intermediate position.

Deleting a node at the beginning

- The following steps are followed, to delete a node at the beginning of the list:

```
1. If start=NULL then,
        print Empty list
        go to step 5
   [End of if]
2. Set temp=start
3. Set start=start. next
4. Free or delete temp
5. Exit
```

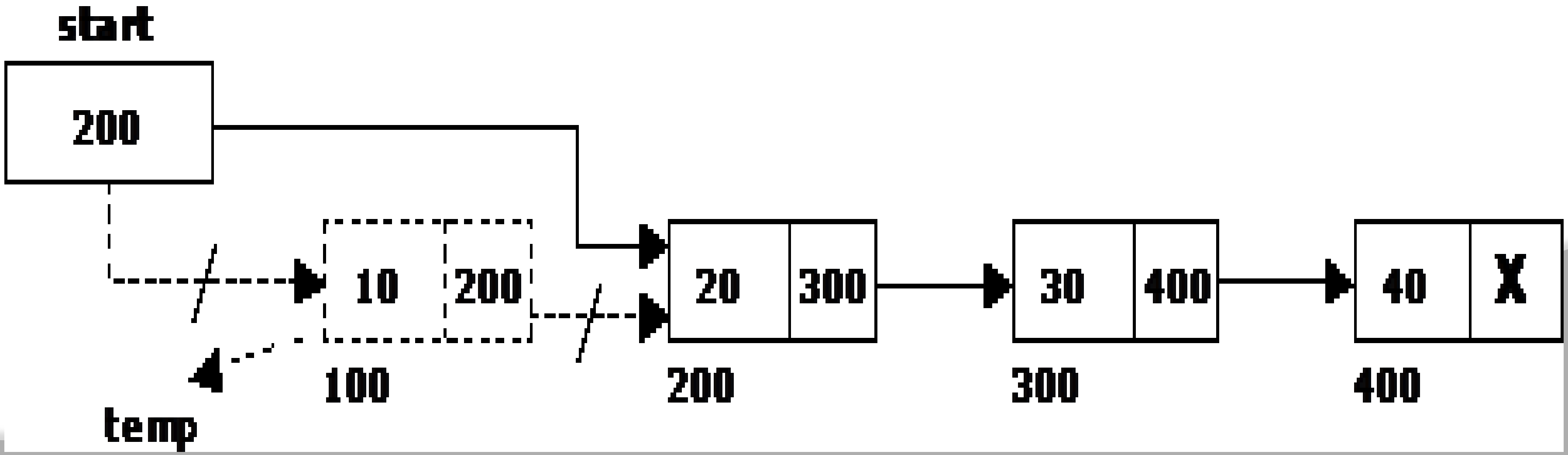


Deleting a node at the beginning

Algorithm:

```
1. If start=NULL then,
        print Empty list
        go to step 5
   [End of if]
2. Set temp=start
3. Set start=start. next
4. Free or delete temp
5. Exit
```

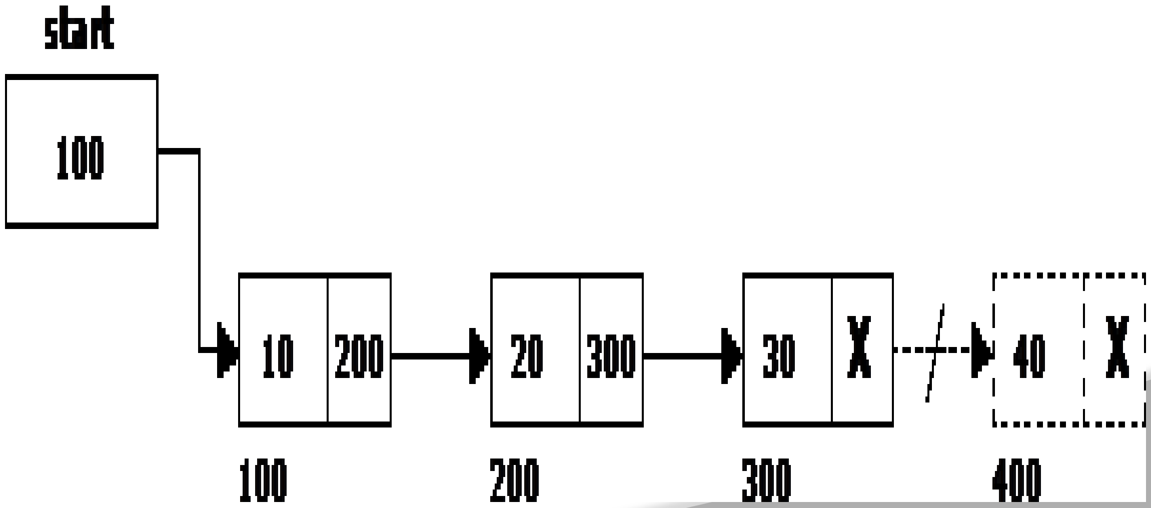
```
#Function to delete a node from the beginning
def deletebegin(self):
    global prev
    if self.start==None:
        print("empty")
    else:
        temp=self.start
        newstart=self.start.next
        del temp
    self.start=newstart
```



Deleting a node at the end

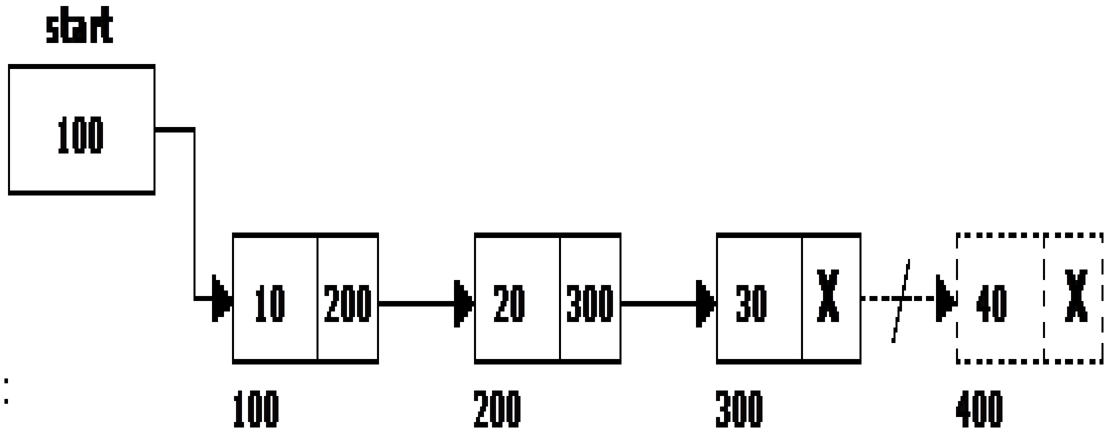
- The following steps are followed to delete a node at the end of the list:

```
1.If start=NULL, then
    print Empty list
    go to step 8
    [end of if]
2.Set temp=start
3.Repeat step 4 and 5 while temp.next!=NULL
4.    Set pretemp=temp
5.    Set temp=temp.next
      [end of loop]
6.Set pretemp.next=NULL
7.Free temp
8.Exit
```



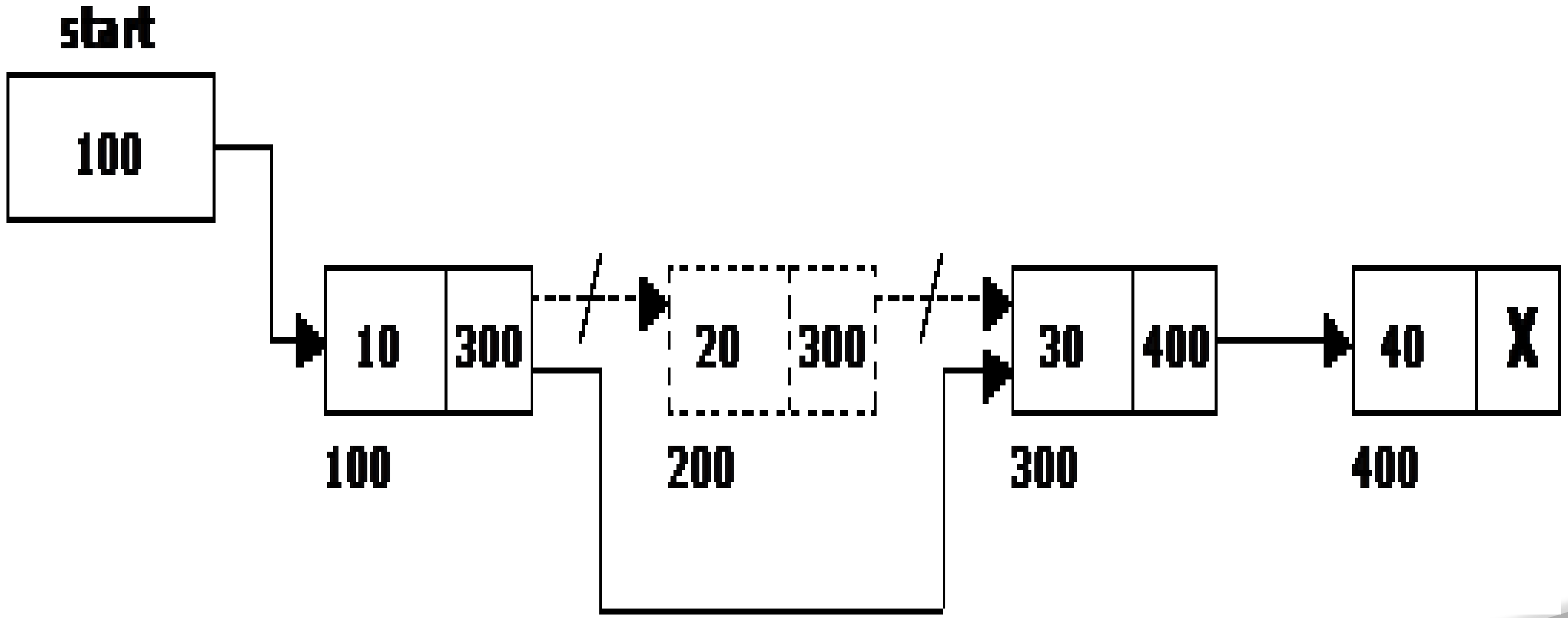
Deleting a node at the end

```
#function to delete last node of a single linked list
def deleteend(self):
        global prev
        if self.start==None:
            print("empty")
        else:
            temp=self.start
            prev=self.start
            while temp.next!=None:
                prev=temp
                temp=temp.next
            prev.next=None
            del temp
```



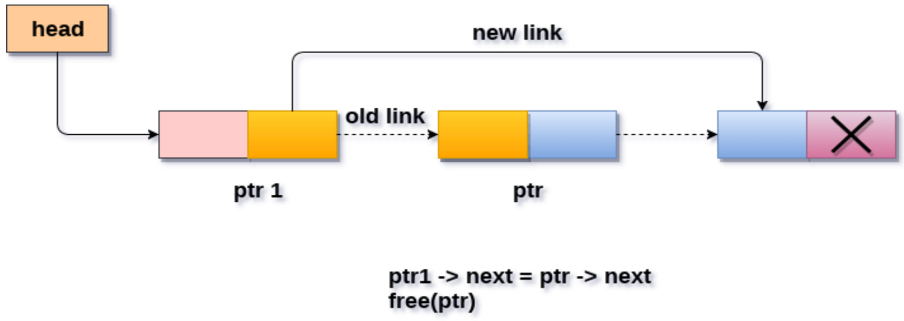
Deleting a node at intermediate position

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted and the other one the node which is present before that node. For this purpose, two variables are used: temp and prev.



Deleting a node at intermediate position

```
STEP 1: IF start = NULL
WRITE Empty List
    GOTO STEP 11
   END OF IF
STEP 2: SET TEMP = start
STEP 3: SET I = 1
STEP 4: REPEAT STEP 5 TO 8 UNTIL I<position
STEP 5: preTEMP = TEMP
STEP 6: TEMP = TEMP .NEXT
STEP 7: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
    GOTO STEP 11
    END OF IF
STEP 8: I = I+1
END OF LOOP
STEP 9: preTEMP.NEXT = TEMP.NEXT
STEP 10: FREE TEMP
STEP 11: EXIT
```



Deletion a node from specified position

Deleting a node at intermediate position

```
def deletemid(self):
        i=1
        if self.start==None:
            print("Empty")
        else:
            position=int(input("enter position"))
            c=self.count()
            if position>c:
                print("check position")
            elif position>1 and position<=c:
                temp=prev=self.start
                while i<position:
                    prev=temp
                    temp=temp.next
                    i+=1
                prev.next=temp.next
                del temp
            else:
                print("check position")
```

Counting number of nodes

```
1.If start=NULL then,
      print count=0
        [end of if]
2.Set count=1
3.Set temp=start
4.Repeat step 5 and 6 while temp.next!=NULL
5.      Set count=count+1
6.      Set temp=temp.next
        [end of loop]
7.Exit
```

Counting number of nodes

```
def count(self):
        nc=0
        temp=self.start
        while temp!=None:
            nc+=1
            temp=temp.next
        print("Number of nodes=",nc)
        return nc
```

Linked list representation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically.

- A stack can be represented by using nodes of the linked list.

- Each node contains two fields **: data(info) and next(link)**

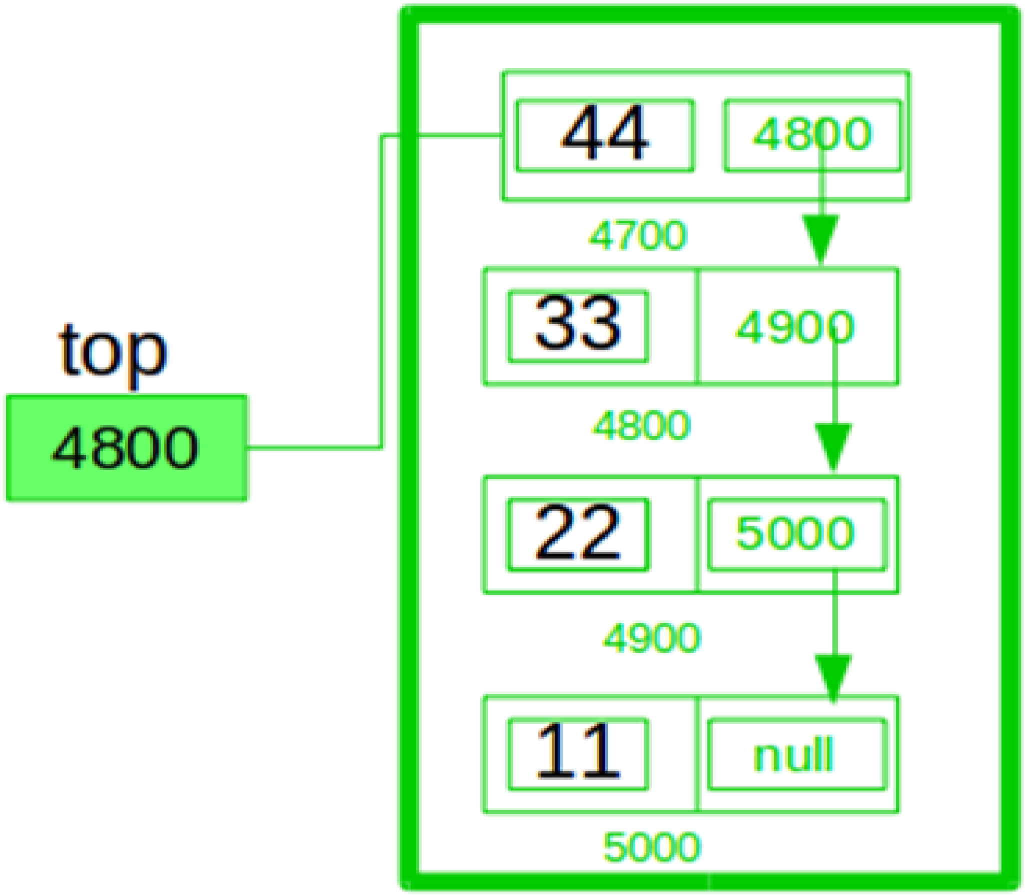
- The data field of each node contains an item in the stack and the corresponding next field points to the node containing the next item in the stack

- The top refers to the top most node (The last item inserted) in the stack.

- all the single linked list operations perform based on Stack operations LIFO(last in first out)

Linked list representation of stack

The **start** variable of the linked list is used as **top**



Operations on a Linked Stack

Stack Operations:

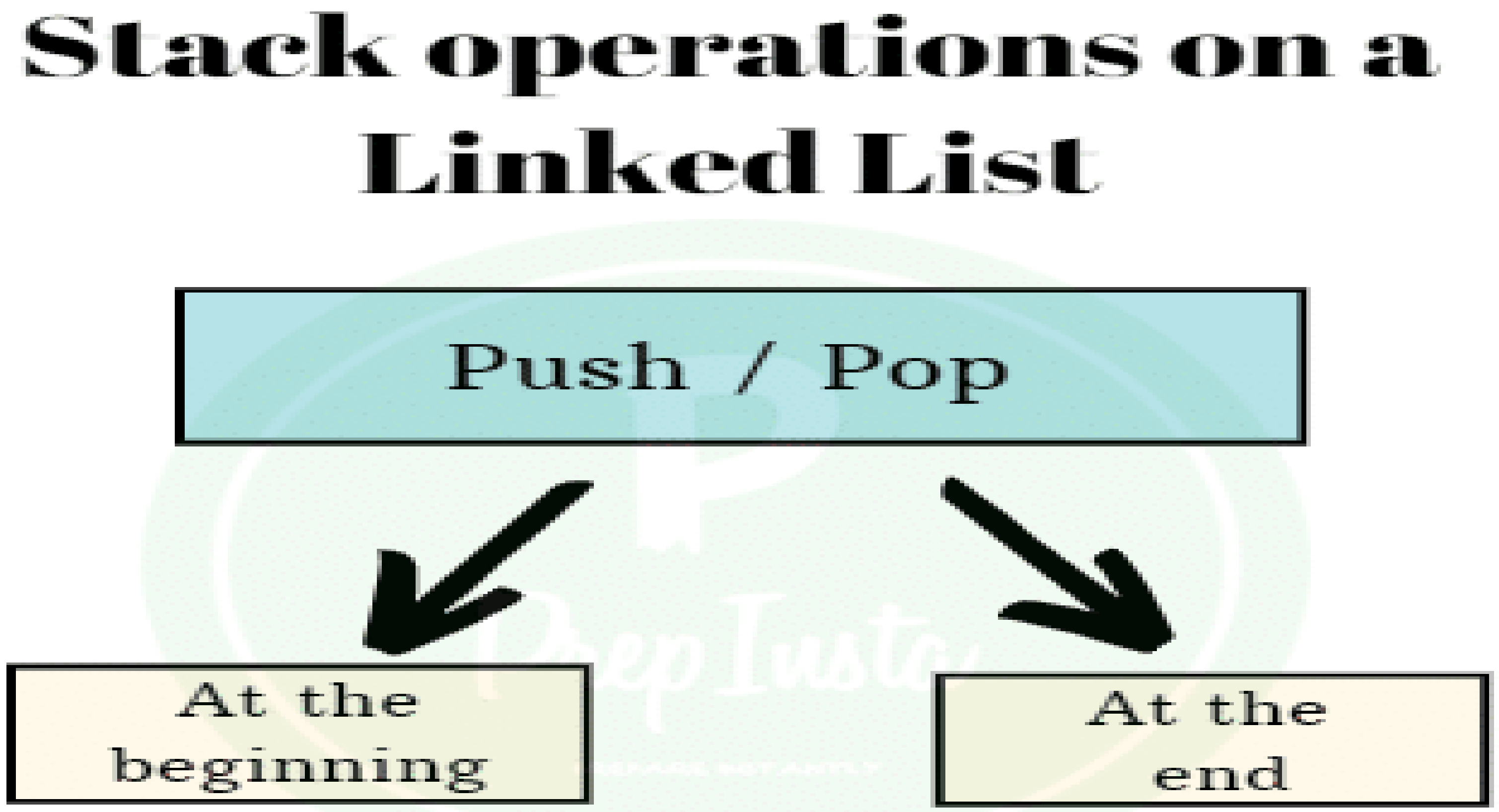
push() : Insert the element into linked list nothing but which is the top node of Stack.

pop() : Return top element from the Stack and move the top pointer to the second node of linked list or Stack.

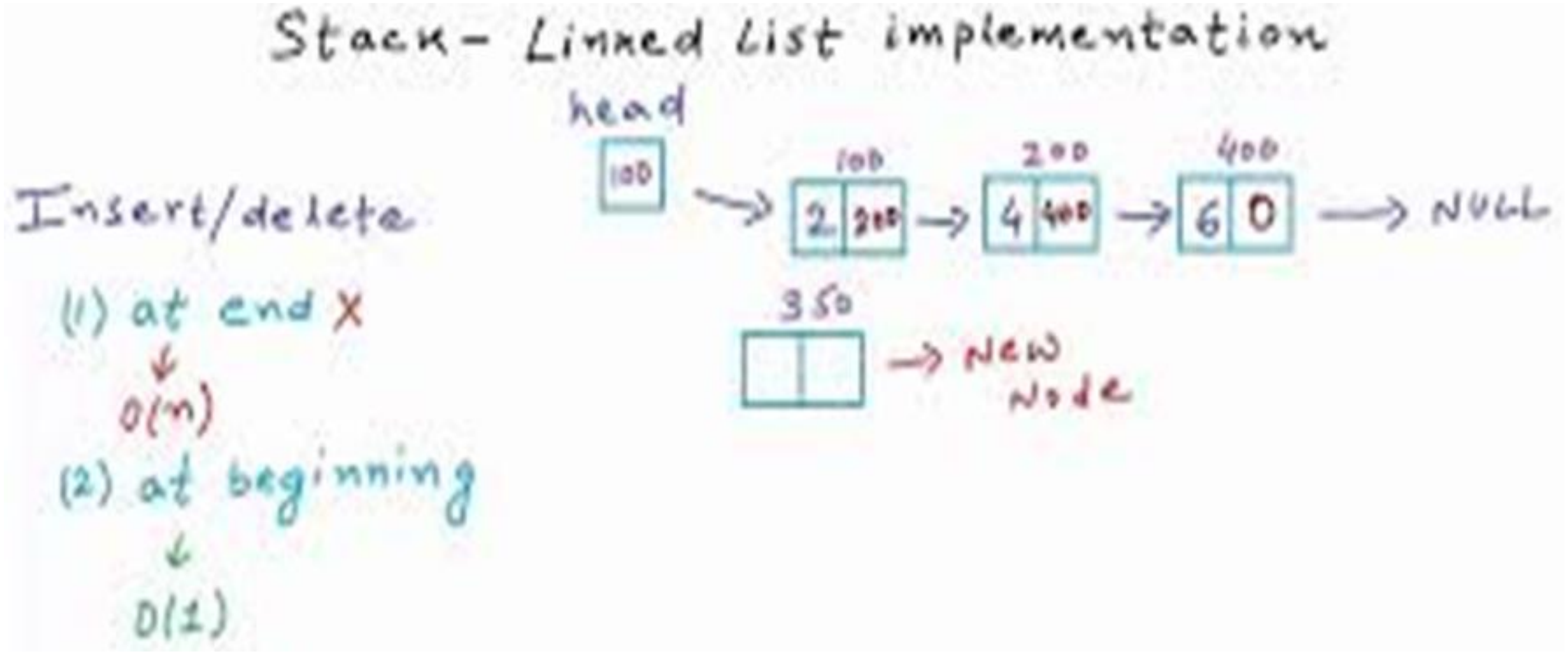
peek(): Return the top element.

display(): Print all element of Stack.

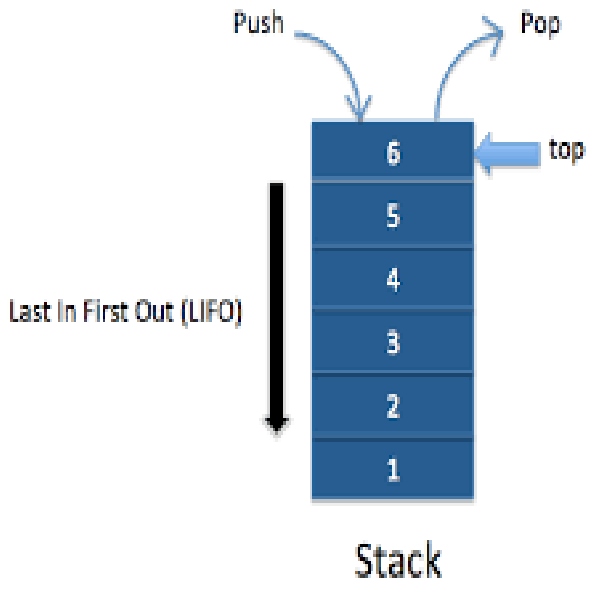
Operations on a Linked Stack

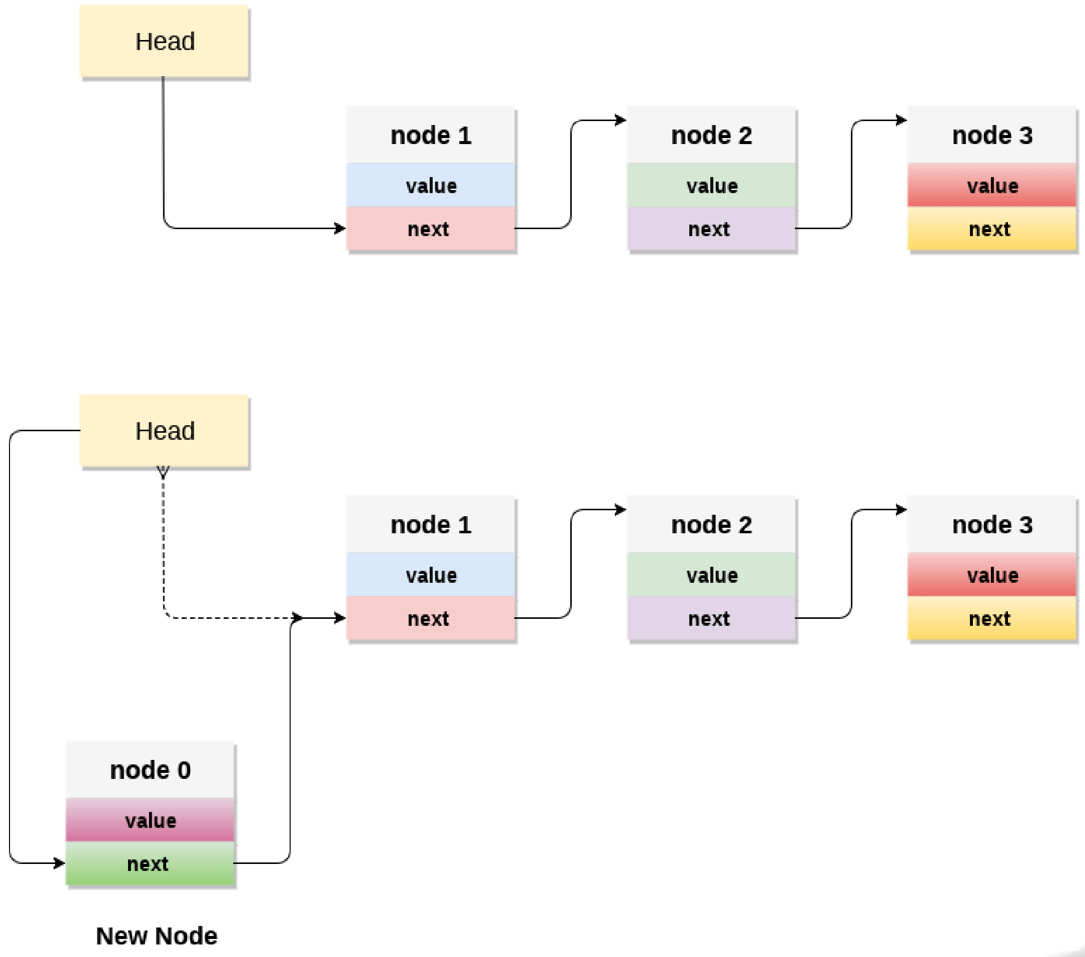


Operations on a Linked Stack



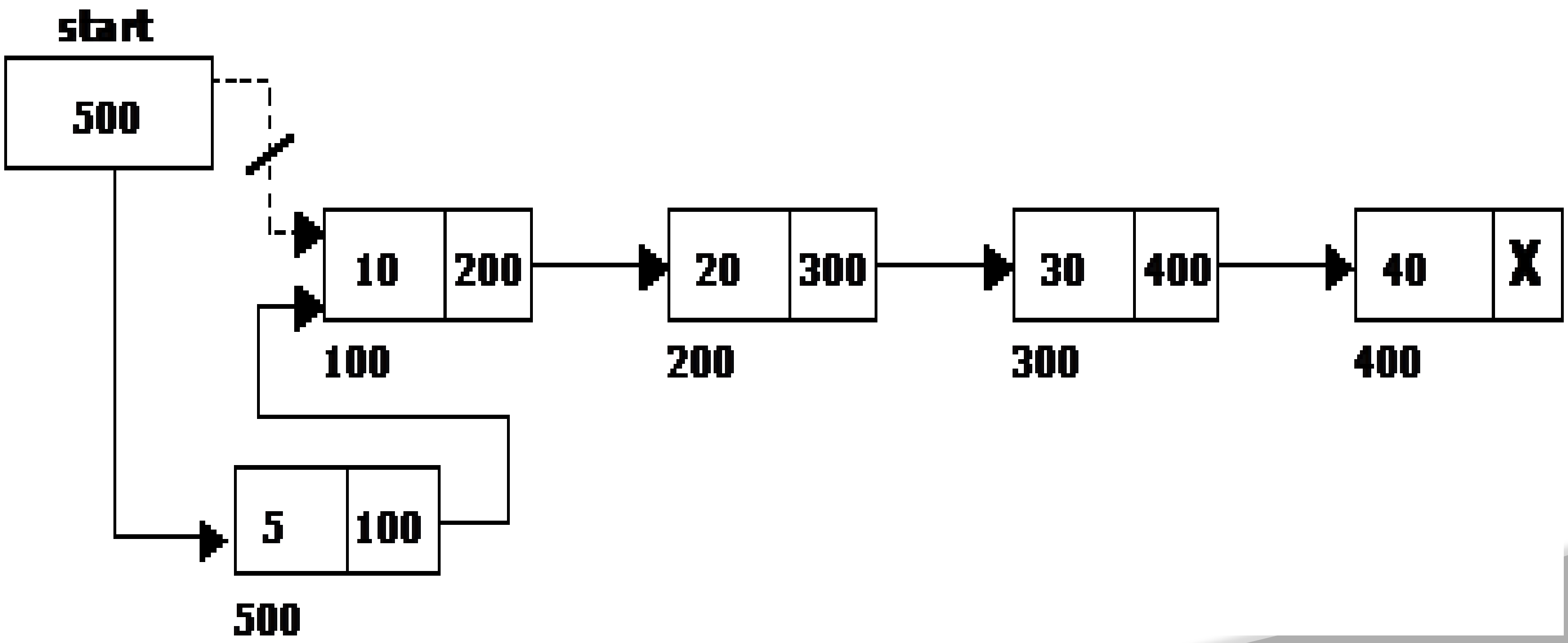
Push Operation





Inserting a node at the beginning

1.Create a nwenode
2.Set newnode.data=data
3.Set newnode.next=start
4.Set start=newnode
5.Exit



Push Operation

push() : Insert the element into linked list.

The new element is added at the top most position of the stack.

Steps to push an element into a stack:

```
1.create the new node
2.set newnode.data=data
3.if top=NULL, then
          set newnode.next=NULL
          set top=newnode
  else
          set newnode.next=top
          set top=newnode
          [end of if]
4.end
```

Function to push an element into a stack:

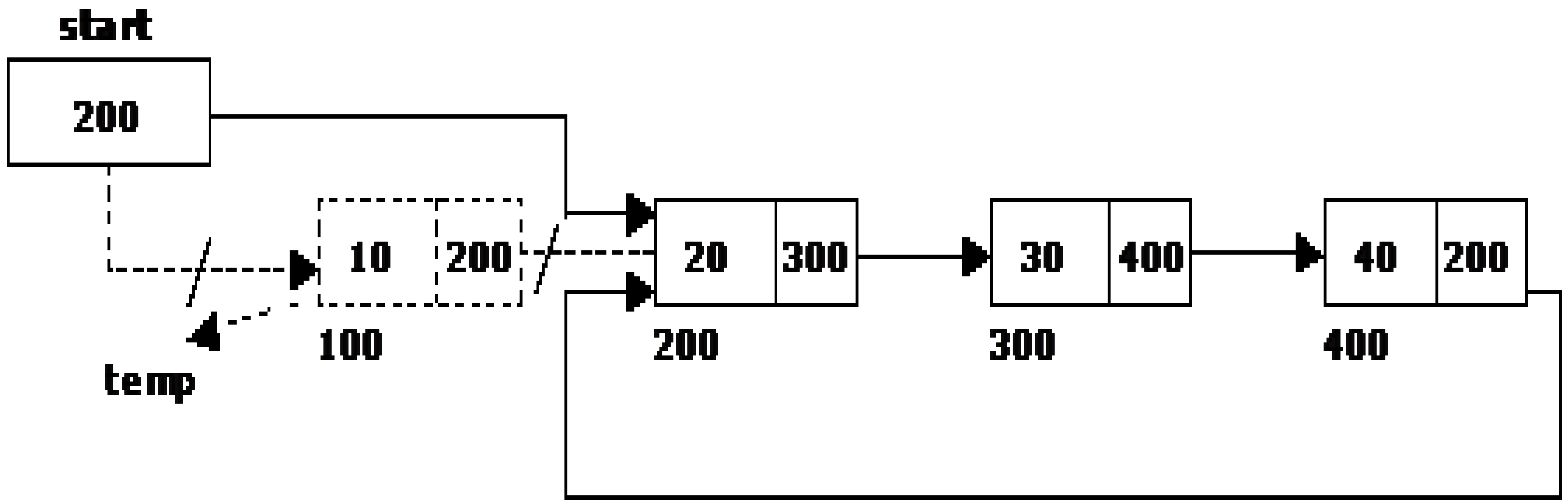
```
def push(self):
        data=int(input("enter value"))
        newnode=Node(data)
        if(self.top==None):
            self.top=newnode
        else:
            temp=self.top
            newnode.next=temp
            self.top=newnode
```

pop() : delete the topmost element from the stack
Return top element from the Stack and move the top pointer to the second node of linked list or Stack.

Steps to pop an element from stack:

```
1.If top=NULL,then
        print Empty or underflow
        [end of if]
2.Set temp=top
3.Set top=top.next
4.Free temp
5.end
```

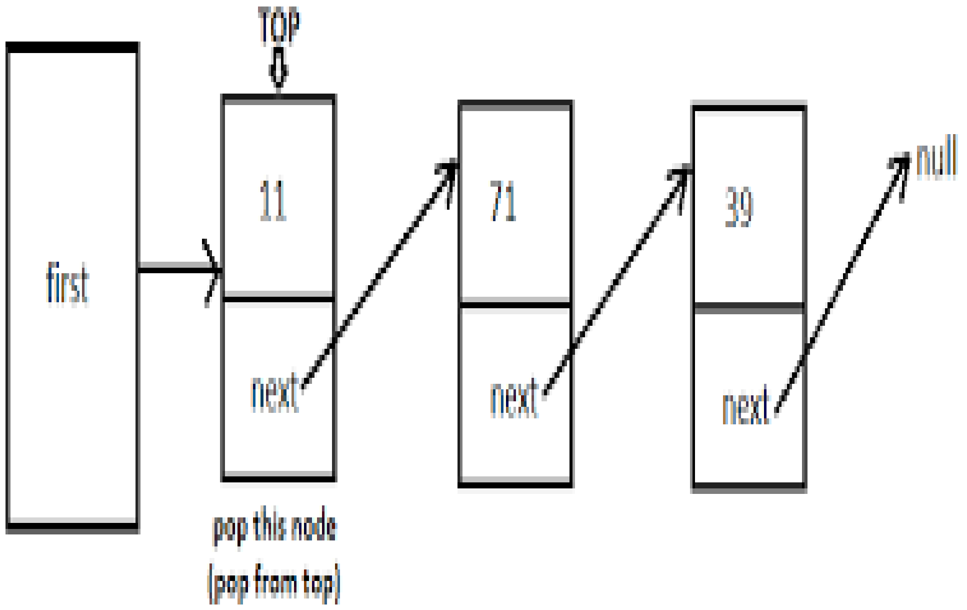
Pop Operation



Steps to pop an element from stack:

```
def pop(self):
        global prev
        if (self.top==None):
            print("empty")
        else:
            temp=self.top
            newstart=self.top.next
            del temp
            self.top=newstart
```

Before popping in stack (implemented using Linked List)



Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1.Copy the head pointer into a temporary pointer.

2.Move the temporary pointer through all the nodes of the list and print

3.value field attached to every node.

Display the nodes (Traversing)

```
def display(self):
        print("elements in stack are:")
        if self.start==None:
            print("empty")
        else:
            temp=self.start
            print(temp.data)
            while temp.next!=None:
                temp=temp.next
                print(temp.data)
```

Peek operation

```
def peek(self):
        if self.start==None:
            print("empty")
        else:
            temp=self.start
            print(temp.data)
```

Linked list representation of Queue

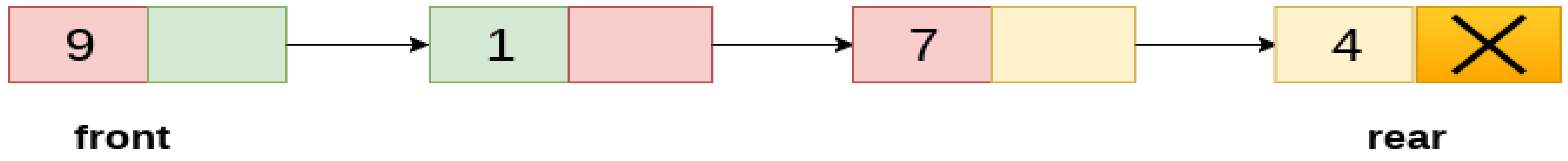
In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Linked list representation of Queue

Insertion and deletions are performed at rear and front end respectively.
If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

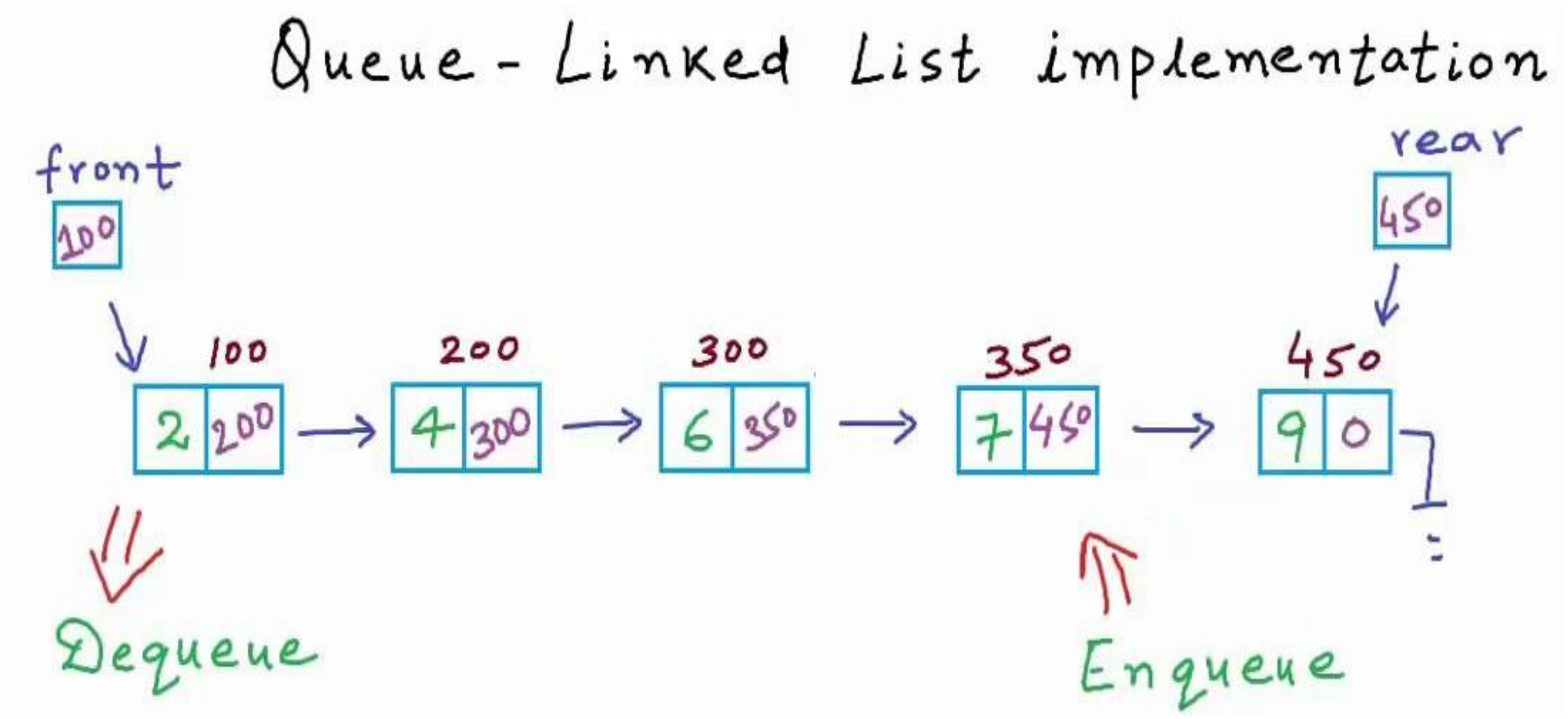
Operation on Linked Queue

enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.

deQueue() This operation removes the front node and moves *front* to the next node.

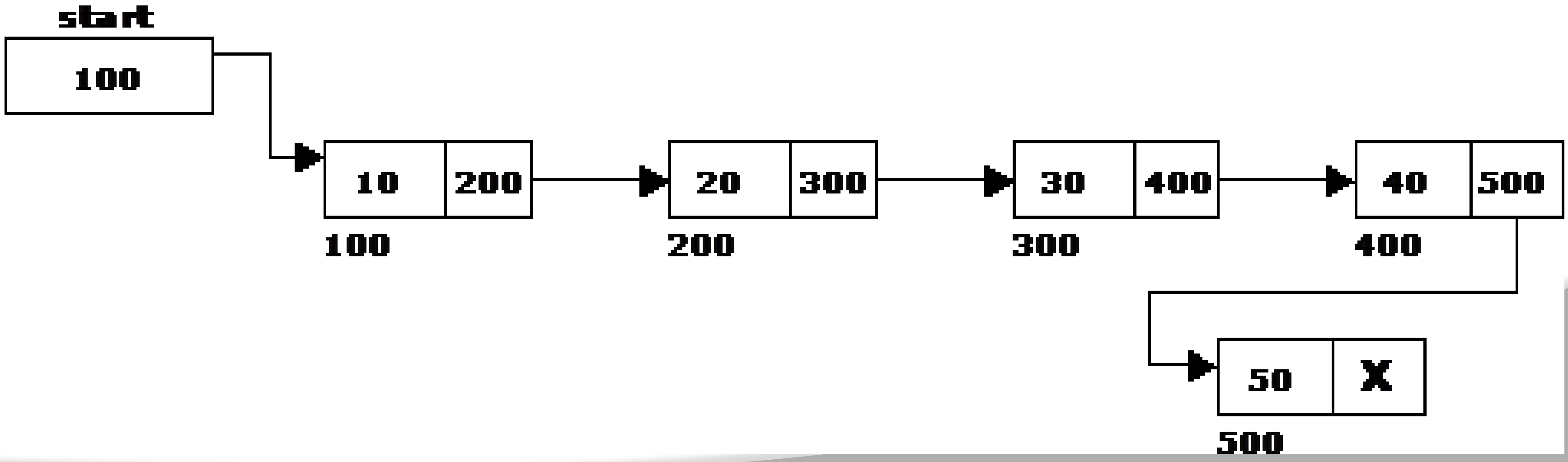
display() - Displaying the elements of Queue

Operation on Linked Queue



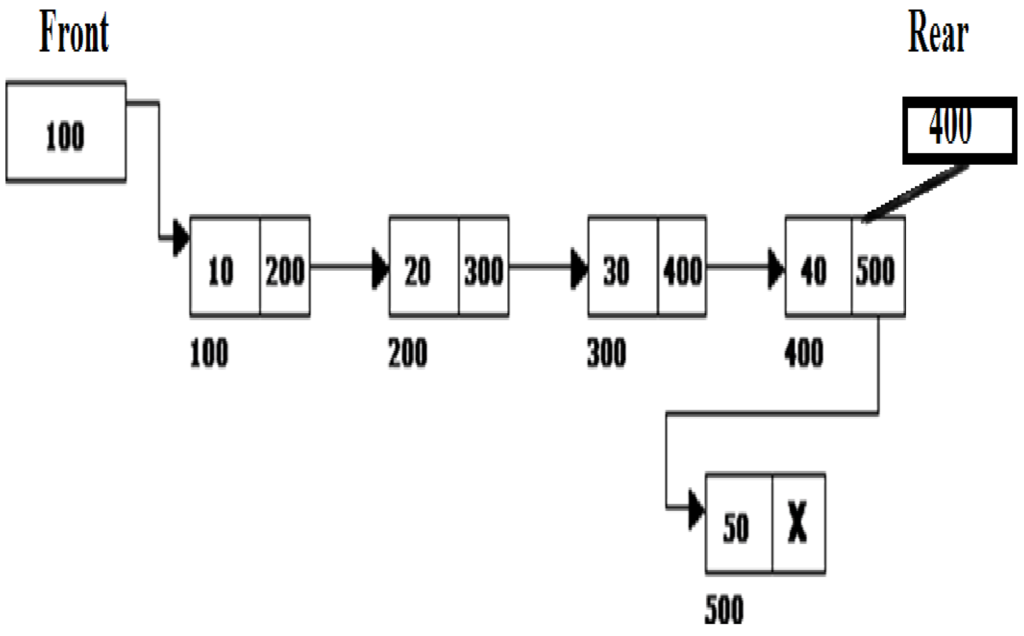
Inserting a node at the end (enQueue)

```
1.Create a newnode
2.Set newnode.data=data
3.Set newnode.next=NULL
4.Set temp=start
5.Repeat step 6 whiletemp.next!=NULL
6.          set temp=temp.next
   [end of loop]
7.Set temp.next=newnode
8.Exit
```



Insertion operation (enQueue)

```
Algorithm
Step 1: Allocate the space for the new node 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
```



Deletion operation(deQueue)

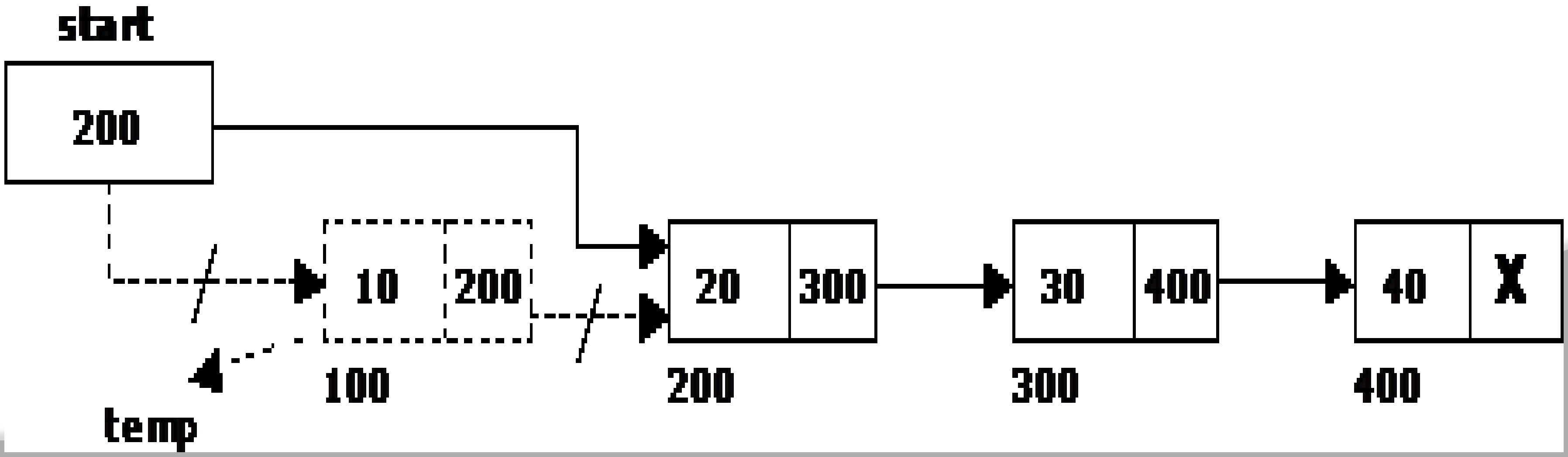
Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

Deleting a node at the beginning(deQueue)

Algorithm:

```
1. If start=NULL then,
        print Empty list
        go to step 5
   [End of if]
2. Set temp=start
3. Set start=start. next
4. Free or delete temp
5. Exit
```



Deletion operation(deQueue)

Algorithm

```
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
```

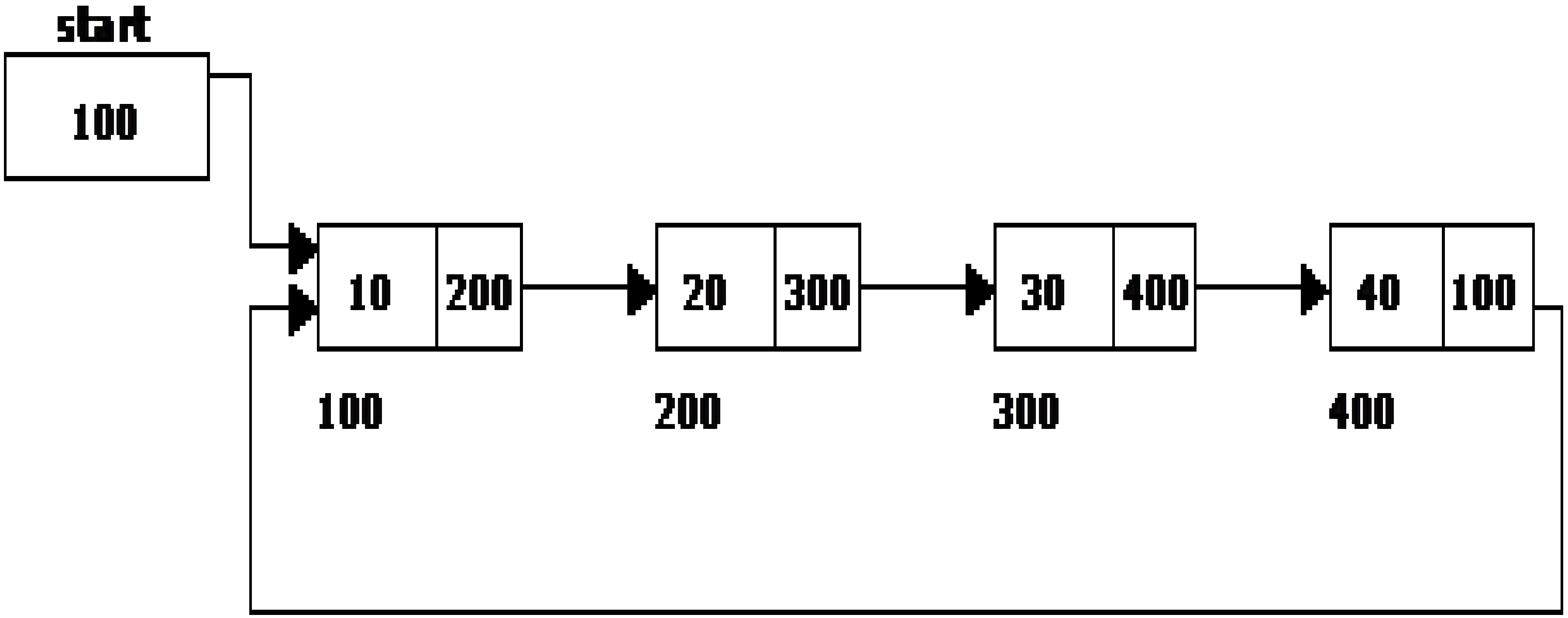
Circular Single Linked List

It is just a single linked list in which the link field of the last node points back to the address of the first node.

A circular linked list has no beginning and no end. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list.

Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

Circular Single Linked List and its basic operations

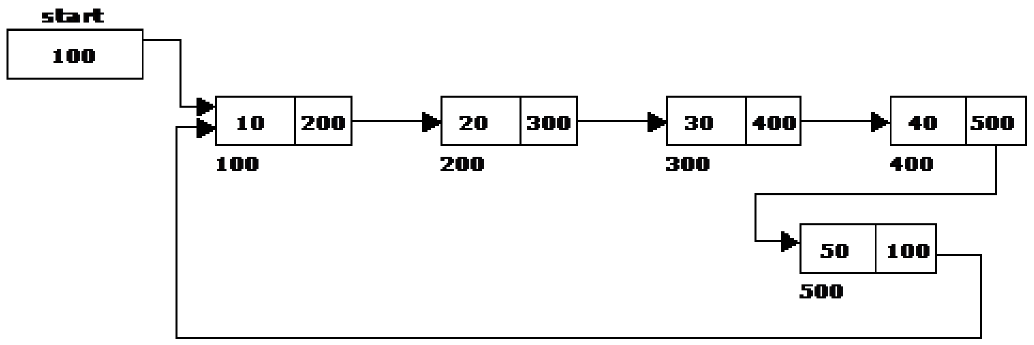


Circular Single Linked List basic operations

The basic operations in a circular single linked list are:

- Creation
- Insertion
- Deletion
- Traversing

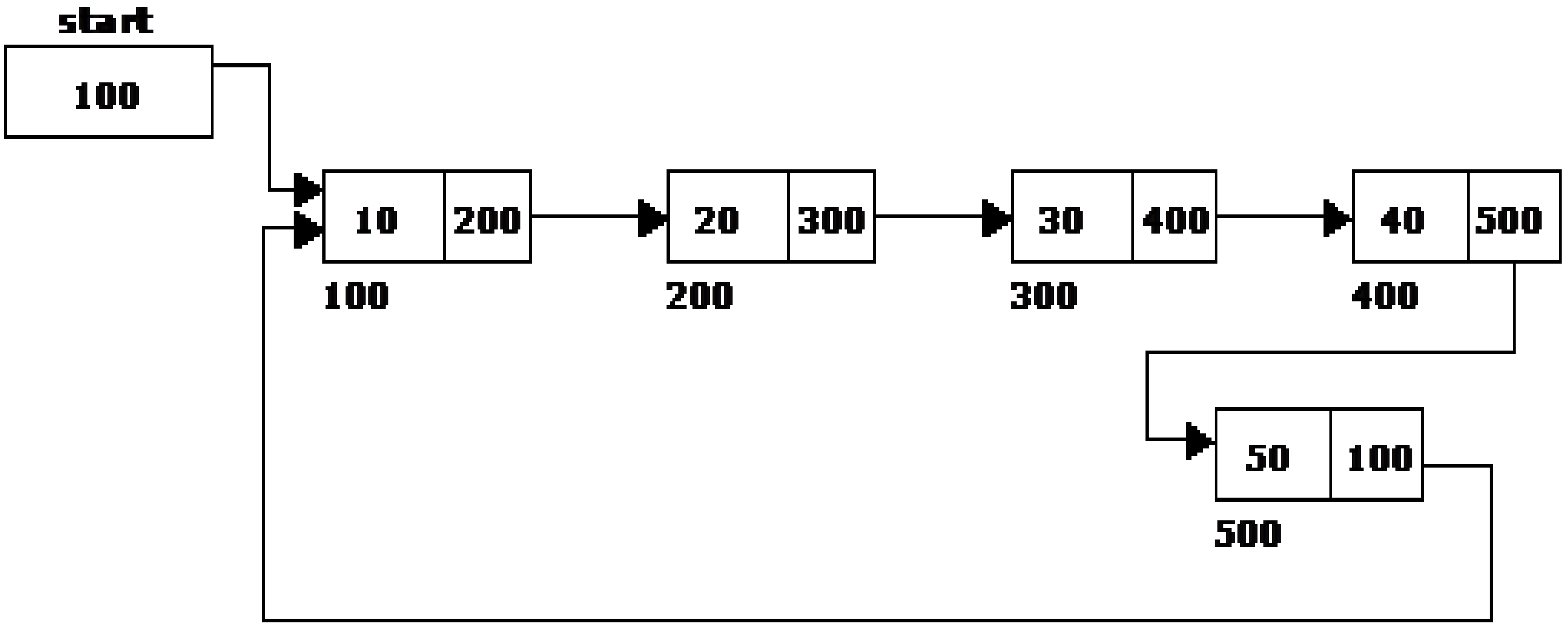
Creating a circular single Linked List with N number of nodes



Steps:

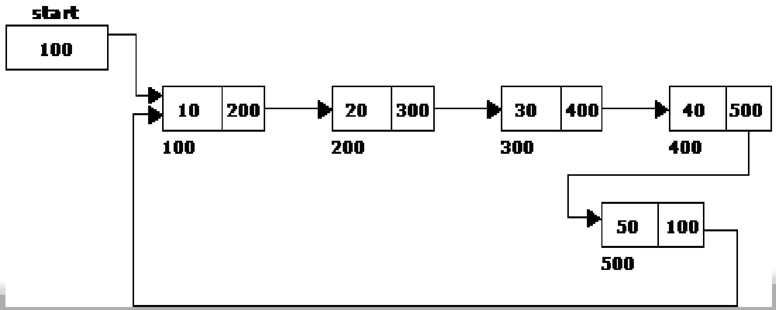
```
1.   Create a newnode
2.   Set newnode.data=data
3    Set newnode.next=start
4.   If start=NULL then,
            start=newnode
     [End of if]

5.   Set temp=start
6.   Repeat step 7 while tem.next!=start
7.          set temp=temp.next
            [end of loop]
8.   Set temp.next=newnode
9.   Exit
```

Creating a circular single Linked List with N number of nodes



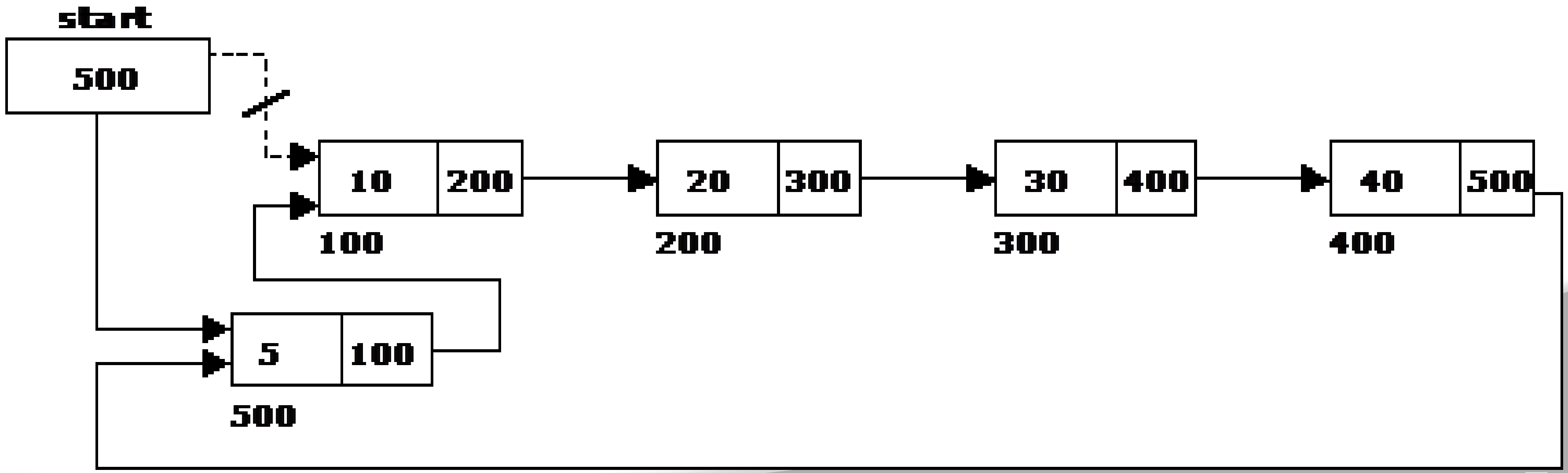
Creating a circular single Linked List with N number of nodes

```
def createlist(self):
        n=int(input("enter number of nodes in the
                                      list"))
        for i in range(n):
            data=int(input("enter value"))
            newnode=Node(data)
            if self.start==None:
                self.start=newnode
                newnode.next=newnode
            else:
                temp=self.start
                while temp.next!=self.start:
                    temp=temp.next
                temp.next=newnode
                newnode.next=self.start
```



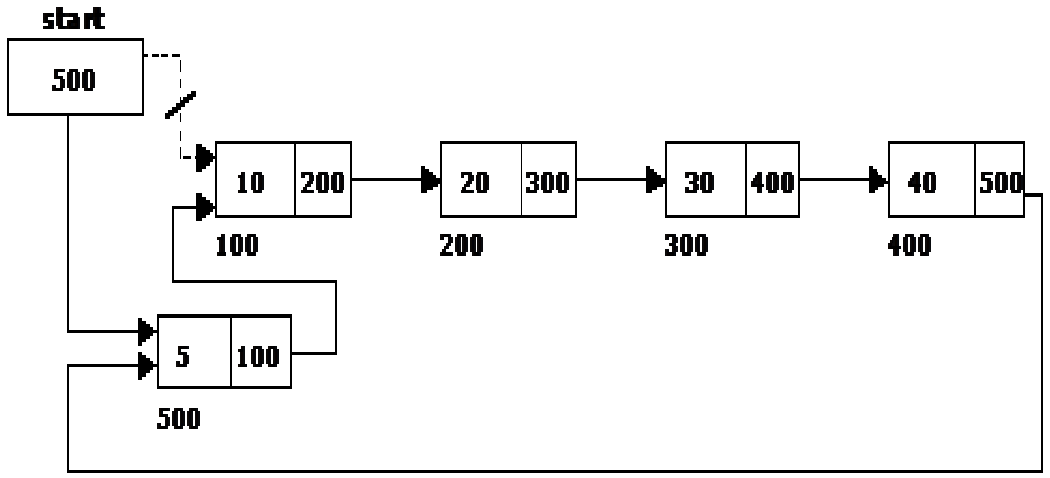
Inserting a node at the beginning

```
1.Create a nwenode
2.Set newnode.data=data
3.Set temp=start
4.Repeat sep 5 while temp.next!=start
5.    tem=temp.next
6.Set newnode.next=start
7.Set temp.next=newnode
8.Set start=newnode
9.exit
```

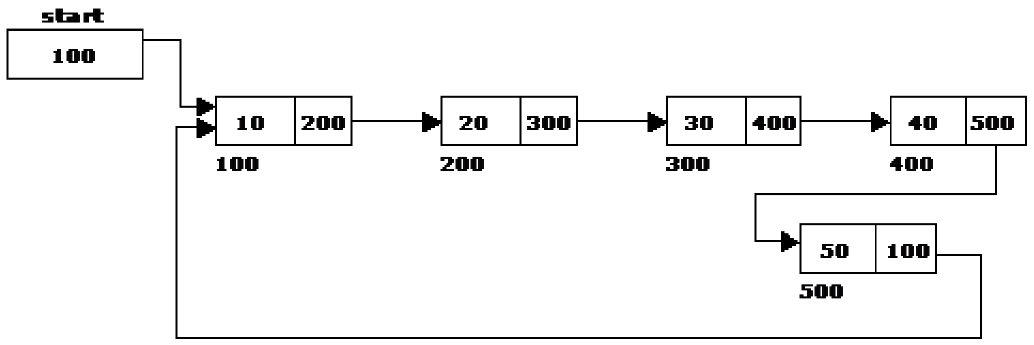


Inserting node at the beginning

```
def insertbegin(self):
        data=int(input("enter value"))
        newnode=Node(data)
        if self.start==None:
            self.start=newnode
            newnode.next=newnode
        else:
            temp=self.start
            while temp.next!=self.start:
                temp=temp.next
            temp.next=newnode
            newnode.next=self.start
            self.start=newnode
```



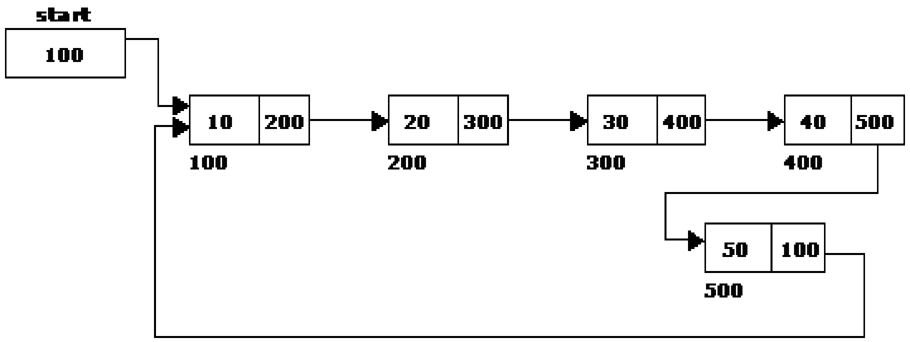
Inserting a node at the end



```
Steps:
1.    Create a newnode
2.    Set newnode.data=data
3.    Set newnode.next=start
4.    If start=NULL then,
            start=newnode
      [End of if]

5.    Set temp=start
6.    Repeat step 7 while tem.next!=start
7.          set temp=temp.next
      [end of loop]
8.    Set temp.next=newnode
9.    Exit
```

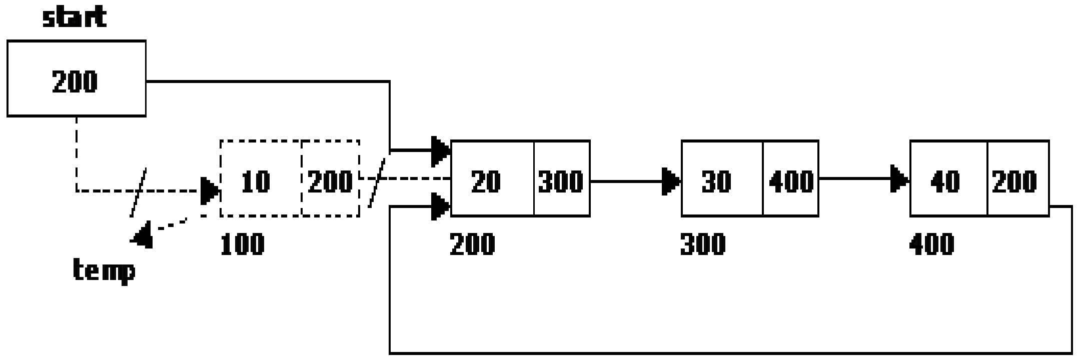
Inserting a node at the end

```
def insertend(self):
        data=int(input("enter value"))
        newnode=Node(data)
        if self.start==None:
                self.start=newnode
                newnode.next=newnode
        else:
            temp=self.start
            while temp.next!=self.start:
                temp=temp.next
            temp.next=newnode
            newnode.next=self.start
```



Deleting a node at the beginning

- The following steps are followed, to delete a node at the beginning of the list:

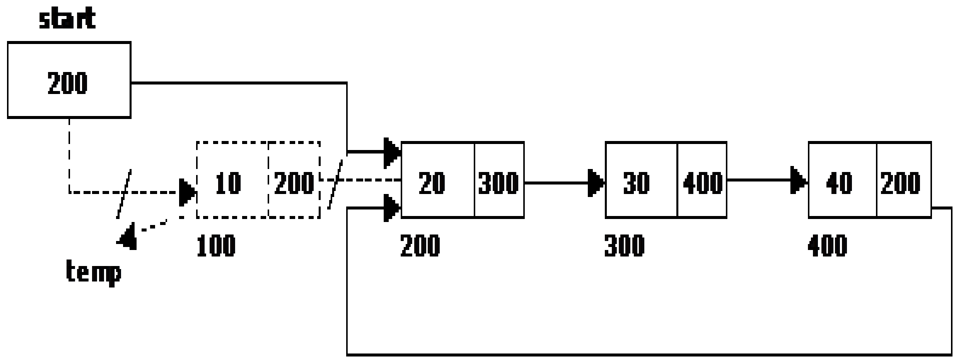


Steps:

```
1.if start=NULL then,
        write underflow
        go to step 8
2.set temp=start
3.Repeat step 4 while temp.next!=start
4.      set temp=temp.next
5.set temp.next=start.next
6.Free start
7.set start=temp.next
8.Exit
```

Deleting a node at the beginning

```
def deletebegin(self):
        temp=self.start
        if self.start==None:
            print("List is empty, deletion not possible")
        elif temp.next==temp:
            print(temp.data,"is deleted success")
            del temp
            self.start=None
        else:
            dtemp=self.start
            while temp.next!=self.start:
                temp=temp.next
            temp.next=dtemp.next
            self.start=dtemp.next
            print(dtemp.data,"is deleted success")
            del dtemp
```

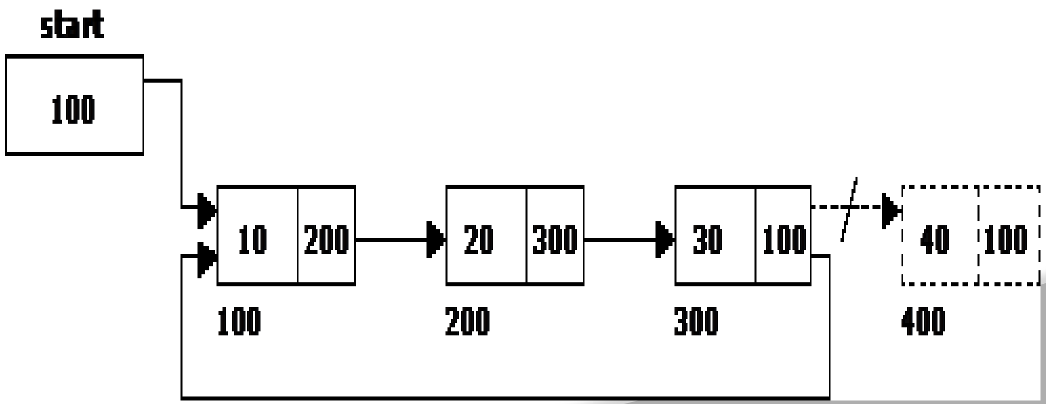


Deleting a node at the end

- The following steps are followed to delete a node at the end of the list:

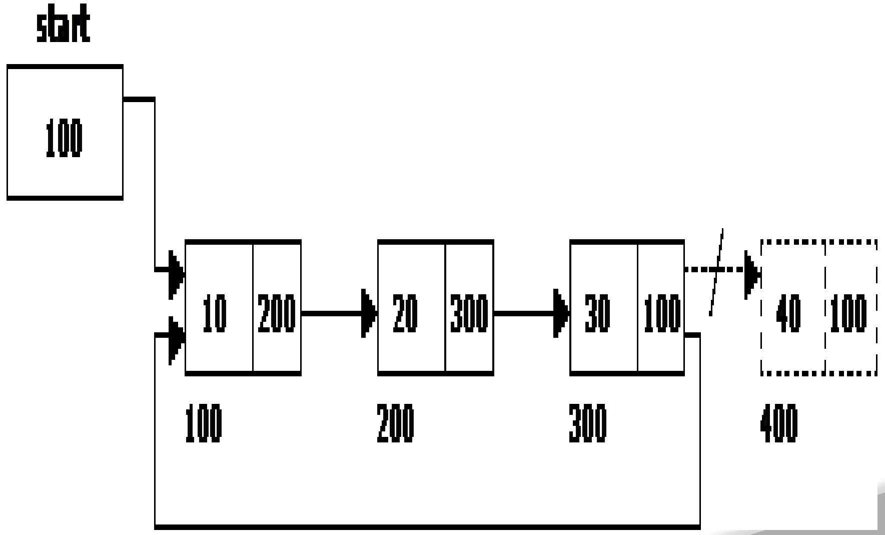
Steps:

```
1.if start=NULL then,
        write underflow
    go to step 8
2.set temp=start
3.Repeat step 4 while temp.next!=start
4.      set pretemp=temp
5.      set temp=temp.next
        [end of loop]
6.set pretemp.next=start
7.free temp
8.Exit
```



Deleting a node at the end

```
def deletend(self):
    temp=self.start
    if self.start==None:
        print("List is empty, deletion not possible")
    elif temp.next==temp:
        printf(temp.data,"is deleted success")
        del temp
        self.start=None
    else:
        ptemp=temp
        temp=temp.next
        while temp.next!=self.start:
            ptemp=temp
            temp=temp.next
        ptemp.next=self.start
        print(temp.data,"is deleted success")
        del temp
```



Traversing a circular single linked list from left to right

- The following steps are followed, to traverse a list from left to right:
- If list is empty then display _Empty List‘ message.
- If the list is not empty, follow the steps given below:

```
temp = start;
print(temp.data ,end=" ")
 while(temp.next != start)
     temp = temp .next;
     print( temp .data)
```

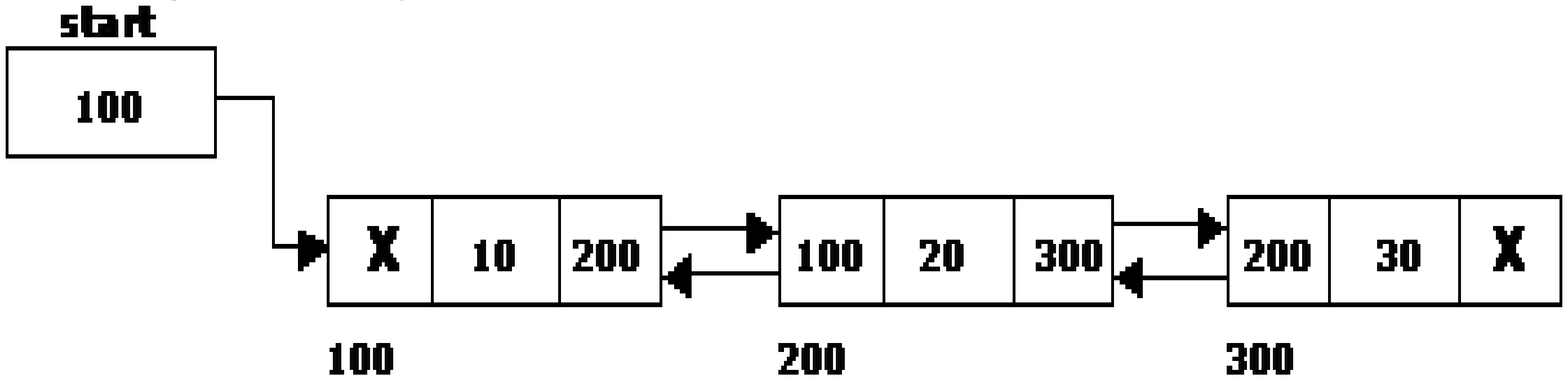
Traversing a circular single linked list from left to right

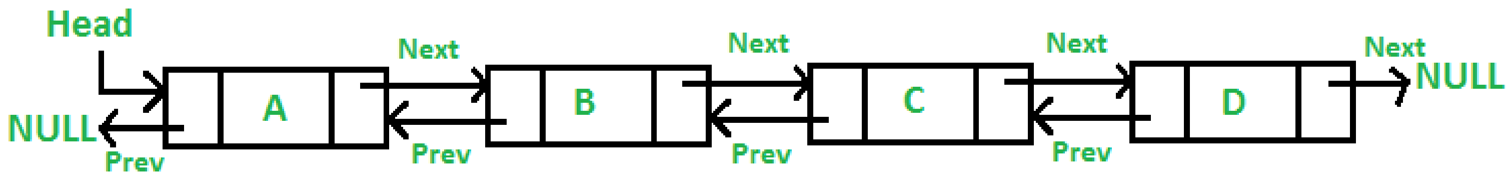
```
def display(self):
    if self.start==None:
        print("Linked list is empty")
    else:
        print("elements in single linked list are:")
        temp=self.start
        print(temp.data ,end=" ")
        while temp.next!=self.start:
            temp=temp.next
            print(temp.data ,end=" ")
```

Double Linked List

- A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes.

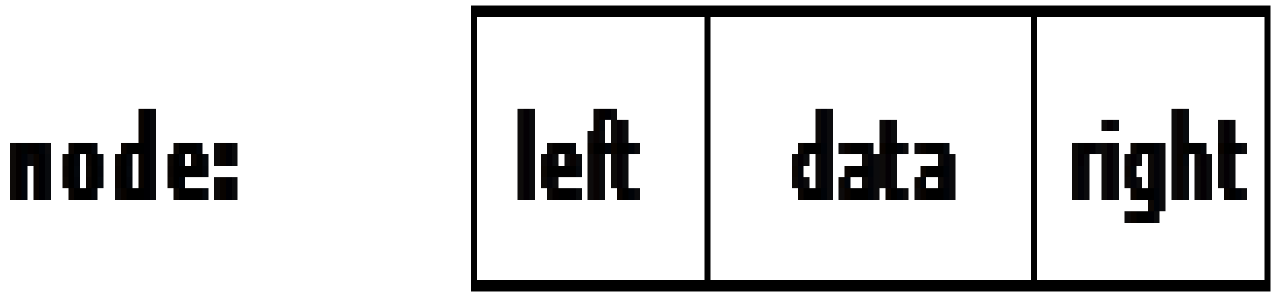
- A double linked list is a two-way list in which all nodes will have two links.
- This helps in accessing both successor node and predecessor node from the given node position.



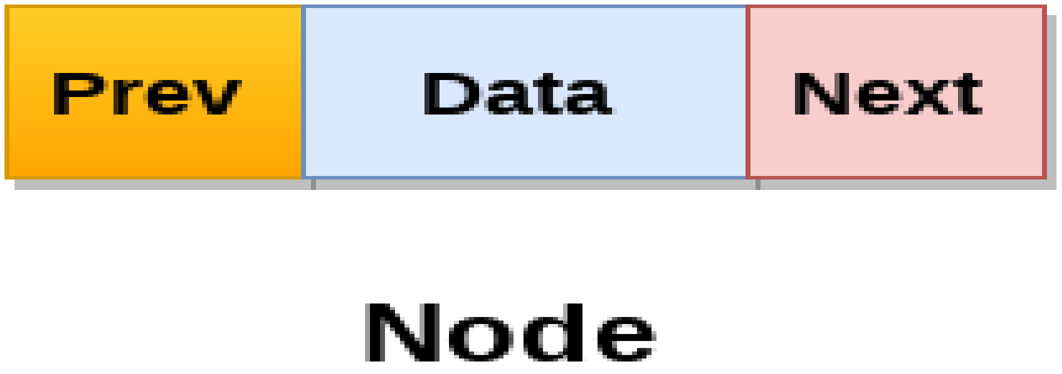


Double Linked List Node Structure

- Doubly linked list provides bi-directional traversing. Each node contains three fields:
  - Left link.
  - Data.
  - Right link.



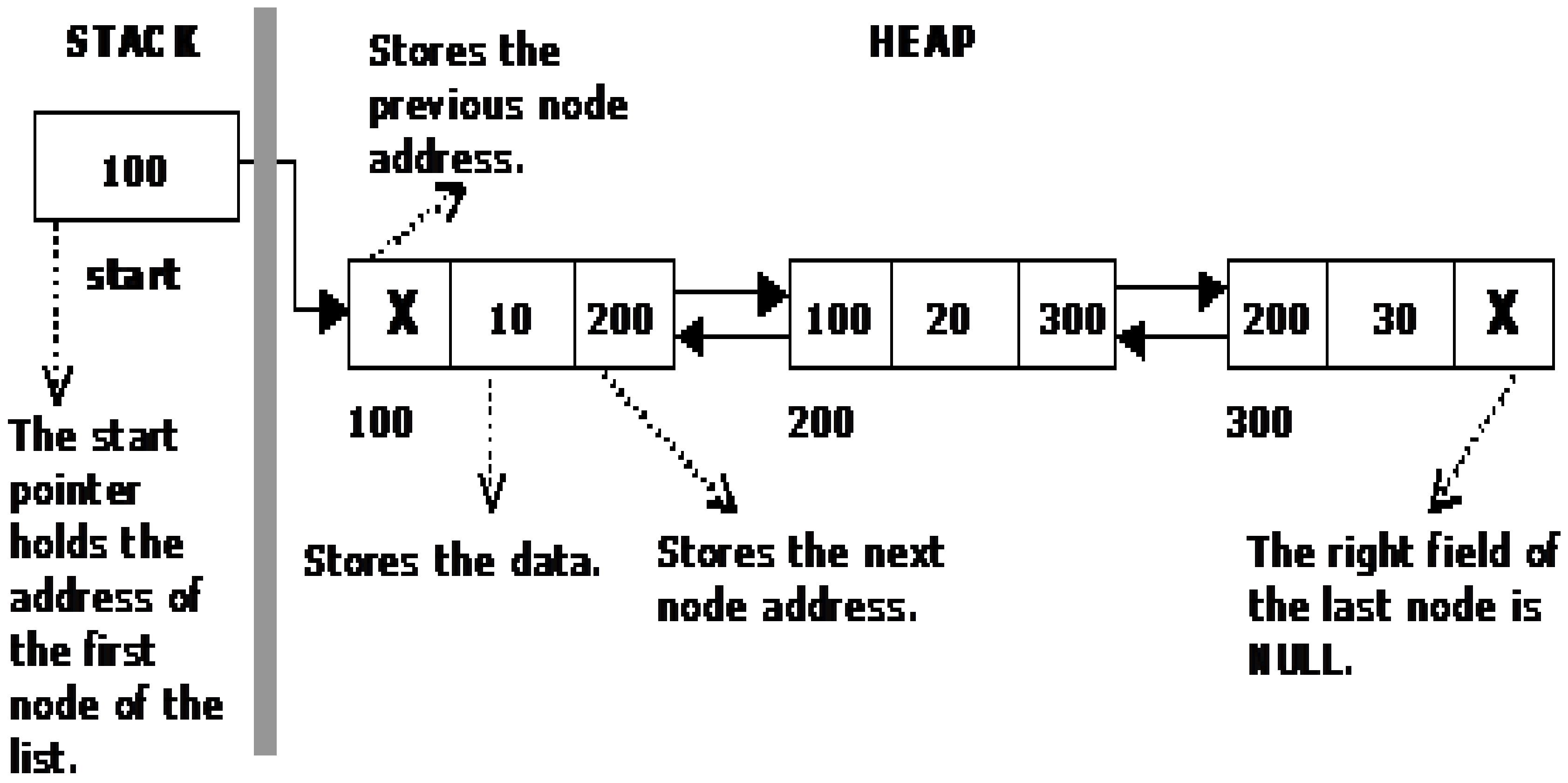
Double Linked List Node Structure



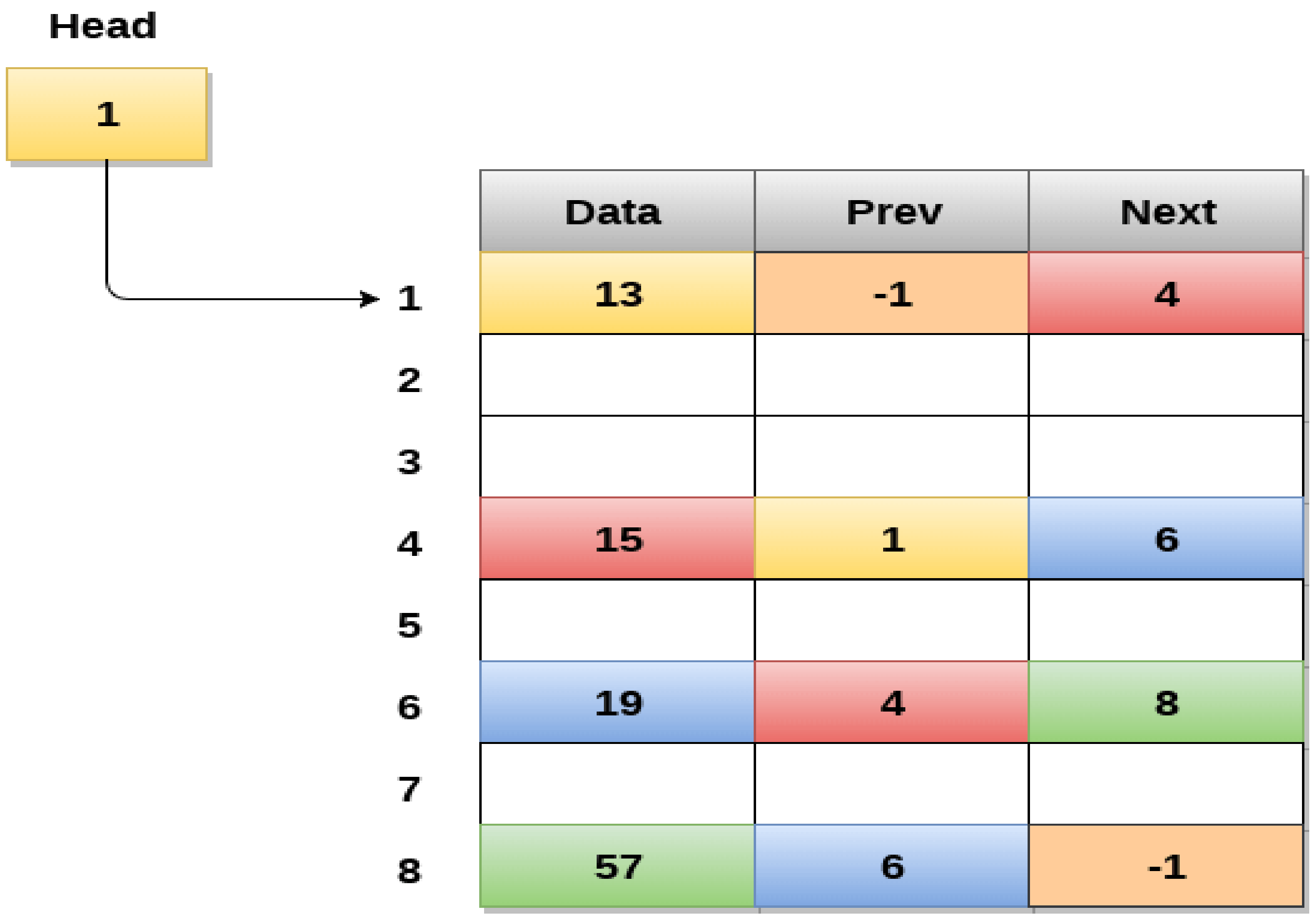
#class to create and initialize a node

```
class Node:
    def __init__(self,data):
        self.data=data
        self.prev=None
        self.next=None
```

Double Linked List


STACK
HEAP
Stores the previous node address.
100
start
The start pointer holds the address of the first node of the list.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Memory Representation of a doubly linked list

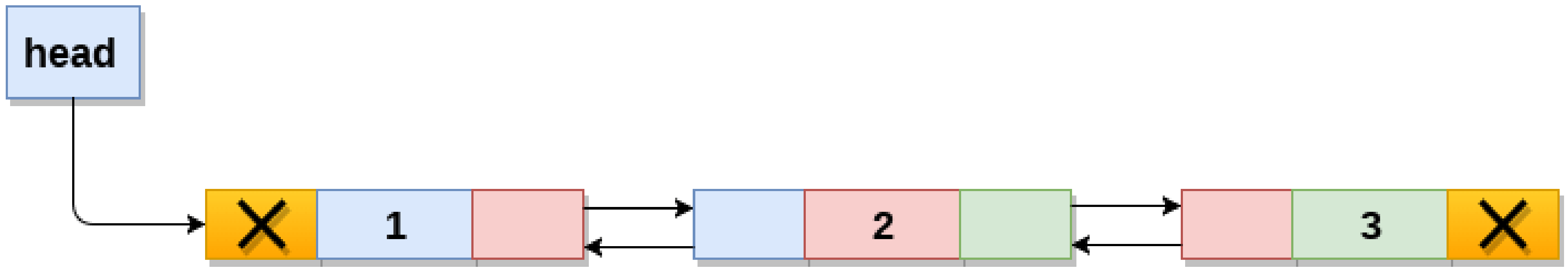


	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Basic operations in a double linked list

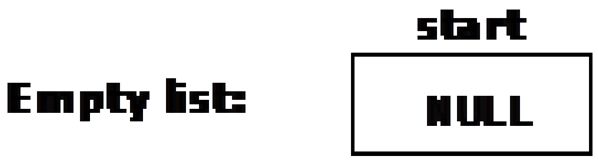
- **Creation**
- **Insertion**
- **Deletion**
- **Traversing**
- The e beginning of the double linked list is stored in a "**start/head**" pointer which points to the first node. The first node‘s left link and last node‘s right link is set to NULL.

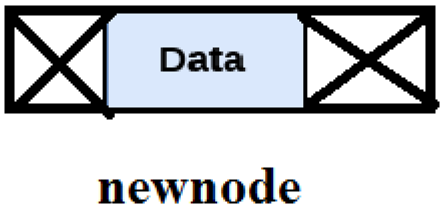


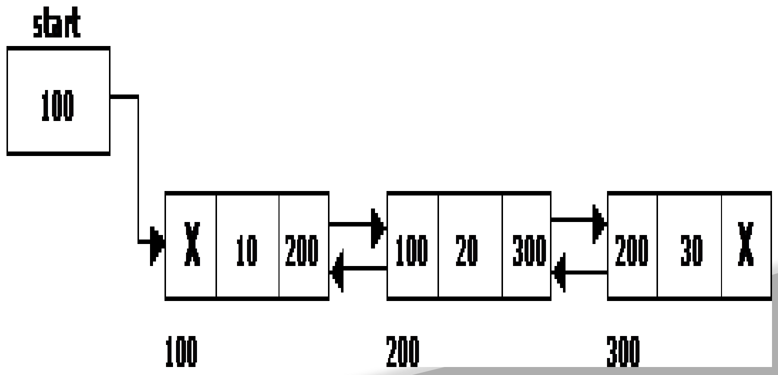
Doubly Linked List

Creating a Double Linked List with N number of nodes

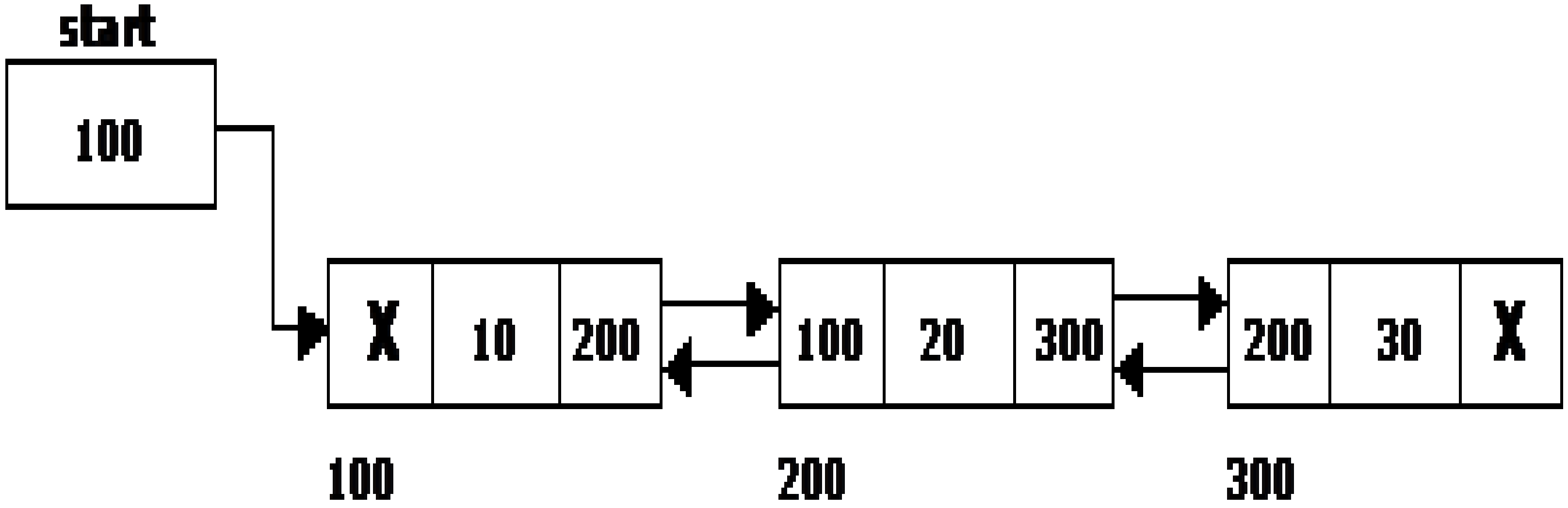
```
class DList:
   def __init__(self):
      self.start=None
   def createlist(self):
      n=int(input("enter number of nodes
in the list"))
      for i in range(n):
         data=int(input("enter value"))
         newnode=Node(data)
         if self.start==None:
            self.start=newnode
         else:
            temp=self.start
            while temp.next!=None:
               temp=temp.next
            temp.next=newnode
            newnode.prev=temp
```







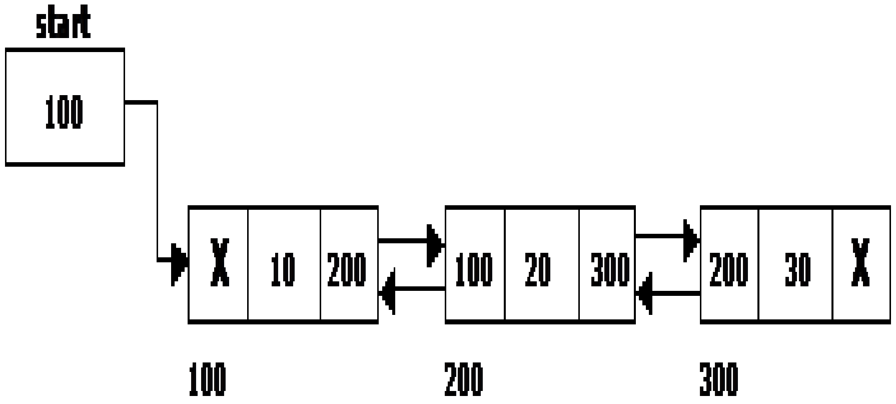
Creating a Double Linked List with N number of nodes

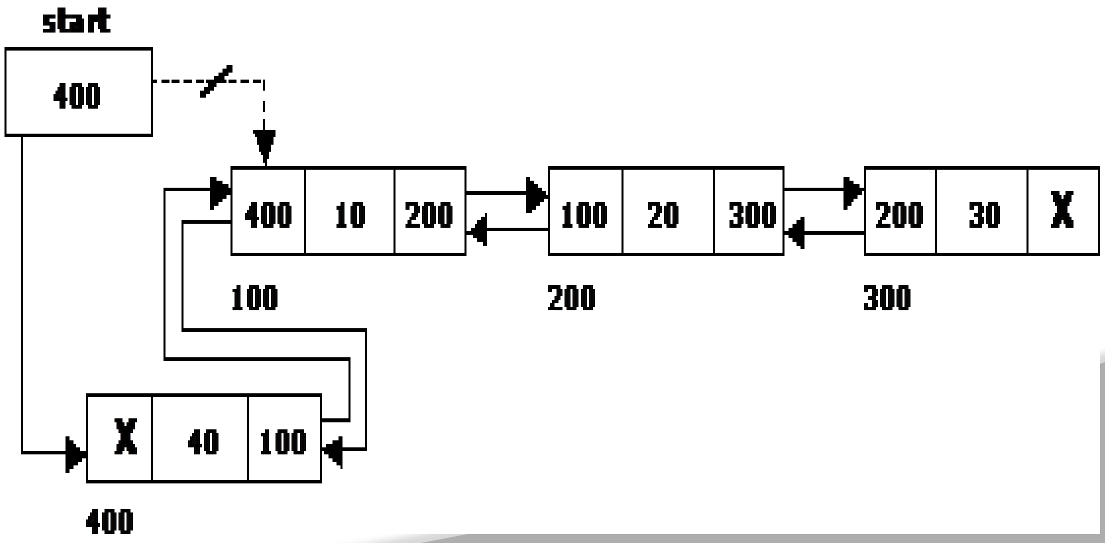


Inserting a node at the beginning

The following steps are to be followed to insert a new node at the beginning of the list:

```
def insertbegin(self):
    data=int(input("enter value"))
    newnode=Node(data)
    if self.start==None:
        self.start=newnode
    else:
        newnode.next=self.start
        self.start=newnode
        start . left = newnode
        newnode.prev=None
```

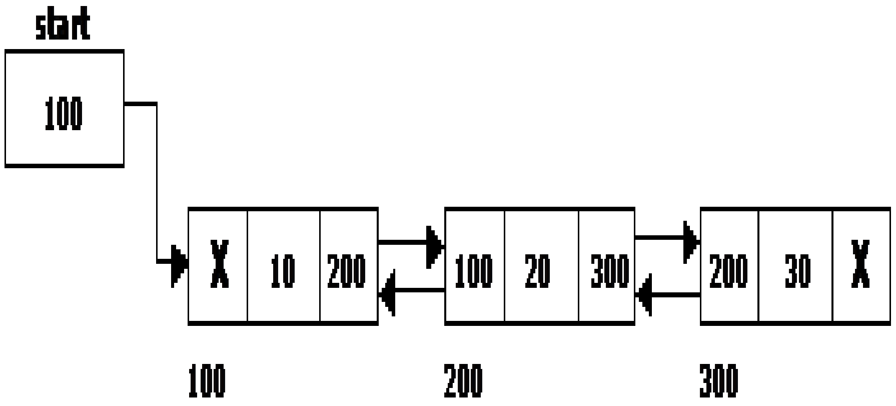


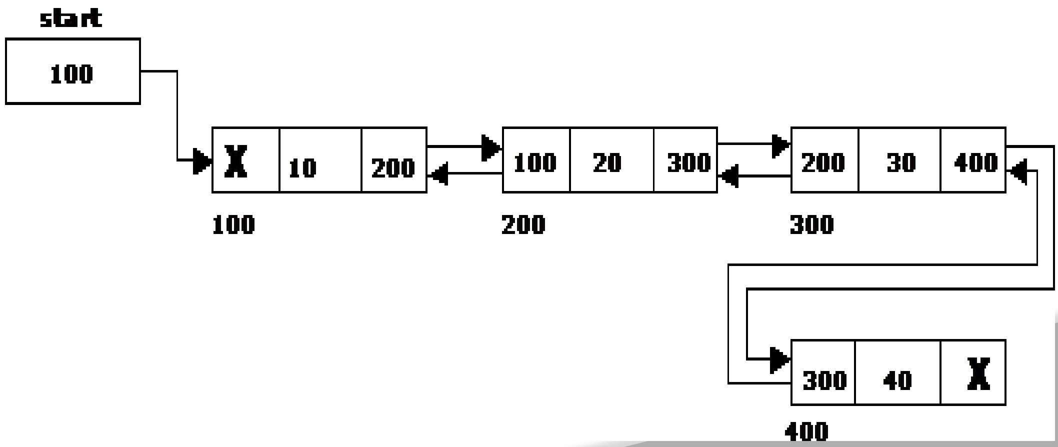


Inserting a node at the ending

- The following steps are followed to insert a new node at the end of the list:

```
def insertend(self):
    data=int(input("enter value"))
    newnode=Node(data)
    if self.start==None:
        self.start=newnode
    else:
        temp=self.start
        while temp.next!=None:
            temp=temp.next
        temp.next=newnode
        newnode.prev=temp
```





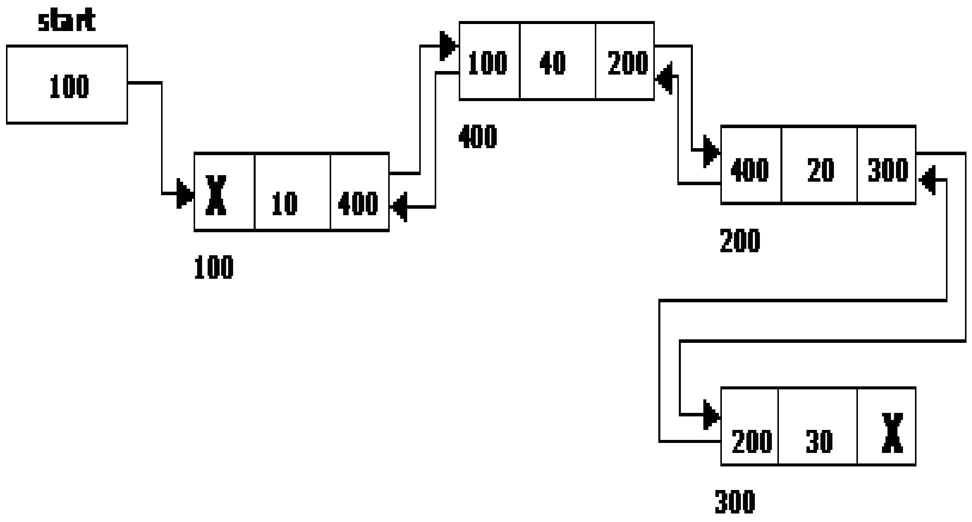
Inserting a node at an intermediate position

- The following steps are followed, to insert a new node in an intermediate position in the list:

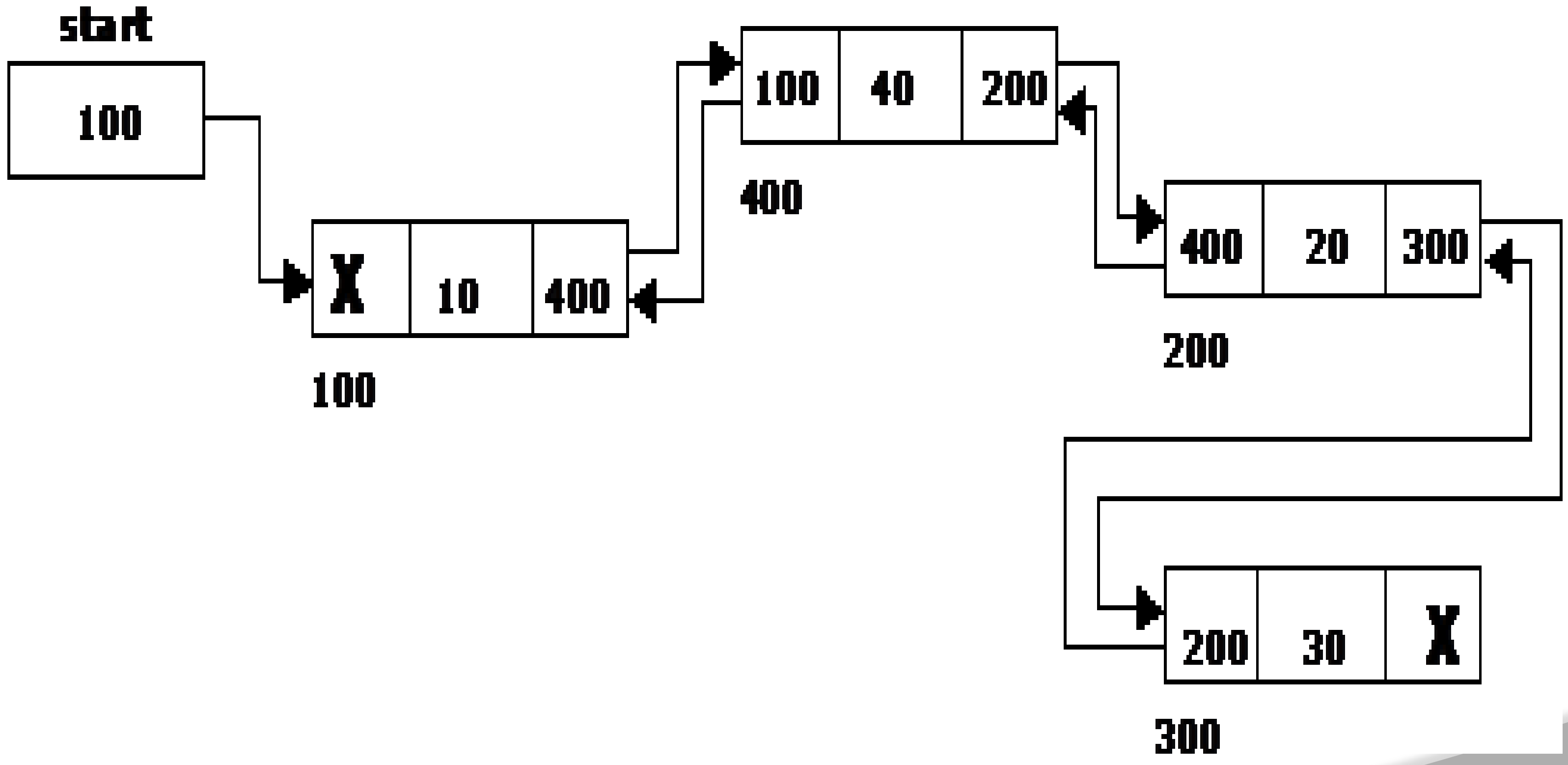
```
def insertmid(self):
    print("enter data before which number is to be inserted")
    num=int(input())
    temp=self.start
    ptemp=temp
    if self.start==None:
        print("List is empty")
    elif num==temp.data:
        self.insertbegin()
```

Inserting a node at an intermediate position Cont..

```
else:
        data=int(input("enter value"))
        newnode=Node(data)
        temp=self.start

        while temp.data!=num:

            temp=temp.next
        newnode.next=temp
        newnode.prev=temp.prev
        temp.prev.next=newnode
```



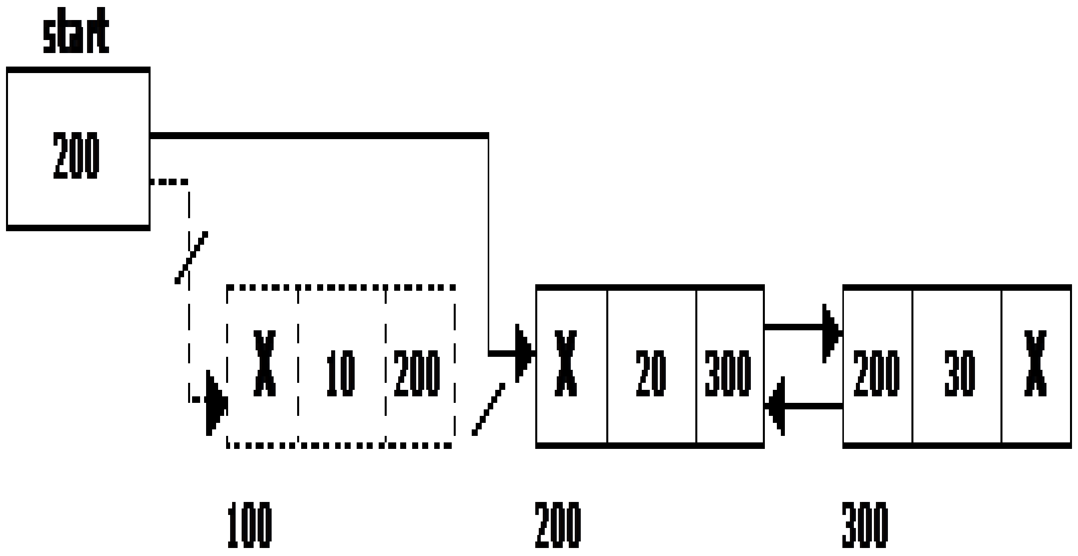
Inserting a node at an intermediate position



Deleting a node at the beginning

- The following steps are followed, to delete a node at the beginning of the list:

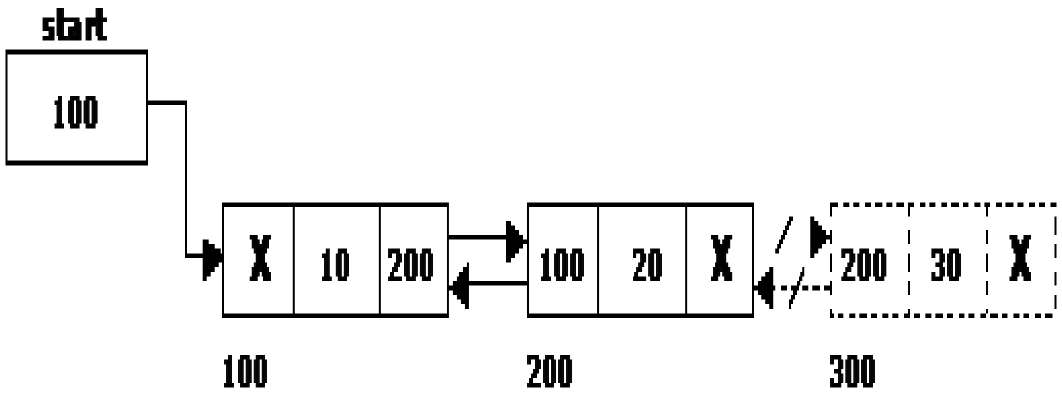
```
def deletebegin(self):
        temp=self.start
        if self.start==None:
            print("List is empty")
        else:
            self.start=temp.next
            self.start.prev=None
            print(temp.data,"is deleted success")
            del temp
```



Deleting a node at the end

- The following steps are followed to delete a node at the end of the list:

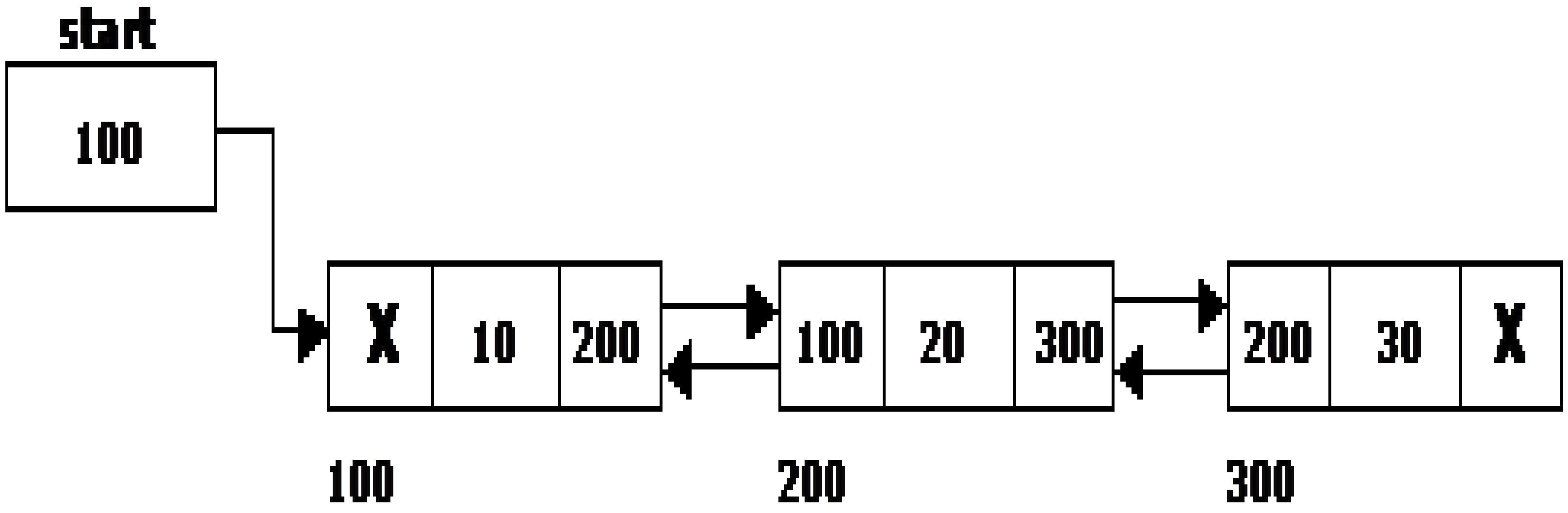
```
def deletend(self):
        temp=self.start
        if self.start==None:
            print("List is empty")

        else:
            ptemp=temp
            temp=temp.next
            while temp.next!=None:
                ptemp=temp
                temp=temp.next
            ptemp.next=None
            print(temp.data,"is deleted success")
            del temp
```



Deleting a specified node

```
def deletion(self):
      print("enter element to delete")
      a=int(input())
      temp=self.start
      if self.start==None:
         print("List is empty, deletion not possible")
         return

      else:
         temp=self.start

         while temp.next!=None and temp.data!=a:
            temp=temp.next

         temp.prev.next=temp.next
         temp.next.prev=temp.prev
         print(temp.data,"is deleted success")
         del temp
```

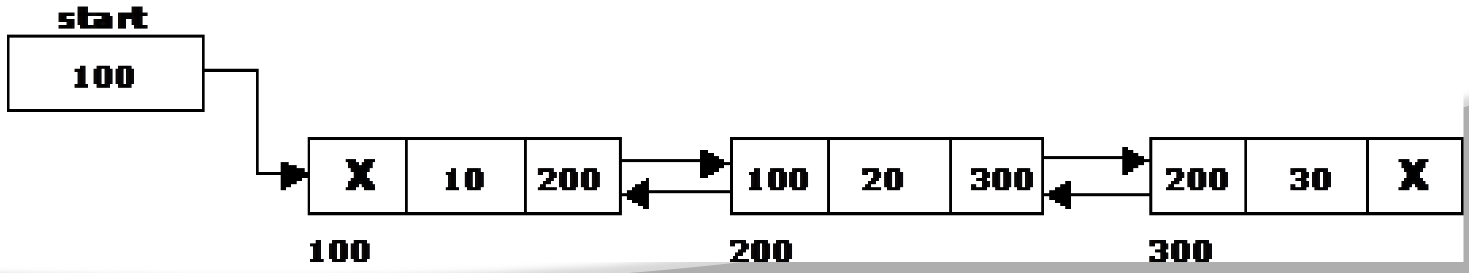
Deleting a specified node



Traversal and displaying a list (Left to Right)

- **The following steps are followed, to traverse a list from left to right:**

```
def display(self):
        print("elements in Doubly linked list are:")
        if self.start==None:
            print("Linked list is empty")
        else:
            temp=self.start
            while temp!=None:
                print(temp.data ,end=" ")
                temp=temp.next
```



Applications of Double linked list

Doubly linked list can be used in

- navigation systems where both front and back navigation is required.
- It is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward button.

Advantages and Disadvantages of Double Linked List

The major **disadvantage** of doubly linked lists (over singly linked lists) is that

- they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major **advantage** of doubly linked lists is that

- they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

Advantages of Circular Lists

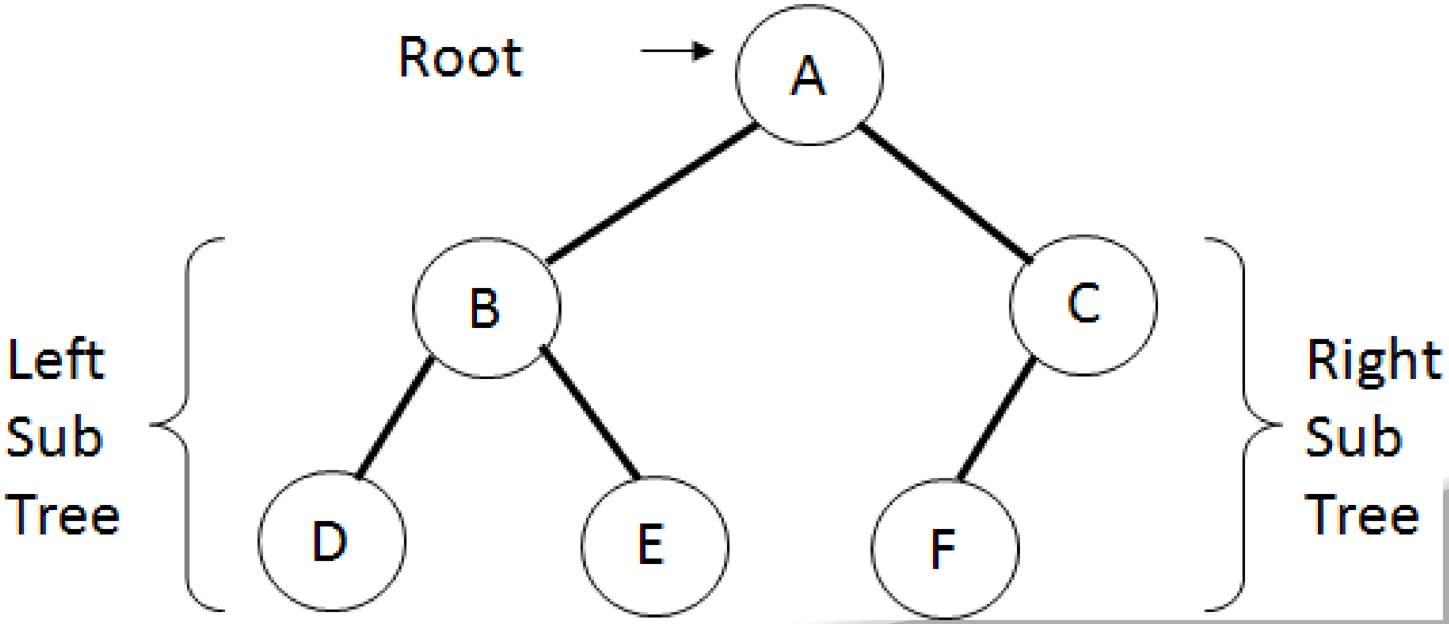
- The major **advantage** of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node).
- Also, some applications lead naturally to circular list representations.
- For example, a computer network might best be modeled using a circular list.

MODULE - IV

Trees: Basic concept, binary tree, binary tree representation, array and linked representations, binary tree traversal, binary tree variants, application of trees; Graphs: Basic concept, graph terminology, graph implementation, graph traversals, Application of graphs.

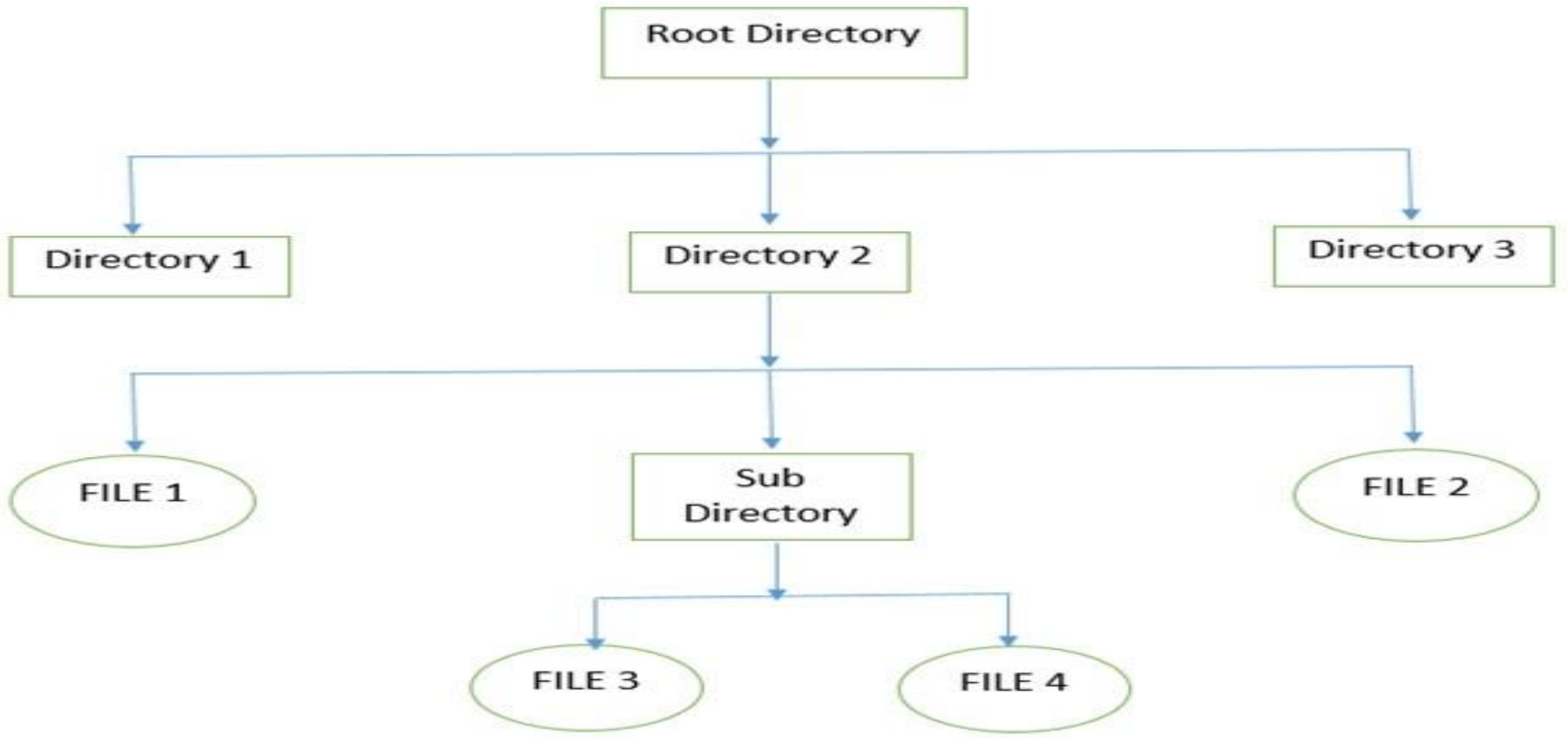
Tree – a Hierarchical Data Structure

- Trees are non linear data structure that can be represented in a hierarchical manner.
  - A tree contains a finite non-empty set of elements.
  - Any two nodes in the tree are connected with a relationship of parent-child.
  - Every individual elements in a tree can have any number of sub trees.



Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



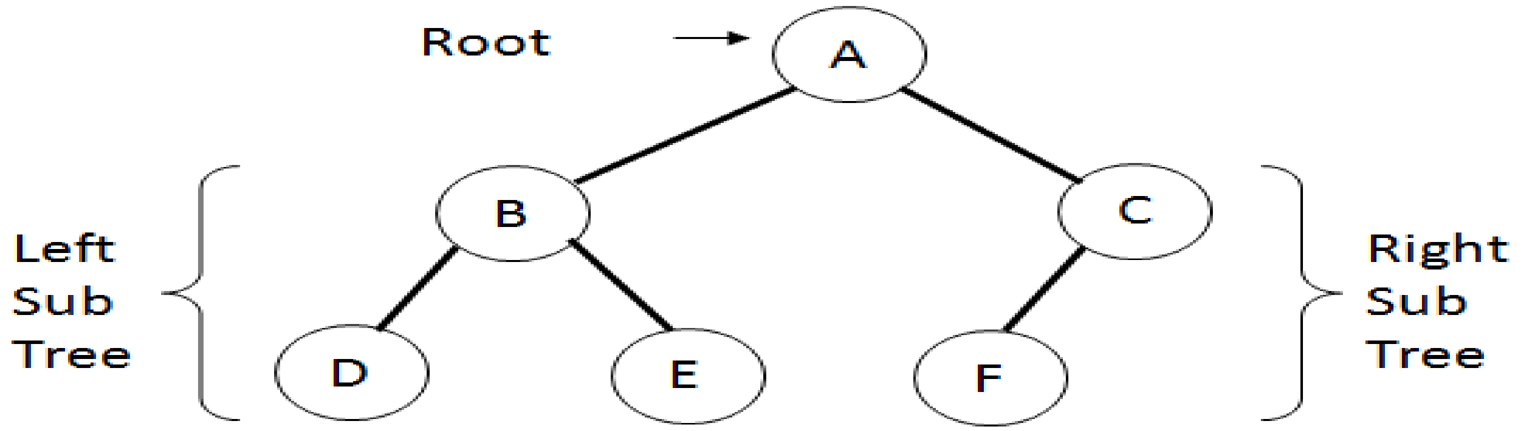
Why Trees?

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

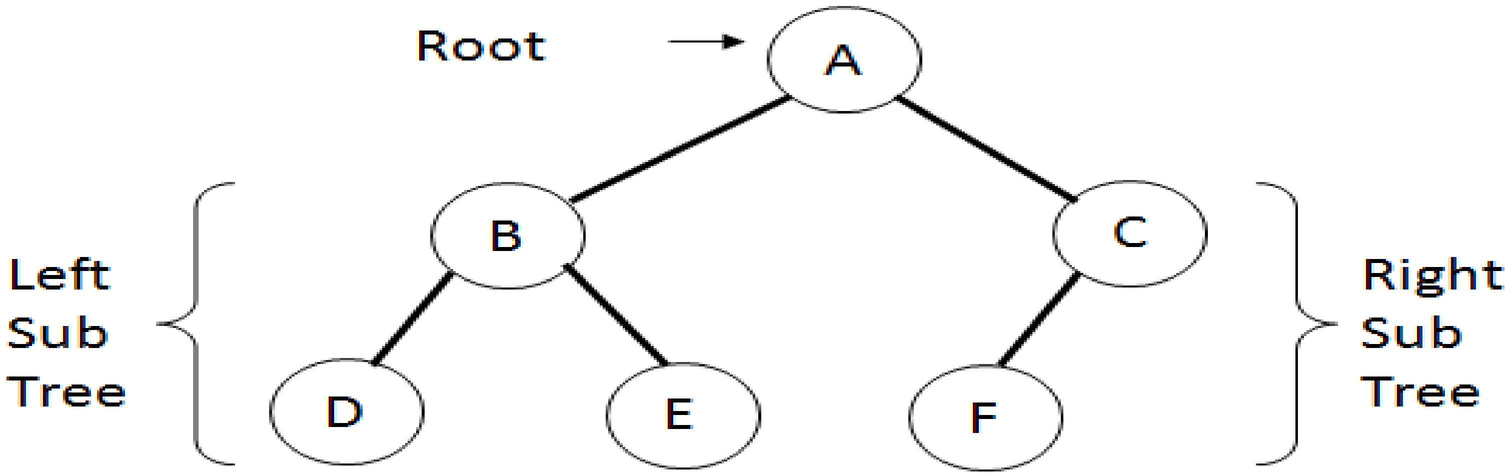
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

4. Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

Tree – Basic Terminology

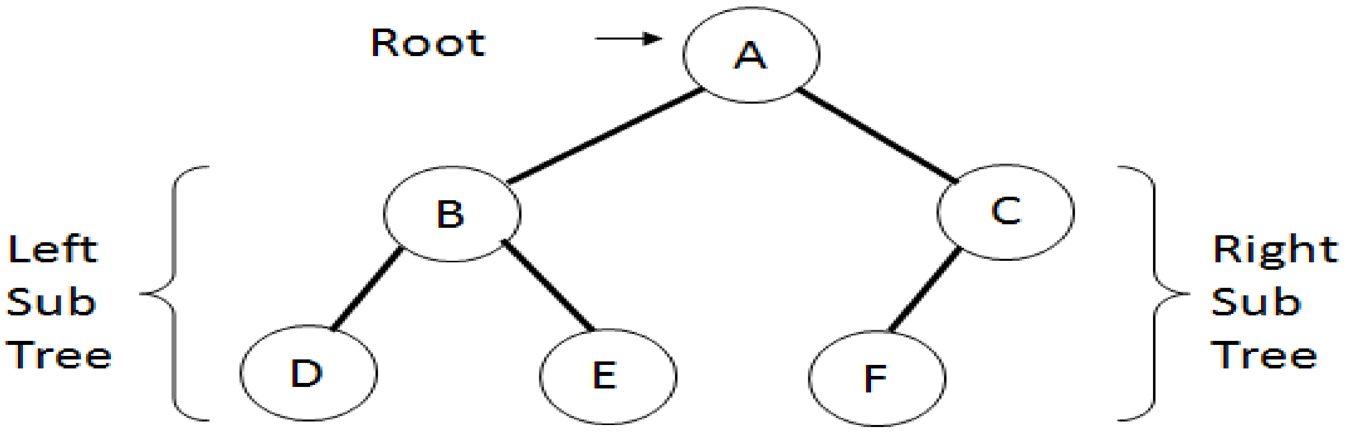


- **Root :** The topmost node is called root of the tree. The basic node of all nodes in the tree. All operations on the tree are performed with passing root node to the functions.
- **Child :** a successor node connected to a node is called child. A node in binary tree may have at most two children(or)The elements that are directly under an element are called its children



- **Parent :** a node is said to be parent node to all its child nodes (or) The element directly above something is called its parent.
- **Leaf :** a node that has no child nodes.
- **Siblings :** Two nodes are siblings if they are children to the same parent node.

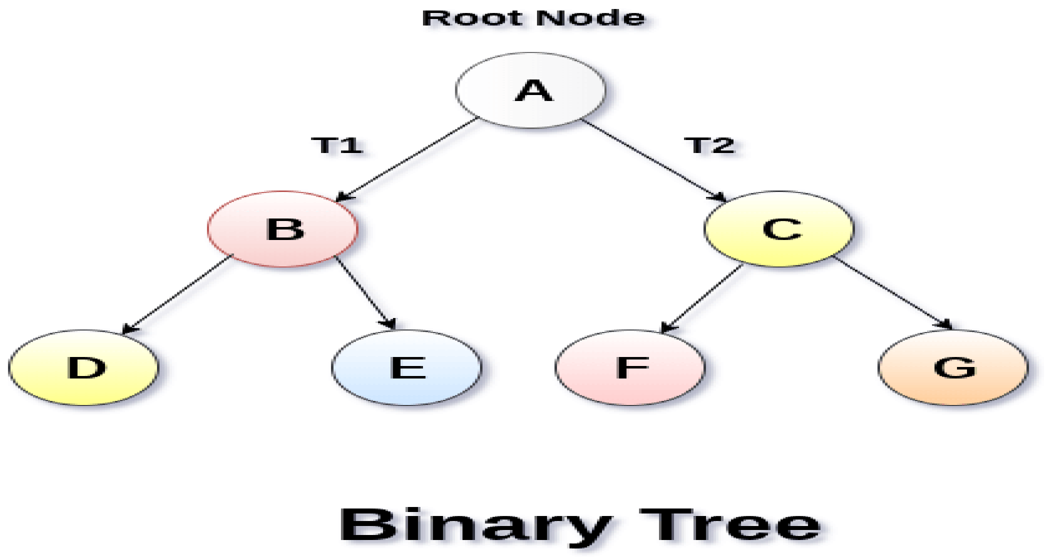
Tree – Basic Terminology Contd...



- **Ancestor :** a node which is parent of parent node (A is ancestor node to D,E and F).
- **Descendent :** a node which is child of child node (D, E and F are descendent nodes of node A)
- **Level :** The distance of a node from the root node, The root is at level – 0,(B and C are at Level 1 and D, E, F have Level 2 (highest level of tree is called height of tree)
- **Degree :** The number of nodes connected to a particular parent node.

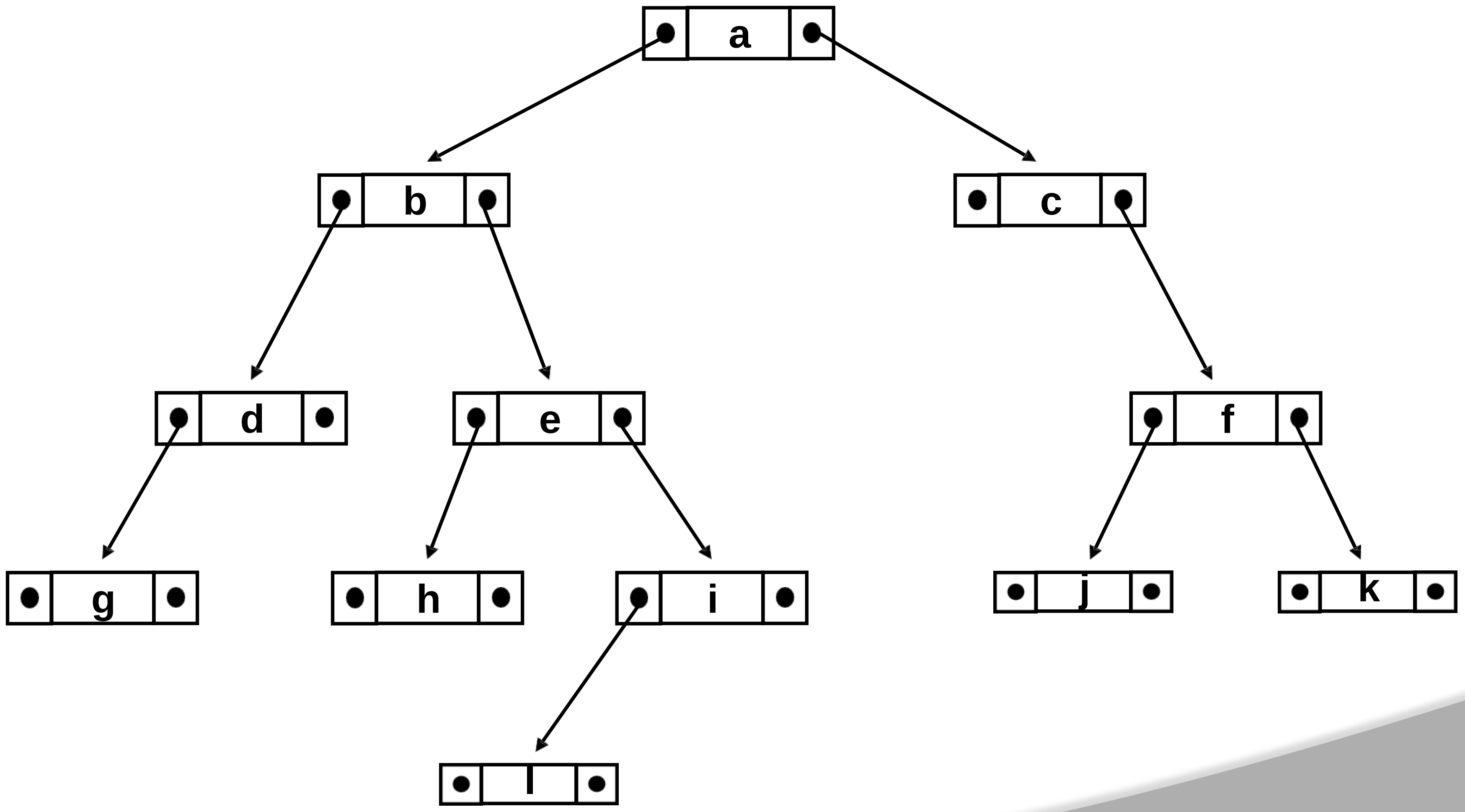
Binary Tree

- A binary tree is a hierarchy of nodes, where every parent node has at most two child nodes. There is a unique node, called the root, that does not have a parent.



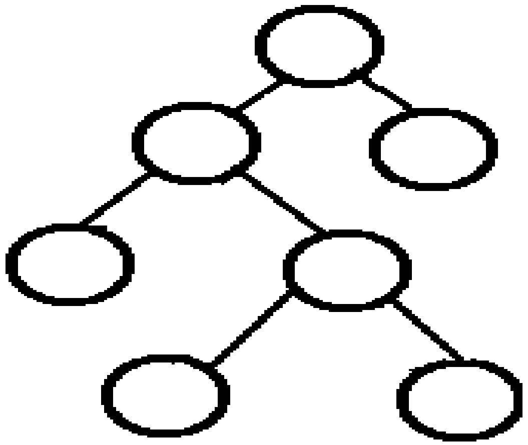
- A binary tree can be defined recursively as
- Root node
- Left subtree: left child and all its descendants
- Right subtree: right child and all its descendants

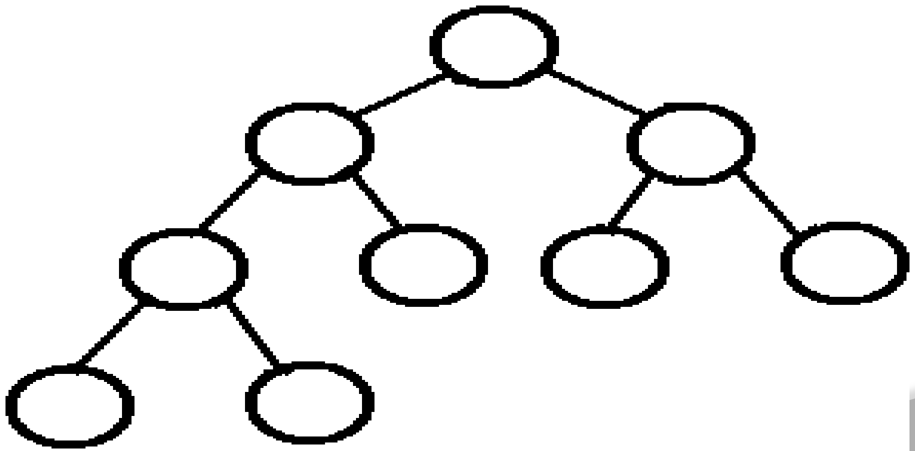
Binary Tree



Full and Complete Binary Trees

- A full tree is a binary tree in which
  - Number of nodes at level *l is 2l–1*
  - Total nodes in a full tree of height *n is*
- A complete tree of height *n is a binary tree*
  - Number of nodes at level 1 *l* *n–1 is 2l–1*
  - Leaf nodes at level *n occupy the* leftmost positions in the tree


full tree


complete tree

Tree Traversals

- **A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree.**
- **To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.**
- **Tree traversals are naturally recursive.**
- **Standard traversal orderings:**
  - **preorder**
  - **inorder**
  - **postorder**
  - **level-order**

Preorder, Inorder, Postorder

- **In Preorder, the root is visited before (pre)**

  The subtrees traversals.
- **In Inorder, the root is visited in-between left and right subtree traversal.**
- **In Preorder, the root is visited after (pre) the subtrees traversals.**

<u>Preorder Traversal</u>:
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

<u>Inorder Traversal</u>:
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

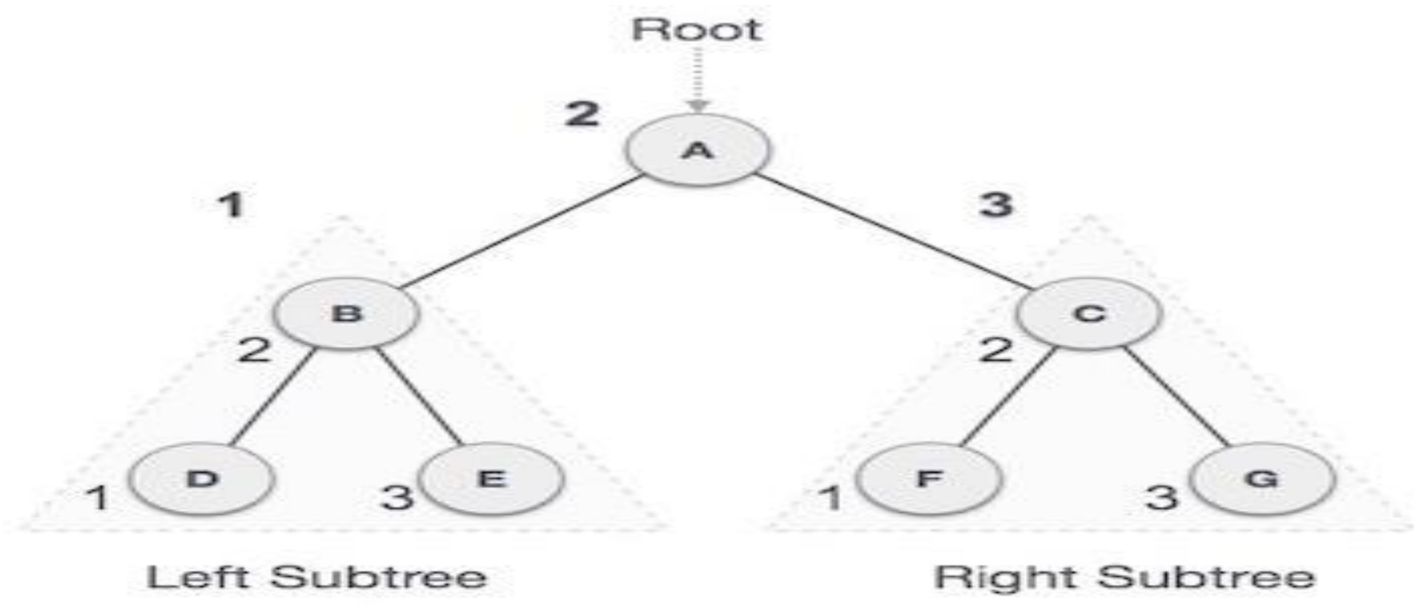
<u>Postorder Traversal</u>:
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

In-order Traversal



D → B → E → A → F → C → G

In-order Traversal

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step3 − Recursively traverse right subtree.

Construct a Binary Tree from Postorder and Inorder

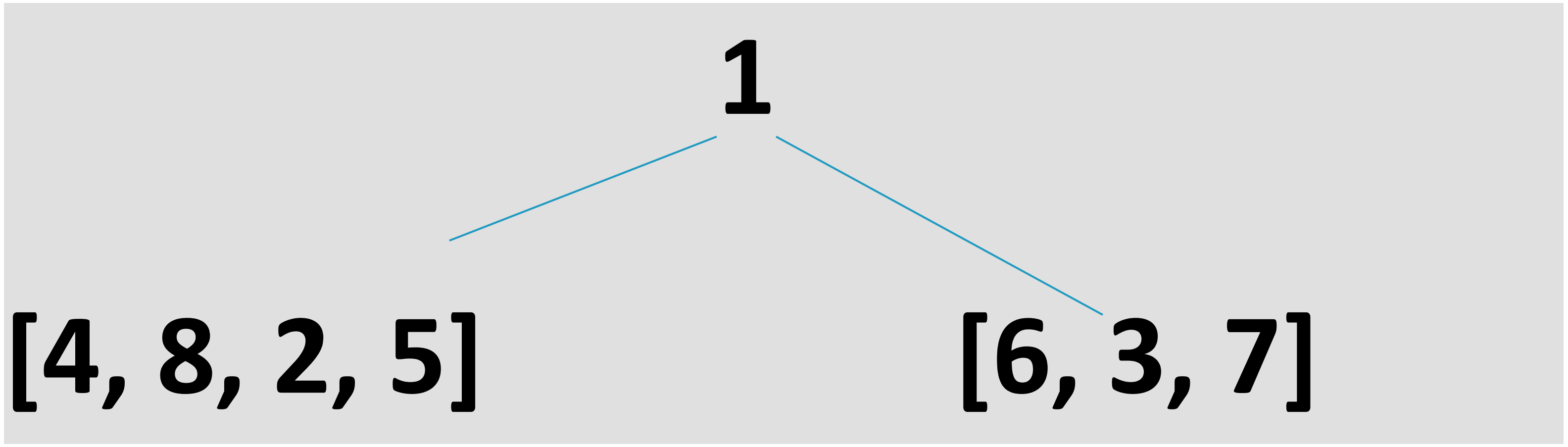
Given Postorder and Inorder traversals, construct the tree.

```
in[] = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}
```

the process of constructing tree from
in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}

1) We first find the last node in post[]. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.

2) We search “1” in in[] to find left and right subtrees of root. Everything on left of “1” in in[] is in left subtree and everything on right is in right subtree.



3) We recur the above process for following two.
….b) Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}
…….Make the created tree as right child of root.
….a) Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.
…….Make the created tree as left child of root.

```
    1
   / \
  2   3
 /\  /\
4  5 6 7
 \
  8
```

Applications of trees

- **Represent organization**
- **Represent computer file systems**
- **Networks to find best path in the Internet**
- **Chemical formulas representation**
- **Outlines, etc**

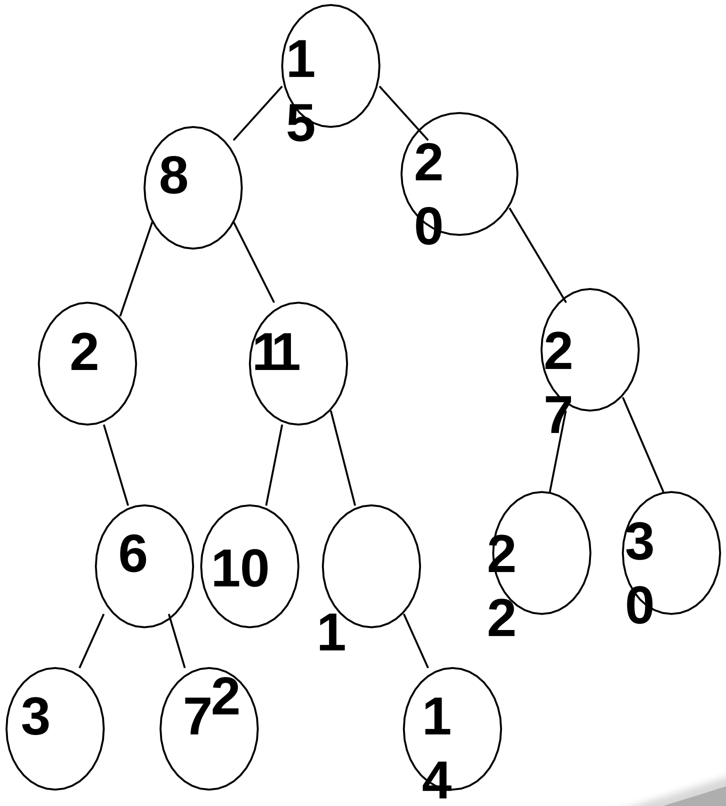
Example of Tree Traversal

- Assume: visiting a node printing its data

•Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30

•Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30

•Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15



Traversal Techniques

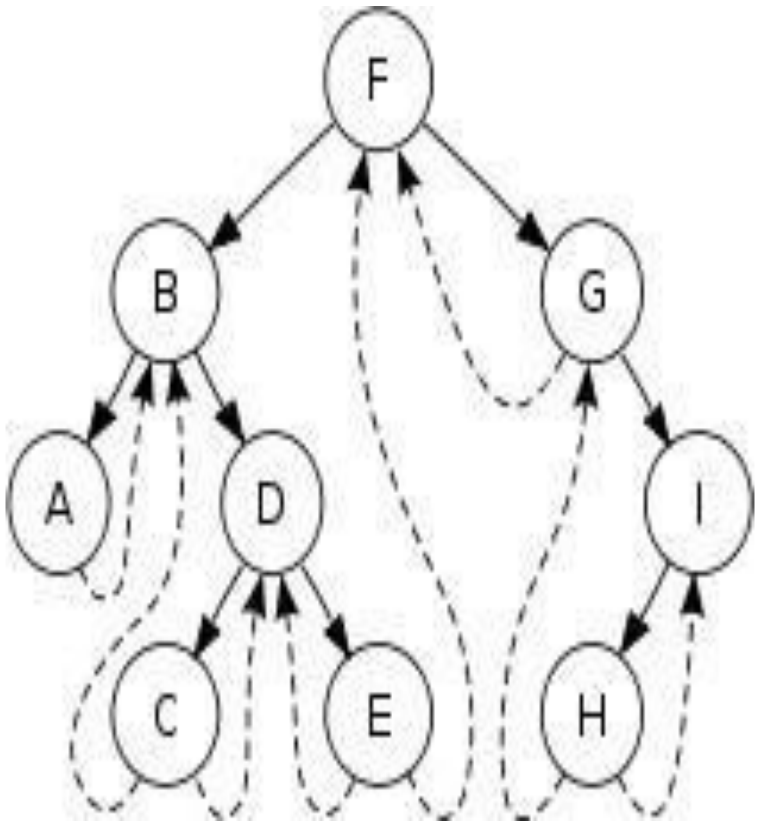
```
void preorder(tree *tree) {
if (tree->isEmpty( ))    return;  visit(tree->getRoot( ));  preOrder(tree->getLeftSubtree());  preOrder(tree->getRightSubtree());
}
```

```
void inOrder(Tree *tree){
if (tree->isEmpty( ))        return;  inOrder(tree->getLeftSubtree( ));
visit(tree->getRoot( ));  inOrder(tree->getRightSubtree( ));
}
```

```
void postOrder(Tree *tree){
if (tree->isEmpty( ))        return;  postOrder(tree->getLeftSubtree( ));
postOrder(tree->getRightSubtree( ));  visit(tree->getRoot( ));
}
```

Threaded Binary Tree

- A threaded binary tree defined as:
- "A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node



Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

in a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences.
By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree.
So we know below structure now.

```
          A
        /   \
       /      \
  D B E        F C
```

We recursively follow above steps and get the following tree.

```
        A
      /   \
     B     C
    / \   /
   D   E F
```

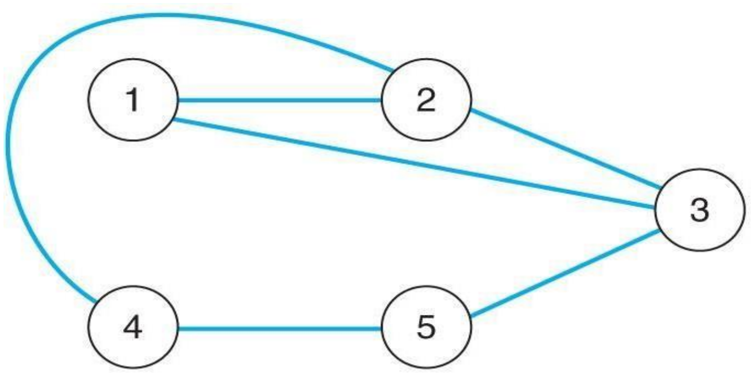
Graph Basics

- Graphs are collections of nodes connected by edges – G = (V,E) where V is a set of nodes and E a set of edges.
- Graphs are useful in a number of applications including
  - Shortest path problems
  - Maximum flow problems
- Graphs unlike trees are more general for they can have connected components.

Graph Types

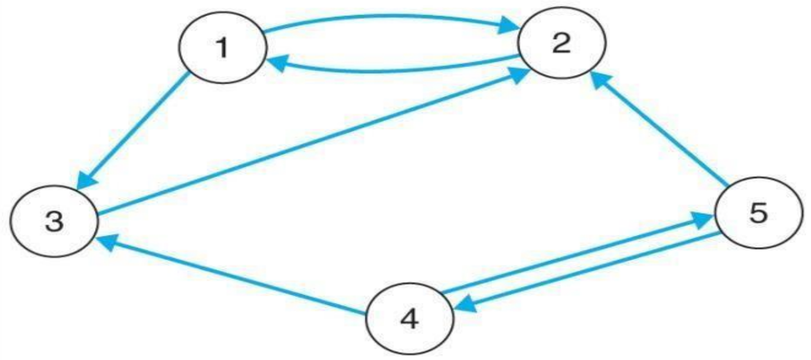
Directed Graphs: A directed graph edges allow travel in one direction.

- Undirected Graphs: An undirected graph edges allow travel in either direction.



■ FIGURE 8.1A

The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})



■ FIGURE 8.1B

The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

Graph Terminology

- A graph is an ordered pair G=(V,E) with a set of vertices or nodes and the edges that connect them.
- A subgraph of a graph has a subset of the vertices and edges.
- The edges indicate how we can move through the graph.
- A path is a subset of E that is a series of edges between two nodes.
- A graph is connected if there is at least one path between every pair of nodes.

Graph Terminology

- The length of a path in a graph is the number of edges in the path.
- A complete graph is one that has an edge between every pair of nodes.
- A weighted graph is one where each edge has a cost for traveling between the nodes.
- A cycle is a path that begins and ends at the same node.
- An acyclic graph is one that has no cycles.
- An acyclic, connected graph is also called an unrooted tree

Data Structures for Graphs An Adjacency Matrix

- For an undirected graph, the matrix will be symmetric along the diagonal.
- For a weighted graph, the adjacency matrix would have the weight for edges in the graph, zeros along the diagonal, and infinity (∞) every place else.

Adjacency Matrix Example 1

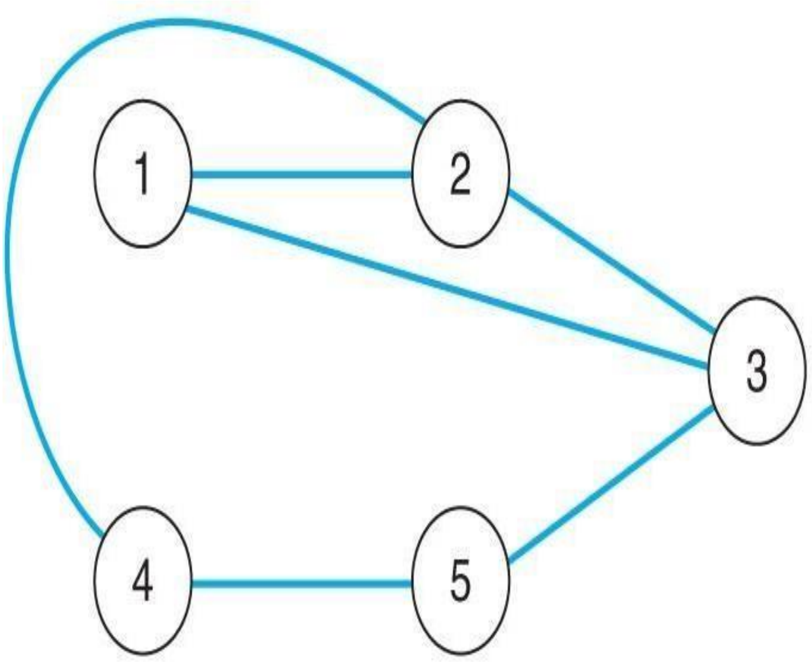


FIGURE 8.1A
The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

$$\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 \\
4 & 0 & 1 & 0 & 0 & 1 \\
5 & 0 & 0 & 1 & 1 & 0
\end{array}$$

FIGURE 8.2A
The adjacency matrix for the graph in Fig. 8.1(a)

Adjacency Matrix Example 2

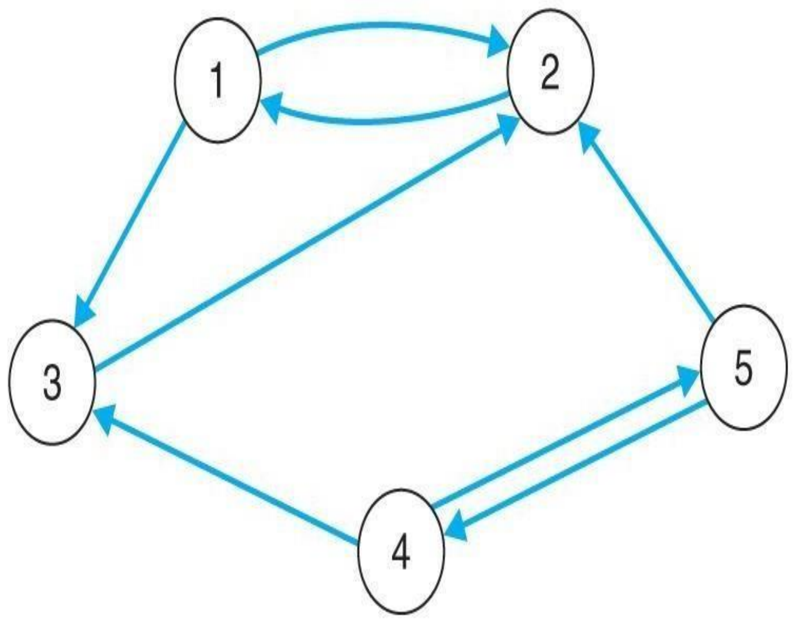


FIGURE 8.1B

The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

Data Structures for Graphs An Adjacency List

- A list of pointers, one for each node of the graph.
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph.
- For a weighted graph, this list would also include the weight for each edge.

Adjacency List Example 1

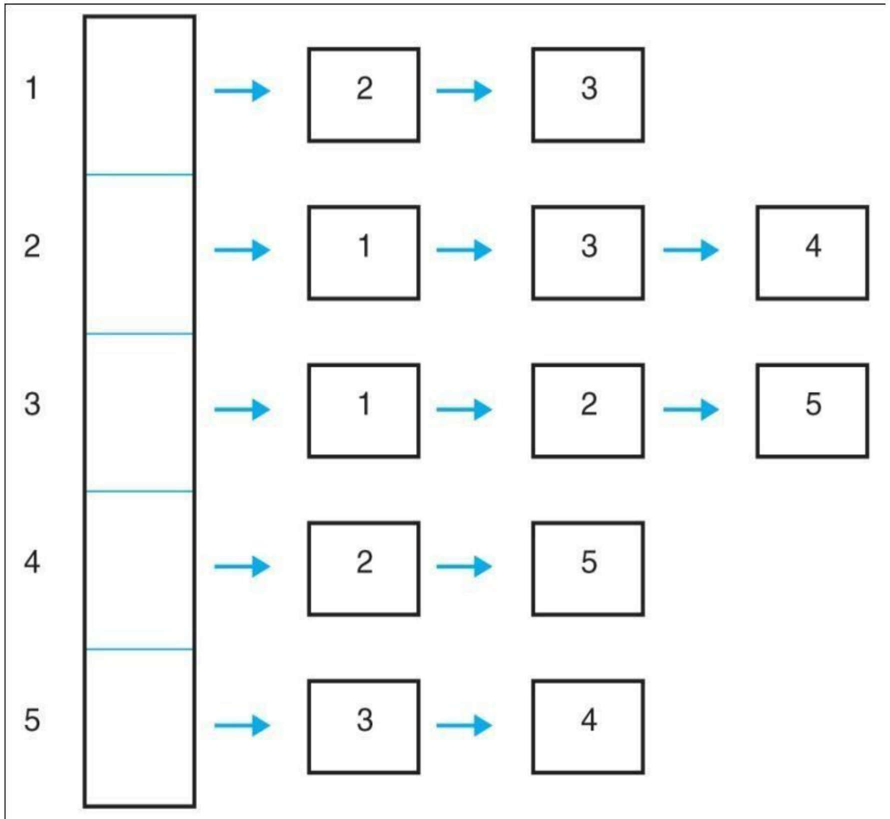


FIGURE 8.3A
The adjacency list for the graph in Fig. 8.1(a)

Adjacency List Example 2

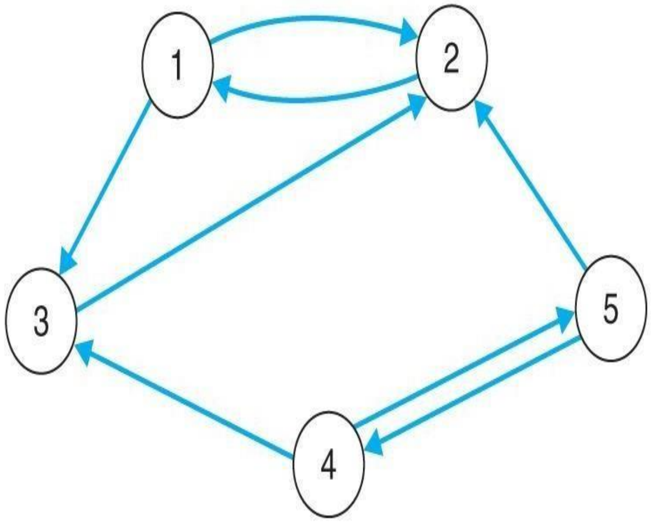


FIGURE 8.1B
The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

The image part with relationship ID rId3 was not found in the file.

Graph Traversals

- Some algorithms require that every vertex of a graph be visited exactly once.
- The order in which the vertices are visited may be important, and may depend upon the particular algorithm.
- The two common traversals:
  - depth-first
  - breadth-first

Graph Traversals: Depth First Search Traversal

- **We follow a path through the graph until we reach a dead end.**
- **We then back up until we reach a node with an edge to an unvisited node.**
- **We take this edge and again follow it until we reach a dead end.**
- **This process continues until we back up to the starting node and it has no edges to unvisited nodes.**

Breadth First Search Traversal

- **From the starting node, we follow all paths of length one.**
- **Then we follow paths of length two that go to unvisited nodes.**
- **We continue increasing the length of the paths until there are no unvisited nodes along any of the paths.**

Breadth First Search Traversal Example

- Consider the following graph:

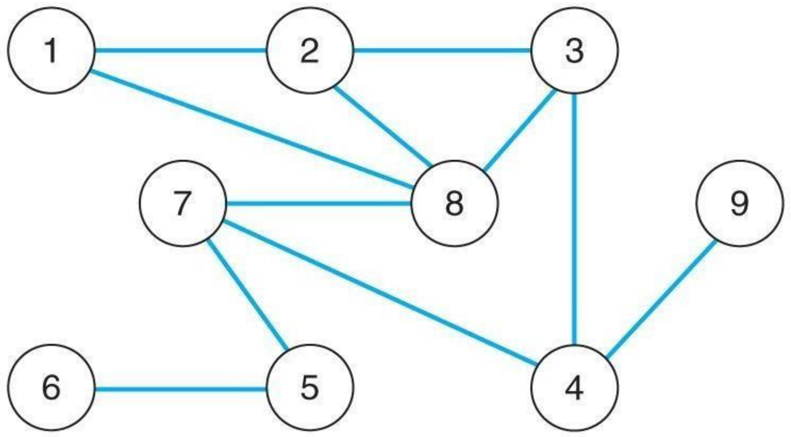


FIGURE 8.4
A graph

- The order of the breadth-first traversal of this graph starting at node 1 would be: 1, 2, 8, 3, 7, 4, 5, 9, 6

MODULE -V

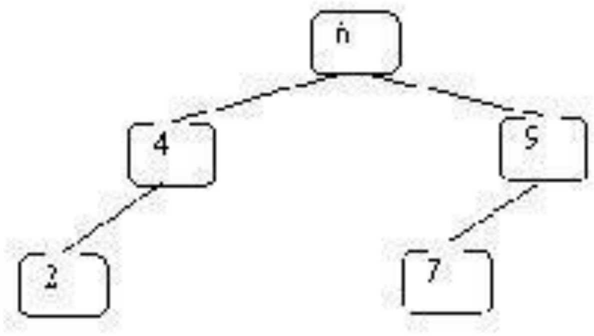
Binary search trees: Binary search trees, properties and operations; Balanced search trees: AVL trees; Introduction to M-Way search trees, B trees; Hashing and collision: Introduction, hash tables, hash functions, collisions, applications of hashing.

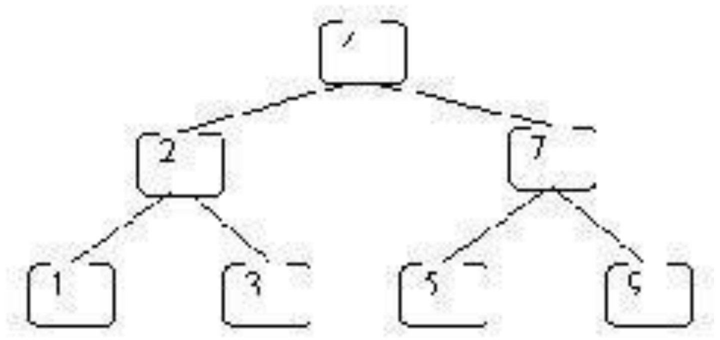
Binary Search Trees

- In a BST, each node stores some information including a unique key value, and perhaps some associated data. A binary tree is a BST iff, for every node n in the tree:
- All keys in n's left subtree are less than the key in n, and
- All keys in n's right subtree are greater than the key in n.
- In other words, binary search trees are binary trees in which all values in the node‘s left subtree are less than node value all values in the node‘s right subtree are greater than node value.

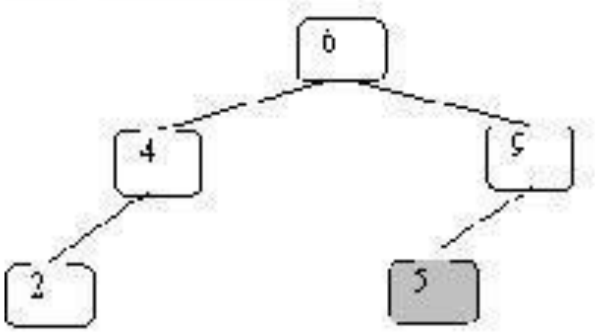
BST Example

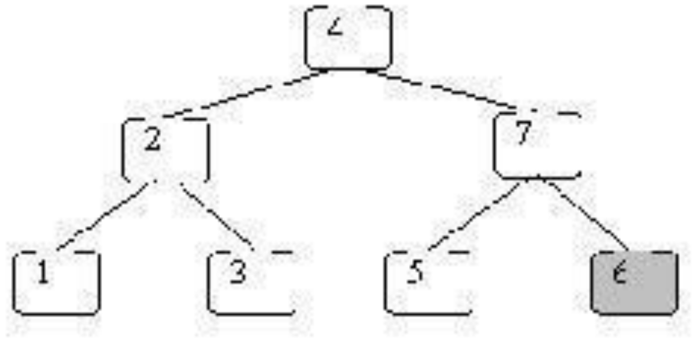
Here are some BSTs in which each node just stores an integer key:





These are not BSTs:





In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

Properties and Operations

A BST is a binary tree of nodes ordered inthe following way:

i. Each node contains one key (also unique)

ii. The keys in the left subtree are < (less) than the key in its parent node

iii. The keys in the right subtree > (greater) than the key in its parent node

iv. Duplicate node keys are not allowed.

Operations - Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root.

We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node.

Operations - Inserting a node

- A recursive algorithm for inserting a node into a BST is as follows.Assume we insert a node N to tree T.

if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

```
Insert(N, T)          = N        if T is empty
= insert(N, T.left)                  if    N < T
= insert(N, T.right)          if             N > T
```

Operations - Searching for a node

- Searching for a node is similar to inserting a node. Westart from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

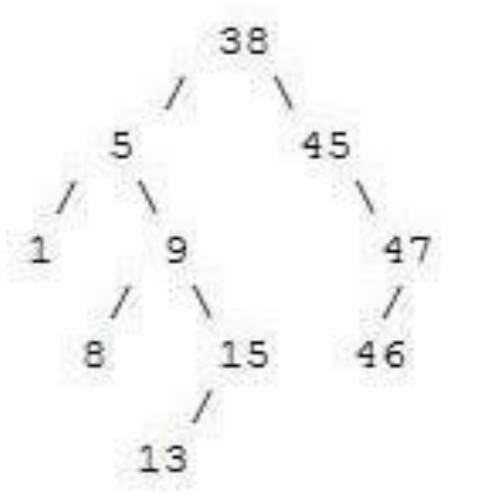
Searching for a node

- Search(N, T) = false if T is empty Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).
- A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N >
T.A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

```
Search(N, T) =    false    if T is empty
               =  true     if T = N
               = search(N, T.left) if N < T
```

Operations - Deleting a node

- A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



Balanced Search Trees

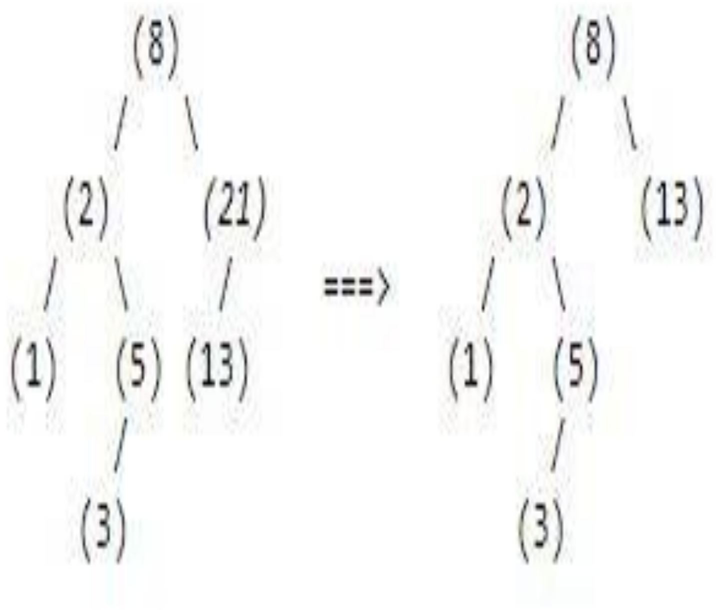
- A self-balancing (or height-balanced) binary search tree is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.
- AVL Trees: An AVL tree is another balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an *O(*log*n)* search time. Addition and deletion operations also take *O(*log*n)* time.

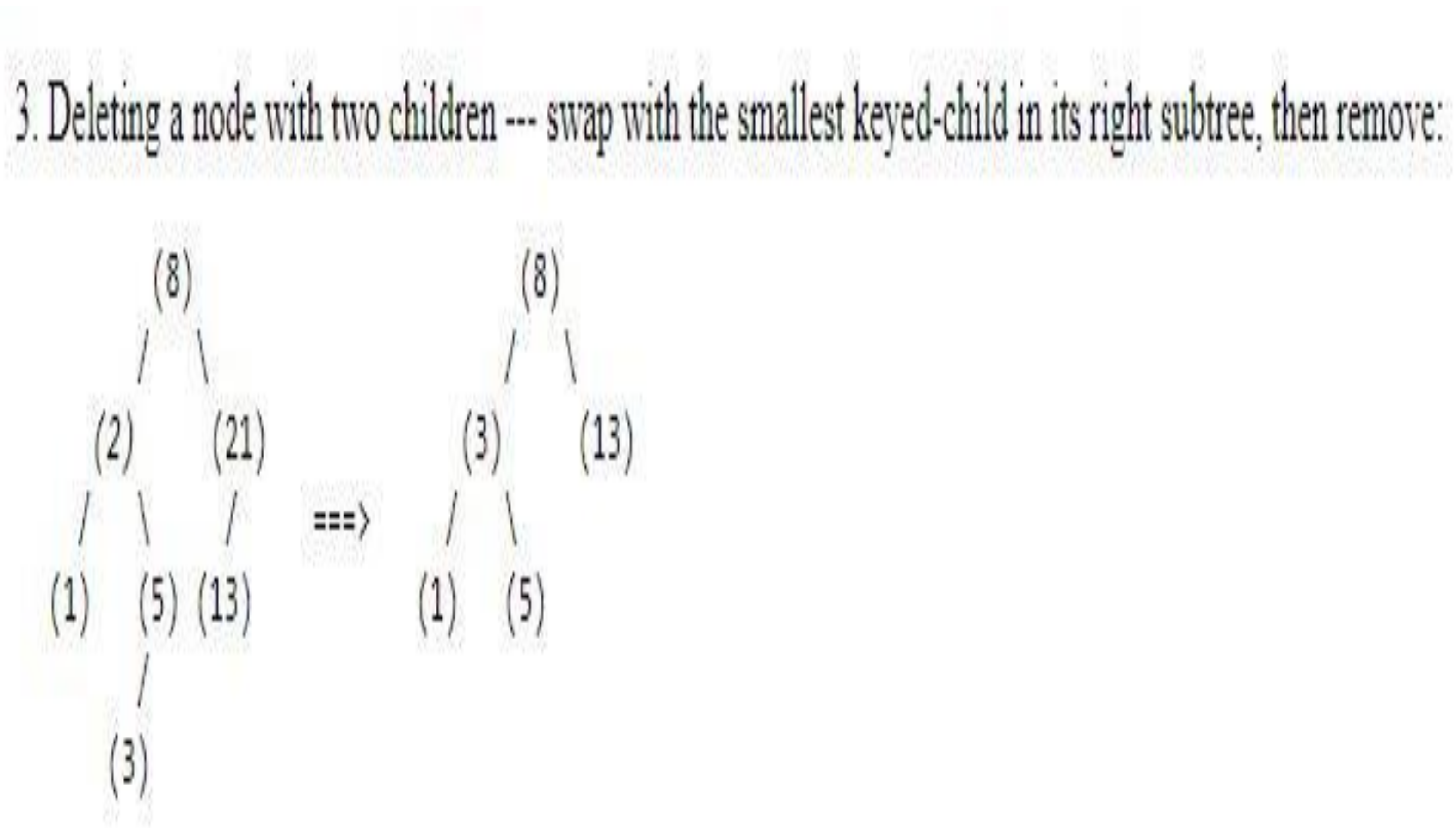
There are three cases we need to consider for deletion:

1. Deleting a leaf --- simply remove it:

```
          (8)                        (8)
         /   \                      /   \
      (2)     (21)               (2)     (21)
     /   \    /       ===>      /   \
  (1)    (5) (13)            (1)    (5)
         /                          /
      (3)                        (3)
```

2. Deleting a node with one child --- remove it and move its child (the subtree rooted at its child) up:

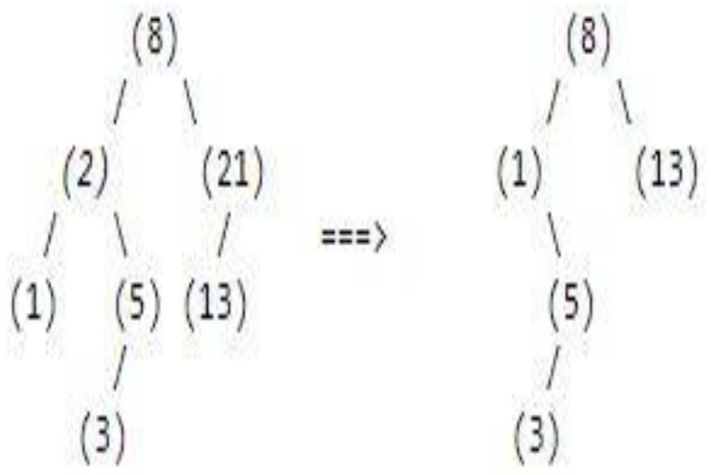




3. Deleting a node with two children --- swap with the smallest keyed-child in its right subtree, then remove:

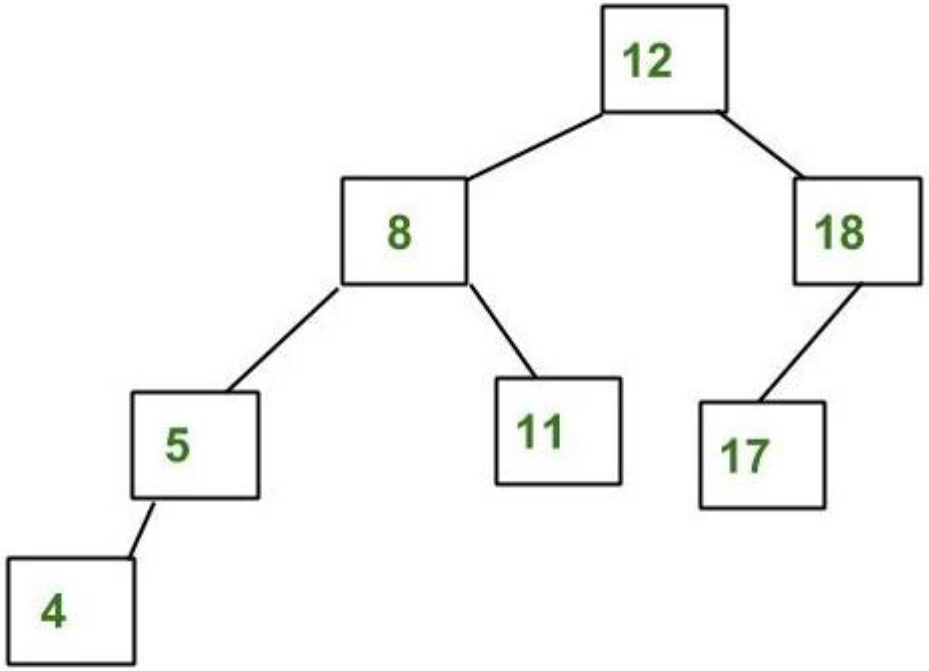
```
          (8)                    (8)
          / \                    / \
      (2)     (21)           (3)     (13)
      / \      /    ===>     / \
   (1)  (5) (13)          (1)   (5)
        /
      (3)
```

or swap with the largest keyed-child in its left subtree, then remove:

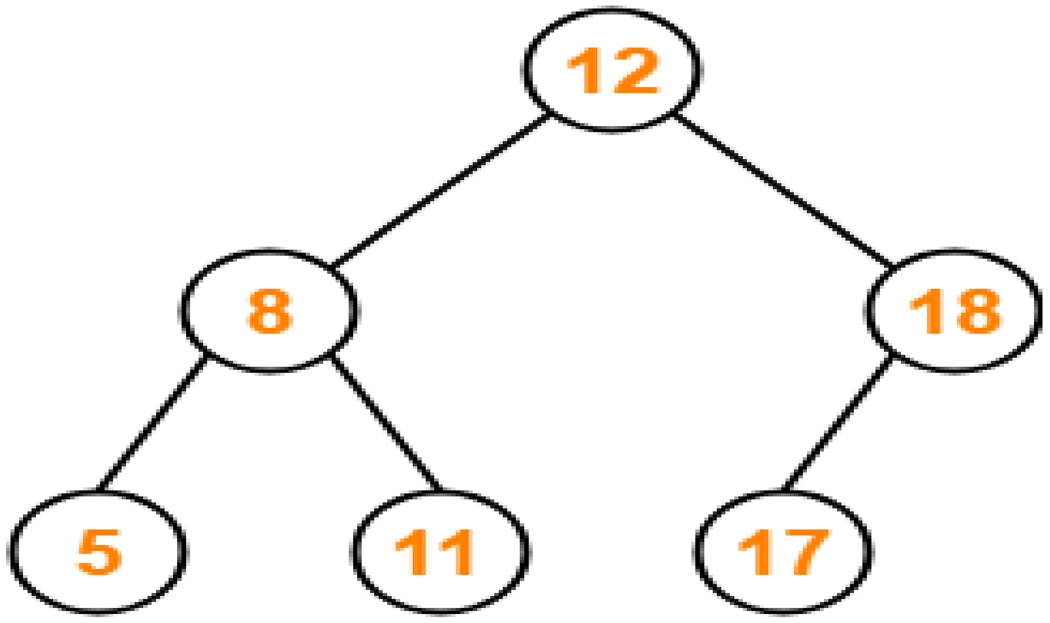


AVL Tree - Definition

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.



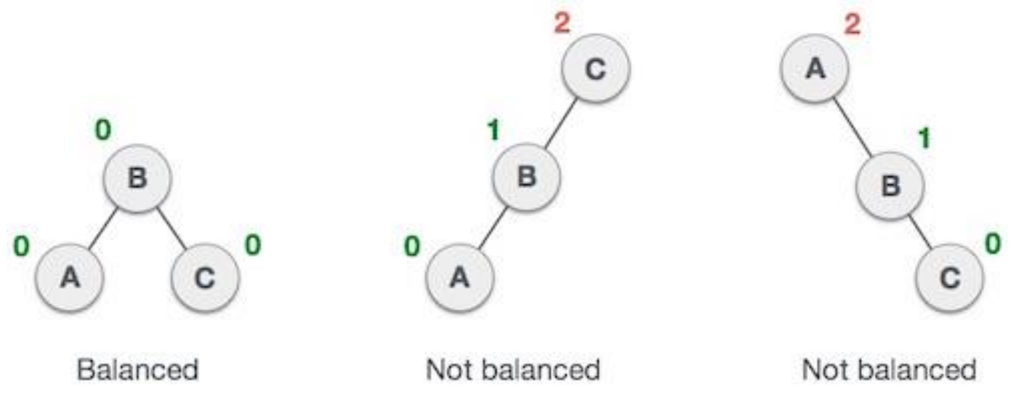
Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees**



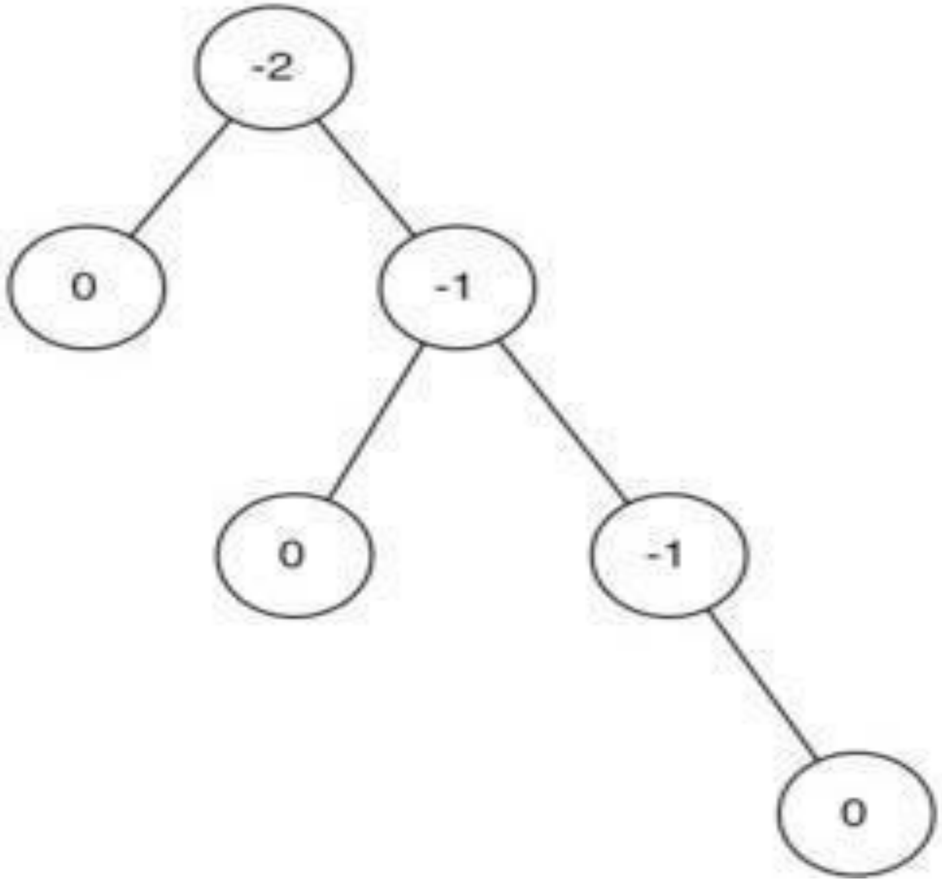
AVL Tree Example

AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$



Balance Factor



AVL Tree Operations-

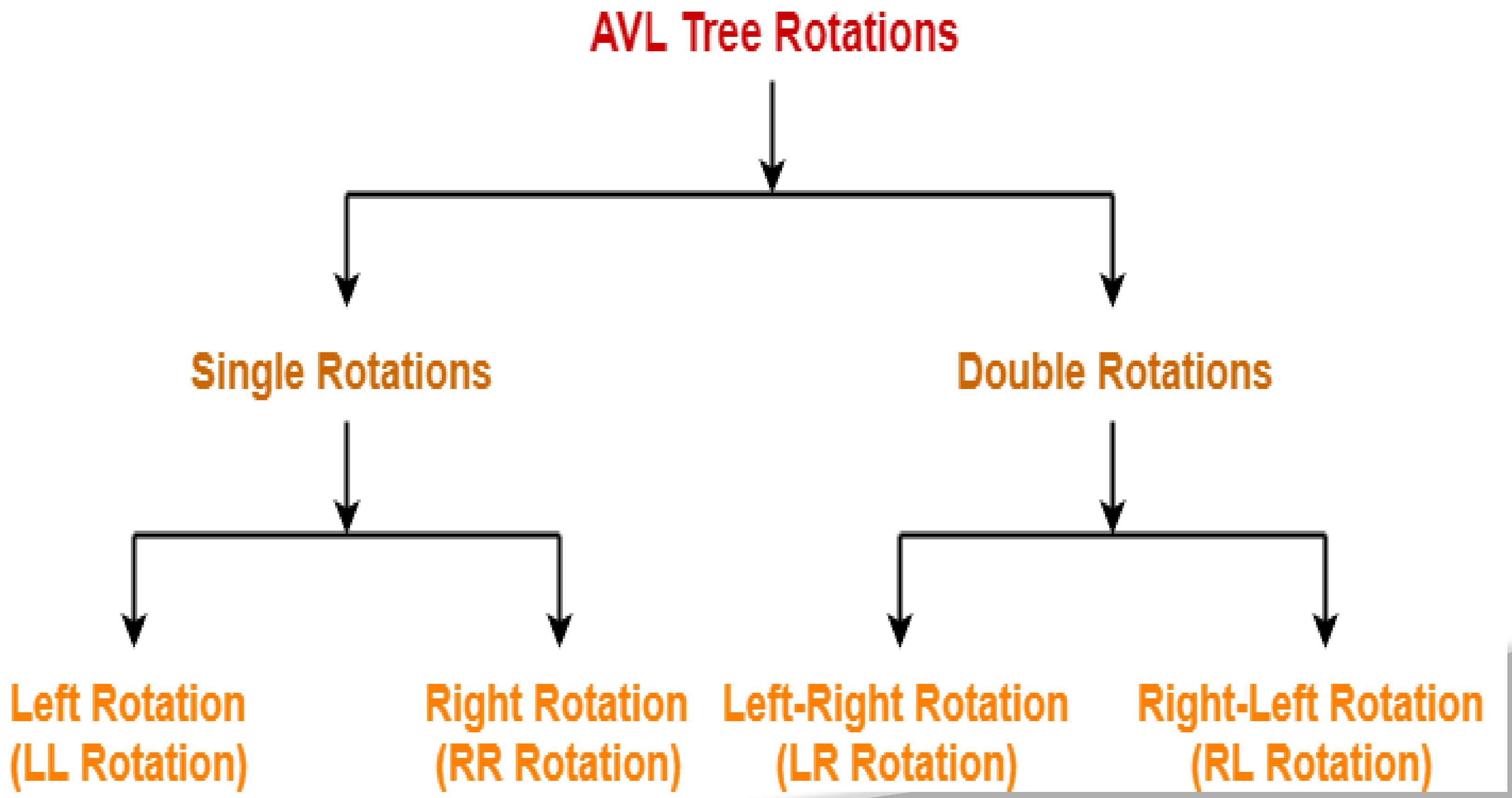
Search Operation

Insertion Operation

Deletion Operation

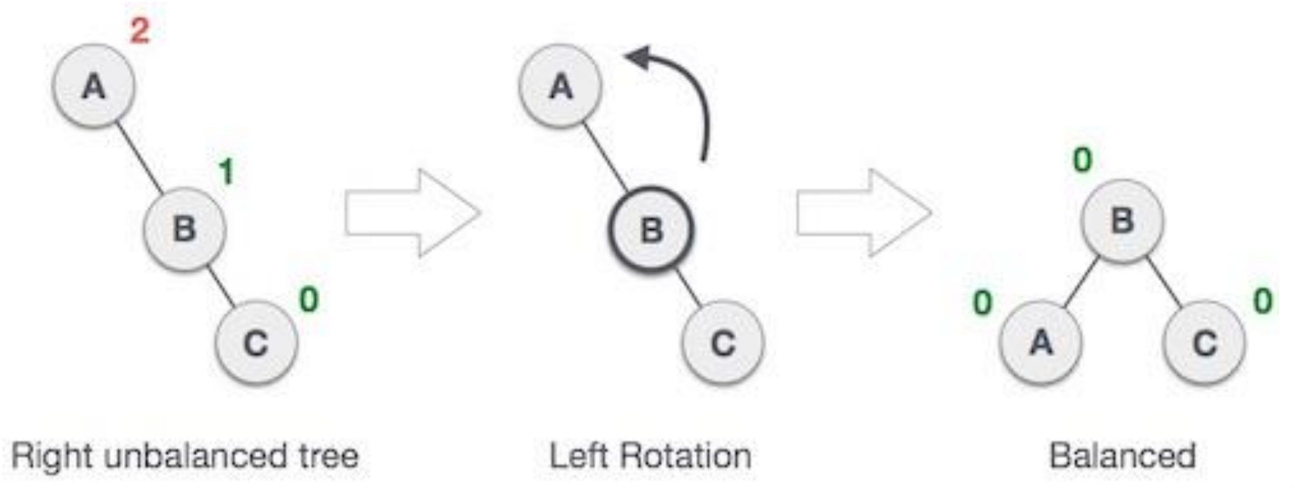
Kinds of Rotations

There are 4 kinds of rotations possible in AVL Trees-


AVL Tree Rotations
Single Rotations
Double Rotations
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left-Right Rotation (LR Rotation)
Right-Left Rotation (RL Rotation)

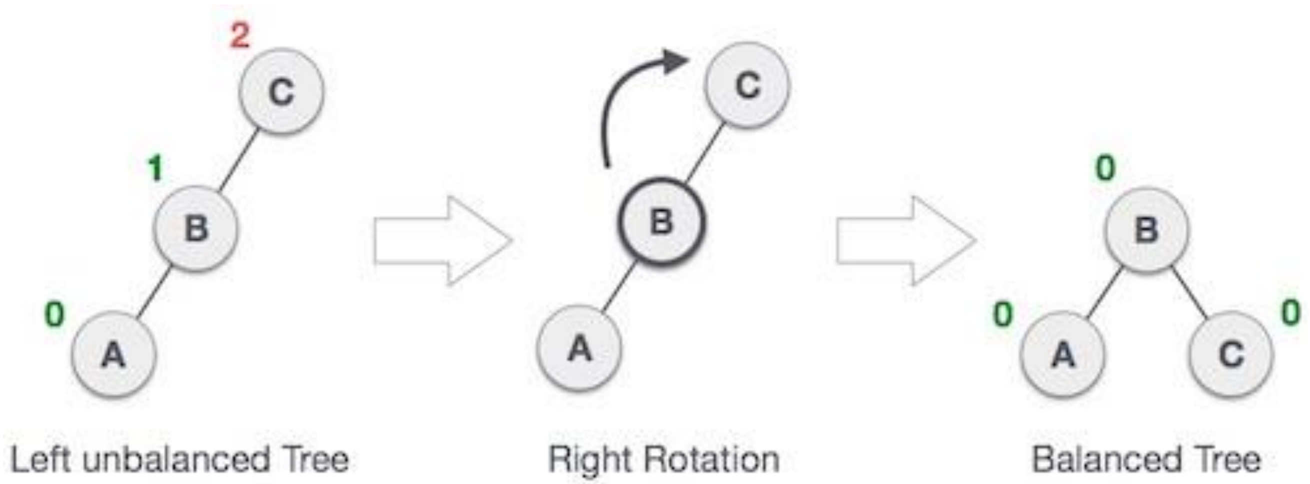
Right Rotation

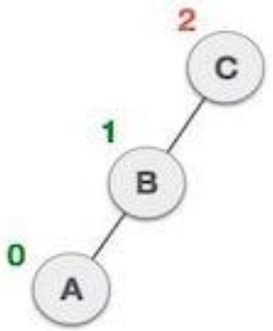
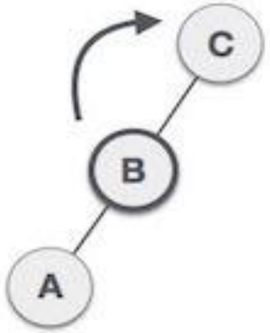
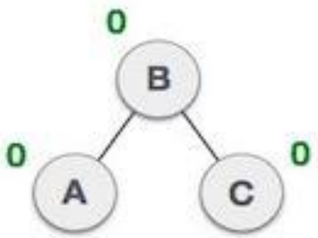
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation



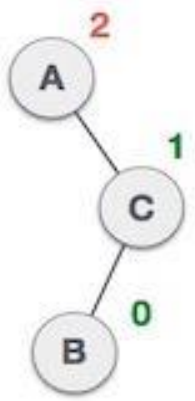
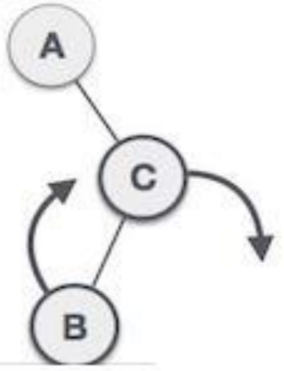
Left Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.

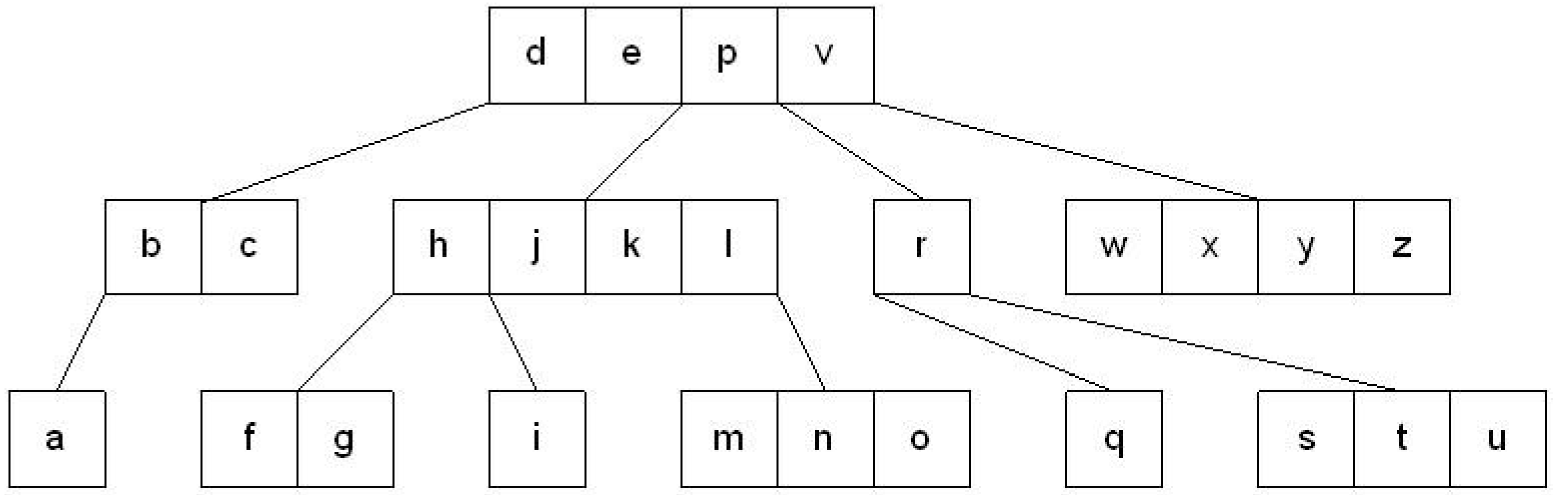
EXAMPLE:
CONSTRUCT AN AVL WITH THE GIVEN VALUES

63,9,19,27,18,108,99,81

Introduction to M-Way Search Trees

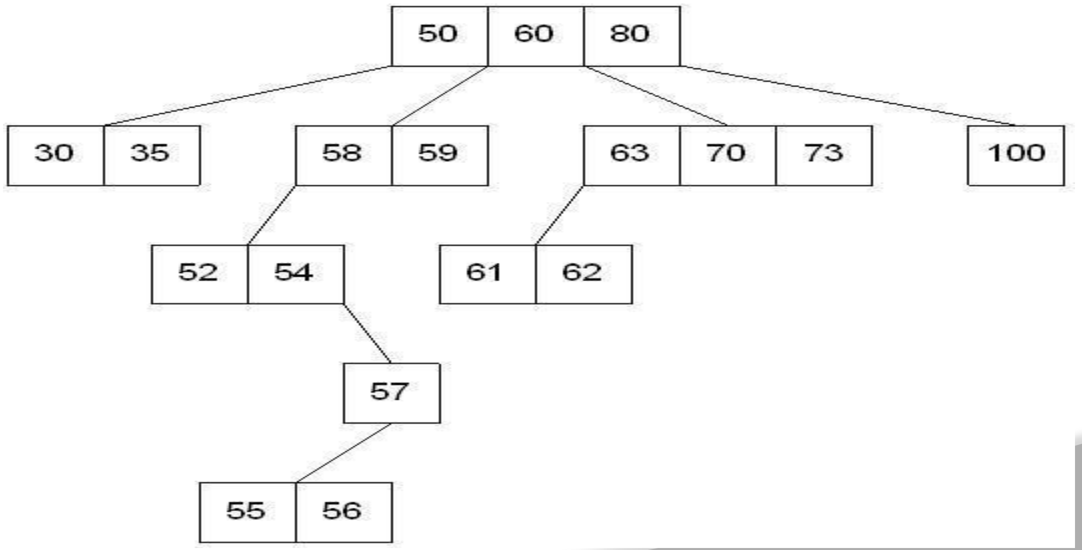
- A multiway tree is a tree that can have more than two children. A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.

- Example:Multi way tree of order 5



Properties of M-way Search Trees

- m-way search tree is a m-way tree in which:
  1. Each node has m children and m-1 key fields
  2. The keys in each node are in ascending order.
  3. The keys in the first i children are smaller than the ithkey
  4. The keys in the last m-i children are larger than theith key
- 4-way search tree



B -Trees

- A B-tree is a self-balancing or perfectly height-balanced M-way search tree.
that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.

- The B-tree is a generalization of a binary search tree in that a node can have more than two children.

- Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs.

- It is commonly used in databases and file systems.

Properties of a B-Tree:

i. It is perfectly height-balanced and therefore every leaf node is at the same depth.

ii. Every internal node, except the root, is at least half-full i.e contains ceil(M/2) or more children.

iii. Every leaf node must contain ceil(M/2)-1 keys, where ceil(x) is the ceiling function.

iv. The root may have any number of value(1 to M-1) when M is the degree of the tree.

iv. Every leaf node must contain ceil(M/2)-1 keys, where ceil(x) is the ceiling function.

Searching a B -Tree

- Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node.
- Follow the appropriate pointer to a child node.
- Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

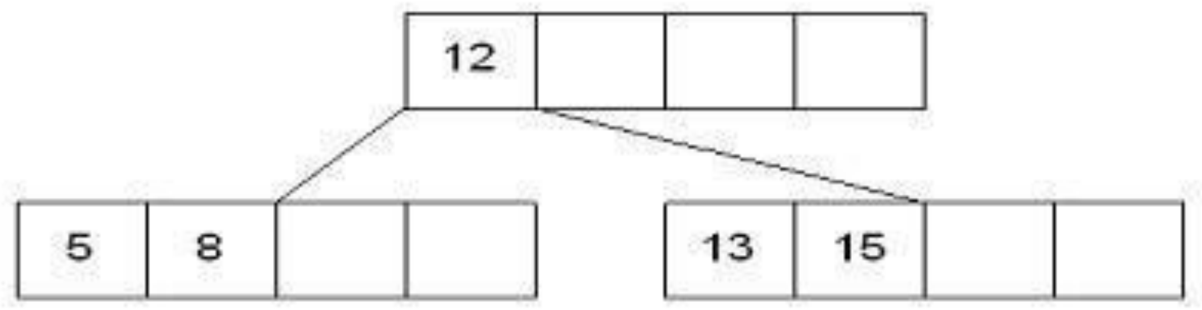
Insertion into a B-Tree

- The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.
- When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:
  - i. A key is placed into a leaf that still has room.
  - ii. The leaf in which a key is to be placed isfull.
  - iii. The root of the B-tree is full.

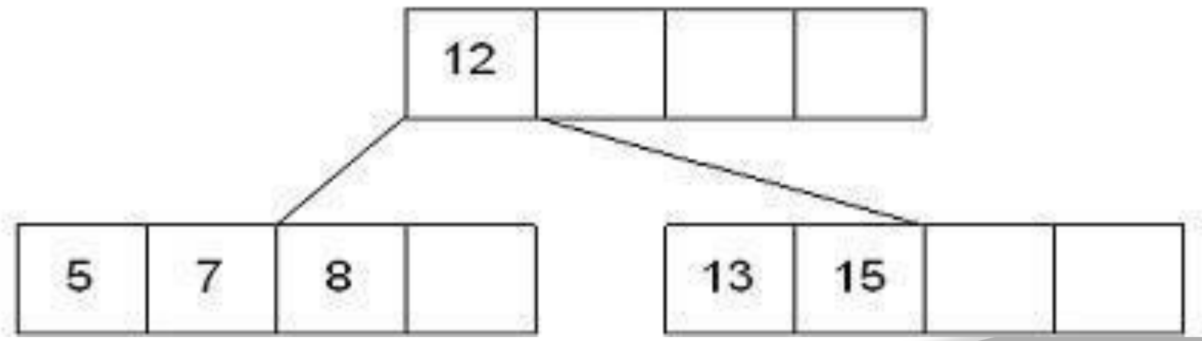
Insertion into a B-Tree

Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:



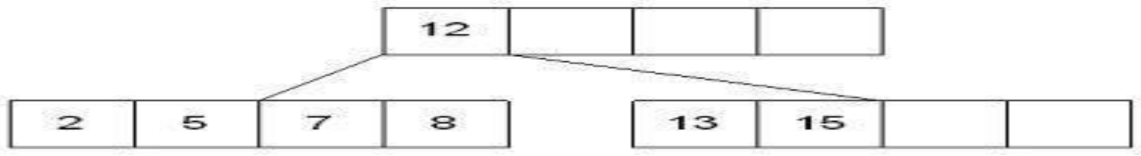
Insertion into a B-Tree

Case 2: The leaf in which a key is to be placed is full

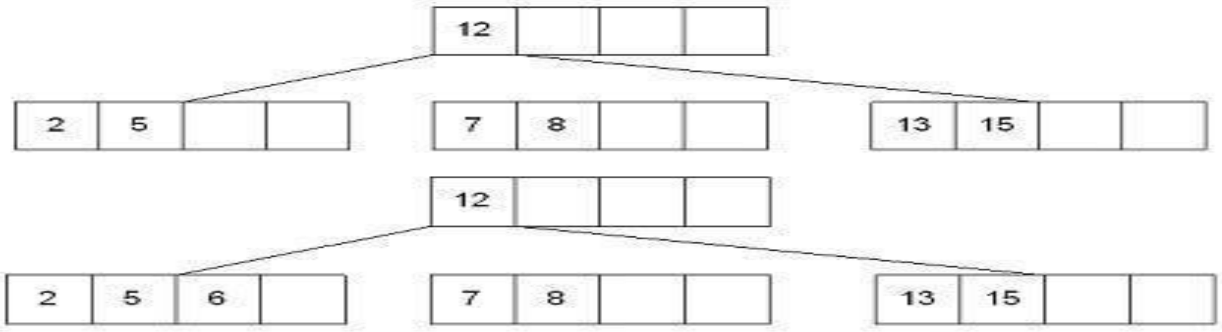
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

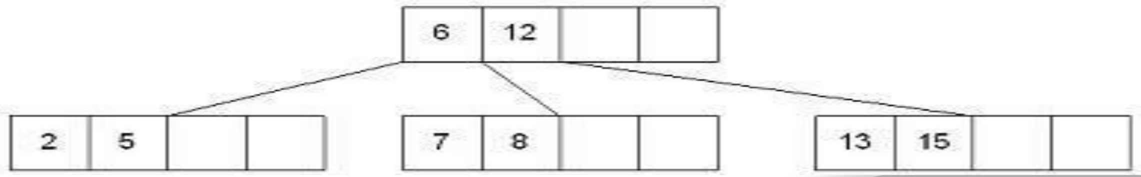
Insert 6 into the following B-tree:



results in a split of the first leaf node:



The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:

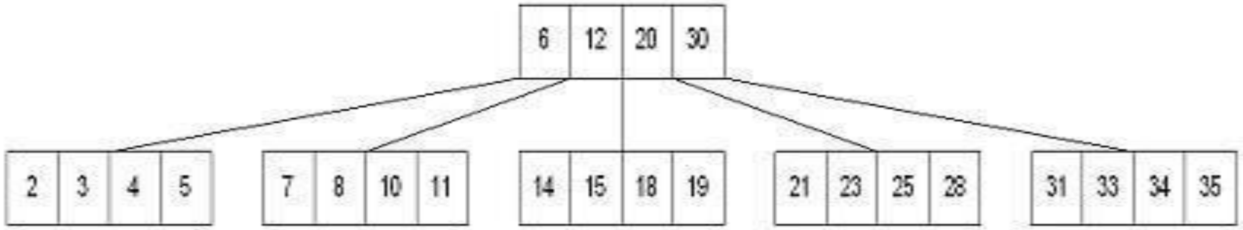


Insertion into a B-Tree

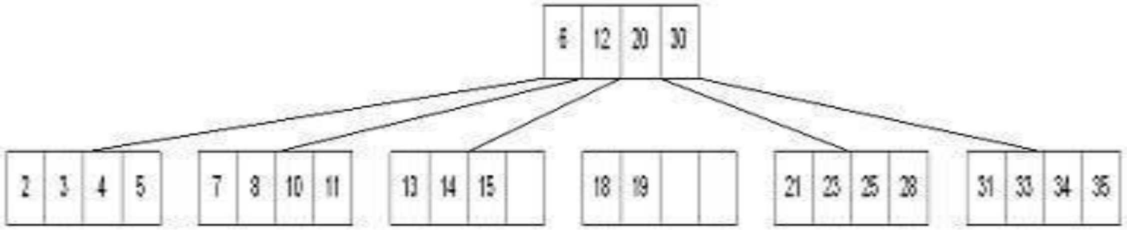
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

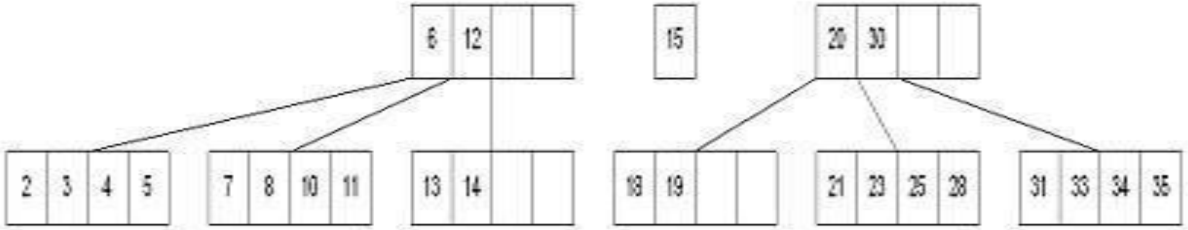
Inserting 13 into the following tree:



Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:

Deleting from a B -Tree

- The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

- There are two main cases to be considered:

  i. Deletion from a leaf

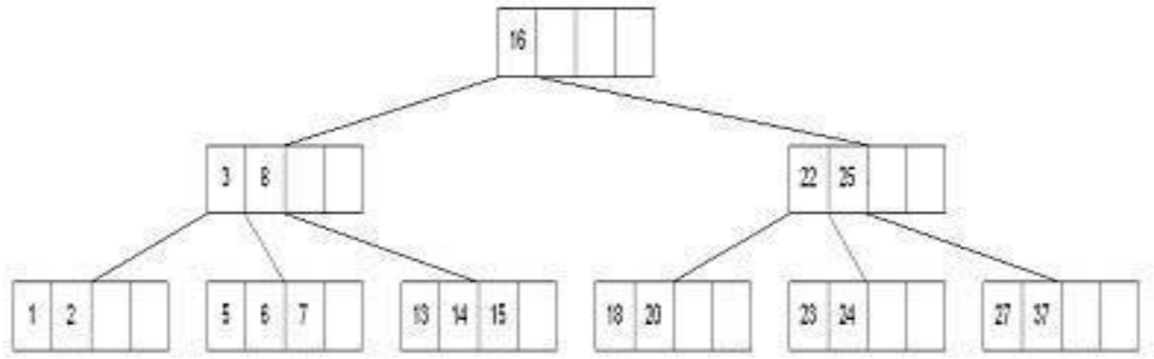
  ii. Deletion from a non-leaf

Deleting from a B -Tree

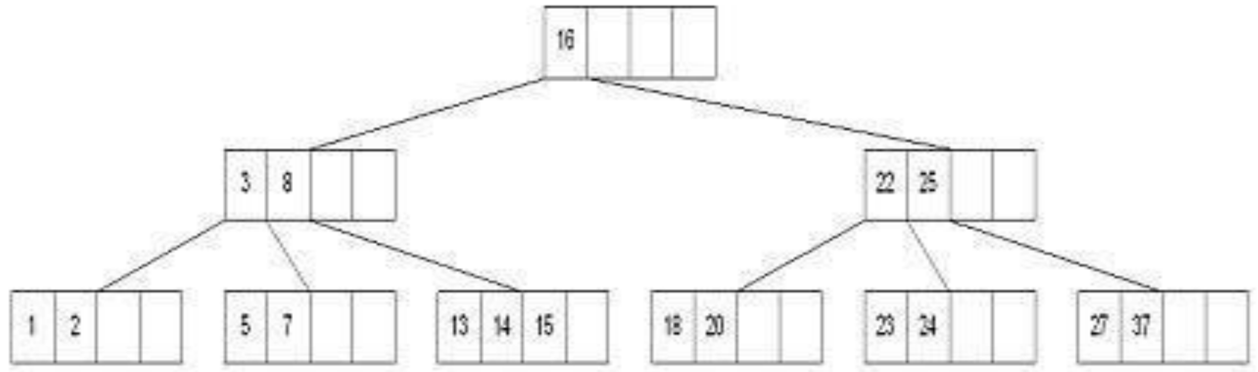
Case 1: Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

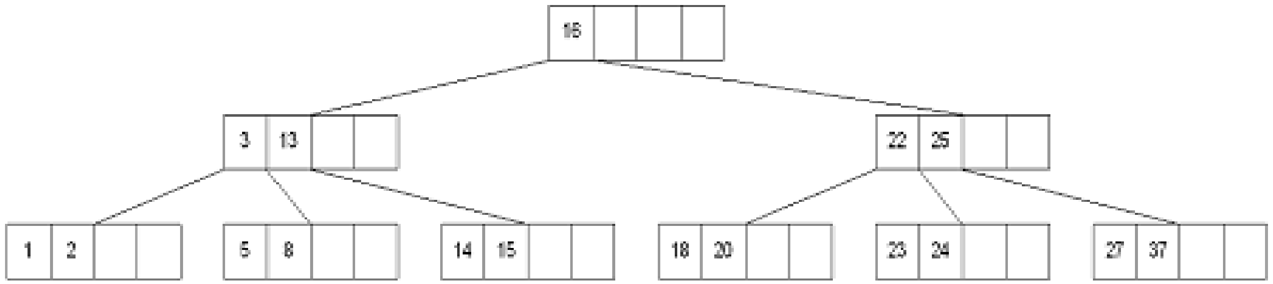


results in:

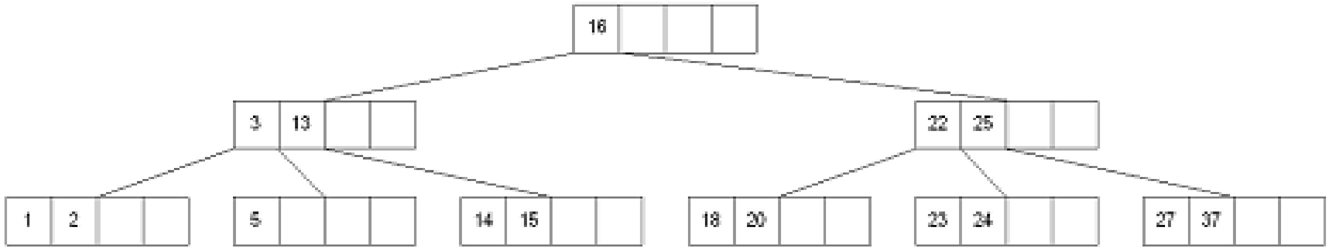


Deleting from a B -Tree

If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.



Now delete 8 from the tree:



If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step 1b-2 until the minimum requirement is met or the root of the tree is reached.

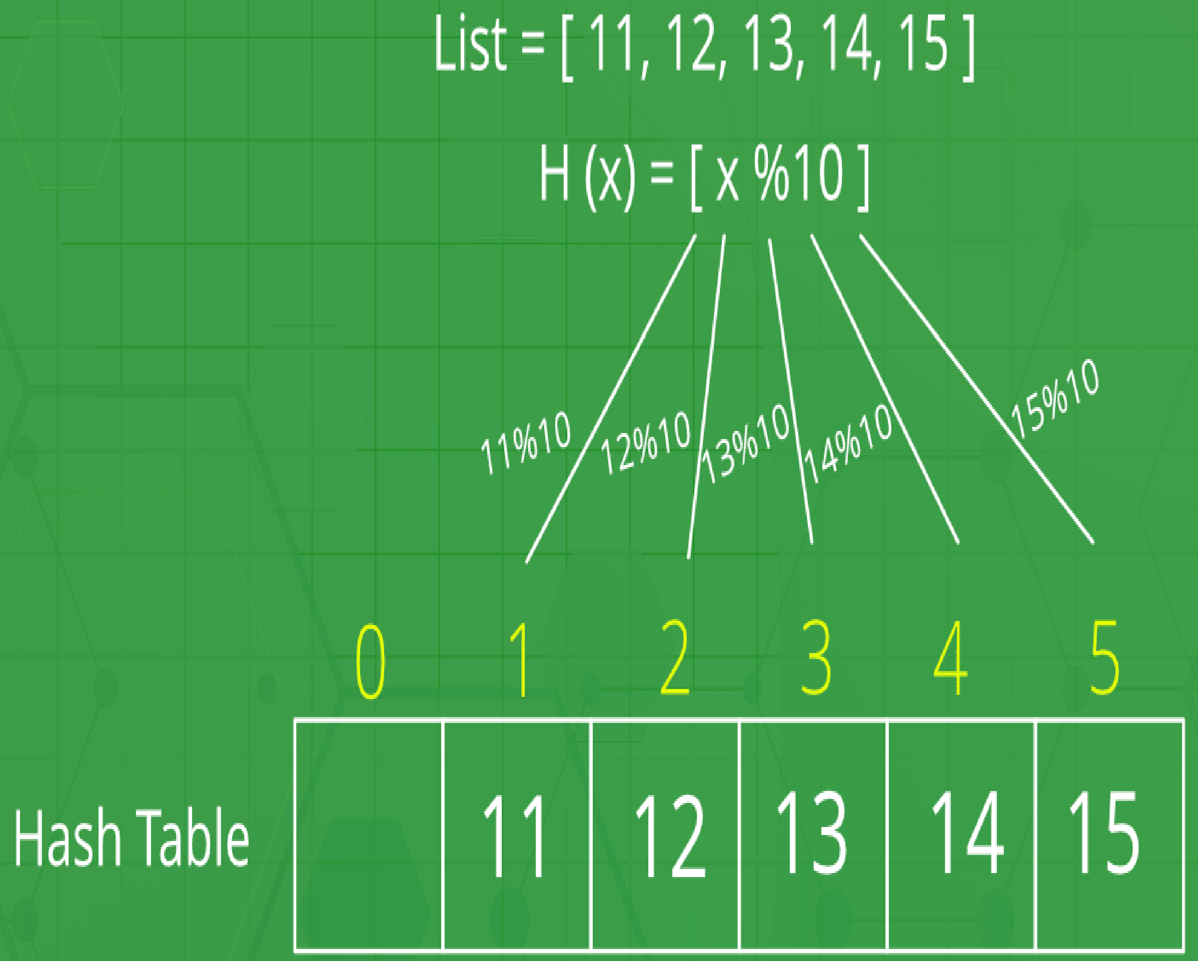
Hashing

- Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.

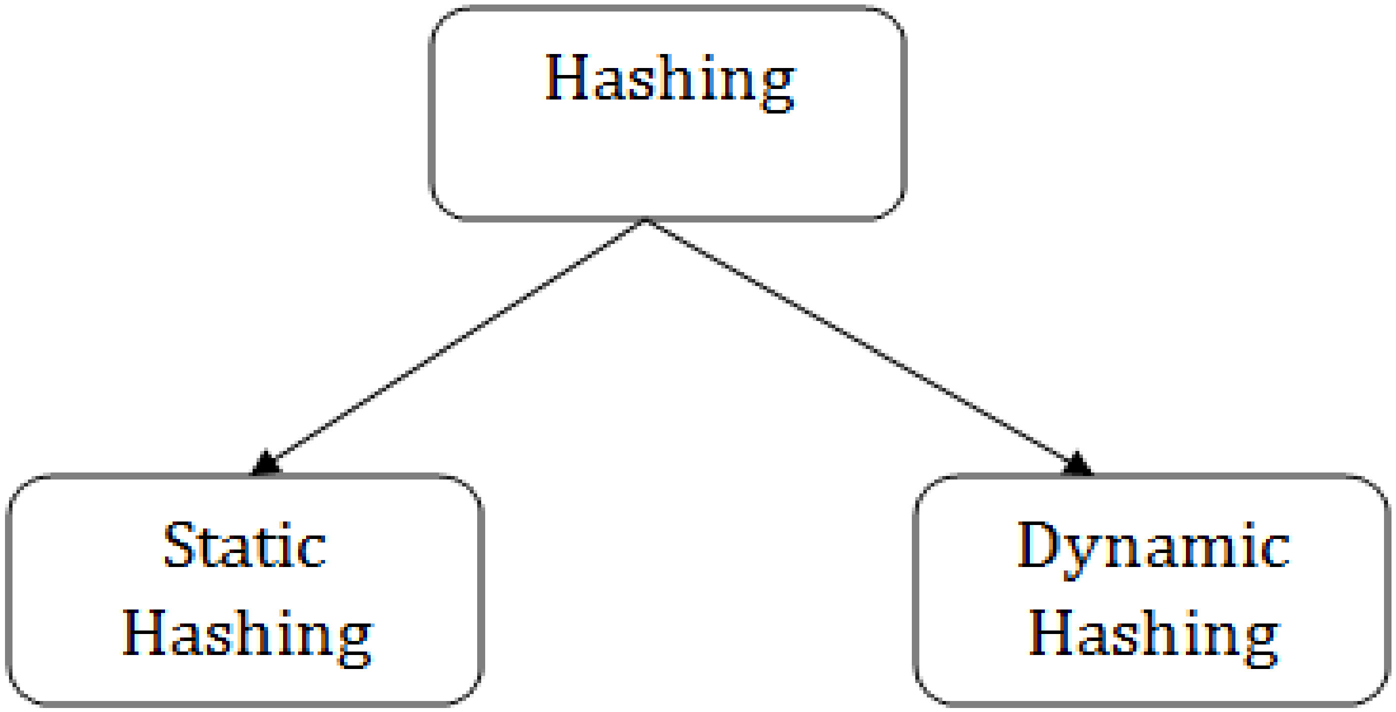
- Taking a very simple example of it, an array with its index as key is the example of hash table. So each index (key) can be used for accessing the value in a constant search time. This mapping key must be simple

- In a hashing system the keys are stored in an array which is called the Hash Table. A perfectlyimplemented hash table would always promise an average insert/ delete / retrieval time of O(1).

Hashing Data Structure



Types of Hashing:



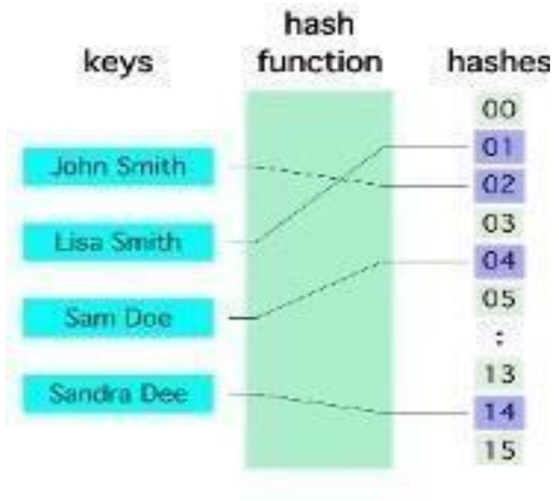
Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value.

Access of data becomes very fast if we know the index of the desired data.
Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing Function

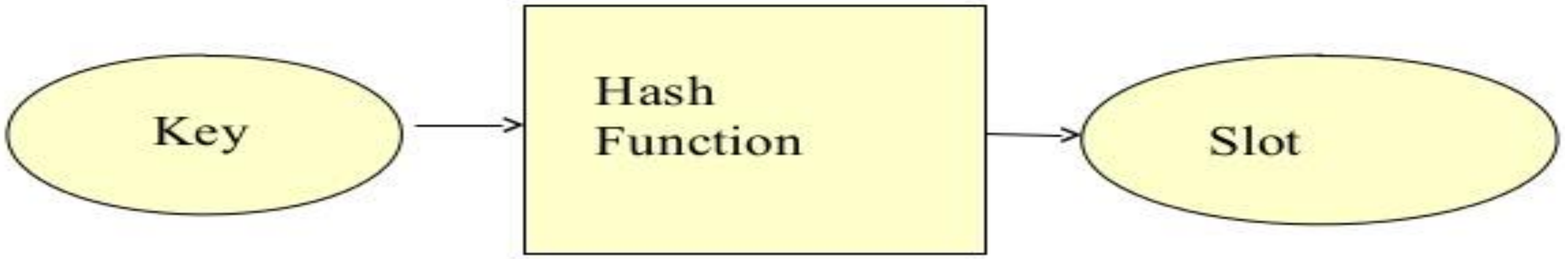
- A function which employs some algorithm to computes the key *K* for all the data elements in the set *U,* such that the key *K* which is of a fixed size. The same key *K* can be used to map data to a hash table and all the operations like insertion, deletion and searching should be possible. The values returned by a hash function are also referred to as hash values, hash codes, hash sums, or hashes.



Hash Functions

A Good Hash function is one which distribute keys evenly among the slots.

And It is said that Hash Function is more art than a science. Becoz it need to analyze the data.



Choice of hash function.

- ***Really tricky!***
- **To avoid collision (two different pairs are in the same the same bucket.)**
- **Size (number of buckets) of hash table.**

Overflow handling method.

- **Overflow: there is no space in the bucket for the new pair.**

Choice of Hash Function

Requirements

easy to compute

minimal number of collisions

If a hashing function groups key values together, this is called clustering of the keys.

A good hashing function distributes the key values uniformly throughout the range.

Some hash functions

Middle of square

H(x):= return middle digits of x^2

Division

H(x):= return x % k

Multiplicative:

H(x):= return the first few digits of the fractional part of x*k, where k is a fraction

Folding:

Partition the identifier x into several parts, and add the parts together to obtain the hash address

e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699

Shift folding vs. folding at the boundaries

Digit analysis:

If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.

Collision

A collision occurs when two different keys hash to the same value

E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value

18 mod 17 = 1 and 35 mod 17 = 1

Cannot store both data records in the same slot in array!

Two different methods for collision resolution:

collision resolution

Two classes:

(1) Closed hashing or open addressing:

search for empty slots using a second function and store item in first empty slot that is found

(2) Open hashing or separate chaining:

Separate Chaining: Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot

- ❖Separate chaining = Open hashing
- ❖Closed hashing = Open addressing

Closed Hashing (Open Addressing)

- In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself.
- While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is   adopted.
- Liner Probing
- Quadratic probing
- Double hashing

D=8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3

- Where do we insert *d*? 3 already filled
- Probe sequence using linear hashing:

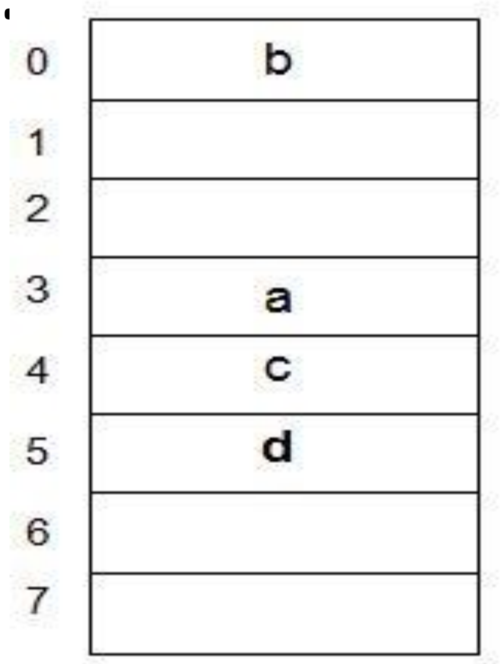
  $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$

  $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$

  $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$

  etc.

  7, 0, 1, 2
- Wraps around the beginning of the table!

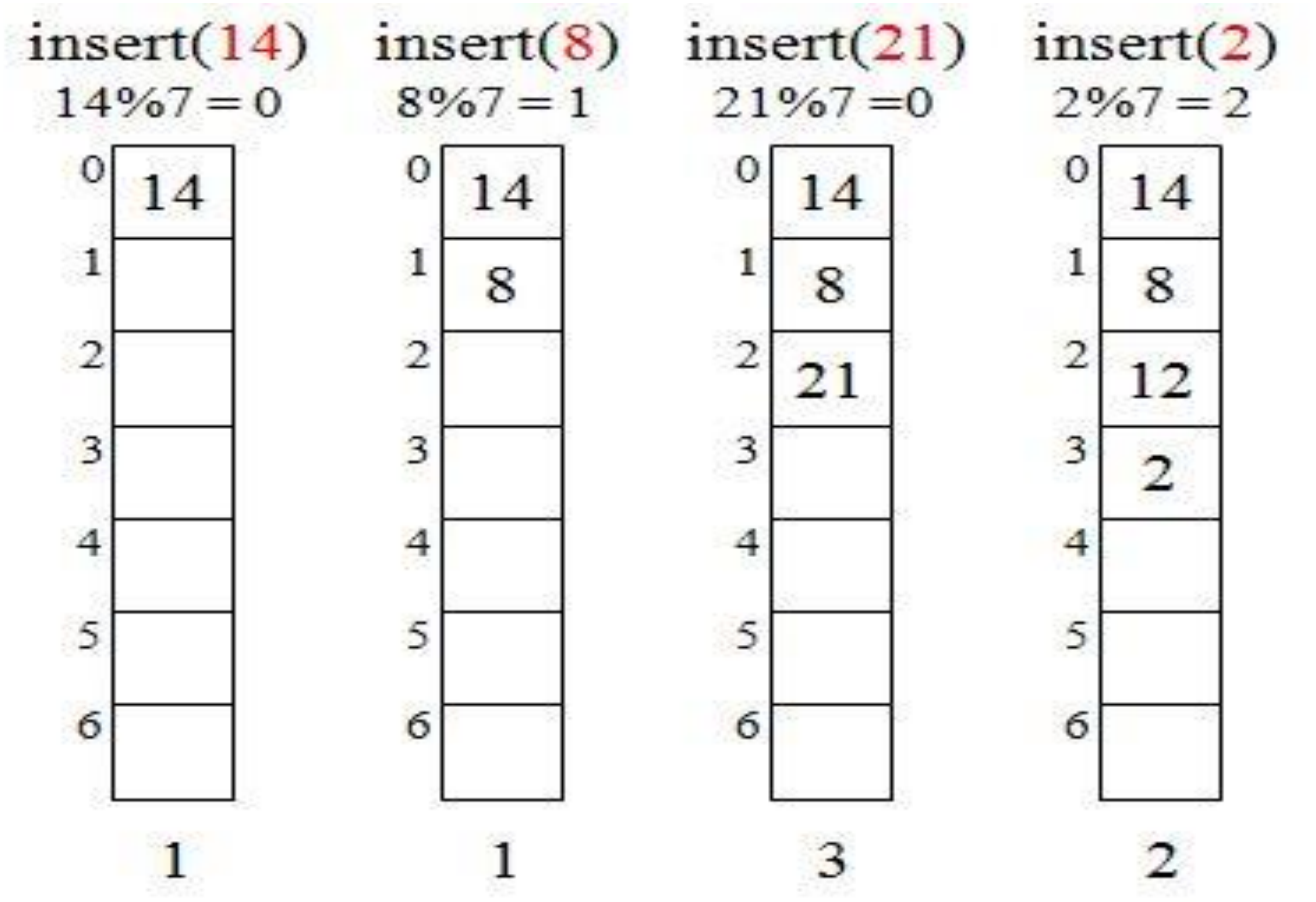


Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell

$h_i(X) = (Hash(X) + i) \bmod \textit{TableSize} \quad (i = 0, 1, 2, \ldots)$

Compute hash value and increment it until a free cell is found

Linear Probing Example



Drawbacks of Linear Probing

Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)

Very prone to cluster formation

Primary clustering – clusters grow when keys hash to values close to each other

Quadratic Probing

Main Idea: Spread out the search for an empty slot
Increment by i^2 instead of i

$h_i(X) = (Hash(X) + i^2)$ % *TableSize*

    h0(X) = Hash(X) % TableSize
    h1(X) = Hash(X) + 1 % TableSize
    h2(X) = Hash(X) + 4 % TableSize
    h3(X) = Hash(X) + 9 % TableSize

Double hashing

Double hashing is one of the best methods for dealing with collisions.

If the slot is full, then a second hash function is calculated and combined with the first hash function.

$H(k, i) = (H_1(k) + i\, H_2(k)) \% m$

Double Hashing

Idea: Spread out the search for an empty slot by using a second hash function

No primary or secondary clustering

$h_i(X) = (Hash_1(X) + iHash_2(X)) \bmod \mathit{TableSize}$

for i = 0, 1, 2, …

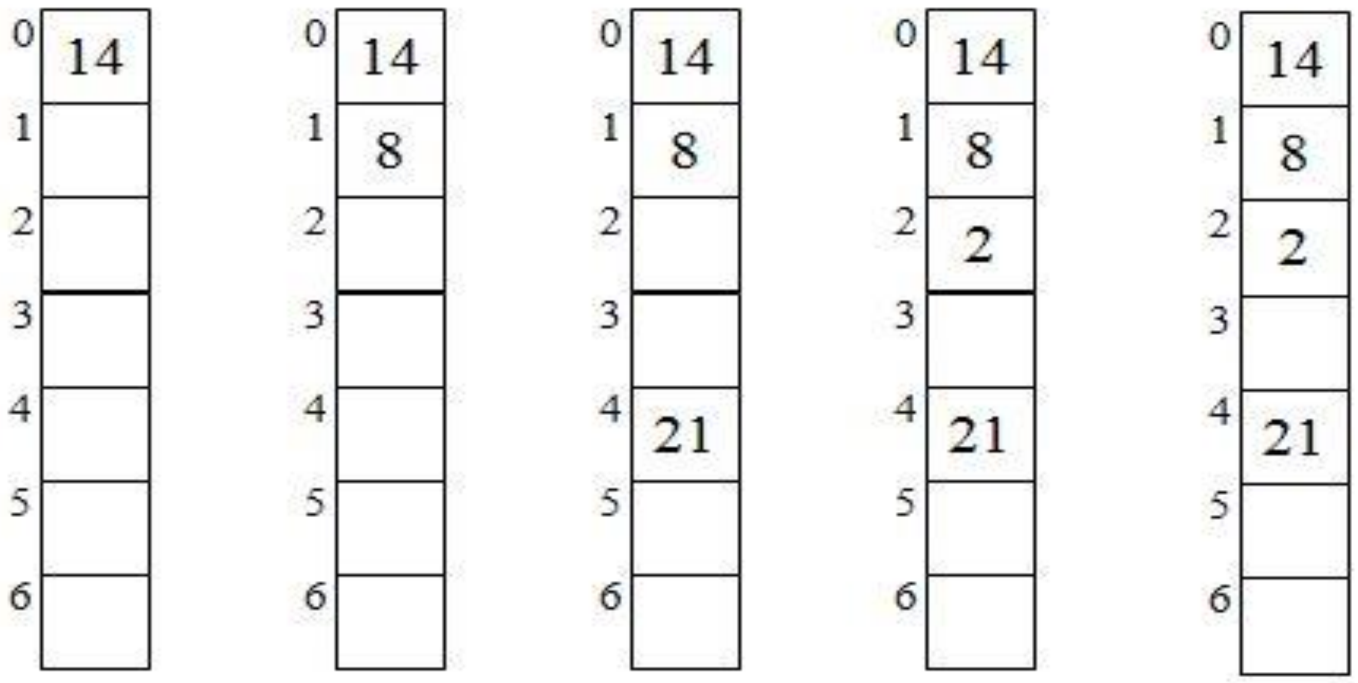
Integer keys:

$Hash_2(X) = R – (X \bmod R)$

where R is a prime smaller than *TableSize*

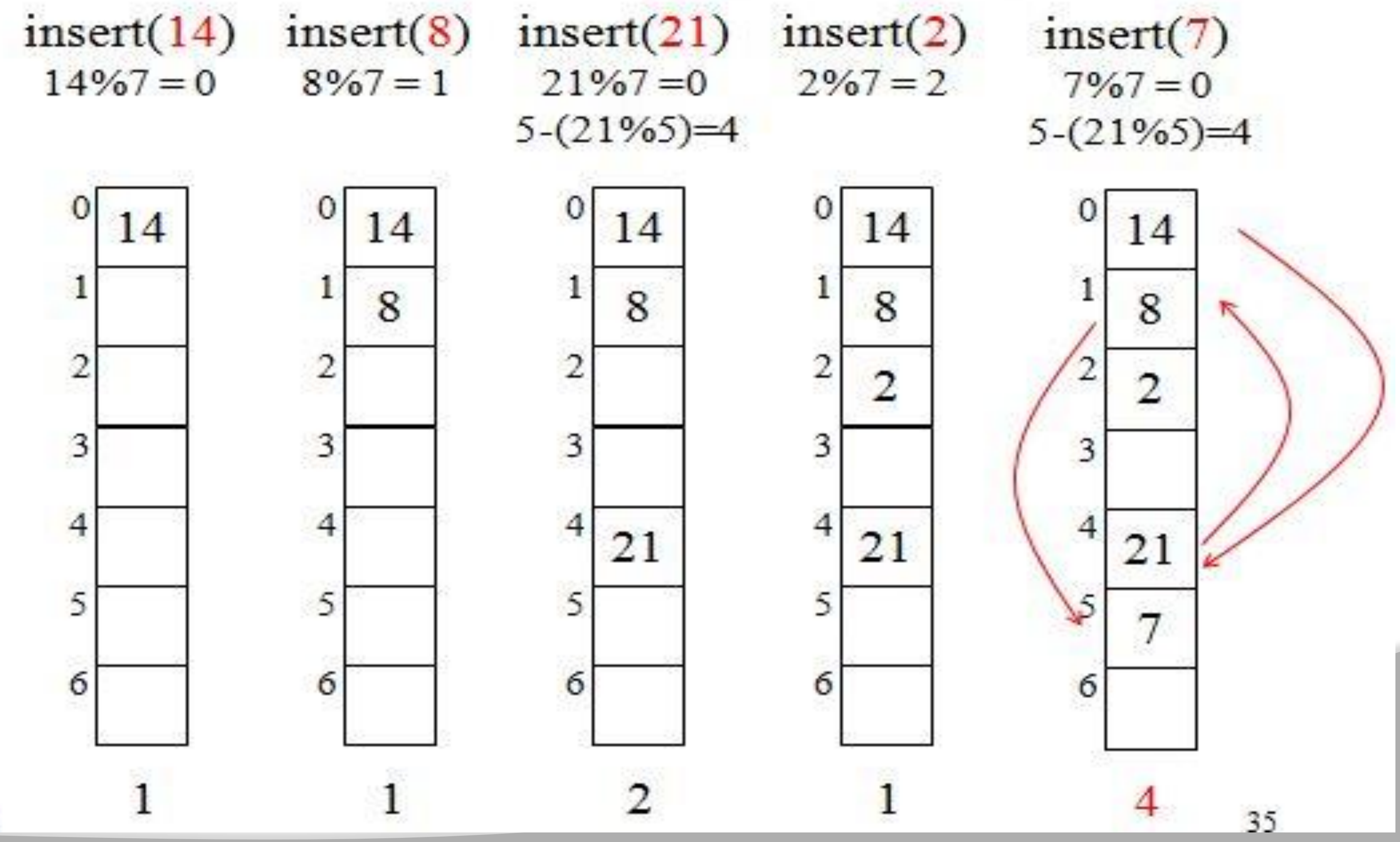
Double Hashing Example

insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
		5-(21%5)=4		5-(21%5)=4



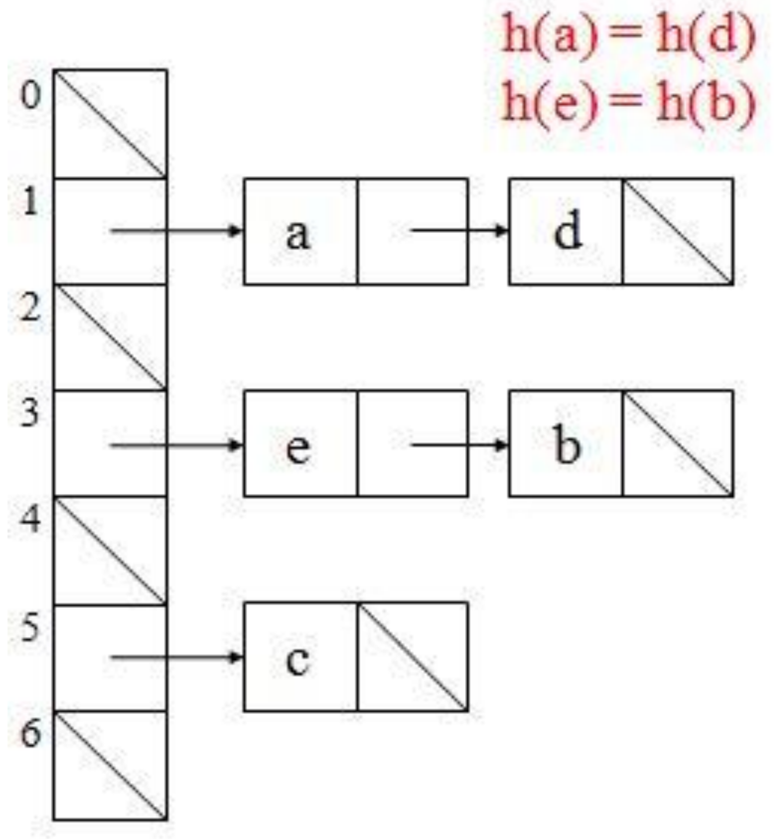
probes: 1 1 2 1 ??

Double Hashing Example



Hashing with Separate Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered linked list (chain)
- Properties
  - performance degrades with length of chains



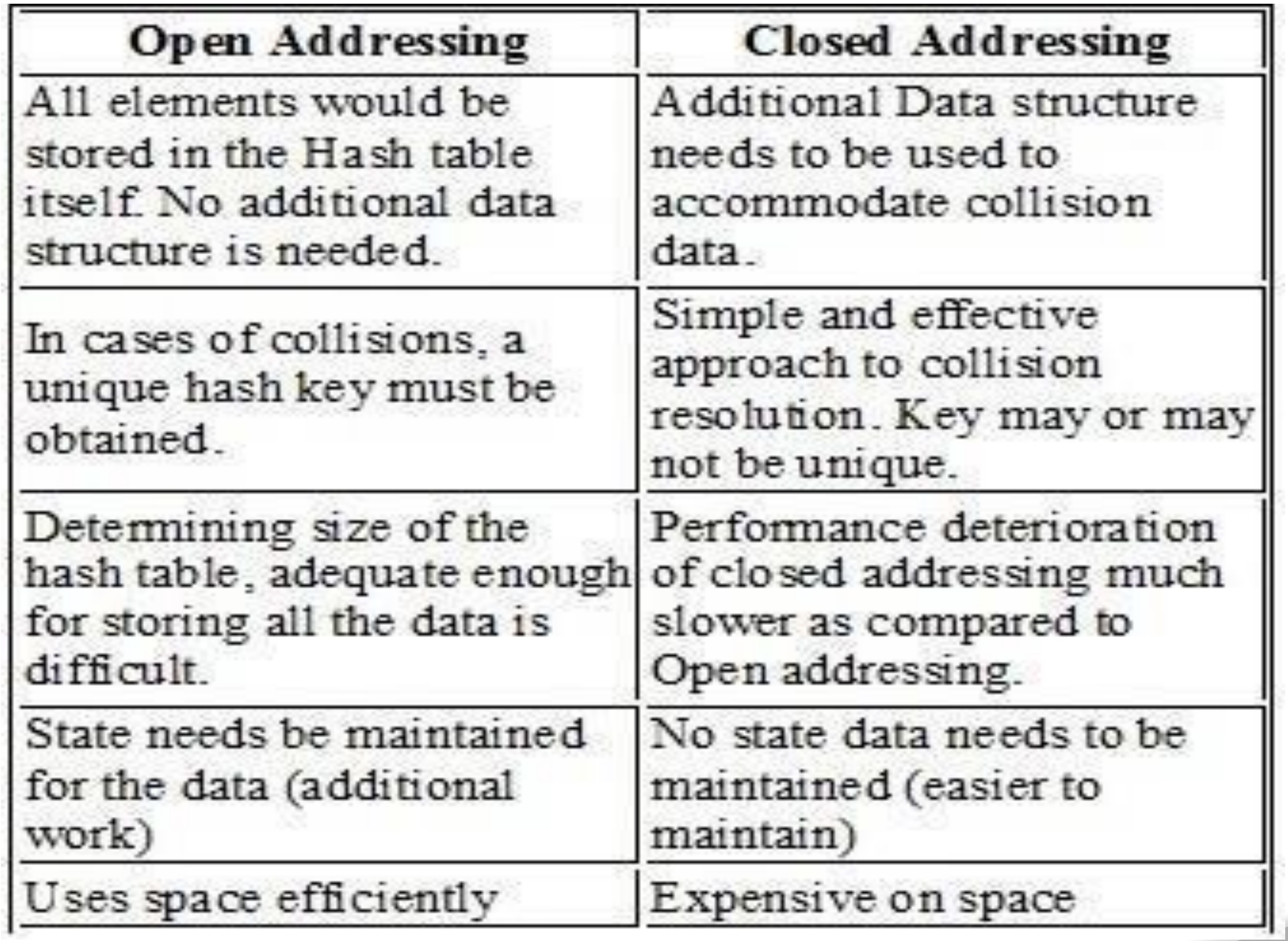
Problem with separate chaining:

Memory consumed by pointers –
32 (or 64) bits per key!

What if we only allow one Key at each entry?

two objects that hash to the same spot can't both go there
first one there gets the spot
next one must *go in another spot*

A Comparative Analysis ofClosed Hashing vs Open Hashing



Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space

Applications of Hashing

- A hash function maps a variable length input string to fixed length output string -- its hash value, or hash for short. If the input is longer than the output, then some inputs must map to the same output -- a hash collision.
- Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, and authentication.
- In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on downloaded files.

Applications of Hashing

- Construct a *message authentication code* (MAC)
- Digital signature
- Make commitments, but reveal message later
- Timestamping
- Key updating: key is hashed at specific intervals resulting in new key

THANK YOU