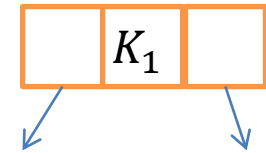


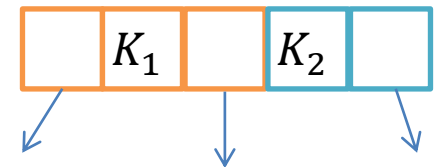
2-3 Search Tree

2-3 Search Trees

- Each node can contain 1 or 2 keys.
- Each node has 2 or 3 children, hence 2-3 trees.



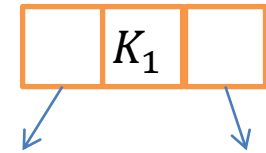
2-node



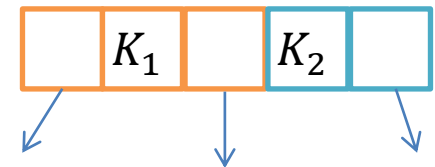
3-node

2-3 Search Trees

- Each node can contain 1 or 2 keys.
- Each node has 2 or 3 children, hence 2-3 trees.
- The keys in the nodes are ordered from **small to large**.



2-node

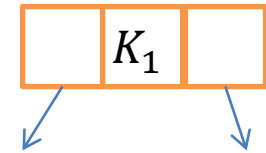


3-node

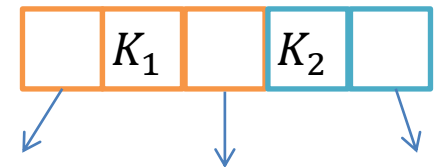
$$K_1 < K_2$$

2-3 Search Trees

- Each node can contain 1 or 2 keys.
- Each node has 2 or 3 children, hence 2-3 trees.
- The keys in the nodes are ordered from **small to large**.
- All leaves are at the same (bottom most) level, meaning **we always add at the bottom**.



2-node

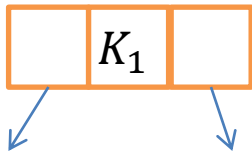


3-node

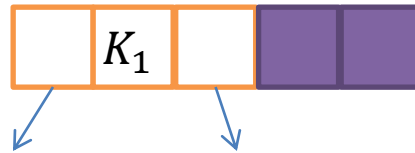
$$K_1 < K_2$$

2-3 Search Tree Node

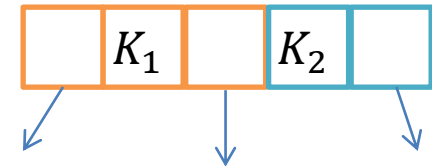
```
struct Node23{  
    Node23 *left, *middle, *right;  
    Key key1, key2;  
};
```



2-node (not showing empty)



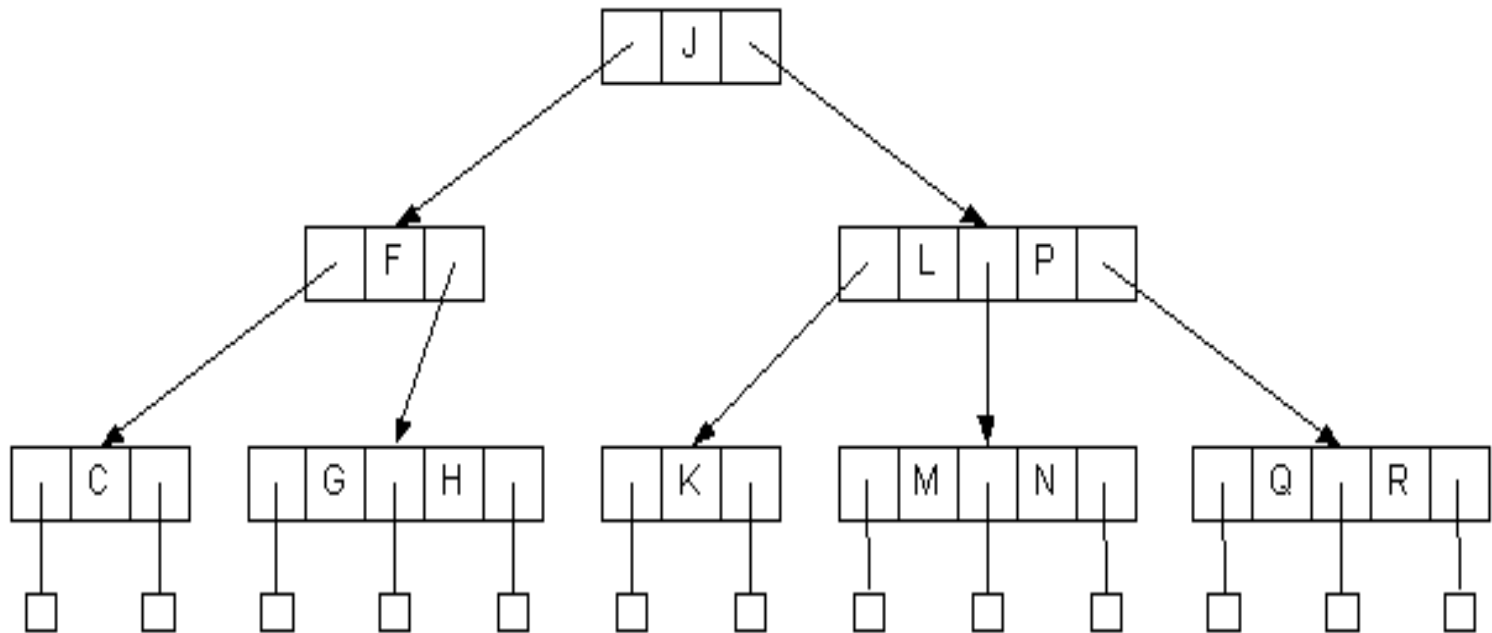
2-node (showing empty)



3-node

$$K_1 < K_2$$

Example

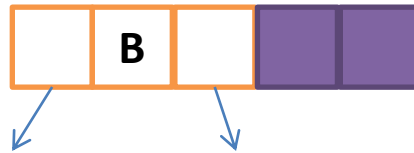


Insertion

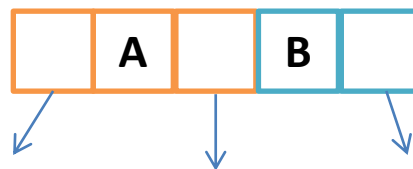
- Splitting this way is called **bottom-up** balancing
 - Insert the node at the bottom-most level at correct location.
 - If the node is a 3-node, split it and pass the middle key to the parent.
 - If the parent is also a 3-node, split the parent and pass the middle key up
 - Etc...
 - Eventually, the root will also be a 3-node and splitting it will **grow** the tree one level.

Insertion

- If the node you insert is a 2-node, simply grow the node to a 3-node

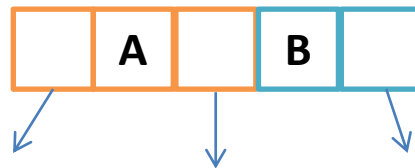


Inserting **'A'**

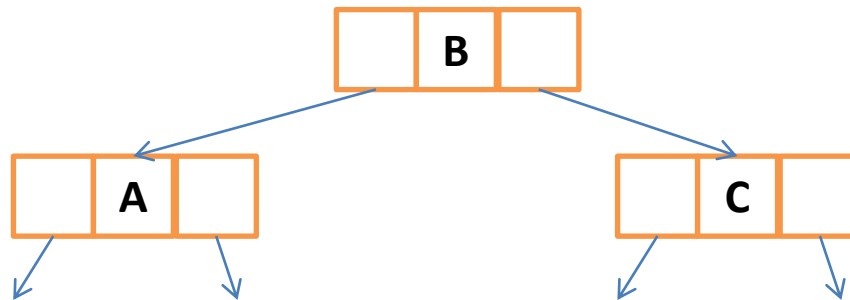


Insertion

- If the node you insert is a 3-node, we cannot grow the node more
 - We split it!

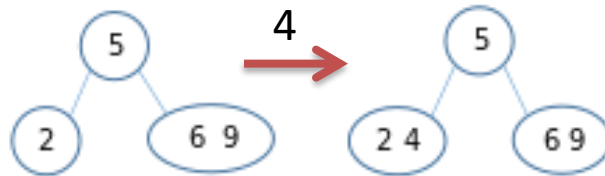


Inserting 'C'

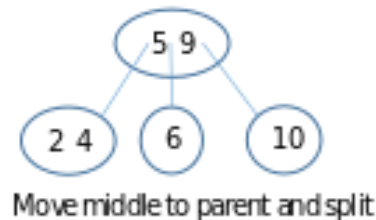
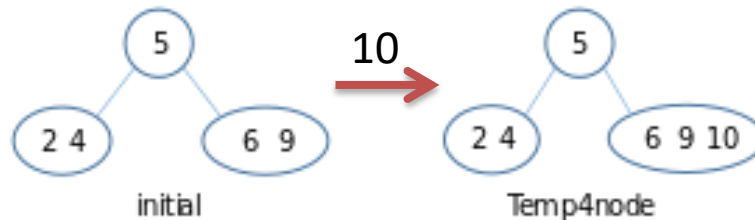


Insertion - Cases

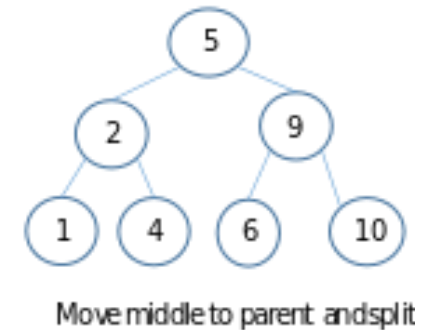
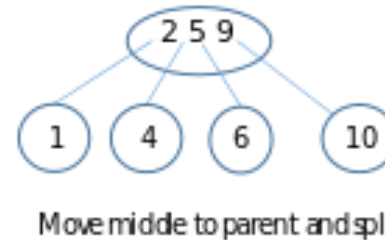
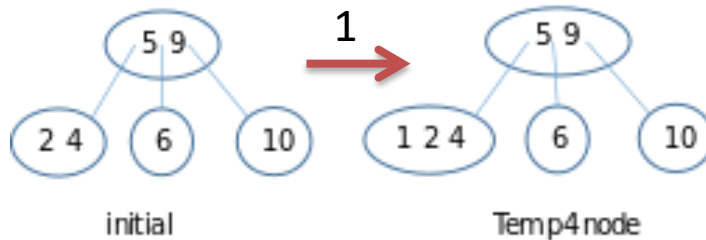
Insert in a 2-node :



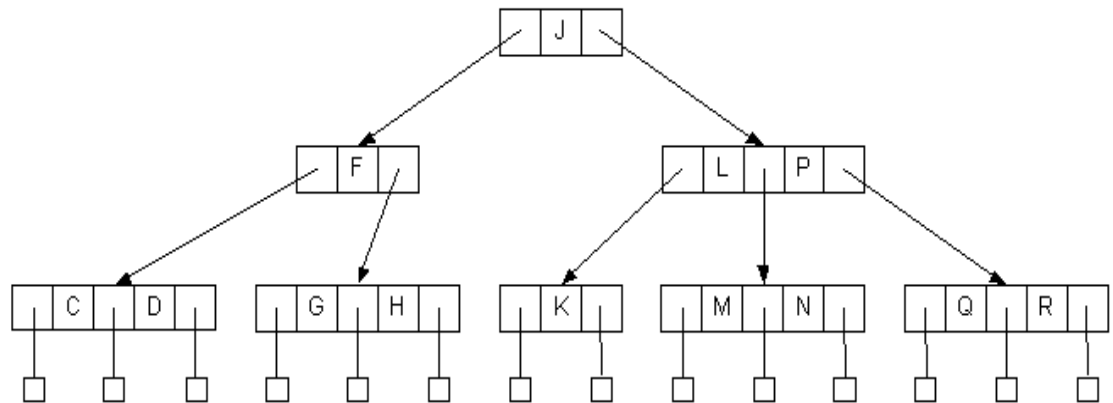
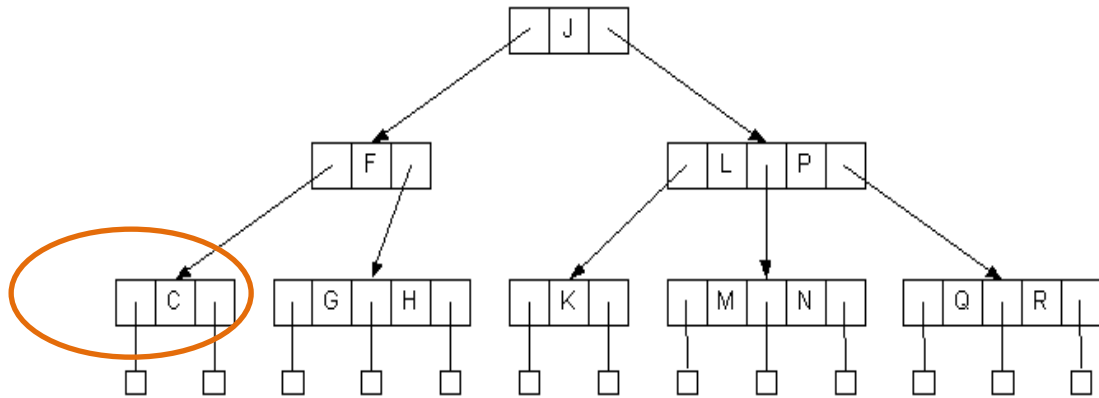
Insert in a 3-node (2 node parent) :



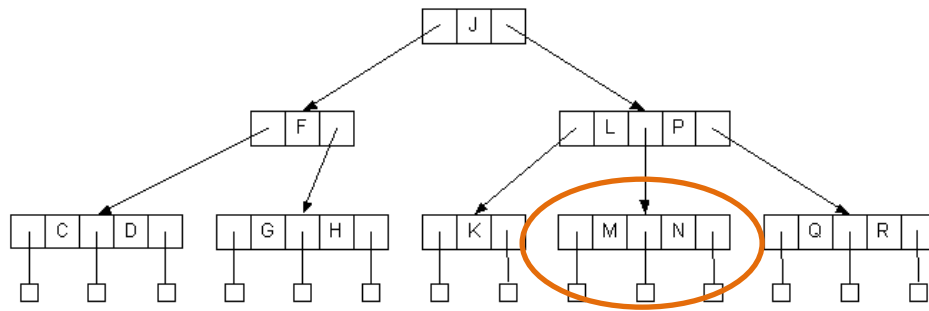
Insert in a 3-node (3 node parent) :



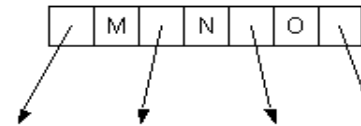
Example: Insert D



Insert O

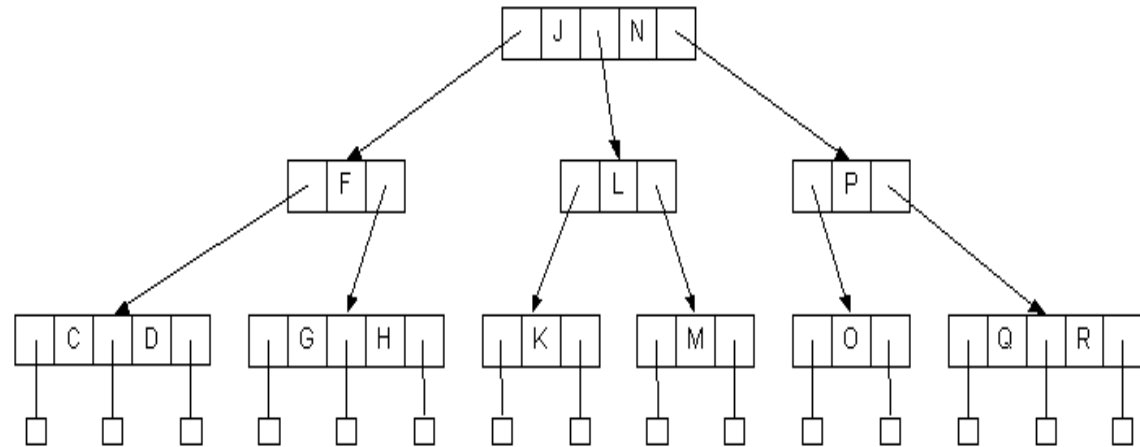
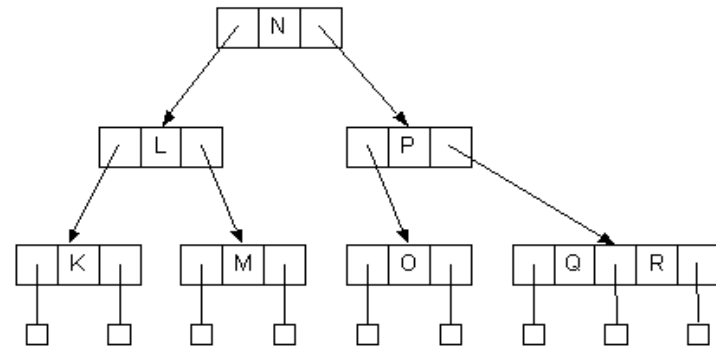
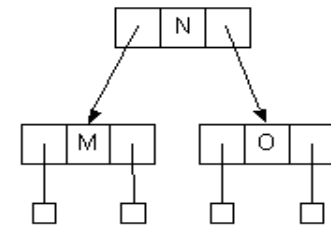


4-node, not valid in 2-3 tree

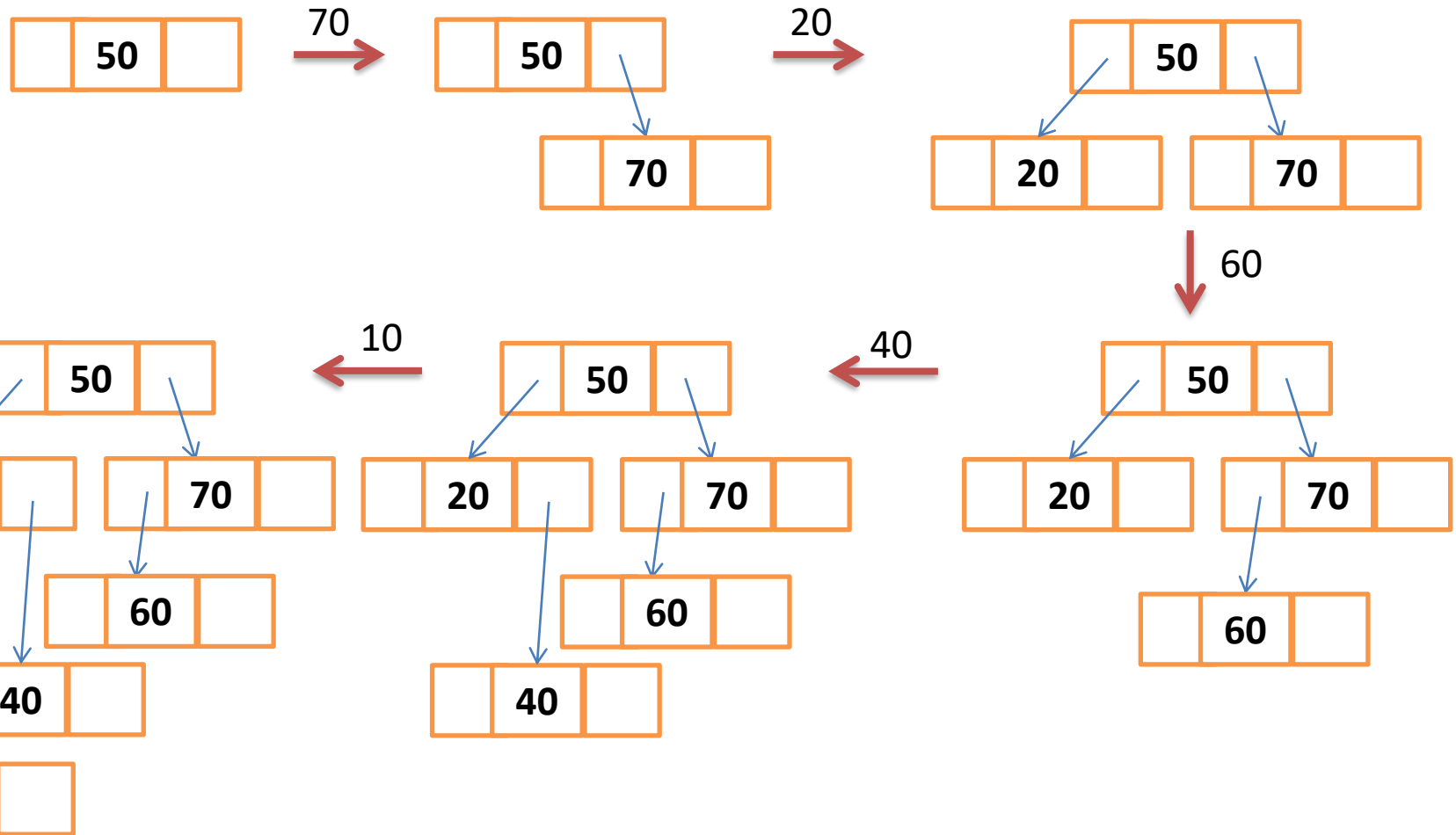


Split →

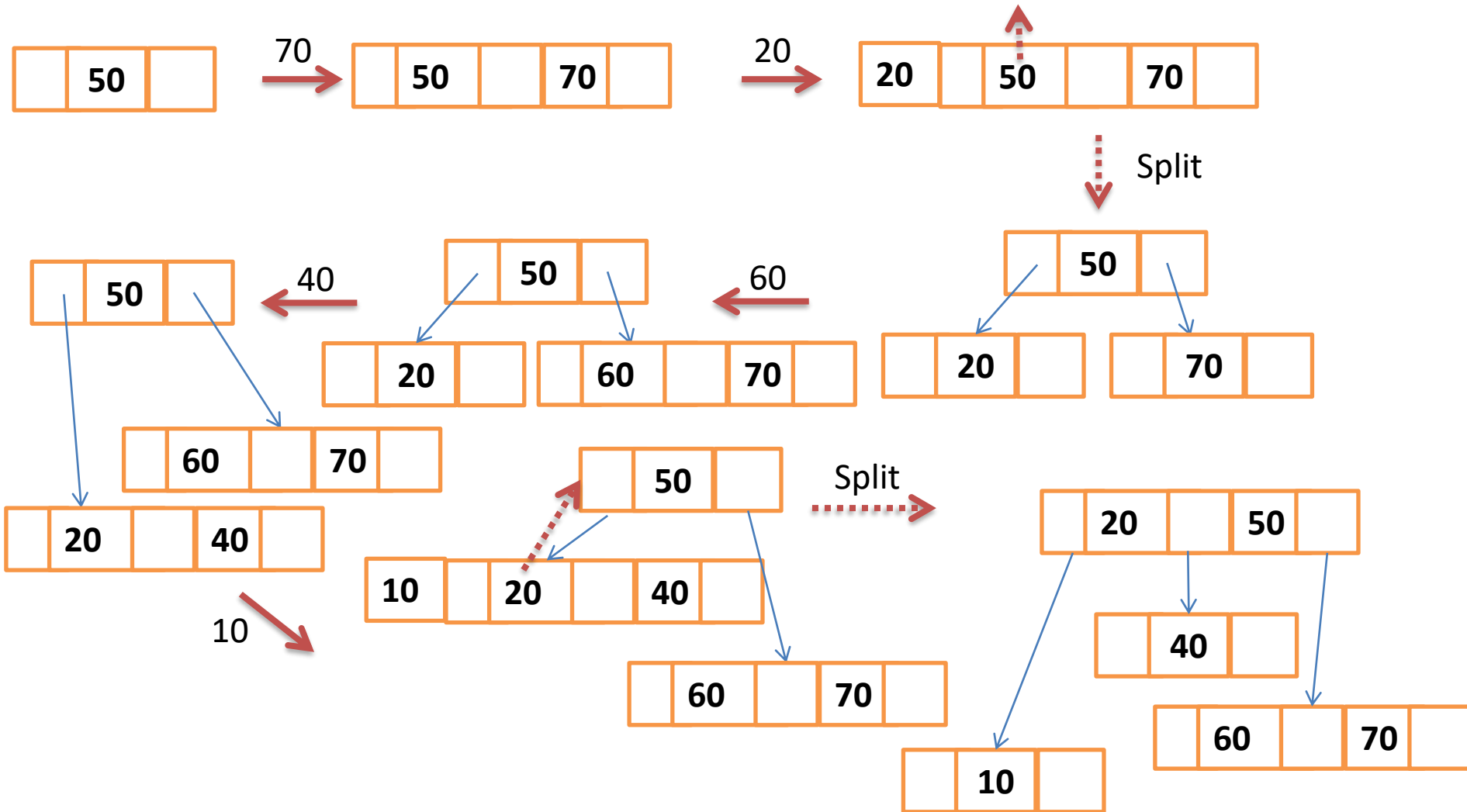
Three 2-nodes



BST with 50, 70, 20, 60, 40, 10



2-3 Tree with 50, 70, 20, 60, 40, 10

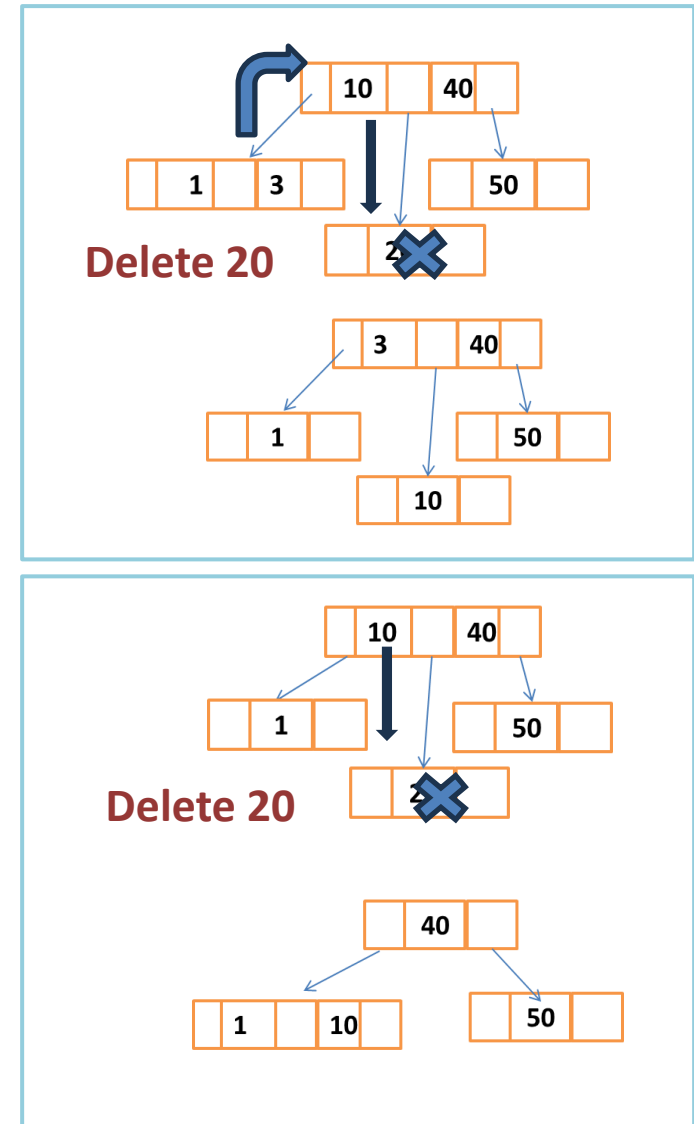


Properties of 2-3 Search Trees

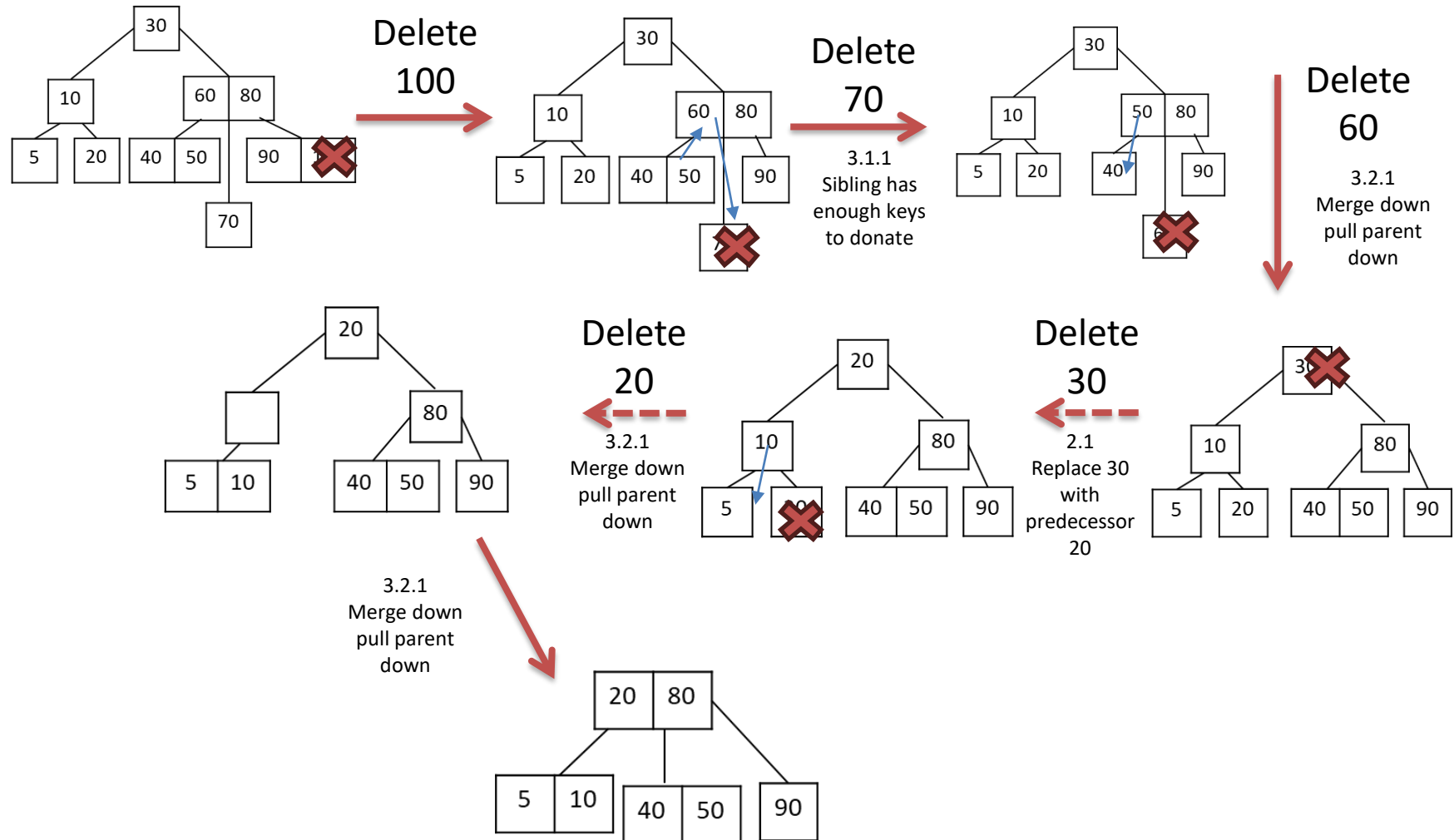
- 2-3 search trees **guarantee to be balanced** at all times.
- Searches are $O(\log N)$ in worst case.
- Balance is maintained during insertion
 - Splitting nodes, worst case $O(\log N)$, average case $O(1)$

Deletion

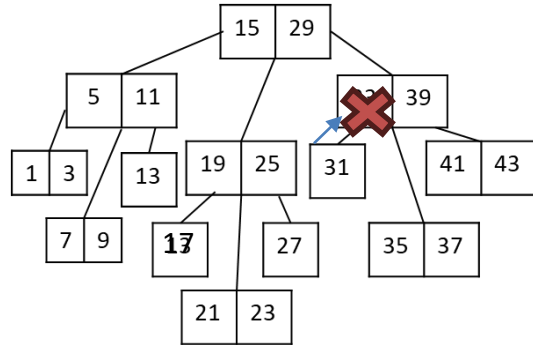
1. Find the key (X) that needs to be deleted
 2. If X is in a non-leaf node
 - 2.1 Replace X with its predecessor
 - 2.2 If no predecessor replace with successor
 3. If X is in leaf node
 - 3.1 if left/right sibling has enough keys
(ie more than 1)
 - 3.1.1 donate by rotating with parent (may need to take over sibling's child)
 - 3.2 else
 - 3.2.1 Merge down (pull the parent down)
- If parent is underfull repeat Step 3 recursively



Example 1



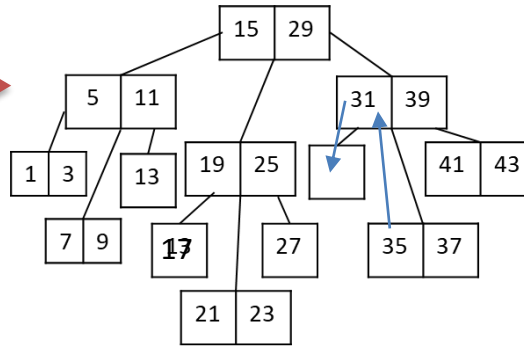
Example 2



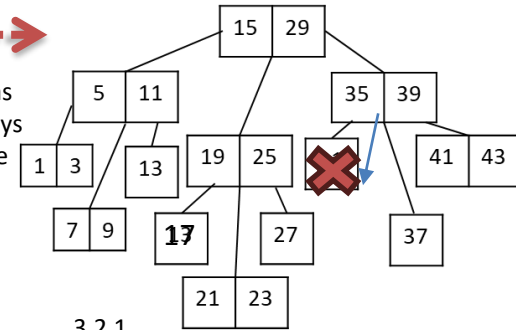
Delete

33

2.1
Replace 33
with
predecessor
31



3.1.1
Sibling has
enough keys
to donate

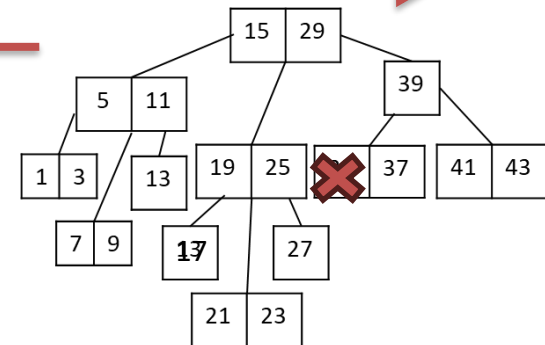
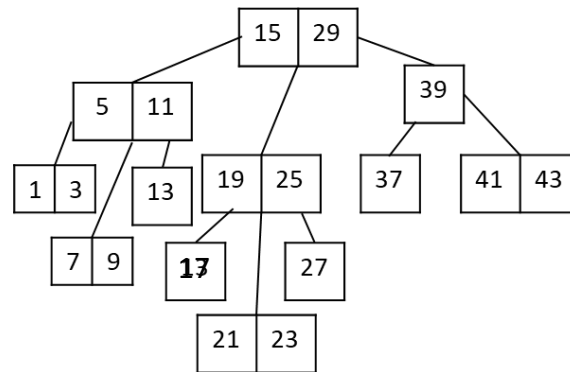


3.2.1
Merge down
pull parent
down

Delete
31

Delete

35



Summary

- 2-3 Trees are always balanced.
- Nodes are **ALWAYS** inserted at the bottom-most level.
- Balance is maintained by splitting full nodes and passing up the middle node.
- This makes the tree's height increase by one, only when the root is split.

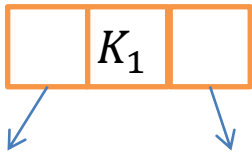
2-3-4 Search Tree

Basic Properties

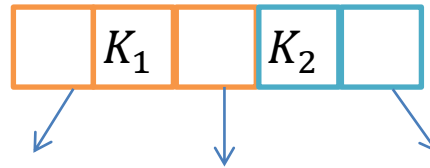
- Similar to 2-3 trees
- Nodes can contain 1, 2, or 3 keys.
- Nodes can have 2, 3, 4 children, hence **2-3-4 tree**.
 - Each can have at most 4 children.
- Similarly to 2-3 trees, 2-3-4 trees are guaranteed to be always balanced.
- Balancing algorithm also relies on Splitting nodes
- Number of splits in the worst-case is **$O(\log N)$**
 - When is the worst-case?
- Average number of splits is very few.

2-3-4 Node

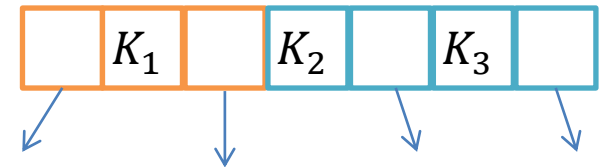
```
struct Node234
{
    Node23 *left, *midleft, *midright, *right;
    Key key1, key2, key3;
};
```



2-node



3-node



4-node
 $K_1 < K_2 < K_3$

Balancing Algorithm

- Balancing also occurs on insertion.
- Modifying the algorithms for balancing can produce **better efficiency**.
- Previously, with 2-3 Trees, we have seen **bottom-up** balancing. $O(\log N)$ splits in worst case where splitting propagates up to the root
- We will see **top-down** balancing
 - As you go down the tree to insert a node, split any full node.
 - **A full node is a 4-node (3 keys/4 child node).**

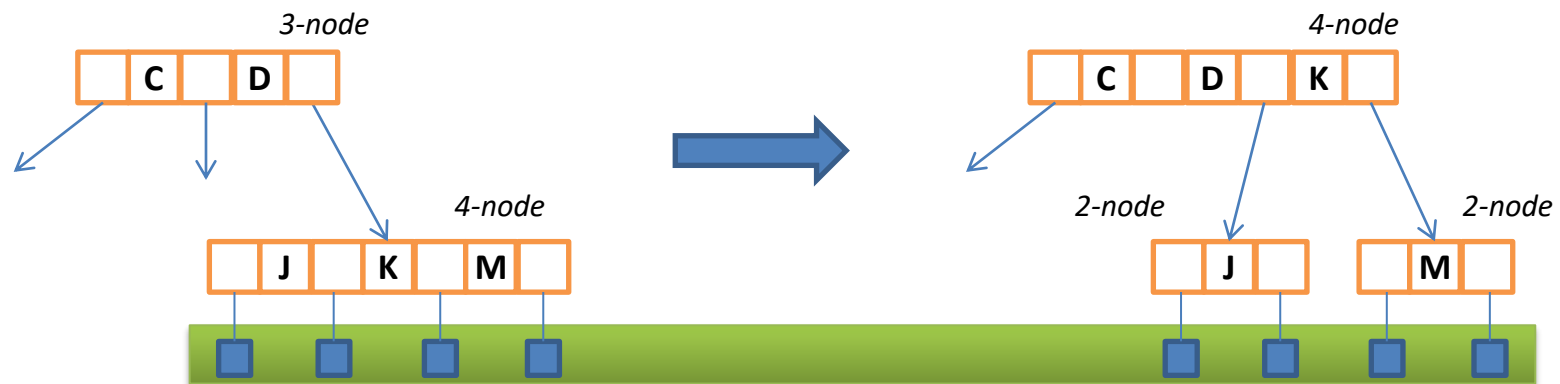
How to split a 2-3-4 node?

Insertion

- Insert at leaf position
- While going down the tree, if encounter a 4 – node
 - Split it, push the middle element to the parent
 - If it's the root, split and create a new root

Advantage of Splitting 2-3-4 Trees

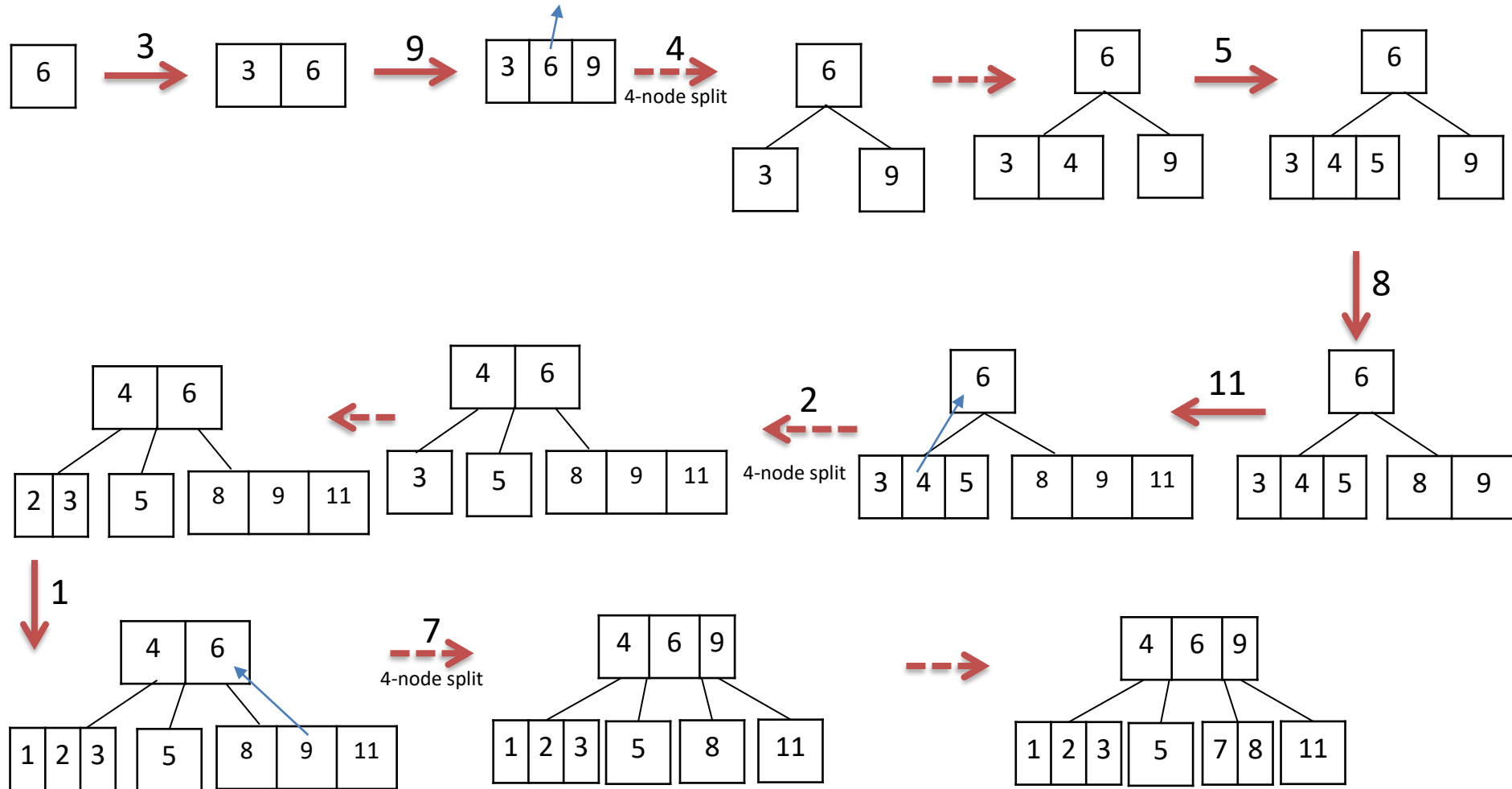
- Splitting a node is **cleaner**.
- Splitting a 4-node into two 2-nodes preserves the number of child links.
- Changes do not have to be propagated. Change **remains local to split**.



Top-Down Balancing

- Split nodes **on the way down**.
 - Guarantees that each node we pass through is not a 4-node.
 - When we reach the bottom, we will not be on a 4-node (think about it)
- This way, we only traverse the tree once, when inserting/balancing.
- After each insertion, check if the root is a 4-node
 - If it is, split it directly. This will avoid to do it at next insertion.
 - Splitting the root is the only way to grow the tree.

2-3-4 tree with 6,3,9,4,5,8,11,2,1,7



Deletion

Find the key (X) that needs to be deleted

1. if (X is in a leaf node with atleast 2 keys)

1.1 remove (X)

2. else if (X is in a non-leaf node)

2.1 if (left_child of the node ≥ 2 keys)

2.1.1 replace (X,predecessor)

2.1.2 remove (predecessor)

2.2 else if (right_child of the node ≥ 2 keys)

2.2.1 replace (X,successor)

2.2.2 remove (successor)

2.3 else // both children have only 1 key each

2.2.1 merge(left_child,X,right_child)

2.2.2 remove (X)

3. else // ch is the node_visited_so_far

3.1 if (ch has only 1 child)

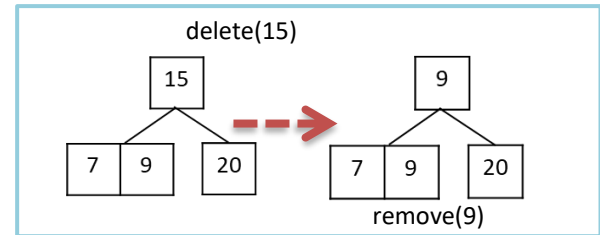
3.1.1 if (ch's sibling ≥ 2 keys)

rotate a key into ch

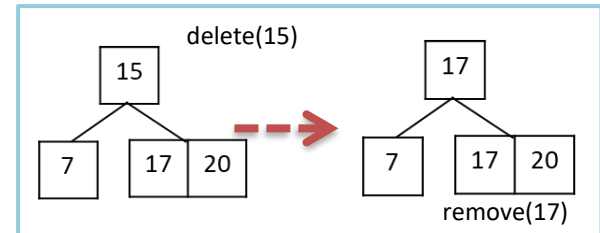
3.1.2 else if (ch's sibling has 1 key)

merge(ch,parent,sibling)

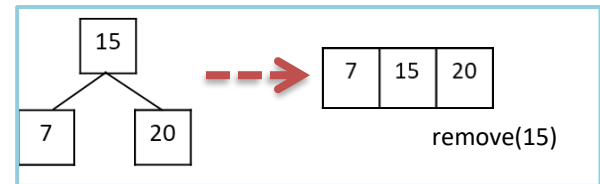
2.1



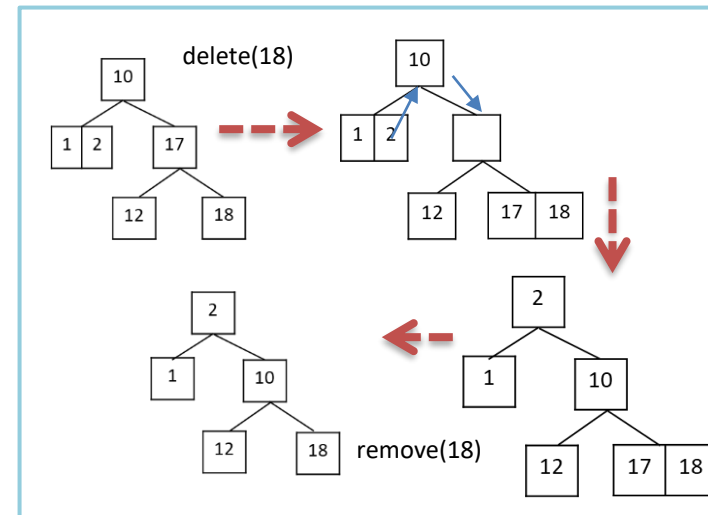
2.2



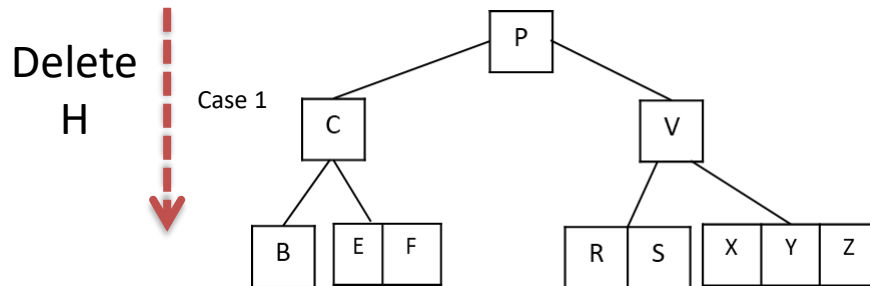
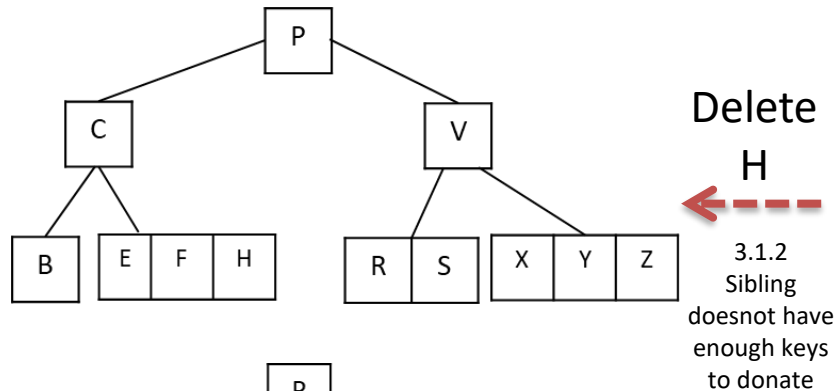
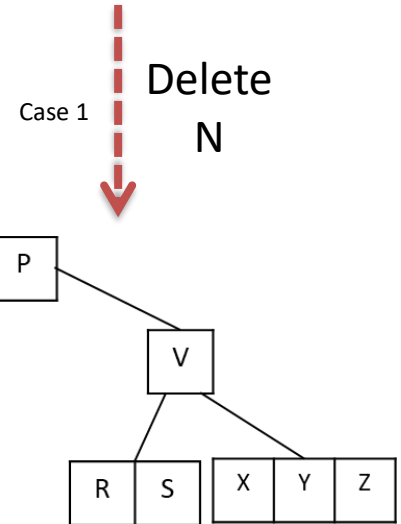
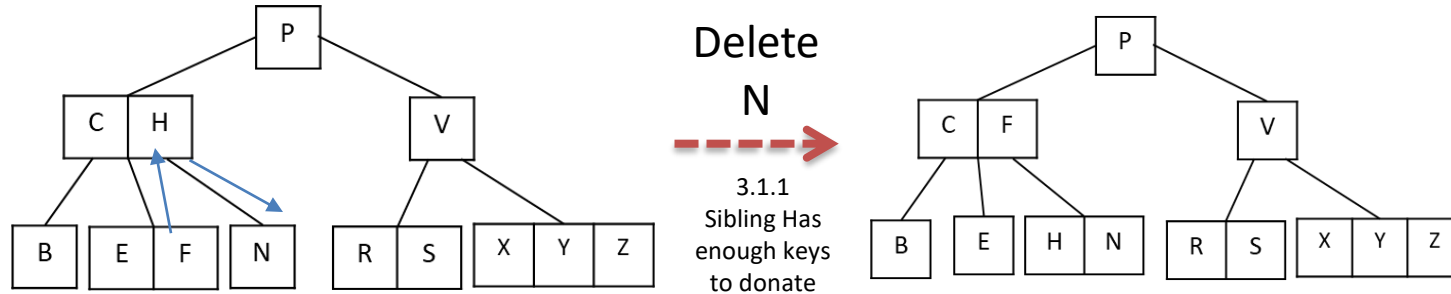
2.3



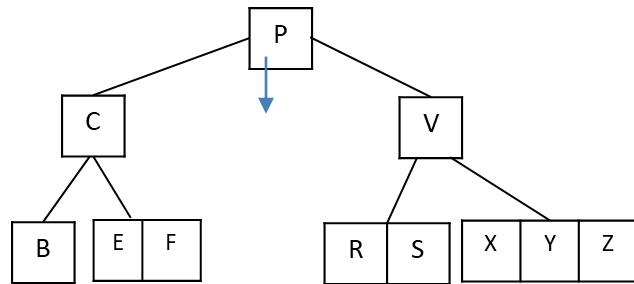
3



Delete N,H,R,C,E,P,V



Delete R,C,E,P,V

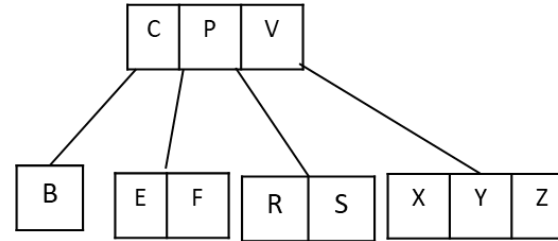


Delete

R



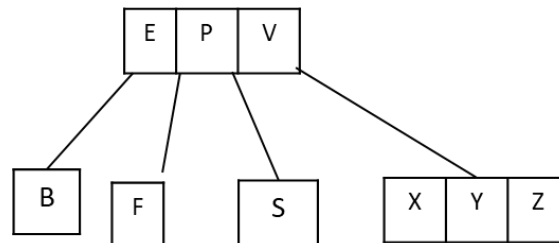
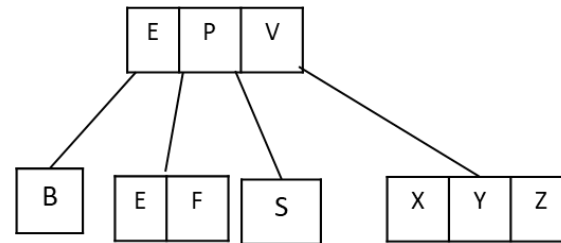
3.1.2
Node V's
siblings
does not have
enough keys
to donate



Case 1

Delete
RDelete
E

Case 1

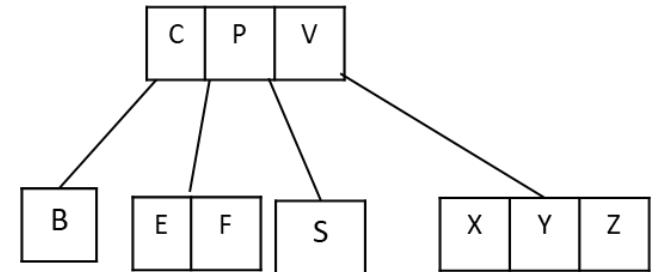


Delete

C



2.2
Replace with
successor
Remove succ



Delete P,V

