Data Structure & Algorithms

Data Structure

1) Linear Data - Data structure in which elements are traversed sequentially one by one.

   1) Array 2) Linked List 3) Stack
   4) Queue

2) Non Linear Data - Data structure in which elements linked to many other data elements (more than one)

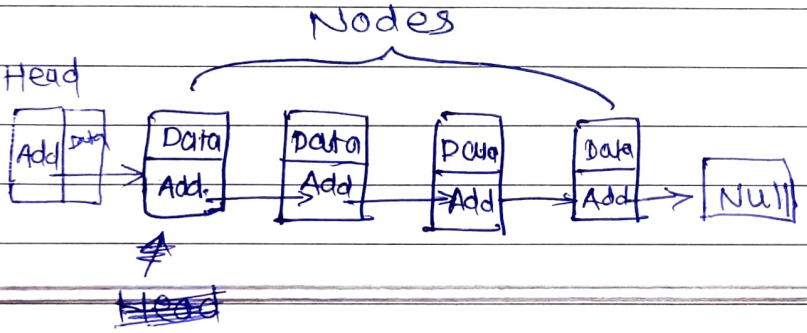
   1) Trees 2) Graphs. 3) Heap

Linked list

Linkedlist – i) linear data structure where unlike array element not stored contigous locations. Every element is with data & address part.
ii) Elements are linked using pointers & address

Advantage – we can dynamically ~~access~~ insert, delete the elements.

Disadvantage – Nodes are not accessible directly have to travel from head



Data only accessible through head.

Syntax ⇒

```
LinkedList<String> object = new LinkedList<String>();
        object.add("A");
        object.add("B");
        object.addLast("C");
        object.addFirst("D");
        object.add(2,"E");
        object.add("F");
        object.add("G");
//1    Sysout("Linked List:" + object);

        Object.remove("B");
        object.remove(3)
        object.removfirst();
        object.removeLast();
//2    Sysout("LinkedList after deleting": + object);
        boolean x = object.contains("E");
```

```
// 3    Sysout (x);
     int size = object.size();
// 4    Sysout (size);

     object element = object.get(2);
// 5    syout (object element);
     object.set (2, "Y");
// 6    sysout ("object after setting" + object);
```

Output
1) [D, A, E, B, C, F, G]
2) Linked list after deleting [A, E, F]
3) True
4) 3
5) F
6) [A, E, Y,]

Some other methods

clear() - clear all elements
clone() - Shallow copy
contains(object o) - boolean check
getFirst() - First element
getLast() - last element
indexOf(object o) - get index of element
remove() - remove head, return removed element and (shift other element)
remove(int index) - remove element at index
removeFirst() - remove first element
removeLast() - remove last element.
removeFirstOccurance (object o)
set (int index, E Element)
toArray() - convert to array.

```
String[] arr = obj.toArray(new String[obj.size()]);
```

Types

1) simple Linked list - i) Item nevagation is forward o
   i) Have Data and link to next element.

2) Doubly Linked list - i) Can be traversed forward and ba
   ii) Have Data & link to previous & forword elements.
   iii) Previous of first & forward of last element linked to n
   iv) deletion is easy.

3) Circular Linked List - Last items have link to first
   i) Circular simple linked list
   ii) Circular Doubly linked list

No beggining no end.

Why to use linked list instead of array?

1) Size of array is fixed it cannot be changed whe we need to add more elements we have to predict the size before creating array this limi is handled by linked list by allocating memory dynamically that means memory allocated at run time by compiler and do not need to ment size while declaration of Linked list.

2) Inserting, deleting elements from array is perf wise expensive as we need shift several elem But in LinkedList only references are shifted and no need to shift elements.

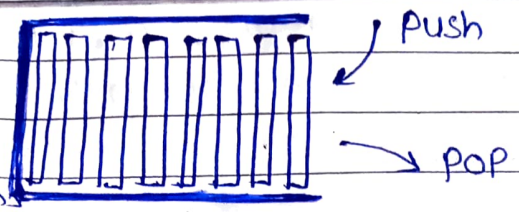
Merge sort ($\Theta(n \log n)$) and insertion sort $O(n^2)$ can be used for linkedlist while merge sort is prefered.

Quick sort is very hard to apply Heap sort is impossi quick sort not suitable for LL because it uses indexes as in array However it is useful for array

Stack data structure

Camlin | Page
Date / /

Stack - Linear data structure which follows particular order of operations performed (LIFO / FILO)



Stack

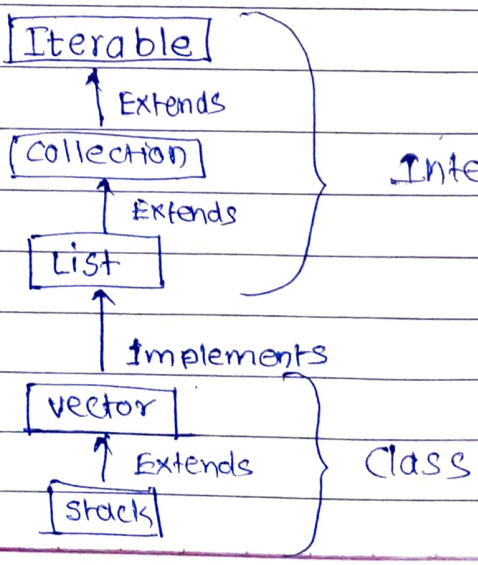
Insertion & deletion happens on same end.

Push - Adds items
Pop - Removes items
overflow stack - Stack is full but tried to Push
underflow Stack - Stack is empty but tried to pop.
peek/top - Returns top element of stacks.
isEmpty - Returns True is stack is empty, else false

Time complexities

* push(), pop(), isEmpty(), peek() all of them take O(1) time.

Java provides stack class which models & implements stack data structure



Methods used

1) Object.push(Object element) : - Push element at top
2) Object.pop() : - Remove & Return top element
3) Object.peek() : - Don't remove but returns top element
4) boolean Empty() : - Return boolean True if empty
5) int search (object element) : - determine object element present or not in stack if present returns position

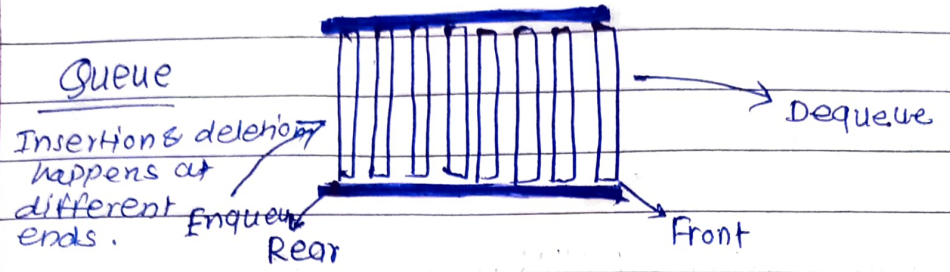
Applications

1) Managing function calls, Recursions
2) The stack span problem
3) Arithmetic expression evaluation (infix, Prefix, Post fix) Post fix or Prefix are evaluated by using only one stack
4) Reversing String.

Queue

Camlin | Page
Date / /

Queue - Linear Data structure follows particular order in which operations performed (FIFO).



Enqueue - Adds items
Dequeue - Remove items
Overflow - Full condition
Underflow - Empty condition
Front - Get front element
Rear - Get last element.

Time Complexity -
Enqueue(), dequeue(), isFull(), isEmpty(), front() & rear() is have O(1).

* Queue implemented by circular array than by linear array is efficient. because space is reused and take less space. for implementation indexes are found using (mod) (rear++ % size of array) given index in circular manner. also Enque & deque done in O(1) time with rear & front reference.
* Queue can be implemented with two stacks. two require for pop/dequeue operations. and stack implemented by two queues.

To implement Stack using Queue.
1) By making push operation costly.

Push(x) - Enqueue element (x) to queue (2) and dequeue all element of queue (1) to queue (2) and swap their names.
Pop(x) - simply dequeu (x).

we can implement stack by making pop operation costly.

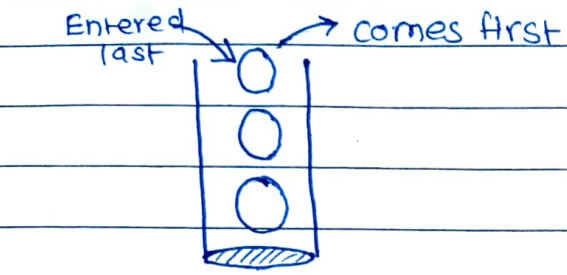
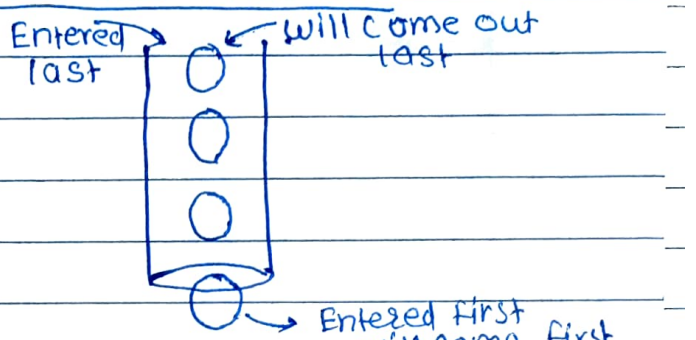
Camlin Page
Date / /

Priority Queue - Queue which doesnot use FIFO but deque according to priorities given to each node and implemation done by Heap data structure.

Application -

1) Dijkstra's Alg -
2) Prism's Alg -
3) Data Compression by Huffman code
4) Artificial intelligence - A* search Algo for shortest path in vertices of graph.
5) Heap sort - using Heap which is implementation of priority heap.
6) Operating systems, Load balancing on Server
7) Traffic lights, - depending on traffic priority given to colours.

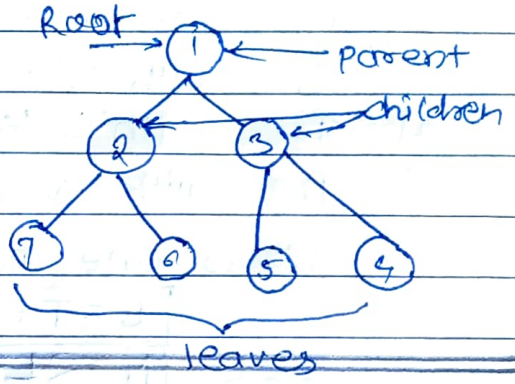
LIFO & FIFO

LIFO	FIFO
1) Last in First out 	1) First in First out 
2)	2)
3) Method of handling Data structure	3) method of handling data structure
4) New Element inserted above Existing can taken out first	4) New Element inserted below existing taken at last.
5) Used as operating system algorithms, which gives every process CPU time in the ~~order they arrive.~~	5) In computing LIFO approach is used as a queuing theory that refers to the way items stored in ~~type~~ type of data structure, (wrong)
6) The data structure that implements LIFO is stack and variant of stack	6) The data structure that implements FIFO is Queue, and variant of queue.
7) Extracting latest information by computers when data ~~stored in Array or data~~ buffer when require to get most recent information ~~entered~~.	7) Disk scheduling - disk controllers can use FIFO as disk scheduling algorithms to determine order in which to service disk I/O requests.
	8) communications & networking bridges, switches & routers used in computers network use FIFO to hold data packets en route to their next destination.

Binary Tree

Camlin Page
Date / /

Binary tree - ↑ Hierarchical Data structure of type tree having atmost two childs (left or right).



Binary tree node has
i) Data
ii) Pointer to left
iii) Pointer to right

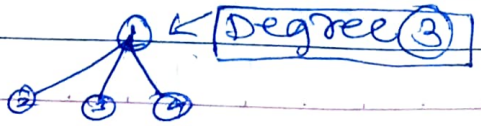
Tree vocabulary

Top Node/Root - Top most nodes
Children - elements directly under element.
Parent - element directly above element
leaves - element with no children.

Advantages/Disadvantages

i) Used to store information which naturally forms hierarchy
ii) Tree provide moderate search & access (quicker than linked, slower than array).
iii) Tree provide moderate insertion & deletion (quicker than array, slower than linked list).
iv) Tree don't have ~~limit on~~ upper limit on number of nodes. Nodes linked by pointer like linked list unlike array.
v) Make easy to search information by tree traversal.
vi) Router algorithms.
vii) Form of multistage decision making.
~~viii) Height is not under control~~

Degree of Node is the no. of child possessed by node and degree of Tree is max degree node.



Properties *(when for one node with consider height)

1) Max. no. of nodes at level 'I' is 2^I
2) Max. no. of nodes in Tree of height 'h' is $2^{h+1}-1$
3) Tree with 'N' nodes minimum possible height or levels are
4) Tree with 'L' leaves has at least $(\log_2 L)$ levels.
5) Tree with every node having 0 or 2 child have no. of leaves one more than no. of total internal nodes.

$L = T + 1$ (L is leaves, T - internal nodes.)

$N = 2L - 1$

~~Types~~

Program:

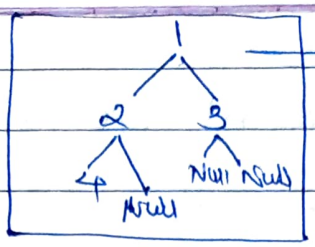
```
class Node{
    int key;
    Node left, Right;
    public Node(int item) {
            key = item;
            left = right = null;
        }
    }
class BinaryTree{
    Node root;
     BinaryTree(int key){
            root = new Node(key); }
     BinaryTree() {
            root = Null; }
main function {
    BinaryTree tree = new BinaryTree();

     tree.root = new Node (1);
     tree.root.left = new Node(2);
     tree.root.Right = new Node(3);
     tree.root.left.left = new Node(4)
}
}
```

Can every Full tree be complete tree or vice versa? No

Camlin Page
Date / /

Output -

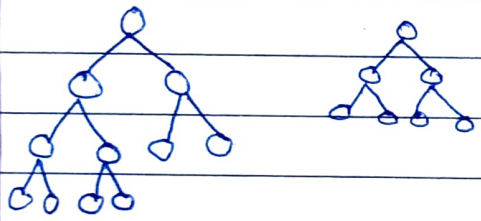


This kind of Data stored.

Types

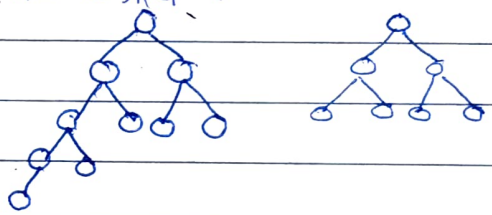
1) Full Binary / Strictly / 2/

Tree with every node having 0 or 2 child.



2) Complete Binary

Tree with all levels completely filled and last level is as left as possible. and can have one child.



$L = T + 1$

Property 5

where root = h = 0

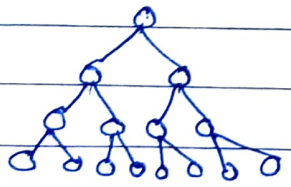
$n_{max} = 2^{h+1} - 1$

$n_{min} = 2^h - 1$

Internal node = floor(n/2)

3) Perfect Binary

Tree with all nodes have two child & leaves at some level.



root = h = 0 then

$2^{h+1} - 1$ = nodes

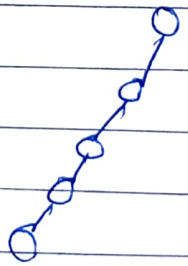
Property 2

otherwise root = h = 1

$2^h - 1$ = nodes

4) Degenerate Tree / Pathological

Tree when have every node with single child only



Performance same as Linked list.

5) Balanced Tree.-

Tree having height O(logn) wher n is no. of nodes

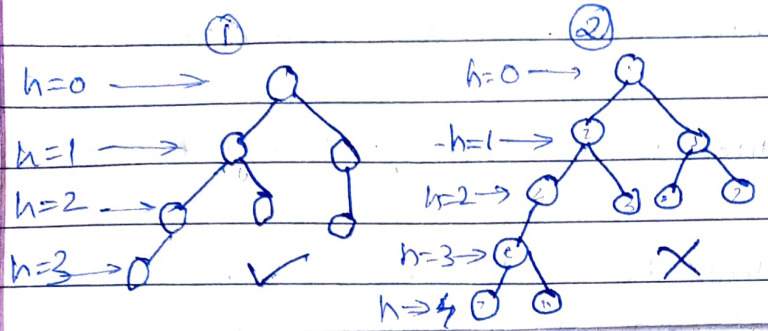
e.g. AVL & Red black trees

Self balancing & self adjusting Binary trees AVL, RB, BFs

time for rotations of tree?

AVL Tree

AVL - self balancing tree where difference in heights of left and right subtree is not more than 1.



Max nodes = $(2^{H+1} - 1)$

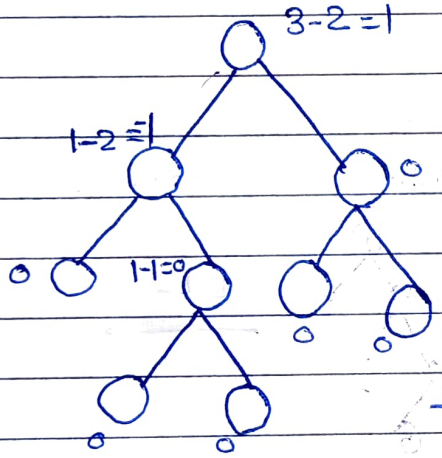
Where H is height

eg. Here if nodes = 15

if nodes = 31

Balance factor = height (left subtree) - height (Right subtree)

Balanced tree - If balance factor is within -1 to 1



Balance factors for each node ∴ hence balanced

left heavy - Balance factor ≥ 0

Right heavy - Balance factor ≤ 0

Balanced - Balance factor = 0

Complexities

Algorithms	Average case	Worst case
Space	O(n)	O(n)
Search	O(logn)	O(logn)
Insert	O(logn)	O(logn)
Delete	O(logn)	O(logn)
for rotation	O(1)	O(1)

Advantage

i) In Binary search tree time/operation is O(h) and can extend to O(h) in worst case if its get skewed. But AVL don't let tree to get skewed and limit hight to logn hence time/operation is O(logn).

ii) we use AVL if searching Traversing is frequent & insertion deletion is rare as it takes lots of rotations.

Operations

1) Searching
2) Traversing

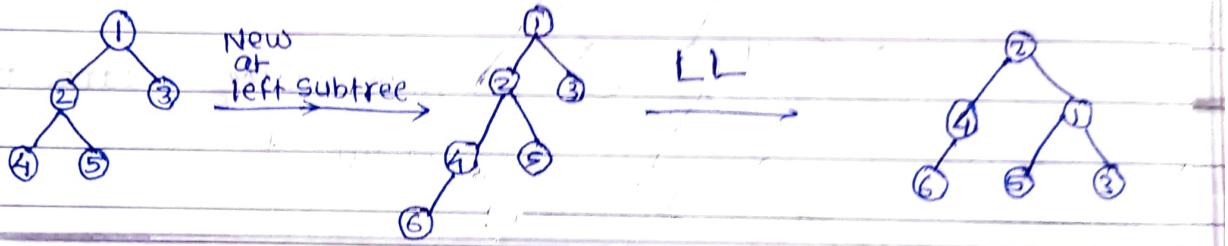
} Same as in Binary Search tree

3) Insertion
4) Deletion

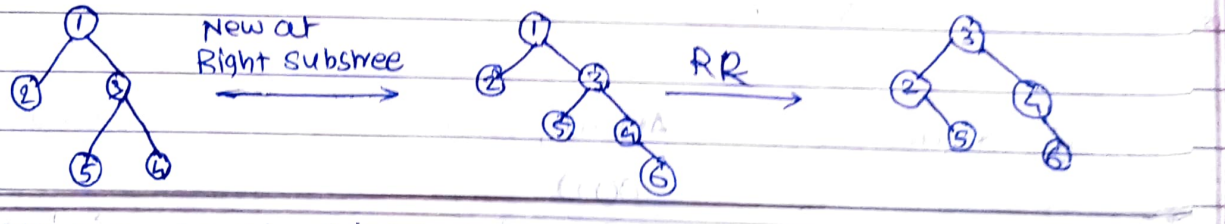
} Same as in Binary search tree But it violation of AVL rule that is of height and balance factor hence Rotations performed to balance it.

Rotations

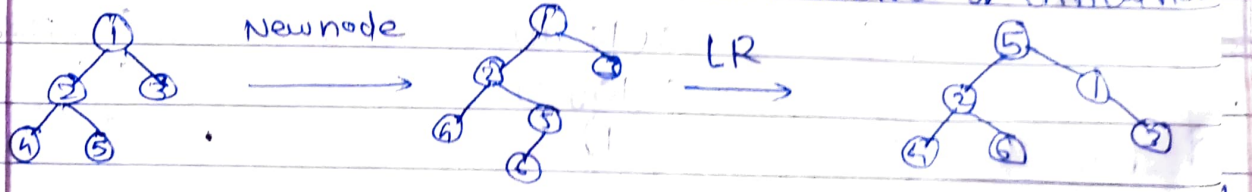
LL New node at left subtree of left subtree of critical node



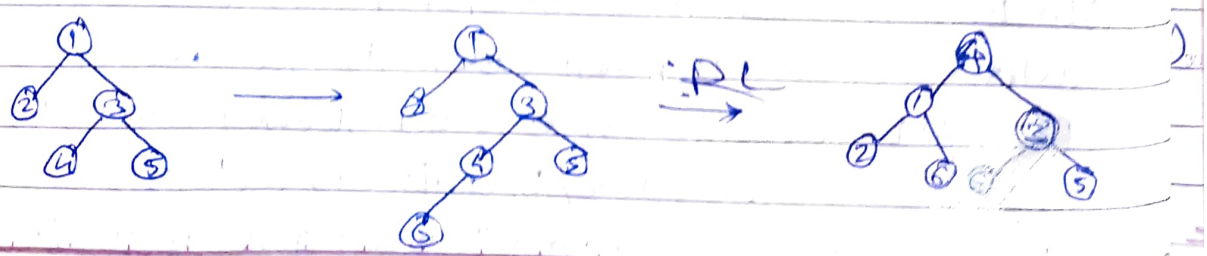
RR New node at Right subtree of Right subtree of critical node



LR New node at Right subtree of left subtree of critical node



RL New node at left subtree of Right subtree of critical node



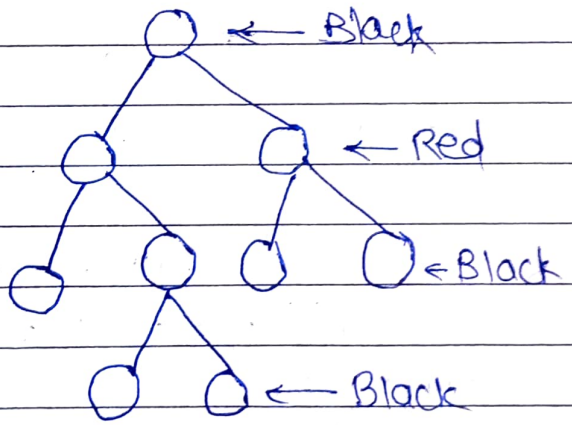
Red Black Tree

Red Black - self balanced Binary tree Data structure with properties as -

i) Leaves & Roots are Black always.
ii) Red Parents have Black childs only, & have Black parent.
iii) Every path from node to its leaves node contain equal number of black nodes.
iv) longest path is not more than twice the shortest path from root node to leaves.
v) Null is considered as Black.

Time Complexity (n is no. of nodes)

Operations	worst case
searching	O(logn)
insertion	O(logn)
deletion	O(logn)



Operations

1) searching
2) Traversing

} This operations are same as in Binary search tree.

For Insertion & delete Basic operations are
i) Recolouring
ii) Rotation. (Left or Right)

3) Insertion -

case 1 - If New node is Root of tree.

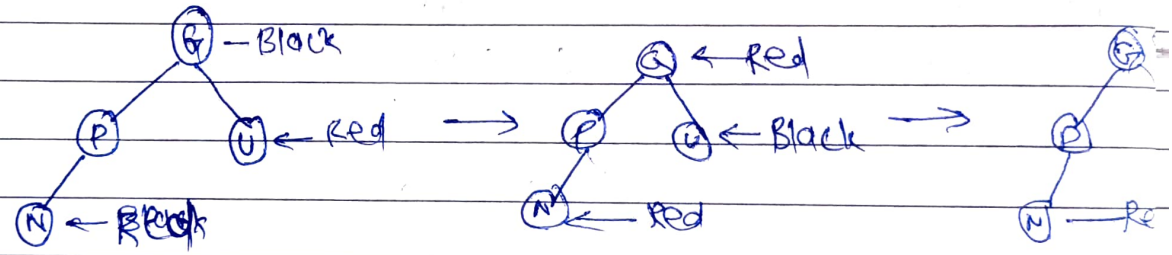
If new node don't have parent then it's a root no and Painted as Black.

case 2 - If New node have parent Black.

case 3 - Parent & uncle of new node is Red.

If will violate rule of same black node on path so have to repaint parent & uncle to Black

Also It may change root to red which violate fir rule so call process to make grandparent Blac



Null is considered as Black.

AVL trees	Red Black Tree
1) Min height = $\log_2(n)$	1) Height <= $2\log_2(n+1)$
2) Max height = $1.44\log_2(n)$	2) max no. Black node >= $(\frac{n}{2})$
3) Min no. of nodes with(h) $N(h) = N(h-1) + (h-2) + 1$ ---with $N(0)=1$ $N(3)=6$ $N(1)=2$ $N(2)=4$	3) Black height = $(\frac{h}{2})$ No. nodes on path from root to leave
4) Max ~~height~~ node = $(2^{H+1}+1)$	

AVL tree is strictly balanced than RedBlack hence hence fewer rotations required for deletion & insertions. here RedBlack preffered for the operations delete & insert are frequent but for searching only AVL is prefered also AVL tree stores balance factor with each node hence need more space. RedBlack take only 1 Bit of me

Binary search Tree

Camlin Page
Date / /

Binary search tree - Binary Tree Data structure which have following properties.

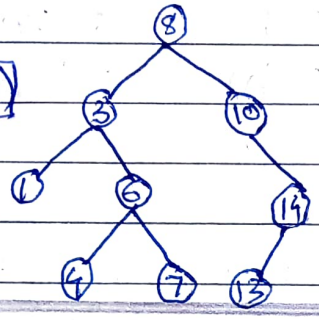
i) The left subtree have nodes with key value less than nodes key value.

ii) The Right subtree have nodes with key value greater than node's key value.

iii) No duplicate node they must be binary search tree, typed subtrees on both side.

Average

height = log(n)



Operations

1) Searching key

i) comapare element to found with root if marches, return location of root.

ii) If element is less than root moves to left branch

iii) If element is Greater than root moves to right branch

iv) Repeat until leaves checked if not found return null.

2) Insertion

i) Allocate memory to tree

ii) Set data to value, pointers to null.

iii) New element if less than node go left else right

iv) Item is inserted. process recursed.

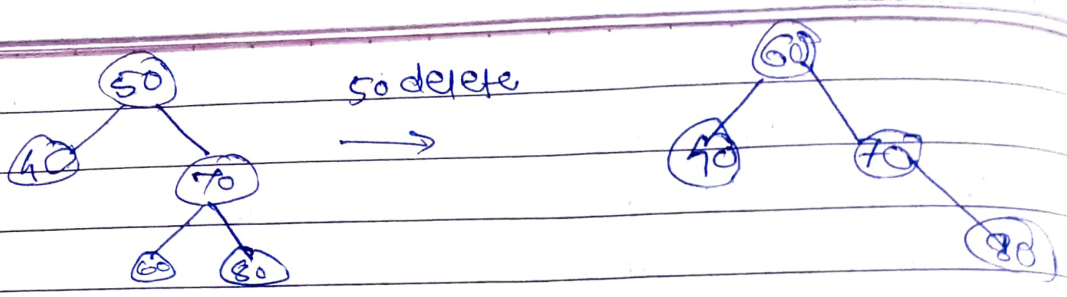
v) Height not under control.

3) Deletion

i) If node at leaves to be delete delete simply.

ii) If node have one child make child a node & delete node.

iii) If node have two child find inorder successor copy to node & delete it.



Time complexity

	Avg case	Best case	Worst case
searching	O(h)	O(logn)	O(n)
Insertion	O(h)	O(logn)	O(n)
deletion	O(h)	O(logn)	O(n)

IF insertion is such that height becomes equal then time taken to search is O(n) in worst co .hence we use AVL tree which arrange its c such that tree becomes balance.

For AVL tree

1) Max nodes = $2^{H+1} - 1$

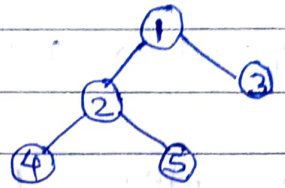
2) Min height = $\log_2 N$

Tree Traversal

Camlin | Page
Date / /

Tree Traversal - Process of visiting and printing value of every ~~root~~ node of tree

I) **level traversal (Depth) Traversal**



Tree

1) **In order**

(Left Root Right)

Visit left subtree ⇒ go to root ⇒ visit Right subtree

In order ⇒ 4 2 5 1 3

Application

1) To get nodes in non decreasing order.

Pre-order

(Root, left, Right)

Visit Root ⇒ go to left Subtrees ⇒ go to Right Subtree

Pre-order - 1 2 4 5 3

Application

1) To create copy of tree
2) To get prefix of expression tree.

Camlin | Page
Date / /

Post order

(left Right Root)

visit left subtree ⇒ go to Right Subtree ⇒ go to root

Post order - 4 5 2 3 1

Application

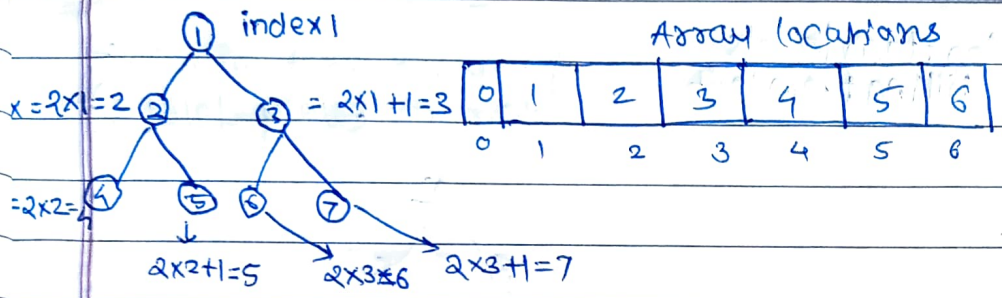
1) To delete the tree
2) To get postfix expression of expression tree.

Heap

Heap is tree based data structure in which all nodes are in specific order, It is complete binary tree.

Types - max heap - Parent node is always greater than child node i.e node value.

min heap - child node is greater than parent



Basic operations - 1) Insert / push
2) Remove / Pop
3) examine / Peek

Internal operation - heapify - maintain (max/min) heap property

Applications

1) To implement priority queue
2) To find shortest path bet vertices in Graph structure such that total sum of weights of constituent edges is minimum.
By min heap or Dijkstra algorithm.

Heaps in Java

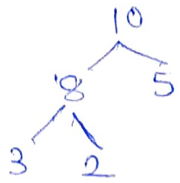
Binary heap tree data structure used to implement priority queues including java.util.priorityQueue & Java.util.concurrent.priorityBlockingQueue.

Time complexity to insert/delete/reduce key = O(log n)

to build heap = O(n) required to build Heap

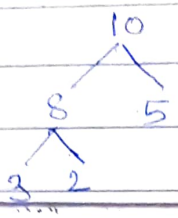
Heapsort = O(n log(n)) O(log(n)) for heapify & O(n) calls made.

Max Heap

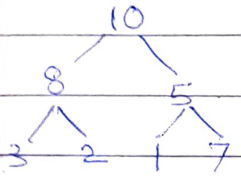


BSF / Level order traversals and insertion & deletions.

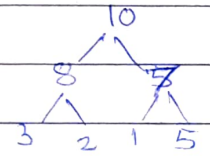
Insert (1, 7)



1) Insert at bottom levels filling from left



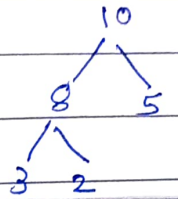
2) Replace according to priority from bottom



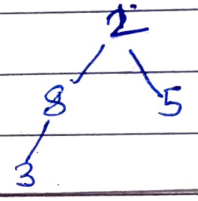
3) Heapify complete

To insert, insert at empty level from left then heapify until level orders satisfied.

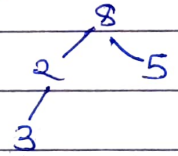
delete (10)



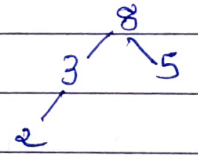
1) Replace root with last element (every delete)



2) Replace according to prio



3) Replace



To delete replace root with very last element at right most at last lvl then heapify until level order satisfy.

Breadth first search / Level order traversal print each level from root to bottom and left to right.

10, 8, 5, 3, 2

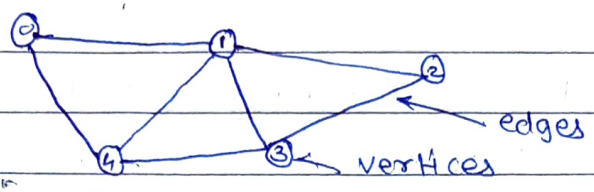
Q. For array 25, 14, 16, 13, 10, 8, 12 After two delete

→ 14, 13, 12, 8, 10

Graph

Graph - Non linear data structure consisting of data nodes and edges. nodes are also called as vertices & edges as line or arc.

G(V,E)



Set of vertices = {0,1,2,3,4}

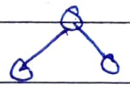
set of edges = {10, 14, 13, 34, 04, 14, 3}

Types

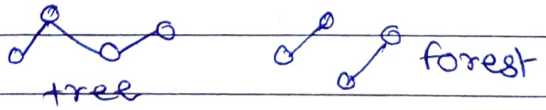
1) undirected / Directed - undirected can traverse in both directio
directed edges travels in give direction

2) Cyclic / Acyclic - Cyclic has at least one cycle of node
Acyclic has no cycles.

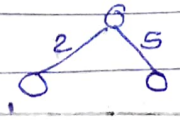
3) Connected / Disconnected - connected has no unreachable vertices
Disconnected has unreached vertices.



4) Tree / Forest - Tree is Asyclic and connected graph
forest is having many tree



5) weighted / unweighted - weighted has weight attached (value) to edges unweighted don't have any value to edges.



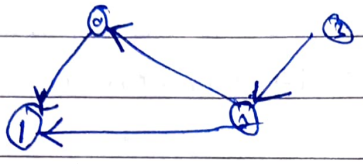
Vertex and Graph ~~tree~~ degree

vertex degree is no. of edges attached to vertex

Graph degree is highest vertex degree from deg(

deg(1) = 2, deg(2) = 3 $\Delta G = 3$

vertex indegree / outdegree - degrees coming to a of leaving vertex.



deg+(0) = 1 , deg-(0)

deg+(1) = 2 , deg-(1)

deg+(2) = 1 , deg-(2)

deg+(3) = 0 , deg-(3)

Graph Representation

1) **Edge list** - Array of all edges where each edge by array of two vertices.

e.g. int[][] Graph = {{1,0},{0,2},{1,2},{2,3

2) **Adjecancy list** - Array having index as vertex value value of index has array of vertex's adjacent.

e.g. int [][] Graph = {{1,2},{0,2},{1,0,3},{2}}

index 0 ⇒ 1 ⇒ 2 ⇒ 3

Values of graph. ⟶

3) **Adjacency matrix** - matrix of 0's and 1's in whi 0 and 1 shows connection of two vertices in row represent source of vertices and column des vertices.

e.g. int [][] graph = { {0,1,1,0},
{1,0,1,0},
{1,1,0,1},
{0,0,1,0} };

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Implementation by adjecency list over adjecency matrix is easy because of less space required, DSF & BSF complexity for adjecency list is $O(V+E)$ while $O(V^2)$ for matrix. and addition is also easy.

Basic Graph Operations -

Add edge : add edge between vertices of graph
Traversal : traversal in graph by DFS & BFS

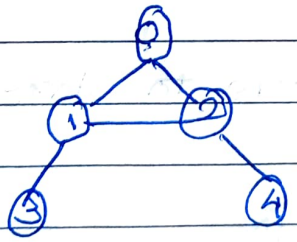
Java don't have default graph implementation However we can use other supporting data structure to implement it.

Graph traversals (DFS / BFS) (algo)

1) **DFS** - starts at arbitrary node and explores as far as possible along each branch before backtracking hence DSF is useful when target is far from node.

DFS → 0 → 1 → 3 → 2 → 4

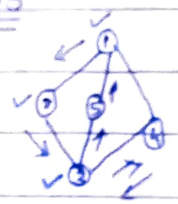
1) go to node
2) go to last node as possible
3) Backtrack
4) go to depth.



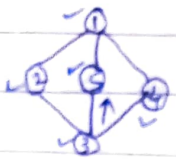
It uses stack to store data.

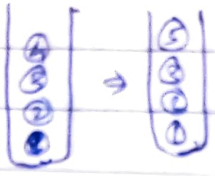
DFS Algorithm

Steps

1) 

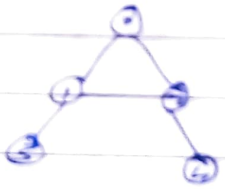
1) Go to adjacent matrix
2) mark as readed
3) push to stack.

2) 



1) If node don't have a
node pop it and
2) find adjacent now
3) Repeat processes.

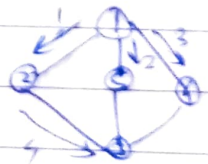
BFS - Starts with arbitrary node explores all ne
nodes at one level then go to nextlevel.
at next level follow same process to store wh
node has been reached or not use boolean
Don't repeat reading for any node. any no
readed only once.



BSF - 0 → 1 → 2 → 3 → 4

It stores Queue to store data.

BSF algorithm

steps 1) 

To find the shortest path in graph
we use (BFS) Dijkstra algorithms to find the
path such that total sum of the weights of
edges is minimum.

we prefere DSF over BSF because it takes linear memory while BSF takes exponential.

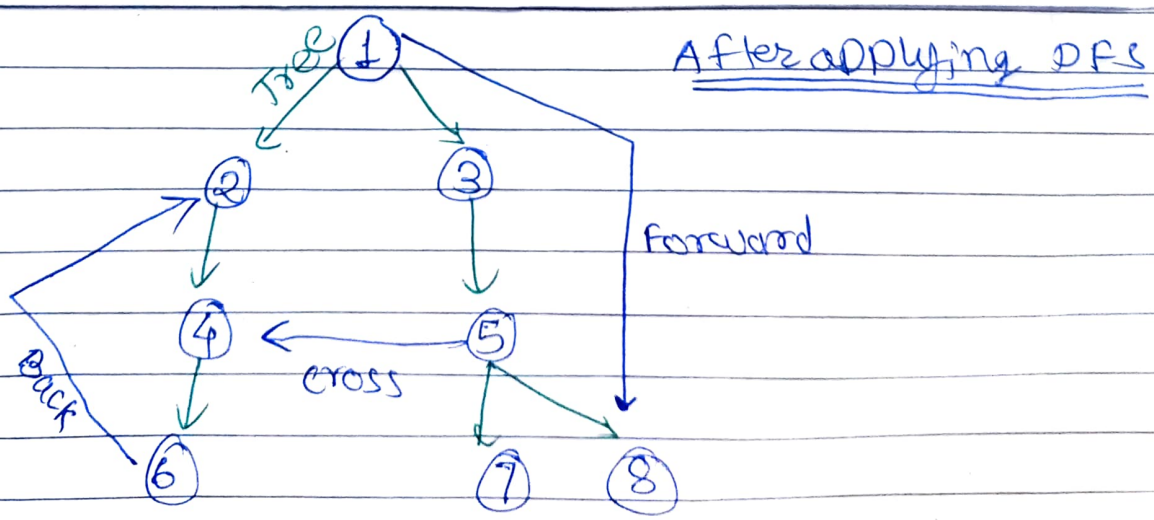
Edges in Graphs

Tree edges - Edges that presents in tree obtained after applying DFS on the graphs. (all green edges)

Forward edge - Edges that represents edge(u,v) such that v is descendant of u but not part of DFS tree (Edge from 1 to 8)

Back edge - Edges that represents edge(u,v) such that v is ancestor of edge u but not part of DFS tree (Edge from 6 to 2) It make cycle.

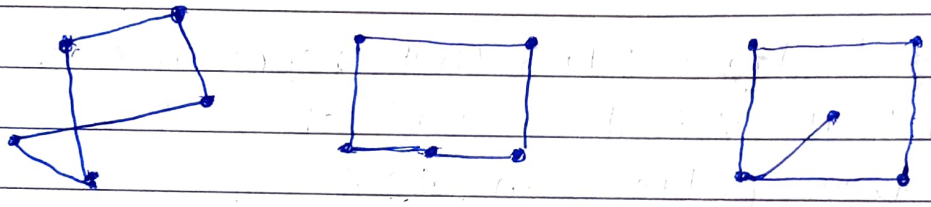
cross edge - Edge that connects two nodes such that they do not have any of ancestor or descendent relation ships between them after DFS tree (Edge from 5 to 4)



Dijkstra's
prism
Kruskal's
} algorithms used in graph to find shortest path.

Isomorphic graphs

Two or more graphs having same degree, same no. edges and same no. of vertices



G_1 G_2 G_3

$G_1 \equiv G_2$ $G_1 \equiv G_2 \neq G_3$

When their are n vertices then possible no. of isomorphous graphs are $\frac{n-1}{2}$

Applications of DS

Linked list - i) **Image viewer** - Prev and next images are linked
ii) **Prev. & next Pages on web**
iii) **Music player**
iv) **circular linked lists** - used by computer running many application to give equal time to each application & repeat cycle.
v) **Cashe in browser** - To hit BACK Button linked list of URLs can be implemented
vi) **Building Stacks & Queues** - using doubly linked lists
vii) Managing relational databases

Stack -

1) **Undo** - Undo mechanism while writting the texts by keeping all text changes in stack.
2) stack is usefull in Backtracking which is a process when you need to access the most recent data elements in a series of elements.
3) **Implementing recursive functions** - The address and returns of methods/functions stored in stack and when last fuction stored it firstly invoked while poping.

Queue - i) Store interrupts in OS
ii) Used by application program to store incoming data. (Input output Buffer)
iii) Synchronisation in O.S. - When transfer is slow. iv) Job scheduling in CPU and disks sched
v) Round Robin scheduling -
vi) Recognizing Palindrome
vii) Undo & Redo in software application

Trees - i) Syntax validation in compilers - Java checks grammatical structures of Java pr by reading the program's words and a to build the programming parse tree.
ii) Used in internet protocols
iii) Trees used in computer networking Used commonly in internet protocols Trees can be used as every high Ba router for storing router table.

Graph -

i) Google maps - for building transportation s
vertices ⇒ intersection of roads, edges ⇒ roads
and system find shortest path by algorit
based on to calculate shortest path between
vertices.

ii) Facebook - To suggest friends 'unidirected
used. Friends ⇒ vertices, edges runns be
them.

iii) web pages on World Wide web - webpages a
vertices and edges between pages are
this is 'directed graph'

Algorithms

Algorithm - set of instructions or finite no. of steps for finite sized inputs to solve particular problem.

Classification -

1) On the basis of Implementation method.

a) Recursive / Iterative -

i) **Recursive** - algorithm which calls itself again and again until base condition is satisfied.

ii) **Iterative** - algorithm that uses loops or Data structures like stack or Queues to solve any problems.

b) Exact or Approximate -

i) **Exact** - algorithm which is finding the optimal solution for any problem is exact algorithm. e.g. sorting algorithms.

ii) **Approximate** - algorithms which do not find most optimal solution of problem is the approximate algorithms.
e.g. NP hard problems.

c) serial or parallel or distributed

i) **Serial** - one instruction is excuted at a time

ii) **Parallel** - algorithm that divide problem into subproblem and excute on different processors.

Page No.:
Date:

Distributed Algorithm - If parallel algorithm distrib on different machines they are known as distr algorithm.

Classification Based on Design methods.

1) **Greedy method** - choose local optimum Solutio without taking into account further consequences.
e.g. Knapsack problems (fractional), activity selection.

2) **Divide and Conquer** - strategy to solve proble dividing them into smaller subpr recursively and solving them and results to get solution to bigger e.g. Quicksort, Merge sort.

3) **Dynamic programming** - The approach of Dy programming is to divide problem subproblems and solve them Divide and conquer only differen that it stores the solutions for use so that same subproblem n to tackle again.
e.g. 0/1 knapsack problem

Other classifications

1) **Randomized Algorithms** - algorithm that m Random choices for faster solutio are known as randomized algo
e.g. Randomized Quick sort.

2) Classification by Complexity :- algorithm that classification on the basis of time taken to get solution of any problem for input size. This is time complex analysis.

e.g. Some algorithm take O(n), Some take exponential time.

Greedy Algorithm

Camlin | Page
Date / /

Greedy - 1) Finding Locally optimized solution at every step which leads to globally optimized solution eventually. (However not always it give globally optimed solution).

2) Greedy mean to make which seems to be best at that moment or step. i.e. locally optimised.

3) It make greedy choice at each step to ensure function is optimised. It has only one shot to compute optimal solution and cannot go back to change choice.

Where to use Greedy Algorithm?

1) Optimal substructure - Optimal soln to problem has optimal solutions to each subproblems.

2) Greedy property - If you make choice that seems best at that moment and solve remaining subproblem later you still reach optimal solution.

Advantages & disadvantages

1) Easy to come up with greedy algorithm and Analysing time complexities also easy as compare to other algorithms like divide and conquer.

2) It is difficult to understand correctness of issues, proving that why it is correct. most of greedy algorithms are not correct/optimal.

Application

1) knapsack problem (fractional) - In bag of W weight capacity we need put different items such that it has max. value. (fraction are allowed).

10kg 60
20kg 100
30kg 120

$\left(\frac{60}{10} + \frac{100}{20} + \frac{120}{30} \times \frac{3}{2}\right) = 50kg$

$= 240₹$

120₹

we make greedy choice at each step by choosing item with max. value to weight ratioes first. we get m value.

Application

1) Activity selection problem
2) Fractional knapsack
3) Job Scheduling problem.
4) Minimum spanning tree
5) Dijkstra's algorithm for shortest path
6) Huffman code.

Greedy algorithm is not always optimal.

E.g. If we need count of 18 ₹ with coins 1 ₹, 5 ₹, 10 ₹ greedy choice will be like -

1) 10 ₹
2) 5 ₹
3) 2 ₹
4) 1 ₹

= 18 ₹

it took (4) coins. which optimal globally.

But if we want to make 14 ₹ from 1 ₹, 2 5 ₹, 7 ₹, 10 ₹ coins with using minimum

Greedy choice ⇒ 10 ₹ + 2 ₹ + 2 ₹ = 14 ₹

locally optimised choices with (4) coins.

Globally optimised solution ⇒ 7 ₹ + 7 ₹ = 14 ₹

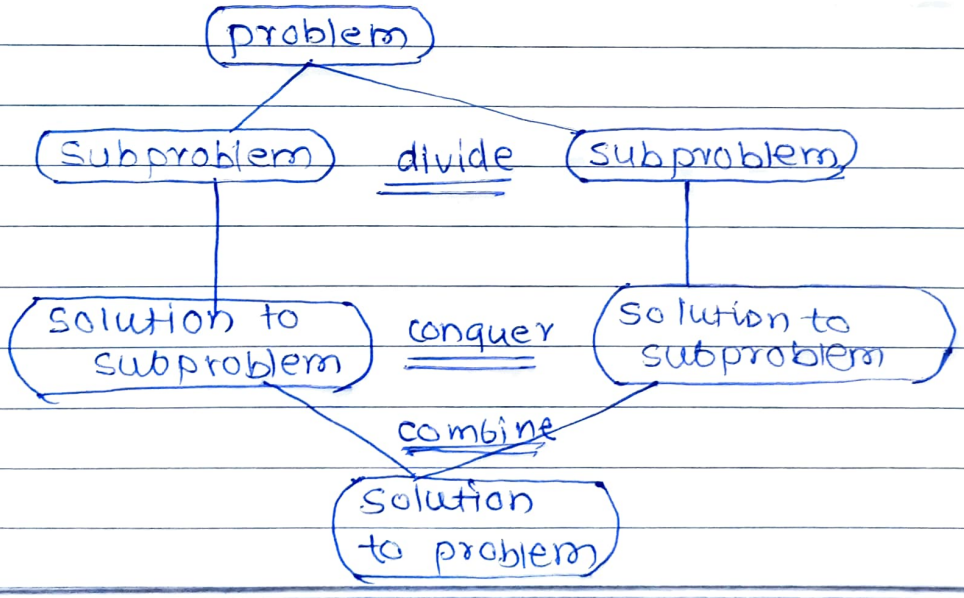
globally optimised solution without greedy choices. with only (2) coins

Divide and Conquer

Divide and conquer breaks a problem into subproblems similar to original problem recursively then solve those subproblems recursively and finally combine the solutions to subproblems to solve the original problem.

Steps

1) Divide - the problem divided into subproblems that are smaller instances of same problem i.e. recursively calling function
2) Conquer - solve subproblems recursively, If they are small enough solve them as base case.
3) Combine - combine the solutions of subproblem to get solution of original problem.



Fundamentals of Divide and Conquer

1) Relational formulae - formula we generate from given technique, which break problem recursively and solve the broken subproblems.
2) Stopping condition - we need condition to stop the recursive call. (Base case).

Applications

1) Maximum and minimum problems.
2) Binary search
3) merge sort
4) quick sort
5) Tower of Hanoi

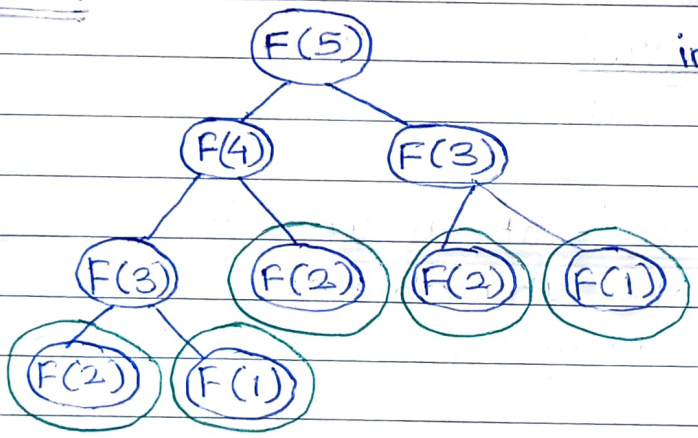
Dynamic Programming

M T W T F S S
Page No.:
Date:
YOUVA

Dynamic programing - Dynamic programing involves technique of problem solving where we divide problem into similar subproblems and solve subproblems and store the result of subproblems so that we need not to solve those subproblems again. then we combine solutions to subproblems to get solution of bigger main problem.
this is applicable when problem has overlapping subproblems.

for. e.g. to find nth element of fibonacci series.

Recursion




```
int fibo(int n){
  if(n<2){
    return 1};
  return(fib(n-1)+
         fibo(n-2)
}
```

When we call recursive function to find fibonacci number (nth). it call function as above we will have to find out F(2), F(1), F(3) multiple times and hence time complexity is increased.

∴ Time complexity = $T(n-1) + T(n-2)$
= Exponential time complexity

We can use Dynamic programming approach if Problem has -

1) Optimal substructure - If optimal solution optimal sub solutions then it is optimal sub i.e. we can't define recursive function.

2) Overlapping subproblems - When recursive fu calls same subproblem reapeatedly.

Elements of Dynamic programming.

1) Substructure - Decompose problem into sub

2) Table structure - After solving subproblems st stored in table

3) combination/ Bottom up computation - combi solutions of sub problems to solve larger s and eventually arrive at solution to comp problem.

e.g. to find nth element of fibonacci serie

```
int fibo(int n){
    int F[] = new int[n+1];
        F[0] = 0;
        F[1] = 1;
  for (int i=2; i<=n ; i++){
      F[i] = F[i-1] + F[i-2];
      };

  return F[n];
```

F	0	1	1	2	3	5
	0	1	2	3	4	5

Two approaches for DP

a) Memoization (Top Down)

1) In this we create an array and initialize it's values to NIL. for each elements.
2) When we need solution to subproblems we lookup to that lookup table if we found value their we return that value.
3) If not there then we calculate value store to lookup table for further use.

for e.g. to find n^{th} fibo element fibo(n)

```
public class fibo{
 final int Max = 100;
 final int NIL = -1;
 int lookup[] = new int[Max]; // we create NIL value array.

public void initialize(){
     for(int i=0 ; i<Max; i++){ // we initiase all element to NIL
         lookup[i] = NIL; }
 }

public int fibo(int n){
   if(lookup[n] == NIL){                  // if value not present in lookup table then run this block
         if(n<=1){
              lookup[n] =n;}
         else{
              lookup[n] = fibo(n-1) + fibo(n-2);}
   return lookup[n];                      // if value present in array simply take value from lookup table and return.
}
```

b) Tabulation (Bottomup)

1) we store solution to subproblem at sta
2) Combine stored solutions to get solution bigger problem.
3) Eventually we solve main problem.

for eg. to find nth fibo element fib

```
public class fibo {
public int fibo(int n){

        int [] F = new int[n+1];
                F[0] = 0;
                F[1] = 1;
        for (int i = 2 ; i<=n ; i++){
                F[i] = F[i-1] + F[i-2];}
        return F[n];
        }
```

Conclusion

Dynamic programming takes less time co
but more space complexity.
As explained by recursion time com
is exponential while by dynamic pr
i.e. memoization take less.

Recursion	Dynamic / Memoization
$O(2^n)$	$O(n)$

Time and space complexities

To decide whether algorithm we use is efficient or not we use two parameters.

i) Time complexity - number of operations hence time.

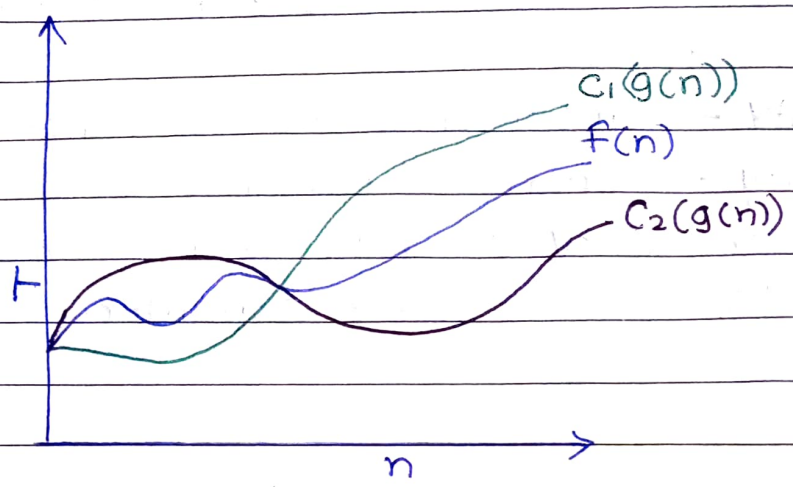
ii) Space Complexity - memory of machine used

time and space complexity depends upon hardware, OS, processors etc. but we don't consider this factors to calculate complexities.

* Why time complexity is important?

As take e.g. of searching element from 4 billion elements For Linear search algorithm it is estimated that it will take 4 billion steps and if 1 ms is required for each then it will be approximately 4 Days. while for Binary search alogorithm it will take only 32 operations and 32 ms is required. So that is the importance of choosing right algorithm according to time complexity.

Asymptotic Notation



$f(n)$ is the function which represent the var of time taken by algorithms with increasing Inputs (n)

Big O

$g(n)$ is the function of time & inputs such th after certain value of n i.e. (n_0) value of $f(n)$ be greater than $C(g(n))$ where C is constant. Hence $g(n)$ is the upper bound to function

$\therefore f(n) < g(n) \times C_1$ where $n > n_0$, $n_0 \geq 1$, $C_1 > 0$

If this conditions satisfy then we say tha

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

Camlin Page
Date / /

Big Ω -

g(n) is the function wh· of time and inputs such that after certain value of n ie. (no) value of f(n) cannot be lower than C_2 (g(n)) where C is consta Hence g(n) is the lower bound to function f(n)

$$\therefore f(n) \geq C_2 \cdot g(n)$$ where $n > n_o$, $n_o \geq 1$, $C > 0$

If this satisfies then we say that -

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

Big Θ

If their exists some function g(n) such that

$$C_2\, g(n) \leq f(n) < C_1 \cdot g(n)$$ where - C_1, C_2 are constants
- $n > n_o$
- $n_o \geq 1$
- $C_1, C_2 > 0$

Then we can say that -

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

Note - no is not same for Ω, Θ it can be vary

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n$

← lower bounds | ↑ Average for $(2n+3)$ | → upper bounds

Any function that has values greater than function can be upper bound or Any fu that has value lower than given function lower bound. But if their is function can be upper bound or lower bound wit variation of constant multiples can be c as Average bound denoted by Θ.

for e.g. lets $f(n) = 2n+3$ that means time by algorithm varies with no. of inputs (n) accordance with this function.

$$C_2 g(n) \leq f(n) < C_1 g(n)$$

Lower bounds can be	function is	upper bounds can b
$\Omega(n) = 1 \times n$	$\leq (2n+3) <$	$2n+3n = O(n)$
$\Omega(\log n) = \log(n)$		$7n = O(n)$
$\Omega(\sqrt{n}) = \sqrt{n}$		$2n^2+3n^2 = O(n^2)$
$\Omega(\log n) = 2\log(n)$		$2n^3+3n^3 = O(n^3)$
$\Omega(1) = 1$		$2^n+3^n = O(2^n)$

$f(n) = 2n+3$

→ $C_1 g(n) = 7n = C_1 \cdot g(n) \Rightarrow$ upper bound

→ $C_2 g(n) = 1 \times n = C_2 \cdot g(n) \Rightarrow$ lower bound

Here $C_1 = 7$, $C_2 = 1$ and $g(n) = n$ hence

$C_2 g(n) \leq f(n) < C_1 g(n)$ hence condition for theta

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

There can be any function which can be satisfy of upper or lower bound but we need to ch the closest functions as upper or lower bounds our function of algorithms.

Camlin Page
Date / /

Space Complexity - Amount of memory used by the algorithms to execute and produce the result.

Space complexity = (Constant space / Input space) + Auxillary space

Memory usase while execution.

1) Instruction Space - To save compiled version of program

2) Envirnmental stack - When one algorithm calls another then current variables pushed to system stack temporarily wait for further excution and call to inside algorithm made. e.g. If method A() calls method B() temporary A() variables pushed to stack till B() is executed.

3) Data Space - space used by variables and constants.

Note - While calculating space complexity we consider only Data space and Instruction space and Envirmental stack is neglected.

Hence some memory is required when program execute for -

1) Program instructions
2) Execution
3) Variables (which are constant values or temp. value)

S(P) = (Constant / Input space) + Auxillary space.

1) Constant space is one which is fixed for algorithm. generally space occupied by input and local variable

2) Auxillary space is extra/temporary space used by algorithms

e.g.
```
int a[] = new int[n];
int n = n;
Sum(a, n){
        s = 0;
        for (int i=0; i<n; i++){
            s = s + a[i]; }
        return s };
```

Here space taken by variables a[], n, s, i

If int takes 4 bytes then -

a[] ⇒ 4 * n
n ⇒ 4
s ⇒ 4
i ⇒ 4

Space complexity = Constant
= (4*n+4+4+4)

Space complexity = (4n + 12) = O(1)

Space complexity is linearly increases with, hence it is Linear time complexities.
Similarly we can have constant, quadratic time complexities.

e.g.
```
int a, b, c;
int z = a+b+c;
return (z);
```

Here

a ⇒ 4
b ⇒ 4
c ⇒ 4
z ⇒ 4
return ⇒ 4

Space complexity = 4*4+4 = O(1)

Here time complexity is constant.

for 1st e.g. here

Space complexity = Constant/input space + Auxillary space
= (a[], s, n) + (i)

here a[], s, n are terms which are constant and input terms whereas (i) is temperory term and we often neglect the constant part and use only auxillary part to calculate space complexity.

for whole costant & Auxillary

$\therefore S(p) = (4*n+4+4) + (4)$

$= O(n)$

for auxillary only

$S(p) = (4)$

$= O(1)$

time Complexities

for loops

`for (i=0, i<n, i++){ count++}`	n times	$O(n)$
`for (i=0, i<n, i++){ for (j=0, j<n, i++){ count++ }}`	1 2 ... n ↓ ↓ ... ↓ n n n $f(n) = n * n$ times	$O(n^2)$
`for (i=0, i<n, i++){ for(j=0, j<i, j++){ count++}}`	1 2 ... n ↓ ↓ ... ↓ 1 2 n $f(n) = \frac{n(n+1)}{2}$ times	$O(n^2)$
`for(i=1; P<=n; i++){ P=P+i; }`	P P ... P ↓ ↓ ... ↓ 1 2 k $f(n) = \frac{(k+1)k}{2}$ $n = k^2$ $\boxed{k = \sqrt{n}}$	$O(\sqrt{n})$
`for(i=1; i<n; i=i*2){ count++;}`	$1 \times 2 = 2$ $2 \times 2 = 2 \times 2$ $4 \times 2 = 2 \times 2 \times 2$ $\vdots$ $k \times 2 = 2^k$ $2^k = n$ $\boxed{k = \log_2 n}$	$O(\log_2 n)$
`for(i=n; i>1; i=i/2){ count++}`	i/n $n/2^2$ $n/2^3$ $n/2^4$ $n/2^k = 1$ $n = 2^k$ $\boxed{k = \log n}$	$O(\log_2 n)$

for(i=0; i*i<n; i++){ count++}	i*i = n 1*1 = n 2*2 = n ⋮ k*k = n $k^2 = n$ $k = \sqrt{n}$	$O(\sqrt{n})$
for (i=0, i<n, i++){ p++ } for (j=0, j<p, j++){ count++}	$p = \log(n)$ $k = \log p$ $k = \log\log(n)$	$O(\log\log n)$
for(i=0, i<n, i++){ for(j=1, j<n; j=j*2)}	$n \times \log n$ $+ n \times \log n$ $+ \quad n$ $2n\log n + n$ $\approx n\log n$	$O(n\log$

Summary

for (i=0; i<n, i++) — $O(n)$
for (i=0; i<n, i=i+2) — $O(n)$
for (i=n; i>1, i--) — $O(n)$
for(i=0; i<n, i++){
  for (j=0; j<n, j++)} — $O(n^2)$
for(i=0; j<=n, i++){
  for(j=0; j<i, j++)} — $O(n^2)$
for(i=0; p<=n, p++){
  p=p+i } — $O(\sqrt{n})$
for (i=0; i<n; i=i/2) — $O(\log n)$
for(i=n; i>n; i=i*2) — $O(\log n)$
for (i=0; i*i<n; i++) — $O(\sqrt{n})$
for (j=0; i<n; i++){
  p++}
  for(j=0 j<p, j++){ — $O(\log\log n)$

while loop

i=0 while(i<n){ i++ }	n times	$O(n)$
a=1 while (a<b){ a = a*2}	a = 1*2 = 2*2 = 2*2 ⋮ $b = 2^k$ $k = \log b$ $k = \log n$	$O(\log n)$
i=k=1 while (k<n){ k = k+i i++}	k = k+i = 1+1 = 2+2 = 2+2+3 = 2+2+3+4 ⋮ $n = 2+2+3+4\ldots m$ $n = \frac{m(m+1)}{2}$ $m = \sqrt{n}$	$O(\log\sqrt{n})$

when loop is executed with increament of certain power or constant then complexity is $(\log\log n)$

Data structure time complexity & space comple

Array & LinkedList

Operations	Array list	Linked list	prefer
insert at first	O(N)	O(1)	LL
insert at last	O(1) O(N) when copy is required	O(1) If have last reference otherwise O(N)	LL
insert at index	O(N) need to move all rest elements	O(N)	LL
remove by index	O(N)	O(N)	LL
search by index	O(1)	O(N)	Arr
remove by ~~value~~ ~~index~~	O(N)	O(N)	LL
Search by value	O(N)	O(N)	Arr

Except insert at last without copying to new array & insert at last of linked list & search by index of array every one has O(N) time complexity.

As we either need to traverse whole list or rearrange the list after removal of element.

BT - Binary Tree

i) insert ii) delete iii) search

we assume worst case complexity as we travel with Breadth first search.

$\therefore t = O(n)$

Binary search tree

i) Insertion ii) deletion iii) search.

Time complexity is $O(h)$ where 'h' is height

i) In worst case tree is skewed hence $h = n$

$\therefore t = O(n)$

ii) In Best (average) case tree is balanced hence $h = \log$

$\therefore t = O(\log n)$

Red black tree and AVL tree

i) insert ii) delete iii) search

As these trees are balanced form of BST th
both time complexities worst & Best case are
same as BST with balanced property that
$t = O(\log n)$

Stack & Queue

Stacks - Implemented by Array & linkedlist all oper-
like peek, push and pop have $O(1)$ comple-
as we have top reference of LL or array on w
all operations performed.

- If we need to copy data then push becomes O

Queue - Implemented by Array and Linkedlist o
like enqueue, deek are $O(1)$ but dequeue is
$O(N)$ because of shifting in array but in
LL it is $O(1)$ when both Head tail referen
are known

Also enqueue becomes $O(N)$ when copying i
required.

Space Complexity

All of Data structure have space complexity of [O(N)]

M T W T F S S
Page No.:
Date:

Sorting Algorithms

	Algorithms	Best	avg	worst	Space
1)	Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
2)	Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
3)	selection sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
4)	Quick sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log N)$
5)	Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
6)	Heap sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
7)	Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
8)	Bucket sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$

searching algorithms

Algorithm	Best	worst
Linear	$O(1)$	$O(N)$
Binary	$O(1)$	$O(\log N)$

Quick sort take less space than merge sort hence usefull for array However merge sort is best for Linkedlist as quick sort not suitable because of indexing.

Hash table

Hashing can be used to build, search delete records from table.

Hash Table - Data structure that store records in array called hash table.

Hash tables are preffered for searching, inserting operations but when inserting & deleting operations are frequent we use ordered tree data structures (e.g. AVL).

Load factor - $\frac{\text{No. of values}}{\text{size of array}}$

Hash function - fuction of hash function is to take values convert them into index value such that it creates less collision.
Goal is to achieve hash function such that probability of any two value have same slot is $\frac{1}{N}$

It should be - 1) Minimize collisions.
2) Key should very close or very far
3) should not generate large keys and wast space.

Collision - After hash function two keys (values) have same index of slot i.e. collision.
No matter how effective hash function is there always a probability of collision.

Goal is to have probability of two having same position is $\boxed{1/N}$

Collision is solved by -

Chaining -

Hash function $h(k) = k \% n$

0 → 72 ⇒ $72 \% 8 = 0$

1 →

2 → 10 ⇒ 18 ⇒ $10 \% 8 = 18 \% 8 = 2$

3 → 43 ⇒ $43 \% 8 = 3$

4 → 36 ⇒ $36 \% 8 = 4$

5 → 5 ⇒ $5 \% 8 = 5$

6 → 6 ⇒ $6 \% 8 = 6$

7 → 15 ⇒ $7 \% 8 = 15$

↑ Array Indexes with Key Index After Hashing

↑ linked lists with keys

↑ take mode of key with size of array or total number of slots.

Average Length - $\frac{N}{M} = \frac{\text{(Size of Array)}}{\text{(No. of Non empty LL)}}$

Open addressing - To find next empty index

1) Linear probing - when two element gives same index we find next empty space to given index. If you end up filling slots at last we move to beggining.

0	1	2	3	4	5	6	7
72	15	18	43	36	10	6	5

36%8 = 4 43%8 = 3 5%8 = 5
18%8 = 2 6%8 = 6 15%8 = 7
72%8 = 0 10%8 = 2

Quadratic Probing

If h(k) gives collision that is two value have same slot then we try for Index obtained by -

(hash(k) + 1*1) %S
(hash(k) + 2*2) %S
(hash(k) + 3*3) %S so on until we get empty slot.

Double probing / Hashing

If collision occur for h(k) then we use method

use $h(k) + 1 * h_2(k)$
$h(k) + 2 * h_2(k)$
$h(k) + 3 * h_2(k)$
$h(k) + 4 * h_2(k)$ so on

$h_2(k)$ is the second function and should not return 0 value

Most popular $h_2(k)$ is

$h_2(k) = \text{Primeno.} - (\text{key} \% \text{primeno.})$

where prime no. is just smaller than table size

e.g. 89, 18, 49, 58 and 69. assume 10 slo

$h(89) = 89 \% 10 = 9$

$h(18) = 18 \% 10 = 8$

$h(49) = 49 \% 10 = 9$

$= h(49) + h_2(49) =$

$= h(49) + 7 - (49 \% 7) =$

$9 + 7 = 7^{th}$ from 9

$h(58) = 58 \% 10 = 8$

$= h(58) + h_2(58)$

$= h(58) + 7 - (58 \% 7)$

$= 8 + 5 = 5^{th}$ from 8

$h(69) = 69 \% 10 = 9$

$= h(69) + h_2(69) =$

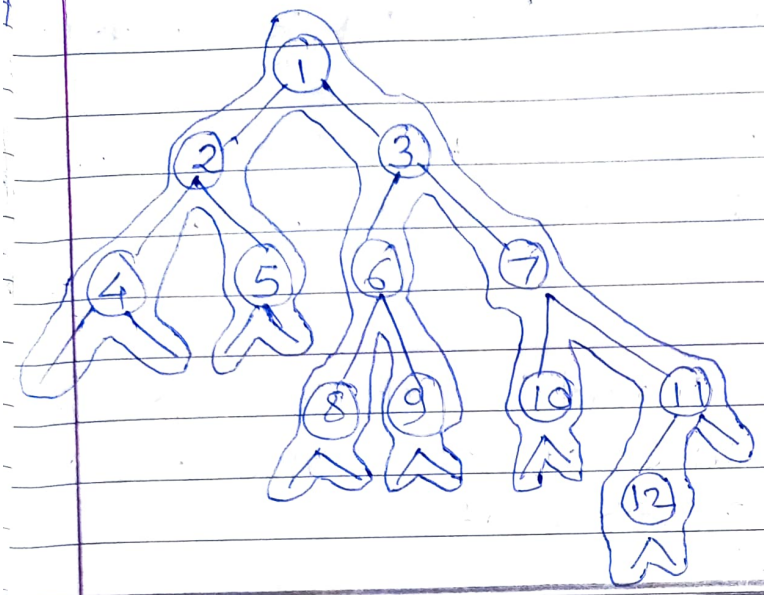
$9 + 1 = 1^{th}$ from 9

0	1	2	3	4	5	6	7	8	9
69			58			49		18	89

Double Hashing Advantage

If Hash function choosen appropriately inser
never fails. if table has at least one em
slot.

Preorder Inorder & Postorder

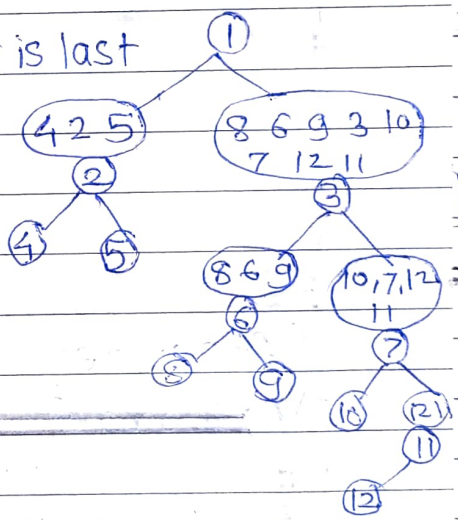


	order	Print at
Preorder	PLR	1st time
Inorder	LPR	2nd time
Postorder	LRP	3rd time

Preorder – 1, 2, 4, 5, 3, 6, 8, 9, 7, 10, 11, 12
Inorder – 4, 2, 5, 1, 8, 6, 9, 3, 10, 7, 12, 11
Postorder – 4, 5, 2, 8, 9, 6, 10, 12, 11, 7, 3, 1

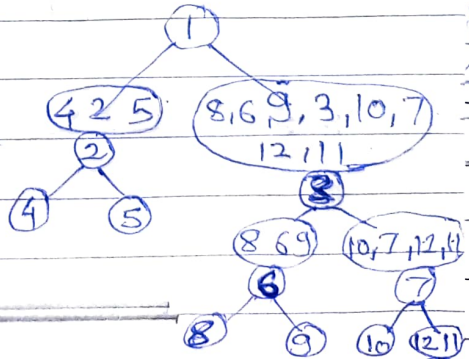
Postorder to preorder

1) From Post order to preorder root is last
2) Spit across root in inorder
3) Most recent to root in left Node in postorder is next root.
4) follow same.



Preorder to postorder

1) From Preorder to postorder root is first node in preorder
2) Spit across root in inorder
3) Most recent to root in preorder is next root
4) Spit across root in inorder.



Note –

1) Inorder used to split across root two parts
2) Pre, Post orders used to get most recent to root and assign as next root.

Expression Trees

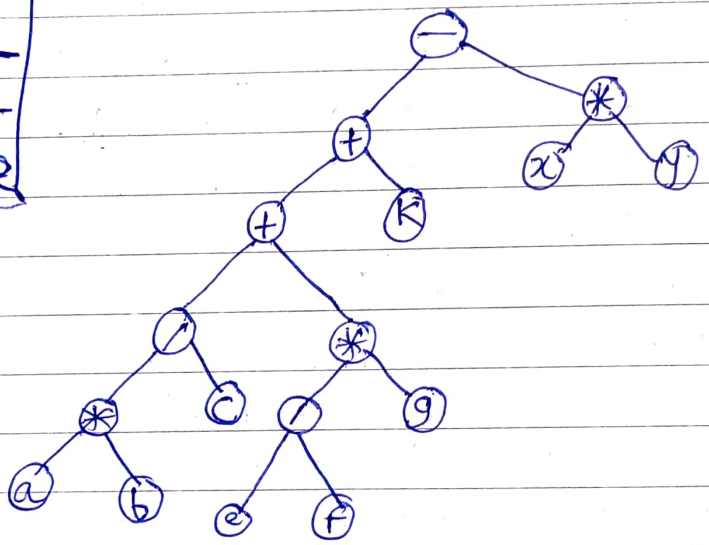
Inorder a * b/c + e/f * g + k − x * y

^	→	R ⇒ L
* /	→	L ⇒ R
− +	→	L ⇒ R

This is the precedance for expression evaluation.

Higher precedance should be at lower side/le and lower precedance at top. hence we cha associativity and reverse it.

1) − +	→	R ⇒ R
2) * /	→	R ⇒ R
3) ^	→	R ⇒ R



1) Preorder of tree gives Prefix
2) Postorder of tree gives Postfix

Prefix and postfix don't requires parenthesis while infix requires

Infix Prefix Postfix

Infix to postfix

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

Symbol	Stack	Post fix.
(	(	A
A	(+	AB
+	(+/	ABC
B		
/		
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
-	(-	ABC/DE+*+
F	(-	ABC/DE+*+F
)		ABC/DE+*+F-

1) When operator come push to stack operand to postfix when bracket completes (-/+/*---) cancel bracket and operators inside pushed to postfix.
2) If operator of less priority comes to after higher priority or with same priority then that initial operate are pushed to postfix one by one from LIFO property
3) priority $\wedge \Rightarrow (\times, /) \Rightarrow (+, -)$

Simplify postfix

~~3524-2A7~~ 6 4 + 3 * 16 4 / - = ?

One by one push each operand to stack if operator comes perform operation with that operator on last two digit.

Ans ⇒ 26

Post fix to infix (same as solving postfix)

abc -+de-fg-h+/*

$\frac{(b-c)}{a} \Rightarrow \frac{f-g}{d-e} \Rightarrow \frac{(f-g+h)}{(d-e)} \Rightarrow \frac{(d-e)/(f-g+h)}{a+(b-c)}$

$\boxed{(a+(b-c)) * (d-e)/(f+g+h)}$

Infix	Prefix	Post fix
(A+B)*(C-D)	(+AB) * (-CD)	(AB+) * (CD-)
	$\boxed{*+AB-CD}$	$\boxed{AB+CD-*}$

Bracket
substract
add

Infix to Prefix

1) reverse infix expression follow all steps of infix to postfix using stack
2) Just don't pop for same precedance of two operators (top and incoming) pop for lower precedance and complete bracket.
3) After conversion again reverse ~~erne~~ obtained expression, this is prefix.

OR

solve as putting sign at front for each sub with proper order.
bracket ⇒ *, / ⇒ +/ and follow scanning left only.

e.g. K+L-M*N+(O^P)* W/U/V *T+Q
K+L-M*N+^OP * W/U/V *T+Q
K+L-*MN+*^OPW/U/V *T+Q
K+L-*MN+/*^OPWUV*T+Q
K+L-*MN+*//*^OPWUVT+Q
+-+KL-**MN**//*^OPWUVTQ
++-+KL**MN**//*^OPWUVTQ

Linked List Questions

1) time to delete node from doubly LL? ⇒ O(1) as time for searching not included.

x node removed by ⇒ x.prev.next = x.next;
x.next.prev = x.prev;

3)
```
current head;
while(current.next != Null)
  if(current.data == current.next.data){
    ——— }
  else{ current = current.next;}
```

To delete duplicate
——— replaced by.

```
p = current.next.next
(current.next) free
current.next = p
```

4) which function is slowest? for linked list
cardinality - It is counting no. of nodes
membership - It is looking if element present in list
Intersection (∩) - common of two lists
union (∪) - combining two list without repeating common

5) If linked list used for Queue and circular then if pointer point to rear we can be do any operation in O(1). But not if it points to first. as to go back we have to traverse whole list. (Assume singly LL).

6) To find 8^{th} element from beggining and end time complexity if list is singly linkedlist?
→ O(1) & O(n). How to travels from last?

7) We can create doubly linked list with only one pointer by storing ———?
→ XOR of addresses of prev. and next nodes.

8) If only pointer to node X is given and not for head can we delete node X?
→ (A) Yes by copying x.next to x and delete x. only if it not a last node

X b) If it is not the first node copy x.next to x and delete x

9) false statement

```
void traverse ( struct Node *head)
 { while (head.next != null){
        printf ("%d", head.data)
        head = head.next;
    }
 }
```

function not incorrect because of head, and program crash because of Null.next is not valid if LL is empty

a) if LL is emp
program cr

b) last node r
printed

c) The function
written with h
hence incorre

10) time for deleting node with pointer to node is for singly linked list? worst case?

→ O(n) copy data of next node to x then dele x i.e. last node.
or
O(1) copy data from next and node.next is node

11) time complexities of operations are
O(N) delete, O(logN) insert, O(logN) find, O(N) decre key. then time complexity is
→ $O(N^2)$

12) concatenation of LL done in O(1) time by
1) singly 2) doubly 3) circular doubly 4) array based
→ cannot get last element in O(1) for other