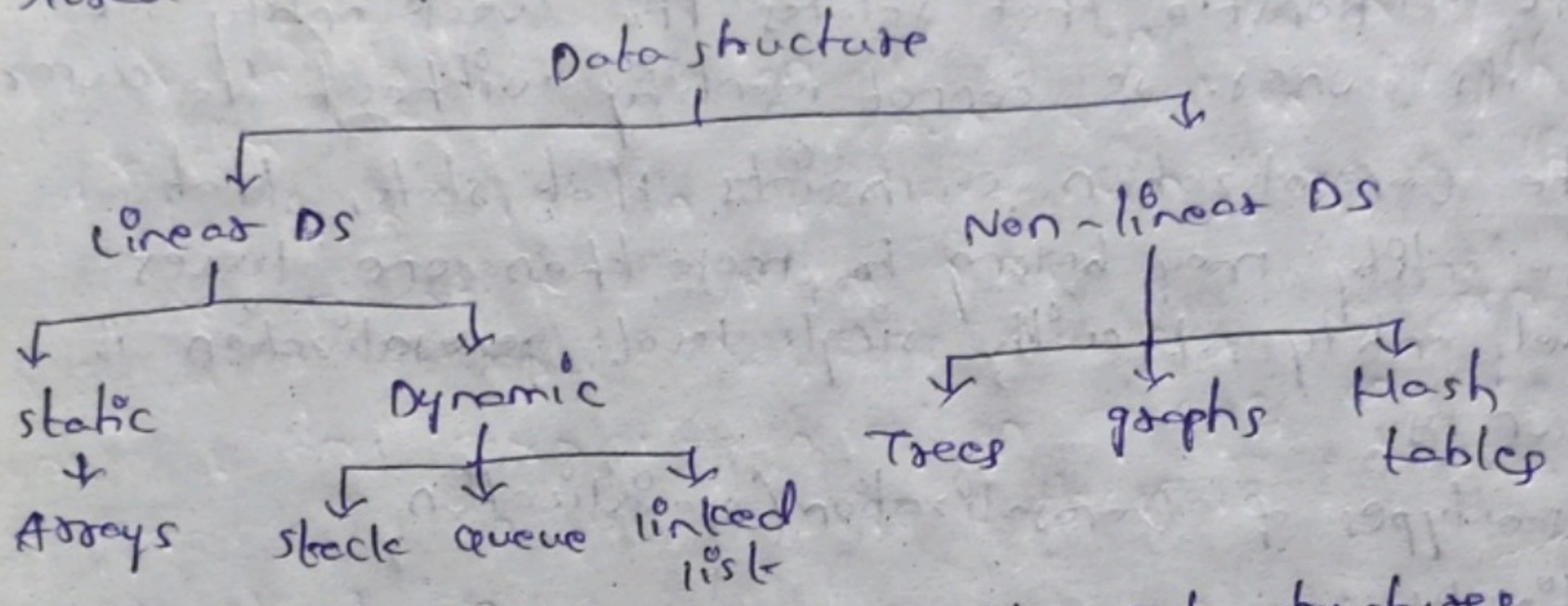
16/2/25

## Data structure (DS)

→ Data Means it contains any kind (type of info.
→ structure means representation
→ Data structure is a storage of technique what we can store data & organize the data so that it can be retrieved efficiently
→ Data structures means the way in which data stored in computes science




→ There are two ways to view the Data structures
1) -) Mathematically
2) logically
3) Abstract models
4) views

ex : LED, screen, smartphone,
call(), text(), video(), photo()

2) Implementation views
ex : class smartphone {
private:
int ramsize;
string processorname;
public:
void call();
void text();
};

### Abstract View

→ It stores the value of int datatype, read the element by position, modify the element by index, perform sorting

Implementation view

```
int arr[] = {11, 12, 13, 14};
cout << arr[1];
arr[2] = 55;
```

16/2/25

## Abstract Datatypes

→ Abstract Datatype means it is a DS defined by its behaviour (operations) without implementation
→ Hiding the implementation
→ C does not have built in support for abstract datatypes like java & python using C classes & objects)
→ But in C lang. we can implement abstract datatypes by using structures, fn's, pointers, arrays, linked lists etc.

### Common type Abstract Datatypes

1) List (Dynamic list using linked list)
→ list is a ordered collection of elements where we can perform inserting, deleting & accessing by using iteration
→ operations like insert (pos, element) search (ele)
delete (pos) Traverse()

2) stack (LIFO)
→ It is a ordered collection of elements where insertion & deletion made from the top end.
→ operations like : push (element), pop(), peek(), isempty(), implementation by using array.

3) Queue (FIFO)
→ It is a ordered collection of elements where elements are inserted at the rare end & ~~checked~~ deleted at front end
→ operations like : enqueue(), dequeue(), isempty()
→ implementation by using stack

4) Hash tables : (key, value pair storage)
→ Hash table stores the data in key value pairs using hash fn's
→ operations like : insert (key, val), search (key)
delete (key)

## Time complexity & space complexity

→ we have diff. algorithms to solve the problems. we need to compare in multiple ways so that we can select efficient algorithm means how much time & resources that algorithm has taken to implement
→ To measure the performance of algorithm we require time complexity & space complexity.

### time complexity

→ It is used to calculate amount of time taken by an algorithm to run as a function based on length of the i/p but not on the execution time of the machine on which algorithm runs
→ By using Asymptotic Notations Analysis we compare space & time complexity based on changes on the performance as i/p size increase & decrease
there are 3 types :
1) Big oh O() → going to considers upperbound
2) Big omega Ω() → lower bound
3) Big theta Θ

1) Big oh O() :
→ Introduced in 1894 by Paul bachman
→ In mathematic Big oh notation is used to describe upperbound of fn
→ Big oh describes worst case of an algorithm in terms of execution time & memory usage
→ Inside the parenthesis there is an expression indicating algorithm run time & it is denoted by "n" which is no. of values in the dataset of an algorithm O(n)

problems.
Hat we
reach
en to implement
we require

aken by an
length
ine of the

we compare
on the

d

to

ithm in

ression
oted by
et of an

2) Big omega $\Omega()$:
→ In mathematic lower bound of function is described by using big omega notation
→ It describes best case scenario ensuring that algorithm takes least time for function call.

3) Big theta $\Theta$: (Tight bound)
→ It has both lower bound & upper bound functioning
→ this algorithm ensures running time always grows at the same rate as $g(n)$

Space complexity
→ It is used to calculate the amount of memory required by an algorithm to execute including i/p storage & Axiliary space during the execution

Components
1) Constant space (fixed part) :- $(O(1))$
→ space required for variables, constants & program instructions
ex: Declarations of datatypes takes fixed amount of spaces.
Dynamic declaration depends on i/p size
stack space for fn calls (recursion)
Heap space by using dynamic allocation (malloc)
Extra space in DS's like arrays, stack, linked list, tress & graphs

| Notation | Description | Example |
|---|---|---|
| $O(1)$ | constant space | using variables swapping two no. |
| $O(\log n)$ | logarithmic space | Recursive binary search |
| $O(n \log n)$ | linearithmic space | merge sort |
| $O(n^2)$ | quadratic space | 2D matrix of size $n \times n$ |
| $O(2^n)$ | Exponential space | Recursive fibonacci |

## Space Complexity for common Ds's

| Data structure | space complexity |
|---|---|
| Array of size n | O(n) |
| linked list (n nodes) | O(n) |
| stack (n elements) | O(n) |
| queue (n elements) | O(n) |
| Hash table (n elements) | O(n) |
| Binary tree (n elements) | O(n) |
| Adjacent matrix (graph) | $O(n^2)$ |
| Adjacent list (graph) | O(n+e) where e is (no. of digits) |

| Notation | Name | Description | example |
|---|---|---|---|
| O(1) | constant time | operation takes the same time regardless of i/p size | Array Access |
| O(logn) | logarithmic time | the operation reduces the problem size in each step | Binary search |
| O(n) | linear time | time ↑ proportionally to i/p | linear search |
| O(nlogn) | linearithmic time | slightly worse than linear, common in efficient storage | merge sort, quick sort |
| $O(n^2)$ | quadratic time | time grows quadratically, seen in nested loops | bubble sort |
| $O(n^3)$ | cubic time | even slower than quadratic, occurs in algo. in three nested loops | Floboracci |
| $O(2^n)$ | exponential time | Growth ↑ with each i/p ↑ | Brute force recursive algo. |

## Time Complexity of Abstract Datatypes

→ the time complexity of abstract types depend on the underlying data structures.

| ADT | Insert | Delete | search | Access |
|---|---|---|---|---|
| Array | O(n) | O(n) | O(n) | O(1) |
| single linked list | O(1) | O(1) | O(n) | O(n) |
| double linked list | O(1) | O(1) | O(n) | O(n) |
| stack | O(1) | O(1) | O(n) | O(1) |
| queue | O(1) | O(1) | O(n) | O(1) |
| Priority queue (Heap) | O(log n) | O(log n) | O(1) | O(n) |
| Hash table | O(1) | O(1) | O(1) | O(n) worst case |
| graph | O(1) | O(1) | O(1) | O(1) |

Sorting algorithm Complexity

| sorting algorithm | Best | Average | worst | space |
|---|---|---|---|---|
| Bubble sort | O(n) | O(n²) | O(n²) | O(1) |
| Insertion sort | O(n) | O(n²) | O(n²) | O(1) |
| Selection sort | O(n²) | O(n²) | O(n²) | O(1) |
| merge sort | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |
| quick sort | O(nlogn) | O(nlogn) | O(n²) | O(logn) |
| Heap sort | O(nlogn) | O(nlogn) | O(nlogn) | O(1) |
| Radix sort | O(nk) | O(nk) | O(nk) | O(n+k) |
| Counting sort | O(n+k) | O(n+k) | O(n+k) | O(k) |

Key takeaways :-

Bubble, insertion, selection are O(n²) slow for large i/p

Merge & quick provide O(nlogn) efficiency

Heapsort has O(nlogn) performance but uses less memory

Searching algorithm Complexity

| Searching algorithm | Best | Average | worst |
|---|---|---|---|
| linear | O(1) | O(n) | O(n) |
| Binary | O(1) | O(logn) | O(logn) |
| Hash table search | O(1) | O(1) | O(n) |

Key takeways :-

→ linear search is slow (O(n) for large i/p)
→ Binary search (for sorted arrays) is O(logn)
→ Hash tables provide O(1) search, but degrade to O(n) in worst case

17/2/25

stack in OS's

→ stack follows the principle LIFO (Last In First out)
→ In a lang. stack can be implemented using
1) Arrays (static implementation)
2) Linked list (Dynamic Implementation)
→ In static implementation they are fixed in size it can overflow if exceeded
→ ~~operations~~ :-
→ Time complexity :-
1) push : O(1)
2) pop : O(1) } → constant time
3) peek : O(1)
4) display : O(n) → linear time
→ operations :-
1) push : Inserting the element at the top
2) pop : Remove & return the element from the top
3) peek : It will not remove the element. It will retrieve the top element & it will display
4) isempty : It checks whether stack is empty/not
5) isfull : It checks whether stack is full/not
→ It is used in only array based implementation
6) display : It displays all the elements from the stack.

→ Maxim
→ the n
if top
retur
else
retur
what is
6 3
4
2
3
6
e) 2 3 1
1
3
2
e) 100 2
200
100
→ by us
→ by us
→ by us
accessin
queue
→ queue
→ It has
1) rate
2) Front
→ from t
by usi
→ from
using
→ by us
buffer

worst

o(n)

o(log n)

o(n)

o)

ogn)

degrade

irst out)

size it

m the top

it will

empty/not

ull /not

ementation

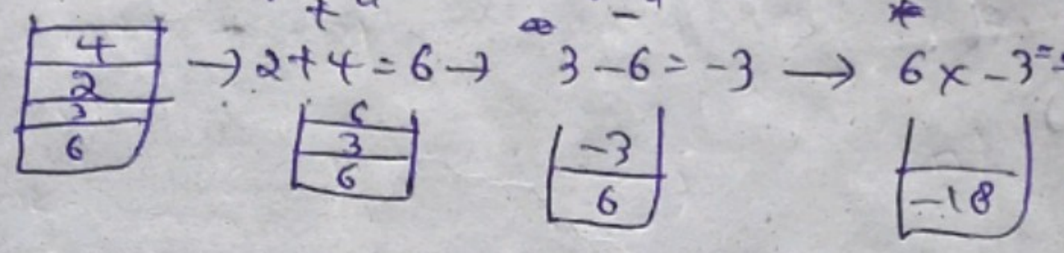
nts

→ Maximum capacity of the stack is MAX-SIZE element
→ the max. value of the top can be MAX-SIZE - 1

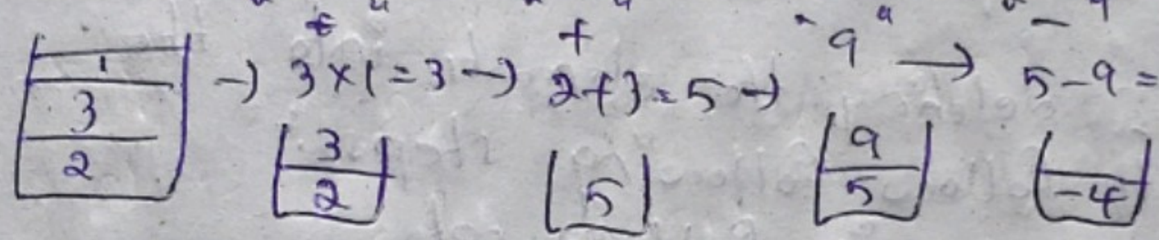
if top >= MAX-SIZE - 1
return true
else
return False

what is the value of post, prefix expression

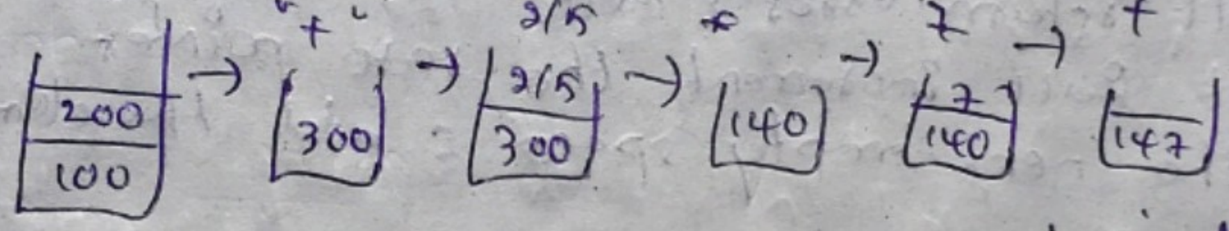
6 3 2 4 + - *




e) 2 3 1 * + 9 -




a) 100 200 + 2/5 * 7 +




→ by using stack we can develop balanced paranthesis.
→ by using stack we can implement recursive algorithm
→ by using stack we can develop the process of accessing the data in serial access memory.

Queue

→ queue follows the priciple FIFO (First In First out)
→ It has two edge :-
1) rare end
2) front end
→ from the rare end we will insert the element by using "enqueue" fn
→ from the front end we delete the element by using "dequeue" fn
→ by using queue we can implement task scheduling.
buffering in data transmission

→ operations :
→ In c lang. we can implement queue by using arrays & linked list

1) enqueue
2) dequeue
3) peek() or front() → to access the element available at front node of the queue without deleting it
4) rare() → to return element at rare node without removing it.
5) isEmpty()
6) isFull()
7) display()
8) size()

→ Before inserting & deleting the elements into the stack we have to follow following steps :-

enqueue
1) check whether queue is full or not
2) If it is full return overflow error & exit
3) If it is not full increment the rare pointer to point to the next empty space & add the element

dequeue
1) check whether queue is empty or not
2) if it is empty returns underflow error & exit
3) if it is not empty access the data where front is pointing. Increment the front pointer to the next available data.

## Types

→ There are 5 types of queues :
1) simple queue — [input restricted queue, output restricted queue]
2) circular queue (Ring Queue)
3) Double ended queue
4) priority queue — [ascending priority queue, descending priority queue]

wing arrays

t available
out deleting it
node

into the
ps :-

e pointer
dd the element

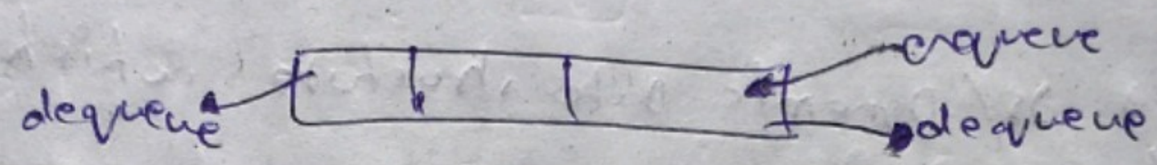
ot & exit
outside
nt pointer

eue

## Circular queue

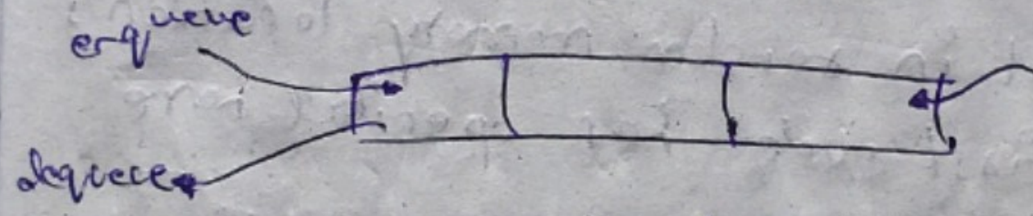
→ It follows FIFO principle
→ Last position is connected back to the first position to form a circle
→ It is also called as "Ring Buffer"
→ Applications
1) Memory mgm
2) Traffic system
3) CPU scheduling (Threads)
4) Multi level Cache mgm
5) Real time task scheduling

## Input Restricted queue




→ Input can be taken from one side only
→ Deletion can be done from both ends (front & rear)
→ It will not follow FIFO principle

## output Restricted queue




→ Input can be taken from both the side (rear & front)
→ Deletion can be done from only one side (front)

## Double Ended Queue

→ From both the ends we can perform insertion & deletion. ex:- sliding windows algorithm
undo operations in the text editors
stock price application, median finding in dat streams

## priority queue

→ In this queue each element is associated with the priority & is accessed according to its priority
→ Elements with highest priority will be dequeue first
→ If both the elements having same priority then we follow FIFO principle
→ It can be implemented by using Arrays & linked list
→ the best performace will be O(log n) for insert & delete

Applications

1) Task scheduling in OS
2) Dijkstra's (shortest path algorithm)

→ the DS which is used in breadth (bfs) on a graph is queue
→ if the element "def" are placed in a queue & deleted one at a time in which order it will be deleted. def (FIFO)

18/2/25

Linked list

→ It is a linear DS/ Dynamic Data structure which consist of nodes
→ Each node is divided into 2 parts:
1) It holds the data
2) pointer always points to the next node
→ If the pointer is null then it is the last node in the list
→ elements will be stored in random memory locations
→ the address of the first node has special name called "Head"
→ linked list can be break & we can insert the elements & we can rejoin
→ Node Representation by using structure

```
struct node
{
 int data;
 struct node *next;
};
```

→ Initializing the nodes:

```
struct node *head;
struct node *one=null;
struct node *two=null;
```

→ Reallocating memory

```
one=malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
```

→ Assigning the value :-

one→data = 1;
two→data = 2;

→ connecting the nodes :-

one→next = two;
two→next = three;
three→next = NULL;

→ head = one;

Applications

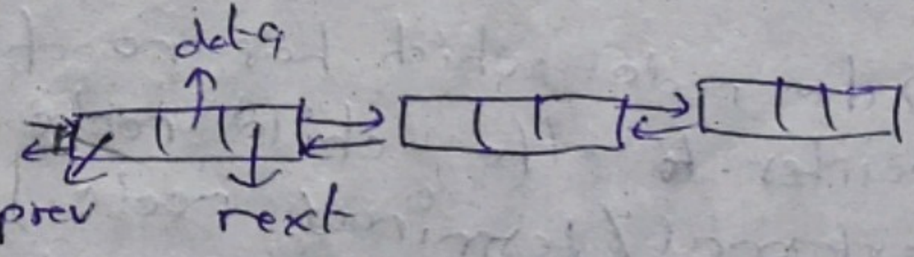
1) stack
2) Queues
3) Hash tables
4) graphs
5) undo functionalities

Double linked list

→ In this list each node is divided into 3 parts.
→ It has 2 pointers i.e: previous & next
1) data → it stores actual value
2) pointer to the next node
3) pointer to the previous node

→ Node Representation by using structure

```
struct node
{
int data;
struct node *next;
struct node *previous;
};
```

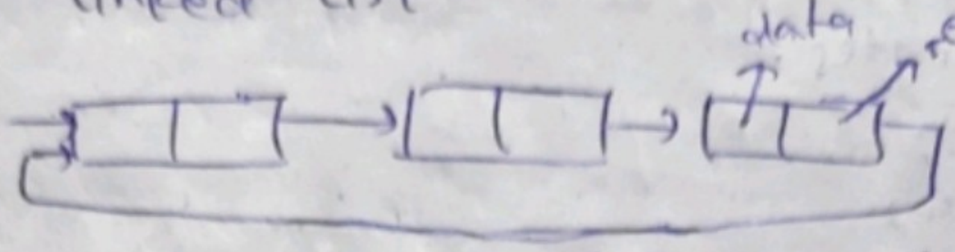



operations

1) insert At Beginning()
2) insert At End()
3) delete Node()
4) printList()

## Circular linked list

→ It can be implemented either by using single / Double linked list



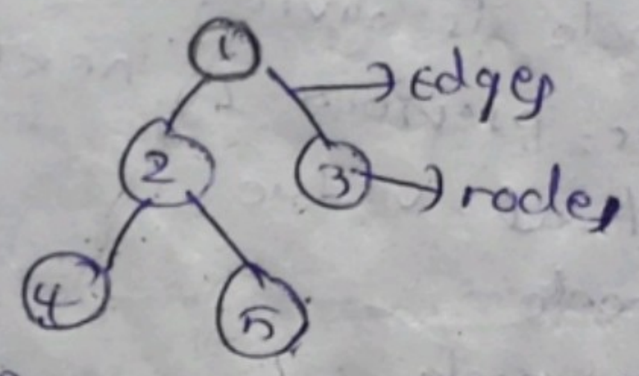

→ Last element pointed to the 1st element to form a circle

→ In Double linked list previous pointer of the first item points to the last item

## Trees

→ It is a Non-linear DS

→ Data is stored in the format of hierarchy which consist of nodes connected with edges




**node** : It is an entity which contains a key or value, & pointer which points to the child node

→ the node which has no child / pointer doesn't have pointer to the child node is called leaf / external / terminal node

→ the node which has atleast one child is called internal node / non-terminal node.

ex : A, B, C, E, G

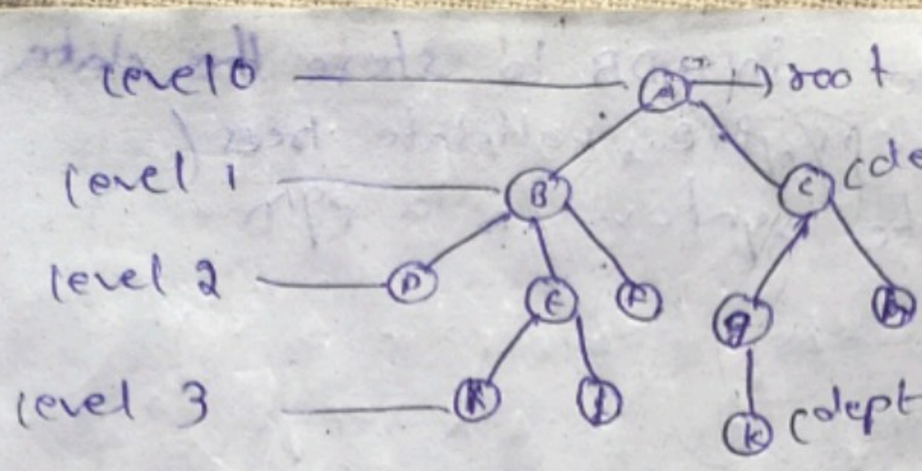
**Edges** : It is used to connect the two nodes

→ If we have n nodes then we have n-1 edges

**Height of the node** : The longest path from the root node to leaf node.

→ no. of edges from root node to deepest leaf node

Ex : A → B → E → J




Forest :- the collection of disjoint trees is called forest

Parent :- the node which is predecessor of any node is called parent node.

Ex : A, B, C, E, G

siblings :- the child nodes of same parent are called siblings Ex : B, C (A), D, E, F (B), G, H (C), I, J (E)

Degree : Total no. of child nodes for a node

Ex : degree of A = 2
" " B = 3
" " F = 0
" " E = 2

level :- the root node is said to be at level 0 & the children of root node are at level 1

~~depth~~ depth :- the total no. of edges from root node to any node

path :- sequence of node & edges from one node to another node

Binary Tree

→ In any tree we can have n no. of nodes. In a Binary tree in which every node have maximum 2 nodes are is left node & one is right node

Ex : A, A, A

→ Applications

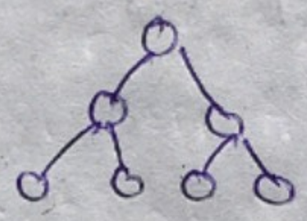
1) Binary search trees (simple & fast)
the modern version of tree is known as tries
it is used in modern routers to store routing info.

2) It can be implemented in DB to store the data
3) It is used in compilers, i.e., validate trees/ syntax tree to validate syntax in a pgm

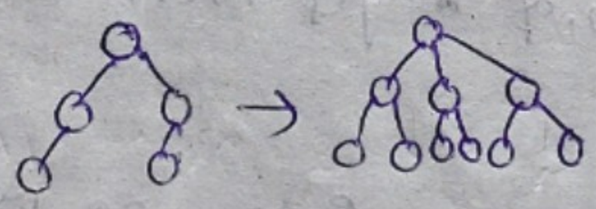
→ Types :-

1) Strictly Binary Tree
→ In this tree every node should have exactly two childrens/none
→ It is also called as Full Binary tree, Proper Binary tree, 2-Tree

2) complete Binary Tree
→ In this tree every node must have exactly two childrens
→ It is also called as perfect Binary tree, complete Binary tree



3) Extended Binary Tree :
→ A Binary tree can be converted into full Binary tree by adding dummy nodes to the existing nodes wherever we required



→ Binary tree can be implemented by using Arrays & linked list

Array based representation
→ In this we follow indexing rules
1) 0 based
2) 1 based
→ Assuming we are using zero based index then root node is at index zero
→ for a node at index i, the index of the left node is 2i+1, the index of the right node is 2i+2

→ For one based indexing:
starting index = 1
left node is 2i
right node is 2i+1

→ ~~Binary tree~~
Traversal

→ there are 3 ways to traverse nodes of a binary tree.
1) Inorder traverse
2) Preorder traverse
3) postorder traverse

Inorder traverse (left - root - right)

→ In this order left child node is visited first then root node is visited & later right child node is visited.

19/2/25

a) In a complete Binary tree with n leaves the total no. of nodes depends on no. of internal nodes = 2n-1

e) which DS is having multiple traverse Non-linear DS

c) polynomial manipulations can be done by using which DS linked list

Arrays &

searching Algorithms

Search → it is a process of finding the element in a list of values

Types
1) Linear search algorithm
2) Binary search algorithm

Linear search algorithm (sequential search algorithm)

→ It is a simple searching algorithm that checks each element sequentially until it finds the target element
→ If loop completes the target is not found it is going to return "-1" indicating element is not found.

→ the Best case is O(1)
→ the worst case & average case is O(n)

Binary search Algorithm

→ In this search we find the elements in a sorted list by repeatedly dividing the search range into half

→ set the two ~~Ka~~ pointers:
1) Low → starting of the list
2) High → ending of the list

→) find the mid element:

$$mid = \frac{high + low}{2}$$

→ if the mid element matches it is going to return the index of the element

→ if the target is less than mid value search in left half i.e., high = mid - 1

→ if the target is greater than mid value search in right half. i.e., low = mid + 1

Time complexity

→ the best case is O(1) → if the target is the mid value

→ the worst case & average case is O(log n) → dividing the search ~~space~~ case by half into each step

→ In the linked list each ~~a~~ node represent a term of a polynomial (coefficient & exponent)

→ A complete Binary tree is full upto second last level because all the leaves are filled & with left to right nodes

→ Total nodes = internal node + leaf node

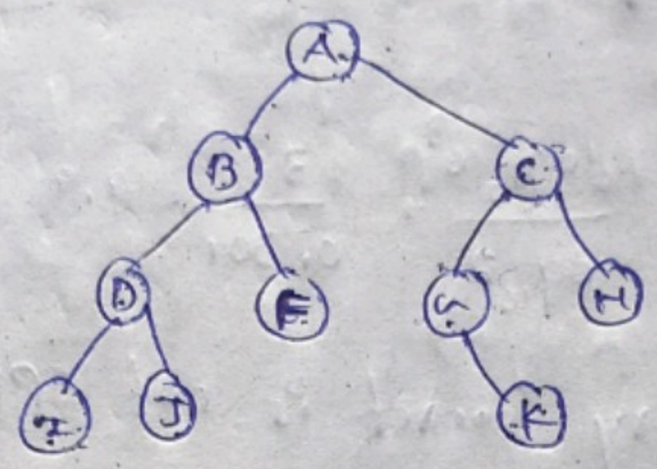
a) let A, be an array then you can access Mid element by using pointers.

a) if the height of a tree with single node is considered as 1 then max no. of nodes in a binary tree of height 5 = $2^{h} - 1 = 2^{5} - 1$
$= 31$

→ By using Double ended queue we can insert & delete the elements from both

→ I/p restricted DQ & o/p restricted DQ

→ linked lists are not suitable in the representation of binary sort




In order Traverse (left-root-right)

→ I, D, J, B, F, A, G, K, C, H

Pre-order Traverse (root-left-right)

→ A, B, D, I, J, F, C, G, K, H

Post-order Traverse (left-right-root)

→ I, J, D, F, B, K, G, H, C, A

In order = D, B, F, E, G, H, A, C

Pre-order = A, B, D, E, F, G, H, C

@) which search algorithm is best more efficient when we have less data sets linear

@) the no. of nodes in a full Binary tree with n leafes $2n-1$

match the following Traverse

1) Pre — a) left-root-right
2) In — b) left-right-root
3) post — c) root-left-right

@) In a queue we remove the item that is least recently added

c) Inorder : 4 2 5 1 3 6
preorder = 1 2 4 5 3 6

c) a+b*(c^b-c)^(p+q*h)-1

c) If an array contain 1000 elements new array comparisions will binary then search takes in worst case!

$\log_2 1000 \rightarrow \frac{\log_{10} 1000}{\log_{10} 2} \rightarrow \frac{\log_{10} 10^3}{\log_{10} 2} \rightarrow \frac{3}{0.301} = 9.9$

a) In which DS the binary search is not applicable

Hash tables

c) advantages of Binary search over linear search
1) low memory
2) fast performance
3) It is suitable for large data sets

c) following is not applicable for binary search
a) It requires sorted data
b) It is fasted than linear search
c) It can be both used for sorted & unsorted data
d) It has logarithmic time complexity

a) space complexity for recursive binary search
O(log n)

20/2/25

sorting algorithms

sorting → It is a process of arranging the data in specific order (either ascending / descending)
→ they are categorized based on complexity, stability, approach (comparision based & Non-comparision based)

## External sorting algorithm

→ An algorithm that uses tape/disk during the sorting. when we have huge data sets

## Internal sorting algorithm

→ An algorithm that uses main memory for sorting when we have less data sets

## Comparision based sort algorithms

→ simple sorting algorithms

1) Bubble sort :
→ Repeatedly swap adjacent elements if they are in wrong order
→ Best case : $O(n)$
→ average case & worst case : $O(n^2)$
→ stable : Yes

2) selection sort :
→ find the smallest element & places in correct position
→ Best case, average case, worst case : $O(n^2)$
→ stable : No, swap may change when there is a duplicate values.

3) Insertion sort
→ pick the element one by one & inserts them in a sorted position
→ best case : $O(n)$
→ average & worst case : $O(n^2)$
→ stable : Yes

## Efficient sorting algorithms

1) Merge sort (Divide & conquer)
→ Recursively split array elements into halfs & merge them in sorting order
→ best & average & worst case : $O(n \log n)$
→ stable : Yes
→ space complexity : $O(n)$

2) Quick sort (Divide & conquer)
→ select the pivot element & partition the elements around it
→ best & average case : $O(n \log n)$
→ worst case : $O(n^2)$
→ stable : No

3) Heap sort (Heap DS)
→ converts the array into heap & extract max/min value
→ Best, average, worst : $O(n \log n)$
→ stable : No

Non-comparision based sorting algorithms

1) Counting sort :-
→ It uses frequency counting to sort the elements
→ Best case = $O(n+k)$. $k$ = range of elements
→ stable : yes
→ average and worst case = $O(n+k)$

2) Radix sort
→ sort the numbers/elements by digit by digit using counting sort as subroutine
→ best, average, worst case = $O(nk)$

3) Bucket sort
→ Distribute the elements into the bucket & sort them individually
→ best case : $O(n)$
→ worst, average case : $O(n^2)$
→ stable : yes

te elements

act max/min

elements

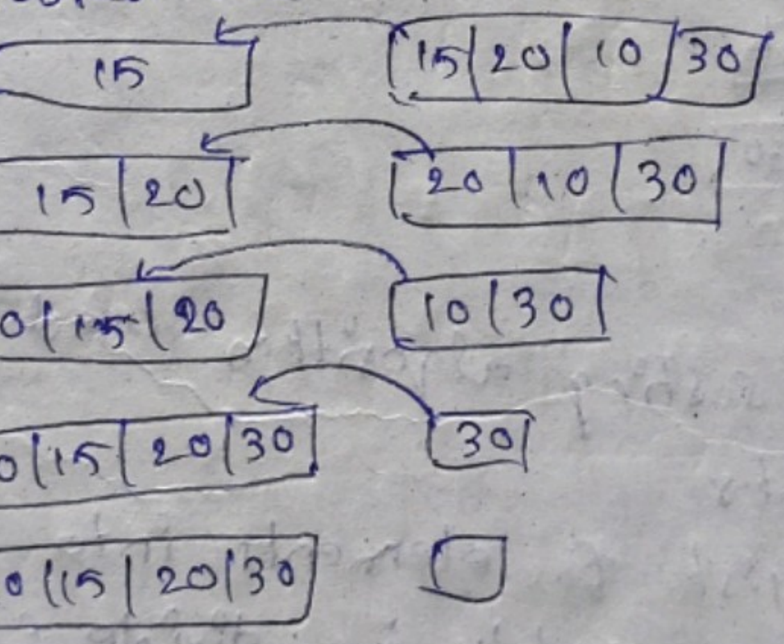
digit using

sort them

1) Bubble sort
→ It is the simplest sorting algorithms that works repeatedy swapping adjacent elements
→ It is not suitable for large data sets
→ All sorting algorithms are faster than Bubble sort
→ no. of swaps in the best case scenario for bubble sort "zero"
→ In Bubble sort how many passes are required in the worst case for sorting an array of size n = "n-1"
→ If the list is ~~orde~~ already sorted how may swaps in bubble sort "zero"

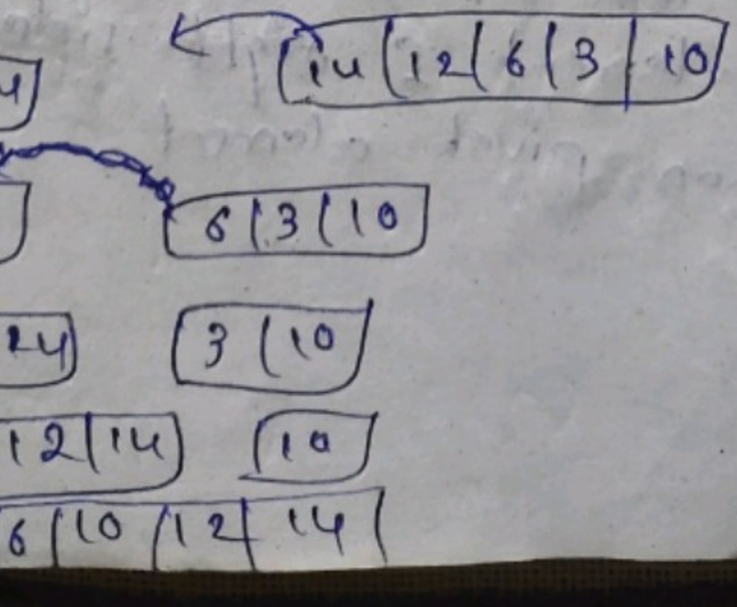
2) Insertion sort
→ It arranges list of elements in particular order
→ Every iteration moves an element from unsorted position to sorted position until ~~the~~ all the elements are sorted.

Ex:




a) what will be the no. of passes element wing insertion sort 14, 12, 16, 6, 3, 10

no. of elements = 6
no. of passes = n-1
= 6-1
= 5



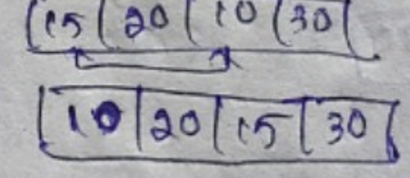
@ How many passes does insertion sorting consist of an array of n elements = "n-1"

e) what is the real time applications is based on insertion sort

1) arranging pack of playing cards (Insertion sort)
2) Database scenario & Distribution scenario (merge sort)
3) Arranging books on a library shelf (stack)
a) Real time system ex: quick sort

@ To c prog. which basic loops are used to perform insertion sort → for loop & while loop (outer for loop & inline while loop)

3) Selection sort
→ It repeatedly finds the minimum element & places in the correct position
→ pick the smallest element & compare with the remaining elements. If the element is smaller than selected element then swap




a) Quick sort:
→ It is the fastest sorting algorithm used for sorting the elements
→ It separates the list of elements into 2 parts. & sorts each part separately i.e., divide & conquer
→ It works on selecting the pivot element all the elements to the left of the pivot element are smaller than pivot element & right side of the pivot are greater than pivot element

ting consist

's based on

section sort)

tio (merge sort)

F (stack)

to perform

er for loop &

ert & places

with the

smaller

sed for

to 2 parts.

e & conquer

rent all

element are

de of the

Advantages

→ suitable for large data sets

→ In-place sorting → it doesn't require any extra space

→ Efficient for general purpose sorting, Ex: python

pgming lang. libraries, sorted fn

Disadvantages

→ Not stable

→ worst case = $O(n^2)$ → if pivot element selection is poor

22/2/25

Quick sort

7 6 10 5 9 2 1 15 7

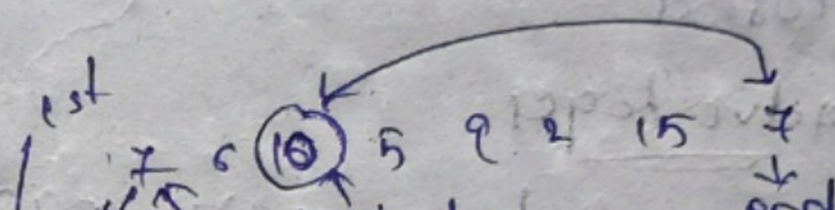
→ 7, 6 7 5 9 2 1 15 10

→ 7 6 7 5 1 2 9 15 10

→ 2 6 7 5 1 (7) 9 15 10

fixed

1st




7 ≤ 7 = T

6 ≤ 7 = T

10 ≤ 7 = F

7 > 7 = F

2nd 7 6 7 5 9 2 15 7

→ Quick sort algorithm is highly efficient sorting algorithm that follows divide & conquer approach

→ It is an in-place sorting algorithm

→ If we select first element as pivot if array is sorted it leads to worst case performance

→ It uses internal sorting algorithm

→ Lomuto partition scheme } → these two techniques
→ Hoare partition scheme } will be used in quick sort

→ if quick sort is implemented purely (choosing bad pivot element). it behaves like selection sort

Merge sort (Divide & conquer)

→ split the array into smaller sub arrays
sort them & merge them back together
→ It is known for the efficiency & stability

pointers :

→ pointer is a variable which holds the address of another variable of same datatype
→ It points to location where the value is stored

Advantages

1) Efficient memory mgm
2) pass by reference in fn's
3) Dynamic memory allocation (linked list, trees etc)

Syntax for declaring a pointer variable :

datatype *pointer_variable;

ex: int *ptr; const int *ptr;
float *ptr; char *ptr;

Deference (Indirection operator (*)) :

→ It is used to access the value stored in the address

Assigning address to the pointer

```
int a=10;
int *ptr;
ptr = &a;
pf ("Address = %u", ptr);
pf ("Value = %d", *ptr);
```

format

%p → hexadecimal
%u → unsigned (decimal)

, arrays
etter
stability

te addsers
ype
alue is

st, taces etc)
e:

sed in

t
decimal
ned (decimal)

## pointers to the Arithmetic

→ we can apply arithmetic operators to the pointers
1) ++ → moves the pointer to the next location
2) -- → moves the pointer to the previous location
3) + → moves the pointer ahead by "n" element
4) - → moves the pointer back by "n" element

ex: int arr = [10, 20, 30, 40]
int *ptr = arr;
pf ("%p", ptr); → address
pf ("%d", *ptr); → 10
ptr++;
pf ("%d", *ptr); → 20

## pointer to arrays

→ An arrayname is itself a pointer to the first element

int arr[] = {10, 20, 30, 40};
int *ptr = arr;
pf ("%d", *(ptr+1)); → 20

int *ptr = &arr, address of array address

## ** (pointer to pointer)

→ A pointer that stores the address of another pointer

int i=10;
int *ptr = &i;
int **ptr2 = &ptr;
pf ("%d", **ptr2); → 10

→ upto 32 (+ve) we can write

## function to pointers

→ we can call the fn by using the pointers

```
void sayhello(){
    pf("Hello");
}
void main(){
    void (*fnptr)() = sayhello;
    fnptr(); → Hlo
}
```

23/2/25

Types of pointers

1) Integer pointer : these are the pointers which are pointing to the integer values
ex : int *ptr;

2) Array Pointer : array name is the pointer to its first element
→ It is also known as pointer to Arrays
ex : int *ptr = arr_name;

3) structure pointer : the pointer pointing to the structure type. syntax
ex : ~~int structure_name~~; struct structure_name *ptr;

4) function pointer : A pointer which points to the fn
syntax
ex : returntype pointer_variable
void (*ptr)()
int (*ptr)()

5) Double pointer : A pointer which stores address of another pointer
ex : syntax : datatype ** pointer_variable;

6) Null pointer : these are the pointers that do not point to any memory location.
→ they can be created by assigning null value to the pointer
ex syntax : datatype *ptr_var = NULL;

7) void pointer : these are the pointers which are pointing to the void type
→ these are also called as Generic pointers they can point to any datatype
syntax : void *ptr_var;

8) wild pointer : these are the pointers that do not have been initialized.
ex : int *ptr;

a) constant pointer :- these are the pointers the memory address stored inside the pointer is constant & can't be modified once it is defined.
→ Always it is pointing to the same address
syntax : datatype * const var_name;

(a) pointer to constant :- these are the pointers that is pointing to constant value that can't be modified.
→ we can change the address stored in the pointer to constant
syntax : const datatype *ptr_var;

→ the size of pointer will be depend on operating system
32-bit os → 4 bytes 64-bit os → 8 bytes

Ex :-
```
struct str
{};
void fn(int a, int b){};
void main(){
  int a=10;
  char c='a';
  struct str x;
 int *ptr_int=&a;
 char *ptr_char=&c;
 struct str *ptr_str=&x;
 void (*ptr_fn)(int, int)=&fn;
 void *ptr_v=NULL;
 pf("size of int = %d", sizeof(ptr_int));
 pf("size of char = %d", sizeof(ptr_char));
 pf("size of struct = %d", sizeof(ptr_str));
}
```
→ 4 bytes

*→ multiplication & Division can't be performed on pointers

```
void main()
{
 int number = 50;
 int *ptr = &number;
 pf("before = %u", ptr); → 0522296
 ptr = ptr + 1;                  + 4
 pf("after = %u", ptr) → 0522300
```

→ int so we have to add 4 bytes.

Arrays to the pointers

ex: int arr[4];
 int i;
 for(int i=0; i<3; i++)
 {
  pf("&arr[%d] = %p\n", i, &arr[i]);
 }
 pf("address of first element = %p", arr);
}

ex: int arr = {1,2,3,4,5};
 int *ptr;
 ptr = &arr[2];
 pf("3rd element value = %d", *ptr); → 3
 pf("4th element value = %d", *(ptr+1)); → 4

→ In c lang. it is possible to pass addresses as argument to the function
→ we can pass pointer to the fn as argument which allow the fn to modify the value of original variable value. It is known as pass by reference

ex: void Func(int a)
 {
  pf("value of a = %d\n", a); → 22
 }
void main()
{
 void (*fn_ptr)(int) = &Func;
 (*fn_ptr)(22);
}

→ int so we have
to add 4 bytes.

arr[i]);

, arr);

*ptr); →3

i)]; →4

ters as argument

gument which
of original
y reference

12

## Dynamic Memory Allocation

→ the process of allocating the memory at the runtime is called DMA

→ It is done by using 4 fn's

1) malloc()
2) calloc()
3) realloc()
4) free()

→ available under "<stdlib.h>"
→ return type is "void"

### malloc (memory allocation):

→ we can apply DMA for any datatype
→ global variables, static variables, pgming instructions are stored in permanent storage area
→ local variables are stored in stack area

malloc()
→ It allocates single block of memory of given size & returns pointer of void datatype

ptr = (casttype) malloc (size of type);
ptr = (int *) malloc (100 * sizeof (int)); → 400 bytes of space

calloc()
→ Contiguous memory allocation
→ It is used to specify no. of blocks of memory
→ It takes 2 arguments

ptr = (casttype) calloc (n, size of type);
no. of elements

ptr = (int *) calloc (10, size of (int));

realloc()
→ Reallocation
→ This fn is used to change the previous memory allocated by malloc

ptr = realloc (ptr, newsize);

free()
→ this fn is used to deallocate the memory used by malloc, & realloc

free (ptr);