Data Structure

http://smartlead.hallym.ac.kr

Instructor: Jin Kim
010-6267-8189(033-248-2318)
jinkim@hallym.ac.kr
Office Hours:

Non Linear Data Structure

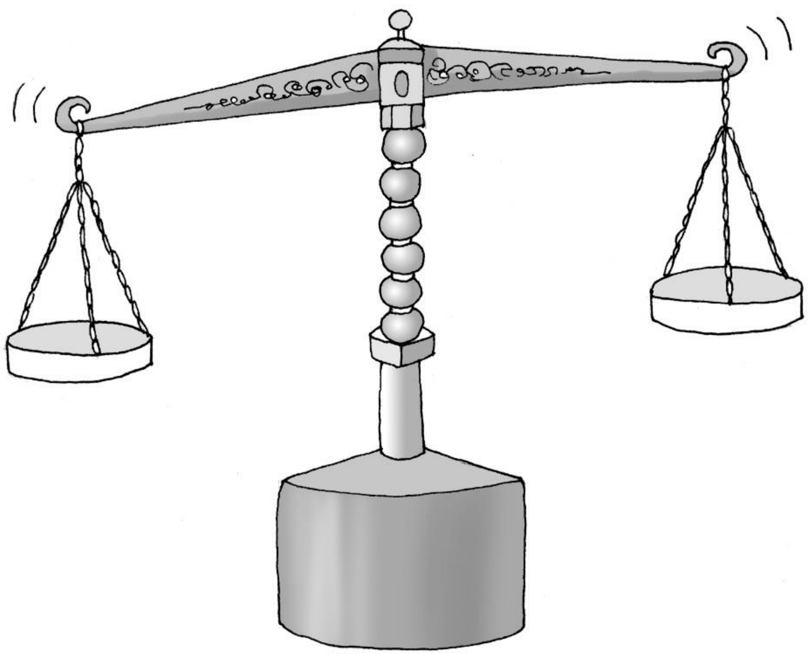
- Data structure we will consider this semister:
  - Tree
  - Binary Search Tree
  - Graph
  - Weighted Graph
  - Sorting
  - Balanced Search Tree

Balanced Search Trees
균형 탐색 트리

2-3 Trees
(2-3 트리)

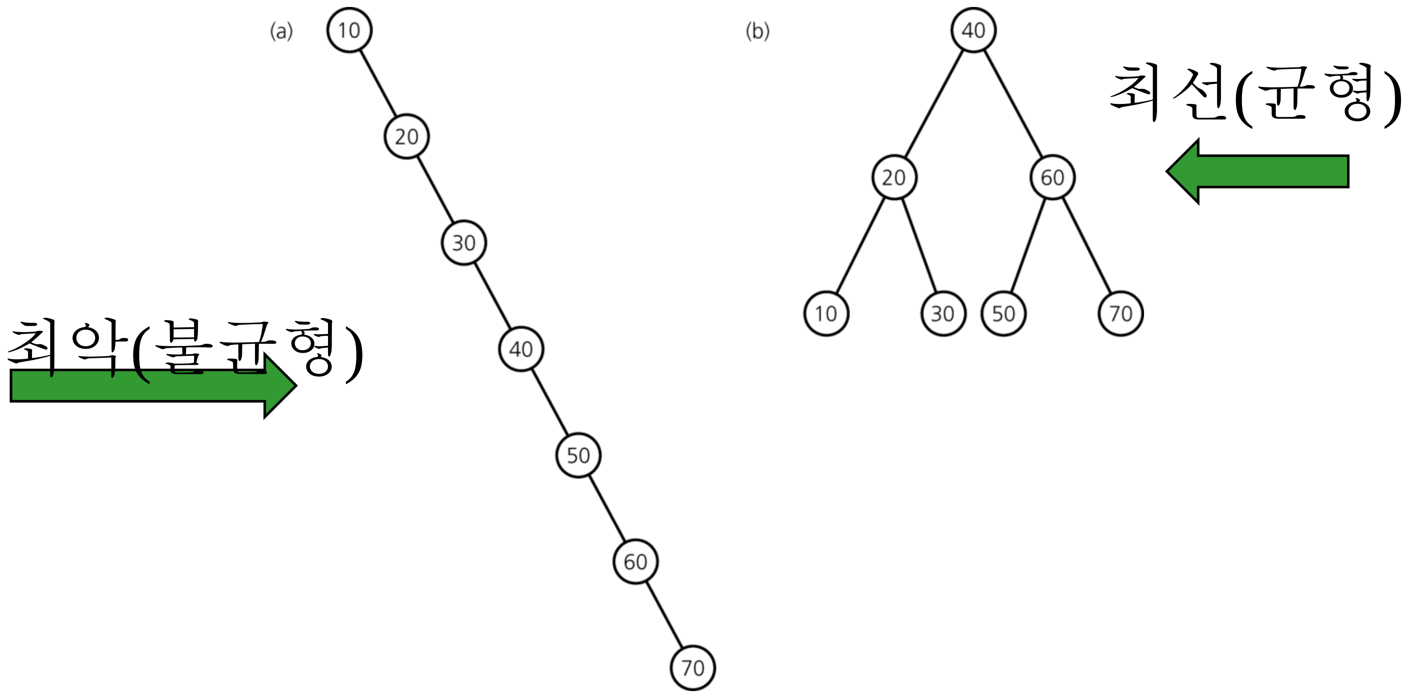
Balanced?

- 균형



Why care about advanced implementations?

Same entries, different insertion sequence:

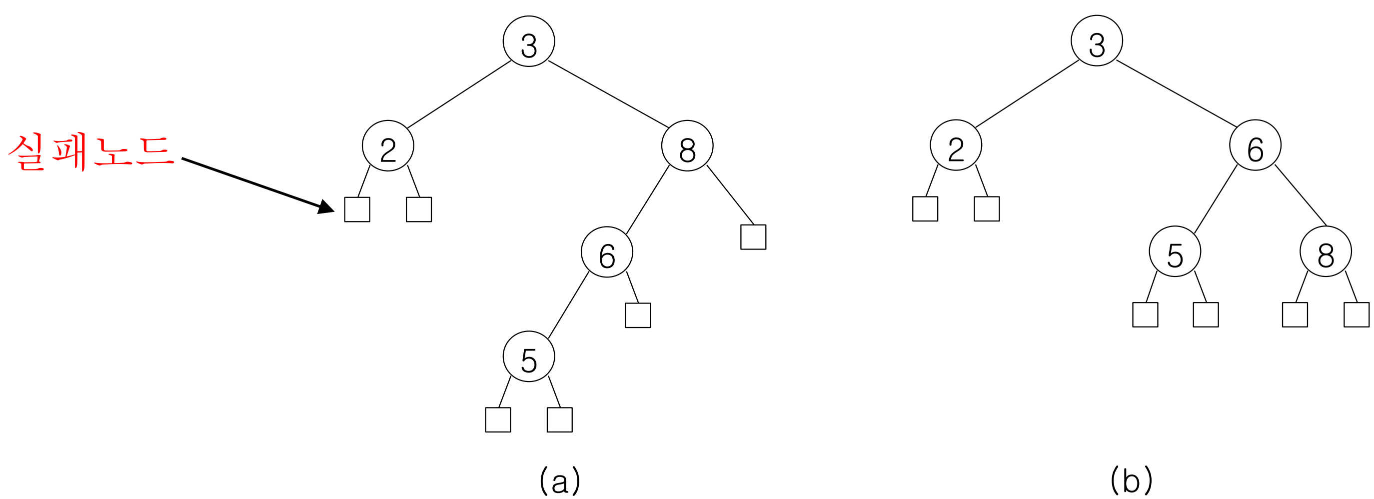


(a) Skewd bst 불균형 (b) complete bst

→ Not good! Would like to keep tree balanced.

Extended binary tree

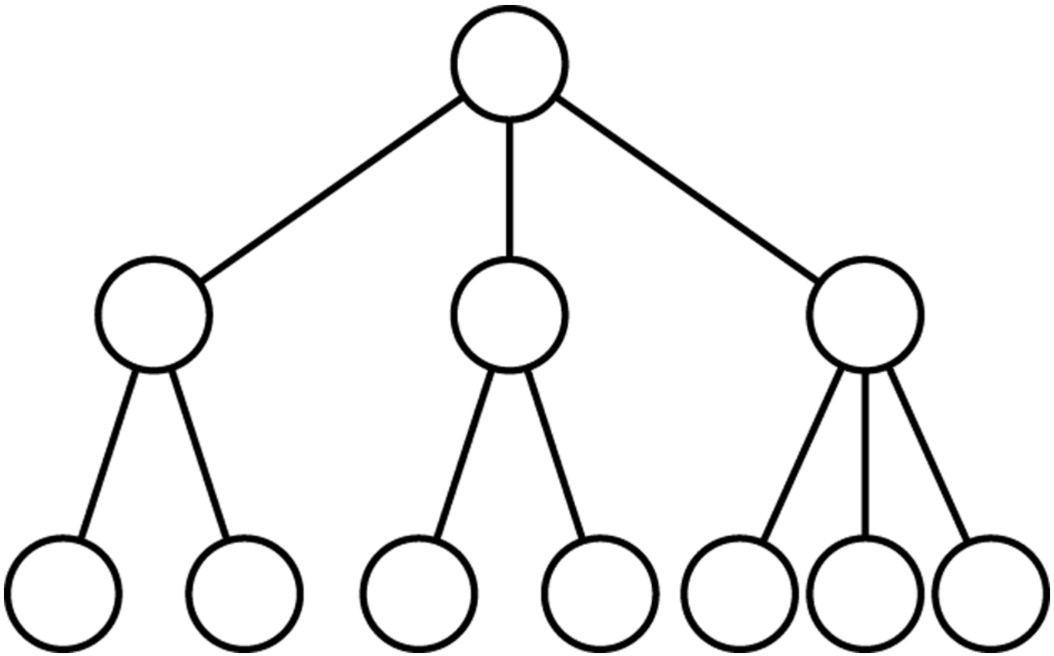
- 외부 노드(external node)
  - 이진 트리 : n개의 노드, n+1개의 널 링크
  - 널 링크에 사각형 노드(외부 노드)를 붙이면 처리에 편리
  - 실패 노드(failure node)라고도 한다.
  - cf) 내부 노드(internal node) : 원래의 트리 노드
- 확장 이진 트리(extended binary tree)
  - 외부 노드가 추가된 이진 트리



2-3 Trees

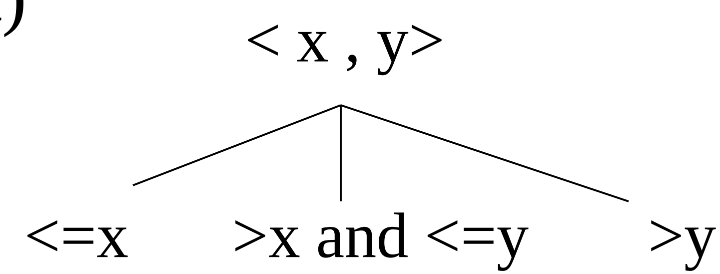
Features

- each internal node has either 2 or 3 children
- all leaves are at the same level (단말노드는 같은 레벨)
- Balanced(균형이 이루어짐)



2-3 Trees

- Relax constraint that a node has 2 children(자식2명제한을 완화)
- Allow 2-child nodes and 3-child nodes
  - With bigger nodes, tree is shorter & branchier
  - 2-node is just like before (one item, two children)
  - 3-node has two values and 3 children (left, middle, right)



2-3 tree searching algorithm

```
twoThreeSearch(x)
    for(p ← root; p; )          // root는 2-3 트리의 루트 노드
        switch (compare(p, x)) {
            case 1 : p ← p.left;     break;
            case 2 : p ← p.middle;     break;
            case 3 : p ← p.right;     break;
            case 4 : return p;     // x는 p의 키 중에 하나
        }
end twoThreeSearch()
```

Why 2-3 tree

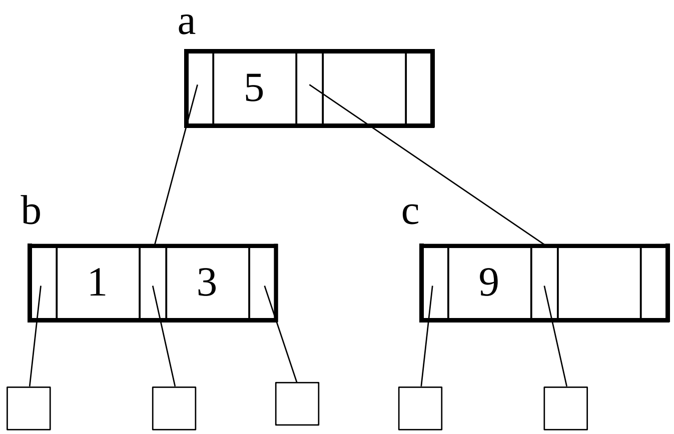
- Faster searching?
  - Actually, no. 2-3 tree is about as fast as an "equally balanced" binary tree, because you sometimes have to make 2 comparisons to get past a 3-node
- Easier to keep balanced?
  - Yes, definitely.
  - Insertion can split 3-nodes into 2-nodes, or promote 2-nodes to 3-nodes to keep tree approximately balanced!

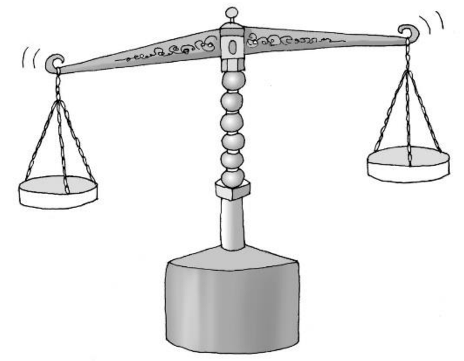
2-3 Trees

- Traversing a 2-3 tree
  - To traverse a 2-3 tree
    - Perform the analogue of an inorder traversal
- Searching a 2-3 tree
  - Searching a 2-3 tree is as efficient as searching the shortest binary search tree
    - Searching a 2-3 tree is $O(\log_2 n)$
    - Number of comparisons required to search a 2-3 tree for a given item
      - Approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible

2-3 트리 (3)

- 2-3 트리의 예



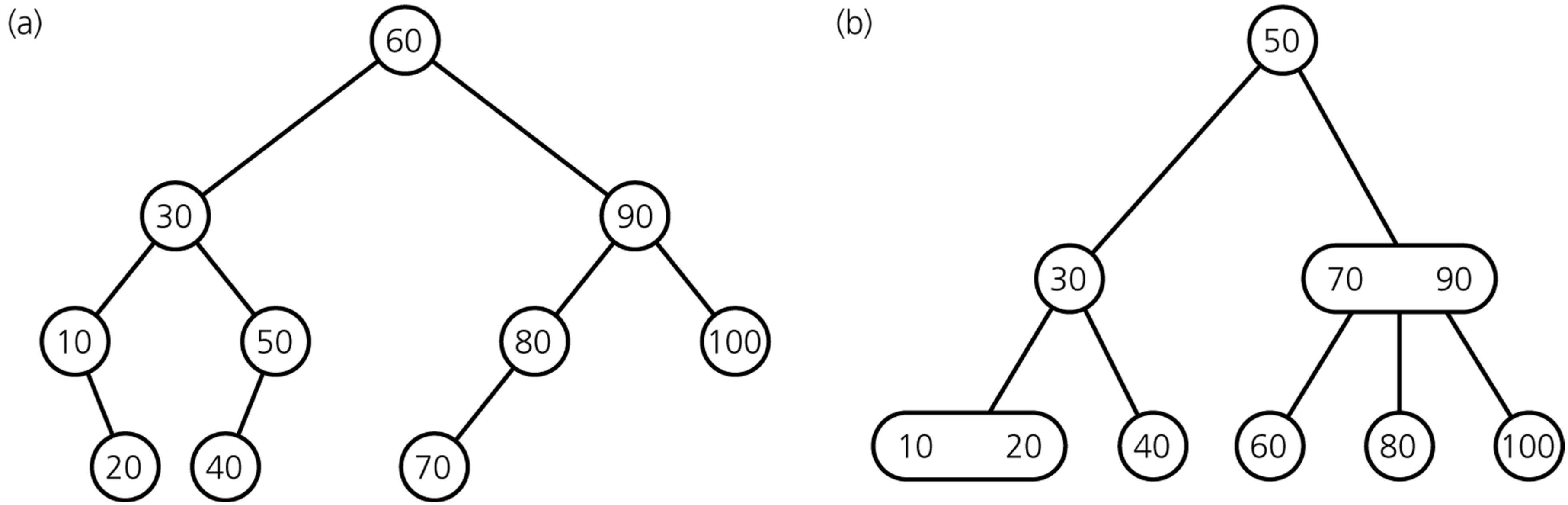


  - a, c : 2-노드, b : 3-노드
- 높이가 h인 2-3 트리의 키수
  - $2^{h+1}-1$과 $3^{h+1}-1$ 사이
- n개의 키값을 가진 2-3 트리의 높이
  - $\lceil \log_3(n+1)-1 \rceil$과 $\lceil \log_2(n+1)-1 \rceil$ 사이

2-3 Trees

- Advantage of a 2-3 tree over a balanced binary search tree
  - Maintaining the balance of a binary search tree is difficult
  - Maintaining the balance of a 2-3 tree is relatively easy

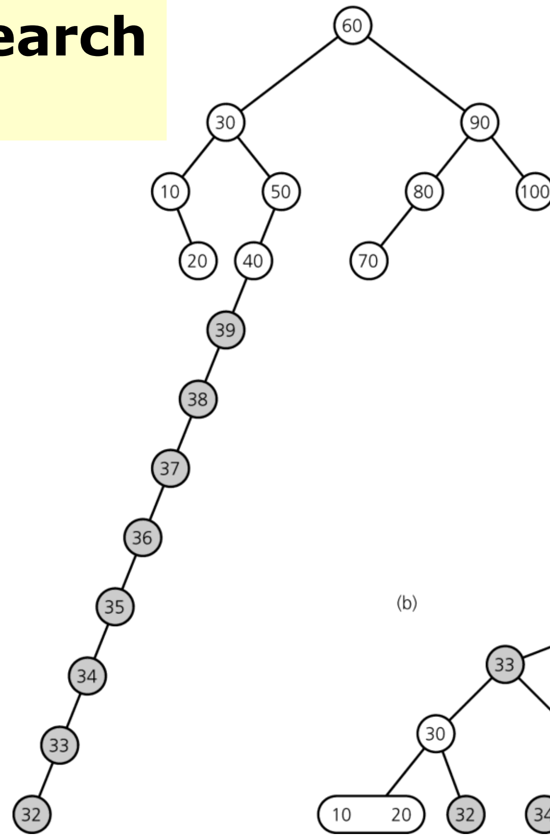
What did we gain?



What is the time efficiency of searching for an item?

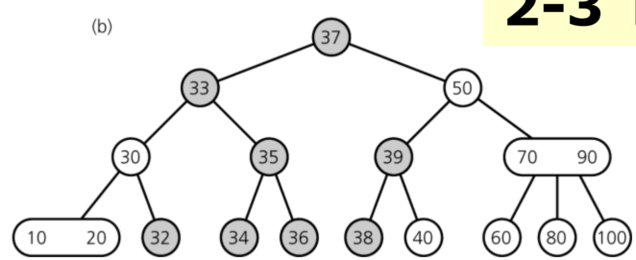
Gain: Ease of Keeping the Tree Balanced (이진탐색트리보다 균형을 유지하기 쉽다)

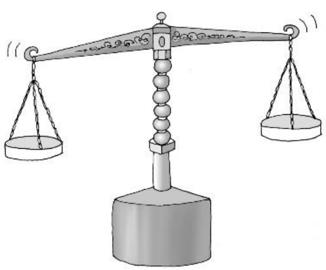
Binary Search Tree



both trees after inserting items 39, 38, ... 32

2-3 Tree

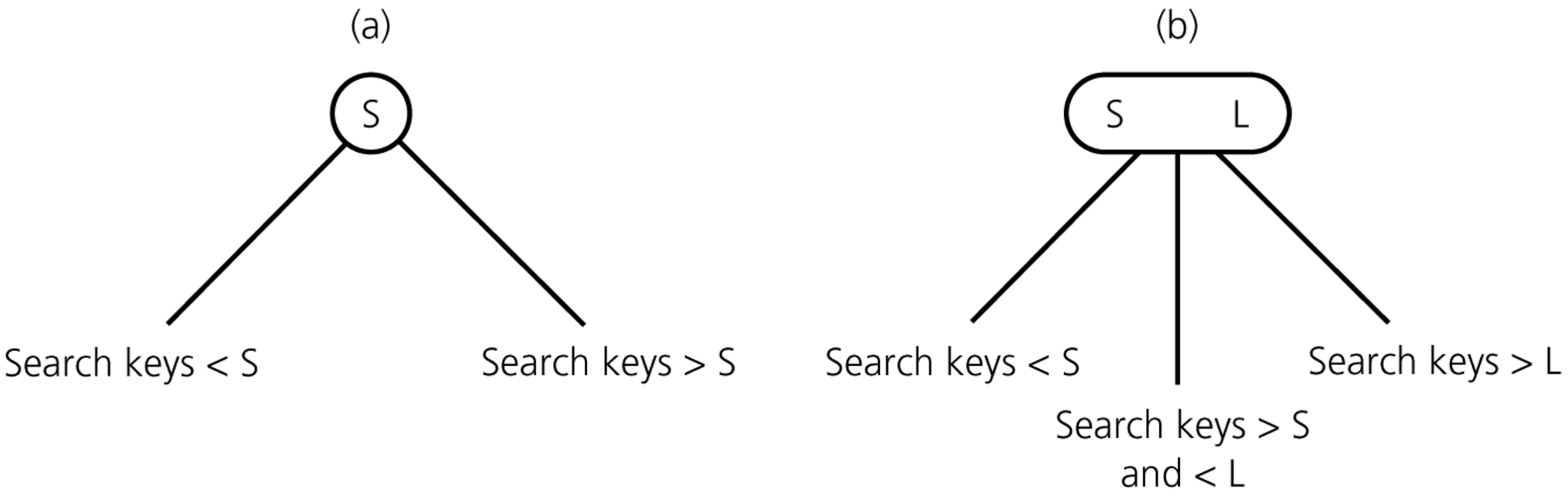




루트에서 단말까지의 깊이가 똑같다. 완전한 균형

2-3 Trees with Ordered Nodes

2-node **3-node**



- leaf node can be either a 2-node or a 3-node

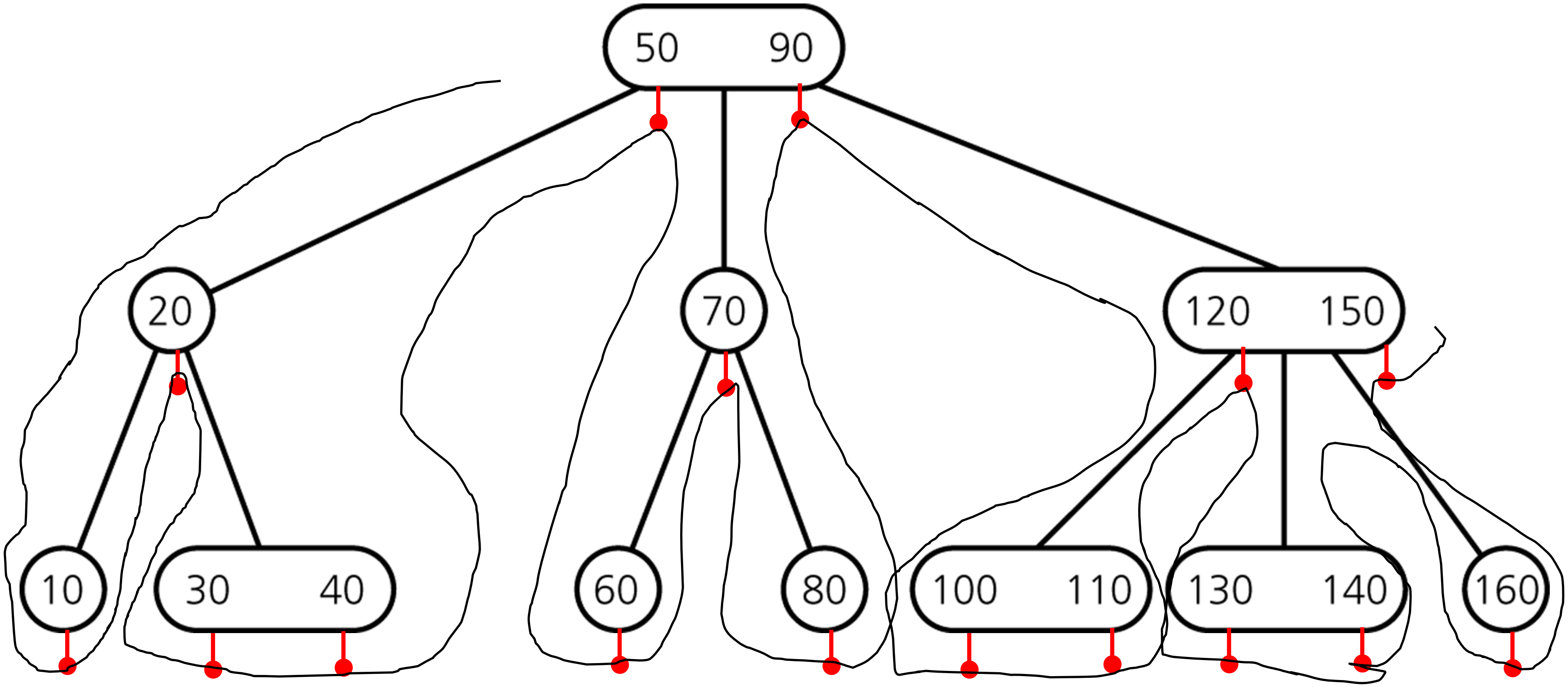
Traversing a 2-3 Tree

```
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)
    }
```

Searching a 2-3 Tree

```
retrieveItem(in ttTree: TwoThreeTree,
             in searchKey:KeyType,
             out treeItem:TreeItemType):boolean
   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)
   }
```

Example of 2-3 Tree



Inorder

2-3 Trees Insertion

- To insert an item, say key, into a 2-3 tree
  1. Locate the leaf at which the search for key would terminate
  2. If leaf is null (only happens when root is null), add new root to tree with item
  3. If leaf has one item insert the new item key into the leaf
  4. If the leaf contains 2 items, split the leaf into 2 nodes n_1 and n_2

2-3 Trees Insertion

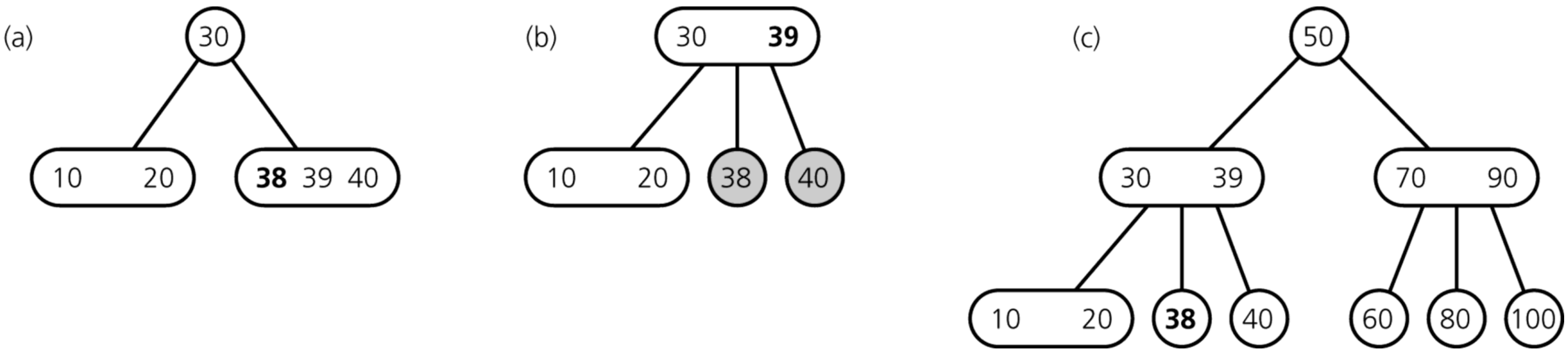
- When an internal node would contain 3 items
  1. Split the node into two nodes
  2. Accommodate the node’s children
- When the root contains three items
  1. Split the root into 2 nodes
  2. Create a new root node
  3. The tree grows in height

Inserting Items

Insert 38

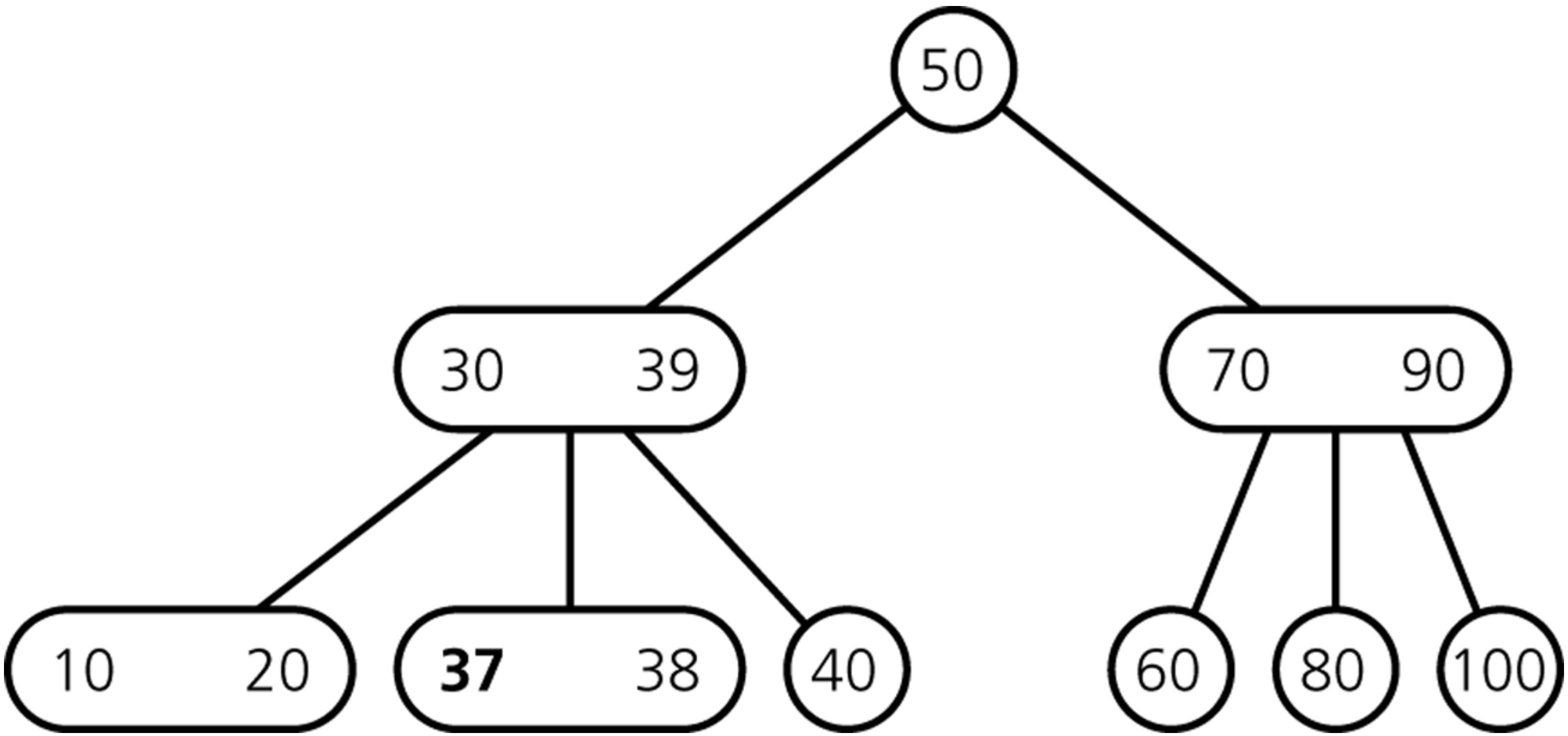
insert in leaf

divide leaf
and move middle
value up to parent



Inserting Items

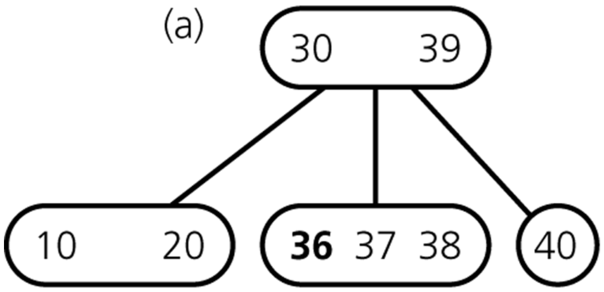
Insert 37



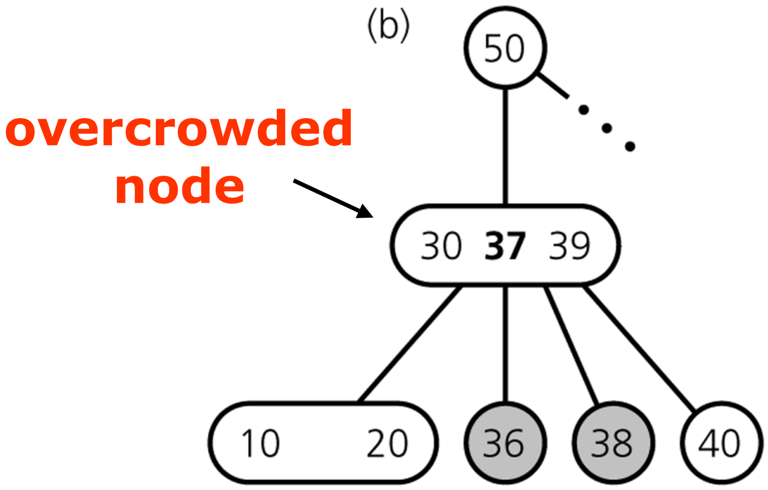
Inserting Items

Insert 36

insert in leaf



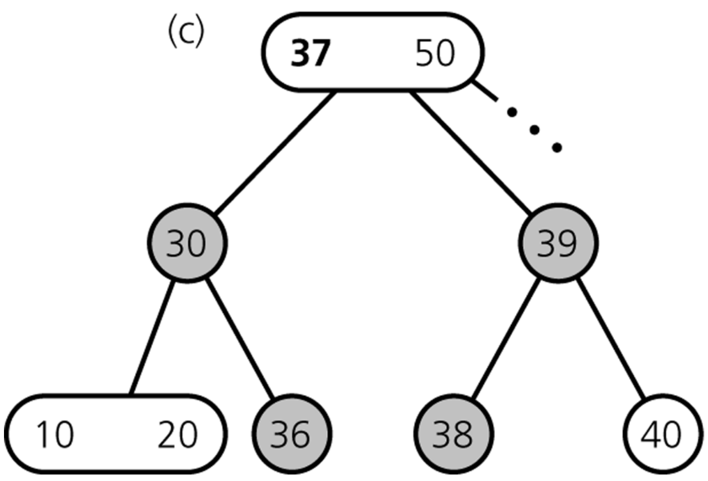
divide leaf
and move middle
value up to parent



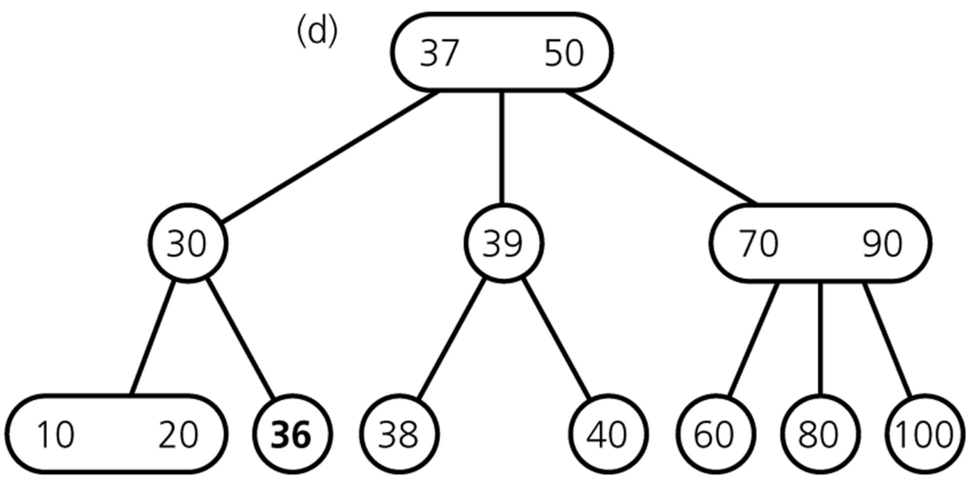
Inserting Items

… still inserting 36

divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

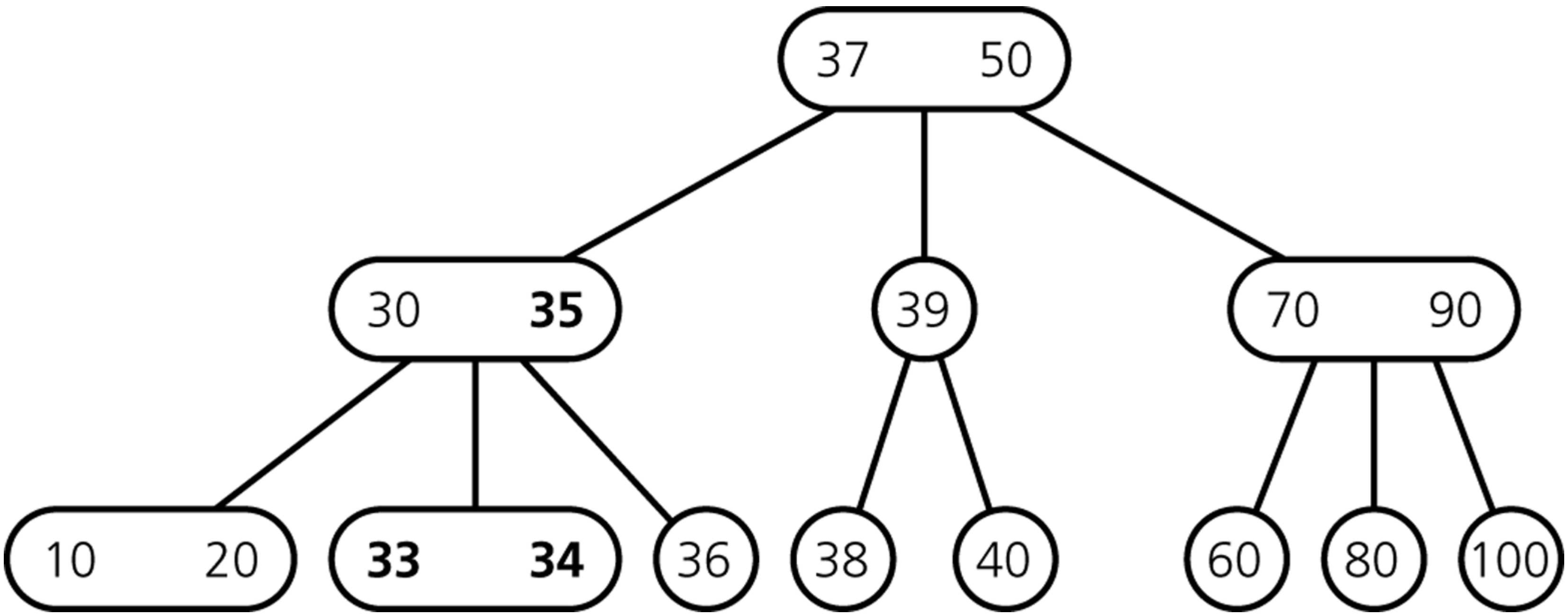


result

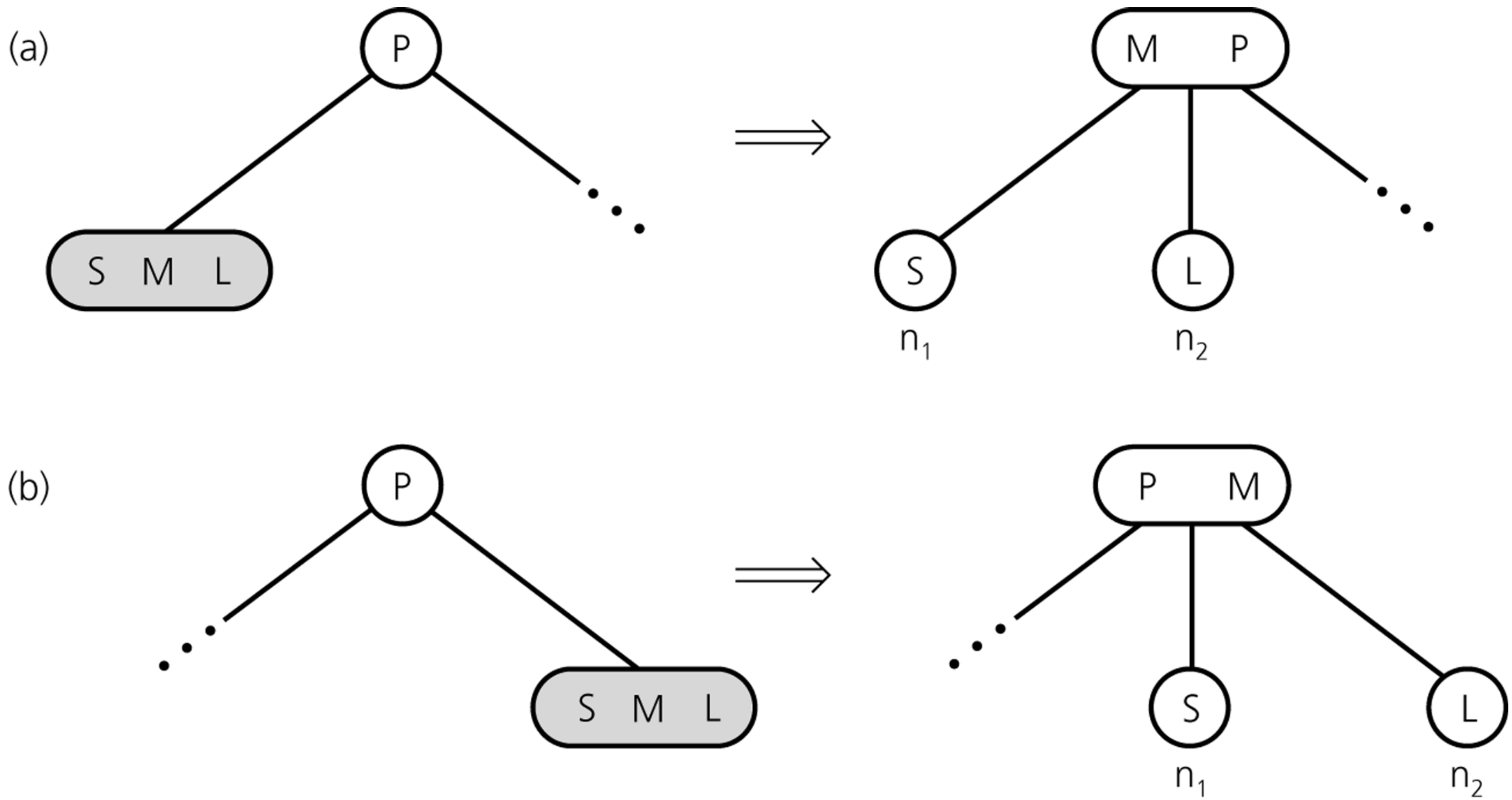


Inserting Items

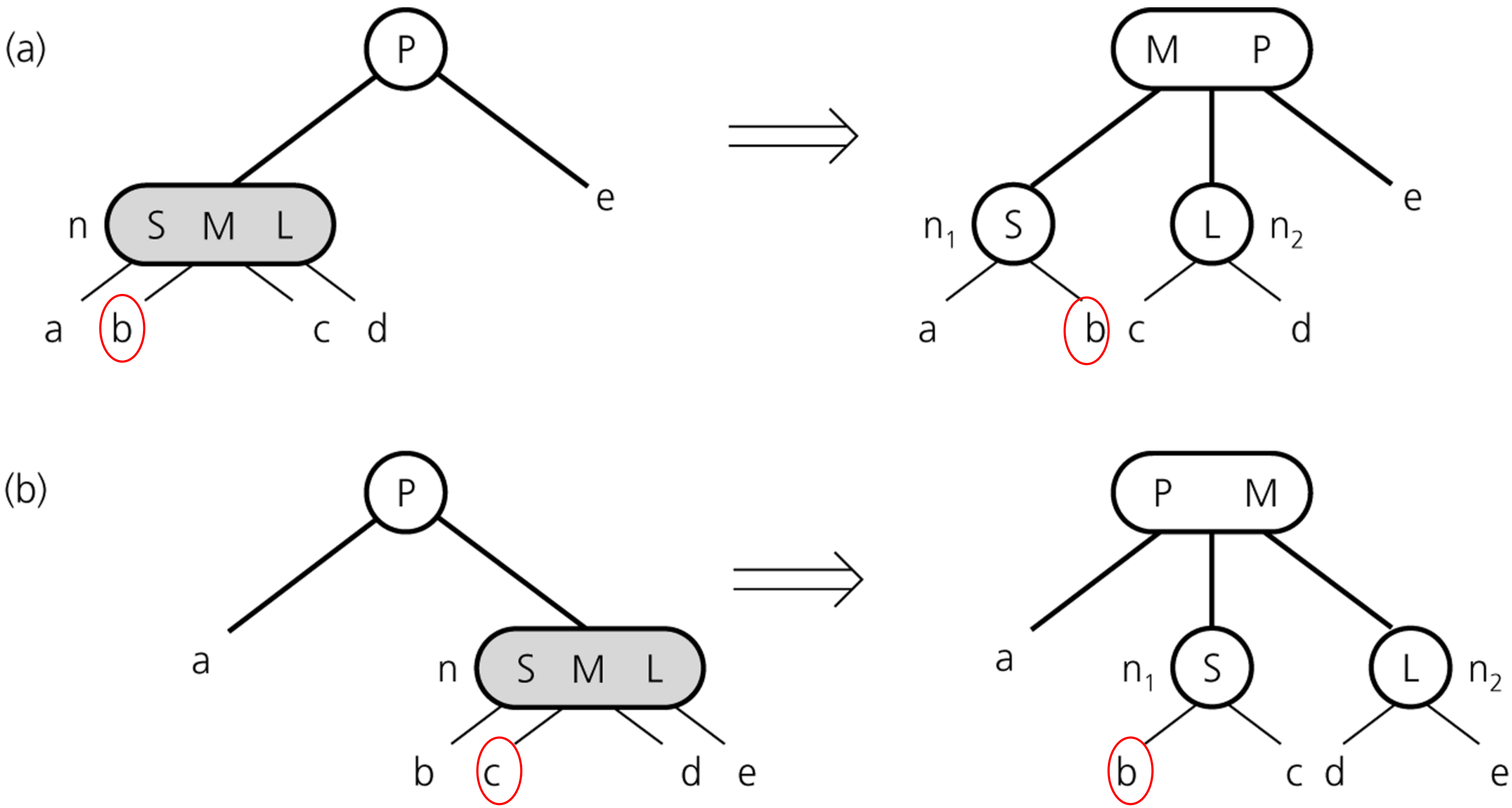
After Insertion of 35, 34, 33



Inserting so far

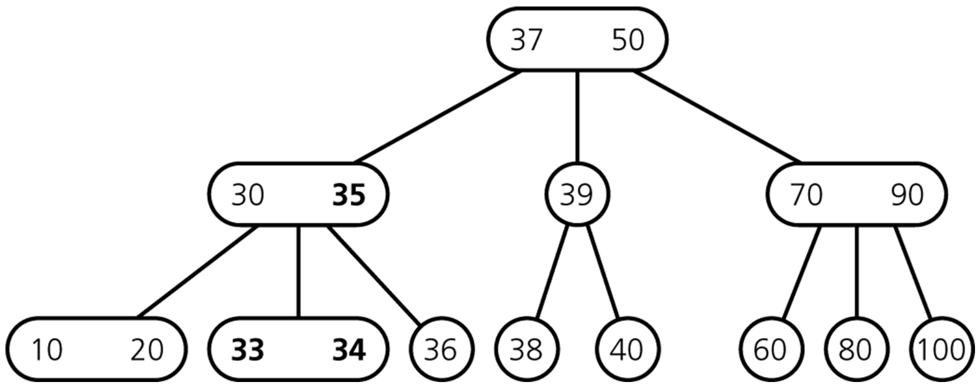


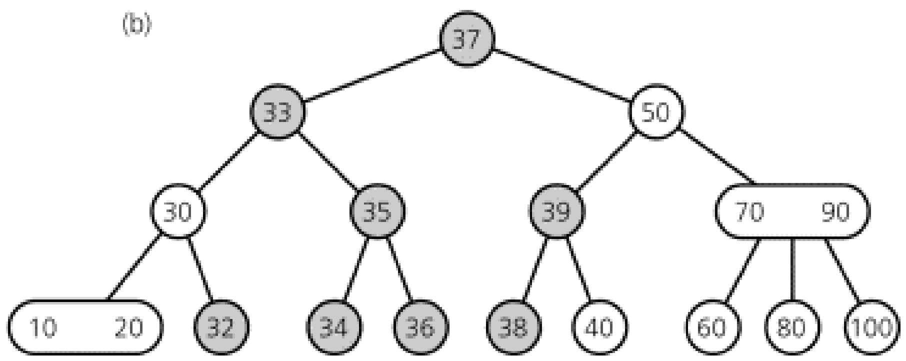
Inserting so far



Inserting Items

How do we insert 32?

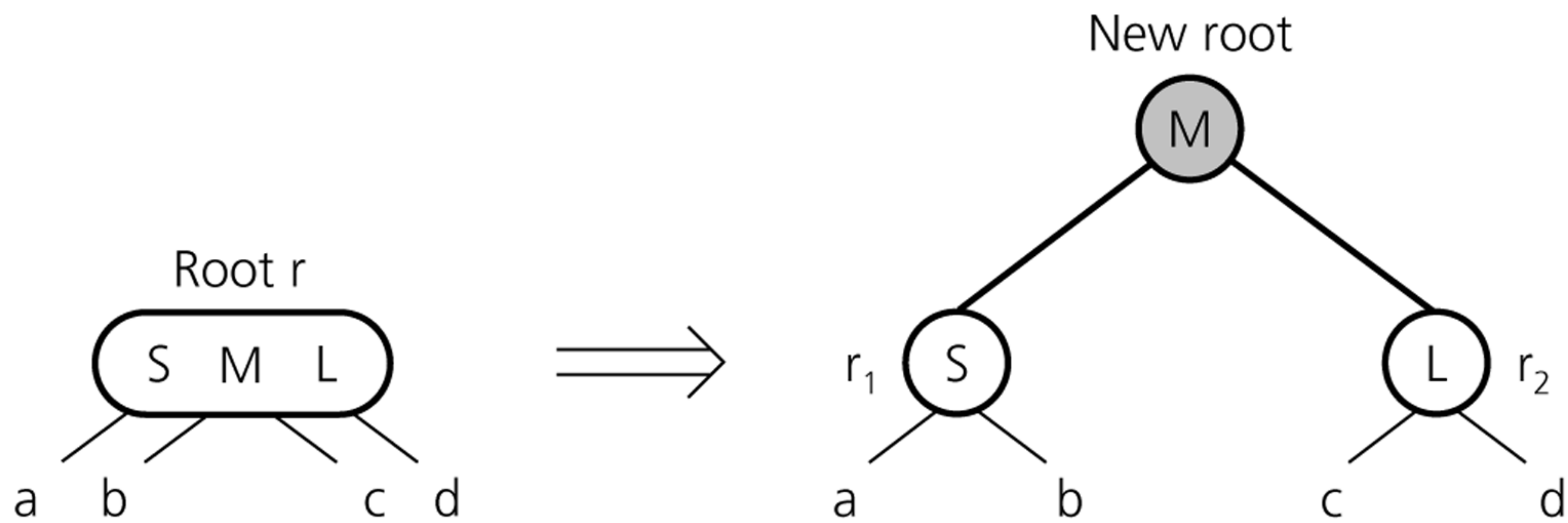




Backward Split(후진분할) with one insertion. Bad. Takes time.

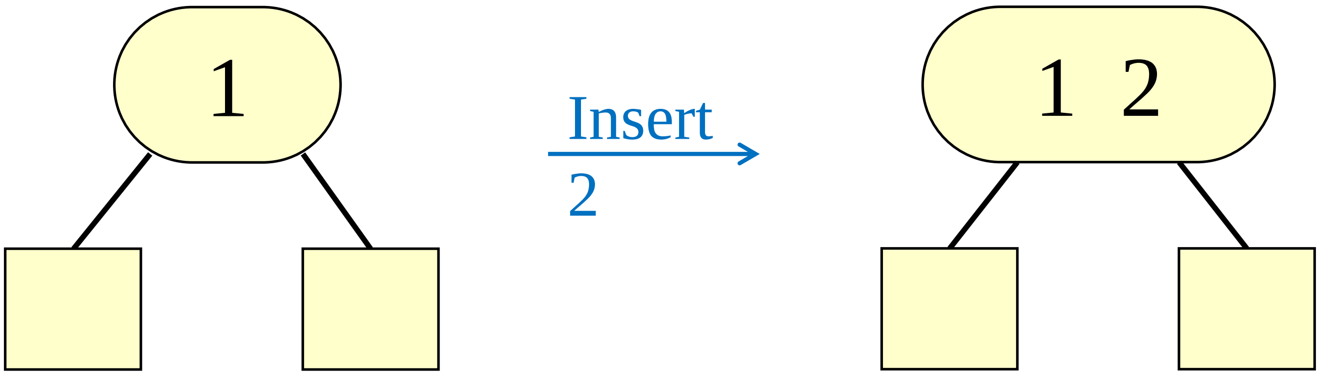
Inserting Items

→ **creating a new root if necessary**
→ **tree grows at the root**



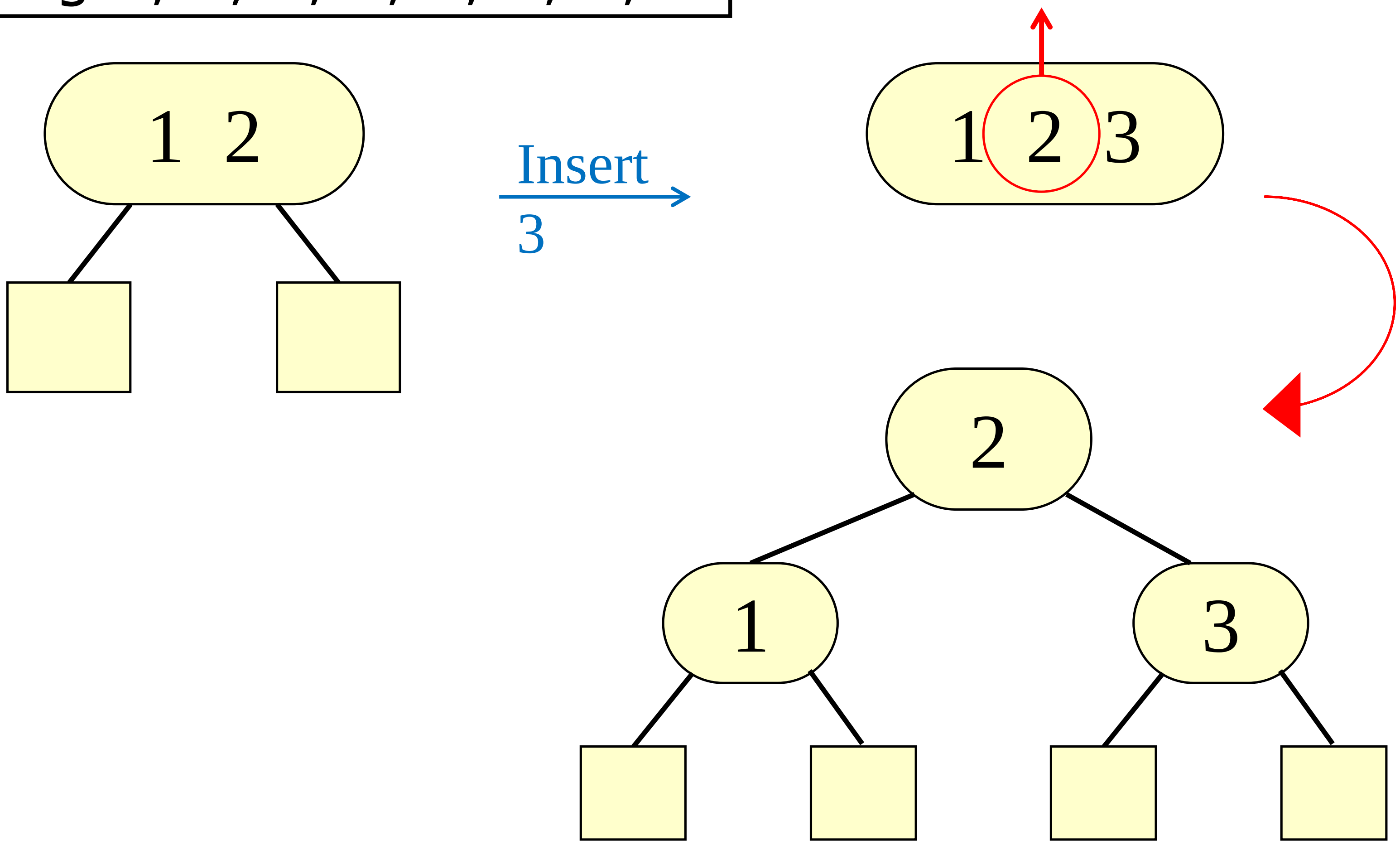
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



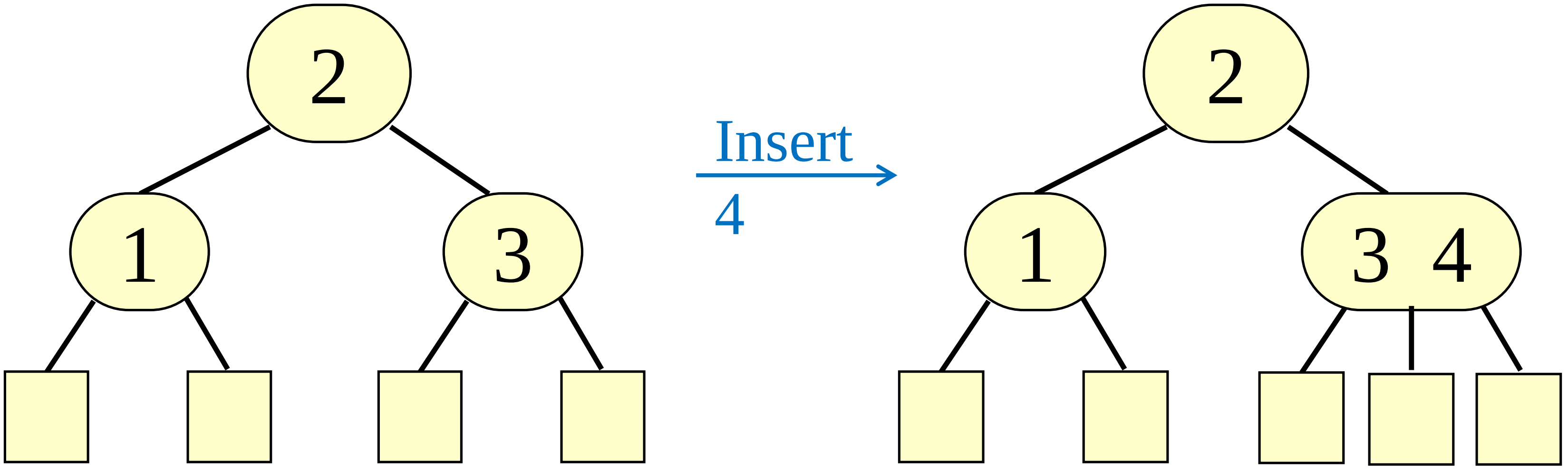
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



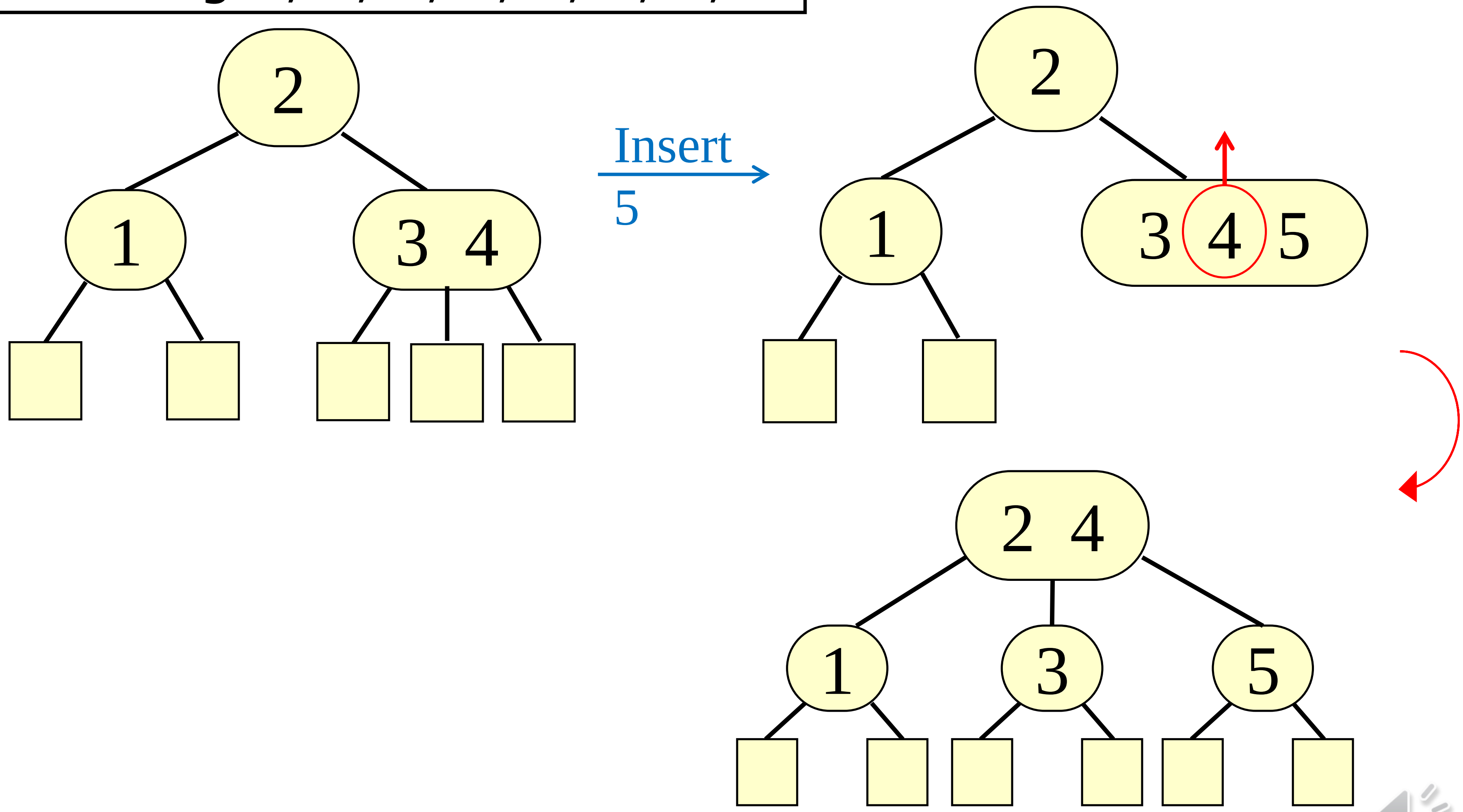
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



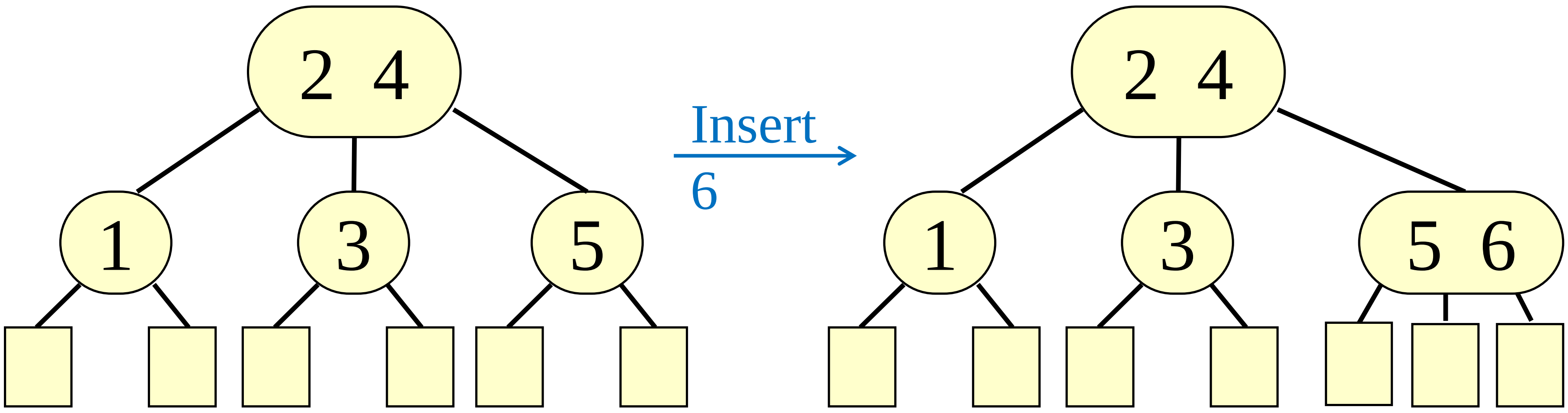
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



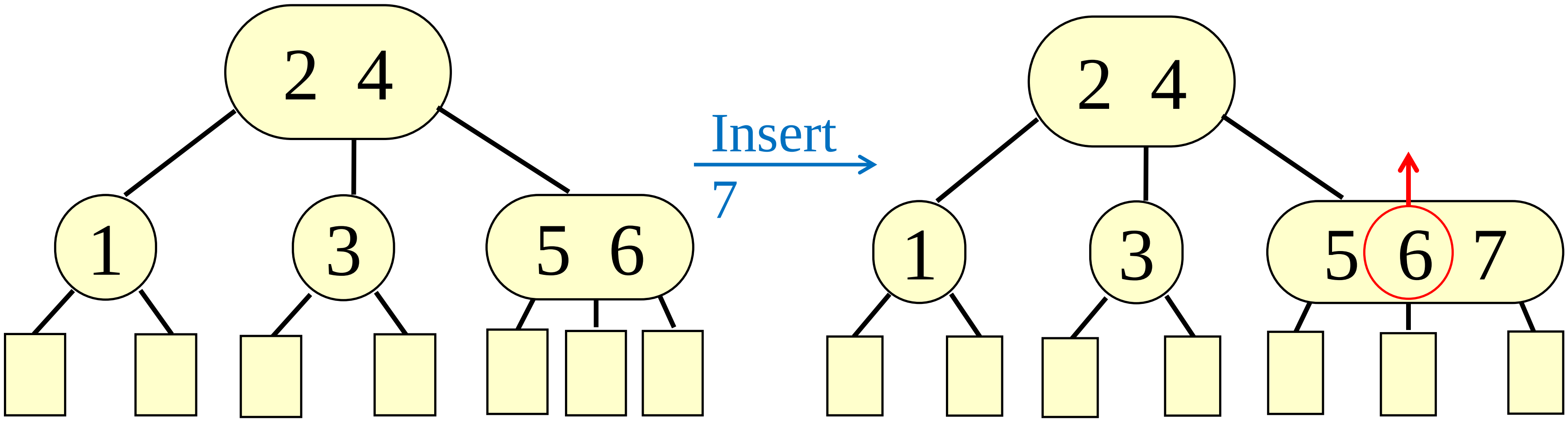
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



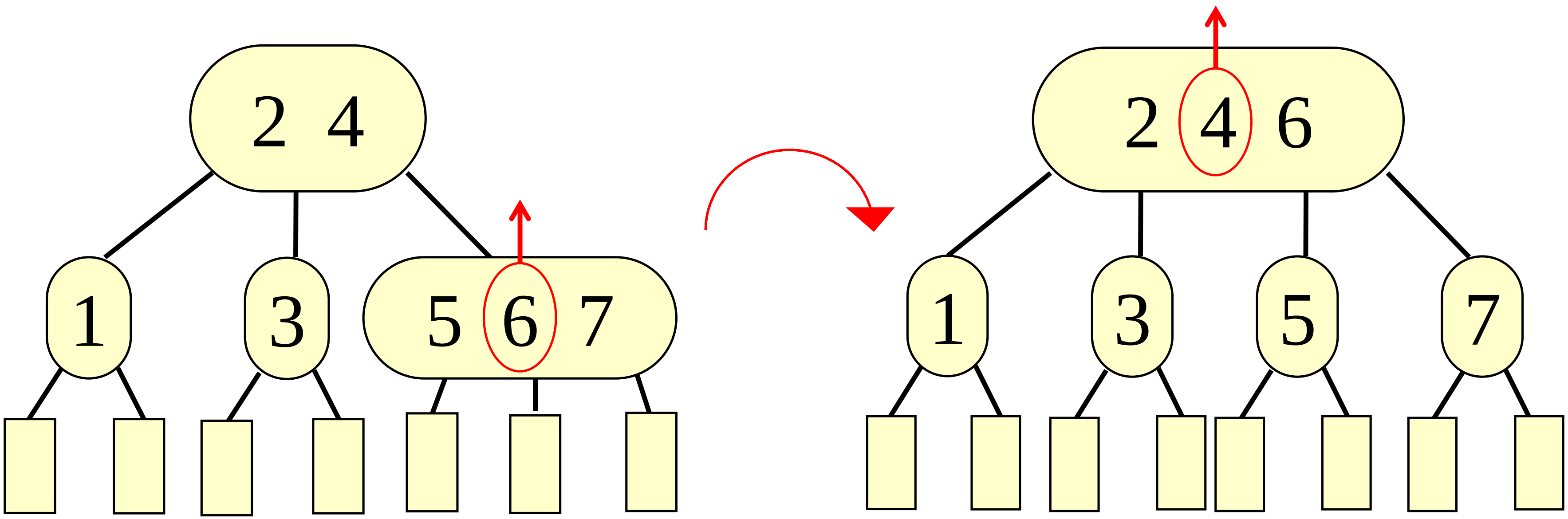
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



2-3 Tree: Insertion

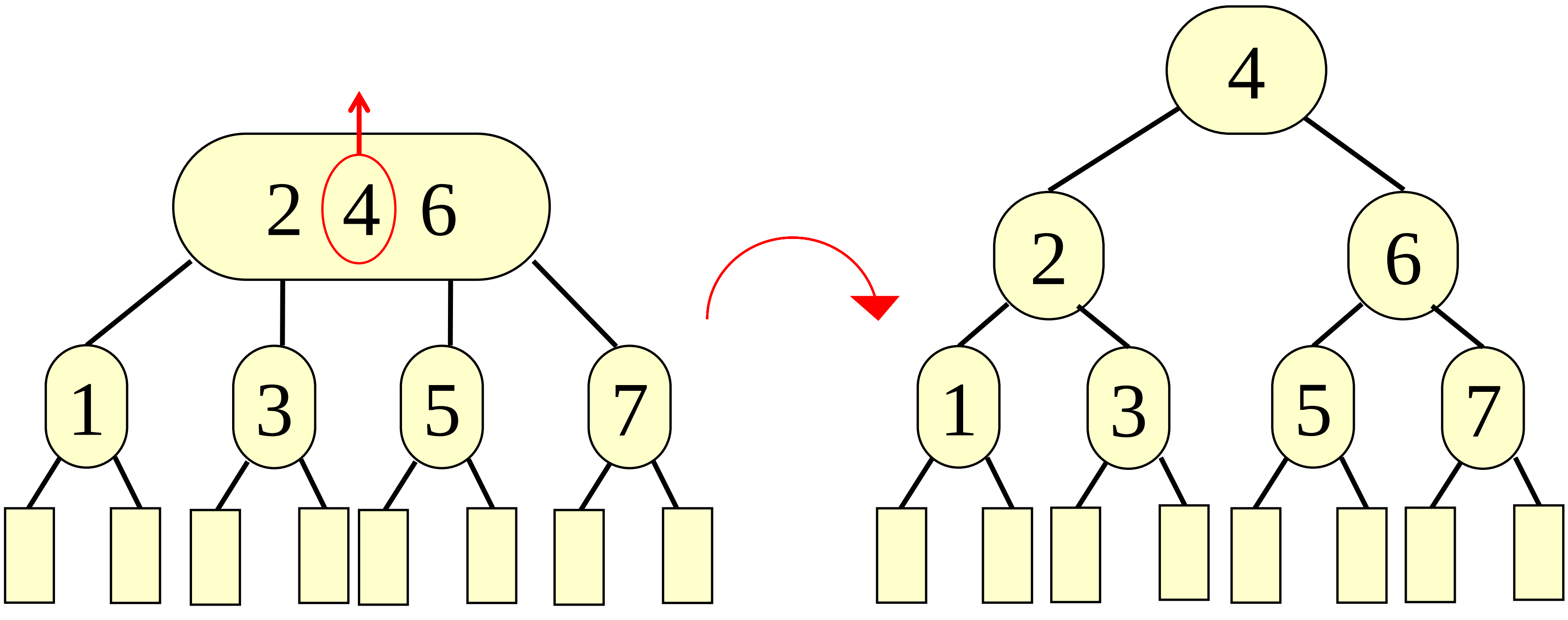
Inserting 1, 2, 3, 4, 5, 6, 7, 8



Backward Split with one insertion. Bad

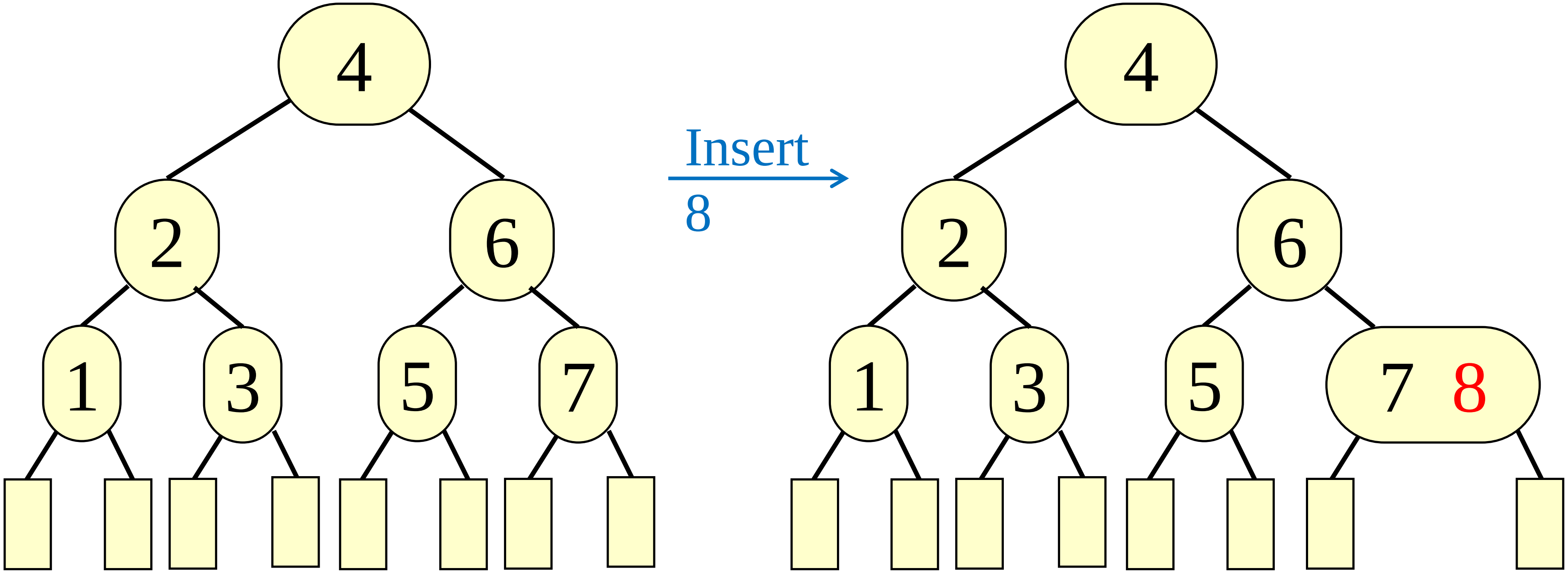
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8



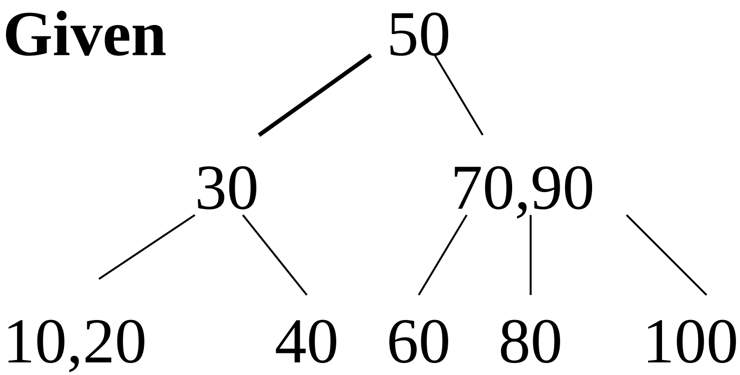
2-3 Tree: Insertion

Inserting 1, 2, 3, 4, 5, 6, 7, 8

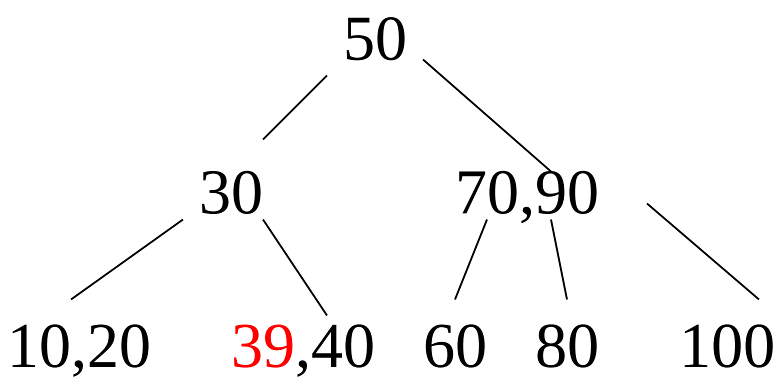


Insertion

Given

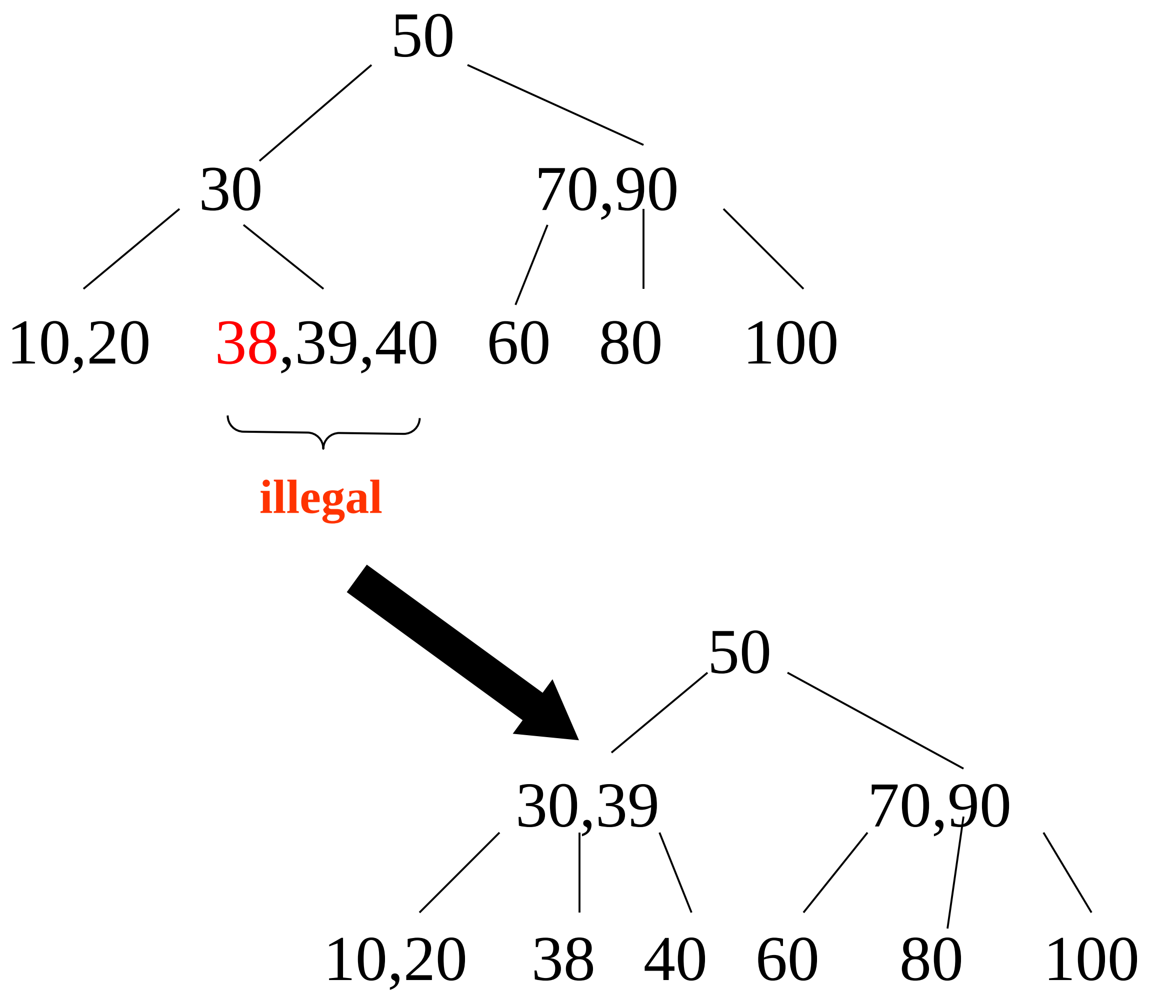


Insert 39

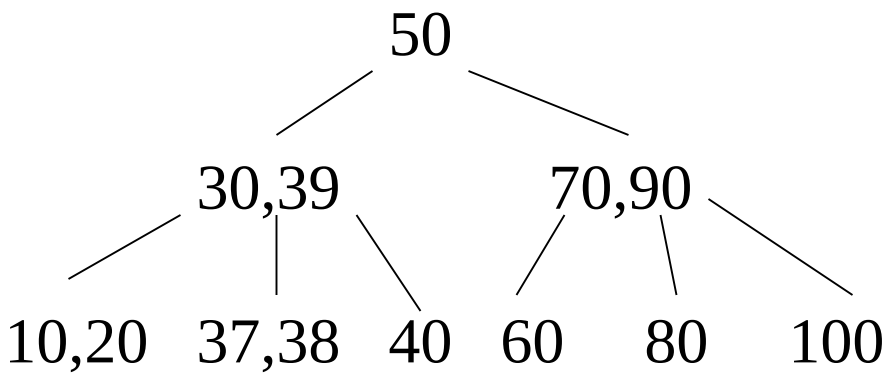


Insertions are always at a leaf

Insert 38

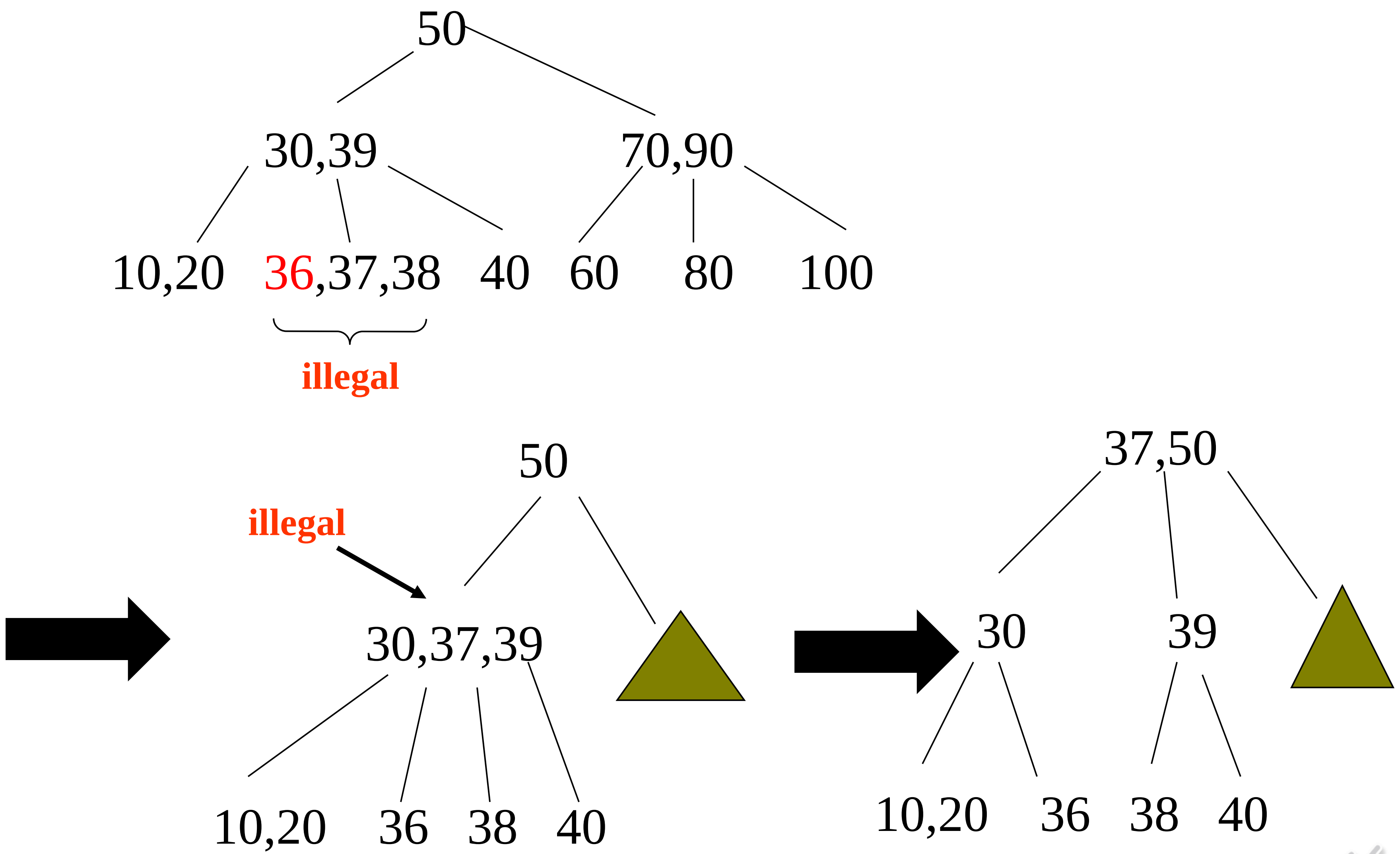

50
30
70,90
10,20
38,39,40
60
80
100
illegal
50
30,39
70,90
10,20
38
40
60
80
100

Insert 37



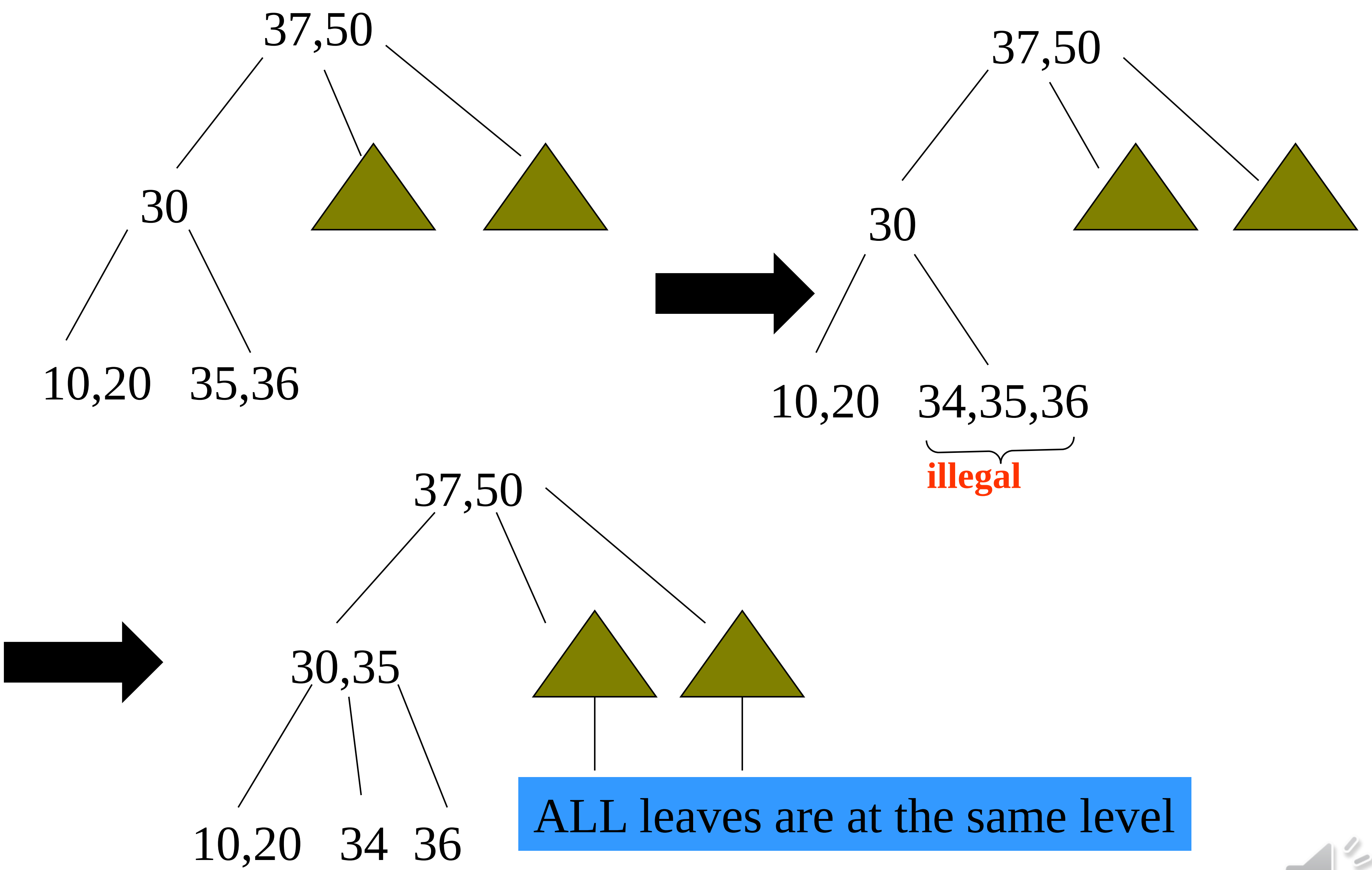
When the height grows it does so from the top.

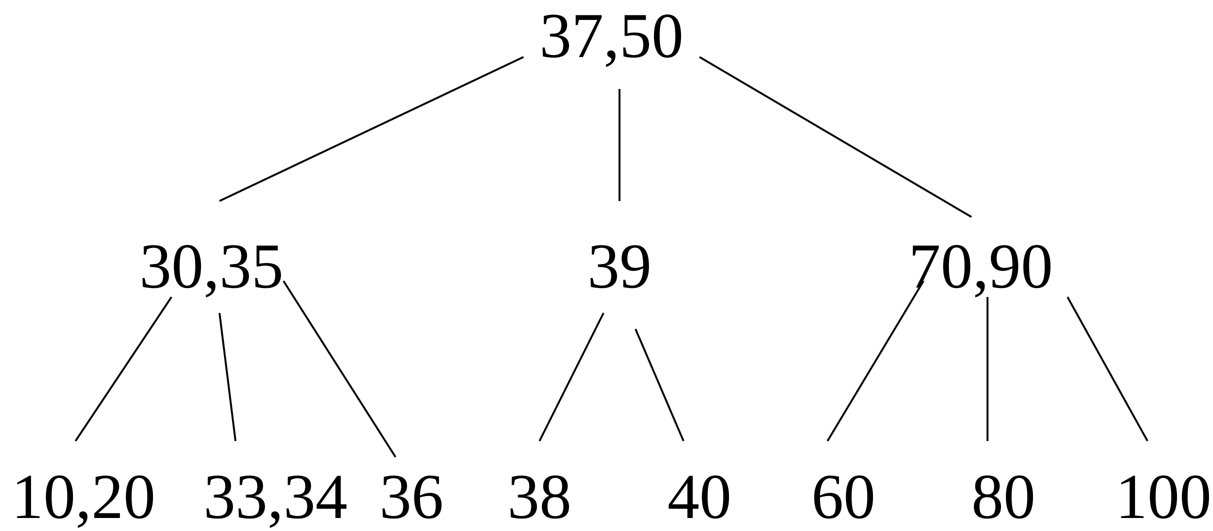
Insert 36



50
30,39
70,90
10,20
36,37,38
40
60
80
100
illegal
50
illegal
30,37,39
10,20
36
38
40
37,50
30
39
10,20
36
38
40

Insert 35, 34, 33



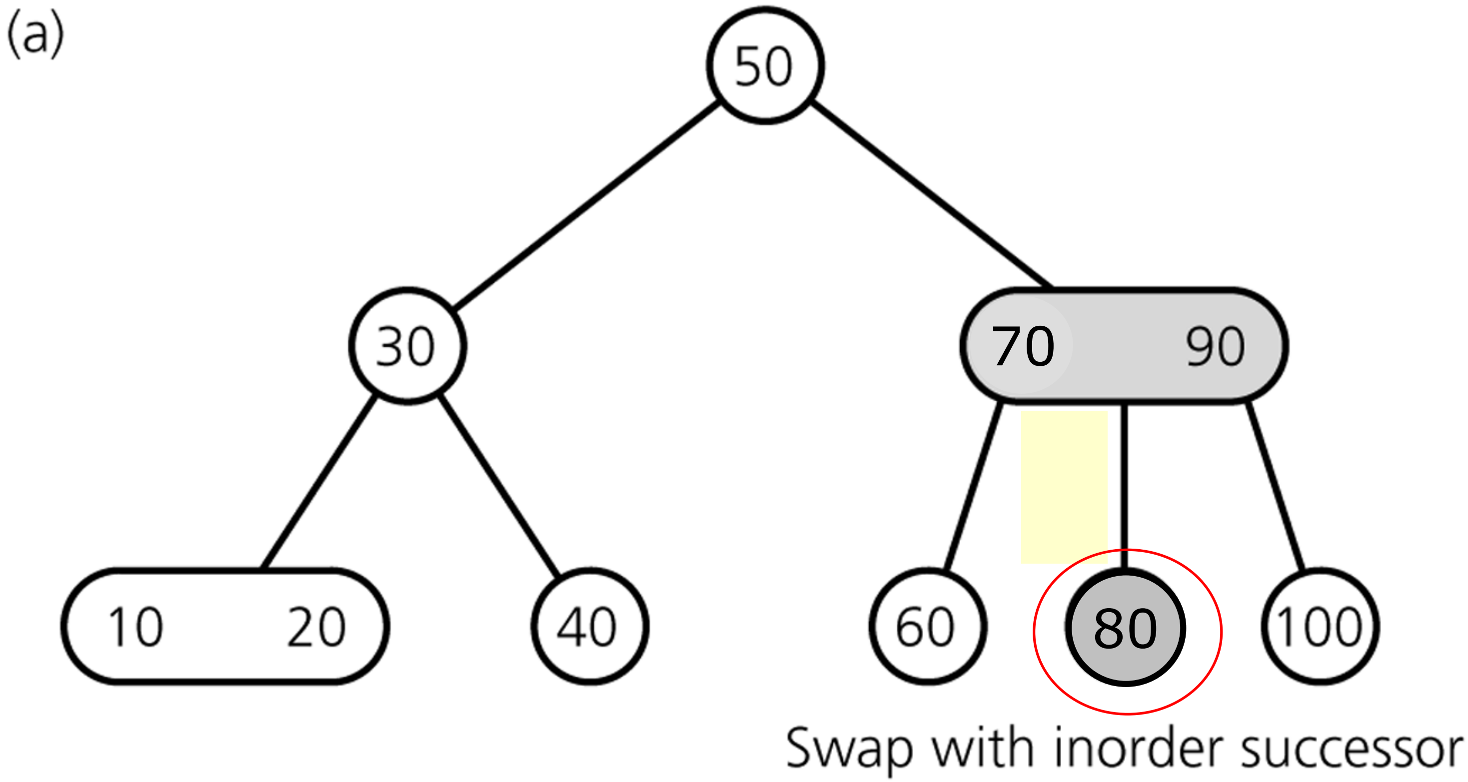

37,50
30,35
39
70,90
10,20
33,34
36
38
40
60
80
100

2-3 Tree: Deletion

We do not discuss deletion due to limited time.

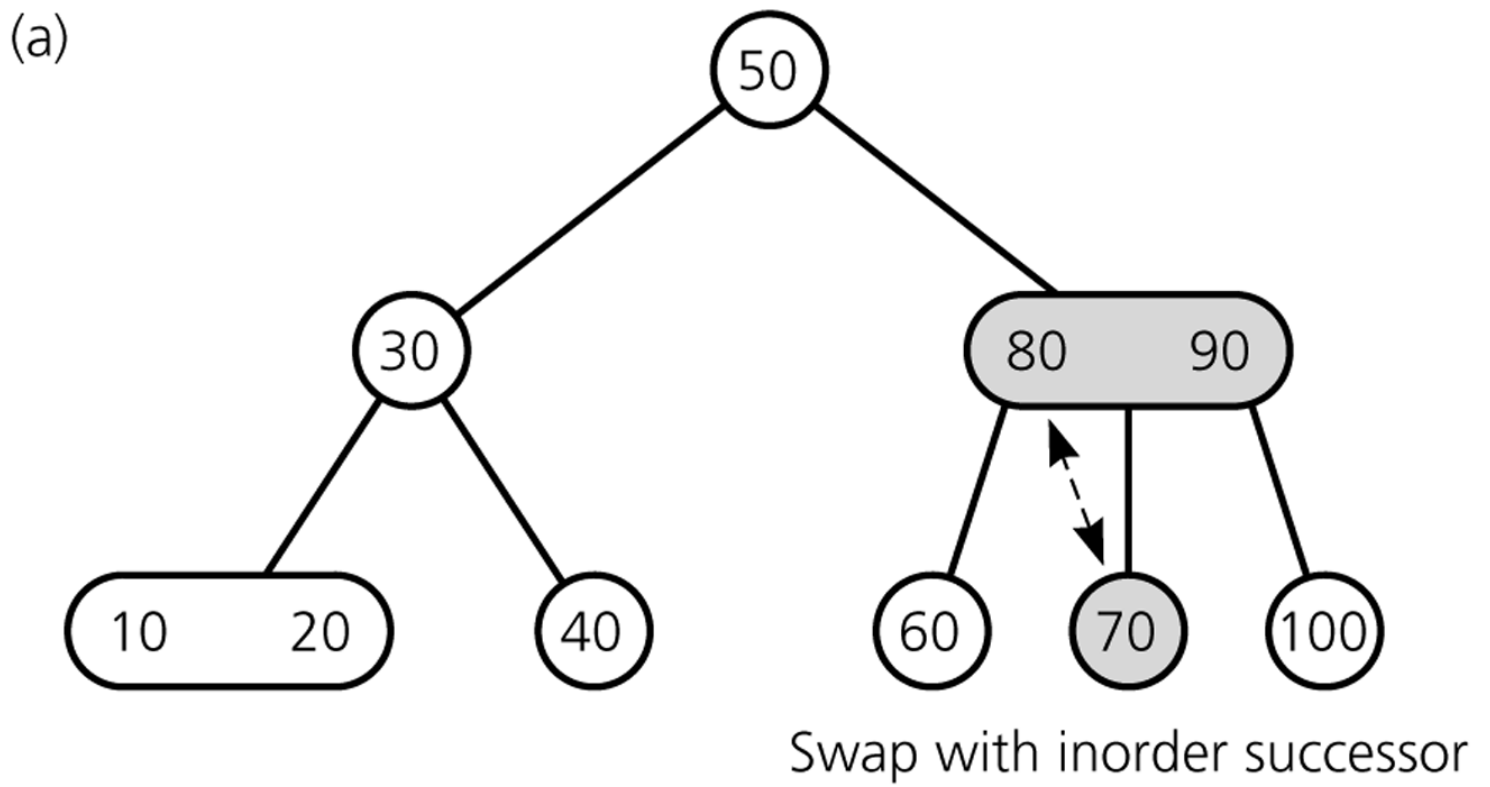
Deleting Items

Delete 70



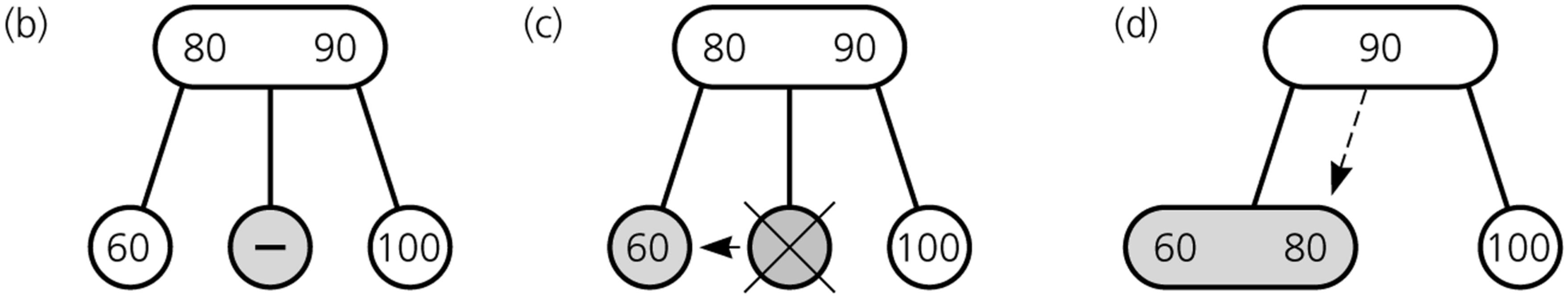
Deleting Items

Deleting 70: swap 70 with inorder successor (80)



Deleting Items

Deleting 70: ... get rid of 70

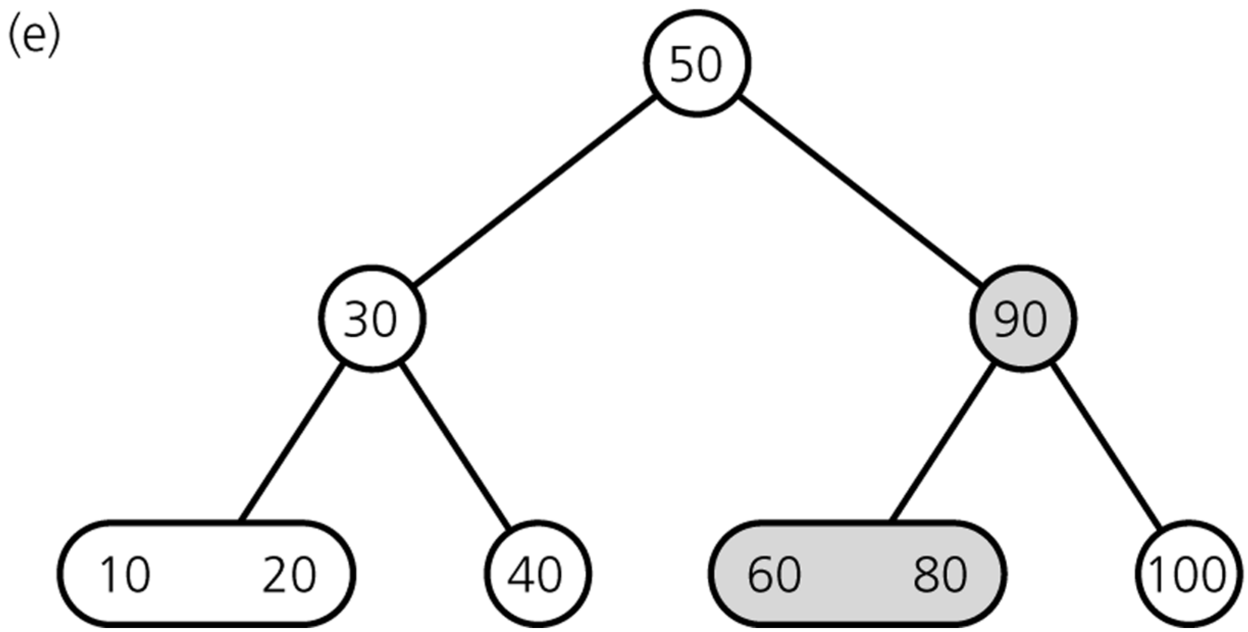


Delete value from leaf

Merge nodes by deleting empty leaf and moving 80 down

Deleting Items

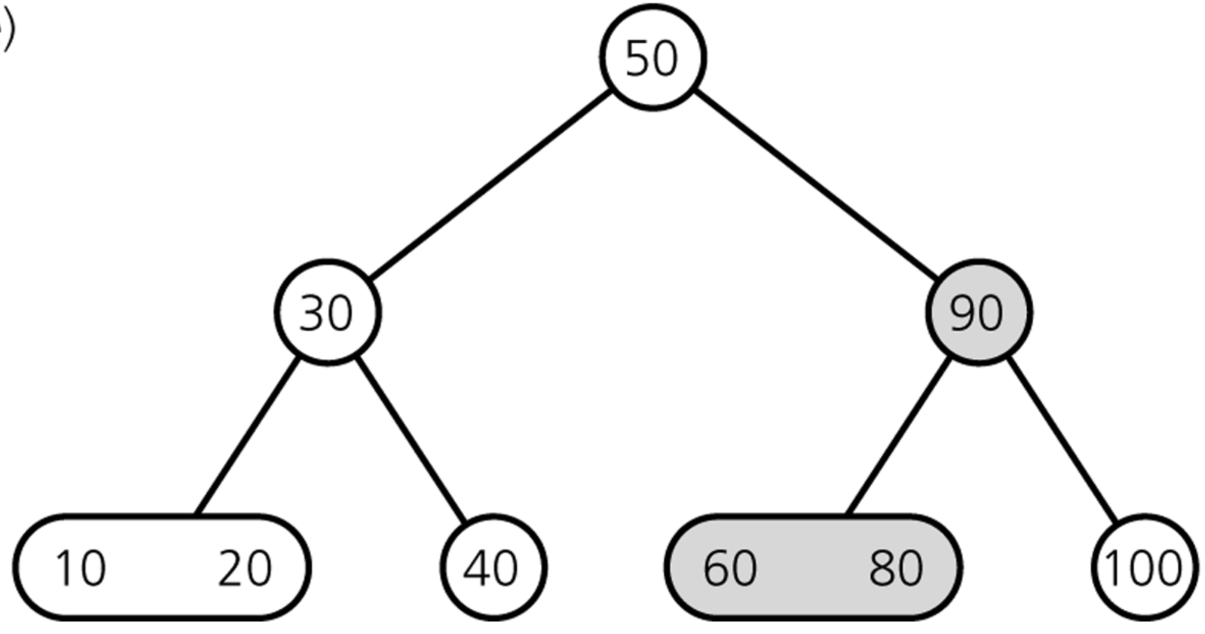
Result



Deleting Items

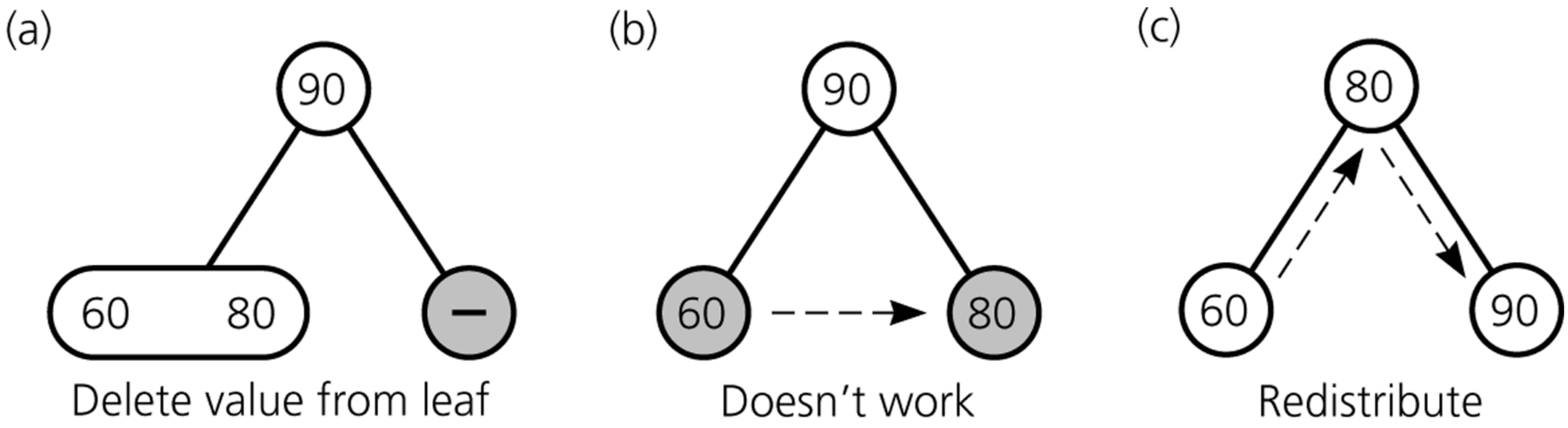
Delete 100

(e)



Deleting Items

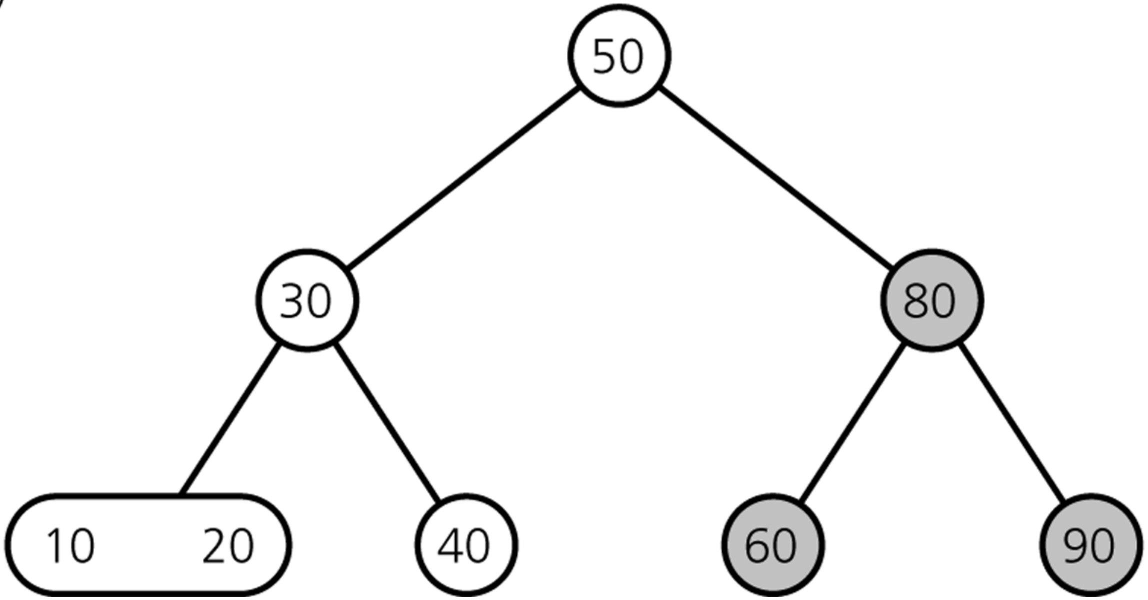
Deleting 100



Deleting Items

Result

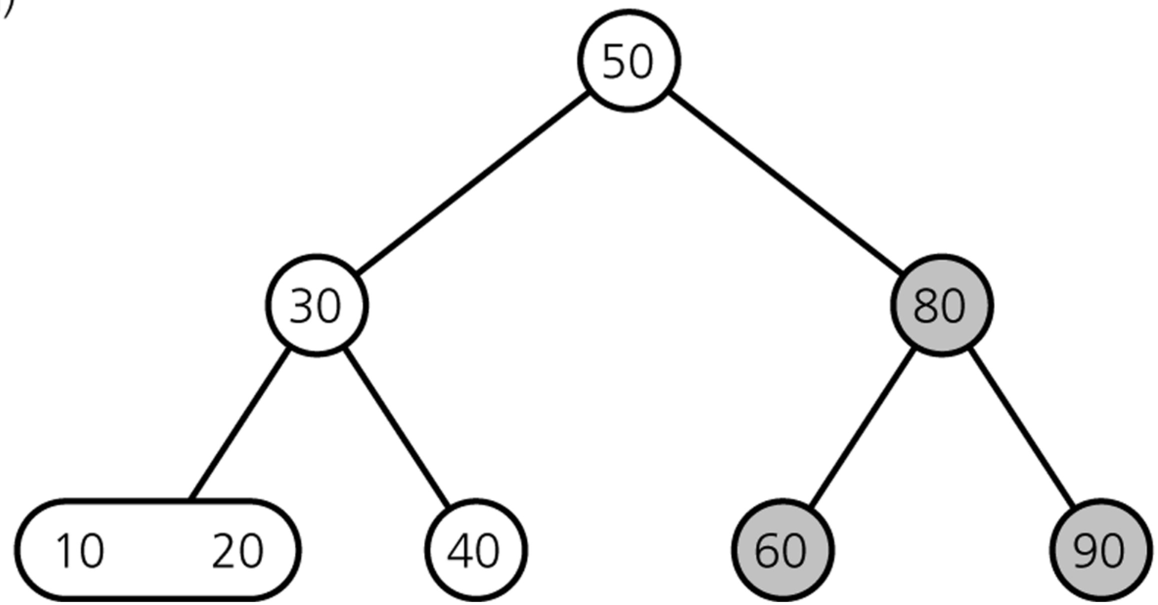
(d)



Deleting Items

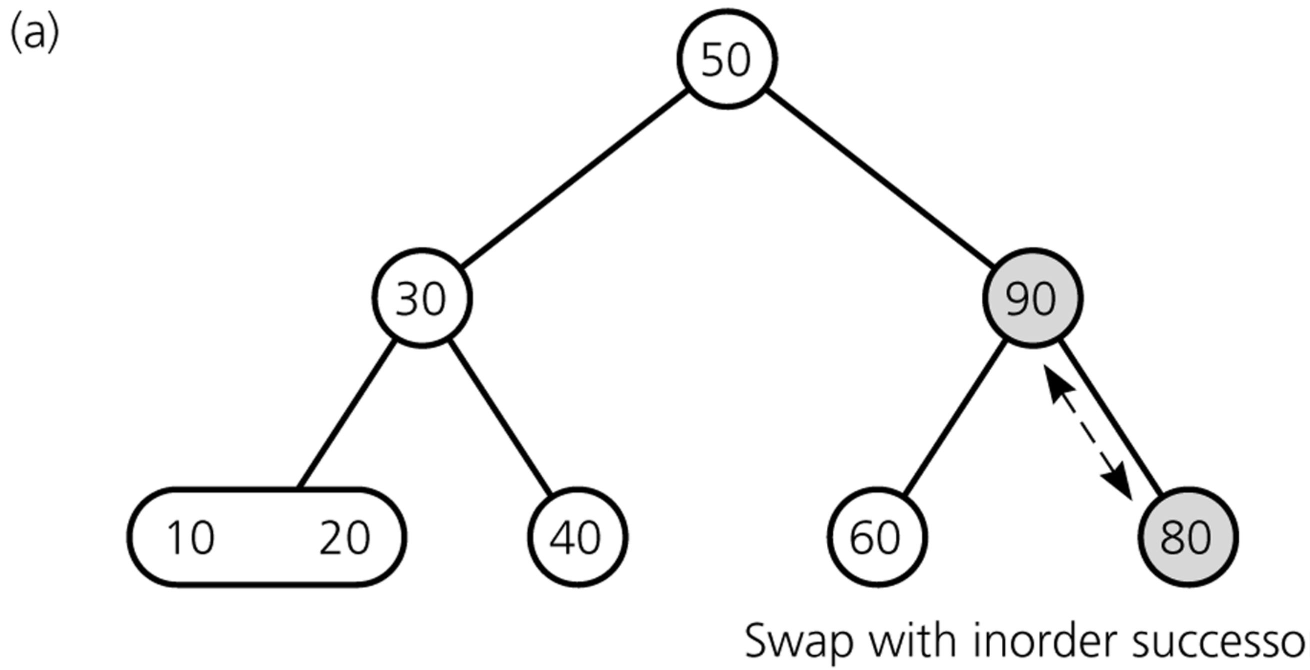
Delete 80

(d)



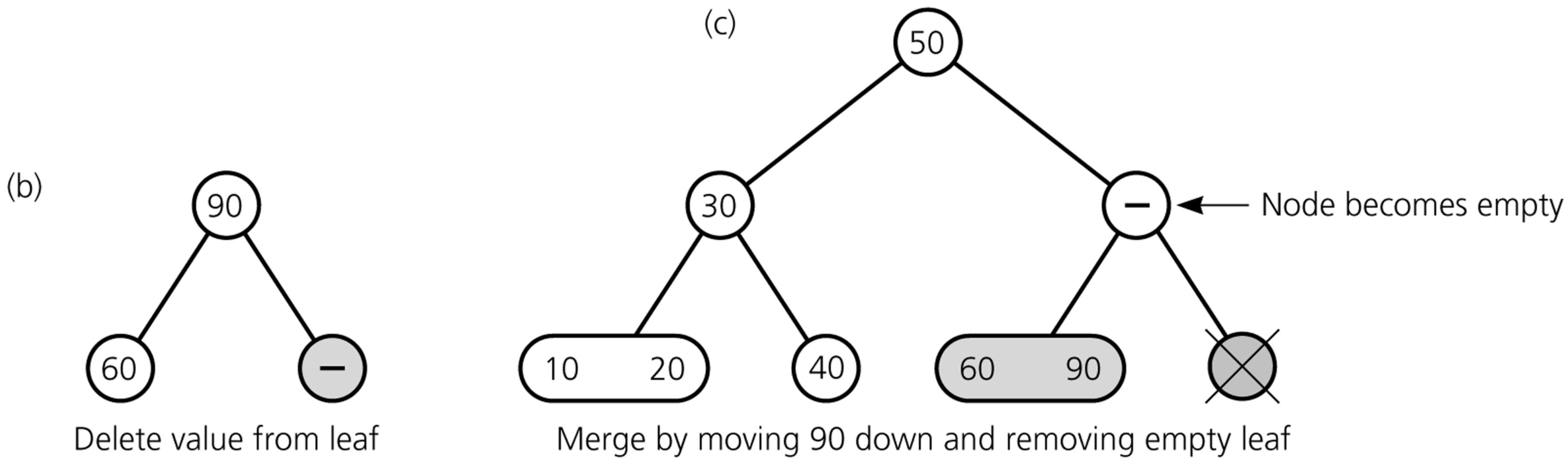
Deleting Items

Deleting 80 ...



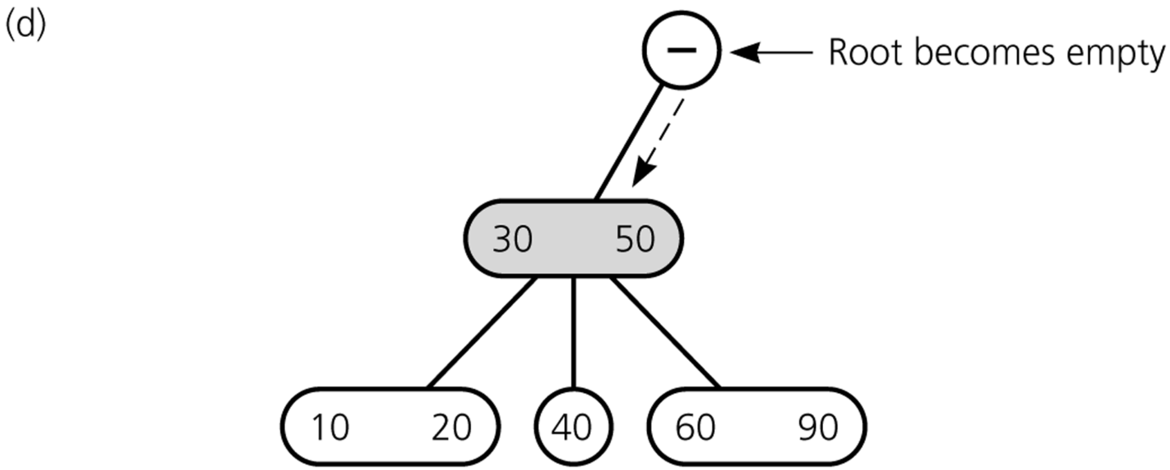
Deleting Items

Deleting 80 ...

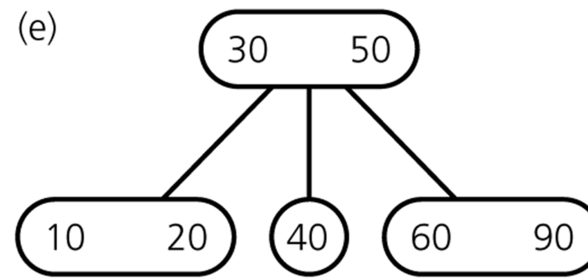


Deleting Items

Deleting 80 ...



Merge: move 50 down, adopt empty leaf's child, remove empty node

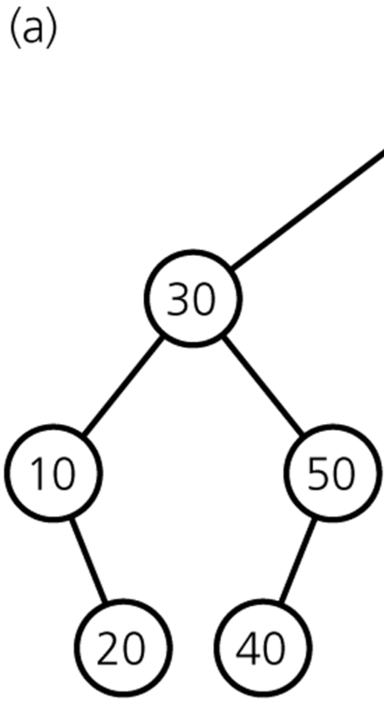


Remove empty root

Deleting Items

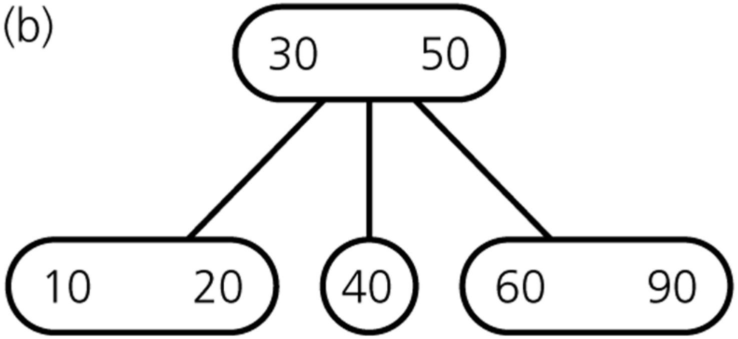
Final Result

(a)



comparison with
binary search tree

(b)



Deletion Algorithm I

Deleting item *I*:

1. Locate node *n*, which contains item *I (may be null if no item)*
2. If node *n* is not a leaf → swap *I* with inorder successor

→ deletion always begins at a leaf

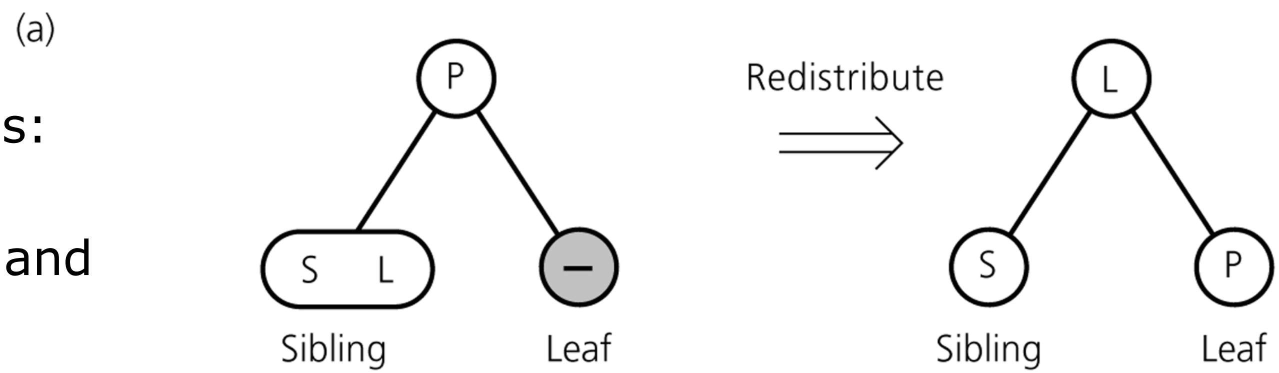
3. If leaf node *n* contains another item, just delete item *I*
   else
   try to redistribute nodes from siblings (see next slide)
   if not possible, merge node (see next slide)

Deletion Algorithm II

Redistribution

A sibling has 2 items:

→ redistribute item between siblings and parent

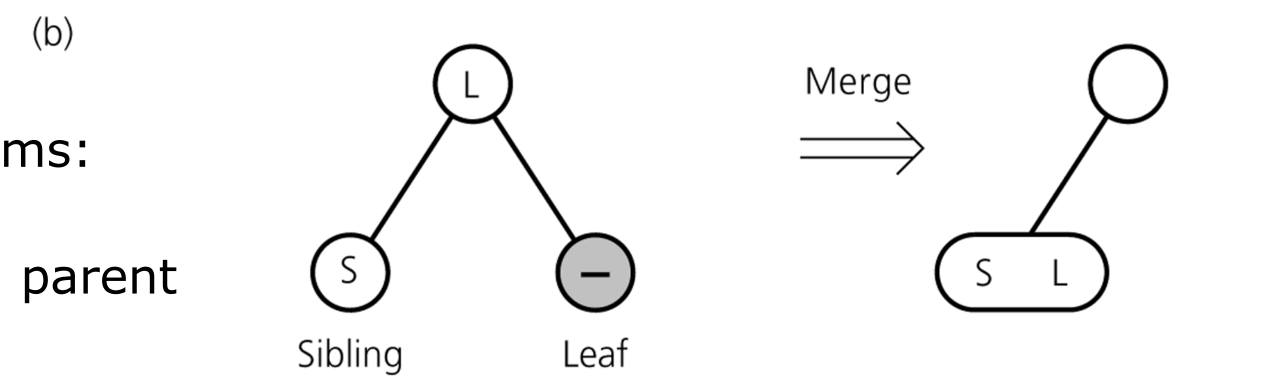


Merging

No sibling has 2 items:

→ merge node

→ move item from parent to sibling

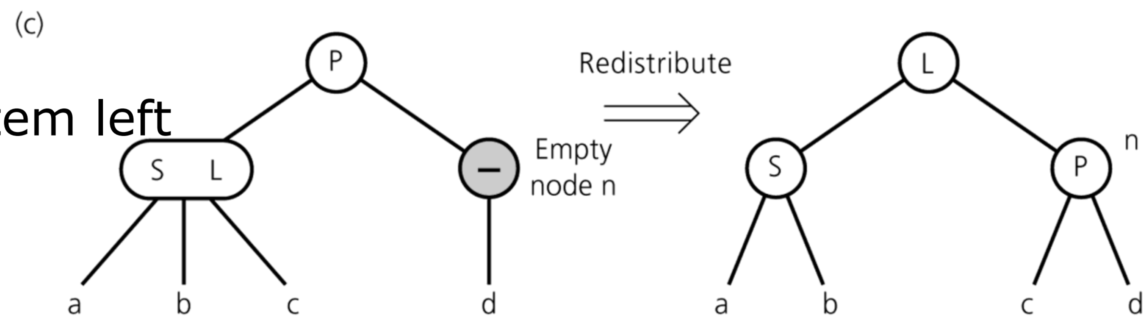


Deletion Algorithm III

Redistribution

Internal node *n* has no item left

→ redistribute



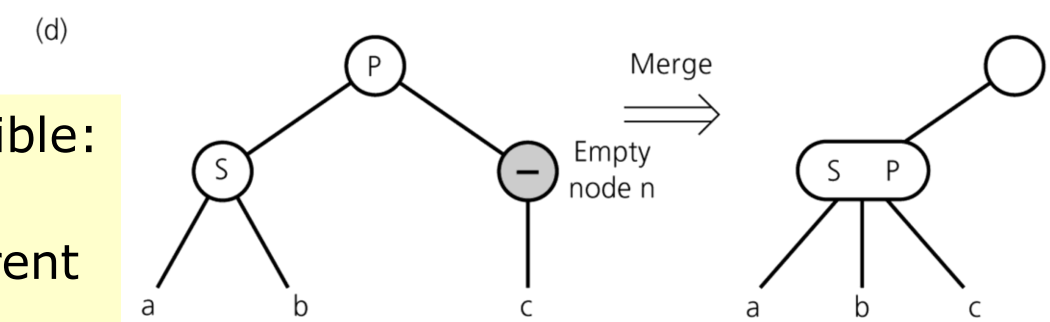
Merging

Redistribution not possible:

→ merge node

→ move item from parent to sibling

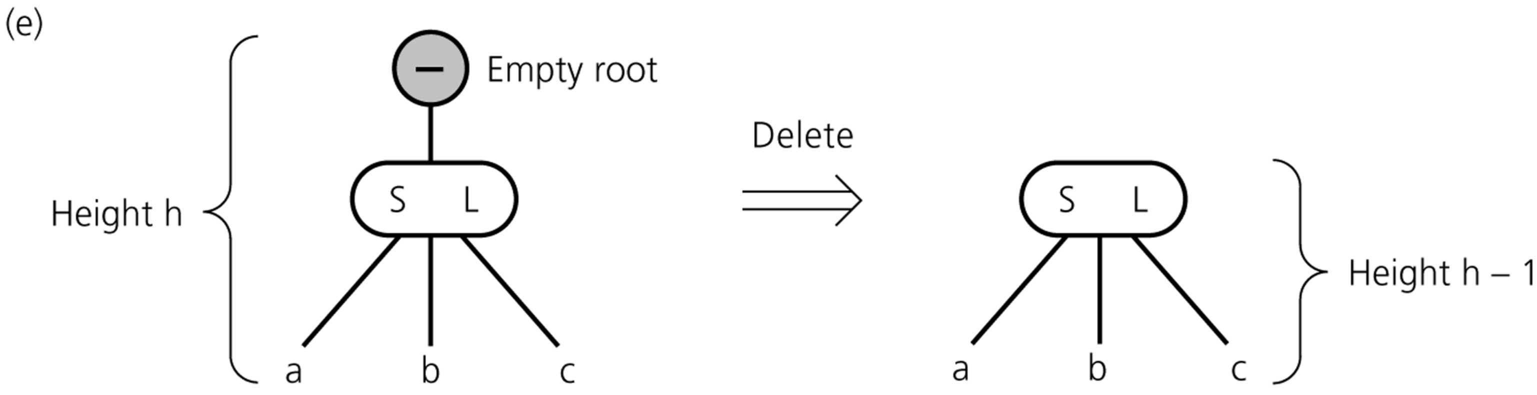
→ adopt child of *n*



If *n*'s parent ends up without item, apply process recursively

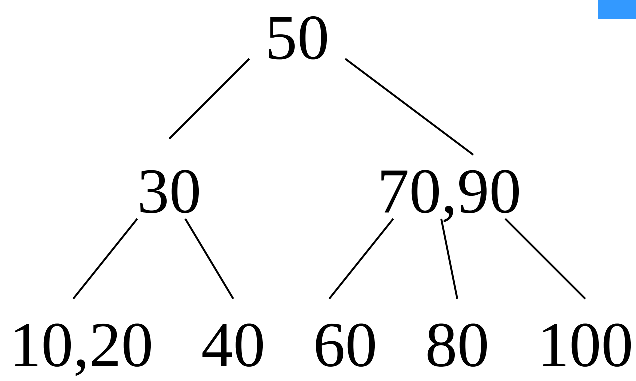
Deletion Algorithm IV

If merging process reaches the root and root is without item
→ delete root

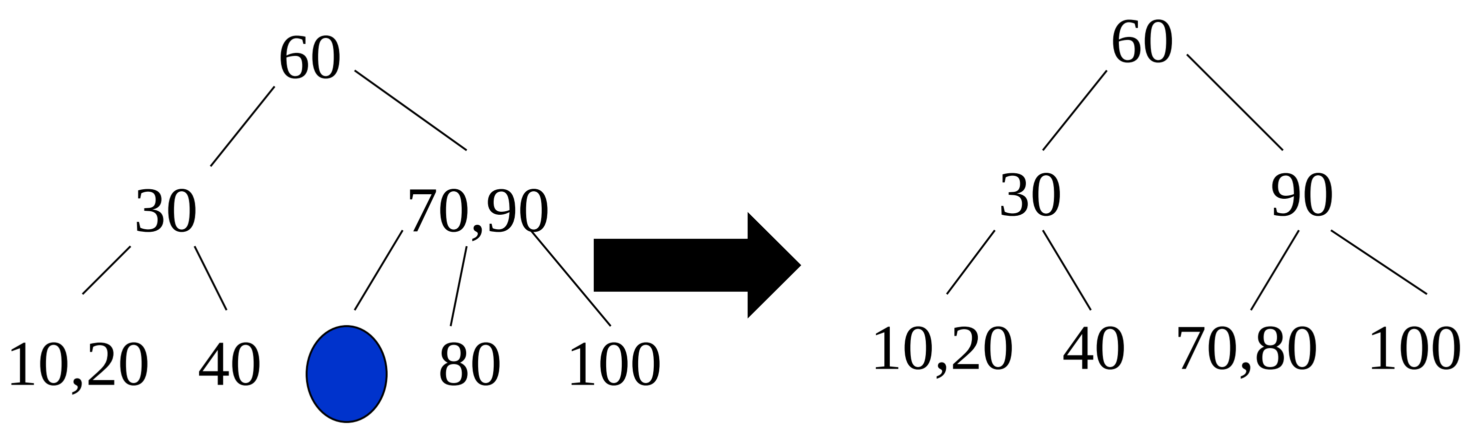


Deletion

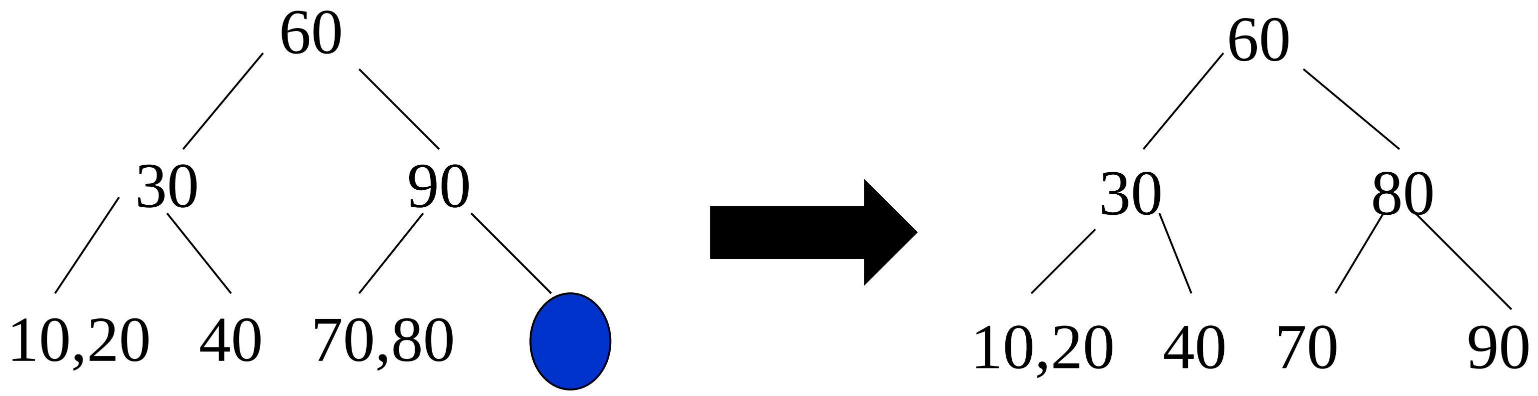
Given



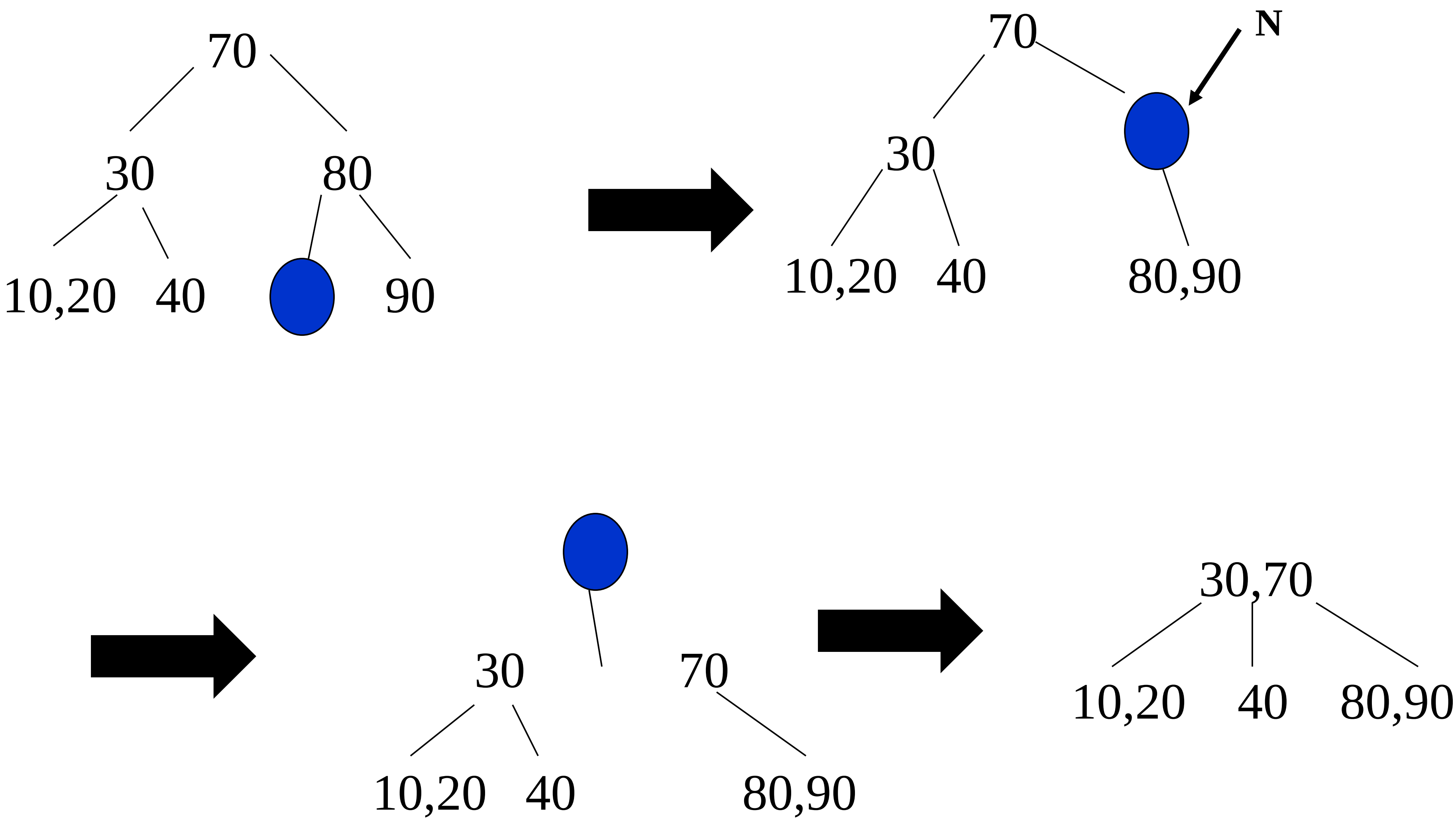
Delete 50



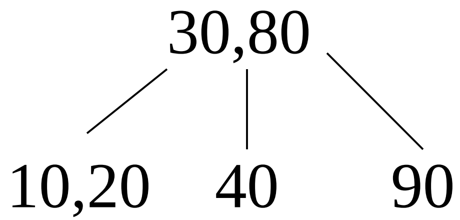
Delete 100



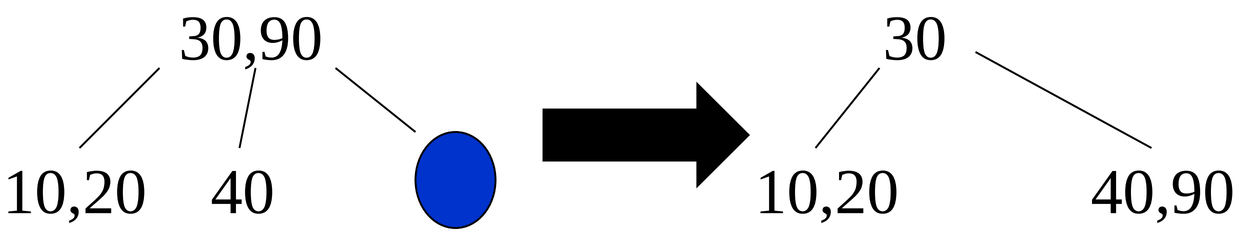
Delete 60


70
30
80
10,20
40
90
70
N
30
10,20
40
80,90
30
70
10,20
40
80,90
30,70
10,20
40
80,90

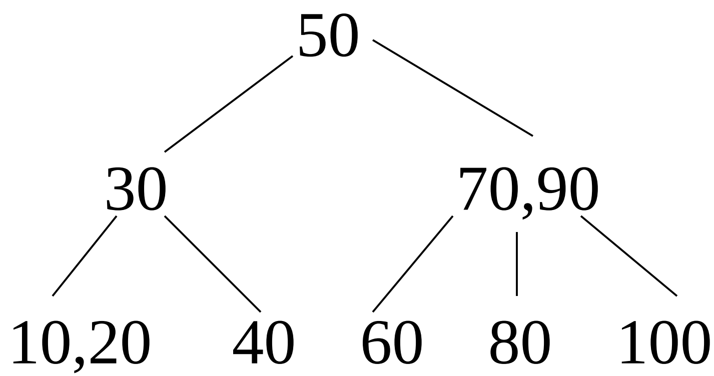
Delete 70



Delete 80

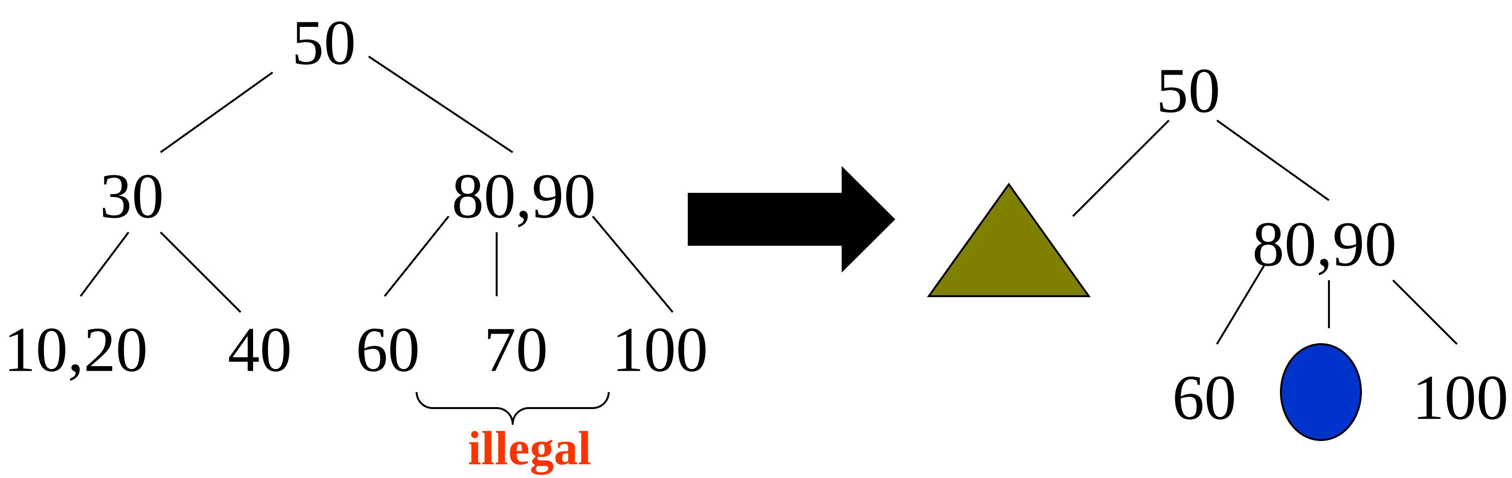


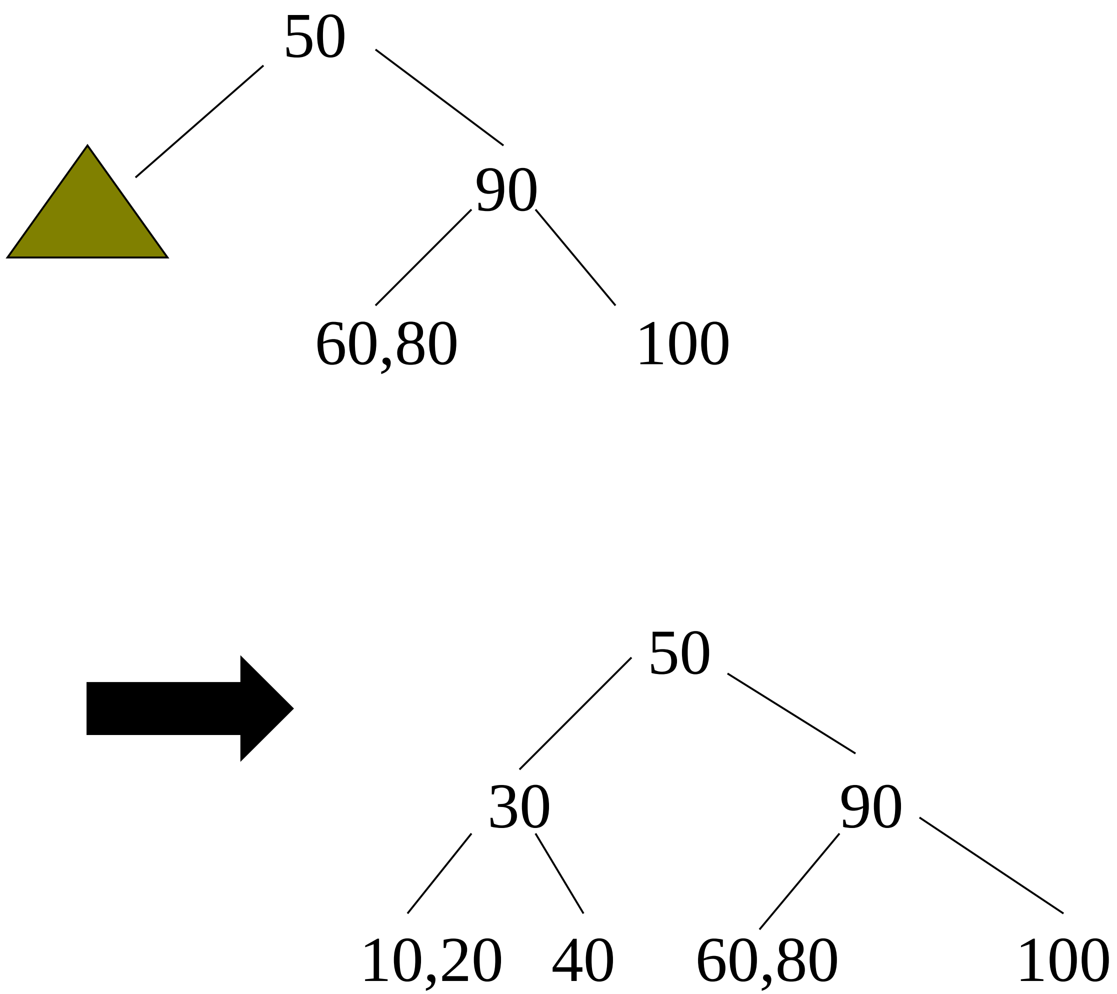
Given



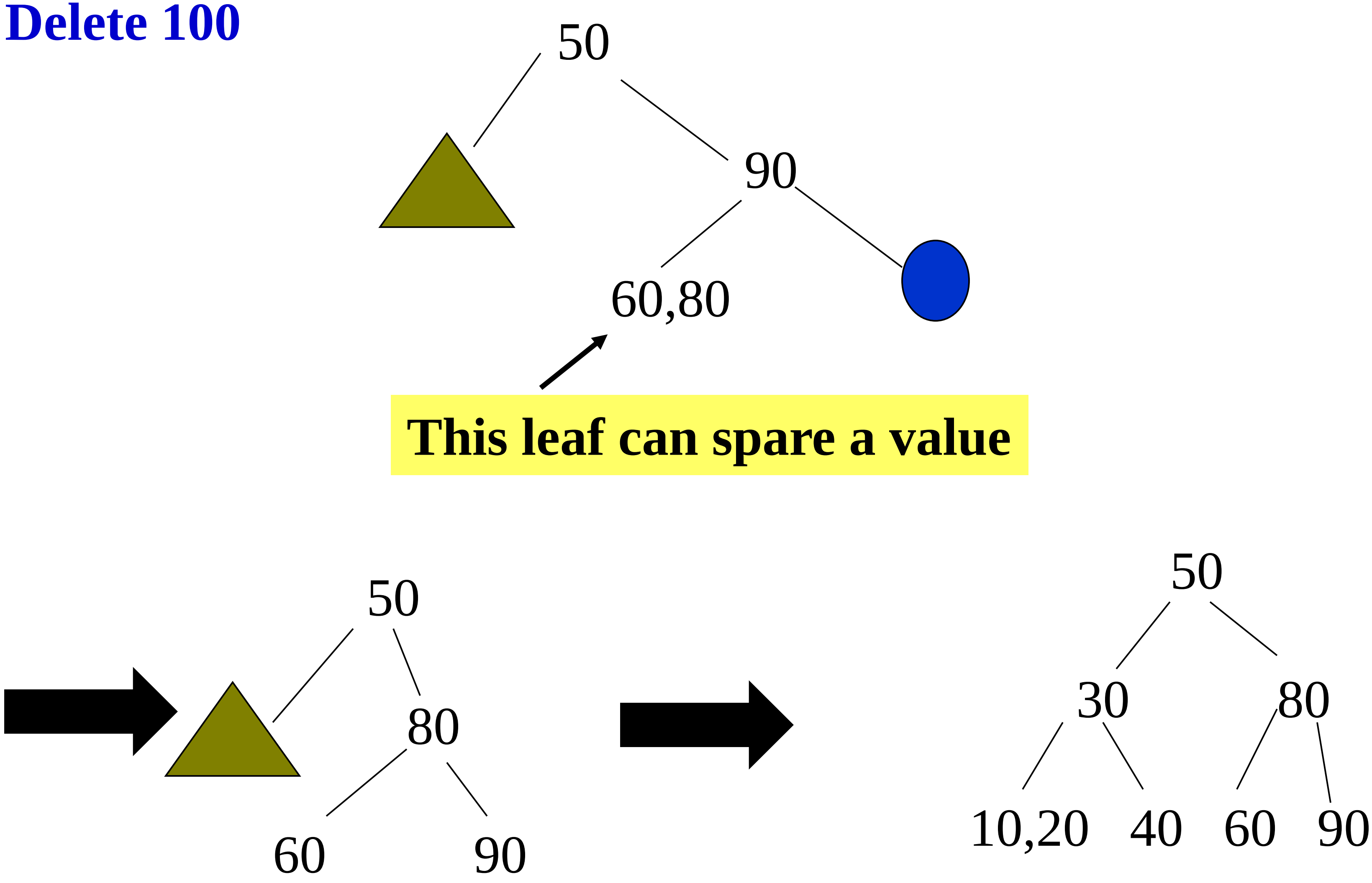
Delete 70

You always begin deletion from a leaf so swap with inorder successor.



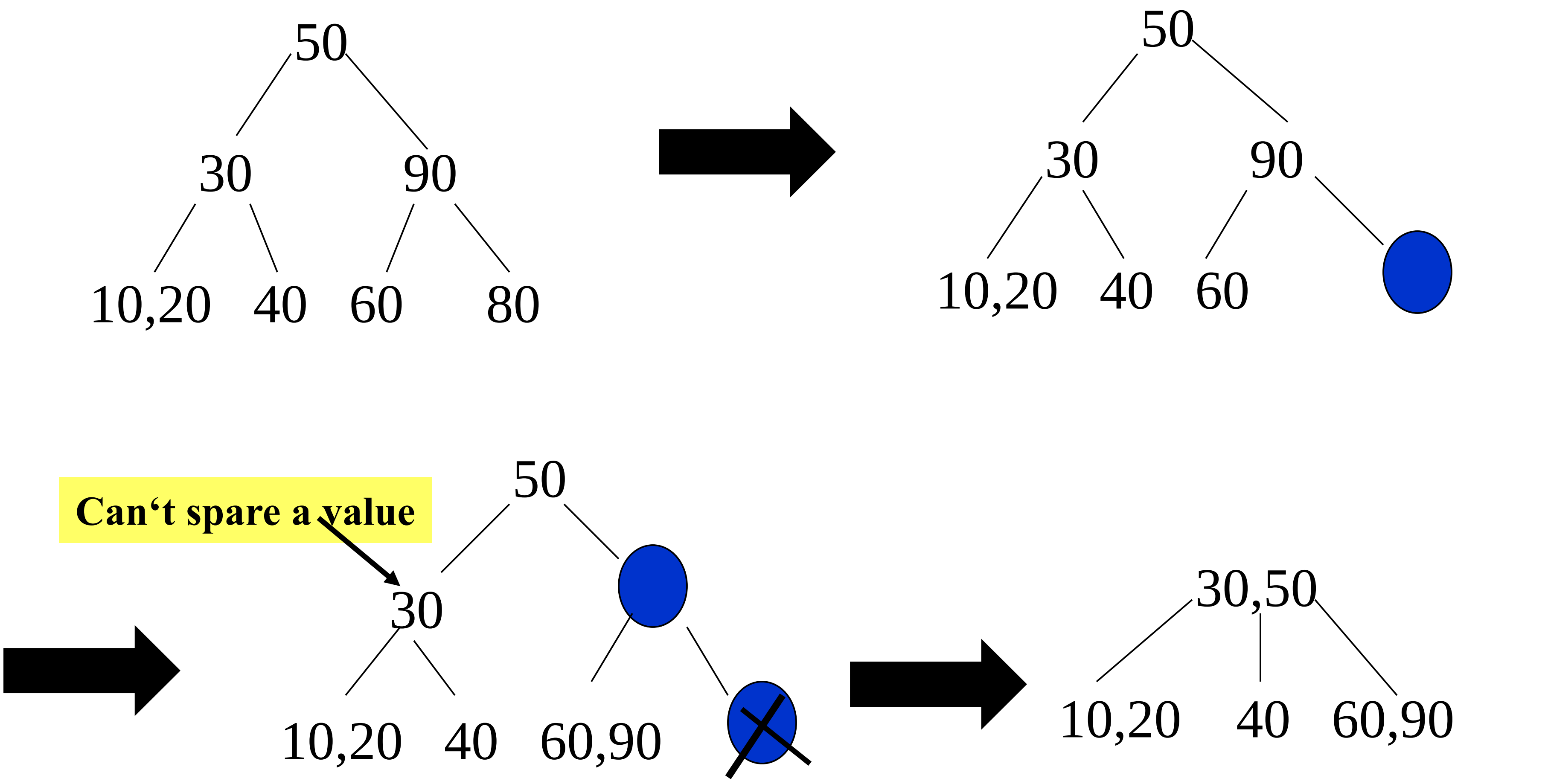

50
90
60,80
100
50
30
90
10,20
40
60,80
100

Delete 100



50
90
60,80
This leaf can spare a value
50
80
60
90
50
30
80
10,20
40
60
90

Delete 80


50
30
90
10,20
40
60
80
50
30
90
10,20
40
60
Can't spare a value
50
30
10,20
40
60,90
30,50
10,20
40
60,90

Operations of 2-3 Trees

all operations have time complexity of log n

Disadvangage :후진 분할(backward split)이 일어남

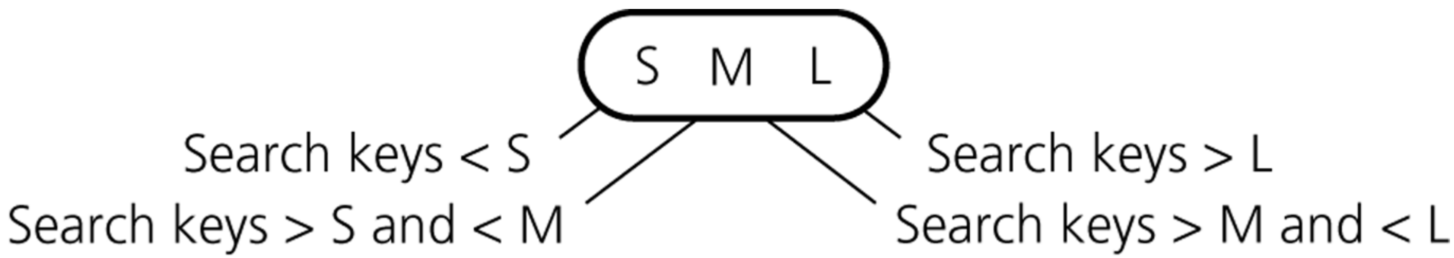
2-3-4 tree

Backward Split(후진분할) does not occur. Good!

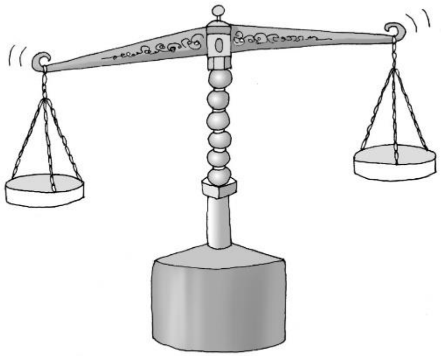
2-3-4 Trees

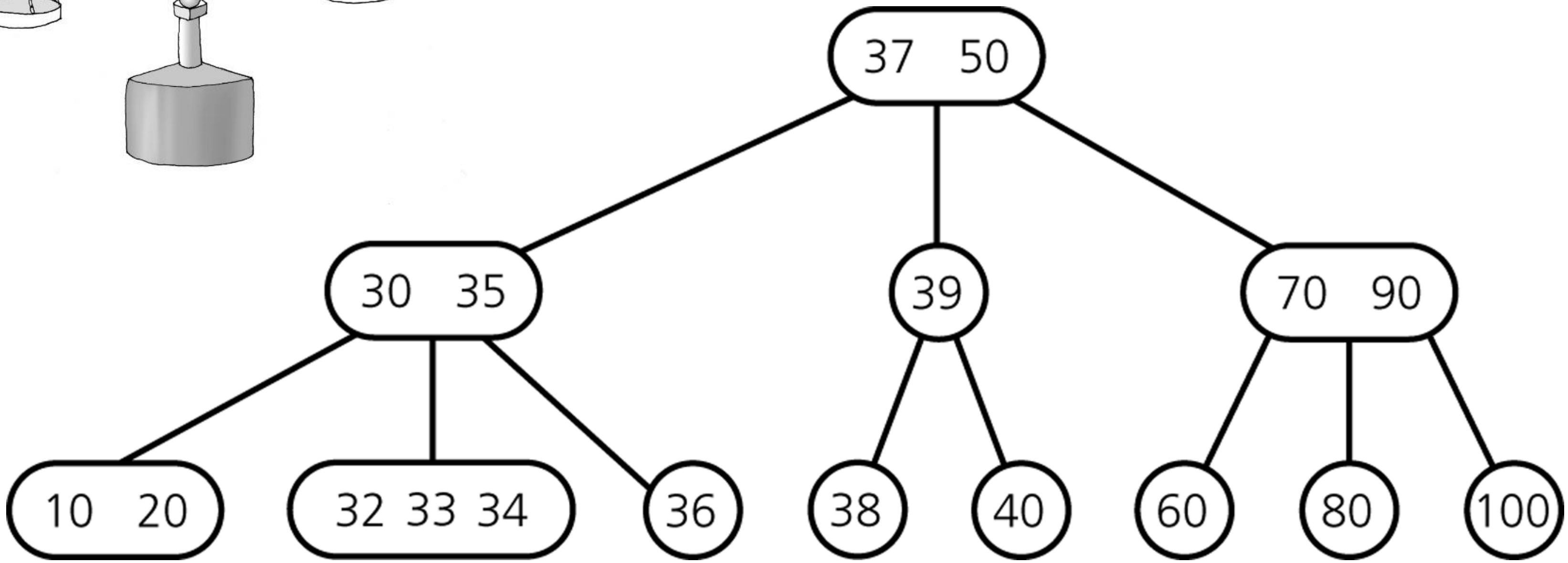
- **similar to 2-3 trees**
- **4-nodes can have 3 items and 4 children**

4-node



2-3-4 Tree Example





2-3-4 Tree: Insertion

Insertion procedure:

- similar to insertion in 2-3 trees
- items are inserted at the leafs
- since a 4-node cannot take another item, 4-nodes are split up during insertion process

Strategy

- on the way from the root down to the leaf: split up all 4-nodes "on the way"

→insertion can be done in one pass (remember: in 2-3 trees, a reverse pass might be necessary). 2-3-4 is better.

삽입은 한번의 패스로 끝. 후진분할이 일어나지 않음. 2-3-4가 더 효율적.

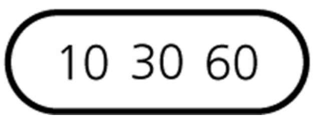
2-3-4 Tree: Insertion

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

2-3-4 Tree: Insertion

Inserting 60, 30, 10, 20 ...

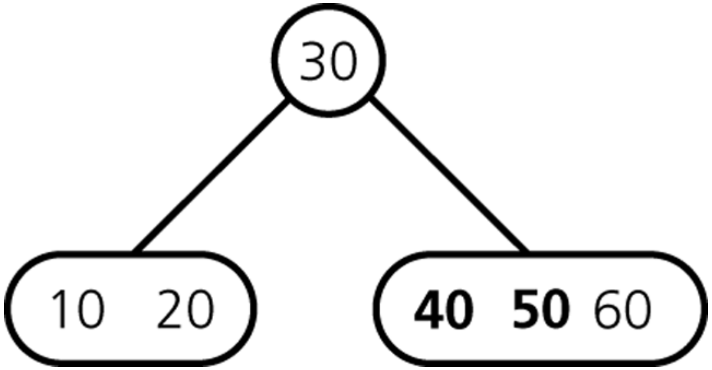
(a)



... 50, 40 ...

2-3-4 Tree: Insertion

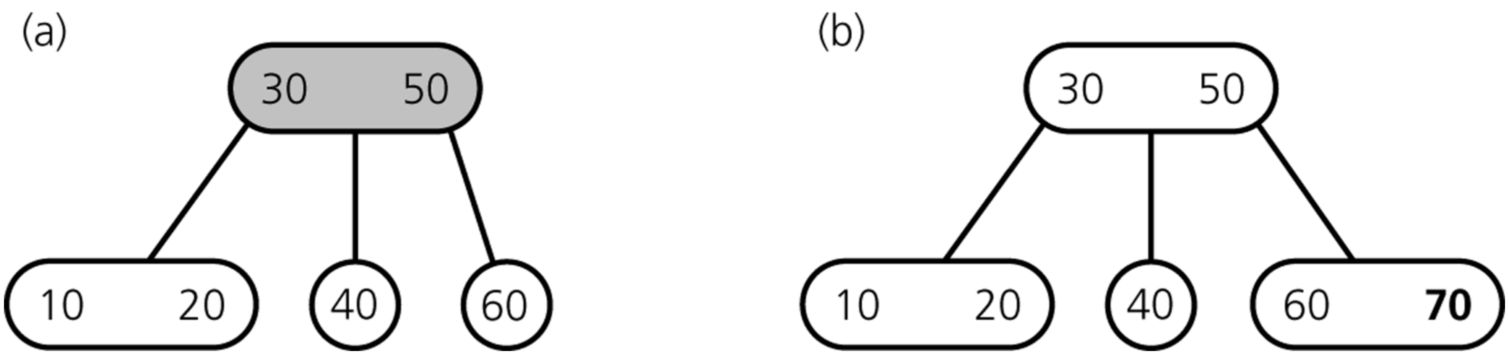
Inserting 50, 40 ...



... 70, ...

2-3-4 Tree: Insertion

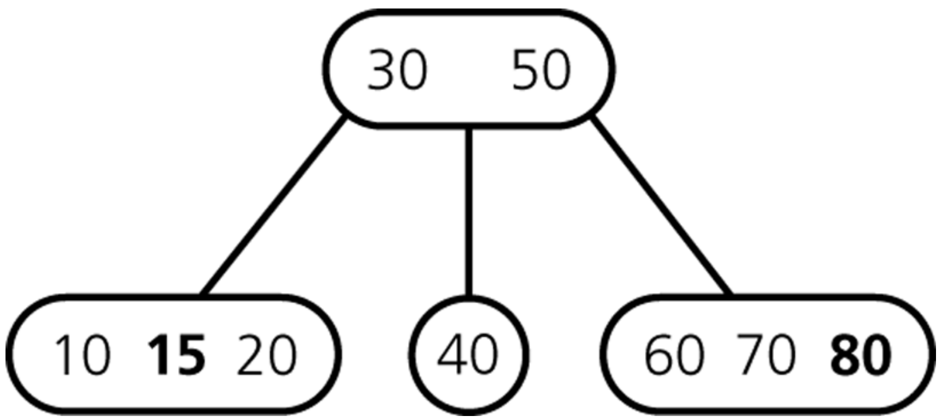
Inserting 70 …



… 80, 15 …

2-3-4 Tree: Insertion

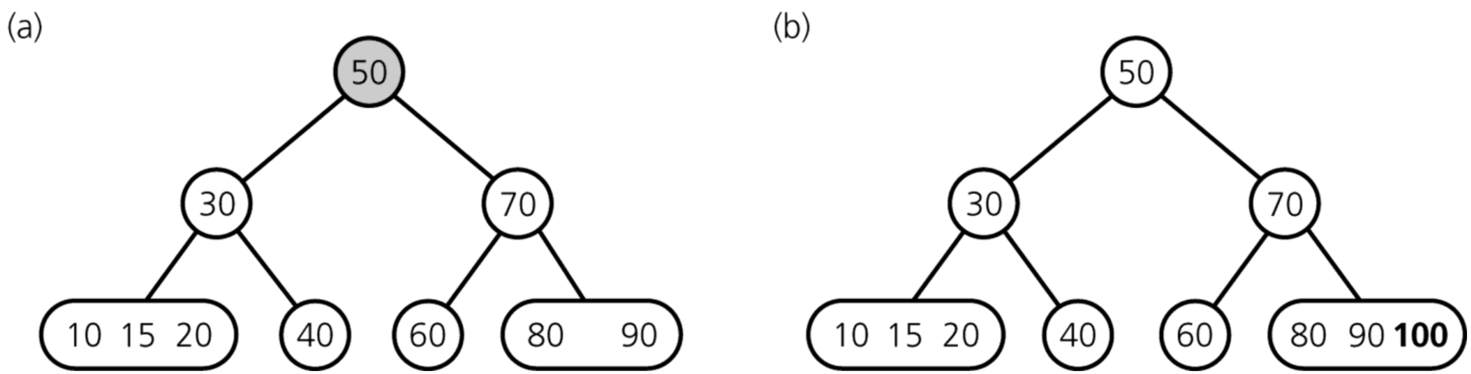
Inserting 80, 15 …



… 90 …

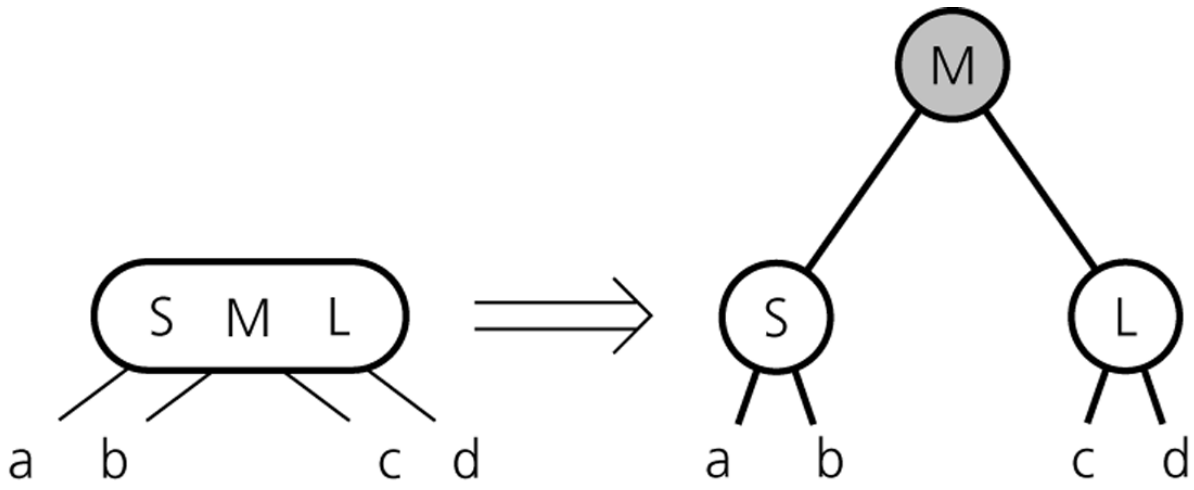
2-3-4 Tree: Insertion

Inserting 100 …



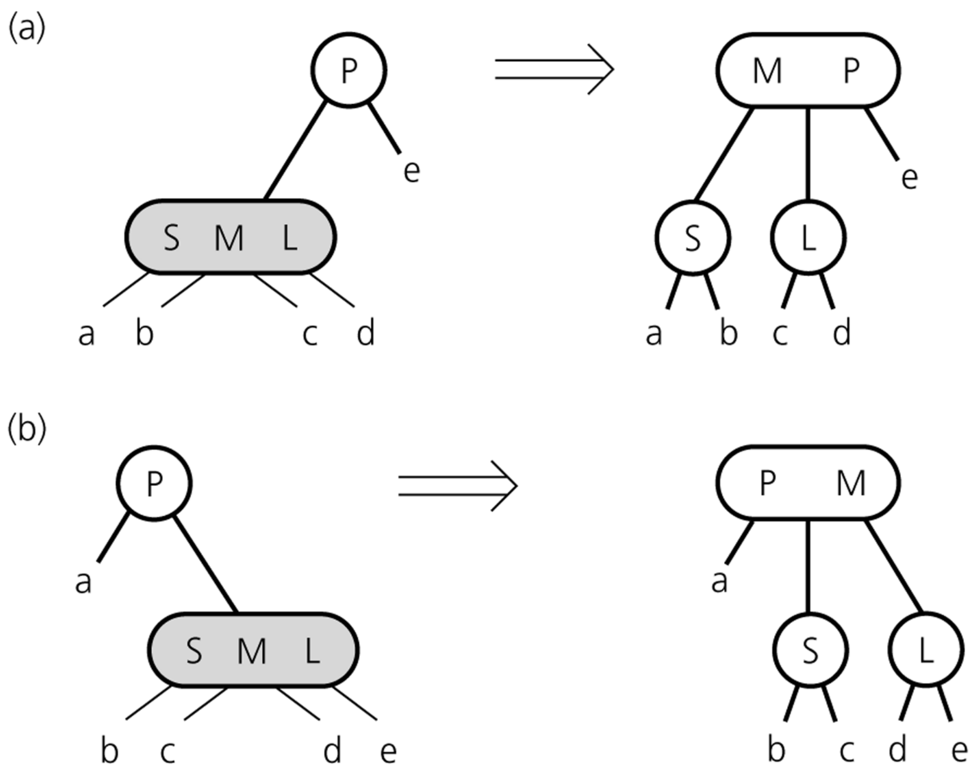
2-3-4 Tree: Insertion Procedure

Splitting 4-nodes during Insertion



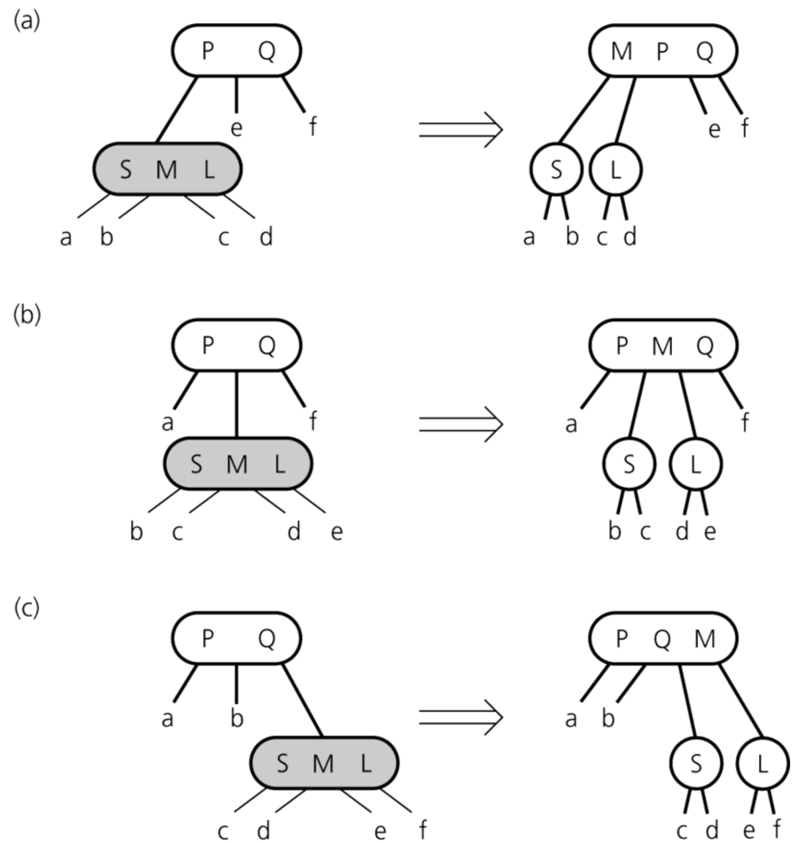
2-3-4 Tree: Insertion Procedure

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



2-3-4 Tree: Insertion Procedure

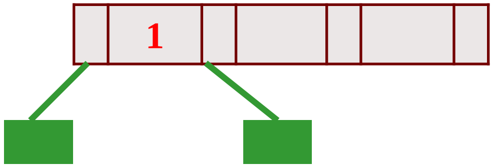
Splitting a 4-node whose parent is a 3-node during insertion



Example 1

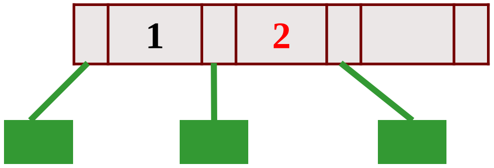
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



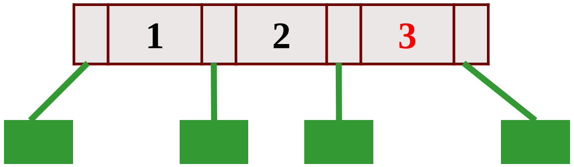
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



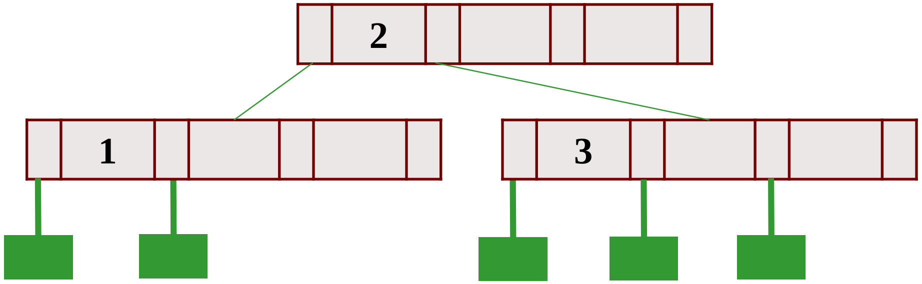
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



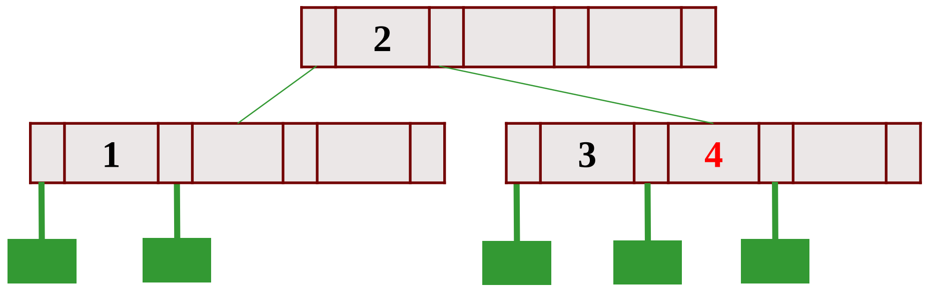
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



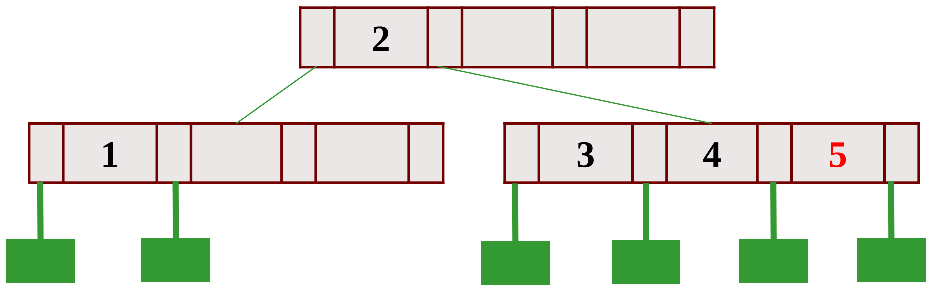
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



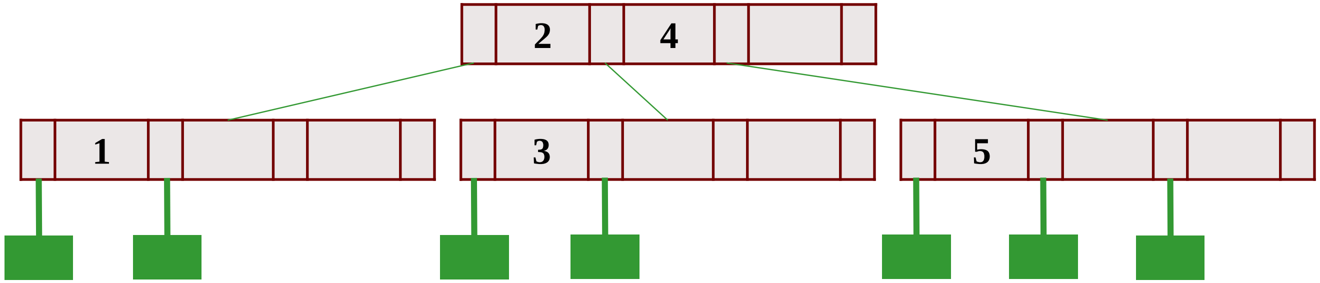
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



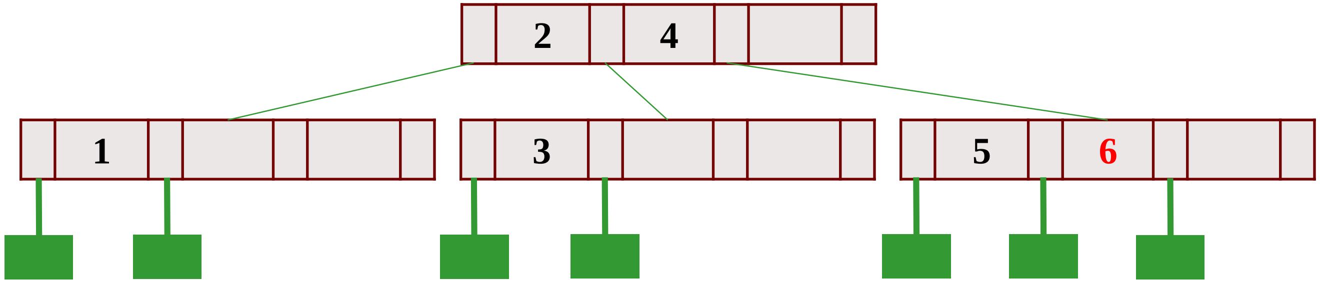
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



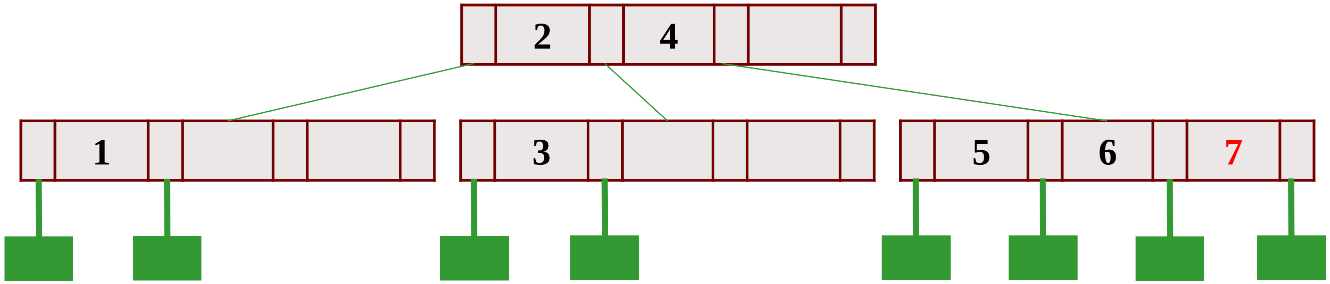
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



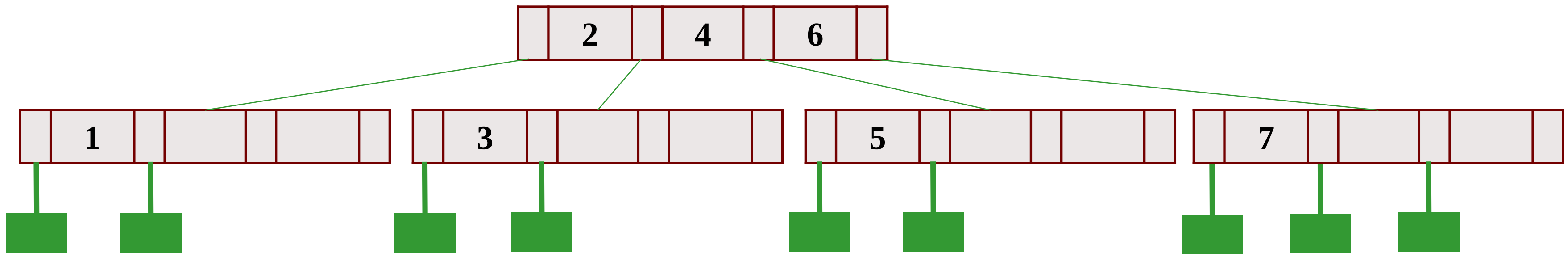
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



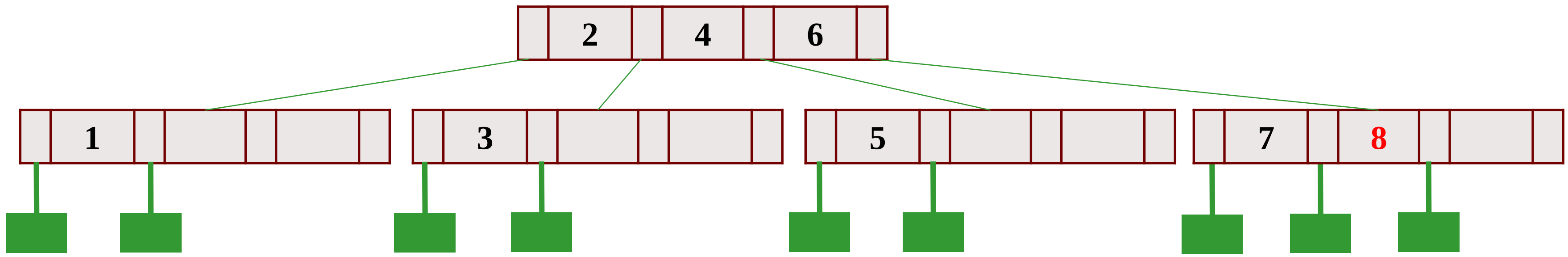
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



2-3-4 TREE

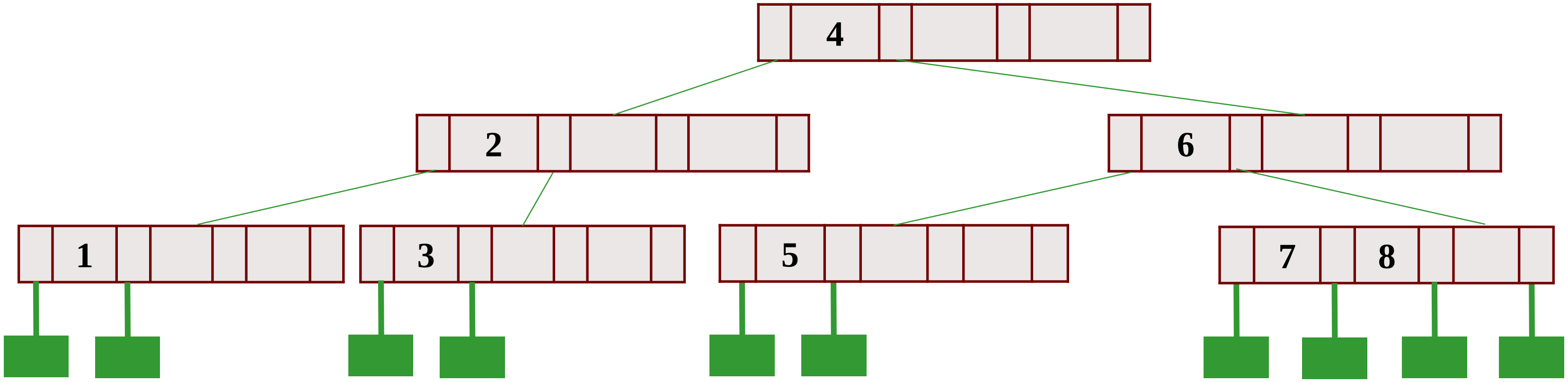
Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10

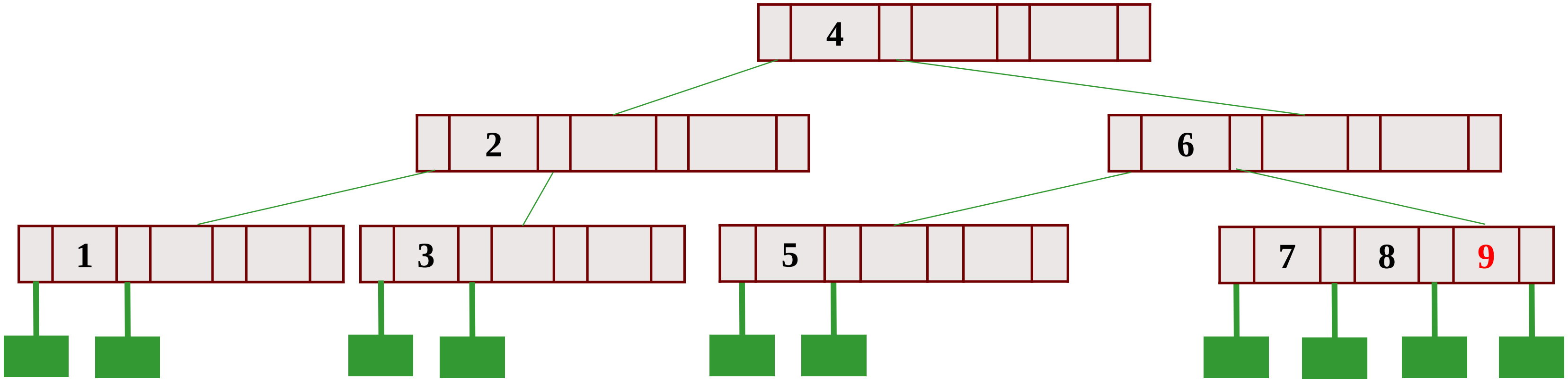
9원소삽입(root거쳐서 삽입)



2-3-4 TREE

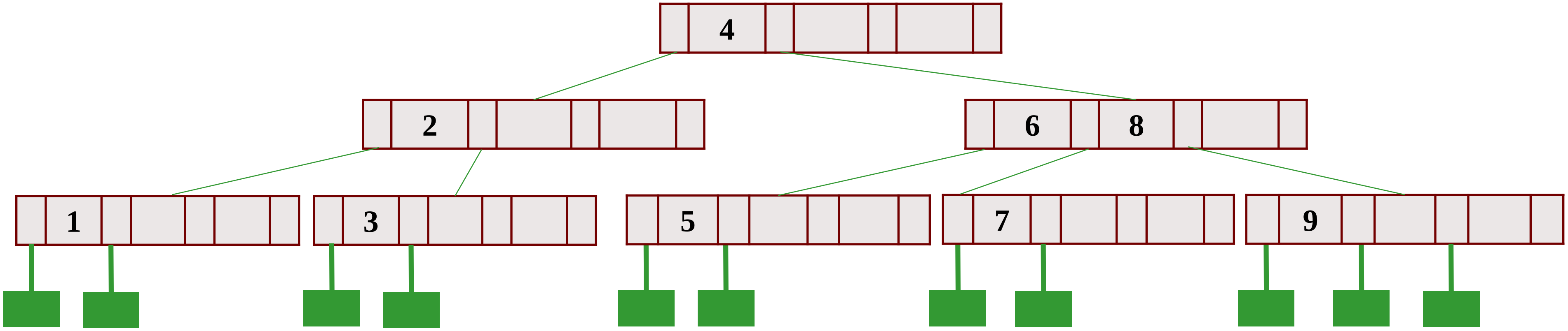
Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10

9원소삽입(root거쳐서 삽입)



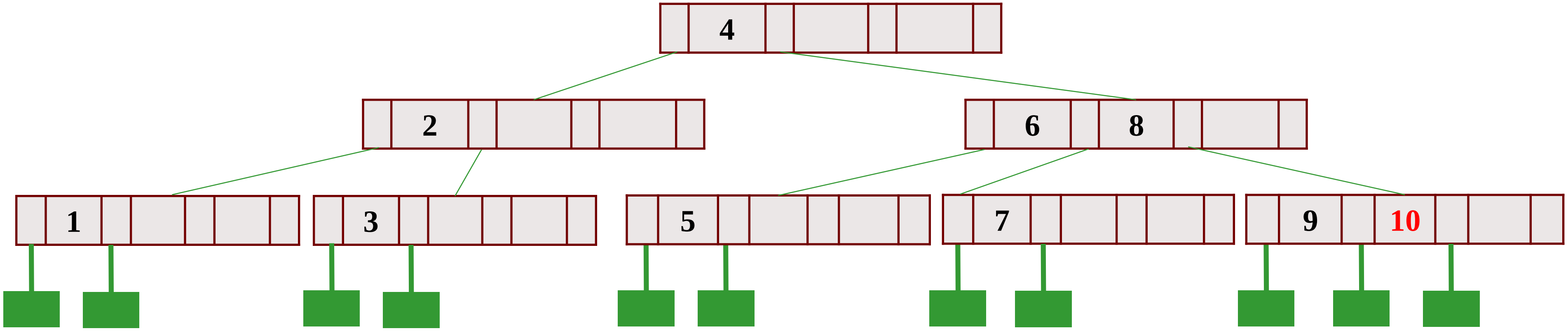
2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



2-3-4 TREE

Insert 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10



Example 2

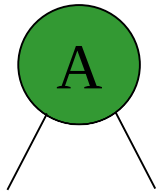
Another example

Keys: A S E R C H I N G X

What would the 2-3-4 tree look like after inserting this set of keys?

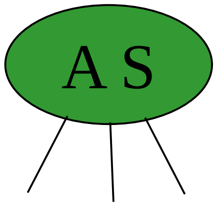
Another example

Keys: A S E R C H I N G X



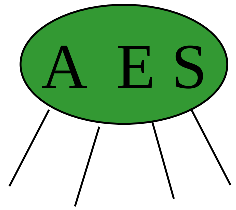
Another example

Keys: A S E R C H I N G X



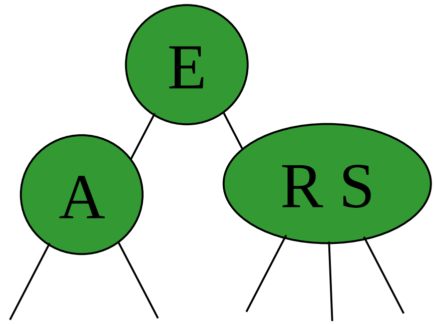
Another example

Keys: A S E R C H I N G X



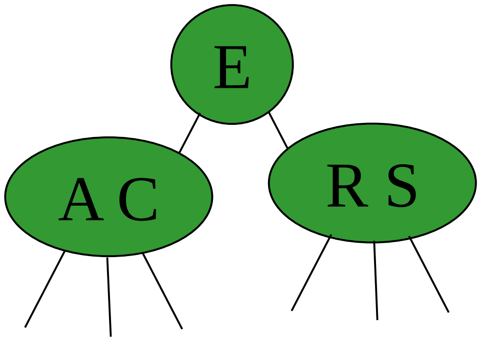
Another example

Keys: A S E R C H I N G X



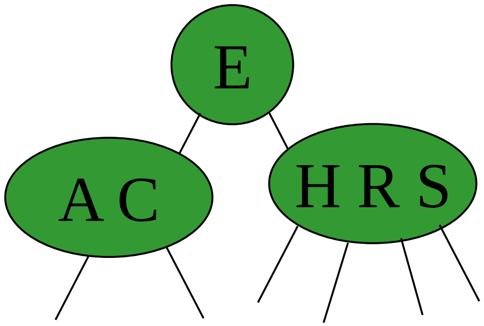
Another example

Keys: A S E R C H I N G X



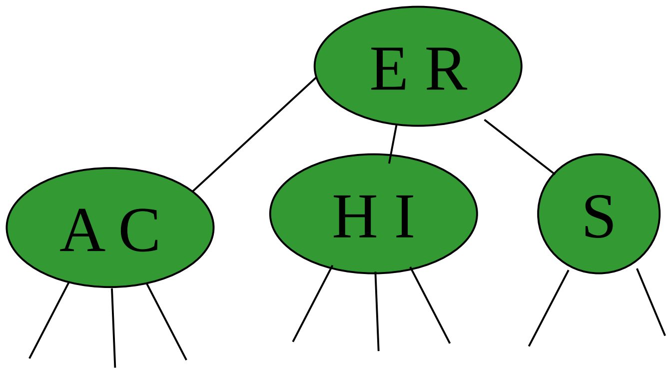
Another example

Keys: A S E R C H I N G X



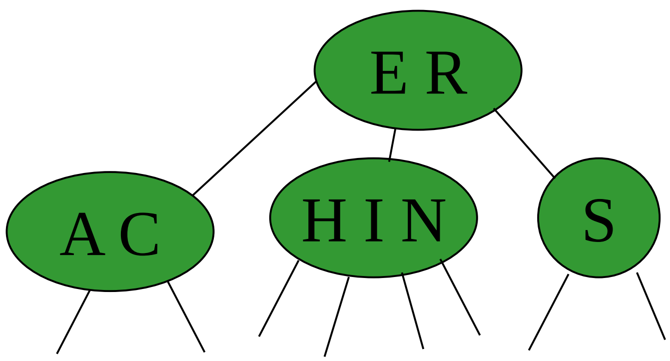
Another example

Keys: A S E R C H I N G X



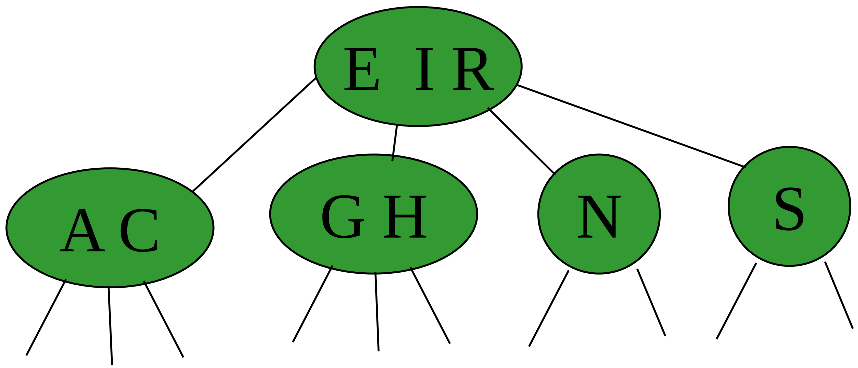
Another example

Keys: A S E R C H I N G X



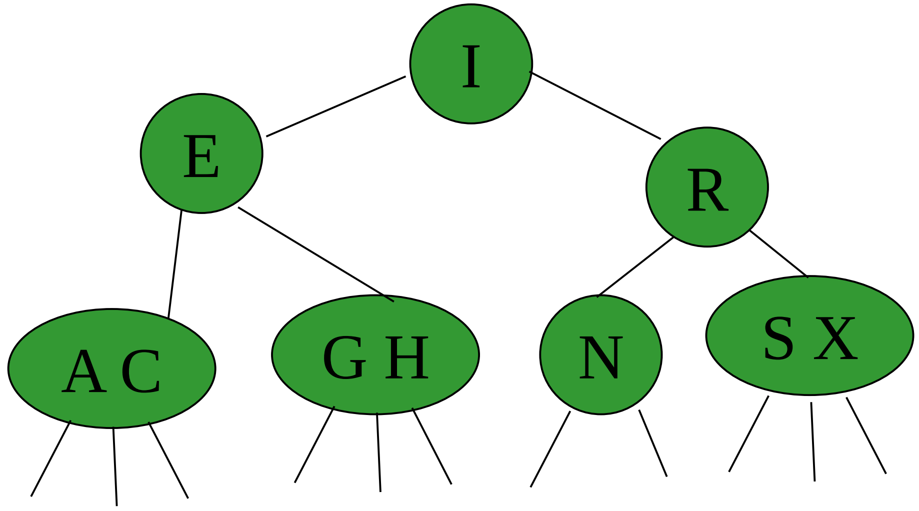
Another example

Keys: A S E R C H I N G X



Another example

Keys: A S E R C H I N G X



2-3-4 Tree: Deletion

Similar to 2-3 tree.

We do not discuss deletion due to limited time.

2-3-4 Trees: Conclusions

- Insertion/deletion algorithms for a 2-3-4 tree require fewer steps than those for a 2-3 tree
  - Only one pass from root to a leaf(삽입삭제시 한번의 패스만 발생. 루트로 영향을 주지 않음)
- A 2-3-4 tree is always balanced
- A 2-3-4 tree requires more storage than a binary search tree
- Allowing nodes with more data and children is counter productive, unless the tree is in external storage

Red black tree(레드블랙트리)

- Reb black tree is binary search tree.(이진탐색트리)

Red-Black Tree

- binary-search-tree representation of 2-3-4 tree(즉 실제로는 234트리와 같은데 이진 탐색트리로 표현)이진트리가 이해하기쉽기때문에
- 3- and 4-nodes are represented by equivalent binary trees
- red and black child pointers are used to distinguish between original 2-nodes and 2-nodes that represent 3- and 4-nodes

Review: Red-Black Trees

- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$ (very good)

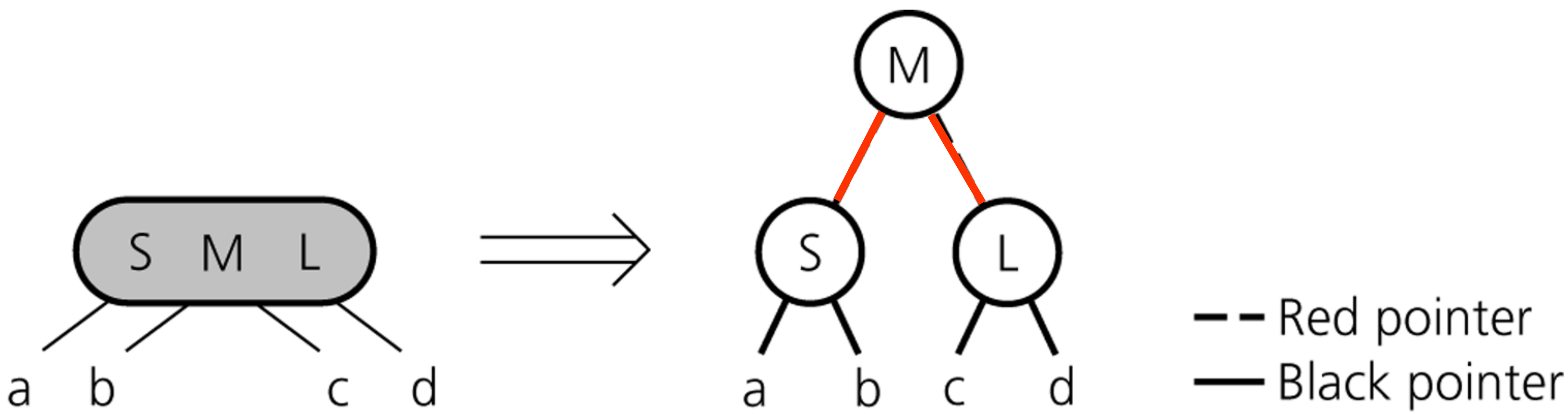
Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black(단말은 항상검정)
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path(연속으로 두개의 빨강노드가 오면 안됨)
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. **The root is always black(루트노드는 항상 검정)**

Black-Height(검은노드높이)

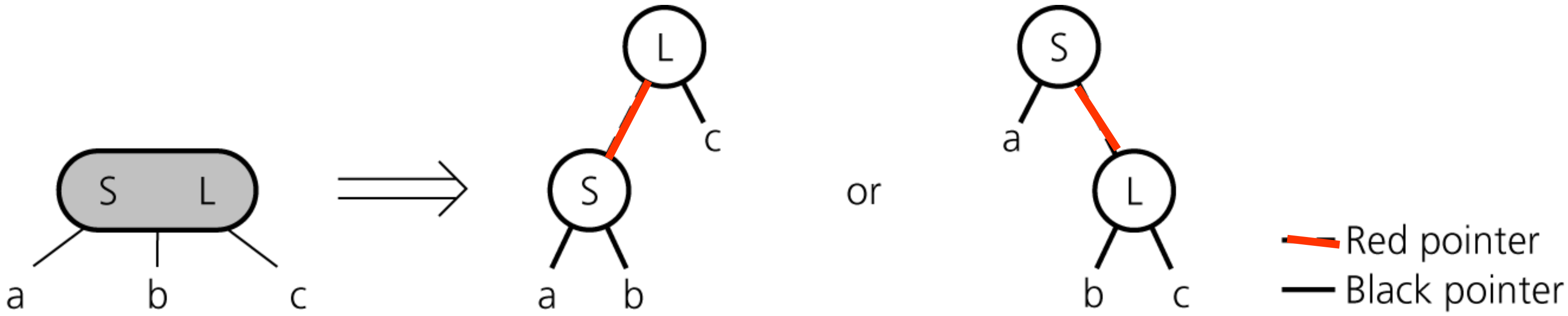
- *black-height*: # black nodes on path to leaf
- *What is the minimum black-height of a node with height h?(검은 노드의 개수는?)*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$
  - Proved by (what else?) induction

Red-Black Representation of 4-node

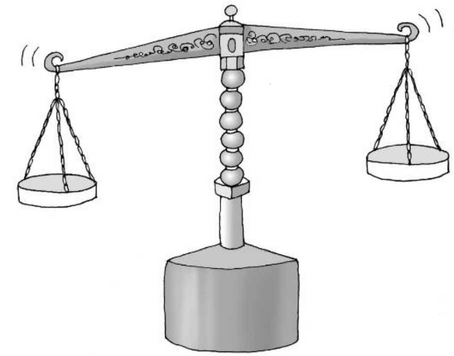


234 트리를 레드블랙트리로 표현. 이진탐색트리로 변환

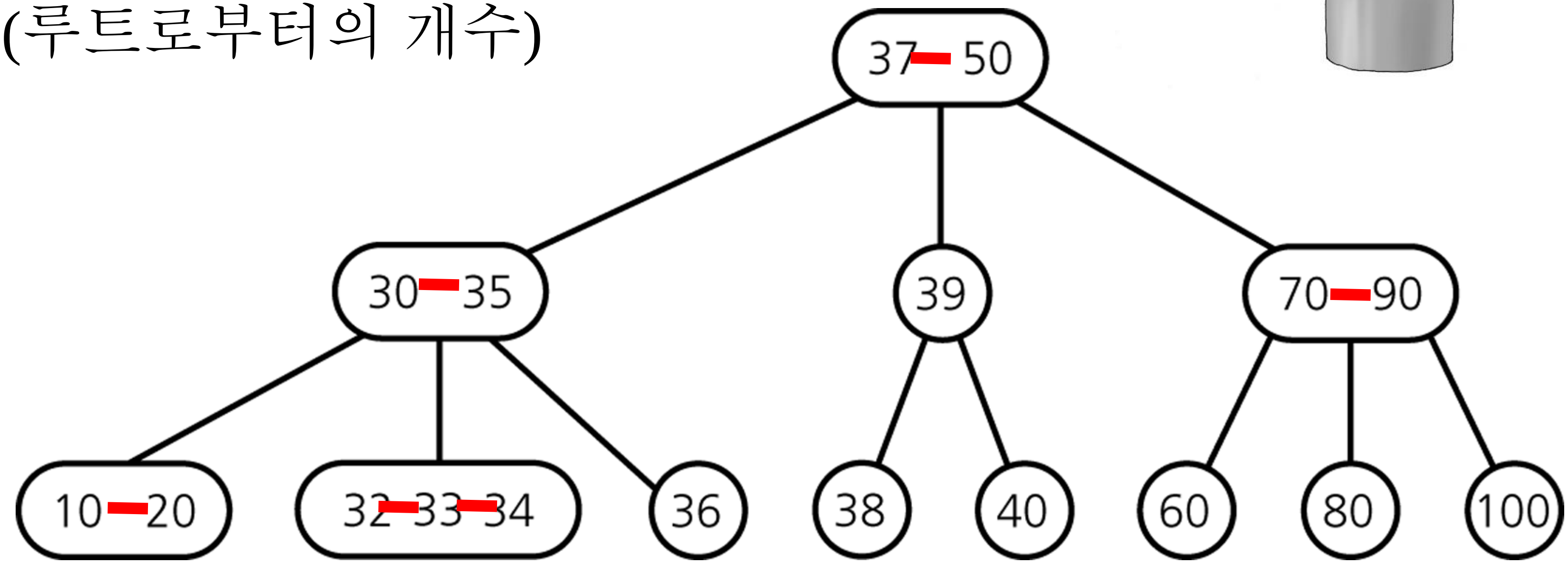
Red-Black Representation of 3-node



Red-Black Tree Exampl

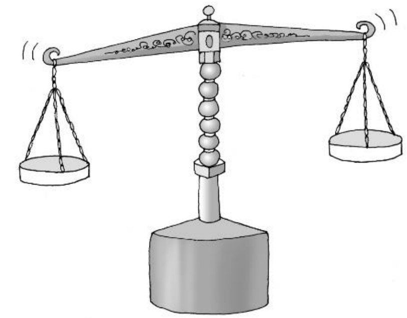


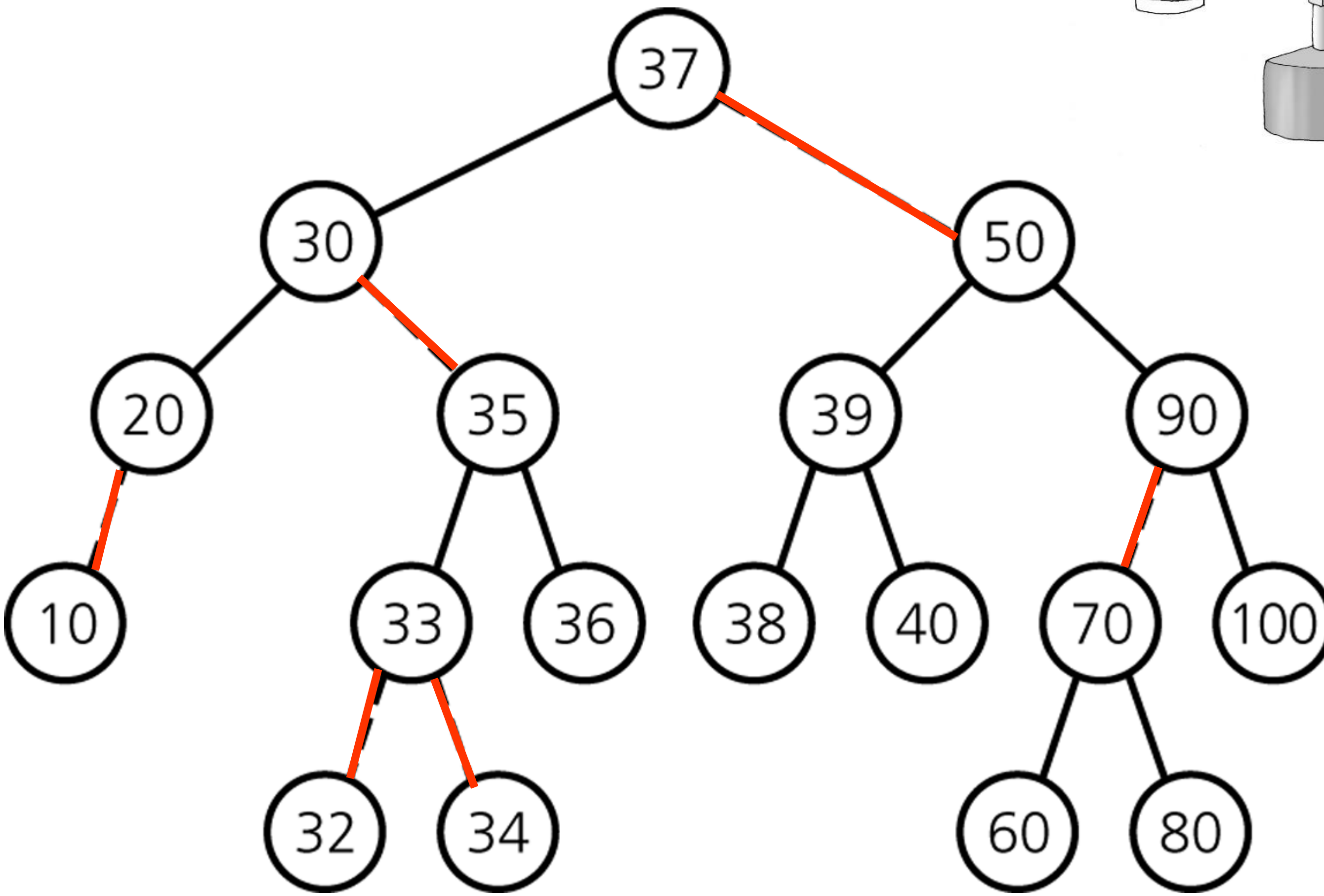
검정노드의 높이
(루트로부터의 개수)



2-3-4 tree

Red-Black Tree Example





RBT violations occur in cases

Case 1:LRr

Case 2: LLr

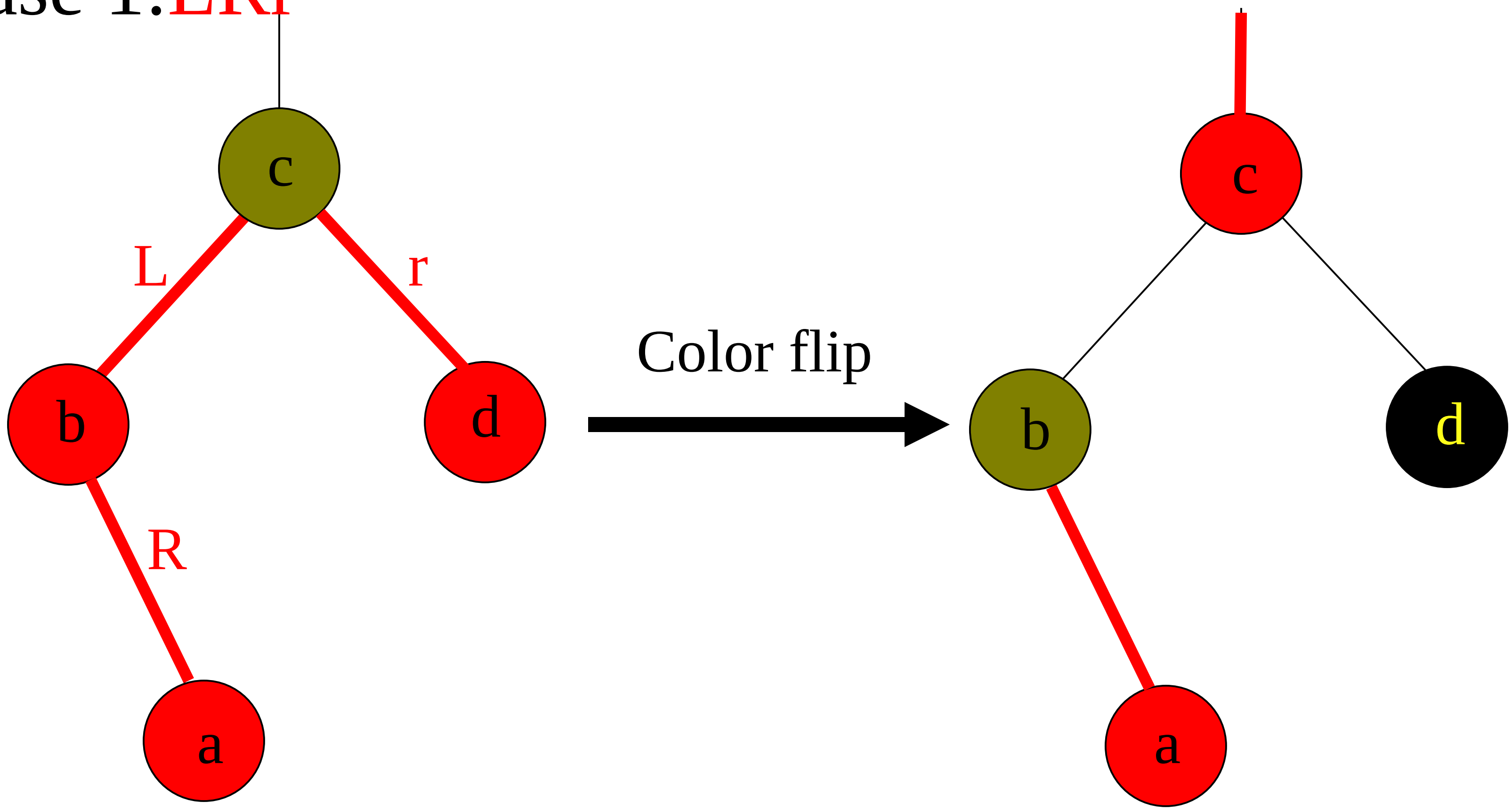
Case 3: LRb

Case 4: LLb

Two consecutive red nodes
(두 개의 연속된 빨강노드)

RBT Violation Fixes

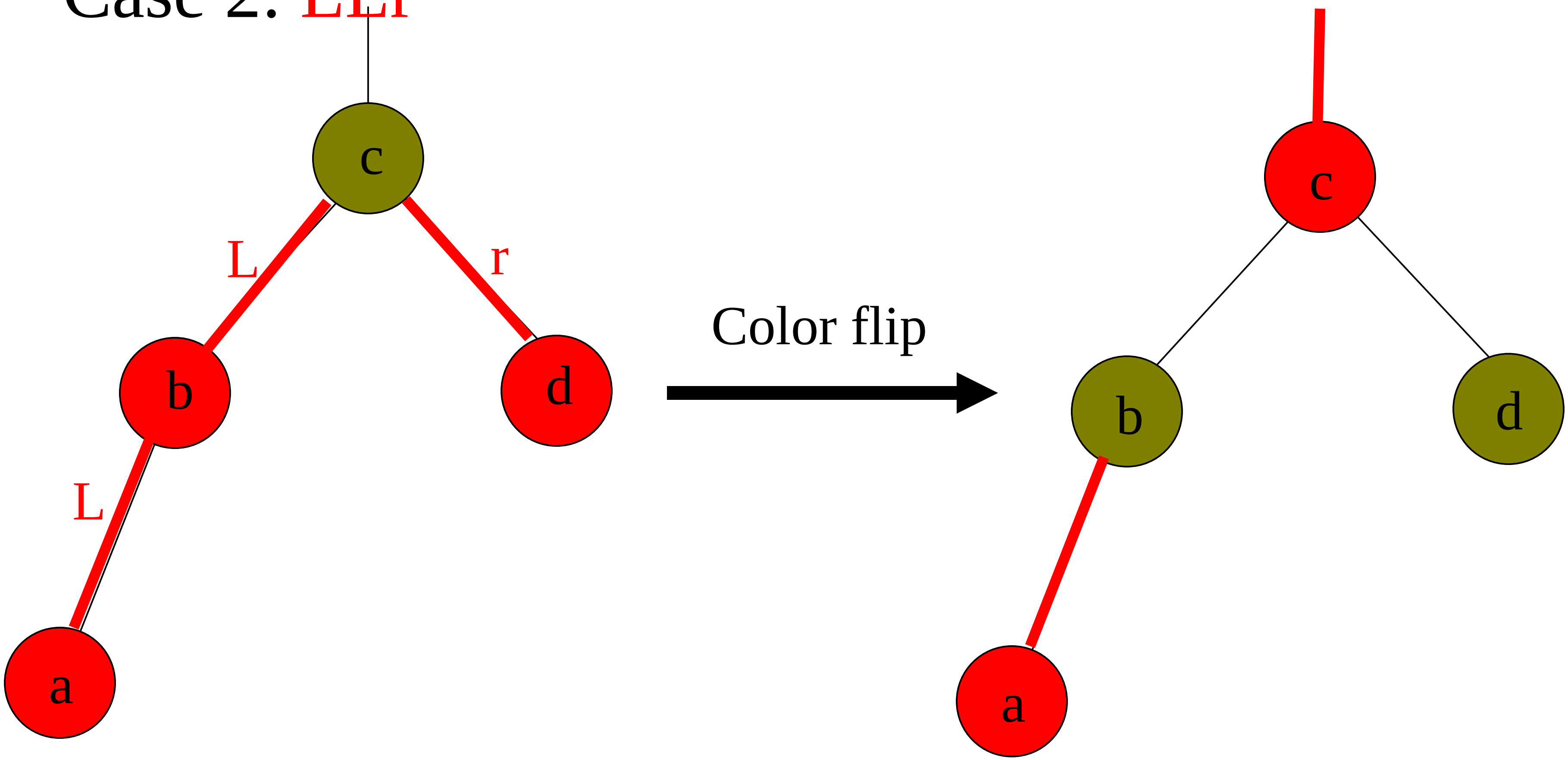
Case 1:LRr



Color flip fixes or moves violation towards the root.

RBT Violation Fixes

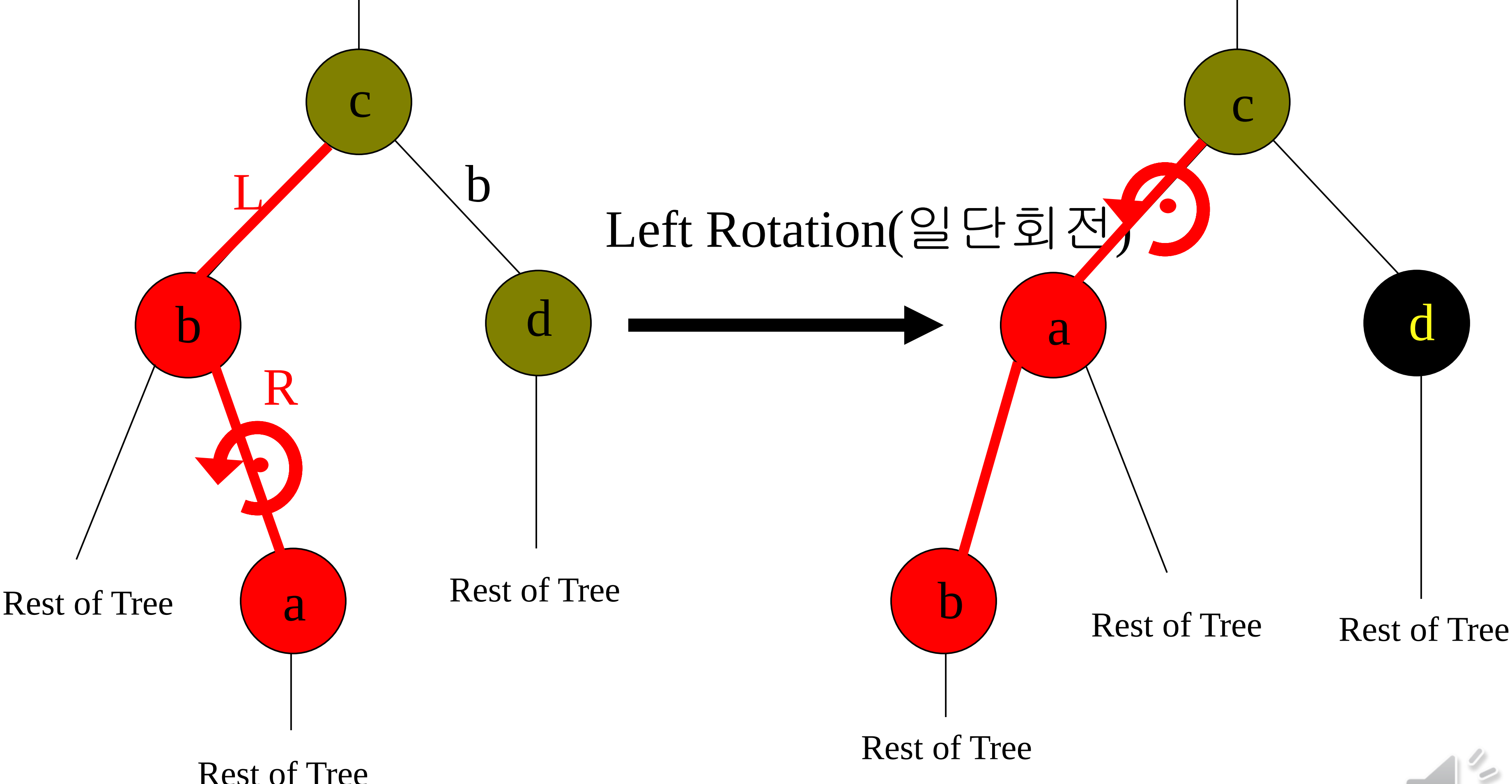
Case 2: LLr



Color flip fixes or moves violation towards the root.

RBT Violation Fixes

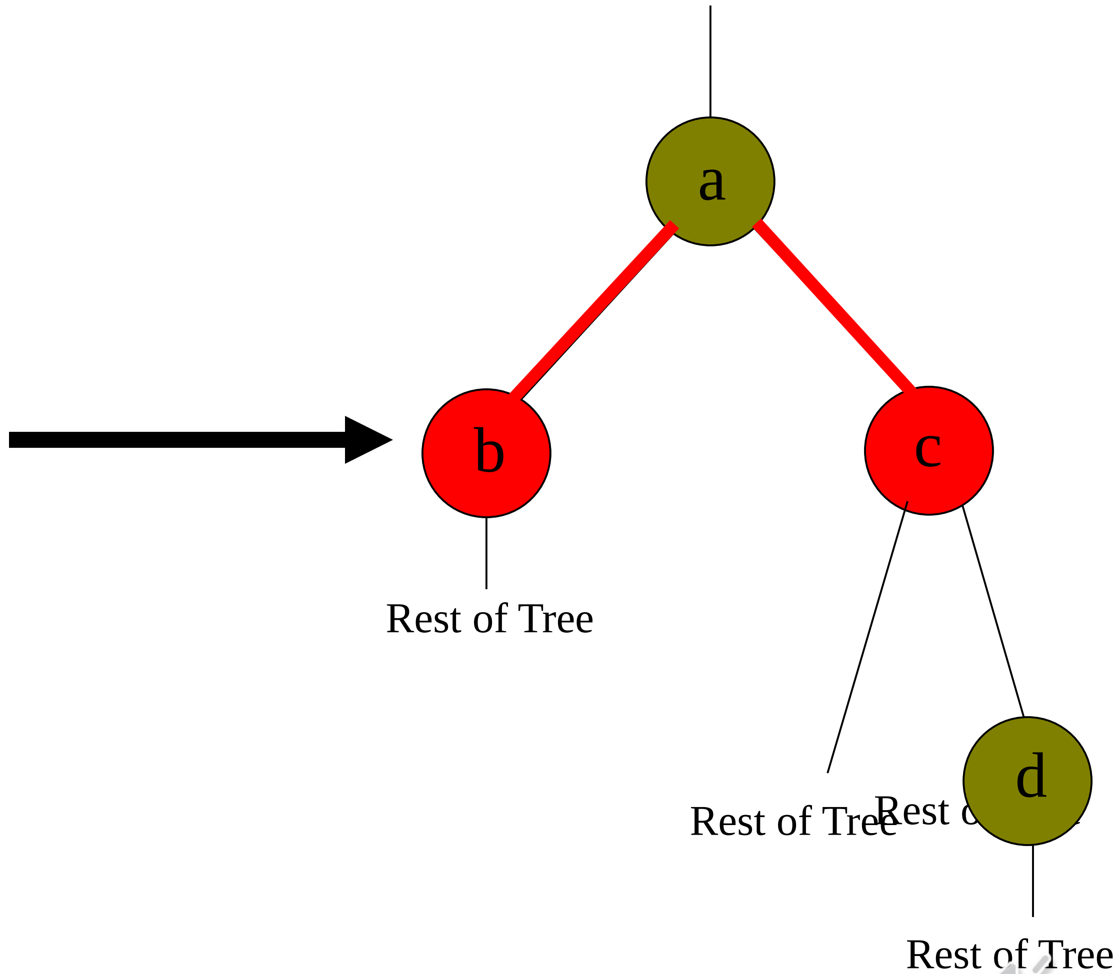
Case 3: LRb



Converted to Case 4

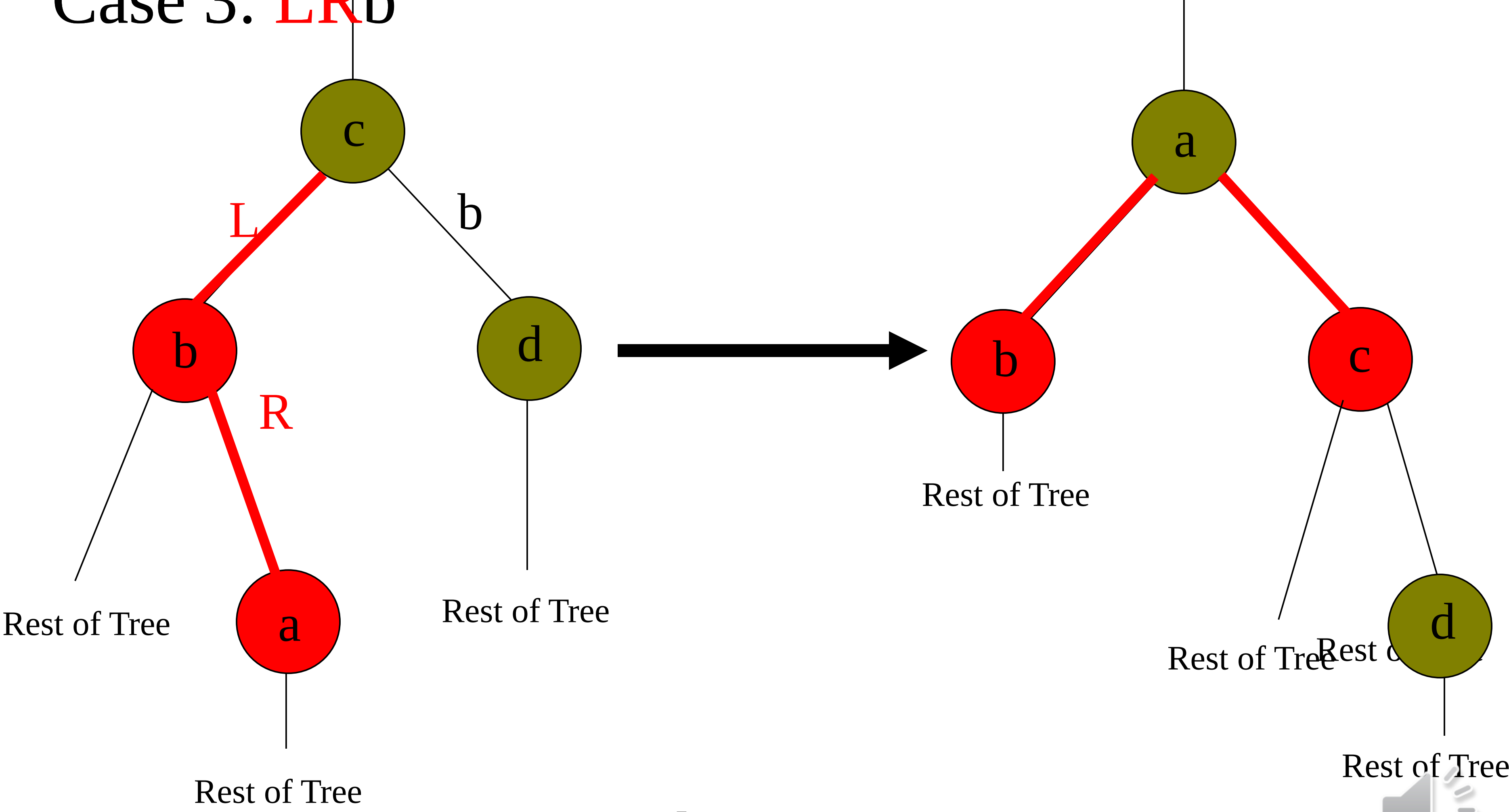
RBT Violation Fixes

Case 3: LRb



RBT Violation Fixes(summary)

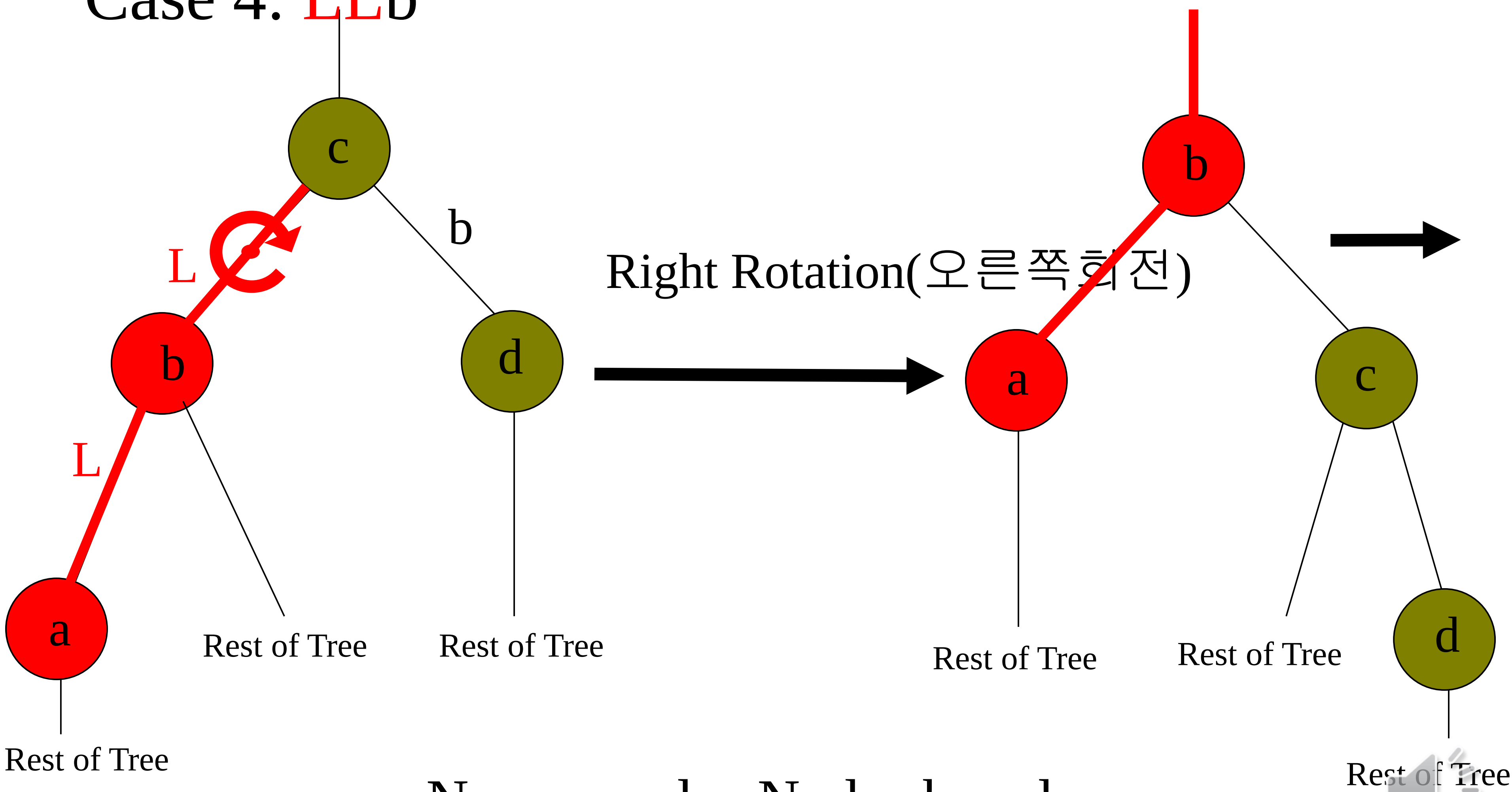
Case 3: LRb



Converted to Case 4

RBT Violation Fixes

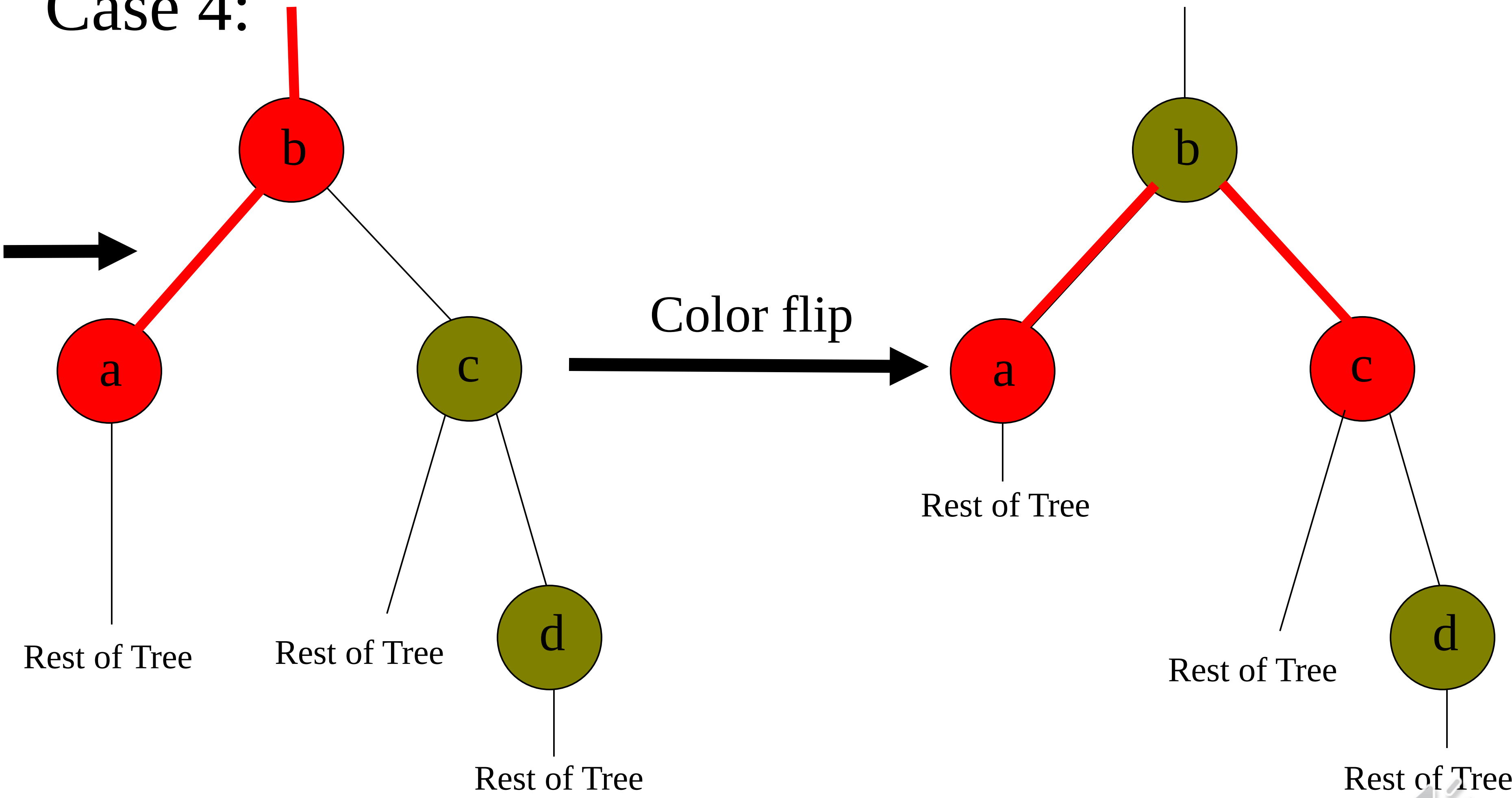
Case 4: LLb



Now re-color Nodes b and c.

RBT Violation Fixes

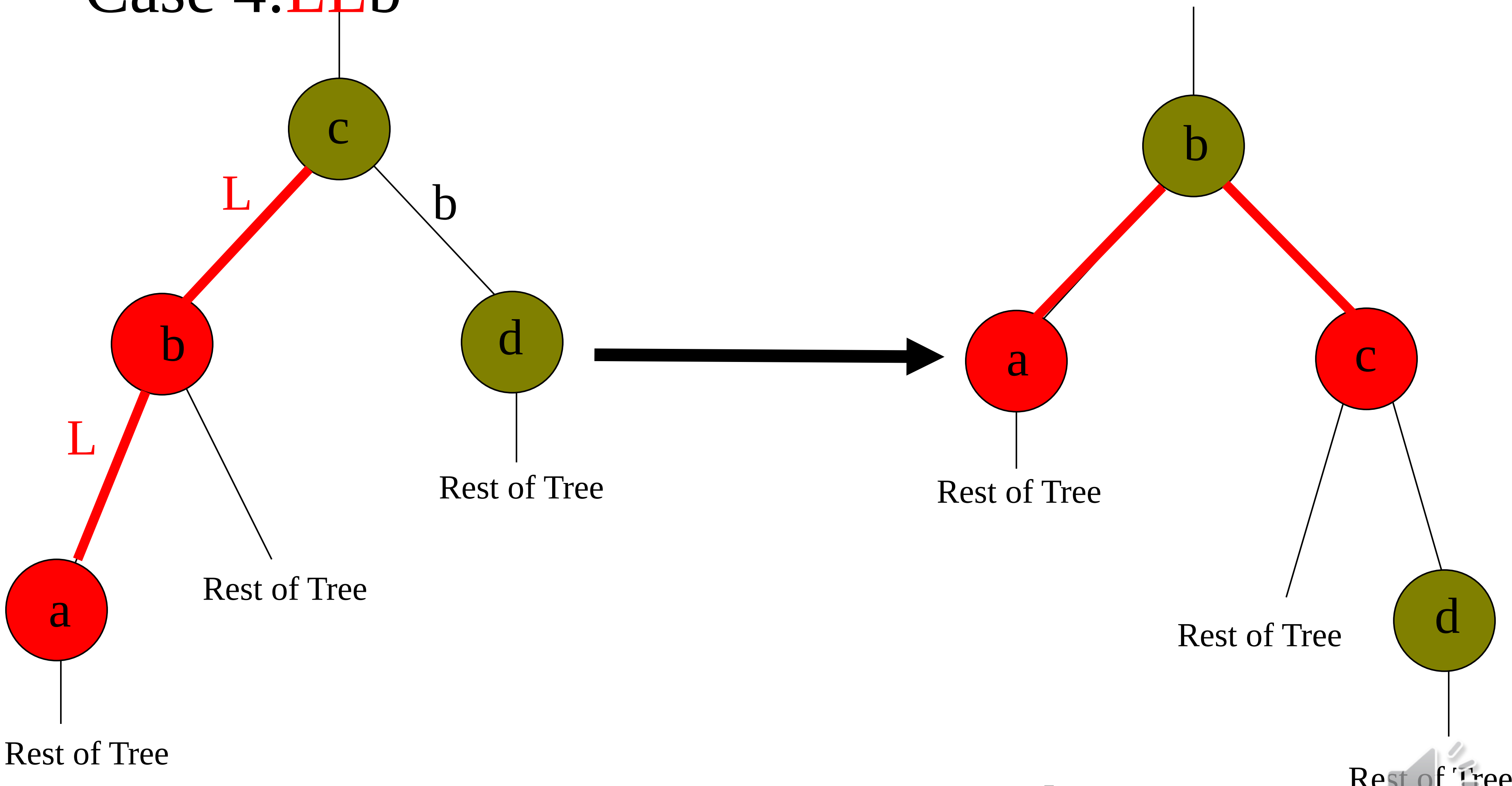
Case 4:



RBT violation is removed (see next slide).

RBT Violation Fixes(summary)

Case 4:LLb



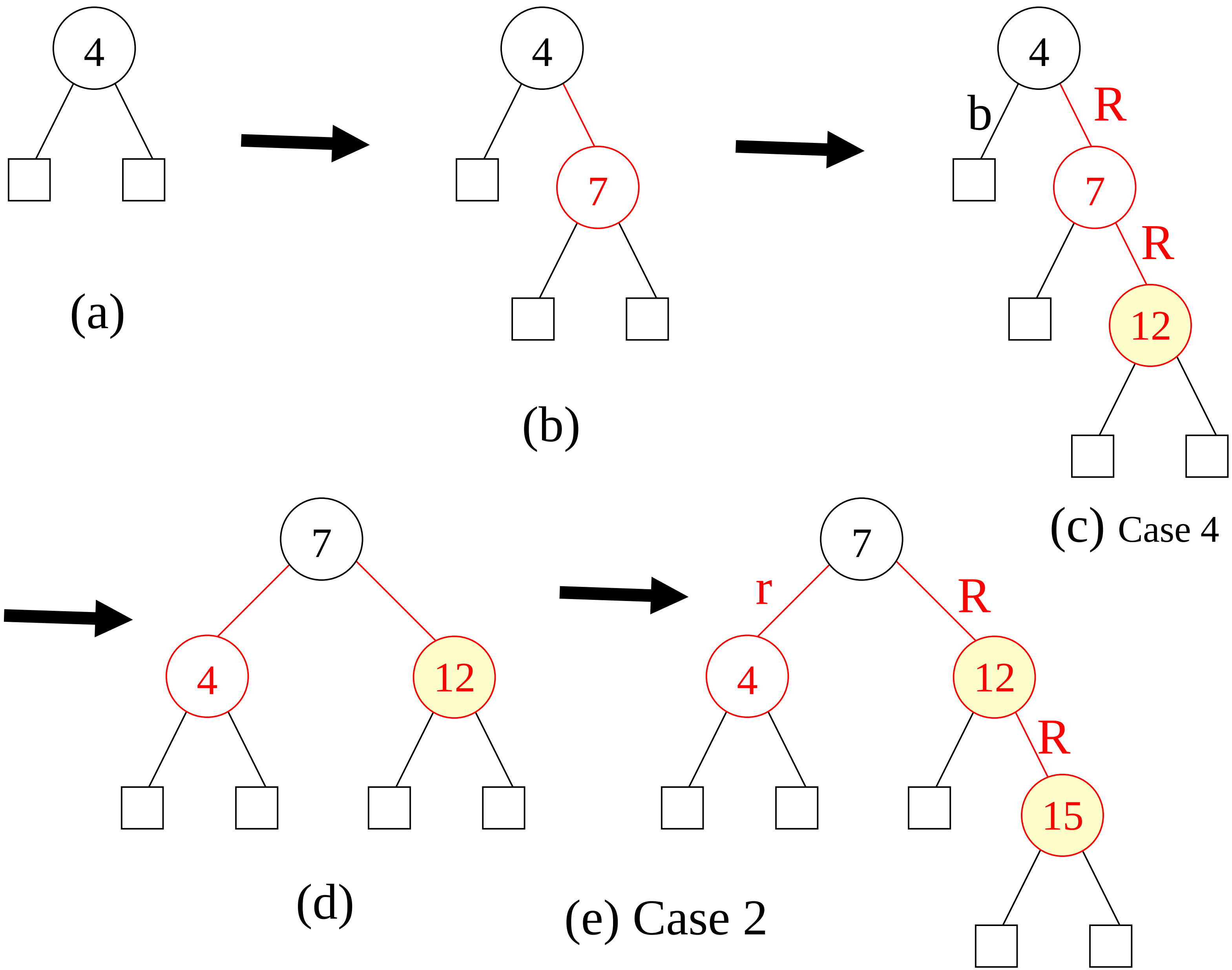
RBT structure is restored.

Mirror images are same

- RRb
- RLb
- RRr
- RLr

Example 1

insertion



(a)

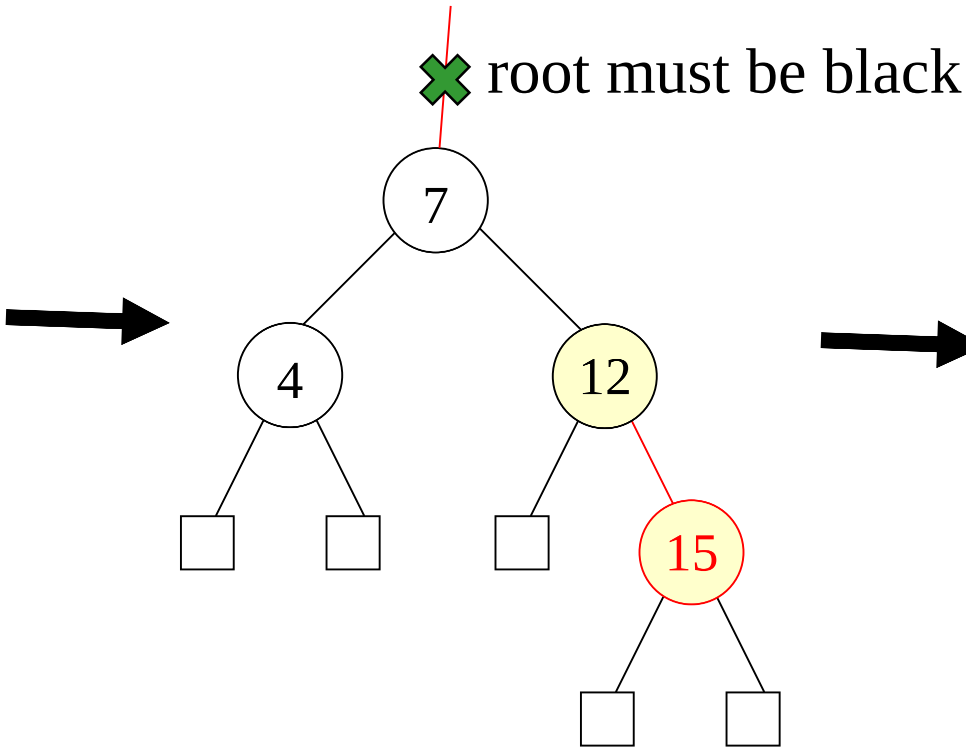
(b)

(c) Case 4

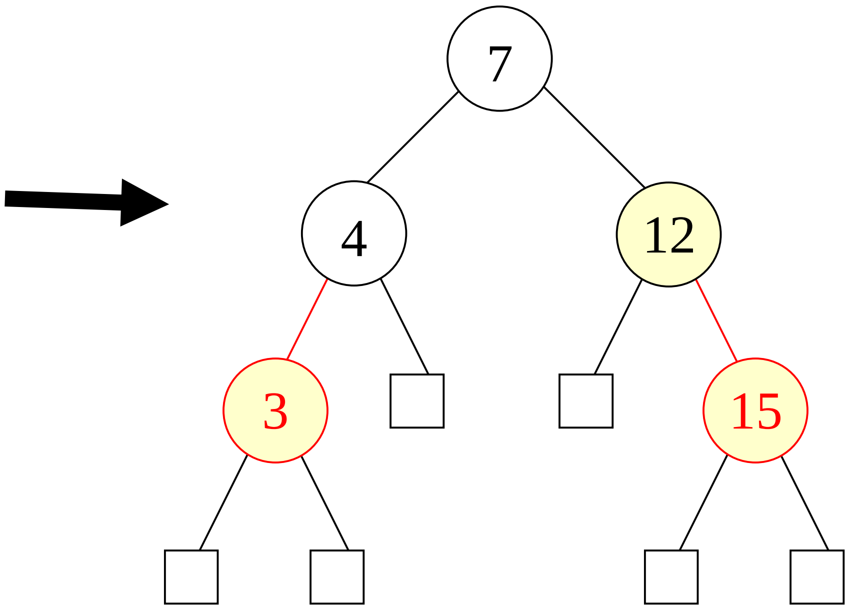
(d)

(e) Case 2

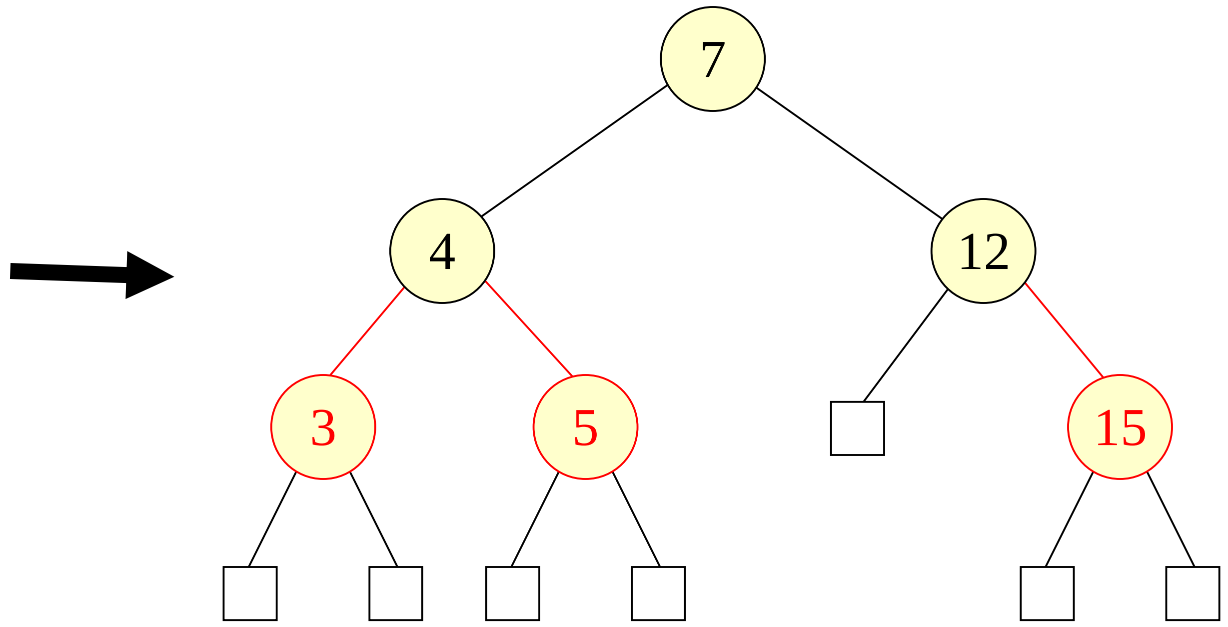
+root must be black



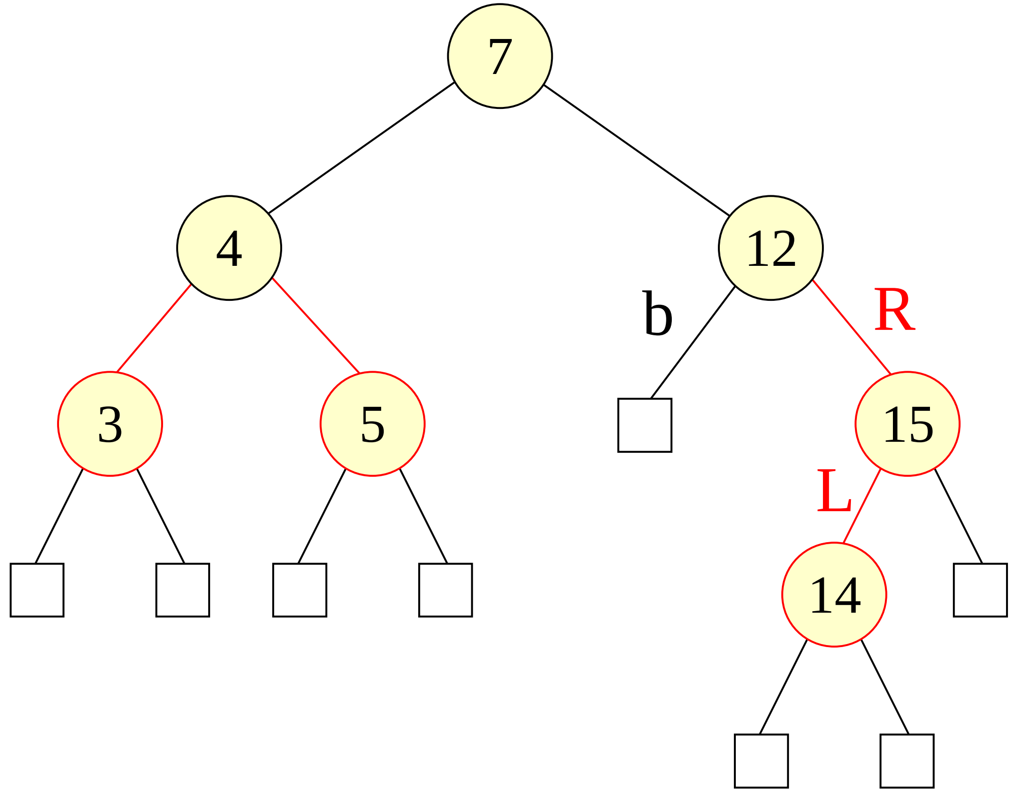
(f)



(g) Insert 3

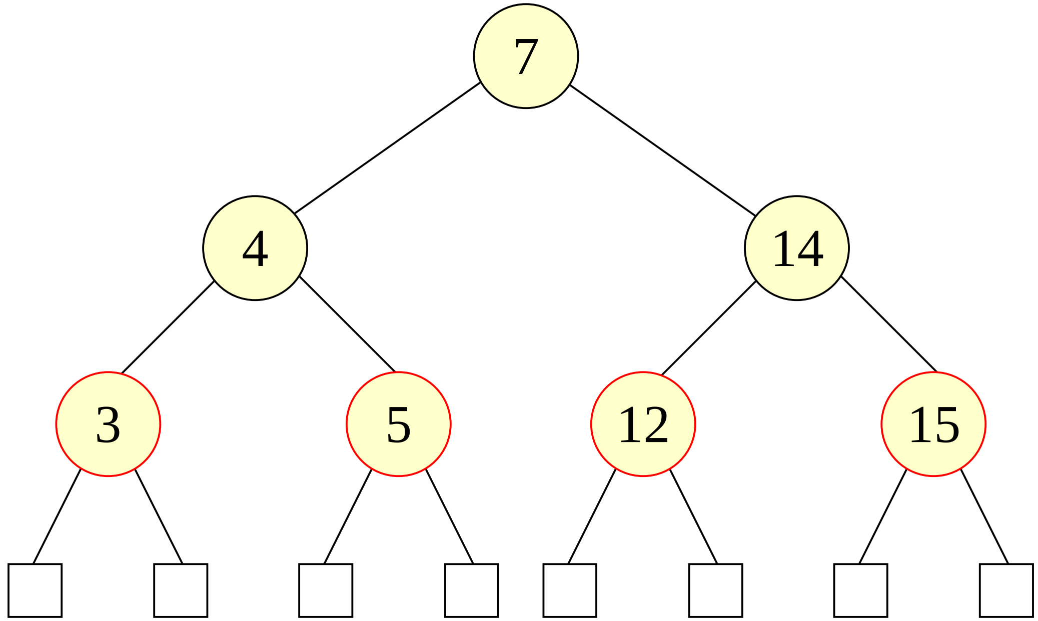


(h) Insert 5

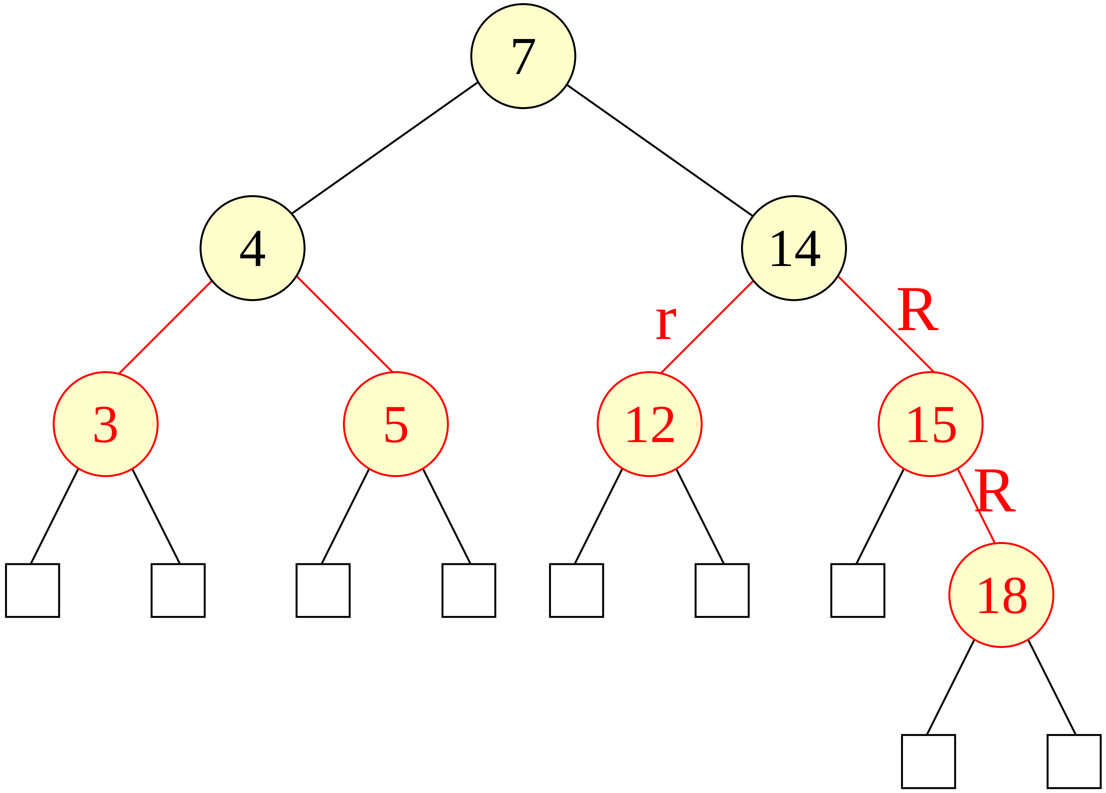


(i)Insert 14

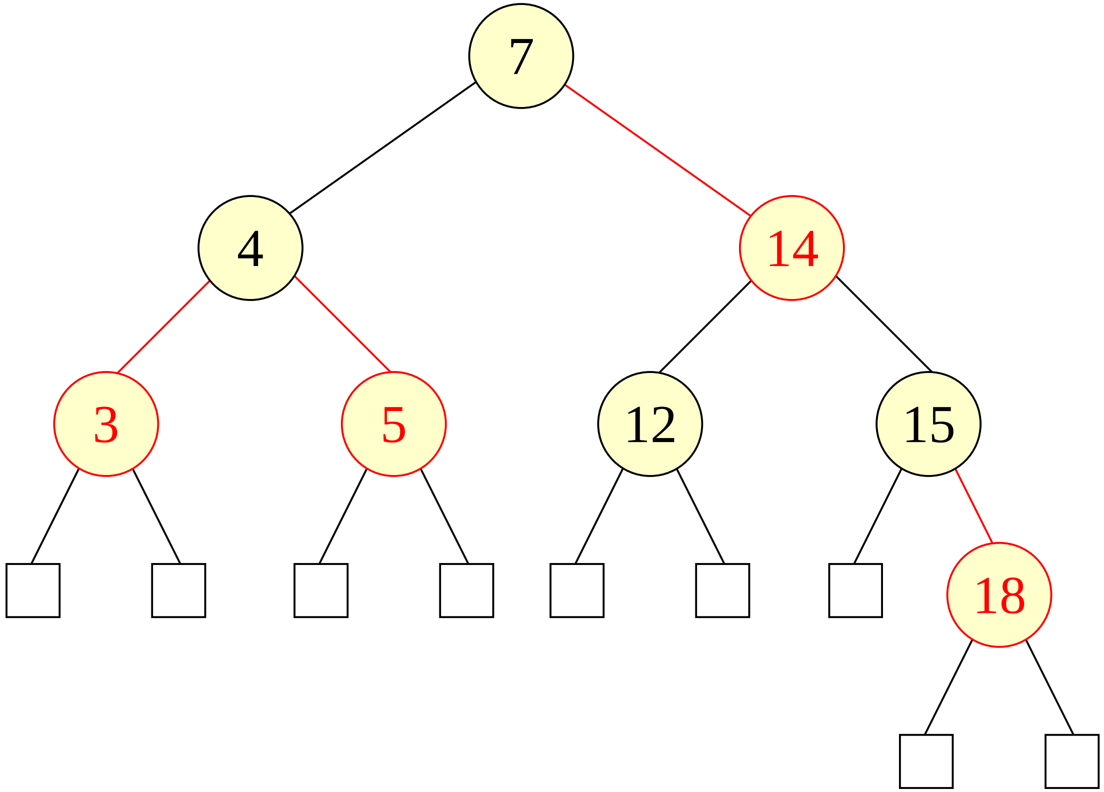
Case 3



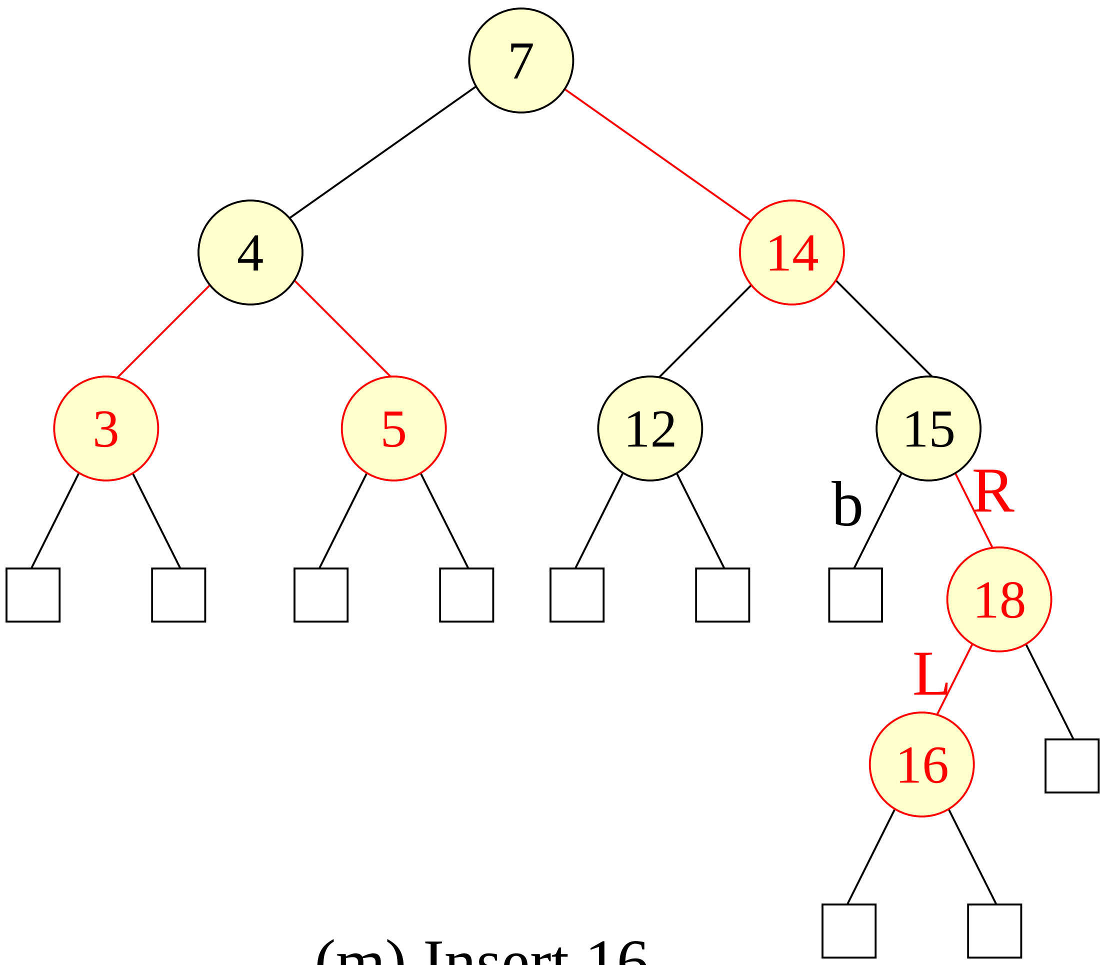
(j)



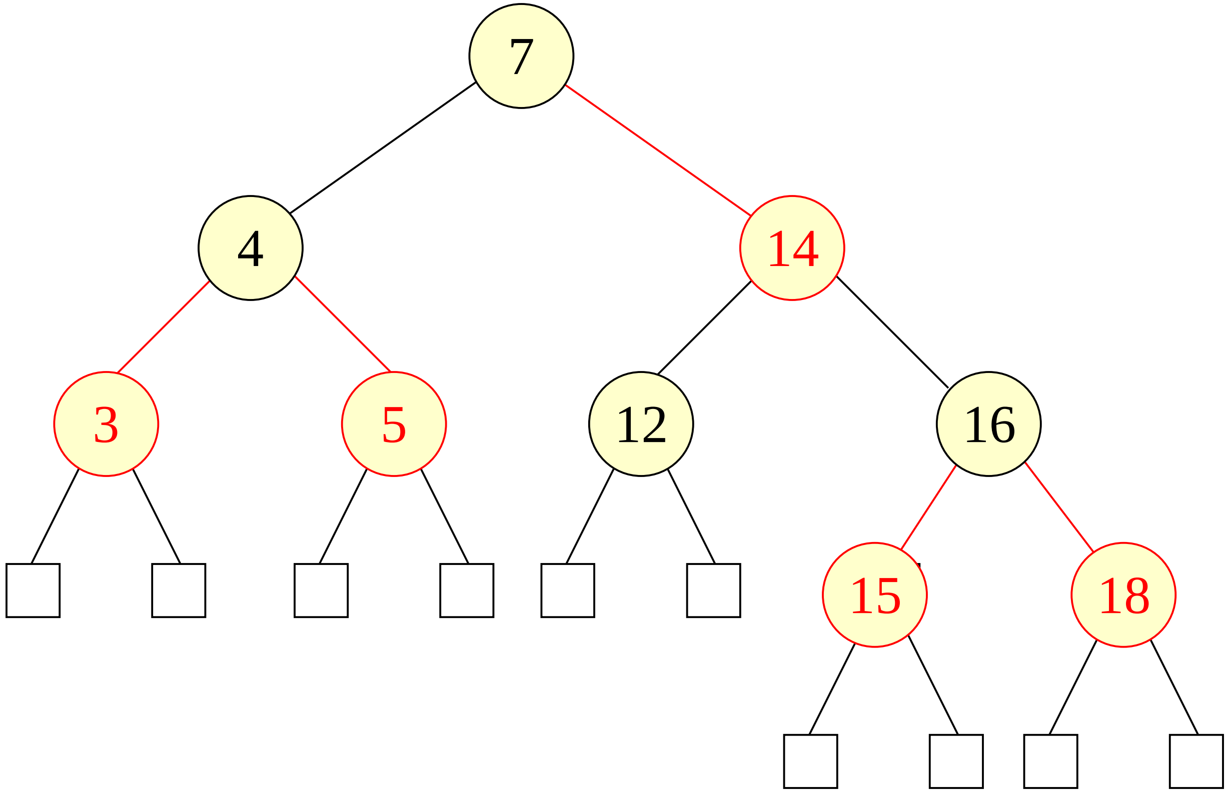
(k) Insert 18



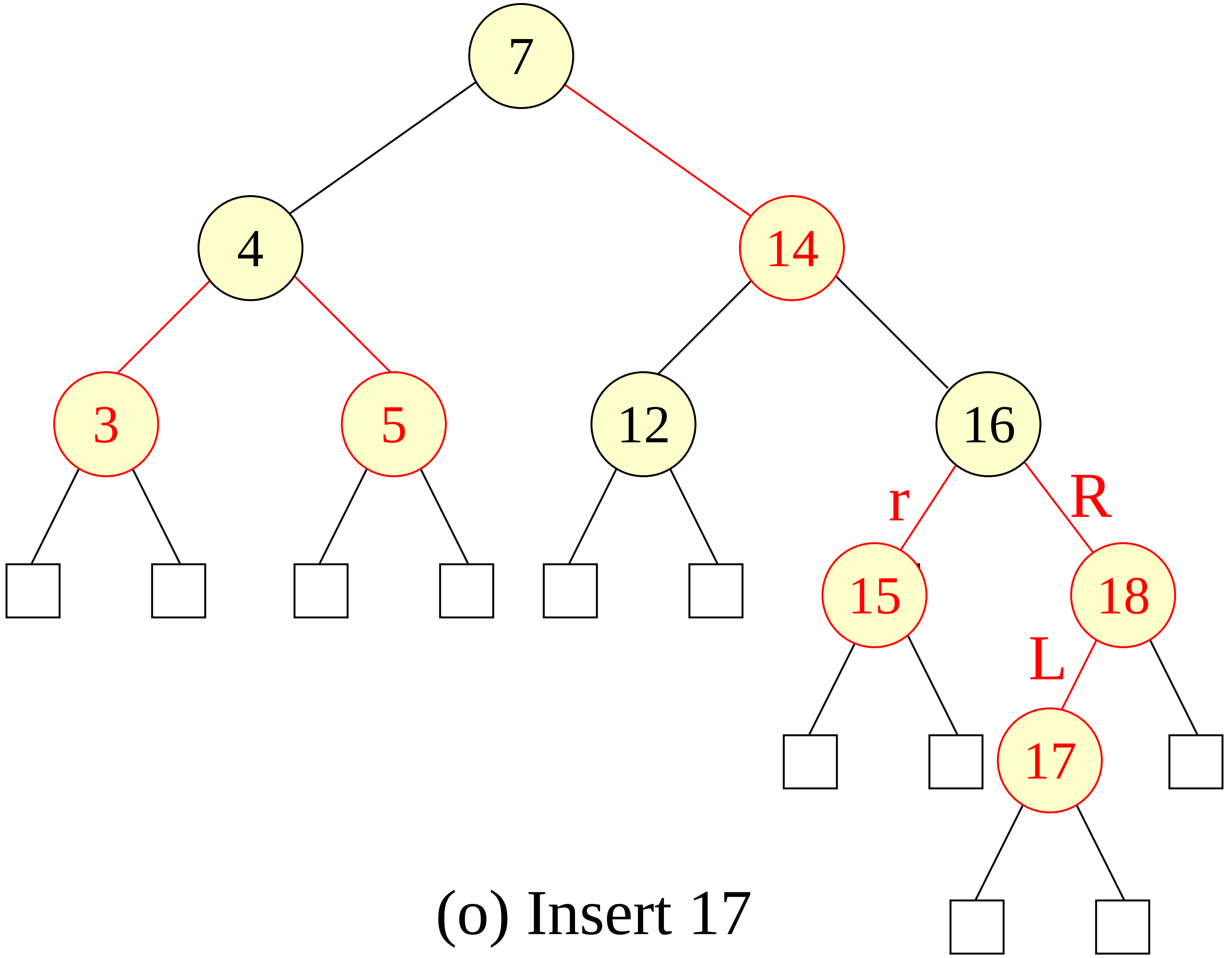
(l)



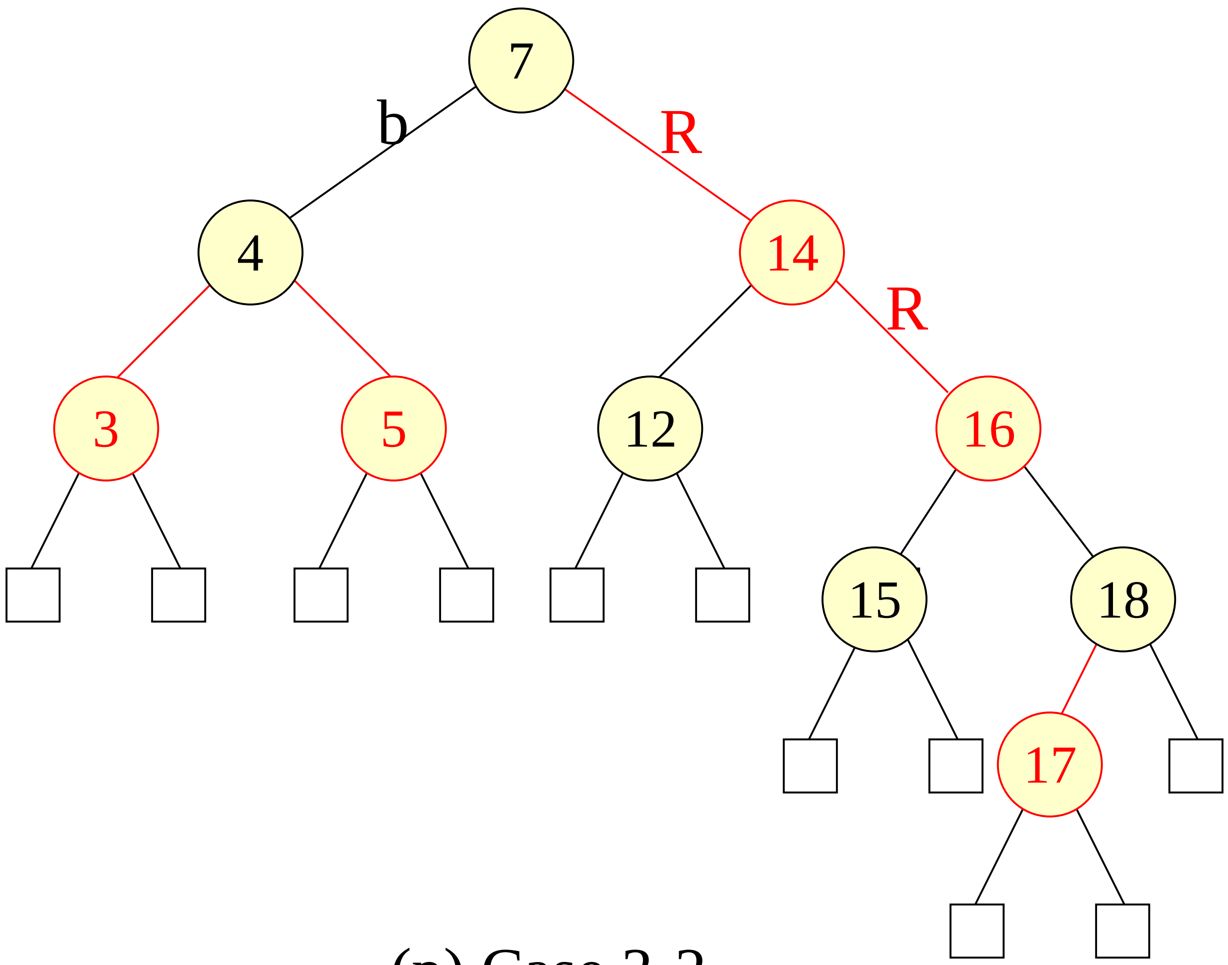
(m) Insert 16



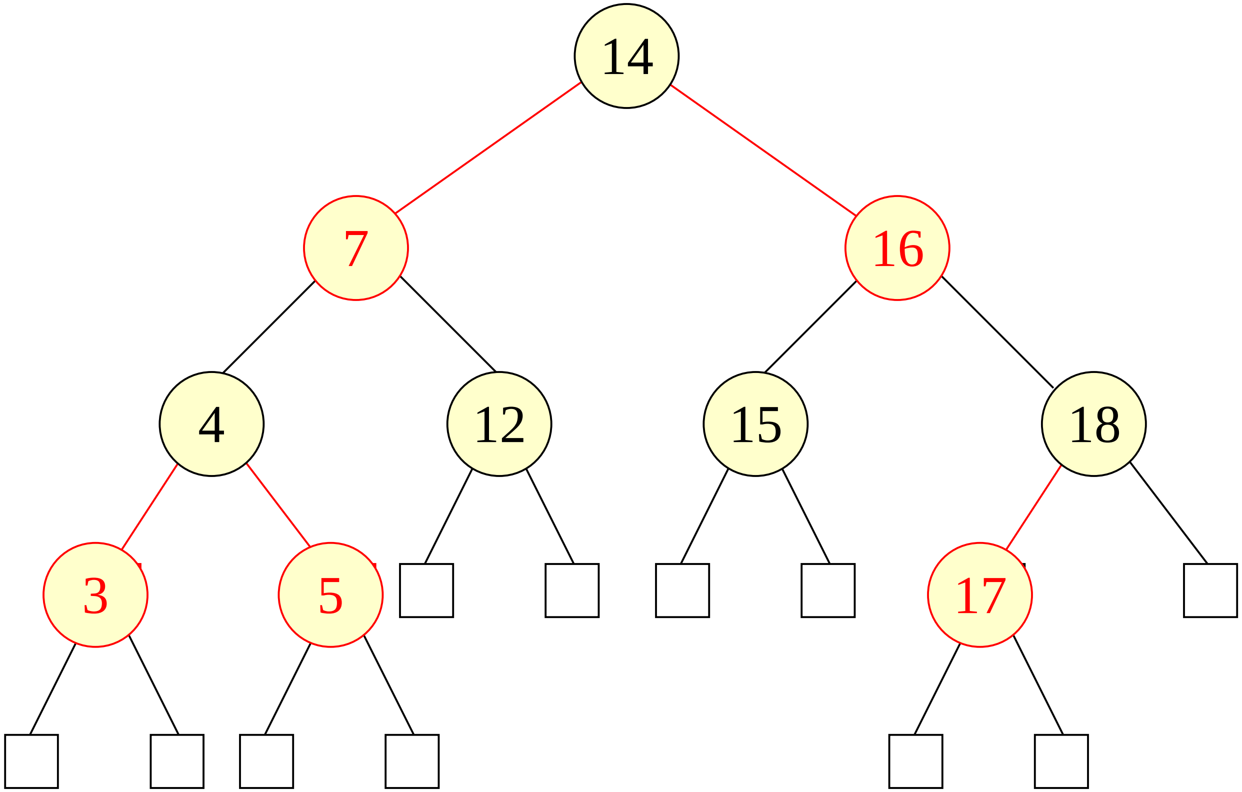
(n)



(o) Insert 17



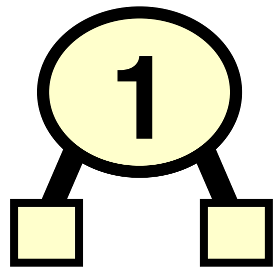
(p) Case 3-2



(q)

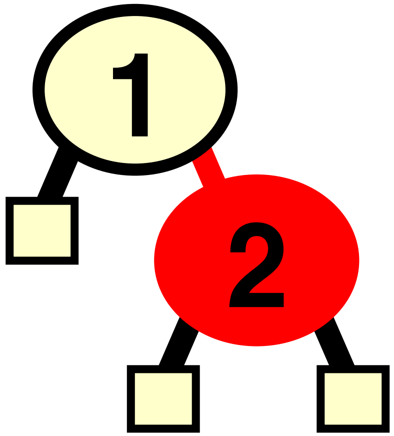
Example 2

Insert 1



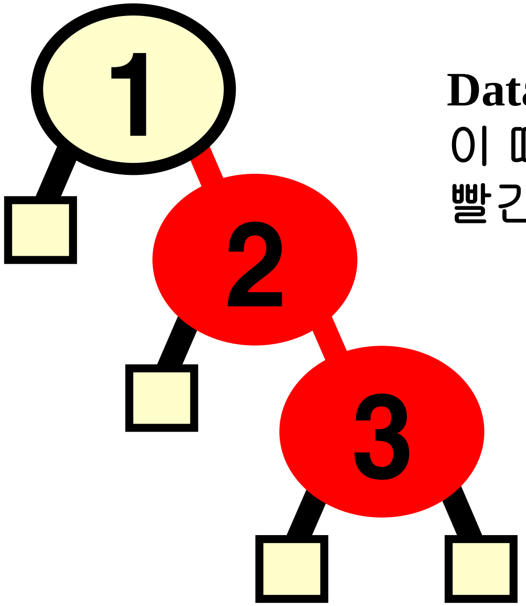
Data 1을 삽입한다.
이 때 **1**은 **root**값이 된다.

Insert 2



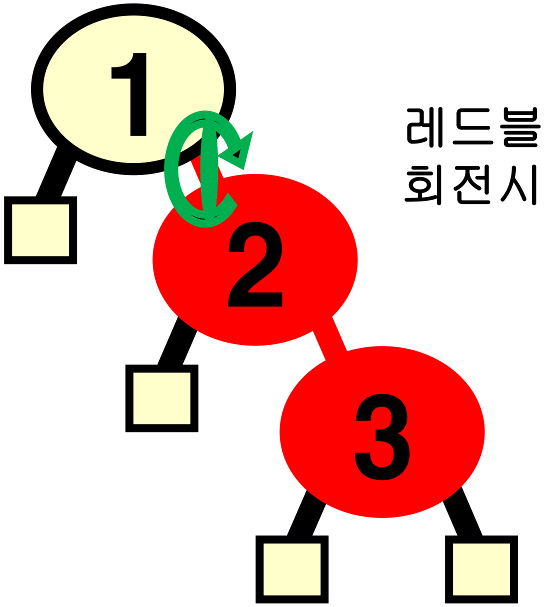
Data 2를 삽입한다.

Insert 3



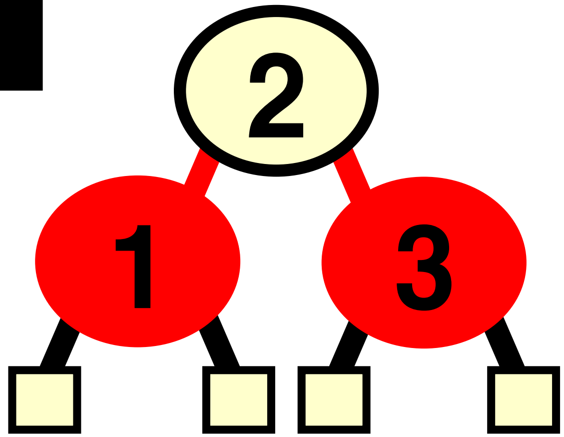
Data 3을 삽입한다.
이 때 2와 3을 연결한 선의 색이 빨간색으로 연속해서 보여진다.

Insert 3



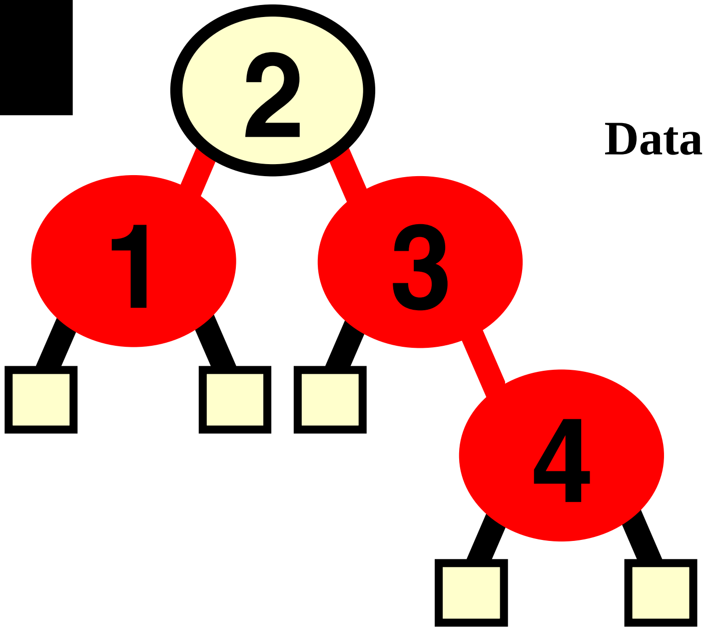
레드블랙트리의 규칙에 어긋나므로
회전시켜준다.

Insert 3



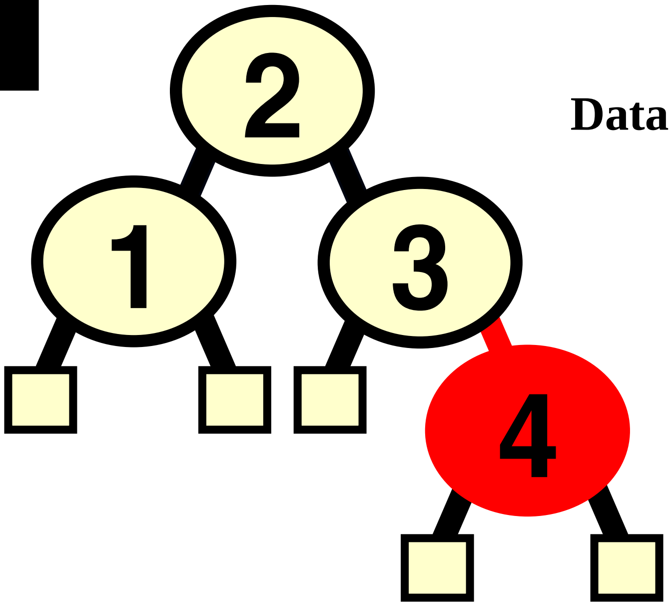
2는 root자리에 오게 되고,
1은 2의 왼쪽 자식,
3은 2의 오른쪽 자식의
자리로 오게 된다.

Insert 4



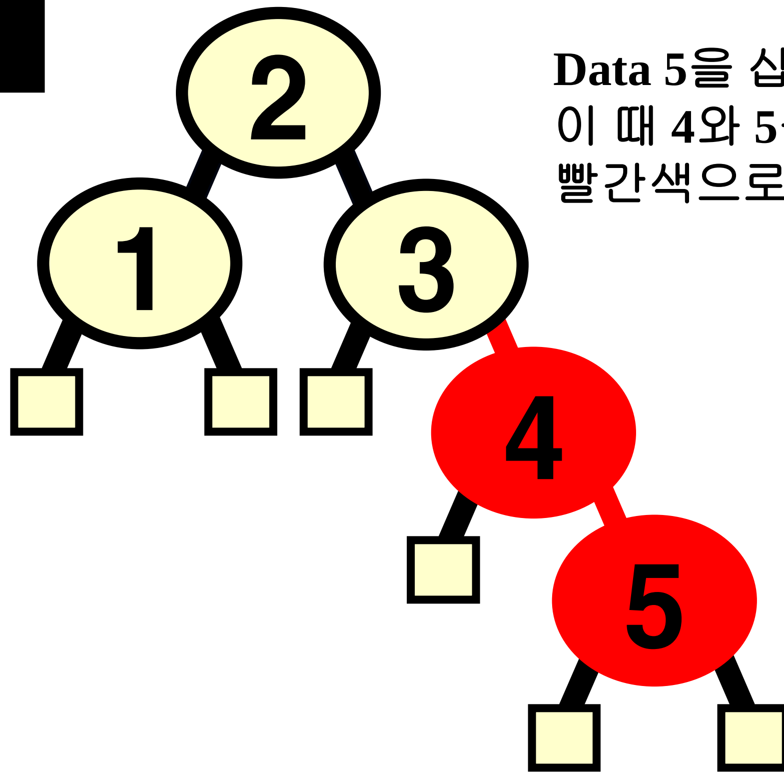
Data 4를 삽입한다.

Insert 4



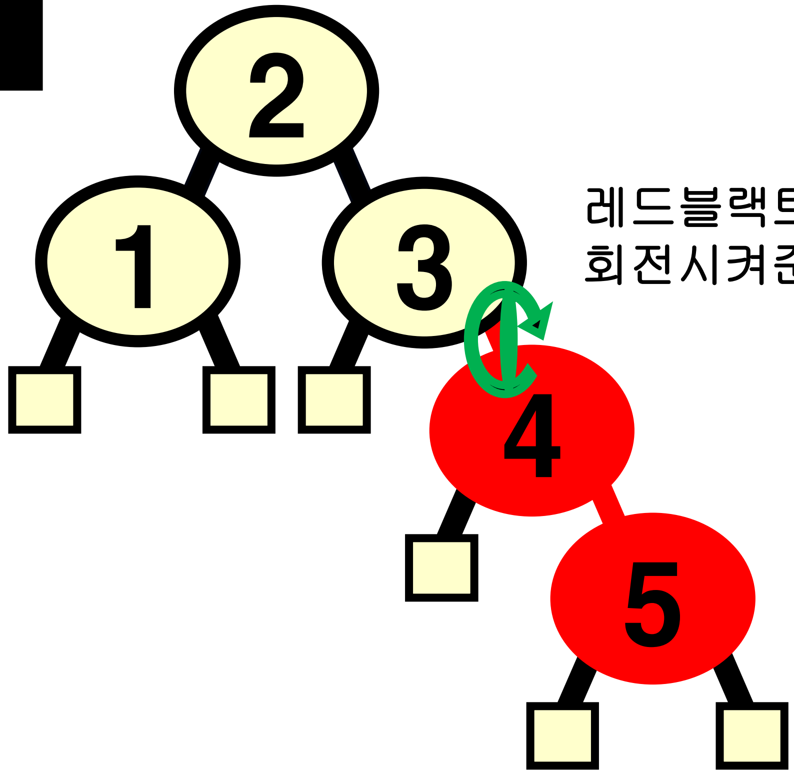
Data 4를 삽입한다.

Insert 5



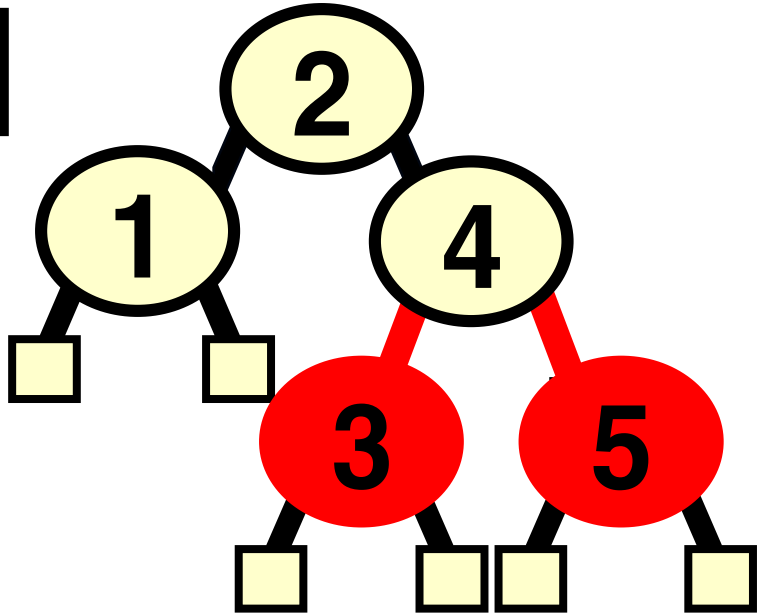
Data 5을 삽입한다.
이 때 **4**와 **5**를 연결한 선의 색이
빨간색으로 연속해서 보여진다.**(RRb)**

Insert 5

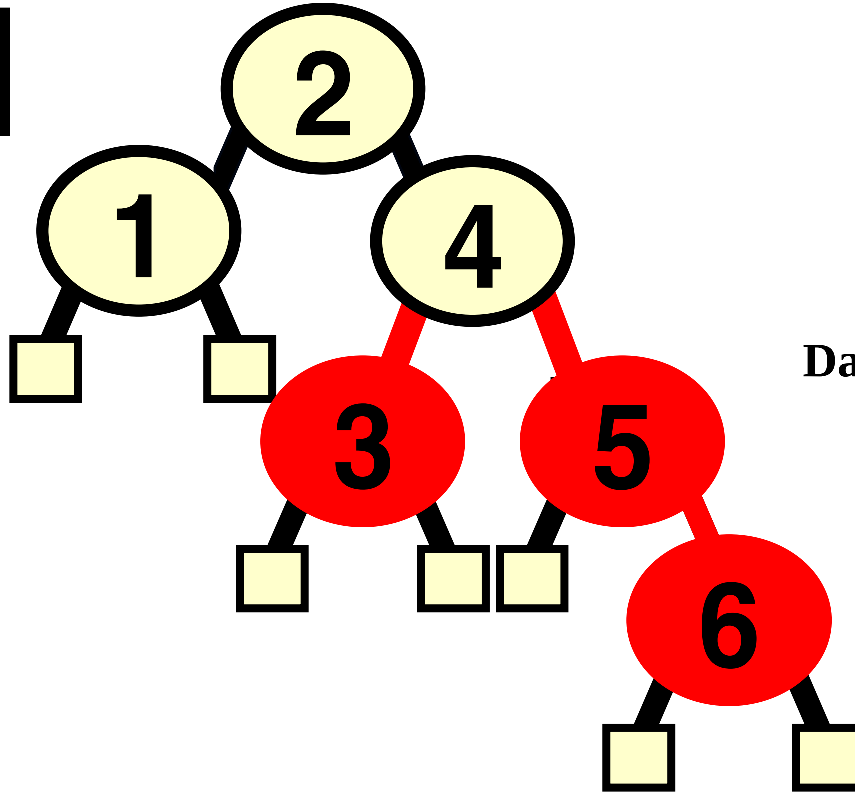


레드블랙트리의 규칙에 어긋나므로
회전시켜준다.

Insert 5

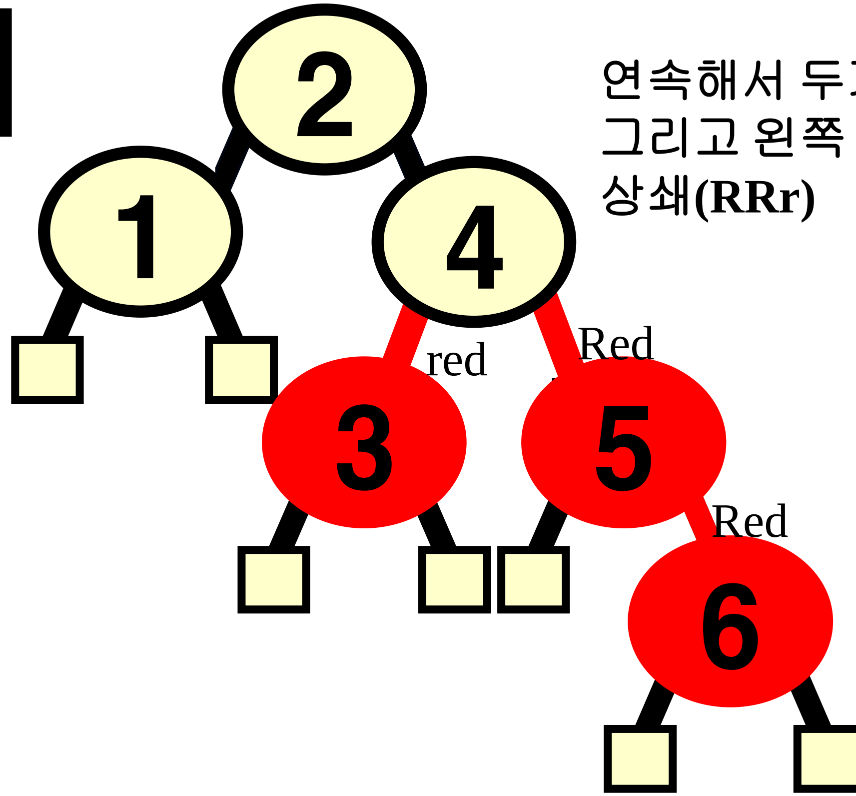


Insert 6



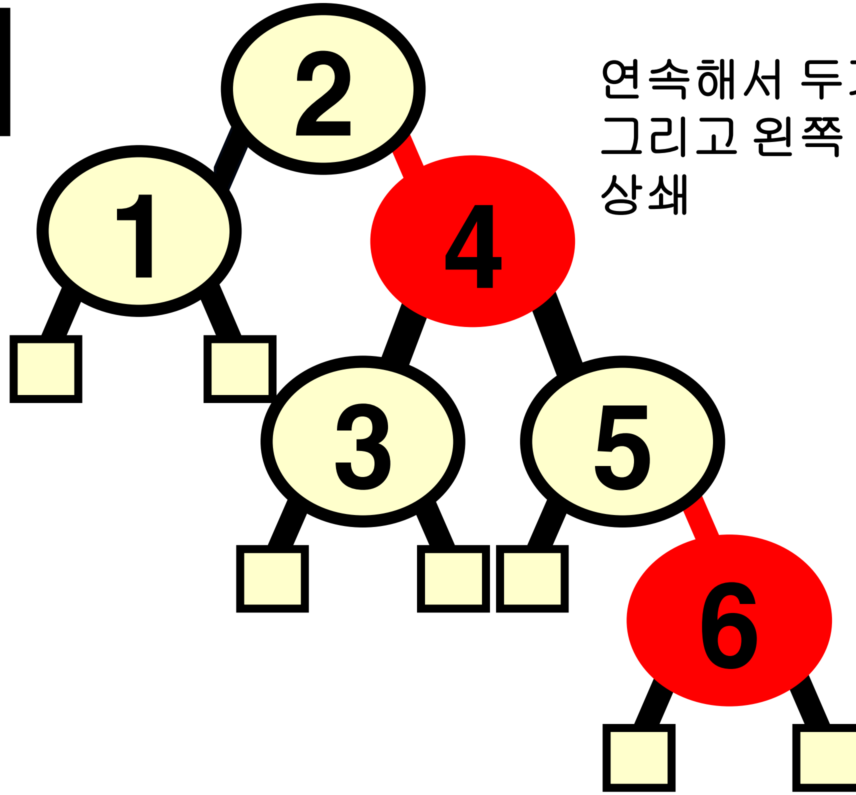
Data 6을 삽입한다.

Insert 6



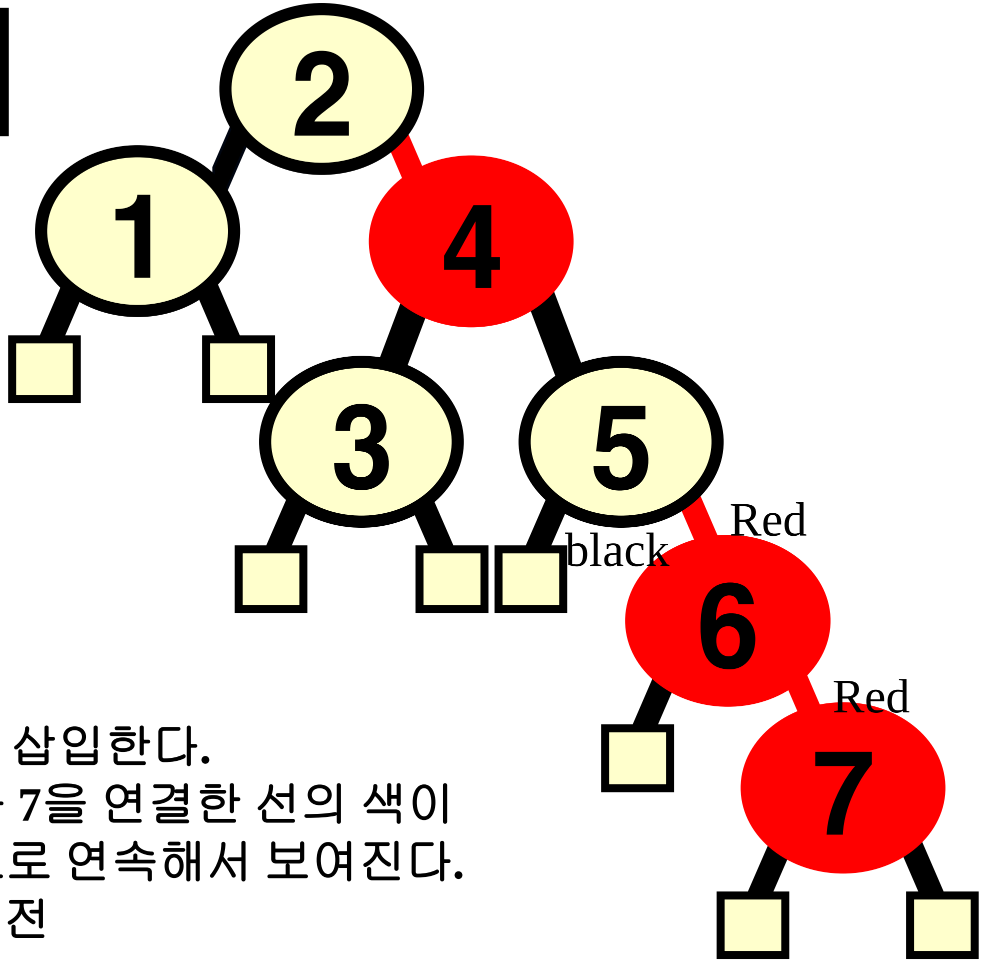
연속해서 두개의 빨간선.
그리고 왼쪽 **red.** 빨간선두개를
상쇄**(RRr)**

Insert 6



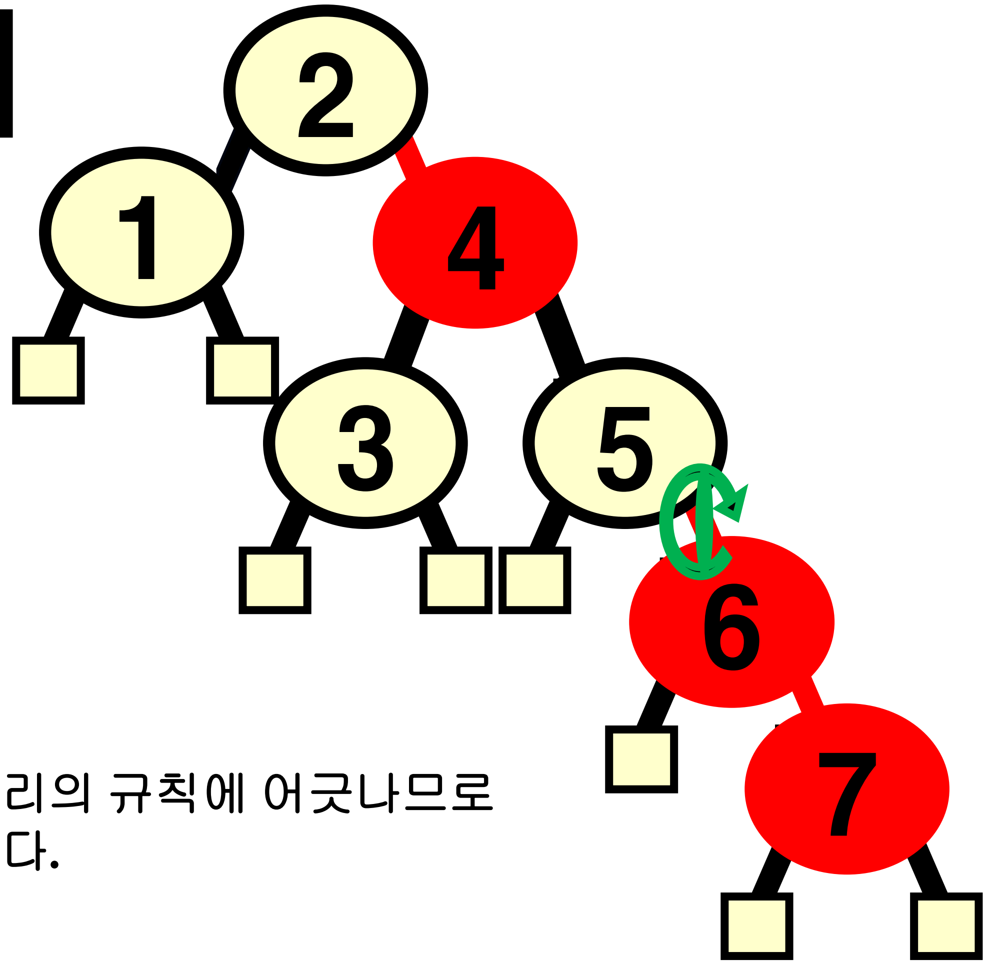
연속해서 두개의 빨간선.
그리고 왼쪽 red. 빨간선두개를
상쇄

Insert 7



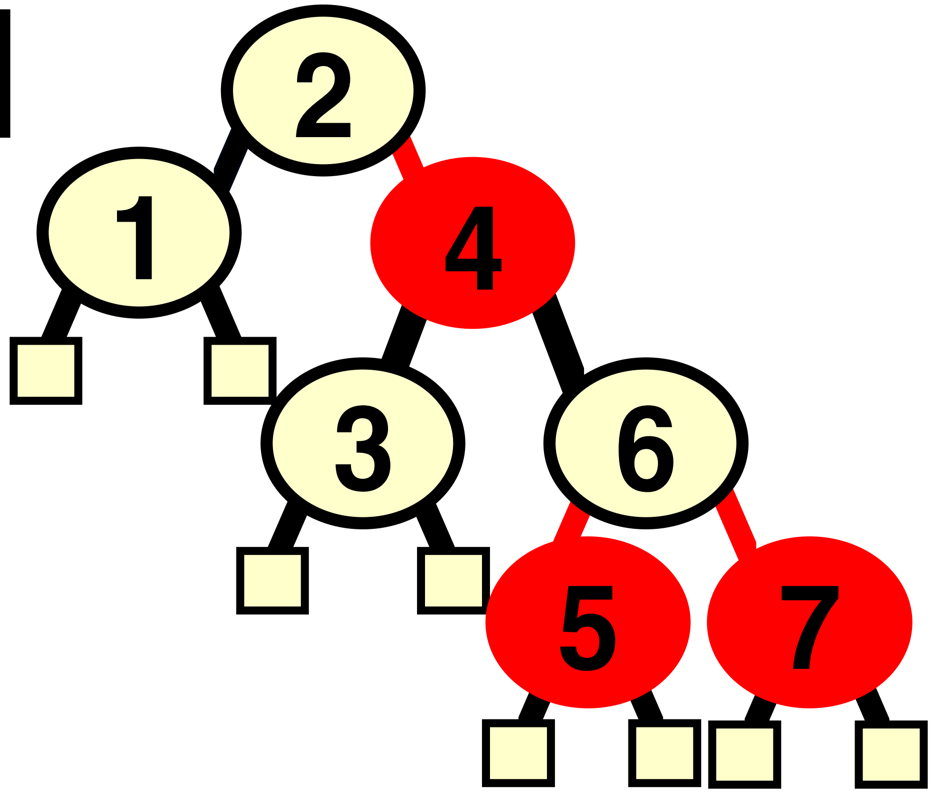
Data 7을 삽입한다.
이 때 **6**과 **7**을 연결한 선의 색이
빨간색으로 연속해서 보여진다.
RRb->회전

Insert 7

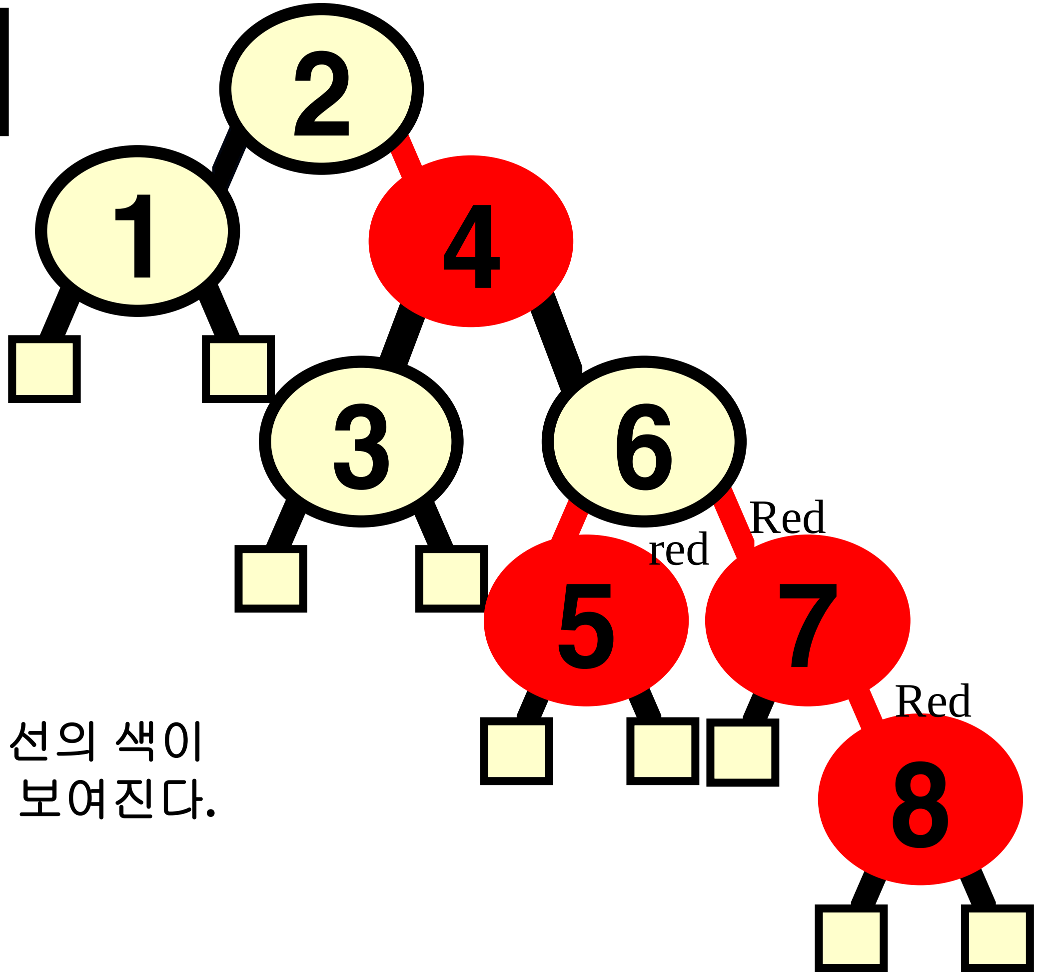


레드블랙트리의 규칙에 어긋나므로
회전시켜준다.

Insert 7

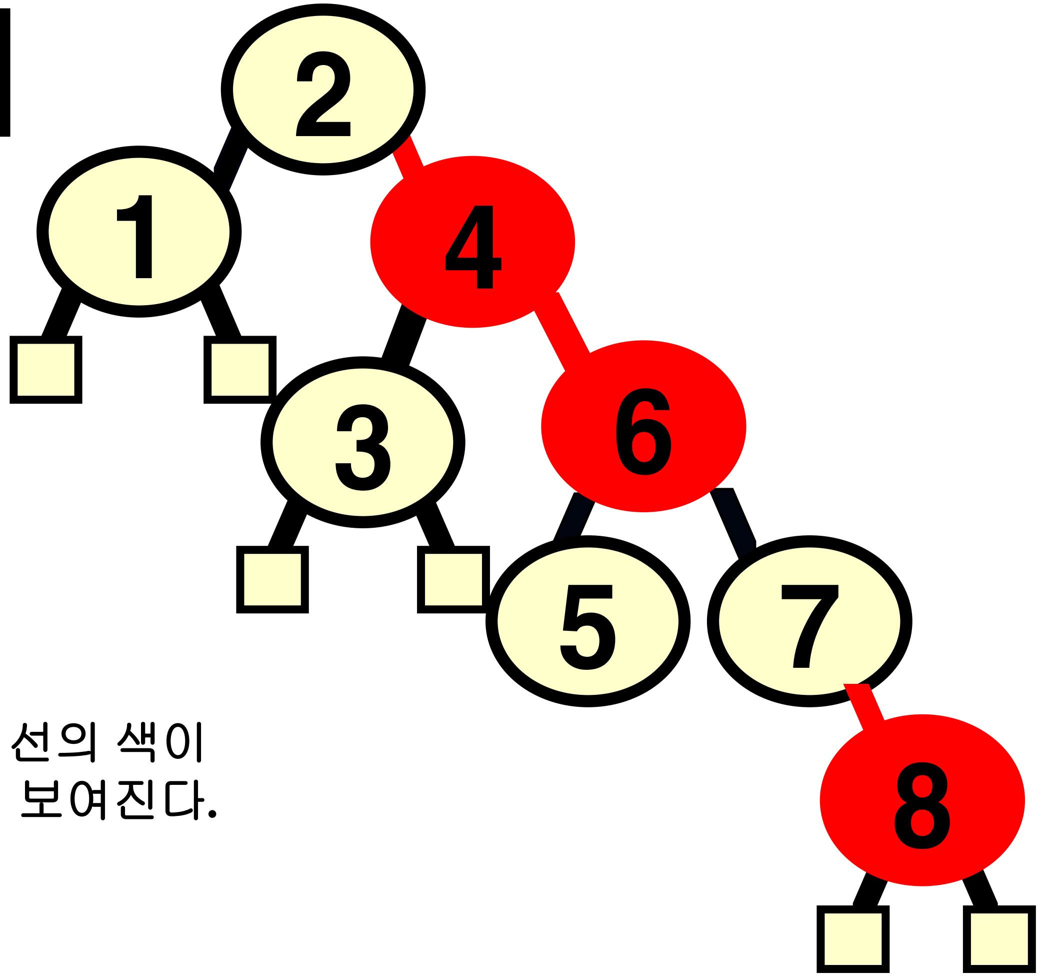


Insert 8



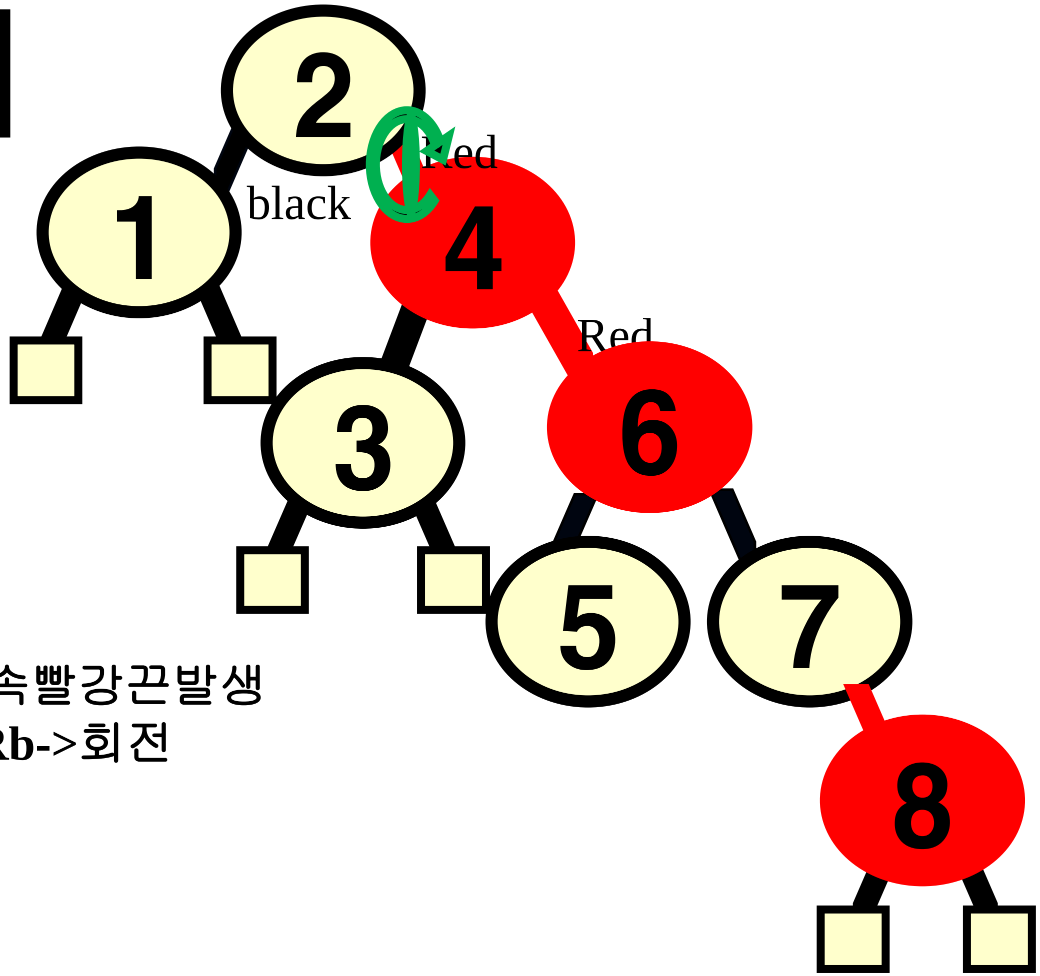
Data 8을 삽입한다.
이 때 **7**과 **8**을 연결한 선의 색이
빨간색으로 연속해서 보여진다.
RRr->빨강끈상쇄

Insert 8



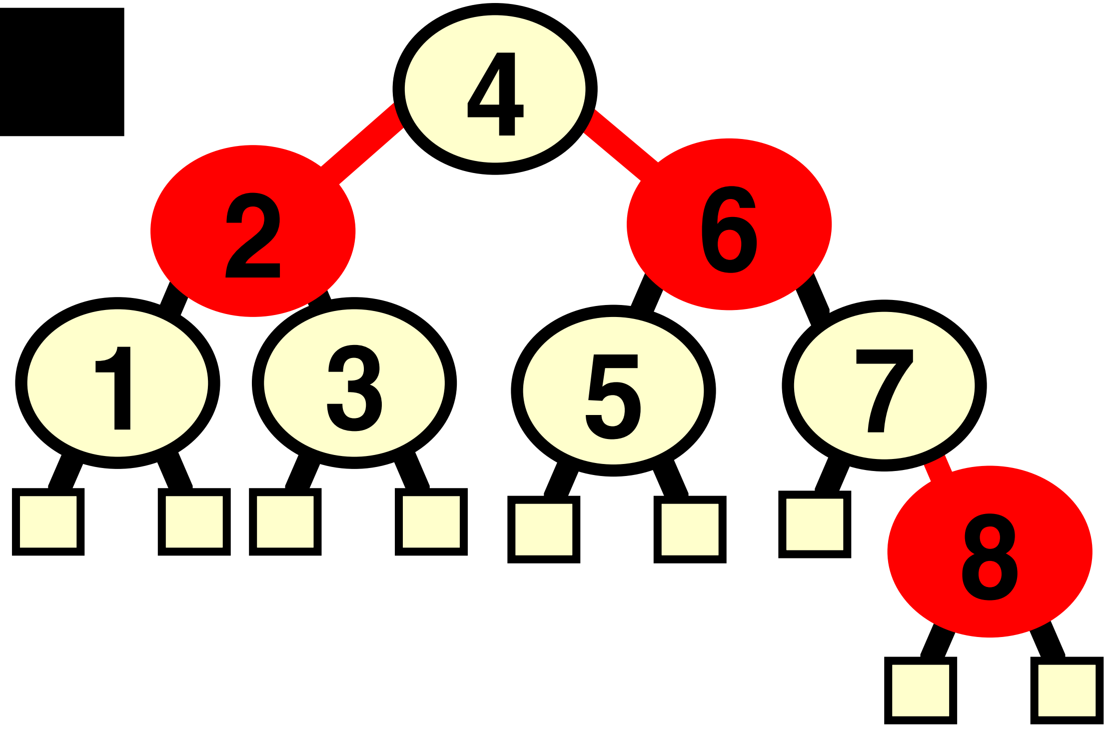
Data 8을 삽입한다.
이 때 **7**과 **8**을 연결한 선의 색이
빨간색으로 연속해서 보여진다.
RRr->빨강끈상쇄

Insert 8

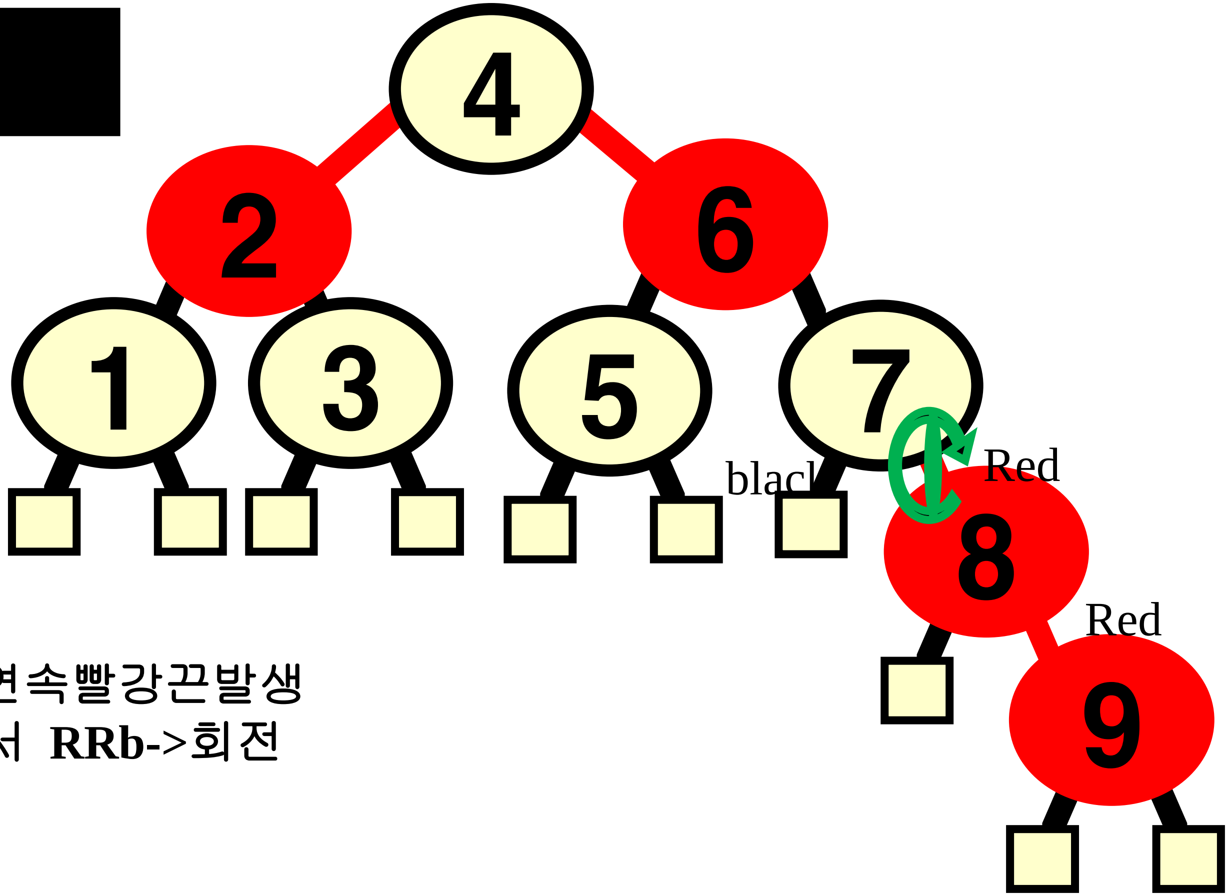


상쇄하니, 두개의 연속빨강끈발생
왼쪽은 **b,** 따라서 **RRb->**회전

Insert 8

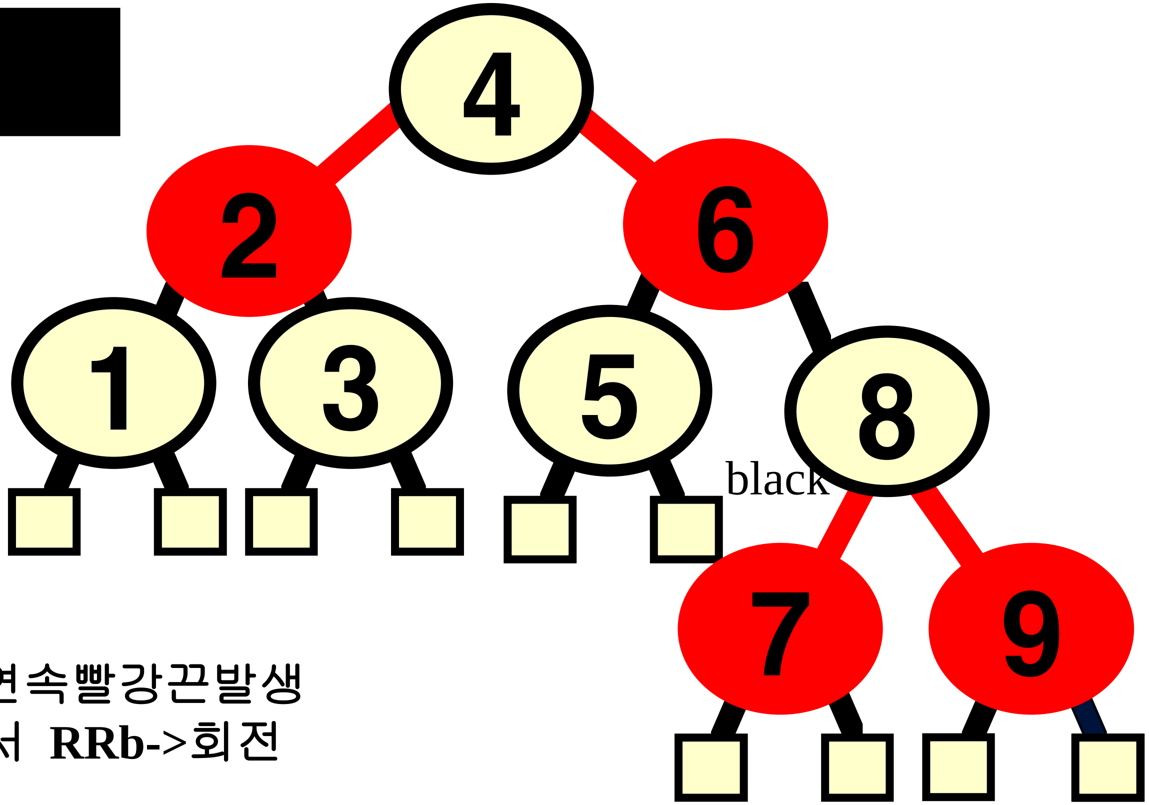


Insert 9



9삽입, 두개의 연속빨강끈발생
왼쪽은 **b,** 따라서 **RRb->**회전

Insert 9



9삽입, 두개의 연속빨강끈발생
왼쪽은 b, 따라서 RRb->회전

레드 블랙 트리

- 사용이유
  - 2-3-4 트리의 복잡한 노드 구조 그리고 복잡한 삽입 삭제 코드
  - 레드 블랙 트리는 이진 탐색트리의 함수를 거의 그대로 사용
  - 2-3-4 트리의 장점인 단일 패스 삽입 삭제가 그대로 레드 블랙 트리에도 적용.
  - 언제 회전에 의해 균형을 잡아야 하는지가 쉽게 판별됨.

레드 블랙 트리의 효율

- 위 예
  - 이진 탐색트리에 10, 20, ..., 60의 순으로 삽입하면 결과는 모든 노드가 일렬로 늘어서서 최악의 효율.
- 탐색 효율
  - 삽입 삭제를 위한 코드의 간결성은 이진 탐색트리와 비슷하면서도
  - 레드 블랙 트리의 높이는 $O(\log_2 N)$에 근접
  - 레드 블랙 트리는 회전에 의해서 어느 정도 균형을 이룸.
  - AVL은 회전시기를 판단하기 위해 복잡한 코드 실행. 회전방법 역시 복잡한 코드 실행. 그에 따를 실행시간 증가
  - 레드 블랙 트리는 빨강 링크의 위치만으로 회전시기를 쉽게 판단, 회전방법도 간단.

Which algorithm is best?

- **Advantages**
  - AVL: relatively easy to program. Insert requires only one rotation.
  - Splay: No extra storage, high frequency nodes near the top
  - RedBlack: Fastest in practice, no traversal back up the tree on insert
- **Disadvantages**
  - AVL: Repeated rotations are needed on deletion, must traverse back up the tree.
  - SPLAY: Can occasionally have O(N) finds, multiple rotates on every search
  - RedBlack: Multiple rotates on insertion, delete algorithm difficult to understand and program

감사합니다.