AVL Tree Deletion, Other Balanced Trees

EECS 233

AVL Trees -- Deletion

- Deletion is more complicated.
  - It requires both single and double rotations
  - We may need more than one rebalance operation (rotation) on the path from the deleted node back to the root.
- Steps:
  - First delete the node the same as deleting it from a binary search tree
    - Remember that a node can be either ***a leaf node*** or ***a node with a single child*** or ***a node with two children***
  - Walk through from the deleted node back to the root and rebalance the nodes on the path if required
    - Since a rotation can change the height of the original tree
- Deletion is O(logN)
  - Each rotation takes O(1) time
  - We may have at most h (height) rotations, where h = O(logN)

AVL Trees -- Deletion

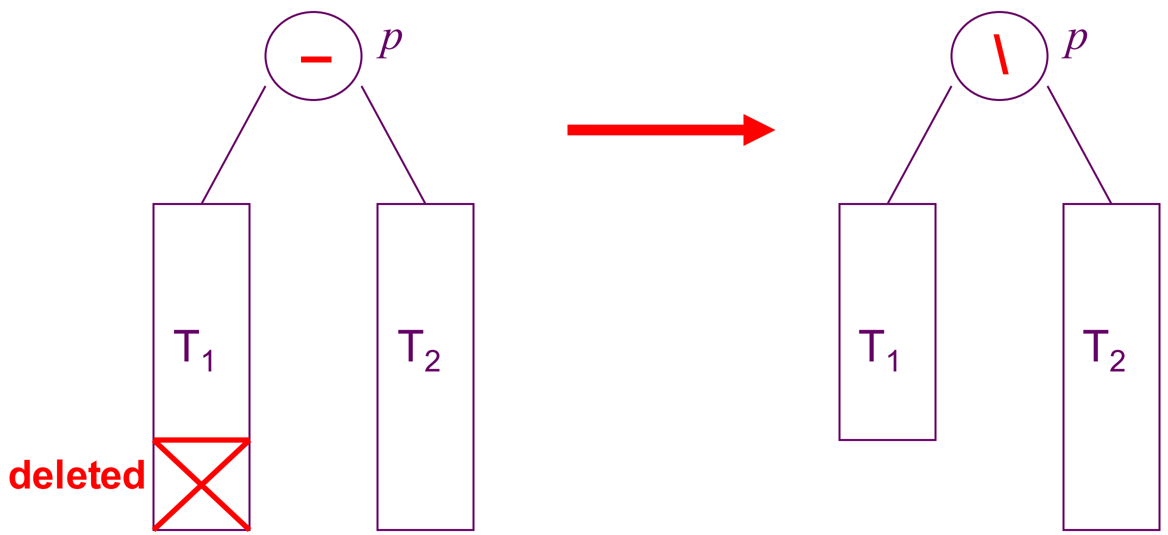
- For the implementation
  - We have a **`shorter`** flag that shows if a subtree has been shortened
  - Each node is associated with a **`balance factor`**
    - ***left-high*** the height of the left subtree is higher than that of the right subtree
    - ***right-high*** the height of the right subtree is higher than that of the left subtree
    - ***equal*** the height of the left and right subtrees is equal

- In the deletion algorithm
  - Shorter is initialized as true
  - Starting from the deleted node back to the root, take an action depending on
    - The value of shorter
    - The balance factor of the current node
    - Sometimes the balance factor of a child of the current node
  - Until shorter becomes false

AVL Trees -- Deletion

Case 1: ***The balance factor of p is equal.***

- Change the balance factor of *p* to *right-high* (or *left-high*)
- Shorter becomes false



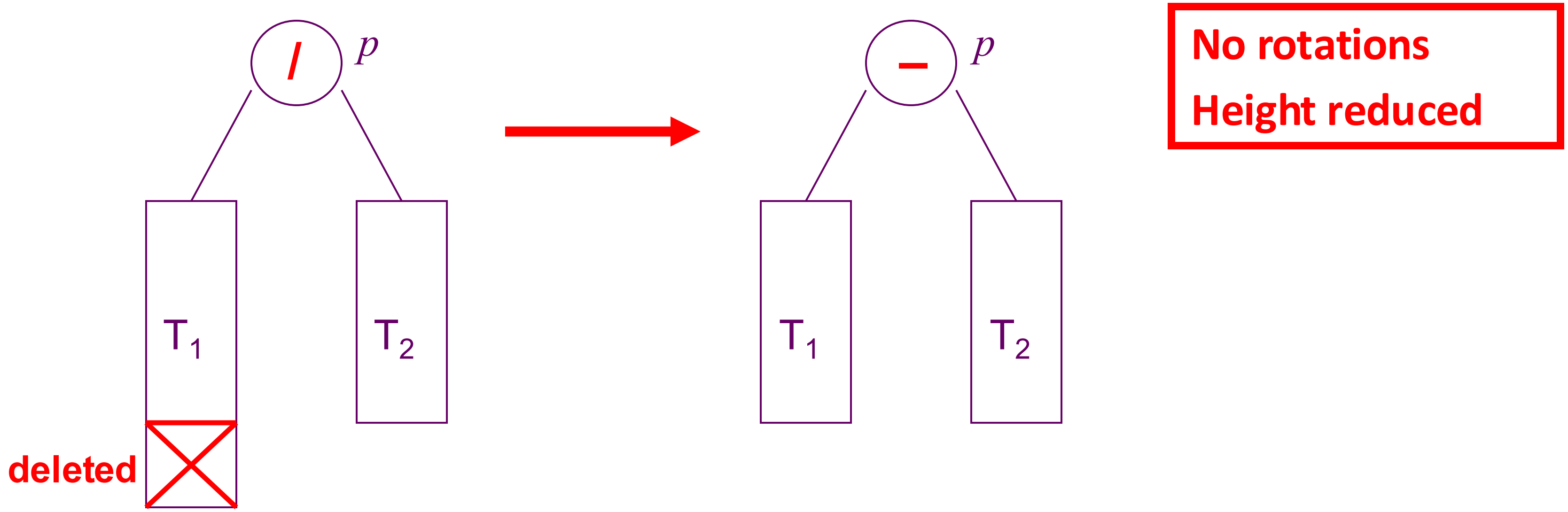
No rotations

Height unchanged

AVL Trees -- Deletion

***Case 2**: The balance factor of p is not equal and the taller subtree is shortened.*

- Change the balance factor of *p* to *equal*
- Shorter remains true



AVL Trees -- Deletion

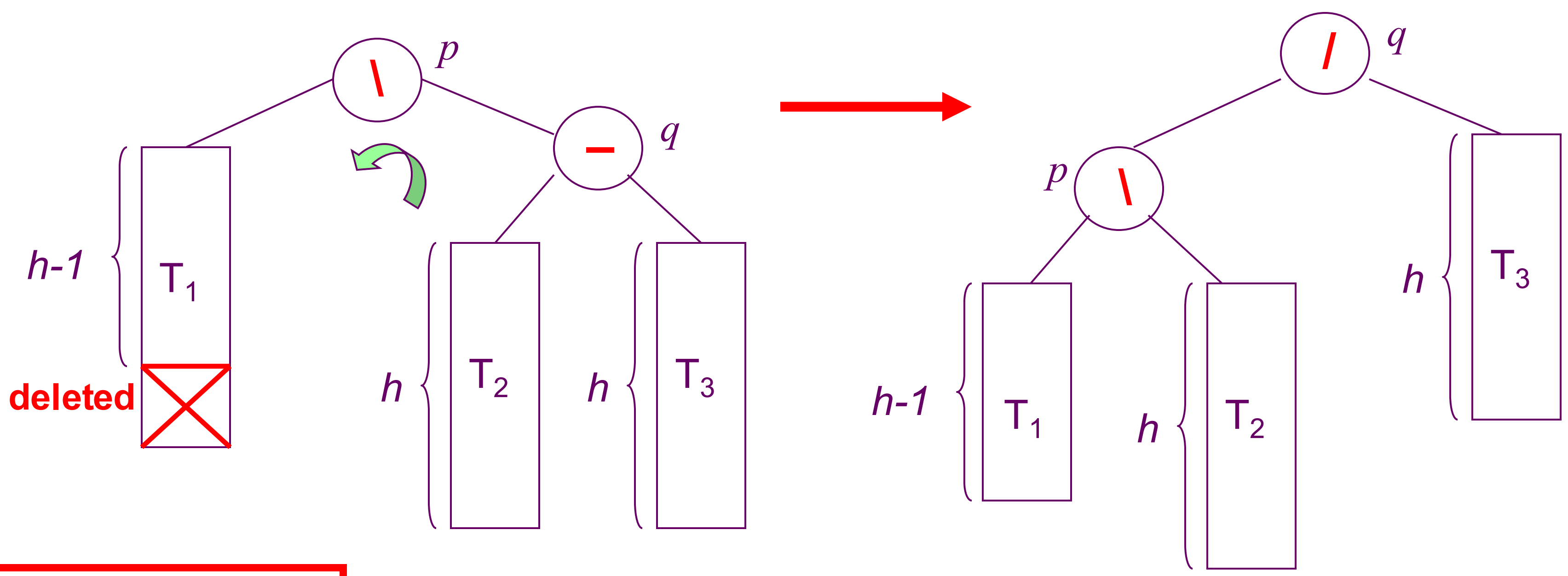
Case 3: ***The balance factor of p is not equal and the shorter subtree is shortened.***

- Rotation is necessary
- Let *q* be the root of the taller subtree of *p*
- We have three sub-cases according to the balance factor of *q*

AVL Trees -- Deletion

Case 3a: ***The balance factor of q is equal.***

- Apply a single rotation
- Change the balance factor of *q* to *left-high* (or *right-high*)
- Shorter becomes false

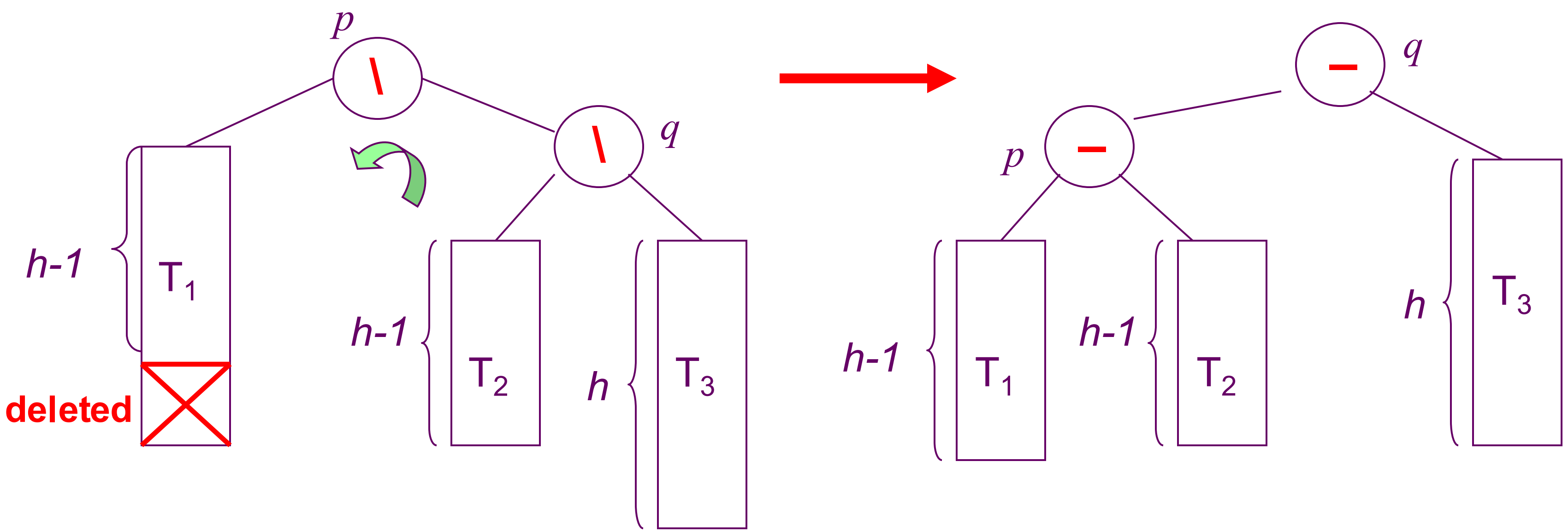


Single rotation
Height unchanged

AVL Trees -- Deletion

<u>Case 3b</u>: *The balance factor of q is the same as that of p.*

- Apply a single rotation
- Change the balance factors of *p* and *q* to *equal*
- Shorter remains true

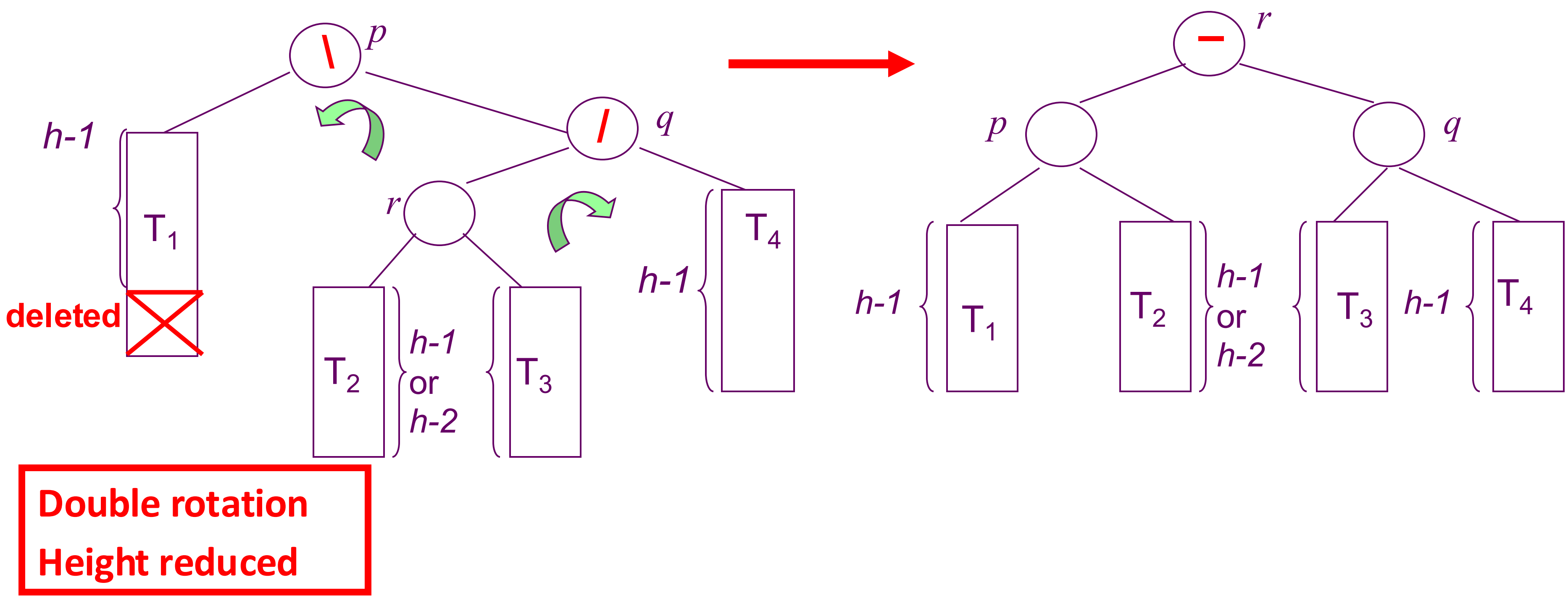


Single rotation
Height reduced

AVL Trees -- Deletion

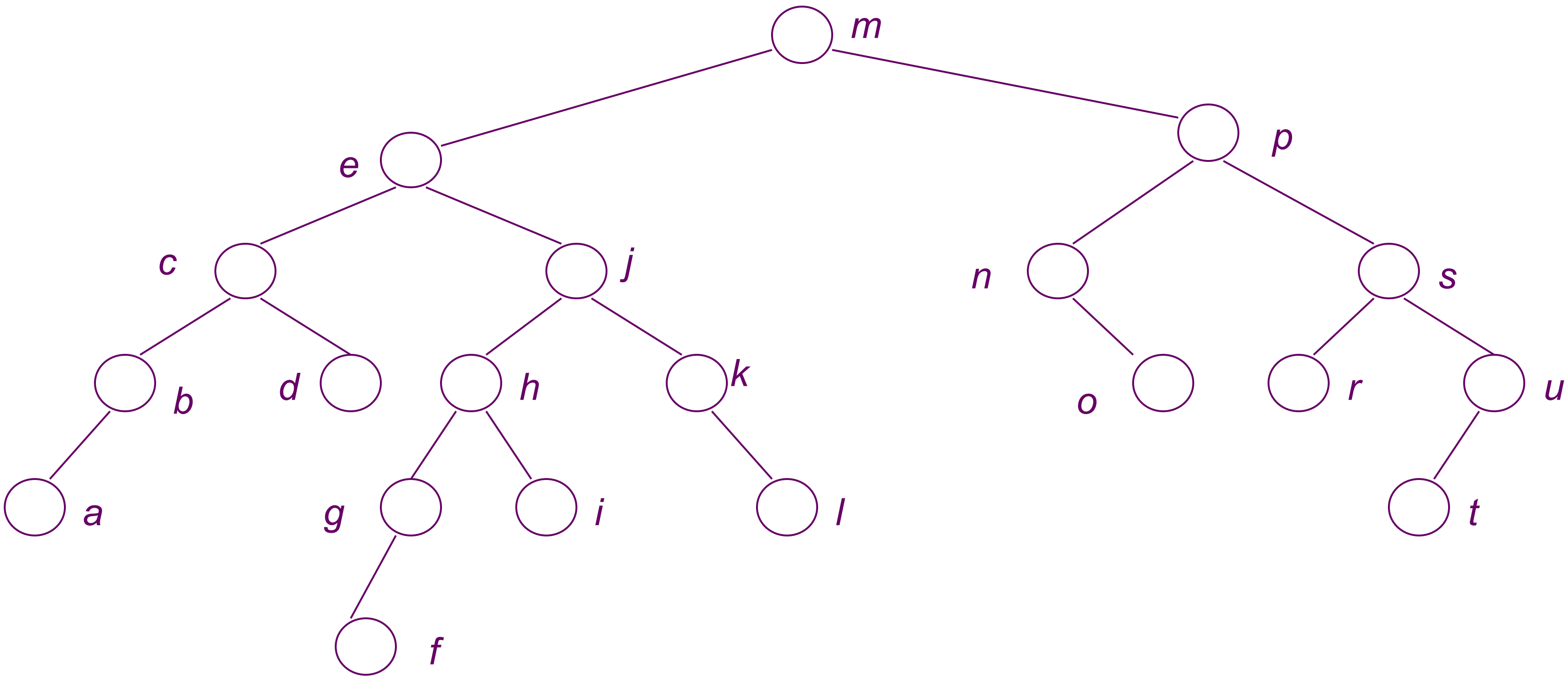
Case 3c: ***The balance factor of q is the opposite of that of p.***

- Apply a double rotation
- Change the balance factor of the new root to *equal*
- Also change the balance factors of *p* and *q*
- Shorter remains true



AVL Trees -- Deletion

***Exercise*:** Delete *o* from the following AVL tree



Efficiency of AVL Trees

- An AVL tree containing n items has a height that is $O(\log_2 n)$.

- Search and insertion are both $O(\log_2 n)$.
  - Search travels at most one path down the tree
  - An insertion goes down one path to the insertion point, and then goes back up adjusting balances/performing rotations
    - in the worst case, both the path down and the path back up consider $O(\log_2 n)$ nodes
    - each rotation requires a constant number of operations, and we perform at most two of them for a given insertion

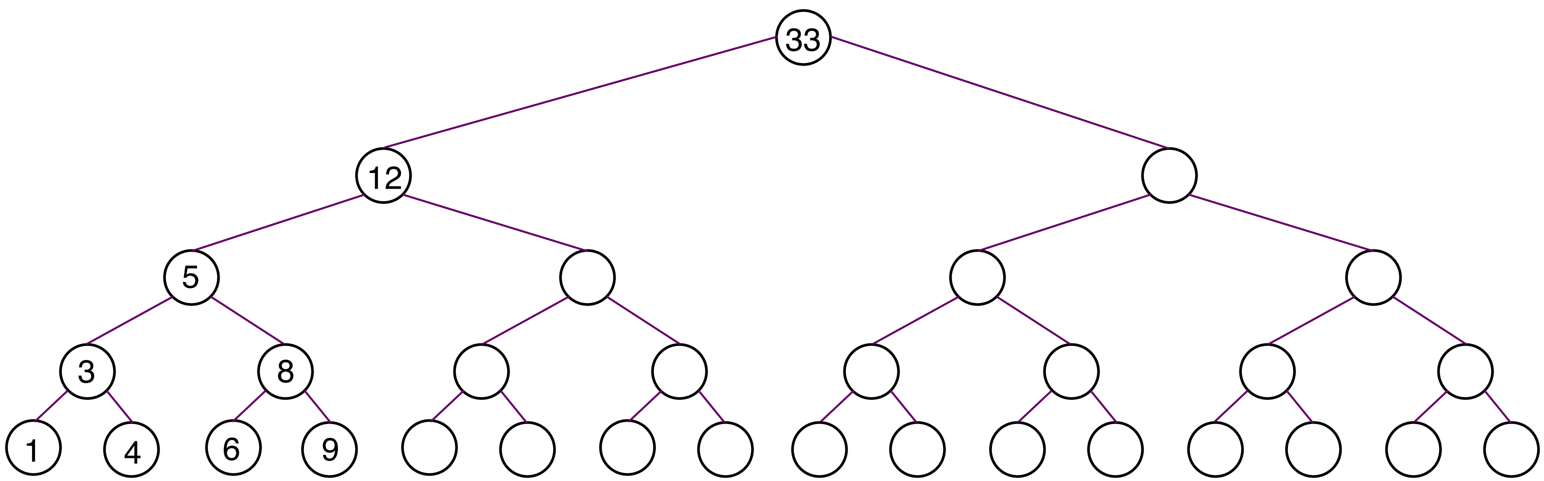
- Deletion begins with the same procedure used in binary search trees ($O(\log_2 n)$ complexity). It can also require rotations to rebalance the tree, also at the complexity of $O(\log_2 n)$.

Other Trees

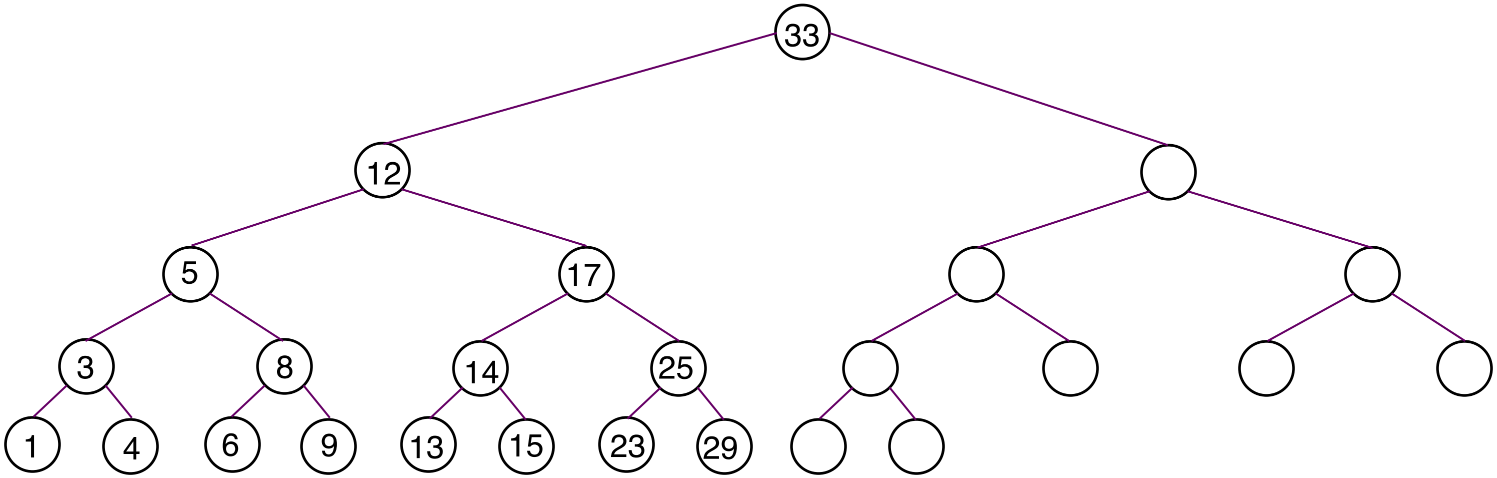
- 2-3 trees
- Splay trees
- Red-black trees
- B-trees

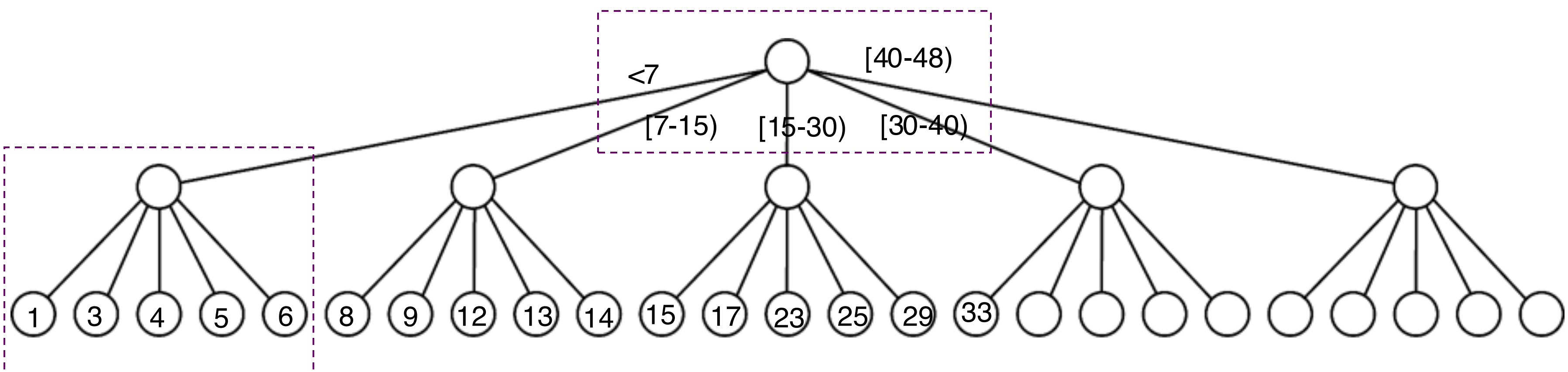
Aren't AVL Trees Perfect?

- AVL tree - binary balanced tree
  - Searching an item requires traversing from root to (at most) leaf
  - logN descents (random accesses)
  - Visiting each node - one comparison.
- On disk:
  - Random access is expensive
  - Disk read is in blocks (e.g. 4K bytes at once)
  - Disk-based AVL trees waste much of a block for an internal node read
    - The node is smaller than the block
    - Payload of the node is not used



Designing for Disk Particulars

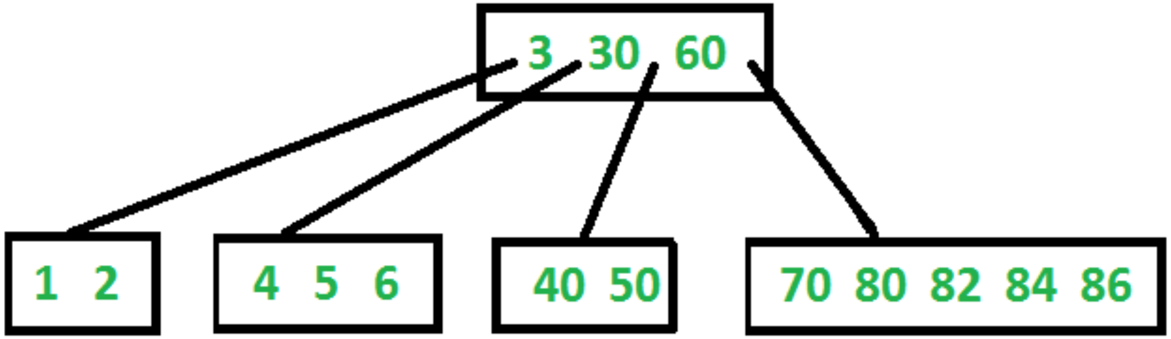




B-Tree

- Reduce the number of disk accesses

- All leaves are at same level
- Defined by its *minimum degree* ‘t’
- **Every node except root must contain at least t-1 keys**
  - Root may contain minimum 1 key
- All nodes (including root) may contain at most 2t – 1 keys
- Number of children of a node is equal to the number of keys in it plus 1



B-Tree – Alternative Definition

- B-Trees are specialized *M*-ary search trees
- Each node has many keys

- maximum branching factor of M
- the root has between 2 and M children *or* at most L keys
- other internal nodes have between $\lceil M/2 \rceil$ and M children
- internal nodes contain only *search* keys (no data)
- each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys
- all leaves are at the same depth

B-Tree Example

B-Tree with $M = 4$

and $L = 4$