Sunbeam Institute of Information Technology
Pune and Karad

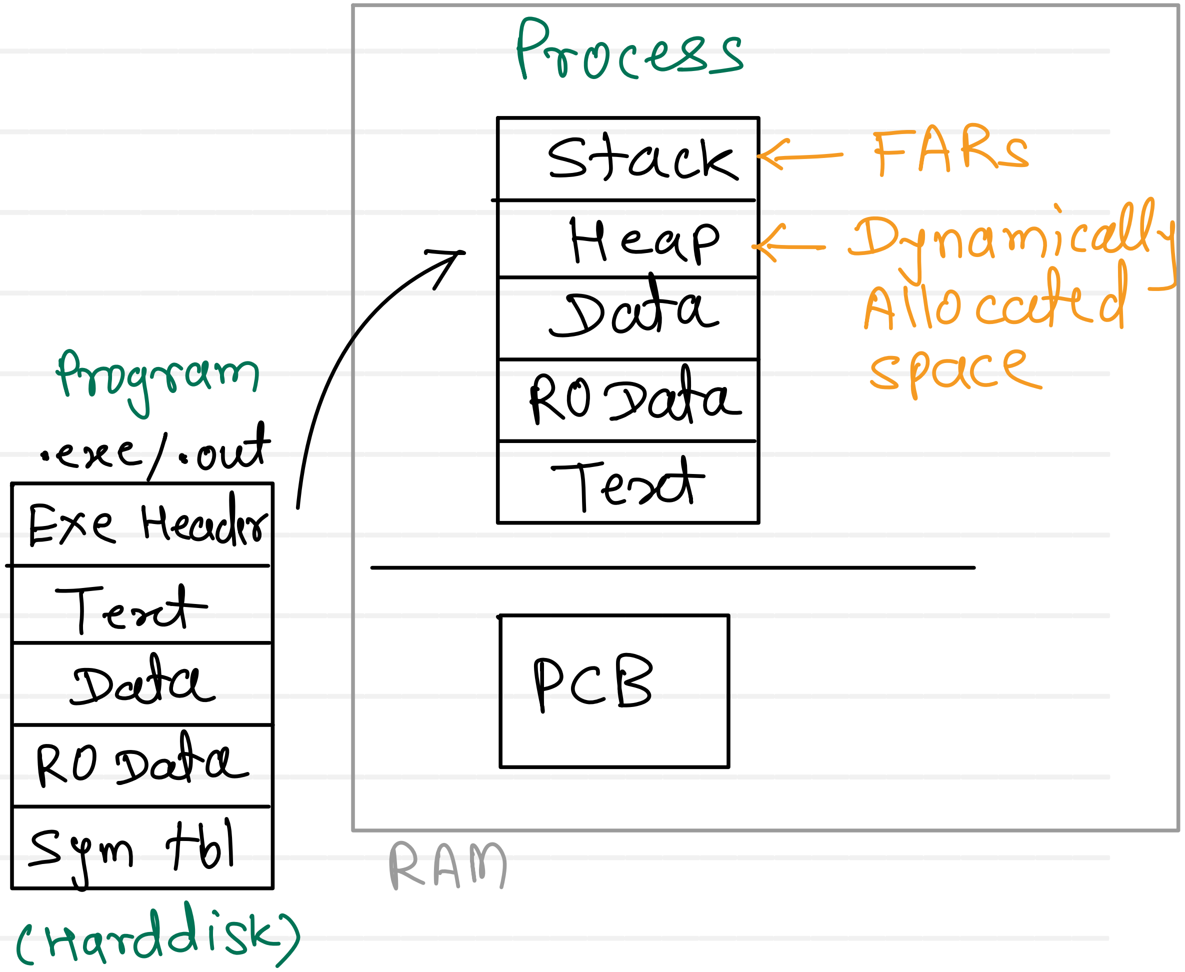
Data structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Stack Vs Heap

OS Interview



DS Interviews

Stack :
- utility DS
- LIFO behaviour
- operations - push/pop/peek
- can be implemented using array or linked list.

Heap :
- hierachical DS
- array implementation of Complete Binary Tree.
- max heap , min heap

Applications – Stack and Queue

Stack

- Parenthesis balancing [lexical analysis]
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Queue

- Jobs submitted to printer [spooler directory]
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Expression :

combination of operands & operators

Notations :

Infix : a + b → human

Prefix : + a b } computer

Postfix : a b + } (machine) → CPU ↓ ALU

Operator priorities :

( ) ↑ (Highest)

$ ← power

* / %

+ − (lowest)

① ② ③ ⑤ ⑥ ④ ⑦

Infix : a * b / c * d + e - f * h + l

Postfix : ab*/c*d+e−f*h+l

ab*c/*d+e−f*h+l

ab*c/d*+e−f*h+l

ab*c/d*+e−fh*+l

ab*c/d*e+−fh*+l

ab*c/d*e+fh*−+l

ab*c/d*e+fh*−l+

① ② ③ ⑤ ⑥ ④ ⑦

Infix : a * b / c * d + e - f * h + l

Prefix : *ab/c*d+e−*fh+l

/*abc*d+e−*fh+l

*/*abcd+e−*fh+l

+*/*abcde−*fh+l

−+*/*abcde*fh+l

+−+*/*abcde*fhl

Postfix Evaluation

- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

Postfix expression : 5 9 + 4 8 6 2 / - * - 1 7 3 - $ +

$l \longrightarrow r$

$-6 + 1 = -5$

$1 \$ 4 = 1$

$7 - 3 = 4$

$14 - 20 = -6$

$4 * 5 = 20$

$8 - 3 = 5$

$6 / 2 = 3$

$5 + 9 = 14$

Result = -5

~~4~~
~~3~~
~~7~~
~~1~~
~~-6~~
~~20~~
~~5~~
~~3~~
~~2~~
~~6~~
~~8~~
~~4~~
~~14~~
~~9~~
~~5~~

stack

~~1~~
-5

'0' = 48
'1' = 49
⋮

'1' - '0' = 1
'5' - '0' = 5

'5' → 5

Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
    - Push it on a stack
- If element is operator
    - Pop two elements (Operands) from stack, in such a way that
        - Op1 – first popped element
        - Op2 – second popped element
    - Perform current element (Operator) operation between Op1 and Op2
    - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

Prefix expression : + - + 5 9 * 4 - 8 / 6 2 $ 1 - 7 3

l ← r

−6 + 1 = −5

14 − 20 = −6

Result = −5

5 + 9 = 14

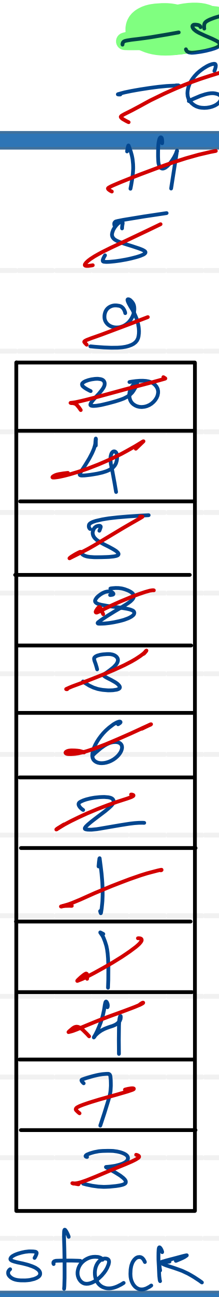
4 * 5 = 20

8 − 3 = 5

6 / 2 = 3

1 $ 4 = 1

7 − 3 = 4



Infix to Postfix Conversion

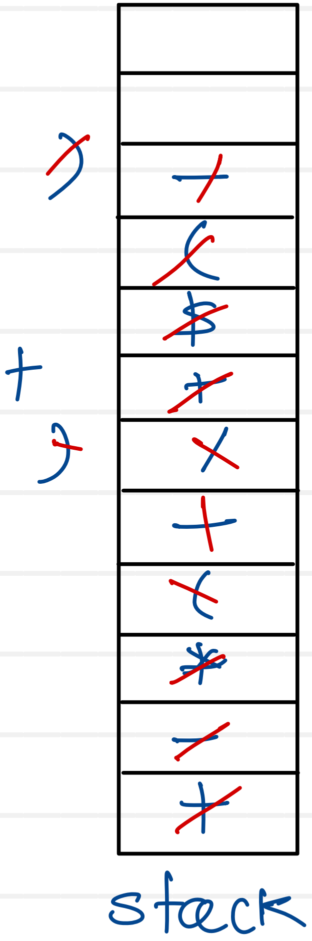
- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Postfix conversion

Infix expression : 5 + 9 - 4 * (8 - 6 / 2) + 1 $ (7 - 3)

l → r

Postfix expression : 59+4862/-*-173-$+



- for opening '(', push it on stack

- for closing ')' pop operators from stack and append into postfix expr untill opening is arived.

Infix to Prefix Conversion

- Process each element of infix expression from right to left ①
- If element is Operand
    - Append it to the prefix expression
- If element is Operator
    - ② If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
    - Repeat above step if required
    - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression ③
- e.g. a * b / c * d + e – f * h + i

Infix to Prefix conversion

Infix expression : 5 + 9 - 4 * (8 - 6 / 2) + 1 $ (7 - 3)

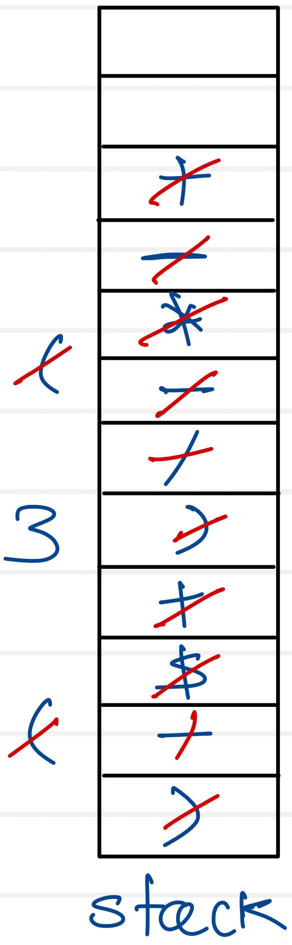
l ← r

Expression : 37-1$26/8-4*95+-+ ~~(~~

Prefix expression : +-+59*4-8/62$1-73

- for opening ')', push it on stack

- for closing '(' pop operators from stack and append into prefix expr untill closing is arrived.



Prefix to Postfix

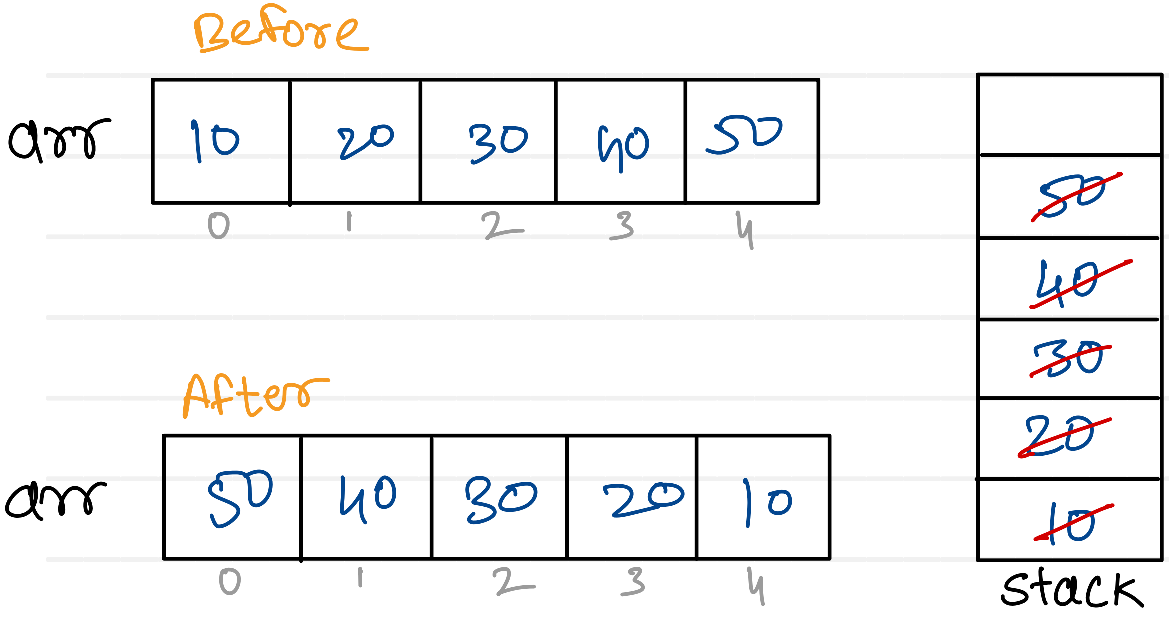
- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Reverse array, string or linked list using stack/queue

int arr[]= {10,20,30,40,50}



Before

arr	10	20	30	40	50
	0	1	2	3	4

After

arr	50	40	30	20	10
	0	1	2	3	4

```
void reverseArray(int arr[]) {
    stack<Integer> st = new stack<>();
    for( i=0; i< arr.length ; i++)
        st.push(arr[i]);
    for( i=0; i<arr.length; i++)
        arr[i] = st.pop();
}
```

Time complexity:

itr = n + n

$T \propto 2n$

$T(n) = O(n)$

Space complexity:

aux space = stack space

$S(n) = O(n)$

Valid Parantheses

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "(]"

Output: false

Example 4:

Input: s = "([])"

Output: true

```
boolean isValid(string s) {
  Stack<Character> st = new Stack<>();
  for( i=0; i< s.length(); i++) {
      char ele = s.charAt(i);
     if(ele == '(' || ele == '[' || ele == '{')
        st.push(ele);
    else if(ele == ')' && !st.isEmpty && st.peek() == '(')
        st.pop();
    else if(ele == ']' && !st.isEmpty && st.peek() == '[')
        st.pop();
    else if(ele == '}' && !st.isEmpty && st.peek() == '{')
        st.pop();
   else
      return false;
  }
  if( !st.isEmpty())
      return false;
  return true;
}
```

Parenthesis balancing using stack

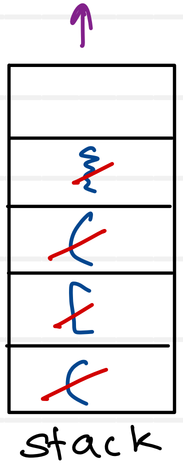
5+([9-4]*(8-{6/2}))

] == [

} == {

) == (

) == (

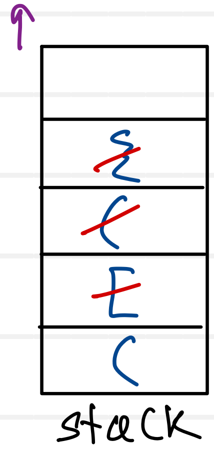


5+([9-4]*(8-{6/2}])

] == [

} == {

] != (



opening	(	[	{
	0	1	2

closing	)	]	}
	0	1	2

string
↓
indexOf()
↓
returns index of char
returns -1 if char not found

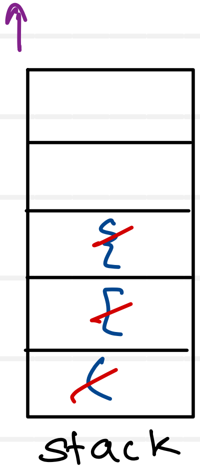
5+([9-4]*8-{6/2}))

] == [

} == {

) == (

) == ?

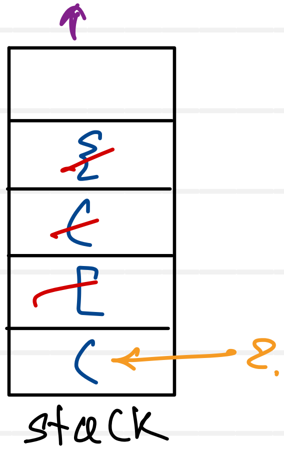


5+([9-4]*(8-{6/2})

] == [

} == {

) == (



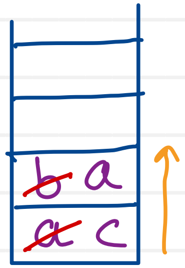
Remove all adjacent duplicates in string

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

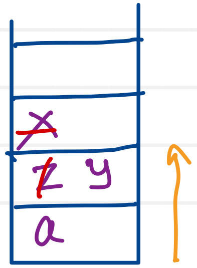
We repeatedly make duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

Example 1:
Input: s = "abbaca"
Output: "ca"



Example 2:
Input: s = "azxxzy"
Output: "ay"



```
String removeDuplicates(String s) {
    int n = s.length();
    char []st = new char[n];   } Auxillary
    int top = -1;              }   space

    for(int i=0; i<n; i++){
        char ch = s.chatAt(i);
        if(top > -1 && ch == st[top])
            top--;
        else {
            top++;
            st[top] = ch;
        }
    }
    return new String(st, 0, top+1);
}
```

array, starting index, length of string

$T(n) = O(n)$

$S(n) = O(n)$