# 算法设计与分析(2026年春季学期)

# Data Structures

授课老师: 栗师

南京大学计算机学院

**Q:** What is a Data Structure?

**A:** A way to store data in memory that supports a specified set of query and update operations.

- The performance of a data structure can be measured by its space usage, and the running time needed for each supported operation.

## Examples of Data Structures

- array, queue, stack, linked list (linear data structures)
- set, map, hash map, priority queue
- binary search tree (BST), self-balancing BST
- union-find

- abstract data structure: a model defined by the set of supported operations, without an implementation. It is not associated with time complexities for operations, or space usage.
- (concrete) data structure: a concrete implementation of some abstract data structure. So, it has a time complexity for each operation, and space usage.

- data structures are often used as tools to design (more) efficient algorithms.
- for a given problem, define operations needed, and choose a data structure that suits the need.

# Outline

- Let $V$ be a ground set of size $n$.

**Def.** A priority queue is an abstract data structure that maintains a set $U \subseteq V$ of elements, each with an associated key value, and supports the following operations:

- insert$(v, key_value)$: insert an element $v \in V \setminus U$, with associated key value $key\_value$.
- decrease_key$(v, new_key_value)$: decrease the key value of an element $v \in U$ to $new\_key\_value$
- extract_min$()$: return and remove the element in $U$ with the smallest key value
- remove, increase_key, get_min, $\cdots$

## Example:

- insert(a, 9), insert(b, 3), insert(c, 5), insert(d, 10), ▷ {a:9, b:3, c:5, d:10}
- extract_min(), return b ▷ {a:9, c:5, d:10}
- decrease_key(d, 4) ▷ {a:9, c:5, d:4}
- extract_min(), return d ▷ {a:9, c:5}
- insert(e, 12), ▷ {a:9, c:5, e:12}
- extract_min(), return c ▷ {a:9, e:12}
- extract_min(), return a ▷ {e:12}

## Algorithms in this course that use priority queue

- Greedy Algorithm for Offline Cache
- Prim's Algorithm for Minimum Spanning Tree
- Dijkstra's Algorithm for Shortest Path

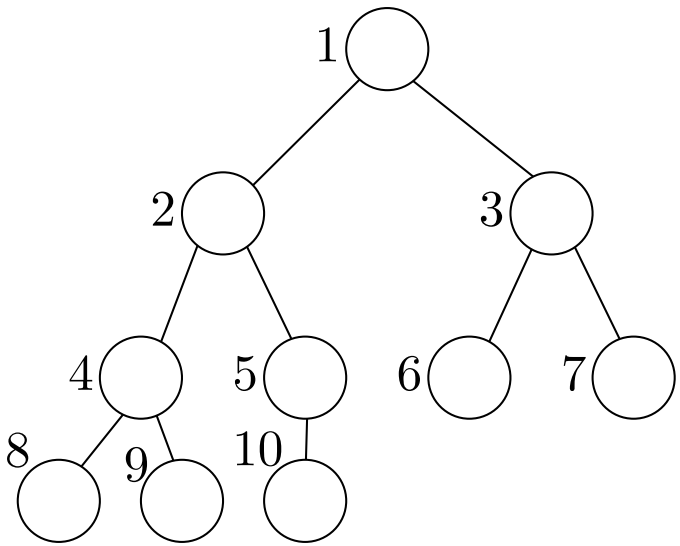
# Simple Implementations for Priority Queue

- $n =$ size of ground set $V$

| **data structures** | **insert** | **extract_min** | **decrease_key** |
|---|---|---|---|
| array | $O(1)$ | $O(n)$ | $O(1)$ |
| sorted array | $O(n)$ | $O(1)$ | $O(n)$ |
| (binary) heap | $O(\lg n)$ | $O(\lg n)$ | $O(\lg n)$ |

# Heap

The elements in a heap is organized using a complete binary tree:

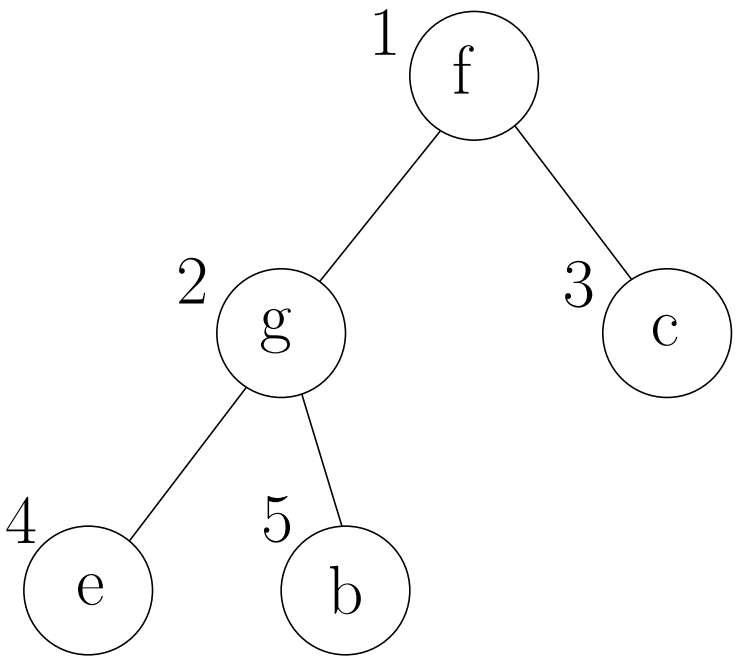



- Nodes are indexed as $\{1, 2, 3, \cdots, s\}$
- Parent of node $i$: $\lfloor i/2 \rfloor$
- Left child of node $i$: $2i$
- Right child of node $i$: $2i + 1$

# (Binary) Heap

A heap $H$ contains the following fields

- $s$: size of $U$ (number of elements in the heap)
- $A[i], 1 \leq i \leq s$: the element at node $i$ of the tree
- $p[v], v \in U$: the index of node containing $v$
- $key[v], v \in U$: the key value of element $v$

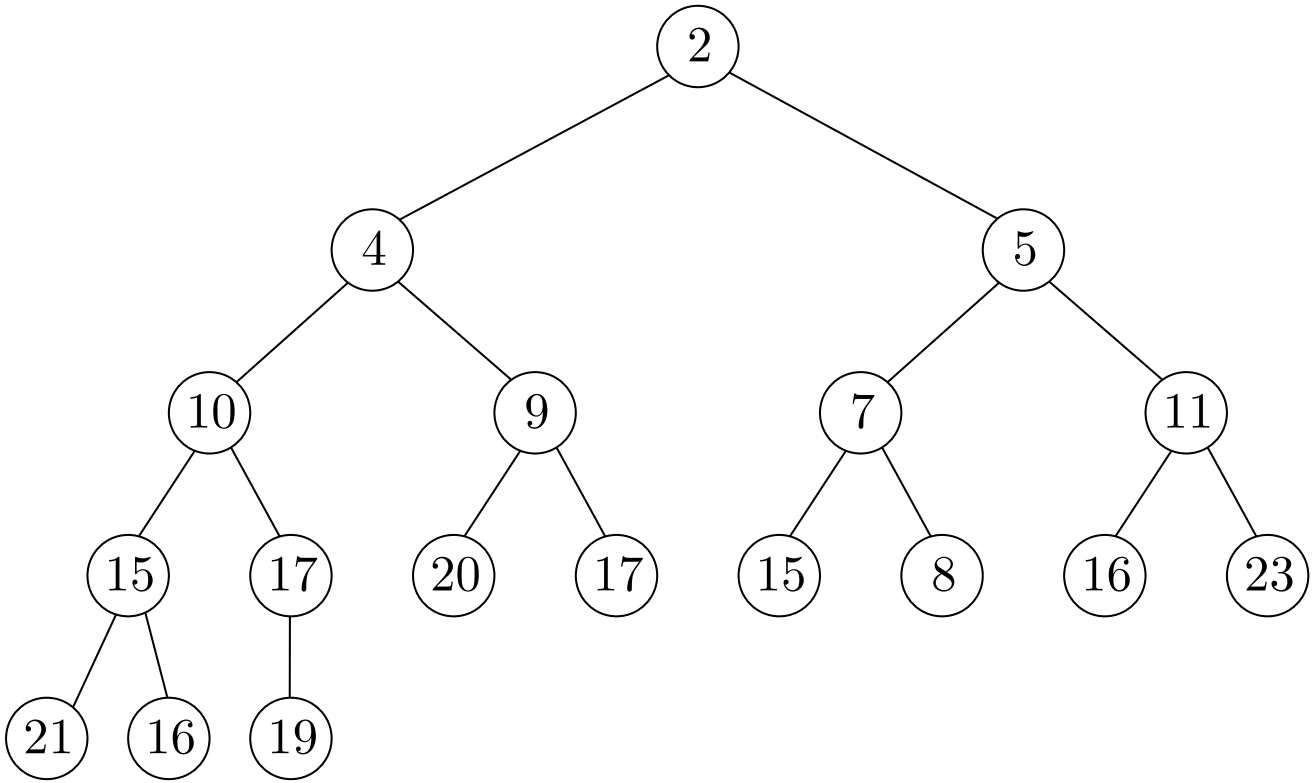



- $s = 5$
- $A = (\mathrm{f}, \mathrm{g}, \mathrm{c}, \mathrm{e}, \mathrm{b})$
- $p[\mathrm{f}] = 1, p[\mathrm{g}] = 2, p[\mathrm{c}] = 3,$
  $p[\mathrm{e}] = 4, p[\mathrm{b}] = 5$

# Heap

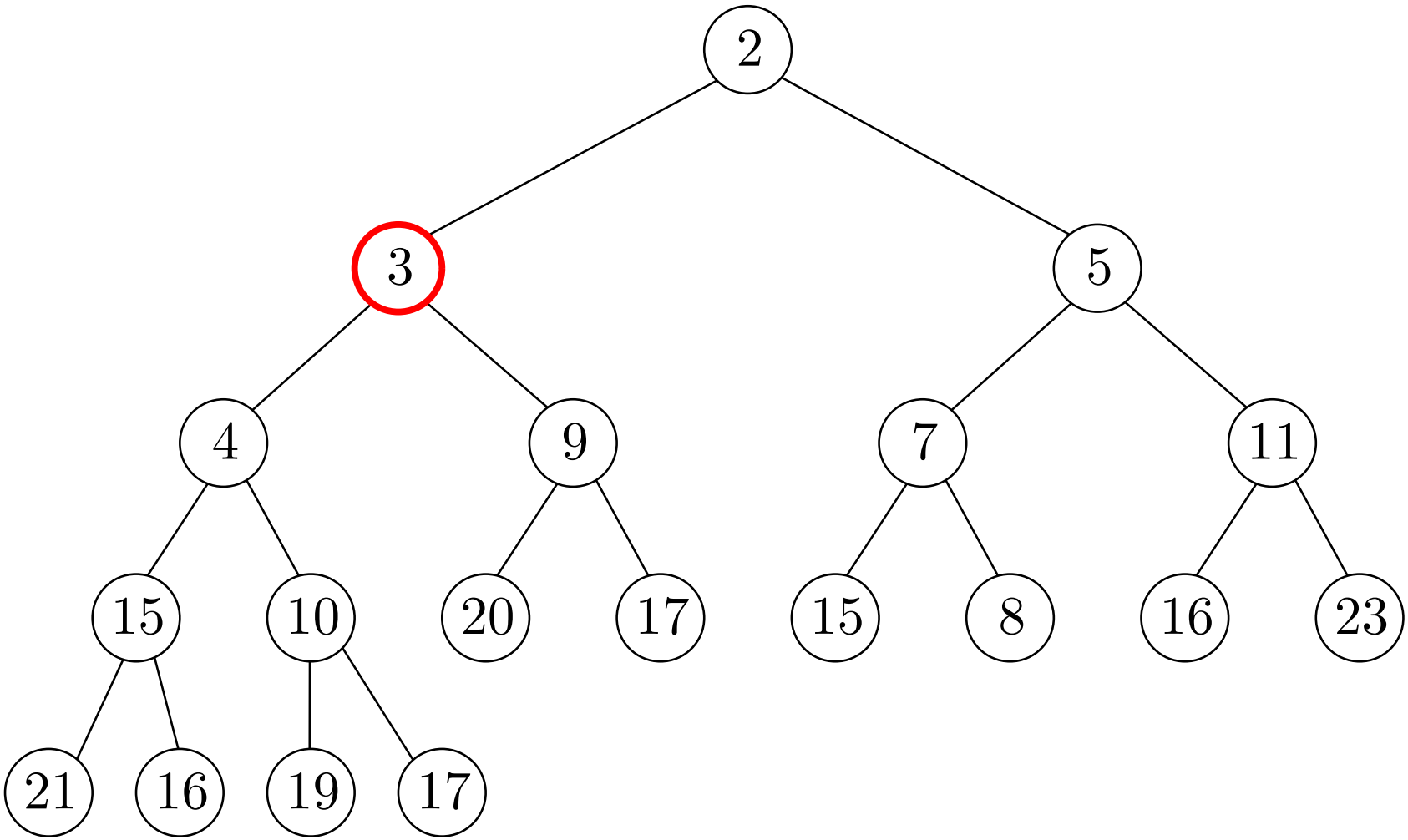
The following heap property is satisfied:

- for any two nodes $i$, $j$ such that $i$ is the parent of $j$, we have $key[A[i]] \leq key[A[j]]$.




A heap. Numbers in the circles denote key values of elements.

# insert$(v, key_value)$



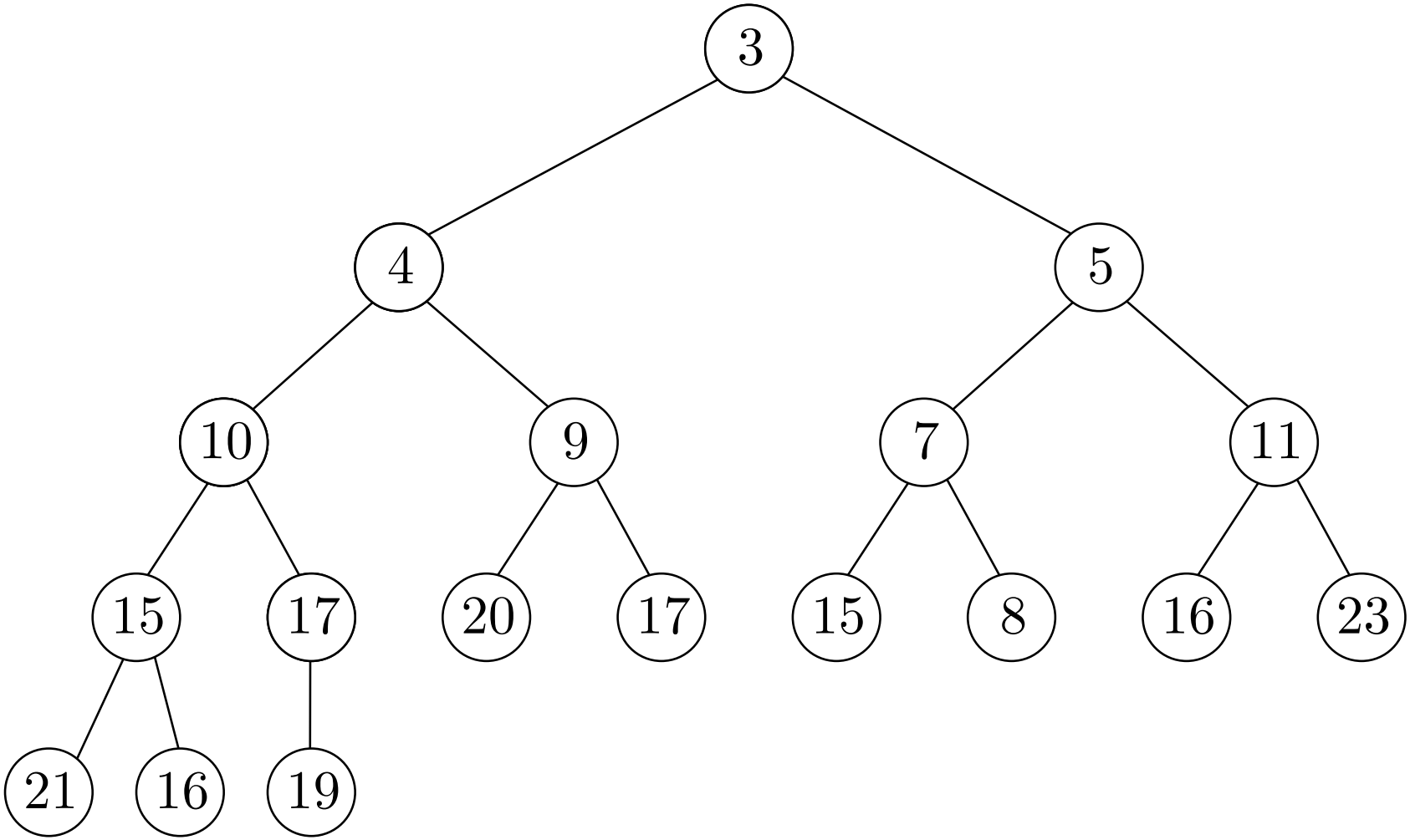
## insert$(v, key_value)$

1: $s \leftarrow s + 1$
2: $A[s] \leftarrow v$
3: $p[v] \leftarrow s$
4: $key[v] \leftarrow key\_value$
5: heapify_up$(s)$

## heapify-up$(i)$

1: **while** $i > 1$ **do**
2: $\quad j \leftarrow \lfloor i/2 \rfloor$
3: $\quad$ **if** $key[A[i]] < key[A[j]]$ **then**
4: $\quad\quad$ swap $A[i]$ and $A[j]$
5: $\quad\quad p[A[i]] \leftarrow i$, $p[A[j]] \leftarrow j$
6: $\quad\quad i \leftarrow j$
7: $\quad$ **else** break

# extract_min()



## extract_min()

1: $ret \leftarrow A[1]$
2: $A[1] \leftarrow A[s]$
3: $p[A[1]] \leftarrow 1$
4: $s \leftarrow s - 1$
5: **if** $s \geq 1$ **then**
6: heapify_down$(1)$
7: **return** $ret$

## decrease_key$(v, key_value)$

1: $key[v] \leftarrow key\_value$
2: heapify-up$(p[v])$

## heapify-down$(i)$

1: **while** $2i \leq s$ **do**
2: **if** $2i = s$ or $key[A[2i]] \leq key[A[2i+1]]$ **then**
3: $j \leftarrow 2i$
4: **else**
5: $j \leftarrow 2i + 1$
6: **if** $key[A[j]] < key[A[i]]$ **then**
7: swap $A[i]$ and $A[j]$
8: $p[A[i]] \leftarrow i$, $p[A[j]] \leftarrow j$
9: $i \leftarrow j$
10: **else** break

- Running time of heapify_up and heapify_down: $O(\lg n)$
- Running time of insert, exact_min and decrease_key: $O(\lg n)$

| **data structures** | **insert** | **extract_min** | **decrease_key** |
|---|---|---|---|
| array | $O(1)$ | $O(n)$ | $O(1)$ |
| sorted array | $O(n)$ | $O(1)$ | $O(n)$ |
| heap | $O(\lg n)$ | $O(\lg n)$ | $O(\lg n)$ |
| Fibonacci heap | $O(1)$ | $O(\log n)$ | $O(1)$ |

- Note: running times for Fibonacci heap are amortized running time:

**Def.** One operation has amortized running time $T$, if for any integer $s \geq 1$, the first $s$ executions of the operation have total running time at most $Ts$.

# Two Definitions Needed to Prove that the Procedures Maintain Heap Property

**Def.** We say that $H$ is almost a heap except that $key[A[i]]$ is too small if we can increase $key[A[i]]$ to make $H$ a heap.

**Def.** We say that $H$ is almost a heap except that $key[A[i]]$ is too big if we can decrease $key[A[i]]$ to make $H$ a heap.

**Lemma** At the beginning of any iteration of the while loop in heapify_up, $H$ is almost a heap except that $key[A[i]]$ is too small.

**Lemma** At the beginning of any iteration of the while loop in heapify_down, $H$ is almost a heap except that $key[A[i]]$ is too big.

# Outline

A self-balancing binary search tree $T$ maintains a set of comparable elements and supports:

- Insertion of an element to $T$
- Deletion of an element from $T$
- Whether an element exists in $T$
- Return the rank of an element in $T$ (i.e, 1 plus number of elements in $T$ smaller than the element)
- Return the $i$-th smallest element in $T$
- Each operation takes time $O(\lg n)$
- A self-balancing BST supports more operations than a priority queue does, thus is harder to implement.

# Example: Counting Inversions

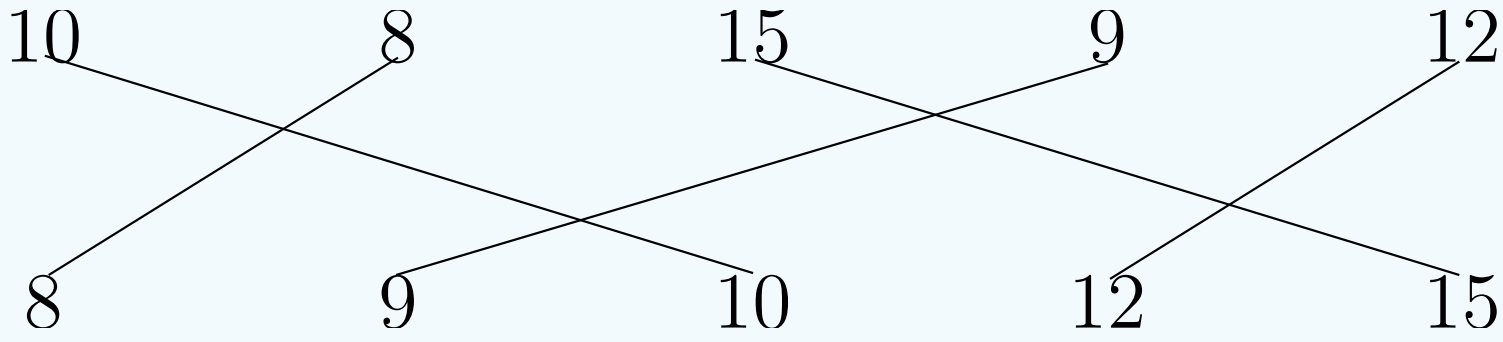
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

## Counting Inversions

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

## Example:




- 4 inversions (for convenience, using numbers, not indices): $(10, 8), (10, 9), (15, 9), (15, 12)$

# Counting Inversions Using Self-Balancing BST

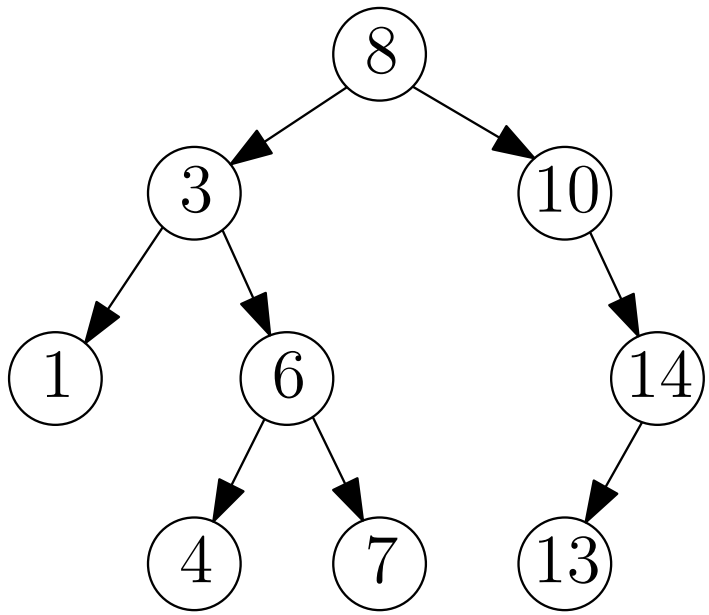
**inversions**$(A, n)$

1: $T \leftarrow$ empty Self-Balancing Binary Search Tree
2: $c \leftarrow 0$
3: **for** $i \leftarrow 1$ to $n$ **do**
4: $\quad c \leftarrow c + i - T$.rank$(A[i])$
5: $\quad T$.insert$(A[i])$
6: **return** $c$

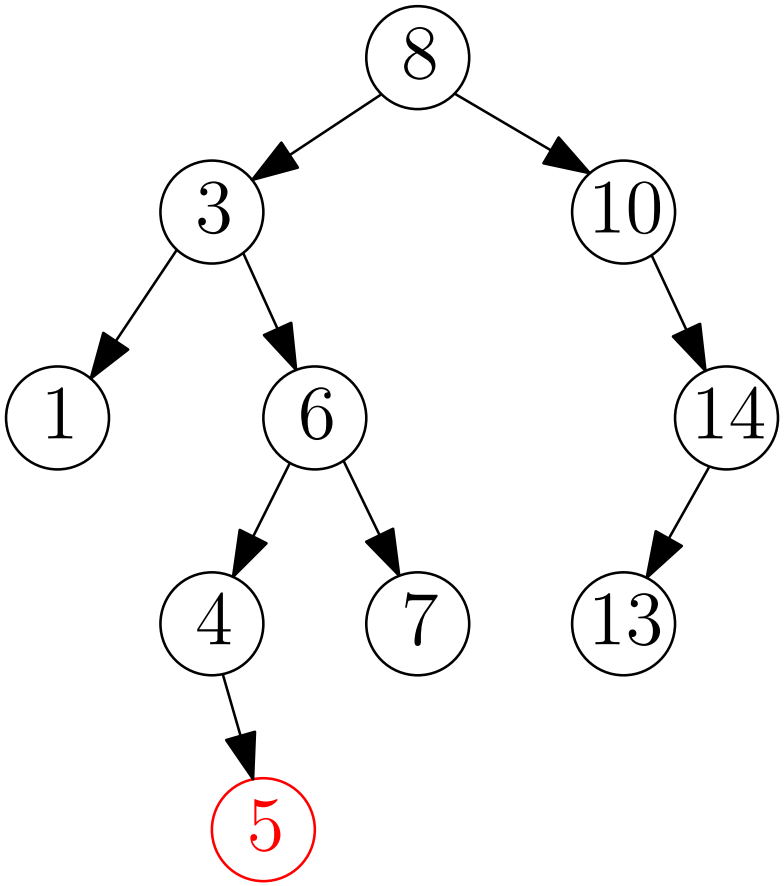
# Binary Search Trees

For any node $v$ in tree:

- key in $v$ must be greater than all keys on the left-sub-tree of $v$
- key in $v$ must be smaller than all keys on the right-sub-tree of $v$
- in-order traversal of tree gives a sorted list of keys



# Binary Search Trees: Insertion



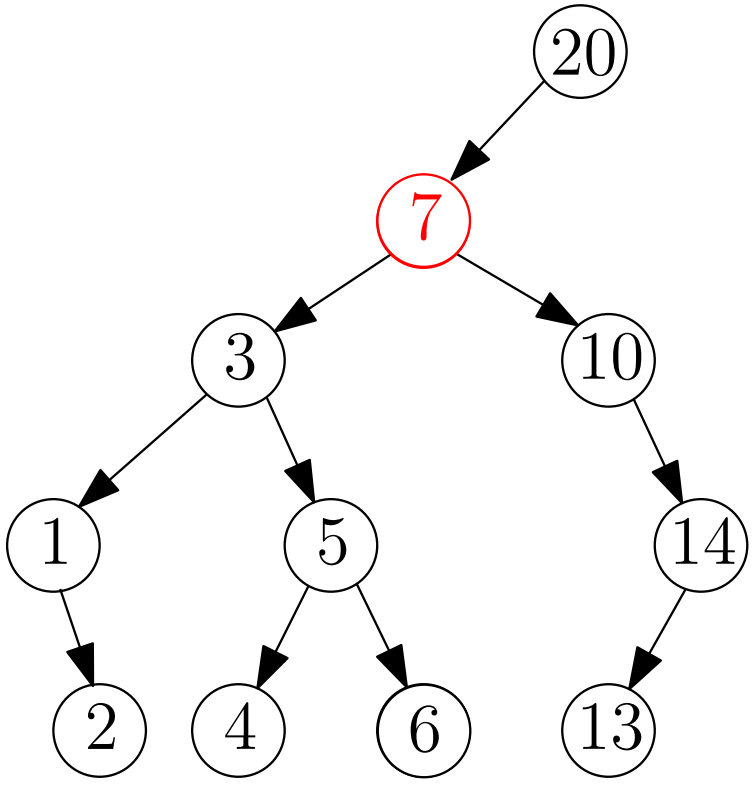
# Binary Search Trees: Insertion

insert$(v, u)$ ▷ $u$ is the new node to be inserted

```
1: if u.key < v.key then
2:     if v.left = nil then v.left ← u
3:     else insert(v.left, key)
4: else
5:     if v.right = nil then v.right ← u
6:     else insert(v.right, key)
```

- Call insert$(root, u)$ if $root \neq nil$, and $root = u$ otherwise.

# Binary Search Trees: Deletion



# Binary Search Trees: Deletion

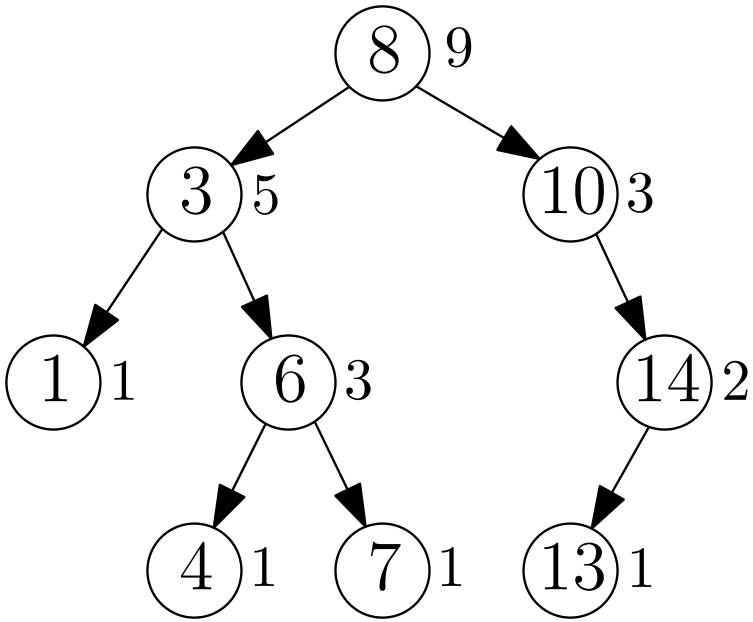
**delete($v$)**

```
 1: if v.left = nil then
 2:     ret ← v.right, remove v, return ret
 3: else
 4:     u ← v.left
 5:     if u.right = nil then
 6:         u.right ← v.right, remove v, return u
 7:     else
 8:         w ← u.right
 9:         while w.right ≠ nil do u ← w, w ← w.right
10:         u.right ← w.left, v.key ← w.key, remove w, return v
```

- procedure returns the new root of the sub-tree after removing $v$
- $v$ is left child of $p$: $p.left \leftarrow$ delete($v$), similar for right child case
- if $v$ is the root: $root =$ delete($v$)

# Rank and Returning $i$-th Smallest Element

- Need to maintain a field "size"




- In both insertion and deletion operations, we need to adjust the size field accordingly.

# Rank and Returning $i$-th Smallest Element

- Engineering trick: let $nil$ be artificial node representing an empty node, with $nil.size = 0$

**rank$(v, key)$**

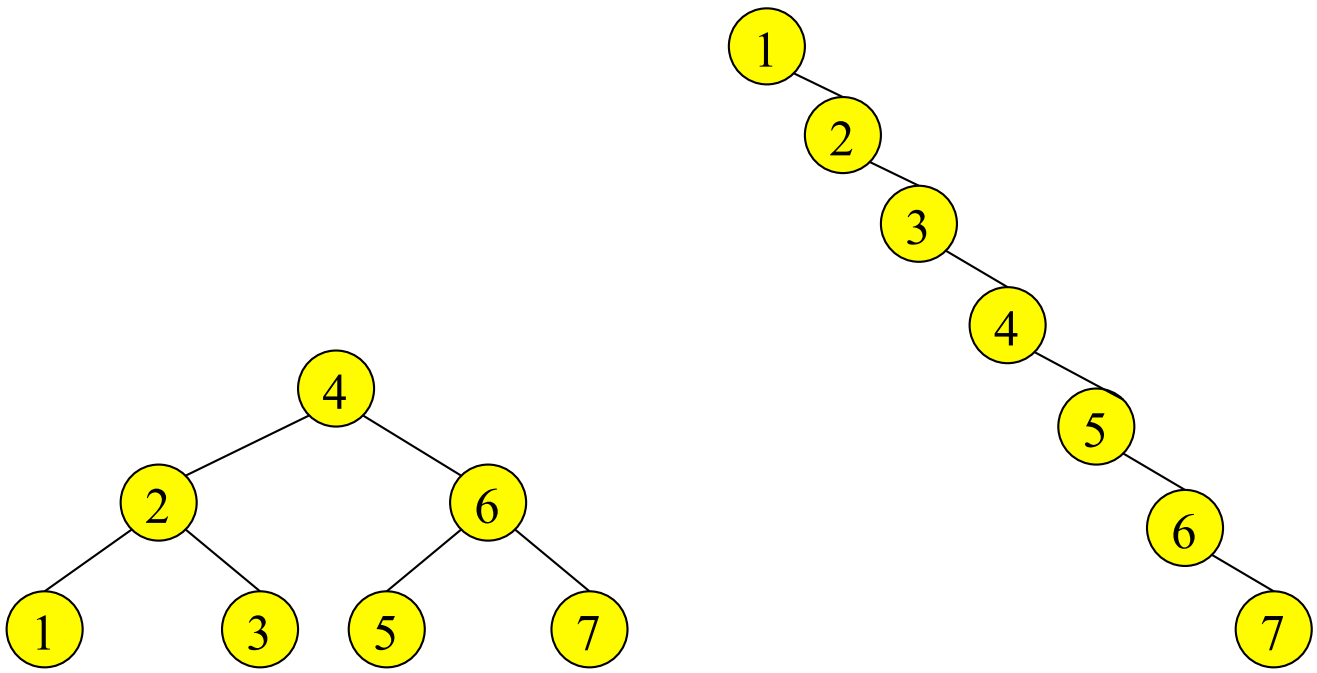
1: **if** $v = nil$ **then return** 0
2: **if** $key < v.key$ **then return** rank$(v.left, key)$
3: **else return** $v.left.size + 1 +$ rank$(v.right, key)$

**selection$(v, i)$**

1: **if** $i \leq v.left.size$ **then return** selection$(v.left, i)$
2: **else if** $i = v.left.size + 1$ **then return** $v.key$
3: **else return** $selection(v.right, i - (v.left.size + 1))$

# Running Time for Operations

- each operation takes time $O(d)$.
- $d =$ depth of tree
- best case: $d = \Theta(\lg n)$
- worst case: $d = \Theta(n)$



Self-Balancing BST: automatically keep the height of tree small

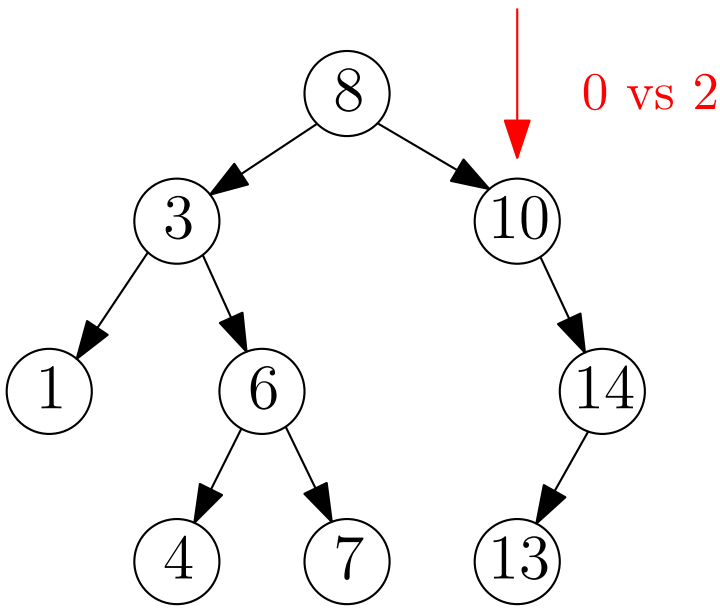
- AVL tree
- red-black tree
- Splay tree
- Treap
- ...

# AVL Tree

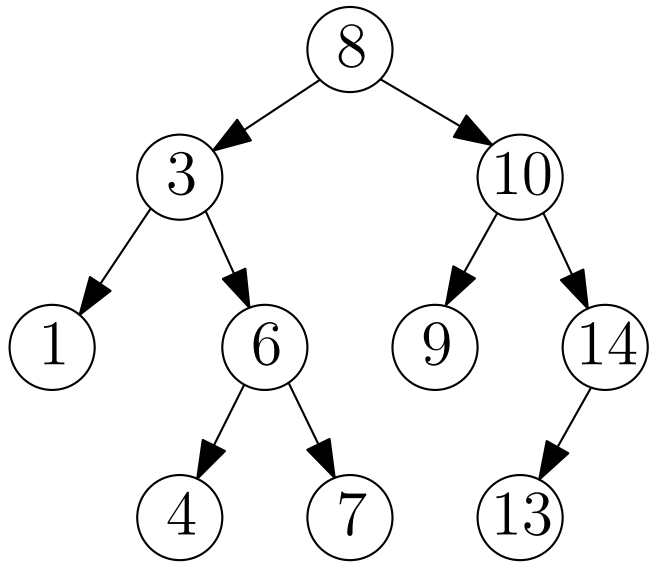
- named after its inventors Adelson-Velsky and Landi.

## Property of an AVL tree

For every node $v$ in the tree, the depths of the left-sub-tree of $v$ and right-sub-tree of $v$ differ by at most 1.




not balanced



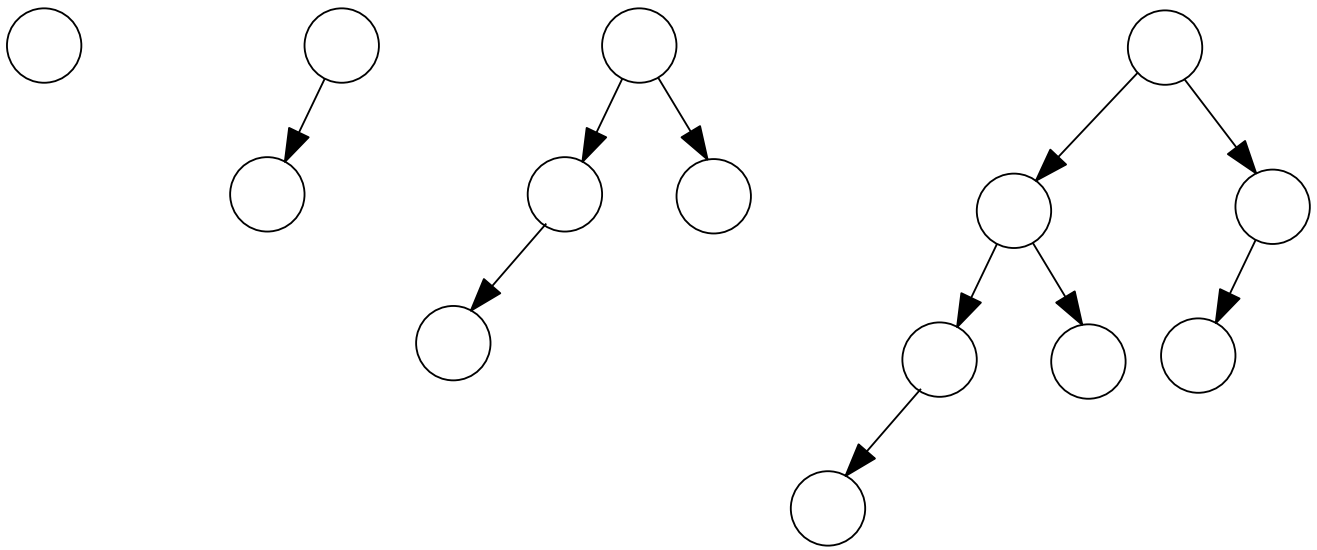

balanced

# AVL Tree

## Property of an AVL tree

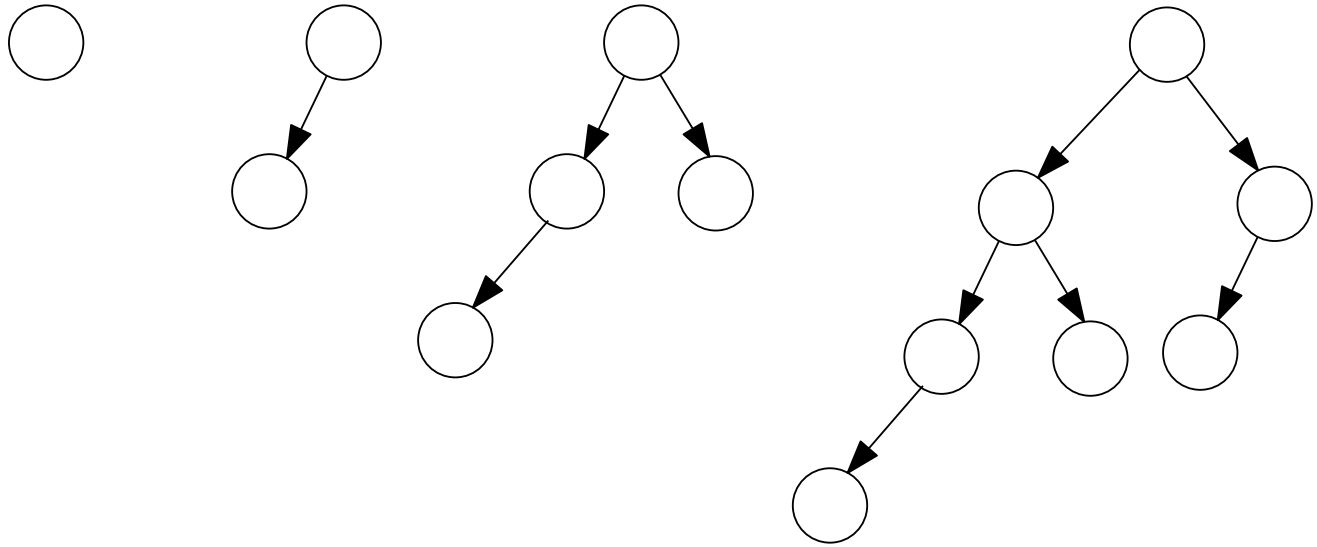
For every node $v$ in the tree, the depths of the left-sub-tree of $v$ and right-sub-tree of $v$ differ by at most 1.

- Why does the property guarantee that the height of a tree is $O(\log n)$?
- $f(d)$: minimum number of nodes in an AVL tree of depth $d$



- $f(0) = 0, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 7 \cdots$

- $f(d)$: minimum number of nodes in an AVL tree of depth $d$



- Recursion:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(d) &= f(d-1) + f(d-2) + 1 \qquad\qquad d \geq 2
\end{aligned}
$$

- $f(d) = 2^{\Theta(d)}$

# Depth of AVL tree

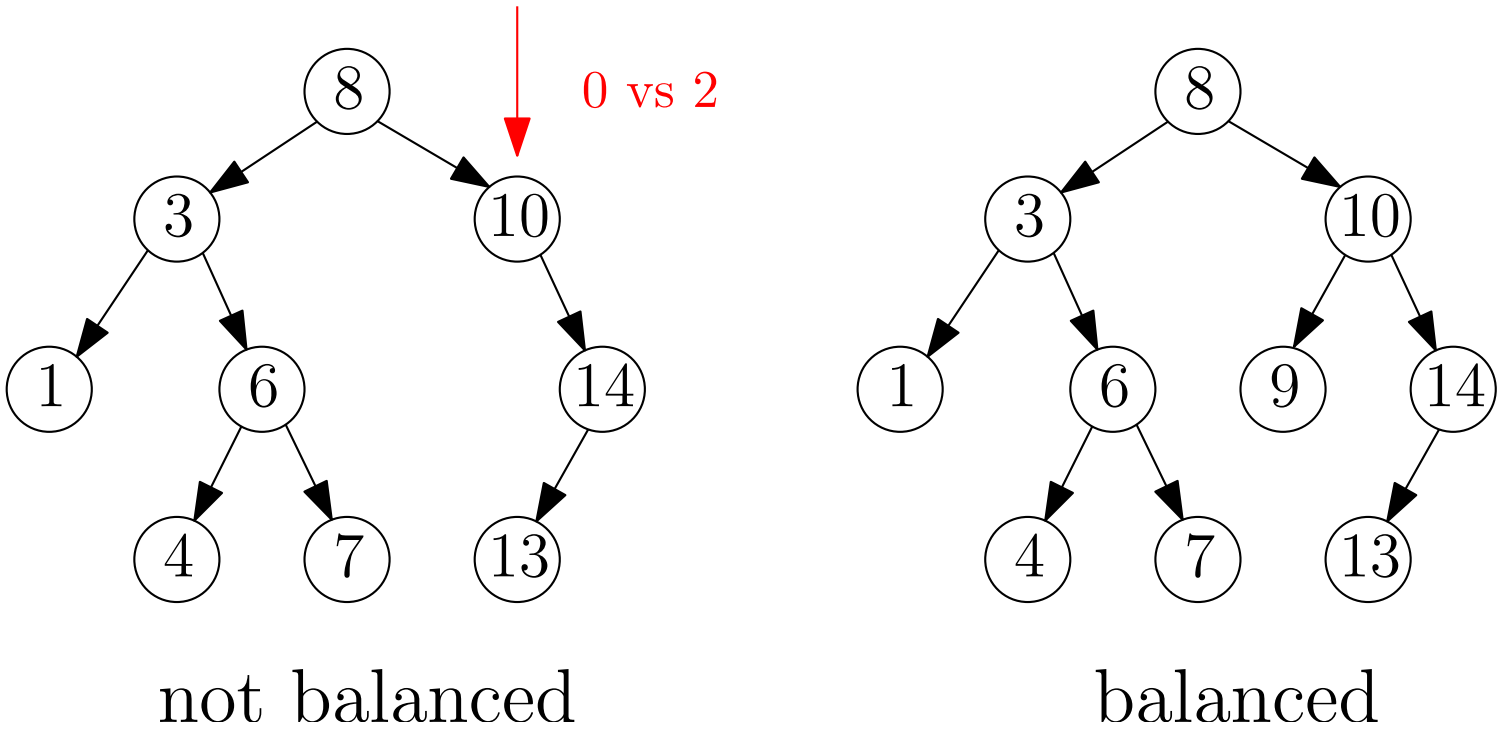
- $f(d)$: minimum number of nodes in an AVL tree of depth $d$
- $f(d) = 2^{\Theta(d)}$
- If a AVL tree has size $n$ and depth $d$, then

$$n \geq f(d)$$

- Thus, $d = O(\log n)$

## Property of an AVL tree

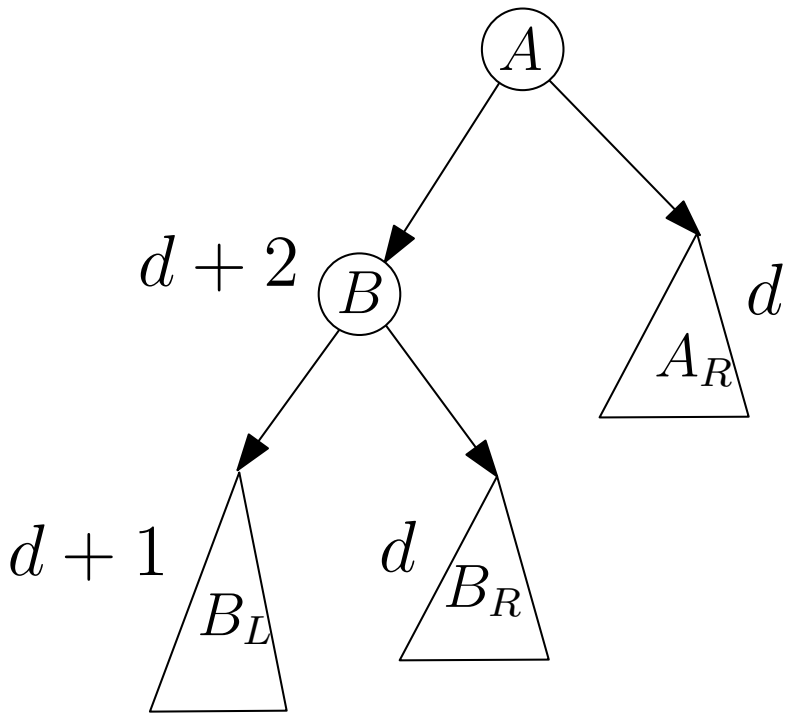
For every node $v$ in the tree, the depths of the left-sub-tree of $v$ and right-sub-tree of $v$ differ by at most 1.

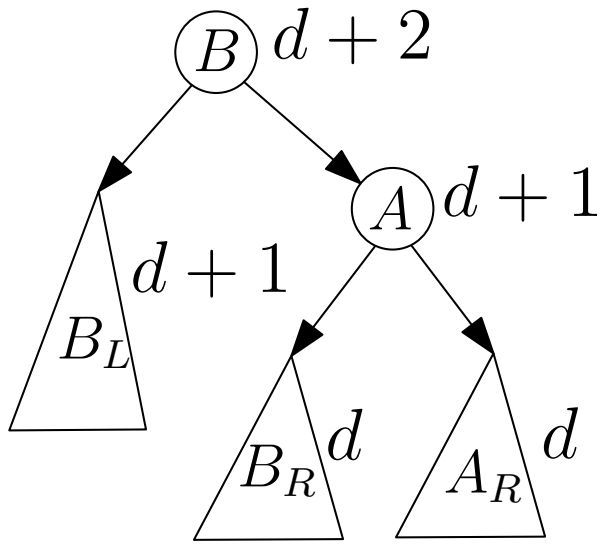



- How can we maintain the property?
- Assume we only do insertions; there are no deletions.

# Maintain Balance Property

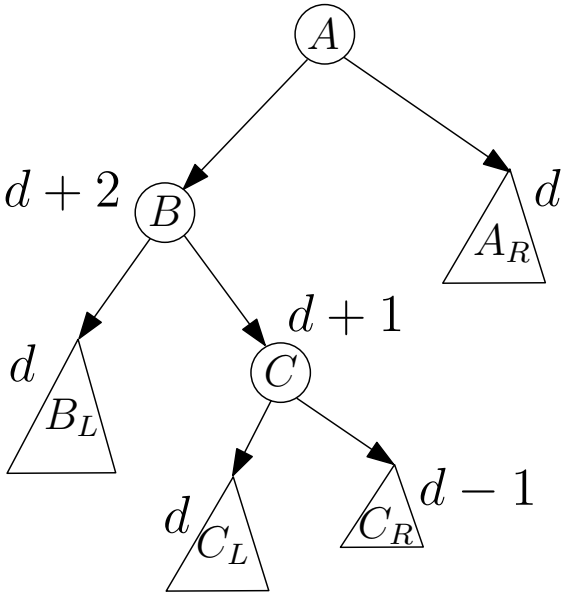
- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 1: we inserted an element to the left-sub-tree of $B$

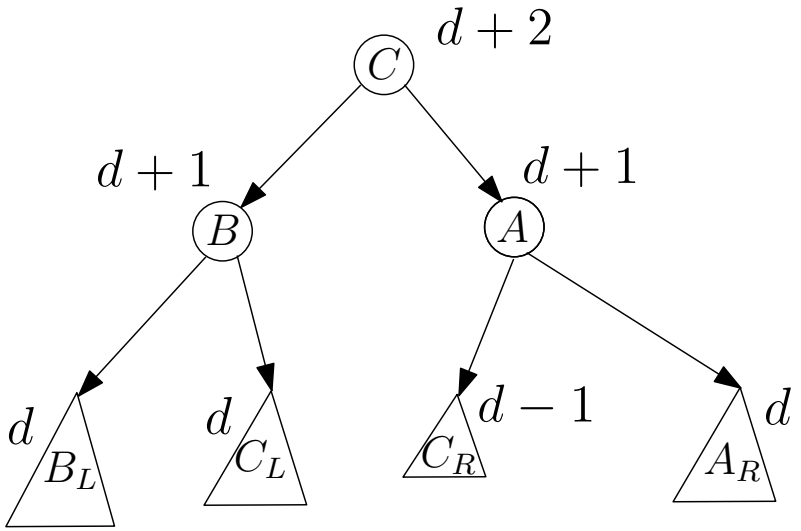


# Maintain Balance Property

- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 2: we inserted an element to the right-sub-tree of $B$
- $C$: the root of right-sub-tree of $B$



# Outline

# Union-Find Data Structure

- $V$: ground set
- We need to maintain a partition of $V$ and support following operations:
  - Check if $u$ and $v$ are in the same set of the partition
  - Merge two sets in partition

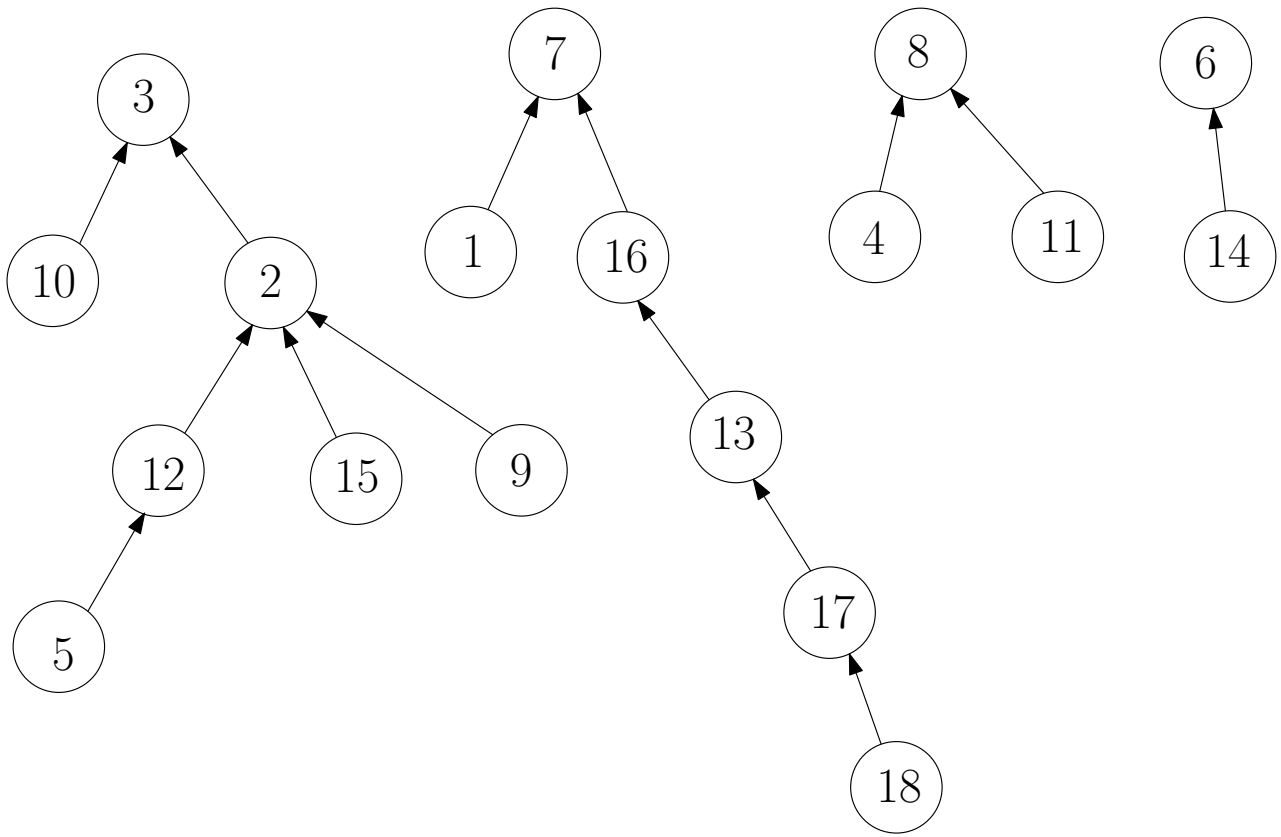
- Algorithm we shall learn Using the data structure: Kruskal's Algorithm for Minimum Spanning Tree

## Example:

- init(6) ▷ {1}, {2}, {3}, {4}, {5}, {6}
- check$(1, 5)$ return **false**
- merge$(2, 3)$ ▷ {1}, {2, 3}, {4}, {5}, {6}
- merge$(1, 6)$, ▷ {1, 6}, {2, 3}, {4}, {5}
- check$(2, 3)$ return **true**
- check$(1, 3)$ return **false**
- merge$(1, 2)$ ▷ {1, 2, 3, 6}, {4}, {5}
- check$(3, 6)$ return **true**
- check$(1, 5)$ return **false**
- merge$(4, 5)$ ▷ {1, 2, 3, 6}, {4, 5}
- merge$(2, 5)$ ▷ {1, 2, 3, 4, 5, 6}
- check$(3, 4)$ return **true**

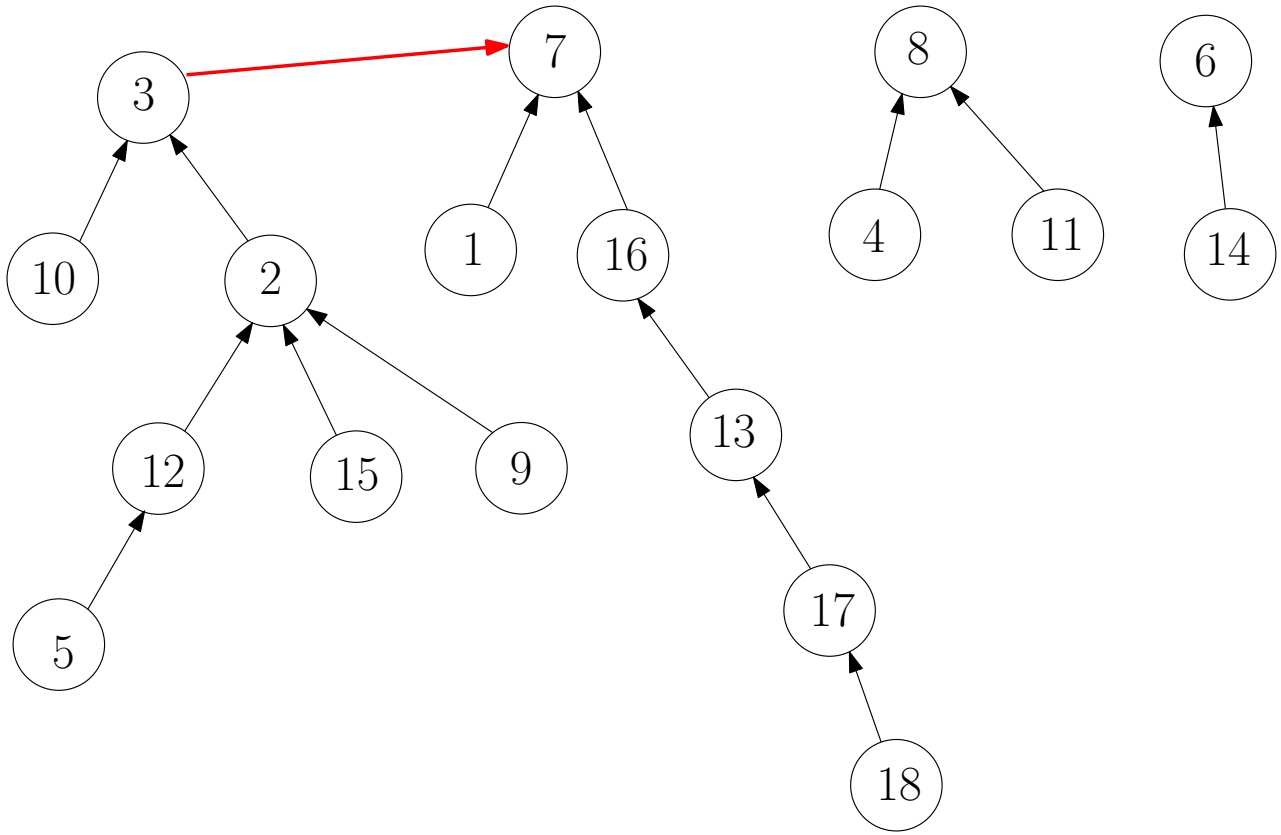
# Using Rooted Trees to Store the Partition

- $V = \{1, 2, 3, \cdots, 16\}$
- Partition: $\{2, 3, 5, 9, 10, 12, 15\}, \{1, 7, 13, 16\}, \{4, 8, 11\}, \{6, 14\}$



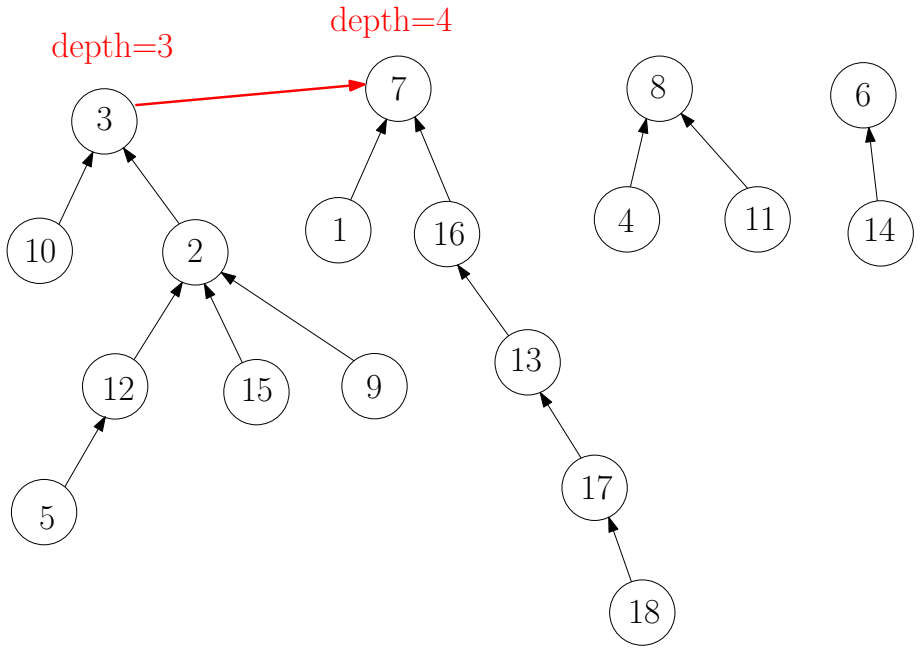

- $par[i]$: parent of $i$, ($par[i] = nil$ if $i$ is a root).

# Union-Find Data Structure




- Q: how can we check if $u$ and $v$ are in the same set?
- A: Check if root$(u) =$ root$(v)$.
- root$(u)$: the root of the tree containing $u$
- Merge the trees with root $r$ and $r'$: $par[r] \leftarrow r'$.

# Union-Find Data Structure




**root($v$)**

```
1: if par[v] = nil then
2:     return v
3: else
4:     return root(par[v])
```

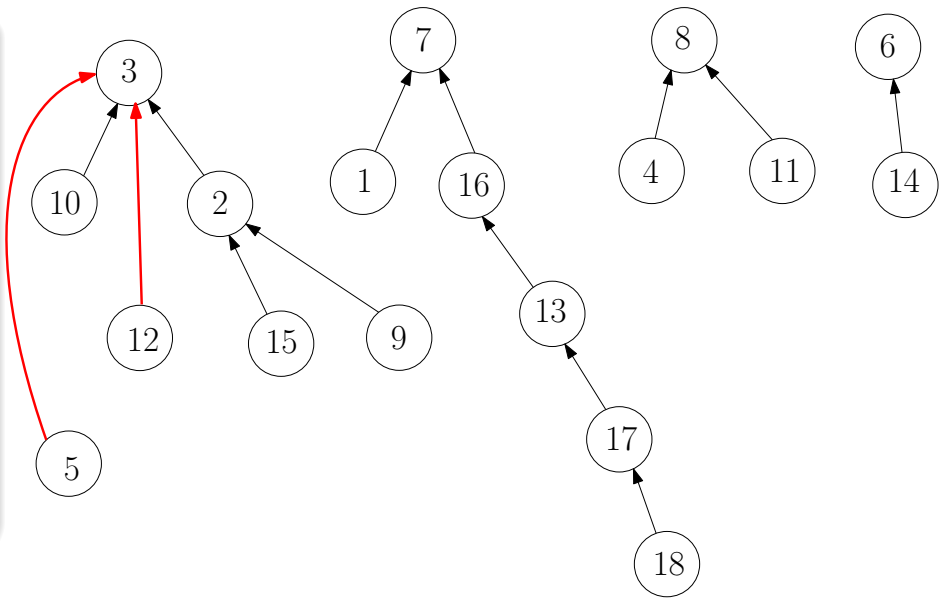
- issue: the tree might be too deep; running time might be large
- ideas for improvements
  - union by size: lighter $\longrightarrow$ heavier
  - union by rank: shallow $\longrightarrow$ deeper
  - path compression

# Union-Find Data Structure: Path Compression

- after calling root$(v)$, directly connecting nodes in path to the root

**root$(v)$**

```
1: if par[v] = nil then
2:     return v
3: else
4:     par[v] ← root(par[v])
5: return par[v]
```



# Time Complexity Using Different Operations

| **Optimization Method** | **Find/Union Operation** |
|---|---|
| No Optimization | $O(n)$ |
| Union by Rank or Size | $O(\log n)$ |
| Path Compression Only | $O(\log n)$ (amortized) |
| Path Compression + Union by Rank | $O(\alpha(n))$ |

- $\alpha(n)$: Inverse Ackermann Function.
- When $n$ is less than the number of atoms in the universe, we have $\alpha(n) \leq 4$. This makes the optimized Union-Find nearly $O(1)$-time in practice.