★ Approaches

1. Treat as stream.
2. Operations on the type (x.methods.sort)
3. Datastructure Size == Runner (Last 'k' Element or window size)
4. Time Runner (At different point in time)
5. Speed Runner (Step Runner)
6. Treat substructure as range [Number based problems]
7. Tree with range labels (B-ST)
8. Hashes map to array when alphabets (Σ) are known
9. Exponential Backoff
10. Generating Configurations (Document/Elaborate)
11. Dynamic Programming
12. Directional Runner (Reverse)
13. Tranform in constant pass(es) & Conquer
14. BUD ⟷ BottleNeck / Unnecessary Work / Duplicate Work
15. DIY → Do It Yourself
16. Base Case & Build
17. Simplify & Generalize
18. Data Structure Brain Storm | Big O Brain Storm
19. BCR → Best Conceivable Runtime
20. State Brute Force
21. Big enough Example

(PT.O)

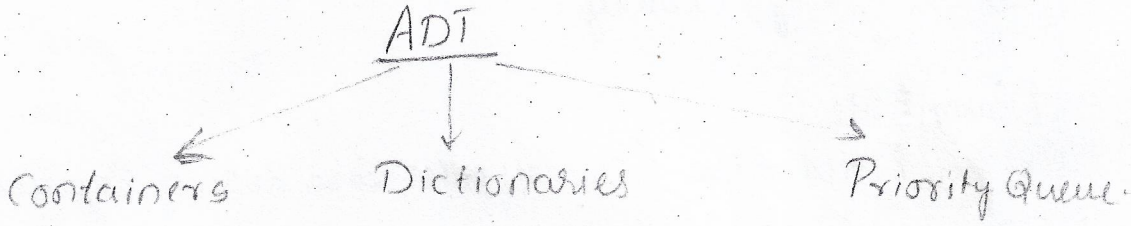
22. Memory Limit

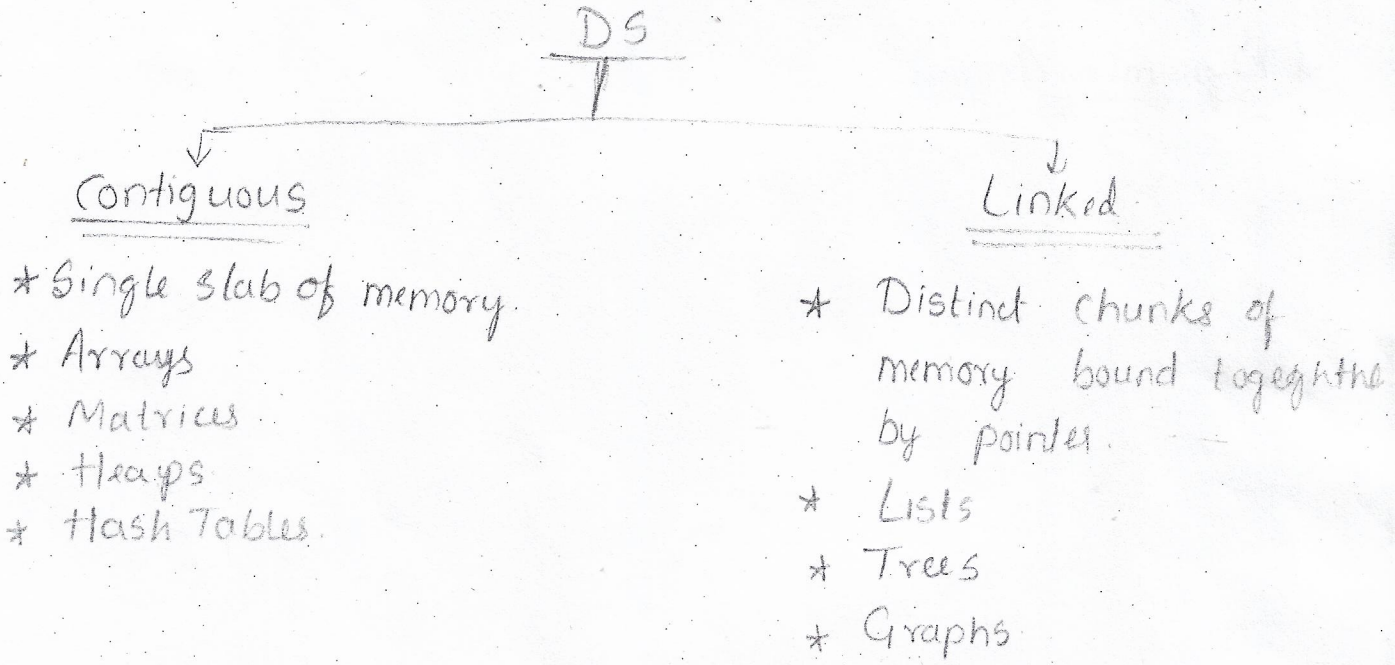
- Chunk
- Stream (Special Case of Chunking)
  → Min(chunk) ⇒ datum ⇒ atomic Unit

23. Stacks for look-ahead problems (Parsing, decisions on prev values based on what's to come)

Data Structure.

NOTE: The maximum benefit from good data structures results from designing your program around them in the first place.





* Arrays.

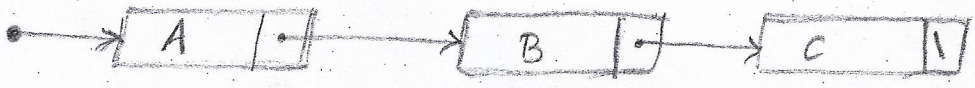
Fundamentally contiguously-allocated data structure.

- Advantages:
  1. Constant time access.
  2. Space Efficiency.
  3. Memory locality.
- Downside:
  - Fixed Size (Cannot alter Size during Execution).

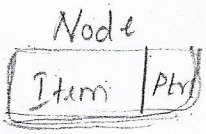
* Dynamic Arrays.

* Pointers & Linked data Structure.

Example:



(Link List)



Operations on list :-

	unsorted	sorted
* Search	O(n)	O(n)
* Insert	O(1)	O(n)
* Delete	O(n)	O(n)

* Stacks & Queues

These are the Containers, which Stores the informati
Distinguished by the Particular ~~retrival~~ retrieval order they suppo

STACK : LIFO

Operations :
* Push
* POP

QUEUE : FIFO

Operations :
* Enqueue
* Dequeue

Implementations :- Arrays
~~Linklist~~ Linked List

★ Dictionaries

The dictionary data type permits access to data items by content. You stick an item into dictionary, so you can find it when needed.

Operations :-

* Search
* Insert
* Delete
* Max / Min
* Predecessor / Successor

Example :- Remove duplicate names from the mail list

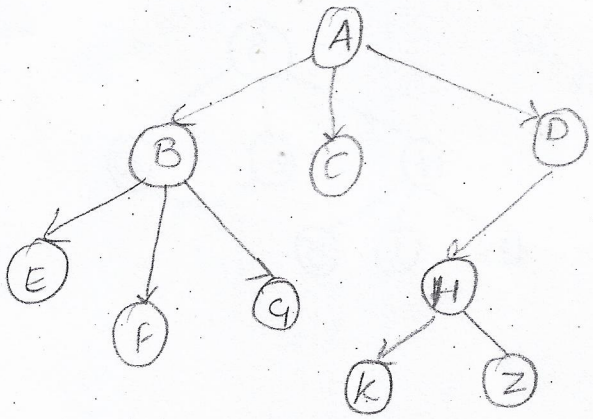
* Complexity

* Trees.

A Tree is collection of nodes connected by directed (or undirected) Edges.

* Arrays, Link lists, Stack & Queues are linear D.S.
* Trees are non linear DS.

* A tree can be Empty with no nodes or a tree is a structure consisting of one node called the 'root' & zero or one or more Sub tree.

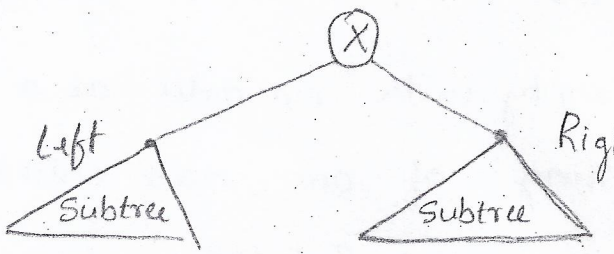


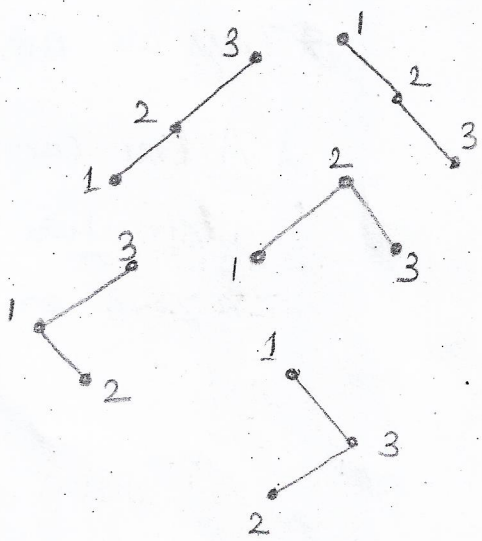
* Root Node
* Internal Nodes
* Leaf nodes
* Height of tree
* Depth of tree/Node

★ Binary Tree

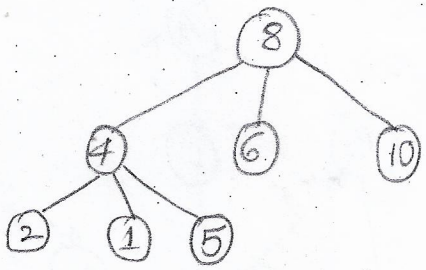
A binary tree is a tree in which Each node has upto two children.

★ Not all trees are binary trees.





★ Ternary tree :-



★ Binary Search Tree

A binary search tree is a binary tree in which Every node fits a specific ordering property.

all left descendants <= n < all right descendants.

Above must be true for Each node n.

* Basic idea behind binary Search tree.

Binary Search requires that we have fast access to two Elements - Specifically the median elements above & below the given node. To Combine these ideas, we need a "linked list" with two pointers per node.

* Searching in Tree

B.S.T

* Input : x (Number to be searched).

$x <=$ Middle $\Rightarrow$ Left Subtree.

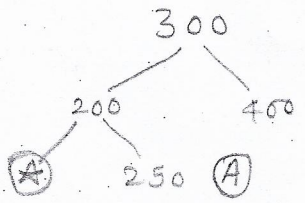
$x >$ Middle $\Rightarrow$ Right Subtree.

* Do it recursively until x is discoverid.

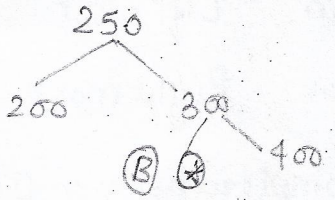
* Complexity $O(h)$, $h \rightarrow$ height of the tree.

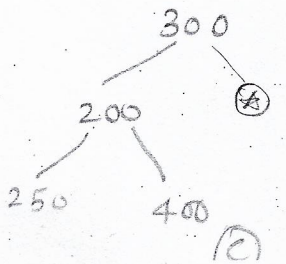
= * Propesty:

* In-Order traversal is not indicative of Structure of tree.



A & B
In-Order traversal
200, 250, 300, 400





A & C

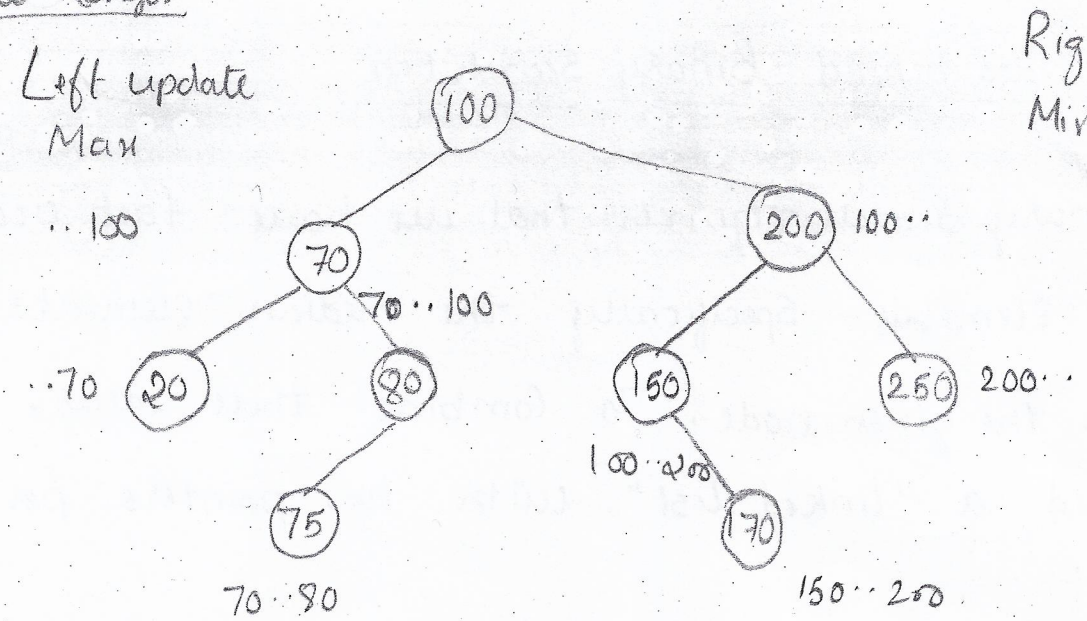
300, 200, 250, 400.

* Pre-Order & Post-Order traversal does not indicate unless "null" is Padded

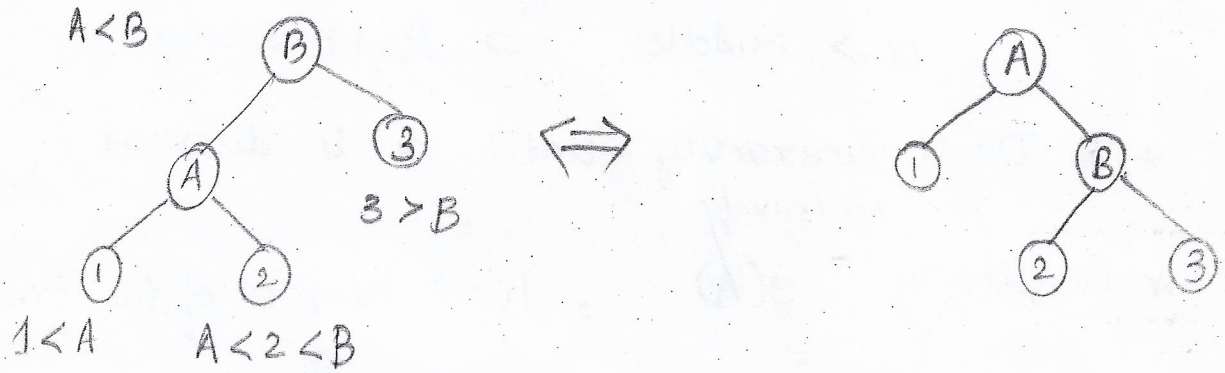
(*) Padding

(2)

*Tree Ranges



* Rotation



* Min & Max.

* Min :- Leftmost (20)
* Max :- Rightmost (250)
* Complexity :- O(h)

✱ Internal node

✱ Predecessor

✱ Maximum value in its left subtree.

✱ Successor

✱ Minimum value in its right subtree.

✱ Leaf node.

✱ Predecessor

✱ First Ancestor which started the right subtree having the element.

✱ Successor

✱ First Ancestor which started the left subtree having the Element.

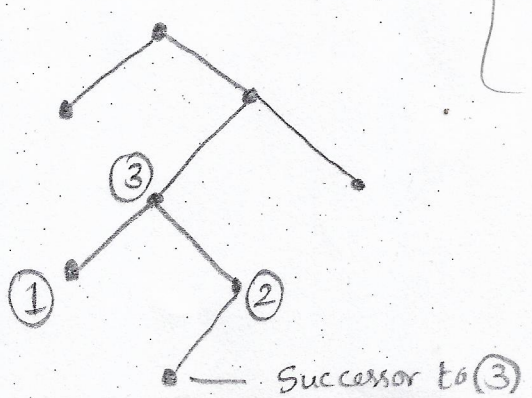
✱ Tree Deletion.

① Leaf :- Delete.

② One child : Swap & Delete or Point parent pointer to c

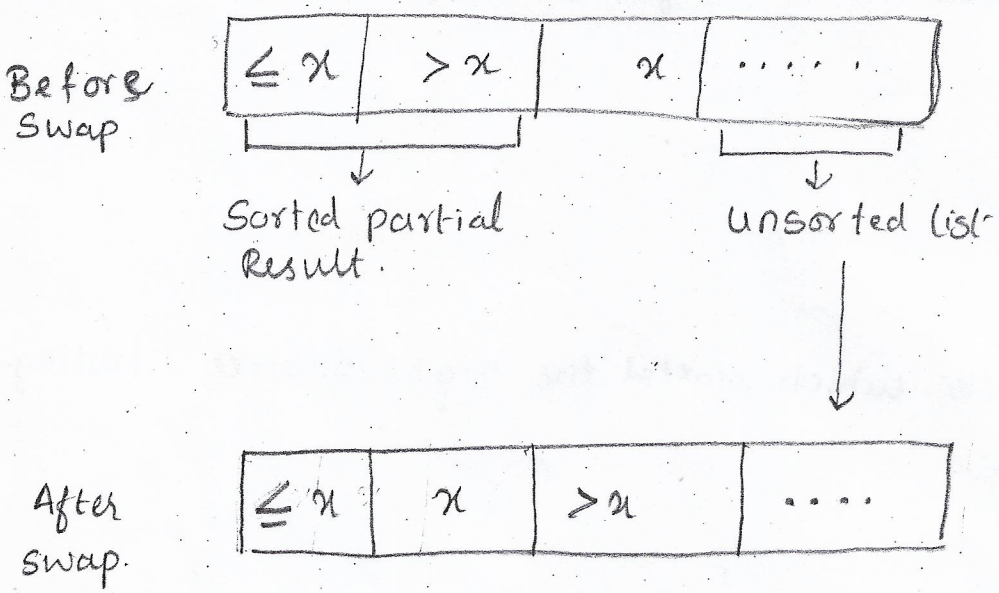
③ Internal node with more than 1 child :- Relabel node as its successor & Delete.

{ Succ will be leaf for an internal node }



★ Insertion Sort:

* Starts with a Single Element & incrementally adds Element
* Swap & Recheck elements to the left.
* The left half of the list stays sorted at Every point.



* Complexity: $O(n^2)$

* Bubble Sort.

* Compare adjacent Elements.
* Exchange if out of order.
* Largest Element "bubbles" to the End at Each iteration.
* Complexity: $O(n^2)$

* Selection Sort.

* Find the Smallest Element
* put the Element in the right place.
* Repeat till the unsorted list is Empty.
* Complexity :- $O(n^2)$

★ Counting Sort

* Keep "counts" of Elements (~~Frequency~~ Frequency Distribution).
* Works on integer
* Output Buffer Size is Equal to the max value in I/p.
* Value Encountered is treated as input index into buffer.
  Eg: acc[3] = Count of ~~occurrences~~ occurrences.
* Walk down the collection, Storing & updating ~~frequency~~ frequency.
* Print "acc" in order to get Sorted output.
* Complexity :. O(N)

★ Merge Sort

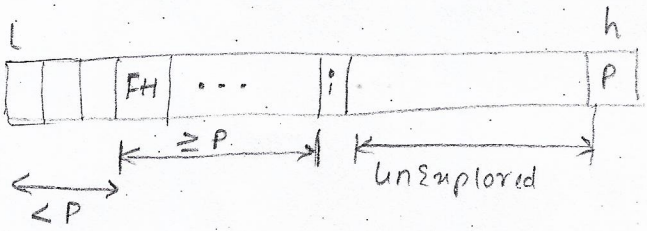
★ Divide & Conquer method

★ Quick Sort

* Select pivot in an array.
* Partition an array based on pivot
* left Array < Pivot < Right Array
* Do it ~~recursively~~ recursively till array is sorted.

★ Array Partition

* Divide collection into 3 parts
  - Unexplored (All Elements which are yet to be compared with 'Pivot' Elem
  - Pointer to the first Element greater than pivot (so far) [FH]
  - Explored (compared Elements with pivot)



< P → Left of FH

≥ P → Between FH & i

Unexplored → Right of i

★ S(i) < S(P)

swap S(i) & FH

FH++

* Heap Sort

* Selection Sort with 'min-heap'
* Extract 'min' 'n' times [$\log n \cdot n$]
* Complexity : $O(n \cdot \log n)$

* Bucket Sort:

* Divides Elements into 'Buckets'.
* Each 'Bucket' is sorted using Either a Separate sort algorithm OR by applying 'Bucket Sort' ~~recursively~~ recursively
* Works well when the input is uniformly distributed over the 'Buckets'
* Bucketing function varies & hence buckets.
  - Could be "Alphabets"
  - Ranges, etc.

* Radix Sort.

* Heaps

* A Heap labelled tree is a binary Search tree where key-labelling of a node dominates the key labelling of its children.
* Min-heap → Parent 'dominates' children by Containing Smaller Element. (→ smaller)
* Max-heap → Opposite of min-heap.
* Stored as an array of keys; Position of keys implicitly used like pointers.
* Parent is at 'k' & children at 2k & 2k+1 (k → index)
* ~~Spare~~ Sparse trees waste space (holes), so heap introduces a Constraint on the Shape → Only last level might be incomplete [Right-most spots do not affect ~~length~~ length of array much]
* Saves ~~memory~~ memory (no pointers), but ~~arbitrary~~ arbitrary trees waste sp
* Maintains partial order on a set, instead of Complete order.
* Efficiently supports Priority Queue operation [Insert & Extract min]

* Hashing

* Hashing function converts input string 'S' into an integer.

* 'S' → String, it is made up of alphabets of size α.

* 'α' → Alphabets treated as integers or digits in base-α system.

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times \text{char}(S_i)$$

* There are 2 techniques.
  - Chaining.
  - Open Addressing.

* Chaining → Array of linked list
  * For array of size 'm', there are n/m elements, if uniformly distributed.

* Open Addressing.
  → Maintained as an array.
  → Collisions are resolved by 'Sequential Probing'
  → Sequential probing looks for the next free spot in the array.

* Tries.

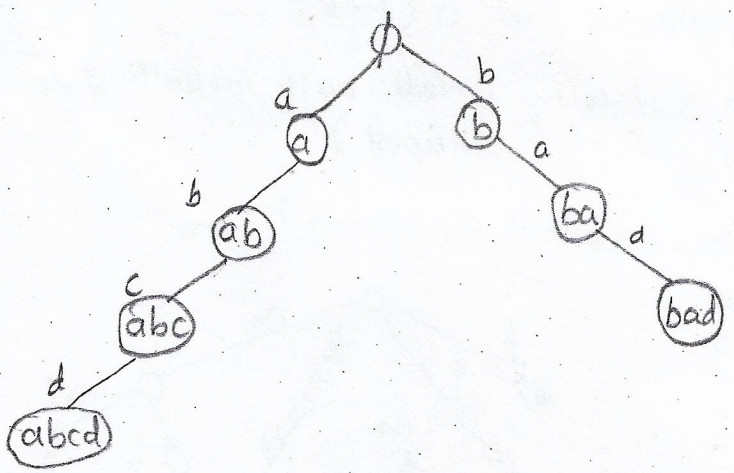
* Tree of prefixes $n = |s|$ $s \in \Sigma$
* Example: $\Sigma = \{a - z\}$ $s = abcd$ $|s| = 4$

  a, ab, abc, abcd.

  → 26 way branching.

  $|bad| = 3$

* Edges store 'alphabet'
* Special terminating char '$'

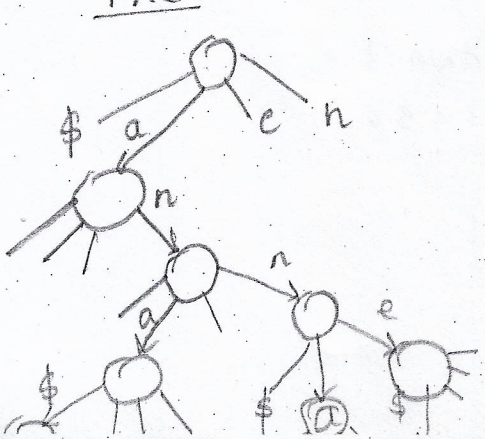


* Compressed Trie.

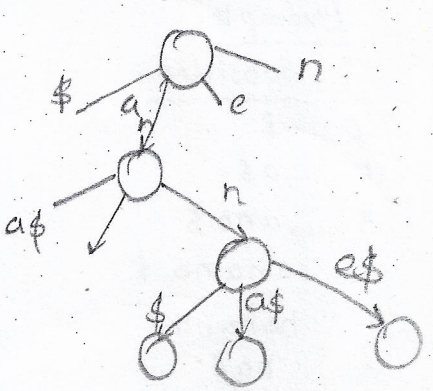
(factorization)

* Contract non-branching path into single Edge →
* Example: {ana, ann, anna, anne}.

Trie



Compressed Trie



* Suffix Trees

* Lexicographically sorted set of suffixes

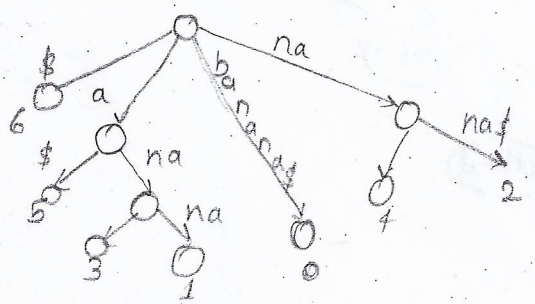
* Example → S = abakan

{ abakan, bakan, akan, kan, an, n } ← unsorted
0 1 2 3 4 5

* Suffix arrays can be built from suffix tree using DFS in O(N)

* Application:
  - Find substring O(P) P → pattern S → string
  - Search for P gives subtree, whose leaves corresponds to all ~~occurences~~ occurrences of P in T
  - First 'K' occurences in O(P+K) — ? Eg
  - Longest repeated substring (Node with atleast 2 descendant leaves & deepest)
  - How long is the repeated substring?

BANANA
0 1 2 3 4 5



* Suffix Arrays

* Conceptual since suffixes are Quadratic in |S|
* Write down all suffixes, Sort them lexically.
* Store the indices.

Example

i	Substring
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

BANANA $
0 1 2 3 4 5 6

* Backtracking & Exhaustive Search.

* Systematically iterate through all the possible Configuration of Search space.
* Configurations could represent subset, Permutation etc.
* At Each Step, Extend a partial solution by adding an Element at the End.
* Check if the partial solutim became an ~~acceptable~~ acceptable solution

  Yes → store.

  NO → Check if the new partial solutim can becom a Complete solution. (Prune if NO)

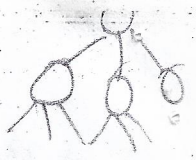
* Backtracking constructs a tree of partial solutions, where Each Vertex represents a partial solution.

* ~~Pruning~~ Pruning → Technique of cutting off the search, the instant we have established that a partial solutim cannot be Extended to a full solution.

* Fundamental Components of Exhaustive search.

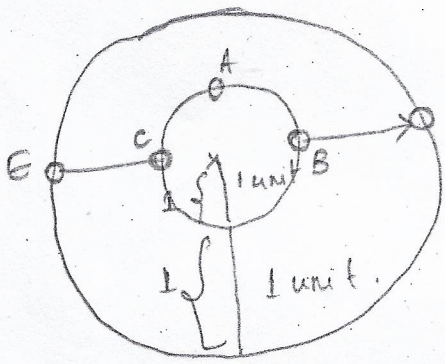
  1. Alphabets
  2. Filters (Pruning)
  3. Termination Condition.

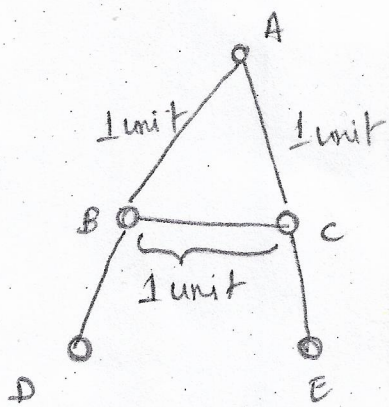
✱ Tree Representation

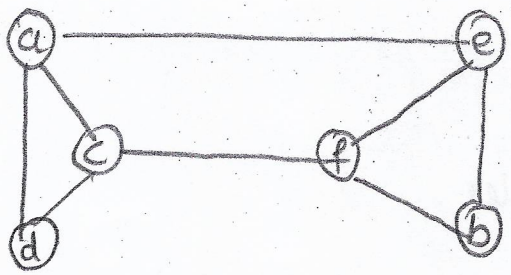


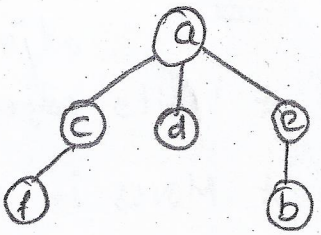
BFS

* Visits ~~adjcent~~ adjacent nodes.
* Moves in Concentric circles.
* Pending/Discovered nodes are pushed into queue.
* Visited Edges are dequeued till queue is empty.
* Works on unweighted graph. (Works on weighted to disc shortest path)
*
* BFS splits the graph into:
  - Tree Edges
  - Cross Edges (Same level)
* Used to find Connected components in graphs
* Vertex Coloring. (~~Bipartied~~ Bipartite?)





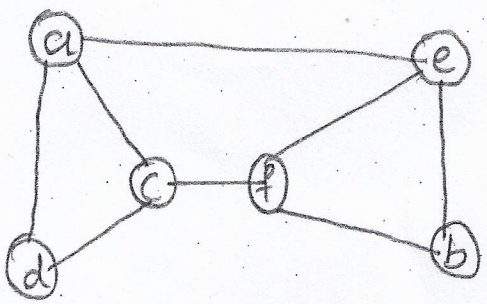




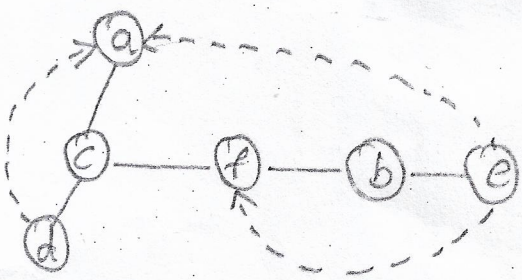
1	2	3	4	5	6
a	c	d	e	f	b

* DFS

* Starts at Vertex 's'
* Visit the ~~adjcent~~ adjacent Vertex (unvisited)
* Repeat the same with the ~~adjcent~~ adjacent Vertex as Starting Vertex.
* Uses 'Stack' DS to push discovered nodes.
* Pop edges till stack becomes Empty.
* DFS partitions into
  - Tree Edge
  - Back Edge (Path back to ancestors)



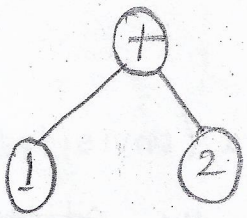
a, c, d, f, b, e



* Used to find cycles/articulation vertex.
* Topological Sort (DAG)
  - Label the vertices in the reverse order (~~processed~~ processed)

★ Traversals.

Refer Photos



In-Order	1 + 2	Ruby.
PreOrder	+ 1 2	Lisp
Postorder	1 2 +	Forth