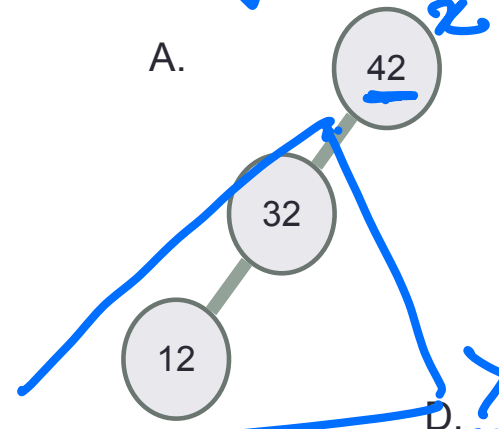
BINARY SEARCH TREE

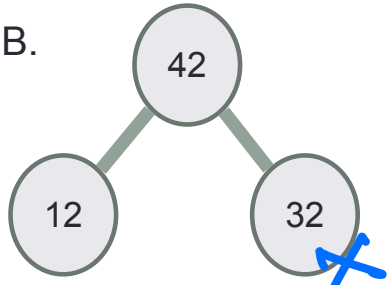
Problem Solving with Computers-II

Which of the following is/are a binary search tree?

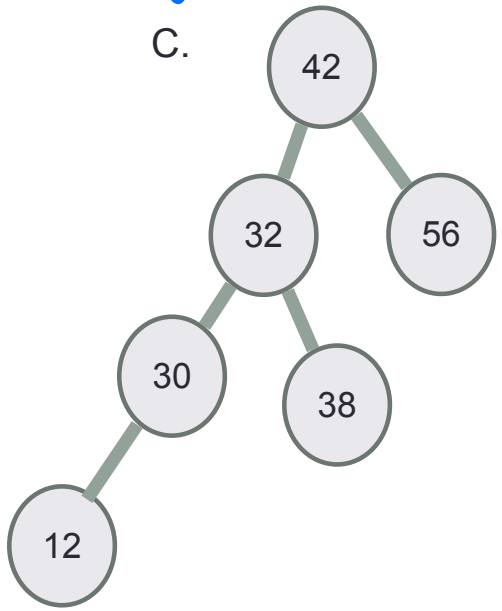
A.



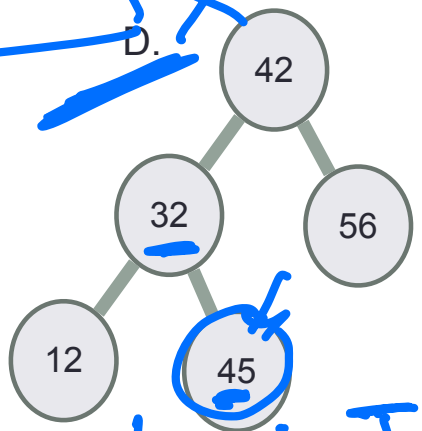
B.



C.



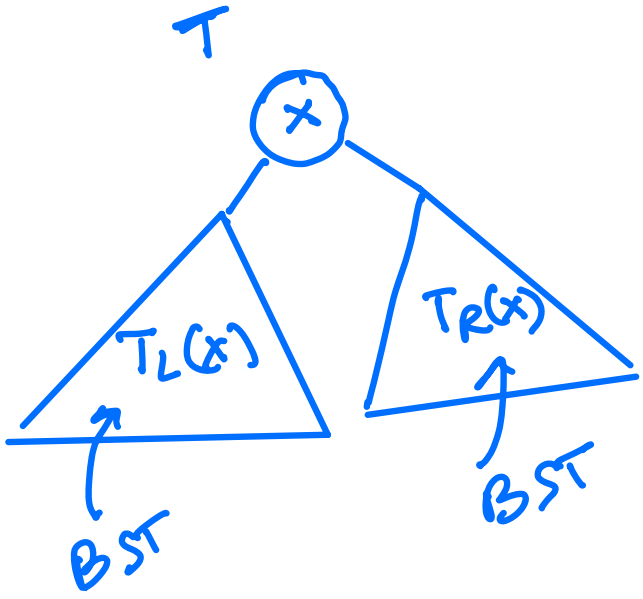
D.



E. More than one of these

BST property: keys in $T_L(x)$ < key(x) < keys in $T_R(x)$

① BST has a recursive structure (ux to write recursive algos) today!

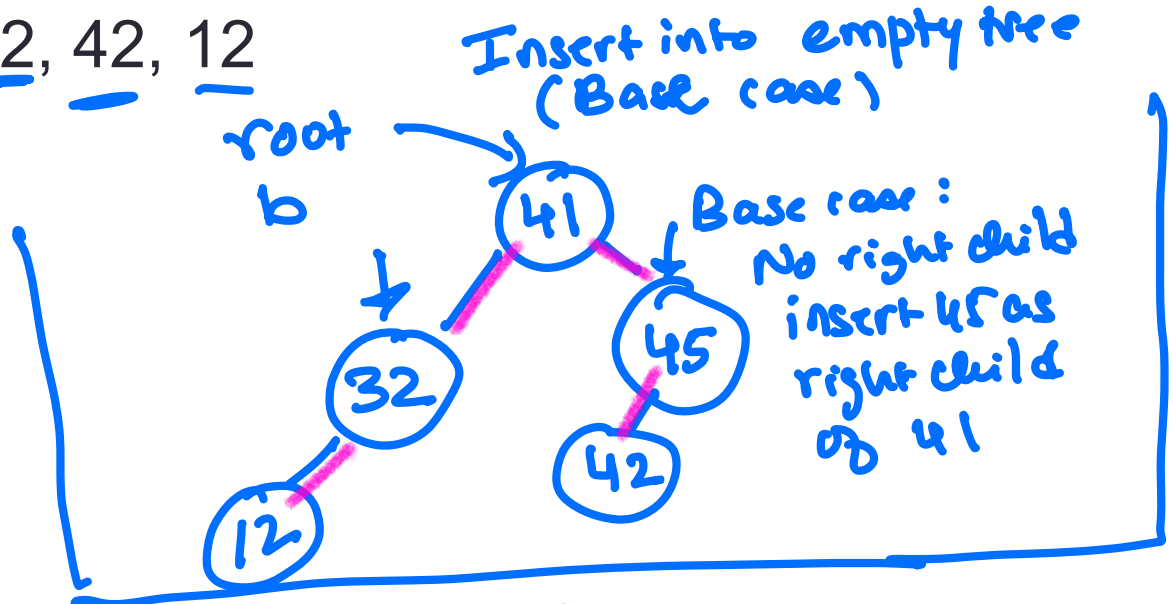


② How do I create a BST from scratch? Repeated insert operations

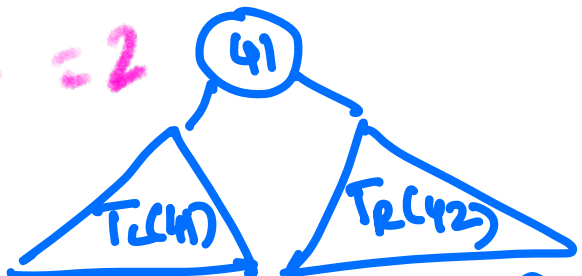
Goal: To articulate the algorithm for inserting a key into a BST

Insert keys: 41, 45, 32, 42, 12

bst b
b.insert(41)
b.insert(45)
b.insert(32)
b.insert(42)
b.insert(12)



Height = 2

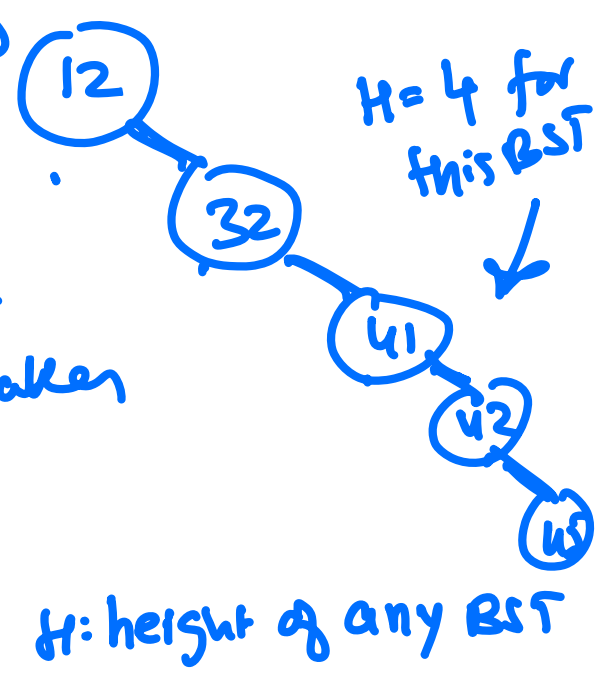


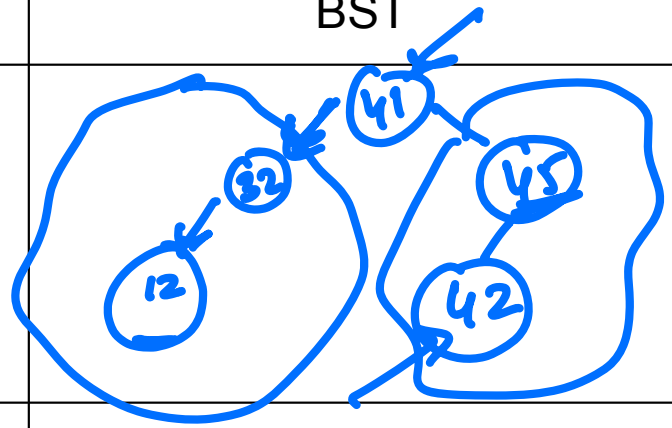
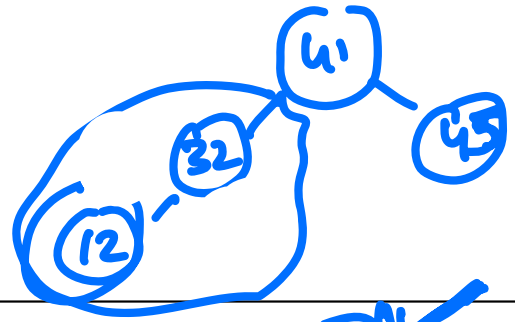
Write recursive insert algo. Click D for Done

Insert key in sorted order: 12, 32, 41, 42, 45
What is the height of the tree?
What is the best/worst case running time for insert?

Worst case running time for search, insert (and many other BST operations) is O(H)

If the BST is skewed to look like a linked list, then H = n-1, which makes the worst case running time O(n)



Traversal algo	BST	Output
inorder(r: pointer to current node): if r is null: return inorder(r->left) process r (e.g., print r->val) inorder(r->right)		12, 32, 41, 42, 45 $T_L(41)$ 41, $T_R(41)$
postorder(r: pointer to current node): if r is null: return postorder(r->left) postorder(r->right) process r (e.g., print r->val)		12, 32, 45, 41 $T_L(41)$ $T_R(41)$ 41
preorder(r: pointer to current node): if r is null: return process r (e.g., print r->val) preorder(r->left) preorder(r->right)		41, 32, 12, 45 41 $T_L(41)$ $T_R(41)$

Post-order traversal: use to recursively clear the tree!

postorder(r : pointer to current node):
    if r is null, return
    postorder(r->left)
    postorder(r->right)
    process r (e.g., print r->val)

```
int bst::getHeight(Node *r) const{
    if (!r)
        return -1;
    int hleft = getHeight(r->left);
    int hright = getHeight(r->right);
    return max(hleft, hright) + 1;
}
```

Why would preorder not work for clear?

```
void bst::clear(Node *r){
    if (!r)
        return;
    clear(r->left);
    clear(r->right);
    delete r;
}
```

post order

When would you use each traversal and why?

Inorder: useful when. . . printing in sorted order

Postorder: useful when. . . clearing the tree

Preorder: useful when. . . copy constructor

Pre-order traversal Game!

1. **Draw a BST:** Individually, draw a BST with 6 distinct keys (e.g., integers 1–10). Your
   Your BST (draw secretly):

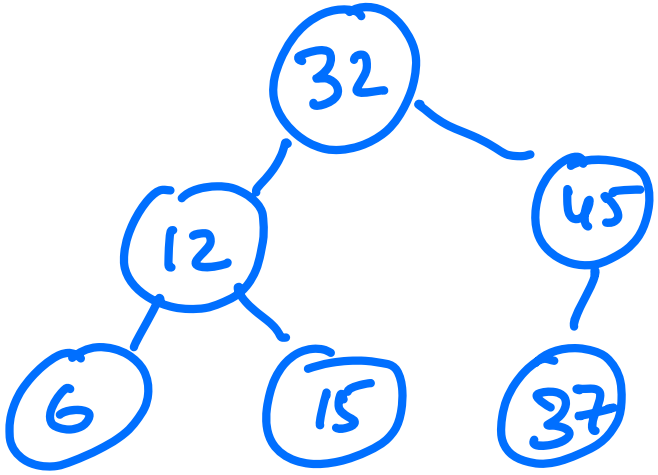
2. Trace Preorder Sequence: Trace the preorder traversal (root, left, right) of your BST and write the sequence of node values.

Your Preorder sequence:_____
(read the sequence to your partner, don't show them your BST diagram!)

Pre-order traversal Game!

3. Write the Pre-order sequence you received: 32, 12, 6, 15, 45, 37

4. Reconstruct the BST: Using your partner's sequence, rebuild their BST by inserting nodes in the given order, respecting BST properties.



5. Compare trees.

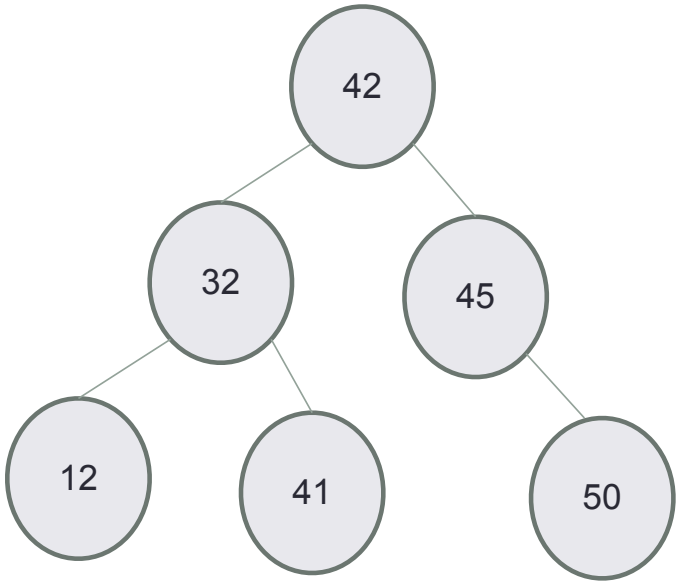
Leetcode problem

Given the `root` of a binary tree, return the preorder traversal of its nodes' values in a vector.

```
class Solution {
  public:
    vector<int> preorderTraversal(TreeNode* root) {}
};
```

https://leetcode.com/problems/binary-tree-preorder-traversal/description/?envType=problem-list-v2&envId=depth-first-search

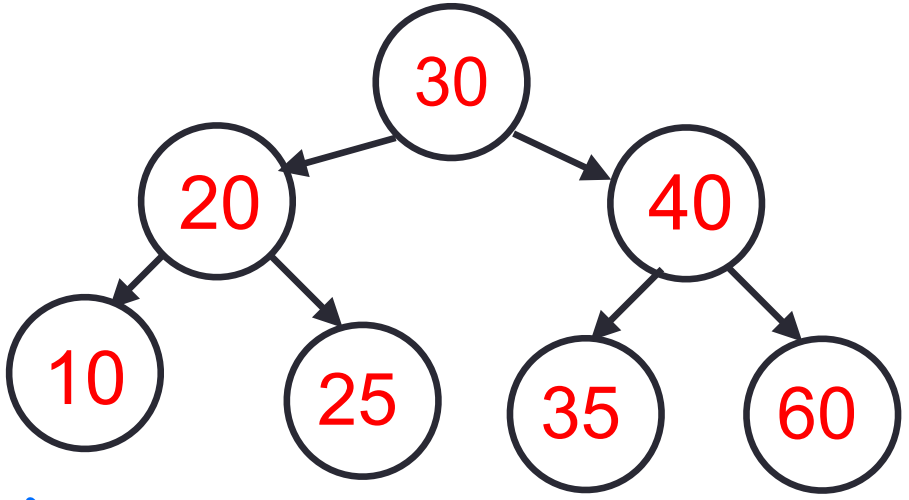
Interview question: Write a function to extract BST keys into a vector. The extraction order must allow reconstructing the exact same tree structure by sequential insertion.



https://leetcode.com/problems/binary-tree-preorder-traversal/description/?envType=problem-list-v2&envId=depth-first-search

What does this code do?

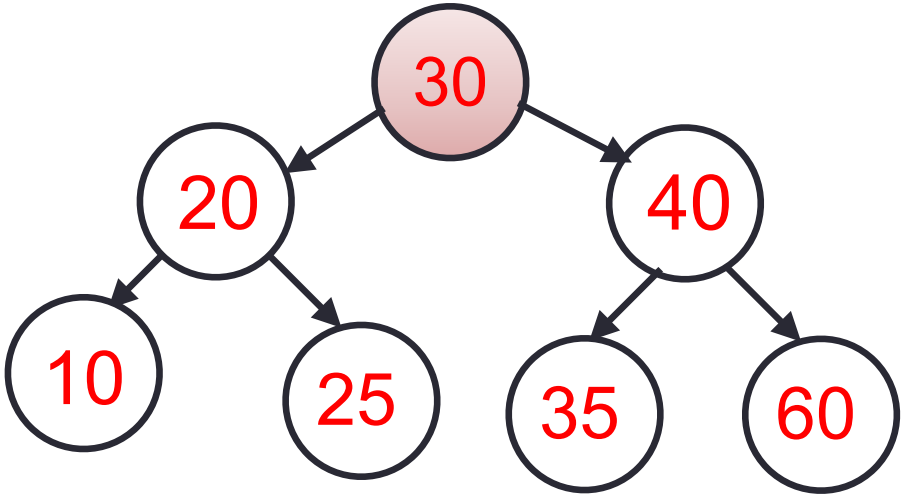
```
Node* r = b.min(root);
while(r){
    cout << r->data << " ";
    r = b.successor(r);
}
```



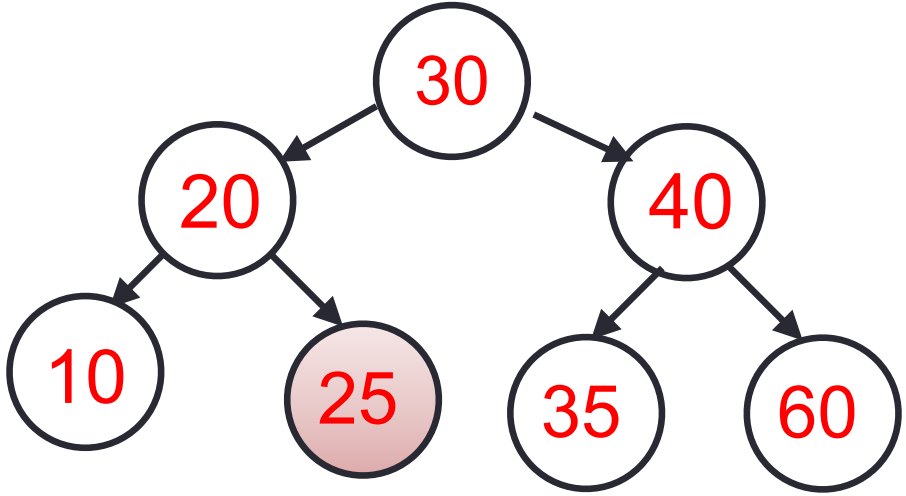
Home work: Write the algo for min() & successor()

- What is the successor of 30?
- What is the successor of 25?

Successor: Next largest element

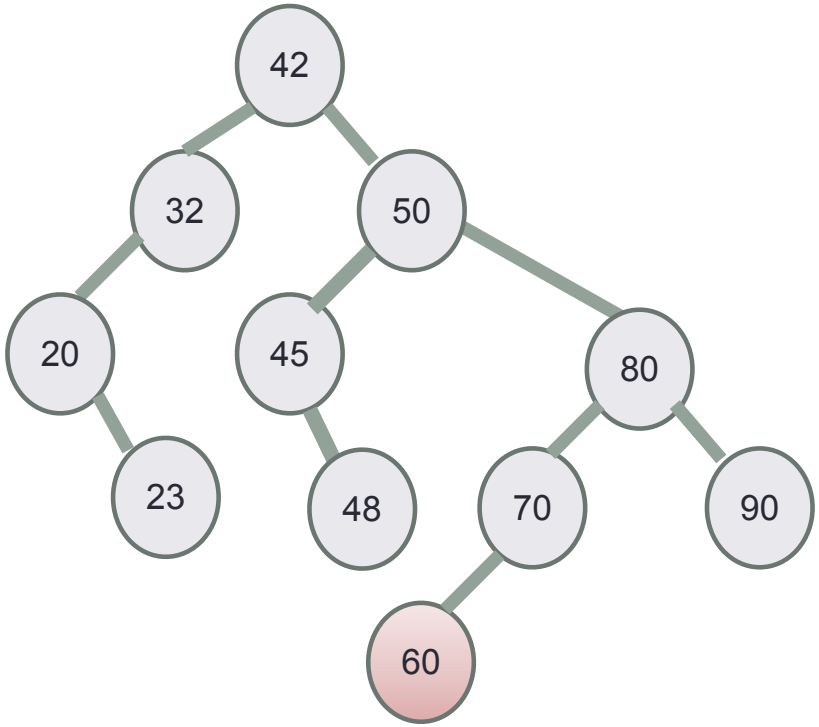


- Case 1: Node (n) has a right child

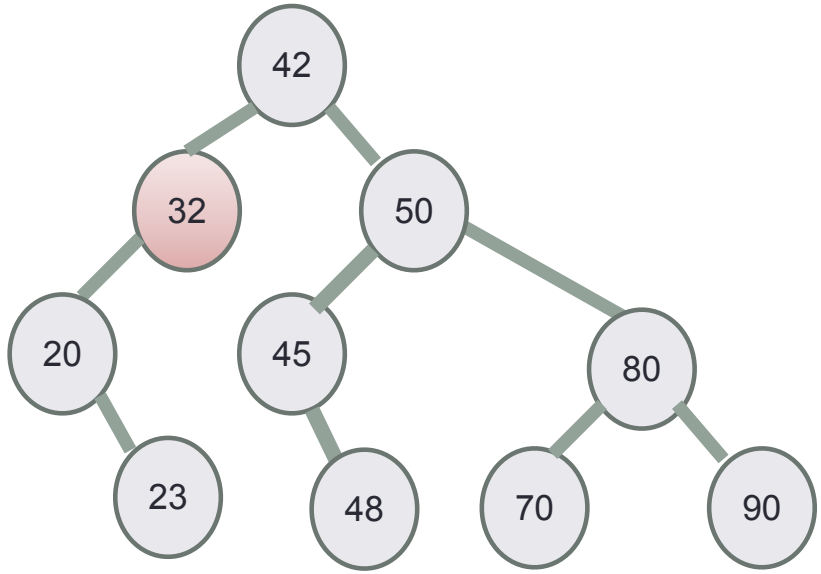


- Case 2: Node (n) has no right child

Delete a specific node

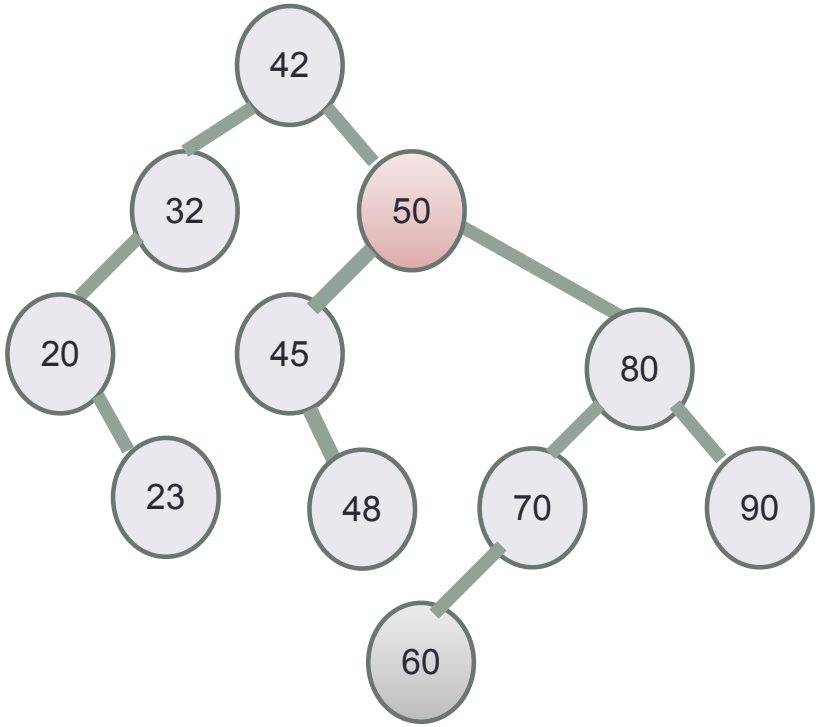


- Case 1: Node is a leaf node
  - Set parent's (left/right) child pointer to null
  - Delete the node



- Case 2: Node has one child (left/right)
  - Replace the node by its only child

Delete a specific node



- **Case 3: Node has two children**

BST ADT

Operations
Search
Insert
Min
Max
Successor
Predecessor
Delete
Print elements In order Preorder, Post order