// DS + Algorithms = Programs

Algorithm:
1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

Good:
1. Correctness
2. Efficiency
3. Ease of implement

Data Type = type + operation
- primitive (value) : int, float, ...
- complex (reference) : list, stack, ...

Loop invariants:
- initializations
- maintenance
- termination

// $f(n) = O(g(n))$: $\exists c: f(n) \le c.g(n)$ $\forall n \ge n_0$

$f(n) = \Omega(g(n))$: $\exists c: f(n) \ge c.g(n)$ $\forall n \ge n_0$

$f(n) = \Theta(g(n))$: $\exists c_1, c_2$: $c_1 g(n) \le f(n) \le c_2 g(n)$ $\forall n \ge n_0$

// Master Theorem: $a \ge 1, b > 1, T(n) = aT(n/b) + f(n)$

1. $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$
   $\rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$, $\varepsilon > 0$
   $\rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$
   $a f(n/b) \le c f(n)$; $c < 1$, $n \ge n_0$
   $\rightarrow T(n) = \Theta(f(n))$

// Brute-force, Divide-and-Conquer, Dynamic Programing (sub-problems overlap)
Greedy

// Sorting: stable vs. unstable

	Alg	Worst-case	Aver/Expected
Comparison $\Omega(n\log n)$	Insertion	$\Theta(n^2)$	$\Theta(n^2)$
	Merge	$\Theta(n\log n)$	$\Theta(n\log n)$
	Heapsort	$\Theta(n\log n)$	—
	Quicksort	$\Theta(n^2)$	$\Theta(n\log n)$ (Ex)
Int	Counting	$\Theta(k+n)$	$\Theta(k+n)$
	Radix	$\Theta(d(n+k))$	$\Theta(d(n+k))$
	Bucket	$\Theta(n^2)$	$\Theta(n)$ (avr)

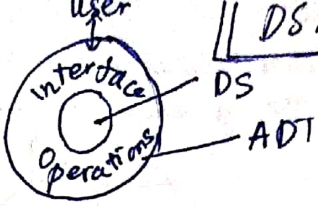
Quicksort: pivot $\rightarrow$ L, G

// List
- Dynamic Arrays-based
- Singly/Doubly Linked List

Stack : LIFO

Queue : FIFO

Circular arrays:
- front = (front + 1) % len
- rear = (rear + 1) % len



// DSA

// Map or Dictionary: searchable dynamic set of key-value entries

Hash table:
- Array 0..N (Table)
- Hash func. h (key $\rightarrow$ index)

Ideal hash func: repeatable, avalanche

$h(x) = h_2(h_1(x))$

Compression func: int $\rightarrow [0, N-1]$

Hash code: key $\rightarrow$ int

Hash code:
- Memory addr (not repeatable)
- Int cast
- Component sum (x permutation)

Polynomial accumulation: $a_0 a_1 \ldots a_{n-1}$

$p(z) = a_0 + a_1 z + \ldots + a_{n-1} z^{n-1}$

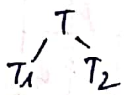
Collision handling:
- Separate chaining: size m << no items n
- open addressing: m >> n
  - Linear probing
  - Double Hashing

// Binary Tree
- Linked
- Array
  - Root $f(p) = 0$
  - p is left child of q: $f(p) = 2f(q) + 1$
  - p .. right ... q: $f(p) = 2f(q) + 2$

Traversal:
- Preorder: node $\rightarrow$ left $\rightarrow$ right
- Postorder: left $\rightarrow$ right $\rightarrow$ node
- Inorder: left $\rightarrow$ node $\rightarrow$ right

// BST: Delete Case 3 - node has 2 children: replace node with its predecessor or successor from inorder traversal of tree, delete that node instead

// AVL Tree: T balanced iff



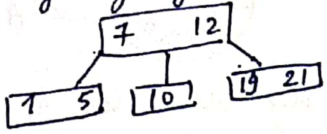
$$\begin{cases} T_1, T_2 \text{ balanced} \\ \text{height}(T_1) - \text{height}(T_2) \le 1 \end{cases}$$

$$h(T) = \begin{cases} 0 \\ 1 + \max(h(T_1) + h(T_2)) \end{cases}$$

Trinode restructing:
- new parent: node with middle key
- left child: smallest key node
- right child: largest key node
- for new parent:
  left subtree goes with new left child
  right subtree goes with new right child

B-Trees

Minimum degree t of B-tree (except root)
no of keys of nodes: $[t-1, 2t-1]$



Complete Binary Tree

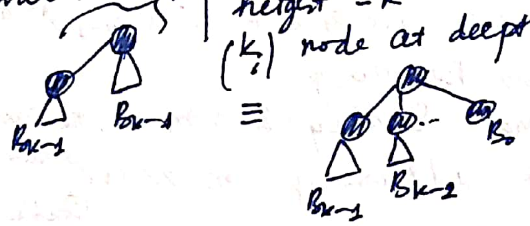
- Filled out on every level, except last one
- All nodes on last level should be as far to the left as possible

Binary Heap $h = \lfloor \log n \rfloor$

Array: $\begin{cases} \text{node } i \text{ has children at } 2i+1, 2i+2 \\ \text{node } i \text{ has parent at } (i-1)/2 \end{cases}$

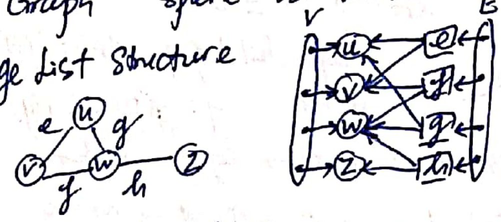
Binomial Heap: set of binomial tree

Binomial tree B_k
- 2^k nodes
- height $= k$
- $\binom{k}{i}$ node at deepth i

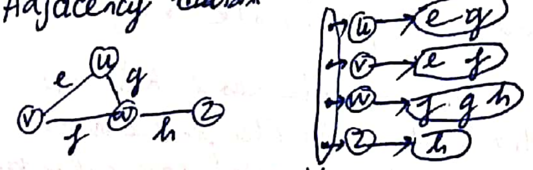


Graph Spare vs. Dense

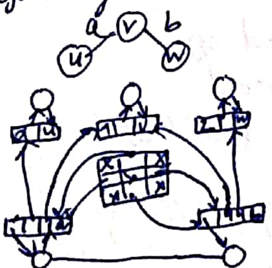
Edge list Structure



Adjacency ~~list~~ list Structure



Adjacency Matrix Structure



$u \to 0$
$v \to 1$
$w \to 2$
$z \to 3$

	0	1	2	3
0		e	g	
1	e		f	
2	g	f		h
3			h	

n vertices, m edges	Edge L	Adj L	Adj matrix
Space	n+m	n+m	n^2
incident Edges(v)	m	deg(v)	n
are Adjacent(v, w)	m	min(deg(v), deg(w))	1
Insert Vertex (o)	1	1	n^2
Insert Edge (v, w, o)	1	1	1
Remove Vertex (v)	m	deg(v)	n^2
Remove Edge(e)	1	1	1

DFS (Stack) vs. BFS (queue)

$O(|V| + |E|)$ shortest path (unweighted

Topological Sorting (Kahn's Algorithm)

DAG (Directed Acyclic Graph):
- step 1. Find vertex has no successors
- step 2. Delete this vertex from graph, insert its label at beginning of the list
- Repeat step 1 & step 2 until all vertices are gone

Minimum Spanning Tree

(only for connected, otherwise Minimum Spanning Forest)

- Unweighted: BFS, DFS
- Weighted: Prim's Algorithm $O(|E| \log |V|)$

1. Start with any city
2. Create an office in that city
3. Measure weight of adjacent edges and insert them in priority queue
   1. From queue, pick cheapest link (Greedy)
   2. Install it
   3. Remove it from queue
   4. Create office in the new city

Shortest path:

- Dijkstra's: $d(z) = \min\{d(z), d(u) + weight(e)\}$
  $O(|E| \log |V|)$ (one-to-all)
- Bellman-Ford: $dist(v) = \min\{dist(v), dist(u) + l(u,v)\}$
  (can handle graphs with negative-weighted graph) $O(|V||E|)$
  (all-to-all)

  repeat $|V|-1$ times:
  for all $e \in E$:
  update (e)

Flow networks

value of max flow = capacity of min cut $C(S, T) = \sum\sum c_{ij}$

- capacities ≡ weights on edges (bandwidth
- Source 's' - no incoming edges
- Sink 't' - no outgoing edges
- Flow: $0 \le flow \le capacity$
  (actual load) flow into = flow out of
  Value: $\sum$ flow into sink

Max flow: flow has maximum value iff it has no augmenting paths

Residual Capacity: $c_f(u,v) = \begin{cases} c(u,v) - f(u,v) \\ f(u,v) \\ 0 \end{cases}$

Residual networks $N_f = (V, E_f)$ can be found by DFS on N_f

$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$