# Data Structure Notes DEMO

Handwritten Notes

For more PDFs and notes visit my telegram page.. Named "BeingPro33"

@beingpro33

18/08/22 Sorting techniques -

→ Critera for Analysis

i) Number of comparisons
It decides the time complexity.

ii) No. of swaps

iii) Adaptive -
If any sorting method takes less time or min^m time over already ~~mi~~ sorted list then we call that algorithm as adaptive.

iv) Extra Memory

v) Stable -
If a sorting algorithm is preserving the order of duplicate elements in the sorted list then that algorithm is called as stable.

Eg:-

| Name- | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| marks- | 5 | 8 | 6 | 4 | 6 | 7 | 10 |

After sorting, if we get like this then it is called stable -

| Name - | D | A | C | E | F | B | G |
|---|---|---|---|---|---|---|---|
| marks - | 4 | 5 | 6 | 6 | 7 | 8 | 10 |

Here 'C & 'E are preserving their order as previous.

If we get like this then it is not stable -

| Name - | D | A | E | C | F | B | G |
|---|---|---|---|---|---|---|---|
| marks - | 4 | 5 | 6 | 6 | 7 | 8 | 10 |

Here 'C & 'E are not preserving their order as previous

# Sorting Algorithm

- Comparision based sorts
  - $O(n^2)$
    1. Bubble
    2. Insertion
    3. Selection
  - $O(n \log n)$
    4. Heap sort
    5. Merge sort
    6. Quick "
    7. tree "
  - $O(n^{3/2})$ → 8. Shell "
- Index based sorts
  - $O(n)$
    9. Count "
    10. Bucket / Bin sort
    11. Radix sort

They are faster but takes lots of space

## 1 Bubble Sort

A | 8 | 5 | 7 | 3 | 2 |
0 1 2 3 4

$n = 5$

**1st Pass**

| | | | | |
|---|---|---|---|---|
| 8 | 5 | 5 | 5 | 5 |
| 5 | 8 | 7 | 7 | 7 |
| 7 | 7 | 8 | 3 | 3 |
| 3 | 3 | 3 | 8 | 2 |
| 2 | 2 | 2 | 2 | 8 |

Comparision = 4
Swap : 4

**2nd Pass**

| | | | | |
|---|---|---|---|---|
| 5 | 5 | 5 | 5 | 5 |
| 7 | 7 | 3 | 3 | 3 |
| 3 | 3 | 7 | 2 | 2 |
| 2 | 2 | 2 | 7 | 7 |
| 8 | 8 | 8 | 8 | 8 |

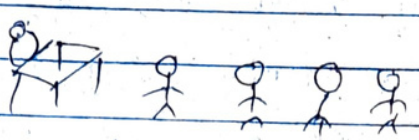
sorted

Comp = 3
Swap = 3

→ When all element compared once then it is called as Pass.

# Queues

It is a logical datastructure and it works on discipline FIFO (First In First Out)



front end (For deletion)

Rear end (For insertion)

* Queue ADT -

Data -

1) Space for storing elements
2) front - for deletion
3) Rear - for insertion

Operations -

1) enqueue(x) - Inserting an element in queue
2) dequeue() - Deleting an element from queue
3) isEmpty()
4) isFull()
5) first()
6) Last()

* Queue can be implimented using two physical data structures that are -

i) Array
ii) Linked list

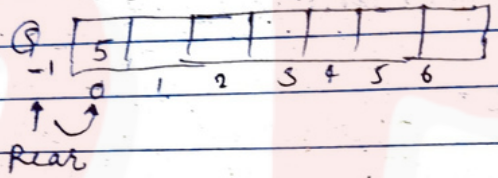
* Implementation of Queue using Array

i) Queue using - single pointer (index pointer)

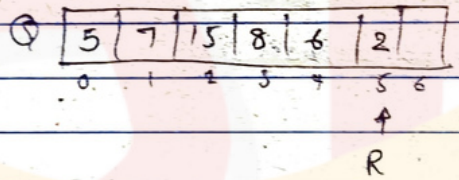
ii) Queue using - front and Rear (Double pointer)

iii) Drawbacks of Queue using Array.

1) Queue using single pointer -

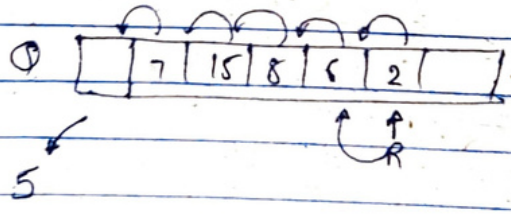



(To insert an element in the queue, just move the rear and insert.)




insert - O(1) → time

(For deleting, we can delete only very first element because it follows FIFO.)




delete - O(n)

13/05/22

# Stack

Stack is a collection of elements which follow discipline LIFO. (Last in first out)

* ADT of stack -

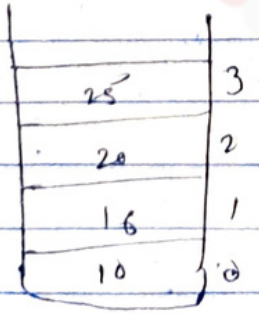
It gives us the definition of stack in terms of data representation and operations

→ Data -

1) space for storing elements
2) Top pointer

→ operations -

1) push(x) → inserting a value
2) POP() → deleting a value
3) Peek(index) → looking at a value at a given position
4) stackTop() → knowing the top most value
5) isEmpty() → knowing the stack is empty or not
6) isFull() → ——————— full or not



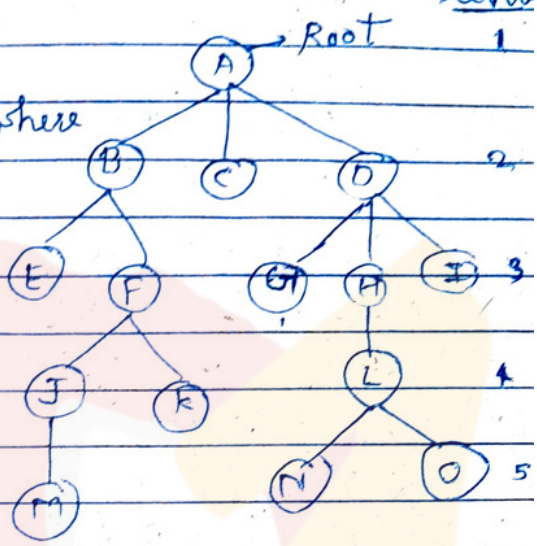
30/05/22

# TREE

Non-linear Data Structure

Tree - It is a collection of nodes or vertices and edges.

"Tree is a collection of nodes where one of the node is taken as root node and the rest of the nodes are divided into disjoint subset and each subset is a tree or subset."




## Terminology

* Root - The very first node on the top is a root.

* Parent - A node is a parent to its very next descendants

  Eg:- "B is a parent of E and F."

  or "E and F are child of B."

* Siblings are childrens of the same parent.

  Eg:- E and F are sibling to each other.

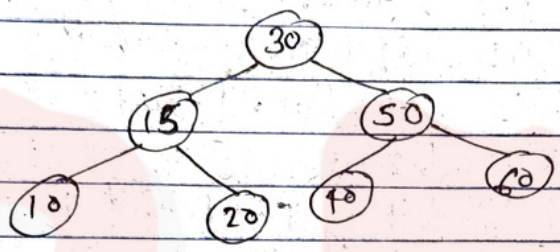
* Descendents - Descendents are all those set of nodes which can be reach from a particular node or under that nodes

  Eg:- For 'B' → E, F, J, K, M all these are descendent of 'B'.

  For 'D' → G, H, I, L, N, O

# Binary Search Tree

In this tree, for every node, all the elements in its left subtree are smaller than that node and its right subtree are greater than that node.




## Properties

i) No Duplicates

ii) Inorder gives sorted order - (10, 15, 20, 30, 40, 50, 60)

iii) If 'n' nodes are given then no. of BST are $\frac{2n_{C_n}}{n+1}$.

* Searching in a binary search tree -

```
Node * Rsearch(Node *t, int key)
{
    if (t == NULL)
        return NULL;
    if (key == t->data)
        return t;
    else if (key < t->data)
        return Rsearch(t->lchild, key);
    else
        return Rsearch(t->rchild, key);
}
```

* It depends upon height. (Here we consider min-height)

Time - O(log n)

30/06/22

# AVL Trees

(Height balanced binary trees are known as AVL trees)

(These trees are height balanced binary trees.)

→ These tree are balanced by using Balanced factors -

Balance factor = height of left-subtree − height of right subtree

This balance factor are calculated on every node of binary search tree.

or, BF = hl − hr = {-1, 0, 1} → Valid balanced factors

if |BF = |hl − hr| ≤ 1| Balanced condition

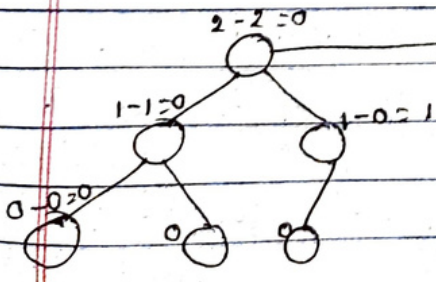
if |BF| = |hl − hr| > 1 → Imbalanced condition

If balanced factor of every node of a tree are '-1' or '0' or '1' then this tree is a height balanced tree means AVL trees. (called) And if not then it is a imbalanced tree.

→ Balanced factor are used to check that trees are height balanced or not.

→ If trees are not balanced then we perform Rotations for balancing that tree.

* How to check trees are height balanced or not using balance factor -




Balanced tree

for this node -

height of left-subtree = 2

" right " = 2

∴ BF = 2 − 2 = 0

Similarly for all nodes.

# Red-Black Tree Creation

Key - 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

N = Newly inserted node
P - Parent node
G - Grand Parent node
U - Uncle node

Insert 10

10 (Root node is always black)

Insert 20

10
20 (Newly inserted node is always red)

Insert 30

10
20
30

two consecutive red are not allowed

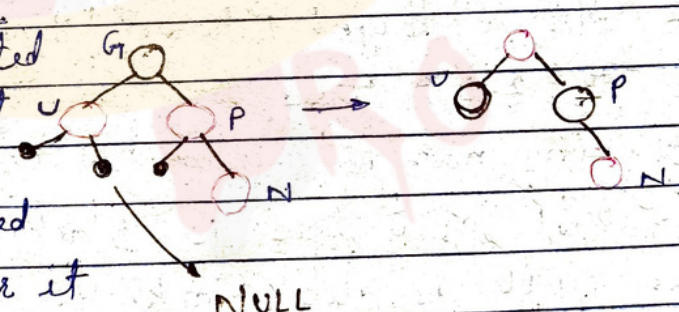
* When there is red-red conflict then we have to do some adjustment for making it as a balance Red-black tree.

→ There are two approch for balancing red-black tree -

i) Recoloring ii) Rotation

When uncle (U) is Red -

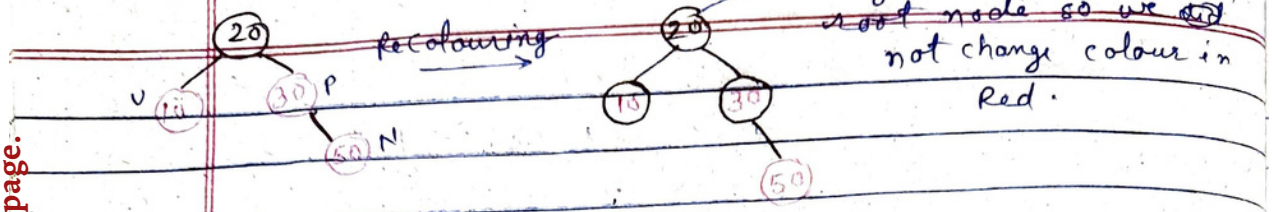
* For newly inserted node, if its parent (P) is Red and uncle (U) is also Red then we recolour it to balance.




→ (Only 'U' and 'P' will be recolour)

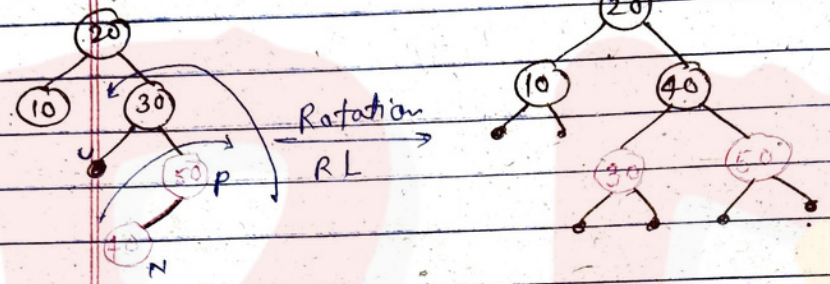
→ If grand Parent is not a root node then its colour will change to Red. Otherwise its colour should be Black. (If grand parent is a root node)

Insert 50

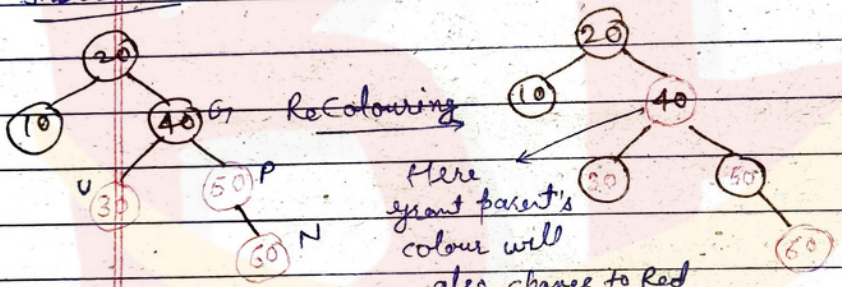



Here grand parent is root node so we will not change colour in Red.

Insert 40




Insert 60




Here grand parent's colour will also change to Red bcoz it is not a root node.

Insert 70