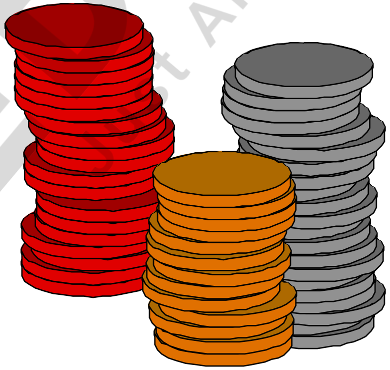
STACKS



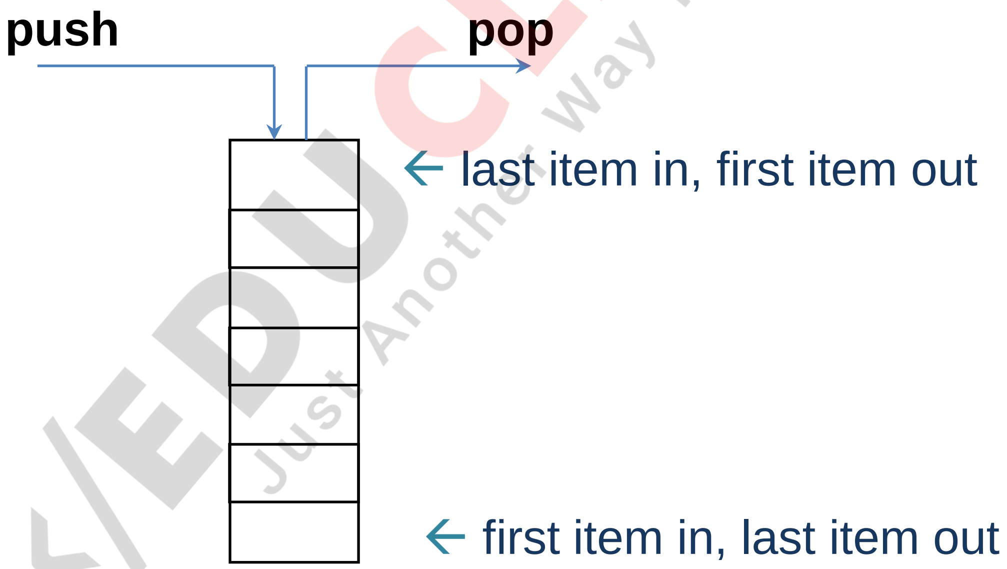
What is a stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT= LIFO

  i.e the last element inserted is the first one to be removed
- Example
  - Stack of plates
  - Stack of coins

stack data structure

Stacks often are drawn vertically:



push

pop

← last item in, first item out

← first item in, last item out

Basic Stack Operations

- Push:
  - Adds an item at the top of the stack.
  - If the stack is full, no more data can be added to the stack and the stack is said to be in the **overflow state**.
  - Diagram

Algorithm to push data into stack

```
Algorithm  push(struct stack *s , int item)
To push the data into the stack using array implementation
 pre:  struct stack *s : pointer to the stack structure
       item : data to be pushed in  the stack
post: push the data into the array
1.  if s->top = ARR - 1
     display "STACK IS FULL"
     return
2.  [increment top by 1]
         s->top++
3. [insert data into the stack]
     s->a[s->top]=item
     count =count +1
```

Basic Stack Operations

- Pop:
  - Removes an item at the top of the stack.
  - When the last item is deleted , the stack must be set to **empty state**. If pop() is called when the stack is empty, it is said to be in the **underflow state**.
  - Diagram

Algorithm to pop data from the stack

```
Algorithm int pop(struct stack *s)
pre : To pop the data from the stack using array
    implementation
      struct stack *s : pointer to the stack structure
post : return the popped data to the main()
return data
1. [declare a variable]
    int data
2. if s->top =-1
          return NULL
3. [remove data from the top of the stack]
    data=s->a[s->top]
    s->top—
   count--
    return data
```

Basic Stack Operations

- Stack top or peep
  - It returns the data at the top of the stack but does not delete it. i.e it only reads the data.
  - if the stack is empty, stack top can result in **underflow state.**

Algorithm to peep/read data from the stack

Algorithm int peep(struct stack *s)

To peep/read the data from the stack using array implementation

pre: struct stack *s : pointer to the stack structure

post: return the peeped data to the main()

Return data

Refer to pop() algorithm and make the necessary changes

Algorithm to display the stack

```
Algorithm displaystack(struct stack *s)
Pre : struct stack *s : pointer to the stack structure
post : Display the contents of the stack
1.[intialize]
    int x=count;
2. Repeat while x>=0
        1 . Display s->a[x]
        2.      x--;
```

Application of stacks

- Region in memory within which the programs temporarily store data as they execute.
- Evaluation of expressions:
  - Process of writing the operators of an expression either before their operands or after their operands is called as “polish notation”
  - 3 forms of polish notation
    - Prefix form : the operators come before operands
    - Postfix form : the operators come after operands
    - Infix form: the operator come in between operands

Algorithm for converting infix to postfix

```
Algorithm infix_to_postfix()
1. Push “(“ onto STACK and add “)” to the end of expression A.
2. Scan expression Q from left to right and repeat 1 to 6 for each element of Q until the stack is empty.
    1. If an operand is encountered , add it to Stack B
    2. If a “(“ is encountered push it onto the stack A.
    3. If an operator is encountered then
        A) if operator in the stack A has same precedence or higher precedence than the operator encountered then
            1. Repeatedly pop the operators from the STACK A and add to Stack B each operator
        B) Add the encountered operator to STACK A.
    6. If “)” is encountered then
        A) Repeatedly pop from the STACK A and add to B each operator(on the top of STACK) until a “(“ is encountered.
        B) Remove the “(“
```

Evaluation of postfix expression

```
Algorithm evaluate_postfix()
    1. createStack(stack)
    2. Loop(for each character)
          If(character is operand)
              1. PushStack(stack,character)
          else
              1. set oper2= popStack(stack)
              2. set oper1=popStack(stack)
              3. operator=character
              4. set value = calculate
                        (oper1,operator,oper2)
              5. pushStack(stack ,value)
          endif
       end loop
    3. result =popStack(stack)
    4. Return result
End evaluate_postfix
```

Algorithm for converting infix to prefix

```
Algorithm infix_to_prefix(s[])
1.  Get the infix expression s.
2.  set i=0
3.  Set top1=top2=-1, indicating
    stacks are empty.
4.  If s[i]='(' , push it in stack1, go
    to step 8
5.  If s[i]=operand , push it in
    stack2, go to step 8
6. If s[i]=operator
        stack1 is empty or stack
        top elements has less
        priority as compared o
        s[i],
            add operator to the stack1
            go to step 8
    else
            p=  pop the operator
                from the stack1
            O2=pop the operand from
            stack2
            O1=pop the operand from
            stack2
            form the prefix expr
            p,O1,O2
            push operator in stack2 and
            go to step 8
    End if
```

Cont

Algorithm for converting infix to prefix

```
7. If s[i]=')' then
     A)  p=pop the operator
         from stack1
            O2=pop the
               operand from
               stack2
            O1=pop the
               operand from
               stack2
            form the prefix expr
             p,O1,O2
            push in stack2 and go to
               step 7A
     B)
         remove "("
         go to step8
8. Increment i
9. If s[i] <>'\0' then go to step 4
10. Everytime pop one operator
    from stack1, pop 2 operands
    from stack 2 , form the prefix
    expr ,O1, O2 , push in stack2
    and repeat till stack becomes
    empty.
11. Pop operand from stack2 and
    print it as expression
12. stop
```

More Applications of stacks

- Parenthesis matching
- Towers of Hanoi
- Rearranging Railroad cars
- Switch box routing
- Rat in a maze