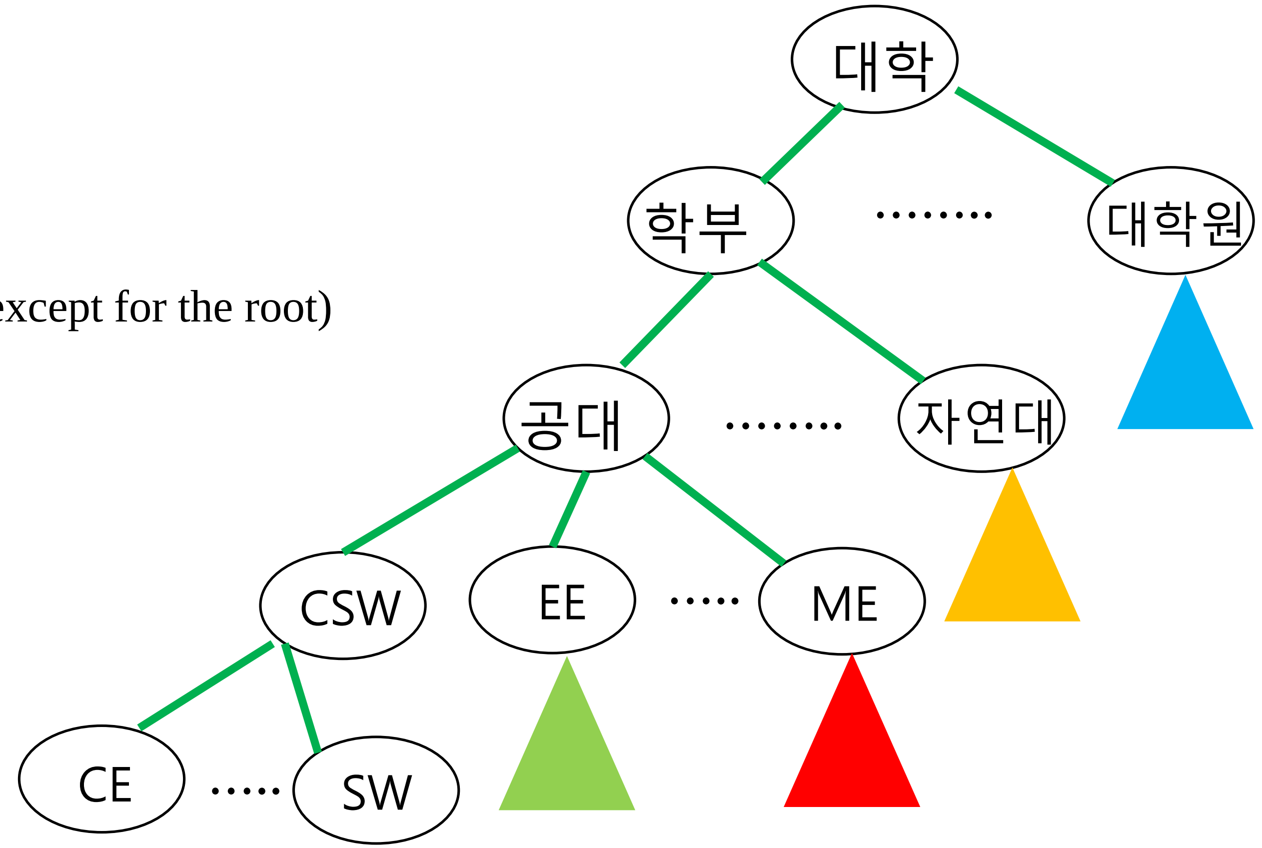
Chap 5. Trees

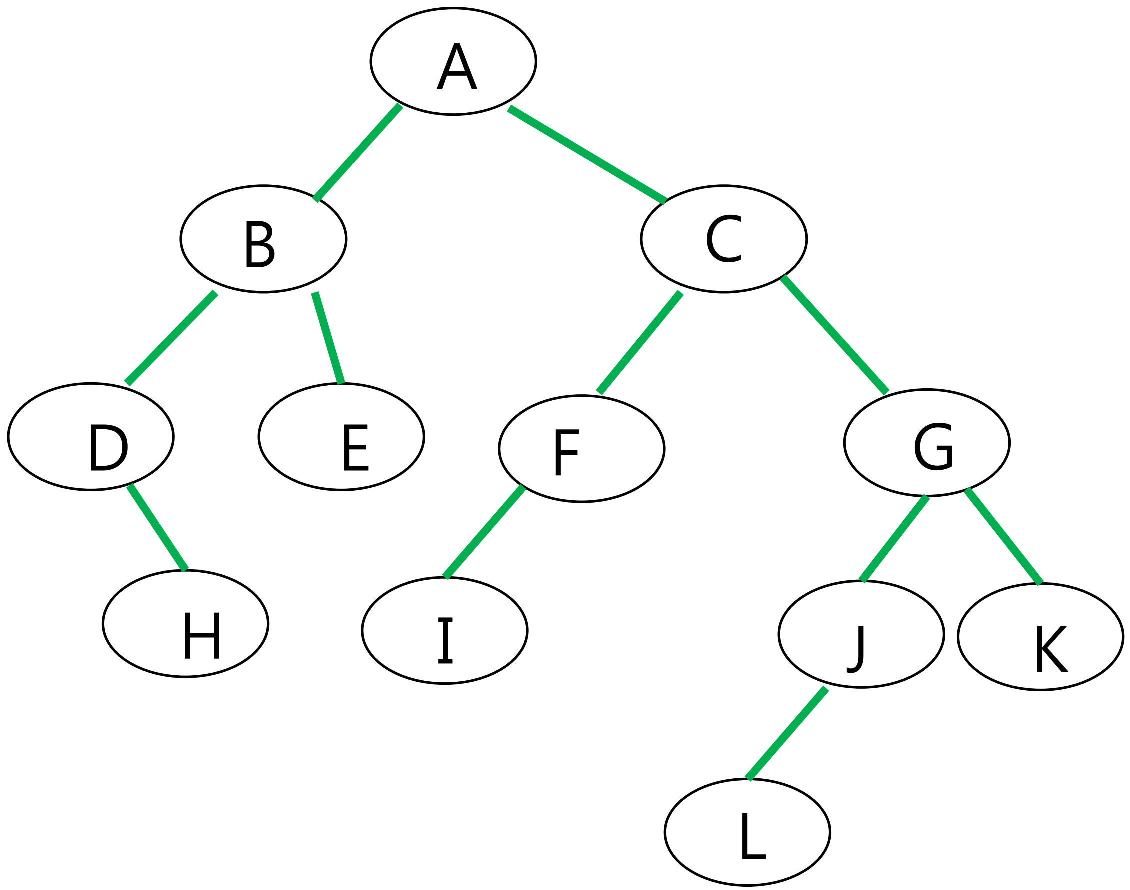
Tree

- 계층(hierarchy) 구조의 데이터를 표현하기 위한 자료 구조
- Terminologies
  - Node
  - Root (node)
  - Child (node), children
    - Degree, fan-out
  - Parent (node)
    - Each node has only 1 parent node (except for the root)
  - Sibling, ancestor, descendant
  - Leaf (node), leaves, Non-leaf node
  - Subtree
  - Forest
  - Path
    - from node x to node y
  - Level
    - 0, 1, 2, …, h-1
    - 1, 2, 3, …, h
  - Height: max. level



Binary tree

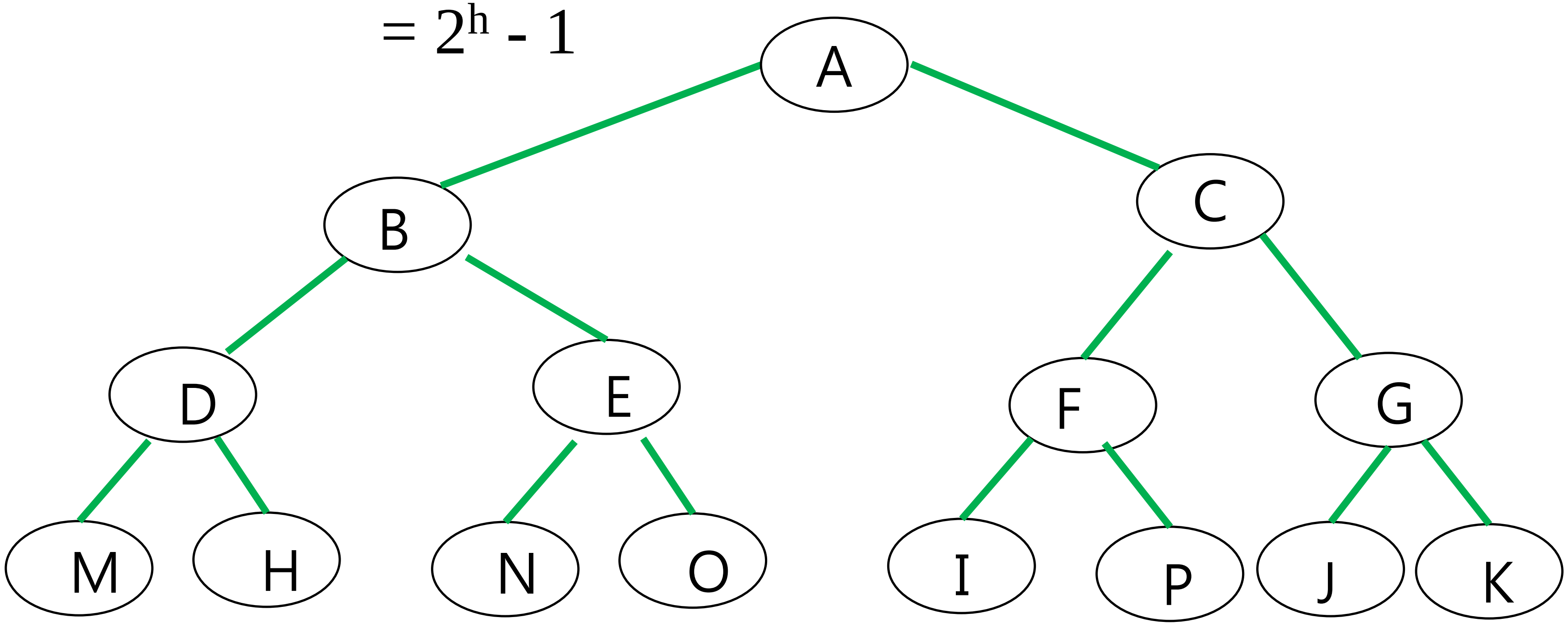
- Each node can have at most 2 child nodes
  - Left child
  - Right child
- Number of children of a node
  - 0 or //leaf node
  - 1 or
  - 2
- cf. n-ary tree
- Subtree
  - Left
  - Right



Full binary tree

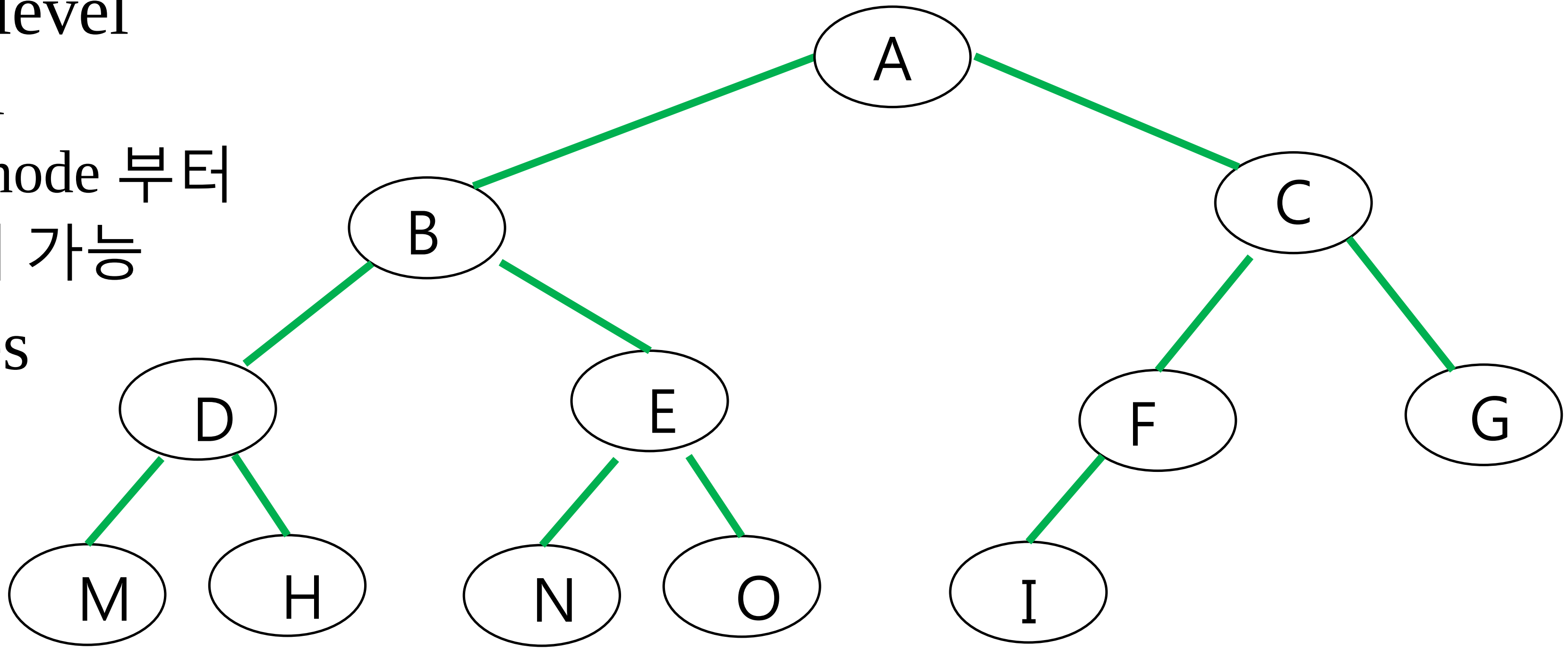
- Every node has 2 children except for the leaf nodes
- Height = h
- Number of nodes = $1 + 2 + 2^2 + 2^3 + \ldots + 2^{h-1}$

  $= 2^h - 1$



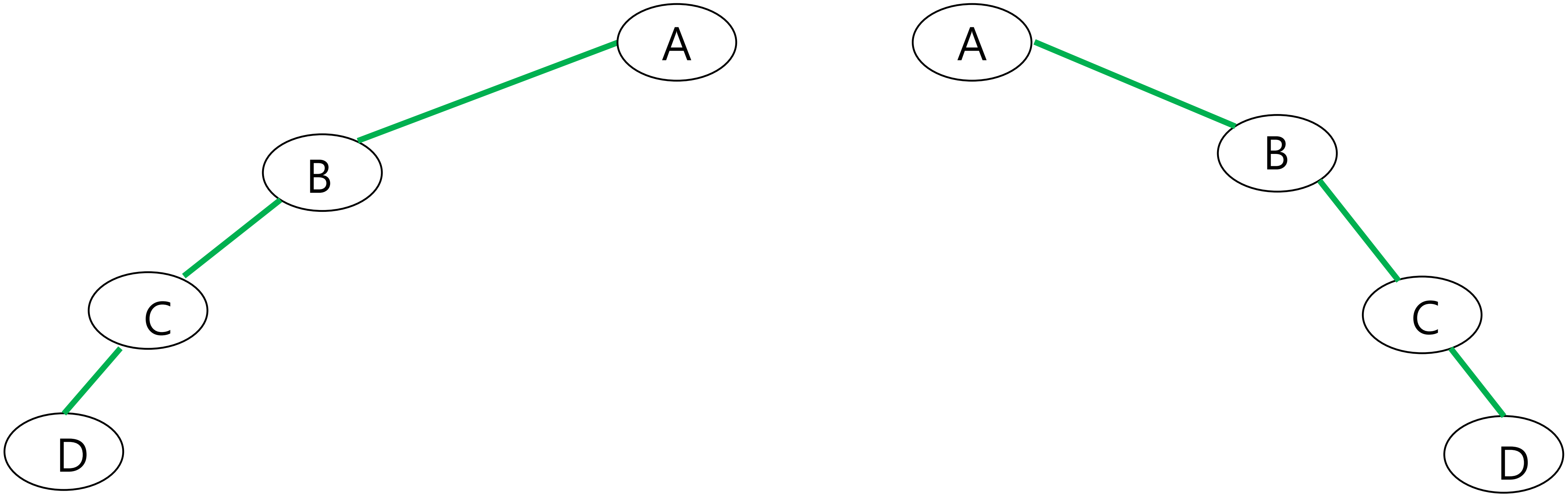
Complete binary tree

- Height = h
- Level: 0, 1, 2, …, h-1
- Level h-2까지는 full binary tree
- Level h-1 //leaf level
  - May not be full
  - rightmost leaf node 부터 한 노드씩 삭제 가능
- Number of nodes
  - Min: 2^{h-1}
  - Max: 2^{h} -1



Skewed binary tree

- Left skewed or right skewed
- Height = number of nodes



노드 수(n)와 높이(h)

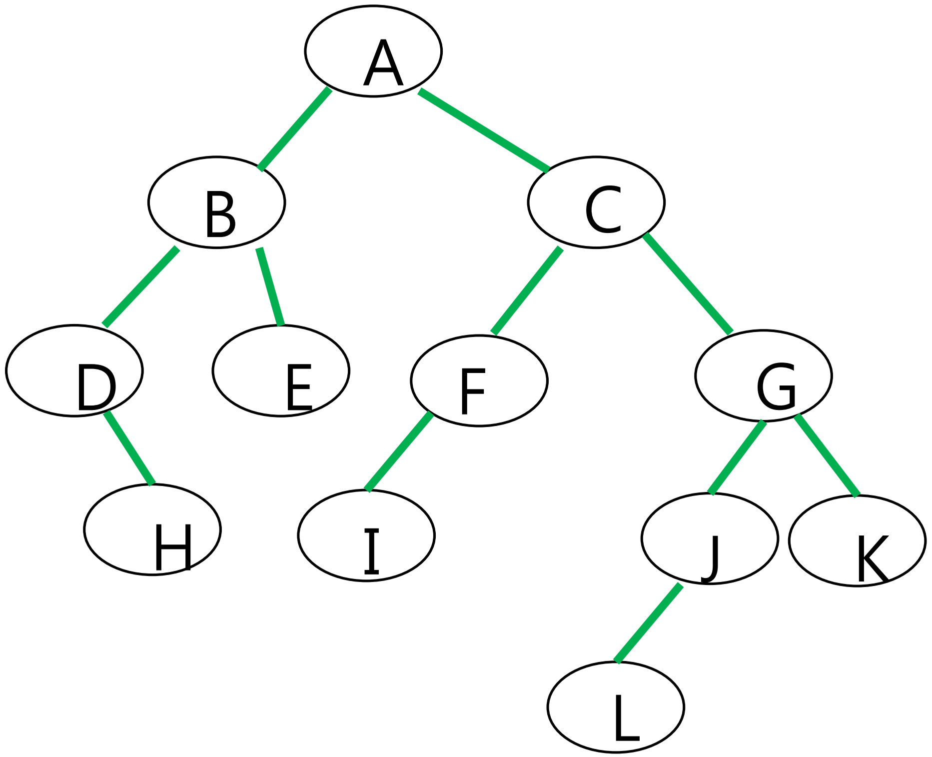
- Skewed binary tree: $h = n$
- Full binary tree
  - $h = \log_2(n+1)$
  - $h = 1 + \log_2 n_0$ (n_0 : leaf node의 수)
- Complete binary tree
  - $2^{h-1} \le n \le 2^h - 1$
    - $2^{h-1} \le n < 2^h \Rightarrow h-1 \le \log_2 n < h \Rightarrow h = \lfloor \log_2 n \rfloor + 1$
    - $2^{h-1} < n+1 \le 2^h \Rightarrow h-1 < \log_2(n+1) \le h \Rightarrow h = \lceil \log_2(n+1) \rceil$
- Arbitrary binary tree
  - $h \le n \le 2^h - 1$
  - $\log_2(n+1) \le h \le n$

Binary tree 의 표현

- Array representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…….	28
	A	B	C	D	E	F	G		H			I		J	K	…..	L

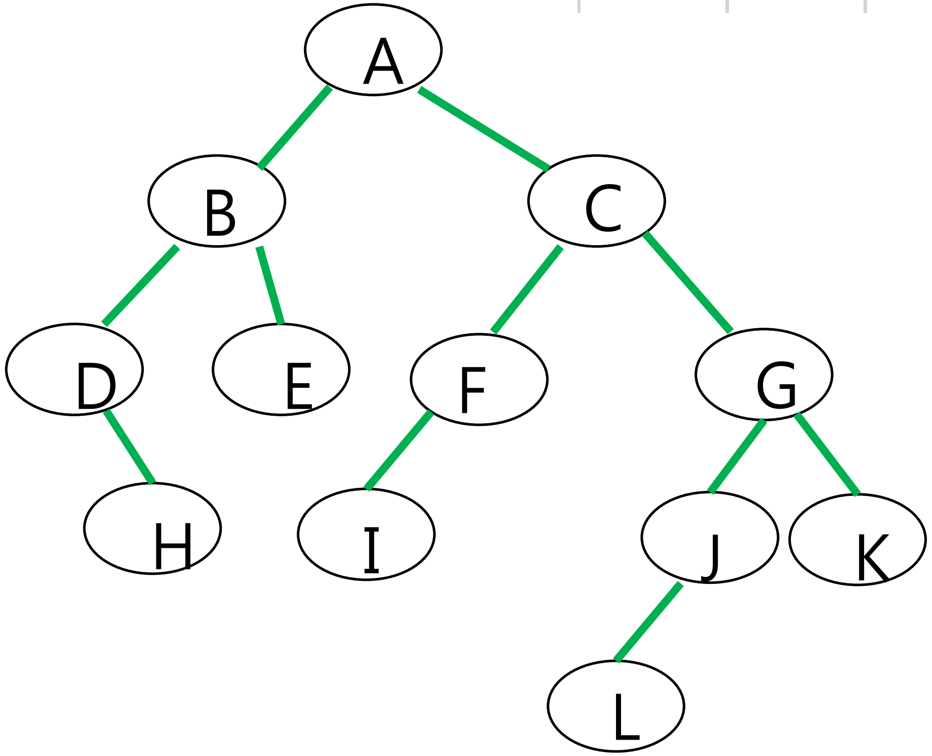
- Node in A[i]
  - Left child: A[2*i]
  - Right child: A[2*i+1]
  - Parent: A[j], j = $\left\lfloor \frac{i}{2} \right\rfloor$

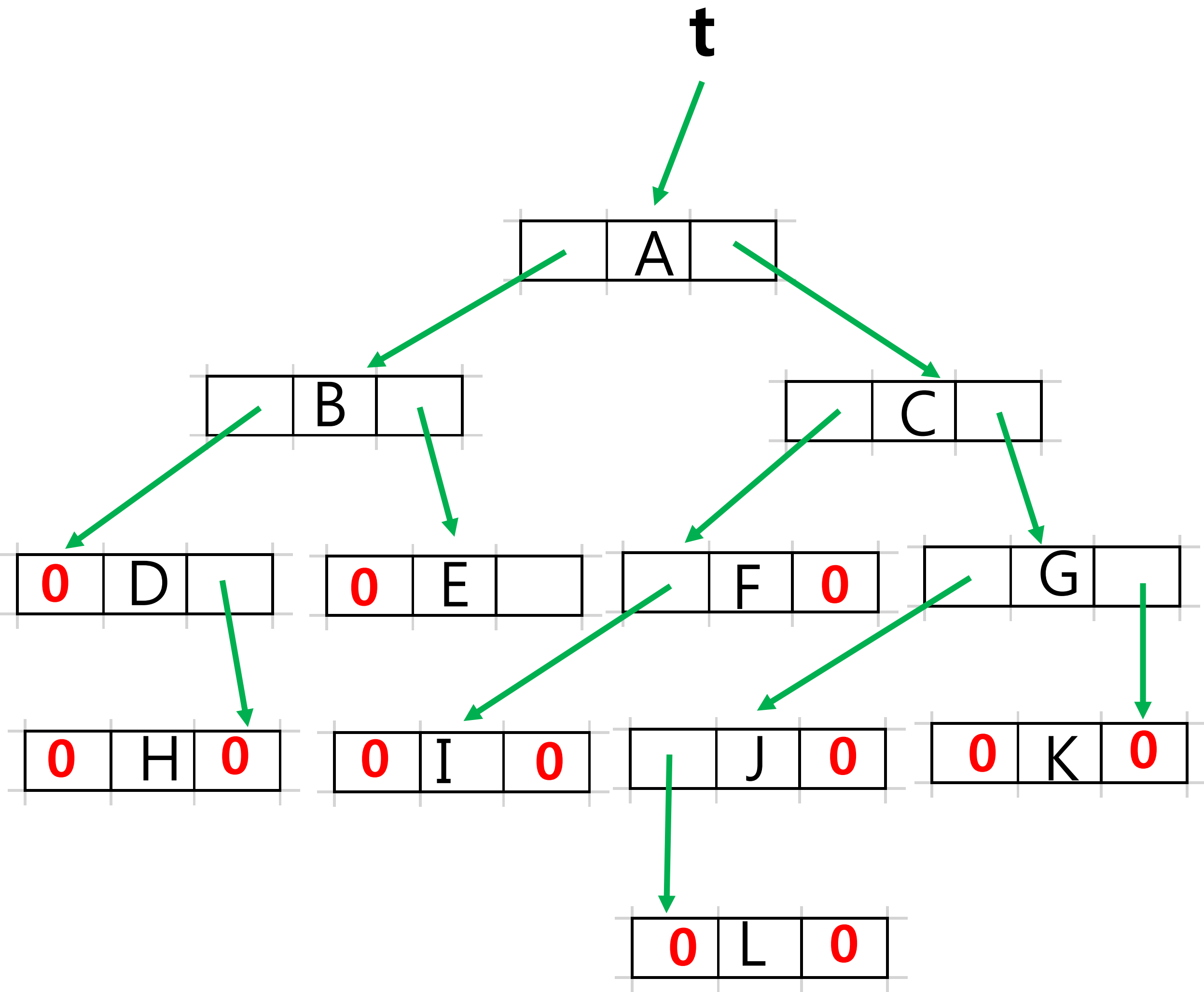


Binary tree 의 표현

- linked representation
  - Node structure

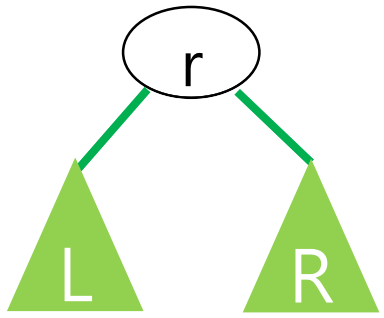
left	data	right

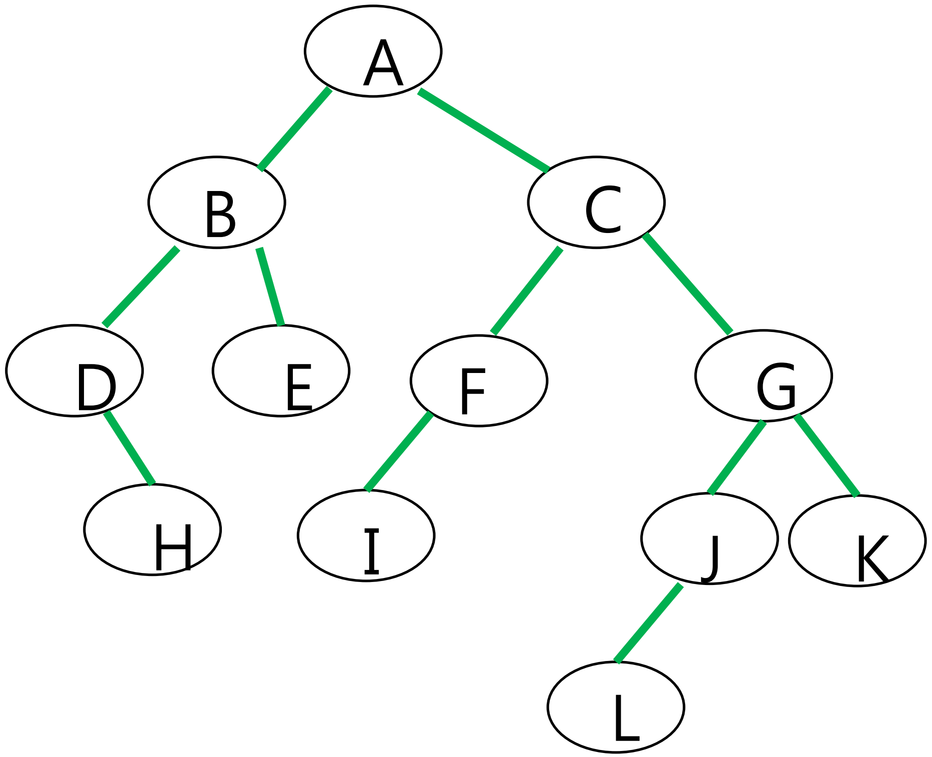




Binary tree 탐색(traversal)

- Preorder: root-Lsubtree-Rsubtree
- Inorder: Lsubtree-root-Rsubtree
- Postorder: Lsubtree-Rsubtree-root
- Level:
  - From root to leaves
  - From left to right
- Preorder: **ABDHECFIGJLK**
- Inorder: **DHBEAIFCLJGK**
- Postorder: **HDEBIFLJKGCA**
- Level: **ABCDEFGHIJKL**

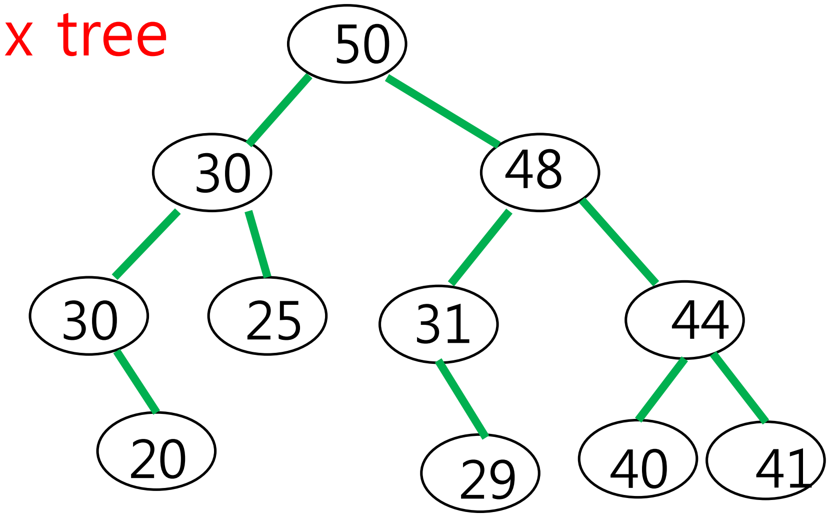




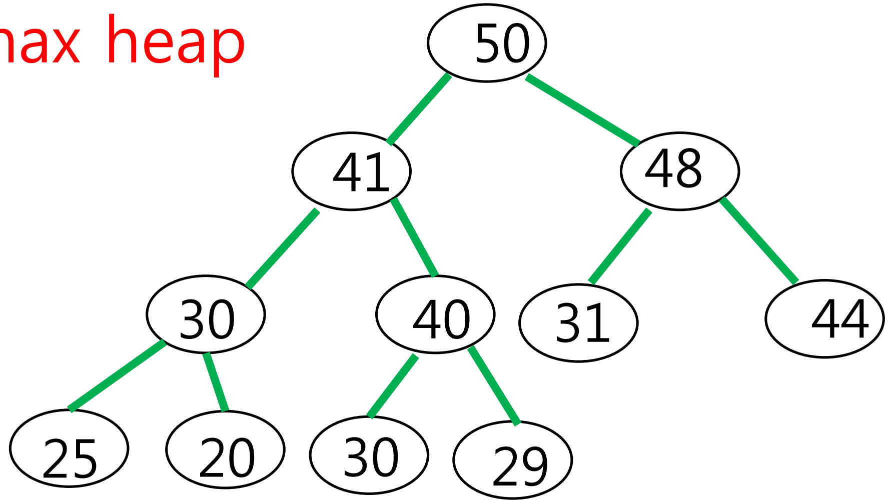
Heap

- Max (min) tree
  - Binary tee
  - Key in a node: not smaller (larger) than
    the keys in its children
- Max (min) heap
  - Complete binary tree
  - Max (min) tree
- Max (min) Priority queue
  - Elements in the queue are with priorities
  - Delete from queue
    - Element with max (min) priority
  - implementation:
    - using max (min) heap
    - Other methods

max tree



max heap



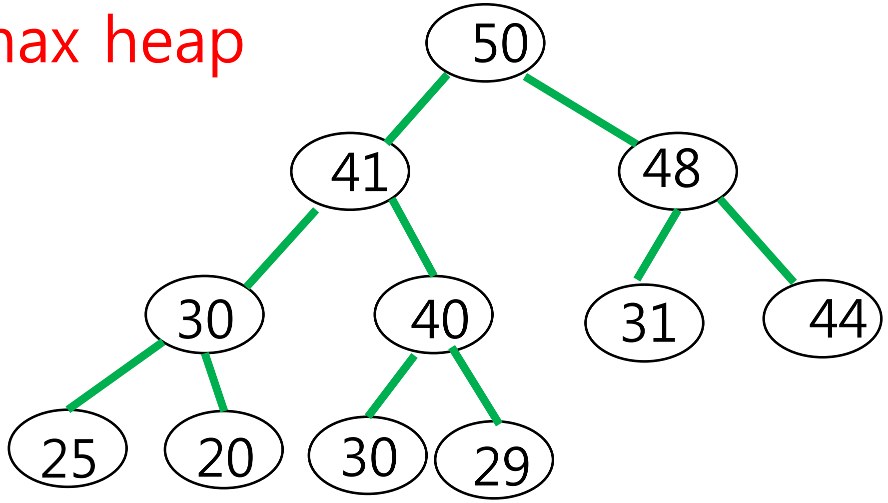
Heap의 표현

- Array representation

0	1	2	3	4	5	6	7	8	9	10	11
	50	41	48	30	40	31	44	25	20	30	29

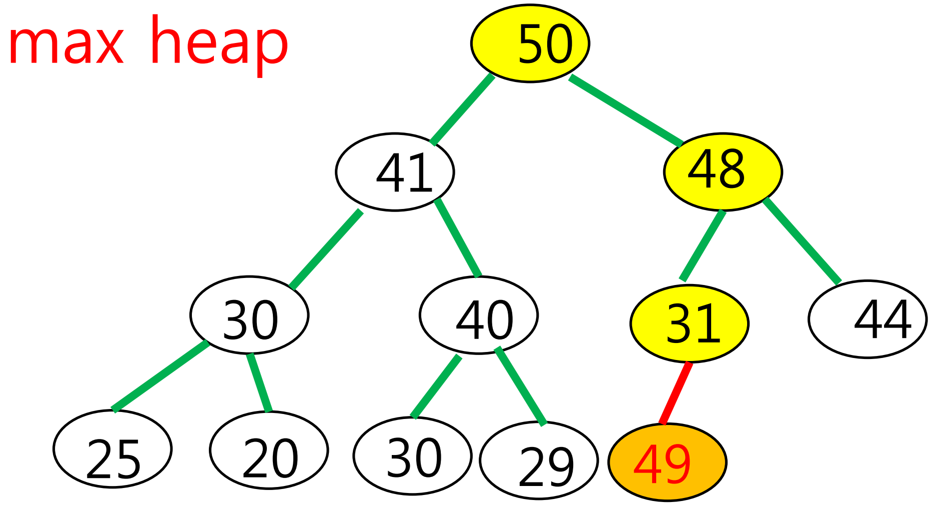
- Heap operations
  - Insert
    - Maintain heap requirements
    - Complexity: O(h) = O(log n)
  - Delete
    - Maintain heap requirements
    - Complexity: O(h) = O(log n)

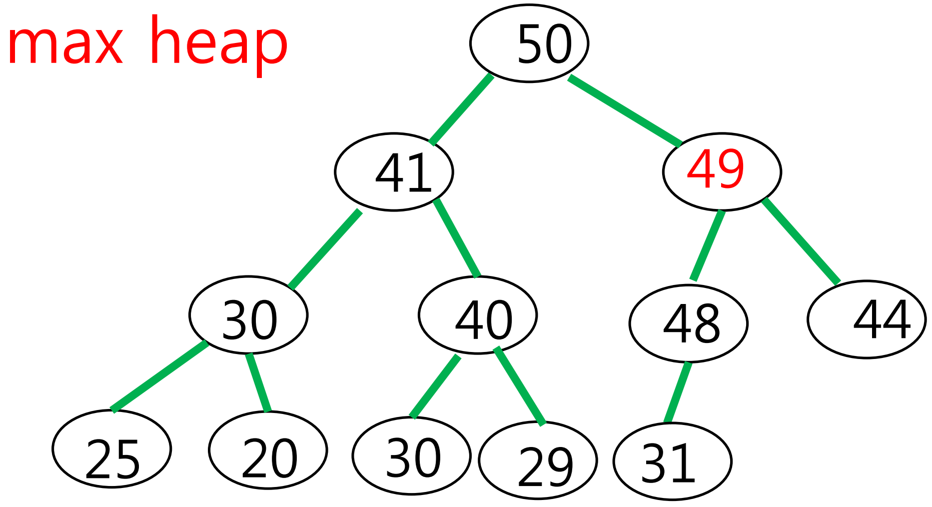
max heap



Heap: insert

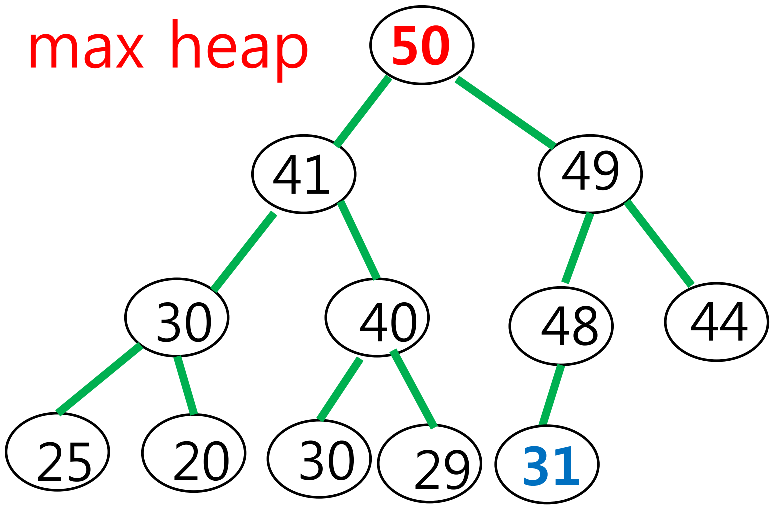
- Insert a new key into the new rightmost leaf node N
- Locate its right position along the path from N to the root
  - A[i]의 parent: A[j], j = $\left\lfloor \frac{i}{2} \right\rfloor$
- Example: insert 49

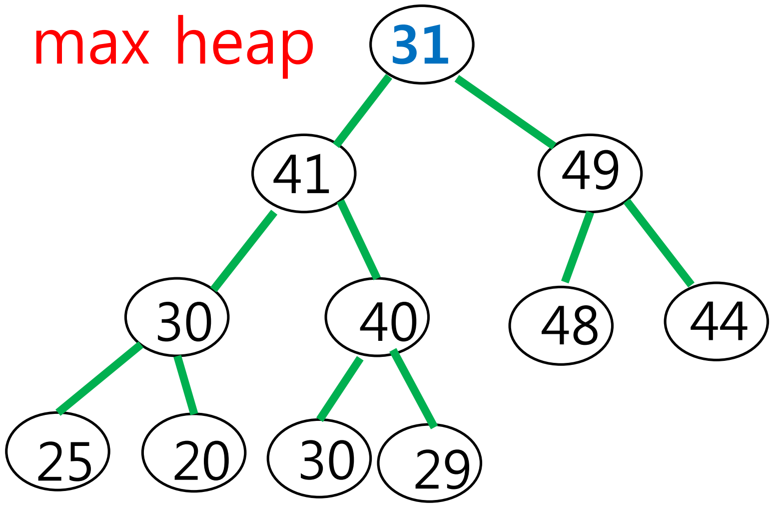


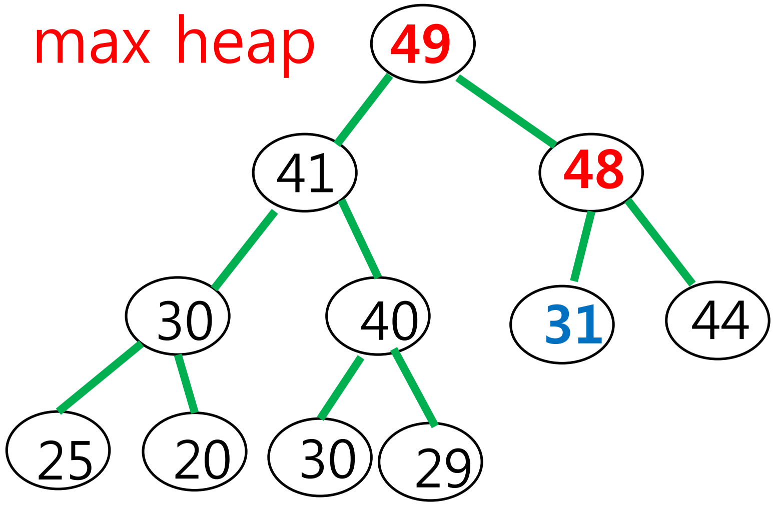


Heap: delete

- Delete the key in the root
- Set the key(root) to the key(rightmost leaf node N)
- Delete N
- Adjust to make the tree a heap







Heap sort

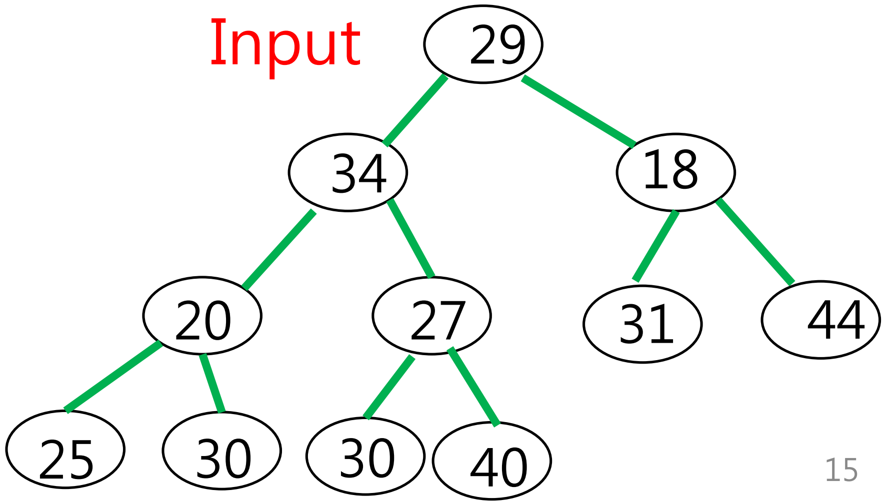
- 7장 참조
- An application of heap
- Input: array of n numbers
- Steps
    1. Heap construction: transforming the input array into a heap
    2. Swap the number in the root & the number in the rightmost leaf
    3. Adjust to make the tree a heap
    4. Repeat 2 & 3 until sorting is completed
- Output: array of n sorted numbers

Input array

0	1	2	3	4	5	6	7	8	9	10	11
	29	34	18	20	27	31	44	25	30	30	40

Output array

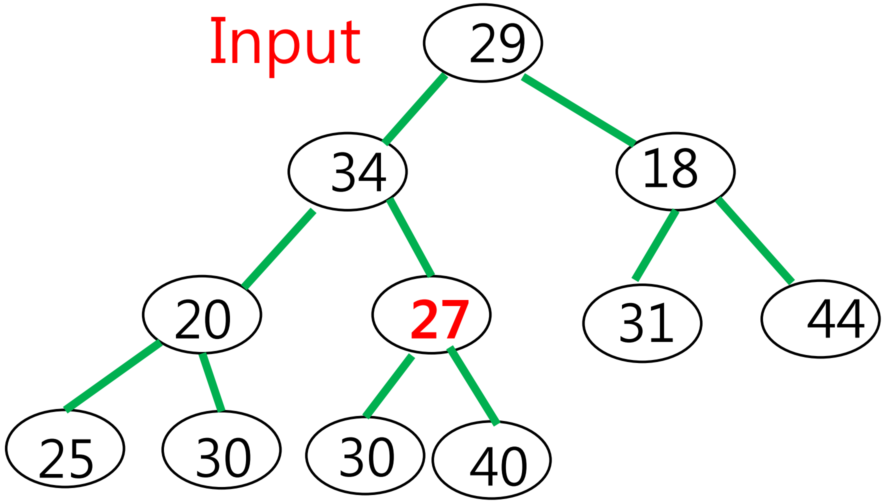
0	1	2	3	4	5	6	7	8	9	10	11
	18	20	25	27	29	30	30	31	34	40	44

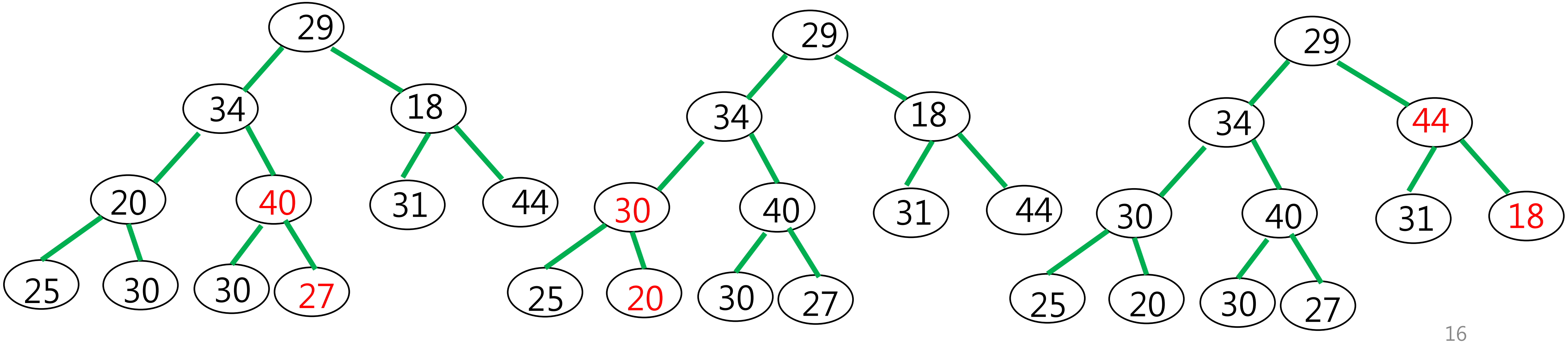


Heap sort: initial heap construction

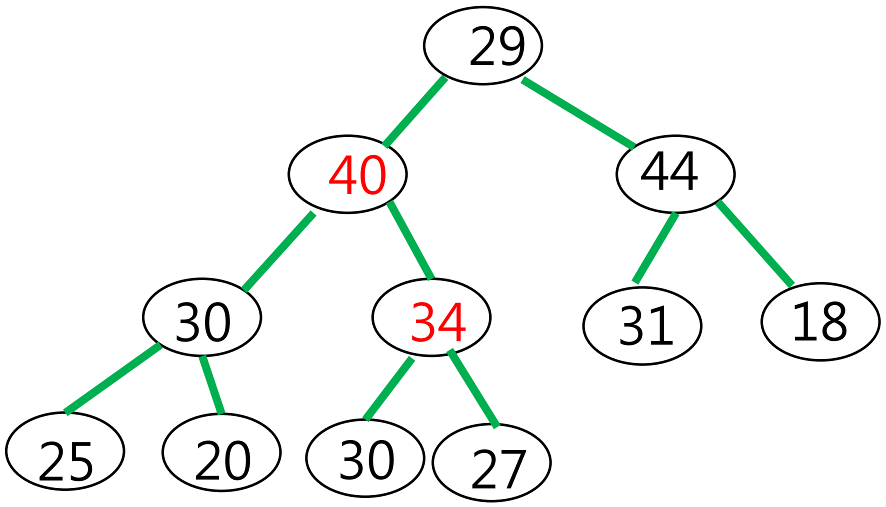
- Make every subtree of height 2 a heap
- Make every subtree of height 3 a heap
- …… until
- The entire tree becomes a heap

Input

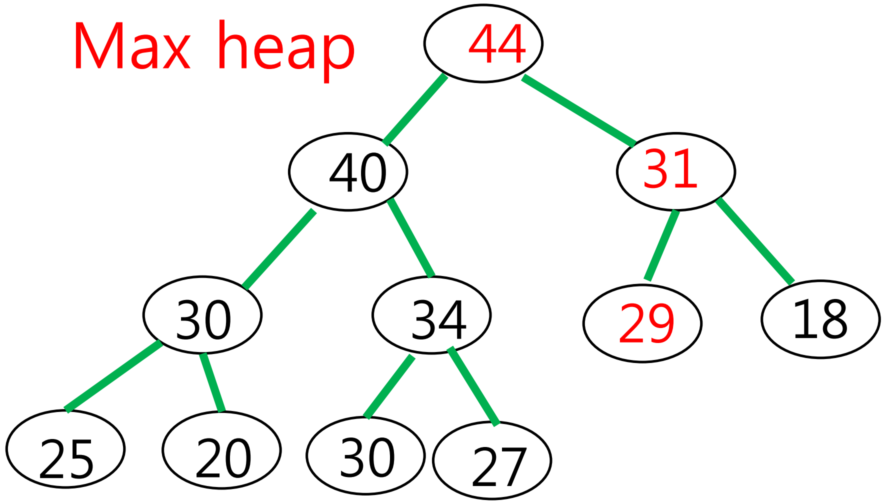




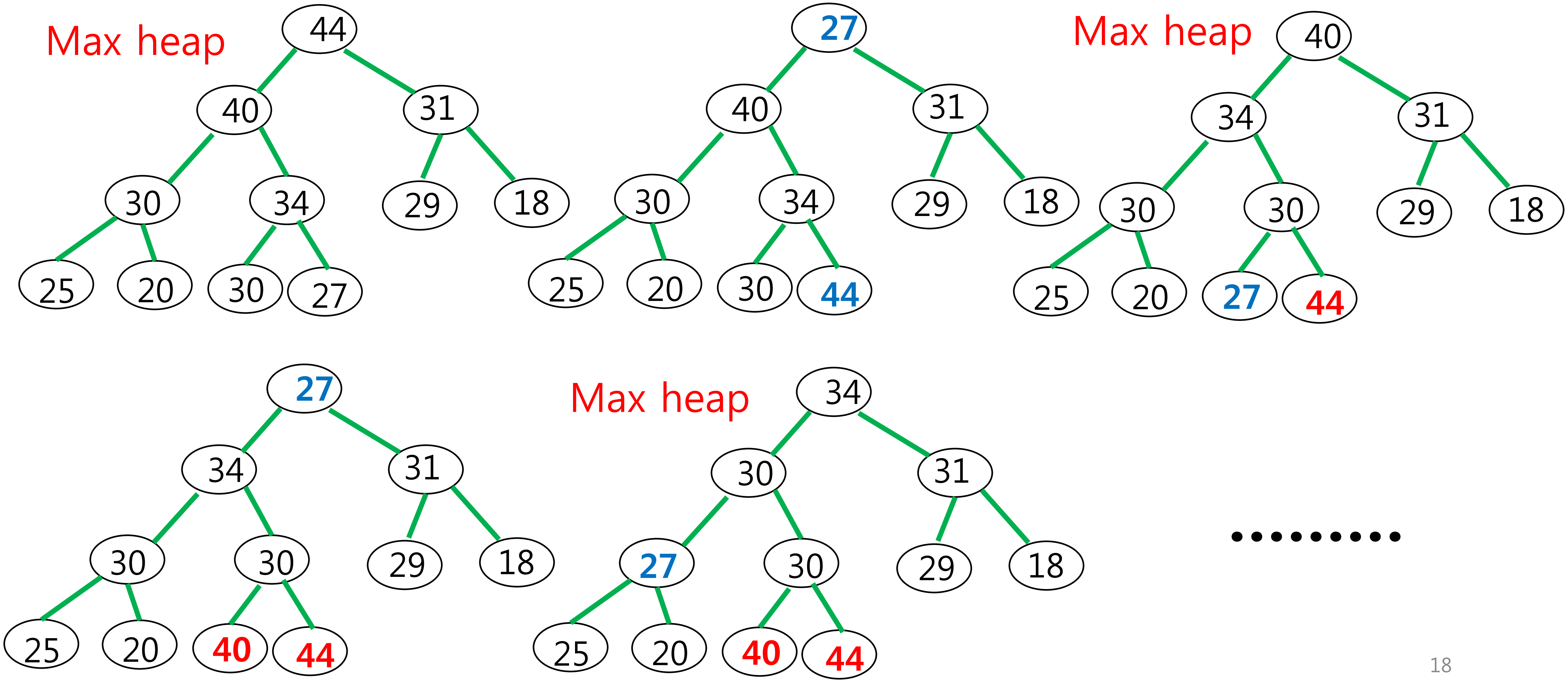
Heap sort: initial heap construction (cont'd)



Max heap



Heap sort



Binary search tree(이진 탐색 트리)

- Search key: **v**
- Recursive routine

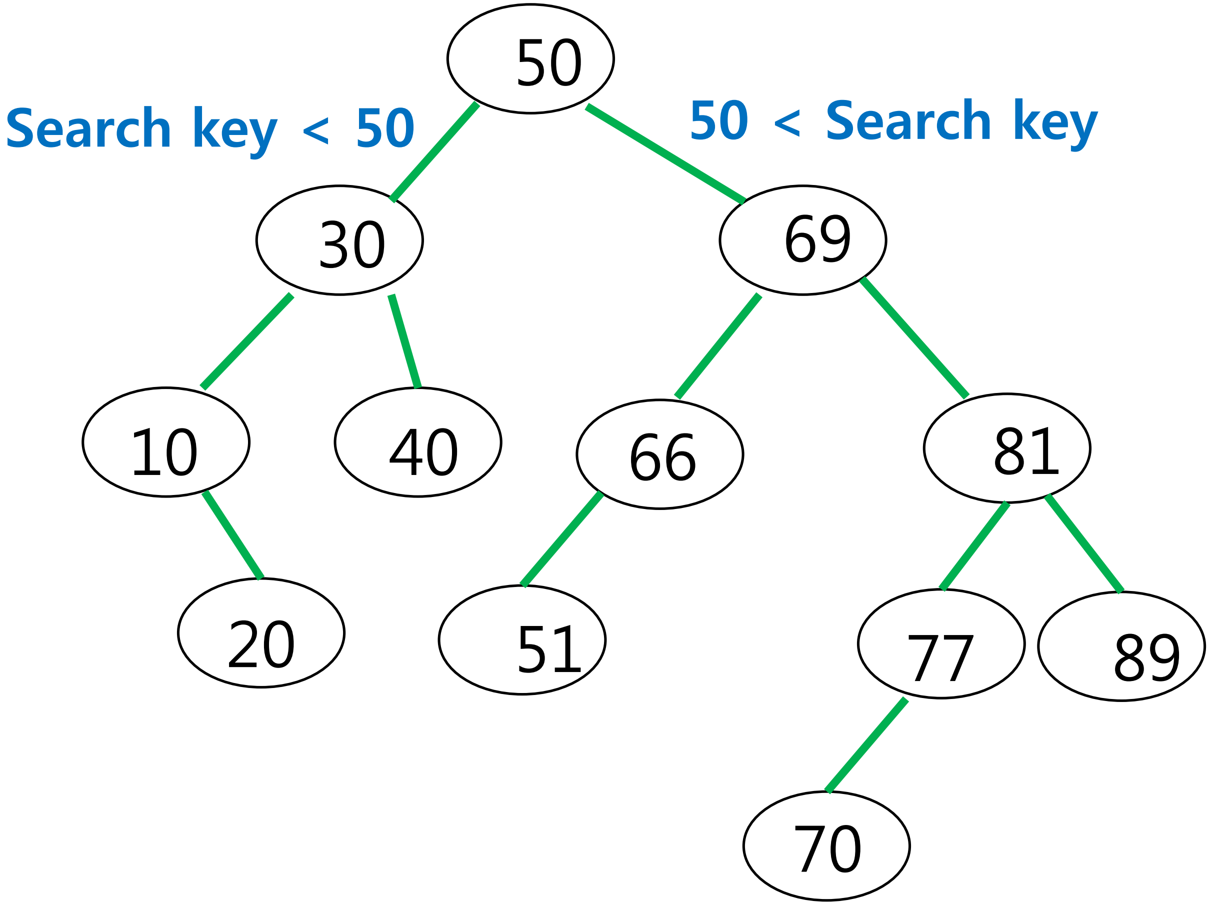
  switch(compare(**v**, key(root)) {

    case ‘<‘: search in left subtree

    case ‘=‘: found

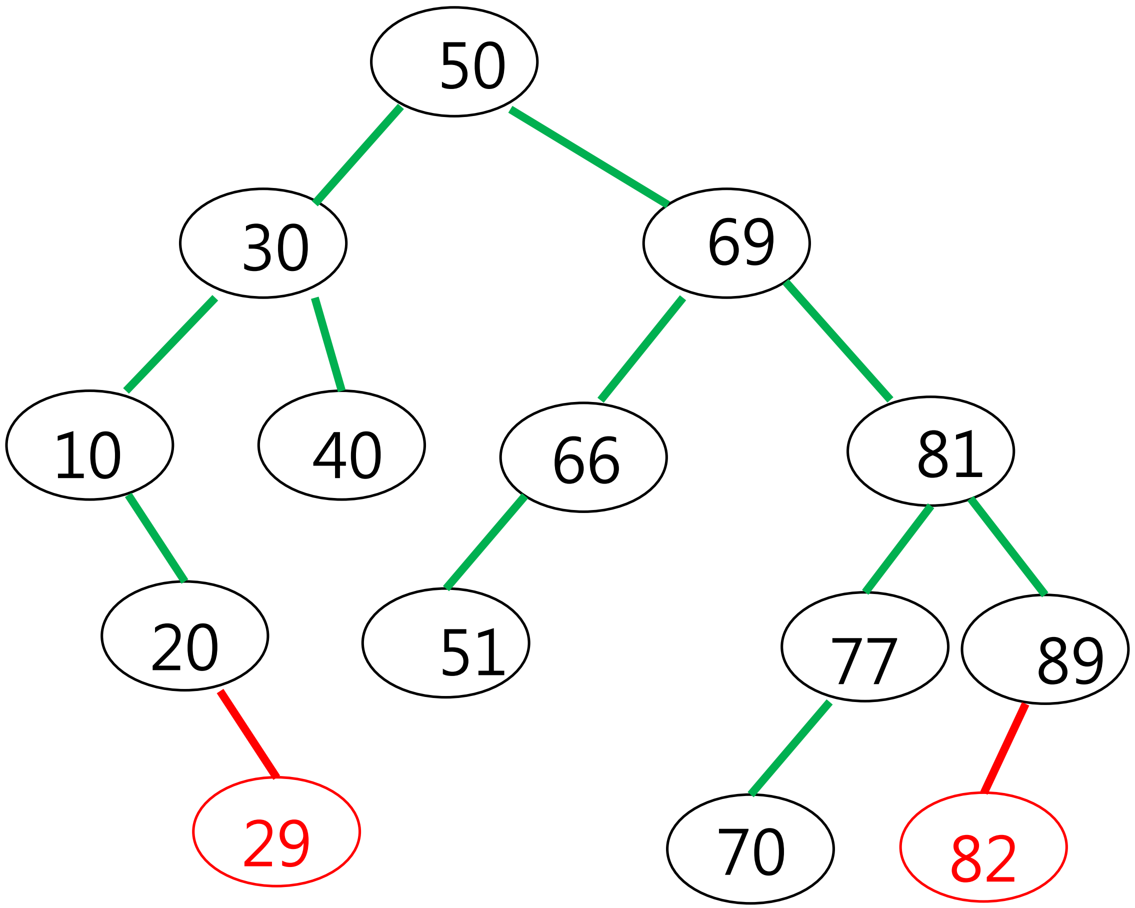
    case ‘>’: search in right subtree

  }
- Insert
- Delete



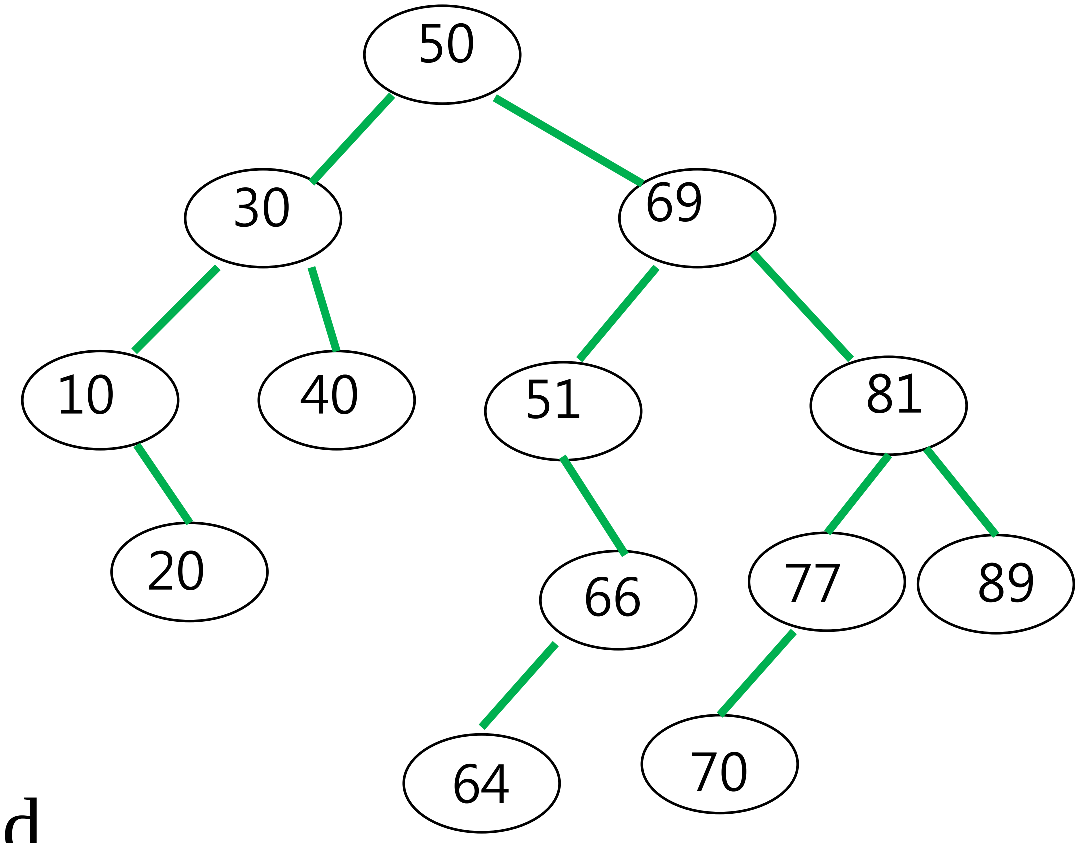
Binary search tree: insert

- Given key **v** to insert
  - Search **v** first
  - insert **v** as a leaf
- Example:
  - Insert 29
  - Insert 82



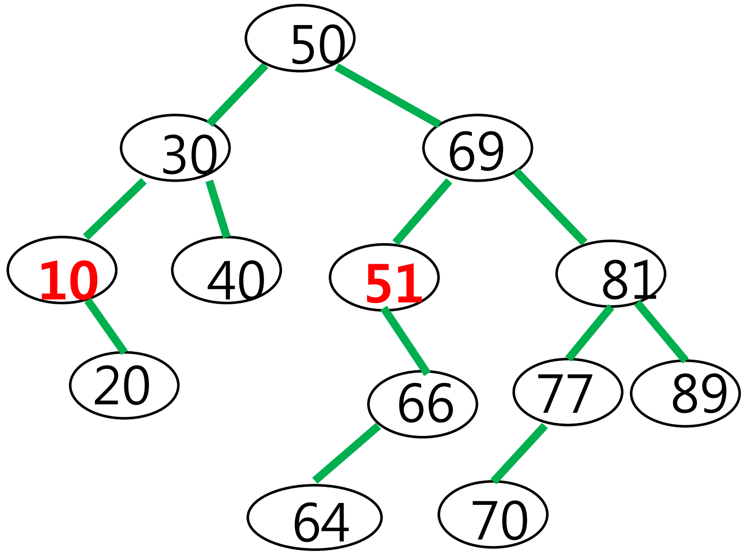
Binary search tree: delete

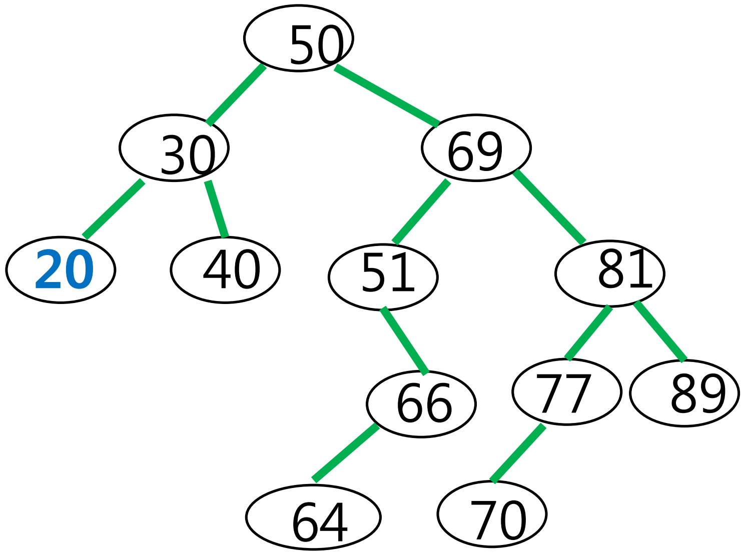
- Given key **v** to delete
  - Search **v** first
  - Cases:
    - **V**: in a leaf node
    - **V**: in a non-leaf node
      - with 1 child
      - with 2 children
- Examples
  - Delete 20 //leaf
  - Delete 51 //non-leaf with 1 child
  - Delete 69 // non-leaf with 2 children

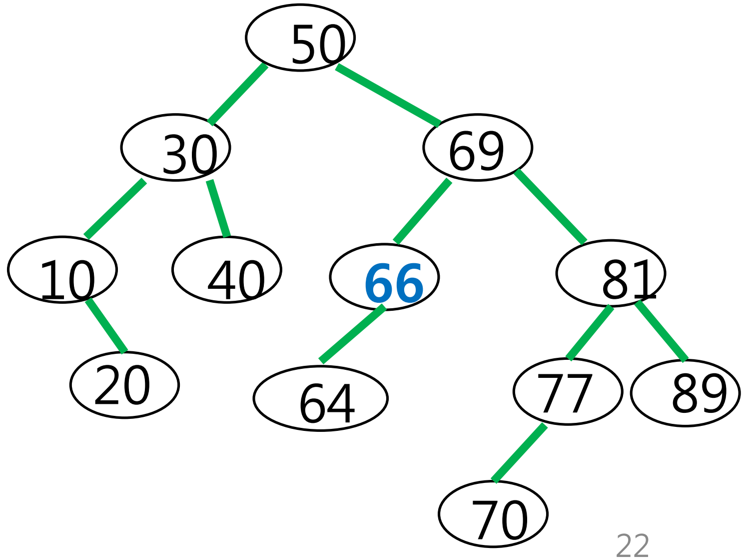


Binary search tree: delete(2)

- Delete a leaf: trivial
- Delete a non-leaf node with 1 child
  - Before delete: p → d → c
  - After delete: p → c
  - Examples
    - Delete 10
    - Delete 51

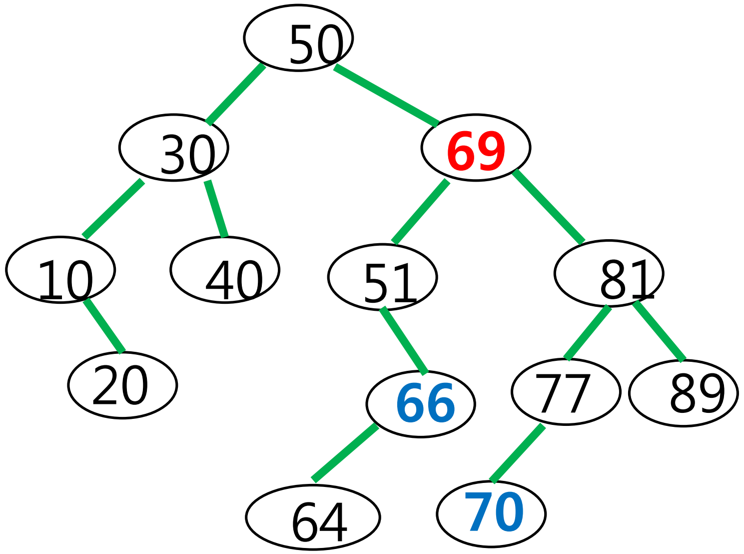


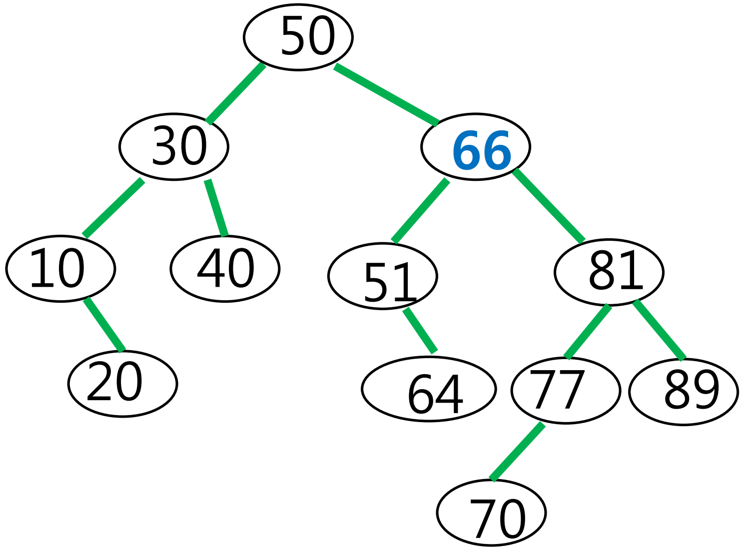


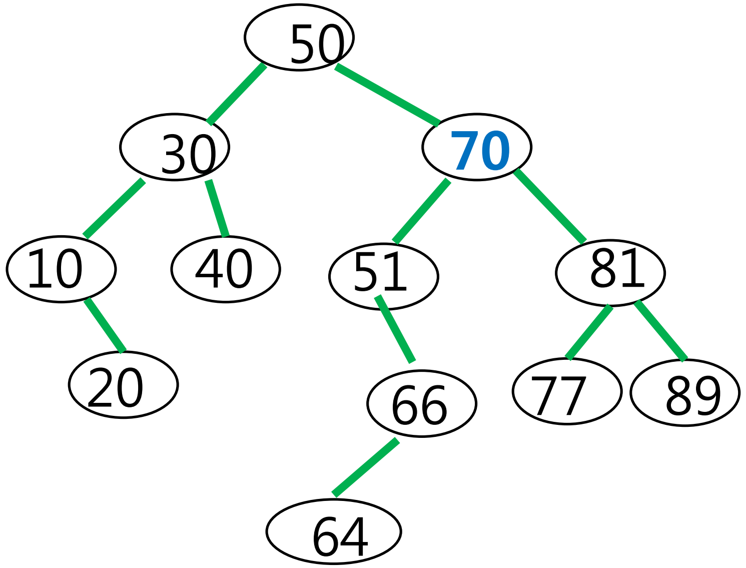


Binary search tree: delete(3)

- Delete a non-leaf node with 2 children
  - Replace deleted key with **w**
  - **w**:
    - Max key in Lsubtree or
    - Min key in Rsubtree
  - Delete **w**
  - Example
    - Delete 69

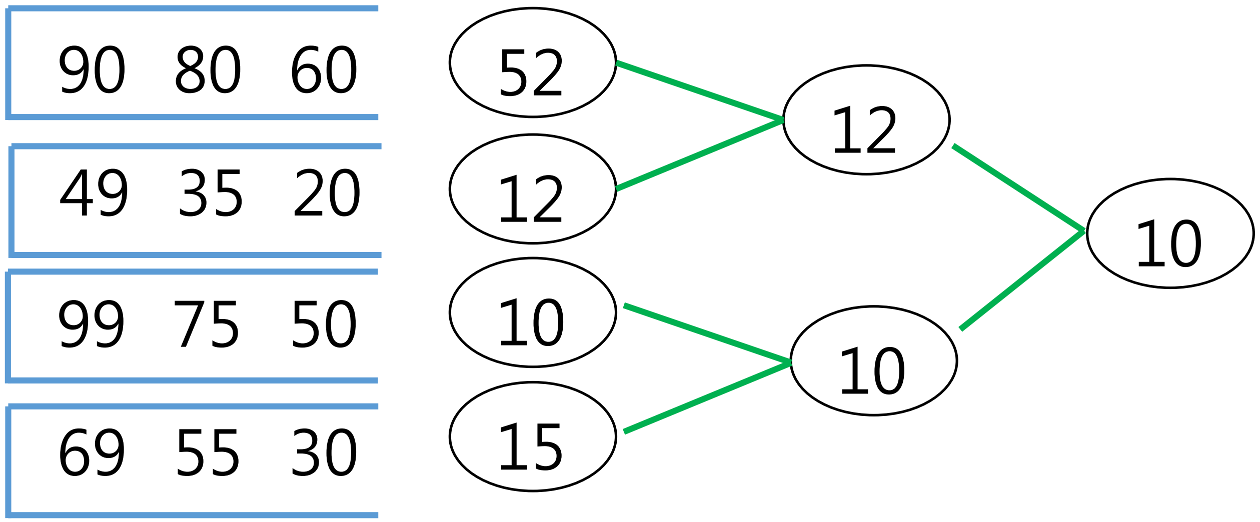






Selection tree

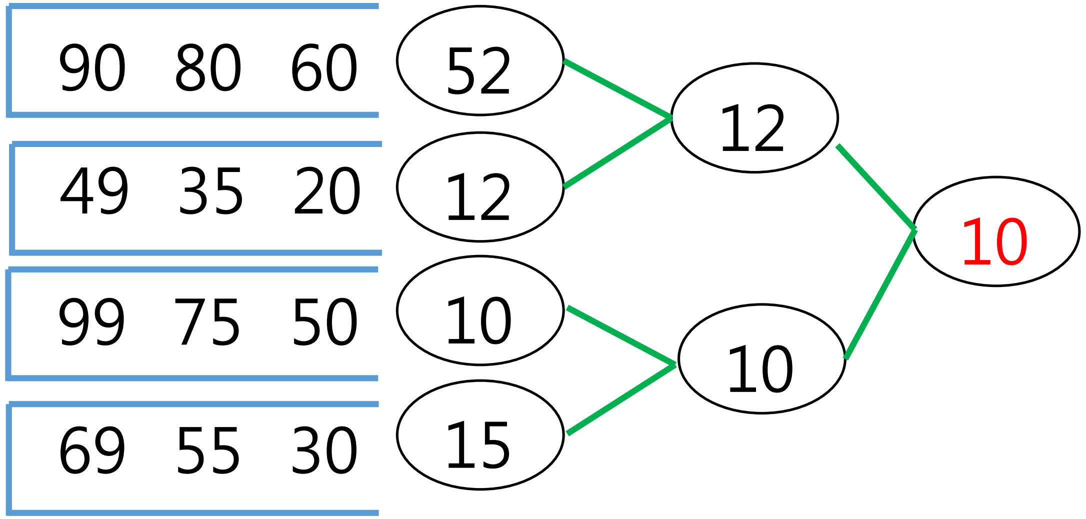
- Merge of sorted lists
- Winner tree
  - Complete binary tree
  - Leaf node: value from the first element in the sorted list
  - Non-leaf node: the winner of two children

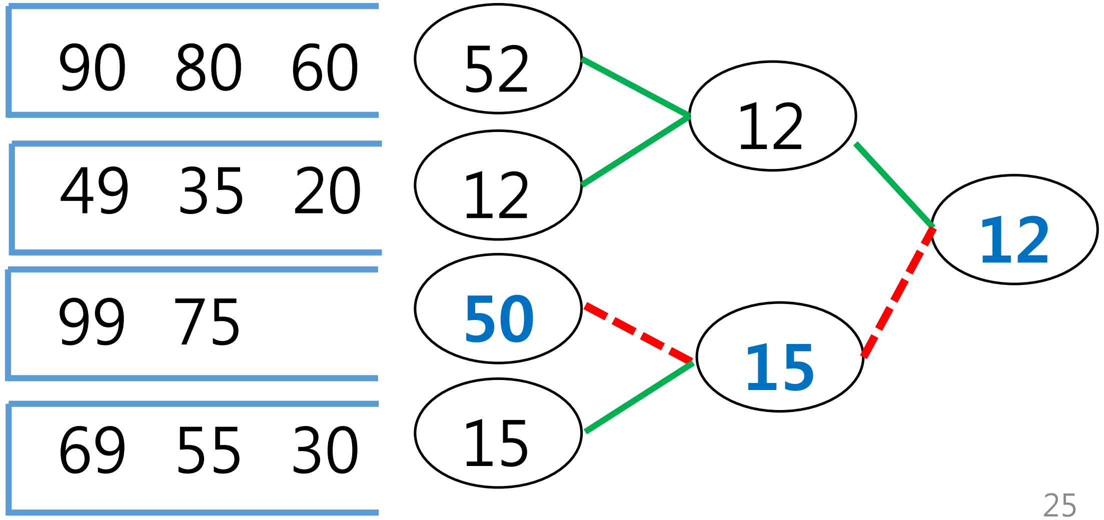


Output (merged & sorted list): 10 12 15 20 30 35 49 50 52 55 60 69 75 80 90 99

Reconstruction of winner tree

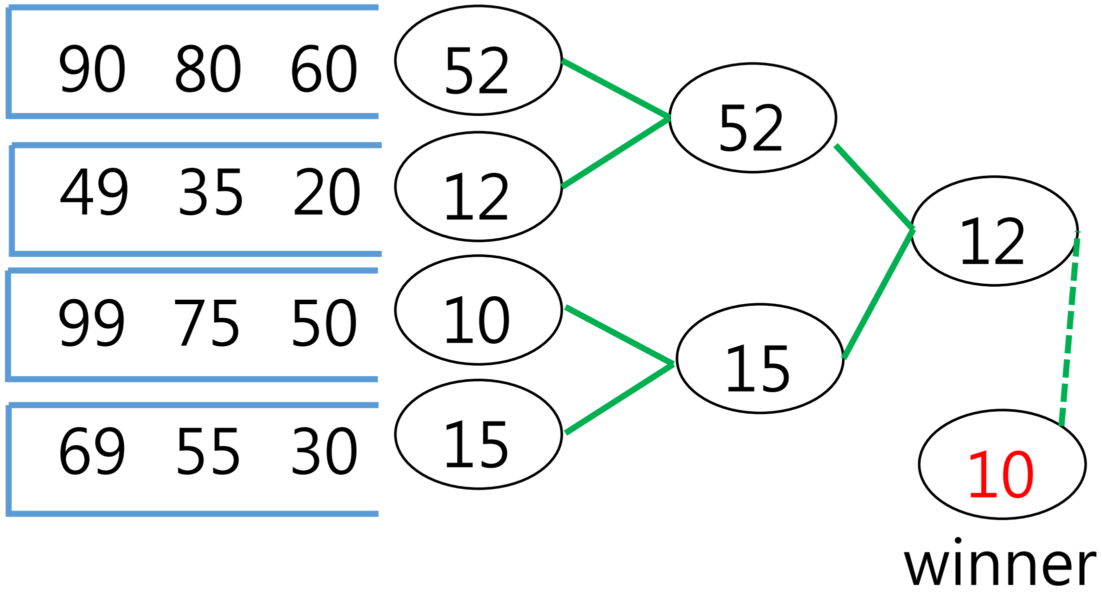
- After output of the value in the root
- Only the path from leaf (with new value) to root is updated
- Why winner tree?
  - A total of *n* values in *m* sorted lists
  - Simple merge: O(n*m) // m-1 comparisons for each output
  - Using winner tree: $O(n*\log_2 m)$

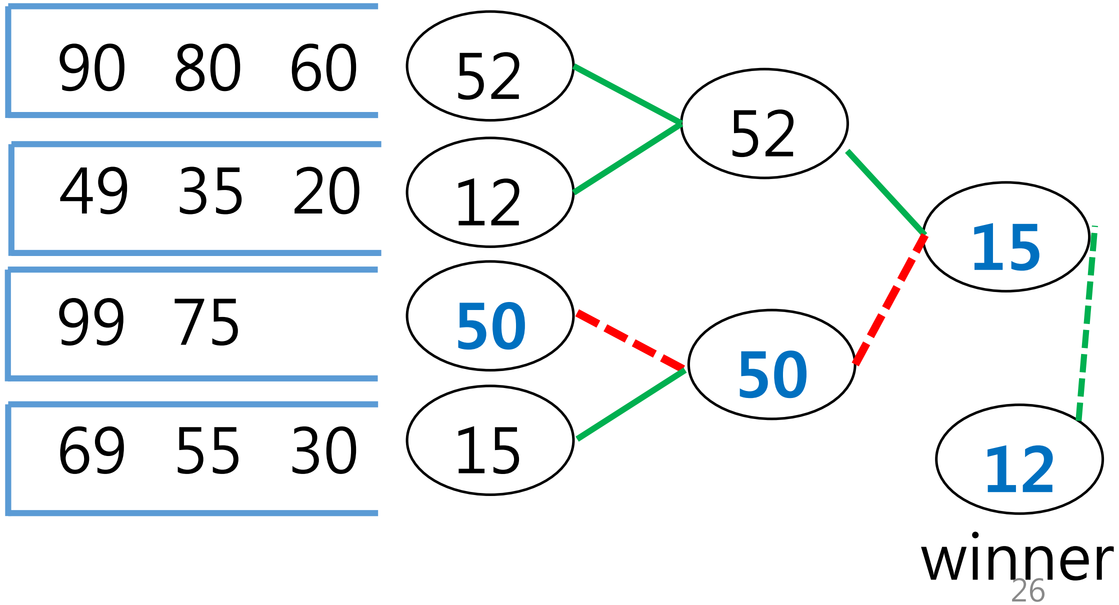




Improvement

- Loser tree
  - Non-leaf node: the loser of two children
  - Special node for the final winner
  - Reconstruction
    - Only parent (direct ancestors) needs to be accessed
    - cf. winner tree: comparison between siblings is needed





Disjoint sets

- Set S1 and S2 are disjoint if $S1 \cap S2 = \{ \}$
- Representation of disjoint sets
  - n-ary tree
  - Each node points to its parent
  - Set ID: root element
  - 1-dim array P[]: when set elements are integers >= 0
    - P[i]: parent of element I
    - Parent of the root: set to -1
- Operations
  - Find(i): returns the set ID of element i
  - Union(i, j): obtains the union of set i and set j