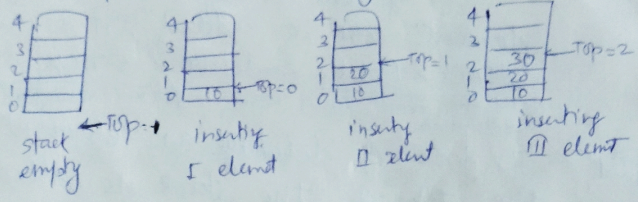
STACKS

Ex- stack of plates in marriage party
Biscuit packet whose cover is torn on one side

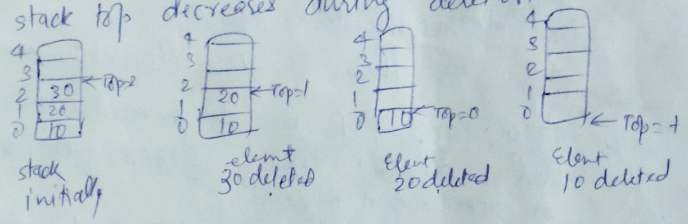
Array Representation & Implementation of stack

Array Representation –

→ stack top increases during insertion



→ stack top decreases during deletion



stack implementation –

stack can be implemented in 2 way < static
Dynamic (linked list representation)

static impⁿ – size of stack is declared during prgm design
- It uses array to create stacks
- Simple tech but not flexible & not too efficient w.r.t. memory

(disadv: size of array is larger → memory wastage
smaller → new element can't be added)

Dynamic impⁿ – It uses pointers to create stacks

Operations on stack → PUSH, POP

PUSH – The process of adding a new element to the top of the stack. is called PUSH
After every PUSH operation, the top is incremented by one.
If the array is full & no new element can be added, it is called STACK-FULL condition / STACK OVERFLOW
(At this condn Top of stack is present at highest location of the stack)

POP – The process of deleting an element from the top of the stack is called POP.
After every POP operation, the stack is decremented by one.
If there is no element on the stack and POP is performed then this will result into STACK UNDERFLOW condn
or STACK-EMPTY condn
(At this condn TOS (top) is present at the bottom of stack)

STACK Terminologies used in algorithms –

1. TOP = Top of stack (TOS)
   It is used to check stack overflow/underflow
   Initially it stores -1
2. stack = it is an array of size MAXSIZE

Algo's for PUSH & POP

① PUSH (Algo for inserting an item into the stack)

PUSH (stack[MAXSIZE], item)

① // check for stack overflow
If TOP = MaxSize - 1
then print : overflow
exit

② // Increase TOP by 1
set TOP ← TOP + 1

③ // Insert Item in new TOP position
set STACK[TOP] := ITEM

④ exit

② POP (Algo for deleting an element from stack)

POP (stack[MaxSize], Item)

① // check for stack underflow
If TOP < 0
then print : stack underflow
exit
else
set item = stack[TOP] // Remove the TOP element

② // ~~Decrement the~~ stack TOP
set TOP = TOP - 1

③ Return the deleted item from the stack

④ Exit

Application of Stack

1. Conversion of Infix to Prefix & Postfix expressions
2. Evaluation of postfix expressions.

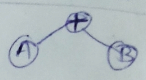
Notations for a mathematical expression ← Infix (A+B)
Prefix (+AB)
Postfix (AB+)

Universally accepted notation for calculation of an expression is postfix. It is most suitable for a computer.
Any expression entered into comp is first converted into postfix notⁿ, stored in stack & then calculated

add (A, B) ⇒ Prefix Notation
Prefix & Postfix notation are not really as awakward to use as they might look at first.
(Human beings work in infix notⁿ which is rath complex because one has to remember some rules = BODMAS, Accedance, Associativity)

~~Postfix~~

Infix = A + B
(A = operand, + = operator, B = operand)



Prefix = Polish notⁿ
Postfix = Suffix notⁿ / reverse polish notⁿ

Operator Precedence

Exponential operator (^) → Highest Precedence
Multiplication / Division (*, /) → Next "
Addition / Substraction (+, −) → least "

BODMAS = Bracket
Of
Division
Multiplication
Addition
Substraction

Note - ^ has the highest priority

Converting Infix to Prefix —

① $(A * B) + C$

$*AB + C$

$+ * ABC$

② $A/(B \wedge C) + D$

$A/ \wedge BC + D$

$/A \wedge BC + D$

$+/A \wedge BCD$

③ $(A - B/C) * (D * E - F)$

$(A - /BC) * (* DEF)$

$(-A/BC) * (-*DEF)$

~~$* A/BC * D - EF$~~

$* -A/BC - * DEF$

④ $(A * B + (C/D)) - F$

$((*AB) + (/CD)) - F$

$(+ * AB/CD) - F$

$- + * AB/CDF$

⑤ $A/B \wedge C - D$

$A/ \wedge BC - D$

$/A \wedge BC - D$

$-/A \wedge BCD$

⑥ $A + B * C$

$A + (* BC)$

$+ A * BC$

(Algo)

Let the expression is ⇒ a+b*c

I^st reverse a+b*c ⇒ c*b+a

conversion in postfix ⇒ cb*a+

reverse this result ⇒ ~~+a*b c~~ +a*bc

i/p Infix Exp^n → O/P Prefix Exp^n

(see example which is later on then see this algo)

Prefix(R, S)

1. Reverse the input string R and get reversed string Q
2. Scan the ~~next~~ elements ~~of~~ the i/p exp^n and repeat the steps (3) to (6) for each element, until stack is empty.
3. If it is operand, add it to the O/P string
4. If it is closing paranthesis ")", push it on stack.
5. If it is an operator, then
   - (i) If stack is empty, push operator on stack.
   - (ii) If top of stack is closing ")" paranthesis, push operator on stack
   - (iii) If it has same/higher priority than the top of stack, push optr on stack
   - (iv) else pop the operator from stack & add it to O/P string repeat 5.
6. If it is an opening paranthesis "(", pop operator from stack & add them to O/P str until a closing paranthesis ")" is encountered. Pop & discard the ")" closing paranthesis
7. If there is more i/p, go to step 2
8. If there is no more i/p, unstack the remaining operators and add them.

② call POSTFIX (Q, P)

③ Reverse the O/P string P and get reversed string S as Prefix

closing paranthesis

opening paranthesis

Converting Infix to Postfix -

(1) A + B * C
A + (B * C)
A + (BC*)
A (BC*) +
ABC*+

(2) A + [(B+C) + (D+E) * F] / G
A + [(B+C) + (D+E) F *] G /
A + (BC+) + (DE+F*) G /
A + (BC+ DE+ F*+) G /
A BC+ DE + F*+ G/ +

(3) A + B - C
(AB+) - C
AB+C -

(4) A * B + C/D
AB* + C/D
(AB*) + (CD/)
AB* CD/ +

(5) A * B+C
AB* +C
AB*C+

(6) A + B/C - D
A + (BC/) - D
(ABC/+) - D
ABC/+D -

⑦ (A+B)/(C-D)

(A+B)(C-D)/

AB+ CD-/

⑧ (A+B) * C/D

~~(A+B) (C/D)* / D~~

~~(A+B) (CD/)*~~

~~AB+ CD/*~~

~~(A+B) C * D/~~

~~A+B~~

~~AB + C * D~~

(A+B) * (CD/)

(A+B) CD/ *

AB+ CD/* ✓

⑨ (A+B) * C/D + E ^ F/G

~~(A+B) * C/D + (EF^) /G~~

~~(A+B) C* /D + (EF^) /G~~

~~((A+B) C * D/) + (EF^ G/)~~

~~AB+ C * D/ EF^G/+~~

(A+B)*(CD/)+(EF^)/G

(A+B) * (CD/)+(EF^G/)

(A+B)(CD/*)+(EF^G/)

A B+ CD/* EF^G/+ ✓

⑩ A+ [(B+C)+ (D+E)*F]/G

A + ((BC+) + (DE+) * F) G /

A + ((BC+) + (DE+ F*)) G /

A+ ((BC+) (DE+ F*)+) G /

A BC+ DE+ F* + G/+

Prefix - + A / + +BC * +DEF G

⑪

~~Prefix~~

Prefix

⑫ A -

A -

A - (

A B

Binary Expression

②

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

D / (E F ^)

(D E F ^ /) * G

D E F ^ / G *

(B C *) - (D E F ^ / G *)

(B C * D E F ^ / G * -) * H

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

A B C * D E F ^ / G * - H * +

+ A * - * B C * / D ^ E F G H

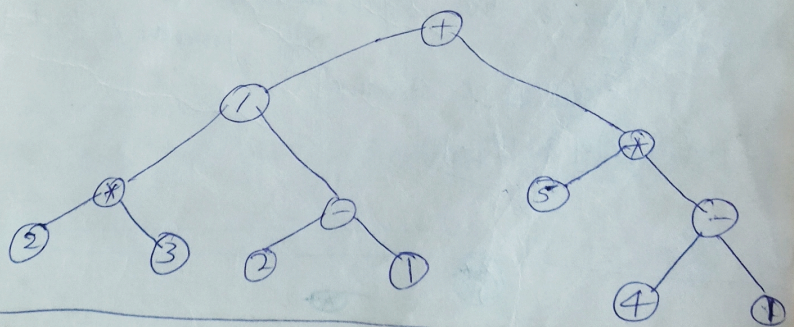
(12) A - B / (C * D ^ E)

A - B / (C D E ^ *)

A - (B C D E ^ * /)

A B C D E ^ * / - [Prefix : - / * ^ D E C B A]

Binary Expression Tree for expression —
2 * 3 / (2 - 1) + 5 * (4 - 1)



Algo (⊗ means any of the operator)

POSTFIX (Q, P)

Q: Arithmatic expression in infix → P: equivalent postfix expression

① Push "(" onto STACK, and add ")" to the end of Q

② Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.

③ If an operand is encountered, add it to P

④ If a ~~left parenthesis~~ "(" is encountered, push it onto STACK

⑤ If an operator is encountered then:

(a) Add ⊗ to STACK
[End of If structure]

(b) Repeatedly pop from STACK and add P each operator (on the top of STACK) which has the same/higher precedence than ⊗.

⑥ If a ~~right parenthesis~~ ")" is encountered, then:

(a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a ~~left parenthesis~~ "(" is encountered.

(b) Remove the ~~left parenthesis~~ "(". [Do not add the left parenthesis to P.]

[End of If structure]

[End of step 2 loop.]

⑦ Exit.

(Ex.)

Note Convert (A+B*C) into prefix expression

1st reverse the above expression i.e. (C*B+A)

1st convert it into postfix expression then reverse the expression which will be the result as prefix expression.

(Postfix of reversed infix) ~~A B C * +~~ C B * A +

(Prefix - of original - Reverse the above) ~~+ A * B C~~ + A * B C

writing systematically

Reverse (A+B*C) ⟹ (C*B+A)

converting (C*B+A) to postfix

Symbol scanned	stack	postfix expression
(	(	
C	(	C
*	(*	C
B	(*	C B
+	(+	C B *
A	(+	C B * A
)	—	C B * A +

This postfix expression will be C B * A +

Now reverse this expression to get result

+ A * B C

This will be prefix expression of (A+B*C)

Evaluation of postfix expression using stack

Algorithm :-

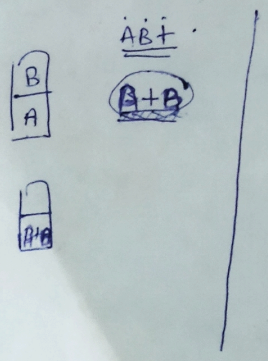
P – postfix expression

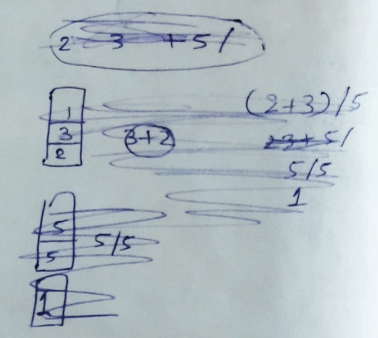
BA+

Evaluation (P)

1. Add a right parenthesis ")" at the end of P [this acts as sentinel]
2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel ")" is encountered
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then :-
   a. Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   b. Evaluate B ⊗ A
   c. Place the result of b back on STACK.
   [End of If structure]

   [End of step 2 loop.]
5. Set VALUE equal to the top element on STACK
6. Exit



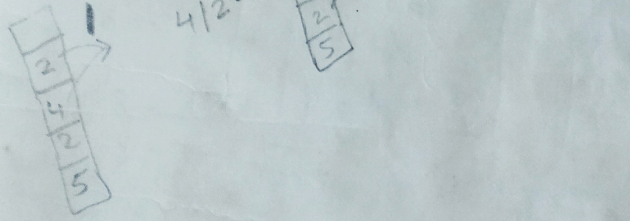


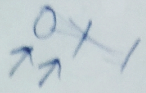
Linked Representation of stack

Page 507 of Balagu

Infix 5-2*4/2 = 1

Postfix 5 2 4 2 / * -





Recursive

Check minor Exam Pat... 2017 (Autumn)

Ex.

Point

Fibonacci series (Recursive dyⁿ)

$$fib(n) = \begin{cases} 0 \\ 1 \\ fib(n-1) + fib(n-2) \end{cases}$$

Position

if $n=0/1$

if $n=2$

if $n>2$

0 1 1 2 3 5

$n=3$ fib(2)+fib(1)
1 + 0
1

$n=4$ fib(3)+fib(2)
1 + 1
2

$n=5$ fib(4)+fib(3)
2 + 1
3

$n=6$ fib(5)+fib(4)
3 + 2
5

Types of Rec

int

Direct

Recursion

Principle of Recursion:
- It is defined as "defining anything in terms of itself."
- In recursion function calls itself

Recursive Definition & Process:
"Recursion is an alternative to iteration in making a function execute repeatedly"

Ex. Factorial fn.

$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$

Recursive definition

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ n * f(n-1) & \text{else} \end{cases}$$

As a C pgm

```
int fact (int n)
{
  if (n==0)
  return 1;
  else
  return n * fact(n-1);
}
```

Types of Recursion < Direct / Indirect

```
int abc()
{
  abc();
}
```

Direct

```
int abc()
{
  xyz();
}
int xyz()
{
  abc();
}
```

Indirect

Tower of Hanoi Problem -

It is a game problem.

→ There are 3 different sized disks having hole in centre, so that they can be stacked on the pegs (x, y, z).

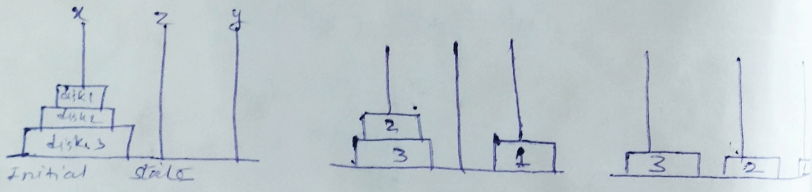
→ In the begining, the discs are stacked on the peg x i.e. the largest sized disk on the bottom & the smallest sized disk on top.

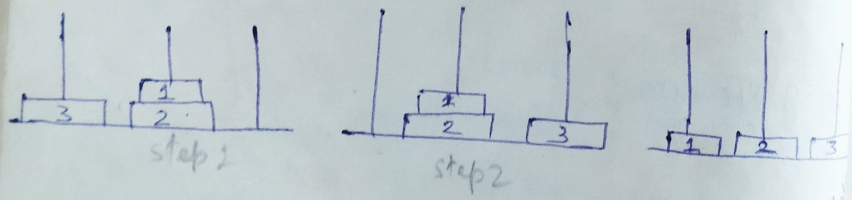
→ Rule of the game :—

① Transferring the disks from the source peg (x) to the destination peg (y) such that at any point of time no larger disk is placed on smaller one.

② Only one disk may be moved at a time

③ Each disk must be stacked on any one of the pegs





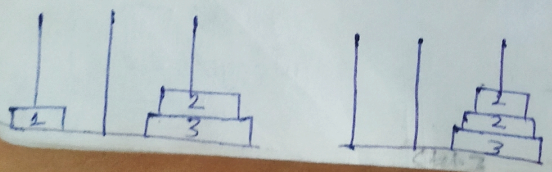


Fig: Pictorial Representation of 7 moves

It can be generalized for n no. of disks as following. ~~The~~ recursive definition ~~given below~~ suggests —

① Move the top (n-1) disks from the x peg to the z peg
② Move the nth disk to y peg.
③ Move the (n-1) disk stacked on the z peg to the y peg

Simulating Recursion :- It can be understood by above problem. that how the Tower of Hanoi Problem is simulated using recursion.

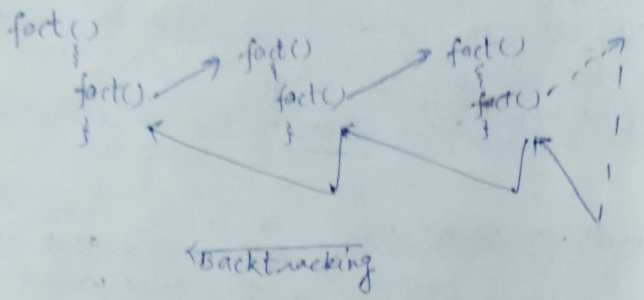
Recursive Algorithms

① factorial —

```
Procedure fact
    Read n
    if n=0   then return 1
    else     return n* fact (n-1)
end
```

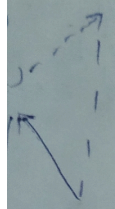
② fibonacci series —

Backtracking



Disadvantages of Recursion – (Removal of Recursion

1. It consumes more storage space
2. The comp may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed & execution time.
4. It does not offer very high advantages over non-recursive functions
5. If proper precautions are not taken, recursion may result in non-terminating iterations
6. Recursion is not suggested when the problem can be solved through iteration.



d of Recursion

if the recursive

of speed &

ges over non-

ken, recursion
terations

en the problem

Tail Recursion.

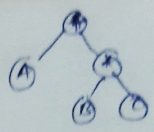
Q.6

A+(B*C) Infix

A BC*+ Postfix

Make the tree

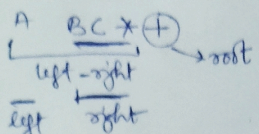
Result :



Soln.

A + B * C — Infix

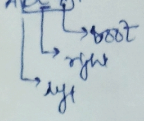
A BC * + — Postfix

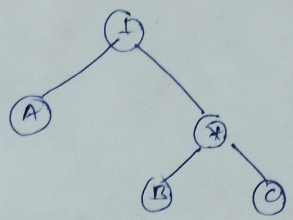


1st step

A+(B*C)

A BC * +





2nd step

Array

Insertion

R=-1

0 1

F R

R=1

16 20

0 1

F R

the Va

front i

the fro