Binary Search Trees: Deletion

Semester 2, 2020

Kris Ehinger

How to delete an item?

- Deletion from a BST involves:
  - the **in-order predecessor**; or
  - the **in-order successor**

- In-order successor and in-order predecessor can be obtained from in-order **traversal**

Traversal

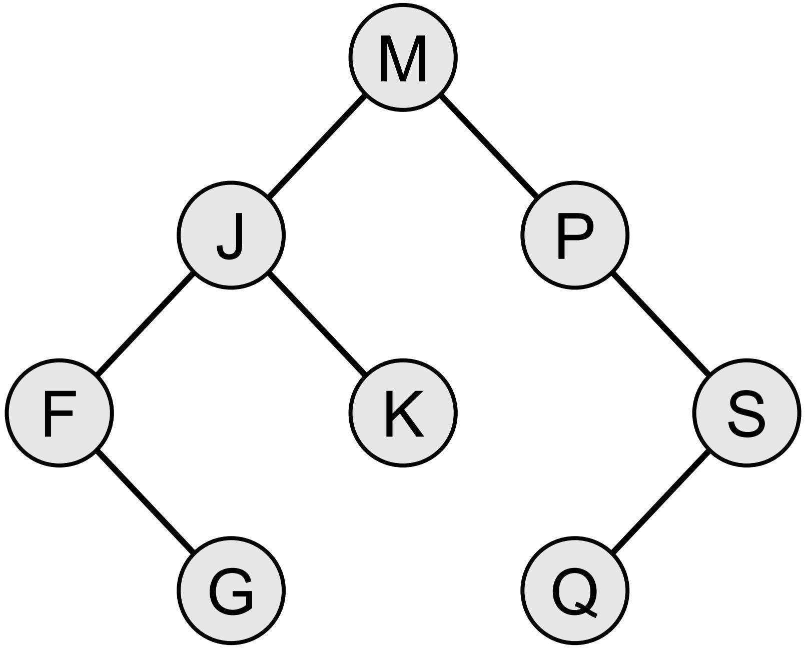
- **Traverse** = visit every node once
- Do something during the visit:
  - Print node value, or
  - Mark node as visited or
  - Check some property of node
- Use in any linked data structure
  - Tree
  - Graph
  - List

Recursive in-order traversal

```
traverse(struct node *t)
{
     if(t!=NULL)
     {
          traverse(t->left);
          visit(t);
          traverse(t->right);
     }
}
```

Recursive in-order traversal

- Example: Assume visit(t) prints the key. What is the output of recursive in-order tree traversal?



In-order traversal

- In a binary search tree, **in-order traversal** prints keys all nodes in key order
- Other ways to traverse a tree:
  - Pre-order traversal: do something at current node, then recurse on left and right nodes
  - Post-order traversal: recurse on left and right nodes, then do something at current node

Pre-order traversal

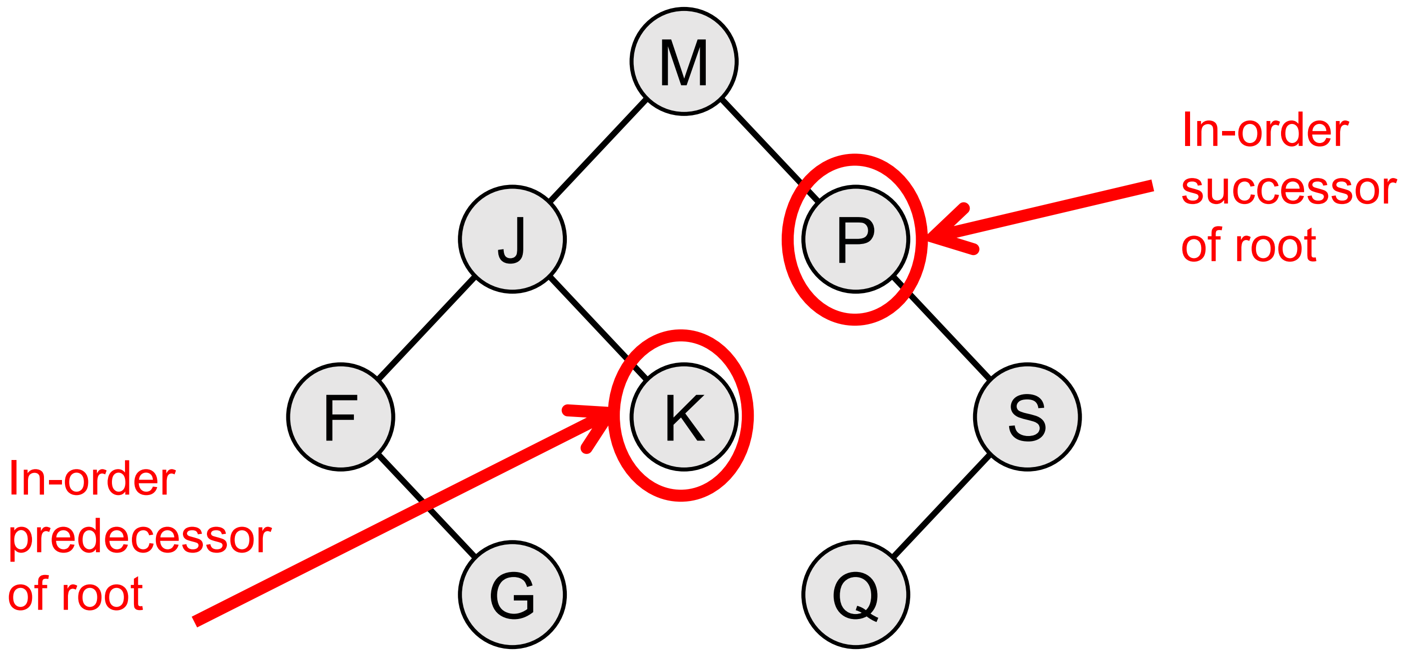
```
traverse(struct node *t)
{
    if(t!=NULL)
    {
        visit(t);
        traverse(t->left);
        traverse(t->right);
    }
}
```

Post-order traversal

```
traverse(struct node *t)
{
        if(t!=NULL)
        {
                traverse(t->left);
                traverse(t->right);
                visit(t);
        }
}
```

In-order traversal and deletion

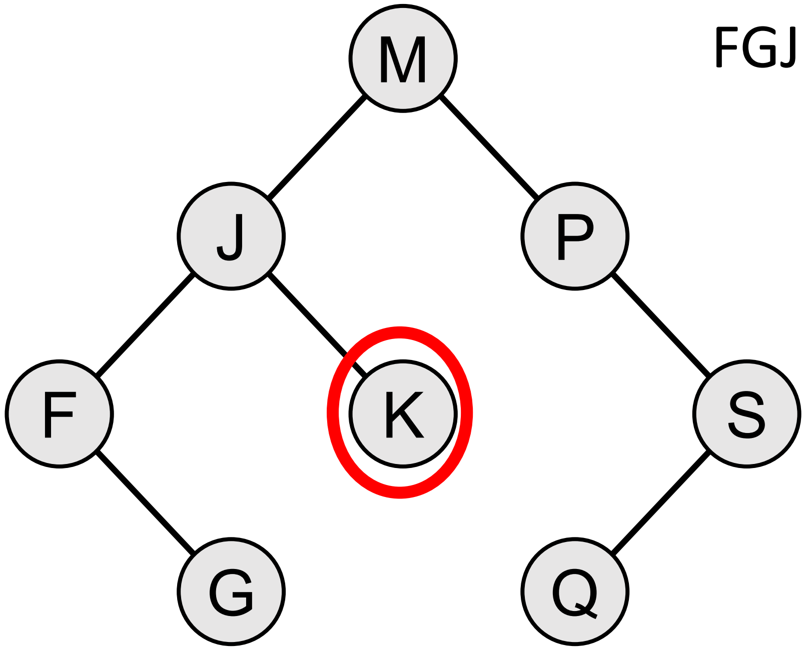
- In-order predecessor / successor = nodes immediately before / after current node in an in-order traversal:



In-order successor / predecessor

- In-order predecessor of root M is rightmost node of left subtree.

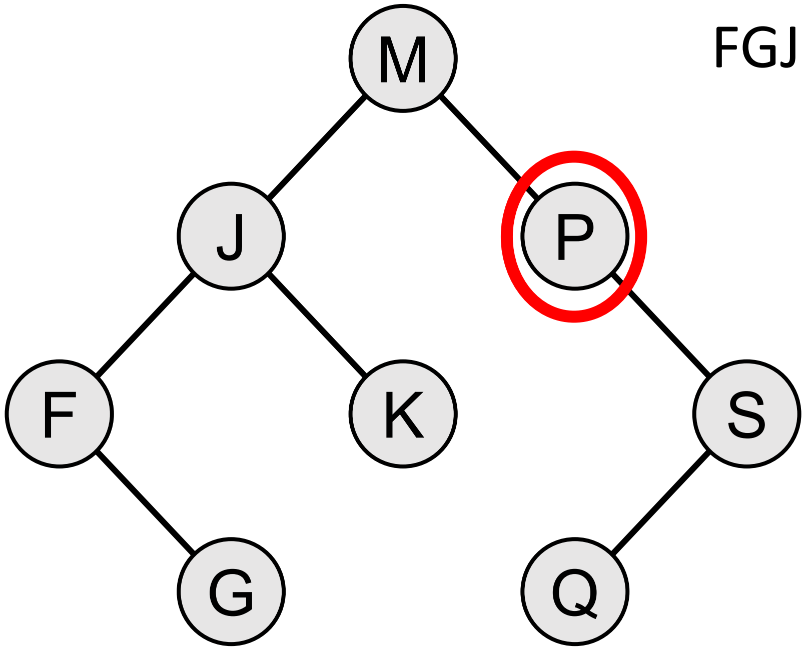
In-order traversal:
FGJKMPQS



In-order successor / predecessor

- In-order successor of root M is leftmost node of right subtree.

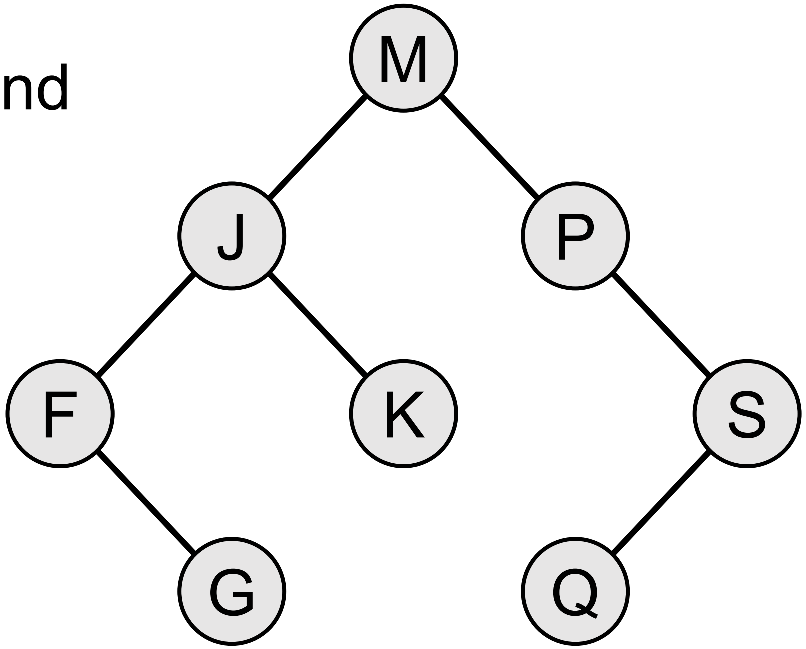
In-order traversal:
FGJKMPQS



In-order successor / predecessor

- Every node has a predecessor (just before) and a successor (just after):

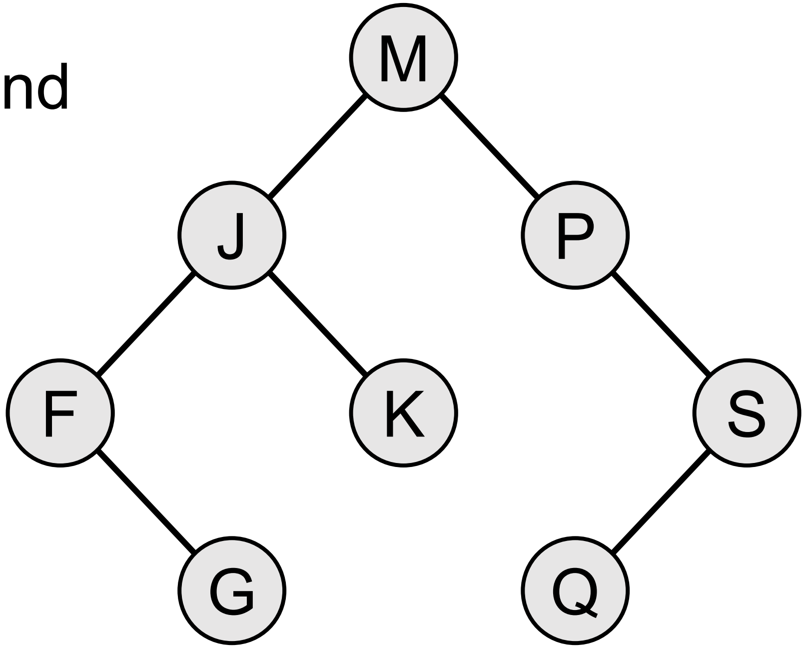
In-order predecessor and successor of node P?



In-order successor / predecessor

- Every node has a predecessor (just before) and a successor (just after):

In-order predecessor and successor of node K?



Deletion from binary search tree

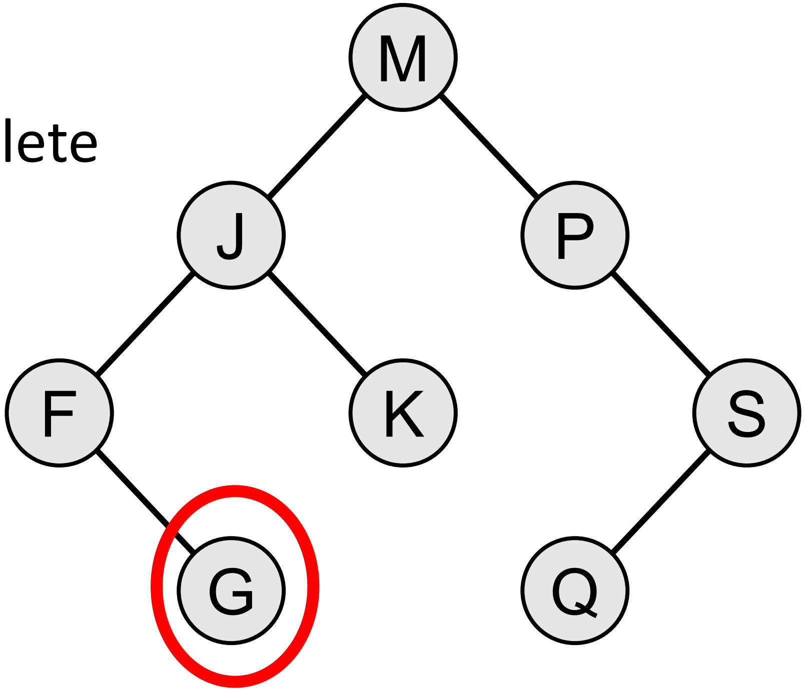
- Step 1: Find the node to be deleted
- Step 2: Delete it!

- Three cases for deletion:
  - Case 1: Node is a leaf
  - Case 2: Node has either a left or right child, not both
  - Case 3: Node has both a left child and a right child

Case 1: Node is a leaf

- Just delete the node

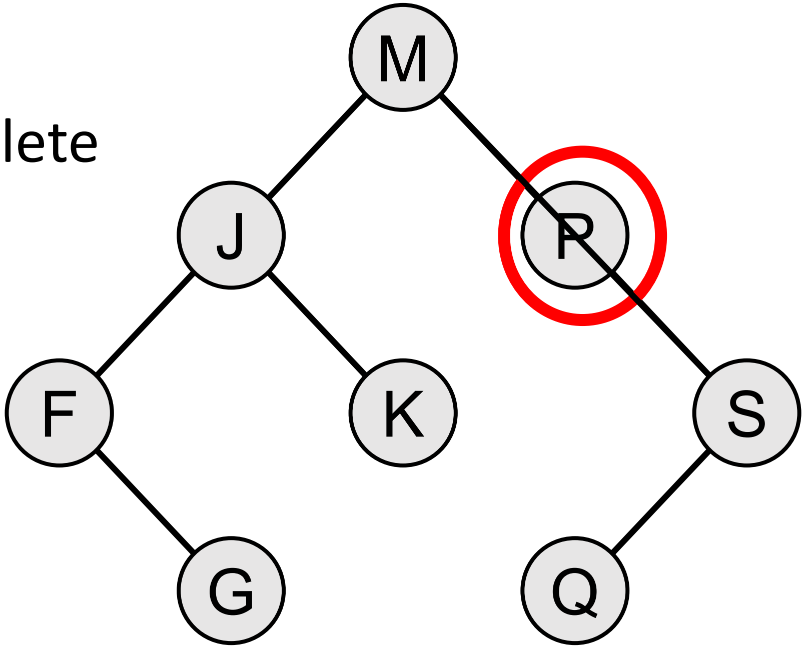
Example: Delete node G



Case 2: Node has *one* child

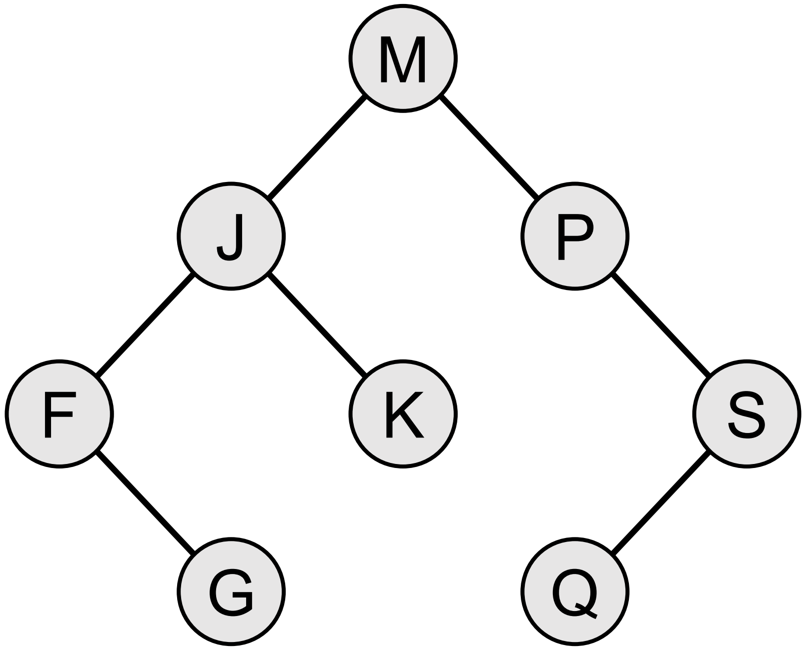
- Replace node with the child

Example: Delete node P



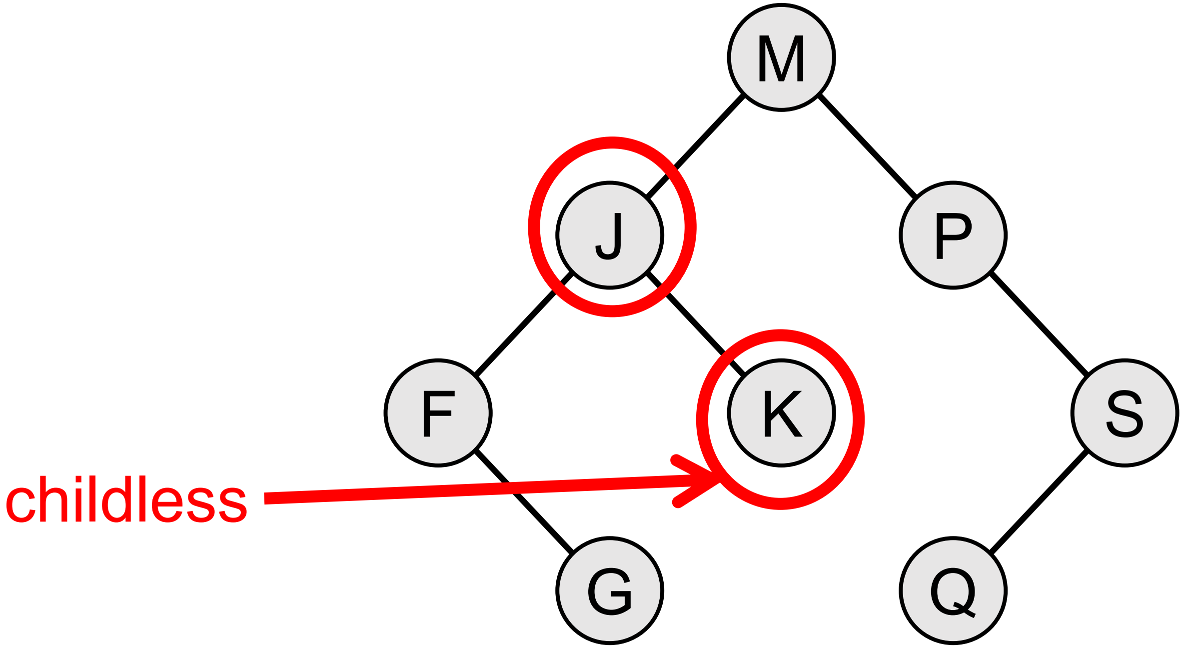
Case 3: Node has *two* children

- In this example: M, J



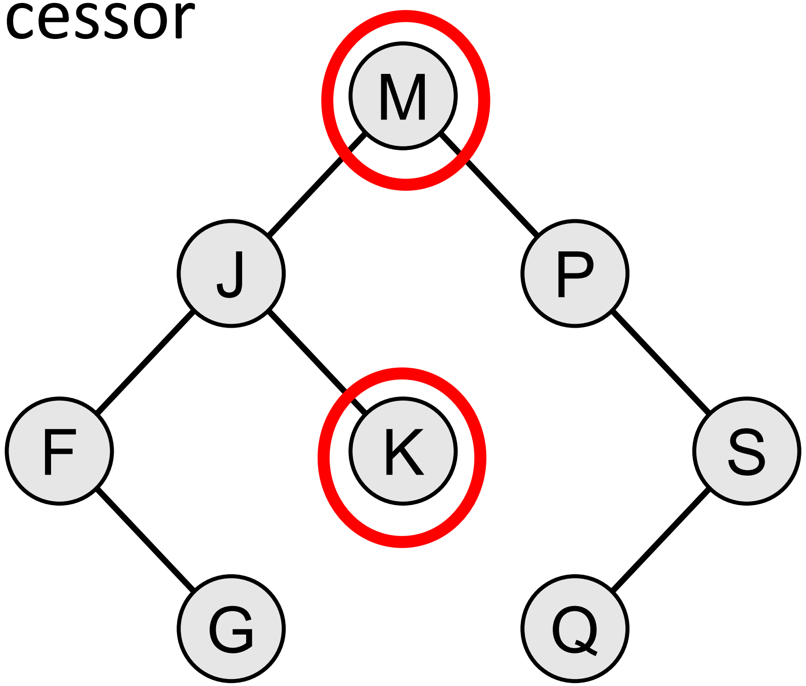
Case 3a: Node has *two* children

- But one of the children has no children (example: J)
- Replace node with the childless child



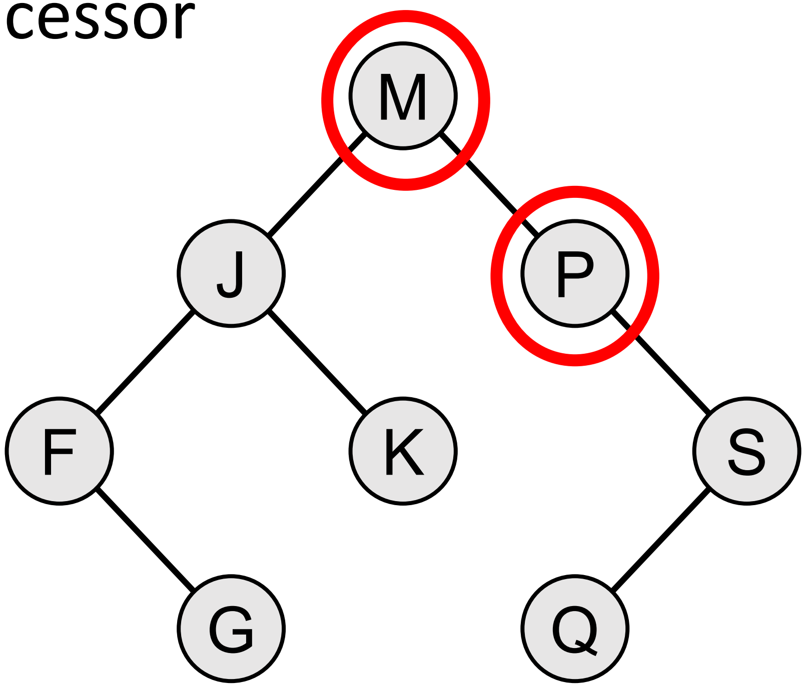
Case 3b: Node has *two* children

- And both children have children (example: M)
- Replace node with either in-order successor or in-order predecessor



Case 3b: Node has *two* children

- And both children have children (example: M)
- Replace node with either in-order successor or in-order predecessor



Deletion from binary search tree

- Step 1: Find the node to be deleted.
- Step 2: Delete it!

- Replace the deleted node with:
  - Case 1: Node is a leaf: nothing
  - Case 2: Node has either a left or a right child, but not both: the single child
  - Case 3: Node has both a left child and a right child: in-order predecessor or successor.

Deletion: time complexity

- Worst case:
  - Time to find the node: O(n)
  - Time to find in-order predecessor / successor: O(n)
  - Total time: O(n)
- Average case:
  - Time to find the node: O(log n)
  - Time to find in-order predecessor / successor: O(log n)
  - Total time: O(log n)