Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

H. Cecotti

Learning outcomes

- More trees
  - 2-3 trees (1970)
    - Definition
    - Search
    - Insert
    - Delete
  - Key aspects
    - Merging
    - Redistribution
- You must:
  - Know how to implement and use these trees
  - Trace the state of a tree after an insert or delete

Introduction

- Binary Search Trees (BSTs)
  - 1 node = 1 key with **2** children
- Next logical step…
  - Multiple keys
  - Multiple children
- Example
  - 2-3 trees
    - **2 keys max + 3 children max**
  - B-trees

2-3 trees

- Definitions
  - An internal node is a 2-node if it has 1 data element and 2 children.
  - An internal node is a 3-node if it has 2 data elements and 3 children.
  - T is a 2–3 tree if and only if one of the following statements hold:
    - T is empty. (no nodes)
    - T is a 2-node with data element a.
      - **If** T has left child L and right child R **then**
        - L and R are non-empty 2–3 trees of the same height;
        - a > than each element in L, and
        - a ≤ to each data element in R.
    - T is a 3-node with data elements a and b, where a < b.
      - **If** T has left child L, middle child M, and right child R **then**
        - L, M, and R are non-empty 2–3 trees of equal height
        - a > than each data element in L and ≤ to each data element in M, and
        - b > than each data element in M and ≤ to each data element in R.

2-3 trees

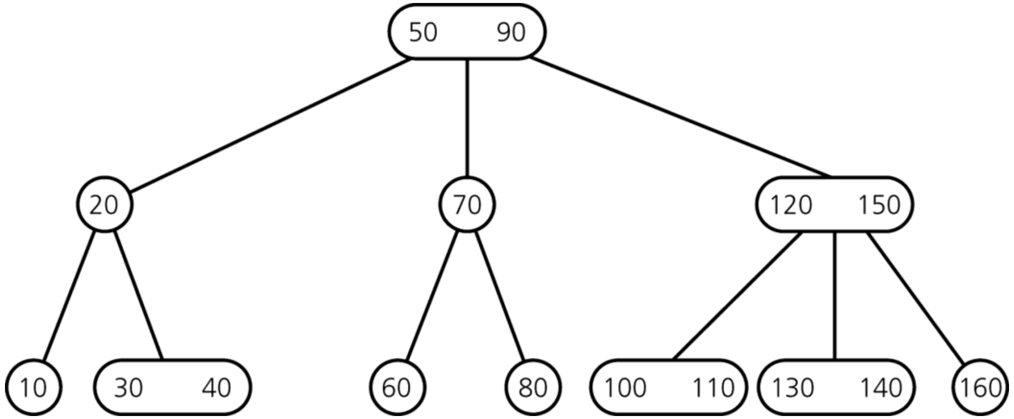
- Properties
  - Every internal node is a 2-node **or** a 3-node.
  - All the leaves are at the same level.
  - All data is kept in sorted order!

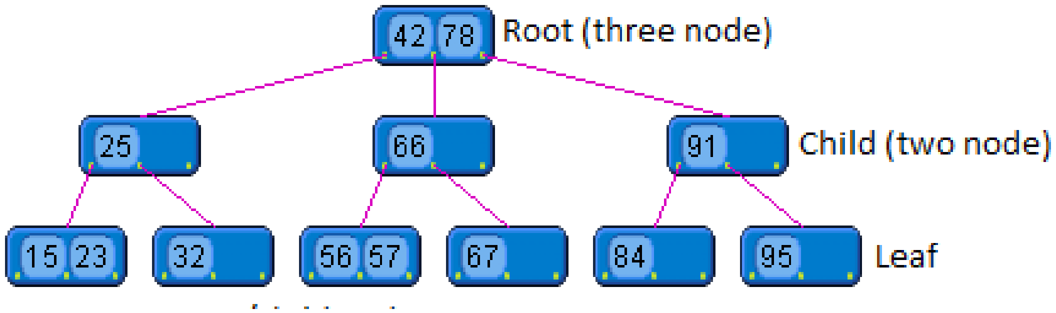
- Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

2-3 trees

- Example





2-3 trees

- Traversing
  - In order
    - Pseudo code

```
inorder(in ttTree: TwoThreeTree) {
    if(ttTree's root node r is a leaf)
        visit the data item(s)
    else if(r has two data items)
    {
        inorder(left subtree of ttTree's root)
        visit the first data item
        inorder(middle subtree of ttTree's root)
        visit the second data item
        inorder(right subtree of ttTree's root)
    }
    else
    {
        inorder(left subtree of ttTree's root)
        visit the data item
        inorder(right subtree of ttTree's root)
    }
}
```

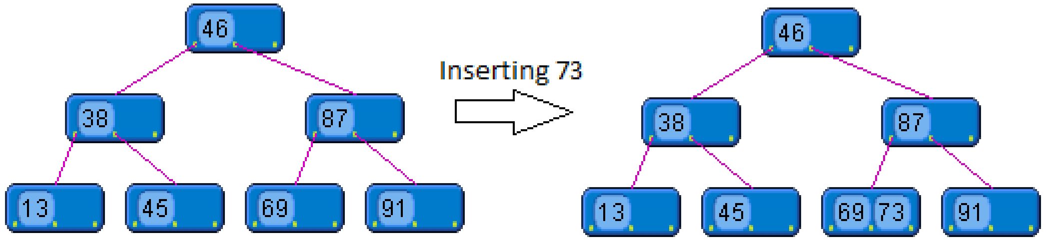
2-3 trees

- Searching
  - ➢Pseudo-code

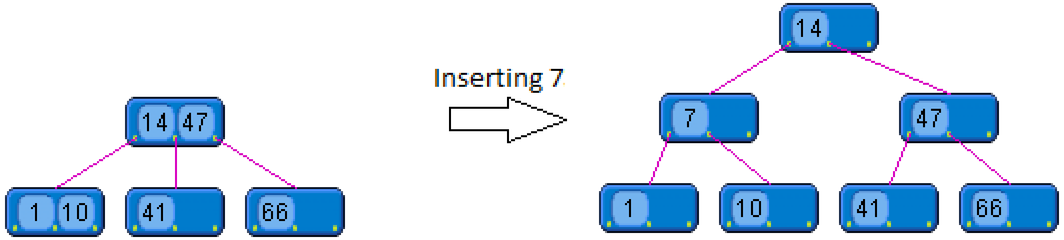
```
bool retrieveItem(in ttTree: TwoThreeTree, in searchKey:KeyType,
            out treeItem:TreeItemType) {
    if(searchKey is in ttTree's root node r)
    {
        treeItem = the data portion of r
        return true
    }
    else if(r is a leaf)
        return false
    else
    {
        return retrieveItem(appropriate subtree,
                searchKey, treeItem)
    }
}
```

2-3 trees

- Insert an element
  - Case 1: Insert in a node with remaining place



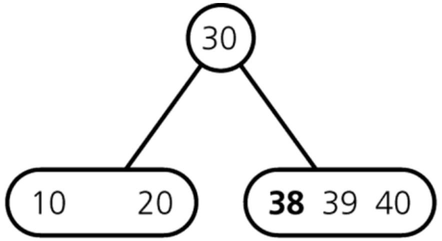
  - Case 2: Insert in a node with already 2 keys



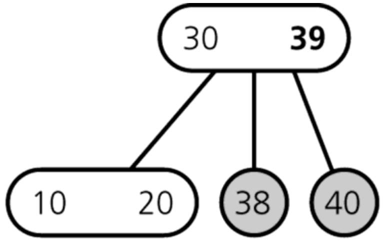
2-3 trees

- Insert an element
  - Insert 38

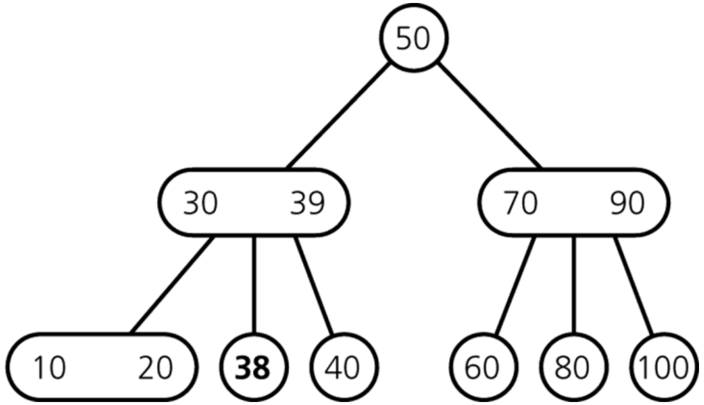
Insert in leaf
Not enough space!



Divide the leaf
and move middle
value up to parent

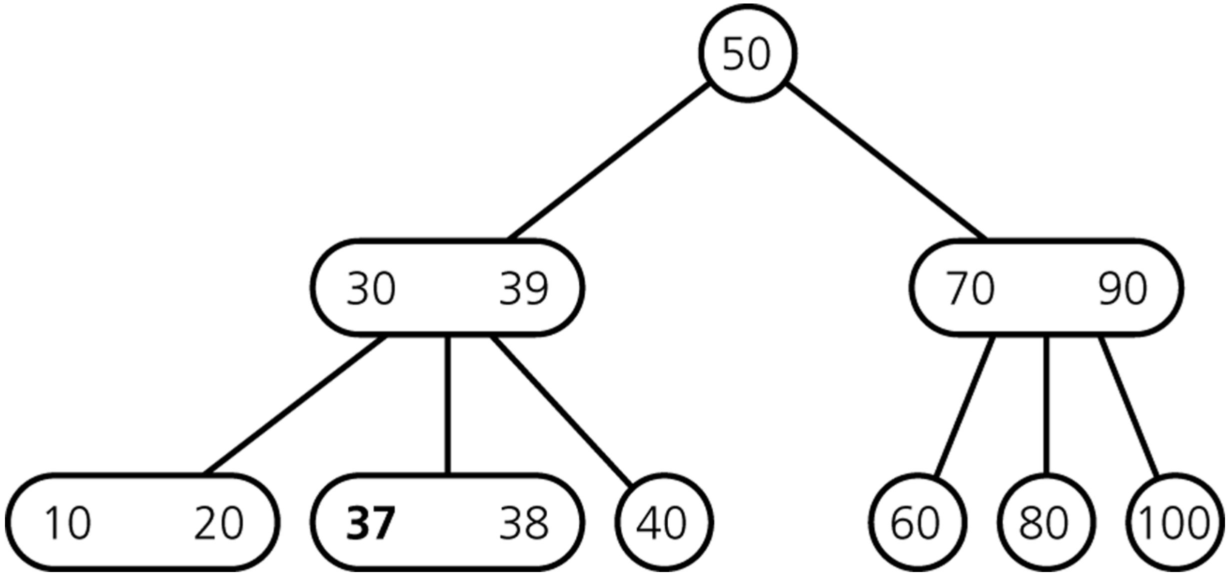


End



2-3 trees

- Insert an element
  - Insert 37
    - Node has only 1 key
    - **Direct** Insert



2-3 trees

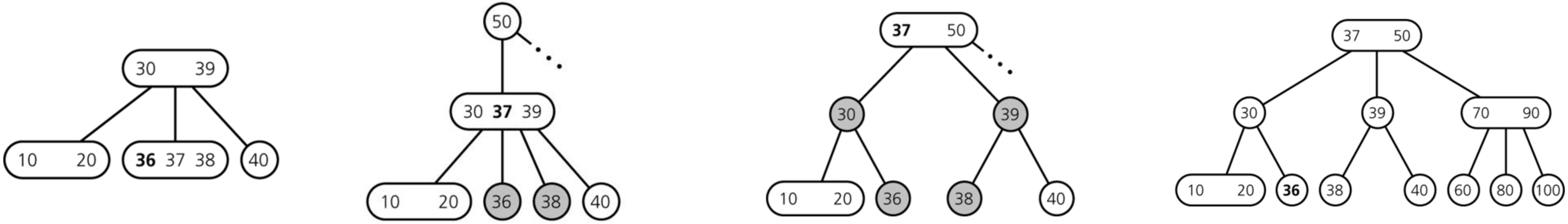
- Insert an element
  - ➢Insert 36

Insert in leaf

Divide leaf and move **middle** value up to parent

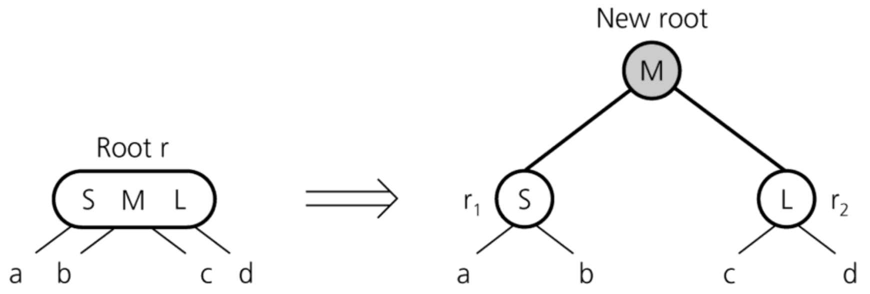
Divide the overcrowded node, move **middle** value up to parent, attach children to smallest and largest

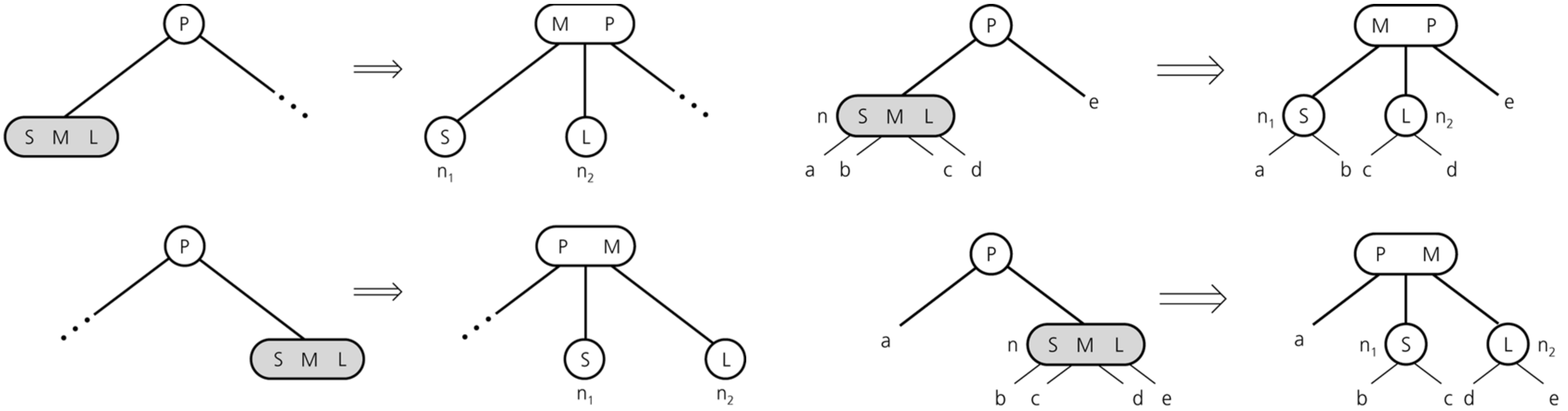
Result



2-3 trees

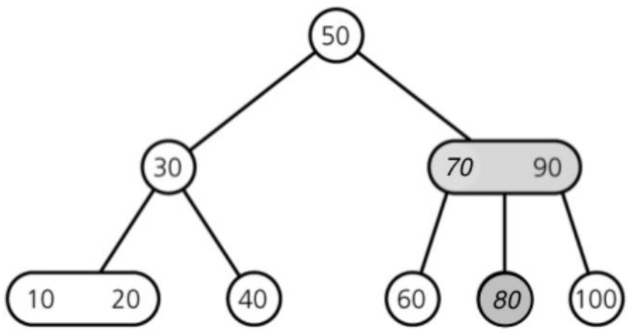
- The main rules for insertion
  - ➢ Common points with rebalancing BST ☺
  - ➢ Warning
    - o Possibility to create new nodes!

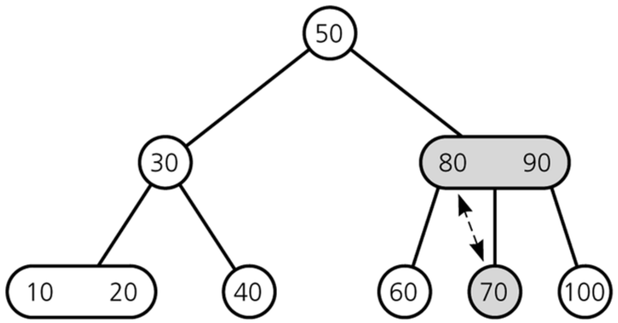




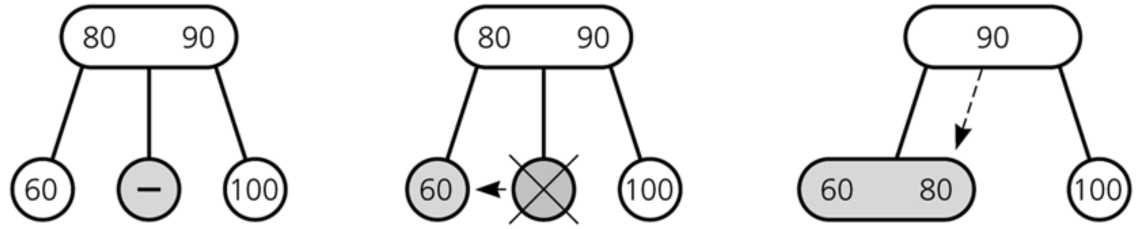
2-3 trees

- Delete an element
  - Delete 70





Swap with inorder successor

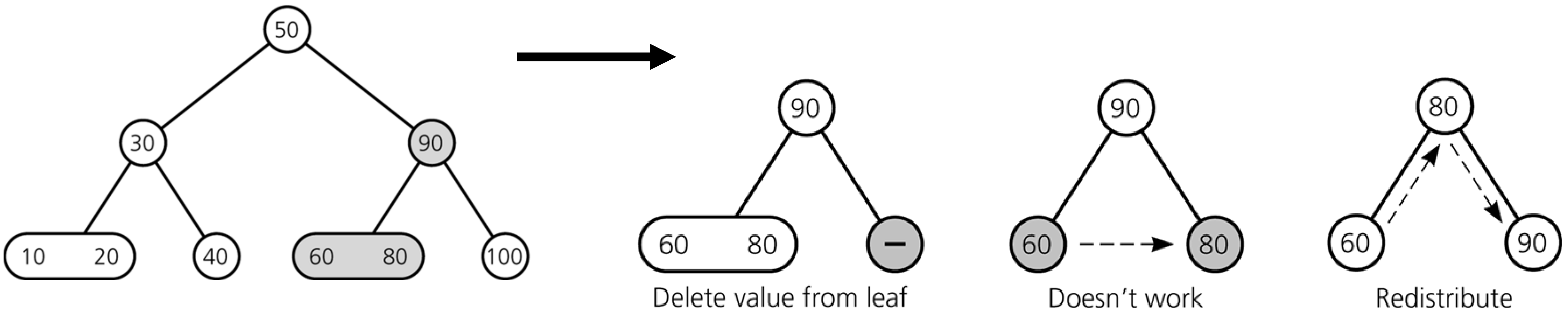


Delete value from leaf

Merge nodes by deleting empty leaf and moving 80 down

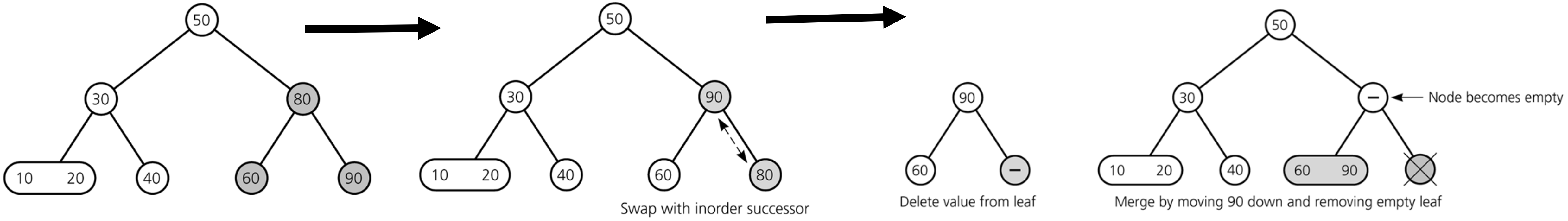
2-3 trees

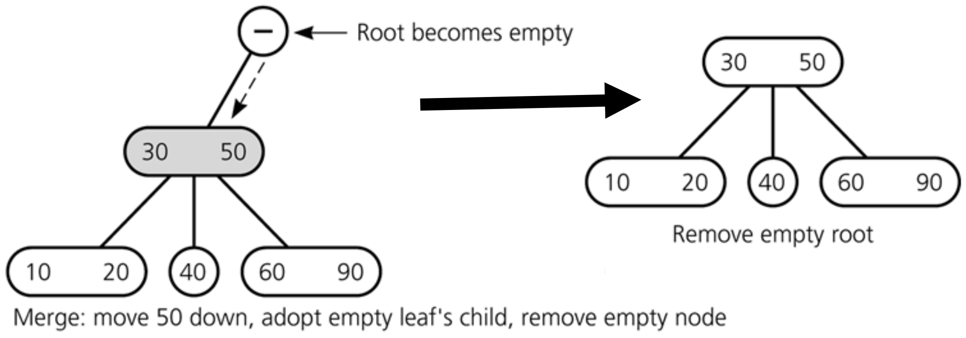
- Delete an element
  - Delete 100

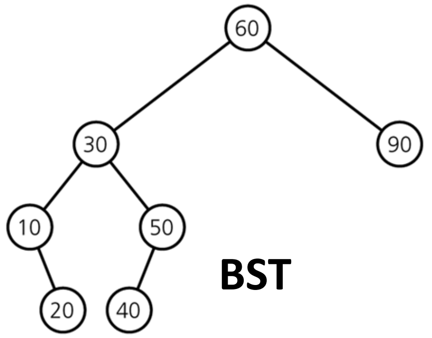


2-3 trees

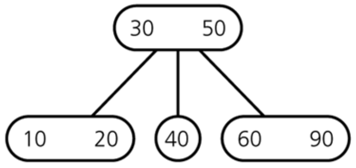
- Delete an element
  - Delete 80







BST



2-3 Tree

2-3 trees

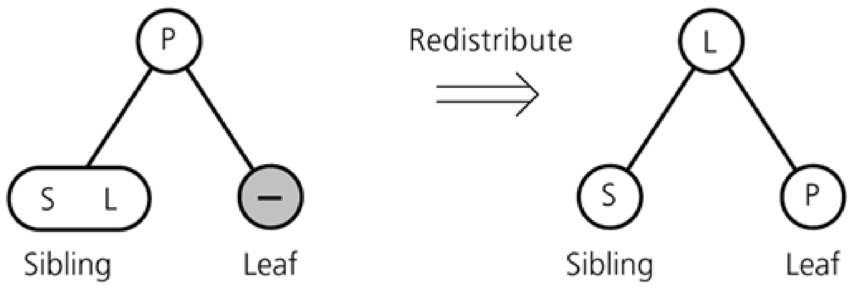
- Delete a node
  - Pseudo code
    - Locate node *n* containing item *i*
      - It may be null if there is no such item
    - If (node *n* is **not** a leaf)
      - Then swap *I* with in-order successor
        - The deletion always begins at a leaf
    - If (leaf node *n* **contains** another item)
      - Then
        - just delete item
      - Else
        - if (possible to redistribute nodes from siblings) (case 1 & 3)
        - Else
        - merge node (case 2 & 4)

2-3 trees

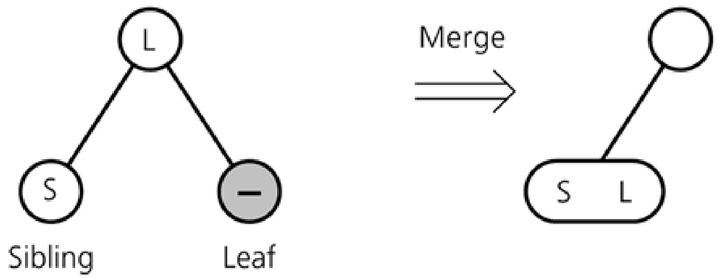
- The main rules for deletion
  - **Case 1**: Redistribution
    - A sibling has 2 items: redistribute item between siblings and parent
  - **Case 2**: Merging
    - No sibling has 2 items: merge node, move item from parent to sibling
  - **Case 3**: Redistribution
    - Internal node n has no item left: redistribute
  - **Case 4**: Merging
    - Redistribution is not possible:
      - 1/ merge node, 2/ move item from parent to sibling, 3/ adopt child of n
  - **Special case**:
    - Reaching the root
      - If merging process reaches the root and root is without item: just delete root ☺

2-3 trees

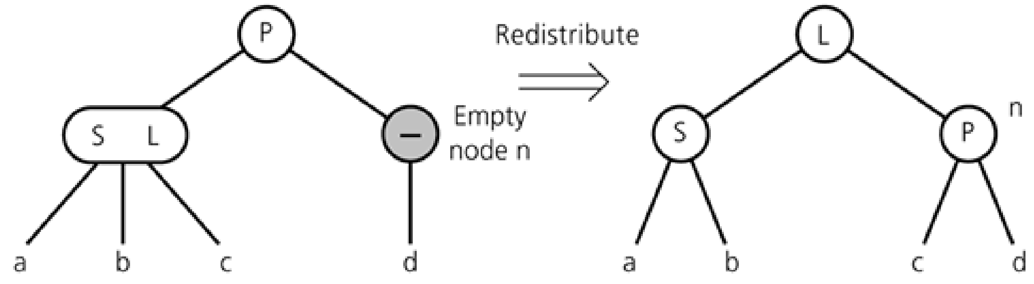
- Case 1:



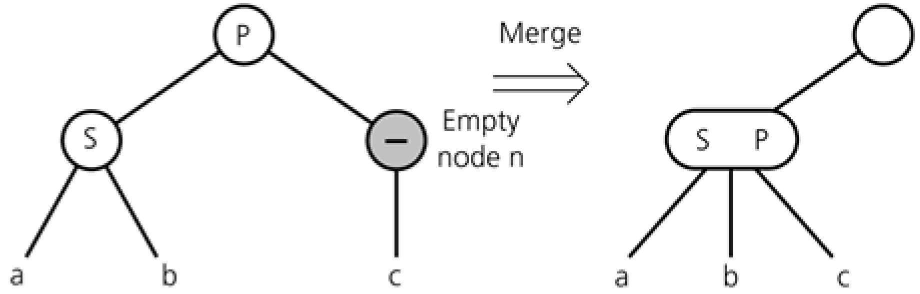
- Case 2:



Case 3:

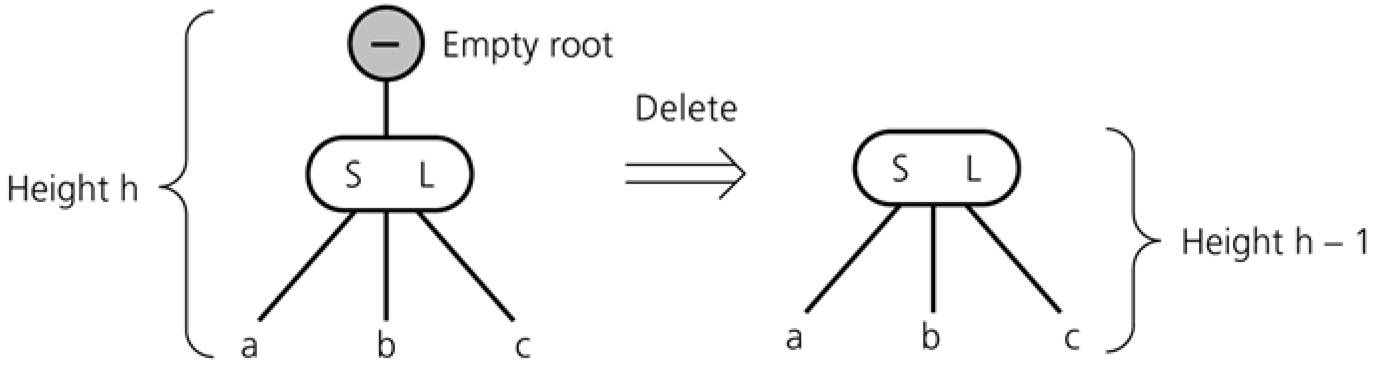


Case 4:



2-3 trees

- Special case
  - ➢Delete root → the node under becomes the root



Conclusion

- B-trees
  - We will see them at the end of the semester
- Complexity
  - 2-3 tree & B-tree

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Questions ?

- Acknowledgment + Reading
  - ➢Canvas: Csci 115 book: Section 7.3
  - ➢Chapter 18, B-trees, Introduction to Algorithms, 3rd Edition.