Binary Search Trees (BST)

Binary tree: Every child is either a left or a right child.

Every node has at most 2 children.

General ADT: Ordered Dictionary < key, value

Dictionary in which the key has a total ORDER.

Main operations: Insert, Remove, Find

Goal: $\boxed{\text{Quickly}}$ find an entry.

3

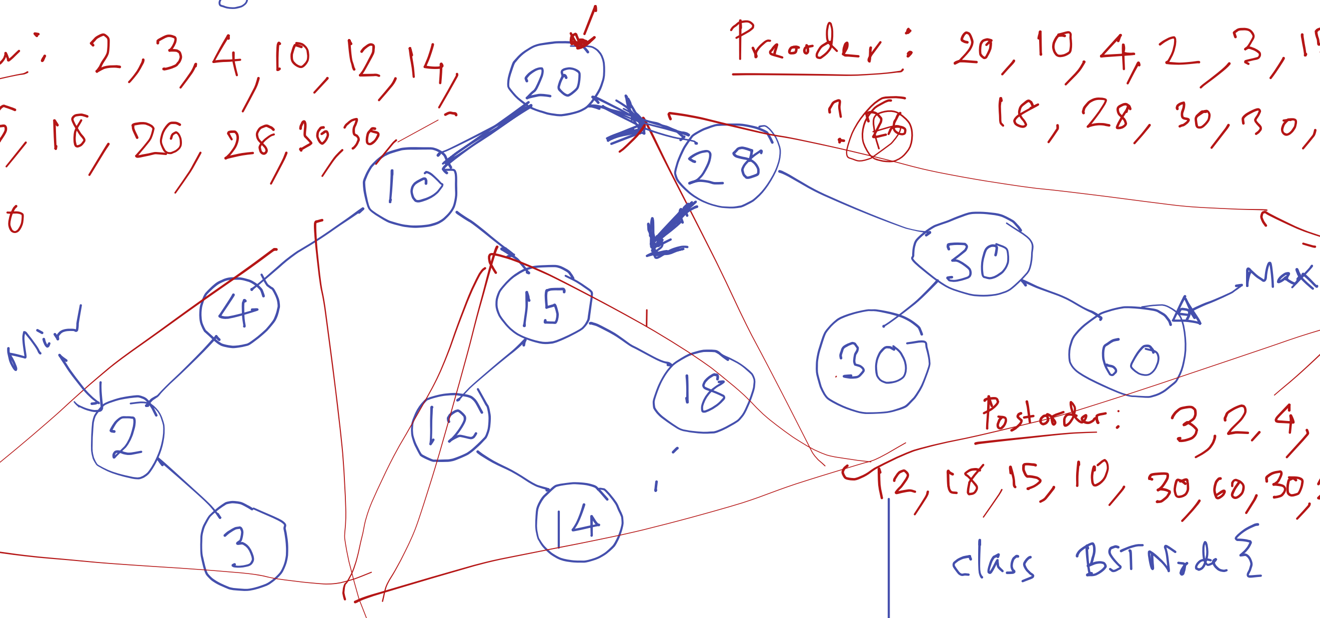
BST
* Easy to locate the Node with MIN as well as MAX key.
* Inexact match.

BST Invariant: For any node n, every key in the left subtree of n is $\leq$ n's key.
Every key in the right " " n is $\geq$ n's key.

SORTED

Inorder: 2, 3, 4, 10, 12, 14, 15, 18, 20, 28, 30, 30, 60

Preorder: 20, 10, 4, 2, 3, 15, 12, 14, 18, 28, 30, 30, 60



Postorder: 3, 2, 4, 14, 12, 18, 15, 10, 30, 60, 30, 28, 20

```
class BSTNode {
    Key k;
    Value v;
    BSTNode* left;
```

BST traversals:

Preorder: <root> - L - R

```
BSTNode :: preorder() {
    visit();
    if (left != nullptr)
        left -> preorder();
    if (right != nullptr)
        right -> preorder();
}
```

```
BSTNode * right;
}
```

Postorder : L - R - <Root>

```
BSTNode :: postorder() {

    if (left != nullptr)
        left -> preorder();
    if (right != nullptr)
        right -> preorder();
    visit();
}
```

Inorder : L - <Root> - R

```
BSTNode :: inorder() {
    if (left != nullptr)
        left -> preorder();
    visit();
    if (right != nullptr)
        right -> preorder();
}
```

Inorder traversal of a BST visits Nodes in SORTED ORDER.

Find, Insert, Remove

1. Node* find(Key k) — Exact / Inexact → Return NULL when not found

→ Follow the BST invariant.

$\boxed{\text{Inexact match}}$:

Find the smallest key $\geq k$

OR " " largest key $\leq k$.

$\left.\right\}$ $k_1 < k < k_2$

When searching in the tree for a key k that is <u>NOT</u> in the tree, we encounter BOTH

(i) the smallest key $\geq k$ AND

(ii) the largest key $\leq k$.

Last time the search went to the left-

Last time the search went to the right

(Before giving up on the search)

② Node * Min() → Min key → Repeatedly go to the <u>left</u> child until hitting the nullptr

Node * Max() → Max key.

Right most node before hitting the nullptr.

③ Node * insert(Key k, Value v)

Follow the same path through the tree as <u>find()</u> follows.

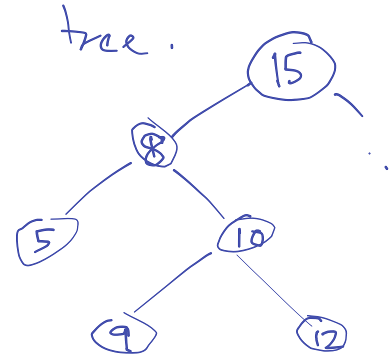
Exact match

When the search reaches NULL, replace NULL with the pointer to the new node constructed with (k, v).

(Duplicate) Put the <u>new node</u> in the left_subtree of the old node (with the same key)

④ Node * remove(Key k)

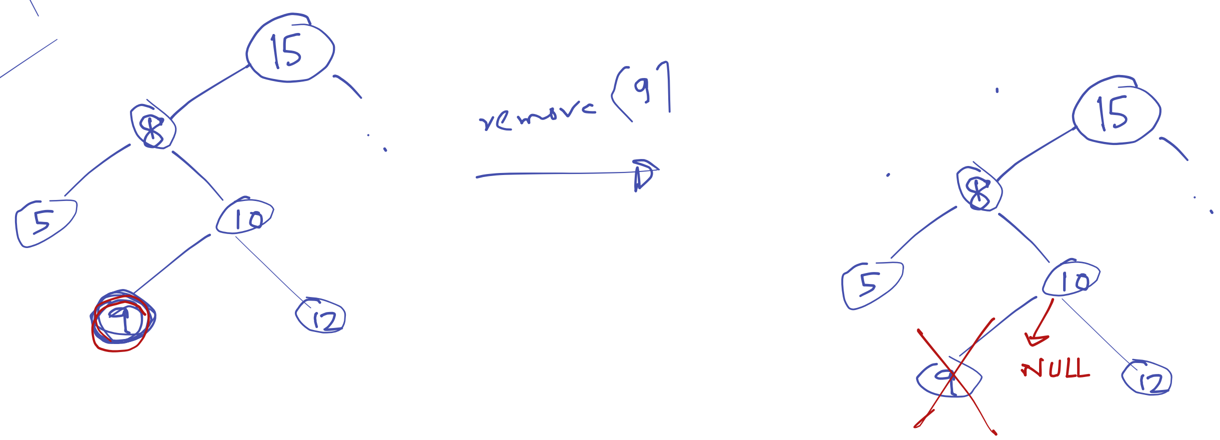
Find a node $\boxed{n}$ with key k.
Return NULL if k is not in the tree.



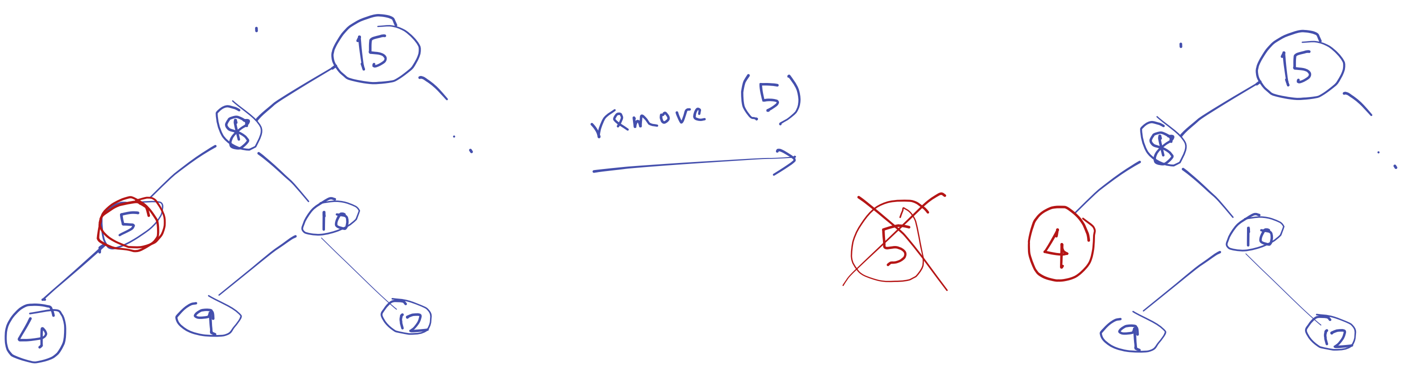
Cases:

? Node* parent

(i) n has NO children.
Detach it from the parent.



(ii) n has ONLY 1 child:
Move n's child up to take n's place.



(iii) n has 2 children.

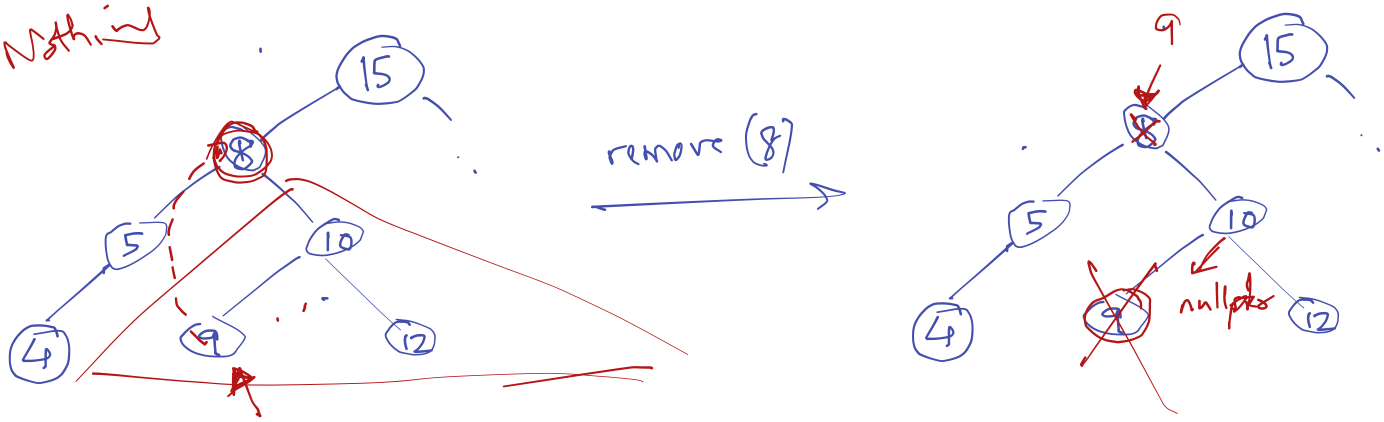
* Let $\boxed{n2}$ be a node in n's right subtree with the smallest key.

* Remove n2 (n2 has no left child & therefore easy to remove)

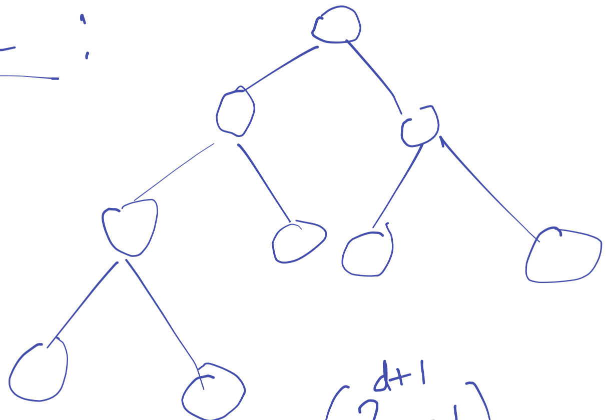
* Replace n's key with n2's key.

$\boxed{8 < 9}$

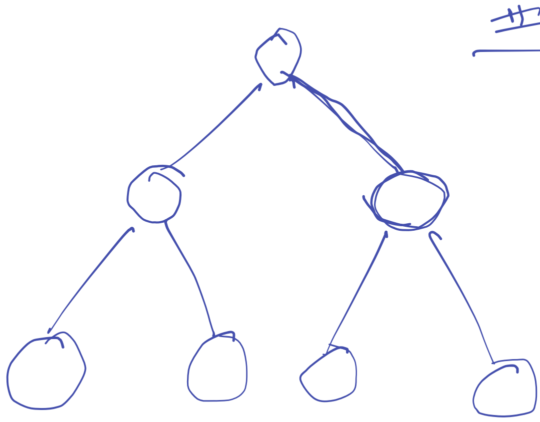
Nothing in between



Running time:



Perfectly Balanced BST



# Nodes $(2^{d+1}-1)$	# depth (d)
1	0
3	1
7	2

nodes $= (2^{d+1} - 1)$ for a perfectly balanced BST

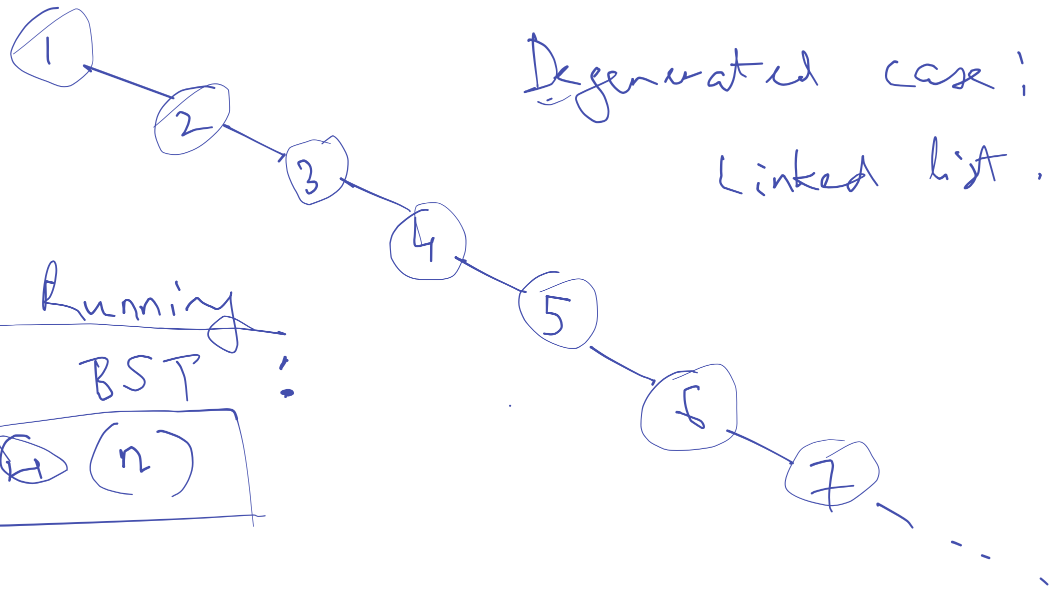
No node has depth $> (\log_2 n)$

Running time: Insert, find, remove, min, max

Proportional to the depth of the deepest node visited.

$\boxed{\Theta(\log_2 n)}$ ← Perfectly balanced BST.

1, 2, 3, 4, 5, 6, 7, ...



Degenerated case: Linked list.

Worst case Running time for BST:

$\boxed{\Theta(n)}$

BST: AVL trees, Red-black trees, Splay trees.

Splay tree.

Add 1 H/w

Best 6 of 7

Week → 12pm EDT → H/w, Lab.

X Late Token

↳ 1 day.

Max: 2 days