Data structure is a process of storing data such that it is easy and fast to access data with reference to its application.

examples:
types of structures
stack
queue
linked list
tree
graph

on memory:
static vs dynamic
static:
-Allocated at compile time.
-easy to give:int a[10];(c++)
-once allocated can not change(-ve)
-memory allocation was on prediction may have underflow or overflow
Mostly:stack/queue/graph

Dynamic: Linked list,tree
-Allocated at run-time.
-may need commands to give: int a[]=new int[10];(-ve)
-once allocated can change as per need(+ve)
-memory allocation is done on demand.

on access Linear vs Non-Liner

Linear:

accessed one after the other.

easy to code

easy to understand

-ve:slow

Non-Linear:

accessed direct or via some guided function

complex to code as need special attention

need time to understand and code

+ve:fast access

ADT:
ADT is Abstract Data Type.
List of operations one must implement in order to implement any data structure. It talks about standardization.

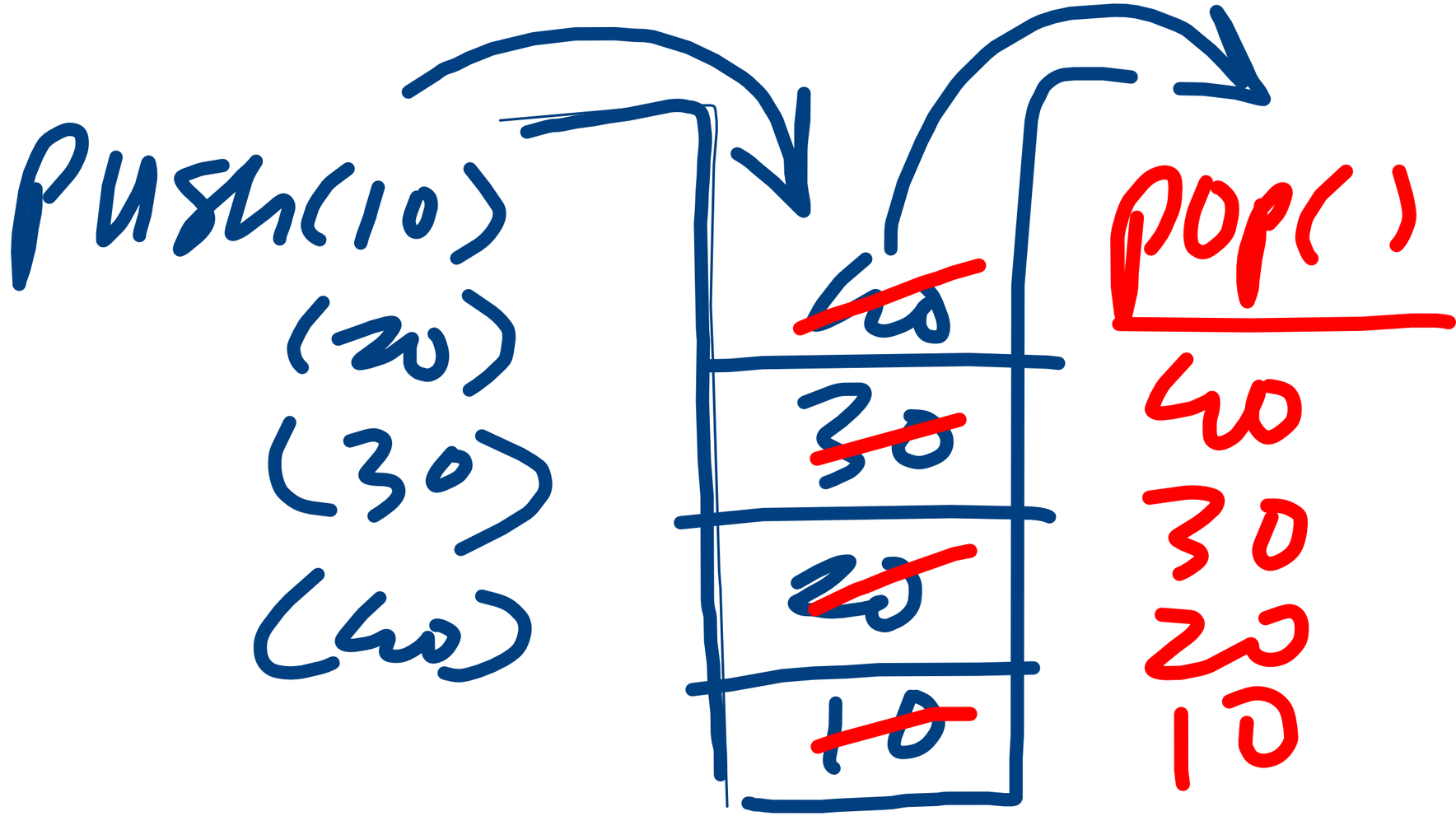
Data structures are complex entities. They require specific sets of operations to operate on, to access, to store, or to retrieve information.
Hence, lists of operations are must.

HABIT
ABIT
BIT
IT

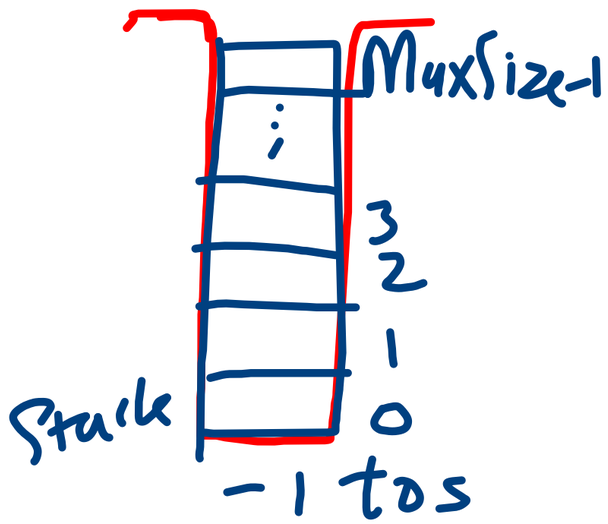
Flow:

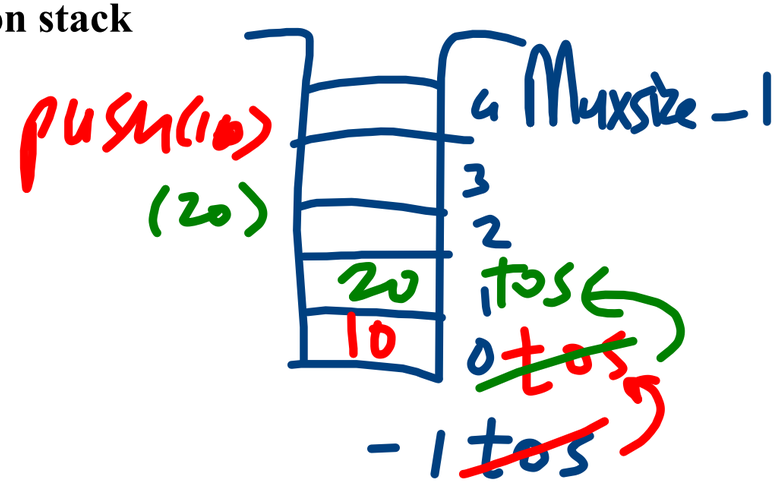
data structure: Diagrams
1.ADT implementation:all done by us
2.Built in implementation:Collection Classes/interfaces
3.Application
4.interview questions

Stack:
single sided
LIFO
linear structure

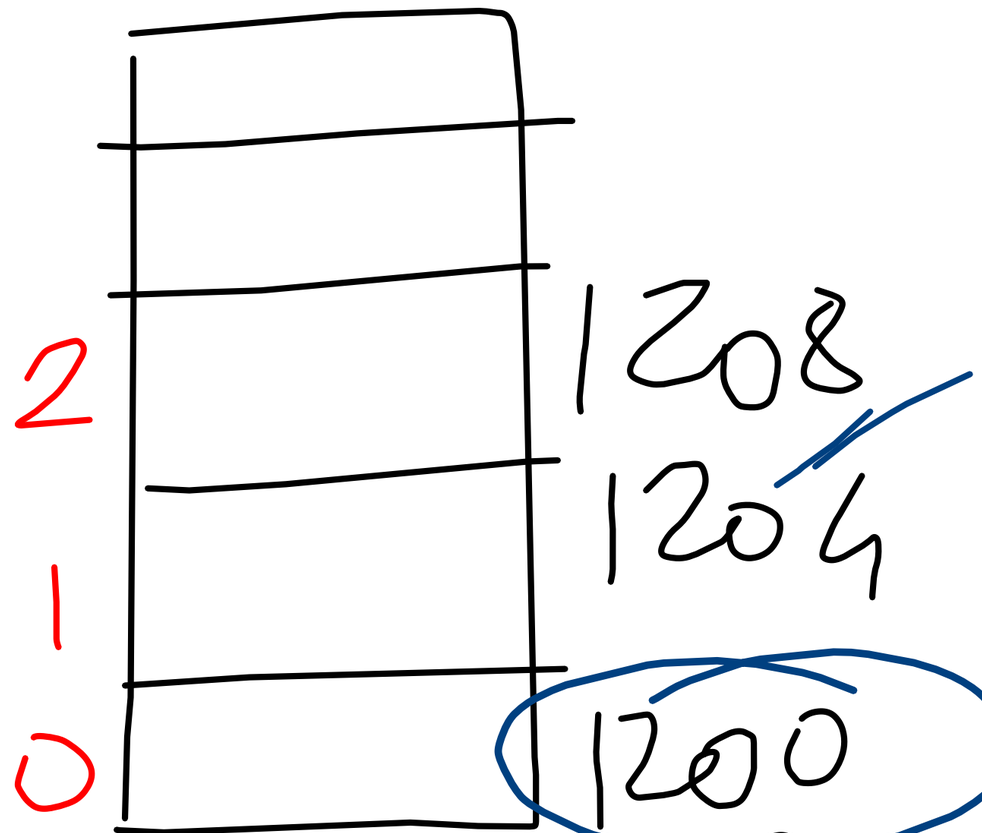


```
class Stack_Class
{
    int stack[];
    int tos,MaxSize;
    void create_Stack(int size)
    {
        tos=-1;
        MaxSize=size;
        stack=new int[MaxSize];
    }

     //push:accepts an element on stack
     //tos+1
     void push(int e)
         {
         tos++;
         stack[tos]=e;
         //stack[++tos]=e;
     }
boolean is_full()
     {
         if(tos==MaxSize-1)
             return true;
         else
             return false;

     //return(tos==MaxSize-1);

      }
```





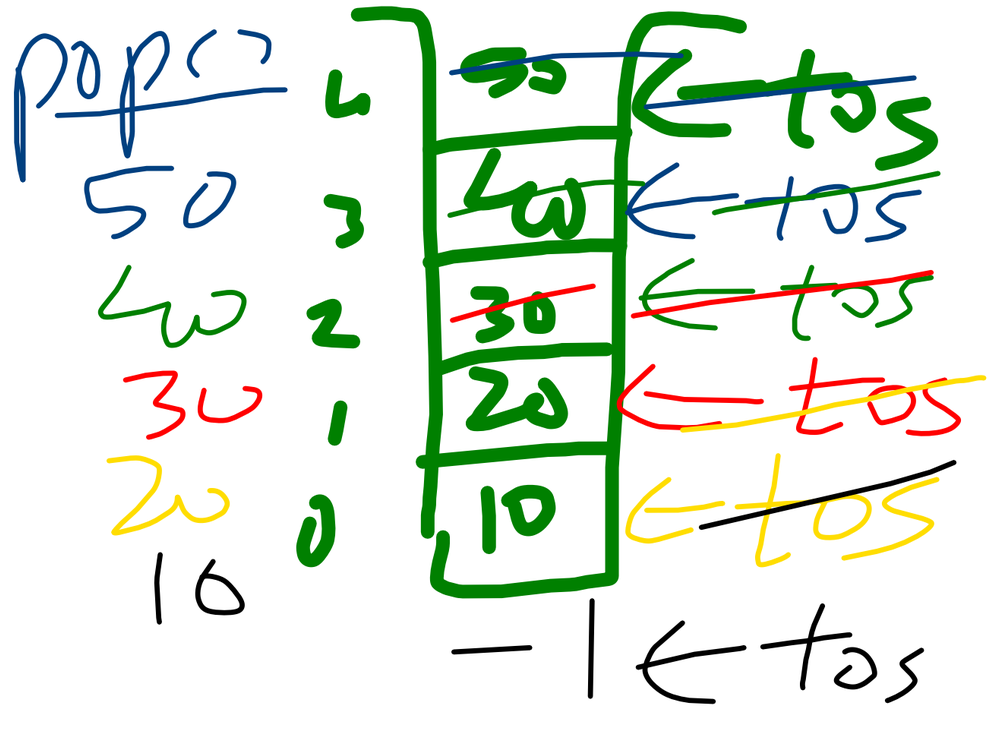
int=4B



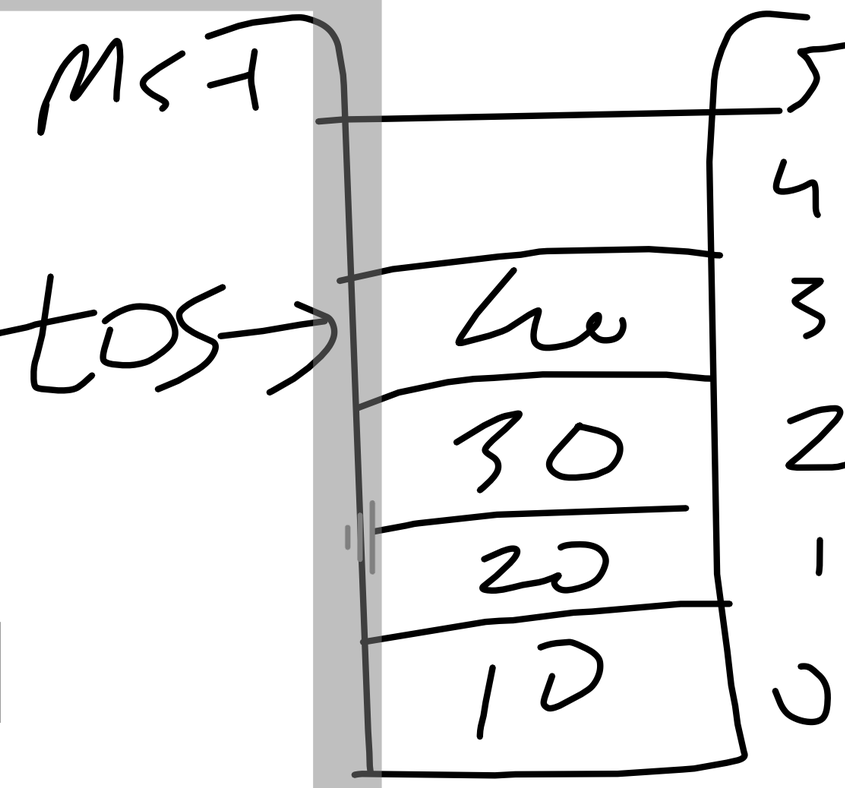
a[2]

↳ Base + 2 Blocks
1200 + 8
1208

```
int pop()
    {
     int temp=stack[tos];
     tos--;
     return(temp);
        //return(stack[tos--]);
    }
boolean is_empty()
    {
    if(tos==-1)
        return true;
    else
        return false;

//return(tos==-1);

 }
```



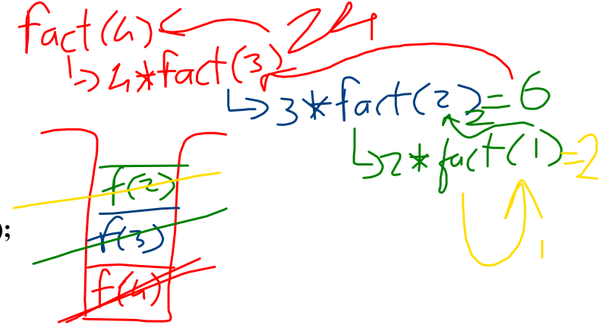
```
void print_stack()//in LIFO
    {
        for(int i=tos;i>=0;i--)
            System.out.println(stack[i]);
    }

int peek()//only return element on top
{
return(stack[tos]);
}
```



Applications:

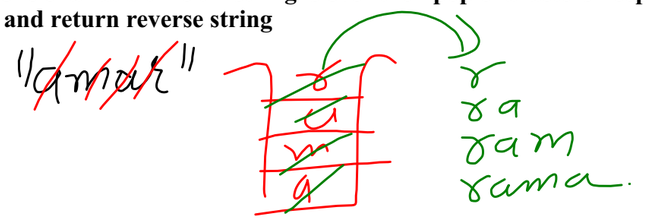
1.used in recursion:

```
int fact(int no)
   {
      if(no==1)
         return 1;
      else
          return no*fact(no--);
   }
```

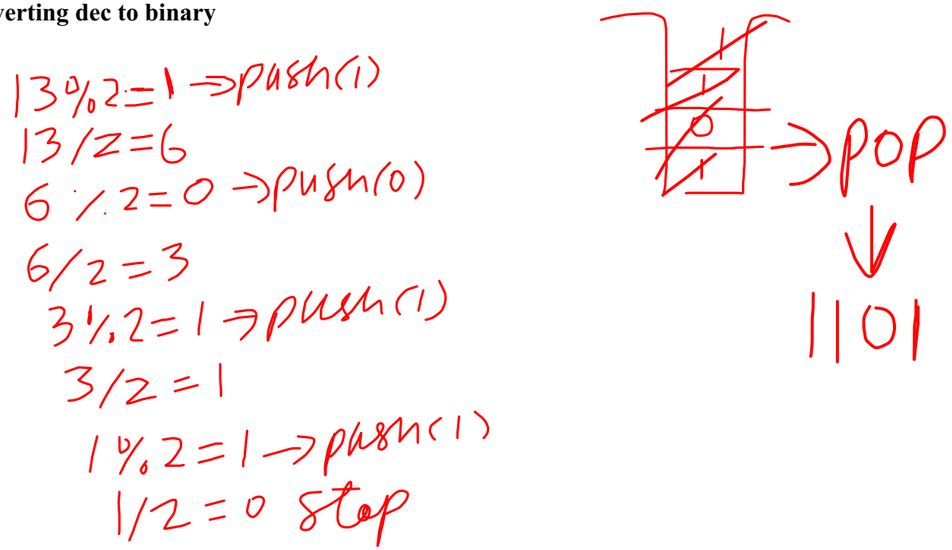


2.reverse string:

add one char at a time till string is over then pop characters keep adding them to string and return reverse string



3.converting dec to binary

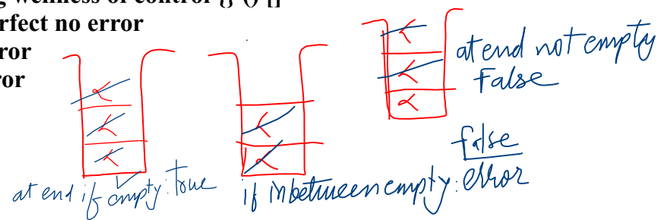


4.checking wellness of control {} () []

{{{}}}:perfect no error

{{{}}: Error

{{{}}:Error



5.Expression conversion and evaluation
different types of expressions:

prefix----------+ab:for s/w add(a,b)

infix----------a+b: created and used for human

postfix----------ab+:for h/w load a,load b, add a,b

.java(source code)---->.class(bytecode)-->jvm(s/w)-->m/c code(post)

convert:

a+b*c-d

a+*bc-d

a+B = +aB

+a*bc-d

A-d = -Ad

-+a*bcd

a+bc*-d

a+B = aB+

abc*+-d

A-d = Ad-

abc*+d-

BODMAS / Operator Precedence

BODMAS =
Brackets → Order → Division → Multiplication → Addition → Subtraction
In order of highest → lowest priority:

Operator	Description	Precedence	Associativity
()	Brackets	Highest	Left to Right
^	Power	Next	Right to Left
*, /, %	Multiply, Divide, Mod	Next	Left to Right
+, -	Add, Subtract	Lowest	Left to Right

*+a/bc+/de*fy

(+a/bc * +/de*fy)

((a+/bc)*(/de+*fg))

((a+b/c)*(d/e+f*g))

④ ① ⑥ ② ⑤ ③

((a+bc/)*(de/+fy*))

(abc/+ * de/fg*+)

abc/+de/fg*+*

do

{ //menu

//choices

switch(ch)

{case1:

.

.

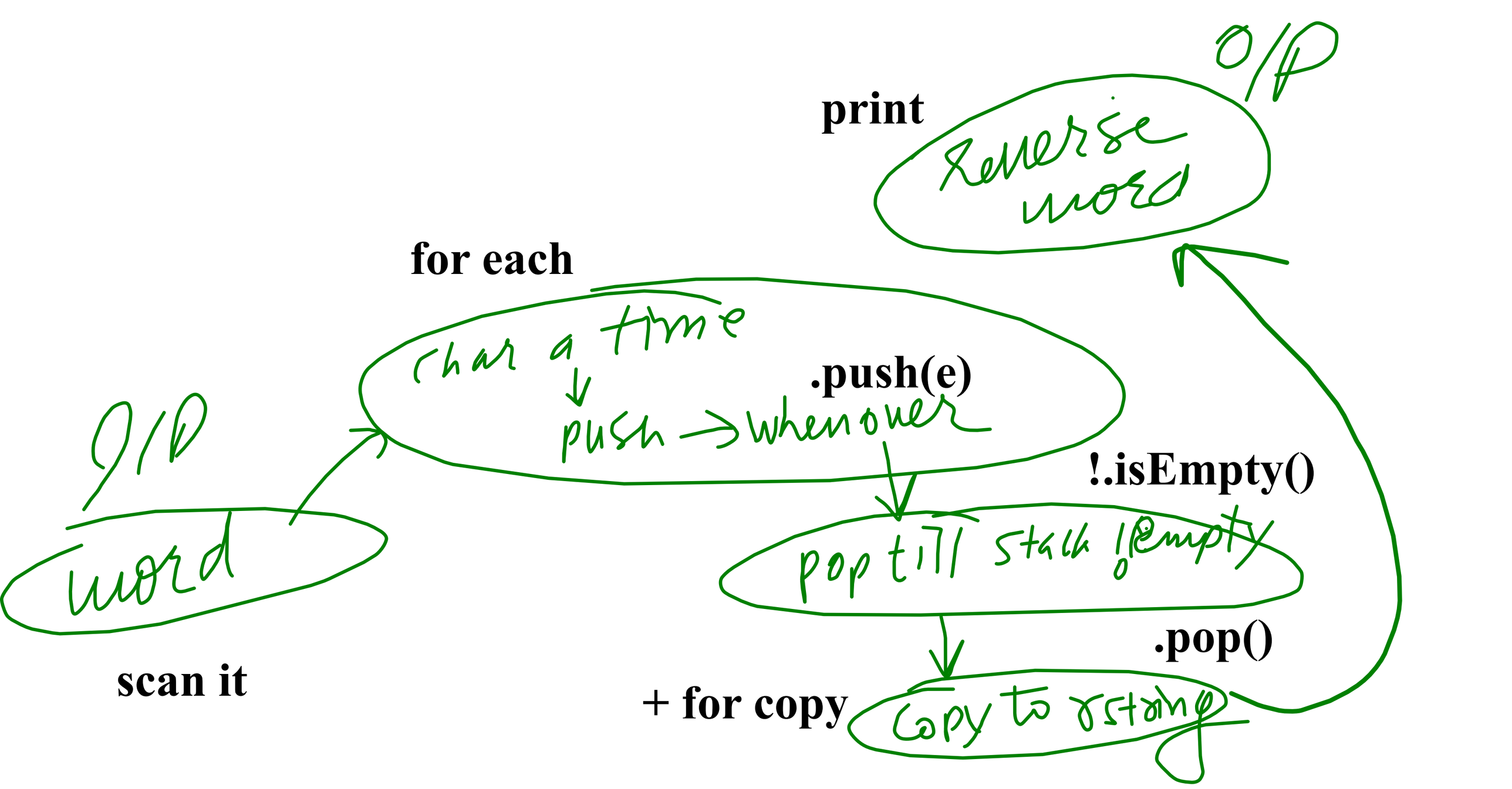
.

default:

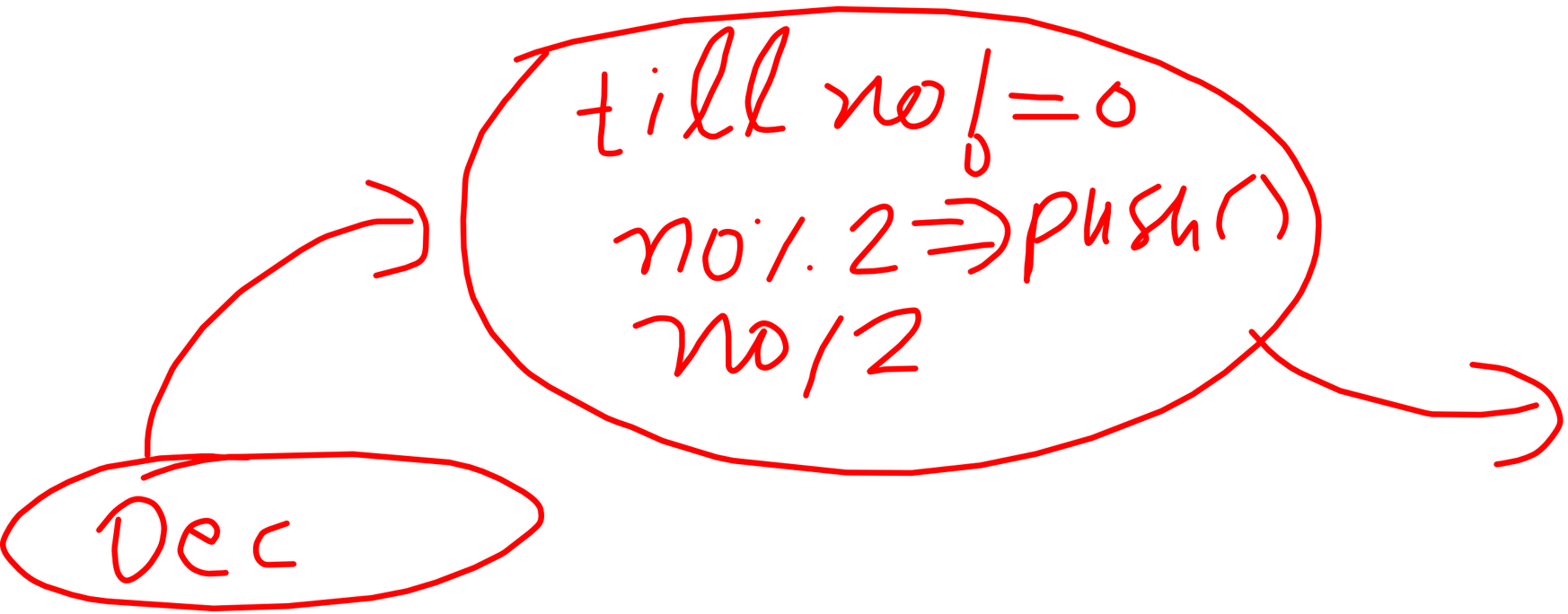
}

}while( );

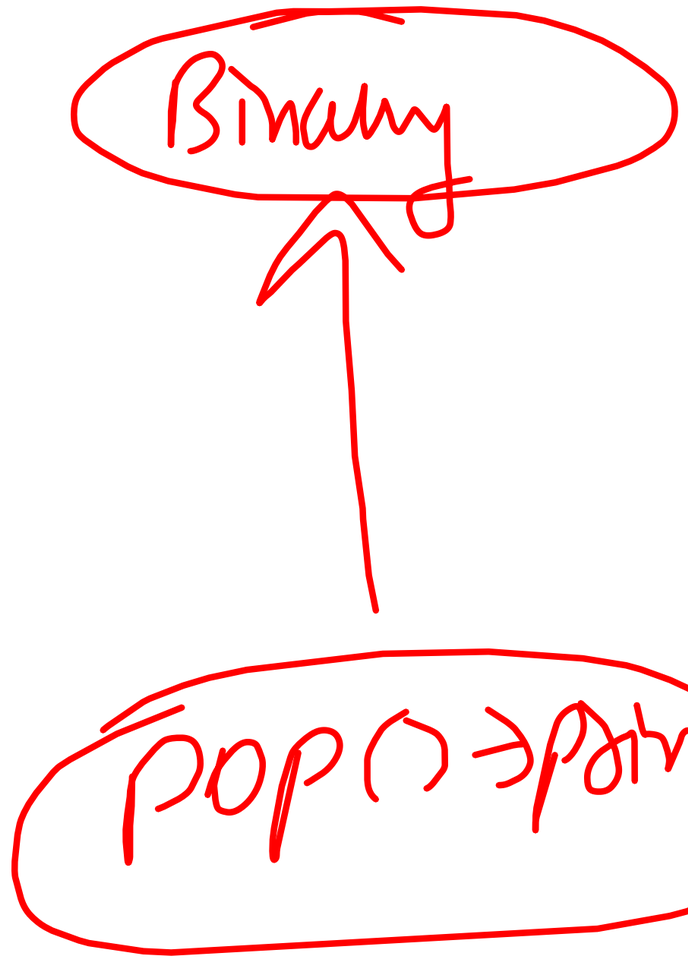
Cond. exit


print
o/p
reverse word
for each
char a time
.push(e)
push → whenever
!.isEmpty()
pop till stack !empty
.pop()
copy to rstring
+ for copy
i/p
word
scan it

stack->Integer

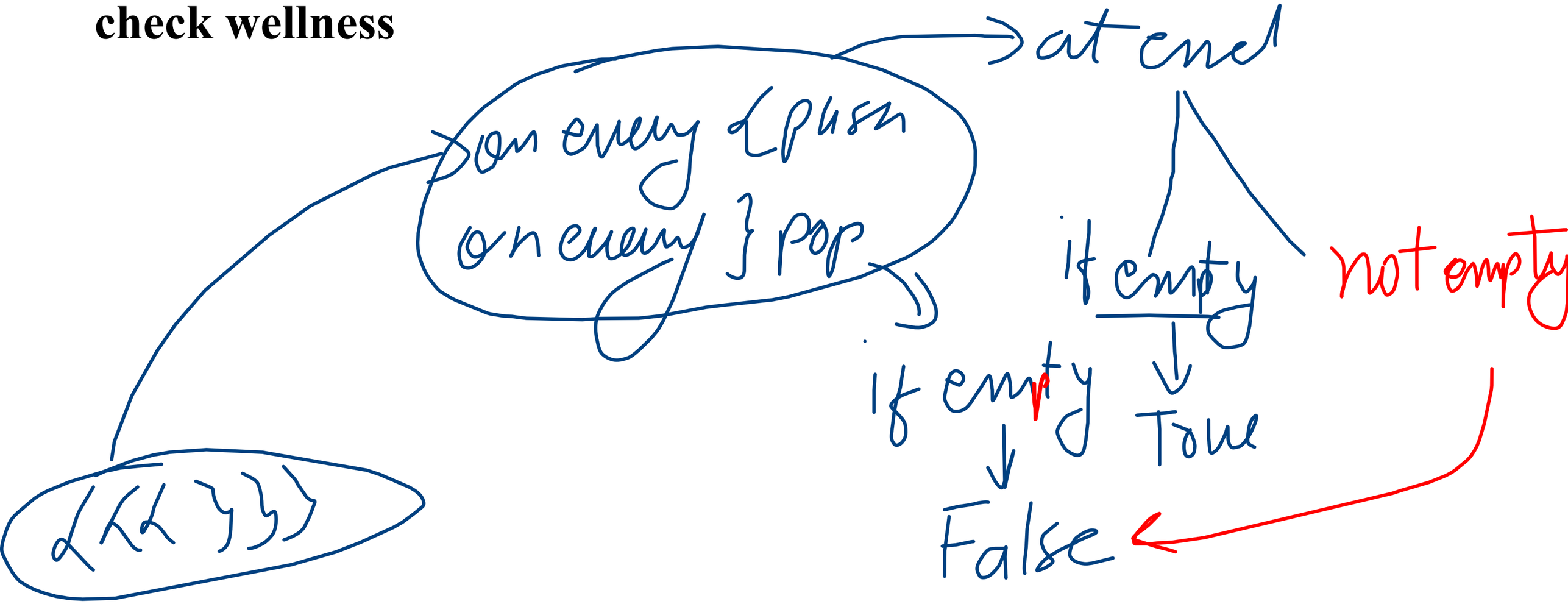


int dec=XX



till !empty()
pop and print

check wellness



char ->stack

Infix → Post

+ to add

string

over → Postfix

till not over

char by char

c=='+' '-' '/' '*'

'a'<=c<='z'

if over

operator

operand

Infix

c==')'

(push

c=='('

) pop till '(' → copy

postfix

P(new)>P(old) push else pop

while/for each

Copy all from stack to post

Postfix