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.

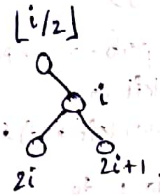
- Total no of nodes = no of internal nodes + no of leaf nodes

$$\left(\begin{matrix}\text{no of nodes} \\ \times \\ \cancel{\text{degree}}\ \text{no of children}\end{matrix}\right) + 1$$

- 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

~~heigh~~ 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.

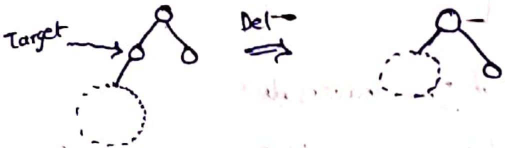
→ BST from preorder / postorder : TC: $\Theta(n)$

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 predecessor (max of LST) and called delete on inorder successor or predecessor.

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 $O(\log n)$

Balancing 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

Scanned with CamScanner

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 $\rightarrow O(\log n)$
- Deletion $\rightarrow O(\log n)$
- Building whole tree $\rightarrow O(n \log n)$

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 $\Rightarrow$ Unique binary tree

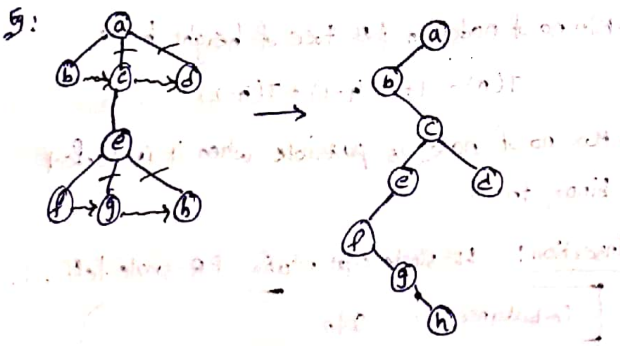
postorder + left child pointer + terminal nodes $\Rightarrow$ unique binary tree.

Leftmost child right sibling representation:

Used to represent any tree with more than 2 child.

→ Using this we can remove the need of ~~at~~ advance knowledge of no of children a node has.

Eg:



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

Expression tree:

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 ~~at~~ ~~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 = $O(n \log n)$
- 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 $i = \lfloor n/2 \rfloor$ to 1 do
  heapify(A, i);

- $\boxed{TC: O(n)}$

Deletion:

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

$\boxed{TC: O(\log n)}$

HeapSort:

- Build Heap ... $O(n)$
- 'n' deletion ... $O(n \log n)$

Incr key / Decr key : $O(\log n)$

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

Note: k^{th} 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 Hasing
  - 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: $k, k+1, k+2, \dots m-1, 1, 2, \dots, k-1$

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: $k, k+1, k+4, k+9, \dots$

Disadv: secondary clustering (working only for primes)

no of locations searched is ~~and~~ 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 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 $1, 2, 3, \dots m$ 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 $= \frac{1}{n+1} \cdot {}^{2n}C_n$

- 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 ~~or~~ than top of stack? or top is opening paranthe 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 ~~and~~ 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 ~~for~~ 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) ~~operand~~ Reverse the string.

(i) operand → push

(ii) 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 ~~for~~ 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 — Ambiguity

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

- In normal Q / circular Q if front = rear then we have exactly one 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)
  Lo → base address.