BINARY SEARCH TREES

Problem Solving with Computers-II

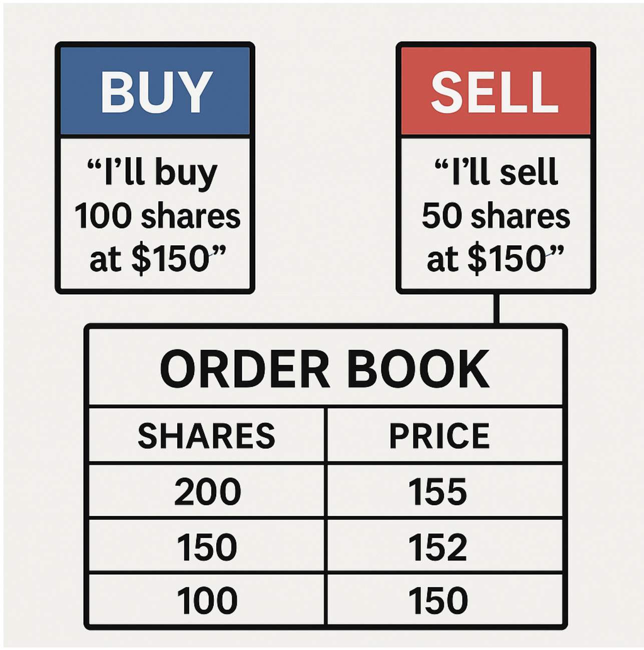
```
#include <iostream>
using namespace std;

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

Imagine you're designing a system for a stock exchange.

Real-Time System: Traders submit buy/sell orders in real-time — thousands per second.

Order Book: List of buy orders, sorted by price (highest to lowest), matches orders at the same price.

Key Operations: Insert new orders, search for matches, print in sorted order, find the best price.



SHARES	PRICE
200	155
150	152
100	150

To manage the buy/sell order book, which data structure should we use?

To manage the buy order book, which data structure should we use?

The system needs to handle a stream of new orders, search for prices to match trades, print orders in sorted order, and find the best price—all as fast as possible.

Discuss with your peers and vote for the best option:

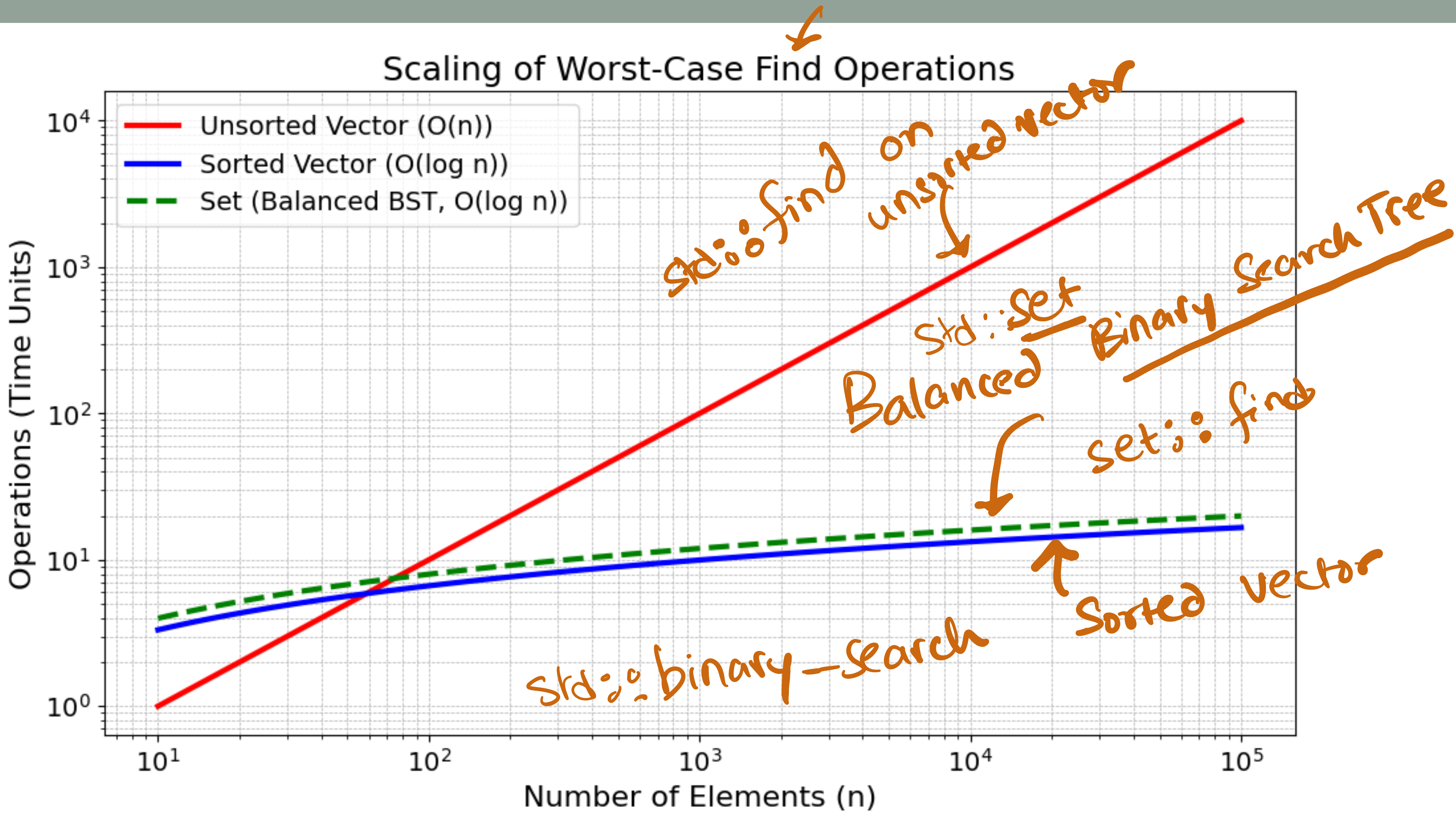
A) **Unsorted Vector**: Add orders with push_back $O(1)$, search with std::find $O(N)$, sort when printing.

B) **Sorted Vector**: Add orders, re-sort after each addition $O(N \log N)$, search with std::binary_search $O(\log N)$.

[✓ C)] **std::set**: Add orders with insert $O(\log N)$, search with find $O(\log N)$, print in order $O(N)$, get best price with begin() ← min() $O(\log N)$.

D) **Array**: Fixed-size array, manually manage sorting and searching. ×

E) More than one options works!

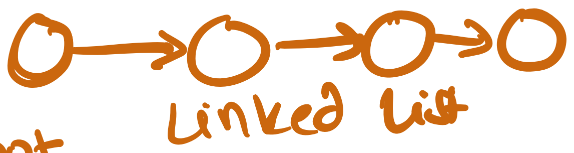

Scaling of Worst-Case Find Operations
Unsorted Vector (O(n))
Sorted Vector (O(log n))
Set (Balanced BST, O(log n))
Operations (Time Units)
Number of Elements (n)
10^0
10^1
10^2
10^3
10^4
10^5
std::find on unsorted vector
std::set
Balanced Binary Search Tree
set::find
Sorted vector
std::binary_search

Binary Search Trees (std::set)

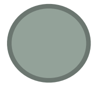
- What are the operations supported? https://cplusplus.com/reference/set/set/?kw=set
- What are the running times of these operations?
- How do you implement the BST i.e. operations supported by it?

Std: Set is a tree!

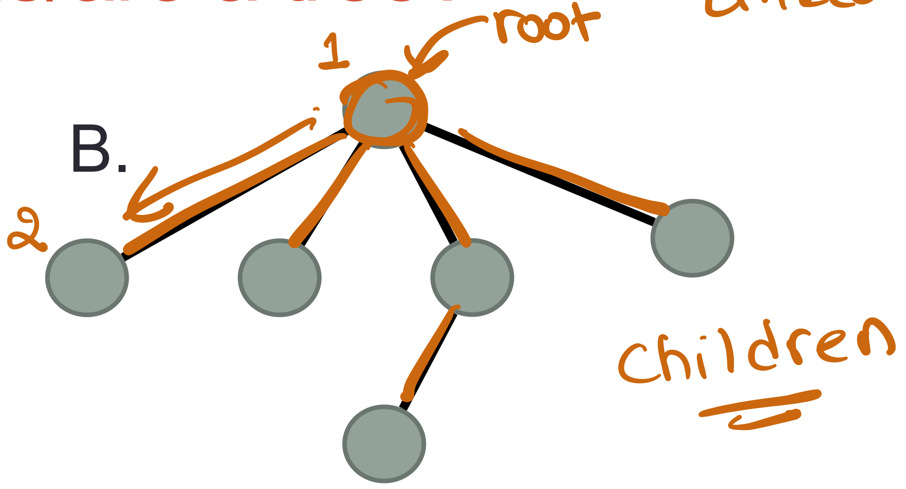
Which of the following is/are a tree?



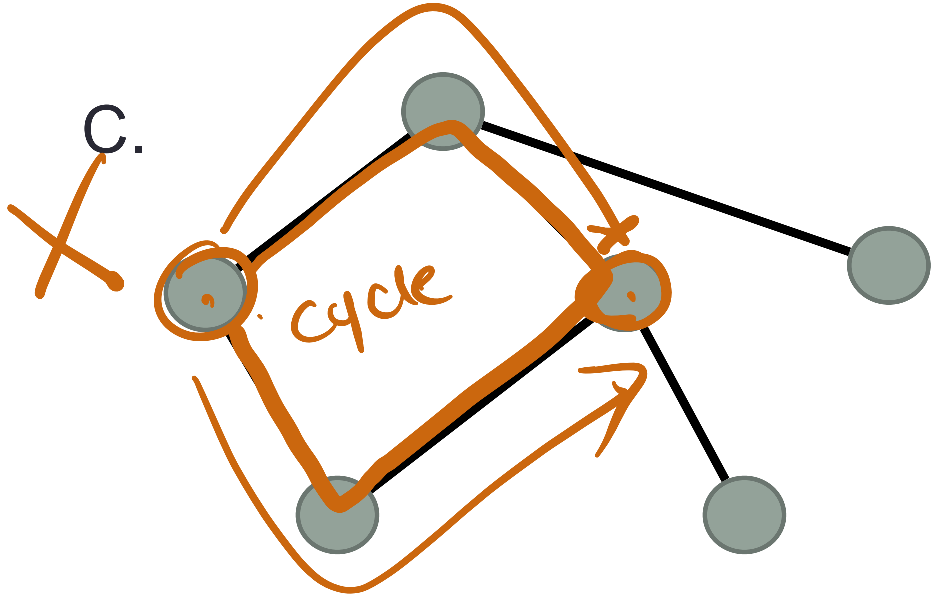
A.



B.



C.

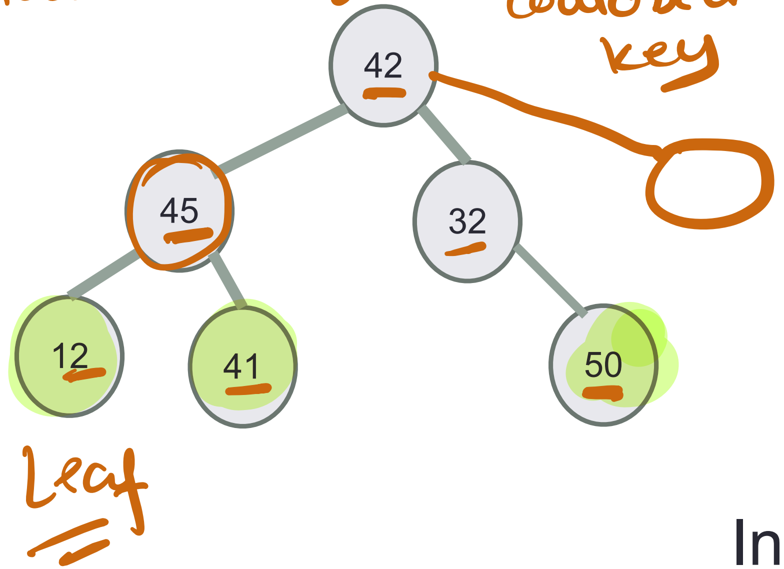


D. A & B

E. All of A-C

Binary Trees

Stock exchange problem price could be a key



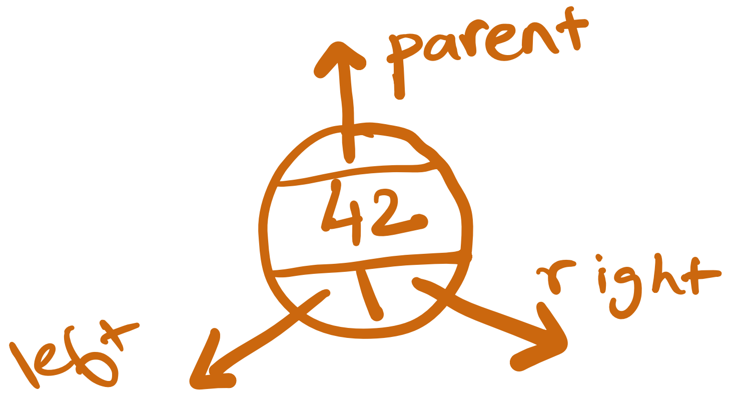
In a tree, nodes are arranged in a heirarchy

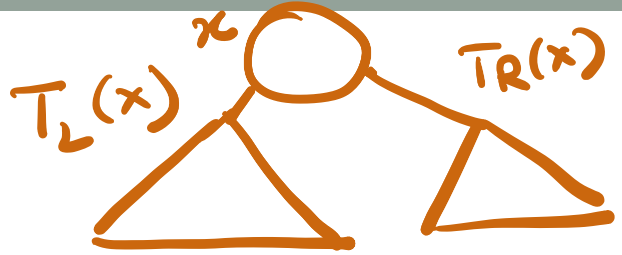
- One node is distinguished as the root
- Each node:
  - stores a key ✓
  - has a pointer to child nodes and parent (optional)
- Unique path between any two nodes
- Leaf nodes have no children

In a binary tree, each node has at most
____two____ children

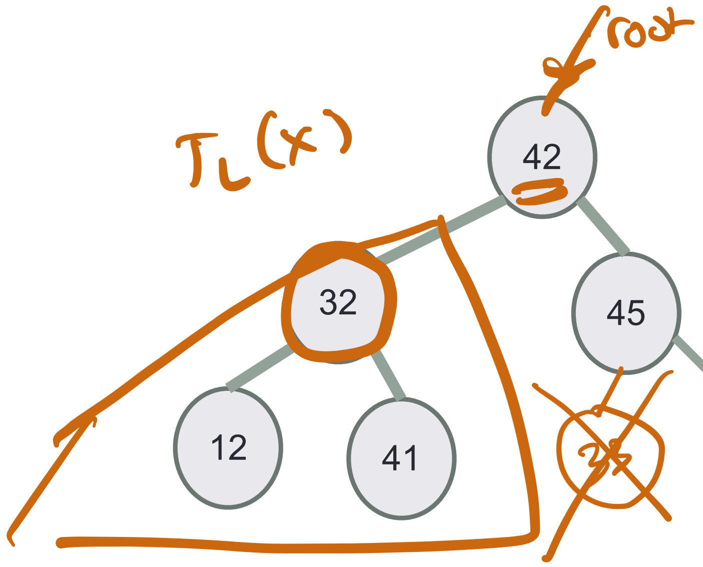
Binary

```
struct TreeNode {

  TreeNode* left;
  TreeNode* right;
  TreeNode* parent;
  int const data;  key

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





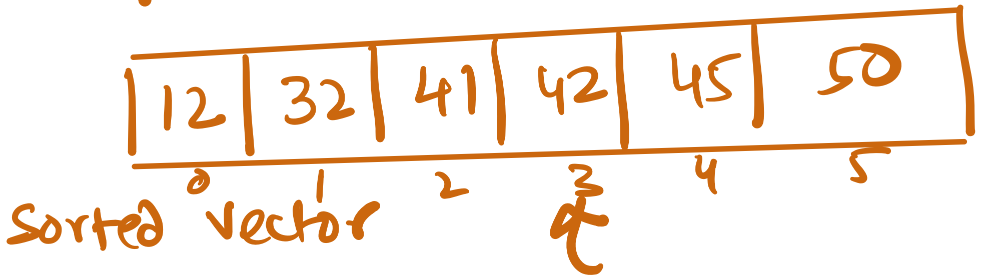
Binary Search Tree – What is it?



BST is a binary tree where each node satisfies the Search Tree Property

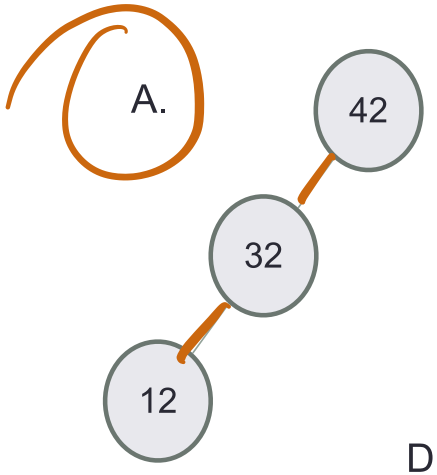
For any node,
Keys in node's left subtree < Node's key < Keys in node's right subtree

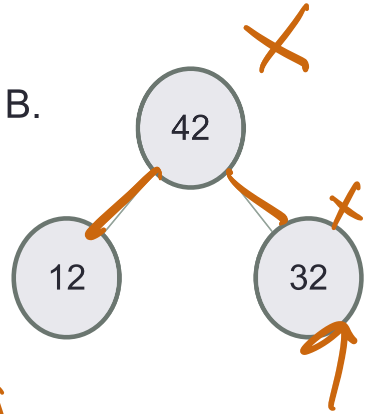
$$\text{keys}(T_L(x)) < \text{key}(x) < \text{keys}(T_R(x))$$

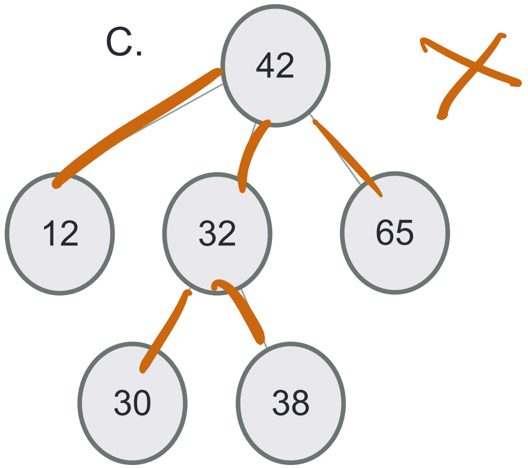


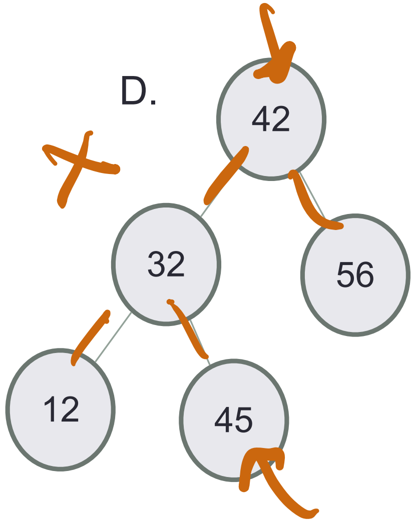
std::set does not store duplicate values
Do the keys have to be integers?

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



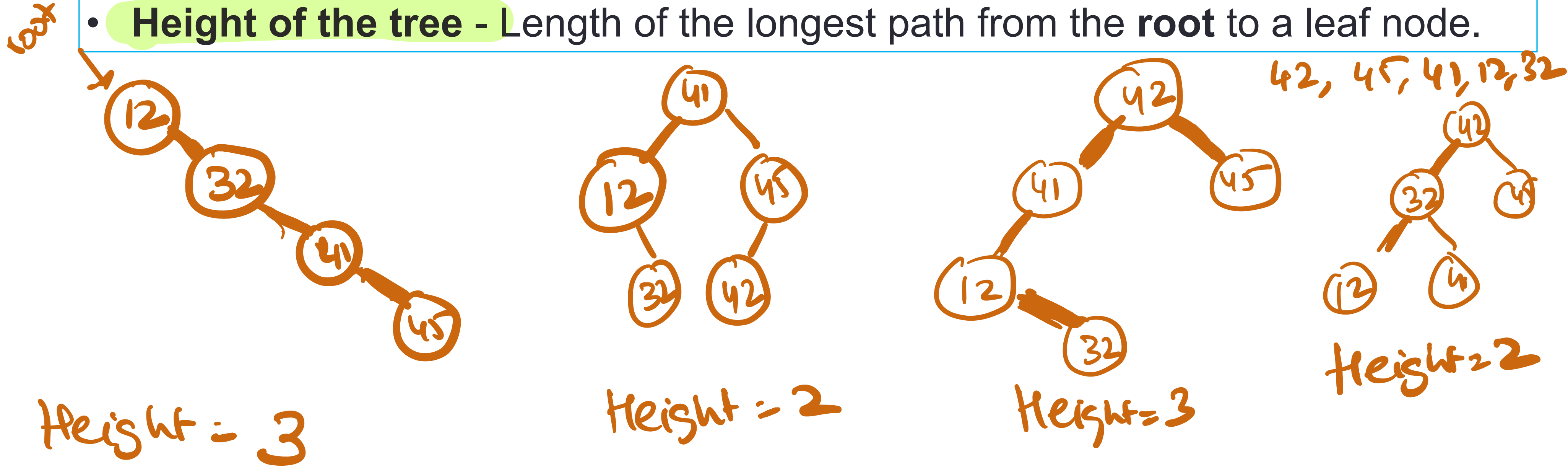






E. More than one of these

- Path – a sequence of (zero or more) connected nodes.
- Length of a path - number of edges traversed on the path
- Height of node – Length of the longest path from the node to a leaf node.
- **Height of the tree -** Length of the longest path from the **root** to a leaf node.



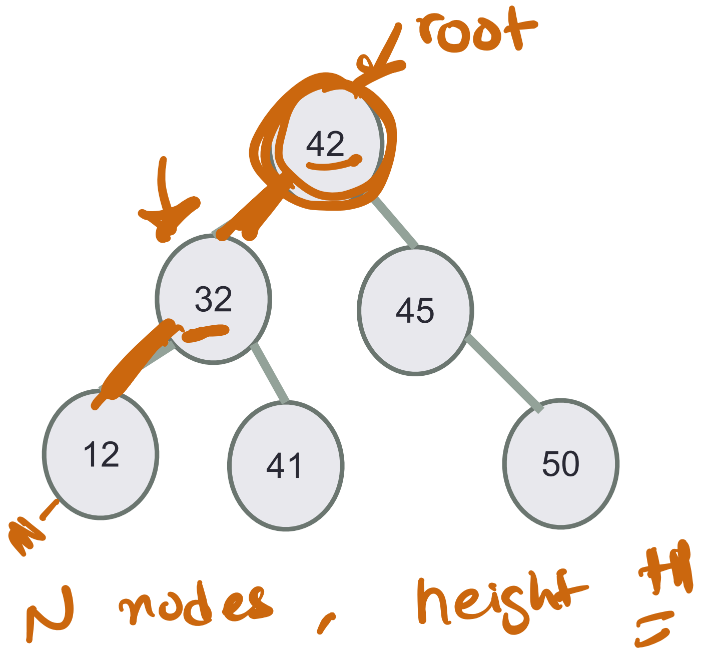
BSTs of different heights are possible with the same set of keys

Examples for keys: 12, 32, 41, 42, 45

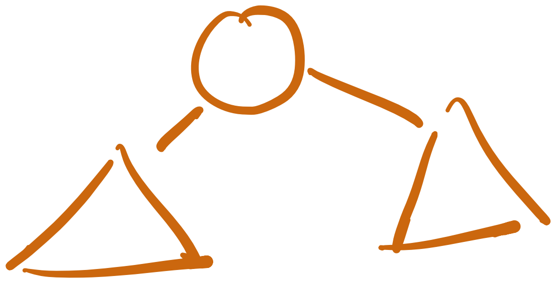
2 1 4 3

BSTs allow efficient search!

Input key k to search for

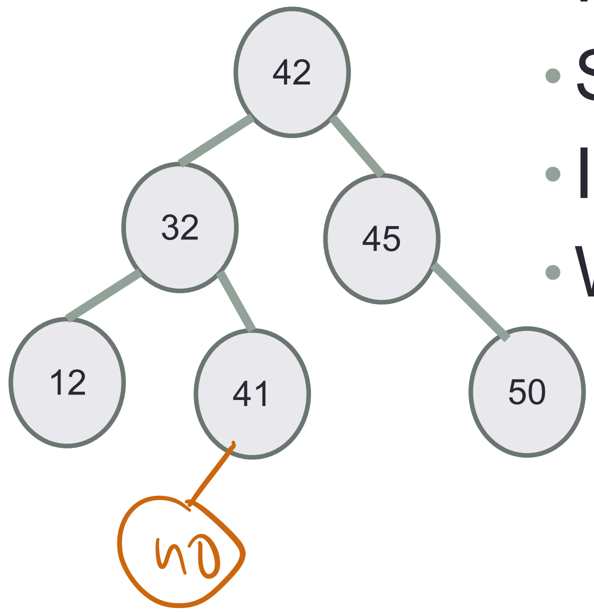


- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x
- What is the running time of search? O(H)



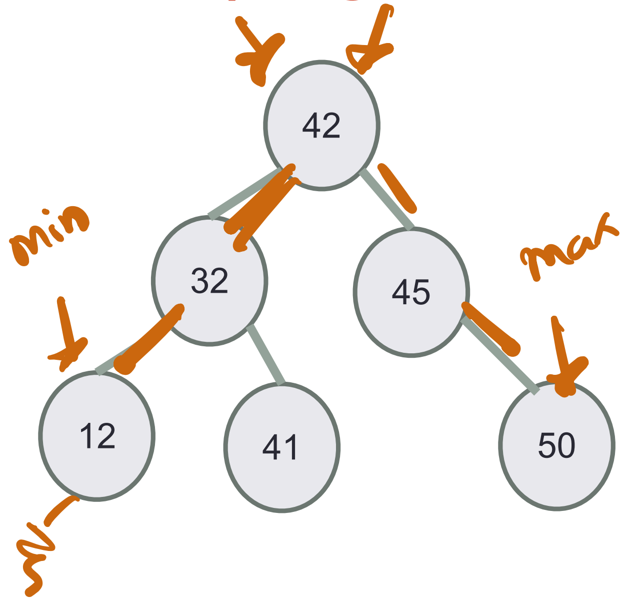
Search for 41, then search for 53

Insert



- Insert 40
- Search for the key
- Insert at the spot you expected to find it
- What is the running time of insert? O(H)

Min/Max



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

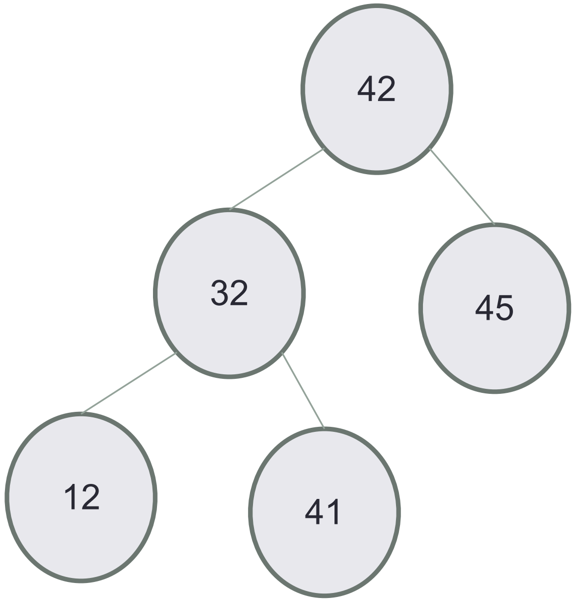
A. Return the root node's value

[✓ 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

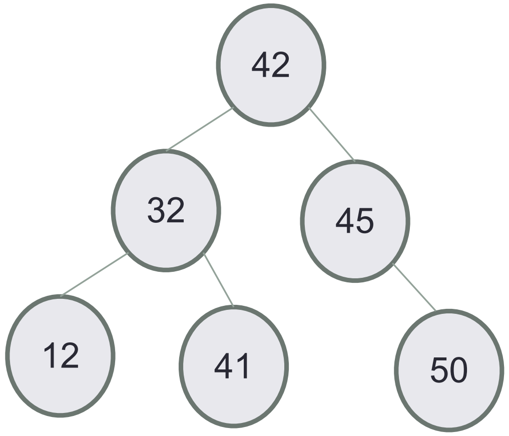
Define the BST ADT

Operations
Search
Insert
Min
Max
Successor (next largest key)
Predecessor (next smaller key)
Delete
Print elements (3 variations)



Write a function to create a small BST manually (not using insert!)

In order traversal: print elements in sorted order

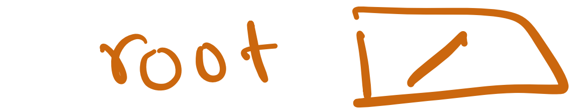


```
Algorithm Inorder(tree)
   If tree is empty, return
   Traverse the left subtree, i.e., call Inorder(left-subtree)
   Visit the root.
   Traverse the right subtree, i.e., call Inorder(right-subtree)
```

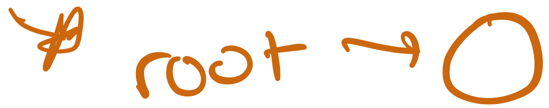
Write a member function for the BST ADT to compute its height

```
int bst::getHeight(Node* ) const{
    return getHeight(root); // Implement the helper recursively
} // returns the height of the tree
```

Base case -

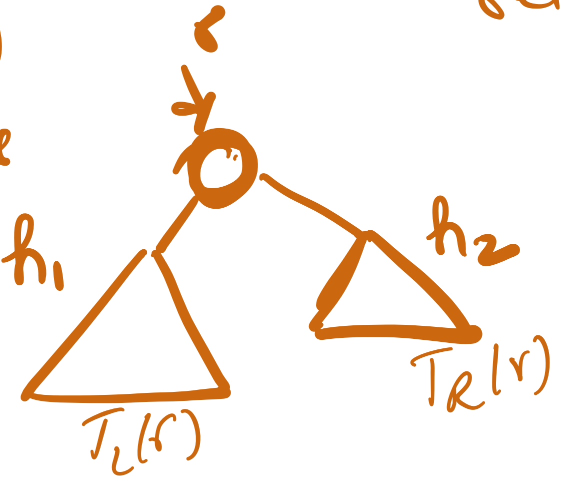


return -1



return 0

Recursive case



height = $\max(h_1, h_2) + 1$