Announcement

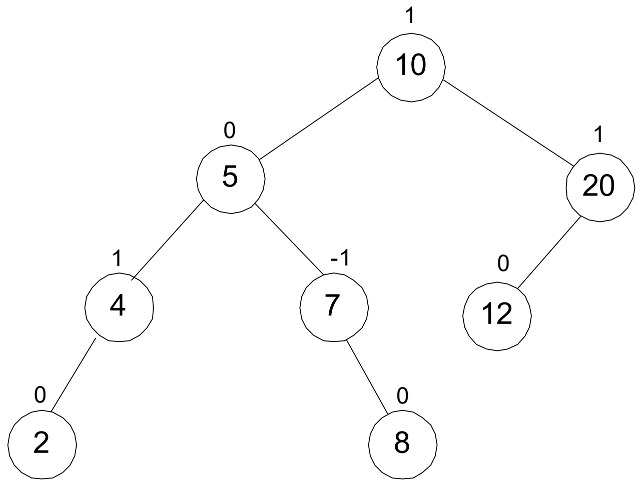
Programming Assignment #2 has been posted in Blackboard and course website

Due at 11:59pm, Tuesday, March 29

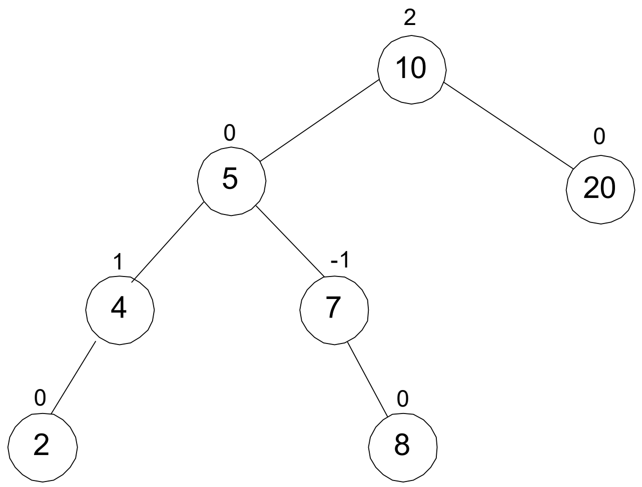
Representation Change – Balanced Binary Search Trees (AVL Trees)

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information.“

AVL tree is a balanced binary search tree.



An AVL tree



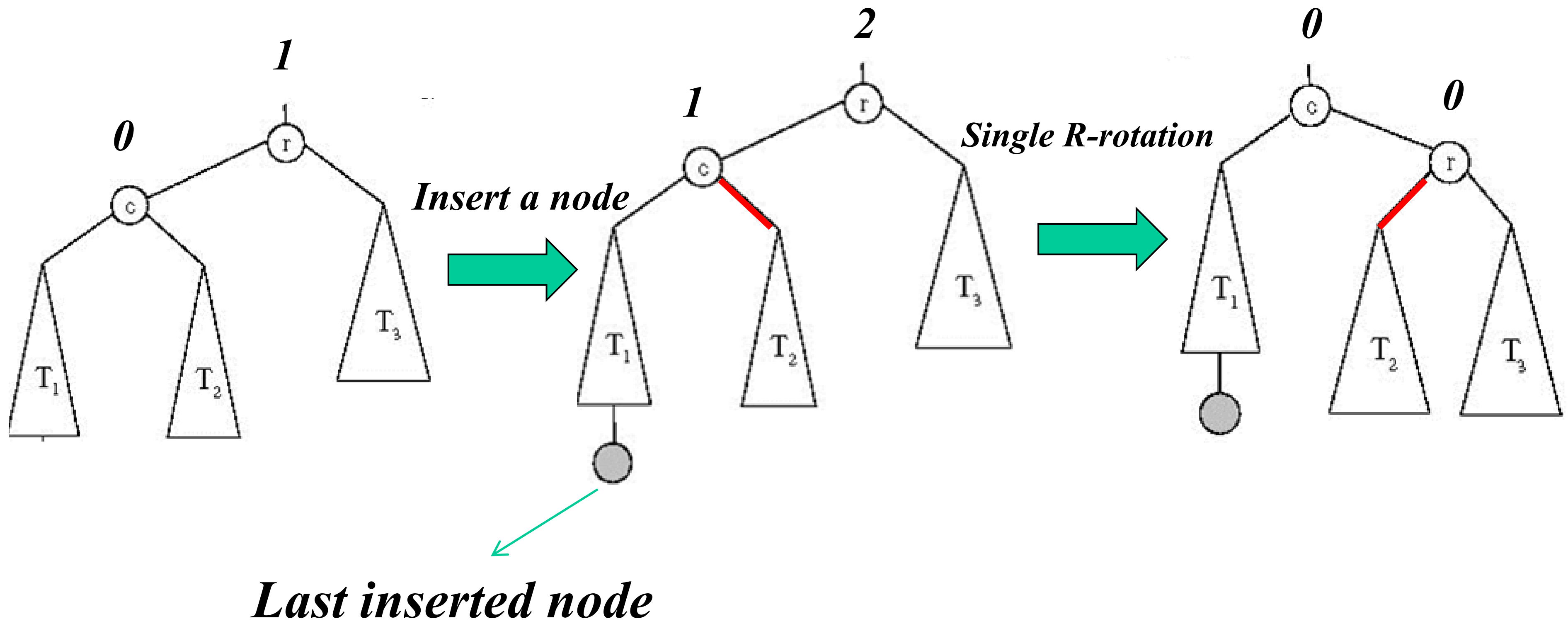
Not an AVL tree

The number shown above the node is its *balance factor*
***balance factor* = height of left subtree - height of right subtree**

For an AVL tree, |balance factor| <=1

General Case: Single R-rotation

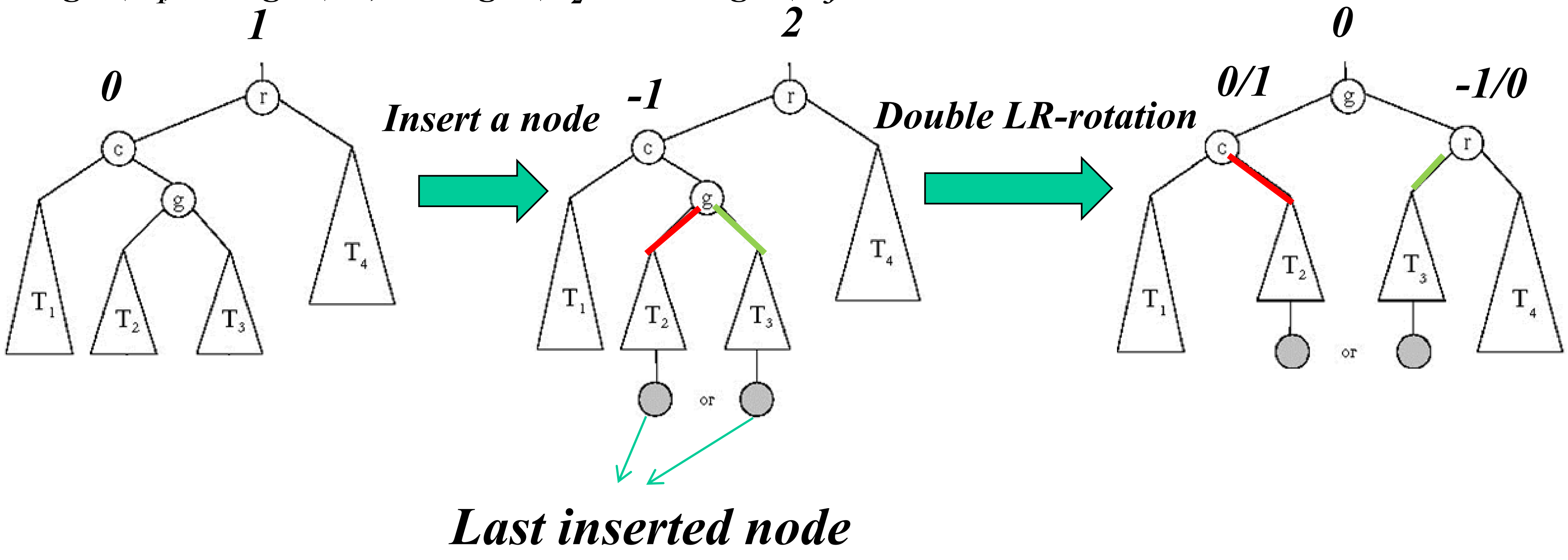
Height(T_1)=Height(T_2)=Height(T_3)



$$T_1 < c < T_2 < r < T_3$$

General Case: Double LR-rotation

Height(T_1)=Height(T4)= Height(T_2)+1= Height(T_3)+1



$$T_1 < c < T_2 < g < T_3 < r < T_4$$

Notes on AVL Tree

Rotations can be done in constant time $\Theta(1)$

Rotations guarantee an AVL tree

- A binary search tree
- A balanced tree

The height (*h*) of an AVL tree with *n* nodes is bounded by

$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2 (n+2) - 1.3277$$

average: $1.01 \log_2 n + 0.1$ **for large *n***

Operations in an AVL Tree

Searching: Θ(log*n*)

Insertion: a new node is inserted at the leaf position

- Searching Θ(log*n*)
- Rebalance (bottom up) Θ(log*n*)

Deletion:

- Searching: Θ(logn)
- Deletion:
  - A leaf or a non-leaf node with only one child, remove it. Θ(1)
  - Otherwise, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node. Θ(logn)
- Rebalance Θ(log*n*)

Drawbacks: need rotation frequently to rebalance the tree

Other Search Trees

Self-balanced BST

- Red-black trees (height of subtrees is allowed to differ by up to a factor of 2: $\frac{h_l}{h_r} \leq 2$ or $\frac{h_r}{h_l} \leq 2$)

Self-optimized BST

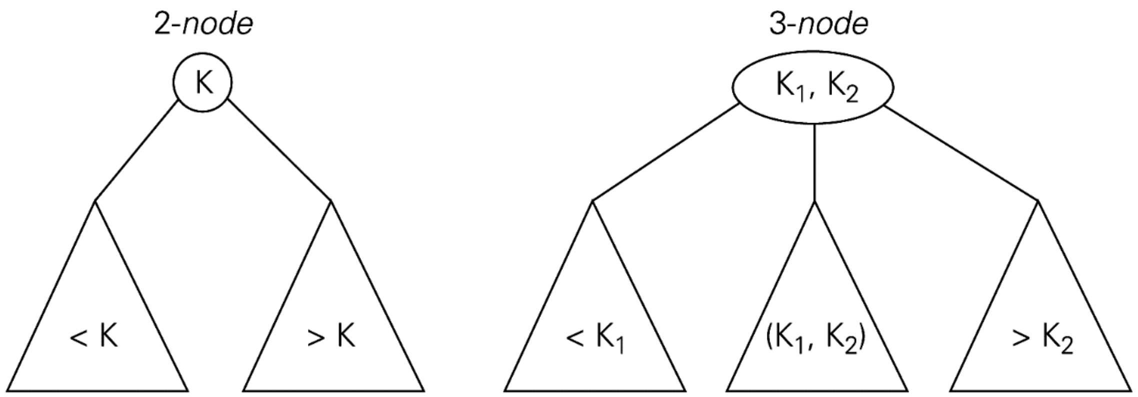
- Splay trees: move the recent visited vertex to root so that recently accessed elements are quick to access again

Multiway search trees

- 2-3 trees, 2-3-4 trees and B-trees (not a binary tree!)
  - allow more than one key in a node of a search tree
  - a node is called an *n*-node if it has at most $n - 1$ ordered keys
  - all leaves are on the same level (perfectly balanced)
  - In practice, parents are for indexing, leaf nodes for storing record

2-3 Tree – A Multiway Search Tree

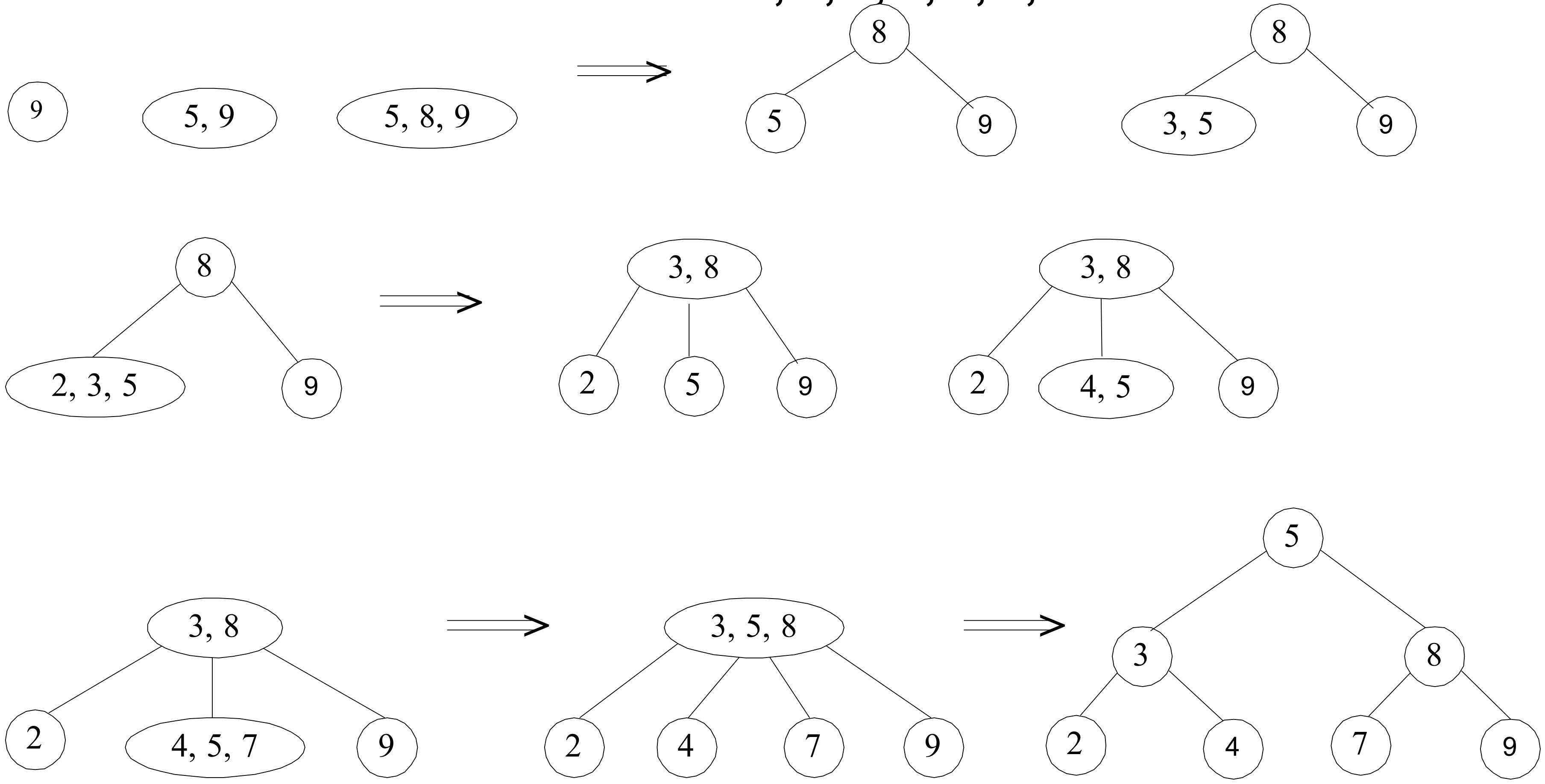
- **A search tree may have 2-node and 3-node**
- **Height balanced – all leaves are on the same level**



- **Constructed by successive insertions of keys**
- **A new key is always inserted into a leaf of the tree. If the leaf is a 3-node (with two keys) already, it's split into two with the middle key promoted to the parent.**

An Example of 2-3 Tree Construction

Construct a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7



Note on 2-3 Tree

- **Height of the tree $\log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1$**
- **Time efficiency**
  - Search, insertion, and deletion are in $\Theta(\log n)$

The idea of 2-3 tree can be generalized by allowing more keys per node

- 2-3-4 trees
- B-trees

Representation Change – Heap and Heapsort

Recall: A *priority queue* is the ADT (abstract data type) of an ordered set with the operations:

- find element with highest priority
- delete element with highest priority
- insert element with assigned priority

Applications:

- Scheduling in computer operating system, traffic management, communication networks
- Critical data structure for implementing many algorithms
  - Prim's algorithm
  - Dijkstra's algorithm
  - Huffman coding

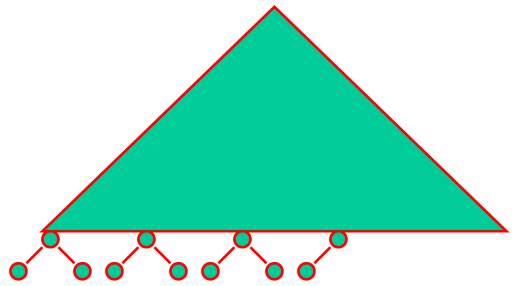
Heaps are very good for implementing priority queues

Representation Change – Heap and Heapsort

<u>Definition:</u>

A *heap* is a binary tree with the following conditions:

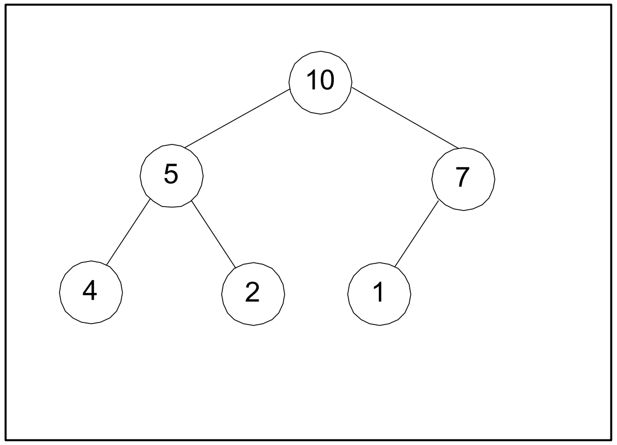
(1) it is essentially complete: all its levels are full except possibly the last level, where only some rightmost leaves may be missing

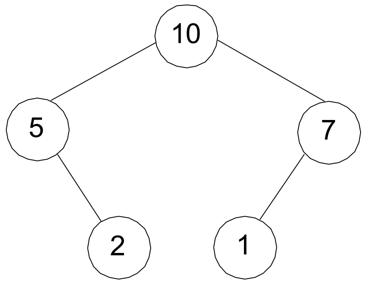


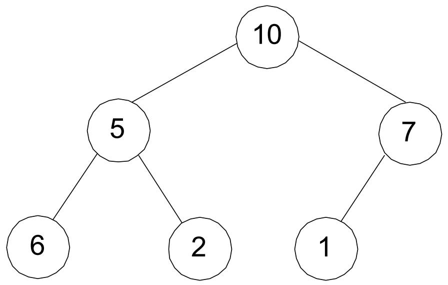
(2) The key at each node is ≥ keys at its children

An Example:

Which tree is a heap, why?







Definition implies

- **Given *n*, there exists a unique binary tree with *n* nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$**
- **Parent dominate**
  - Maxheap
    - The parent has a value larger or equal than its children
    - The root has the largest key
  - Don't confuse it with a BST
    - There is no relationship between the left and right subtrees
- **The subtree rooted at any node of a heap is also a heap**
- **Priority queue**
  - A useful structure in many algorithms

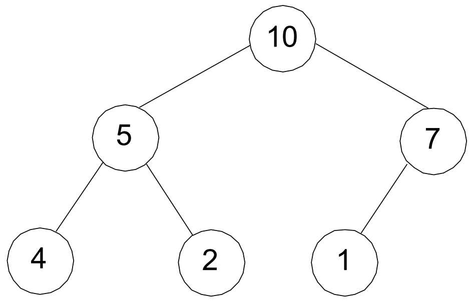
Heap Implementation

A heap can be implemented as an array *H*[1..*n*] by recording its elements in the top-down left-to-right fashion.

Leave *H*[0] empty

First $\lfloor n/2 \rfloor$ elements are parental node keys and the last $\lceil n/2 \rceil$ elements are leaf keys

i-th element's children are located in positions 2*i* and 2*i*+1



Index	0	1	2	3	4	5	6
value		10	5	7	4	2	1

Therefore

A heap with *n* nodes can be represented by an array $H[1..n]$ **where**

$$H[i] \geq \max\{H[2i], H[2i+1]\}, \quad \textbf{for } i = 1,2,\ldots,\lfloor n/2 \rfloor$$

If 2*i*+1>*n* (the last parent only has one child), just *H*[*i*]≥*H*[2*i*] needs to be satisfied

Heap operations include

- Heap construction
- Insert a new key into a heap
- Delete the root of a heap
- Delete an arbitrary key from a heap

Important! – after any such operations, the result must be still a heap

Heap Construction -- Bottom-up Approach

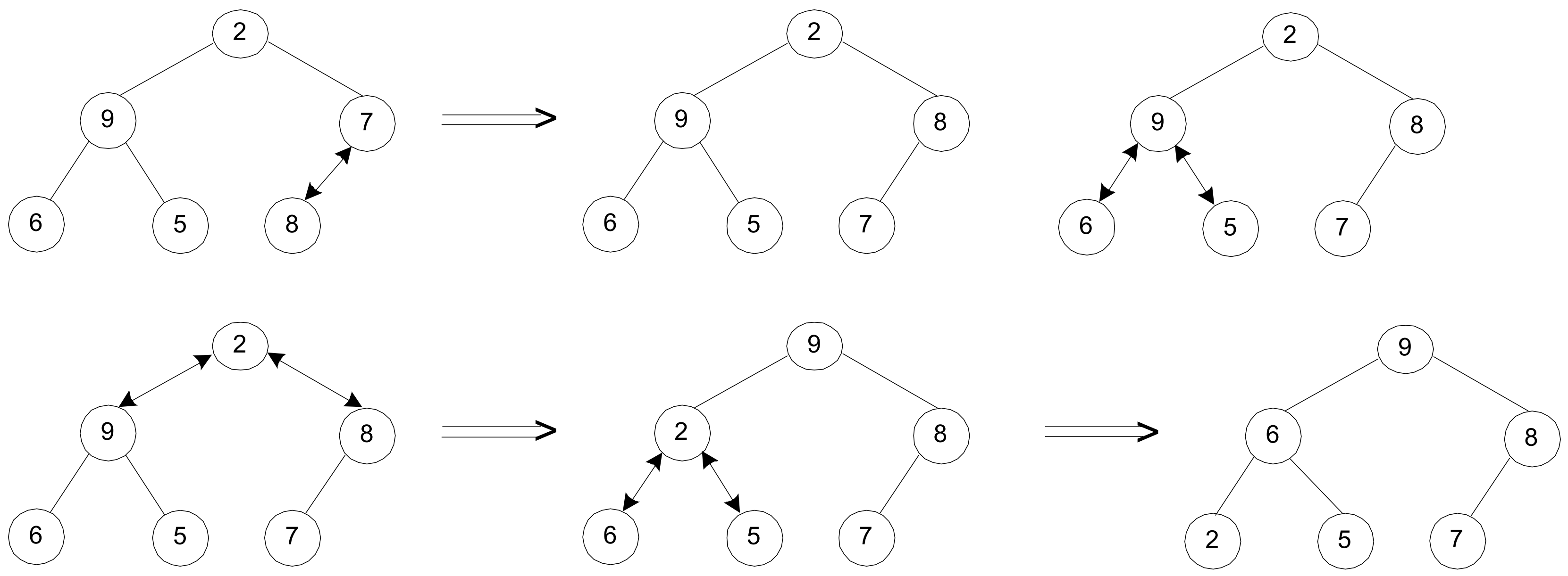
Heap Construction -- Construct a heap for a given list of keys

Initialize an *essentially complete* binary tree with the given order of the *n* keys

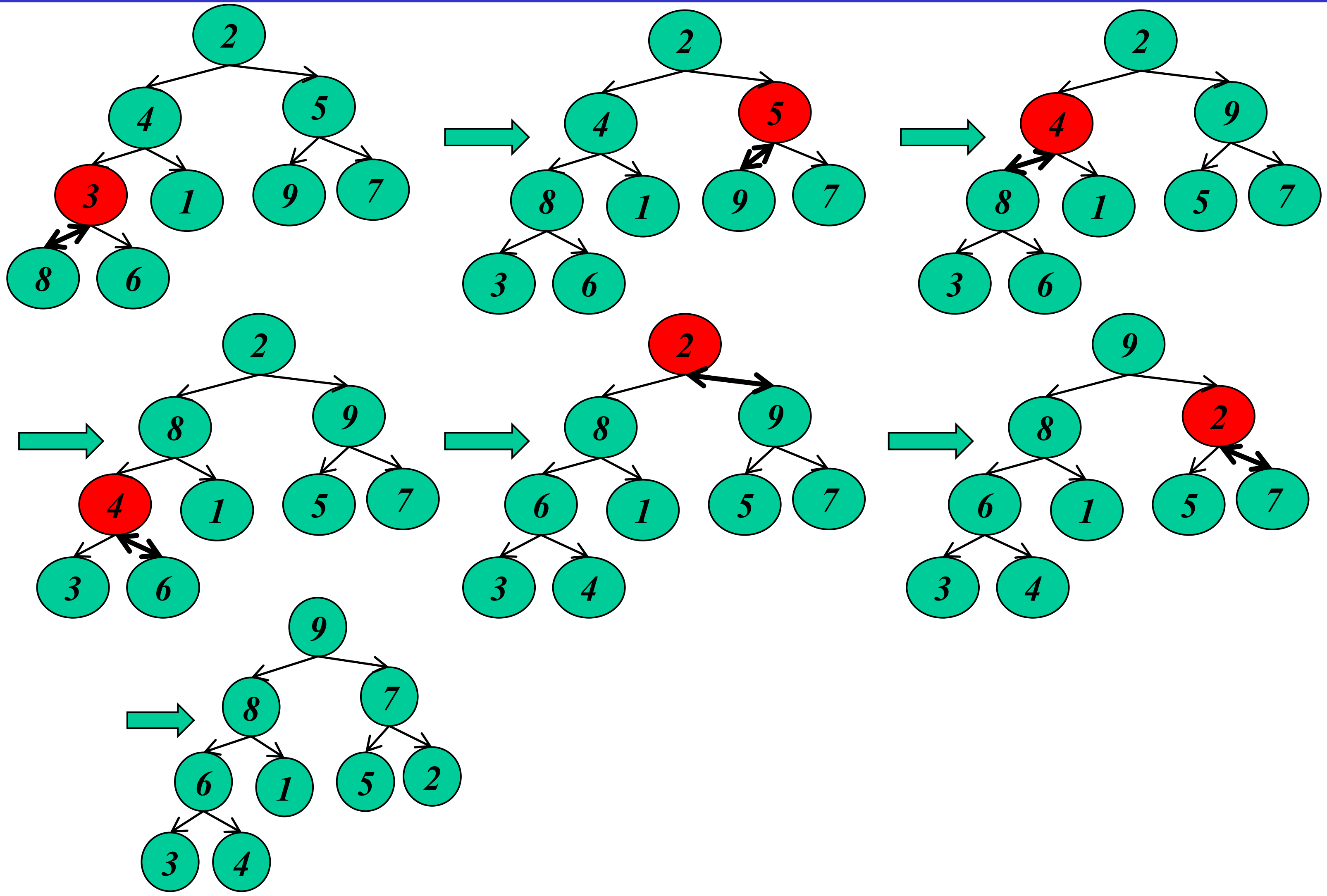
- Starting from the last parental node to the first parental node, check whether $H[i] \geq \max\{H[2i], H[2i+1]\}$
- If not, swap parental and child keys to satisfy this requirement

Note that if a certain parental key is swapped with one child, we need to keep checking this key at its new location until no more swap is required or a leaf key is reached

An Example:



Another Example: {2 4 5 3 1 9 7}



HeapBottomUp Code

```
**Algorithm** $HeapBottomUp(H[1..n])$
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
**for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i$;   $v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    **while not** $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        **if** $j < n$   //there are two children
            **if** $H[j] < H[j+1]$   $j \leftarrow j + 1$ $\longrightarrow$ *Use the larger children*
        **if** $v \geq H[j]$
            $heap \leftarrow$ **true**
        **else** $H[k] \leftarrow H[j]$;  $k \leftarrow j$ $\longrightarrow$ *Keep checking the key*
    $H[k] \leftarrow v$
```

Swap the parent key with the larger children

Algorithm Efficiency

In the worst case, the tree is complete, i.e, n=2^k-1

The height of the tree $h = \lfloor \log_2 n \rfloor = k-1$

In the worst case, each key on level *i* of the tree will travel to leaf level *h*

Two key comparisons (finding the larger children and determine whether to swap with the parental key) are needed to move down one level (level *i* has 2^i keys)

The level above the leaf level

$$T_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\substack{\text{all keys}\\ \text{in level } i}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n-\log_2(n+1))$$

$\in \Theta(n)$

The root

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\sum_{i=1}^{h} i2^i = (h-1)2^{h+1} + 2$$

Heap Construction – Top-down Approach

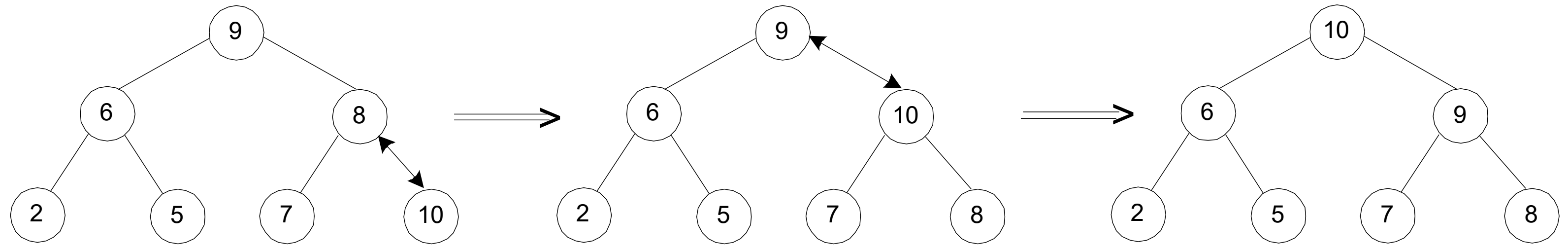
It is based on the operation of inserting a new item to an existing heap, and maintain a heap

Inserting a new key to the existing heap (analogue to insertion sort) is achieved by

- Insert the new key as the last element in array *H* as a leaf of the binary tree
- Compare this new key to its parent and swap if the parental key is smaller
- If such a swap happened, repeat this for this key with its new parent until there is no swap happened or it gets to the root

An Example:

Insert a new key 10 into the heap with 6 keys [9 6 8 2 5 7]



Note

The time efficiency of each insertion algorithm is $O(\log n)$ because the height of the tree is $\Theta(\log_2 n)$

A heap can be constructed by inserting the given list of keys into the heap (initially empty) one by one.

Construct a heap from a list of *n* keys using this insertion algorithm, in the worst case, will take the time

$$\sum_{i=1}^{n} \log i \in \Theta(n \log n)$$

Bottom-up Versus Top-down

Time efficiency:

- **Bottom-up** $O(n)$
- **Top-down** $O(n \log n)$

$\Rightarrow$ The top-down heap construction is less efficient than the bottom-up heap construction

Space:

- **Bottom-up: fixed size *n+1* array**
- **Top-down: need to allocate array every time of insertion**

When we use top-down?

The application of priority queue.

Delete an Item From the Heap

Let’s consider only the operation of deleting the root’s key, i.e., the largest key

It can be achieved by the following three consecutive steps

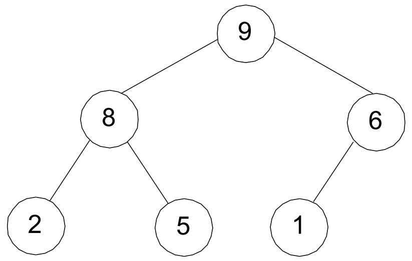
(1) Exchange the root’s key with the last key *K* of the heap

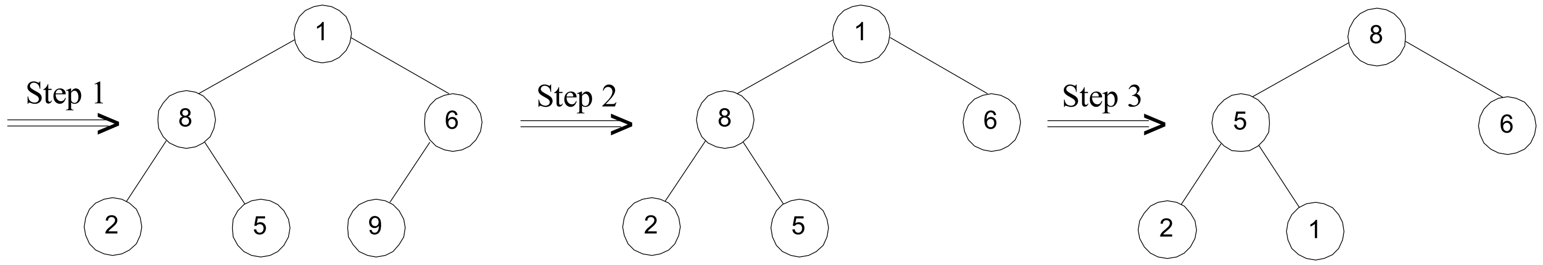
(2) Decrease the heap’s size by 1 (remove the last key)

(3) “Heapify” the remaining binary tree by shifting the key *K* down to its right position using the same technique used in bottom-up heap construction (compare key K with its child and decide whether a swap with a child is needed. If no, the algorithm is finished. Otherwise, repeat it with its new children until no swap is needed or key *K* has become a leaf)

An Example:

Delete the largest key 9





Notes On Key Deletion

The required # of comparison or swap operations is no more than the height of the heap. The time efficiency of deleting the root's key is then $O(\log n)$

Question: How to delete an arbitrary key from the heap?

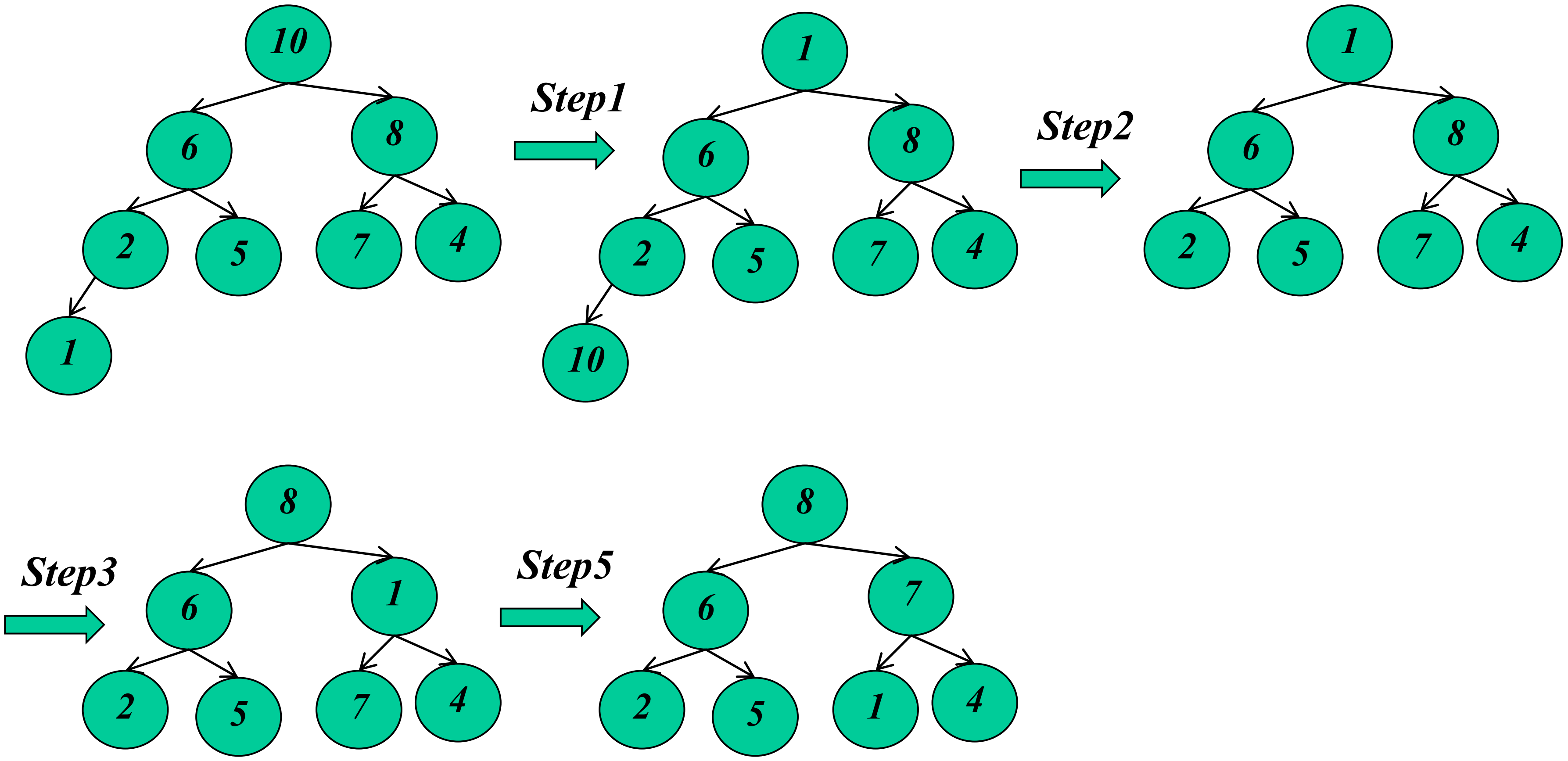
- Search for the key $O(n)$
- It is similar to the three-step root-deletion operation $O(\log n)$
  - Exchange with the last element *K*
  - "Heapify" the new binary tree. But it may be shift up or down, depending on the value of *K*

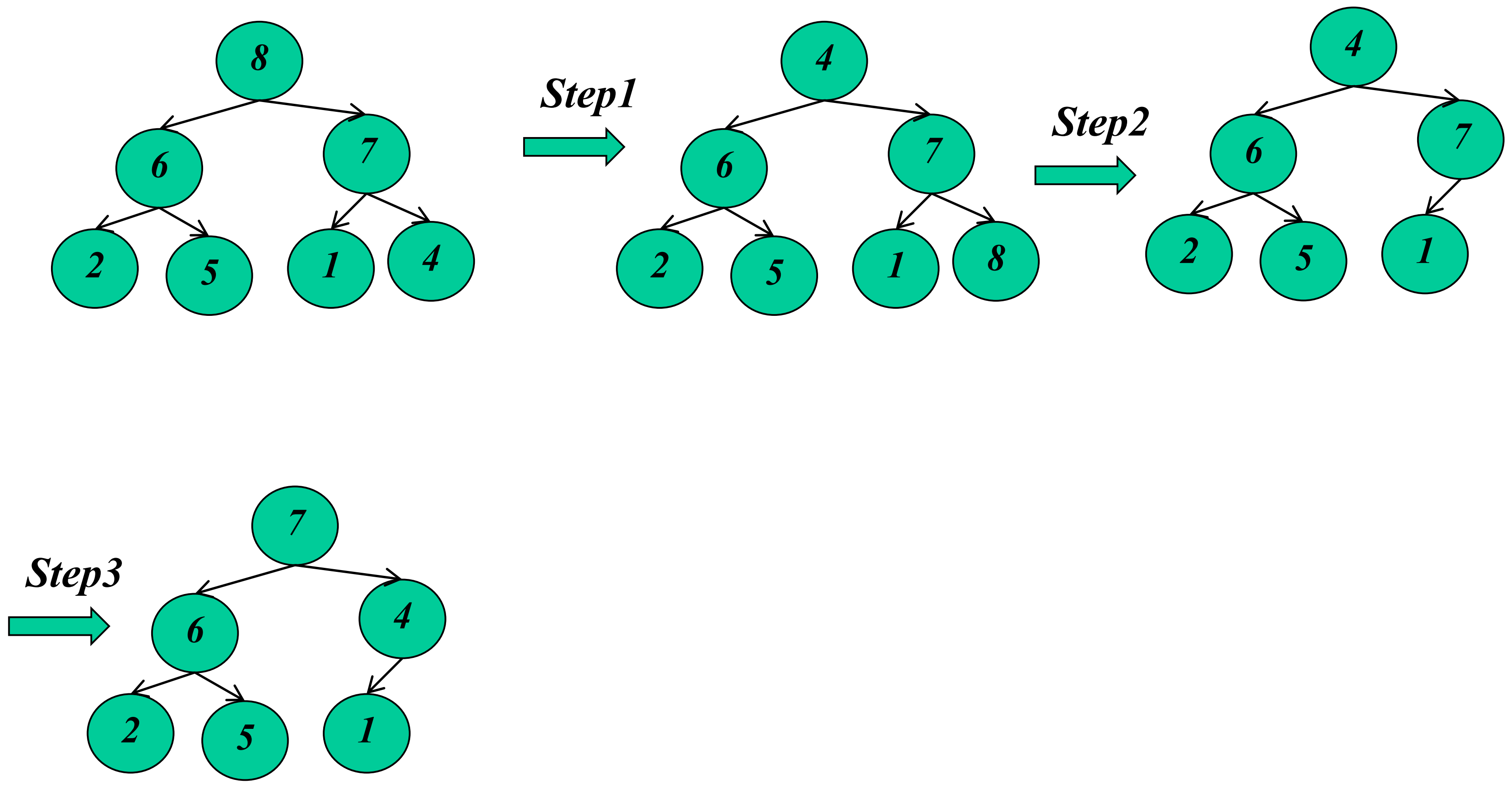
Heapsort

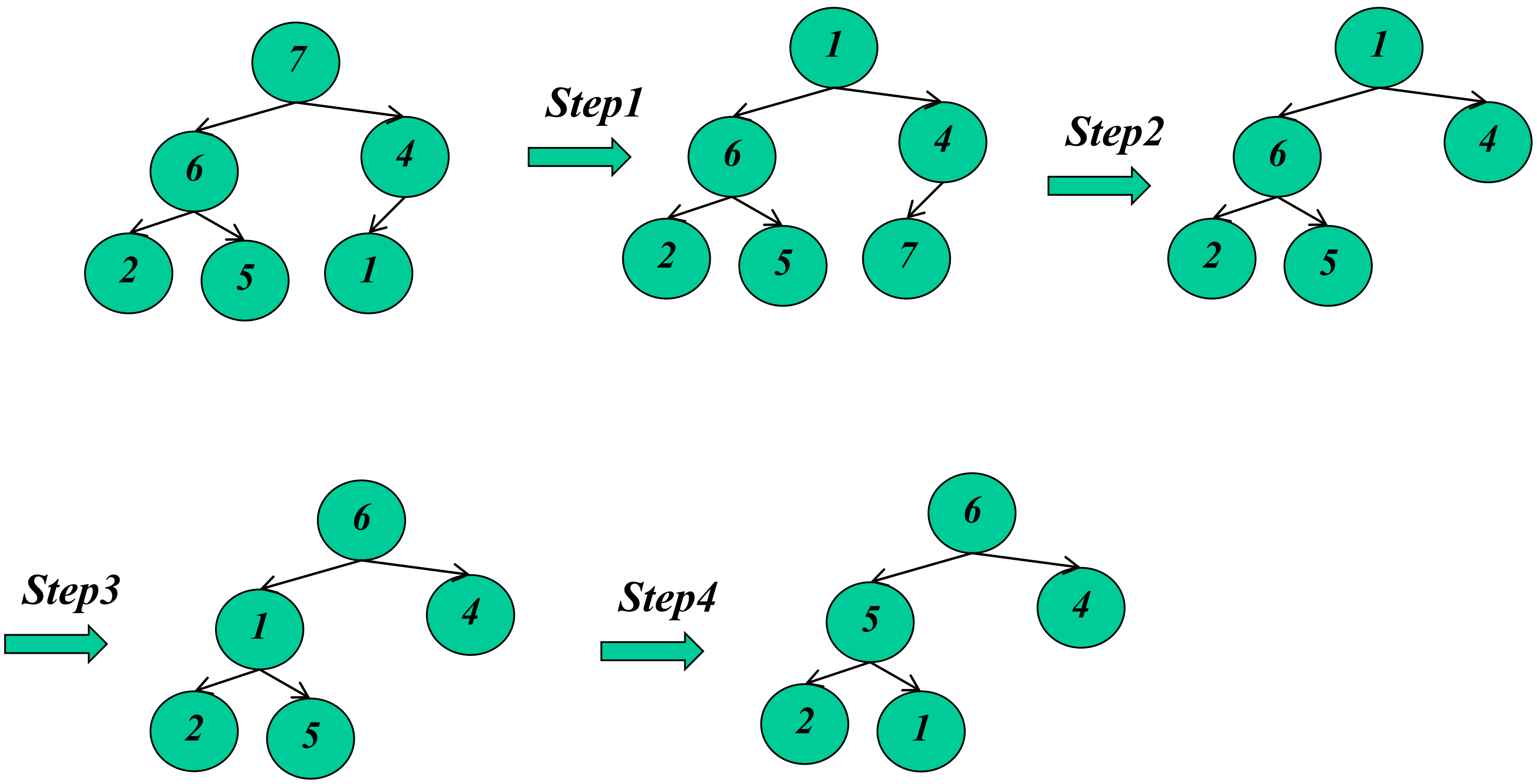
Two Stage algorithm to sort a list of *n* keys

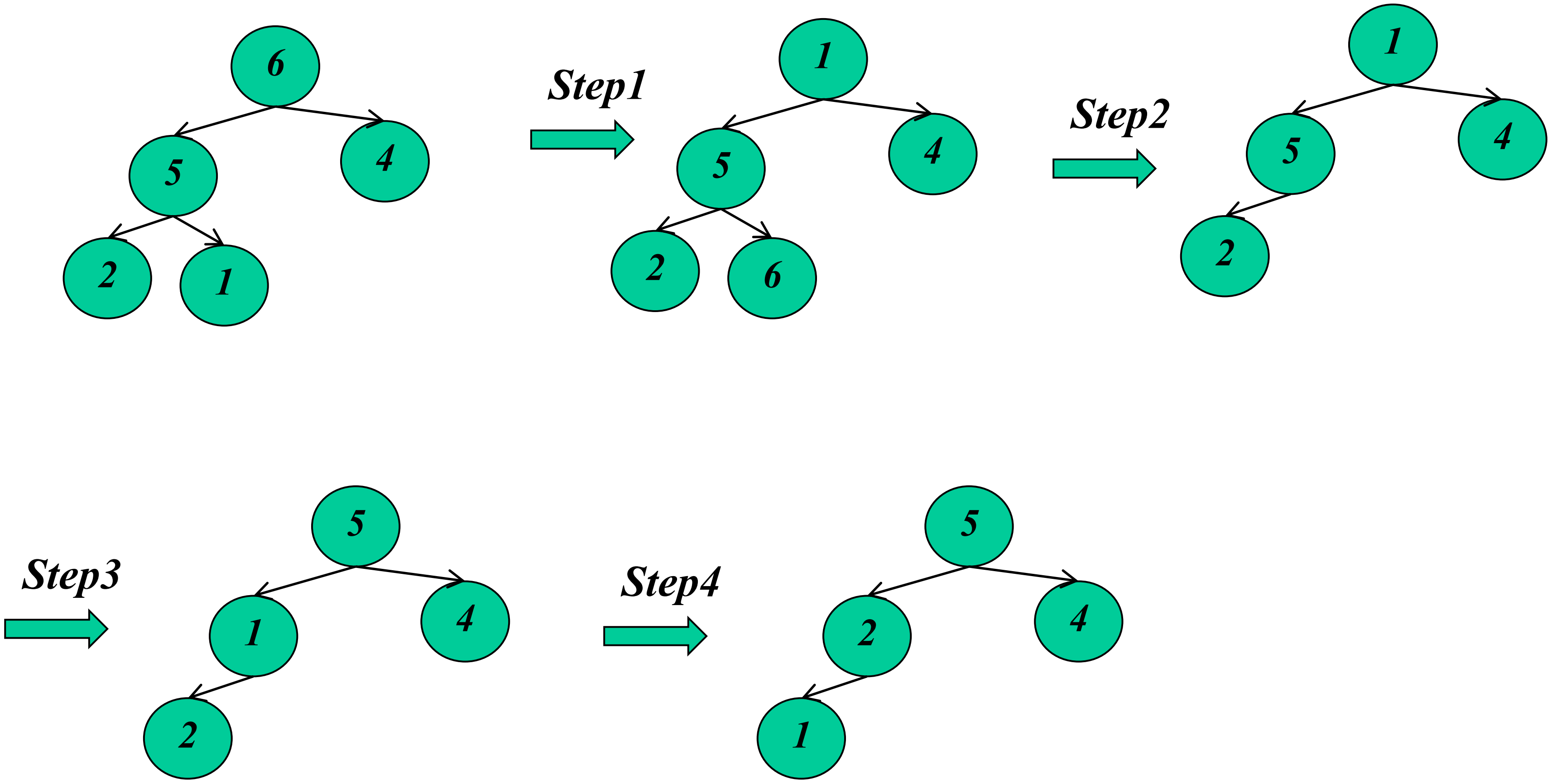
First, heap construction $O(n)$

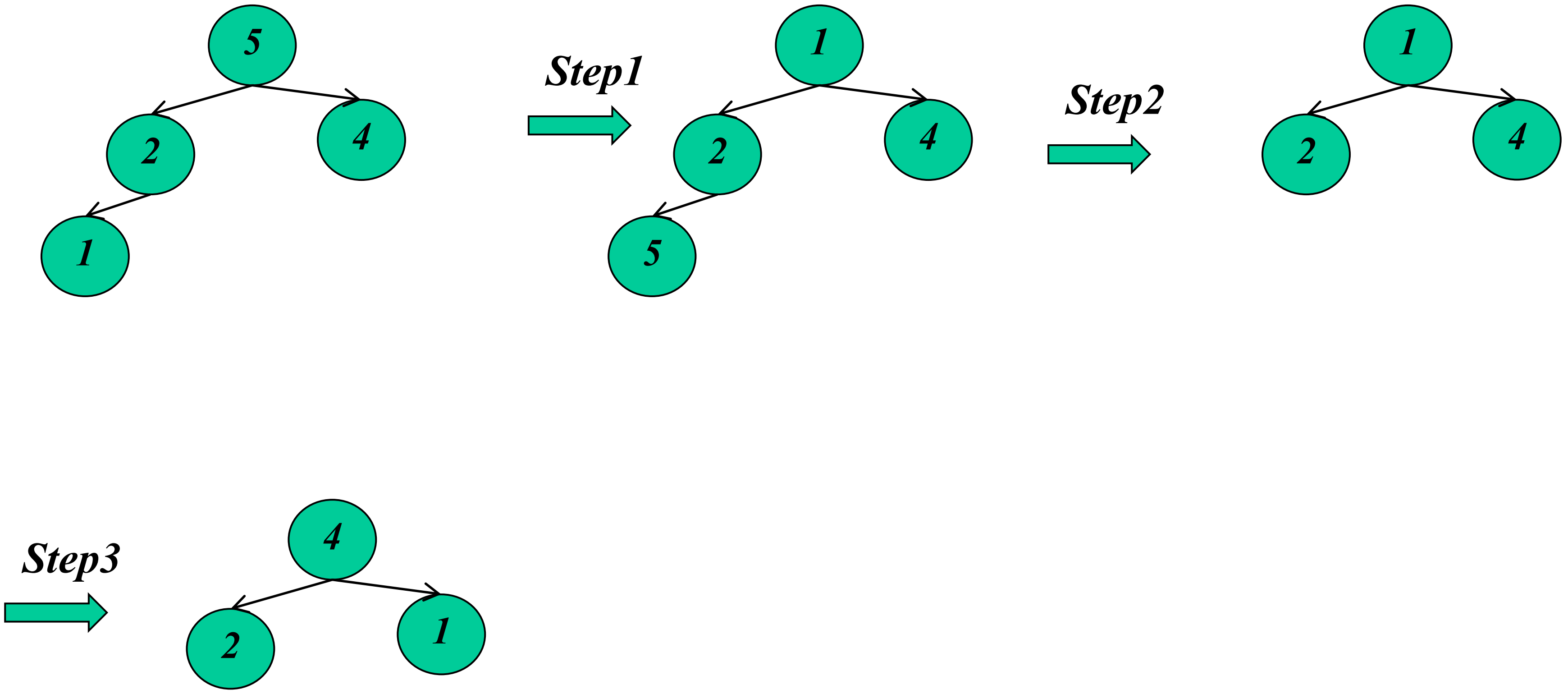
Second, sequential root deletion (the largest is deleted first, and the second largest one is deleted second, etc …)

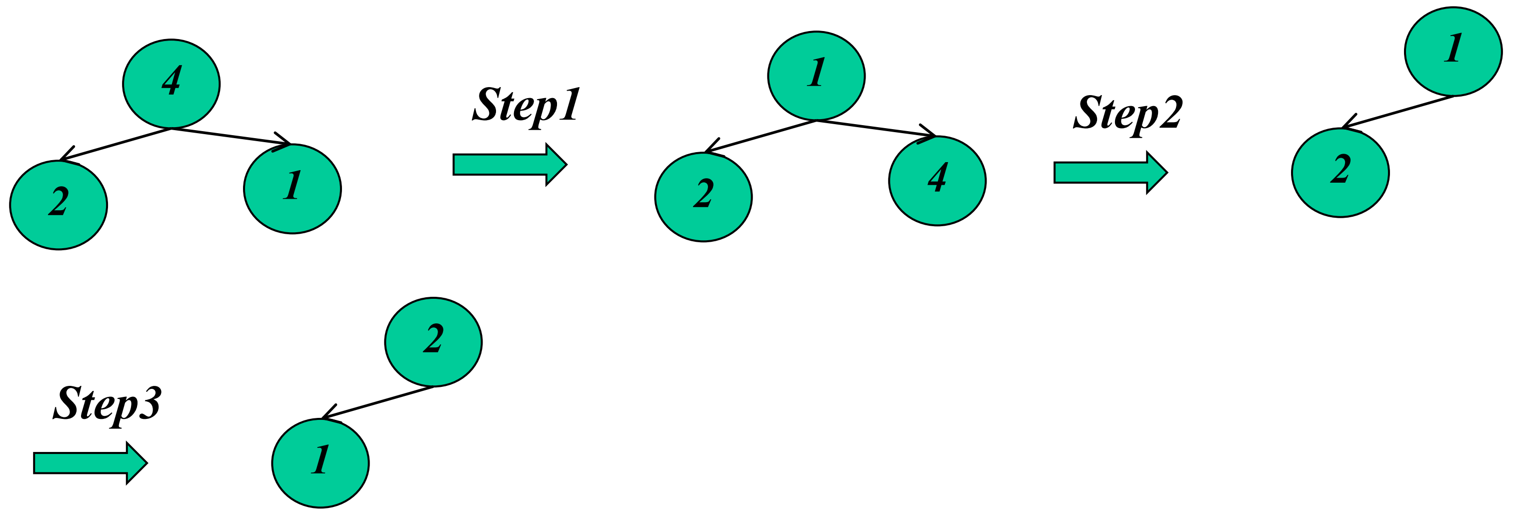

Step1
Step2
Step3
Step5


Step1
Step2
Step3


7
6
4
2
5
1
Step1
1
6
4
2
5
7
Step2
1
6
4
2
5
Step3
6
1
4
2
5
Step4
6
5
4
2
1


6
5
4
2
1
Step1
1
5
4
2
6
Step2
1
5
4
2
Step3
5
1
4
2
Step4
5
2
4
1


5
2
4
1
Step1
1
2
4
5
Step2
1
2
4
Step3
4
2
1



Step1
Step2
Step3

Notes on Heapsort

Time efficiency:

- Worst case

$$C(n) = 2\sum_{i=1}^{n-1} \log_2 i \in \mathrm{O}(n \log n)$$

- Average case efficiency is also $\mathrm{O}(n \log n)$

Advantage: in place – no additional space needed

Disadvantage: not stable

Reading Assignment

Chapter 6.5 and 6.6