Announcement

We will have an in-class quiz (Quiz #3) on Thursday, March 17
It is open-book and open-notes.

The question will ask you to create an AVL tree on a given list of numbers. There will be a bonus question on tree traversal – preorder, inorder, and postorder.

Large Integer Multiplication

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long

Such integers are too long to fit a single word of a computer

Therefore, they require special treatment

Consider the multiplication of two such long integers

$$X = x_{n-1}x_{n-2}\cdots x_1x_0 = \sum_{i=0}^{n-1} x_i r^i$$

Classic paper-and-pencil algorithm

$$Y = y_{n-1}y_{n-2}\cdots y_1y_0 = \sum_{j=0}^{n-1} y_j r^j$$

***n*² digit multiplications**

$$XY = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i y_j r^{i+j}$$

Large Integer Multiplication – Divide&Conquer

We want to calculate 23 x 14

Since $23 = 2\cdot 10^1 + 3\cdot 10^0$ **and** $14 = 1\cdot 10^1 + 4\cdot 10^0$

We have

$$23*14 = (2\cdot 10^1 + 3\cdot 10^0)*(1\cdot 10^1 + 4\cdot 10^0)$$
$$= (2*1)10^2 + (3*1 + 2*4)10^1 + (3*4)10^0$$

Which includes four digit multiplications (n^2)

But

Computed already!

$$3*1 + 2*4$$
$$= (2+3)*(1+4) - \boxed{(2*1)} - \boxed{(3*4)}$$

Therefore, we only need three digit multiplications

One Formula

Given $a=a_1a_0$ and b=b_1b_0, compute $c=a$*b

We have

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0 10^0$$

where

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

That means only three digit multiplications are needed to multiply two 2-digit integers

To Multiply Two *n*-digit integers

Assume *n* is even, write

$a = a_1 10^{n/2} + a_0$ **and** $b = b_1 10^{n/2} + b_0$ ***For example, for "1234", a_1= 12, a_0=34, n=4***

Then

$$c = a * b = c_2 10^n + c_1 10^{n/2} + c_0 10^0$$

where

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

To calculate the involved three multiplications – recursion!
Stops when *n*=1

Efficiency

The recurrence relation is

$$T(n) = 3T(n/2) \textbf{ for } n > 1, T(1) = 1$$

Solving it by backward substitution for $n=2^k$ yields

$$\begin{aligned} T(2^k) &= 3T(2^{k-1}) = 3^2 T(2^{k-2}) \\ &= 3^k T(2^{k-k}) = 3^k \end{aligned}$$

Therefore, $\boxed{T(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585} < n^2}$

Reading Assignments

Chapter 5.4 Strassen's Matrix Multiplication

Chapter 5.5 Closest pair and convex-hull by divide-and-conquer

Transform and Conquer

Solve problem by transforming into:

a more convenient instance of the same problem (*<u>instance simplification</u>*)

- presorting
- Gaussian elimination

a different representation of the same instance (*<u>representation change</u>*)

- balanced search trees
- heaps and heapsort

a different problem with available algorithms (*<u>problem reduction</u>*)

- reductions to graph problems

Instance Simplification - Presorting

Solve problem by transforming into another simpler/easier instance of the same problem

Presorting:

Why: Many problems involving lists are easier when list is sorted.

When: A preprocessing step if multiple operations of following are needed:

- searching
- computing the median (selection problem)
- computing the mode
- finding repeated elements

Example 1: Searching Problem

Find a value *v* in A[1],…A[*n*].

Brute-force search:

- Sequential search with
  - worst case $\Theta(n)$.

Presorted search $T(n)=T_{sort}(n)+T_{search}(n)$

$$= \Theta(n \log n) + \underbrace{\Theta(\log n)}_{\textit{Binary search}} = \Theta(n \log n)$$

- **For a single search, the presorted search is inferior to the brute-force search**
- **For repeated searches in the same list, presorted search may be more efficient because the sorting need not be repeated**

Example 2: Selection Problem

Find the k^{th} smallest element in A[1],…A[n]. Special cases:

- *minimum*: $k = 1$
- *maximum*: $k = n$
- *median*: $k = \lceil n/2 \rceil$

Brute-force $\Theta(n)$

Partition-based algorithm (Variable decrease & conquer):

- worst case: $T(n) = T(n-1) + (n+1) \rightarrow \Theta(n^2)$
- best case: $\Theta(n)$
- average case: $T(n) = T(n/2) + (n+1) \rightarrow \Theta(n)$

Presorting-based algorithm

- sort list
- return A[k]
- $\Theta(n\log n) + \Theta(1) = \Theta(n\log n)$

Notes on Selection Problem

Partition-based algorithm (Variable decrease & conquer):

- worst case: $T(n) = T(n-1) + (n+1) \rightarrow \Theta(n^2)$
- best case: $\Theta(n)$
- average case: $T(n) = T(n/2) + (n+1) \rightarrow \Theta(n)$

Presorting-based algorithm: $\Omega(n\lg n) + \Theta(1) = \Omega(n\lg n)$

Special cases of max, min: brute-force algorithm is better $\Theta(n)$

Example 3: Finding Repeated Elements/Array Uniqueness

Presorting-based algorithm:

- Sort the array
- Scan array to find repeated <u>adjacent</u> elements:

ALGORITHM $\text{Pr}\,esortUniqueElements(A[0,\cdots,n-1])$

//Input : An array $\text{A}[0,\cdots,\text{n}-1]$ of orderable elements

//Output : Returns "true" if no equal elements, otherwise return "false"

sort the array A

for $i \leftarrow 0$ to n-2 do

    if $A[i] = A[i+1]$ return false

return true

Example 3: Finding Repeated Elements/Array Uniqueness

Brute force algorithm:

- Worst case: $\Theta(n^2)$

ALGORITHM $UniqueElements(A[0..n-1])$
for $i \leftarrow 0$ **to** n-2 **do**
    for $j \leftarrow i+1$ **to** n-1 **do**
        if $A[i] = A[j]$ **return false**
return true

Presorting-based algorithm:

- Sort the array: $\Theta(n\log n)$
- scan array to find repeated <u>adjacent</u> elements: $\Theta(n)$

$$\Theta(n\log n)$$

<u>Conclusion:</u> Presorting yields <u>significant</u> improvement

Example 4: Computing A Mode

A mode is a value that occurs most often in a given list of numbers

For example: the mode of [5, 1, 5, 7, 6, 5, 7] is 5

Brute-force technique: construct a list to record the frequency of each distinct element

- In each iteration, the *i*-th element is compared to the stored distinct elements. If a matching is found, its frequency is incremented by 1. Otherwise, current element is added to the list as a distinct element
- Worst case complexity $\Theta(n^2)$, when all the given n elements are distinct

Example 4: Computing A Mode With Presorting Algorithm

ALGORITHM $PresortMode(A[0..n-1])$

$Step1$: Sort the array A

$Step2 : i \leftarrow 0$

$modfrequency \leftarrow 0$ // highest frequency seen so far

while $i \leq n-1$ do

$\quad runlength \leftarrow 1; runvalue \leftarrow A[i]$

$\quad$ while $i + runlength \leq n-1$ and $A[i + runlength] = runvalue$

$\quad\quad runlength \leftarrow runlength + 1$

$\quad$ if $runlength > modefrequency$

$\quad\quad modefrequency \leftarrow runlength; modevalue \leftarrow runvalue$

$\quad i \leftarrow i + runlength$

return $modevalue$

How many elements have the same value

Example 4: Complexity of *PresortMode()*

Step1: Sorting $\Theta(n\log n)$

Step2: $\Theta(n)$ since each element will be visited once for comparison

Overall complexity of *presortMode* is $\Theta(n\log n)$

Much more efficient than the brute-force algorithm $\Theta(n^2)$

Summary: Presorting

Solve problem by transforming into another simpler/easier instance of the same problem

For a single operation:

- Searching: $\Theta(n \log n)$ inferior to brute-force search $\Theta(n)$
- Selection problem: $\Theta(\text{n} \log n)$ inferior to Partition-based selection $\Theta(n)$
- Finding repeated elements: $\Theta(n \log n)$ better than brute-force $\Theta(n^2)$
- computing the mode: $\Theta(n \log n)$ better than brute-force $\Theta(n^2)$

For multiple operations on the same list, presorting is preferred

Efficient sorting algorithms should be employed such as MergeSort.

Representation Change – Balanced Binary Search Trees

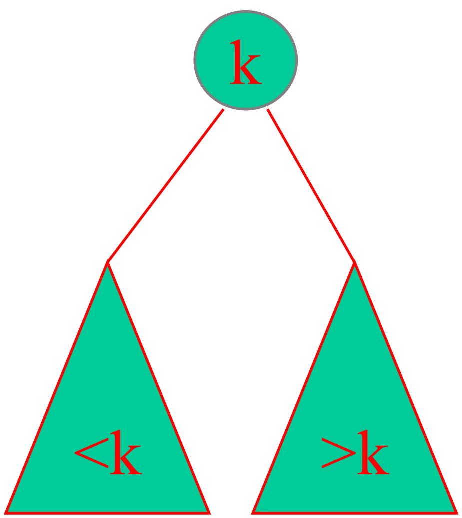
Search a Key in a Binary Search Tree

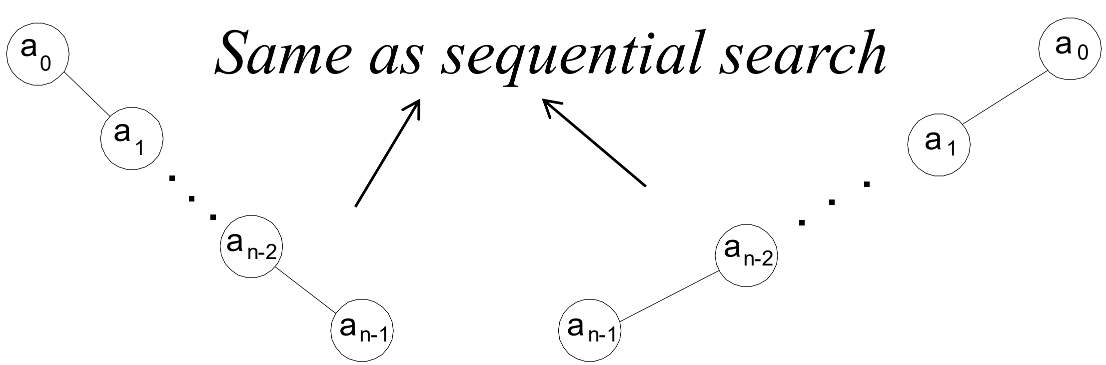
Basic operation: *key comparison*

\# of comparisons in the worst case: $h + 1$

$$\log|V| \leq h \leq |V| - 1$$

Worst case: the tree degrades to a singly linked list Θ(|V|)
Average case: Θ(log|V|)



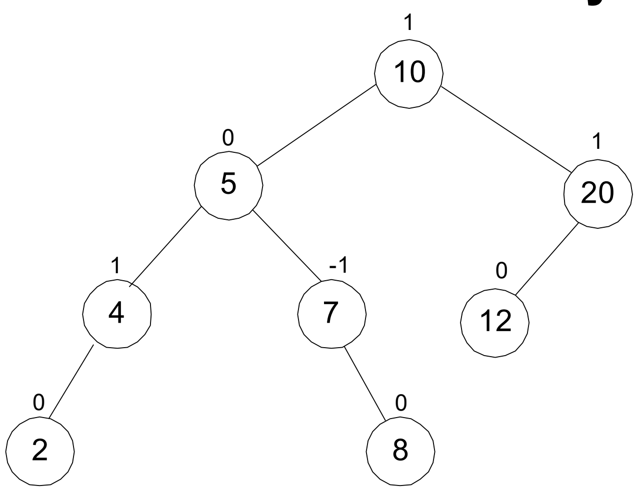


(a) ***How we can improve it?*** (b)

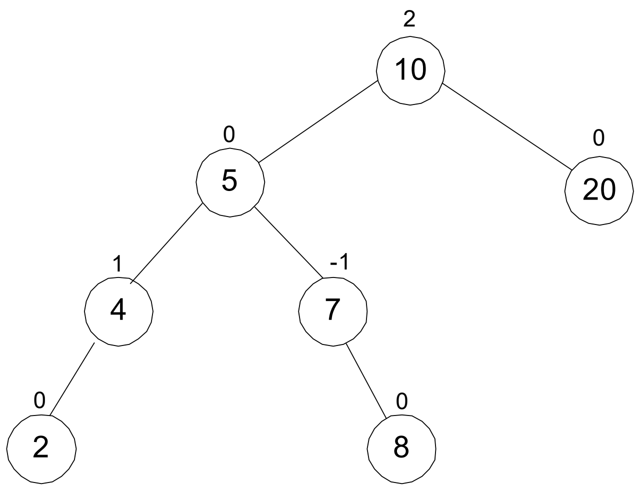
Representation Change – Balanced Binary Search Trees (AVL Trees)

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information.“

AVL tree is a **balanced** binary search tree.



An AVL tree



Not an AVL tree

The number shown above the node is its ***balance factor***

balance factor = height of left subtree - height of right subtree

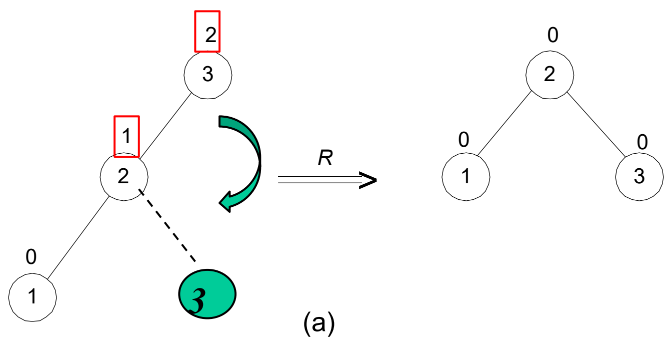
For an AVL tree, |balance factor| <=1

Maintain the Balance of An AVL Tree

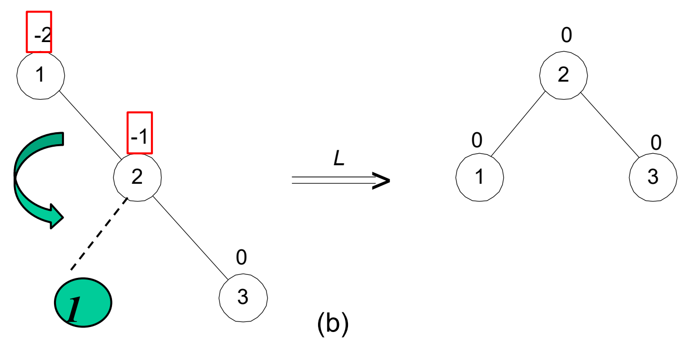
- **When?**
  - Insert a new node or delete a node may make it unbalanced – the balance factors of one or more nodes become +2 or -2.
- **How?**
  - By *rotation* operations
  - Four types of rotations
    - – two of them are mirror images of the other two
- **Where?**
  - Rotate a subtree rooted at the unbalanced node (whose *balance factor* has become either +2 or -2) closest to the change

Four Types of Rotations for Three-Node AVL Trees

Case 1: Balance factors of the unbalanced node and its child have same sign



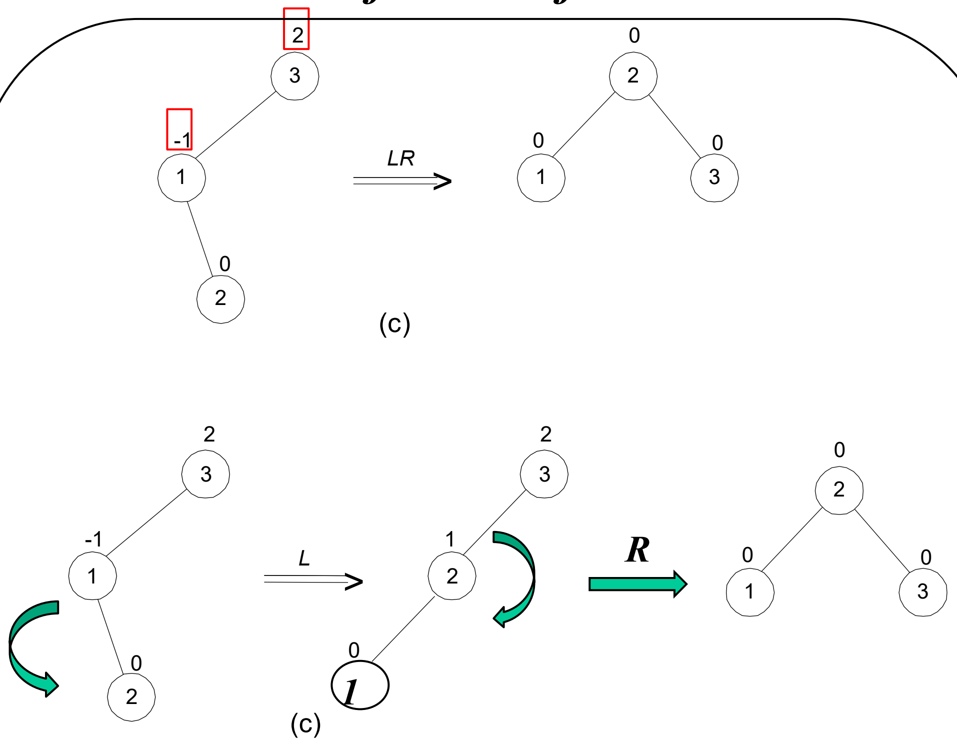
Type 1: Single right rotation



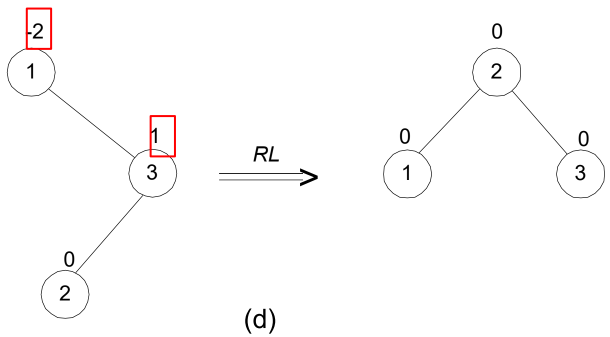
Type 2: Single left rotation

Four Types of Rotations for Three-Node AVL Trees

Case 2: Balance factors of the unbalanced node and its child have different signs



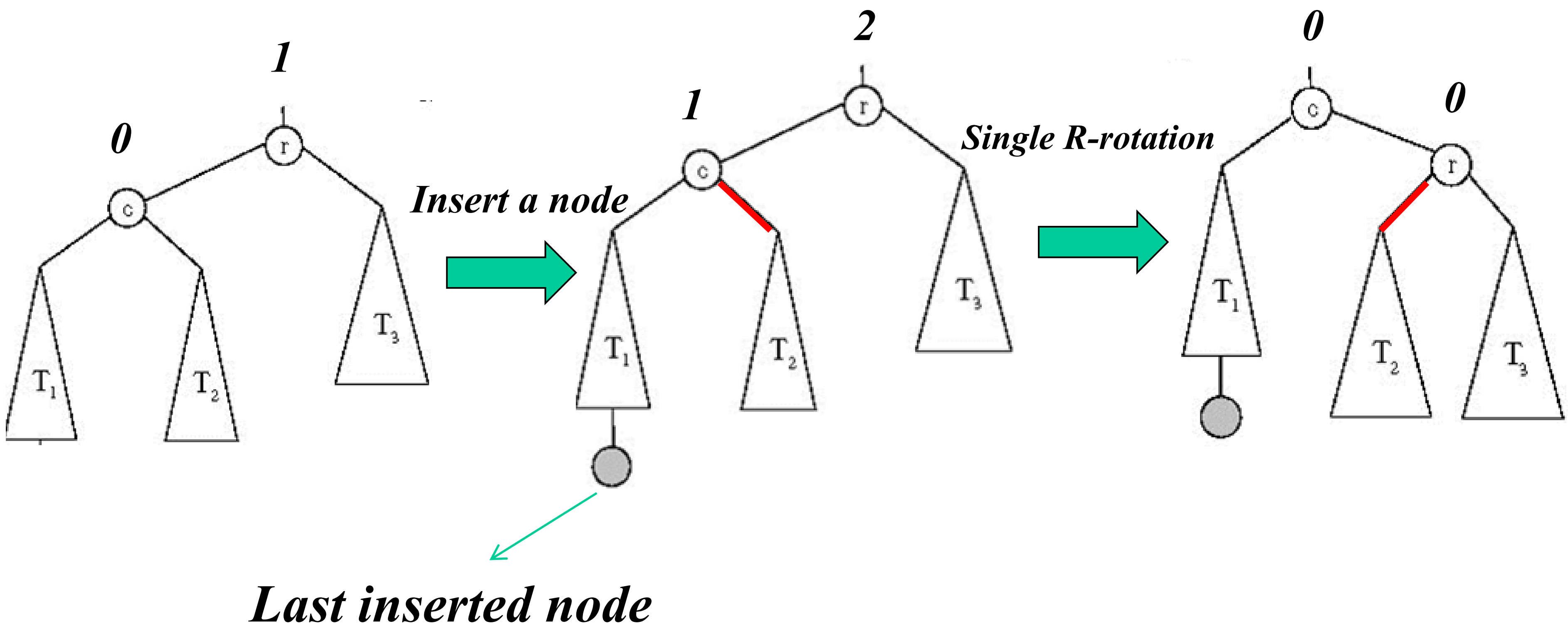
Type 3: Double left-right rotation



Type 4: Double right–left rotation

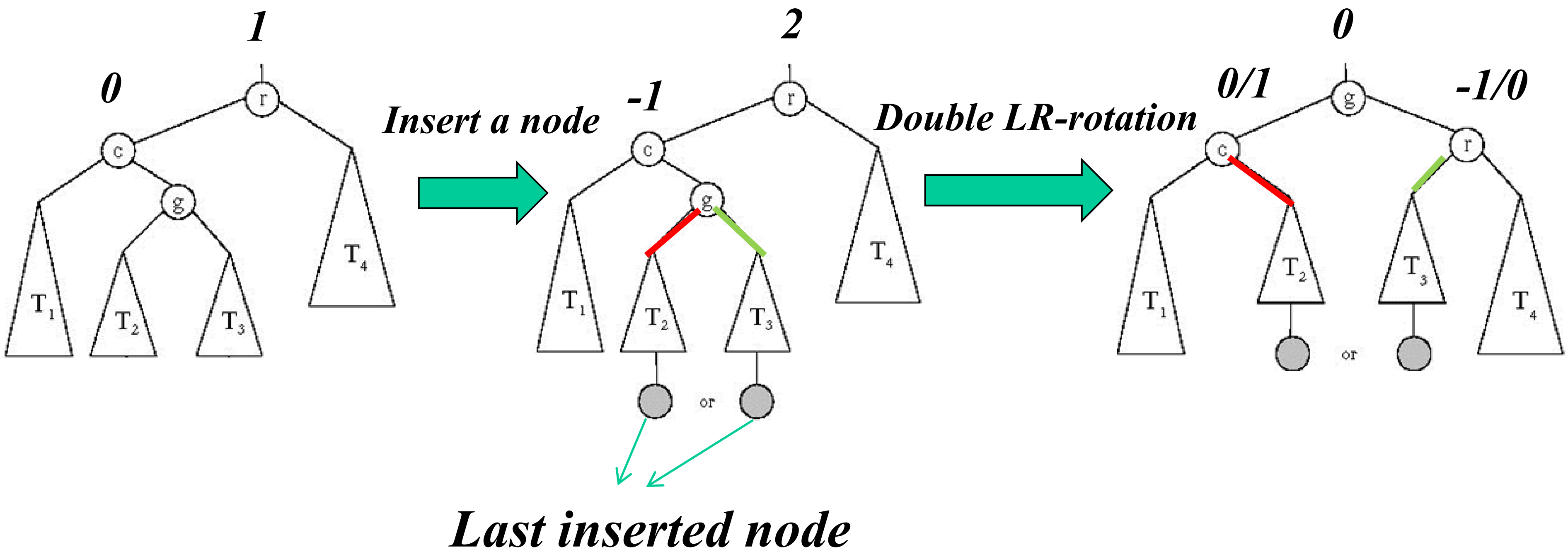
General Case: Single R-rotation

Height(T_1)=Height(T_2)=Height(T_3)



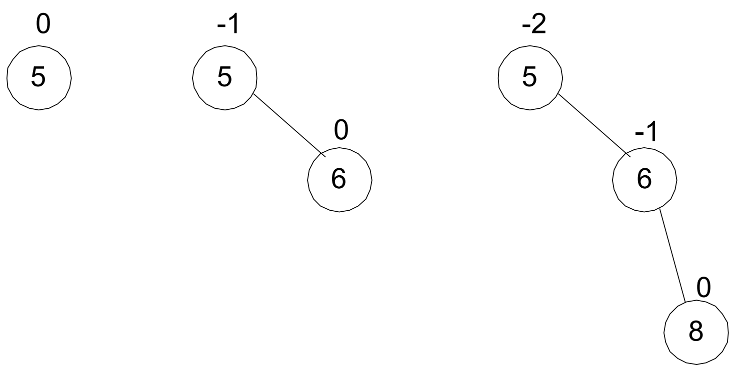
$$T_1 < c < T_2 < r < T_3$$

General Case: Double LR-rotation

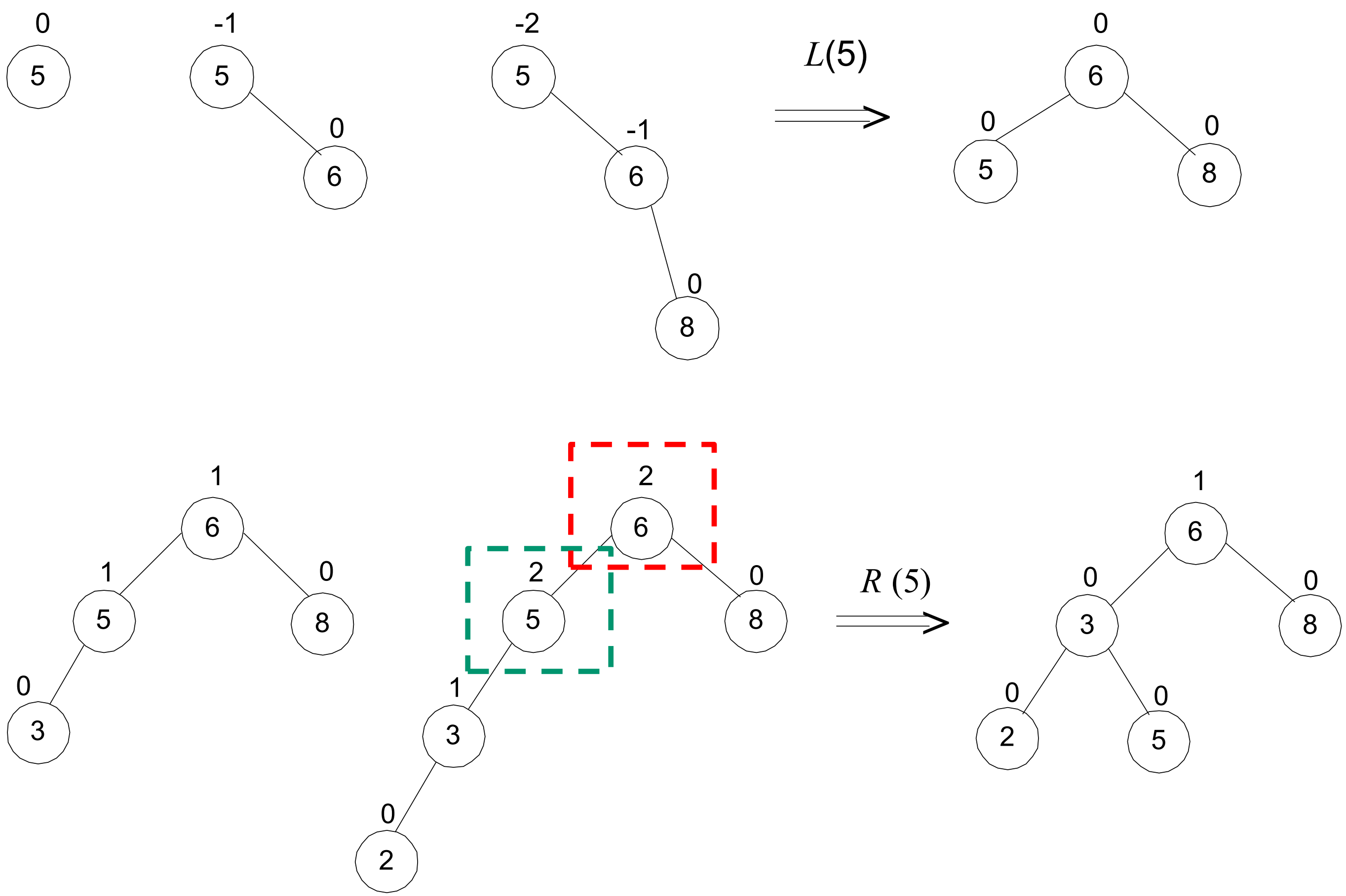


$$T_1 < c < T_2 < g < T_3 < r < T_4$$

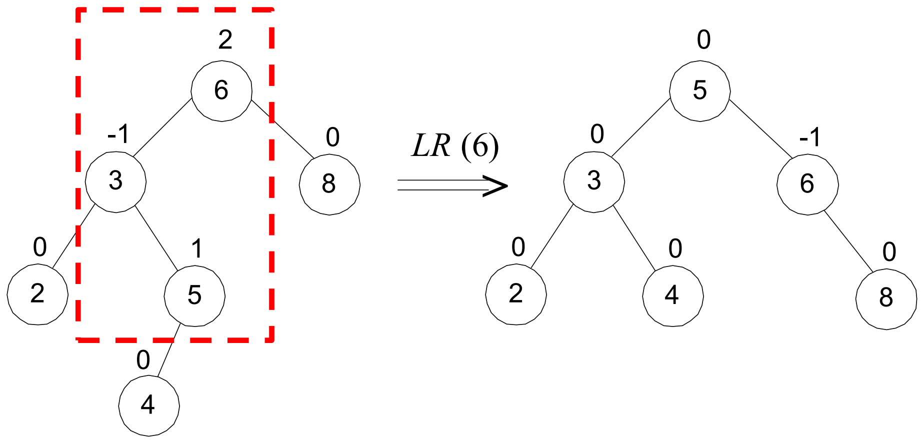
Example: Construct an AVL Tree for the List [5, 6, 8, 3, 2, 4, 7]

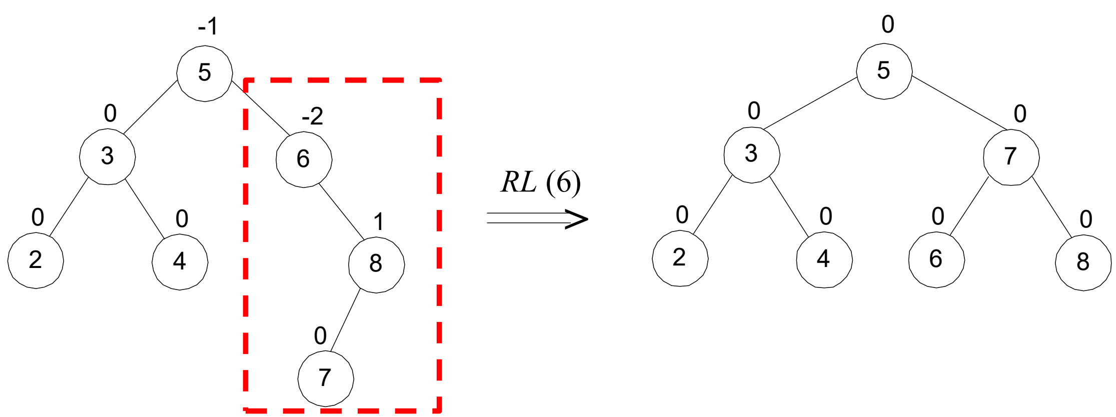


Example: Construct an AVL Tree for the List [5, 6, 8, 3, 2, 4, 7]



Continued
[5, 6, 8, 3, 2, 4, 7]





Notes on AVL Tree

Rotations can be done in constant time $\Theta(1)$

Rotations guarantee an AVL tree

- A binary search tree
- A balanced tree

The height (*h*) of an AVL tree with *n* nodes is bounded by

$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2 (n+2) - 1.3277$$

<u>average:</u> $1.01 \log_2 n + 0.1$ **for large *n***

Operations in an AVL Tree

Searching: $\Theta(\log n)$

Insertion: a new node is inserted at the leaf position

- Searching $\Theta(\log n)$
- Rebalance (bottom up) $\Theta(\log n)$

Deletion:

- Searching: $\Theta(\text{logn})$
- Deletion:
  - A leaf or a non-leaf node with only one child, remove it. $\Theta(1)$
  - Otherwise, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node. $\Theta(\text{logn})$
- Rebalance $\Theta(\log n)$

Drawbacks: need rotation frequently to rebalance the tree

Other Search Trees

Self-balanced BST

- Red-black trees (height of subtrees is allowed to differ by up to a factor of 2: $\frac{h_l}{h_r} \leq 2$ or $\frac{h_r}{h_l} \leq 2$)

Self-optimized BST

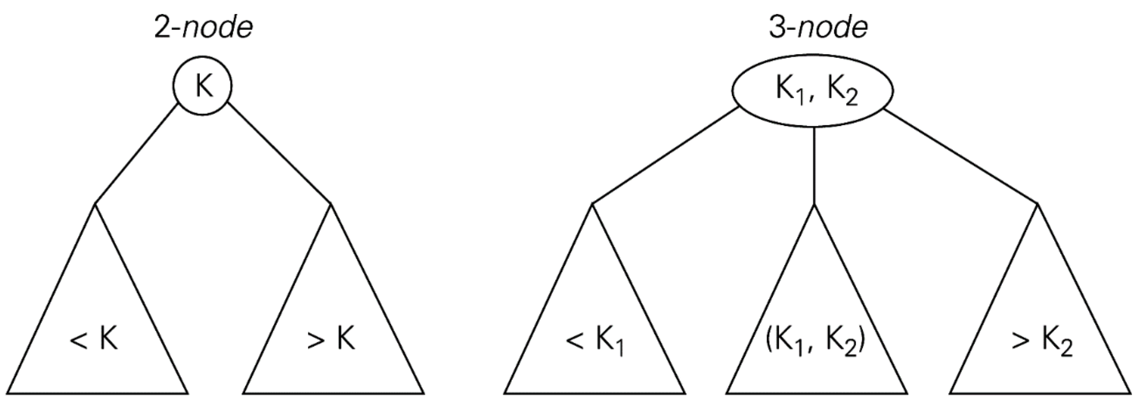
- Splay trees: move the recent visited vertex to root so that recently accessed elements are quick to access again

Multiway search trees

- 2-3 trees, 2-3-4 trees and B-trees (not a binary tree!)
  - allow more than one key in a node of a search tree
  - a node is called an *n*-node if it has at most $n - 1$ ordered keys
  - all leaves are on the same level (perfectly balanced)
  - In practice, parents are for indexing, leaf nodes for storing record

2-3 Tree – A Multiway Search Tree

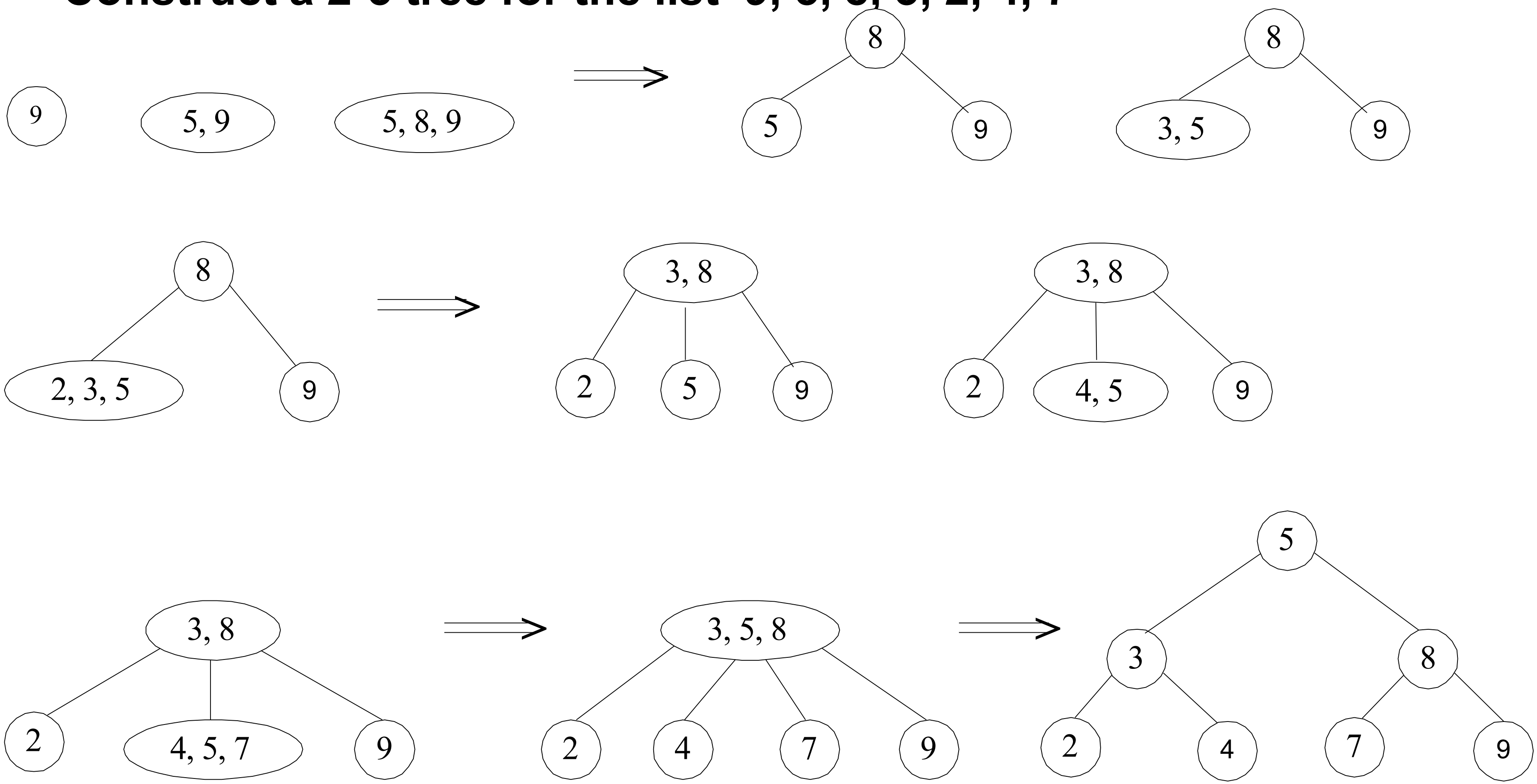
- A search tree may have 2-node and 3-node
- Height balanced – all leaves are on the same level



- Constructed by successive insertions of keys
- A new key is always inserted into a leaf of the tree. If the leaf is a 3-node (with two keys) already, it's split into two with the middle key promoted to the parent.

An Example of 2-3 Tree Construction

Construct a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7



Note on 2-3 Tree

- **Height of the tree $\log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1$**
- **Time efficiency**
  - Search, insertion, and deletion are in $\Theta(\log n)$

The idea of 2-3 tree can be generalized by allowing more keys per node

- 2-3-4 trees
- B-trees