Data Structure and algorithms

18/7/22

Data - Raw set of values

Information - meaningful data

[me - Entity
my infos - Attributes]

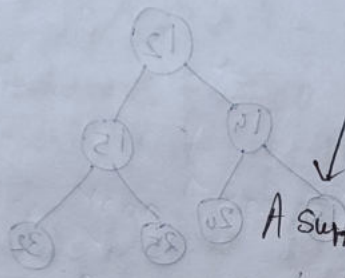
Build in datatype - Int, float... etc

User defined datatype / Abstract datatype (stack etc)

Data structure have 3 things → Domain of the data (The range of the data)

→ Functions (set of operations)

→ Asumps



- struct complex
{
float real;
float imaging;
}

Real = { set of int } Domain

imag = { , , , }

F = { +, -, *, / }

A = $z_1 = a_1 + b_1 i$, $z_2 = a_2 + b_2 i$ } { $(a_1 + a_2) + i(b_1 + b_2)$ }

→ Data structure

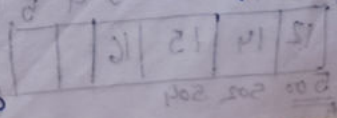
Fundamental d.s

(Classical d.s) → linear and non linear

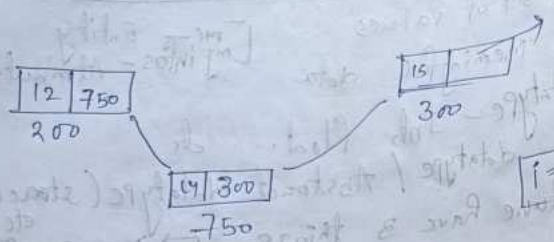
linear: Array, (stack, queue, linked list) [sequential ordering]

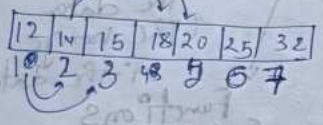
non linear (maps, Tree, Graph)

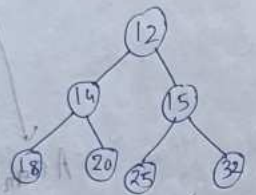
As it has contigious memory allocations



Linear DS







not in linear relationship

- L.C = 2×i (ith index)
- R.C = 2×i + 1 (ith index)

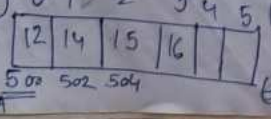
Array → Composite d.S

Properties → ① Linear [Contigious memory location]
② Finite [fix no. of elements]
③ Homogeneous [Same types of elements]

① Base Address - The address of the 1st element

② Indices - Subscript by which we can know the index of the array (Indexes)

③ Range



A = {0:5}

- No of elements in array - size of the array = U - L + 1 [5 - 0 + 1 = 6]
[U = Upper bound, L = Lower bound]

- Address of ith element in array -

Base Address - 500
Find address of 4th element -

B.A + (i - 1) × w [w = 2] [A[w] when L.B is starting from '1']

[500 + (4 - 1) × 2 = 506]

B.A + (i - L) × w [when the L.B (Lowers bound) is ~~not~~ starting from '-1', array index]

Operations on array

① Traversing / Array traversed
② Inserting
③ Deletion
④ Sort
⑤ merge
⑥ Search
⑦ Reversal

Algorithim (step by step Procedure to execute a Programme)

5 Characteristics
- Input
- Output
- Procedure
- unambigious
- Finite
- Definite

Algorithim traversal

Input: An array A with elements, L is lower bound and a is upper bound
Output: Print elements of A
D.S: 1D-Array A
Steps:
1: i=L // start from location i
2: WHILE (i<=u) DO
3: PRINT A[i]
4: i=i+1 // move to next location
5: ENDWHILE
6: STOP

22/7/22

Suppose an array A[-15 ... 64] is stored in a memory with starting address 459. Assume the word size of each element is 2.
i) How many number of elements one must in A?
ii) How much memory is required to store A?
iii) what is the location for A[50]?
iv) what is the location of 10th element?
(v) which element is stored at 589?

(i) u-L+1
= 64-(-15)+1
= 80 elements

(ii) 160 [∵ 80×2 (int's Byte size)]

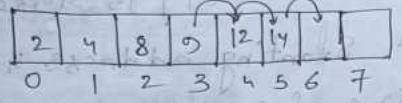
memory = No. of elements × w
location/address here

(iii) ~~50th index's ele~~
The location of A[50] is - B.A + (i-L)×w
459 + (50+15)×2
= 589

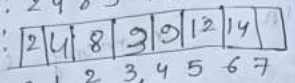
(iv) 459 + (10+15)×2 [∵ B.A + (i-L)×w]
= 509

(v) A[50]
the element located at

Algorithm Insertion



step 1: 2 4 8 9 12 14 14*
2: 2 4 8 9 12 12 14
3:



9 is out
3 is in

key = 3
position = 4

A[i] = A[i-1]
↳ (shifting the elements backward)
A[loc] = key
↳ (put the element in the location)

Algo Insertion

Input: KEY is the item, LOCATION is the index of the element where it is to be inserted.
output: Array enriched with KEY
D.S: Array A[L ... U]
steps:-

1. IF A[U] ≠ NULL THEN
2. PRINT "Array is full: Insertion not possible"
3. EXIT
4. ELSE
5. i = U
6. WHILE(i > LOCATION) DO
7. A[i] = A[i-1]
8. i = i - 1
9. ENDWHILE
10. A[LOCATION] = KEY
11. ENDIF
12. STOP

Searching

12	13	15	18	5	2	1	7
0	1	2	3	4	5	6	7

key = 5
O/P: Location = 4 (index of key)] successful search

Here location is index

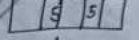
If the element is not present in the array then the case is unsuccessful search.

Algo searching

Input: KEY is the item to be searched
output: Index of KEY, or a failure message
D.S: Array A[L-U] where L is the lower bound and U is the upper bound.
steps:
1. i = L, found = 0, location = -1 // found = 0 indicates search is not finished and is unsuccessful
2. WHILE(i <= U) and (found = 0) DO
3. IF (A[i] = KEY) THEN
4. found = 1
5. location = i

and = simultaneously work
If 1 condition fails

6. BREAK
7. ELSE
8. i = i+1
9. ENDIF
10. ENDWHILE
11. IF (found = 0) THEN
12. PRINT "SEARCH UNSUCCESSFUL"
13. ELSE
14. PRINT "SEARCH SUCCESSFUL: KEY is Present at location" location
15. ENDIF
16. Return (location)
17. Stop

Hint:  same algo

multiple occurance of same elements.

— searching

Algo searching (for multiple occurance of same elements)

Input: KEY is the item to be searched.
output: Index of the KEY or a faliure message.
D.S: Array A[L ... U], where L is lower bound and U is upper bound.

step 1: i = L, found = 0, location = -1 // found = 0 indicates search is not finished and is unsuccessful
count = 0

2: WHILE (i <= U) and (found = 0) DO
3: IF (A[i] = KEY) THEN
4: found = 1
5: location = i
6: count ++,
7: BREAK
8: ELSE
9: i = i+1
10: ENDIF
11: ENDWHILE
12: IF (found = 0) THEN
13: PRINT "SEARCH UNSUCCESSFUL"
14: ELSE
15: PRINT "SEARCH SUCCESSFUL: KEY is Present at location in array" location "for following times"
16: ENDIF
17: FOR (i <= U) and (found = 1) DO
18: PRINT "The location is", location
19: ENDFOR
20: Return (location)
21: STOP

Input: KEY is the item to be searched.
output: Index of the KEY or a faliure message
D.S: Array A[L ... U], where L is lower bound and U is upper bound.

step:
1. i = L, found = 0, location = -1 // found = 0 indicates search is not finished and is unsuccessful
count = 0
2. WHILE (i <= U) and (found = 0) DO
3. IF (A[i] = KEY) THEN
4. found = 1
5. location = i

Input: KEY is the elements to be searched, B is an empty array which's starting index is L, count is a variable that counts the indexes where the elements KEY is present

Output: Indexes of KEY if failure message.

D.S.: 1D Array

Step 1: ~~i=L~~, ~~found=0~~, ~~location=-1~~, count=0

2: WHILE ((i<=U) AND (found=0)) DO

3: IF (A[i] = KEY) THEN

4: found = 1

5:

2: FOR (i=L to U) DO

3: IF (A[i] = KEY) THEN

4: B[j] = i

5: count = count + 1

6: END IF

7: END FOR

8: IF (count > 0) THEN

9: ~~PRINT~~ FOR (j=L' to count-1) DO

10: PRINT (B[j])

11: END FOR

12: ELSE

13: PRINT ("KEY NOT found")

14: END IF

~~15: RETURN (count)~~ ~~B[j]~~

16: STOP

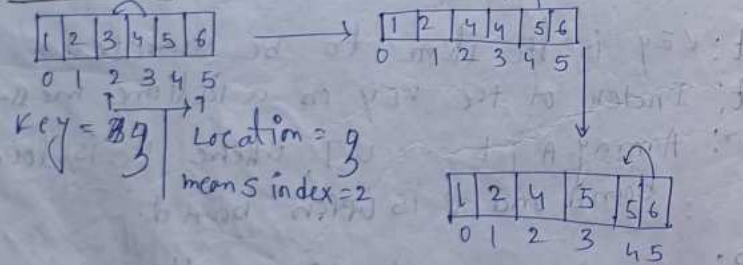
// consider L' is starts from 0th index (if L' starts from 1 then j=L' to count)

6. count ++
7. ELSE
8. i = i+1
9. END IF
10. END WHILE
11. IF(found=0) THEN
12. PRINT "SEARCH UNSUCCESSFUL"
13. ELSE
14. PRINT "SEARCH SUCCESSFUL: KEY is present at location, for following times", location, count
15. ENDIF
16. Return (location)
17. STOP

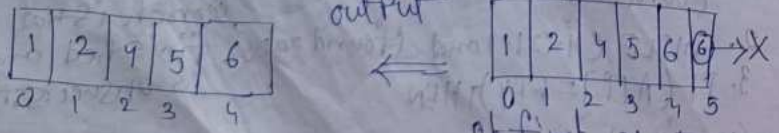
Deletion

25/7/22

overwrite (from where to delet to last[u])



output



At first u = 5 (n)
Now u = 4 (n-1)

A[i] = A[i+1]

If the location (the index of the element to be deleted) is given then there is no need to search that the key is present in array or not as in that location some value must be present and we have to delet it.

Algo Deletion

Input: KEY the element to be deleted
output: Array without KEY
D.S: Array A[L -- U]

step 1: i = search array(A, KEY) // Perform a search operation on A & return the location

2: IF(i = -1) THEN
3. PRINT "KEY not found: no deletion"
4. EXIT
5. ELSE
6. WHILE i<U DO // i = location
7. A[i] = A[i+1]
8. i = i+1
9. ENDWHILE
10. ENDIF
11. A[U] = NULL
12. U = U-1
13. STOP

(in int array we can not use line no (11) in char array we can do that)

[If index is given (from which loc) we have to be delet then start from line no. (6)]

② Deletion, (by locating if location is given) Algorithm.

Algo Deletion (If the location is given)

Input: The element, that Present in the given location to be deleted.

output: Array without the element, which is Present in the given location.

D.S: Array A[L — U]

Step:

1. WHILE (i<U) Do // i= location
2. A[i] = A[i+1]
3. i=i+1
4. ENDWHILE
5. A[U] = NULL
6. U = U - 1
7. STOP

Concatenation (add, Not merge)

A

12	15	17	20	22
0	1	2	3	4

B

11	13	18	19	24
0	1	2	3	4

C

12	15	17	20	22	11	13	18	19	24
0	1	2	3	4	5	6	7	8	9

[output]

Algo concatenate

Input: Two array s A[L_1 — U_1], B[L_2 — U_2]

output: Resultant array C[L — U] where L = L_1, U = (U_1 + (U_2 - L_2 + 1))

D.S: Array

steps:

1. i = L_1, j = L_2 // Initialization of control variables
2. L = L_1, U = U_1 + (U_2 - L_2 + 1) // Initialization of upper & lower bounds

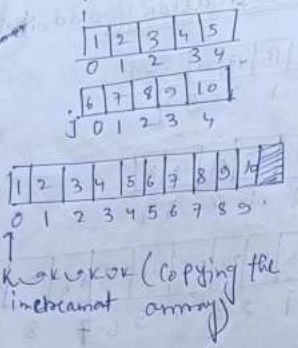
[If we concat 2 array then U = (U_1 + (U_2 - L_2 + 1)) is the formula to calculate the upper bound of the output array]

3. K = L
4. Allocate memory (size (U - L + 1)) // Allocate memory for array C

[have to use calloc or malloc if we don't use a[0] or something else]

5. WHILE (i <= U_1) Do
6. C[K] = A[i]
7. i = i + 1, K = K + 1
8. ENDWHILE
9. WHILE (j <= U_2) Do
10. C[K] = B[j]

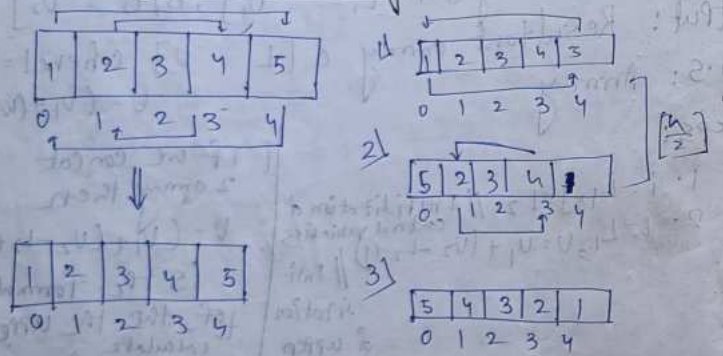
11. j = j+1, k = k+1
12. ENDWHILE
13. STOP



CW

Reverse an array

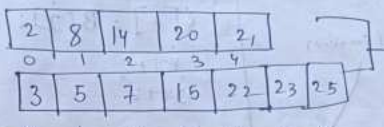
29/7/22



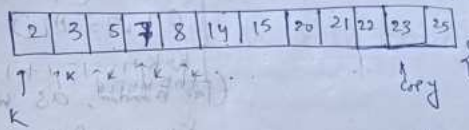
Algo Reverse

Input: Array A[L ... U] and SIZE is the size of array, SIZE = U-L+1
output: Reverse array
D.S : 1D array A[L ... U]
step 1: FOR (i=0 TO [SIZE/2] DO
2: SWAP (A[i], A[SIZE - i - 1])
3: ENDFOR
4: STOP

Array merging



This two array should be sorted individually



Input: Array A[L1 ... U1], B[L2 ... U2]
where L is lower bound and U is upper bound.
output: merged Array C[L ... U]
D.S : 1D array
step :
1. i = L1, j = L2
2. L = L1, U = U = U1 + (U2 - L2 + 1)
3. K = L
4. Allocate memory (size (U-L+1))
5. WHILE (i <= U1) and (j <= U2) DO
6. IF (A[i] < B[j]) THEN
7. C[K] = A[i]
8. K = K+1, i = i+1
9. ENDIF
10. ELSEIF (A[i] > B[j]) THEN
11. C[K] = B[j]
12. K = K+1, j = j+1
13. ENDELSEIF
14. ELSE IF (A[i] = B[j])
15. C[K] = A[i],
16. K = K+1, i = i+1, j = j+1

```
while (i<=U1 && j<=U2)
{ if (A[i] < B[j])
  { C[K] = A[i];
    K++;
    i++;
  }
}
else if (A[i] > B[j])
  { C[K+1] = B[j+1];
  }
if (A[i] == B[j])
  { C[K] = A[i]
    K++;
    i++;
    j++;
  }
}
while (i<=U1)
{ C[K] = A[i];
  K++;
  i++;
```

Algo reverse

Input - Array A with elements

Output - Reversed array

DS - 1D array

Step 1: FOR(i = u To i >= L) Do

2: PRINT (A[i])

3: i = i - 1 → No need

4: END FOR

5: STOP

17. ENDELSEIF
18. ENDWHILE
19. WHILE (j<=U2) DO (from here)
20. C[k] = B[j]
21. k=k+1, j=j+1
22. ENDWHILE
23. STOP

```
while(i<=U1)
{
  C[k] = B[j];
  k++;
  j++;
}
WHILE (i<=U1) DO
  C[k] = A[i]
  K=K+1, i=i+1
END WHILE
```

(For random, as we dont know which one is size is greater).

I) Array union — Array merging — done (previously)
II) Array intersection —

Union = merge sorting
Intersection = common element

only (←) [⊙] pointed one will be in coding.

Input: Array $A[L_1 -- U_1]$, $B[L_2 -- U_2]$
where L is lower bound and U is upper bound.

output: $C[L -- U]$ the array, with common element which is presents in both array.

D.S: 1D array

step:

1. $i = L_1$, $j = L_2$
2. $L = L_1$, $U = U_1 + (U_2 - L_2 + 1)$
3. $K = L$
4. Allocate memory (Size (U - L + 1))
5. WHILE ($i <= U_1$) and ($j <= U_2$) DO
6. IF (A[i] = B[j]) THEN
7. C[k] = A[i]
8. k = k+1, i = i+1, j = j+1
9. ENDIF
10. ENDWHILE
11. STOP

2-D array [matrix]

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}

1st index = Row
2nd index = Column

For 2-D array index i starts from 1

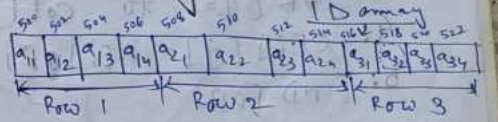
Representation
A [3, 4] ← array
m → no. of rows
n → no. of columns

element
a_{ij} → col no.
↓
row no.

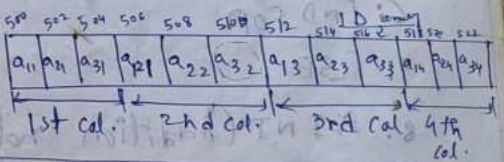
Two representations

memory Representation of 2D Array

① Row major →



② Column major ↓



Row major

Address of a_{ij} = $B \cdot A + [(i-1) \times n + (j-1)] \times w$

Column major

Address of a_{ij} = $B \cdot A + [(j-1) \times m + (i-1)] \times w$

[w = word gap]

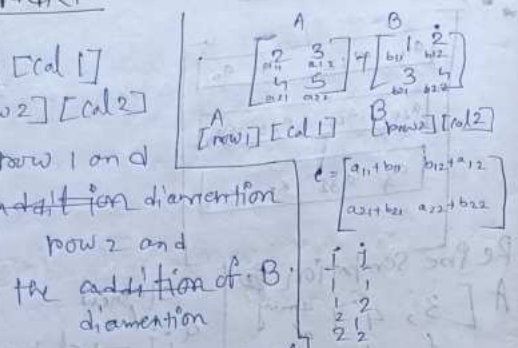
Here i start from 1, j starts from 1 so → (i-1) (j-1); otherwise if i, j starts from 0, then → i, j

no. of elements in row = n
... col = m

11/8/22

Algo matrix Addition

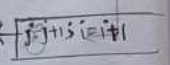
Input: A[row1][col1] and B[row2][col2] where row1 and col1 are ~~addition~~ dimention of A and row2 and col2 are the ~~addition~~ dimention of B.



output: C[row][col] where row & col are the dimention of the resultant matrix

D.S: 2D array.

step 1: IF (row1 = row2) AND (col1 = col2) THEN
2: FOR (i = 1 to row1) DO
3: FOR (j = 1 to col1) DO
4: C[i][j] = A[i][j] + B[i][j]
5: END FOR
6: END FOR
7: ELSE
8: PRINT ("Addition not Possible")
9: ENDIF
10: STOP



Algo mat Transpose

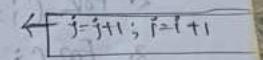
Input: A[row1][col1] where row1 and col1 are dimention of A.

output: C[row][col] where row and col are the dimention of the ~~resultant~~ transpose matrix

D.S - 2D Array

step:

1: FOR (i = 1 to row1) DO
2: FOR (j = 1 to col1) DO
3: CB[j][i] = A[i][j]
4: END FOR
5: END FOR
6: STOP



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} \\ 1 & 4 \\ b_{21} 2 & 5 b_{22} \\ 3 & 6 \\ b_{31} & b_{32} \end{bmatrix}$$

i	j
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

Algo mat subtraction

Input: A[row1][col1] and B[row2][col2]
where row1 and col1 are the dimention of A and row2 and col2 are the dimention of B.

output: C[row][col] where row and col are the dimention of the resultant matrix.

D.S: 2D Array

step:

1. IF(row1=row2) AND (col1=col2) THEN
2. FOR(i=1 to row1) DO
3. FOR(j= to ~~row~~ col1) DO
4. C[i][j] = A[i][j] - B[i][j]
5. END FOR
6. END FOR
7. ELSE
8. PRINT("Subtraction not possible")
9. ENDIF
10. STOP

$$A\begin{bmatrix} a_{11} & a_{12} \\ 2 & 3 \\ a_{21} & a_{22} \\ 4 & 5 \end{bmatrix} - B\begin{bmatrix} b_{11} & b_{12} \\ 1 & 2 \\ 3 & 4 \\ b_{21} & a_{22} \end{bmatrix} = C\begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

i j
1 1
1 2
2 1
2 2

matrix multiplication

$$A\begin{bmatrix} a_{11} & a_{12} \\ 2 & 3 \\ a_{21} & a_{22} \\ 1 & 2 \end{bmatrix} \quad B\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ 4 & 5 & 1 \\ b_{21} & b_{22} & b_{23} \\ 2 & 1 & 2 \end{bmatrix}$$

2×2 col1 row2 2×3

i×k k×j

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ 0+8+6 & 0+5+3 & 0+1+6 \\ c_{21} & c_{22} & c_{23} \\ 8+4 & 0+5+5 & 0+1+4 \end{bmatrix} = \begin{bmatrix} 14 & 13 & 8 \\ 8 & 10 & 5 \end{bmatrix}$$

2×3 row1 × col2

i×j

i j k
1 1 1
1 1 2
1 2 1
1 2 2
1 3 1
1 3 2
2 1 1
2 1 2
2 2 1
2 2 2
2 3 1
2 3 2

Algo mat multiplication

Input: A[row1][col1] & B[row2][col2]

output: C[row1][col2] where row & col are the dimention of resultant matrix

D.S: 2-D Array

step 1: IF(col1 = row2) THEN
2: FOR(i=1 to row1) DO
3: FOR(j=1 to col2) DO
4: C[i][j]=0
5: FOR(k=1 to col1) DO — [this condition is for square mat] col2/row2
6: C[i][j] = C[i][j] + (A[i][k] * B[k][j])

```
7: END FOR
8: END FOR
9: ELSE
10: PRINT("multiplication Not possible")
11: ENDIF
12: STOP
```

Sparse matrix

A matrix which have most of it elements ($\frac{2}{3}$) is zero. more than 2/3 elements are zero.

$$\begin{bmatrix} -1 & 0 & 0 & 0 & 2 \\ 4 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow \text{Sparse matrix.}$$

- It is a 2-D array where majority of the elements (more than 2/3 of the elements) have a value of zero.

$$\begin{bmatrix} -1 & 0 & 0 & 0 & 2 \\ 4 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- 3-tuple (row no., col no., value)

$$A = \begin{bmatrix} \text{row} & \text{col} & \text{value} \\ 1 & 1 & -1 \\ 1 & 5 & 2 \\ 2 & 1 & 4 \\ 2 & 3 & 5 \\ 3 & 5 & 1 \\ 4 & 1 & 7 \end{bmatrix}$$

A[1--4, 1--3] [tuple-info/dimension]

[Row no. = No. of non zero elements]

[col No. = No. of col non zero elements]

- Sparse matrix

Sparse matrix → lower triangle matrix, upper triangle matrix

lower triangle matrix

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

[Diagonal → 0 ~~below~~ upper]

(Row major, Col major)

upper triangle matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix}$$

[Diagonal → 0 lower]

(Col major, Row major)

Row major:

500	502	504	506	508	510
a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}
1		2		3	

Address of a_{ij} = Total No. of element upto a_{ij}

= No. of elements in first (i-1) rows
+ No. of elements upto jth column in ith row

$= B.A + \left([1+2+3 \cdots + (i-1)] + (j-1)\right) \times w$

$= \boxed{B.A + \left[\frac{i(i-1)}{2} + (j-1)\right] \times w}$ **

row major formula for lower triangular matrix.

*Col major of upper triangular matrix

Address of a[i,j] =

lower's row m. = upper's col m.

a_{11}	a_{12}	a_{22}	a_{13}	a_{23}	a_{33}
col 1	col 2		col 3		

Address of a_{ij} = total number of element upto a_{ij}

No. of elements in first (j-1) col + No. of elements upto ith column in the jth col

$= B.A + \left([1+2+3 \cdots + (j-1)] + (i-1)\right) \times w$

$= \boxed{B.A + \left[\frac{j(j-1)}{2} + (i-1)\right] \times w}$ **

col major formula for upper triangular matrix

*** Col major representation of low triangular matrix

Address of a_{ij} = total no. of elements upto a_{ij}

= total no. of elements in the first (j-1) col + no. of elements upto ith row in jth col.

$= B.A + \left[[n + (n-1) + (n-2) \cdots (n-j+1)] + (i-j)\right] \times w$

$= \boxed{B.A + \left[n \times (j-1) - \left\{\frac{j(j-1)}{2}\right\} + i\right] \times w}$ **

Column major formula for lower triangular matrix

a_{11}	a_{21}	a_{31}	a_{22}	a_{32}	a_{33}

skip 2, skip 1, skip 0

a_{11} 0 0
a_{21} a_{22} 0
a_{31} a_{32} a_{33}

row i = 3
col j = 2

Lower's col m. = upper's row m.

*** Row major representation of low triangular matrix

Address of a_{ij} = total no. of elements upto a_{ij}

= total no. of elements in first (i-1) row + No. of elements upto jth col in ith row.

$= B.A + \left[[n + (n-1) + (n-2) \cdots (n-i+1)] + (j-i)\right] \times w$

$= \boxed{B.A + \left[n \times (i-1) - \left\{\frac{i(i-1)}{2}\right\} + j\right] \times w}$ **

Address of a_{ij} = total no. of elements upto a_{ij}

= total no. of elements in the first $(j-1)$ col + no. of elements upto ith row in jth col.

$i=3, j=2, s=1$
$i=3, j=1, s=0$
$i=3, j=3, s=2$
} $(j-1)$

$i=3, j=1, s=2$
$i=3, j=2, s=1$
$i=3, j=3, s=0$
} $(i-j)$

$(i-j)$ Sum series

2
1
0

$(j-1)$ Sum

0
1
2

dress of

= total no. of elements in
no. of elements upto jth col in

B.A + [[n + (n-1) + (n-2) - (n-i+1)] + (i-1)]

Row major for lower triangular matrix:

Linked list

Pointers ① contigious memory allocation 5/8/22

int a[100]; ② size of spaces

① 10 is declared this cannot be changed

static memory allocation } pervedy

dynamic memory allocation

(we can allocate memory as when we required it) (No need of contigious memory)

Dynamic memory Allocation → linked list (Continious / linear D.S)

[NULL = Last element of linked list]

node → 2 parts → data Path, link Path

400

650

300

200

100

NULL

Linked Path = null

Header → 1st node

node

Have 2 data Path, ① null data Path which have no data, ② linked Path } ll

[must remember the address of the header, then we can know the all info about whole linked list]

structure – containing different data type

12 | 650 — 400

structure

struct node

{

int data;

struct node * link;

}

structure Pointer

data type

Pointer

name of pointer

int a = 3 (100 | 3 a | 150 j)

int *j; // j is an integer pointer.

j = &a;

printf("%d", a); 3

printf("%d", *j); 3

printf("%d", j); 100

printf("%d", &a); 100

printf("%d", *(&a)); 100

1) Single linked list (Previous one)

2) Double "

3) Circular "

①

malloc → It is a func (for memory allocation) (std lib)

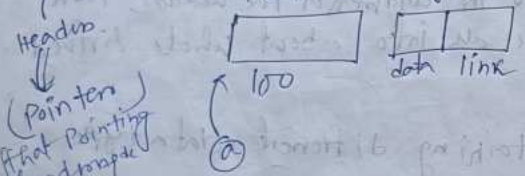
[It contains a garbage value]

a = sizeof(int)

a = 2

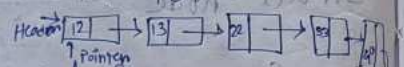
a = (struct node*) malloc (sizeof(struct node))

(int *) - stores any [dynamically taking an array]



Algo Traversal SL

Input: HEADER is the pointer to the header node



Output: Print elements of the list.

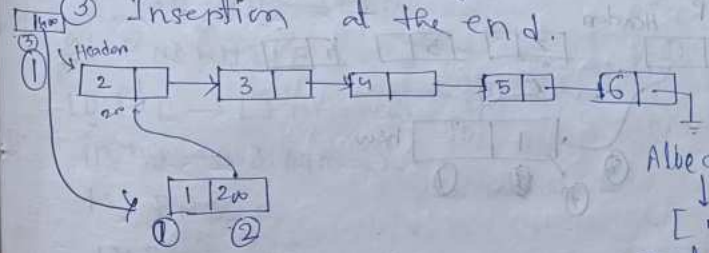
D.S: Single linked list : 12 13 22 33 40

step1: ptr = HEADER → LINK
2: WHILE (ptr ≠ NULL) DO
3: PRINT (ptr → DATA)
4: ptr = ptr → LINK
5: ENDWHILE
6: STOP

12/8/22

Insertion in a single linked list

① Insertion at the beginning
② Insertion at any position
③ Insertion at the end.



Allocate (in algo) memory ↓ in C [malloc]

Last link with others whole node, then link that node with header]

Input: HEADER is a pointer to the header node and x is the data to be inserted.

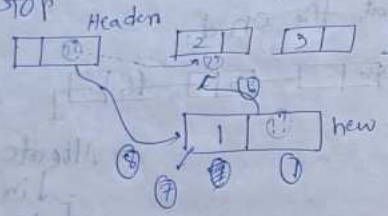
output: A single linked list with newly inserted node at the begining of the linked list.

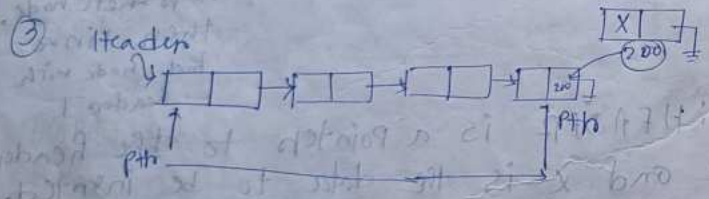
D.S: Single linked list

step1: new = GETNODE (NODE) // GETNODE is a function to allocate a new node

2: If (new = NULL) THEN // memory bank returns NULL

3: PRINT "memory full: no insertion"
4: EXIT
5: ELSE
6: new→LINK = HEADER → LINK
7: new → DATA = x
8: HEADER → LINK = new
9: ENDIF
10: STOP





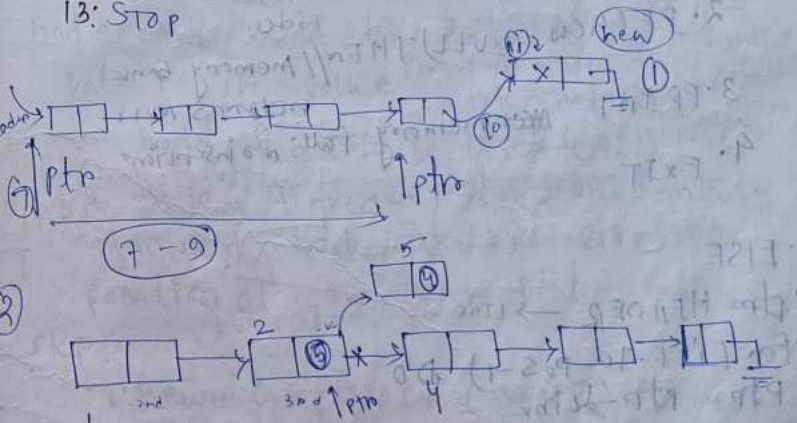
Algo. Insertion at the end - SL

Input: HEADER is a pointer to the header node and x is the data to be inserted.

O/P: A single linked list with newly inserted node at the end of the linked list.

D.S - single linked list

step 1: new = GETNODE (NODE)
2: IF (new = NULL) THEN
3: PRINT "memory full! No insertion"
4: EXIT
5: ELSE
6: ptr = HEADER → LINK
7: WHILE (ptr → LINK ≠ NULL) DO
8: ptr = ptr → LINK
9: ENDWHILE
10: ptr → LINK = new
11: new → DATA = x
12: ENDIF
13: STOP



Data = x
Position = 3rd

Algo Insertion at any position - S2

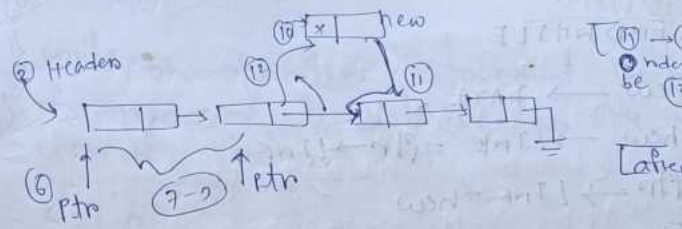
Input; HEADER is a pointer to the header node and x is the data to be inserted, pos is the position to where node has to be inserted where data is to be inserted.

O/P: A single linked list with newly inserted node at the specified position of the link list.

D.S: single linked list

Step 1: new = GETNODE (NODE) // GETNODE is a function to allocate a new node

2. IF (new = NULL) THEN // memory bank returns NULL
3. PRINT "memory full: No insertion"
4. EXIT
5. ELSE
6. ptr = HEADER → LINK
7. FOR (i = 1 TO pos-1) DO
8. ptr = ptr → Link
9. END FOR
10. new → DATA = x
11. new → LINK = ptr → LINK
12. ptr → LINK = new
13. ENDIF
14. STOP



HW

(1) Insertion after a given data.
[First search → data (by searching) then do insertion after the node where x(data) is present]

Input: HEADER is a pointer to the header node and x is the data to be inserted, val is the value present in a node after which the new node has to be inserted.

O/p: A single linked list with newly inserted node at the specified position of the link list.

Step:

1: new = GETNODE (NODE)
2: IF (new = NULL) THEN
3: PRINT ("memory full; NO insertion"
4: EXIT
5: ELSE
6: ptr = HEADER
7: WHILE (ptr → data != val) DO
8: ptr = ptr → LINK.

5: ENDWHILE

10: new → DATA = X

11: new → LINK = ptr → LINK

12: ptr → LINK = new

13: ENDIF

14: ~~END~~ STOP

(keeping 'if element not found condition')

Algo: insertion — random —

Input, output, DS will be same like previous one.

Step 1: new = GET node(NODE)

2: IF (new = NULL) THEN

3: PRINT ("memory full, NO insertion")

4: EXIT

5: ELSE

6: ptr = HEADER → LINK

7: WHILE (ptr ≠ NULL) DO

8: IF (ptr → DATA ≠ ~~NULL~~ VAL) THEN

9: ptr = ptr → LINK

10: ~~END~~ ELSE

11: new → DATA = X

12: new → LINK = ptr → LINK

13: ptr → LINK = new

14: ENDIF

15: ENDWHILE

16: ENDIF

17: STOP

WHILE (ptr ≠ NULL) AND (found = 0) DO

IF (ptr → DATA == VAL) THEN

→ found = 1

→ BREAK

→ ELSE

→ ptr = ptr → LINK

ENDIF

ENDWHILE

IF (found = 0) PRINT ("DATA not found")

→ ELSE

from (11) → (17)

clw

22/8/19

1) write an algorithm a) to count the number of nodes in the list b) find the minimum element from the list.

a) Input: HEADER is the pointer to the header node, COUNT is counting the no. of nodes, it's a variable.

output: ~~Print elem~~

Print ~~count~~ the numbers of nodes in the list.

D.S: Single linked list

Step 1: ptr = HEADER → LINK

→ COUNT = 0

3: WHILE (ptr ≠ NULL) DO

3: ~~COUNT = COUNT + 1; PRINT (ptr → DATA)~~

4: COUNT = COUNT + 1

5: ptr = ptr → LINK

6: ENDWHILE

7: ~~STOP~~ PRINT (COUNT)

8: STOP

b) Input: HEADER is the pointer to the header node, ~~COUNT~~ MIN is a pointer variable ~~that holding counting~~ the address of the minimum elements of the list.

output: ~~Find~~ print the minimum element of the list.

D.S: single linked list.

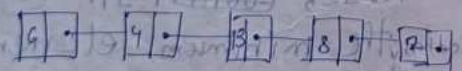
1: new = GETNODE(NODE)
2: IF (new = NULL) THEN
3: PRINT("memory full: no insertions")
4: EXIT
5: ELSE
6: ~~WHILE~~ ptr = HEADER → LINK
7: WHILE ((ptr ≠ NULL) AND (found = 0)) DO
8: IF (ptr → DATA = VAL) THEN
9: found = 1
10: BREAK
11: ELSE
12: ptr = ptr → LINK
13: ENDIF
14: ENDWHILE
15: IF (found = 0) THEN
16: PRINT("VAL not found")
17: ELSE
18: new → LINK = ptr → LINK
19: new → DATA = x
20: ptr → LINK = new
21: ENDIF
22: ENDIF
23: STOP

WHILE (ptr ≠ NULL) DO
IF (ptr → DATA ≠ VAL) THEN
ptr = ptr → LINK
ELSE
from (18) -- (23)

(m) WHILE (ptr → DATA ≠ VAL) ~~THEN~~ DO
ptr = ptr → LINK
ENDWHILE
from (18) -- (23)

Step 1: i = L1, j = ~~L1~~ i+1
2: ptr = HEADER → LINK
3: IF (LINK ≠ NULL) ~~DO~~ THEN
4: PRINT ("there are no searching of minimum element of the list, so that element belongs to the list is printing")
5: PRINT (HEADER → Data = X)
6: ENDIF
7: EXIT
8: ELSE

step: ptr = HEADER → LINK
1. min = ~~HEADER~~ → Data = X → ptr. ~~da~~ → LINK
2. ~~FOR~~ WHILE (i < HEADER → LINK to ptr ≠ NULL)
3. IF (ptr → Data < min) DO
4. min = ptr → Data = X (fresh ptr.)
5. END IF
→ ptr = ptr → link
6. PRINT (MIN) → ENDWHILE
7. STOP



Step 1. ptr = HEADER → LINK
2. min = ptr → Data = X
3. ~~WHILE~~ ptr = ptr → LINK
4. WHILE (ptr ≠ NULL) DO
5. IF (ptr → Data < min) THEN
6. min = ptr → Data = X,
7. END IF
8. ptr = ptr → LINK
9. END WHILE
10. PRINT (min)
11. STOP

(c) Find the minimum element in an array.

Input: An array with A elements. L is lower bound and U is upper bound.

output: Print the min element of array

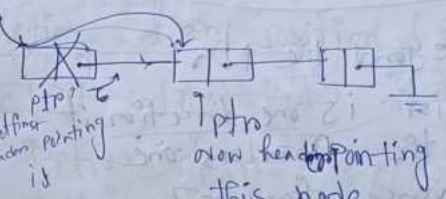
D:S :- 1D array

Step1: ~~i=L~~ ~~(Starting from lower bound)~~

1. MIN = A[L] to U
   ~~WHILE~~ FOR (i <= ~~MIN~~) DO
2. IF (A(i) <= MIN) ~~THEN~~ THEN
3. MIN = A[i]
4. ~~ENDIF~~ END IF
   ~~i=i+1~~
5. END ~~WHILE~~ FOR
6. PRINT (MIN)
7. STOP

Deletion from a Single linked list

① Begining
② Any Position
③ END

① 

~~Algo~~ Delete from the begining

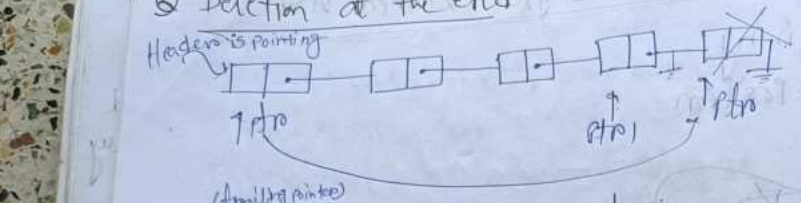
Input: HEADER is a pointer to the header node

output: A single linked list after eleminating the node at the first

D.S : single linked list

Step 1: ptr = HEADER → LINK
2: IF (ptr = NULL) THEN
3: PRINT ('Deletion not Possible')
4: EXIT
5: ~~ENDIF~~ ELSE
6: HEADER → LINK = ptr → LINK
7: RETURN (ptr) // Return Node (L) is a function to return the deleted node to memory bank.
8: ENDIF
9: STOP

② Deletion at the end



(trailing pointer)
ptr1 = walking beyond the ptr

As Single linked list is one direction it can not point the Previous one it'll only traverse forward.

Algo Delete_end

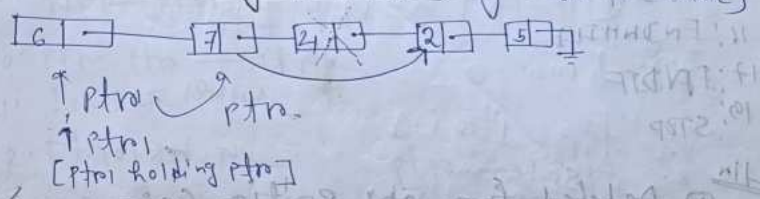
Input: HEADER is a pointer to the header node

output: A single link list after eleminating the node at end.

D.S - Single linked list

```
Step1: ptr = HEADER → LINK
2: IF (ptr = NULL) THEN
3: PRINT ('Deletion Not Possible')
4: EXIT
5: ELSE
6: WHILE( ptr → LINK ≠ NULL) DO
7: ptr1 = ptr  //trailing
8: ptr = ptr → LINK
9: END WHILE
10: ptr1 → LINK = NULL
11: RETURN (ptr)
12: ENDIF
13. STOP
```

③ Deletion at any position (By a given value)



Algo Delete_value from any position

Input: HEADER is a pointer to the header and KEY is the element to be deleted.

Output: A single linked list after eleminating the node corresponding to KEY.

D.S: Single Linked List

```
step 1: ptr = HEADER → LINK
2: ptr1 = ptr
2: IF ( ptr = NULL) THEN
3: PRINT ('Deletion not Possible')
4: EXIT
5: ELSE
6: WHILE ( ptr ≠ NULL) DO
7: IF ( ptr → DATA ≠ KEY) THEN
8: ptr1 = ptr
9: ptr = ptr → LINK
```

10: ENDIF
11: ELSE
12: ptr1 → LINK = ptr → LINK
13: RETURN NODE (ptr)
14: EXIT
15: ENDIF
16: ENDWHILE
17: ENDIF
19: STOP

Hin

① Deleted from any position (given position):

② Searching in a single list [Algo delete by value]

① Ddelete – by position

Algo

Input: HEADER is a pointer to the header node, pos is the position of the node after which the node has to be deleted.

output - A single linked list after eleminating the node from specified position.

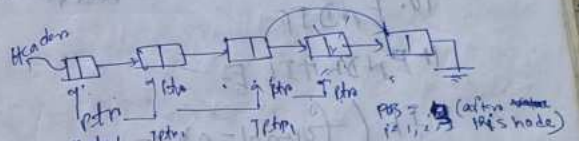
D·S - Single linked list.

Step:

1. ptr = HEADER → LINK
2. ptr1 = ptr
3. If (ptr = NULL) THEN

9. PRINT (Deletion not possible)
5. EXIT
6. ELSE
7. WHILE (ptr ≠ NULL) DO i<=pos ++
8. ~~FOR (i = 1 to pos-1) DO~~
9. ptr1 = ptr
   i = i + 1
10. ptr = ptr → LINK
11. ENDFOR WHILE
12. ptr1 → LINK = ptr → LINK
    → RETURN (ptr)
13. ENDWHILE
14. ENDIF
15. STOP

pos which has to be deleted = i < pos
after which node
= i <= pos



② Algo searching

Input: HEADER is a pointer to the header and KEY is the element to be searched.

output: The position/location of the key (search value)

D·S - Single linked list.

Step:

~~1. found = 0; location~~ pos // found indicates search is not finish and it is unsuccessful

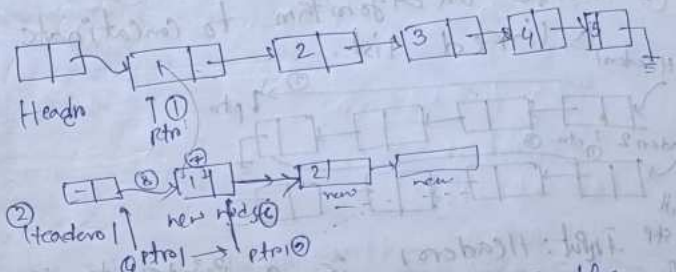
~~IF WHILE (i ≠ NULL) and (found = 0)~~

~~IF (~~

1. i=0; found=0, POS. //found and indicates search is not finished and it is unsuccessful
2. ptr = HEADER → LINK
3. WHILE (ptr ≠ NULL and (found=0) DO
4. IF (ptr → Data = KEY) THEN
5. found = 1
6. POS = i
7. BREAK
8. ELSE i = i+1
9. ptr = ptr → LINK
10. ENDIF
11. ENDWHILE
12. IF (found=0) THEN
13. PRINT (Search unsuccessful variable not found)
14. ELSE
15. PRINT (search successful, KEY found at location, POS)
16. ENDIF
17. RETURN (POS)
18. STOP

26/8/22

① write an algorithm to copy a linked list into a new list.



Input: HEADER is the pointer to the header node

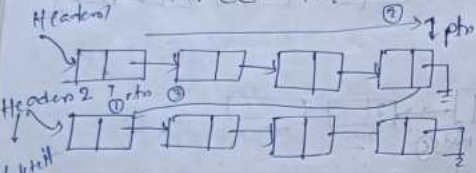
output: Header1 is a pointer to a duplicate list.

DS: single linked list

Step 1: ptr = Header → Link
2: Header1 = GETNODE (NODE)
3: Header1 → Data = NULL
4: ptr1 = Header1
5: WHILE (ptr ≠ NULL) DO
6: new = GETNODE (NODE)
7: new → DATA = ptr → DATA
8: ptr1 → LINK = new
9: ptr1 = new
10: ptr = ptr → LINK
11: ENDWHILE

12: Return(HEADER1)
13: STOP

② write an algorithm to concatinate 2 linked list.



Input: Header1 is a pointer to the header node of 1st linked list, Header 2 is ~~output:~~ a pointer to the header node of 2nd linked list.

output: A single linked list: merged / concatinated

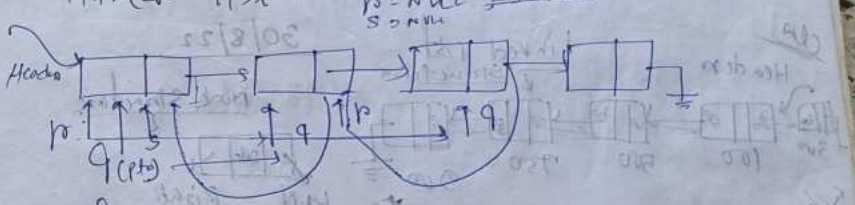
DS: A single linked list

Step 1; ptr = Header1 → LINK
2: ~~WHILE~~ IF(ptr ≠ NULL)
3: ~~ptr = ptr → LINK~~
3: ptr = Header2 → LINK
3: ptr = ptr → LINK
4: ELSE
5: ptr → LINK = Header2 → LINK
6: ~~ptr =~~ END IF
7: return (Header1)
8: Header = Header1 //Header1 is the header node of merged linked list
9: STOP

1. ptr = Header1 → Link
2. WHILE (ptr ≠ NULL) → LINK
3. ptr = ptr → Link
4. ENDWHILE
5. ptr → Link = Header2 → Link
6. return (Header1)
7. Header1 = Header1
8. STOP

(5) ptr = 5-
(6) HEADER = HEADER1
(7) RETURN (HEADER)
(8) STOP

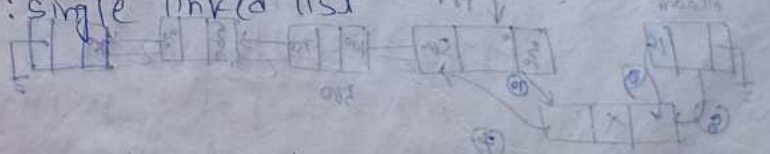
③ write an algorithm to reverse a linked list.



Input: Header is a pointer to ~~the~~ header node

output: ~~Header1~~ is a pointer to ~~a~~ duplicate list the reverse of the link list

D.S: single linked list



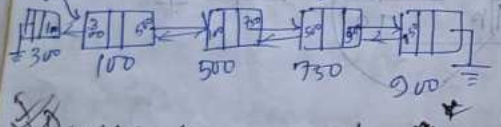
Step 1: q(ptr) = Header → LINK
2: p = NULL, S = NULL
3: r = q
4: p → LINK = q → LINK
5: q = q → LINK

1: q=Header → LINK
2: p=NULL, s=NULL
3: WHILE (q ≠ NULL) DO
4: s=p
5: p=q
6: q = q → LINK
7: p → LINK = s
8: ENDWHILE
9: HEADER → LINK = p
10: STOP

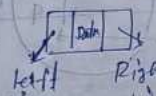
- Comparison between array and linked list
- what are the differences of single linked list and how it can be removed

30/8/22

Linked list Structure

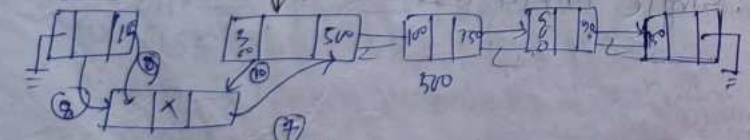


Node structure



Doubly linked list

Insertion



Algo Insertion at the begining in doubly linked list

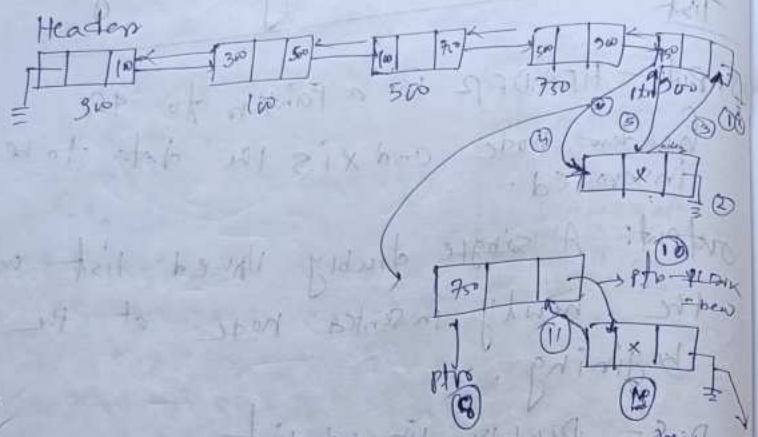
Input – HEADER is a pointer to the header node and x is the data to be inserted.

output: A single doubly linked list with the newly inserted node at the begining.

DS - Doubly linked list

Step: 1. new = GETNODE (NODE) // GETNODE is a function to allocate a new node
2. IF (new = NULL) THEN // memory bank returns null
3. PRINT "memory full: insertion not possible"
4. EXIT
5. ELSE
6. ptr = HEADER → RLINK
7. new → RLINK = ptr
8. new → LLINK = HEADER
9. HEADER → RLINK = new
10. ptr → LLINK = new
11. new → data = x
12. ENDIF
13. STOP

Insertion at the end of a doubly linked list



Algo Insertion at the end of a doubly linked list

Input — HEADER is a pointer to the header node.

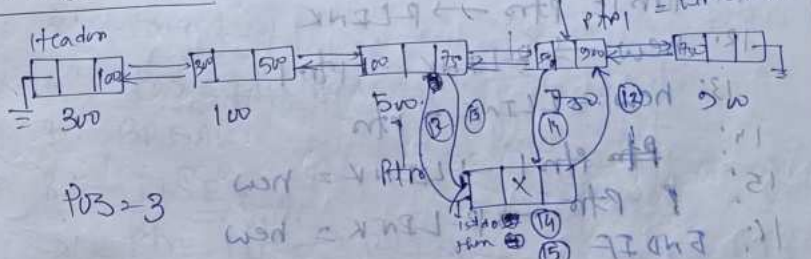
output; A doubly linked list with newly inserted node at the end.

D.S: Doubly linked list.

Step: 1. new = GETNODE(NODE)
2. IF (new = NULL) THEN
3. PRINT memory full, the insertion is not possible.
4: EXIT
5: ELSE
6: ptr = HEADER → RLINK
7: WHILE (ptr → RLINK ≠ NULL) DO
8: ptr = ptr → RLINK
9: ENDWHILE
10: ptr → RLINK = new
11: new → LLINK = ptr
12: new → data = x
13: ENDIF
14: STOP

→ // new → RLINK = NULL

Algo. Insertion at any position of a doubly linked list



Input; HEADER is a pointer to the header node and x is the data to be inserted, pos is the position where data is to be inserted.

Output: A doubly linked list with the newly inserted node at specific position of the link list.

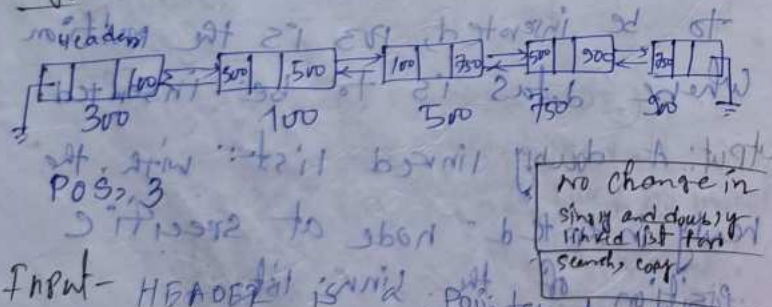
D.S: Doubly linked list.

Step 1; new = GETNODE (NODE)
2. IF (new = NULL) THEN

3: PRINT "memory full! insertion not possible".
4: EXIT
5: ELSE
6: Ptr = HEADER → RLINK
7: FOR (I = 1 to POS - 1) DO
8: Ptr = Ptr → RLINK
9: ENDFOR
10: new → DATA = x
11: Ptr1 = Ptr → RLINK
12: new → RLINK = Ptr1
13: new → LLINK = Ptr
14: Ptr1 → LLINK = new
15: Ptr → RLINK = new
16: ENDIF
17: STOP

Searching in doubly linked list

Algo



Input- HEADER is the pointer to the header node to search key in the element to be searched

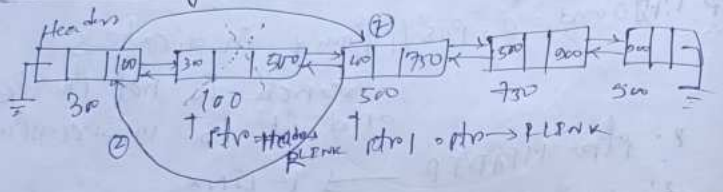
output; the position/location of the KEY

DS: A doubly linked list

Step 1: found = 0, pos // found indicates search is not finished and it is unsuccessful

2: Ptr = HEADER → RLINK
3: WHILE (Ptr ≠ NULL) and (found = 0) DO
4: IF (Ptr → DATA = KEY) THEN
5: found = 1
6: POS = i
7: BREAK
8: ELSE (found = 0, i = i+1)
9: Ptr = Ptr → RLINK
10: ENDIF
11: ENDWHILE
12: IF (found = 0) THEN
13: PRINT (Search unsuccessful! variable not found)
14: ELSE
15: PRINT (Search successful, KEY found at location POS)
16: ENDIF
17: RETURN (POS) PRINT RETURN
18: STOP

Algo deletion from the begining from the doubly linked list (DLL)



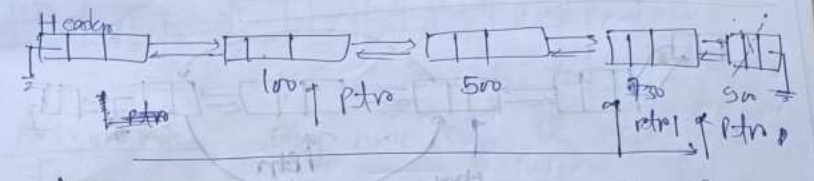
Input: HEADER is a pointer to the header node.

output: A doubly linked list after eleminating the node from the first / begining.

D.S: Doubly linked list

STEP 1: ptr = HEADER → RLINK
2: IF (ptr = NULL) THEN
3: PRINT "Deletion is not possible"
4: EXIT
5: ELSE
6: ptr1 = ptr → RLINK
7: HEADER → RLINK = ptr1
8: ptr1 → LLINK = HEADER
9: RETURN (ptr) // return node (is a function to return the deleted node to the memory bank
10: ENDIF
10: STOP

Algo deletion at the end in the doubly linked list



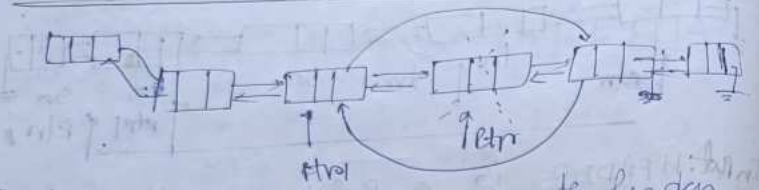
Input: HEADER is a pointer to the headers node.

output: A doubly linked list after eleminating the node from the end

D.S: Doubly linked list

STEP 1: ptr = HEADER → RLINK
2: IF (ptr = NULL) THEN
3: PRINT ("Deletion not possible")
4: EXIT
5: ELSE
6: WHILE (ptr → RLINK ≠ NULL) DO
7: ptr1 = ptr // trailing
8: ptr = ptr → RLINK
9: ENDWHILE
10: ptr
11: ptr1 → RLINK = NULL
12: RETURN (ptr)
13: ENDIF
14: STOP

Algo ***
Deletion at any position in doubly linked list

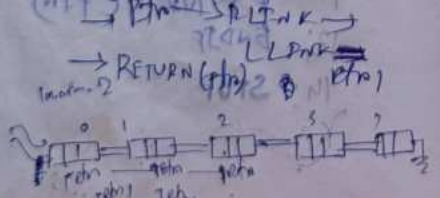


Input: HEADER is a pointer to the header node, pos is the position after which the node has to be deleted

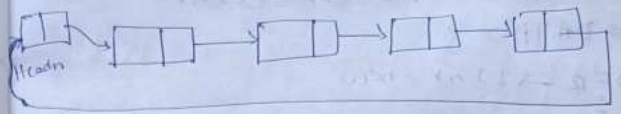
output: A doubly linked list after eliminating the node from a specific position

D.S: A doubly linked list

Step 1: ptr = HEADER → RLINK
2: IF (ptr = NULL) THEN
3: PRINT "Deletion not possible"
4: EXIT
5: ELSE
6: ~~WHILE (ptr ≠ NULL) DO~~
6: WHILE (i<=POS and ptr≠NULL) DO
7: ptr1 = ptr
8: i = i+1
9: ptr = ptr → RLINK
10: END WHILE
11: ptr1 → RLINK = ptr → RLINK
12: FNDIOP
13: Stop



Circular linked list



Advantages - Any node can be accessable from any other node.
② NULL pointer problem
③ Easy implementation of some problems.

Disadvantages of circular ~~loop~~ link list
① Infinite loop problem.

Algo
① Insertion at begining in CLL (or SLL)

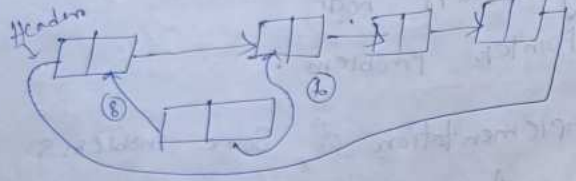
Input: HEADER is a pointer to the header node and X is the data to be inserted

output - A circular linked list with newly inserted node at the begining of the linked list.

DS- Circular link list

Step 1: new = GETNODE (NODE) // GETNODE is a function to allocate a new node
2: IF (new = NULL) THEN // memory bank returns NULL
3: PRINT "memory full! No insertion"
4: EXIT

5: ELSE
6: new → LINK = HEADER → LINK
7: new → DATA = X
8: HEADER → LINK = new
9: END IF
10: STOP



(2) Algo Insertion at any position

Input: HEADER is a pointer to the header node and x is the data to be inserted, pos is the position at where the node that contain x has to be inserted

O/P: A circular linked list with newly inserted node at the specific position

DS: Circular link list

STEP 1: new = GETNODE (NODE) // GETNODE is a func to allocate a new node.
2: IF (new = NULL) THEN
3: PRINT "memory full: Insertion not possible"
4: EXIT
5: ELSE
6: ptr = HEADER → LINK
7: FOR (i=1 TO POS-1) DO
8: ptr = ptr → LINK
9: ENDFOR
10: new → DATA = X
11: new → LINK = ptr → LINK
12: ptr → LINK = new
13: ENDIF
14: STOP

At the end:

(3) Algo Insertion at the end of CLL

Input: HEADER is a pointer to the header node and x is data to be inserted.

O/P: A circular linked list with newly inserted node at the end.

D.S- circular link list.

STEP 1: new = GETNODE (NODE)
2: If (new = NULL) THEN
3: PRINT "memory full. No insertion"
4: EXIT
5: ELSE
6: Ptr = Header → LINK

7: WHILE (ptr → LINK ≠ HEADER) DO
8: ptr = ptr → LINK
9: END WHILE
10: new → LINK = Header
11: ptr → LINK = new
12: new → data = x
13: END IF
14: STOP

If we hold header 1st then we have the whole node

(4) Deletion at the begining of circular LL

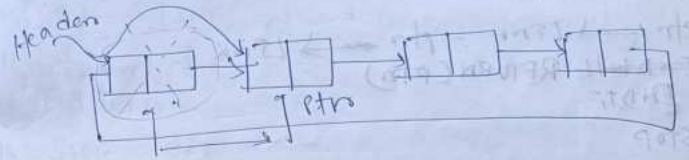
Algo

Input: HEADER is a pointer to the header node.

output: A circular linked list after eleminating the node at the first.

D.S- Circular link list.

Step1: ptr = HEADER → LINK
2: If (ptr = NULL) THEN
3: PRINT ("Deletion not possible")
4: EXIT
5: ELSE
6: HEADER → LINK = ptr → LINK
7: Return (ptr) // Return node() is a func to return the deleted node to memory bank.
8: END IF
9: STOP



(5) Algo deletion at any position of CLL

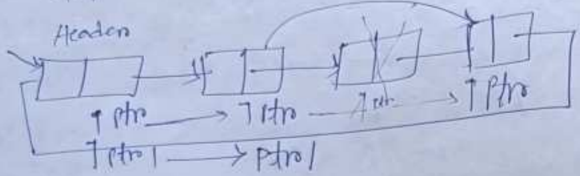
Input: HEADER is a pointer to the header node of circular

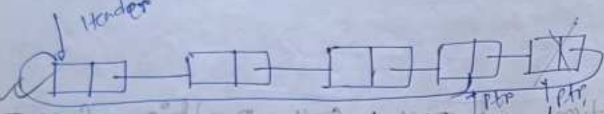
output: A single linked list after eleminating the node from specified position

D.S- Circular link list

Step1: ptr = HEADER → LINK
2: ptr1 = ptr
3: IF (ptr = NULL)
4: PRINTF ("Deletion not possible");
5: EXIT
6: ELSE
7: WHILE (i <POS AND ptr ≠ NULL) DO
8: ptr1 = ptr
9: i = i+1
10: ptr = ptr → LINK

11: END WHILE
12: ptr1 → LINK = ptr → LINK
~~13: ENDWHILE~~ RETURN(ptr)
13: ENDIF
14: STOP





⑥ Algo deletion at the end of a circular LL

Input: Header is a pointer to the header node

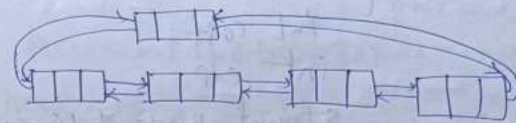
O/P: A circular LL after the deleting the node at the end.

D.S - Circular LL

STEP 1: ptr = Header → LINK
2: IF (ptr = NULL) THEN
3: PRINT ("Deletion NOT POSSIBLE")
4: EXIT
5: ELSE
6: WHILE(ptr → LINK ≠ Header) DO
7: ptr1 = ptr // trailing
8: ptr = ptr → LINK
9: END WHILE

10: ptr1 → LINK = Header
11: Return (ptr)
12: ENDIF
13: STOP

Circular Data Structure



(Circular doubly link list)

1) Advantages of link list
Dynamic memory allocation - write something about this
Non - contiguous

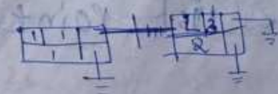
Apply of LL

1) Sparse matrix manipulation } just concept algo

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 \end{bmatrix}$$

node structure for a sparse matrix -

struct Sparse node
{



}

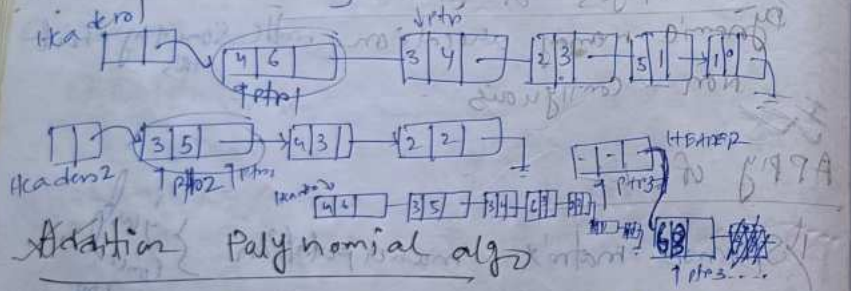
② Polynomial Representation

exponent

$3x^4 + 2x^3 + 5x + 1 \quad + 3x^5 + 4x^3 + 2x^2$

Coefficient $= 4x^6 + 3x^5 + 3x^4 + 6x^3 + 2x^2 + 3x + 1$

```
struct node
{
    int coff;
    int exp
    struct node * link;
};
```



Addition Polynomial algo

Input: A and B are two Polynomials represented as single linked list with HEADER1 and HEADER2 as headers of A and B respectively.

O/p: Polynomial C as the sum of A and B represented as a single linked list pointed to by HEADER.

D.S - single linked list

```
Step 1: ptr1 = HEADER1 -> LINK
2:  ptr2 = HEADER2 -> LINK
3:  WHILE ((ptr1 -> LINK != NULL) AND
          (ptr2 -> LINK != NULL)) DO
9:  IF (ptr1 -> exp == ptr2 -> exp) THEN
10:     new = GETNODE()
11:     new -> coff = ptr1 -> coff + ptr2 -> coff
12:     new -> exp = ptr1 -> exp
13:     ptr3 -> LINK = new
14:     ptr3 = new
15:     ptr1 = ptr1 -> LINK
16:     ptr2 = ptr2 -> LINK
17:     ENDIF
18:     IF (ptr1 -> exp > ptr2 -> exp) THEN
19:     new = GETNODE()
20:     new -> coff = ptr1 -> coff
21:     new -> exp = ptr1 -> exp
22:     ptr3 -> LINK = new
23:     ptr3 = new
24:     ptr1 = ptr1 -> LINK
25: ENDIF
26: IF (ptr1 -> exp
        ptr2 -> exp)
    new -> exp = ptr2 -> exp
    new -> exp = ptr2.
```

```
3: HEADER = GETNODE()
4: HEADER -> coff = NULL
5: HEADER -> exp = NULL
6: HEADER -> LINK = NULL
7: ptr3 = HEADER
```

```
26: new = GETNODE()
27: new -> coff = ptr2 -> coff
28: new -> exp = ptr2 -> exp
```

29: ptr3 → LINK = new

30: ptr3 = new

31: ptr2 = ptr2 → LINK

32: ENDIF

33: ENDWHILE

34: WHILE (ptr1 → link ≠ NULL)

35: new = GETNODE ()

36: new → coff = ptr1 → coff

37: ~~ptr3~~ new → exp = ptr1 → exp

38: ptr3 → LINK = new

39: ptr3 = new

40: ptr1 = ptr1 → LINK

41: ENDWHILE

42: WHILE (ptr2 → LINK ≠ NULL)

43: new = GETNODE ()

44: new → coff = ptr2 → coff

45: new → exp = ptr2 → exp

46: ptr3 → LINK = new

47: ptr3 = new

48: ptr2 = ptr2 → LINK

49: ENDWHILE

50: STOP

$(2x^4 + 2x^3 + 2x^2 + 2x) + (x^4 + x^3 + x^2 + x)$

(1) merging single Link list. (and concatination)

1 type - done

Another type -

Input: A and B are two single linked list. HEADER1 and HEADER2 are the pointers to the header node of A and B respectively.

output: C is the merged list of A and B and HEADER is the pointer to the header node of C.

D.S = Single linked list.

Step 1: ptr1 = HEADER1 → LINK

2: ptr2 = HEADER2 → LINK

3: HEADER = ~~new~~ GETNODE (NODE)

4: HEADER → DATA = NULL

5: HEADER → LINK = NULL

6: ptr3 = HEADER

7: WHILE (ptr1 → LINK ≠ NULL AND ptr2 → LINK ≠ NULL) DO

8: IF (ptr1 → DATA < ptr2 → DATA) THEN

9: new = GETNODE (NODE)

10: new → DATA = ptr1 → DATA

11:

12: ptr3 → LINK = new
ptr3 = new

13: ptr1 = ptr1 → LINK

14: ENDIF

15: ELSE IF (ptr1 → DATA > ptr2 → DATA) THEN

16: new = GETNODE (NODE)

17: new → DATA = ptr2 → DATA

18: ptr3 → LINK = new

19: ptr3 = new

20: ptr2 = ptr2 → LINK

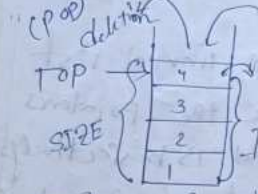
21: ENDELSEIF

22. ELSE IF (ptr1 → DATA = ptr2 → DATA) THEN
23: new = GETNODE (NODE)
24: ptr new → DATA = ptr1 → DATA
25: ptr3 → LINK = new
26: ptr3 = new
27: ptr1 = ptr1 → LINK
28: ptr2 = ptr2 → LINK
29: ENDELSEIF
30: ENDWHILE
31: WHILE (ptr1 →LINK ≠ NULL) DO
32: new = GETNODE (NODE)
33: new → DATA = ptr1 → DATA
34: ptr3 → LINK = new
35: ptr3 = new
36: ptr1 = ptr1 → LINK
37: ENDWHILE
38: WHILE (ptr2 →LINK ≠ NULL) DO
39: new = GETNODE(NODE)
40: new → DATA = ptr2 → DATA
41: ptr3 → LINK = new
42: ptr3 = new
43: ptr2 = ptr2 → LINK
44: ENDWHILE
45: STOP

CW 13/9/22

Stack (Abstract data type)

Linear data structure.



Last in first out
(LIFO Data Structure)

order of insertion — 1 2 3 4
Orders of retrival — 4 3 2 1
We to recursion using stack

Characters:-

1) Linear
2) Homogeneous
3) Ordered
4) LIFO

Why Stack is called a LIFO Data structure?

The last element inserted in the stack is the 1st element to be deleted. Hence it is called LIFO or FIFO (first in last out)

A stack is an ordered collection of homogeneous data elements where the insertion and the deletion operation take place at one end only.

In stack the insertion is called - push
" " deletion " - pop

The position of the stack where insertion and deletion is happening is called top.

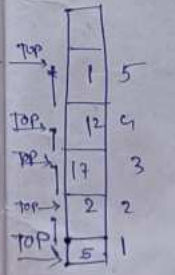
Each element of a stack is called - item

The no. of elements of the stack can accomodated - size

Representation of stack in memory

1. Array
2. link list

Array representation of stack



l = 1
u = SIZE
No of elements in the stack
SIZE = u - l + 1

Initially (TOP=0) // Stack is empty condition

5 2 17 12 1 TOP=0

TOP = TOP + 1
A[TOP] = ITEM

TOP = Size // my stack is full

Algo PUSH_Array

Input: The new item ITEM to be pushed onto it.
output: A stack with the newly pushed ITEM at the TOP position.
D.S: An array A with TOP as a pointer.

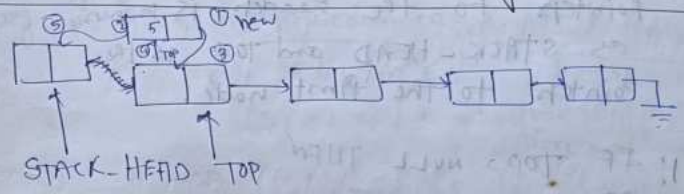
Step 1: IF TOP >= size THEN
2: PRINT "Stack Full"
3: ELSE
4: TOP = TOP + 1
5: A[TOP] = ITEM
6: ENDIF
7: STOP

Algo Pop_Array

Input: A stack with elements
output: Remove an item from top of the stack if it is not empty
D.S - An array A with top as a pointer.

Step 1: IF TOP < 1 THEN
2: PRINT "Stack Empty"
3: ELSE
4: ITEM = A[TOP]
5: TOP = TOP - 1
6: ENDIF
7: STOP

Representation of stack using linked list



Algorithm PUSH_Linked List

Input: ITEM is the item to be inserted.
output: A single linked list with a newly inserted node with data content ITEM.
D.S: - A single linked list whose pointer to header is known as STACK_HEAD and TOP is the pointer to the first node.

Step 1: new = GETNODE (NODE)
2: new → DATA = ITEM
3: new → LINK = TOP
4: TOP = new
5: STACK_HEAD → LINK = TOP
6: STOP

3: new → LINK = TOP
4: STACK_HEAD → LINK = new
5: TOP = new
6: STOP

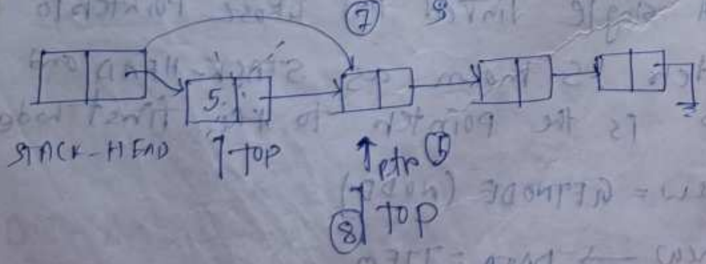
Algo POP-LL

Input: A Stack with elements.

Output: The removed item is stored in ITEM

D.S: A single linked list whose pointer to the header is known as STACK-HEAD and TOP is the pointer to the first node.

Step 1: IF TOP = NULL THEN
2: PRINT " Stack Empty"
3: EXIT
4: ELSE
5: ptr = TOP → LINK
6: ITEM = TOP → DATA
7: STACK-HEAD → LINK = ptr
8: TOP = ptr [while writing program write
⑧ → free (top)]
9: ENDIF
10: STOP



Applications of STACK

i) Execution of recursive programme (details)
ii) Evaluation to Arithmatic expressions (")
iii) Binding Rules (")

Static binding, Dynamic Binding

5+3 ÷ 4 × 5 × 10 + 3

① Precedence & Associativity

( ), ^, %, /, *, +, − L → R

[Except exponentiation, 3^{2^5} → R to L]

② ( ) → (bracket)

15/9/22

	Pr	Ass
+(unary), −(unary), NOT	6	
^	6	R → L
*, /	5	L → R
+, −	4	L → R
<, =, >, =, !=, (≠)	3	L → R
AND	2	L → R
OR, XOR	1	L → R

(A + (B * (C / D)) + ((E ^ F) * G)) → Parenthesis

Polish Notation

Infix – A + B (the operators are sitting between the operands)
Infix – +AB (Polish notation)
Postfix – AB+ (Reverse Polish notation – RPN)
(... before ...)
(... after ...)

1) ((A + ((B ^ C) – D)) * (E – (A / C))

Convert this infix into Prefix

((A + ((B ^ C) – D)) * (E – (A / C))

= ((A + (^BC) – D)) * (E – (/AC))

= ((A + (–^BCD)) * (E – /AC))

= (+A – ^BCD) * – E/AC

= +A – ^BCD * – E/AC

= * + A – ^BCD * – E/AC (Prefix)

Postfix

((A + ((B ^ C) – D)) * (E – (A / C))

= ((A + (BC^) – D)) * (E – AC/)

= (A + (BC^ – D)) * (E – AC/)

= (A + BC^D –) * (E – AC/)

= (A + BC^D – +) * (E – AC/)

= A + BC^D – + E – AC/ –

Q) Why do we use Post-fix eqn for evaluating the arithmatic expressions?

The postfix notation is used to represent algebraic expressions. the expressions written in postfix form are evaluated faster compared to infix notation as parenthesis is not required in postfix.

Symbol	In stack Priority value	In coming Priority value
+, –	2	1
*, /	4	3
^	5	6
Operand	8	7
(	0	9
)	–	0

Conversion of Infix expression to Postfix expression

(A+B)^C − ((D*E)/F)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

No low Priority operator will stand below the high P.o

Read symbol	stack	output
Initial	(	
1	((	
2	((+	A
3	((+	AB
4	(	AB+
5	(^	AB+
6	(^	AB+C
7	(-	AB+C^
8	(-(	AB+C^
9	(-(	AB+C^D
10	(-(*	AB+C^D
11	(-(*	AB+C^DE
12	(-	AB+C^DE*
13	(-/	AB+C^DE*
14	(-/	AB+C^DE*F
15		AB+C^DE*F/-

ISP & ICP

[(+^]

[operator will be sent, brace will disappear]

[High High Priority will in coming to the low Priority than everything.

in the stack will be popped out, and the braces will disappeared. This will continue till we get the opening brace.

Algo Infix to Postfix

Input: E is an arithmatic expression with ending delimited by ')' and incoming and in stack Priority values for all possible symbols

O/P: An arithmatic expression in Postfix form

PS: stack

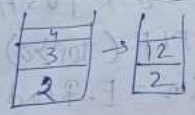
Step 1: TOP = 0 ; PUSH ('(')
2: WHILE (TOP > 0) DO
3: item = E.ReadSymbol() // ~~Same~~ the next scan symbol from the expression
4: x = POP()
5: Case : item = OPERAND
6: PUSH(x)
7: output (item)
8: Case : item = ')'
9: WHILE x ≠ '(' DO
10: output (x)
11: x = POP()
12: ENDWHILE
13: Case : ICP(~~item~~ x) >= ICP(item)
14: WHILE (ISP(x) >= ICP(item)) DO
15: output (x)
16: x = POP()

17: ENDWHILE
18: PUSH (x)
19: PUSH (item)
20: Case : ISP(x) ICP (item)
21: PUSH (x)
22: PUSH (item)
23: Default:
24: Print "Invalid expression"
25: ENDWHILE
26: STOP

Evaluating of postfix expression

Infix : A + (B * C) / D

Postfix: ABC * D/+

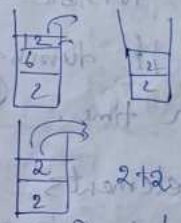


Input: AB(*D/+# A = 2, B = 3, C = 4, D = 6

↑ Ending delimiter (the end)

Read symbol	Stack	
A	2	PUSH (A=2)
B	2 3	PUSH (B=3)
C	2 3 4	PUSH (C=4)
*	2 12	POP(4) POP(3) PUSH (12)
D	2 12 6	PUSH (D=6)

/	2 2	POP(6), POP(12), PUSH(2)
+	4	POP(2), POP(2), PUSH(4)



[1st Popped out = Right most operator]

① Comparison between array and linked list

Array

① An array is a grouping of data elements of equivalent data type.

Linked list

① A linked list is a group of entities called a node. The node includes two segments: data and address.

2) It stores the data elements in a contiguous memory zone.	2) It stores elements randomly or can say anywhere in the memory zone
3) In case of an array, memory size is fixed, and it is not possible to change it during the run time.	3) In the linked list, the placement of elements is allocated during the run time.
4) the elements are not dependent on each other.	4) the data elements are dependent on each other.
5) The memory is assigned at the compile time.	5) the memory is assigned at run time.
6) It is easier and faster to access the element in an array.	6) In a linked list, the process of accessing elements takes more time.
7) In the case of an array memory utilization is ineffective	7) In the case of the linked list, memory utilization is effective.
8) when it comes to executing any operation like insertion, deletion array takes more time.	8) when it comes to executing any operation like insertion, deletion the linked list takes less time.

(2) Disadvantages of single linked list and it's remedy.

Disadvantages— 1) It requires more space as pointers are also stored with information. 2) Different amount of time is required to access each element. 3) If we have to go to a particular element then we have to go through all those traverse elements that come before that element. 4) we can not traverse it from last and only from the beginning.

The problem 1 is solved by using circular linked list.
The 1st problem is solved by using doubly linked list

23/5/22

CW Application of stack

Recursion

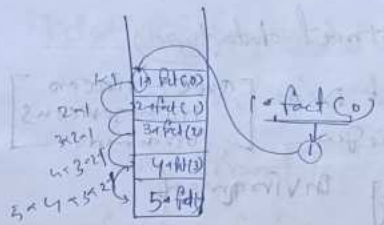
5! = 5×4×3×2×1

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

Iterative version

```
int fact (int n)
{
  int i, f=1;
  for (i=1; i<=n; i++)
    f = f*i;
}
```

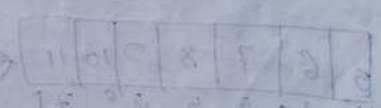
① What is the difference between iterative version and recursion, recursion Basics definition Prog.



function calling is maintained by stack.

* This DSA is LIFO DSA.

```
int i = 3;
main()
{
  pf(i) = {3,
  int i = 1;
  func();
  pf(i);7)
}
func()
{
  int i = 2
  pf(i); 5)
}
```



①

Queue (It is a abstract data type)

First In First out DSA [queue - linear, homogeneous, ordered]

The application of queue -

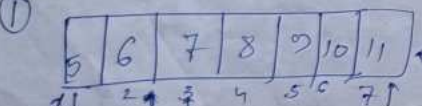
① multiprogramming environment
- Resource sharing
- Processor sharing

Representation

[Queue has 2 pointers = rear, front]

Queue = empty
front = 0
rear = 0
] condition

①



← Insertion (enqueue)

rear = rear + 1

queue [rear] = element

if (front = 0)
front = 1

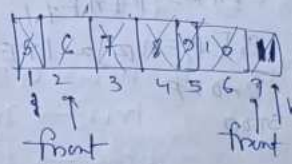
Queue = full
rear = size

Deletion (dequeue)

The deletion will be held front front.

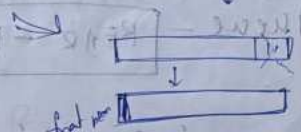
Item = q[front]

front = front + 1



Queue has 1 element -
front = rear

if (front = rear)
front = 0
rear = 0



Algo Insertion - Array / Algorithm ENQUEUE

(enqueue in)

Input: element ITEM to be inserted

output: The element is at the rear of the queue.

D.S - Q is the array representation of a queue structure; two pointers FRONT and REAR of Q are known.

Step 1: IF (REAR = N) THEN [N = size]
2: PRINT ("Queue Full")
3: EXIT
4: ELSE

```
5: IF (REAR = 0 AND FRONT = 0) THEN
6:   FRONT = 1
7: ENDIF
8: REAR = REAR + 1
9: Q[REAR] = ITEM
10: ENDIF
11: STOP
```

27/9/22

No of elements in the queue — REAR − FRONT + 1

Algo Deletion in queue in array / Algo DEQUEUE

Input: A queue with elements. FRONT and REAR are two pointers of the queue.

O/P: the deleted element is stored in ITEM.

P.S: Q is the array representation of queue structure.

```
Step 1: IF (FRONT = 0) THEN
2:   PRINT ("queue empty")
3:   EXIT
4: ELSE
5:   ITEM = Q[FRONT]
6:   IF (FRONT = REAR) THEN   // queue has only one element
7:     REAR = 0
8:     FRONT = 0
9:   ELSE
10:    FRONT = FRONT + 1
11:  ENDIF
12: ENDIF
13: STOP
```

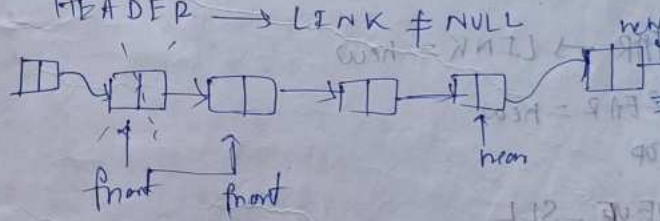
Queue empty (SLL)

FRONT = HEADER

REAR = HEADER

HEADER → LINK = NULL

Queue has 1 element



HW Insertion, deletion using SLL.

Algo ENQUEUE − SLL

Input: ~~A queue with~~ Element ITEM to be inserted

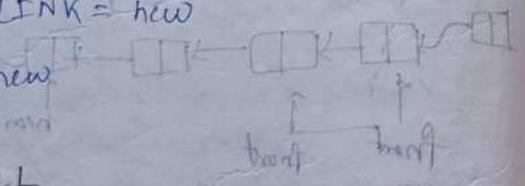
O/P: A single LL with a newly inserted node with the data content ITEM.

D.S: A single linked list whose pointers are known as front and rear.
Rear is a pointer to the new node and front is a pointer to the node to be deleted.

Step 1: new = GETNODE (NODE)
new → DATA = ITEM
new → LINK = NULL

1: IF (FRONT = HEADER) AND (REAR = HEADER) AND (HEADER → LINK = NULL)
2: // free the element

1: new = GETNODE (NODE)
2: new → DATA = ITEM
3: new → LINK = NULL
4: REAR → LINK = new
5: REAR = new
6: STOP



Algo DEQUEUE_SLL

Input: A Queue with elements. FRONT and REAR are two pointers of queue.
O/P: the deleted item is stored in ITEM.
D.S: A single linked list whose pointers are known as front and rear.
Rear is a pointer to the new node and front is a pointer to the node to be deleted.

Deletion → empty condition
Insertion → full condition (array)
(as LL does not get to be full)

Step 1: IF (FRONT = HEADER) AND (REAR = HEADER) AND (HEADER → LINK = NULL) THEN [or front = rear = NULL]
2: PRINT ("Queue is empty")
3: EXIT
4: ELSE
5: ITEM = FRONT → DATA
6: FRONT = FRONT + 1
7: HEADER → LINK = FRONT → LINK
8: ENDIF → FRONT = FRONT → LINK
9: STOP

Ptr (may, front → link if rear Possible)

① Recursion – A function is called 'recursive' if a statement within the body of a function calls the same function again and again. Sometimes called 'Circular definition' recursion is thus the process of defining something in terms of itself.

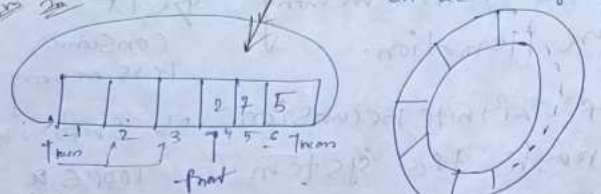
② Difference between recursion and iterative version (iteration)

Recursion	Iterations
1) Recursive function is a function that is partially defined by itself.	1) Iterative instructions are loop based repetitions of a process
2) It uses selection structure	2) It uses repetition structure
3) It terminates when a base case is recognized.	3) Iteration terminates when the loop-condition fails.
4) It uses more memory than iteration.	4) It consume less memory.
5) Iff Infinite recursion can crash the system	5) Infinite loops uses CPU
6) It makes code smaller.	6) It makes code longer

③ Difference between stack and queue one more point is- the name of insertion and deletion.

Stack	Queue
1) Stacks are based on LIFO principle.	1) Queues are based on FIFO principle.
2) Here insertion and deletion takes place only from one end.	2) Insertion takes place ~~from~~ at rear and deletion takes place at front. i.e. insertion and deletion takes place at opposite ~~si~~ ends.
3) There are only one pointer.	3) There are two pointers.
4) Used in solving problems works on recursion.	4) Used in solving problems having sequential processing.
5) It has not any types.	5) It is ~~two~~ of three types. 1) Circular 2) Priority 3) Double ended
6) Can be considered vertically	

Disadvantage of queue (many elements are not present but we following the queue rule)



Circular queue

next
~~any index~~ to ~~any~~

* The formula to go ~~1st~~ back index -

(Queue increament) = [(rear MOD size) + 1]

(ex - [6 % 6) + 1 = 1)

* Queue is full - [(REAR MOD SIZE) + 1 = FRONT]

(9 % 10) + 1 = f
= 9 + 1 = f
= 10 = f

Algo DEQUEUE_CQ

Input: A queue with elements. FRONT and REAR are two pointers of the queue

O/P: The deleted element is stored in ITEM

D.S. CQ is the array representation of circular queue structure.

Step 1: IF (FRONT = 0) THEN
2: PRINT ("queue empty")
3: EXIT
4: ELSE
5: ITEM = CQ [FRONT]
6: IF (FRONT = REAR) THEN
7: REAR = 0
8: FRONT = 0
9: ELSE
10:
10: FRONT = (FRONT MOD SIZE) + 1
11: ENDIF
12: ENDIF
13: STOP

Algo ENQUEUE_CQ

Input: An element ITEM to be inserted in the queue.

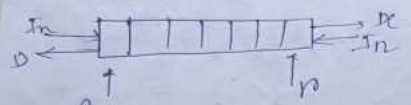
O/P: CQ with item.

D.S: CQ is the array representation of Circular queue structure.

Step 1: IF ((REAR MOD SIZE) + 1) = FRONT) THEN
2: PRINT ("queue full")
3: EXIT
4: ELSE
5: IF (REAR = 0 AND FRONT = 0) THEN
6: FRONT = 1
7: ENDIF
8: REAR = (REAR MOD SIZE) + 1

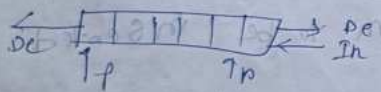
9: QUEUE[REAR] = ITEM
10: END IF
11: STOP

Double ended queue (definition & types)

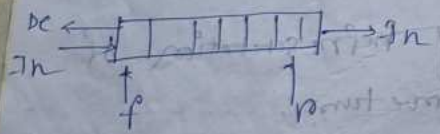


In case dequeue, the insertion and deletion can be done by both end.

1) I/P restricted dequeue.



2) O/P restricted dequeue.



Double ended queue (dequeue)

In computer Double ended queue is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front or back. It is also often called a head-tail linked list, though properly this refers to a specific data structure implementation of a deque.

There are 2 types of deque -

1) Input restricted queue - Here insertion operation can be performed at only one end, while deletion can be performed from both ends.
2) Output restricted queue - Here deletion operation can be performed at only one end while insertion can be performed from both ends.

Linear data structure - It is a type of D.S where the arrangement of data follows a linear trend. In this D.S the element is directly linked to its previous and next elements.

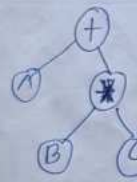
11/10/22

Non linear data structure - we can not represent them in 1 Dimention representation, we need 2 dimentional representation. we need

For hierarchy representation, 2d representation.

It is a type of D.S where the arrangements of data follows a linear trend. Here the elements are hierarchically attached

A + B * C



Expression tree

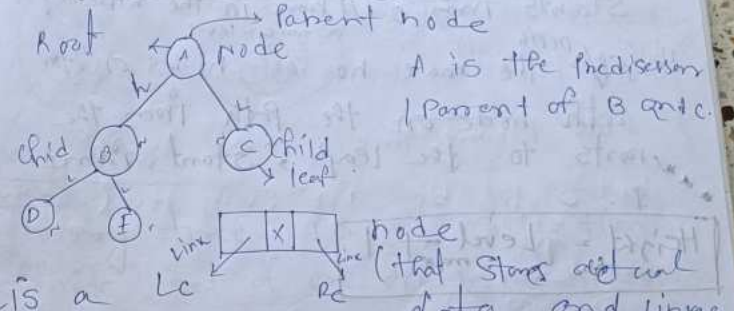
For this this type of rep. we need ~~tree~~ 2D rep.

In 1D — time complexity more, 2D — less

Advantages of 2D rep. > Comparing 1D representation.

In the linear data structure the data components ~~the~~ are sequentially attached and each component is traversable by way of a single run whereas non-linear data structure stands for data components components that are hierarchically attached and therefore are existing at different levels, which is much more benificial from a linear D.S. on the other hand the time complexity of linear D.S is more than non linear DS.

Tree (Abstract data type)



Link is a Pointer to a node in a tree

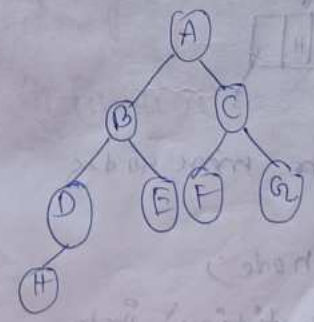
// Link is also called edge

Parent = Immidiate predisessors of a node

Child = Immidiate successors of a node

Root = Specially designated node which has no Parent

Terminal nodes / Leaf node = If a node has no child then it's called leaf node



Level

0

1

2

3

Height = 3

Deg(A) = 2

Siblings of B = 2 (D, F)

Level = IS a rank of the hirerchy of a node, always Starts from 0. // rank in the hirerchy of a tree - level

Height/depth = The max no. of nodes on the path node on the path from the roots to the leaf. stand from one

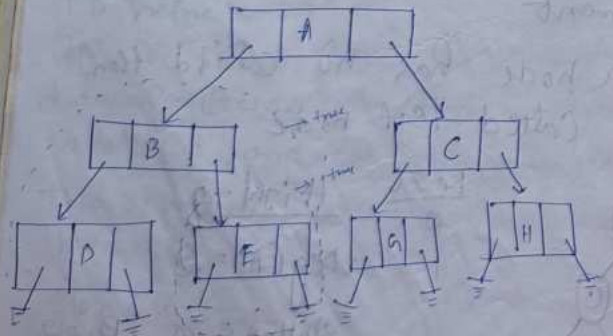
I.

Height = Level + 1
tree max

Degree - The max no. of children that a node can have. (out degree mainly degree = indeg. + outdeg. i.e. edges connected to a node) max no. of

Siblings - children of the same Parent

Tree - Finite collection of one or more nodes. It should have a root.



To Finite collection of one or more nodes such that

1) Root (Specialy designated node)
2) the remaining nodes are partitioned into n (n>0) subsets, where each of the subset is a tree.

→ Tree is collection of trees.

Binary tree

Definition -

A binary tree is a collection of finite set of one or more nodes such that -

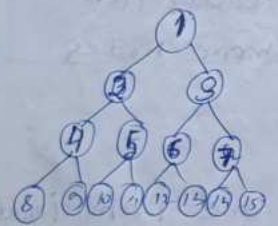
1) Root
2) the remaining nodes are partitioned into 2 disjoin subsets, where each of the subset is a tree.

Difference between binary tree and tree -

Tree	Binary tree
1) Tree can be empty	1) Binary tree can not be empty
2) Definition	2) Definition

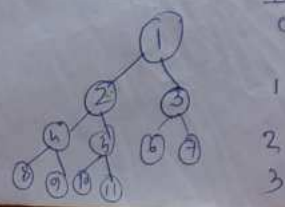
1) Binary tree (Classification) (Imp) ***

1) Full Binary tree - max no. of nodes possible in every level/all level ← either 0 or 2 child



2) Complete B.T (Imp) ***

A b.t is called C.B.T if it has max possible no. of nodes in all the level except possibly the last level and the filling of the last node, from start from as left as possible.



Every C.B.T is B.T but reverse is not true

Difference between B.T and C.B.T -

Binary tree (Full)

A tree is called if where there

(1) Full binary tree is max no. of nodes possible in all levels. i.e. have a node can't have just one child.

(2) There is no order of filling nodes in a full binary tree.

(3) Full B.T has no application as such but is also called a strict b.t.

(4) It

Complete Binary Tree

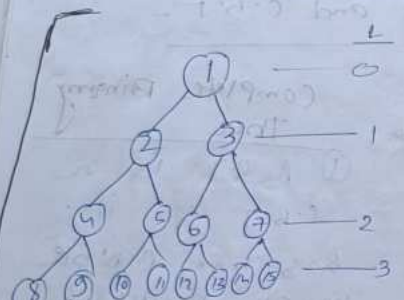
(1) A B.T is a C.B.T if it has max possible no. of nodes in all level except possibly the last level and the filling of last level will start from as left as possible. i.e. here a node in the last level can have only one child.

(2) Here, the level node should be filled from the left to right.

(3) C.B.T are mainly used in heap based data structure.

Lemma 1:

The maximum possible number of nodes on level l is 2^l ($l \geq 0$).



level	
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$

Proof:

Basis Root is the only node at level 0

So the max possible no. of nodes at level 0 is $2^0 = 1$

Hypothesis: Suppose for all i, $i \leq l$, the above formula is true.

i.e — The max possible no. of nodes for level i is 2^i.

Induction Step- Since each node can have the max degree 2, so the no. of nodes at level $i+1$ is -

$2^i \times 2 = 2^{i+1}$

This formula is true for level $i+1$.

Hence proved.

Lemma 2:

The maximum number of nodes possible in a binary tree of height h is $2^h - 1$

(Previous diagram $[2^4 - 1 = 15]$)

Proof: the max ~~Possible~~ no of nodes is ✓ in a B.T if each level has max possible no. of nodes (According to lemma 1)

(we can write Lemma 1)

the max. no. of nodes in a B.T is

$n_{max} = 2^0 + 2^1 + \dots + 2^{l_{max}} = \sum_{i=0}^{l_{max}} 2^i$

$\Rightarrow n_{max} = \sum_{i=0}^{l_{max}} 2^i$

(Coz here l_{max} is the max level of the tree)

$$\Rightarrow n_{max} = \frac{2^{l_{max}+1} - 1}{2-1}$$

[$\because h = l_{max} + 1$]

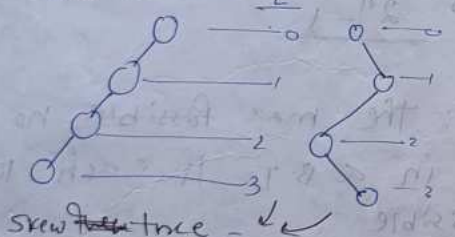
$\Rightarrow n_{max} = 2^h - 1$ [Substituting $l_{max}+1$ into h]

Hence Proved.

Qu

Lemma 3: 14/10/22

The minimum no. of nodes that is possible in a binary tree of height h is h.



Skew tree

Proof:

If it is a binary tree that contains min no. of nodes.

The minimum no. of nodes is possible in a binary tree if each level has minimum no. of nodes, that is every parent has only one child, such kind of trees are called skew binary trees. A skew binary tree is a tree having only one path containing h no. of nodes.

So $n_{min} = h$ (Proved)

Lemma 4: For any non-empty binary tree if n is the number of nodes and e is the number of edges then $n = e + 1$

Basis

If $n = 1$, $e = 0$

O Root

Hypothesis

Let us assume that the above formula is ~~time~~ true for all $e' = e - 1$, $n' = n - 1$ such that $n' = e' + 1$

Induction step

$$n' = e' + 1$$

$$\Rightarrow n' + 1 = (e' + 1) + 1$$

$$\Rightarrow n' + 1 = (n' - 1 + 1) + 1$$

$$= n' + 1$$

Hence this formula is true for all n, hence n=e+1. Hence (proved).

Lemma 5: ***

For any nonempty B.T if n_0 is the number of leaf nodes and n_2 is the number of internal nods then $n_0 = n_2 + 1$

Proof:

Let N be the total no. of nodes such that

$N = n_0 + n_1 + n_2$ — (i)

[n_1 = no of nodes of deg 1
n_2 = ... deg 2
n_0 = ... deg 0]

If e is the no. of edges then $N = e+1$ — (ii)

$e = n_0 \times 0 + n_1 \times 1 + n_2 \times 2$ — (iii)

$\Rightarrow e = n_1 + 2n_2$ — (iii)

~~Comparing (i) and (iii)~~

Substituting (iii) into (ii)

$N = 2n_2 + n_1 + 1$ — (iv)

equating (i) and (iv)

$n_0 + n_1 + n_2 = 2n_2 + n_1 + 1$ $\Rightarrow n_0 = n_2 + 1$ (proved)

Lemma 6: ***

The height of a complete B.T with n nodes is $\lceil \log_2(n+1) \rceil$ → Lower bound [] ceiling []

Proof: Let h be the height of a complete B.T

Then, we can write

Full b.t = Complete b.t

$n \leq 2^0 + 2^1 + 2^2 + \dots + 2^{h-1}$

$\Rightarrow n \leq 2^h - 1$ [$l = h-1$]

$\Rightarrow 2^h \geq n+1$

Taking log in both sides

$h = \lceil \log_2(n+1) \rceil$ (Proved)

[As $\geq$, so $\lceil\ \rceil$]

Lemma 7: ***

The total no. of Bintrees possible with n nodes is $\frac{1}{n+1} {}^{2n}C_n$

Formula

HW

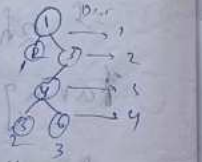
Lemma 8:

- In a BT of height h there are atmost 2^{h-1} leaf nodes. (max no. of) (ind = in induction)

Basis: Root is the only node at height = 1 [level = 0] [∴ h = 0+1]

So, max no. of leaf nodes at height 1 is $2^{1-1} = 2^0 = 1$

Hypothesis: Suppose ~~for~~ Let us assume that the above formula is true for all ~~h~~ $2^{h'-1}$ ~~h''~~ h', $h'-1$, such that $2^{h''} = 2^{h'-1}$ ~~where h' - 1 is the height of 2nd last height~~.



Induction Step: Since each ~~v~~ leaf nodes can have the ~~max~~ deg = 1, so no. of leaf nodes at height $h'+1$ =

$\{2^{h'+1-1} \times 1 = 2^{h'} \cdot 2^{(h'+1)} - 1$ $\left[2^{h'-1} < 2^{h'}, \; 2^{h'} = 2^{(h'+1)-1}\right]$

Here the formula is true for all h', hence 2^{h-1} is the proved.

Rout this

Chw

Representation of tree in memory — 18|10|22

- static representation
- dynamic representation

Array (static representation):

The maximum and minimum sizes that an array may require to store a binary tree with n nodes is

size max = $2^n - 1$ **

size min = $\lceil \log_2(n+1) \rceil$ **

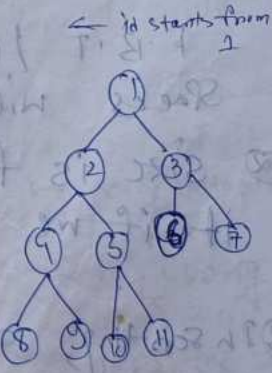
** Node i

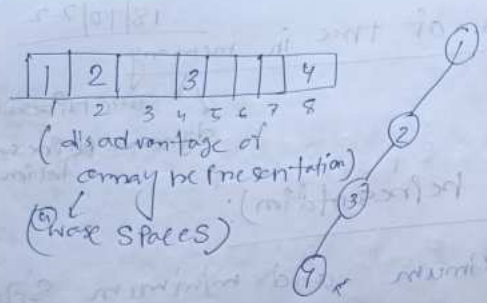
Parent (i) = $\lfloor i/2 \rfloor$, i ≠ 0

Left child (i) = 2i, 2i ≤ n

Right child (i) = 2i+1, 2i ≤ n

1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11





(disadvantage of array representation)
(wise spaces)

Advantages of array representation-

① Any node can be accessed by any node by calculating the index. others

② Datas are stored without any pointers to their successors and predecessors.

Disadvantages

① In case of not dealing with F.B.T / C.B.T, the majority of spaces will be wasted.

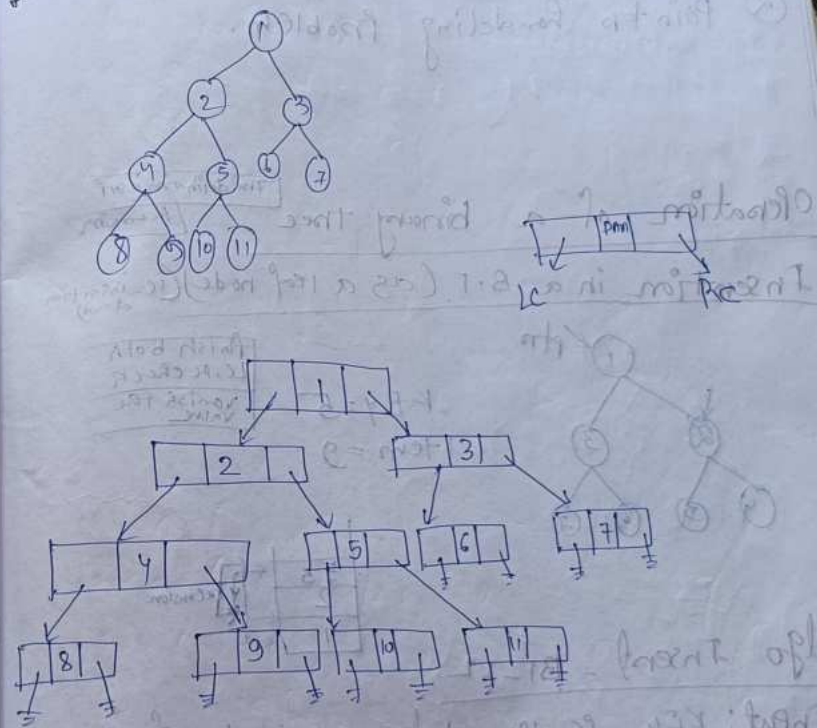
② size is fixed, we can't change it if we require.

③ In sorting, deleting of a node is inefficient in this representation.

④ waste of spaces.

Remedy: Linked Listed representation of tree

Linked Representation



DisAdvantages

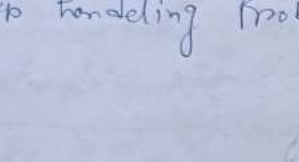
① It allows dynamic memory allocation hence the size of ~~tree~~ list can be changed.

② The visualization of ~~tree~~ Linked List is as same of the tree visualization.

Disadvantages

① Predecessor of the node is not very easy handling

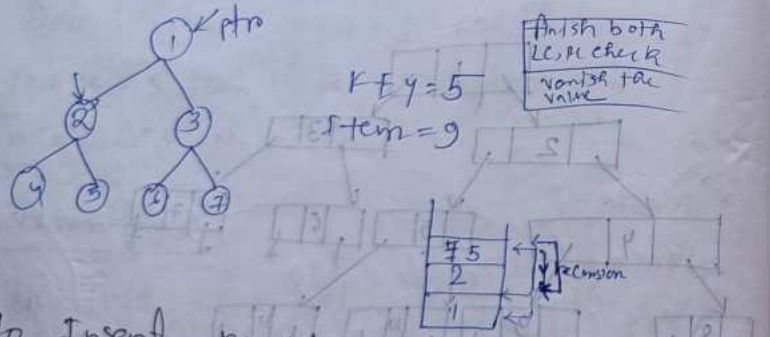
② Pointer handeling problem.



Operation of a binary tree

The definition itself a recursion

Insertion in a B.T (as a leaf node) (ie insertion at end)



Algo Insert_BT_LL

Input: KEY is the data content of the node after which the new node is to be inserted and ITEM is the data content of the node to be inserted.

QP: A node with data content ITEM inserted as an EXTERNAL node.

P's: Linked Representation of Binary tree.

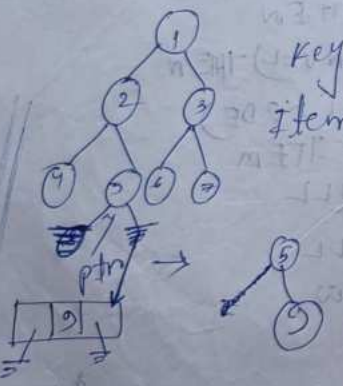
```
Step 1:  ptr = SEARCH (ROOT, KEY)
     2:  IF (ptr = NULL) THEN
     3:  PRINT ("search unsuccessful: NO INSERTION")
     4:  EXIT
     5:  ENDIF
     6:  IF (ptr → LC = NULL) OR (ptr → RC = NULL) THEN
     7:  READ(OPTION as L OR R)
     8:  IF (OPTION = L) THEN
     9:  IF (ptr → LC = NULL) THEN
    10:  new = GETNODE (NODE)
    11:  new → DATA = ITEM
    12:  new → BC = NULL
    13:  new → RC = NULL
    14:  ptr → LC = new
    15:  ELSE
    16:  PRINT "INSERTION NOT POSSIBLE AS LEFT CHILD"
    17:  EXIT
    18:  ENDIF
    19:  ELSE
    20:  IF (ptr → RC = NULL) THEN
    21:  new = GETNODE (NODE)
    22:  new → DATA = ITEM
    23:  new → LC = NULL
```

1) Ptr = SEARCH(ROOT, KEY)
2) IF (ptr = NULL) THEN
3) PRINT "search unsuccessful, key is the Present; insertion not Possible"
4) EXIT
5) ENDIF
6) IF (ptr → LC = NULL) OR (Ptr → RC = NULL) THEN
7) READ(OPTION as L OR R)
8) IF (OPTION = L) THEN
9) IF (Ptr → LC = NULL) THEN
10) new = GETNODE(NODE)
11) new → DATA = ITEM
12) new → LC = NULL
13) new → RC = NULL
14) ptr → LC = New
15) ELSE
16) PRINT "insertion not possible as LC"
17) ENDIF
18) ELSE
19) IF (ptr → RC = NULL) THEN

20) IF (PTR → RC = NULL) THEN

20) new = GETNODE (NODE)
21) new → DATA = ITEM
22) new → LC = NULL
23) new → RC = NULL
24) ptr → RC = new
25) ELSE
26) PRINT "in. not p. as RC"
27) ENDIF
28) ENDIF
29) ELSE
30) PRINT "KEY has already child in not possible"
31) ENDIF
32) STOP

24: new → RC = NULL
25: ptro → RC = new
26: ELSE
27: PRINT ("INTERSECTION NOT POSSIBLE AS RIGHT CHILD")
28: EXIT
29: ENDIF
30: ELSE
31: PRINT ("Key node already has a child")
32: ENDIF
33: ENDIF
34: STOP



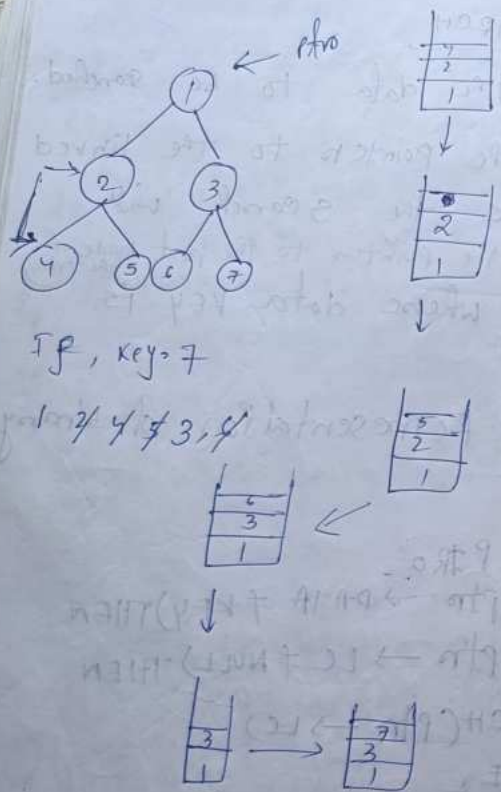
Key = 5
Item = 9

Insertion in B. search Tree → done will be

Algorithm SEARCH

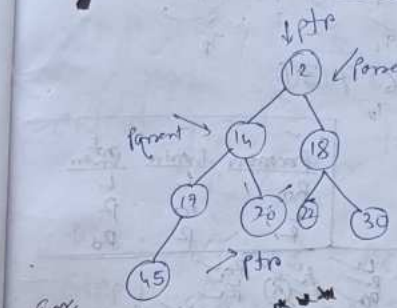
Input: KEY is the data to be searched.
PTRO is the pointer to the linked list where the search will start. (PTRO is the pointer to the root node of the tree)
O/P: LOCATION, where data KEY is located.
D.S: Linked Representation of binary tree.

Step 1: ptro = PTRO
2: IF (ptro → DATA ≠ KEY) THEN
3: IF (ptro → LC ≠ NULL) THEN
4: SEARCH (ptro → LC)
5: ELSE
6: RETURN (0)
→ ENDIF
7: IF (ptro → RC ≠ NULL)
8: SEARCH (ptro → RC)
9: ELSE
10: RETURN (0)
11: ENDIF
12: ELSE
13: RETURN (ptro)
14: ENDIF
15: STOP



21/10/22

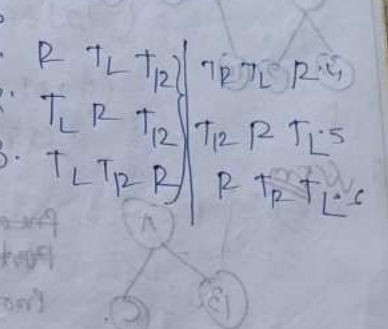
Deletion from a Binary Tree (leaf node)



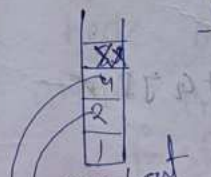
(No need to write algo)

~~Traversal~~

Preorder
1. R T_L T_{12}
Inorder 2. T_L R T_{12}
Postorder 3. T_L T_{12} R



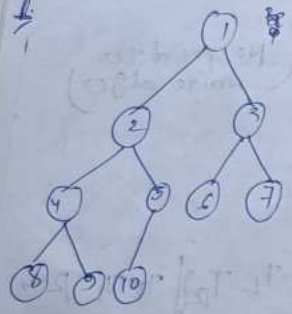
Preorder

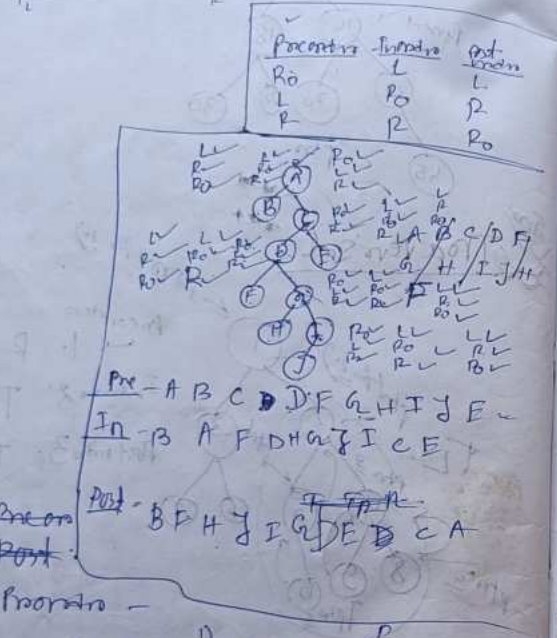


Preorder := 1 2 4 8 9 5 10 3 6 7

Inorder = 8 4 9 2 10 5 1 6 3 7

Postorder 8 9 4 10 5 2 6 7 3 1





Pre - A B C D F G H I J E

In - B A F D H G J I C E

Post - B F H J I G E D C A

Preorder -

A B C D F G H J E

E

Inorder - B A F D H G J I

C E

Postorder - B F H J H G E D A C

Algorithm Inorder

Input: ROOT is a pointer to the root node of the binary tree.

Output: Print all the nodes in inorder fashion.

D.S: Linked Representation of B.T

Step 1: ptr = ROOT

2: IF (ptr ≠ NULL) THEN

3: Inorder(ptr → Left)

4: PRINT (ptr → DATA)

5: Inorder (ptr → RC)

6: ENDIF

7: Stop

Algorithm preorder

Input: ROOT is a pointer to the root node of the binary tree

output: Print all the nodes in Preorder fashion.

D.S: Linked Representation of B.T

Step 1: ptr = ROOT

2: IF (ptr ≠ NULL) THEN

3: PRINT (ptr → DATA)

4; Preorder (ptr → LC)
5; Preorder (ptr → RC)
6; ENDIF
7; STOP

Algorithm post order

Input; Root is a pointer to the root node of a binary tree
O/P: Print all the nodes in post order fashion
D'S: Linked representation of B.T

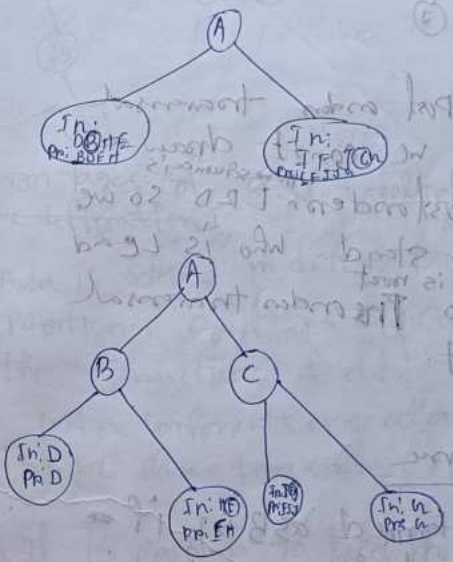
Step 1; ptr = ROOT
2; IF (ptr ≠ NULL)
3; Postorder (ptr → LC)
4; Preorder (ptr → RC)
5; PRINT (ptr → DATA)
6; ENDIF
7; STOP

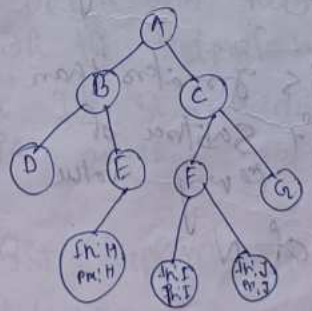
10/11/22

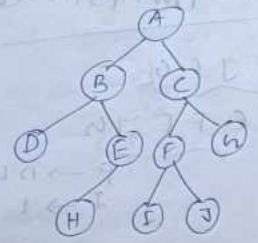
Formation of Binary tree from its traversal

Inorder; D B H E A I F J C G
Preorder; A B D E H C F I J G

P → D L R
I → L D R





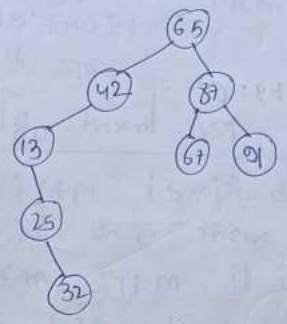


① If Preand Post order traversal is given then we can't draw a tree, as in Postorder LRD so we can not understand who is L and who is R. So Inorder traversal is needed must.

Binary Search tree

A BST tree is formed as BST if each node N satisfies the following properties:

i) The value at N is greater than every value in the left subtree of N and its less than every value in the right subtree of N.



Complexity

In BST = $O(\log_2 n)$

B-tree, B+ tree Definations only

Advantages of BST, application of BST

① Data is stored in different nodes in an arranged pattern, so it is easy to remember the structure of data hierarchically

2) Data components are attatched hierarchically so, we can traverse any way.

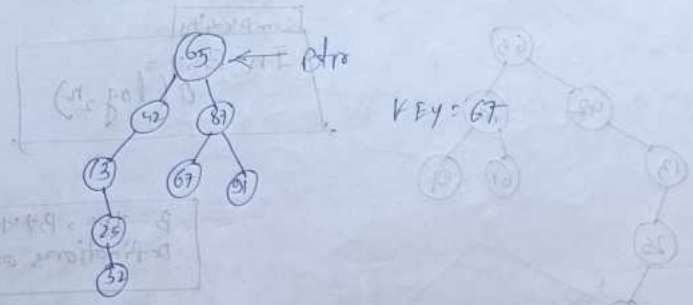
3) It is easier to update data in BST.

4) Depth and height is determined in BST with the help of nodes.

5) It search data very fastly, this is another main advantage.

② BST supports operations like insertion, deletion, search, floor, greater, smaller etc in $O(h)$ time, where h is the height of the BST.

Searching in a BST



Algo Searching - BST

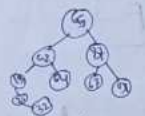
Input: ITEM is the data that is to be searched.

output: If found then Points to the node containing the ITEM, else a message.

D.S: Linked representation of ~~BST~~ B.T.

Step: 1: ptr = ROOT, flag = FALSE
2: WHILE (ptr ≠ NULL) AND (flag = FALSE) DO
3: Case: ITEM < ptr → DATA
4: ptr = ptr → LCHILD
5: Case: ITEM = ptr → DATA
6: flag = TRUE
7: Case: ITEM > ptr → DATA
8: ptr = ptr → RCHILD
9: ENDCASE
10: ENDWHILE
11: IF (flag = TRUE) THEN
12: PRINT ("Item is found at ptr")
13: ELSE
14: PRINT ("Item doesn't exist")
15: ENDIF
16: STOP

Algo Insert - BST



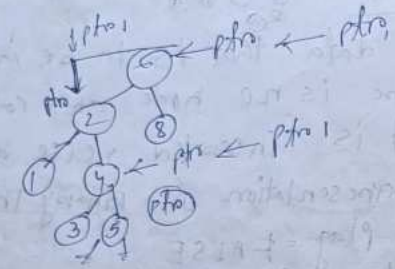
I/p: ITEM is the data that is to be inserted.

o/p: If there is no node containing ITEM, it is inserted, else a message.

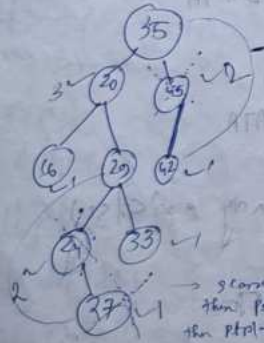
D.S: Linked representation of Binary Tree.

Step: 1: ptr = ROOT, flag = FALSE
2: WHILE (ptr ≠ NULL) AND (flag = FALSE) DO
3: Case: ITEM < ptr → DATA
4: ptr1 = ptr
5: ptr = ptr → LCHILD
6: Case: ITEM = ptr → DATA
7: flag = TRUE
8: PRINT ("Item already exists")
9: EXIT
10: Case: ITEM > ptr → DATA
11: ptr1 = ptr
12: ptr = ptr → RCHILD
13: ENDCASE
14: ENDWHILE
15: IF (ptr = NULL) THEN
16: new = GETNODE (node)
17: new → LCHILD = NULL, new → DATA = ITEM
18: new → RCHILD = NULL
19: IF (ptr1 → DATA < ITEM) THEN
20: ptr1 → RCHILD = new

21: ELSE
22: ptr1 → LCHILD = new
23: ENDIF
24: ENDIF
25: STOP



** Deletion (Don't have to write the algo)



ptr1 → RCHILD = ptr → LCHILD [Case 2] (in any)

→ search 29 then Ptr1 = ptr then ptr1 → RCHILD = NULL (in any) [Case 1]

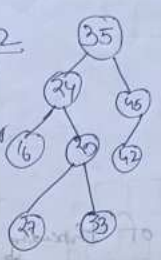
case 1: deletion of leaf node
case 2: ~~deletion~~ of 1 child Node with
case 3: Node with 2 children (in any)

Step 1 for case 3: 16 ~~28~~ 24 29 29 33

*** (inorder traversal of BST = Sorting)

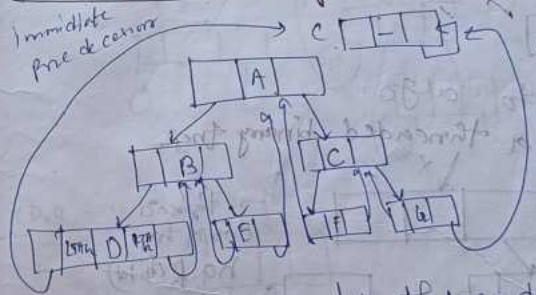
we have to remove the node with its inorder successor.

Step 2 (overwrite)
Step 3 Retrieve it by eliminating 24



Complexity of Sorting O(n)

Threaded Binary Tree



D B E A F C G

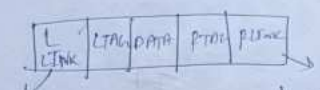
Inorderly threaded B.T

TAG → LTAG
→ RTAG

If LTAG = 1, RTAG = 0

↓ normal tag Pointer
↓ threade

Threaded B. T representation

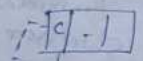


Connecting Predecessors

Connecting successors

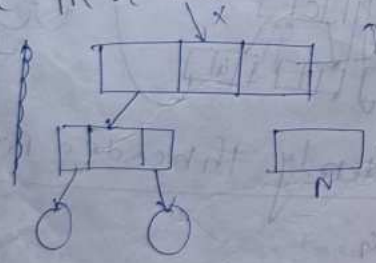
What is T B T, imp, diff types of threading representation, advantages, why dummy node is required, why do we need AVL tree, 4 types of AVL, what is AVL tree

If the tree is empty then



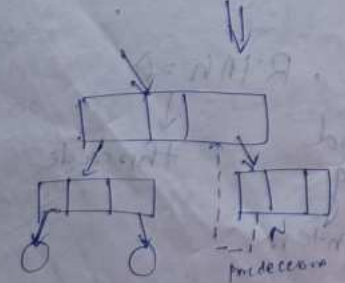
No needs to do algo.

Insertion in a threaded binary tree

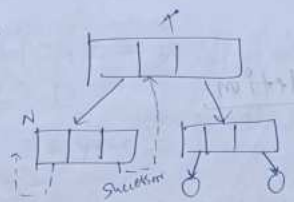


Type-1

Case-1



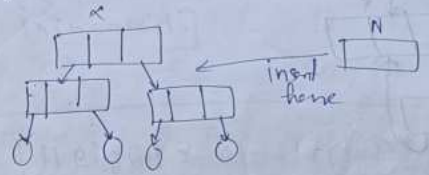
Case-1:

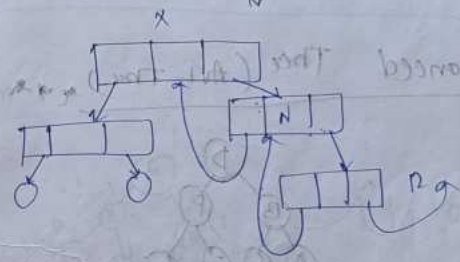


Case-II

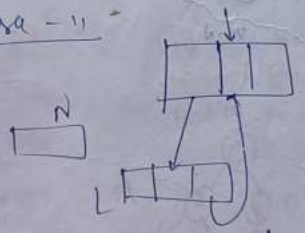
Type-II

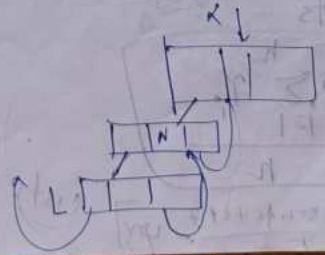
Case-1

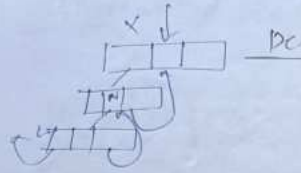




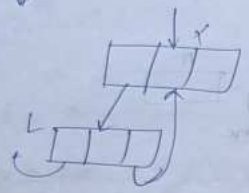
Case-II



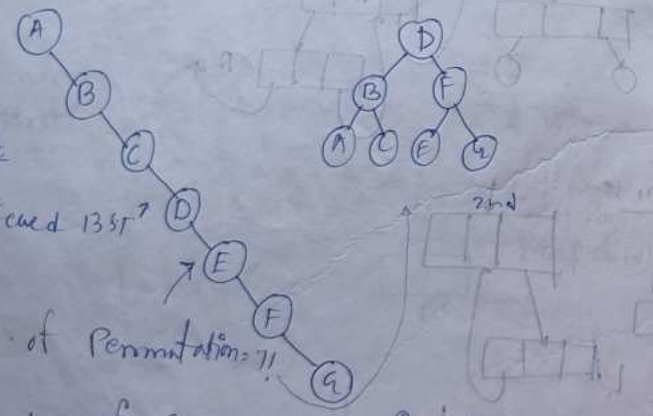




Deletion



Height Balanced Tree (AVL Tree) ***



Y_i = no of comparison for ith element

n = number of elements

The avg search time $= \frac{Y \cdot \sum_{i=1}^{n} Y_i}{n}$

$\left[\frac{1+2+3+4+5+6+7}{7} = 4 \text{ yrs}\right]$ (ans)

$Y = \frac{1+2+2+3+3+3+3}{7}$ $\left(\frac{1+4+12}{7}\right)$

$= \frac{17}{7}$ (ans)

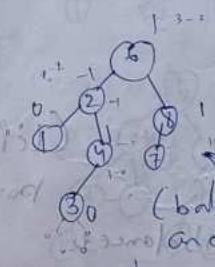
$= 2.4 \text{ yrs}$

[The tree should not be skewed, as if it is skewed then the complexity will increased.]

***

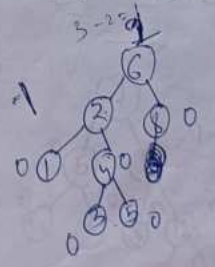
bf = [Height of left subtree − Height of right subtree]

↓ Balance factor



bf = -1, 0, 1

(balanced tree as bf of each and every node is -1, or 0, or 1)

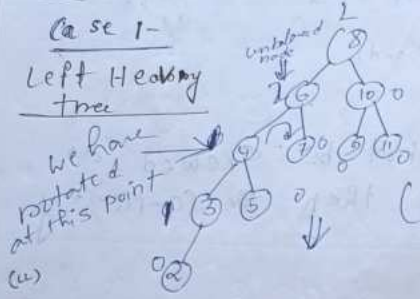


[The tree will be disbalance when we will insert and or delete a node.]

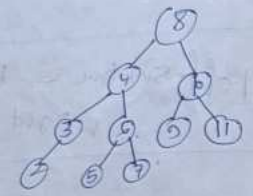
Four different types of notation -

(Single notation)

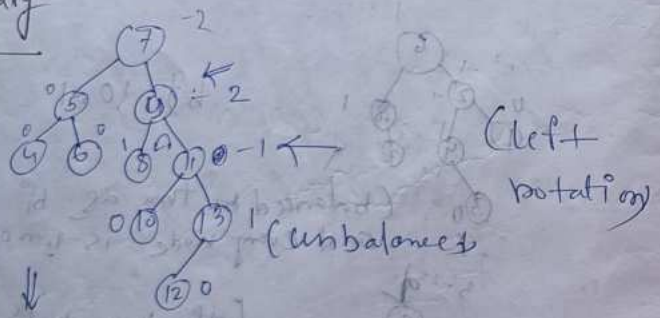
Case 1 -
Left Heavy tree

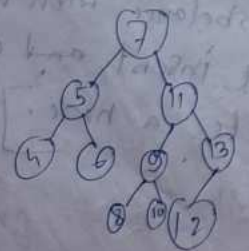


(Right rotation)



Case - 2
Right Heavy tree



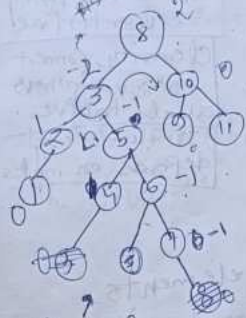


(Balanced)

(double notation)

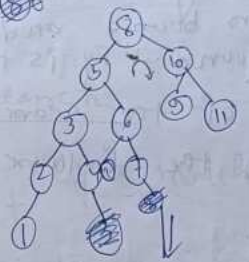
[+ve → L heavy
-ve → R heavy]

Case - 3



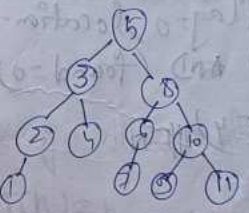
[2 means Left Subt. is heavy
-1 means the right of left sub tree is heavy]

(CL-R heavy tree)



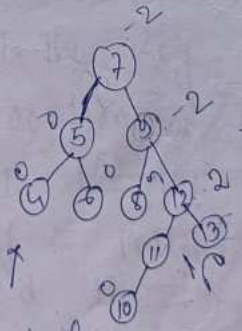
[Balanced - either tree diff of h. is -1 or 0 or 1]

[h = 2 diff unbalanced]



[-2 → R is heavy
2 → is Left is heavy]

Case 4:



(L-R rotation)

Analysis of Algorithm

Algorithm, Time - Space Comp, The characteristics of algorithm

Time Complexity
Space Complexity
→ quantitive

Clearity, Correctness, Shortness
qualitative

Time complexity depends on input size

* Linear search of ~~an~~ algo (Im)

Input: A ~~an~~ is an array with elements, L is the lower bound and U is the lower bound, Key is the element to be searched.

output: the index of the searched element or the failure message.

D'S; 1D Array

```
Step 1: ~~I=1 to n~~ flag=0, location=1
     2: WHILE (i<=U) AND (found=0) DO
     3: IF (A[i]=KEY) THEN
     4:    found=1
     5:    location=i
     6:    BREAK
     7:    ELSE
     8:    i=i+1
     9:    ENDIF
    10: ENDWHILE
    11: IF (found=0) THEN
    12: PRINT "SEARCH UNSUCCESSFUL"
    13: ELSE
    14: PRINT "SEARCH SUCCESSFUL: KEY IS
    15:    present at location, location
    16: ENDIF
    17: RETURN (location)
    18: STOP
```

* Linear Search

Case 1: The key matches with the first element

T(n) = the numbers of comparision

T(n) = 1

O(n) = 1 O(1)

Case 2: key does not exit

T(n) = n O(n)

Case 3: The key is present in any location of the array.

Let P_i be the probability that the key is present at ith position. To need the ith position we need i comparisions. ~~the expect~~

the expected number of

Comparision, $T(n) = \sum_{i=1}^{n} P_i \cdot i$

$= \frac{1}{n} \sum_{i=1}^{n} i$ ($\frac{1}{n}$ is constant, it is independent on [illegible])

$= \frac{n(n+1)}{2 \times n} = \frac{n+1}{2} = O(n)$

= O(n) the worst case of random [illegible]

Binary search

→ complexity $[O(\log n)]$

2	8	12	14	18	27	32	47	56	91
1	2	3	4	5	6	7	8	9	10

(mid = 5; l = 1, u = 10)

1st iteration

Step-1 = $mid = \lfloor \frac{l+u}{2} \rfloor = \lfloor \frac{1+10}{2} \rfloor = 5$ (floor value for int)

Step 2 If Key > A[mid], l = mid+1 = 6

or key < A[mid]

or key = A[mid]

2nd iteration

	27	32	47	56	91
	6	7	8	9	10

(l = 6, u = 10)

$mid = \lfloor \frac{6+10}{2} \rfloor = 8$

key < A[mid]

32 < 47

u = mid - 1 = 7

3rd iteration

			27	32			
			6	7			

(l = 6, u = 7)

$mid = \lfloor \frac{6+7}{2} \rfloor = 6$

Key > A[mid]

l = mid+1 = 7

4th iteration

$mid = \lfloor \frac{7+7}{2} \rfloor = 7$

A[mid] = key

Algorithm Binary search

Input: An array A[1 ... n] and K is the element to be searched.

Output: If search successful return index else return error message.

DS: 1D Array

Step 1: l = 1, u = n

2: flag = FALSE

3: WHILE (flag ≠ TRUE) and (l ≤ u) DO

```
4: mid = [(l+u)/2]
5: IF (K = A[mid]) THEN
6: PRINT "Successful"
7: flag = TRUE
8: RETURN mid
9: EXIT
10: ENDIF
11: IF (K < A[mid]) THEN
12: U = mid - 1
13: ELSE
14: l = mid + 1
15: ENDIF
16: ENDWHILE
17: IF (flag = FALSE) THEN
18: RETURN (-1)
19: PRINT "Unsuccessful"
20: ENDIF
21: STOP
```

Analysis of Binary search

At iteration 1 = Length of array = n

At iteration 2 = length of array = $\frac{n}{2}$

At iteration 3 = Length of array = $(n/2)/2 = \frac{n}{2^2}$

Therefore after k iteration $= \frac{n}{2^k}$ [k is the no of iterations]

we know after k iteration the length of array because 1.

Therefore,

$$\frac{n}{2^k} = 1$$

$$\Rightarrow n = 2^k$$

[As we are dividing the array and finally it's little size will = 1]

Applying log both side

$$\Rightarrow \log_2 n = \log_2 2^k$$

$$\Rightarrow \log_2 n = k * \log_2 2$$

As $(\log_2 2 = 1)$, Therefore $k = \log_2 n$

$$= O(\log_2 n)$$ ~~$O(\log(n))$~~

22/11/22

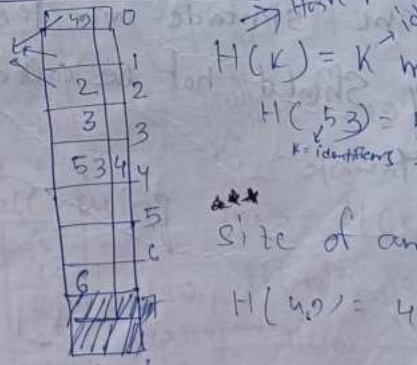
Hashing

Linear DS = $O(n)$ [+, -]

Nonlinear DS = $O(\log_2 n)$ [*, /]

Hashing = $O(1)$ [%]

Buckets



2 slots

Hash function → identifiers

$H(K) = K \bmod 7$ = size of array (m, table size)

$H(53) = 53 \bmod 7$ (K = identifiers)

$= 4$

Size of array should be = Prime

$H(49) = 49 \bmod 7$

$H(7) = 0$ = collision

Two different identifiers (keys) hashes to the same location.

H(2) = 2
H(3) = 3
H(6) = 6

A bucket may have 1 slot, 2 slot... the no of identifiers that hash to same loc.
the no. of salts should be equal having
~~If bucket has~~ when we are, one bucket and that bucket has 2 slots then we can't insert more that 2 elements = overflow.

If the bucket has more than 2 slots then it isn't overflow.

Characteristics of hash function

1) easy
2) the h.f. should insure all the identifiers are spreade over the buckets, they should not be collide in same place (evenly distributed (uniform distribution)

~~Proofer~~

Properties of hash function

1. They are collision-free.
2. They can be hidden. It should be difficult
3. to guess the input value for a hash function from its output.
3. It should have a bucket and every bucket should have some slots.
4. Here, all of the identifiers should be equally distributed.

Discuss 2 hash functions.

1) Division method, 2) mid square method, 3) folding method.

Different types of hash functions

1) Division method/Division modulo function

$H(k) = k \bmod H$ of indices start from 0

$= (k \bmod H) + 1$ if indices start from 1

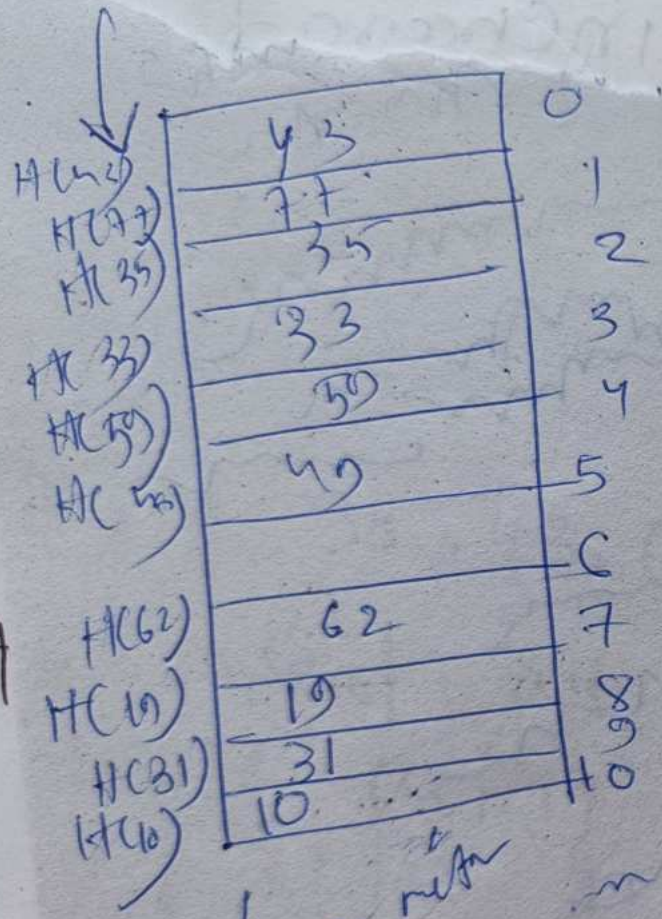
1) 10, 19, 35, 43, 62, 59, 31, 49, 77, 33 - try to insert all this elements using hash table using this - $H(k) = k \bmod 11$

77	0
43	1
~~25~~	2 H(35)
33	3 H(~~59~~) = (62 mod 11) / 7
59	4 H(49)
49	5
	6 H(62) = 62 mod 11 = 7
62	7
10	8 H(10)
31	9 H(10)

H(10) = 10 mod 11 = 10
H(19) = 19 mod 11 = 8
H(35) = 35 mod 11 = 3

Types of hash functions

(1) Division method/Division modulo function



of indices start

$H(k) = k \bmod 11$

$H(10) = 10 \bmod 11$

$H(19) = 19 \bmod 11$

$H(35) = 35 \bmod 11$

$H(43) = 43 \bmod 11$

$H(62) = 62 \bmod 11$

$H(59) = 59 \bmod 11$

$H(10) = 10 \bmod 11$

$H(35) = 35 \bmod 11$

• Division method - this is most simple and easiest method to generate a hash value. the hash func. divides the value k by m and then uses the remainder obtained.

Collision Resolution techniques

① open Hashing (chaining) - (Linear Probing)
② closed Hashing (Linear Probing)

The classification of closed hashing

① Linear Probing → search the array and put the element for which collision occurs in the 1st empty place

Resolve using closed hashing

21 mod 11
m 2 mod 11
43 mod 11
54 mod 11

77
21
32
43
54
10

Clustering (Primary clustering) the searching time is getting increased with the progression of linear p. here 1, 1+1, 1+2 ...

the Problem of linear Probing → clustering

② Quadratic Probing (Remedy)

the search time is = $i, i+1^2, i+2^2, i+3^2$ ...

the hash function for Quadratic Probing $(H(K) + i^2) \bmod 11$

It will search element by jumping (as $i, i+1^2, i+2^2$ ...) empty place. So the search time will reduced.

③ Double hashing (Remedy of Clustering)

Ex = $H(K_1) = K \bmod 11$ — ①
$H(K_2) = H(K_1) \bmod (K-4)$ — ②

Here we are using 2 diff hash functions. The advantage of using this methode is if collision occurs to generate hash table by using one hash function, we will use the another one.

④ Random hashing (Remedy of clustering)

$H(K) = (K+m) \bmod H$ [m is a random element]

A technique for randomizing the input to a cryptographic hash function.

① Resolve using open hashing

0
1
2
3
4
5
6

$H(K) = K \bmod 7$

In open hashing if the collision will occure in hash table then we will make a node of put the element for which the collision occurs in that node and link it that place where the collision got occured and this process will be continued and form a LL.

19 → 40 → 12

Disadvantage of this - time complexity O(n) as the linked list is img (linear d.s and t.c of l.d.s is O(n))

Linear Probing =

During the generation of hash table, if collision occurs we have to seach the array and put the element for which the collision occurs in the 1st empty space. By this method we can be solve the problem of collision during generation of hash table. But there is a disadvantage of linear Probing. The search time is getting increased with the progression of linear probing. This is called Clastering. so, the linear Probing is not efficient for collision resolution. The remedy of Clastering are, i) Quadratic Probing, ii) Double hasing, iii) Random hasing.

Disadvantag of this —

O(n) as the linked list is implemented

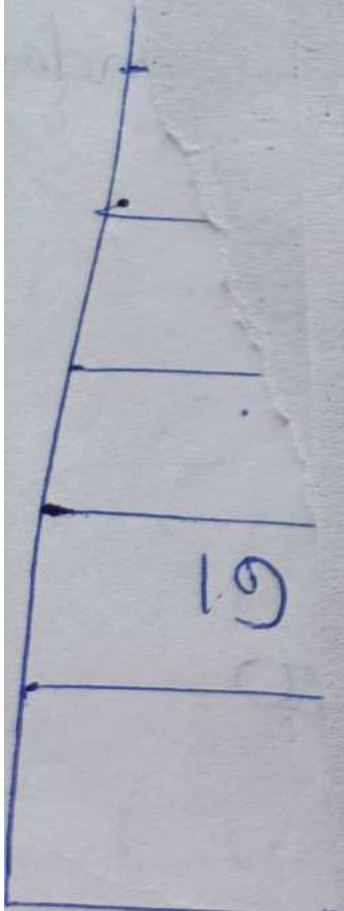
(linear d.s and it of d.s is O(n))

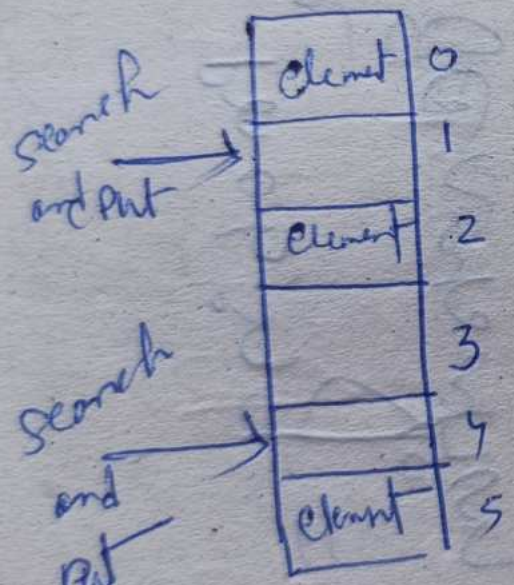
tion.

Eg:

Quadratic Probing

i = 0
i+1² = 1
i+2² = 4





od 7

hasing

sion the oc

ple then

a node

element

ch the Cal

in that

that Plac

② Folding method-

K = 15 27 32 43

H (K) = K1 + K2 + K3 + K4 → it is the index where we will put the K (= rec. hash va...)
= 15 + 27 + 32 + 43

51 + 27 + 23 + 43 ↓ shift folding

This method involves 2 steps -
i) Divide the key-value K into a numbers of parts i.e. K1, K2, K3 ... Kn, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
ii) Add the individual parts. The hash value is obtained by ignoring the last carry (if any.)

51 + 27 + 32 + 34 ↓ folding at the boundary

③ mid square methode

K = 4321 25 | K 21 1000
K^2 = 625 | K^2 = 1000000

this is my array index

The mid - square method is a very good hashing method. It involves two steps to compute hash value i) Square the value of the key K i.e. K^2
ii) Extract the middle r digits as the h. value.

Sorting

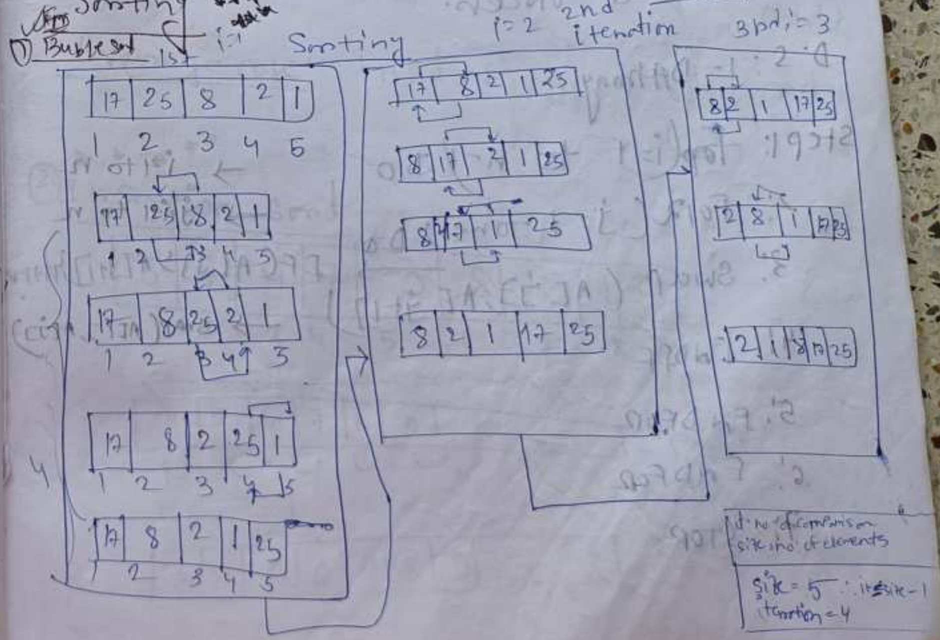
$H_1(K) = (K \bmod H) + 1$ (given) = 4 for 10
$H_2(K) = (K \bmod (H - 4)) + 1$ = 4 (given)
H = 11

10, 2, 5, 8, 11

$i = (H_1(K) + H_2(K)) \bmod H$ (given)
= (11 + 4) mod H
= 4
↓
this is our array index

24/11/22

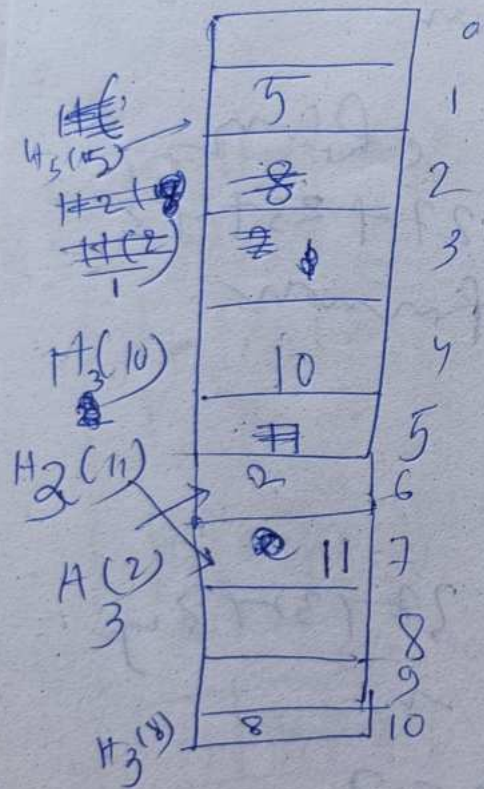
Sorting
1) Bubble sort



Double $H_1(K) = (K \bmod H) + 1$

$H_2(K) = (K \bmod (H-4)) + 1$

10, 2, 5, 8, 11



$H = 11$, size =

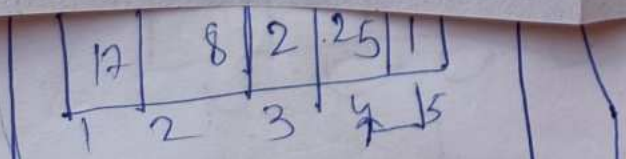
if 2 hash func is not working then

$H_3(K) = [H_1(K) + H_2(K)] \bmod H$

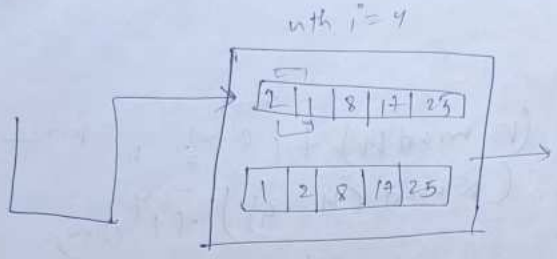
or something other formula that will be given in the paper

$= (6 + 5) \bmod 11$

$= 12 \bmod 11$



it: no of comparisons
size: no of elements



Algorithm Bubble Sort (Ascending order)

Input: Array A[1-n] where n is the number of element.

output: An array A with all elements in sorted order.

D.S: 1-D Array

Step 1: for(i=1 to n-1) DO → i=1 to n

2: FOR(j=1 to n-i) DO → j=1 to n

3: SWAP (A[j], A[j+1]) → IF(A[j]>A[j+1]) THEN swap(A[j], A[j])

4: ENDIF

5: ENDFOR

6: ENDFOR

7: STOP

Best case:- Input Array is already Sorted.

1	2	8	4	5

in figure short the sorted array is worst case

Time complexity = $O(n^2)$

$$C(n) = \sum_{i=1}^{\frac{n}{2}-1} (n-i)$$

$$= (n-1) + (n-2) + \dots + 2 + 1$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

we can refine the sorting by best case.

worst case: Input is reversely sorted. $O(n^2)$

Average case: Input is in random order. $= O(n^2)$

Insertion sort

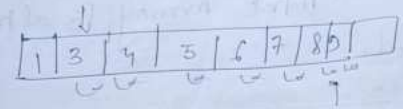
5	6	4	7	9	8	3	1	2
1	2	3	4	5	6	7	8	9

4	5	6	7	8	9

1st pass

3	4	5	6	7	8	9

2nd pass



for each of comp. show the array

[No. of comparison = no. of elements (n) - 1]

Algorithm Insertion sort

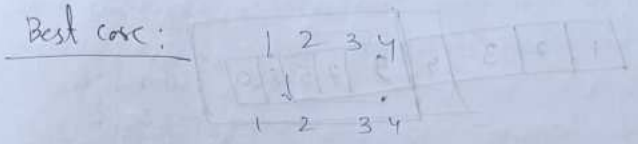
I/P: Array A[1 — n] where n is the number of element

O/P: An array A with all elements in sorted order.

DS: 1-D array

step 1: FOR (i=1 to n) DO
2: key = A[i]
3: j = i - 1
4: WHILE (j >= 0 AND A[j] > key) DO
5: A[j+1] = A[j]
6: j = j-1
7: END WHILE
8: A[j+1] = key
9: END FOR

Best case:



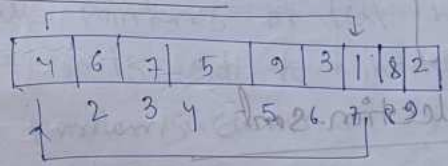
1 + 1 + 1 — n times
= O(n)

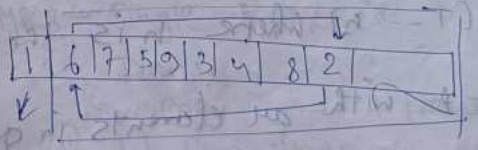
worst case: Reversed array
O(n²)

time C = no. of comparisons

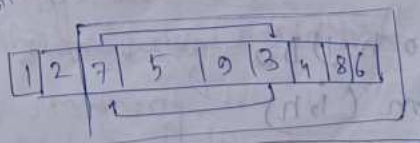
Average case: O(n²)

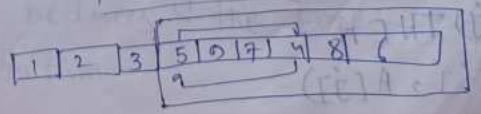
③ Selection sort





don't consider that position





1: FOR(i=1 to n-1) DO

2: j = i - 1

3: WHILE (j >=0 AND A[j]>A[j]) DO

4: TEmp = A[j]

5: A[j] = A[i]

6: A[i] = TEmp

7: j = j-1

8; ENDWHILE

9; ENDFOR

10; STOP

Size = 5

2	3	1	5	4
0	1	2	3	4

insertion sort

4	1	3	2	5
0	1	2	3	4

Size = 5 = n

iterations should be ≤ 4

∴ Size = no. of elements

iterations = no. of comparisons should be ≤ 4

i = 1, j = i - 1 = 0

1st it =

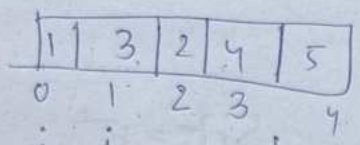
1	4	3	2	5
0	1	2	3	4

1	3	4	2	5
0	1	2	3	4

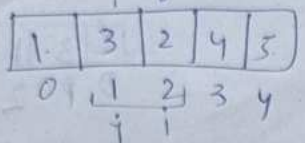
1	3	2	4	5
0	1	2	3	4

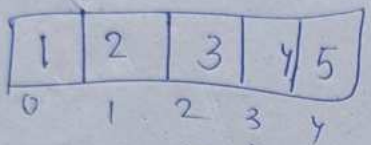
[For bubble & insertion sort & selection sort]

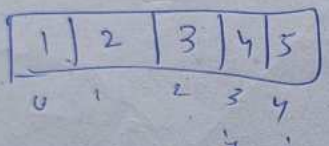
4 (size 5)



2nd it =



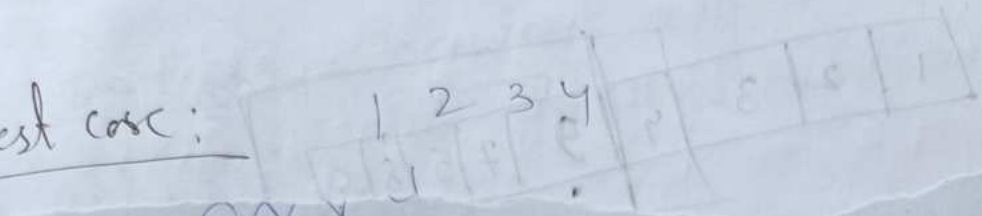


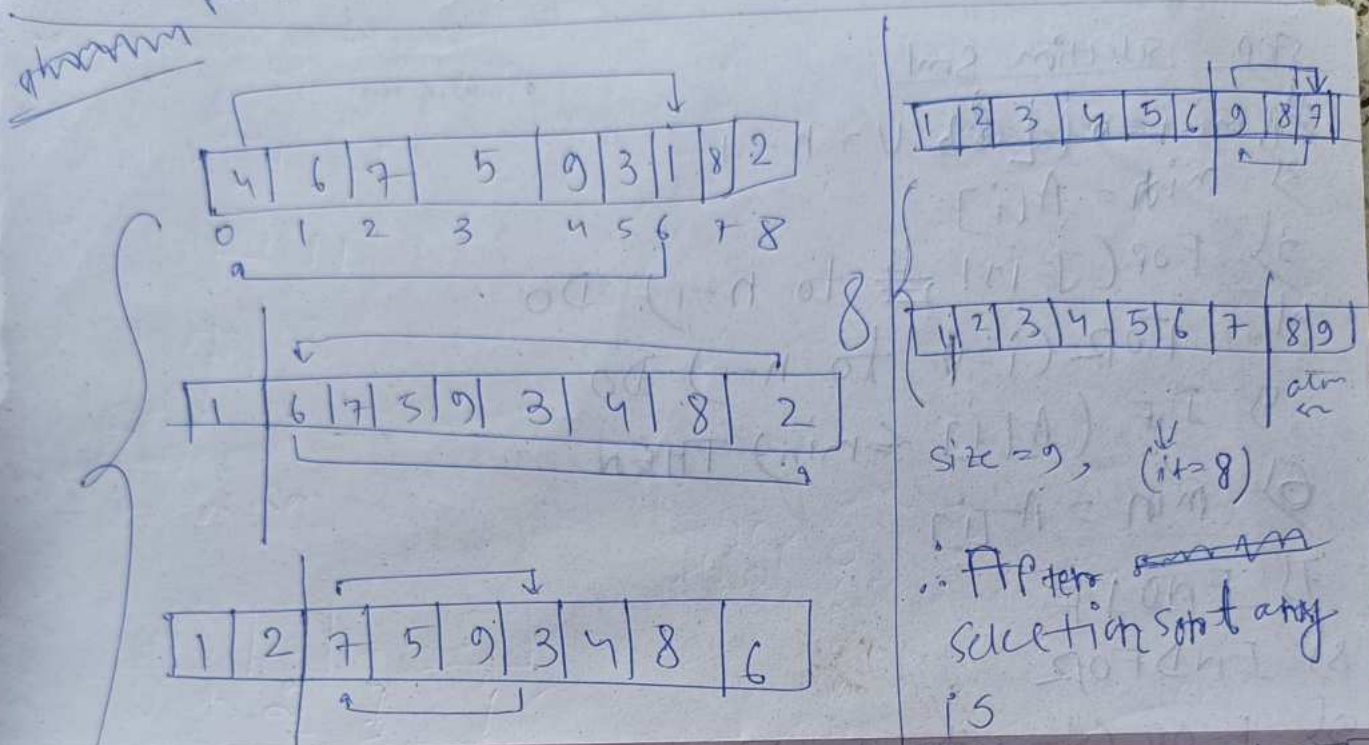


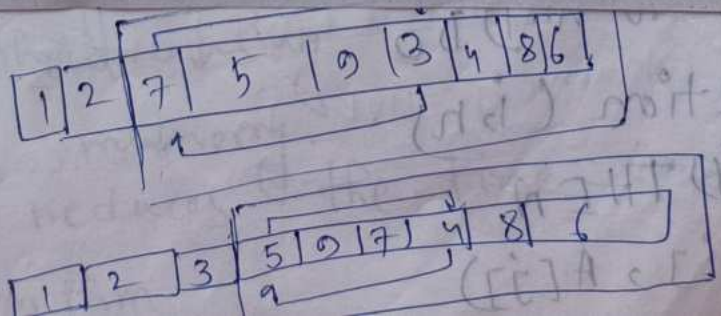
4 (Site 5)

Any Sorted
by insertion Sort

Best case:


1 2 3 4


4 6 7 5 9 3 1 8 2
0 1 2 3 4 5 6 7 8
1 6 7 5 9 3 4 8 2
1 2 7 5 9 3 4 8 6
1 2 3 5 9 7 4 8 6
1 2 3 4 9 7 5 8 6
1 2 3 4 5 7 9 8 6
1 2 3 4 5 6 9 8 7
1 2 3 4 5 6 7 8 9
size = 9, (it = 8)
∴ After selection sort array is
1 2 3 4 5 6 7 8 9


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

STEP Selection Sort

~~ ~~ i=L=0, U = n-1

1) min = A[i]

2) FOR(J=i+1 to n-1) DO

3) FOR (i=0 to n-1) DO

4) IF (A[i] <=min) THEN

5) min = A[i]

6) END IF

7) ENDFOR

8) FOR (i=0 to n-1) DO

9) ~~FOR (j~~ IF (A[i] ≠ min) THEN

10) TEmp = A[i]

11) A[i] = min

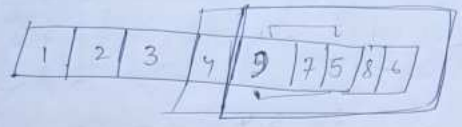
12) min = TEmp

13) ENDIF

14) ENDFOR

15) ENDFOR

16) STOP



1 2 3 4 5 | 7 9 8 6

1 2 3 4 5 6 | 9 8 7

1 2 3 4 5 6 7 8 9

Best case
Worst case
Average case
} $= O(n^2)$

Algorithm selection-sort

I/P: Array A[1 -- n] where n is the number of element

O/P: An array A with all elements in sorted order

DS: 1-D array

Step 1: FOR (i=1 to n-1) DO
2: j = selection (i, n)
3: IF (i ≠ j) THEN
4: SWAP (A[i], A[j])
5: ENDIF
6: END FOR
7: STOP

selection (i, n) - done by yourself
↓
(find the minimum element in array)
(done)

Appendix - A
time complexity to quantitative

(1) Threaded binary tree: it is a simple binary tree but they have a speciality that null pointers of leaf node of the binary tree is set to inorder predecessor or inorder successor.

(2) Types - Single threaded binary tree
Double threaded binary tree (inorder & preorder)

(3) Imp: It can traverse the tree very fastly without using additional data structure or memory.

(4) It reduces the time complexity of algorithm.

③ operation like traversal, insertion, deletion can be done without using recursive algorithm.

④ It becomes very easy to find out inorder successor of any node 'n' in a threaded binary tree.

④ what is dummy node — A dummy node is a node that will not contain any usable value but will always carry the location of the front of the list. A dummy node also means that the linked list is circular.

⑤ why dummy node is required in t.b.t — As we saw that reference left most reference and right most reference pointer has nowhere to point to, so we need a dummy node. and this node will always present even when tree is empty.

⑥ AVL tree — AVL tree is invented by GM Addson-Velsky and EM Landis in 1962. The tree is named AVL in honour of inventors.

AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
Tree is said to be balanced if balance factor of each node is in between -1 to 1.

⑦ types of rotation on AVL tree — (for balancing the tree)
1) Left rotation
2) Right rotation
3) Left-right rotation
4) Right-Left rotation.

⑧ Imp of AVL tree —
AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a b.s.t. of height h is O(h). However, it can be extended to O(n) of the BST becomes skewed (i.e. worst case)

⑨ Data Structure - Data structure is the way, the data is organized & manipulated. It seeks to find ways to make data access more efficient. (Data. S. is a way of organizing & manipulating the data in such a way so that we can easily access manipulate, retrive, store the data, i.e ~~for more~~ So that we can access the data more efficiently.

Previous year Qn

DSA

Qn-1

1) Why is Queue data structure called FIFO?

The Queue data structure is called FIFO or first in first out data structure as the first element inserted in the queue is the first element to be deleted. The insertion of an element on a queue is called an enqueue operation and the deletion of an element is called a dequeue operation.

2) What is ADT?

In computer science, an abstract data type (ADT) is a mathematical model for data types. An abstract data type is defined by its behaviours (semantics) from the point of a user, of the data. It is just opposite of built in data type. It is also called user defined data type, as the computer doesn't know the behaviors of ADT. Ex - Stack, Queue, Tree etc.

3) which D.S is used to perform recursion? why?

Stack is used to perform recursion. Because of its LIFO property it remembers its caller so knows whom to return when the function has to return. [Recursion makes use of system stack for storing the return addresses of the function calls. Every recursive function has its equivalent iterative (non-recursive) function.]

4) Define binary tree.

A b.t is a collection of finite set of one or more nodes such that -

1) root (specially designated node).
2) Remaining nodes are partioned into 2 disjoin subsets, where each of the subsets are a tree.

5) Arrange the given array using bubble sort {12, 4, 5, 10, 1}.

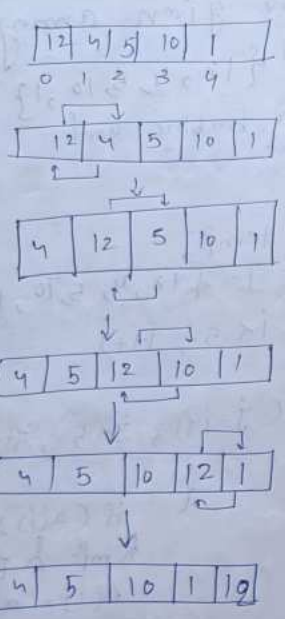
Programme for bubble sort -

```
#include<stdio.h>
int main()
{
    int i, j, temp;
    int a[] = {12, 4, 5, 10, 1};
    for(i=0; i<5; i++)
    {
        for(j=i+1; j<5; j++)
        {
            if(a[i]>a[j])
            temp= a[i];
            a[i]= a[j];
            a[j]= temp;
        }
    }
    printf("Sorted array \n");
    for(i=0, i<5, i++)
    {
        printf("%d\n", a[i]);
    }
return 0;
}
```

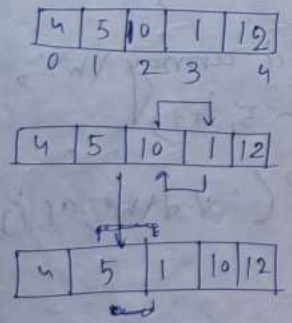
O/P = {1, 4, 5, 10, 12}

STEPS:A

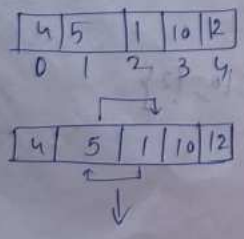
1st iteration

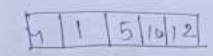


2nd iteration

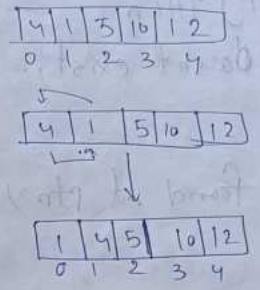


3rd iteration





4th iteration



Qn-B

1) Formulate an algorithm to search a node in b.s.t.

I/P: ITEM is the data ## to be searched.

O/P: If found then Pointer to the node or a message.

D.S: Linked list representation of B.T.

Step 1: ptr = ROOT, flag = FALSE

2: WHILE (ptr ≠ NULL) AND (flag = FALSE) DO

3: Case: ITEM < ptr → DATA

4: ptr = ptr → LCHILD

5: Case: ITEM = ptr → DATA

6: flag = TRUE

7: case: ITEM > ptr → DATA

8: ptr = ptr → RCHILD

9: ENDCASE
10: ENDWHILE
11: IF (flag = FALSE) THEN
12: PRINT "ITEM doesn't exist";
~~13: EXIT~~
13: ELSE
14: PRINT "(ITEM found at ptr")
15: ENDIF
16: STOP

2) write an algo to search a particular data in SLL.

I/P: HEADER is the pointer to the header node, KEY is the element to be searched.

O/P: The location of the KEY or a failure message

D.S. - A single LL.

Step 1: i=1, found=0, location
2) ptr = HEADER → LINK
3) WHILE (ptr ≠ NULL) AND (found=0) DO
4) IF (ptr → DATA = KEY) THEN
5) found = 1
6) Location = i
7) BREAK
8) ELSE
9) i = i+1
10) ptr = ptr → LINK
11) ENDIF
12) ENDWHILE
13) IF (found = 0) THEN
14) PRINT ("KEY not found")
15) ELSE
16) PRINT ("KEY found at location", location)
17) RETURN (location)
18) STOP

3. Convert the following infix expression to postfix expression using stack.

$(A+B) * C - (D - E) / (F+G)$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Read symbol	Stack	O/P
Initial	(	
1.	((	
2.	((+	A
3.	((+	AB
4.	(	AB+
5.	(*	
6.	(*	AB+C
7.	(-	AB+C*
8.	(-(	AB+C*

9.	(-(-	AB+C+D
10.	(-(-	AB+C+DE
11.	(-	AB+C+DE--
12.	(-/	AB+C+DE--
13.	(/C	AB+C+DE-/-
14.	(/C+	AB+C+DE-/-F
15.	~~(~~C/C+	AB+C+DE-/-FG
16.	C	AB+C+DE-/-FG+/

O/P

After converting the infix exp. into postfix we get - AB+C+DE-/-FG+/

4) Write the binary search algorithm and give its time complexity.

I/P: An array A[1 — n] with elements and KEY is the element to be searched.

O/P: the location of KEY or a failure message.

D.S: 1D array.

Step 1: l=1, u=n, 2: flag=FALSE

3: WHILE (l<=u) AND (flag=FALSE) DO

3: mid = $\lfloor \frac{l+u}{2} \rfloor$

4: IF (KEY = A[mid]) THEN

5: PRINT("search successful KEY found at loc: mid)

6: ~~ELSE~~ flag = TRUE

7: RETURN (mid)

8: ENDIF

9: IF (KEY < A[mid]) THEN

10: u = mid - 1

11: ELSE

12: l = mid + 1

13: ENDIF

14: ENDWHILE

15: IF (flag = FALSE) THEN

~~PRINT~~

16: RETURN (-1)

17: PRINT ("search unsuccessful")

18: ENDIF

~~19~~: STOP te- O(log n) in O($\log_2 n$)

5) Define recursion. Write a recursive function to reverse a string.

A function is called recursive function, if the statement within the body of that function calls the

Same function again and again. ~~It is~~ The recursive function is a function which is partially defined by itself. It is ~~also~~ also called 'circular definition'. It uses selection structure. It make the code smaller. Infinite recursion causes the crash of memory. It uses more memory than iteration. It terminates when the base case is recognited.

```
#include<stdio.h>
void reversal (char *str)
{
  if (*str)
  {
    reversal (str+1);
    Printf ("%c", *str);
  }
}
int main ()
{
  Char a[] = "Geeks for Geeks";
  reversal (a);
  return 0;
}
```

5) Differentiate single linked list and doubly linked list state advantages of doubly linked list over single linked list.

Singly Linked List	Doubly Linked List
1) A singly linked has node's with a data field and a next field.	1) A Doubly Linked List has a previous link field along with a data field and a next link field.
2) In a singly linked list, the traversal can only be done using the link of the next node.	2) In a Doubly linked List, the traversal can be done using the next node link as well as the previous node link.
3) A singly linked list occupies less memory than the Doubly Linked List, as it has only 2 fields.	3) A Doubly linked List occupies more memory than the singly linked list is as it

has 3 fields.

4) Accessing elements in a singly linked list is less efficient when compared to a Doubly linked list, as only forward traversal is possible

4) Accessing elements in a doubly ll is more efficient when compared to a singly linked list as both forward and backward traversal is possible.

5) The time complexity of inserting or deleting a node at a given position (if the pointer to that position is given) in a singly linked list is O(n).

5) The time complexity of inserting or deleting a node at a given position (if the pointer to that position is given) in a doubly linked list is O(1).

6) Singly linked list is preferred when we have memory limitation (we can't use much memory) and searching is not required.

6) Doubly linked list is preferred when we don't have memory limitation and searching is required (we need to perform search operation on the linked list.

Advantages of DLL:

As with a singly linked list, it is the easiest data structure to implement.

The traversal of d.l.l is bidirectional which is not possible in a singly l.l.

Con-c Deletion of nodes is easy as compared to a singly linked list.

1) Suppose the following eight numbers are inserted in order into an empty binary search tree T: 50, 33, 44, 22, 77, 35, 60, 40.

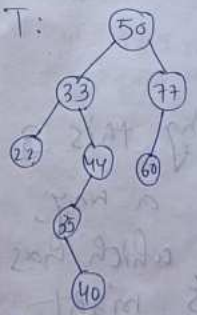
i) Draw the tree T.

ii) Write the inorder, preorder and postorder traversals for the tree T.

iii) Delete 33 from the tree. Show the resulting tree.

iv) Write the algo for inorder traversal of a tree.

i) T:



ii) Inorder traversal: 22 33 35 40 44 50 60 77

Preorder traversal: 50 33 22 44 35 40 77 60

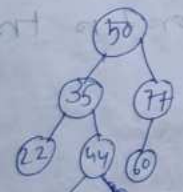
Postorder traversal: 22 40 35 44 33 60 77 50

(ii) 33 is a node with 2 child, to delete 33 we have to follow some steps -

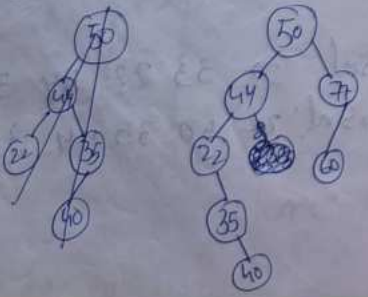
1) Inorder traversal -
22 33 35 40 44 50 60 77
We have to delete/remove 33 by its inorder successor (i.e 35).

2) we have to overwrite the inorder successor of 33.

3) we have to redraw the tree T



(iii) another way, instead of doing this 3 steps we can also apply a way, we can replace the node which has to be deleted with its right child, then the tree will be like,



as, there is no right child of 44, so the RC of 44 will remain NULL and 35, who will sit like this in the tree.

(iii) I/P: ROOT is the Pointer to the Root node of the tree.

O/P: The tree is in inorder fashion / Print all nodes of the tree in inorder fashion.

D.S: linked representation of B.T

Step 1: ptr = ROOT
2; IF (ptr ≠ NULL) THEN
3; Inorder (ptr → LC)
4; PRINT (ptr → DATA)
5; Inorder (ptr → RC)
6; ENDIF
7; STOP

Q2) what are the drawbacks of using sequential storage to represent stacks? Describe the linked representation of stack. write an algo for PUSH operation on stack using L.L.

I/P: ITEM is the new element to be inserted.

(o/p: The stack with newly inserted element at top.

P.S: A single l.l. with the a pointer TOP. STACK_HEAD is the header to the header node and TOP is denoting the first element.

STEP 1: new = GETNODE(NODE)

2: new → DATA = ITEM

3: new → LINK = TOP

4: STACK_HEAD → LINK = new

5: TOP = new

6: STOP

Drawbacks -

(Same for queue)

1) Sequential representation or array representation is not so efficient to represent a stack because the size of array is fixed, there static memory allocation happens, we cannot allocate memory during runtime.

2) It takes more time while dft is the time for execution (i.e. it is very slow)

3) It cannot store memory anywhere in the memory zone.

4) It can only stores data, it don't have any links to another nodes.

(same for queue)

Linked representation of stack -

In case of linked representation, there happens dynamic memory allocation. So we can allocate memory whenever we required. It can stores d memory randomly or we can say anywhere in the memory zone which is very useful. It takes less time for execution. Here in the linked representation, there is no condition for the stack to be full. we can use as much memory we have required, i.e. we can allocate required no of nodes. (In stack it has one pointer named top) The condition for the stack to be empty is TOP = NULL, (where TOP is a pointer to the 1st element of the stack). Here in the linked representation It can store both data and link to the another nodes.

[Added - for queue the queue has 2 pointers, they are front & rear. The condi: for the queue to be empty is front = NULL]

3. Write an algo for inserting a node in a doubly l.l. and deleting a node from a doubly l.l. consider all cases.

done

4) (i) write algo for inserting an element into a queue and deleting an element from a queue. what is circular queue?

done

Circular queue (ring buffer) - A circular queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle. In linear queue if we don't use all of the spaces, but it satisfies the condition ~~for the queue to be full~~ (rear=size) we will declare that the queue is full, but practically it isn't. [ ] For this the spaces are wasted. Circular queue is the remedy of that. If the queue staisfies the condition (rear mod size) + 1 = front, it means the queue is full and no spaces are in waste.

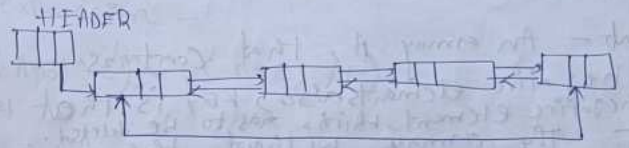
5) write short notes on

i) AVL tree - done

ii) Circular Doubly linked list -

A circular doubly l.l or circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as to the previous node in the sequence. the difference between a doubly and circular doubly linked list is as same as that exist between singly and circular l.l. [the circular d.l.l. doesn't contain NULL in the previous field of the first node of the list, i.e. & HEADER. Similarly the previous field of the first field stores the address of the last node.]

A circular d.l.l. is shown in the fig



* the C.d.l.l doesn't contain NULL in any of node. the last node of the list contains the address of the first node of the list. the first node of the list also contain address of the last node in its previous pointer.

DSA Internal 9m

1) Write an algo/pseudocode to delete the first negative element from an array A. Assume A containing negative elements.

Input – An array A, that contains negative elements also, KEY is that 1st negative element, which has to be deleted.
O/P – The array without the first negative element.

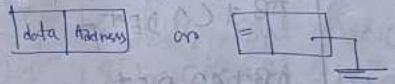
D.S – 1D array.

```
STEP 1: i = L, found=0, location=-1
     2: WHILE (i<=U) AND (found=0) DO
     3: IF (A[i] == KEY) THEN
     4: found = 1
     5: location = i
     6: BREAK
     7: EXIT
     8: ELSE
     9: i = i+1
    10: ENDIF
    11: ENDWHILE
    12: i = location
    13: WHILE (i < U) DO
    14: A[i] = A[i+1]
    15: i = i+1
    16: ENDWHILE
    17: A[U] = NULL
    18: U = U-1
    19: STOP
```

2) How will you represent the Header node in a linked list? Trace the steps to converting the given infix expression to postfix expression –

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

Header node is a segment of ~~th~~ a l.l. It has two parts. one part contains data and another part contains link to the another node / address of another node. If we remember the address of the header node, we can know the info about the whole linked list.



Header node

((A+B)^C) - (C D * E) / F)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Read Symbol	stack	o/p
~~initial~~	(	
1)	((	
2)	((	A
3)	((	AB
	CC+	
	CC+	

4]	(	AB+
5]	(^	AB+
6]	(^	AB+C
7]	(-	AB+C^
8]	(-(	AB+C^
9]	(-(	AB+C^D
10]	(-(+	AB+C^D
11]	(-(+	AB+C^DE
12]	(-	AB+C^DE+
13]	(-/	AB+C^DE+
14]	(-/	AB+C^DE+F
15]		AB+C^DE+F/-

∴ the result of infix to postfix conversion is - AB+C^DE+F/-

3] Assume that a stack is represented using a linked list. write the algo for the following operations (i) Push (V), Pop

Done

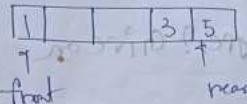
4] write an algo/pseudocode to count the nodes containing in a S.L.L.

done

5] what is DEQUEUE? what are the disadvantages of representing? How are they overcome?

Done

the disadvantages of representing-



In the linear queue if we don't use all the spaces of the queue and the queue satisfies the condition rear = size, then we have to declear the queue to be full. But practically it isn't. It is basically the wase of spaces. This the the disadvantage of ~~wsin~~ the array/linear representation of queue.

Remedy - This Problem can be overcomed by using circular queue. If the C.Q. satisfies the condition ~~from~~ (rear % MOD) + 1 = front then the queue must have to all the spaces of it be filled, i.e the queue is full no spaces are in waste.

6) Write an algo / pseudo code to the below operation i) insertion at any position in a doubly d.l. ii) Deletion from end of a doubly d.l.
Done.

Previous year QP (2020)

DSA

Gp-A

1) What is an Abstract Data type? Give an example.

Done.

2) What are the applications of stack and queue.

the applications of stack -

i) Execution of Recursive Programme (details done)

ii) Evaluation of Arithmatic expressions (details done)

iii) Binding Rules [static binding, dynamic binding]

the applications of queue -

i) multiprogramming environment -

ii) Resource sharing

iii) Processor sharing

[Static binding - the binding which can be resolved at compile time by the compiler. Dynamic binding - when type of the object is determined at run-time, it is known as D.B.]

3) Distinguish between a linear and non-linear data structure.

Linear D.S	Non linear D.S
1) Defination	1) Defination [In a non-linear d.s data elements are attatched in hierarchically manner]
2) TC = O(n)	2) TC = O(log n)
3) Here, single level is involved.	2) Here multiple levels are involved.
3) It's implementation is easy as compared to non-linear d.s.	3) It's implementation is complex as compared to ~~lin~~ linear d.s.
4) Here, data elements can be traversed in a single way or run only.	4) Here, data elements can't be traversed in a single run, they can be ~~runned~~ traversed in any way / run.
5) Here, memory is not utilized in an efficient way. (takes much memory)	5) Here, memory is utilized in an efficient way. (takes less memory)
6) Ex - array, stack, queue, linked list etc.	6) Ex - Tree, graphs, maps etc.

4] Compare and contrast Greedy Algorithm and dynamic programming.
not in syllabus.

5] What is the disadvantage of a single linked list? How can it be solved?
done

~~AB-S Gr-B~~

1] (i) Prove that maximum number of nodes on level i of a binary tree is 2^i ($i \geq 0$)

Proof

Basis - Root is the only node at level 0.

So, max. no. of nodes possible at level 0 is $2^0 = 1$.

Hypothesis: suppose for all k ($k \leq i$) the above formula is true i.e, the max possible no. of nodes for level k is 2^k.

Induction step - since each node can have the max degree 2, so the no. of nodes possible at level $k+1$ is-

$2^k \times 2 = 2^{k+1}$.

The above formula is true for level $k+1$.

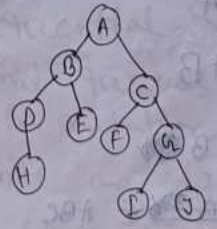
∴ the max no. of nodes on level i of a b.t is 2^i ($i \geq 0$) [k+1-i]

(ii) Prove that the maximum no. of nodes in b.t. of height/depth K is $2^K - 1$.
done [k will be replaced by K]

2] An initial array is given as
25 57 48 37 12 92 86 33.
Perform quick sort procedure and produce the array in an ordered form.
not in syllabus.

3] Write an algorithm to reverse the direction of all the links of a s.l.l.
done

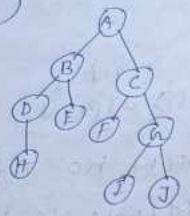
4] (i) Trace the given tree using Inorder, Preorder and Postorder Traversals.



(ii) Write an algorithm to traverse a tree using Preorder.

(ii) done

(i)



Preorder – A B D H E C F G I J
Inorder – D H B E A F C I G J
Postorder – H D E B F I J G C A

5) consider the following infix expression

A + B * (C + D) / F + D * E ;

Convert the above expression in postfix form using stack.

Read symbol	Stack	o/p
initial		
1)	(	
2)	(+	A
3)	(+*	AB
4)	(+*(	AB
5)	(+*(+	~~AB*C~~ ABC
6)	(+*(+	~~AB*C+D~~ ABCD
7)	(+*	ABCD+
8)	(/	ABCD+*+
9)	(/	ABCD+*+F
10)	(+	ABCD+*+F/
11)	(+*	ABCD+*+F/D
12)	(+*	ABCD+*+F/DE
13)	(	ABCD+*+F/DE*+

∴ the result is – ABCD+*+F/DE*+

6) write the binary search algorithm and give its time complexity.

done

Ch-6

1) write algorithm for inserting an element into a circular queue and deleting an element from circular que.

done

2) what are the drawbacks of using sequential storage to represent stacks and queues? Describe the linked representation of stacks and queues. write an algorithm for PUSH operation on stack using linked list.

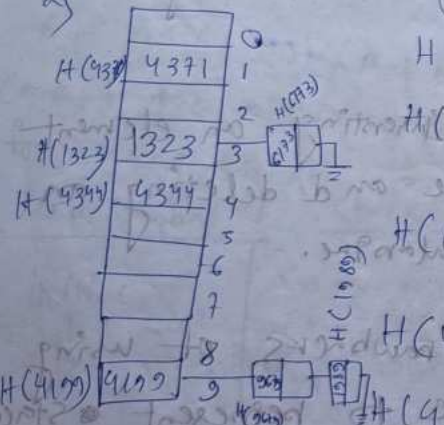
done

a) (i) Given the input: {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \bmod 10$ show the resulting:

a) separate chaining hash table
b) open Addressing Hash table using Linear Probing.
c) open Addressing Hash table using Quadratic Probing.

Separate chaining hash table

a)



$H(4371) = 4371 \bmod 10$
$= 1$

$H(1323) = 1323 \bmod 10$
$= 3$

$H(6173) = 6173 \bmod 10$
$= 3$

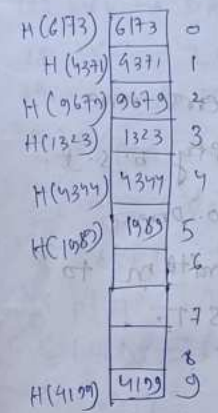
$H(4199) = 4199 \bmod 10$
$= 9$

$H(4344) = 4344 \bmod 10$
$= 4$

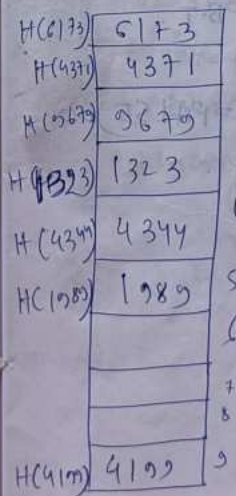
$H(9679) = 9679 \bmod 10$
$= 9$

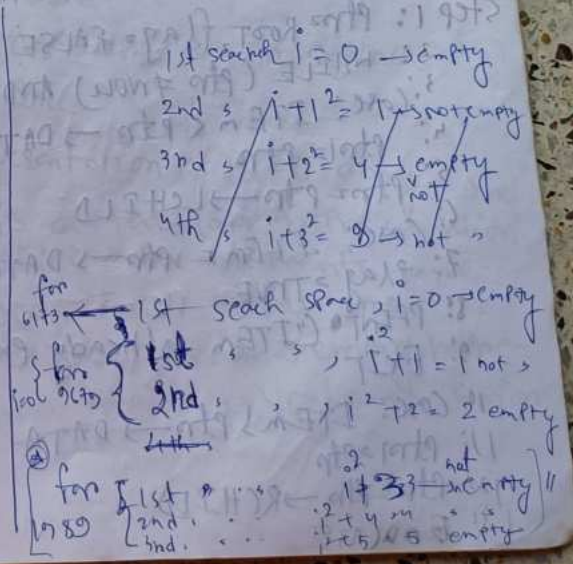
$H(1989) = 1989 \bmod 10$
$= 9$.

b) Open Addressing Hash table using Linear Probing



c) Open Addressing Hash table using Quadratic probing.





(ii) What are the properties of good hash function?

done

4] Suppose the following numbers are inserted in order an empty b.s.t.
T: 50, 33, 44, 22, 77, 35, 69, 40. Draw the tree T. Formulate an algorithm to insert an element in a B.S.T.

Done

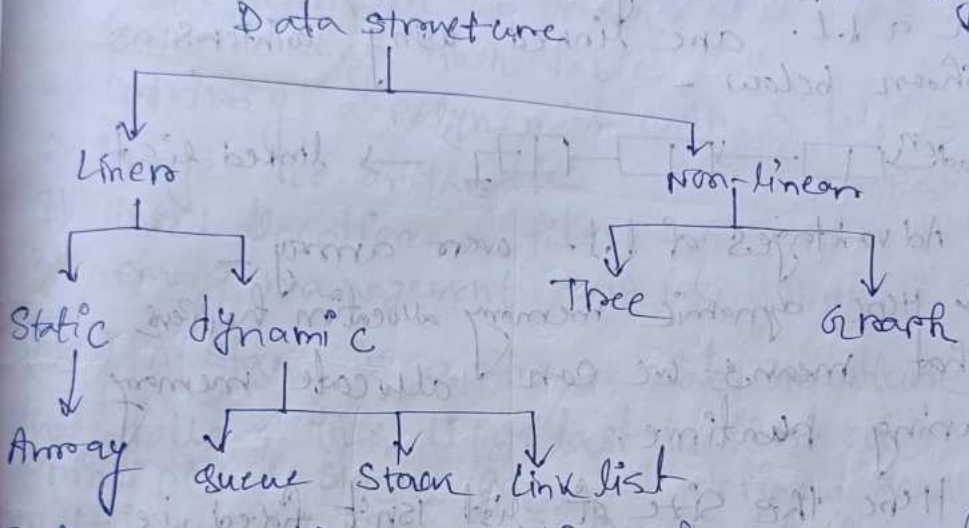
I/P: ITEM is the data to be inserted.
O/P: If there is no ~~data~~ node containing ~~item~~ ITEM, it is inserted else a message.
D.S- Linked representation of B.T

```
Step 1: Ptr=ROOT, flag=FALSE
     2: WHILE (Ptr ≠NULL) AND (flag=FALSE) DO
     3: Case: ITEM < Ptr → DATA
     4:    Ptr1=Ptr
     5:    Ptr=Ptr→LCHILD
     6: Case: ITEM = Ptr→DATA
     7:    flag=TRUE
     8:    PRINT "ITEM already exists"
     9:    EXIT
    10: Case: ITEM > Ptr→DATA
    11:    Ptr1=Ptr
    12:    Ptr=Ptr→RCHILD
    13: ENDCASE
    14: ENDWHILE
    15: If(Ptr=NULL) THEN
    16:    new=GETNODE(NODE)
    17:    new→DATA=ITEM
    18:    new→LCHILD=NULL
    19:    new→RCHILD=NULL
    20:    If(Ptr1→DATA<ITEM) THEN
    21:       Ptr1→RCHILD=new
    22:    ELSE
    23:       Ptr1→LCHILD=new
    24:    ENDIF
    25: ENDIF
    26: STOP
```

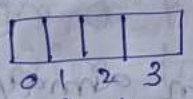
5] Write short notes on:
(i) AVL trees — Done
(ii) Divide and conquer - not in syllabus
(iii) Recursion - Done
(iv) memory representation of a 2D array - Done

A 2D array is a tabular representation of data in which the elements are kept in rows and columns.
A 2D array has 2 types of memory representation: 1] row major, 2] column major.

① A data structure is a storage that is used to store & organize data. It is a way of arranging data on a computer so that it can be accessed & updated efficiently.



② An array is a collection of or group of similar datatypes or data items or elements. It has contiguous memory locations. Here static memory allocation happens.



③ A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays is generally stored in row-major order in the memory. The general form of declaring N-dimensional arrays is shown below -

```
data_type array_name[size1][size2]...[sizeN];
```

④ A L.L. is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a L.L. are linked using pointers as shown below -

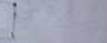
→ linked list

⑤ Advantages of L.L. over array -

1) Here dynamic memory allocation happens that means we can allocate memory during runtime.

2) Here the size of list isn't fixed, i.e we can allocate as much nodes that are returned at any time. no of

3) L.L can store ~~memory~~ anywhere in the memory. into.

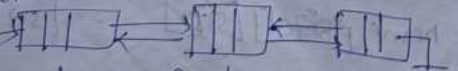


4) It is more efficient than array.

5) Here insertion & deletion operation is faster.

6) A D.L.L is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list & also the data.

Header

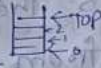

→ D.L.L

Advantages of D.L.L over S.L.L - In copy

⑦

1) Implementing Stacks
2) Queues using linkedlist
3) Implementation of graphs
4) Implementing hash tables
5) Portray a polynomial with a L.L:
6) large no. arithmetic
7) linked allocation of files
8) memory management with L.L.

⑧ A stack is a linear data structure that follows the LIFO (last in first out) principle. A stack can be defined as a containers in which insertion & deletion can be done from the one end known as the top of the stack.



Applications of stack -

- Execution of recursive programme
- Evaluation to arithmetic expressions
- Binding rules (static binding & dynamic binding)

Basic operations performed on stack -

1) push
2) pop
3) isEmpty
4) isFull
5) top (Display the topmost element of the stack)

⑨ A Queue is defined as a linear data structure that is open at both ends & the operations are performed in first In First Out (FIFO) order. we define a queue

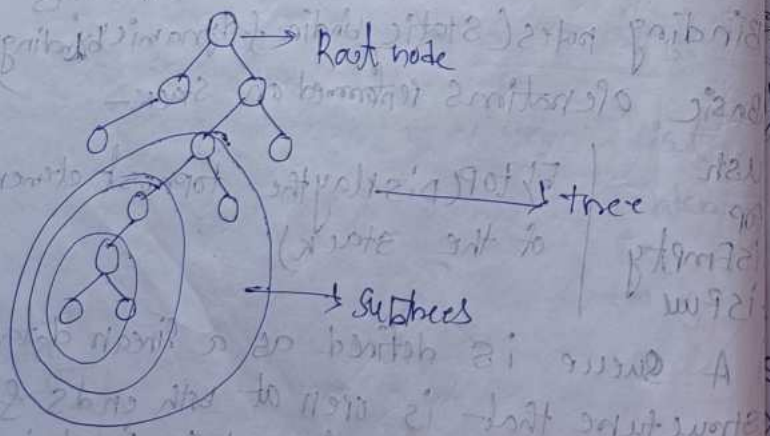
→ Insert the node in increasing order.

to be a list in which all additions to the list are made at one end, & all deletions from the list are made at the other end.

11) Basic operations on queue -

1) enqueue
2) dequeue
3) peek
4) isFull
5) isEmpty

12) A tree is a non-linear abstract data type with a hierarchy based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node & has subtrees connected to the root.



13) Basic terminology in tree -

1) Root
2) Node
3) Parent
4) Sibling
5) Edge
6) Child
7) General tree (tree)
8) Left child
Leaf node
9) Subtree
10) Binary search tree
11) Height
12) Leaf
13) Degree
14) Depth
15) Level
16) NUL tree
17) Binary tree
18) path
19) spanning tree
20) Heap
21) Descendent (Successor node)
22) Ancestor (Predecessor)

14) Binary trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree & right subtree. Based on the no. of children, binary trees are divided into 3 types - full binary tree, complete binary tree & perfect binary tree.

15) Tree traversal is a process to visit all the nodes of a tree & may print their values too.

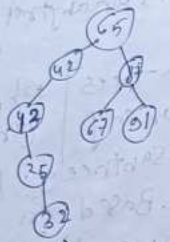
tree data structure can be traversed in following ways (types):

1) Depth First search or DFS -
   1) Inorder traversal
   2) Preorder traversal
   3) Postorder traversal
2) Level order traversal or Breadth First search or BFS.
3) Boundary Traversal
4) Diagonal traversal

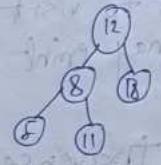
-> Insert the node in increasing order.

(16) A BST BT is termed as BST. If each node 'N' satisfies the following properties:
i) the value at N is greater than every value in the left subtree of N & is less than every value in the right subtree of N.

62, 42, 13, 87, 91, 67, 25, 32



(17) Full binary tree - A full binary tree is a binary tree with either zero or 2 child nodes for each node.

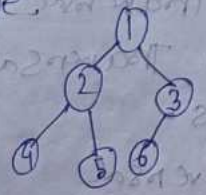


→ full binary tree.

A tree called bi tree if there is

Complete binary tree - max. no. of nodes possible in all levels. i.e it cannot just 1 of

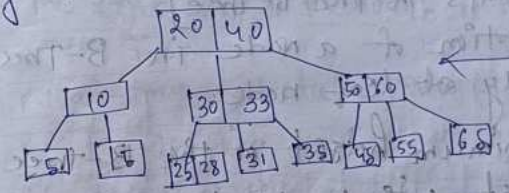
A BT is called CBT if it has max no. of nodes in all level except possibly that last level & the filling of last level will start from as left as possible.



→ CBT

(18) B-Trees - B-tree is a specialized m way tree that can be widly used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node & large key values by keeping the height of the tree relatively small.

B-tree is a special type of self-balancing search tree in which each node can contain more than one key & can have more than 2 children. It is a generalized form of binary search tree. It is also known as a height-balanced m-way tree.



B-Tree

Properties - i) All leaves are at the same level.
ii) B-Tree is defined by the term minimum degree 't'. The value of 't' depends upon disk block size.
iii) Every node except the root must contain at least t-1 keys. The root may contain a minimum of 1 key.

$\lceil \frac{m}{2} \rceil = t$

→ The root node must be in increasing order.

iv) All nodes (including root) may contain at most $(2^*t - 1)$ keys.

v) No. of children of a node is equal to the number of keys in it plus 1.

vi) All keys of a node are sorted in increasing order. The child between two keys k_1 & k_2 contains all keys in the range from k_1 & k_2.

vii) B-Tree grows & shrinks from the root which is unlike Binary search trees. Binary search trees grow downward & also shrink from downward.

viii) The tc of to search, insert & delete is $O(\log n)$ [like other BSTs, n = total no. of elements in the B-Tree]

ix) Insertion of a node in B-Tree happens only at leaf node.

→ The minimum height of the B-tree that can exist with n number of nodes & m is the maximum no. of children of a children of a node can have is:

$$h_{min} = \lceil \log_m (n+1) \rceil - 1$$

→ The maximum height of the B-Tree that can exist with n number of nodes & t is the minimum number of children that a non-root node can have is:

$$t = \lceil \frac{m}{2} \rceil$$

Operations on a B-Tree

1) Traversal - Traversal is also similar to Inorder traversal of B.T. We start from the left-most child, recursively print the leftmost child, then repeat the same process for the remaining children & keys. In the end, recursively print the right most child.

2) Searching an element in a B-Tree

Searching for an element in a B-Tree is the generalized form of searching an element in a Binary search Tree. The following steps are followed -

i) Starting from the root node, compare k with the first key of the node.

If k = the first key of the node, return the node & the index.

ii) If k.leaf = true, return NULL (i.e. not found).

iii) If k < the 1st key of the root node, search the left child of this key recursively.

iv) If there is more than one key in the current node & k > the 1st key, compare k with the next key in the node.

If k < next key, search the left child of this key (ie. k lies in between the 1st & the 2nd keys).

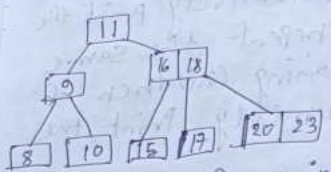
Else, search the right child of the key.

5) Repeat steps 1 to 4 until the leaf is reached.

→ Insert the node in increasing order.

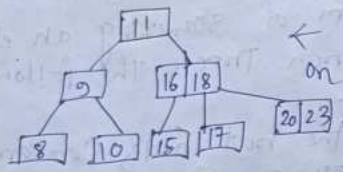
Searching Example

i) Let us search key K=17 in the below of degree 3.



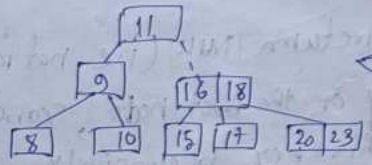
← B Tree

ii) K is not found in the root so, compare it with the root key.



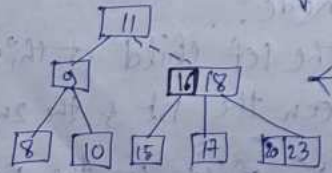
← K isn't found on the root node

iii) Since K > 11, go to the right child of the root node.



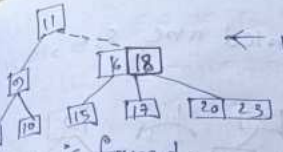
← go to the right subtree

iv) Compare K with 16. Since K > 16, Compare K with the next key 18.



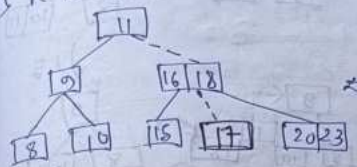
← Compare with the keys from left to right

v) Since K < 18, K lies between 16 & 18. Search in the right child of 18.



← K lies in between 16 & 18

vi) K is found.



← K is found

Insertion on B-tree

Inserting an element on a B-tree consists of 2 events: searching the appropriate node to insert the element & splitting the node if required. Insertion operation always takes place in the bottom-up approach.

Insertion operation

i) If the tree is empty, allocate a root node & insert the key.

ii) Update the allowed number of keys in the node.

iii) Search the appropriate node for insertion.

iv) If the node is full, follow the steps below

v) Insert the elements in increasing order.

vi) Now, there are elements greater than its limit. So, split at the median.

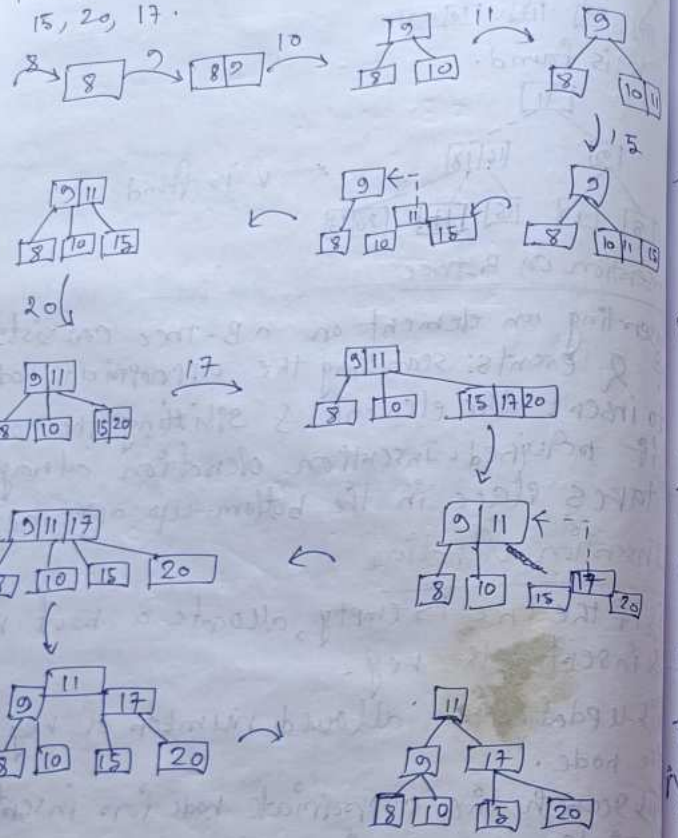
vii) Push the median key upwards & make the left keys as a left child & the right keys as right child.

viii) If the node is not full, follow the steps below.

ix) Insert the node in increasing order.

Insertion Example

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.



Inserting elements into a B-tree

Deletion from a B-Tree – Deleting an element on a B tree consists of 3 main events: searching the node where the key to be deleted exists, deleting the key & balancing the tree if required.

While deleting a tree, a condition called underflow may occur. Underflow occurs when a node contains less than the minimum numbers of keys it should hold.

Deletion operation's terms

i) **Inorder predecessor** – The largest key on the left child of a node is called its inorder predecessor.

ii) **Inorder successor** – The smallest key on the right child of a node is called its inorder successor.

Deletion operation – Before going through the steps below, one must known these facts about a B tree of degree m.

i) A node can have a max of m children. (i.e 3)

ii) A node can contain a maximum of $m-1$ keys (i.e 2)

iii) A node should have a min of $\lceil m/2 \rceil$ children (i.e 2)

iv) A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ keys (i.e 1)

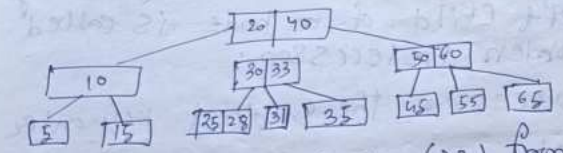
There are 3 main cases for deletion operation in a B tree.

Case I – The key to be deleted lies in the leaf. There are 2 cases for it.

i) The deletion of the key doesn't violate the property of the minimum numbers of keys a node should hold.

In the tree below, deleting 32 does not violate the above properties.

delete 32



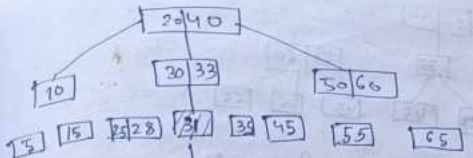
Deleting a leaf key (32) from B-tree

(i) The deletion of the key violates the property of the minimum no. of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

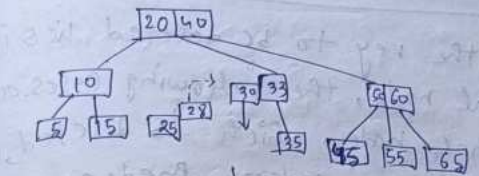
First, visit the immediate left sibling. If the left sibling node has more than a minimum no. of keys then borrow a key from this node.

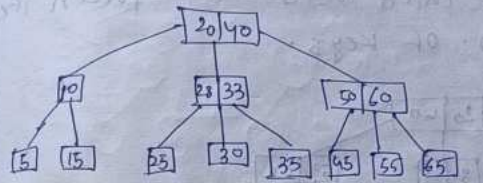
Else check to borrow from the immediate right sibling node.

In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.



delete 31

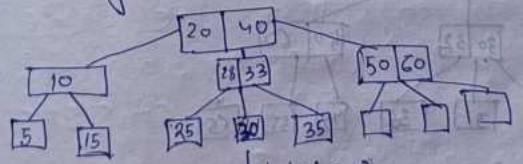




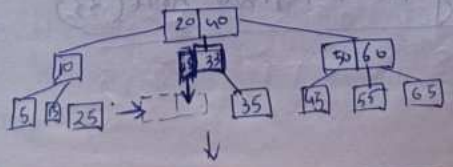
Deleting a leaf key (31)

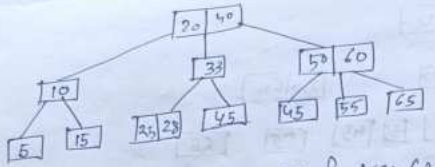
If both the intermediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.

Deleting 30 results in the above case



delete 30

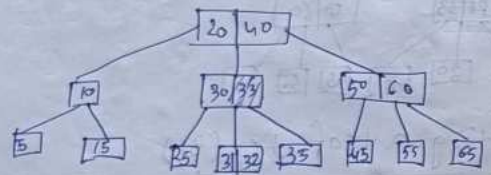




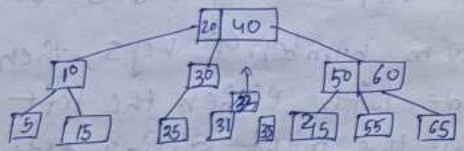
Delete a leaf key (30)

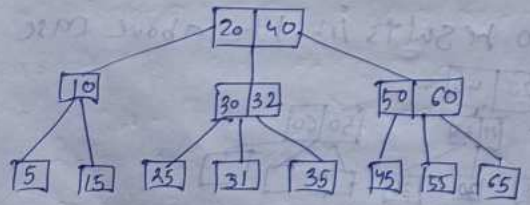
Case II - If the key to be deleted lies in the internal node, the following cases occur.

I) The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum no. of keys.



↓ delete 33

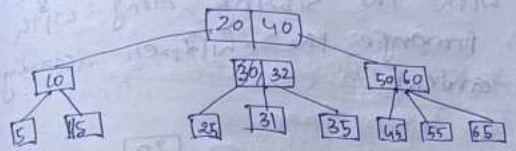




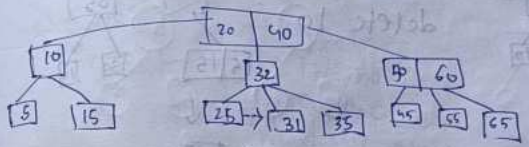
Deleting an internal node (33)

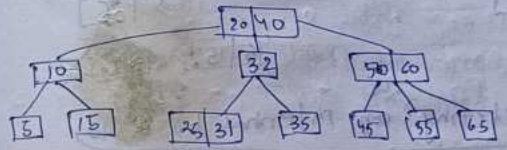
II) If the internal node, which is deleted, is replaced by an inorder successor if the right child has more than the min no. of keys.

III) If either child has exactly a min no. of keys then, merge the left and right children.



↓ delete 30





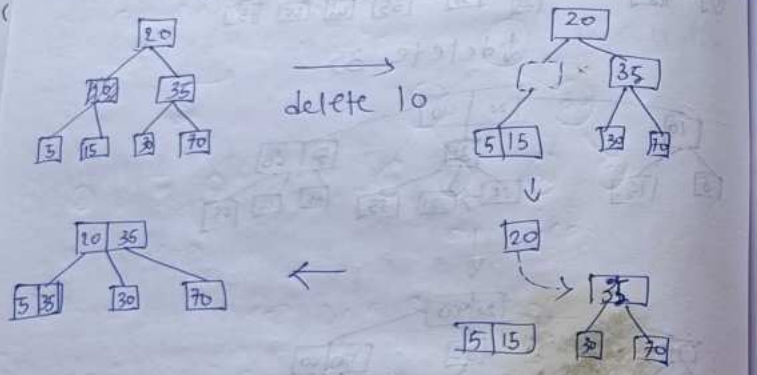
Deleting an internal node (30)

After merging if the parent node has less than the min. no. of keys then, look for the siblings as in case I.

Case III - In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer no. of keys in the node (i.e less than the min required) then look for the inorder predecessor & the inorder successor. If both the childn

-en contain a min. no. of keys then, borrow-ing cannot take place. This leads to case II (3) i.e merging the children.

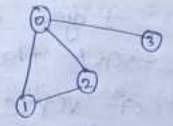
Again look for the sibling to borrow a key. But, if the sibling also has only a min no. of keys then, merge the node with the sibling along with the parent. Arrange the children according (increasing order).



Deleting an internal node (10)

1) Graph data structure – A graph ds is a collection of nodes that have data & are connected to other nodes. Every relationship is an edge from one node to another. More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u, v)



← vertices & edges

In the graph,

$V = \{0, 1, 2, 3\}$

$E = \{(0,1), (0,2), (0,3), (1,2)\}$

$G = \{V, E\}$

2) Difference b/w directed & undirected graph

	Directed graph	Undirected graph
i) Choosing the root	– The root is the node with no incoming edges	Any node can be chosen as the root
ii) DFS check	– No node must be visited more than once	No node must be visited more than once. Also, the parent shouldn't be considered as a child
iii) Connectivity check	– Check that all node are visited.	Check that all nodes are visited.
iv) TC	– O(V+E)	O(V+E) //
v) Defi	– A typed graph that contains ordered pairs of vertices	A type of graph that contains unordered pairs of vertices.

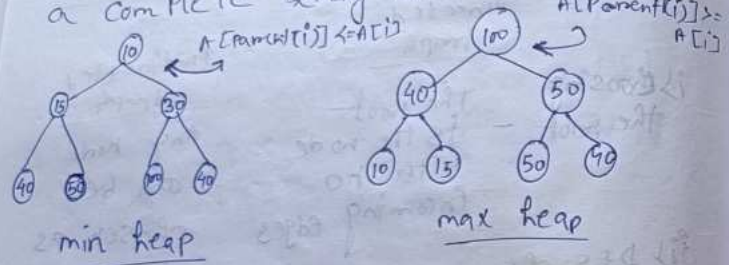
vi) Edges represent the direction of vertexes

vii) An arrow represents the edges

Edges do not represent the direction of vertexes.

* Undirected arrows represent the edges.

2) Heap - A heap is a special Tree-based data structure in which the tree is a complete binary tree.



Operation of Heap Data Structure

i) Heapify: A process of creating a heap from an array.

ii) Insertion: process to insert an element in existing heap time complexity O(log n)

iii) Deletion: Deleting the top element of the heap or the highest priority element and then organizing the heap & returning the element with time complexity O(log n)

iv) Peek: to check or find the first (or can say the top) element of the heap.

Types of Heap Data Structure

Generally, Heaps can be of two types:

i) max-Heap: In a max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

ii) Min-Heap: In a min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

For max Heap - The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete b.t. is always log n; therefore, the time complexity would also be O(log n).

For min-heap - The total no. of comparisons required in the min heap is according to the height of the tree. The height of the complete binary tree is always log n; therefore, the time complexity would also be O(log n).

Properties of Heap

i) CBT – A heap tree is a complete b.t., meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right. This property ensures that the tree is efficiently represented using an array.

ii) Heap Property – This property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.

iii) Parent-child Relationship – The relationship between a parent node at index 'i' & its children is given by the formulas: left child at index 2i+1 & right child at index 2i+2 for 0-based indexing of node numbers.

iv) Efficient Insertion & Removal – Insertion & removal operations in heap trees are efficient. New elements are inserted at the next available position in the bottom-rightmost level, and the heap property is restored by comparing the element with its parent & swapping if necessary. Removal of the root element involves replacing it with the last element & heapifying down.

v) Efficient Access to External – The minimum or maximum element is always at the root of the heap, allowing constant-time access.

Heap operations –

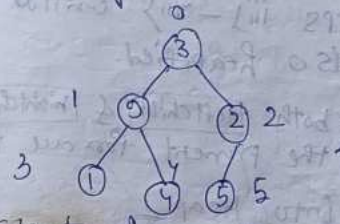
Heapify – It is the process of creating a heap data structure from a binary tree. It is used to create a min-Heap or a max-Heap.

i) Let the i/p array be –

3	9	2	1	4	5
0	1	2	3	4	5

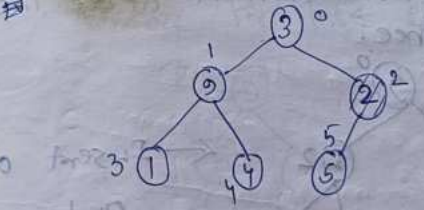
← Initial Array

ii) Create a complete binary tree from the array



← Complete binary tree

iii) Start from the 1st index of non-leaf node whose index is given by $n/2 - 1$.



← Start from the 1st non-leaf node.

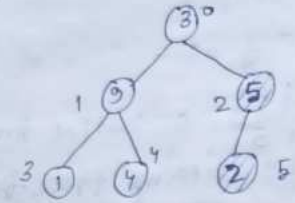
iv) set current element i as largest.

v) The index of left child is given by $2i+1$ & the right child is given by $2i+2$.

If leftchild is greater than current Element (i.e element at ith index)

- set leftChildIndex as largest. If rightChild is greater than element in largest, set rightChildIndex as largest.

iii) Swap largest with currentElement



← swap if necessary

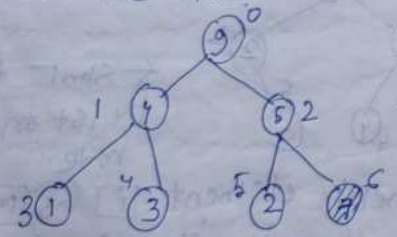
viii) Repeat steps iii) – viii) until the sub-trees are also heapified.

For min-Heap, both leftChild & rightChild must be longer than the parent for all nodes.

Insert Element into Heap –

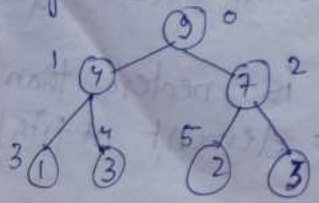
For max-Heap,

i) Insert the new element at the end of the tree.



← Insert at the end

ii) Heapify the tree.



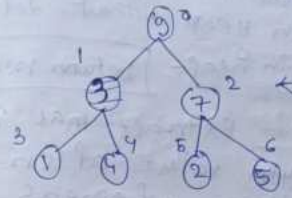
← Heapify the array

For min-Heap, the above algo. is modified so that parentNode is always smaller than newNode.

Delete Element from Heap

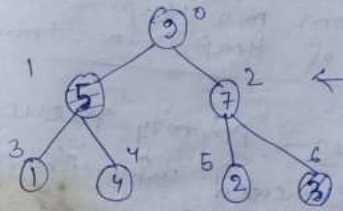
For max-Heap,

i) Selected the element to be deleted.



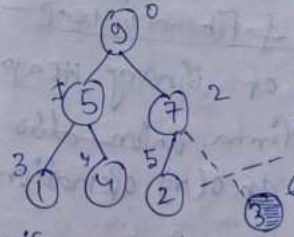
← selected the element to be deleted

ii) Swap it with the last element.



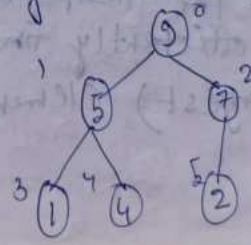
← Swap with the last element

iii) Remove the last element.



← Remove the last element

iv) Heapify the tree.



← Heapify the array

For min-Heap, above algo. is modified so that both [child Nodes] are greater smaller than [current Node].

Peek (Find max/min) - Peek operation returns the maximum element from max Heap or minimum element from min Heap without deleting the node. For both max & min heap - [return rootnode]

Extract-max/min - Extract max returns the node with maximum value after removing it from a max-Heap whereas Extract-min returns the node with minimum after removing it from min Heap.

Implementation of Heap Data Structure

Applications

i) Priority queues - Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() & extractmax(), decreasekey() operation is O(log N) time.

ii) Binomial Heap & Fibonacci Heap - They are the variations of Binary Heap. These variations perform union also in O(log N) time which is an O(N) operation in Binary Heap.

iii) order Statistics - The Heap data structure can be used to efficiently find the Kth smallest (or largest) element in an array.

Advantages of Heaps -

i) Fast access to max/min element (O(1))
ii) Efficient insertion & deletion operations (O(log n))
iii) Can be efficiently implemented as an array.
iv) Flexible size.
v) Suitable for real-time applications.

Disadvantages of Heaps -

i) Not suitable for searching for an element other than max/min (O(n) in worst case)
ii) Extra memory overhead to maintain heap structure.
iii) Slower than others data structures like arrays & linked lists for non-Priority queue operations.

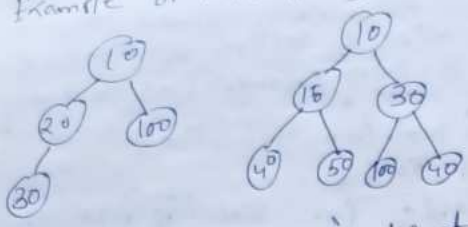
Applications of Heaps (others)

i) Heap sort - It uses Binary Heap to sort an array in O(n log n) time.
ii) Graph algorithms - The priority queues are specially used in Graph algorithms like Dijkstra's shortest path & Prim's minimum spanning Tree.

Binary heap - A Binary Heap is a Complete B.T. which is used to store data efficiently to get the max or min element based on its structure. It is represented basically as an array.

A binary heap is either min Heap or max Heap. In a min Binary heap, the key at the root must be minimum among all keys present in Binary Heap. The same Property must be recursively true for all nodes in Binary Tree. max Binary Heap is similar to min Heap.

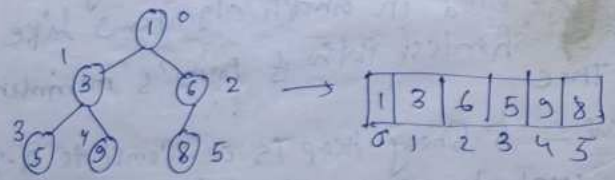
Example of min Heap,



- The root element will be at Arr[0].
- The below table shows indices of other nodes for the ith node, i.e., Arr[i]:

Parent node — Arr[(i-1)/2]
L child " — Arr[(2*i)+1]
R child " → Arr[2*i+2]

The traversal method use to achieve array representation is **Level order traversal**. (For Binary heap)



Heap	Tree
1) It is a kind of tree itself.	1) The tree isn't a kind of heap
2) Usually, heap is of 2 types, max & min heap.	2) Tree can be of various types, Eg (AVL BT, BST)
3) It is ordered	3) BT isn't ordered but BST is ordered.

4) Insert & remove operation takes time of $O(\log(N))$.

3) It can also be referenced to as priority queue.

6) finding max/min value in Heap is $O(1)$ in the respective min/max heap.

2) It can be built in linear tc.

4) Insert & remove operation will take time of $O(N)$

5) A tree can also be referred to as connected undirected graph with no cycle.

6) Finding min/max value in BST is $O(\log(N))$ & Binary tree is $O(N)$

7) BST $O(N + \log(N))$ & Binary tree $O(N)$.

Is the structure of a heap unique?

No, because there can be multiple valid heap structures for a given set of values. However, the property of being a min-heap or max-heap is unique for a given set of values. Both of these structures are valid heap structures, but one is a min heap & the other is a max heap.

Some other types of Heaps—

1) Binomial Heap - The main application of Binary heap is as implement a

priority queue. Binomial Heap is an extention of Binary heap that provides faster union or merge operation with other operations provided by Binary Heap.

A binomial heap is a collection of binomial trees.

A binomial trees of order 0 has 1 node. A binomial tree of order k can be constructed by taking 2 binomial trees of order k-1 & making one the leftmost child of the other.

2) Fibonacci heap - It is a ds for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap & binomial heap.

(3) Leftist Heap, (4) K-ary heap

Greedy algorithm - It is an algo. para-digm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious & immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for greedy.