Introduction to Algorithms

Meng-Tsung Tsai

09/17/2019

Course Materials

Textbook

Introduction to Algorithms (I2A) 3rd ed. by Cormen, Leiserson, Rivest, and Stein.

Reference Book

Algorithms (JfA) 1st ed. by Erickson. An e-copy can be downloaded from author's website: http://jeffe.cs.illinois.edu/teaching/algorithms/

Websites

http://e3new.nctu.edu.tw for slides, written assignments, and solutions.

http://oj.nctu.me for programming assignments.

Office Hours

Lecturer's

On Wednesdays 16:30 - 17:20 at EC 336 (工程三館).

TA. Erh-Hsuan Lu (呂爾軒) and Tsung-Ta Wu (吳宗達)

On Mondays 10:10 - 11:00 at ES 724 (電資大樓).

TA. Yung-Ping Wang (王詠平) and Chien-An Yu (俞建安)

On Thursdays 11:10 - 12:00 at ES 724 (電資大樓).

Announcements

Programming Assignment 0 is for practice only.

Programming Assignment 1 is due by Oct 9, 23:59. at https://oj.nctu.me

We will not normalize the points that you receive from assignments. 100 points is a perfect score, and extra points are considered as a bonus.

Caution: it is very difficult to solve all problems in an assignment.

I am heading to a conference to present my paper. The lecture on Sep 19 will be given by some TA.

Data Structures

What are data structures?

A data structure is a way to store and organize data in order to facilitate access (e.g. search) and modifications (e.g. insertion, deletion).

No single data structure works well for all purposes, so it is important to know the strengths and limitations of a data structure.

n elements	search cost	insertion cost
sorted array	O(log n)	O(n)
unsorted array	O(n)	O(1)

What are data structures?

A data structure is a way to store and organize data in order to facilitate access (e.g. search) and modifications (e.g. insertion, deletion).

No single data structure works well for all purposes, so it is important to know the strengths and limitations of a data structure.

n elements	search cost	insertion cost
sorted array	O(log n)	O(n)
unsorted array	O(n)	O(1)

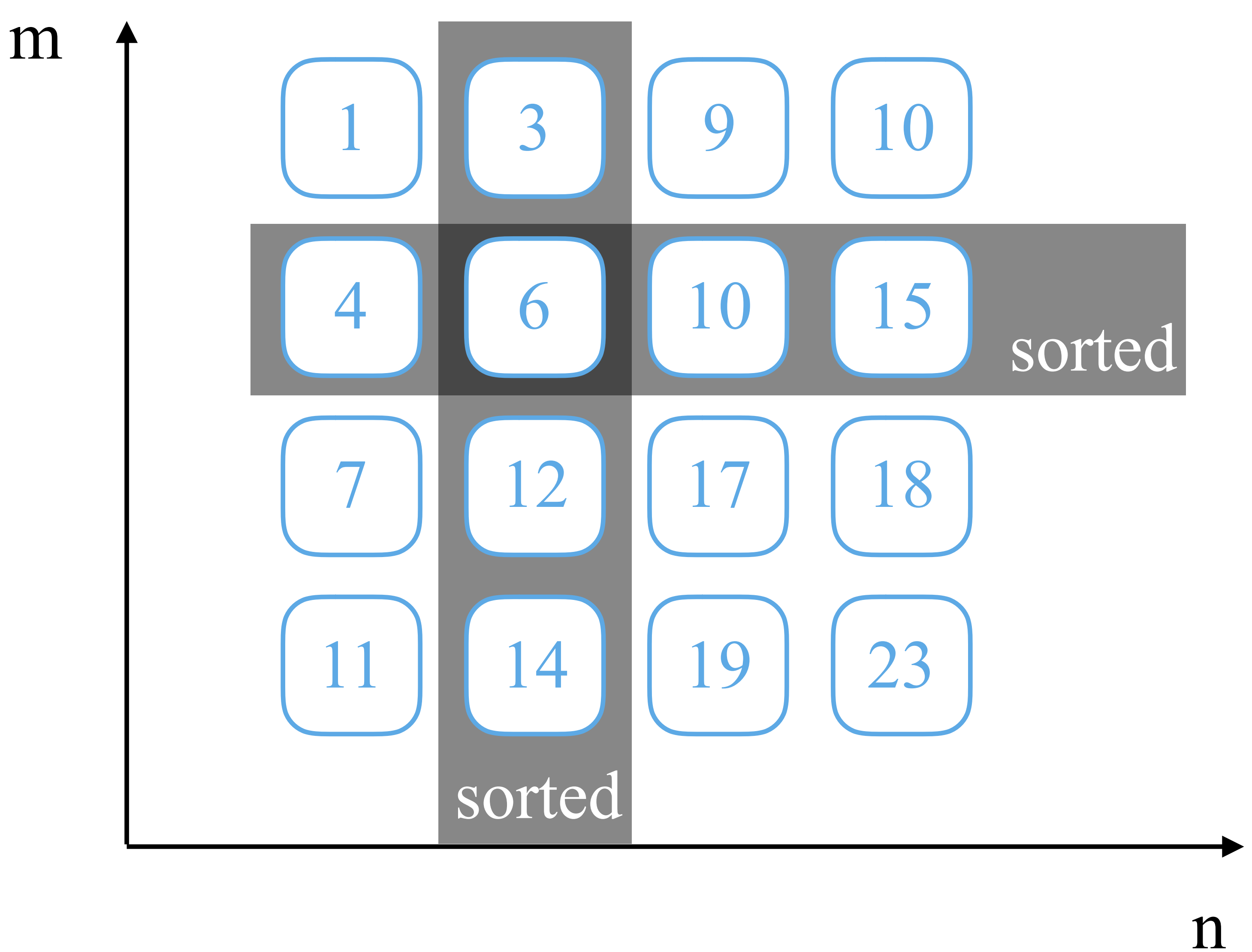
When do unsorted arrays outperform sorted ones?

2D Sorted Arrays

Young Tableau

An m by n Young Tableau is an m by n matrix so that each row and column is sorted in nondecremental order.

<u>Example</u>.

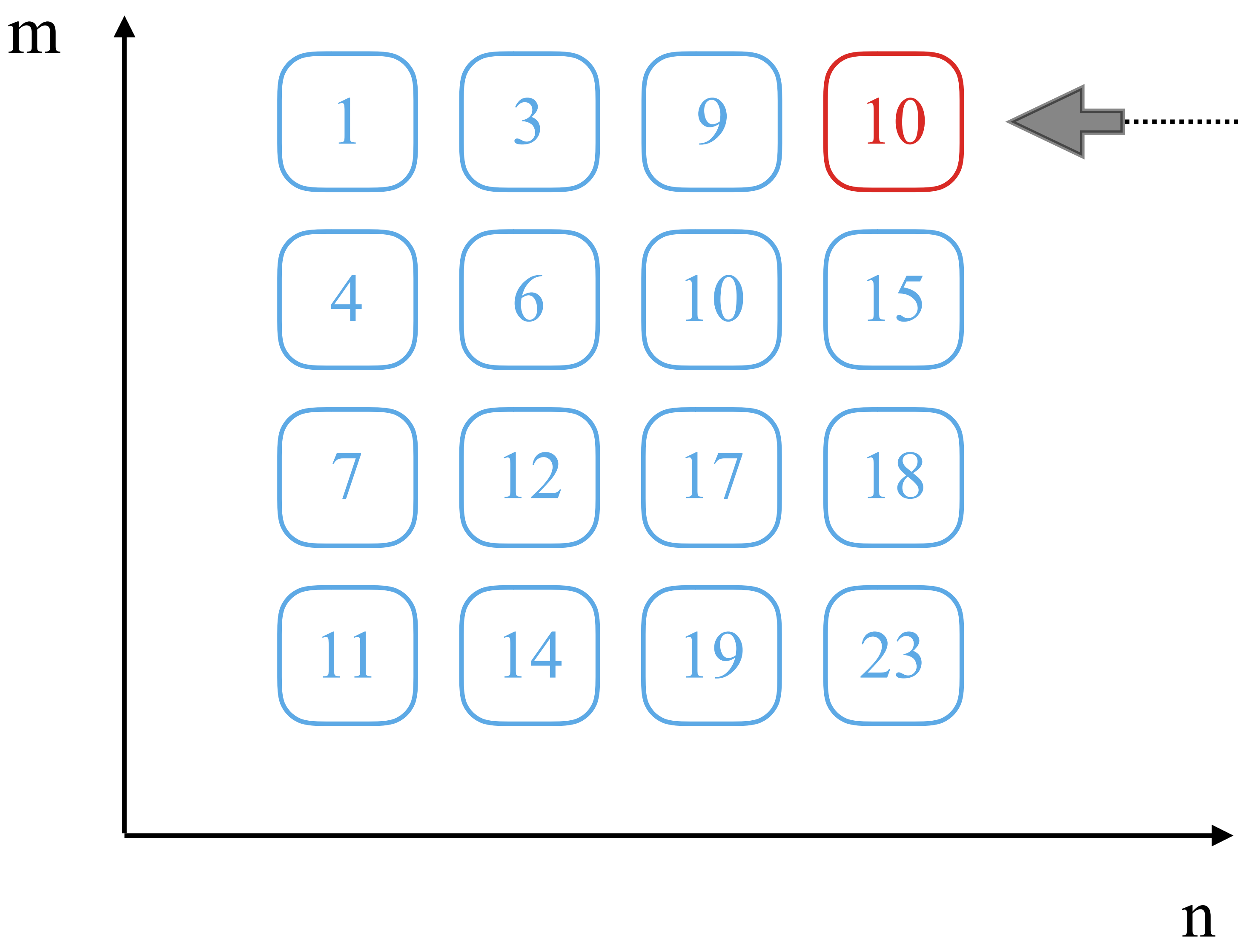


Search on a Young Tableau

Input: a query *x*.

Output: "Yes," if *x* is a member in the tableau; "No," otherwise.

Search cost is O(n+m).



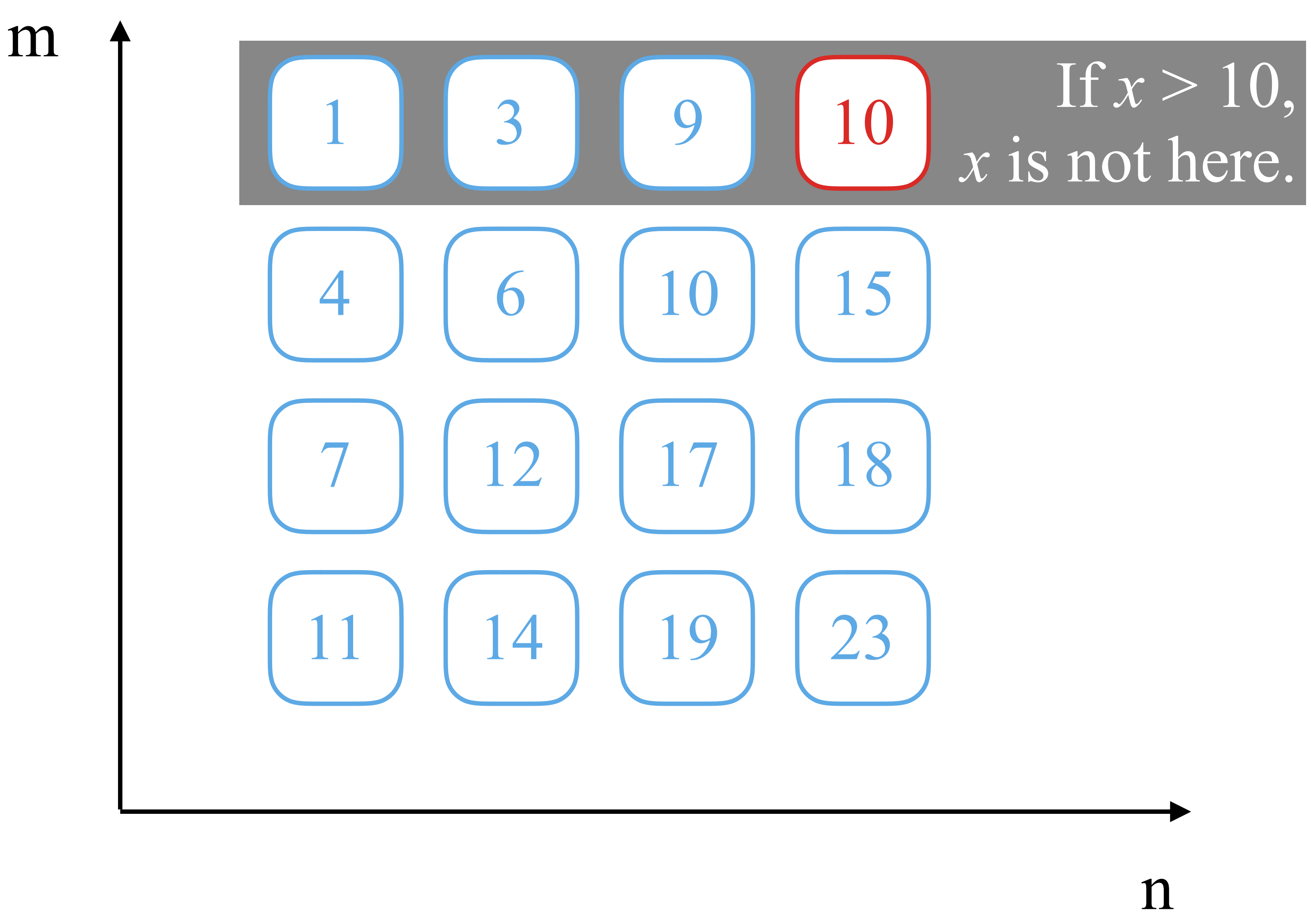
If x = 10, we are done.

Search on a Young Tableau

Input: a query *x*.

Output: "Yes," if *x* is a member in the tableau; "No," otherwise.

Search cost is O(n+m).

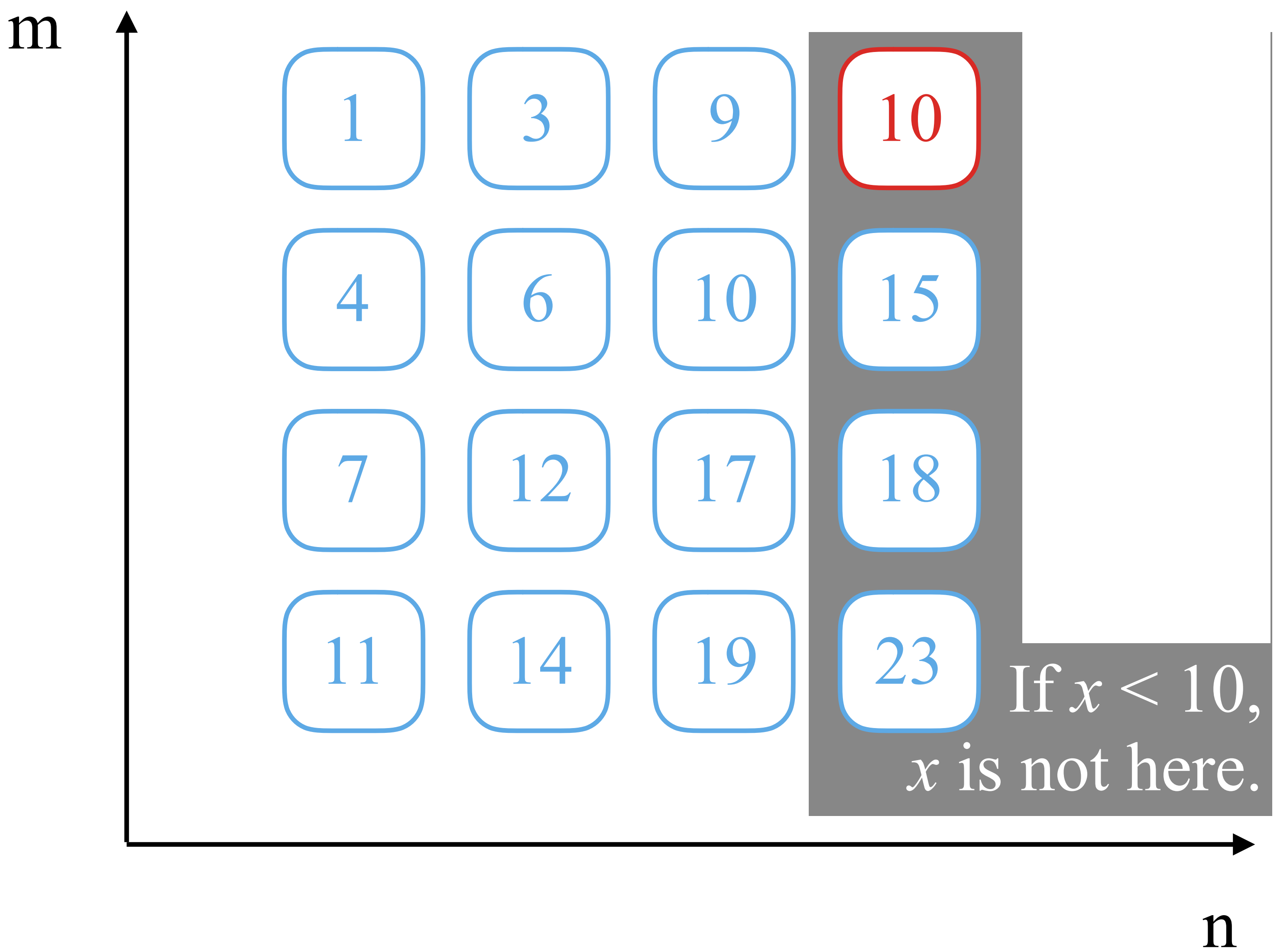


Search on a Young Tableau

Input: a query *x*.

Output: "Yes," if *x* is a member in the tableau; "No," otherwise.

Search cost is O(n+m).

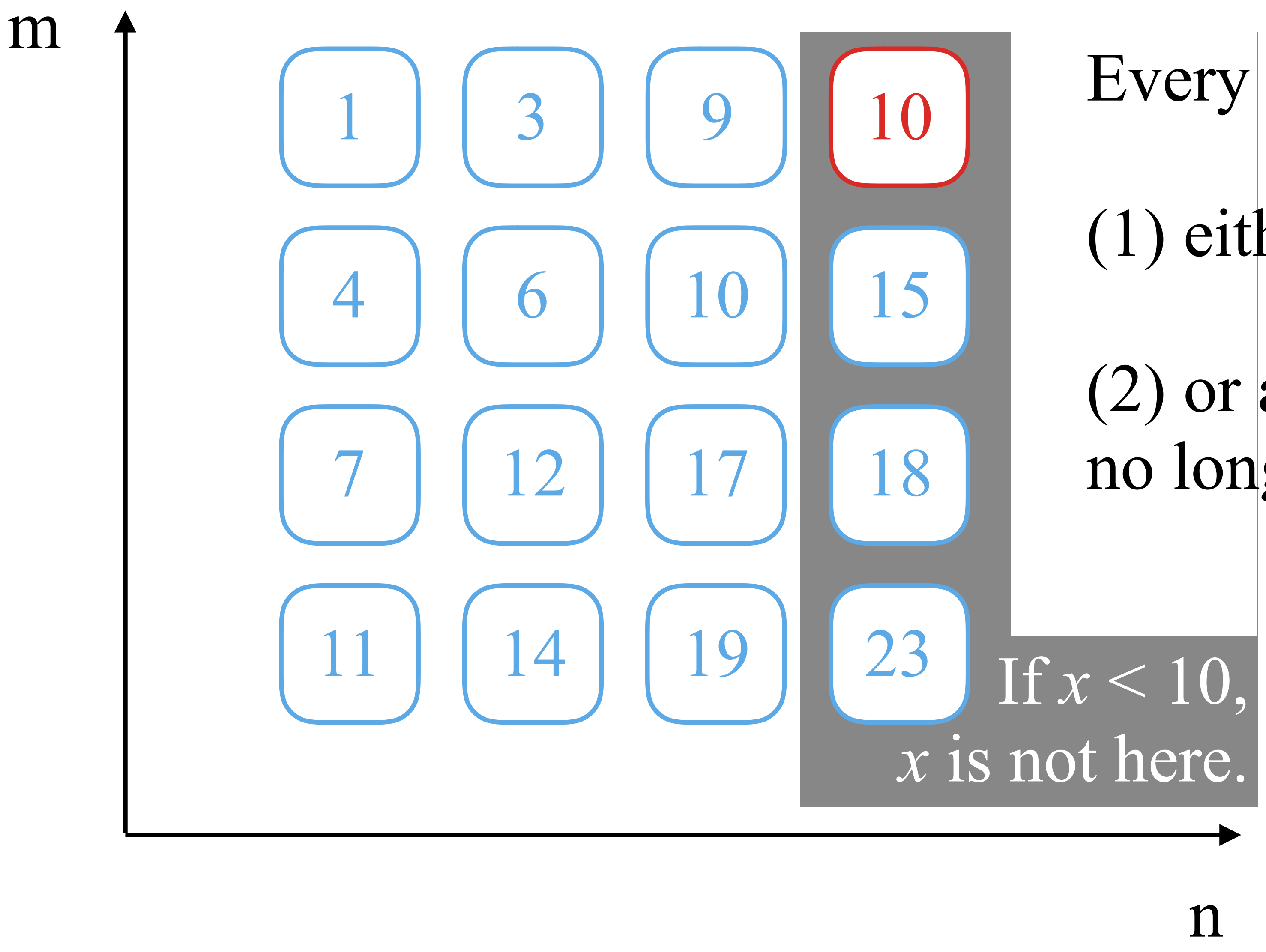


Search on a Young Tableau

Input: a query *x*.

Output: "Yes," if *x* is a member in the tableau; "No," otherwise.

Search cost is O(n+m).



Every time a comparison is made,

(1) either *x* is found,

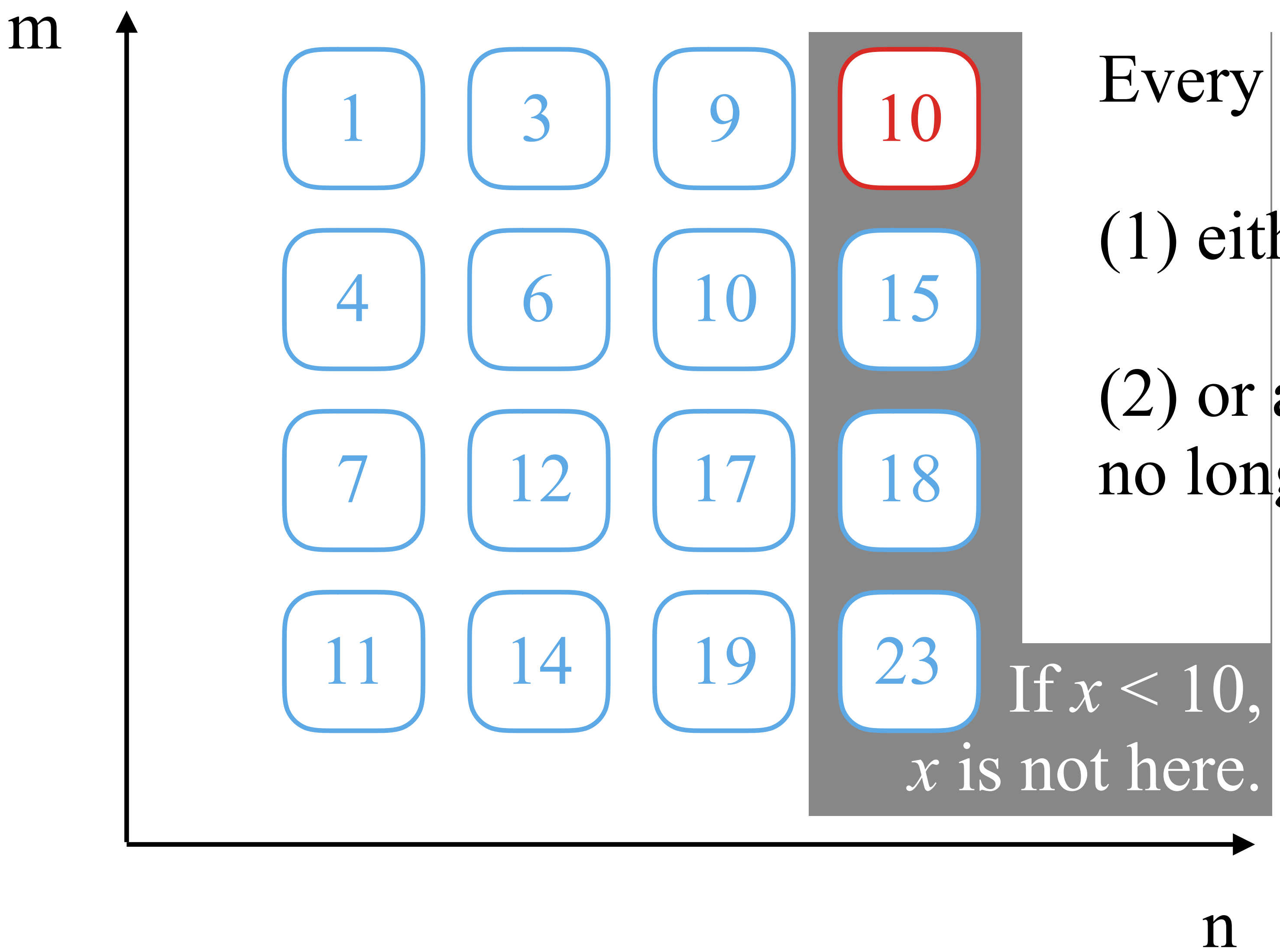
(2) or a row (column) of candidates are no longer possible.

Search on a Young Tableau

Input: a query x.

Output: "Yes," if x is a member in the tableau; "No," otherwise.

Search cost is O(n+m).



Every time a comparison is made,

(1) either x is found,

(2) or a row (column) of candidates are no longer possible.

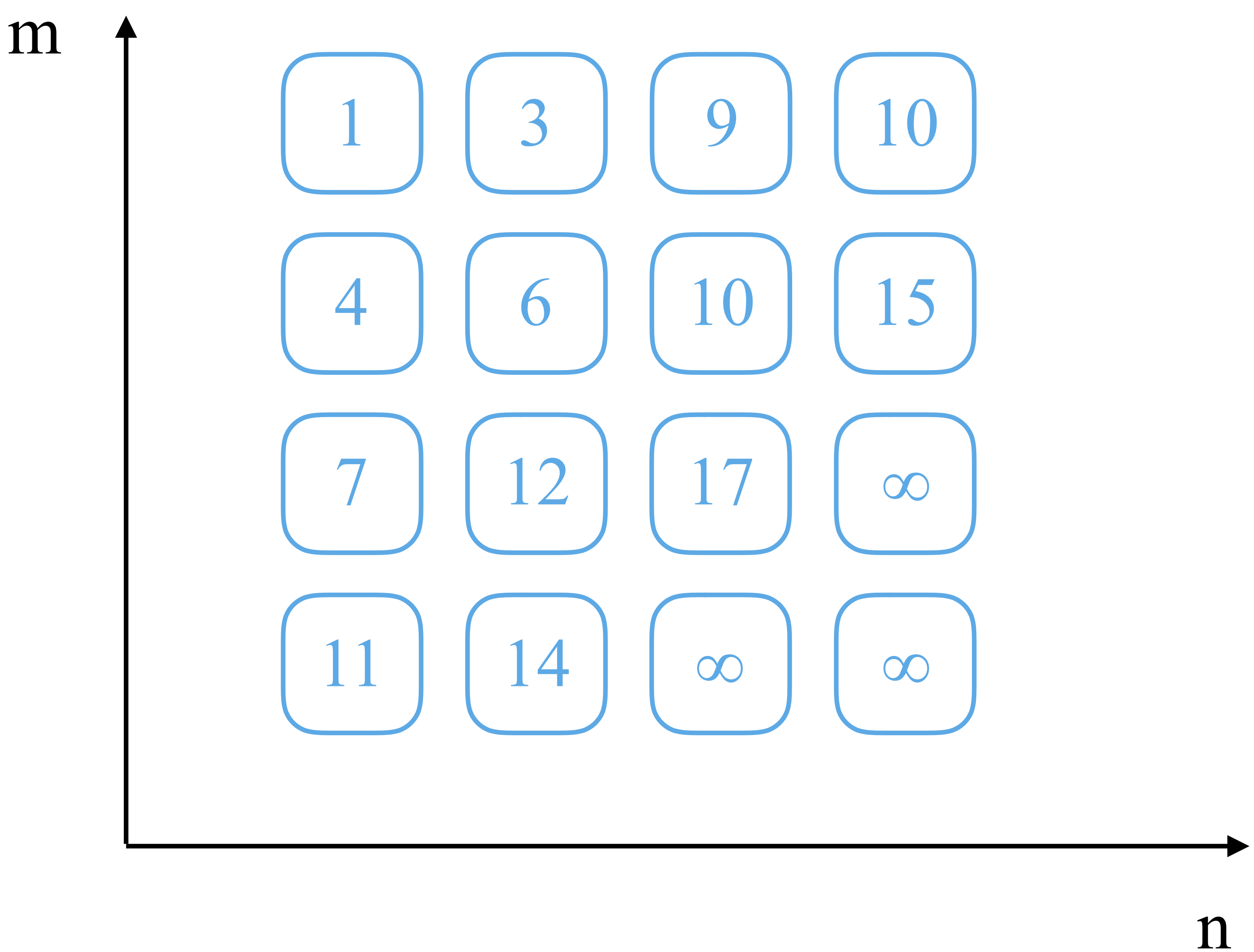
There are O(n+m) rows and columns, yielding the bound.

Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).



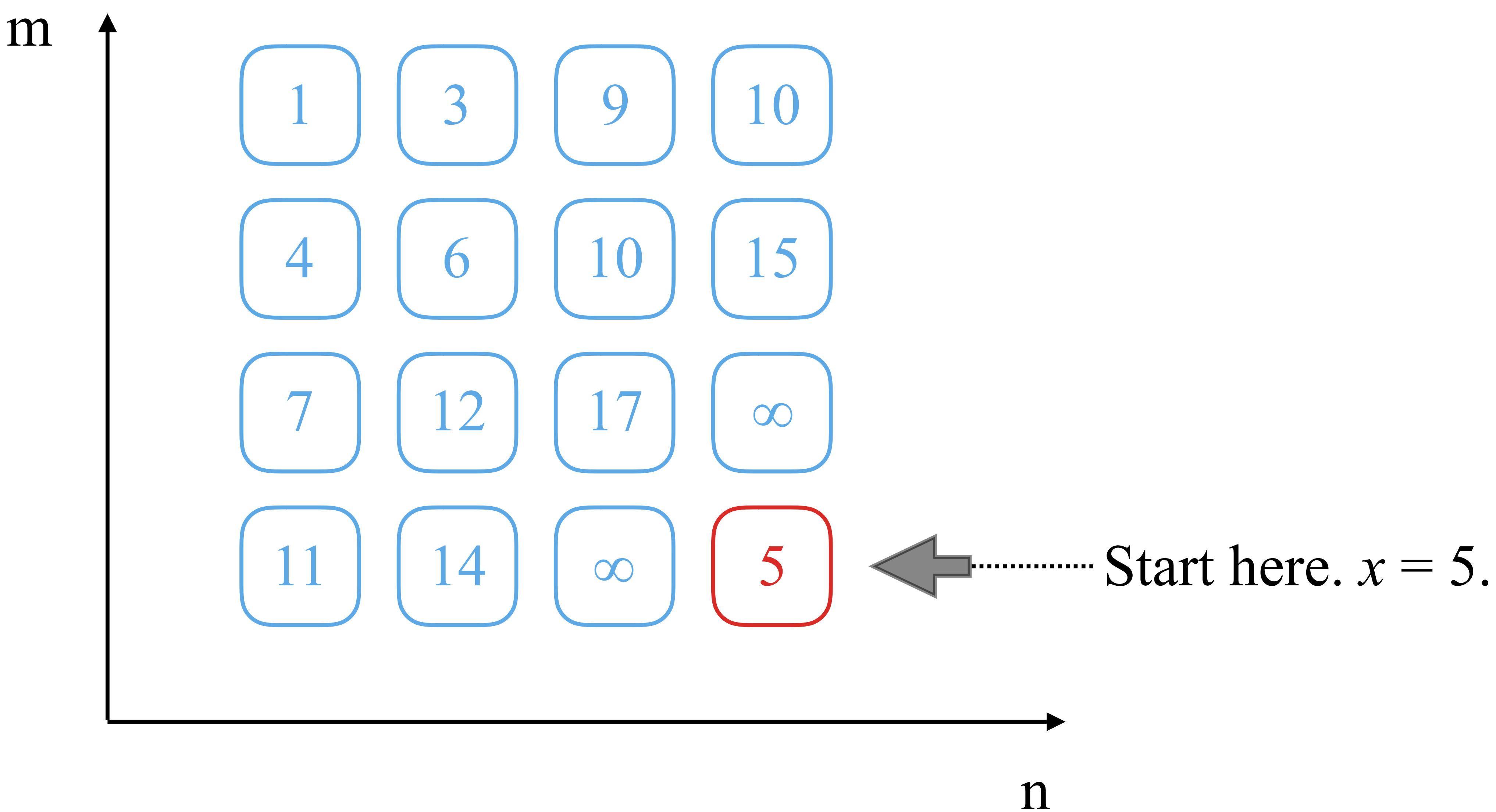
∞ denotes an empty slot.

Insertion on a Young Tableau

Input: a newly-added element x.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

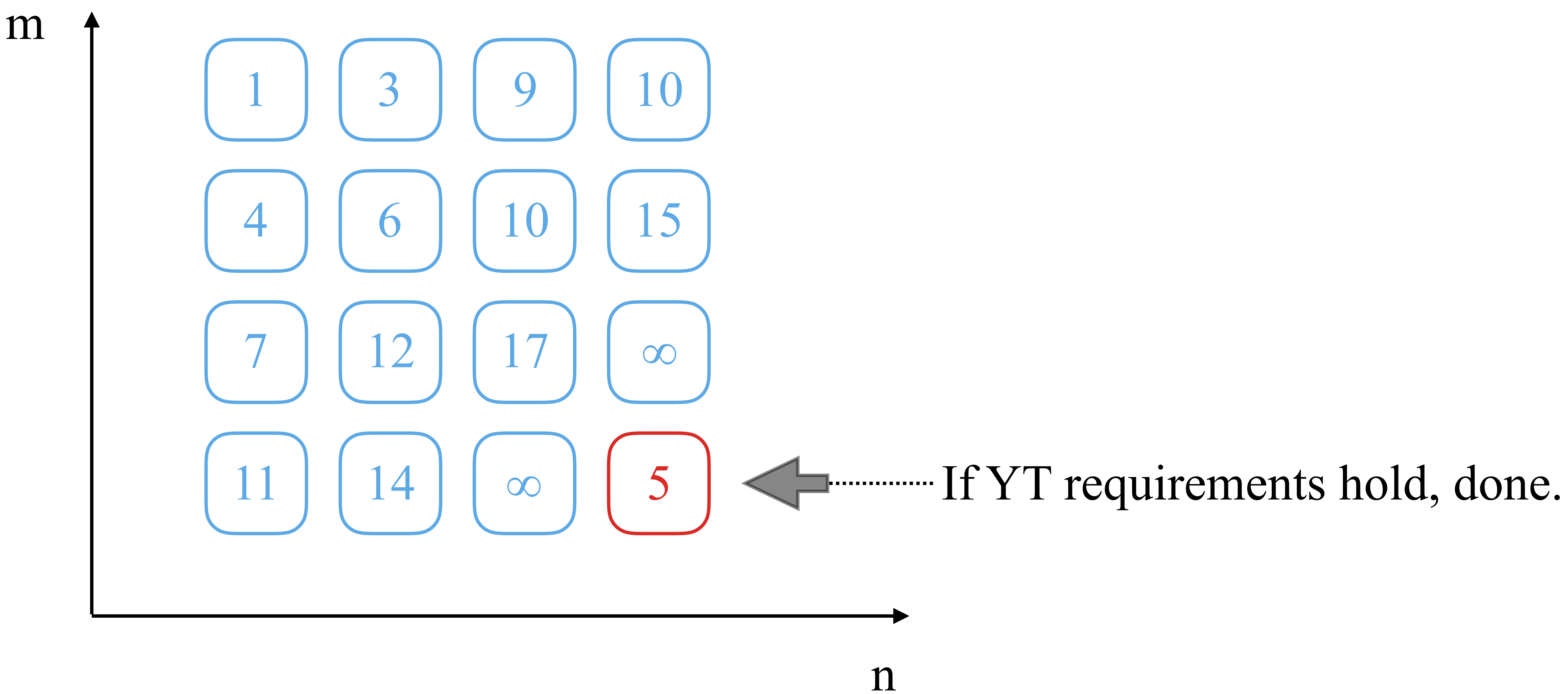


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

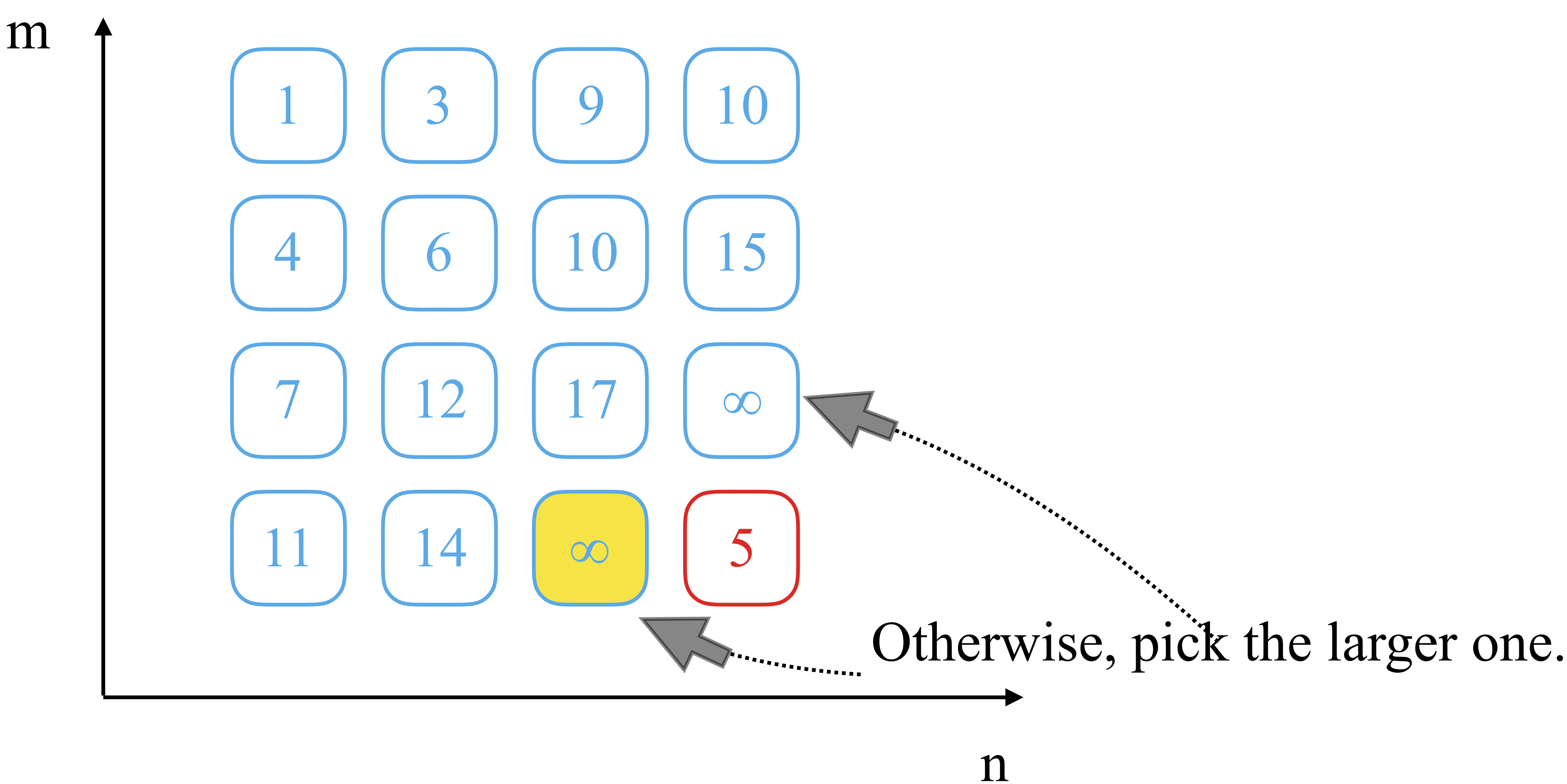


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

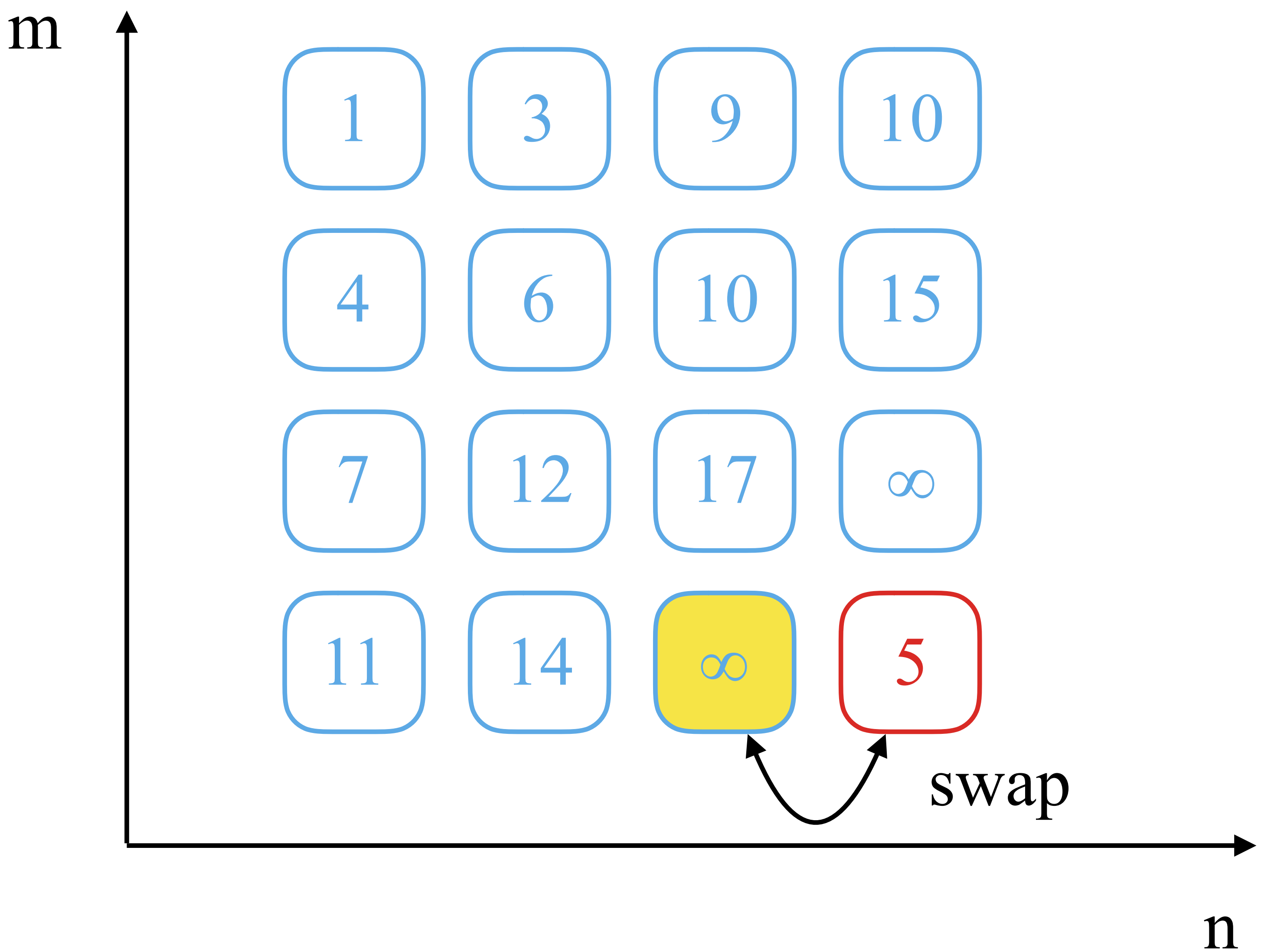


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

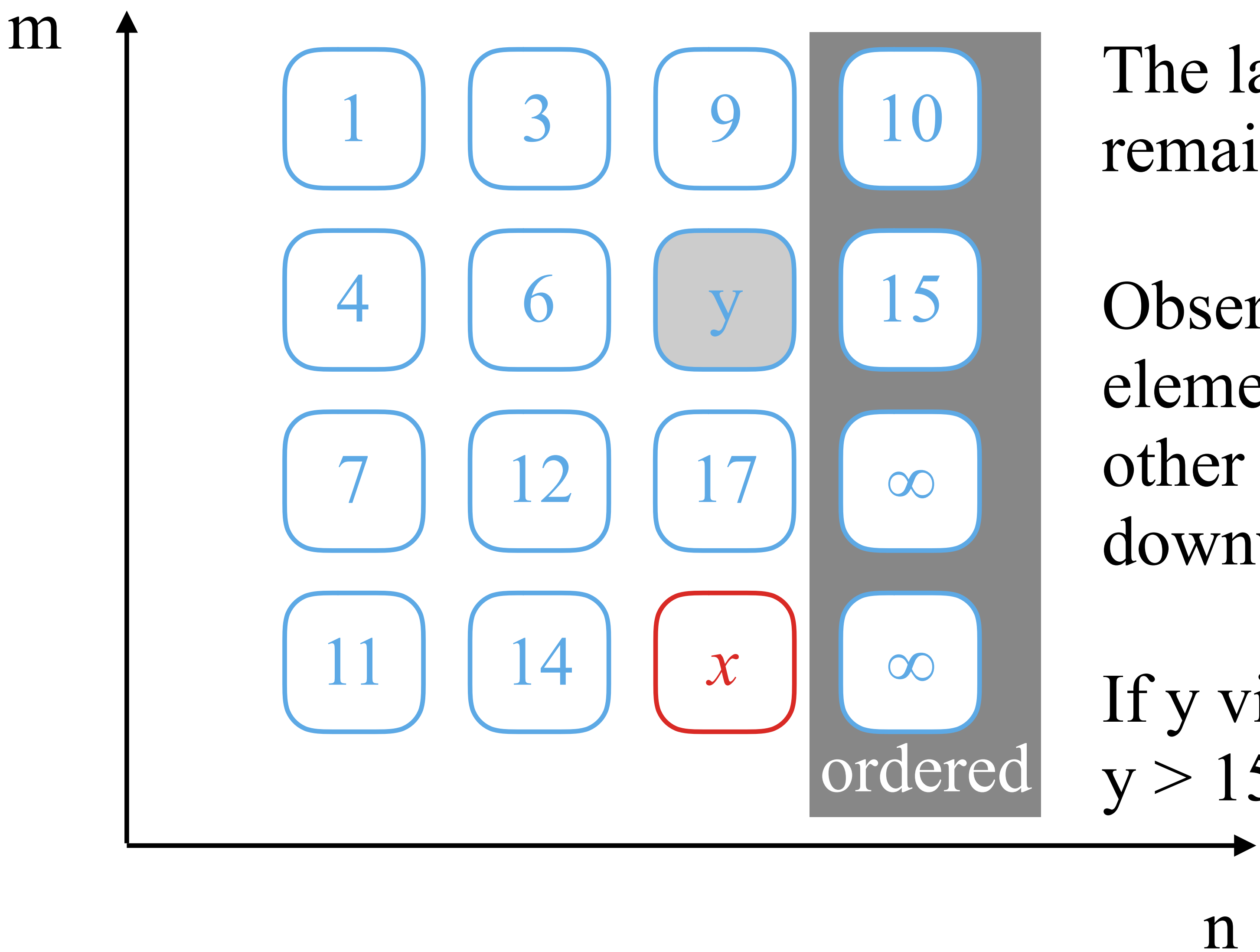


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).



The last column is ordered now, and remains ordered in the subsequent steps.

Observe: (1) *x* is smaller than any element in the last column, and (2) other elements can move only downward and rightward.

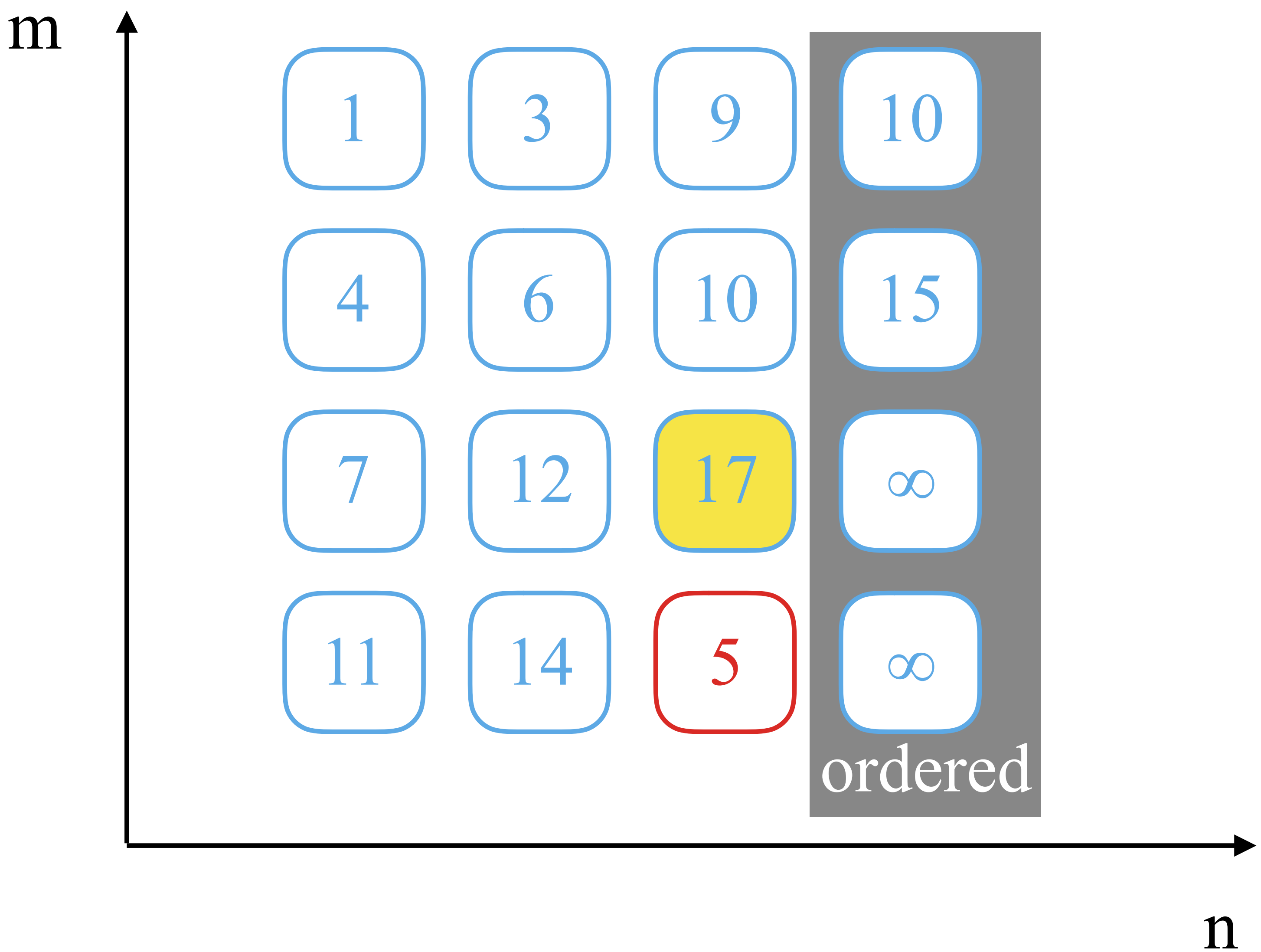
If y violates the requirements, then y > 15, which is impossible.

Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

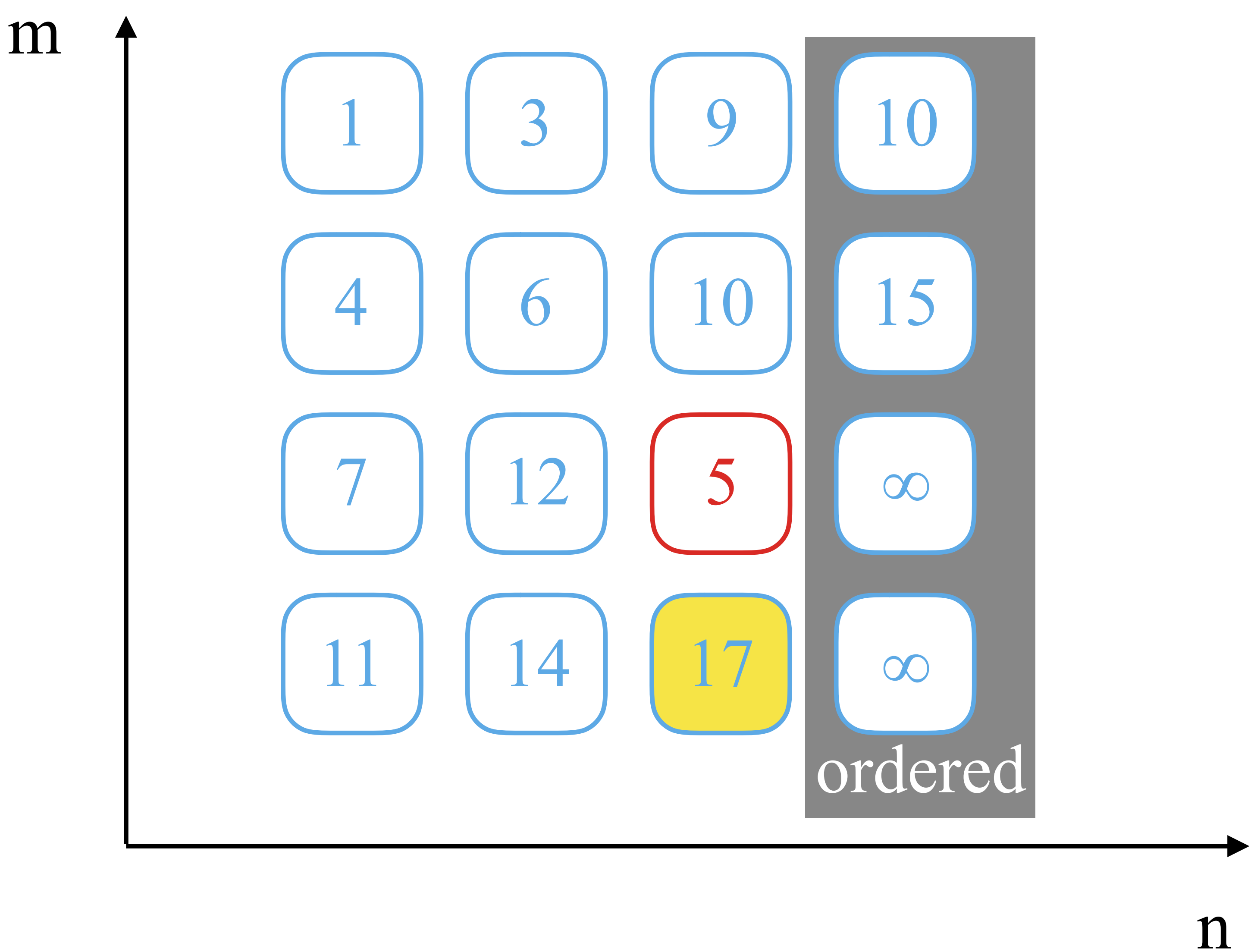


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

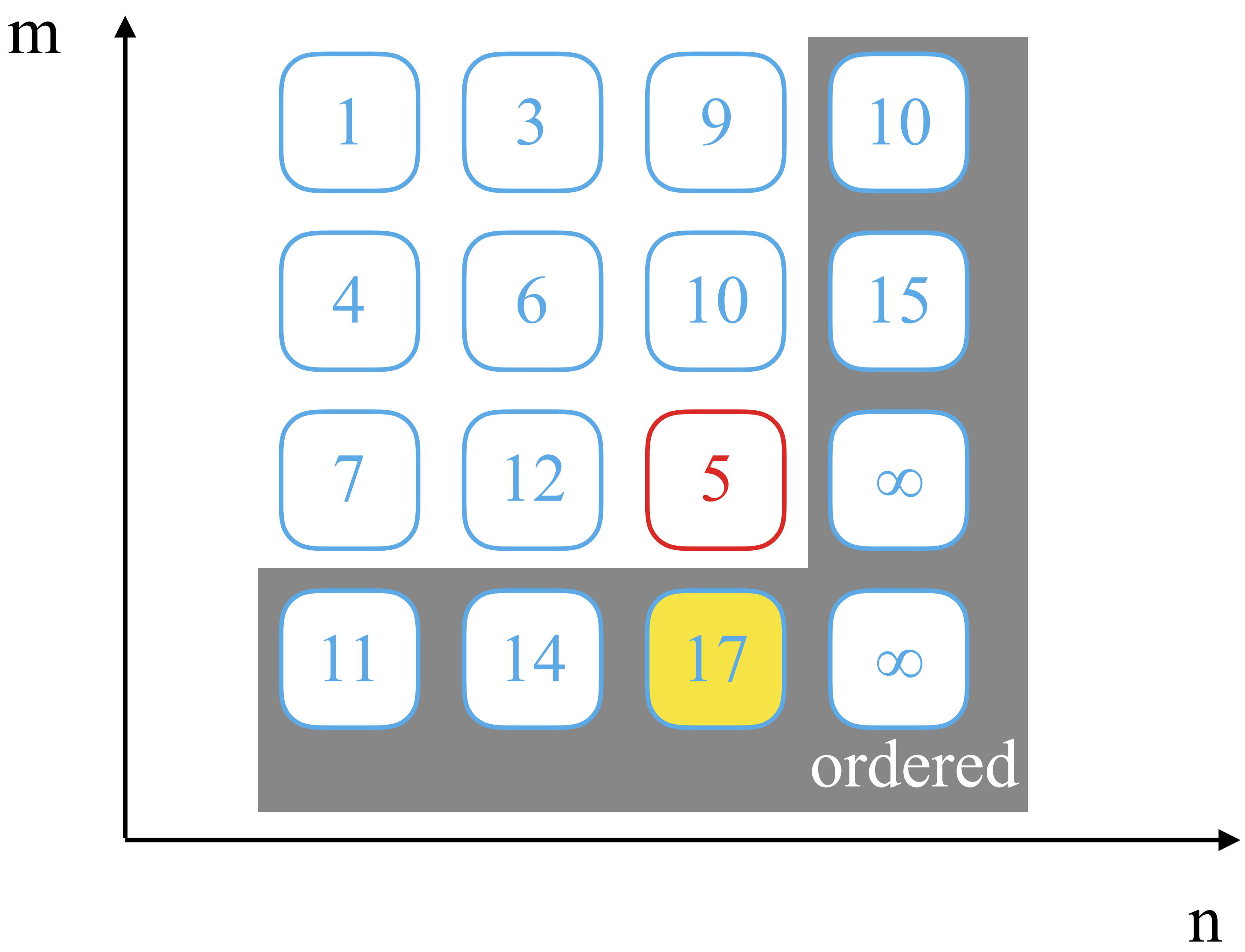


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

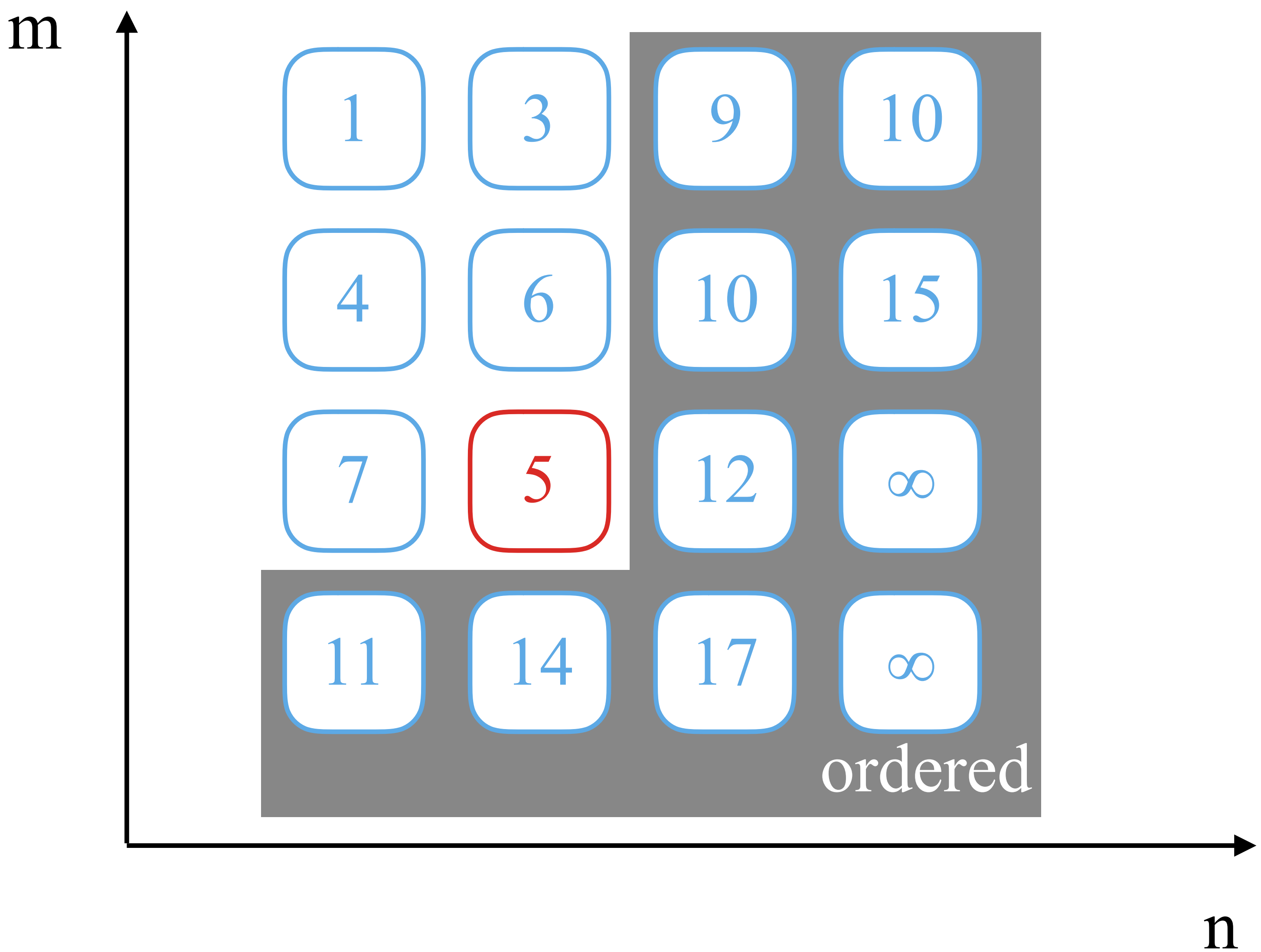


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).

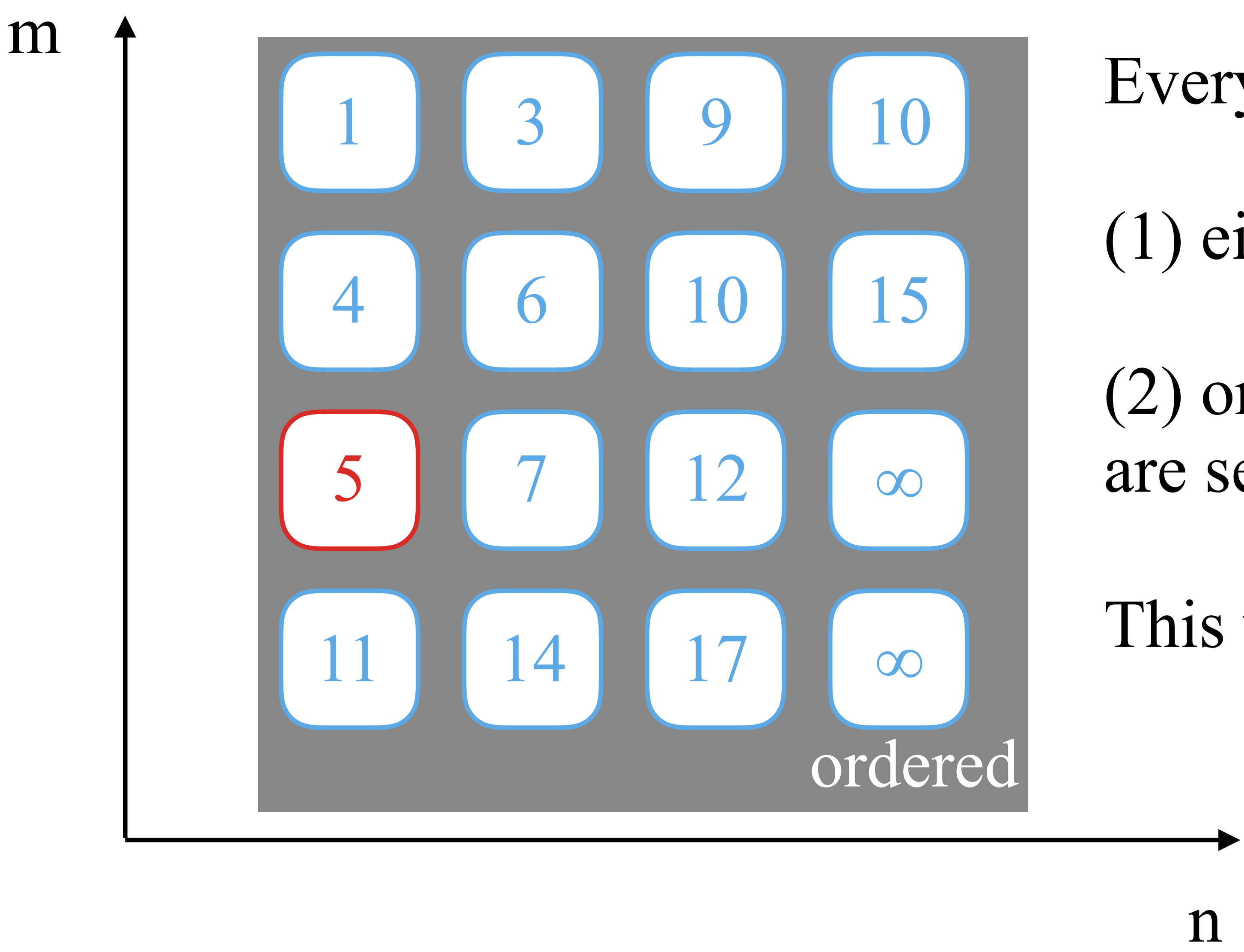


Insertion on a Young Tableau

Input: a newly-added element *x*.

No output. Rearrange data to satisfy the requirements of Young Tableau.

Insertion cost is O(n+m).



Every time a comparison is made,

(1) either *x* is settled,

(2) or a row (column) of elements are settled.

This yields a time bound O(n+m).

Summary

n elements	search cost	insertion cost
sorted array	$O(\log n)$	$O(n)$
Young tableau	$O(n^{1/2})$	$O(n^{1/2})$
unsorted array	$O(n)$	$O(1)$

Exercise

Input: an array A of n integers and a query x.

Output: "Yes," $x = A[i]+A[j]$ for some i, j in [1, n]; "No," otherwise.

Can you solve this problem in O(n) time?

Exercise (3SUM)

Input: an array A of n integers.

Output: "Yes," $A[k] = A[i]+A[j]$ for some i, j, k in $[1, n]$; "No," otherwise.

Can you solve this problem in $O(n^2)$ time?

Exercise (3SUM)

Input: an array A of n integers.

Output: "Yes," A[k] = A[i]+A[j] for some i, j, k in [1, n]; "No," otherwise.

Can you solve this problem in $O(n^2)$ time?

3SUM Conjecture: no algorithm can solve 3SUM in $n^{2-\Omega(1)}$ time on a RAM.

Heaps

Heaps

An array A is a max heap if for every i in [1, n], element A[i] has value less than or equal to its parent A[parent(i)] where

$$\text{parent(i)} = \lfloor i/2 \rfloor.$$

Min heaps are ordered in the opposite way; that is, A[i] ≥ A[parent(i)] for every i in [1, n].

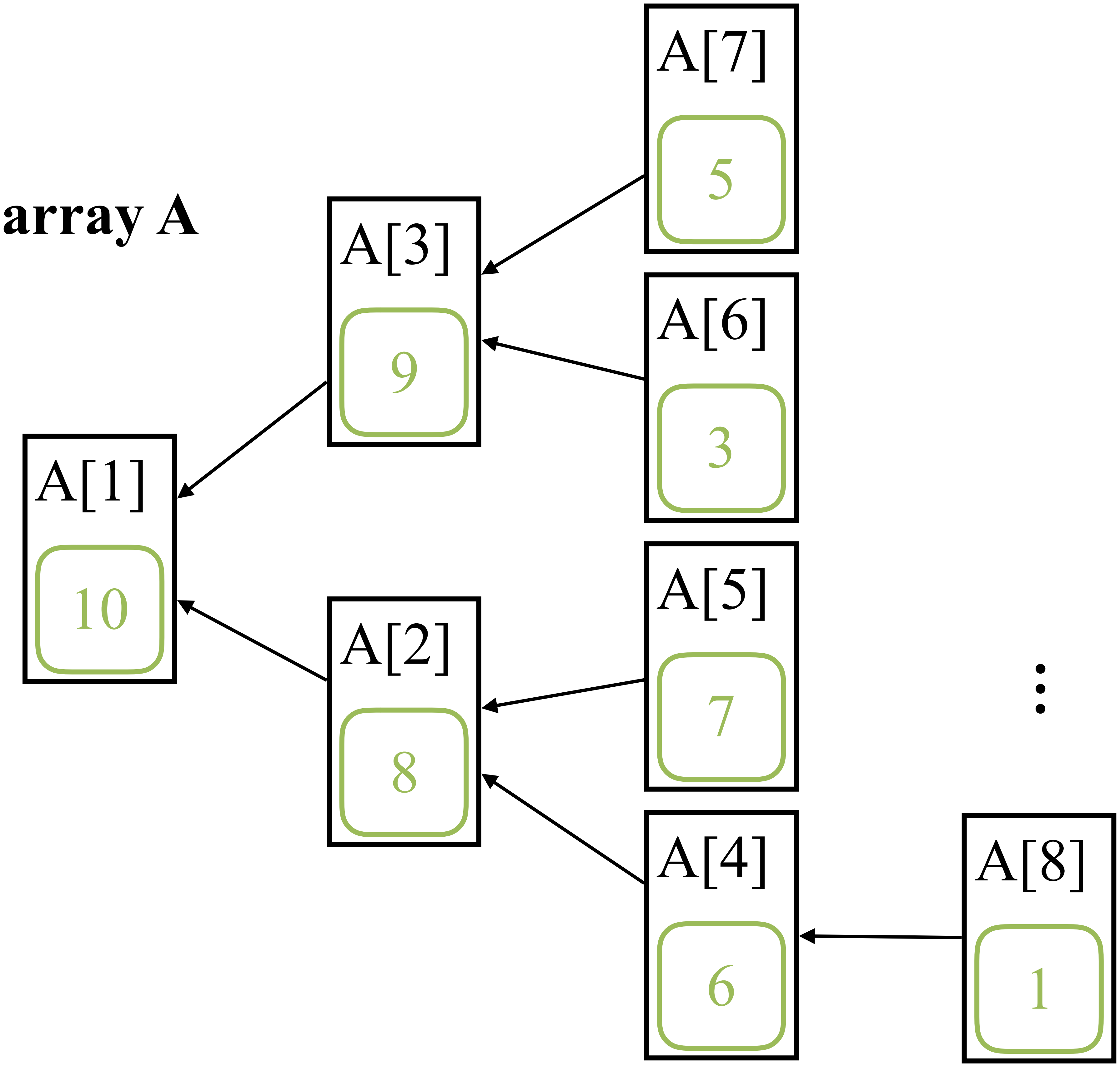
Example.

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
array A	10	8	9	6	7	3	5	1

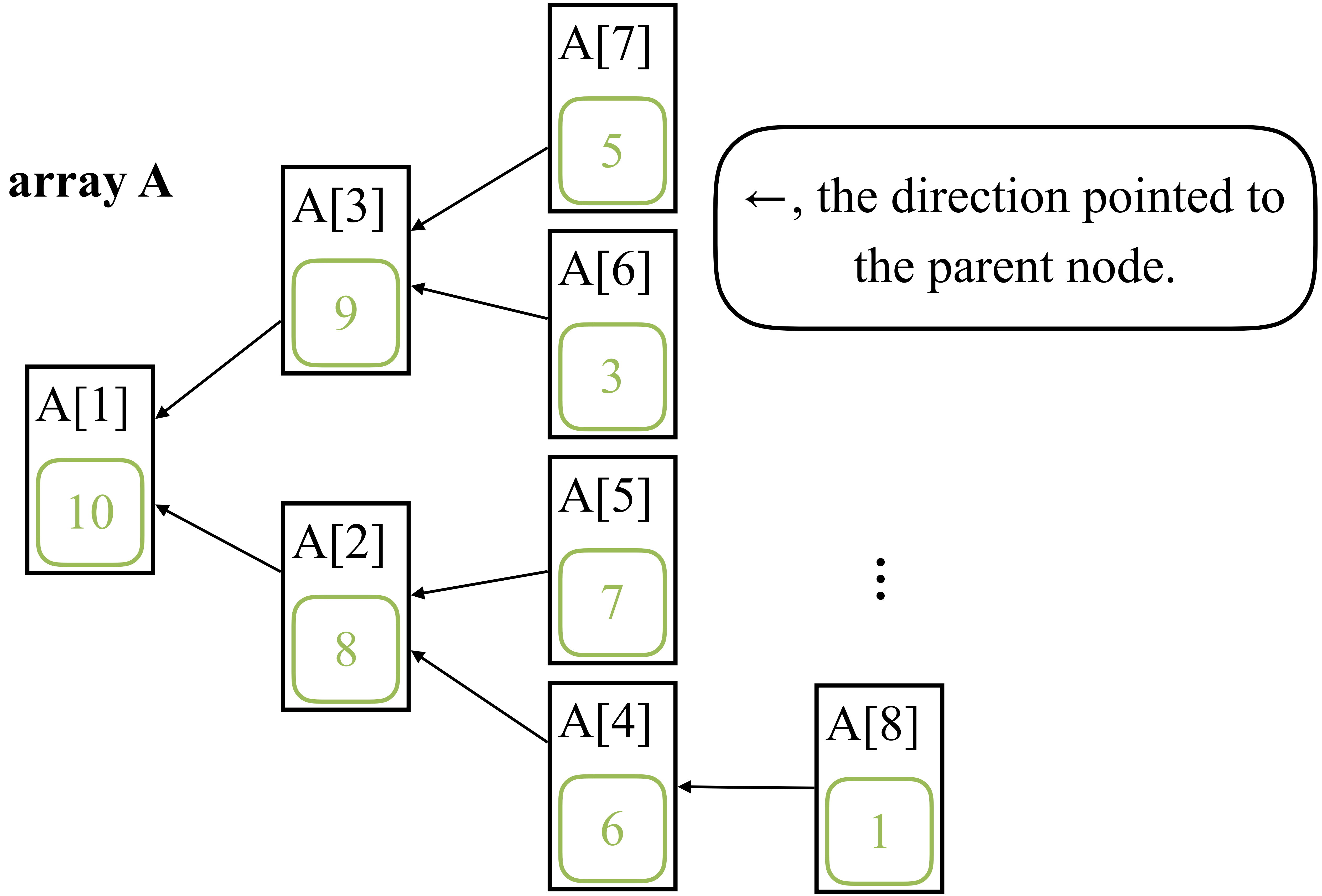
View heaps as binary trees

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
array A	10	8	9	6	7	3	5	1

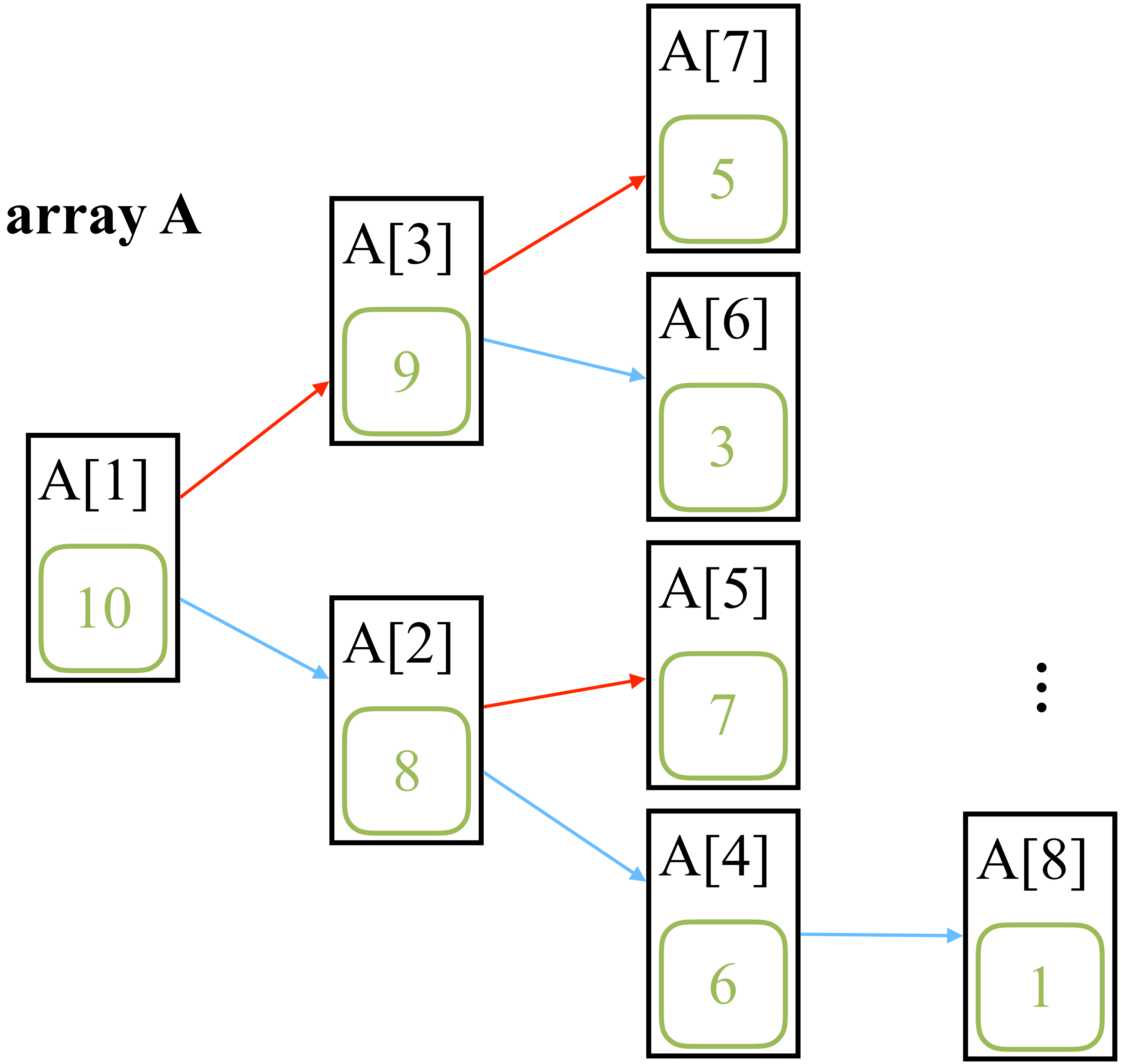
View heaps as binary trees



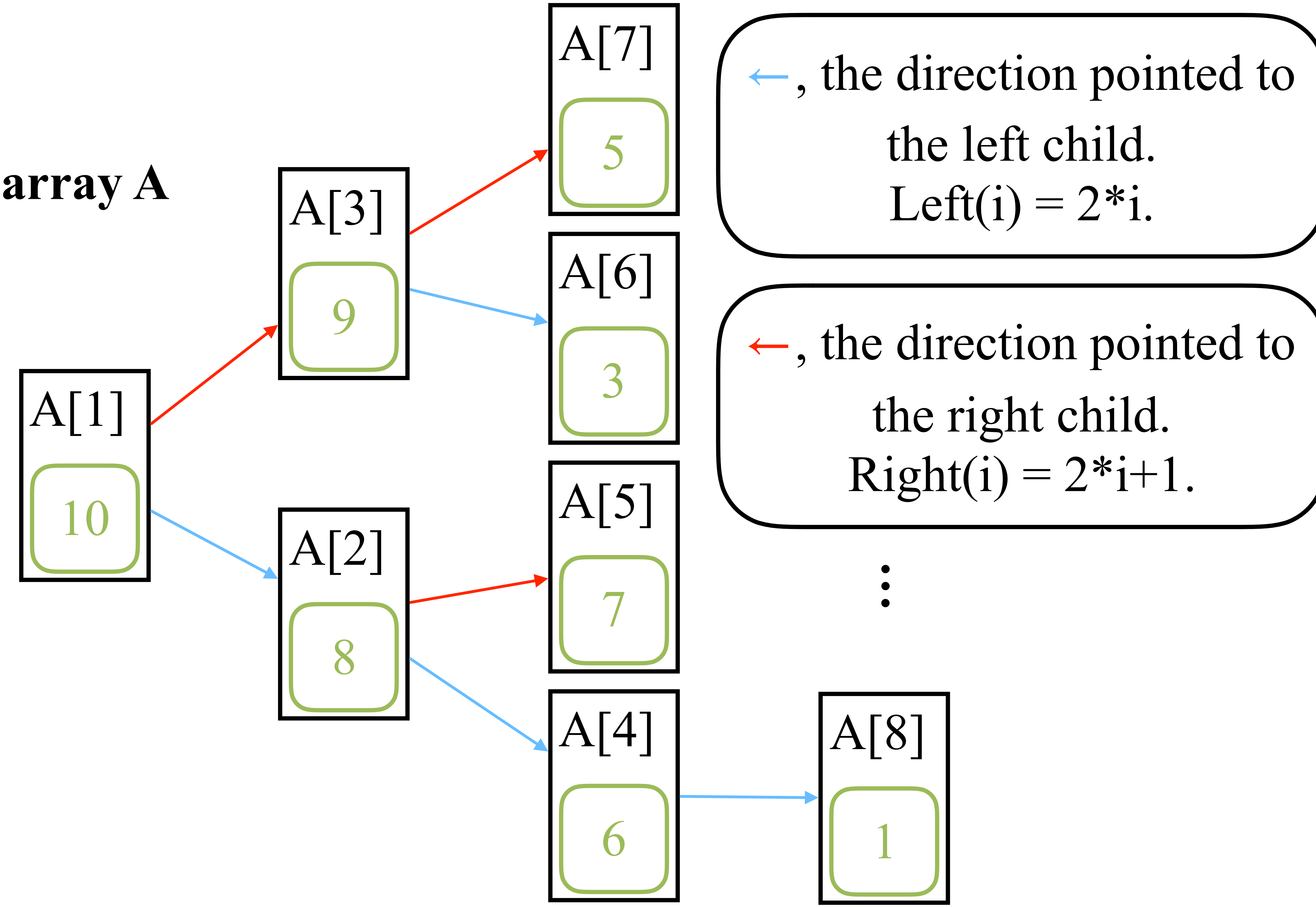
View heaps as binary trees



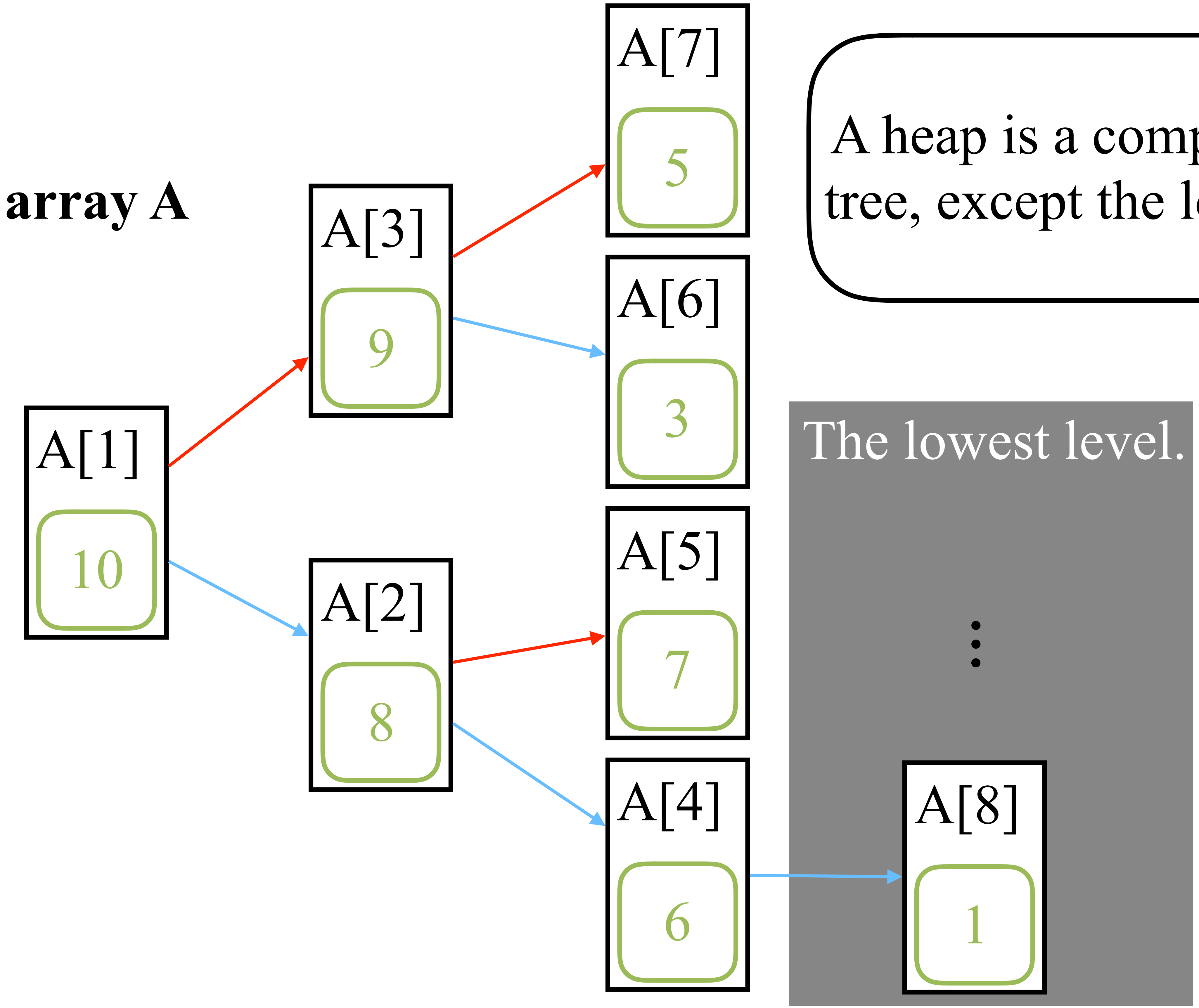
View heaps as binary trees



View heaps as binary trees

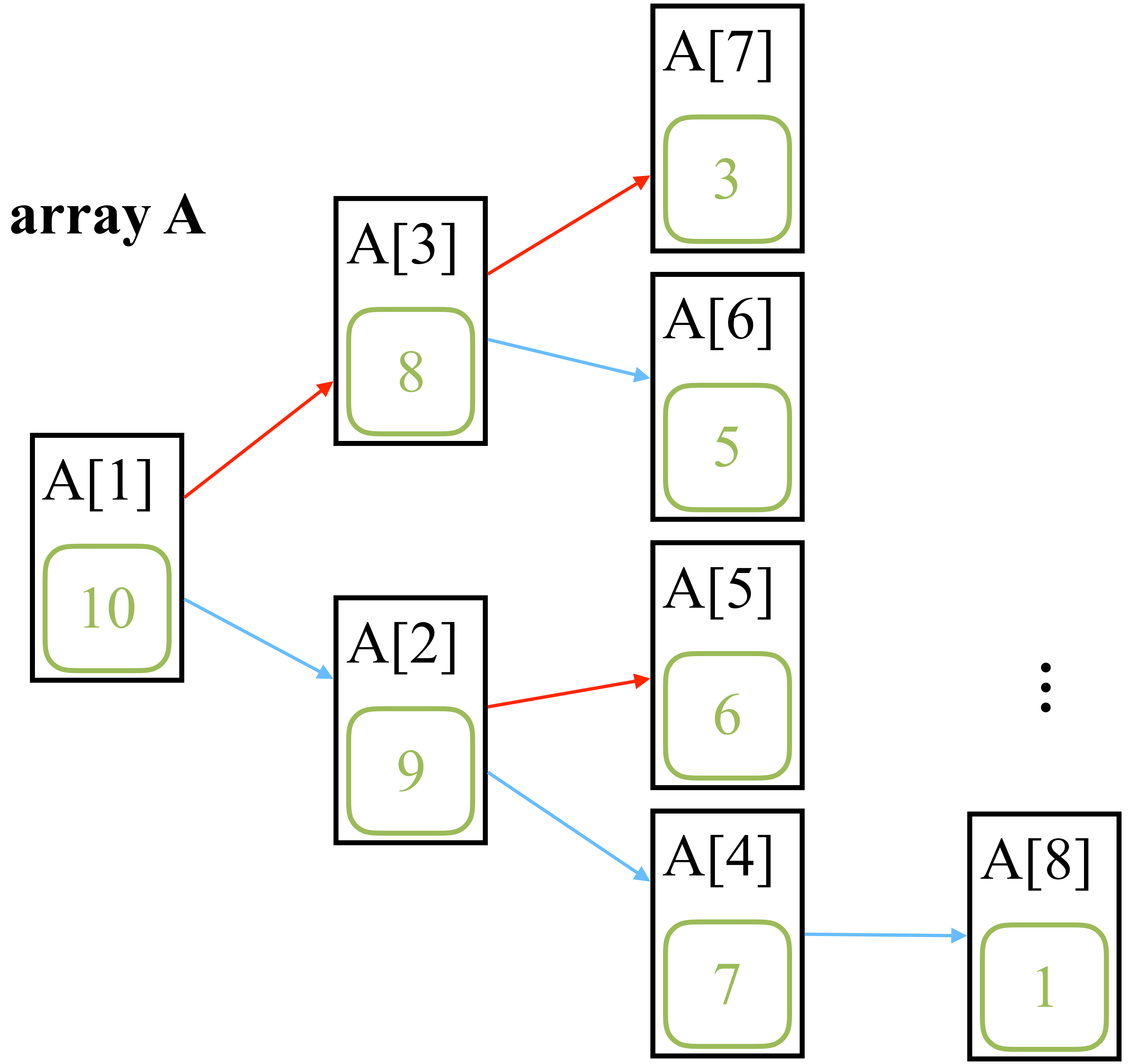


View heaps as binary trees

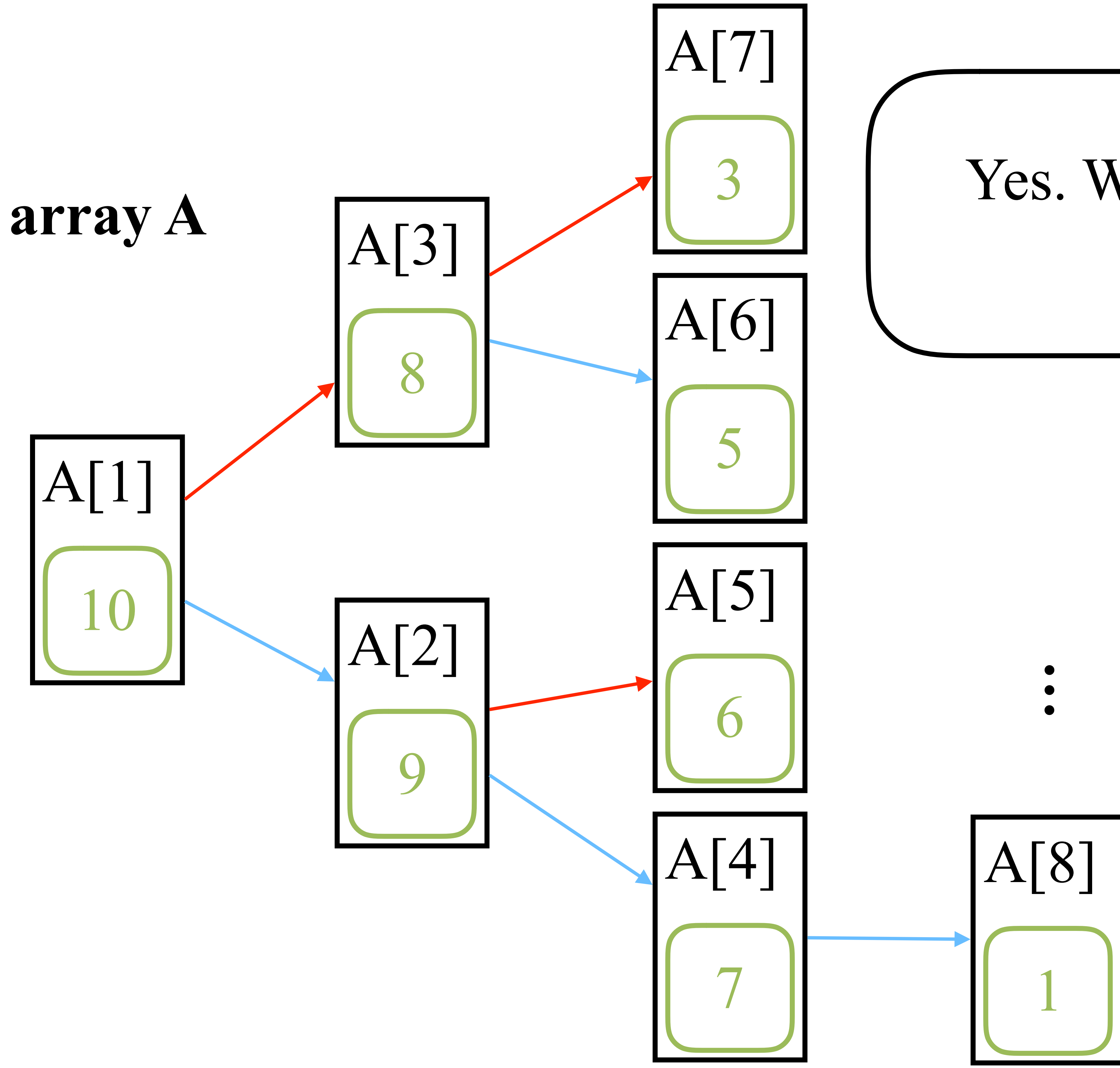


A heap is a complete binary tree, except the lowest level.

Is sorted array a heap?



Is sorted array a heap?



Yes. Why do we need heaps?

Construction Time

Given n elements, constructing a heap of the n elements needs $O(n)$ time.

However, sorting the n elements requires $\Omega(n \log n)$ time.

Construction Time

Given n elements, constructing a heap of the n elements needs O(n) time.

However, sorting the n elements requires Ω(n log n) time.

That is why heaps are not subsumed
by sorted arrays.

Heapification

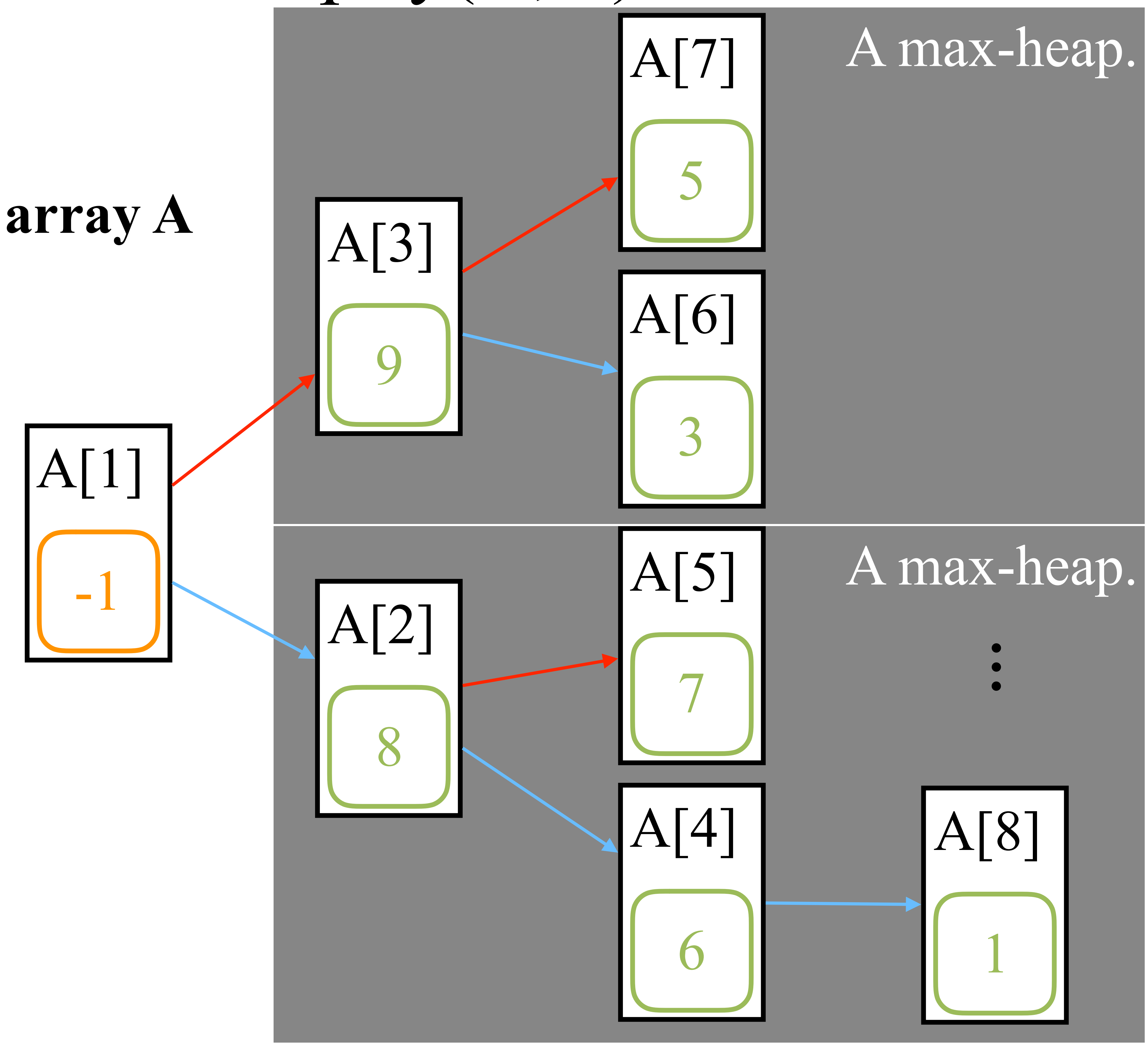
Max-Heapify(A, i)

convert the subtree rooted at node i into a max heap assuming that

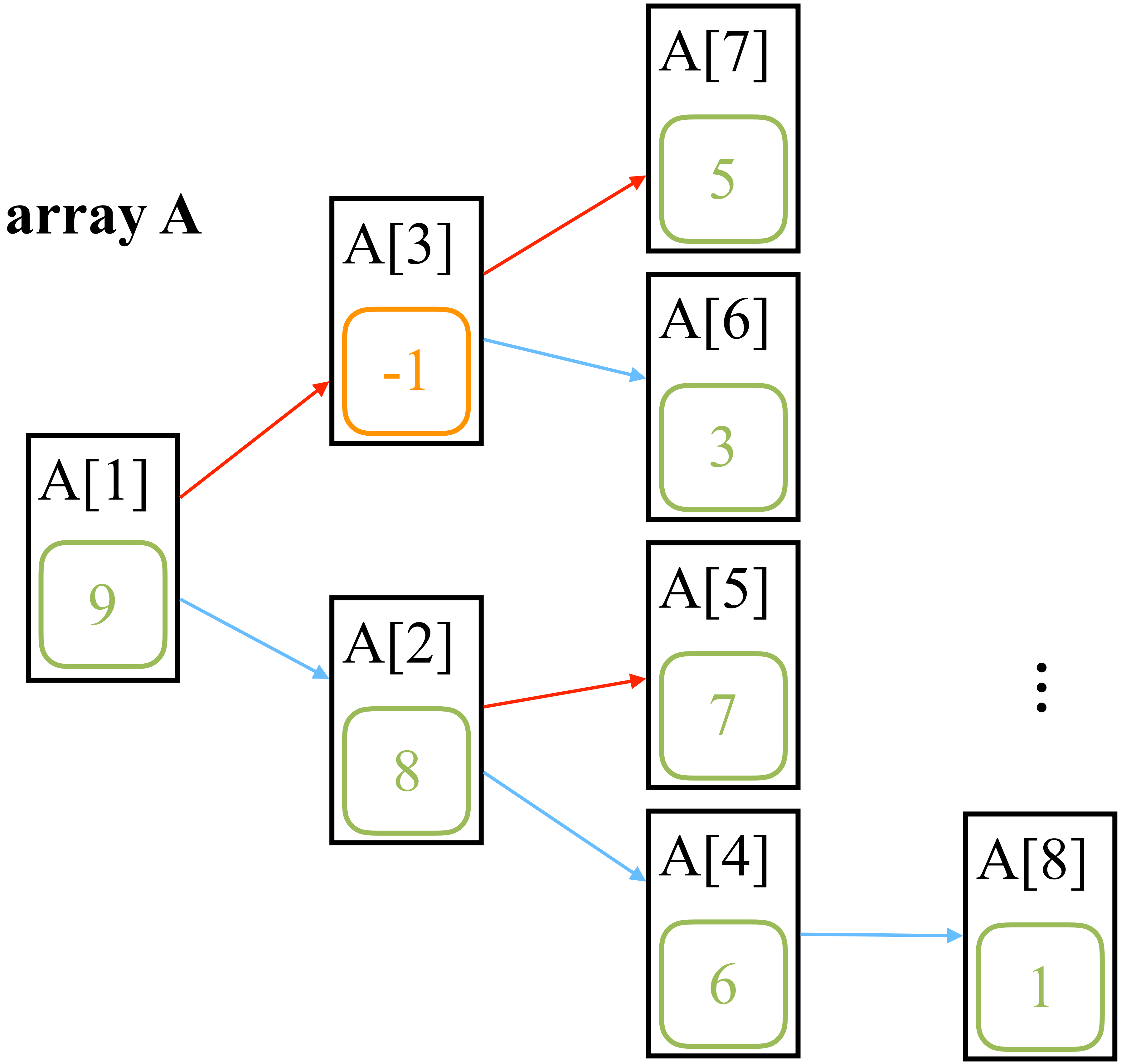
(1) the subtree rooted at node Left(i) is a max heap, and

(2) the subtree rooted at node Right(i) is a max heap.

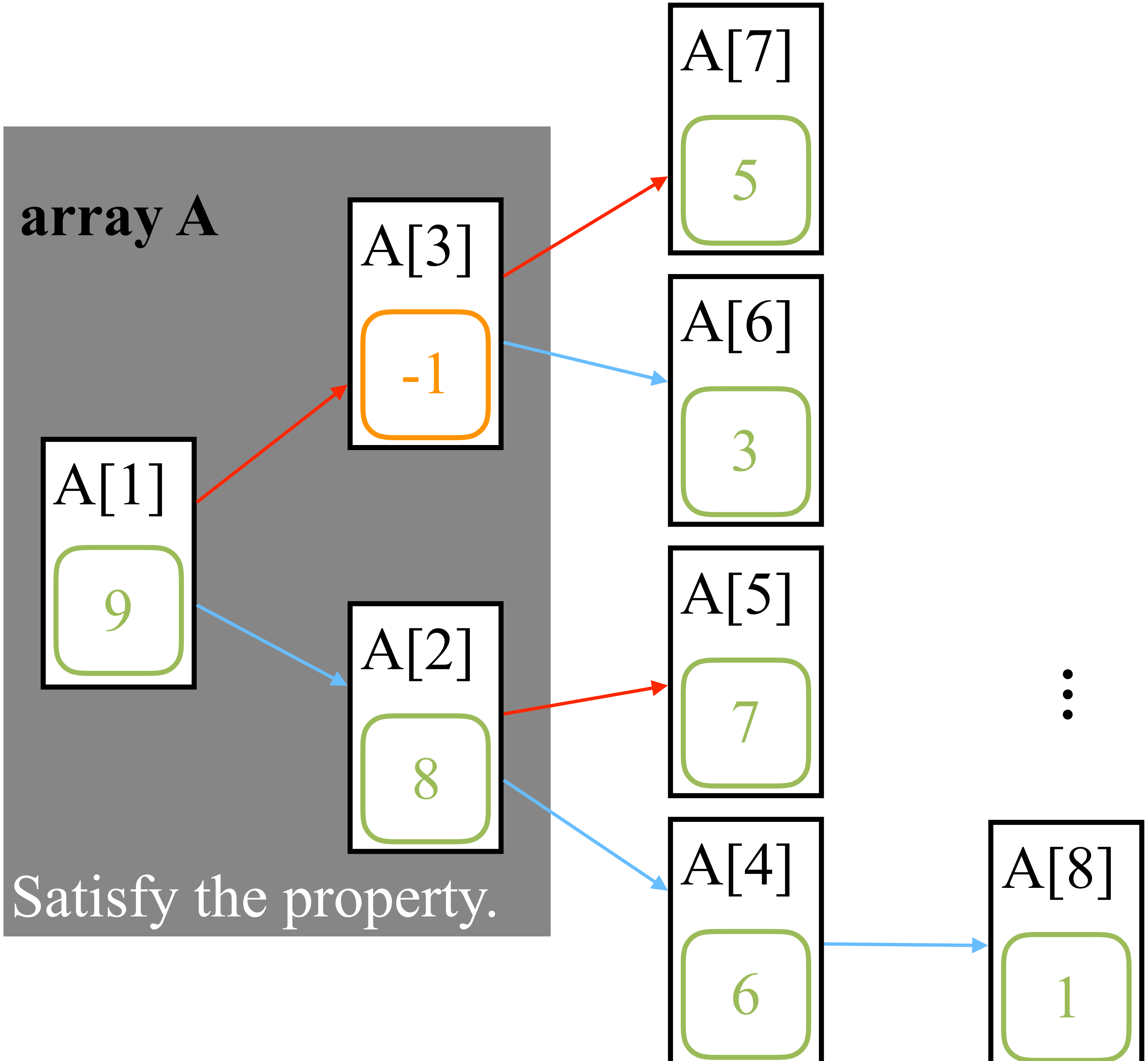
Max-Heapify(A, 1)



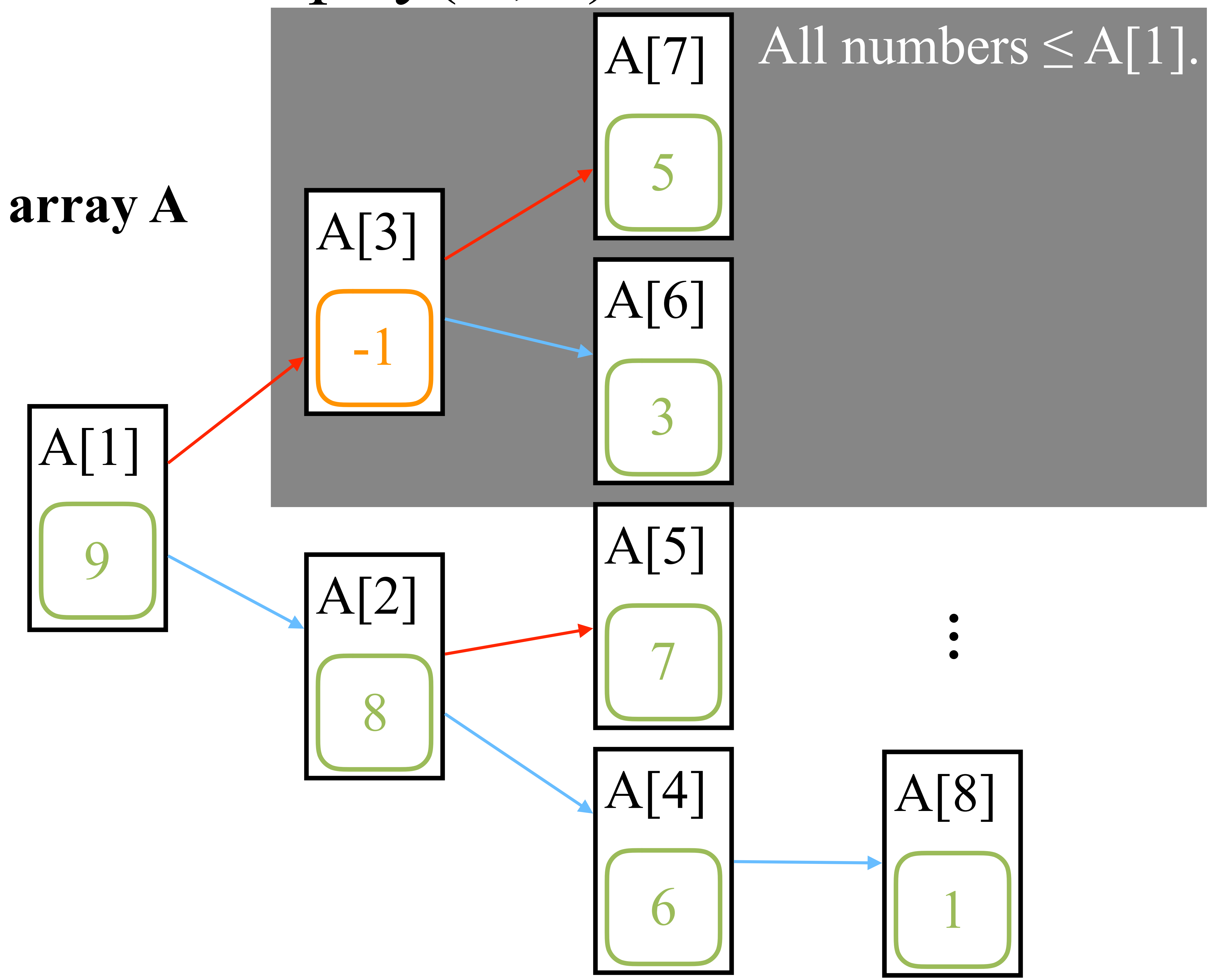
Max-Heapify(A, 1)



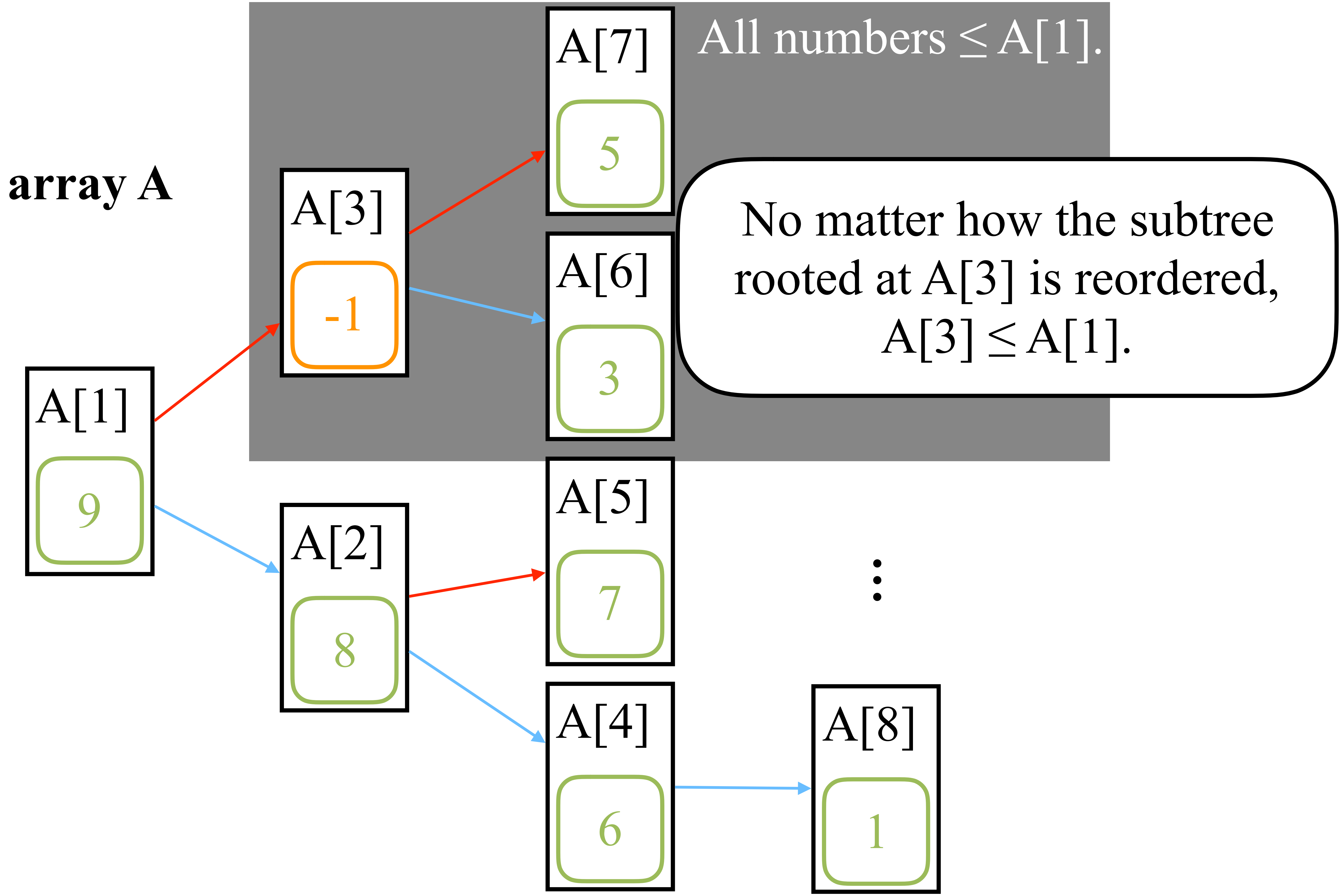
Max-Heapify(A, 1)



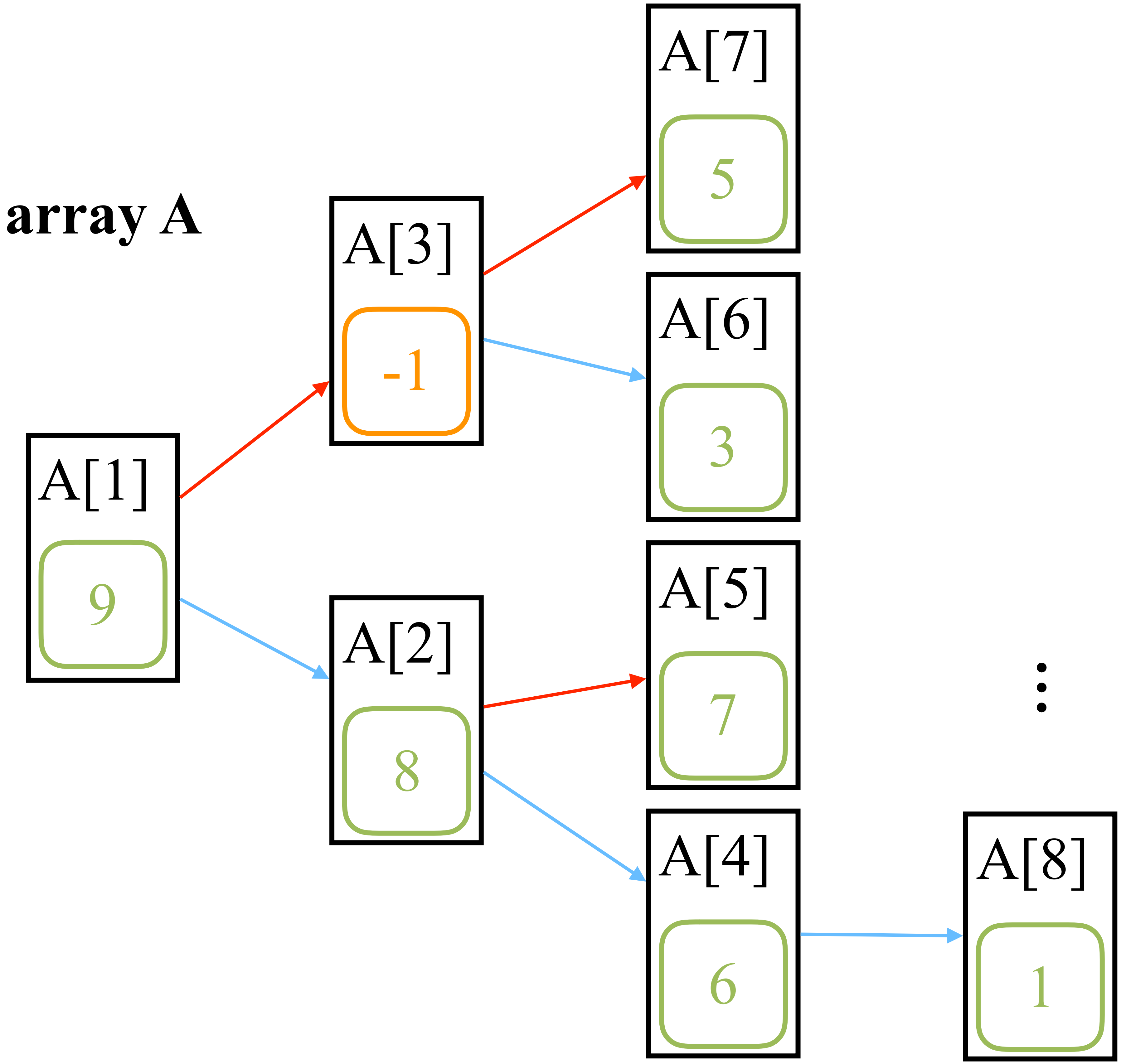
Max-Heapify(A, 1)



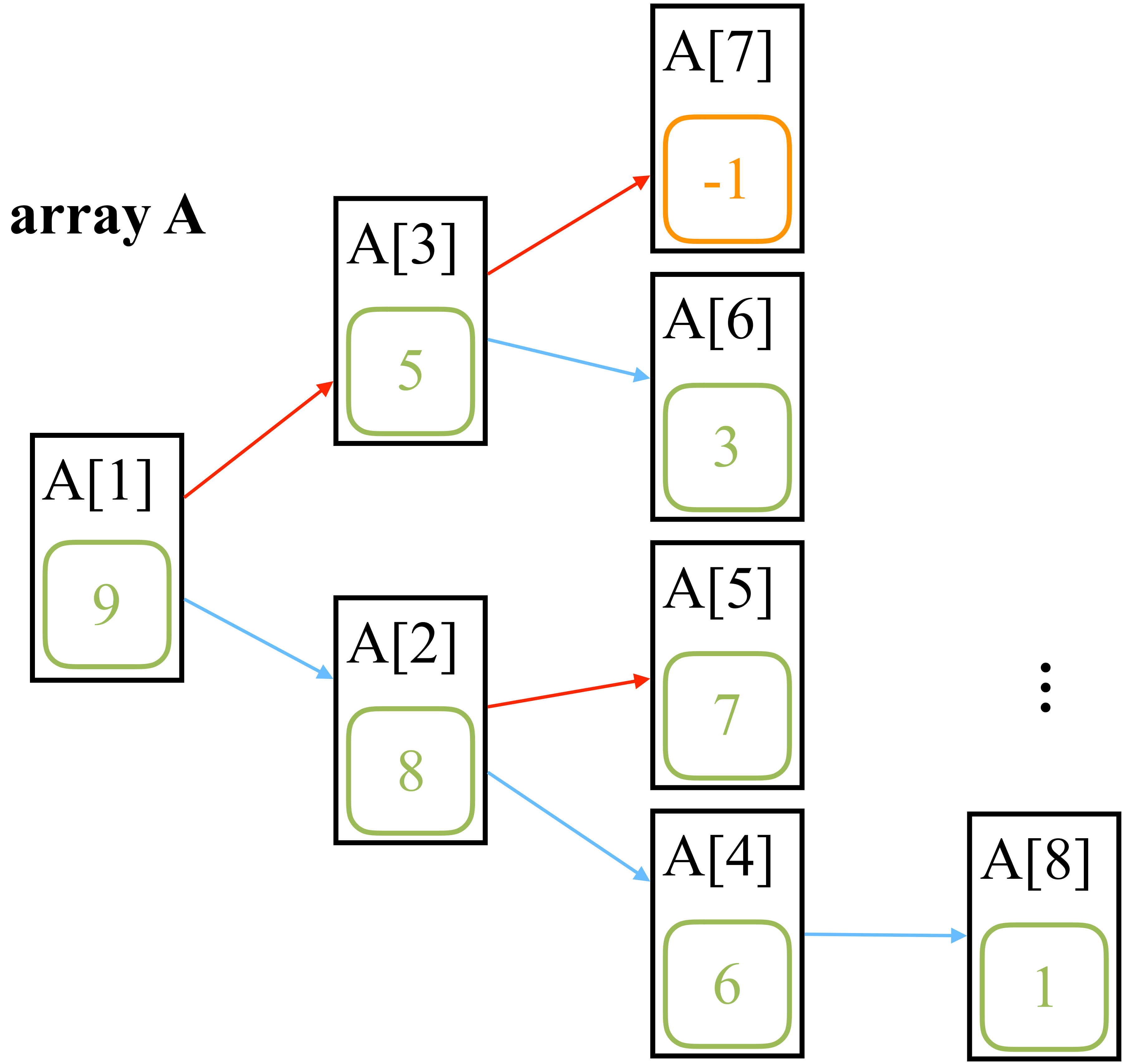
Max-Heapify(A, 1)



Max-Heapify(A, 3)

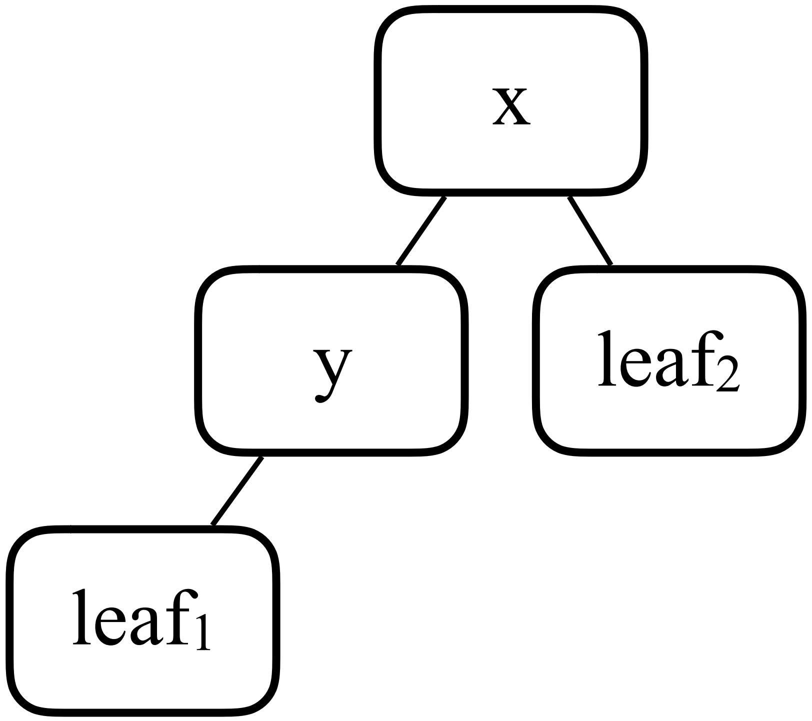


Max-Heapify(A, 3)



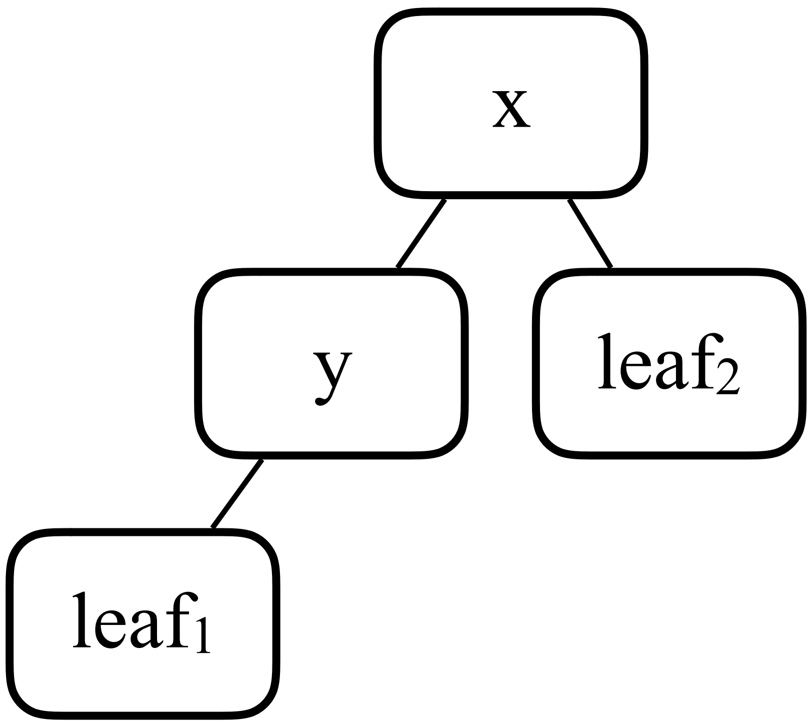
Height

Define *height* of a node *v* in a tree to be the number of edges on a longest simple path from *v* to its descendant.



Height

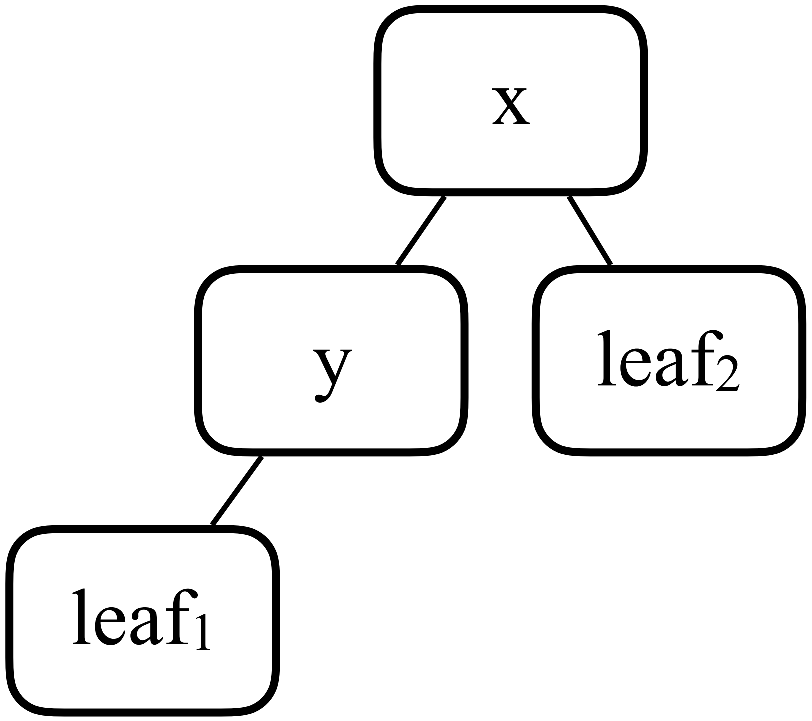
Define *height* of a node v in a tree to be the number of edges on a longest simple path from v to its descendant.



$$\text{height}(x) = 2,$$
$$\text{height}(y) = 1,$$
$$\text{height}(\text{leaf}_1) = 0,$$
$$\text{height}(\text{leaf}_2) = 0.$$

Height

Define *height* of a node v in a tree to be the number of edges on a longest simple path from v to its descendant.



height(x) = 2,
height(y) = 1,
height($leaf_1$) = 0,
height($leaf_2$) = 0.

Heapify(A, i) takes O(height(i)) time, and height(i) = O(log n). (Why?)

Build a Heap

Build-Map-Heap(A, i)

Convert the subtree rooted at node i into a max heap without the assumption that heapification uses.

Build a Heap

```
Build-Map-Heap(A, i){

    Build-Max-Heap(A, Left(i));
    Build-Max-Heap(A, Right(i));

    Max-Heapify(A, i);
}
```

--- Runtime ---

One may guess that $T(n) = 2T(n/2) + O(\log n)$.

By Master Theorem, the guess implies that $T(n) = O(n)$.

Build a Heap

```
Build-Map-Heap(A, i){

    Build-Max-Heap(A, Left(i));
    Build-Max-Heap(A, Right(i));

    Max-Heapify(A, i);
}
```

--- Runtime ---

One may guess that $T(n) = 2T(n/2) + O(\log n)$.

By Master Theorem, the guess implies that $T(n) = O(n)$.

It is simply a guess rather than a proof because the above algorithm does not necessarily split a problem into two subproblems evenly.

A (More) Rigorous Proof

Observe that $T(n) \leq T(n+1)$ for every n. Thus

$$T(n) \leq T(n')$$

where n' is the smallest power of 2 no less than n.

We can write $T(n) \leq T(n') = 2T(n'/2) + O(\log n')$.

By Master Theorem, we have $T(n) = O(n')$.

Because $n' < 2n$, $O(n') = O(n)$.

Summary

Build a heap of n elements needs O(n) time, which is asymptotically faster than sorting n elements.

Build a Young Tableau of n elements (naively) needs O(n log n) time, which is no faster than sorting n elements.

Summary

Build a heap of n elements needs O(n) time, which is asymptotically faster than sorting n elements.

Build a Young Tableau of n elements (naively) needs O(n log n) time, which is no faster than sorting n elements.

Why sorted arrays do not subsume Young Tableau?

Extract-Max

Extract-Max(A, n)

Remove the maximum from an n-element array A and keep the rest of A as a max heap.

Extract-Max

```
Extract-Max(A, int& n){
    int ret = A[1]; A[1] = ∞;
    Max-Heapify(A, 1);
    n --;
   return ret;
}
```

--- Runtime ---

O(log n).

HeapSort

Input: an array A of n integers.

Output: the same array with the n integers ordered nondecrementally.

```
HeapSort(A, n){
    while(n > 1){
        k = Extract-Max(A, n);
        A[n+1] = k;
    }
}
```

--- Runtime ---

There are O(n) loop iterations, and each needs O(log n) time. In total, the running time is O(n log n), which is asymptotically optimal in the comparison-based model.

Exercise

Input: a heap A of n elements and a newly-added element x.

Output: a heap containing A and x.

Can you solve it in O(log n) time?

Summary

n elements	search cost	insertion cost	extract-max
sorted array	O(log n)	O(n)	O(n)
Young tableau	$O(n^{1/2})$	$O(n^{1/2})$	?
unsorted array	O(n)	O(1)	O(n)
heap	?	O(log n)	O(log n)

Exercise

Input: k sorted arrays of length $n_1, n_2, \ldots, n_k$.

Output: a single sorted array.

Can you solve it in $O((n_1 + n_2 + \ldots + n_k) \log k)$ time?

Exercise

Analyze the runtime of each operation for d-ary heaps with d > 2. Note that d can be a superconstant, so you cannot simply copy the runtime of each operation for binary heaps.

n elements	cost
Max-Heapify	
Build-Max-Heap	
HeapSort	

Outside the Comprison-Based Model

SleepSort (Engineer Humor)

Let the unit time U be 1 second.

```
SleepSort(A, n){

    for(i in [1, n]){
        fork a child thread to do{
            (1) sleep A[i]*U seconds;
            (2) print A[i];
        }
    }
}
```

SleepSort (Engineer Humor)

Let the unit time U be 1 second.

```
SleepSort(A, n){

    for(i in [1, n]){
        fork a child thread to do{
            (1) sleep A[i]*U seconds;
            (2) print A[i];
        }
    }
}
```

U can be very small, so SleepSort breaks the $\Omega(n \log n)$ sorting lower bound?