\[ \text{depth} = \# \text{ levels} = \]
\[ \# \text{ times we can double} \]
\[ = 0 \left( \log \left( \frac{n}{2} \right) \right) = O \left( \log(n) \right) \]

\[ \therefore \text{Runtime} = O(\text{depth}) = \text{worst-case: } O(n) \]
\[ \quad \text{avg-case: } O(\log(n)) \]

Similar for most operations

Next operation: insert

Similar to contains

```
(6) ex: insert(5)
     
5      6
2 4
1 3
```

\[ \text{Where should it go?} \]
\[ \text{Only one spot!} \]

Idea: Search for element (contains()), if not in tree, insert

Where? Child of last (non-null) node visited

will have another version of contains() that returns last non-null node visited

(modified-contains)
// One implementation approach
insert (Object e, BinaryNode t) // insert element e in tree with
// root t

// create new node
BinaryNode n = new BinaryNode (e)

// get last visited non-null node in search for e
BinaryNode p = modifiedContains (e, t)

// pseudocode for compareTo()
if (p.element < e) // add to left of p
p.left = n
else if (p.element > e) // add to right of p
p.right = n
else // p.element == e (duplicate)
p.addToduplicate (e)

3

insert (5, 6)

2

6

2

8

1

4

5,

3

p = 4 \Rightarrow 4 < 5, right

What about duplicates?

Not all implementations

every node has duplicate array

<add to BinaryNode class> do this!

Run Time: O(depth) = O(n) worst
O(log n) average
BST

remove (Object x)  // remove the node containing x

3 Cases: In all 3, need to:
- Search for node containing x, call it n
- Keep track of n's parent

case t: (Simplest) n is a leaf:

\[
\text{ex. remove (3)}
\]

\[
\begin{array}{c}
\text{2} \\
\text{8} \\
\text{1} \\
\text{4} \\
\text{3} \\
\end{array}
\]

\[
\text{4's left ptr. should point to null}
\]

\[
(\text{p = 4, p.left = null})
\]

⇒ Set n's parent's left or right ptr. to null

remove (Object x, BinaryNode t)  // Get the node containing x
1. n = get(x, t)
2. if (n.left == n.right == null) \{ 7
3. \} // Get last non-null node visited in search for x

\[
\begin{array}{c}
\text{p = modified contains (x, t)} \\
\end{array}
\]

4. if (p.right == n), p.right = null
5. else if (p.left == n), p.left = null

\[
\text{Also by remove(x)}
\]
(2) \( n \) has one child

\[ \text{ex: remove (4)} \]

\[ \text{Make (4)'s parent point to (4)'s child} \]

\[ \quad \]

\[ \quad \]

\( n \) is node containing \( x \)

\( n \) is node containing \( x \)

Make \( n \)'s parent's left/right ptr point to \( n \)'s child

(3) \( n \) has 2 children.

\( \text{(new example)} \)

\[ \text{ex: remove (2)} \]

\[ \]
- successor of \( n \) - lowest-valued node in right subtree of \( n \)

ex: successor(6) = 8

\[ \text{successor}(3) = 3 \]

Can replace \( n \) with either (we will use predecessor).

Let \( y = n \)'s predecessor.

(Main Idea: Replace \( n \) with \( y \)
Reorder using fewest \# of pointer changes)

2 sub-cases

- Case 3(a): \( y \)'s parent = \( n \)

\[
\begin{array}{c}
\text{\textbf{y = 1}} \\
\end{array}
\]

\[
\begin{array}{c}
\text{remove (2) \((n = 2)\)} \\
\end{array}
\]

Replace \( n \) with \( y \)
Case 3(b): Y's parent ≠ n.

1. Replace n with Y.
2. Make Y's old parent point to Y's (only) child

(Note if Y had another child, this child would be the pred of n.)

In all cases, check:
1. root = n
2. root = Y.
BST remove (more examples)

ex (1):

```
       17
       / \
      11   24
       /   /  \
        9   14  20
           /  /  \
          13 19 22
              /  \
             23
```

\[ y = 19 \]
\[ n = y's\ parent \]
\[ replace \ n \ w/ y \]
\[ \Rightarrow \]

```
remove(20)
(\ n = 20 \)
```

ex (2):

```
       17
       / \
      11   24
       /   /  \
        9   14  19
           /  /  \
          13 22 23
```

\[ y = 14 \]
\[ n! = y's\ parent \]

```
Step 1: replace \ n \ with \ y \]
\[ \Rightarrow \]

remove(17)
\[ n = 17 \]

Step 2: Make \ y's\ old\ parent\ point\ to \ y's\ child.\]
BST:
- contains \(\text{Time} = O(\text{depth})\)
- insert \(\text{worst} = O(n)\)
- remove \(\text{average} = O(\log n)\)

If we could guarantee depth is always \(O(\log n)\), could get worst-case \(O(\log n)\) times.

**Balanced Binary Search Tree (AVL Trees)**:
- BST with worst-case depth of \(O(\log n)\).
- Specifically: for every node \(n\), heights of left and right subtrees of \(n\) differ by at most 1.

**Ex**: AVL Tree  
Not AVL Tree (Why not? 6)

Operations are same as in BST, but after operation, tree is "rebalanced" to have depth \(O(\log n)\).
Since depth is worst-case $O(\log n)$, all operations take $O(\log n)$ in the worst-case.

Time = $O(\text{depth}) = O(\log n)$ (worst & avg).

How to "rebalance"?

ex: insert

```
  a
 / \  \
 b   c
```

"outer" insertion

"rebalance" at $a$ = deepest imbalanced node

= deepest node whose left + right subtrees have height difference $> 1$

Balance with Rotations (4 cases)

1. Insert occurs in right subtree of $a$'s right child.
2. "" "" "" left "" "" left "" "" outer "" insert
   Require single rotation
   (Above example is "outer").
3. "" "" "" left "" "" right ""
4. "" "" right "" "" left ""