BINARY SEARCH TREES

Problem Solving with Computers-II

Frequency

AB

Sorted arrays, Balanced Binary Search Trees

arr | ~~2~~ | 10 | 15 | 500 | 600 |

N: number of elements

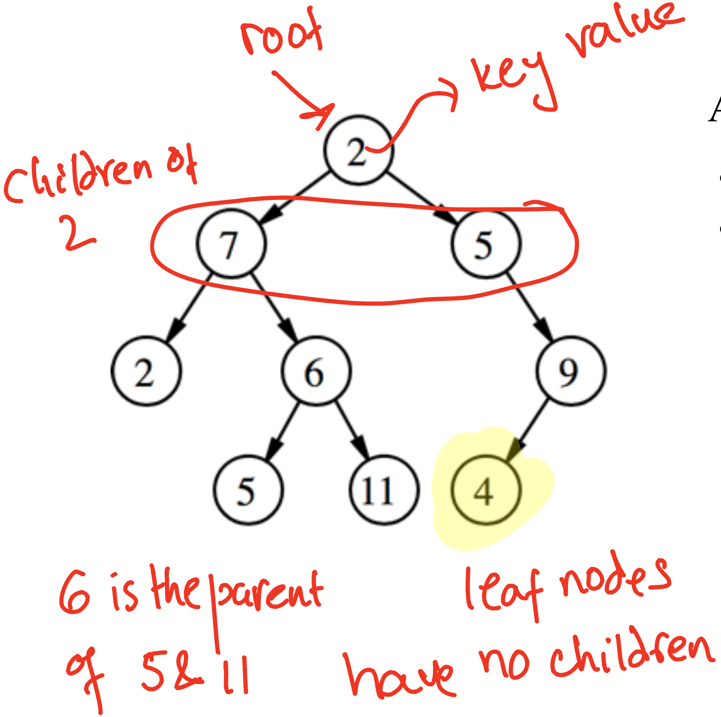
Operations	Sorted Array	Balanced BST
Min	O(1)	
Max	O(1)	
Successor	O(1)	
Predecessor	O(1)	
Search	O(log N)	
Insert	O (N)	
Delete	O (N)	
Print elements in order	O(N)	

O(log N)

O(N)

Assume Binary Search

Trees

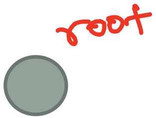


A tree has following general properties:

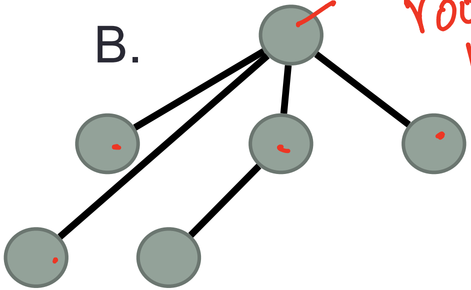
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*

Binary tree: Each node has at most 2 children

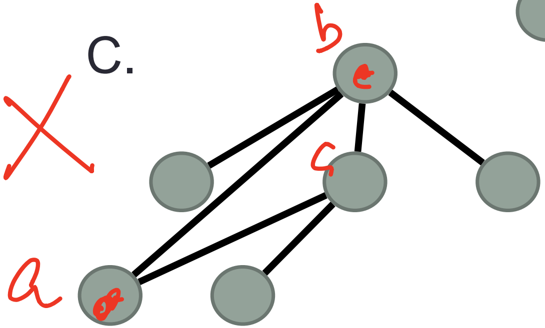
Which of the following is/are a tree?

A. 

(example of a single node tree

B. 

root has 4 children but that's okay in a generic tree (not a binary tree)

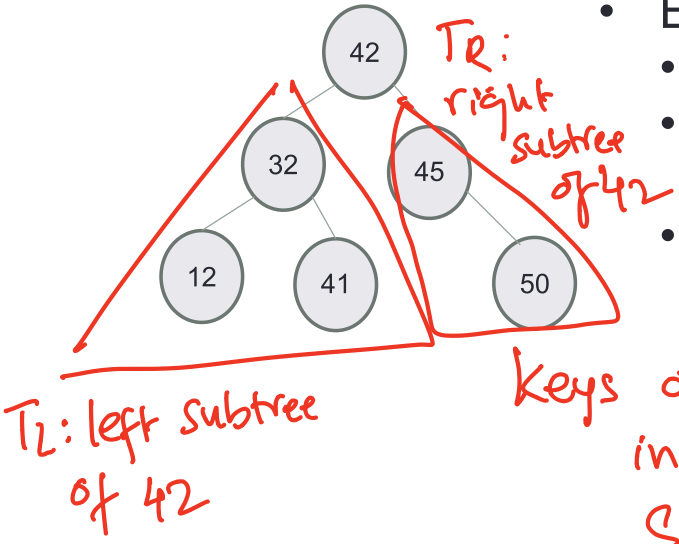
C. 

a has two parents,

D. A & B

E. All of A-C

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

Keys of all nodes in its left subtree (T_L) $\leq k <$ keys of all nodes in its right subtree (T_R)

Do the keys have to be integers?

No!

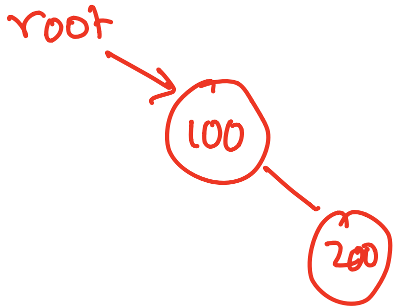
A node in a BST

```
class BSTNode {

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

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

root = new BSTNode(100);

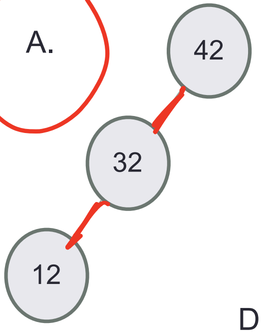
root → right = new BSTNode(200);

root → right → parent = root;

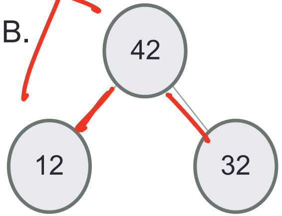


Which of the following is/are a binary search tree?

A.

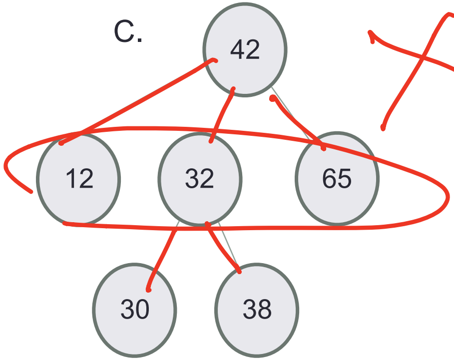


B.



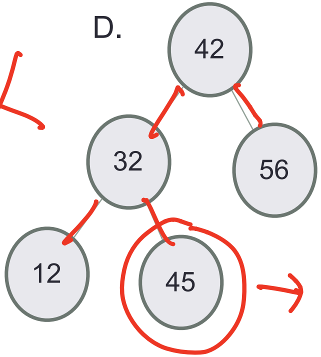
32 < 42

C.



Not a binary tree

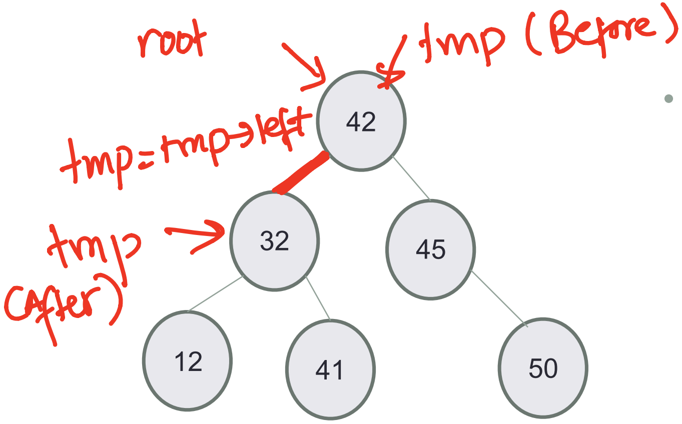
D.



→ should not be in the left subtree of 42

E. More than one of these

BSTs allow efficient search!



- Start at the root; trace down a path

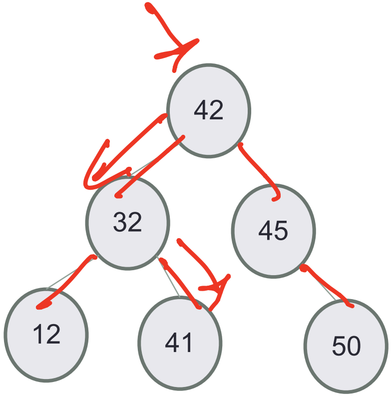
  by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x

```
BSTNode * tmp = root;
if (tmp->data == key)
        return true;
else if (tmp->data > key)
        tmp = tmp->left;
else
   tmp = tmp->right
```

Search for 41, then search for 53

Notice how we never search the right subtree of 42 when looking for 41!

Search



How many edges need to be traversed to search for 40?

A. One
B. Two
C. Three
D. Four

} Both are acceptable depending on your implementation

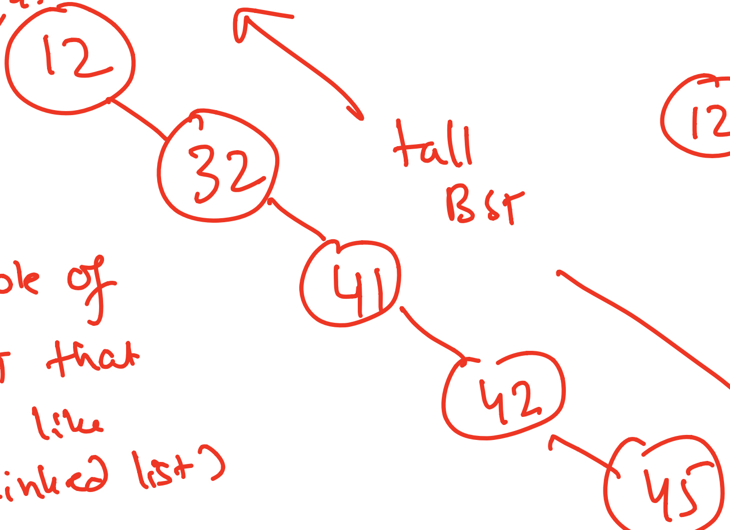
The main point is that the complexity of search depends on the number of times we need to traverse edges during search. → depends on the value we are looking for structure of the tree

How fast is the BST search algorithm?

The structure of the BST depends on the order in which keys are inserted.

- Many different BSTs are possible for the same set of keys
- Examples for keys: 12, 32, 41, 42, 45

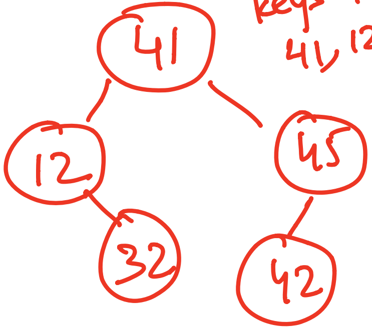
keys inserted in the order 12, 32, 41, 42, 45



tall BST

(example of a BST that looks like a linked list)

keys inserted in the order 41, 12, 32, 45, 42



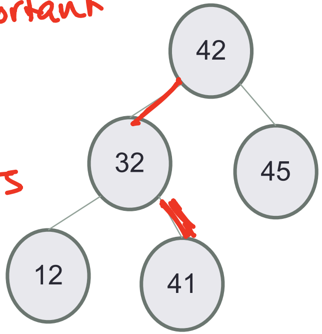
short BST

(shorter BSTs are faster to search)

How fast is BST search algorithm?

The complexity of search depends on the HEIGHT of the BST!

Height is an important attribute of the structure of the tree that affects the complexity of many operations



The height of this tree is 2.

Height of a node: the height of a node is the number of edges on the longest path from the node to a leaf
Height of a tree: the height of the root of the tree
Height of this tree is 2.

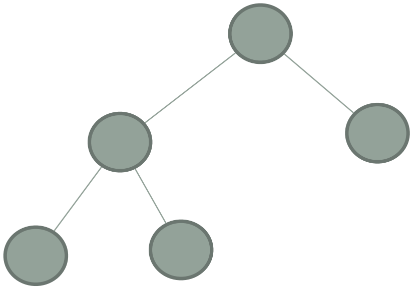
Worst case analysis

Are binary search trees *really* faster than linked lists for finding elements?

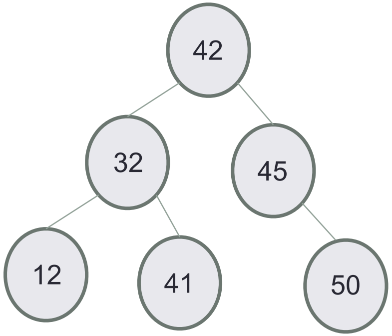
- A. Yes
- B. No

Worst case BST looks like a linked list

Search O(N)



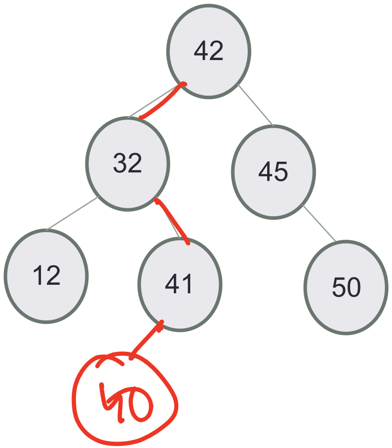
Balanced BSTs



BST with height = O(log N)

number of nodes

Insert



- Insert 40
- Search for the key — $O(H)$
- Insert at the spot you expected to find it

  constant time

Running Time $= O(H) + O(1)$

$= O(H)$

Max

O(H)

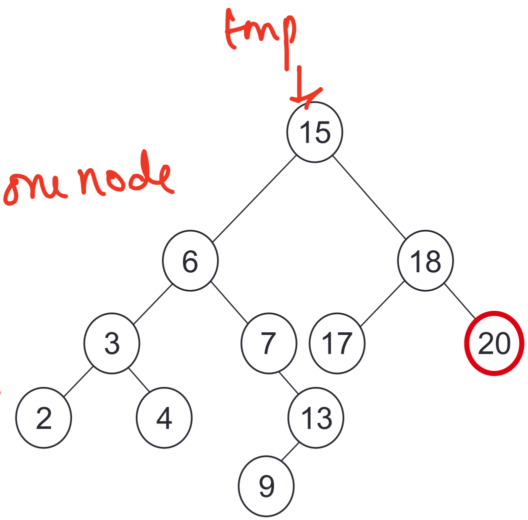
Goal: find the maximum value in a BST

Following right child pointers from the root, until a leaf node is encountered

// Precondition a tree with atleast one node

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

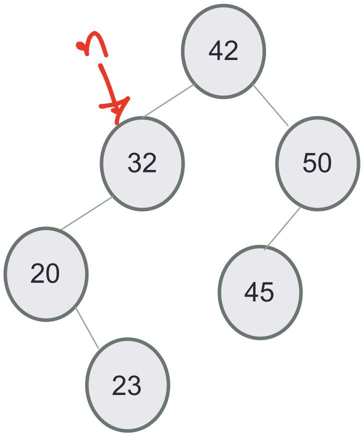
```
BSTNode * tmp = root;
while (tmp && tmp->right){
    tmp = tmp->right;
}
return tmp->data;
}
```



Maximum = 20

Keep going right!

Predecessor: Next smallest element

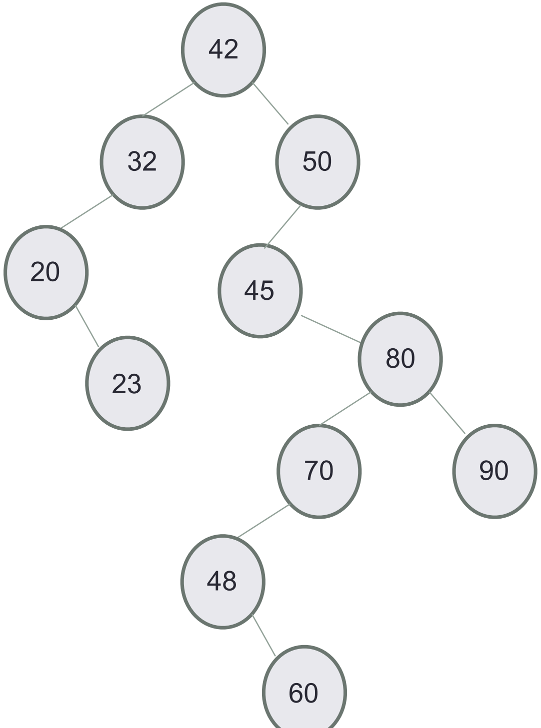


- What is the predecessor of 32? 23
- What is the predecessor of 45? 42

predecessor (BSTNode * n) :
// Assume k is the key of node N
if (n→left) { // n has a left subtree
// predecessor is the max node in the left subtree
} else {
// go up the tree until you find a node with a smaller key value
}

20 23 32 42 45 50

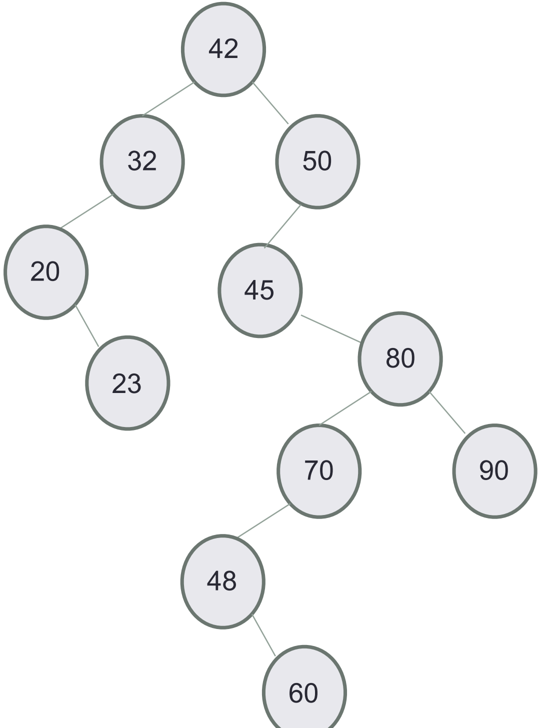
Successor: Next largest element



- What is the successor of 45?
- What is the successor of 48?
- What is the successor of 60?

Next lecture

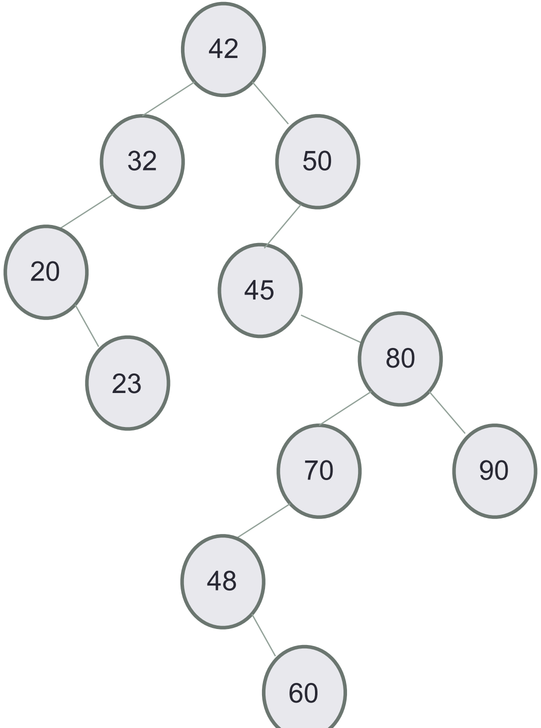
Delete: Case 1: Node is a leaf node



- Set parent's appropriate child pointer to null
- Delete the node

Next lecture

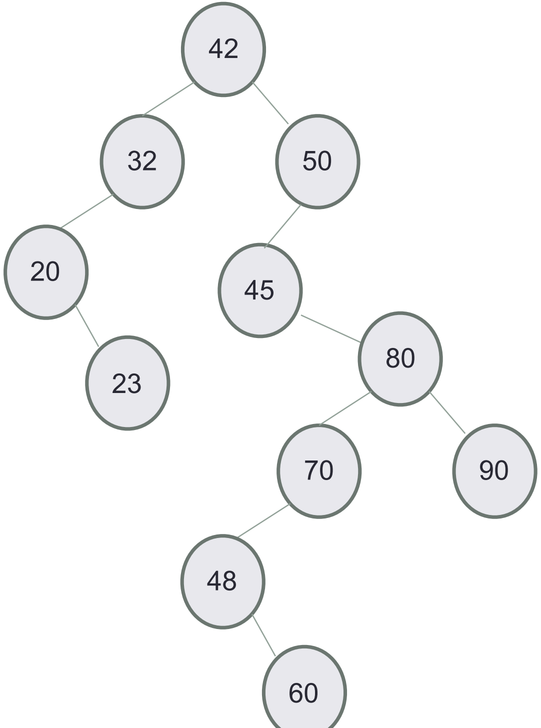
Delete: Case 2 Node has only one child



- Replace the node by its only child

Next lecture

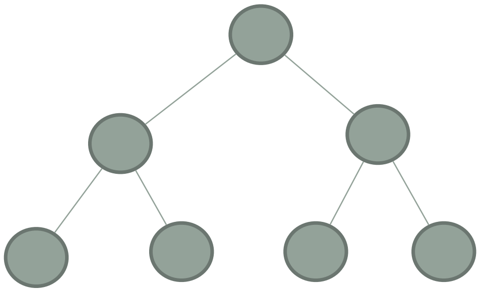
Delete: Case 3 Node has two children



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

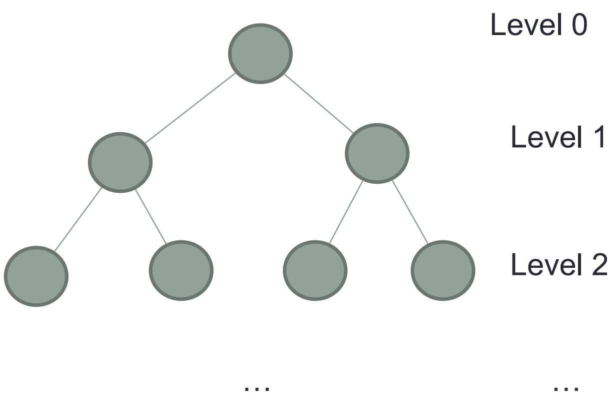
Next lecture

Completely filled BSTs



Next lecture

Relating H (height) and N (#nodes)
find is O(H), we want to find a f(N) = H

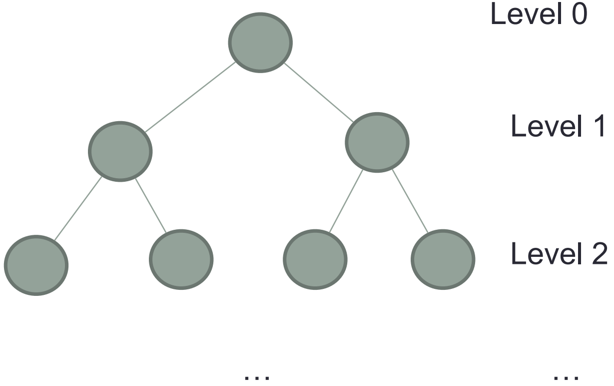


How many nodes are on level L in a completely filled binary search tree?

A. 2
B. L
C. 2*L
D. 2^L

Next lecture

Relating H (height) and N (#nodes)
find is O(H), we want to find a f(N) = H



Finally, what is the height (exactly) of the tree in terms of N?

Next lecture

And since we knew finding a node was O(H), we now know it is $O(\log_2 N)$

Sorted arrays, linked-lists, Balanced Binary Search Trees

Operations	**Sorted Array**	**Balanced BST**	**Linked list**
Min			
Max			
Successor			
Predecessor			
Search			
Insert			
Delete			
Print elements in order			

Next lecture