BINARY SEARCH TREES

Problem Solving with Computers-II

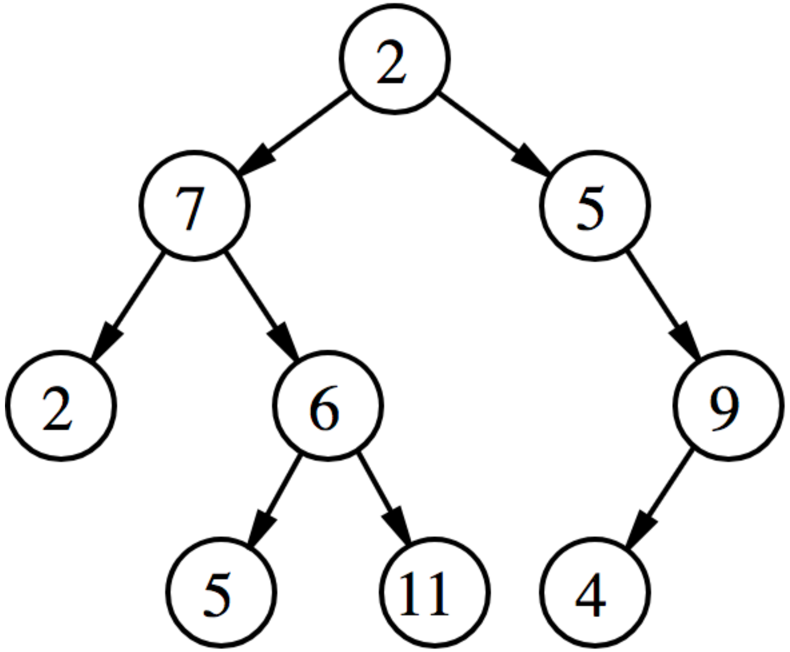
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo														hi

Trees

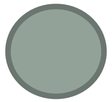


A tree has following general properties:

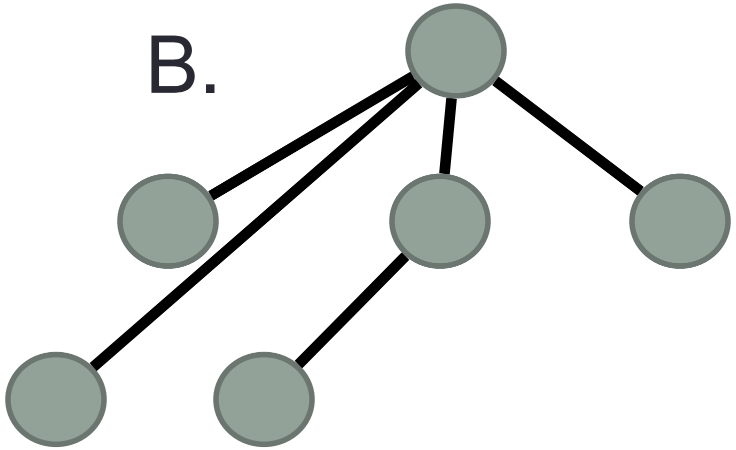
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*

Which of the following is/are a tree?

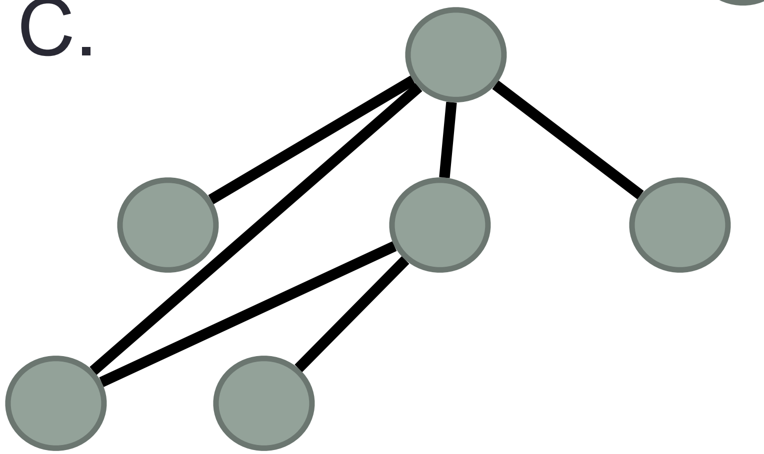
A.



B.



C.



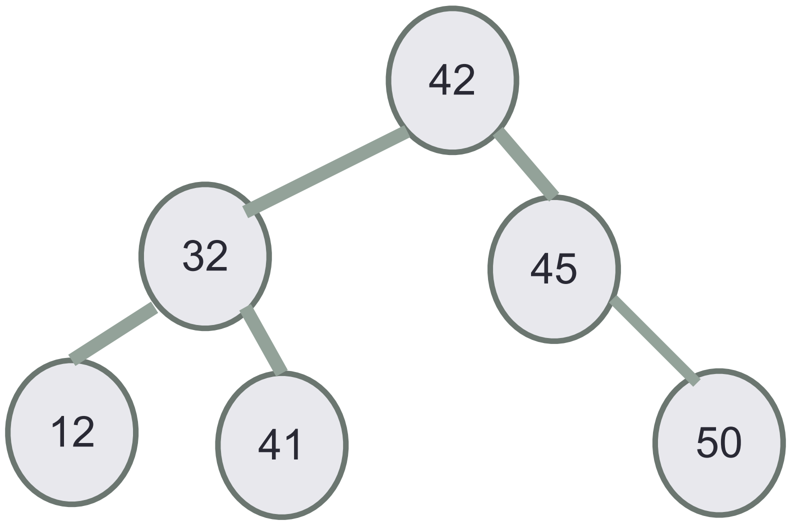
D. A & B

E. All of A-C

Binary Search Trees

- What are the operations supported?
- What are the running times of these operations?
- How do you implement the BST i.e. operations supported by it?

Binary Search Tree – What is it?



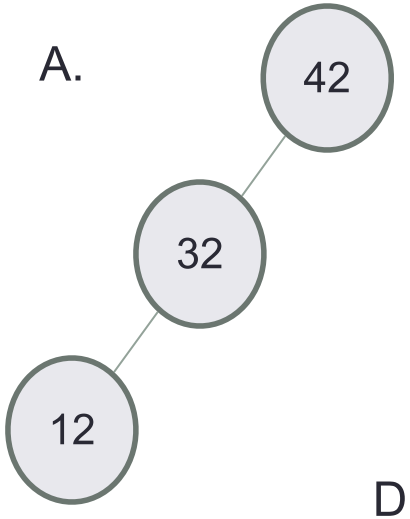
- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property

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

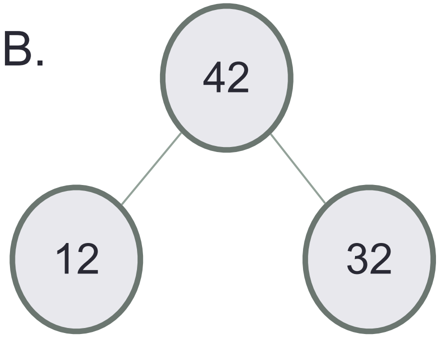
Do the keys have to be integers?

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

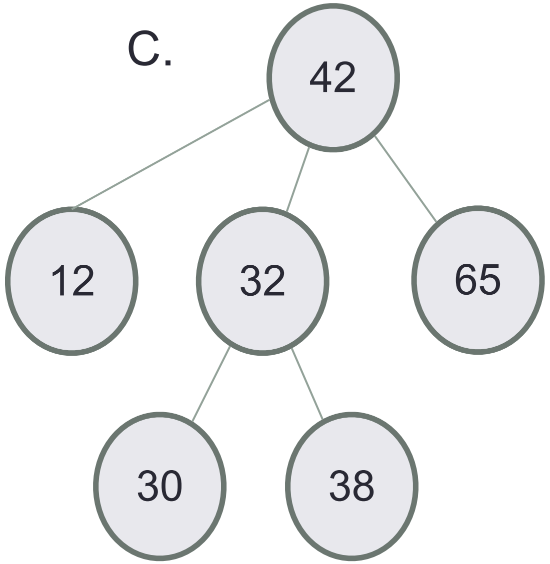
A.



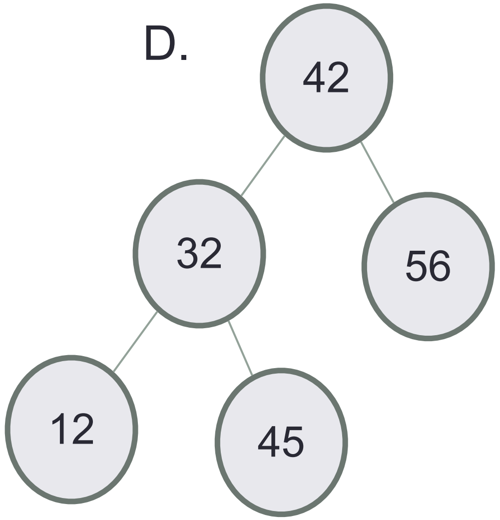
B.



C.

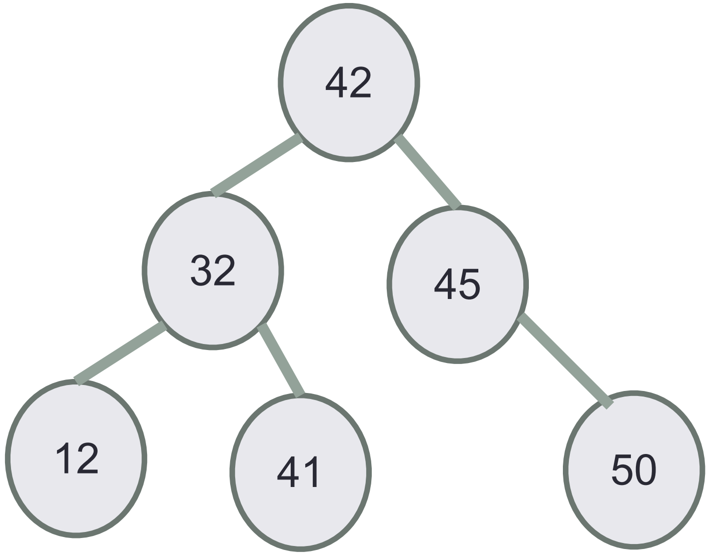


D.



E. More than one of these

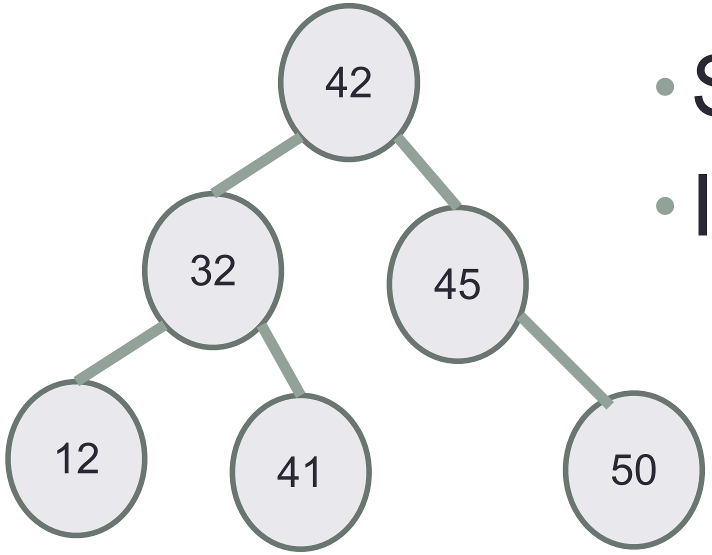
BSTs allow efficient search!



- 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

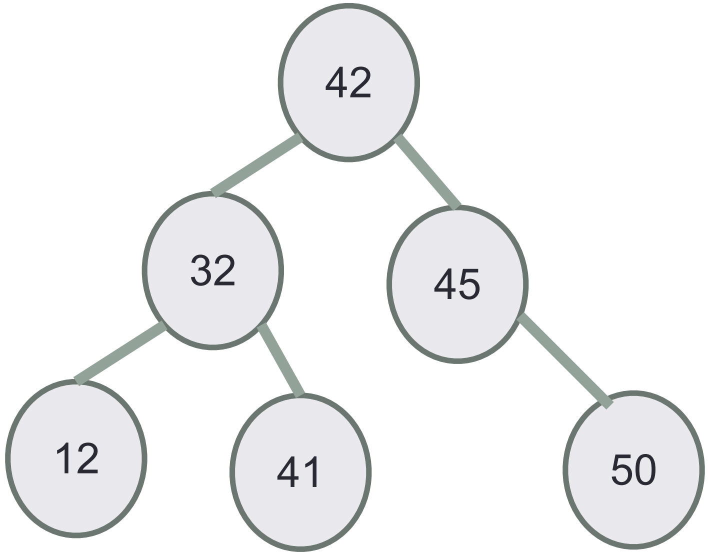
Search for 41, then search for 53

Insert



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

Min/Max



Which of the following described the algorithm to find the maximum value in the BST?

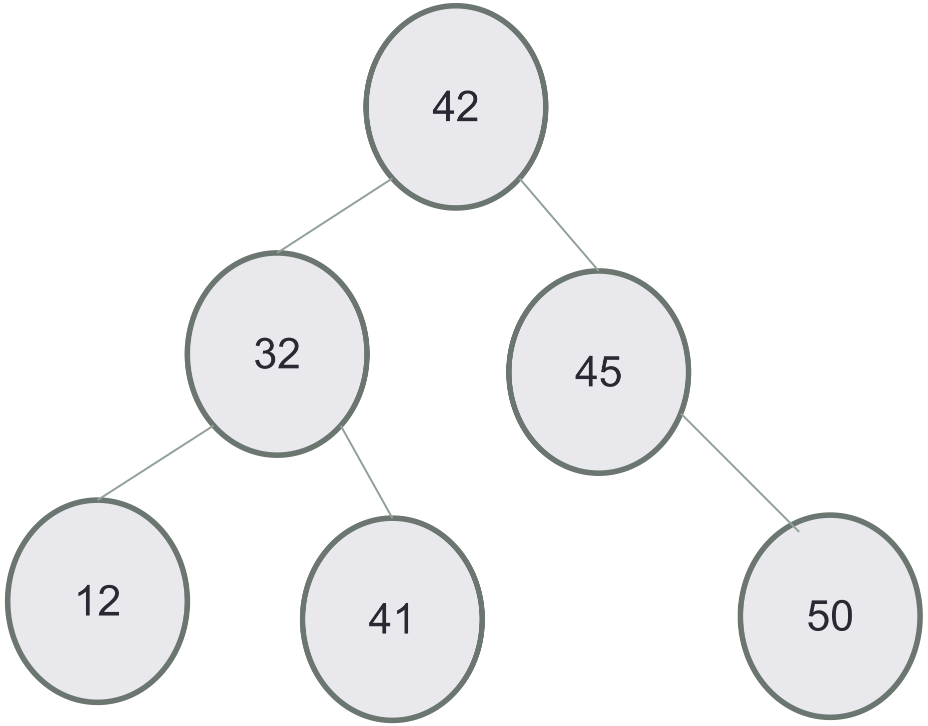
A. Return the root node's value

B. Follow right child pointers from the root, until a node with no right child is encountered, return that node's key

C. Follow left child pointers from the root, until a node with no left child is encountered, return that node's key

Define the BST ADT

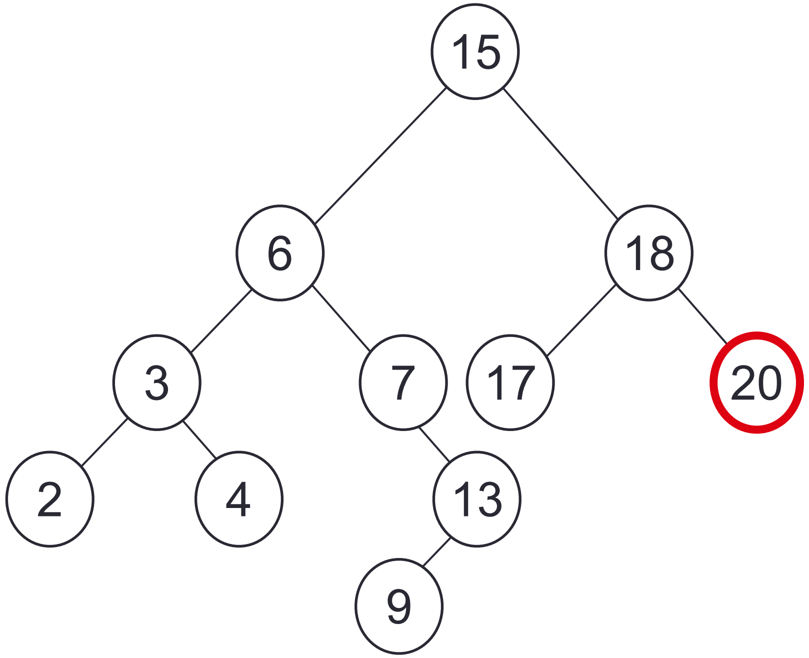
Operations
Search
Insert
Min
Max
Successor
Predecessor
Delete
Print elements in order



```
class BSTNode {

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

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

Max: find the maximum key value in a BST

Alg: `int BST::max()`

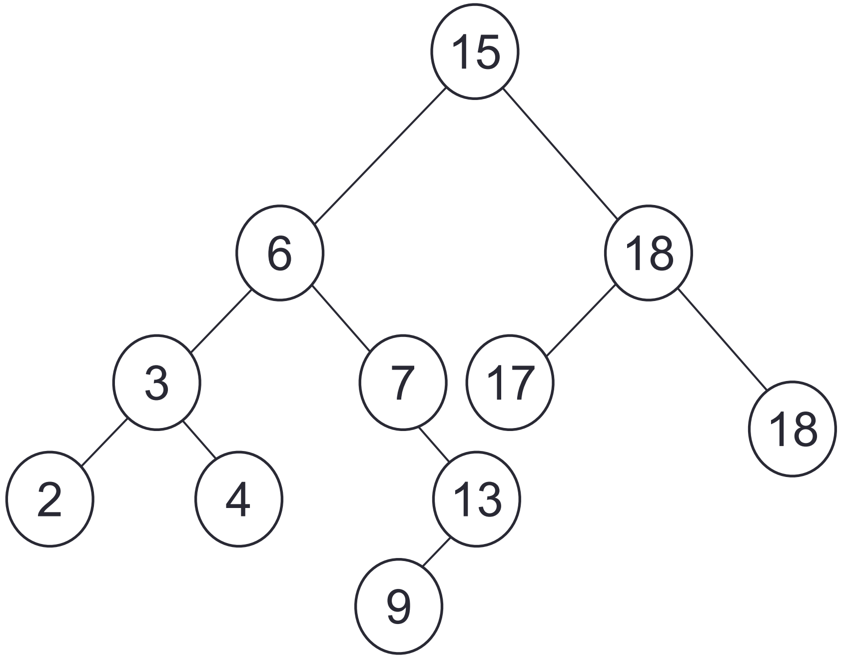


Maximum = 20

Min: find the minimum key value in a BST

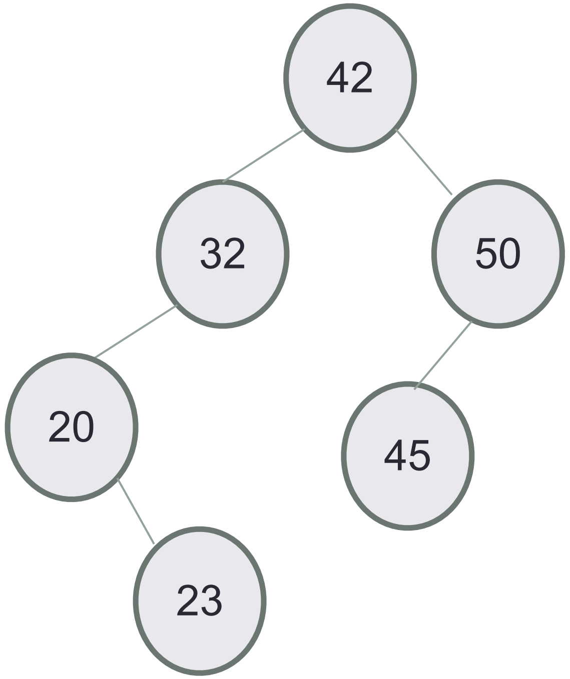
Alg: `int BST::min(){`

```
Start at the root.
Follow __________ child
pointers from the root, until
a node with no left child is
encountered.
Return the key of that node

}
```



Min = ?

Predecessor: Next smallest element



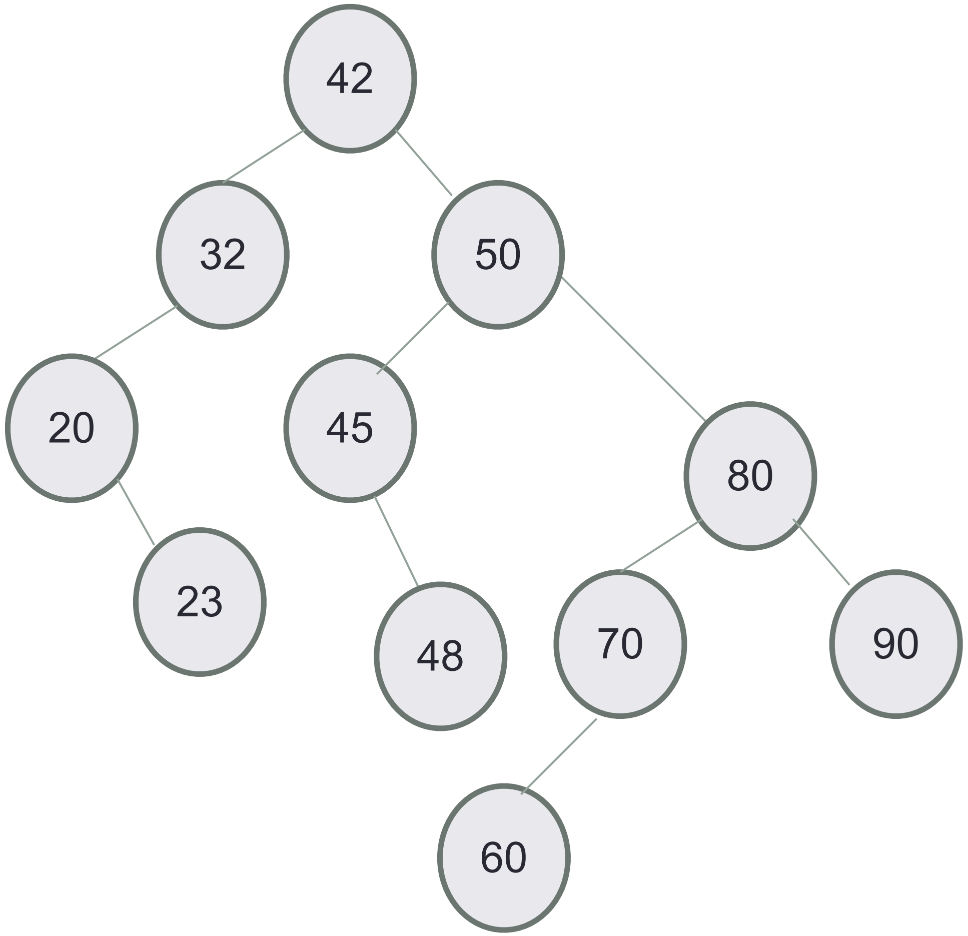

```
int bst::predecessor(BSTNode* n, int value) const{
    if(!n) return std::numeric_limits<int>::min();
    if(n->left){
        //Case 1
        return ________________________;

    }else{
        //Case 2


    }
}
```

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

Fill in the blank for case 1 using min/max helper functions

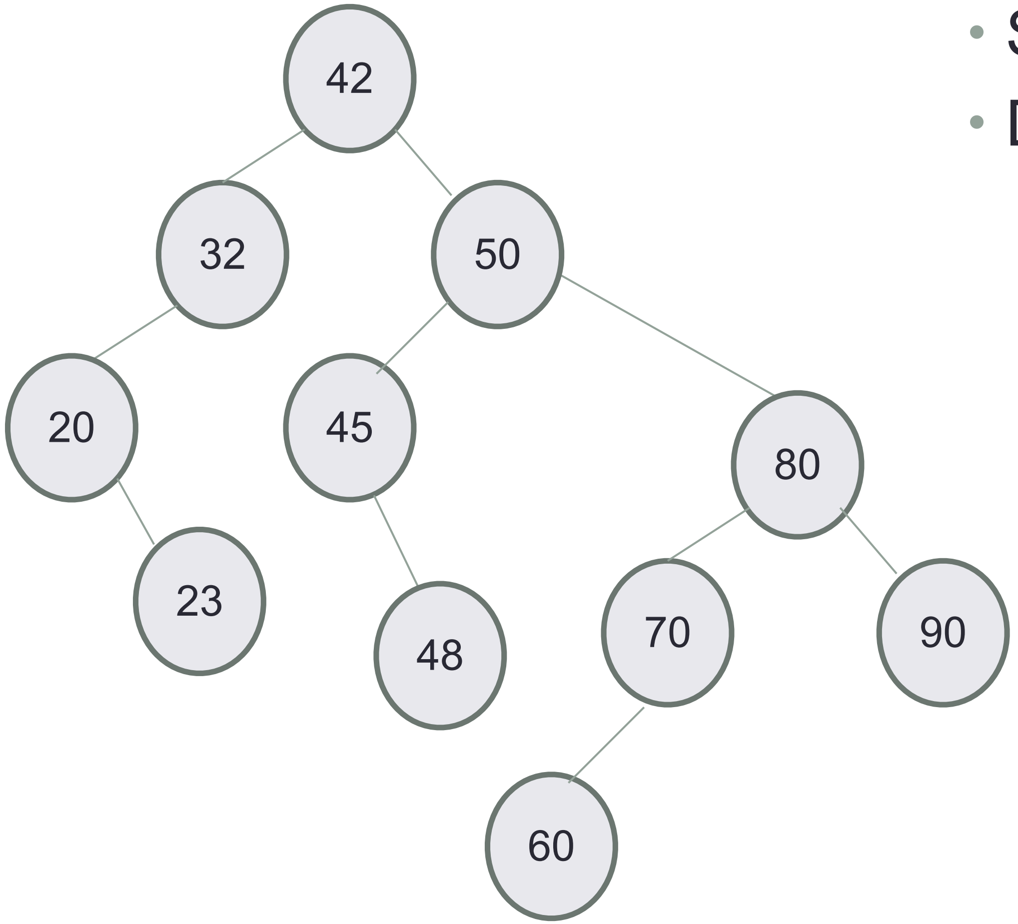
A. `n->left;`
B. `min(n)`
C. `max(n)`
D. `min(n->left)`
E. `max(n->left)`

Successor: Next largest element



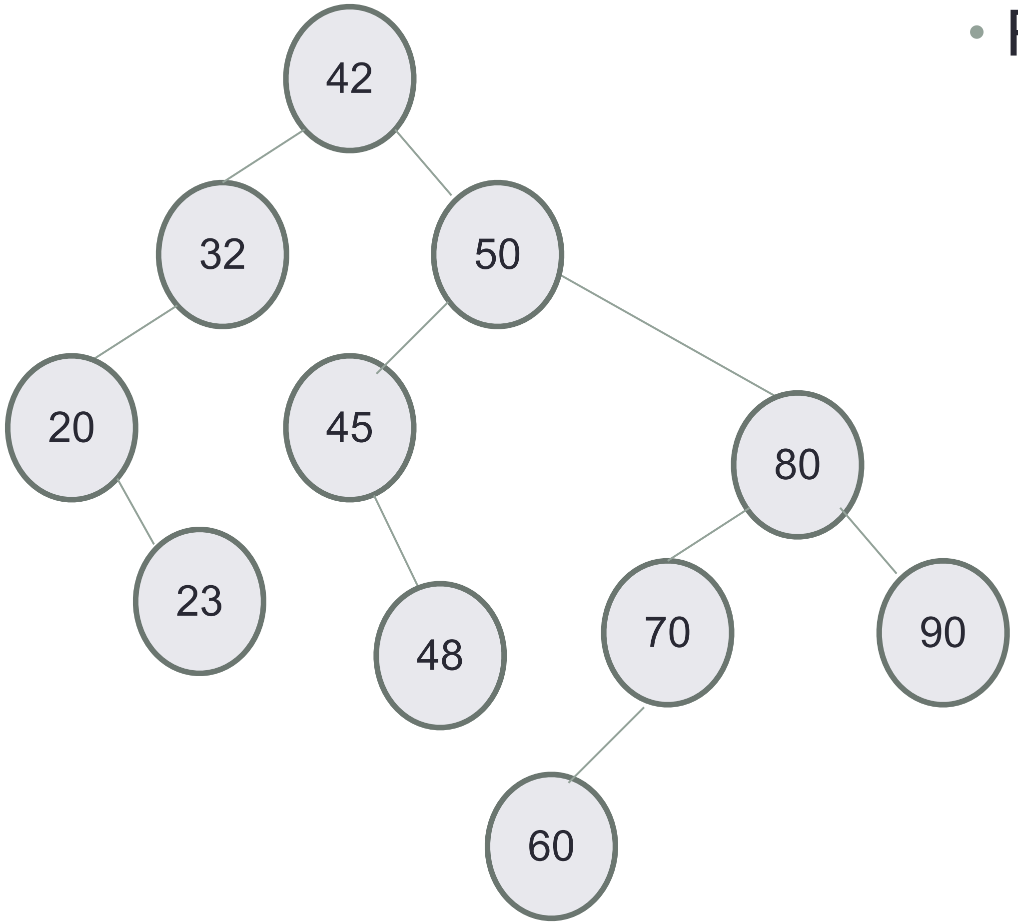
- What is the successor of 45?
- What is the successor of 50?
- What is the successor of 60?

Delete: Case 1 - Node is a leaf node



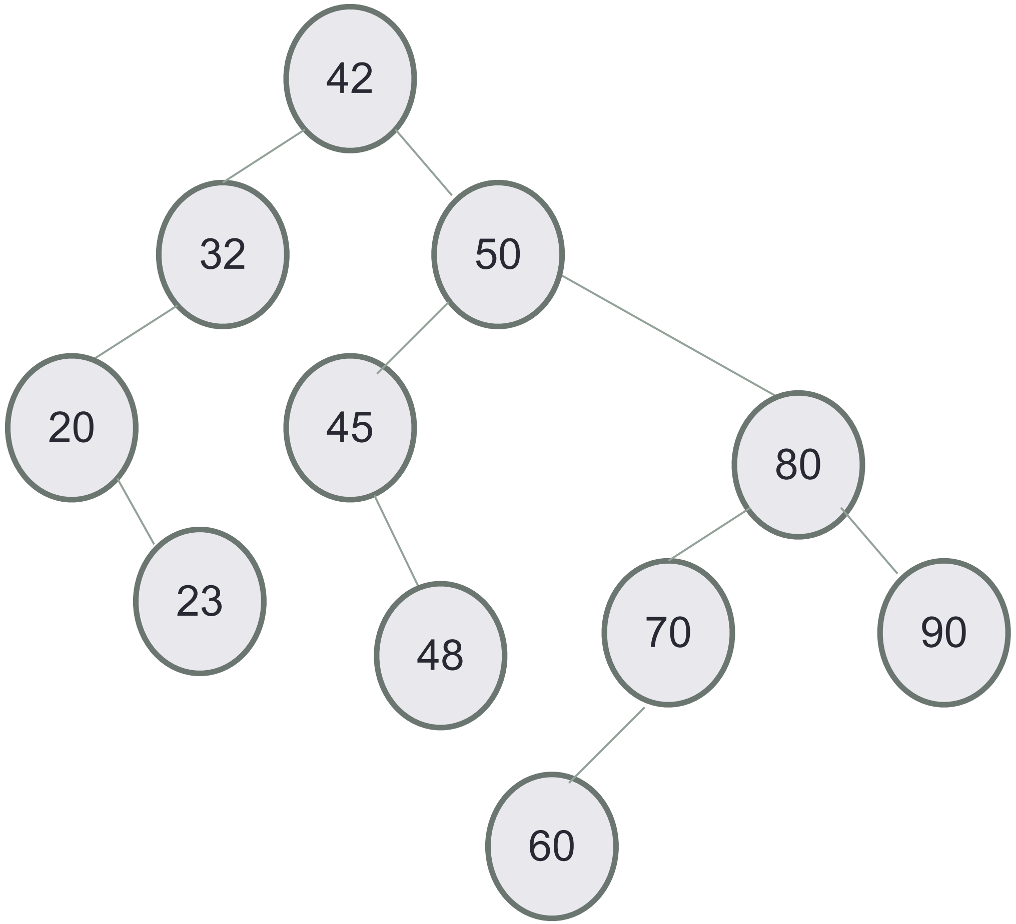
- Set parent’s (left/right) child pointer to null
- Delete the node

Delete: Case 2 - Node has only one child



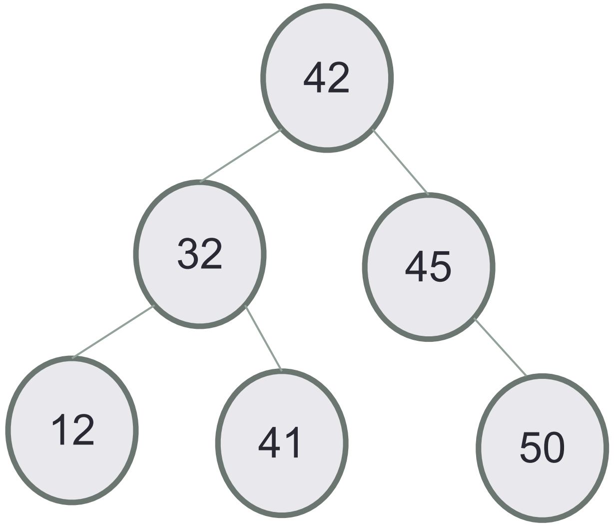
- Replace the node by its only child

Delete: Case 3 - Node has two children



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

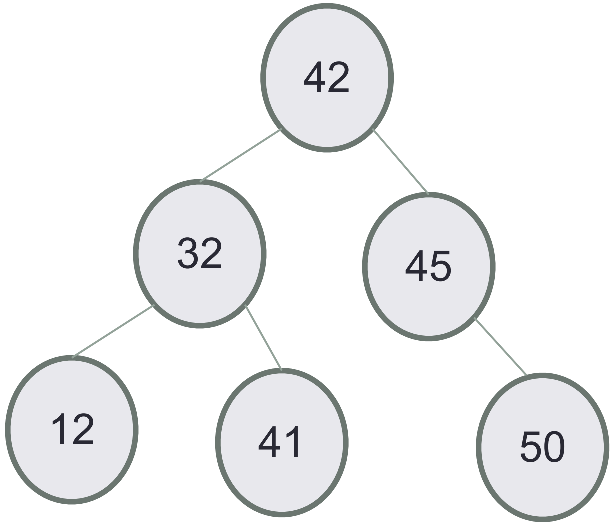
In order traversal: print elements in sorted order



Algorithm Inorder(tree)

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

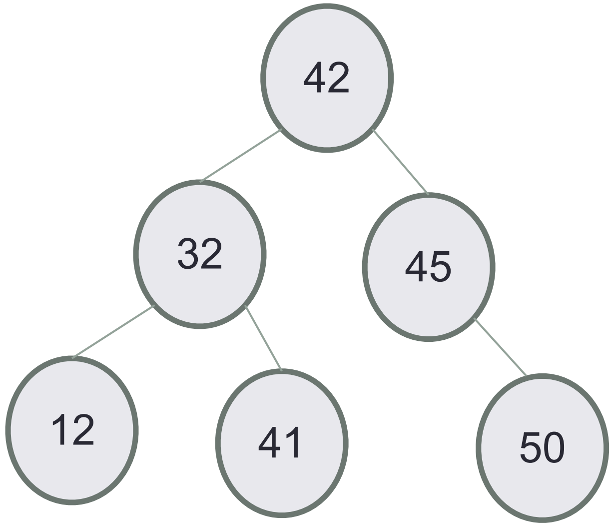
Pre-order traversal: nice way to linearize your tree!



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 to recursively clear the tree!



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.