2-3-4 Trees

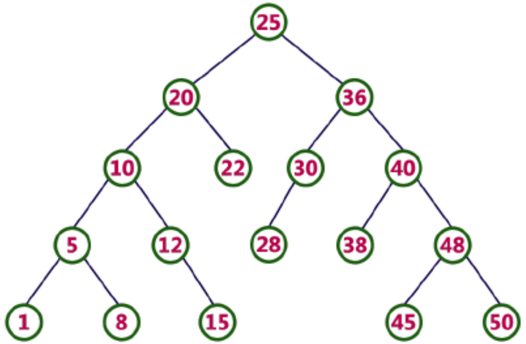
Search Data Structures

- How to search a key in a list of n data items?
  - Linear Search: O(*N*): Find 28 →16 comparisons
    - Unordered items in an array: Search sequentially
    - Unordered / Ordered items in a list: Search sequentially

22	50	20	36	40	15	8	1	45	48	30	10	38	12	25	28	5	END

- Binary Search: O($\log_2 N$): Find 28 → 4 comparisons – 25, 36, 30, 28
- Ordered items in an array: Search by divide-and-conquer
- Binary Search Tree: Recursively on left / right

1	5	8	10	12	15	20	22	25	28	30	36	38	40	45	48	50	END



Search Data Structures

- Worst case time (*N* data items in the data structure):

Data Structure	Search	Insert	Delete	Remarks
Unordered Array	O(n)	O(1)	O(1)	The time to Insert / Delete an item is the time after the location of the item has been ascertained by Search.
Ordered Array	O(log n)	O(n)	O(n)	
Unordered List	O(n)	O(1)	O(1)	
Ordered List	O(n)	O(1)	O(1)	
Binary Search Tree	O(h)	O(1)	O(1)	

- Between an array and a list, there is a trade-off between search and insert/delete complexity
- For a BST of N nodes, $\log N \leq h \leq N$, where h is the height of the tree
- A BST is balanced if $h \sim O(\log N)$: This what we desire

Balanced Binary Search Trees

- A BST is balanced if *h* ~ O(log *N*)

- Balancing guarantees may be of various types:
  - Worst-case: AVL Tree
  - Randomized: Randomized BST, Skip List
  - Amortized: Splay

- These data structures have optimal complexity for all of search, insert and delete: O(log *N*)

- However:
  - Good for in-memory operations
  - Work well for small volume of data
  - Has complex rotation and / or similar operations
  - Do not scale for external data structures

2-3-4 Trees

- All leaves are at the same depth (the bottom level)

- Height, h, of all leaf nodes are same
  - $h \sim O(\log N)$
  - Complexity of search, insert and delete: $O(h) \sim O(\log N)$

- All data is kept in sorted order

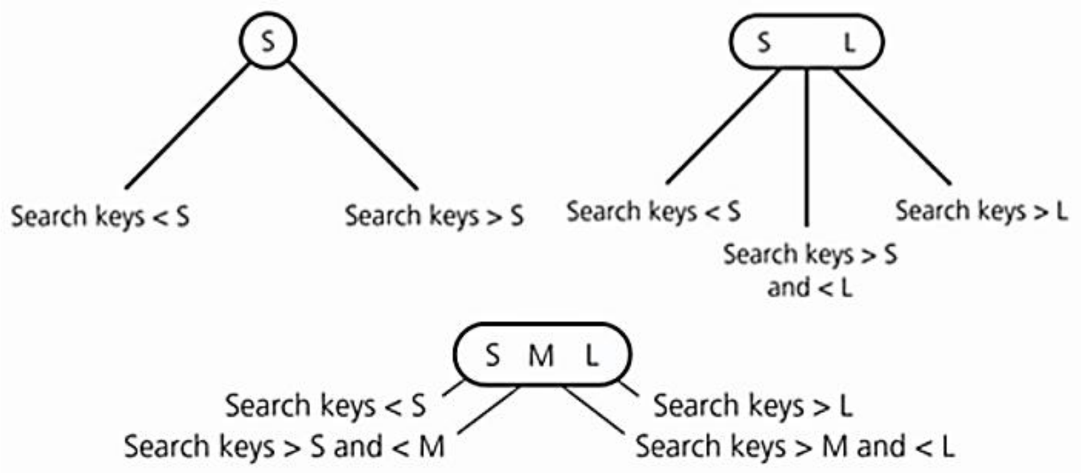
- Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively

- Generalizes easily to larger nodes

- Extends to external data structures

2-3-4 Trees

- Uses 3 kinds of nodes satisfying key relationships as shown below:
  - A 2-node must contain a single data item (S) and two links
  - A 3-node must contain two data items (S, L) and three links
  - A 4-node must contain three data items (S, M, L) and four links
  - A leaf may contain either one, two, or three data items



2-3-4 Trees: Search

- Search
  - Simple and natural extension of search in BST

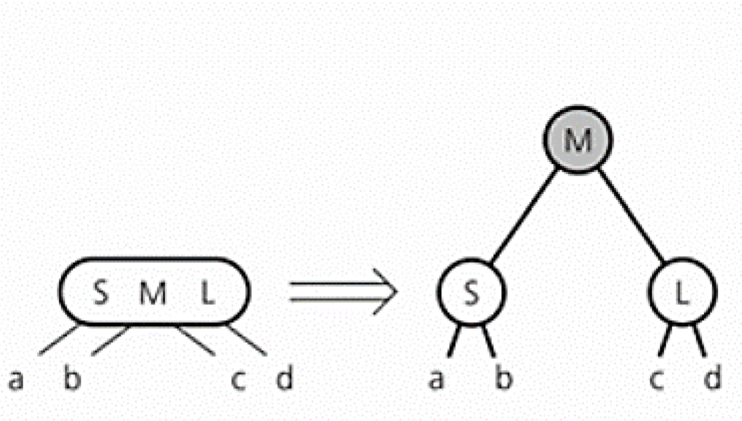
2-3-4 Trees: Insert

- Insert
  - Search to find expected location
    - If it is a 2 node, change to 3 node and insert
    - If it is a 3 node, change to 4 node and insert
    - If it is a 4 node, split the node by moving the middle item to parent node, then insert

- Node Splitting
  - A 4-node is split as soon as it is encountered during a search from the root to a leaf
  - The 4-node that is split will
    - Be the root, or
    - Have a 2-node parent, or
    - Have a 3-node parent

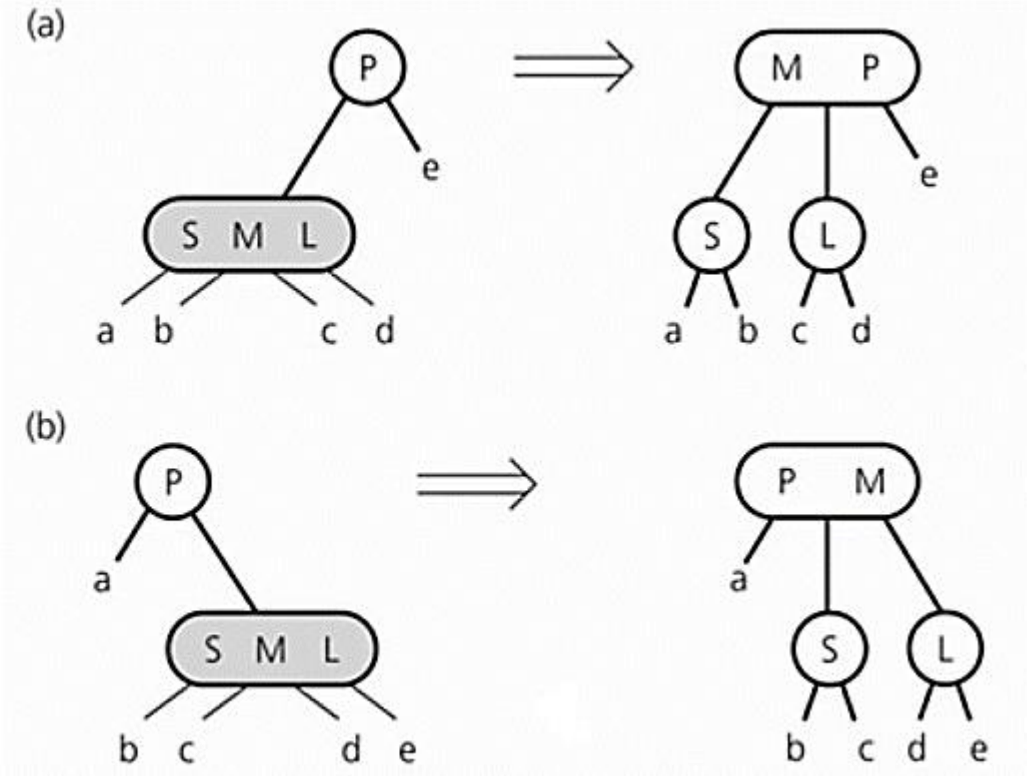
2-3-4 Trees: Insert

- Splitting at Root



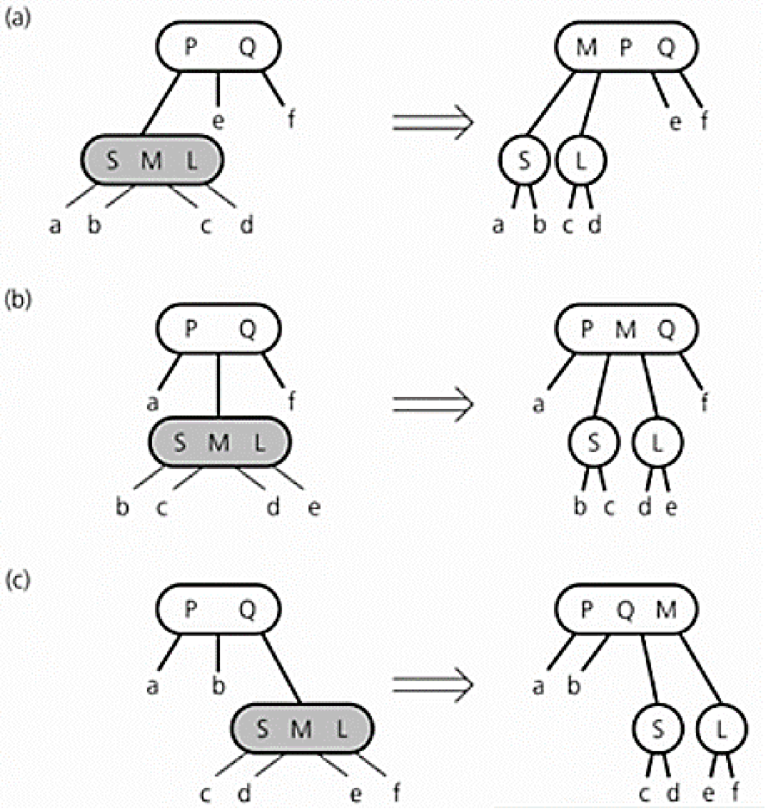
2-3-4 Trees: Insert

- Splitting with 2 node parent



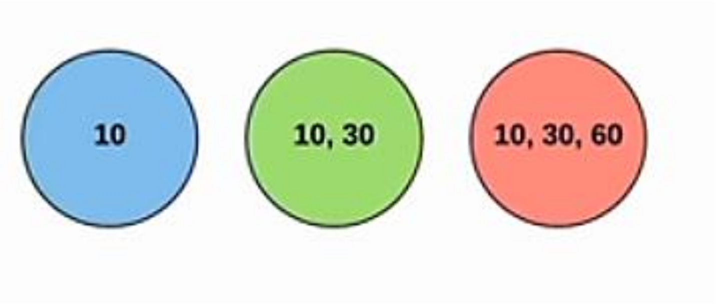
2-3-4 Trees: Insert

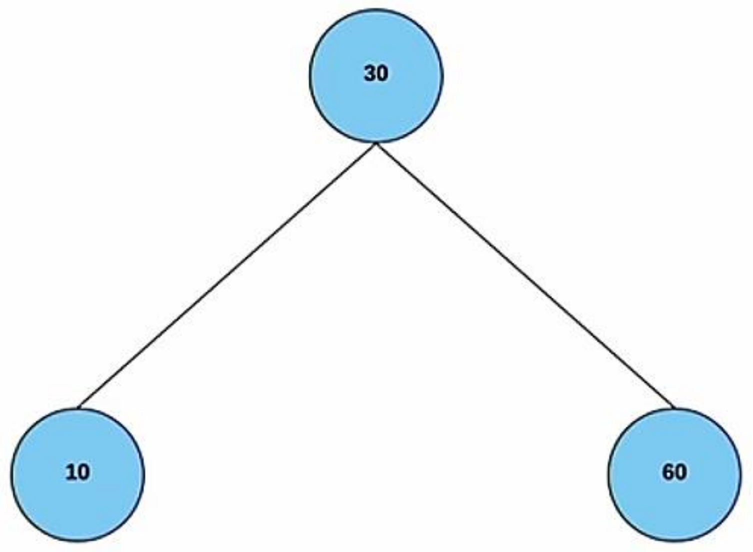
- Splitting with 3 node parent



2-3-4 Trees: Insert: Example

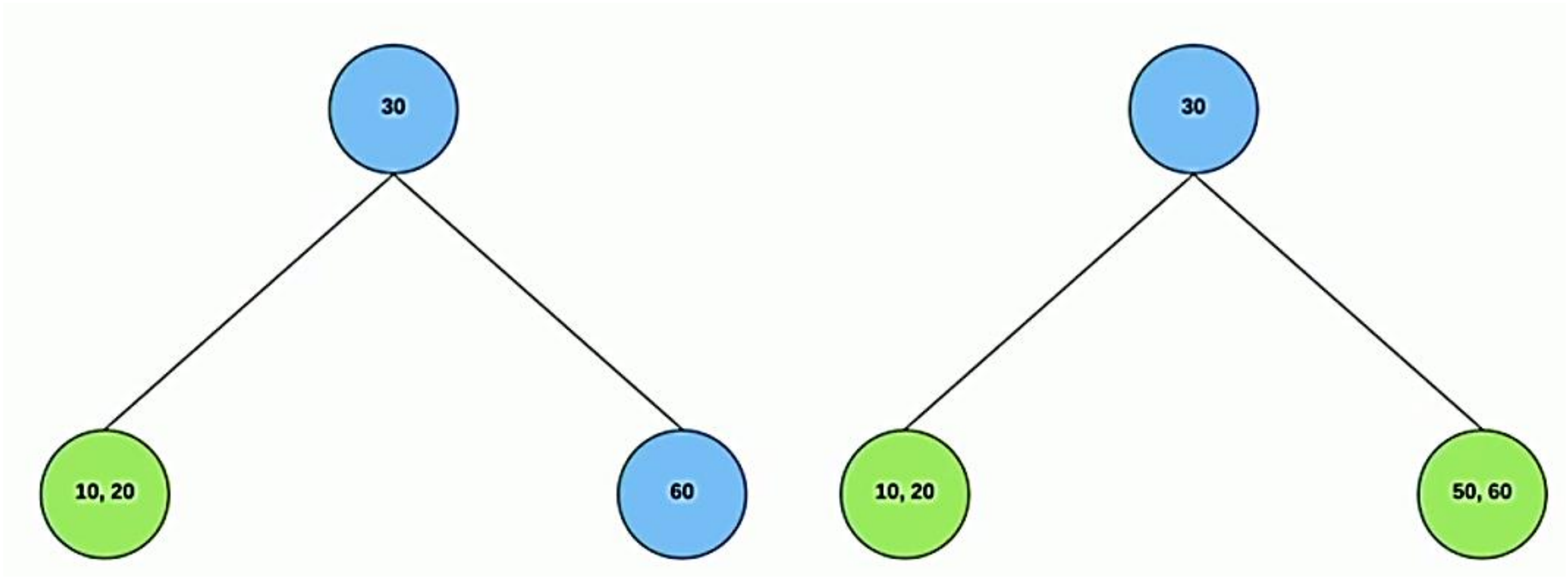
- Insert 10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100
- 10
- 10, 30
- 10, 30, 60
- Split for 20





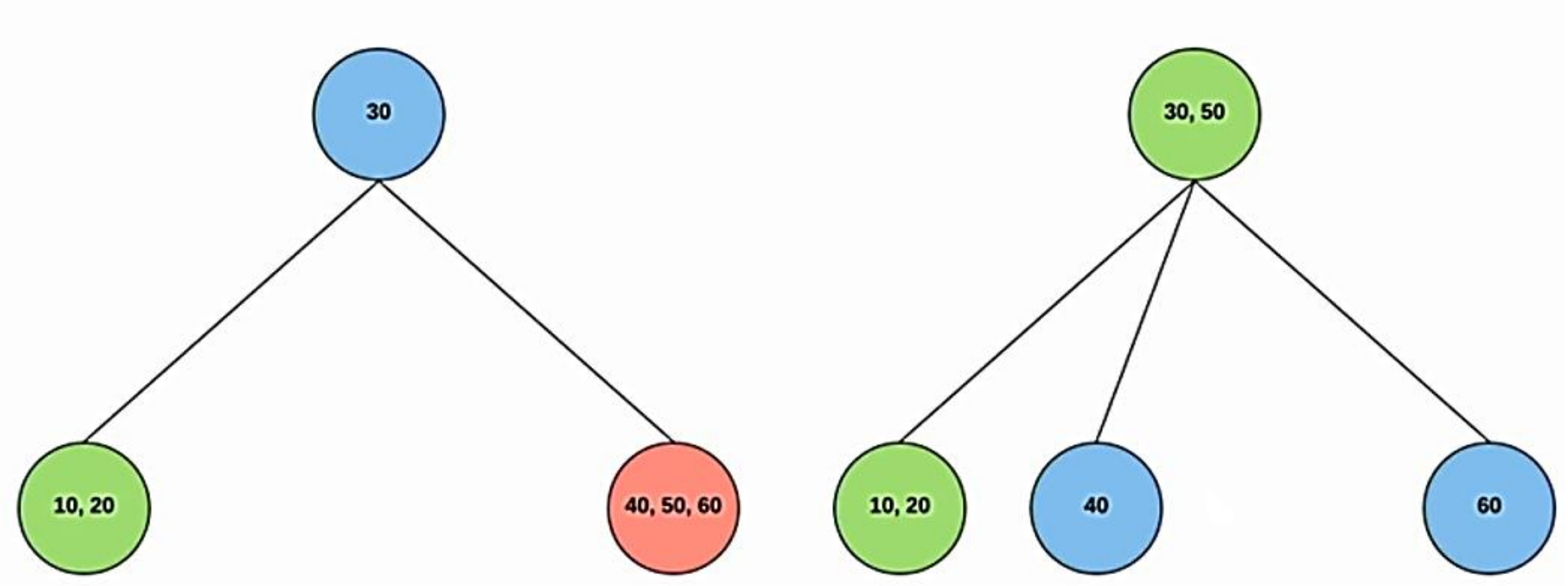
2-3-4 Trees: Insert: Example

- 10, 30, 60, 20
- 10, 30, 60, 20, 50



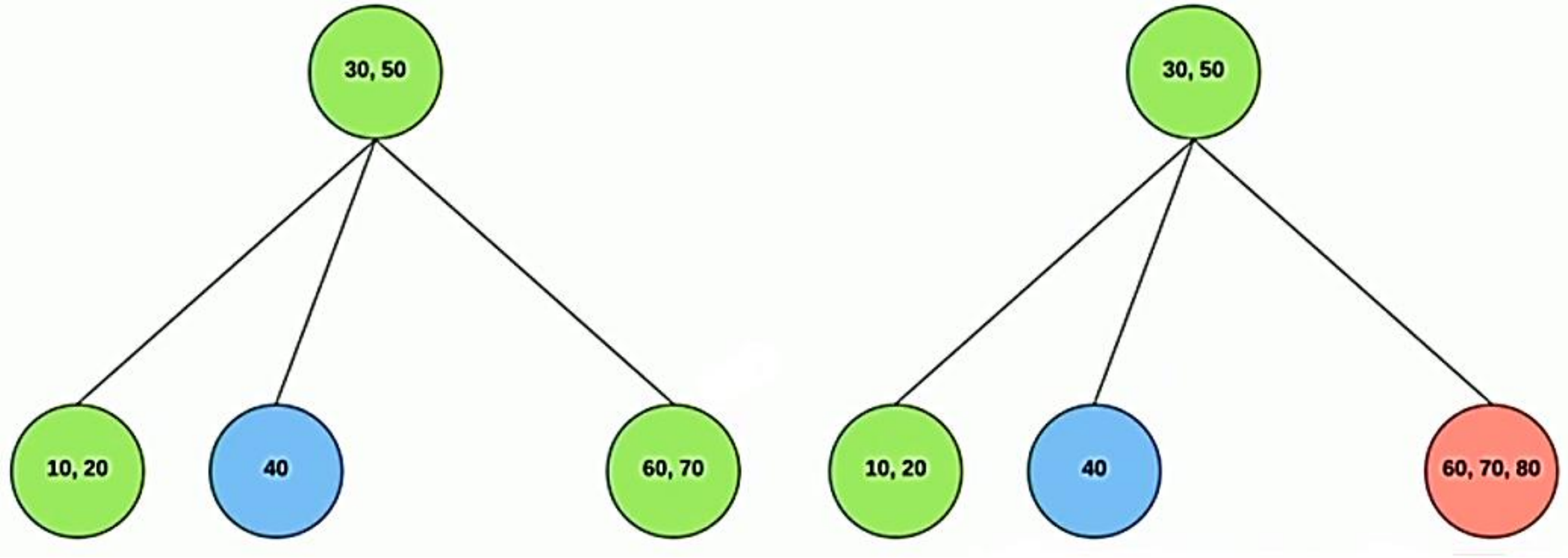
2-3-4 Trees: Insert: Example

- 10, 30, 60, 20, 50, 40
- Split for 70



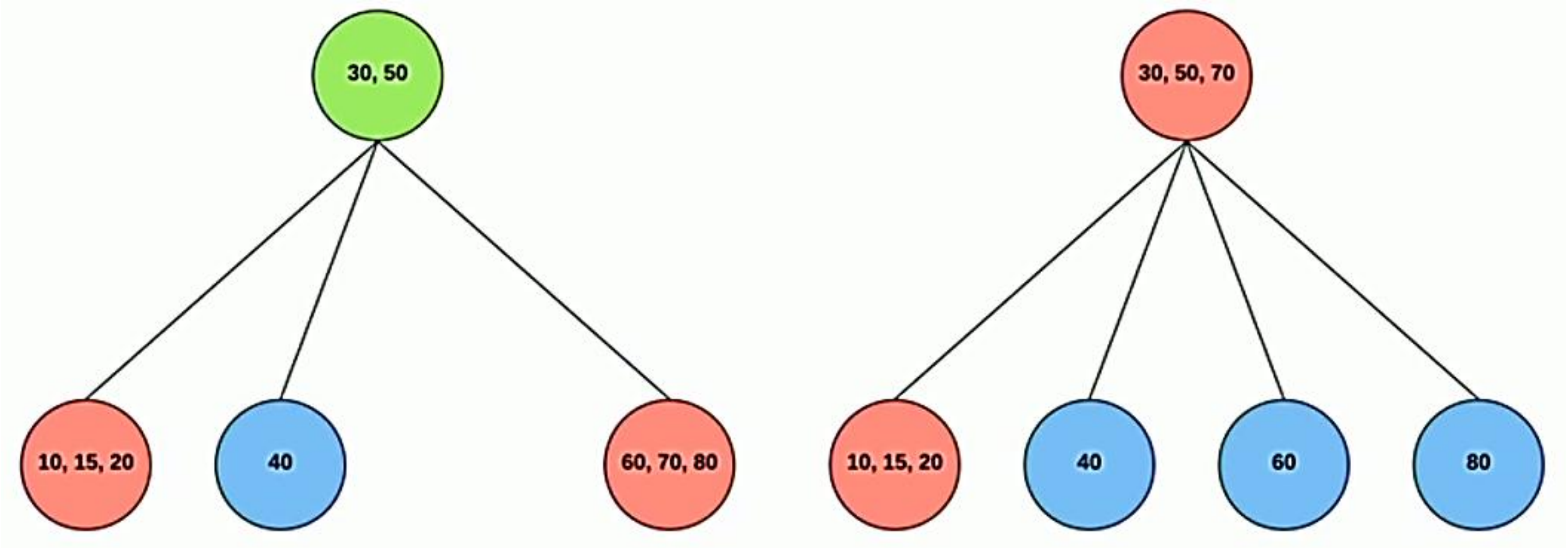
2-3-4 Trees: Insert: Example

- 10, 30, 60, 20, 50, 40, 70
- 10, 30, 60, 20, 50, 40, 70, 80



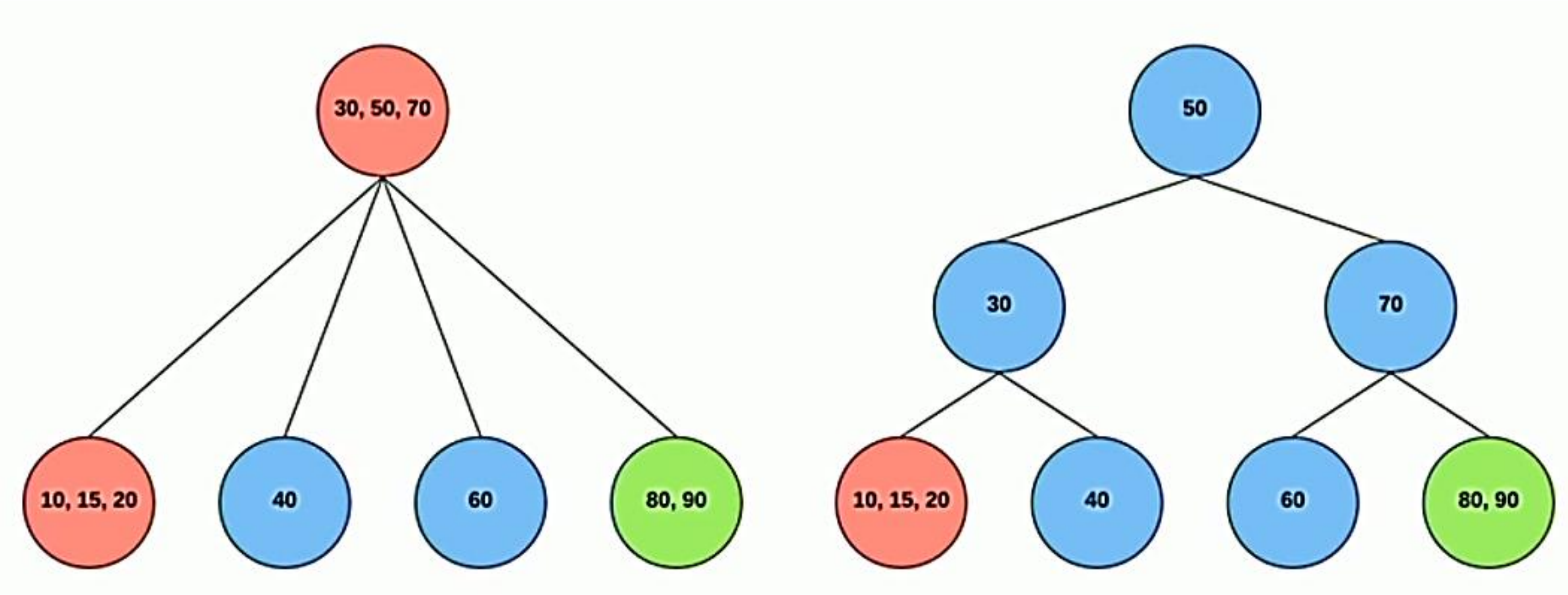
2-3-4 Trees: Insert: Example

- 10, 30, 60, 20, 50, 40, 70, 80, 15
- Split for 90



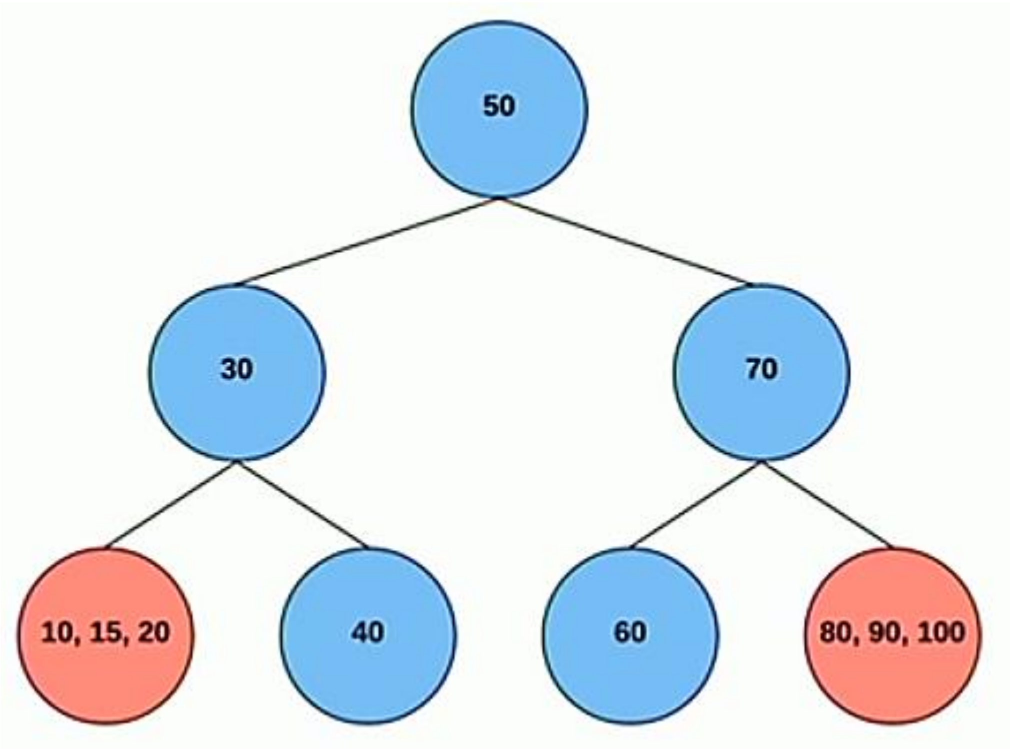
2-3-4 Trees: Insert: Example

- 10, 30, 60, 20, 50, 40, 70, 80, 15, 90
- Split for 100



2-3-4 Trees: Insert: Example

- 10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100



2-3-4 Trees: Delete

- Delete
  - Locate the node *n* that contains the item *theItem*
  - Find *theItem*'s inorder successor and swap it with *theItem* (deletion will always be at a leaf)
  - If that leaf is a 3-node or a 4-node, remove *theItem*
  - To ensure that *theItem* does not occur in a 2-node
    - Transform each 2-node encountered into a 3-node or a 4-node
    - Reverse different cases illustrated for splitting

2-3-4 Trees

- Advantages
  - All leaves are at the same depth (the bottom level): Height, $h \sim O(\log N)$
  - Complexity of search, insert and delete: $O(h) \sim O(\log N)$
  - All data is kept in sorted order
  - Generalizes easily to larger nodes
  - Extends to external data structures

- Disadvantages
  - Uses variety of node types: Need to destruct and construct multiple nodes for converting a 2 Node to 3 Node, a 3 Node to 4 Node, for splitting etc.

2-3-4 Trees

- Consider only one node type with space for 3 items and 4 links
  - Internal node (non-root) has 2 to 4 children (links)
  - Leaf node has 1 to 3 items
  - Wastes some space, but has several advantages for external data structure

- Generalizes easily to larger nodes
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil N/2 \rceil$ and N children
  - A leaf node has between $\lceil (N - 1)/2 \rceil$ and $N - 1$ values
  - Special cases:
  - If the root is not a leaf, it has at least 2 children
  - If the root is a leaf, it can have between 0 and $(N - 1)$ values

- Extends to external data structures
  - B-tree
  - 2-3-4 Tree is a B-tree where $N = 4$

Next Lecture

B$^+$ Tree and B Tree

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in