Eg comp-struct (void* a , void* b)

return [(((struct st*) a) → val1no) - (((struct st*) b) → val1no)]

int (*compare) (const void* , const void*)

compare = comp-struct

~~qsort~~ int a [10] = { 1,3,4,5,6,7,8,9,0,1 };

int size = sizeof (a) / sizeof (a[0]);

~~qsort (a, 10, 4, compare);~~

qsort ((void*) a , size, sizeof (a[0]), compare);

~~printf~~

for (int i=0; i<10; i++)

printf ("%d \n");

Arrays

→ finite, ~~set~~ ordered set of homogeneous elements.

finite → has definite size; ordered → elements are arranged in an order; homogeneous → same data type

→ two type:

i) Base Type → type of element (data type)

ii) Index Type → index of an array → Integer → 0 index → C++, 1 index → Matlab; Name

Size / length = $U_b - l_b + 1$

U_b = upper bound, l_b = lower bound

C / C++ : a [i]

a → name of array; i → index position

Three thing to ~~remember~~ specify array

i) name of array

ii) Data type

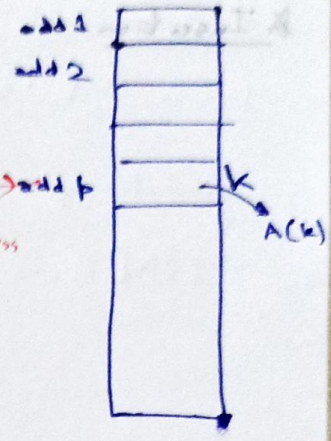
iii) index

int a [10];

~~Location~~

→ Loc (A[k]) := address of k^{th} element of array A

loc (A[k]) = p



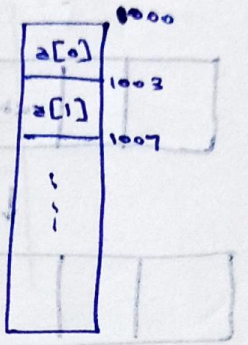
→ Array is stored contiguously
→ sequentially

→ base address := & a[0] = 1000.

~~→ also loc(a[k]) = & a[0] + k * sizeof~~

→ Loc(a[k]) = & a[0] + w(k - lb)

↑ lower bound



→ ~~if~~ where w = size of (a[0])

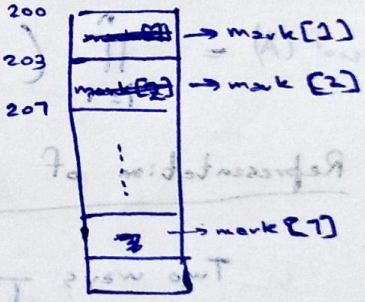
→ due to this, we can access array element in constant time.

Eg: int mark [20];

∴ loc (Mark [7]) = base address + (7-1)4

= 200 + 24

= 224

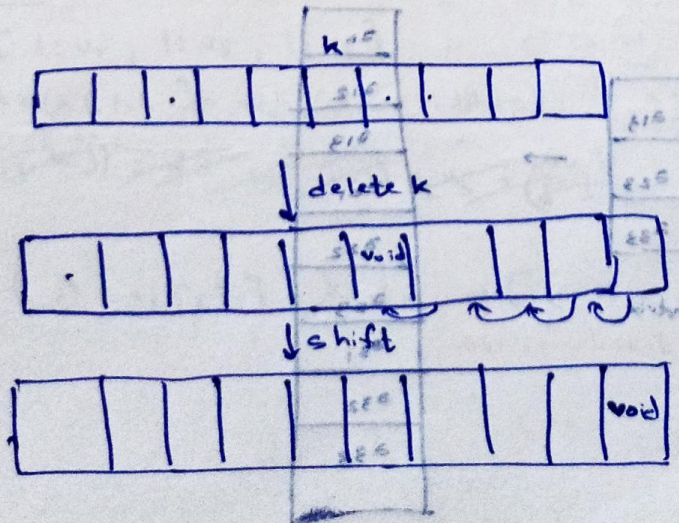


Operations on Array

1) Traverse : Printing array, etc
2) Insertion
3) Deletion.

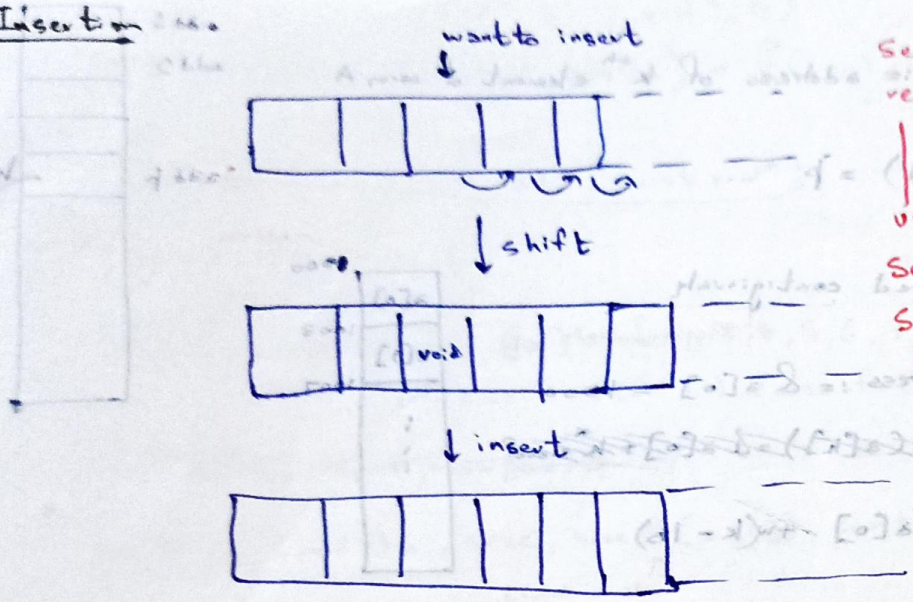
Deletion

~~Eg:~~



Set ITEM = A[k]
for j = k to N-1 do:
set A[j] = A[j+1];
Set N = N-1;
Exit;

Insertion



Set j = N
repeat
  Set A[j+1] = A[j]
  j = j - 1
until j < k;
Set A[k] = ITEM;
Set N = N+1
Exit

Multidimentional Array

$$A[l_1 \dots u_1, l_2 \dots u_2, \dots, l_n \dots u_n]$$

$$\text{sizeof}(A) = \prod_{i=1}^{n} (u_i - l_i + 1)$$

u → upper bounds
l → lower bounds

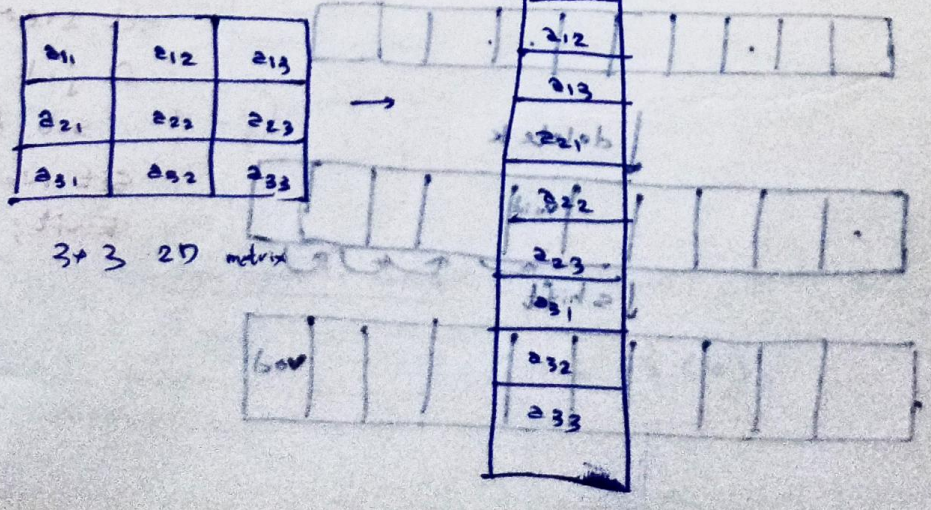
Representation of 2D array in memory

Two ways → Row major Order
  → Column major Order.

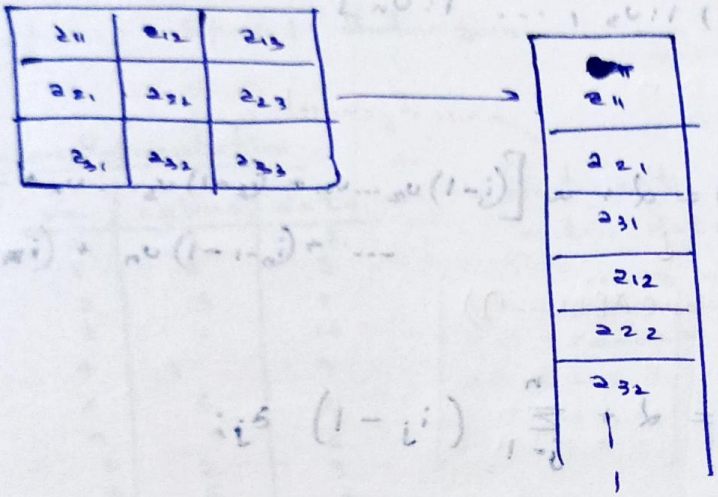
Row major

→ element stored row wise.

Eg

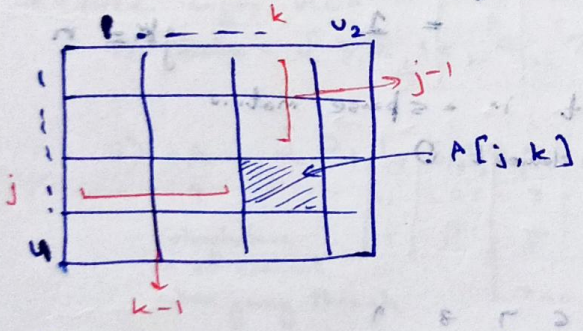


Column Major



~~Multidimensional~~ Location in 2D array

A [1 : u_1 ; 1 : u_2]



let Loc (A[1,1]) = α

so Loc (A[j,k]) = α + w [u_2 (j - 1) + (k-1)] [row ~~column~~ major]

↓ size of (A[1,1])

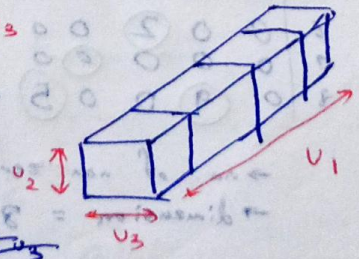
3D array

→ u_1 2D arrays of $u_2 \times u_3$

A [1 : u_1 , 1 : u_2 , 1 : u_3]

→ let loc(A[1,1,1]) = α (base address)

~~loc (A[i,j,k]) = α + w [(i-1) $u_2 u_3$ + (j-1) u_3~~



$$loc\ (A[i,j,k]) = \alpha + w\,[(i-1)\,u_2 u_3 + (j-1)\,u_3 + (k-1)]$$

↑ size of element

n dimensional

$A[l_1:u_1, l_2:u_2, l_3:u_3, \ldots l_n:u_n]$

generalizing

$$\text{loc}(A[i_1, i_2, \ldots i_n]) = \alpha + w\left[(i_1-1)u_2 \ldots u_n + (i_2-1)u_3 \ldots u_n + \ldots + (i_{n-1}-1)u_n + (i_n-1)\right]$$

(α = loc(A[1,1,...1]), w = size of element)

$$= \alpha + \sum_{j=1}^{n} (i_j - 1)\, a_j$$

$$a_j = \begin{cases} \prod_{k=j+1}^{n} u_k & 1 \le j \le n \\ 1 & j = n \end{cases}$$

Sparse Matrix

→ a lot of 0 present in a sparse matrix

→ generally 70% element 0

Two way of represant

Eg:

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	3	0
2	0	0	8	0	0	10	0	0	0
3	0	0	0	0	0	0	0	0	0
4	14	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	2	0	0	0	0	0	0
7	0	0	0	6	0	0	0	0	0
8	0	9	0	0	5	0	0	0	0

→ no. of non zero element = 8

→ dimension = 8 × 9 = 72 element if used 2D. array

I) use co-ordinate list (three column representation)

II) Compressed Spased ~~Matrix~~ Array

3 Column Representation

row	column	data
0	0	8
1	0	3
2	3	6
2	6	10
4	1	9
6	3	2
7	4	C
8	2	9
8	5	5

→ first set is generally kept as matrix data. where

row = no. of rows of ~~as~~ matrix
column = no. of column of matrix
data = no. of non zero element

Compressed Spased Array

→ three array used.

i) ~~data~~ A = { 3, 8, 10, 9, 2, 6, 9, 5 } (top to bottom, left to right)

ii) iA = { 0, 1, 3, 3, 4, 4, 5, 6, 8 }

cumulative no. of element when going through top to bottom

→ no. of element in row 4 = iA[4] − iA[3] = 1

iii) jA = { 3, 3, 6, 1, 3, 4, 2, 5 }

~~cumulative~~ column no. ~~wise~~ wrt A

~~Original Matrix~~ Original Matrix Size = 72

→ Compressed Size = 8 + (8+1) + 8 = 25

(8 → A, (8+1) → iA, 8 → jA)

Addition using 3 column Representation

Eg: A:

	1	2	3	4	5	6
1	0	0	0	6	0	0
2	0	7	0	0	0	0
3	0	2	0	5	0	0
4	0	0	0	0	0	0
5	4	0	0	0	0	0

B =

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	0	3	0	0	5	0
3	0	0	2	0	0	7
4	0	0	0	9	0	0
5	8	0	0	0	1	0

$$C = \begin{bmatrix} 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 10 & 0 & 0 & 5 & 0 \\ 0 & 2 & 2 & 5 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 \\ 12 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

In 3 column Representation

A:

		i →	i →	i →	i →	i →	i →
	row	5	1	2	3	3	5
i →	col	6	4	2	2	4	1
	data	5	6	7	2	5	4

B

		j →	j →	j →	j →	j →	j →
	row	5	2	2	3	3	4	5
j →	col	6	2	5	2	6	4	1
	value	6	3	5	2	7	9	8

C

row	5	1	2	2	3	3	3	3	4	5
Col	6	4	2	5	2	3	4	2	4	1
Value	9	6	7	3	2	2	5	7	9	12

→ dimension will be same as A and B, or it will not be possib

→ increment i & j, and keep using i or j, which has minimum & row value.

→ if row same, check column

→ if i^{th} row & j^{th} row, and i^{th} col j^{th} col are same, add their value, else just copy paste.

→ after adding no. of non zero element can be determined.

~~Stack~~ Array Stack

Stack: Ordered list in which all insertions and deletion takes place at the top. (or one end).

$A = (a_1, a_2, \ldots a_n)$

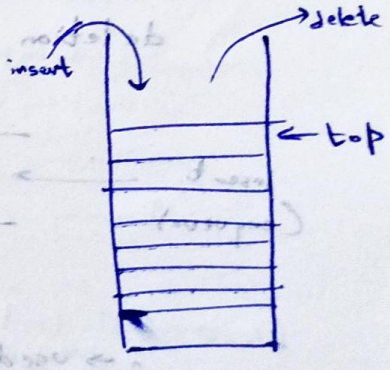
→ LIFO data structure [Last in First Out]

Eg: stack (1 : n) , n = no. of element in the stack array.

(1 ↑ lower bound, n ↑ upper bound)

→ first element would be stored at stack [1], second at stack [2] generally i^{th} element will be in stack [i]

→ top pointer points to the top of the stack.



→ Two operation on stack

i) Push (item)

ii) pop () → returns top element which was deleted.

```
→ bool Push (item)
{
    if (top == n) {
    printf ("Stack Full");
    return ~~1~~ false;
    }
    else
    {
        top = top + 1;
        stack (top) = item;
        return ~~0~~ true;
    }
}
```

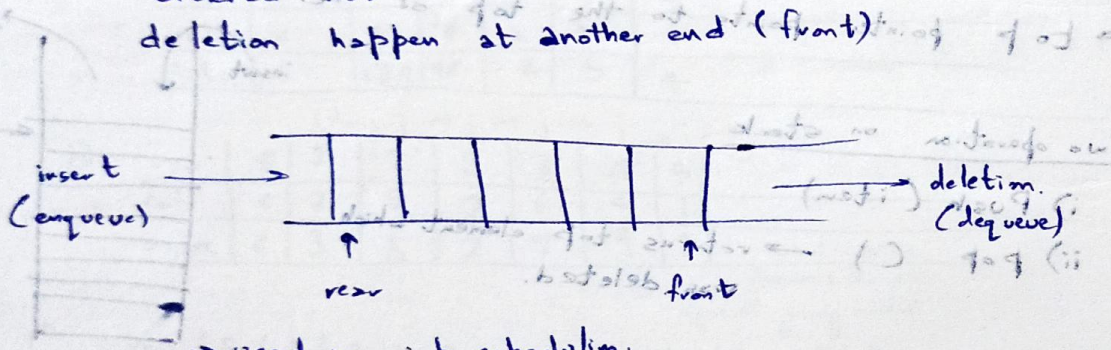
→ <item type> pop ( )

```
{
    if (top == 0)
    {   printf ("Already empty");
        return NULL;
    }
    else{
        item = stack [top];
        top = top - 1;
        return item;
    }
}
```

Array queue;

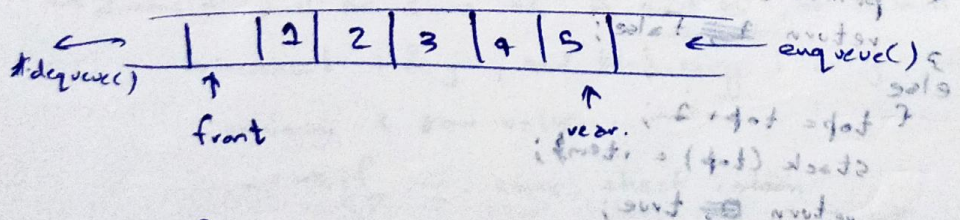
(rear)

→ Ordered list in which all insertion happen in one end and deletion happen at another end (front)



→ used in job scheduling:

→ Q(1:n) front rear, front points to 1 less than actual front



→ Initially front = rear = 0

```
bool
AddQueue (item)
{
    if (rear == n)
    { printf ("Queue Full")
      return false;
    }
    else
    {
      rear = rear + 1;
      Q[rear] = item;
      return true;
    }
}

delqueue ()
{
    if (front == rear)
    { printf ("queue empty ");
      return NULL;
    }
    else
    {
      front = front + 1;
      item = Q[front];
      return item;
    }
}
```

⇒ generally queue will shift to right as time progress, we could refresh the queue, shifting front to 0 and rear to left according to create space for queue.

Eg:	front	rear	Q[1]	Q[2]	Q[3]	...	Q[n-1]	Q[n]
initial →	0	n	J_1	J_2	J_3		J_{n-1}	J_n
delqueue →	1	n	–	J_2	J_3		J_{n-1}	J_n
refresh →	0	n-1	J_2	J_3	J_4		J_n	–
add →	0	n	J_2	J_3	J_4		J_n	J_{n+1}

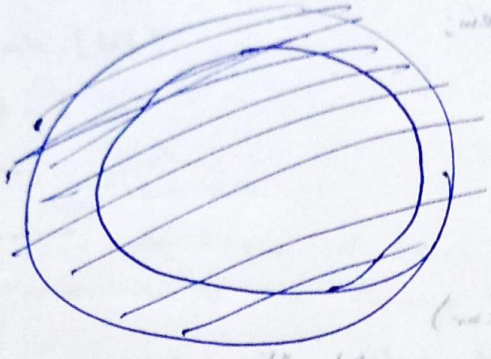
Circular Queue Array

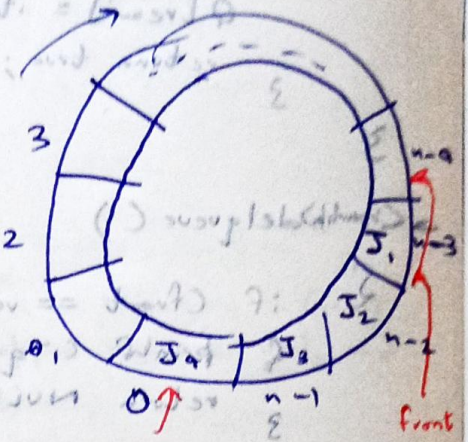
→ To avoid right shifting of queue, we make the array as circular

Q(0 : n-1) → array representing circular queue

→ Inserting from Q[n-1] insert value at Q[0], if vacant, else error

→ front position counter clockwise from the front element





~~if front = rear → empty~~

→ this implementation makes 1 element useless. or wasted.

→ otherwise, more checking will be required, harming efficiency.

front = n-1
rear = 0

→ general test

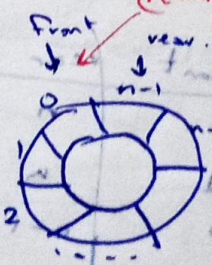
→ if (rear == n-1)
rear = 0
else
rear = rear + 1

→ rear = (rear + 1) % n ;

→ Add Q (item)

```
{
    rear = (rear + 1) % n;
    if (front == rear)
    {   printf("queue full");
        return;
    }
    else
        Q(rear) = item;
}
```

→ (go back to earlier value)



```
→ del Q()
{
  if (front = = rear)
  { printf ("queue empty");
    return nullptr;
  }
  else
  { front = (front + 1) % n;
    item = Q (front)
    return item;
  }
```

Use of stack in arithmetic operation

→ an arithmetic ~~form~~ expression can be written in three form:

i) infix → operators are placed in between operands.

Eg: a+b*c

ii) prefix → operator are ~~ahead~~ ahead of operands. (prefixed)

Eg: a+b*c ⇔ +a*bc
infix prefix

→ also called polish notation.

iii) postfix: the operators is are postfixed to the operands.

Eg: a+b*c ⇔ abc*+

→ also called reverse polish notations

Usage

→ consider expression

$$a/b^{ec} + d*e - a*c$$

is costly for computer by standard infix notation.

→ if we convert it to prefix or postfix to make it less ~~tot~~ taxing for computer.

Bodmas Rules

1) Precedence
→ when an operant in between two operators, which operator will take the operand first is decided by the precedence

Eg: a + b * c
↑
b will be used first with * as it has higher precedence

2) Associativity

Eg: a + b + c
↕ ↕
both have same precedence
↓
as + have left to right associativity, a+b will occur first

→ Associativity describes the rule where operators with same precedence appear in a expression

Two type
i) Left associative : +, −, *, /
ii) Right associative : ^

Eg: $a \hat{} b \hat{} c = a^{b^c} = a \hat{} (b \hat{} c)$

Eg a = b * c
a = b = c

Sr. no	Operator	Precedence	Associativity
1	. ^, v+, v-	6	R - L
2	*, /	5	L - R
3	+, -	4	L - R
4	<, ≤, >, ≥, !=, ≯, ≮	3	L - R
5	and	2	L - R
6	OR	1	L - R

Eg: $a / b^c + d^e - a * c$

$\Rightarrow [(a / (b \hat{} c)) + (d \hat{} e) - (a * c)]$

Eg: $a + b^c \quad (\text{postfix}) \quad a\ b\ c^{\wedge}\ +$

Advantages of Postfix

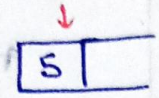
- no need of parenthesis
- no need of precedence and associativity.
- we can use stack to evaluate postfix expression.
  - we start from left, if a number token, push in stack
  - if operator token, pop element (1 for unary, 2 for binary operator), do processing and push the answer back to stack.

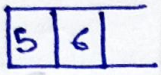
Eg infix → 5 + 6 * 3

put a char signifying end → 5 + 6 * 3 $ ← end

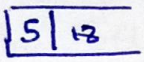
postfix → 5 6 3 * + $

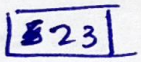
evaluation I) 5 → stack



II) 6 → 

III) 3 → 

IV) * → 

V) + → 

VI) $ → ans = 23

~~Stack~~ Postfix evaluation algorithm

```
Eval (E)        // $ end marker.
{
  // assume nexttoken() , which returns next token
  // assume stack (1:n)
  ~~top if (top = 0)~~
    top = 0;
    while (not at the end of input)
    {
      n = nexttoken (E);
      if (n = $)
        return;
      else
        :
        :
```

```
else if (x is an operand)
    push (x);
else
{   pop correct number of operands. operate it and push back
    result.
}
}  //end of while
}
```

→ as unary - and binary - will collide,
use different character

Eg binary - → −
unary - → # or anything which will not collide

Book → Programming and Data Structure : ~~Horo~~ Horwitz & Sahani
→ Tanenbourn 2

Infix to postfix

Eg: 1) a + b * c ⇔ a b c * +
2) a * b + c ⇒ a b * c +

Algorithm

i) Parenthesize the expression
ii) move all the operators so that they replace their corresponding right parenthesis
iii) Remove all parenthesis

Eg: a / b ^ c + d * e − a * c

i) (((a / (b ^ c)) + (d * e)) − (a * c))
ii)
iii) a b c ^ / d e * + a c * −

To implement this, we need to go through expression two time

1) for Parenthesation

2) moving the operator.

→ These two can be done at a same time.

Eg: a + b * c $ → end

Step	Next Token()	Stack	expression output
1	a	* empty	a
2	+	+	a
3	b	+	a b
4	*	+ * ← this is decided by precedence rule	a b
5	c	+ *	a b c
6	$	* empty	a b c * +

Eg: a * (b+c) * d

Step	Next token()	Stack	expression output
1	a	empty	a
2	*	*	a
3	(	* (	a
4	b	* (	a b
5	+	* (+	a b
6	c	* (+	a b c
7	) → pop everything until we get '('	*	a b c + ← also don't output parenthesis
8	* ↑ Same precedence as Stack ∴ pop the stack	*	a b c + *
		*	a b c + * d
9	d		
10	$	empty	a b c + * d *

Eg: ~~a * b * c * d~~ a ^ b * c + d

Stack
* * +

→ a b ^ c * d +

Eg: a + b * c ^ d

Stack
+ * ^

ans → a b c d ^ * +

→ This algo have a problem with right to left operator

Eg: a ^ b ^ c ⇔ a b ^ c ^ Stack: ^

Step	Token	Stack	output
1	a	empty	a
2	^	^	a
3	b	^	a b
4	^	^	a b ^
5	c	^	a b ^ c
6	$	empty	a b ^ c ^

→ now $ab^{\wedge}c^{\wedge} = (a^b)^c = a^{bc} \neq a^{b^c}$

→ The postfix should have been a b c ^ ^

→ To avoid this, we specify two type of precedence

i) ISP → Input stack precedence

ii) ICP → Incoming precedence

and ISP(^) > ICP(^)

Eg $a \wedge_1 b \wedge_2 c$

Token	Stack	output
a	– –	a
$\wedge_1$	$\wedge_1$	a
b	b $\wedge_1$ ← ISP	ab
$\wedge_2$ ← ICP	$\wedge_1 \wedge_2$	ab
c	a $\wedge_1 \wedge_2$	abc
$	– –	abc $\wedge_1 \wedge_2$

Eg: a + b – c

Step	Token	stack	output
1	a	empty	a
2	+	+	a
3	b	+ → This pop	ab
4	–	– ← same precedence	ab +
5	c	–	ab + c
6	$	empty	ab + c –

→ for left to right → ISP(op) = ICP(op)

Priority Table

Symbol	ISP	ICP
)	-	-
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Infix to Postfix Algorithm

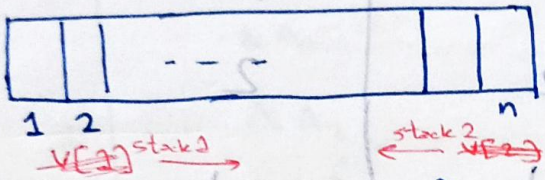
// convert the infix expression E to postfix. Assume the last character of E is '∞'. Stack (1:n) is used as a stack and the character '∞' with ISP('∞') = -1 is used at the bottom of stack.

```
STACK(1) = '∞';
top = 1
while (not at the end of input
{   x = NEXT-TOKEN(E)      // returns next symbol
    if(x == '∞')
    |  while (top > 1)
    |      y = pop();
    |      print(y);
    if (x is an operand)
    |   print(x);
    else if x = ')'
    |  while ((y = pop()) ≠ '(')
    |  |  print y;
    else
    |  while ISP(y = pop()) ≥ ICP(x)
    |  |  print(y);
    |  push(x);
}
```

Multiple Stacks

V(1:m) is to be made to store multiple stack.

→ if we have to store only two stack.

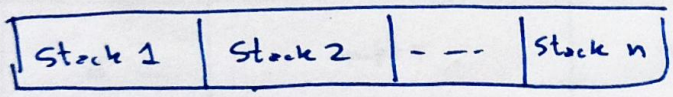


V[1] → bottom element of stack[1]

V[n] → bottom element of stack[2]

→ but this is not possible for more than two stack

→ for n stacks, we segment V(1:m) to n parts, each used by a stack

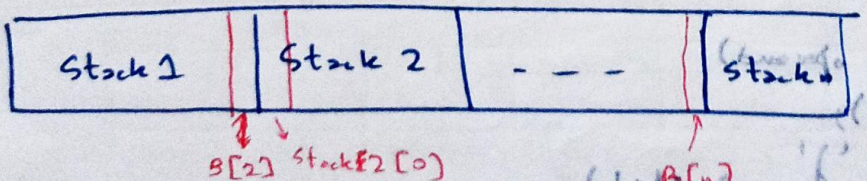


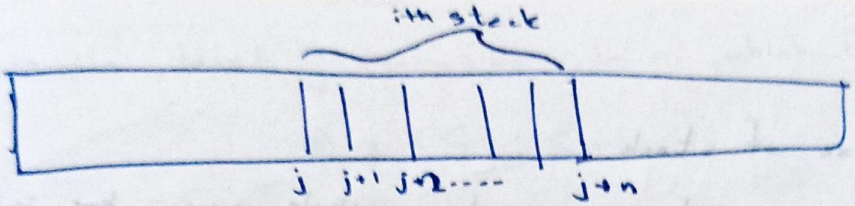
→ if approx size is known, we can have varying size, otherwise we can use, size = m/n

→ to specify bottom of each stack, we use

B[i] = bottom of the i^{th} stack

↳ one less than actual bottom





$B[i] = j - 1$

$T[i] \Rightarrow$ j to j+n (range)

↓

top of i^{th} stack

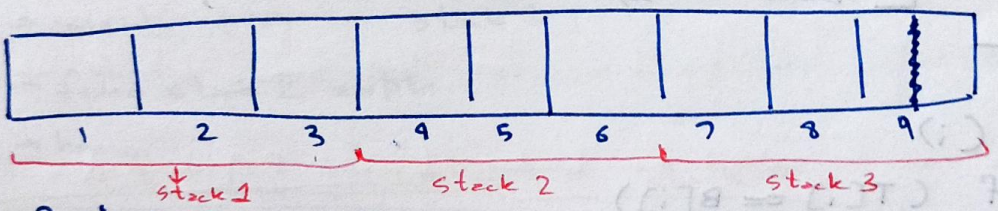
→ when $Top[i] = Bottom[i+1]$, i^{th} stack is full.

→ Initial conditions: (assuming same size)

floor function

$$B[i] = T(i) = \left\lfloor \frac{m}{n} \right\rfloor (i-1) \quad \text{for } 1 \le i \le n$$

Eg:



3 stacks, $m = 9$, $n = 3$

initial

$B[1] = 0 = \left\lfloor \frac{9}{3} \right\rfloor (1-1)$ ⇒ $T[1] = 0$

$B[2] = \left\lfloor \frac{9}{3} \right\rfloor (2-1) = 3$ $\quad T[2] = 3$

$B[3] = \left\lfloor \frac{9}{3} \right\rfloor (3-1) = 6$ $\quad T[3] = 6$

→ ~~stack~~ stack i

range: $B[i]+1 \rightarrow B[i+1]$

→ we set $B(n+1) = m$, (hypothetical) if required.

Disadvantage

→ Fixed Size of stack

→ There may be case where whole array ~~has~~ is not full, but stack may be full

Algorithms

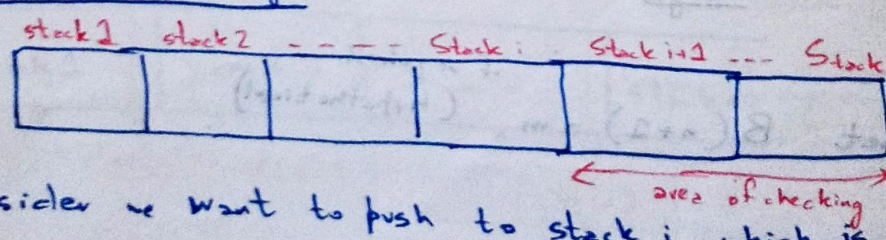
I) Push (~~stacknumber~~ i, item)

```
if (T[i] == B[i+1])
  └ "stack full"      ← This doesn't mean array is full,
                        only stack [~~stack number~~] is full
else
  T[i]++;
  ~~T[i] = item;~~ value (T[i]) = item;
```

II) pop (i)

```
if (T[i] == B[i])
  └ "stack empty"
else
  a = value (T[i])
  T[i] = T[i]-1;
  return a;
```

A Solution to the Disadvantage



→ consider we want to push to stack i, which is already full

S_i full → hence $T[i] = B[i+1]$

↑ shorthand for stack i

1) Determine least j (going from i to n), which follows

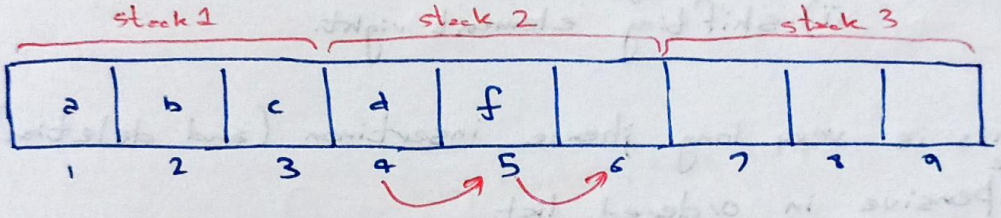
a) $i < j \leq n$

b) j^{th} stack is not empty

i.e $T[j] < B[j+1]$

2) If there exist such j, then move stack $i+1, i+2, \ldots j$ one position to the right, thereby creating space between the stack i and $i+1$

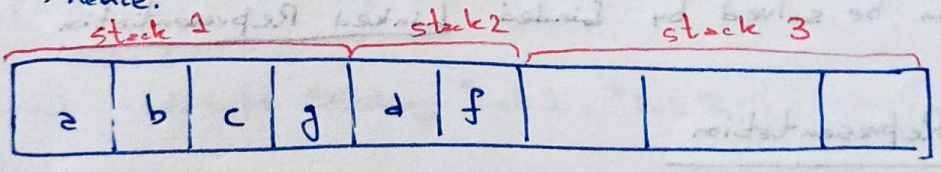
Eg:



→ consider push in stack 1

→ found stack 2 empty

→ hence.



3) if step ① fails, check to the left ~~to the left~~ of i to find largest j such that there is a spare between j and $j+1$ stack.

i.e $T[j] < B[j+1]$

if there is such a j, then move stack $j+1, j+2, \ldots i$ one position left, making spare between stack i & stack $i+1$

4) if step ③ fail, array is full and nothing can be done

Problem

Eg consider ordered list :

45 goes here

10, 15, 25, 35, 40, 50, 60, 75, 80

→ we have to insert 45

→ we have to make room for 45 to insert it

→ can be done by

i) shifting element left

ii) shifting element right.

→ This is very long hence insertion (and deletion) is time expensive in ordered list

→ We also have fixed size of ~~list~~ array (sequencial list), which wastes memory

→ This can be solved by ~~Linked~~ Linked Representation

Linked Representation

→ Data are linked to other data by something (address)

Eg