Introduction to Data Structures

Definition

- **Data:** Collection of **raw facts**.
- **Data structure** is representation of the logical relationship existing between individual elements of data.
- **Data structure** is a specialized format for organizing and storing data in memory that considers not only the elements stored but also their relationship to each other.

Introduction

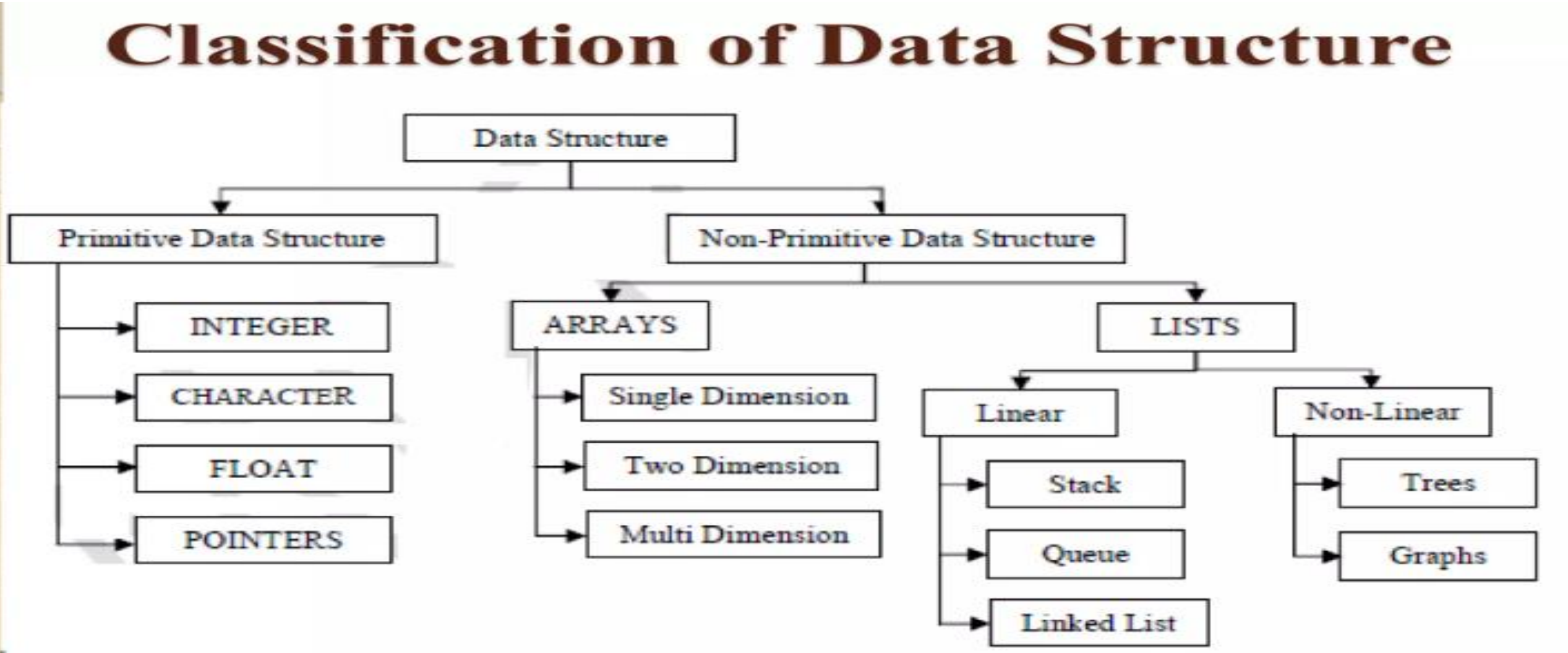
- Data structure affects the design of both structural & functional aspects of a program.

Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.

Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - Primitive Data Structure
  - Non-Primitive Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- ***Data structures that are directly operated upon the machine-level instructions are known as primitive data structures.***
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Destroy or Delete

Non-Primitive Data Structure

- There are more sophisticated data structures.
- ***The Data structures that are derived from the primitive data structures are called Non-primitive data structure.***
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

Linear Data structures:

- *Linear Data structures are kind of data structure that has homogeneous elements.*
- The data structure in which elements are in a sequence and form a liner series.
- Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.
- Some commonly used linear data structures are **Stack, Queue and Linked Lists.**

Non-Linear Data structures:

- *A Non-Linear Data structures is a data structure in which data item is connected to several other data items.*
- Non-Linear data structure may exhibit either a hierarchical relationship or parent child relationship.
- The data elements are not arranged in a sequential structure.
- The different non-linear data structures are **trees and graphs.**

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Traversal
  - Insertion
  - Selection
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

- **A primitive data structure** is generally a basic structure that is usually built into the language, such as an integer, a float.

- **A non-primitive data structure** is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

One dimensional array:

- ***An array with only one row or column is called one-dimensional array.***
- It is finite collection of n number of elements of same type such that:
  - can be referred by indexing.
  - The syntax Elements are stored in continuous locations.
  - Elements x to define one-dimensional array is:
- **Syntax: Datatype Array_Name [Size];**
- Where,

  Datatype : Type of value it can store (Example: int, char, float)

  Array_Name: To identify the array.
- Size : The maximum number of elements that the array can hold.

Arrays

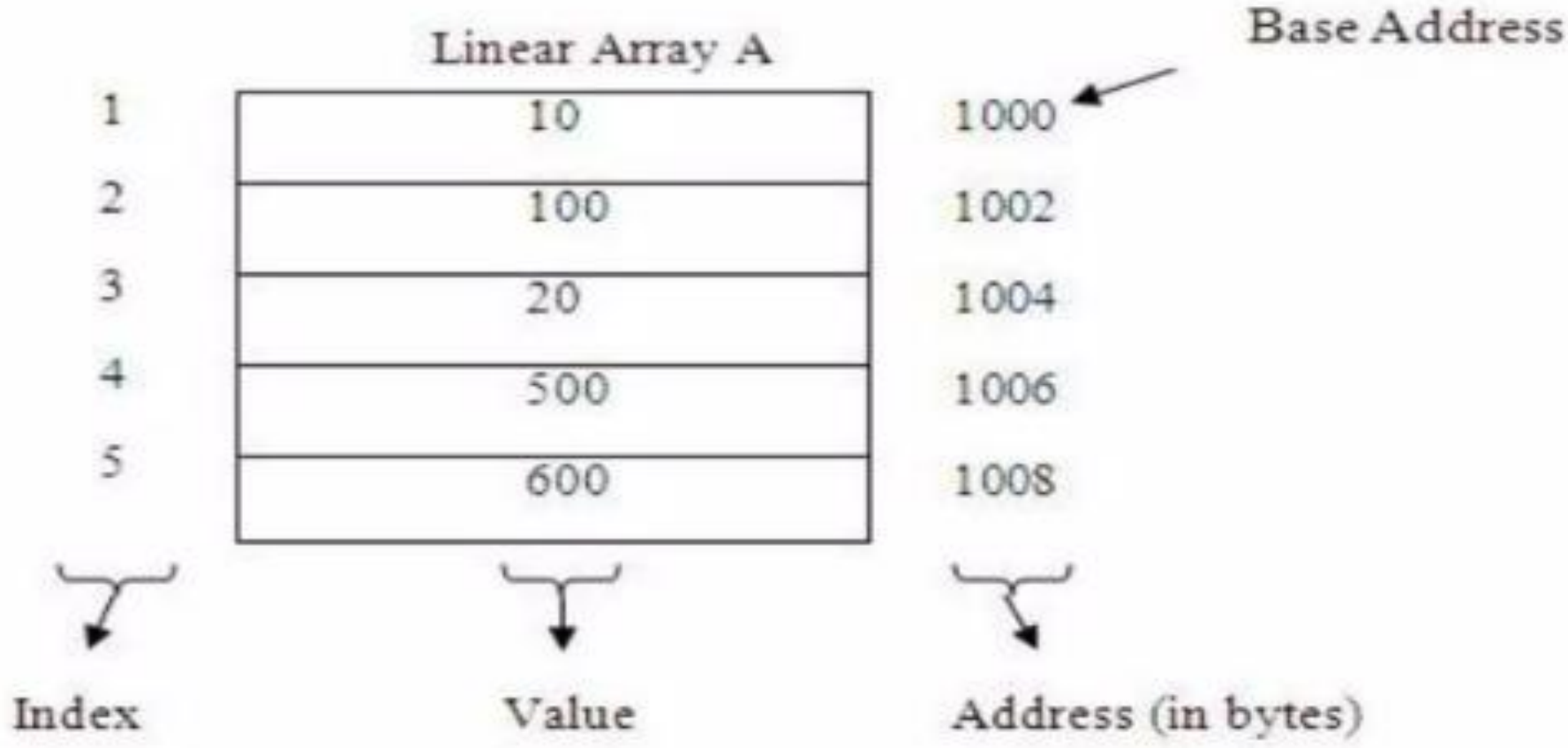
- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Represent a Linear Array in memory

- The elements of linear array are stored in consecutive memory locations. It is shown below:



Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1
- For the above array it would be (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop.

```
For(i=0;i<=9;i++)
{   scanf(“%d”,&arr[i]);
   printf(“%d”,arr[i]); }
```

Arrays types

- Single Dimension Array
  - Array with one subscript
- Two Dimension Array
  - Array with two subscripts (Rows and Column)
- Multi Dimension Array
  - Array with Multiple subscripts

Basic operations of Arrays

- Some common operation performed on array are:
  - Traversing
  - Searching
  - Insertion
  - Deletion
  - Sorting
  - Merging

Traversing Arrays

- **Traversing:** It is used to access each data item exactly once so that it can be processed.
  E.g.
  We have linear array A as below:

1	2	3	4	5
10	20	30	40	50

Here we will start from beginning and will go till last element and during this process we will access value of each element exactly once as below:

A [1] = 10
A [2] = 20
A [3] = 30
A [4] = 40
A [5] = 50

ALGORITHM: Traversal (A, LB, UB) A is an array with Lower Bound LB and Upper Bound UB.

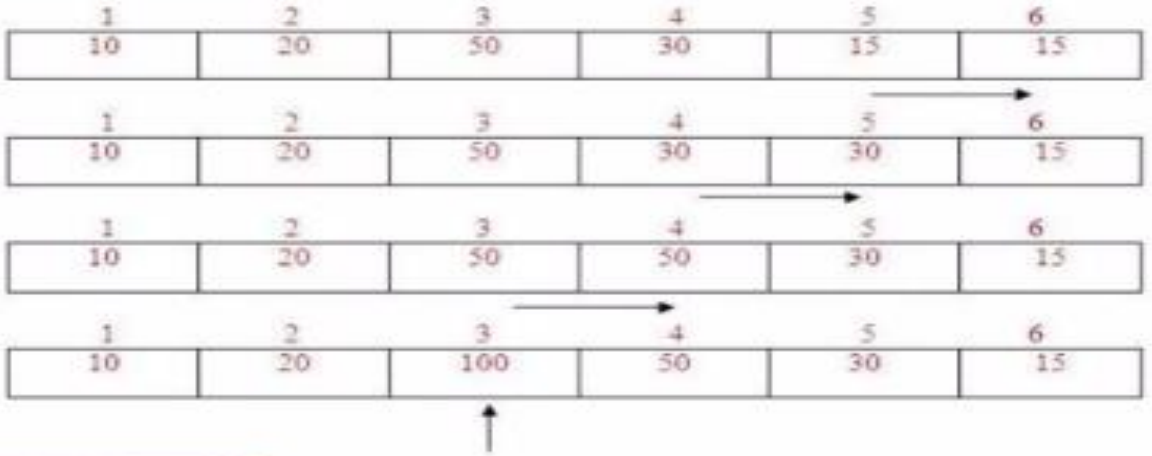
```
Step 1:    for LOC = LB to UB do
Step 2:        PROCESS A [LOC]
           [End of for loop]
Step 3:    Exit
```

Insertion into Array

- **Insertion:** It is used to add a new data item in the given collection of data items.
E.g. We have linear array A as below:

1	2	3	4	5
10	20	50	30	15

New element to be inserted is 100 and location for insertion is 3. So shift the elements from 5th location to 3rd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below:



ALGORITHM: Insert (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be inserted in the position Pos.

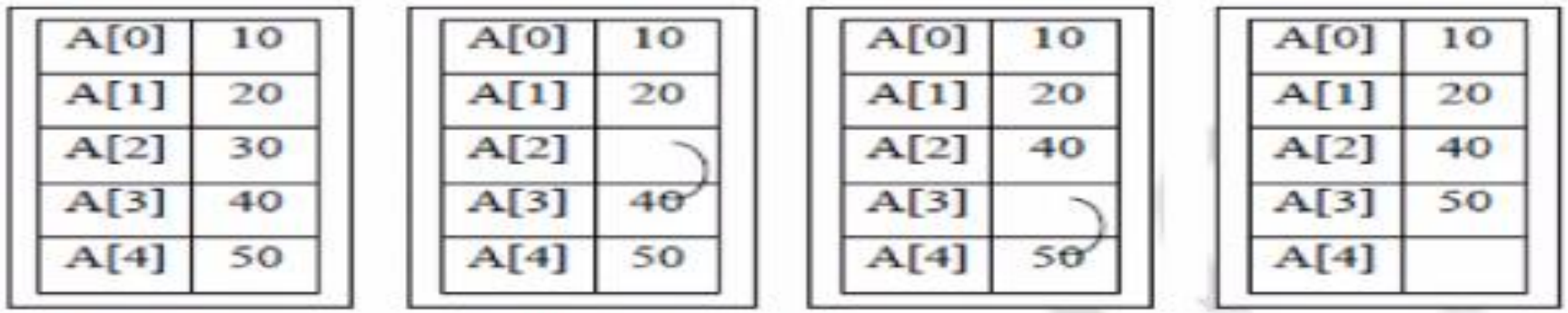
```
Step 1:   for I = N-1 down to Pos
              A[ I + 1] = A[I]
          [End of for loop]
Step 2:   A [Pos] = ITEM
Step 3:   N = N+1
Step 4:   Exit
```

Deletion from Array

- **Deletion:** It is used to delete an existing data item from the given collection of data items.

For example: Let A[4] be an array with items 10, 20, 30, 40, 50 stored at consecutive locations. Suppose item 30 has to be deleted at position 2. The following procedure is applied.

- Copy 30 to ITEM, i.e. Item = 30.
- Move Number 40 to the position 2.
- Move Number 50 to the position 3.



ALGORITHM: Delete (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be deleted in the position Pos and it is stored into variable Item.

```
Step 1:     ITEM = A [Pos]
Step 2:     for I = Pos down to N-1
                A[ I ] = A[I+1]
            [End of for loop]
Step 3:     N = N-1
Step 4:     Exit
```

Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.

E.g. We have linear array A as below:

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

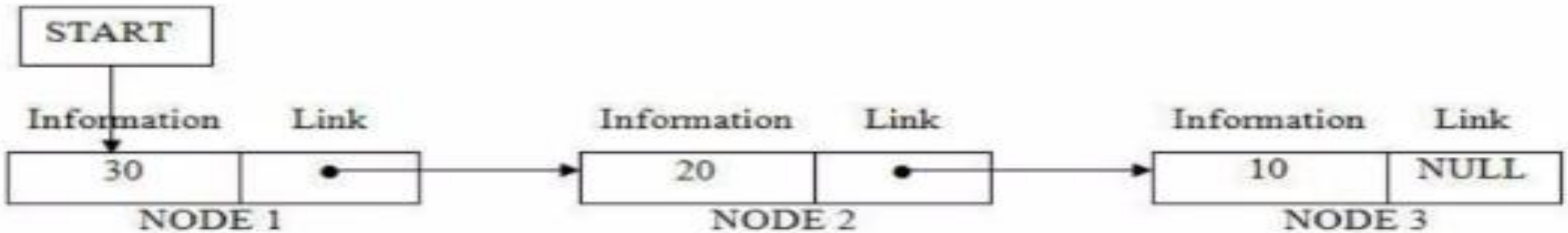
1) Compare 20 with 15
   20 # 15, go to next element.
2) Compare 20 with 50
   20 # 50, go to next element.

3) Compare 20 with 35
   20 #35, go to next element.

4) Compare 20 with 20
   20 = 20, so 20 is found and its location is 4.

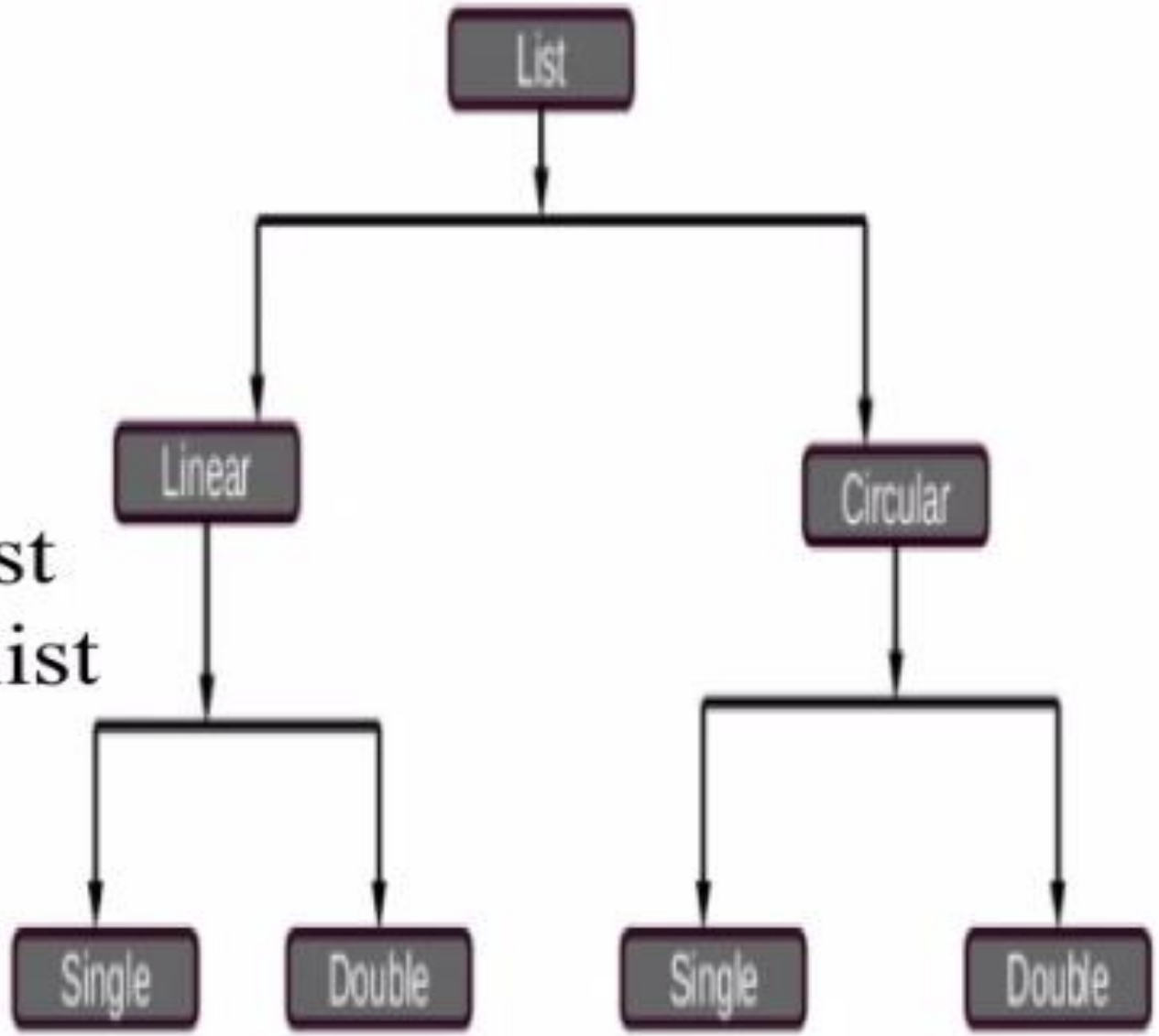
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items called ***nodes***.
- Lists are the most commonly used non-primitive data structures.
- Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - o The second part contains the memory address of the next node in the list. Also called Link part.



Lists

- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list

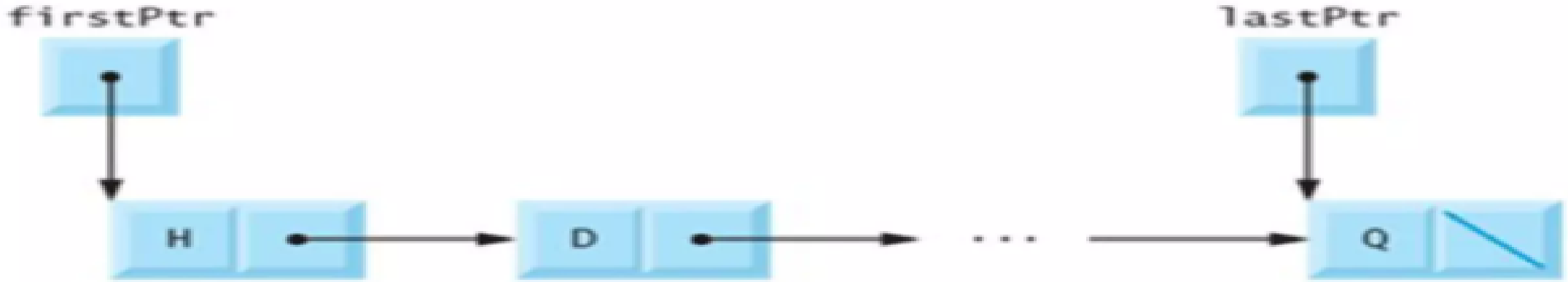


Single linked list

A singly linked list contains two fields in each node – an information field and the linked field.

- The ***information*** field contains the data of that node.
- The ***link*** field contains the memory address of the next node.

There is only one link field in each node, the linked list is called singly linked list.

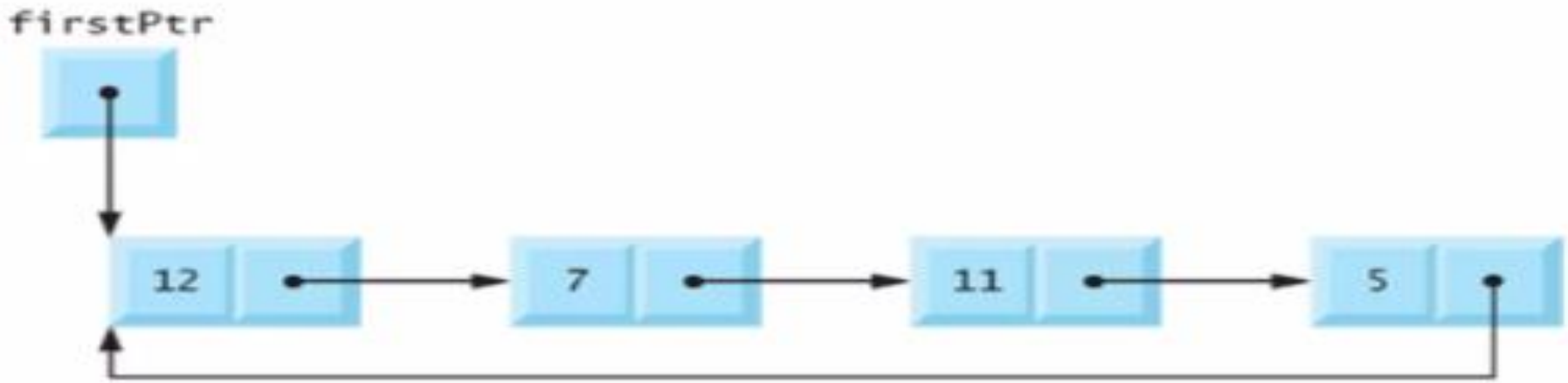


A graphical representation of a list.

Single circular linked list

The link field of the last node contains the memory address of the first node, such a linked list is called circular linked list.

· In a circular linked list every node is accessible from a given node.

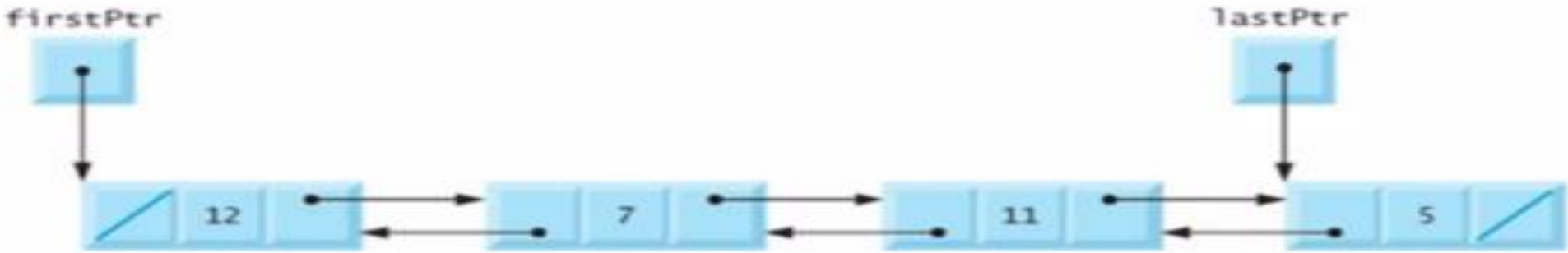


Circular, singly linked list.

Doubly linked list

It is a linked list in which each node is points both to the next node and also to the previous node.

- In doubly linked list each node contains three parts:
  - FORW : It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.
- In the first node, if BACK contains NULL, it indicated that it is the first node in the list.
- The node in which FORW contains, NULL indicates that the node is the last node.



Doubly linked list.

Doubly circular linked list

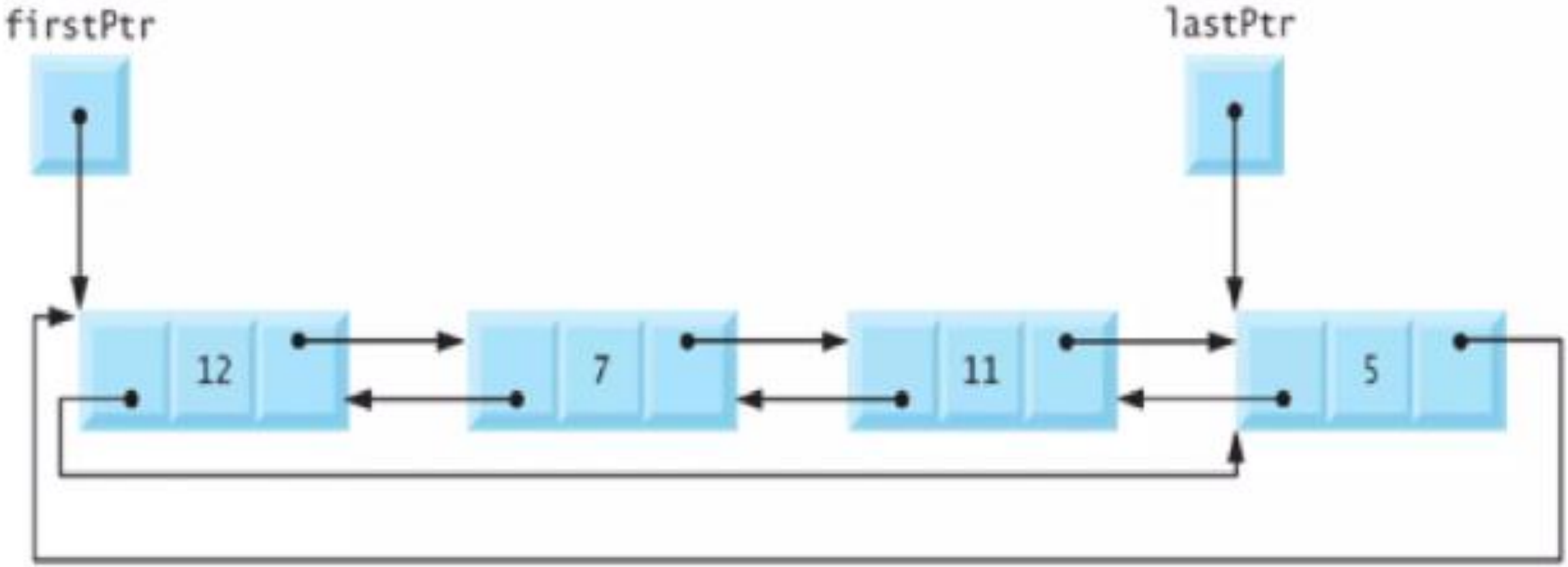


Fig. 19.12 | Circular, doubly linked list.

Operation on Linked List

- The operation that are performed on linked lists are:
  - Creating a linked list
  - Traversing a linked list
  - Inserting an item into a linked list.
  - Deleting an item from the linked list.
  - Searching an item in the linked list
  - Merging two or more linked lists.

Creating a linked list

- The nodes of a linked list can be created by the following structure declaration.

```
struct Node
{
        int info;
        struct Node *link;
}*node1, node2;
```

- Here info is the information field and link is the link field.
- The link field contains a pointer variable that refers the same node structure. Such a reference is called as ***Self addressing pointer***.

Operator new and delete

- Operators new allocate memory space.
  - Operators new [ ] allocates memory space for array.
- Operators delete deallocate memory space.
  - Operators delete [ ] deallocate memory space for array.

Traversing a linked list:

- Traversing is the process of accessing each node of the linked list exactly once to perform some operation.
- **ALGORITHM: TRAVERS (START, P)** START contains the address of the first node. Another pointer p is temporarily used to visit all the nodes from the beginning to the end of the linked list.

Step 1: P = START

Step 2: while P != NULL

Step 3: PROCESS data (P) [Fetch the data]

Step 4: P = link(P) [Advance P to next node]

Step 5: End of while

Step 6: Return

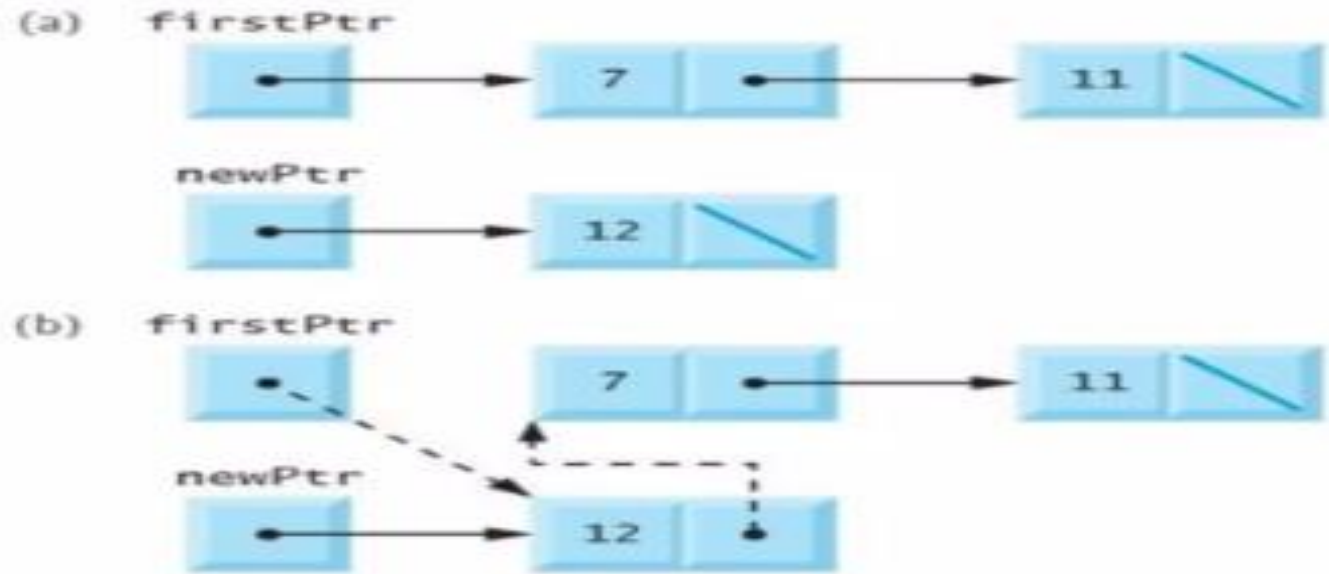
Inserting a node into the linked list

- Inserting a node at the beginning of the linked list
- Inserting a node at the given position.
- Inserting a node at the end of the linked list.

Inserting node at Front

Inserting a node at the beginning of the linked list

1. Create a node.
2. Fill data into the data field of the new node.
3. Mark its pointer field as NULL
4. Attach this newly created node to START
5. Make the new node as the START node.



Operation insertAtFront represented graphically.

Inserting node at Front

- ALGORITHM: INS_BEG (START, P)
  START contains the address of the first node.

  Step 1: P ← new Node;

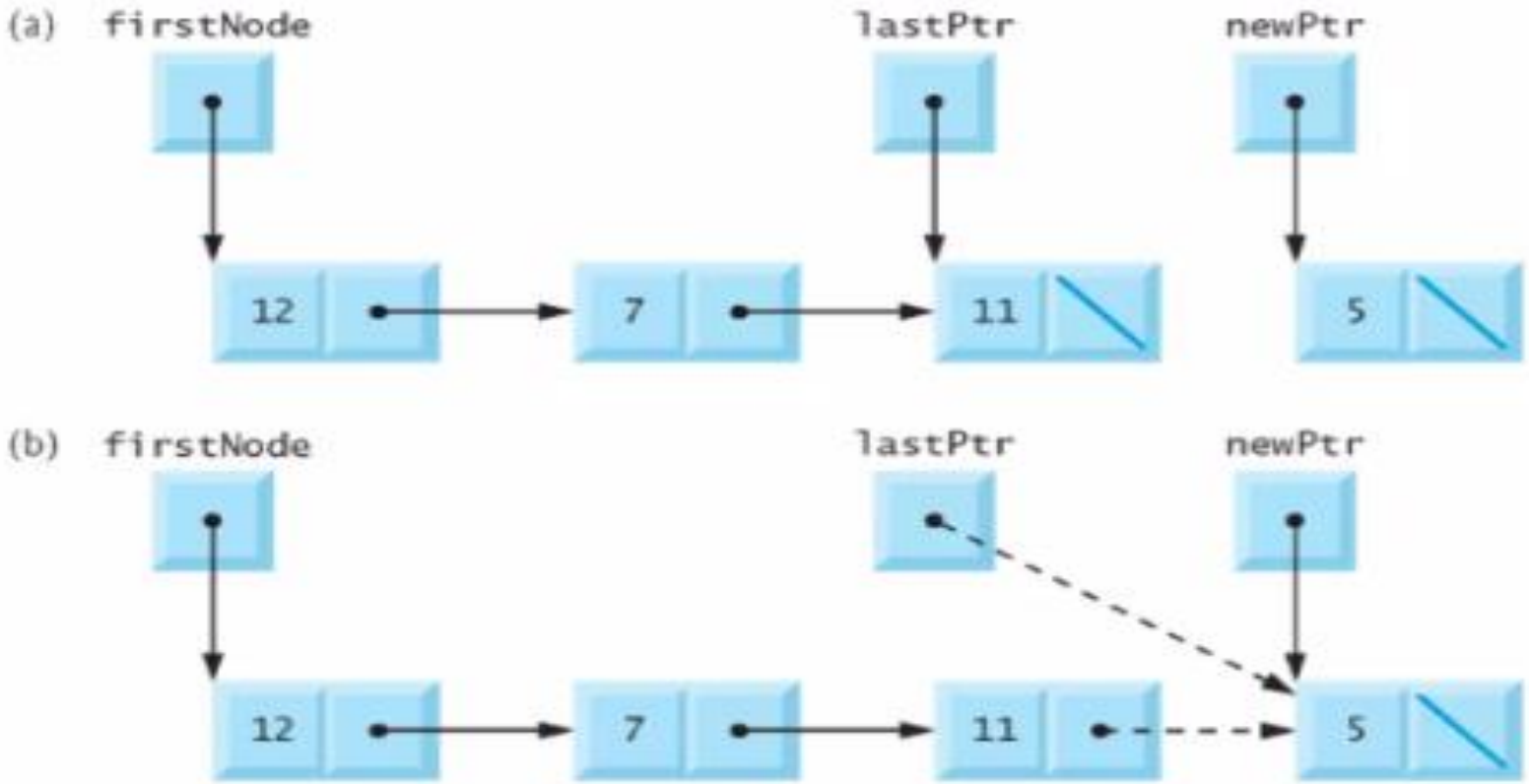
  Step 2: data(P) ← num;

  Step 3: link(P) ← START

  Step 4: START ← P

  Step 5: Return

Inserting node at Last



| Operation `insertAtBack` represented graphically.

Inserting node at Last

- ALGORITHM: INS_END (START, P) START contains the address of the first node.

Step 1: START

Step 2: P ← START [identify the last node]

while P!= null

P ← next (P)

End while

Step 3: N ← new Node;

Step 4: data(N) ← item;

Step 5: link(N) ← null

Step 6: link(P) ← N

Step 7: Return

Inserting node at a given Position

ALGORITHM: INS_POS (START, P) START contains the address of the first node.

Step 1: START

Step 2: P ← START [Initialize node]
Count ← 0

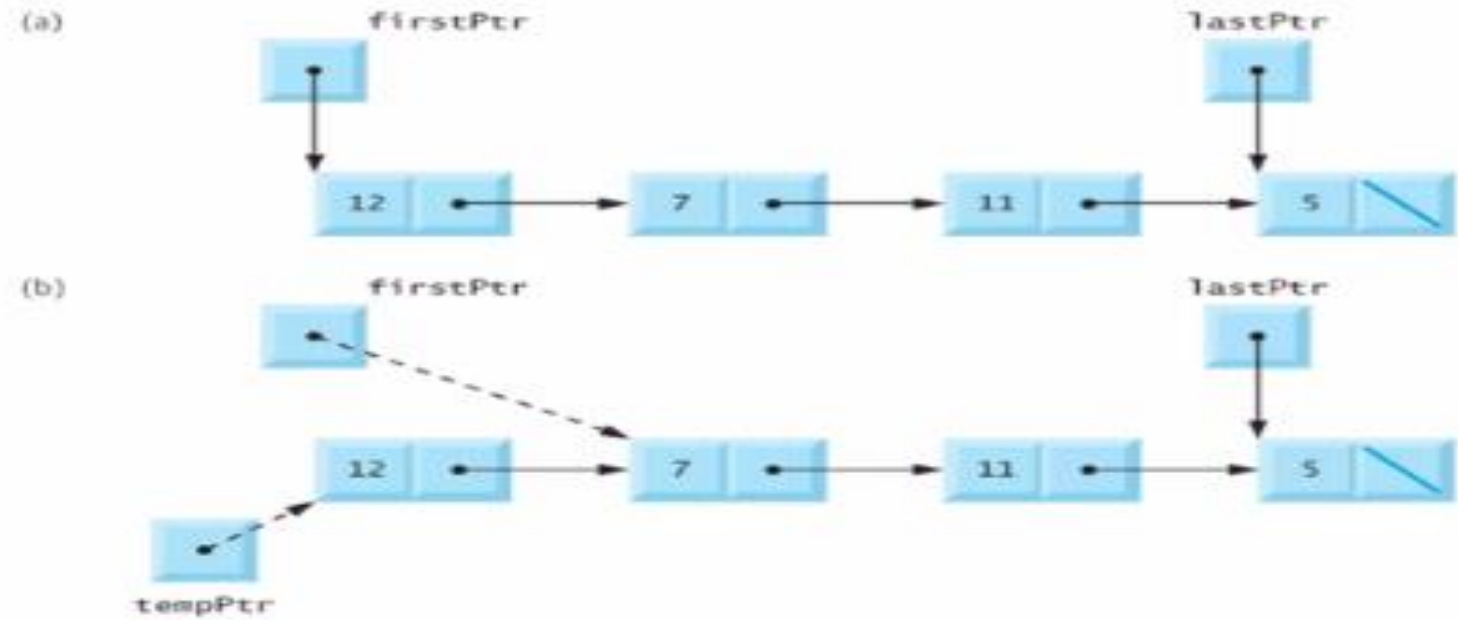
Step 3: while P!= null
count ← count+1
P ← next (P)
End while

Step 4: if (POS=1)
Call function INS_BEG( )
else if (POS=Count +1)
Call function INS_END( )
else if (POS<=Count)
P ← Start
For(i=1; i<=pos; i++)
P ← next(P);
end for
[create] N ← new node
data(N) ← item;
link(N) ← link(P)
link(P) ← N
else
PRINT "Invalid position"

Step 5: Return

Deleting an node

- **Deleting an item from the linked list:**
  - o Deletion of the first node
  - o Deletion of the last node
  - o Deletion of the node at the give position



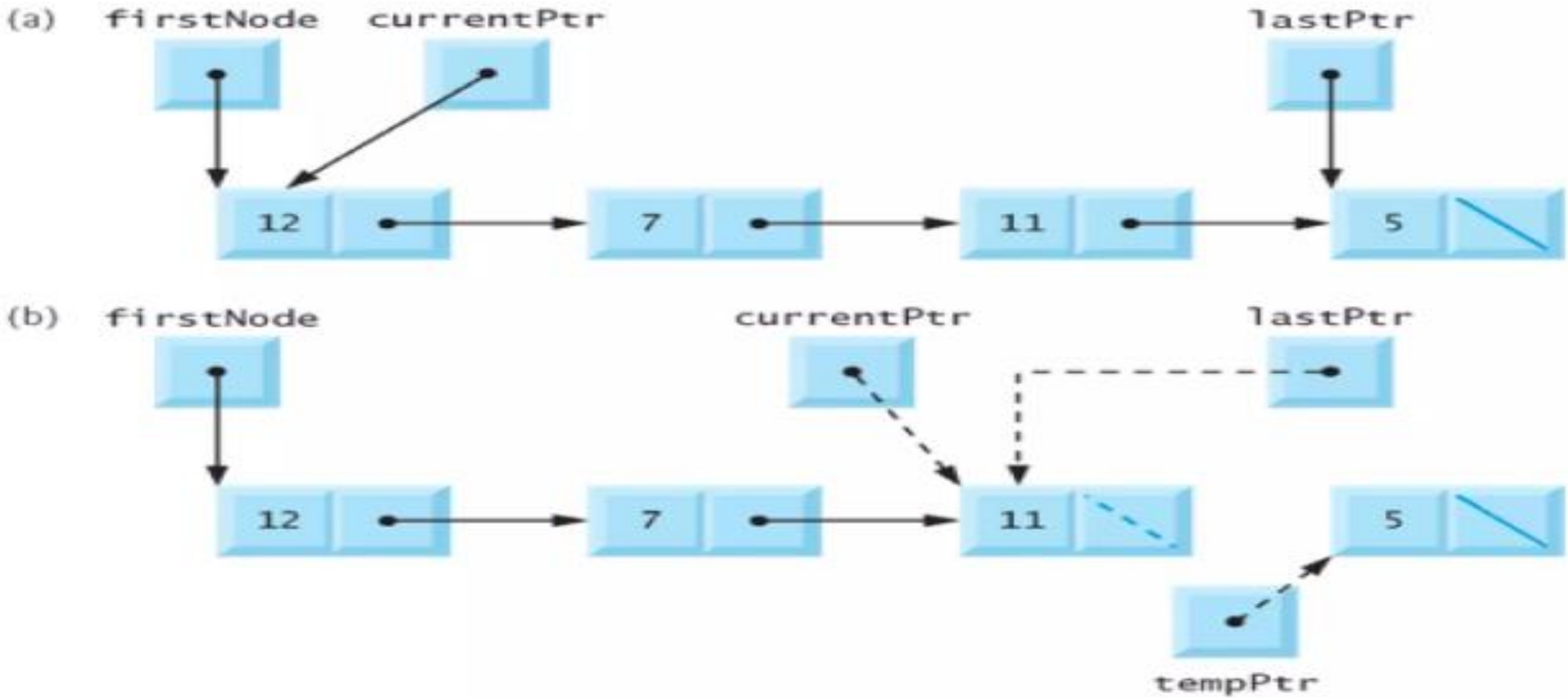
19.8 | Operation removeFromFront represented graphically.

Deleting node from end

ALGORITHM: DEL_END (P1, P2, START) This used two pointers P1 and P2. Pointer P2 is used to traverse the linked list and pointer P1 keeps the location of the previous node of P2.

Step 1: START
Step 2: P2 ⟵ START;
Step 3: while (link(P2) ! = NULL)
P1 ⟵ P2
P2 ⟵ link(P2)
While end
Step 4: PRINT data(p2)
Step 5: link(P1) ⟵ NULL
Free(P2)
Step 6: STOP

Deleting node from end



| Operation removeFromBack represented graphically.

Operations on Linked Lists

Insertion: Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list

Deletion: Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.

Searching: Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

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

struct Node     {
                        int data;
                        struct Node *next;      };

void deleteStart  (struct Node** head)        {
struct Node* temp = *head;
// If head is NULL it means Singly Linked List is empty
if(*head == NULL){
printf("Impossible to delete from empty Singly Linked List");
return;
}
 // move head to next node

*head = (*head)->next;
printf("Deleted: %d\n", temp->data);
free(temp);
            }
```

```
void insertStart(struct Node** head, int data){   // dynamically create memory for this new Node
struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));       // assign data value
newNode->data = data;
// change the next node of this new Node     // to current head of Linked List
newNode->next = *head;
//re-assign head to this newNode
*head = newNode;
printf("Inserted %d\n",newNode->data);                }

void display(struct Node* node){
printf("\nLinked List: ");                    // as linked list will end when Node is Null
while(node!=NULL){   printf("%d ",node->data);
node = node->next;
}                    printf("\n");     }

int main()
{          struct Node* head = NULL;
insertStart(&head,100);          insertStart(&head,80);          insertStart(&head,60);
          insertStart(&head,40);
insertStart(&head,20);          display(head);          deleteStart(&head);
          deleteStart(&head);
display(head);          return 0;          }
```

Output

```
Inserted 100

Inserted 80

Inserted 60

Inserted 40

Inserted 20


Linked List: 20 40 60 80 100

Deleted: 20

Deleted: 40

Linked List: 60 80 100
```

Linear Search

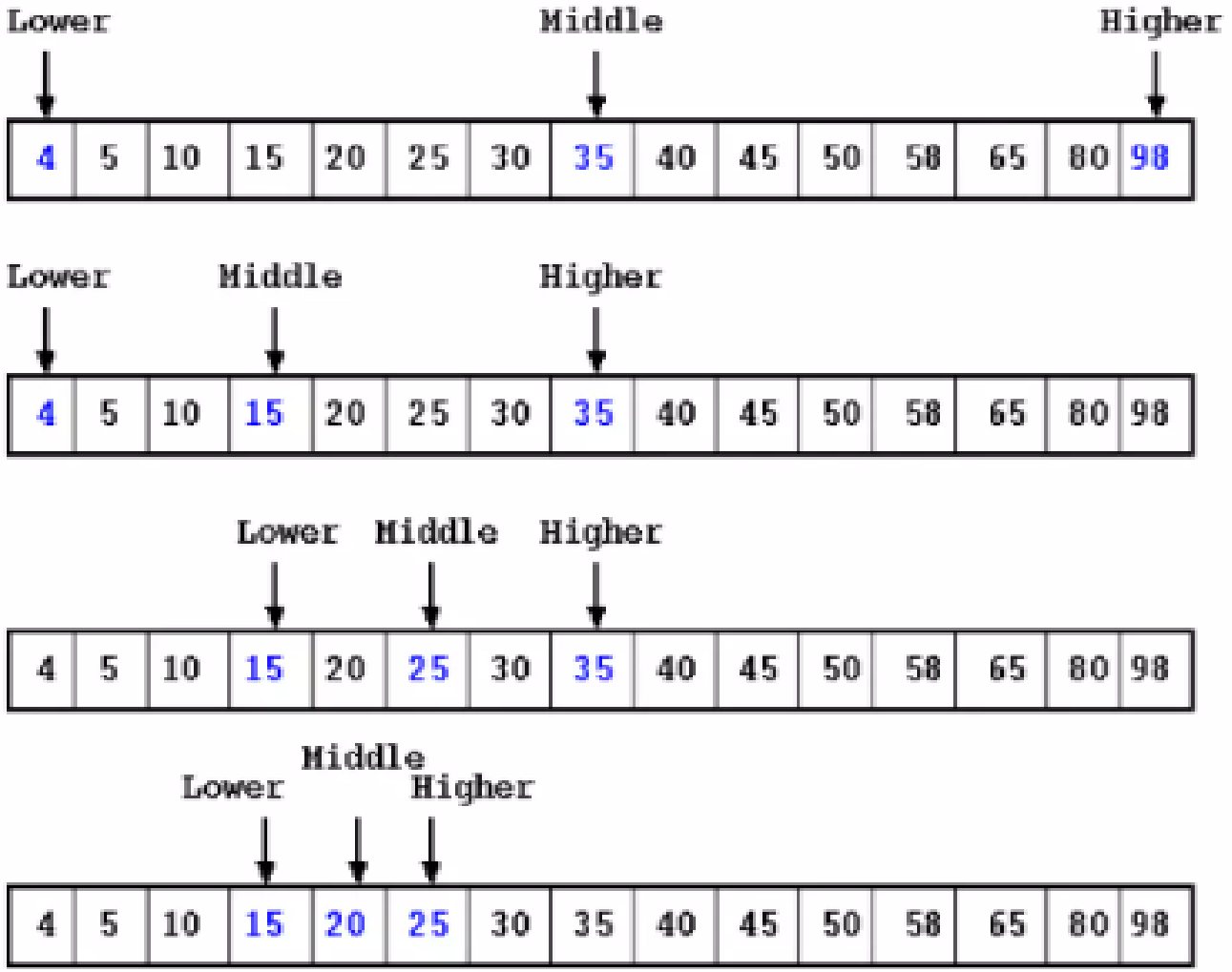
Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Binary Search

- The binary search algorithm can be used with only sorted list of elements.
- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step



Binary Search

- **Example**: Consider the following elements stored in an array and we are searching for the element 67. The trace of the algorithm is given below.

	BEG & END	MID = (BEG+END)/2	Compare	Location
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEG MID END	BEG = 0 END = 4	MID = (0+4)/2 MID = 2	67>39 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 3 END = 4	MID = (3+4)/2 MID = 3	67>47 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the muddle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 4 END = 4	MID = (4+4)/2 MID = 4	67>57 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 ENDBEG	BEG = 5 END = 4	Since the condition (BEG <= END) is false the comparison ends		
We get the output as 67 Not Found				

Binary Search

```
Procedure binary_search
    A ← sorted array
    n ← size of array
    x ← value to be searched

    Set lowerBound = 1
    Set upperBound = n

    while x not found
        if upperBound < lowerBound
            EXIT: x does not exists.

        set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

        if A[midPoint] < x
            set lowerBound = midPoint + 1

        if A[midPoint] > x
            set upperBound = midPoint - 1

        if A[midPoint] = x
            EXIT: x found at location midPoint
    end while

end procedure
```

Searching

- **Difference between Linear Search and Binary Search**

	Linear Search	Binary Search
1	This can be used in sorted and unsorted array	This can be used only in sorted array
2	Array elements are accessed sequentially	One must have direct access to the middle element in the sub list.
3	Access is very slow	Access is faster.
4	This can be used in single and multi dimensional array	Used only in single dimensional array.
5	This technique is easy and simple in implementing	Complex in operation

Sorting

➢ **Sorting the elements in an array:**

- ***Sorting is the arrangement of elements of the array in some order.***
- There are different sorting methods like Bubble Sort, Selection Sort, Shell Sort, Quick sort, Heap Sort, Insertion Sort etc.

➢ **Insertion Sort:**

- In Insertion sort, the first element of the array is assumed to be in the correct position next element is considered as the key element and compared with the elements before the key element and is inserted in its correct position.
- **Example**: Consider the following array contains 8 elements as follows:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
45	26	23	56	29	36	12	4

Pass	Location	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
I=1	J=0	45	26	23	56	29	36	12	4
I=2	J=0	26	45	23	56	29	36	12	4
I=3	J=3	23	26	45	56	29	36	12	4
I=4	J=2	23	26	45	56	29	36	12	4
I=5	J=3	23	26	29	45	56	36	12	4
I=6	J=0	23	26	29	36	45	56	12	4
I=7	J=0	12	23	26	29	36	45	56	4
Sorted List		4	12	23	26	29	36	45	56

Insertion Sort

- **ALGORITHM: Insertion Sort (A, N) A is an array with N unsorted elements.**
  - Step 1: for I=1 to N-1
  - Step 2: J = I

```
While(J >= 1)
if ( A[J] < A[J-1] ) then
        Temp = A[J];
        A[J] = A[J-1];
        A[J-1] = Temp;
[End if]
J = J-1
[End of While loop]
[End of For loop]
```

  - Step 3: Exit

Merging from Array

- **Merging:** It is used to combine the data items of two sorted files into single file in the sorted form

We have sorted linear array A as below:

1	2	3	4	5	6
10	40	50	80	95	100

And sorted linear array B as below:

1	2	3	4
20	35	45	90

After merging merged array C is as below:

1	2	3	4	5	6	7	8	9	10
10	20	35	40	45	50	80	90	95	100

Two dimensional array

- ***A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[3] [3])***
- The elements are stored in continuous memory locations.
- The elements of two-dimensional array as rows and columns.
- The number of rows and columns in a matrix is called as the order of the matrix and denoted as mxn.
- The number of elements can be obtained by multiplying number of rows and number of columns.

	A[0]	A[1]	A[2]
A[0]	10	20	30
A[1]	40	50	60
A[2]	70	80	90

Representation of Two Dimensional Array:

- A is the array of order m x n. To store m*n number of elements, we need m*n memory locations.
- The elements should be in contiguous memory locations.
- There are two methods:
  - o Row-major method
  - o Column-major method

Two Dimensional Array:

- Row-Major Method: All the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on. Ex: A[Row][Col]
- Column-Major Method: All the first column elements are stored in sequential memory locations and then all the second-column elements are stored and so on. Ex: A [Col][Row]

1000	10	A[0][0]
1002	20	A[0][1]
1004	30	A[0][2]
1006	40	A[1][0]
1008	50	A[1][1]
1010	60	A[1][2]
1012	70	A[2][0]
1014	80	A[2][1]
1016	90	A[2][2]

Row-Major Method

1000	10	A[0][0]
1002	40	A[1][0]
1004	70	A[2][0]
1006	20	A[0][1]
1008	50	A[1][1]
1010	80	A[2][1]
1012	30	A[0][2]
1014	60	A[1][2]
1016	90	A[2][2]

Col-Major Method

Advantages of Array:

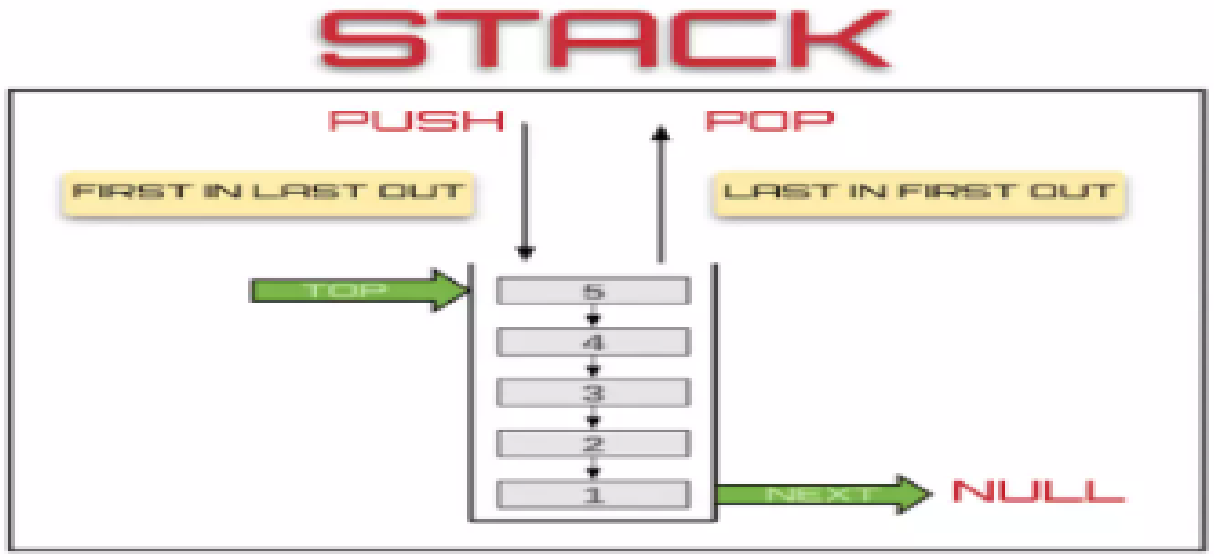
- It is used to represent multiple data items of same type by using single name.
- It can be used to implement other data structures like linked lists, stacks, queues, tree, graphs etc.
- Two-dimensional arrays are used to represent matrices.
- Many databases include one-dimensional arrays whose elements are records.

Disadvantages of Array

- We must know in advance the how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or decreased.
- Array is fixed size; if we allocate more memory than requirement then the memory space will be wasted.
- The elements of array are stored in consecutive memory locations. So insertion and deletion are very difficult and time consuming.

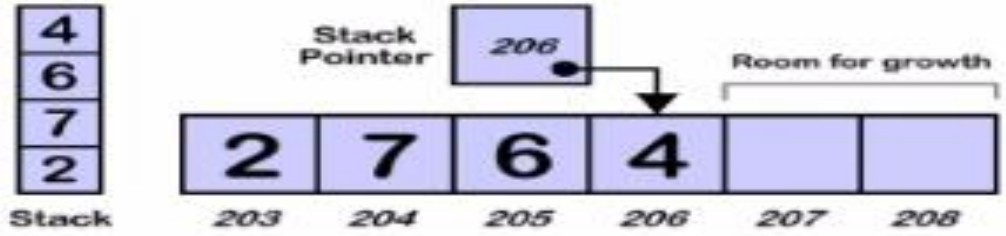
Stack

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

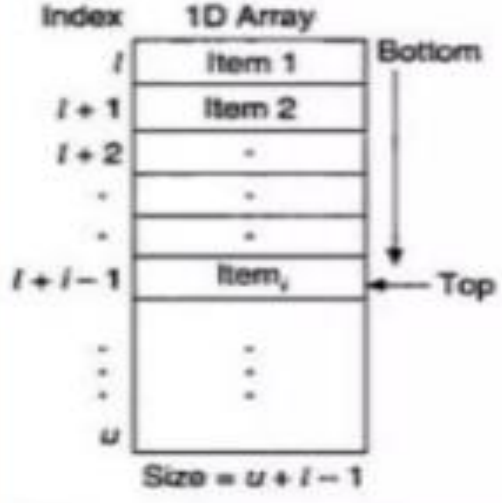


Representation of Stack in Memory

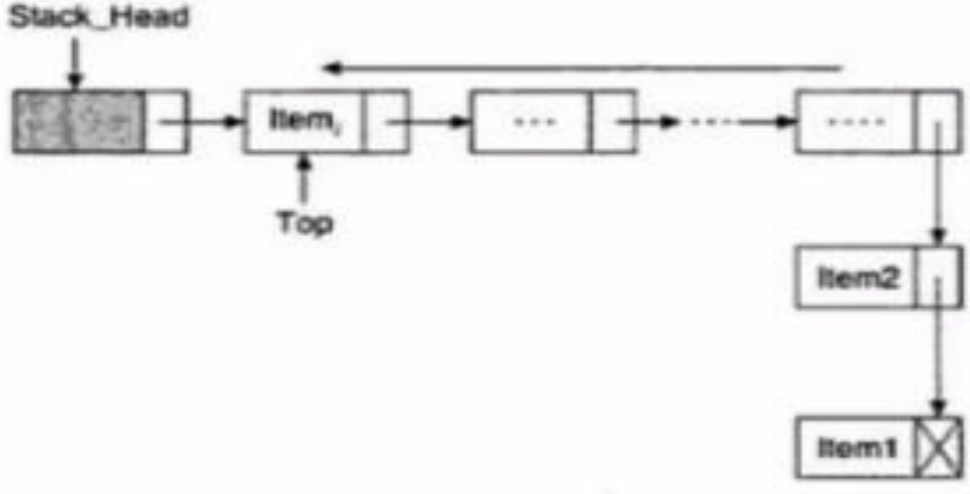
- The stack can be implemented into two ways:



  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)



(a) Array representation of a stack



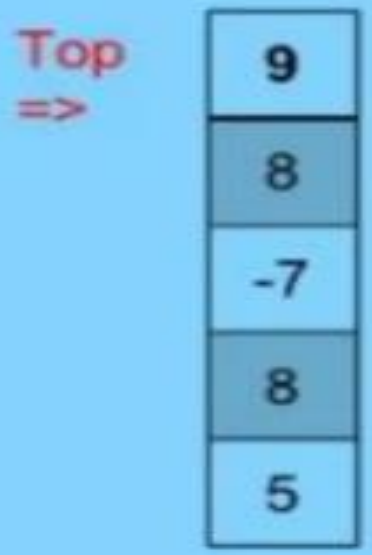
(b) Linked list representation of a stack

Operation on Stacks:

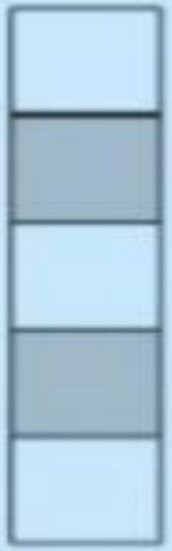
- Stack( ): It creates a new stack that is empty. It needs no parameter and returns an empty stack.
- push(item): It adds a new item to the top of the stack.
- pop( ): It removes the top item from the stack.
- peek( ): It returns the top item from the stack but does not remove it.
- isEmpty( ): It tests whether the stack is empty.
- size( ): It returns the number of items on the stack.

Stack Conditions

- Depending on implementation, may be necessary to check if stack is full -- attempt to add item to a full stack is an overflow error

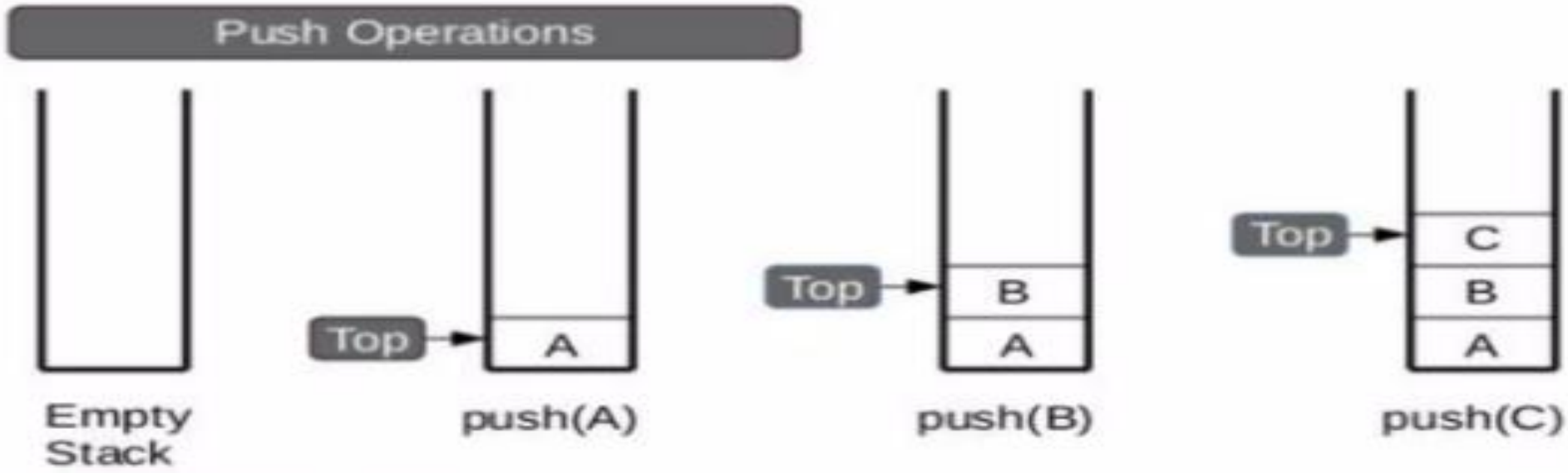


- Important to know if stack is empty -- attempt to remove an item from an empty stack is an underflow error.



PUSH Operation

- *The process of adding one element or item to the stack is represented by an operation called as the PUSH operation.*



OVERFLOW STATE

If the stack is full and does not contain enough space to accept the given item

PUSH Operation:

- ***The process of adding one element or item to the stack is represented by an operation called as the PUSH operation.***
- The new element is added at the topmost position of the stack.

ALGORITHM:

PUSH (STACK, TOP, SIZE, ITEM)

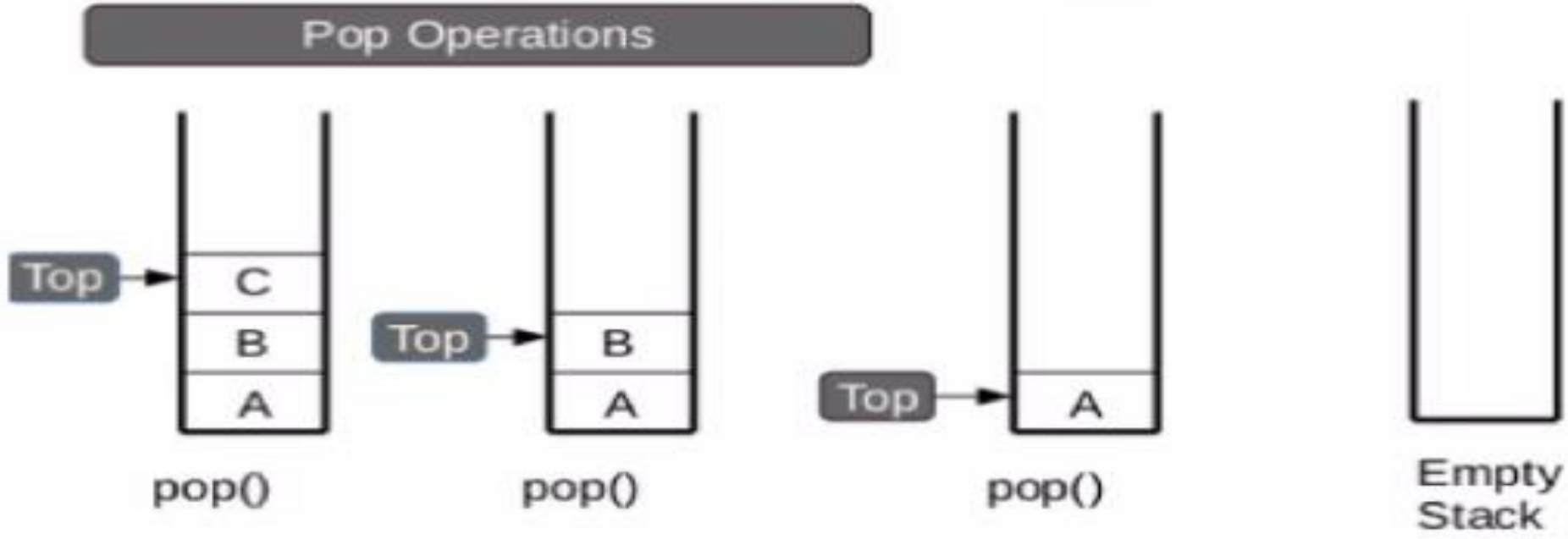
STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be inserted.

```
Step 1: if TOP = N then                    [Check Overflow]
                PRINT " STACK is Full or Overflow"
                Exit
        [End if]
Step 2: TOP = TOP + 1                      [Increment the TOP]
Step 3: STACK[TOP] = ITEM                  [Insert the ITEM]
Step 4: Return
```

POP Operation

The process of deleting one element or item from the stack is represented by an operation called as the POP operation.

When elements are removed continuously from a stack, it shrinks at same end i.e., ***top***



POP Operation

The process of deleting one element or item from the stack is represented by an operation called as the POP operation.

ALGORITHM: POP (STACK, TOP, ITEM)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be inserted.

Step 1: if TOP = 0 then [Check Underflow]

PRINT " STACK is Empty or Underflow"

Exit

[End if]

Step 2: ITEM = STACK[TOP] [copy the TOP Element]

Step 3: TOP = TOP - 1 [Decrement the TOP]

Step 4: Return

PEEK Operation

The process of returning the top item from the stack but does not remove it called as the POP operation.

ALGORITHM: PEEK (STACK, TOP)
STACK is the array with N elements. TOP is the pointer to the top of the element of the array.
Step 1: if TOP = NULL then [Check Underflow]
PRINT " STACK is Empty or Underflow"
Exit
[End if]
Step 2: Return (STACK[TOP] [Return the top element of the stack]
Step 3:Exit

Application of Stacks

- It is used to reverse a word. You push a given word to stack – letter by letter and then pop letter from the stack.
- "Undo" mechanism in text editor.
- Backtracking: This is a process when you need to access the most recent data element in a series of elements. Once you reach a dead end, you must backtrack.
- Language Processing: Compiler' syntax check for matching braces in implemented by using stack.
- Conversion of decimal number to binary.
- To solve tower of Hanoi.
- Conversion of infix expression into prefix and postfix.
- Quick sort
- Runtime memory management.

Arithmetic Expression

- An expression is a combination of operands and operators that after evaluation results in a single value.

· Operand consists of constants and variables.

· Operators consists of {, +, -, *, /,),] etc.

- Expression can be

Infix Expression: If an operator is in between two operands, it is called infix expression.

- Example: a + b, where a and b are operands and + is an operator.

Postfix Expression: If an operator follows the two operands, it is called postfix expression.

- Example: ab +

Prefix Expression: an operator precedes the two operands, it is called prefix expression.

- Example: +ab

Arithmetic Expression

Three notations for the given arithmetic expression are listed below:

Infix: ((A + ((B ^ C) – D)) * (E – (A/C)))

Prefix: * + A – ^ BCD – E/AC

Postfix: ABC ^ D – + EAC / – *

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:

Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

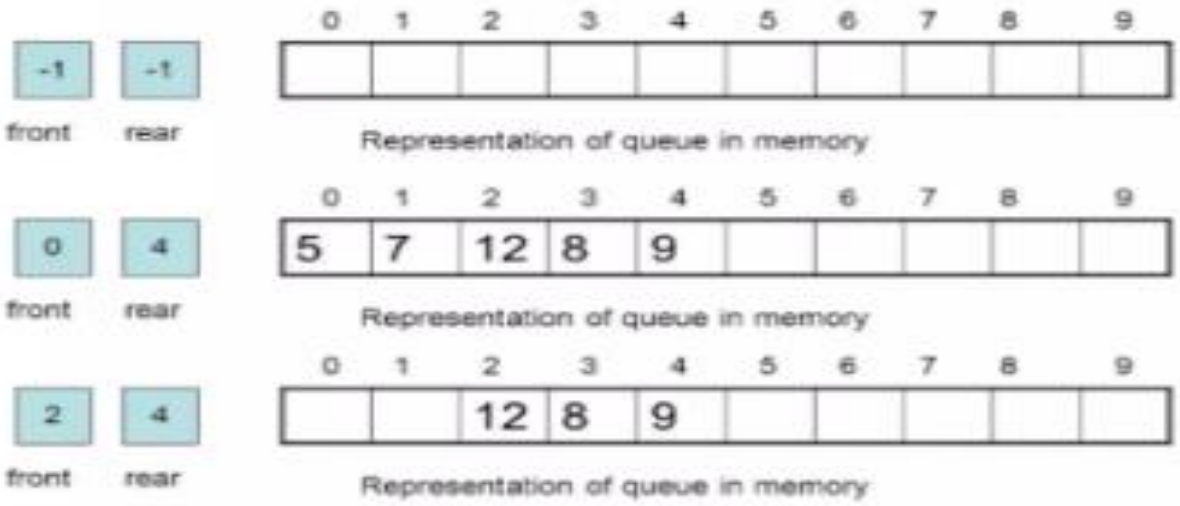
Queue

- *A queue is an ordered collection of items where an item is inserted at one end called the "rear" and an existing item is removed at the other end, called the "front".*
- Queue is also called as FIFO list i.e. First-In First-Out.
- In the queue only two operations are allowed enqueue and dequeue.
- Enqueue means to insert an item into back of the queue.
- Dequeue means removing the front item.The people standing in a railway reservation row are an example of queue.

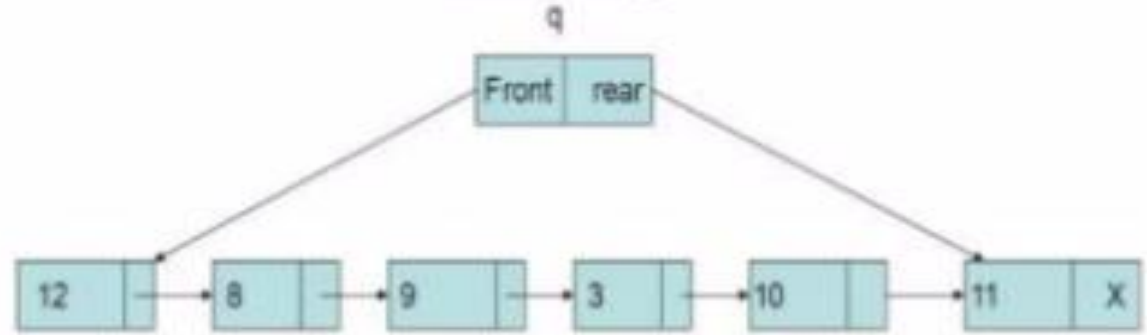
Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Array representation of linear queue



Representation of a queue in memory



Types of Queues

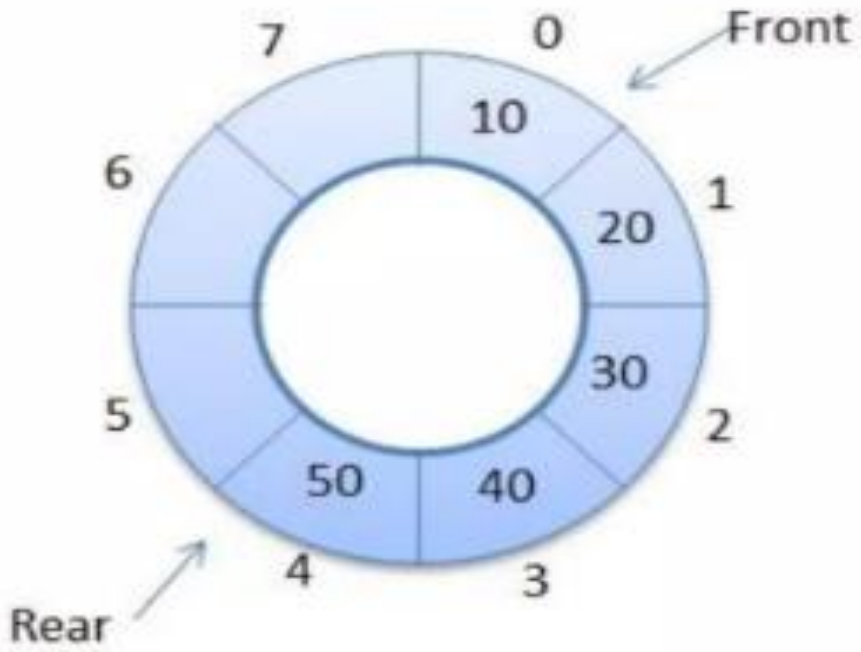
- Queue can be of four types:
  - o Simple Queue
  - o Circular Queue
  - o Priority Queue
  - o De-queue (Double Ended Queue)

Simple Queue

- Simple Queue: In simple queue insertion occurs at the rear end of the list and deletion occurs at the front end of the list.

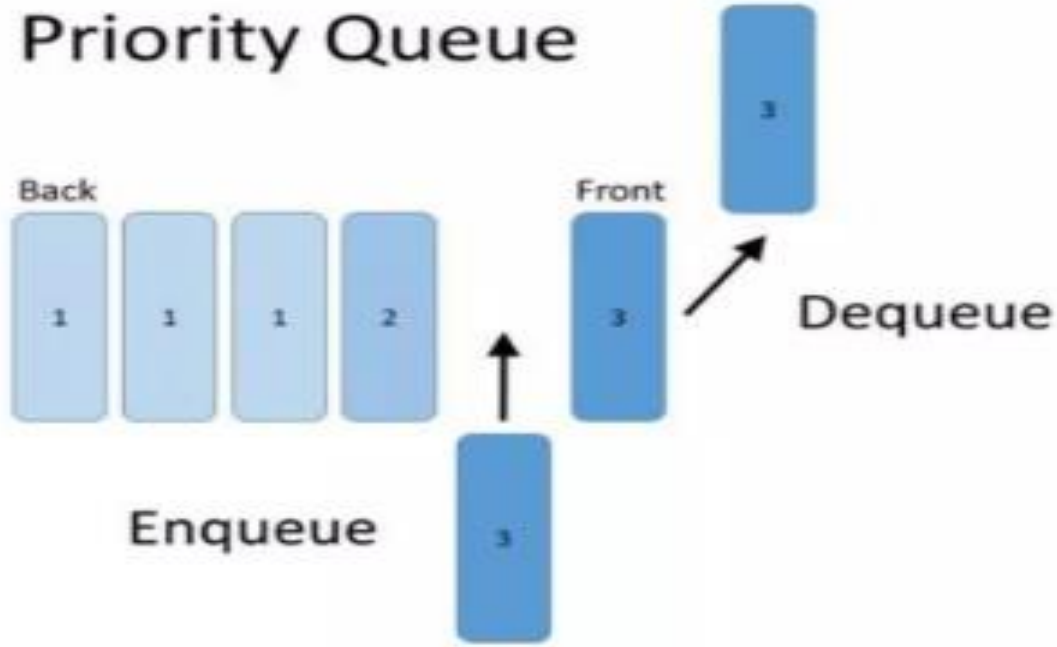
Circular Queue

- Circular Queue: A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.



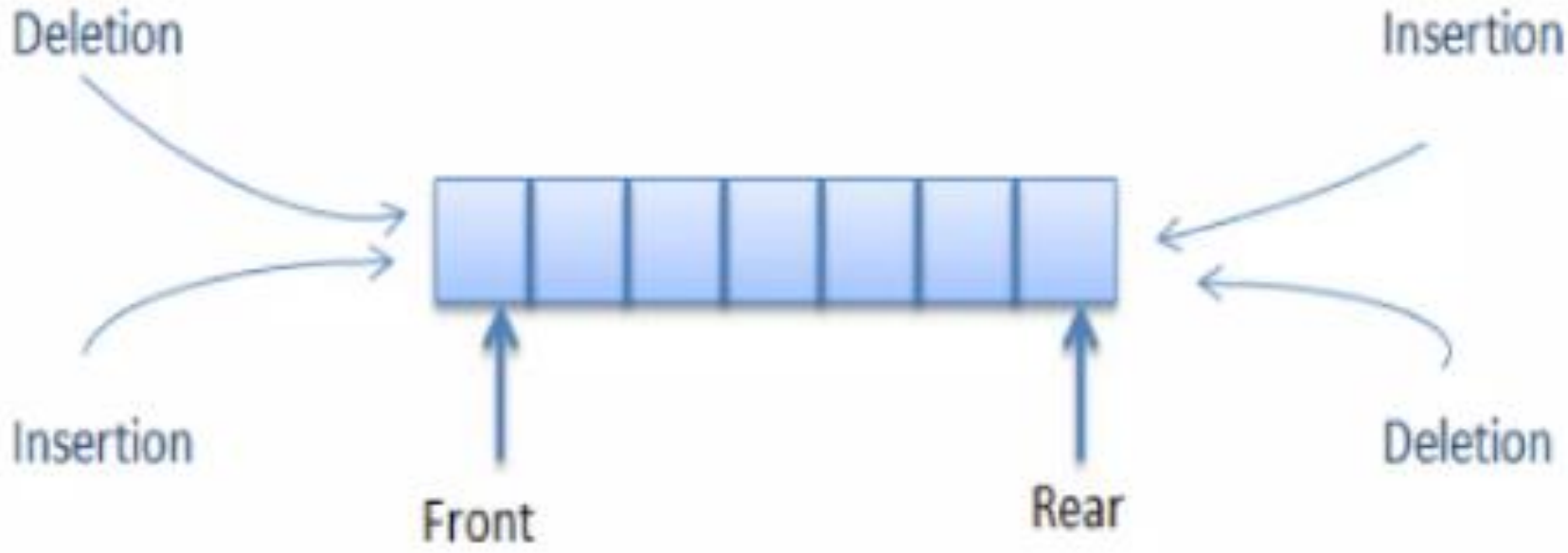
Priority Queue

- **A priority queue is a** queue that contains items that have some present priority. An element can be inserted or removed from any position depending upon some priority.



Dequeue Queue

- Dequeue: It is a queue in which insertion and deletion takes place at the both ends.



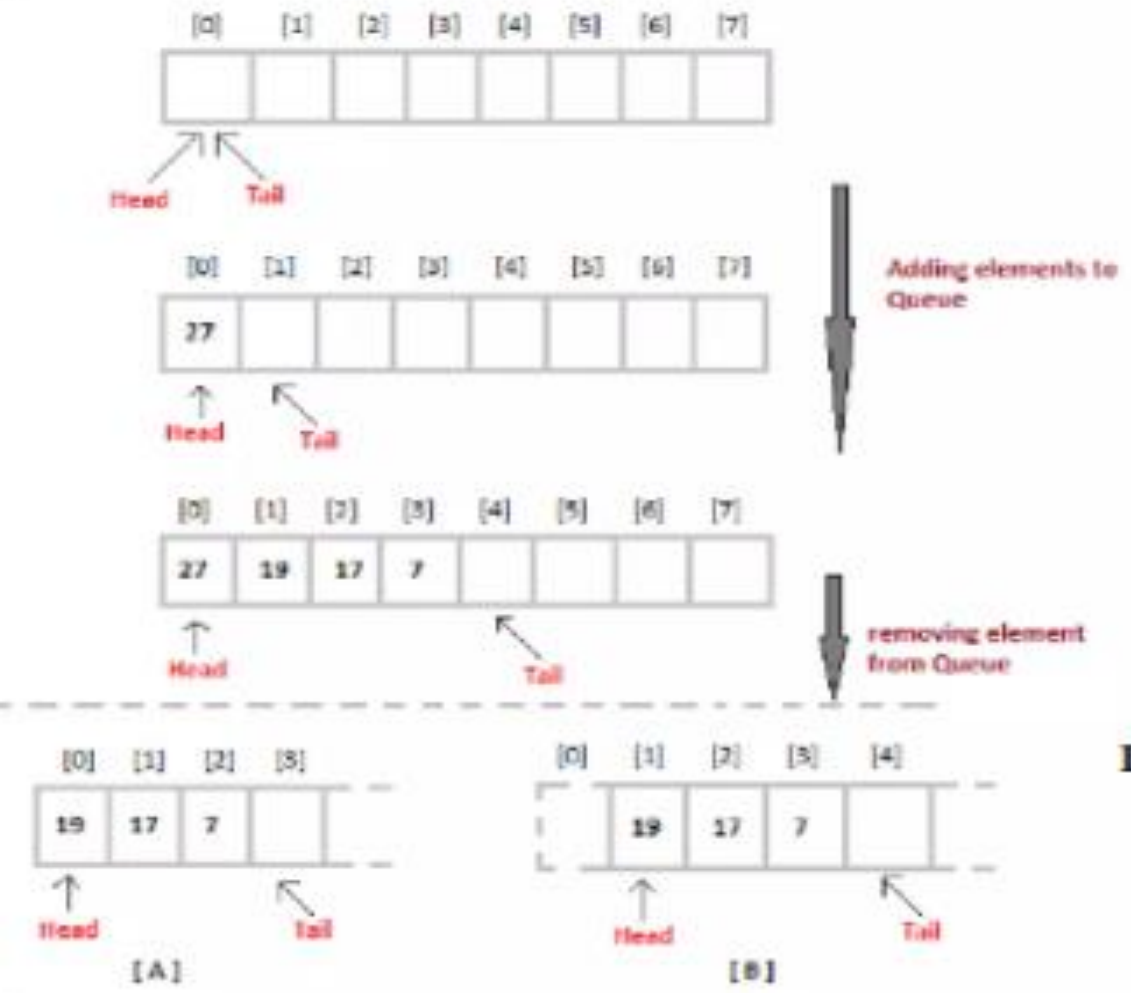
Operation on Queues

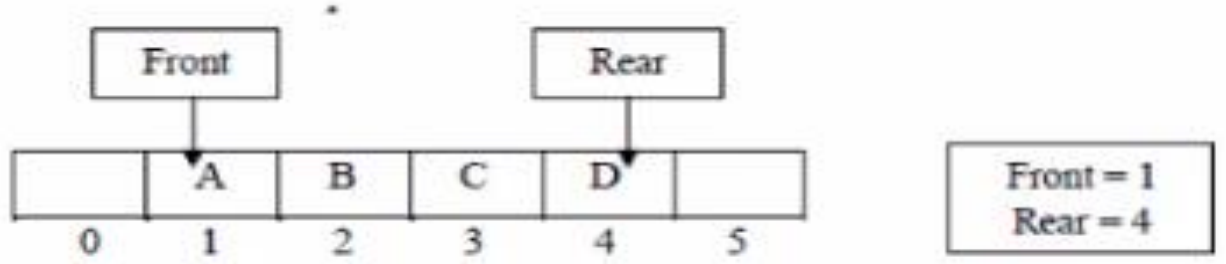
- **Queue( ):** It creates a new queue that is empty.
- **enqueue(item):** It adds a new item to the rear of the queue.
- **dequeue( ):** It removes the front item from the queue.
- **isEmpty( ):** It tests to see whether the queue is empty.
- **size( ):** It returns the number of items in the queue.

Memory Representation of a queue using array

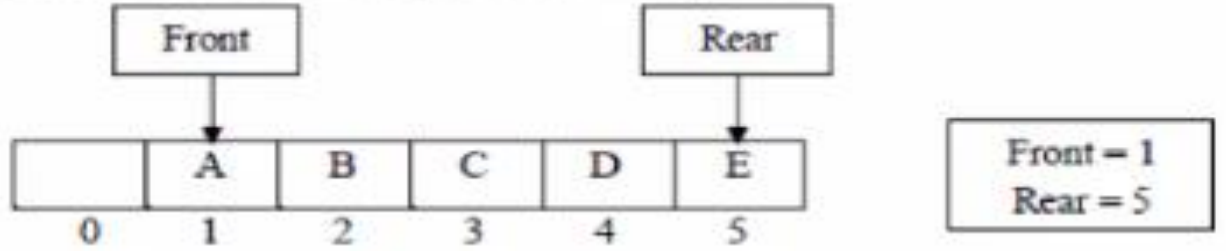
- Queue is represented in memory using linear array.
- Let QUEUE is a array, two pointer variables called FRONT and REAR are maintained.
- The pointer variable FRONT contains the location of the element to be removed or deleted.
- The pointer variable REAR contains location of the last element inserted.
- The condition FRONT = NULL indicates that queue is empty.
- The condition REAR = N-1 indicates that queue is full.

Memory Representation of a queue using array

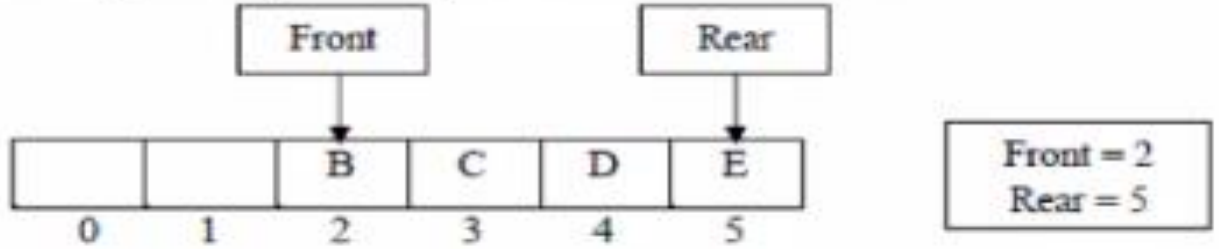




REAR = REAR + 1, QUEUE [REAR] = 'E'



ITEM = QUEUE [FRONT], FRONT = FRONT + 1



Queue Insertion Operation (ENQUEUE):

- **ALGORITHM: ENQUEUE (QUEUE, REAR, FRONT, ITEM)**

  QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]

PRINT "QUEUE is Full or Overflow"

Exit

[End if]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12	4	18	34

Step 2: if FRONT = NULL then [Check Whether Queue is empty]

FRONT = -1

REAR = -1

else

REAR = REAR + 1 [Increment REAR Pointer]

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

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Step 3: QUEUE[REAR] = ITEM [Copy ITEM to REAR position]

Step 4: Return

Queue Deletion Operation (DEQUEUE)

ALGORITHM: DEQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if FRONT = NULL then [Check Whether Queue is empty]

PRINT "QUEUE is Empty or Underflow"

Exit

[End if]

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

Step 2: ITEM = QUEUE[FRONT]

Step 3: if FRONT = REAR then [if QUEUE has only one element]

FRONT = NULL

REAR = NULL

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8				

else

FRONT = FRONT + 1 [Increment FRONT pointer]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Step 4: Return

Application of Queue

- Simulation
- Various features of Operating system
- Multi-programming platform systems.
- Different types of scheduling algorithms
- Round robin technique algorithms
- Printer server routines
- Various application software's is also based on queue data structure.

Non-Linear Data structures

- ***A Non-Linear Data structures is a data structure in which data item is connected to several other data items.***
- The data items in non-linear data structure represent hierarchical relationship.
- Each data item is called node.
- The different non-linear data structures are
  - Trees
  - Graphs.

Trees

- *A tree is a data structure consisting of nodes organized as a hierarchy.*
- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

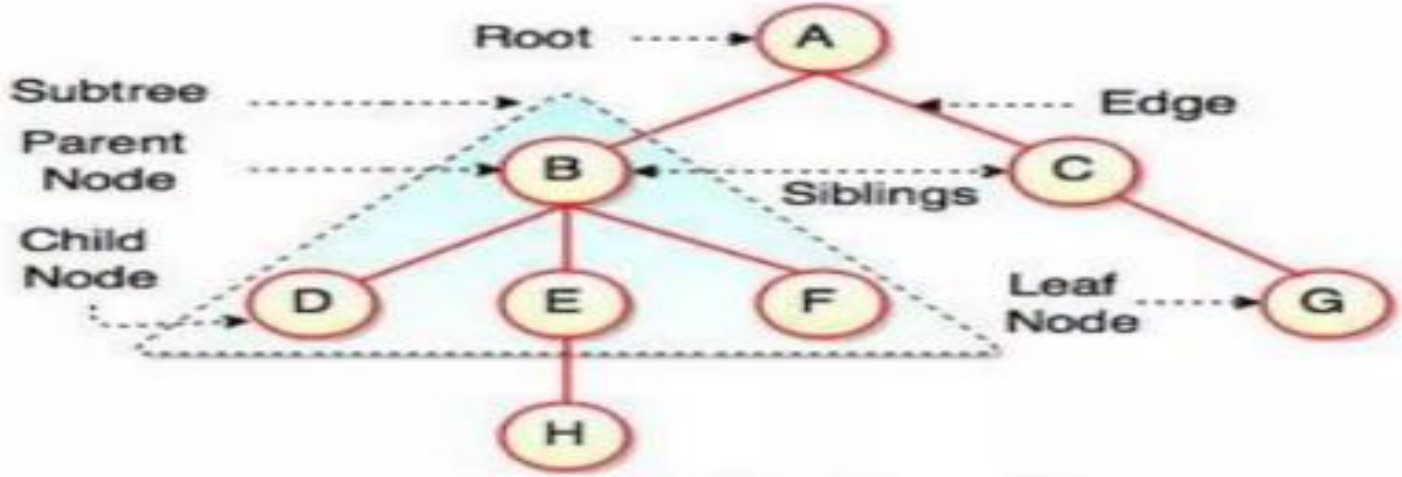
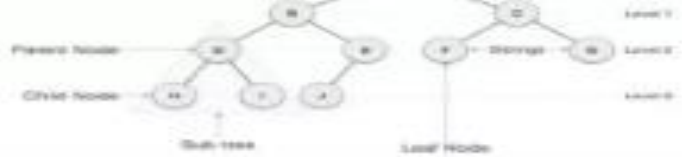


Fig. Structure of Tree

Terminology of a Tree



Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

Binary Tree

- *A binary tree is an ordered tree in which each internal node can have maximum of two child nodes connected to it.*
- A binary tree consists of:
  - A node (called the root node)
  - Left and right sub trees.
- A Complete binary tree is a binary tree in which each leaf is at the same
- distance from the root i.e. all the nodes have maximum two subtrees.

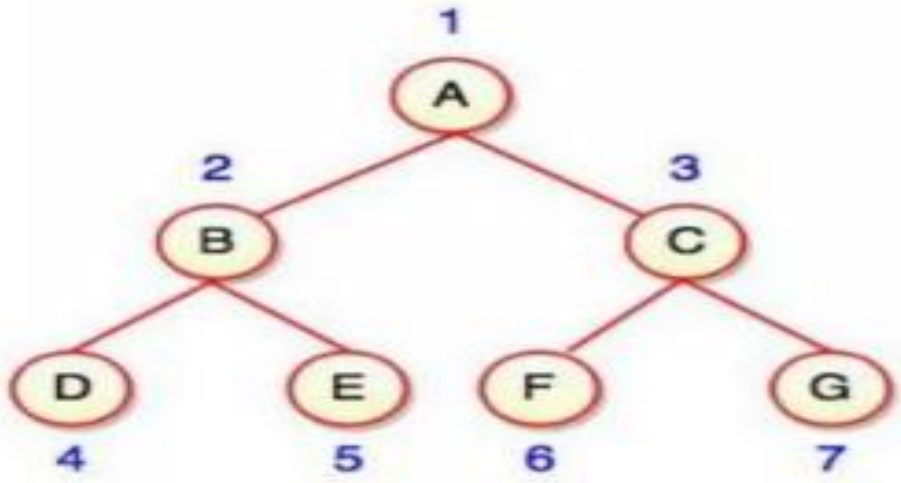


Fig. Binary Tree using Array

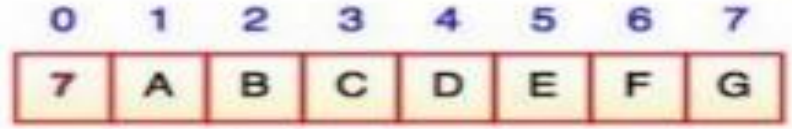


Fig. Location Number of an Array in a Tree

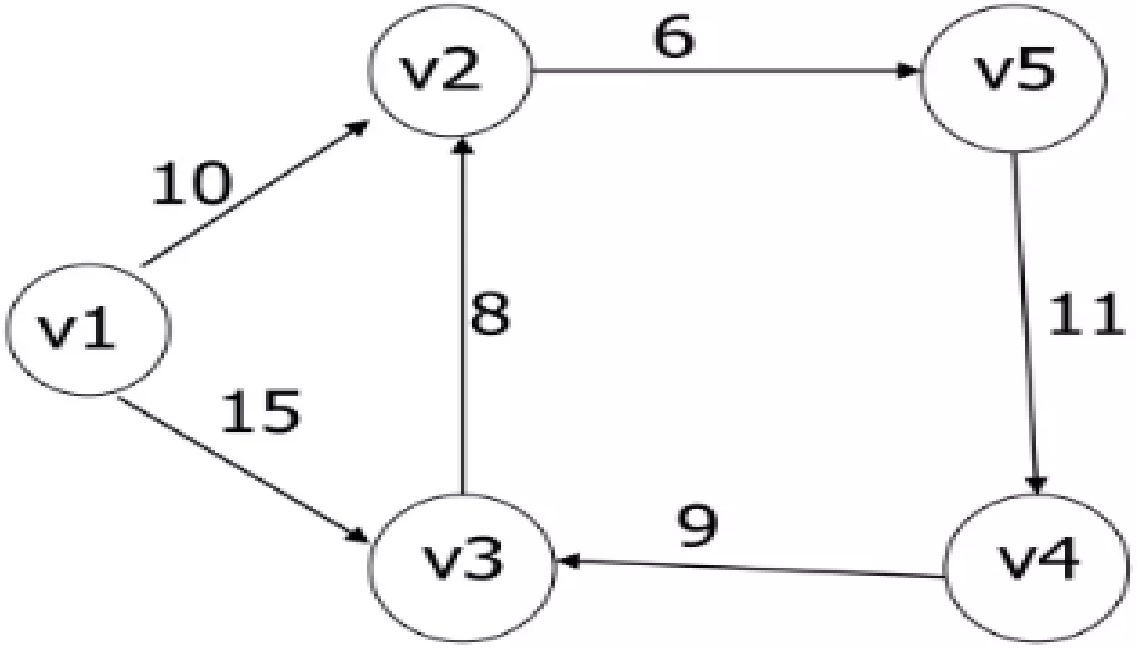
Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

Graph

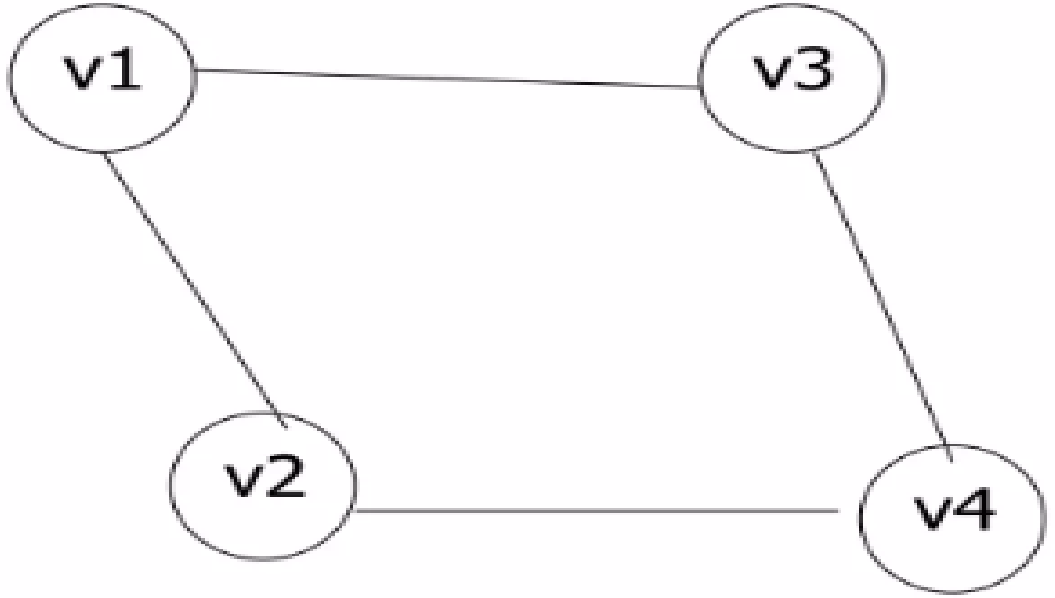
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- ***A graph is a set of vertices and edges which connect them.***
- A graph is a collection of nodes called vertices and the connection between them called edges.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

- Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Graph

- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

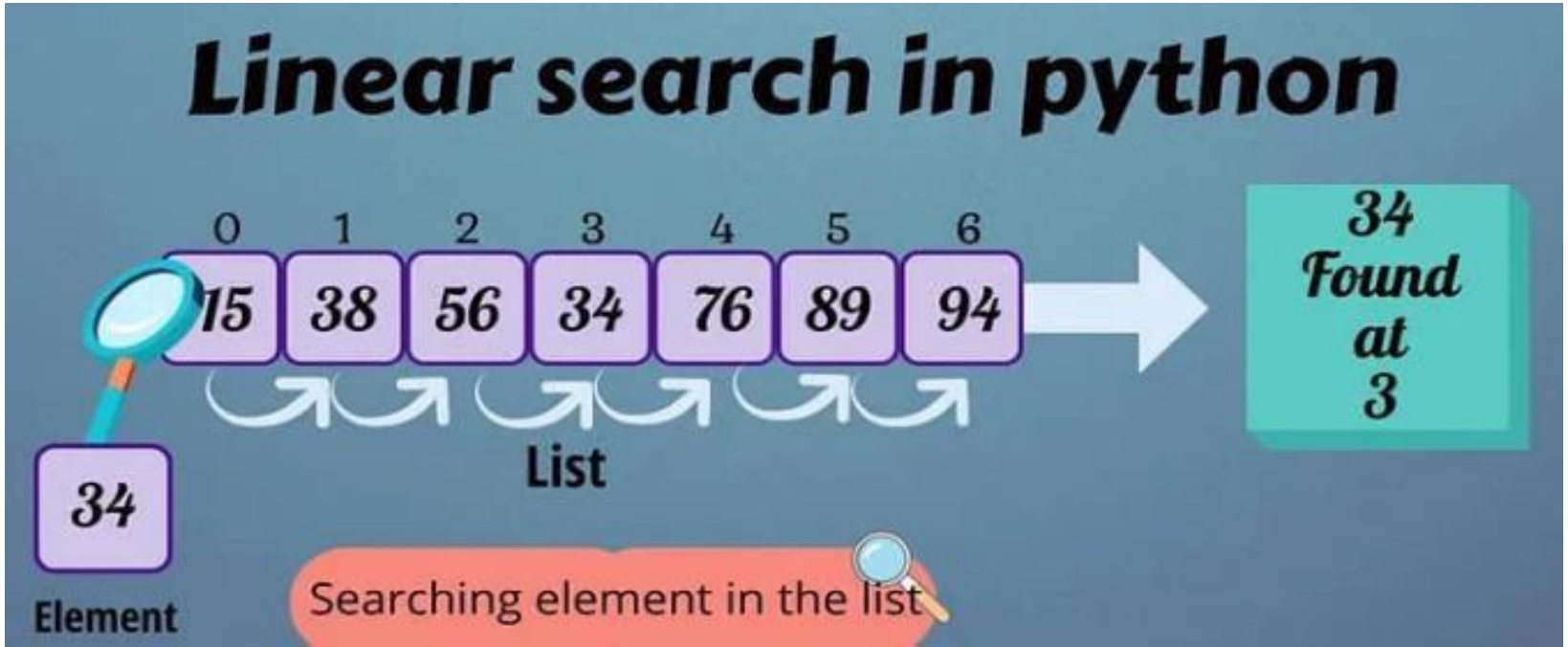
- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph

What is linear search in Python?

Linear search is the most basic type of search that is performed. It is also called the sequential search. In this search, we check each element in the given list one by one until a match is found.

This method is often used in our daily life like when we check grocery items, we do so in a linear search manner.

Technically in Python, This search method compares each and every element with the particular value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.



Algorithm of linear search:

Before writing any code of the program, we must know its algorithm. So let's understand the algorithm of this program.

1.Input a list of numbers.
2.Input the item to be searched.
3.Compare that element with each and every element of the list one by one.
4.If the match is found, then return True.
5.If the match is not found in the whole list, then return False.

#Method 1: Linear search Using range()

```
list1 = [16, 2, 7, 5, 12, 54, 21, 9,  64, 12]
print('List has the items: ', list1)
Item = int(input('Enter a number to search for:  '))
found = False
for i in range(len(list1)):
if list1[i] == Item:
found = True
print(Item, ' was found in the list at index  ', i)
break
if found == False:
print(Item, ' was not found in the list!')
```

#Output

```
List has the items: [16, 2, 7, 5, 12, 54, 21, 9, 64, 12]
Enter a number to search for: 7
        7 was found in the list at index 2
```

#Method 2: Linear search using def().

```
def linear(x,y): for i in x:

if i ==y:
   return true return false
x = [12, 37, 45, 89, 1, 2, 34, 48, 9]


linear(x,2)

linear(x,9) linear(x,5)
```

#Output

```
True
True
False
```

Binary Search

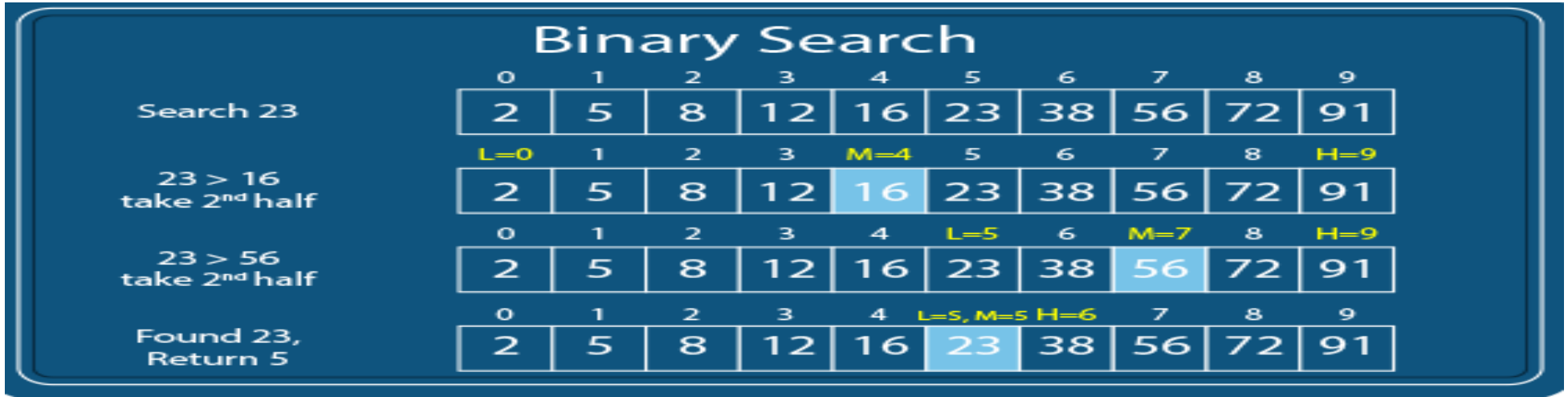
Binary search in python uses a divide-and-conquer strategy to significantly speed up searching in sorted data as opposed to linear search, which examines each element one at a time.

Binary search in C only works on sorted data sets. The collection must be arranged in ascending or descending order. This sorting enables the divide-and-conquer approach.

The key steps are:

1. Find midpoint index of the entire dataset.
2. Compare element at midpoint against target value.
3. If equal, return index of match.
4. If less, repeat search on left half.
5. If greater, repeat search on right half.

This recursive halving of search space enables tremendous efficiency gains.



We have a sorted array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91} and we want to search for the value 23.

The steps are:

1. Set lower index to 0 and the upper index to n-1.
2. Here, n is the number of elements in the array, which is 10. So lower = 0 and upper = 9 (10-1 = 9)
3. Calculate the mid index as mid = (lower + upper) / 2. In this case, mid = (0 + 9) / 2 = 4.
4. Compare the value at arr[mid] with the target value 23. Here arr[mid] is arr[4], which is 16.
5. Since 16 is less than 23, the target is greater than the mid element. So set lower = mid +

   This eliminates the left half of the array.
6. Recalculate mid index with new lower and upper bounds. mid = (lower + upper) / 2 = (5+ 9) / 2 = 7
7. Check if arr[mid] (which is 38) equals target. It does not. Since 38 > 23, set upper = mid - 1

   This eliminates the right half.
8. Recalculate mid as (lower + upper) / 2 = (5 + 6) / 2 = 5
9. Check arr[mid] (which is 23) against the target 23. It matches!
10. Return the index mid.

Concept of Binary Search

In the binary search algorithm, we can find the element position using the following methods:

1. Recursive Method
2. Iterative Method

The divide-and-conquer approach technique is followed by the recursive method. In this method, a function is called again and again until it finds an element in the list.

We have a sorted list of elements, and we are looking for the index position of 45. [12, 24, 32, 39, 45, 50, 54]

So, we are setting two pointers in our list. One pointer is used to denote the smaller value called low, and the second pointer is used to denote the highest value called high.

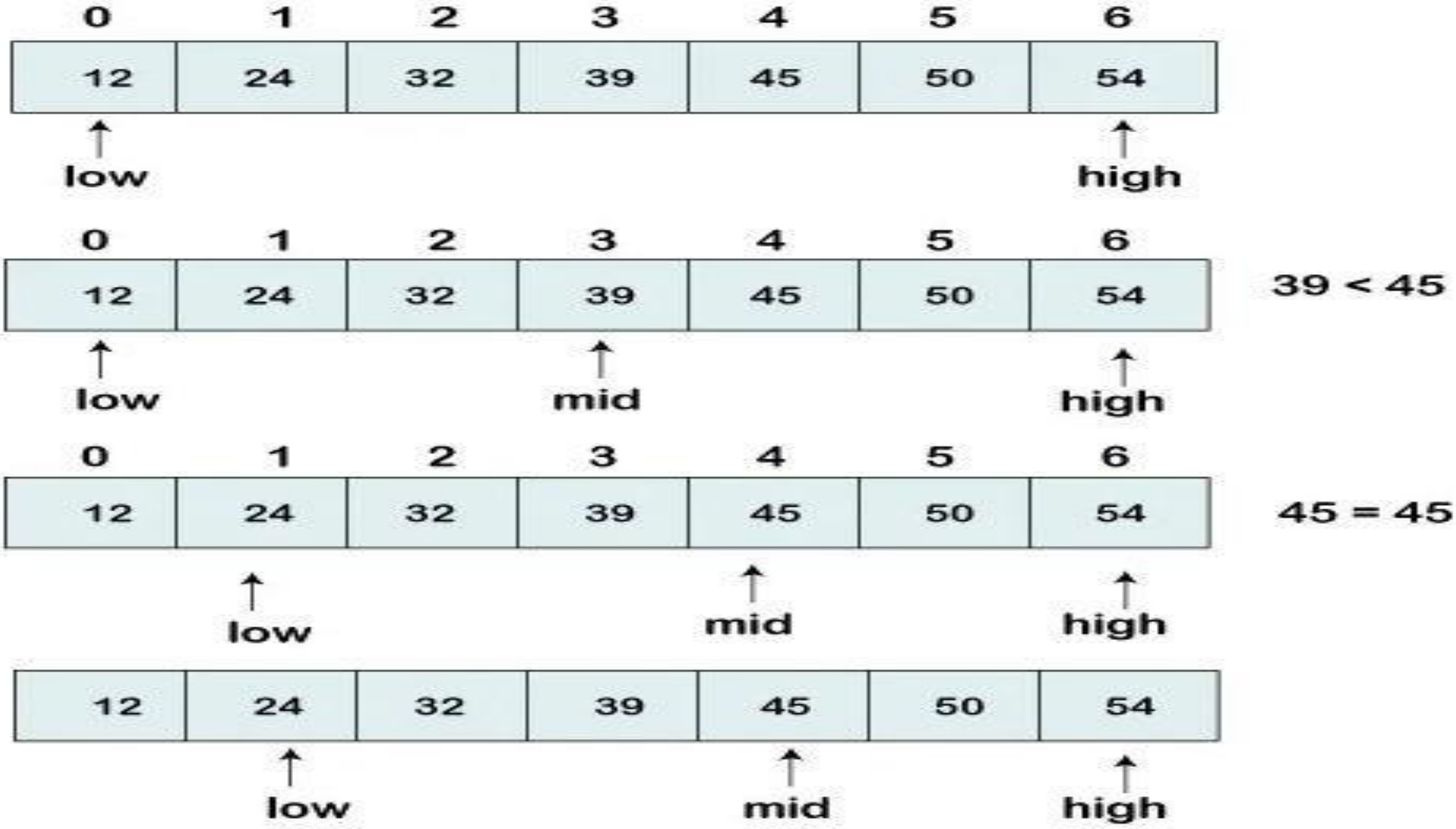
Next, we calculate the value of the middle element in the array.

Now, we will compare the searched element to the mid-index value. In this case, 32 is not equal to 45. So we need to do further comparisons to find the element.

If the number we are searching for is equal to the mid, then return the mid; otherwise, move to the next comparison.

The number to be searched is greater than the middle number, so we compare the n with the middle element of the elements on the right side of mid and set low to low = mid + 1.

Otherwise, compare the n with the middle element of the elements on the left side of mid, and set high to high = mid—1.


0 1 2 3 4 5 6
12 24 32 39 45 50 54
low
high
0 1 2 3 4 5 6
12 24 32 39 45 50 54
low
mid
high
39 < 45
0 1 2 3 4 5 6
12 24 32 39 45 50 54
low
mid
high
45 = 45
12 24 32 39 45 50 54
low
mid
high

 # Iterative Binary Search Function method Python Implementation
 # It returns index of n in given list1 if present, # else returns -1

```
def binary_search (list1, n): low = 0

high = len(list1) - 1 mid = 0

while low <= high:

# for get integer result mid = (high + low) // 2

# Check if n is present at mid if list1[mid] < n:

low = mid + 1

# If n is greater, compare to the right of mid elif list1[mid] > n:

high = mid – 1
# If n is smaller, compared to the left of mid else:
return mid

# element was not present in the list, return -1 return -1

# Initial list1
list1 = [12, 24, 32, 39, 45, 50, 54]
n = 45
```

 # Iterative Binary Search Function method Python Implementation
 # It returns index of n in given list1 if present, # else returns -1

```
# function call   result = binary_search(list1, n)
if result != -1:

print("Element is present at index", str(result))
else:
print("Element is not present in list1")
```

OUT PUT

Element is present at index 4

Explanation:

In the above program - We have created a function called binary_search() function which takes two arguments — a list to sorted and a number to be searched.

We have declared two variables to store the lowest and highest values in the list.

The low is assigned initial value to 0, high to len (list1) — 1 and mid as 0.

Next, we have declared the while loop with the condition that the lowest is equal and smaller than the highest

The while loop will iterate if the number has not been found yet.

In the while loop, we find the mid value and compare the index value to the number we are searching for.

If the value of the mid-index is smaller than n, we increase the mid value by 1 and assign it to The search moves to the left side.

Otherwise, decrease the mid value and assign it to the high. The search moves to the right side. If the n is equal to the mid value then return mid.

This will happen until the low is equal and smaller than the high.

If we reach at the end of the function, then the element is not present in the list. We return -1 to the calling function.

Recursive Binary Search:

The recursion method can be used in the binary search. In this, we will define recursive function that keeps calling itself until it meets the condition.

Python Program

Python program for recursive binary search

Returns index position of n in list1 if present, otherwise -1 def binary_search (list1, low, high, n):

Check base case for the recursive function if low <= high:

mid = (low + high) // 2

If element is available at the middle itself then return the its index if list1[mid] == n:

return mid

```
# If the element is smaller than mid value, then search moves
# left sublist1
elif list1[mid] > n:
return binary_search(list1, low, mid - 1, n)
# Else the search moves to the right sublist1 else:
return binary_search(list1, mid + 1, high, n) else:
# Element is not available in the list1 return -1

# Test list1ay
list1 = [12, 24, 32, 39, 45, 50, 54]
n = 32
# Function call
res = binary_search(list1, 0, len(list1)-1, n) if res != -1:
print("Element is present at index", str(res))
else:
print("Element is not present in list1")
```

Output:

Element is present at index 2

Explanation

The above program is similar to the previous program. We declared a recursive function and its base condition. The condition is the lowest value is smaller or equal to the highest value.

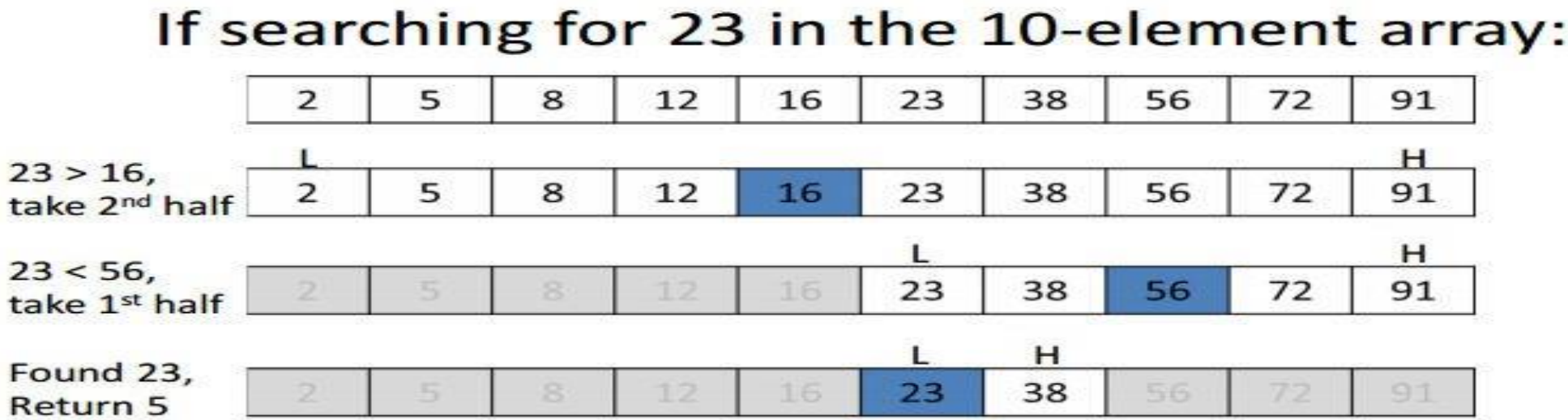
1. We calculate the middle number as in the last program.
2. We have used the if statement to proceed with the binary search.
3. If the middle value equal to the number that we are looking for, the middle value is returned.
4. If the middle value is less than the value, we are looking then our recursive function
5. binary_search() again and increase the mid value by one and assign to low.
6. If the middle value is greater than the value we are looking then our recursive function
7. binary_search() again and decrease the mid value by one and assign it to low.

In the last part, we wrote our main program. It is the same as the previous program, but the only difference is that we have passed two parameters in the binary_search() function.

This is because we can't assign the initial values to the low, high and mid in the recursive function. Every time the recursive is called the value will be reset for those variables. That will give the wrong result.

Binary Search

Binary search is very straight forward, it only works when list is sorted. I will quickly explain its working steps with an example:



1. your search value = 23. In a given list, go to its middle element which is 16. If search value is greater than middle element, take upper part of the list(after 16 until the end).
2. Again compare search value with middle element from upper part we've just taken and repeat step 1.
3. If only 2 element left, choose first element, in this case 23 as your midpoint. Compare it and since it is what we are looking for, we are done.

So main goal of binary search is finding the midpoint.

there are few different ways of choosing a midpoint

1. (lowest_index + highest_index)/2 and round it down
2. lowest_index + (highest_index - lowest_index)/2

so using the first method in our case of 10-element list. first midpoint would be 0 + 9/2 = 4.5, rounded down to 4. and 4 means index, 4th element which is 16.

Python implementation

```
import math

def binarySearch(lst, key):

low = 0

end = len(lst) -1
while low <= end:
mid = math.floor((low + end)/2)
if key == lst[mid]:
return f  'key {key} = lst[{mid}]'

elif

key < lst[mid]: end = mid - 1

else:
low = mid + 1
return f'{key} is not in this list'
```