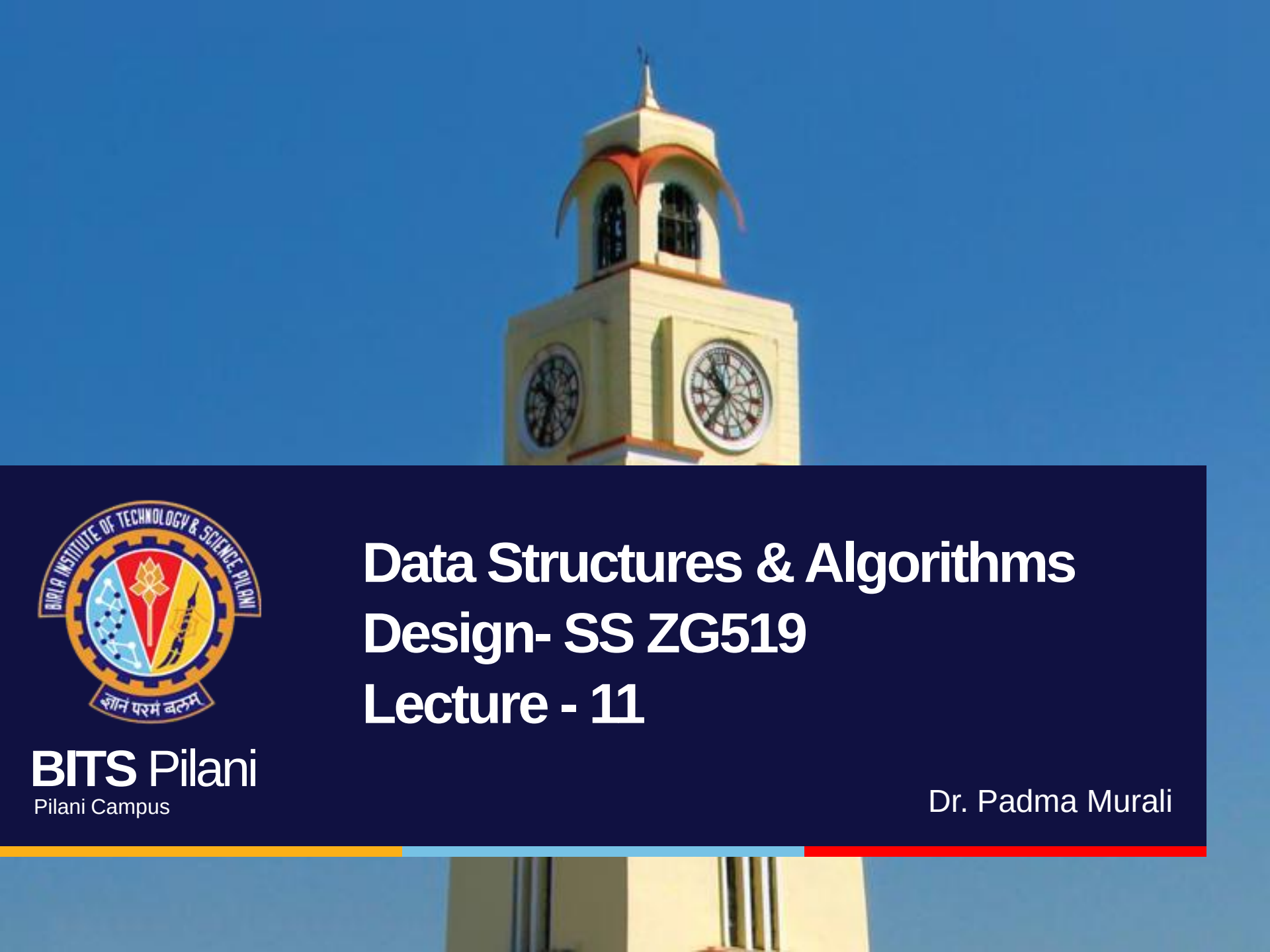


Data Structures & Algorithms Design- SS ZG519
Lecture - 11

BITS Pilani
Pilani Campus

Dr. Padma Murali

Lecture 11 Topics

- Quick Sort
- AVL Trees

RUNNING TIME OF HEAPSORT

RUNNING TIME OF MAX-HEAPIFY

Running time of MAX-HEAPIFY depends on the height h of a heap i.e O(h)

Height of a heap = lg n

T(n) = O(lg n)

RUNNING TIME OF HEAPSORT


```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2
        do exchange A[1] <-> A[i]
                heap-size [A] ← heap-size[A] – 1
                    MAX-HEAPIFY(A,1)
```

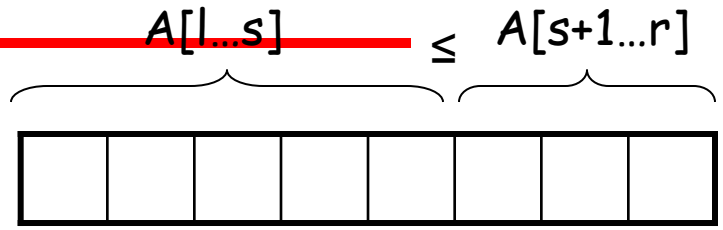
- Running time of BUILD-MAX-HEAP is O(n)
- MAX-HEAPIFY is executed n-1 times with running time O(log n)
- Total Running time of Heap Sort is

  O(n) + n-1(O(log n) = O(n log n)

Quicksort

- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).

Quicksort

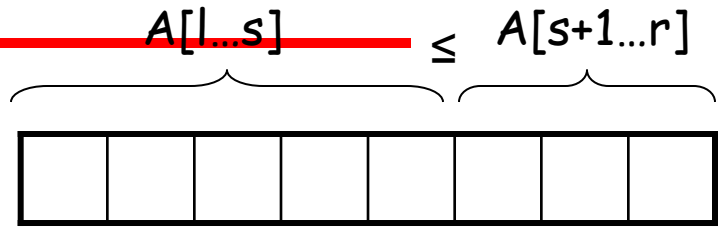


Sort an array $A[l...r]$

Divide

- Partition the array A into 2 subarrays $A[l..s]$ and $A[s+1..r]$, such that each element of $A[l..s]$ is smaller than or equal to each element in $A[s+1..r]$
- Need to find index s to partition the array

Quicksort



Conquer

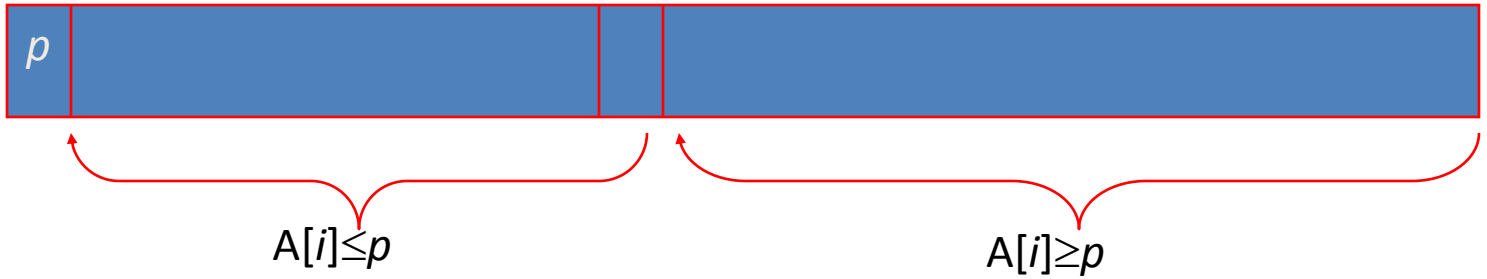
- Recursively sort $A[l..s]$ and $A[s+1..r]$ using Quicksort

Combine

- Trivial: the arrays are sorted in place
- No additional work is required to combine them
- The entire array is now sorted

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quicksort

Initially: l=0, r=n-1

Alg.: QUICKSORT(A, l, r)

if l < r

then s ← PARTITION(A, l, r)

QUICKSORT (A, l, s-1)

QUICKSORT (A, s+1, r)

Recurrence:

T(n) = T(s) + T(n – s) + f(n)

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \cdot p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Worst Case Partitioning

Worst-case partitioning

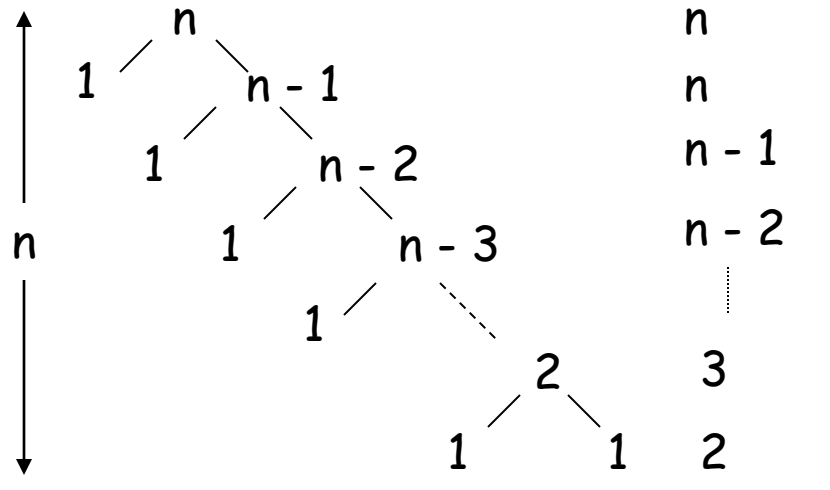
- One region has one element and the other has n – 1 elements
- Maximally unbalanced

Recurrence: q=1

$T(n) = T(1) + T(n-1) + n,$

$T(1) = \Theta(1)$

$T(n) = T(n-1) + n$



$$= \quad n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

$\Theta(n^2)$

Best Case Partitioning

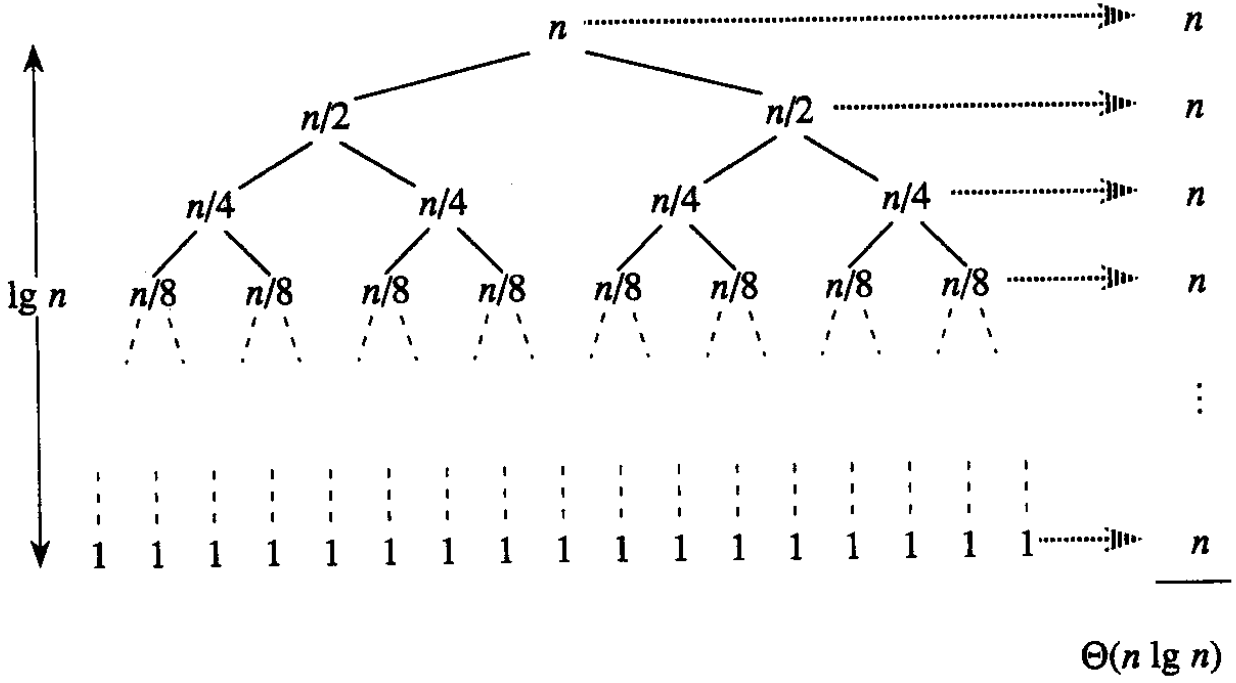
Best-case partitioning

- Partitioning produces two regions of size $n/2$

Recurrence: q=n/2

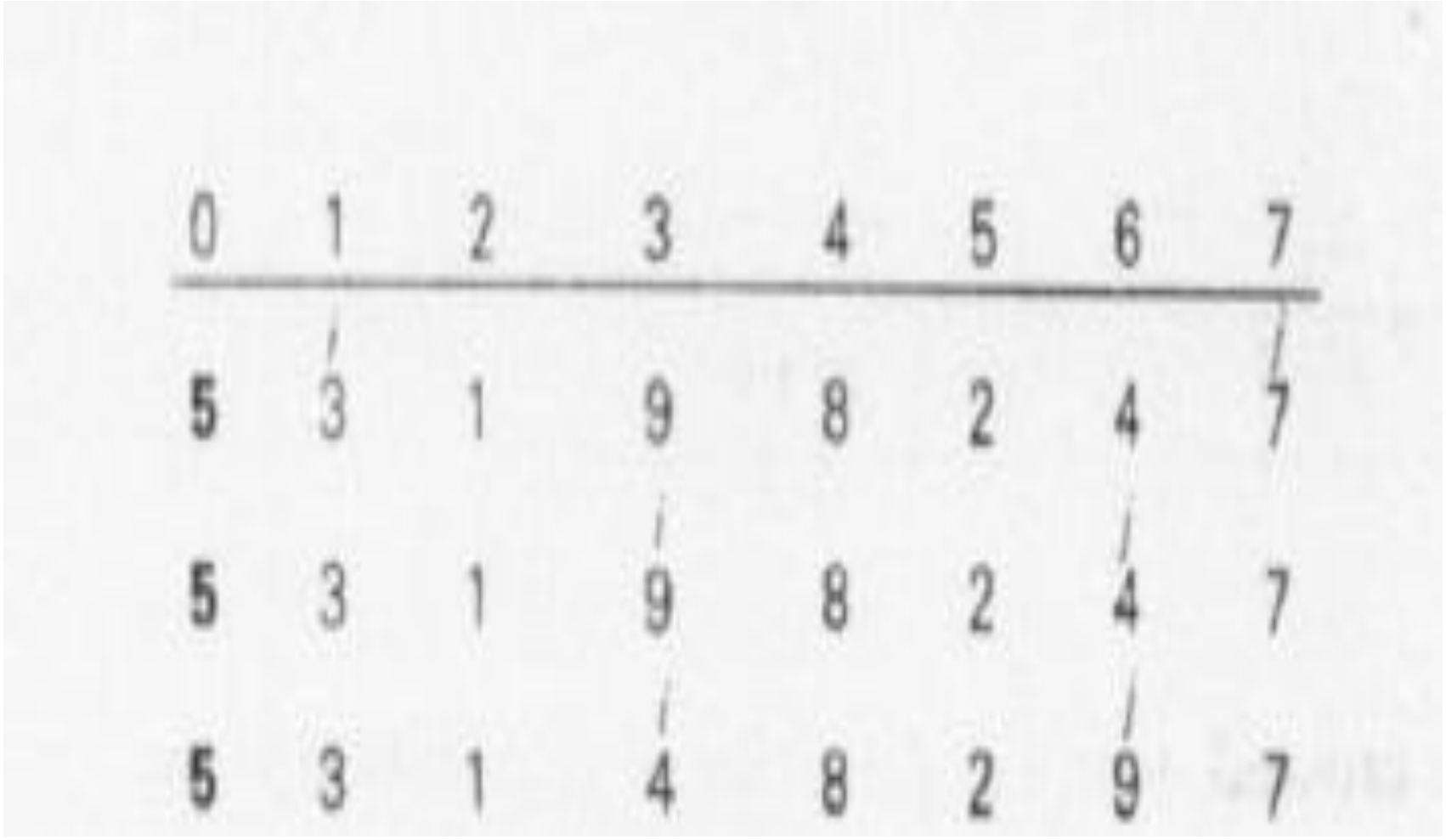
$T(n) = 2T(n/2) + \Theta(n)$

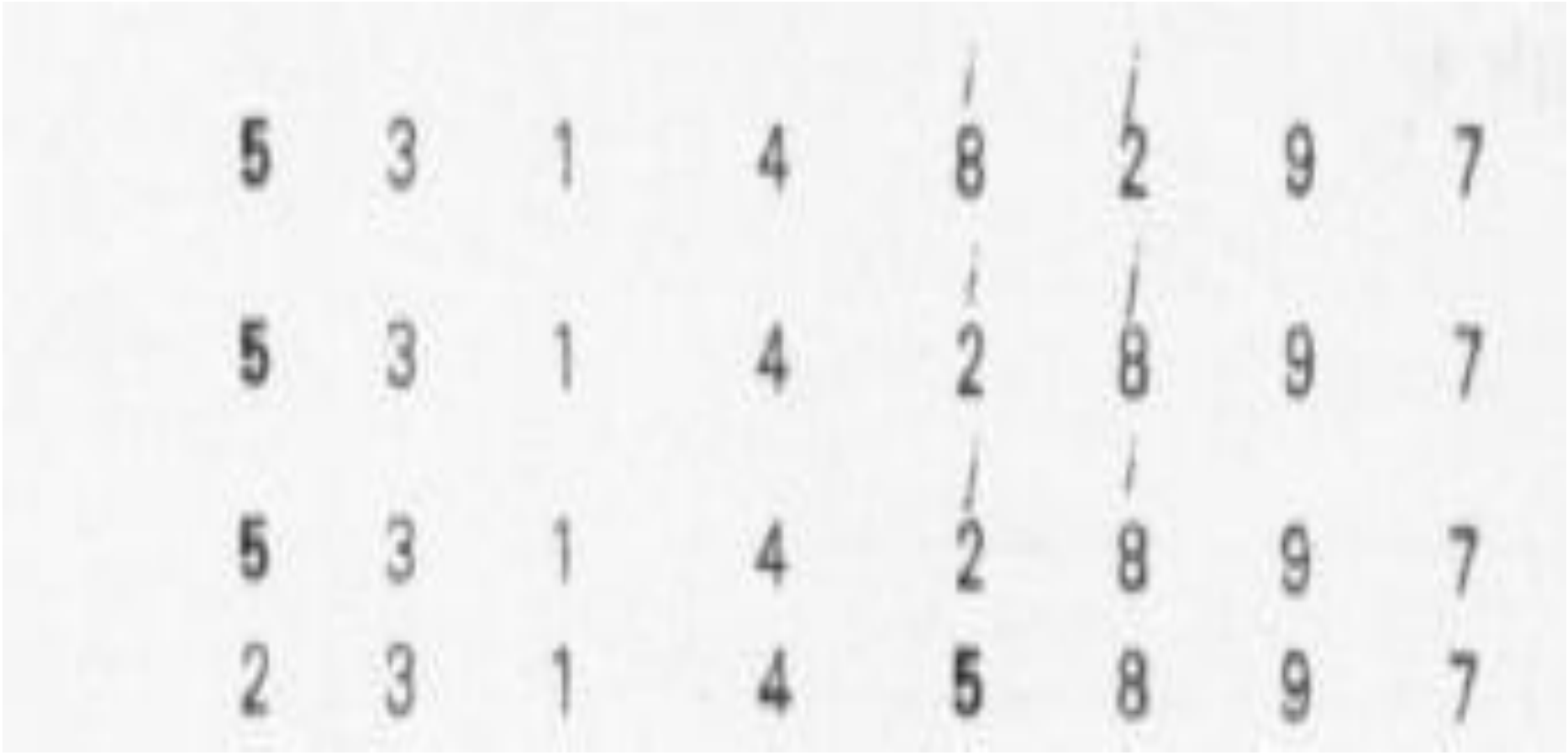
$T(n) = \Theta(n \lg n)$ (Master theorem)

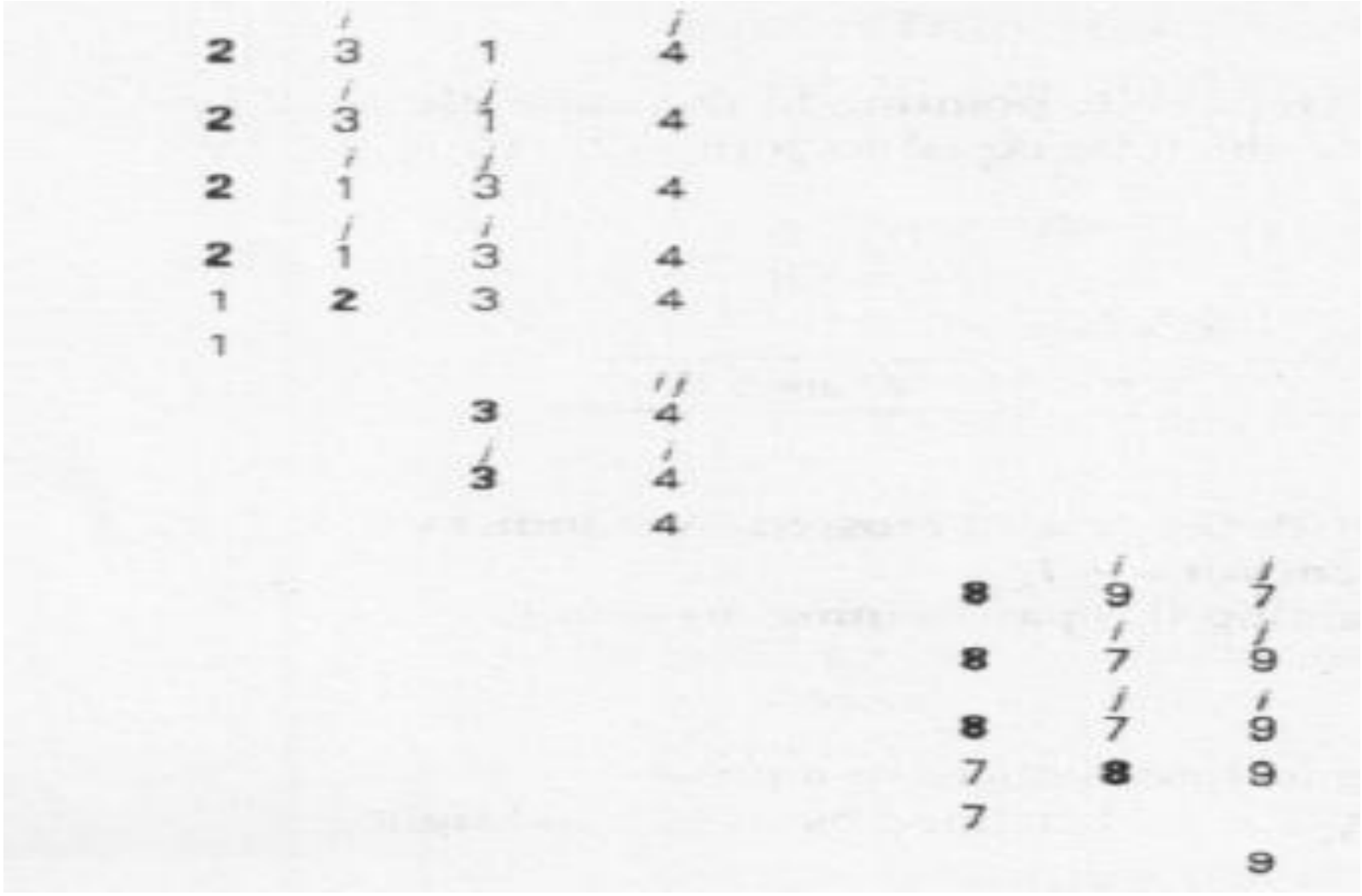


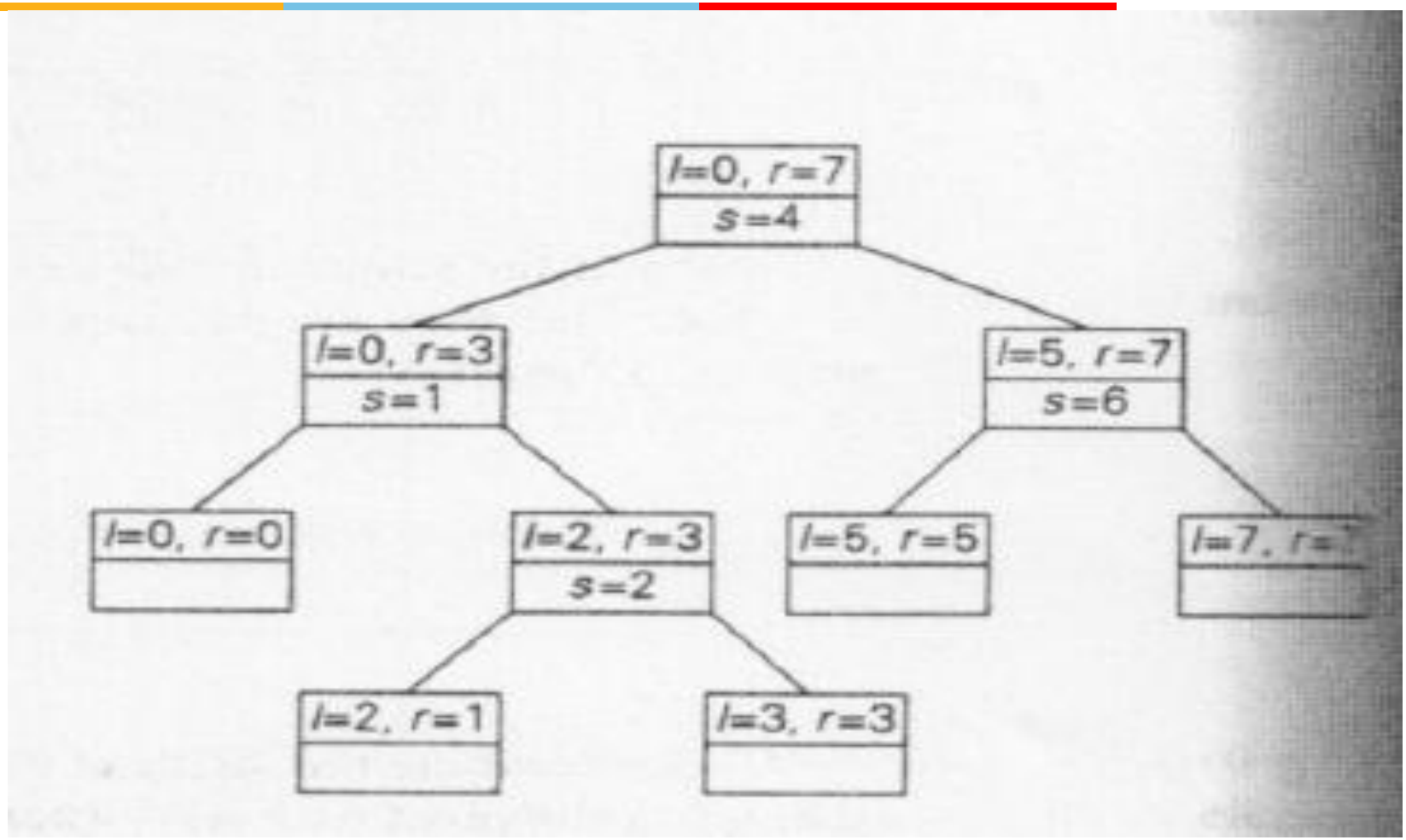
Quicksort Example

5 3 1 9 8 2 4 7


0 1 2 3 4 5 6 7
5 3 1 9 8 2 4 7
5 3 1 9 8 2 4 7
5 3 1 4 8 2 9 7


5 3 1 4 8 2 9 7
5 3 1 4 2 8 9 7
5 3 1 4 2 8 9 7
2 3 1 4 5 8 9 7




l=0, r=7
s=4
l=0, r=3
s=1
l=5, r=7
s=6
l=0, r=0
l=2, r=3
s=2
l=5, r=5
l=7, r=
l=2, r=1
l=3, r=3

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Quicksort

- Similar to mergesort - divide-and-conquer recursive algorithm
- One of the fastest sorting algorithms
- Average running time O(NlogN)
- Worst-case running time $O(N^2)$

Quicksort

- Pick one element in the array, which will be the *pivot*.
- Make one pass through the array, called a *partition* step, re-arranging the entries so that:
  - the pivot is in its proper place.
  - entries smaller than the pivot are to the left of the pivot.
  - entries larger than the pivot are to its right.
- Recursively apply quicksort to the part of the array that is to the left of the pivot,
  and to the right part of the array.
- Here we don't have the merge step, at the end all the elements are in the proper order.

Quicksort

innovate achieve lead

•**Algorithm**

STEP 1. Choosing the pivot

Choosing the pivot is an essential step.

Depending on the pivot the algorithm may run very fast, or in quadratic time.**:**

1.Some fixed element: e.g. the first, the last, the one in the middle

This is a bad choice - the pivot may turn to be the smallest or the largest element,

then one of the partitions will be empty.

2.Randomly chosen (by random generator) - still a bad choice.

3.The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.

4.The median-of-three choice: take the first, the last and the middle element.

Choose the median of these three elements.

Quicksort

innovate achieve lead

- **Algorithm**

STEP 1. Choosing the pivot

Choosing the pivot is an essential step.

Depending on the pivot the algorithm may run very fast, or in quadratic time.**:**

1. Some fixed element: e.g. the first, the last, the one in the middle

This is a bad choice - the pivot may turn to be the smallest or the largest element,
then one of the partitions will be empty.

2. Randomly chosen (by random generator) - still a bad choice.
3. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.
4. The median-of-three choice: take the first, the last and the middle element.
   Choose the median of these three elements.

Quicksort

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

The first element is 8, the middle - 10, the last - 5.

The median of [8, 10, 5] is **8**

STEP 2. Partitioning

Partitioning is illustrated on the above example.

1. The first action is to get the pivot out of the way - swap it with the last element

5, 3, 25, 6, 10, 17, 1, 2, 18, **8**

2. We want larger elements to go to the right and smaller elements to go to the left.

Two "fingers" are used to scan the elements from left to right and from right to left:

Quicksort

[5, 3, 25, 6, 10, 17, 1, 2, 18, 8]

•i j

•While **i** is to the left of **j**, we move **i** right, skipping all the elements less than the pivot.

•If an element is found greater then the pivot, **i** stops

•While **j** is to the right of **i**, we move **j** left, skipping all the elements greater than the pivot.

• If an element is found less then the pivot, **j** stops

•When both **i** and **j** have stopped, the elements are swapped.

•When **i** and **j** have crossed, no swap is performed, scanning stops, and the element pointed to by **i** is swapped with the pivot .

In the example the first swapping will be between 25 and 2, the second between 10 and 1.

Quicksort

Restore the pivot.
After restoring the pivot we obtain the following partitioning into three groups:
[5, 3, 2, 6, 1] [8] [10, 25, 18, 17]
STEP 3. Recursively quicksort the left and the right parts

Quicksort

- **Complexity of Quicksort**

Worst-case: $O(N^2)$

This happens when the pivot is the smallest (or the largest) element.

Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Best-case O(NlogN) The best case is when the pivot is the median of the array,
and then the left and the right part will have same size.
There are **logN** partitions, and to obtain each partitions we do **N** comparisons
(and not more than **N/2** swaps). Hence the complexity is **O(NlogN)**
Average-case - O(NlogN)

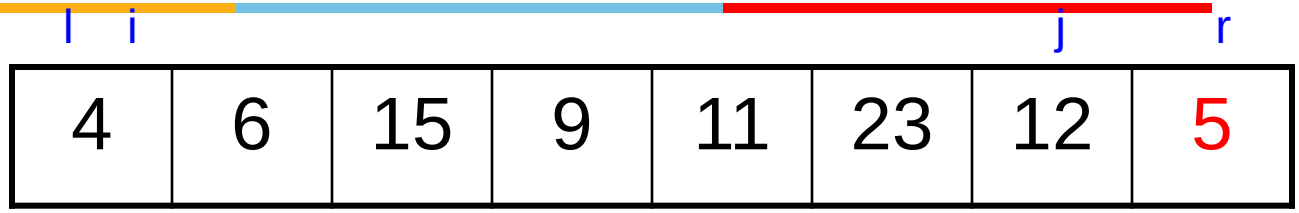
Quicksort

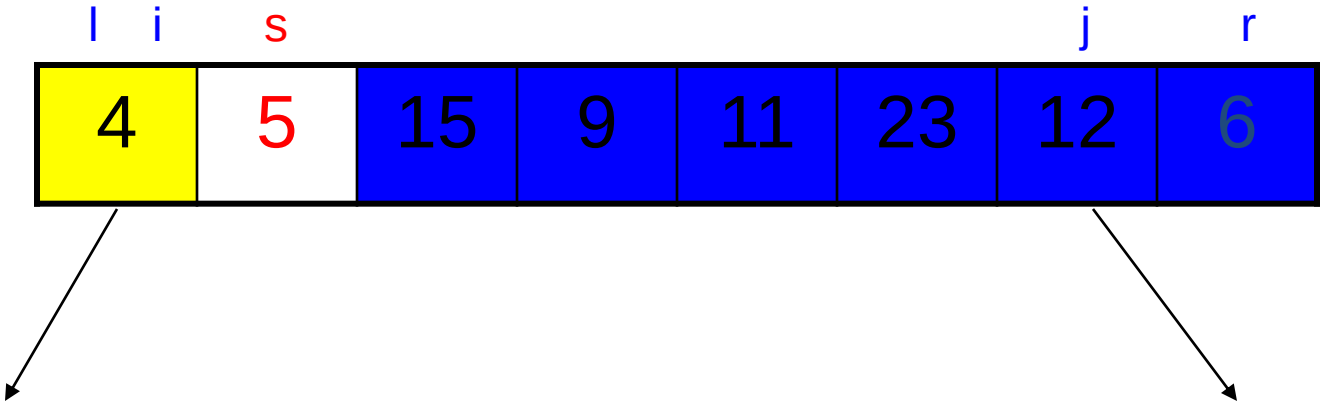
- **Advantages**:
- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing). Compare with mergesort: mergesort needs additional memory for merging.
- **Disadvantages:** The worst-case complexity is $O(N^2)$
- **Applications:**
- Commercial applications use Quicksort - generally it runs fast, no additional memory,
this compensates for the rare occasions when it runs with $O(N^2)$

Quicksort

- **Comparison with mergesort:**
- mergesort guarantees O(NlogN) time, however it requires additional memory with size N.
- quicksort does not require additional memory, however the speed is not quaranteed
- usually mergesort is not used for main memory sorting, only for external memory sorting.
- So far, our best sorting algorithm has **O(*n*log *n*)** performance:

Example of partitioning





Quicksort(A,1, 1)

Quicksort(A,3,8)

Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11

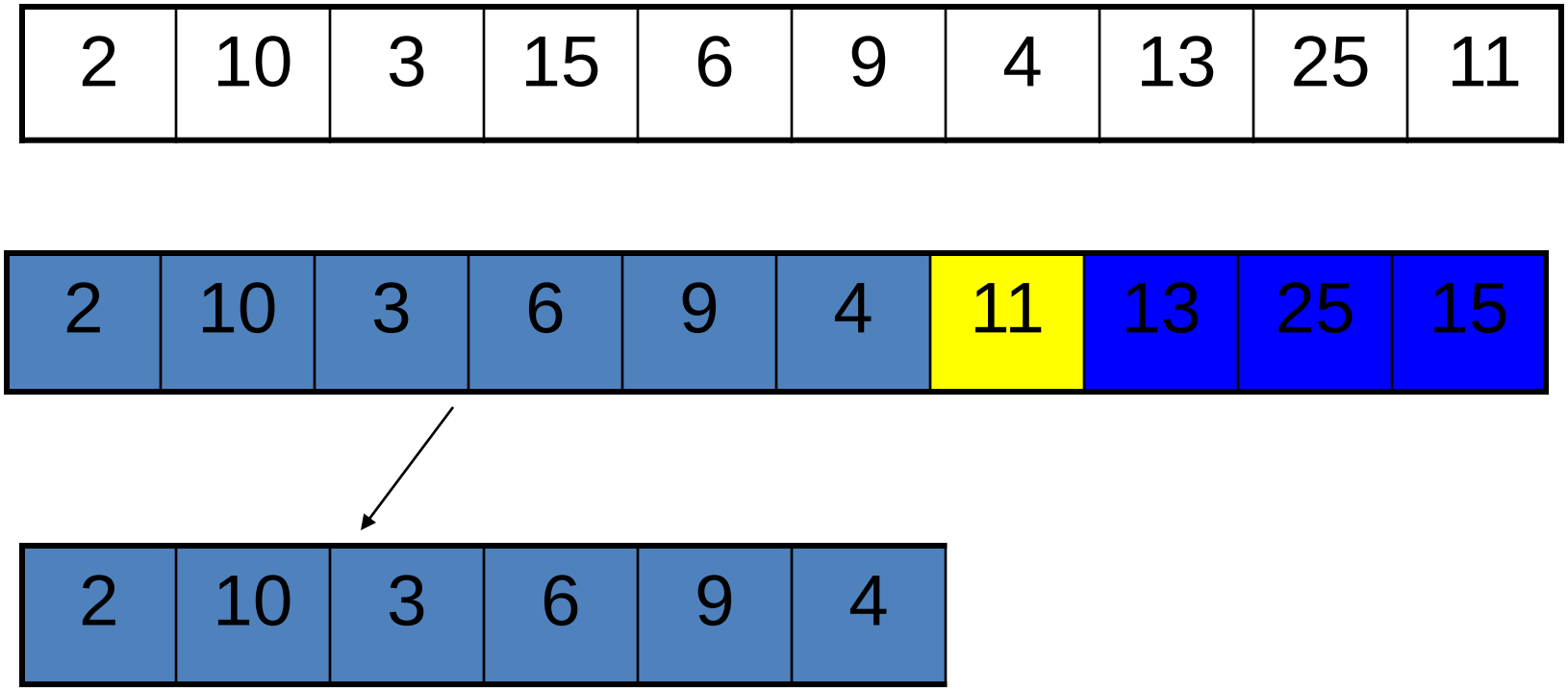
Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11

2	10	3	15	6	9	4	13	25	11

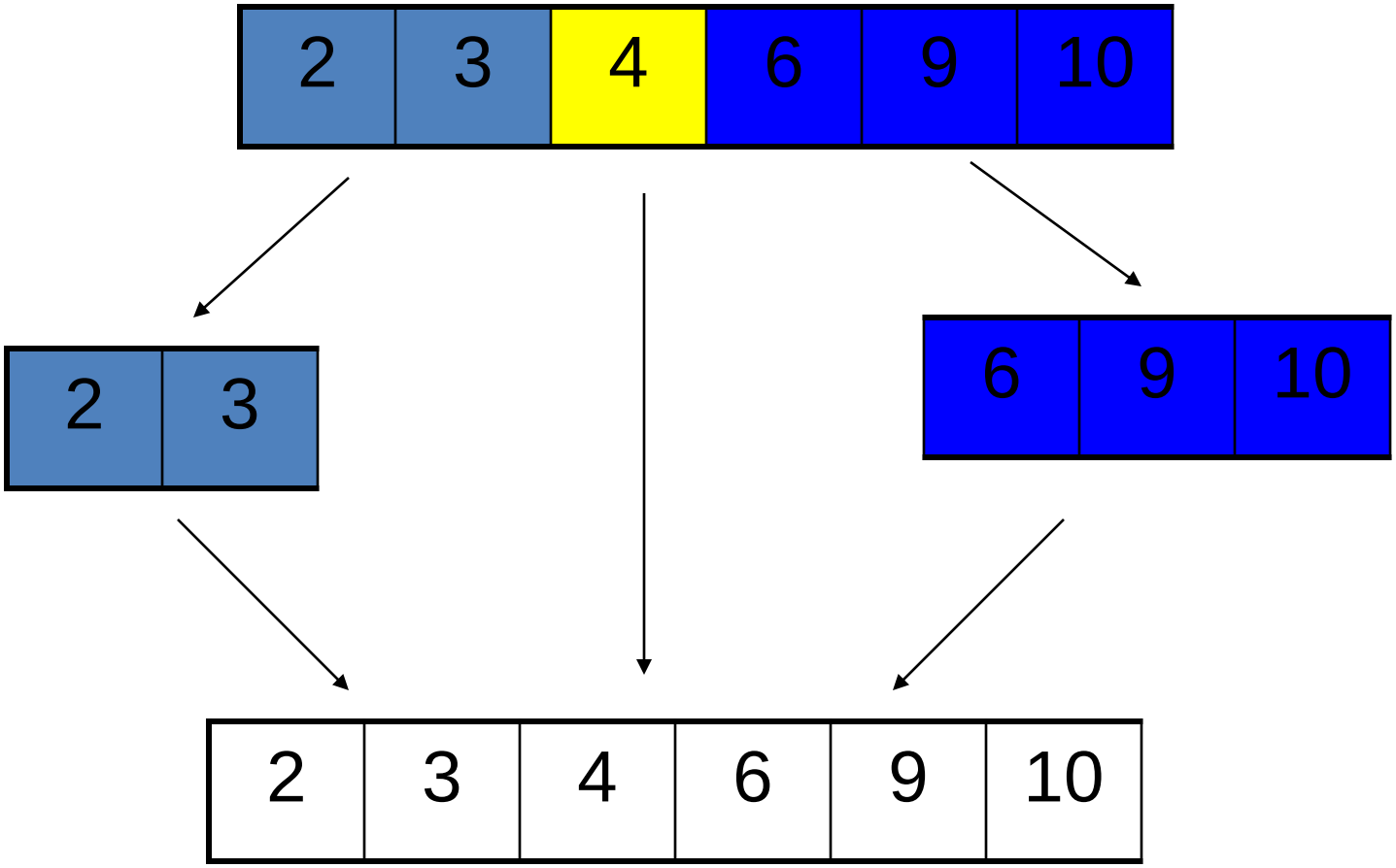
Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11



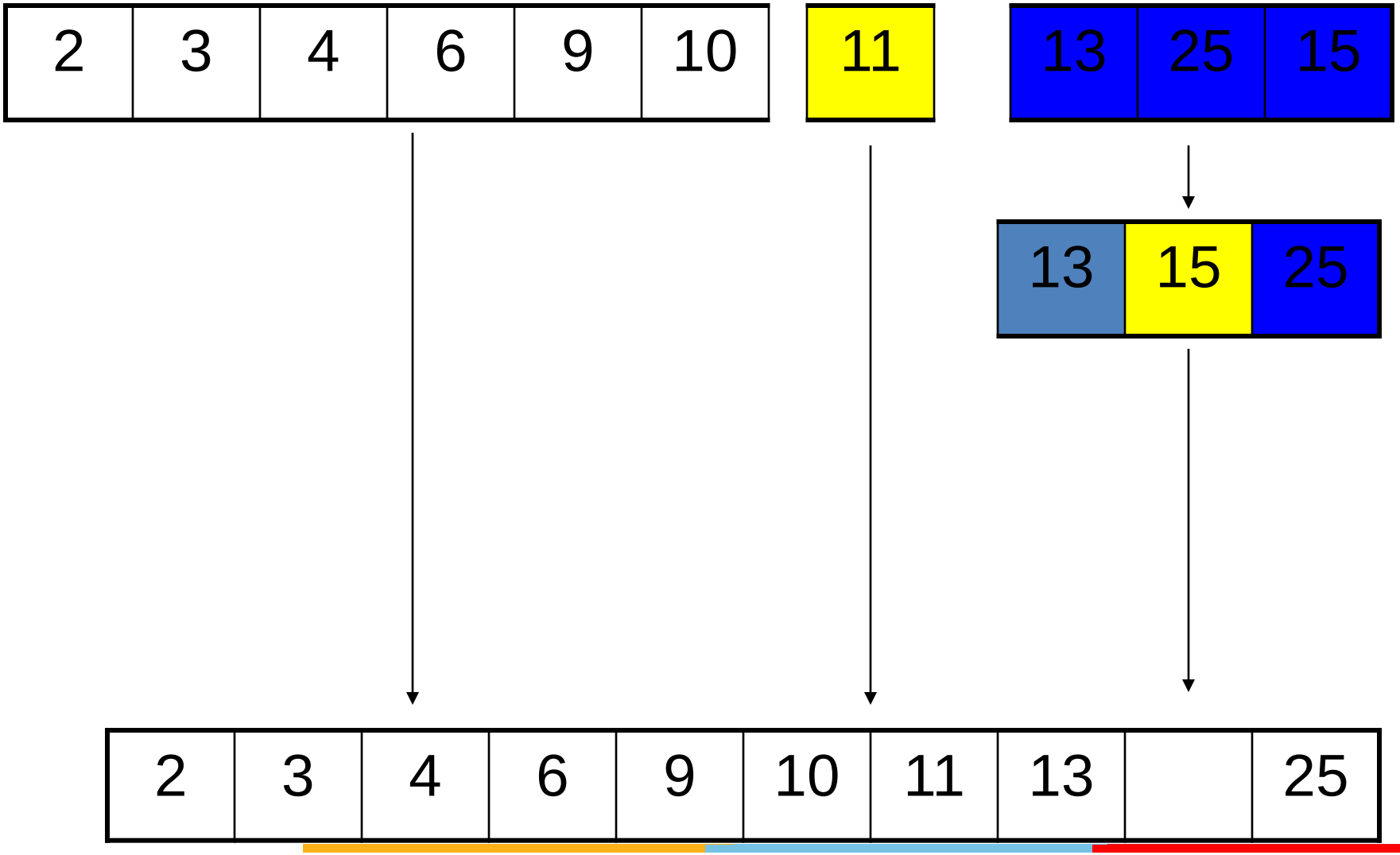
Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11



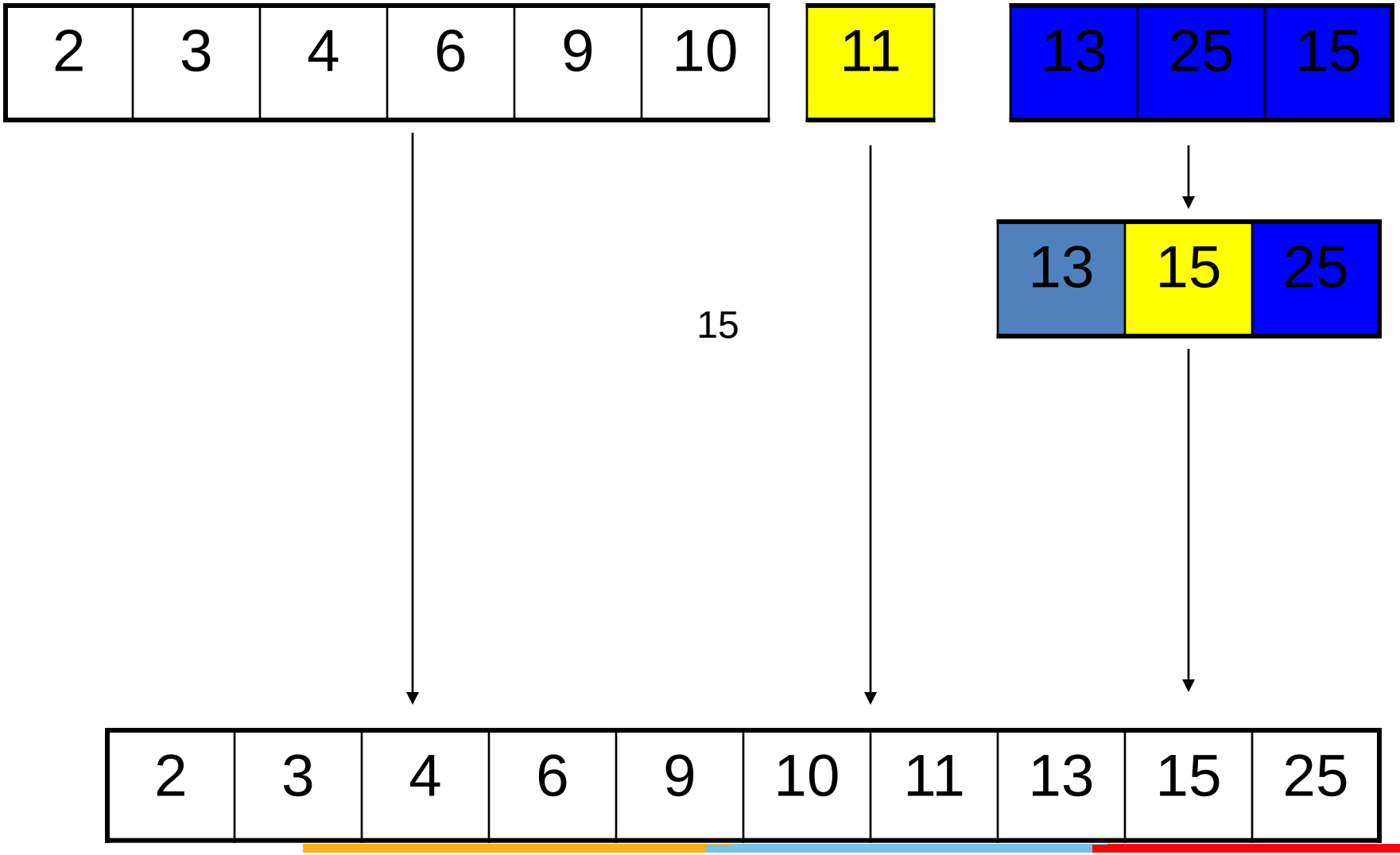
Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11



Quicksort

Sort the array 2,10,3,15,6,9,4,13,25,11



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

data[0], data[n/2], and data[n-1].

Use this median value as pivot.

innovate achieve lead

BITS Pilani
Pilani Campus

- AVL (Adelson-Velski & Landis)Trees

Balanced Binary Search Tree

- Worst case height of binary search tree: N-1
  - Insertion, deletion can be O(N) in the worst case
- We want a tree with small height
- Height of a binary tree with N node is at least $\Theta(\log N)$
- Goal: keep the height of a binary search tree $O(\log N)$
- Balanced binary search trees
  - Examples: AVL tree, red-black tree

AVL Tree

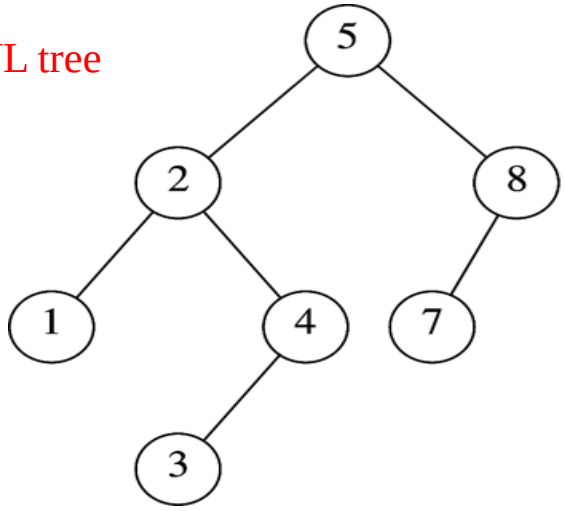
An AVL tree is a binary search tree in which

- for *every* node in the tree, the height of the left and right subtree differ by at most 1.

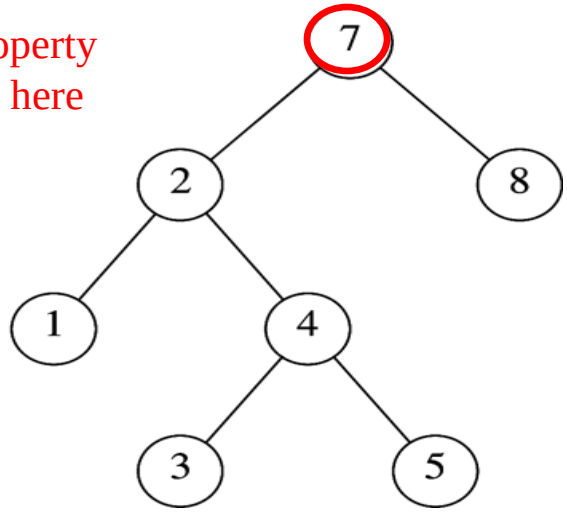
Height of subtree: Max # of edges to a leaf

Height of an empty subtree: -1

AVL tree



AVL property violated here



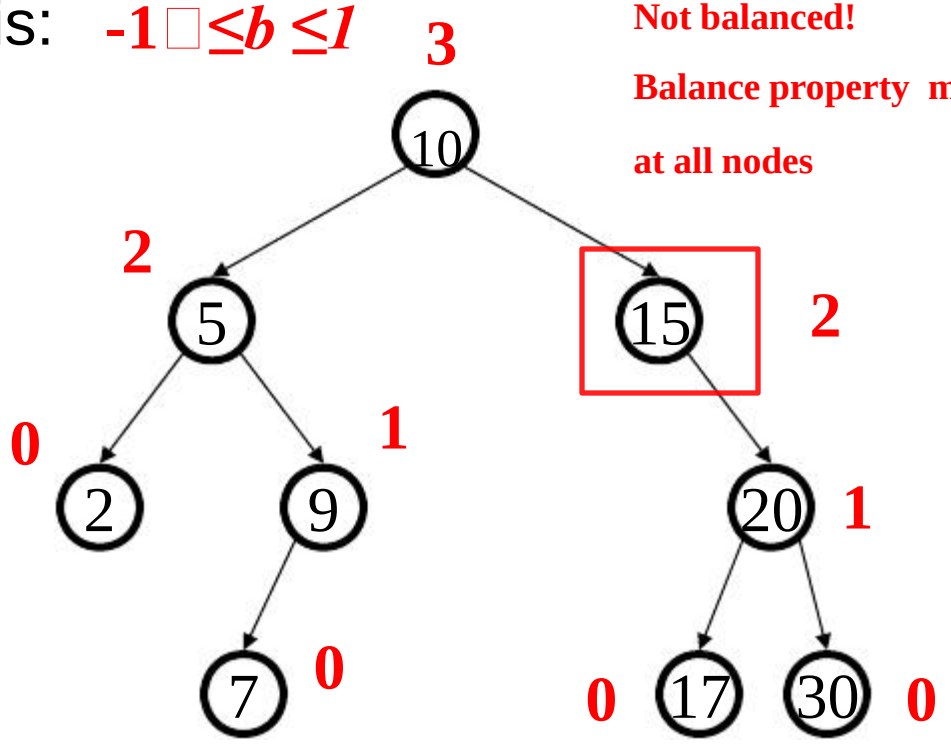
Balance Property

Balance property

b = height (left subtree) - height(right subtree)

– balance of every node is: $-1 \leq b \leq 1$

– depth is θ**(log n)**



Not balanced!

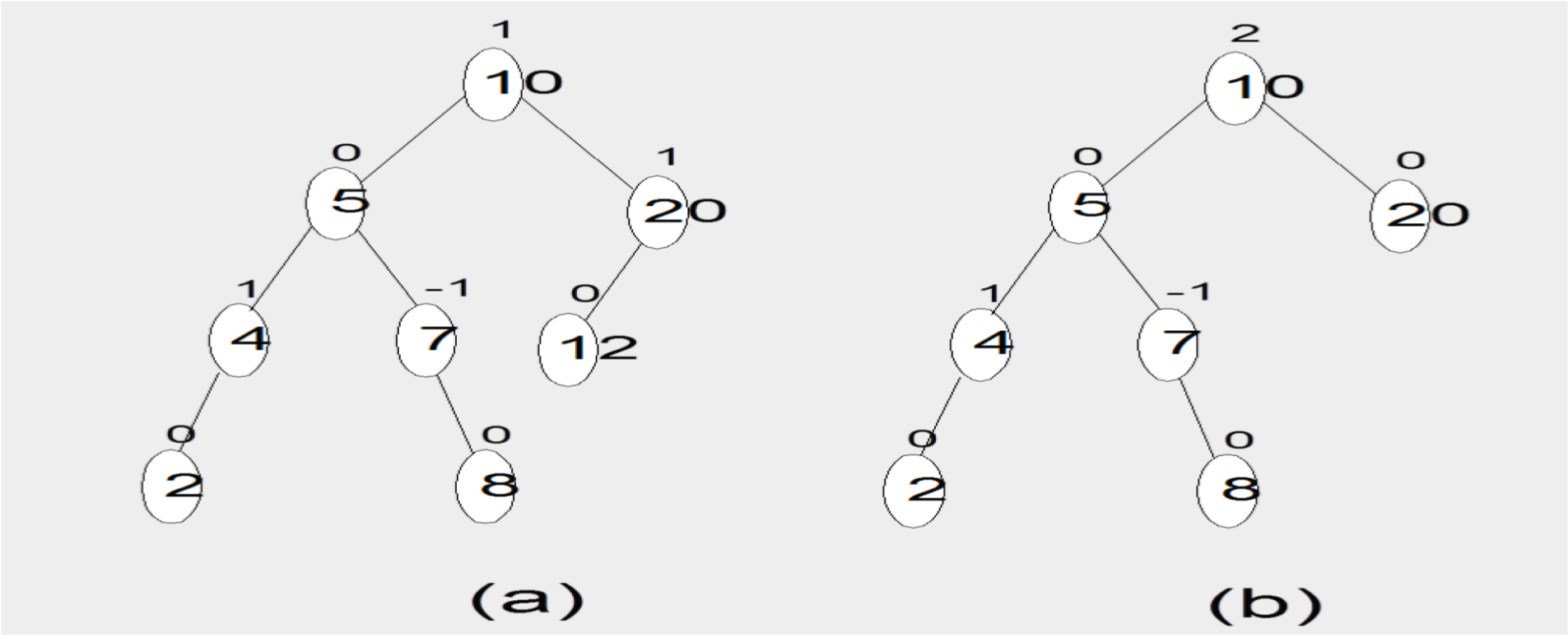
Balance property must hold at all nodes

Balanced Trees: AVL Trees

An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)
the *balance factor*, is at most 1 (with the height of an empty tree

defined as -1)

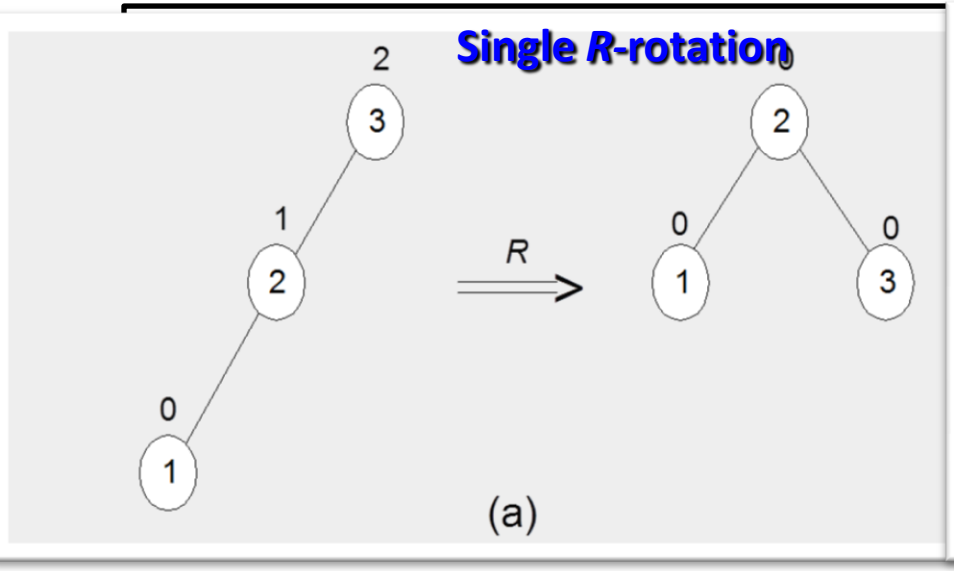
Balanced Trees: AVL Trees

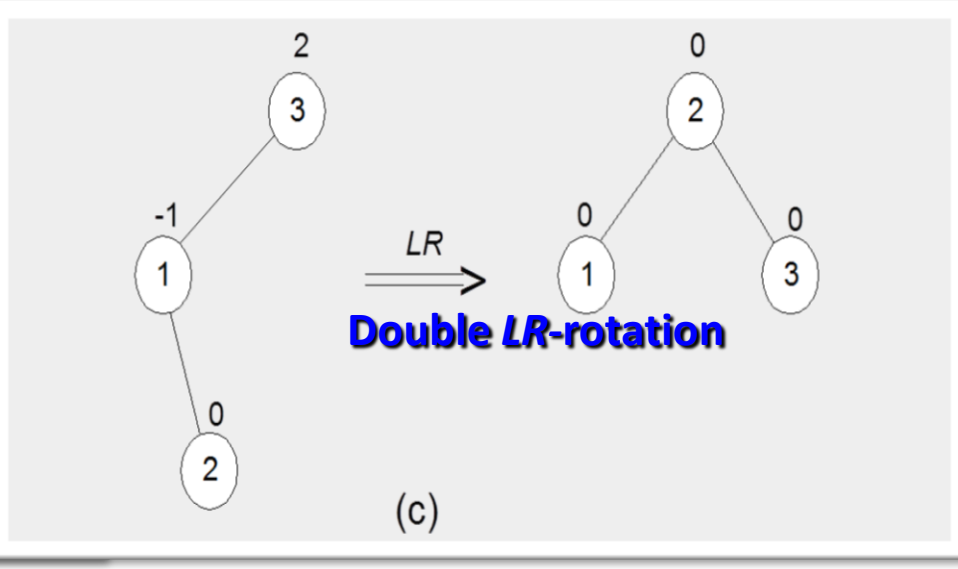


Tree (a) is an AVL tree; tree (b) is not an AVL tree

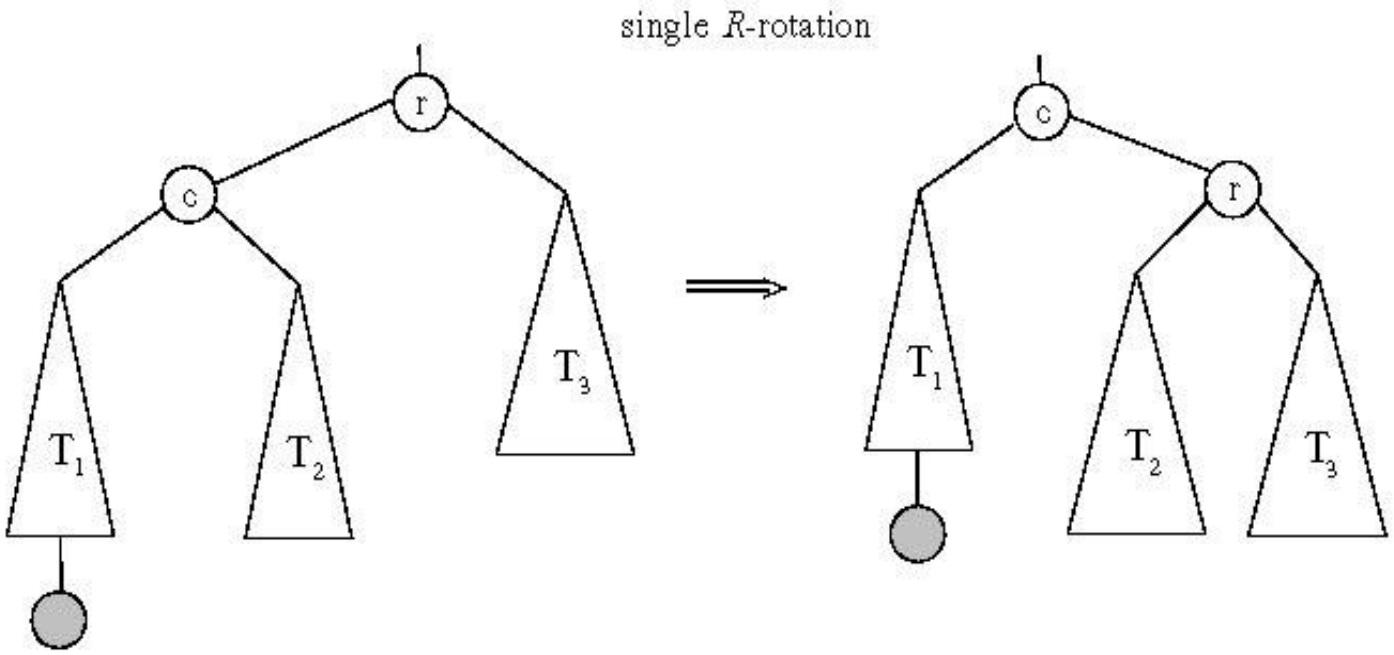
Rotations

If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations*. (The rotation is always performed for a subtree rooted at an "unbalanced" node closest to the new leaf.)





General case: Single R-rotation



General case: Double LR-rotation