562.613 APPLIED DATA STRUCTURES

Lecture 08

Dr. M. G. Abbas Malik
muhammad.malik@manukau.ac.nz

Lecture 09

- Heap Data Structure
- Heap Sort algorithm
- Merge Sort and it algorithmic analysis
- Counting Sort and it algorithmic analysis

Heap

- A Specialized Tree-based Data Structure that satisfies the heap property:
  1. If node B is a child of Node A, then key(A) ≥ key(B) – MAX-HEAP
  2. If node B is a child of Node A, then key(A) ≤ key(B) – MIN-HEAP
- In MAX-HEAP, the largest element is always in the root of the tree
- In MIN-HEAP, the smallest element is always in the root of the tree

Heap

- Term coined in the context of Heap Sort
- Not a garbage-collected storage – LISP and JAVA
- Can be viewed as a nearly complete binary tree
- The tree is completely filled on all levels except possibly the lowest
- Suitable for implementing Priority Queue

Heap

Definition

- A heap is a **Binary Tree** with unique keys assigned to all nodes, meeting the properties:

1. **Shape Property**: Essentially a complete Binary Tree
2. **Heap Property**:

   MAX-HEAP: **Key(Parent) ≥ Key(Child)**

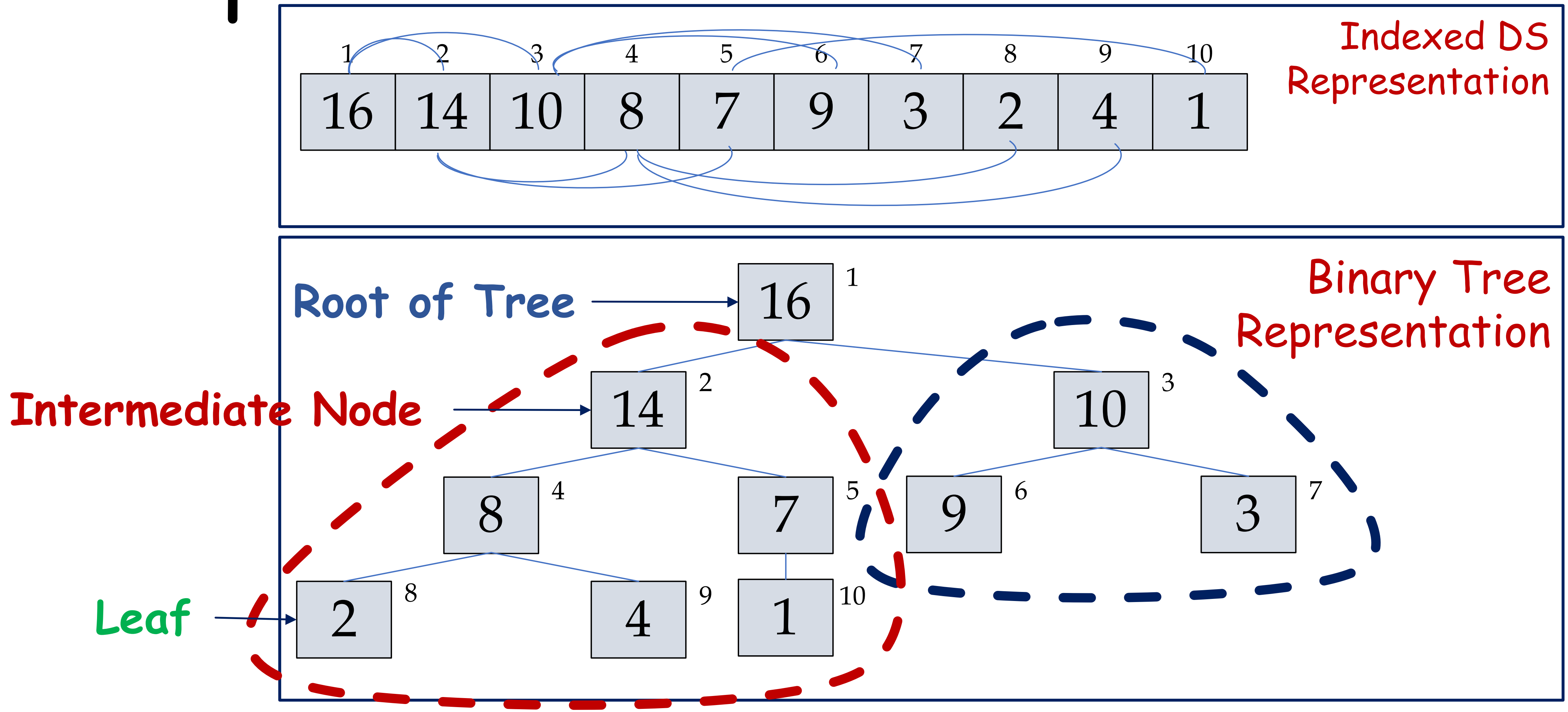
   MIN-HEAP: **Key(Parent) ≤ Key(Child)**

Heap

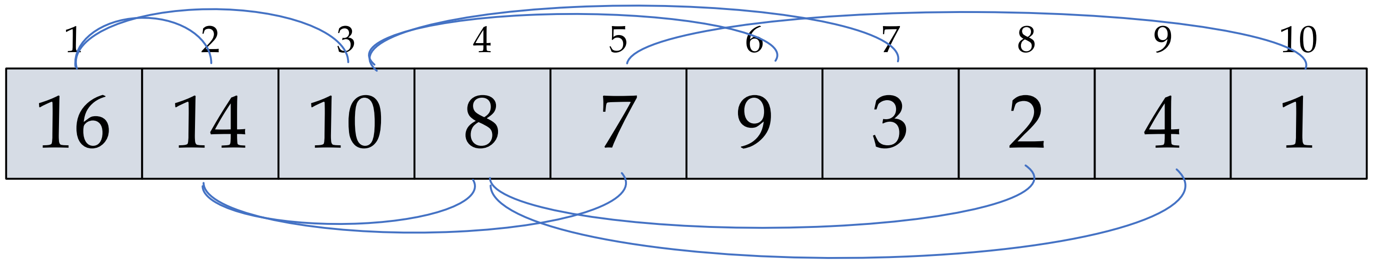
Heap Properties

- Exactly one essentially complete binary tree with n nodes with height is equal to $\lfloor \log_2 n \rfloor$
- The root of a heap always contain the largest (**MAX-HEAP**) or smallest (**MIN-HEAP**) element.
- Every sub-tree in a heap is also a Heap
- Can be implemented as an indexed DS

Heap

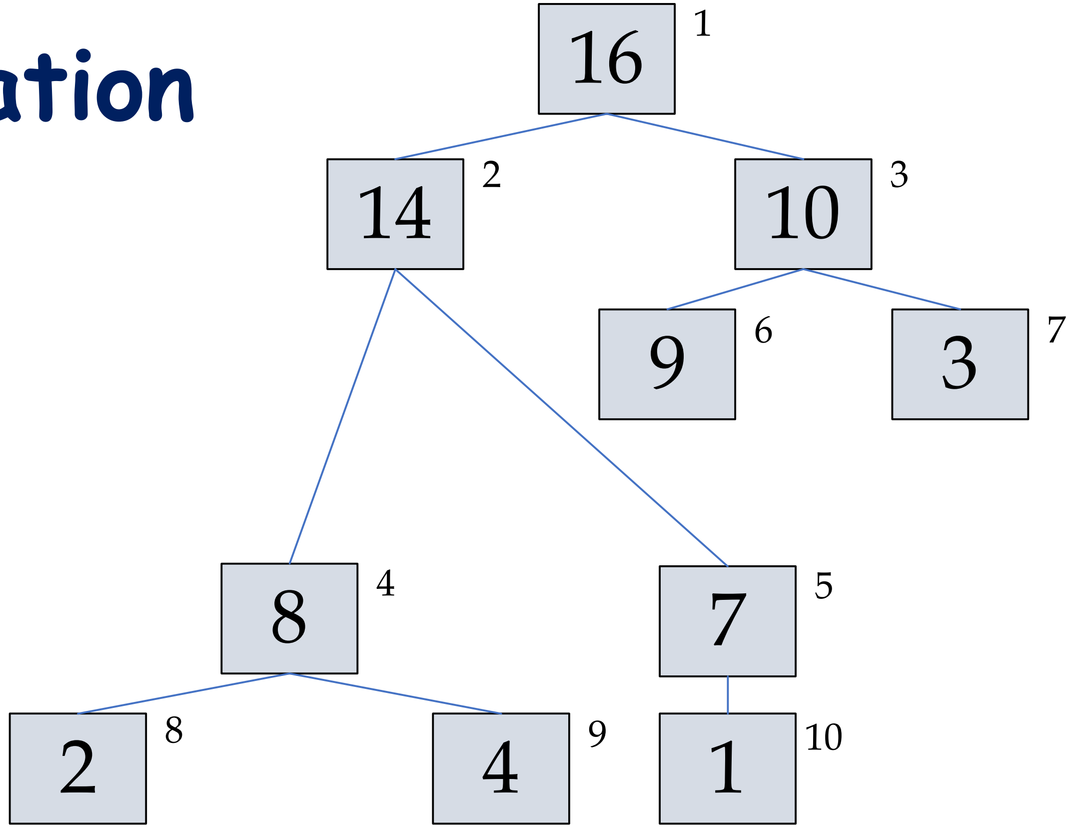

1 2 3 4 5 6 7 8 9 10
16 14 10 8 7 9 3 2 4 1
Indexed DS Representation
Root of Tree
16 1
14 2
10 3
8 4
7 5
9 6
3 7
2 8
4 9
1 10
Binary Tree Representation
Intermediate Node
Leaf

Heap



Indexed DS Representation

- Root of tree is A[1]
- Parent(i)=$\lfloor i/2 \rfloor$
- LeftChild(i)=2×i
- RightChild(i)=2×i+1



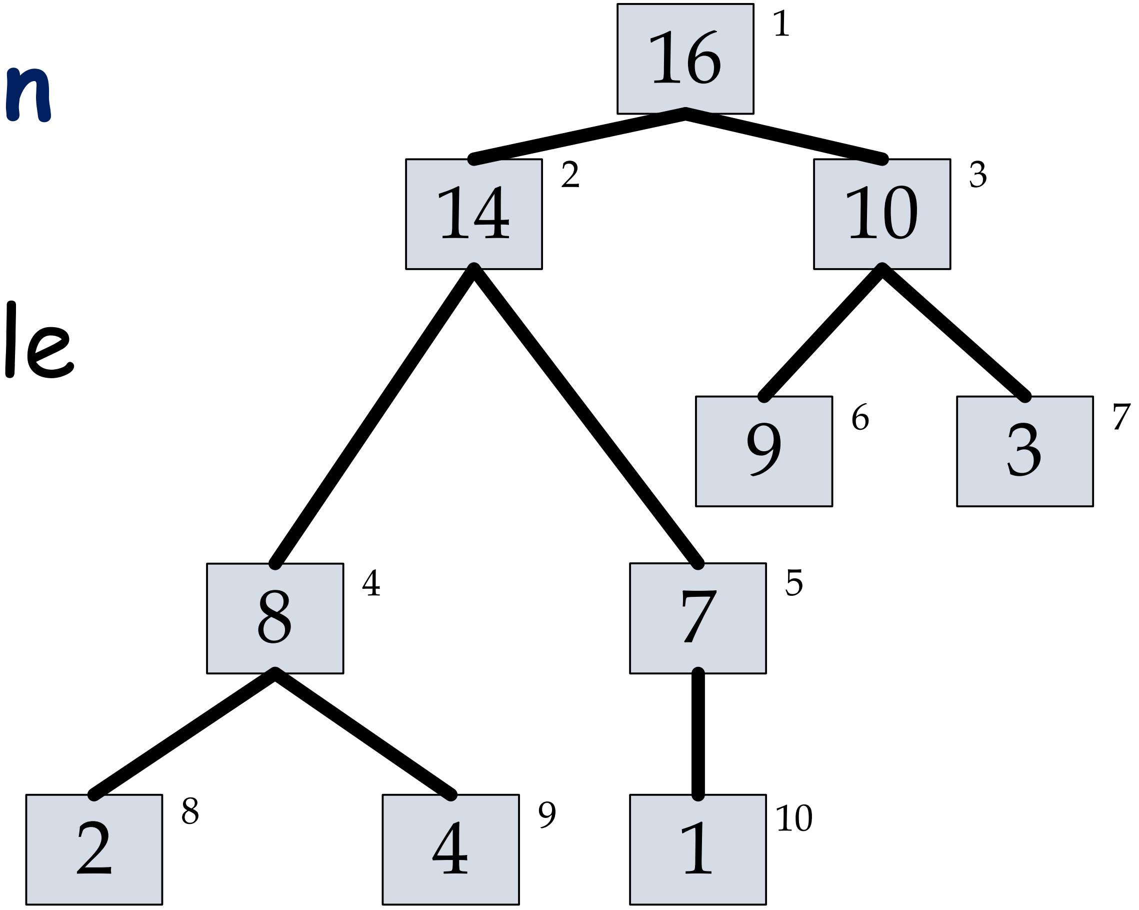
Computing is fast with an Indexed DS Binary Tree implementation

Heap

Binary Tree Representation

- **Height of Node**: No. of edges on the longest simple downward path from the node to a leaf
- **Heap Height**: Height of root

$$Heap\ Height = \lfloor \log_2 n \rfloor$$



Maintaining Heap Property

- **MAX-HEAPIFY** – an algorithm to maintain Heap Property
- Suppose A[i] may be smaller than its children
- Assumes the left and right sub-trees of the node **i** are **MAX-HEAPs**
- After **MAX-HEAPIFY**, sub-tree at the index i will become a **MAX-HEAP**

Maintaining Heap Property

```
Max-Heapify (list, i, listSize)
    L = Left-Child(i)
    R = Right-Child(i)
    If L ≤ listSize and list[L] > list[i] then
        Largest = L
    Else
        Largest = i
    If R ≤ listSize and list[R] > list[Largest] then
        Largest = R
    If Largest ≠ i then
        Swap list[i] ↔ list[Largest]
        MAX-Heapify(list, Largest, listSize)
```

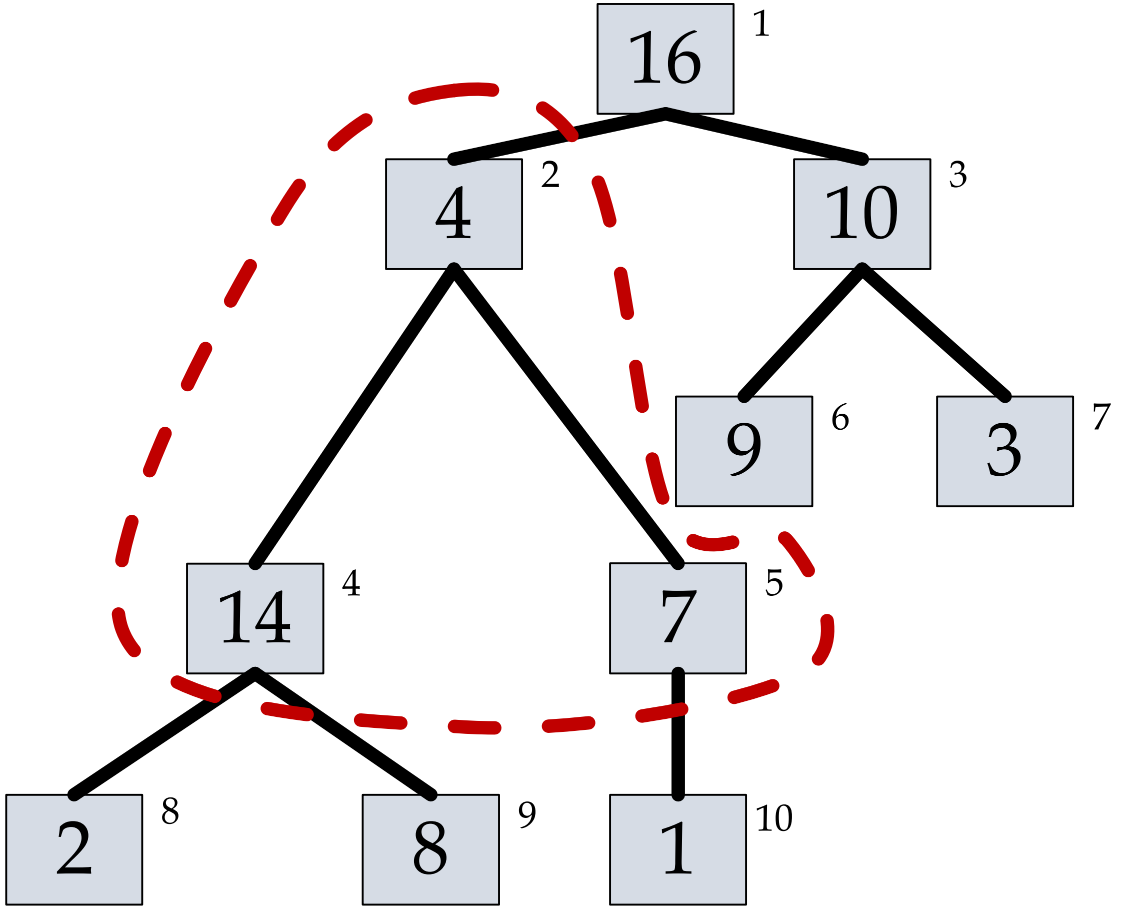
Maintaining Heap Property

MAX-HEAPIFY (list, 2, 10)

- i = 2
- listSize = 10
- L = 4
- R = 5
- Largest = 4

Swap list[i] ↔ list[Largest]

Recursive call: Max-Heapify(list, 4, 10)



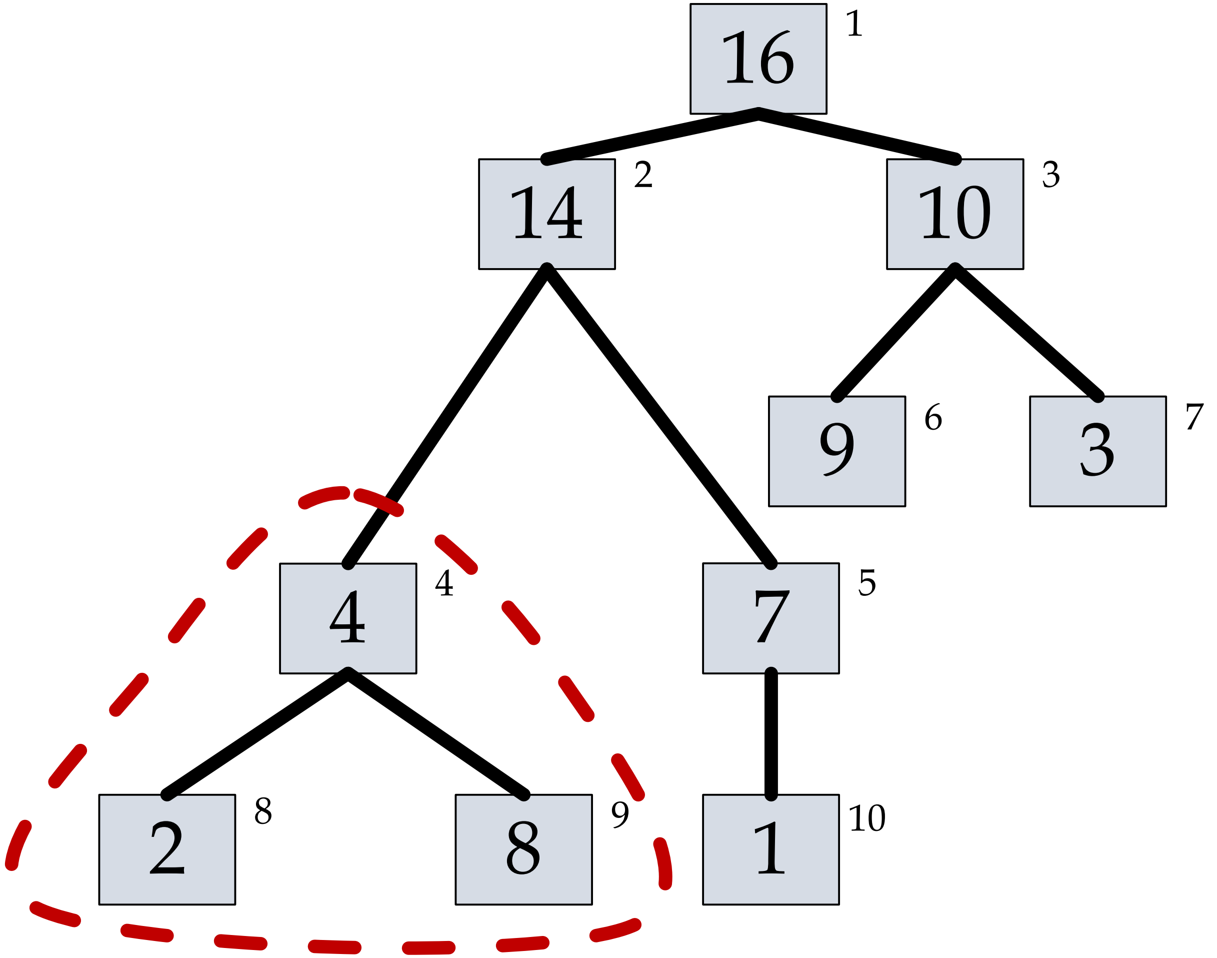
Maintaining Heap Property

MAX-HEAPIFY (list, 4, 10)

- i = 4
- listSize = 10
- L = 8
- R = 9
- Largest = 9

Swap list[i] ↔ list[Largest]

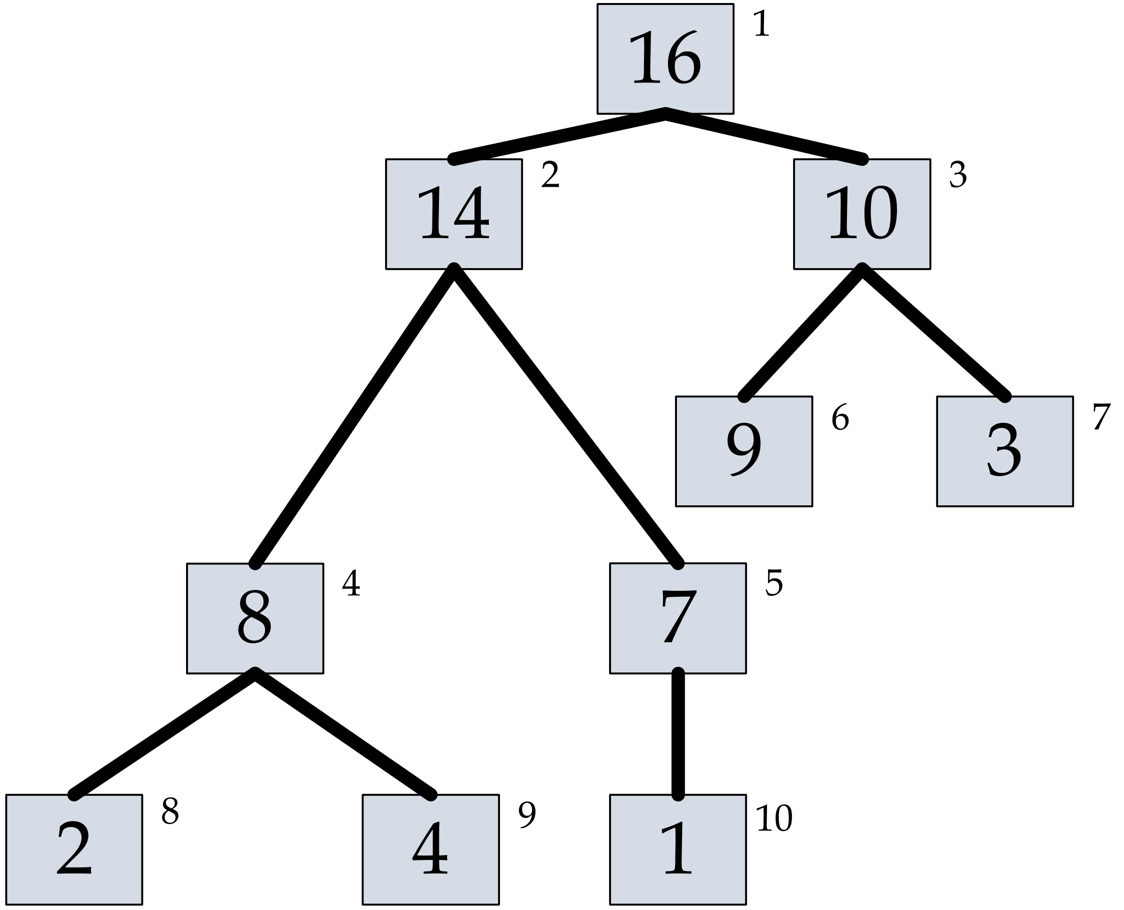
Recursive call: Max-Heapify(list, 9, 10)



Maintaining Heap Property

MAX-HEAPIFY (list, 9, 10)

- i = 9
- listSize = 10
- L = 18
- R = 19
- Largest = 9



Analysis of Max-Heapify

Constant Time – $\Theta(1)$

Max-Heapify (list, i, listSize)	
L = Left-Child(i)	c_1
R = Right-Child(i)	c_2
If L ≤ listSize and list[L] > list[i] **then**	c_3
Largest = L	c_4
Else	
Largest = i	c_5
If R ≤ listSize and list[R] > list[Largest] **then**	c_6
Largest = R	c_7
If Largest ≠ i then	c_8
Swap list[i] ↔ list[Largest]	c_9
MAX-Heapify(list, Largest, listSize)	c_{10}

Analysis of Max-Heapify

- Max-Heapify – Constant time $O(1)$ to fix the relation
- Complexity lies in the total number of recursive calls.
- **Worst-Case Analysis**: When the last row of the tree is exactly half-full.
- $T(n) = O(\log_2 n)$

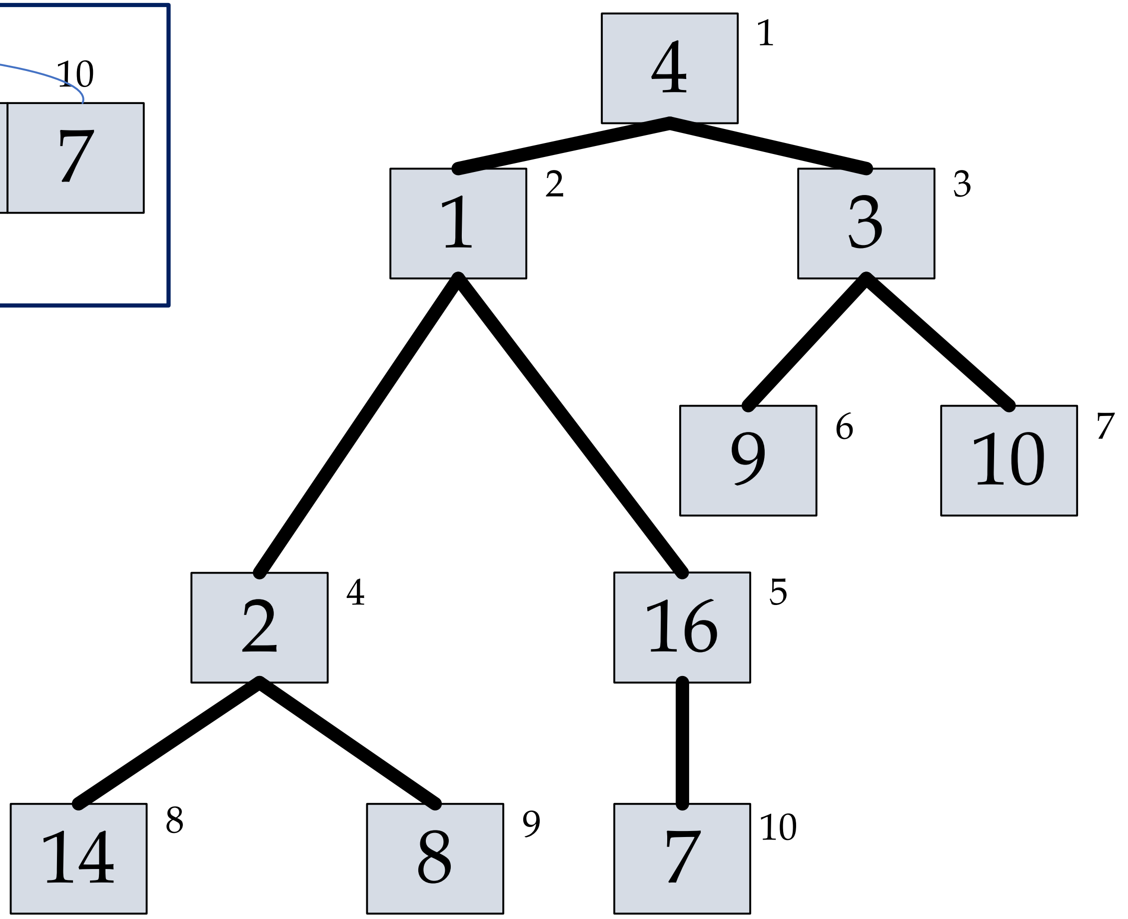
Building Heap

- Building from a list of numbers
- **Input**: an ordinary list of numbers
- **Output**: a heap containing all the numbers

```
Build-Max-Heap (list, n)
1  For i = ⌊n/2⌋ to 1
2        Max-Heapify(A, i, n)
```

Building Heap - Example

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Build-Max-Heap (list, n)
1 For i = $\lfloor n/2 \rfloor$ to 1
2 Max-Heapify(A, i, n)

Analysis of Building Heap

- Cost of Max-Heapify $O(\log_2 n)$
- Total calls to Max-Heapify $O(n)$
- Thus $O(n \log_2 n)$ is an upper bound to build a heap from a list of n numbers.

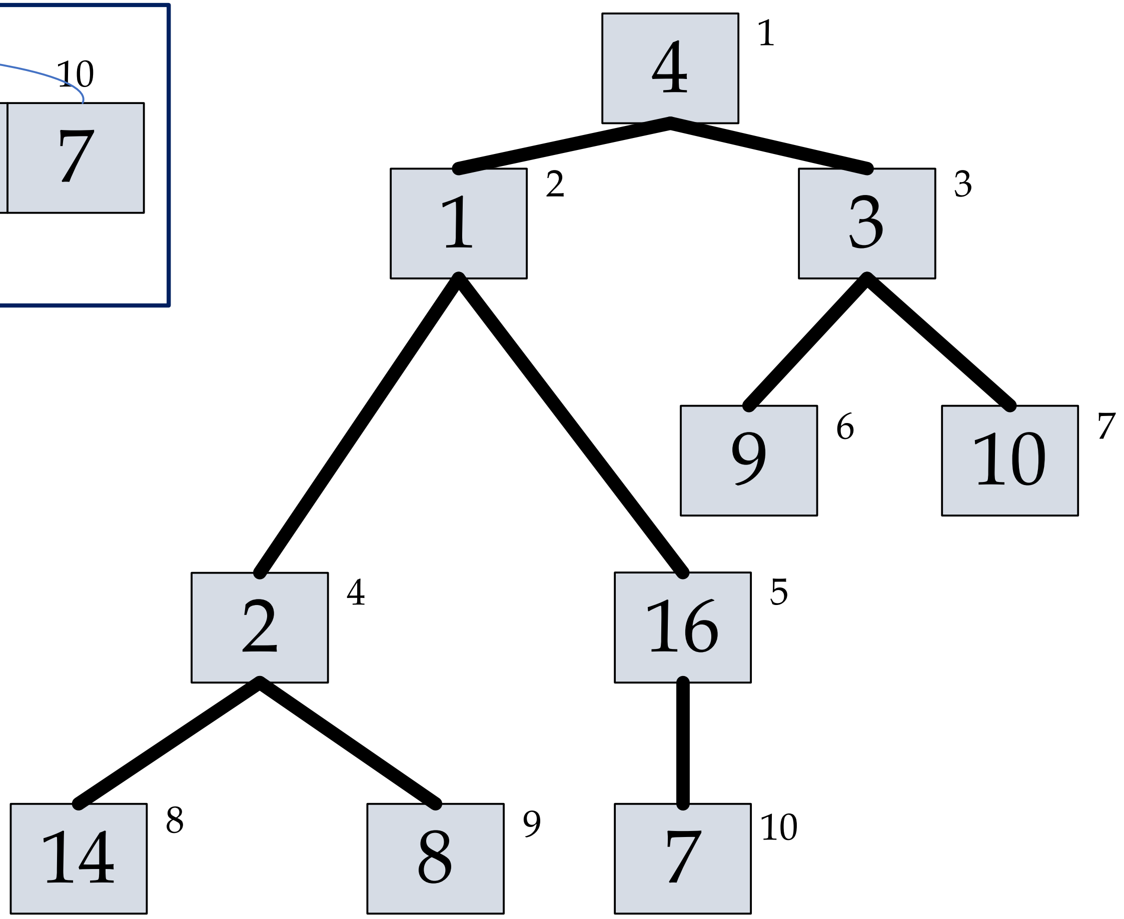
Heap Sort

```
Heap-Sort (A, n)
    Build-Max-Heap(A, n)
    For i = n to 2
        Exchange A[1] ↔ A[i]
        Max-Heapify(A, 1, i – 1)
```

Heap Sort

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

```
Heap-Sort (A, n)
1   Build-Max-Heap(A, n)
2   For i = n to 2
3         Exchange A[1] ↔ A[i]
4         Max-Heapify(A, 1, i – 1)
```

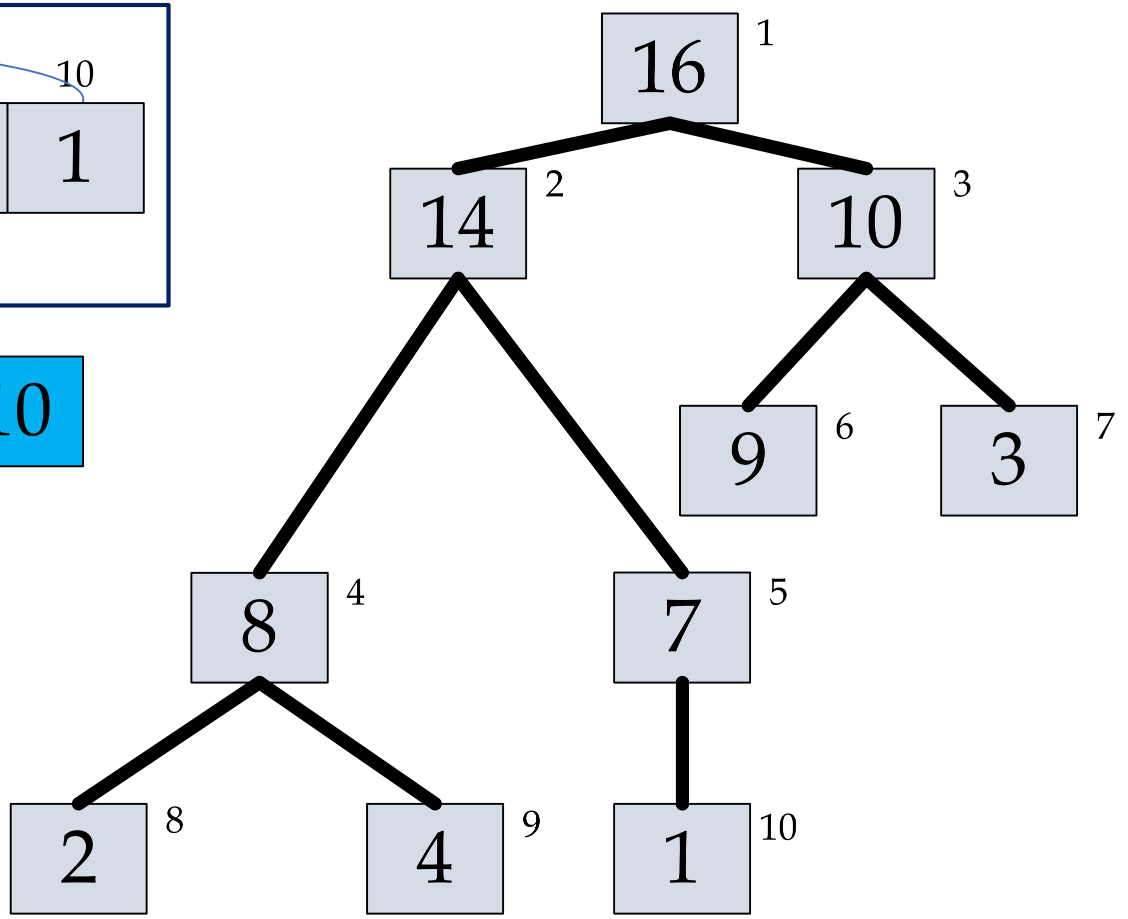


Heap Sort

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

```
Heap-Sort (A, n)
Build-Max-Heap(A, n)
For i = n to 2
        Exchange A[1] ↔ A[i]
        Max-Heapify(A, 1, i – 1)
```

i 10

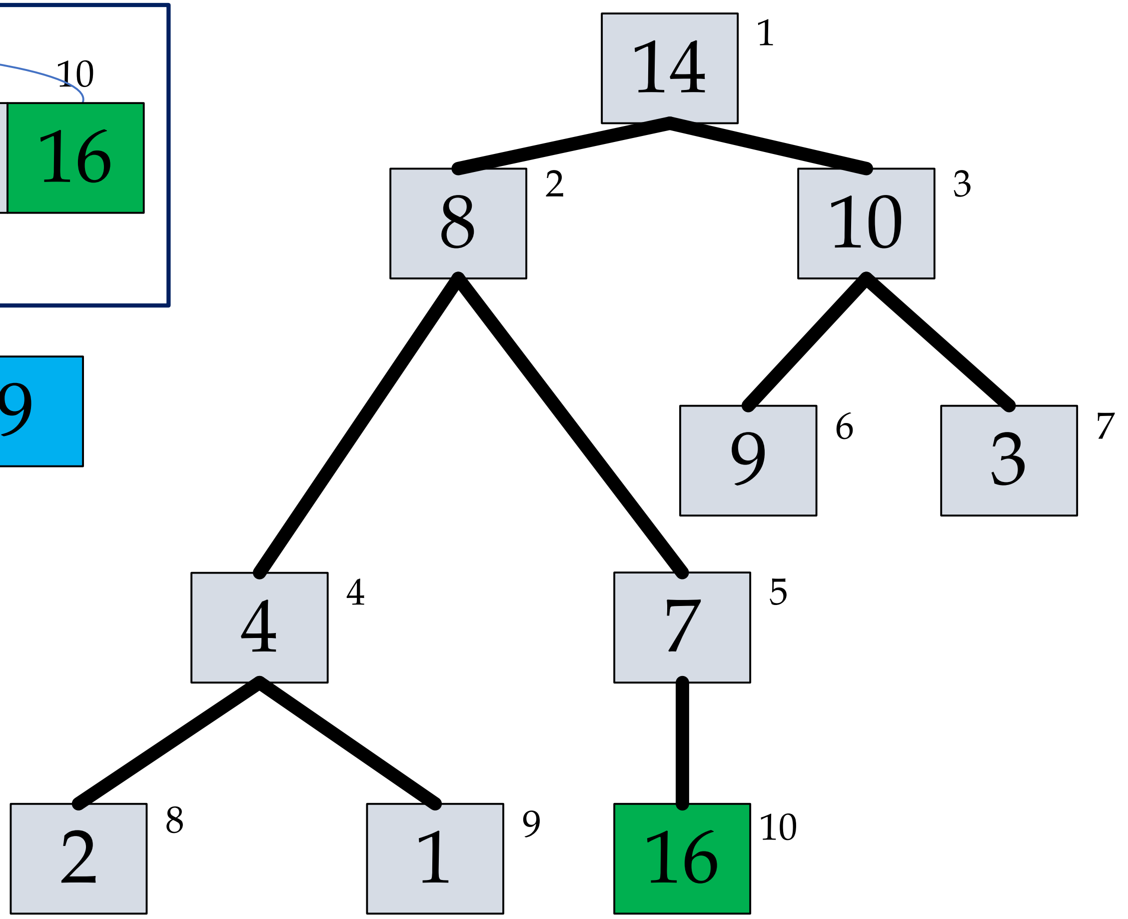


Heap Sort

1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

```
Heap-Sort (A, n)
    Build-Max-Heap(A, n)
    For i = n to 2
        Exchange A[1] ↔ A[i]
        Max-Heapify(A, 1, i – 1)
```

i 9

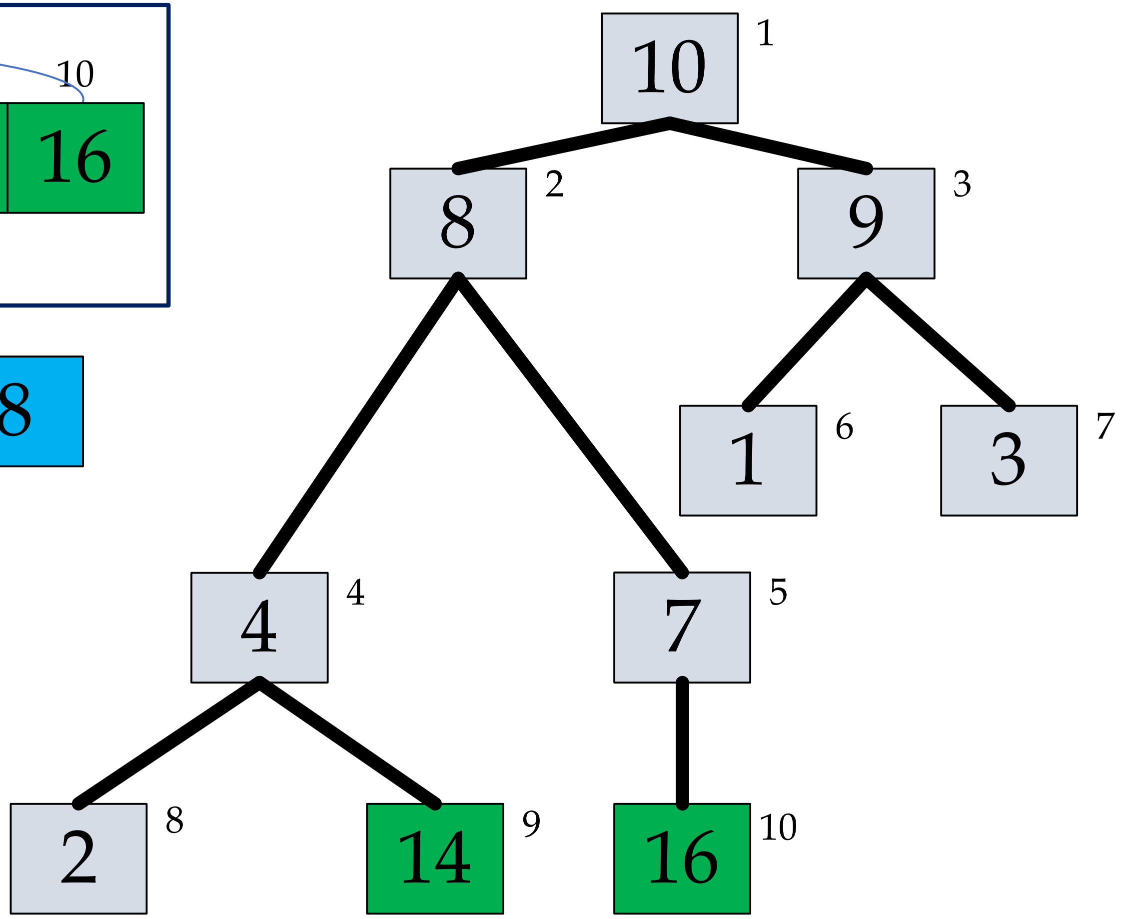


Heap Sort

1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

```
Heap-Sort (A, n)
Build-Max-Heap(A, n)
For i = n to 2
        Exchange A[1] ↔ A[i]
        Max-Heapify(A, 1, i – 1)
```

i 8



Heap Sort - Analysis

Heap-Sort (A, n)
1 Build-Max-Heap(A, n) $O(n \log_2 n)$
2 For i = n to 2 $O(1) \rightarrow n - 1$
3 Exchange A[1] ↔ A[i] $O(1) \rightarrow n - 1$
4 Max-Heapify(A, 1, i - 1) $O(\log_2 n) \rightarrow n - 1$

$$T(n) = O(n \log_2 n) + (n-1)c_1 + (n-1)\log_2 n$$
$$T(n) = O(n \log_2 n) ----- -Complexity\ of\ Heap\ Sort$$

Priority Queue

- A data structure for maintaining a set of elements such that the next element in the queue is with the maximum/minimum priority.
  - Hospital Emergency
  - Printer Job Scheduling
  - Processer Job Scheduling

Priority Queue

Operations

- Make a priority queue
- Insert an element with given priority
- Return Max/Min element
- Return and delete Max/Min element
- Increase/Decrease the priority of an element
- Union of two priority queues
- Delete queue

Priority Queue

Implementation

- Unsorted Array
- Sorted Array
- Heaps

Comparison

Operations	Unsorted	Sorted
Build	$\Theta(1)$	$O(n\log_2 n)^*$
Insert	$\Theta(1)$	$O(n)^*$
Find Max/Min	$O(n)^*$	$\Theta(1)$
Delete Max/Min	$O(n)^*$	$\Theta(1)$
Union	$\Theta(1)$	$O(n)^*$
Increase Priority	$\Theta(1)$	$\Theta(1)$
Is-empty	$\Theta(1)$	$\Theta(1)$

* Complexity depends on the algorithm used

Priority Queue

Comparison

Operations	Unsorted	Sorted		Binary Heaps
Build	$\Theta(1)$	$O(n\lg n)^*$		$\Theta(n)$
Insert	$\Theta(1)$	$O(n)^*$		$\Theta(\lg n)$
Find Max/Min	$O(n)^*$	$\Theta(1)$		$\Theta(1)$
Delete Max/Min	$O(n)^*$	$\Theta(1)$		$\Theta(\lg n)$
Union	$\Theta(1)$	$O(n)^*$		$\Theta(n)$
Increase Priority	$\Theta(1)$	$\Theta(1)$		$\Theta(\lg n)$
Is-empty	$\Theta(1)$	$\Theta(1)$		$\Theta(1)$

* Complexity depends on the algorithm used

Priority Queue

Find Max/Min

```
Heap-Maximum (A)
1  Return A[1]
```

What is the worst case running time?

$$\theta(1)$$

Priority Queue

Extract (Find and Delete) Max/Min

```
Heap-Extract-Max (A, n)
    If n > 0 then
        Max = A[1]
        A[1] = A[n]
        n = n-1
        Max-Heapify (A,1,n)
        Return Max
    Else
        Return "Error: Heap underflow!"
```

Replacing root with last Element

Reduce the size of Array by 1

Maintaining Heap property

What is the worst case running time?

$O(lg\, n)$

Priority Queue

Increase Priority

```
Heap-Increase-Key (A, i, Key)
    If Key > A[i] then
        A[i] = Key
        While i > 1 and A[Parent(i)] < A[i]
            Exchange A[i] ↔ A[Parent(i)]
            i = Parent(i)
    Else
        "Error: New key is smaller than
        current key"
```

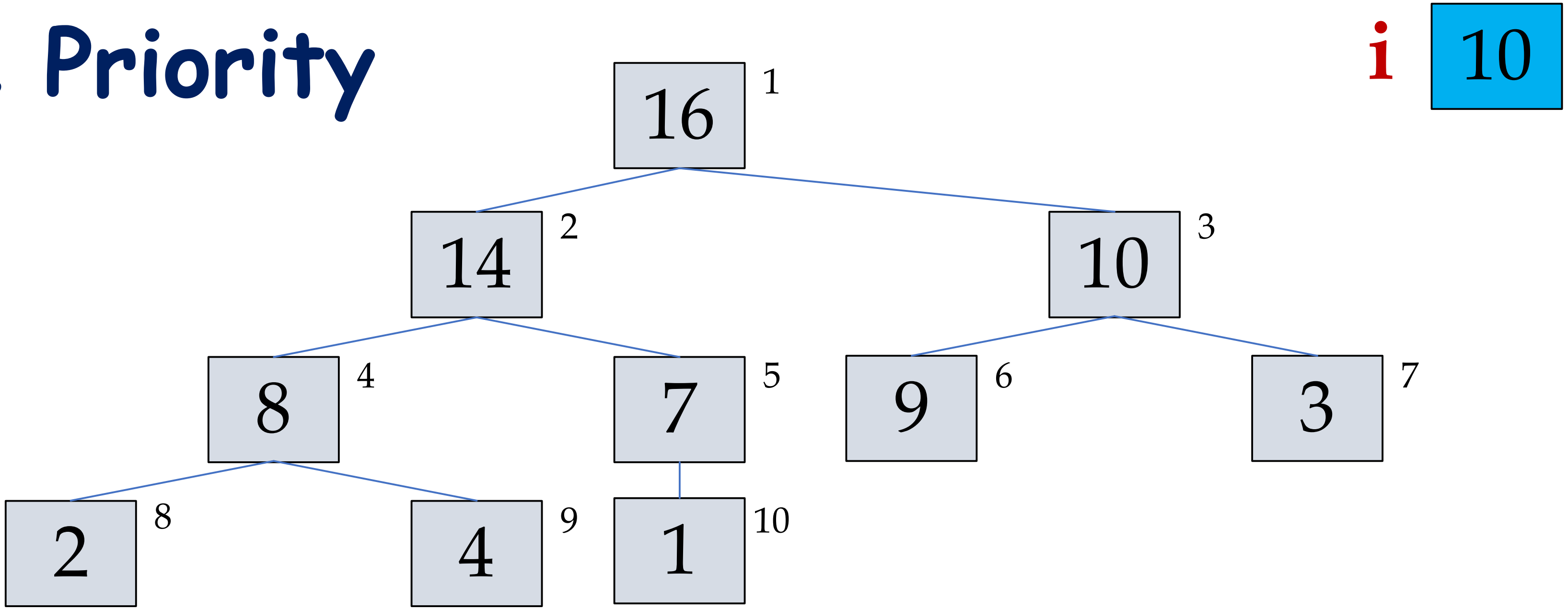
Maintaining Heap property

What is the worst case running time?

$$O(lg\, n)$$

Priority Queue

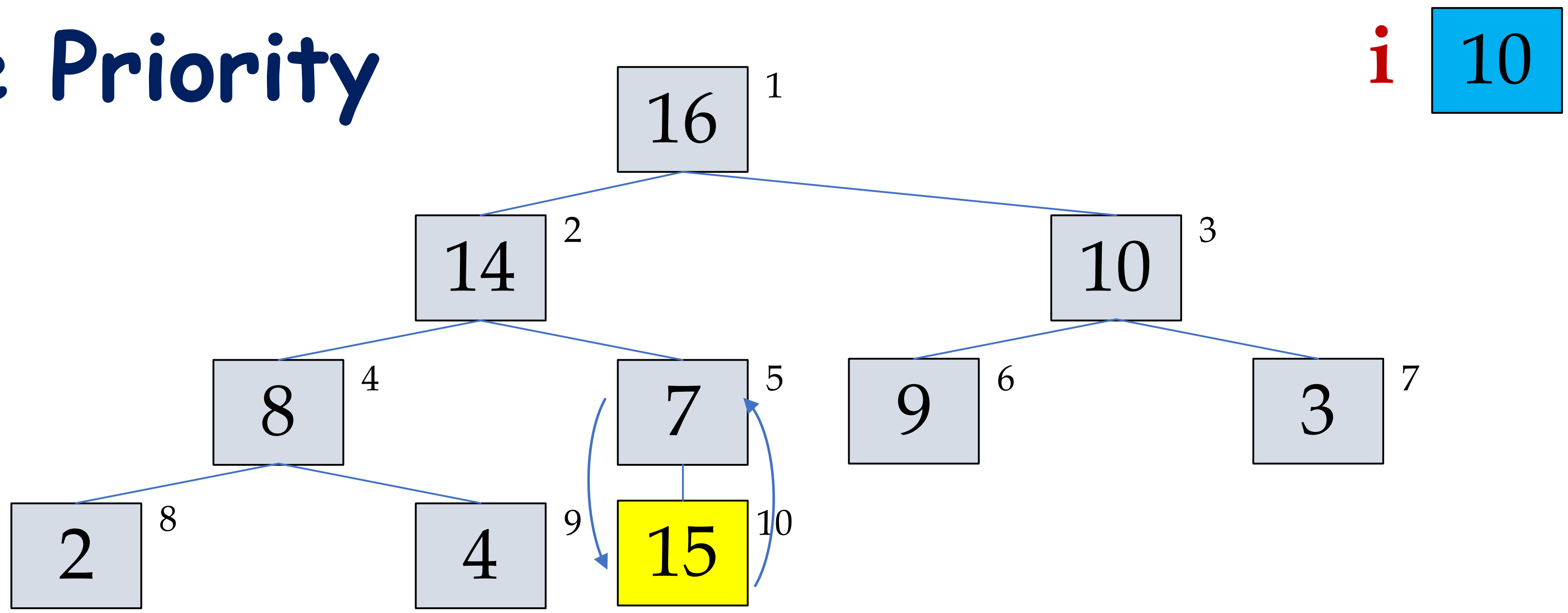
Increase Priority



Heap-Increase-Key(A,10,15)

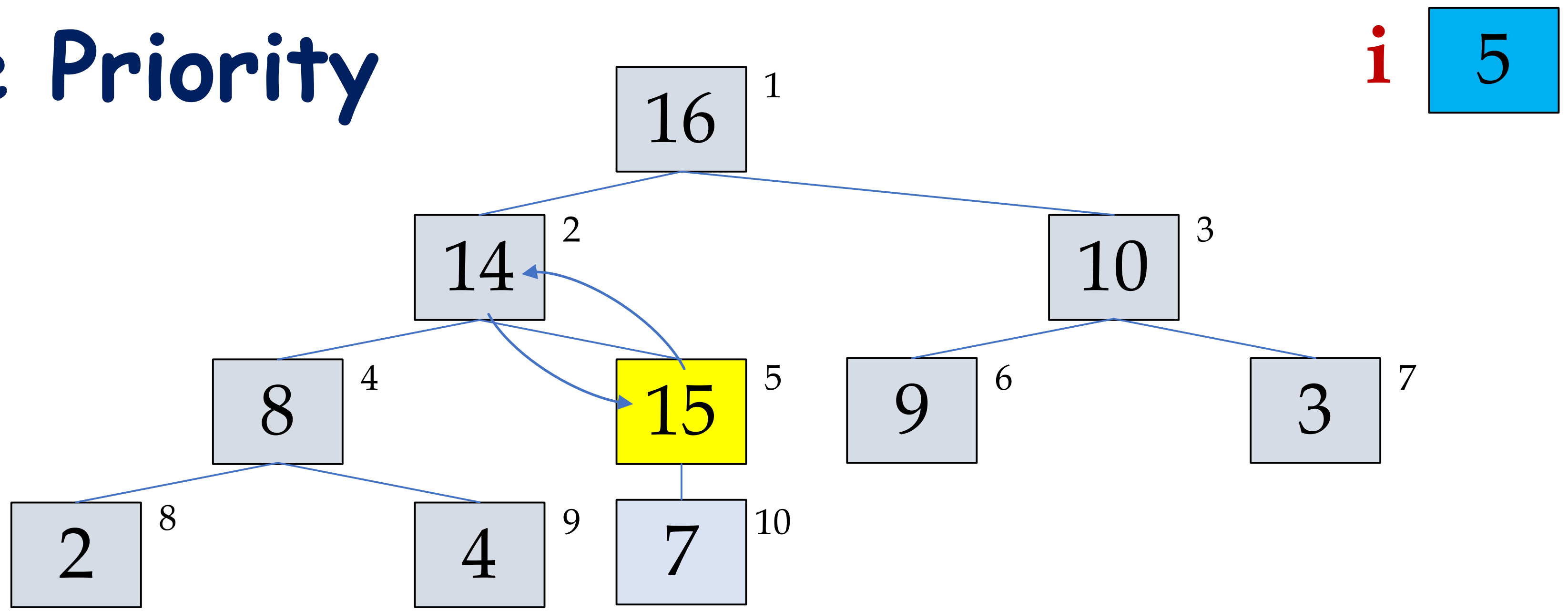
Priority Queue

Increase Priority



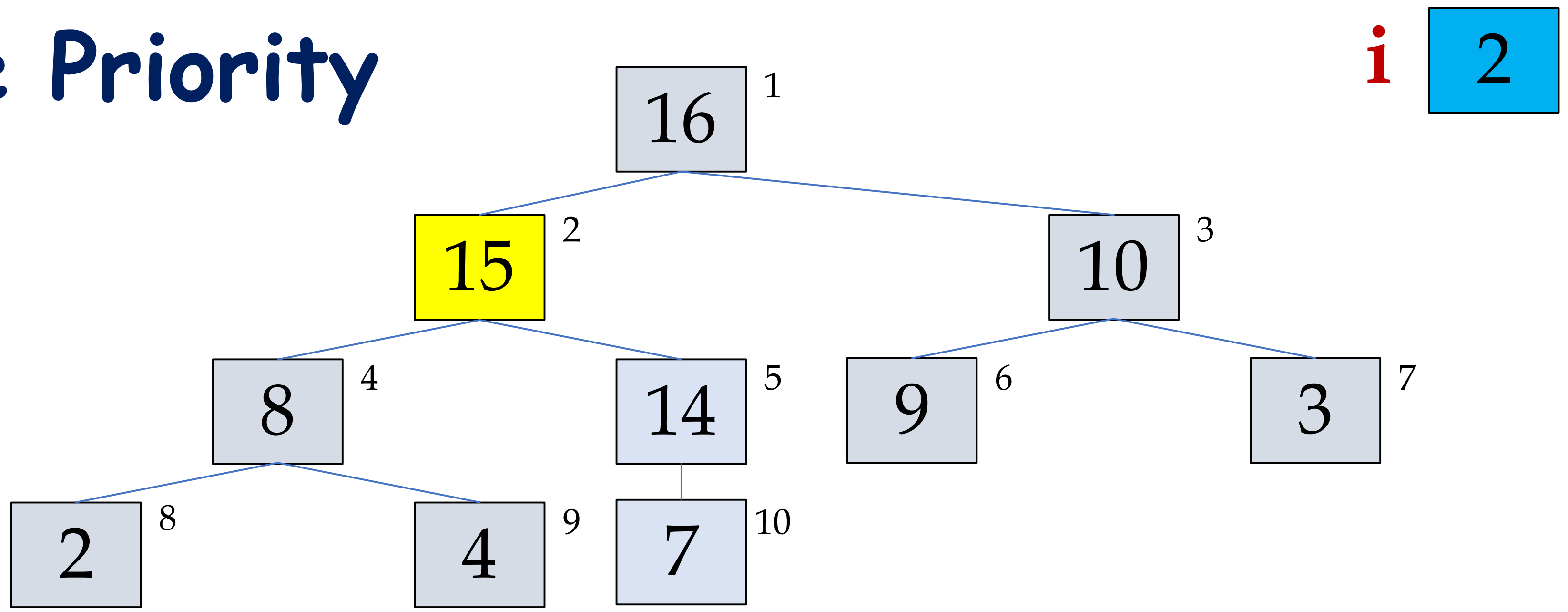
Priority Queue

Increase Priority



Priority Queue

Increase Priority



What is the worst case running time? $O(lg\, n)$

Priority Queue

Insert a new element

```
Max-Heap-Insert (A, Key, n)
1    n = n + 1
2    A[n] = -∞
3    Heap-Increase-Key(A, n, Key)
```

Increase the size of Array by 1

Initialize the value by a very small value

What is the worst case running time?

$O(lg\,n)$

Priority Queue

Union

- How to design an algorithm for union?
- What is the input of the algorithm?
- What is the output of the algorithm?
  1. Input: Two Heaps
  2. Output: one Heap that contain all the elements of two input heaps.

Priority Queue

Union

- What steps we will follow in the algorithm?
  1. Create an array of with size equal to the sum of sizes of two input heaps
  2. Copy all elements of two input heaps in the new array
  3. Build a heap from the new array

Priority Queue

Union

```
MAX-HEAPS-UNION (A1, A2, n1, n2)
    Build an Array A of size n1 + n2
    For i = 1 to n1
        A[i] = A1[i]
    For j = 1 to n2
        A[i] = A2 [j]
        i = i + 1
    Build-Max-Heap (A, n1 + n2)
```

What is the worst case running time? $\theta(n)$

Comparison Sorting Algorithms

1. All sorting algorithms so far are **Comparison sorting algorithms**
2. The best worst-case running time is $O(n \lg n)$

Algorithm	Worst Case Sorting Time
Insertion Sort	$O(n^2)$
Merge Sort	$\Theta(n\lg n)$
Quick Sort	$O(n^2)$
Heap Sort	$\Theta(n\lg n)$

Is $O(n \lg n)$ the best we can do?

Merge Sort

```
MERGE–SORT (A, left_index, right_index)
if  left_index < right_index then
        q = Floow((left_index+right_index)/2)
      MERGE–SORT(A, left_index, q)
      MERGE–SORT(A, q+1, rigth_index)
      MERGE(A, left_index, q, right_index)
```

Merge Sort

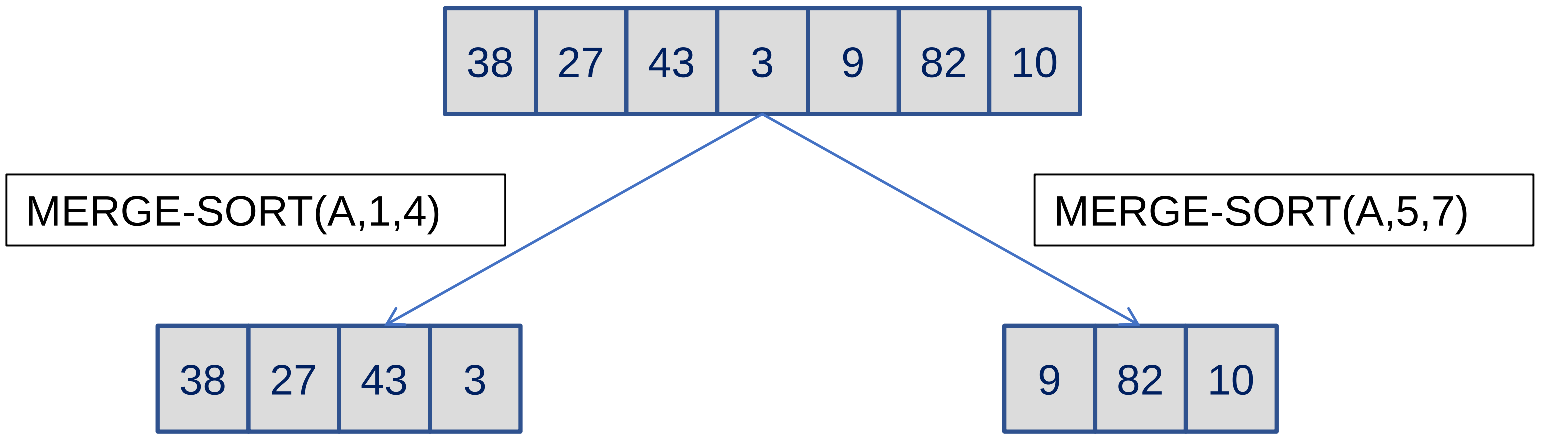
```
MERGE–SORT (A, left_index, right_index)
    if  left_index < right_index then
        q = Floow((left_index+right_index)/2)
        MERGE–SORT(A, left_index, q)
        MERGE–SORT(A, q+1, rigth_index)
        MERGE(A, left_index, q, right_index)
```

Input

38	27	43	3	9	82	10

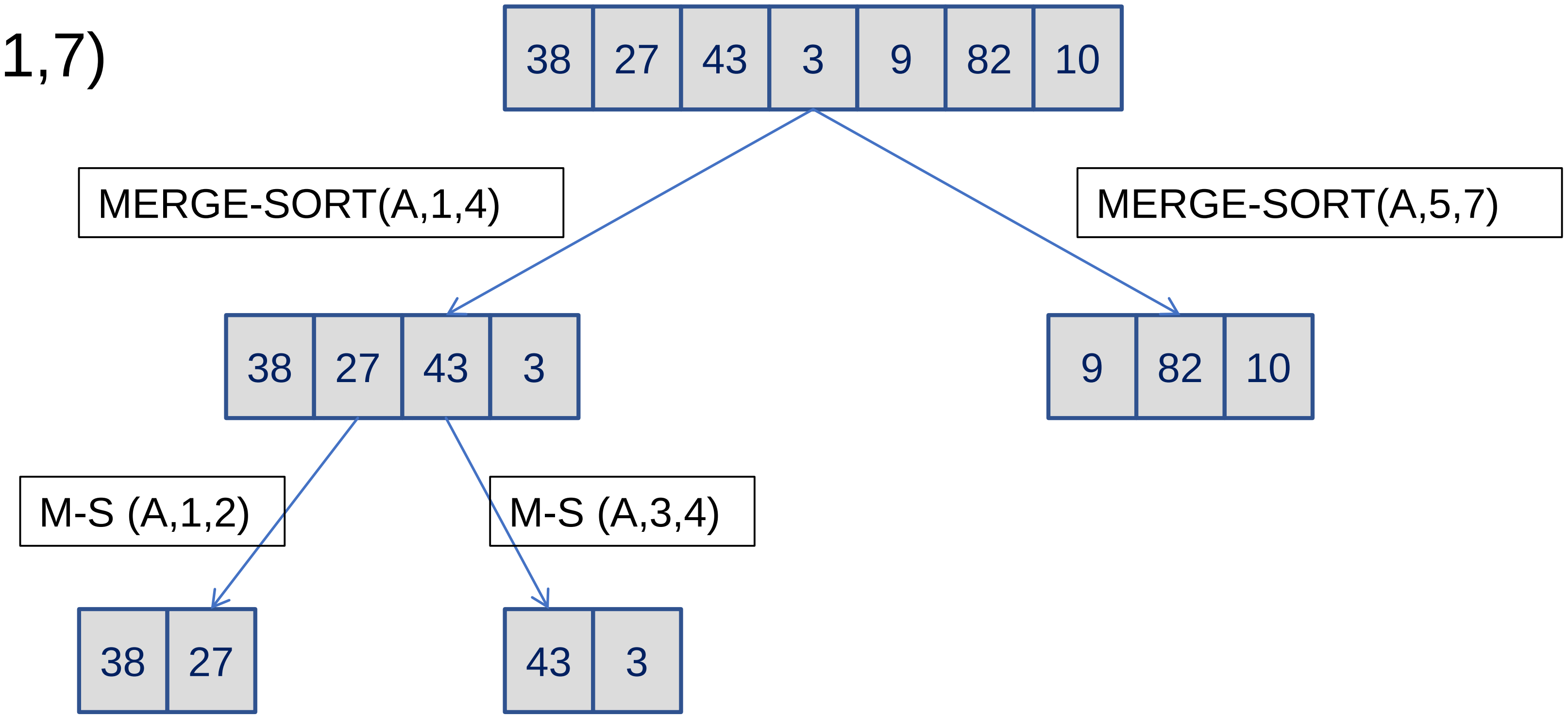
Merge Sort - Dividing into sub-problem

MERGE-SORT(A,1,7)

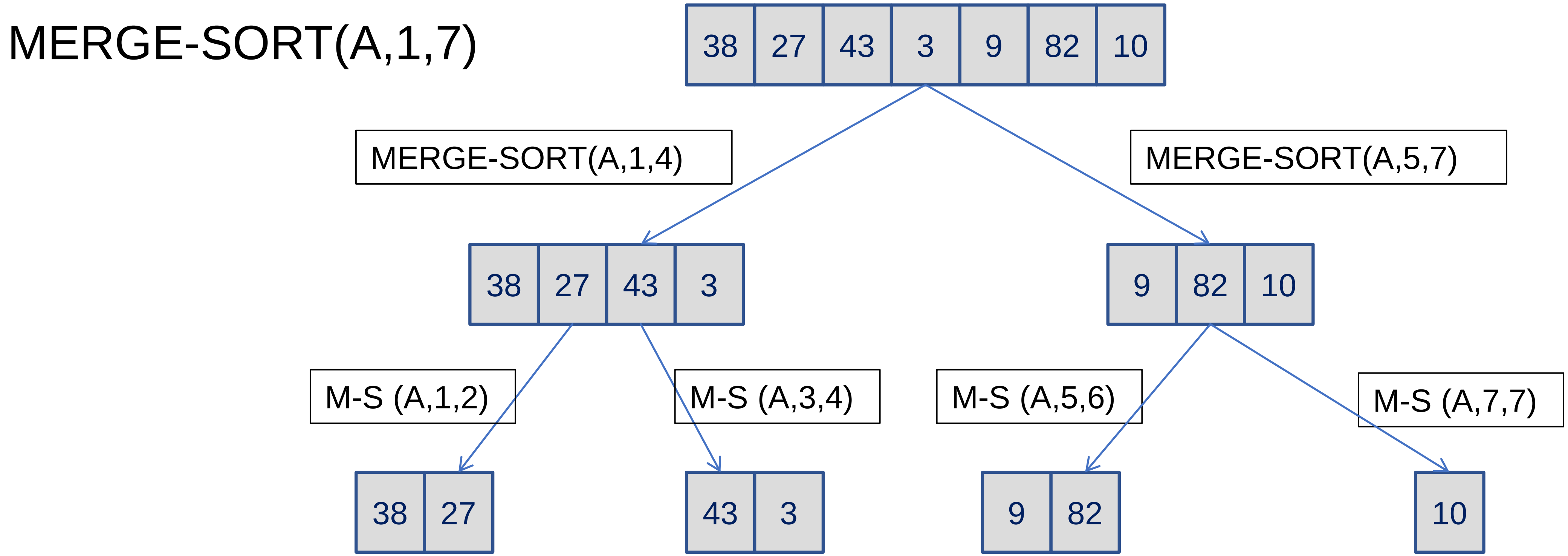


Merge Sort - Dividing into sub-problem

MERGE-SORT(A,1,7)

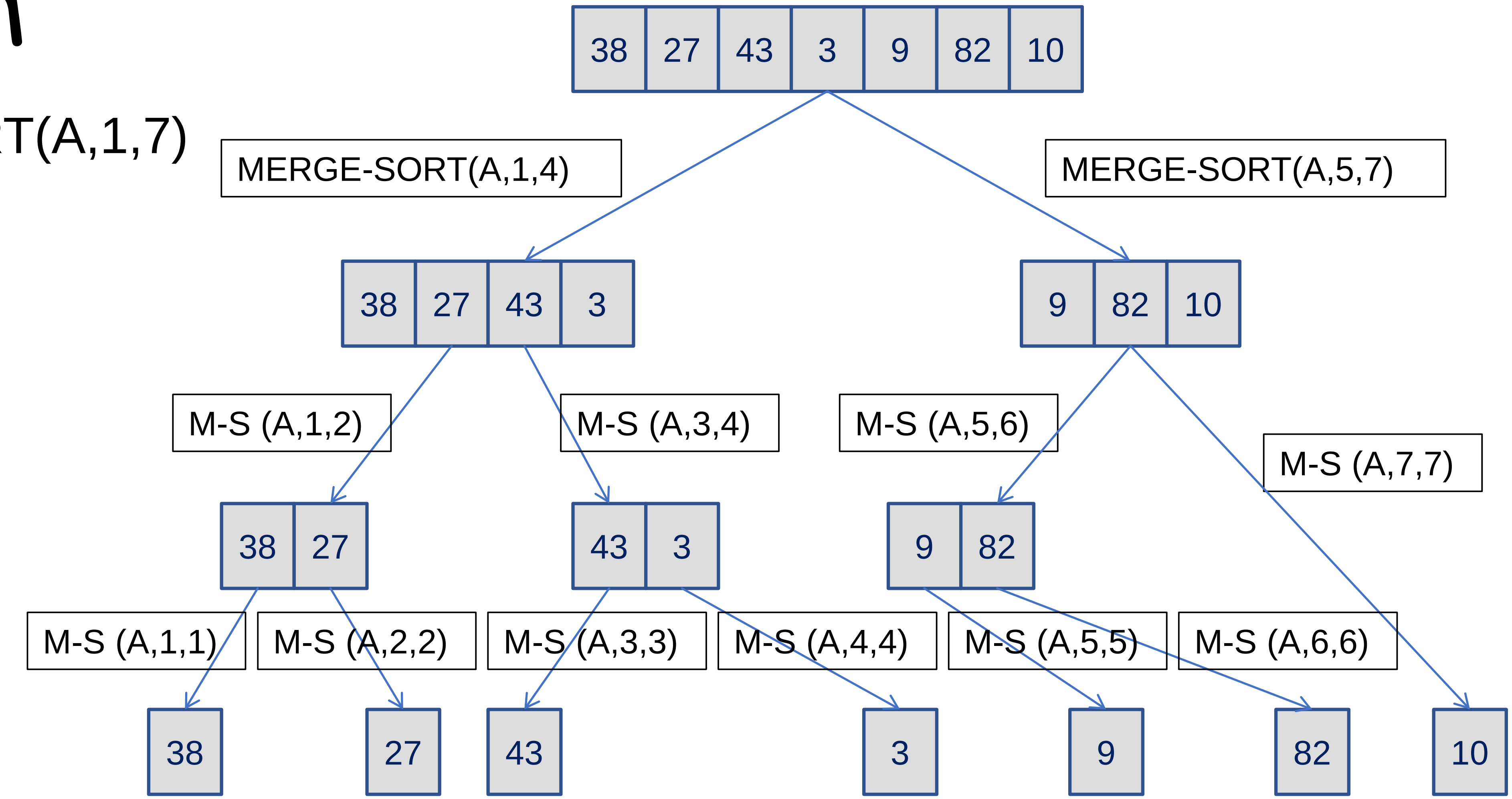


Merge Sort - Dividing into sub-problem



Merge Sort – Dividing into sub-problem

MERGE-SORT(A,1,7)



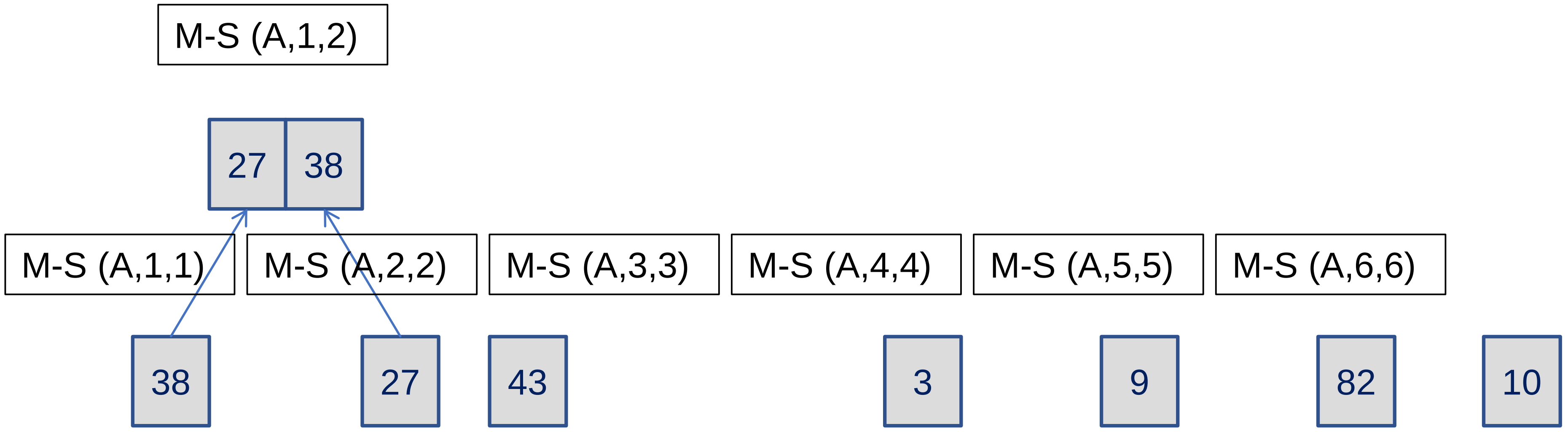
Merge Sort

```
MERGE (A, left_index, q, right_index)
{
      n1 = q – left_index + 1
      n2 = right_index – q
      create arrays L[1 … n1+1] and R[1 … n2+1]
      for i = 1 to n1
        L[i] = A[left_index + i – 1 ]
       for j = 1 to n2
        R[j] = A[q + j]
      L[n1+1] = ∞
      R[n2+1] = ∞
```

```
       i = j = 1
       for k = left_index to r
        if L[i] ≤ R[j] then
                A[k] = L[i]
                i = i + 1
        Else
                A[k] = R[j]
                j = j + 1
}
```

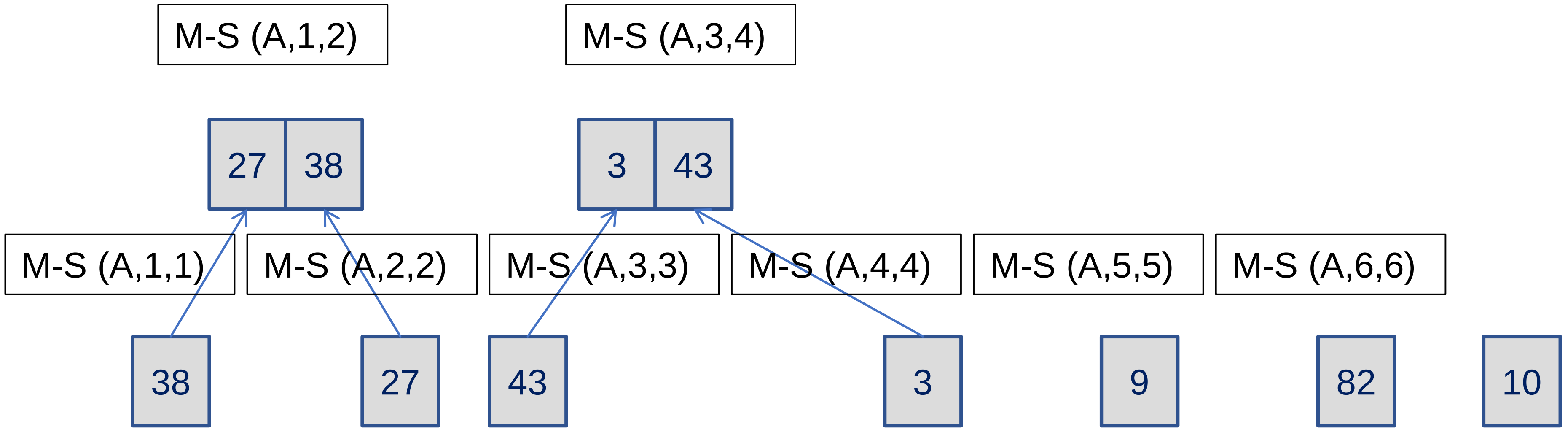
Merge Sort - Merging Sorted Arrays

MERGE-SORT(A,1,7)



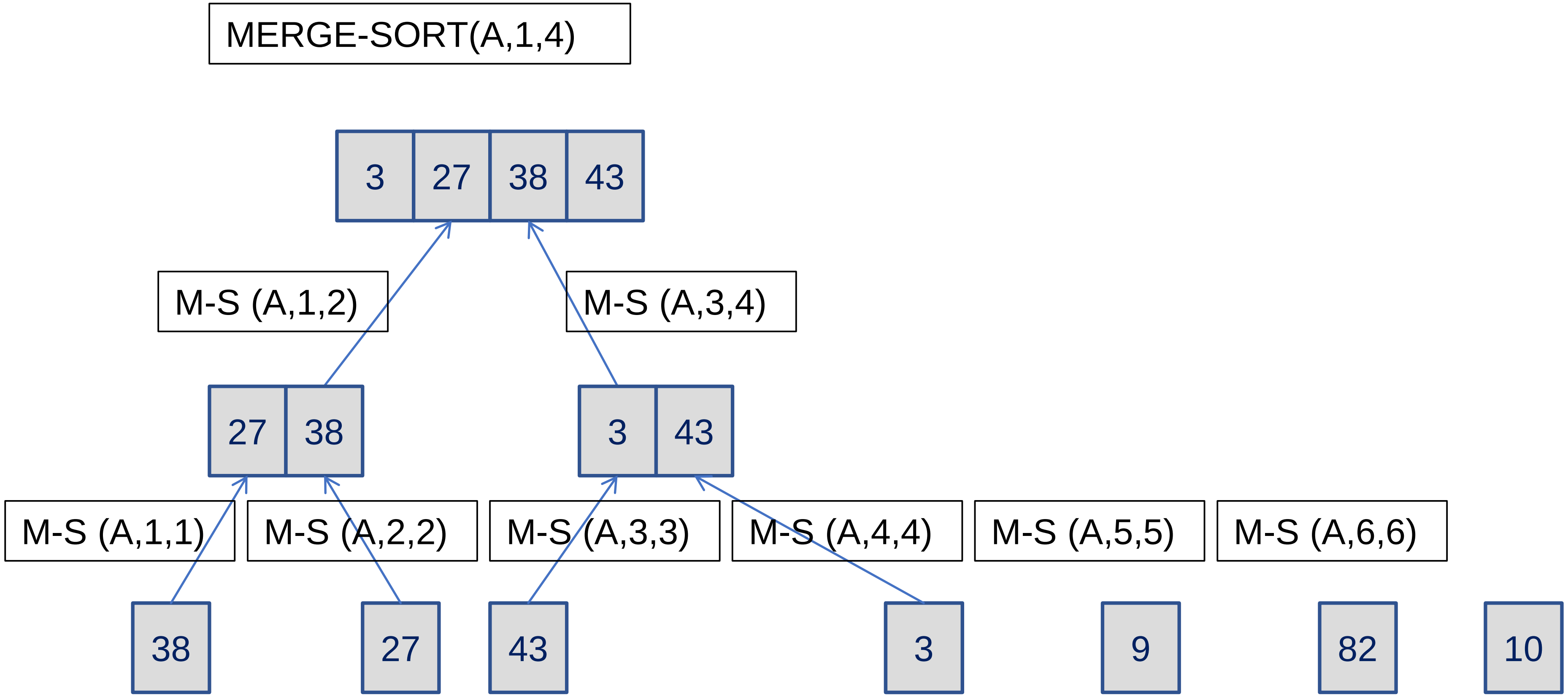
Merge Sort - Merging Sorted Arrays

MERGE-SORT(A,1,7)



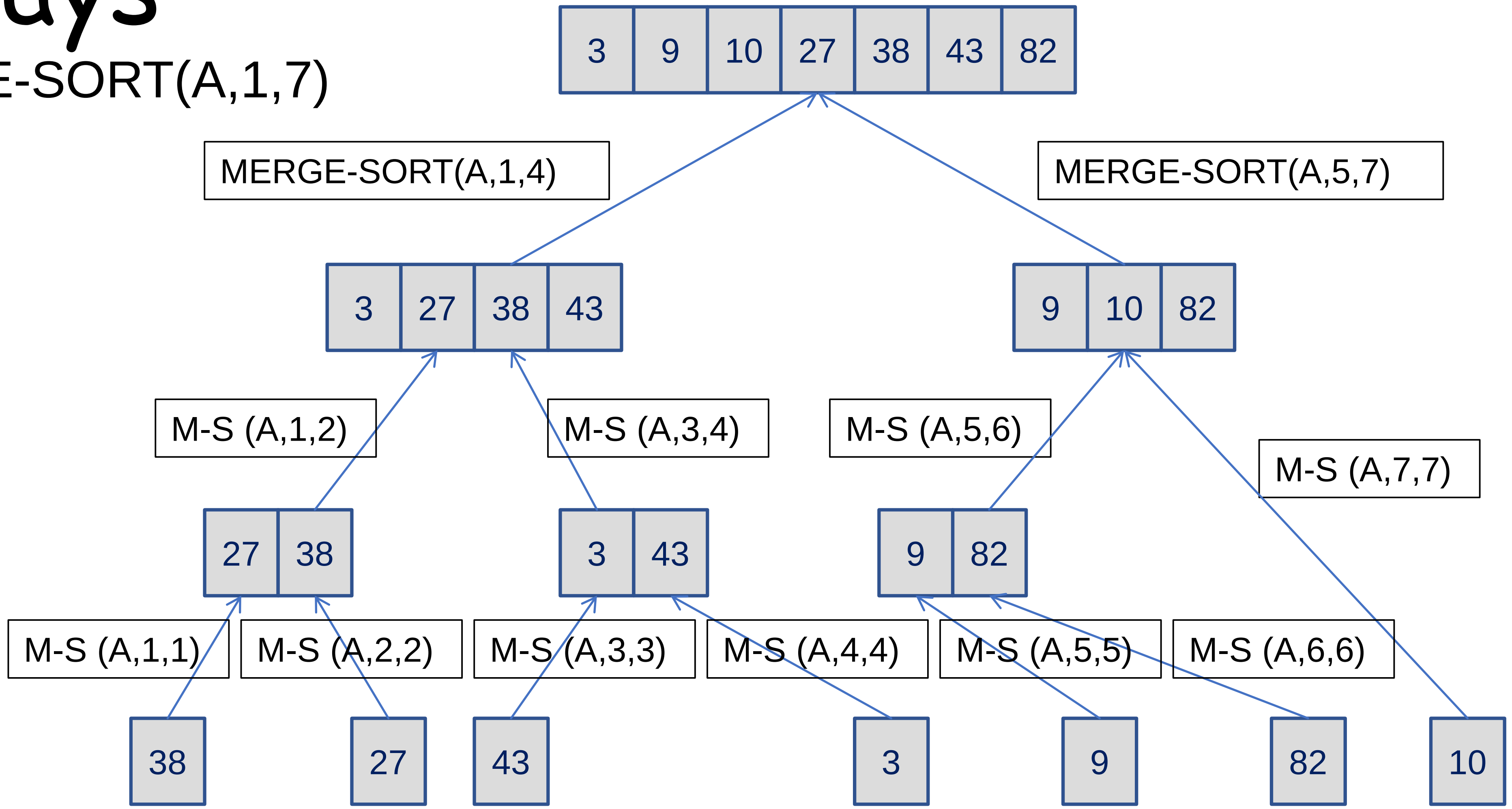
Merge Sort - Merging Sorted Arrays

MERGE-SORT(A,1,7)

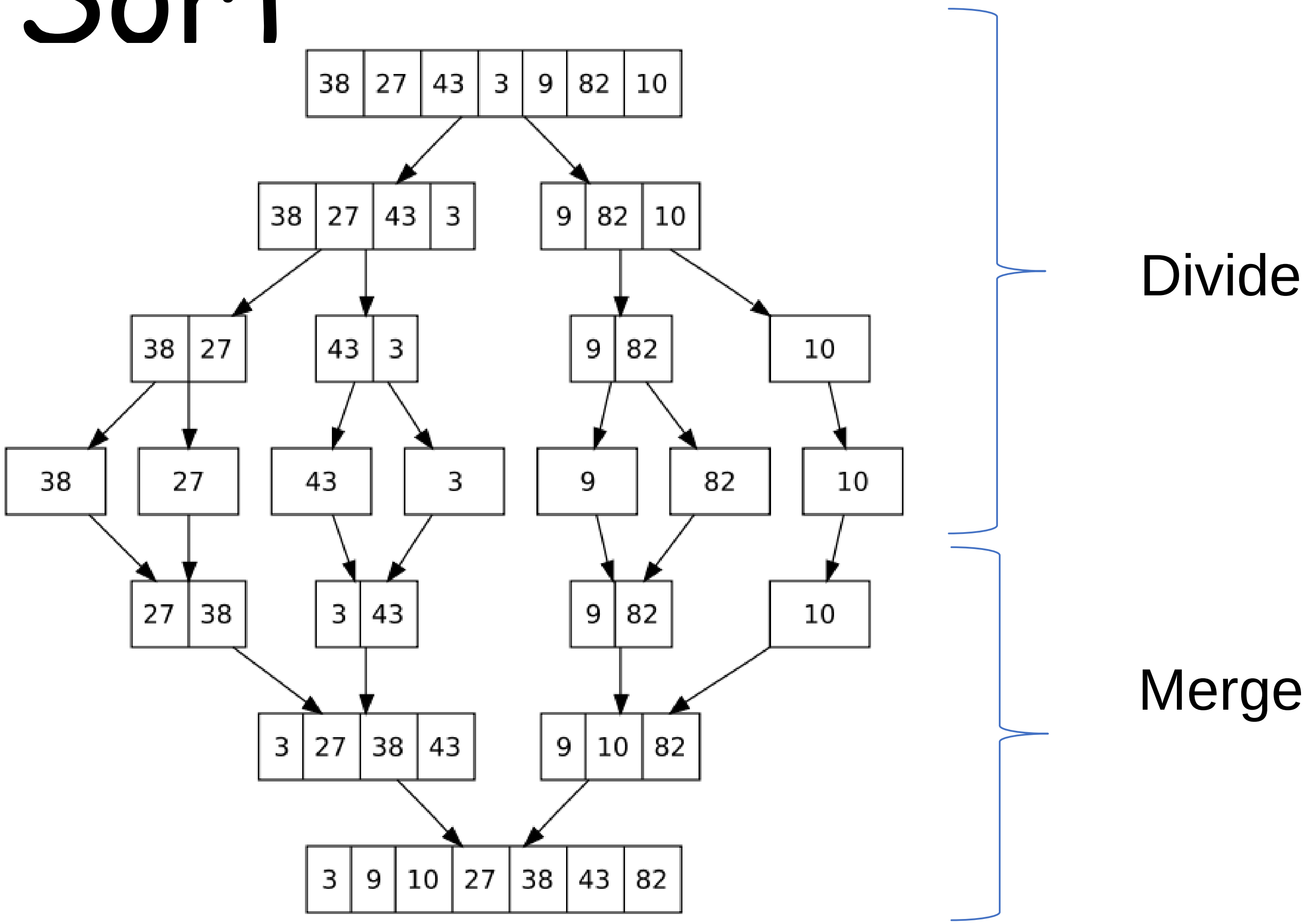


Merge Sort - Merging Sorted Arrays

MERGE-SORT(A,1,7)

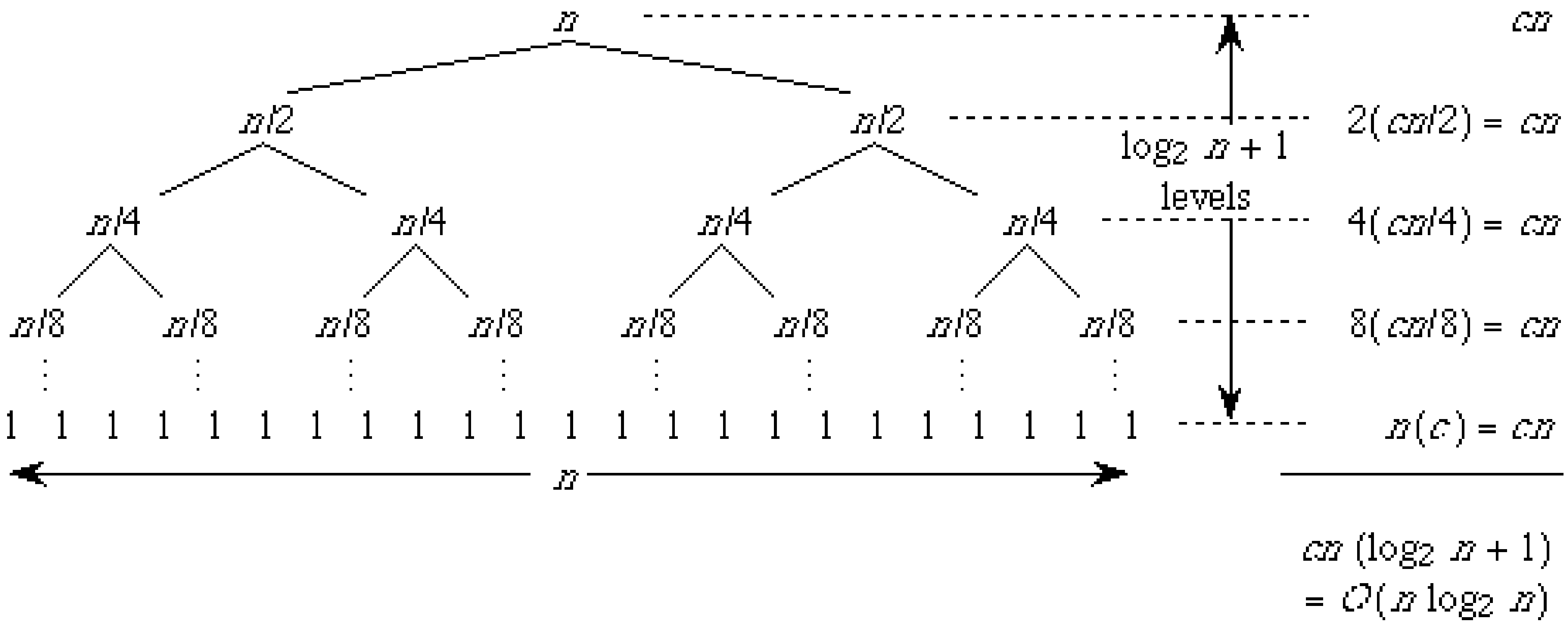


Merge Sort



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82
Divide
Merge

Merge Sort - Recursion Tree

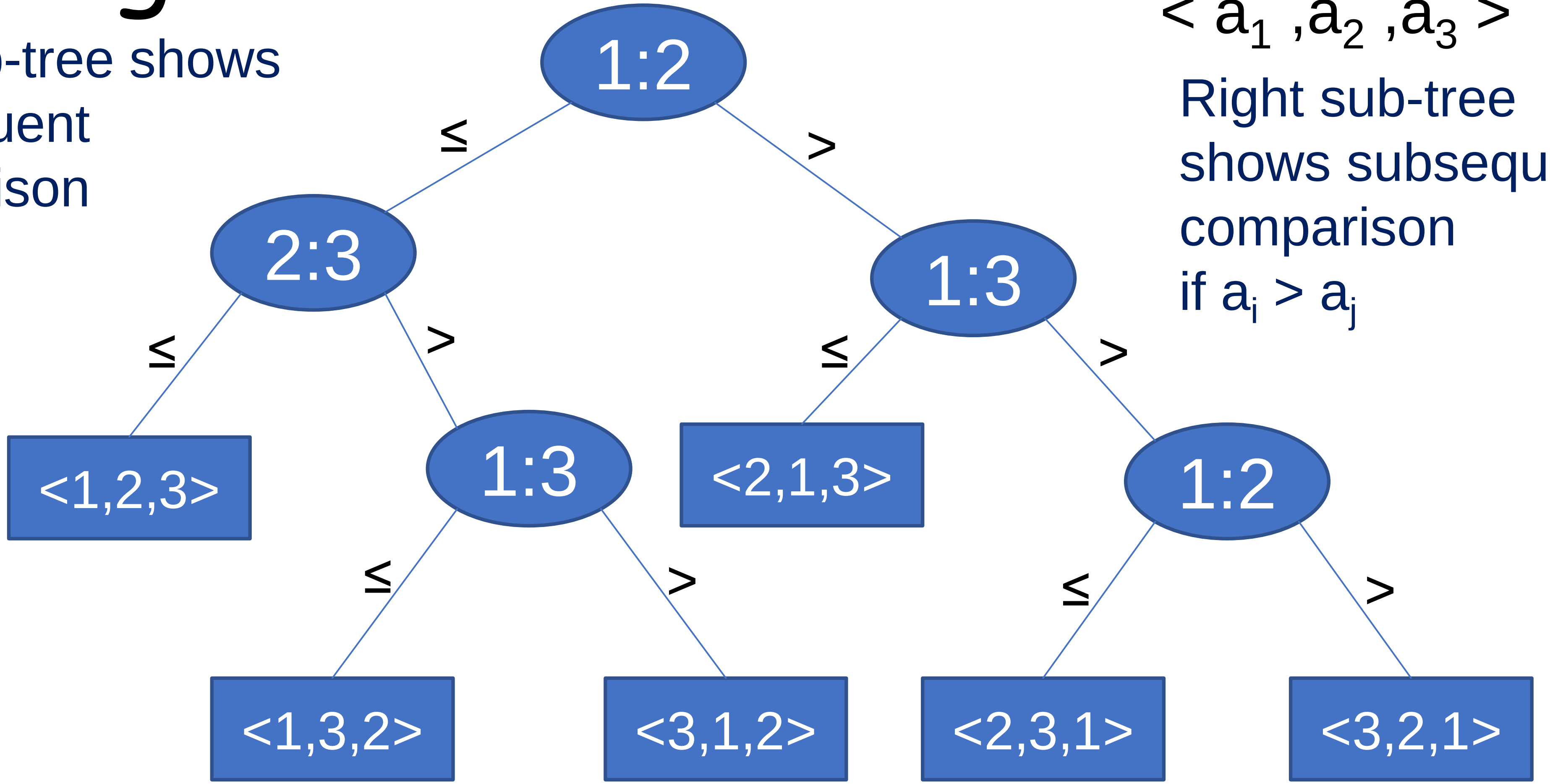


Decision Tree for Comparison Sorting

$< a_1, a_2, a_3 >$

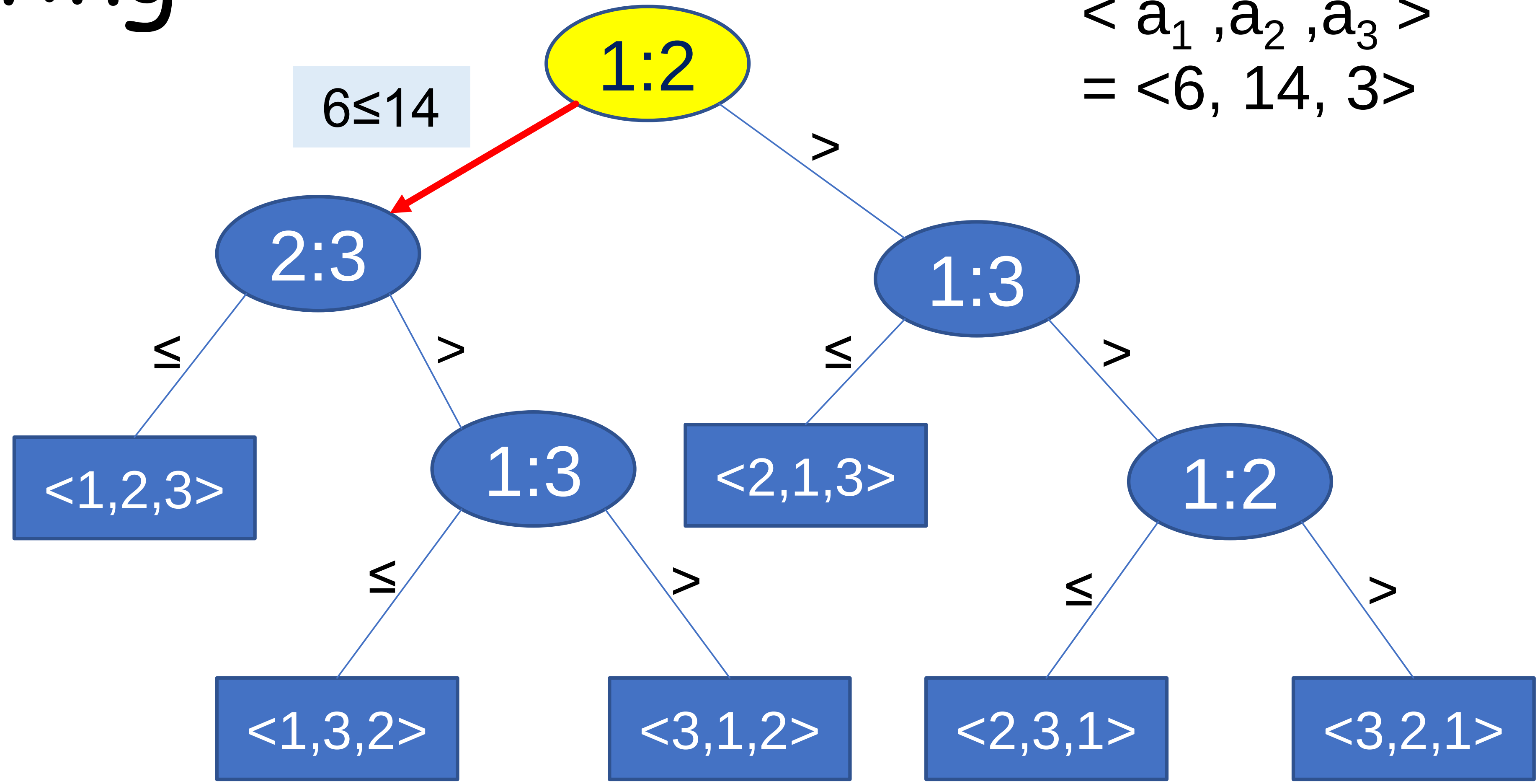
Left sub-tree shows subsequent comparison if $a_i \le a_j$

Right sub-tree shows subsequent comparison if $a_i > a_j$



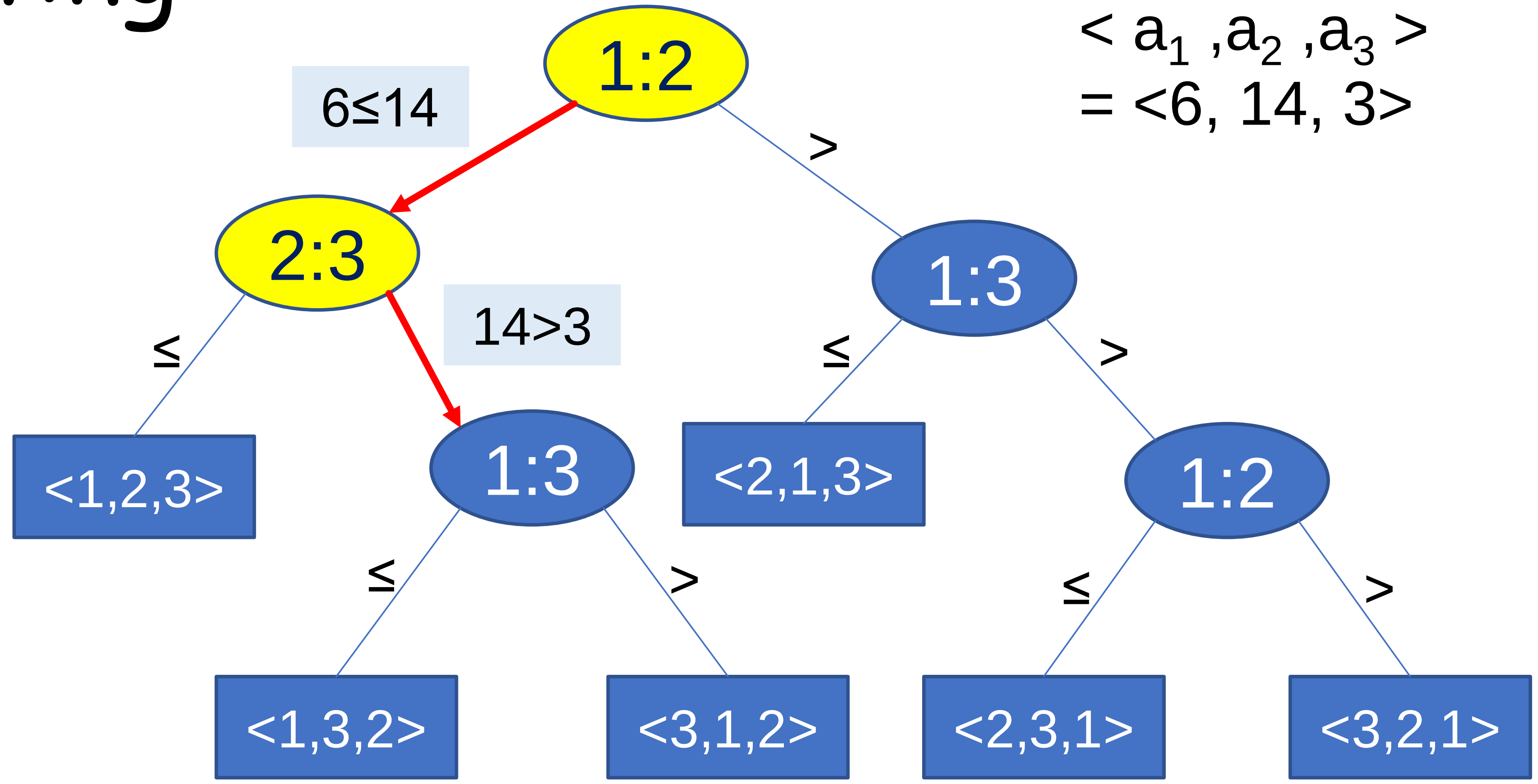
Decision Tree for Comparison Sorting

$< a_1 ,a_2 ,a_3 >$
$= <6, 14, 3>$



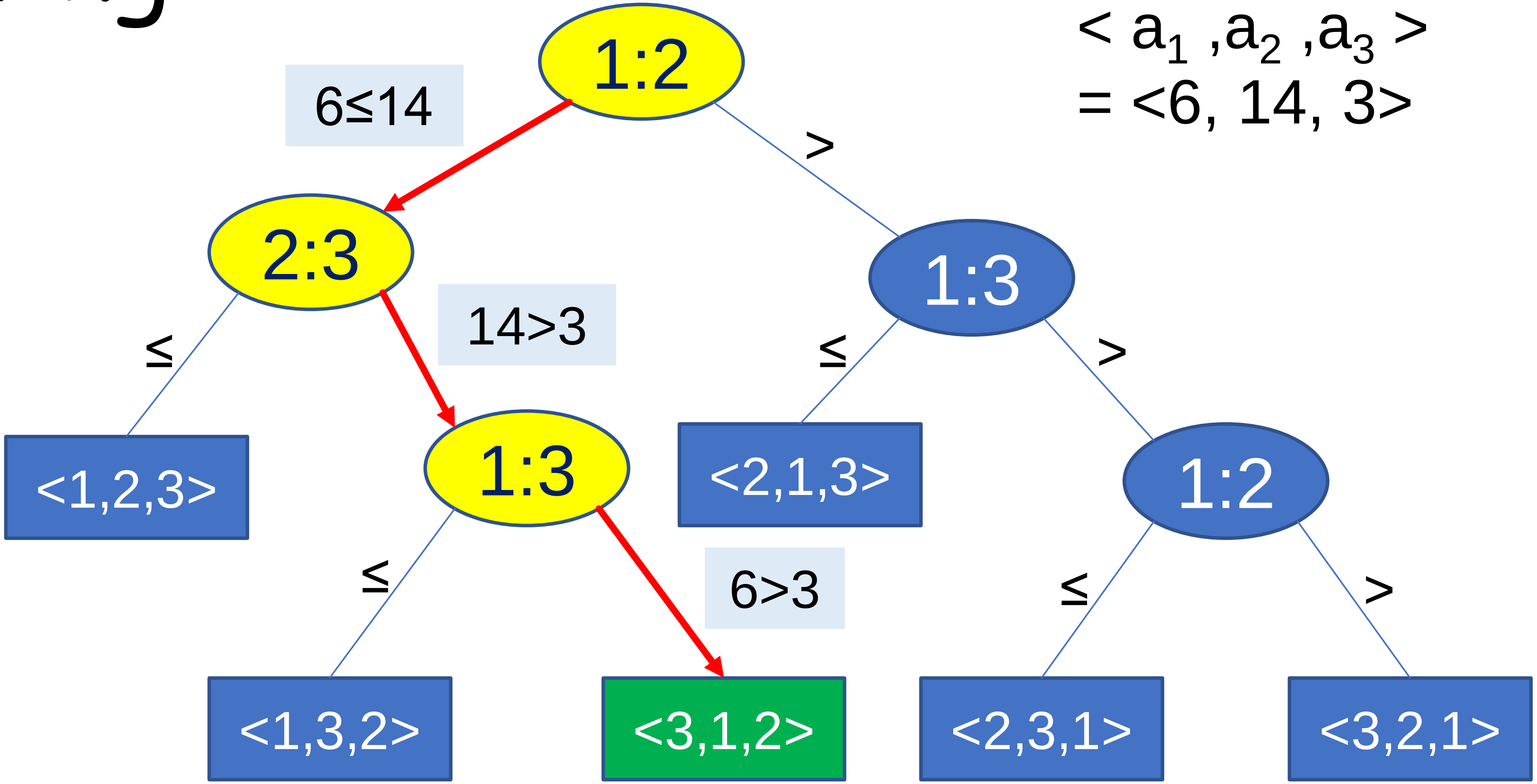
Decision Tree for Comparison Sorting

$< a_1, a_2, a_3 >$
$= <6, 14, 3>$

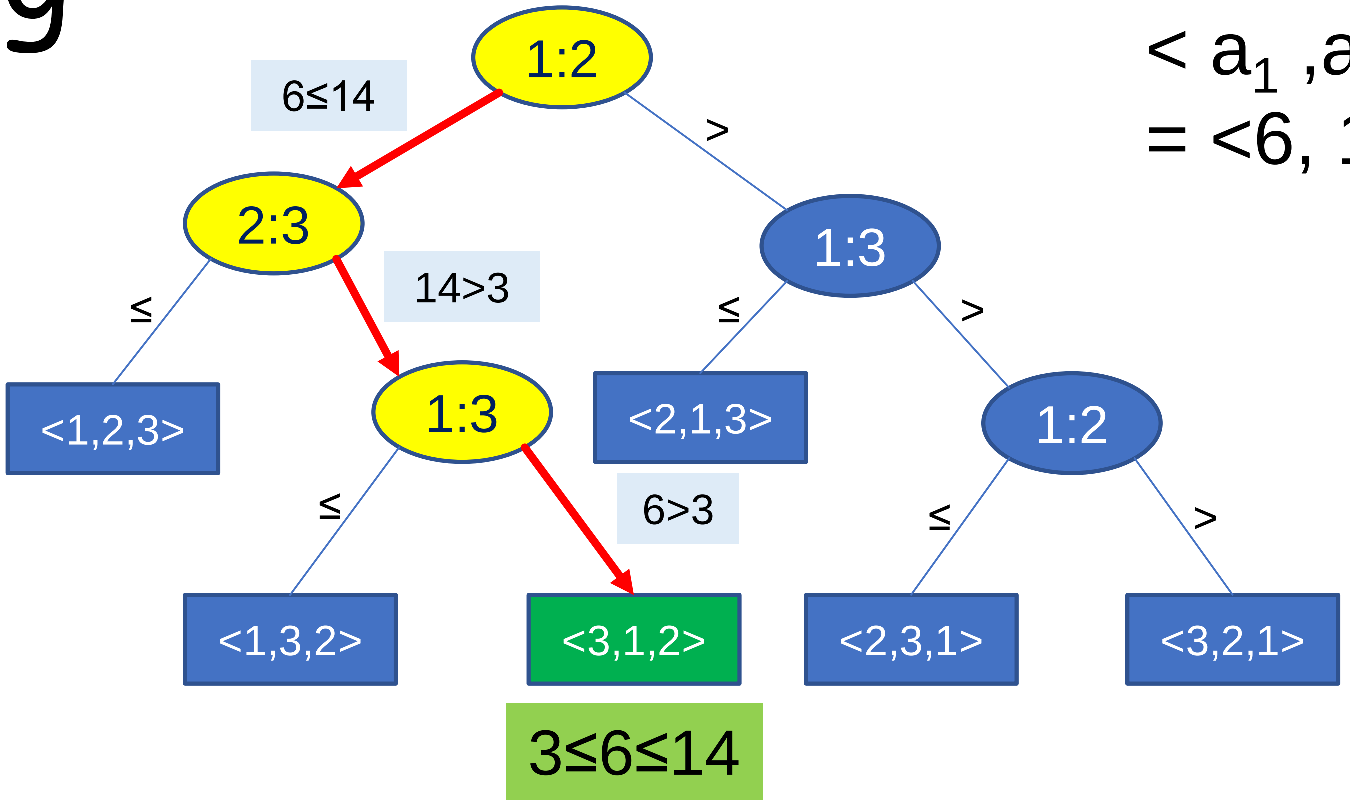


Decision Tree for Comparison Sorting

$< a_1, a_2, a_3 > = <6, 14, 3>$



Decision Tree for Comparison Sorting



$< a_1, a_2, a_3 >$
$= <6, 14, 3>$

- Each leaf contains a permutation $<\pi(1), \pi(2), \ldots, \pi(n)>$ to indicate that the ordering $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)},$

Decision Tree Model

A **Decision Tree** can model the execution of any **Comparison Sort**.

- One tree for each input size n.
- View the algorithm as splitting whenever it compare two elements.
- The ***Decision Tree*** contains the comparisons along its path.
- Running Time = The Length of the Path
- Worst-case Running Time = Height of Tree

Lower Bound for Decision Tree Sorting

Theorem: Any decision tree that can sort n elements must have height $\Omega(n\lg n)$.

Proof: The tree must have $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$\therefore\ h \geq \lg(n!)$ [lg is monotonically increasing function]

$\geq \lg((n/e)^n)$ [Stirling's formula]

$\geq n\lg n - n\lg e$

$\geq \Omega(n\lg n)$.

Lower Bound for Comparison Sorting

Corollary: Heap Sort and Merge Sort are asymptotically optimal comparison sorts.

Sorting in Linear Time

Counting Sort

- No comparison between elements
- Input: A[1…n], where A[j]∈{1,2,…,k}
- Output: B[1…n], sorted
- Auxiliary storage: C[1…k]

Counting Sort

```
Counting Sort(A, B, n, k)
For i = 1 to k
        C[i] = 0
For j = 1 to n
        C[A[j]] = C[A[j]] + 1
For i = 2 to k
        C[i] = C[i] + C[i-1]
For j = n to 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] – 1
```

Counting Sort

A	6	1	2	5	2	
B						
C	1	2	0	0	1	1
	1	2	3	4	5	6
j	1	2	3	4	5	

```
Counting Sort(A, B, n, k)
For i = 1 to k
    C[i] = 0
For j = 1 to n
    C[A[j]] = C[A[j]] + 1
For i = 2 to k
    C[i] = C[i] + C[i-1]
For j = n to 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] – 1
```

Counting Sort

A	6	1	2	5	2	
B						
C	1	3	3	3	4	5
	1	2	3	4	5	6
j	1	2	3	4	5	
i	2	3	4	5	6	

```
Counting Sort(A, B, n, k)
For i = 1 to k
    C[i] = 0
For j = 1 to n
    C[A[j]] = C[A[j]] + 1
For i = 2 to k
    C[i] = C[i] + C[i-1]
For j = n to 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] – 1
```

Counting Sort

A	6	1	2	5	2	
B	1	2	2	5	6	
C	0	1	3	3	3	4
	1	2	3	4	5	6
j	5	4	3	2	1	

Array is sorted!

```
Counting Sort(A, B, n, k)
1    For i = 1 to k
2        C[i] = 0
3    For j = 1 to n
4        C[A[j]] = C[A[j]] + 1
5    For i = 2 to k
6        C[i] = C[i] + C[i-1]
7    For j = n to 1
8        B[C[A[j]]] = A[j]
9        C[A[j]] = C[A[j]] – 1
```

Counting Sort - Analysis

Counting Sort(A, B, n, k)

```
For i = 1 to k
    C[i] = 0
For j = 1 to n
    C[A[j]] = C[A[j]] + 1
For i = 2 to k
    C[i] = C[i] + C[i-1]
For j = n to 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] – 1
```

$\theta(k)$

$\theta(n)$

$\theta(k)$

$\theta(n)$

$\Theta(n+k)$

Counting Sort - Analysis

- If K=O(n) then counting sort takes Θ(n) time.
- But, sorting takes Ω(nlgn) time!!!!!
- How Counting sort is linear?

Answer

- *Comparison Sort* takes Ω(nlgn) time.
- Counting Sort is not *Comparison Sort*.
- Not a single comparison is done!!!!!!!

Reference and Reading Material

- Heap Sort – Link, Link, Link
- Merge Sort – Link
- Counting Sort – Link