CS202 - Algorithm Analysis

Tree Algorithms - Module 3

Aravind Mohan

Allegheny College

March 21, 2023

Discussion Based On ...

Sedgewick 3.2, 3.3 BST and 2-3 Trees

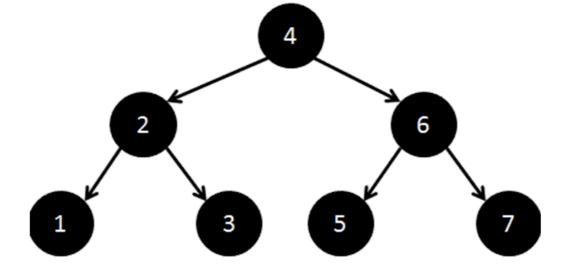
Tree Traversals

Processing the nodes in a tree exactly once in some order.

- Breadth-first traversal
- Depth-first traversal

Breadth First Traversal

- **Processing** the nodes in a tree in a level by level fashion.



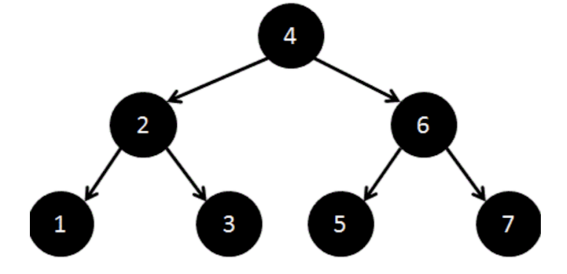
4,2,6,1,3,5,7

Also known as level ordered traversal - discussed previously

Depth First Traversal

- Pre-order
- Post-order
- In-order

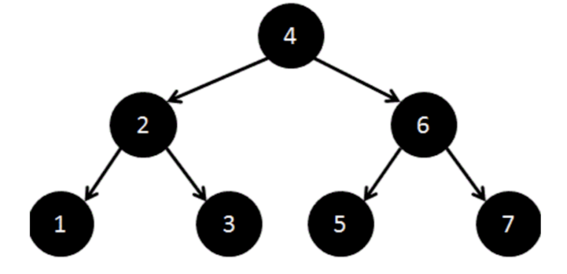
Pre-order Traversal



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

<root><left><right> (recursively)

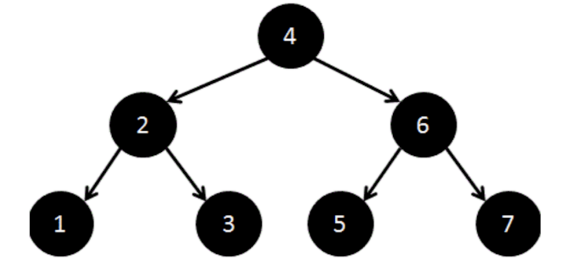
Post-order Traversal



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

<left><right><root> (recursively)

In-order Traversal



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

<left><root><right> (recursively)

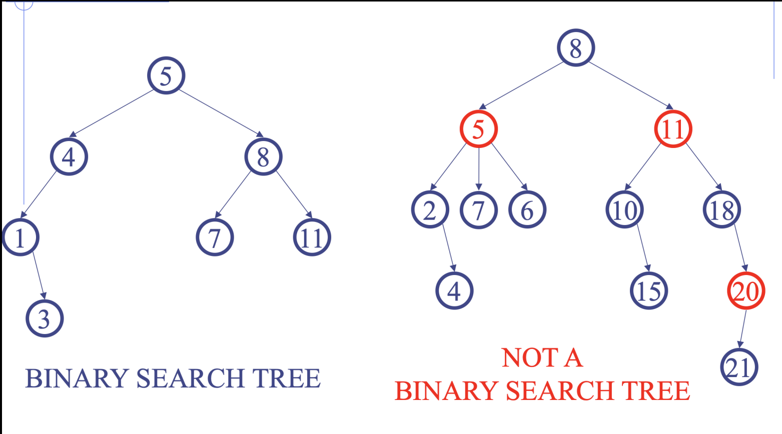
Binary Search Tree

- **Definition:**
  - A binary search tree is a binary tree in which all nodes in the left subtree of a node have lower values than the node.
  - All nodes in the right subtree of a node have a higher value than the node.

- A binary tree can be in any of the forms below:
  - complete or proper
  - in-complete
  - atmost complete

Refer previous lecture slides and notes for types of binary trees

BST Example


5
4
8
1
7
11
3
BINARY SEARCH TREE
8
5
11
2
7
6
10
18
4
15
20
21
NOT A
BINARY SEARCH TREE

BST Node Implementation

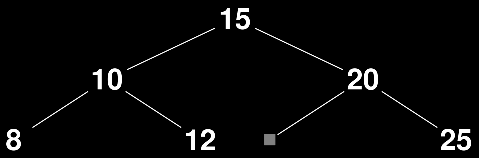
```
class BST:

    def __init__(self, val=None):
        self.left = None
        self.right = None
        self.val = val
```

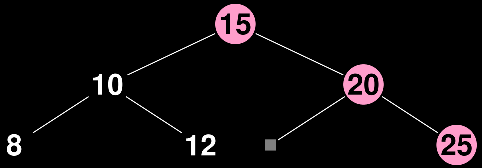
BST Search

- **Start:** from the root. If match found then return. Go to the next step otherwise.
- **Traverse:** left or right to continue search. If match found then return. Repeat this step otherwise, till exploring the leaf node.
- **Return:** not found if there is no match identified.

BST Search - Example

Search 25

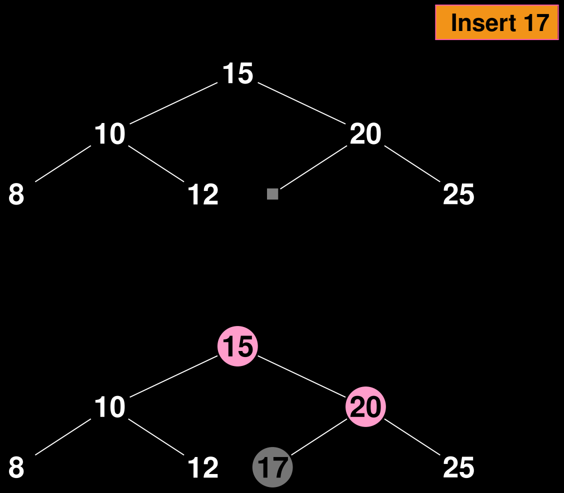




BST Insert

- Prioritize on no changes done to the position of the current nodes in the tree.
- The new node is inserted as a leaf node. But where? is it left sub-tree or right sub-tree?

BST Insert Example



BST Insert Implementation

```
def insert(self, val):
        if not self.val:
            self.val = val
            return

        if self.val == val:
            return

        if val < self.val:
            if self.left:
                self.left.insert(val)
                return
            self.left = BST(val)
            return

        if self.right:
            self.right.insert(val)
            return
        self.right = BST(val)
```

BST Delete

- **Case 1:** Deleting a leaf node.
- **Case 2:** Deleting an internal node with one child node.
- **Case 3:** Deleting an internal node with two child nodes.

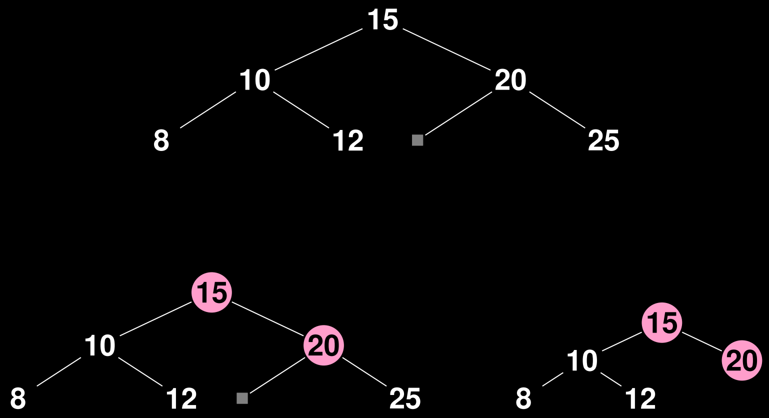
BST Case 1 Delete

- **Case 1:** Deleting a leaf node.
- **Find:** the node to be deleted by traversing left or right sub-tree.
- **Remove:** the node from the tree.

Simple Case

BST Case 1 Delete - Example

Delete 25

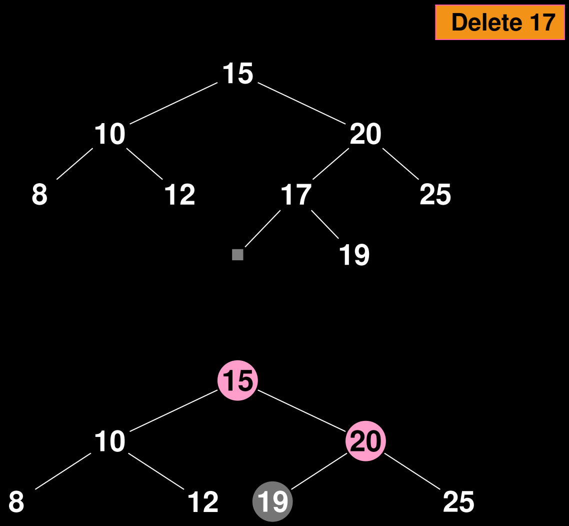


BST Case 2 Delete

- **Case 2:** Deleting an internal node with one child node.
- **Find:** the node to be deleted by traversing left or right sub-tree.
- **Point:** the parent of the node (to be deleted) to the node's only child.
- **Remove:** the node from the tree.

Not complicated

BST Case 2 Delete - Example



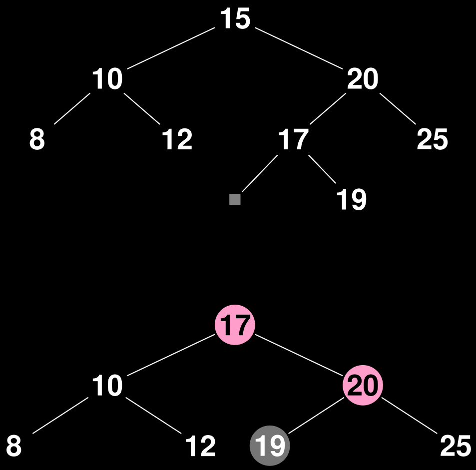
BST Case 3 Delete

- **Case 3:** Deleting an internal node with two child nodes.
- **Find:** the next biggest to the node (to be deleted). The next biggest is the node to the far left in the right sub-tree.
- **Replace:** the node to be deleted with the next biggest node.
- **Remove:** the node from the tree. After replacement, the node transformed from an internal node to a leaf node.
- **Inorder successor:** is the next biggest node.

Most interesting case

BST Case 3 Delete - Example

Delete 15



BST Analysis

Best and Average Case:

- $O(\log(n))$ - search, insert, and delete

Worst Case:

- $O(n)$ - search, insert, and delete

Worst case occurs only if the tree is completely unbalanced. That is, if the tree is either left or right skewed. There are ways to fix this and make it more balanced. (Later)

Are we ready to take up a challenge?

How do we find the lowest common ancestor in a BST?

Definition:

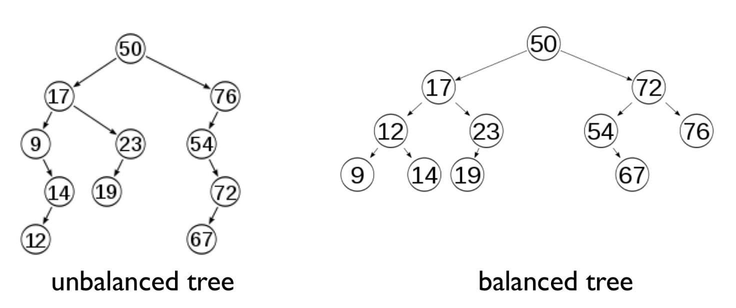
B-Tree is a self-balancing search tree.

Why?

- Faster search on external devices.
- Implement indexing feature in databases and filesystem.

Definition:

A tree in which the heights of subtrees are approximately equal or all terminal nodes are of same depth.



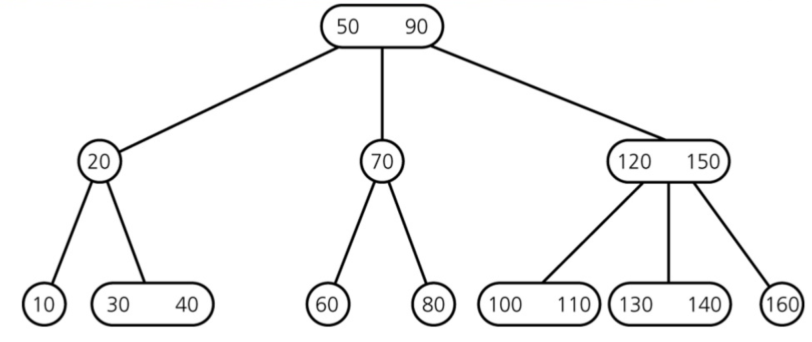
2-3 trees

- It is a **B-Tree** of order 3.

Properties

- Each node has either 1 or 2 keys.
- Each internal node has either 1 key with 2 children or 2 keys with 3 children.
- Leaf nodes has either 1 or 2 keys with no children.
- All leaves are at same level.
- Keys in a node are arranged in ascending order.

2-3 Tree Example



2-3 Tree Insert Algorithm

- **Step 1:** If the tree is empty, create a new node and insert the new key into the node.
- **Step 2:** Otherwise find the leaf node where the new key belongs.
- **Step 3:** If the leaf node has only one key, insert the new key into the node.
- **Step 4:** If the leaf node has more than two keys, split the node and promote the median of the three keys to the parent.
- **Step 5:** If the parent has more than two keys (upon promotion), continue to split and promote, form a new root if necessary.

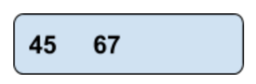
2-3 Tree Insert Example

- Construct 2-3 tree with the seqeunce of elements 45, 67, 35, 17, 9, 8, 4, and 50.

2-3 Tree Insert Example

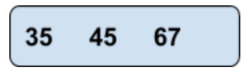
Insert 45

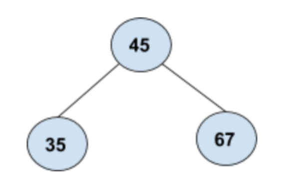
Insert 67



2-3 Tree Insert Example

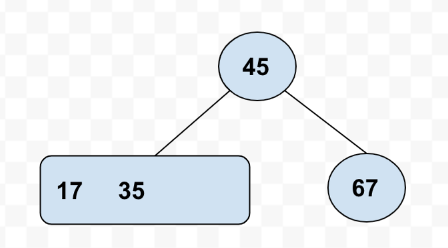
Insert 35





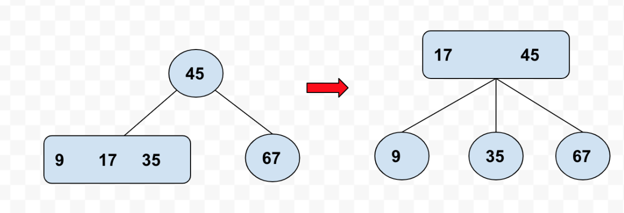
2-3 Tree Insert Example

Insert 17



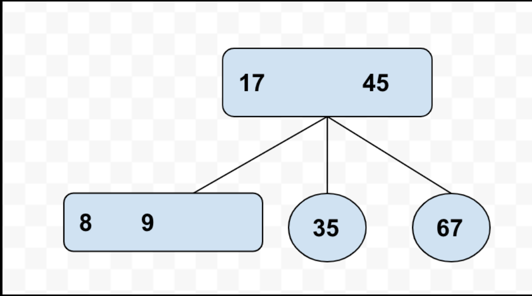
2-3 Tree Insert Example

Insert 9



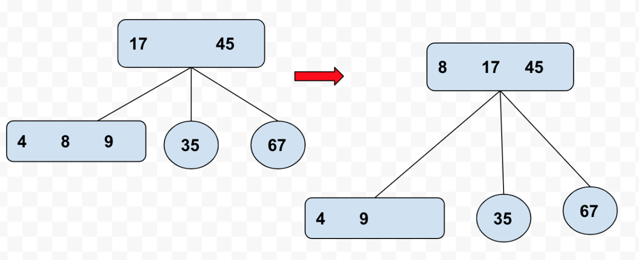
2-3 Tree Insert Example

Insert 8



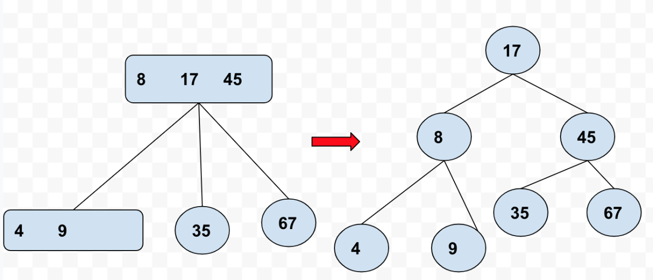
2-3 Tree Insert Example

Insert 4



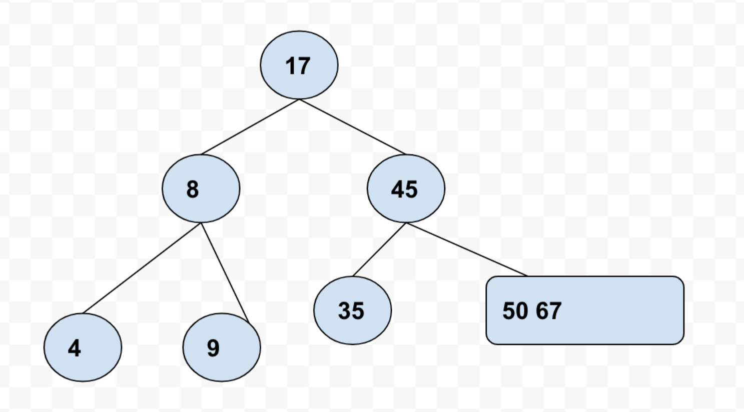
2-3 Tree Insert Example

Insert 4



2-3 Tree Insert Example

Insert 50



BST Worst case and how 2-3 Tree makes it better?

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

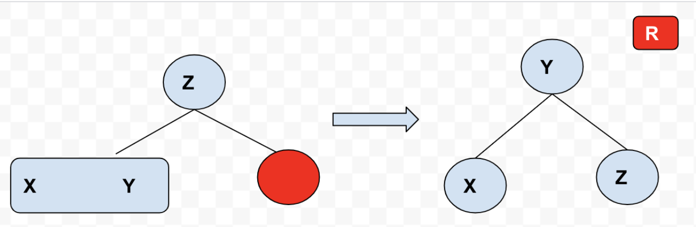
Try out?

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

2-3 Tree Delete Algorithm

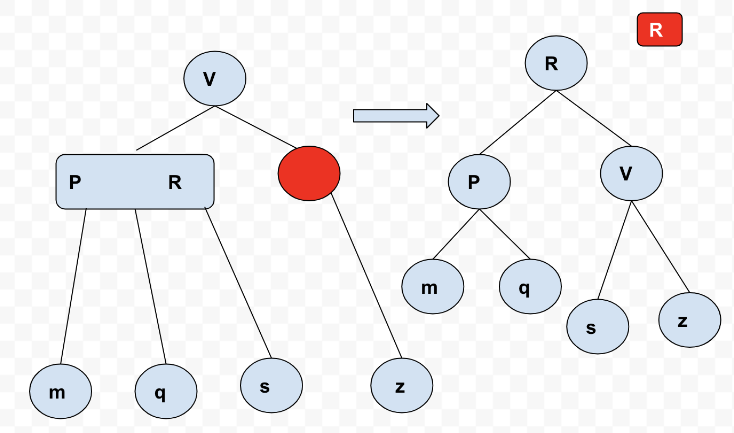
- **Step 1:** Conduct a search to locate the node $\mathbf{n(k_1)}$ with the key $\mathbf{k_1}$ to be deleted.
- **Step 2:** If the node $\mathbf{n(k_1)}$ is not a leaf node, then swap the key $\mathbf{k_1}$ with the in-order successor of $\mathbf{k_1}$.
- **Step 3:** If the node $\mathbf{n(k_1)}$ is a leaf node and contains two keys $\mathbf{k_1}$ and $\mathbf{k_2}$, then just delete $\mathbf{k_1}$.
- **Step 4:** If the node $\mathbf{n(k_1)}$ is a leaf node and contains only one key, $\mathbf{k_1}$, then try to redistribute nodes from siblings or merge node otherwise.

- **R1:** Delete leaf node with a sibling that has two keys. Redistribute keys between sibling and parent.



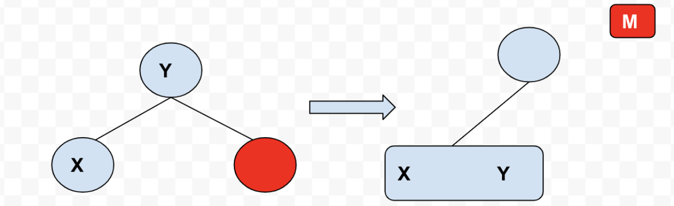
2-3 Tree Delete Cases

- **R2:** Delete internal node with no keys and sibling with two keys.



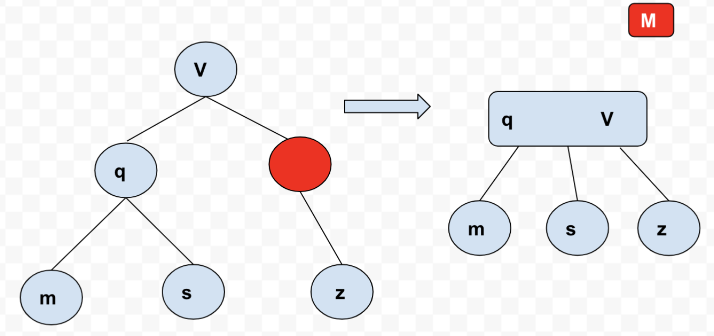
2-3 Tree Delete Cases

- **M1:** Delete leaf node with a sibling that has one key. Merge node by moving key from Parent to Sibling.



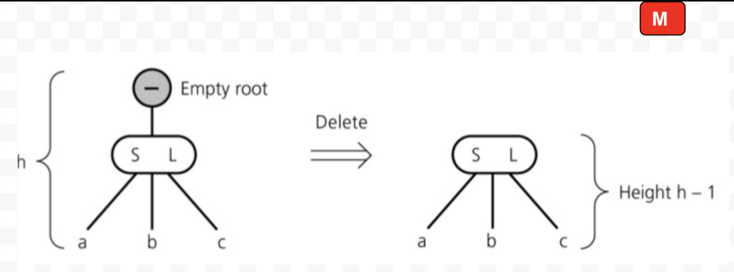
2-3 Tree Delete Cases

- **M2:** Delete internal node with no keys and sibling with one key. Merge node by moving key from Parent to Sibling and adopt child of the deleted node. Apply this rule upwards if needed.



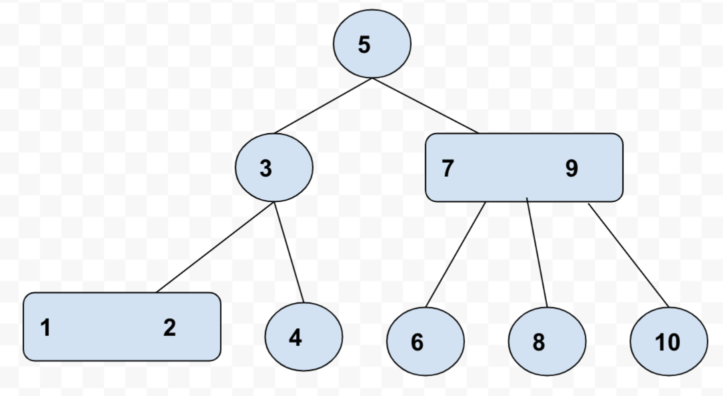
2-3 Tree Delete Cases

- **M3:** If merging process reaches root and the root is without a key, then delete root.



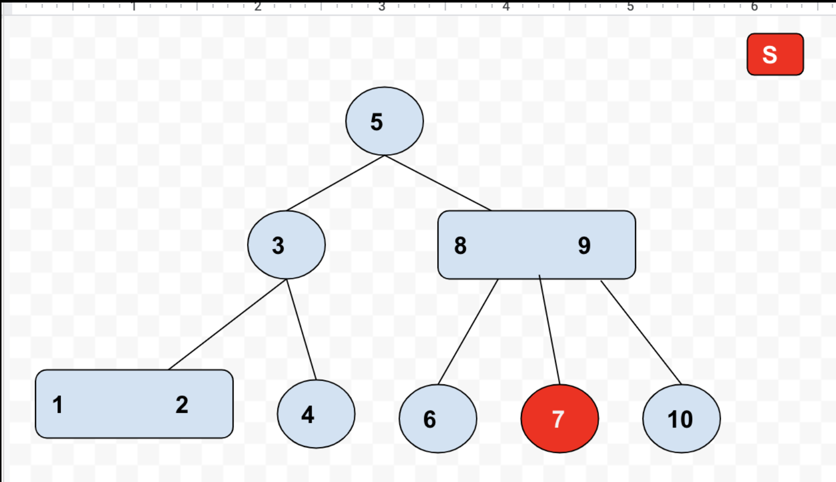
2-3 Tree Delete Example

- Delete 7, 10, and 8 from the 2-3 tree provided below:



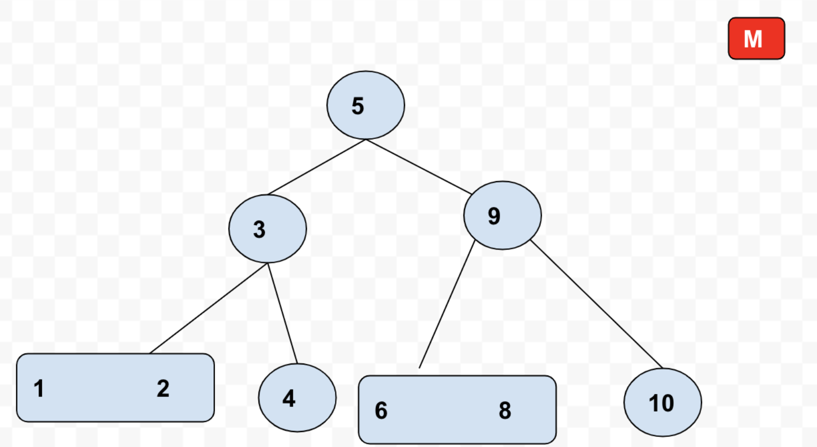
2-3 Tree Delete Example

Delete 7



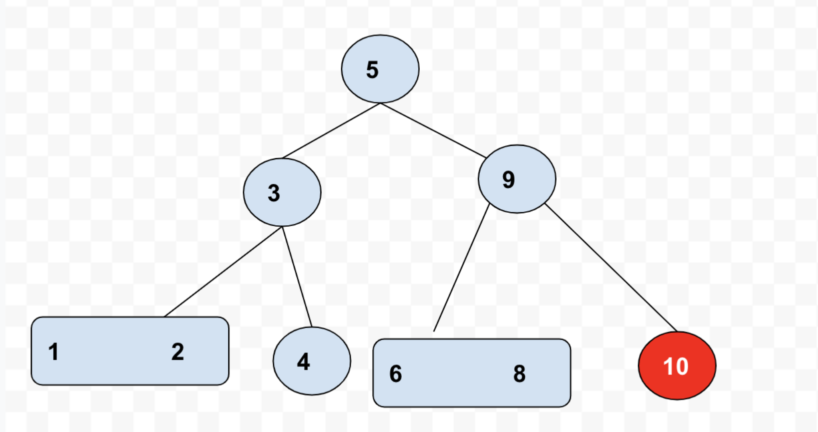
2-3 Tree Delete Example

Delete 7



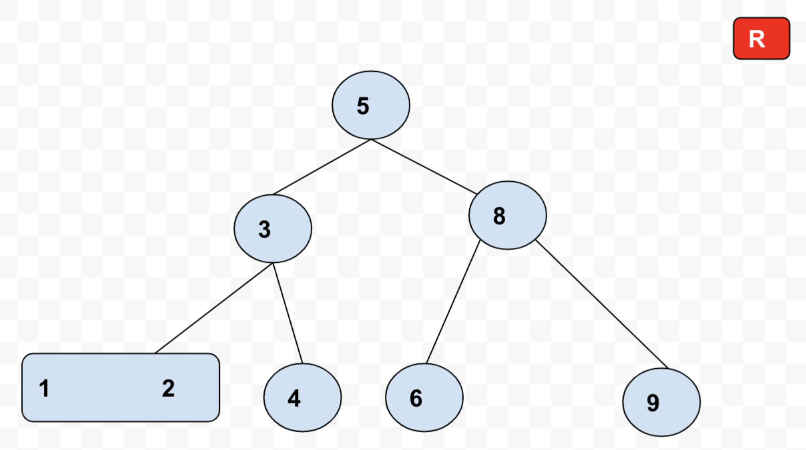
2-3 Tree Delete Example

Delete 10



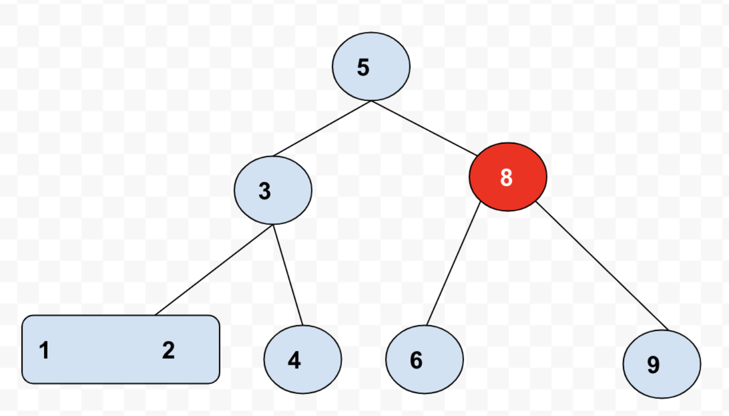
2-3 Tree Delete Example

Delete 10



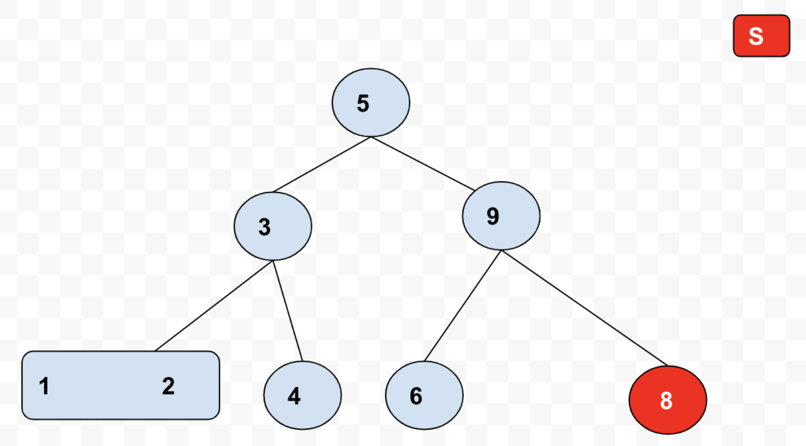
2-3 Tree Delete Example

Delete 8



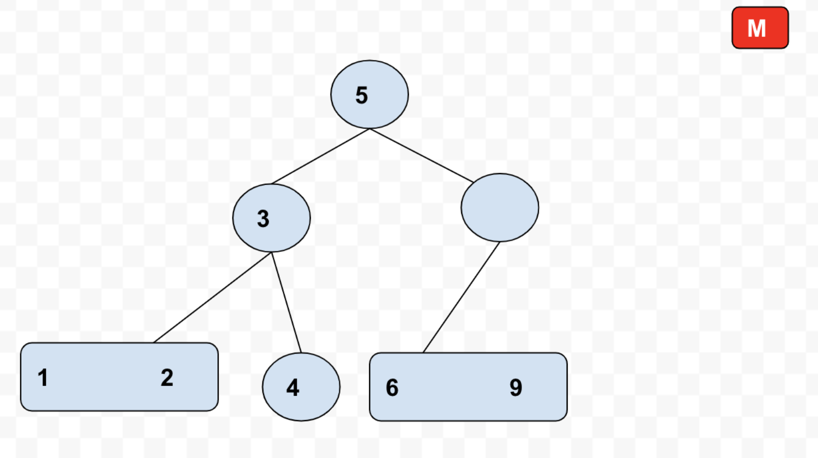
2-3 Tree Delete Example

Delete 8



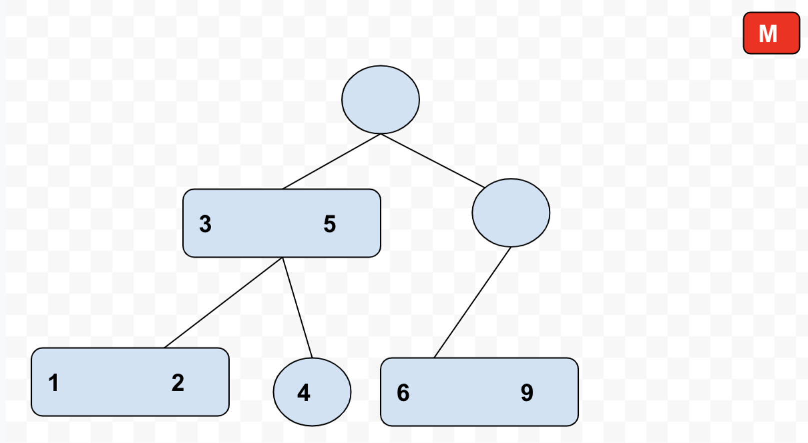
2-3 Tree Delete Example

Delete 8



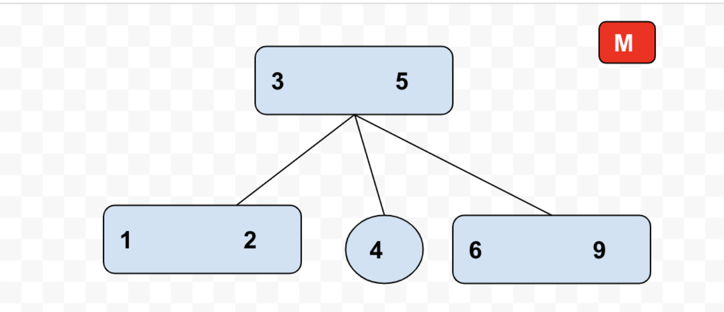
2-3 Tree Delete Example

Delete 8



2-3 Tree Delete Example

Delete 8



Time Complexity

Search, Insert, and Delete - **O(log n)** because of the balanced tree structure.

Reading Assignment

Sedgewick 3.2, 3.3 BST and 2-3 Trees

Questions?

Please ask if there are any Questions!