AVL Trees

- **Adelson-Velsky, Landis, 1962**

- The **balance factor** of a node in a BST is **the difference of the heights of its left and right subtrees**.

- An **AVL tree** is a BST in which the balance factor of every node is either 0 or $+1$ or -1.

- The height of the empty tree is -1.

AVL Trees

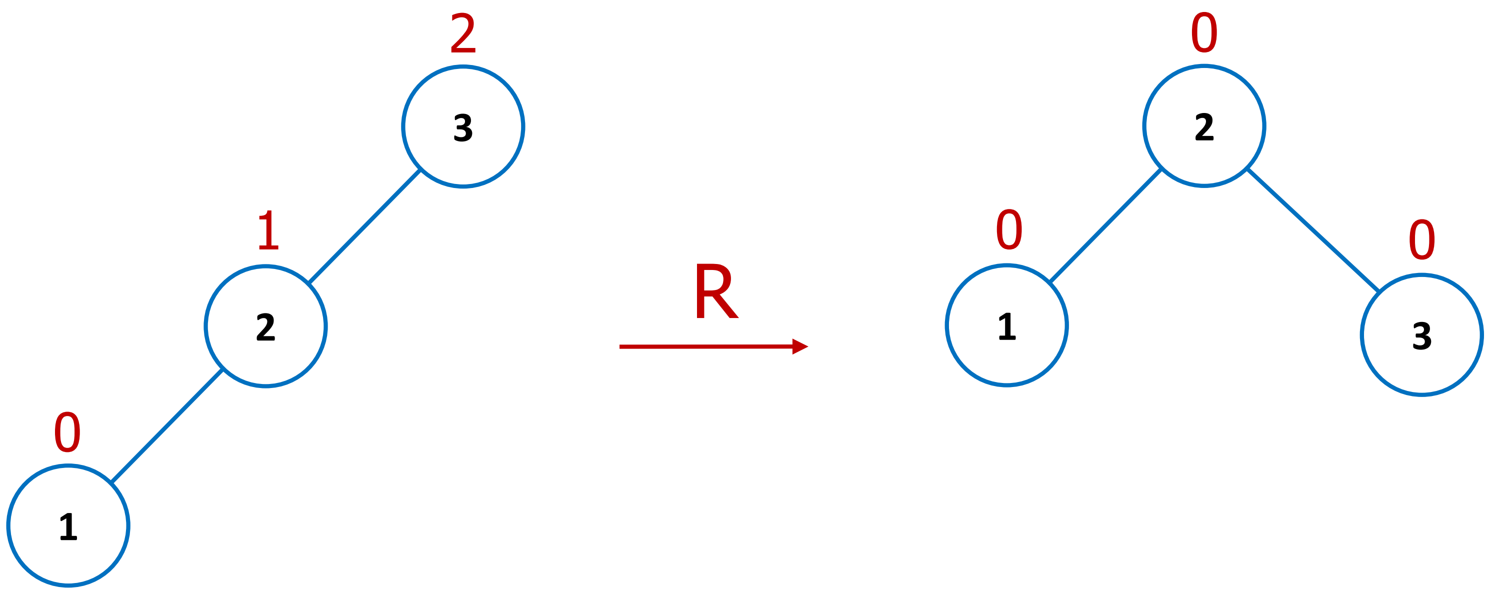
- If an **insertion** of a new node or an **deletion** of a node makes an AVL tree **unbalanced**, we transform the tree by a **rotation**.

- A **rotation** in an AVL tree is a **local transformation** of its subtree rooted at a node whose **balance factor** became either $+2$ or -2.

- If there are **several** such nodes, we rotate the subtree rooted at the unbalanced node that is the **closest to the newly inserted leaf**.

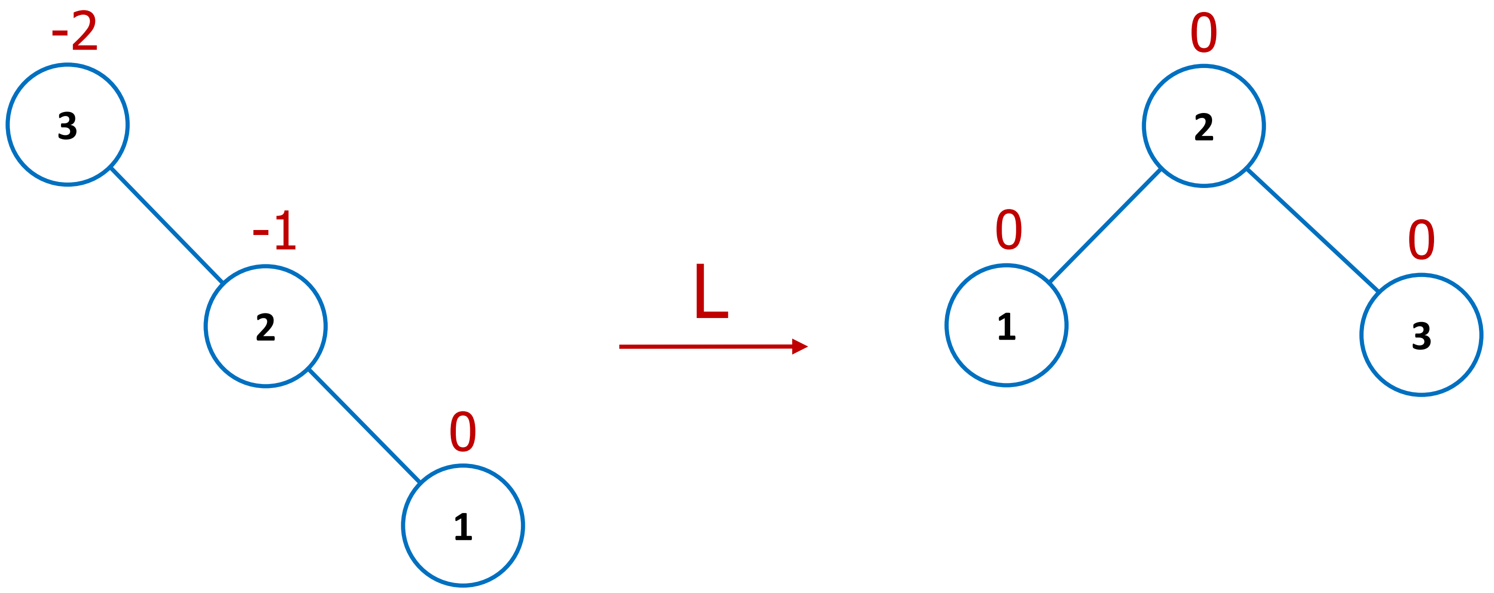
Types of Rotations

Single right rotation (R-rotation)



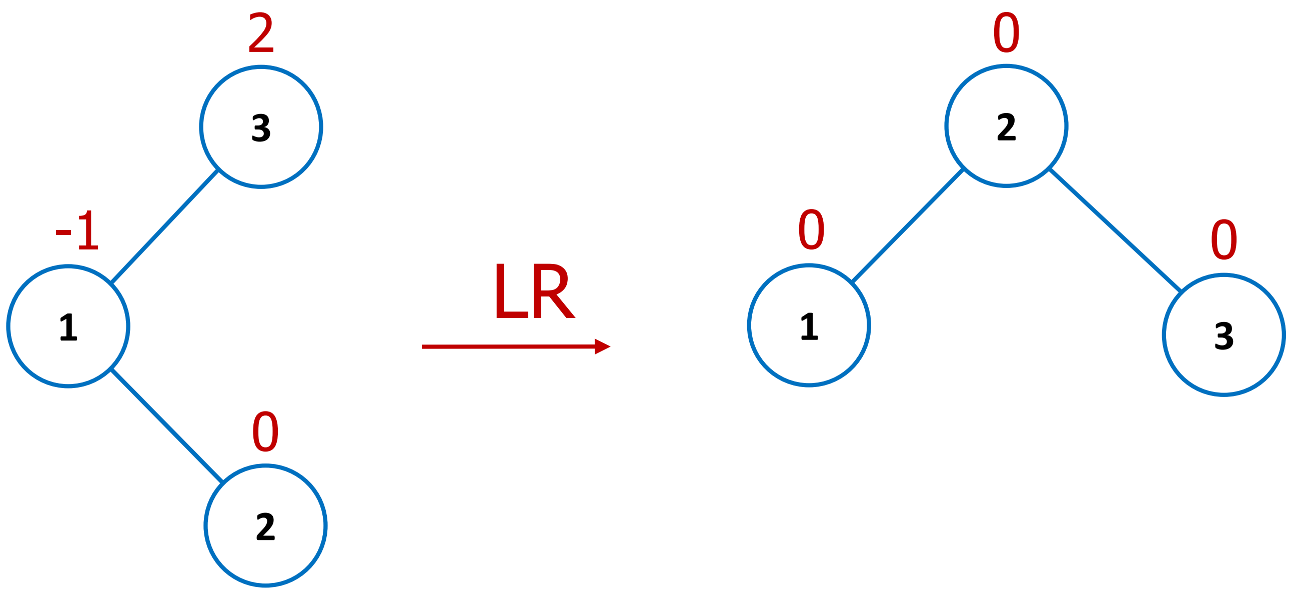
Types of Rotations

Single left rotation (L-rotation)



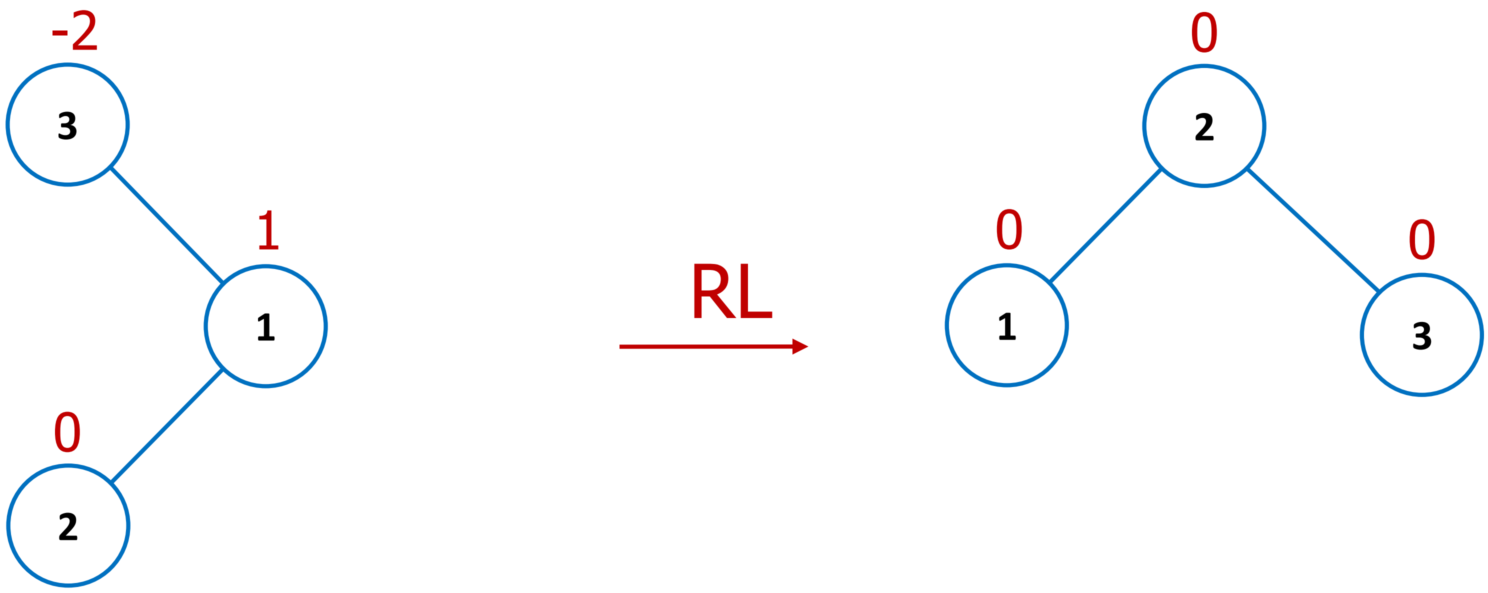
Types of Rotations

Double left-right rotation (LR-rotation)



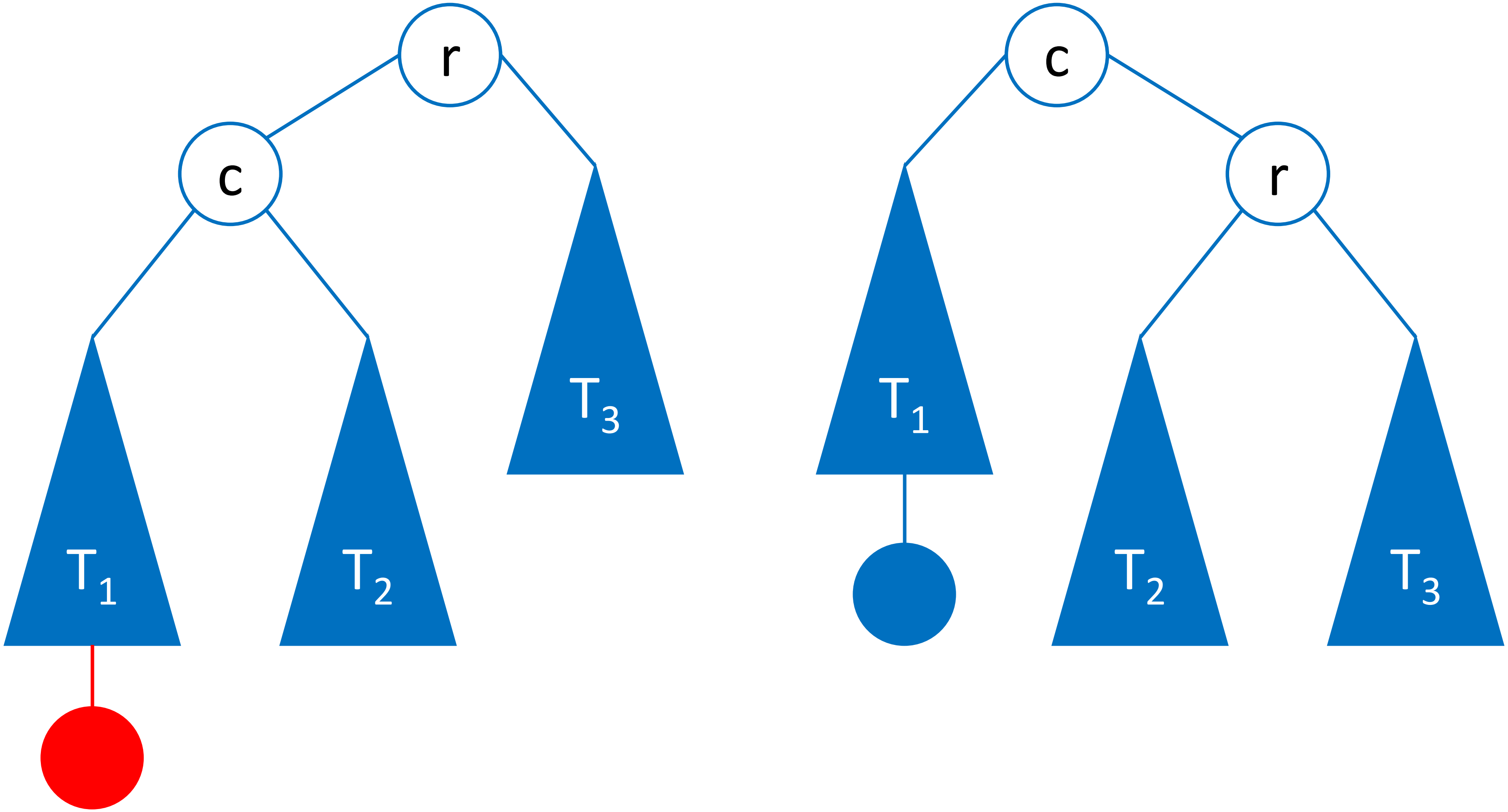
Types of Rotations

Double right-left rotation (RL-rotation)



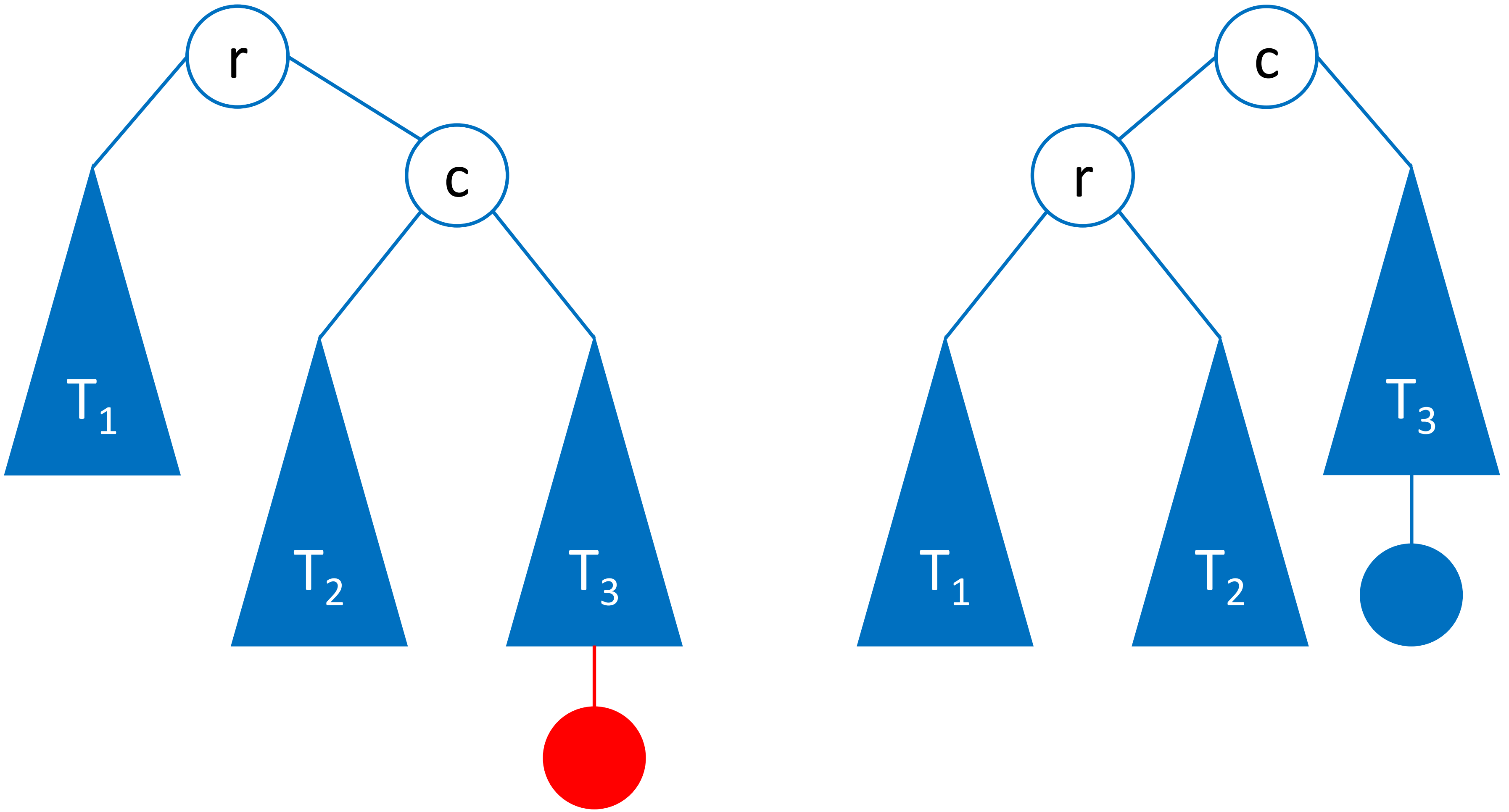
Types of Rotations

Single right rotation (R-rotation)



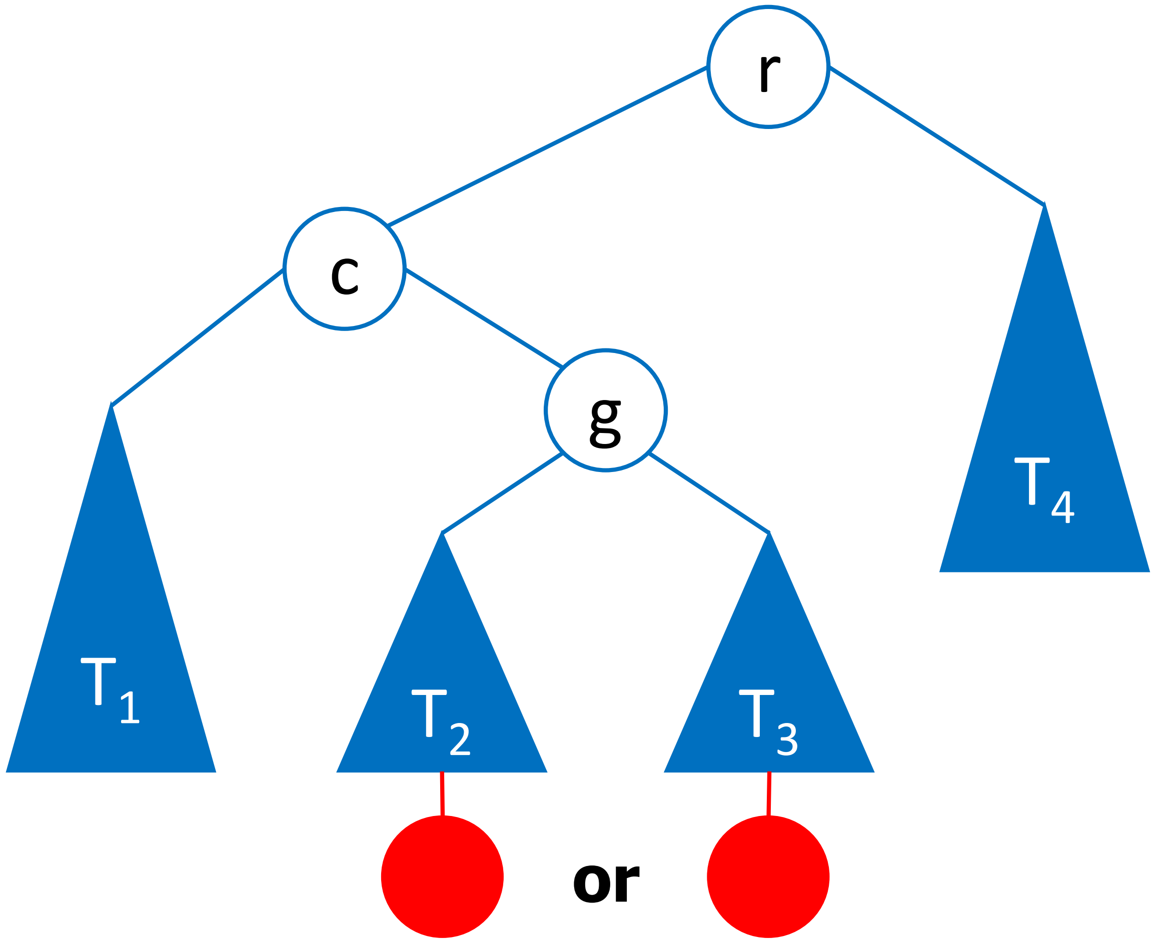
Types of Rotations

Single left rotation (L-rotation)



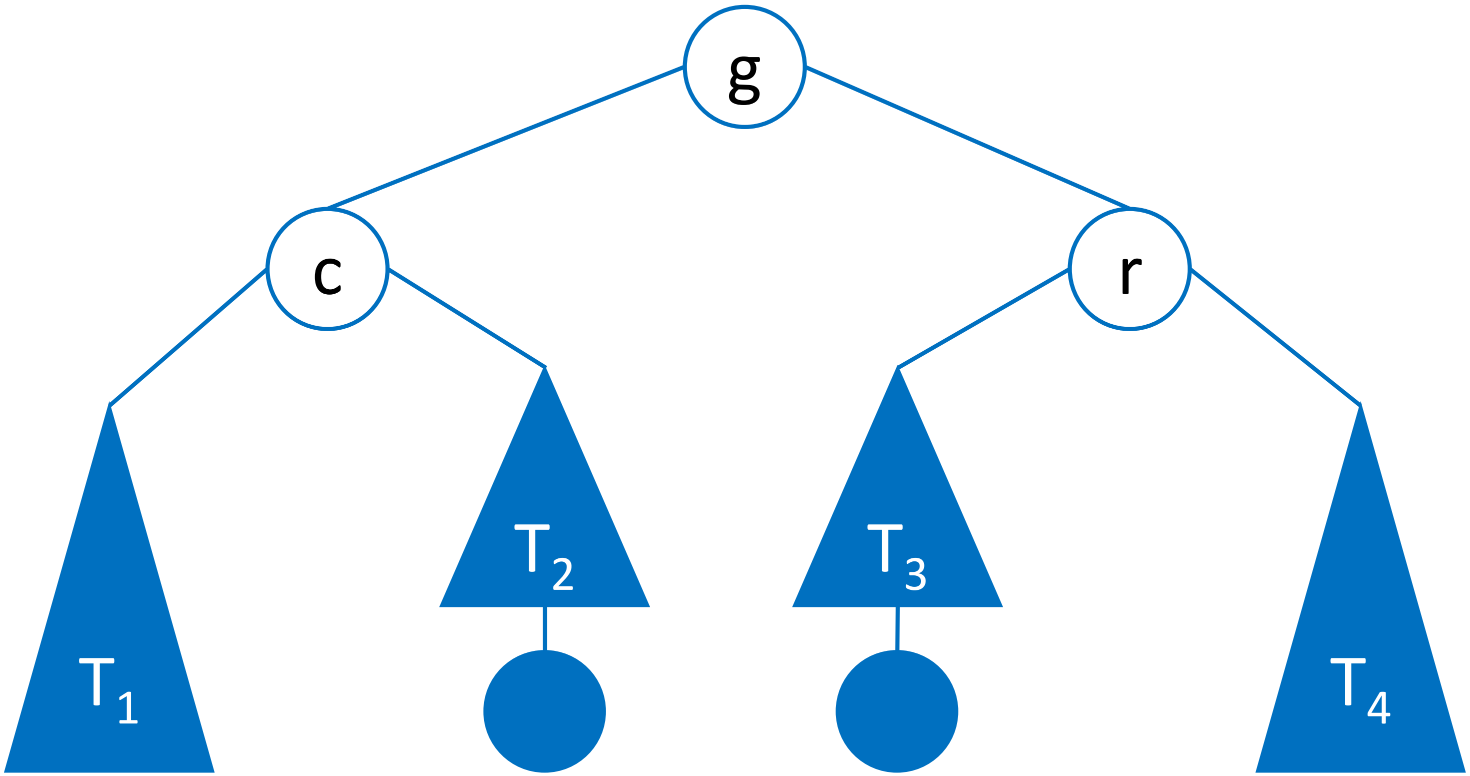
Types of Rotations

Double left-right rotation (LR-rotation)



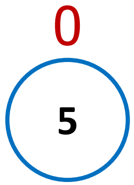
Types of Rotations

Double left-right rotation (LR-rotation)



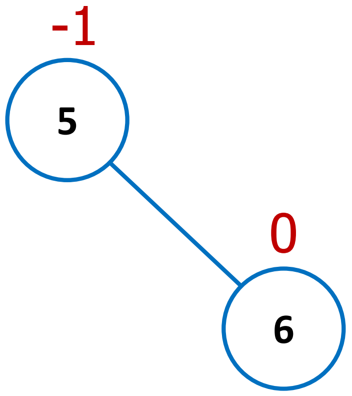
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



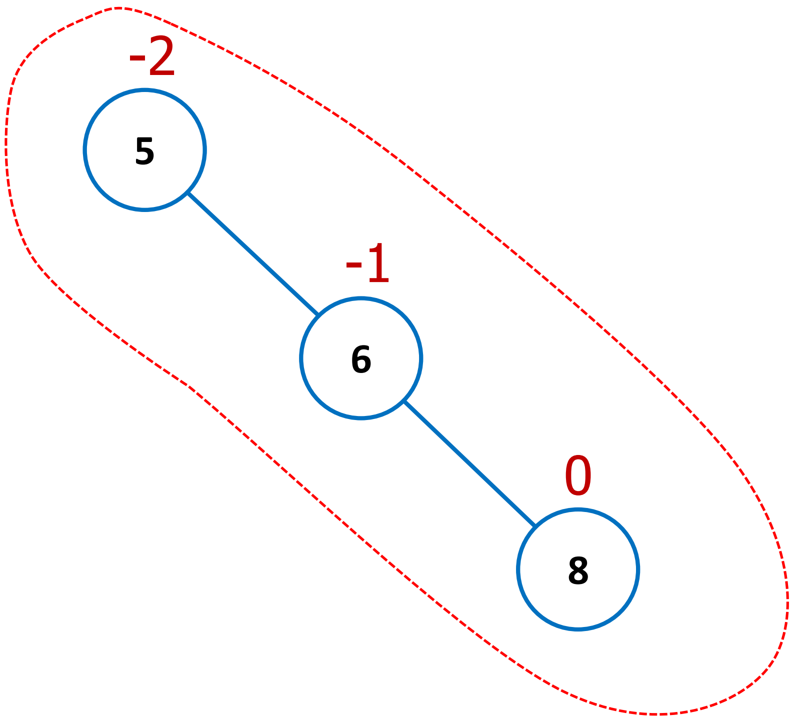
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



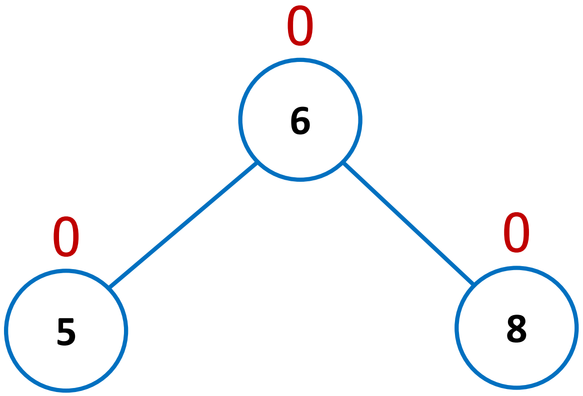
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



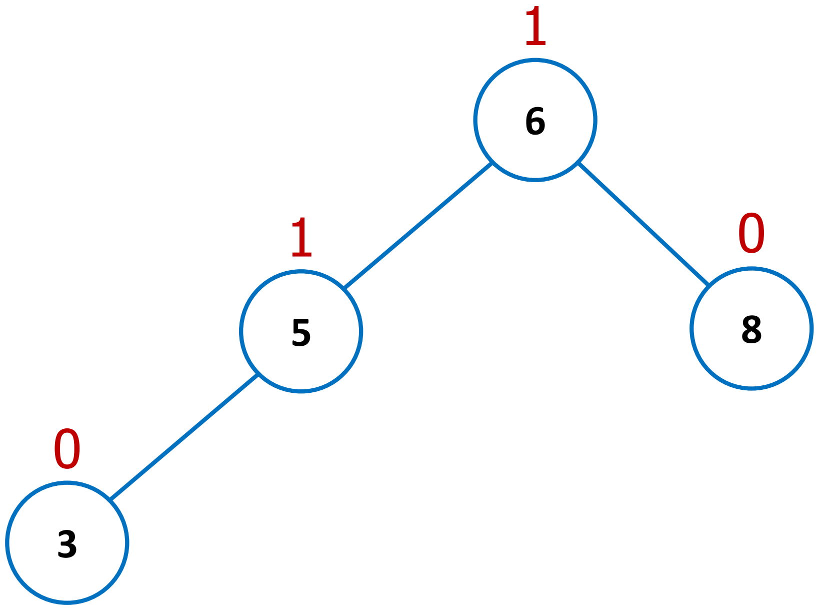
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



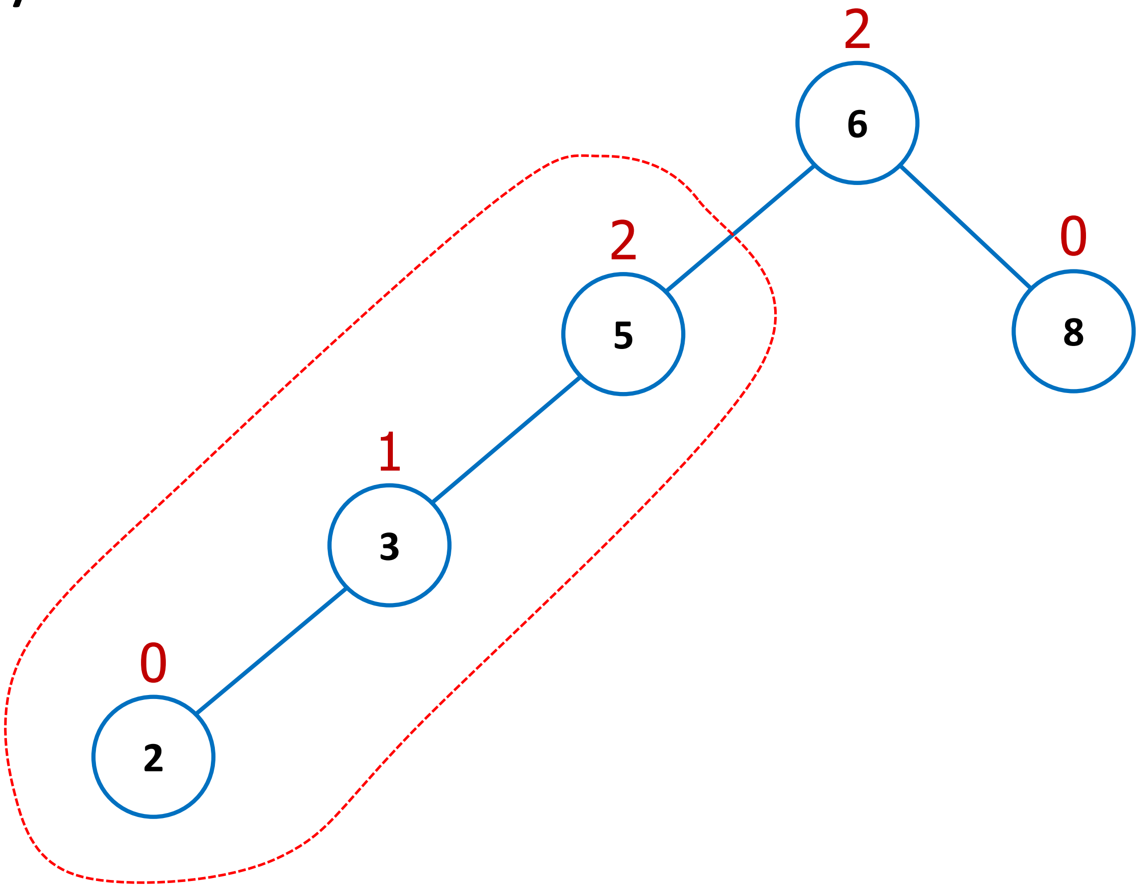
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



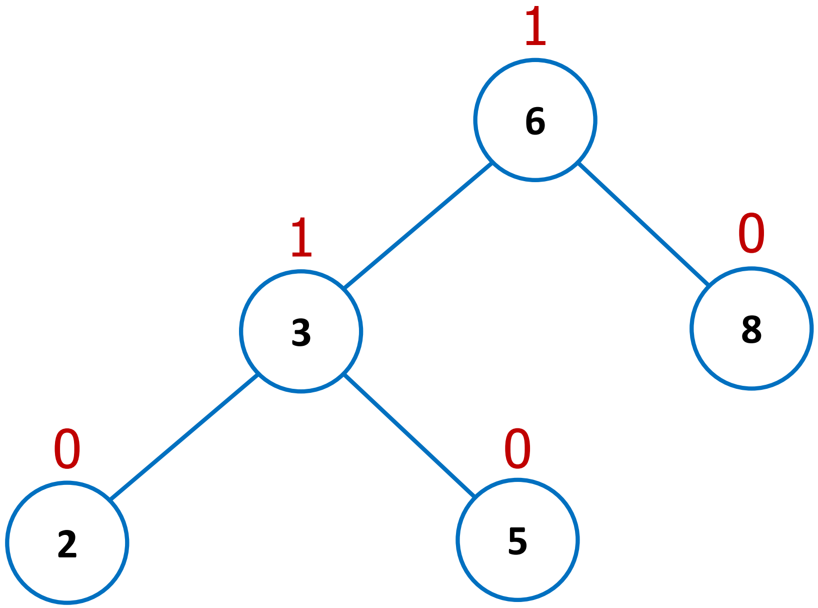
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



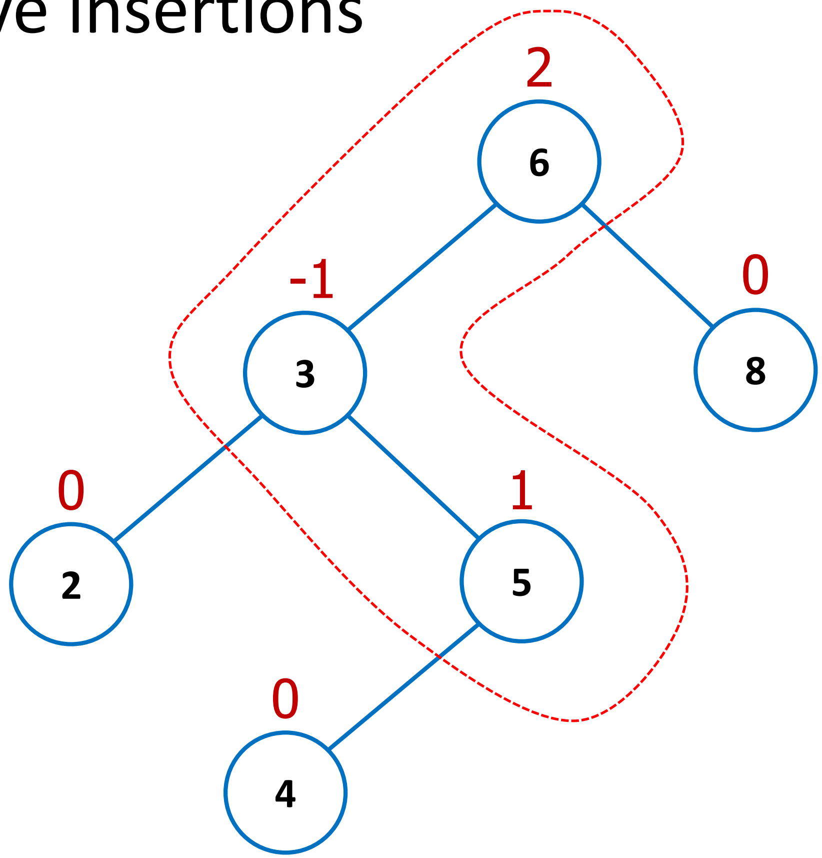
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



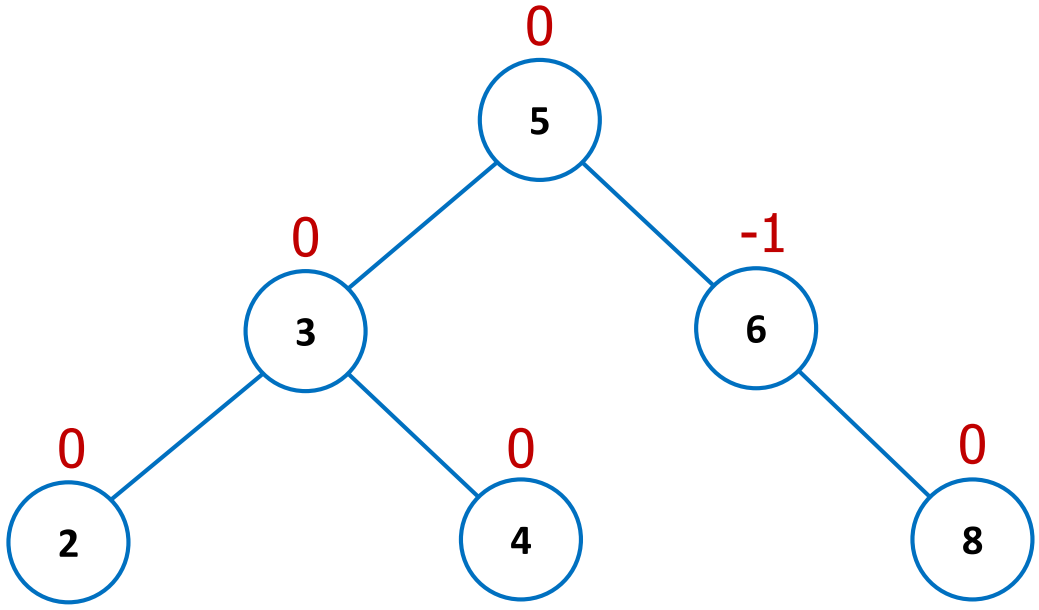
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



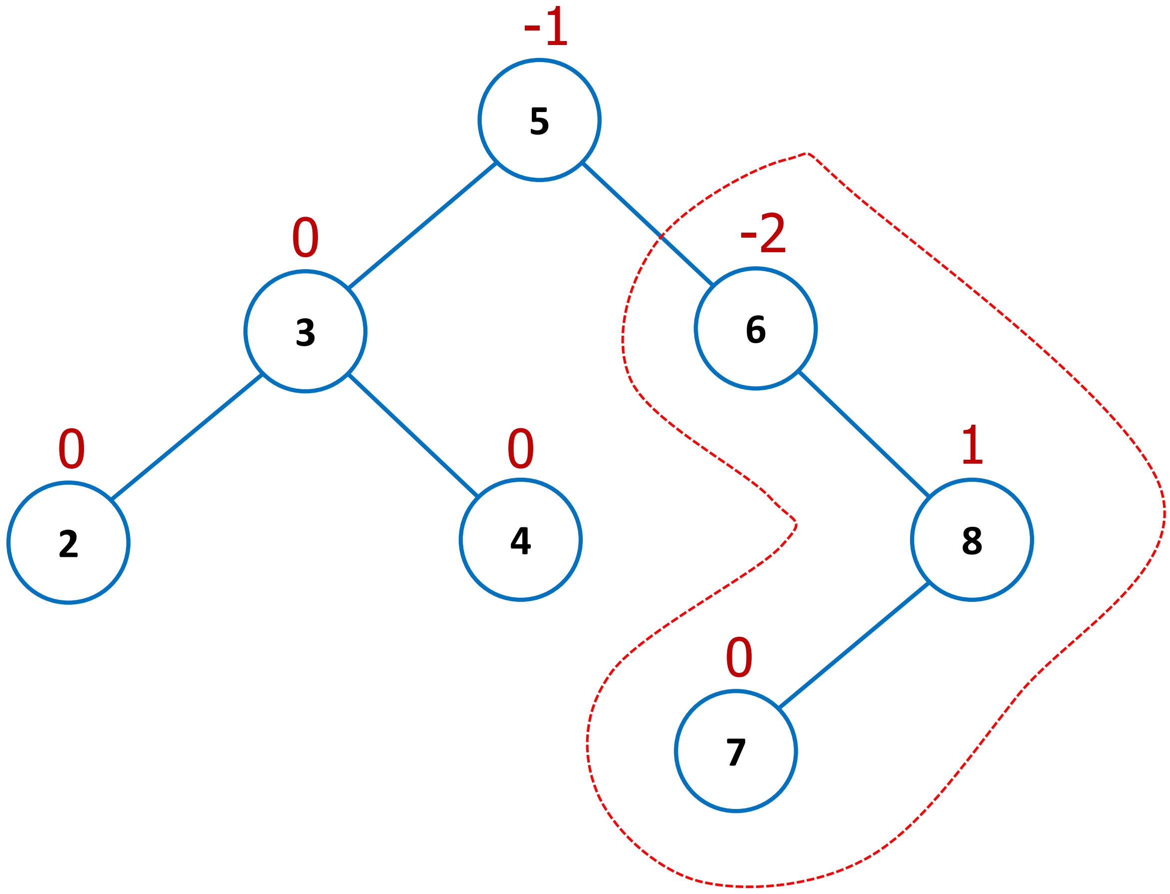
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



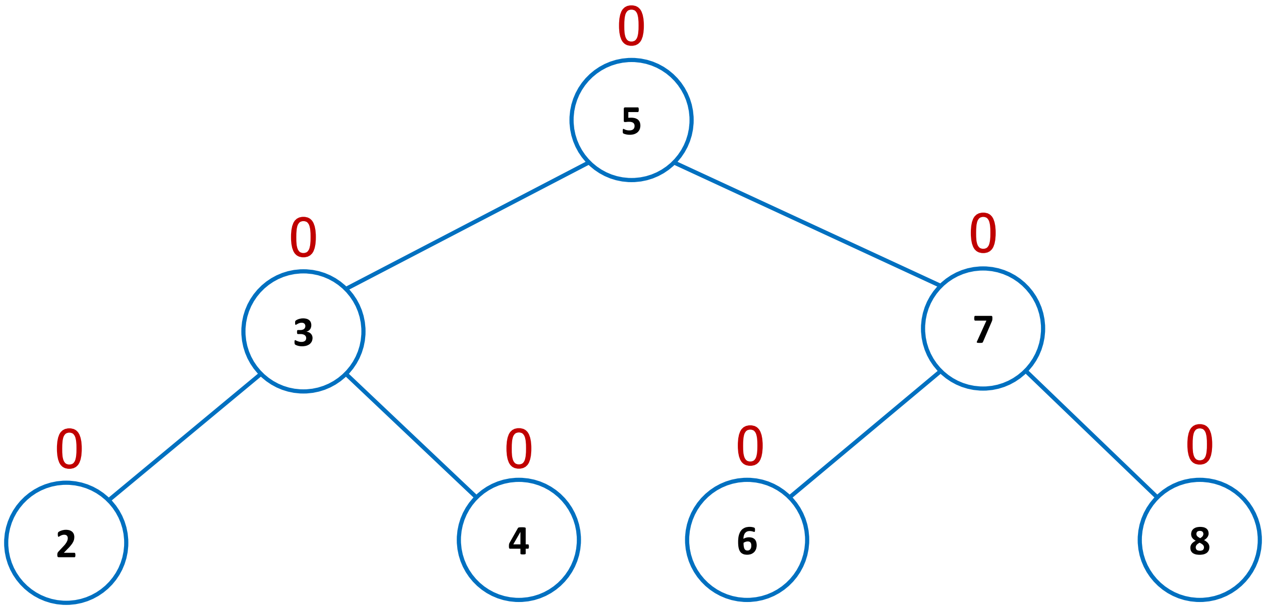
AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



AVL Trees

Example: Construct an AVL tree for the list $5, 6, 8, 3, 2, 4, 7$ by successive insertions



(2,4) Trees

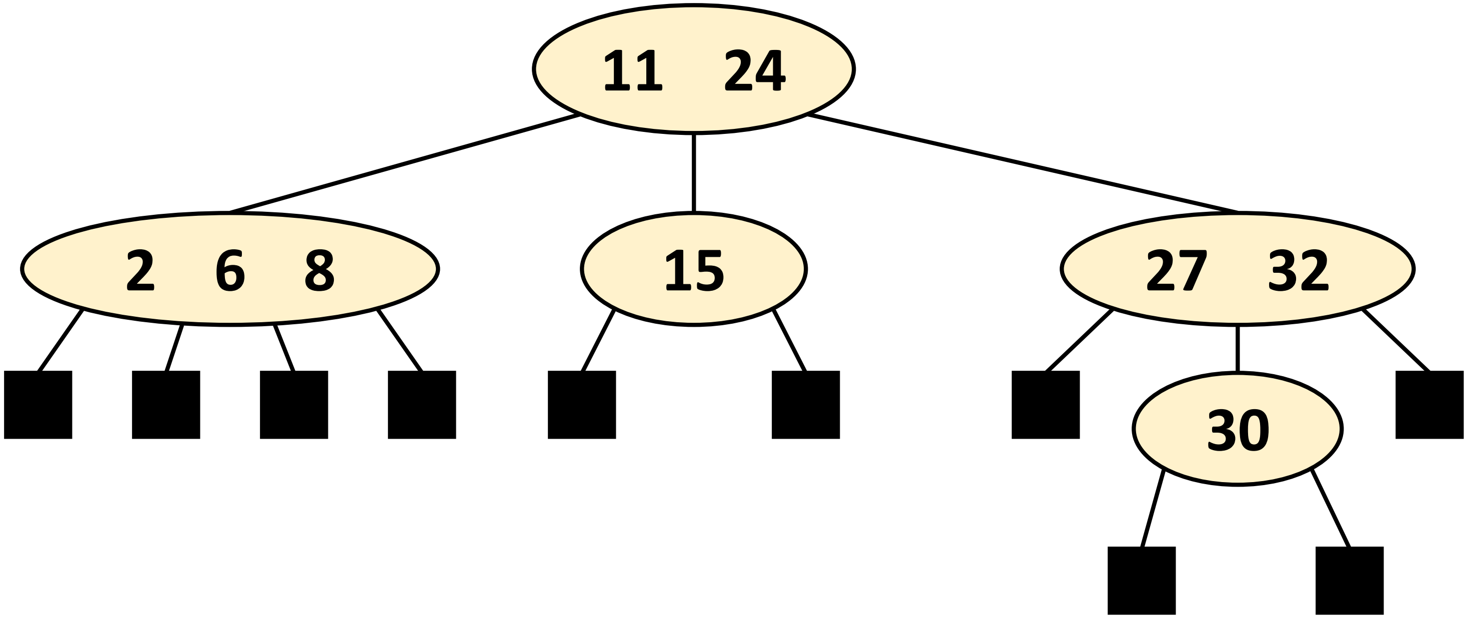
Multi-Way Search Tree

Definition: A **multi-way search tree (MST)** is an ordered tree such that

- each **internal (d -) node** has **at least two children** and stores $d-1$ **keys**: $k_1 \leq k_2 \leq \cdots \leq k_{d-1}$ where d is the **number of children**
- for a node with children $v_1, v_2, \ldots, v_d$ storing keys $k_1, k_2, \ldots, k_{d-1}$
  - keys in the subtree of v_1 are less than k_1
  - keys in the subtree of v_i are between k_{i-1} and k_i
  - keys in the subtree of v_d are greater than k_{d-1}
- the **leaves** store no items and serve as **placeholders**

Multi-Way Search Tree

Example:

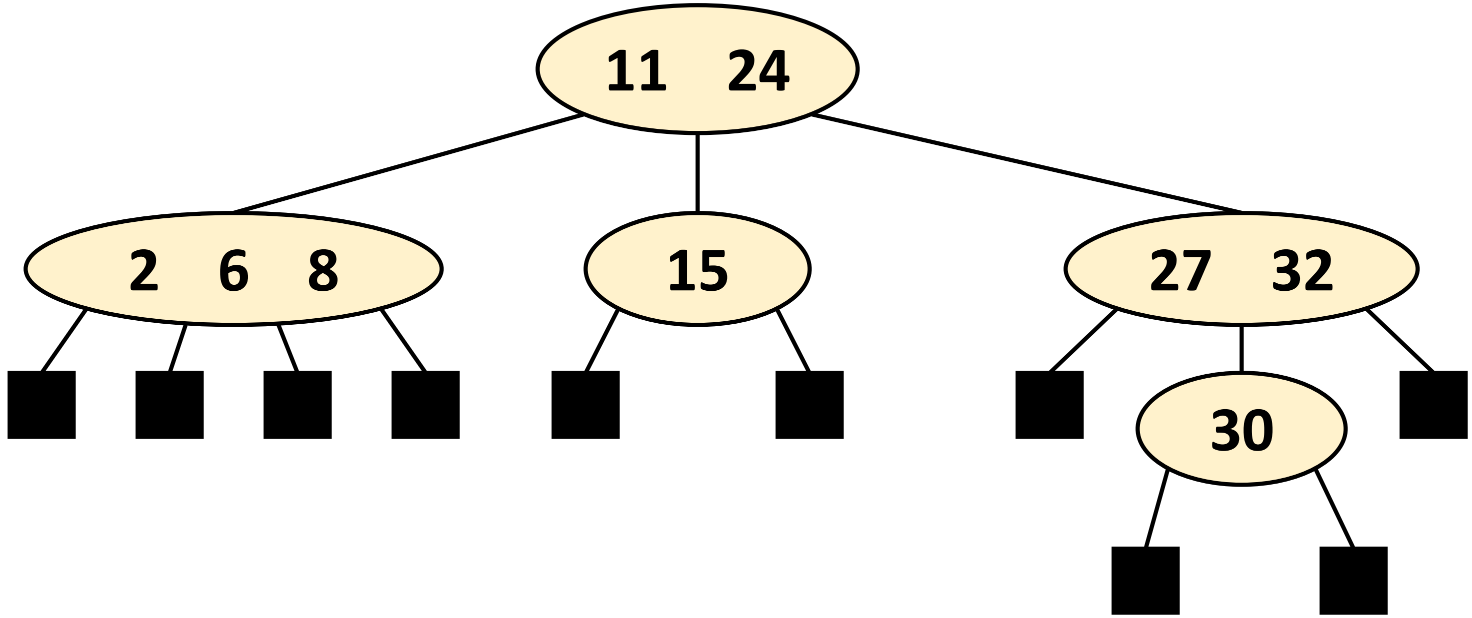


Multi-Way Searching

- At each internal node with children $v_1, v_2, \ldots, v_d$ and keys $k_1, k_2, \ldots, k_{d-1}$ storing keys
  - $k = k_i$ $(i = 1, 2, \ldots, d-1)$: the search terminates **successfully**
  - $k < k_1$: continue the search in child v_1
  - $k_i < k < k_{i+1}$: continue the search in child v_i
  - $k > k_{d-1}$: continue the search in child v_d
- Reaching an **external node** terminates the search **unsuccessfully**

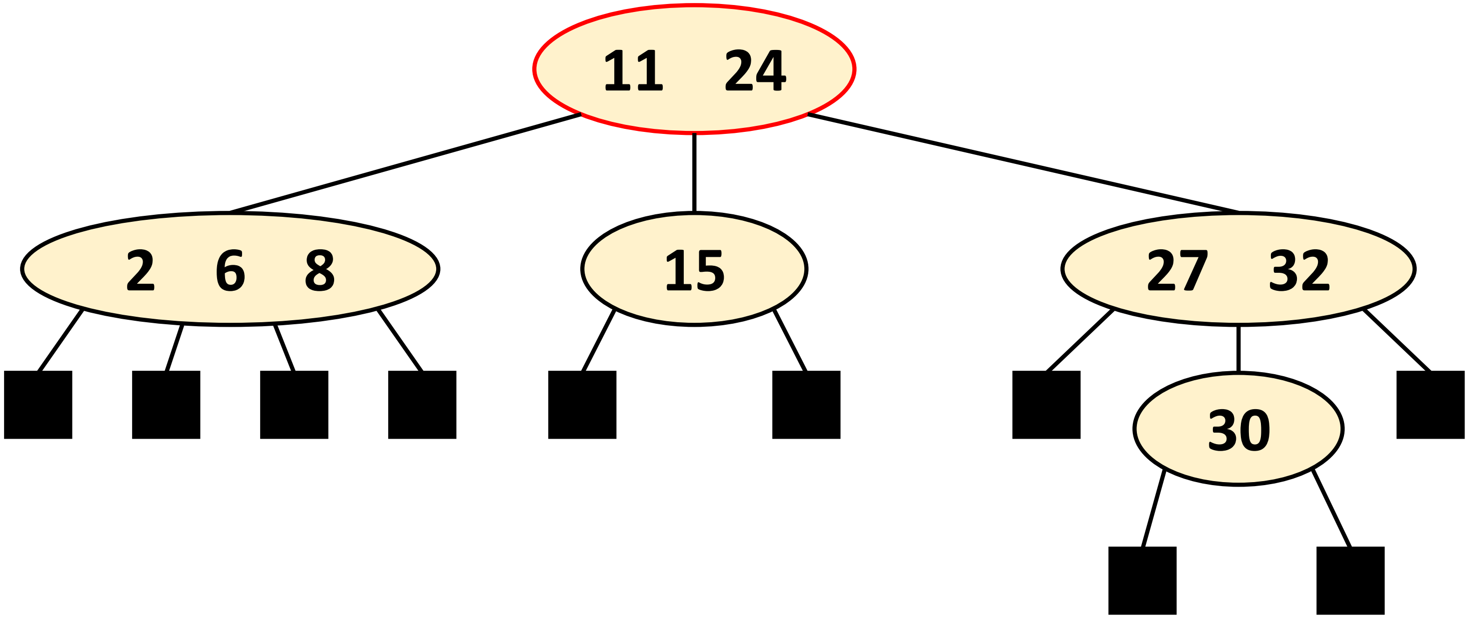
Multi-Way Searching

Example: $\mathrm{k} = 30$



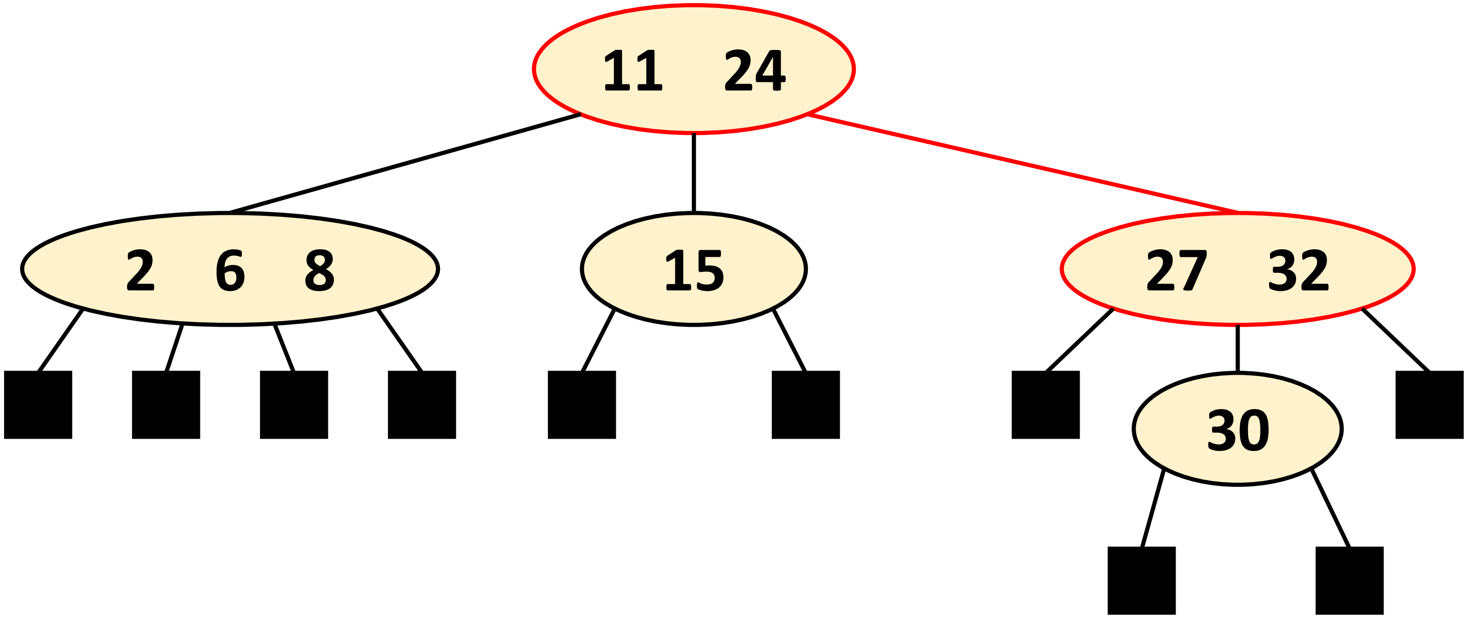
Multi-Way Searching

Example: $k = 30$



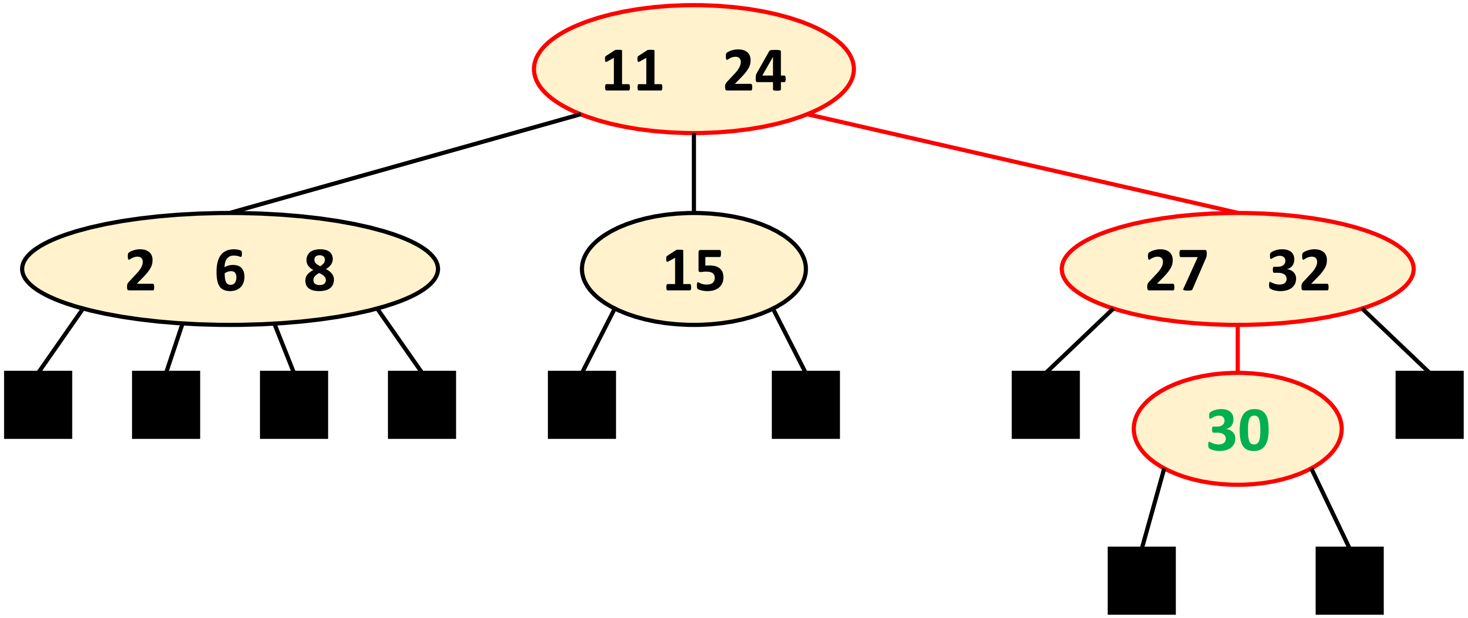
Multi-Way Searching

Example: $\mathrm{k} = 30$



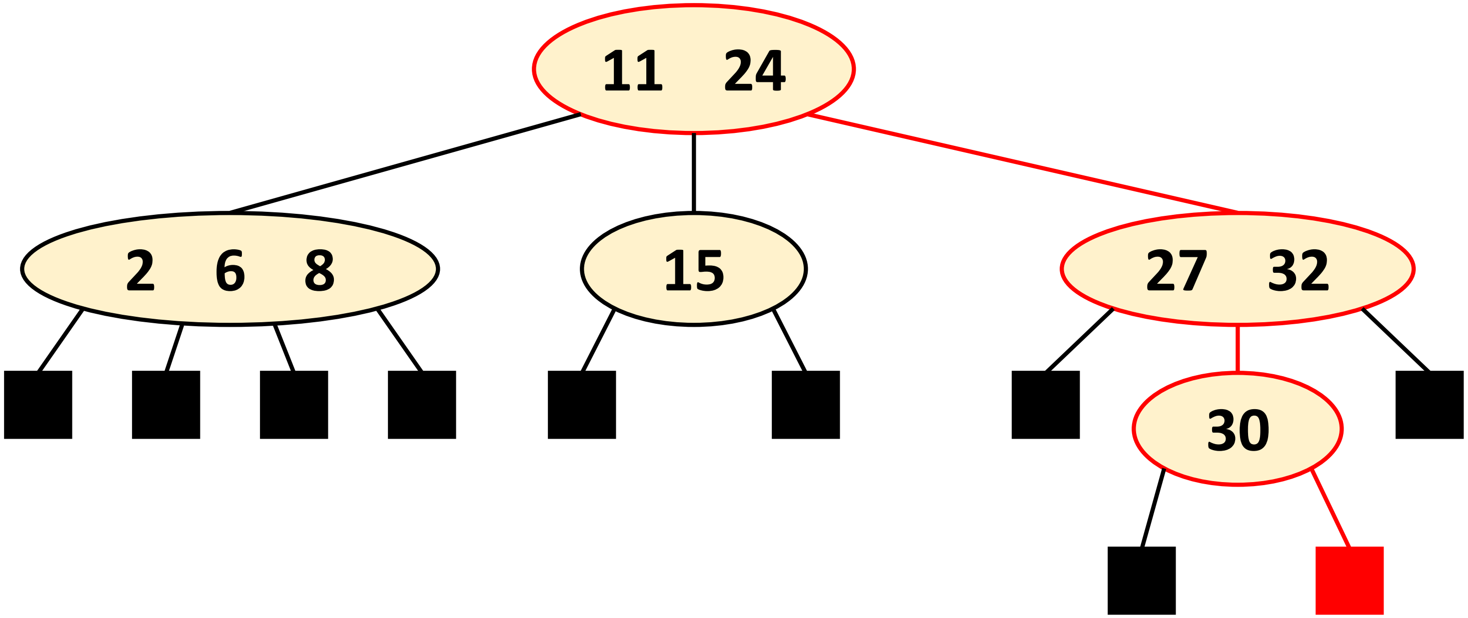
Multi-Way Searching

Example: $k = 30$



Multi-Way Searching

Example: $k = 31$



(2,4) Trees

Definition: A **(2,4) tree** (also called **2-4 tree** or **2-3-4 tree**) is a multi-way search tree with the following properties:

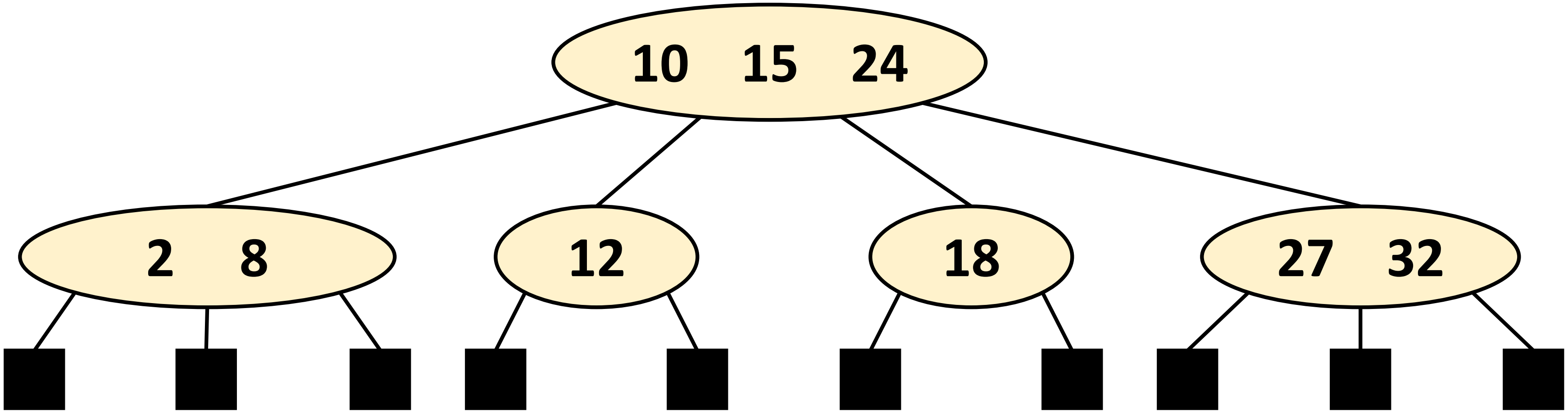
- **Node-Size Property:**
  every internal node has at least two children and at most four children

- **Depth Property:**
  all the external nodes have the same depth

- Depending on the number of children, an internal node of a (2,4) tree is called a **2-node**, **3-node** or **4-node**

(2,4) Trees

Example:



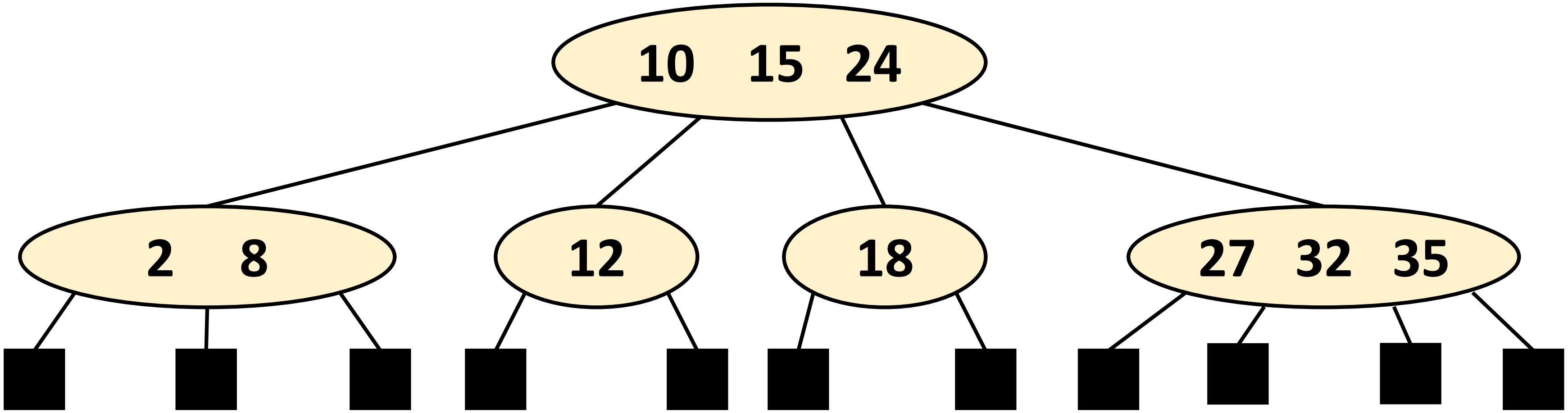
Insertion

We insert a **new key** k at the **parent** v of the **leaf** (external node) reached by searching for k

- we preserve the **depth property** but
- we may cause an **overflow**, i.e., node v may become a **5-node**

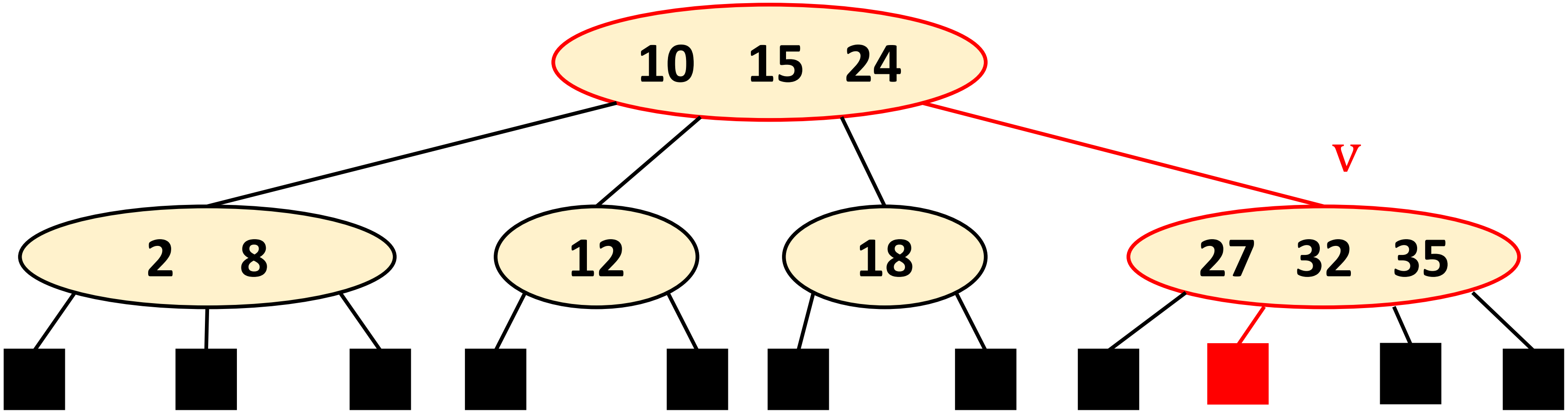
Insertion

Example: $k = 30$



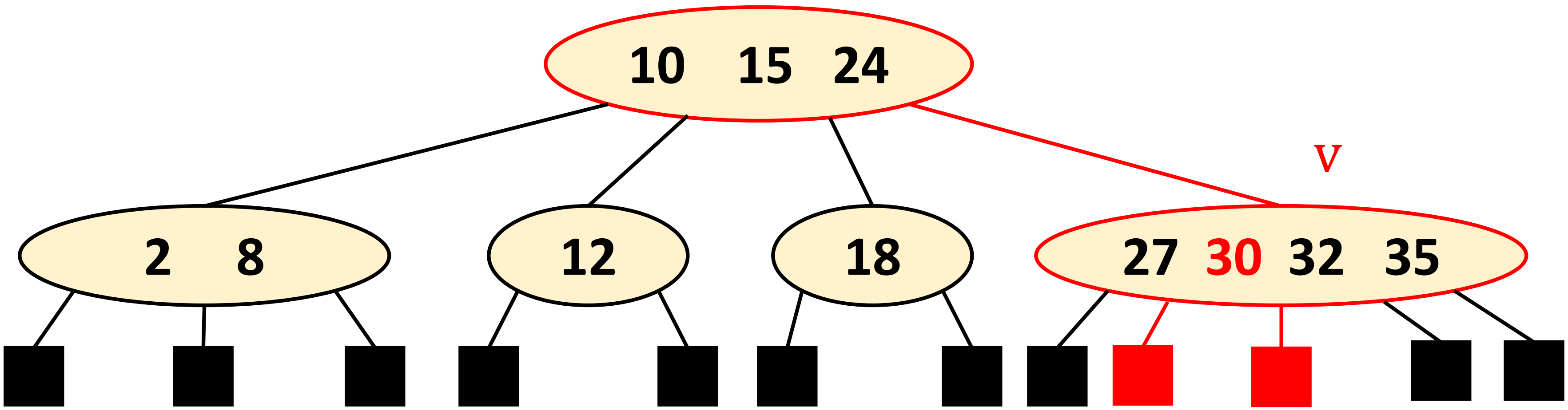
Insertion

Example: $k = 30$



Insertion

Example: $k = 30$



Overflow & Split

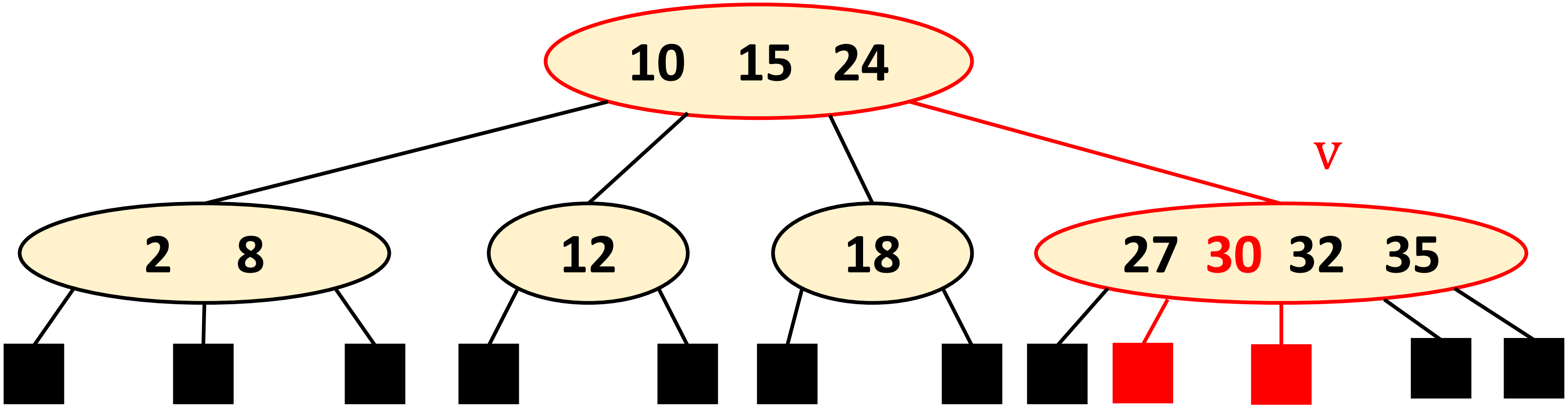
We handle an **overflow** at a **5-node** v with a **split operation**:

- let $v_1, \dots, v_5$ be the children of v and $k_1, \dots, k_4$ be the keys of v
- node v is replaced nodes v' and v''
- v' is a **3-node** with keys k_1, k_2 and children v_1, v_2, v_3
- v'' is a **2-node** with key k_4 and children v_4, v_5
- key k_3 is inserted into the **parent** u of v (a new root may be created)

- The **overflow** may propagate to the parent node u

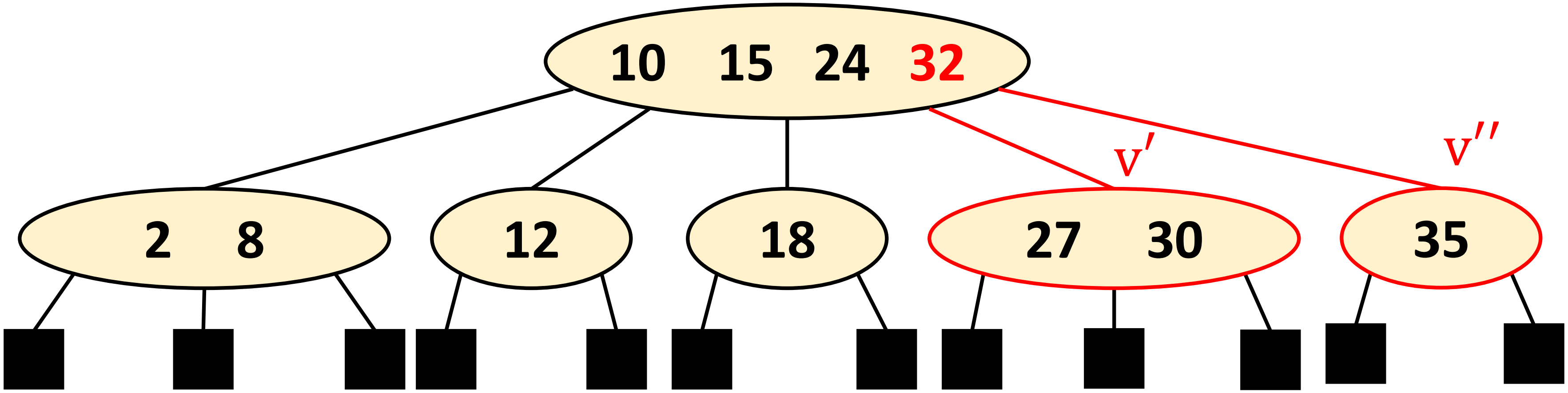
Insertion

Example: $k = 30$



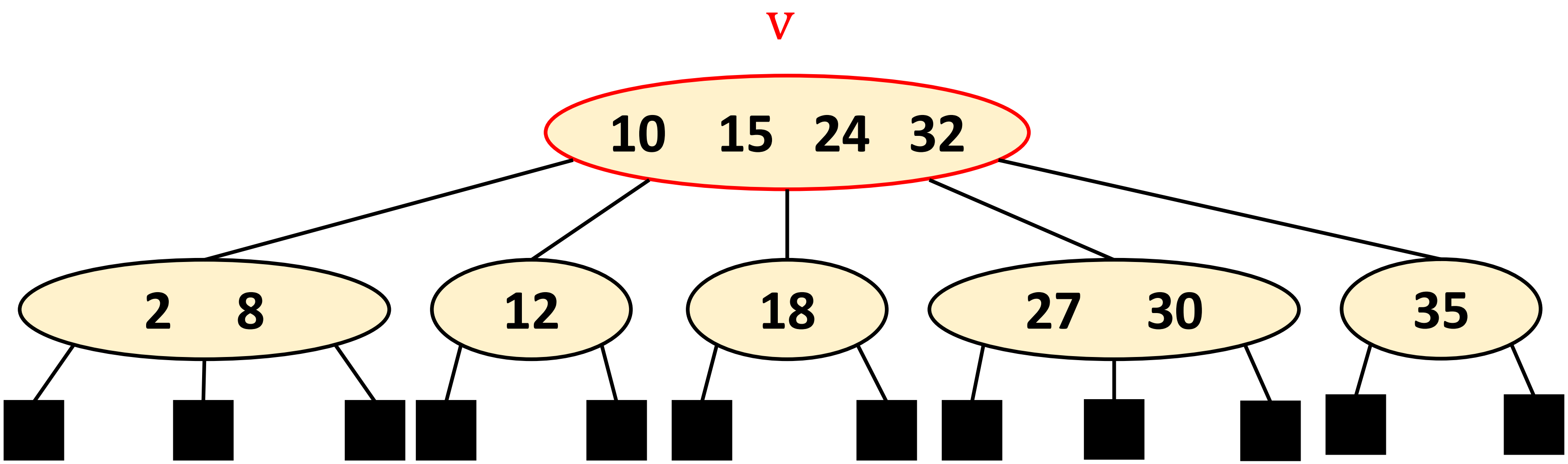
Insertion

Example: $k = 30$



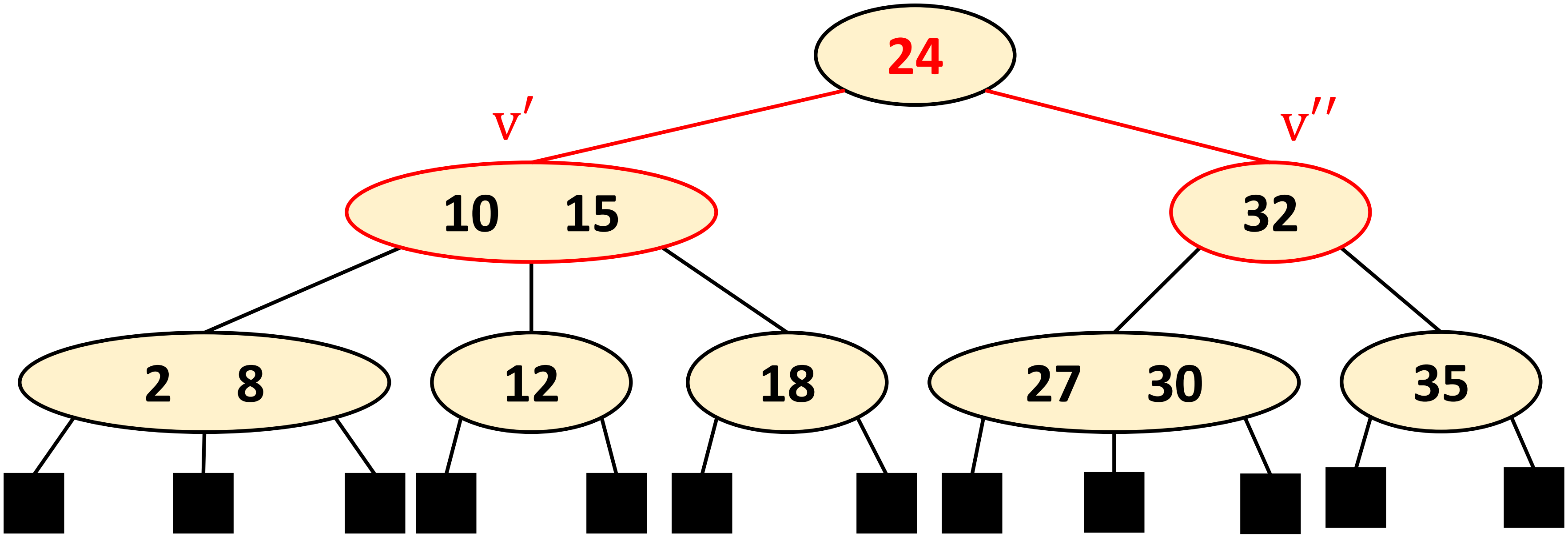
Insertion

Example: $k = 30$



Insertion

Example: $k = 30$



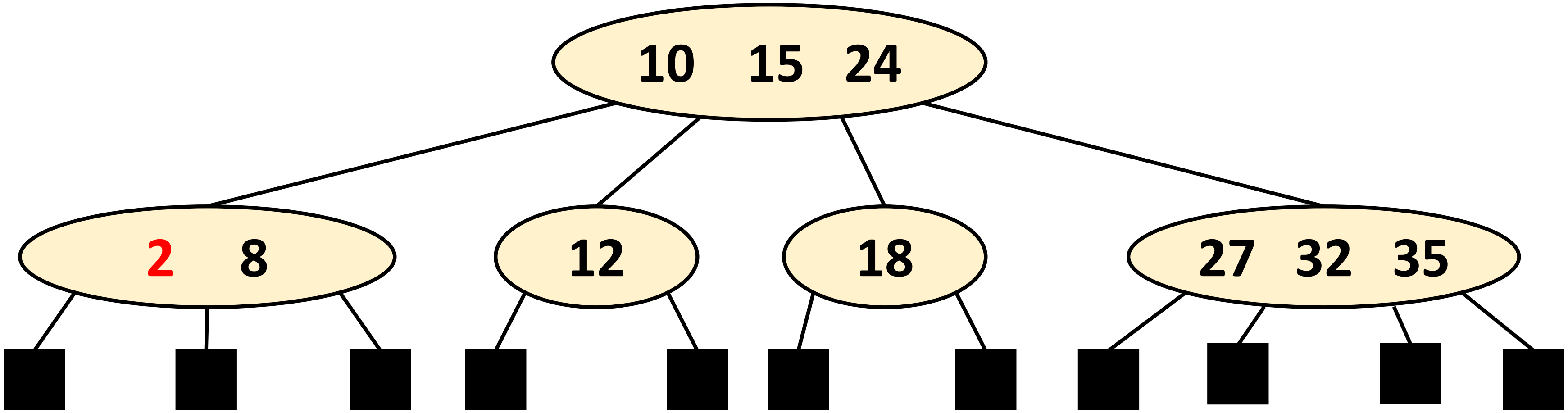
Deletion

1. **Reduce** deletion of an **entry** to the case where the key is **at the node with leaf children**

2. Otherwise, **replace** the entry with its **inorder successor** (or, equivalently, with its **inorder predecessor**) and **delete** the **latter entry**

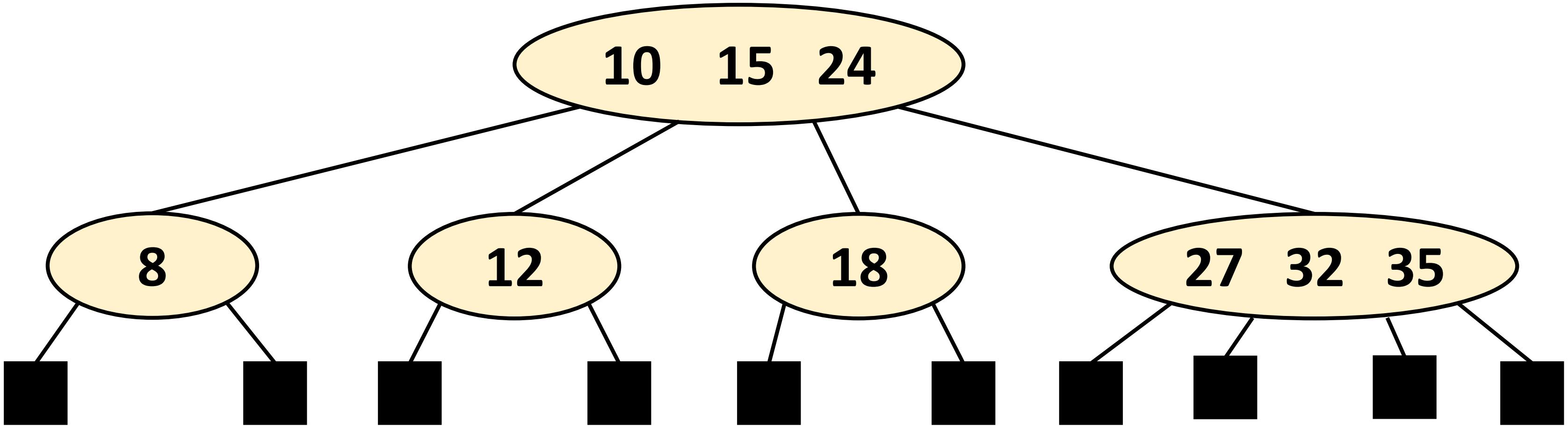
Deletion

Example: delete $k = 2$



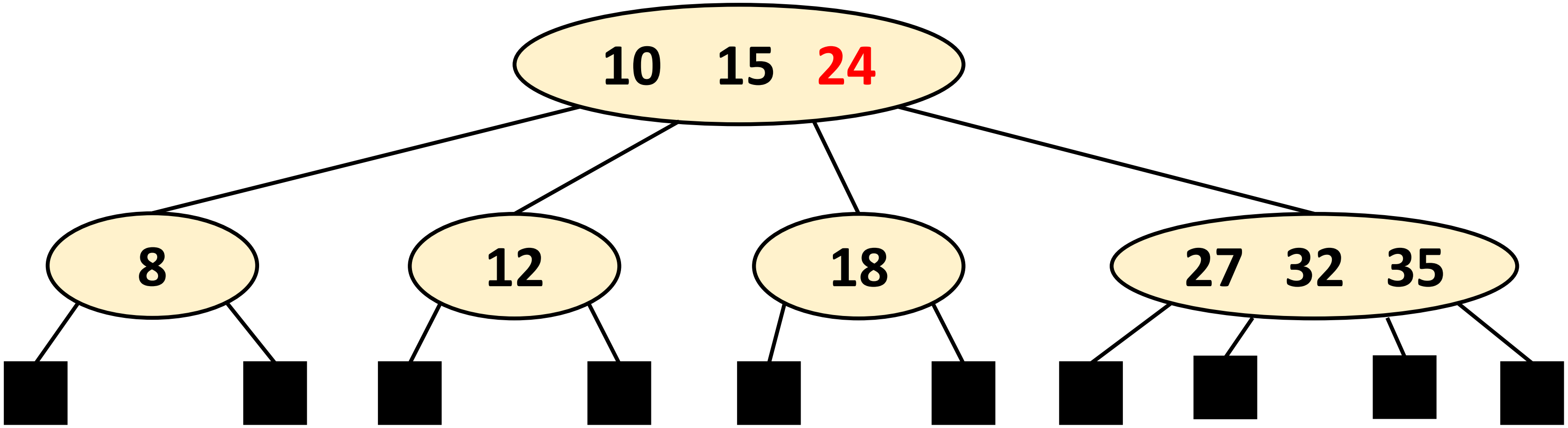
Deletion

Example: delete $k = 2$



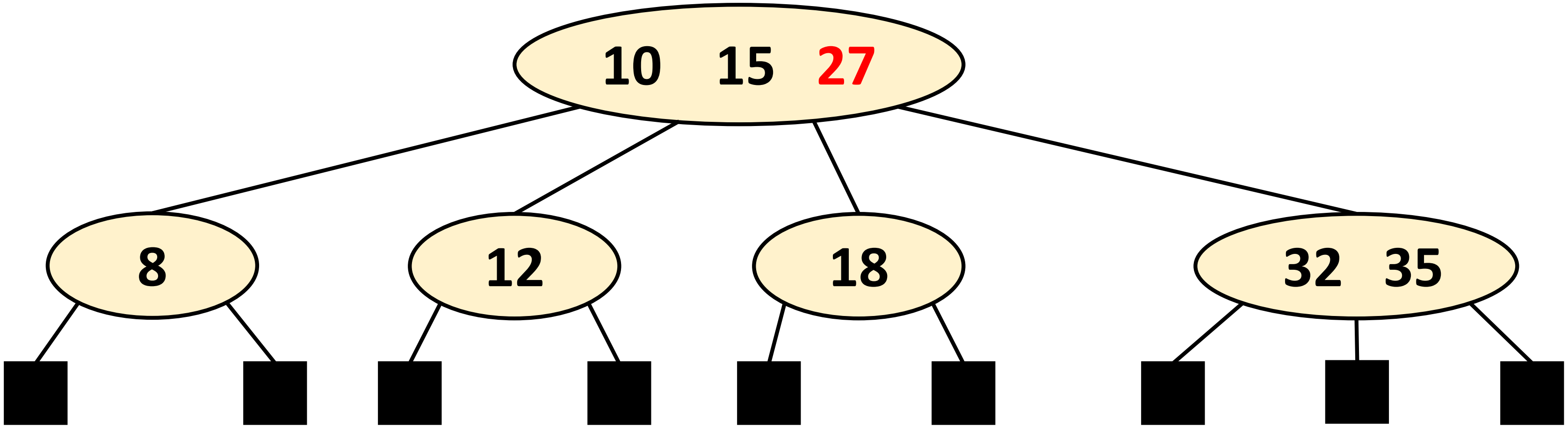
Deletion

Example: delete $k = 24$



Deletion

Example: delete $k = 24$



Underflow & Fusion

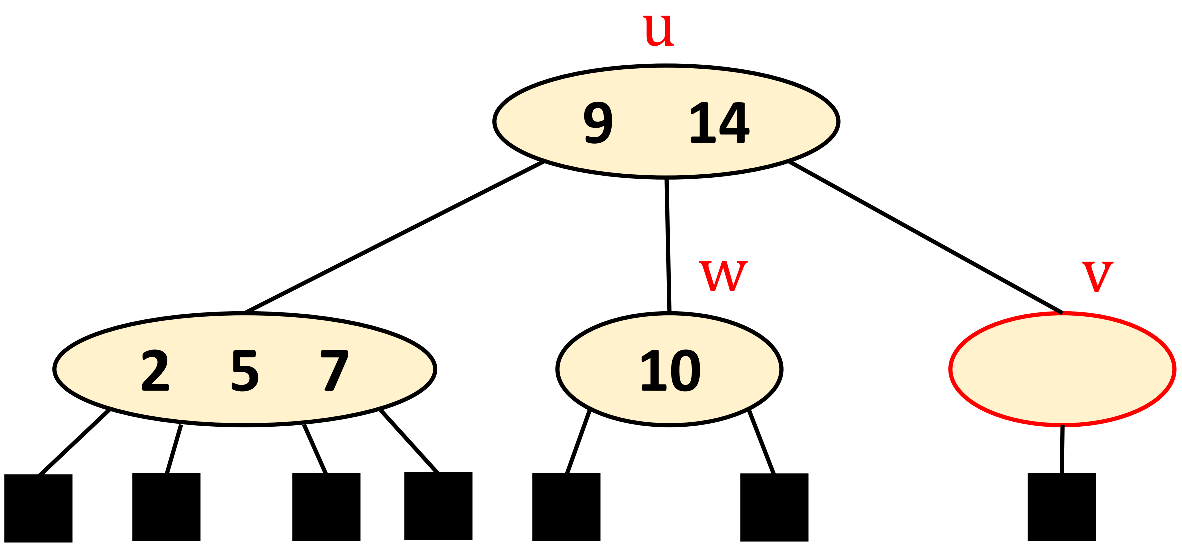
- Deleting an entry from a node v may cause an **underflow**, where node v becomes a **1-node** with one child and **no keys**
- To handle an **underflow** at node v with parent u, we consider **two cases**:

Case 1: the adjacent siblings of v are **2-nodes**

- **Fusion operation**: we **merge** v with an **adjacent sibling** w and **move** an entry from u to the merged node v'
- After a **fusion**, the **underflow** may propagate to the **parent** u

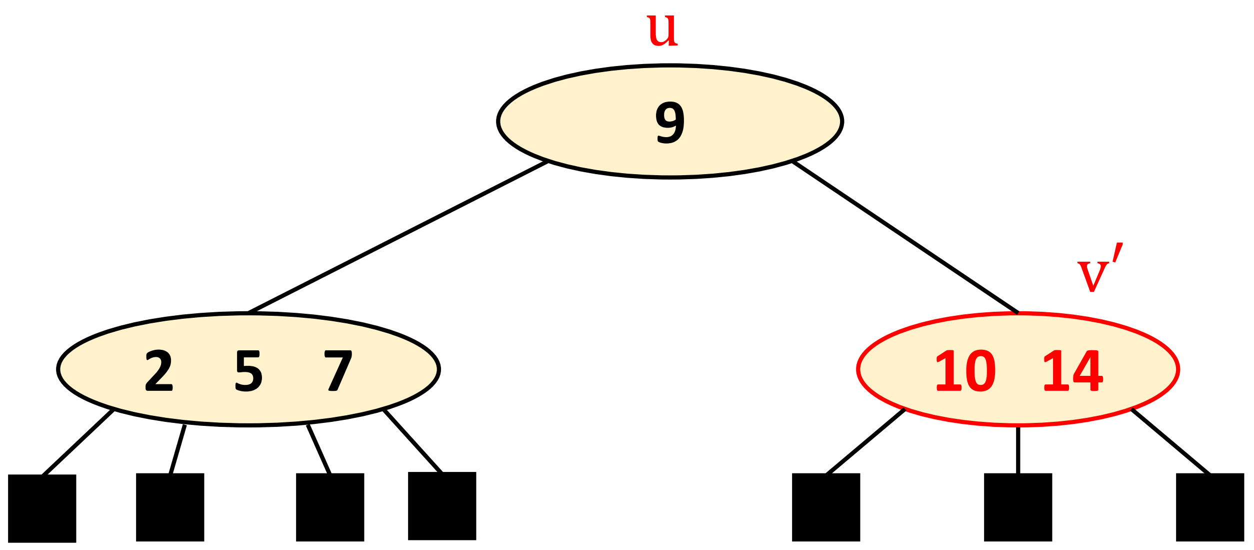
Underflow & Fusion

Example:



Underflow & Fusion

Example:



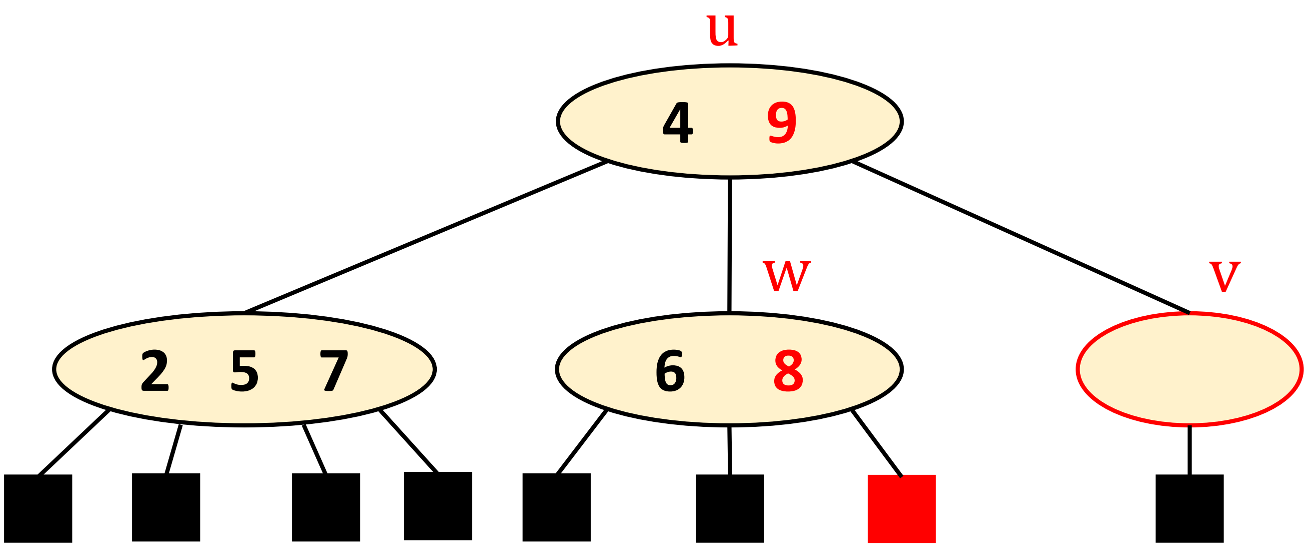
Underflow & Transfer

Case 2: the adjacent sibling w of v is a **3-node** or **4-node**

- **Transfer operation**:
  - move a **child** of w to v
  - move a **key** from u to v
  - move a **key** from w to u
- After a **transfer**, no **underflow** occurs

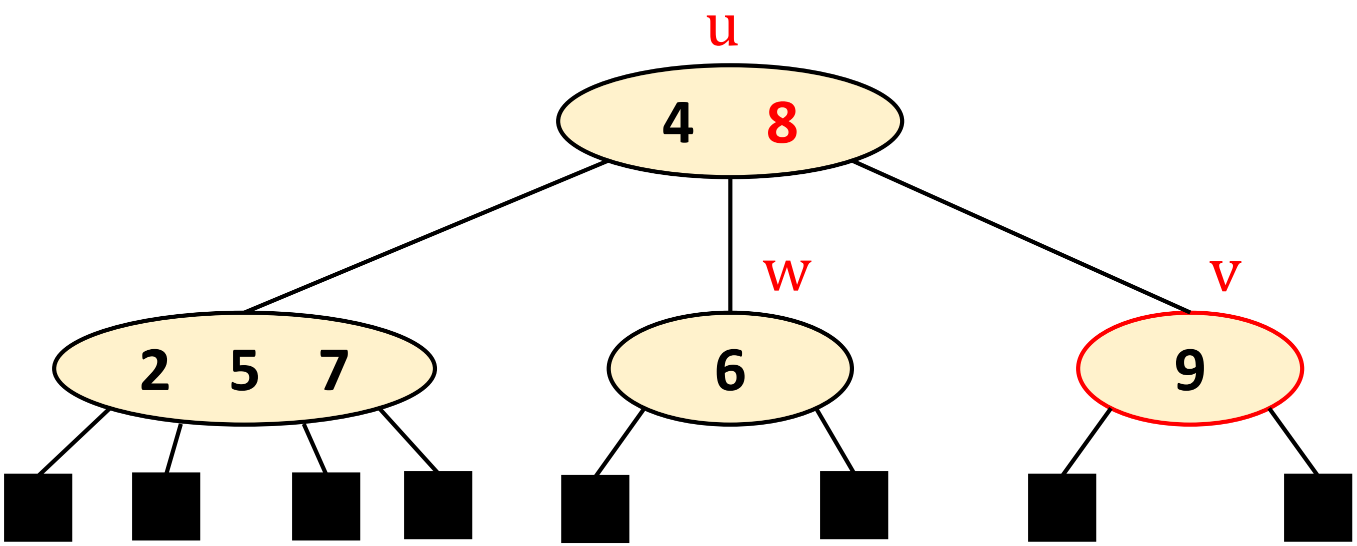
Underflow & Fusion

Example:



Underflow & Fusion

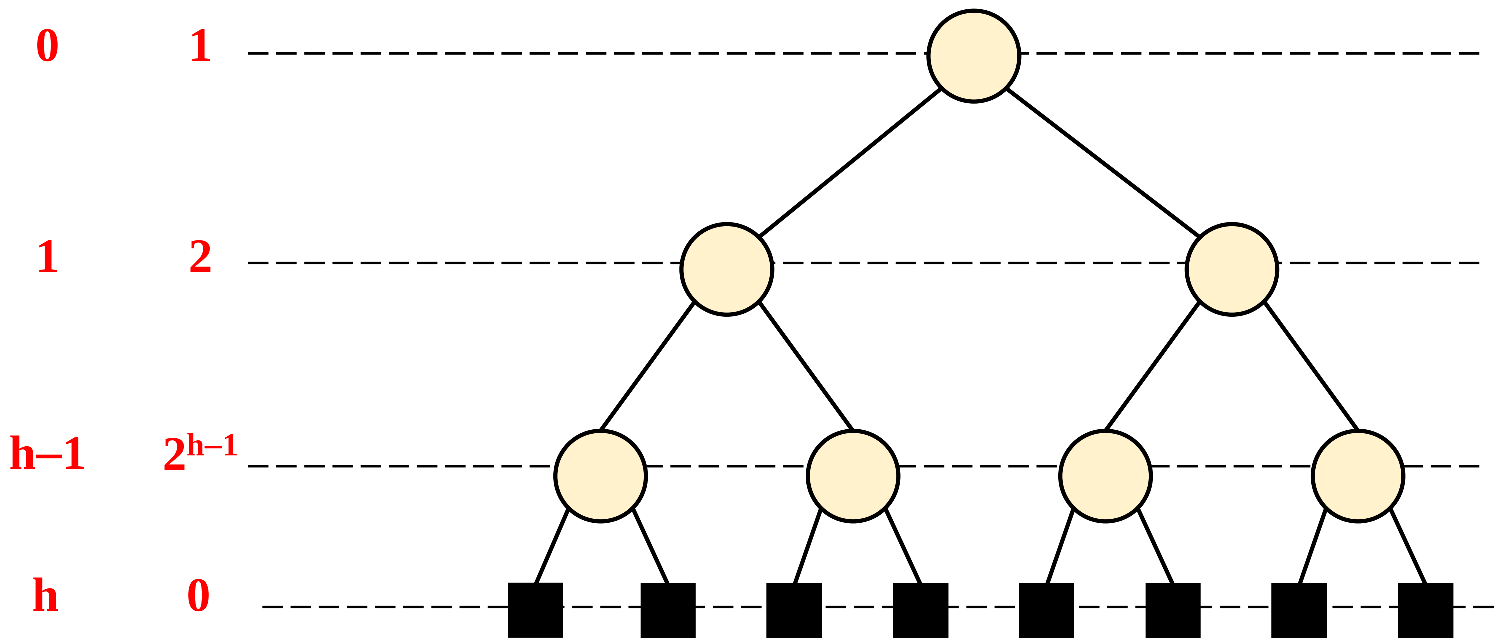
Example:



Height of (2,4) Trees

Theorem: A (2,4) tree storing n keys has height $O(\log n)$.

Idea:



Height of (2,4) Trees

Theorem: A (2,4) tree storing n keys has height $O(\log n)$.

Proof:

- Let h be the **height** of a (2,4) tree with n keys
- Since there are at least 2^i **keys** at depth $i = 0, \dots, h-1$ and **no keys** at depth h:

$$n \geq 1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$$

- Thus, $h \leq \log(n+1)$

Corollary: Searching in a (2,4) tree with n keys takes $O(\log n)$ time.

Analysis of Deletion

- Let T be a (2,4) tree with n keys
  - tree T has $O(\log n)$ **height**
- In a **deletion operation**:
  - we visit $O(\log n)$ **nodes** to **locate** the node from which to delete the entry
  - we handle an **underflow** with a series of $O(\log n)$ **fusions**, followed by **at most one transfer**
  - **each fusion** and **transfer** takes $O(1)$ time
- Thus, **deleting an item** from a (2,4) tree takes $O(\log n)$ **time**

Exercise

- Construct a 2-4 tree from a list $\mathbf{2, 13, 7, 16, 19, 9, 22, 10, 14, 17}$.
- Then, delete **19** from the 2-4 tree that you got.