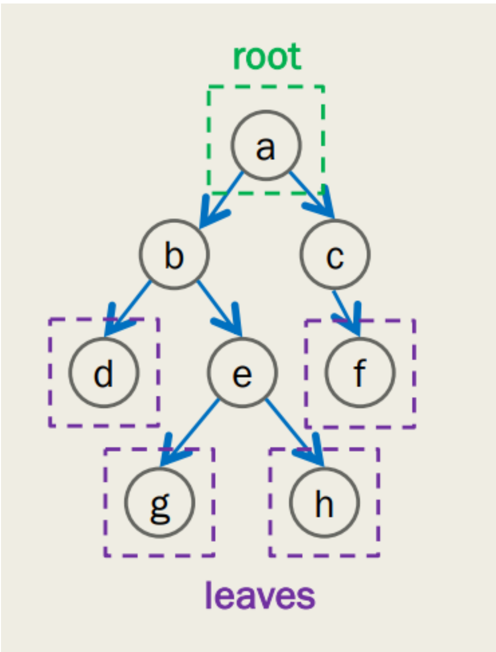
Part3 - Tree&Heap

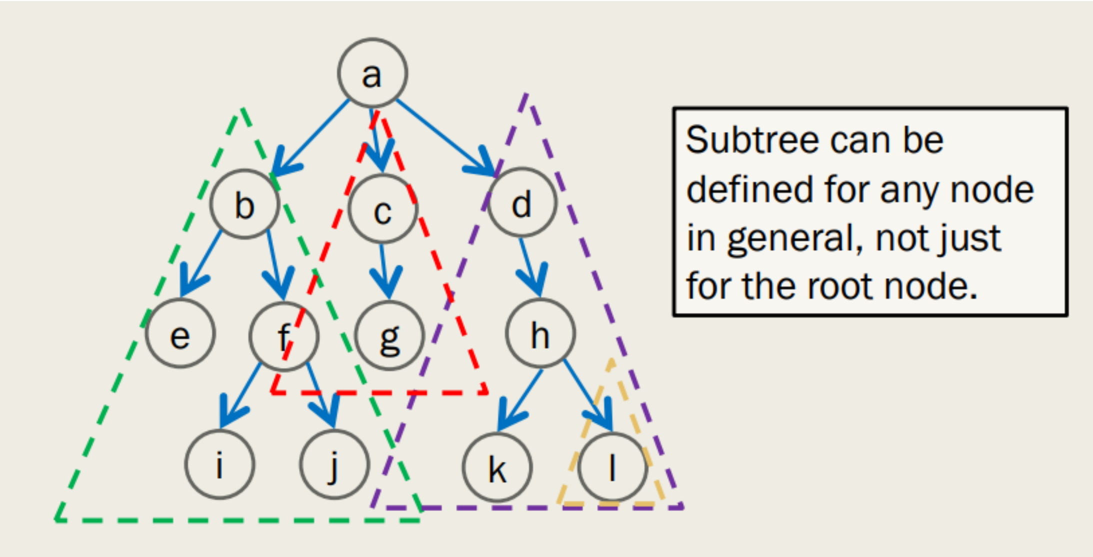
Tree

Several Concepts



- root: node at the top.

- parent-child relationship.
- leaf: node without children.
- subtree



- sibling
- path:
  - sequence of nodes such that next node in the sequence is a child of the previous.
  - a -> c -> g. path length 2.
  - path length can be 0. one node to itself.
- ancestor / descendant: If there exists a path from a node A to a node B, then A is an ancestor of B and B is a descendant of A.
- depth / level: length of the path from the **root** to the node.
- height of **node**: length of the **longest** path from the node to a **leaf**.
- height of **tree** / depth of **tree**: height of the **root**.
- number of **levels** of a **tree**: height of **tree** + 1.
- degree of **node**: number of children.
- degree of **tree**: the **maximum** degree of a **node** in tree.

Binary Tree

Definition

- at most two children
- or empty tree

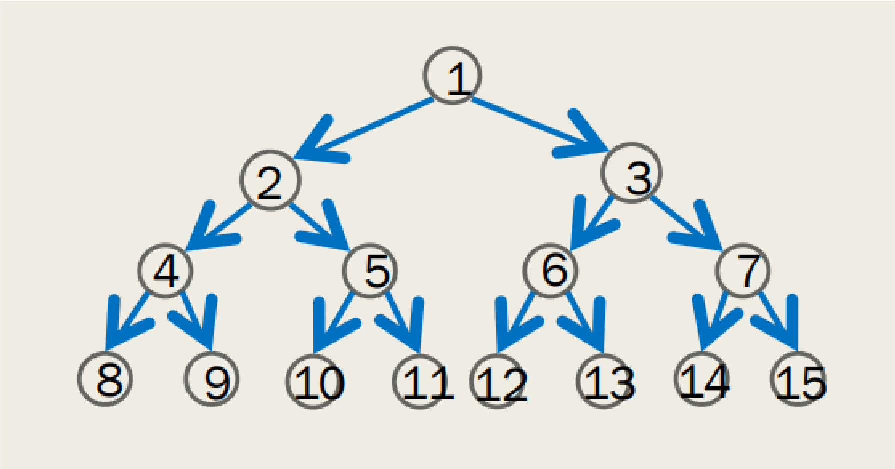
Properties

- $h+1 \leq n \leq 2^{h+1}-1$
- or $\log_2(n+1)-1 \leq h \leq n-1$

Different Types of Binary Tree

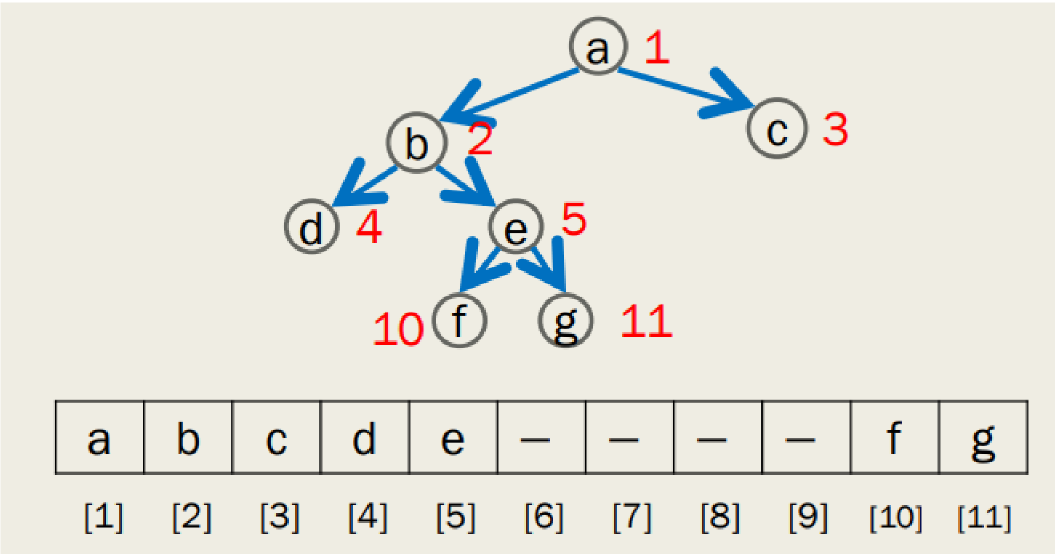
- proper: every node has **0 or 2** children
- complete:
  - every level **except** the **lowest** is **fully populated**
  - the **lowest** level is populated from **left to right**
- perfect: fully populated; $2^{h+1}-1$

Numbering Nodes In a Perfect Binary Tree



Representing Binary Tree

- array



- linked structure

```
struct node {
 Item item;
 node *left;
 node *right;
};
```

Binary Tree Traversal

Depth-First Traversal

- pre-order: node - left subtree - right subtree
- in-order: left subtree - node - right subtree
- post-order: left subtree - right subtree - node

Level-Order Traversal

- top from bottom
- left from right
- queue
  - enqueue the root into an empty queue
  - while the queue is not empty, dequeue a node from the front of the queue
    - visit the node
    - enqueue left child (if exists) and right child (if exists)

Rebuild the Tree from Traversal Sequences

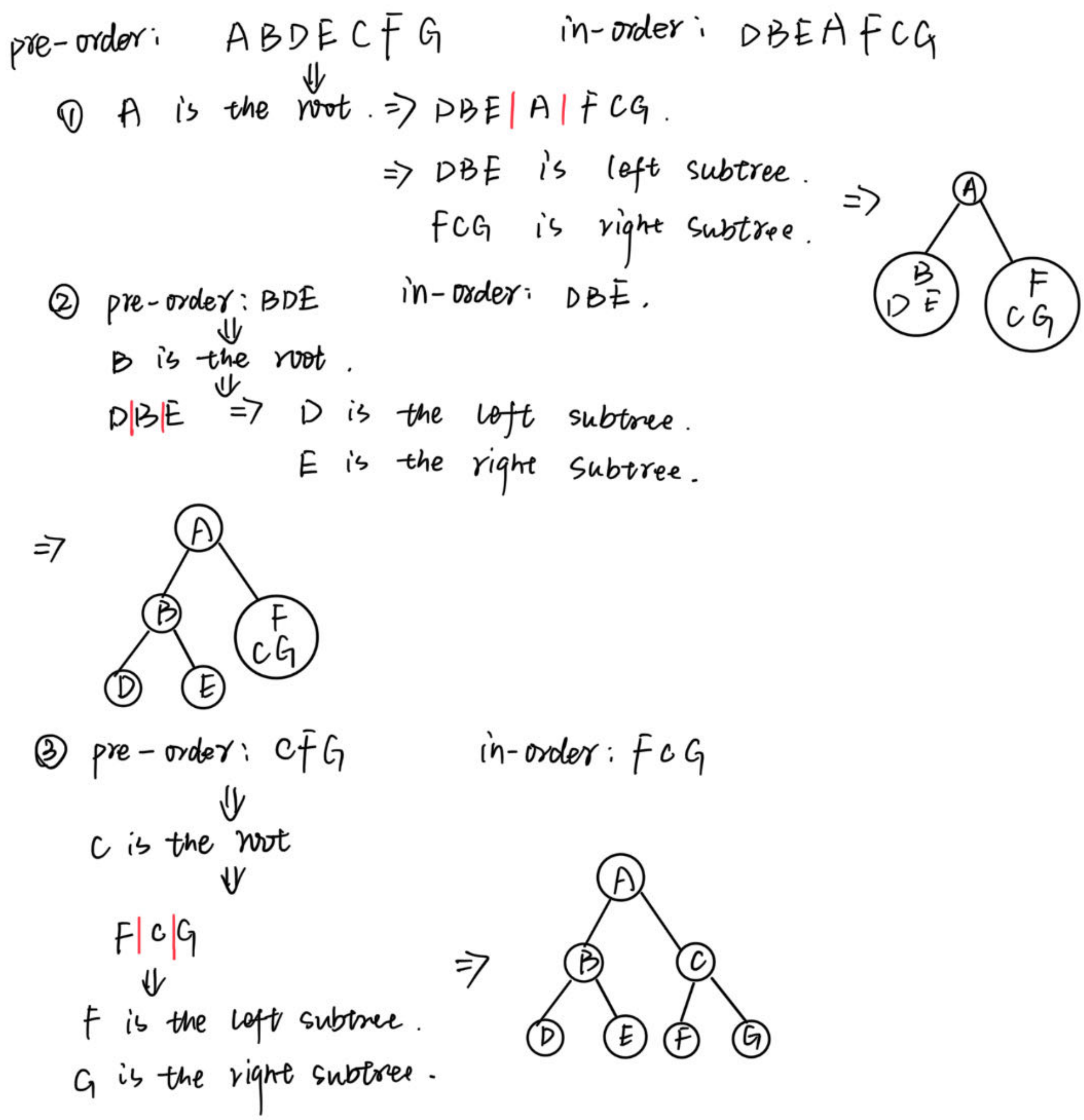
- We can determine one tree from the **in-order** traversal and **any** of the **pre-order** and **post-order** traversal.
- and we **CANNOT** do it **without in-order traversal**. Because the pre-order and post-order traversal give only the parent-child relationship. **Only** the in-order traversal gives the information about left and right subtrees.
- several different trees can have exactly the **SAME** pre-order and post-order traversal.

Method to Rebuild

- find the root of the tree or subtree from the **post-order** or **in-order** traversal
- divide the left and right subtree in the **in-order** traversal
- repeat the previous two steps until the whole tree is determined

Exercise

Given pre-order traversal sequence ABDECFG and in-order traversal sequence DBEAFCG, rebuild the binary tree.



Priority Queue and Heap

Priority Queue

- isEmpty
- size
- enqueue: put an item into the priority queue
- dequeueMin: remove element with min key
- getMin: get item with min key
- implement with STL (C++ built-in libraries)

```
priority_queue<int, vector<int>, greater<int> > my_heap; // min_heap
my_heap.empty();
my_heap.top(); // getMin
my_heap.pop(); // dequeueMin
my_heap.push(); // insert
```

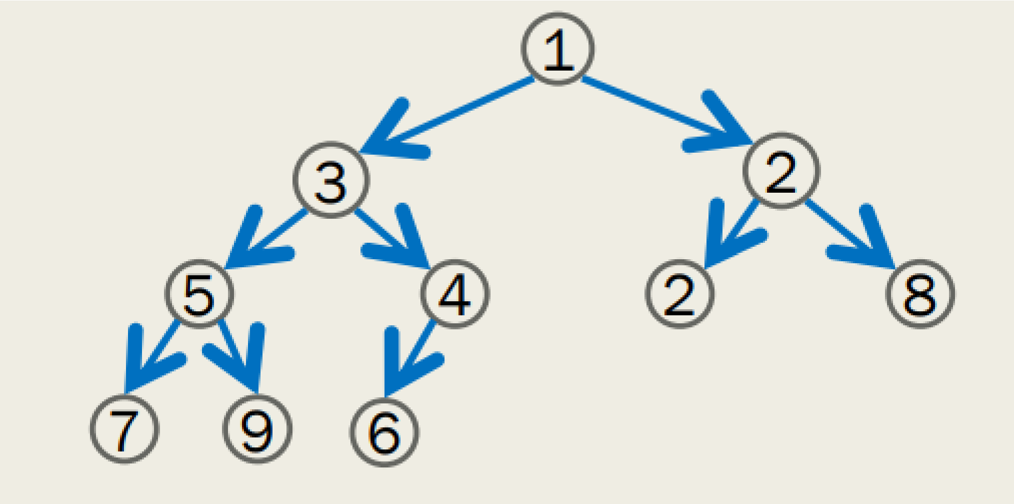
Min Heap

Definition

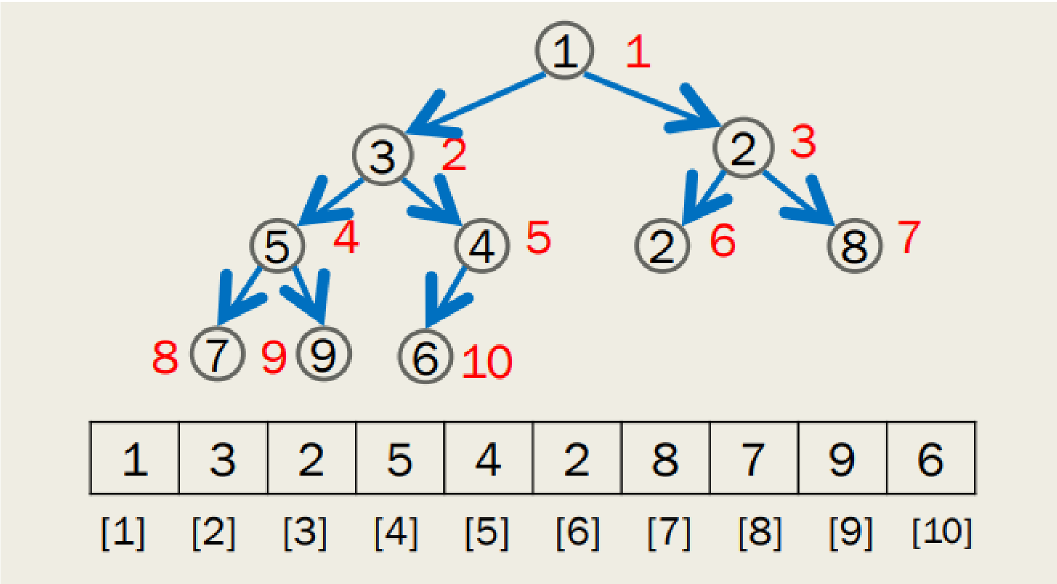
- binary heap (complete binary tree)
- a tree where for **any** node v, the key of v is smaller than or equal to (≤) the keys of **any descendants** of v.

Properties

- the key of the **root** of any subtree is always the smallest among all the keys in that subtree.
- the keys of nodes across **subtrees** have no required relationship



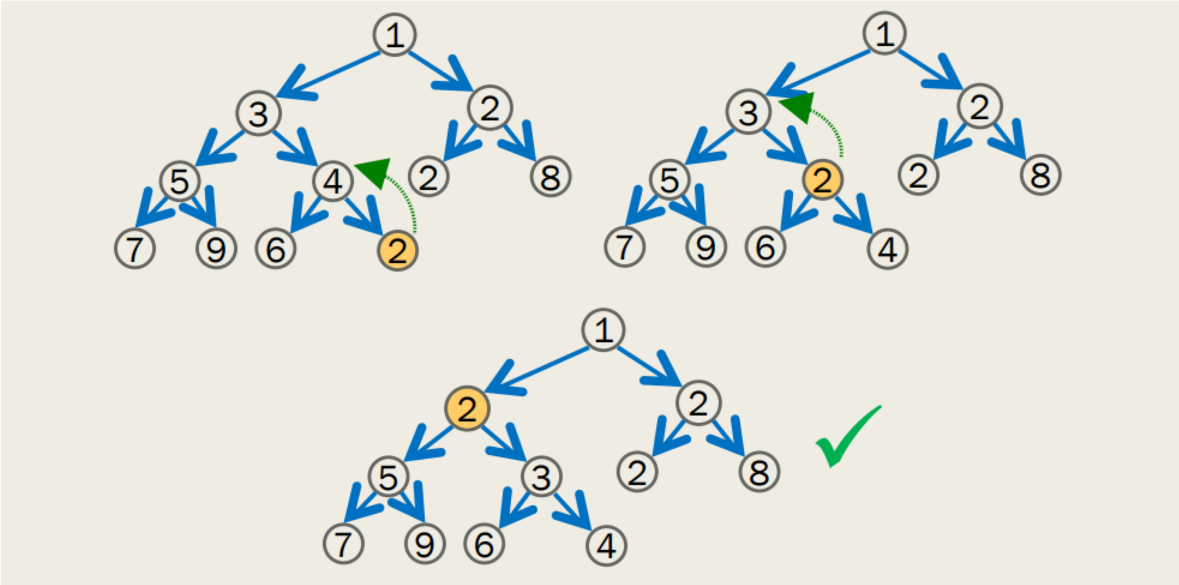
Binary Heap Implementation as an Array



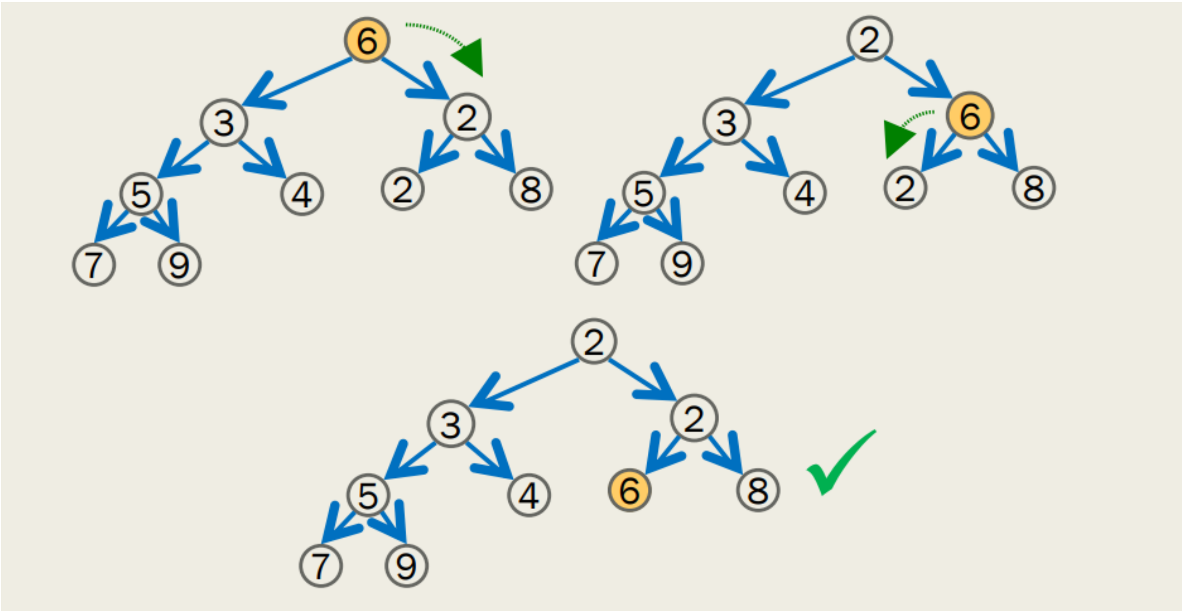
Don't worry about the start index.

Operations

- percolating-up; $O(\log n)$



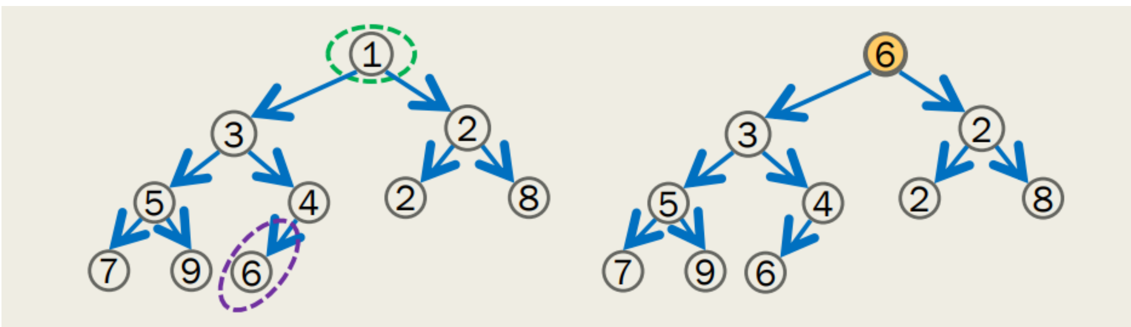
- percolating-down; $O(\log n)$



- enqueue; $O(\log n)$

```
void minHeap::enqueue(Item newItem) {
 heap[++size] = newItem;
 percolateUp(size);
}
```

- decrease key; $O(\log n)$
- dequeueMin; $O(\log n)$



```
Item minHeap::dequeueMin() {
 swap(heap[1], heap[size--]);
 percolateDown(1);
 return heap[size+1];
}
```

Initialize Heap

- insert each entry one by one; $O(\log(n!)) = O(n \log n)$
- initialize from array / **heapify**; $O(n)$
  - put all the items into a complete binary tree (array)
  - starting at the rightmost array position that has a child, percolate down all nodes in reverse level-order

```
for (i = size / 2; i >= 1; i--){
    percolateDown(i);
}
```

Q: Can we initialize the array with **percolateUp** instead of **percolateDown**?

A: Yes. But the time complexity is $O(n \log n)$, and we should start from the top instead of the bottom. The analysis is similar to the one in Prof. Ban's slides.

Application

- sorting
- median maintenance