Data structures

09 June 2021 07:07

1. Array

W

2. Linked list allocations are scattered in heap. Use only memory needed

When u need allocate else release.
However LL uses more memory than array for eg
Dynamic memory alloc uses LL to keep track of alloc memory. OS scheduler process probably uses Circ LL.

```
// NOTES:
// 1. This is a sequencial data structure and is used to store data.
// 2. Linked list is special data structure in which data elements are linked to one another.
// 3. The pictorial look of a linked list is:

// OPERATIONS:
// Insertion, Deletion, Traversing.

// PROS
// 1. Linked list is dynamic data structure, it can grow and shrink at run time.
// 2. It is used to efficiently utilize memory
// 3. Insertion and Deletion is very easy at any position.

// CONS
// 1. More memory is required if list is too big.
// 2. Accessing elements is time consuming.

// TYPES OF LINKED LIST:
// 1. Singly Linked List
// 2. Doubly Linked List
// 3. Circular Linked List
// 4. Doubly Circular Linked List
```

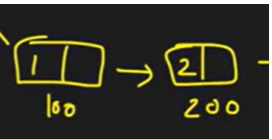
3. XOR LL

XOR of prev and next stored. Hence we need just one next. Behave like Doubly LL.

```
// NOTES:
// 0. This is singly linked list but act like doubly linked list.
// 1. We can go forward and reverse in XOR Linked List.
// 1. It is memory efficient doubly linked list.
// 2. Instead of storing address of next node, it stores XOR of previous and next address
// 3.

// General Linked List
//      A   ->  B   ->  C   ->  D   ->  NULL

// XOR Linked List
// Data       A         B          C          D
// Next    0 xor B   A xor C    B xor D    C xor 0

// PROS:
// 1. It takes less memory than doubly linked list but act like doubly linked list.

// CONS (Wiki):
// 1. Debugging becomes hard. (because we don't store actual address)
// 2. Code complexity.
// 3. Garbage Collection is difficult (because no proper addess is stored)
```

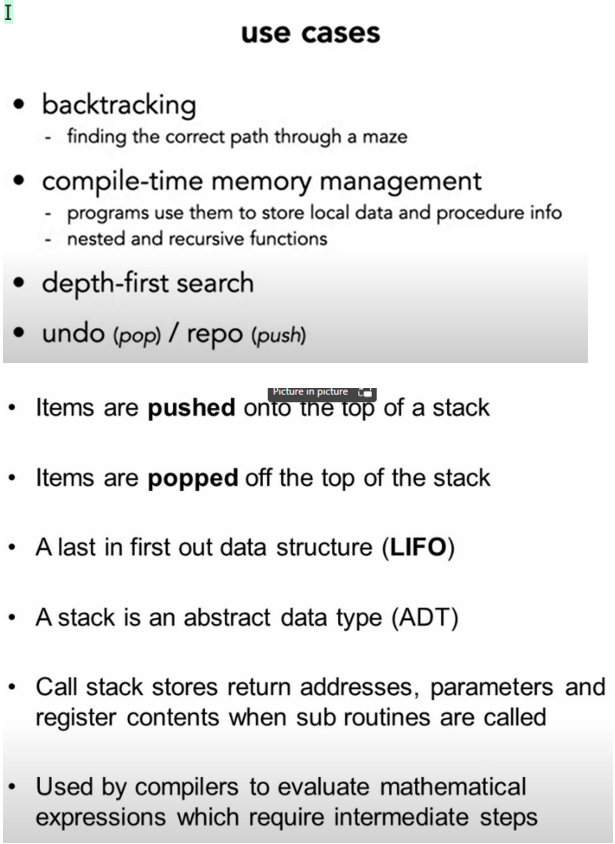
4. Circ LL

```
// NOTES:
// 1. Circular linked list is modified version of singly linked list.
// 2. In circular linked list last node point to first node.

// INSERTIONS:
// 1. Insertion in an empty list
// 2. Insertion at the beginning of the list
// 3. Insertion at the end of the list
// 4. Insertion in between the nodes

// ADVANTAGES:
// 1. CPU sheduling
// 2. Queue can be implemented using one pointer (last). Becuase next to last is always first !
//    so we don't have to implement first pointer at all.
// 3. Applications where you don't want to reset once you reach to last node.
```

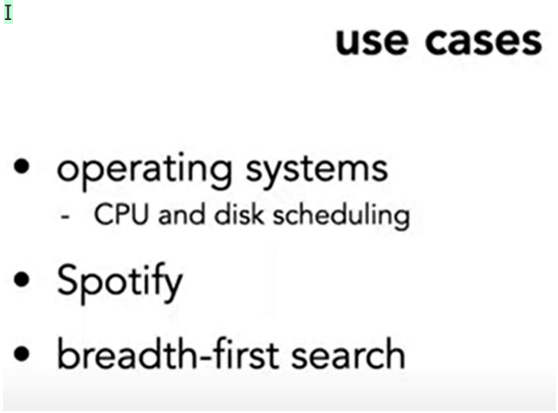


I

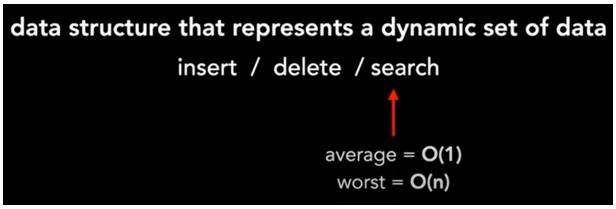
5. F
6. F
   I
7. F
   I
8. F
   I
9. Stacks
   I

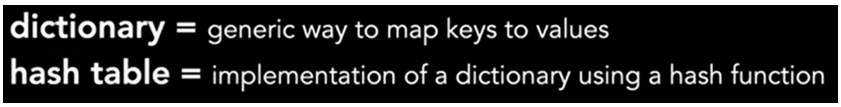


10. Queues
    I

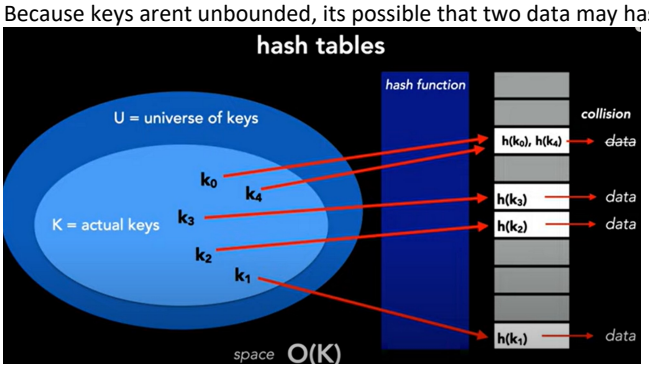


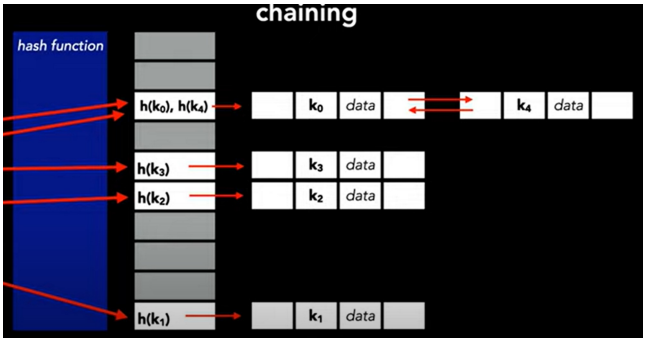
11. F
    I
12. Hash tables

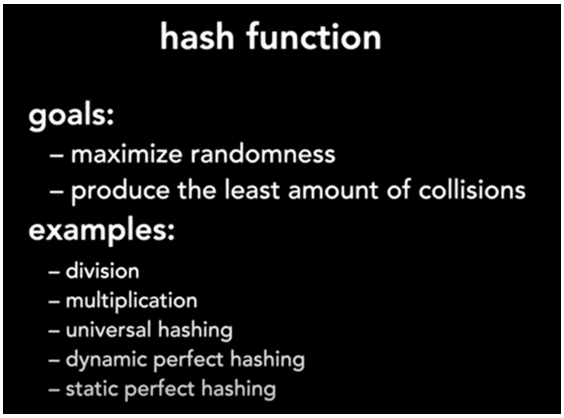




Because keys arent unbounded, its possible that two data may hash to same key. So we have LL to maintain list of data that hash to same key







Objectives of Hash Function

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

Linear probing/ Open addressing.. Linear search o find next avail slot (or) to find item

$$\text{Load Factor} = \frac{\text{Total number of items stored}}{\text{Size of the array}}$$

Hash Tables and Hash Functions

Collison Resolution

- Open addressing
  - Linear probing
  - Plus 3 rehash
  - Quadratic probing (*failed attempts*)^2
  - Double hashing
- Closed addressing

I

13. Heap use nearly complete binary tree-i.e. all levels are filled except last

What is a heap?

A heap has to be a ***balanced*** binary tree, i.e. either the lowest level or the level above the lowest level must all have ***leaves***.

In other words, this tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

In simple terms, it's a nearly complete binary tree.

Max-Heap:
for every node i other than the root: A[parent(i)] >= A[i]
in other words, the value of a node is at most the value of its parent
which indicates, the largest element in a max-heap is at the root

Min-Heap:
the opposite of Max-Heap
for every node i other than the root: A[parent(i)] <= A[i]
the value of a node is at least the value of its parent
the smallest element in a min-heap is at the root

height
O(logn)

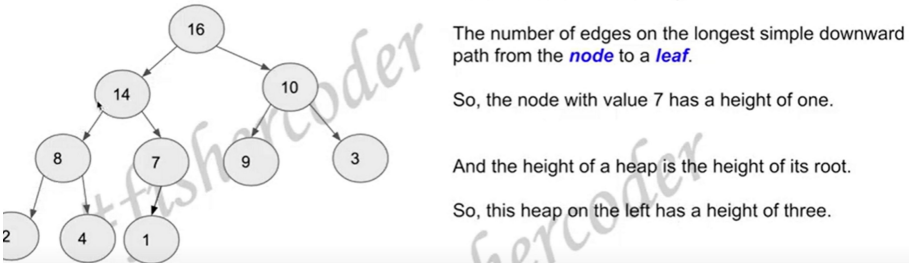
This could be simplified to:

$$2^h <= n < 2^{(h+1)}$$

which could be transformed to:

$$h <= \boldsymbol{lgn} < h + 1$$

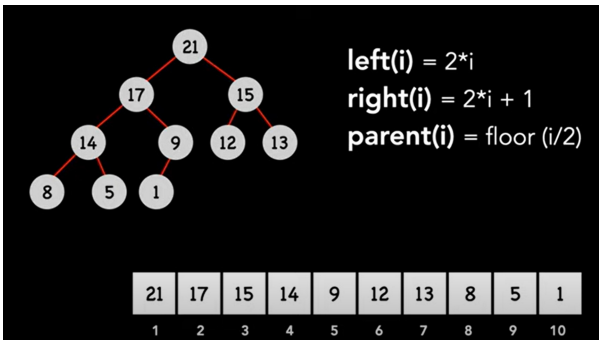
so, in θ notation, it could be written as $h = \theta(\boldsymbol{lgn})$



Height - 0th to xth … log2(n)
Min number of nodes for height = 2^height
Max number of nodes for height = 2^(height+1) - 1

2^h <= n <=	2^(h+1) - 1

h <= log2(n) <= h+1
Height = Θ(n)
All operations are in terms of height i.e. Θ(n)



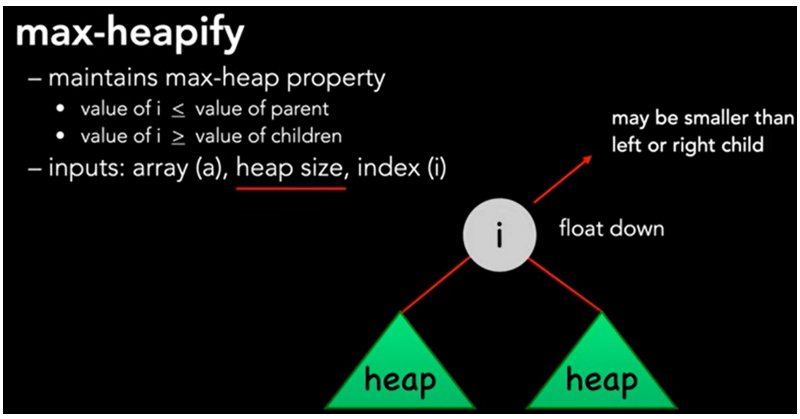
Max-heapify assumes already bottom nodes are heaps. So if imbalance is found then it has to go down by O(log n) only to fix it

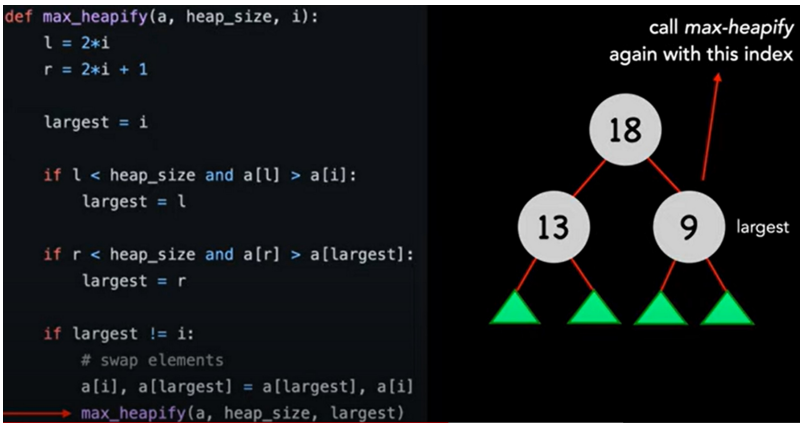
MAX-HEAPIFY

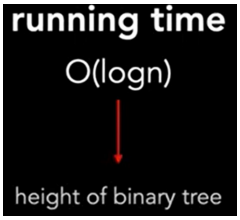
This is a procedure call to maintain the max-heap property.

MAX-HEAPIFY assumes the subtrees rooted at LEFT(i) and RIGHT(i) are max heaps, only that A[i] might be smaller than its children, thus violating the max-heap property.

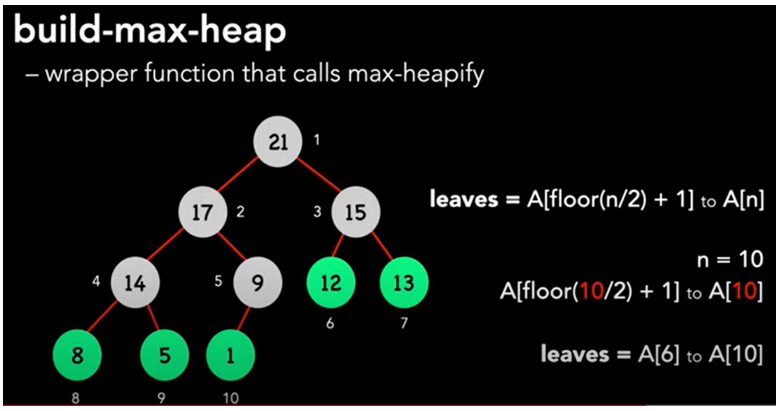
So, MAX-HEAPIFY lets the value at A[i] float down in the max heap so that the subtree still obeys the max-heap property.

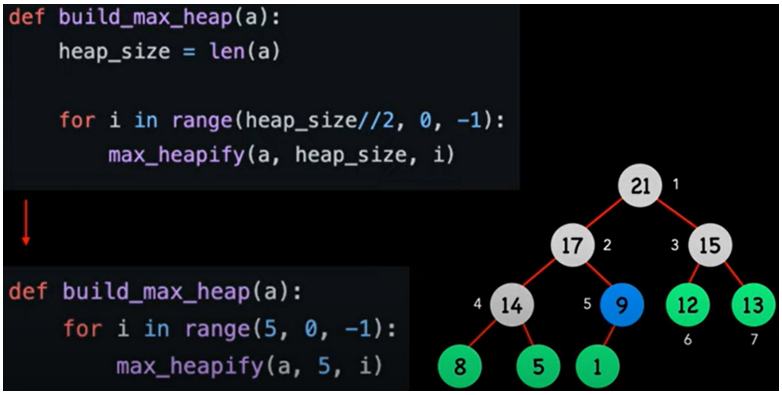


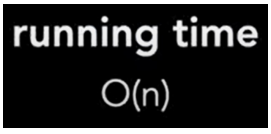




Leaves are already heaps. So start from non-leaf node and call max-heapify on it since nodes below it are already heaps.







I

14. Priority Queue

Priority Queue:

Max PQ -> backed by Max Heap

Scheduled jobs on a shared computer, etc.

Min PQ -> backed by Min Heap

Event-driven simulator, etc.

I

15. F

I

16. F
17. F

I

18. F

I

19. F

I

20. F

I

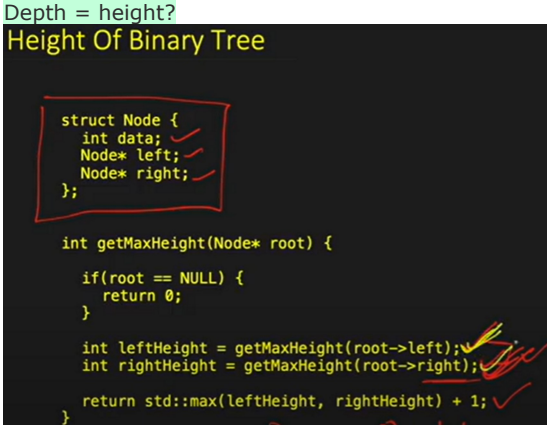
21. Tree

```
// TOPIC: Tree Data Structure Introduction

// NOTES:
// 0. Tree is a hierarchical data structure which stores the information in the form of hierarchy.
// 1. Tree is a non-linear data structure compared to arrays, linked lists, stack and queue.
// 2. Tree represents the nodes connected by edges.
// 3. Tree is one of the most powerful and advanced data structures.

// TREE TERMINOLOGIES:
// Root, Parent Node, Child Node, Siblings, Path, Height of Node, Height of Tree, Depth of Node, Degree of Node, Edge.


// TYPES OF TREE WE WILL COVER IN THIS SERIES:
// Binary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, Splay Tree, N-ary Tree, Tri Structure,
// Suffix Tree, Huffman Tree, Heap Structure, B Tree, B+ Tree, R Tree, Counted-B Tree, K-D Tree,
// Decision Tree, Markel Tree, Fenwick Tree, Range Tree


// ADVANTAGES OF TREE DATA STRUCTURE:
// 0. Tree reflects structural relationships in the data.
// 1. It is used to represent hierarchies.
// 2. It provides an efficient insertion and searching operations.
// 3. Trees are flexible. It allows to move subtrees around with minimum effort.
```

https://towardsdatascience.com/5-types-of-binary-tree-with-cool-illustrations-9b335c430254

Edge - connecting line
Degree - number of children

Height - number of nodes below
Depth = height?

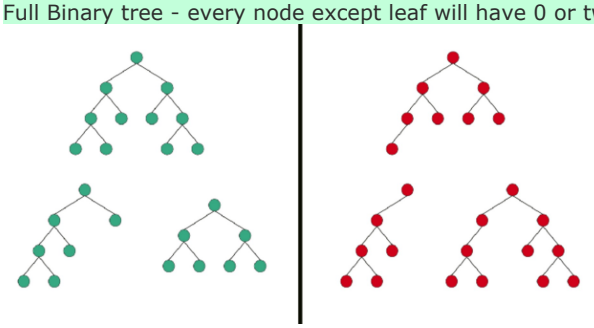


Size = number of nodes

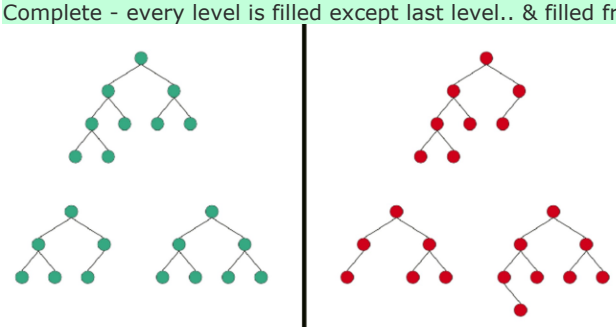
Size of Binary Tree

```
int getSize(Node* root) {

  if(root == NULL) {
    return 0;
  }

  int leftSize = getSize(root->left);
  int rightSize = getSize(root->right);

  return leftSize + rightSize + 1;
}
```

Full Binary tree - every node except leaf will have 0 or two children



Almost complete - every level except last and last-1 levels have no children
Complete - every level is filled except last level.. & filled from left



Left/Right skewed - every node has left or right children only

Perfect Binary tree

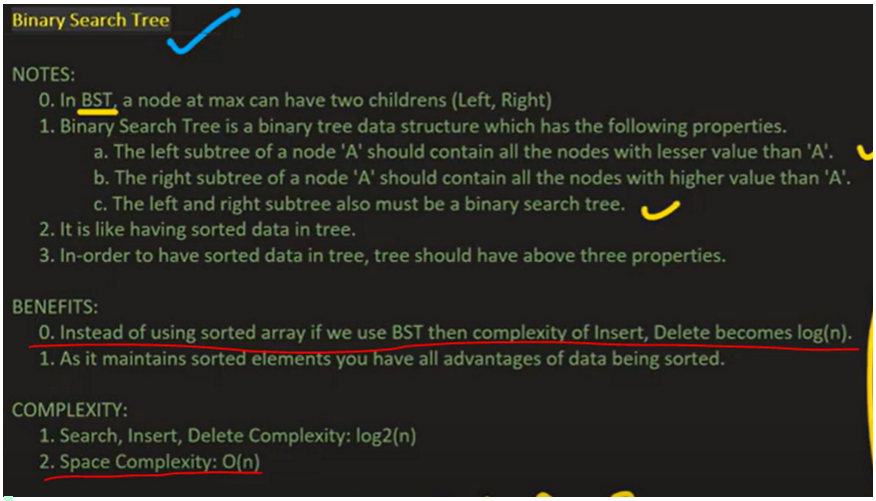
Balanced - height of left and right part of tree differ by 1

balanced search trees

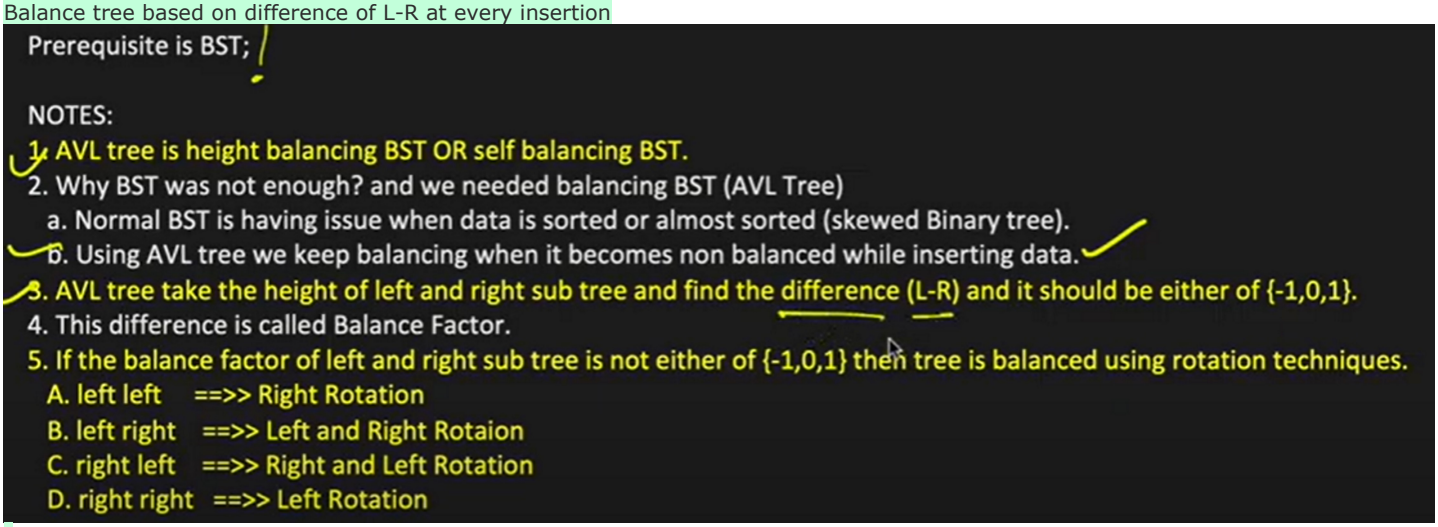
guaranteed height of **O(log n)** for **n** items

D

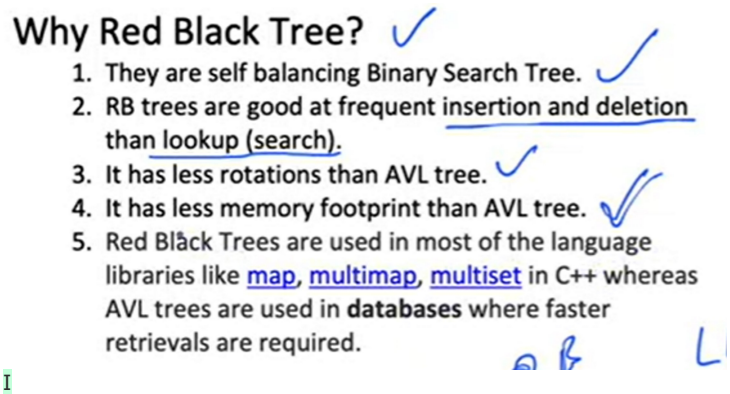
22. F
    I
23. BST



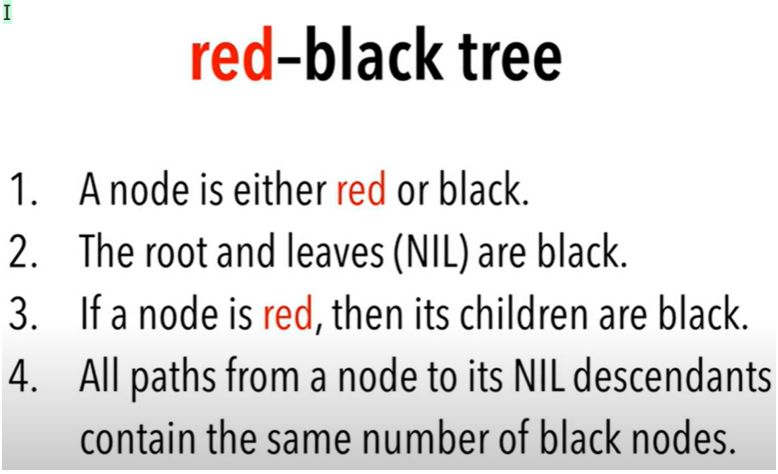
    I
24. AVL Tree
    https://www.youtube.com/watch?v=20cnaIwJY9I&list=PLk6CEY9XxSIBG2Gv6-d1WE3Uxqx94o5B2&index=19&ab_channel=CppNuts
    Balance tree based on difference of L-R at every insertion



    I
25. Red black tree



I



Red-black trees in 4 minutes — Intro

26. F
    I
27. F
28. F
    I
29. F
    I
30. F
    I
31. F
    I
32. F
    I
33. F
    I
34. D
    I
35. F
    I

36. F
    I
37. F
    I
38. F
39. F
    I
40. F
    I
41. F
    I
42. F
    I
43. F
    I