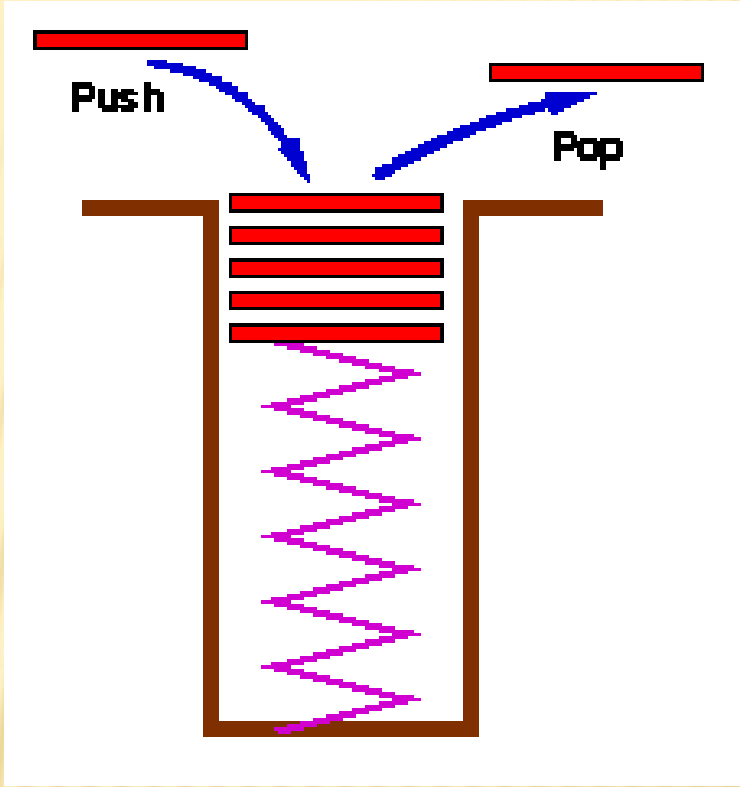
STACK

STACK

- A data structure to store data in which the elements are added and removed from one end only: a Last In First Out (LIFO) data structure

- Real life examples
  - Stack of coins
  - Stack of books
  - Stack of plates
  - Stake of bags


Push
Pop

STACKS

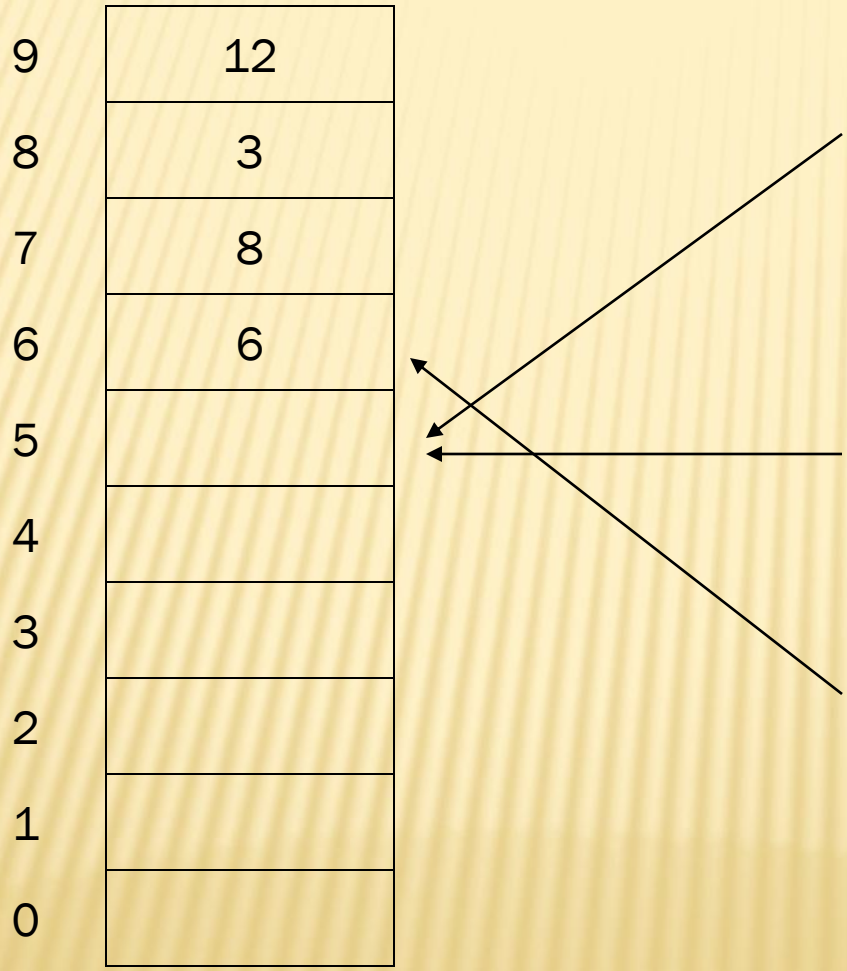
- The operations defined on a stack are:
  1. Push - Store onto a stack
  2. Pop - retrieve from stack
  3. Top - examine the top element in the stack
  4. Is_empty - check if the stack is empty
  5. Is_Full - check if the stack is full
- A stack can be very easily implemented using arrays.
- Stack is implemented by maintaining a pointer to the top element in the stack. This pointer is called the *stack pointer.*

STACKS – ARRAY IMPLEMENTATION

If a stack is implemented using arrays, the following two conventions can be used:

1. A stack can grow upwards, i.e., from index 0 to the maximum index, or it can grow downwards, i.e., from the maximum index to index 0.
2. *Stack pointer* can point to the last element inserted into the stack or it can point to the next available position.

GROWING DOWNWRDS

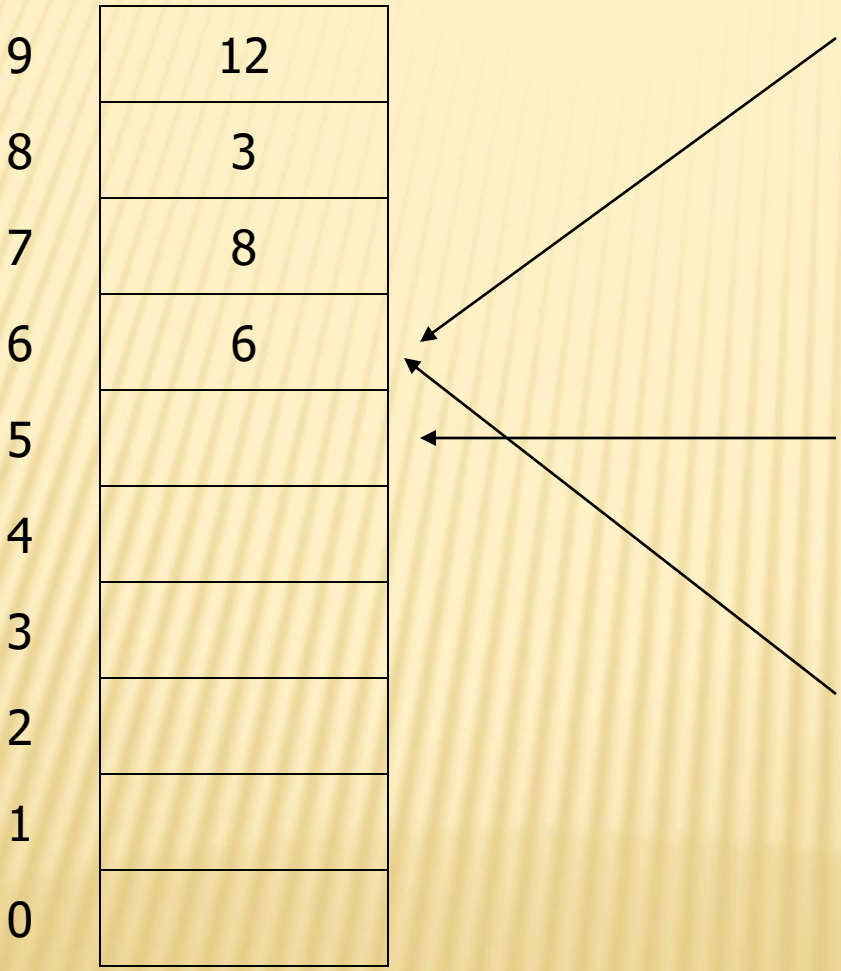


Initial state: stk_ptr = MAX - 1

- stk_ptr points to the next empty location
- Push – first add data to the stack, then decrement stk_ptr
- Pop – first increment stk_ptr and then take data out

GROWING DOWNWRDS

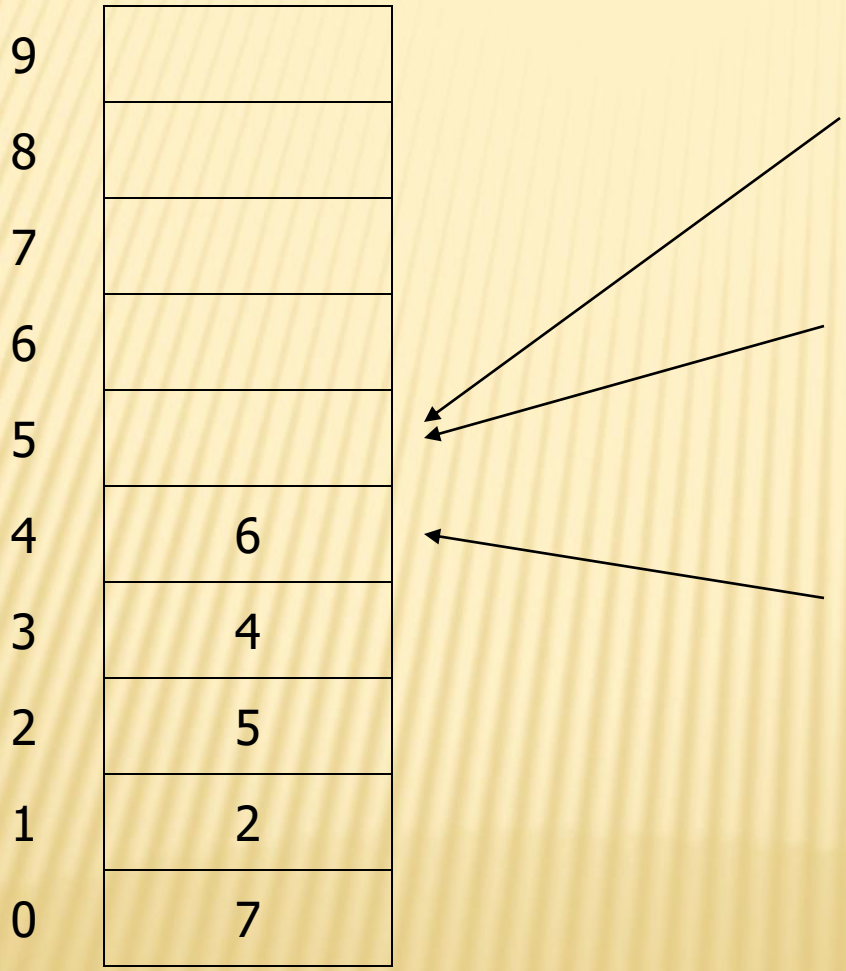
Initial state: stk_ptr = MAX



- stk_ptr points to the last element added to the stack
- Push – first decrement the stk_ptr and then add data
- Pop – first take data out and then increment stk_ptr

GROWING UPWRDS

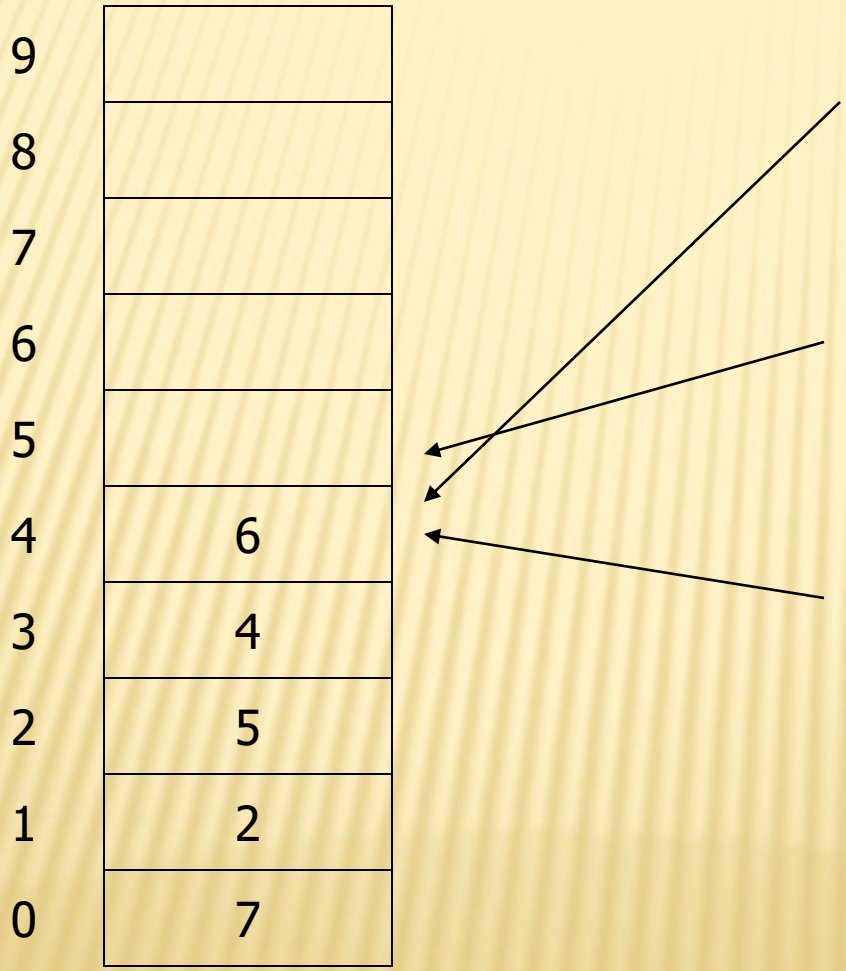
Initial state: stk_ptr = 0



- stk_ptr points to the next empty location
- Push – first add data to the stack then increment stk_ptr
- Pop – first decrement stk_ptr and then take data out

GROWING UPWRDS

Initial state: stk_ptr = -1



- stk_ptr points to the last element added to the stack
- Push – first increment the stk_ptr and then add data
- Pop – first take data out and then decrement stk_ptr

STACKS – ARRAY IMPLEMENTATION

```
class Stack {
private:
    int size;                          // maximum storage capacity
    int stk_ptr;                       // stack pointer
    int *stackArray;                   // array used to implement stack
public:
    Stack(int s );                     // constructor
    ~Stack() {delete [ ] stackArray; } // destructor
    bool push (int);                   // add an element to the stack
    bool pop(int &);                   // remove an element from stack
    bool isFull();                     // check if the stack is full
    bool isEmpty();                    // check if the stack is empty
};
```

```
Stack::Stack(int s)
{
    size = s;
    stk_ptr = 0;
    stackArray = new int[size];
}
```

```
bool Stack::ifEmpty()
{
    return (stk_ptr == 0);
}
```

```
bool Stack::ifFull()
{
    return (stk_ptr == size);
}
```

```
bool Stack::push(int n)
{
    if (! isFull() ) {
        stackArray[stk_ptr] = n;
        stak_ptr = stk_ptr + 1;
        return true;
    }
    else return false;
}
```

```
bool Stack::pop(int &n)
{
    if (! isEmpty() {
        stk_ptr = stk_ptr – 1;
        n = stackArray[stk_ptr];
        return true;
    }
    else return false;
}
```

APPLICATION OF STACKS EVALUATION OF EXPRESSION

- Evaluation of expression like
  a+b/c*(e-g)+h-f*i was a challenging task for compiler writers.
- It is a problem of parenthesization of the expression according to operator precedence rule.
- A fully parenthesized expression can be evaluated with the help of a stack.

ALGORITHM TO EVALUATE FULLY PARENTHESIZED EXPRESSIONS

1. while (not end of expression) do
    1. get next input symbol
    2. if input symbol is not “)”
        1. push it into the stack
    3. else
        1. repeat
            1. pop the symbol from the stack
        2. until you get “(“
        3. apply operators on the operands
        4. push the result back into stack
2. end while
3. the top of stack is the answer

EVALUATION OF FULLY PARENTHESIZED EXPRESSION

(a+(b/c))
Assuming a=2, b=6, c=3

Input Symbol	Stack	Remarks
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push
c	(a+(b/c	Push
)	(a+2	Pop”(b/c” and evaluate and push the result back
)	4	Pop”(a+2” and evaluate and push the result back

EVALUATION OF EXPRESSIONS

- The normal way of writing expressions i'.e., by placing a binary operator in-between its two operands, is called the *infix* notation.
- It is not easy to evaluate arithmetic and logic expressions written in infix notation since they must be evaluated according to operator precedence rules. E.g., a+b*c must be evaluated as (a+(b*c)) and not ((a+b)*c).
- The *postfix* or *Reverse Polish Notation* (RPN) is used by the compliers for expression evaluation.
- In RPN, each operator appears after the operands on which it is applied. This is a parenthesis-free notation.
- Stacks can be used to convert an expression from its infix form to RPN and then evaluate the expression.

APPLICATION OF STACKS:

- Post Expression calculator

Infix Expression	Eqaivalent Postfix Expression
a+b	ab+
a+b*c	abc*+
a*b+c	ab*c+
(a+b)*c	ab+c*
(a-b)*(c+d)	ab-cd+*
(a+b)*(c-d/e)+f	ab+cde/-*f+

INFIX AND POSTFIX

Infix	Postfix
a+b*c	abc*+
a*b+c*d	ab*cd*+
(a+b)*(c+d)/e-f	ab+cd+*e/f-
a/b-c+d*e-a*c	ab/c-de*+ac*-
a+b/c*(e+g)+h-f*i	abc/eg+*+h+fi*-

ALGORITHM TO EVALUATE EXPRESSIONS IN RPN

1. while (not end of expression) do
    1. get next input symbol
    2. if input symbol is an operand then
        1. push it into the stack
    3. else if it is an operator then
        1. pop the operands from the stack
        2. apply operator on operands
        3. push the result back onto the stack
2. End while
3. the top of stack is answer.

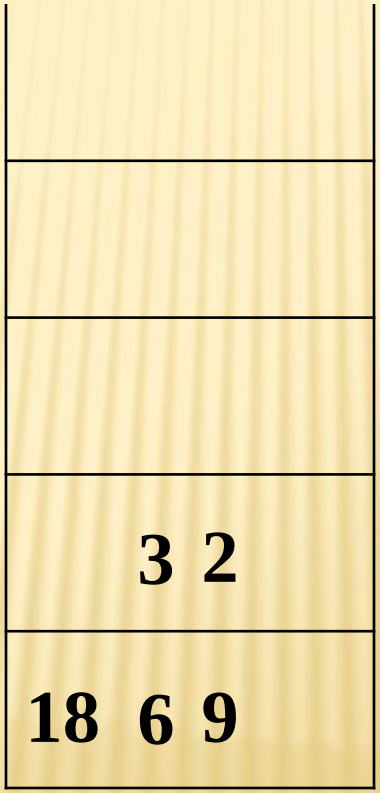
POST EXPRESSION CALCULATOR

- Expression: 6 3 + 2 * = 18

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

6+3 = 9

9*2 = 18



1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times
7. Push result = 18
8. Symbol is = so pop & display

ALGORITHM TO EVALUATE EXPRESSIONS IN RPN

(a+b)*(c+d) → ab+cd+*
Assuming a=2, b=6, c=3, d=-1

Input Symbol	Stack	Remarks
a	a	Push
b	a b	Push
+	8	Pop a and b from the stack, add, and push the result back
c	8 c	Push
d	8 c d	Push
+	8 2	Pop c and d from the stack, add, and push the result back
*	16	Pop 8 and 2 from the stack, multiply, and push the result back. Since this is end of the expression, hence it is the final result.