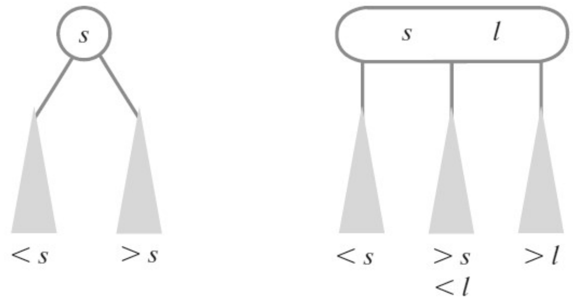
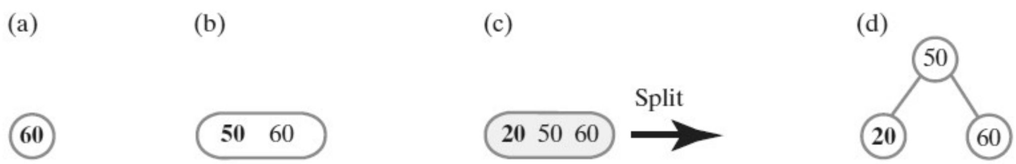
2-3 Trees

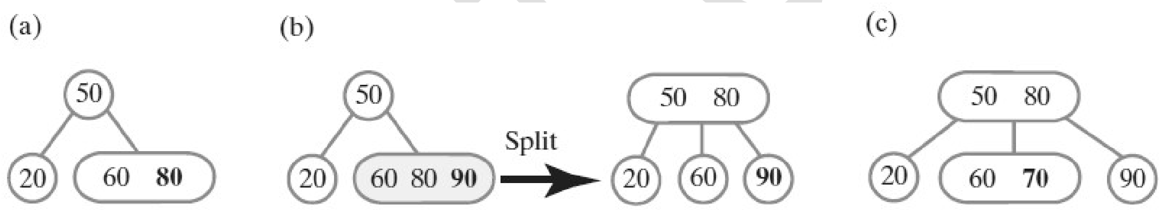
- **Definition**: general search tree whose interior nodes must have either **2** or **3** children.
  - A **2-node** contains one data item ***s*** and has two children.
  - A **3-node** contains two data items, ***s*** and ***l***, and has three children.



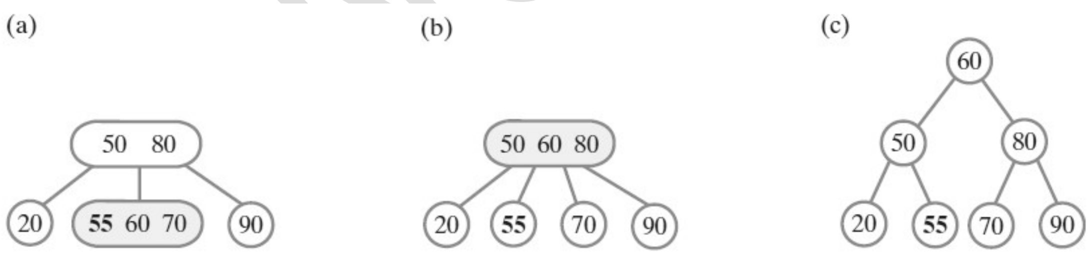
Adding Entries to a 2-3 Tree:



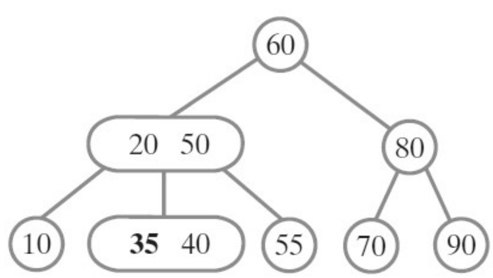
Adding (a) **60** and (b) **50**; (c), (d) adding **20** causes the 3-node to split



The 2-3 tree after adding (a) **80**; (b) **90**; (c) **70**



Adding **55** to the 2-3 tree, causes a leaf and then the root to split

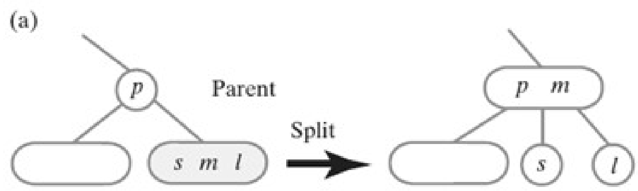


The 2-3 tree, after adding **10, 40, 35**

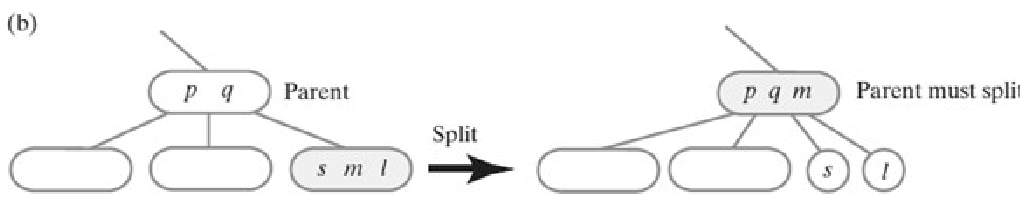
Splitting Nodes during Addition:

- Splitting a **leaf** to accommodate a new entry when the leaf's **parent** contains:

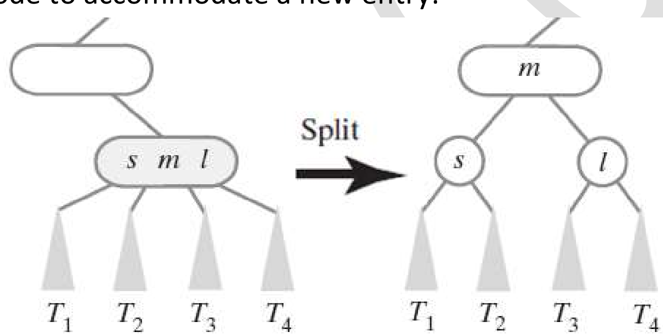
  (a) one entry:



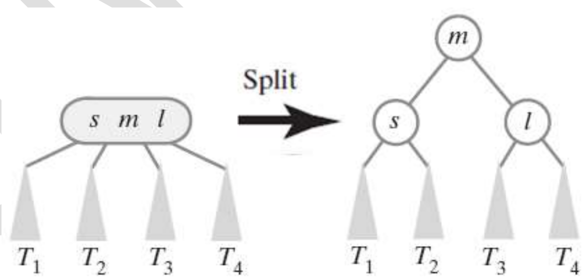
  (b) two entries:



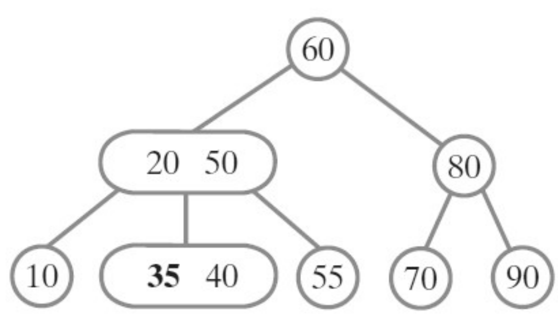
- Splitting an internal node to accommodate a new entry:



- Splitting the root to accommodate a new entry:



Searching a 2-3 Tree:



2-3 tree: performance:

2-3 tree is a perfect balanced tree: Every path from **root** to a **leaf** has same length.

Tree height:

- Worst case: **log N**. [all 2-nodes]
- Best case: **$\log_3$ N ≈ .631 log N**. [all 3-nodes]
  - Between 12 and 20 for a million nodes.
  - Between 18 and 30 for a billion nodes.

2-3 tree: implementation?

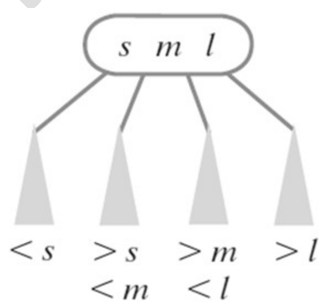
Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

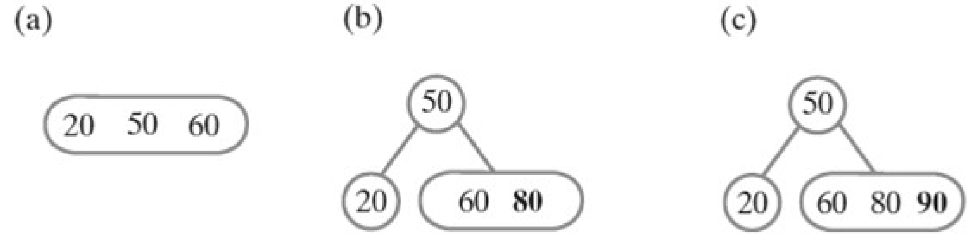
Exercise: 50 60 70 40 30 20 10 80 90 100

2-4 Trees

- Sometimes called a 2-3-4 tree.
  - General search tree
  - Interior nodes must have either two, three, or four children
  - Leaves occur on the same level
  - A 4-node contains three data items ***s***, ***m***, and ***l*** and has four children.

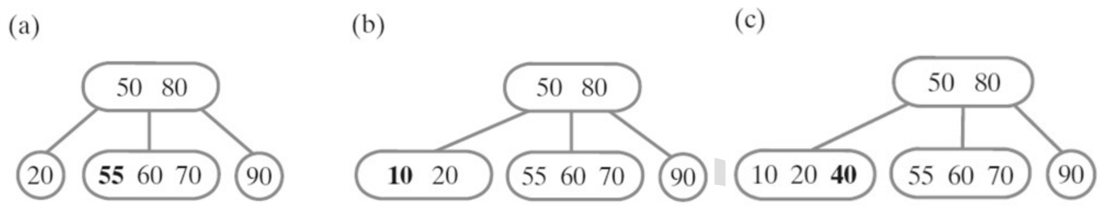


Adding Entries to a 2-4 Tree



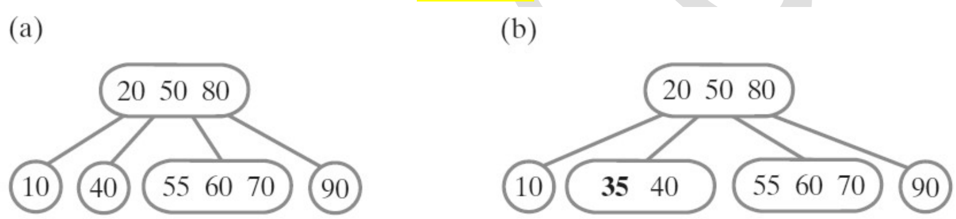
The 2-4 tree, after (a) adding **20**, **50**, and **60** (b) adding **80** and splitting the root; (c) adding **90**

Adding 70



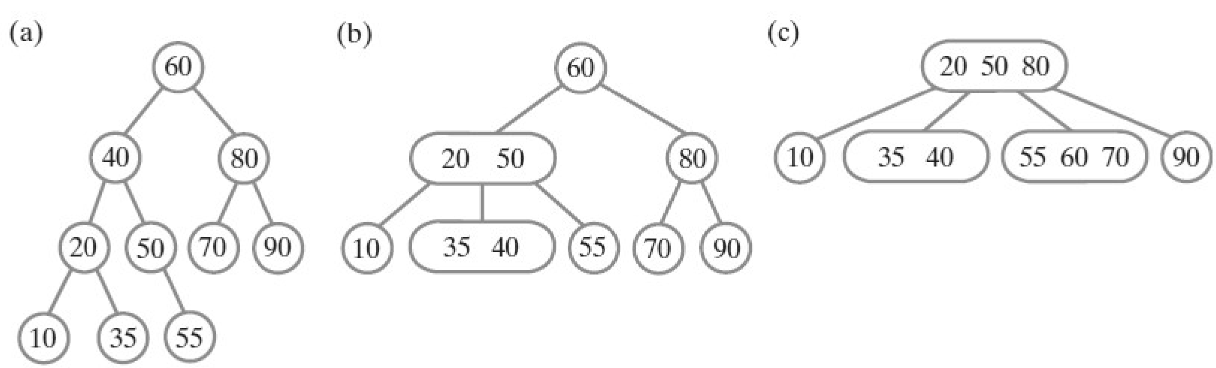
The 2-4 tree after adding (a) **55**; (b) **10**; (c) **40**

Adding 5



The 2-4 tree after (a) splitting the leftmost 4-node; (b) adding **35**

Comparing AVL, 2-3, and 2-4 Trees:



Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35:
(a) AVL tree; (b) 2-3 tree; (c) 2-4 tree

B-Trees

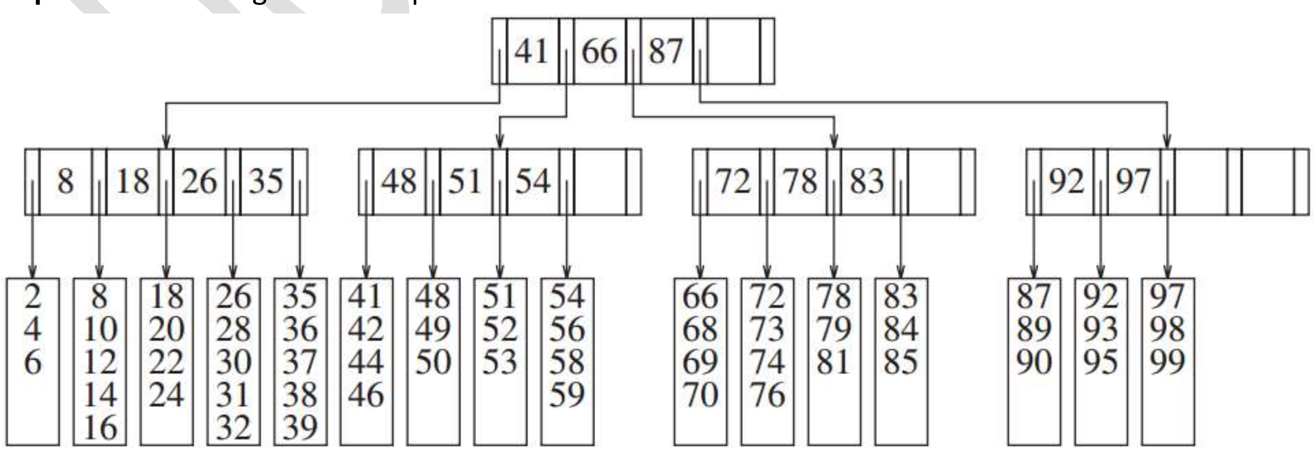
B-trees (Bayer-McCreight, 1972)

- **Definition**: multiway search tree of order ***m***
  - A general tree whose nodes have up to ***m*** children each
- A binary search tree is a multiway search tree of order 2. In a binary search tree, we need one key to decide which of two branches to take. In an M-ary search tree, we need M−1 keys to decide which branch to take.
- 2-3 trees and 2-4 trees are balanced multiway search trees of orders 3 and 4, respectively.
- As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly $\log_2 N$, a complete M-ary tree has height that is roughly $\log_M N$.
- The B-tree is the most popular data structure for disk bound searching.
- To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way.
- Additional properties to maintain balance:
  - The **root** has either no children or between **2** and ***m*** children.
  - Other interior nodes (non-leaves) have between $\lceil m/2 \rceil$ and ***m*** children each.
  - All leaves are on the same level.

A B-tree of order ***M*** is an ***M-ary*** tree with the following properties: (**B⁺ tree**)

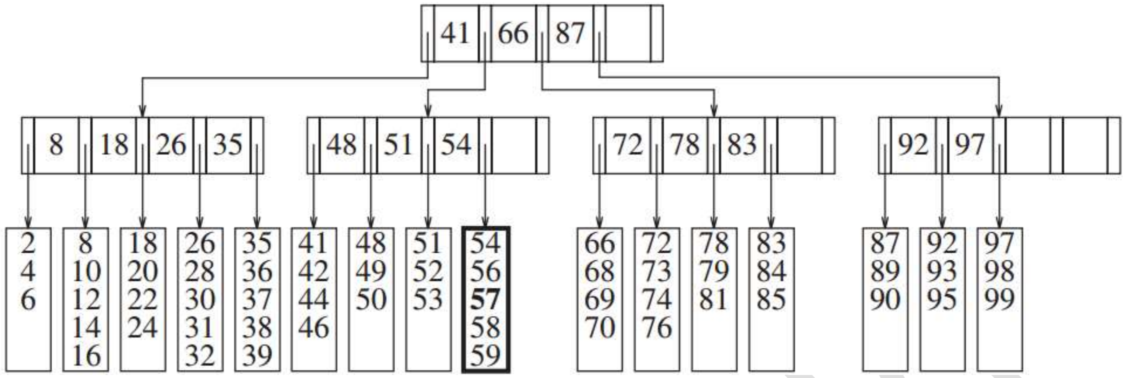
1. The data items are stored at leaves.
2. The non-leaf nodes store up to ***M−1*** keys to guide the searching; key ***i*** represents the smallest key in subtree ***i+1***.
3. The **root** is either a leaf or has between two and ***M*** children.
4. All non-leaf nodes (except the **root**) have between ***M/2*** and ***M*** children.
5. All leaves are at the same depth and have between ***L/2*** and ***L*** data items, for some ***L*** (the determination of L is described shortly).

Example: The following is an example of a B^+ tree of order ***5*** and ***L=5***

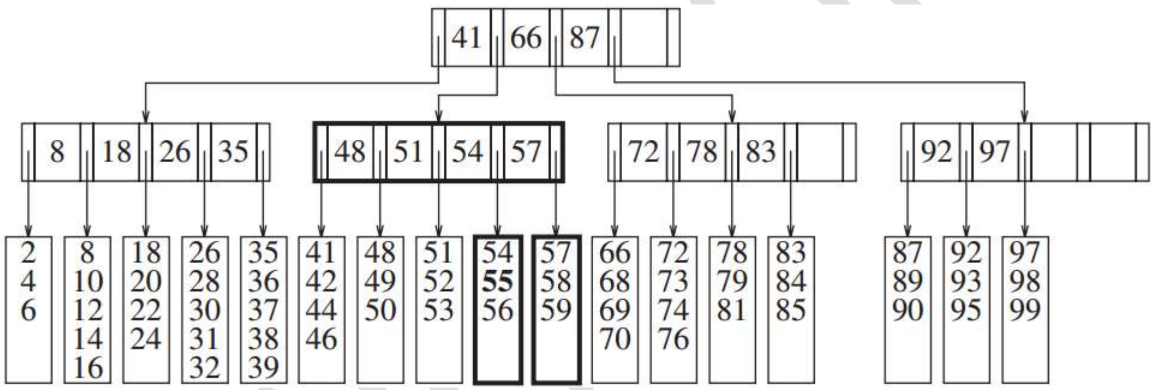


Add items from the B^{+} tree:

- **Insert 57**: A search down the tree reveals that it is not already in the tree. We can then add it to the leaf as a fifth item:

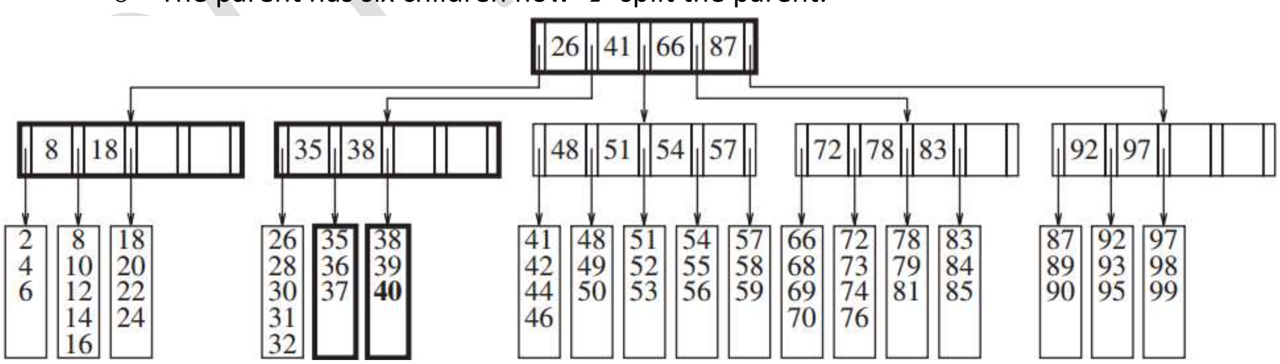


- **Insert 55**: The leaf where 55 wants to go is already full. Solution: split them into two leaves:



Note: The node splitting in the previous example worked because the parent did not have its full complement of children.

- **Insert 40**: We have to split the leaf containing the keys 35 through 39, and now 40, into two leaves.
  - The parent has six children now ➔ split the parent.



Note:
  - When the parent is split, we must update the values of the keys and also the parent's parent.
  - if the parent already has reached its limit of children? In that case, we continue splitting nodes up the tree until either we find a parent that does not need to be split or we reach the root. Then we split the root and this will generate a new level.