BINARY SEARCH TREES

Problem Solving with Computers-II

Binary Search

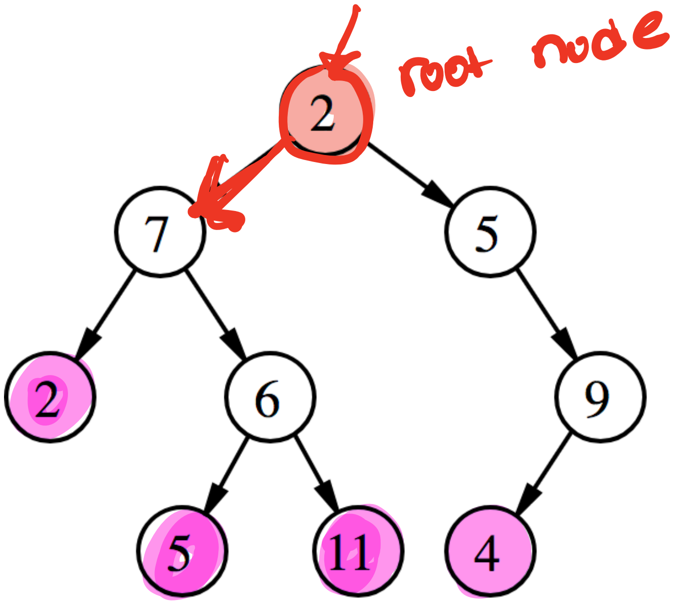
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Insert 5

- 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

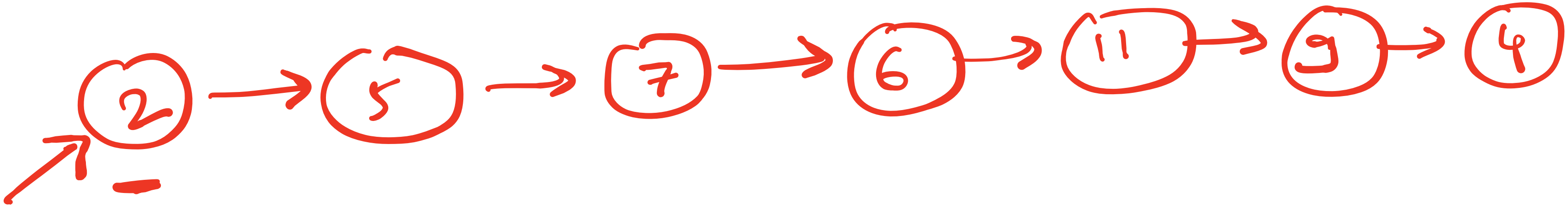
Trees



A tree has following general properties:

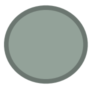
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*

Node 2 is parent of 7 & 5

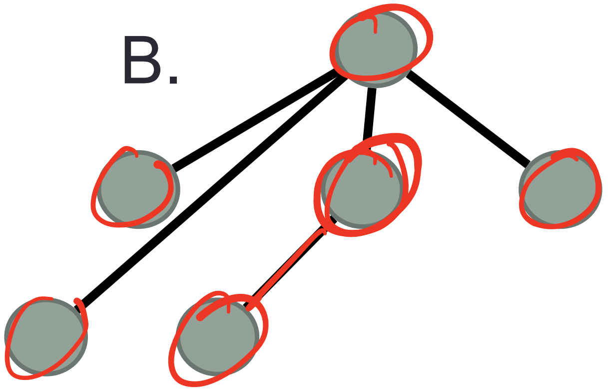


Which of the following is/are a tree?

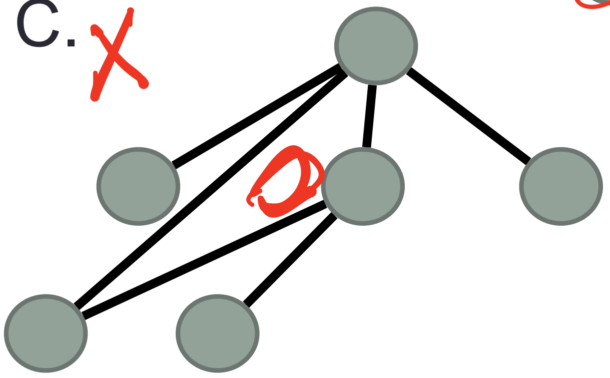
A.



B.



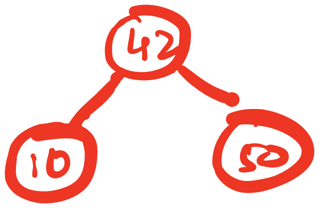
C.



D. A & B

E. All of A-C

Empty root



42's left child is 10
42's right child is 50

Binary Search Trees

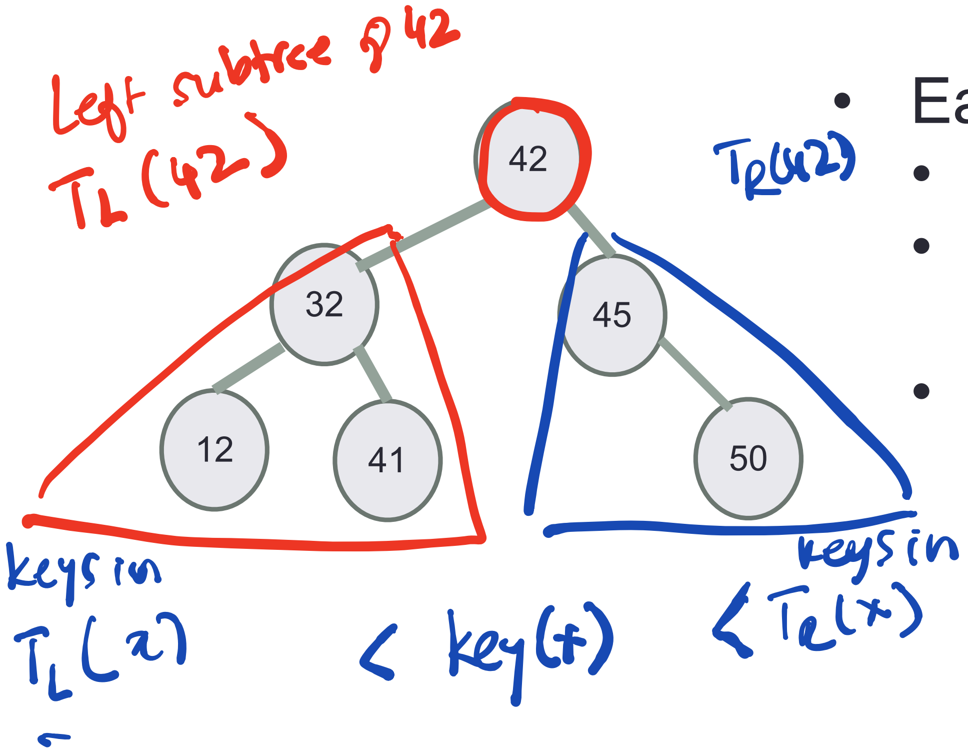
- What are the operations supported?

Efficient Search + fast insert & deletions

- 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?



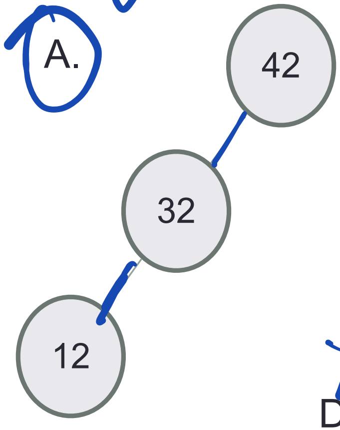
- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property

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

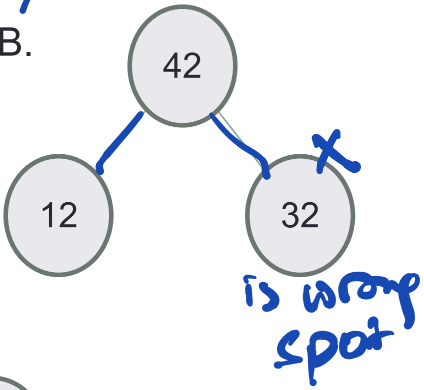
Do the keys have to be integers?

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

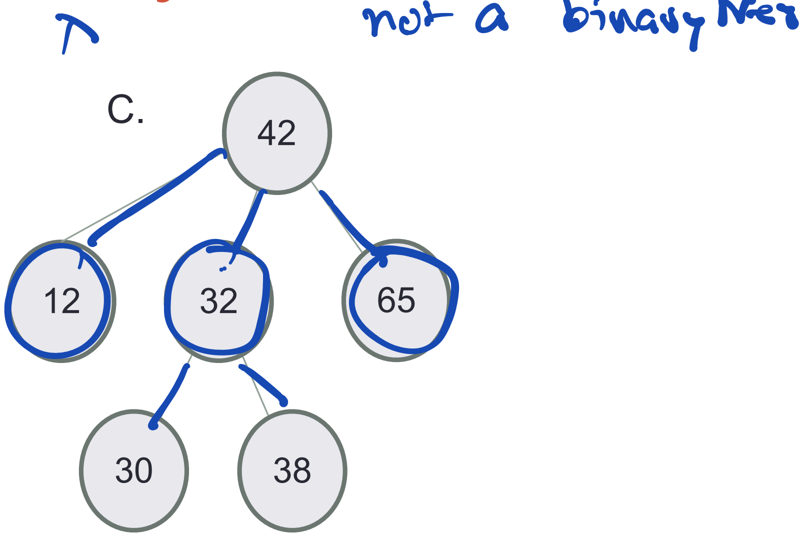
A.



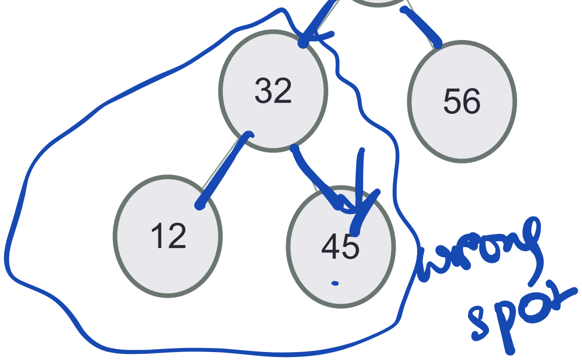
B.



C.

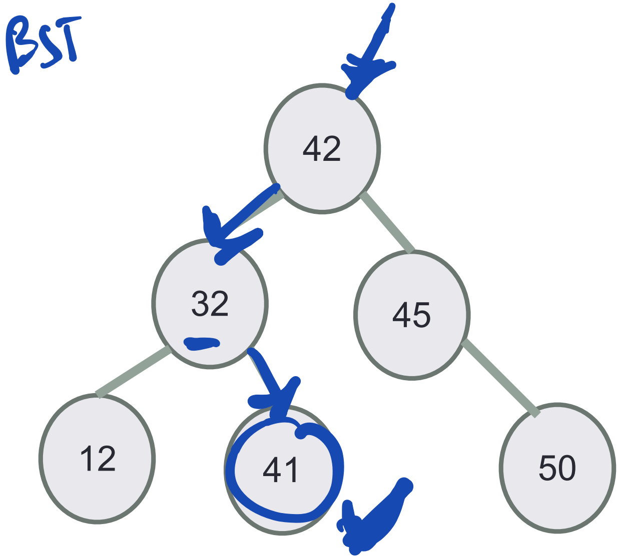


D.

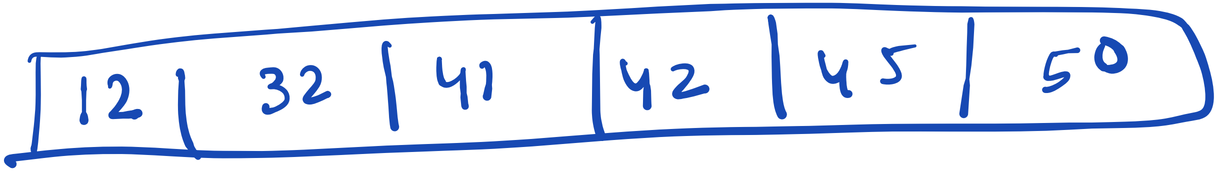


E. More than one of these

BSTs allow efficient search!



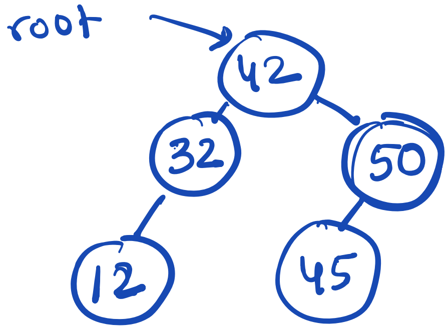
- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x



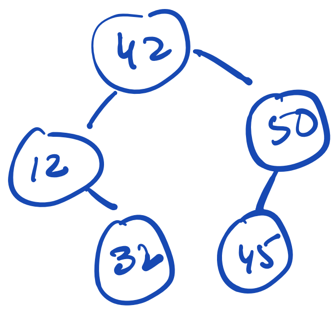
Search for 41, then search for 53

42, 50, 45, 32, 12

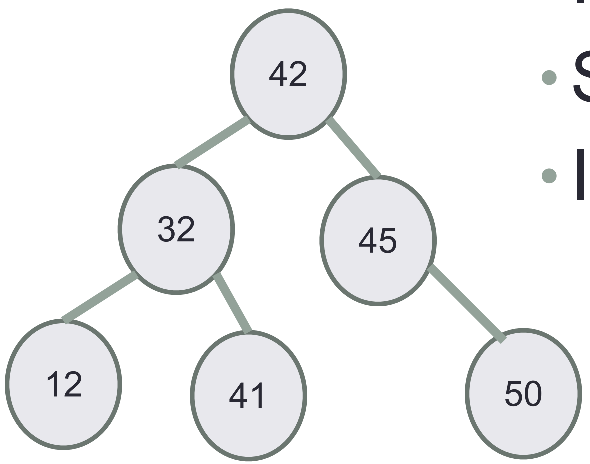
root [/]



42, 12, 32, 50, 45

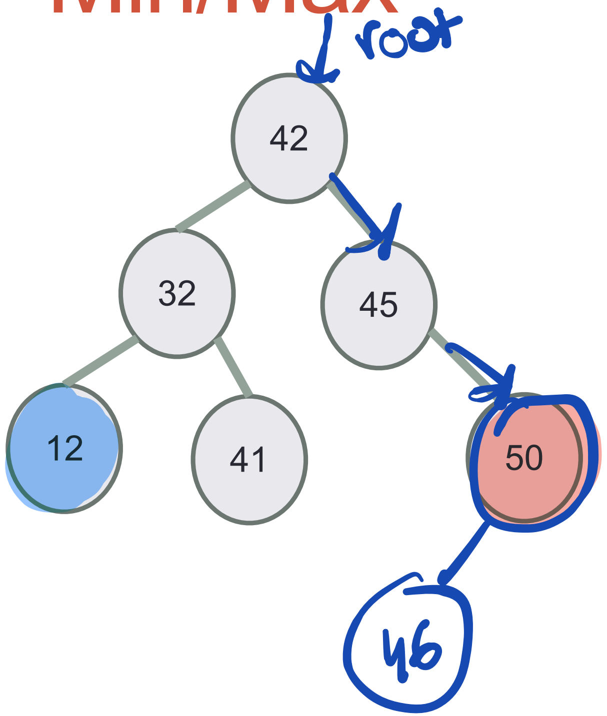


Insert



- 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

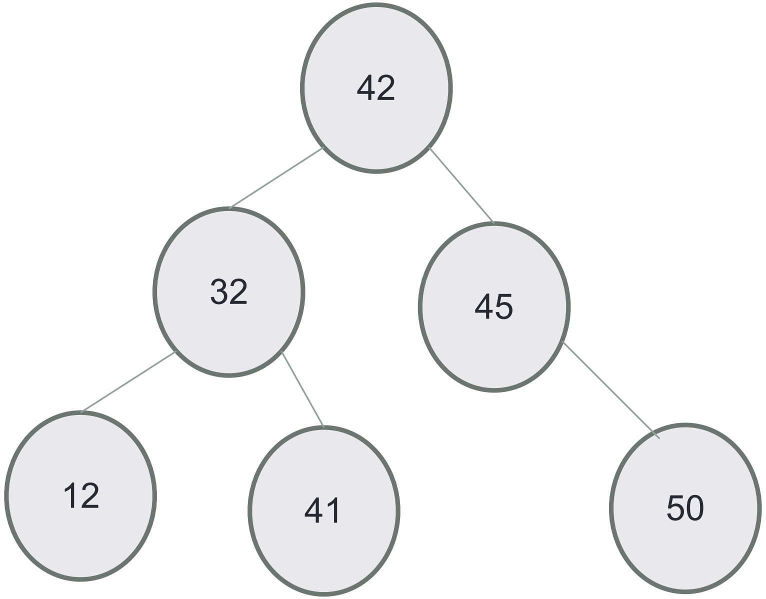
B. Follow right child pointers from the root, until a node with no right child is encountered, return that node's key

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

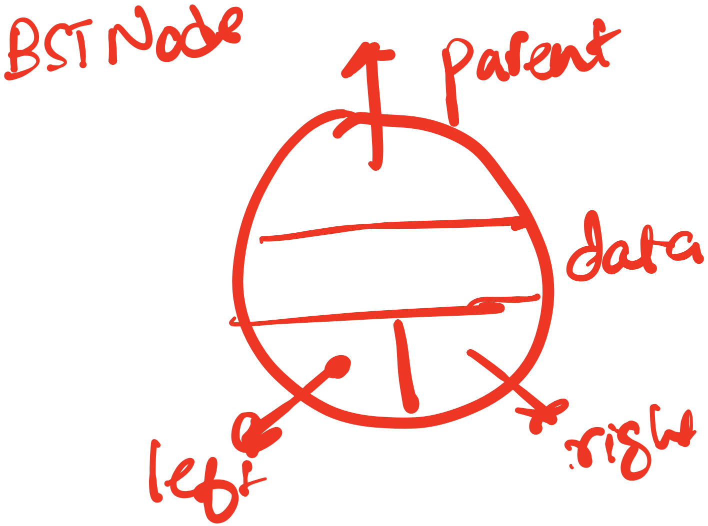
min

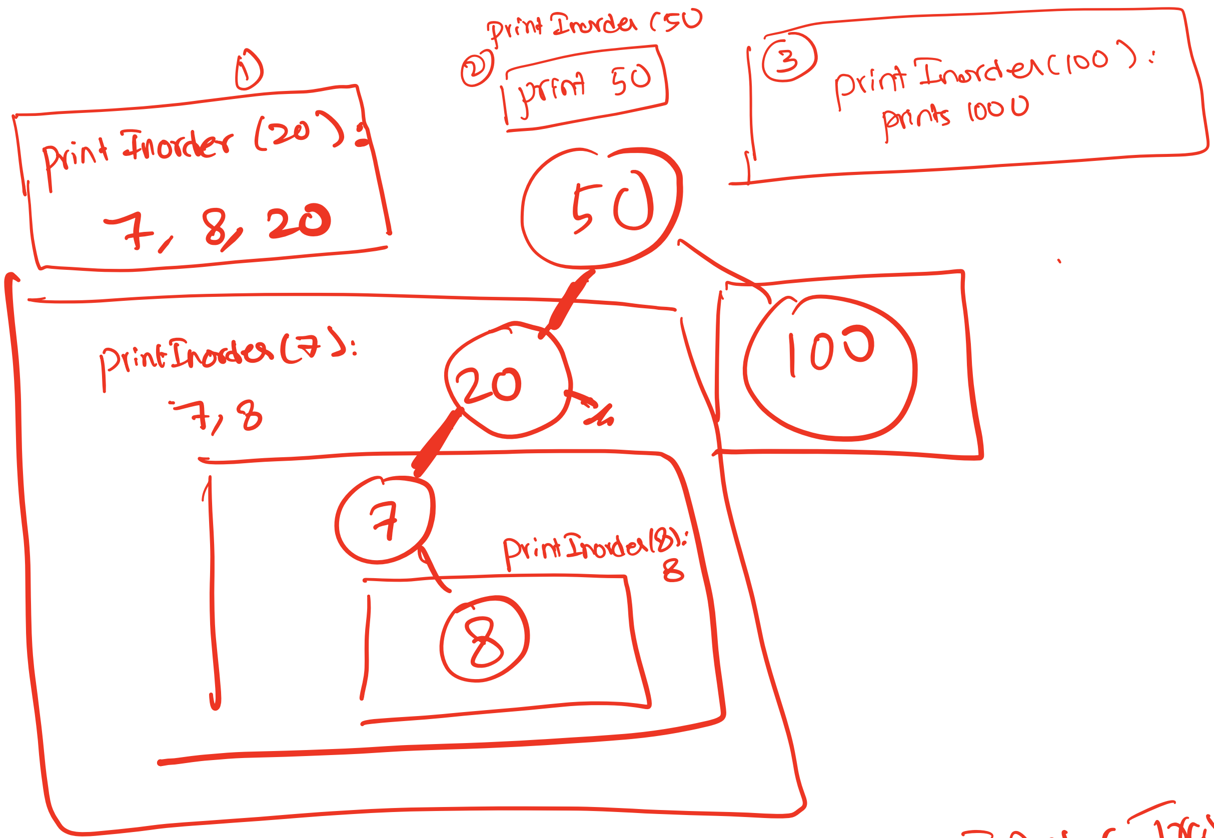
Define the BST ADT

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



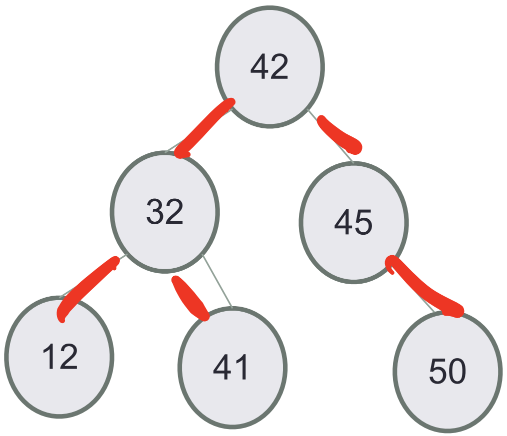
```
class BSTNode {

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

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





Tracing recursive calls in an InOrder Traversal

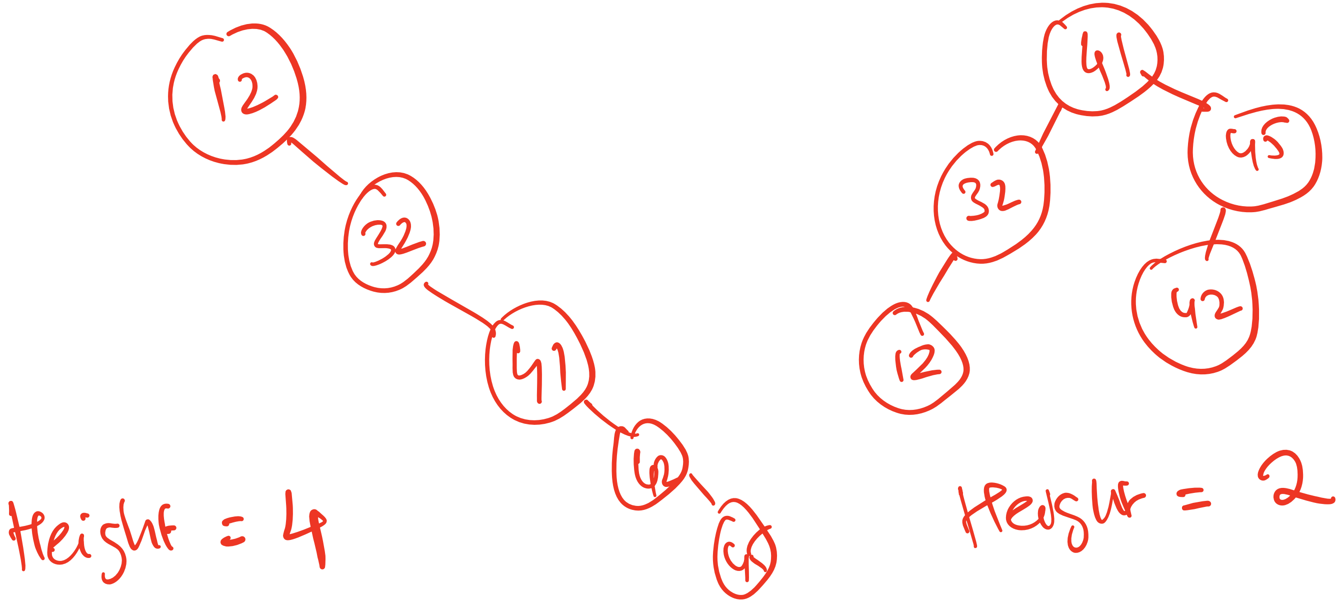
In order traversal: print elements in sorted order



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)

- 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

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