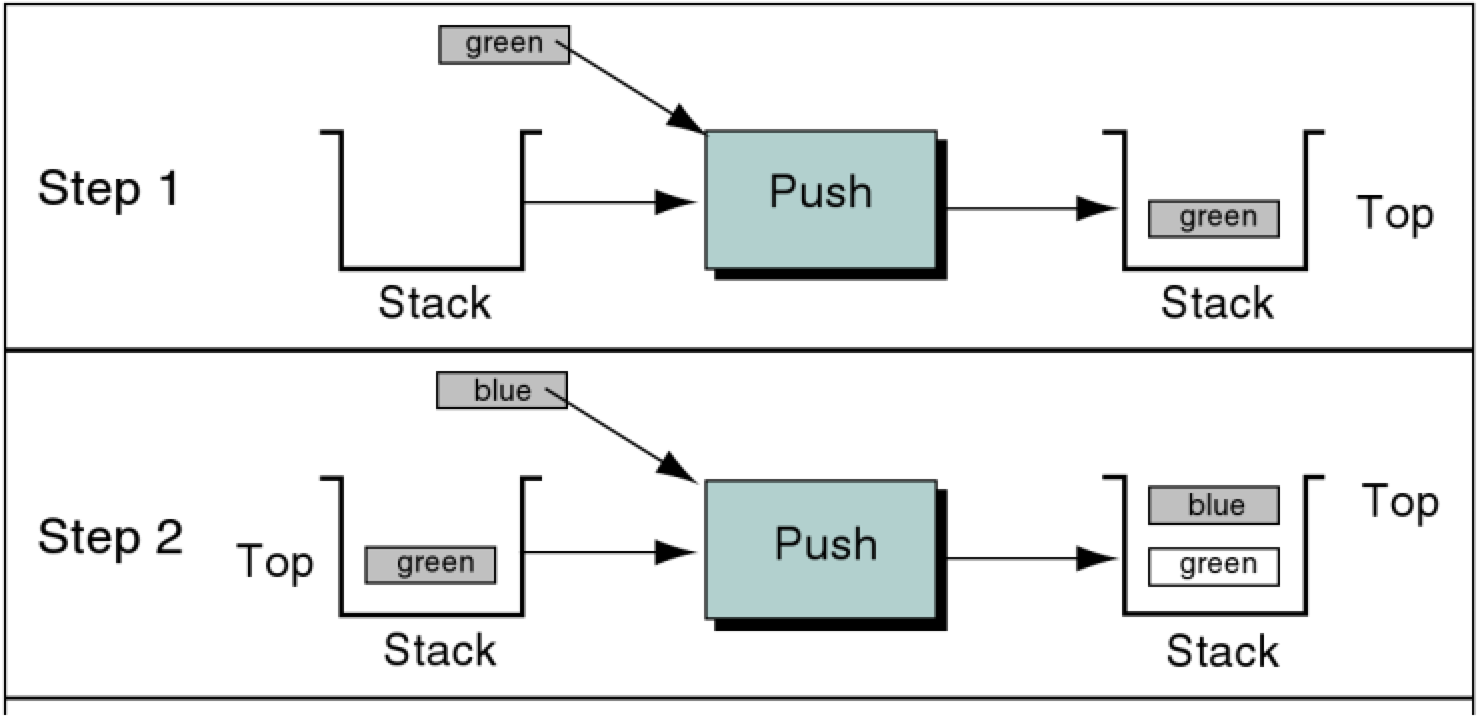
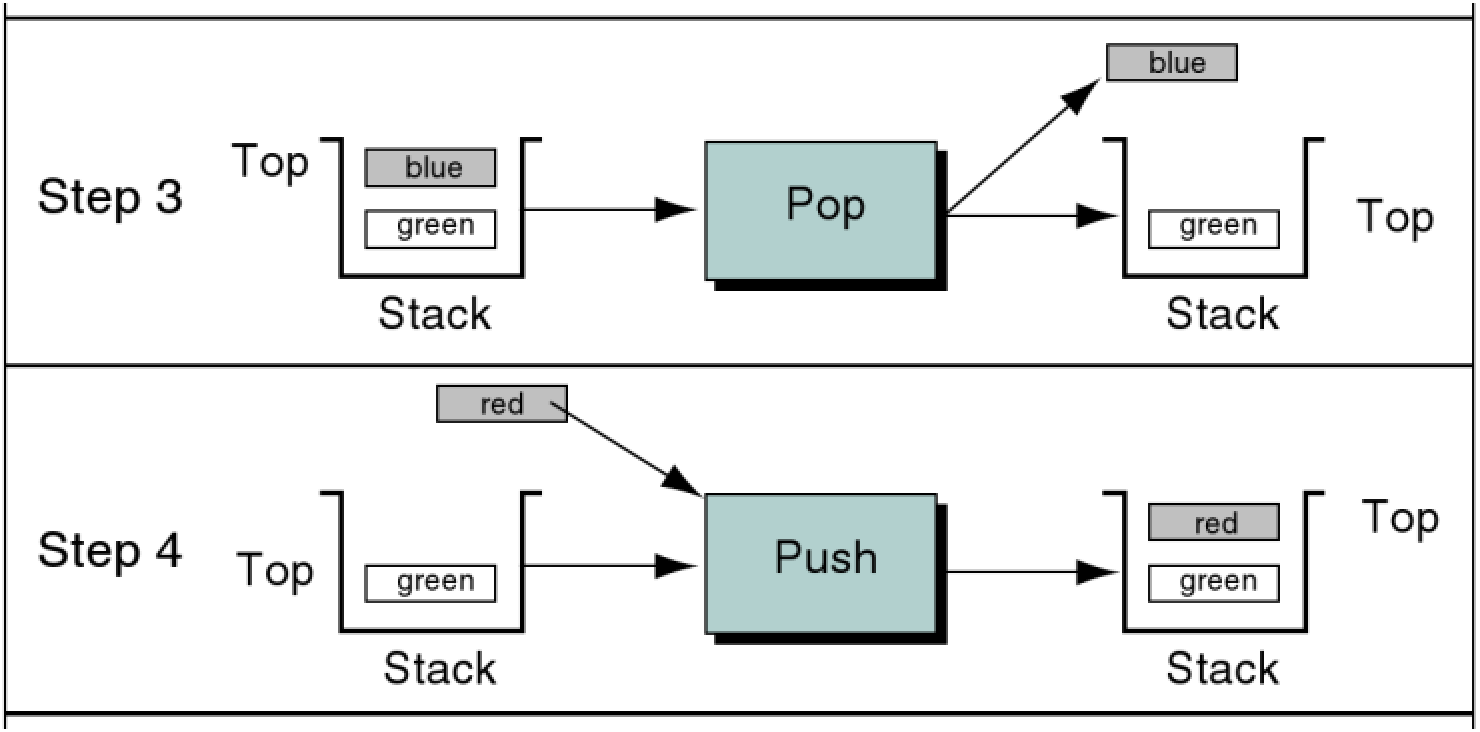
Introduction To Stack

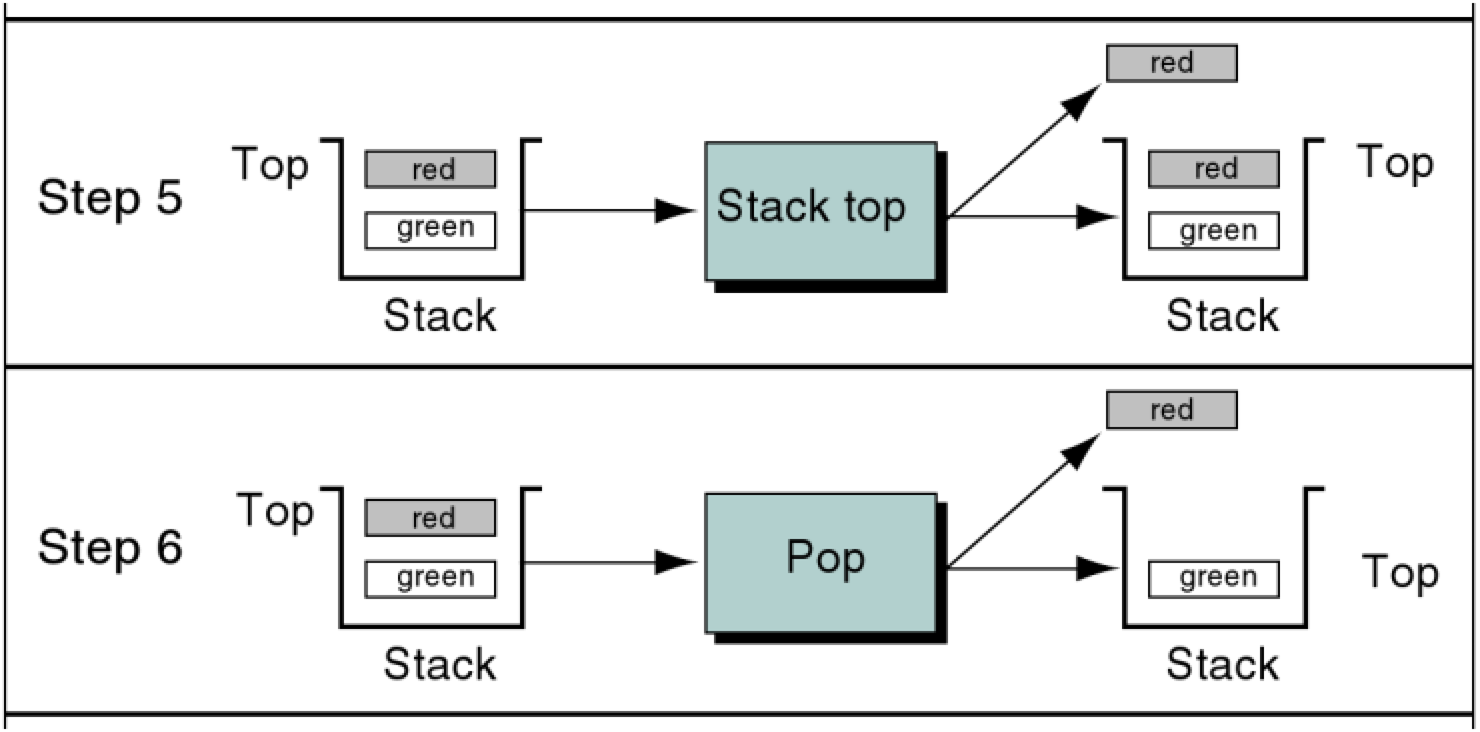
By Yash Gupta

Operations

- Push
  - – Inserts object on stack
- Pop
  - – Deletes the object from top of stack
- StackTop
  - – Shows the object on top of stack without deleting
- isEmpty
  - – Determines if stack is empty (underflow) or full (overflow)


green
Step 1
Stack
Push
green
Top
Stack
blue
Step 2
Top
green
Stack
Push
blue
green
Top
Stack

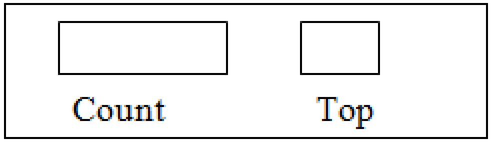

Step 3
Top
blue
green
Stack
Pop
blue
green
Top
Stack
red
Step 4
Top
green
Stack
Push
red
green
Top
Stack


Step 5
Top
red
green
Stack
Stack top
red
red
green
Top
Stack
Step 6
Top
red
green
Stack
Pop
red
green
Top
Stack

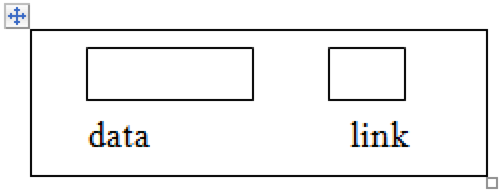
Link List Implementation

- Stack Head
  - Top pointer
  - count of the no of elements in the stack

- Stack Data Node
  - Data Node contains data
  - Link Node is a pointer



StackHead



StackNode

```
Algorithm createStack
Creates and initializes meta data structure
    Pre : Nothing
    Post : structure created and initialized
    Return : stackhead
Allocate memory for stackhead
Set count to 0
Set top to NULL
return stackhead
end createStack
```

```
Algorithm pushStack(stack, d )
Inserts one item into the stack
Pre : stack passed by reference
      : d contains data to be pushed onto the stack
Post : data have been pushed on the stack
Allocate new node
 node->data = d
 node->next=stack->top
 stack->top=node
 (stack->count)++
end pushStack
```

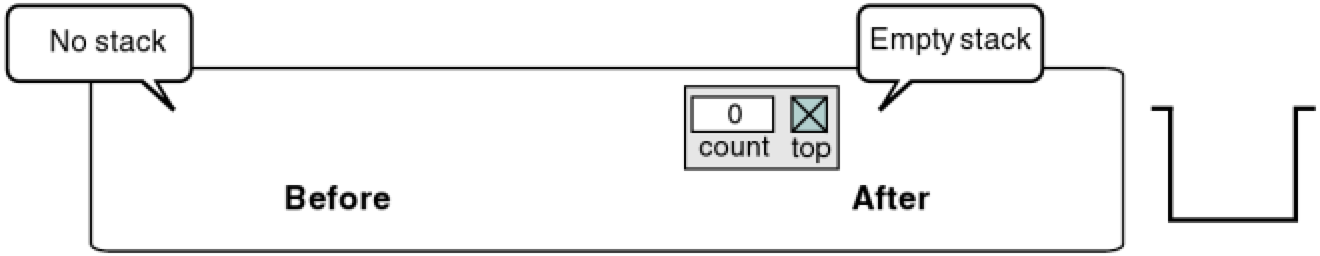
```
Algorithm popStack(stack )
Removes top item from stack
    Pre : stack passed by reference
    Post : top item data returned
if( isEmpty(stack) )
     return -1
else
     create temporary node
     node =stack->top;
     stack->top=node->next;
     set d to node->data
     free(node)
     (stack->count)--
     return d
end if
end popStack
```

```
Algorithm isEmpty(stack )
    Checks the status of stack
    Pre : stack passed by reference
    Post : True if stack empty and false if stack is not empty
if(stack->count==0)
     return true
else
     return false
end if
end isEmpty
```

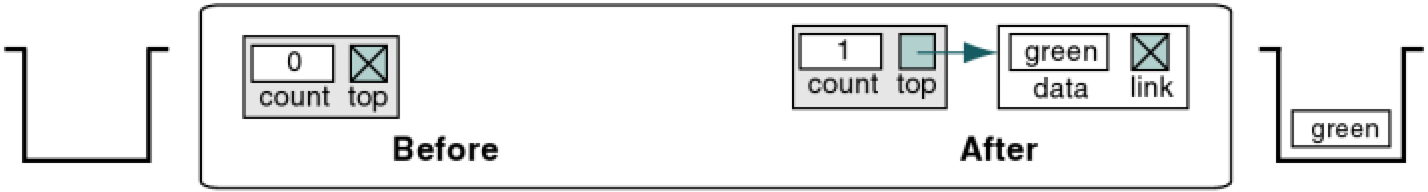
```
Algorithm stackTop(stack )
    Returns the top of stack without deletion
    Pre : stack passed by reference
    Post : Return Stacktop node
if( isEmpty(stack) )
     return NULL
else
     return stack->top
end if
end stackTop
```

```
Algorithm stackCount(stack)
    Returns the count of nodes currently in stack
    Pre : stack passed by reference
    Post : Returns the stack count
return stack->count
end stackCount
```

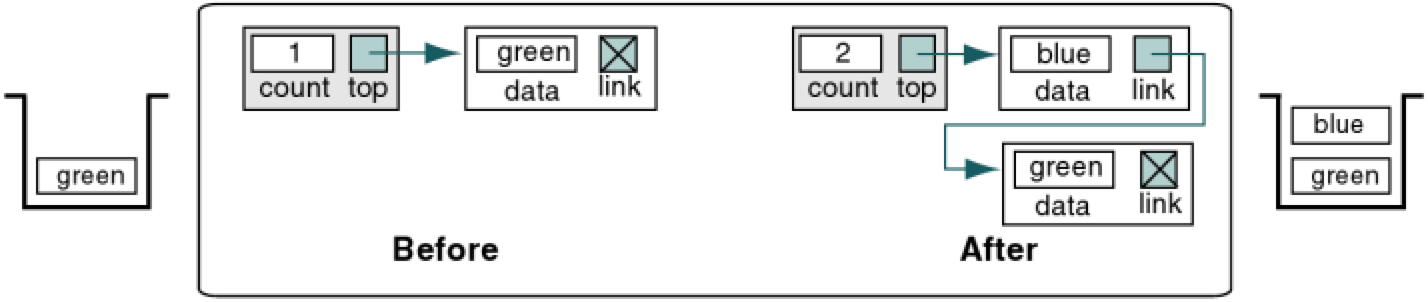
```
Algorithm destroyStack(stack)
Deletes all the nodes currently in stack
   Pre : stack passed by reference
   Post : Clears the stack
while( !isEmpty(stack) )
    delete top node
end while
delete stackhead

end destroyStack
```



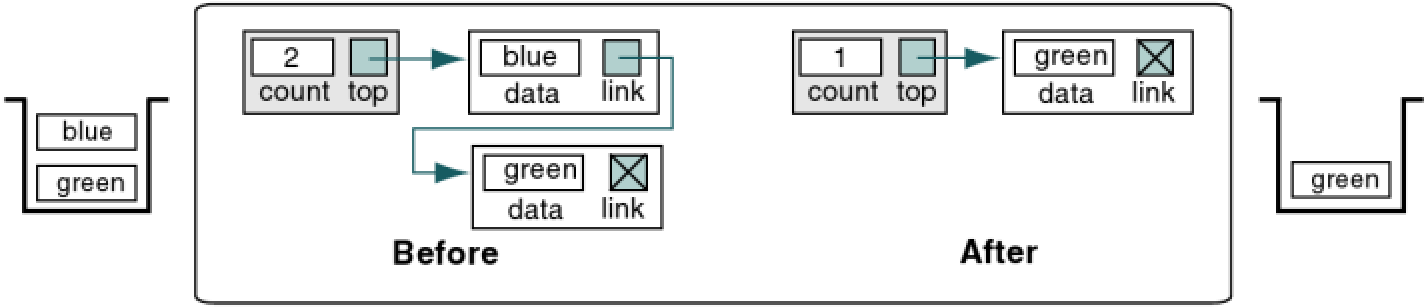
Create stack



Push stack



Push stack



Pop stack

Time Complexity

Push	Pop	StackTop
$T(n) = c$ $= O(1)$	$T(n) = c$ $= O(1)$	$T(n) = c$ $= O(1)$

Applications

- Expression conversion and evaluation
- Backtracking
  - Goal Seeking
  - N-Queens Problem

Expression Types

- Infix expression for humans
- Prefix and postfix expression for computers

Notation	**Description**	**Format**	**Example**
Infix	The operator comes between the two operands	Operand1 operator Operand2	A+B
Postfix	The operator comes after the two operands	Operand1 Operand2 operator	AB+
Prefix	The operator comes before the two operands	Operator Operand1 Operand2	+AB

Operator Priority & Precedence

Operator	**Associativity**
( )	Left – Right
^ or &	Left – Right
/ , *	Left – Right
+ , -	Left – Right

Manual Transformation

- Infix to Postfix
- Infix to Prefix
- Example : (A+B)*(C*D-E)*F/G

Infix-Postfix

Step 1 : Left-Bracket Pair is evaluated first

= **AB+** * (C * D – E) * F / G

Step 2 : Next bracket is evaluated in which multiplication is transformed first

= AB+ * (**CD*** - E) * F / G

Now the subtraction operation is transformed remaining in bracket

= AB+ * **CD*E -** * F / G

Step 3 : Now there are two multiplications & one division operation but both multiply & divide have equal priority and precedence is left-right so transformed from left-right

= AB+ CD*E -* * F / G

= AB+ CD*E -* F* / G

= AB+ CD*E -* F* G/

Infix-Prefix

Step 1 : Left-Bracket Pair is evaluated first

= +**AB** * (C * D – E) * F / G

Step 2 : Next bracket is evaluated in which multiplication is transformed first

= +AB * (***CD** - E) * F / G

Now the subtraction operation is transformed remaining in bracket

= +AB * -***CDE** * F / G

Step 3 : Now there are two multiplications & one division operation but both multiply & divide have equal priority and precedence is left-right so transformed from left-right

= *+**AB-* CDE** * F / G

= ****+AB-* CDE**F / G

= **/**+AB-* CDEFG**

Algorithmic Transformation

- Algorithms used
  - getPriority
  - isOperator
  - Infixtopostfix or infixtoprefix

```
Algorithm getPriority(char)
Returns the priority of operator
Pre       : char contains the operator whose priority is to be determined
Post      : Priority is evaluated
Return    : Corresponding Priority number returned
if( char == '^' )
    return 3
else  if(char=='/' || char=='*' )
    return 2;
else if(char=='+' || char=='-' )
    return 1
else
    return 0
end getPrority
```

```
Algorithm isOperator(char)
Checks weather the character is operator or operand
Pre       : char contains the operator/operand
Post      : char is evaluated
Return  : True if char is operator & false if char is operand
if( (ch>='A' && ch<='Z') || (ch>='a' && ch<='z') || (ch>=’0’
    && ch<=’9’) )
        return 0
else
        return 1
end isOperator
```

Infix To PostFix Logic

- Scan string from left to right
- If character *C* is operand
  - Append *C* to *postString*
- Else if character *C* is operator
  - If *C* is ‘)’ pop all operators until ‘(‘ and append to *postString*
  - If *C* has higher priority than operator on stack push *C* on stack
  - If *C* has lower or equal priority than operator on stack pop all the operator on stack which have priority lower than or equal to *C* and append to postfix

Example

- Solve : (A+B)*(C*D-E)*F/G

SCAN INDEX	SCANNING	POSTEXP	STACK
1	(		(
2	A	A	(
3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	(	AB+	*(
8	C	AB+C	*(
9	*	AB+C	*(*
10	D	AB+CD	*(*
11	-	AB+CD*	*(-
12	E	AB+CD*E	*(-
13	)	AB+CD*E-	*
14	*	AB+CD*E-*	*
15	F	AB+CD*E-*F	*
16	/	AB+CD*E-*F*	/
17	G	AB+CD*E-*F*G	/
		AB+CD*E-*F*G/	

```
Algorithm infixTopostfix(infixexp)
Converts an infix expression to postfix expression
    Pre : infixexp contains the string in infix format
    Post : infix expression converted to postfix expression
    Return : postexp in string format
stack=createStack()
for each char in infixexp
    if( isOperator(char) )
        if( isEmpty(stack) || char=='(' )
            push(stack,char)
        else if(char==')' )
            while( stacktop(stack)!='(')
                token=pop(stack)
                concatenate token to postexp
            pop(stack)
        else
            while( getPriority(char) <= getPriority( stacktop(stack) ) )
                token=pop(stack)
                concatenate token to postexp
            push(stack,char)
        end if
    else
        concatenate char to postexp
    end if
end for
```

```
while( !isEmpty(stack) )
        token=pop(stack)
        concatenate token to postexp
return postexp
end infixTopostexp
```

Infix To Prefix Logic

- Reverse Input String
- Scan string from left to right
- If character C is operand
  - Append C to *preString*
- Else if character C is operator
  - If C is ‘(‘ pop all operators until ‘)’ and append to *preString*
  - If C has higher or equal priority than operator on stack push C on stack
  - If C has lower priority than operator on stack pop all the operator on stack which have priority lower than or equal to C and append to *preString*
- Reverse *preString*

Infix To Prefix

- Infix exp

  (A+B)*(C*D-E)*F/G
- Reverse

  G/F*)E-D*C(*)B+A(

SCANNING	PREEXP	STACK
G	G	
/	G	/
F	GF	/
*	GF	/*
)	GF	/*)
E	GFE	/*)
-	GFE	*)-
D	GFED	/*)-
*	GFED	/*)-*
C	GFEDC	/*)-*
(	GFEDC*-	/*
*	GFEDC*-	/**
)	GFEDC*-	/**)
B	GFEDC*-B	/**)
+	GFEDC*-B	/*)+
A	GFEDC*-BA	/*)+
(	GFEDC*-BA+	/**
	GFEDC*-BA+**/	
REVERSING	/**+AB-*CDEFG	

```
Algorithm infixTopostfix(infixexp)
Converts an infix expression to prefix expression
    Pre : infixexp contains the string in infix format
    Post : infix expression converted to prefix expression
    Return : preexp in string format
stack=createStack()
reverse infixexp
for each char in infixexp
    if( isOperator(char) )
        if( isEmpty(stack) || char==')' )
            push(stack,char)
        else if(char=='(' )
            while( stacktop(stack)!=')')
                token=pop(stack)
                concatenate token to preexp
            pop(stack)
        else
            while( getPriority(char) < getPriority( stacktop(stack) ) )
                token=pop(stack)
                concatenate token to preexp
            push(stack,char)
        end if
    else
        concatenate char to preexp
    end if
end for
```

```
while( !isEmpty(stack) )
    token=pop(stack)
    concatenate token to preexp
Reverse preexp
return preexp
end infixTopreexp
```

Expression Evaluation

```
Algorithm evaluate(left, right, char)
Performs operation on operands based on operator
Pre   :  left is the leftside operand
       : right is the rightside operand
       : char is the operator
Post : operation is performed
Return : Result of operation is returned
switch(char)
        case '+' : return left + right
        case '-' :  return left - right
        case '*' : return left * right
        case '/'  : return left/right
end switch
end evaluate

end evaluate
```

PostFix Evaluation Logic

- Scan postexp from left to right
- If character C is operand or digit
  - Push on stack
- else if C is operator
  - Push two operand from stack and evaluate
  - The first poped operand is rightoperand and second poped is leftoperand
  - Evaluate Ans =leftoperand operator rightoperand
  - Push Ans on Stack

- InfixExp : (5+2) * 10 - 3
- Solve : 23*4-5+

Postfix Evaluation

POSTEXP=23*4-5+		
SCANNING	EVALUATE	STACK
2		2
3		2,3
*	2*3	6
4		6,4
-	6-4	2
5		2,5
+	2+5	7

```
Algorithm postfixEval(postexp)
Evaluates postfix expression
Pre : postexp contains the string in postfix format
Post : postfix expression evaluated
Return : Result of postfix evaluation
stack=createStack()
for each char in postexp
    if( isOperator(char) )
        right= pop(stack)
        left = pop(stack)
        result=evaluate(left,right,char)
        push(stack,result)
    else
        push(stack,char)
end if
end for
finalresult = pop(stack)
return finalresult
end postfixEval
```

Prefix Evaluation Logic

- Reverse preExp
- Scan preExp from left to right
- If character C is operand or digit
  - Push on stack
- else if C is operator
  - Push two operand from stack and evaluate
  - The first poped operand is leftoperand and second poped is rightoperand
  - Evaluate Ans :leftoperand operator rightoperand
  - Push Ans on Stack

Example

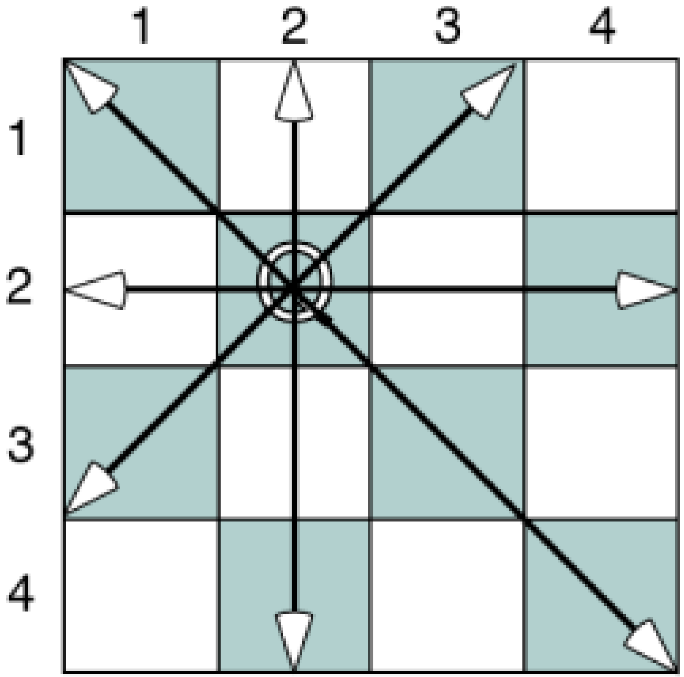
- InfixExp : (5+2) * 10 - 3
- Solve : 543+2*-

Prefix Evaluation

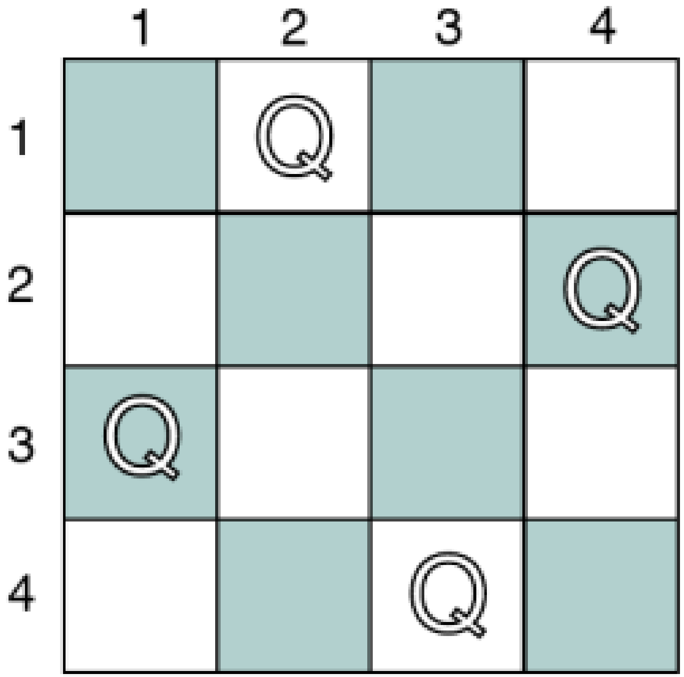
REVPRE = 543+2*-		
SCANNING	EVALUATE	STACK
5		5
4		5,4
3		5,4,3
+	3+4	5,7
2		5,7,2
*	2*7	5,14
-	14-5	9

```
Algorithm prefixEval(postexp)
Evaluates postfix expression
Pre : preexp contains the string in prefix format
Post : prefix expression evaluated
Return : Result of prefix evaluation
stack=createStack()
Reverse the preexp
for each char in preexp
        if( isOperator(char) )
                left= pop(stack)
                right = pop(stack)
                result=evaluate(left,right,char)
                push(stack,result)
        else
                push(stack,char)
end if
end for
finalresult = pop(stack)
return finalresult

end prefixEval
```

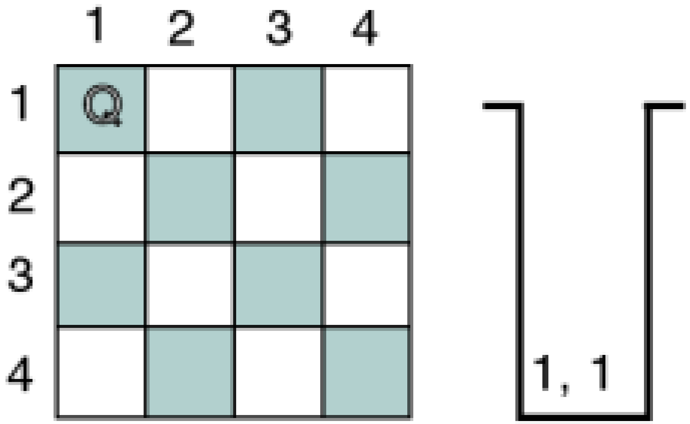
BackTracking N-Queen Problem



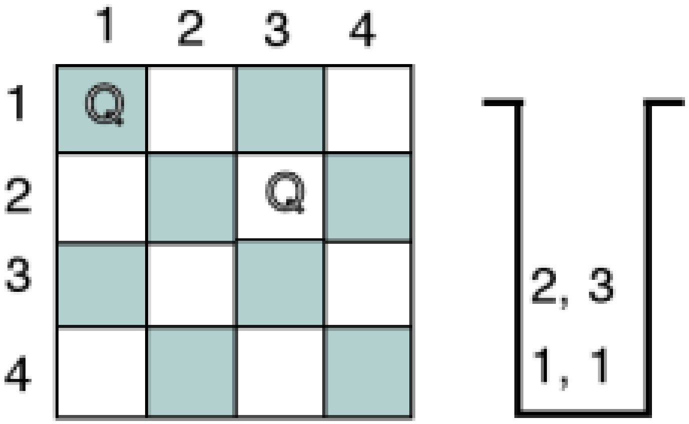
(a) Queen capture rules



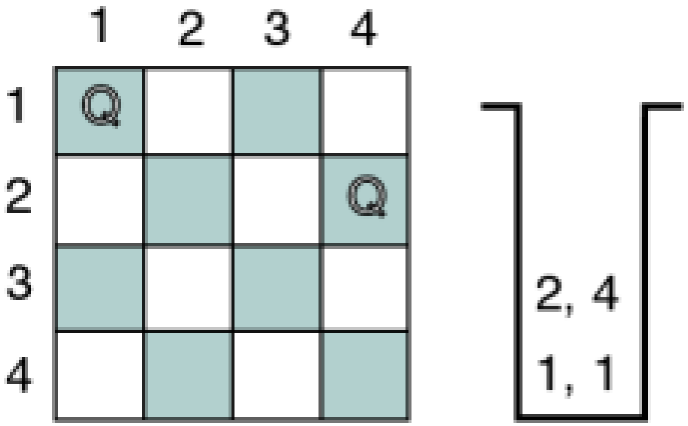
(b) First four queens solution



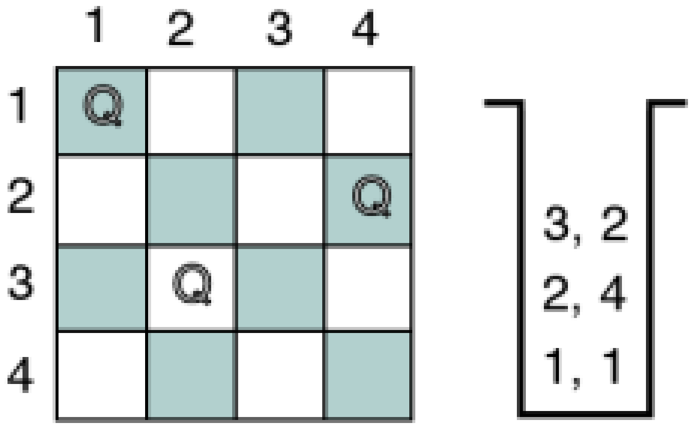
Step 1



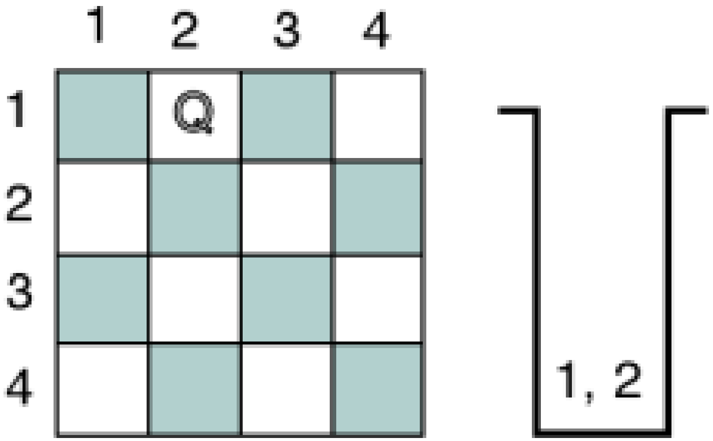
Step 2



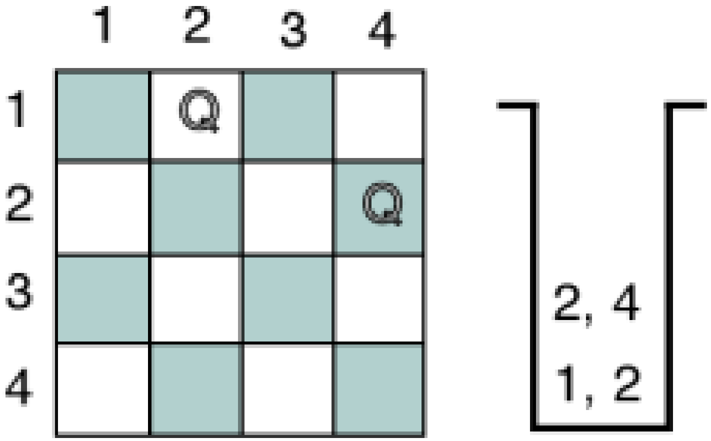
Step 3



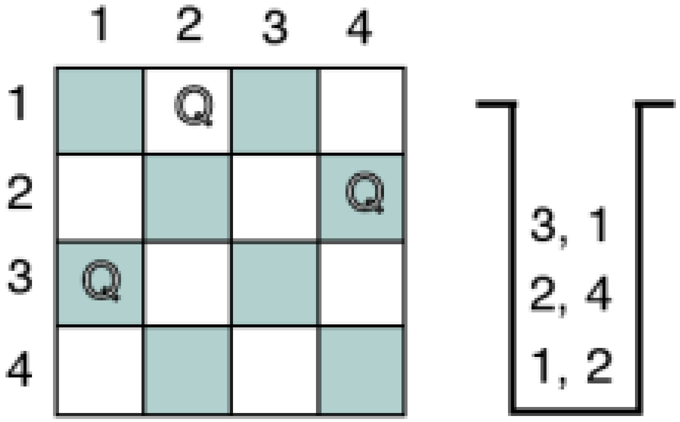
Step 4



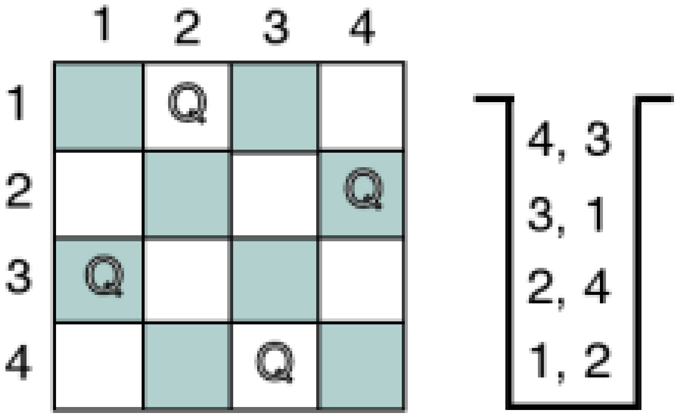
Step 5



Step 6



Step 7



Step 8

Contact Info

- trainers@finaldesk.com
- rishabh@finaldesk.com
- nilesh@finaldesk.com
- jignesh@finaldesk.com
- yash@finaldesk.com
- anand@finaldesk.com