Lecture 5 - Tree

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

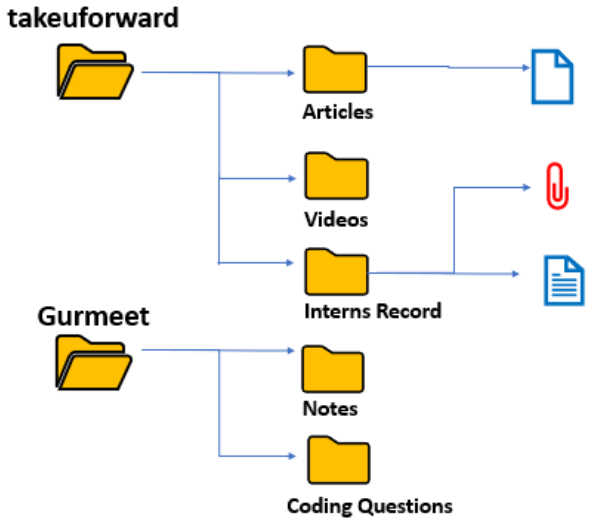
- Trees
- Binary Tree
- Heap
- Heap Sort
- Disjoint Set

Trees

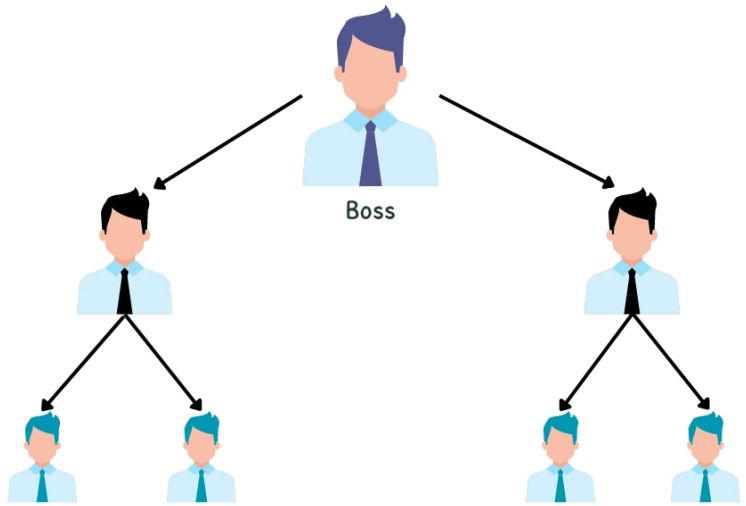
Tree

A hierarchical structure that consists of nodes connected by edges, a special directed graph

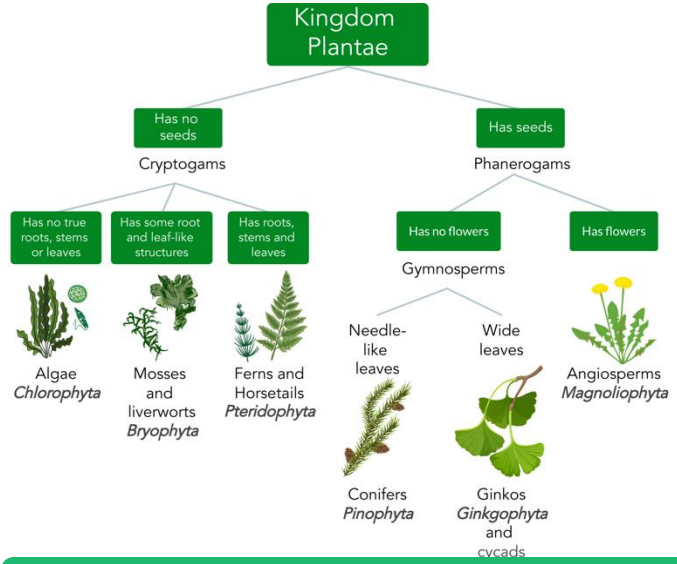
- Each node can be connected to many children, but must be connected to exactly one parent, except for the root node (no parent) – exactly n - 1 edges
- Sometimes we omit the direction



Computer file system



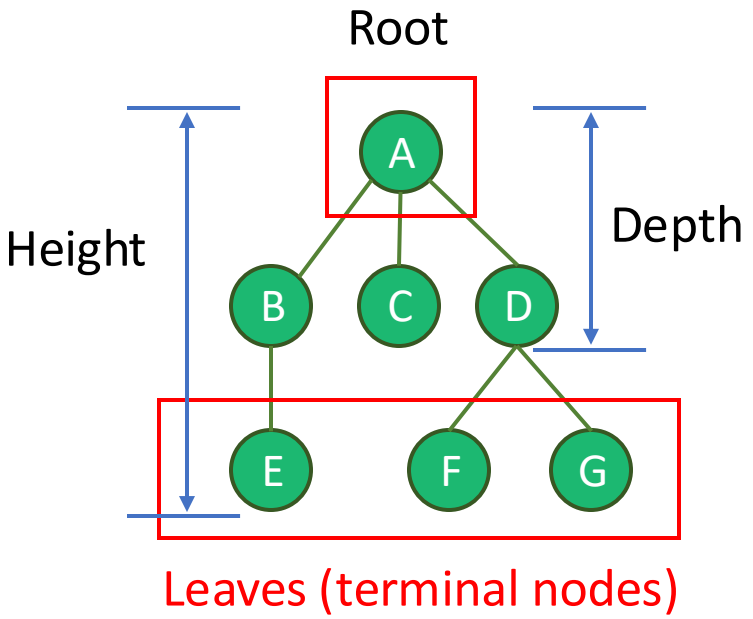
Company organizational structure



Species Taxonomy

Terminologies

- **Parent**: the node directly above a node
- **Children**: nodes directly below a node
- **Siblings**: nodes that share the same parent
- **Leaves (terminal nodes)**: nodes with no children
- **Root**: the node at the top, no parent
- **Height**: maximum distance from root to leaf
- **Depth**: the distance from the root to a node
- **Ancestors**: any node locating on the path to the root
- **Descendant**: all reachable nodes via child edges and any subsequent child edges
- **Subtree**: a smaller tree formed by a node (i.e., the root of the subtree) along with all its descendants

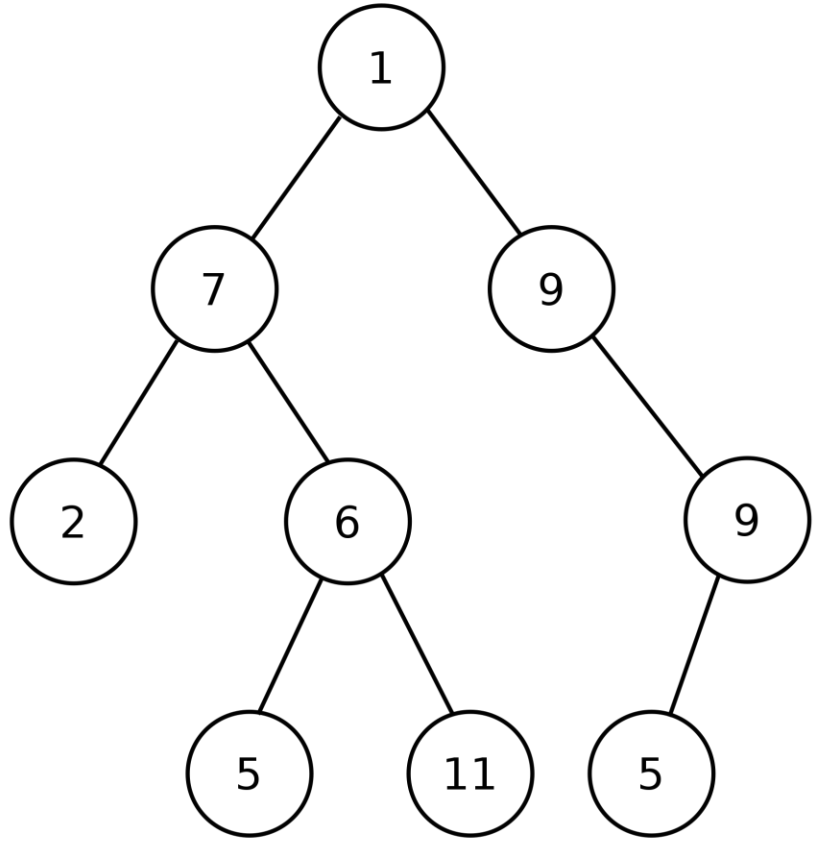


Binary Tree

Binary Tree

A tree data structure where each node has at most 2 children (named left and right child)

- Properties (for a binary tree of height h)
  - The max number of leaves: 2^h
  - The min number of leaves: 1
  - The max number of nodes: $2^{h+1} - 1$
  - The min number of nodes: $h + 1$

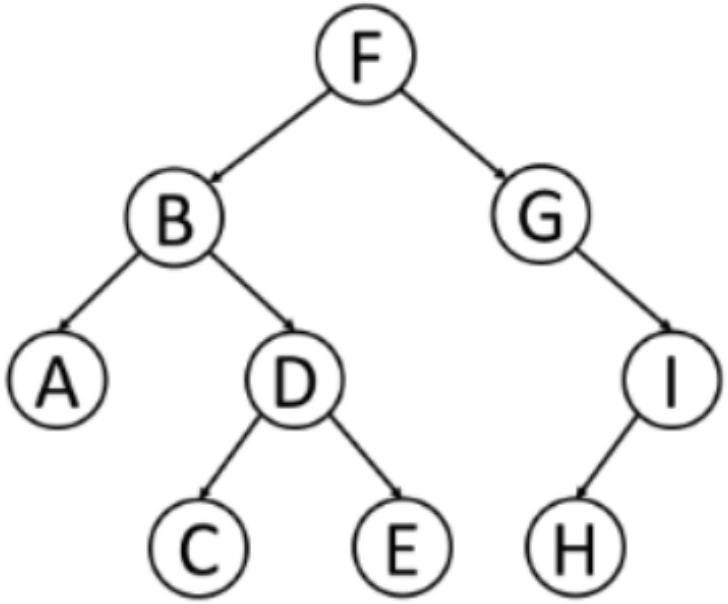


Binary Tree Traversals

Visit all the nodes of a tree in certain order

- Pre-order: visit current node -> traverse left subtree -> traverse right subtree

```
PREORDER(x)
1. if x != NIL
2.       print x.key
3.       PREORDER(x.left)
4.       PREORDER(x.right)
```



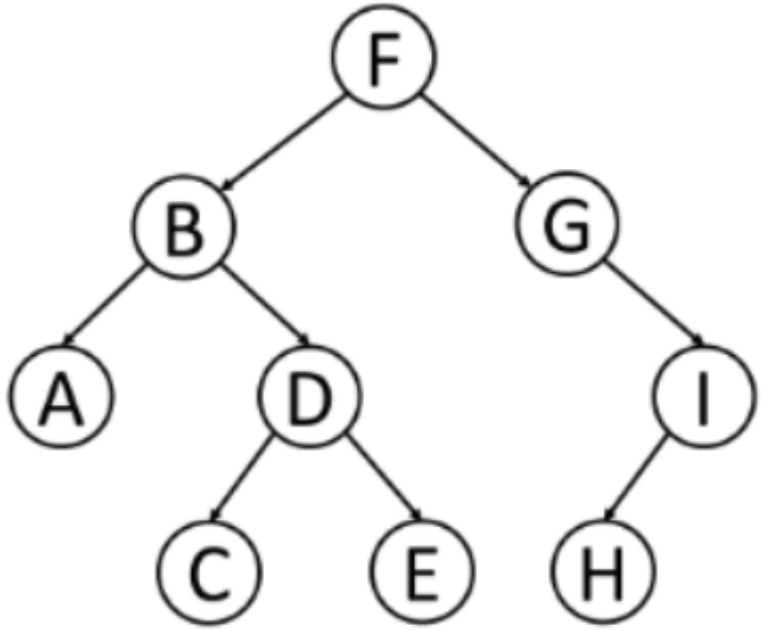
Preorder:

Binary Tree Traversals

Visit all the nodes of a tree in certain order

- In-order: traverse left subtree -> visit current node -> traverse right subtree

```
INORDER(x)
1. if x != NIL
2.      INORDER(x.left)
3.      print x.key
4.      INORDER(x.right)
```



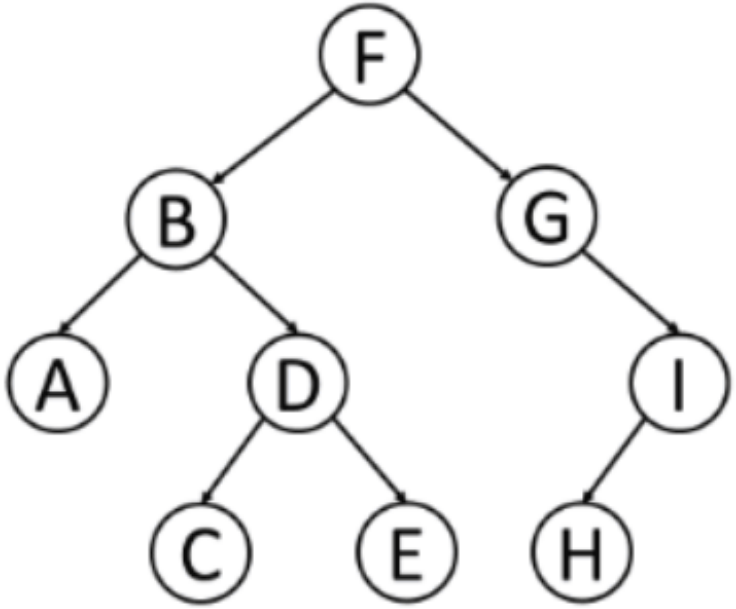
Inorder:

Binary Tree Traversals

Visit all the nodes of a tree in certain order

- Post-order: traverse left subtree -> traverse right subtree -> visit root

```
POSTORDER(x)
1. if x != NIL
2.      POSTORDER(x.left)
3.      POSTORDER(x.right)
4.      print x.key
```

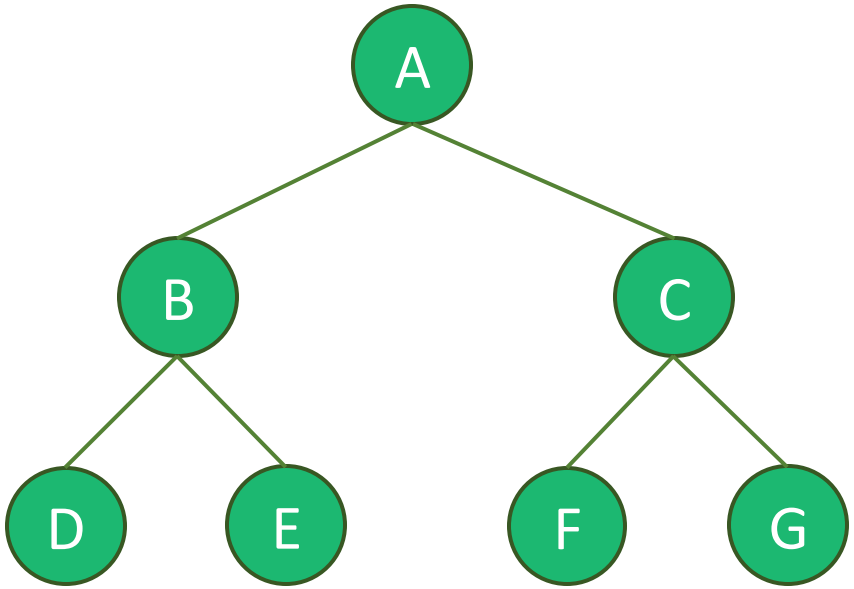


Postorder:

Binary Tree Traversals

Visit all the nodes of a tree in certain order

- Complexity: $T(n) = T(m) + T(n-m) + \Theta(1) = \Theta(n)$
- Practice: what is the sequence of output in pre-order, in-order, post-order traversal of the following tree?



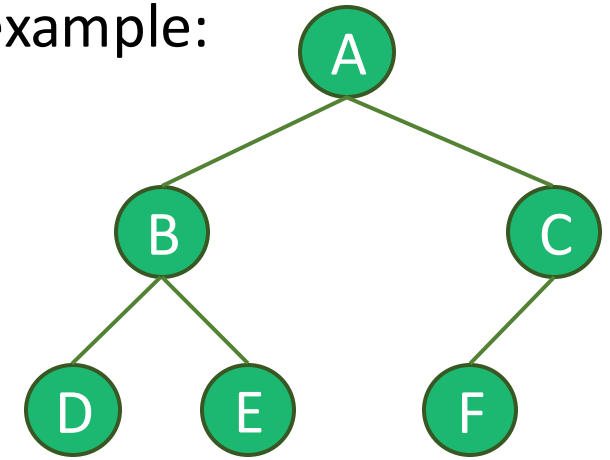

```
PREORDER(x)
1. if x != NIL
2.      print x.key
3.      PREORDER(x.left)
4.      PREORDER(x.right)
```

```
INORDER(x)
1. if x != NIL
2.      INORDER(x.left)
3.      print x.key
4.      INORDER(x.right)
```

```
POSTORDER(x)
1. if x != NIL
2.      POSTORDER(x.left)
3.      POSTORDER(x.right)
4.      print x.key
```

Construct Tree with Traversals

Given inorder and preorder traversal sequence, reconstruct the tree, example:



- Inorder sequence: DBEAFC - [left subtree] [root] [right subtree]
- Preorder sequence: ABDECF - [root] [left subtree] [right subtree]

Observations:

- The first node in preorder sequence is the root - ABDECF
- Root separates inorder sequence into left and right subtrees' inorder sequence - DBEAFC
- The left subtree's inorder sequence is the same length as left subtree's preorder sequence – ABDECF

Solution:

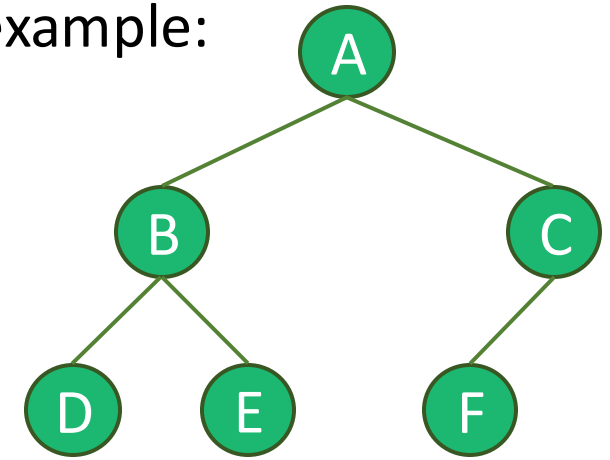
- Divide sequences for subtrees
- Construct the two subtrees

```
construct(S_pre, l_pre, r_pre, S_in, l_in, r_in)
1. if l_pre > r_pre: return NIL
2. root = Node(S_pre[l_pre])
3. m = S_in.find(S_pre[l_pre])
4. sz = m – l_in
5. root.left = construct(S_pre, l_pre+1, l_pre+sz, S_in, l_in, m-1)
6. root.right = construct(S_pre, l_pre+sz+1, r_pre, S_in, m+1, r_in)
7. return root
```

Construct Tree with Traversals

Given inorder and preorder traversal sequence, reconstruct the tree, example:

- Inorder sequence: DBEAFC - [left subtree] [root] [right subtree]
- Preorder sequence: ABDECF - [root] [left subtree] [right subtree]



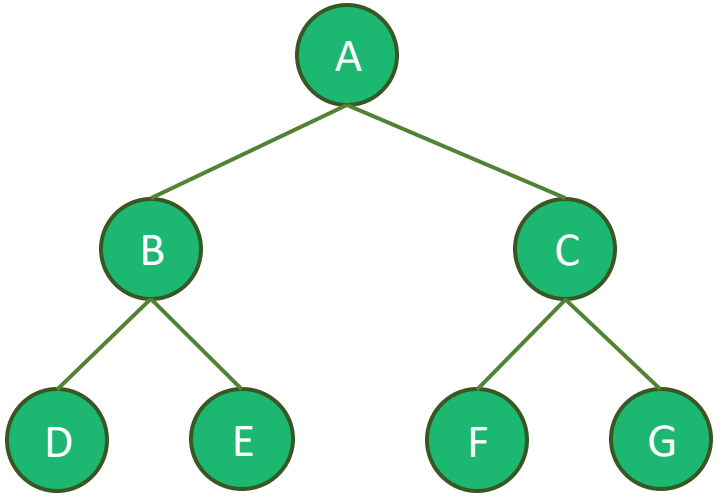
How to *find* the root in the inorder sequence?

- Naïve: compare 1-by-1, $\Theta(n)$ find
  - $T(n) = T(m) + T(n-m-1) + \Theta(n)$
  - $T(n) = \Theta(nlogn)$
- Direct addressing table in advance: $\Theta(1)$ find
  - $T(n) = T(m) + T(n-m-1) + \Theta(1)$
  - $T(n) = \Theta(n)$

Types of Binary Trees

Perfect binary tree: all interior nodes have exactly two children, and all leaves are at the same level

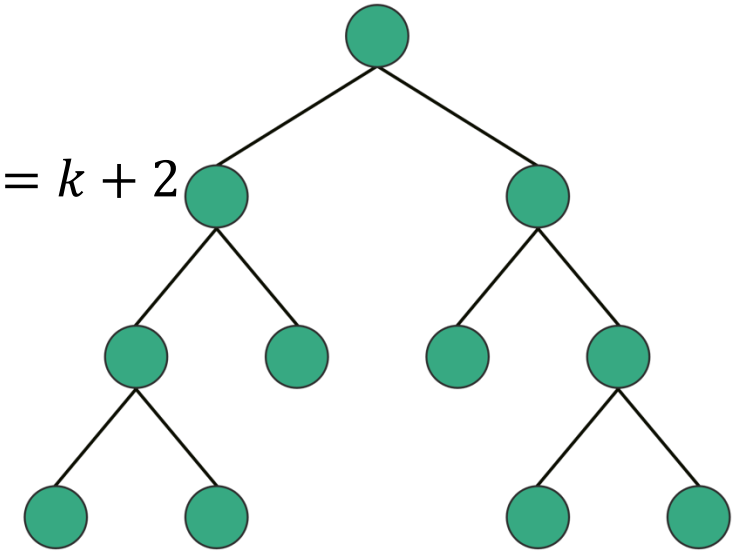
- Property:
  - Level i has 2^i nodes (prove with mathematical induction)
  - A perfect binary tree of height h has $2^{h+1} - 1$ nodes $(2^0 + 2^1 + 2^2 + \cdots + 2^h)$
  - Half of the total nodes $(2^h - 1)$ are interior nodes $(2^0 + 2^1 + 2^2 + \cdots + 2^{h-1})$
  - 2^h leaves



Types of Binary Trees

Full binary tree: all the interior nodes have 2 children

- A perfect binary tree is always a full binary tree
- Property: let i be the number of interior nodes, the number of leaves is i + 1
- Proof (mathematical induction)
  - Base case: when $\#interior = 1, \#leaves = 2$
  - Inductive step: supposing when $\#interior = k, \#leaves = k + 1$
    - Only adding two child nodes to a leaf make $\#interior = k + 1$
    - i.e., change 1 leaf to interior node and add two leaves, $\#leaves = k + 2$
  - Therefore, when $\#interior = i, \#leaves = i + 1$ for any $i \geq 0$



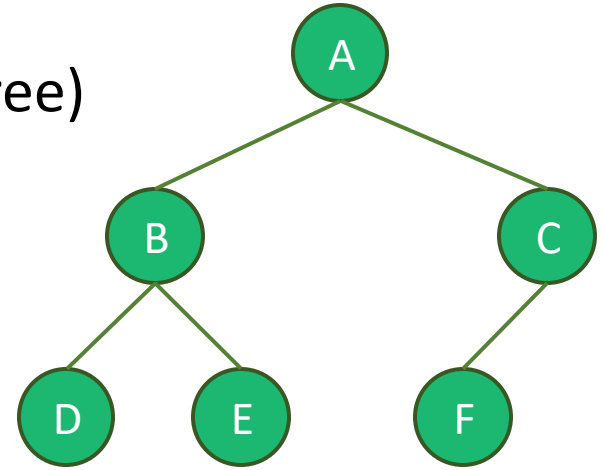
Types of Binary Trees

Complete binary tree: all the levels are filled completely except the lowest level nodes which are filled from as left as possible

A perfect binary tree is always a complete binary tree

Properties:

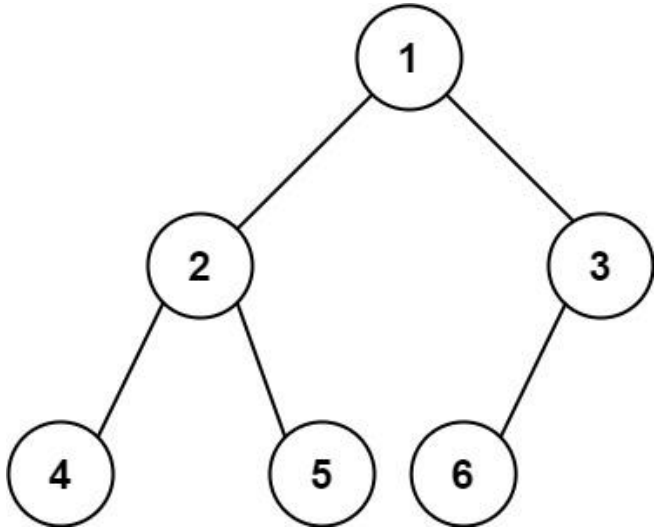
- Level $i \leq h - 1$ has 2^i nodes
- Has 2^h to $2^{h+1} - 1$ nodes in total
- 2^{h-1} (one node in the last layer) to 2^h leaves (perfect binary tree)



Count Complete Binary Tree Nodes

Problem: Given the root of a complete binary tree, output the number of the node.

- Naïve solution: binary tree traversal
  - $\#nodes = \#nodes\_in\_left\_subtree + \#nodes\_in\_right\_subtree + 1$
  - $f(root) = f(left\_subtree) + f(right\_subtree) + 1$
  - $\Theta(n)$, traverse each node once
- Better solution? ---- Hint: complete binary tree



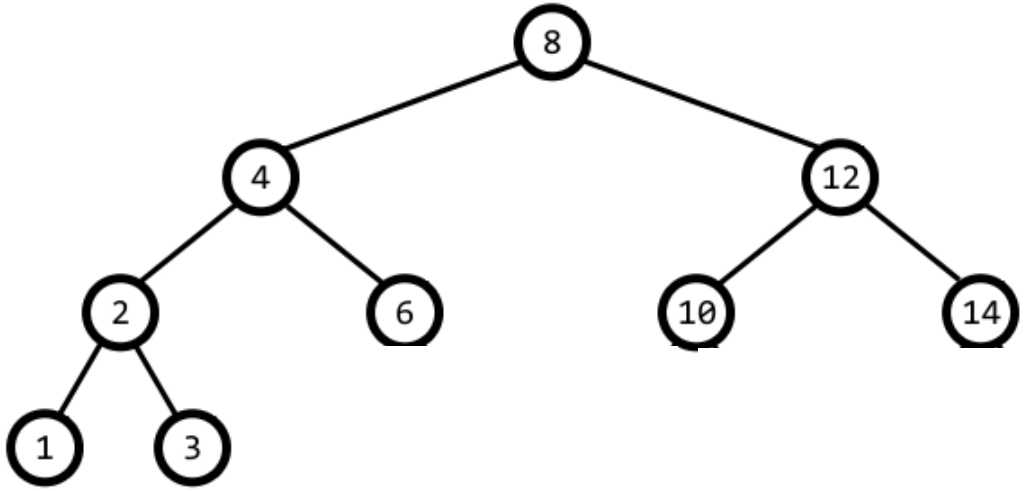
Count Complete Binary Tree Nodes

Given the root of a complete binary tree, output the number of the node.

- $f(root) = f(left\_subtree) + f(right\_subtree) + 1$

Observation: both left and right subtree are complete binary tree and at least one of them is a perfect binary tree

- Perfect binary tree of depth h has $2^{h+1} - 1$ nodes



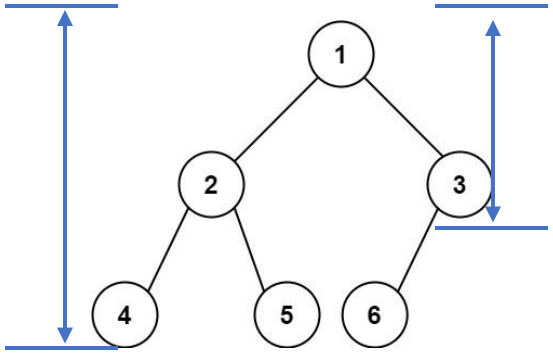
- Both perfect
- Left is perfect
- Right is perfect

Solution: check whether the current tree is perfect before recursion!

Count Complete Binary Tree Nodes

Problem: Given the root of a complete binary tree, output the number of the node

- Solution: check whether the current tree is perfect before recursion!
- **Check perfection:** equal distance of leftmost path and rightmost path
  - $d_l(T) = d_l(T.left) + 1$ ---- tell the height of tree
  - $d_r(T) = d_r(T.right) + 1$ ---- tell whether the node is absent
  - Time complexity: $\Theta(h)$




```
function findLeftHeight(node):
    height = 0
    while node is not null:
        height += 1
        node = node.left
    return height
```

```
function findRightHeight(node):
    height = 0
    while node is not null:
        height += 1
        node = node.right
    return height
```

Count Complete Binary Tree Nodes

Problem: Given the root of a complete binary tree, output the number of the node

- Solution: check whether the current tree is perfect before recursion!
- **Check perfection:** equal distance of leftmost path and rightmost path

```
function countNodes(root):
    if root is null:
        return 0

    leftHeight = findLeftHeight(root)
    rightHeight = findRightHeight(root)

    if leftHeight == rightHeight:
        return (2^(leftHeight+1)) - 1
    else:
        return 1 + countNodes(root.left)
                 + countNodes(root.right)
```

```
function findLeftHeight(node):
    height = 0
    while node is not null:
        height += 1
        node = node.left
    return height

function findRightHeight(node):
    height = 0
    while node is not null:
        height += 1
        node = node.right
    return height
```

Count Complete Binary Tree Nodes

Problem: Given the root of a complete binary tree, output the number of the node

- Solution: check whether the current tree is perfect before recursion!
- **Check perfection:** equal distance of leftmost path and rightmost path
- Evaluate time complexity T(h):
  - The two sub-trees must have one perfect binary tree (1 recursive subtree, 1 non-recursive subtree with $\Theta(h)$, $\Theta(h)$ for judging its own perfectness):

$$T(h) = T(h-1) + \Theta(h) = \Theta(h^2)$$

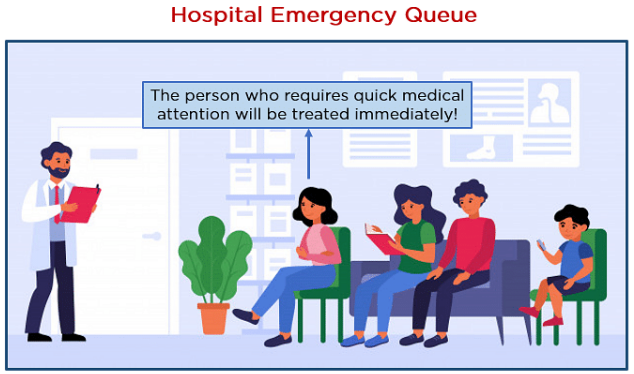
  - $h = \Theta(\log n)$ in a complete binary tree

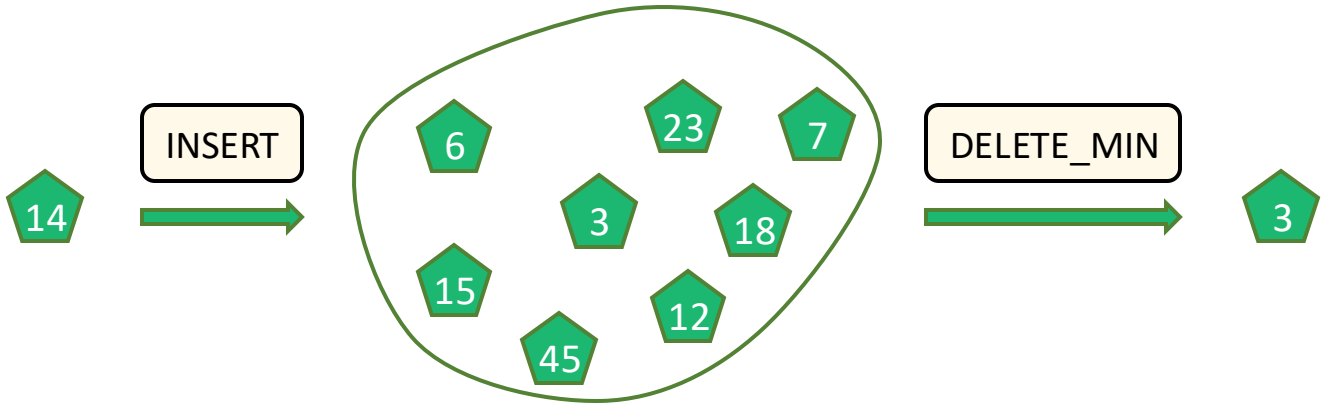
$$T(n) = \Theta(\log^2 n)$$

Heap

ADT: Priority Queue

- A dynamic collection where elements have associated *priority*
  - Elements with higher priority are served first (not FIFO and LIFO)
- **Operations**:
  - Pop the smallest element (i.e., largest priority)
  - Insert an element





Heap

- A tree-based data structure that satisfies the ***heap property***:
  - Each node has higher or equal priority than its children
  - ⇒ Recursively, grand children
  - ⇒ The root always has the highest priority
- E.g., min heap: each node has smaller key than the children's
- Common implementation: complete binary tree-based
  - Advantage 1: low height – $O(\log n)$
  - Advantage 2: can be simulated with array

Pop Minimum

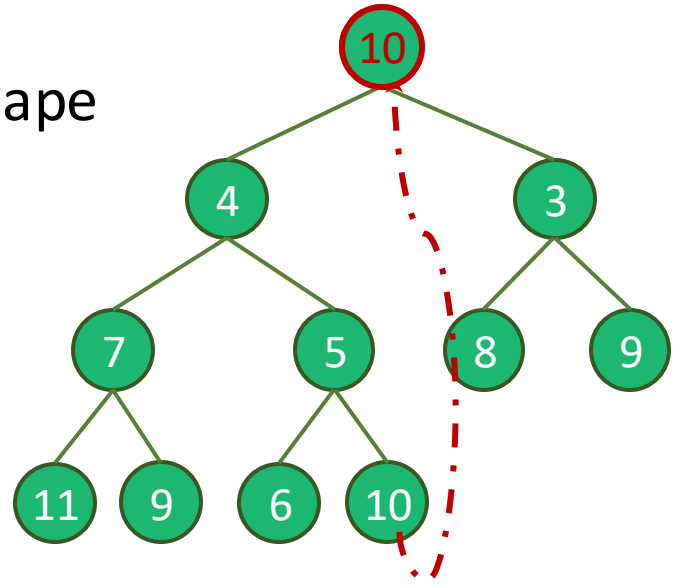
Return the root node and remove it from the heap

Problem: replace the root with another node

- Requirement: (1) heap property (2) complete binary tree property
- Recursively move the child node to the root position?
  - Not complete binary tree
- Hint: a complete binary tree with fix number of nodes has fixed shape

Solution: move the last element to the root position

- **How to Restore Heap Property?**

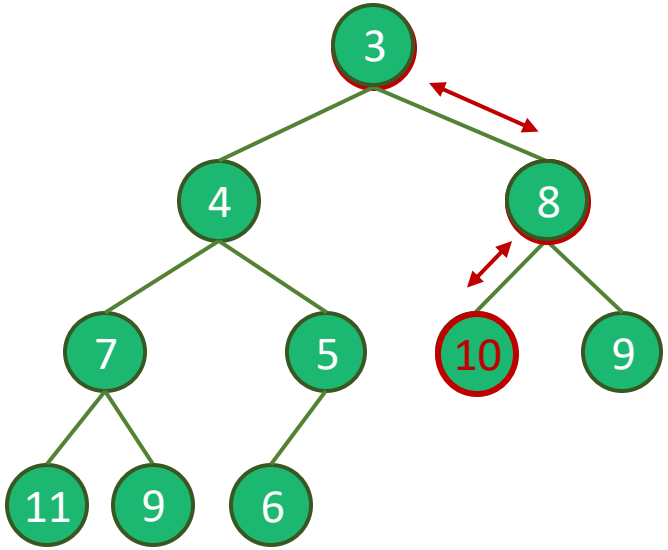


Pop Minimum

Restore Heap Property: Percolate Down

Core idea: switch the nodes

- Compare the modified node's priority with its children
  - Higher: current subtree is a heap, stop
  - Lower: swap with the child with highest priority
- Repeat until current subtree is a heap or reach a leaf node
- Complexity: $\Theta(h) \Rightarrow \Theta(\log n)$ for being complete binary tree

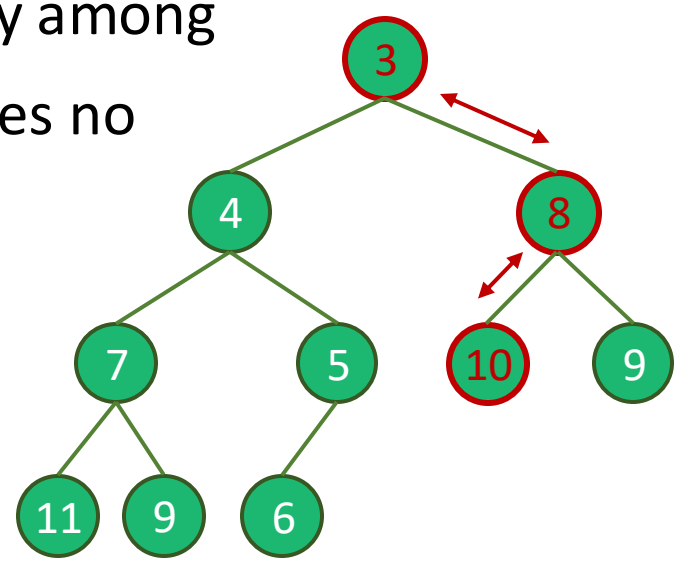


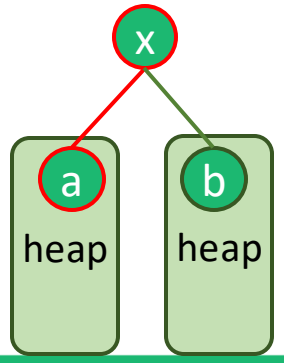
Pop Minimum

Restore Heap Property: Percolate Down

- Compare the modified node's priority with its children
- Swap with the child with highest priority
- Repeat until current subtree is a heap or reached a leaf node

Heap Property: During each swap, the node with the highest priority among the descendants is moved up (like merge sort's merging). This ensures no child has a higher priority than the current node after the swap

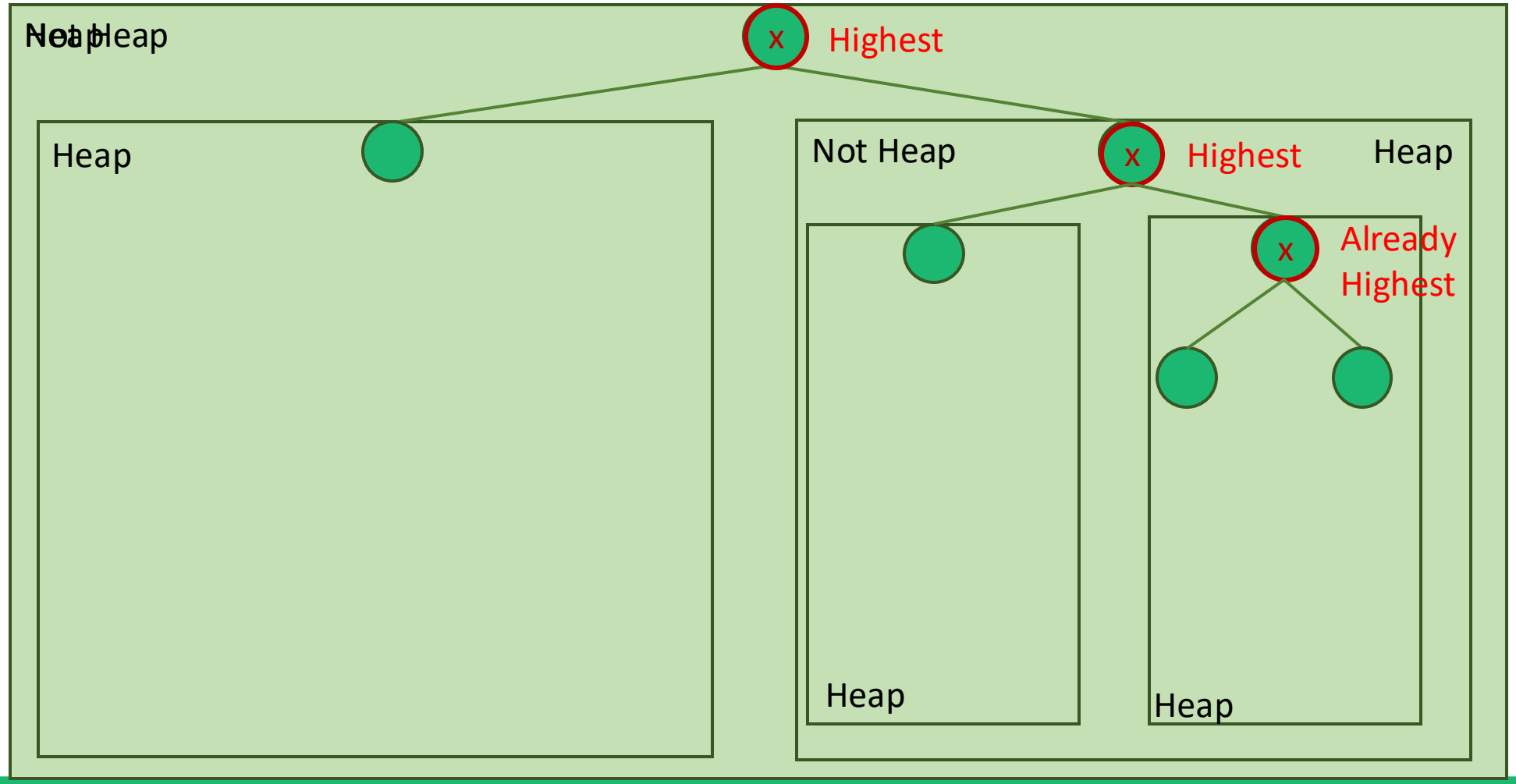




- The highest priority must between a and b (top of their sub_heap)

Pop Minimum

Swapping procedure - supposing always swap with the right child



Insert Element

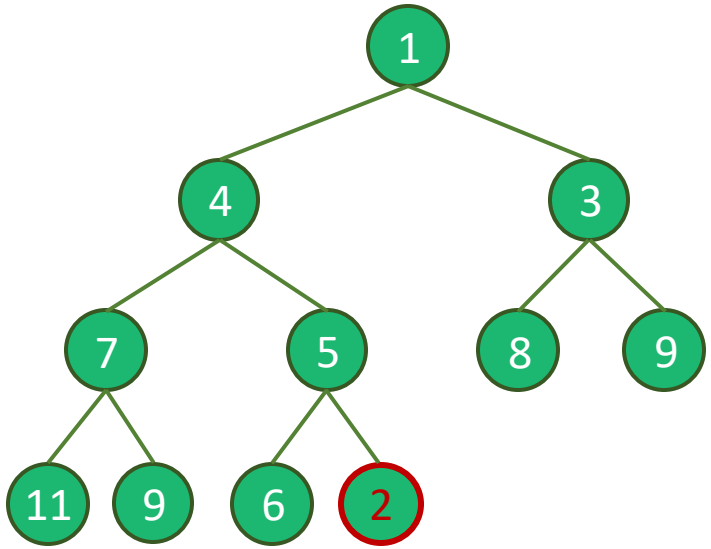
Add a node to the tree

Problem: where to put the node

- Requirement: (1) heap property (2) complete binary tree property
- Hint: a complete binary tree with fix number of nodes has fixed shape

Solution: append to the end of the complete binary tree

- How to restore Heap Property?



Insert Element

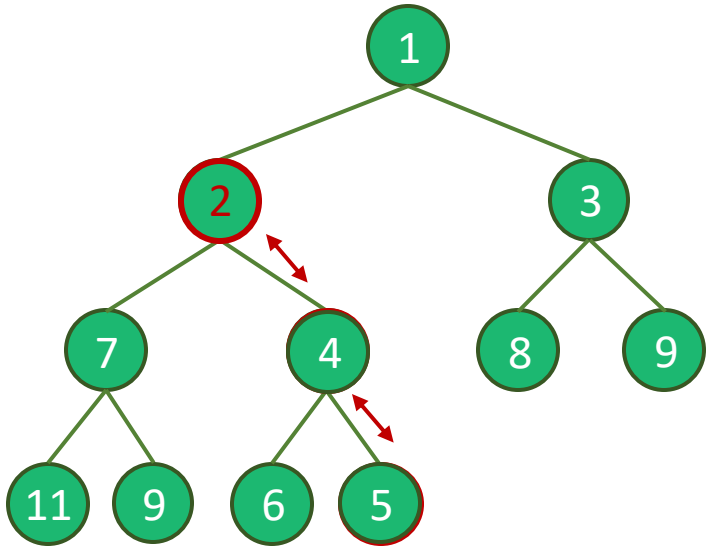
Restore heap property: Percolate Up

- Compare the modified node's priority with its parent
  - Lower: the parent's subtree is a heap, stop
  - Higher: swap with parent
- Repeat until parent has higher priority or reached root
- Complexity:$\Theta(h) \Rightarrow \Theta(\log n)$ for being complete binary tree

Proof of correctness:

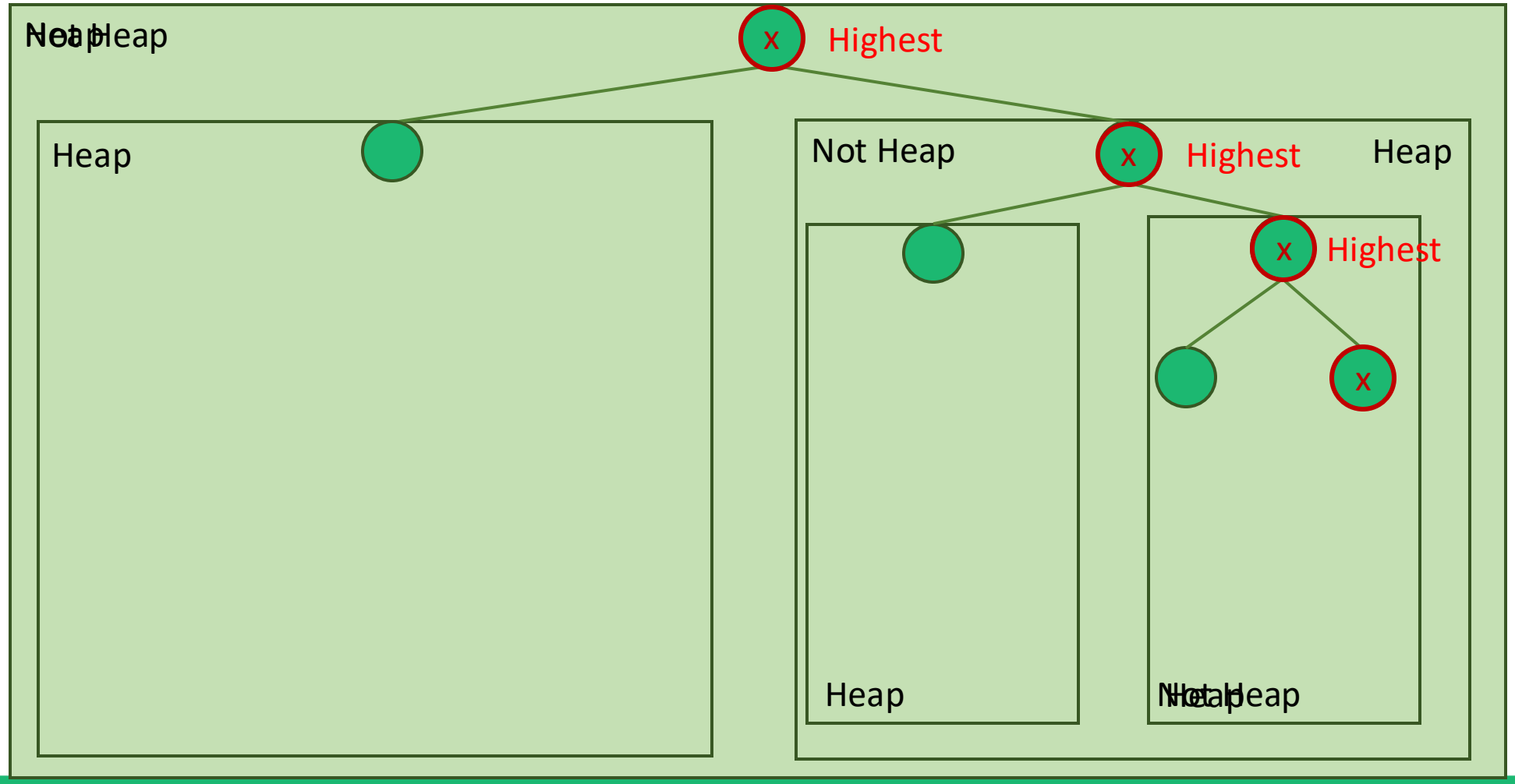
- Initially added a leaf node, a root for a heap subtree
- Swap: since the node exceeds its parent in priority and the parent dominates its subtree, after swapping, the node still roots a valid heap subtree.

Notice: If items arrive in random order, then insert is $\mathcal{O}(1)$ on average



Insert Element

Swapping procedure



Summary of Operations

- Overall Requirement
  - Preserve structure property
  - Restore heap property
- insert: append at new last position, percolate-up $\Theta(\log n)$, on average $\mathcal{O}(1)$
- pop: remove root, put last element at root, percolate-down $\Theta(\log n)$

Heap Sort

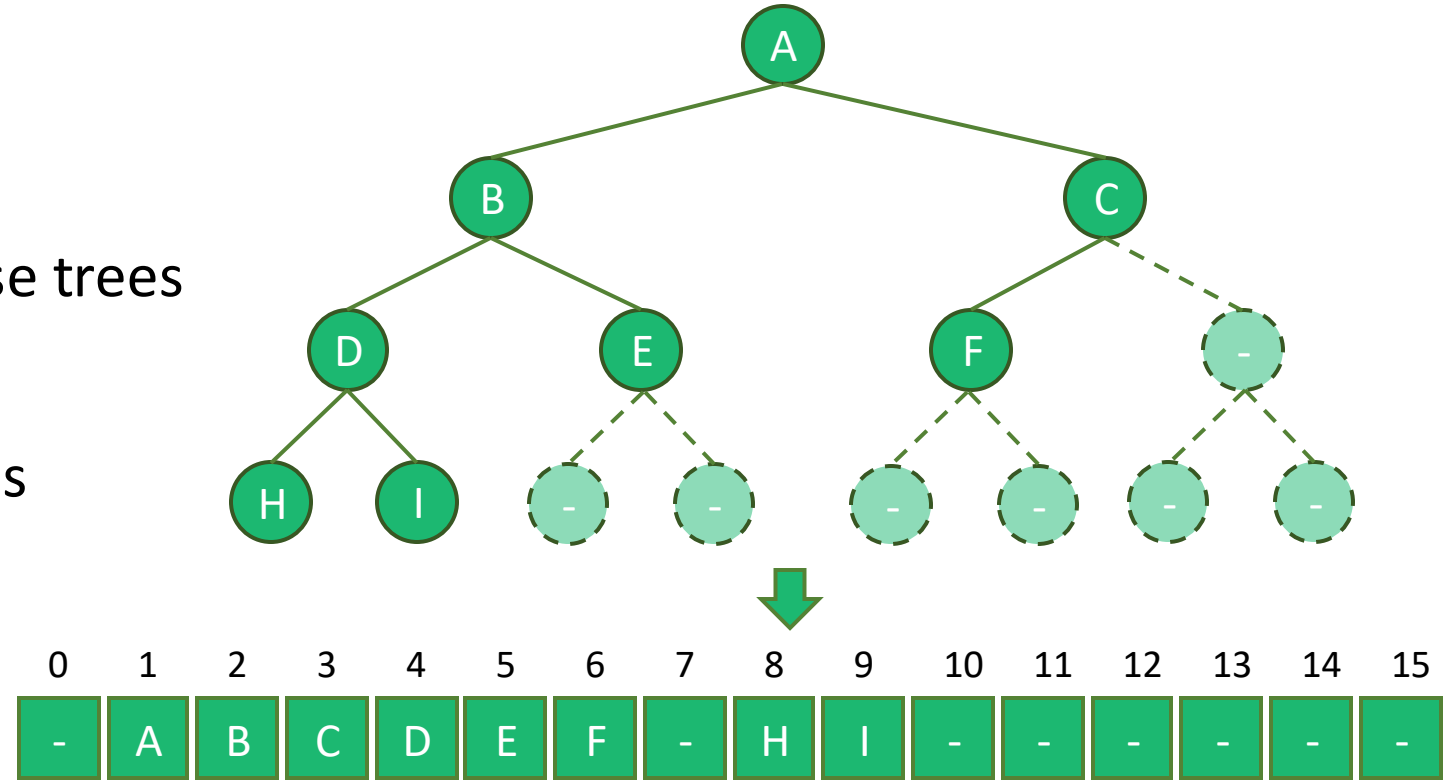
Heap Sort

Use heap to sort elements in an array

- Add elements in the array into a min-heap
  - $\mathrm{O}(1)$ for each element on average if add randomly
  - Add n elements: $\mathrm{O}(n)$
- Pop the root and restore the min-heap one by one
  - $\mathrm{O}(\log n)$ for popping each element
  - Pop n elements: $\mathrm{O}(n\log n)$
- Auxiliary space complexity: $\Theta(n)$
- *Question: can we achieve inplace sorting with this idea?*

Array-based Binary Tree Representation

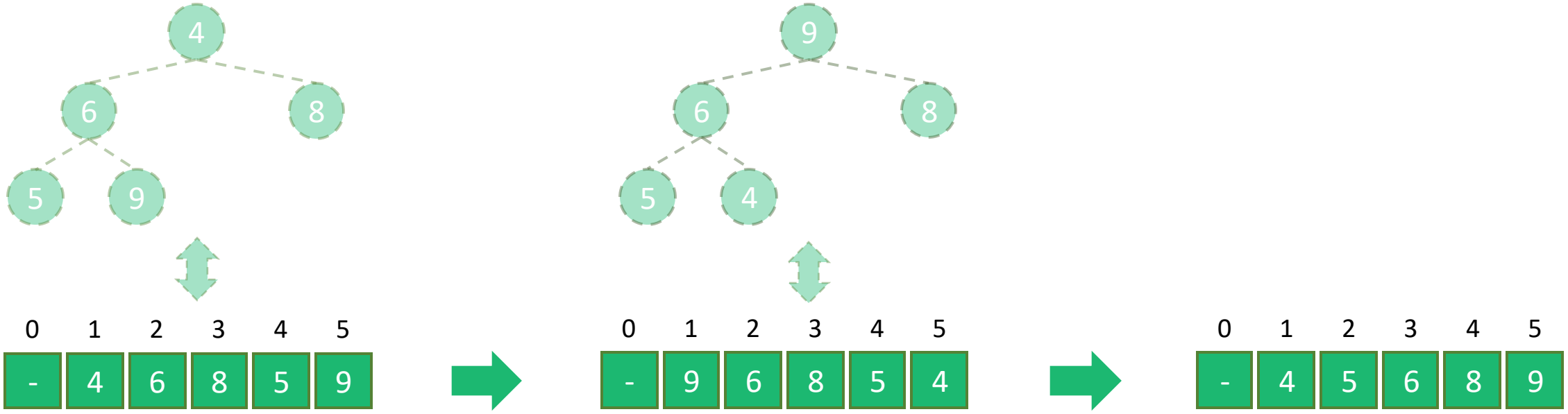
- Place root node in index 1 (wasting index 0 brings convenience for index arithmetic)
- For node with index i:
  - Left child: $i * 2$
  - Right child: $i * 2 + 1$
  - Parent: $\lfloor i/2 \rfloor$
- Problem: waste of memory for sparse trees
  - e.g., chain
- Complete binary tree: no empty slots



Heap Sort

Core idea for inplace heap sort:

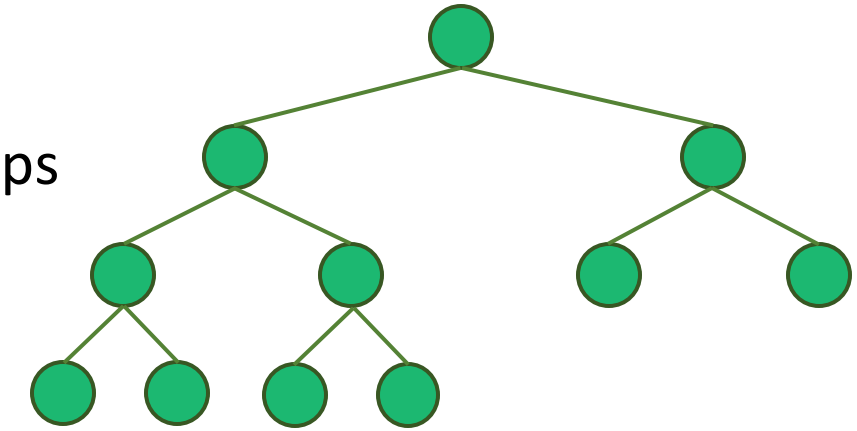
- Regard the input array a complete binary tree
- Adjust the array to make a max-heap
- Pop the highest and add it from the tail of the array



Heap Sort

Adjust the array into a heap

Key idea: Given a tree where the left and right subtrees are heaps, we can adjust it to a heap with PERCOLATE_DOWN procedure



⇒ we can adjust subtrees in the second last layers into heaps

⇒ we can adjust subtrees in the third last layers into heaps

⇒ … ⇒ we can adjust the tree into a heap

Solution: start from the last non-leaf node, PERCOLATE_DOWN on the corresponding subtrees

In reverse order, we adjust the nodes in depth-decreasing order

```
BUILD_HEAP(array)
1. for i from array.length / 2 to 1:
2.    PERCOLATE_DOWN(array, array.length, i)
```

The last non-leaf node is the parent of the last leaf node

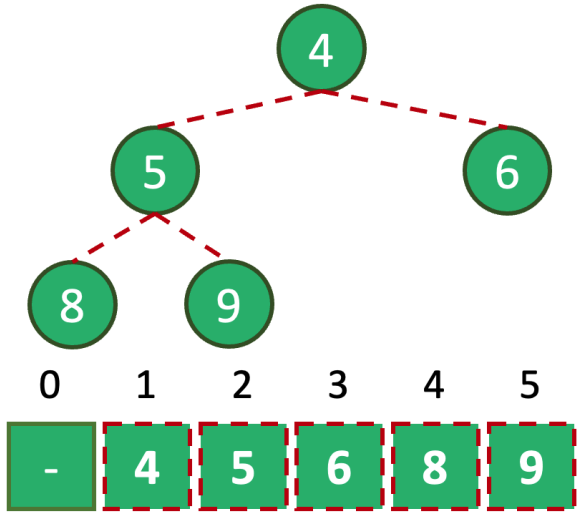
Time complexity: $O(n\log n)$

Heap Sort

Pop the highest and add it from the tail of the array

Key idea: after popping, the size of heap decrease by 1, the emptied slot can store the popped largest value

- Recall pop: replace root with the last element, then PERCOLATE_DOWN
- Extension for inplace sorting: swap the root with the last element, then PERCOLATE_DOWN

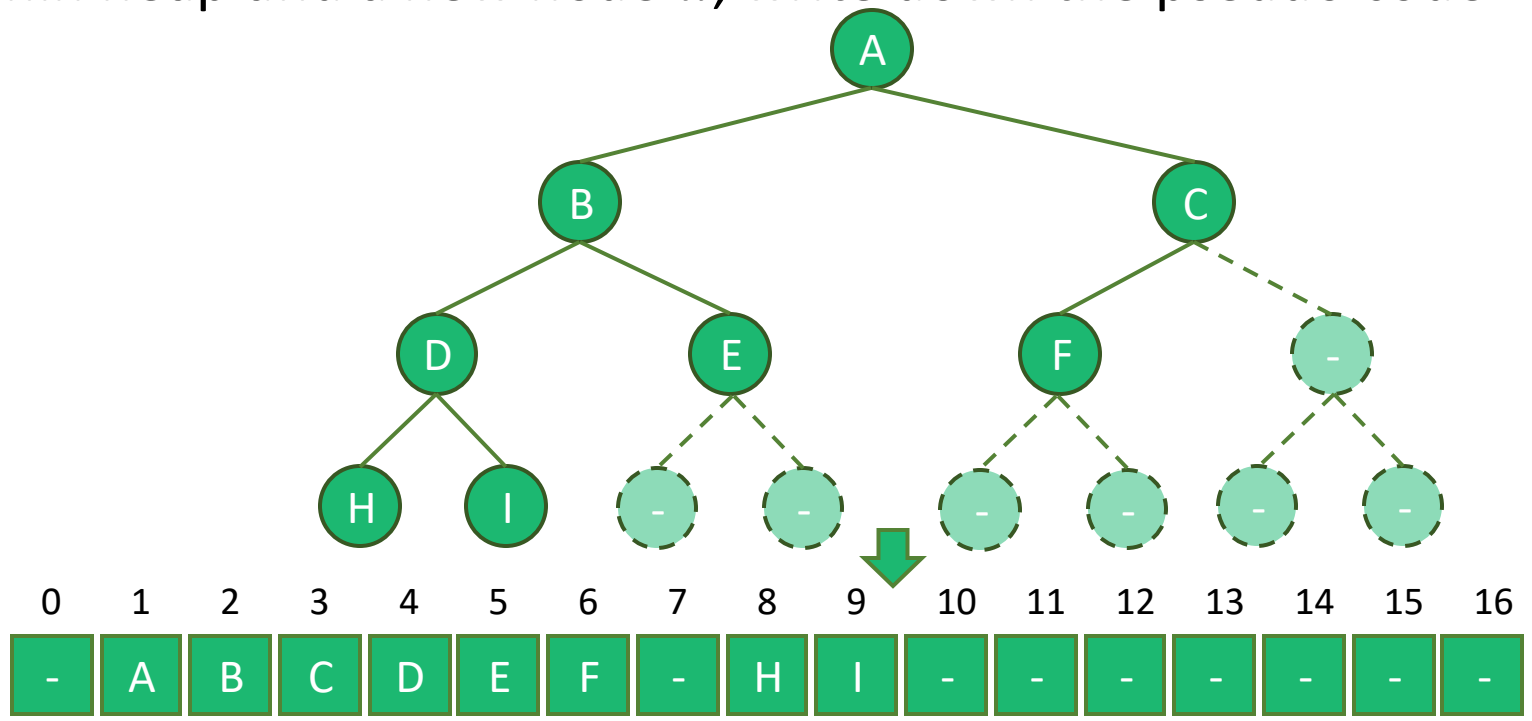



```
HEAPSORT(array)
1. BUILD_HEAP(array)
2. for i from array.length to 2:
3.     SWAP(array[1], array[i])
4.     PERCOLATE_DOWN(array, i, 1)
```

Heap Sort

Practice (with array-based representation):

- Given an array representing the min heap, write the pseudo code for popping the minimum and do percolate down for a heap
- Given an array representing the min heap and a new node u, write down the pseudo code for insert it into the heap



Pop Minimum

```
PERCOLATE_DOWN(heap, size, index)
1. left_child_index = 2 * index
2. right_child_index = 2 * index + 1
3. smallest = index
4. if left_child_index < size\
5.    and heap[left_child_index] < heap[smallest]:
6.       smallest = left_child_index
7. if right_child_index < size \
8.    and heap[right_child_index] < heap[smallest]:
9.       smallest = right_child_index
10.if smallest != index:
11.      SWAP(heap[index], heap[smallest])
12.      PERCOLATE_DOWN(heap, smallest)
```

```
DELETE_MIN(heap)
1. min_element = heap[1]
2. last_element = heap[heap.length]
3. heap[1] = last_element
4. heap.length -= 1
5. PERCOLATE_DOWN(heap, 1)
6. return min_element
```

Insert Element

```
PERCOLATE_UP(heap, index)
1. parent_index = index / 2
2. # Compare the element with its parent
3. while index > 1 and heap[index] < heap[parent_index]:
4.     SWAP(heap[index], heap[parent_index])
5.     index = parent_index
6.     parent_index = index / 2
```

```
INSERT(heap, element)
1. heap[heap.length+1] = element
2. heap.length += 1
3. PERCOLATE_UP(heap, heap.length)
```

Disjoint set

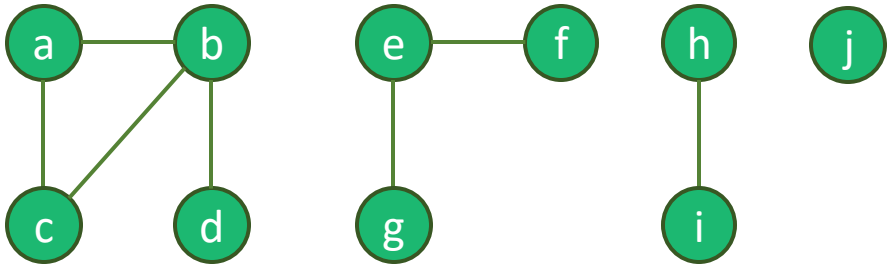
Requirement

Consider a situation: given some persons, how to quickly support the following operations?

- Find whether individual x and y are linked through direct/indirect friend relations
- Add a new friendship relation between x and y

Naïve solution: record the links, then search for all the reachable nodes from x (graph searching)

- High time and space complexity



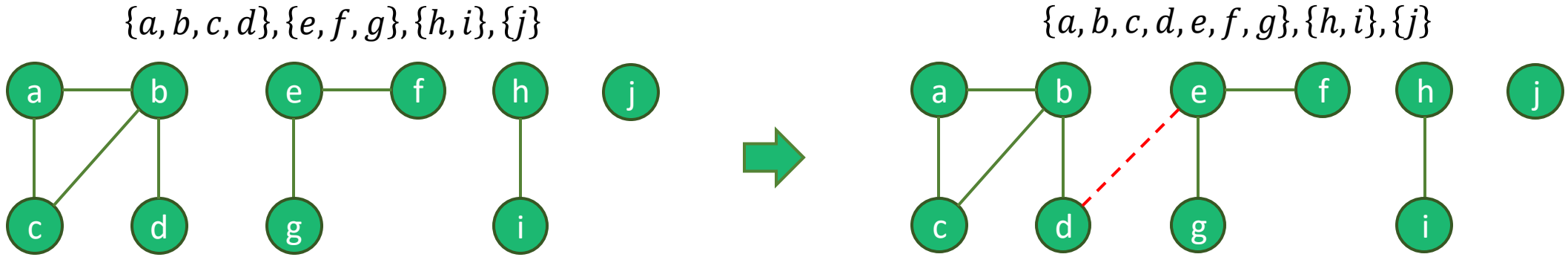
Requirement

Consider a situation: given some persons, how to quickly support the following operations?

- Find whether individual x and y are linked through direct/indirect friend relations
- Add a new friendship relation between x and y

A set-based view

- Nodes in each linked components can reach each other
- Only care about the linked components – ignore links and record with set
- Add a link is to union two sets



Disjoint Set

Store a collection of non-overlapping sets, also called union–find data structure or merge–find set, support operations:

- Distinguish: whether two elements are in the same set

$$< \{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}, < a, c >> \Rightarrow same$$

- Union: given two elements, merge their set

$$\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\} \Rightarrow \{a, b, c, d\}, \{e, f, g, h, i\}, \{j\}$$

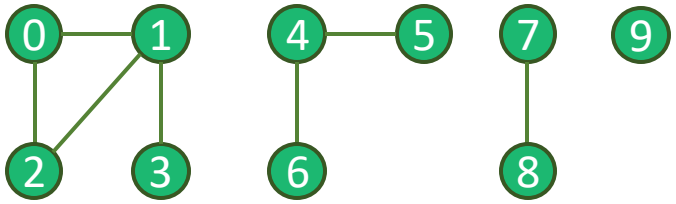
How to realize a disjoint set?

Straightforward solution: create separate collection for each set

- Distinguish: traverse all the collections and elements
- Union: find the corresponding collection, copy elements in one to the other
- **Problem**: large space and time complexity for massive number of sets and operations

Another Naïve Solution

Core idea: represent each set with a member (representative), record each element's set's representative with an array



0, 1, 2, 3 belongs to 0's set
4, 5, 6 belongs to 5's set,
…

id[]

0	1	2	3	4	5	6	7	8	9
0	0	0	0	5	5	5	7	7	9

Distinguish: whether two elements are in the same set

- Find and compare the representative: $\Theta(1)$

FIND(2)

0	1	2	3	4	5	6	7	8	9
0	0	0	0	5	5	5	7	7	9

→ 0

Union: given two elements, merge their set

- Traverse the array and modify the corresponding elements: $\Theta(n)$

0	1	2	3	4	5	6	7	8	9
0	0	0	0	5	5	5	7	7	9

UNION(2, 5)

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	7	7	9

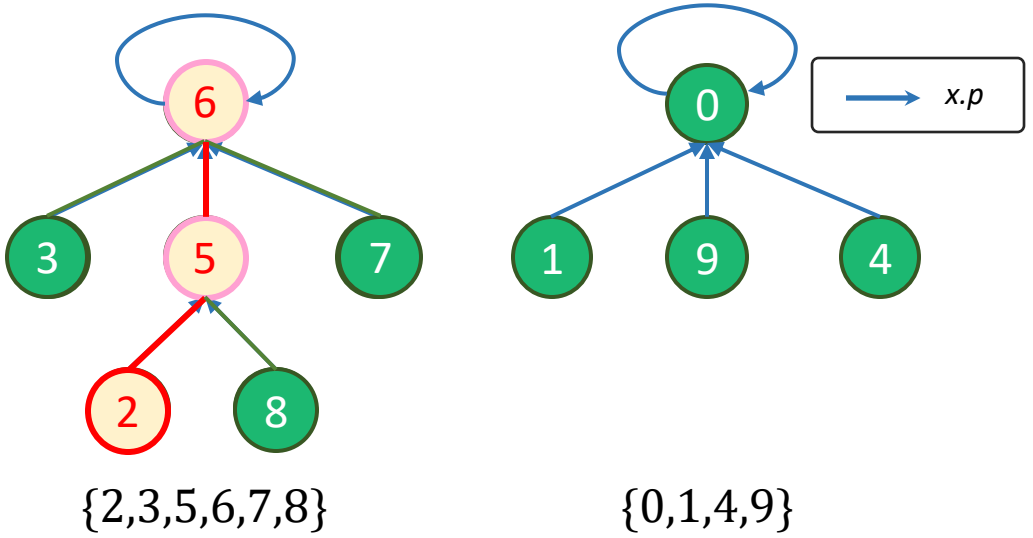
Disjoint Set with Tree

Represent each set with a tree, where each element is a node

- Use root node as the representative

Distinguish whether two elements are in the same set

- Find the root through parent pointer (set the parent of the root to itself to ease coding)



$\{2,3,5,6,7,8\}$ $\{0,1,4,9\}$

```
FIND(x)
1. if x.p != x:
2.     return FIND(x.p)
3. else:
4.     return x
```

Time complexity: $O(h)$

Disjoint Set with Tree

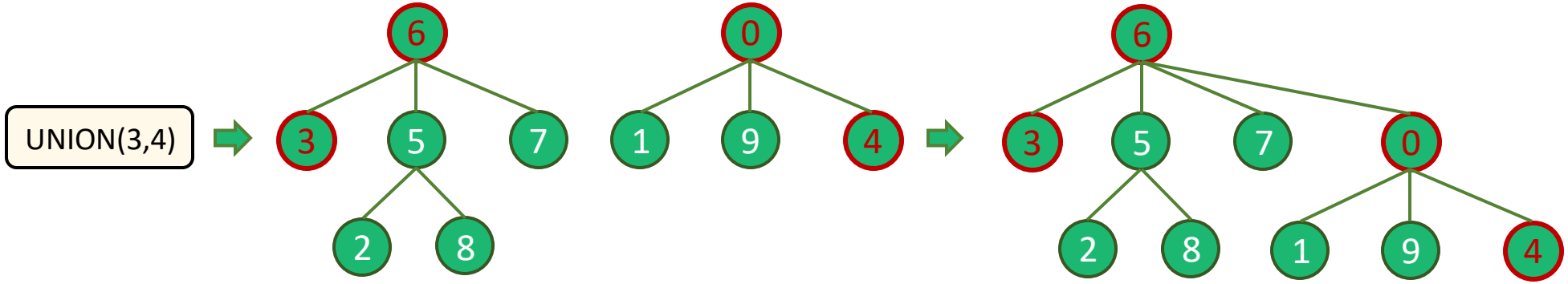
Represent each set with a tree, where each element is a node

- Use root node as the representative

Union: given two elements (x and y), merge their set

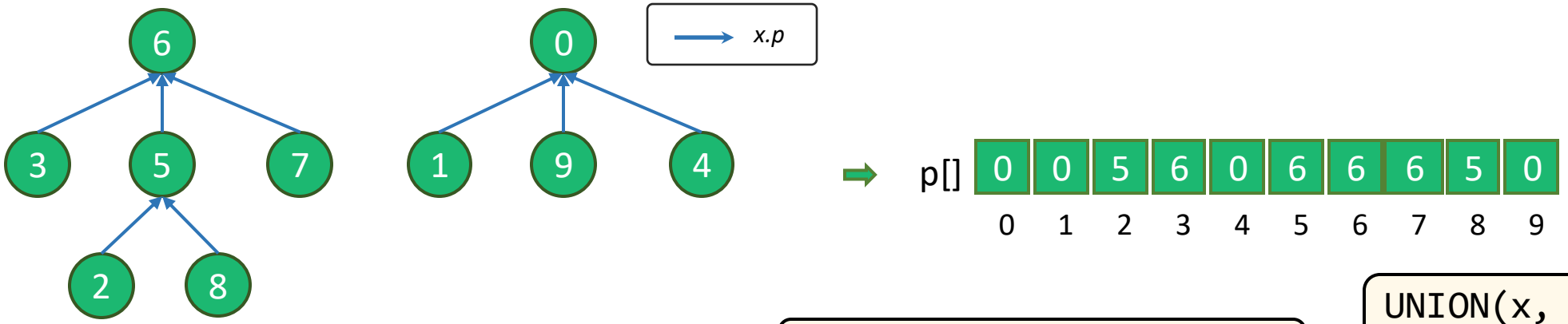
- Find the roots of x and y's corresponding tree $O(h)$
- Make x's roots as the parent of y's root $O(1)$

```
UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     root_y.p = root_x
```



Disjoint Set with Tree

Store tree structure in an array: since only parent information is needed



{2,3,5,6,7,8}

{0,1,4,9}

```
FIND(x)
1. if p[x] != x:
2.     return FIND(p[x])
3. else:
4.     return x
```

```
UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     p[root_y] = root_x
```

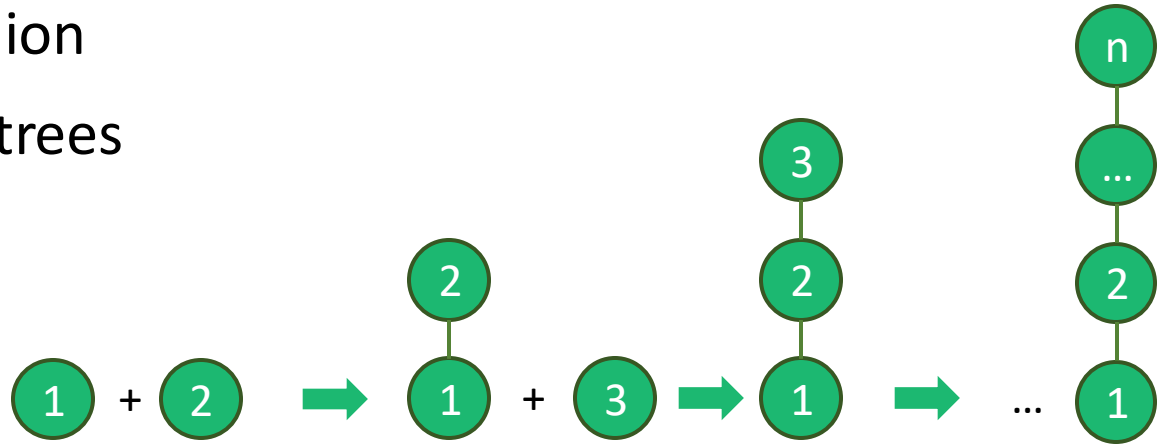
Disjoint Set with Tree

Any potential deficiency?

- Link $< 1, 2 >, < 2, 3 >, < 3, 4 >, < 4, 5 >, < 5, 6 >, \ldots \ < n-1, n >$
- $\mathrm{T}(n) \ = \Theta(n)$ for find and union

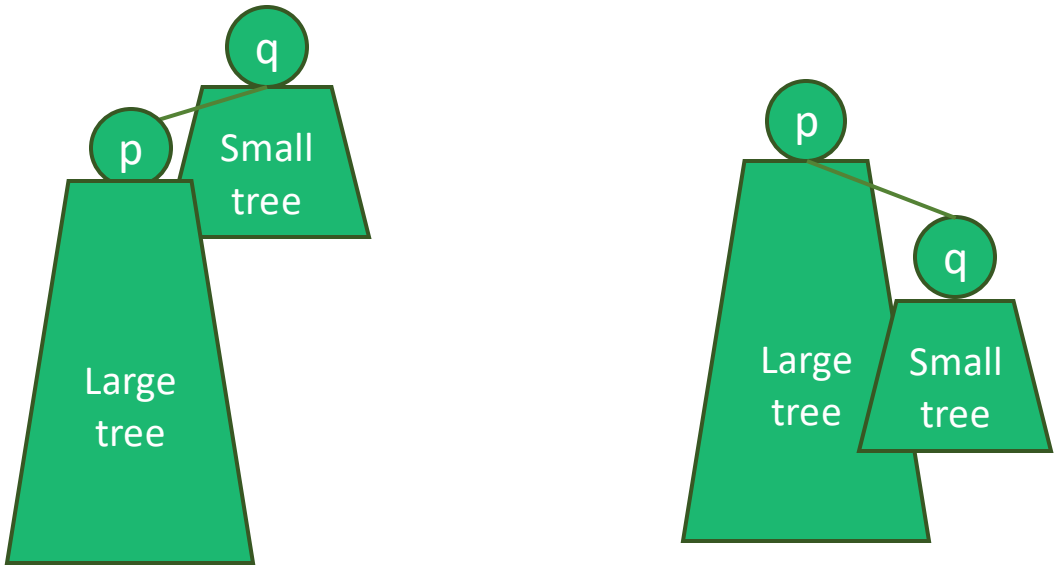
Improve complexity: avoid tall trees

- Union-by-Rank
- Path Compression



Union by Rank

Keep track each tree's size (number of nodes), link roots of smaller tree to root of larger tree.



Intuitively reduce height

```
Weight_UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     if root_x.size < root_y.size:
7.         root_x.p = root_y
8.         root_y.size += root_x.size
9.     else:
10.        root_y.p = root_x
11.        root_x.size += root_y.size
```

Union by Rank

Proposition: With union by rank, depth of any node x is at most $\log n$.

Proof:

- Notation: $d(x)$ - depth of x, $|T|$ - size of a tree T
- $d(x)$ possibly increase when the tree (T) containing x is merged with another tree T':
  - $d(x)$ increases by 1 only when $|T'| \geq |T|$
  - The new tree's size $|T''| = |T| + |T'| \geq 2|T|$, at least doubles.
  - If $d(x) = h$, then $2^h \leq |T|$. Since $|T| \leq n$, $h \leq \log n$

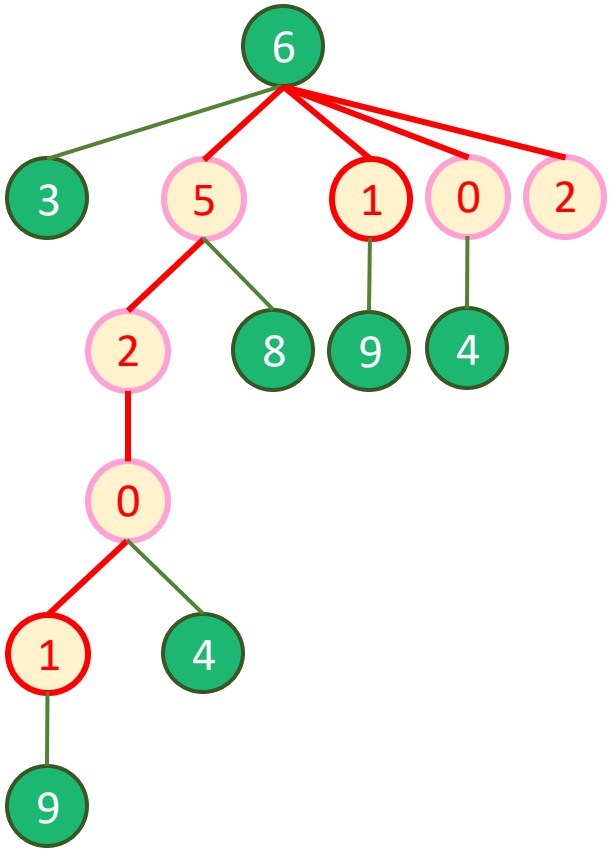
Conclusion: with union by rank, time complexity for find and union operation are $O(\log n)$

Path Compression

After finding the root for each node, move the node to be its child

- Find: O(1) after many find operations
- Union: with m arbitrary linking operations, time complexity is roughly $O((m+n)\log n)$[1]

Path compression + union by rank: nearly O(1) on average




```
FIND_PC(x)
1. if x.p != x:
2.     x.p = FIND_PC(x.p)
3. return x.p
```

[1] R. Seidel and M. Sharir. Top-Down Analysis of Path Compression. Siam J. Computing, 2005, Vol. 34, No. 3, pp. 515-525.

Summary

- Trees
- Binary Tree
- Heap
- Heap Sort
- Disjoint Set

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures