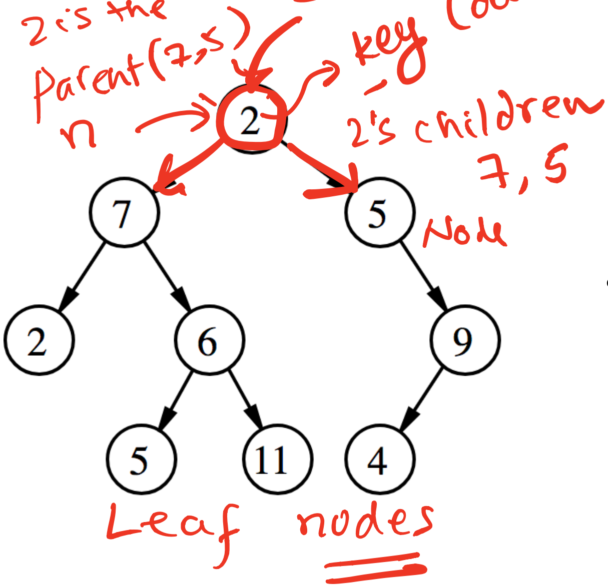
# BINARY SEARCH TREES

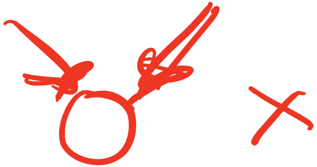
Problem Solving with Computers-II

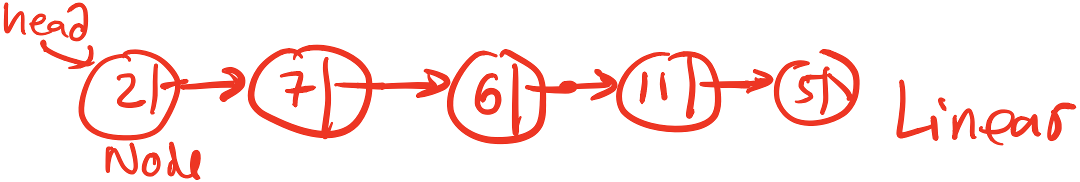
# Trees




A tree has following general properties:

- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*



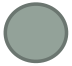


# Which of the following is/are a tree?

Binary Tree

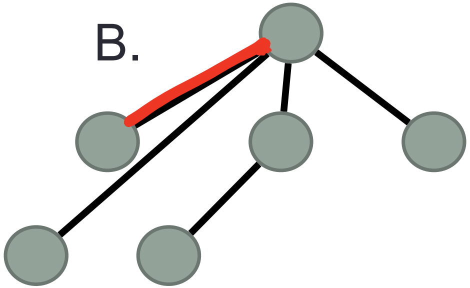
Tree where every node can have at most two children

A.

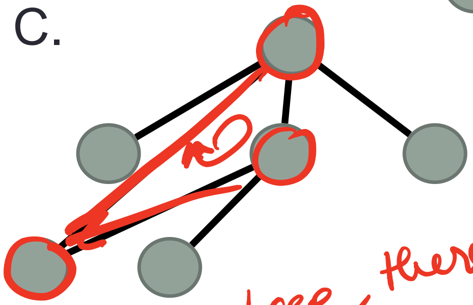


(Single node tree)

B.



C.

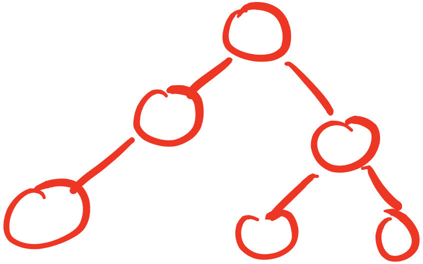


not a tree, there is a loop

D. A & B

E. All of A-C

# Binary Search Trees



- What are the operations supported?

  Same operations as linkedlist or array :

  Sorted array + fast insert & delete.

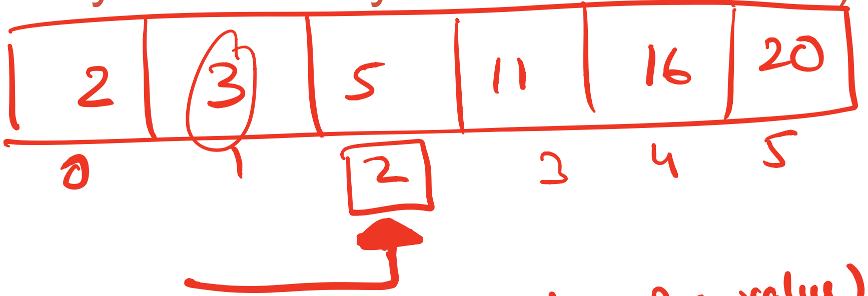
- What are the running times of these operations?

  Build intuition → formalize complexity next week.

- How do you implement the BST i.e. operations supported by it?

# Operations supported by Sorted arrays and Binary Search Trees (BST)

Sorted array

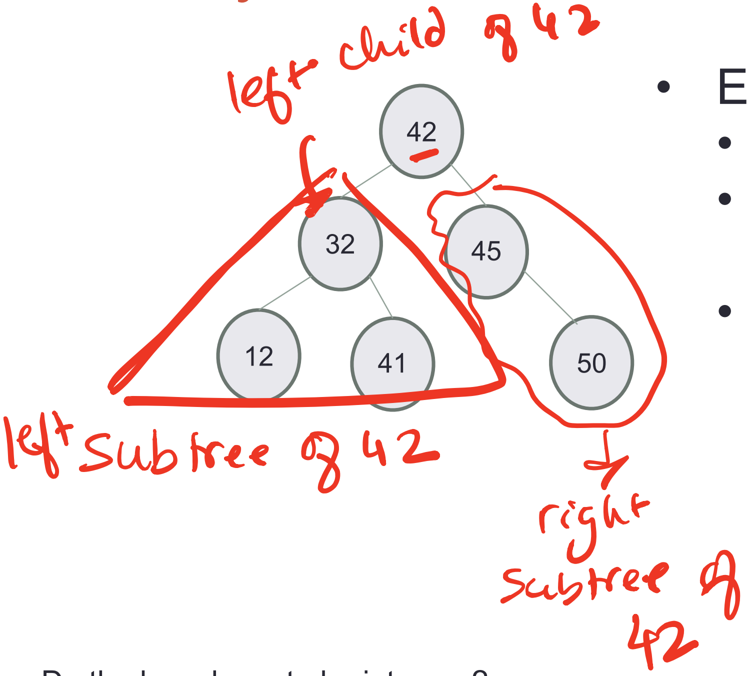



| Operations |
|---|
| Min |
| Max |
| Successor |
| Predecessor |
| Search |
| Insert |
| Delete |
| Print elements in order |

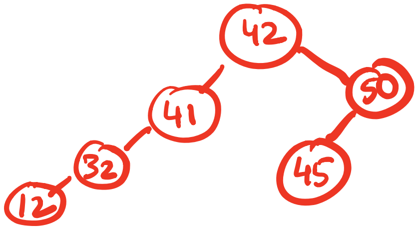
Successor → given a value (or index of a value) find the next largest value

Insert, Delete → slower in sorted arrays.

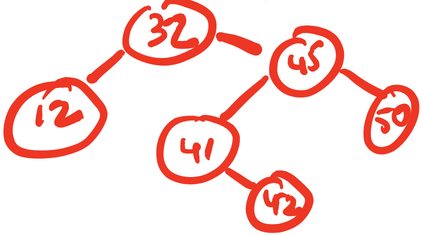
# Binary Search Tree – What is it?




- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property



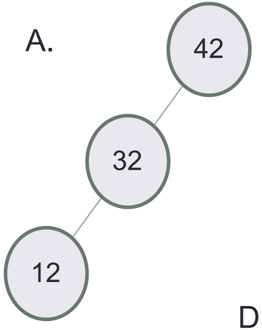

For any node,
Keys in node's left subtree <= Node's key
Node's key < Keys in node's right subtree



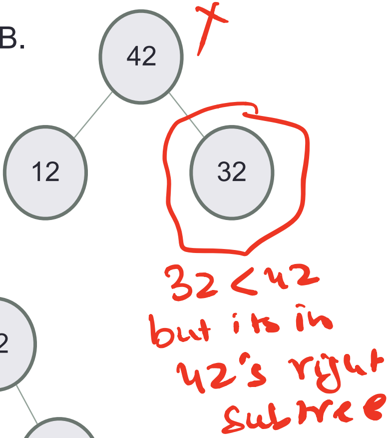

Do the keys have to be integers?

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

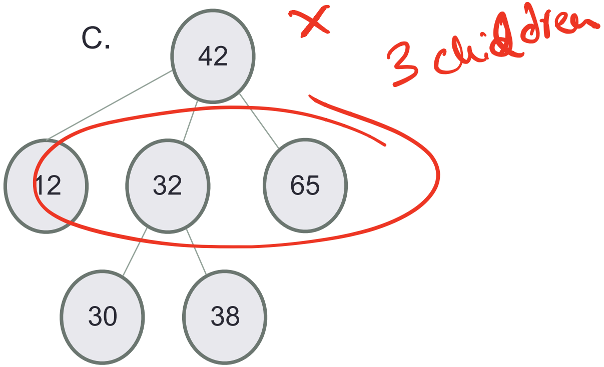
A.



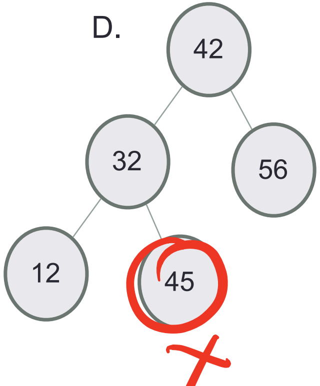

B.




C.

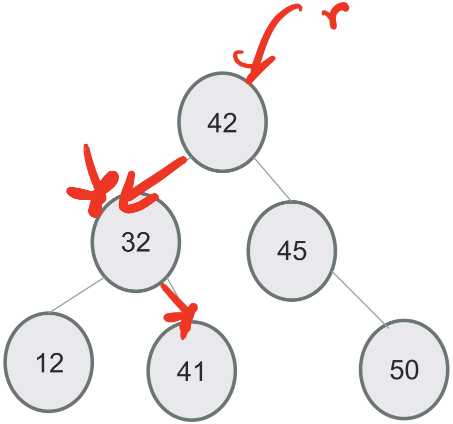



D.



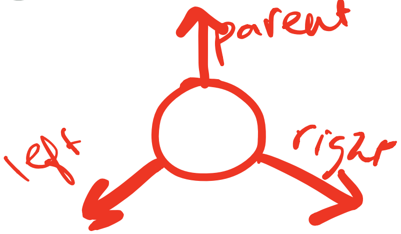

E. More than one of these

# BSTs allow efficient search!




- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x

Value 41




```
class Node {
public:
    int data;
    Node* left;
    Node* right;
    Node* parent;
}
```

**Search for 41, then search for 53**

# A node in a BST

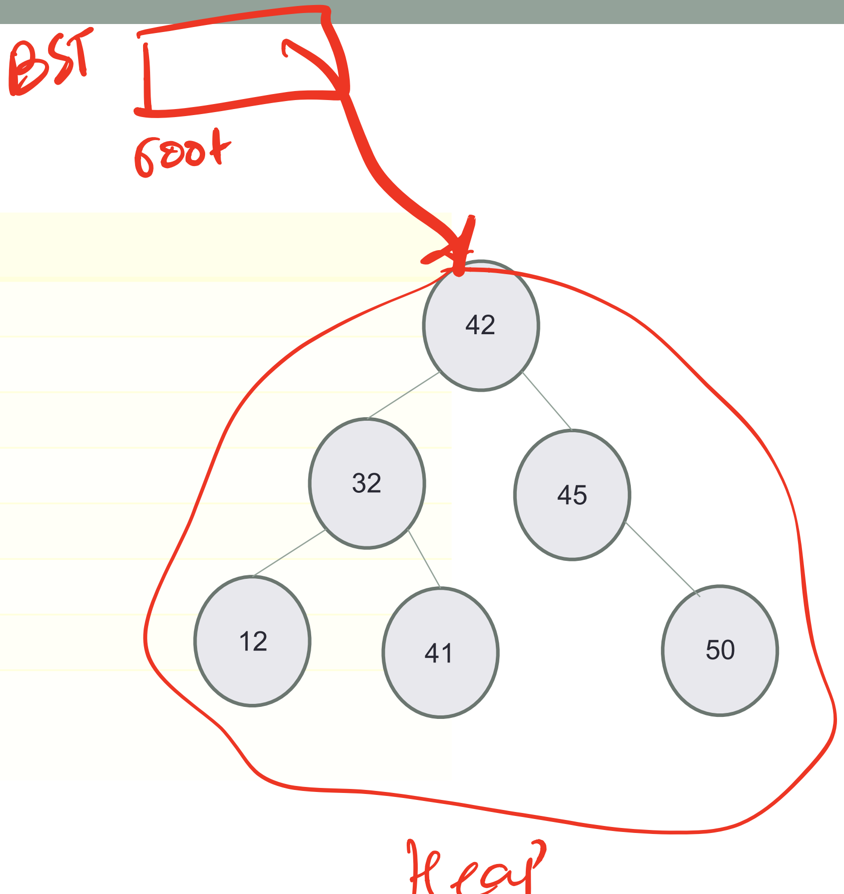
```
class BSTNode {

public:
  BSTNode* left;
  BSTNode* right;
  BSTNode* parent;
  int const data;

  BSTNode( const int & d ) : data(d) {
    left = right = parent = 0; nullptr;
  }
};
```

# Define the BST ADT

| Operations |
|---|
| Search |
| Insert |
| Min |
| Max |
| Successor |
| Predecessor |
| Delete |
| Print elements in order |

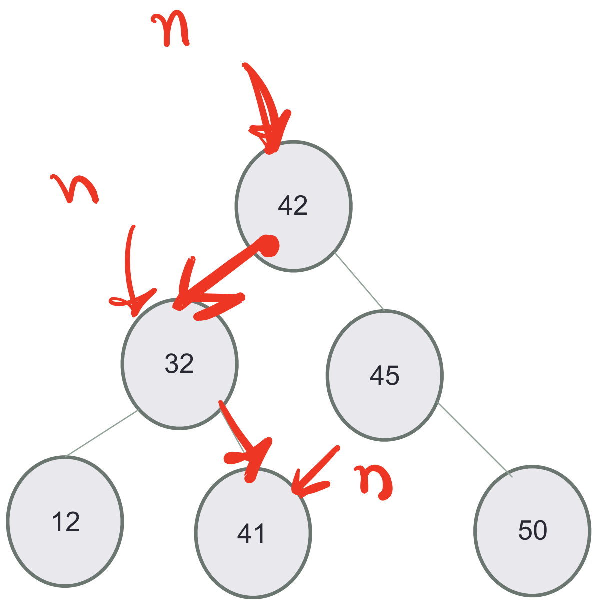


# Traversing down the tree

- Suppose n is a pointer to the root. What is the output of the following code:

```
n = n->left;
n = n->right;
cout<<n->data<<endl;
```

A. 42
B. 32
C. 12
D. 41
E. Segfault



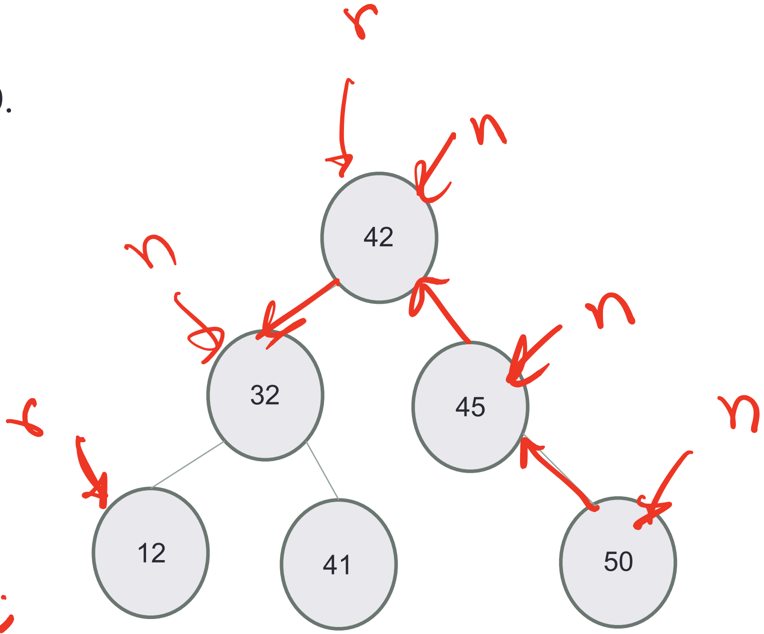
# Traversing up the tree

- Suppose n is a pointer to the node with value 50.
- What is the output of the following code:

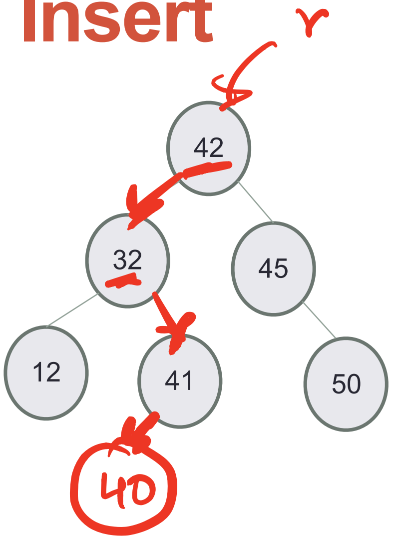
```
n = n->parent;
n = n->parent;
n = n->left;
cout<<n->data<<endl;
```

A. 42
B. 32
C. 12
D. 45
E. Segfault

```
while(r&&r->left)
    r= r->left;
```



# Insert




- Insert 40
- Search for the key
- Insert at the spot you expected to find it

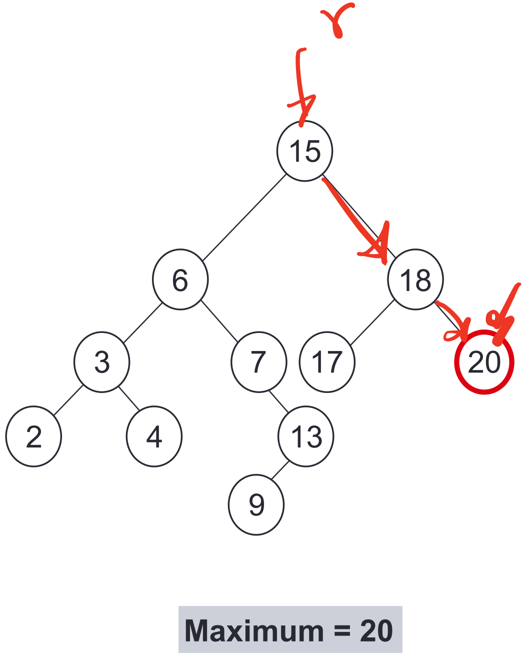
# Max

**Goal**: find the maximum key value in a BST

Following right child pointers from the root, until a leaf node is encountered. The least node has the max value

**Alg:** `int BST::max()`

keep going right:



# Min

**Goal**: find the minimum key value in a BST

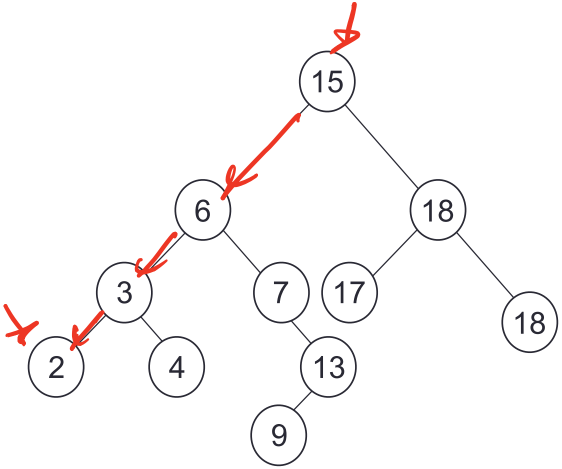
Start at the root.

Follow __left__ child pointers from the root, until a leaf node is encountered

Leaf node has the min key value

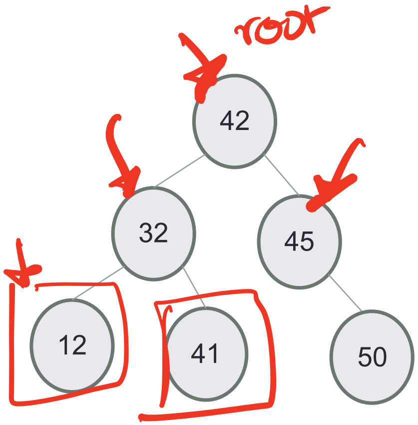
**Alg:** `int BST::min()`

keep going left




Min = ?

# In order traversal: print elements in sorted order




Algorithm Inorder(tree)

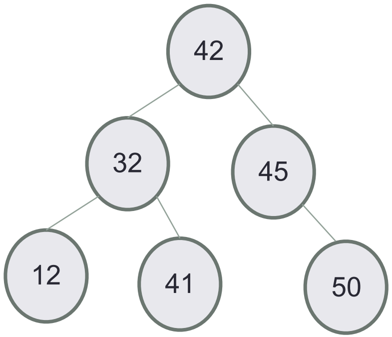
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root. ← print the root
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

```
void Inorder( Node * r) {
    if (!r) return
    Inorder(r->left);
    cout<< r->data;
    Inorder(r->right);
```

12 32 41 42 45 50

2

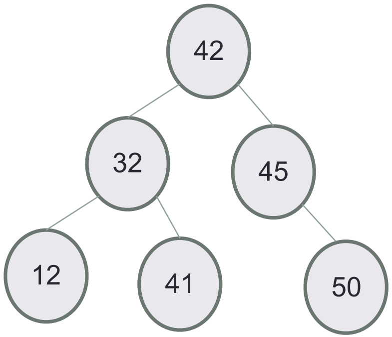
# Pre-order traversal: nice way to linearize your tree!




Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

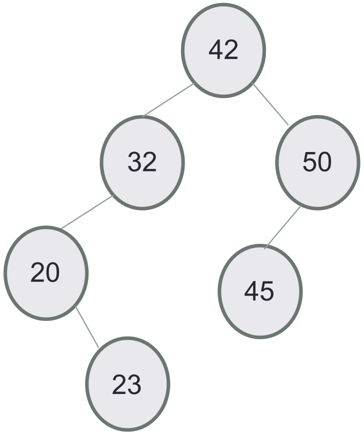
# Post-order traversal: use in recursive destructors!




Algorithm Postorder(tree)

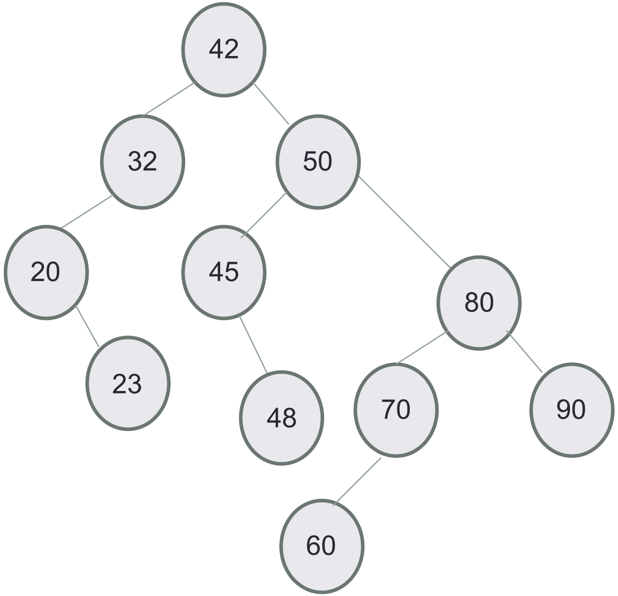
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

# Predecessor: Next smallest element



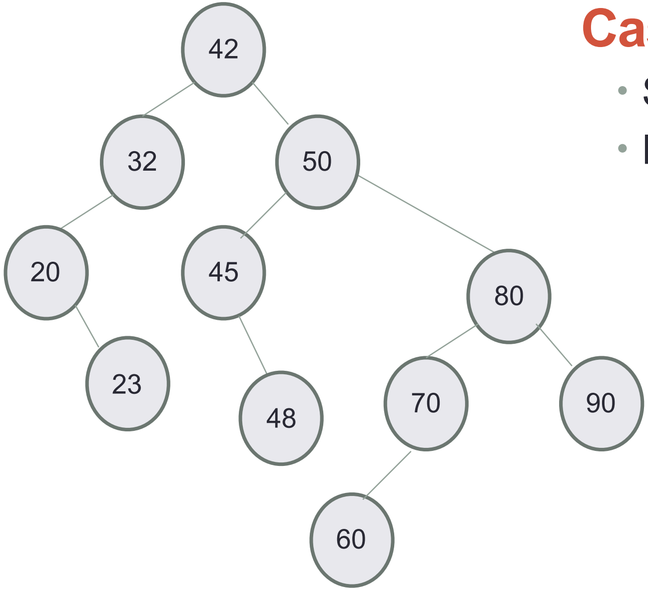

- What is the predecessor of 32?
- What is the predecessor of 45?

# Successor: Next largest element




- What is the successor of 45?
- What is the successor of 50?
- What is the successor of 60?

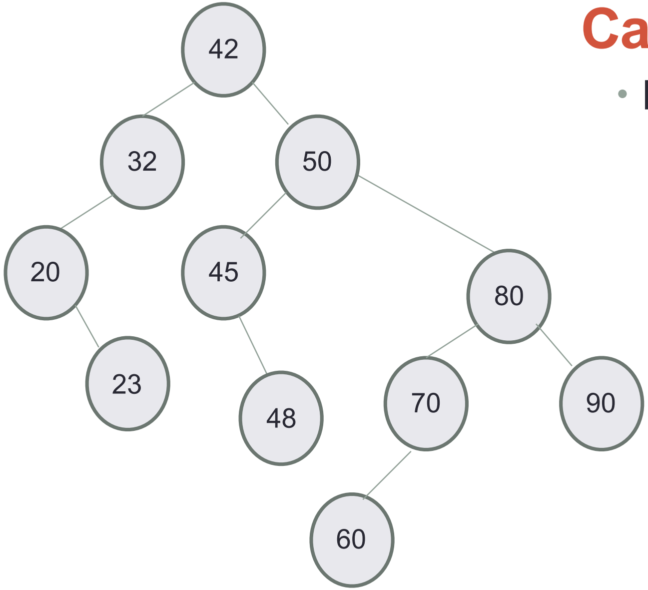
# Delete: Case 1




## Case 1: Node is a leaf node

- Set parent’s (left/right) child pointer to null
- Delete the node

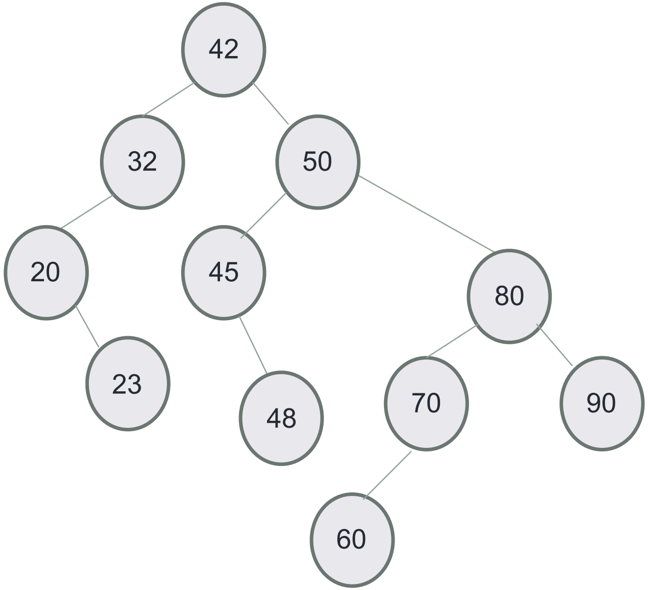
# Delete: Case 2




## Case 2 Node has only one child

- Replace the node by its only child

# Delete: Case 3




## Case 3 Node has two children

- Can we still replace the node by one of its children? Why or Why not?