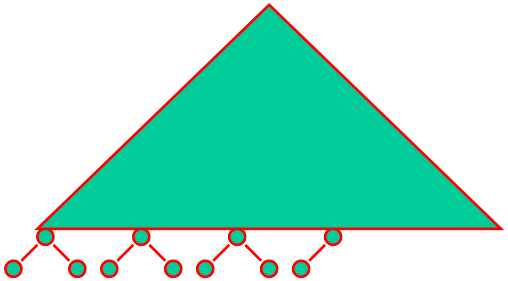
Representation Change – Heap and Heapsort

Definition:

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

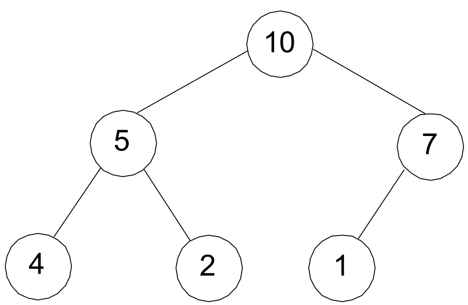
Heap Implementation

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

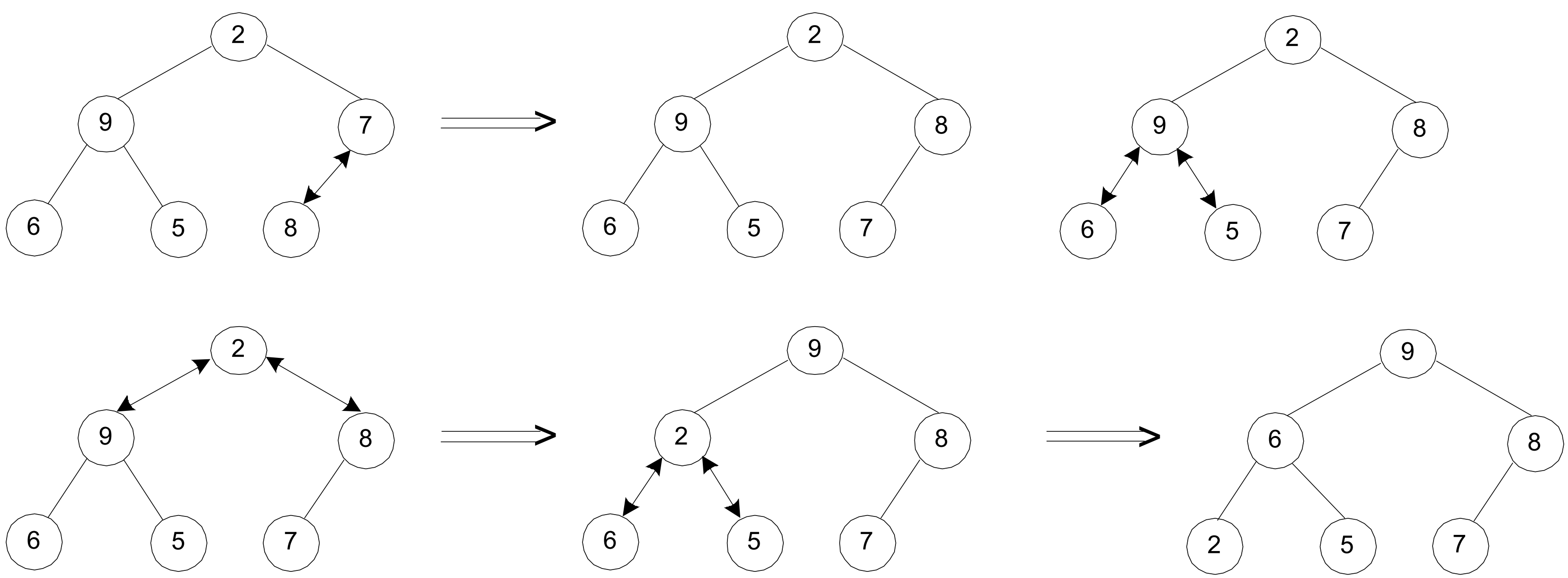
Heap Construction -- Construct a heap for a given list of keys

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

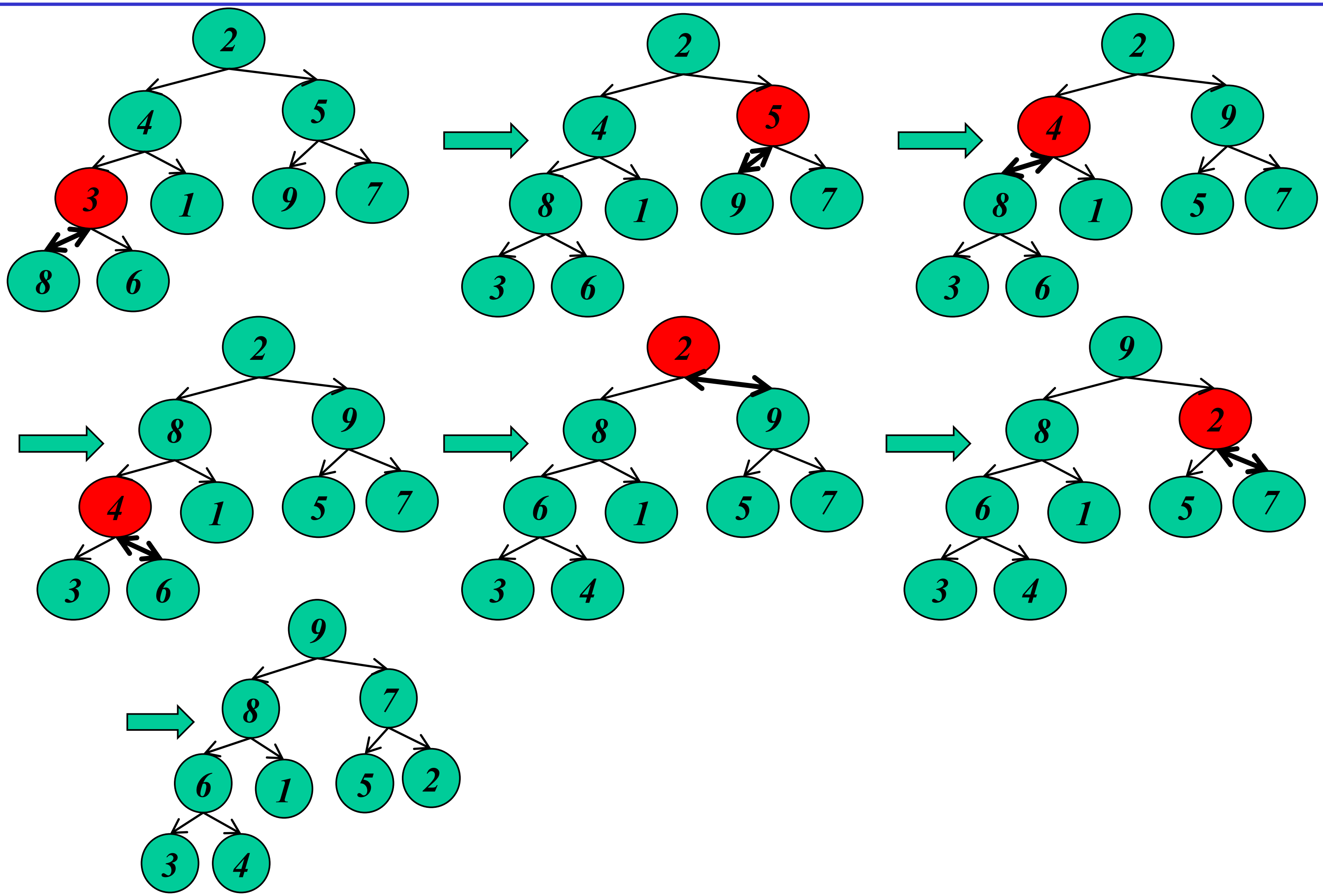
- Starting from the last parental node to the first parental node, check whether $H[i] \geq \max\{H[2i], H[2i+1]\}$
- If not, swap parental and child keys to satisfy this requirement

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**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    $heap \leftarrow$ **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

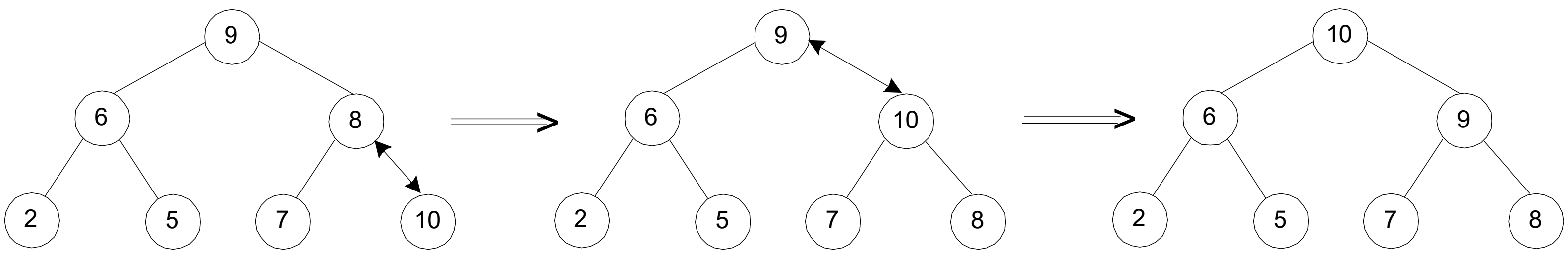
It is based on the operation of inserting a new item to an existing heap, and maintain a heap

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 $\mathrm{O}(\log n)$ 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** $O(n)$ → The top-down heap construction is less efficient than the bottom-up heap construction
- **Top-down** $O(n \log n)$

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

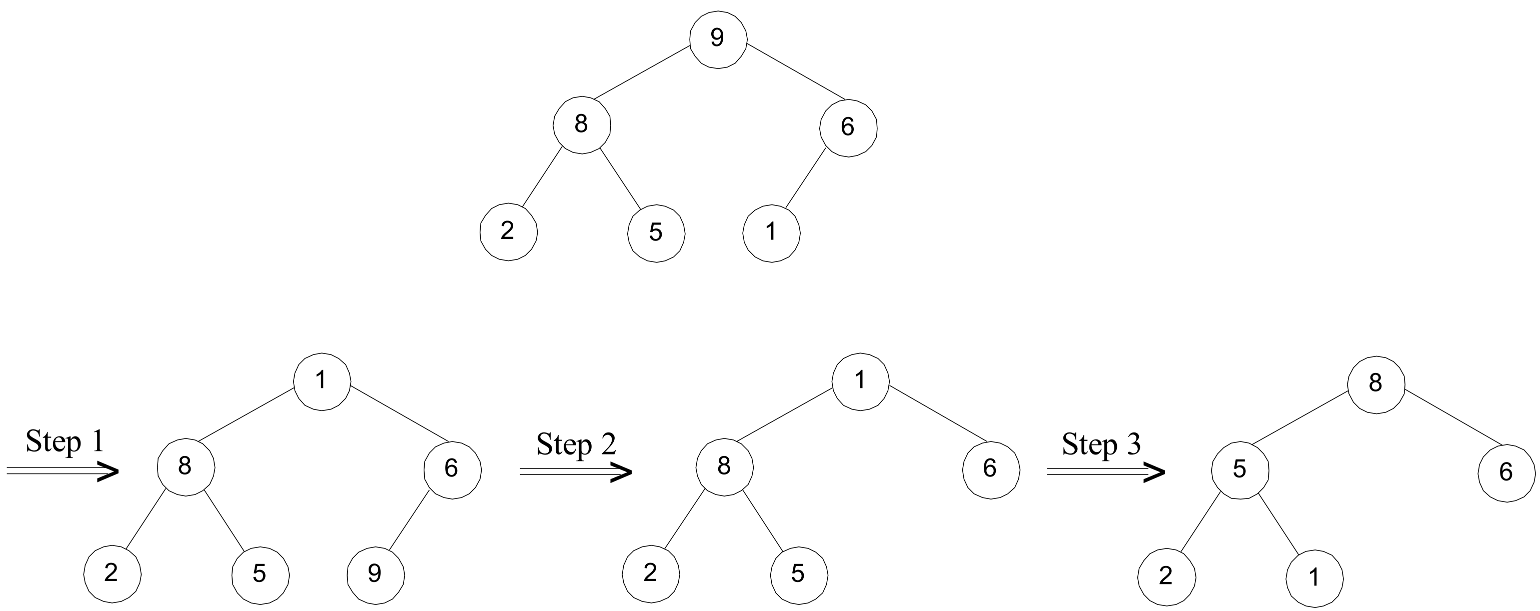
(1) Exchange the root's key with the last key *K* of the heap

(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?

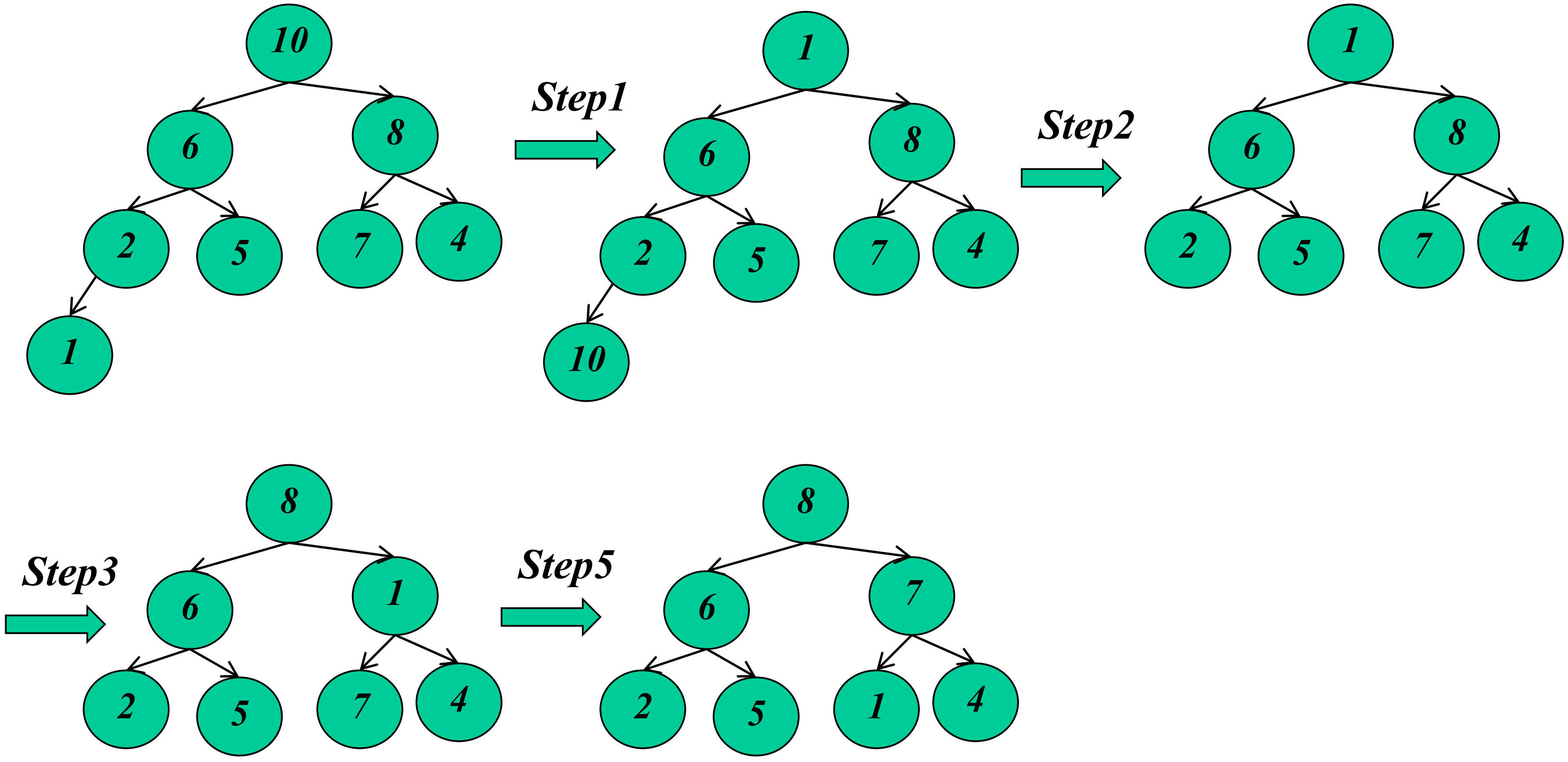
- Search for the key $O(n)$
- It is similar to the three-step root-deletion operation $O(\log n)$
  - Exchange with the last element *K*
  - "Heapify" the new binary tree. But it may be shift up or down, depending on the value of *K*

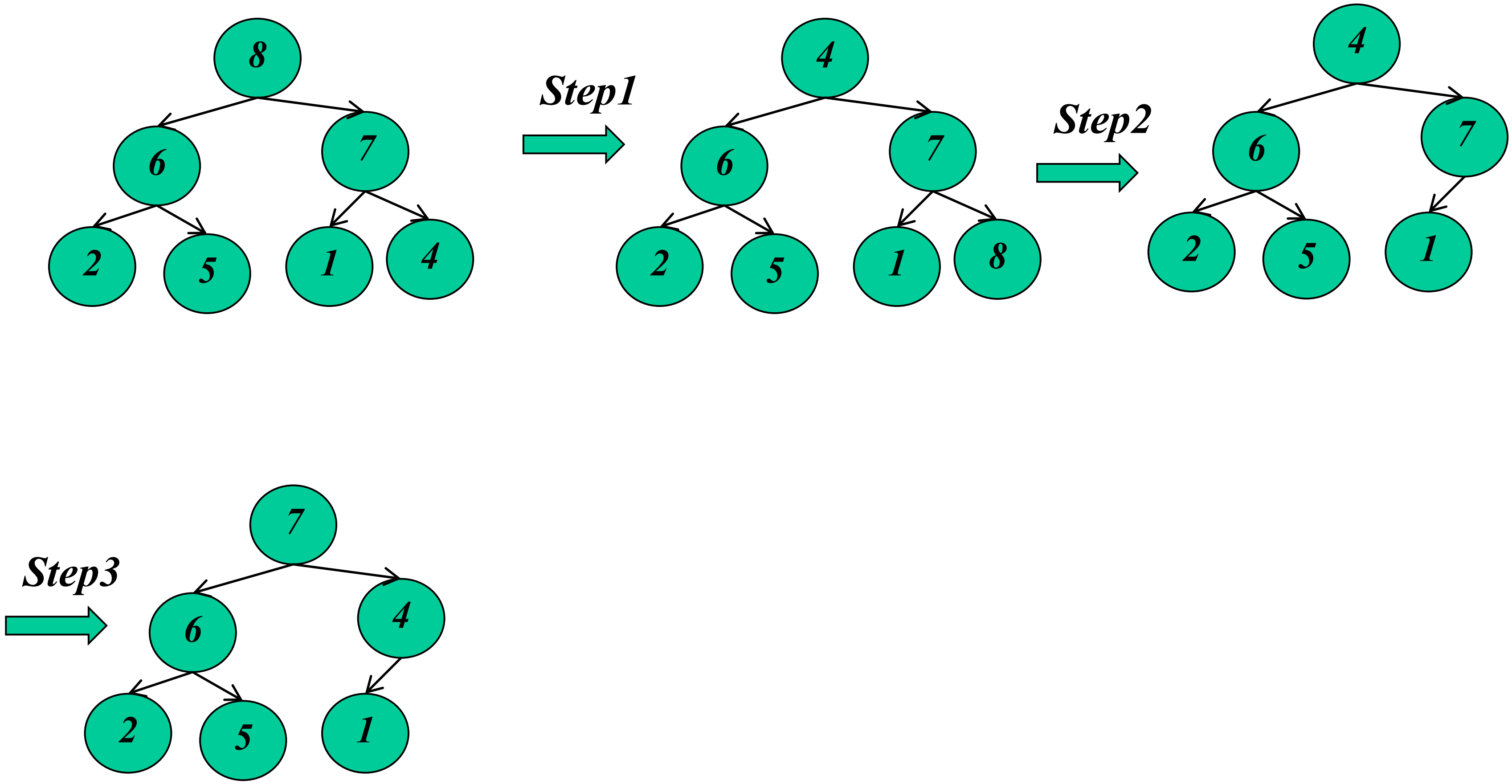
Heapsort

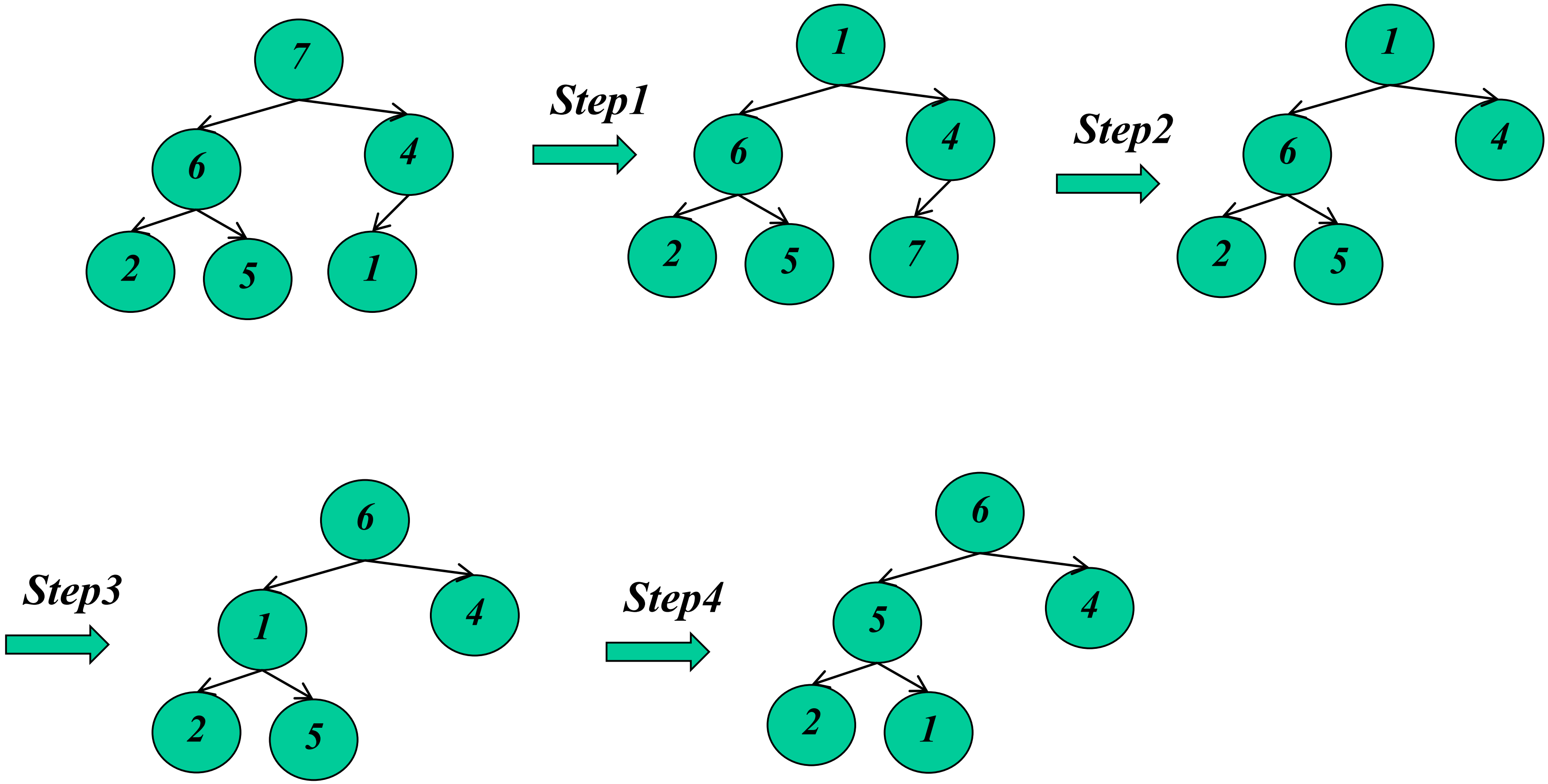
Two Stage algorithm to sort a list of n keys

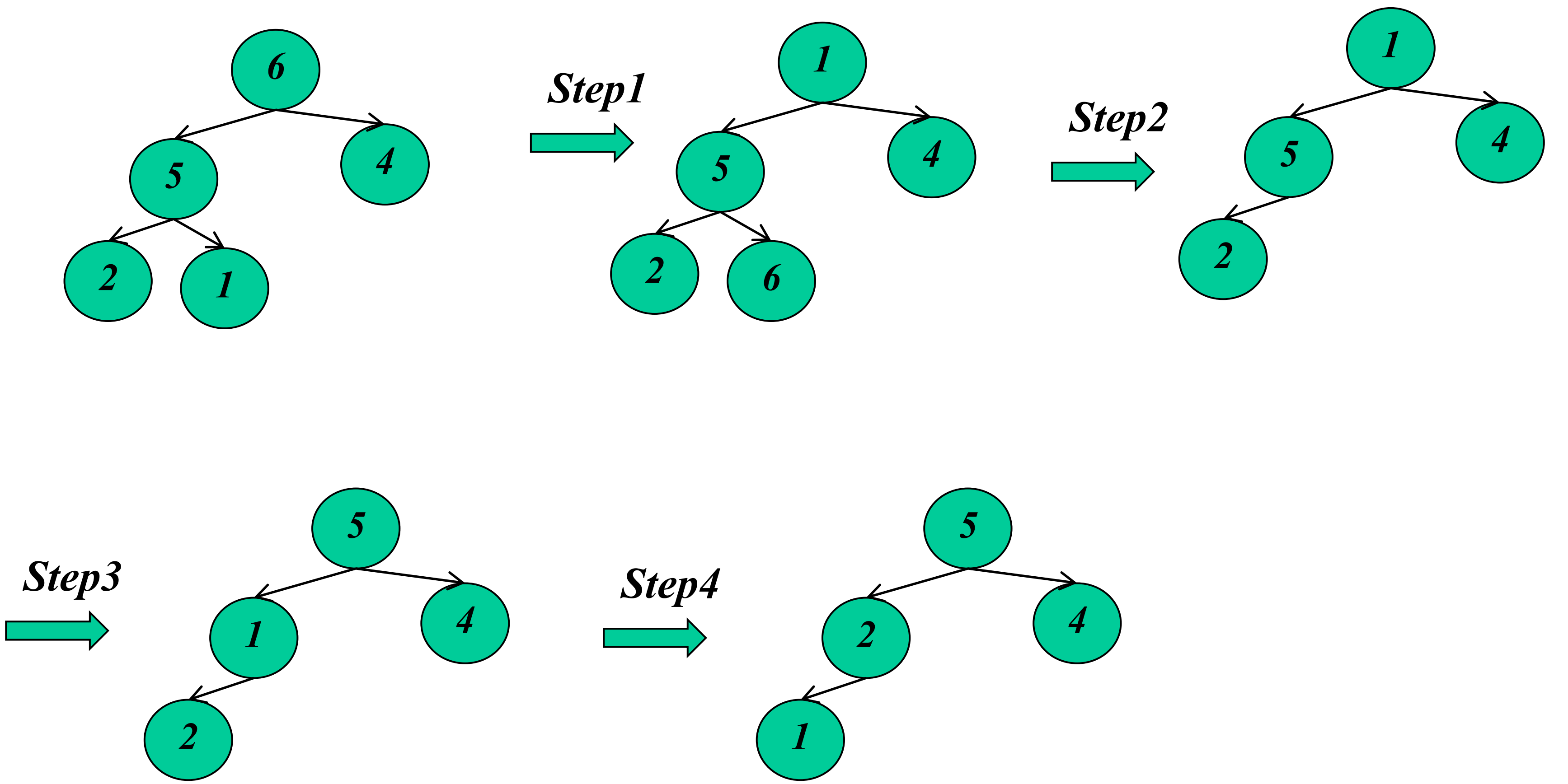
First, heap construction $O(n)$

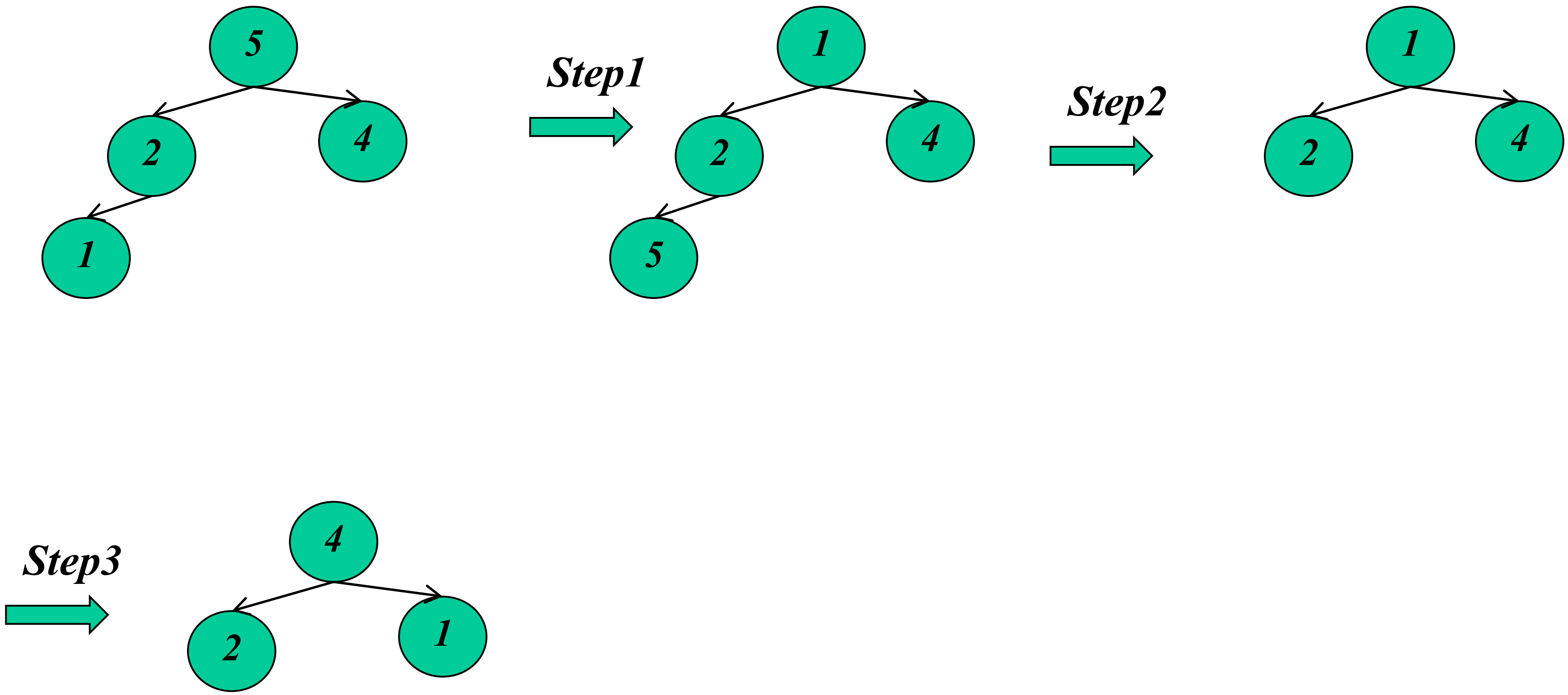
Second, sequential root deletion (the largest is deleted first, and the second largest one is deleted second, etc …)

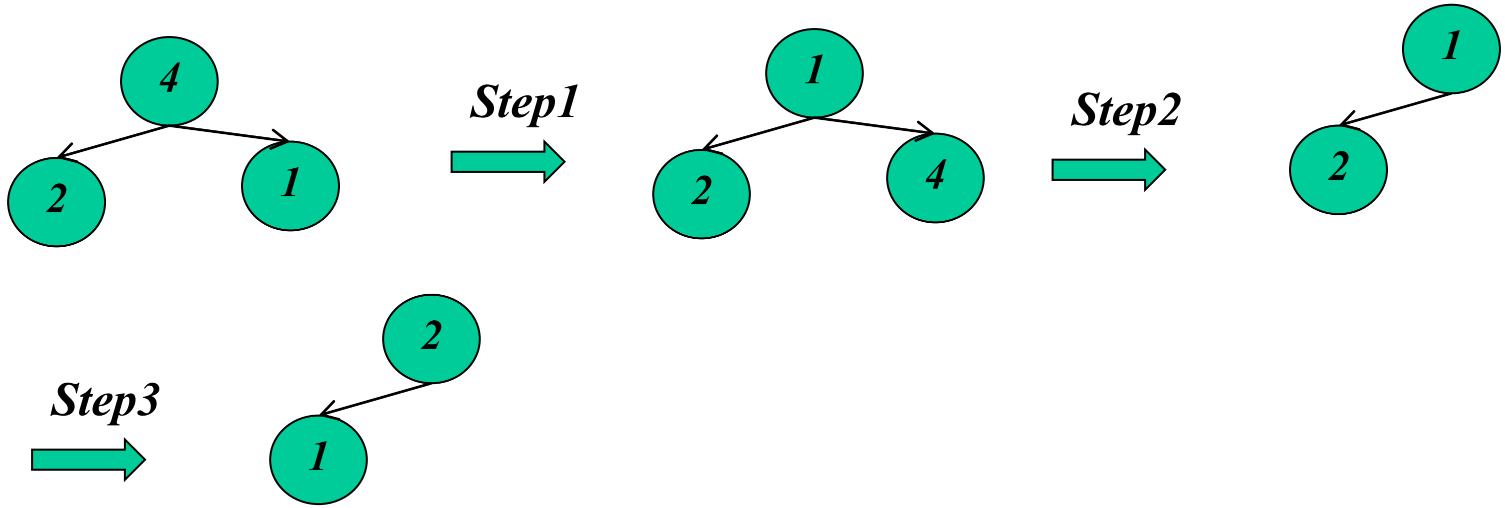

Step1
Step2
Step3
Step5


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


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

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

Review for Midterm Exam 2 – Chapter 3

Brute Force – the straightforward algorithm-design strategies

Simple but may not be efficient

Typical brute-force algorithms we learned (you should know they are brute force methods)
- Selection Sort, Bubble Sort
- Sequential Search
- String matching

Exhaustive Search
- List all the solutions in the problem domain
- TSP, knapsack problem, assignment problem
  - Efficiency class

Review for Midterm #2 - Chapter 4

Decrease by one:

- Insertion sort
  - How to perform the insertion sort
  - Time efficiency of best case $\Theta(n)$, worst case $\Theta(n^2)$, and average case $\Theta(n^2)$
- Graph search algorithms
  - DFS
    - Perform a DFS and record the orders of push-in and pop-out
    - Construct a DFS forest and identify different types of edges
  - BFS
    - Perform a BFS and record the order of visiting vertices (queue)
    - Construct a BFS forest and identify different types of edges
  - Topological sorting
    - DFS
    - Source removal

Review for Midterm #2 - Chapter 4

Variable-size decrease

- Binary search tree
  - How to construct a BST?
- Selection by partition
  - What is the basic idea?
  - Time efficiency class: best case and average case $\Theta(n)$; worst case $\Theta(n^2)$

Review for Midterm #2 - Chapter 5

- **Mergesort**
  - Time efficiency of best case $\Theta(nlogn)$, worst case $\Theta(nlogn)$, and average case $\Theta(nlogn)$
  - Perform a mergesort

- **Quicksort**
  - Partition scheme (other applications like selection)
  - Perform a quicksort
  - Time efficiency of best case $\Theta(nlogn)$, worst case $\Theta(n^2)$, and average case $\Theta(nlogn)$
  - How to improve a quicksort

- **Tree traversal**
  - Perform a preorder, inorder, and postorder traversal
  - Check the height of a tree needs n additions and $2n+1$ checking

Review for Midterm #2 - Chapter 5

- **Examples using divide and conquer**
  - Large integer multiplication
- **Design a divide and conquer algorithm for a given problem**

Review for Midterm #2 - Chapter 6

Instance simplification

- Presorting: examples of using presorting
  - Searching, computing the mode, finding repeated elements, etc
  - Selection problem
  - Design a presorting-based algorithm

Representation change

- balanced search trees
  - How to construct an AVL tree? Rotations!
- heaps and heapsort
  - Construct a heap: top-down, bottom-up
  - Insertion and deletion
  - Perform a heapsort