B-tree provides an efficient way to insert and read data. In actual database implementation, the database uses both B-tree and B+tree together to store data. B-tree used for indexing and B+tree used to store the actual records.

Introduction of B-Tree

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Note: “n” is the total number of elements in the B-tree

According to Knuth's definition, a B-tree of order m is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node, except for the root and the leaves, has at least ⌈m/2⌉ children.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

Applications of B-Trees:

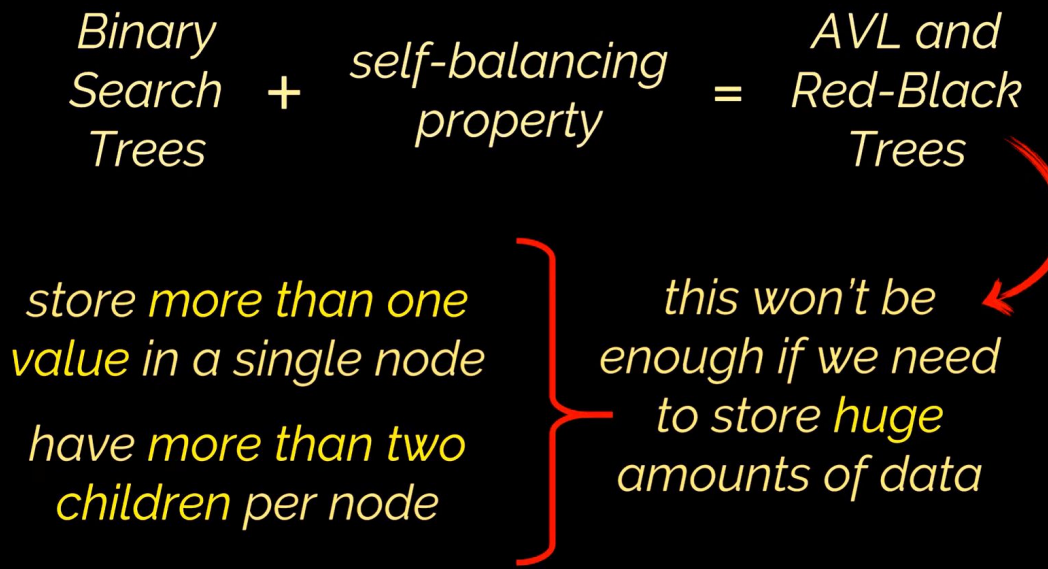
- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures.


Binary Search Trees
+
self-balancing property
=
AVL and Red-Black Trees
store more than one value in a single node
have more than two children per node
this won't be enough if we need to store huge amounts of data

B-Trees

store more than one value in a single node

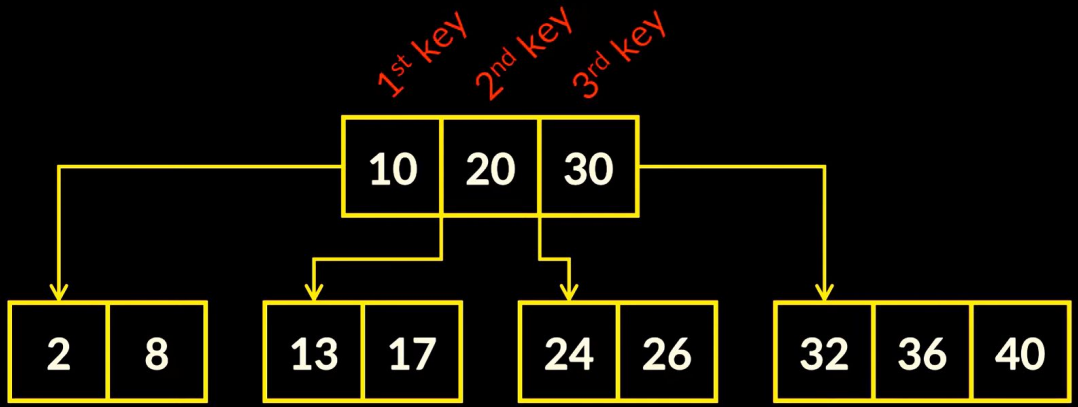
have more than two children per node

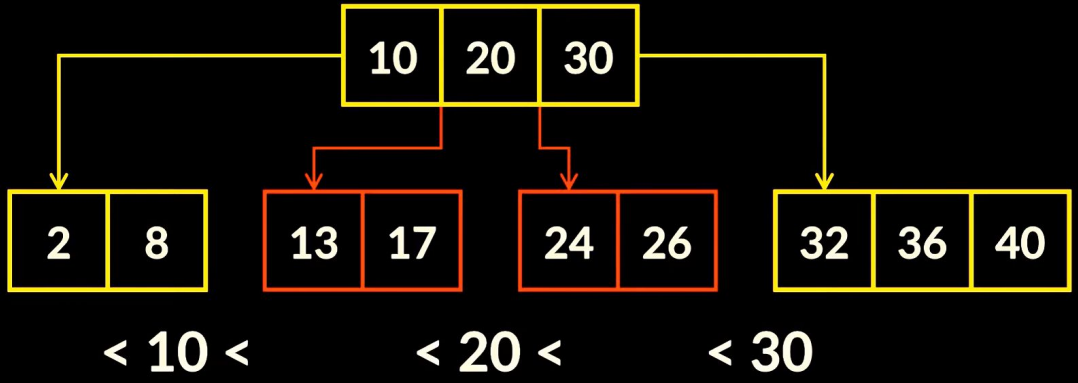
will fatten the tree making all the BST operations still applicable and efficient on large amounts of data

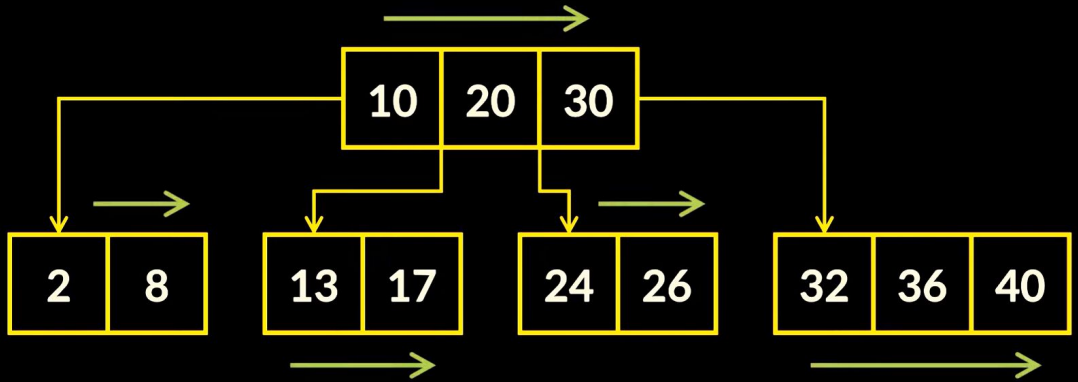
B-Tree of order ‘m’

if smaller go to the left, if greater go to the right

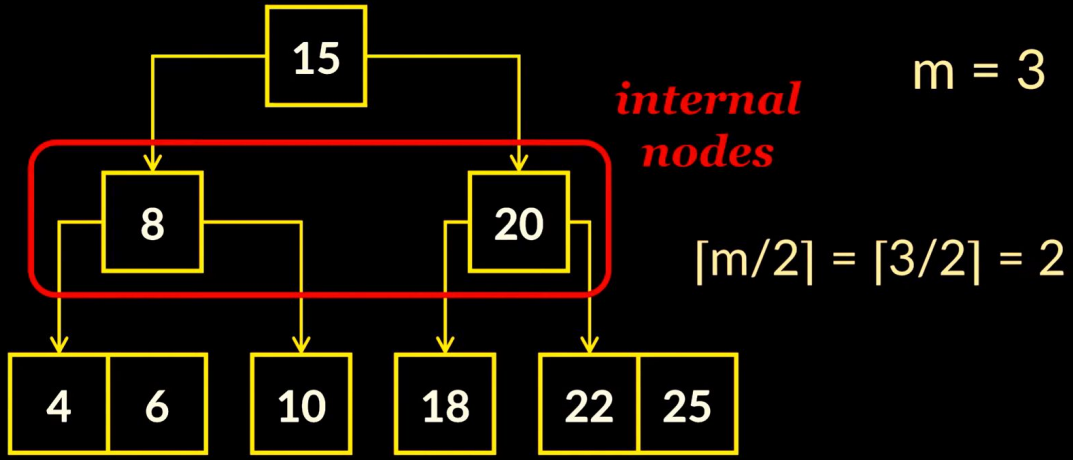
*every node can have at most ‘**m**’ children and can store at most ‘**m-1**’ keys or values*


1st key
2nd key
3rd key
10
20
30
2
8
13
17
24
26
32
36
40

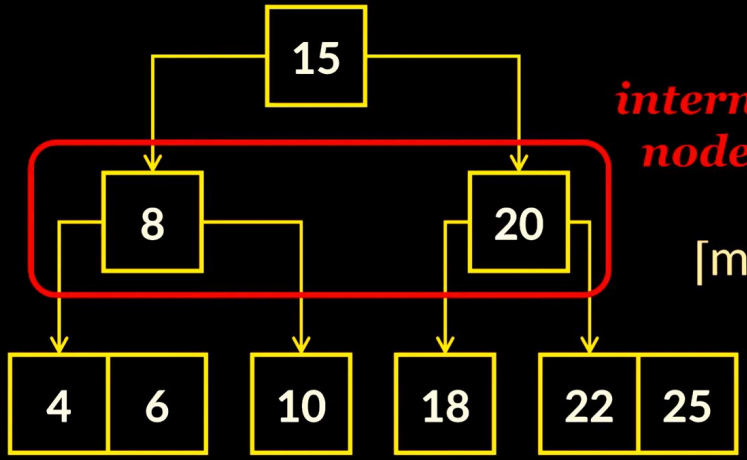

10
20
30
2
8
13
17
24
26
32
36
40
< 10 <
< 20 <
< 30


10
20
30
2
8
13
17
24
26
32
36
40

for a tree to be declared as a valid B-Tree of order 'm', it should satisfy three additional properties



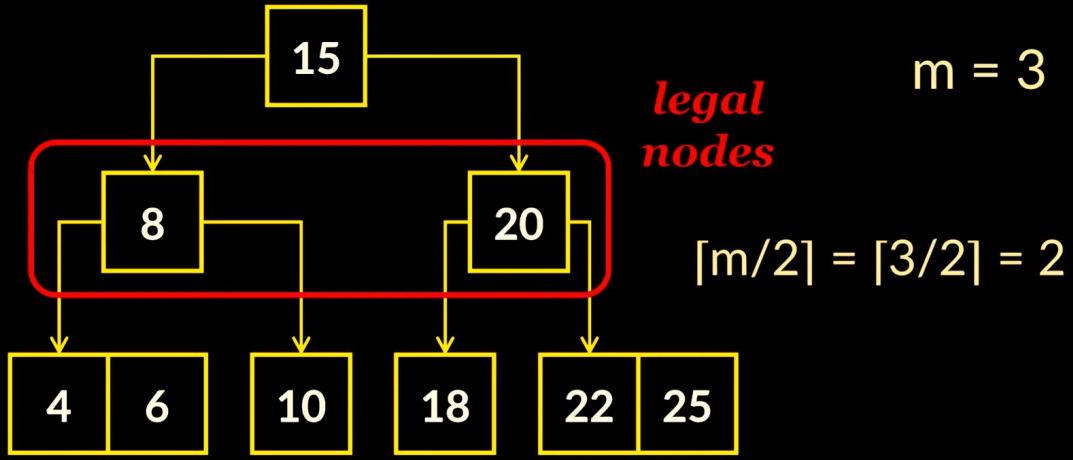
every internal node should have at least **[m/2]** child nodes



m = 3

[m/2] = [3/2] = 2

this property ensures that each internal node is at least **half-full**

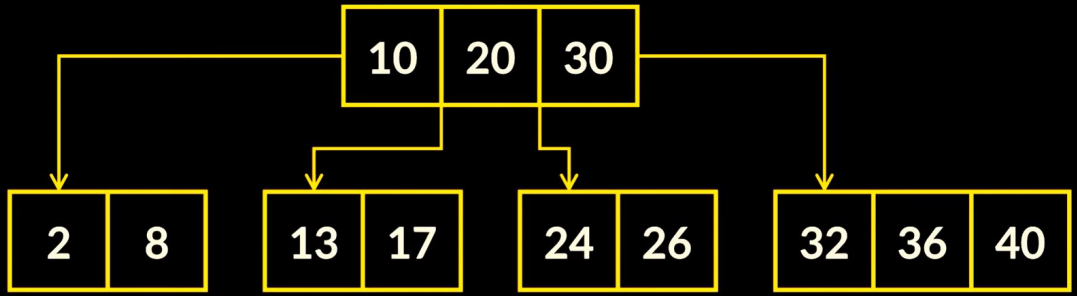


m = 3

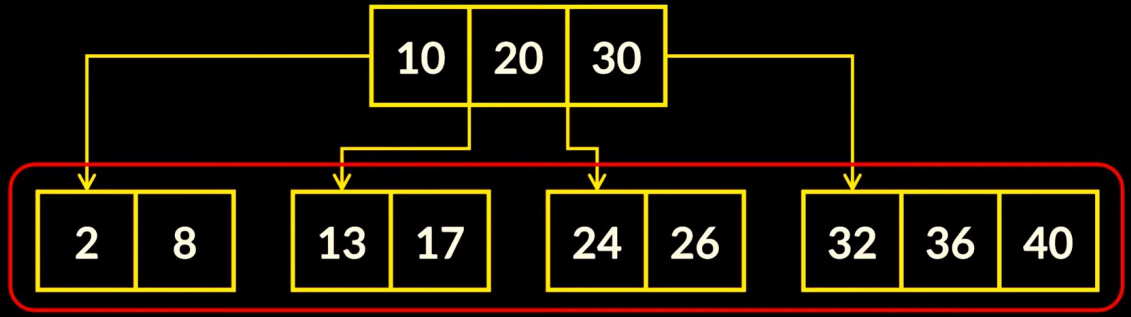
[m/2] = [3/2] = 2

this property ensures that each
internal node is at least **half-full**

this node has 4 children then it MUST have 3 keys

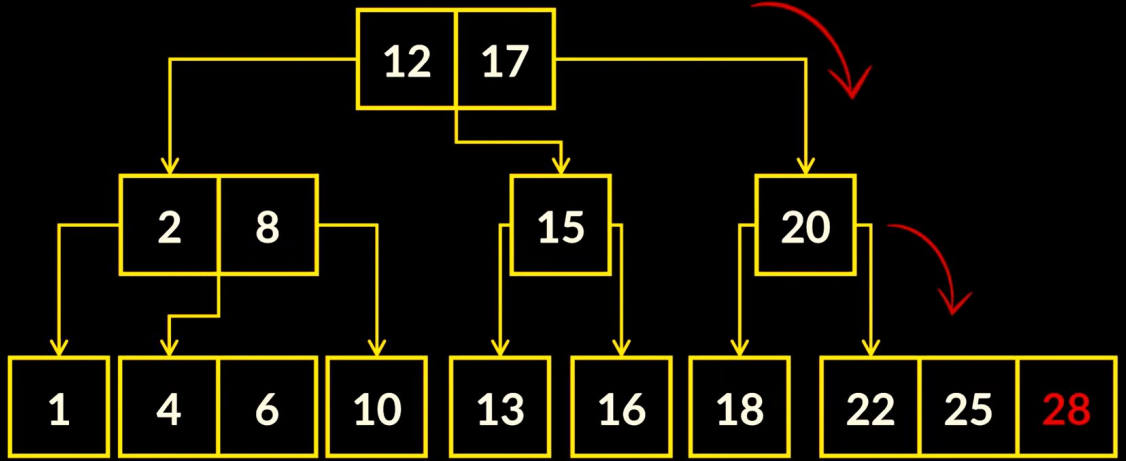


if a non-leaf node has **'n' children**, then it must contain **'n-1' keys or values**

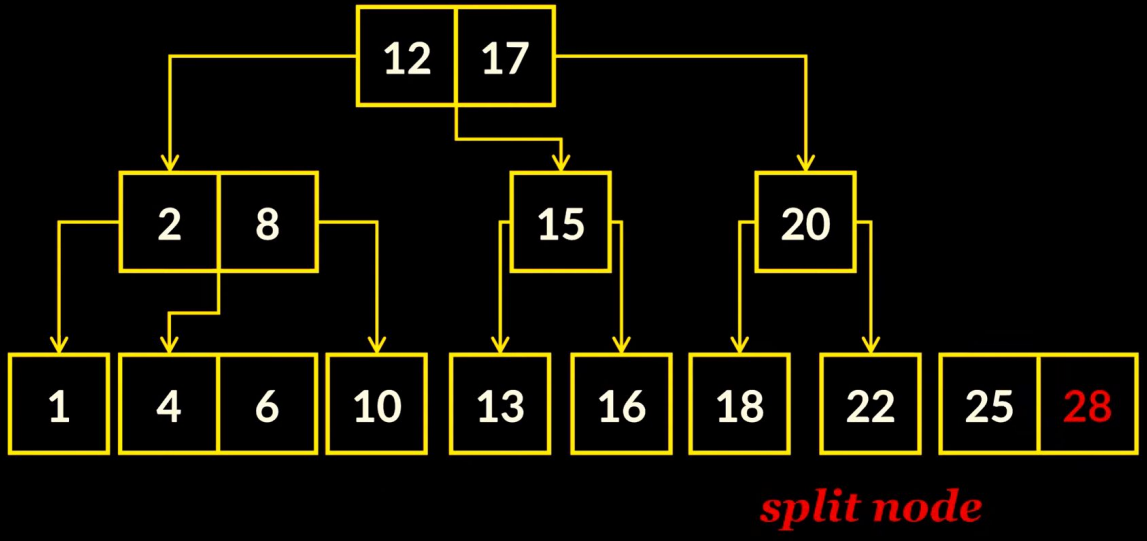


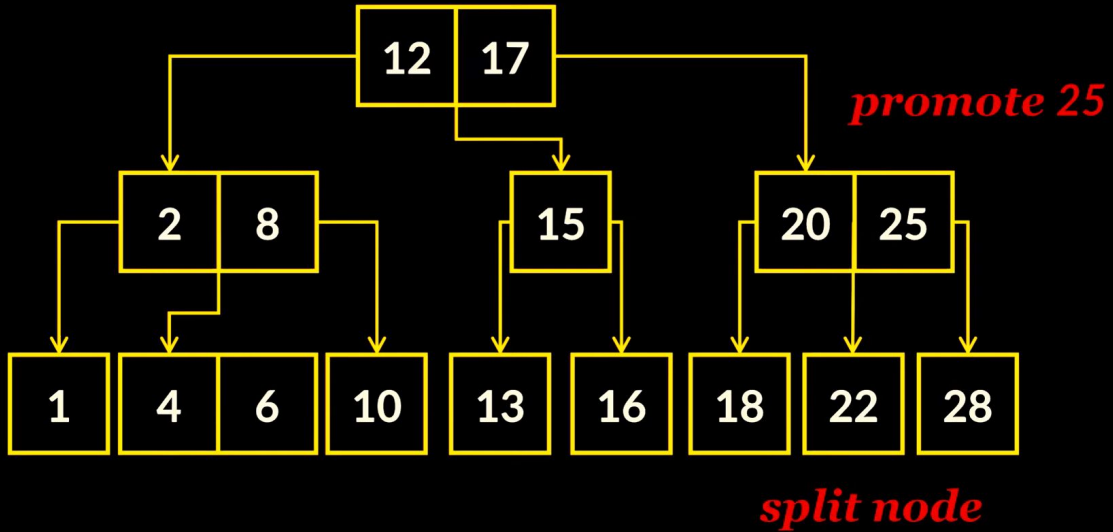
all **leaf** nodes should appear at the
same level of the tree

```
insert(28)
```

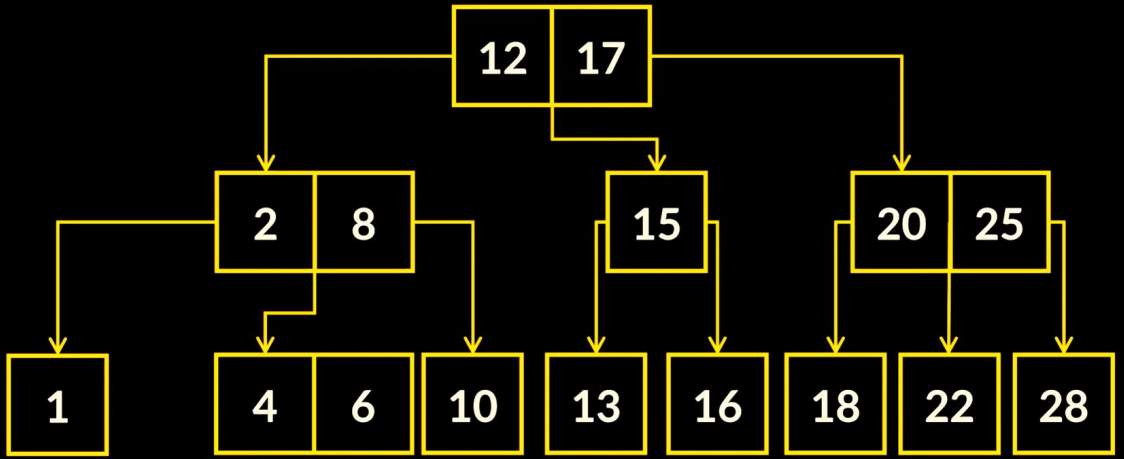


the maximum number of keys each node is allowed to hold is two

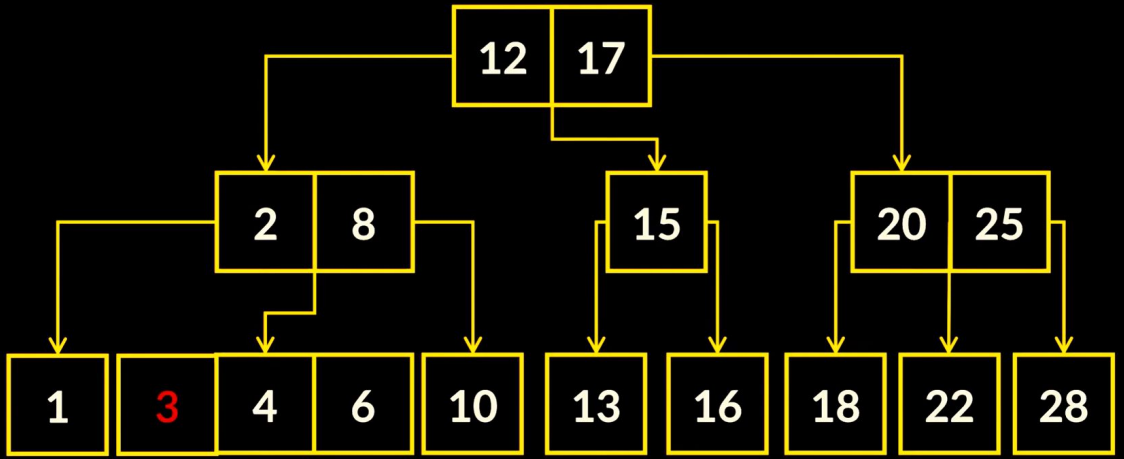

12
17
2
8
15
20
1
4
6
10
13
16
18
22
25
28
split node

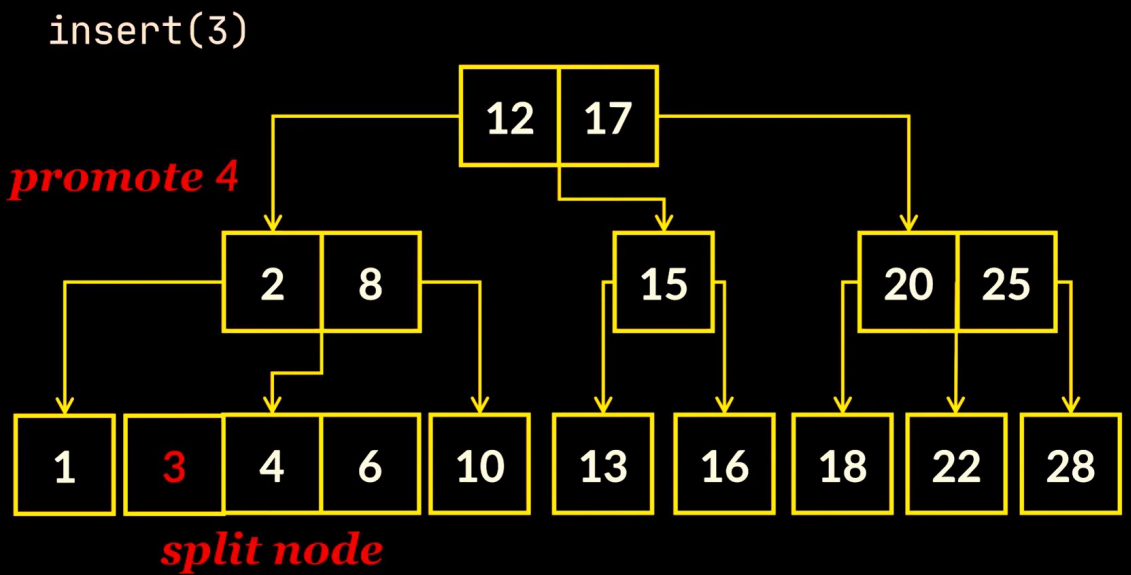

12
17
promote 25
2
8
15
20
25
1
4
6
10
13
16
18
22
28
split node

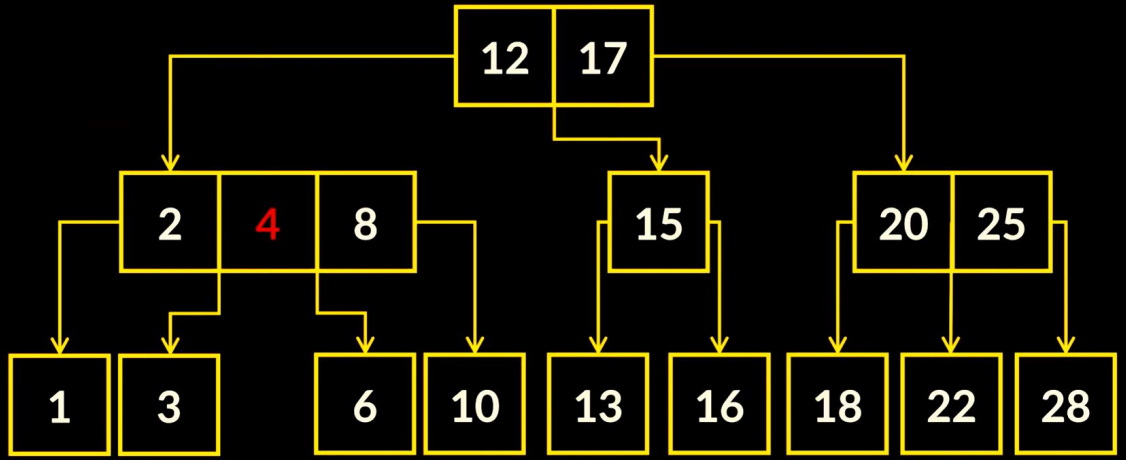
insert(3)

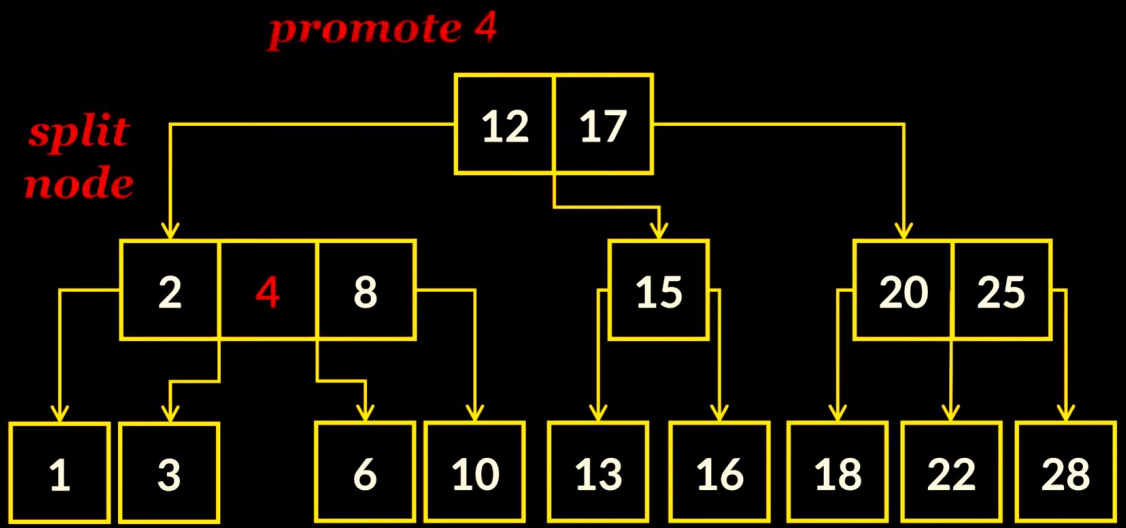


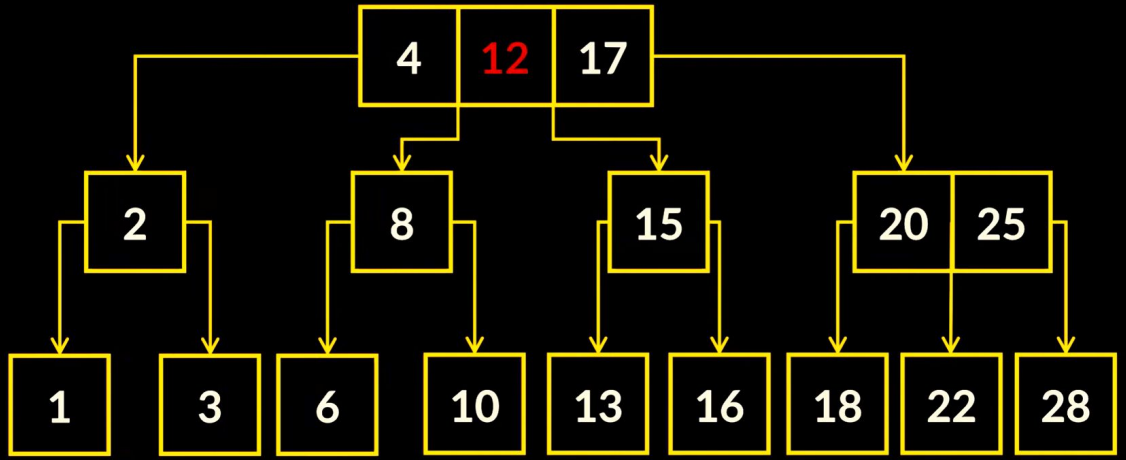
insert(3)

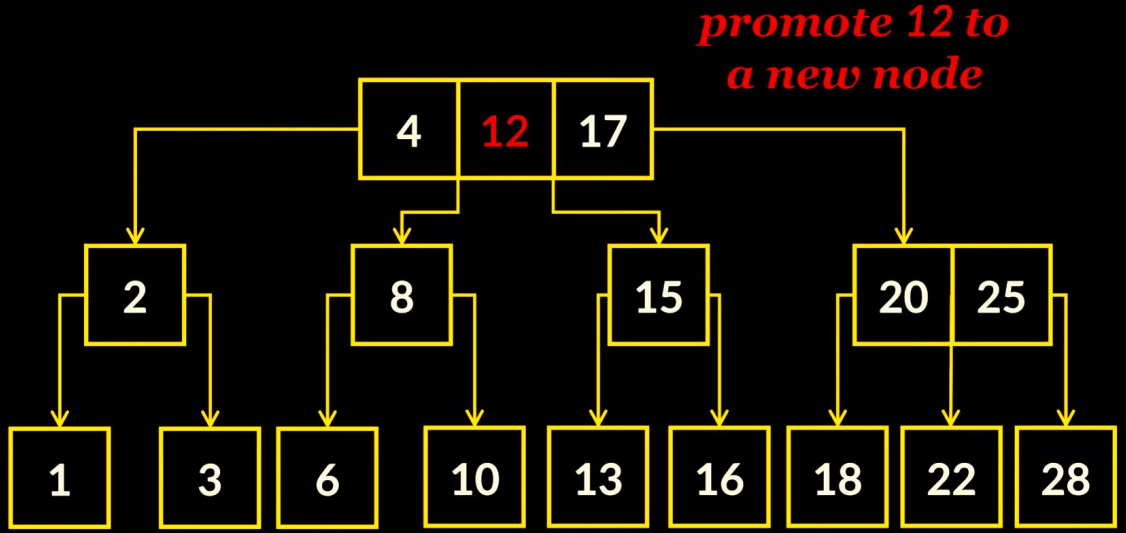


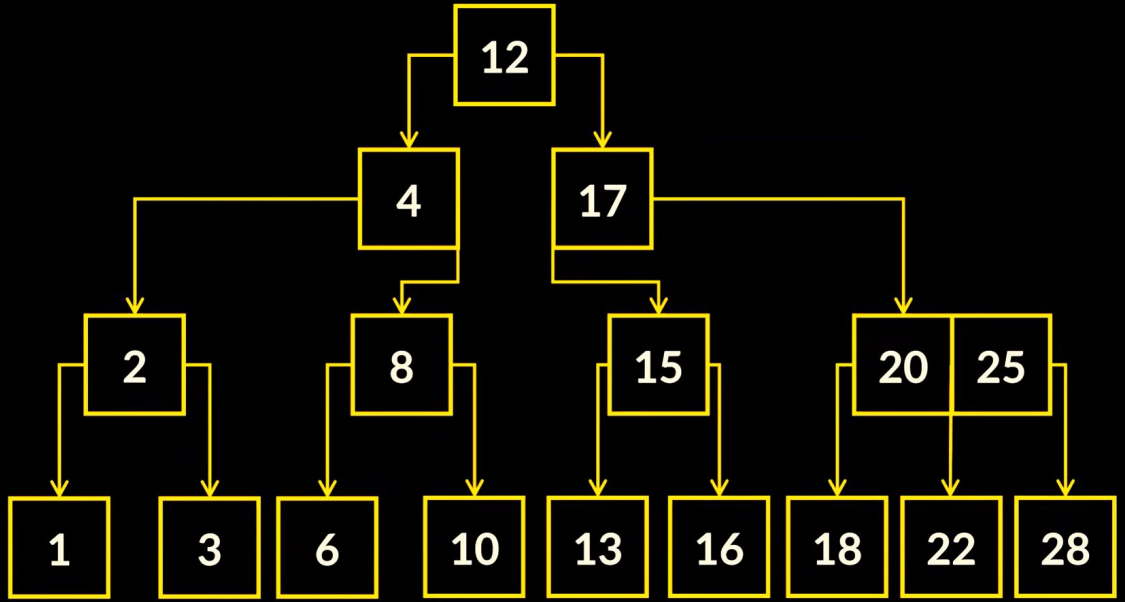

insert(3)
12
17
promote 4
2
8
15
20
25
1
3
4
6
10
13
16
18
22
28
split node

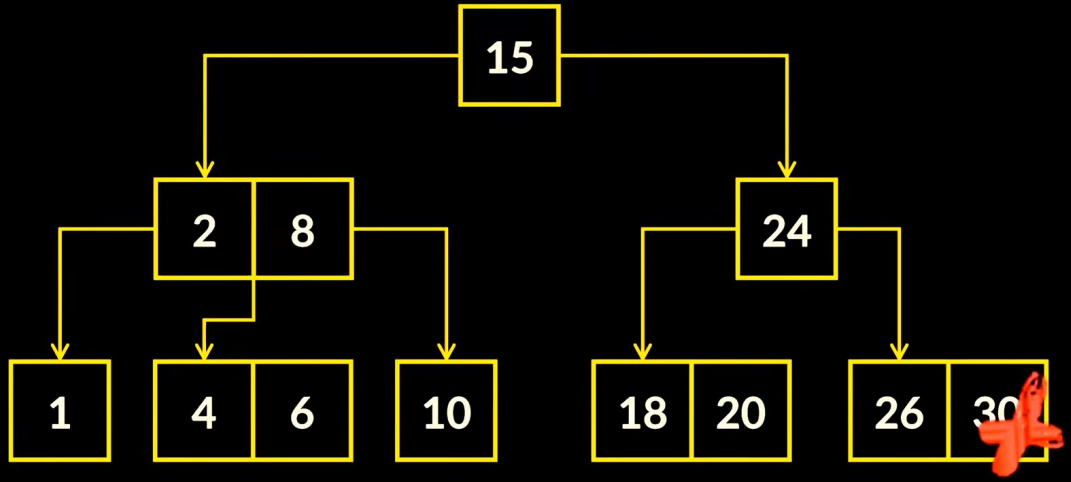

12
17
2
4
8
15
20
25
1
3
6
10
13
16
18
22
28

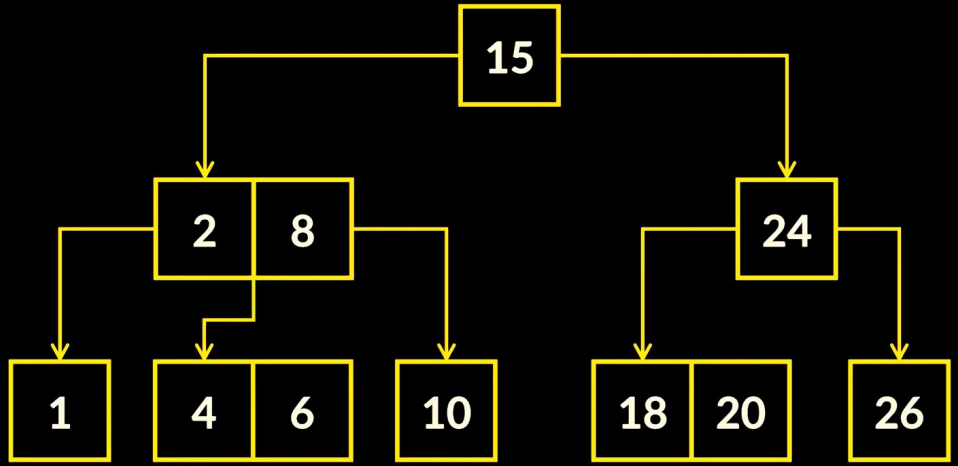

promote 4
split node
12
17
2
4
8
15
20
25
1
3
6
10
13
16
18
22
28

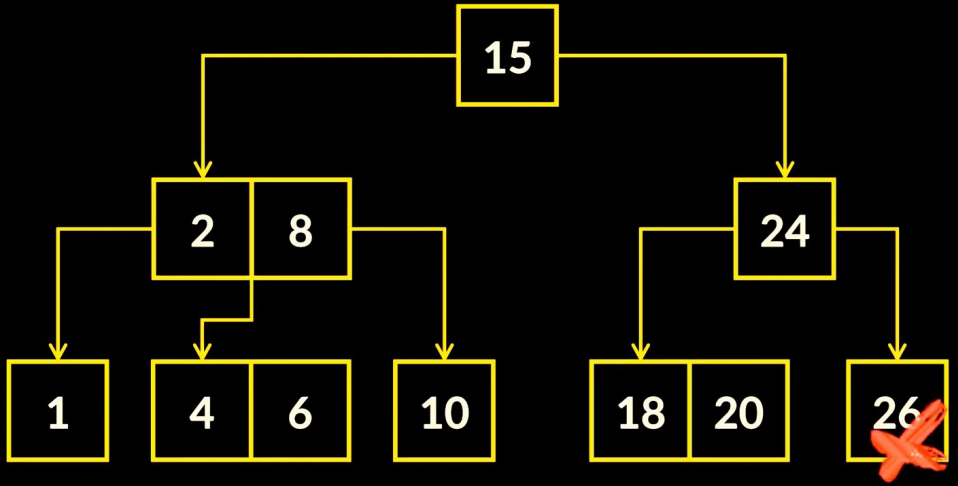

4
12
17
2
8
15
20
25
1
3
6
10
13
16
18
22
28

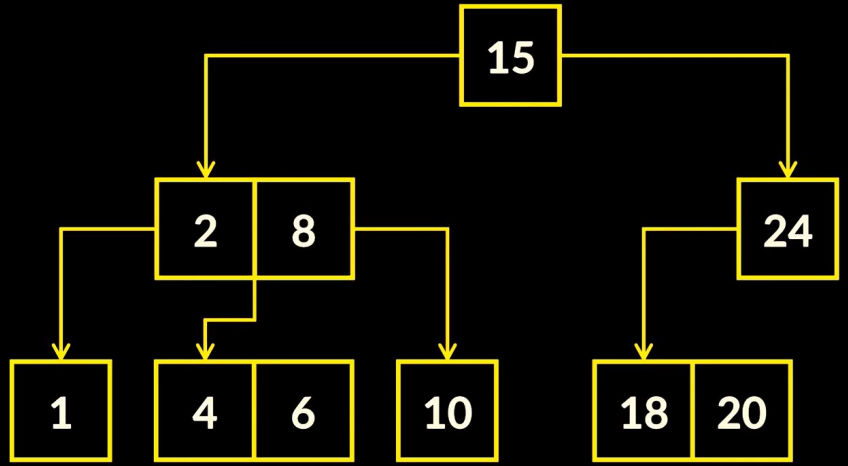

promote 12 to
a new node
4
12
17
2
8
15
20
25
1
3
6
10
13
16
18
22
28


12
4
17
2
8
15
20
25
1
3
6
10
13
16
18
22
28

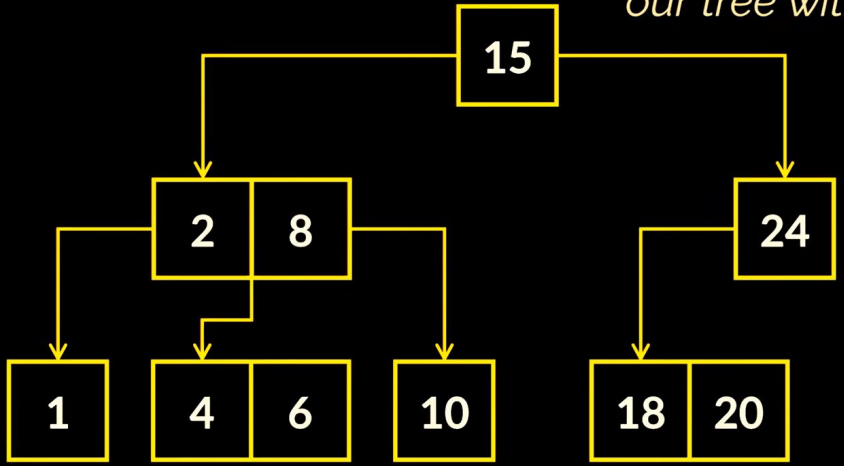

15
2
8
24
1
4
6
10
18
20
26
30


15
2
8
24
1
4
6
10
18
20
26

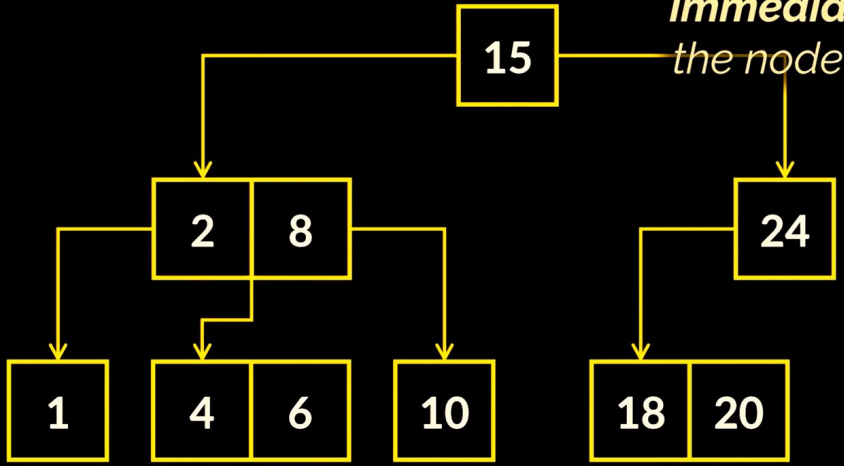

15
2
8
24
1
4
6
10
18
20
26

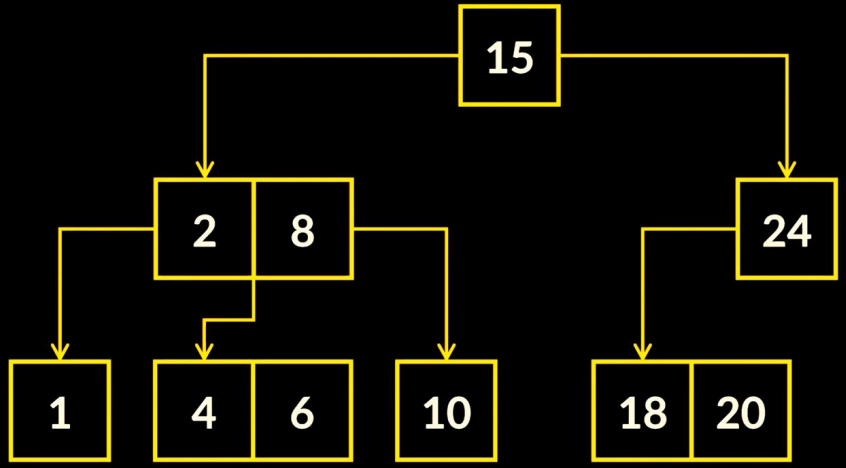

15
2
8
24
1
4
6
10
18
20

*we remove it, then **balance** our tree with the help of rotations*

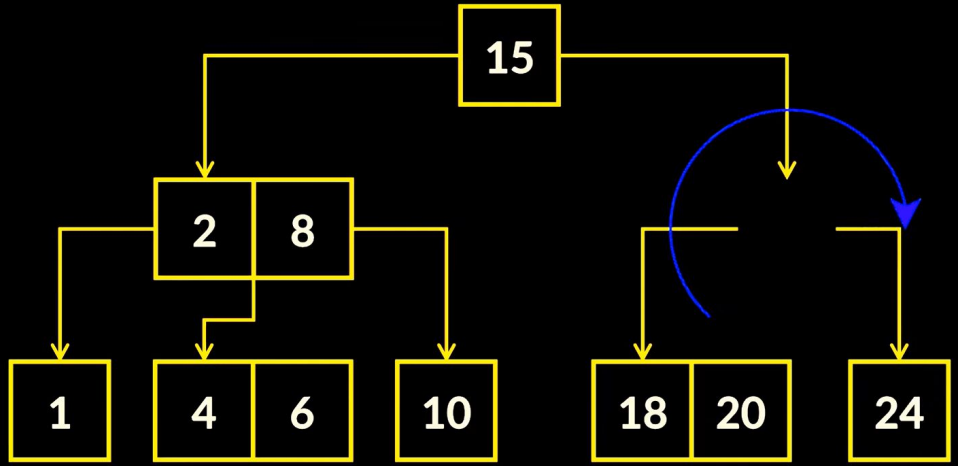


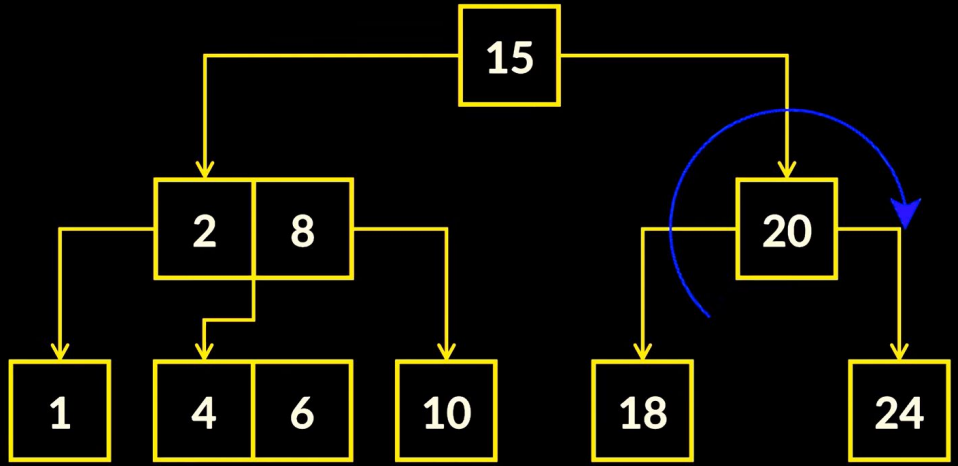
*we need to look at the **immediate** siblings of the node we removed*

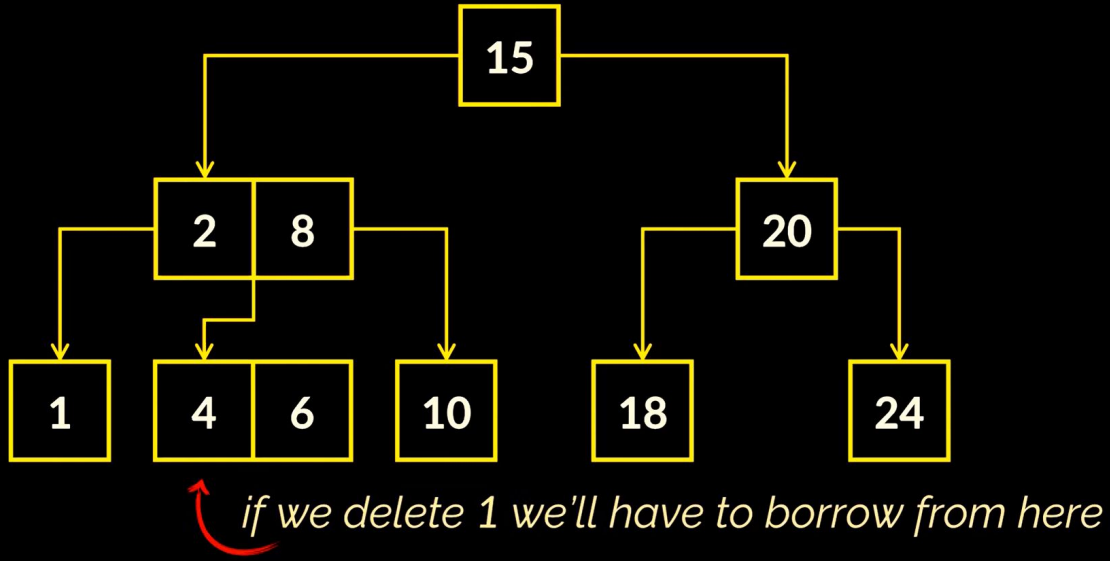


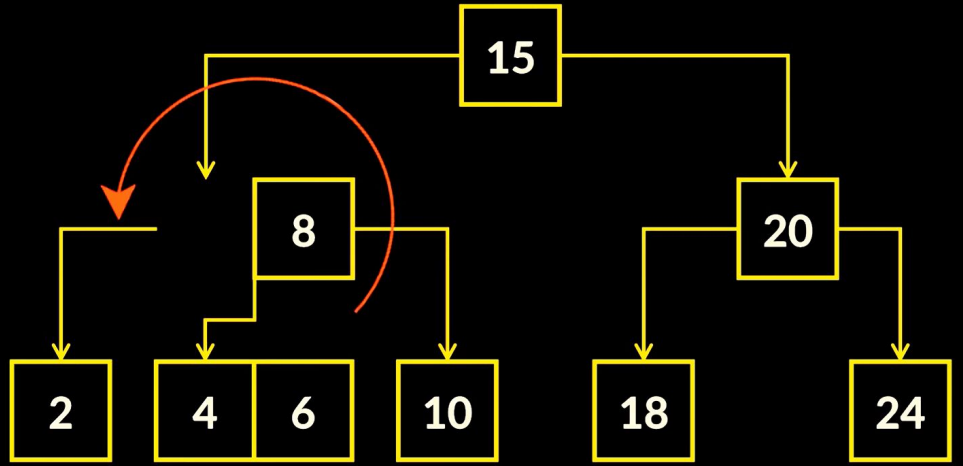


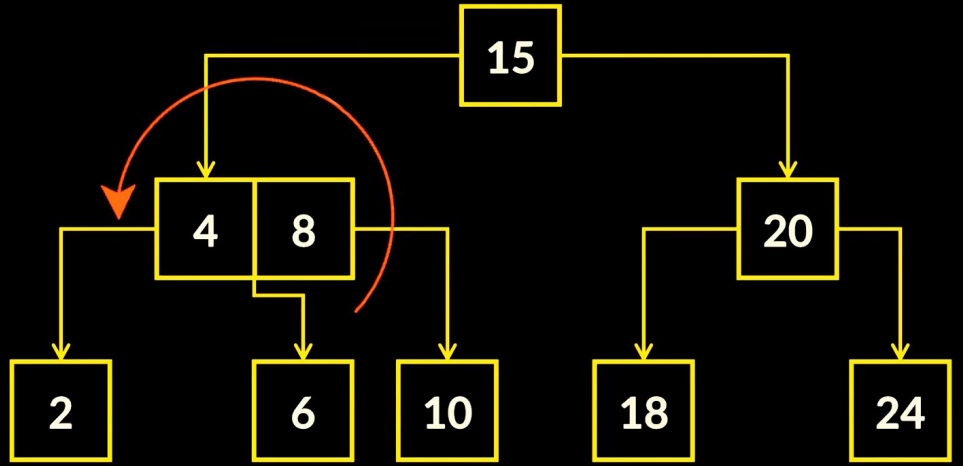
we can balance our tree with the help of a right rotation

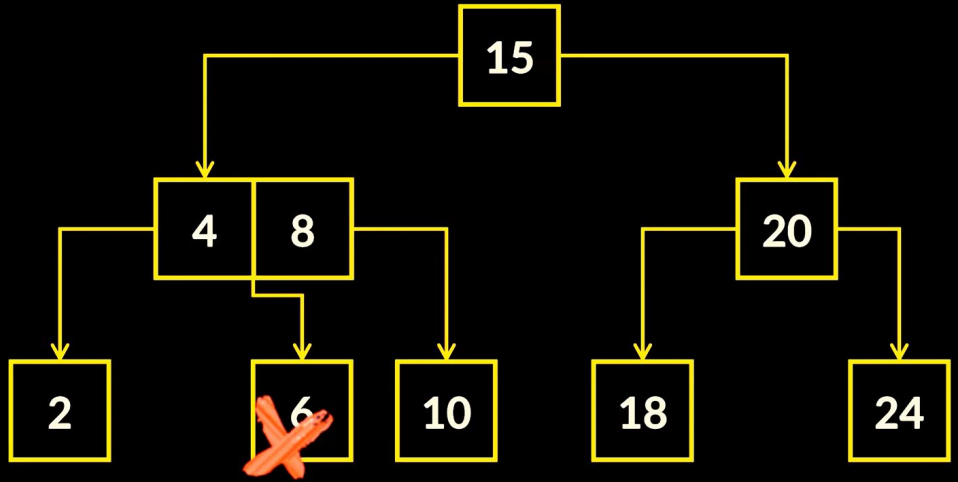

15
2
8
1
4
6
10
18
20
24

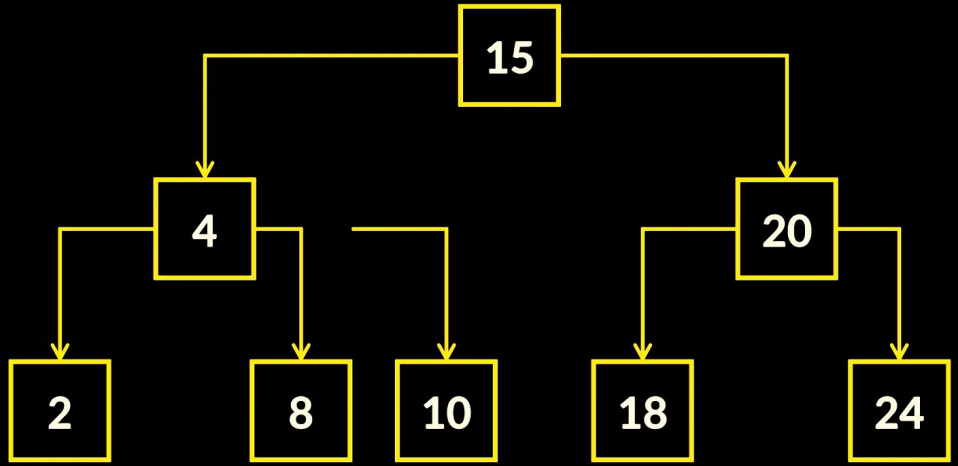

15
2
8
20
1
4
6
10
18
24

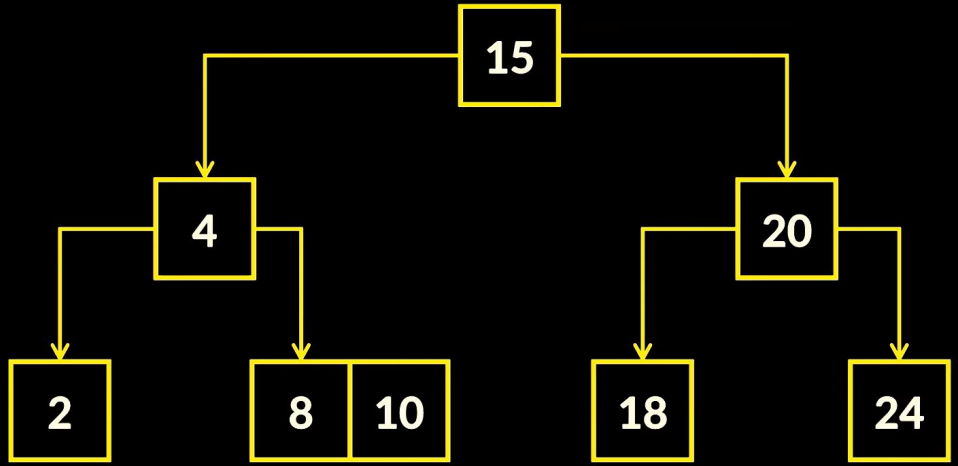

15
2
8
20
1
4
6
10
18
24
if we delete 1 we'll have to borrow from here

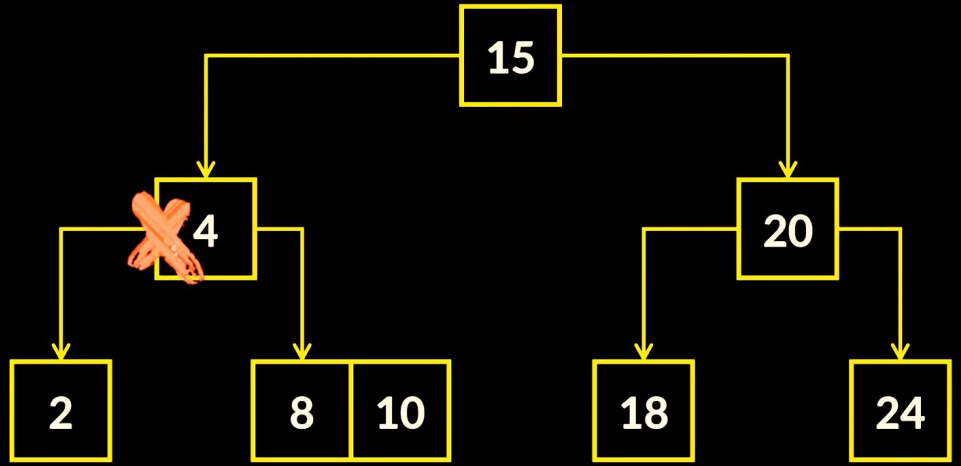

15
8
20
2
4
6
10
18
24

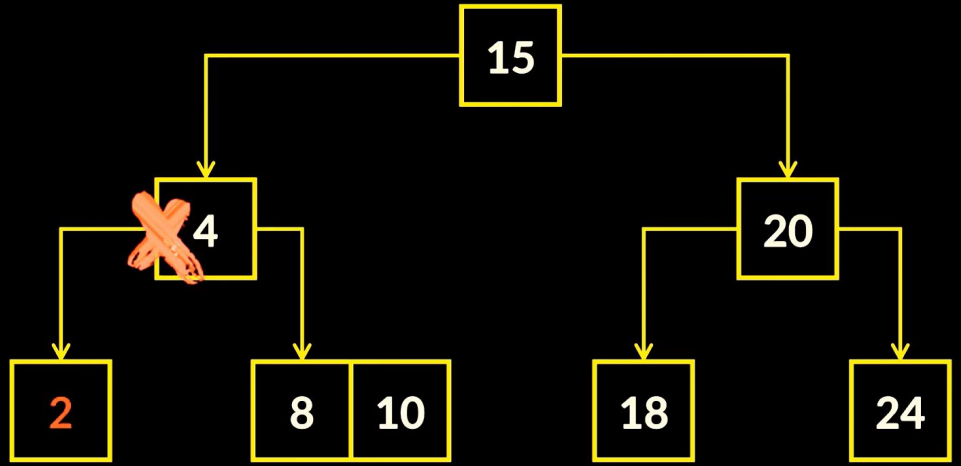

15
4
8
20
2
6
10
18
24

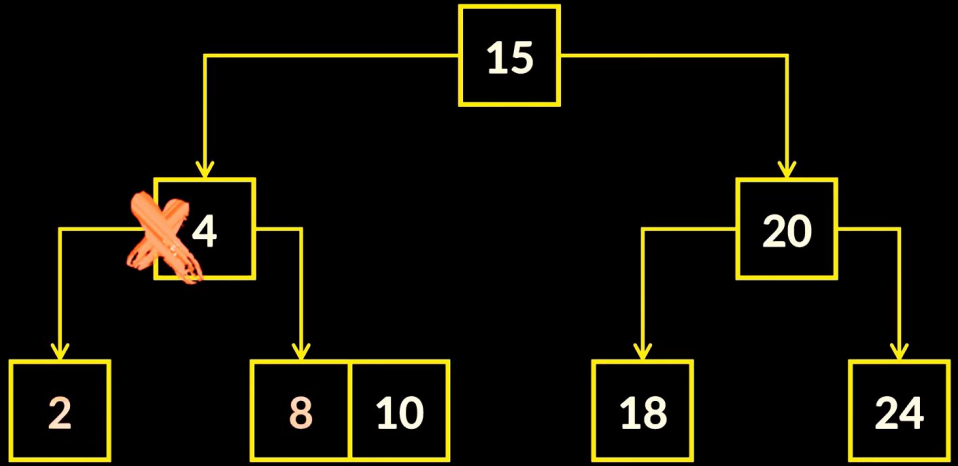

15
4
8
20
2
6
10
18
24

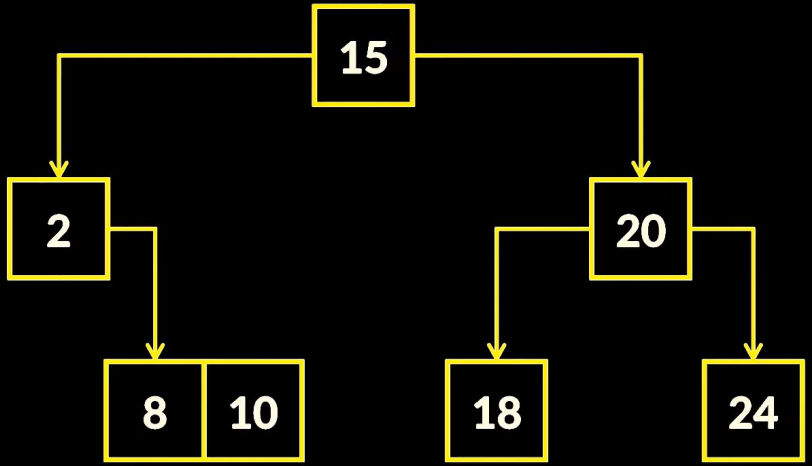

15
4
20
2
8
10
18
24

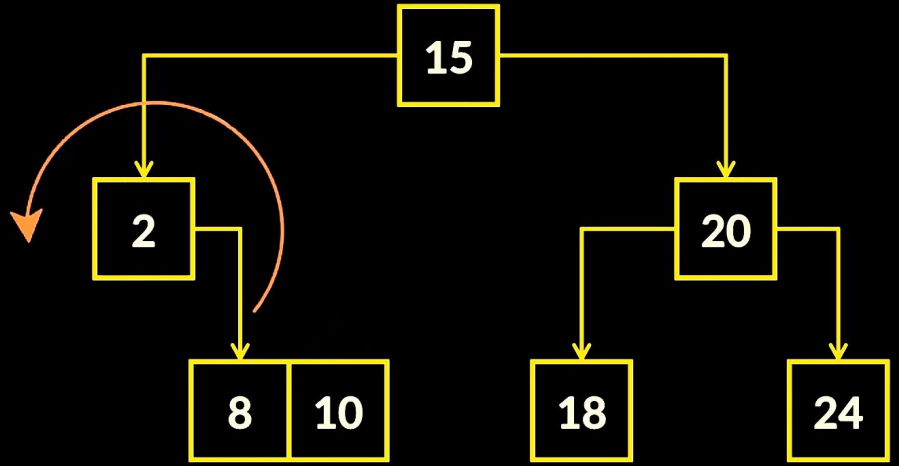

15
4
20
2
8
10
18
24

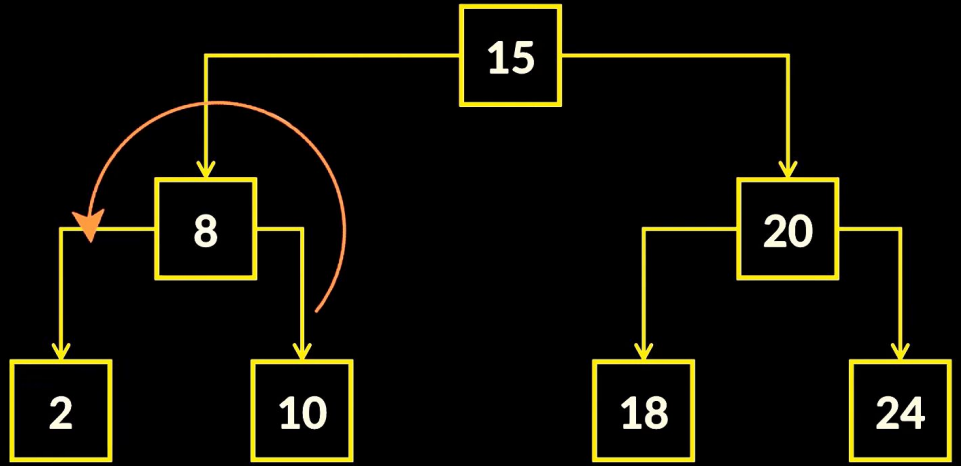

15
4
20
2
8
10
18
24

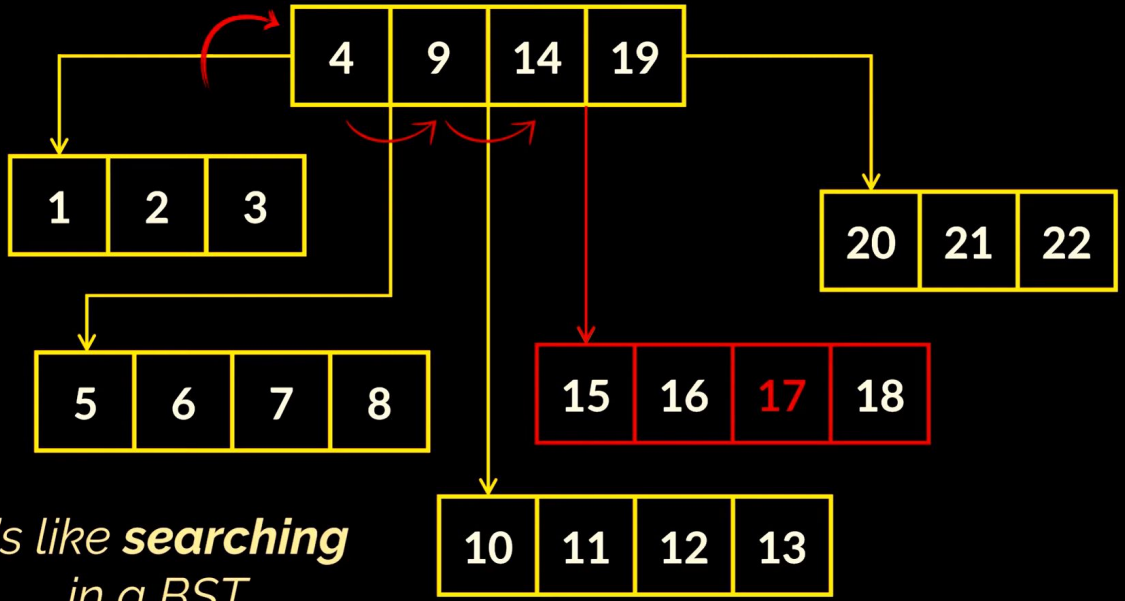

15
4
20
2
8
10
18
24


15
4
20
2
8
10
18
24

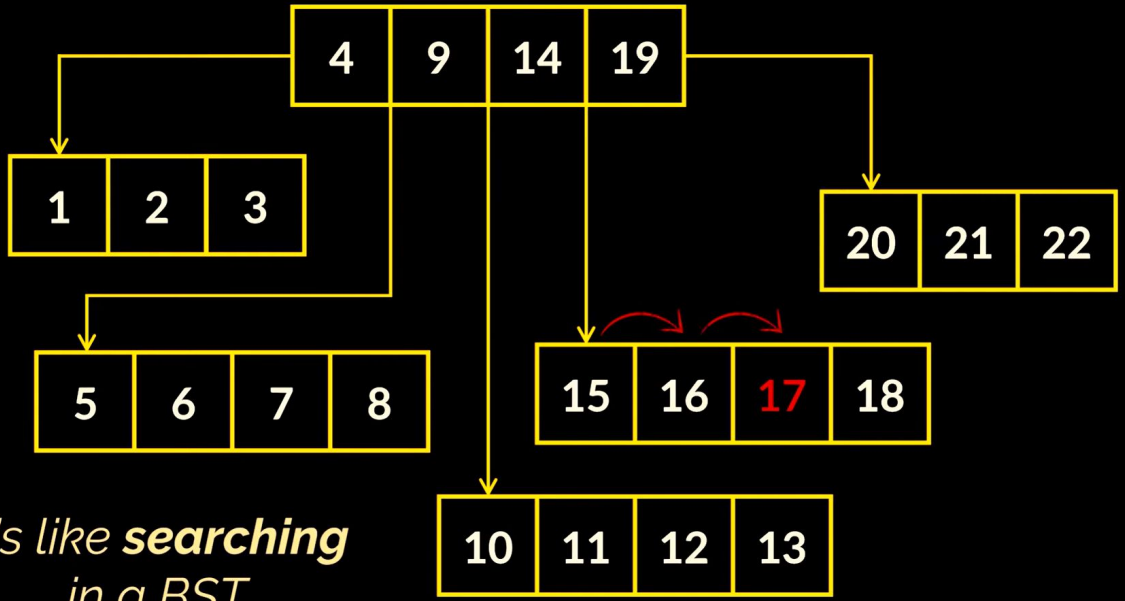

15
2
20
8
10
18
24


15
2
20
8
10
18
24

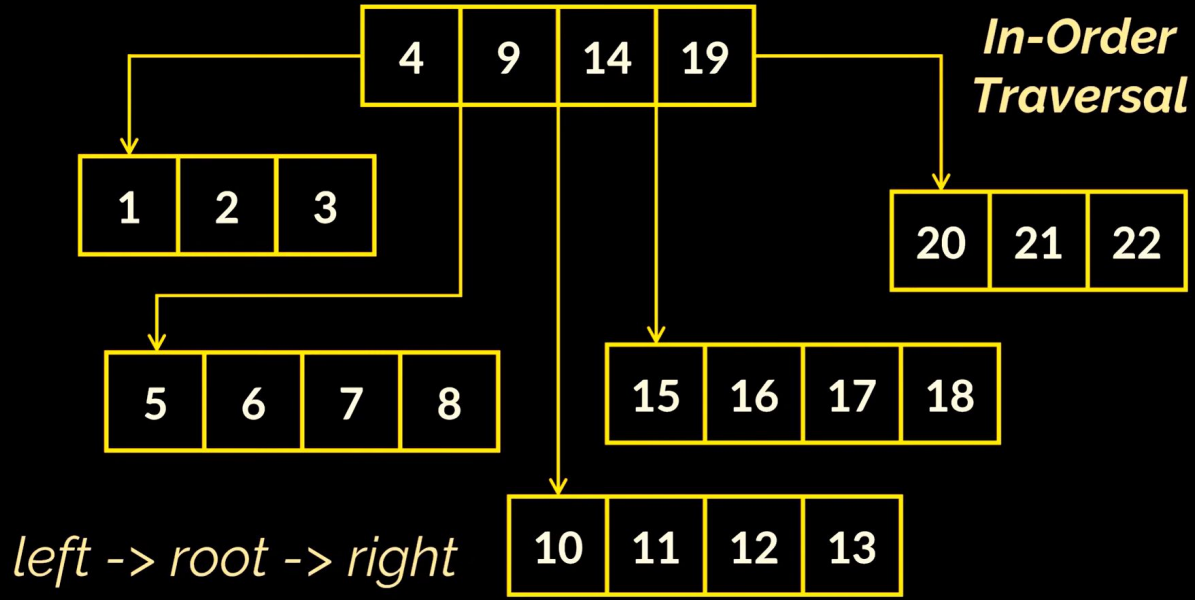

15
8
20
2
10
18
24

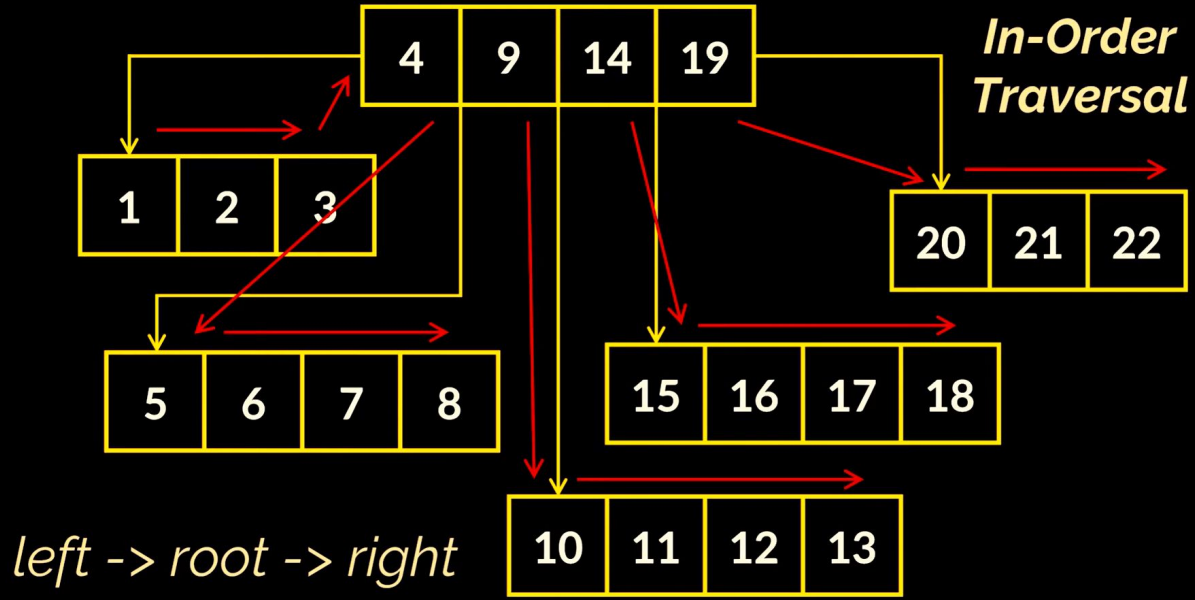


*it's like **searching** in a BST*



*it's like **searching** in a BST*


4 9 14 19
In-Order Traversal
1 2 3
20 21 22
5 6 7 8
15 16 17 18
10 11 12 13
left -> root -> right


4
9
14
19
1
2
3
5
6
7
8
10
11
12
13
15
16
17
18
20
21
22
In-Order Traversal
left -> root -> right