INT202
Complexity of Algorithms
Sorting Algorithms

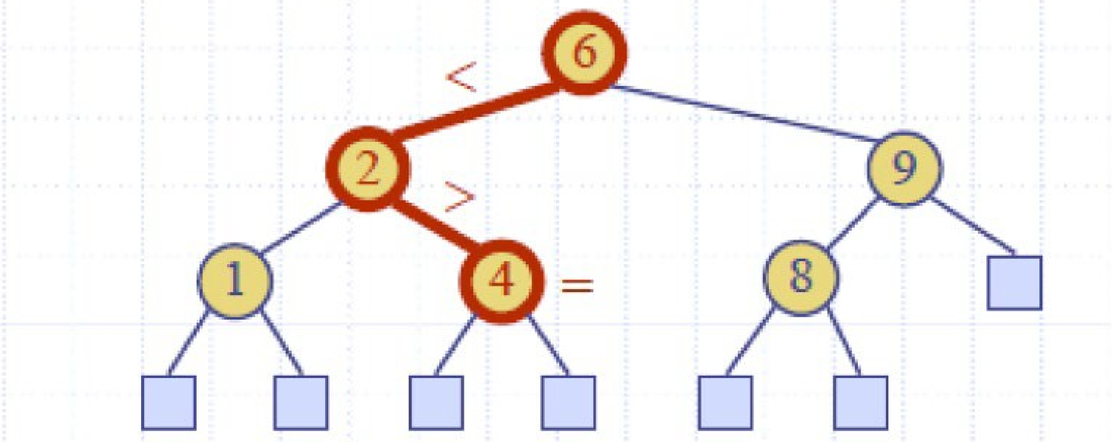
XJTLU/SAT/INT

SEM2 AY2020-2021

Review

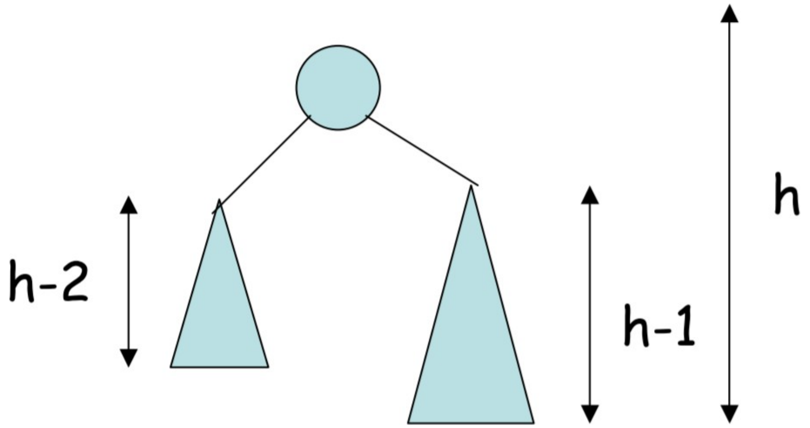
- **Binary Search Tree (BST)**
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO—SUCH KEY

Algorithm *findElement*(k, v)
    if *T.isExternal* (v)
        return *NO_SUCH_KEY*
    if $k < key(v)$
        return *findElement*(k, *T.leftChild*(v))
    else if $k = key(v)$
        return *element*(v)
    else { $k > key(v)$ }
        return *findElement*(k, *T.rightChild*(v))

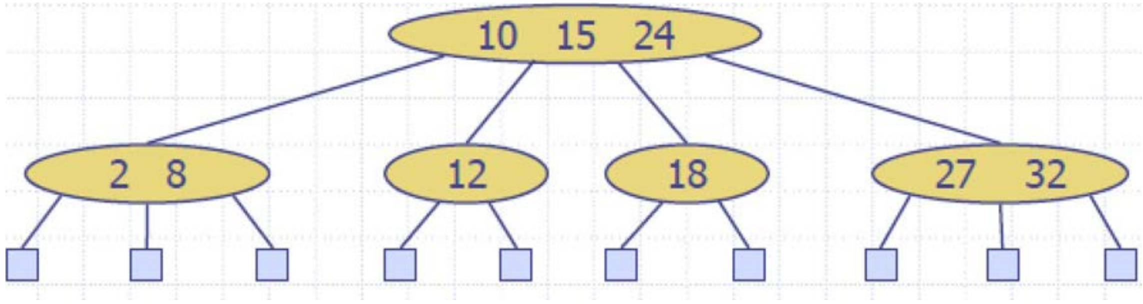


Review

- **Binary Search Tree (BST)**
- **AVL Tree**
- *Height-Balance Property*: for every *internal* node, *v*, of *T*, the heights of the children of *v* can differ by at most 1.



- **(2,4) tree**
- A multi-way search
- Node-Size Property: every internal node has at most four children
- Depth Property: all the external nodes have the same depth



Review

findElement, insertItem, removeElement

- **BST**

  All operations in a BST are performed in $O(h)$, where h is the height of the tree.

- **AVL Tree , (2,4) Tree**

  All these operations are performed in O(log n)

Sorting

Sorting problem: Given a collection, *C*, of *n* elements (and a total ordering) arrange the elements of *C* into *non-decreasing* order, e.g.

45	3	67	1	5	16	105	8

1	3	5	8	16	45	67	105

Sorting

Sorting is a fundamental algorithmic problem in computer science.

We will investigate various methods that we can use to sort items.

Many algorithms perform sorting (as a subroutine) during their execution. Hence, efficient sorting methods are crucial to achieving good algorithmic performance.

We may not always require a fully sorted list, so some methods might be more appropriate depending upon the exact task at hand.

Sorting algorithms might be directly adaptable to perform additional tasks and directly provide solutions in this fashion.

Priority Queues

A **Priority Queue** is a container of elements, each having an associated *key*.

Keys determine the *priority* used in picking elements to be removed.

A *priority Queue* (PQ) has these fundamental methods:

- *insertItem(k,e)*: insert element *e* having key *k* into PQ.
- *removeMin()*: remove minimum element.
- *minElement()*: return minimum element.
- *minKey()*: return key of minimum element.

PQ Sorting - Algorithm

How can we use a priority queue to perform sorting on a set *C*?

Do this in two phases:

- *First phase*: *Put* elements of *C* into an initially empty priority queue, *P*, by a series of *n* insertItemoperations.
- *Second phase*: *Extract* the elements from *P* in *non-decreasing* order using a series of *n* removeMin operations.

Heap Data Structure

A *heap* is a realization of a Priority Queue that is *efficient* for both *insertions* and *deletions*.

A *heap* allows insertions and deletions to be performed in *logarithmic* time.

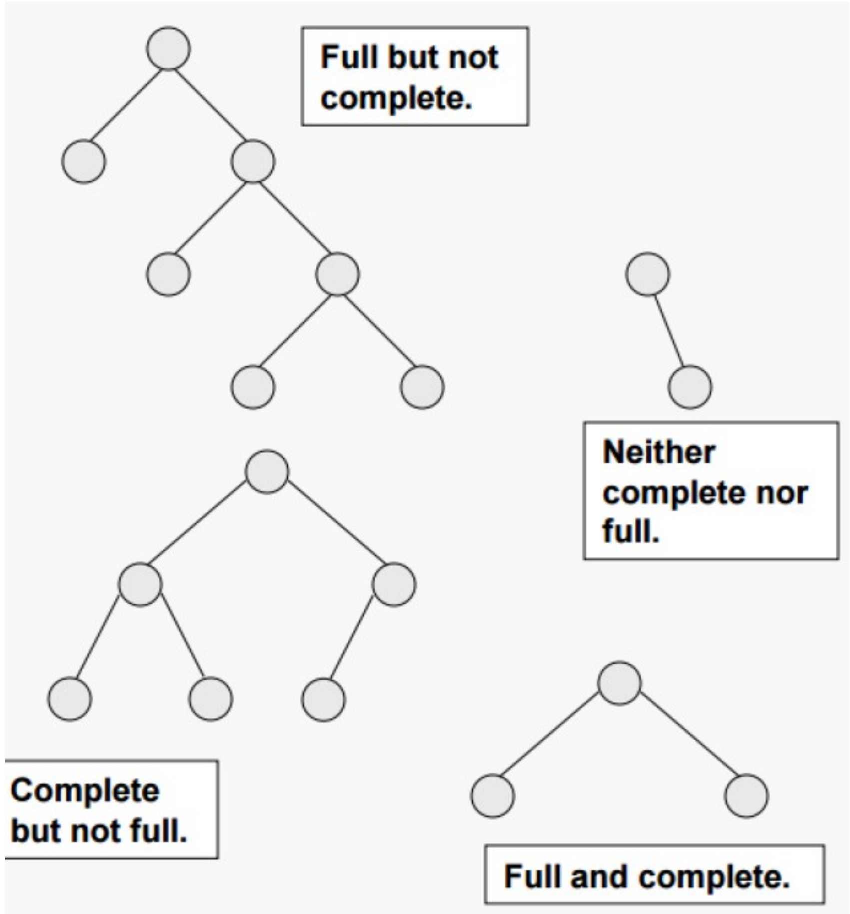
In a *heap* the *elements* and their *keys* are stored in an almost complete binary tree. Every level of the binary tree, except possibly the last one, will have the maximum number of children possible.

Complete Binary Tree

- Here are two important types of binary trees. Note that the definitions, while similar, are logically independent.

<u>Definition</u>: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.

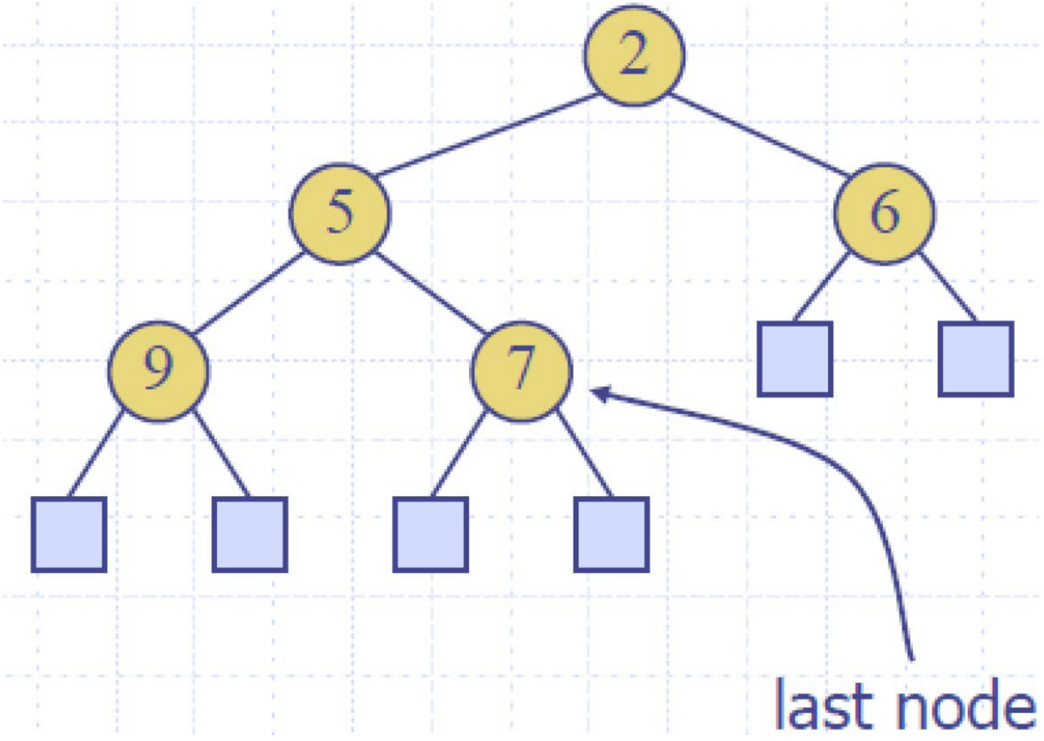
<u>Definition</u>: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side



Heap Data Structure

- ❖ A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root,
    key(v) ≥key(parent(v))

The Min Heap



Heap Data Structure

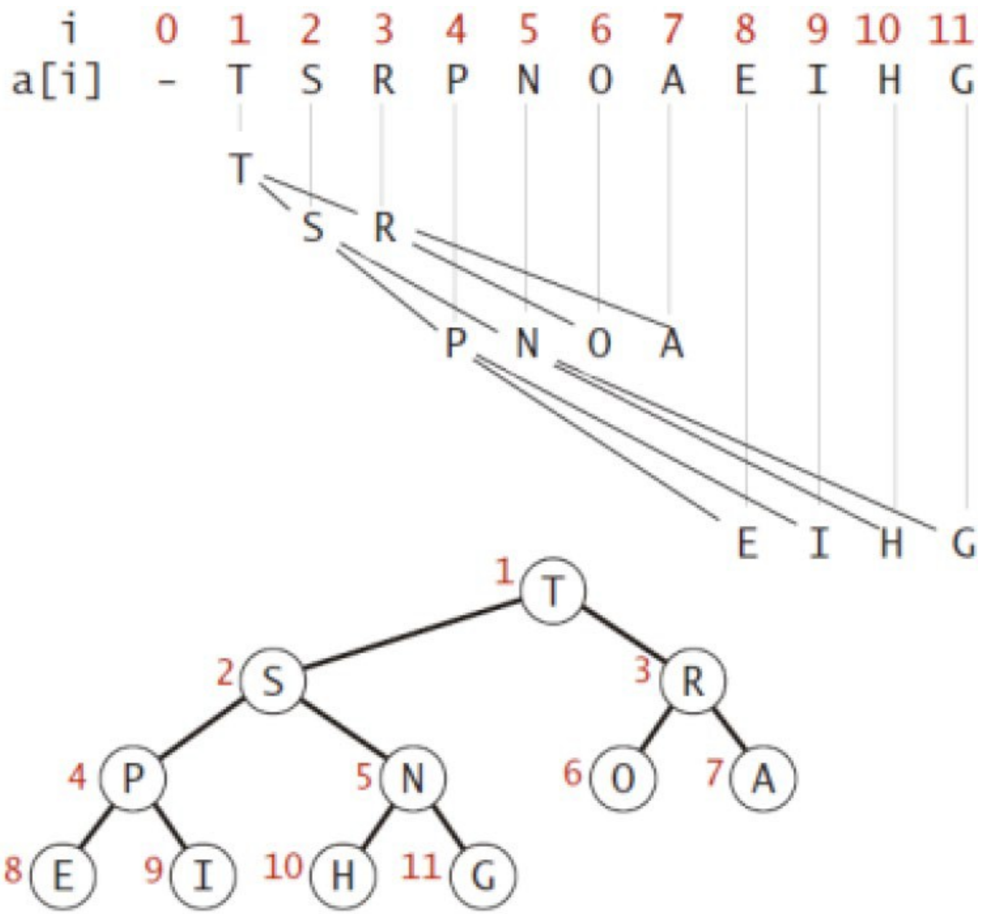
Binary heap. Array representation of a heap-ordered complete binary tree

Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no smaller than children's keys.

Array representation.

- Indices start at 1 .
- Take nodes in level order.
- No explicit links needed !



Heap representations

An efficient realization of a heap can be achieved using an array for storing the elements.

Heap Data Structure

Binary heap. Array representation of a heap-ordered complete binary tree

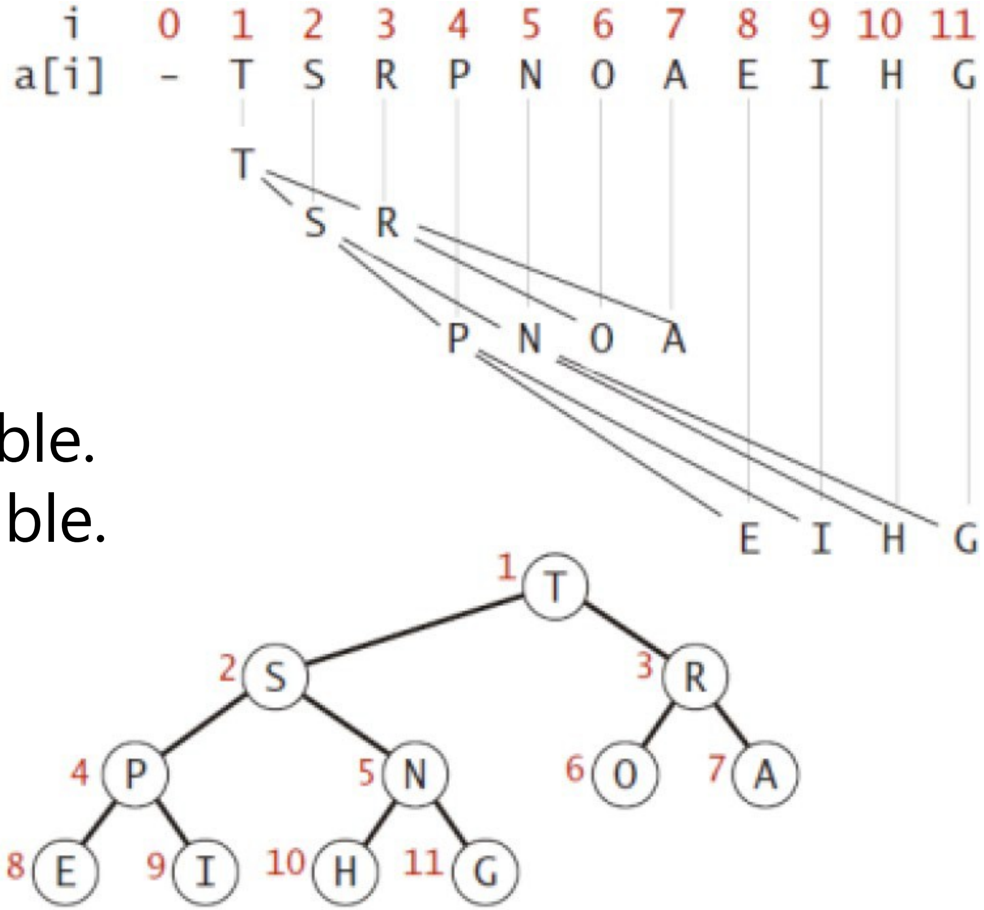
For any given node at position i:

•Its **Left Child** is at **[2*i]** if available.

•Its **Right Child** is at **[2*i+1]** if available.

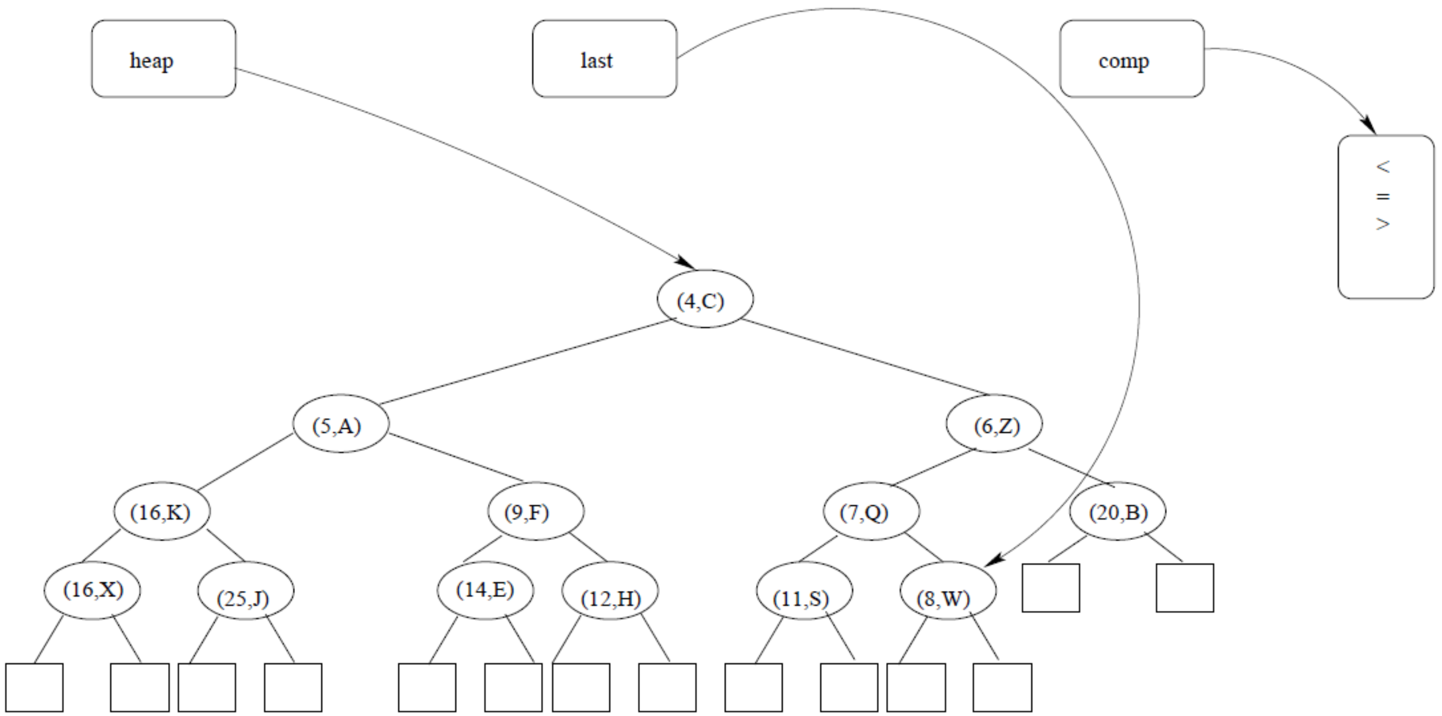
•Its **Parent Node** is at **[⌊i/2⌋]** if available.

An efficient realization of a heap can be achieved using an array for storing the elements.



Heap representations

PQ/Heap implementation



heap: A (nearly complete) binary tree T containing elements with keys satisfying the heap-order property, stored in an array.

last: A reference to the last used node of T in this array representation.

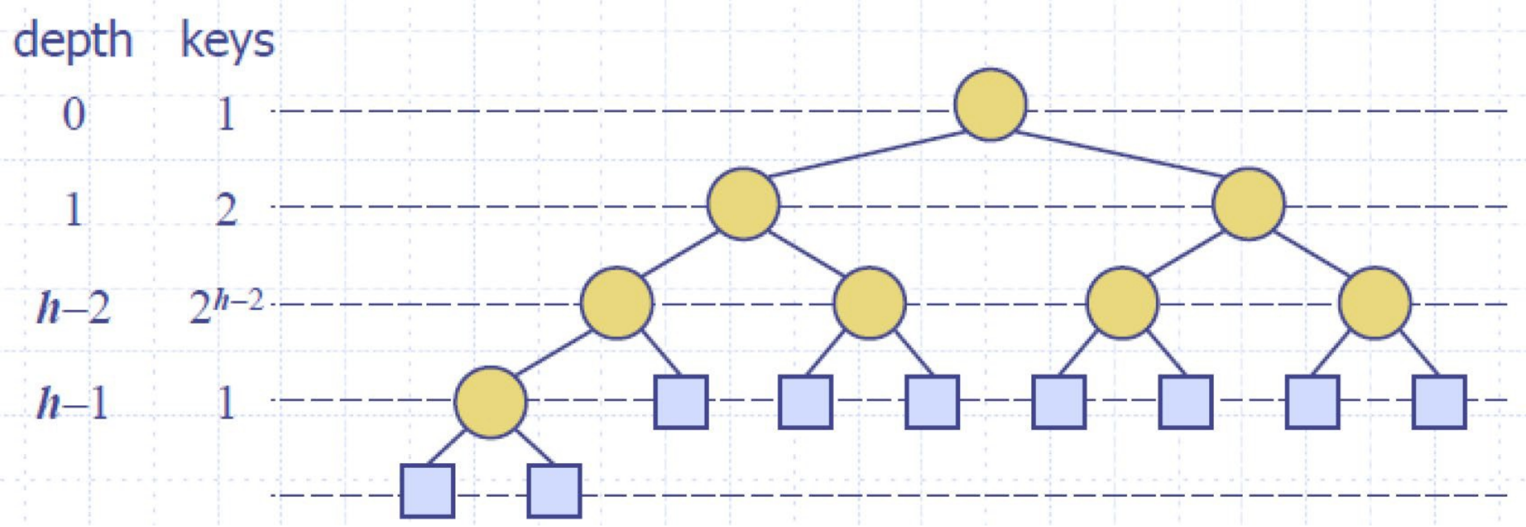
comp: A comparator function that defines the total order relation on keys and which is used to maintain the minimum (or maximum) element at the root of T .

PQ/Heap implementation

❖ Theorem: A heap storing m keys has height $\boldsymbol{O}(\log n)$

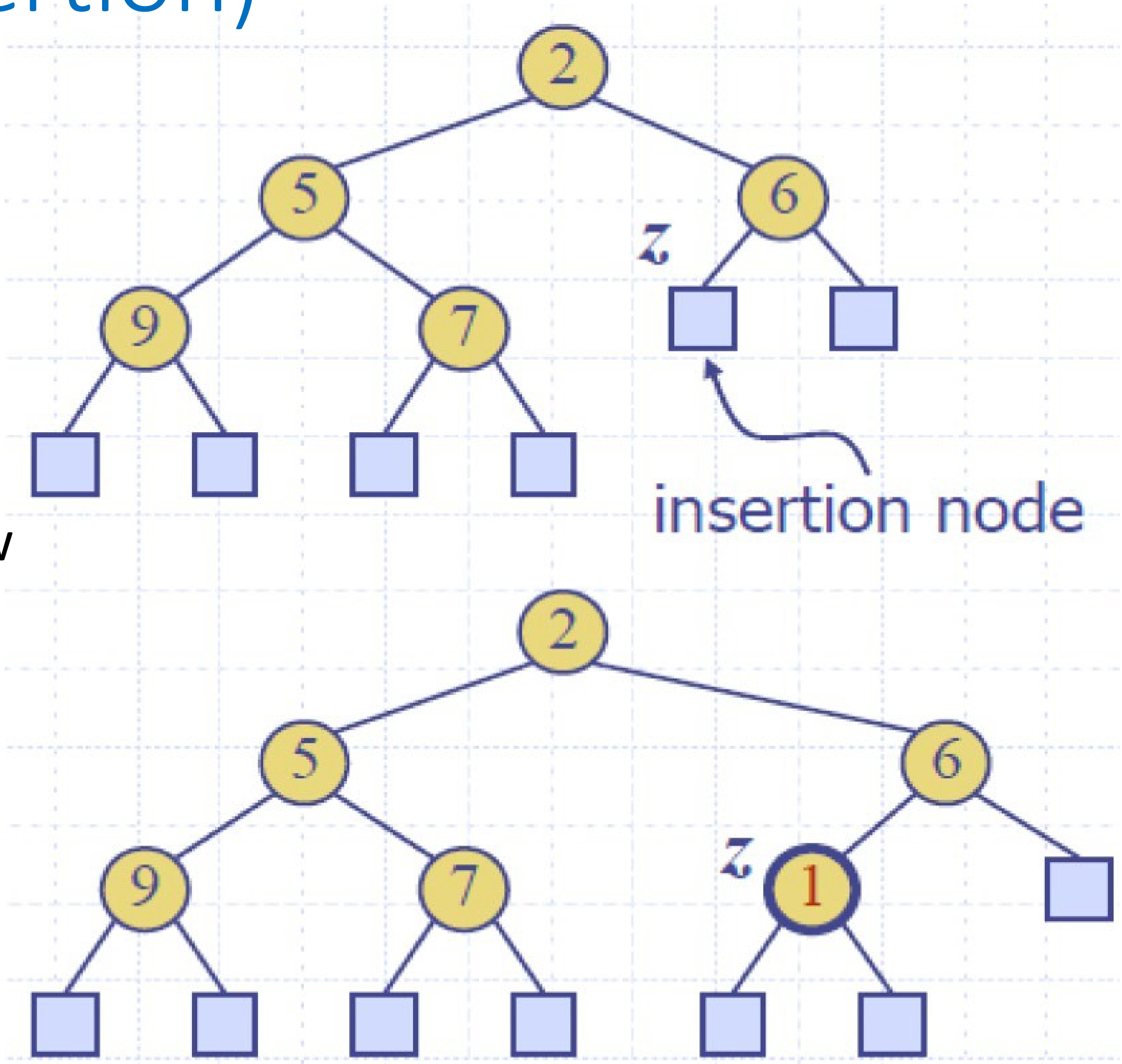
Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i \;=\; 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $\mathrm{n} \geq 1+2+4+\cdots+2^{h-2}+1$
- Thus, $\mathrm{n} \geq 2^{h-1}$, i.e., $h \;\leq \log n \;+1$



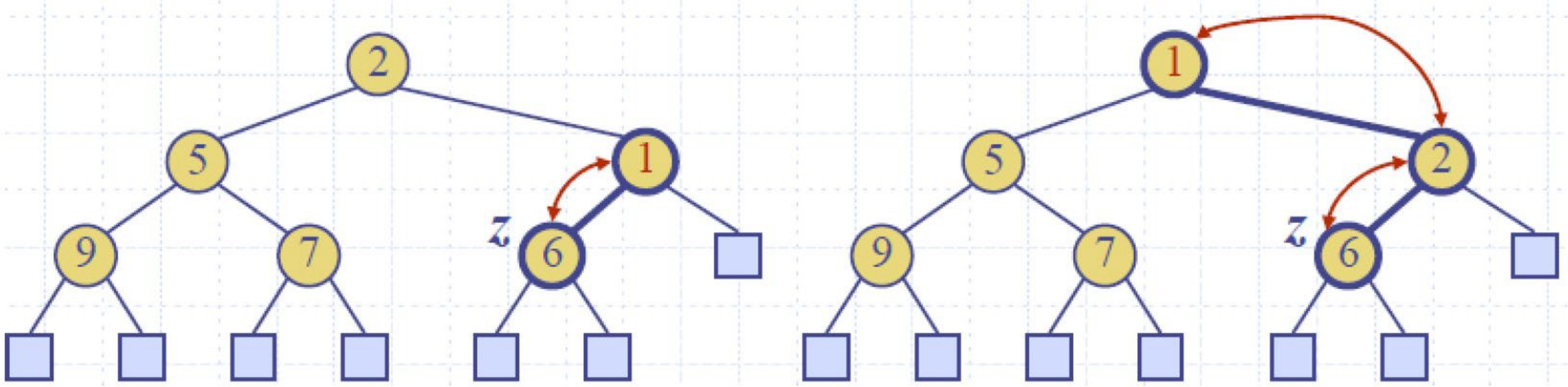
Up-heap bubbling (insertion)

- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



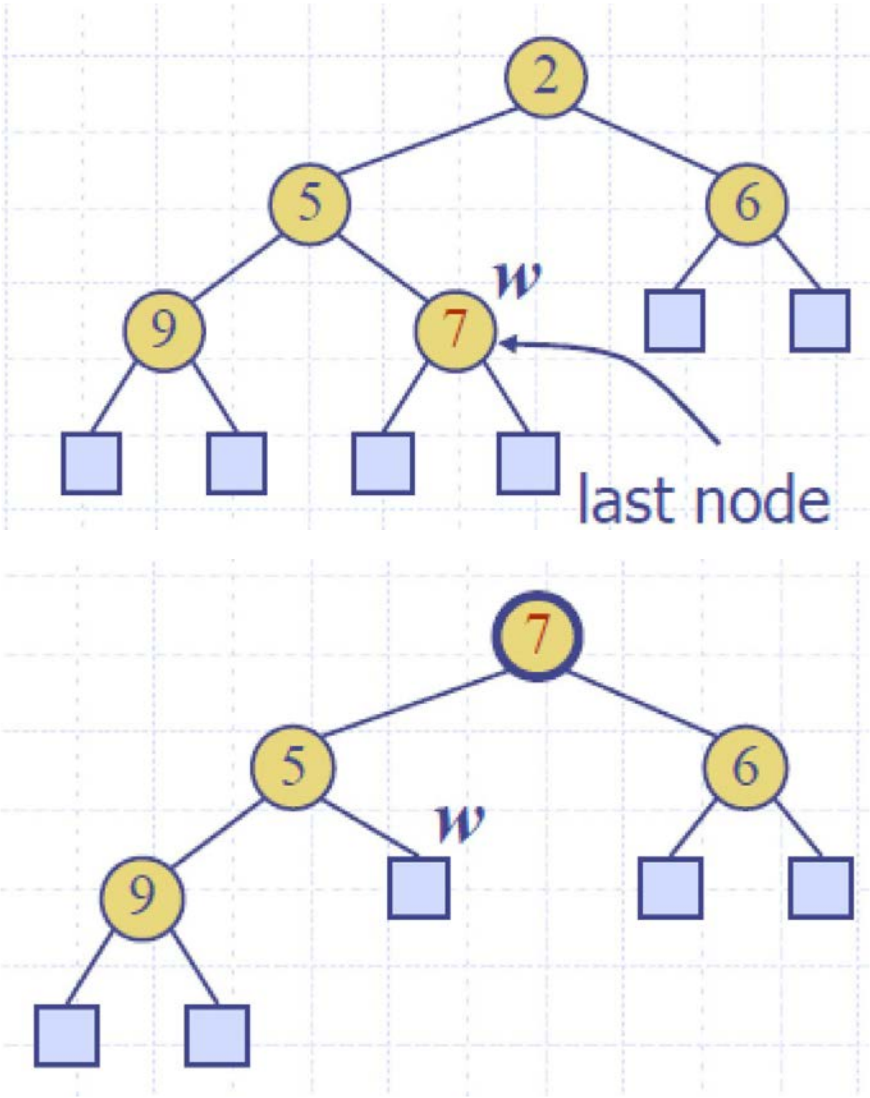
Up-heap bubbling (insertion) (cont.)

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



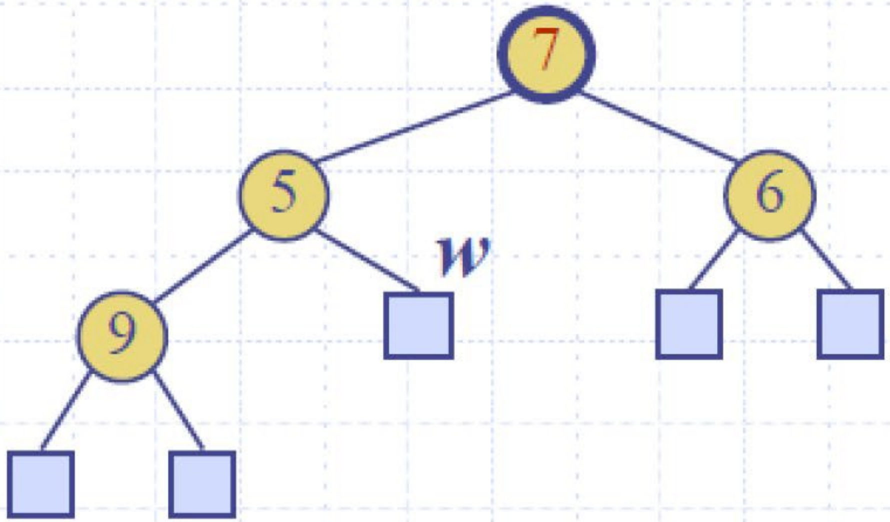
Down-heap bubbling (removal of top element)

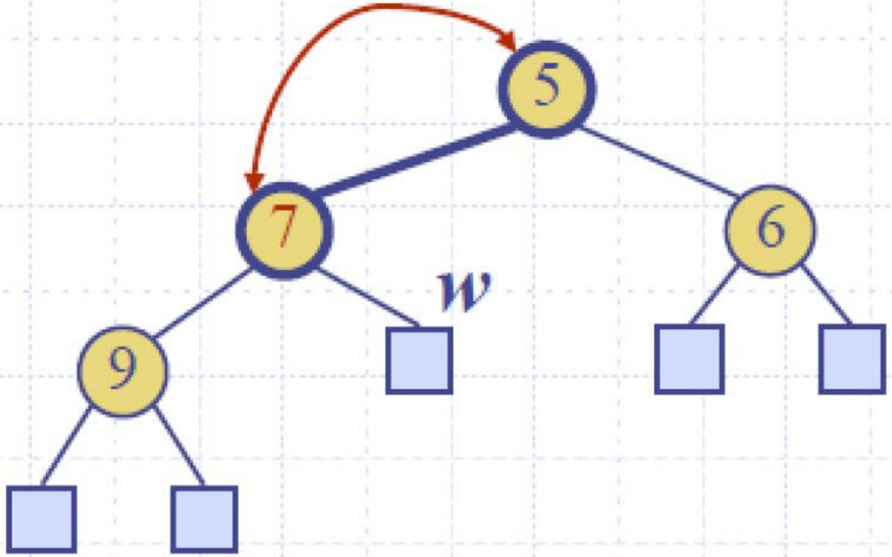
- Method *removeMin* of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
  - Restore the heap-order property (discussed next)



Down-heap bubbling (cont.)

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time





Heap-Sorting

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods *insertItem* and *removeMin* take $O(\log n)$ time
  - methods *size, isEmpty, minKey*, and *minElement* take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort

Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

▶ *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more *disjoint* subsets.

▶ *Recur*: *Recursively* solve the sub-problems associated with subsets.

▶ *Conquer*: Take the solutions to sub-problems and *merge* into a solution to the original problem.

MergeSort

Merge-sort on an input sequence S with n elements consists of three steps:

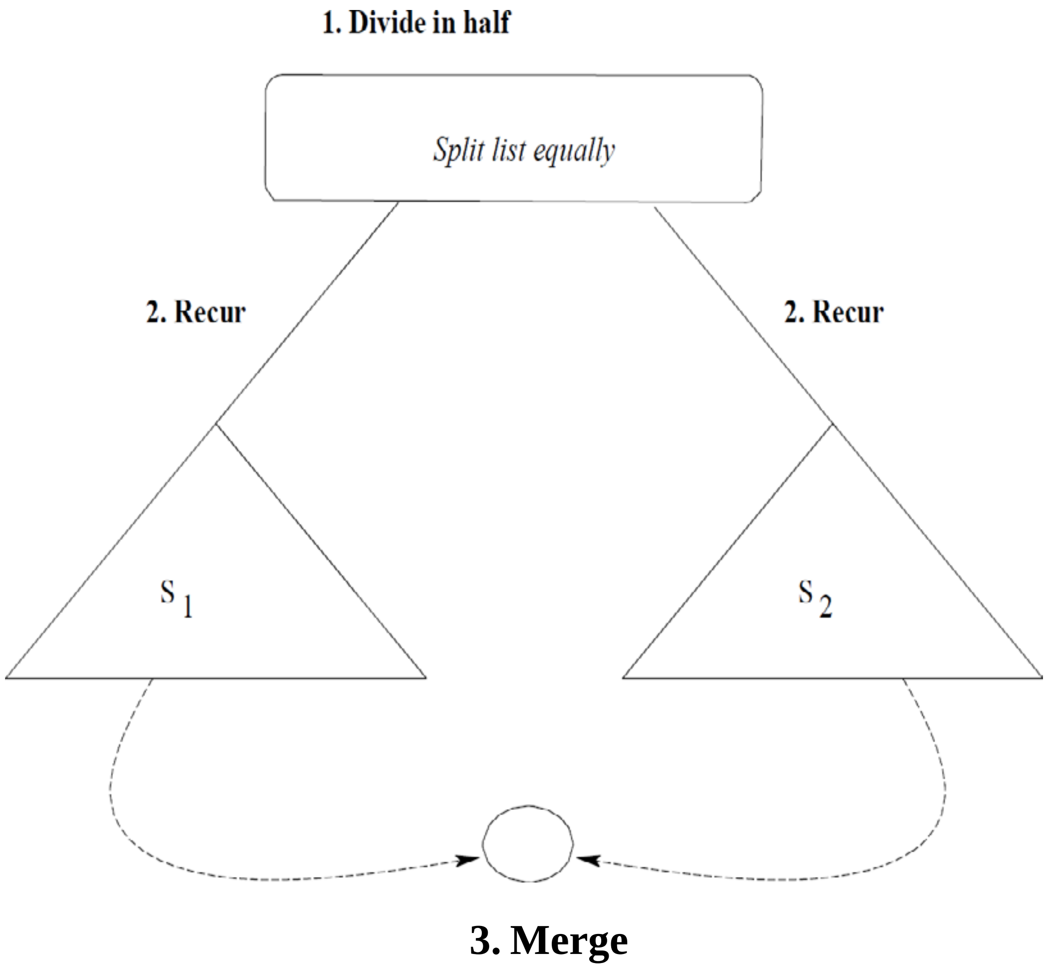
- Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    If S.size() > 1
        (S1 , S2) ← partition(S, n/2)
        mergeSort(S1, C)
        mergeSort(S2, C)
        S ← merge(S1, S2)
```

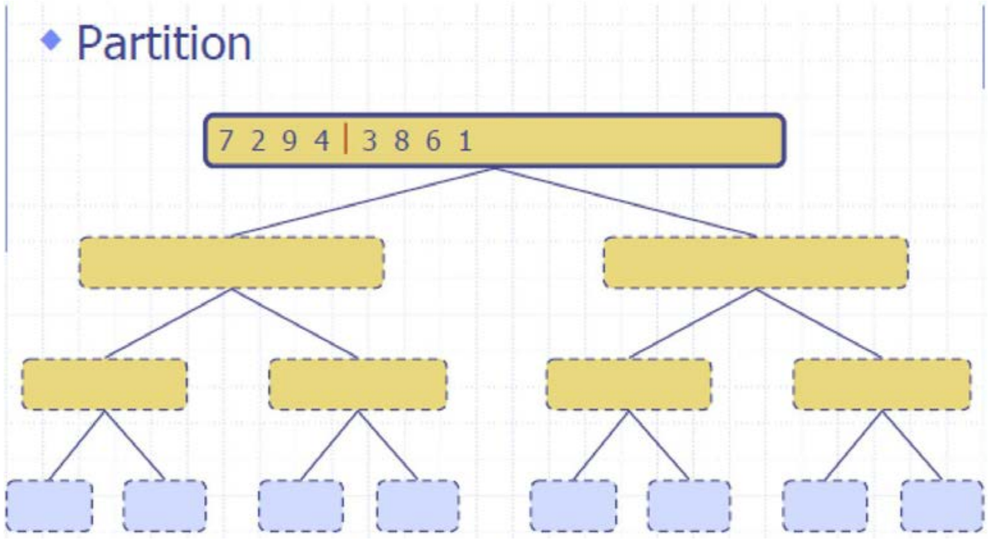
MergeSort - Illustration

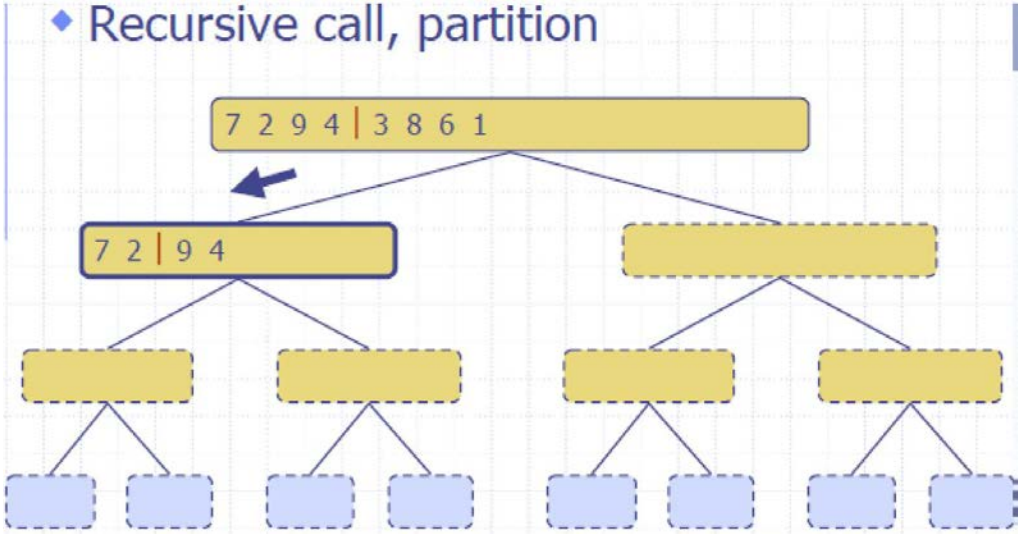
Merge-sort on an input sequence S with n elements consists of three steps:

- Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

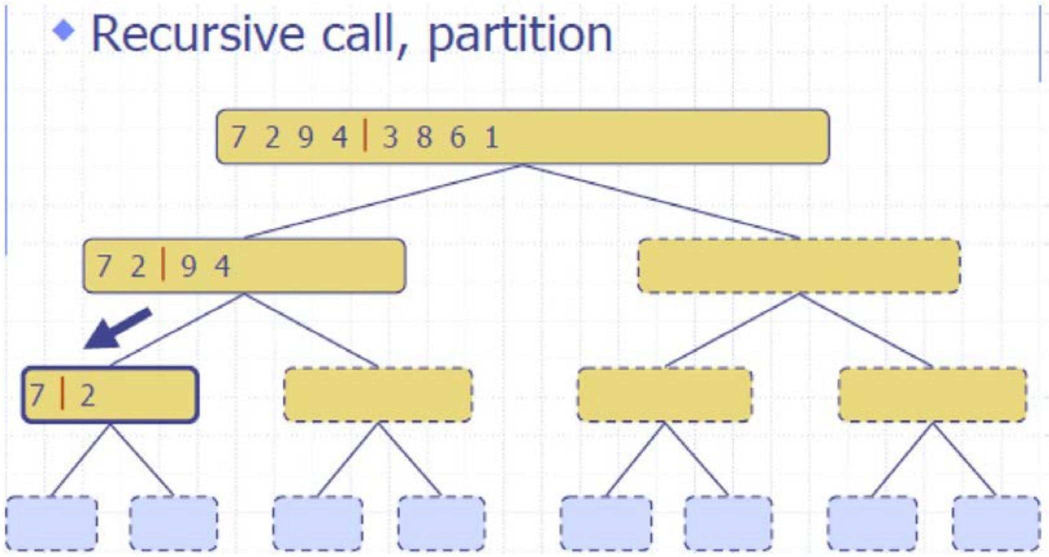


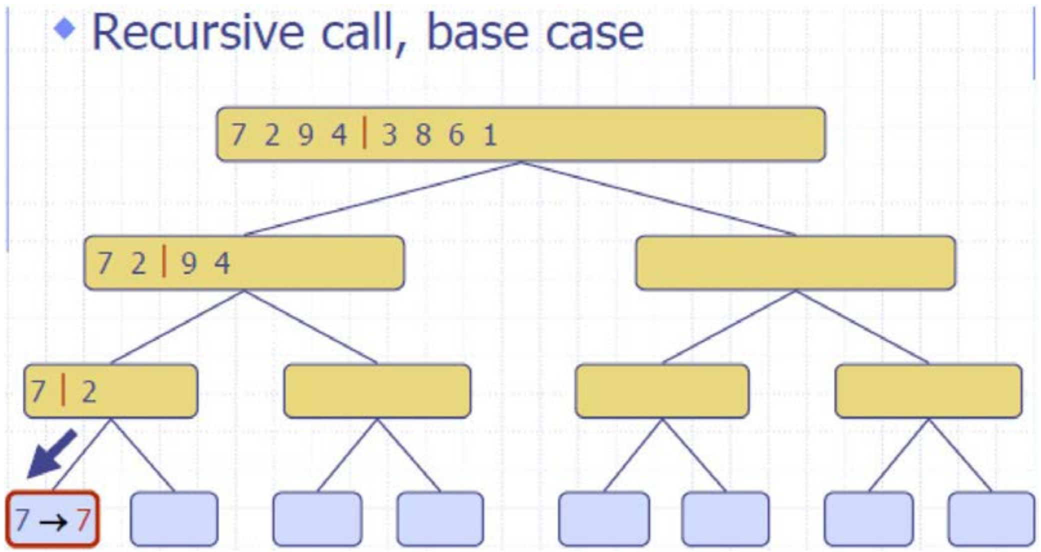
MergeSort - Example



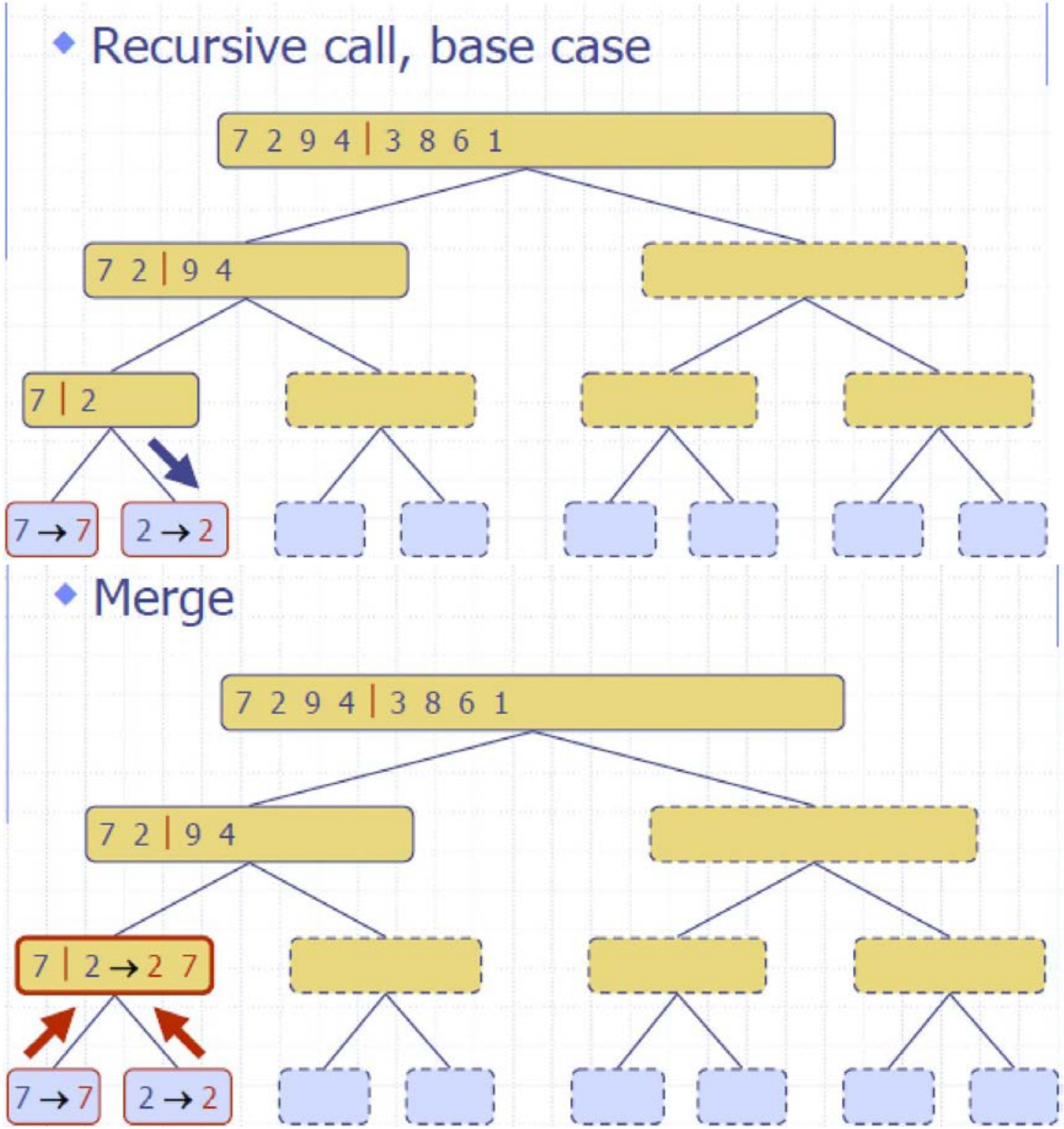


MergeSort - Example

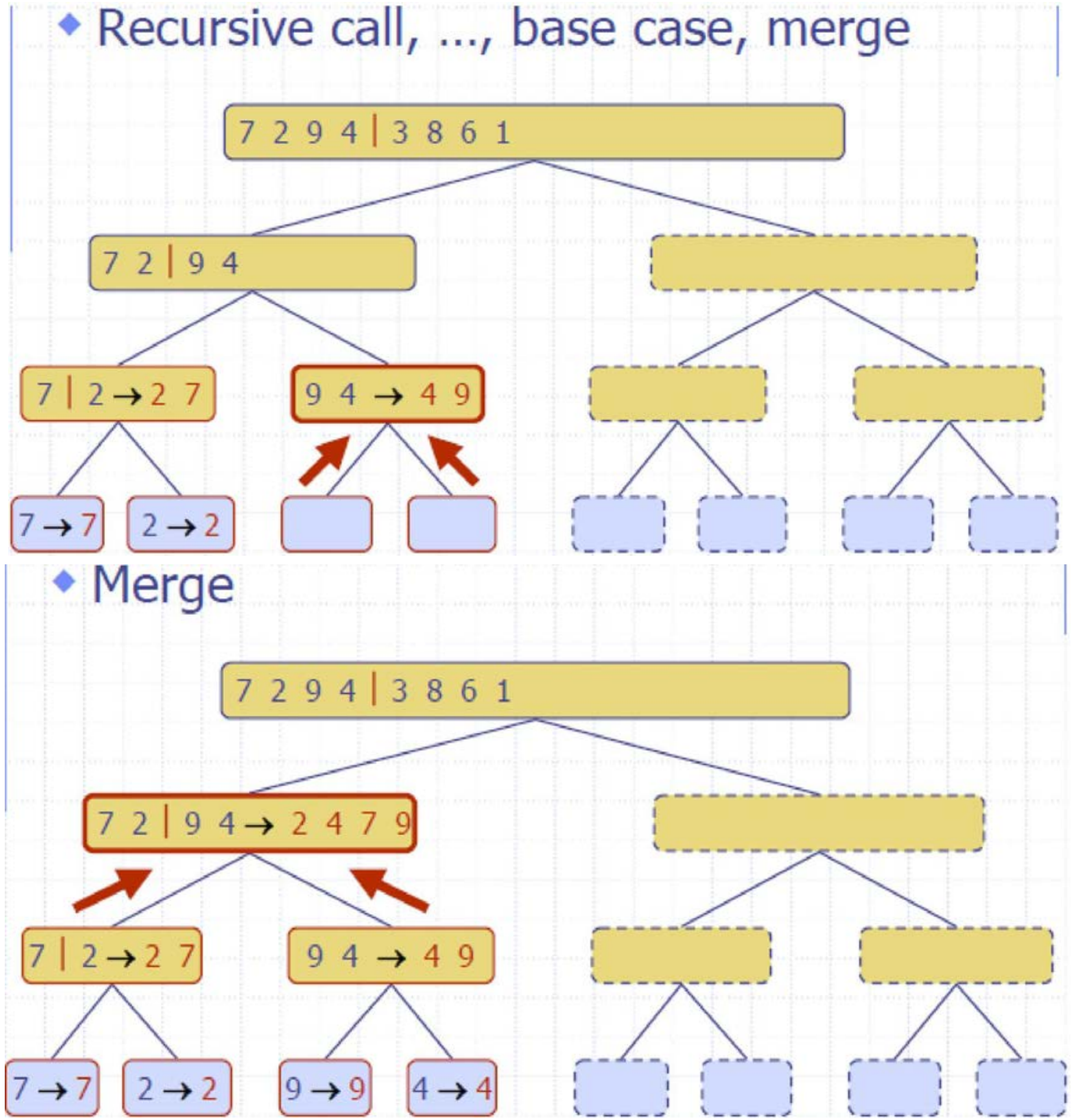




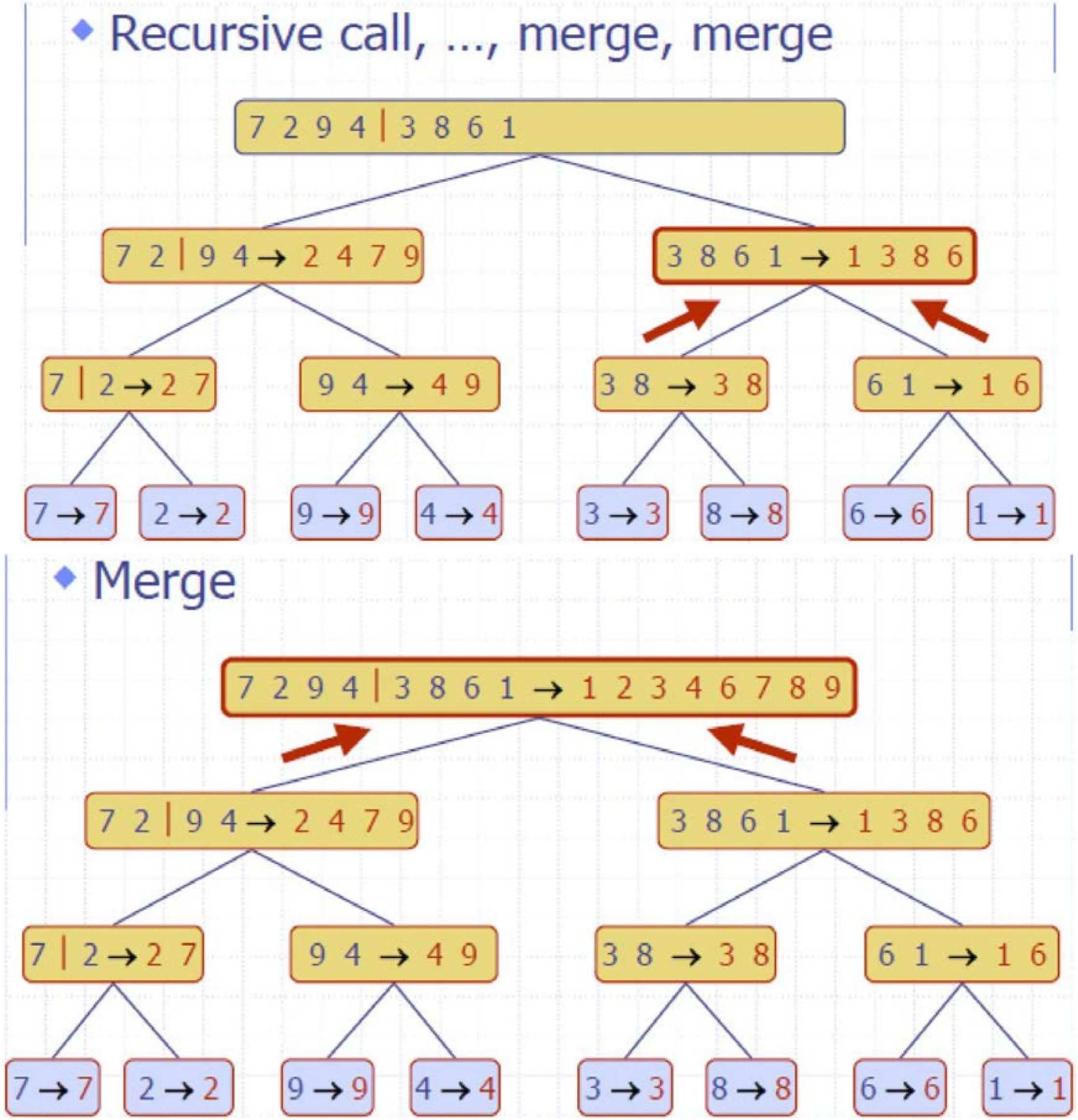
MergeSort - Example



MergeSort - Example

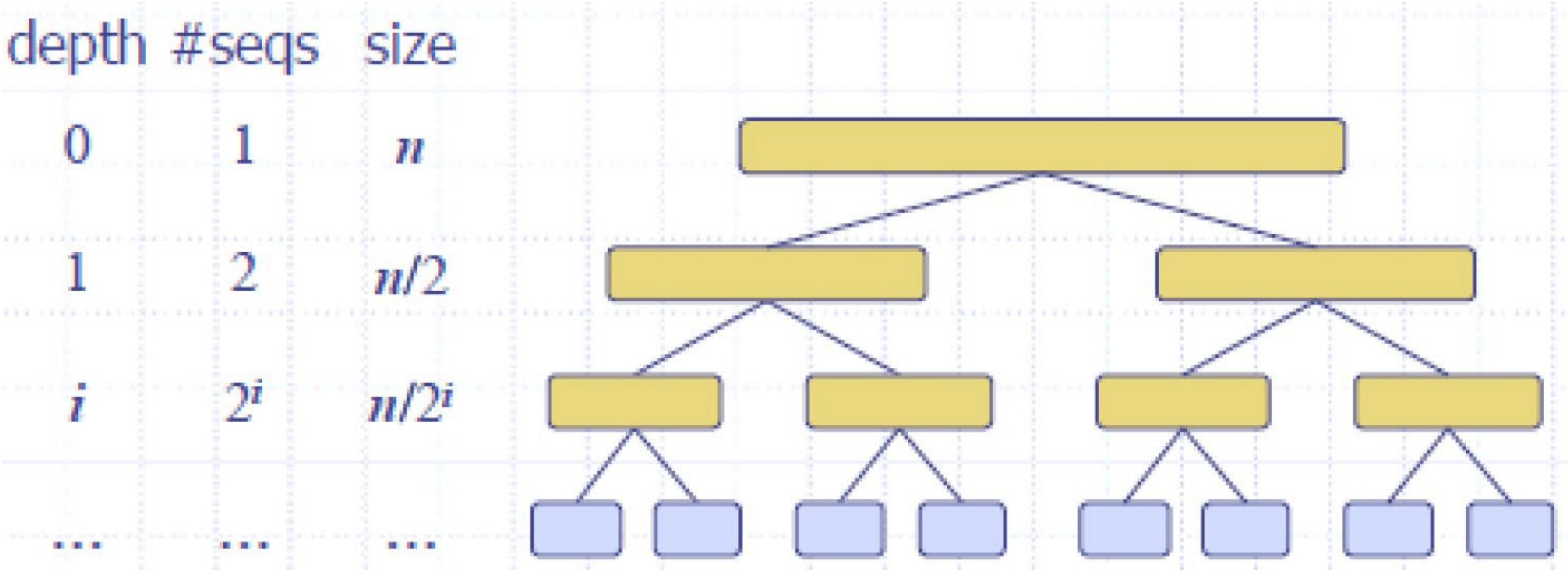


MergeSort - Example

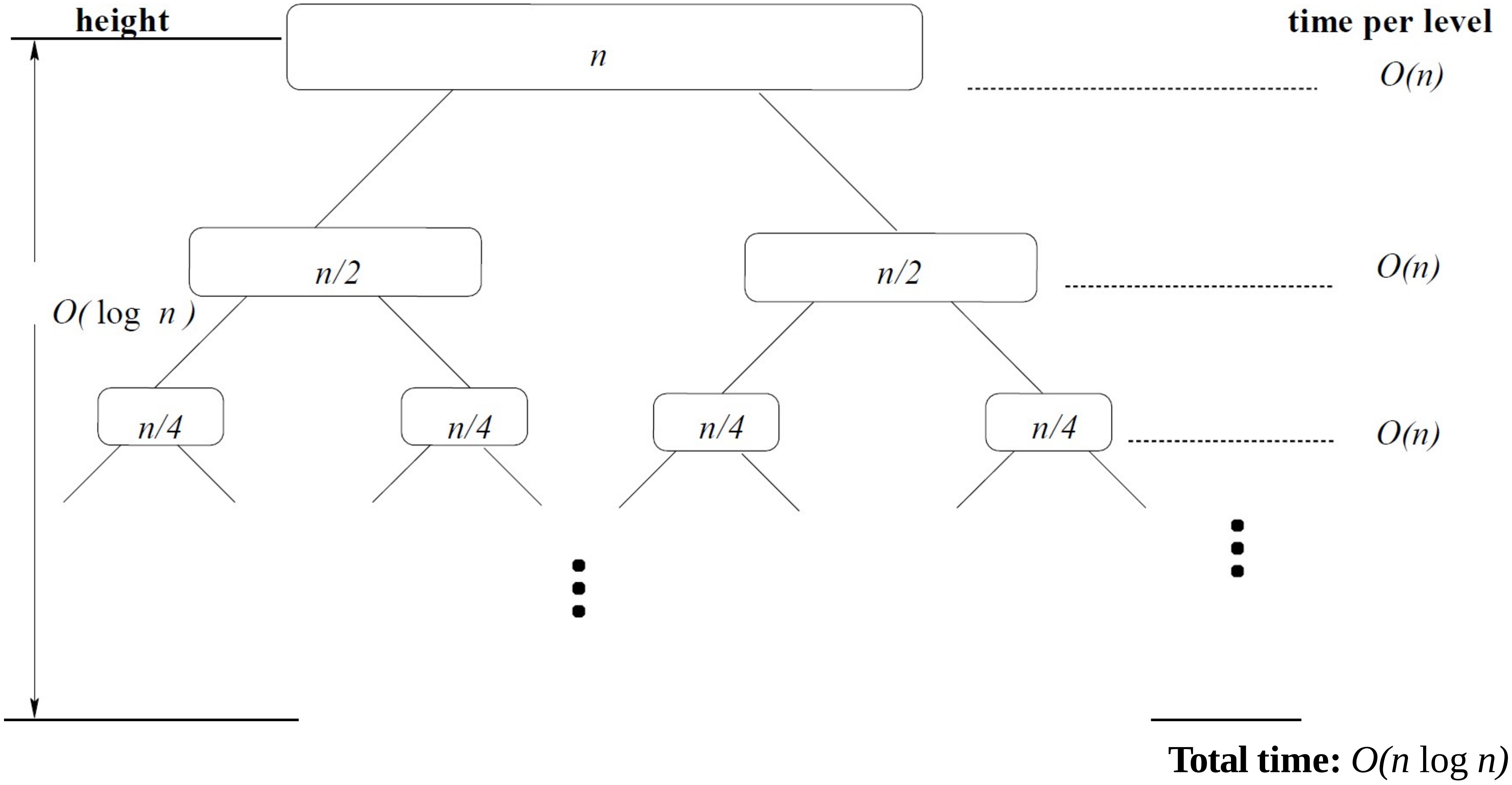


MergeSort - Analysis

- ❖ The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence
- ❖ The overall amount or work done at the nodes of depth i is $O(n)$
- ❖ Thus, the total running time of merge-sort is $O(n \log n)$



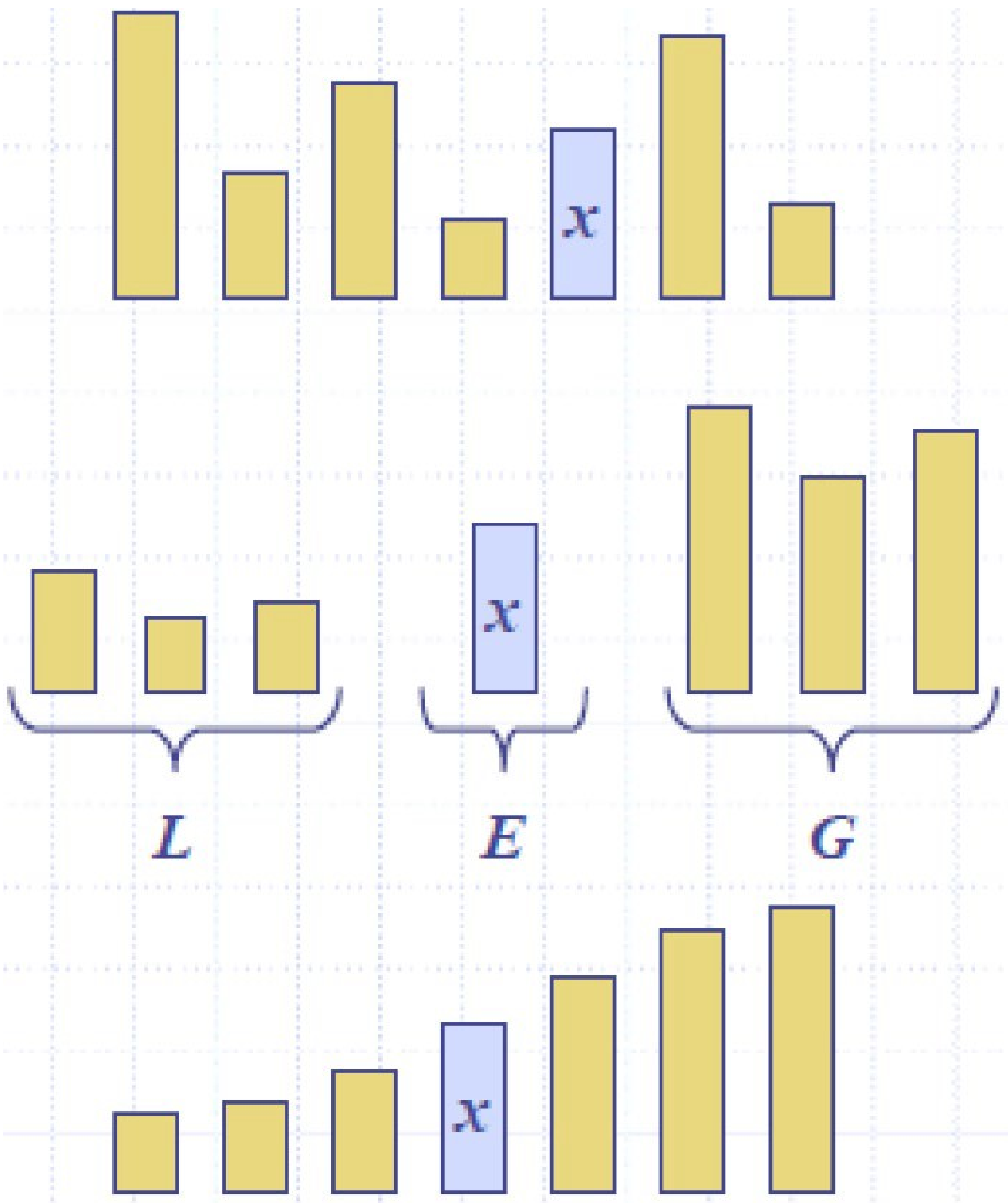
MergeSort - Analysis



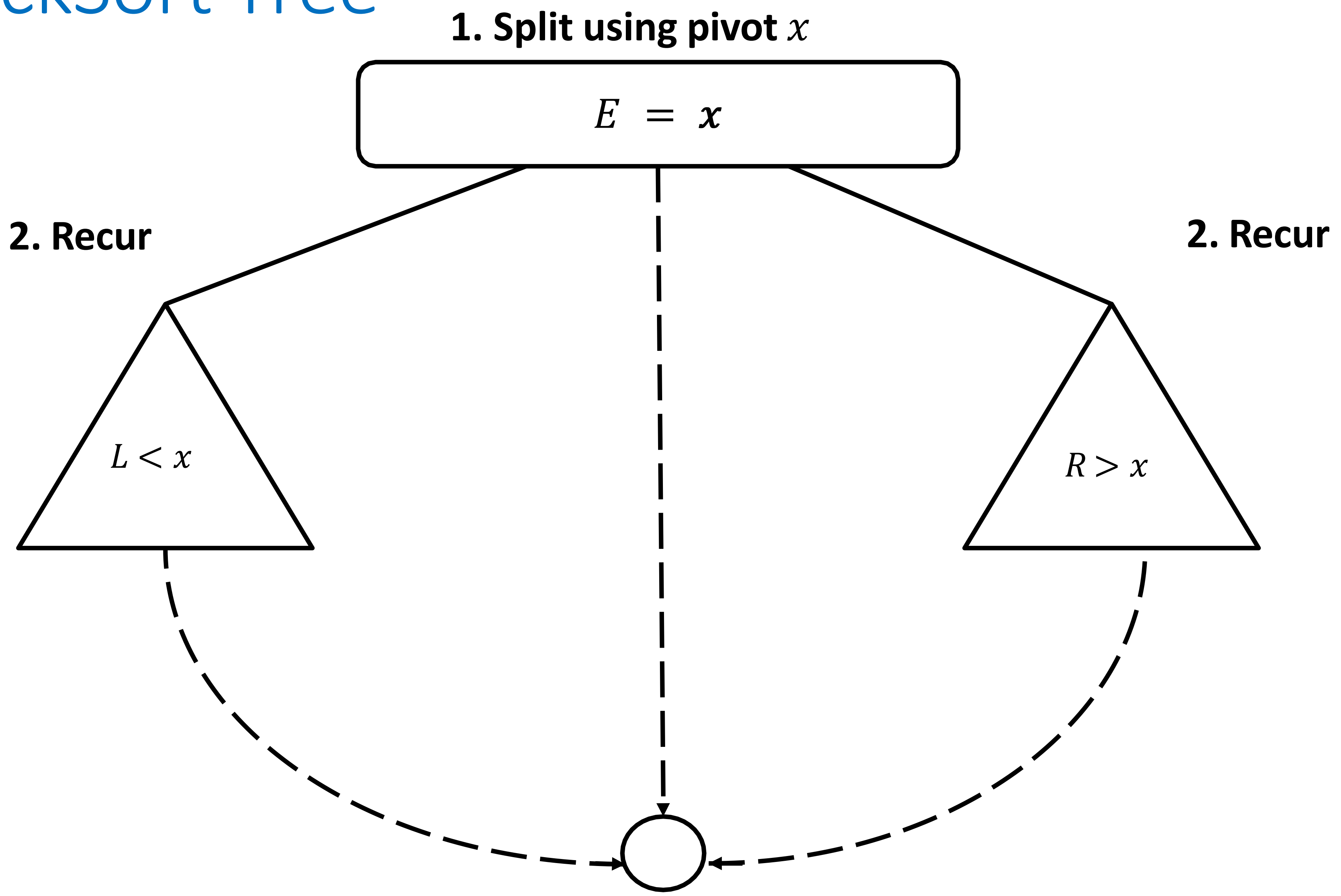
QuickSort

Quick-sort is a randomized sorting algorithm based on the divide-and conquer paradigm:

- Divide: pick a random element x (called pivot) and partition S into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- Recur: sort L and G
- Conquer: join L,E and G



QuickSort Tree



QuickSort -worst-case running time

❖ The worst case for quick-sort occurs when the pivot is the unique **minimum** or **maximum** element

❖ One of L and G has size n-1 and the other has size 0

❖ The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

❖ Thus, the worst-case running time of quick-sort is $O(n^2)$