

BINARY SEARCH TREES

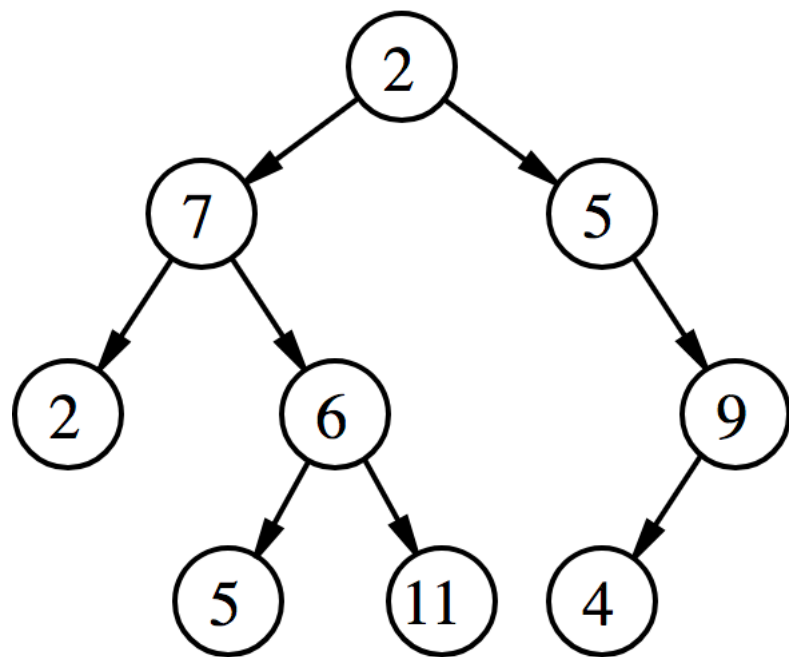
Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook!n";
    return 0;
}
```

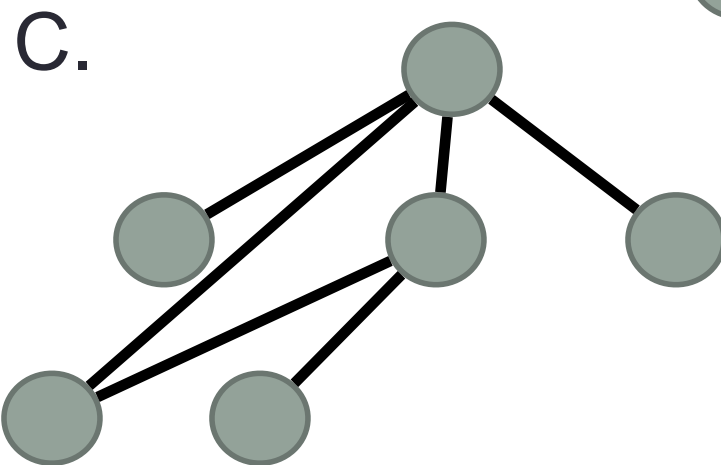
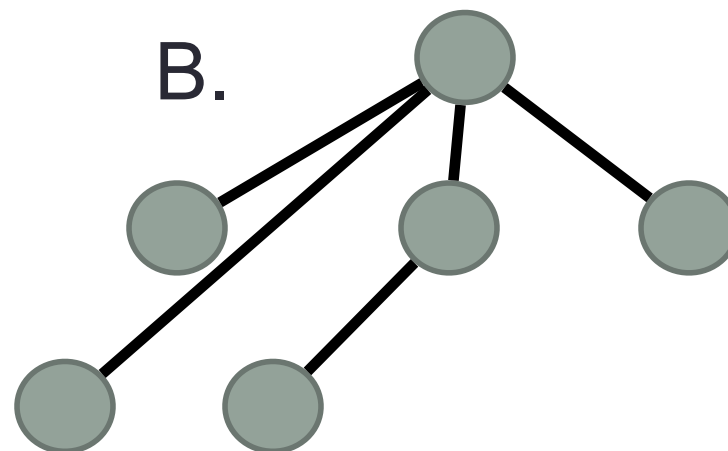

Trees



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?



D. A & B

E. All of A-C

Binary Search Trees

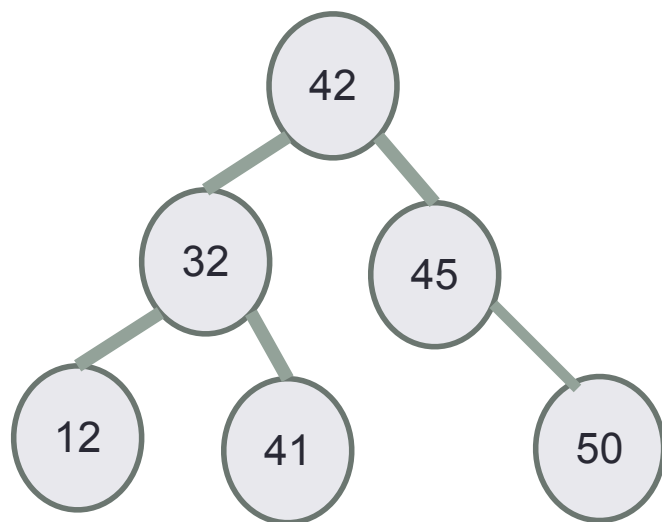
- What are the operations supported?
- What are the running times of these operations?
- How do you implement the BST i.e. operations supported by it?

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

Example keys: 42, 32, 45, 12, 41, 50

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

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**

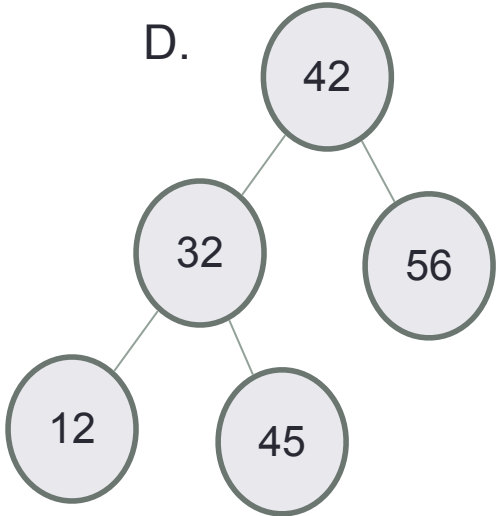
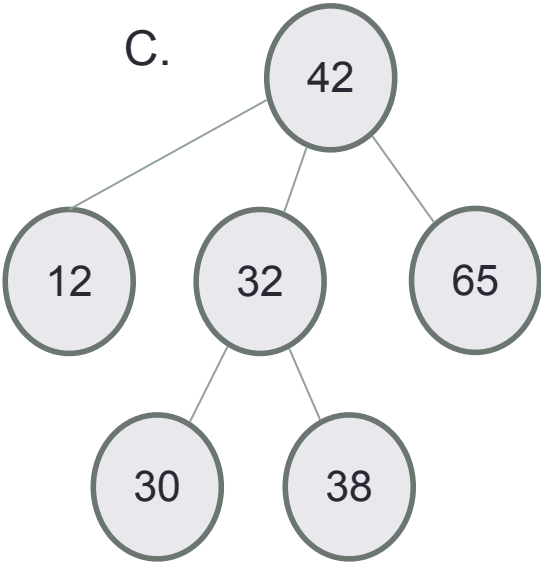
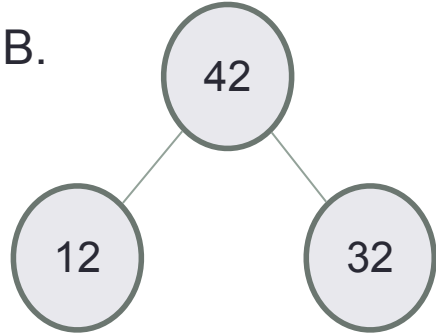
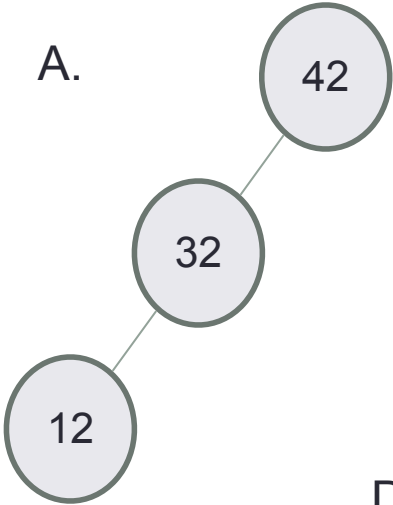
For any node,

Keys in node's left subtree < Node's key

Node's key < Keys in node's right subtree

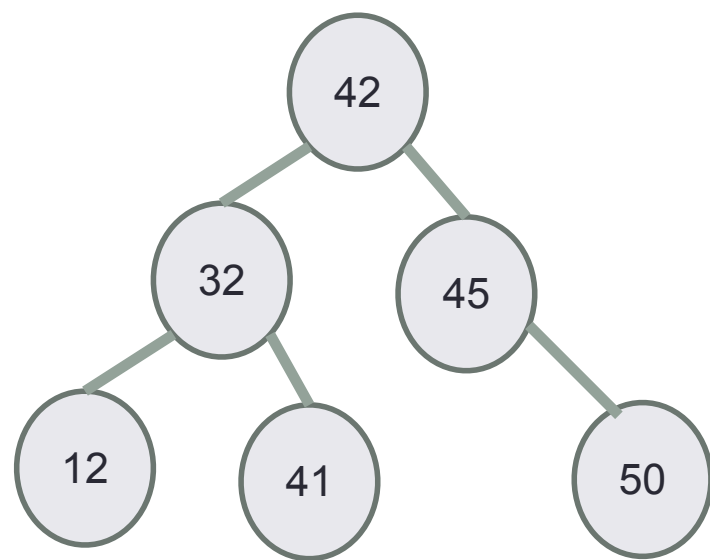
Do the keys have to be integers?

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



E. More than one of these

BSTs allow efficient search!

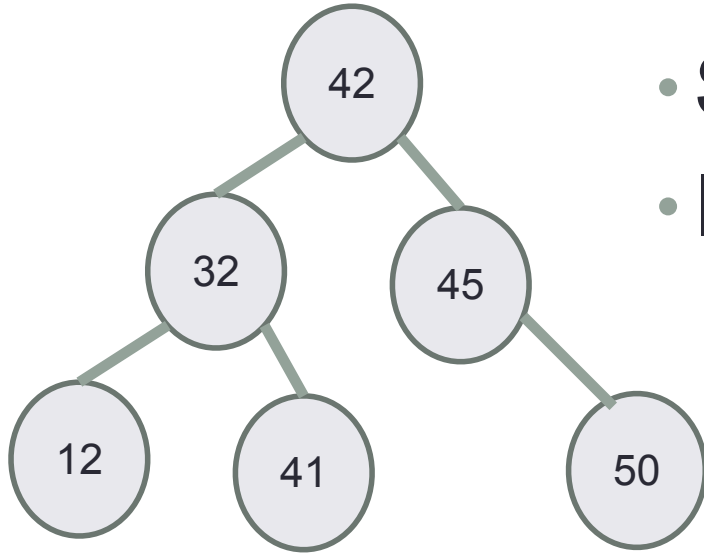


- Start at the root;
- Trace down a path by comparing k with the key of the current node x :
 - If the keys are equal: we have found the key
 - If $k < \text{key}[x]$ search in the left subtree of x
 - If $k > \text{key}[x]$ search in the right subtree of x



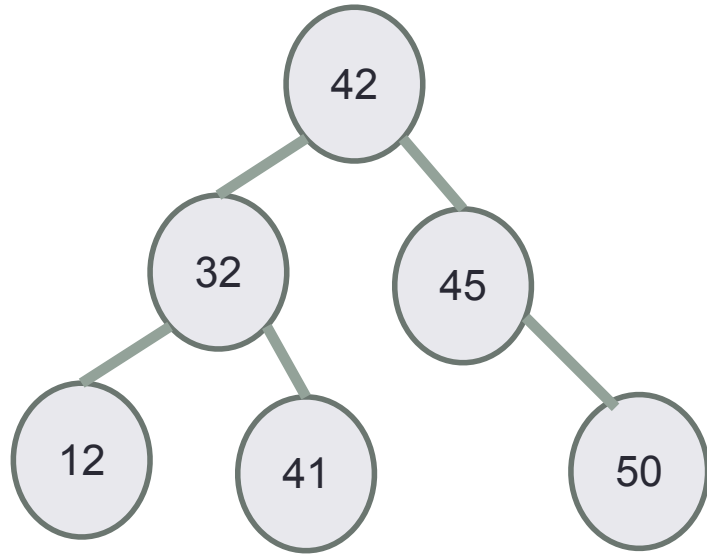
Search for 41, then search for 53

Insert



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

Min/Max

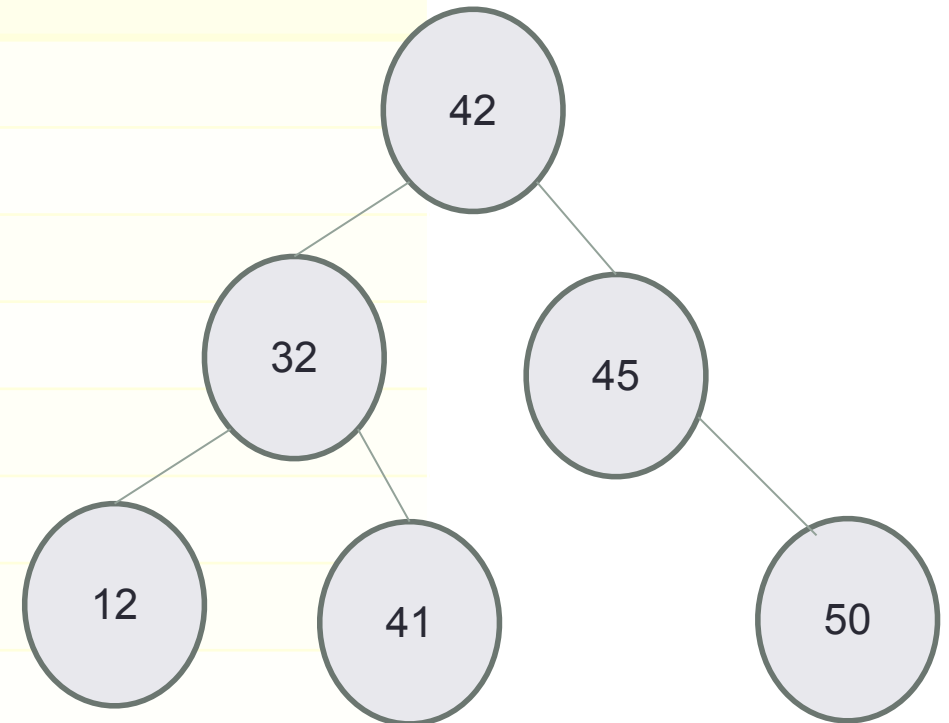


Which of the following described the algorithm to find the maximum value in the BST?

- A. Follow **right child** pointers from the root, until a node with no right child is encountered, return that node's key
- B. Follow **left child** pointers from the root, until a node with no left child is encountered, return that node's key
- C. Traverse to the **last level in the tree** and traverse the tree left to right, return the key of the **last node in the last level**.

Define the BST ADT

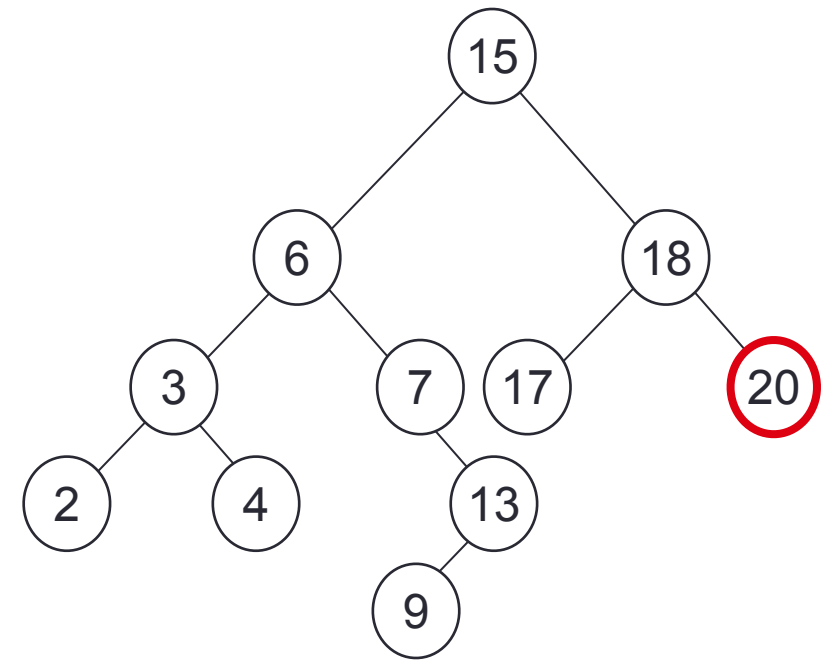
Operations
Search
Insert
Min
Max
Successor
Predecessor
Delete
Print elements in order



```
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

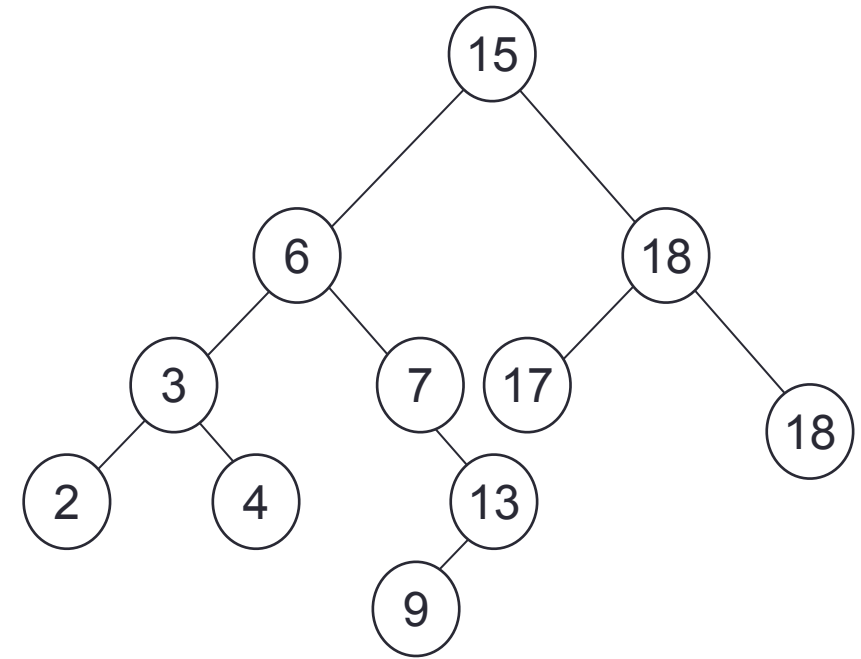
```
Alg: int BST::min() {
```

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 = ?

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;
```

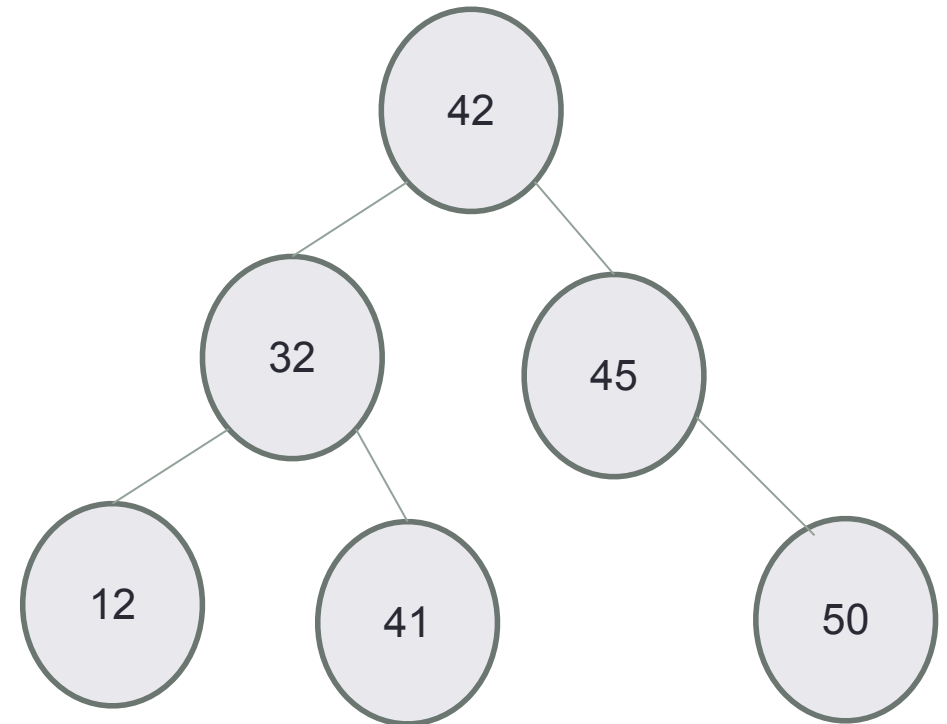
A. 42

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;
```

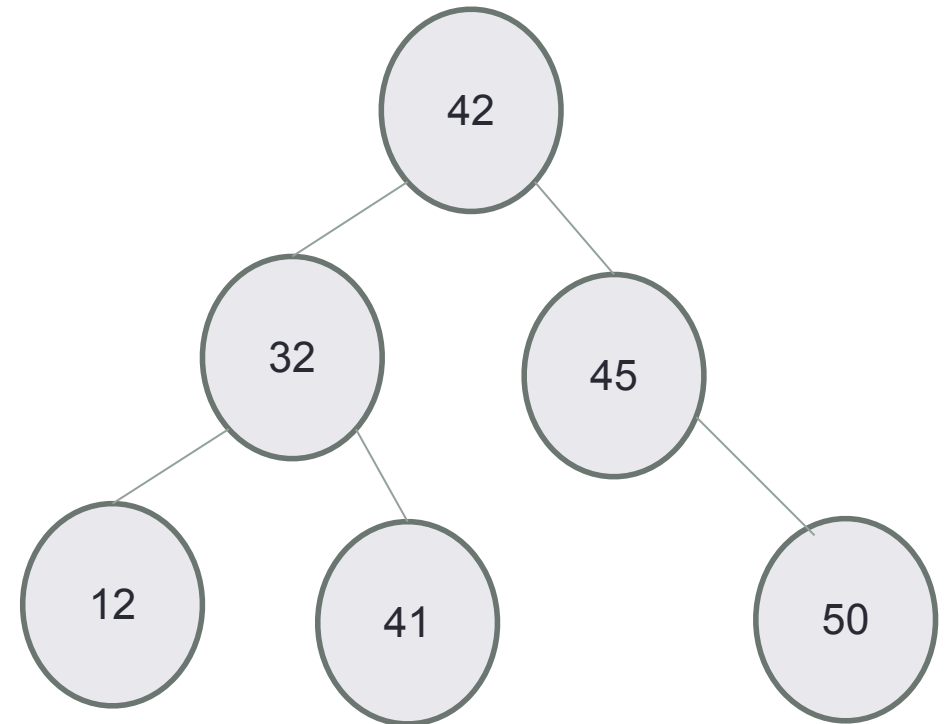
A. 42

B. 32

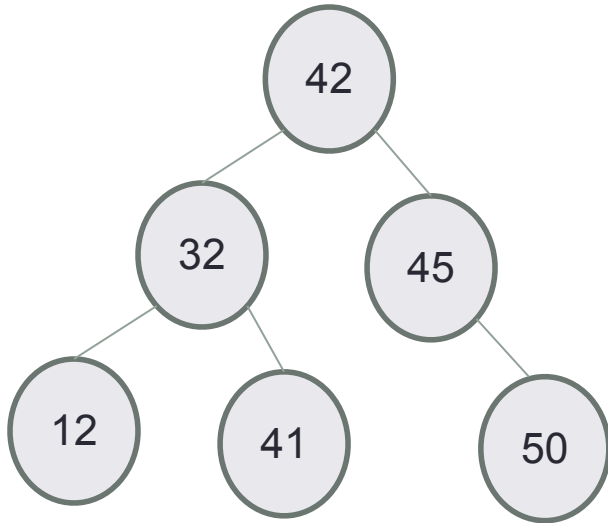
C. 12

D. 45

E. Segfault



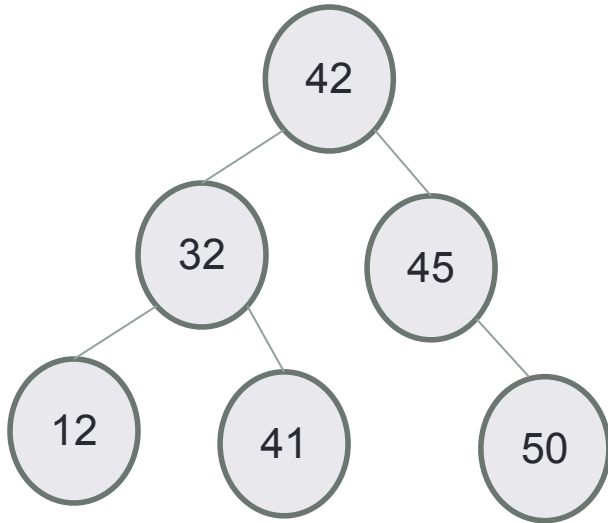
In order traversal: print elements in sorted order



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)

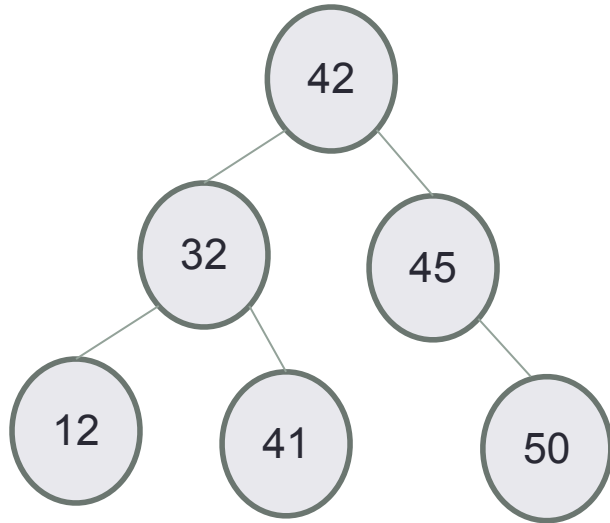
Pre-order traversal: nice way to linearize your tree!



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)

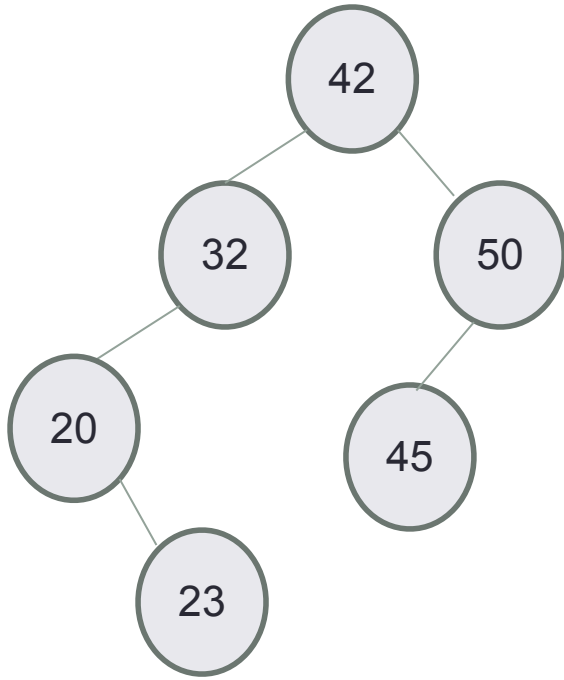
Post-order traversal: use to recursively clear the tree!



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.

Predecessor: Next smallest element



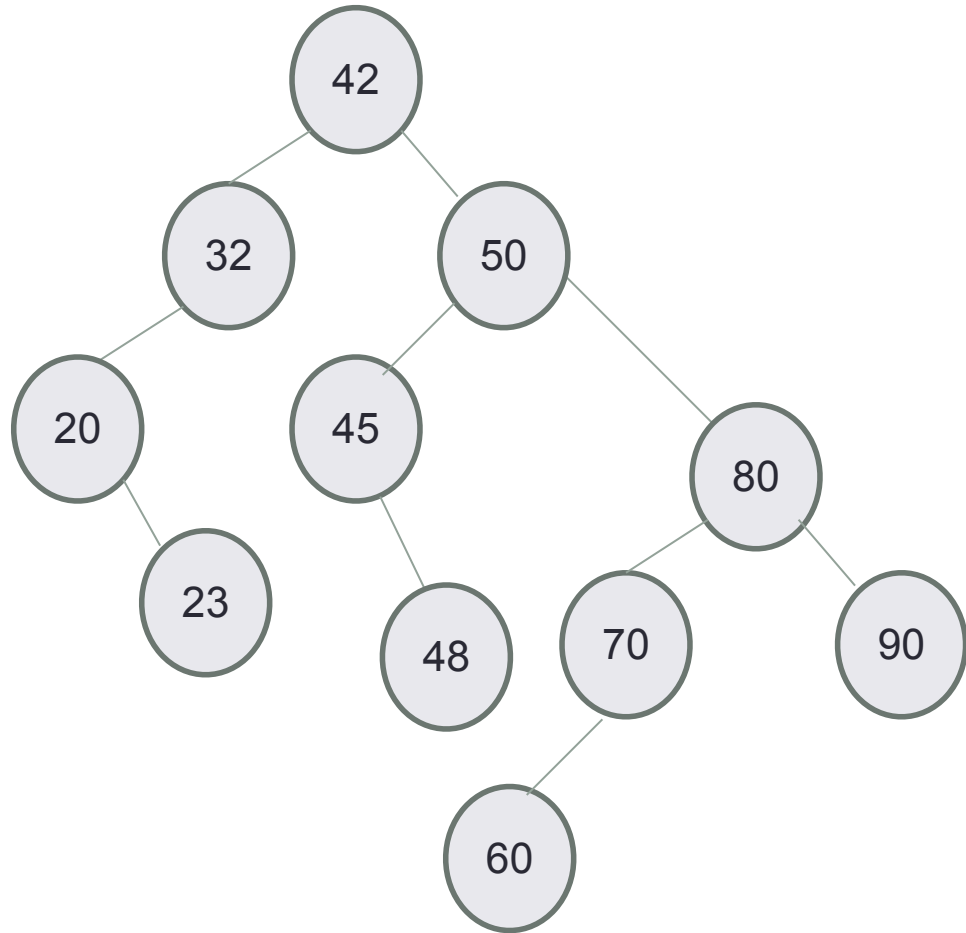
- What is the predecessor of 32?
- What is the predecessor of 45?

```
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  
    }  
}
```

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)

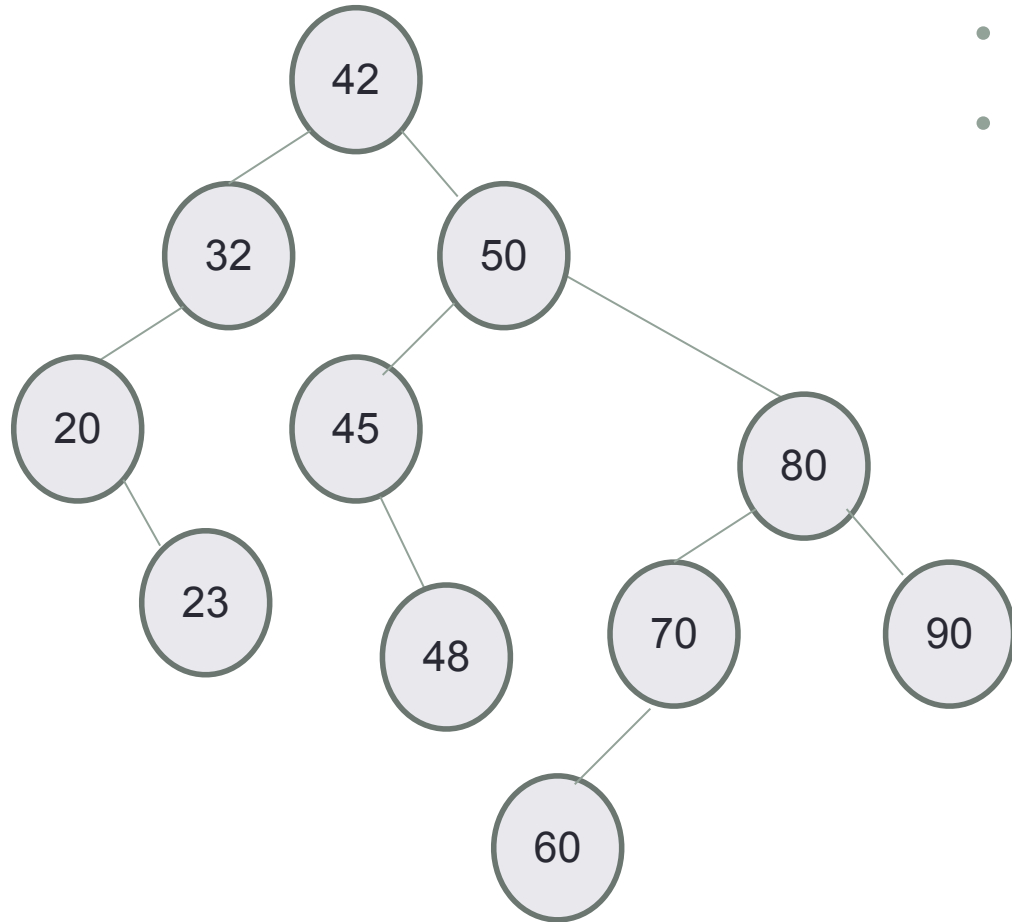
Successor: Next largest element



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

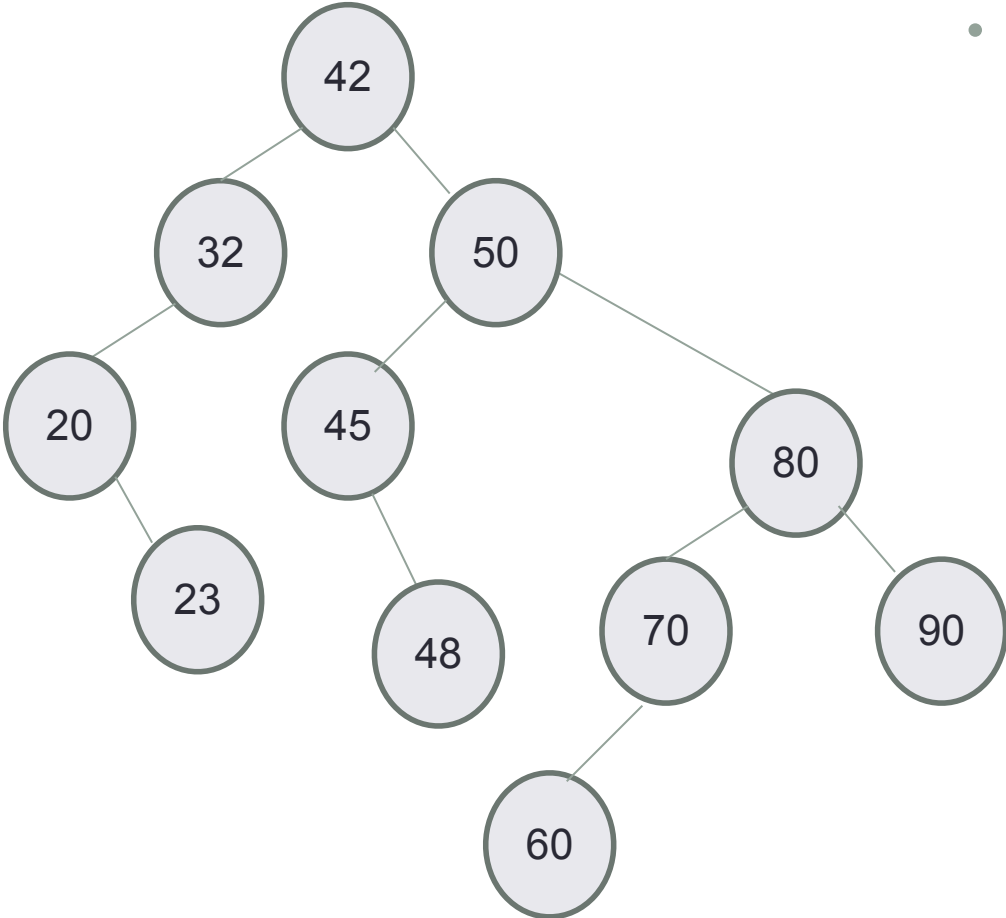
Delete: Case 1 - Node is a leaf node

- 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?

