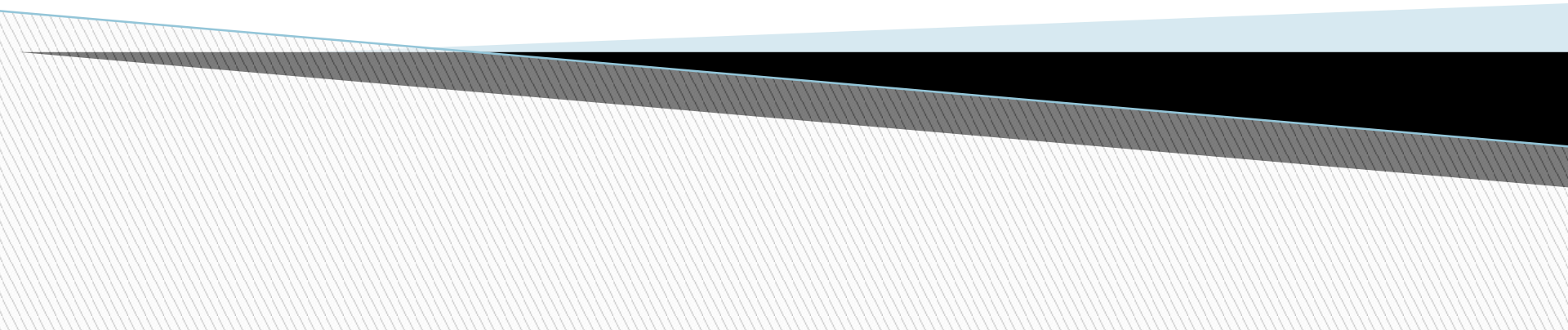
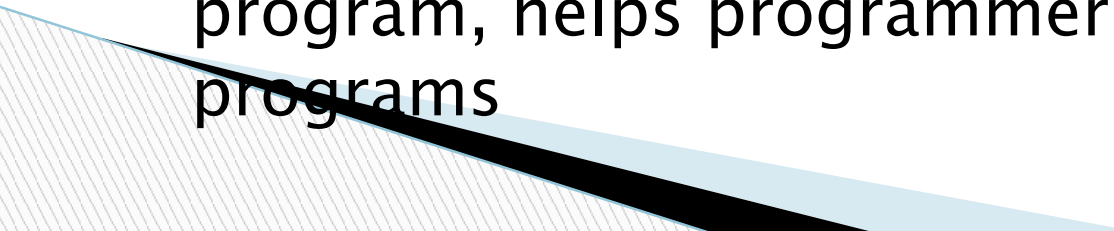


DATA STRUCTURES WITH ALGORITHMS

MODULE 1
(Principles of DS using C and C++)



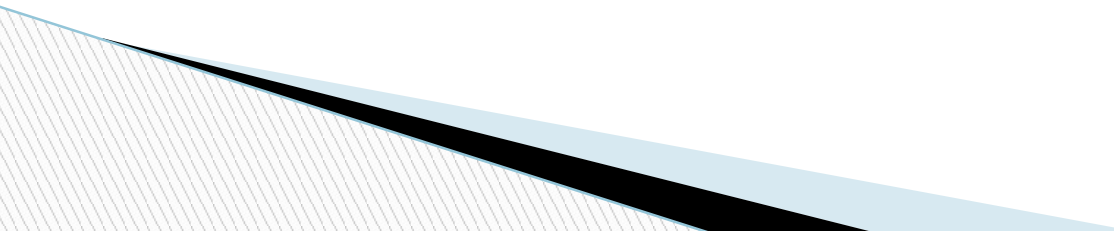
Data Structures

- Data structure is the structural representation of logical relationships between elements of data.
 - Data structure is a way of organizing data items by considering its relationship to each other.
 - Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity.
 - Data structure affects the design of both the structural and functional aspects of a program.
 - Algorithm + Data Structure = Program
 - Data structures are the building blocks of a program, helps programmer to design efficient programs
- 

Classification of Data Structures

- ❑ Primitive data str : basic DS, can be directly operated upon by m/c instr which are at primitive level.
- ❑ EG:int, float, char,pointer etc
- ❑ Non – primitive DS : a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- ❑ EG: Array, list, files, linked list, trees and graphs
- ❑ Diagram (p-9,10)

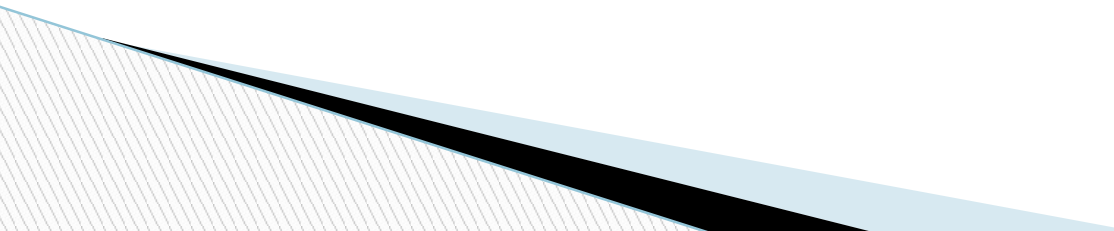
- Array : collection of homogeneous data elements described by a single name, accessed using index, 1D also called as linear array
- Multi-dimension array : sparse array imp application of arrays where nearly all elements have same value (usu. Zero). 1D sparse array is sparse vector , 2D sparse array is sparse matrix
- Similar to array is vector which too has fixed size, 1D, can be column vector. Can contain real num or complex num($a+bi$)

- ❑ Disadvantage of array : can't insert or delete an element and re-adjust the memory, fixed len, list overcomes all these
 - ❑ List : ordered set containing varying num of elements, where insertion and deletion can be performed. List that is displaying relation to adjacent ele, is linear list which is implemented using pointers, which is collection of node that contains data and pointer to next node
- 

□ Files and records :

- A file : large list that is stored in the external memory of a computer.
- A record is a collection of information about a particular entity Or a collection of related data items, each of which is called a field or attribute
- A file is a collection of similar records.
- A record differs from a linear array in following ways:
 - (a) A record may be a collection of non-homogeneous data; may have different data types.
 - (b) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

- Strings : finite seq of zero or more char, if string len is zero, called as null or empty string
- Can be stored or represented in memory by using following three types of structures
 - Fixed length structures : adv is can be viewed as record, so data can be accessed or updated easily, disadv : entire record is read even if it holds blank spaces, data may be longer than fixed size
 - Variable len str with fixed maximum : data terminated using special marker
 - Linear structures: each char is sequentially arranged in mem cells called nodes, also contains sub-string, to access we need name of str, position and len of sub-string

- Linear and non-linear DS representation :
 - [Difference between Linear and Non-linear Data Structures - GeeksforGeeks.html](#)
 - [Linear vs Non-Linear data structure - javatpoint.html](#)
 - Linear and Binary Search : [DS_search.pptx](#)
 - Program : [Search\(Lab_1\).C](#)
 - Bubble and selection sort : [DS_sort.pptx](#)
 - Program : [Sort\(Lab_2\).c](#)
- 

Binary Search

- A binary search is a simple searching technique which can be applied if the items to be compared are either in ascending order or descending order.

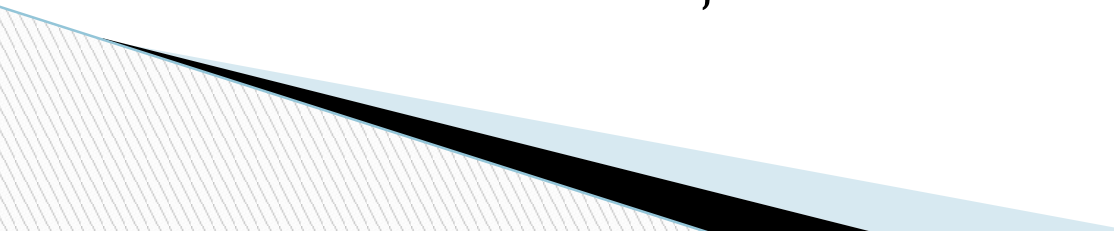
Rule:

$\text{Mid} = (\text{low} + \text{high})/2 ;$

If ($\text{key} == \text{a}[\text{mid}]$)
 return mid;

If($\text{key} < \text{a}[\text{mid}]$)
 High = mid - 1;

If($\text{key} > \text{a}[\text{mid}]$)
 low = mid + 1;

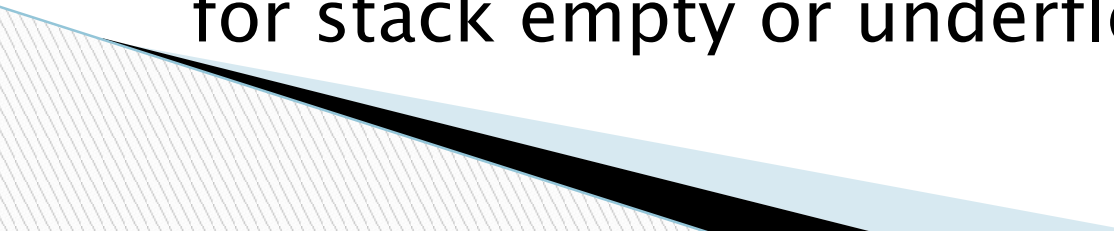


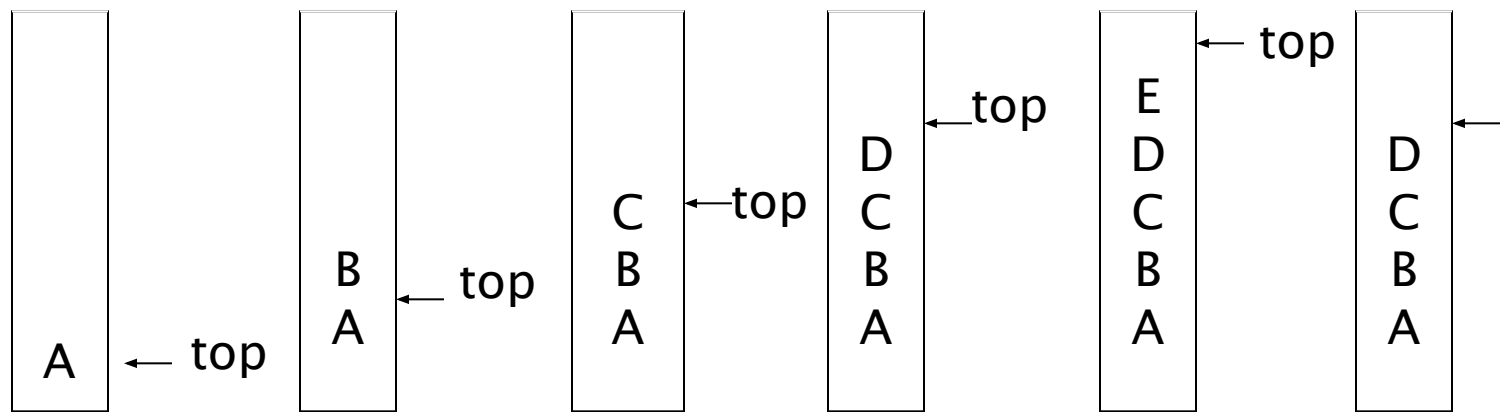
```
int search(int key , int a[], int low, int high)
{
    int mid;
    if(low>high) return -1;
    mid = (low+high)/2;
    if(key == a[mid]) return mid;
    if(key < a[mid]) return search(key , a , low , mid-1);
    else
        return search(key , a , mid+1, high);
}

#include<stdio.h>
void main()
{
    int n , i , a[20], item , pos;
    printf("enter the number of elements\n");
    scanf("%d",&n);
    printf("enter %d items\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    pos = search(item, a, 0,n-1);
```

```
if(pos==-1)
    printf("item not found\n");
Else
    printf("item found at %d position\n", pos);
```

Stack

- ❑ One of most imp non-primitive linear DS
 - ❑ An ordered collection where new ele can be added or deleted only at one end which is called the Top
 - ❑ Hence called LIFO(Last in First out)
 - ❑ Most recent ele on top and least accessed ele at bottom
 - ❑ Insert operation : push, here need to check for stack full or overflow cond
 - ❑ Delete operation : pop, here need to check for stack empty or underflow cond
- 



Primitive operations

- When item added to a stack – pushed onto stack
- When an item is removed – popped from stack
- Given a stack s and item I , push operation performed by $\text{push}(s, i)$ to the top of stack
- Operation $\text{pop}(s)$ removes ele from top and returns top ele as function value
 $I = \text{pop}(s)$
- Operations performed in previous slide –
 $\text{push}(s, 'A')$, $\text{push}(s, 'B')$, $\text{push}(s, 'C')$, $\text{push}(s, 'D')$,
 $\text{push}(s, 'E')$, $\text{pop}(s)$
- B'coz of push operations, stack sometimes called as pushdown list.

- Stack has no upper limit on number of items added
- If stack has 1 item and is popped, then resulting stack is known as empty stack
- Although `push(s, i)` is applicable on any stack, `pop(s)` can't be performed on an empty stack, as stack has no elements to remove.
- Hence, before applying `pop(s)`, check to ensure, stack is not empty, by implementing `empty(s)`, which returns true or false
- Another operation that can be performed on stack is to find which element is at the top of the stack without deleting it – `stacktop(s)` (basically combination of `pop()` and `push()`)
- Result of attempting to remove or read element from empty stack – underflow
- Underflow can be avoided by ensuring `empty(s)` is false and then calling `pop(s)` or `stacktop(s)`

- An **abstract data type** in stack includes the following operations
 - **MAKENULL(S)**
 - Stack ***S*** is made ***empty***.
 - **TOP(S)**
 - The element at the ***top*** of the stack ***S*** is ***returned***.
 - **POP(S)**
 - The ***top*** element of the stack ***S*** is ***deleted***.
 - **PUSH(x, S)**
 - The element ***x*** is ***inserted*** into the stack ***S***.
 - **EMPTY(S)**
 - It returns ***true*** if stack ***S*** is ***empty*** otherwise it returns ***false***.

□ Primitive Operations on Stack

□ **push**

- An item is added to a stack i.e. it is **pushed** onto the stack.

□ **pop**

- An item is deleted from the stack i.e. it is **popped** from the stack.

- The operations take place on the stack – **Stack**, which contains **max_size** number of elements.

- The element **item** is inserted in the stack at the position **top**.

- Initially **top** is set to **-1**.

- Since the push operation adds elements into a stack, the stack is also called as **pushdown list**.

- There is no upper limit on the number of items to be inserted into a stack.

- If there are no elements in a stack it is considered as **empty stack**.

- The **push** operation is applicable to any stack whereas the **pop** operation is not applicable on an empty stack.

- We need to have a new operation ***empty(s)*** that checks whether a stack ***s*** is empty or not.
 - If the stack is empty ***empty(s)*** will return a value ***TRUE*** else it will return a value ***FALSE***.
 - When there is an attempt to pop or access an item from an empty stack, the result is an ***underflow*** condition.
- Another operation on a stack is to determine what is the top item of the stack without removing it.
 - ***stacktop(s)*** returns the top element of the stack ***s***.

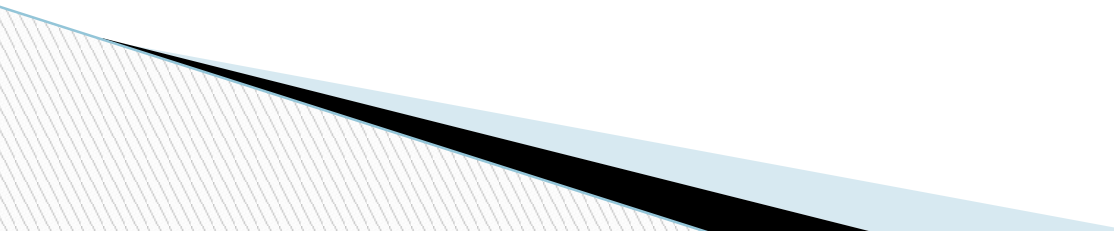
Stack as an ADT

- **etype** denotes the type of the stack element.
- **abstract typedef** <<etype>> STACK (etype);
- **abstract** empty(s)
- STACK(etype) s;
- **postcondition** empty==(len(s)==0)
-
- **abstract** etype pop(s)
- STACK(etype) s;
- **precondition** empty(s)==FALSE;
- **postcondition** pop == first(s');
 s == sub(s', 1, len(s') -1);
- **abstract** etype push(s, elt)
- STACK(etype) s;
- etype elt;
- **postcondition** push == (s == <elt> + s');

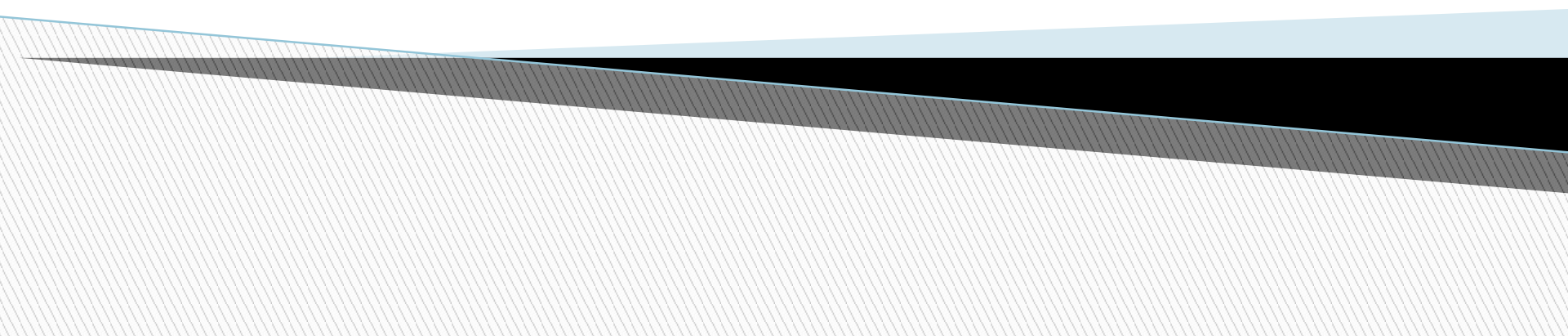
Representing Stacks in 'C'

- Since stack is an ordered collection of items, it can be represented easily using the concept of **arrays** in C.
- One dimensional array can be used to represent.
- Example: **Stack[max_stack_size]**
where **max_stack_size** is the max number of entries.
- The first element of the stack is stored in **stack[0]**, the second in **stack[1]** and the i^{th} element in **stack[i-1]**.
- The variable **top** points to the top element of the stack.
- Initially **top=-1** to indicate an **empty** stack.
- The operations on a stack are
 - **Push – inserting** an element into the stack.
 - **Pop– deleting** an element from the stack.
- the position of the current stack top within the array

- Each time an element is entered in to the stack **top** is **incremented**.
- The elements can be entered into the stack until **top** $\geq \text{max_stack_size} - 1$.
- When a new insertion is attempted after this condition it becomes the **STACK OVERFLOW** condition.
- Each time an element is deleted from the stack **top** is **decremented**.
- The elements can be deleted from the stack until **top** $= - 1$.
- When a new deletion is attempted after this condition it becomes the **STACK UNDERFLOW** condition.
- It can be declared as a structure using two objects:
 - Array to hold the elements of the stack
 - Integer to indicate

- ▣ Operations : push and pop with cond verified
 - ▣ Can be implemented using static(arrays) and dynamic(pointers) implementation
 - Static impl problem : if array size is too large for small data or size of array not sufficient for larger data as memory can't be re-allocated
 - Alg for Stack operations for array(push)
 - [Stack_Impl.pptx](#)
 - [Stack.c](#)
 - [str_Stack.c](#)
 - [list_stack.c](#)
- 

Applications of Stack



Polish and reverse polish expressions

- **Prefix Expression** : also called as ***Polish Notation***.
- The operator symbols are placed before the operands.
- EG : +AB, -AB, -+ABC

- **Postfix Expression** : also called as ***reverse polish notation***.
- The operator symbols are placed after the operands.
- EG : AB+, AB-, AB+C-

Infix to postfix conversion

- Operator precedence

Exponential operator \wedge	Highest
Multiplication/Division $*, /$	Next
Addition/Subtraction $+, -$	Last

- The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated operator having higher precedence are first parenthesized.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

□ **A + B**

A B +

□

□ **A + B - C**

(A + B) - C

□

AB+ - C

□

AB+C-

□

□ **(A+B)*(C-D)**

(AB+)*(C-D)

□

AB+*(CD-)

□

AB+*CD-

□

AB+CD-*

□ **A - B / (C * D \$ E)**

A - B/(C*DE\$)

□

A - B/CDE\$*

□

A - BCDE\$*/

□

A BCDE\$*/ -

□ **A + [(B + C) + (D + E) * F] / G (question)**

□ A + { [(BC +) + (DE +) * F] / G }

□ A + { [(BC +) + (DE + F *)] / G }

□ A + { [(BC + (DE + F * +)] / G } .

□ A + [BC + DE + F * + G /]

□ ABC + DE + F * + G / + (Postfix Form) [infix_postfix\(Lab_4\).c](#)

Algorithm

1. Push "(" onto stack, and add ")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q(output).
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator \otimes is encountered, then:
 - (a) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than \otimes .
 - (b) Add \otimes to stack.
6. If a right parenthesis is encountered, then:
 - (a) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
 - (b) Remove the left parenthesis. [Dont add the left parenthesis to P]
7. Exit.

Note. Special character \otimes is used to symbolize any operator in P.

❑ **Algorithm**

- ❑ **Step 1** : Scan the Infix Expression from left to right.
- ❑ **Step 2** : If the scanned character is an operand, append it with final Infix to Postfix string.
- ❑ **Step 3** : Else,
 - **Step 3.1** : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '(', '[', or '{'), push it on stack.
- ❑ **Step 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.)
- ❑ **Step 4** : If the scanned character is an '(', '[', or '{', push it to the stack.
- ❑ **Step 5** : If the scanned character is an ')', ']', or '}', pop the stack and output it until a '(', '[', or '{' respectively is encountered, and discard both the parenthesis.
- ❑ **Step 6** : Repeat steps 2-6 until infix expression is scanned.
- ❑ **Step 7** : Print the output
- ❑ **Step 8** : Pop and output from the stack until it is not empty.

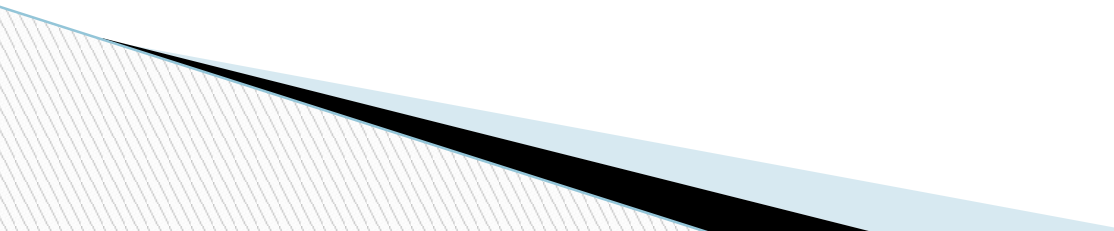
Evaluate Postfix expression

1. Add a right parenthesis “)” at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel “)” is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator \otimes is encountered, then:
 - (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
 - (b) Evaluate $B \otimes A$.
 - (c) Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

- 1) Create a stack to store operands
(or values).
- 2) Scan the given expression and do following
for every scanned element.
 -a) If the element is a number, push it into
the stack
 -b) If the element is a operator, pop
operands for the operator from stack.
Evaluate the operator and push the result back
to the stack
- 3) When the expression is ended, the number
in the stack is the final answer



Converting infix to prefix

- Reverse the given infix expression
 - Apply the same method of conversion to a postfix expression
 - Once the expression is converted reverse it back to get a prefix expression.
 - If expression is unparanthetic , make it fully paranthetic using operator precedence rule.
- 

- ❑ Step 1. Push “)” onto STACK, and add “(“ to end of A
- ❑ Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
- ❑ Step 3. If an operand is encountered add it to B
- ❑ Step 4. If a right parenthesis is encountered push it onto STACK
- ❑ Step 5. If an operator is encountered then:
 - ❑ a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
 - ❑ b. Add operator to STACK
- ❑ Step 6. If left parenthesis is encountered then
 - ❑ a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)
 - ❑ b. Remove the left parenthesis
- ❑ Step 7. Exit

□ Eg:

1. $A + B * C / (D + E)$

2. $A + B * C$

Postfix to infix expression

1. While there are input symbol left
 - ...1.1 Read the next symbol from the input.
2. If the symbol is an operand
 - ...2.1 Push it onto the stack.
3. Else
 - ...3.1 the symbol is an operator.
 - ...3.2 Pop the top 2 values from the stack.
 - ...3.3 Put the operator, with the values as arguments and form a string.
 - ...3.4 Push the resulted string back to stack.
4. If there is only one value in the stack
 - ...4.1 That value in the stack is the desired infix string.

- [Postfix to Infix Conversion.html](#)
- [Postfix to Infix in C++.html](#)

Converting prefix to postfix

- Reverse the given prefix expression(or read in reverse order)
- If scanned character is an operand , push it on to the stack.
- If scanned character is an operator , pop two operands into op1 and op2 respectively and concatenate op1 and op2 with the operator. Result $\text{str} = \text{op1} + \text{op2} + \text{operator}$, push it back to stack
- Eg: $-a/b * c^d e$