

VE281

Data Structures and Algorithms

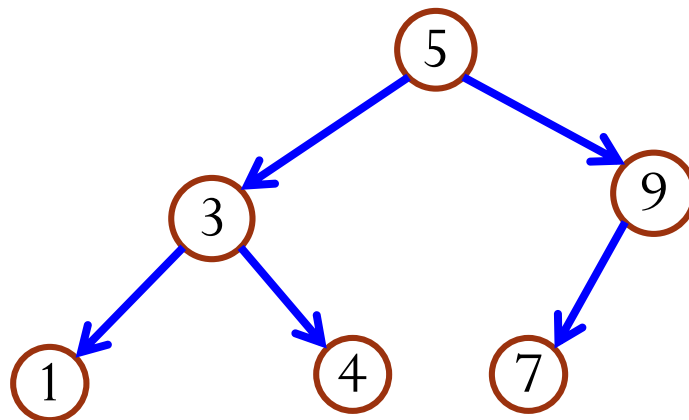
Binary Search Trees

Learning Objectives:

- Know what a binary search tree is
- Know how to do search, insertion, and removal for a binary search tree

Binary Search Tree

- A **binary search tree (BST)** is a binary tree with the following properties:
 - Each node is associated with a **key**.
 - A key is a value that can be compared.
 - **Assume**: all the keys are **distinct**.
 - The key of **any** node is greater than the keys of all nodes in its left subtree and smaller than the keys of all nodes in its right tree.



?

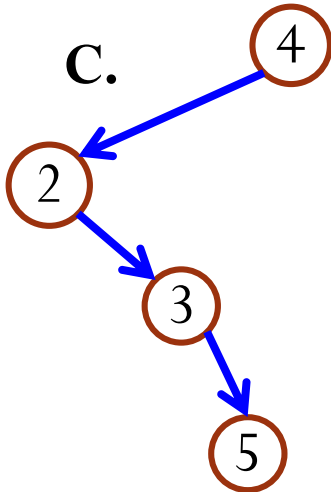
Which of the Following Trees Are BST?

- Select all the BSTs.

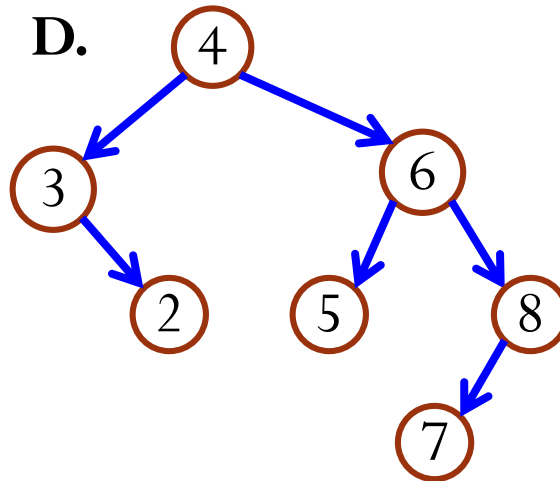
A. an empty tree

B. 5

C.



D.



Basic Binary Search Tree Operations

- A BST allows search, insertion, and removal by key.
 - The **average case** time complexities for these operations are $O(\log n)$.
 - **Average over all possible BSTs.**

Binary Search Tree

Search

```
node *search(node *root, Key k)
// EFFECTS: return the node whose key is k.
// If no matching node, return NULL.
```

- Procedure: Compare the search key with the key of the root
 - If they are equal, return the root.
 - If search key $<$ root key, search the left subtree.
 - If search key $>$ root key, search the right subtree.
 - Recursively applying the above procedure.

Binary Search Tree

Search

```
struct node {  
    Item item;  
    node *left;  
    node *right;  
};
```

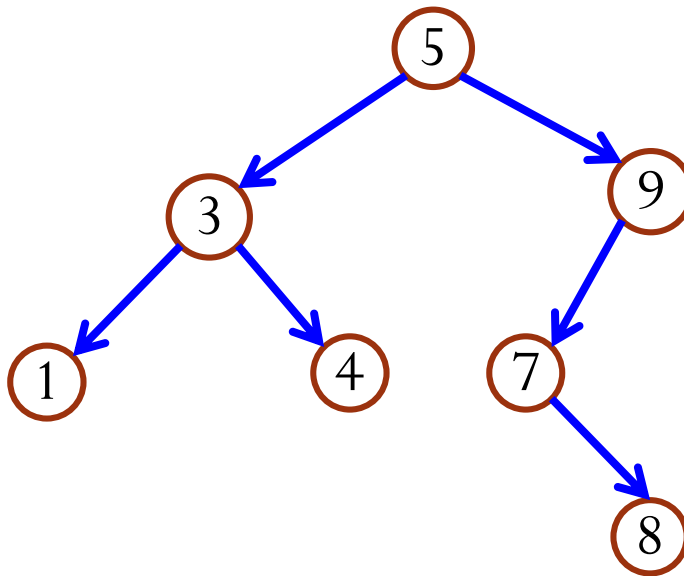
```
struct Item {  
    Key key;  
    Val val;  
};
```

```
node *search(node *root, Key k) {  
    if(root == NULL) return NULL;  
    if(k == root->item.key) return root;  
    if(k < root->item.key)  
        return search(root->left, k);  
    else return search(root->right, k);  
}
```

Binary Search Tree

Insertion

- Insertion inserts the item **as a leaf** of the BST.
- It inserts at a proper location in the BST, maintaining the BST properties.
 - **Pretend** we are searching the key.



Insert a node with key = 8

Binary Search Tree

Insertion

```
void insert(node *&root, Item item)
// EFFECTS: insert the item as a leaf,
// maintaining the BST property.
{
    if(root == NULL) {
        root = new node(item);
        return;
    }
    if(item.key < root->item.key)
        insert(root->left, item);
    else if(item.key > root->item.key)
        insert(root->right, item);
}
```

Question: why define root as the reference-to-pointer?

Question: what happens if the key is already in the BST?

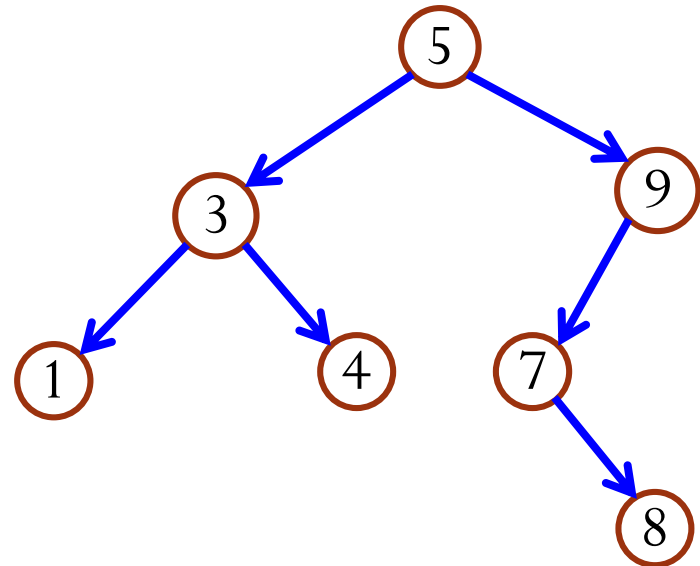
Binary Search Tree

Removal

reference to pointer

```
void remove(node *&root, Key k) {  
    if(root == NULL) return;  
    if(k < root->item.key) remove(root->left, k);  
    else if(k > root->item.key)  
        remove(root->right, k);  
    else { // root->item.key == k  
        // What to do when root->item.key == k?  
    }  
}
```

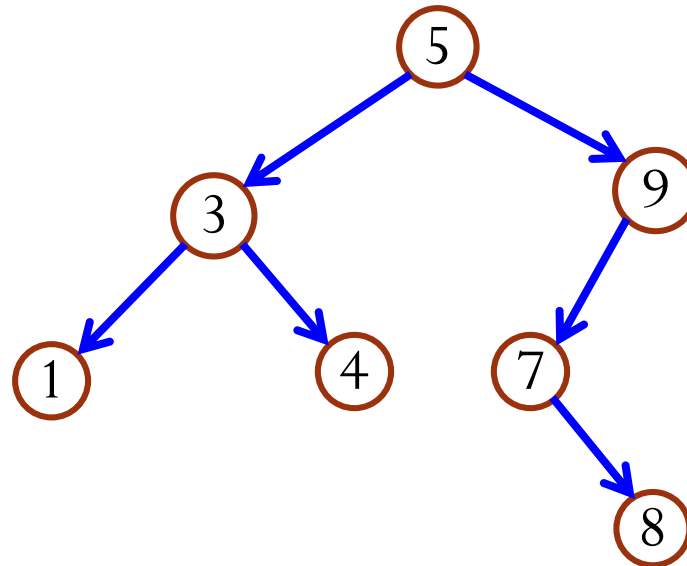
- How will you remove 8?
- How will you remove 9?
- How will you remove 5?



Binary Search Tree

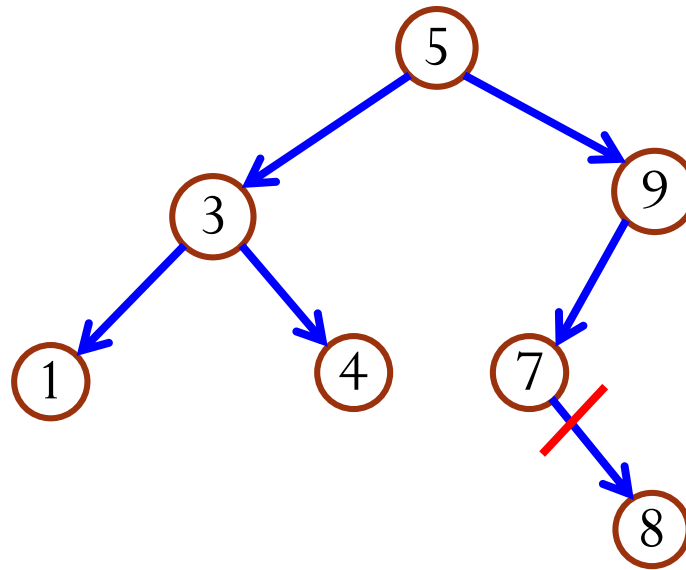
Removal

- We distinguish three cases:
 - Node to be removed is a leaf.
 - Node to be removed is a degree-one node.
 - Node to be removed is a degree-two node.



Remove A Leaf

- Remove node 8



Remove A Leaf

Code

```
else { // root->item.key == k
```

```
    if(isLeaf(root)) {  
        delete root;  
        root = NULL;  
    }
```

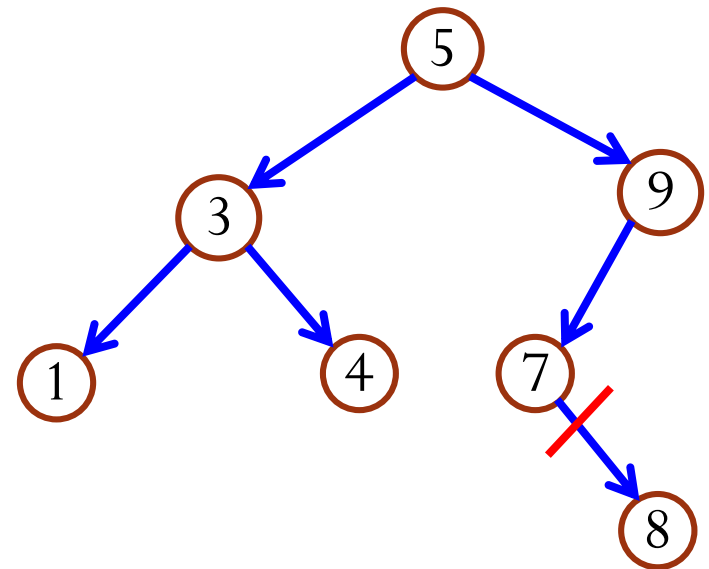
```
    else { // remove degree-one or two node
```

```
        ...
```

```
    }
```

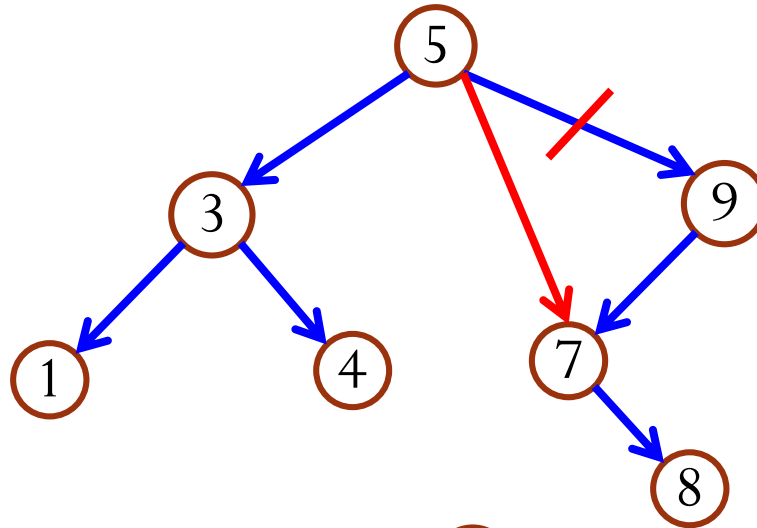
```
}
```

Note: **root** is a **reference to a pointer**, which could be its parent's **left** pointer or **right** pointer. Our code effectively changes that pointer to NULL.

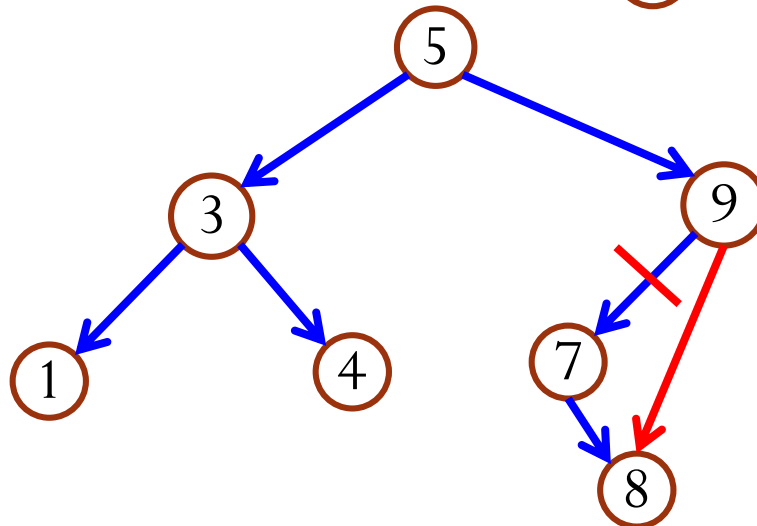


Remove A Degree-One Node

- Remove node 9



- Remove node 7

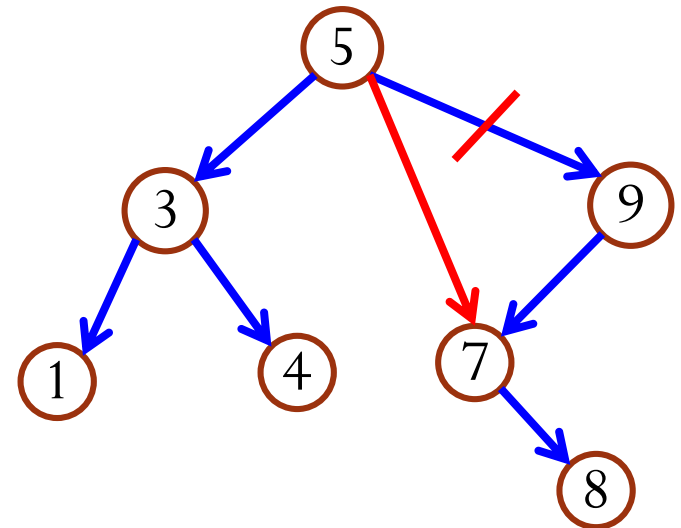


Remove A Degree-One Node

Code

```
else { // remove degree-one or two node
    if(root->right == NULL) { // no right child
        node *tmp = root;
        root = root->left;
        delete tmp;
    }
```

Note the order!

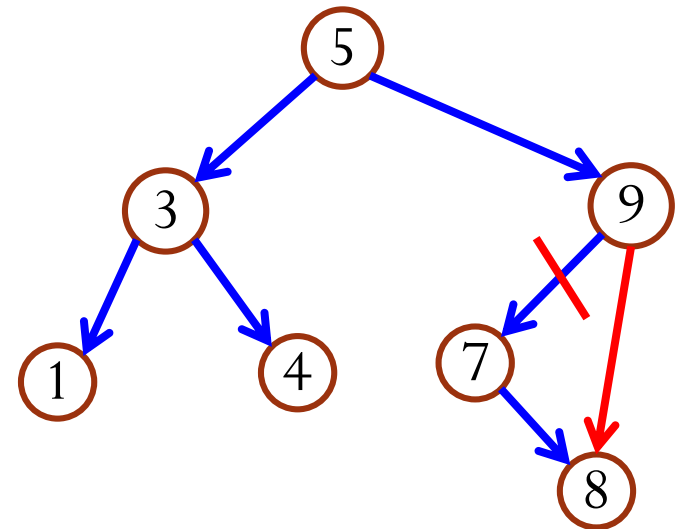


}

Remove A Degree-One Node

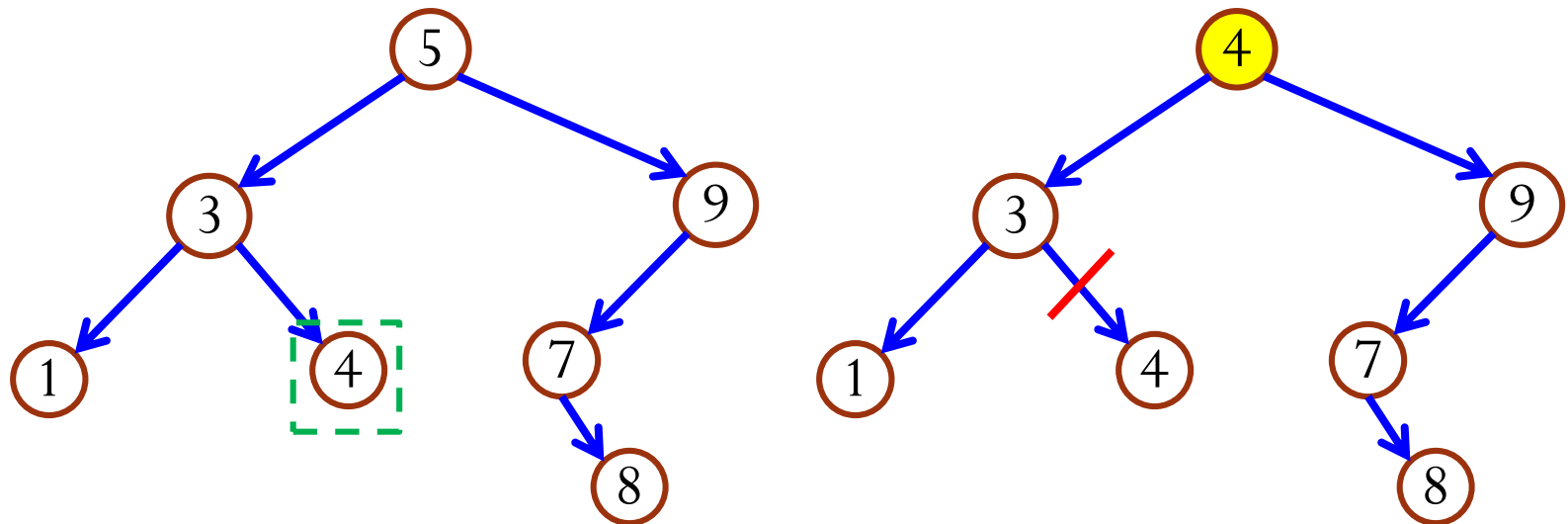
Code

```
else { // remove degree-one or two node
    if(root->right == NULL) { // no right child
        node *tmp = root;
        root = root->left;
        delete tmp;
    }
    else if(root->left == NULL) { // no left child
        node *tmp = root;
        root = root->right;
        delete tmp;
    }
    else {
        // remove degree-two node
    }
}
```



Remove A Degree-Two Node

- Remove node 5 How shall we do this?
- Idea: Replace with the largest key in the left subtree.
 - or replace with the smallest key in the right subtree.
- Claim: The largest key must be in a leaf node or in a degree-one node. Great! We know how to remove such a node!

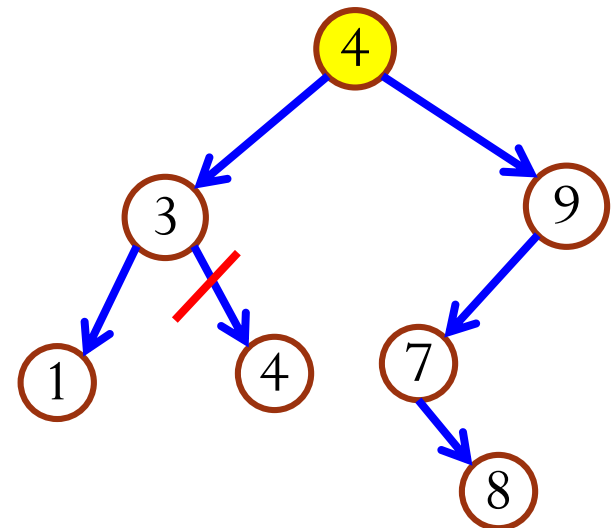


Remove A Degree-Two Node

Code

```
else { // remove degree-two node
    node *&replace = findMax(root->left);
    root->item = replace->item;
    node *tmp = replace;
    replace = replace->left;
    // both leaf and degree-one node are OK
    delete tmp;
}
```

```
node *&findMax(node *&root)
// REQUIRES: tree is non-empty.
// EFFECTS: return the reference
// to the left/right pointer of
// the parent of the node
// that has the largest key in
// the tree rooted at root
```

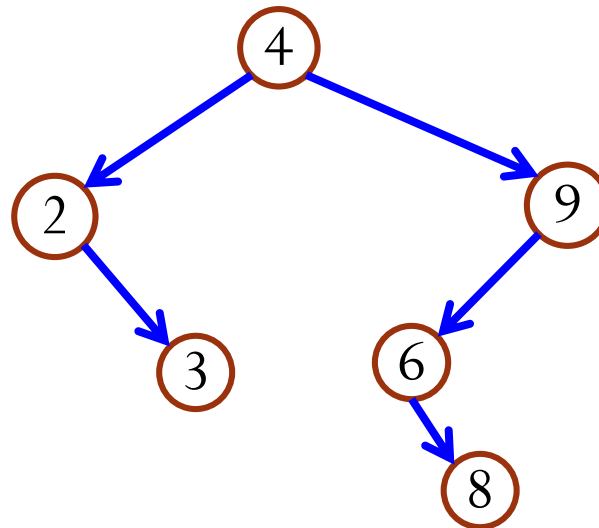


Remove A Degree-Two Node

Code

- How do you implement the function **findMax()**?

```
node *&findMax(node *&root) {  
    if(root->right == NULL) return root;  
    return findMax(root->right);  
}
```



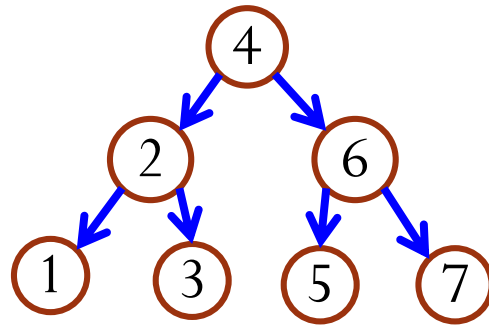
Removal of Binary Search Tree

Summary

- Node to be removed is a leaf.
 - Delete the node.
- Node to be removed is a degree-one node.
 - “Bypass” the node from its parent to its child.
- Node to be removed is a degree-two node.
 - Replace the node key with the largest key in the left subtree and remove the node with the largest key

Exercise

- Insert 4, 2, 6, 3, 7, 1, 5



- Delete 2, insert 9, delete 5, delete 1

