infix a + b

post fix a b +

infix a + (b * c) = (a) + (b c *)

infix

post fix a b c * + .

a + b - c * c / d * f - g

algorithm :

Algorithm → infix to postfix.

a) Create a stack

b) For each character 't' in i/p

{ if (t is an operand)

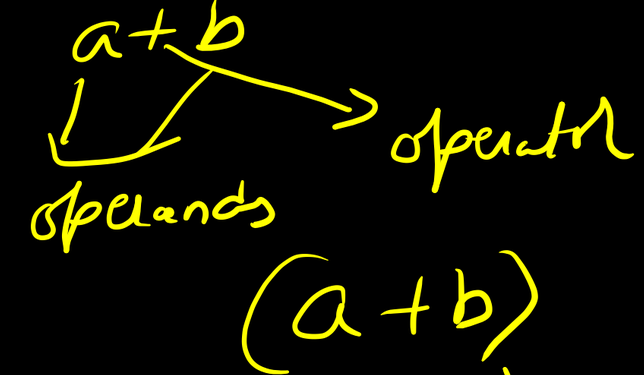
o/p t;

else if (t is a right parenthesis)

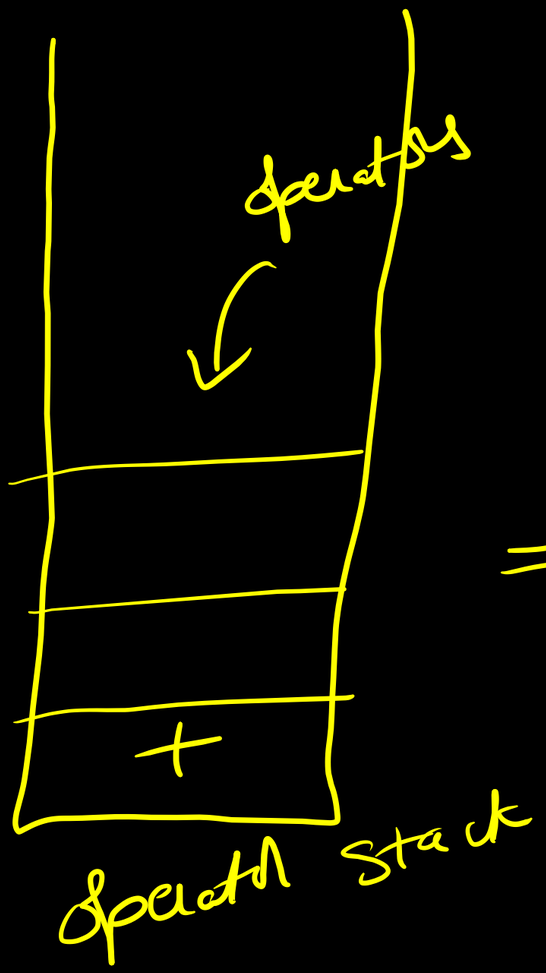
{ pop and o/p tokens until a

left parenthesis is popped (but not o/p)

}



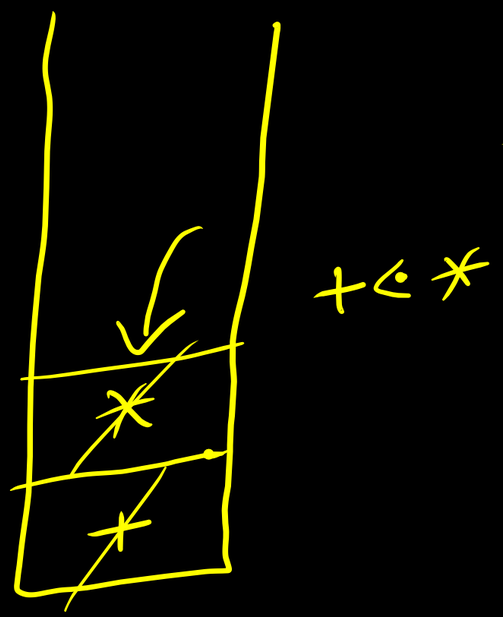
```
else { // t is an operator or left parenthesis
    pop and o/p tokens until one of the
    lower priority than 't' is encountered
    or a left parenthesis is encountered
    or stack is empty
    push t
}
```



$a + b * c.$

$\Rightarrow$ always higher priority operator should sit on top of low priority oper.

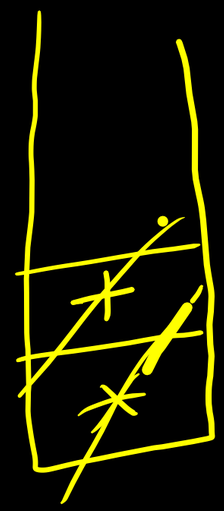
a + b * c



a b c * +

+ < *

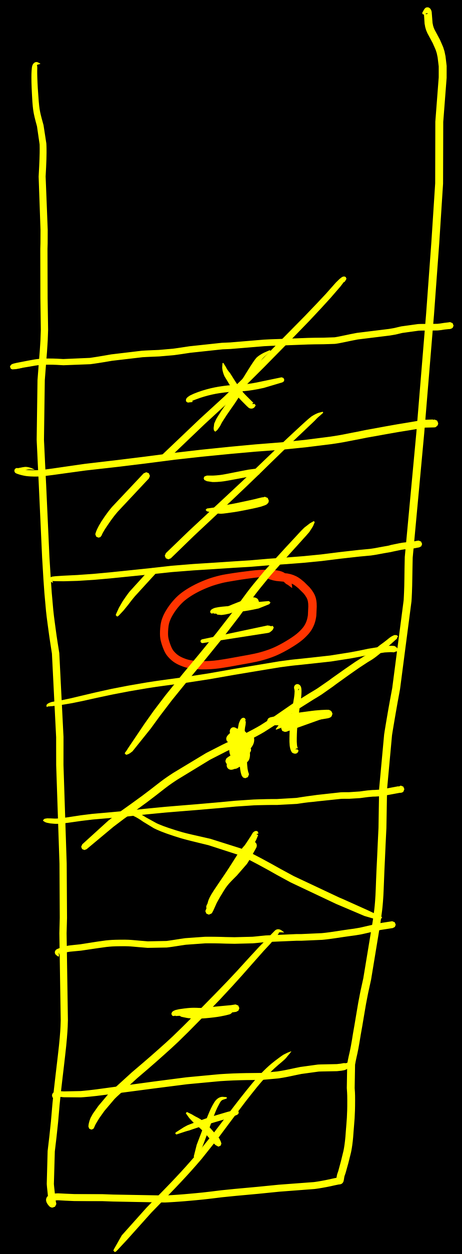
a * b + c



a b * c +

(d * a - b / c + e = f = g * h) ✓ higher priority can sit on lower priority.

d a * b c / - e + f g h * = =



= < =

- > +

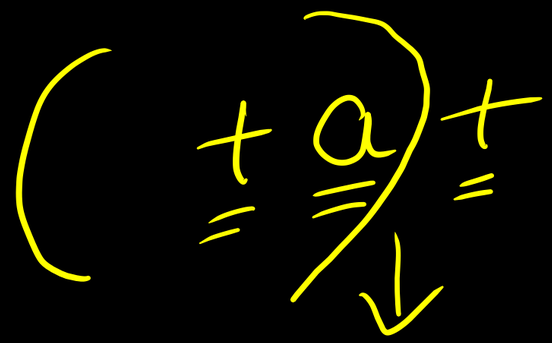
= =

\+ +

\- +

same precedence

associativity



+ is left associative

Infix → post fix → evaluate post fix.

Evaluate Infix → multiple scans are required

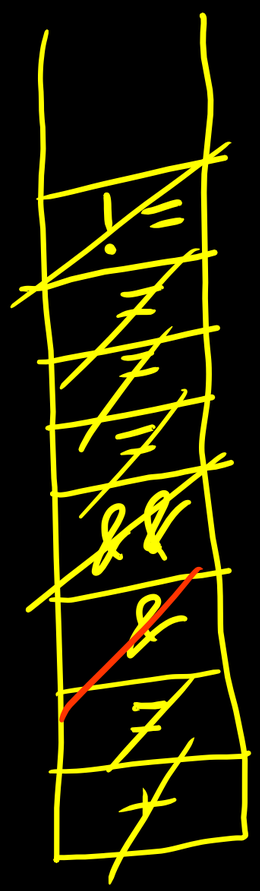
a + b - c * d / e * f

Infix - postfix → 1 scan & 1 pass

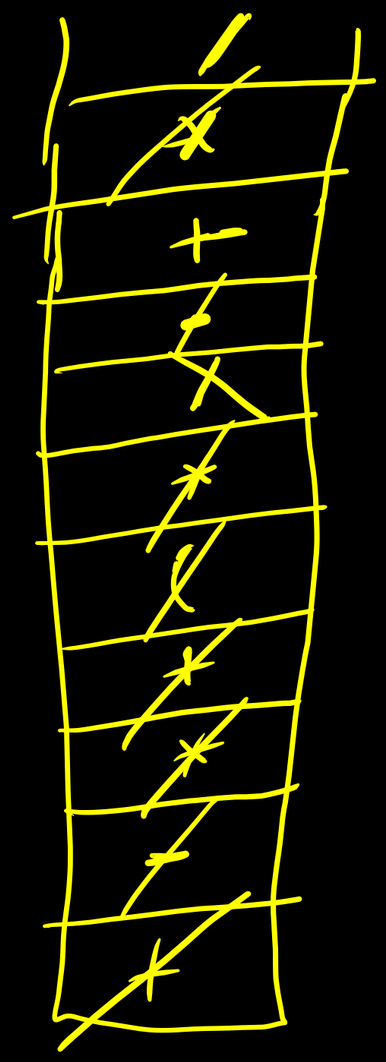
postfix evaluation → 1 pass.

a + b = c & d && e = f = g = h != i

a b + c d & e && f g h i != = = = =



2 + 3 - 1 * 2 + (8 * 2 / 16) - 3 + 4 * 3 / 6



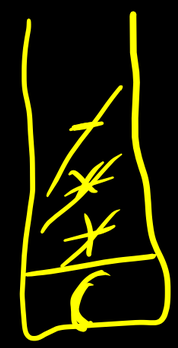
2 3 + 1 * - 8 2 * 16 / + 3 -

4 3 * 6 / +

op < (

(< op

(< /

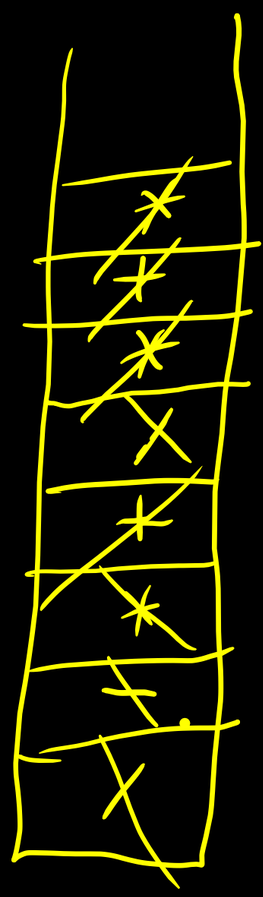


(- * + .

Postfix will not contain parenthesis

10/5 - 3 * 7 + 8 / 1 * 9 + 3 - 4 * 7

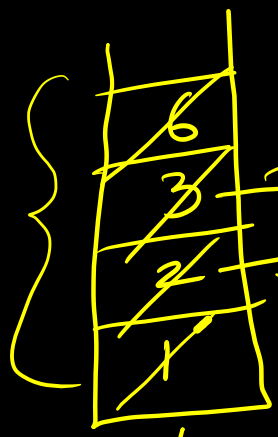
10 5 / 3 7 * - 8 1 / 9 * + 3 + 4 7 * =



Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *` (*type*) `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Infix → postfix ⟶ Evaluate a Postfix.

1 2 3 (*) (+)

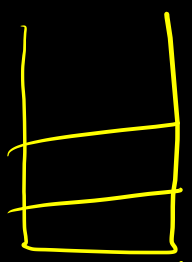


operand stack

2 * 3 = 6.

1 + 6 = (7) ✓

Infix - postfix



operand stack.

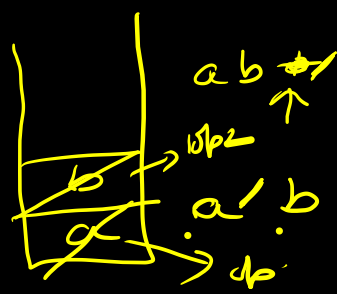
Algorithm for postfix evaluation:

1) Scan the postfix from left to right
2) Initialize an empty stack
3) Repeat steps 4 and 5 till all the characters are scanned
4) If the scanned character is an operand, push it on to the stack
5)

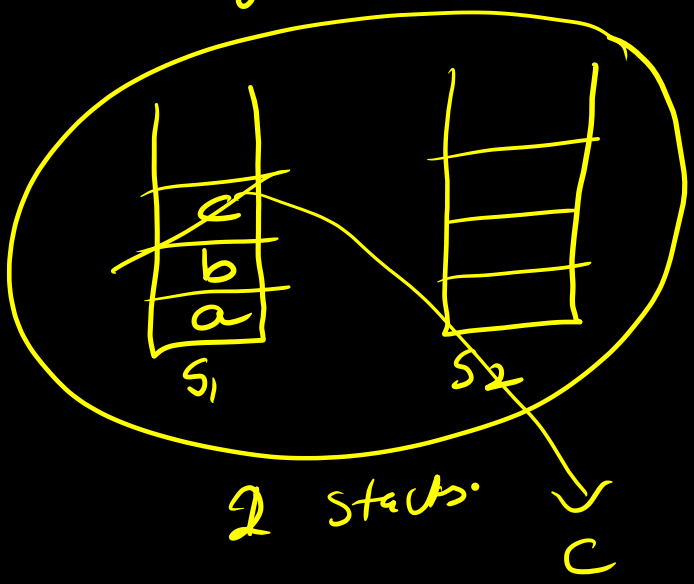
5) If the scanned character is an operand and if the operand is unary, then pop an element from the stack. If the operator is binary, then pop two elements from the stack. After popping the elements, apply the operator to those popped elements. Push the results on to the stack

Simp algo:

```
if (operand)
    push
else if (operator)
{
    op2 = p.p()
    op1 = pop()
    result = op1 operator op2
    push(r)
}
```

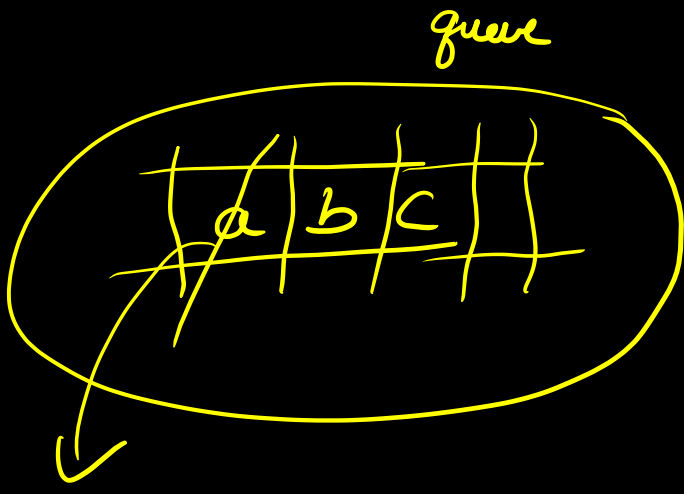


implementing queue using two stacks.



2 stacks.

c



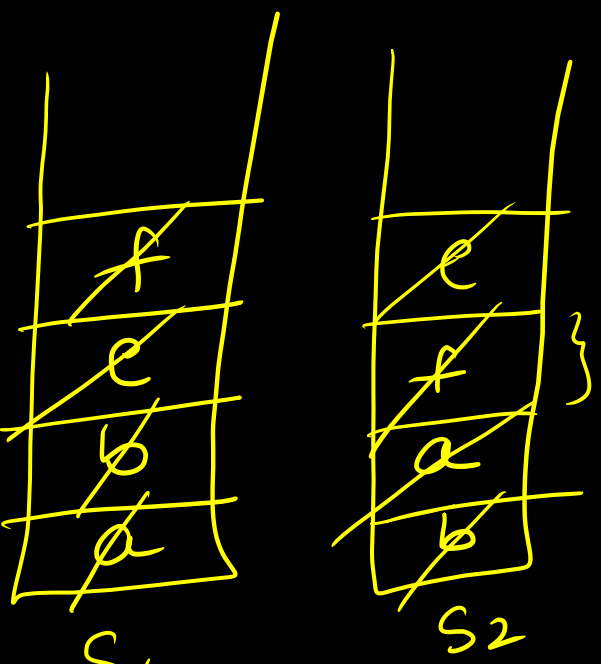
a.

S_1

S_2

a b c d e f

a b c d e f



S_1

=

S_2

abef

push → S_1

pop → S_1

push → S_2

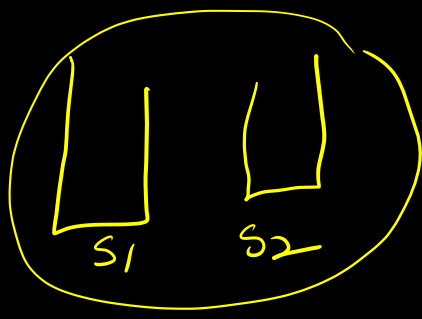
pop → S_2

$(n * 4) = O(n)$

abe

Alg:

```
void imax (Q, x)
{ push(S1, x)
}
```



Q.

```
void delete (Q, x)
{
    if (! stack_empty (S2))
        x = pop S2
    if (stack_empty (S2)) then
        if (stack_empty (S1)) then
            printf (Q is empty
        }
```

delete

else // S_1 is not empty S_2 is empty

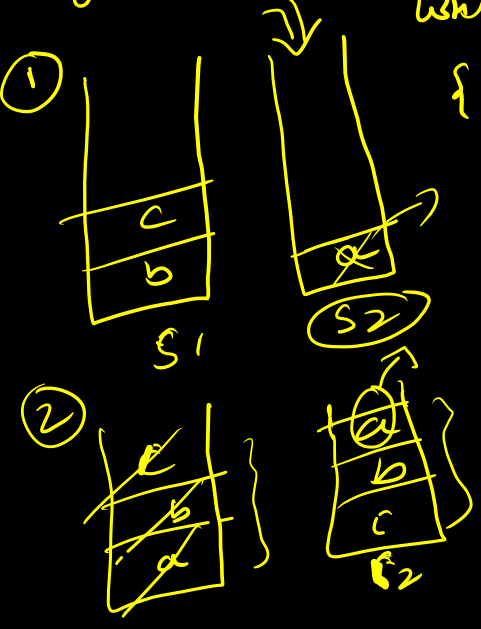
while (! stack empty (S_1))

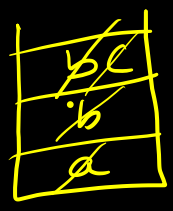
{ x pop (S_1);

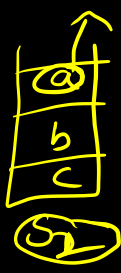
push (S_2, x)

}

x = pop(S_2)







Q → 2 S

⇓

15 Eq

4 dq

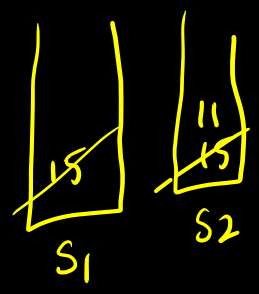
25 Eq

19 dq

⇓

pop = ?

push = ?



push - 80

pop - 63

15 ques = 15 push

1 dq = 15 pop (S1)

15 push (S2)

1 pop (S2)

3 dq - 3 pop (S2)

25 Eq - 25 push (S1)

11 dq - 11 pop (S2)

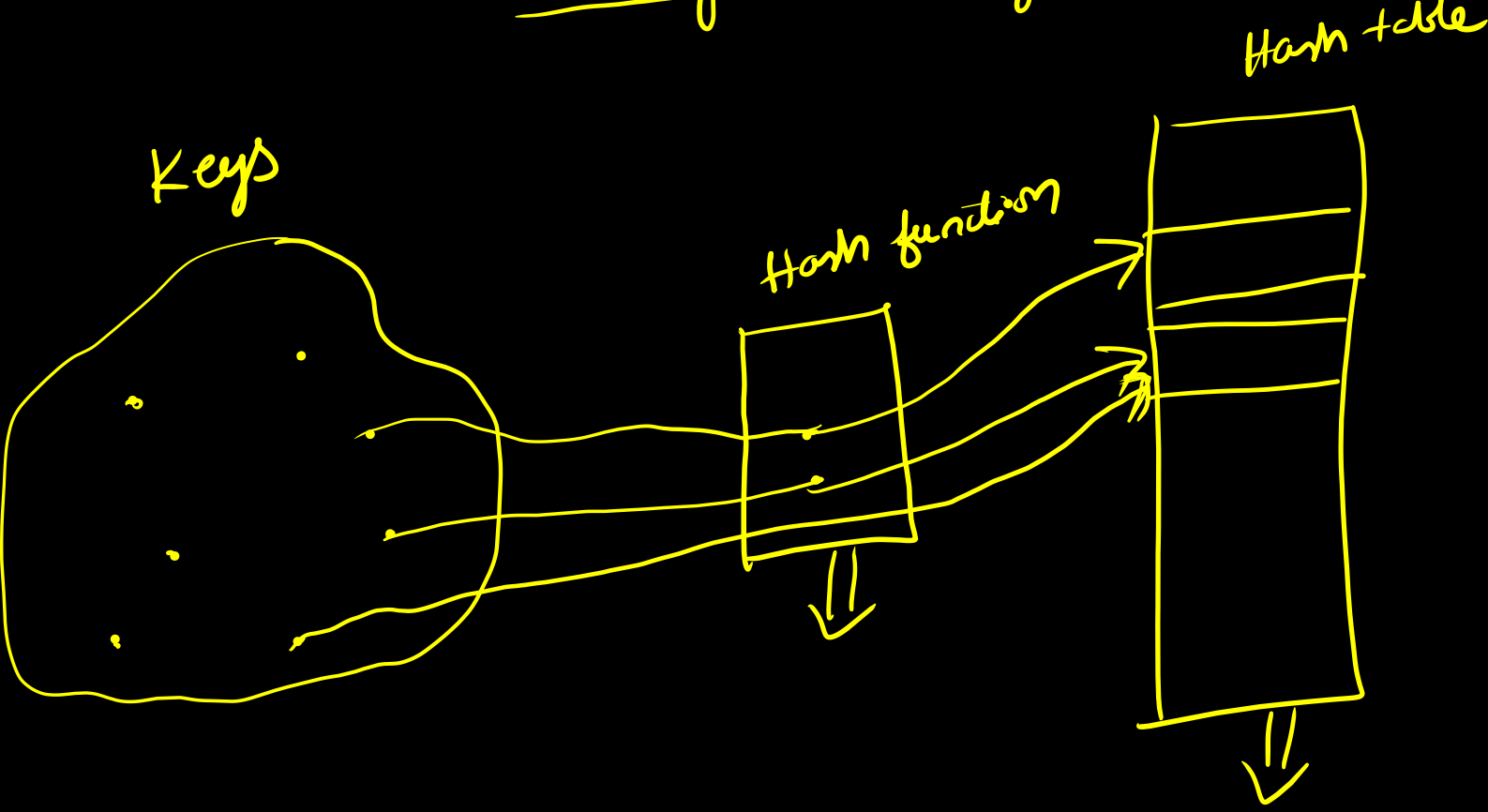
12th dq - 25 (pop - S1)

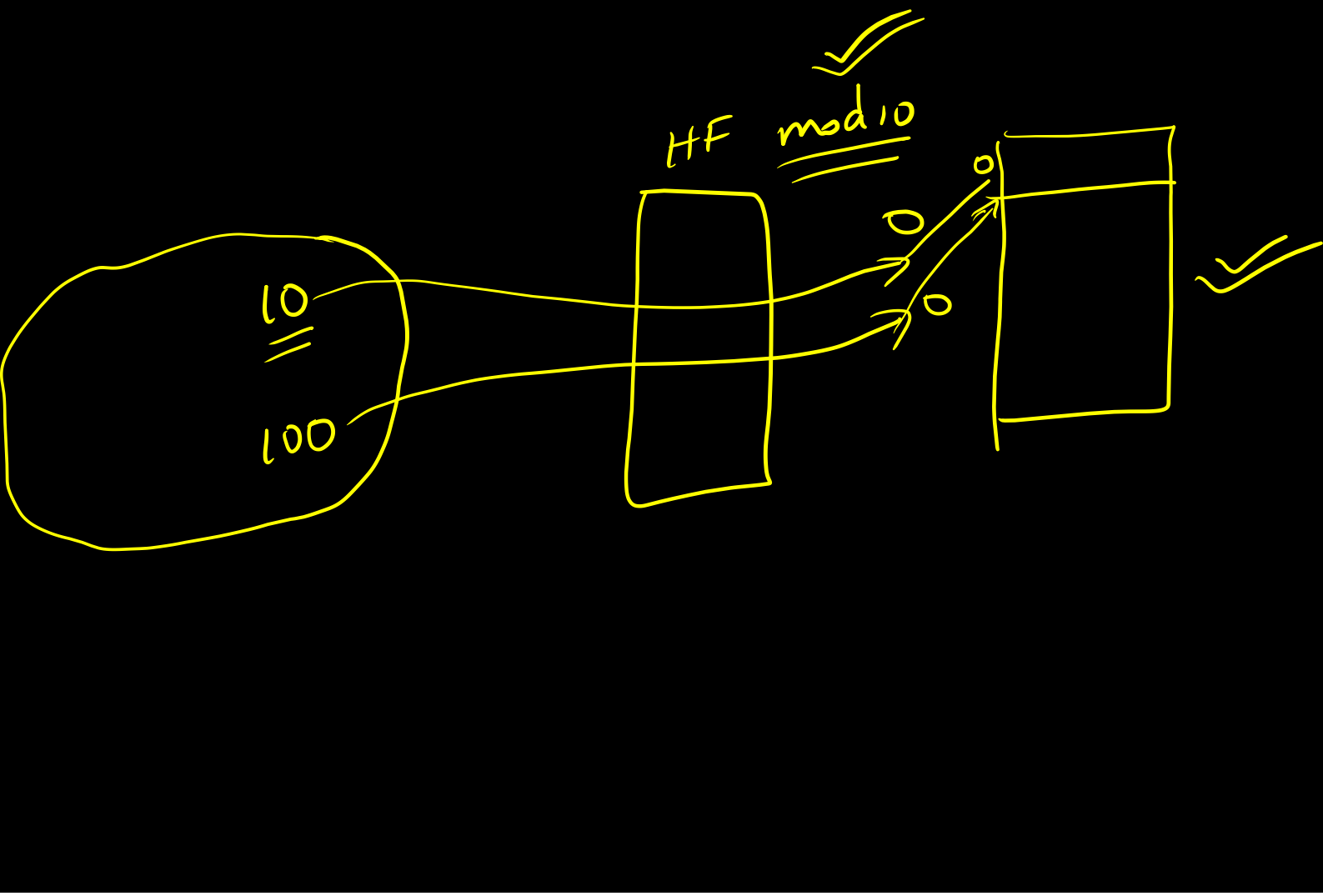
25 (push - S2)

1 pop

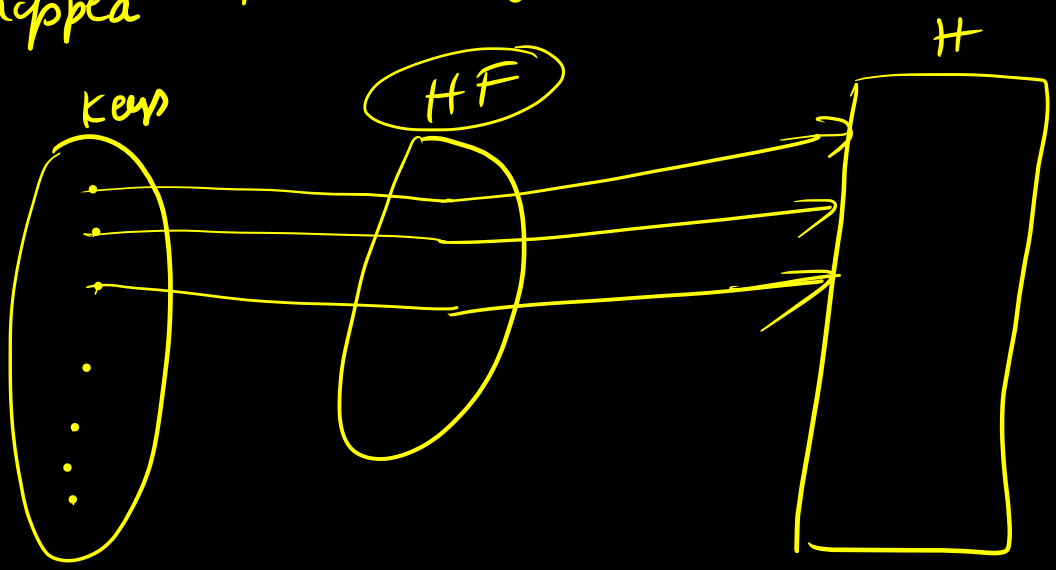
7 dq - 7 pop

Hashing: → easy:




HF mod 10
10
100
0
0
0

Hash function is a mapping technique where one key is mapped to one of the index in Hash table



Disadvantages of HF:

Collisions

Types of Hash functions:

1) Division modulo method
2) mid square method
3) Digit extraction method
4) Fold boundary method
5) Fold shifting method
6) Digit truncation method.

1) Division modulo method:

$HF(key) = key \bmod m$

$m = 1000$

$HF(25) = 25 \bmod 1000 = 25$

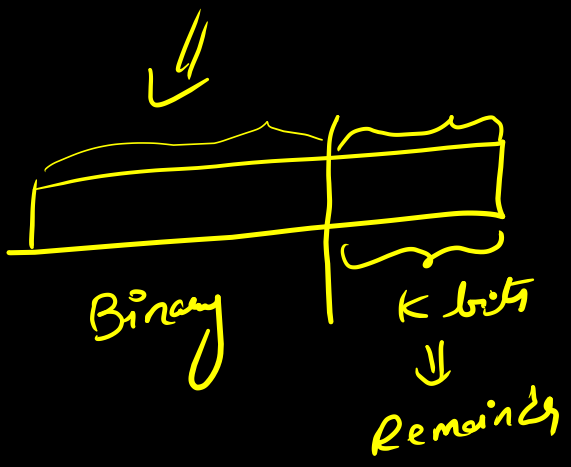
$HF(1625) = 625$

$HF(3005) = 5$ >> collision

$HF(4005) = 5$

in division modulo method → don't take power of as a m

$N \bmod 2^k$ → remainder ⇒



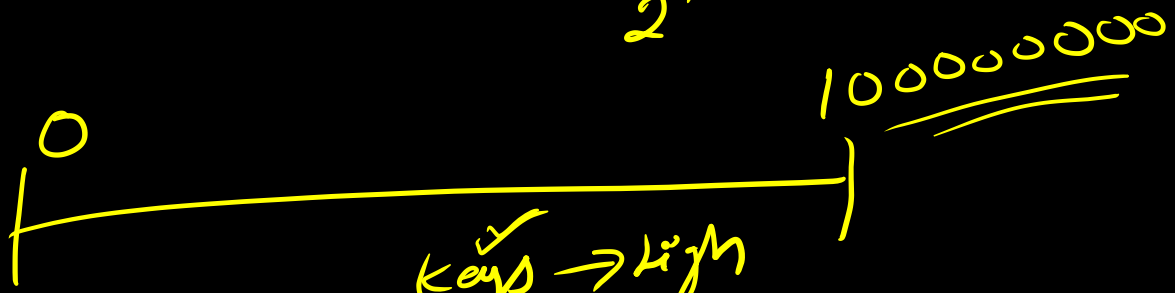
Recommended $m \rightarrow$ prime number far away from power of '2'

61 ✗ → 64

511 ✗ → 512

701 ✓ → 512, 1024

701 → 2^k



Keys → high

take a big number considering HT size also

Mid Square method: (uses all the digits for ~~Key~~ Hash) =>

Key = 5 4 3 8 9

⇓ Square

2 9 5 (8 1 6) 3 2 1

⇓ take mid

816 / 163 (007)

if HT has 0 – 999 values

1000

⇓

1 digit → 0-9 ⇒ 10

2 digit - 10-99 ⇒ 90

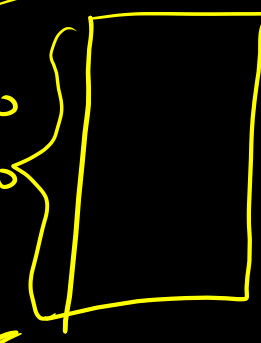
3 digit ⇒ 900

⇓

900

⇓

majority are 3 digits



Digit extraction method :-

HT [1000] (0-999)

Key = 5 8 9 4 6 7 3 2 0

5 6 2

5 6 8 9 6 12 2 1

→ collide.

Fold boundary method:

HT[1000] [0-999]

Key = 589 624 932

1 2 3 4 5 6 7 8 9

589
932
1521

152
1
153

932 123 589

932
589
1521

153

Fold shifting method: HT[1000] (0-999)

Key = 589 624 932

123 456 789

589
624
932
2145

214
5
219

Key$_1$ = 624 589 932

219

Best methods: mid square, Fold shifting.

all digit