Data Structures

Defn : It is the way of organizing data in a particular manner so that some specific op can be performed efficiently

Algorithm

↳ finite set of instructions to solve a specific task.

Features

① **Input** : An algorithm may have no input
② **Output** : It should have at least one output
③ **Finiteness** : It should terminate in finite time
④ **Definiteness** : It should have 0 ambiguity
⑤ **Effectiveness** : An istruction is said to be effective when it can be performed by any person in pen & paper in finite time.

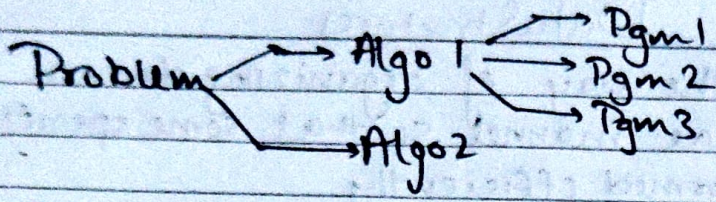
Problem
↓
Algorithm
↓
i/p → [Computer] → o/p

Implementation

① Natural language
② Flow chart
③ Pseudo code

Diff b/w algo & program

Algo	Program
a) finite instruction to perform tasks	a) implementation of algo in code.
b) must terminate	b) might not terminate.
c) written in language, flowchart, pseudo code	c) written in programming language
d) independent of hardware & OS	d) dependent on those.



Correctness of Algo
↳ An algo must give desired output ∀ inputs.

Measure of efficiency
① Time complexity
② Space ...

Running Time depends on
↳ hardware
↳ program.

It can be :-
↳ Practical
↳ Theorifical.

Worst Case
$W(n) = \max(T(i)); i \in D_n$

Avg Case
$A(n) = \sum p(i) T(i); i \in D_n$

Best Case
$B(n) = \min(T(i)); i \in D_n$

Time complexity
↳ Defn : The amt of time taken by an algo to run as a fx^n of the length of the input.

Linear Search

A → array of numbers
k → key is element we've to search

input : A, k
output : true if k is found else no.

Algorithm

```
① i = 1
② while (i ≤ n and A(i) != key)
③     i++
④     if(i ≤ n) then
⑤         return true
⑥     else
⑦         return false
```

Worst Case (Not in all)

$T(n) = 1 + n+1 + n + 1 + 1$

$= 2n + 4$

Best case (1st element is k)

$T(n) = 1 + 1 + 1 + 1$

$= 4$

Avg. Case (middle element is k)

$T(n) = 1 + n/2 + n/2 + 1 + 1$

$= n + 3$

Suppose $T(n) = 5n^3 + 2n^2 + 2$

↳ n^2, 2 term can be ignored for $n \to \infty$.

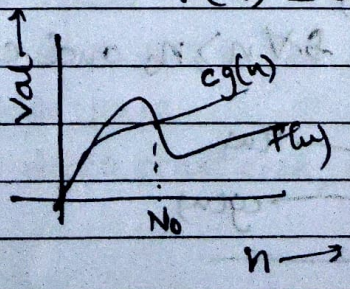
Asymptotic Complexity

a) **Big O Notation (O(n))**

$f(n) = O(g(n))$

if there exists some constant c s.t

$f(n) \leq c\,g(n)$ for $n \geq n_0$



Eg $f(n) = 2n+10$

Ans Let $g(n) = n$

$\Rightarrow f(n) \leq c\,g(n)$ if $c=3$, $n_0=10$

$2n+10 \leq cn$ So, $f(n) = O(g(n))$

$cn-2n \geq 10$ $= O(n)$

$n(c-2) \geq 10$

$n \geq \frac{10}{c-2}$

Eg $f(n) = n^2+3$

Ans Let $g(n) = n^2$

$\Rightarrow f(n) \leq c\,g(n)$ Let, $c=4$, $n_0=1$

$\Rightarrow n^2+3 \leq cn^2$ So, $f(n) = O(n^2)$

$\Rightarrow n^2(c-1) \geq 3$

$\Rightarrow n^2 \geq \frac{3}{c-1}$

Defn of asymptomatic notation

↳ Mathematical notations used to describe the running time of an algorithm as input size increases.

Eg $f(n) = \sum a_m x^m$

Ans $f(n) = \sum a_m x^m$

$= x^m \left(a_m + \underbrace{\frac{a_{n-1}}{x} + \cdots}_{c}\right)$

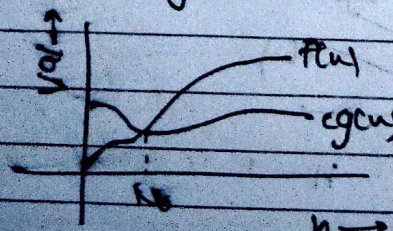
$\leq cx^m$

$f(n) = O(n^m)$

b) Big Omega Notation (Ω)

$f(n) = \Omega(g(n))$

if $\exists\ f(n) \geq c\,g(n)\ \forall n \geq n_0$ and $c > 0$

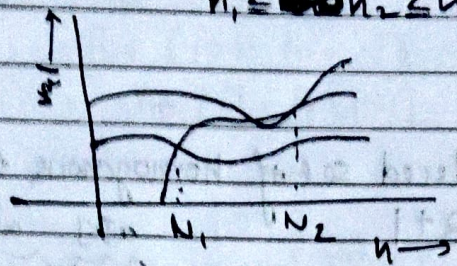


c) Big Theta notation (Θ)

$f(n) = \Theta(g(n))$

if $\exists\ c_1, c_2$ s.t $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$n_1 \leq n_2 \leq n$



d) small o notation

$f(n) = o(g(n))$

if $f(n) \leq c\,g(n)$ $\forall\ c > 0$ & $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

Eg $f(n) = n$.

Ans Let $g(n) = n^2$

$\lim_{n \to \infty} \frac{n}{n^2} = 0$ So, $f(n) = o(n^2)$

e) small omega notation (Ω)

$f(n) = \Omega(g(n))$

if $f(n) \geq c\,g(n)$ $\forall\ c > 0$ & $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Eg $f(n) = n^3$

Ans Let $g(n) = n$

$\lim_{n \to \infty} \frac{n^3}{n} = \infty$ So, $f(n^3) = \Omega(n^1)$

Space Complexity

① Fixed space
↳ instruction space, vars, const

② Variable space
↳ data struct size, env stack

Defn
Amt of space taken by an algo to run as a fn of input length.

Abstract Datatype

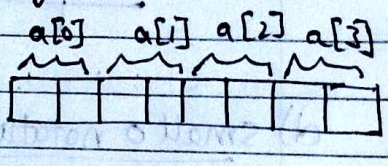
↳ implementation details of a data structure are hidden, but we know the operations of the DS.

Arrays

↳ finite, ordered ~~set~~ of homogenous elements

size = UB - LB + 1

int a[4]; → 0 → 3



int a[13:19]; → 13 → 19

$loc(a[k]) = BA + (\text{size of each element}) \times k$

(BA = base address)

$loc(a[2]) = BA + 2 \times 2 = BA + 4$

if we have a LB,

$loc[a[k]) = BA + (size) \times (k - LB)$

Accessing elements ⇒ independant of time

contiguous memory allocation ⇒ linear datatype

Inserting

can't insert at a[k] if i < k ⇒ a[i] isn't filled

algo (n elements there)

```
insert(A, k){
    i = n
    repeat A[i+1] = A[i]
           i = i - 1
    until k ≤ i
    A[k] = item
    n = n + 1
```

Deletion

Algo:

```
Delete(A, k)
    item = A[k]
    for (i = k to n-1)
        A[i] = A[i+1]
    n = n-1
```

2D array

int a[3][4]; ⇒ arr[m][n] Column major

$A_{11}\ A_{12}\ A_{13}\ A_{14}$
$A_{21}\ A_{22}\ A_{23}\ A_{24}$
$A_{31}\ A_{32}\ A_{33}\ A_{34}$

$A_{11} \to A_{21} \to A_{31} \to \cdots \to A_{34}$

Row Major

$A_{11} \to A_{12} \to A_{13} \to \cdots \to A_{34}$

$$Loc(A[j,k]) = BA + w \times (n(j-1) + (k-1))$$

(Row Major)

$$Loc[A[j,k]) = BA + w \times (m(k-1) + (j-1))$$

(Column Major)

Now if,

$$A[L_1 : U_1, L_2 : U_2]$$

Row major:

$$loc(A[j,k]) = BA + w \times ((U_2 - L_2 + 1)(j - L_1) + (k - L_2))$$

Column Major

$$loc(A[j,k]) = BA + w \times ((U_1 - L_1 + 1)(k - L_2) + (j - L_1))$$

3D Array

↳ pages of 2D arrays.

Row major:

$$loc(A[i,j,k]) = BA + w(i-1)(x_2 x_3) + (j-1)x_3 + (k-1))$$

nD array

$$loc(A[i_1, i_2, \ldots, i_n]) = BA + \sum_{k=1}^{n} (i_k + 1)\left(\prod_{j=k+1}^{n} U_j\right)$$

$\hookrightarrow U_1, U_2, \ldots, U_n$

Advantage of Array

1. All elements can be accessed in constant time.

Disadvantages of Array

1. Size of array is fixed
2. If we want to insert or delete elements, we have to shift and its time consuming.

Sparse Matrix

Lower Triangular Matrix

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \vdots & & & & \\ a_{n1} & \cdots & & & a_{nn} \end{bmatrix}$$

Row Major $Loc_{(i,j)} = BA + \left[\frac{i(i-1)}{2} + j - 1\right] w$

Column Major $Loc_{(i,j)} = BA + \left[\sum_{i=0}^{j-2} n - i + i - j\right] w$

Upper Triangular Matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ & a_{22} & & & a_{2n} \\ & & & & \\ & & & & a_{nn} \end{bmatrix}$$

Row Major $Loc_{(i,j)} = BA + \left[\sum_{i=0}^{i-2} n - i + j - i\right] w$

Column Major $Loc_{(i,j)} = BA + \left[\frac{(i-1)j}{2} + i - 1\right] w$

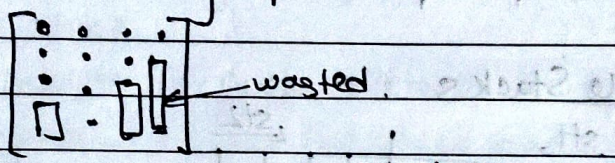
Tridiagonal Matrix

$$\begin{bmatrix} a_{11} & a_{12} & & \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

Address of A_{ij} = $BA + w[2 + 3(i-2) + j - i + 1]$

Pointer Array

in 2D arr, many space are wasted



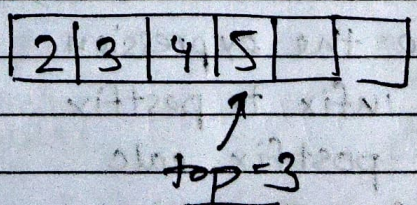
Q Why pointer array?

Ans When size of all rows or cols is not same, there is a lot of memory wasted in using 2D arr. If col size is diff, we can store all of it in a 1D array and store the starting address of all the arrs in another pointer array.

Stack

Defn: Stack is a ADT, coz we don't know the implementation details but we know what operations we can perform.

Top: a var that stores the position of the last element.



int arr[n];

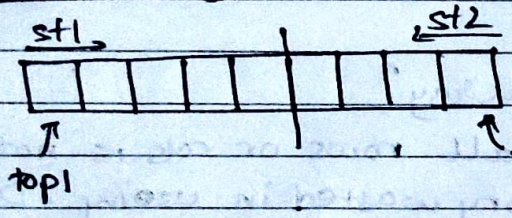
push (item)

```
if (top ≥ n)
   then print (Stack is full)
else top = top+1
   A[top] = item
```

Pop ()

```
if (top < 1)
then print (Empty Stack)
else top = top-1
```

Multiple Stacks



done to save space

full condition.

↳ top2 = top1 + 1

Prefix, Postfix, Infix

Infix: Normal expression
Prefix: first operators, then operands
Postfix: first operands, then operators.

Infix to Prefix & Prefix expression Calc

① Change (to) & (to)
② reverse the expression
③ Apply infix to postfix and postfix calc
④ Final Answer.

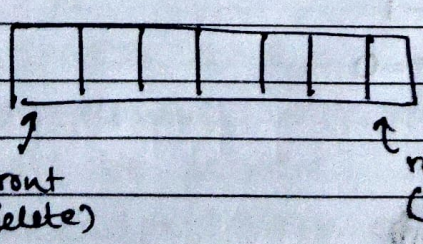
Infix to Postfix

① Scan the infix exp left to right
② if it is an operand, print it
③ Else,
a) if precedence of the scanned operator is greater than the precedence of operator in stack, or if stack has (or is empty, push character in stack
b) else, pop all operator from stack which are greater or equal to in precedence with the scanned operator. After doing that, push the scanned operator to the stack.
④ if char = (, push to stack
⑤ if char =), pop the stack and output till (is found
⑥ repeat 2-5 till expression is scanned
⑦ print output

Calculate Postfix

① Create a stack to store operands
② Scan the expression L to R
a) if element is a number, push it to stack
b) if operator, pop 2 operands and calc the result and then push the result into the stack
③ pop to get final answer

Queues

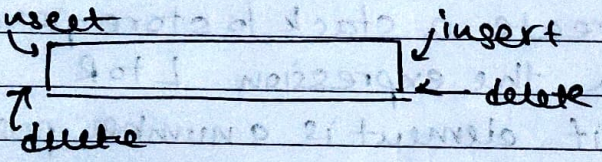


n ⇒ size of array
policy ⇒ First in First Out.

AddQ(item)
if (r=n-1)
then print (Queue full) return;
else
if (f-r=-1) [initial condition]
then f=r=0
else
r=r+1
Q(r)=item

deleteQ()
if (f<0)
then print (Queue empty), return;
else
f=f+1

Deque
↳ Delete ended queue
we can insert or delete at any of the two ends.



InsertF(item)
if f ≤0
then print (Queue Full) return
else
if f=r=-1
then f=r=0
else
f=f-1
Q[f]=item;

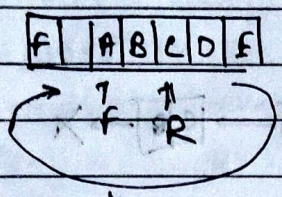
deleteR()

```
if f=r=-1 then
    print (Empty Queue) return
else if f=r
        item=Q(r)
        f=r=-1
    else
        item=Q(r)
        r=r-1
```

Input restricted Queue
↳ insertion at only 1 end

Delete restricted Queue
↳ delete at only 1 end

Circular Queue



if (R >= F)
↳ num = R - F + 1

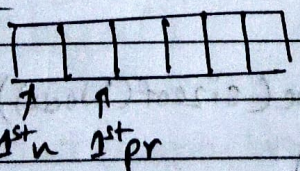
if (F > R)
↳ num = N - (F - R - 1)

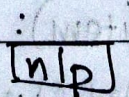
if (F = R = -1)
↳ num = 0

Priority Queue

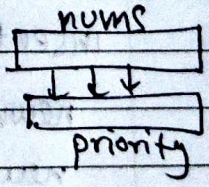
The elements in the queue are stored in priority. Number with highest priority is deleted first.

Way #1





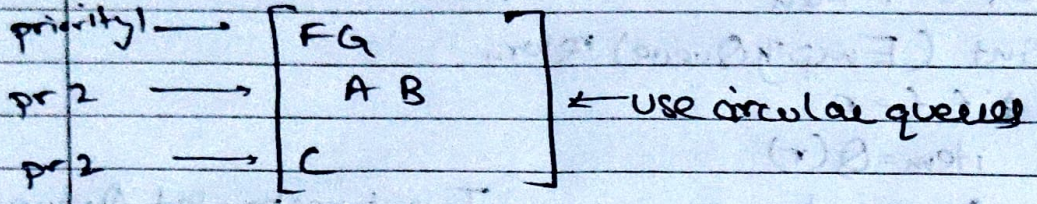
Store like this



- bad space usage.
- if two elements with same priority, the earlier will be deleted.

We need to traverse every time, so its bad.
O(n) → time.

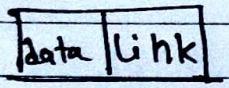
Best option



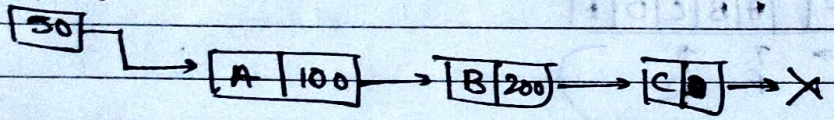
Better soln : Linked list & heaps.

Linked list

node



start



We can implement this using two arrays - one with data, one with link but it's not dynamic & wrong.

```
struct Node{
  int data;
  struct Node* next;};
Node* start = NULL
```

Node Structure

insert begin(&item):

```
newnode = (Node*) malloc(sizeof(Node))
newnode → data = item
if start == NULL
then, newnode → next = NULL
else
  newnode → next = start
  start = newnode
```

```
insertend(item)
  nn=(Node*)malloc(sizeof(Node))
  nn->data=item.
  nn->next=NULL
  if start==NULL
  then start=nn
  else temp=start
  while(temp!=NULL)
     temp=temp->next
  temp->next=nn

insertmid(k,item)
  nn=(Node**)malloc(sizeof(Node))
  nn->data=item
  temp=start
  while(temp->data!=key && temp!=NULL)
     temp=temp->next
  if temp==NULL
     print(can't be added)
  else
     nn->next=temp->next
     temp->next=nn
deletebegin()
  if start==NULL
     print(empty list)
     end
  else
     temp=start
     start=ptr->next;
     free(temp)
```

```
deleteend()
 if start= NULL
   print(Empty list)
 else
   temp=start
   if(temp→next=NULL)
       curr=start
       start=NULL
       free(curr)
    else
        curr=start
        while(curr→next!=NULL)
            prev=curr
            curr=curr→next
        prev→next=NULL
        free(prev)
deletemid(k)
 if start=NULL
   printlempty list)
 else
   temp=start
    if(temp→data=key)
    {start=temp→next
      free(temp)}
    else
      temp=start
      while(temp!=NULL)
          if(temp→data!=key)
```

```
        prev = temp
        temp = temp->next
    else
        prev->next = temp->next
        free(prev)
    if(ptr = NULL)
        print("Element not in list")
```

Stacks in LL

push → insertend
pop → delete end

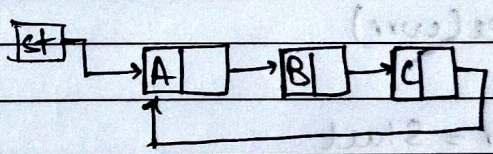
top ⇒ last node

Queues in LL

push → insertend
pop → delete front.

front ⇒ first node
rear ⇒ last node.

Circular Linked Lists



start → ptr1
end → last node

```
insertbeg(item)
    nn = (Node*) malloc(sizeof(Node))
    nn->data = item
    if(start == NULL)
        nn->next = nn
        start = end = nn
    else
        nn->next = start
        start = nn
        end->next = start.
```

```
insertend(item)
  nn = (Node*)malloc(sizeof(Node))
  nn->data = item
  if(end == NULL)
     nn->next = nn
     start = end = nn
  else
     nn->next = start
     end->next = nn
     end = nn
deletebeg()
 if(start = NULL)
    printf("Empty list")
 else
    if(start = end)
       curr = start
       start = end = NULL
       free(curr)
    else
       curr = start
       temp = start
       temp = temp->next
       start = temp
       last->next = start
       free(curr)
deleteend()
 if(start = NULL)
    printf("Empty List")
 else
    curr = start
    while(curr != start)
       prev = curr
       curr = next
```

prev->next = start

(*) if(start = end)
curr = start
start = end = NULL
free(curr)

essentially insertbeg is insertend.