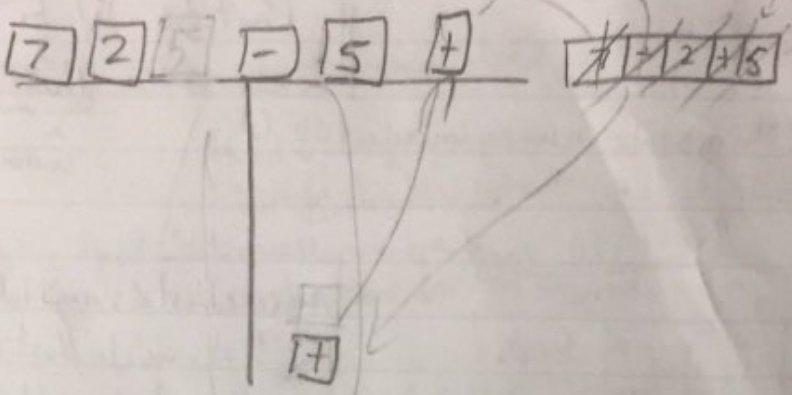
7-2+5

postfix

infix



7 2 5 + -

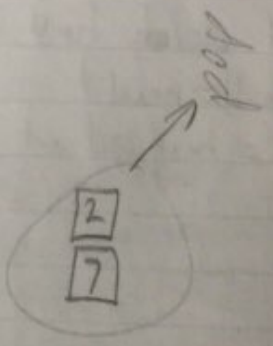
7 7 -

0

- 7 2 + 5 +

5 5 +

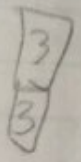
10

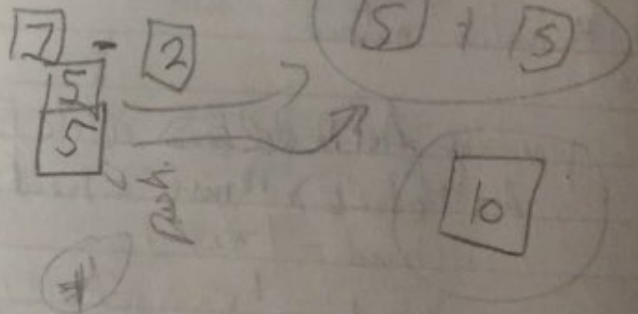


3 3 + 3 +

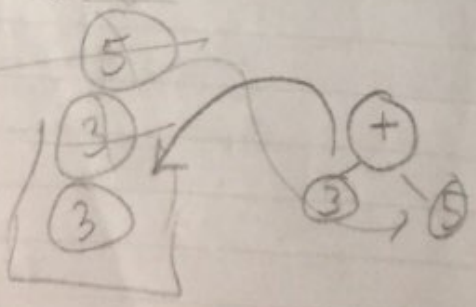
6 3 +

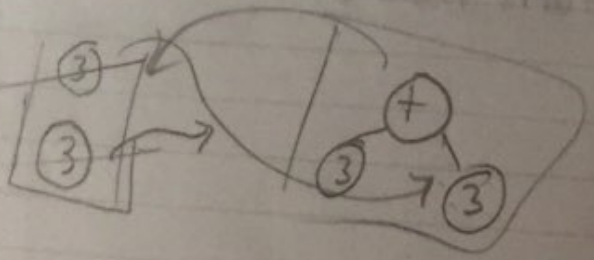
9





3 3 5 * -





Youtube channel: myschool code
Playlist Data Structure
Title: infix to Postfix

infix: A + (B * C)

Precedence of mul is higher so

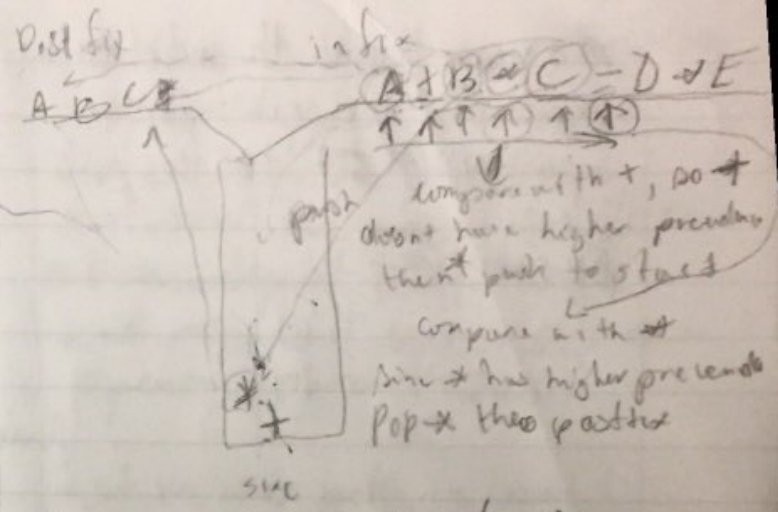
A + (BC *)

A (BC *) +

Postfix: ABC * +

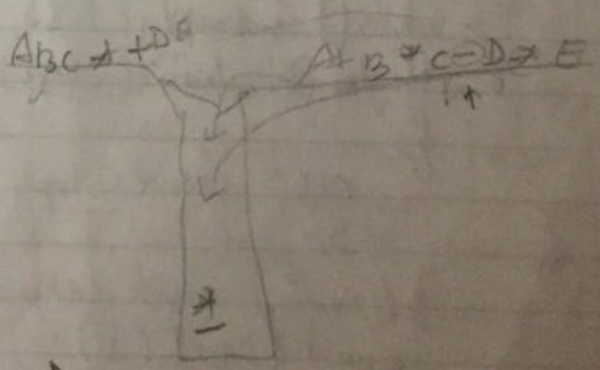
• In infix to postfix conversion the position of operands and operators may change but the order in which operand occur from left to right will not change.
✓ the order of operator may change

• In postfix form we will always have the operators in the same order in which they should be executed



• Any operator in the stack having higher precedence than the one we looking at can be popped and placed @ the postfix expression

• if an element in the stack has something on top of it then that something will always be of higher precedence
when pointing @ -



• There are 2 ways we can find the end of the right operand for an operator
a) if we get an operator of lesser precedence
b) if we reach the end of the expression (newline)

• once we reach the end of the expression we can pop the operators (* -) to the post fix
ABC*+DE*-

Pseudo code.

operators Queue

InfixToPostfix(exp)
{
String result (empty) ← Queue postfix.
create a stack of String S
for i = 0 to length(exp) - 1)
{ if exp[i] is operand → char, token
res += exp[i] , postfix.enqueue()
else if exp[i] is operator
{
while (! S.empty() && has there
has Higher Precedence (s.top(), exp[i]))
{
res ← res + S.top; ← postfix.enqueue(S.top())
S.pop() ← remove the last element.
} // end of while loop
S.push(exp[i]);
}
} // end of for loop; @ the end of it you need remove operators in the stack
while (!S.empty())
{ res ← res + S.top } → postfix.Enqueue(S.pop);
S.pop
return result } // end of infixtoPostfix ← this does not take care of Parentheses

Scanned by CamScanner

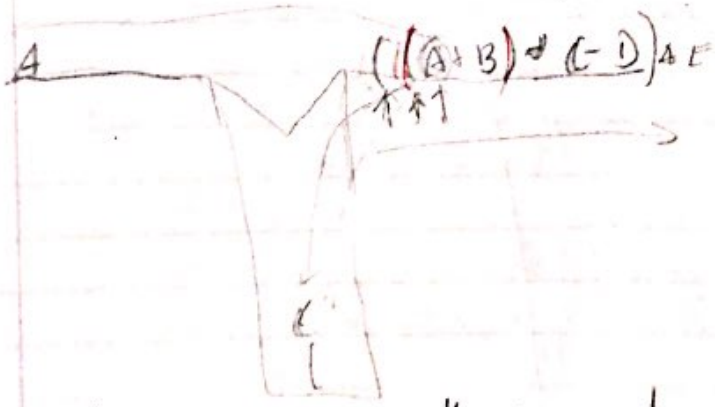
Now with parentheses

$((A+B)*(C-D))*E$

↳ its execution will be influenced by * only

- with parenthesis any part of the expression within parentheses should be treated as independent, complete expression in itself. and no element outside the parentheses will influence its execution.

- If parentheses are nested, inner parenthesis is sorted out or resolve first, another only outer parenthesis can be resolved.



- if opening parenthesis, push onto the stack.

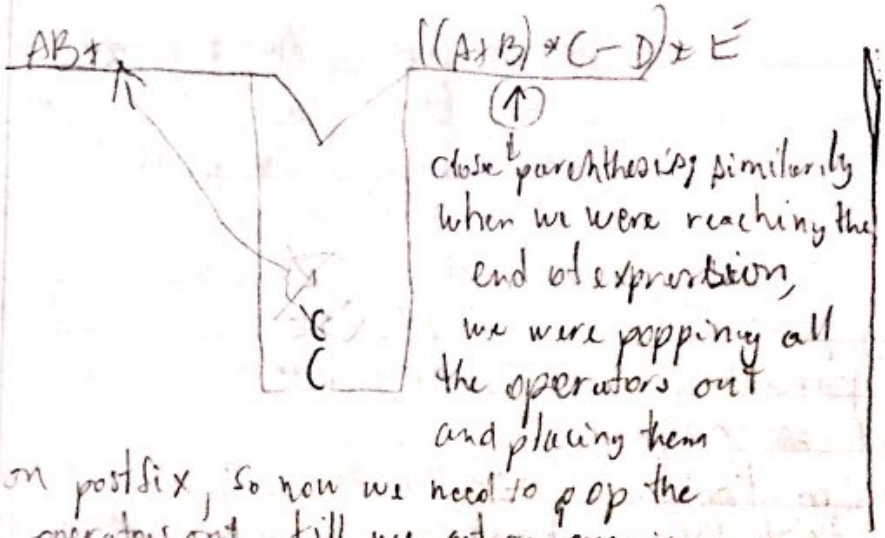
earlier when we looking @ operator we were looking at top of stack and popping as long we were getting operator of higher precedence

- earlier without parentheses we could go on popping and empty the stack but now we need to look at the top of the stack and pop till we get an opening parenthesis.

- If we are getting an opening parenthesis, then it's the boundary of the last opend parenthesis. and the operator does not influence any influence outside that

Rules

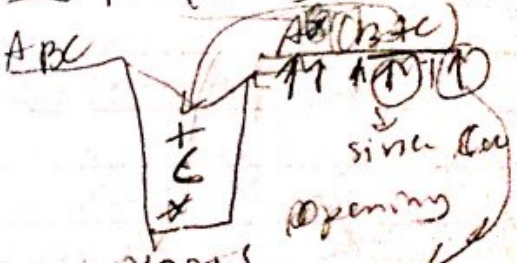
- If you're seeing an operator look @ the top of stack
  - in stack is higher precedence you can pop, look @ the next top
  - if higher precedence pop
  - stop when you see an open parentheses



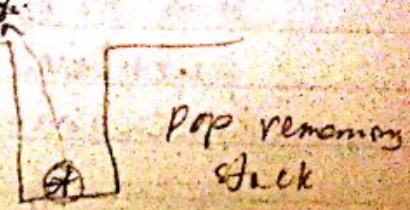
close parenthesis similarly when we were reaching the end of expression, we were popping all the operators out and placing them on postfix, so now we need to pop the operators out till we get an opening parenthesis, then pop the opening parenthesis without enqueuing it

- Rule for closing parenthesis
  - pop operators till you find an opening parenthesis then

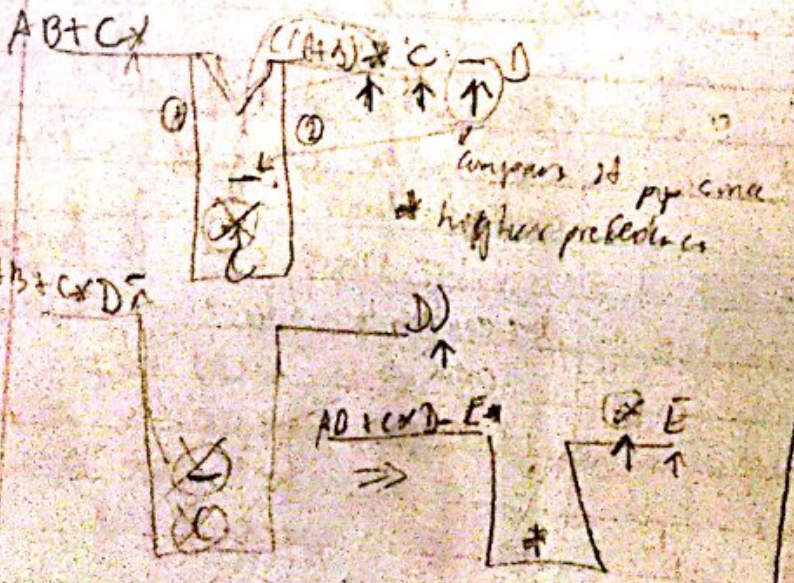
Ex A*(B+C)



Closing parenthesis pop stack engine to delete opening parenthesis.



- only part of the code that need to be change is the



Complete pseudocode

```
infixtoPostfix ( exp)
{
   string result        Queue <string> postfix
   create a stack of string S

   for i = 0 to length.exp. -1          while(!infix.isempty() /
   {
     if exp[i] is operand     //need to be able to   take a token and try
                              is                     to parse it to an integer
         rest ← res + exp[i]  ← postfix.enq          in a try catch block,
                                                     catch will return fal
     else if exp[i] is operator
     {
  →      while(! S.Empty() && ! IsOpeningParentheses(S.top())
            && hasHigherPrec(S.top(), exp[i])
            { res ← res + S.top
                                  → postfix.enque(S.pop())
                 S.pop
            } end of while loop.     ← pop w/ higher precedence
         S.push(exp[i])  ← then push to top of openin parenthesis
     } //end of else if

new →  [ else if exp[i] == '(' //else if is openingparentheses (exp[i]) ← more general
              S.push(exp[i])
     else if exp[i] == ')'  //else if is closing parentheses (exp[i])
     {   while( ! S.empty() && ! IsOpening parentheses (S.top())
         {   rest ← rest + S.top
                                  ← postfix.enque(S.pop())
              S.pop()
         }
         S.pop()      ← delete the open parentheses in stack.
     } //end of else state
   } // end of for loop
   while ( ! S empty()
   {  res ← res + S.top
        S.pop
   }
   return result.
}
```

complete pseudocode

```
infixToPostfix ( exp)
{
  string result          Queue < string> postfix
  create a stack of string S

  for i = 0 to length.exp. - 1      while.(infix.isempty() /
  {                                                     // need to be able to   take in token and try
    if exp[i]  is operand                               is                    to parse it as an integer
        rest ← res + exp[i]   ← postfix.en...                                  in a try catch block
    else if exp[i] is operator                                                 catch will return false
    {
  →   while(! S.empty() && ! IsOpeningParenthesis(S.top())
      && hasHigherPrec(S.top(), exp[i])
      {  res ← res + S.top
                                  ] → postfix.enqueue(S.pop())
          S.pop
      } end of while loop.       ← pop w/ higher precedence
      S.push (exp[i]) ← then push to top of openin parenthesis
    } // end of else if
new →  [ else if exp[i] == '(' // else if IsOpeningParenthesis (exp[i]) ← more general
           S.push(exp[i])
    else if exp[i] == ')'  // else if IsClosingParenthesis (exp[i])
    {  while( ! S.empty() && ! IsOpeningParenthesis (S.top())
       {  rest ← rest + S.top
                                  ] ← postfix.enqueue(S.pop())
            S.pop()
       }
       S.pop()     ← delete the opening parenthesis in stack.
    } // end of else state
  } // end of for loop
  while ( ! S.empty()
  {  res ← res + S.top
       S.pop
  }
  return result.
}
```

Title: check for balanced parenthesis using stack.
youtube channel: my code school
Published date: Oct 28, 2013

Expression	Balanced?
(A+B)	yes
{(A+B)+(C+D)}	yes
{(x+y)*z)	No
[203]+(A)]	No
{A+Z)	No
)C	No
[( )]	yes
[( ])	No

So count the # of times u see closing & open with work

- Every opening parenthesis must find a closing counterpart to its right
- every closing parenthesis must find an opening counterpart to its left.
- A parenthesis can close only when all the parenthesis open after it are closed
  Last opened first closed

- As we scan the expression from left to right, any closure should be for the previous unclosed parenthesis
- any closure should be for the last unclosed

[( ) ( )]

- scan the expression from left to right
- keep track of all the unclosed parenthesis as we scan in any stage
  - when you come across an opening symbol add it to a list. (push to stack)
  - if closing symbol
    - should be the closure of the last element in the list.
    - if inconsistent like we

last opening symbol of the list is not of the same type as the closing symbol. or if there is no last opening symbol at all (list is empty)
  stop the whole process
- else
  remove the last opening symbol in the list since matched
  & continue this whole process (loop)

[( ) ( )]

list: empty

list: [(← it's closing

Do look @ the last element in the list of same type of matching

remove (it) & move on to the next

now list become

List: [

[( ) ( )]

list: [(← remove

list: [← remove

list: empty

• If every thing is all balance we will always end up with an empty list.

Pseudo Code

check balanced parenthesis (string Exp)

{ n = length(exp)

create stack S

for i = 0 to n-1

{ if exp[i] is '(' or '{' or '['

push (exp[i])

else if exp[i] is ')' or '}' or ']'

{ if (s.isEmpty) || (s.top does not pair with exp[i])

{ return false ← not balanced

} // end of else if

else

pop()

} // end of for

return S is empty ? true : false

check for ( )

(A not valid.

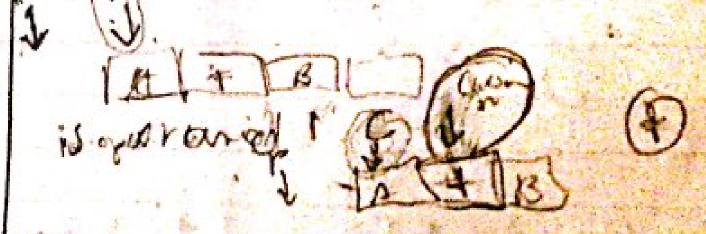
(+ ← not valid ✓ taken care by is operator

((← valid

if previous is not)(would be invalid

)# empty not valid

↑ operator valid list



now then switch through the state of while v. write

Current = + next

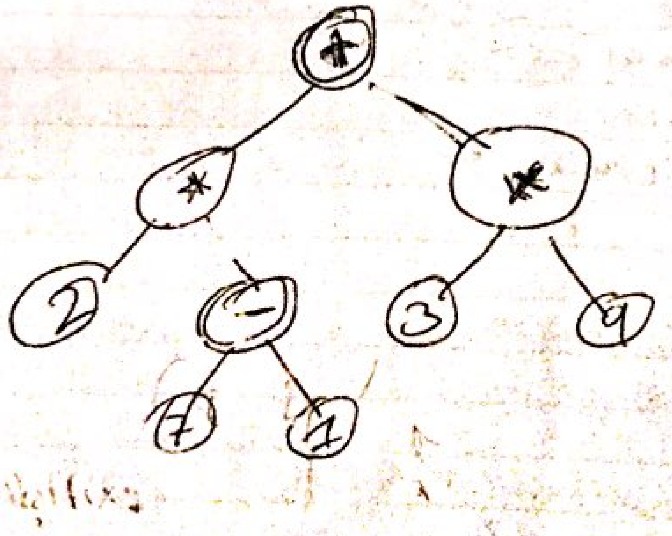
but after

Current. current = B

order of operation

1) Parentheses ( ) { } [ ] → Right associative
2) exponents (right to left) 2^3^2
3) multiplicate & division (left to right) ↑
4) Addition or sub (left to right)
Ex: 4-2+1

((2×(7-1)) + (3×9))



Infix:

Preorder: + * 2 - 7 1 * 3 9
inorder: 2 * 7 - 1 + 3 * 9
Post order: 2 7 1 - * 3 9 * +
Postfix: 2 7 1 - * 3 9 * +