

Module 2

DATA STRUCTURES AND ALGORITHMS

Definitions

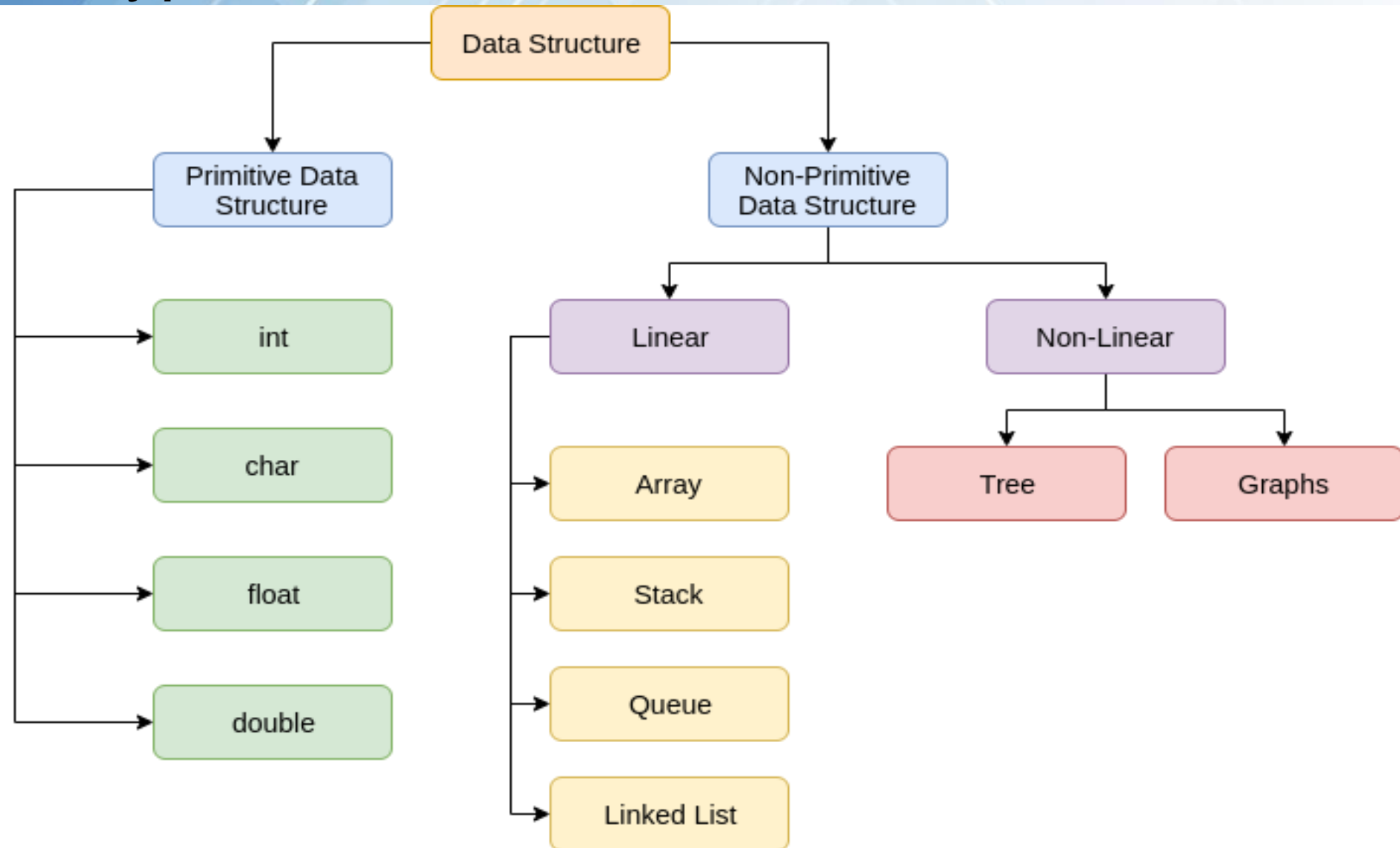
What is Data Structure?

“It is set of procedures to define, store, access and manipulate data”

or

“The organized collection of data is called data structure”

Types of Data Structures



Types of Data

- Primitive
- Non-primitive
- Records and Files
- Primitive(Elementary Data Types)
 - The data types which are processed or manipulated as a single whole (i.e, alone)
 - Example: integer, real, charater etc.
 - They are also called built-in data types

Types of Data

- Non-Primitive(Structure Data Types)
 - The items which are collection of other data structures are called non-primitive items.
 - If more than one values are combined in the single name is called non-primitive data structures
 - Example: Arrays, Stack, Queues etc.
- Types of non-Primitive Data Structures
 - Linear
 - Non-Linear

Linear Data Structures

- The DS in which there is concept of linearity b/w the components is called linear data structure.
- The components exists in a certain sequence one after another.
- They have some successor or predecessor
Example, Arrays, Stack, Queues etc.

Types of Linear Data Structures

- **Physical Linear Data Structures**

- The linear data structures whose successive components occupy consecutive memory locations are called physical linear DS.
- Example: Array

- **Logical Linear Data Structures**

- The linear DS whose components are accessed in certain sequence but they are not necessarily stored in consecutive memory locations.
- Example: Linked Lists

Queues and Stacks are both logical and physical linear data structures

Non-Linear

- The data structure in which the order of data structure does not matter and is not fixed
 - Example Tree, Graph

- Record

The collection of fields is called record.

What is Domain?

- Set of possible values of an item is called its domain
- For example, set of possible values of Boolean type is 2

Arrays

- A collection of consecutive locations having same type that can be accessed randomly. They are physical linear data structure
- Types of Arrays
 - One Dimensional Array
 - Two Dimensional Array
 - Multidimensional Array

One Dimensional Arrays

- 1-dimensional Arrays are also called vectors
 - Example, Int A[20];
 - The elements of 1-d array are stored in consecutive memory locations
 - The start of array is called its Base Address (BA)
 - Let BA denotes the base address of the array and S denotes the size for each location, then the address of i^{th} location can be accessed as:

$$A(i) = BA + (i-1) * S$$

Generally

$$A(i) = BA + (i-lb) * S$$

2-d Array

- Two dimensional array are used to store matrices
- There are two methods for storing 2-d Arrays

Row Major Order

Column Major Order

For Example `int A[3][2];`

Row Major Order

1 2 3 4 5 6

Column Major Order

1 3 5 2 4 6

1	2
3	4
5	6

3-d arrays

1D Array

3	2
---	---

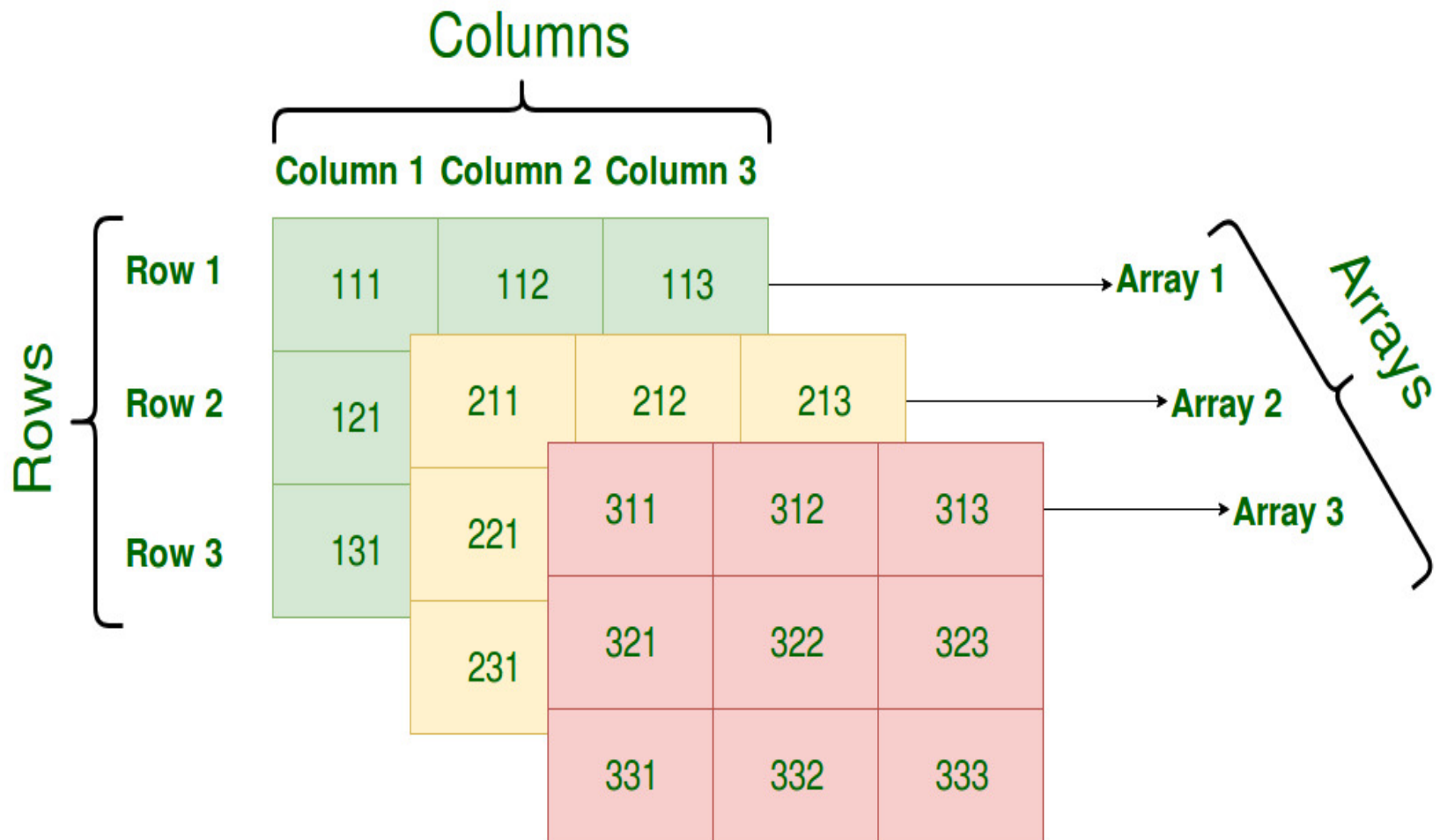
2D Array

1	0	1
3	4	1

3D Array

1	7	9
5	9	3
7	9	9

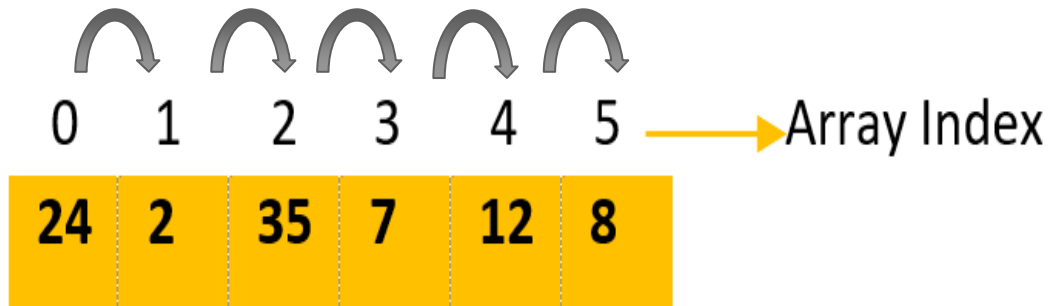
3 D Array



Operations in Array

- **Traversing** - print all the array elements one by one.
- **Insertion** - Adds an element at the given index.
- **Deletion** - Deletes an element at the given index.
- **Updation** - Updates an element at the given index.
- **Searching** - Searches an element using the given index or by the value.
- **Sorting** - Sort the elements in the array in increasing or decreasing order.

Traversing



Array of 6 integer element

Step 01: Start

Step 02: [Initialize counter variable.] Set $i = \text{LB}$.

Step 03: Repeat for $i = \text{LB}$ to UB .

Step 04: Apply process to $\text{arr}[i]$.

Step 05: [End of loop.]

Step 06: Stop

Time Complexity:

2	4	6	8	12
---	---	---	---	----

Initial Array

2	4	6	8	12
---	---	---	---	----

X i.e., 10 approaches to 5th position

Pos = 5

10

X

2	4	6	8	12	
---	---	---	---	----	--

Increasing the size of the array by one (1)

2	4	6	8		12
---	---	---	---	--	----

12 shifted to one position forward

2	4	6	8	10	12
---	---	---	---	----	----

Array with X inserted at position pos

Insertion


- Step 01: Start
- Step 02: [Reset size of the array.] set $\text{size} = \text{size} + 1$
- Step 03: [Initialize counter variable.] Set $i = \text{size} - 1$
- Step 04: Repeat Step 05 and 06 for $i = \text{size} - 1$ to $i \geq \text{pos} - 1$
- Step 05: [Move i th element forward.] set $\text{arr}[i+1] = \text{arr}[i]$
- Step 06: [Decrease counter.] Set $i = i - 1$
- Step 07: [End of step 04 loop.]
- Step 08: [Insert element.] Set $\text{arr}[\text{pos}-1] = x$
- Step 09: Stop

Deleting an element in Array

2	4	6	8	12
---	---	---	---	----

Initial Array

2	4	6	8	12
---	---	---	---	----



Pos = 3

Move each element backward by one place whose position is greater than the element you wish to delete.

2	4	8	12
---	---	---	----

Array with 6 deleted from 3rd position

Deletion

Step 01: Start

Step 02: [Initialize counter variable.] Set $i = \text{pos} - 1$

Step 03: Repeat Step 04 and 05 for $i = \text{pos} - 1$ to $i < \text{size}$

Step 04: [Move i th element backward (left).] set $a[i] = a[i+1]$

Step 05: [Increase counter.] Set $i = i + 1$

Step 06: [End of step 03 loop.]

Step 07: [Reset size of the array.] set $\text{size} = \text{size} - 1$

Step 08: Stop

Updation

Step 01: Start

Step 02: Set $a[\text{pos}-1] = x$

Step 03: Stop

1	3	5	7	9
---	---	---	---	---

Initial Array

1	3	5	7	9
---	---	---	---	---

Pos = 2

1	84	5	7	9
---	----	---	---	---

Array with 84 updated at 2nd position

Searching an element in Array

Step 01: Start

Step 02: [Initialize counter variable.] Set $i = 0$; specify the search term X

Step 03: Repeat Step 04 and 05 for $i = 0$ to $i < n$

Step 04: if $a[i] = x$, then jump to step 07

Step 05: [Increase counter.] Set $i = i + 1$

Step 06: [End of step 03 loop.]

Step 07: Print x found at $i + 1$ position and go to step 09

Step 08: Print x not found (if $a[i] \neq x$, after all the iteration of the above for loop.)

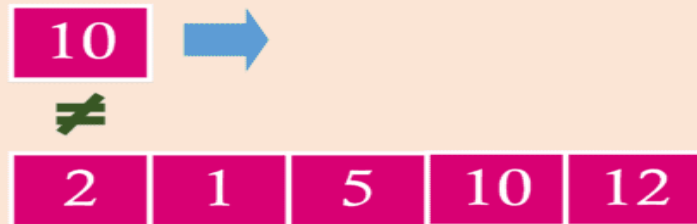
Step 09: Stop

Linear search

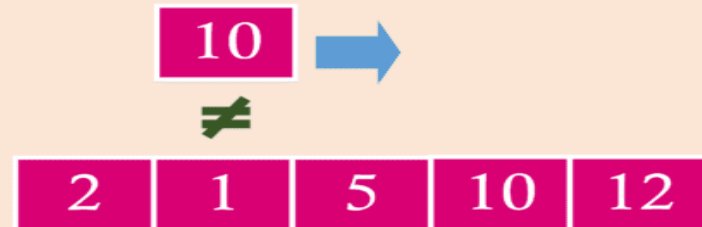
Search number **10**

List of numbers **2** **1** **5** **10** **12**

Step 1



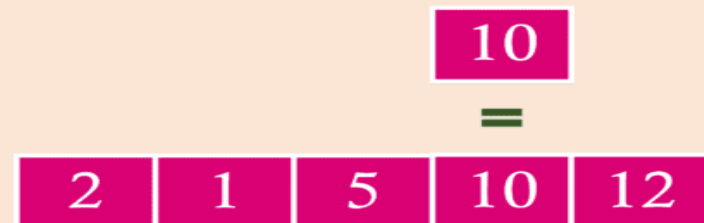
Step 2



Step 3



Step 4



Sorting

Step 01: Start

Step 02: Repeat Step 03 to 07 for $i = 0$ to $i < n$

Step 03: Repeat Step 04 and 05 for $j = i + 1$ to $j < n$

Step 04: Check a statement using the if keyword, If $\text{arr}[i] > \text{arr}[j]$, Swap $\text{arr}[i]$ and $\text{arr}[j]$.

Step 05: [Increase counter.] Set $j = j + 1$

Step 06: [End of step 03 loop.]

Step 07: [Increase counter.] Set $i = i + 1$

Step 08: [End of step 02 loop.]

Step 09: Stop

Sorting

Original Array

-5	2	33	10	-7
----	---	----	----	----

Iteration 1

-5	2	10	33	-7
----	---	----	----	----

-5	2	10	-7	33
----	---	----	----	----

Iteration 2

-5	2	-7	10	33
----	---	----	----	----

Iteration 3

-5	-7	2	10	33
----	----	---	----	----

Iteration 4

-7	-5	2	10	33
----	----	---	----	----

learnersbucket.com

Sort this array in ascending order

As 33 is greater than 10
swap 33 and 10

As 33 is greater than -7
swap 33 and -7

As 10 is greater than -7
swap 10 and -7

As 2 is greater than -7
swap 2 and -7

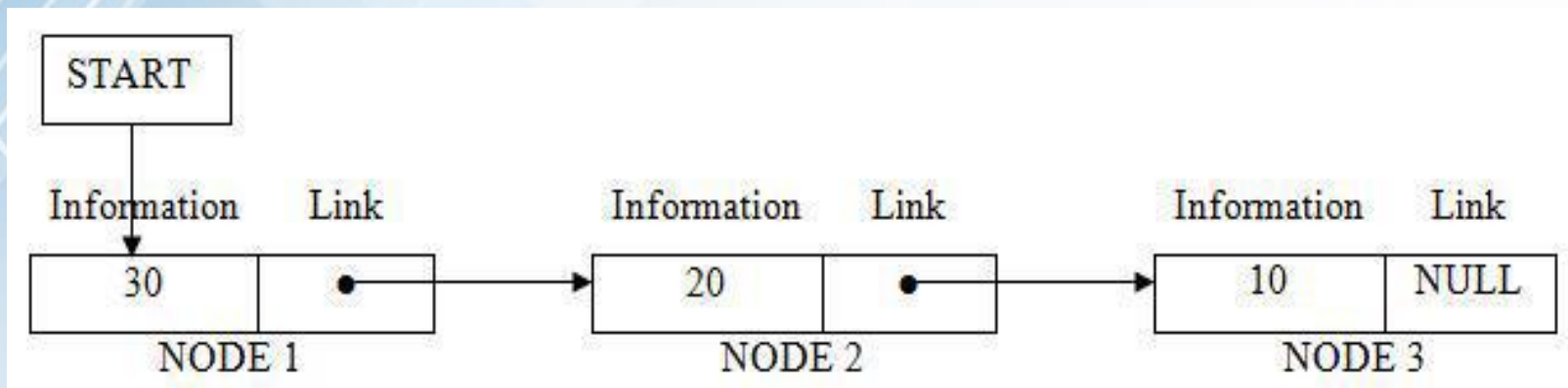
As -5 is greater than -7
swap -5 and -7

LIST

List is a Sequential data structure and consists of n no of nodes

Each nodes is divided into two parts:

- The first part contains the information of the element.
- The second part contains the memory address of the next node in the list. Also called Link part.



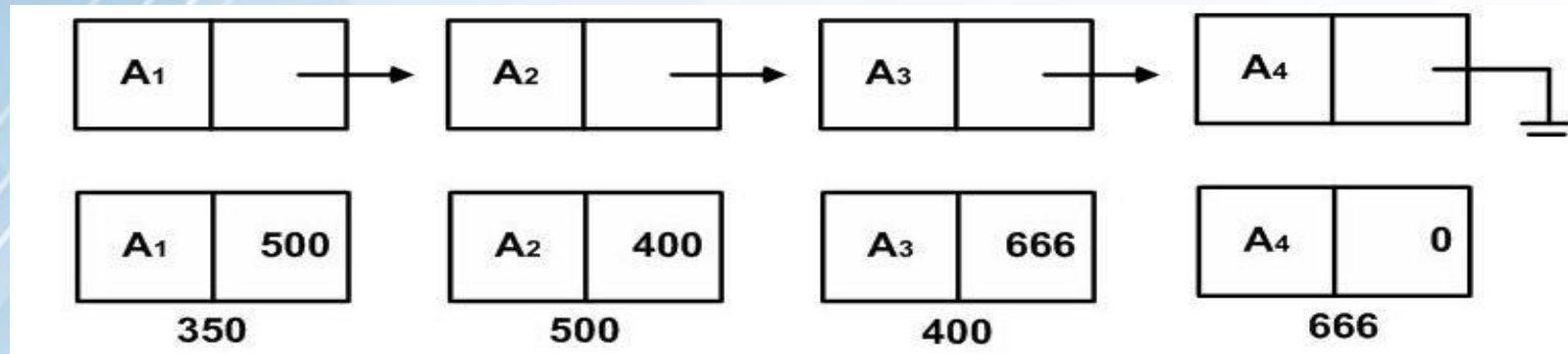
Example of operations on list

The elements of a list are 34, 12, 52, 16, 12

- Find (52) -> 3
- Insert (20, 4) -> 34, 12, 52, 20, 16, 12
- Delete (52) -> 34, 12, 20, 16, 12
- FindKth (3) -> 20

Linked List

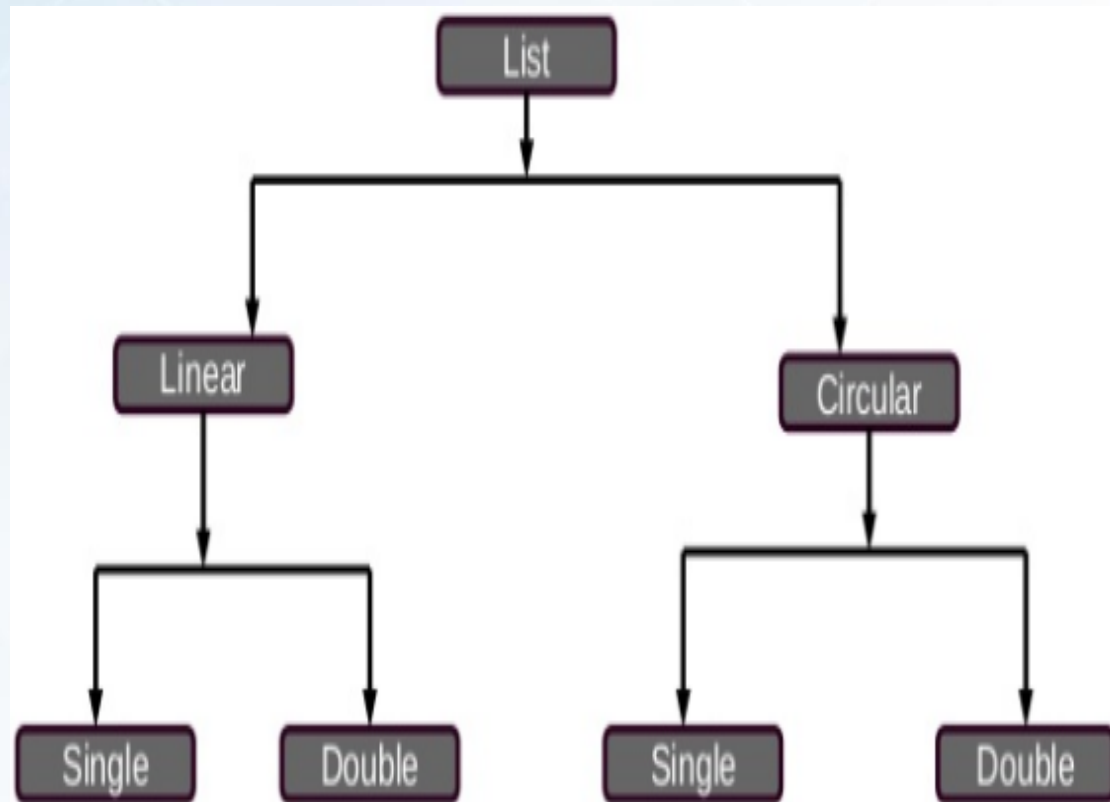
A Linked list is a linear collection of data elements .It has two part one is info and other is link part.info part gives information and link part is address of next node.



Types of List

Types of linked lists:

- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list

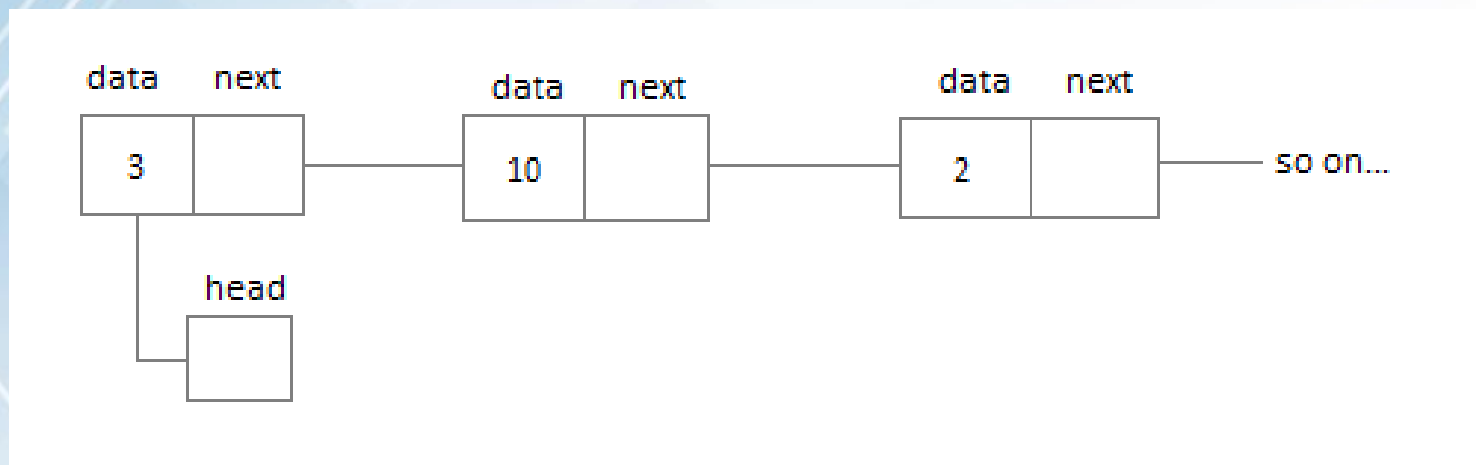


Singly Linked List

Singly 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.



Doubly linked list

Doubly linked list

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

In doubly linked list each node contains three parts:

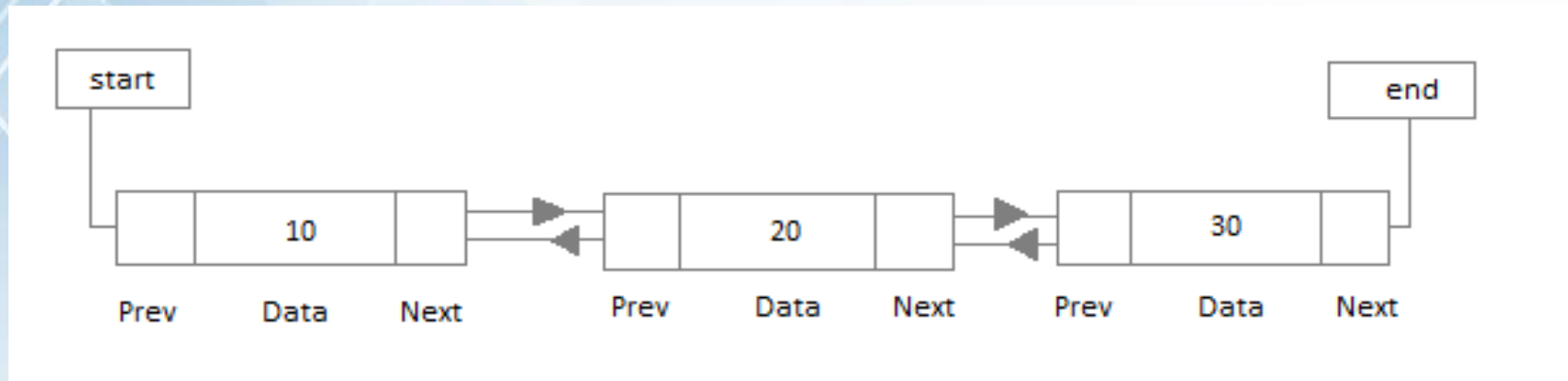
FORW : It is a pointer field that contains the address of the next node

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.

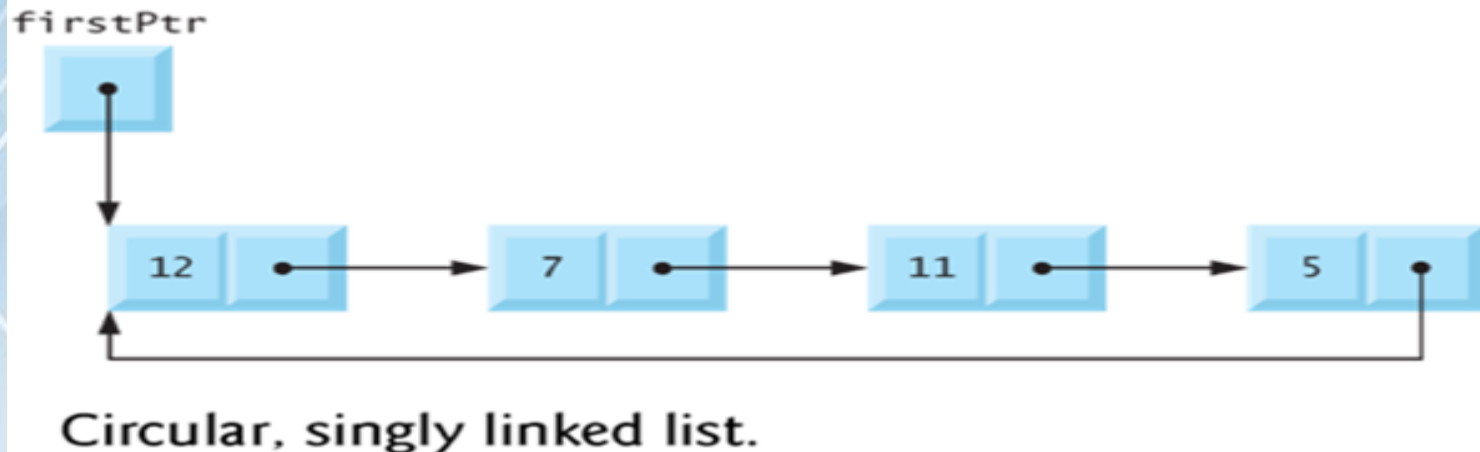
The node in which **FORW** contains, **NULL** indicates that the node is the last node.



single circular LL

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 doubly linked list

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.

- Circular doubly linked list doesn't contain NULL in any of the nodes.
- The last node of the list contains the address of the first node of the list.
- The first node of the list also contains the address of the last node in its previous pointer.

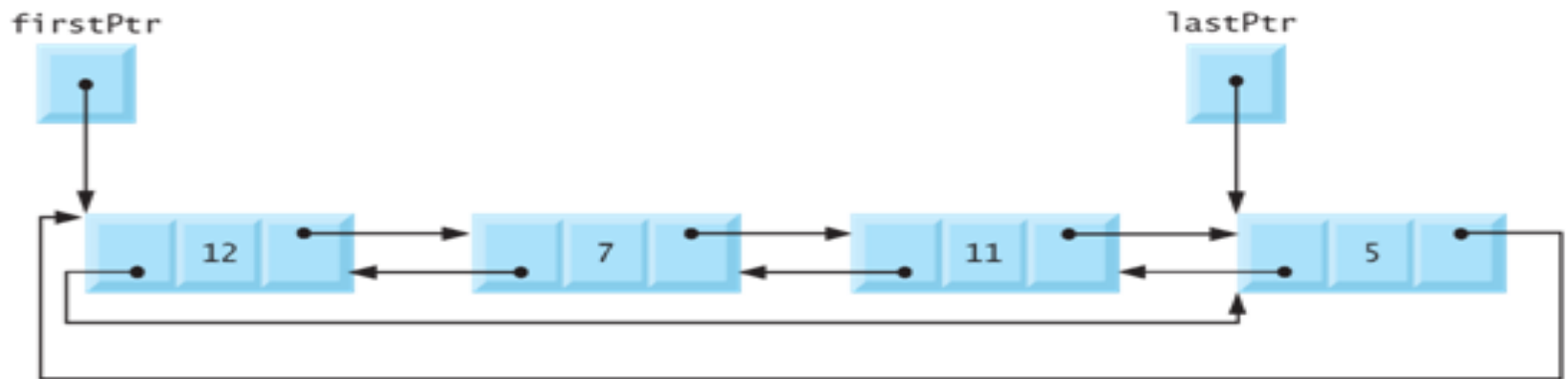


Fig. 19.12 | Circular, doubly linked list.

Creating a linked list

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

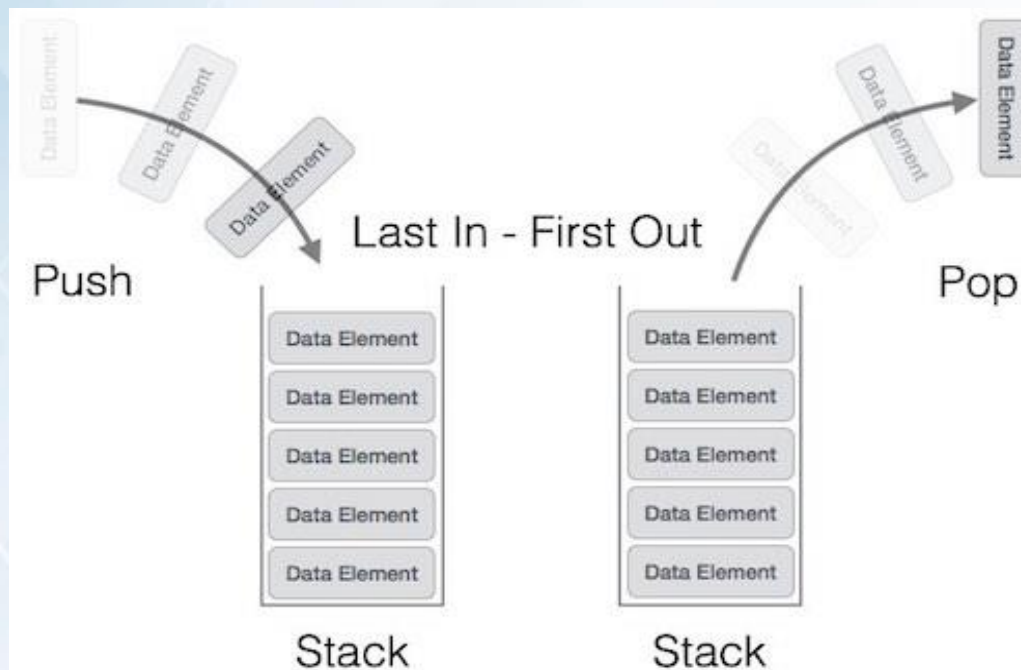
```
struct Node
{
    int info;
    struct Node *link;
} *node1, node2;
```

Here info is the information field and link is the link field.

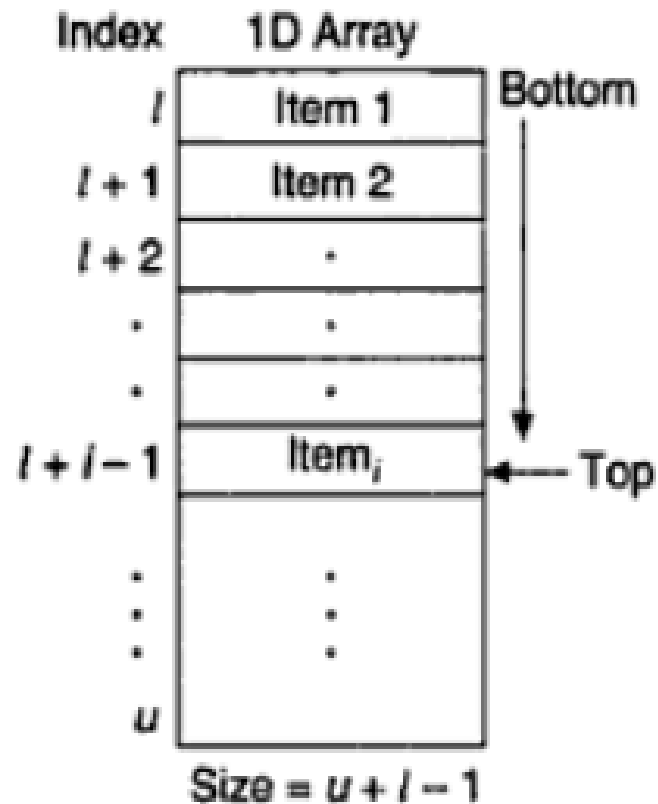
The link field contains a pointer variable that refers the same node structure. Such a reference is called as *Self addressing pointer*.

Stack

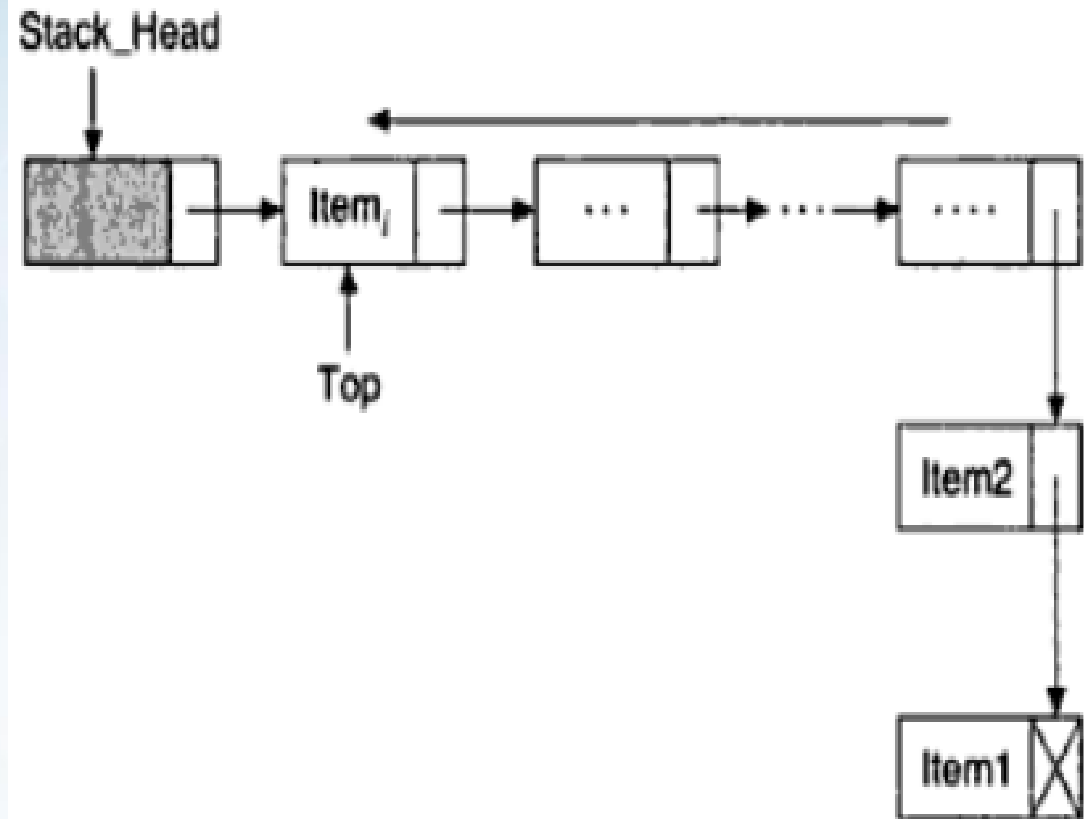
- Stack is a linear data structure
- In which the insertion and deletion operations are performed at only one end.
- In a stack, adding and removing of elements are performed at a single position which is known as "**top**".
- A Stack is a collection of objects inserted and removed according to the Last In First Out (LIFO) principle. Think of a stack of books



Representation of Stack



(a) Array representation of a stack



(b) Linked list representation of a stack

Push()

Algorithm Push_Array

Input: The new item ITEM to be pushed onto it.

Output: A stack with a newly pushed ITEM at the TOP position.

Data structure: An array A with TOP as the pointer.

Steps:

1. **If** TOP \geq SIZE **then**
2. **Print** "Stack is full"
3. **Else**
4. TOP = TOP + 1
5. A[TOP] = ITEM
6. **EndIf**
7. **Stop**

Pop ()

Algorithm Pop_Array

Input: A stack with elements.

Output: Removes an ITEM from the top of the stack if it is not empty.

Data structure: An array *A* with TOP as the pointer.

Steps:

1. **If** TOP < 1 **then**
2. **Print** "Stack is empty"
3. **Else**
4. ITEM = A[TOP]
5. TOP = TOP - 1
6. **EndIf**
7. **Stop**

Status of Stack

Algorithm Status_Array

Input: A stack with elements.

Output: States whether it is empty or full, available free space and item at TOP.

Data structure: An array *A* with TOP as the pointer.

Steps:

1. **If** TOP < 1 **then**
2. **Print** "Stack is empty"
3. **Else**
4. **If** (TOP ≥ SIZE) **then**
5. **Print** "Stack is full"
6. **Else**
7. **Print** "The element at TOP is", A[TOP]
8. free = (SIZE – TOP)/SIZE * 100
9. **Print** "Percentage of free stack is", free
10. **EndIf**
11. **EndIf**
12. **Stop**

peek()

Algorithm of peek() function:

- begin procedure peek
- return stack[top]
- end procedure

isfull()

Algorithm of isfull() function

- begin procedure isfull
- if top equals to MAXSIZE
- return true
- else
- return false
- endif
- end procedure
- }

Algorithm Push_LL

Input: ITEM is the item to be inserted.

Output: A single linked list with a newly inserted node with data content ITEM.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

1. new = **GetNode(NODE)**
/ Insert at front */*
2. new→DATA = ITEM
3. new→LINK = TOP
4. TOP = new
5. STACK_HEAD→LINK = TOP
6. **Stop**

Algorithm Pop_LL

Input: A stack with elements.

Output: The removed item is stored in ITEM.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

1. **If** TOP = NULL
2. **Print** "Stack is empty"
3. **Exit**
4. **Else**
5. ptr = TOP→LINK
6. ITEM = TOP→DATA
7. STACK_HEAD→LINK = ptr
8. TOP = ptr
9. **EndIf**
10. **Stop**

Algorithm Status_LL()

Input: A stack with elements.

Output: Status information such as its state (empty or full), number of items, item at the TOP.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

1. ptr = STACK_HEAD→LINK
2. **If** (ptr = NULL) **then**
3. **Print** "Stack is empty"
4. **Else**
5. nodeCount = 0
6. **While** (ptr ≠ NULL) **do**
7. nodeCount = nodeCount + 1
8. ptr = ptr→LINK
9. **EndWhile**
10. **Print** "The item at the front is", TOP→DATA, "Stack contains", nodeCount, "Number of items"
11. **EndIf**
12. **Stop**

Advantages of Stack

- A Stack helps to manage the data in the 'Last in First out' method.
- It allows you to control and handle memory allocation and deallocation.
- It helps to automatically clean up the objects.

Disadvantages of Stack

- It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
- It has very limited memory.
- In Stack, random access is not possible.

Application of the Stack

- A Stack can be used for evaluating expressions consisting of operands and operators.
- Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
- It can also be used to convert one form of expression to another form.
- It can be used for systematic Memory Management.

Expression Evaluation

- Expression consists of operands and operators such as arithmetic, logical, relational, unary and parenthesis.
- simple arithmetic expression

$$A + B * C / D - E \wedge F * G$$

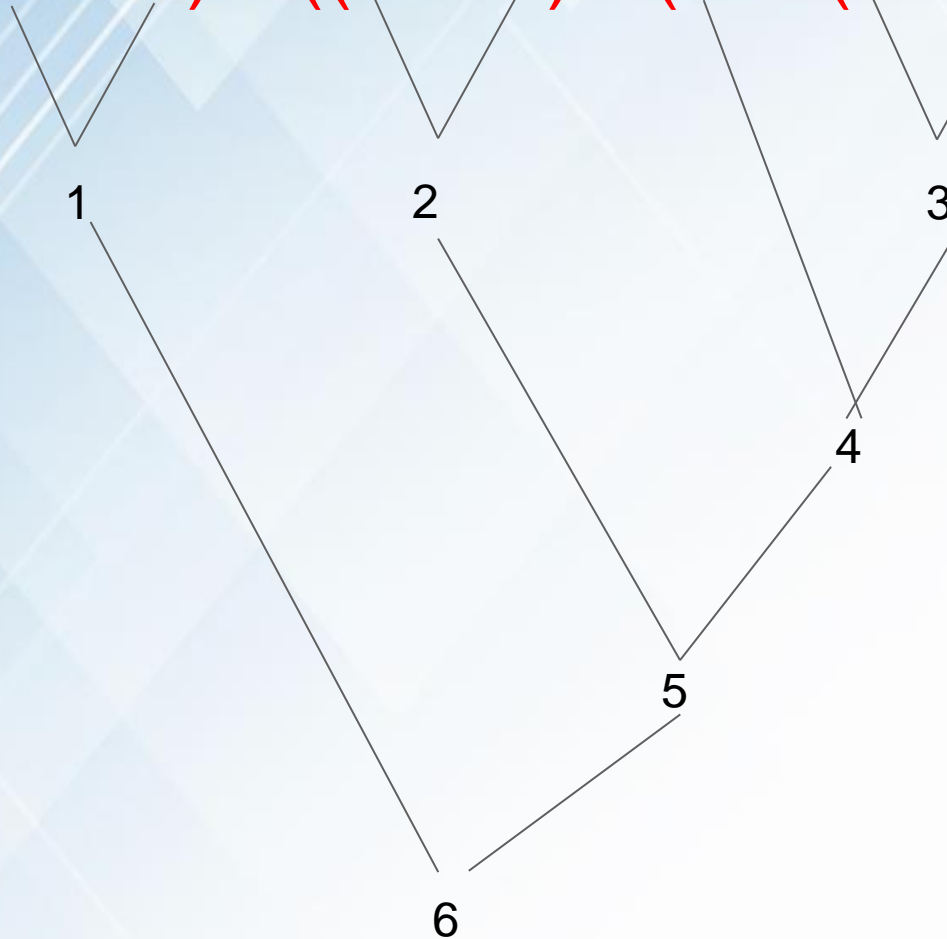
- The problem to evaluate this expression is the order of evaluation.

Precedence and Associativity of operators

<i>Operators</i>	<i>Precedence</i>	<i>Associativity</i>
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right

Paranthesised Infix expression

$((A + B) * ((C / D) - (E \wedge (F * G))))$



Problems in previous methods

- Repeatedly scanning the expression from left to right
- Ambiguous in generating corresponding code for given expression
- problem arises if partially paranthesised

These problems can be fixed in two steps

1. conversion of given expression to special notation
2. evaluate the object code using stack

Notations for arithmetic expression

- Conventional way of writing an expression is called **Infix notation**

<operand> <operator> <operand>

eg. $A+B$, $4*7$, $c-9$

- **Prefix notation** always uses the operator before the operands

<operator> <operand> <operand>

eg. $+AB$, $*47$, $-c9$

- **Postfix notation** always writes the operator after the operands

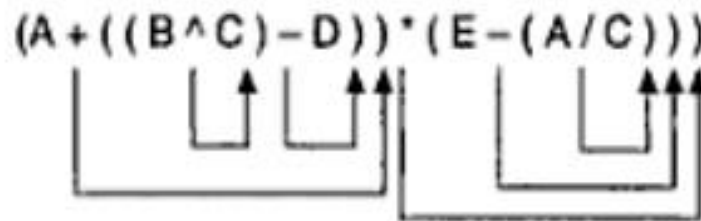
<operand> <operand> <operator>

eg. $AB+$, $47*$, $c9-$

Converting Infix to postfix and prefix

- Assume fully parenthesized version of infix expression.
- Move all operators so that they replace their corresponding right/left part of parentheses
- Remove all parentheses

Input: $((A + ((B \wedge C) - D)) * (E - (A/C)))$
(A fully parenthesized expression)



(Arrows point from operators to their corresponding right parenthesis.)

(Arrows point from operators to their corresponding right parenthesis.)

$((A ((B C ^ D - + (E (AC / - *$

(Operators are moved to their respective right parentheses.)

Output: $A B C ^ D - + E A C / - *$

(All parentheses are removed yielding the postfix expression.)

- infix : $(A + ((B ^ C) - D)) * E - (A / C))$
- postfix : $A B C ^ D - + E A C / - *$
- prefix : $* + A - ^ B C D - E / A C$

Evaluation of Infix Expression

- two steps involved in evaluation of expression
- 1. Conversion of Infix to Postfix
- 2. Evaluating Postfix Notation

Rules for converting Infix to Postfix

- Print the operands as they arrive Infix $A+B/C$
- If stack is empty or contains '(' then push the operator onto the stack
- If incoming symbol is '(' push it onto stack
- If incoming symbol is ')' pop the stack until & print operators until '(' is found
- If incoming symbol is **higher precedence** than top of the stack **push** in onto the stack
- If incoming symbol is **lower precedence** than top of the stack then **pop and print the top** (test the symbol with new top)
- If incoming symbol is having **equal precedence** with top use **associativity rule**

Associativity L->R then pop & print top of the stack & push the incoming operator

Associativity R->L then push the incoming operator

- 7/29/2024
- If end of the expression pop and print all operators.

Example Conversion of Infix to Postfix

$$A + B / C$$

Read symbol	Stack	postfix expression
A		A
+	+	A
B	+	AB
/	+/	AB
C	+/	ABC
end		ABC/+

Example 2

$$A-B/C*D+E$$

Read symbol	Stack	Postfix expression
A		A
-	-	A
B	-	AB
	- /	AB
C	- /	ABC
*	- *	ABC/
D	- *	ABC/D
+	+	ABC/D*-
E	+	ABC/D*-E
END		ABC/D*-E+ Postfix expression

Algorithm to evaluate Postfix expression

```
1. Append a special delimiter '#' at the end of the expression
2. item = E.ReadSymbol() // Read the first symbol from E
3. While (item ≠ '#') do
4.     If (item = operand) then
5.         PUSH(item) // Operand is the first push into the stack
6.     Else
7.         op = item // The item is an operator
8.         y = POP() // The right-most operand of the current operator
9.         x = POP() // The left-most operand of the current operator
10.        t = x op y // Perform the operation with operator 'op' and operands x, y
11.        PUSH(t) // Push the result into stack
12.    EndIf
13.    item = E.ReadSymbol() // Read the next item from E
14. EndWhile
15. value = POP() // Get the value of the expression
16. Return(value)
17. Stop
```

Example to evaluate the postfix expression

Infix: $A + (B * C) / D$

Postfix: $A B C * D / +$

Input: $A B C * D / + \#$ with $A = 2$, $B = 3$, $C = 4$, and $D = 6$

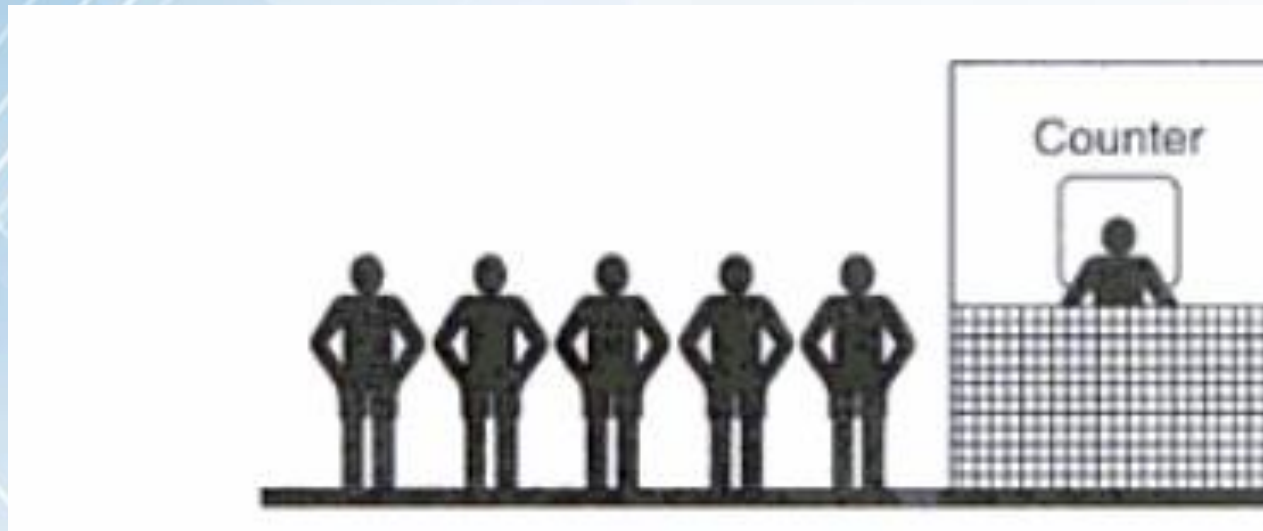
<i>Read symbol</i>	<i>Stack</i>	
A	2	PUSH(A = 2)
B	2 3	PUSH(B = 3)
C	2 3 4	PUSH(C = 4)
*	2 12	POP(4), POP(3), PUSH(T = 12)
D	2 12 6	PUSH(D = 6)
/	2 2	POP(6), POP(12), PUSH(T = 2)
+	4	POP(2), POP(2), PUSH(T = 4)
#		value = POP()

Infix to Prefix

- First reverse the given expression
- If the scanned character is an operand, put it into prefix expression.
- If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.
- If the operator's stack is not empty, there may be following possibilities.
 - If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.
 - If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack until we find a low precedence operator than the scanned character.
 - If the precedence of scanned operator is equal then check the associativity of the operator. If associativity **left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.**
- If the scanned character is opening round bracket ('('), push it into operator's stack.
- If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').
- Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Queue

- A queue is the simple data structures and it is very powerful in solving numerous computer application



- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, adding and removing elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.
- The insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.

- **Example:**

- Any waiting line is a queue:
 - • The check-out line at a grocery store
 - • The cars at a stop light
 - • An assembly line

Functions of Queue

The insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

