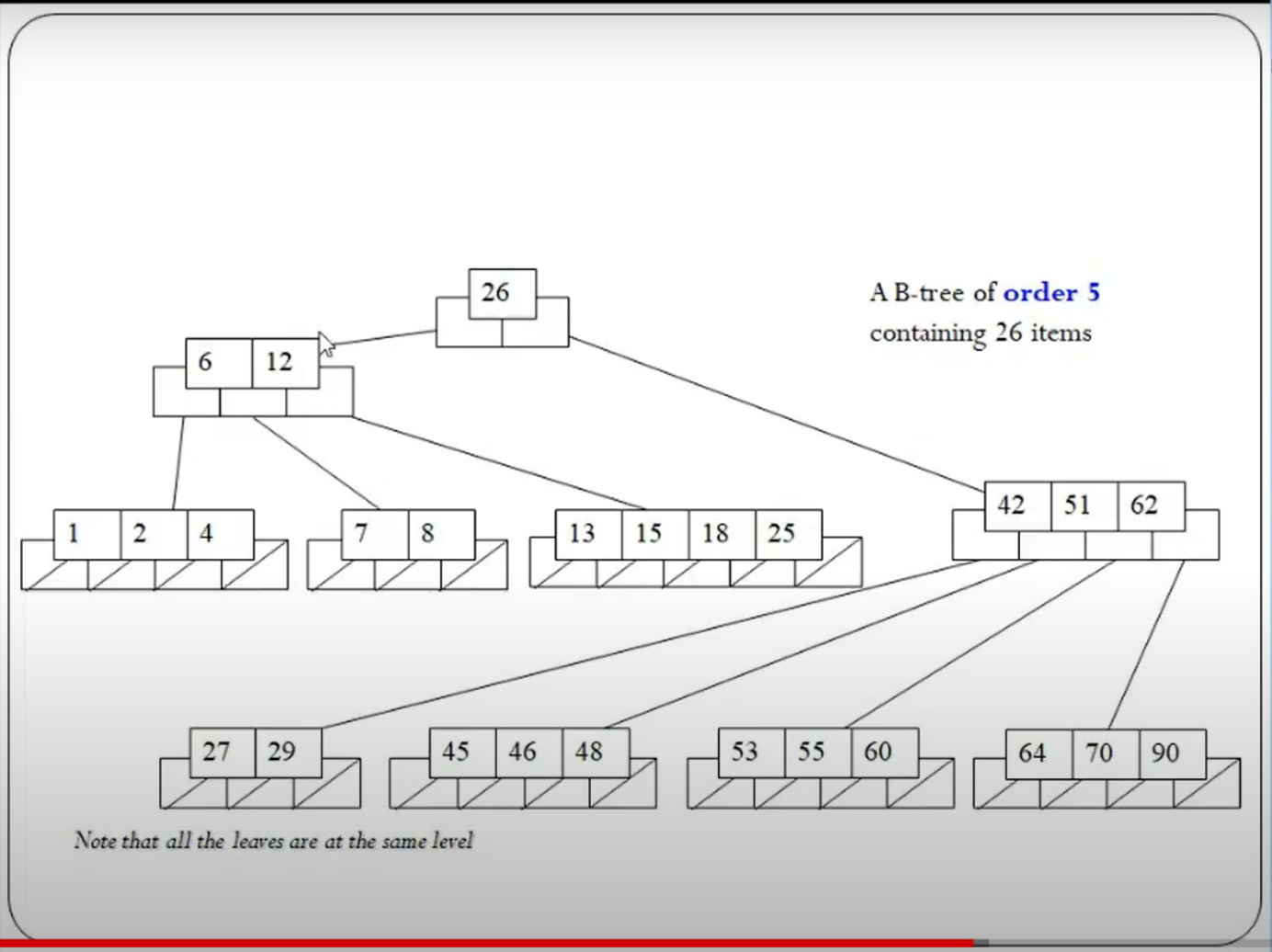
# B-tree

- B-tree of order m is an *m*-way search tree, where
  - Maximum number of children of a non-leaf node(except root) : m
  - Minimum number of children of a non-leaf node(except root): $\lceil m / 2 \rceil$
  - The root has minimum 2 and maximum *m* children
  - Number of keys in each non-leaf node is one less than the number of its children
  - Keys partition the keys in the children in the fashion of a search tree
  - All leaves are on the same level
  - A leaf node contains minimum $\lceil m / 2 \rceil$-1 and maximum $m - 1$ keys



Microsoft PowerPoint - [PowerPoint Slide Show - [B tree]]

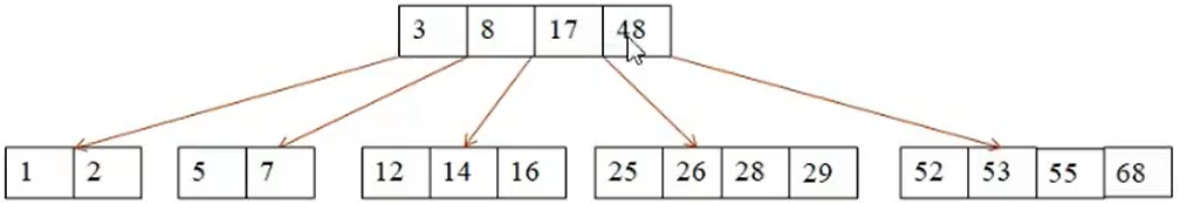
# Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- Remember, Key inside a node are always sorted to maintain search


Alka Jindal

# Constructing a B-tree (cont'd)

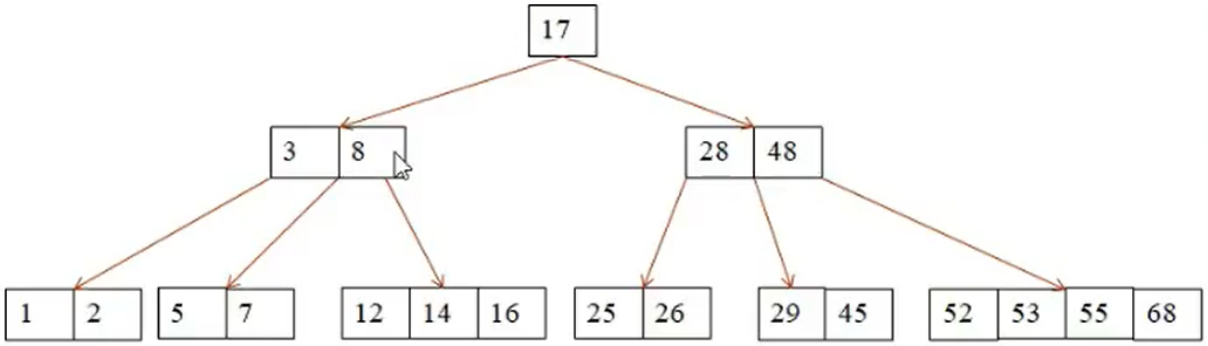
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



# Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- As 45 can not be accommodated in leaf node, we will split the node and promote middle key 28 to parent
- Further parent node also needs to be split
- As the parent is also root node, will increase the height of B-tree



# Insertion in B-tree: Algo

1. Attempt to insert the new key into a leaf
2. If this could result in too many keys in leaf node, split the leaf node and promote the key upward to parent node
3. Repeat step 2 all the way to the top
4. If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Microsoft PowerPoint - [PowerPoint Slide Show - [B tree]]

# Insertion in B-tree : Analysis

- Finding the correct location:
  - In worst case, we might require to compare input "key" with all key present in one node i.e. *m-1* comparison. But *m* is a constant, thus time required is O(1)
  - Total number of levels to be traversed is O(h) or O(log n)

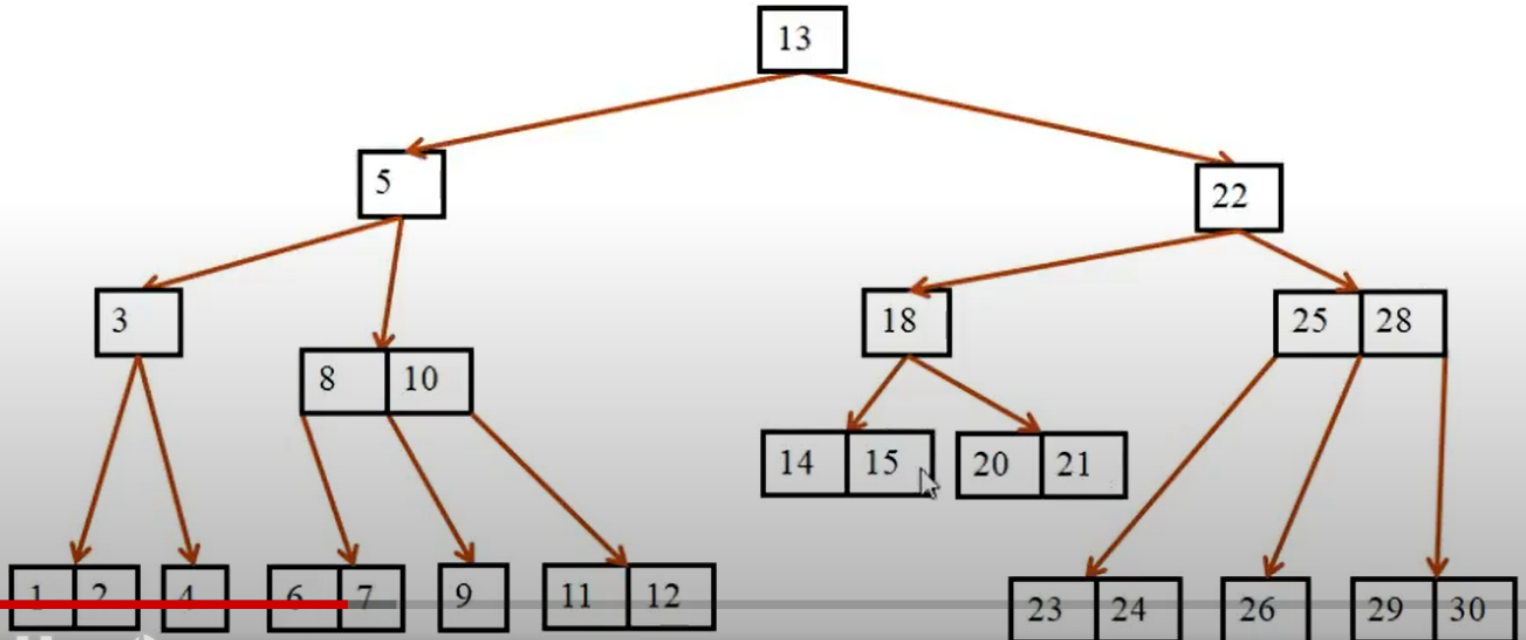

Alka Jindal

# Insertion in B-tree : Analysis

- Finding the correct location:
  - In worst case, we might require to compare input "key" with all key present in one node i.e. *m-1* comparison. But *m* is a constant, thus time required is O(1)
  - Total number of levels to be traversed is O(h) or O(log n)
- Checking the tree after insertion is B-tree
  - Split takes constant time as few pointers only needs to be updated. Hence, O(1)
  - Might have to do up to root. Thus, O(log n)
- Total time complexity for insertion: O(log n)

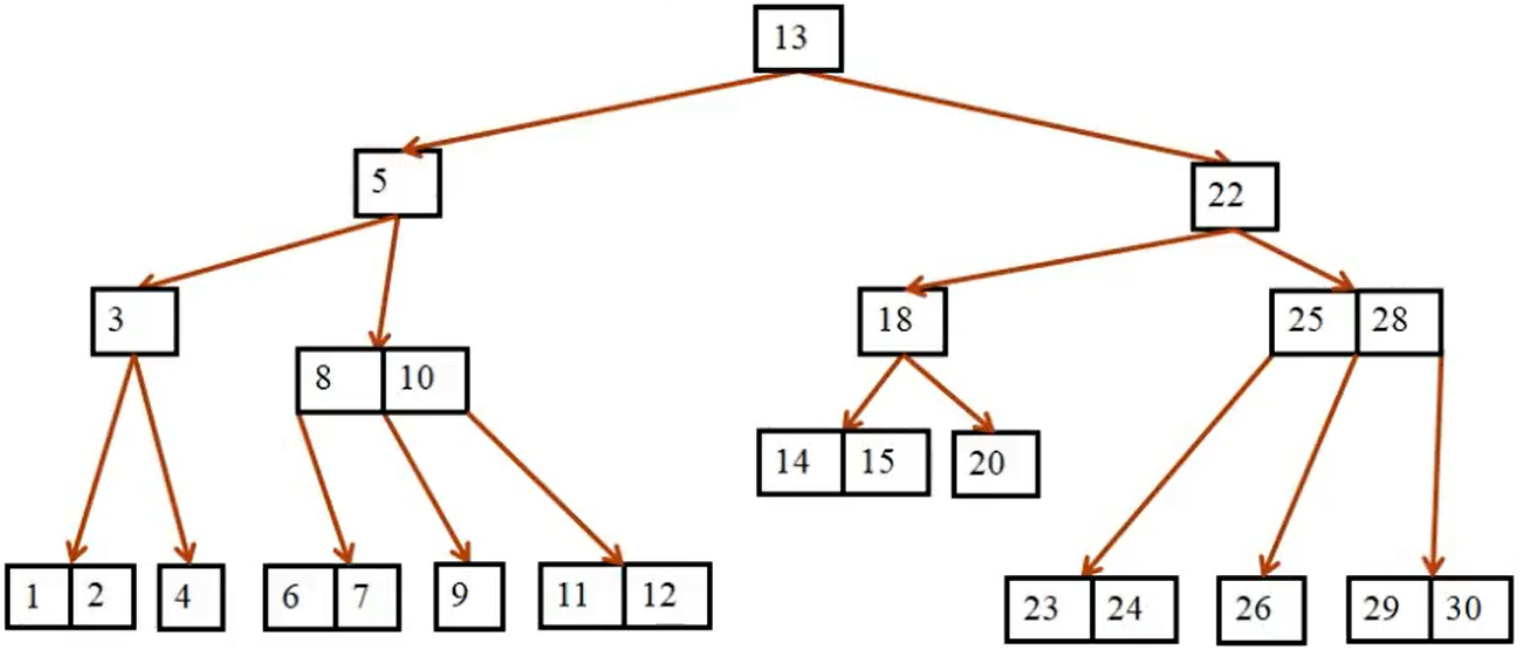
# Deletion in B-tree

- Order of B-tree is 4. Max and min children are 4 and 2.
- Delete key in given order: 21, 25, 20, 23, 18



# Deletion in B-tree(Cont'd)

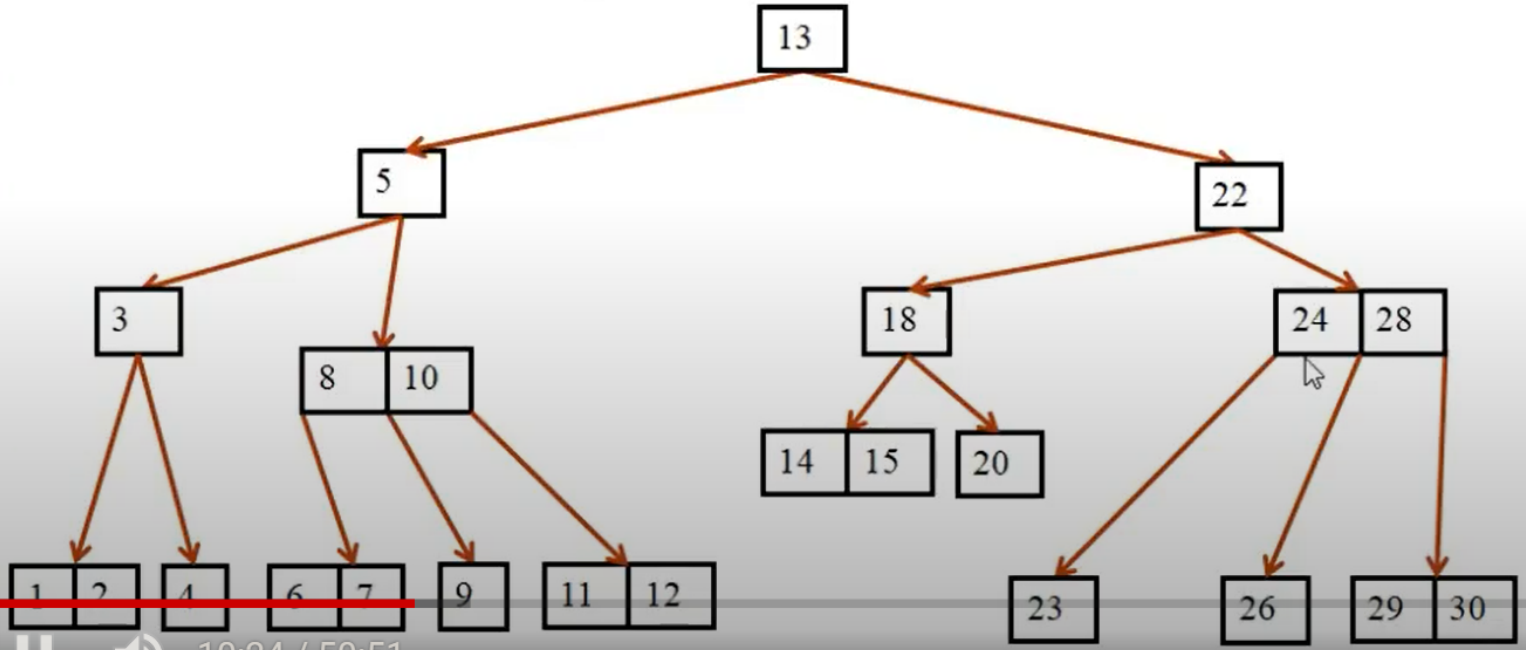
- Delete key in given order: 21, 25, 20, 23, 18
- When deletion is done from leaf node, if node satisfies minimum key requirement, do nothing



Microsoft PowerPoint - [PowerPoint Slide Show - [B tree]]

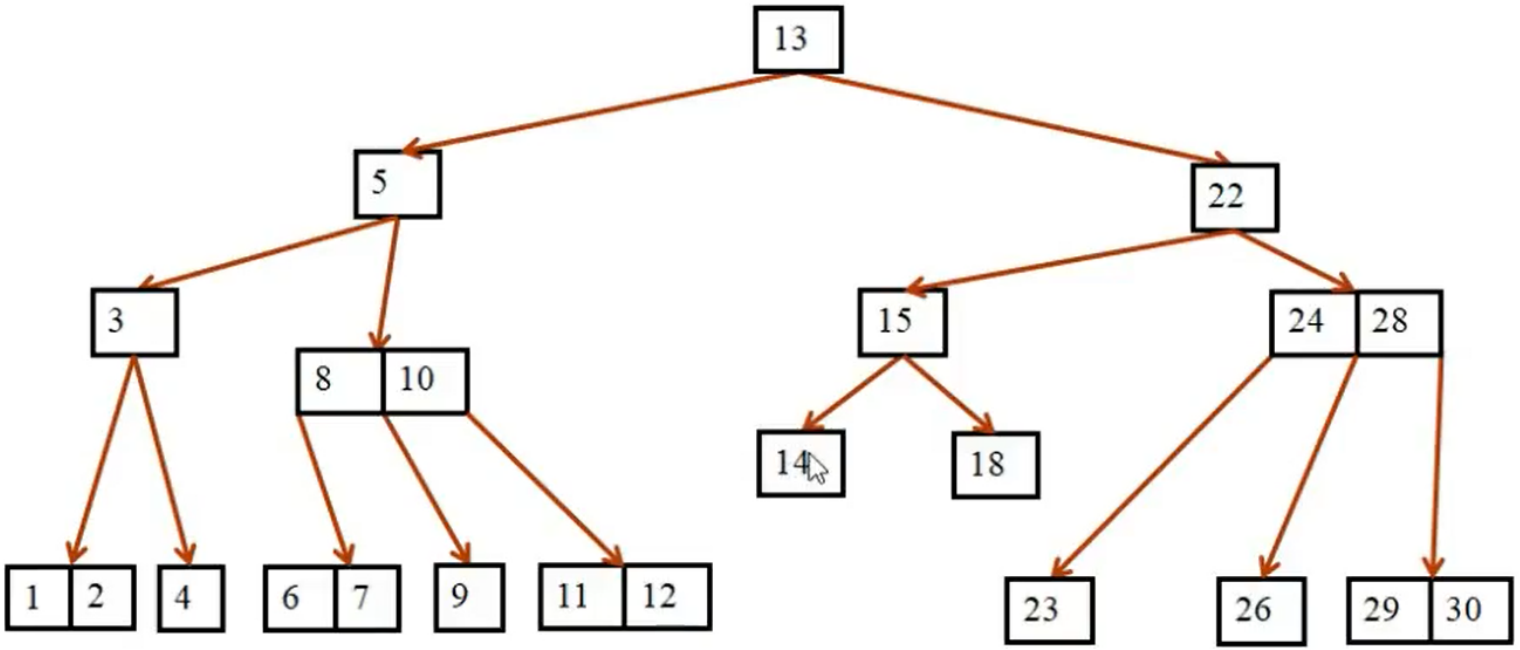
# Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- When deletion is done from internal node, swap it with predecessor and delete predecessor



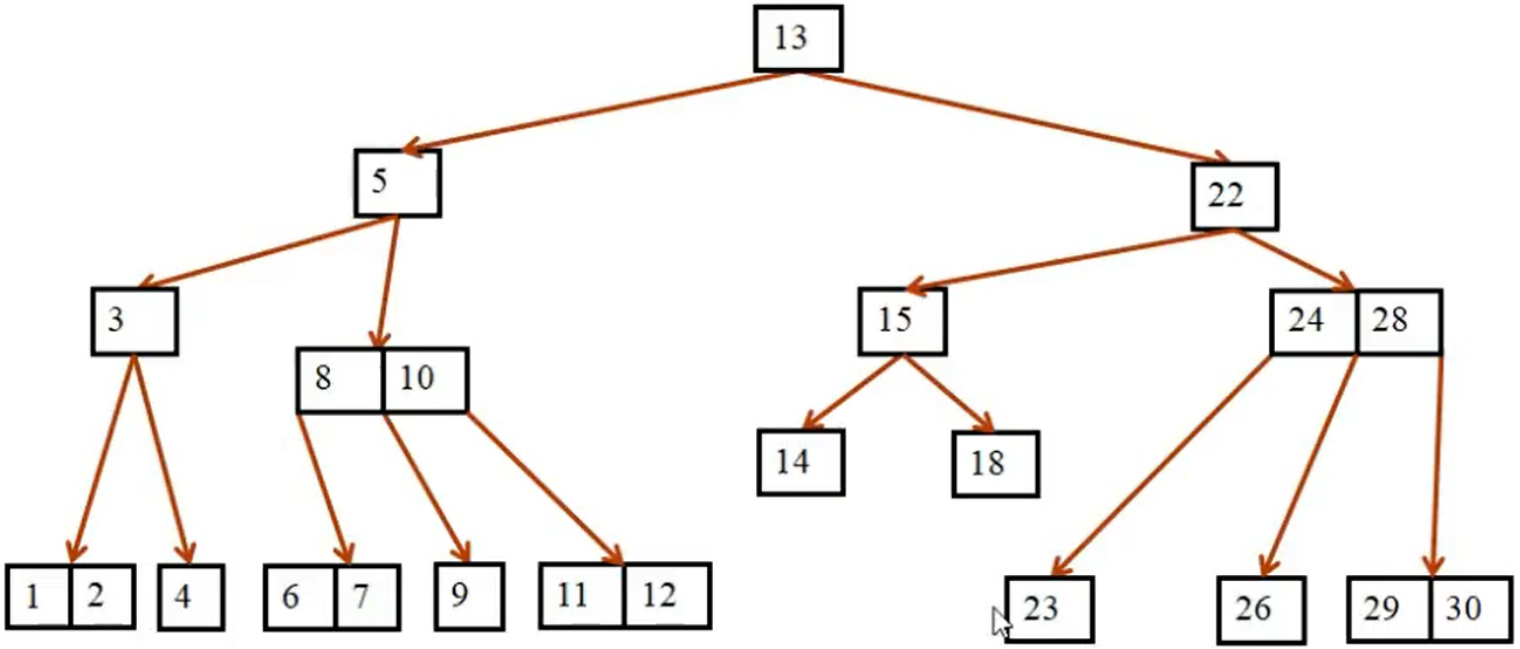
# Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- If deleting a key violates minimum key requirement, borrow key from adjacent sibling(s)



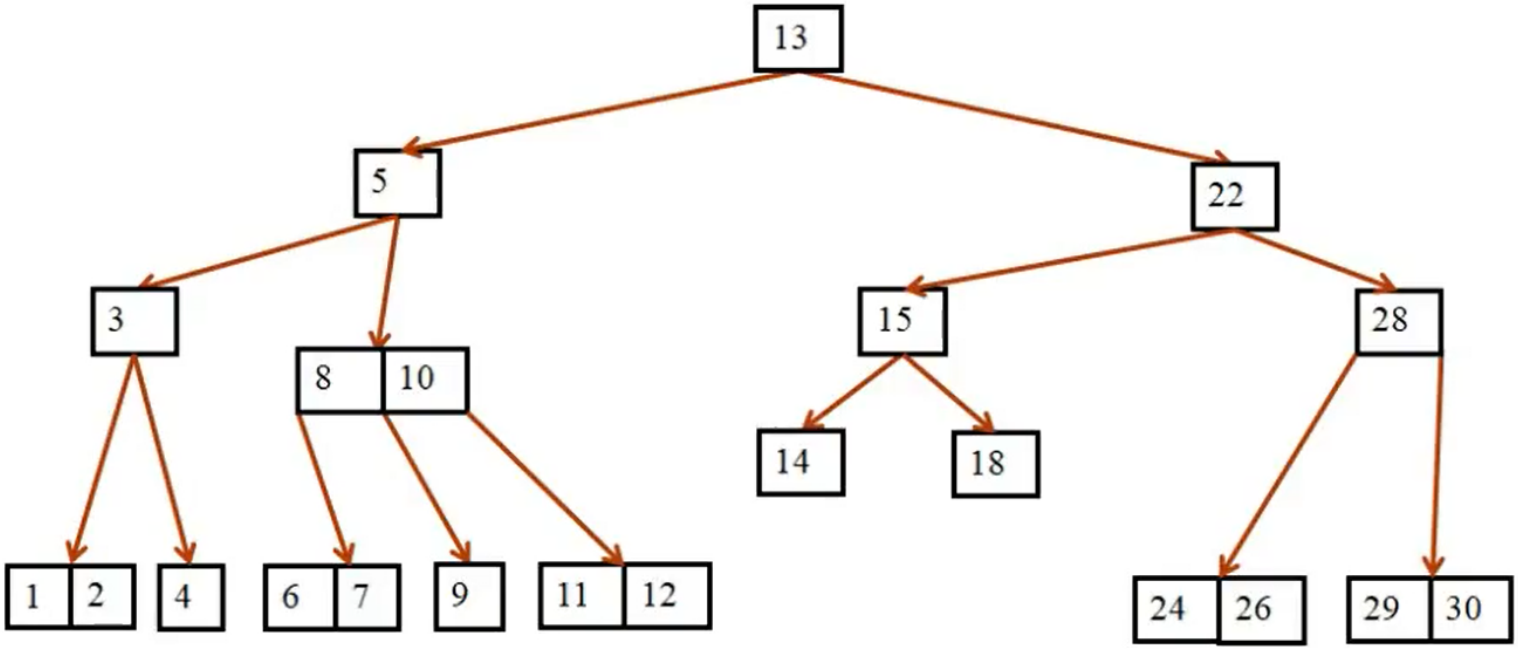
# Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- If deleting a key violates minimum key requirement, borrow key from adjacent sibling(s)



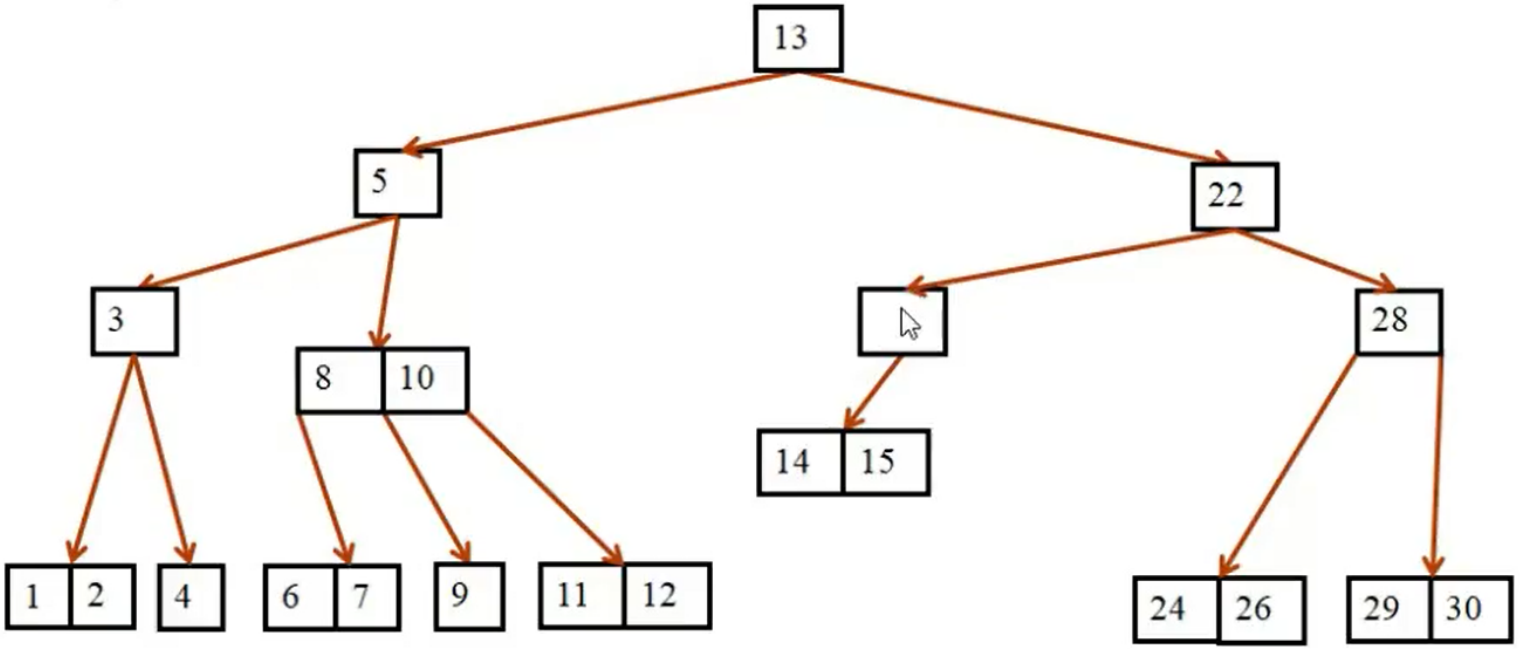
# Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- If adjacent sibling(s) can not lend key, merge siblings and bring down the key in parent node separating them

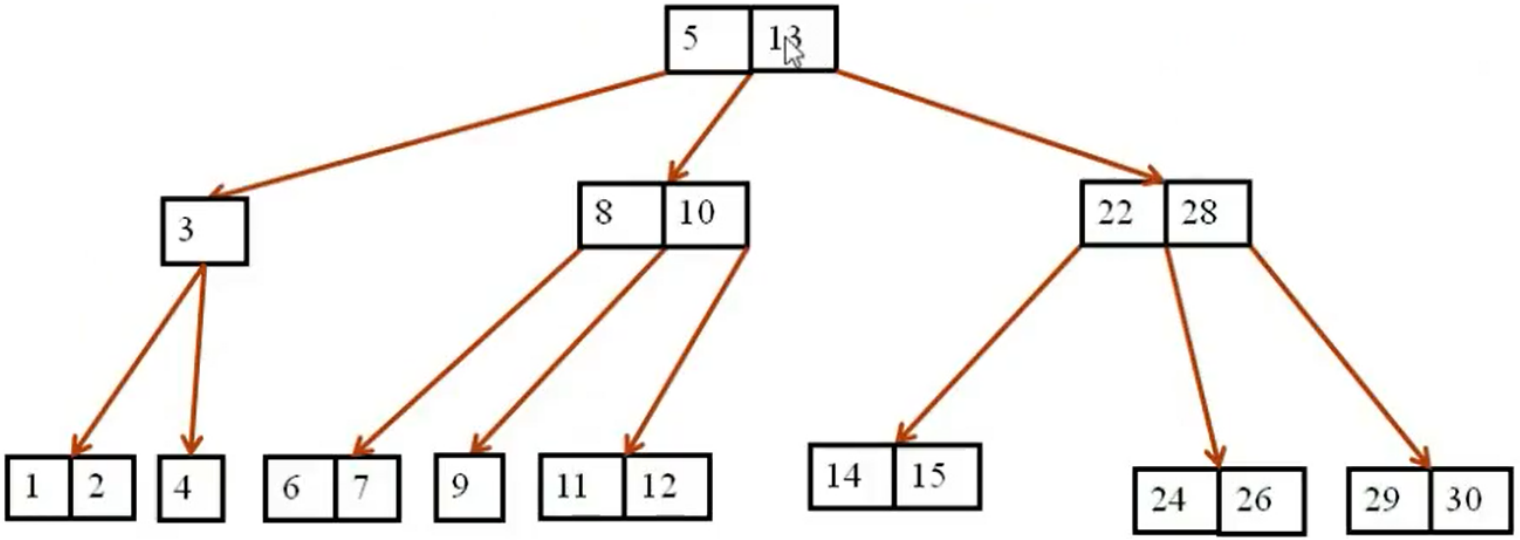


# Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- Moving down key from parent node might result in parent node violating minimum key requirement. Perform deletion algo at parent.



# Deletion in B-tree(Cont'd)



# Deletion in B-tree : Algo

1. **If deletion is from leaf node, delete key and check if minimum key requirement is satisfied or not. If not then,**
   1. Try to borrow key from left sibling if left sibling has extra key
   2. Try to borrow key from right sibling if right sibling has extra key
   3. If not possible, merge left (right if left is null) sibling and key in parent node separating them
2. **If deletion is to be done from internal node, replace with predecessor and delete predecessor key if possible. Else replace with successor and delete successor key. Then repeat step 1 for deletion from leaf**
3. **Keep repeating steps for ancestors of node where deletion happened.**

# Deletion in B-tree : Analysis

- Deleting the key from B-tree : $O(\log n)$
- Borrowing key from adjacent sibling : $O(1)$
- Merging of siblings and bringing it down : $O(1)$
- In worst case, borrowing and merging might be done for all the nodes up to root. Therefore, might take $O(\log n)$
- Thus, time complexity of deletion : $O(\log n)$

# B+ tree : Introduction

- A key is present in internal node as well as at leaf level
- When a node is split, middle key go to parent as well as a copy of that is also kept at right child (or left but follow one convention through out)
- Therefore, internal nodes only has key and pointers to children. Data is stored in leaves only
- B+ tree also allows sequential access at leaf level i.e. we can access leaves left to right as well
- Advantage:
  - Internal nodes can have more keys thus reducing the height
  - Sequential access at leaves help accessing data in ascending order

# Priority Queue

- List of elements where each element also has priority associated
- Priority values not necessarily be unique
- Priority value are total ordered
- Total Order Relation, denoted by ≤
  - Reflexive: K ≤ K
  - Anti-symmetric: if K1 ≤ K2 and K2 ≤ K1 then, K1 = K2
  - Transitive: if K1 ≤ K2 and K2 ≤ K3 then K1 ≤ K3

- Example: Queue of passengers at check-in counters. Allow passengers with shorter time to fly to check-in first.
- Application: job scheduling in OS etc.

# Priority Queue : Operations

- Priority Queue should support operations **insert()**, **minimum()** and **delete-min()**.
- For our purpose, an element with lower priority value has higher priority. For instance if A and B have priority value 5 and 10 respectively then A will be deleted first from the priority queue.