We always return the parent node of the subtree. Why?

- Most of times it doesn't change.
- It does change when we perform the insertion.
Iterative insert

Insert (3, t).

Insert (7, t).

1. Find the parent of the new node for insert.
2. If node already exists, do nothing.
3. When inserting, figure out which child should the new node be.
Summary of runtimes. \( n \) is total # of items.

- **Sorted array**
  - Insert: \( O(n) \)
  - Search: \( O(\log n) \)

- **BST**
  - Insert: \( O(n) \)
  - Search: \( O(n) \)

- **Balanced BST**
  - Insert: \( O(\log n) \)
  - Search: \( O(\log n) \)

**Sorted array:**
- Search: use binary search \( O(\log n) \)
- Insert: shift all elements \( O(n) \)

**BST:** Depends on what the tree looks like.

**Advantages of BST:**
- Can search in \( O(\log n) \) if balanced.
- Insert/remove in \( O(\log n) \) if balanced.

**Disadvantages of BST:**
- Tricky to rebalance.
- If unbalanced, a linked list.

Search, insert, delete are all \( O(n) \).

Possible to implement a self-rebalancing tree. AVL tree DEMO.
Tree A [40, 20, 60, 10, 30, 50, 70]

- Size: 7
- Height: $3 \times \log_2 7$

Balanced tree. Short & flat.

For every node, sizes of subtrees are similar.

Runtime of insert $O(\log n)$. E.g., insert (55) go thru 3 nodes.

Tree B [10, 20, 30, 40, 50, 60, 70]

- Size: 7
- Height: 7

Unbalanced tree. Long & thin.

Very skewed to the right.

Runtime of insert $O(n)$. E.g., insert (80) go thru 7 nodes.
Size node augmentation
- store size of subtree in each node.
- rebalance a tree.
- get node given an index. select(k, t)
  - index, starts from 0.

Insert into the tree. What do you need to update?

Index 3 = # of nodes in left subtree.
Index < 3 left subtree same index.
Index > 3 right subtree. Index = me - left size.

select (2, t). left size = 3. 2 < 3.
select (2, node 20) left size = 1. 2 > 1.
select (0, node 30) left size = 0. 0 == 0.
  2-1-1 return 30;
Array-based trees:

advantages:
- few pointers
- easily access parent

disadvantages:
- waste space if tree not complete
- self-balancing can be awkward.

---

e.g. the heap data structure often a complete tree.

---

![Tree Diagram]

Level 1
- 40

Level 2
- 20
- 30
- 50

Level 3
- 10
- 30
- 60

Choose a sentinel value.