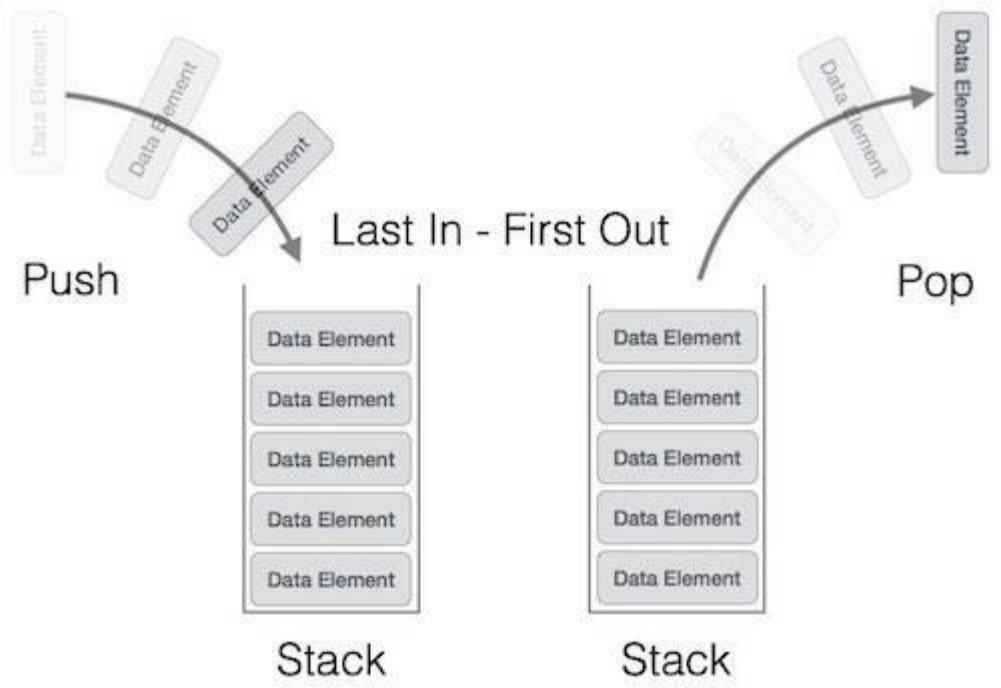
Chapter 2

Stack

BASIC CONCEPTS

- A stack is a linear data structure in which an element may be inserted or deleted at only one end called *top* of the stack.
- Stack is also called **last-in-first-out (LIFO)** list because the element which is inserted last, is removed first.



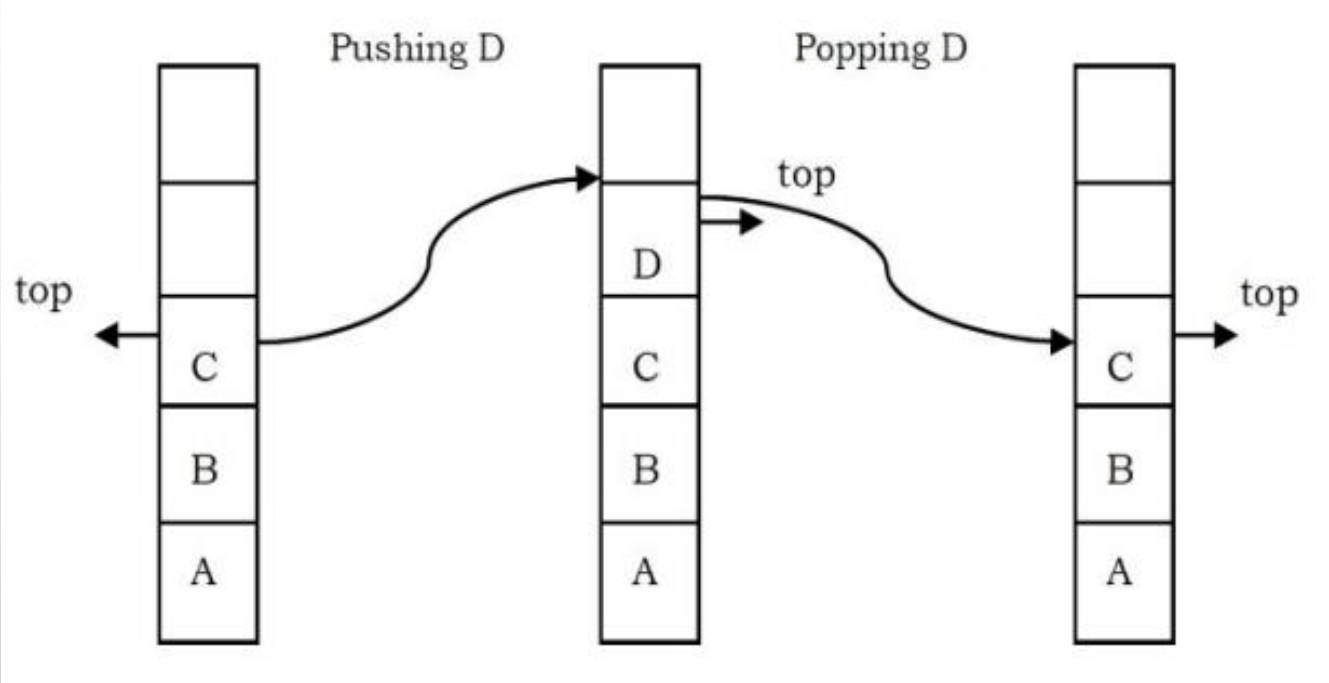
STACK OPERATIONS

- Stack uses a variable called *top* which points topmost element in the stack.
- Insertion operation (inserting an element) is known as *pushing .*
- Removing or deleting operation is known as *popping*.
- The elements are removed from a stack in the reverse order.

STACK OPERATIONS

- Two primary operations are **push** and **pop**.
- In stack terminology, **push** is the insertion operation and **pop** is the removal operation.
- Because of **push** operation, stack is sometimes called a **pushdown list**.

STACK OPERATIONS



- Stack operations

STACK OPERATIONS

- To use a stack efficiently, we also need some additional operations.
  - **peek** – get top data element of the stack, without removing it.
  - **isfull** – check if stack is full.
  - **isempty** – check if stack is empty.

- The result of an illegal attempt to **pop** or **peek** an item from an empty stack is called **underflow**.
- Array implementation may introduce **overflow** if the stack grows larger than the size of the array.

STACK OPERATIONS

- To perform the undo sequence in a text editor.
- To keep the page-visited history in a web browser.
- To evaluate the expressions (postfix, prefix)
- To check the correctness of parenthesis sequence
- To pass the parameters between the functions in a c program
- Matching tags in HTML and XML

IMPLEMENTING STACK

- *Push (int data):* Inserts data onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.
- *int Top():* Returns the last inserted element without removing it. •
- *int Size():* Returns the number of elements stored in the stack.
- *int IsEmptyStack():* Indicates whether any elements are stored in the stack or not.
- *int IsFullStack():* Indicates whether the stack is full or not.

IMPLEMENTATIONS OF STACK

- Stack can be implemented in following two ways:
- Array implementation of stack (or static implementations)
- Linked list implementations (or dynamic)

ARRAY IMPLEMENTATION OF STACK

- In this implementation top is an integer value that indicates the top position of stack
- *Initially it is set to -1 (top = -1)*
- Data is added (push operation) by incrementing top by 1
- Data is removed (pop operation) by decrementing top by 1

- Stack empty or underflow

```
int isempty()
{
    if(top == -1)
            return 1
    else
            return 0
}
```

ARRAY IMPLEMENTATION OF STACK

Stack Full or overflow

```
int isfull()
{
    if (top == SIZE -1)
        return 1
    else
        return 0
}
```

ARRAY IMPLEMENTATION OF STACK

- Algorithm for push operation

1. Start
2. Check for stack overflow as
   if top = SIZE – 1 then
      print “Stack overflow” and exit the program
   else
      Increase top by 1 as
      set, top = top + 1
3. Read element to be inserted say ‘data’
4. Set, stack[top] = element
5. Stop

ARRAY IMPLEMENTATION OF STACK

- Algorithm for pop operation

1. Start
2. Check for stack underflow condition as
   if top == -1 then
      print “Stack is empty” and exit the program
   else
      set a variable ‘data’ to store top element
         data = stack[top]
      remove element by decrementing top by 1
         set, top = top – 1
3. print ‘data’ as deleted item from the stack
4. Stop

C-IMPLEMENTATION USING ARRAY

```
#include <stdio.h>
#define MAXSIZE 10
int stack[MAXSIZE];
int top = -1;
int isempty()
{
        if(top == -1)
                return 1;
        else
                return 0;
}
int isfull()
{
        if(top == MAXSIZE - 1)
                return 1;
        else
                return 0;
}
```

```
void peek()
{
    if(isempty())
        printf("Stack is empty.\n");
    else
        print("Element at top: %d",stack[top]);

}
void pop()
{
        int data;
        if(isempty())
            printf("Stack is empty");
        else
        {
            data = stack[top];
            top = top - 1;
            printf("Popped element: %d",data);
        }
```

C-IMPLEMENTATION USING ARRAY

```
void push(int data)
{
        if(isfull())
            printf("Stack is full.\n");
        else
        {
            top = top + 1;
            stack[top] = data;
        }
}
void display()
{
    if(isempty())
        printf(“Stack is empty”);

    else
    {
        for(int i = top; i>=0,i--)
            printf(“%d\n”,stack[i]);
    }
```

```
int main()
{
        push(3);
        push(5);
        push(9);
        push(1);
        push(12);
        push(15);
        peek();
        pop();
        pop();
        peek();
        display();
}
```

ANALYSIS OF STACK OPERATIONS

- While push and pop we always have to insert or remove the data from the top of the stack, which is one step process.

- Below mentioned are time complexities for various operations:

  - Push operation: O(1)
  - Pop operation: O(1)
  - Top operation: O(1)
  - Search operation: O(n)

YOUR WORK

- Implement stack using array with *'switch case'*

INFIX, PREFIX AND POSTFIX

- One of the applications of the stack is to evaluate the expression. We can represent expression following three types of notation:
- Infix expression
- Prefix expression
- Postfix expression

- *Infix expression:* It is an ordinary mathematical notation of expression where operator is written in between operands.

A + B (A+B) + (C – D)

INFIX, PREFIX AND POSTFIX

- *Prefix notation:* In this notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation.

  +AB ++AB – CD

- *Postfix notation:* In this notation the operators are written after the operands . In this notation operator follows the two operand.

  AB+ AB + CD -+

INFIX, PREFIX AND POSTFIX

- Prefix and postfix notations *are methods of writing mathematical expressions without parenthesis.*
- Time to evaluate a postfix and prefix expression is O(n), where n is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

PRECEDENCE AND ASSOCIATIVITY

- When there are multiple operators in an expression then *precedence* determines which of the operator in the expression is evaluated first.
- For eg: A + B * C then B * C is evaluated first then A is added.

- *Associativity:* When operators with same precedence occurs in an expression then associativity determines which part of the expression to be evaluated first.
- For eg: A + B - C

PRECEDENCE AND ASSOCIATIVITY

- Table : See in book for all operators
- Some are here:

Token	Operator	Associativity
( )	Parenthesis	Left-to-right
^	Exponential	Right-to-left
/ , *	Multiplication and division	Left-to-right
+, -	Addition and subtraction	Left-to-right

ALGORITHM TO CONVERT INFIX TO POSTFIX NOTATION

- Two stacks are used: operator stack(opstack) and postfix stack(posstack)
- The opstack is used to store operators of given expression and posstack is used to store converted corresponding postfix expression:

1. Start
2. For given arithmetic infix expression, scan each symbol (sym) from left to right
3. Repeat till there is data in infix expression

   If sym is operand then push it to posstack

   If sym is '(' then push it to the opstack

ALGORITHM TO CONVERT INFIX TO POSTFIX NOTATION

If sym is operator then

Check the precedence with operators in opstack,

If sym has higher precedence or opstack is empty, push it on operator stack

Else

pop all the operators having higher or equal precedence form opstack and push it on posstack , push the scanned sym to opstack

4. If sym is ‘)’ then pop from opstack and push to posstack until ‘(‘ is found, ignore both symbols.
5. Pop and push into posstack until opstack is empty.
6. Stop

CONVERT INFIX TO POSTFIX NOTATION

- A + B * C – D + E
- A * B – C % D + E
- (A + B) * ((C – D) + E) / F
- A – B + C * (D – E/F) ^ G
- A * (B * C + (D – E)) / F + G ^ H

ALGORITHM TO CONVERT INFIX TO PREFIX NOTATION

- Two stacks are used: operator stack(opstack) and prefix stack(prestack)
- The opstack is used to store operators of given expression and prestack is used to store converted corresponding prefix expression:

1. Start
2. For given arithmetic infix expression, scan each symbol (sym) from right from right to left
3. Repeat till there is data in infix expression

   If sym is operand then push it to posstack

   If sym is ‘)‘ then push it to the opstack

ALGORITHM TO CONVERT INFIX TO POSTFIX NOTATION

If sym is operator then

Check the precedence with operators in opstack,

If sym has higher precedence or opstack is empty,

push it on operator stack

Else

pop all the operators having higher

precedence form opstack and push it on

prestack , push the scanned sym to opstack

4. If sym is ‘(’ then pop from opstack and push to prestack until ‘)‘ is found, ignore both symbols.
5. Pop and push into prestack until opstack is empty.
6. Stop

CONVERT INFIX TO PREFIX NOTATION

- A + B * C – D + E
- A * B – C % D + E
- (A + B) * ((C – D) + E) / F
- A – B + C * (D – E/F) ^ G
- A * (B * C + (D – E)) / F + G ^ H

EVALUATING POSTFIX AND PREFIX EXPRESSION

- Prefix and postfix expressions can be evaluated faster than an infix expression
- Because there is *no need to process any brackets* or *follow precedence rule.*
- In postfix and prefix expression which ever operator comes before will be evaluated first, irrespective to its priority.

EVALUATING POSTFIX EXPRESSION

1. Scan one symbol at a time from left to right of given postfix expression
2. If scanned symbol is operand then
    2.1 Push it to opstack
3. If scanned symbol is operator then
    3.1 Pop opstack and place it into A
    3.2 Pop opstack and place it into B
4. Compute result as *B operator A* and push result into opstack
5. Pop opstack and display which is required value of the given postfix expression
6. Stop

EVALUATING POSTFIX EXPRESSION

- Example:

9 2 3 + - 2 8 4 / + *	
= 9 5 – 2 8 4 / + *	(2 + 3)
= 4 2 8 4 / + *	(9 – 5)
= 4 2 2 + *	(8 / 4)
= 4 4 *	(2 + 2)
= 16	(4 * 4)

- Evaluate using stack

9 2 3 + - 2 8 4 / + *

4, 5, 2, *, 10, 6 -, +, *, 2, /, 3, 4, ^, +

EVALUATING PREFIX EXPRESSION

1. Scan one symbol at a time from right to left of given prefix expression
2. If scanned symbol is operand then
    2.1 Push it to opstack
3. If scanned symbol is operator then
    3.1 Pop opstack and place it into A
    3.2 Pop opstack and place it into B
4. Compute result as *A operator B* and push result into opstack
5. Pop opstack and display which is required value of the given prefix expression
6. Stop

EVALUATING PREFIX EXPRESSION

Example: (Evaluate using stack)

1. * 1 + 5 - / 9 3 2
2. + 1 - / 2 $ 3 2 * 1 + 5 - / 9 3 2

EVALUATING PREFIX EXPRESSION

Example:

1. * 1 + 5 - / 9 3 2

= * 1 +5 – 3 2	(9 / 3)
= * 1 + 5 1	(3 - 2)
= * 1 6	(5 + 1)
= 6	(1 * 6)

EVALUATING INFIX EXPRESSION

Example:

1. A + B – C * D A = 1, B = 4, C = 2, D = 8
2. A * B – C + D
3. A + B – C * (D + E – F)

Hints:

1. Implement two stack operator stack and operand stack
2. Scan one symbol from left to right
3. If symbol is operand push it to operand stack
4 . If symbol is operator push it to operator stack, if scanned operator has higher than operator in operator stack
5. Perform operation by popping operand from operand stack

THANK YOU !