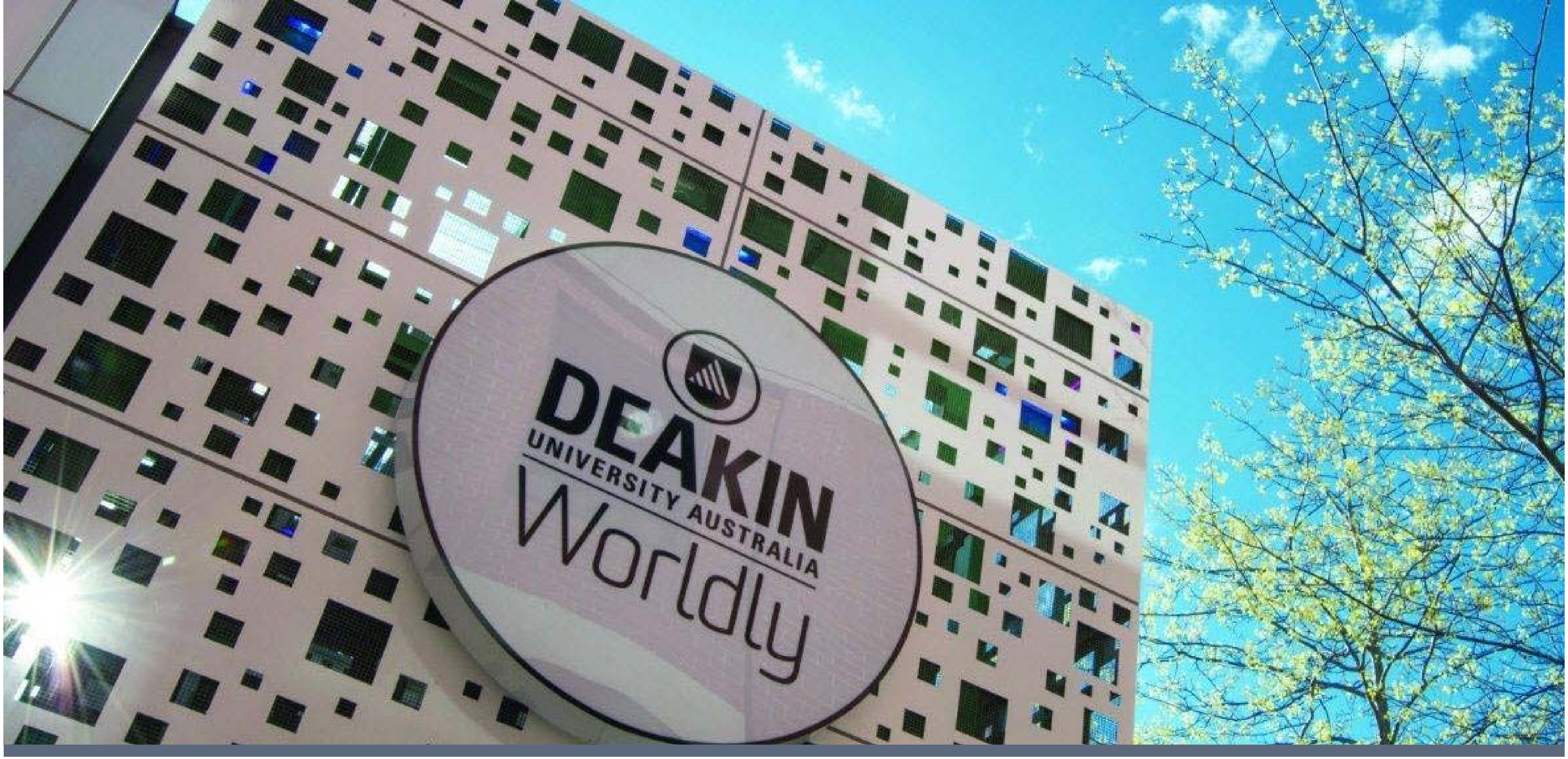


Lecture 7. Heap Sort. AVL-Trees.

Graphs and their Representation.

SIT221 Data Structures and Algorithms

Sorting with Priority Queues

- We use a Priority Queue
  - Insert the elements with a series of Insert operations.
  - Remove the elements in sorted order with a series of DeleteMin operations.
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives the Selection Sort: $O(n^2)$ time
  - Sorted sequence gives Insertion Sort: $O(n^2)$ time
- Can we do better?

Sorting with Priority Queues

Algorithm *PriorityQueueSort(S, C)*

Input: sequence S, comparator C for the elements of S

Output: sequence S sorted in increasing order according to C

// Build priority queue P applying comparator C

while (**not** S.isEmpty()) **do**

Element e = S.First();

P.Insert(e);

S.Remove(e);

// Build back the (sorted) sequence S

while (**not** P.isEmpty()) **do**

Element e = P.DeleteMin();

S.AddLast(e);

Sorting with a Minimum Binary Heap: Heap Sort

Want to have a sorting algorithm based on heaps that runs in time $O(n\log n)$.

Idea:

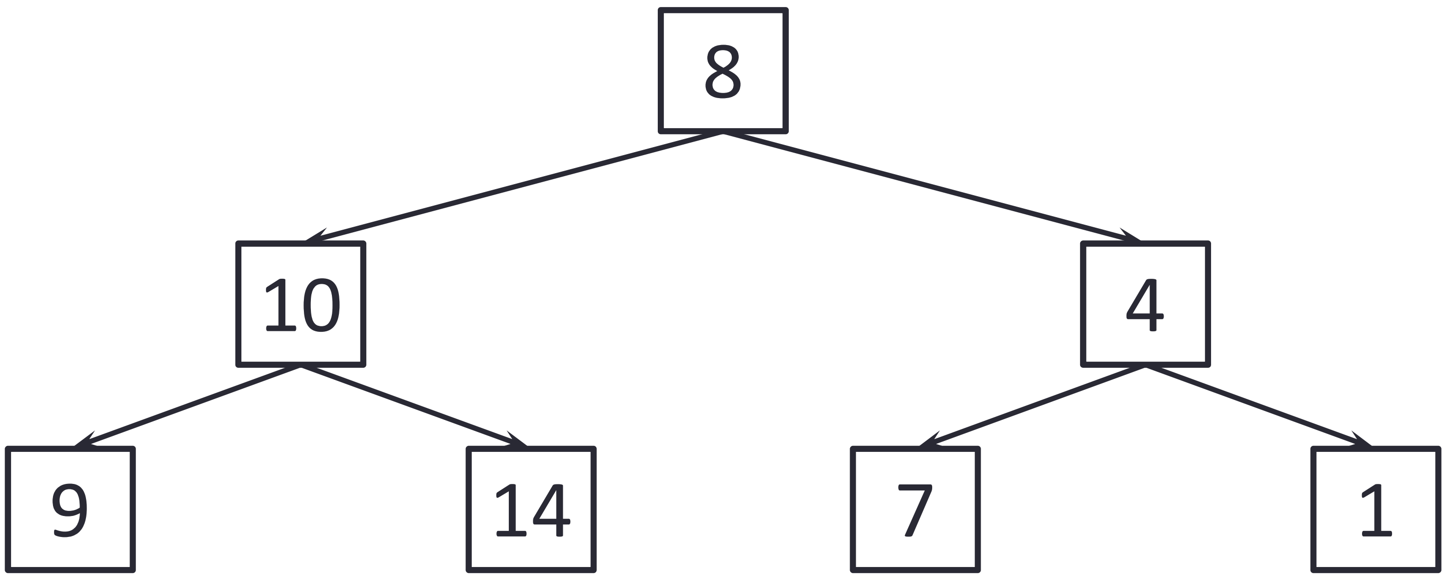
- Build (the bottom-up strategy) the heap for n elements in time $O(n)$.
- Pick in each step the minimum element and delete it in time $O(\log n)$.
- Iterate until heap is empty.
- The space used is $O(n)$.

In total, n iterations implies the total runtime of $O(n\log n)$

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 1. Bottom-up heap construction

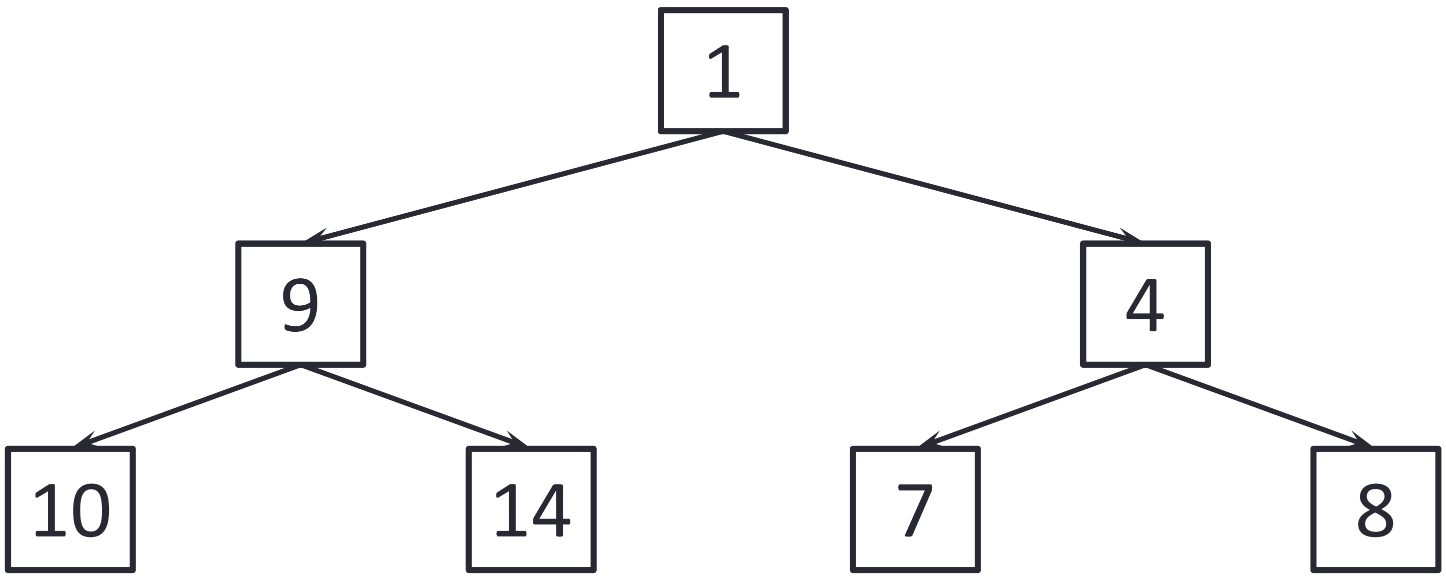


value:	8	10	4	9	14	7	1
indices:	1	2	3	4	5	6	7

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 1. Bottom-up heap construction

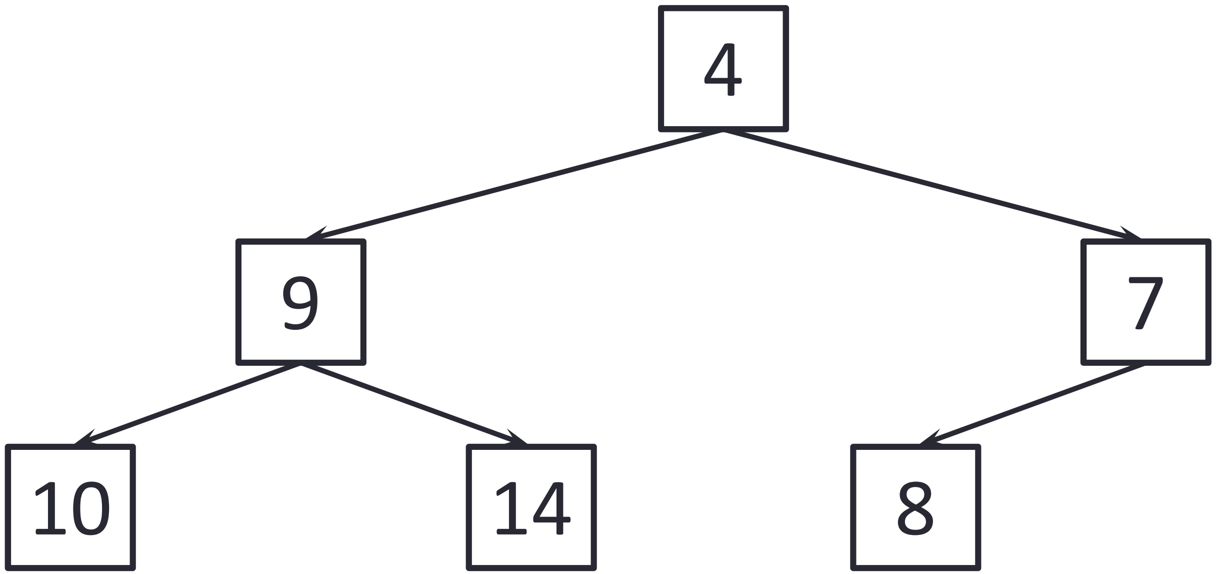


value:	8	10	4	9	14	7	1
indices:	1	2	3	4	5	6	7

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 1



value:

4	9	7	10	14	8	

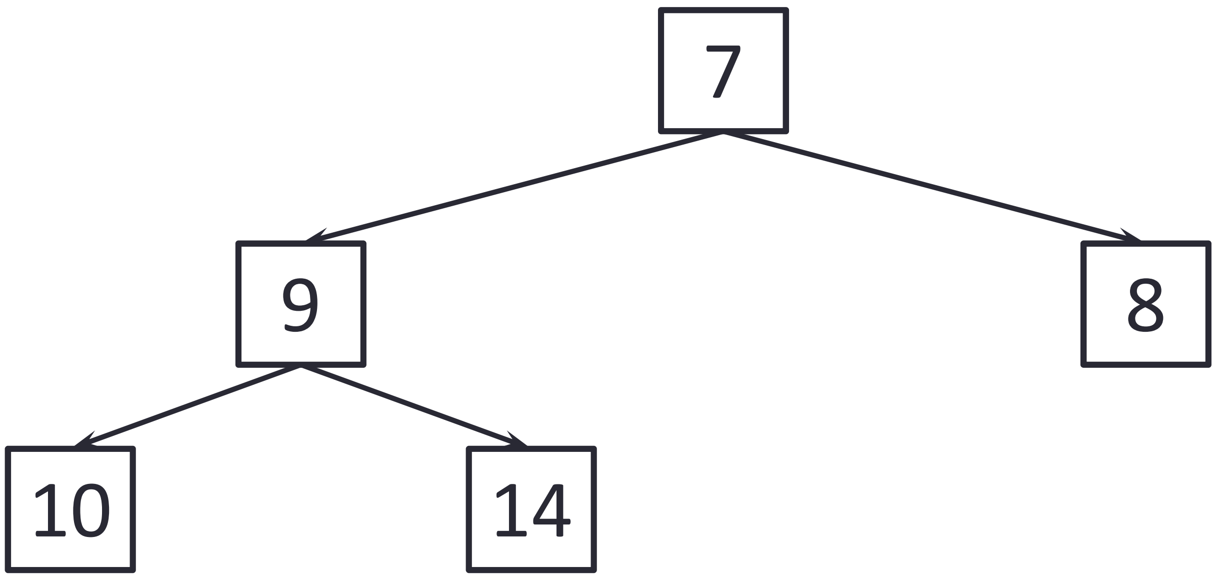
Sorted array:

1						

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 4

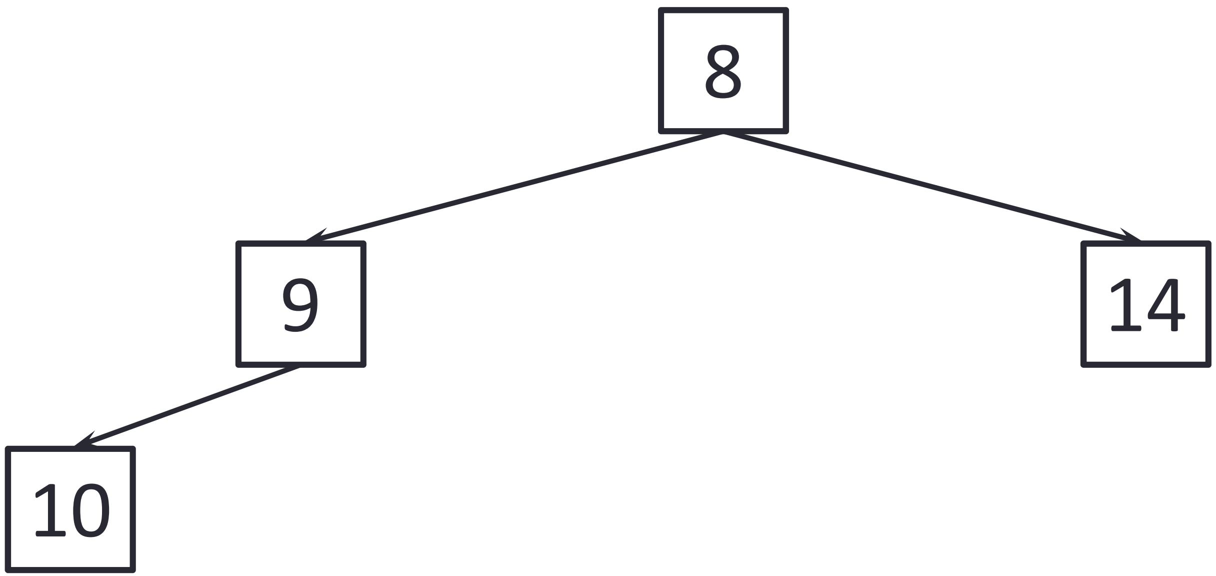


value:	7	9	8	10	14		
Sorted array:	1	4					

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 7

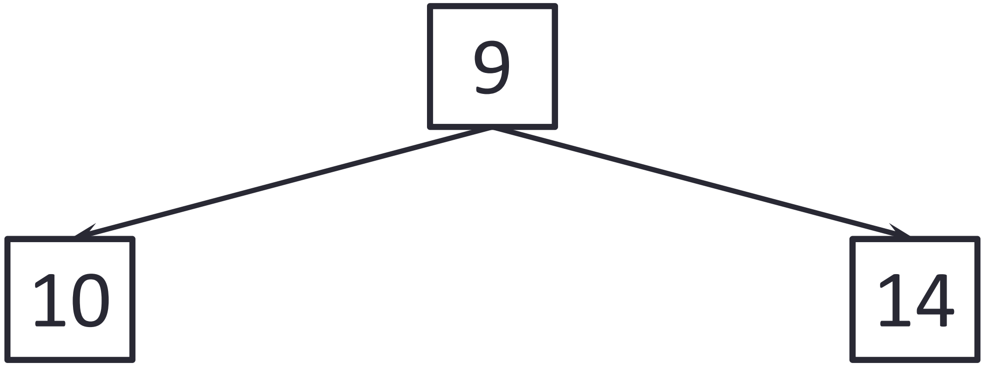


value:	8	9	14	10			
Sorted array:	1	4	7				

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 8

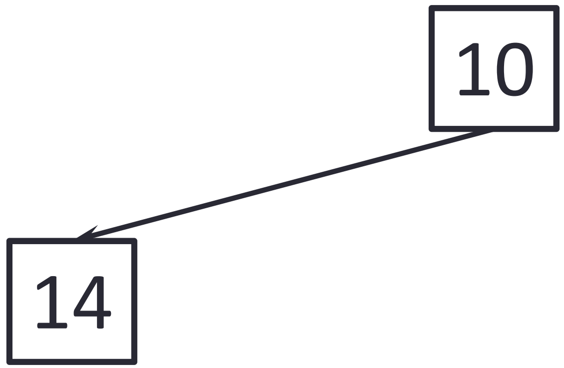


value:	9	10	14				
Sorted array:	1	4	7	8			

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 9



value:	10	14					
Sorted array:	1	4	7	8	9		

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 10

14

value:	14						
Sorted array:	1	4	7	8	9	10	

Heap Sort: Example

Sort the sequence [8,10,4,9,14,7,1]

Step 2. Iterative Deletion: Delete 14

value:							
Sorted array:	1	4	7	8	9	10	14

Heap Sort: Properties and Complexity

- Heapsort is in-place, but is **not** a stable sort.
- Requires only a constant amount of auxiliary space, i.e. less than the Merge Sort needs.
- Slower in practice on most machines than a well-implemented Quick Sort, it has the advantage of a more favourable worst-case $O(n\log n)$ runtime.

- Worst case: $\mathrm{T(n)} = O(n\log n)$ comparisons
- Best case: $\mathrm{T(n)} = O(n\log n)$ comparisons
- Average case: $\mathrm{T(n)} = O(n\log n)$ comparisons
- Worst-case space complexity $O(1)$ auxiliary

Short Summary of Sorting Algorithms

Algorithm	Time	Notes
Selection Sort	$O(n^2)$	slow in-place for small data sets (< 1K)
Insertion Sort	$O(n^2)$	slow in-place for small data sets (< 1K)
Heap Sort	$O(n \log n)$	fast in-place for large data sets (1K — 1M)
Merge Sort	$O(n \log n)$	fast sequential data access for huge data sets (> 1M)

Runtimes for Binary Search Tree

Find, insert, and remove:

- Worst case: $\theta(n)$
- Best case: $\theta(\log n)$ and $\theta(1)$ for Find(k)
- Average case: $\theta(\log n)$

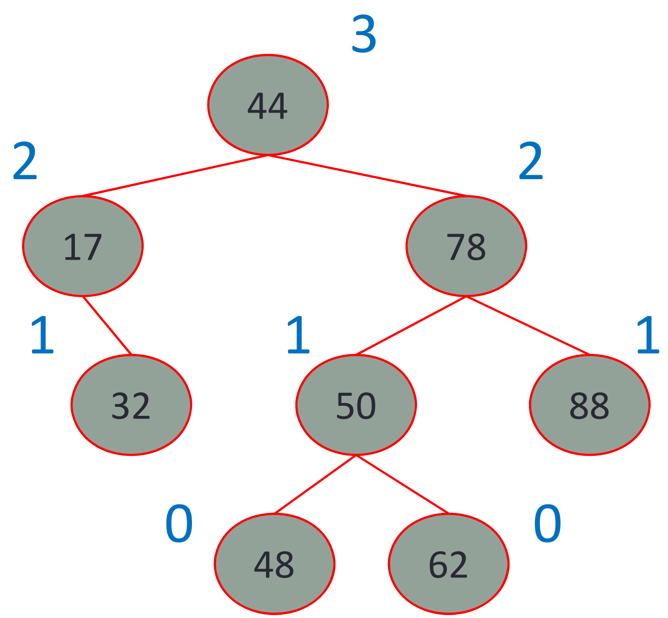
Observation: Binary search trees can get imbalanced when applying insert and/or remove operations.

Aim: Time $O(\log n)$ in the worst case for all operations

Idea: Whenever a subtree rooted at a node v gets imbalanced, apply operations that balance it out in time $O(\log n)$.

AVL-Tree: Definition

- An **AVL-Tree** is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1.
- AVL-trees are balanced.
- The height of an AVL-Tree storing n keys is $O(\log n)$.



An example of an AVL-tree where the heights are shown next to the nodes.

AVL-Tree: Formal Definition

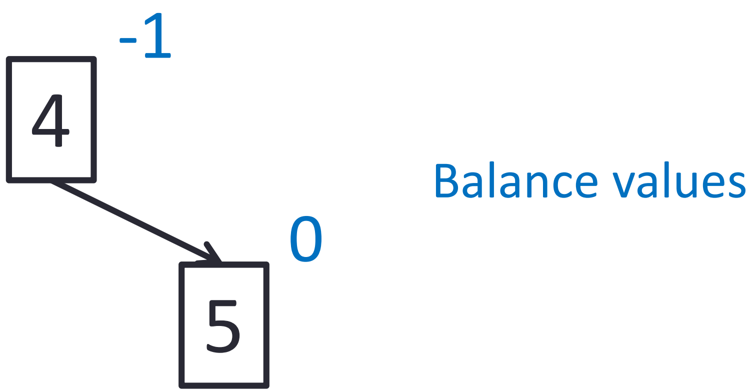
- Let $h(T)$ be the height of a tree T.
- Let v be a node in T, and T_l and T_r be the left and right subtree of v.
- We denote by $b(v) = h(T_l) - h(T_r)$ the balance degree of v.

Definition: A binary search tree T is called an AVL-tree if for each $v \in T$, $b(v) \in \{-1,0,1\}$ holds.

AVL-Tree: Insertion

- Insertion is as in a binary search tree.
- Always done by expanding an external node.

Example:

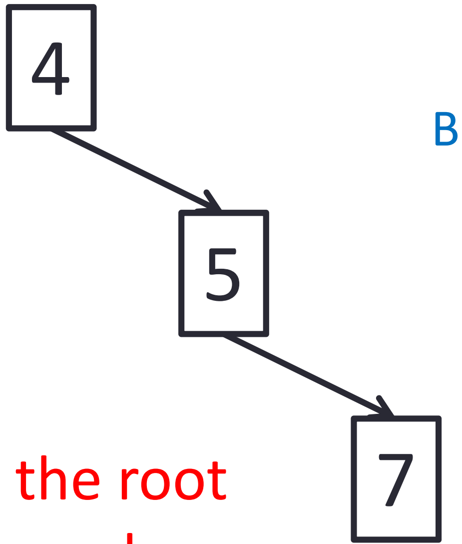


Insert 7

AVL-Tree: Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.

Example:



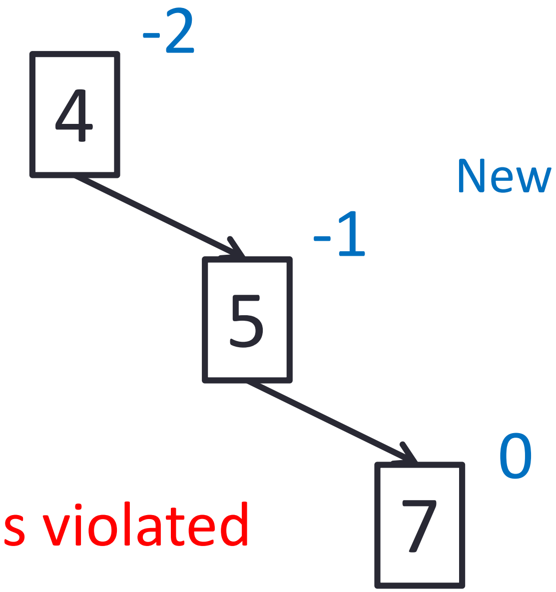
Balance values

Consider path from new leaf to the root
and check the balance values

AVL-Tree: Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.

Example:



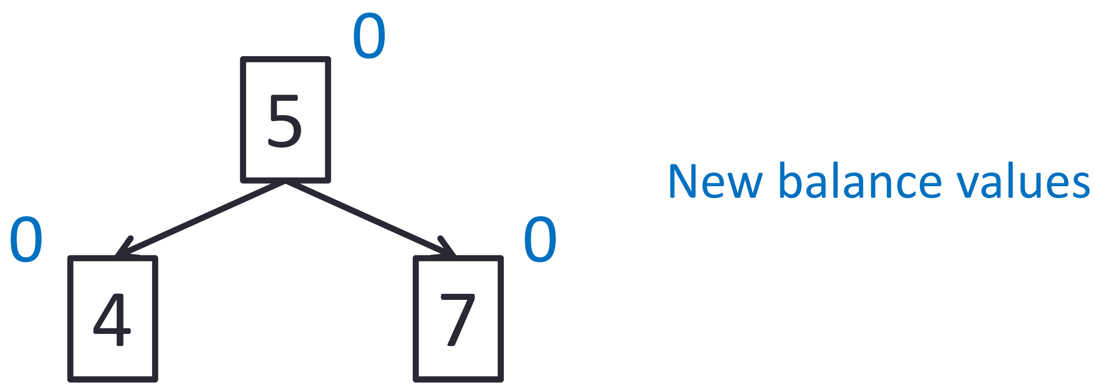
New balance values

AVL-property at node 4 is violated

AVL-Tree: Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.

Example:



Rotation establishes AVL-property again

AVL-Tree: Rebalancing after Insertion

- Inserting a new element z can violate the AVL-property.
- Consider path from the newly inserted leaf z to the root.
- Update the balance values.
- Repair AVL-property (if necessary).

AVL-Tree: Rebalancing after Insertion

- We insert new node z as for Binary Search Trees.
- $b(z) = 0$ holds after insertion.
- $b(v)$ might change by 1 for a node v on the path from z to the root.
- If $b(v) \notin \{-1,0,1\}$ then rebalance.

AVL-Tree: Rebalancing (Left Rotation)

Assume we have added z into the right subtree of node v.
Start examining for v, where v is the parent of z, and continue with the parent of v (if necessary).

Before insertion → After Insertion:

- $b(v) = 1 \ \rightarrow b(v) = 0$
  (height of tree rooted at v has not changed, stop rebalancing)

- $b(v) = 0 \ \rightarrow b(v) = -1$
  (height of tree rooted at v has increased by 1, stop rebalancing only if v is root, otherwise examine parent of v)

- $b(v) = -1 \ \rightarrow b(v) = -2$
  (AVL-property violated, carry out rotation)

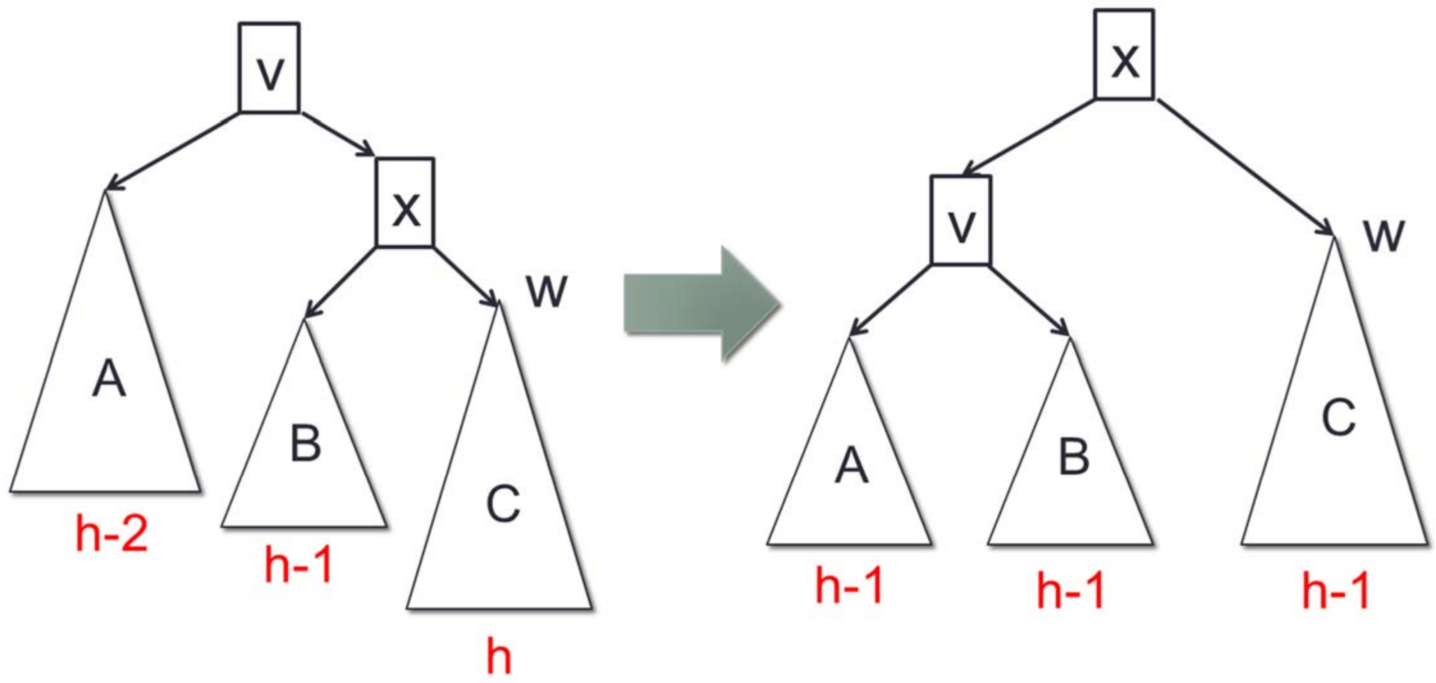
AVL-Tree: Left Rotation

Assume node v and right child x of node v is on the path from z to the root.

w denotes the right child of x on the path

$\Longrightarrow$Left rotation

New balance values: $b(x) = 0$ and $b(v) = 0$



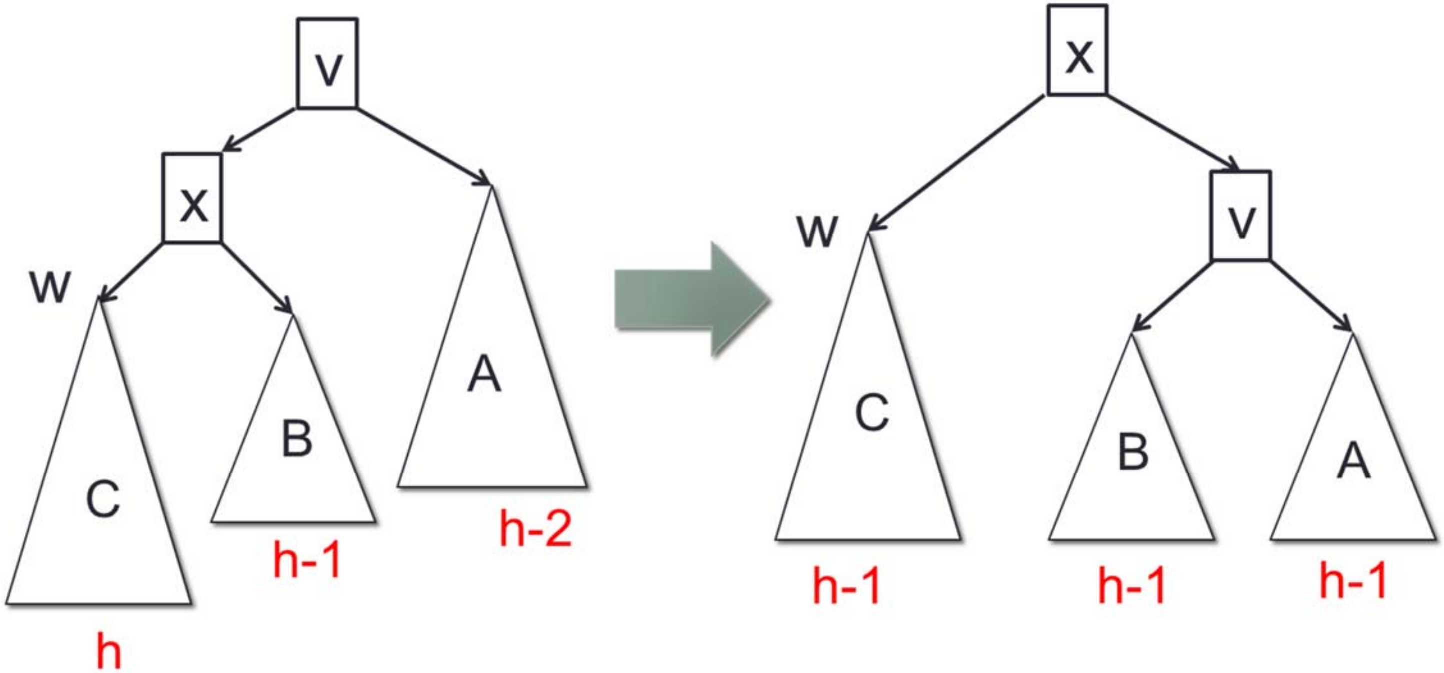
AVL-Tree: Right Rotation

Assume node v and left child x of node v is on the path from z to the root.

w denotes the left child of x on the path

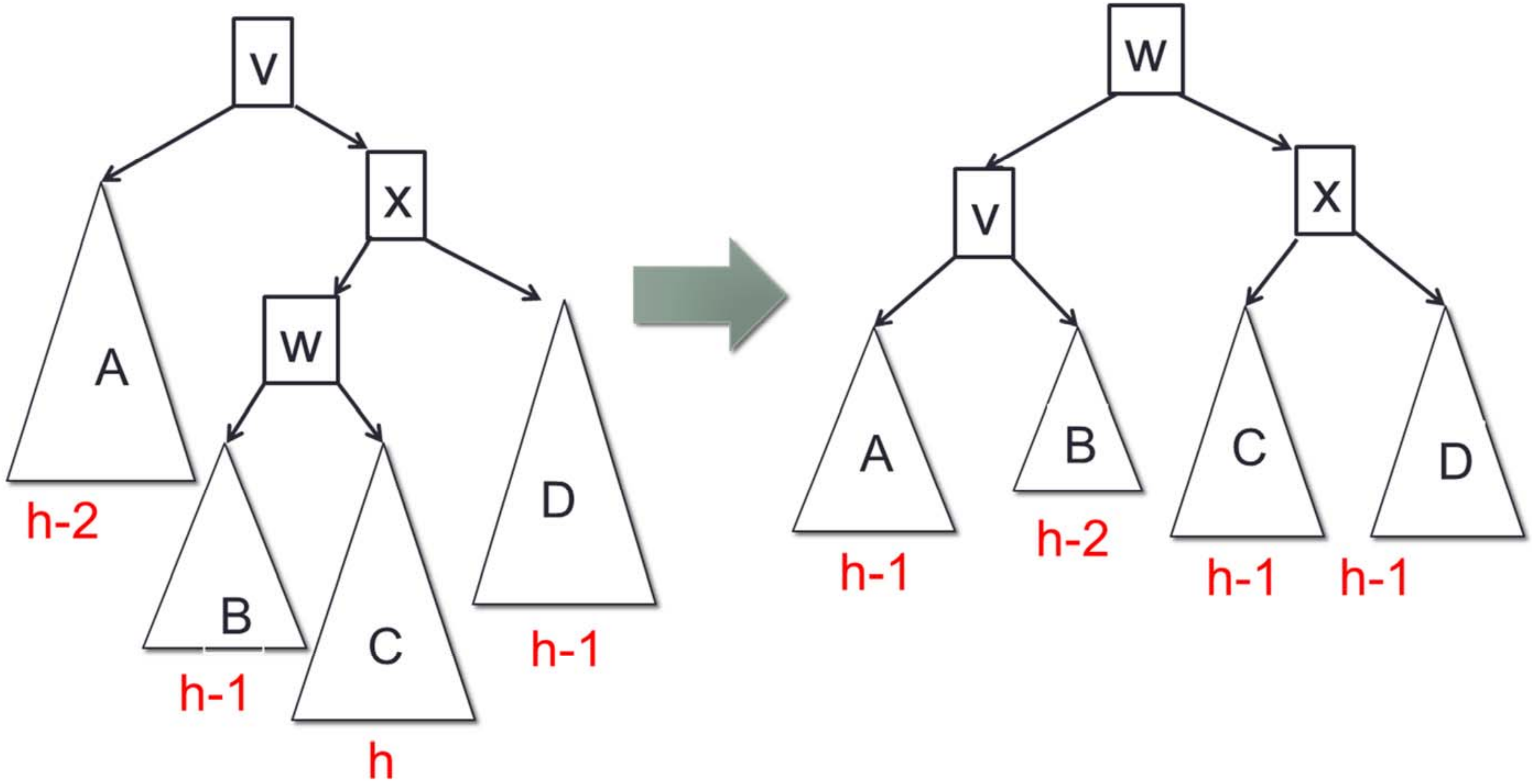
$\Longrightarrow$ Right rotation

New balance values: $b(x) = 0$ and $b(v) = 0$



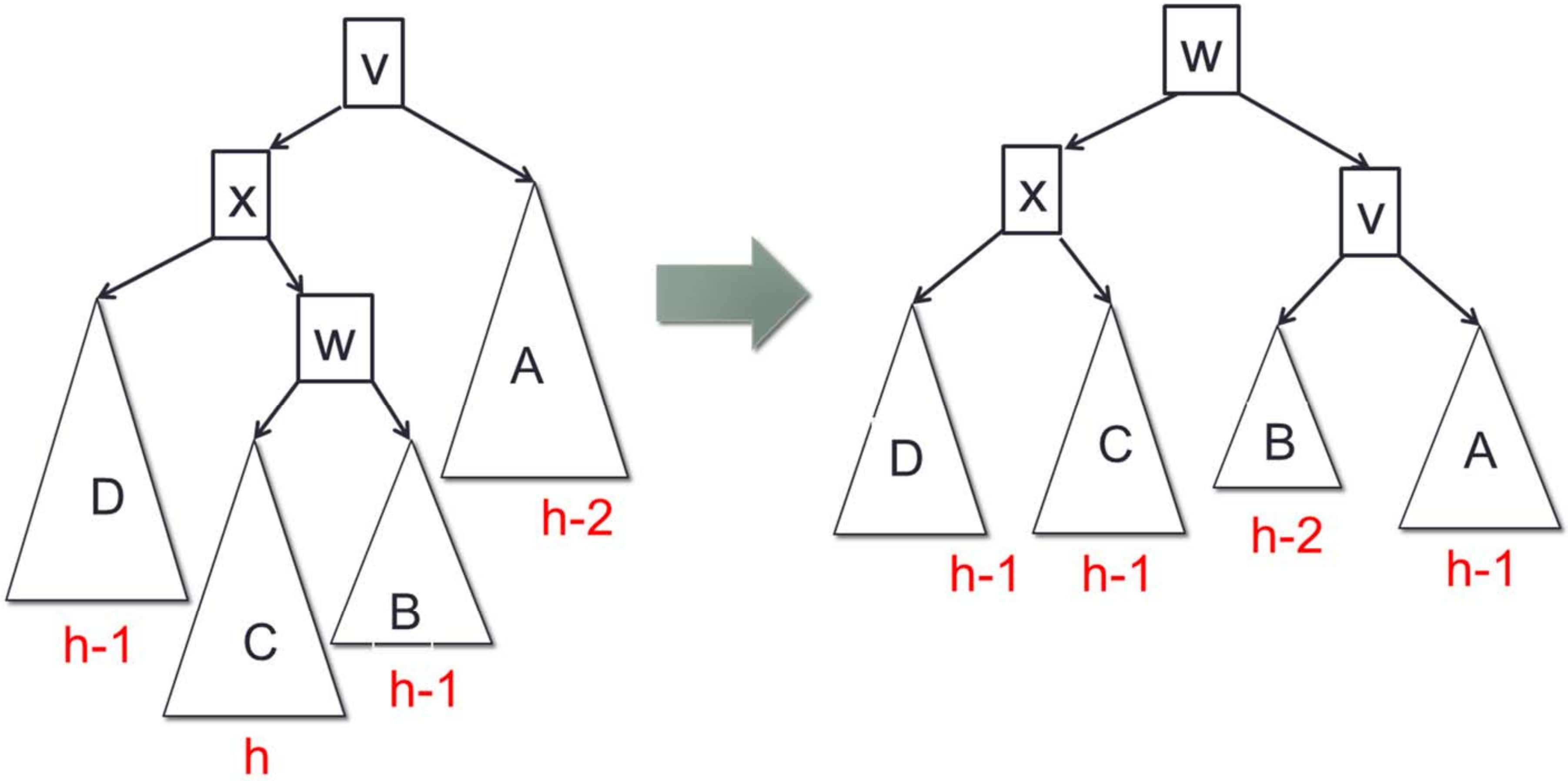
AVL-Tree: Right-Left Rotation

w is left child of x on the path $\Longrightarrow$ Right-Left Rotation.



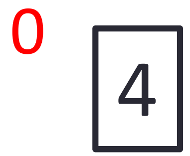
AVL-Tree: Right-Left Rotation

w is right child of x on the path $\Longrightarrow$ Left-Right Rotation.



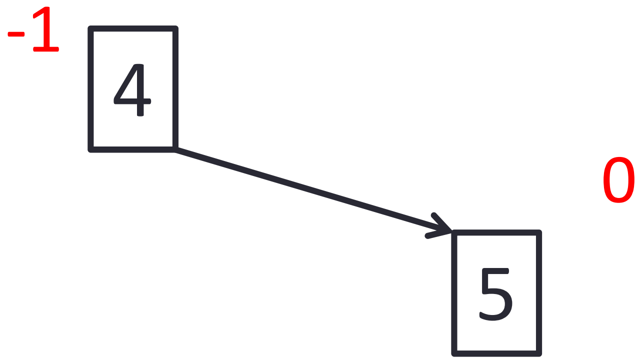
AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



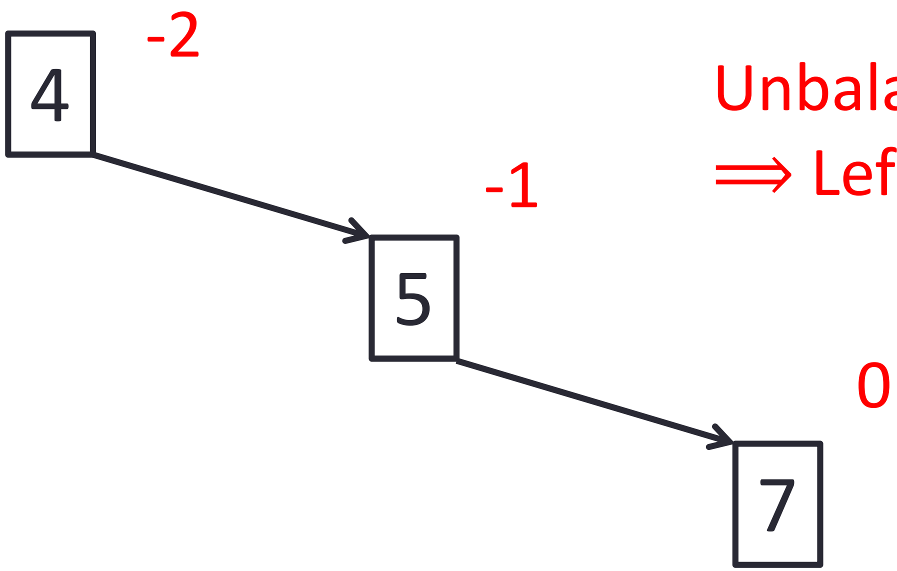
AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6

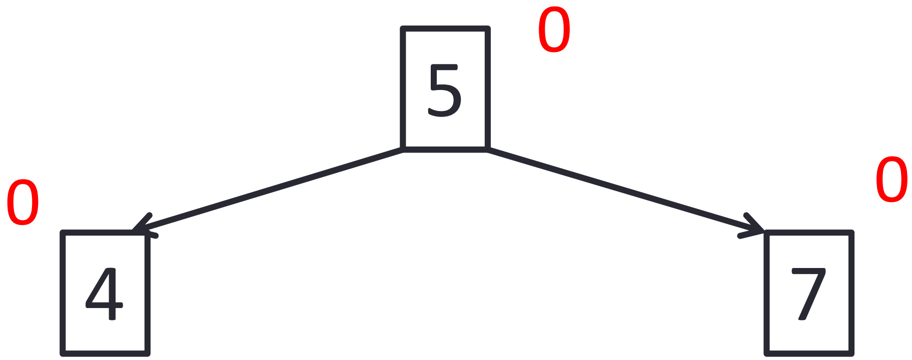


Unbalanced

⟹ Left Rotation

AVL-Tree: Insertion

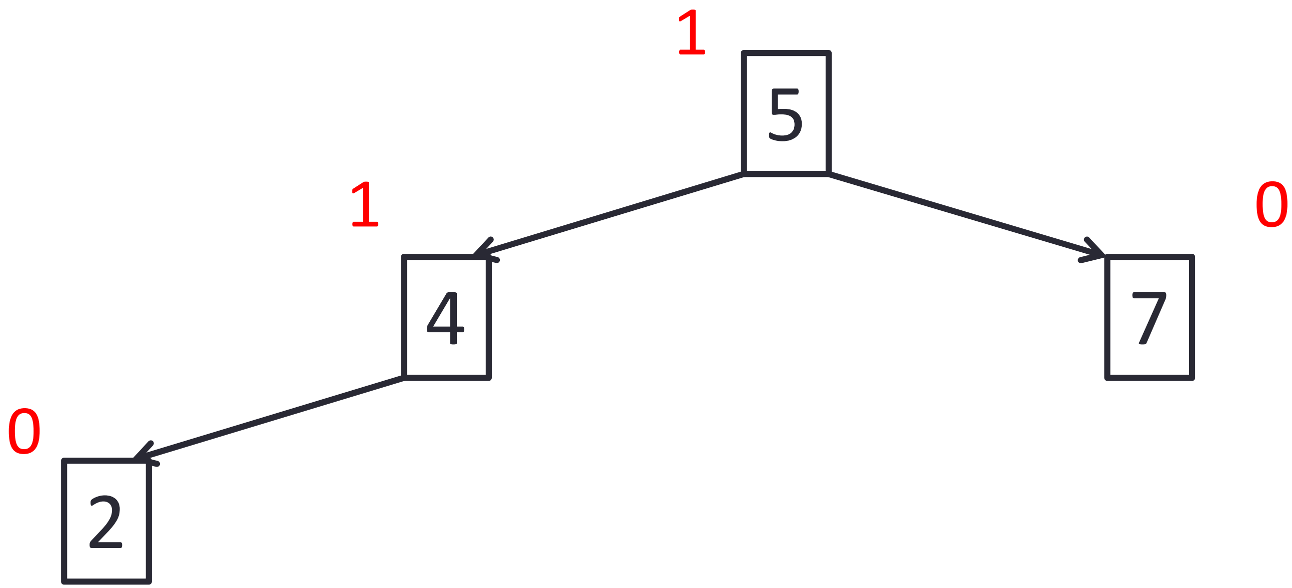
Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



Balance OK

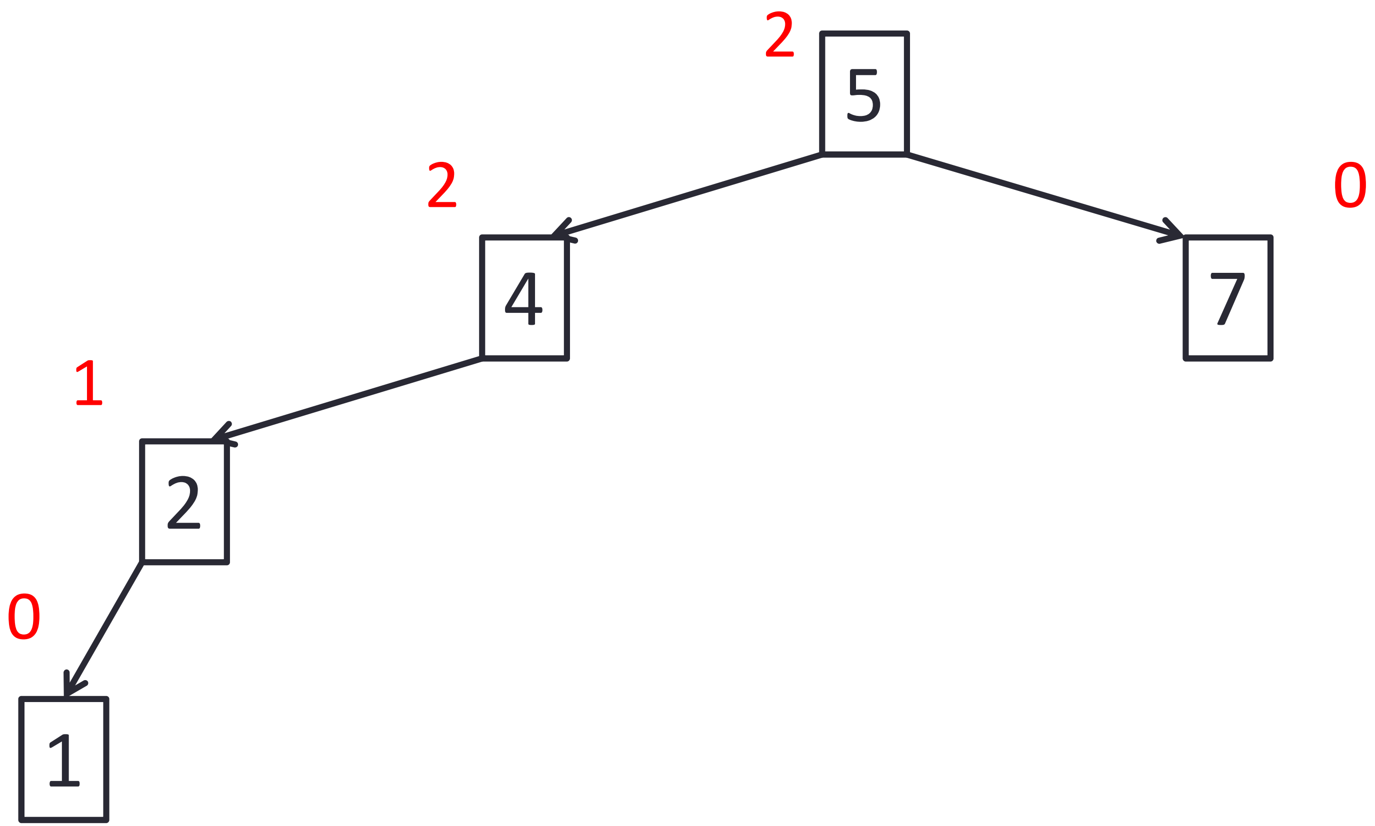
AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



AVL-Tree: Insertion

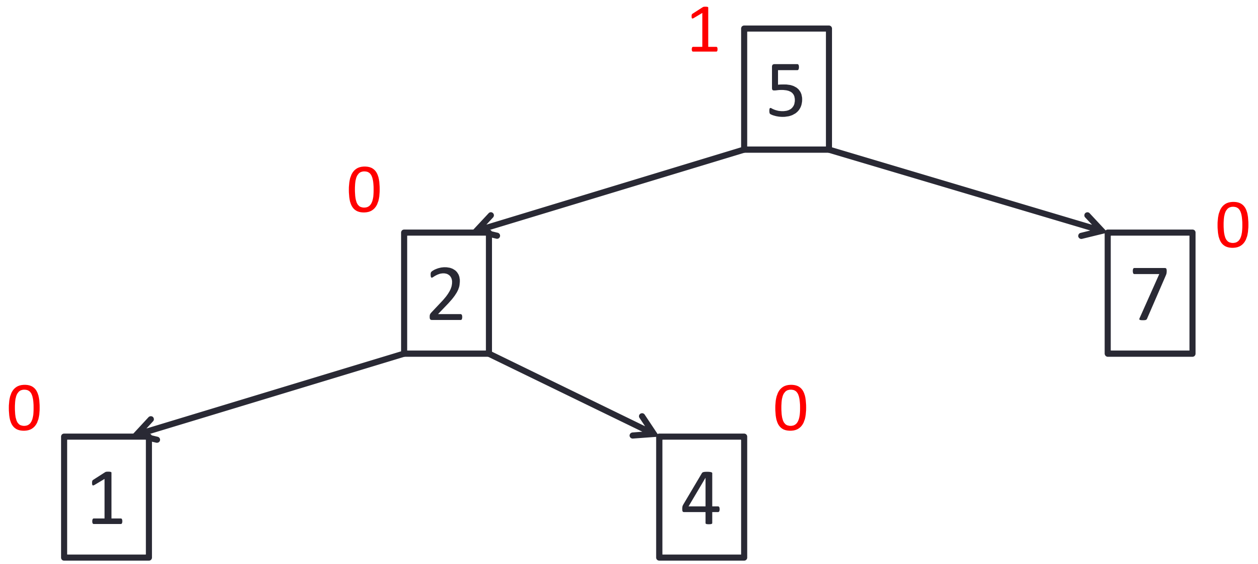
Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



Unbalanced
$\Longrightarrow$ Right Rotation

AVL-Tree: Insertion

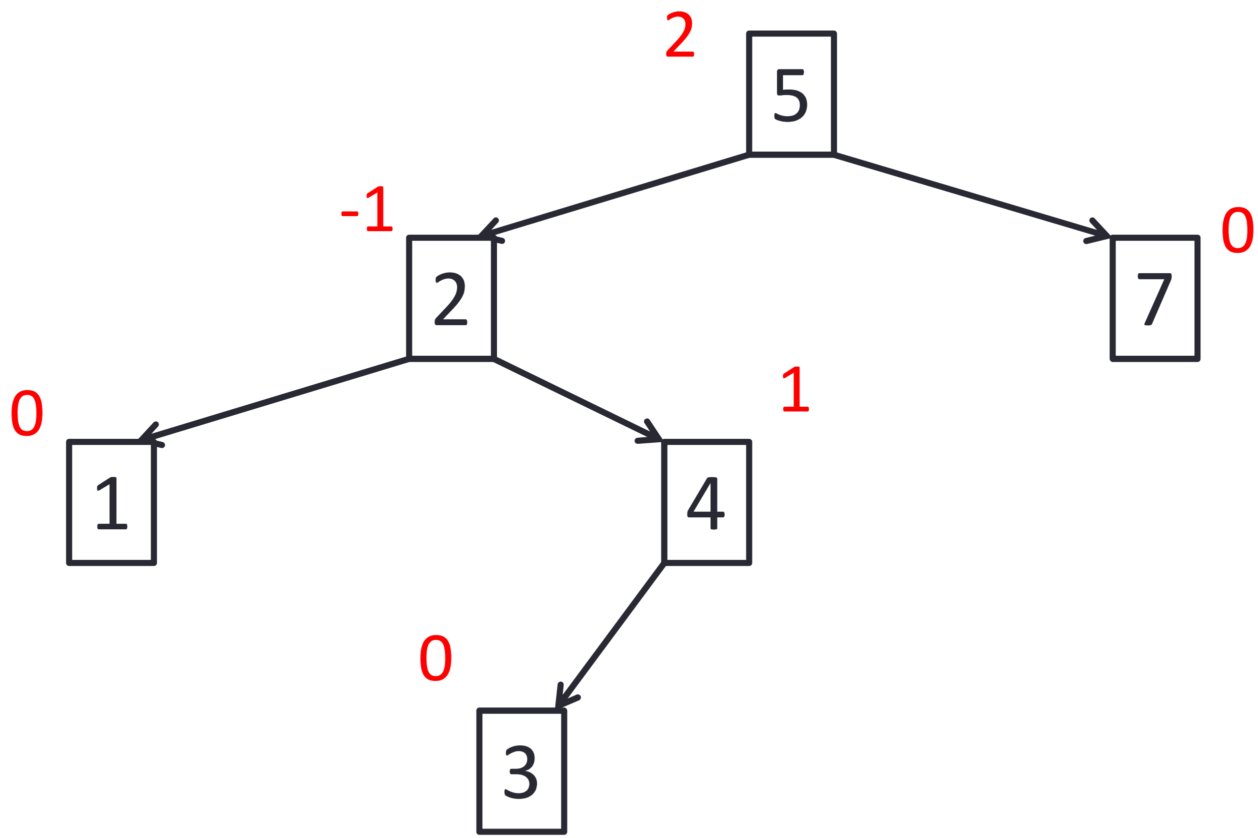
Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



Balance OK

AVL-Tree: Insertion

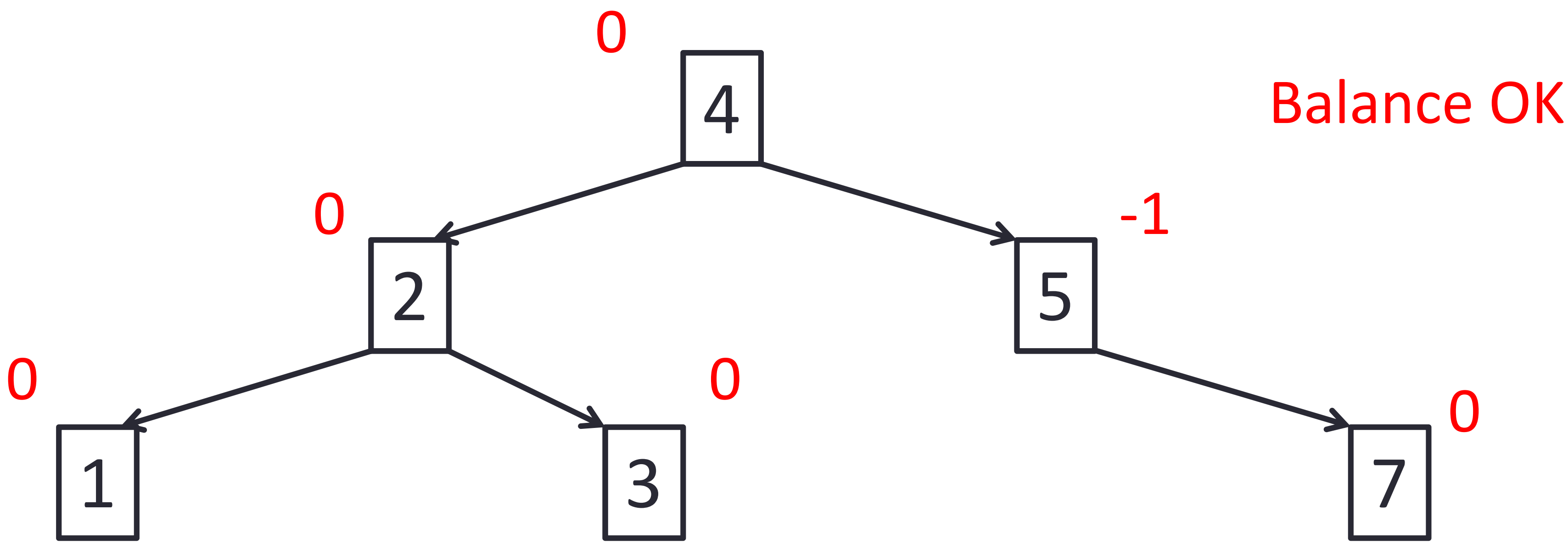
Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



Unbalanced
⟹ Left-right Rotation

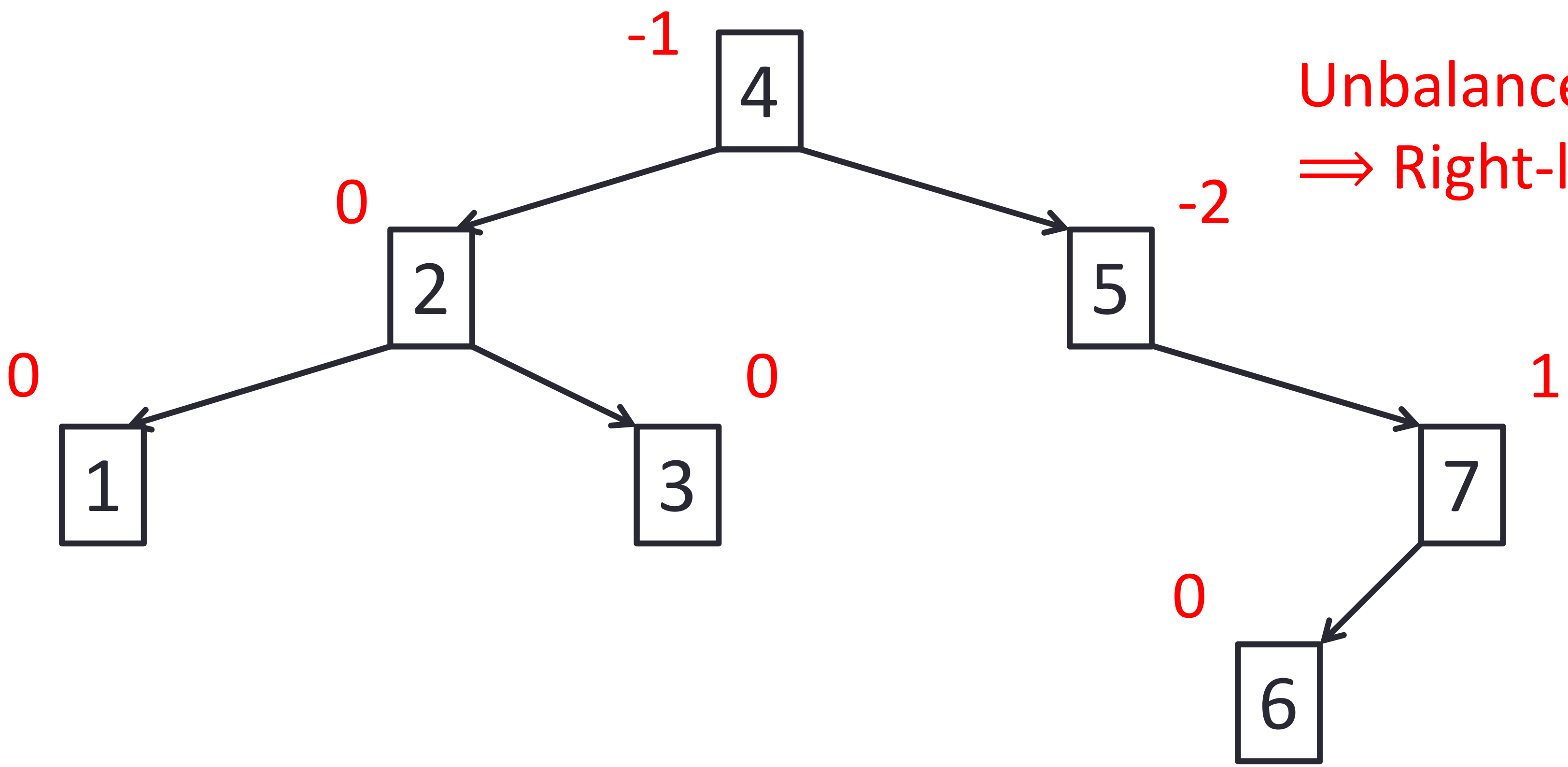
AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6

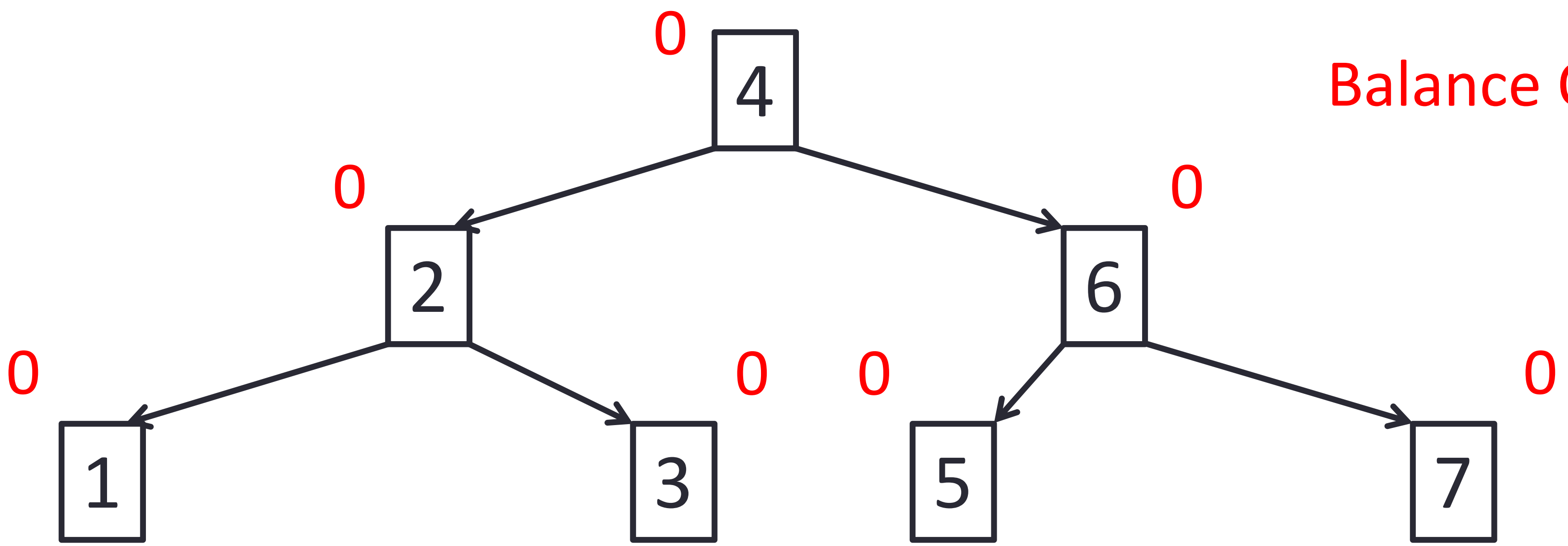


Unbalanced

$\Longrightarrow$ Right-left Rotation

AVL-Tree: Insertion

Example: Create AVL-Tree for sequence 4, 5, 7, 2, 1, 3, 6



Balance OK

AVL-Tree: Algorithm to perform Rotations

- **IF** (*tree is right heavy*)
  - **IF** (*tree's right subtree is left heavy*) perform Right-Left Rotation **ELSE** perform Single Left Rotation
- **ELSE IF** (*tree is left heavy*)
  - **IF** (*tree's left subtree is right heavy*) perform Left-Right Rotation
  - **ELSE** perform Single Right Rotation

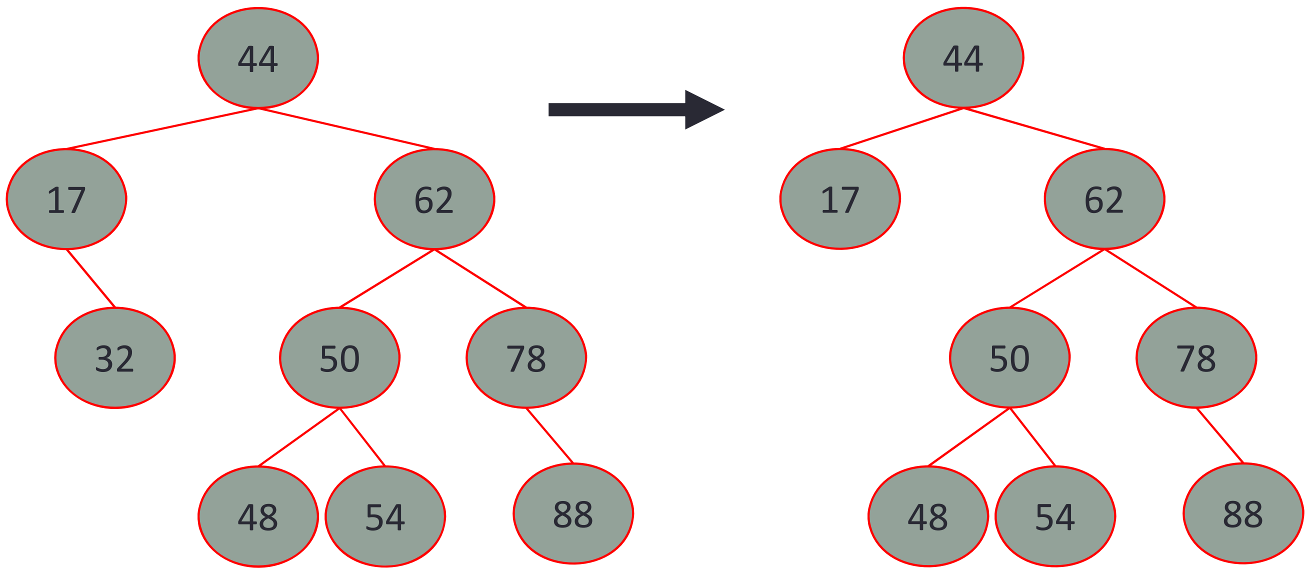
IMPORTANT: Maintain the Binary Search Tree Property

Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. Then we have

$$key(u) \leq key(v) \leq key(w)$$

AVL-Tree: Deletion

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, v, may cause an imbalance.

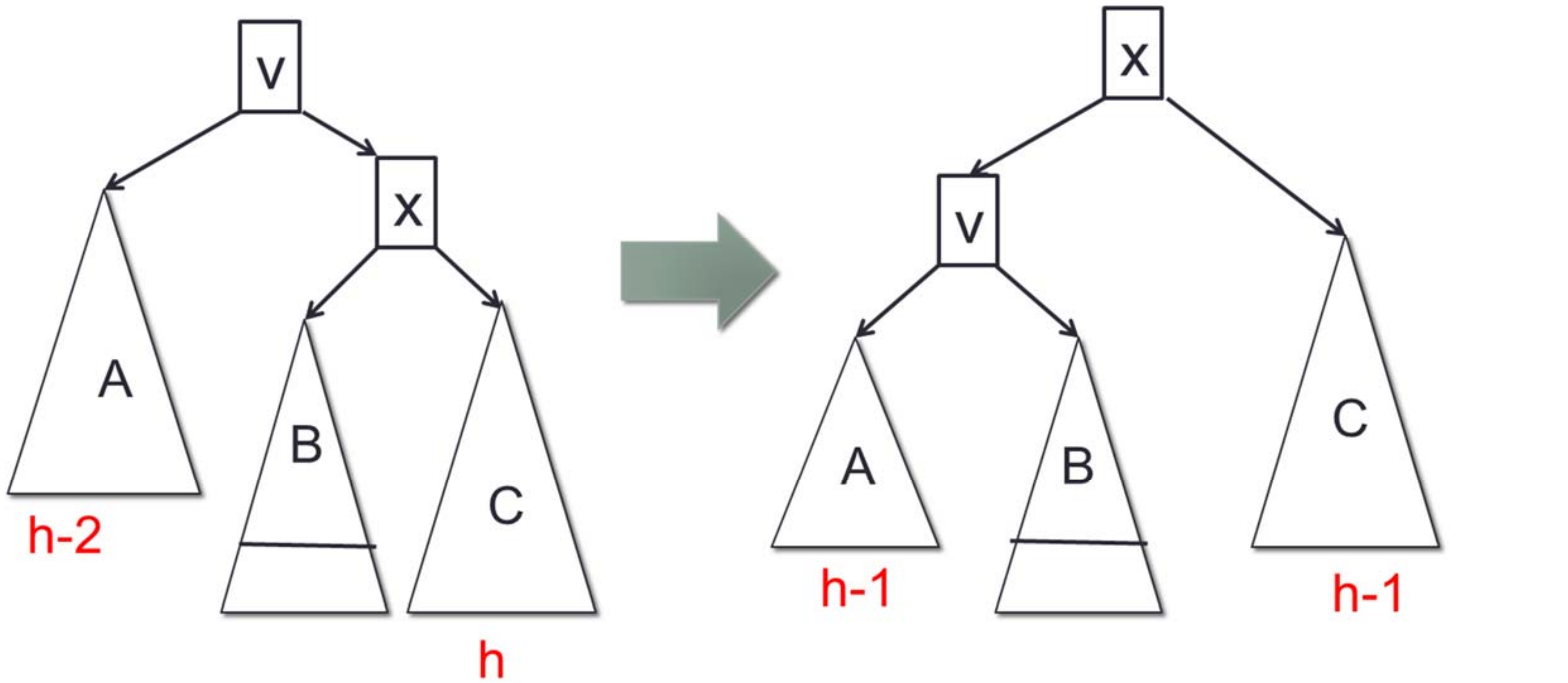


before deletion of 32

after deletion

AVL-Tree: Left Rotation after Deletion

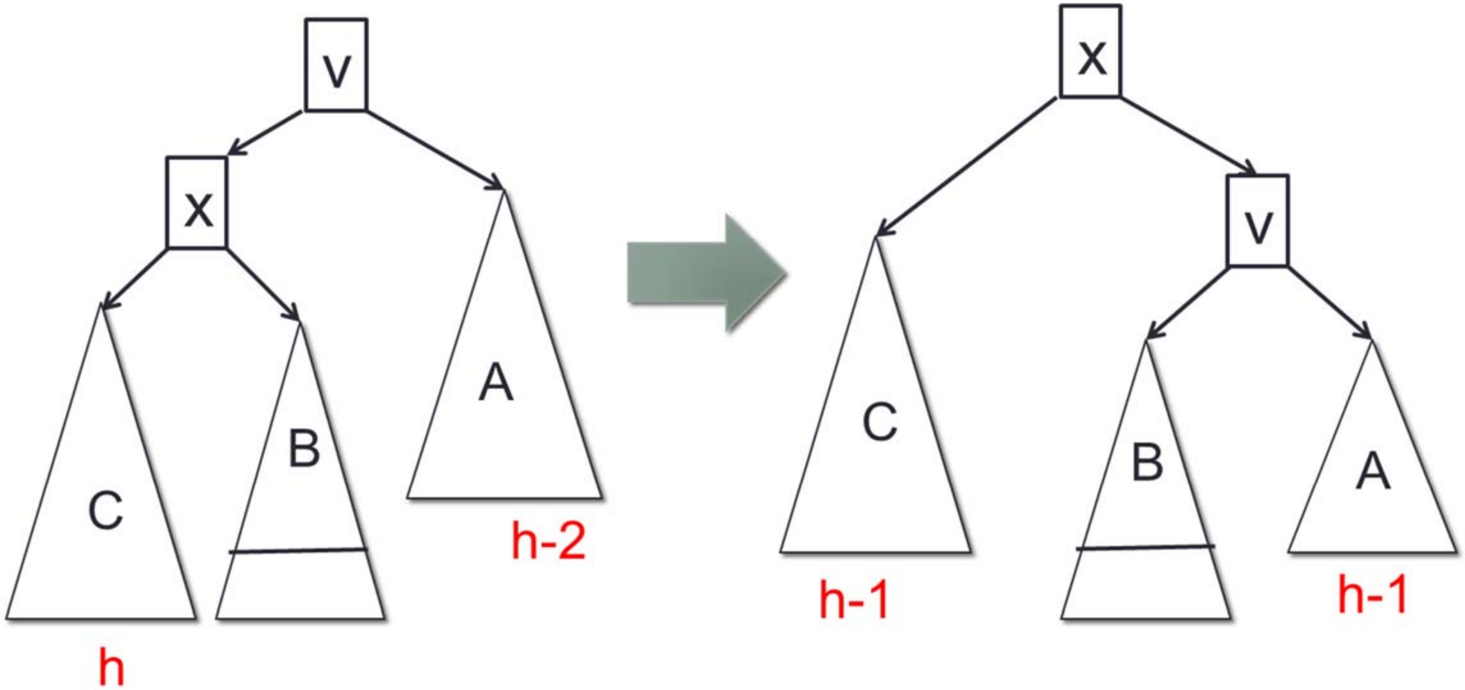
Assume that the deleted node was in the left subtree of v and height of this tree has decrease by 1.



If B had height h-1 before deletion,
the height of the subtree has decreased

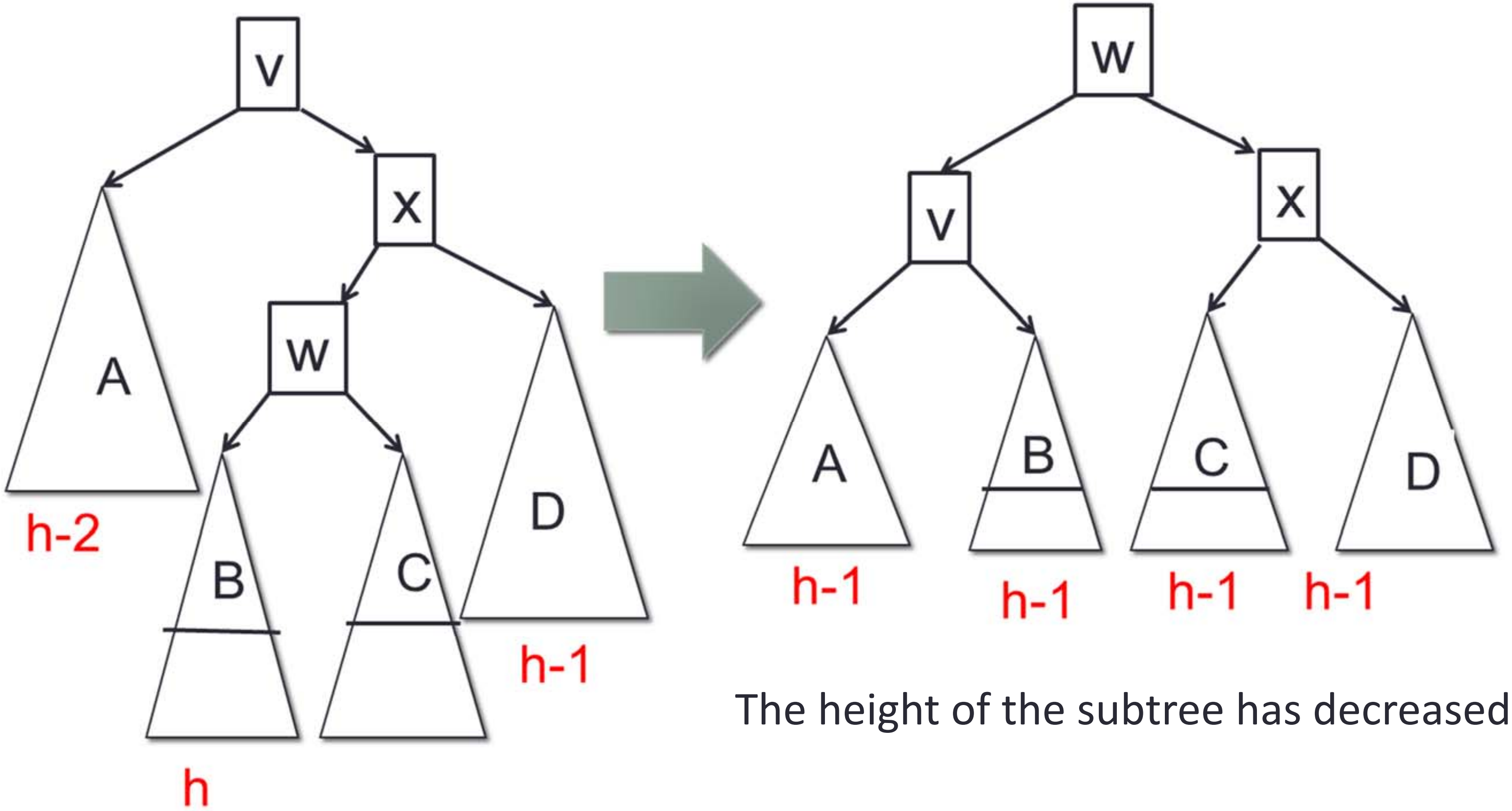
AVL-Tree: Right Rotation after Deletion

Assume that the deleted node was in the right subtree of v and height of this tree has decrease by 1.



If B had height h-1 before deletion,
the height of the subtree has decreased

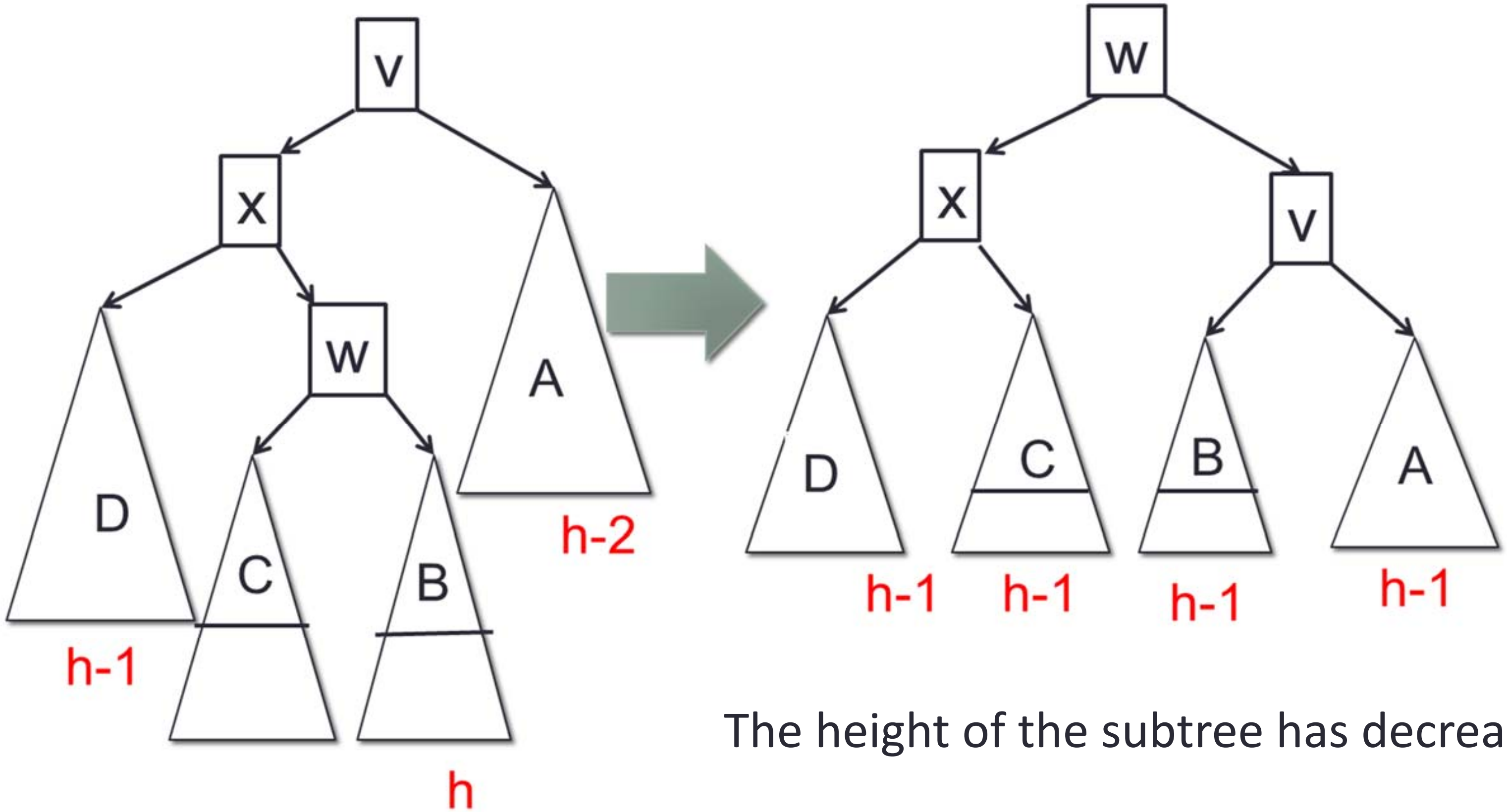
AVL-Tree: Right-Left Rotation after Deletion



The height of the subtree has decreased

Either B or C might have height h-1

AVL-Tree: Left-Right Rotation after Deletion



The height of the subtree has decreased

Either B or C might have height h-1

Rebalancing after Deletion

- After having rebalanced for node v the height of the tree previously rooted at v might have decreased after deleting and rebalancing.
- If this is the case, old parent of v might be imbalanced.
- We might have to **continue rebalancing until the root has been reached**.

Running Times for AVL Trees

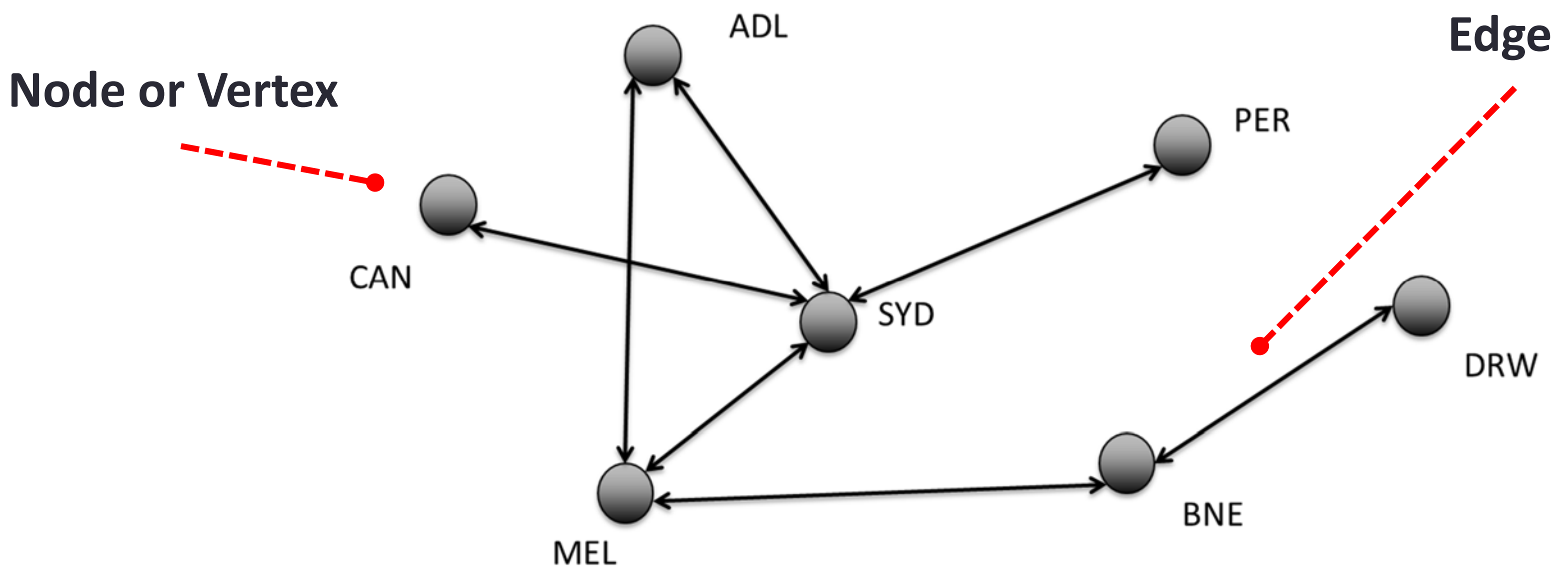
- A single restructure is $O(1)$
  - Using a linked-structure binary tree
- Find is $O(\log n)$
  - Height of tree is $O(\log n)$, no restructures needed
- Insert is $O(\log n)$
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- Remove is $O(\log n)$
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$

Graph: Terminology and Representations

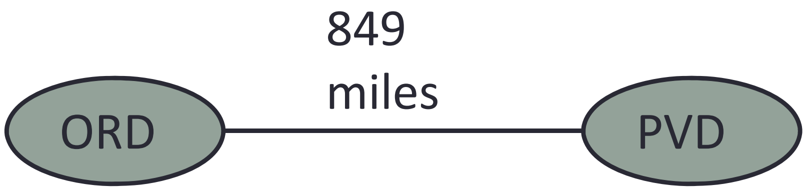
The metropolitan area of Milan, Italy at night. Astronaut photograph ISS026-E-28829, 2011. U.S. government image. NASA-JSC.

Graph: Terminology and Representations

- A graph is a pair $G = (V, E)$, where
  - V is a set of nodes, called vertices.
  - E is a collection of pairs of vertices, called edges.
  - We denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges.
- Example:
  - A vertex represents an airport and stores the three-letter airport code.
  - An edge represents a flight route between two airports and stores the mileage of the route.



Graph: Edge Types

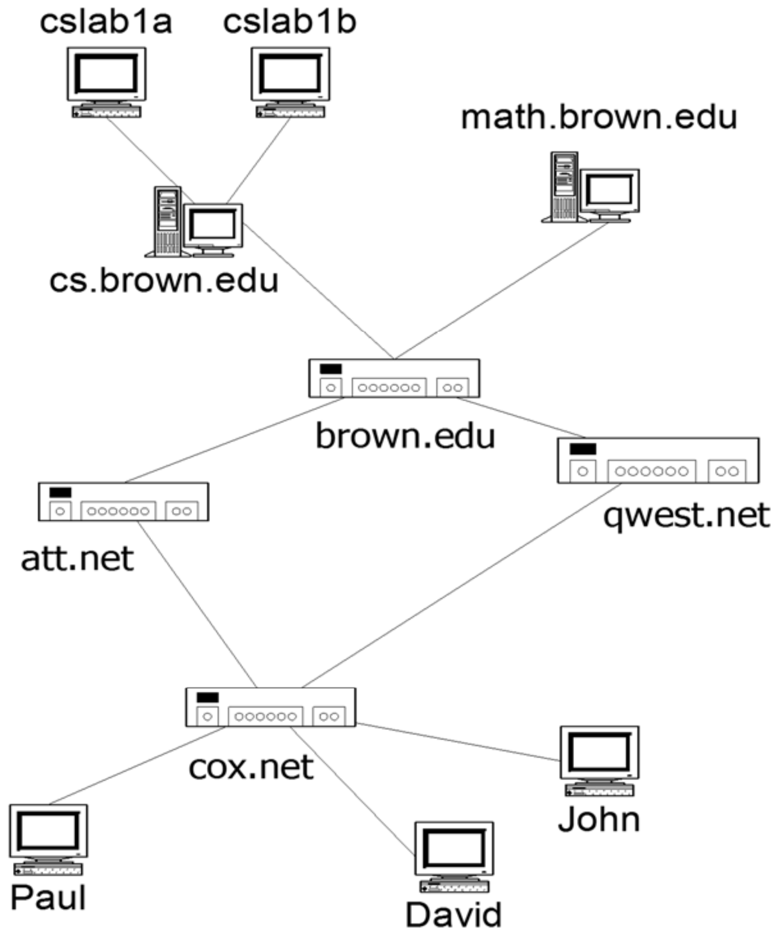


- **Directed edge**
  - ordered pair of vertices (u, v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- **Directed graph**
  - all the edges are directed
  - e.g., route network

- **Undirected edge**
  - unordered pair of vertices (u, v)
  - e.g., a flight route
- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network

Graph: Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

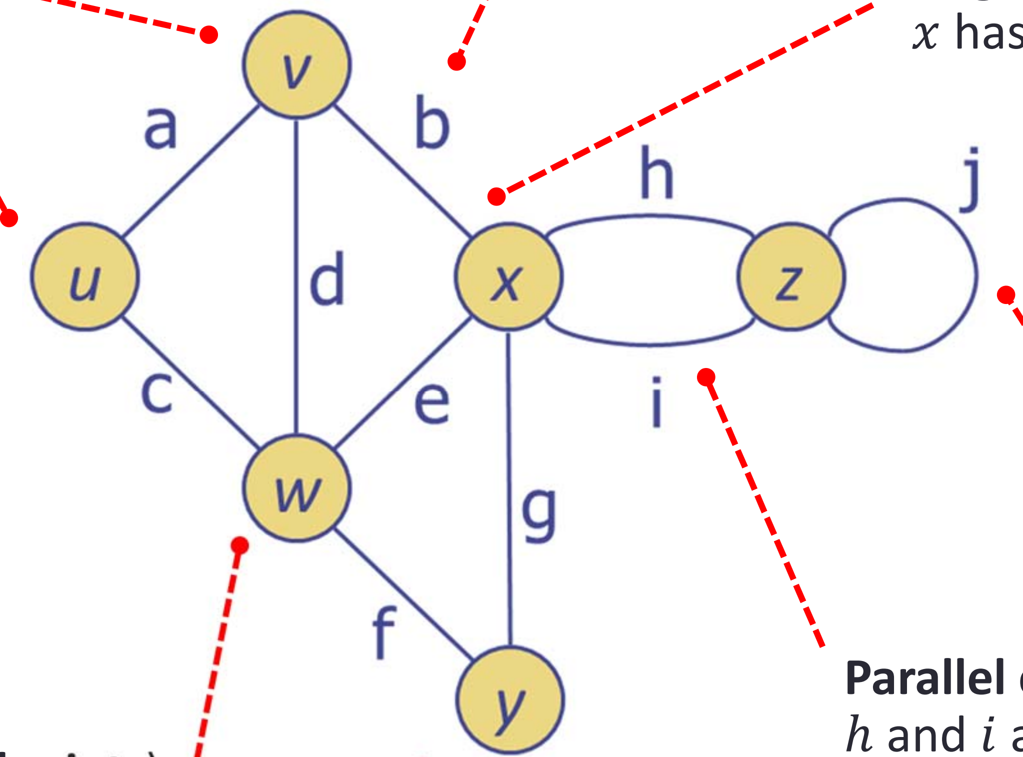


Graph: Terminology

Adjacent vertices:
u and v are adjacent

Edges **incident** on a vertex:
a, d, and b are incident on v

Degree of a vertex:
x has degree 5



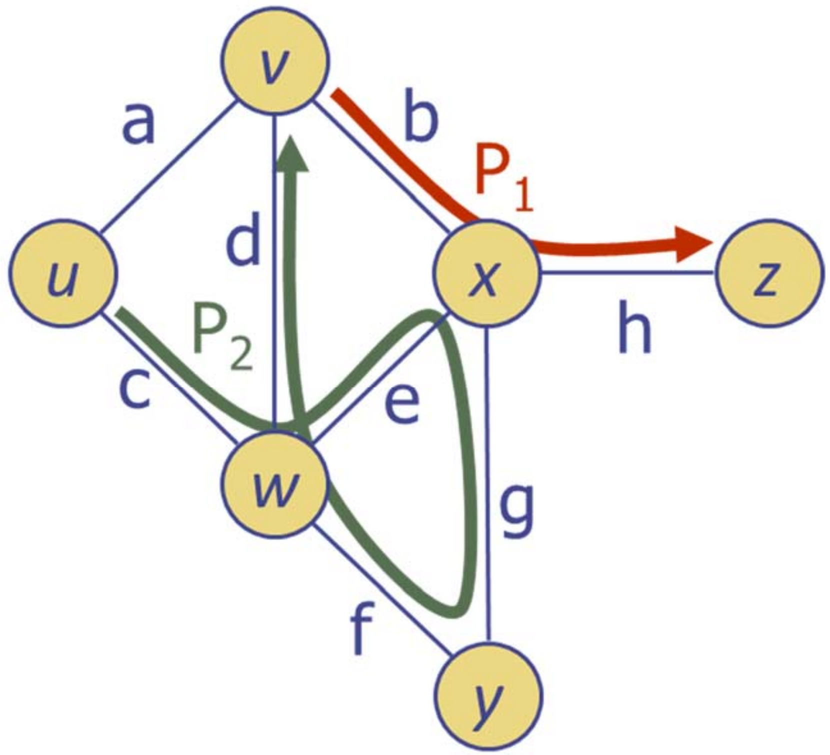
Self-loop:
edge j is a self-loop

Parallel edges:
h and i are parallel edges

End vertices (or **endpoints**) of an edge: w and y are the **endpoints** of edge f

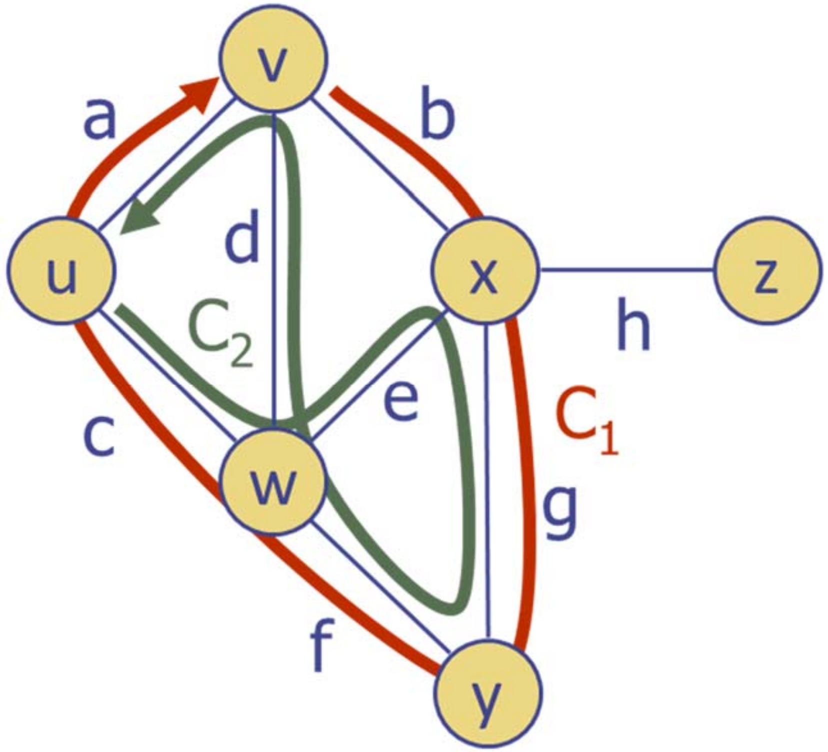
Graph: Terminology

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - $P_1=(v,b,x,h,z)$ is a simple path
  - $P_2=(u,c,w,e,x,g,y,f,w,d,v)$ is a path that is not simple



Graph: Terminology

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - $C_1=(v,b,x,g,y,f,w,c,u,a)$ is a simple cycle
  - $C_2=(u,c,w,e,x,g,y,f,w,d,v,a)$ is a cycle that is not simple



Graph: Terminology

- The number of outgoing edges of a vertex v is called the outdegree of v :

$$outdegree(v) = |\{(v,u) \in E\}|$$

- The number of incoming edges of a vertex v is called the indegree of v :

$$indegree(v) = |\{(u,v) \in E\}|$$

Graph: Terminology

- A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if

$$V' \subseteq V \text{ and } E' \subseteq E.$$

- Given a graph $G = (V, E)$ and a subset $V' \subseteq V$, the subgraph induced by V' is defined as

$$G' = (V', E \cap (V' \times V'))$$

Graph: Simple Graph Algorithm

Given a directed graph $G = (V, E)$. Is G acyclic?

Observation:

Node with outdegree zero can not appear in a cycle.

Idea for an algorithm:

- If there is a node v with outdegree zero, delete v (and the incoming edges) to obtain a graph G';
- G is acyclic if and only if G' is acyclic.

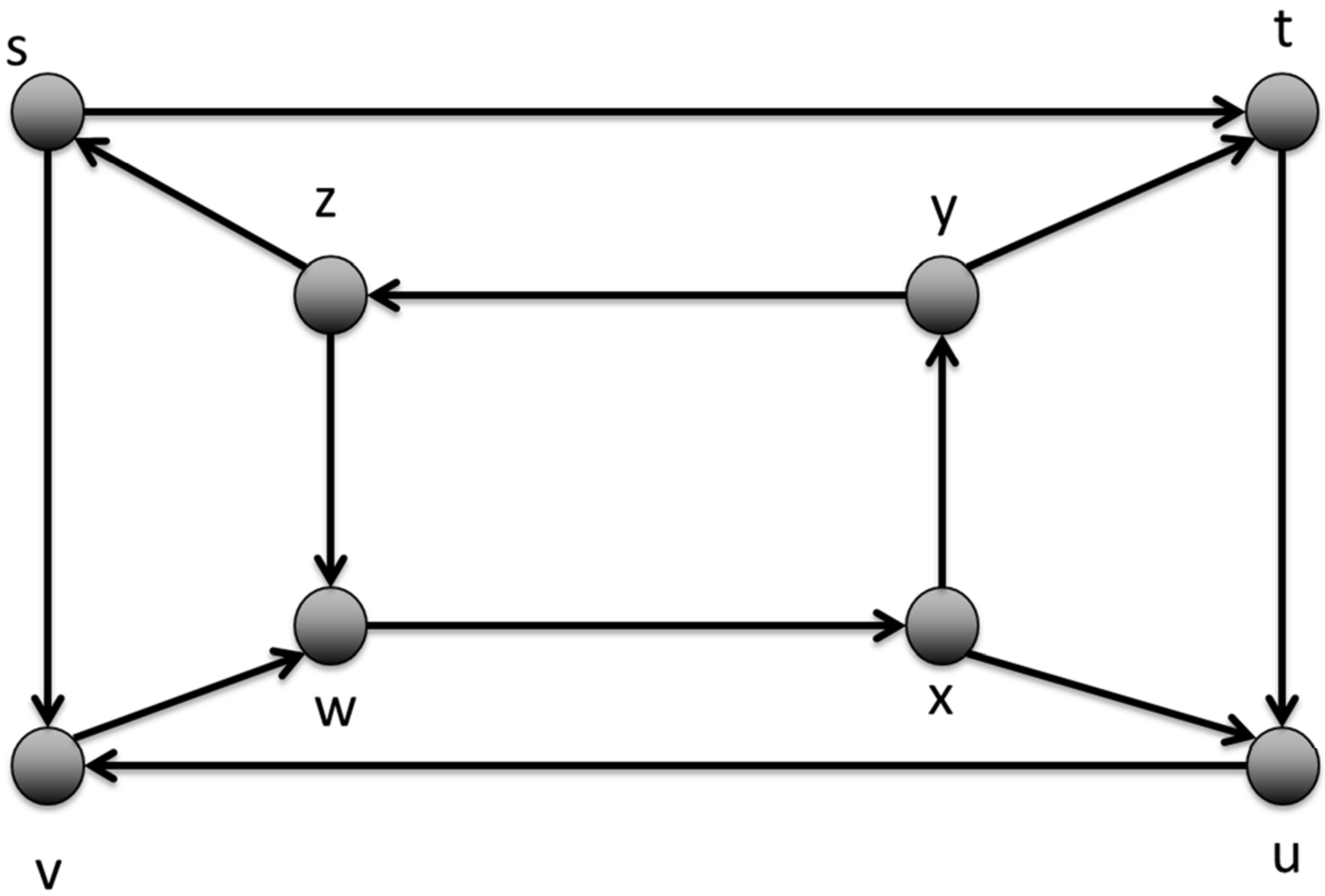
Graph: Simple Graph Algorithm

- If there is a node v of outdegree zero, delete v and its incoming edges to obtain a graph G'.
- Iterate the transformation.

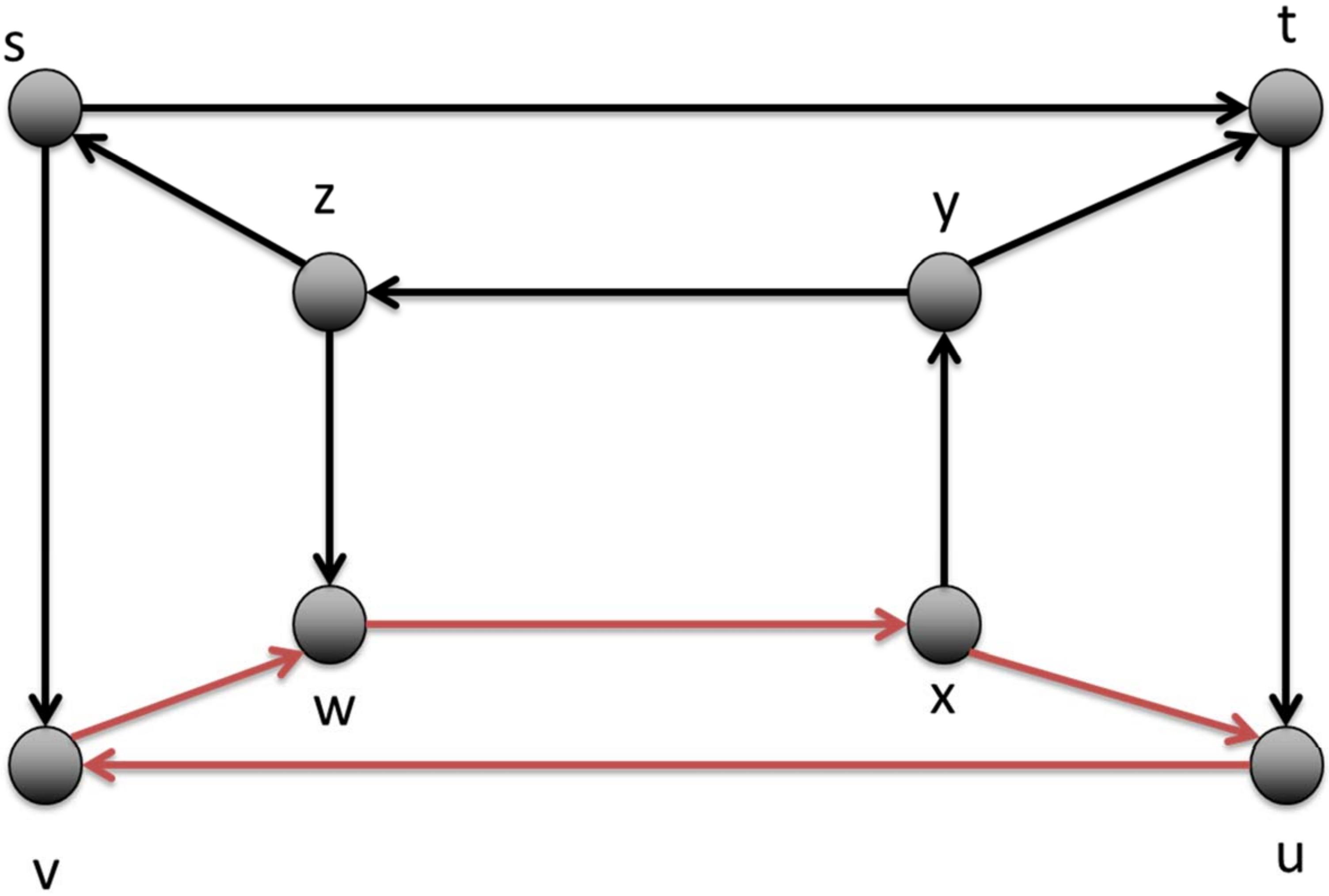
Arrive at a graph G^*

- If G^* is the empty graph then G is acyclic;
- If G^* is not the empty graph, we can find a cycle in G^* that is also present in G.

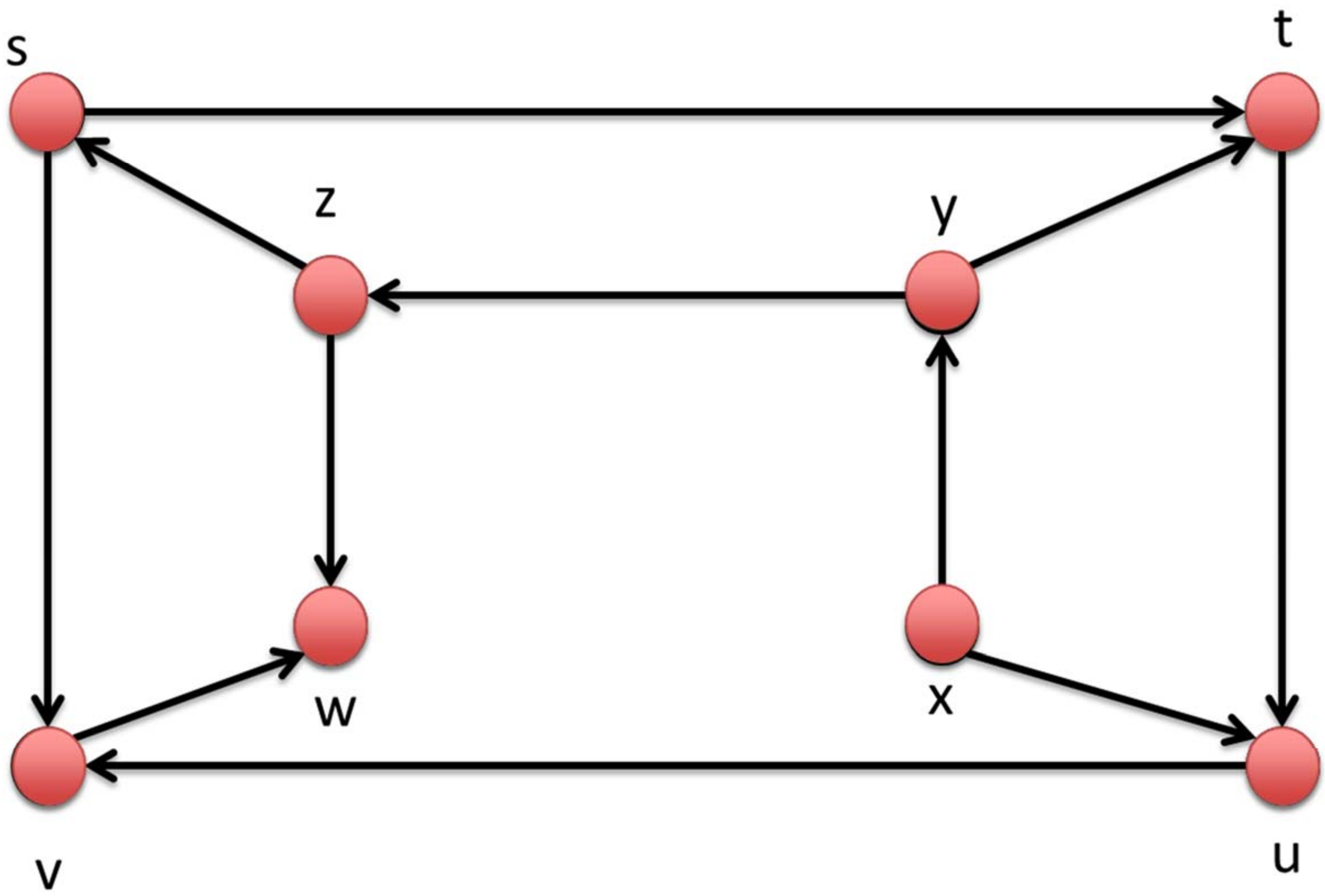
Graph: Graph containing a cycle



Graph: Graph containing a cycle



Graph: Acyclic Graph



Empty Graph G^* implies that G is acyclic

Graph: Trees

An undirected graph is called a tree if there is exactly one path between any pair of nodes.

The following properties of an undirected graph G are equivalent:

1. G is a tree.
2. G is connected and has exactly $n - 1$ edges.
3. G is connected and contains no cycles.

Graph: Operations

We want efficiently support the following operations for graphs:

- **Accessing associated information**
  (get the information stored at nodes and edges)
- **Navigation** (access the edges incident to a node)
- **Edge queries** (ask whether an edge is in the graph, query its reverse edge)
- **Construction, conversion and output**
  (translate one graph representation into another)
- **Update** (Add and remove nodes and edges)

Graph: Representation

Simplest choice:

Unordered sequence of edges
(e.g. linked list of edges).

Good if you just want to output the edges of the graph.

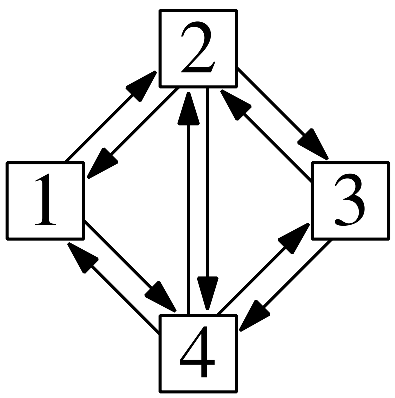
Problem:

Most interesting operations take time $\Theta(m)$.

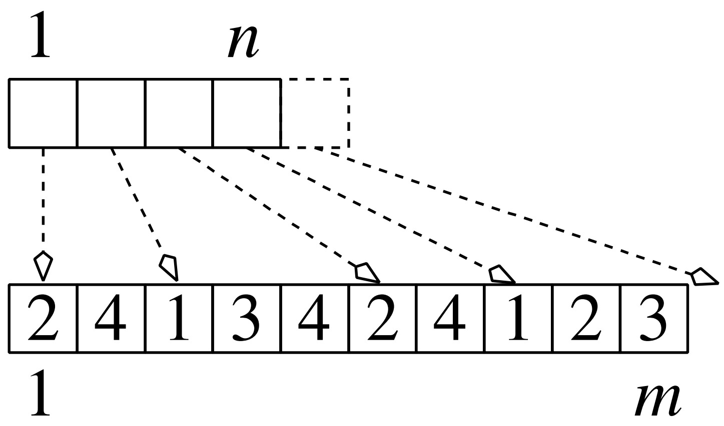
Graph: Adjacency Arrays

- Assume that the graph is static (i.e. it does not change).
- Then we can store the graph in an array.
- Store the outgoing neighbors of each node in a subarray and concatenate these subarrays into a single edge array E.
- Use an additional array V to store the starting positions of the subarrays.
- Memory consumption: $\mathrm{n} + \mathrm{m} + \Theta(1)$.

Graph: Adjacency Arrays



(Bi)-directed Graph



Adjacency Array

- For any node v, $V[v]$ is the index of the first outgoing edge of v.
- Add dummy entry $V[n+1] = m+1$.
- Outgoing edges of node v are accessible at

  $\mathrm{E}\big[V[v]\big], \ldots, \mathrm{E}[V[v+1]-1]$

Graph: Adjacency Arrays

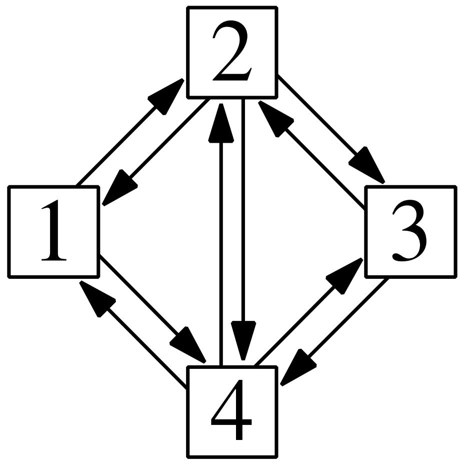
Are there better representations that allow to add or remove edges in constant time?

Two popular choices:

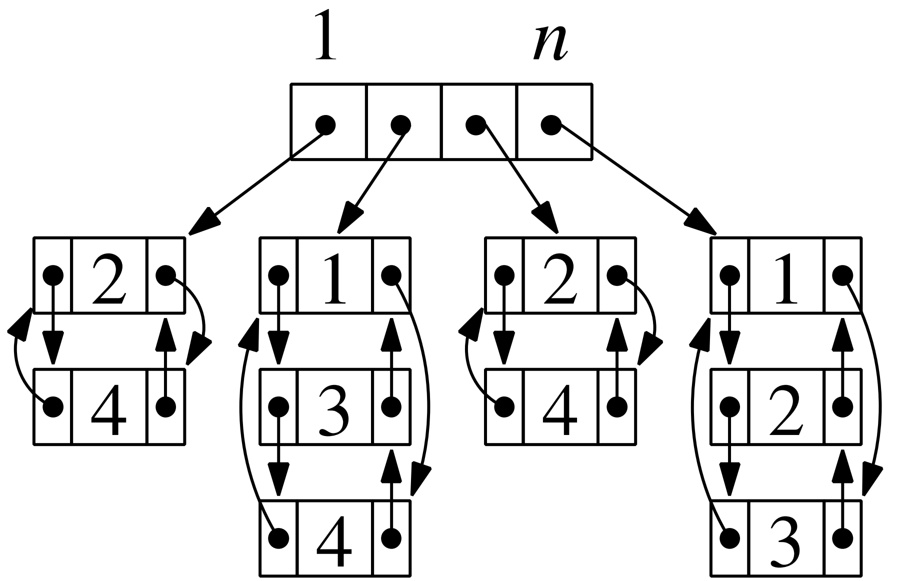
Adjacency Lists

Adjacency Matrices

Graph: Adjacency Arrays



(Bi)-directed Graph



Adjacency List

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency Matrix

Graph: Adjacency Arrays

Idea: Use for each node v a double-linked list that stores its outgoing neighbors (alternatively we can also use the incoming neighbors or lists for both).

Advantage:

- Insertion of edges goes in constant time.
- Well suited for sparse graphs (occur often in practice).

Graph: Adjacency Matrices

Idea: Represent a graph consisting of v nodes by an $v \times v$ matrix A. Set

$$A_{ij} = 1 \text{ if } (i, j) \in E$$
$$A_{ij} = 0 \text{ otherwise}$$

Insertion, removal, edge queries work in constant time.
$O(n)$ to obtain an edge entering or leaving a node.

Disadvantage: Storage requirement n^2 even for sparse graphs.

Other references and things to do

- Have a look at the attached references in CloudDeakin.
- Read chapters 9.4.2, 11.3, and 14.2 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.