ANALYSIS AND DESIGN OF ALGORITHMS

UNIT-III

CHAPTER 6:

TRANSFORM-AND-CONQUER

OUTLINE

- Transform-and-Conquer strategy
  - Three major variations
    - Instance simplification
    - Representation change
    - Problem reduction

- Applications
  - Presorting
  - Balanced Search Trees
    - AVL Trees
    - 2-3 Trees
  - Heaps and Heapsort

Introduction

- *Transform-and-conquer technique*
  - These methods work as **two-stage procedures**.

1. **Transformation stage**: the problem's instance is modified to be, for one reason or another, more amenable to solution.

2. In the **conquering stage**, it is solved.

Transform and Conquer

Three major variations that differ by what we transform a given instance to:

- transformation to a **simpler or more convenient instance** of the same problem - we call it ***instance simplification***. (Ex: Presorting, rotation in AVL trees etc.) .
- transformation to a **different representation** of the same instance - We call it ***representation change***. (Ex: 2-3 trees, Heaps and Heapsort etc.).
- transformation to an instance of a **different problem** for which an algorithm is already available - We call it ***problem reduction***. (Ex: reduction to graph problems etc.).

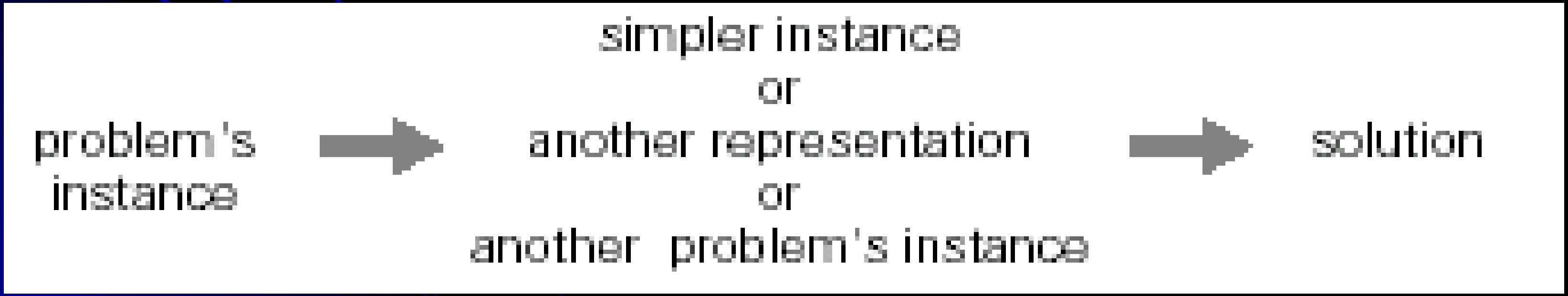


Figure: Transform-and-conquer strategy.

Instance simplification - Presorting

Solve a problem’s instance by **transforming it into another Simpler or easier instance** of the same problem

Presorting :

Many problems involving lists are easier when **list is sorted**.

- searching
- checking if all elements are distinct (element uniqueness)

Checking Element Uniqueness in an array

- **Presorting-based algorithm**

  Stage 1: sort by efficient sorting algorithm (e.g. merge sort).
  Stage 2: scan array to check pairs of adjacent elements.

  Worst-case Efficiency:
  $T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n\log n) + \Theta(n) = \Theta(n\log n)$

- The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements. It is the sorting part that will determine the overall efficiency of the algorithm.

- **Brute force algorithm**
  Compared pairs of the array's elements until either two equal elements were found or no more pairs were left.

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

Example 1 *Checking element uniqueness in an array.*

```
ALGORITHM PresortElementUniqueness(A[0..n - 1])
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n - 1] of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
Sort the array A
for i ← 0 to n - 2 do
    if A[i]= A[i + 1] return false
return true
```

We can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other and vice versa.

Example 2 *Computing a mode*

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

  Example: for 5, 1, 5, 7, 6, 5, 7, the mode is 5.

- The **brute-force approach** for computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.

Example 2 *Computing a mode*

- To implement the brute-force approach, we can store the values already encountered, along with their frequencies, in a separate list.
  - On each iteration, the *i*th element of the original list is compared with the values already encountered by traversing this auxiliary list.
  - If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

Example 2 *Computing a mode*

- The worst-case input for the Brute-force approach is a list with no equal elements.

  For such a list, its i^{th} element is compared with $i - 1$ elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1.

  As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is

$$C(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \ldots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

  The additional $n - 1$ comparisons are needed to find the largest frequency in the auxiliary list. But the efficiency remains quadratic.

Example 2 *Computing a mode*

- In **presort mode computation algorithm**, the list is sorted first. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

- The efficiency of this presort algorithm will be ***nlogn***, which is the time spent on sorting.

Example 2 *Computing a mode*

```
ALGORITHM PresortMode(A[0..n - 1])
//Computes the mode of an array by sorting it first
//Input: An array A[0..n - 1] of orderable elements
//Output: The array's mode
    Sort the array A
    $i \leftarrow 0$    //current run begins at position i
    $modefrequency \leftarrow 0$   //highest frequency seen so far
    while $i \le n - 1$ do
        $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
        while $i + runlength \le n - 1$ and $A[i + runlength] = runvalue$
            $runlength \leftarrow runlength + 1$
        if $runlength > modefrequency$
            $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
        $i \leftarrow i + runlength$
    return $modevalue$
```

Example 3: *Searching Problem*

Problem: Search for a given K in A[0..n-1]

Presorting-based algorithm:

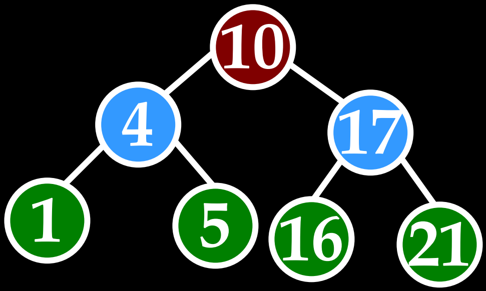
Stage 1 Sort the array by an efficient sorting algorithm.

Stage 2 Apply binary search.

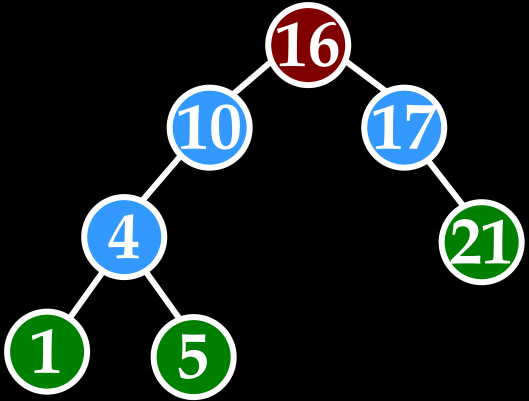
Efficiency:

$T(n) = T_{\text{sort}}(n) + T_{\text{search}}(n) = \Theta(n\log n) + \Theta(\log n) = \Theta(n\log n)$

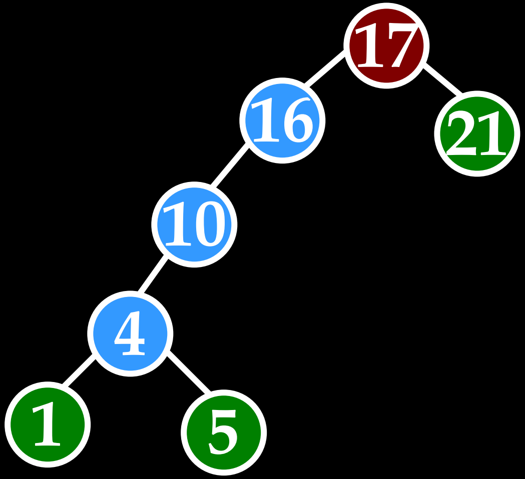
BINARY SEARCH TREES



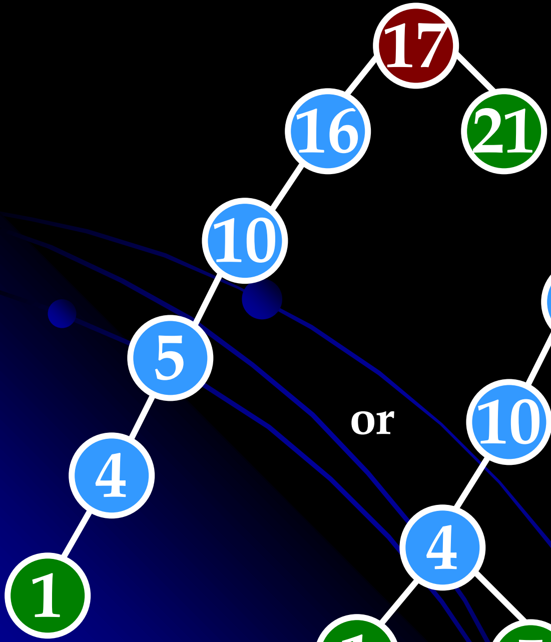
HEIGHT 2



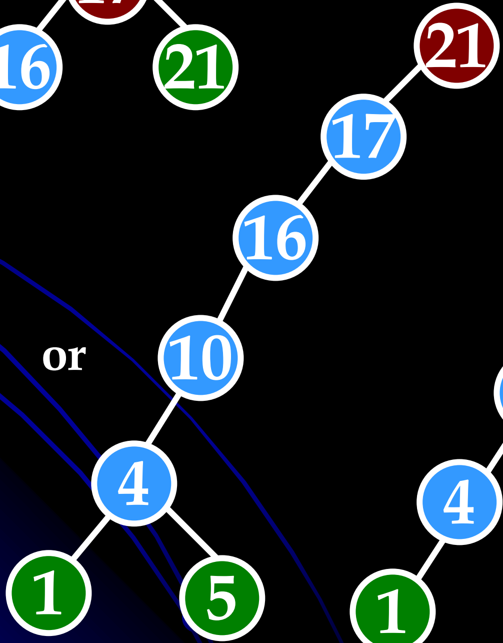
HEIGHT 3



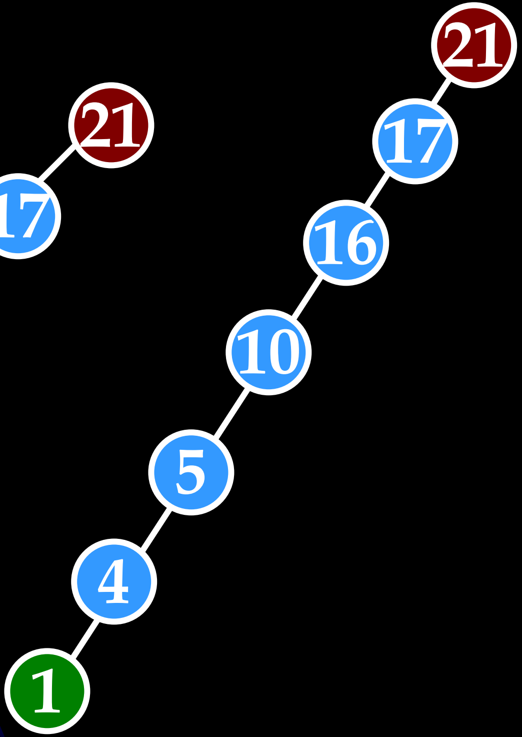
HEIGHT 4



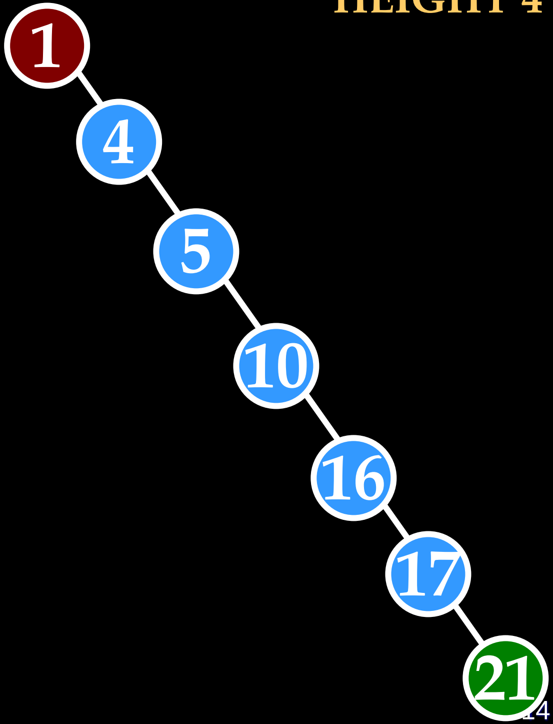
or



HEIGHT 5



or



HEIGHT 6

Balanced Search Trees

- Binary search tree is one of the principal data structures for implementing **dictionaries**.

- BST is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.

- The transformation from a set to a **binary search tree** is an example of the **representation-change technique**.

- By doing this transformation, we **gain in the time efficiency of searching, insertion, and deletion,** which are all in **Θ(log *n*)**, but only in the average case. In the worst case, these operations are in **Θ(*n*)** since trees become severely unbalanced.

Balanced Search Trees

- *There are two approaches to balance a search tree:*

  - **The first approach is of instance – simplification variety:** an unbalanced binary search tree is transformed to a balanced one. An **AVL tree** requires the difference between the heights of the left and right subtrees of every node never exceed 1. Other types of trees are **red-black** trees and **splay** trees.

  - **The second approach is of the representation-change variety:** allow more than one element in a node of a search tree. Specific cases of such trees are **2 – 3** trees, **2-3-4** trees, **B-trees**. They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced.

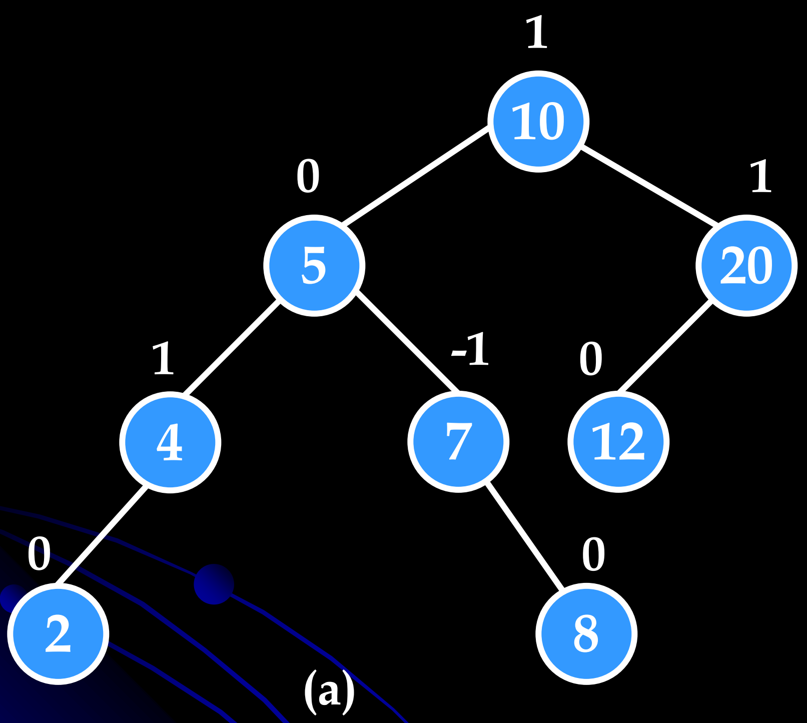
AVL TREES

- **AVL trees** were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis, after whom this data structure is named.

- **Definition:**

  An AVL tree is a binary search tree in which the **balance factor** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0, +1 or -1. (The height of the empty tree is defined as -1).

AVL TREES



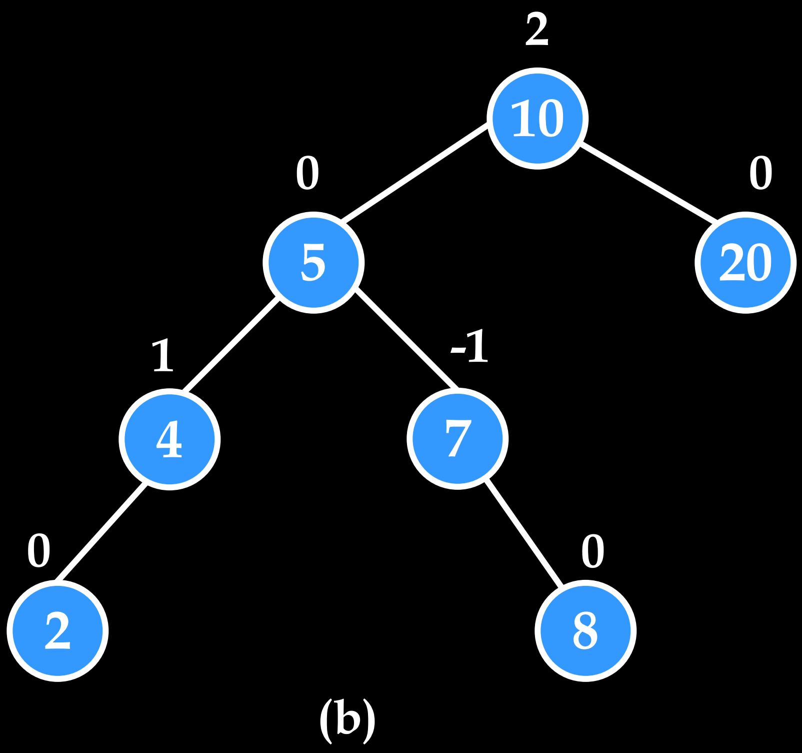


Figure: (a) AVL tree. (b) Binary search tree that is not an AVL tree. The number above each node indicates that node's balance factor.

AVL TREES

- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a **rotation**.

- **A rotation in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2**; if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

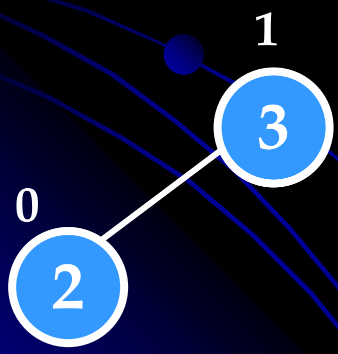
- **There are only four types of rotations**:
    - Single right rotation (R-rotation)
    - Single left rotation (L-rotation)
    - Double left-right rotation (LR – rotation)
    - Double right-left rotation (RL – rotation)

AVL TREES

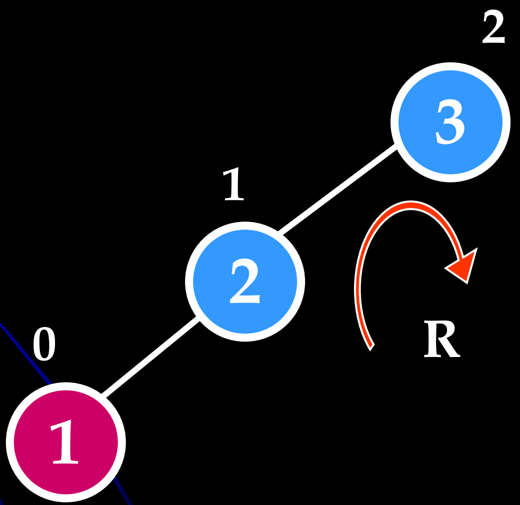
❖ **Single right rotation (R-rotation):**

This rotation is performed after a new key is inserted into the **left subtree of the left child** of a tree whose root had the Balance of +1 before the insertion.

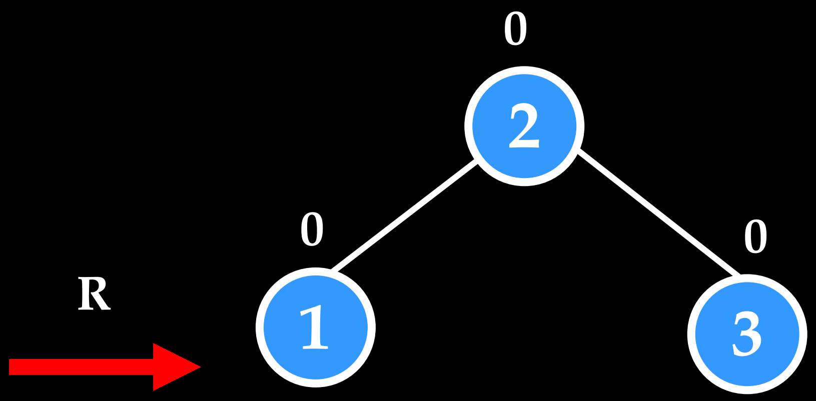
(Imagine rotating the edge connecting the root and its left child in the binary tree in below figure to the right.)



(a) Balanced tree



(b) Unbalanced tree after inserting 1



(c) Balanced tree

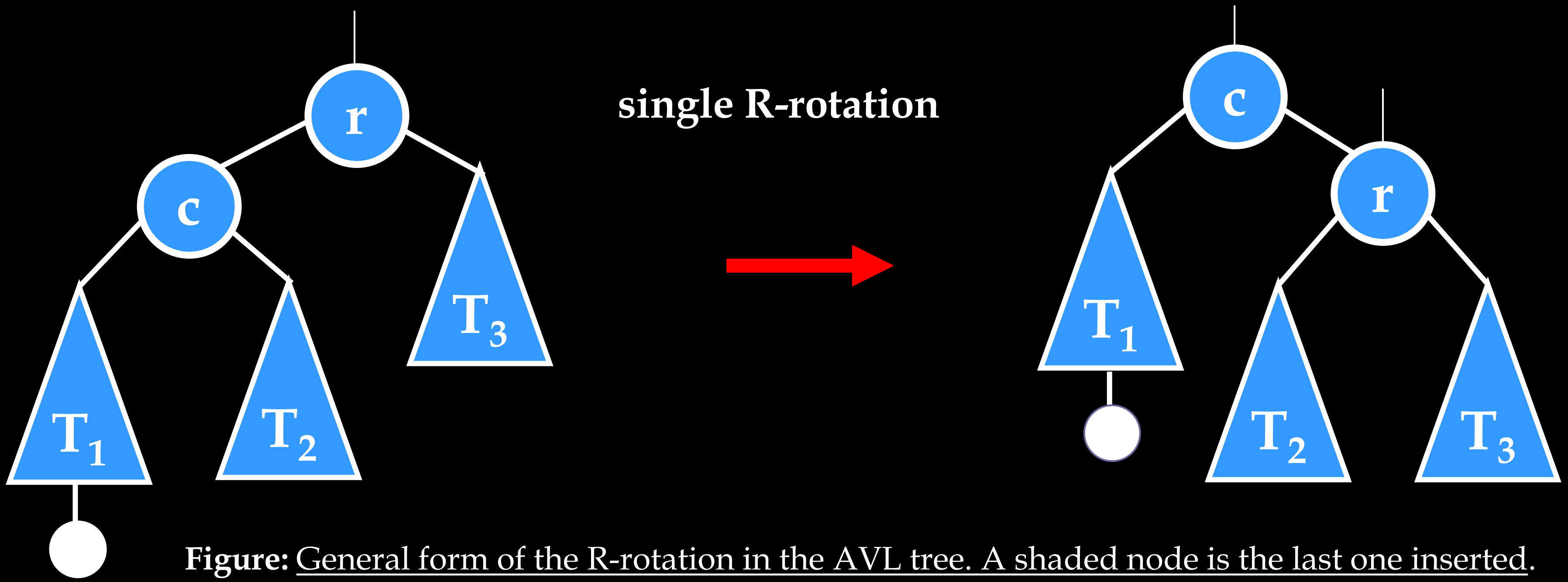


Figure: General form of the R-rotation in the AVL tree. A shaded node is the last one inserted.

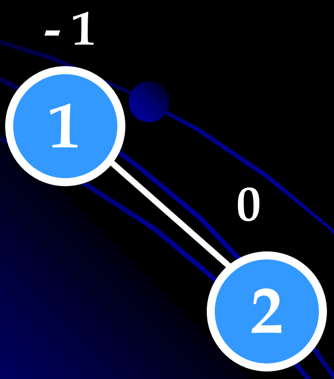
Note:

- These transformations should not only guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree.
- For example, in the initial tree in the above figure, all the keys of subtree T_1 are smaller than c, which is smaller than all the keys of subtree T_2, which are smaller than r, which is smaller than all the keys of subtree T_3.
- The same relationships among the key values should hold for the balanced tree after rotation.

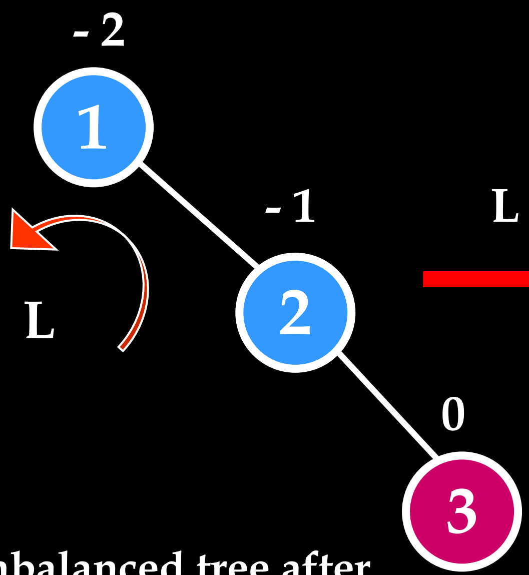
AVL TREES

- **Single left rotation (L-rotation):**
- This is the mirror image of the single R-rotation.
- This rotation is performed after a new key is inserted into the **right subtree of the right child** of a tree whose root had the balance of -1 before the insertion.

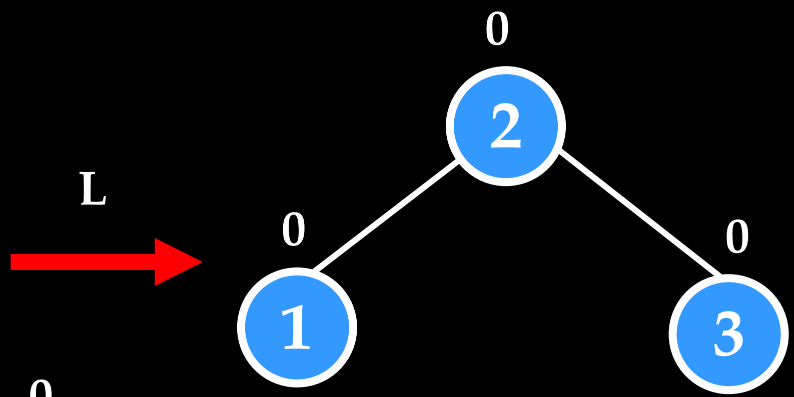
(Imagine rotating the edge connecting the root and its right child in the binary tree in below figure to the left.)



(a) Balanced tree



(b) Unbalanced tree after inserting 3



(c) Balanced tree

AVL TREES

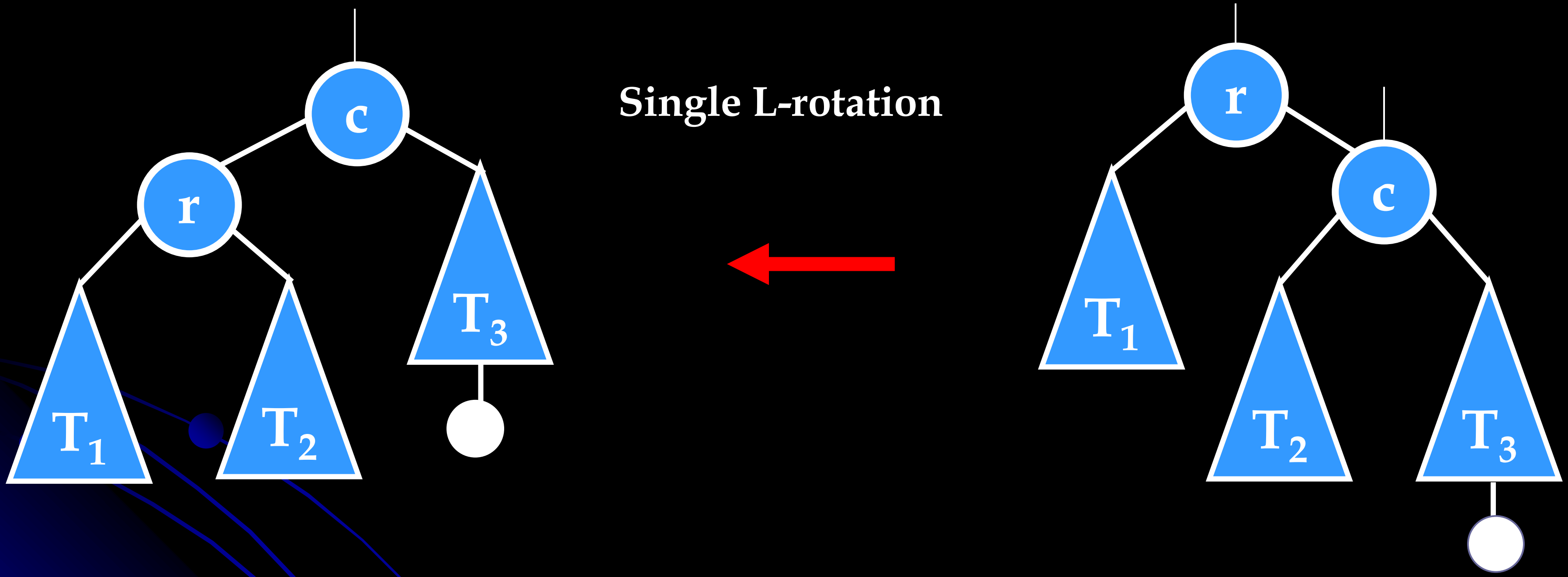
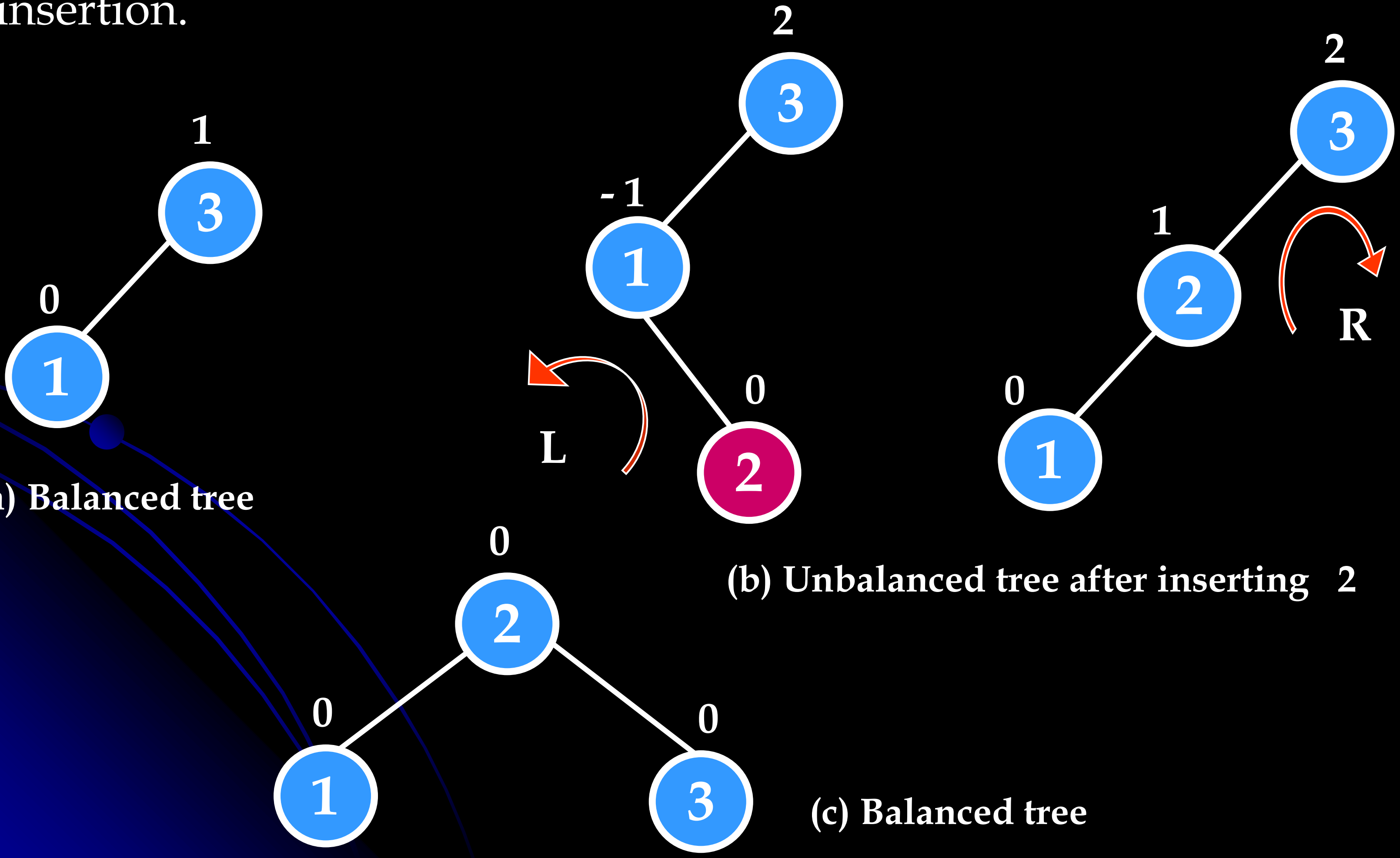


Figure: General form of the L-rotation in the AVL tree. A shaded node is the last one inserted.

AVL TREES

- **Double left-right rotation (LR-rotation):**

It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



(a) Balanced tree

(b) Unbalanced tree after inserting 2

(c) Balanced tree

AVL TREES

❖ **Double left-right rotation (LR-rotation):**

It is a combination of two rotations: We perform the L-rotation of the left subtree of root *r* followed by the R-rotation of the new tree rooted at *r* in below figure.

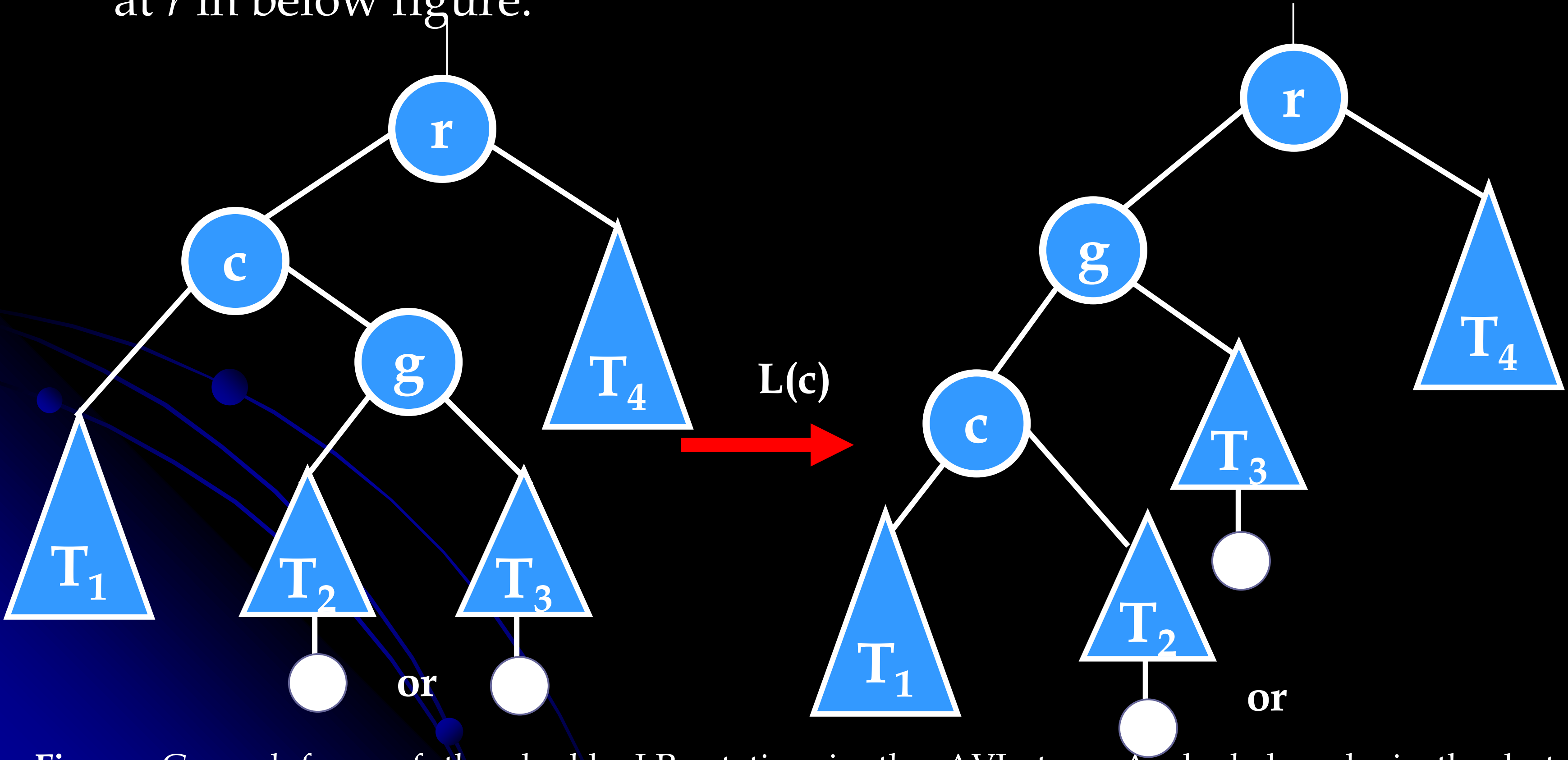
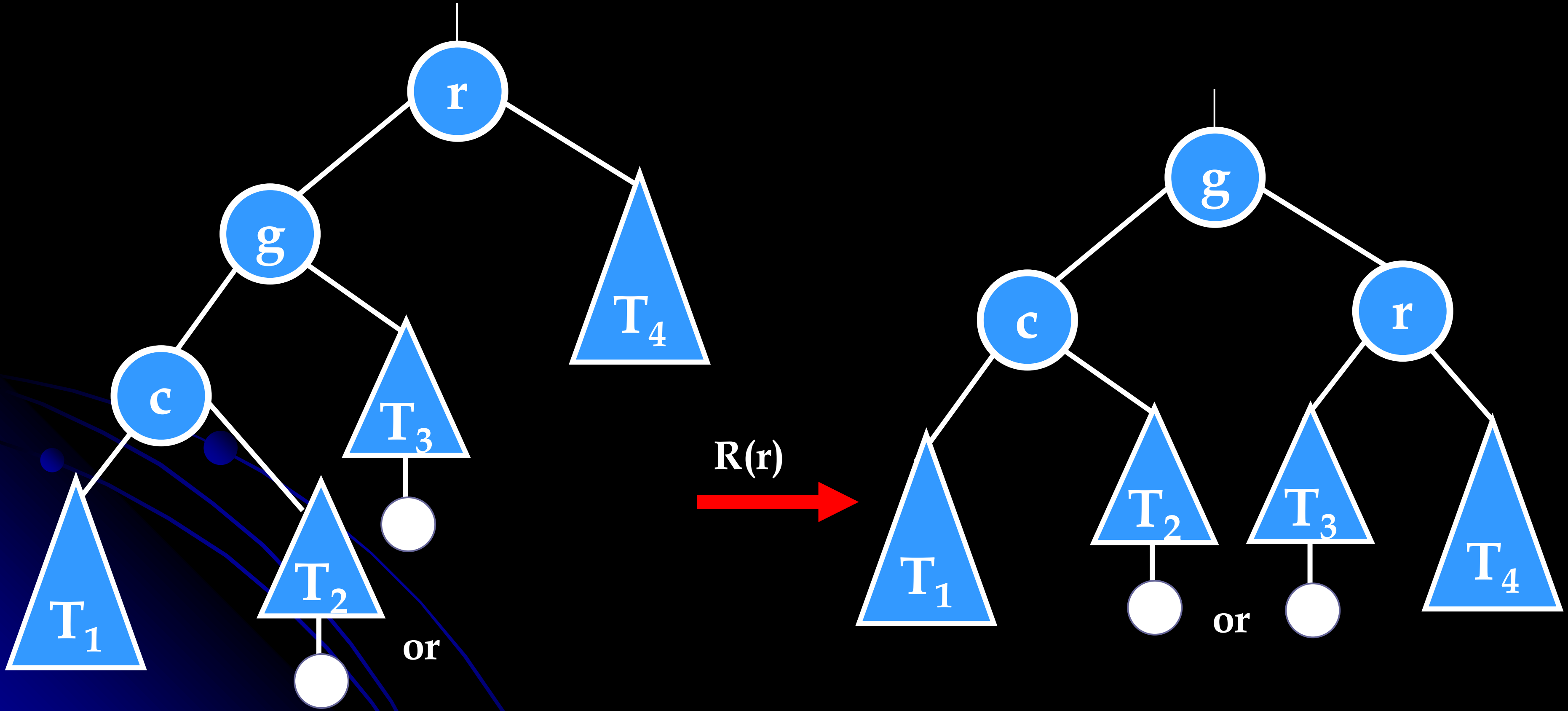


Figure: General form of the double LR-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

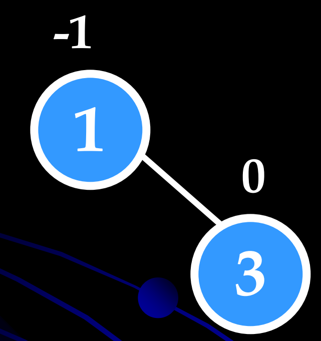
AVL TREES

- **Double left-right rotation (LR-rotation):**

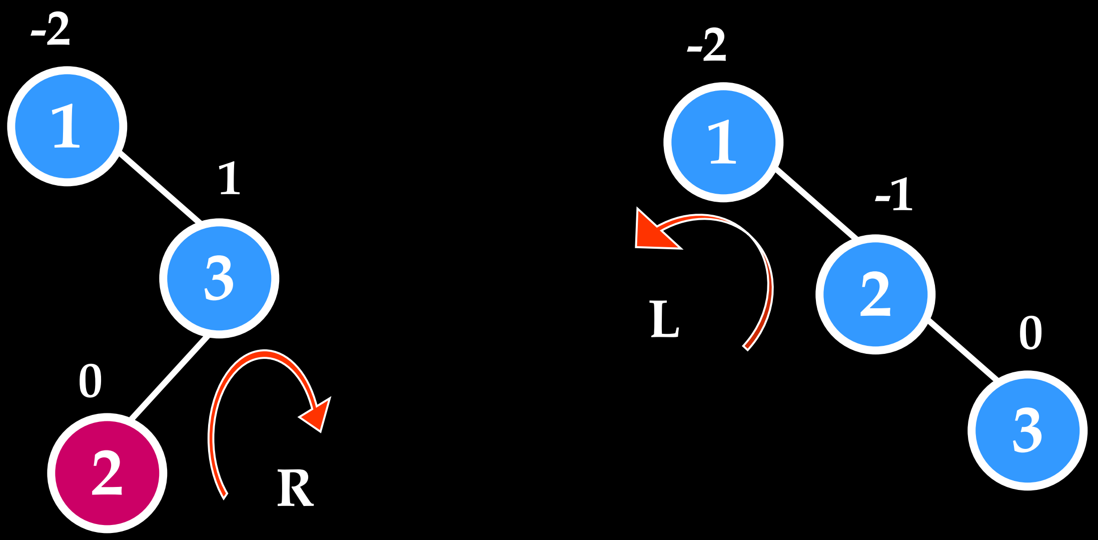


AVL TREES

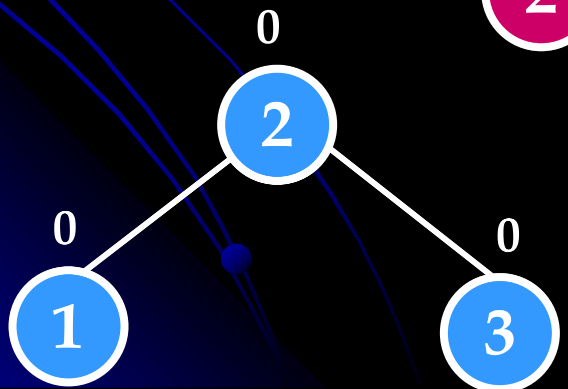
- **Double right-left rotation (RL-rotation):**
- It is the mirror image of the double LR-rotation.
- It is performed after a new key is inserted into the left subtree of the right child of a tree whose root had the balance of -1 before the insertion.



(a) Balanced tree



(b) Unbalanced tree after inserting 2



(c) Balanced tree

AVL TREES

❖ **Double right-left rotation (RL-rotation):**

It is a combination of two rotations: We perform the R-rotation of the right subtree of root *r* followed by the L-rotation of the new tree rooted at *r* in below figure.

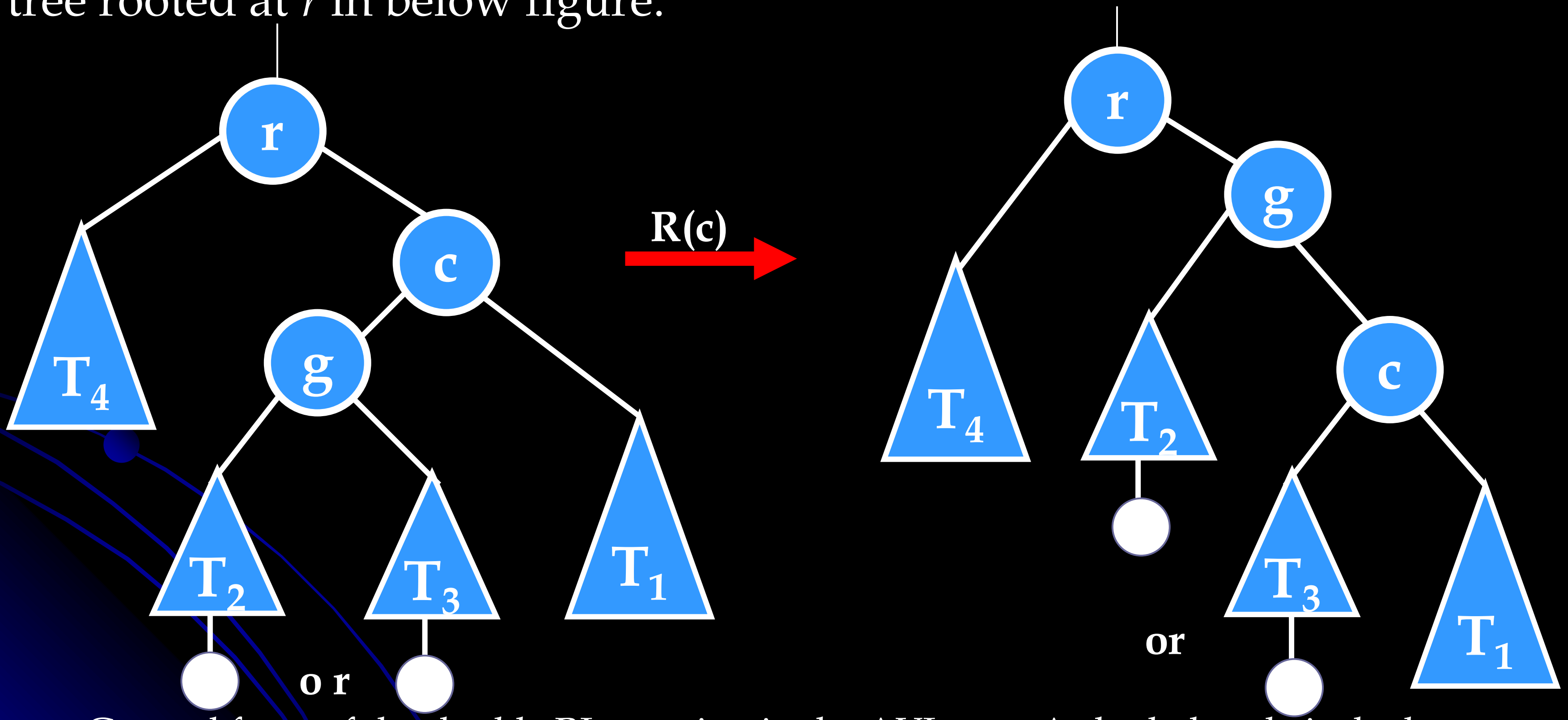
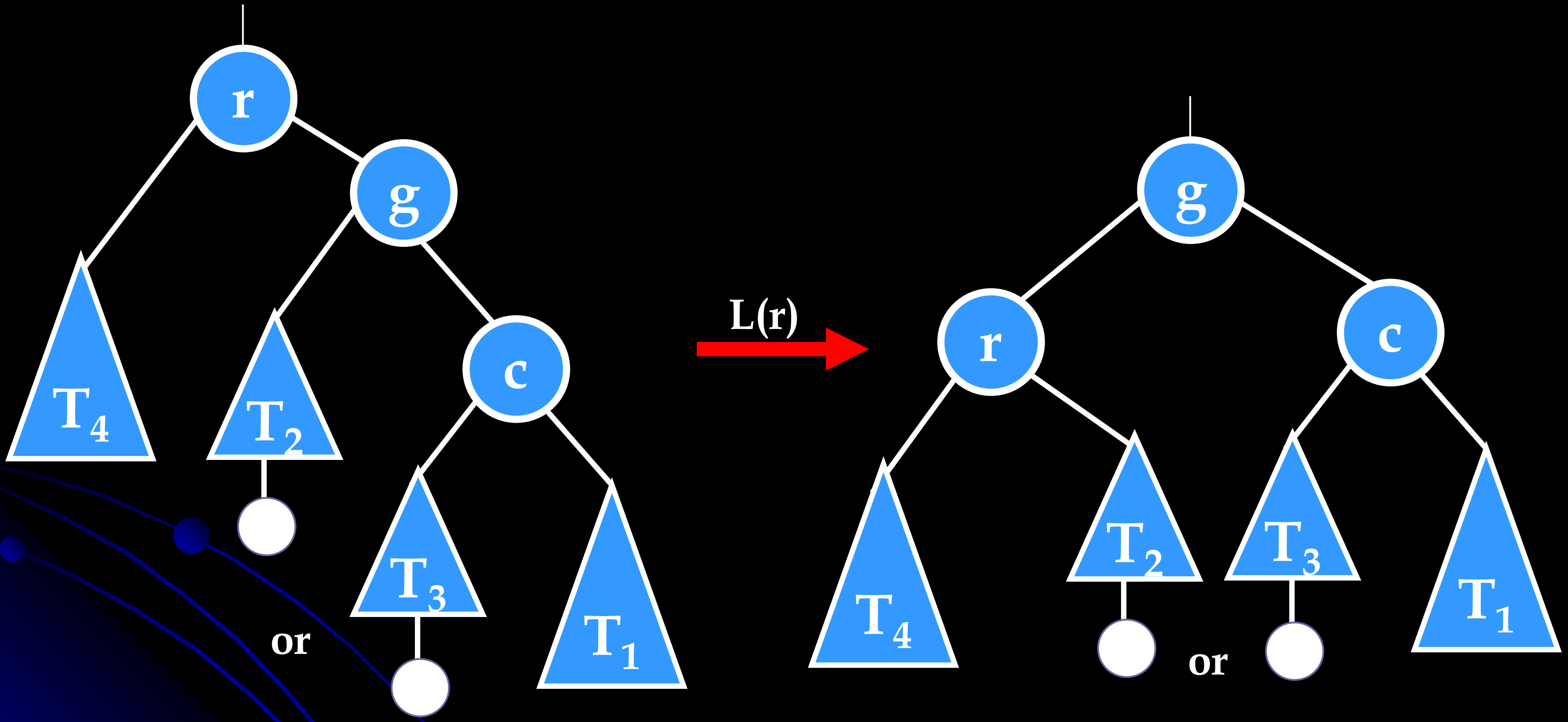


Figure: General form of the double RL-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

AVL TREES

- **Double right-left rotation (RL-rotation):**



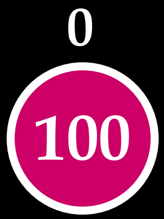
AVL TREES

- The balance factor of a node is 0 if and only if the height of the left subtree and height of right subtree are same.
- The balance factor of a node is 1 if the height of the left subtree is one more than height of right subtree (left heavy).
- The balance factor of a node is -1 if the height of the right subtree is one more than height of left subtree (right heavy).
- The height of empty tree is -1.
- If there are *several nodes with the ±2 balance*, the rotation is done for the tree rooted at the unbalanced node *that is the closest to the newly inserted leaf*.

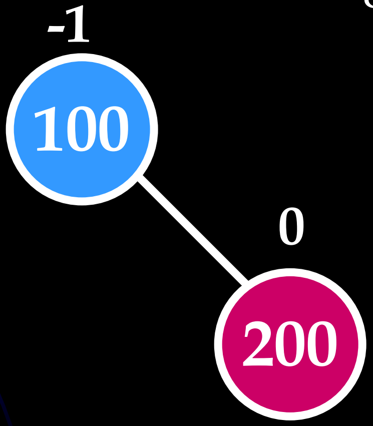
Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40

Step1: Element 100 is inserted into the empty tree as shown below:



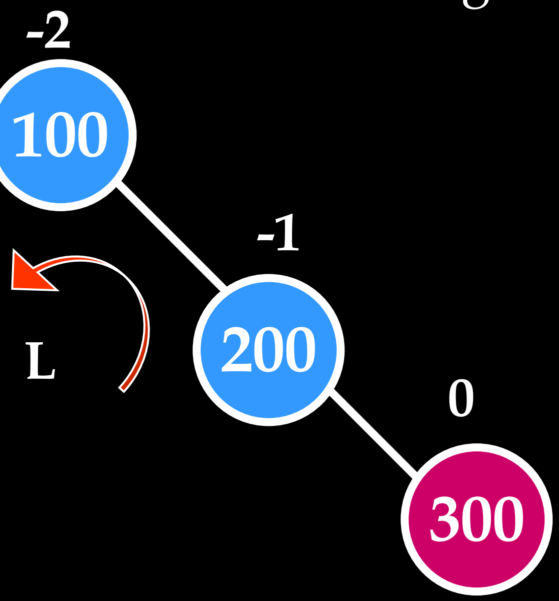
Step2: Element 200 is inserted to the right of 100 as shown below:



Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40

Step3: Element 300 is inserted towards right of 200 as shown below:

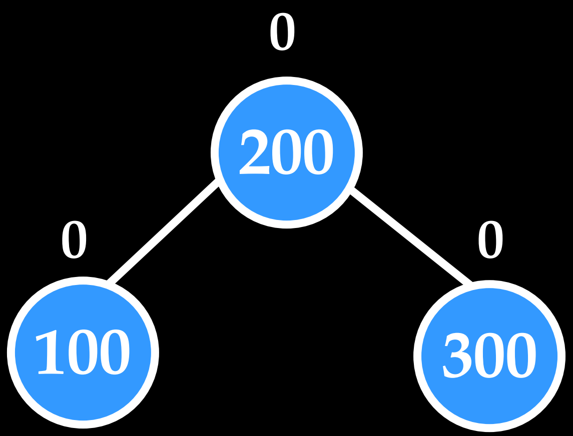


Above tree is unbalanced since balance factor at node 100 is -2. Here, the new node is inserted into right subtree of the right child. Hence single rotation is sufficient to balance the tree.

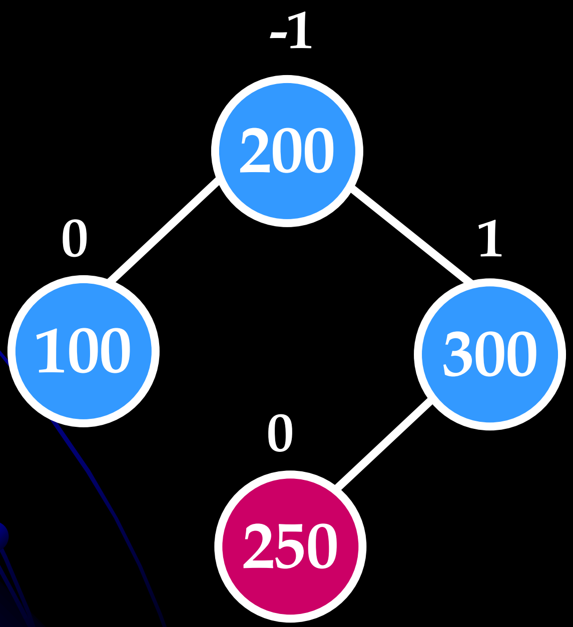
Since the tree is heavy towards right, L-rotation is made. i.e., the edge connecting 100 to 200 is rotated left making 200 as the root.

Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40



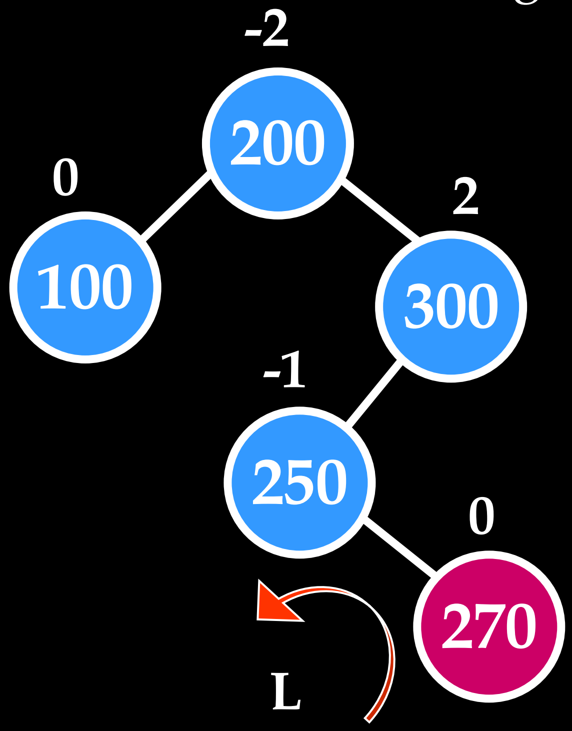
Step4: Element 250 is inserted as the left child of 300 as shown below:



Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40

Step5: Element 270 is inserted as the right child of 250 as shown below:



The youngest ancestor which is out of balance is 300. So we consider node 300 with two nodes below it i.e., 250 and 270.

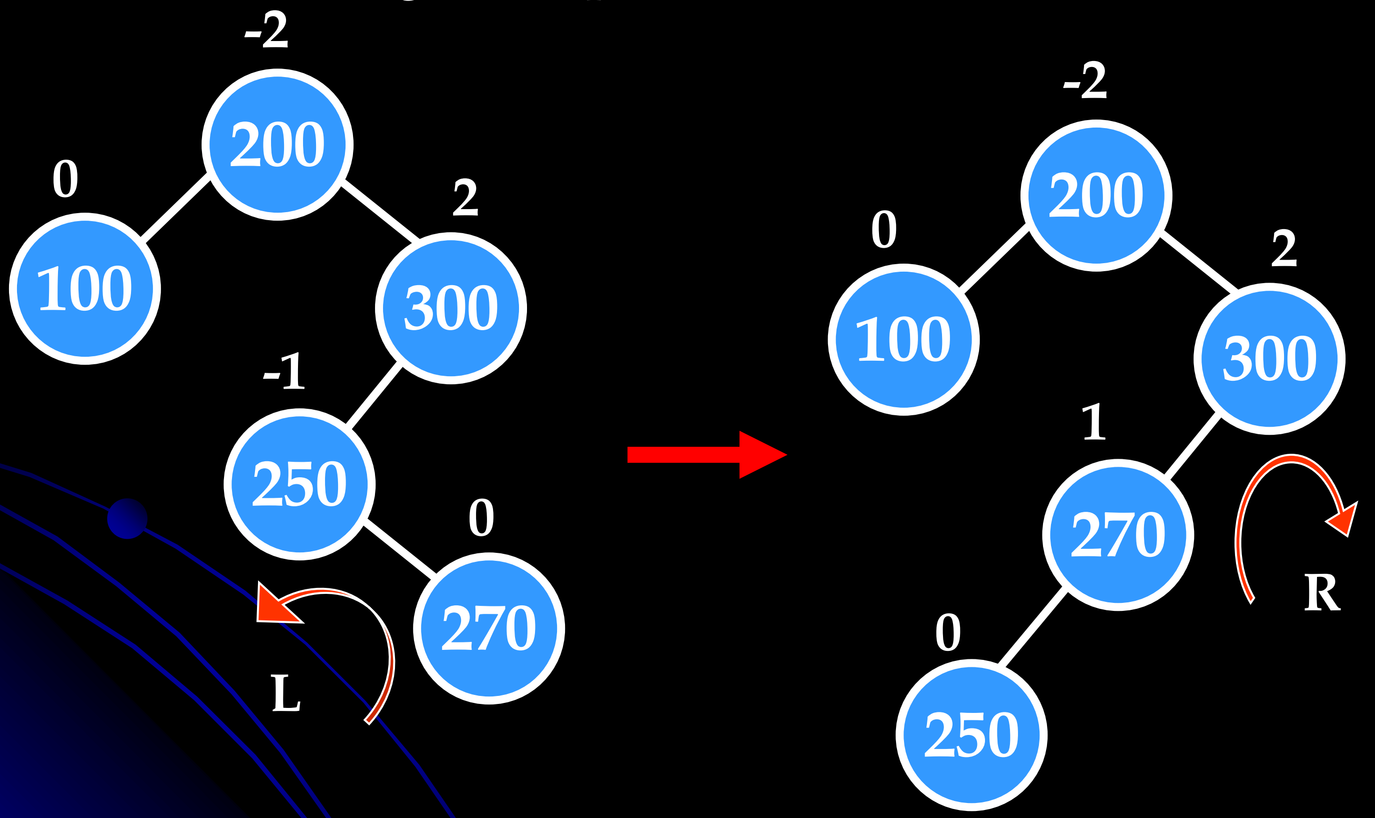
Since the new node is inserted into the right subtree of left child, the tree requires double rotation (LR-rotation).

Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40

The first rotation occurs at node 250.

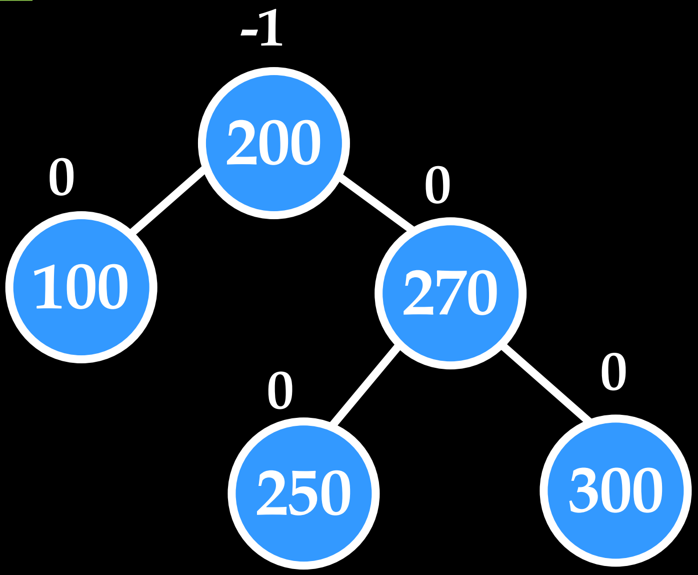
Left rotation brings 270 up and 250 becomes left child of 270.



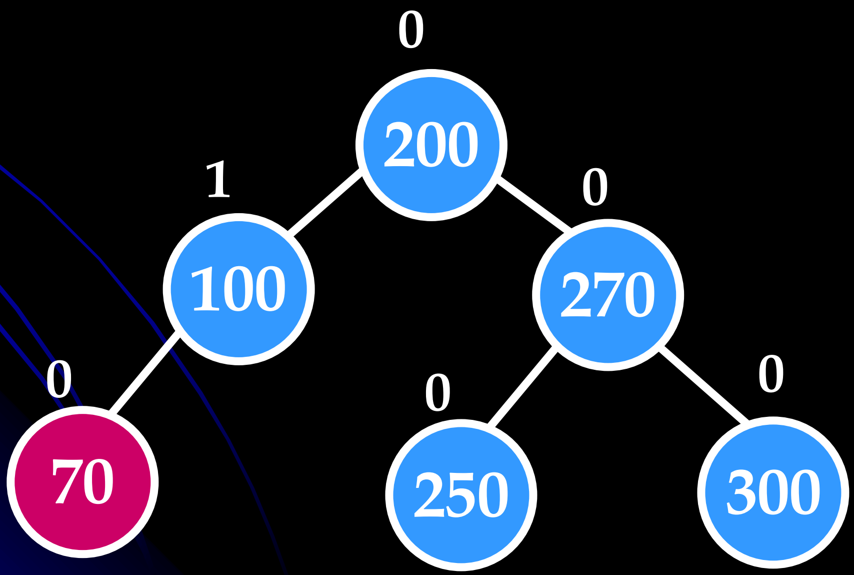
Now, right rotation is made at node 300. This makes node 300 to become right child of 270.

Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40



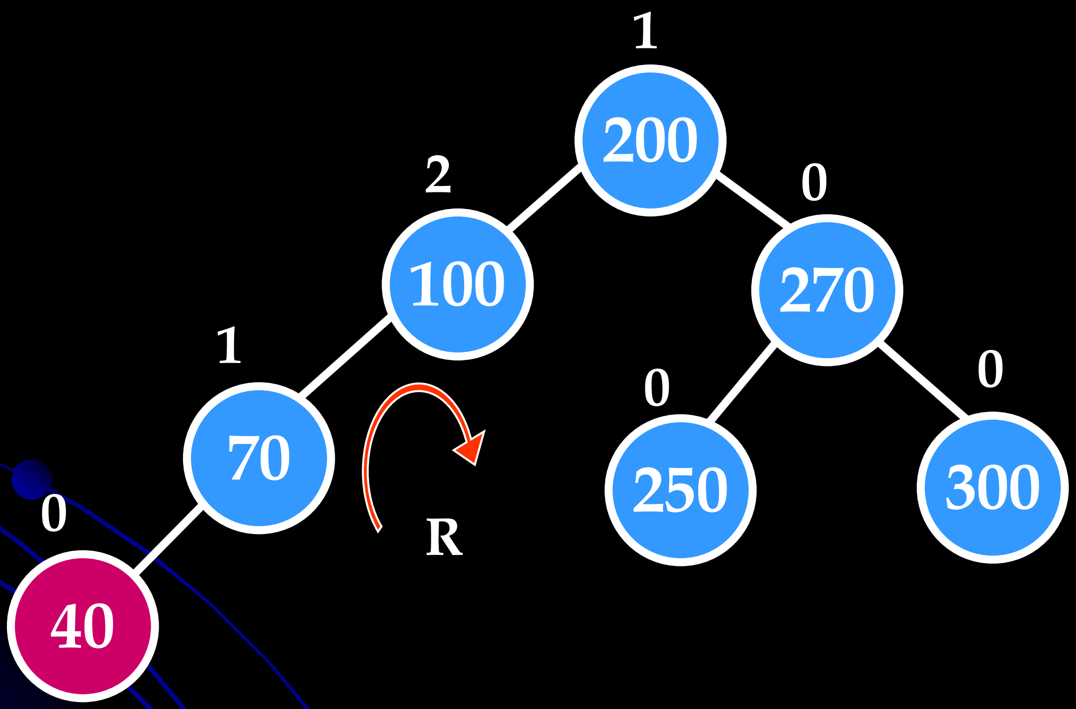
Step6: Element 70 is inserted as left child of 100 as shown below:



Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40

Step7: Element 40 is inserted as left child of 70 as shown below:



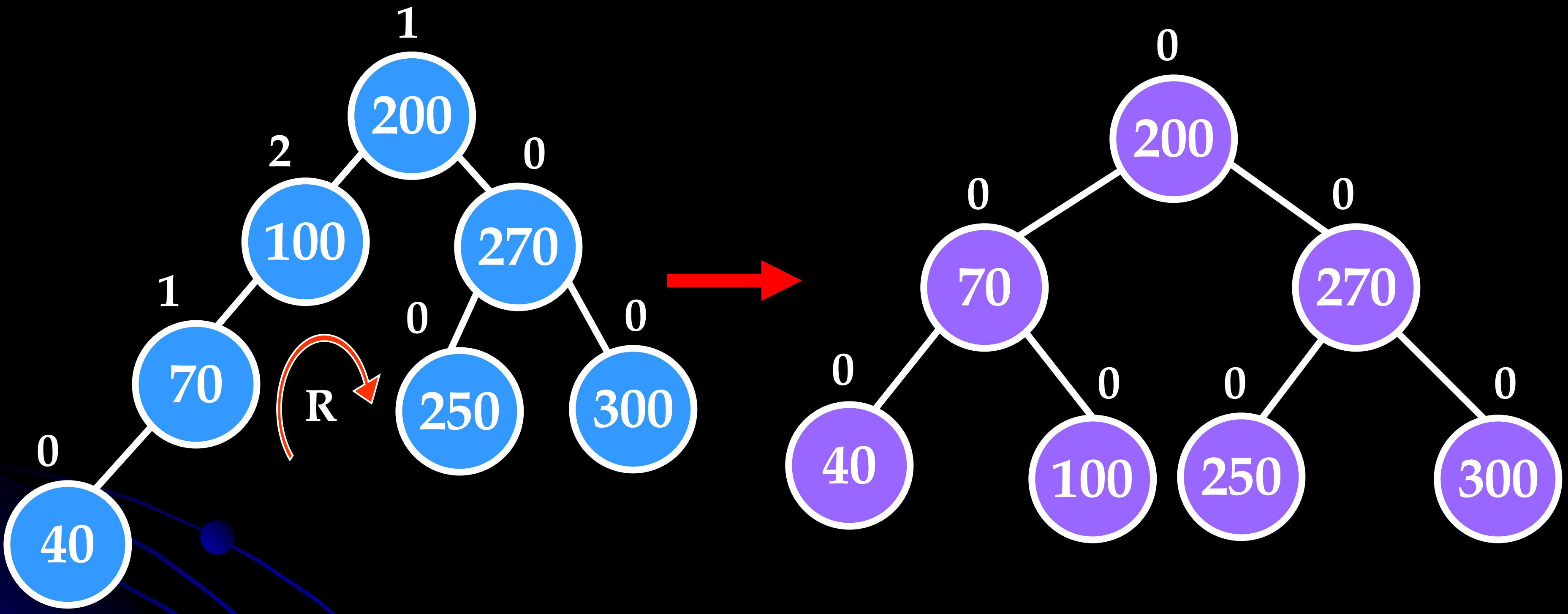
The tree is unbalanced since node 100 has the balance factor of 2.

Hence we consider node 100 and its below nodes 70 and 40.

Since new node is inserted into left subtree of left child, a single rotation is made.

Construction of an AVL tree

EXAMPLE 1: 100, 200, 300, 250, 270, 70, 40



Right rotation is performed since the tree is left heavy.

Construction of an AVL tree

EXAMPLE 2: **5, 6, 8, 3, 2, 4, 7**

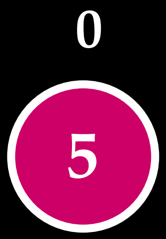


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

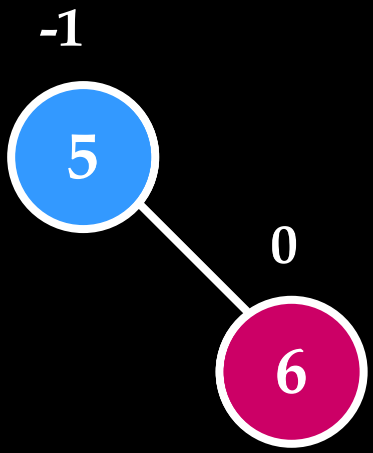


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

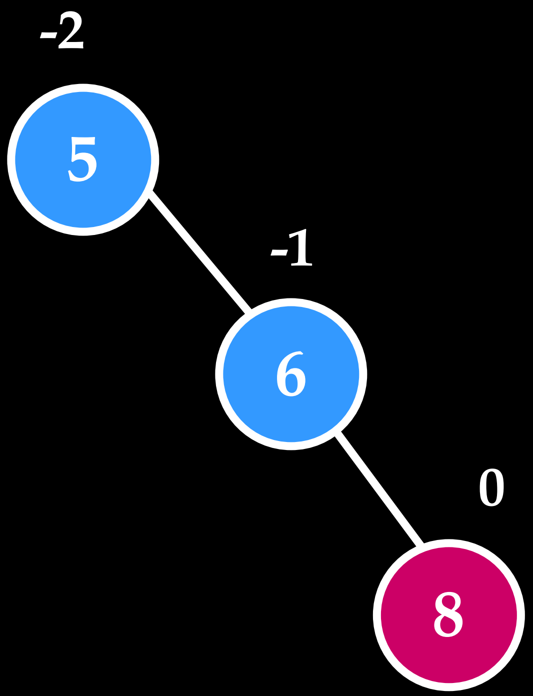


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

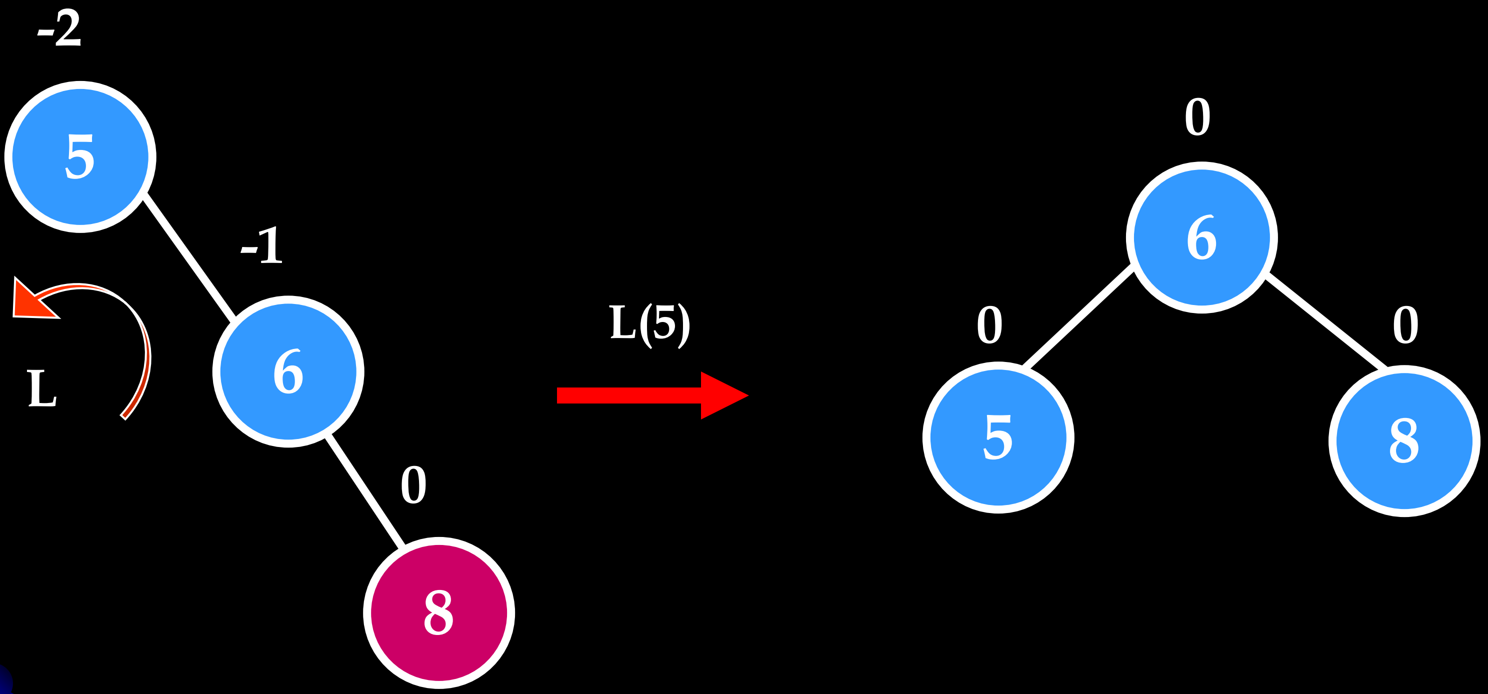


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

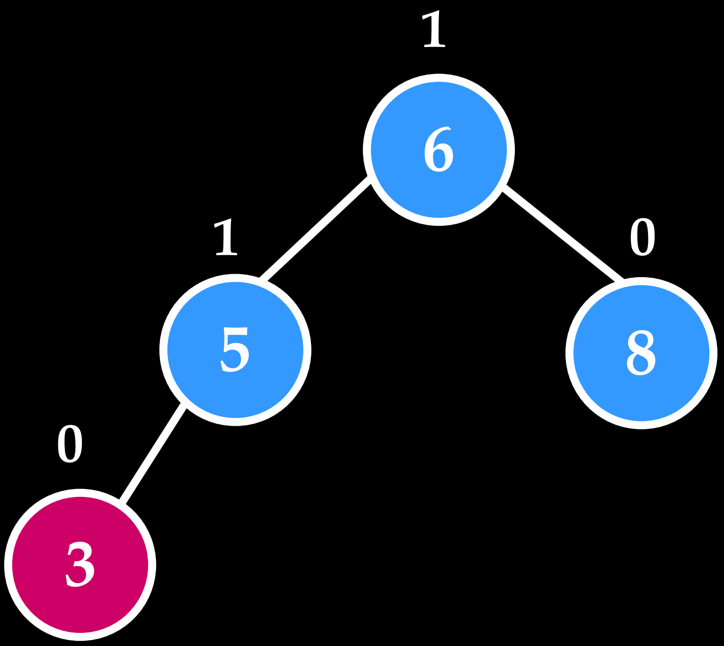


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

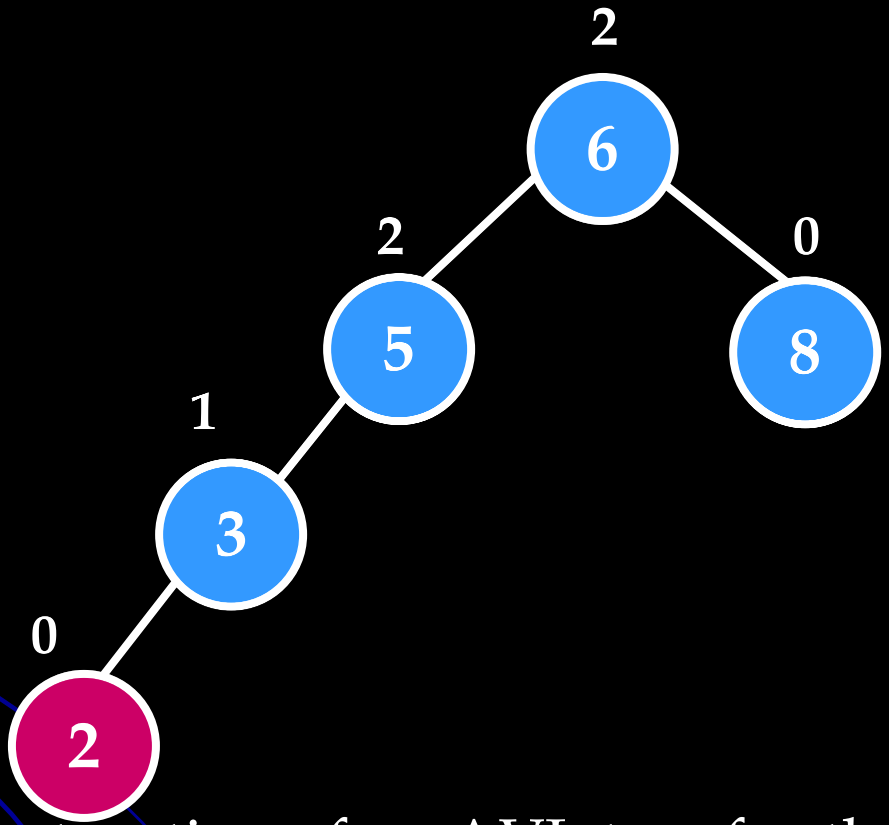


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7

by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

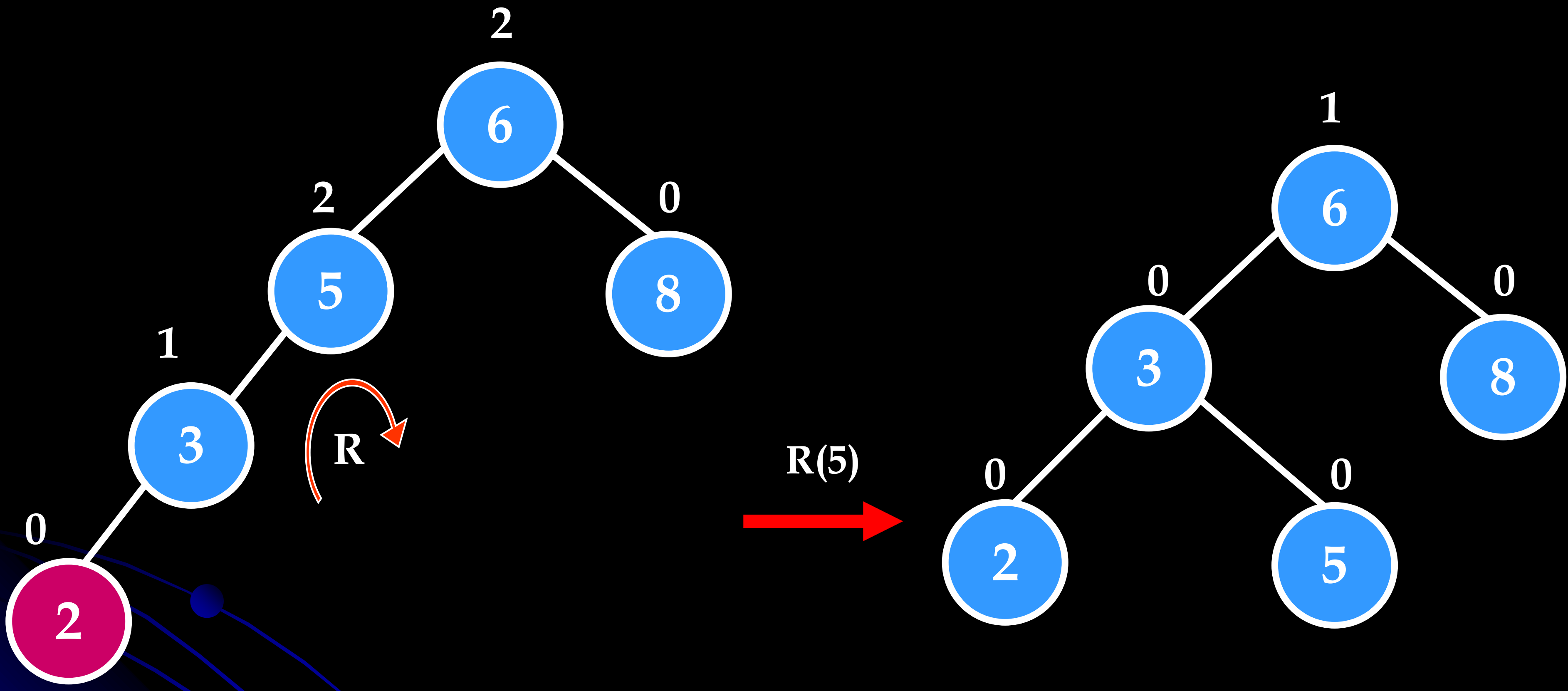


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

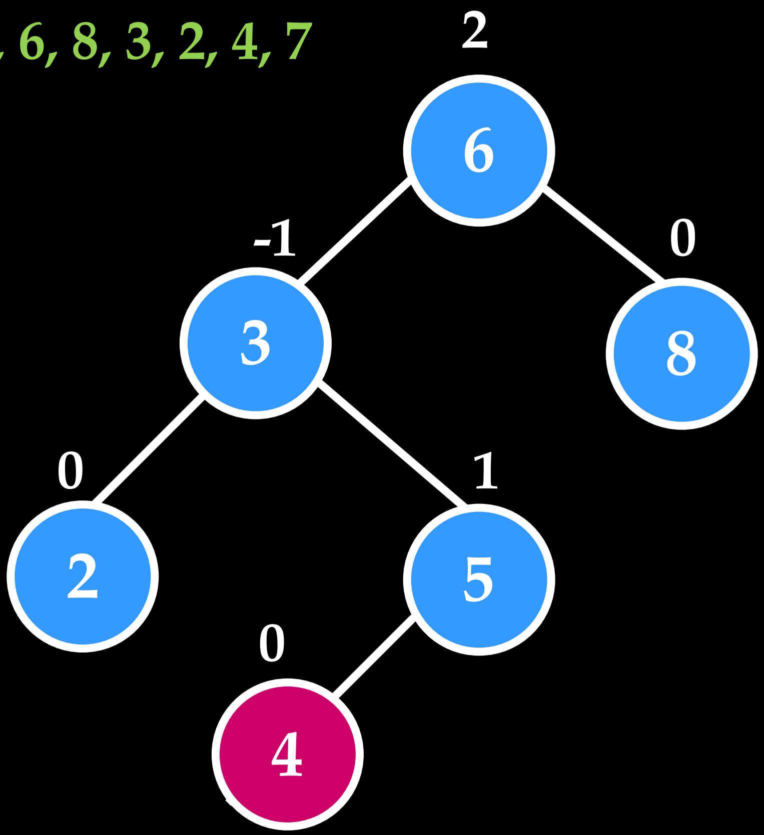


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

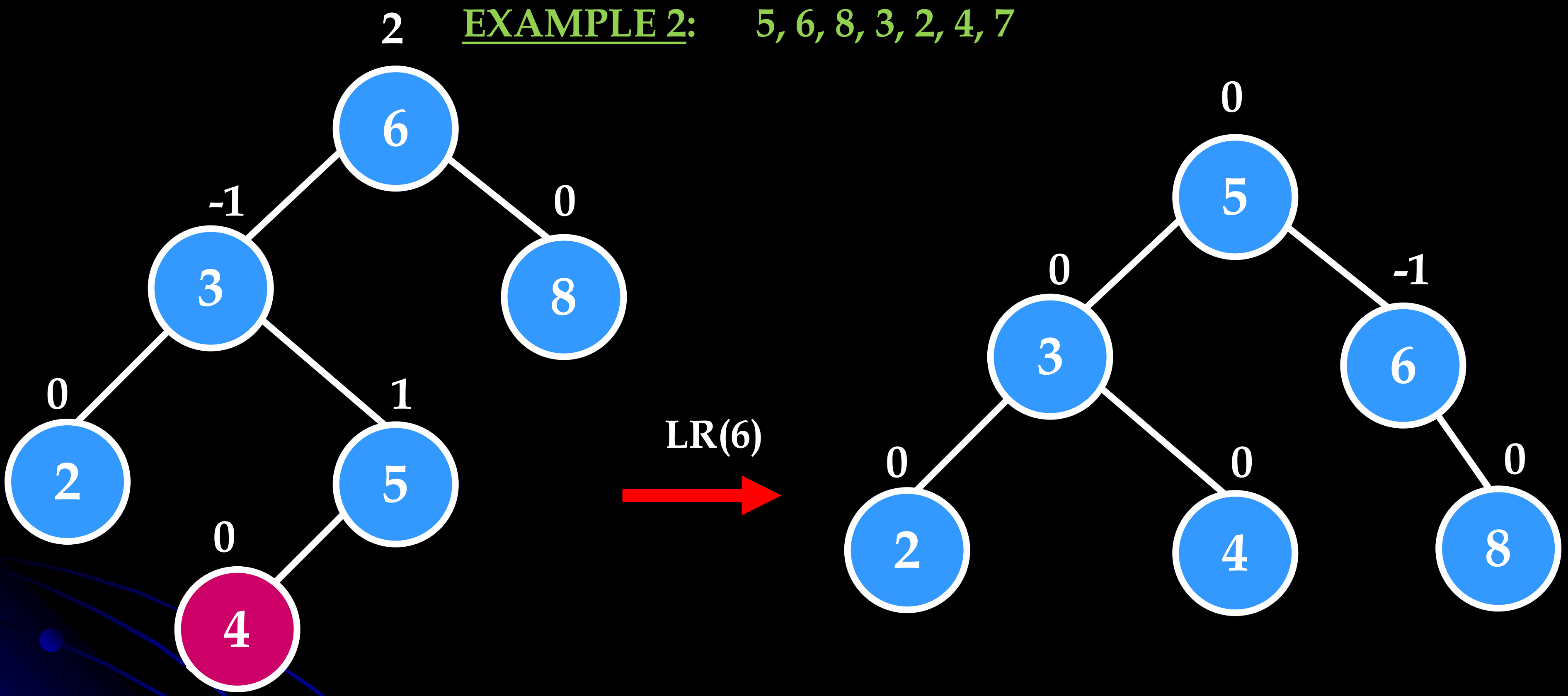


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

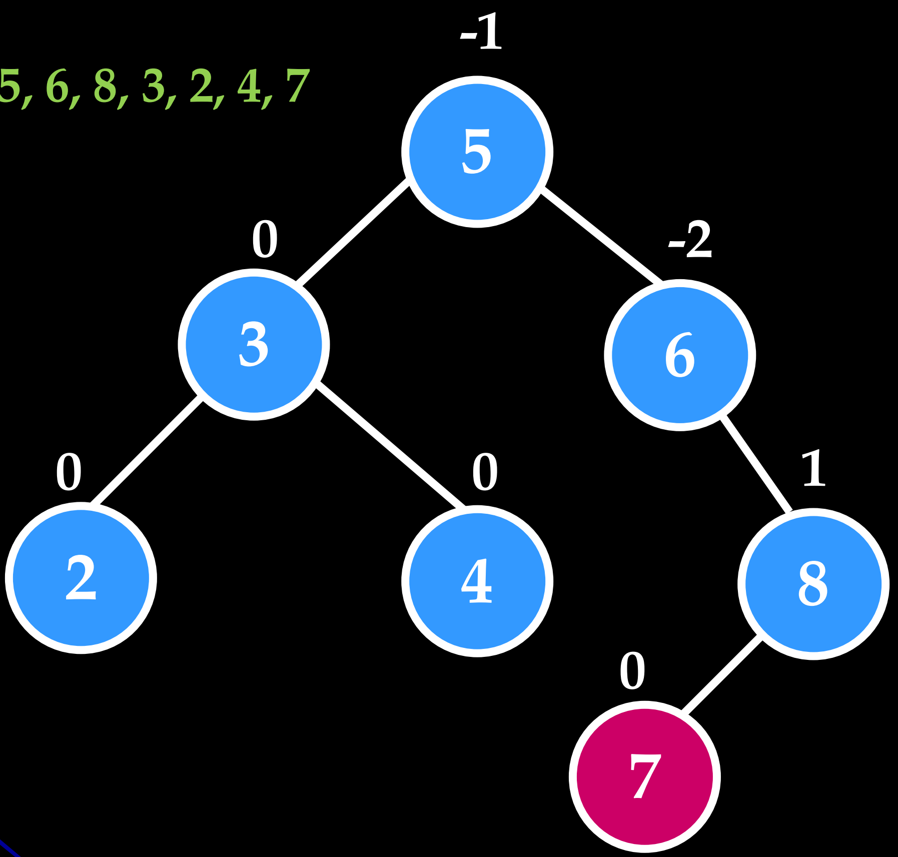


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions.

Construction of an AVL tree

EXAMPLE 2: 5, 6, 8, 3, 2, 4, 7

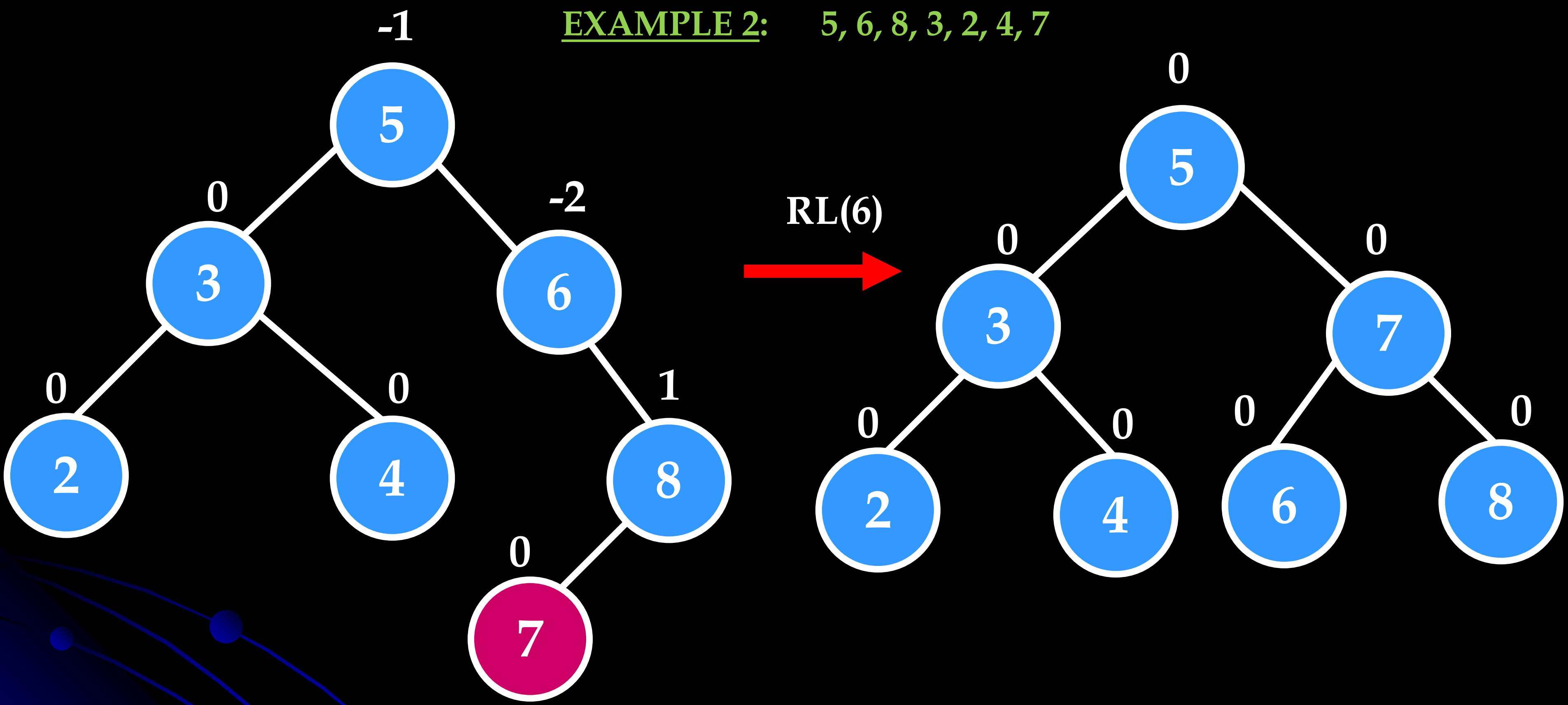


Figure: Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

Efficiency of AVL Trees

- The **height** h of any AVL tree with n **nodes** satisfies the inequalities

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

- These inequalities imply that the operations of **searching and insertion** are **Θ(log *n*)** in the **worst case**.

- The operation of **key deletion in an AVL tree** is considerably more difficult than insertion, but fortunately it turns out to be in the same efficiency class as insertion, i.e., **logarithmic**.

- The **drawbacks of AVL trees** are **frequent rotations**, the need to **maintain balances** for the tree's nodes, and **overall complexity**, especially of the **deletion operation**.

- These drawbacks have **prevented AVL trees** from becoming the standard structure for **implementing dictionaries**.

2-3 TREES

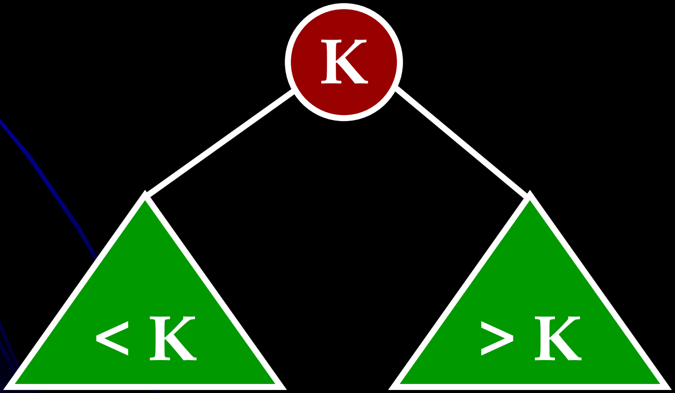
- Another idea of balancing a search tree is to **allow more than one key in the same node** (Ex: 2-3 trees).

- **2-3 trees** was introduced by the U. S. computer scientist **John Hopcroft** in 1970.

- A **2-3 tree** is a tree that can have nodes of two kinds: **2-nodes** and **3-nodes**.

- A **2-node** contains a single **key K** and has two children: the **left child** serves as the root of a subtree whose keys are less than K and the **right child** serves as the root of a subtree whose keys are greater than K.

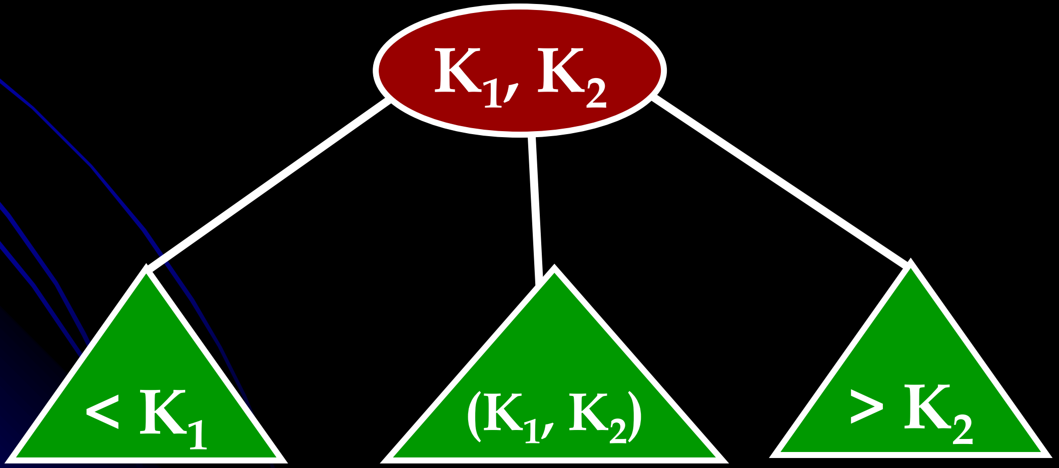
2 - node



2-3 TREES

- A **3-node** contains **two ordered keys K_1 and K_2** ($K_1 < K_2$) and has three children.
  - The ***leftmost child*** serves as the root of a subtree with keys less than K_1
  - The ***middle child*** serves as the root of a subtree with keys between K_1 and K_2.
  - The ***rightmost child*** serves as the root of a subtree with keys greater than K_2.

3 - node



2-3 TREES

- In **2-3 trees, all the leaves must be on the same level.** i.e., a ***2-3 tree is always perfectly height-balanced***: the length of a path from the root of the tree to a leaf must be the same for every leaf.

- **Searching in a 2-3 Tree:**

- Searching for an element always **starts from the root**.

- If the **root is a 2-node**, then the searching is similar to the way we search in a binary search tree: We either stop if search key K is equal to the root's key or continue the search in the left subtree if K is less than the root's key or continue the search in the right subtree if K is larger than the root's key.

2-3 TREES

- **Searching in a 2-3 Tree:**

- If the **root is a 3-node**, then K can be compared with two elements in the root. If it doesn't match with root elements, then we need to search either left subtree, or middle subtree or right subtree.
  - If K is less than the first element in the root, continue searching in the left subtree.
  - If K is in between the two elements of root, continue searching in the middle subtree.
  - If K is larger than the second element of the root, continue searching in the right subtree.

Creating a 2-3 Tree

- If the 2-3 tree is empty, insert a new key at the root level. Except the first node, rest of the nodes are always inserted in the leaf.

- By performing a search for the key to be inserted, appropriate leaf node is found.

- If the leaf is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger then the node’s old key.

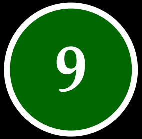
Creating a 2-3 Tree

- If the leaf is a 3-node, we split the leaf into two parts: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, while the middle key is promoted to the old leaf's parent.
- If the tree's root itself is the leaf node, a new root is created to accept the middle key.
- If the promotion of a middle element to its parent leads to a 3-node, it may be required to split along the chain of leaf's ancestor.
- By repeating this procedure, a 2-3 tree can be constructed.

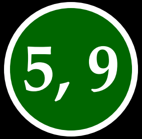
An example of a 2-3 tree construction is given below:

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

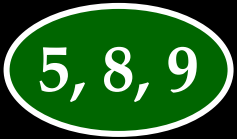
Step1: Insert the element 9 into the empty tree as shown below:



Step2: Element to be inserted is 5. Since there is only one element in the root node, the element 5 is inserted along with 9 as shown below:



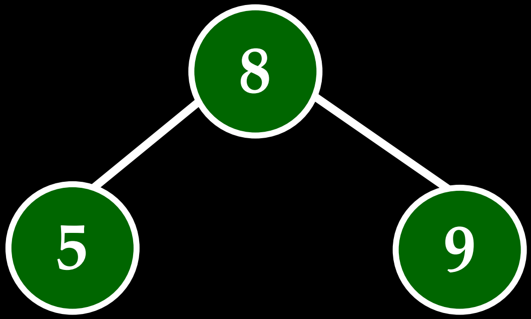
Step3: Element to be inserted is 8. Since 8 is greater than 5 and less than 9, it is inserted between 5 and 9 as shown below:



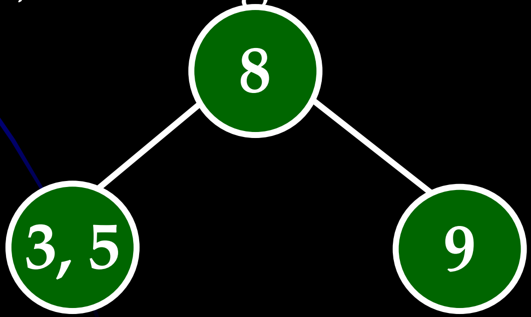
An example of a 2-3 tree construction is given below:

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

Since a node in a 2-3 tree cannot contain more than 2 elements, we move the middle node to the parent position, left element 5 towards left of the parent and right element 9 towards right of the parent as shown below:



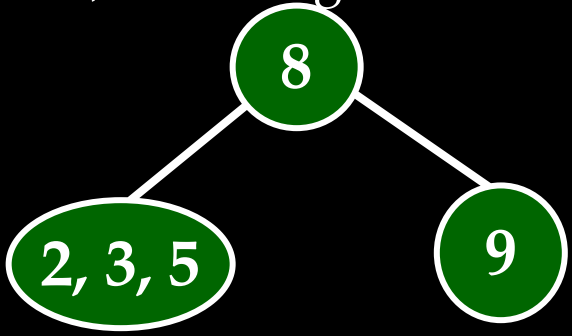
Step4: Element to be inserted is 3. Since 3 is less than 8, it can be inserted towards left of 8, but along with node 5 as shown below:



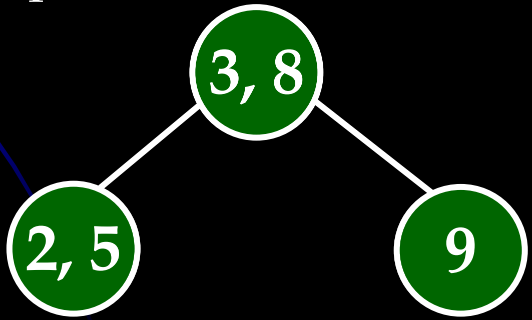
An example of a 2-3 tree construction is given below:

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

Step5: Element to be inserted is 2. Since 2 is less than 8, it can be inserted towards left of 8, but along with nodes 3 and 5 as shown below:



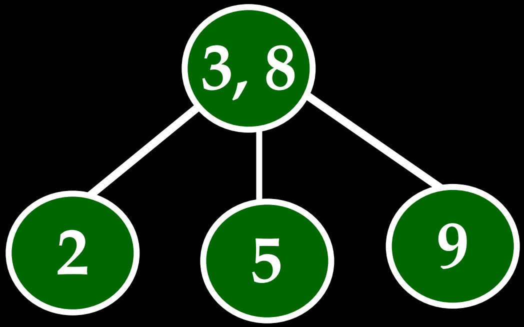
But, a node cannot have more than 2 elements. So, move the middle element 3 to its parent position as shown below:



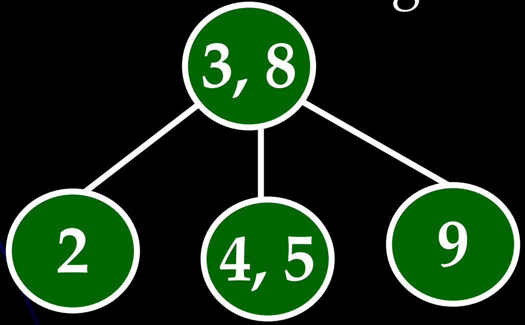
An example of a 2-3 tree construction is given below:

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

But, if a node has two elements i.e., 3, 8 (except the leaf i.e., 2, 5), it should have 3 children: left child containing all elements less than 3, middle child containing all elements between 3 and 8, and the right child containing the elements larger than 8. So, the above tree can be modified as shown below:



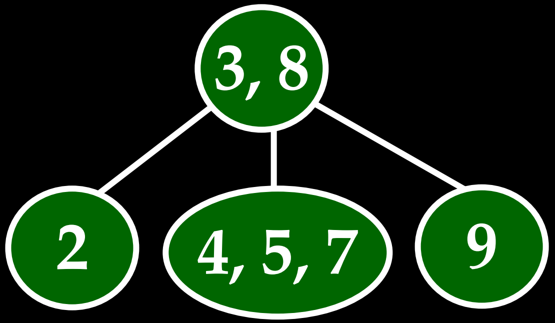
Step 6: Element to be inserted is 4. Since, 4 is between 3 and 8, it can be inserted in the middle node along with 5 as shown below:



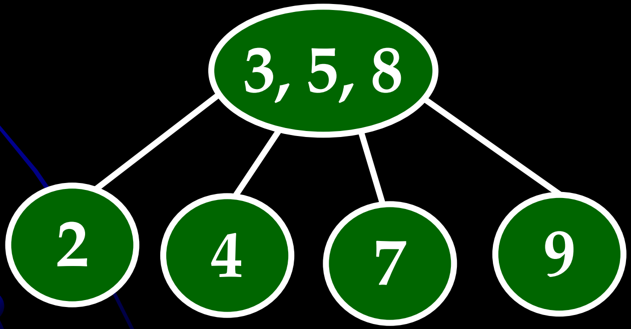
An example of a 2-3 tree construction is given below:

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

Step 7: Element to be inserted is 7. Since, 7 is between 3 and 8, it can be inserted in the middle node along with 4, 5 as shown below:



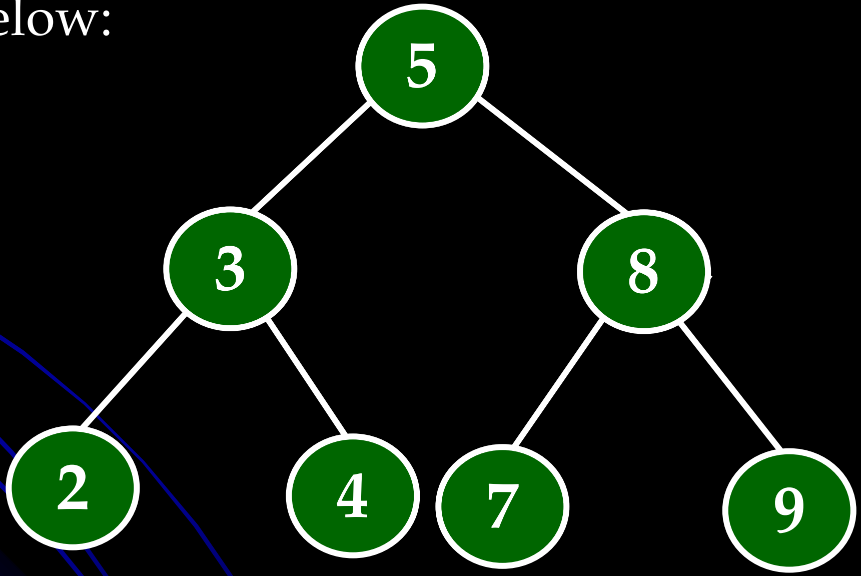
Since a node cannot have more than 2 elements, move 5 to the parent position and the resulting tree is shown below:



An example of a 2-3 tree construction is given below:

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

Since, there are 3 elements namely 3, 5, 8 in the node, it is required to move the middle element to the root. Elements 2 and 4 can be the children of node 3 and the elements 7 and 9 can be the children of node 8 as shown below:



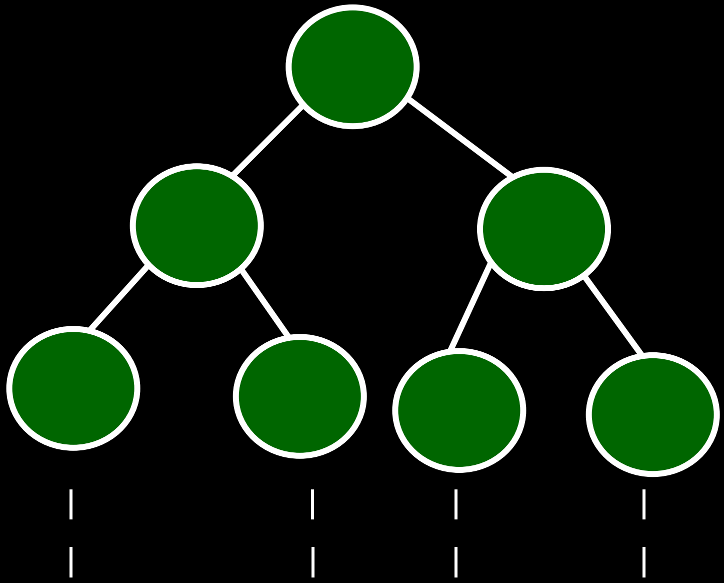
Efficiency of 2-3 Trees

- Since a 2-3 tree is always height balanced, the efficiency of any operation depends on the **height of the tree**.
- To find the time complexity, it is required to find the lower bound as well as the upper bound.
- A 2-3 tree of height *h* with the smallest number of keys is a **full tree of 2-nodes**.
- A 2-3 tree of height *h* with the largest number of keys is a **full tree of 3-nodes**, each with two keys and three children.

Efficiency of 2-3 Trees

- **To find the lower bound for 2-3 tree:** Consider the 2-3 tree with each node exhibiting the property of 2-nodes.

Level 0	Number of nodes at level 0 = 1 = 2^0
Level 1	Number of nodes at level 1 = 2 = 2^1
Level 2	Number of nodes at level 2 = 4 = 2^2
Level *i*	Number of nodes at level *i* = 2^i



- Therefore, for any 2-3 tree of height h with n nodes, we get the inequality:

$$n \geq 2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$$

$$n + 1 \geq 2^{h+1}$$

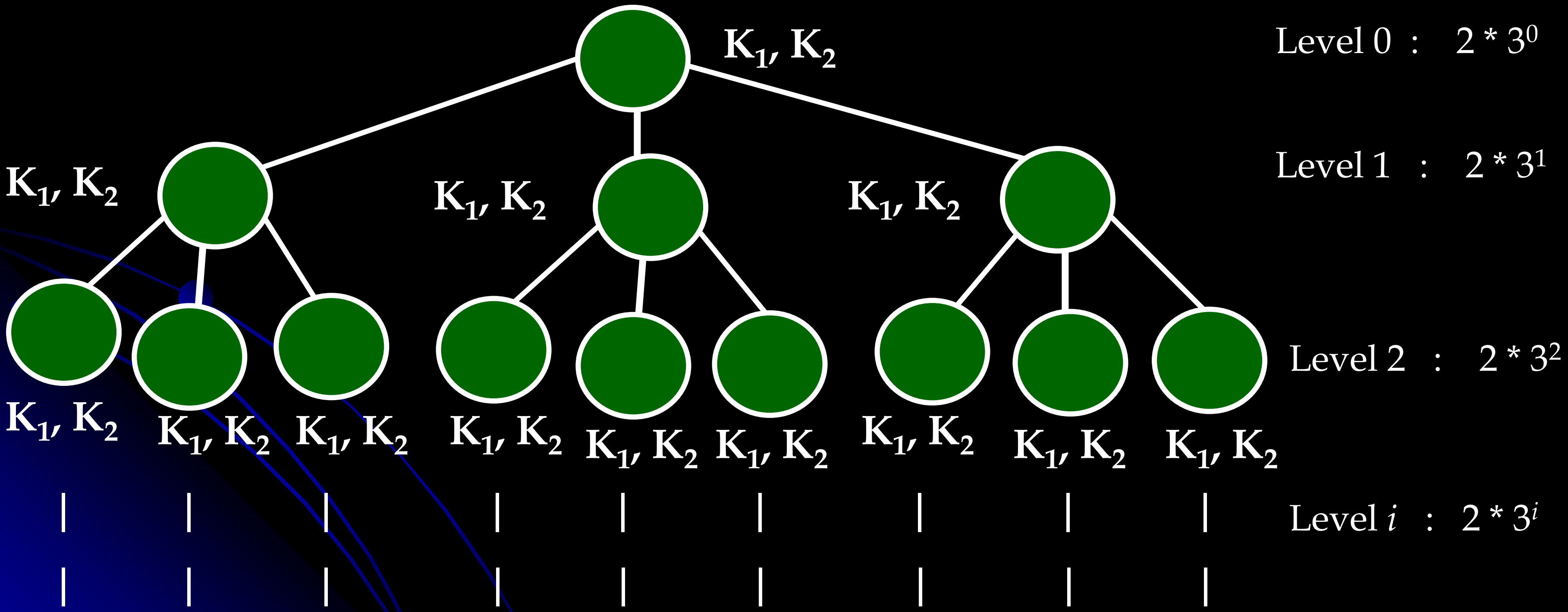
So, taking log on both sides we get,

$$\log_2(n+1) \geq h + 1$$

$$\mathbf{h \leq \log_2(n+1) - 1}$$

Efficiency of 2-3 Trees

- **To find the upper bound for 2-3 tree:** Consider the 2-3 tree with each node exhibiting the property of 3-nodes with each node having maximum of two elements as shown below:



Efficiency of 2-3 Trees

- Therefore, for any 2-3 tree of height *h* with *n* nodes, we get the inequality:

$$n \le 2 . 3^0 + 2 . 3^1 + 2 . 3^2 + \ ... + 2 . 3^h$$

$$= 2(3^0 + 3^1 + 3^2 + 3^h) = 3^{h+1} - 1 \qquad \left(\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \right)$$

$$n + 1 \le 3^{h+1}$$

So, taking log on both sides we get,

$$\log_3 (n + 1) \le h + 1$$

- These lower and upper bounds on height *h*,

$$\log_3 (n + 1) - 1 \le h \le \log_2 (n + 1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in **Θ (log *n*) in both the worst and average case**.

Heaps

- A **heap** is an ordered data structure which is suitable for implementing ***priority queue*** (Using priority queue, elements can be inserted into queue based on the priority and elements can be deleted from the queue based on the priority) and also suitable for important sorting technique called heap sort.

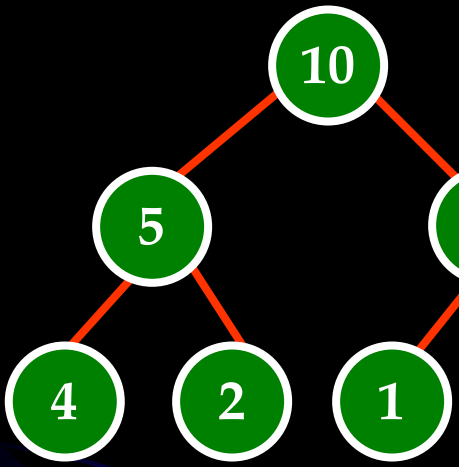
Notion of the Heap

- **DEFINITION:** A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

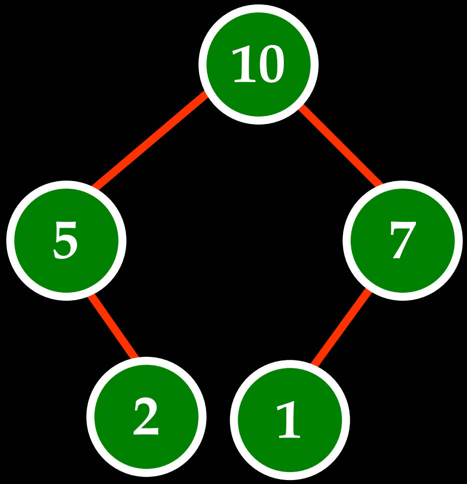
1. *The tree's shape requirement* - the binary tree is almost complete or complete. i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. *The parental dominance requirement*
   - the key at each node is greater than or equal to the keys at its children. We call this variation a **max-heap**.

     or
   - The key at each node is smaller than or equal to the keys at its children. We call this variation a **min-heap**.

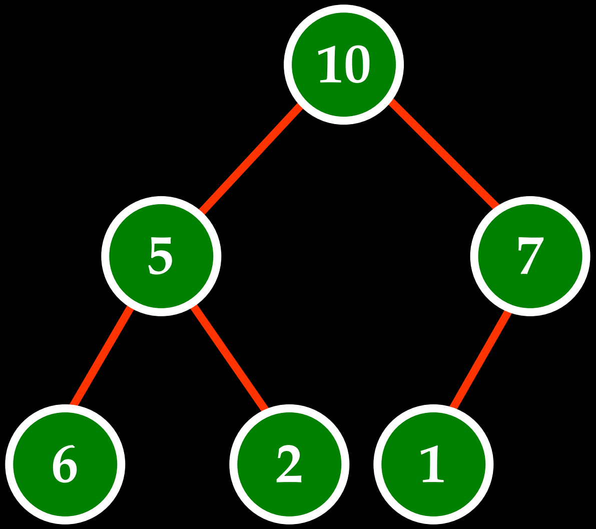
Heaps



(a) Heap



(b) Not a Heap since tree's shape requirement is violated



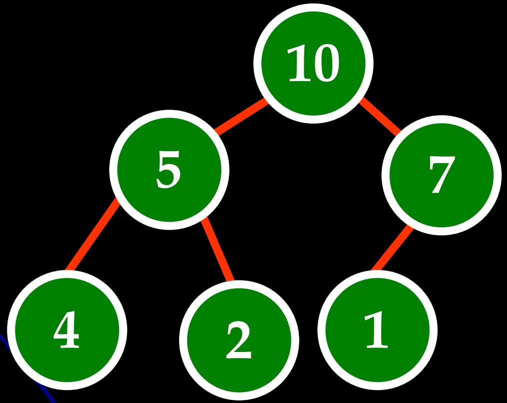
(c) Not a Heap since the parental dominance requirement fails for the node with key 5.

Figure: Illustration of the definition of "heap": only the leftmost tree is a heap.

Note: Key values in a heap are ordered top down. There is no left-to-right order in key values. i.e., there is no relationship among key values for nodes either on same level of tree or in the left and right subtree of same node.

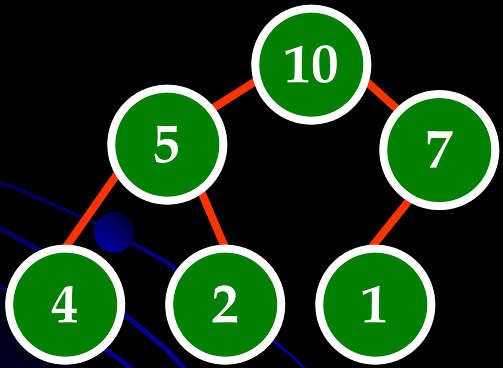
Important properties of Heaps

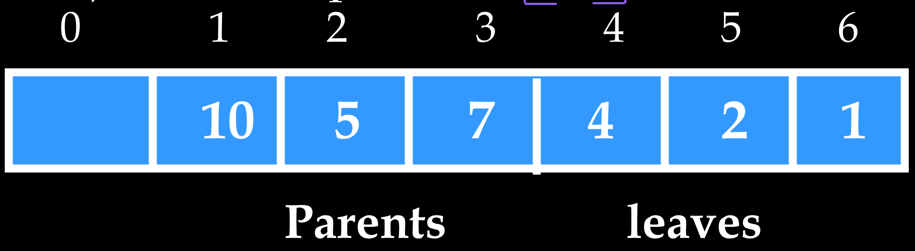
1. There exists exactly one almost complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.



Important properties of Heaps

4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ unused. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i ($1 \le i \le \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i + 1$, and correspondingly, the parent of a key in position i ($2 \le i \le n$) will be in position $\lfloor i/2 \rfloor$.





0	1	2	3	4	5	6
	10	5	7	4	2	1

Array Representation of heap shown in figure.

Thus, we could also define a heap as an array $H[1 \ldots n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max\{ H[2i], H[2i + 1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor .$$

Construction of a Heap

Bottom-up heap construction algorithm:

It initializes the almost complete binary tree with *n* nodes by placing keys in the order given and then "heapifies" the tree as follows:

- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key at this node.
- If it doesn't, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.
- This process continues until the parental dominance requirement for K is satisfied.
- After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.
- The algorithm stops after this is done for the tree's root.

Construction of Heap

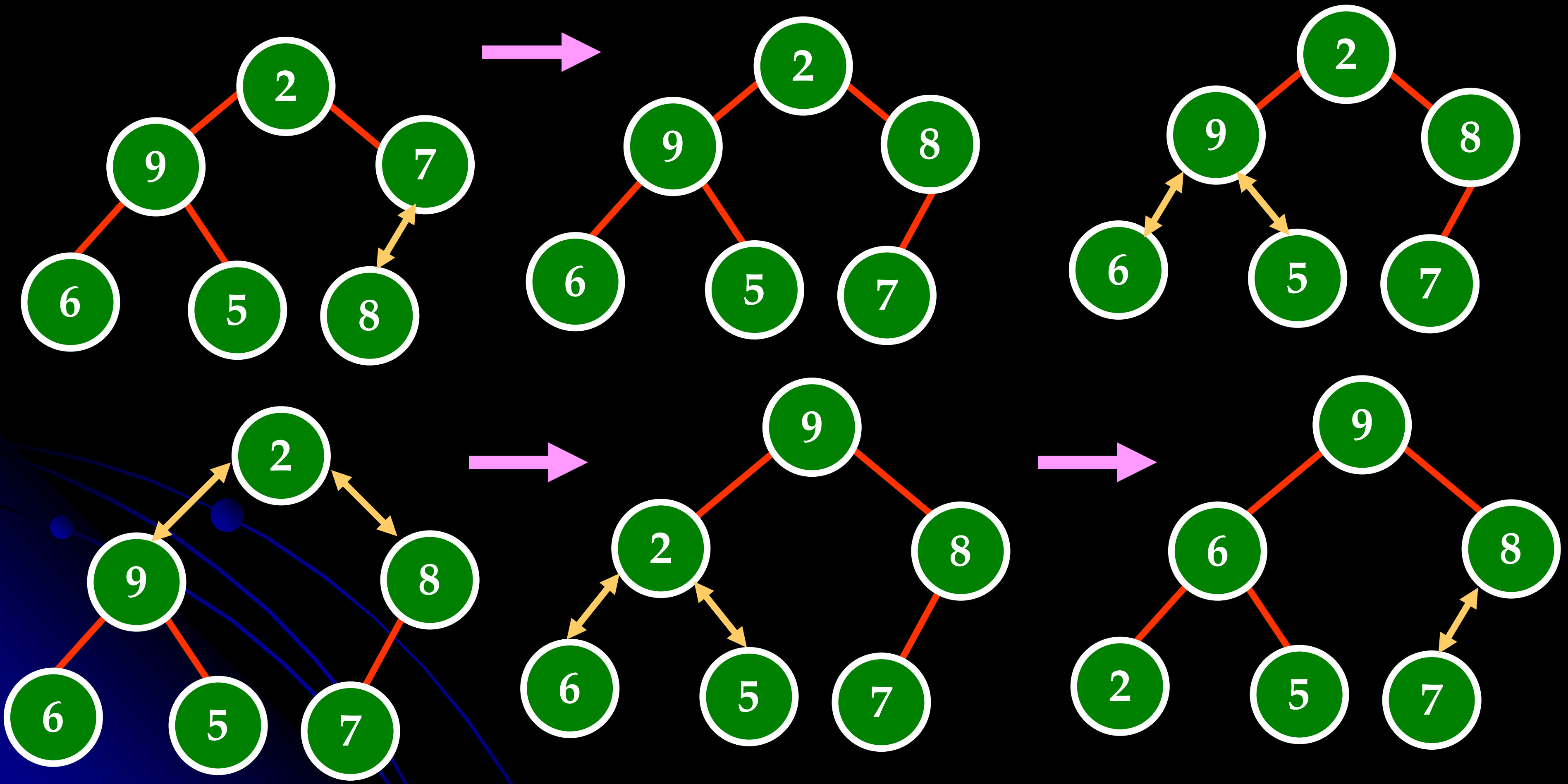
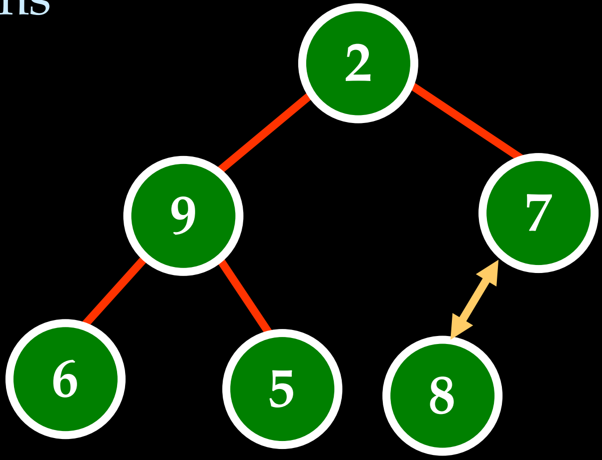
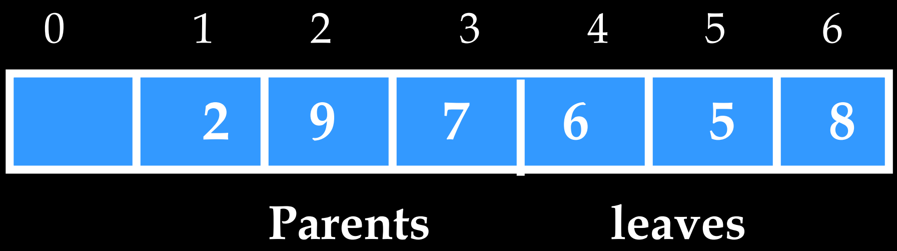


Figure: Bottom-up construction of heap for the list 2, 9, 7, 6, 5, 8

Bottom-up Heap Construction

```
ALGORITHM    HeapBottomUp ($H[1 ... n]$)
    //Constructs a heap from the elements of a given array
    //by the bottom-up algorithm
    //Input: An array $H[1 ... n]$ of orderable items
    //Output: A heap $H[1 ... n]$
    for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do
        $k \leftarrow i$; $v \leftarrow H[k]$
        heap $\leftarrow$ false
        while not heap and $2 * k \leq n$ do
            $j \leftarrow 2 * k$
            if $j < n$        //there are two children
                if $H[j] < H[j+1]$  $j \leftarrow j+1$
            if $v \geq H[j]$
                heap $\leftarrow$ true
            else $H[k] \leftarrow H[j]$; $k \leftarrow j$
        $H[k] \leftarrow v$
```





Efficiency of Bottom-up algorithm in the worst case:

- Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the maximum number of nodes occurs on each level.
- Let h be the height of the tree, where $h = \lfloor \log_2 n \rfloor$ (or just $\lceil \log_2(n+1) \rceil - 1 = k - 1$).
- Each key on level i of the tree will traverse to the leaf level h in the worst case of the heap construction algorithm.

Efficiency of Bottom-up algorithm in the worst case:

- Since moving to next level down requires two comparisons – one to find the larger child and the other to determine whether the exchange is required, the total number of key comparisons involving a key on level i will be $2(h - i)$.

- Therefore, the total number of key comparisons in the worst case will be

$$C_{\text{worst}}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h - i) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i$$

$$= 2 \left[\sum_{i=0}^{h-1} h 2^i - \sum_{i=0}^{h-1} i\, 2^i \right]$$

Efficiency of Bottom-up algorithm in the worst case:

$$= 2\left[h\sum_{i=0}^{h-1} 2^i - ((h-2).2^h + 2)\right] \qquad \left(\sum_{i=1}^{h} i2^i = (h-1)2^{h+1} + 2\right)$$

$$= 2[h(2^h - 1) - ((h-2).2^h + 2)] \qquad \left(\sum_{i=0}^{h} 2^i = 2^{h+1} - 1\right)$$

$$= 2[h2^h - h - h2^h + 2.2^h - 2] = 2[-h + 2^{h+1} - 2]$$

$$= 2[-\log_2(n+1) + 1 + n + 1 - 2] \quad (n = 2^{h+1} - 1\text{, therefore, } h = \log_2(n+1) - 1)$$

$$C_{\text{worst}}(n) = 2(n - \log_2(n+1))$$

- Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than **2n comparisons**.

Construction of a Heap

- The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap. It is called as **top-down heap construction algorithm**.

- **To insert a new key K into a heap**, first attach a new node with key K in it after the last leaf of the existing heap.

- Then shift K up to its appropriate place in the new heap as follows:
  - Compare K with its parent's key: if the latter is greater than or equal to K, stop ; otherwise, swap these two keys and compare K with its new parent.
  - This swapping continues until K is not greater than its last parent or it reaches the root.

- This insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the **time efficiency of insertion is in O(log n)**.

Construction of a Heap using Top-down algorithm

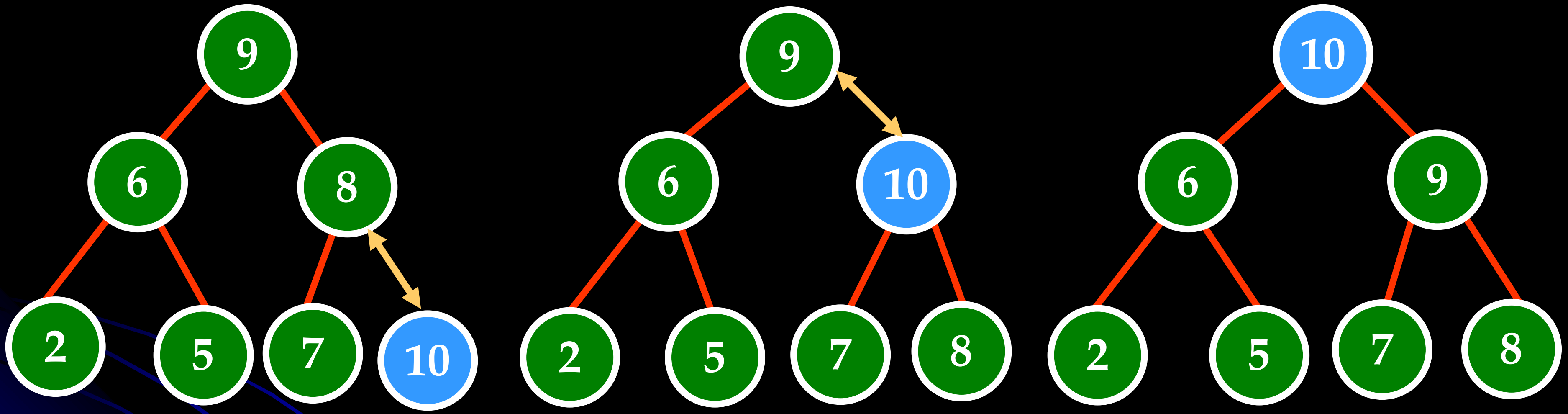


Figure: Inserting a key (10) into the heap. The new key is shifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Deletion from a Heap

- Consider the deletion of the root's key.

  Maximum Key Deletion from heap

  Step1 Exchange the root's key with the last key K of the heap.

  Step2 Decrease the heap's size by 1.

  Step3 "Heapify" the smaller tree by shifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. i.e., verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

- The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after swap has been made and size of the tree is decreased by 1. The t**ime efficiency of deletion is in O(log *n*)** as well.

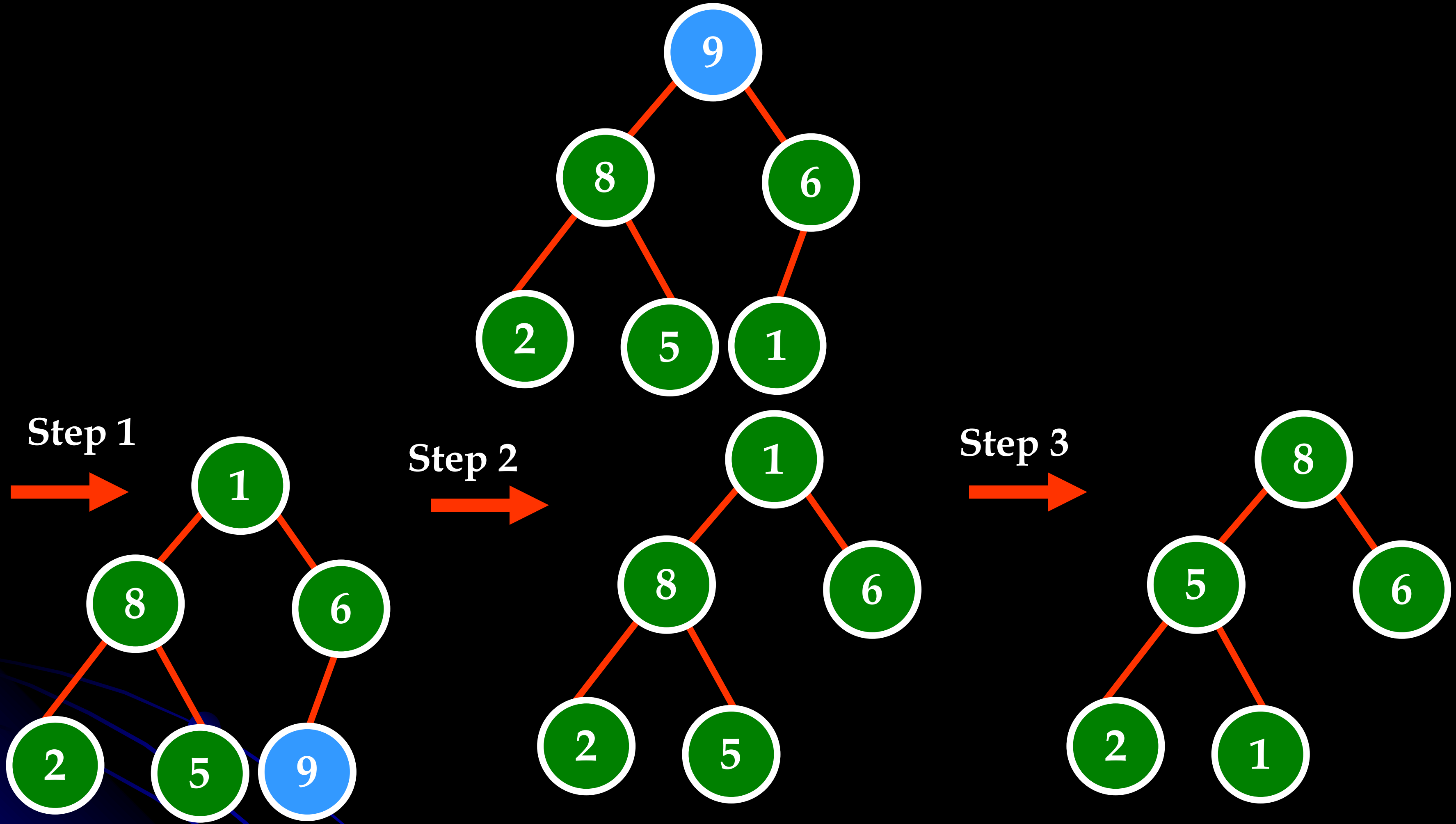


Figure: Deleting root's key from heap. The key to be deleted is swapped with the last key, after which the smaller tree is "heapified" by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

HEAPSORT

- This sorting algorithm was discovered by ***J. W. Williams***.

- This is a **two-stage algorithm** that works as follows:

Stage 1 (**Heap construction**): Construct a heap for a given array.

Stage 2 (**Maximum deletions**): Apply the root-deletion operation n-1 times to the remaining heap.

- As a result, the array elements are *eliminated in decreasing order*.

- But since under the array implementation of heaps, an element being deleted is placed last, the resulting array will be exactly the original array sorted in ascending order.

HEAPSORT

Stage 1 (Heap construction)

2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

9 6 8 2 5 7

Stage 2 (Maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5

5 6 7 2 | 8

7 6 5 2

2 6 5 | 7

6 2 5

5 2 | 6

5 2

2 | 5

2

Figure: Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

HEAPSORT

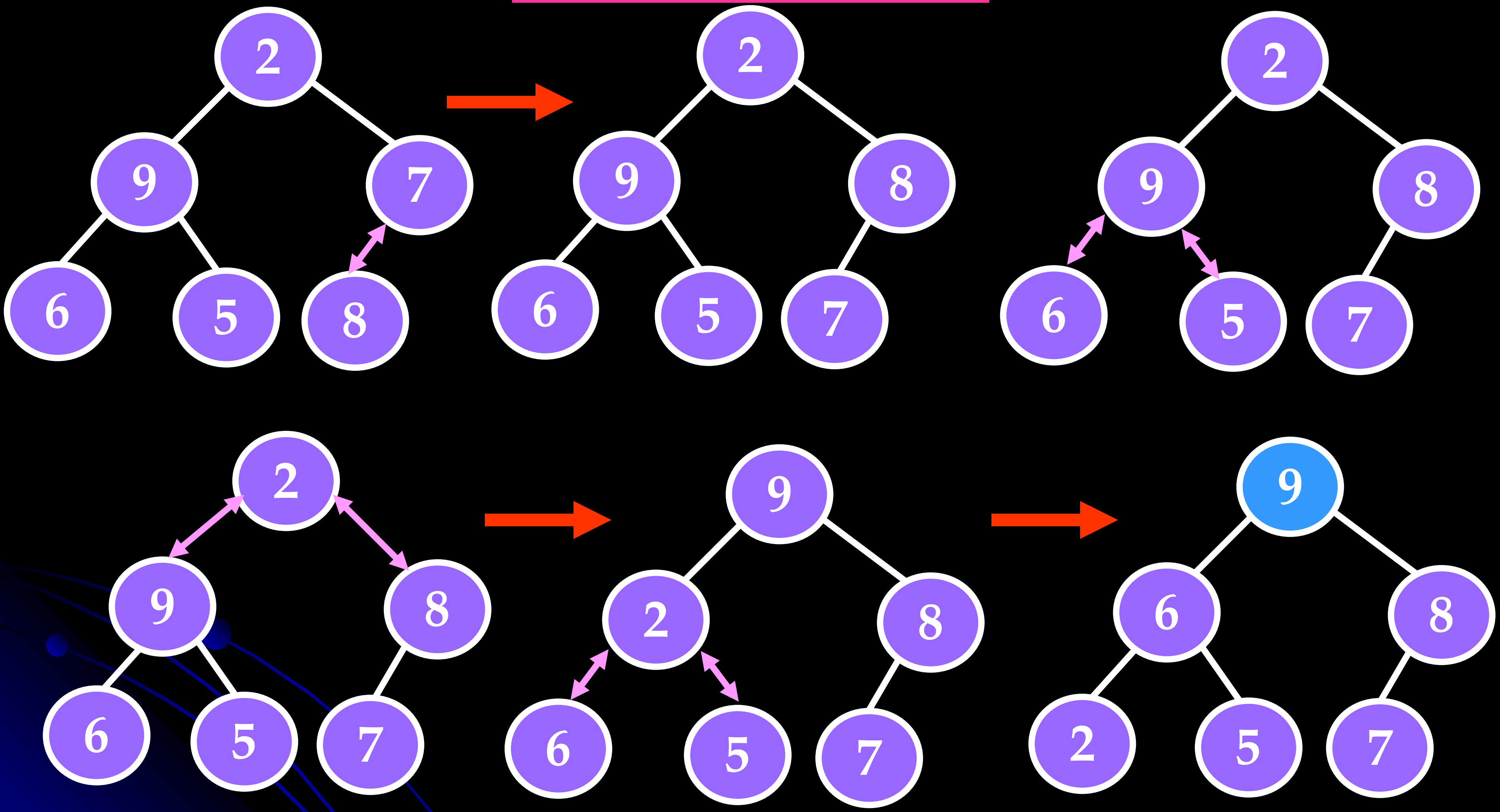
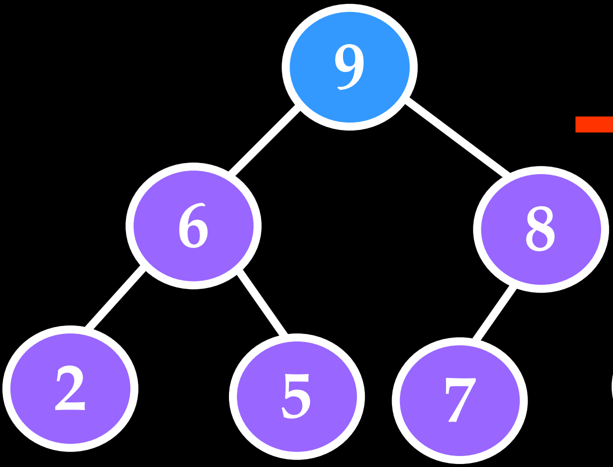


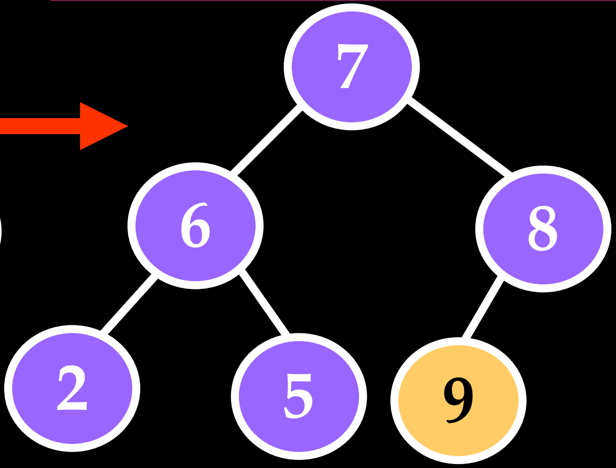
FIGURE: Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.

After "Heapification", the list becomes 9, 6, 8, 2, 5, 7.

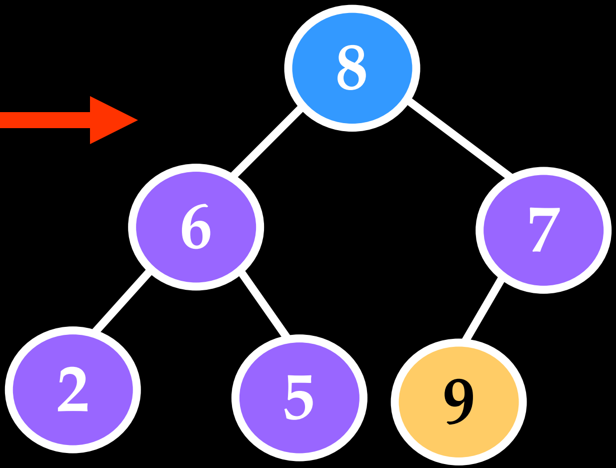
HEAPSORT



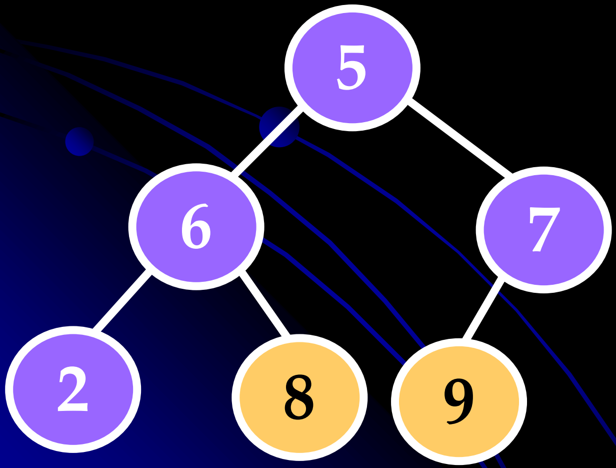
HEAP: Exchange 9 and 7



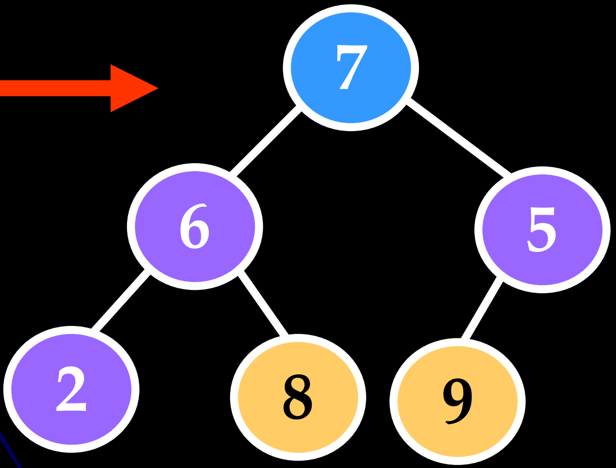
Heapify the list 7, 6, 8, 2, 5



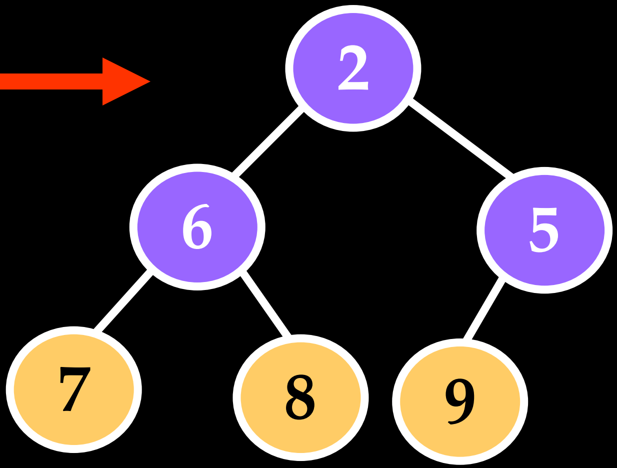
HEAP: Exchange 8 and 5



Heapify the list 5, 6, 7, 2

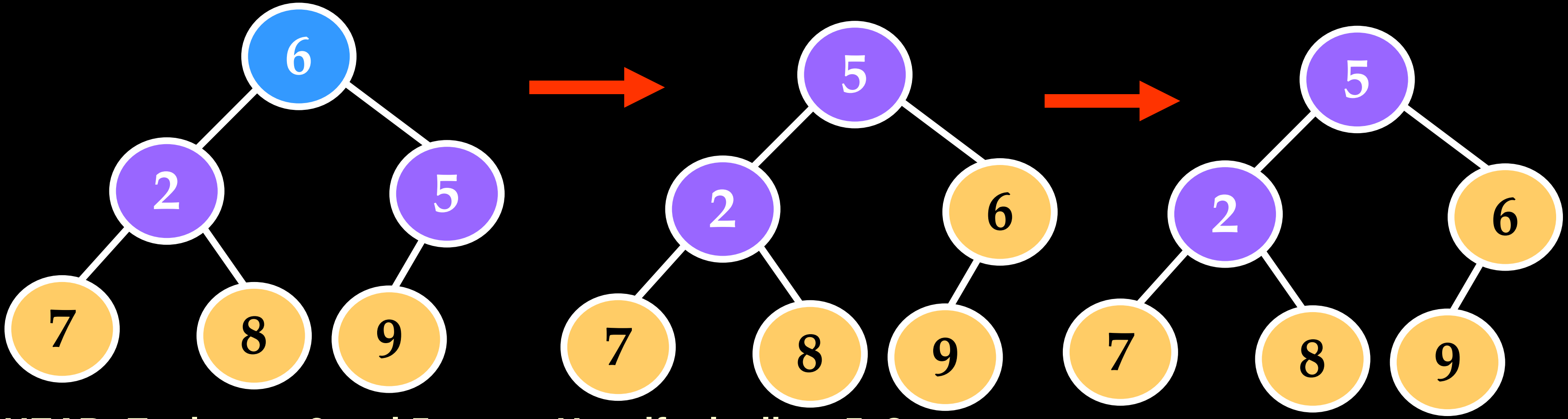


HEAP: Exchange 7 and 2



Heapify the list 2, 6, 5

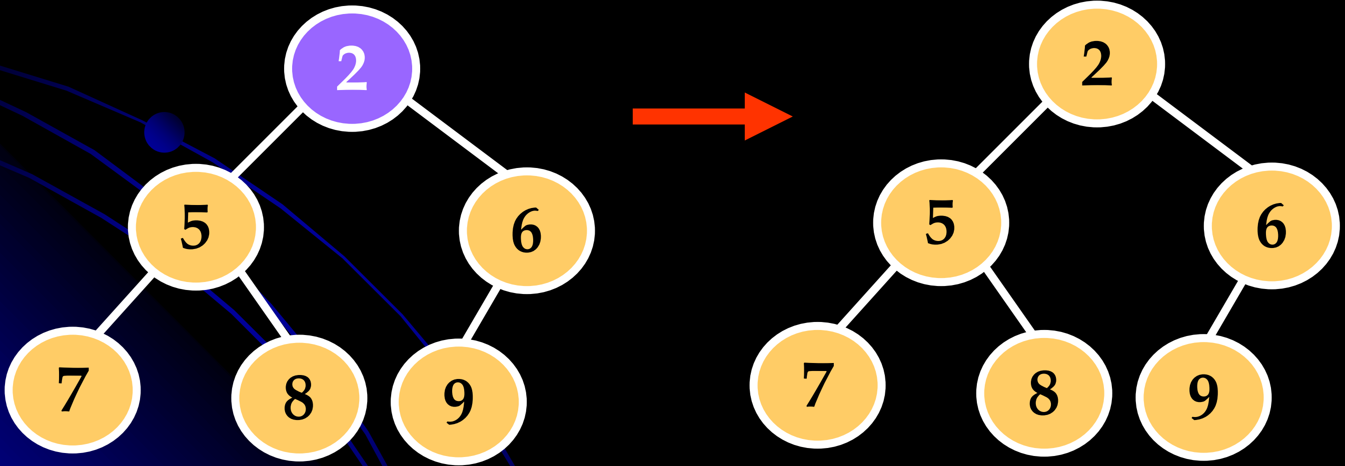
HEAPSORT



HEAP: Exchange 6 and 5

Heapify the list 5, 2

HEAP: Exchange 5 and 2



Sorted list 2, 5, 6, 7, 8, 9

Analysis of Heapsort using bottom-up approach

- The **heap construction stage** of the algorithm is in **O(*n*).**
- Now, it is required to **analyze only the second stage**.

  ($h = \log_2 n$. No. of key comparisons $= 2h = 2\log_2 n$)

  - Observe that after exchanging the root with (n-1)th item, we have to reconstruct the heap for (n-1) elements which depends only on height of the tree.
  - So, time complexity to reconstruct the heap for (n-1) elements $= 2 \log_2(n-1)$
  - After exchanging 0th item with n-2 item, time complexity to reconstruct the heap $= 2 \log_2 (n-2)$
  - After exchanging 0th item with n-3 item, time complexity to reconstruct the heap $= 2 \log_2 (n-3)$

  . . .   . . .

  - After exchanging 0th item with 1st item, time complexity to reconstruct the heap $= 2 \log_2 1$

Analysis of Heapsort using bottom-up approach

- So, for the number of key comparisons, $C(n)$, needed for eliminating the root keys from heaps of diminishing sizes from n to 2 is given by the inequality:

$$C(n) \leq 2\log_2(n-1) + 2\log_2(n-2) + 2\log_2(n-3) + \ldots + 2\log_2 1$$

$$C(n) \leq 2\sum_{i=1}^{n-1} \log_2 i = 2(n-1)\log_2(n-1) \leq 2n\log_2 n. \quad [\sum_{i=1}^{n} \lg i \approx n\lg n]$$

Therefore,

the time complexity for the second stage = $2n\log_2 n \approx \boldsymbol{n\log_2 n}$.

Analysis of Heapsort using Bottom-up approach

- So, the time complexity of heap sort =

Time complexity of stage 1 + time complexity of stage 2

$$= O(n) + O(n \log_2 n)$$
$$= O(\max\{ n, n \log_2 n \})$$
$$= O(n \log_2 n)$$

Therefore, the time complexity of heapsort = $O(n\log_2 n)$

End of Chapter 6