Balanced Search Tree

Balanced + ***Binary Search Tree***

Outline

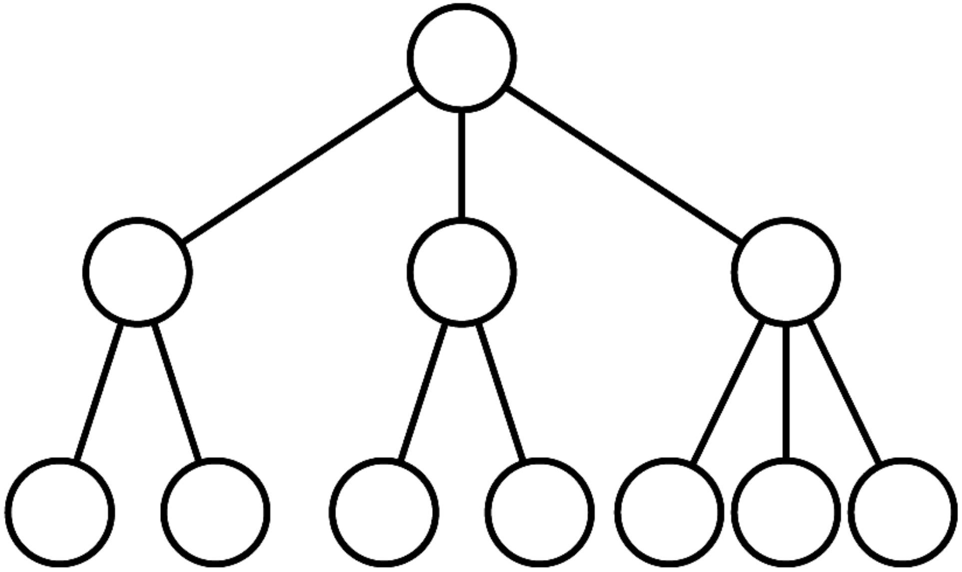
Data Structures

- 2-3 tree
- 2-3-4 tree
- AVL tree
  - [Adelson-Velskii & Landis, 1962]
- Red-black tree
  - [Rudolf Bayer, 1972]… B-tree

2-3 Tree

Data Structures

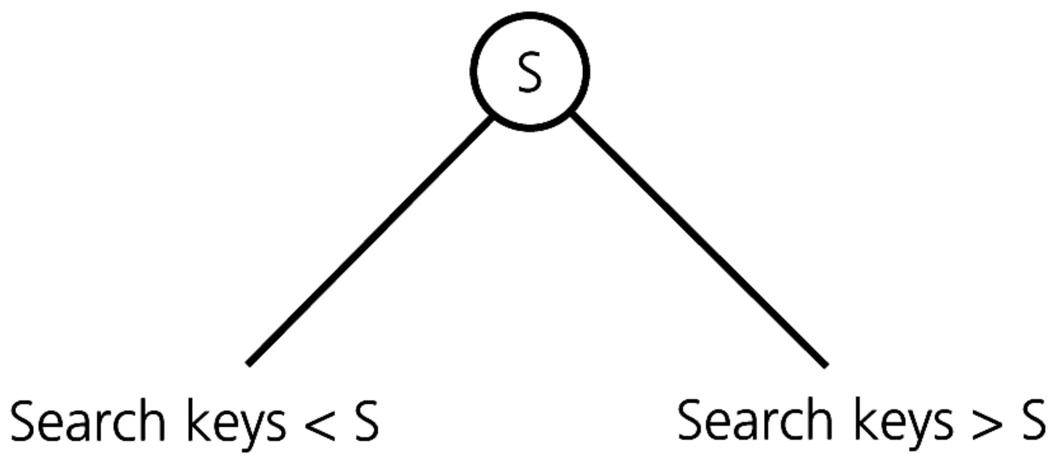
- **Have 2-nodes and 3-nodes**
  - A 2-node has one data item and two children
  - A 3-node has two data items and three children



Placing Data Items in a 2-3 Tree

- **A 2-node contains a single data item whose search key *S* satisfies the following:**
  - *S* > the left child's search key(s)
  - *S* < the right child's search key(s)

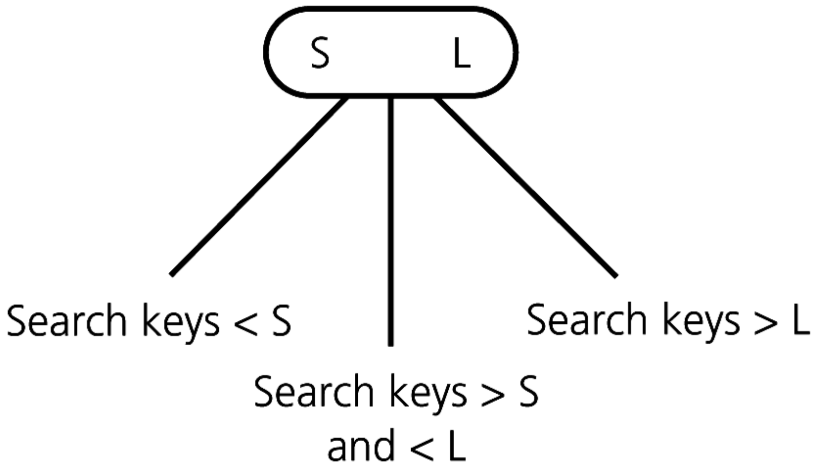
(a)



Placing Data Items in a 2-3 Tree

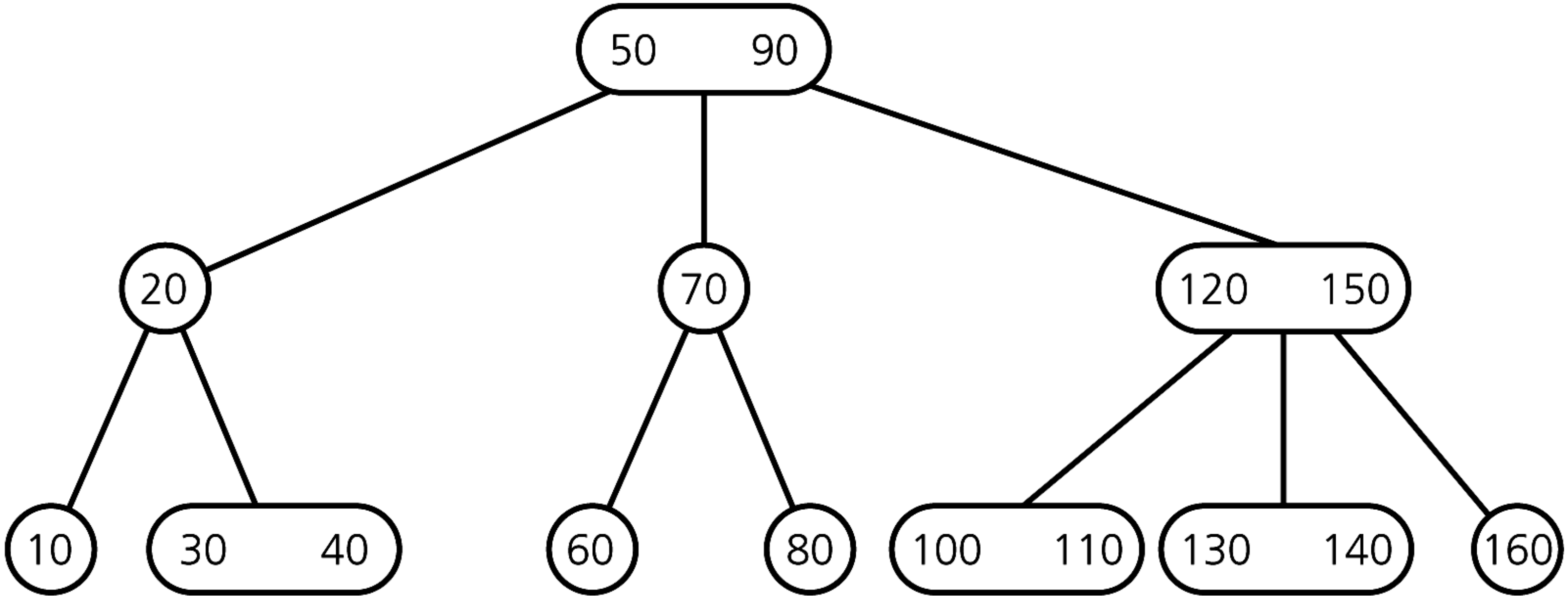
- A **3-node** contains two data items whose search keys *S* and *L* satisfy the following:
  - *S* > the left child's search key(s)
  - *S* < the middle child's search key(s)
  - *L* > the middle child's search key(s)
  - *L* < the right child's search key(s)

(b)



Placing Data Items in a 2-3 Tree

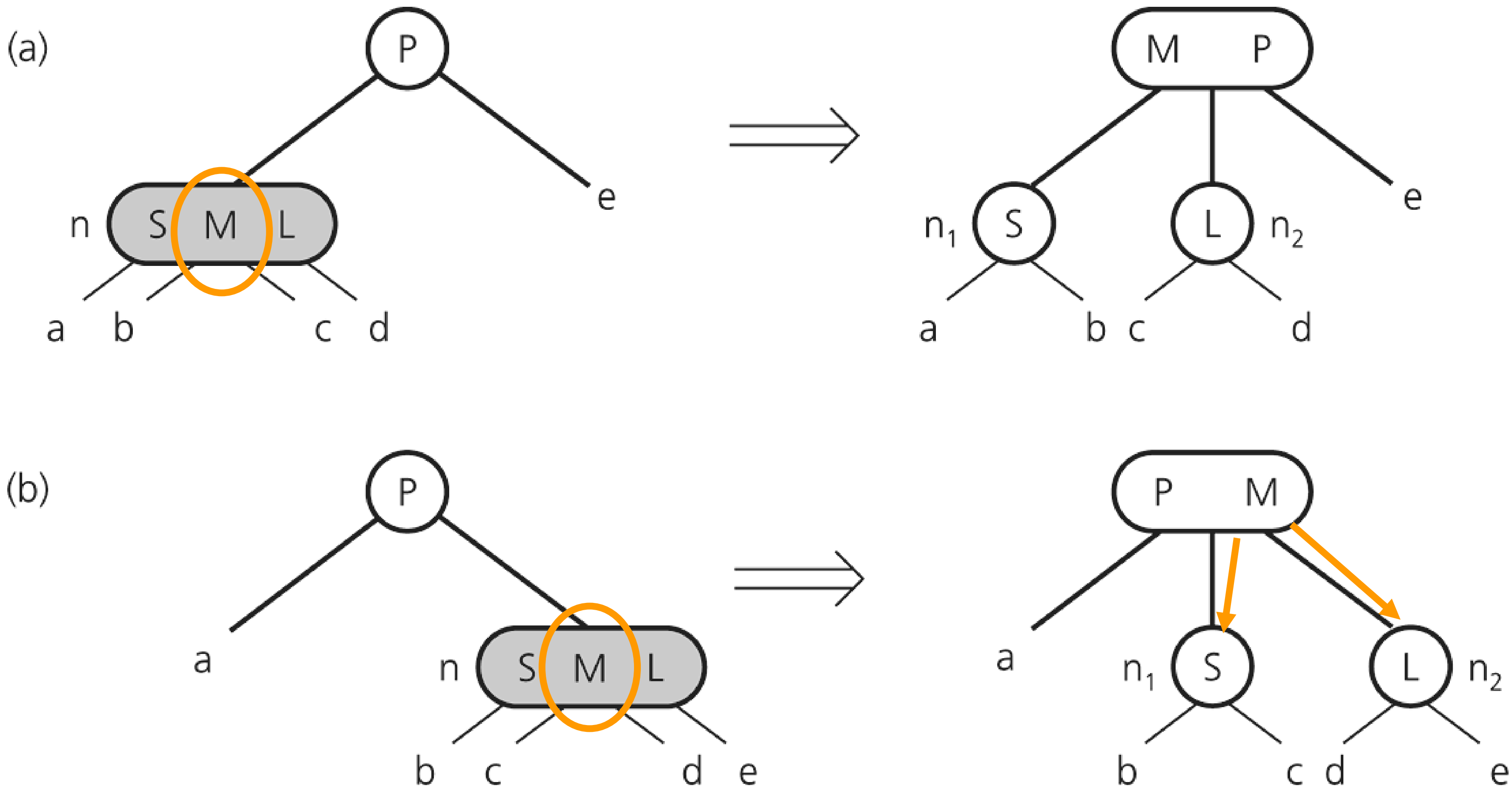
- A **leaf** may contain either one or two data items



2-3 Tree: *Operations*

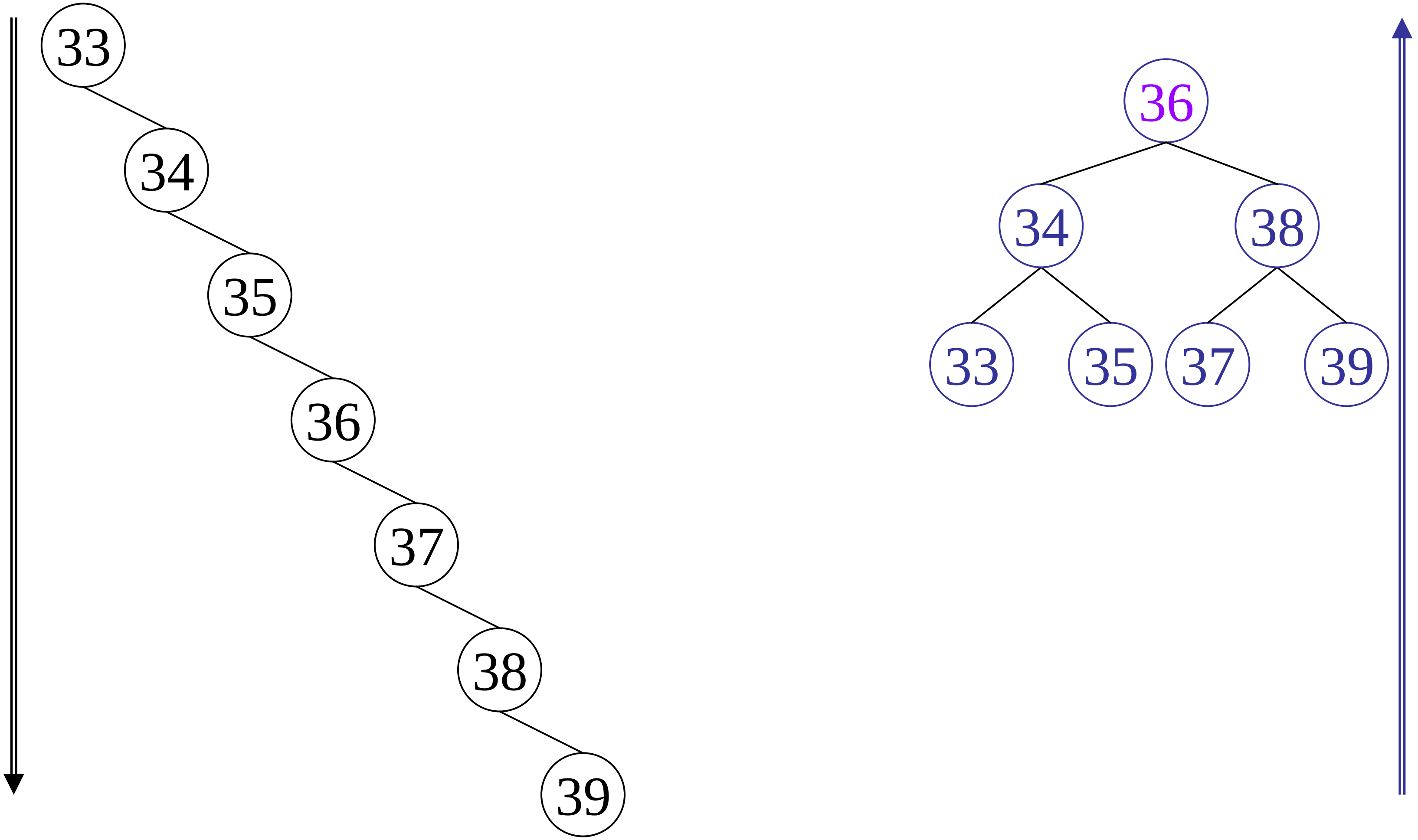
- **To traverse a 2-3 tree**
  - Perform the analogue of an inorder traversal
- **Searching a 2-3 tree is as efficient as searching the shortest binary search tree**
  - $O(\log_2 n)$
- **Insertion into a 2-node leaf is simple**
- **Insertion into a 3-node causes it to divide**

Inserting into a 2-3 Tree



2-3 Trees: *Advantage*

- Insert 33, 34, 35, 36, 37, 38, 39

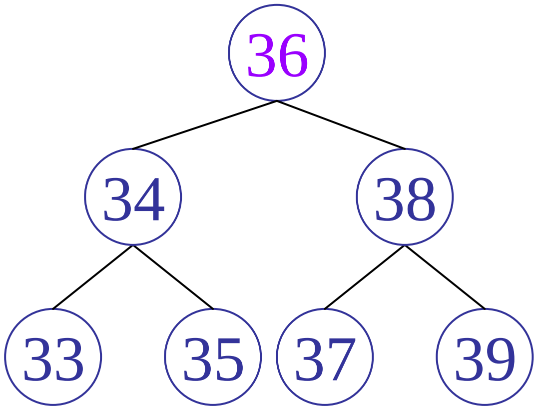


Inserting into a 2-3 Tree: Steps

- **To insert an item *I* into a 2-3 tree**
  1. Locate the leaf at which the search for *I* would terminate
  2. Insert the new item *I* into the leaf
  3. If the leaf now contains only two items, you are done
  4. If the leaf now contains three items, split the leaf into two nodes, n_1 and n_2

Inserting into a 2-3 Tree: Cases

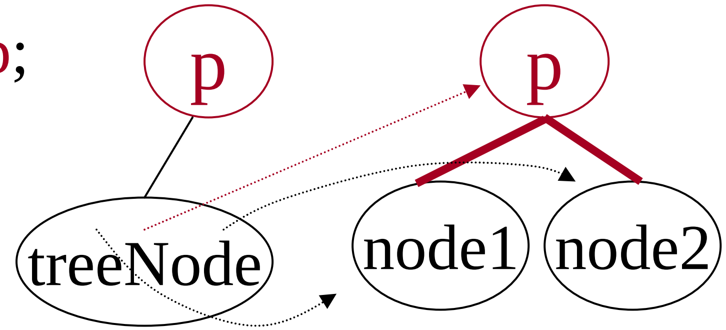
- **When an *internal node* contains three items (*recursion*)**
  - Split the node into two nodes
  - Accommodate the node's children
- **When the *root* contains three items**
  - Split the root into two nodes
  - Create a new root node
  - Tree grows in height



2-3 Tree: *Insertion Algorithm*

```
insertItem(in ttTree, in newItem)
    Locate and add newItem to leafNode;
    if (leafNode now has three items)
        split(leafNode);
```

```
split(inout treeNode)
    if (treeNode == root)        create a new node p;
    else                         p = parent of treeNode;
    Replace treeNode with node1 and node2, so that p is their parent;
    Place the smallest and largest keys into node1 and node2;
    Move the middle key up to p;
    if (p now has three items)
        split(p);
```



2-3 Tree: *Insertion Algorithm*

if (treeNode is not a leaf)

node1 = parent of two *leftmost* children under treeNode;

node2 = parent of two *rightmost* children under treeNode;

split(inout treeNode)

if (treeNode == root) create a new node p;

else p = parent of treeNode;

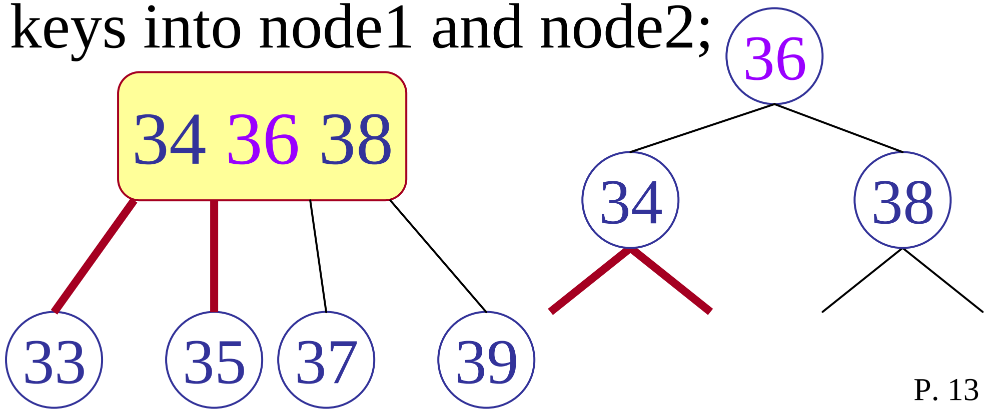
Replace treeNode with node1 and node2, so that p is their parent;

Place the *smallest* and *largest* keys into node1 and node2;

Move the *middle* key up to p;

if (p now has three items)

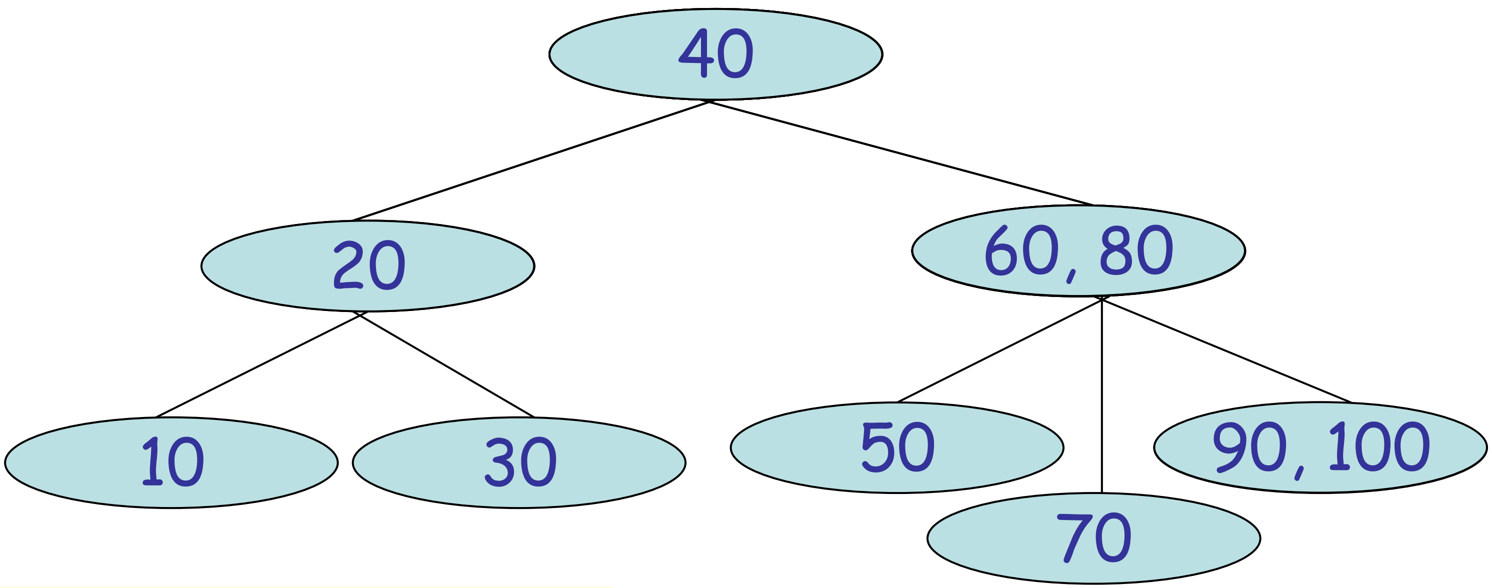
split(p);



2-3 Tree: *Insertion*

Build *2-3 tree* by insertion:

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



Insert into a leaf:
•Split (upward recursion)

Practice 1: Binary Search Tree vs. 2-3 Tree

- Input order: 10 12 30 8 60 40 70

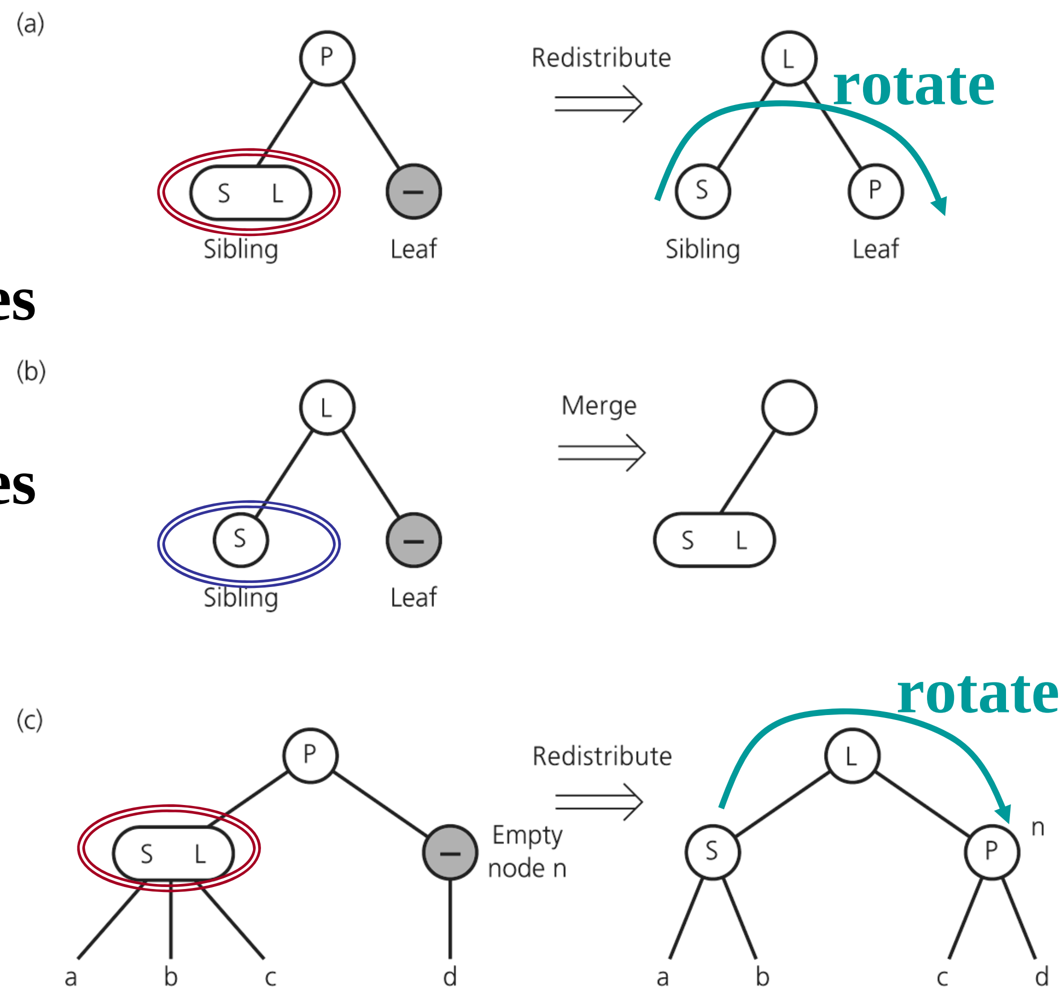
Deleting from a 2-3 Tree

What if the node is empty?

a) **Redistribute values**

b) **Merge into a leaf**

c) **Redistribute values and children**



Deleting from a 2-3 Tree

What if the node is empty?

a) Redistribute values

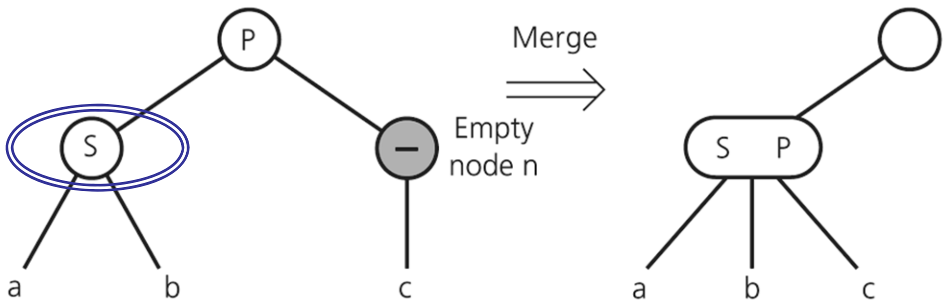
b) Merge a into leaf

c) Redistribute values and children

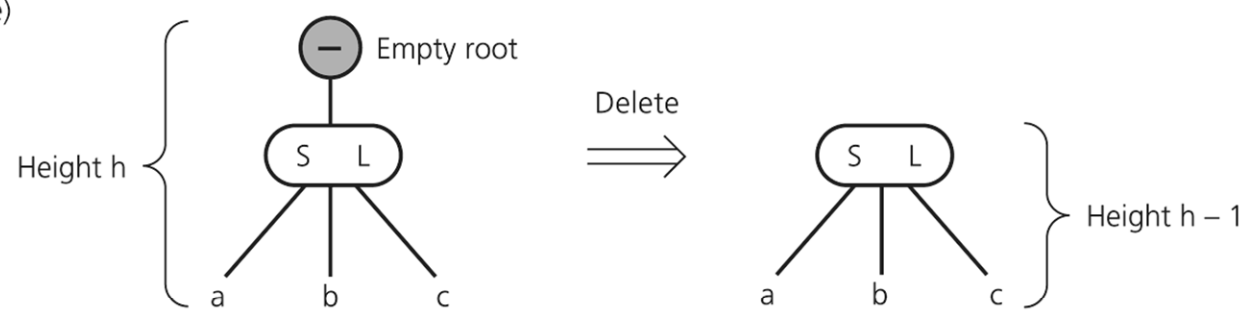
d) Merge into an internal node

e) Delete the root

(d)



(e)



Deleting from a 2-3 Tree: Steps

- **To delete an item *I* from a 2-3 tree**
  1. Locate the leaf at which the search for *I* would terminate
  2. Delete *I* from the leaf
  3. If the leaf now contains one item, you are done
  4. If the leaf now contains no item, choose one of the following operations to fix

  (a) Redistribute the values: *retain the tree structure*

  (b) Merge into a leaf: *its parent has one less child*

  Check if the nearest sibling is (a) 3-node or (b) 2-node.

Deleting from a 2-3 Tree: Cases

Data Structures

- **When an *internal node* contains no item (*recursion*)**
  - (c) Redistribute the values and children
  - (d) Merge into an internal node
- **When the *root* contains no item**
  - (e) Delete the root

- **What if item I is on an *internal node*?**
  1. Swap with the in-order successor on a leaf
  2. Delete I from the leaf…

2-3 Tree: *Deletion Algorithm*

deleteItem(in ttTree, in theKey)

    X = the tree node whose search key equals theKey;

    if (X is not a leaf)

        Y = *Successor*(X);

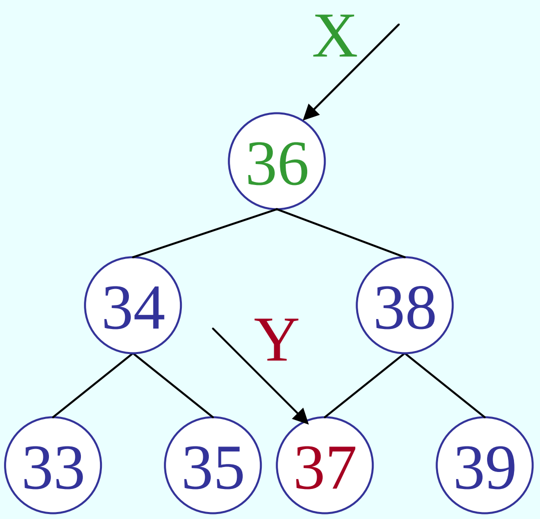
        swapKey(X, Y);

        X = Y;

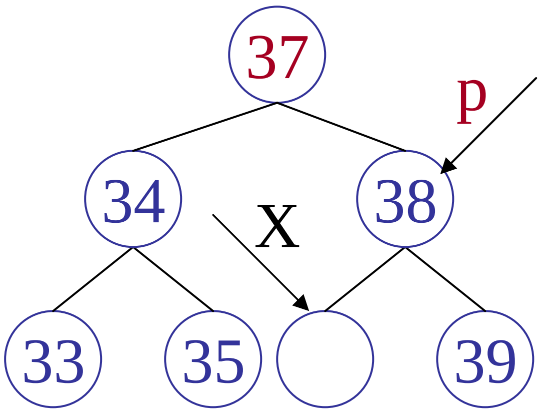
    Delete theKey from X;

    if (X now has no item)

        fix(X);



2-3 Tree: *Deletion Algorithm*




```
fix(in X)
    if (X == root)
        remove the root;
    else
        p = parent of X;
        if (the nearest sibling of X has two items)
            Redistribute items among the sibling, p and X;
            if (X is not a leaf)
                Move appropriate child from sibling to X;
        else            // merge
```

2-3 Tree: *Deletion Algorithm*

else // merge

s = the nearest sibling of X;

Move appropriate item down from p to s;

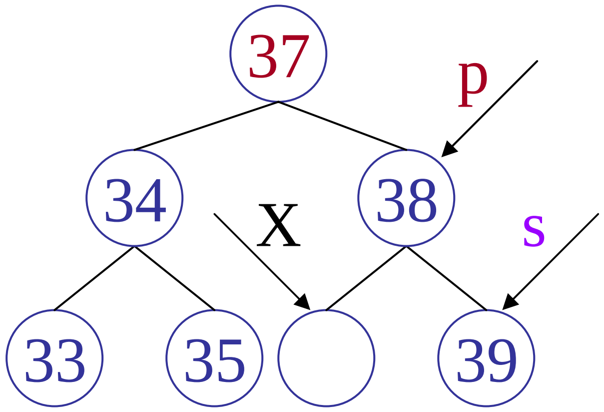
if (X is not a leaf)

Move X's child to s;

remove X;

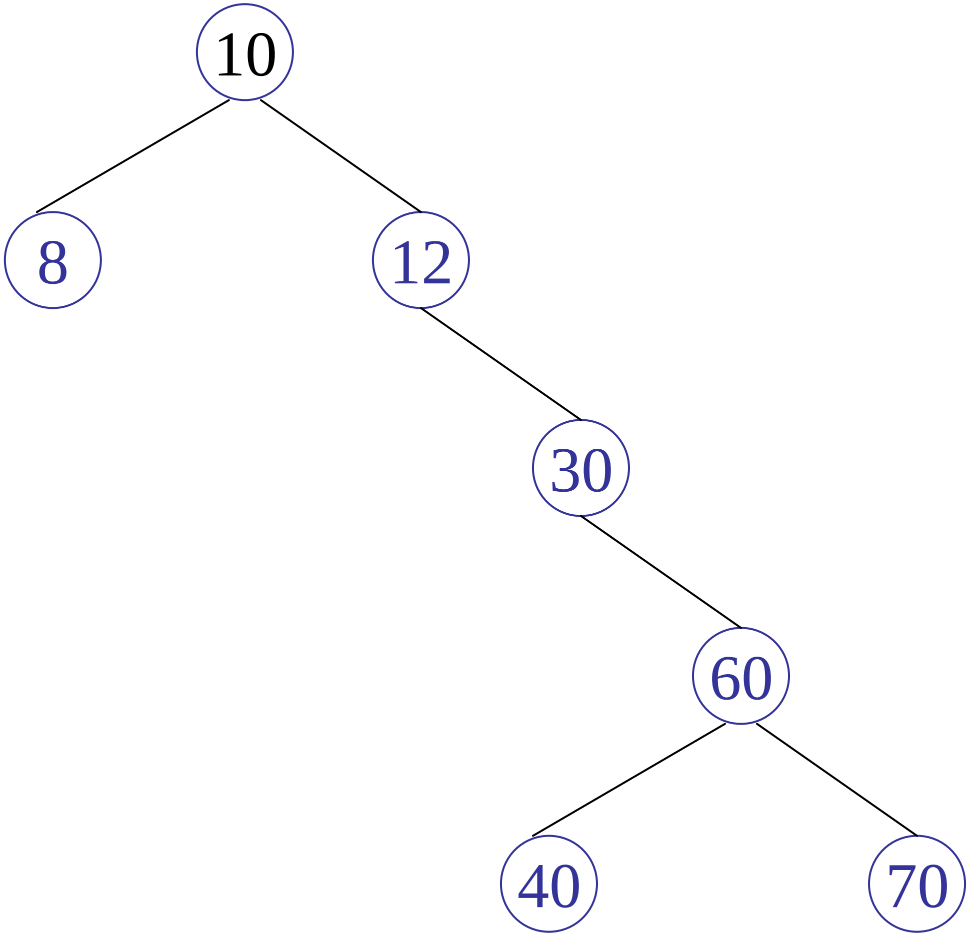
if (p now has no item)

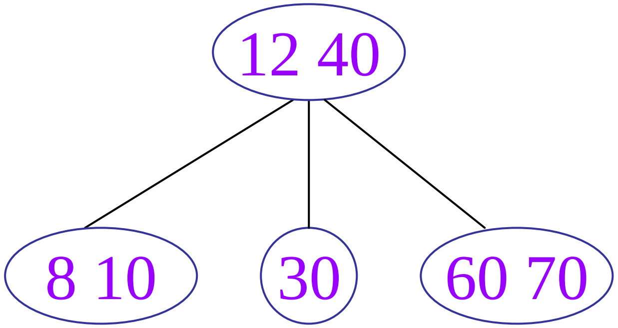
fix(p);



Practice 2: Binary Search Tree vs. 2-3 Tree

- After the deletions of 30, 10, 60





2-3 Tree: *Summary*

- **A 2-3 tree is a compromise**
  - Searching a 2-3 tree is not more efficient than searching a binary search tree of minimum height
    - **The 2-3 tree might be *shorter*, but that advantage is offset by the extra comparisons in a node having two values (3-node)**
  - Maintaining the balance of a 2-3 tree is relatively easy
    - **Maintaining the balance of a binary search tree is difficult**