Algorithms

Method of Analysis

Aposkriori (Platform dependent)
- Adv: give exact values
- Disadv: difficult for comparision
  - → Difficult to determine exact times

Apriori (Platform independent)

Algo → (Ram model) → (Time, space)

- → Here we use metric i.e., No of fundamental operations
- Adv: Easy for comparision
- 2 methods: Step count, Order of magnitude.

→ Input class: certain ordering of an i/p

→ Classification of i/p: Best, worst, Average class

Avg case TC: $\sum_{i=1}^{k} t_i * P_i$

k → no of i/p class
t_i → time for i/p class i
P_i → probability.

Asymptotic Notations:

(i) **Big-Oh (O)**: $f(x)$ is $O(g(x))$ iff
$f(x) \le c \cdot g(x)$ for some c, $\forall x > k$

(ii) **Big-Omega (Ω)**: $f(x)$ is $\Omega(g(x))$ iff
$f(x) \ge c \cdot g(x)$ for some c, $\forall x > k$

(iii) **Theta (θ)**: $f(x)$ is $\theta(g(x))$ iff
$c_1 \cdot g(x) \le f(x) \le c_2 \cdot g(x)$,
for some c, $\forall x > k$

(iv) **Small Oh (o)**: $f(x)$ is $o(g(x))$ iff
$f(x) < c \cdot g(x)$ for all c, for all $x > k$

(v) **Small omega (ω)**: $f(x)$ is $\omega(g(x))$ iff
$f(x) > c \cdot g(x)$ $\forall c$, $\forall x > k$

* $f(x) = \theta(g(x)) \Leftrightarrow f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$
* $f(x) = O(g(x)) \Leftrightarrow g(x) = \Omega(f(x))$
* $f(x) = o(g(x)) \Leftrightarrow g(x) = \omega(f(x))$
* $f(x) = O(g(x)) \Rightarrow f(x) = O(g(x))$
* $f(x) = \omega(g(x)) \Rightarrow f(x) = \Omega(g(x))$
* Discrete properties: reflexive, symmetric, transitive
  - reflexive: f
  - symmetric: $+$
  - Eg. $f(n) = O(f(n)$, $f(n) = \theta(g(n))$, $g(n) = \theta(f(n))$

Transpose symmetry: (O, Ω), (o, ω)

* If $f(n) = O(g(n))$ & $d(n) = O(e(n))$ then
  $f(n) + d(n) = O[\max(g(n), e(n))]$
  $f(n) * d(n) = O(g(n) \cdot e(n))$
* Tricotomy property is not satisfied by functions.

Note:

→ $\sum_{i=1}^{n} i \cdot 2^i = 2^{n+1}(n-1) + 2$

→ $n! = O(n^n)$

$n! = \Omega(a^n)$

→ $(\log n)^x = O(n^y)$

→ $n^x = O(a^n)$

$a^{\log_c b} = b^{\log_c a}$

$\log_a b = \frac{1}{\log_b a}$

$\log_a b = \frac{\log_z b}{\log_z a}$

Note:

```
Algo(n)
{
  if(n>0)
    algo(n-1);
}
```

$T(n) = a + T(n-1)$

Time for comparision must be included.

Space Complexity:

* It is space req for computation.
* Space complexity, $S(n) = C + S_p$
  - C: fixed (Instn & fixed mem)
  - S_p: depends on i/p (temporary data, recursion stack)
* $S(n) = O(T(n))$
* If $S(n)$ is $O(1)$ or $O(\log n)$ for rec algo then algo is said to be space efficient.

Divide & Conquer

* TC: $\begin{cases} f(n) & , n \text{ is small} \\ aT(n/b) + g(n) & , n \text{ is large} \end{cases}$

$a \rightarrow$ no of subproblems

$n/b \rightarrow$ size of each subproblem

$a \geq 1 ; \quad b > 1$

Maxi-min: no of comp be T(n)

$$T(n) = \begin{cases} 0, & \text{if } n=1 \\ 1, & n=2 \\ 2T(n/2)+2 & n>2 \end{cases}$$

$\Rightarrow T(n) = \frac{3n}{2} - 2$

$S(n) = O(\log n)$

Merge Sort:

Merge process:

no of comp < best case: $\min(m,n)$; worst case: $m+n-1$

$\therefore TC : O(n+m)$

algo:
```
mid ← (l+h)/2;
MS(a, l, mid)
MS(a, mid+1, h)
merge(a, l, mid, h)
```

$\therefore TC = \begin{cases} c, & n=1 \\ 2T(n/2) + bn, & n>1 \ (b>0) \end{cases}$

$SC = O(n) + O(\log n) = O(n)$

* 2-way / bottom up merge sort.

Binary Search:

$$T(n) = \begin{cases} c, & n=1 \\ T(n/2) + a, & n>1 \end{cases}$$

$T(n) = O(\log n)$

$S(n) = O(\log n)$

Linear Search:

Avg no of comp: $\frac{n+1}{2}$

Matrix Multiplication:

In naive approach & normal D&C approach

TC is $O(n^3)$

In normal D&C

$TC : 8T(n/2) + bn^2 \quad , n>2$ (addition)

SC: ~~O(n^2)~~ $O(\log n)$

Strassens Matrix Multiplication:

It reduces no of multiplications from 8 to 7.

$\therefore TC = O(n^{\log_2 7}) \simeq O(n^{2.81})$

SC: $O(n^2)$

Quick Sort:

Partitioning:

→ choose leftmost as pivot (index l)

→ i ← l+1 ; j ← h+1;

→ loop
```
{
  inc i until A[i] ≥ pivot
  dec j until A[j] ≤ pivot
  if (i<j) swap(A[i], A[j]);
  else break;
}
```

→ A[l] = A[j] ; A[j] = pivot

A[h+1] is set to ∞ and this useful when the pivot is the largest element. (This helps avoid inf loop)

→ no of comp in partitioning: n+1

$\therefore TC : O(n)$

QS Algo:
```
if(l<h)
{
  m ← partition(a, l, h);
  QS(a, l, m-1);
  QS(a, m+1, h);
}
```

TC: best case: $2T(n/2) + O(n) = O(n \log n)$

worst case (sorted): $T(n-1) + n = O(n^2)$

avg case: $O(n \log n)$

SC: best case: $O(\log n)$

worst case: $O(n)$

Long Integer Multiplication:

conventional approach: $O(n^2)$

D&C approach: u, v be n digit numbers

$m \leftarrow n/2$;

$u = w \times 10^m + x$; $w = u/10^m$, $x = u \% 10^m$

$v = y * 10^m + z$; $y = v/10^m$, $z = v \% 10^m$

$uv = (w \times 10^m + x)(y * 10^m + z)$

$T(n) = 4T(n/2) + bn$ (addition)

$\Rightarrow T(n) = O(n^2)$

Anatolli karatsuba's optimization:

Let $t = (w+x)(y+z)$

$= wy + wz + xy + xz$

$\Rightarrow wz + xy = t - wy - xz$

now compute $P_1 = wy$; $P_2 = xz$; $P_3 = (w+x)(y+z)$

$uv = (P_1 \times 10^{2m}) + (P_3 - (P_1+P_2))10^m + P_2$

$\Rightarrow T(n) = 3T(n/2) + O(n)$

$T(n) = a(n^{\log_2 3}) = O(n^{1.58})$

multi-way split for LIM:

D & C: $k^2 T(n/k) + O(n) = O(n^2)$

Anatolli: $(k^2-1)T(n/k) + O(n) = O(n^{\log_k k^2-1})$

Toom & Cook: $(2k-1)T(n/k) + O(n) = O(n^{\log_k (2k-1)})$

Master Theorem:

$T(n) = aT(n/b) + f(n)$, $n > d$, $a \geq 1, b > 1$

$= c$, $n \leq d$

$f(n)$ is +ve

Case(i): $f(n) = O(n^{\log_b a - \epsilon})$, $\epsilon > 0$

$T(n) = \Theta(n^{\log_b a})$

Case(ii): $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$

$k \geq 0 \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$k = -1 \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$

$k < -1 \Rightarrow T(n) = \Theta(n^{\log_b a})$

Case(iii): $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and

$af(n/b) \leq sf(n)$, for some $s < 1$

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

→ Solve $T(n) = 2T(\sqrt{n}) + \log n$ (use transformation)
(refer notes if any doubt).

→ $\frac{n}{\log n} \neq O(n^{1-\epsilon})$

→ Refer recursion tree method in notes.

Problems < decision / optimization

Problem defn; constraints (implicit, explicit); soln space + satisfying explicit constraint.

Feasible soln; objective func; optimal soln.
↳ doesn't exist for decision problems

Greedy Method

fractional / real greedy knapsack:

given P_i & w_i; find x_i such that

$\Sigma x_i P_i$ is maximized subjected to $\Sigma x_i w_i \leq M$

→ think its soln. TC: $O(n \log n)$ → including sort

Job sequencing with Deadlines (JSD):

→ n-jobs; all arrive at 0; deadline d_i; profit P_i

→ soln: arrange profits in desc-
pick each job and ~~sheedule~~ at max possible time

TC: $O(n^2)$

Optimal Merge Pattern:

At any point choose two records with least weight merge them and put them in list: Continue this untill all are merged.

→ If two files have n & m records then no of record movements = n+m
$\therefore O(n+m)$

→ Total record movements
= weighted external path length
= sum of internal nodes

TC: < Sorted / unsorted array / list: $O(n^2)$
Heap: $O(n \log n)$

SC: $O(n)$ (for any implementation)

Huffman Coding: application of OMP

→ non-uniform coding.

→ Build tree & assign prefix code.

→ avg no of bits needed = weighted external path length
= sum of internal nodes.

→ no of bits req ≤ no of bits in uniform encoding

Spanning Trees

→ Any graph algo with adj list has min TC of $O(n+e)$ and with adj. matrix has min TC of $O(n^2)$.

→ max no of spanning tree for n = n^{n-2} complete graph

→ Refer notes for method of calculating no of spanning trees (PYQ: 8)

Prims Algo: (Prims actually starts choosing a single vertex)

→obtain least cost edge $\langle k,l \rangle$

→Compute near values and set near(k) = near(l) = 0

→for i := 1 to n-1

{ get j such that near(j) ≠ 0 and cost(j, near(j)) is minimum.
near(j) ← 0
recompute near
}

TC: adj. matrix : $O(n^2)$ {SC: $O(n)$ near}
heap : $O((n+e)\log n)$ {SC: $O(e+n)$ heap, near}

Kruskals Algo:

→ Construct heap out of b edge — $O(e)$
→ i ← 0
while i < n-1 and heap not empty — $O(e)$
{
get min cost edge $\langle u,v \rangle$ — $O(\log e)$
j ← find(u); k ← find(v)
if j ≠ k
{
i ← i+1
Add edge to MCST
union(j,k)
} — $O(1)$
}
if i ≠ n-1 then print "no spanning tree"

TC: - using heap - $O(e \log e)$
SC: $O(n)$ - for parent array

Dijkstra's Algo:

Randomly choose an edge and add it. If cycle is formed remove the largest edge of the cycle. Continue this until all the edges are added.

Note:

Cost of MCST generated by prims & kruskal is same. But trees generated by both algorithms may differ when there are multiple edges of same weight.

→ If all the edges cost are distinct then trees produce will also be same.

Single Source Shortest Path: Dijkstra's Algorithm

→ Matrix method & spanning tree method
↳ shows path also

→ Choose vertex closest so far and recompute distances. Do this process n-1 time.

→ Implementation is similar to prims

∴ TC: adj. matrix : $O(n^2)$ SC: $O(n)$
heap : $O[(n+e)\log n]$ SC: $O(n+e)$

Note:

→ Min cost edge of a cycle may or may not be in the MCST.
→ Max cost edge of a cycle never exists in MCST.
→ If max cost edge is present in MCST iff it is a bridge.

Problem	TC	SC
Max-Min	$O(n)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n)+O(\log n)$
Merge process	$O(n)$	-
Binary search	$O(\log n)$	-
Quick sort	$O(n \log n)$ Avg/best $O(n^2)$	$O(\log n)$ Avg/best $O(n)$
Strassen's	$O(n^{2.81})$	$O(n^2)$
real knapsack	$O(n \log n)$ (including sort)	$O(1)$
JSD	$O(n^2)$	-
OMP	$O(n^2)$ (list) $O(n \log n)$ (Heap)	$O(n)$
Huffman	$O(n^2)$ (list) $O(n \log n)$ (Heap)	$O(n)$

Problem	TC	SC
Prim's	$O(n^2)$ $O((n+e)\log n)$	$O(n)$ $O(n+e)$
Kruskal	$O(e\log e)$ $=O(e\log n)$	$O(n+e)$
Dijkstra's	$O(n^2)$ $O[(n+e)\log n]$	$O(n)$ $O(n+e)$

Dynamic Programming

DP approach

- Top Down (Memoization) (Recursive)
- Bottom Up (Non-recurse) — More efficient

Multistage graph:

$COST(i,j) = \min\{COST(i+1,x) + c(j,x)\}$

stage, vertex

$COST(l-1,x) = c(x,t)$

$l \rightarrow$ no of stages

$D(i,j) = x$ (used for path computation)

TC: $O(n^2)$, $O(n+e)$

↓ adj. matrix ↓ adj list

SC: $O(n)$

↳ dist values

Travelling Salesman Problem:

$g(i,s) = \min\{c(i,x) + g(x, s-\{x\})\}$

$g(i,\phi) = c(i,v_0)$

↳ home city

$J(i,s) = x$

TC: $O(n^2 \cdot 2^n)$. SC: $O(n \cdot 2^n)$

Floyd Warshall's Algorithm (All pair shortest)

$A^k(i,j)$ represent cost of shortest path from i to j with intermediate vertex not greater than k.

$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$

$A^0(i,j) = C(i,j)$

TC: $O(n^3)$ S: $O(n^2)$

→ This algo can be used to find transitive closure & reflexive transitive closure in $O(n^3)$

→ Floyd warshall's works with -ve weighted edge graph too.

Bellman Ford (Single Source Shortest Path):

$d^l[x]$ represent cost of path from source s to vertex x with atmost l edges.

$d^l[x] = \min\{d^{l-1}[x], \min_{\forall k \in V}\{d^{l-1}[k] + C[k,x]\}\}$

$d^1[x] = C[s,x]$

TC: $O(n^3)$: adj. ~~dist~~ matrix

$O(ne)$: adj list

Algo	TC	SC
Multistage graph	$O(n^2)$ $O(n+e)$	$O(n)$
TSP	$O(n^2 \cdot 2^n)$	$O(n \cdot 2^n)$
Floyd warshall's	$O(n^3)$	$O(n^2)$
Bellman ford	$O(n^3)$ $O(n \cdot e)$	
0/1 knapsack	$O(n \cdot m)$ $O(n \cdot 2^n)$ brute	$O(n \cdot m)$
LCS	$O(nm)$ $O(n \cdot 2^m)$ $O(2^n \cdot m)$ } brute	$O(nm)$
Matrix chain prod	$O(n^3)$	$O(n^2)$
OBST	$O(n^3)$	
Realiable system		

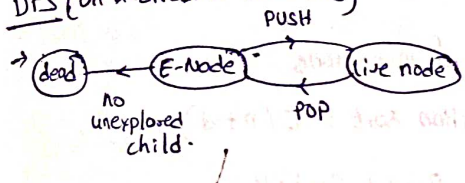
Graph Techniques

→ Tree traversal is unique, graph traversal need not be unique.

Status of node:

- Enode: node that is currently under exploration
- live node: nodes that are not fully explored (stack in DFS, Q in BFS store these)
- dead node: fully explored node.

DFS (on undirected connected)



Discovery time: time at which node is 1st visited

finishing time: time at which node becomes dead.

→ Before any node is explored, it must be popped.

→ A graph is connected

⇔ finishing time of 1st node is highest

⇔ traversal finishes when 1st node becomes dead.

DFS on undirected disconnected graph:

→ Here depth first search spanning forest is formed.

→ no of connected components = no of spanning trees.

DFS on directed graph:

DFS on a directed graph produces 4 types of edges.

(i) Tree edge: part of depth first tree

(ii) Forward edge: parent to non-child descendant

(iii) Back edge: node to non-parent ancestor. Self loop are back edges.

(iv) Cross edge: node to neither descendant nor ancester. Cross edge may even be btw vertices of different DFTs

Paranthesis theorem:

Let (u,v) be a directed edge.

d(u) < d(v) f(v) > f(u) ⇒ tree edge / frwd edge

d(v) < d(u) f(u) > f(v) ⇒ back edge

[d(v) < f(v)] < [d(u) < f(u)] ⇒ cross edge

DFS on DAG:

→ Source vertex: no incoming edges

Sink vertex: no outgoing edges.

~~Reverse of~~

→ Descending order of finishing times gives topological sort.

Breadth First Search:

FIFO BFS:

→ 1st node never becomes live node i.e., 1st node never gets on the Q.

→ All lives nodes are stored in the Q. (trace parent so that drawing BFT will be easier)

→ A node is marked visited as soon as it is pushed into Q.

→ BFS can be used to find shortest cycle containing given vertex.

LIFO BFS (or) Dsearch:

→ Here live nodes in the Q are made E-nodes in LIFO order.

→ can be implemented using stack.

Priority Queue:

→ pushing into Q is done normally.

→ But removing is done based on some criteria.

TC of BFS, DFS < $O(n^2)$: adj matrix; $O(n+e)$: adj list.

Note:

→ DFS and BFS can be used to determine presence of cycle (using backedge)

→ DFS and BFS can be used to determine connectivity of graph.

→ BFS is optimal algo for find shortest path in an undirected unweighted graph.

→ DFS is used to find components & articulation pts.

connected comp — strongly connected comp — Biconnected comp.

Connected Components

→ Maximal subgraph that is connected is called connected component (applies only for undirected graphs)

Strongly Connected Components (only for directed graph)

→ 2 vertices are strongly connected ⇔ there exists directed path from u to v & v to u.

Strongly connected comp: Maximal subgraph in which there is directed path ~~between~~ from any vertex to any vertex

Metagraph: subgraph with all the vertices

Note:

→ Every directed graph is DAG of it ~~com~~ strongly connected components.

→ C_1, C_2 be two strongly connected components such that there is an edge from vertex in C_1 to vertex in C_2. then finishing time of C_1 will be greater than finishing time of any vertex in C_2.

If there edge from C_1 to C_2 then there will be no edge from C_2 to C_1.

Articulation Point & Biconnected Components

→ Graph without AP is called biconnected.

→ Biconnected component: Maximal subgraph that is biconnected

---

SORTING TECHNIQUES

→ stable vs unstable.

→ internal vs external

→ inplace vs not inplace

→ TC of comparision based sort $= O(\max\{\text{comp}, \text{swaps}\})$

Bubble sort:

no of comp $= (n-1)+(n-2)+\cdots+(1) = \frac{n(n-1)}{2} = O(n^2)$

no of swaps = no of inversions

Selection Sort:

In each pass i, select i^{th} smallest and place it in the correct position.

→ no of comp $= \frac{n(n-1)}{2}$

→ no of swap $= n-1$

Insertion sort:

→ Take an element (starting one) from unsorted list and place in its correct position in ~~unsorted~~ sorted list.

i.e., In each pass i, list will be sorted till index i.

→ Insertion acts like external sorting but it is not.

TC: Best case: $O(n)$
worst case: $O(n^2)$

→ If d is no of inversions,

TC of insertion sort $= O(n+d)$

Non-Comparision Based Sorting

Radix sort:

→ If base is b, take b buckets.

→ From LSB to MSB
for each digit, distribute into buckets an reorder.

TC: $O(d(n+b)) = O(nd)$ ($\because$ b is const.)

d → max no of digits
b → base

→ If x is maximum number, then

$TC = O(n \log_b x)$

if $x = O(n^c)$

$\Rightarrow TC = O(n \log n)$

∴ Radix sort is good when $b \geq n$

$\Rightarrow TC = O(n)$

→ Radix sort need special implementation for sorting negative numbers.

→ sorting fractional numbers is not possible.

Data Structures

Full binary tree: Every node has 0 or 2 children

Complete binary tree: Every lvl is filled except last and last is filled from left to right

Perfect binary tree: All levels are completely filled

k-ary tree: Every node has either 0 or k children.

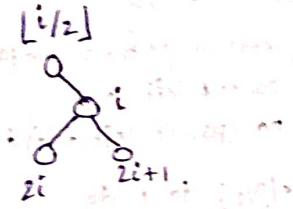
- Total no of nodes = no of internal nodes + no of leaf nodes
  ↓
  $\left(\begin{matrix}\text{no of nodes} \\ \times \\ \text{no of children}\end{matrix}\right) + 1$

- Sum of all heights in perfect binary tree or complete binary tree = $O(n)$.

Array representation of complete binary tree:

(Assuming root node index is 1)

adv: Consumes less space.



- If tree has n nodes, then

  ~~height~~ highest non-leaf index = $\lfloor \frac{n}{2} \rfloor$

  starting leaf node index = $\lfloor \frac{n}{2} \rfloor + 1$

Tree Traversals

- Inorder, Preorder, postorder.
- For a given inorder / preorder / postorder we can construct $\frac{1}{n+1} {}^{2n}C_n$ binary tree.
- no of unlabled binary trees possible = $\frac{1}{n+1} {}^{2n}C_n$
- For a given preorder & inorder atmost 1 tree exists
- For a given postorder & inorder atmost 1 tree exists.
- no of labled binary trees with n nodes = $\frac{n!}{n+1} {}^{2n}C_n$
- For a given preorder & postorder more than 1 tree may exist.
- TC of traversals: $O(n)$ (iterative/recursive)

  $T(n) = T(k) + T(n-k-1) + C$

TC for construction binary tree from preorder & inorder / postorder & inorder $= O(n)$

Binary Search Tree:

→ Inorder on BST is ascending order.

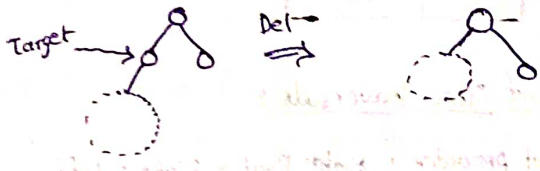
→ BST from preorder/postorder: TC: $\Theta(n)$

Insertion: Best case: $O(\log n)$; worst case: $O(n)$

Deletion:

(i) Deleting leaf: Delete directly.

(ii) Deleting node with one child



(iii) Deleting node with 2 children:

replace node with inorder successor (min of RST) or inorder predeccessor (max of LST) and called delete on inorder successor or predeccessor.

TC: Best: $O(\log n)$; worst: $O(n)$

- no of BSTs possible with n nodes = $\frac{1}{n+1} {}^{2n}C_n$

AVL Tree:

Balanced tree: height is $O(\log n)$

Blancing factor: Height of LST − Height RST

→ In AVL tree the balancing factor is $-1, 0, 1$

→ Min no of nodes in AVL tree of height h is

$$T(h) = 1 + T(h-1) + T(h-2)$$

→ Max no of nodes is possible when it is perfect binary tree.

Insertion: LL - single right rotation RR - single left rot.

Imbalances	Soln
LL	LL rot on I
RR	RR rot on I
LR	RR on L(I) & LL on I
RL	LL on R(I) & RR on I

I - Imbalanced node

Deletion:

R case: Deletion is on right of imbalanced node.
L case: Deletion is on left of imbalanced node.

R_i: balance factor of L(I) is i.
L_i: balance factor of R(I) is i.

Case	Soln
R_{-1}	LR
R_0	LL/LR
R_1	LL
L_{-1}	RR
L_0	RR/RL
L_1	RL

TC:
Insertion → O(logn)
Deletion → O(logn)
Building a complete tree → O(nlogn)

Converse of Tree Traversals:

Converse of preorder: ~~Right~~ Root; Right; left
Converse of Inorder: Right; Root; left
Converse of postorder: Right; Left; Root

Note:

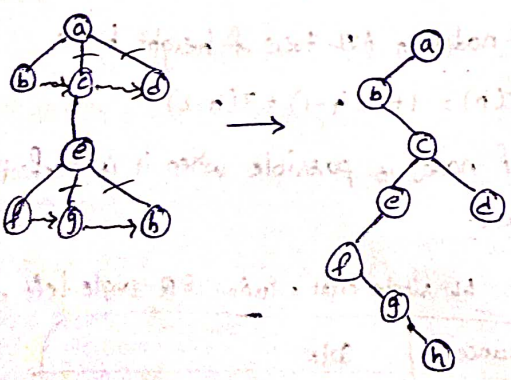
Preorder + Right child pointer + terminal nodes ⇒ Unique binary tree

postorder + left child pointer + terminal nodes ⇒ unique binary tree.

Leftmost child right sibling representation:

Used to represent any tree with more than 2 child.
→ Using this we can remove the need of ~~of~~ advance knowledge of no of children a node has.

Eg:



→ To convert a forest, add a virtual node and remove it in the end.

choose least precedence operator and
Put this as root and remaining parts as LST and RST.
→ Do the same process of LST and RST recursively.
→ while choose least precedence operator, if ~~opera~~ more than one operator has same least precedence ~~and it is~~ then:
if it is left associative, choose rightmost one.
if it is right associative, choose leftmost one.

HEAPS

- Heap is an implementation of priority queue.
- TC for constructing heap tree by inserting nodes in given order = O(nlogn)
- No of min heaps possible with n keys is given by:

$$T(n) = T(k) \cdot T(n-k-1) \, {}^{(n-1)}C_k$$

Build Heap (Heapify)

- At every node we perform 2 comparisions
  i.e., parent & left child
  max(parent, left child) & right child.

```
for i ← ⌊n/2⌋ to 1 do
    heapify(a, i);
```

- TC: O(n)

Deletion:

- Swap(A[1], A[n])
- Apply heapify on A[1] considering there are only (n-1) n

TC: O(logn)

HeapSort:

- Build Heap ... O(n)
- 'n' deletion ... O(nlogn)

Incr key / Decr key: O(logn)

- Find minimum in max heap takes O(n) time.
- Finding maximum in min heap takes O(n) time.

Note: kth smallest element in min heap must be in a level $\leq k$.

Hashing

- load factor (λ) = $\frac{\text{no of keys}}{\text{hash table size}}$.

Collision Resolving

- open addressing / closed hashing
  - linear probing
  - quadratic probing
  - Double Hashing
- Seperate chaining or External Hashing

- To apply closed hashing, $\lambda \leq 1$

Linear Probing :

$h(x) = x \bmod m$

$H(x,i) = (h(x)+i) \bmod m$ (i → probe number)

Searching order: $k, k+1, k+2, \ldots m-1, 1, 2, \ldots k-1$

Disadv: Primary clustering (grp of records stored next to each other)

Quadratic Probing :

$h(x) = x \bmod m$

$H(x,i) = (h(x) + i^2) \bmod m$

Searching order: $k, k+1, k+4, k+9, \ldots$

Disadv: secondary clustering (working only for primes)

no of locations searched is only $\frac{m+1}{2}$

m → hash table size.

So there is a chance that we cannot insert a node even if the hash table is has empty slots.

Note: It is better to have hash table of prime size.

Double Hashing:

$h(x) = x \bmod m$

$H(x,i) = [h(x) + i * h'(x)] \bmod m$

- Primary & secondary clustering are resolved

disadv: more time for computation.

Note: In double hashing, we need to ensure that:

(i) $h'(x)$ is never 0.

(ii) $h'(x)$ is relatively prime to m so that distribution is uniform.

Seperate Chaining (external hashing)

→ Every hash table entry has head point to a linked list.

→ can be used when load factor is > 1

Adv: Deletion is easy compared to closed hashing.

Note: Deletion in closed hashing requires rehashing.

Note: If hash indices are $1, 2, 3, \ldots m$ then we use hash function, $h(x) = (x \bmod m) + 1$

STACK

Stack permutation: generated by inserting key in given order and popping off in any order.

no of stack permutations possible with n keys $= \frac{1}{n+1} {}^{2n}C_n$.

- To find no of function calls in recursion develop a reccurence relation.

```
f(n)
{
    if (n==0 || n==1) return n;
    else return f(n-1) + f(n-2);
}
```

If $T(n)$ is no of func calls in $f(n)$ then

$T(n) = 2\,fib(n+1) - 1$

no of '+' operations $= f(n+1) - 1$

Ackermann's number :

$$A(x,y) = \begin{cases} y+1 & \text{if } x=0 \\ A(x-1,1) & \text{if } y=0 \\ A(x-1, A(x,y-1)) & \text{otherwise} \end{cases}$$

- $A(0,y) = y+1$
- $A(1,y) = y+2$
- $A(2,y) = 2y+3$
- $A(3,y) = 2^{y+3} - 3$

- This is an example of total computable function that is not primitive recursive

Towers of Hanoi :

(i) Move $(n-1)$ disks from source to auxilary

(ii) Move remaining one disk from source to destination.

(iii) Move $(n-1)$ disks from auxilary to destination using source as intermediate needle.

$T(n) = 2T(n-1) + c$

$\Rightarrow TC = O(2^n)$

Infix to Postfix:

(i) operand → print

(ii) opening paranthesis → push

(iii) operator: push if it has higher priority than top of stack (top is opening paranthesis) or top of stack is opening paranthesis. Else, pop and repeat (iii)

(iv) closing paranthesis: pop until opening paranthesis is met.

Infix to Prefix:

Reverse the i/p.

(i) operand → push

(ii) closing parenthesis → push

(iii) operator: push if it has higher priority than top of stack or top of stack is closing parantesis. Else, pop and repeat (iii)

(iv) Opening paranthesis: pop until closing parenthesis is met.

Reverse the o/p.

Note: In above 2 conversion, make sure that a only high priority operator sits on low priority. If priorities are same consider associativity from original string.

Postfix evaluation:

(i) operand → push

(ii) operator → pop and make it right child
pop and make it left child

(iii) push the result of (ii) into stack.

Prefix Evaluation:

(i) Reverse the string.

(ii) operand → push

(iii) operator → pop and make it left child
pop and make it right child

(iii) push result of (ii) into stack.

→ These 2 evaluations constructs expression tree

→ traversals on expression tree gives corresponding expressions.

Stack operations:

→ top is initialized to -1.

push: inc top and push

pop: pop and dec top

Queue:

- initially front = -1 & rear = -1
- Enqueue: insertion is done at rear end:
  - inc rear and insert
  - If Q is empty initialize front to 0.
- if rear ≠ -1 and front ≠ -1 then no of elements in queue = rear - front + 1

deletion: incr front.

disadv: once rear reaches the end we cannot insert even if we have space.

Circular Queue:

Queue full ⇒ front = (rear+1) % MAX

Queue empty ⇒ front = -1 or front = (rear+1) % MAX

ambiguity

- In normal Q / circular Q if front = rear then we have exactly on element we use this stmt to resolve ambiguity.
  i.e., while deleting if front = rear then we set front and rear to -1.
  ∴ front = -1 ⇒ empty
  front = (rear+1) % MAX ⇒ full

ARRAYS

- A[ub...lb] then no of elements in array are lb - ub + 1.
- To access an element at i^{th} index, no of elements crossed = i - lb
- Adresses of an element
  = Lo + (no of elements crossed) * (size of each element)
  ↓
  base address.