

Content:

- Priority Queue. Possible representations
- Binary Heap, Heap-sort



Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2022 - 2023



Source: <https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494>

Priority Queue - Representation

Think about:

Efficient representation for Priority Queue
over Dynamic Array or a Linked List

- How do we store the elements ?
- What are the complexities of operations?

Priority Queue - Representation

- If the representation is a Dynamic Array or a Linked List we have to decide how we store the elements in the array/list:
 - we can keep the elements ordered by their priorities
 - Where would you put the element with the highest priority?
 - we can keep the elements in the order in which they were inserted
- Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

Binary Heap

- A binary heap is a data structure that can be used as an efficient representation for Priority Queues.

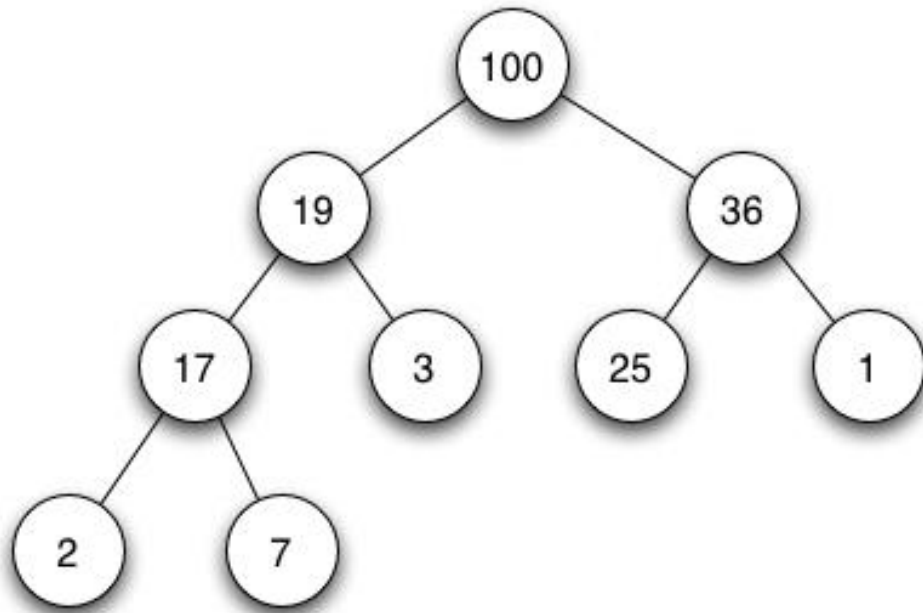
If we use the \geq relation, we will have a MAX-HEAP.

(default for us, if nothing else is specified)

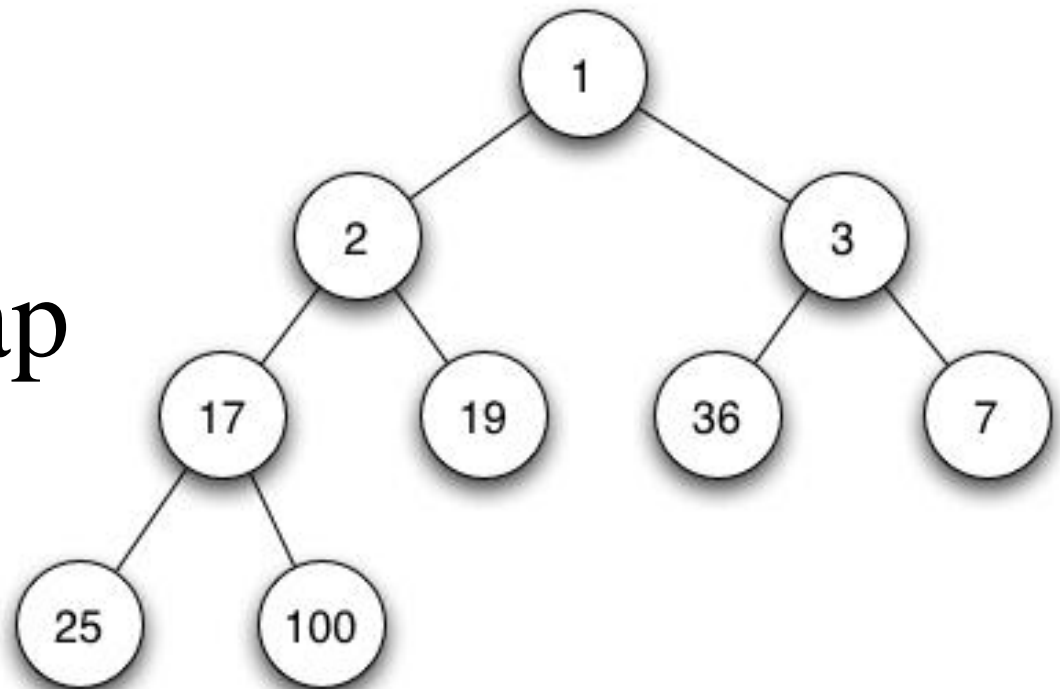
If we use the \leq relation, we will have a MIN-HEAP.

- The elements of a binary heap are usually stored in a (dynamic) array, but the array is visualized as a binary tree.
 - is kind of a hybrid between a dynamic array and a binary tree

Max binary heap

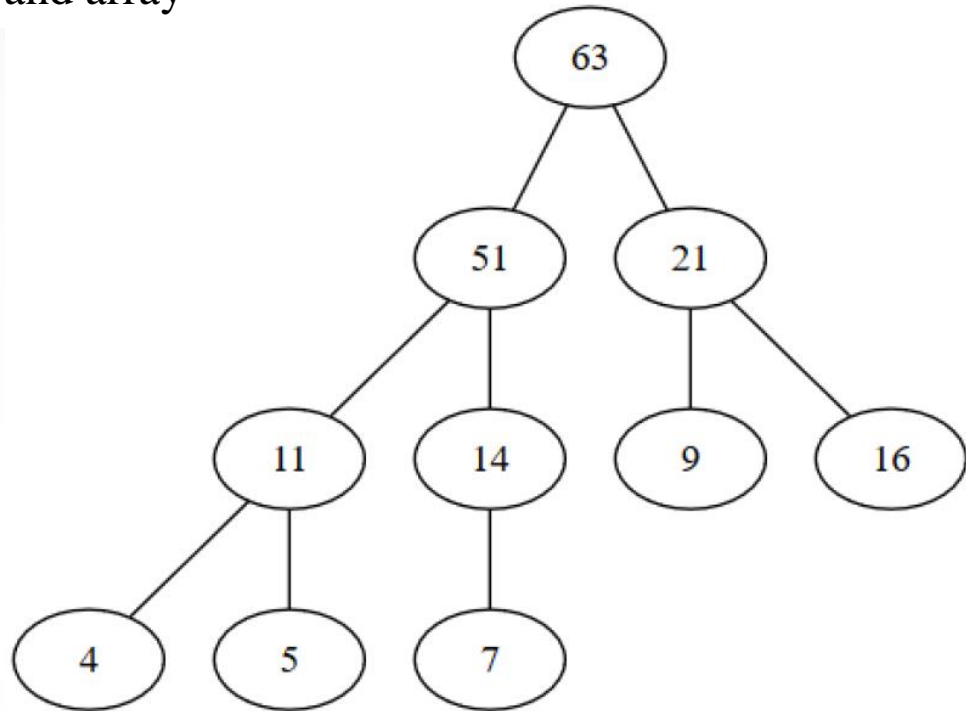


Min binary heap



Binary Heap

e.g.: MAX Heap as binary tree and array



63	51	21	11	14	9	16	4	5	7
----	----	----	----	----	---	----	---	---	---


Binary Heap

A **binary heap** is an array that can be visualized as a binary tree having a heap structure and a heap property.

- **Heap structure:** in the binary tree every node has exactly 2 children, except for the last two levels, where children are filled from left to right.
- **Heap property:**
(for MAX-Heap, when using \geq relation)
the value in a node of the tree is higher than the values in its children

Heap property is defined based on relation used to get a MIN or MAX binary heap

- Specific operations:
 - add a new element in the heap
 - remove root



operation remove, by default

- The relation between a node and both its descendants can be.

Binary Heap

- We can visualize this array as a binary tree, where the root is the first element of the array, its children are the next two elements, from left to right, and so on.

How do we go from 1-based array positions to positions in the tree?

- the elements of the array are: $a_1, a_2, a_3, \dots, a_n$
- a_1 is the root of the heap
- for a_i (element from position i),
its children are on positions $2*i$ and $2*i+1$
(if the children exist)
- the parent of the element is on position $\left\lfloor \frac{i}{2} \right\rfloor$ (floor of $i/2$)
(if the parent exists) (integer part of)

How do we go from 0-based array to positions in the tree? [...]

Binary Heap for Priority Queue

A heap can be used as representation for a Priority Queue and it has two specific operations:

- add a new element in the heap.
- remove

we always remove the root of the heap (by default)

in such a way that we keep both the heap structure and the heap property

Heap:

cap: Integer
len: Integer
elems: TElem[]

Subalg. add(heap, elem)

- bubble-up

Function remove(heap) => elem

- bubble-down

Binary Heap

Heap:

cap: Integer

len: Integer

elems: TElem[]

subalgorithm add(heap, e) **is:**

//heap - a heap

//e - the element to be added

if heap.len = heap.cap **then**

 @ resize

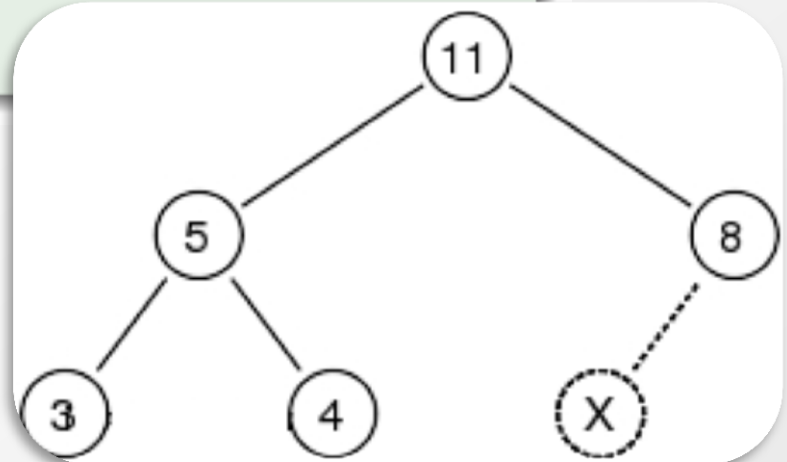
end-if

heap.elems[heap.len+1] \leftarrow e

heap.len \leftarrow heap.len + 1

bubble-up(heap, heap.len)

end-subalgorithm



Binary Heap

Heap:

cap: Integer

len: Integer

elems: TElem[]

```
subalgorithm bubble-up (heap, p) is:  
  //heap - a heap  
  //p - position from which we bubble the new node up  
  poz ← p  
  elem ← heap.elems[p]  
  parent ← p / 2  
  while poz > 1 and elem > heap.elems[parent] execute  
    //move parent down  
    heap.elems[poz] ← heap.elems[parent]  
    poz ← parent  
    parent ← poz / 2  
  end-while  
  heap.elems[poz] ← elem  
end-subalgorithm
```

Complexity: $\mathbf{O}(\log n)$

BC, WC ?

Binary Heap

Heap:

cap: Integer

len: Integer

elems: TElem[]

function remove(heap) **is:**

//heap - is a heap

if heap.len = 0 **then**

 @ error - empty heap

end-if

deletedElem \leftarrow heap.elems[1]

heap.elems[1] \leftarrow heap.elems[heap.len]

heap.len \leftarrow heap.len - 1

bubble-down(heap, 1)

remove \leftarrow deletedElem

end-function

Binary Heap

Heap:

cap: Integer
len: Integer
elems: TElem[]

subalgorithm bubble-down(heap, p) is:
//heap - is a heap
//p - position from which we move down the element

```
    poz ← p
    elem ← heap.elems[p]
    while poz < heap.len execute
        maxChild ← -1
        if poz * 2 ≤ heap.len then
            //it has a left child, assume it is the maximum
            maxChild ← poz*2
        end-if
        if poz*2+1 ≤ heap.len and heap.elems[2*poz+1] > heap.elems[2*poz] then
            //it has two children and the right is greater
            maxChild ← poz*2 + 1
        end-if
        if maxChild ≠ -1 and heap.elems[maxChild] > elem then
            tmp ← heap.elems[poz]
            heap.elems[poz] ← heap.elems[maxChild]
            heap.elems[maxChild] ← tmp
            poz ← maxChild
        else
            poz ← heap.len + 1
            //to stop the while loop
        end-if
    end-while
end-subalgorithm
```

Complexity: **O**(log n)

BC, WC ?

Binary Heap

Think about:

- Consider add operation. Describe a possible situation for getting:
 - BC complexity
 - WC complexity
 - Consider remove operation. Describe a possible situation for:
 - BC complexity
 - WC complexity
-
- Complexity for Priority Queue operations
push, pop, top, isEmpty, init

Think about:

- Consider an initially empty Binary MAX-HEAP and insert the elements 8, 27, 13, 15*, 32, 20* in it. Draw the heap in the tree form after the insertion of the elements marked with a * (2 drawings).
 - Remove 2 elements from the heap and draw the tree form after every removal (2 drawings).
-
- Insert the following elements, in this order, into an initially empty MIN-HEAP: 15, 17, 9, 11. Draw the heap after every second operation (after adding 17, etc.)
 - Remove all the elements, one by one, in order from the resulting MIN HEAP.

Binary Heap

Think about:

- Complexity for searching for an element in a binary heap.
 - Complexity for verifying for an array to be a heap.
 - Complexity for deleting an element based on its position in the array.
-
- The position of the last non-leaf node of a heap of size n
-
- Get the element representing the k -th largest element in a container in $O(n \log k)$.
 - Get the element representing the k -th largest element in an array in $O(n + k \log k)$.

HeapSort

- Build a heap
- Remove elements from the heap:
 - they will be removed in the sorted order.
 - Put the elements back into the list.

Complexity:
 $O(n \log n)$

Do we need an extra array?

HeapSort

(V1)

Input: any container

Build the heap by successive insertions.

- Use an extra array for heap
- Extract one by one the elements and add them to the heap

Time complexity of the algorithm is $O(n \log n)$

The extra space complexity of the algorithm is $\Theta(n)$

HeapSort

(V2)

Input: an array

Build the heap:

transform the array into a heap.

- the second half of the array will contain leaves:
they are left where they are.
- Starting from the first non-leaf element (and going towards the beginning of the array), we will just call bubble-down for every element.
- Time complexity of this approach: $O(n)$
(but removing the elements from the heap is still $O(n \log n)$)

Build heap

Build heap - complexity

- obvious: complexity $\in O(n \cdot \log(n))$
- not obvious, but proved: complexity $\in O(n)$

Proof ideas:

- nodes of height h :
- complexity
(no. of steps)

$$nrNodes_h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$$\begin{aligned} \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &= O \left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \\ &\leq O \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \end{aligned}$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

HeapSort – complexity overview

HeapSort for an array

without an extra-array

- build a MAX-heap $\Rightarrow O(n)$
- repeatedly extract maximum $\Rightarrow n * O(\log(n))$
and put it in its final position

$\Rightarrow O(n * \log(n))$ (even in the worse case)

What about the complexity in best case ?

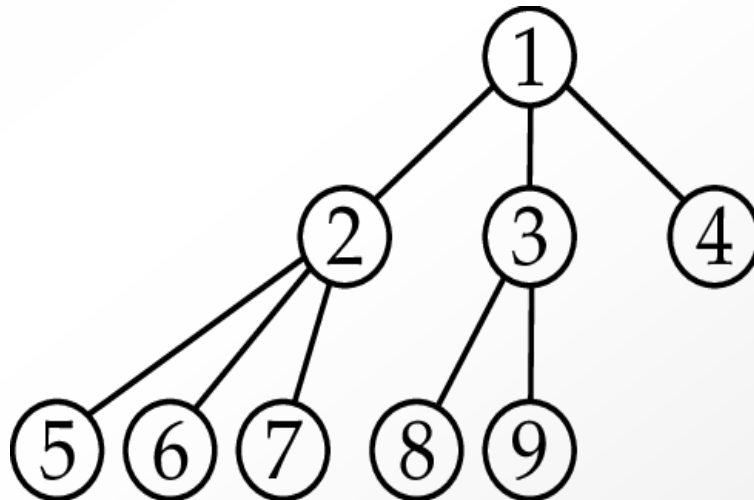
Think about:

Problem taken from <https://open.kattis.com/problems/chewbacca>

Accessed: 2022

Chewbacca

You are given a tree of out-degree with nodes or, in other words, each node can have at most children. The tree is constructed so it is of the “lowest energy”: the nodes are placed in a new depth of the tree only when all the places (from left to right) in the previous depth have been filled. This is also the order of enumerating the nodes, starting with 1.



Source Croatian Open
Competition in Informatics
2015/2016, contest #4

Chewbacca

Given two nodes, x and y ($x \neq y$, both valid numbers in the tree) determine the minimum number of steps to get from node x to y .

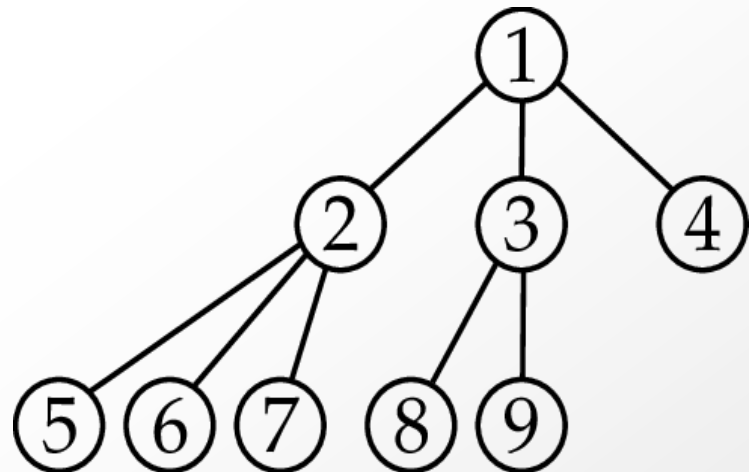
Example:

$$\text{minDist}(5, 7) = 2$$

$$\text{minDist}(7, 2) = 1$$

$$\text{minDist}(5, 9) = \dots$$

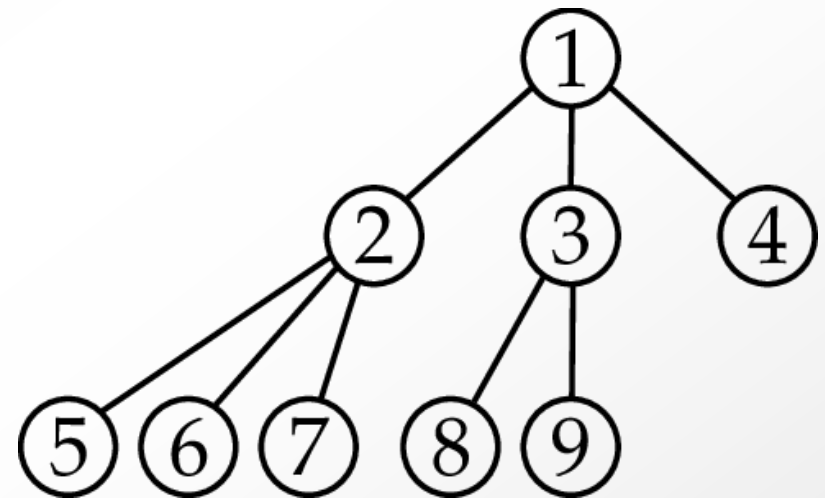
$$\text{minDist}(5, 3) = \dots$$



Chewbacca

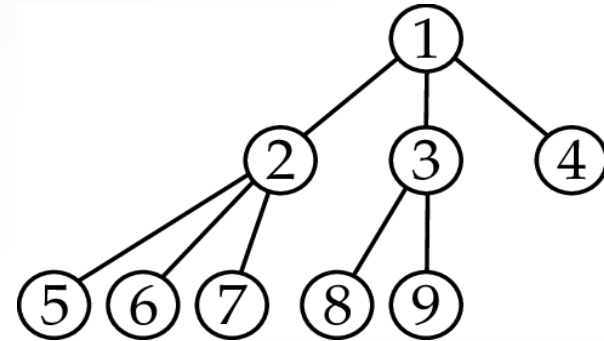
Find the problem in the next subalgorithm idea:

1. **while** $x \neq y$ **execute**:
2. $x \leftarrow \text{parent of } x$
3. $y \leftarrow \text{parent of } y$
4. $\text{steps} \leftarrow \text{steps} + 2$
5. **end-while**



How can we go from a node to its parent?
Do we have a formula?

Chewbacca



```
function minDist(N, K, x, y) is:  
    distance  $\leftarrow$  0  
    while  $x \neq y$  execute:  
        if  $x > y$  then  
             $x \leftarrow (x + K - 2) / K$   
            distance  $\leftarrow$  distance + 1  
        else  
             $y \leftarrow (y + K - 2) / K$   
            distance  $\leftarrow$  distance + 1  
        end-if  
    end-while  
    minDist  $\leftarrow$  distance  
end-function
```

K-ary Heap

- K-ary heaps are a generalization of binary heap ($K=2$) in which each node have K children instead of 2
 - **Heap structure:** every node has exactly k children, except for the last two levels, where children are filled from left to right.
 - **Heap property:**
(for MAX-Heap, when using \geq relation)
the value in a node of the tree is higher than the values in its children
- We can determine the relation between parent - child positions
e.g.: Assuming 0-based indexing of array:
 - Parent of the node at index i (except root node) is located at index $(i-1)/k$
 - Children of the node at index i are at indices $(k*i)+1$, $(k*i)+2$ $(k*i)+k$

Comparison sort

comparison operation : \Leftarrow properties of a total order

1. transitivity : if $a \leq b$ and $b \leq c$ then $a \leq c$
2. total relation : for all a and b , either $a \leq b$ or $b \leq a$

A **comparison sort** is a type of sorting algorithm

Input:

- elements
- a comparison operation
 - determines which of two elements should occur first in the final sorted list*
 - (often a "less than or equal to" operator)*

Output

- list of elements
 - order determined by comparison operation

Possible: $a \leq b$ and $b \leq a$; in this case either may come first in the sorted list.

Stable sort: equal elements ($a \leq b$ and $b \leq a$)

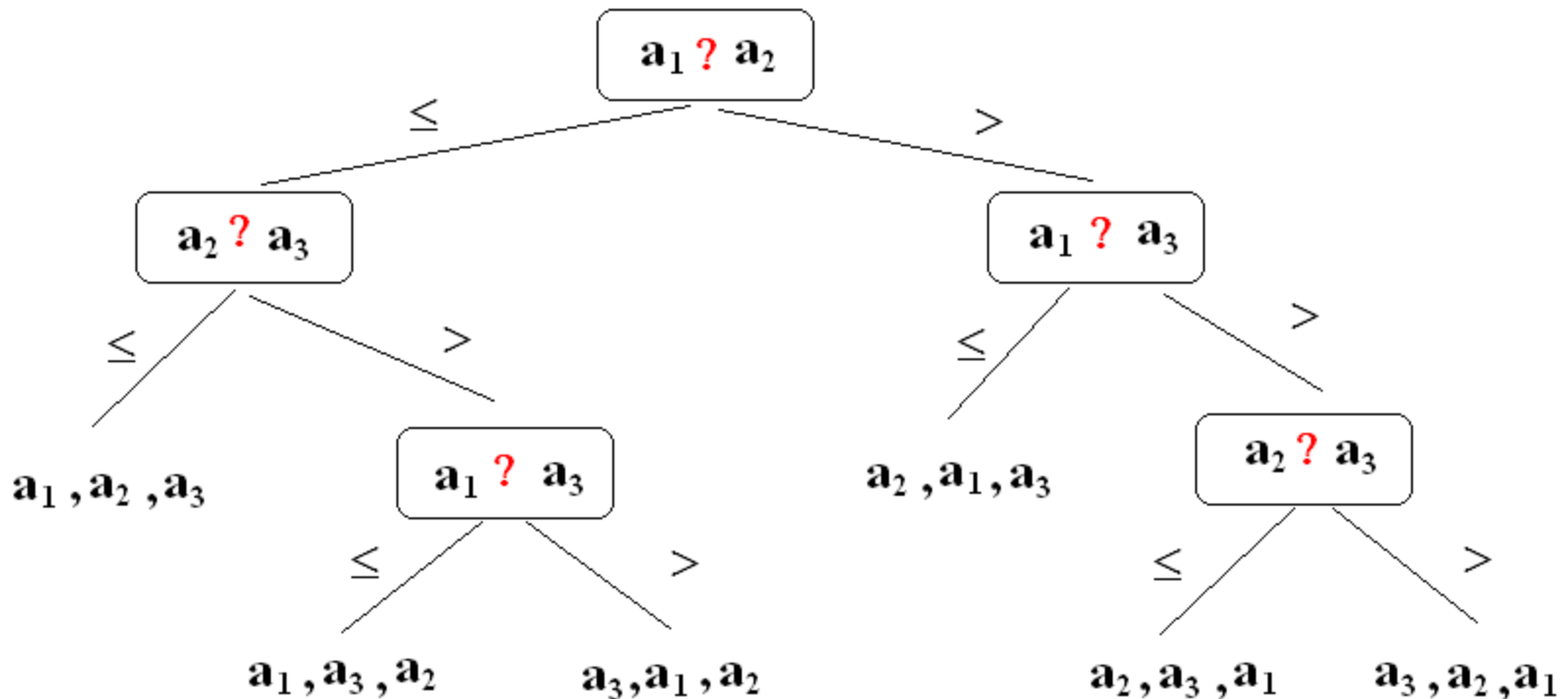
appear in the output in the same order as they do in the input

Decision tree

A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size

- Example:

A decision tree for insertion sort operating on 3 elements



Stirling's approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta \left(\frac{1}{n}\right)\right)$$

(Cormen)

$$\Rightarrow \log(n!) = \Theta(n \log n)$$

Lower bounds for comparison sort

- **Theorem**

Any decision tree that sorts n elements has height $(\geq) \Omega(n \lg n)$.

- **Consequence**

Heapsort is asymptotically optimal comparison sort.

Think about:

- Can we make a heap-sort stable ?