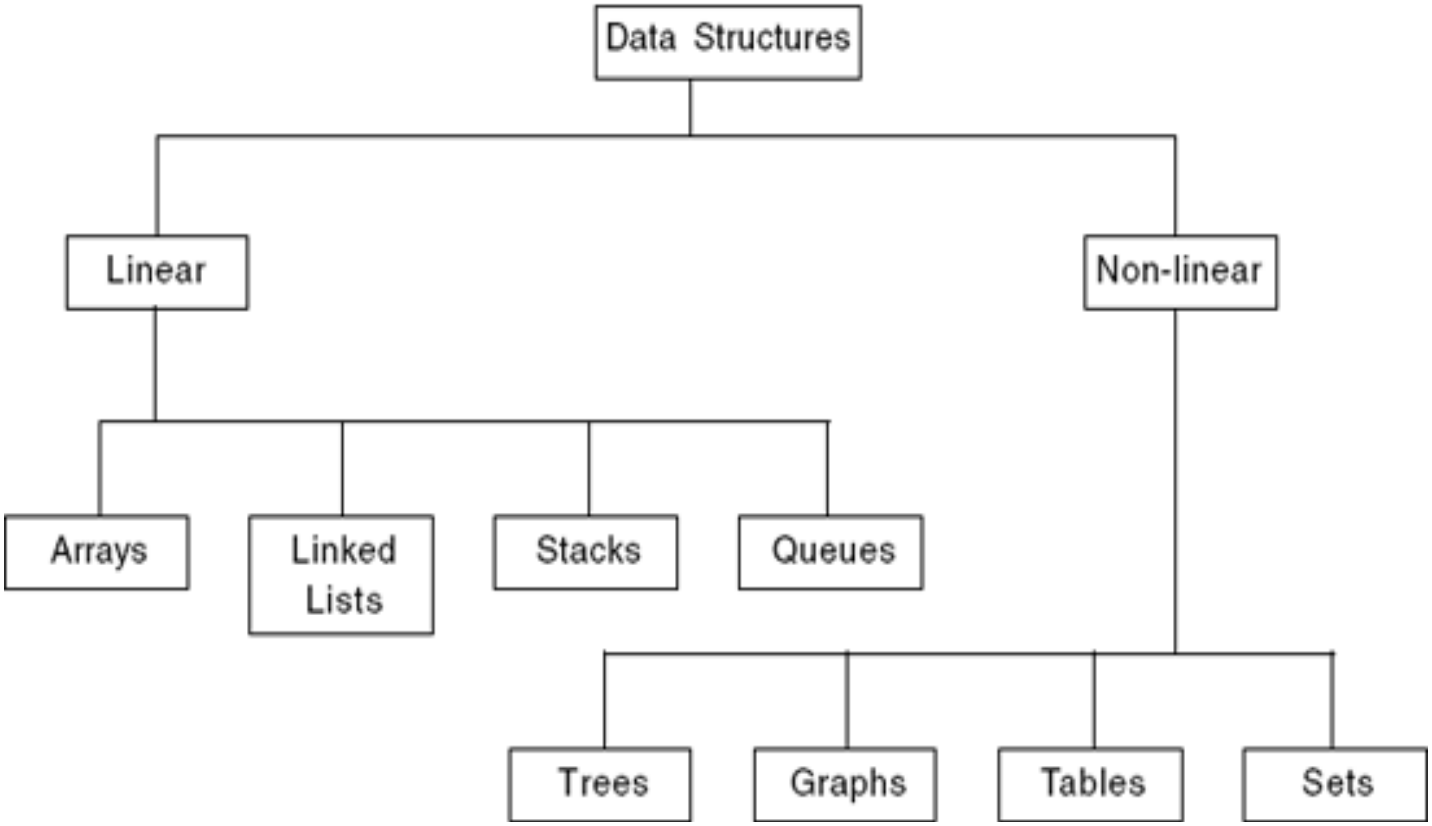
Stack

Unit-2

Presented By: Dr. Aparna Kumari

Classification

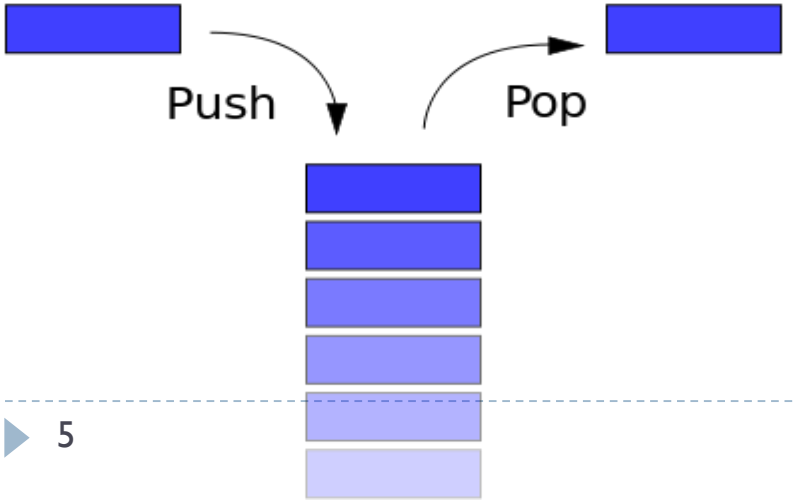


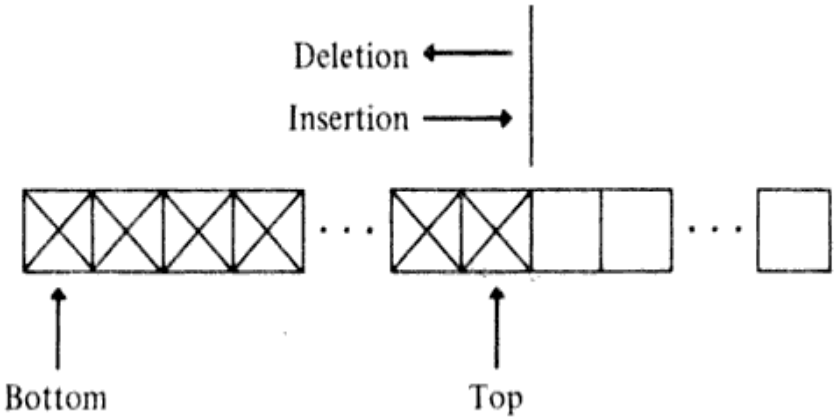
- Stacks
  - Allow insertions and removals only at top of stack
- Queues
  - Allow insertions at the back and removals from the front
- Linked lists
  - Allow insertions and removals anywhere
- Binary trees
  - High-speed searching and sorting of data and efficient elimination of duplicate data items

Stacks

Introduction

- Stack is an important data structure which stores its elements in an ordered manner.
- A stack is a linear data structure which uses the principle, i.e., the elements in a stack are added and removed only from one end, which is called the *top*.
- Hence, a stack is called a LIFO (Last-In, First-Out) data structure as the element that is inserted last is the first one to be taken out.





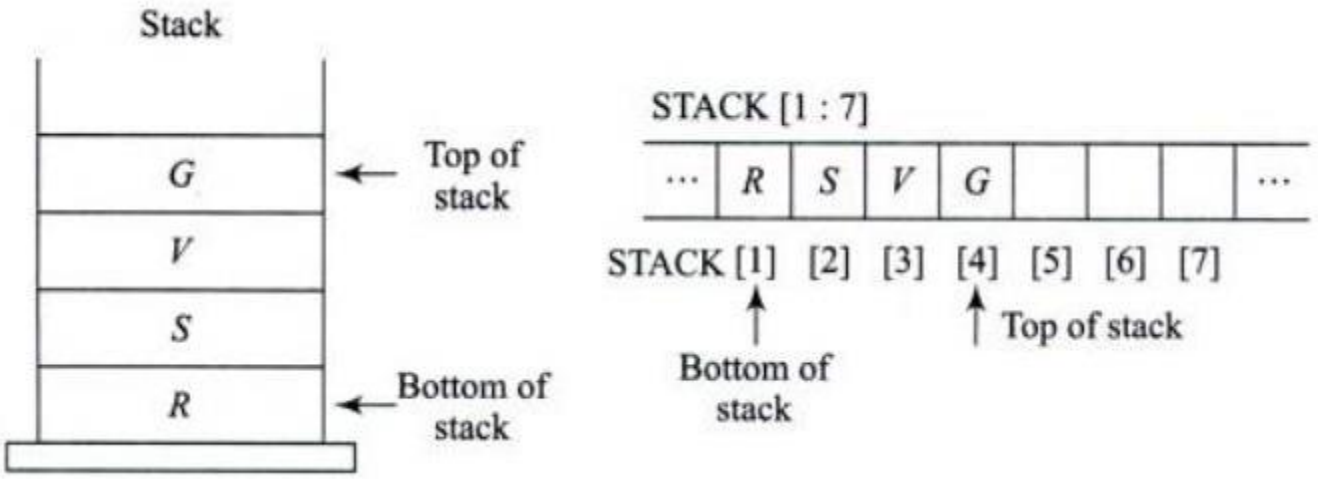
Introduction

- Real life examples of stack:
  - Suppose we have created stack of the book
  - How books are arranged in the stack?
    - Books are kept one above the other
    - Books which are inserted first is taken out last. (brown)
    - Book which is inserted lastly is served first. (light green)
  - Suppose at your home you have multiple chairs then you put them together to form a vertical pile. From that vertical pile, the chair which is placed last is always removed first.
  - Chair which was placed first will be removed last.

Array representation of stacks

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it.
- TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX -1, then the stack is full.

Array representation of stacks



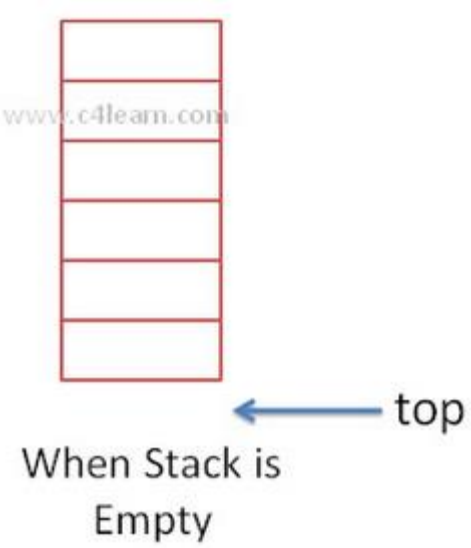
- *typedef struct {*
  int size;
  int top;
  int items[STACKSIZE];
  } STACK;

Operations of Stack

- Push : inserting element onto stack.
- Pop : removing element from stack.
- Peep : returns the top element of the stack.
- Change : changes the i^{th} element from top of stack to the mentioned element.

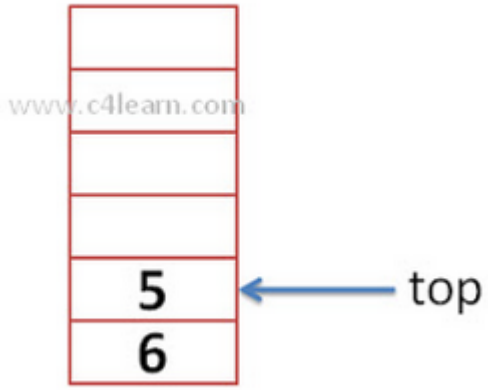
Visual Representation of Stack

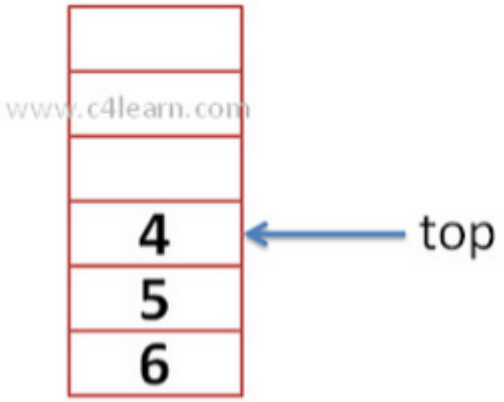
- View 1: When stack is empty
  - When stack is empty then it does not contain any element inside it. Whenever stack is empty, the position of topmost element is -1.



Visual Representation of Stack

- View 2: When stack is not empty
  - Whenever we add very first element then topmost position will be increment by 1. After adding first element, top = 0.





- View 3: After deletion of 1 element top will be decremented by 1.

Visual Representation of Stack

- Position of top and its value:

Position of Top	Status of Stack
-1	Stack is Empty
0	First Element is Just Added into Stack
N-1	Stack is said to Full
N	Stack is said to be Overflow

- Values of stack and top:

Operation	Explanation
top = -1	-1 indicated Empty Stack
top = top + 1	After push operation value of top is incremented by integer 1
top = top – 1	After pop operation value of top is decremented by 1

Push Operation

- Procedure PUSH(S, TOP, X): This procedure inserts an element X on the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

```
Algorithm to PUSH an element in a stack

Step 1: IF TOP = N-1, then
             PRINT "OVERFLOW"
             Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET S[TOP] = X
Step 4: END
```

Pop Operation

- Procedure POP(S, TOP): This procedure removes the element from the stack which is represented by a vector S and returns the element.

```
Algorithm to POP an element from a stack

Step 1: IF TOP = -1, then
           PRINT "UNDERFLOW"
           Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP - 1
Step 3: Return S(TOP+1)
Step 4: END
```

Peek Operation

- Procedure PEEK(S, TOP, I): Given the vector S (consisting of N elements) representing a sequentially allocated stack, and a variable TOP denoting the top element of the stack, this function returns the value of the i^{th} element from the top of the stack. The element is not deleted by this function.

```
Algorithm to PEEP an element from a stack

Step 1: IF TOP – I + 1< 0, then
           PRINT "Invalid Index"
           Goto Step 3
        [END OF IF]
Step 2: Return S(TOP – I + 1)
Step 3: END
```

Applications of Stack

- Expression conversion
- Expression evaluation
- Recursion

Expression conversion

- Conversion from infix to postfix expression
- Conversion from infix to prefix expression
- Conversion from prefix to infix expression
- Conversion from prefix to postfix expression
- Conversion from postfix to prefix expression
- Conversion from postfix to infix expression

Expressions

- Expressions is a string of operands and operators. Operands are some numeric values and operators are of two types: Unary and binary operators. Unary operators are '+' and '-' and binary operators are '+', '-', '*', '/' and exponential. In general, there are three types of expressions:
  - Infix expression : operand1 operator operand2
  - Postfix expression : operand1 operand2 operator
  - Prefix expression : operator operand1 operand2

Infix	Postfix	Prefix
(a + b)	ab +	+ ab
(a + b) * (c - d)	ab + cd - *	* + ab - cd
(a + b / e) * (d + f)	abe /+ df + *	* + a/be + df

Conversion from infix to postfix

- **Algorithm**
  1. Scan the infix expression from left to right.
  2. If the scanned character is an operand, output it.
  3. Else,
  1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
  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.)
  4. If the scanned character is an '(', push it to the stack.
  5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
  6. Repeat steps 2-6 until infix expression is scanned.
  7. Print the output
  8. Pop and output from the stack until it is not empty.

Example

- **Infix Expression : 3+4*5/6**
- **Postfix Expression : 3 4 5 * 6 / +**

Conversion from infix to postfix (without parenthesis)

```
1.  [Initialize the stack]
        TOP ← 1
        S[TOP] ← '#'
2.  [Initialize output string and rank count]
        POLISH ← ''
        RANK ← 0
3.  [Get first input symbol]
        NEXT ← NEXTCHAR(INFIX)
4.  [Translate the infix expression]
        Repeat thru step 6 while NEXT ≠ '#'
5.  [Remove symbols with greater or equal precedence from stack]
        Repeat while f(NEXT) ≤ f(S[TOP])
            TEMP ← POP(S, TOP)   (this copies the stack contents into TEMP)
            POLISH ← POLISH ○ TEMP
            RANK ← RANK + r(TEMP)
            If RANK < 1
            then   Write('INVALID')
                   Exit
6.  [Push current symbol onto stack and obtain next input symbol]
        Call PUSH(S, TOP, NEXT)
        NEXT ← NEXTCHAR(INFIX)
7.  [Remove remaining elements from stack]
        Repeat while S[TOP] ≠ '#'
            TEMP ← POP(S, TOP)
            POLISH ← POLISH ○ TEMP
            RANK ← RANK + r(TEMP)
            If RANK < 1
            then   Write('INVALID')
                   Exit
8.  [Is the expression valid?]
        If RANK = 1
        then   Write('VALID')
        else   Write('INVALID')
        Exit
```

Symbol	*Precedence f*	*Rank r*
+, −	1	−1
*, /	2	−1
a, b, c, ...	3	1
#	0	−

Conversion from infix to prefix (with parenthesis)

- Step 1: Reverse the infix expression and convert '(' to ')' and ')' to '('.
- Step 2: Read this reversed expression from left to right one character at a time.
- Step 3: Rest of the steps remains same as in case of "conversion from infix to postfix (with parenthesis)".
- Step 4: After all the elements are popped, reverse the expression obtained in prefix expression.

Expression Evaluation

- Evaluation of postfix expression
- Evaluation of prefix expression
- Evaluation of infix expression

Evaluation of postfix expression

- Step 1: If char read from postfix expression is an operand, push operand to stack.
- Step 2: If char read from postfix expression is an operator, pop the first 2 operand in stack and implement the expression using the following operations:
  - **pop(opr1) then pop(opr2)**
  - **result = opr2 operator opr1**
- Step 3: Push the result of the evaluation to stack.
- Step 4: Repeat steps 1 to steps 3 until end of postfix expression
- Finally, At the end of the operation, only one value left in the stack. The value is the result of postfix evaluation.

Evaluation of postfix expression

postfix	Ch	Opr	Opn1	Opn2	result	stack
2 7 * 18 – 6 +						
7 * 18 – 6 +	2					2
* 18 – 6 +	7					2 7
18 – 6 +	*	*	7	2	14	14
– 6 +	18					14 18
6 +	–	–	18	14	-4	-4
+	6					-4 6
	+	+	6	-4	2	2

Input:
Postfix expression: 53+62/*35*+
Output:

The result is:

THANK YOU!!

ANY QUESTIONS??