2-3 Trees & Red-Black Trees

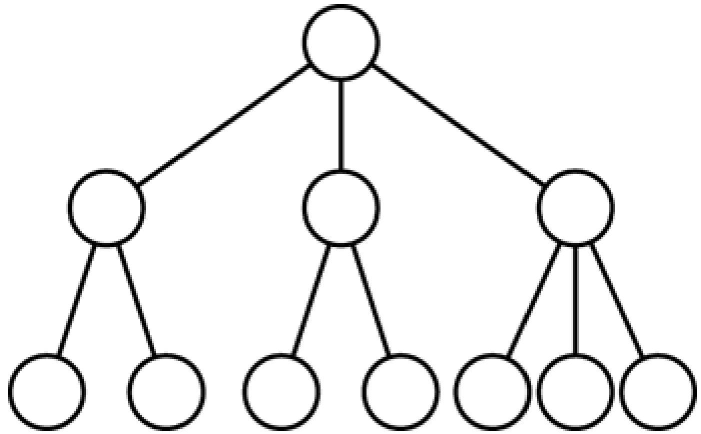
Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

2-3 Trees

Definition:

A 2-3 tree is a tree in which each internal node has either two or three children, and all leaves are at the same level.

- **2-node:** a node with two children
- **3-node:** a node with three children



An example of a 2-3 tree

no 1 or 0 child allowed

grows/shrink from the root side

→A 2-3 tree is not a binary tree

→A 2-3 tree is never taller than a minimum-height binary tree

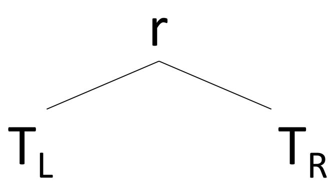
→A 2-3 tree with N nodes never has height greater than $\lceil \log 2(N+1) \rceil$

→A 2-3 tree of height h always has at least 2^h-1 nodes.

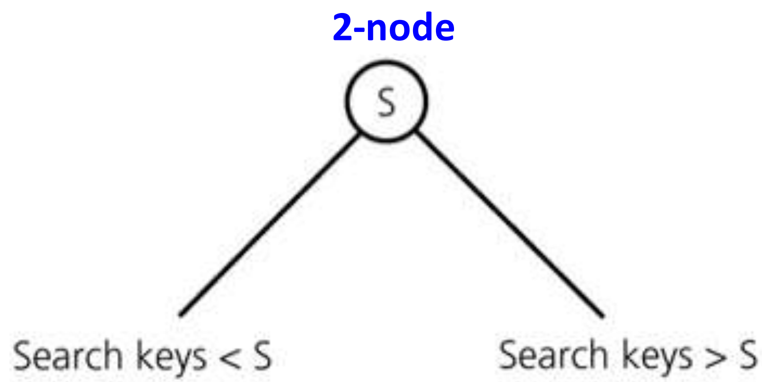
2-3 Trees

T is a 2-3 tree of height h if

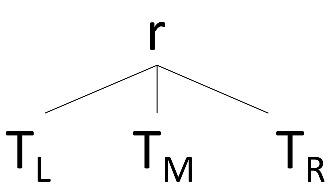
1. T is empty (a 2-3 tree of height 0), or
2. T is of the form



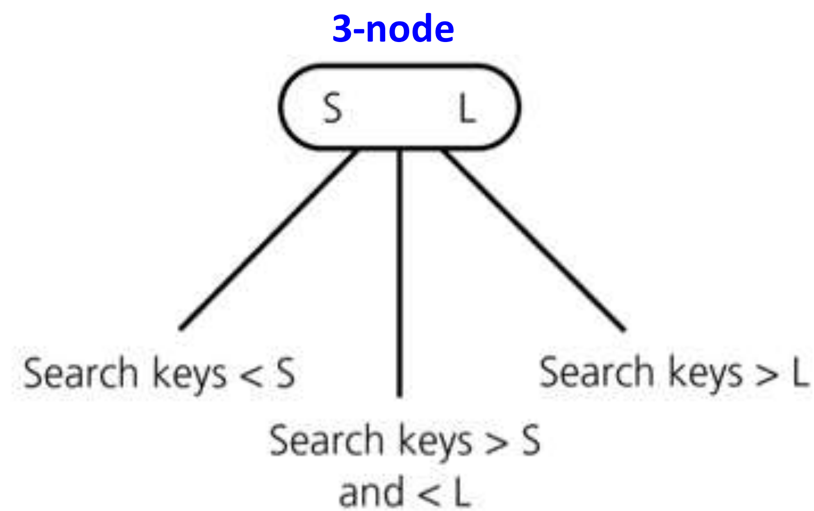
   where r is a node that contains one data item and T_L and T_R are both 2-3 trees, each of height h-1, or



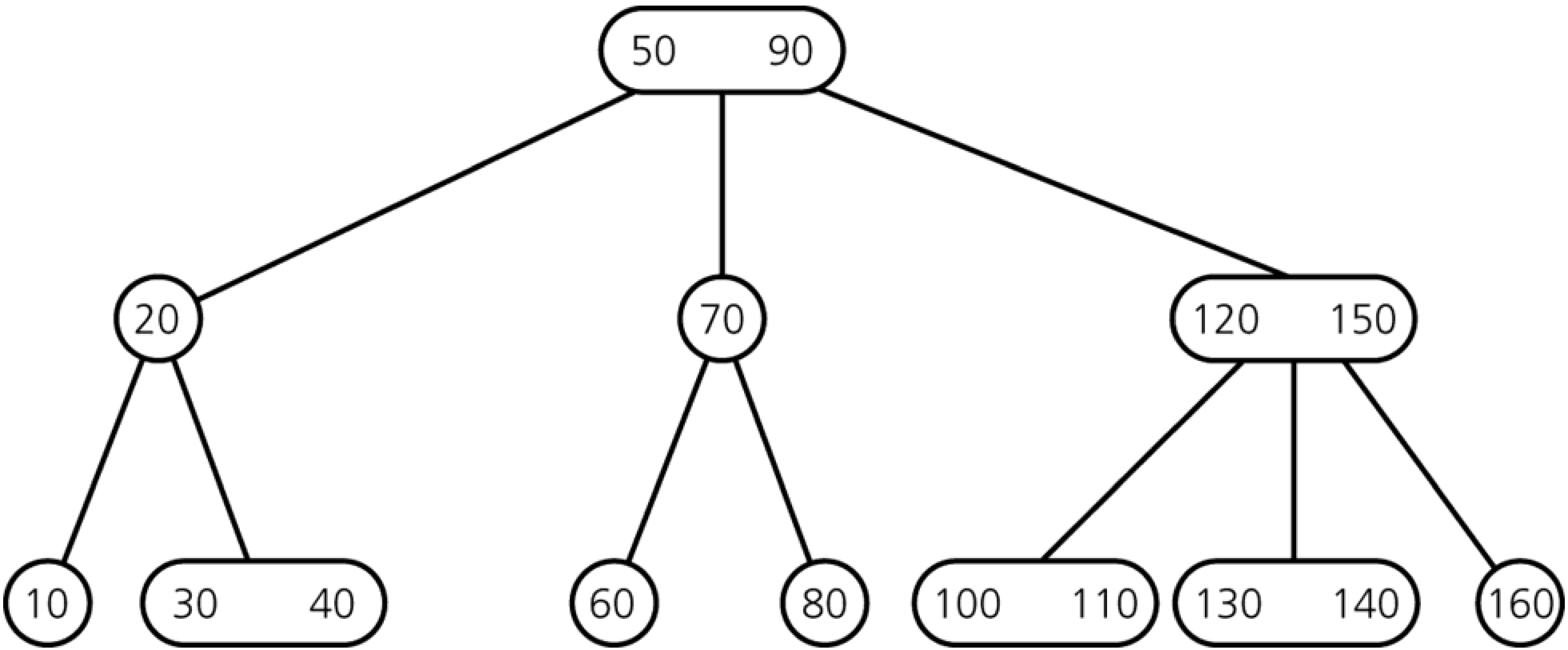
3. T is of the form



   where r is a node that contains two data items and T_L , T_M and T_R are 2-3 trees, each of height h-1.



2-3 Trees -- Example



C++ Class for a 2-3 Tree Node

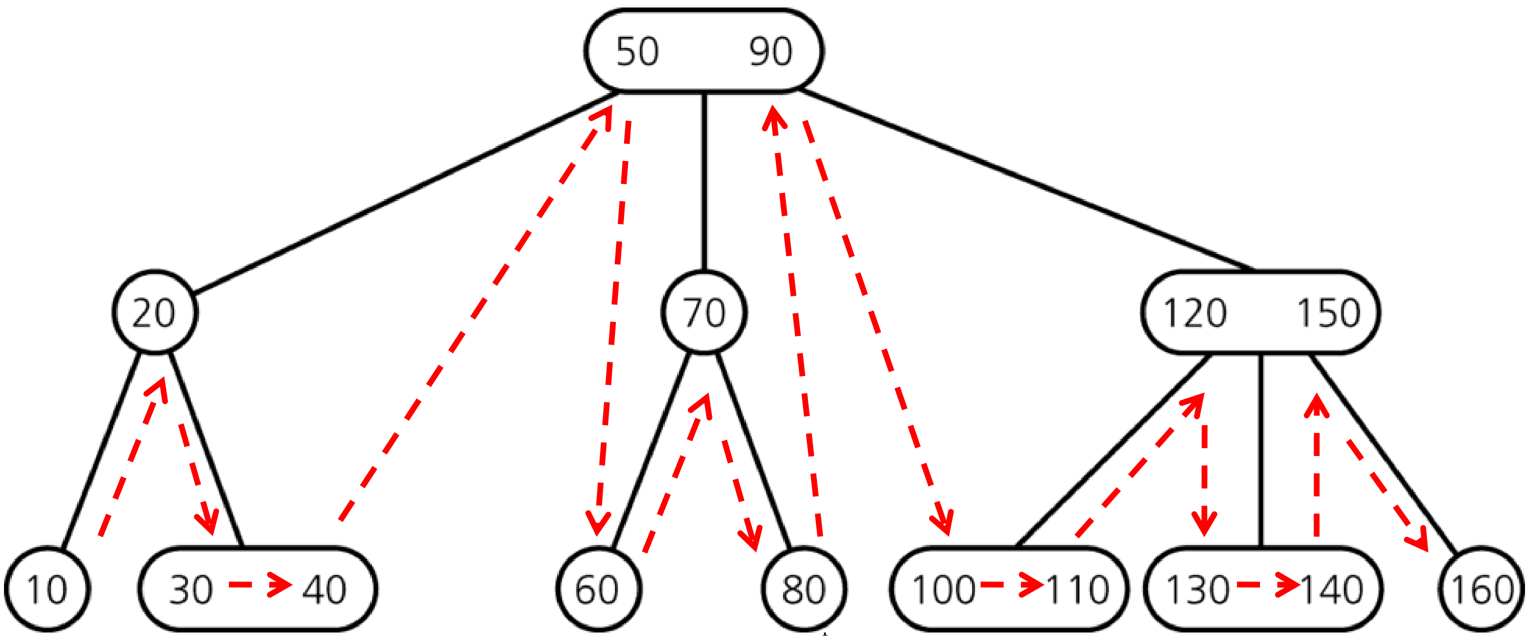
```
classTreeNode {
private:
   TreeItemTypesmallItem, largeItem;
   TreeNode *leftChildPtr, *midChildPtr, *rightChildPtr;

   // friend class-can access private class members
   friend classTwoThreeTree;
};
```

extra storage

- When a node is a 2-node (contains only one item)
  - Place it in `smallItem`
  - Use `leftChildPtr` and `midChildPtr` to point to the node's children
  - Place `NULL` in `rightChildPtr`

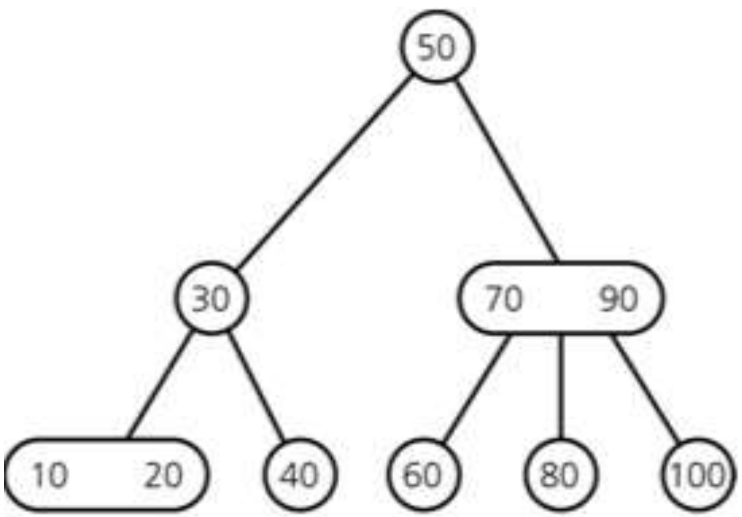
Traversing a 2-3 Tree

- Inorder traversal visits the nodes in a sorted search-key order
  - Leaf node:
    - Visit the data item(s)
  - 2-node:
    - Visit its left subtree
    - Visit the data item
    - Visit its right subtree
  - 3-node:
    - Visit its left subtree
    - Visit the smaller data item
    - Visit its middle subtree
    - Visit the larger data item
    - Visit its right subtree

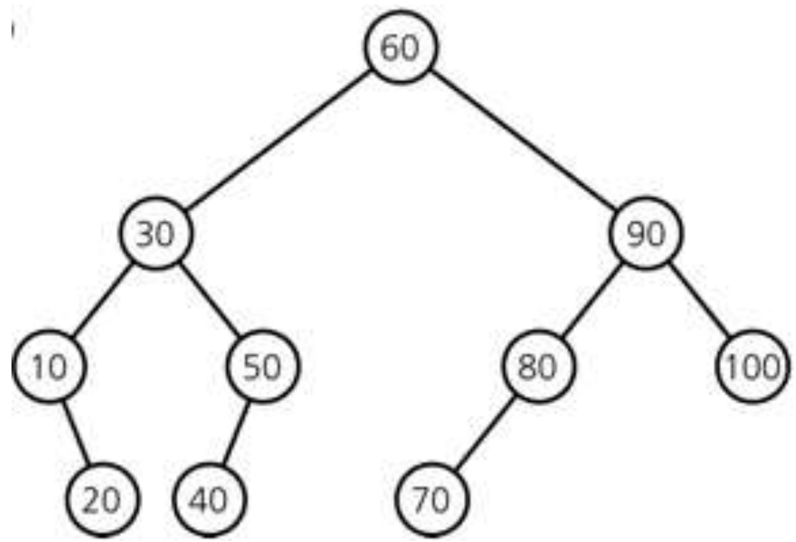


Searching a 2-3 Tree

- Searching a 2-3 tree is similar to searching a binary search tree
  - For a 3-node, compare the searched key with the two values of the 3-node and select one of its three subtrees according to these comparisons
- Searching a 2-3 tree is O(log N)
  - Searching a 2-3 tree and the shortest BST has approximately the same efficiency.
    - A binary search tree with N nodes cannot be shorter than $\lceil \log_2(N+1) \rceil$
    - A 2-3 tree with N nodes cannot be taller than $\lceil \log_2(N+1) \rceil$

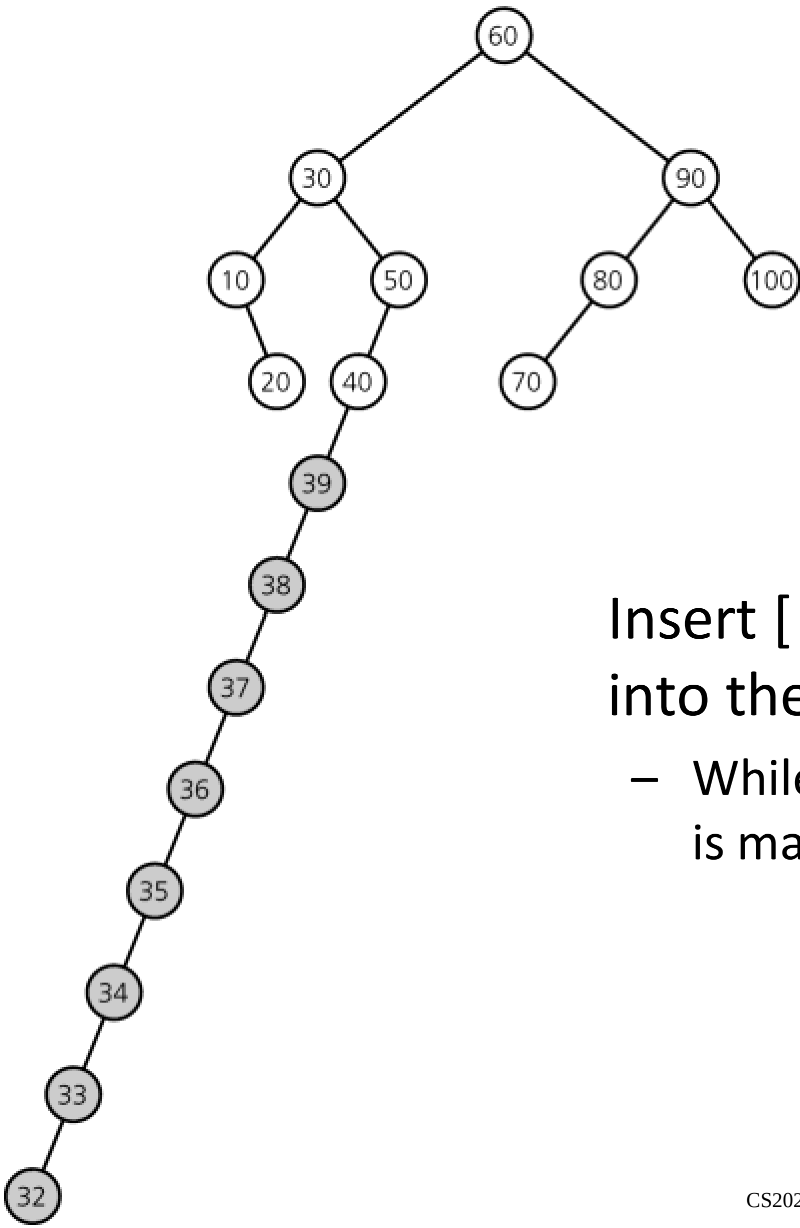


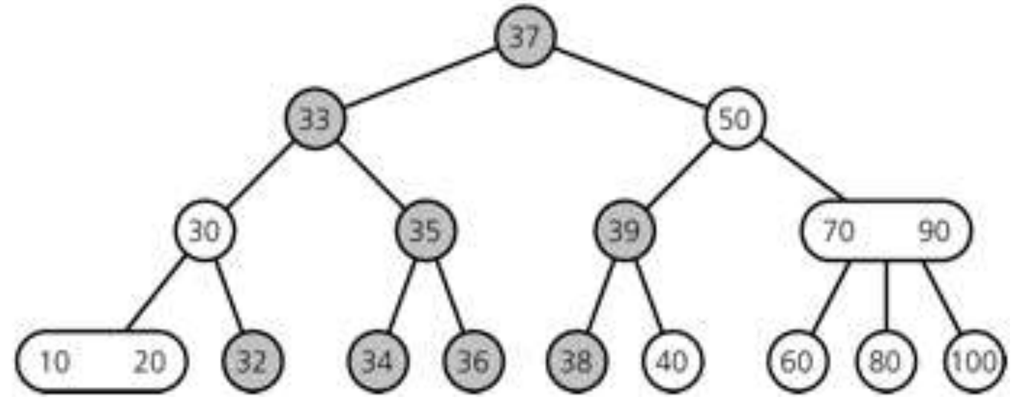
A 2-3 tree with the same elements



A balanced binary search tree

Inserting into a 2-3 Tree



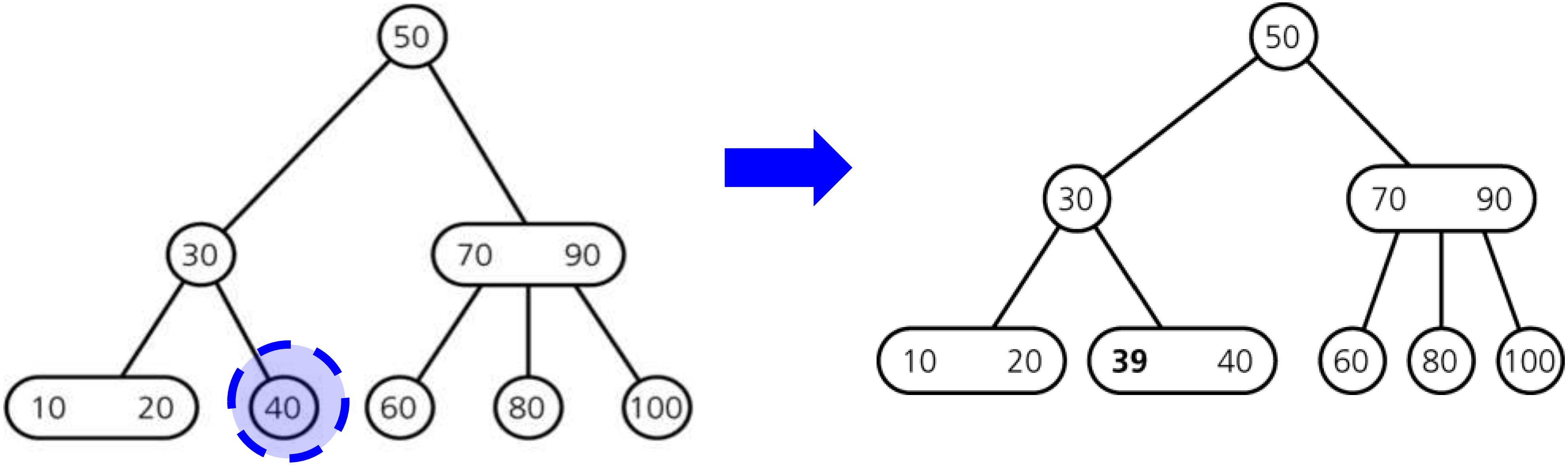


Insert [39 38 37 36 35 34 33 32]
into the trees given in the previous slide

– While we insert items into a 2-3 tree, its shape is maintained

Inserting into a 2-3 Tree -- Example

Starting from the following tree, insert [39   38   37   36   35   34   33   32]

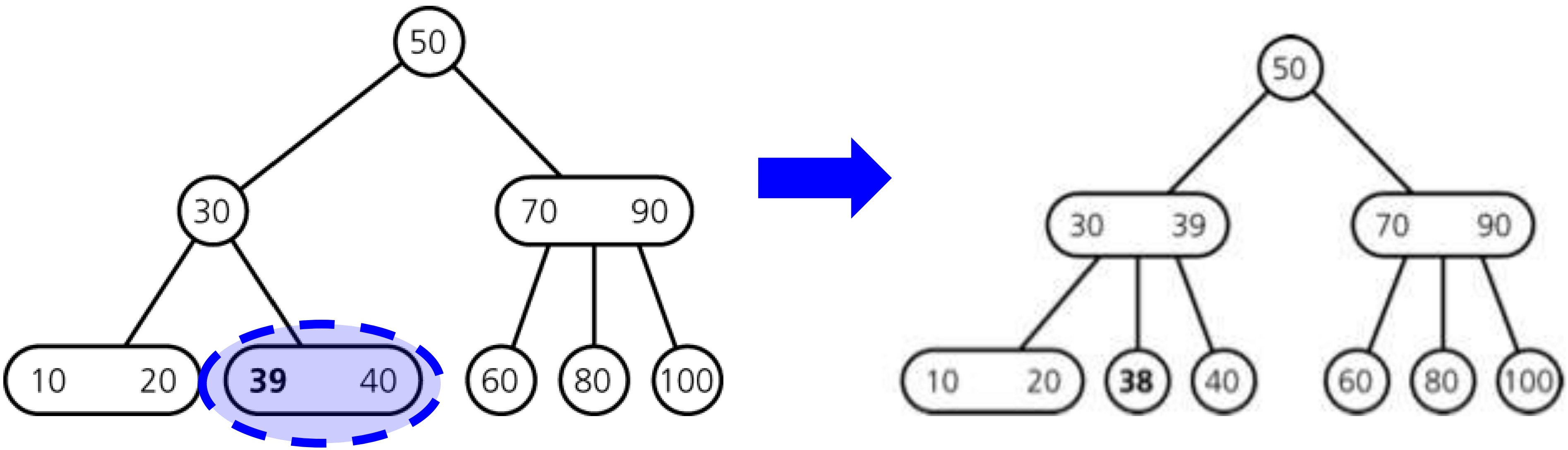


Insert 39

- Find the node into which you can put 39

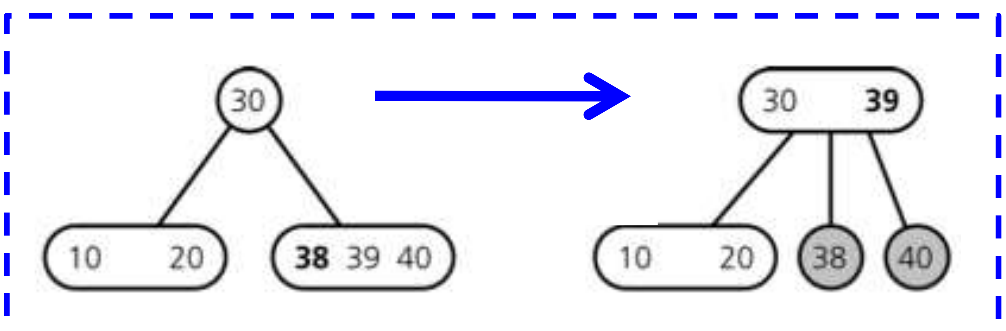
Insertion into a 2-node leaf is simple

Inserting into a 2-3 Tree -- Example



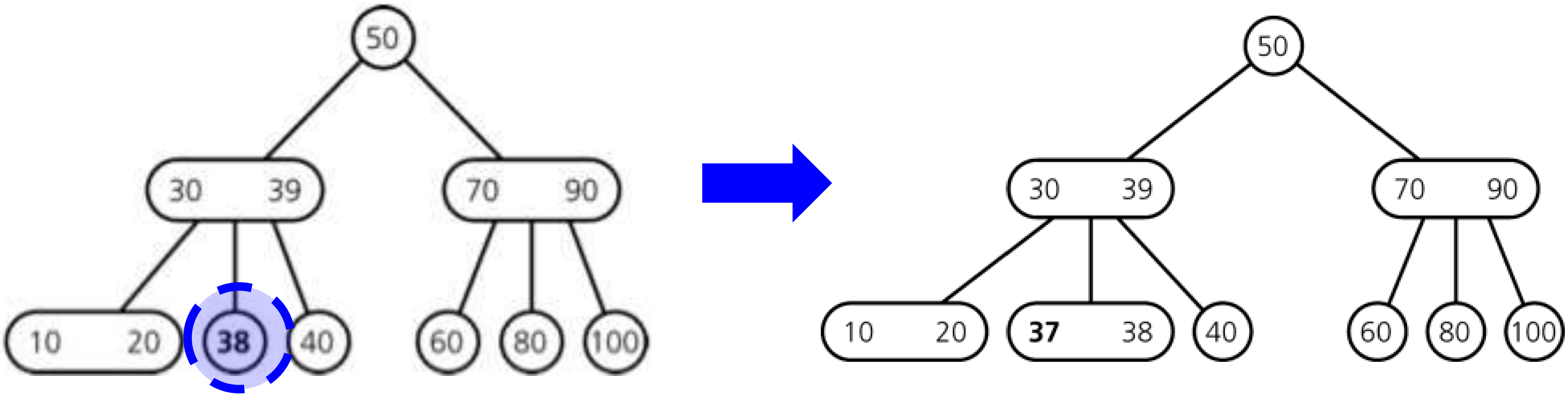
Insert 38

- Find the node into which you can put 38



Insertion into a 3-node causes it to divide

Inserting into a 2-3 Tree -- Example

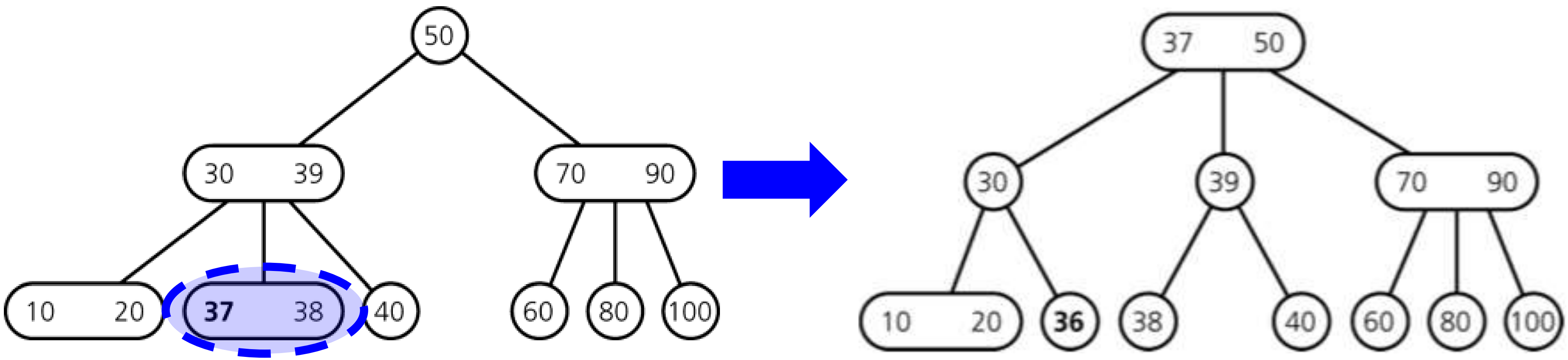


Insert 37

- Find the node into which you can put 37

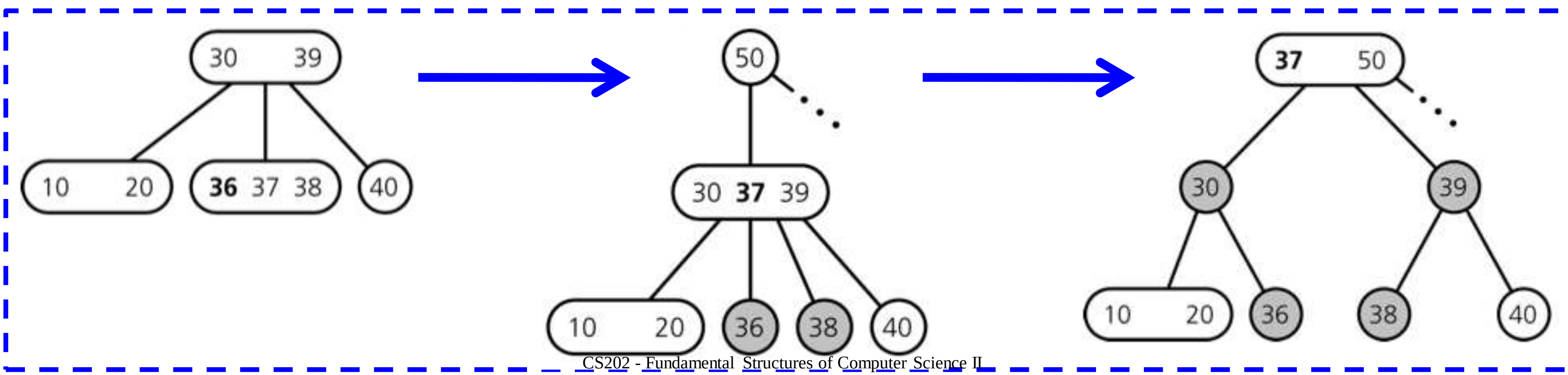
Insertion into a 2-node leaf is simple

Inserting into a 2-3 Tree -- Example

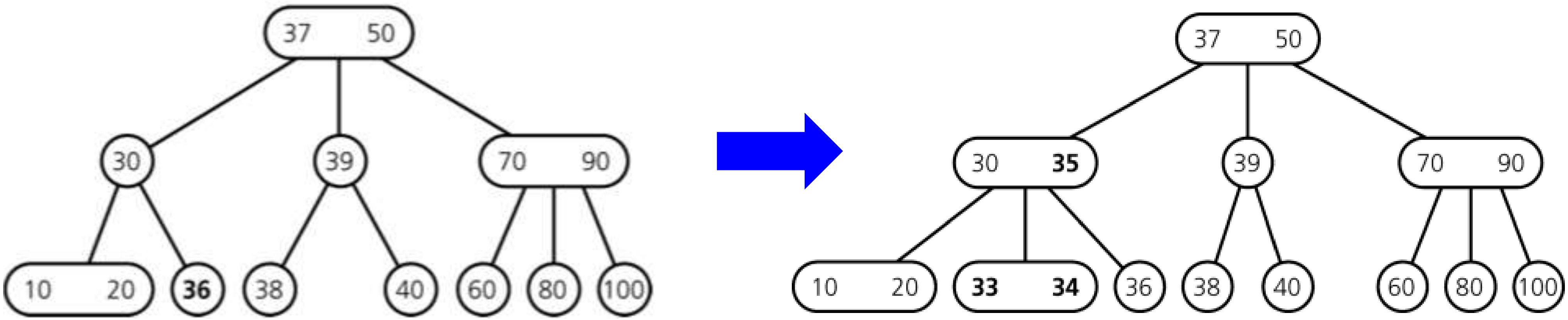


Insert 36

- Find the node into which you can put 36



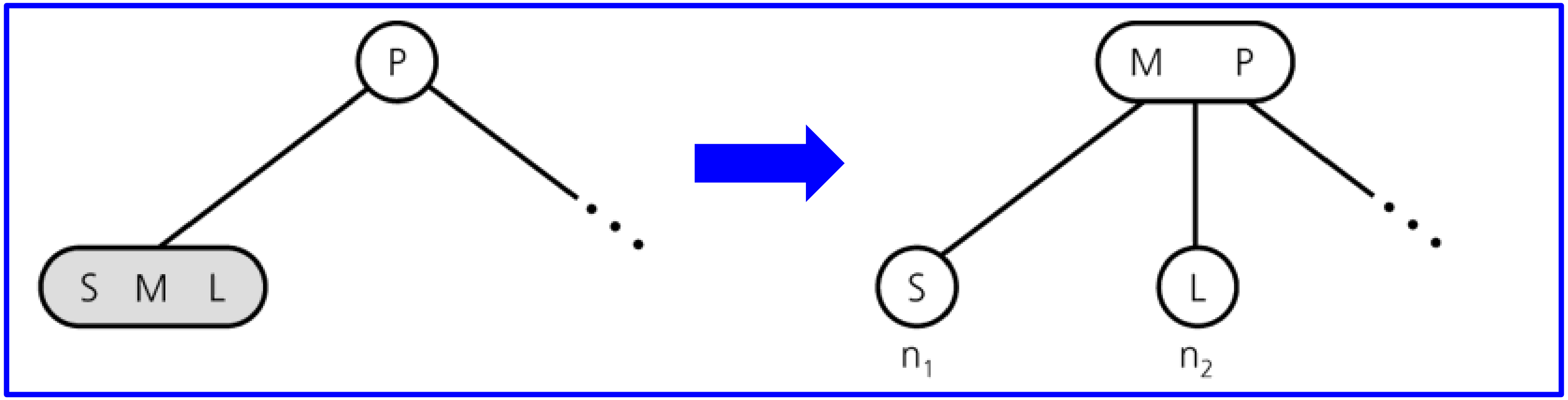
Inserting into a 2-3 Tree -- Example

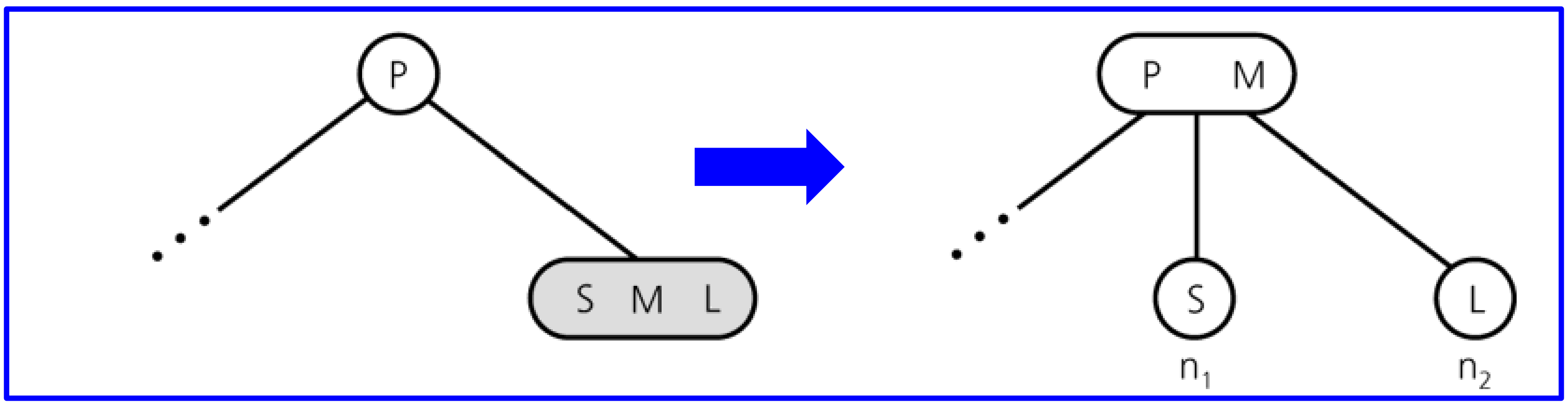


Insert 35, 34, 33

2-3 Trees -- Insertion Algorithm

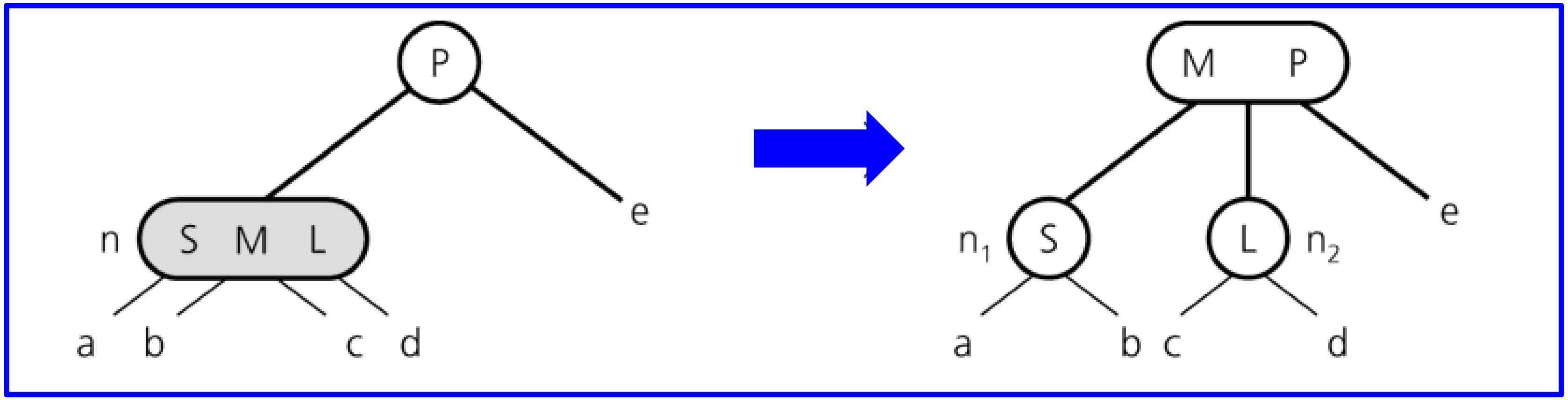
Splitting a leaf in a 2-3 tree

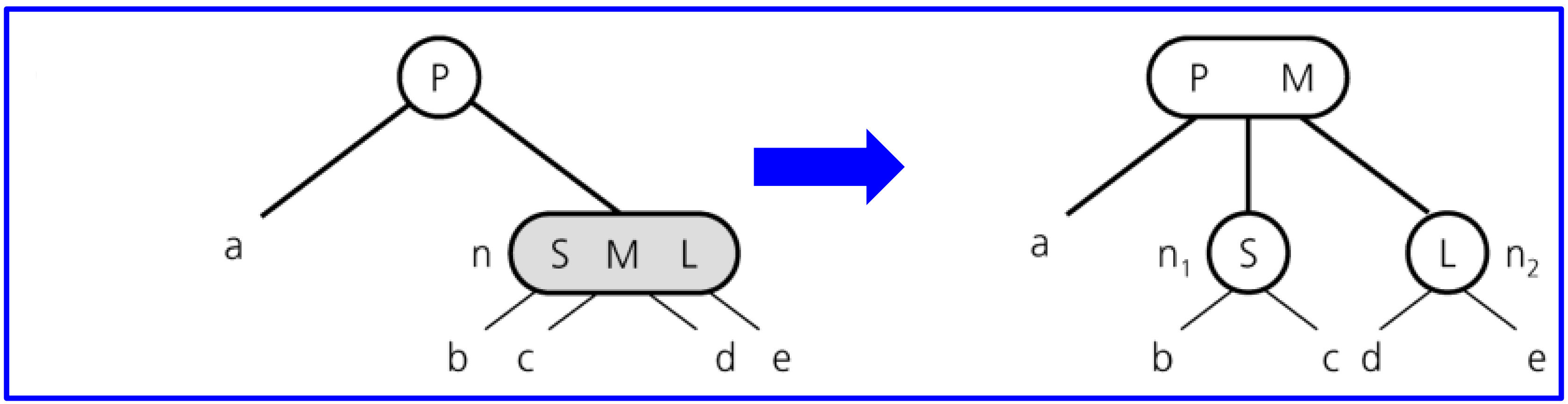




2-3 Trees -- Insertion Algorithm

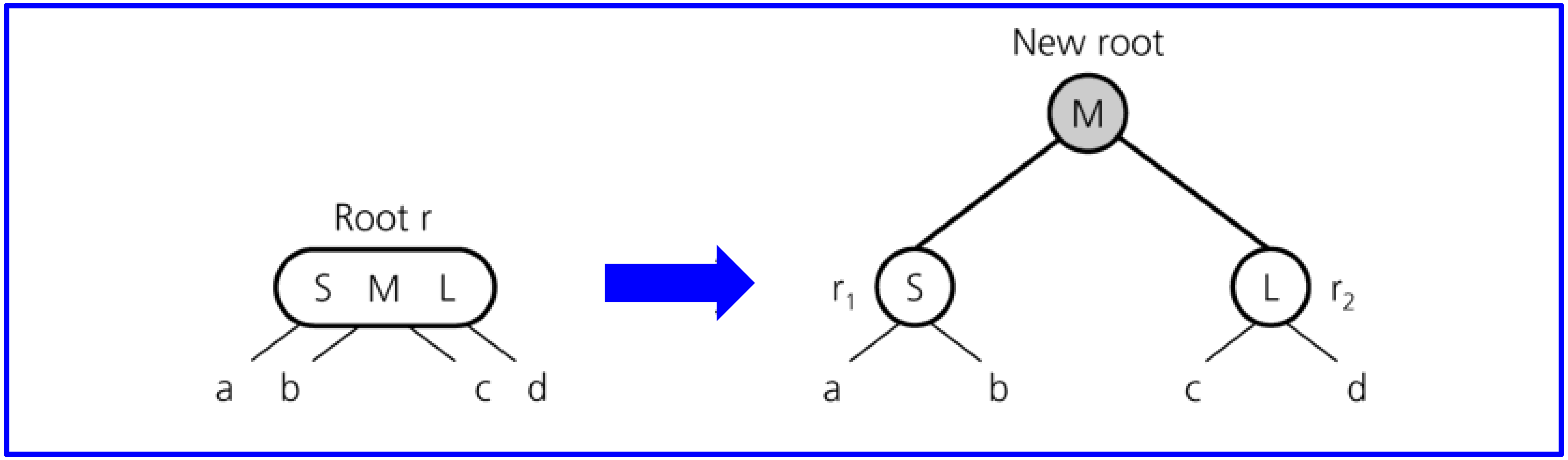
Splitting an internal node in a 2-3 tree





2-3 Trees -- Insertion Algorithm

Splitting the root of a 2-3 tree

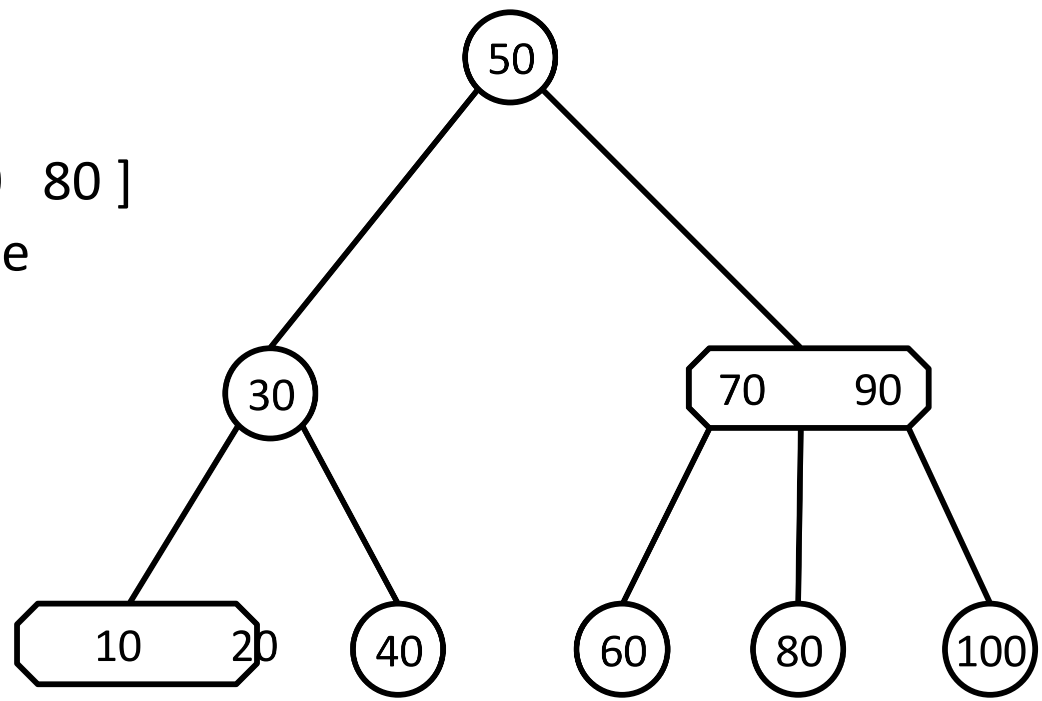


2-3 tree grows from the root side

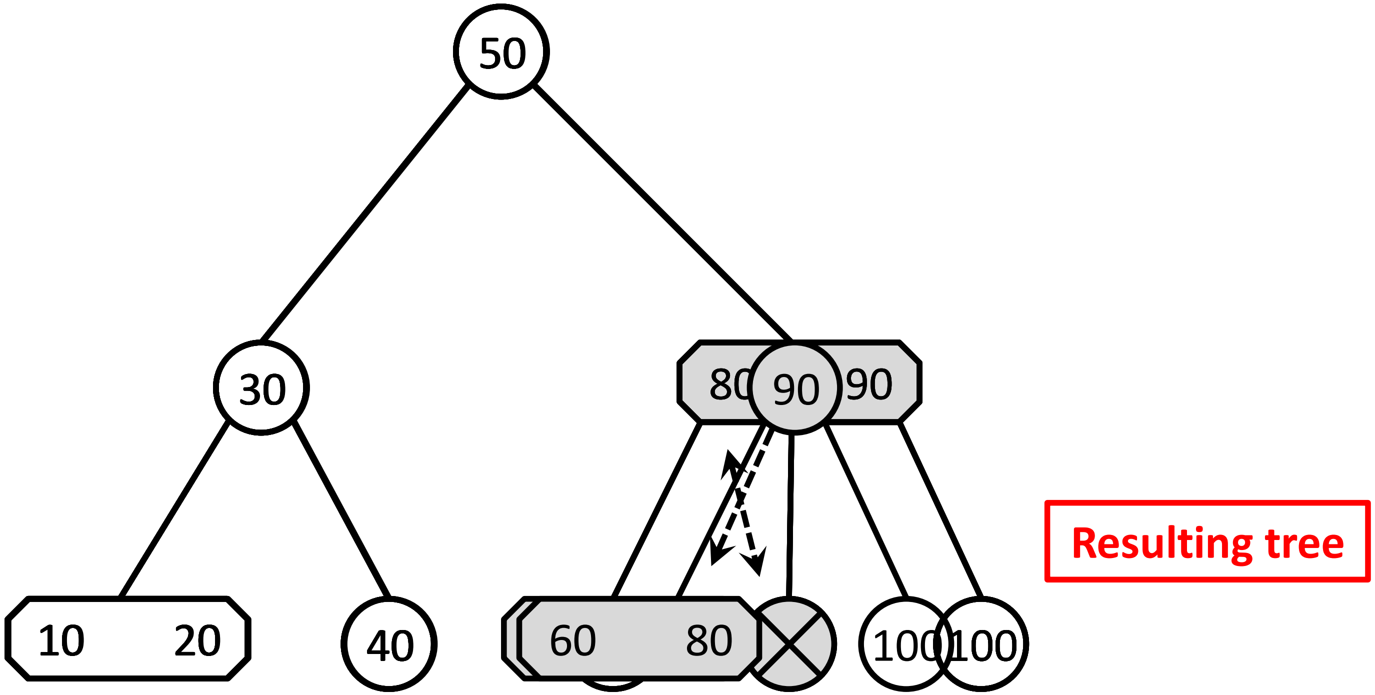
Deleting from a 2-3 tree

- Deletion strategy is the inverse of insertion strategy.
- Deletion starts like normal BST deletion (swap with inorder successor)
- Then, we merge the nodes that have become underloaded.

Delete [70   100   80]
from this 2-3 tree



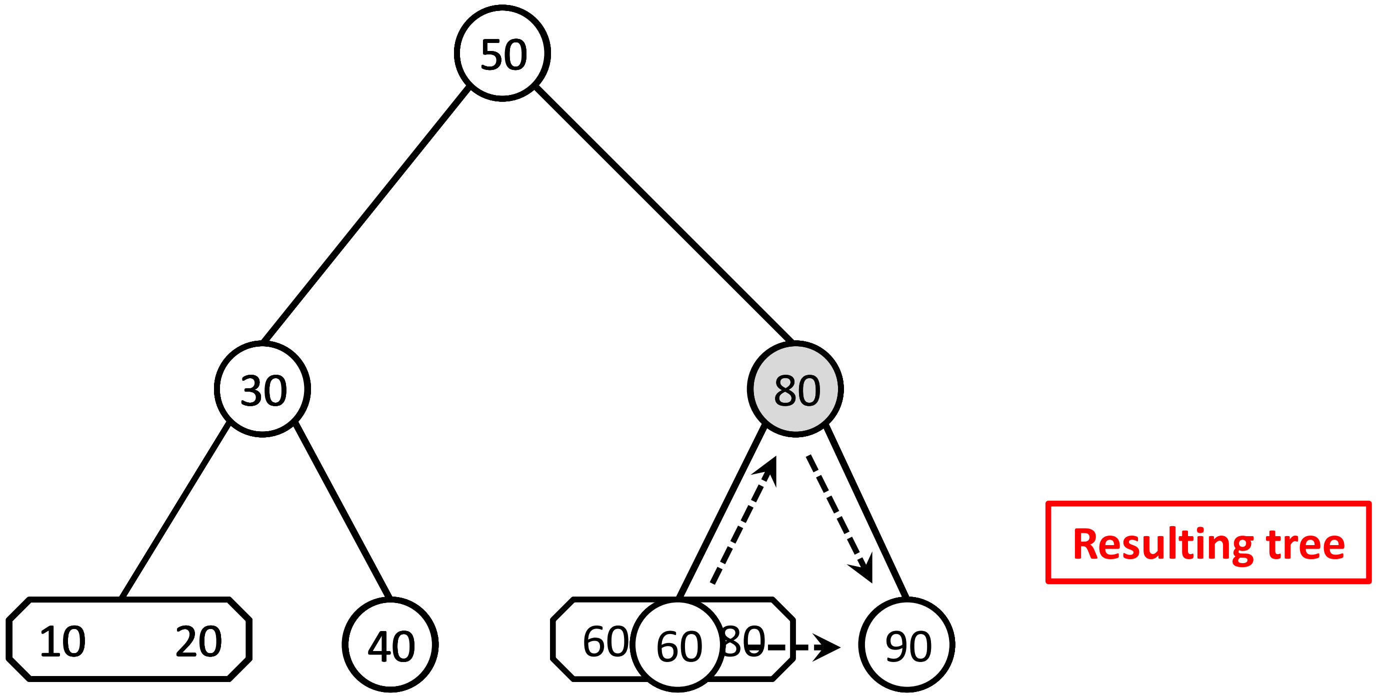
Deleting from a 2-3 Tree -- Example



Delete 70

- Swap with inorder successor
- Delete value from leaf
- Delete the empty leaf
- Shrink the parent (no more mid-pointer)

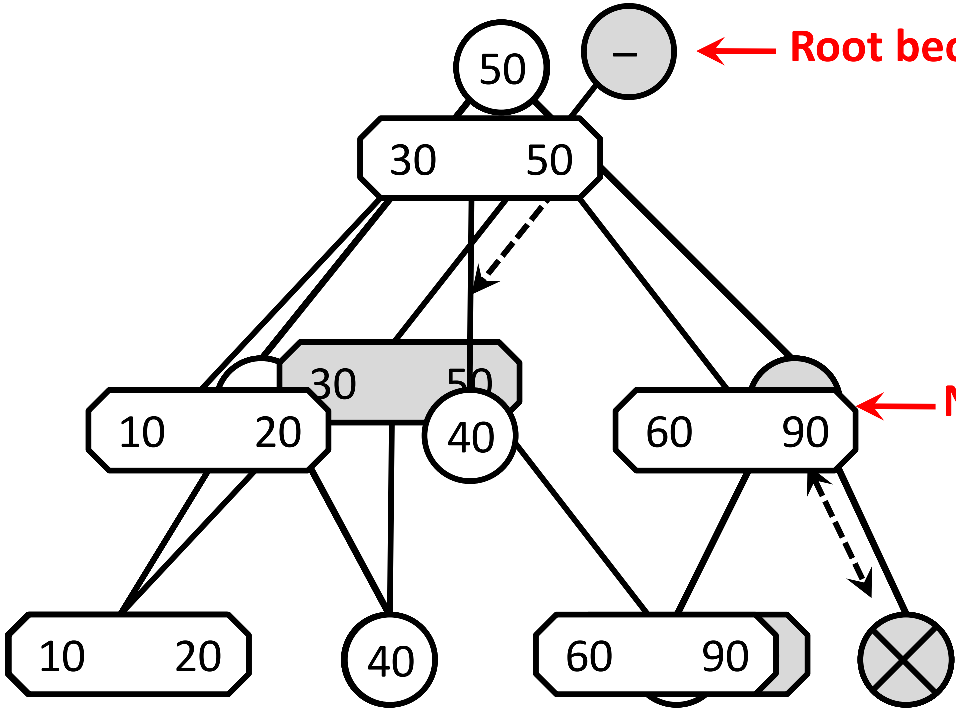
Deleting from a 2-3 Tree -- Example



Delete 100

- Delete value from leaf
- Distribute the children →Doesn’t work
- Redistribute the parent and the children

Deleting from a 2-3 Tree -- Example



Delete 80

- Swap with inorder successor
- Delete value from leaf
- Merge by moving 90 down and removing the empty leaf
- Merge by moving 50 down, adopting empty node's child and removing the empty node
- Remove empty root

all leaves should be on same level

2-3 Trees -- Deletion Algorithm

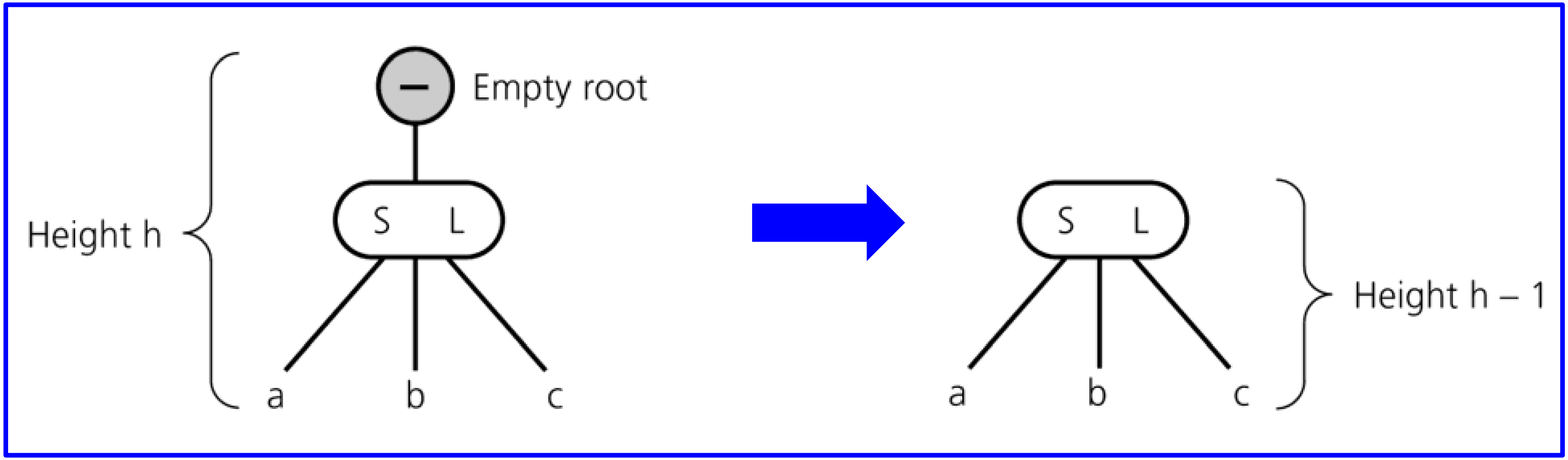
- To delete an item X from a 2-3 tree:
  - First, we locate the node n containing X.
  - If n is not a leaf, we find X's inorder successor and swap it with X.
  - After the swap, the deletion always begins at the leaf.
  - If the leaf contains another item in addition to X, we simply delete X from that leaf, and we are done.
  - If the leaf contains only X, deleting X would leave the leaf without a data item. In this case, we must perform some additional work to complete the deletion.

- Depending on the empty node and its siblings, we perform certain operations:
  - Delete empty root
  - Merge nodes
  - Redistribute values

- These operations can be repeated all the way upto the root if necessary.

2-3 Trees -- Deletion Operations

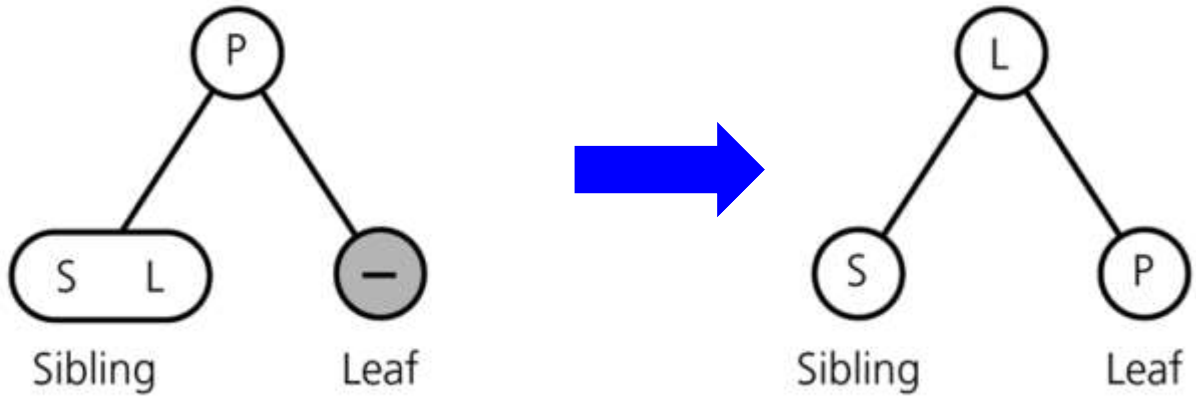
Deleting the root



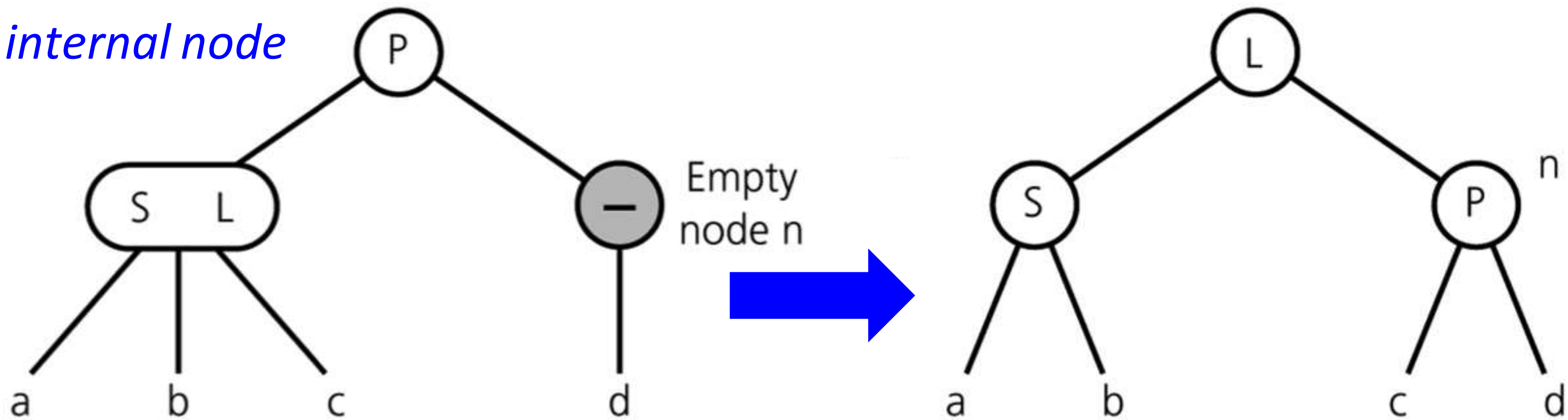
2-3 Trees -- Deletion Operations

Redistributing values (and children)

For a leaf



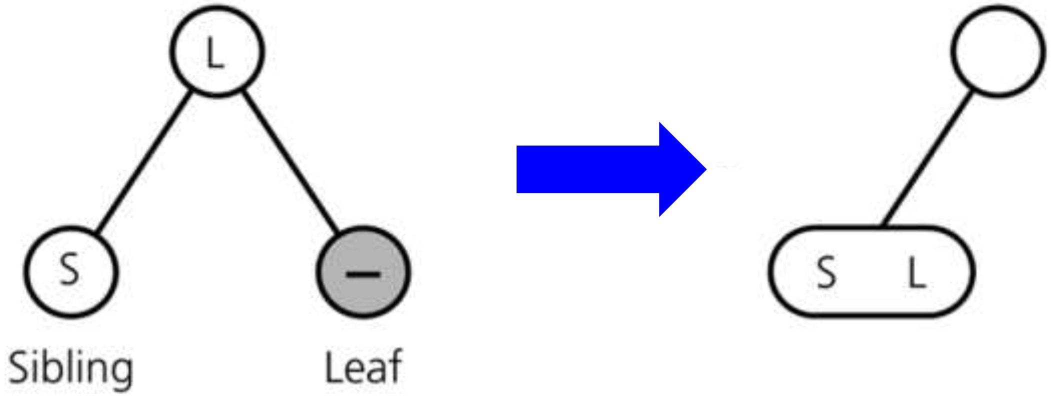
For an internal node



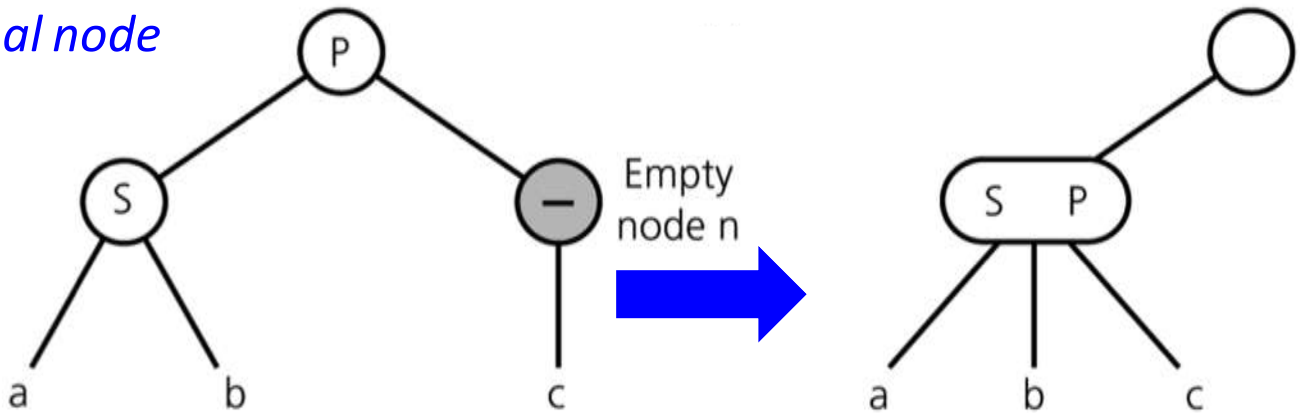
2-3 Trees -- Deletion Operations

Merging

For a leaf



For an internal node



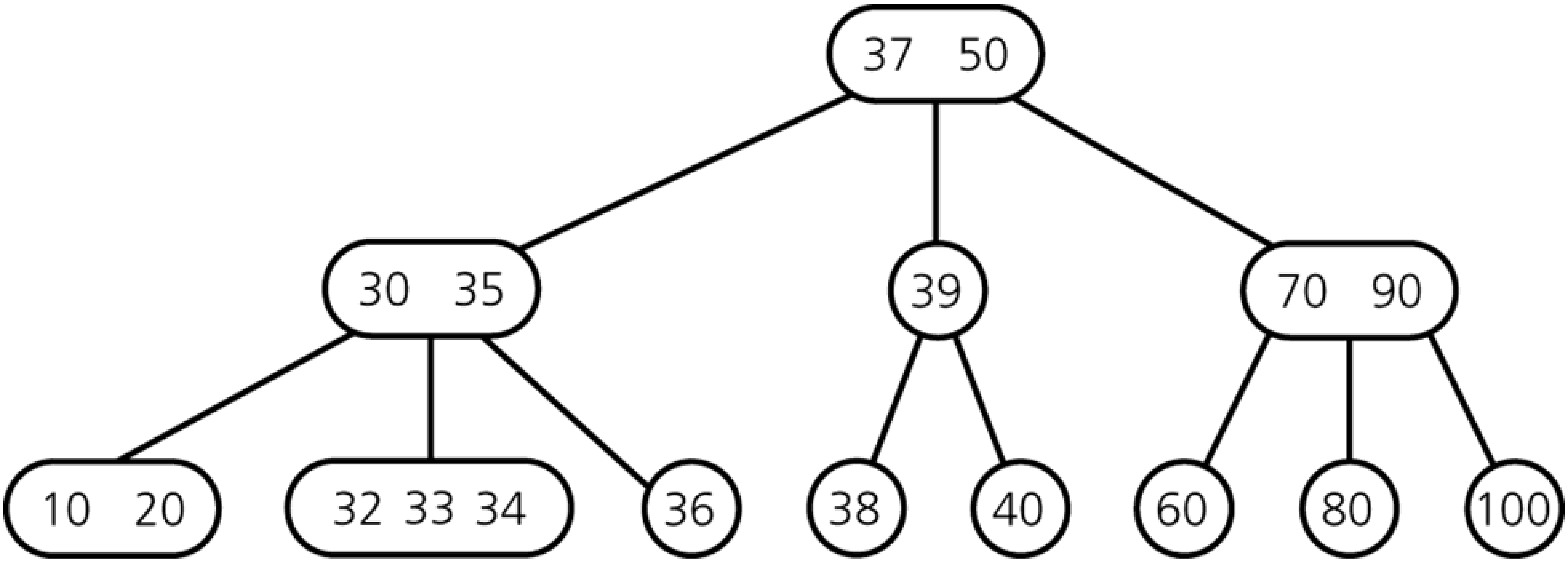
2-3 Trees -- Analysis

- We can use a 2-3 tree in the implementation of tables.
- A 2-3 tree has the advantage of always being balanced.
- Thus, insertion and deletion operations are O(log N)
- Retrieval based on key is also guaranteed to O(log N)

2-3-4 Trees

- A 2-3-4 tree is like a 2-3 tree, but it allows 4-nodes, which are nodes that have four children and three data items.
- There is a close relation between 2-3-4 trees and red-black trees.
  - We will look at those a bit later
- 2-3-4 trees are also known as 2-4 trees in other books.
  - A specialization of M-way tree (M=4)
  - Sometimes also called 4th order B-trees
  - Variants of B-trees are very useful in databases and file systems
    - MySQL, Oracle, MS SQL all use B+ trees for indexing
    - Many file systems (NTFS, Ext2FS etc.) use B+ trees for indexing metadata (file size, date etc.)
- Although a 2-3-4 tree has more efficient insertion and deletion operations than a 2-3 tree, a 2-3-4 tree has greater storage requirements.

2-3-4 Trees -- Example

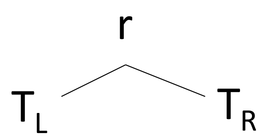


2-3-4 Trees

T is a 2-3-4 tree of height h if

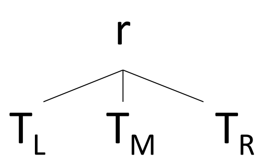
1. T is empty (a 2-3-4 tree of height 0), or

1. T is of the form



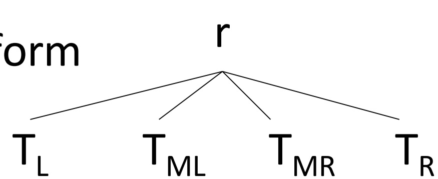
where r is a node containing one data item and T_L and T_R are both 2-3-4 trees, each of height h-1, or

3. T is of the form

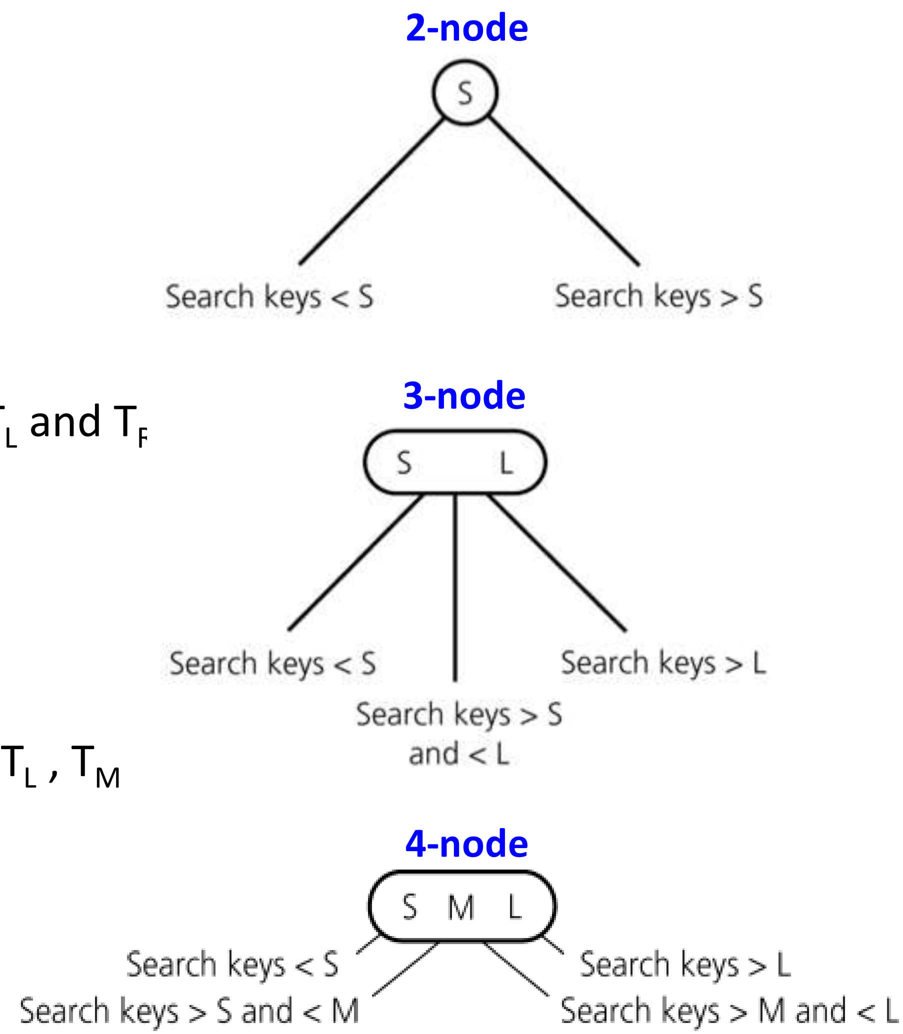


where r is a node containing two data items and T_L , T_M and T_R are 2-3-4 trees, each of height h-1, or

4. T is of the form



where r is a node containing three data items and T_L , T_{ML} , T_{MR} , and T_R are 2-3-4 trees, each of height h-1.



C++ Class for a 2-3-4 Tree Node

```
classTreeNode {
private:

        TreeItemTypesmallItem, middleItem, largeItem;

        TreeNode *leftChildPtr, *lMidChildPtr;
        TreeNode *rMidChildPtr, *rightChildPtr;

friendclassTwoThreeFourTree;
};
```

- When a node is a 3-node (contains only two items)
  - Place the items in `smallItem` and `middleItem`
  - Use `leftChildPtr, lMidChildPtr, rMidChildPtr` to point to the node's children
  - Place `NULL` in `rightChildPtr`

- When a node is a 2-node (contains only one item)
  - Place the item in `smallItem`
  - Use `leftChildPtr, lMidChildPtr` to point to the node's children
  - Place `NULL` in `rMidChildPtr` and `rightChildPtr`

2-3-4 Trees -- Operations

- Searching and traversal algorithms for a 2-3-4 tree are similar to the 2-3 algorithms.
- For a 2-3-4 tree, insertion and deletion algorithms that are used for 2-3 trees, can similarly be used.
- But, we can also use a slightly different insertion and deletion algorithms for 2-3-4 trees to gain some efficiency.

Inserting into a 2-3-4 Tree

- Splits 4-nodes by moving one of its items up to its parent node.

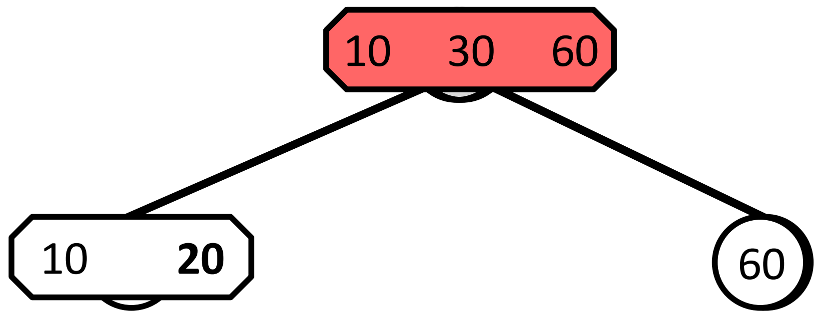
- For a 2-3 tree, the insertion algorithm traces a path from the root to a leaf and then backs up from the leaf as it splits nodes.

- *To avoid this return path after reaching a leaf*, the insertion algorithm for a 2-3-4 tree splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf.
  - As a result, when a 4-node is split and an item is moved up to node's parent, the parent cannot possibly be a 4-node and so can accommodate another item.

Insert[20 50 40 70 80 15 90 100] to this 2-3-4 tree

10 30 60

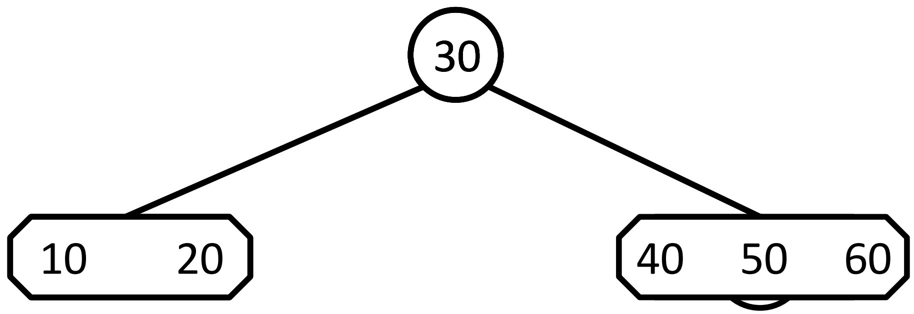
Inserting into a 2-3-4 Tree -- Example



Insert 20

- Root is a 4-node →**Split 4-nodes as they are encountered**
- So, we split it before insertion
- And, then add 20

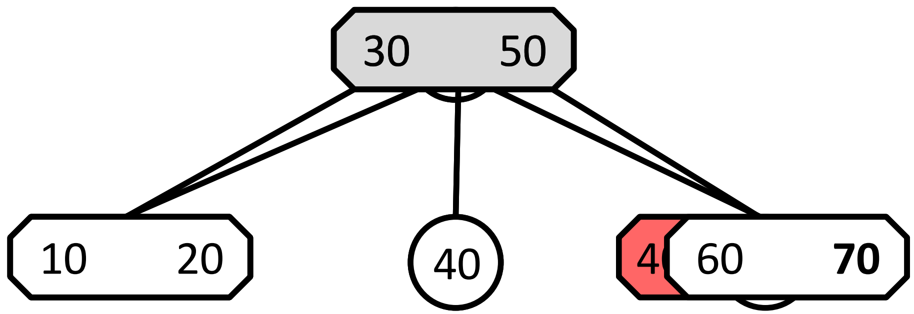
Inserting into a 2-3-4 Tree -- Example



Insert 50 and 40

- No 4-nodes have been encountered during their insertion → **No split operation**

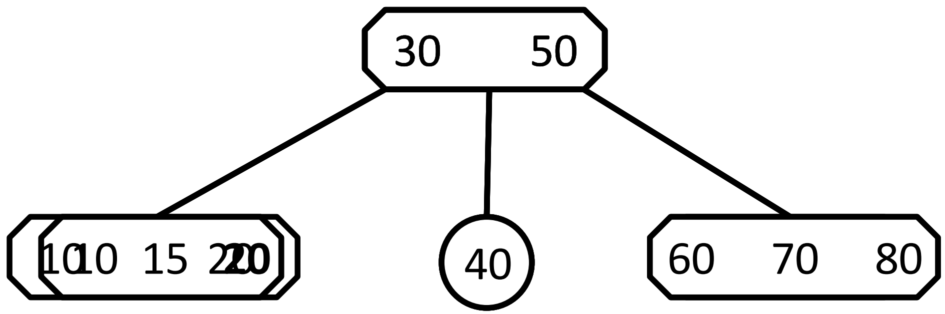
Inserting into a 2-3-4 Tree -- Example



Insert 70

- A 4-node is encountered
- So, we split it before insertion
- And, then add 70

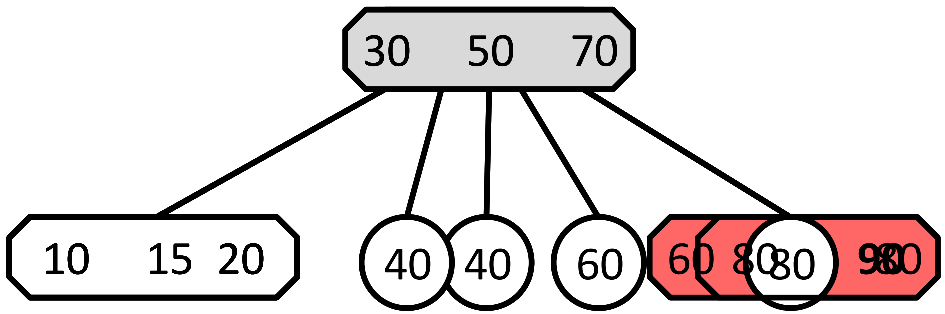
Inserting into a 2-3-4 Tree -- Example



Insert 80 and 15

- No 4-nodes have been encountered → **No split operation** during their insertion

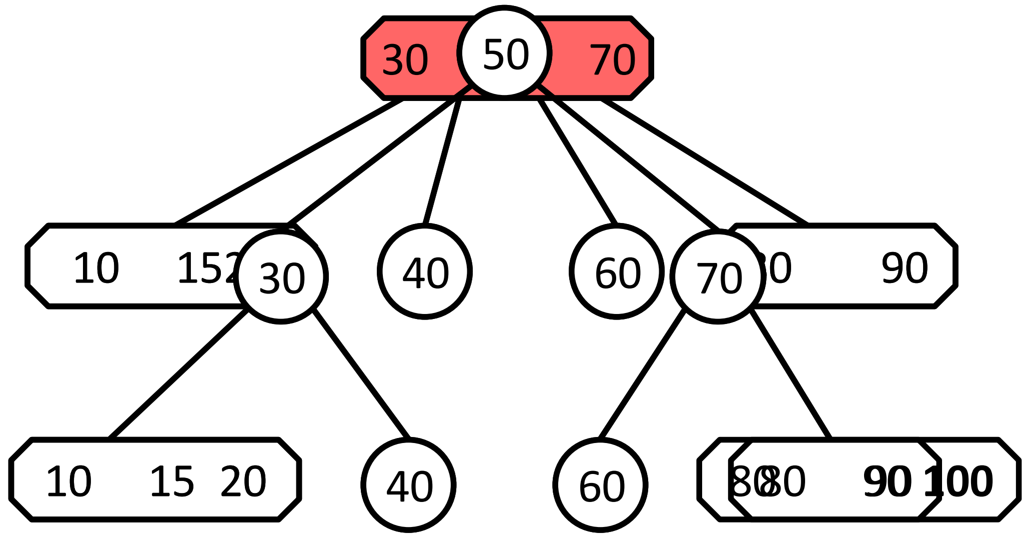
Inserting into a 2-3-4 Tree -- Example



Insert 90

- A 4-node is encountered
- So, we split it before insertion
- And, then add 90

Inserting into a 2-3-4 Tree -- Example



Insert 100

- A 4-node is encountered root
- So, we split it before insertion
- And, then add 100

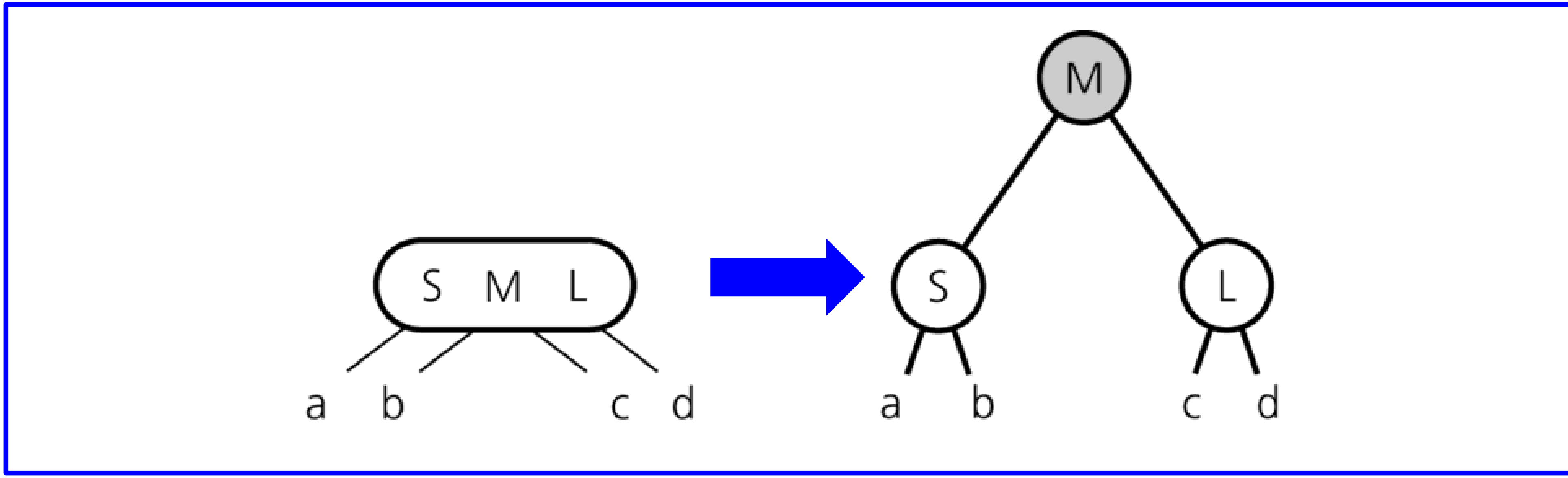
Splitting 4-nodes during insertion

- We split each 4-node as soon as we encounter it during our search from the root to a leaf that will accommodate the new item to be inserted.

- The 4-node which will be split can:
  - be the root, or
  - have a 2-node parent, or
  - have a 3-node parent.

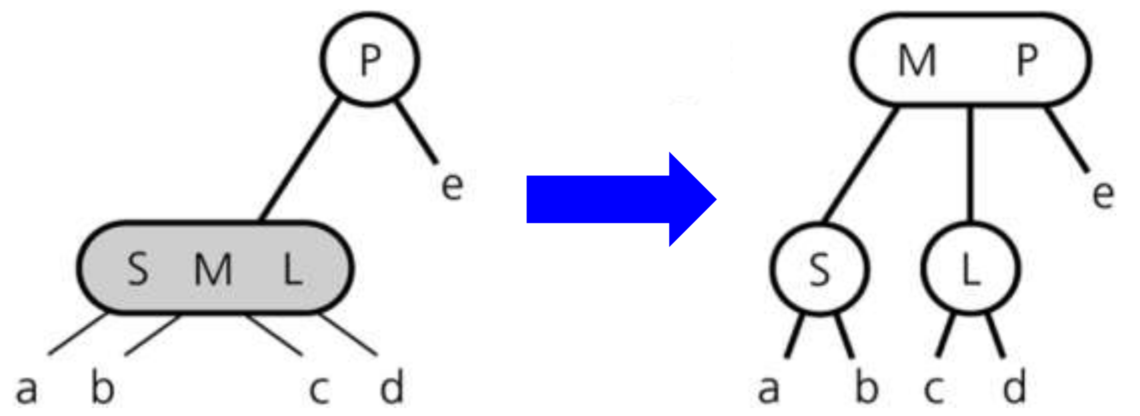
Splitting 4-nodes during insertion

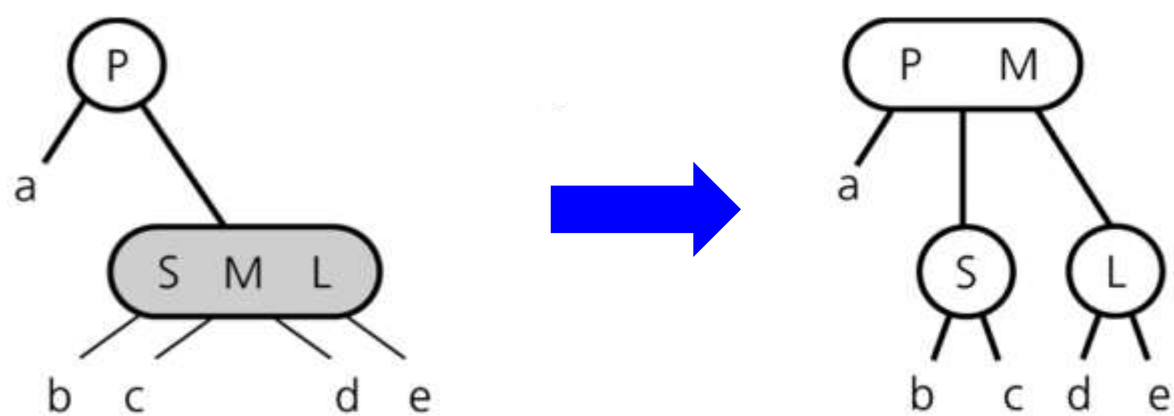
Splitting a 4-node root



Splitting 4-nodes during insertion

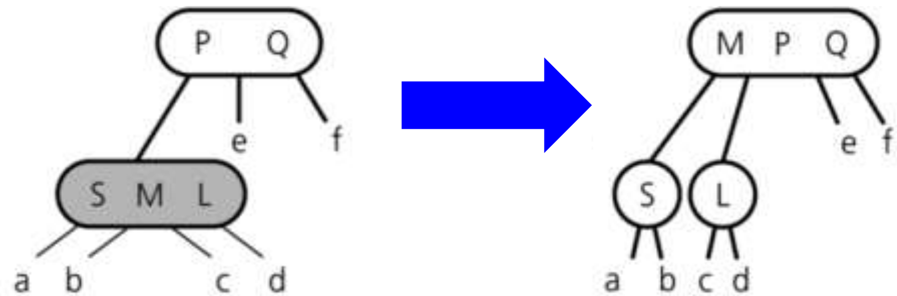
Splitting a 4-node whose parent is a 2-node

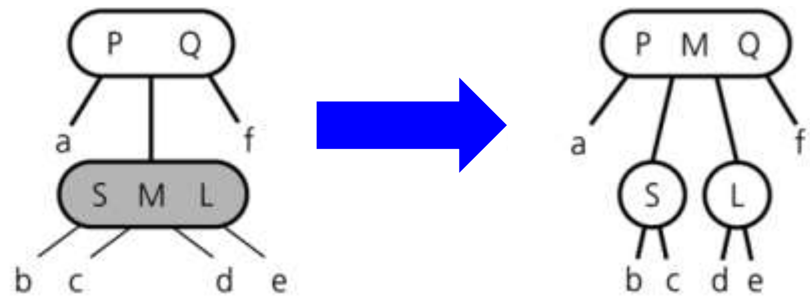


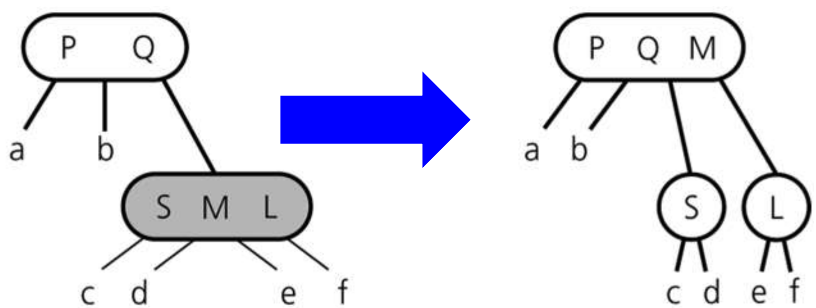


Splitting 4-nodes during insertion

Splitting a 4-node whose parent is a 3-node







Deleting from a 2-3-4 tree

- For a 2-3 tree, the deletion algorithm traces a path from the root to a leaf and then backs up from the leaf, fixing empty nodes on the path back up to root.

- *To avoid this return path after reaching a leaf*, the deletion algorithm for a 2-3-4 tree transforms each 2-node into either 3-node or 4-node as soon as it encounters them on the way down the tree from the root to a leaf.
  - If an adjacent sibling is a 3-node or 4-node, transfer an item from that sibling to our 2-node.
  - If adjacent sibling is a 2-node, merge them.

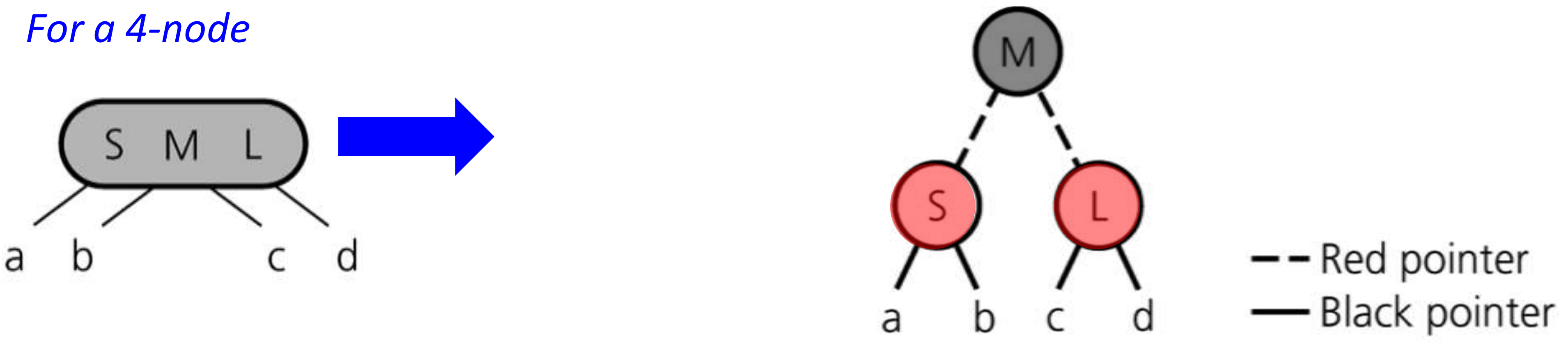
Red-Black Trees

- In general, a 2-3-4 tree requires more storage than a binary search tree.
- A special binary search tree, the **red-black-tree**, can be used to represent a 2-3-4 tree, so that we can retain advantages of a 2-3-4 tree without a storage overhead.
  - 3-node and 4-nodes in a 2-3-4 tree are represented by a binary tree.
  - To distinguish the original 2-nodes from 2-nodes that are generated from 3-nodes and 4-nodes, we use red and black pointers.
  - All original pointers in a 2-3-4 tree are black pointers, red pointers are used for child pointers to link 2-nodes that result from the split of 3-nodes and 4-nodes.

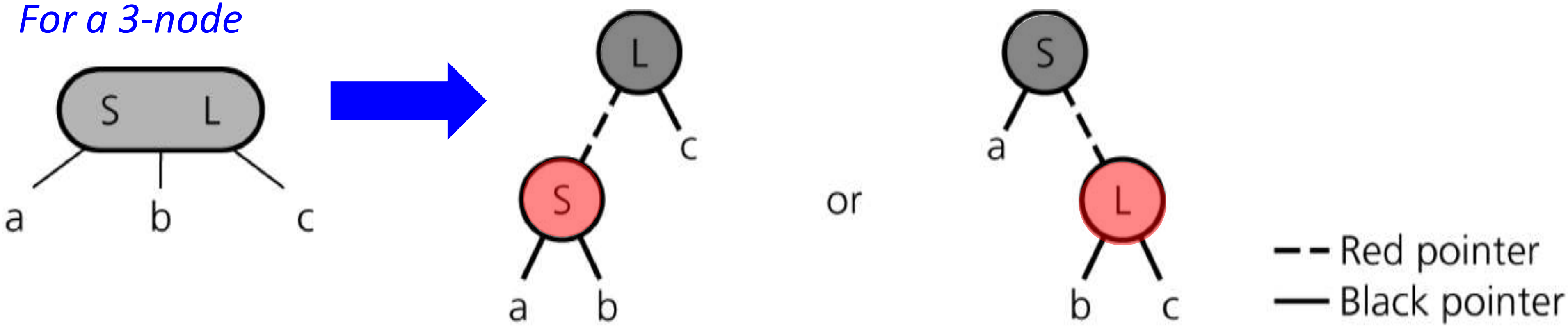
Red-Black Trees

Red-black tree representation

For a 4-node



For a 3-node

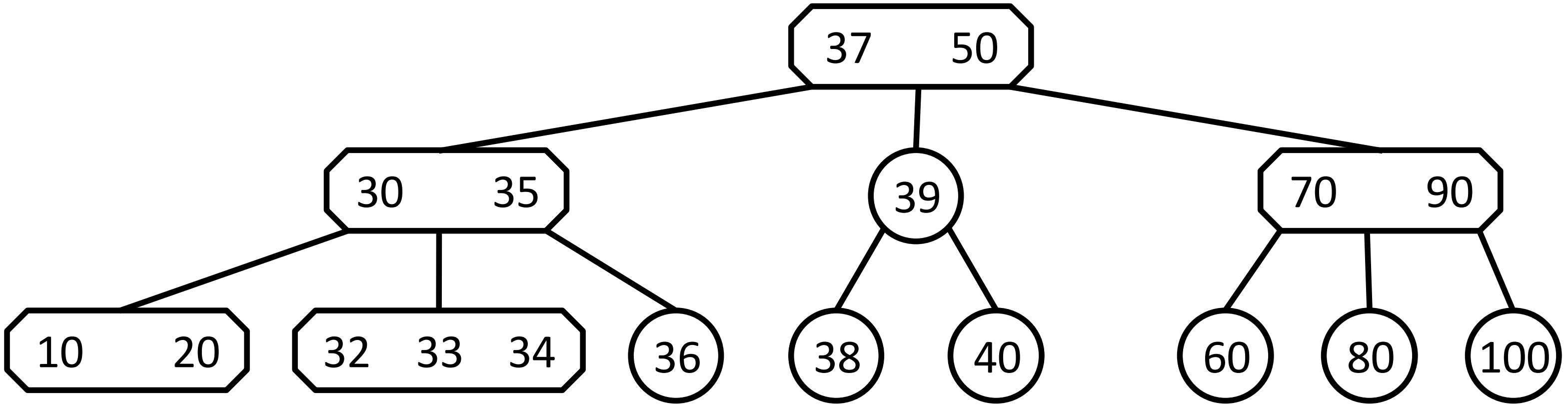


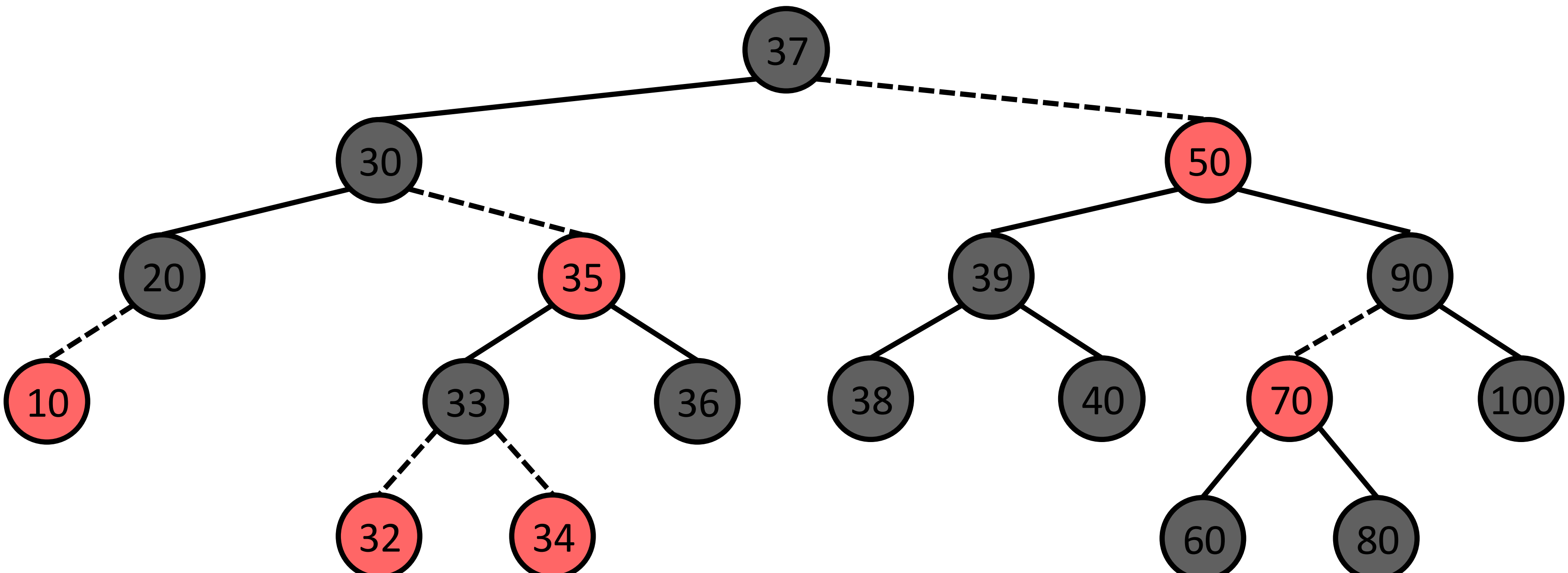
Red-Black Trees -- Properties

- Root is always a ***black node.***
- The children of a ***red node (pointed by a red pointer)*** are always ***black nodes (pointed by a black pointer)***
- All external nodes (leaves and nodes with a single child) should have **the same number of black pointers** on the path from the root to that external node.

perfect balance

A 2-3-4 Tree and Its Corresponding Red-Black Tree





C++ Class for a Red-Black Tree Node

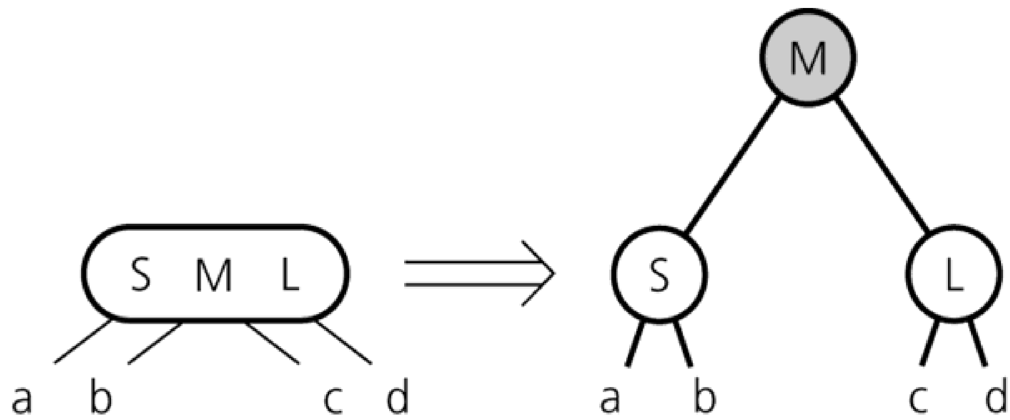
```
enum Color {RED, BLACK};

classTreeNode {
private:
      TreeItemType Item;
      TreeNode *leftChildPtr, *rightChildPtr;
      Color     leftColor, rightColor;
```

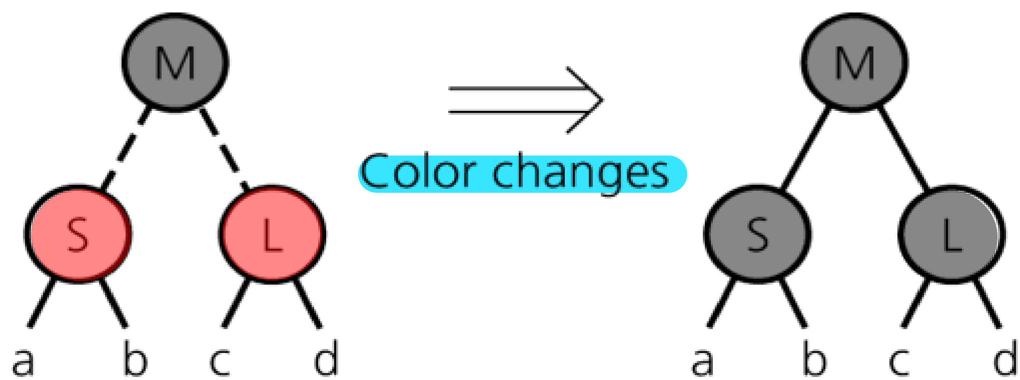
only one color variable for node, no red or black pointers is ok too

```
friendclassRedBlackTree;
};
```

Splitting in a Red-Black Tree Representation

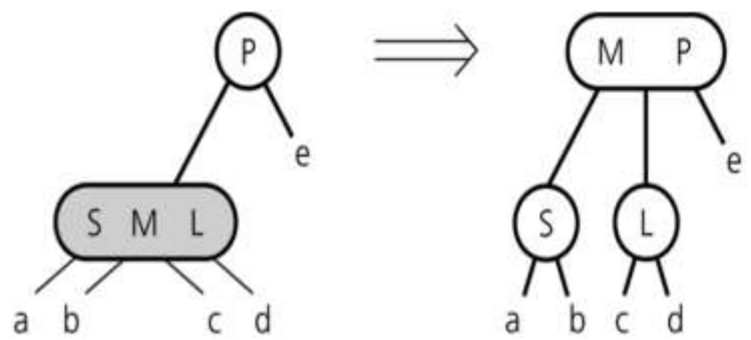
For a 4-node that is the root

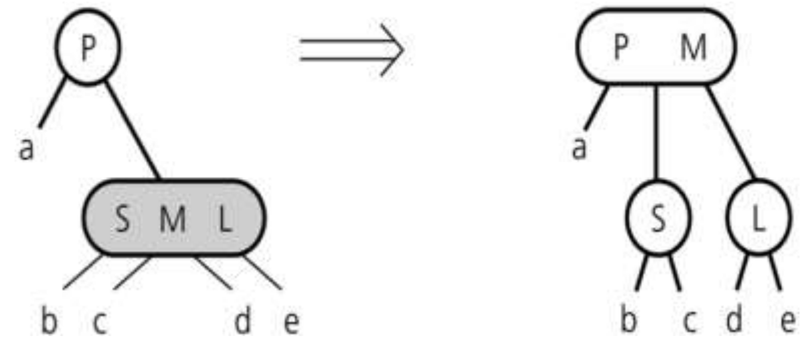


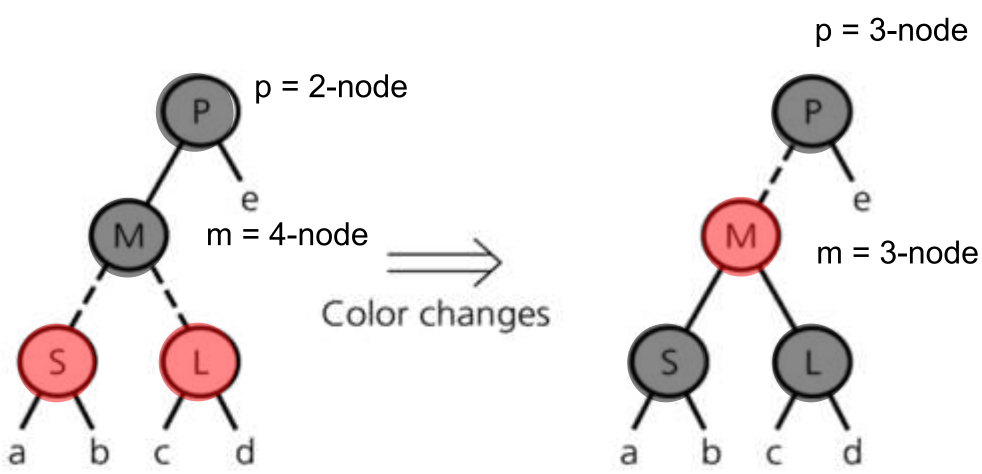


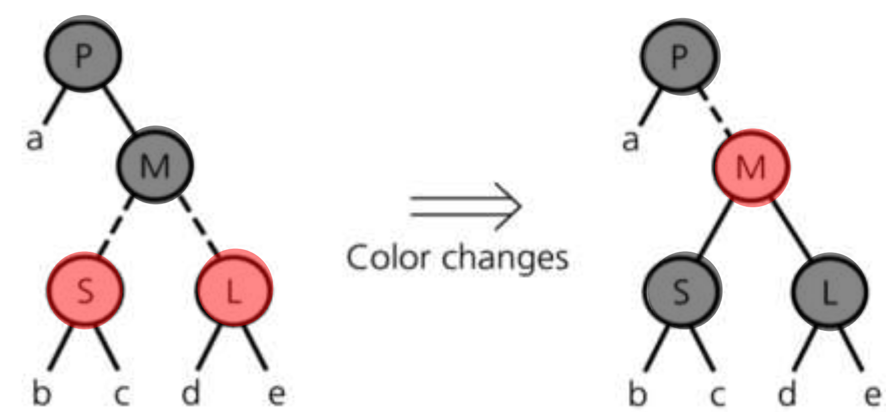
Splitting in a Red-Black Tree Representation

For a 4-node whose parent is a 2-node



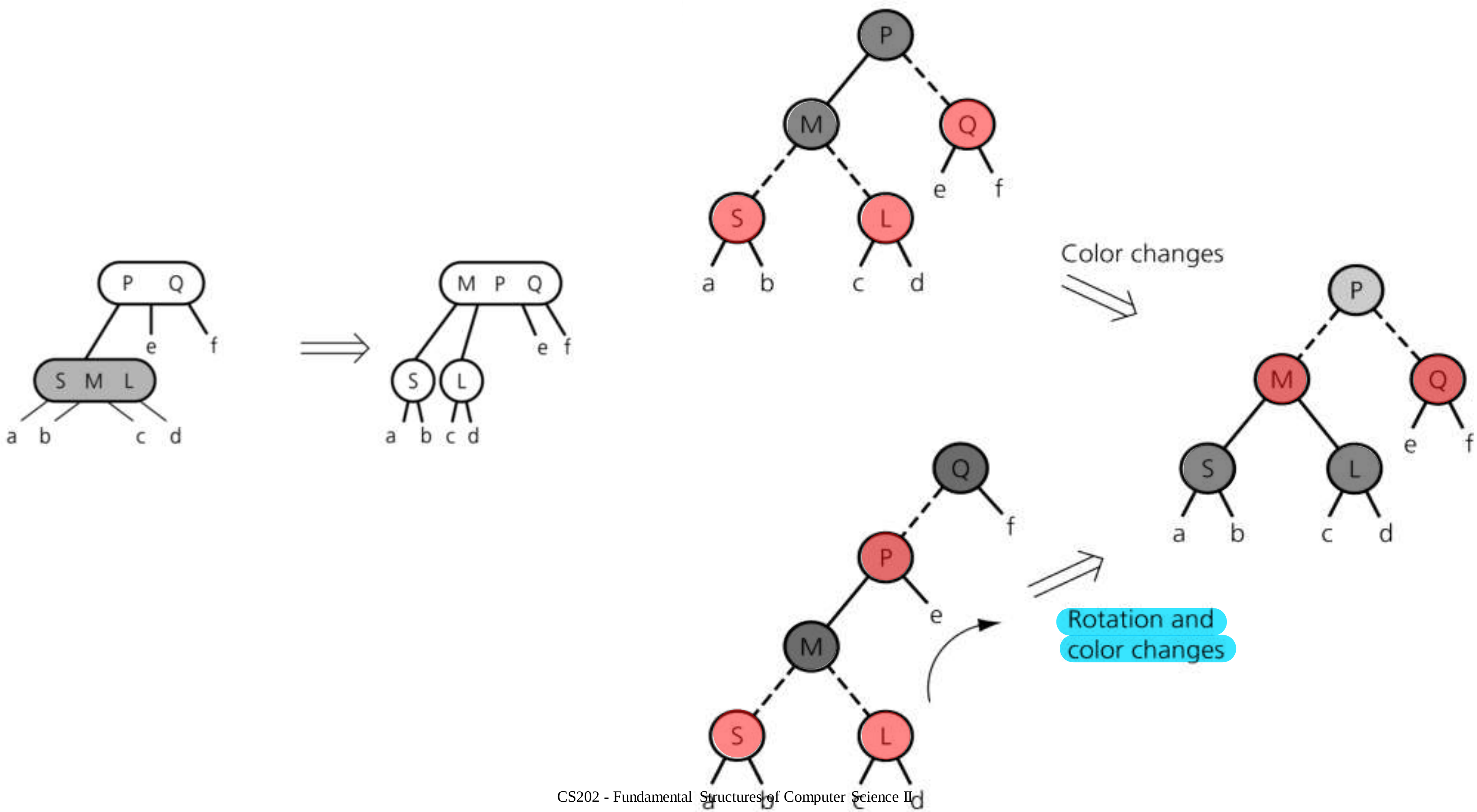






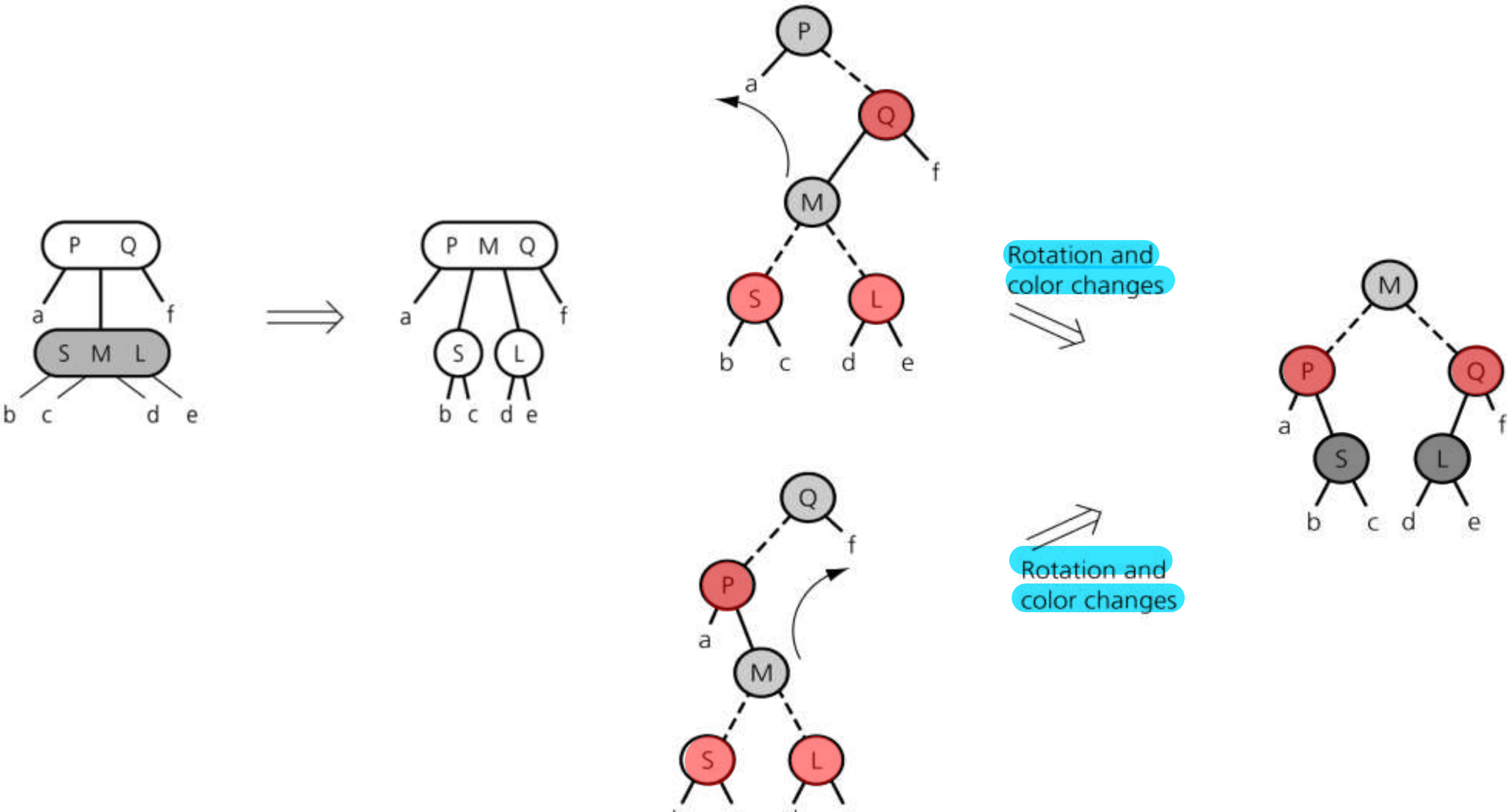
Splitting in a Red-Black Tree Representation

For a 4-node whose parent is a 3-node



Splitting in a Red-Black Tree Representation

For a 4-node whose parent is a 3-node



Splitting in a Red-Black Tree Representation

For a 4-node whose parent is a 3-node