Sorting (cont.)

Alexandre David
B2-206

Heap Data Structure

- Binary tree with some properties:
  - **root**=A[1] - root of the tree
  - **parent**(i)=i/2 - for a node i
  - **left**(i)=2i - left child
  - **right**(i)=2i+1 - right child
  - **max-heap** property: A[parent(i)]≥A[i]
  - (or min-heap A[parent(i)]≤A[i])
  - height is $\Theta(\lg n)$.

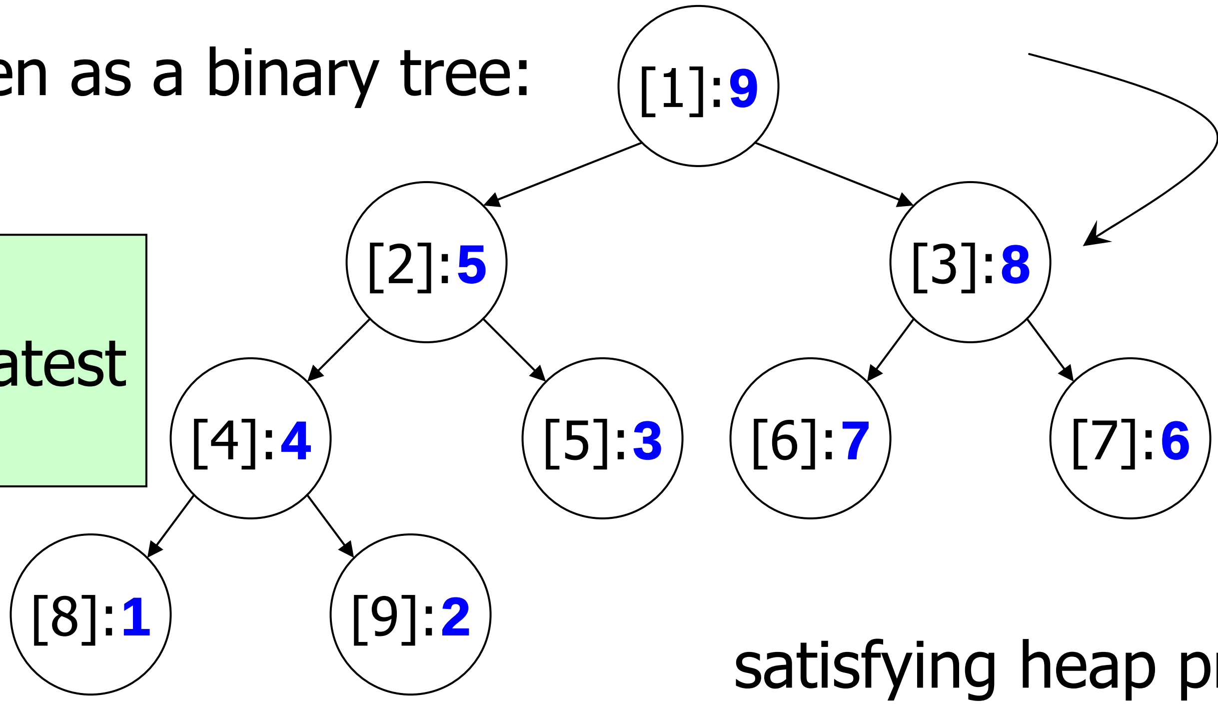
Heap Data Structure

- **root**=A[1]
- **parent**(i)=i/2
- **left**(i)=2i
- **right**(i)=2i+1
- **max-heap**
  A[parent(i)]≥A[i]

Array:

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
9	**5**	**8**	**4**	**3**	**7**	**6**	**1**	**2**

seen as a binary tree:



Max-heap:
root = greatest element.

double number of nodes

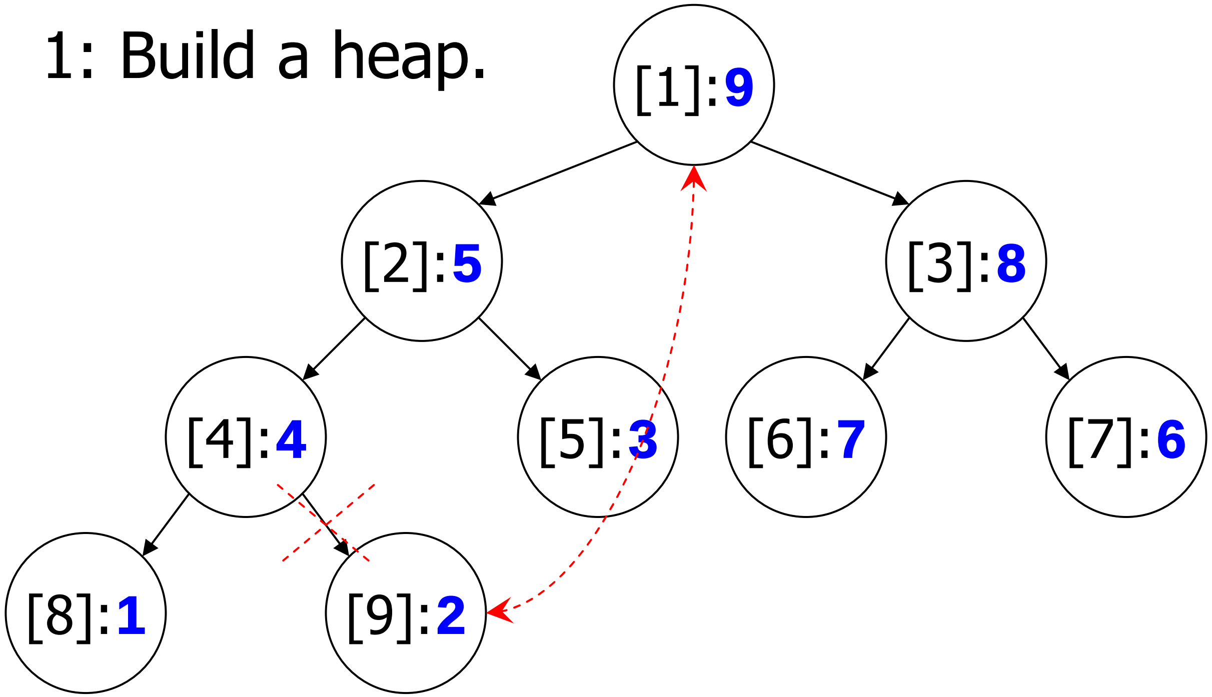
satisfying heap properties

Heap Sort

- Idea:
  - Use a heap data structure.
  - Max-heap for ascending: 1st element is the max $\Rightarrow$ move it at the end, restore the heap, loop.
- Basic procedures:
  - **max-heapify**: maintains max-heap property $O(\lg n)$, knowing that sub-trees are heaps.
  - **build-max-heap**: computes a max-heap from scratch in $O(n)$.
  - **heap-sort**: sorts an array in-place in $O(n \lg n)$.

Idea of Heap Sort

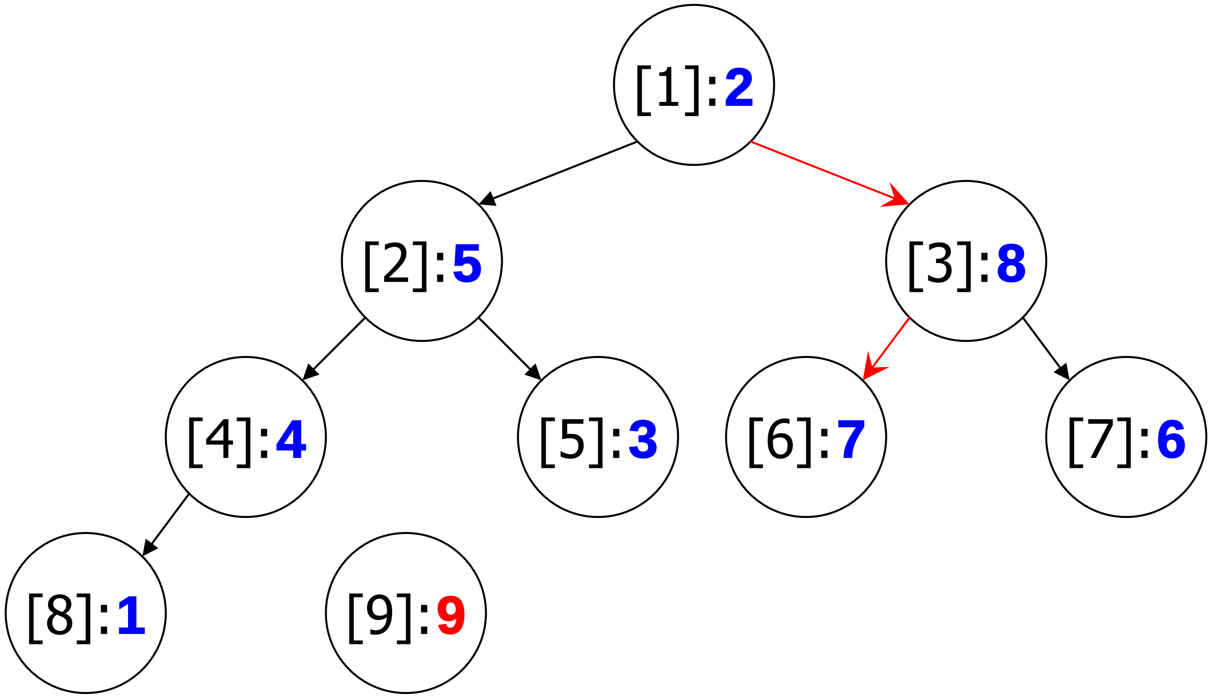
1: Build a heap.



2: Repeat

- move 1st at the end
- restore heap

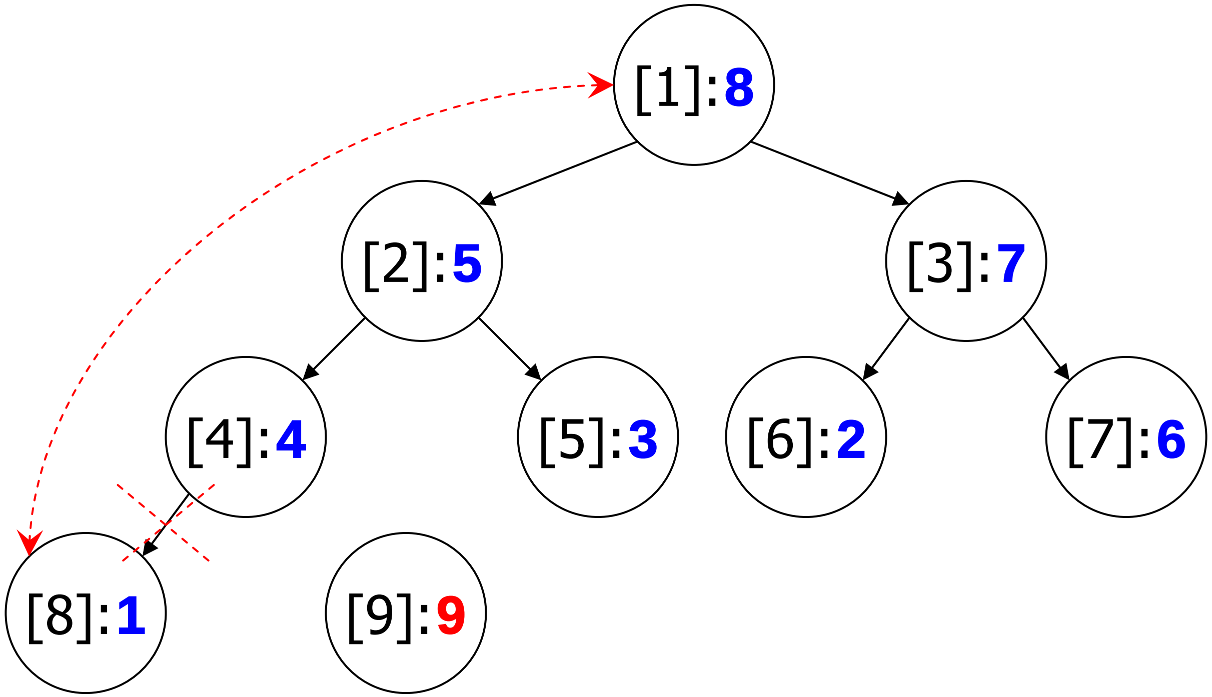
Idea of Heap Sort



2: Repeat

- move 1st at the end
- restore heap

Idea of Heap Sort



2: Repeat

- move 1st at the end
- restore heap

Maintaining a Heap

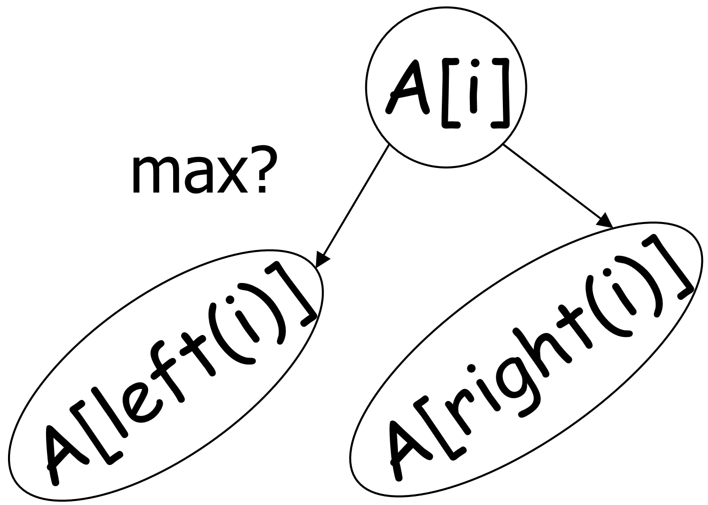
- Procedure **max-heapify**(A,i):
  - Assume that left(i) and right(i) are max-heaps.
  - A[i] may be smaller than its children, we're going to fix that.
  - Propagate A[i] down on the "right" path.
  - When the algorithm terminates, the max-heap property is restored.

Maintaining a Heap

simplified

```
max-heapify(A,i)
    if A[left(i)] > A[right(i)] then
        child = left(i)
    else
        child = right(i)
    fi
    if A[i] < A[child] then
        swap(A[i], A[child])
        max-heapify(A, child)
    fi
```

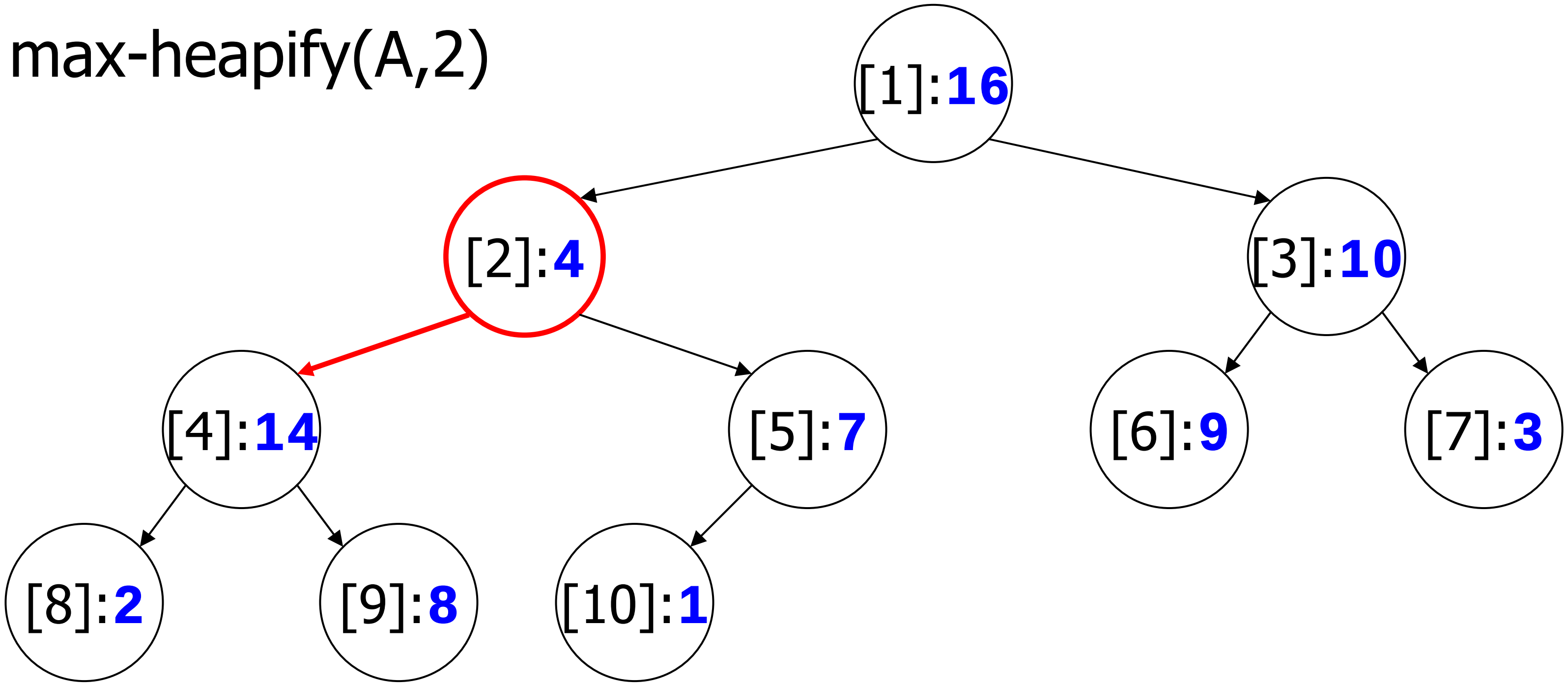
Clearly $O(\lg n)$.



Why is it correct?

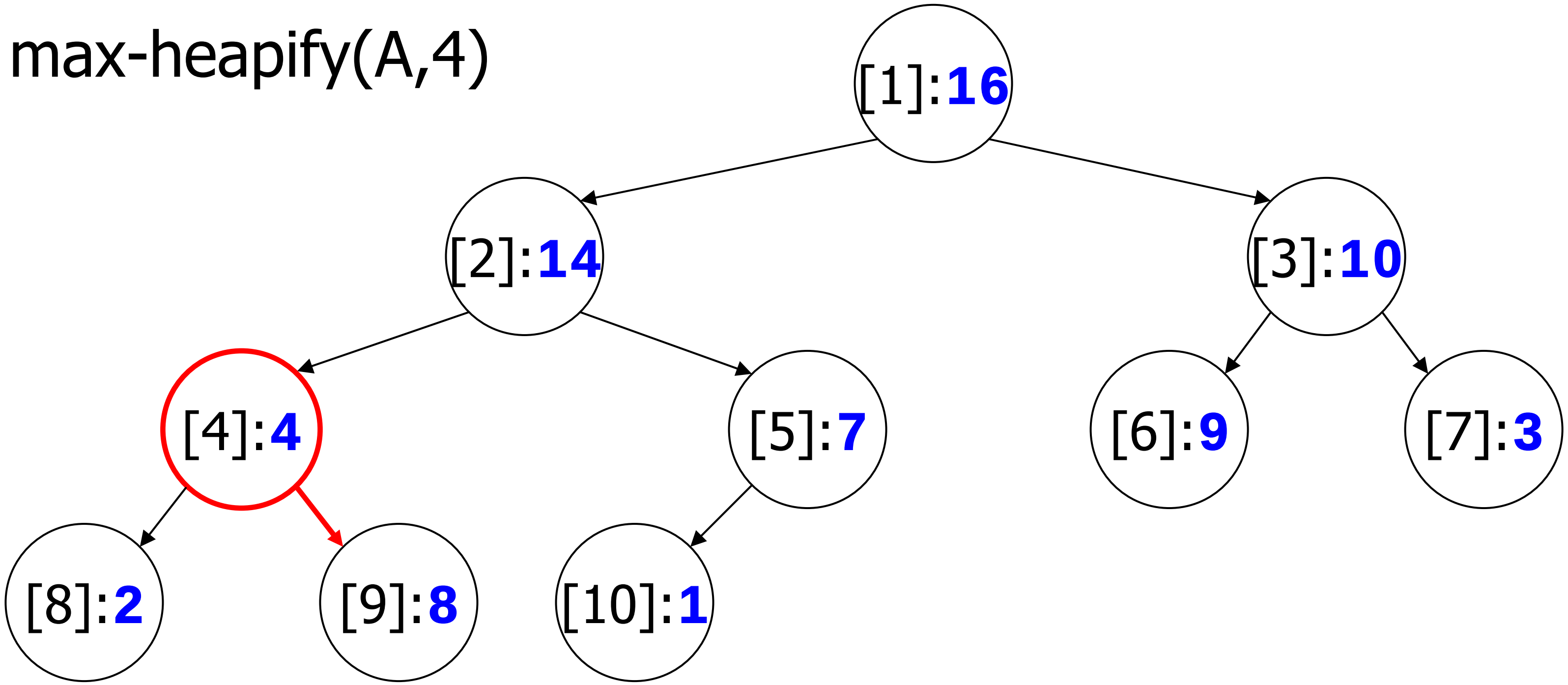
Maintaining a Heap

max-heapify(A,2)



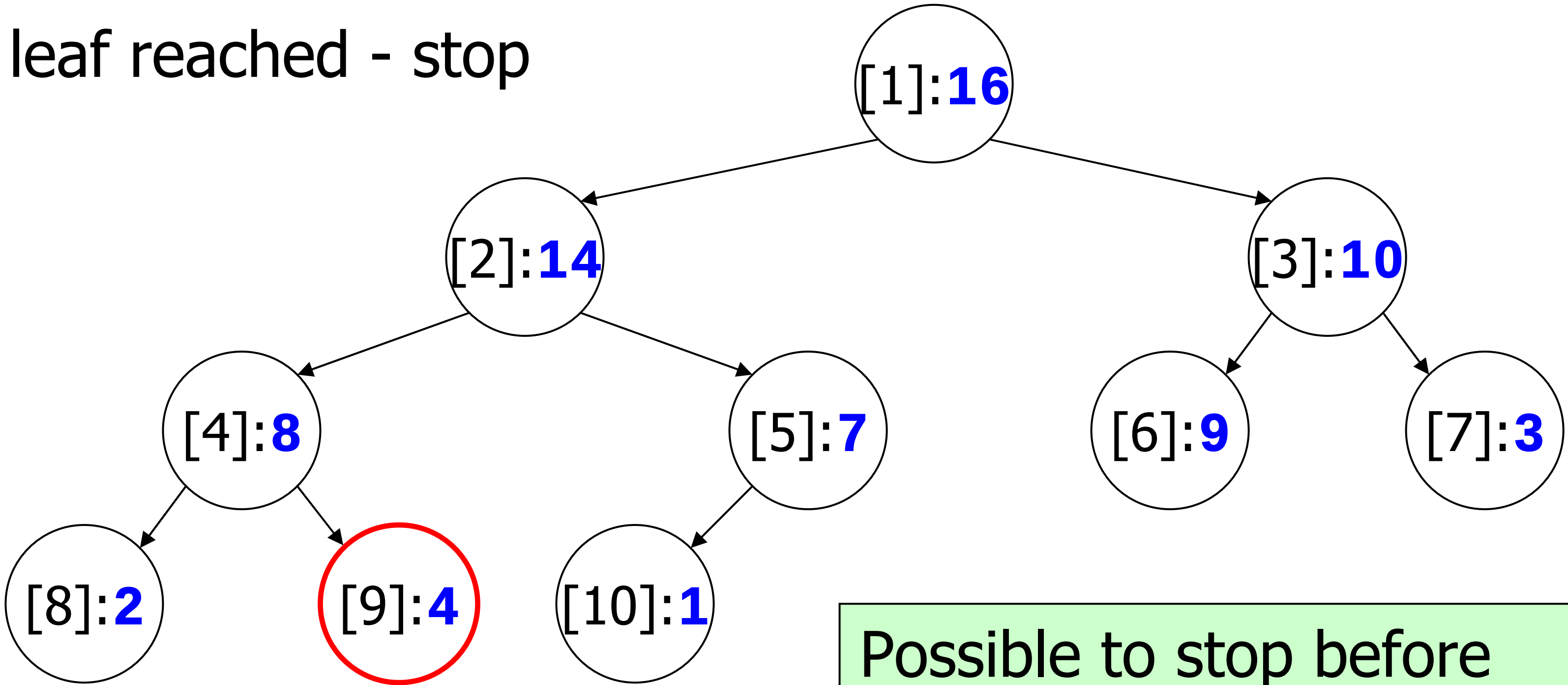
Maintaining a Heap

max-heapify(A,4)



Maintaining a Heap

leaf reached - stop



Possible to stop before
if A[i] ≥ max(A[left],A[right]).

Building a Heap

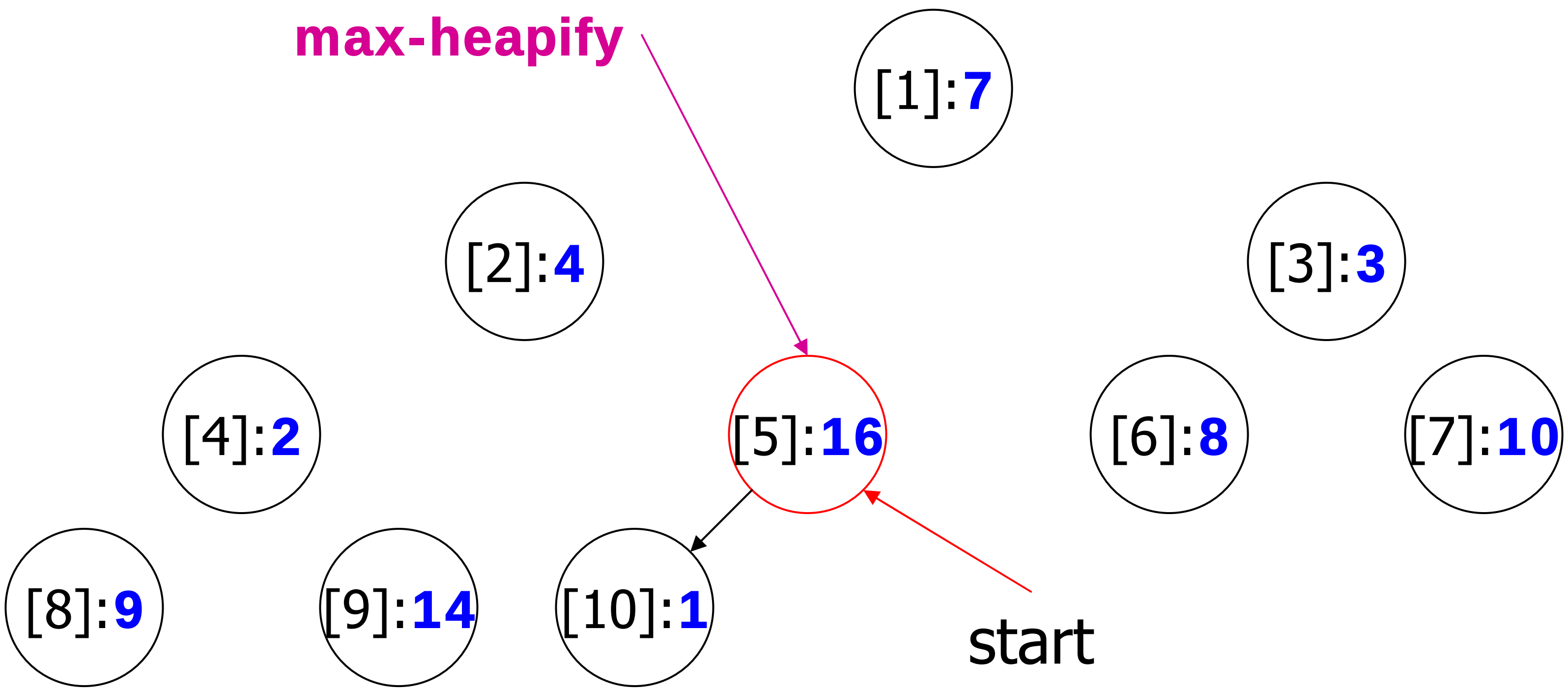
- We can use **max-heapify** in a bottom-up manner to build the tree incrementally.

```
build-max-heap(A):
heap_size(A) = length(A)
for i = length(A)/2 downto 1 do
    max-heapify(A,i)
done
```

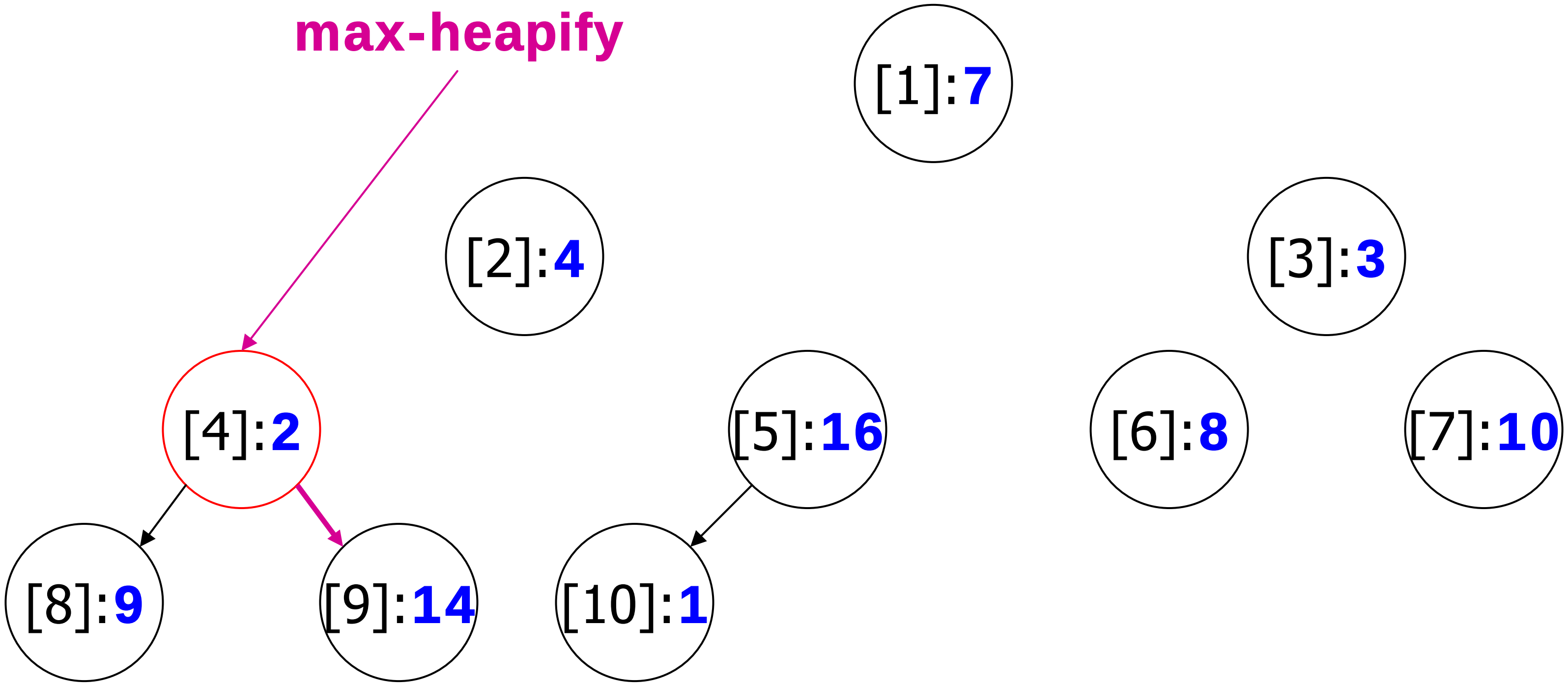
Part of A being a heap.

- Seems $O(n \lg n)$ but is in fact $O(n)$.
- ? Why do we start from length(A)/2?
- ? Why does it work?

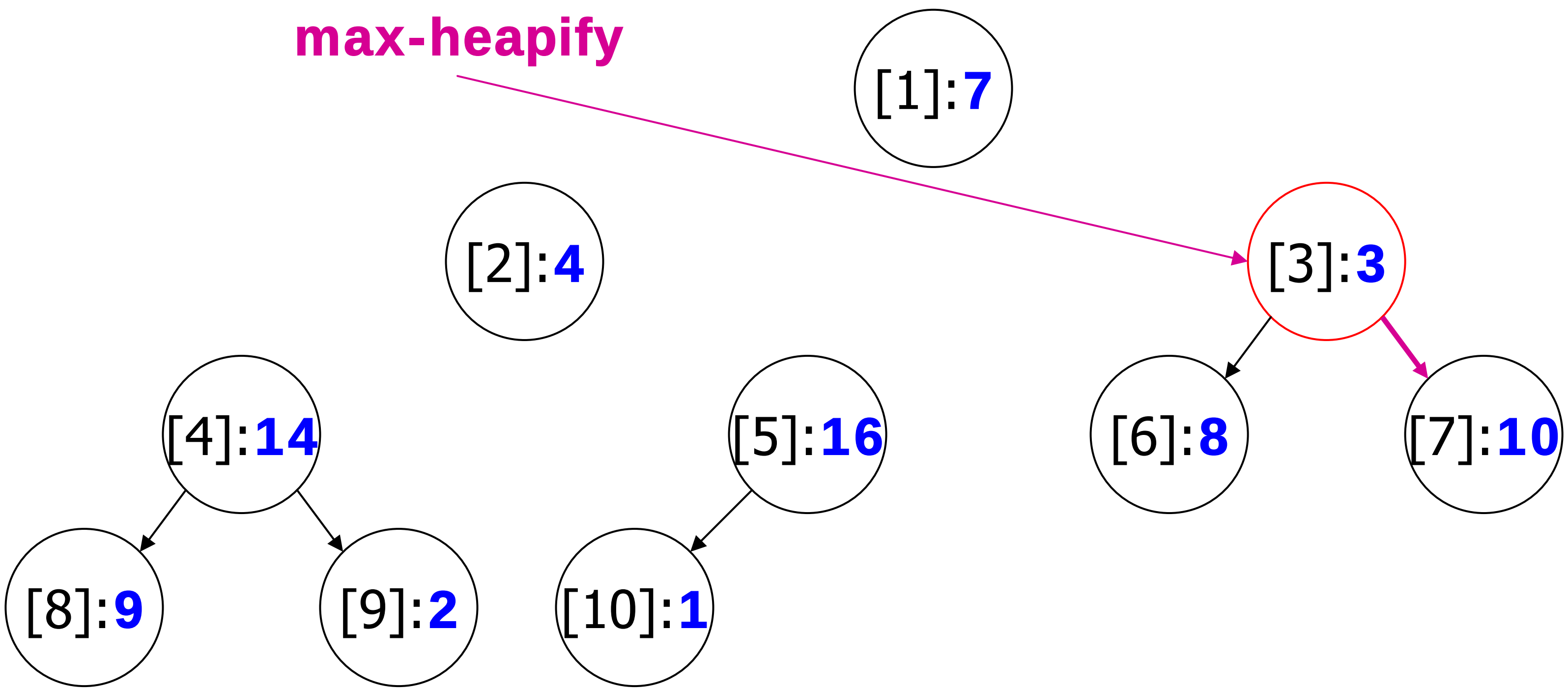
Building a Heap



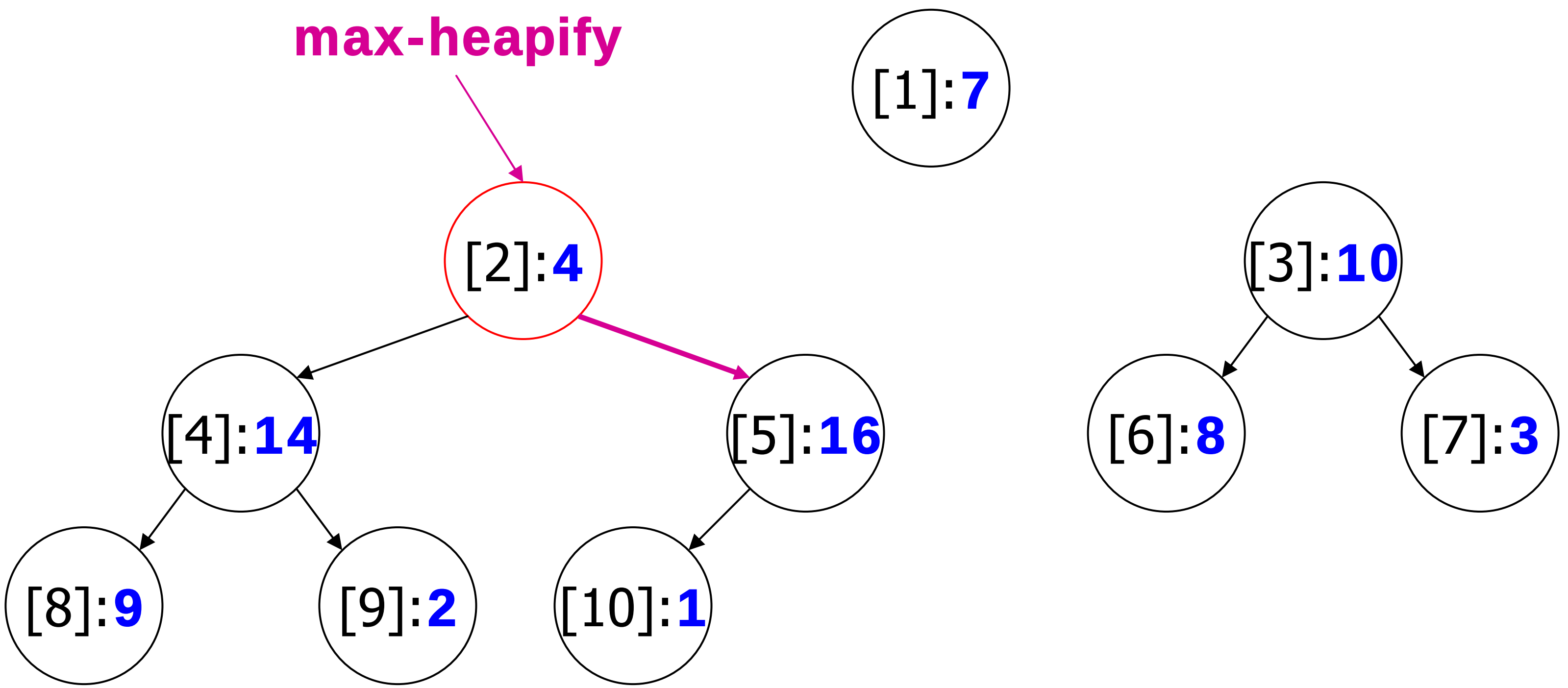
Building a Heap



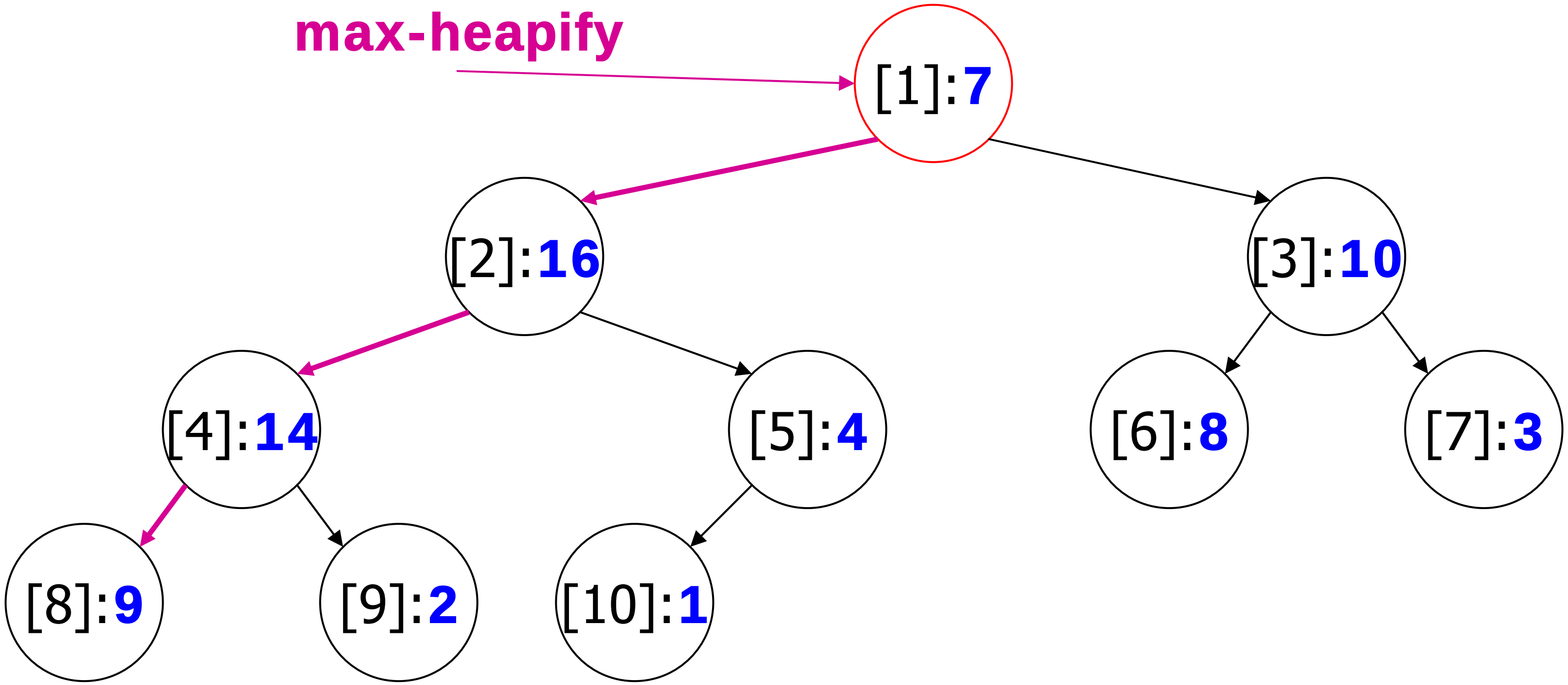
Building a Heap



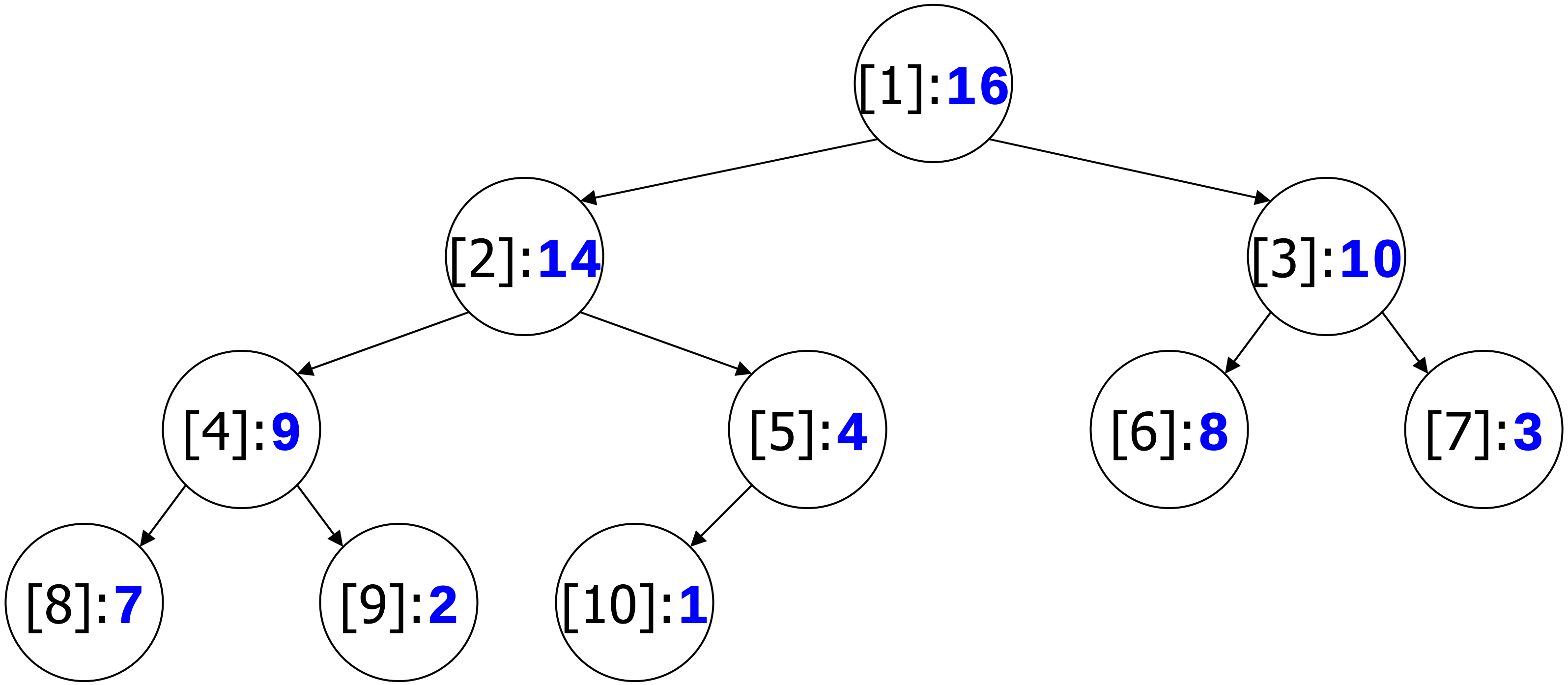
Building a Heap



Building a Heap



Building a Heap



All Together: Heap Sort

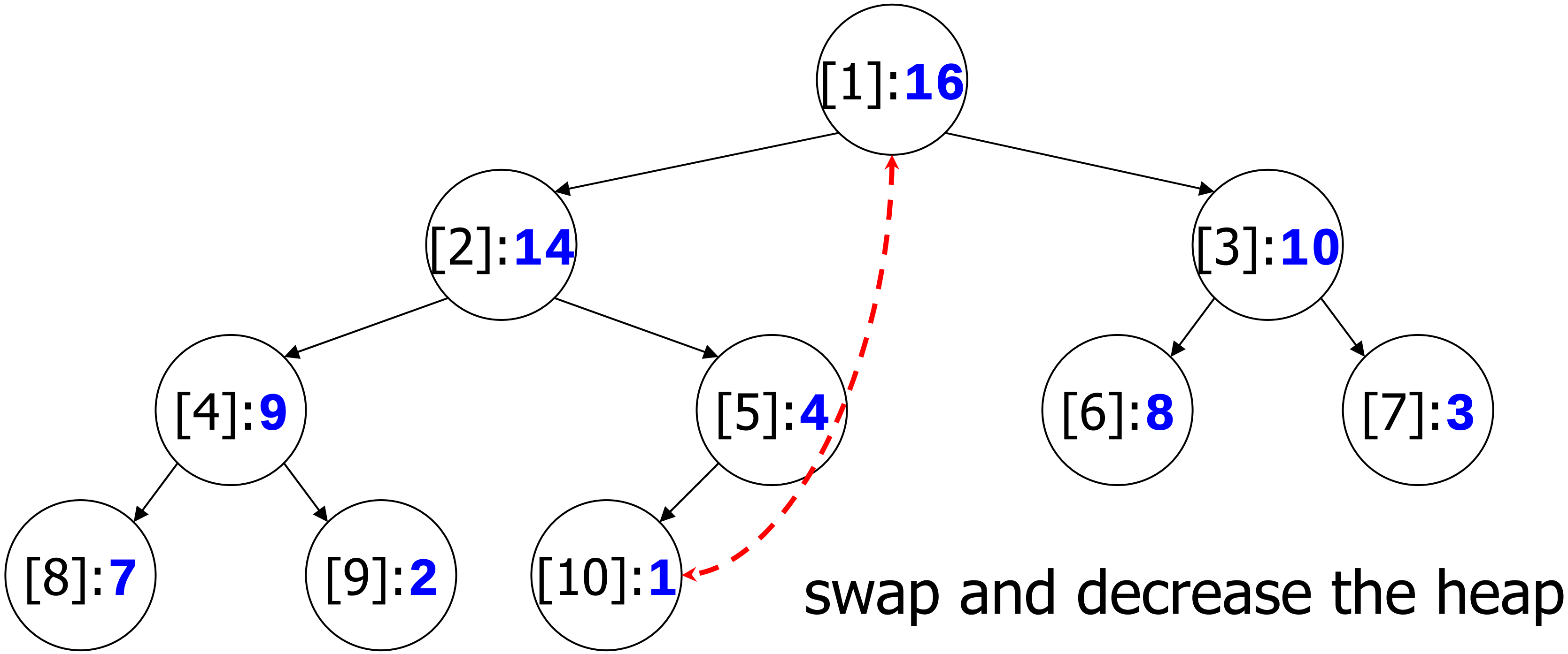
- Build a max-heap.
- Repeat:
  - Put the root at the end of the heap,
  - max-heapify on the smaller heap.

```
heap-sort(A):
build-max-heap(A)
for i = length(A) downto 2 do
    swap(A[1], A[i])
    heap_size(A) = heap_size(A) - 1
    max-heapify(A, 1)
done
```

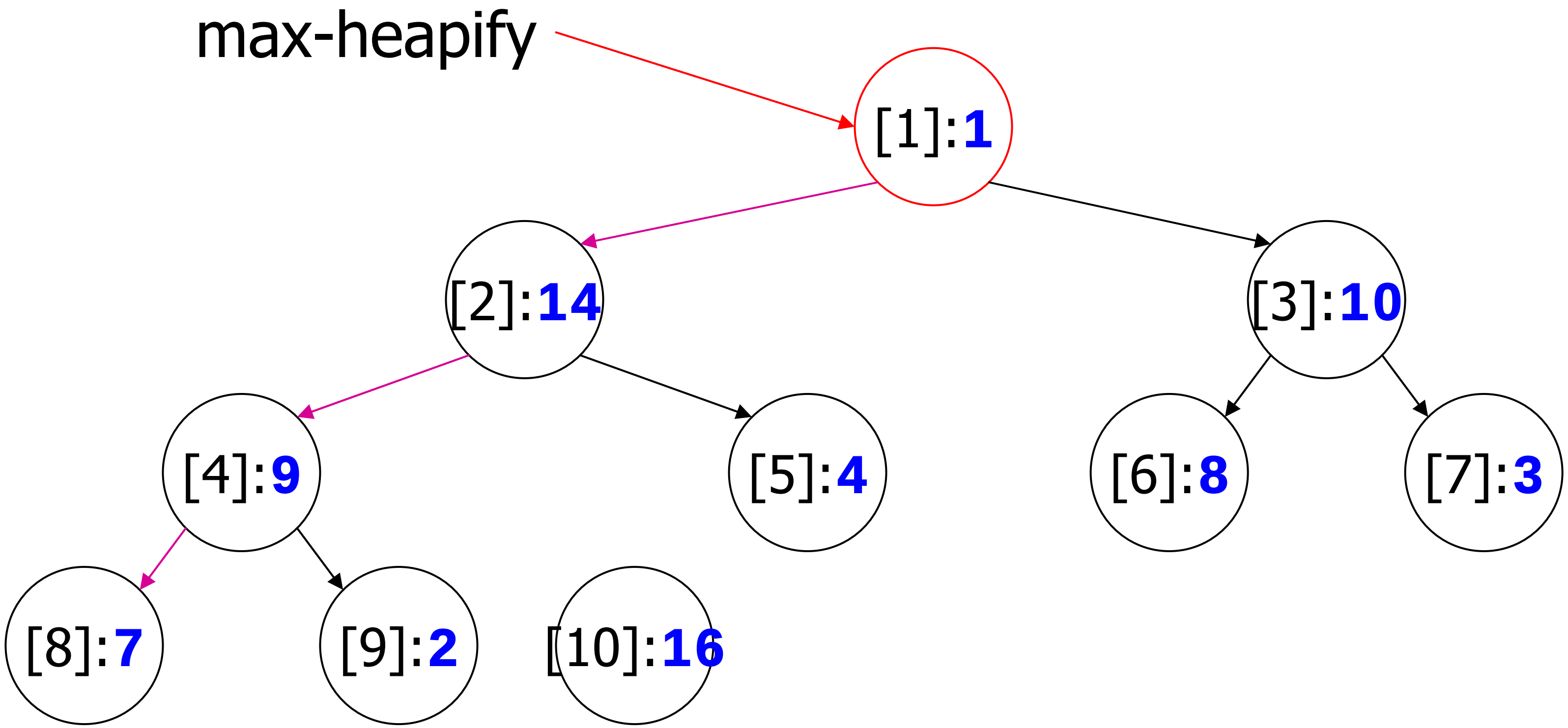
Clearly $O(n \lg n)$ but also $\Omega(n \lg n)$.

Heap Sort

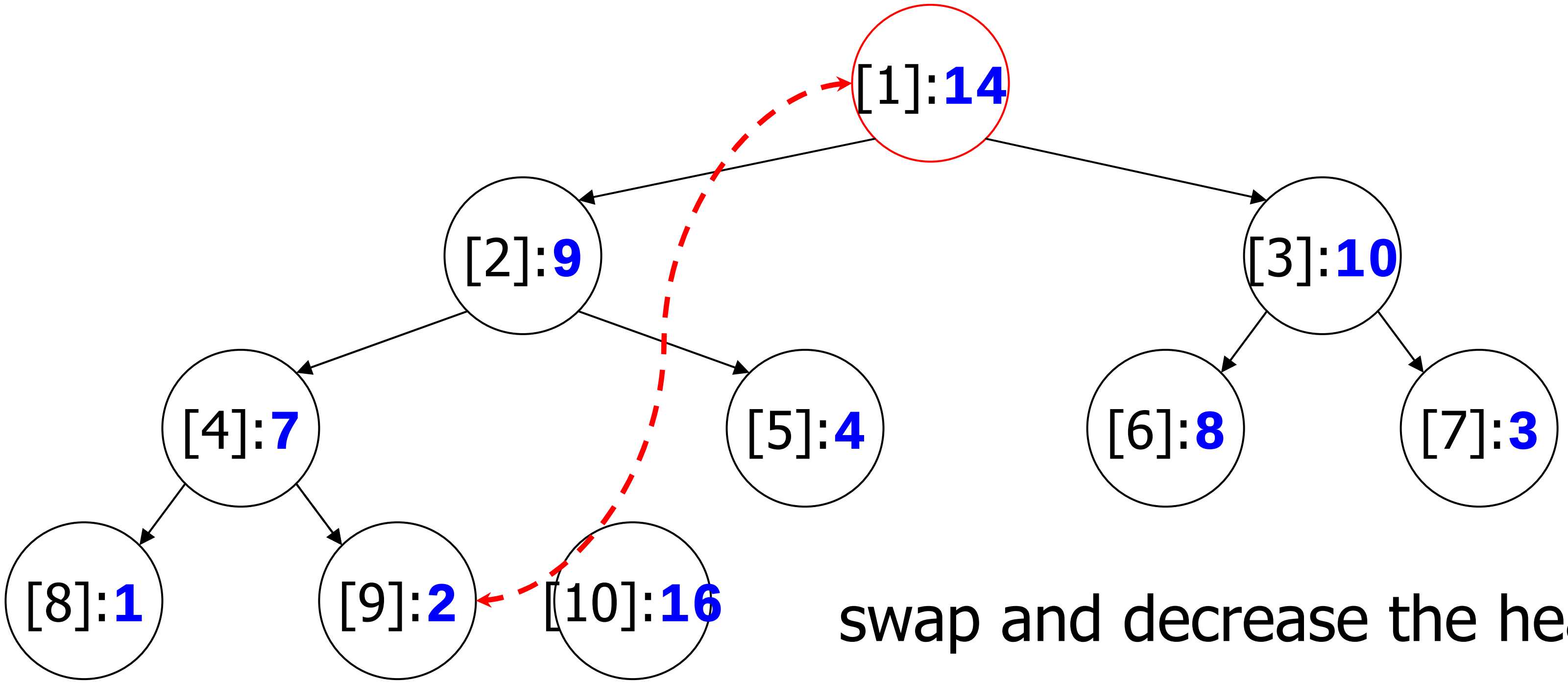
After building the heap:



Heap Sort

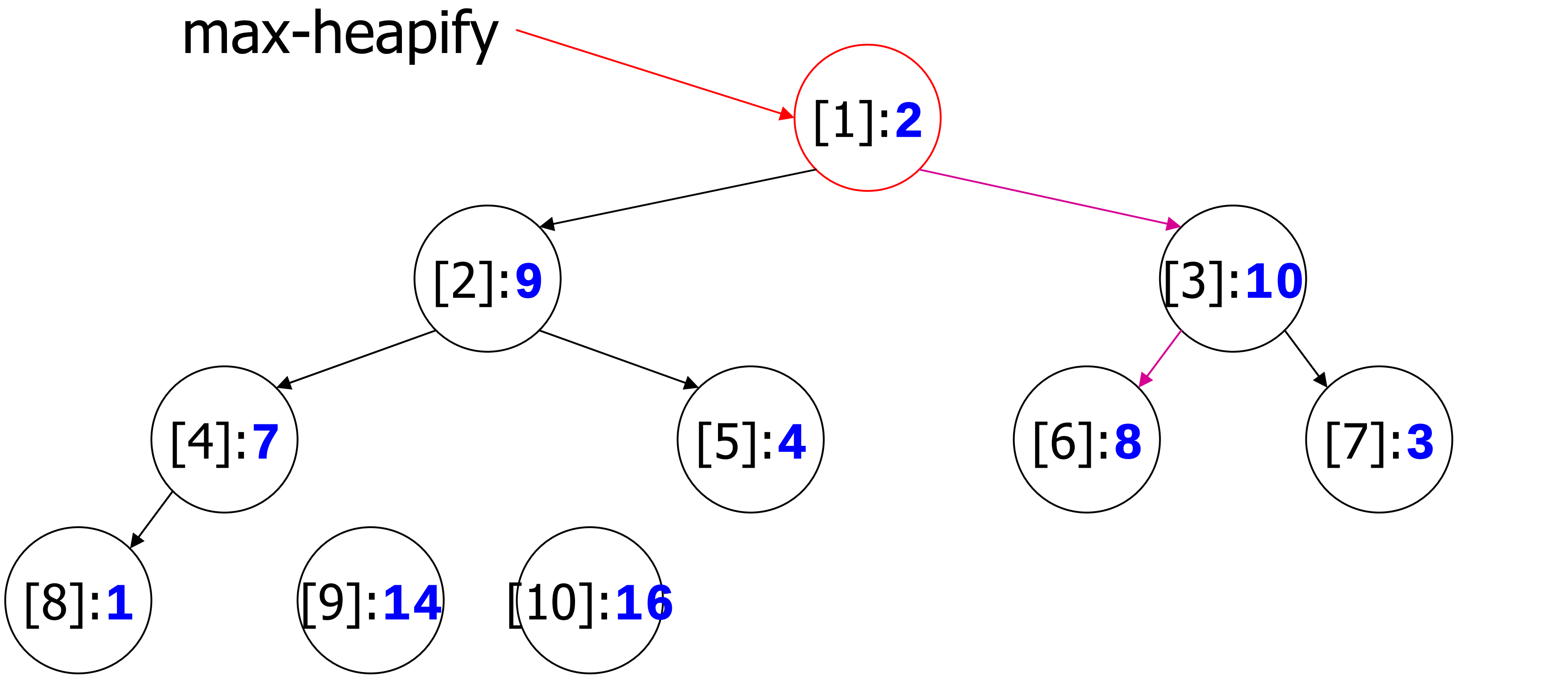


Heap Sort

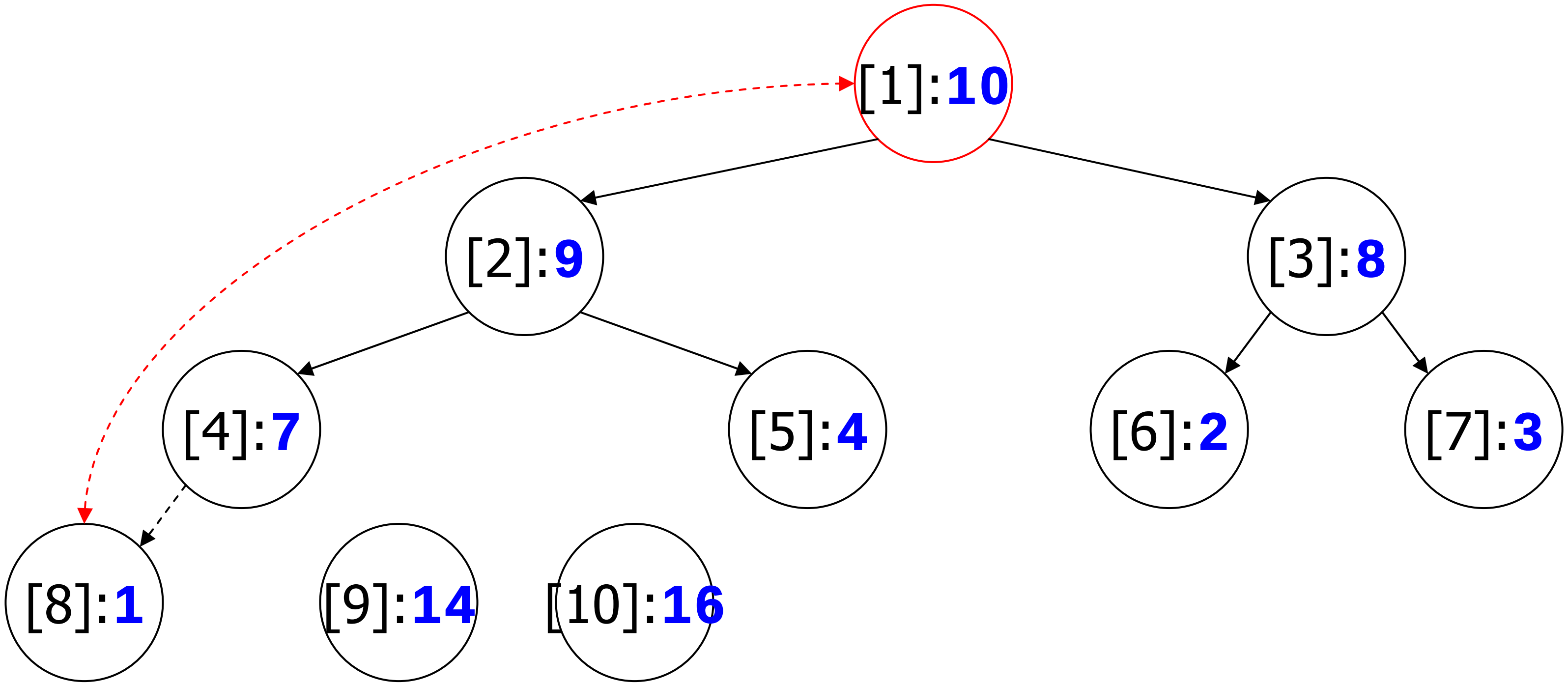


swap and decrease the heap

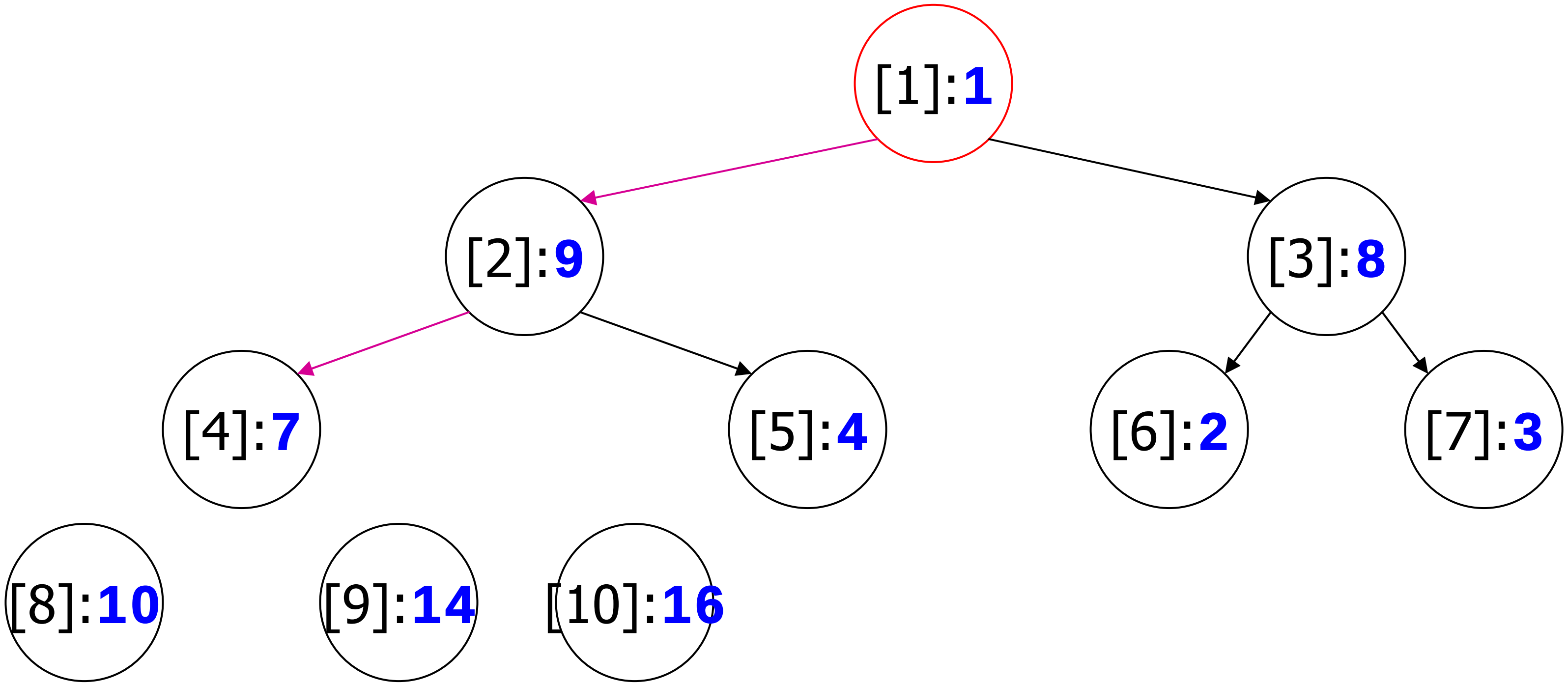
Heap Sort



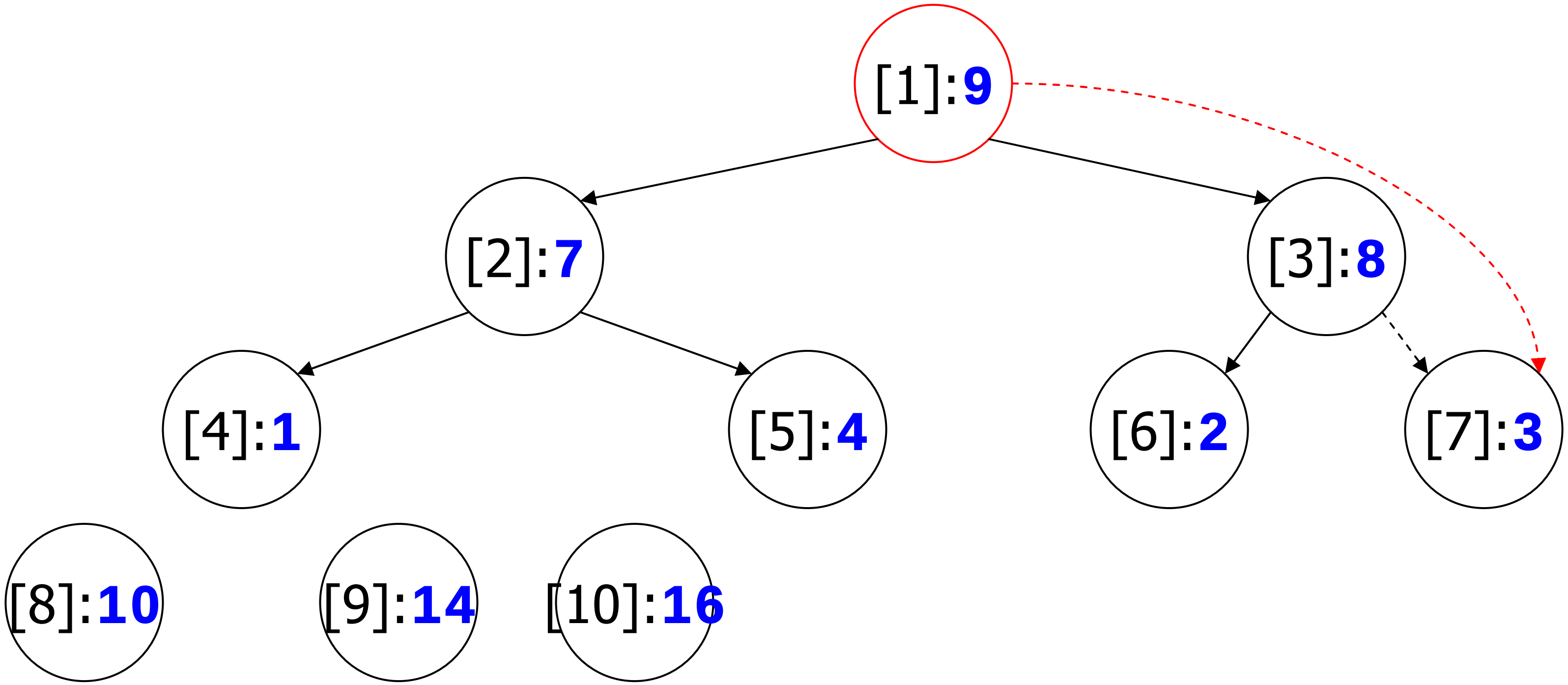
Heap Sort



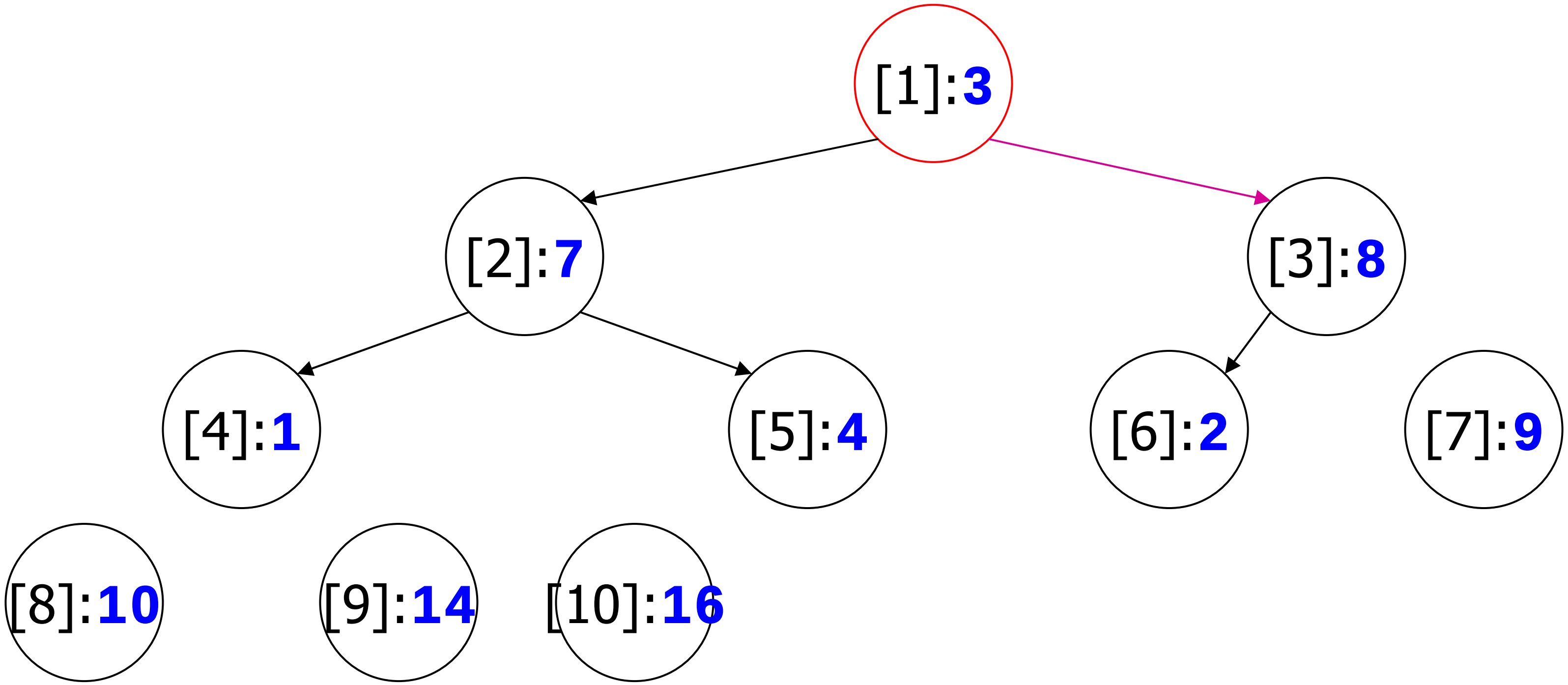
Heap Sort



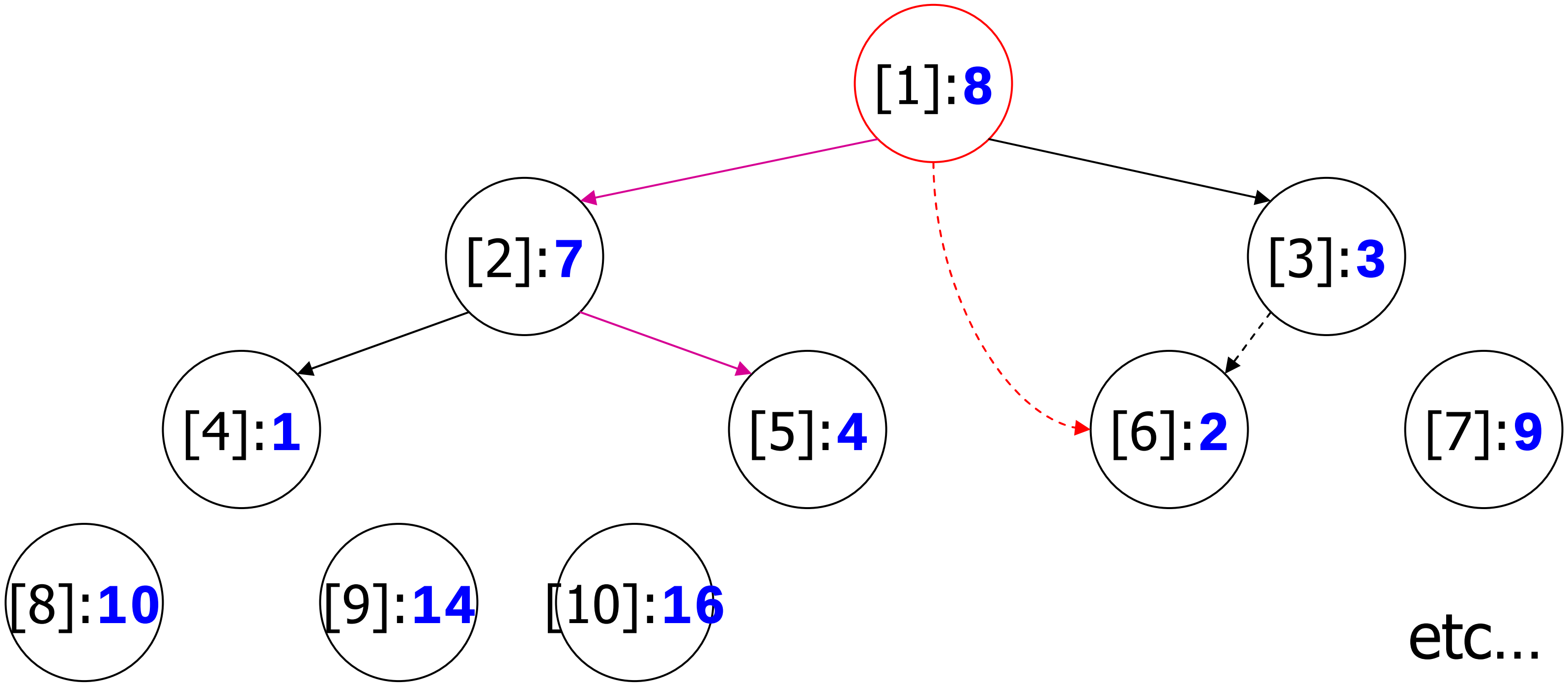
Heap Sort



Heap Sort

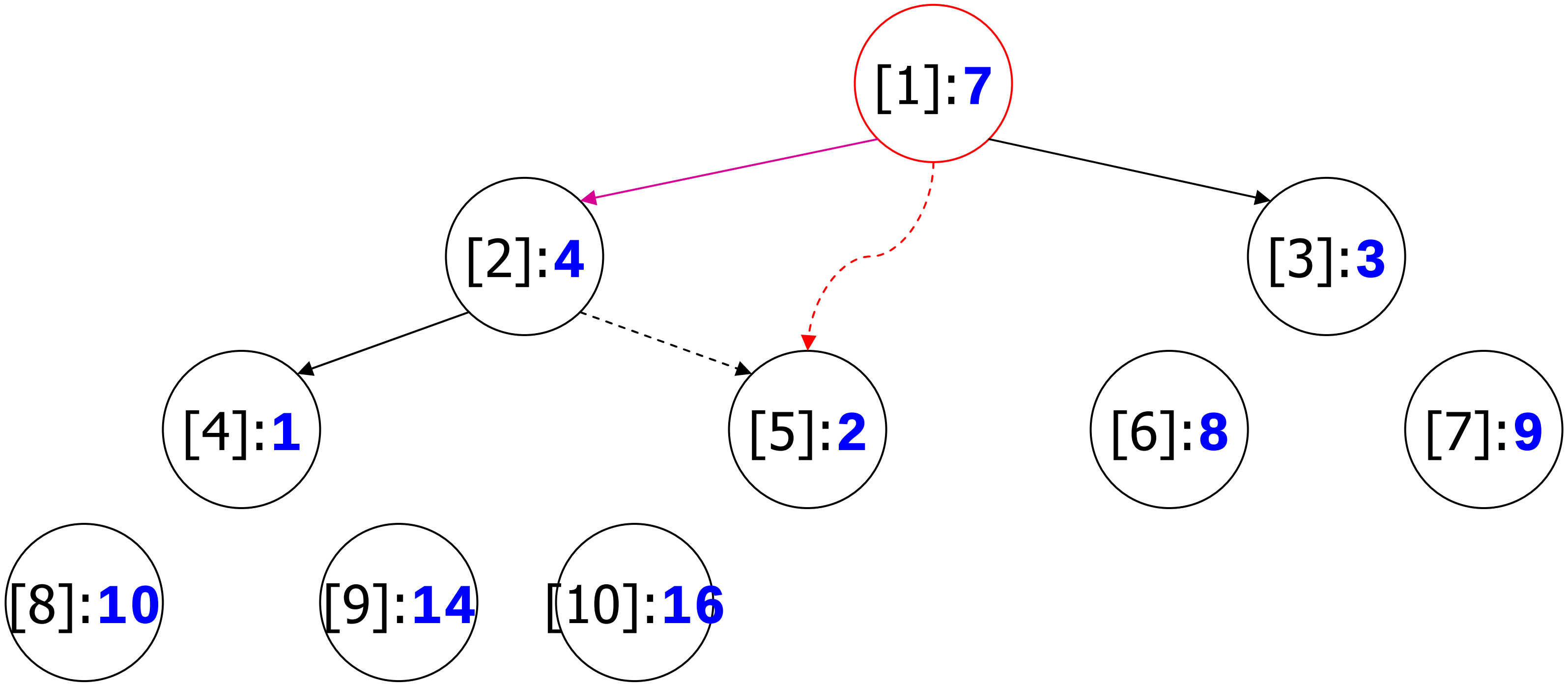


Heap Sort

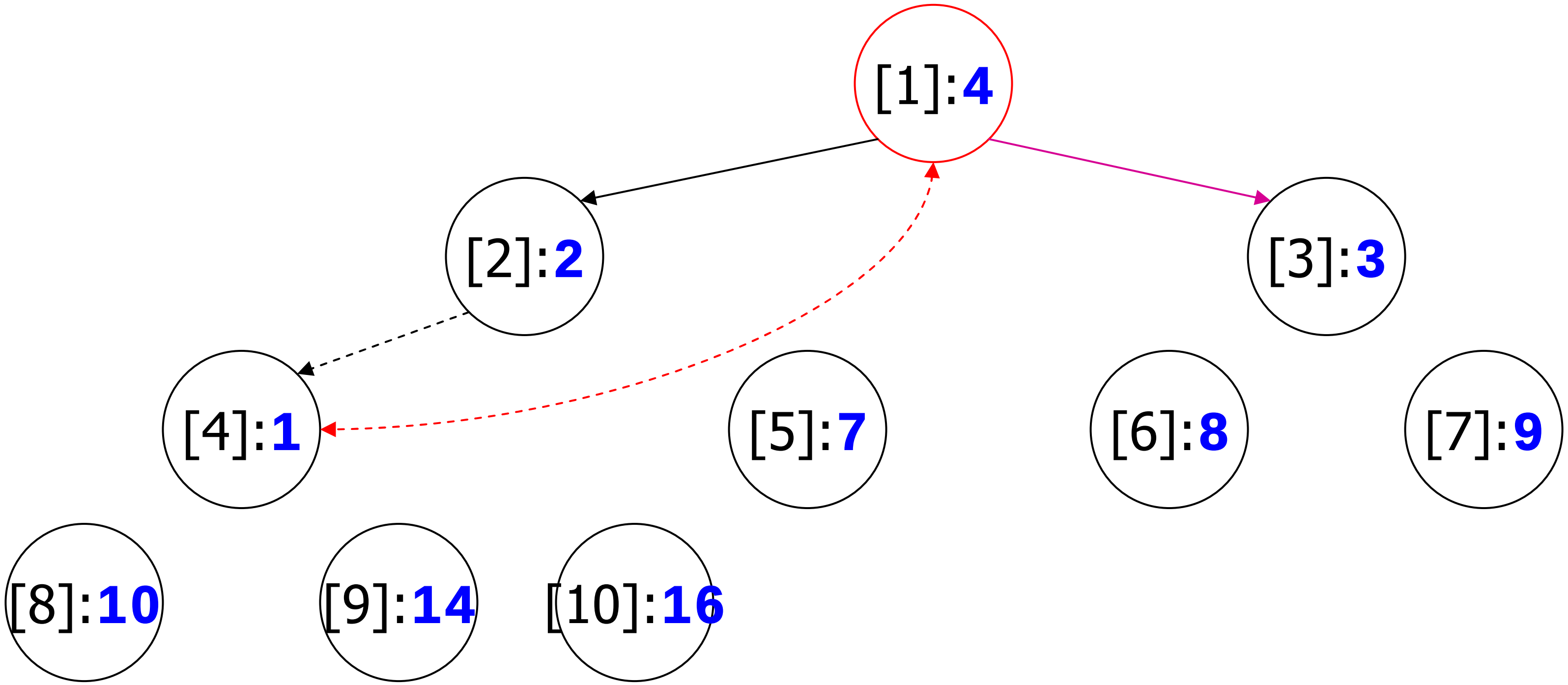


etc...

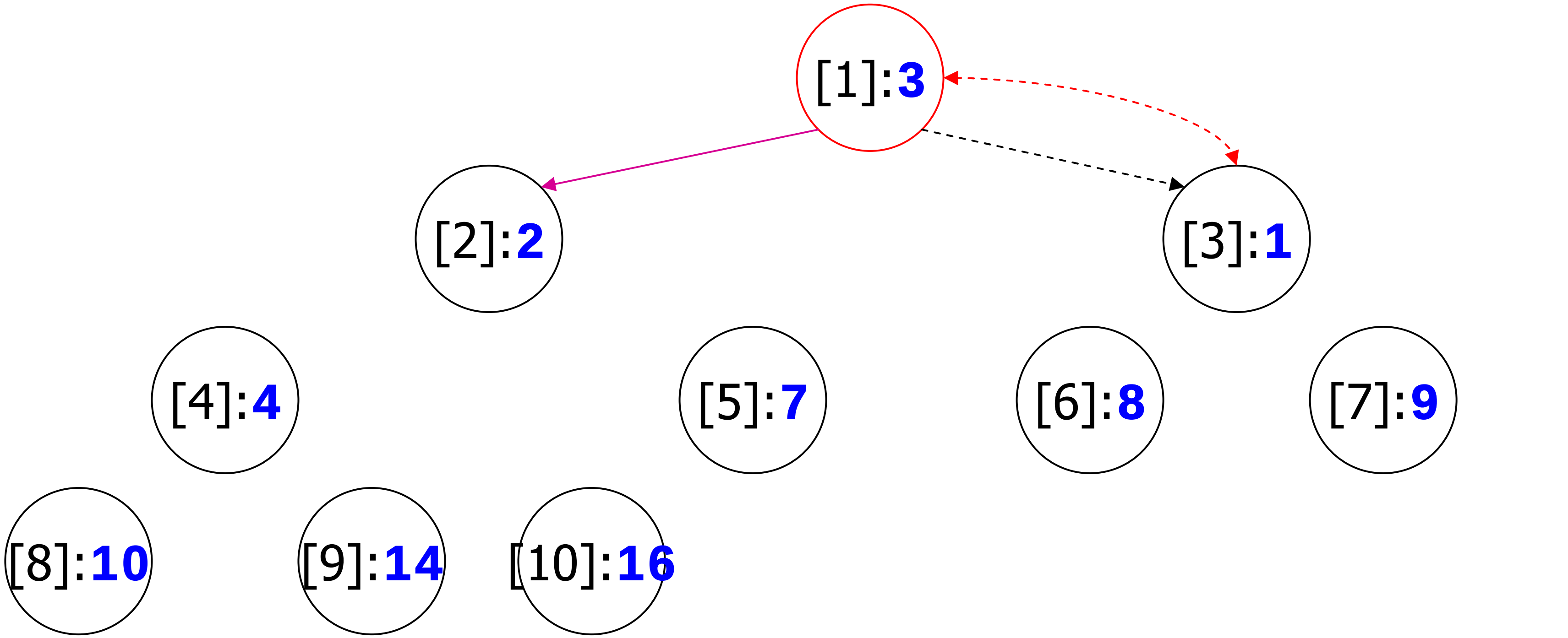
Heap Sort



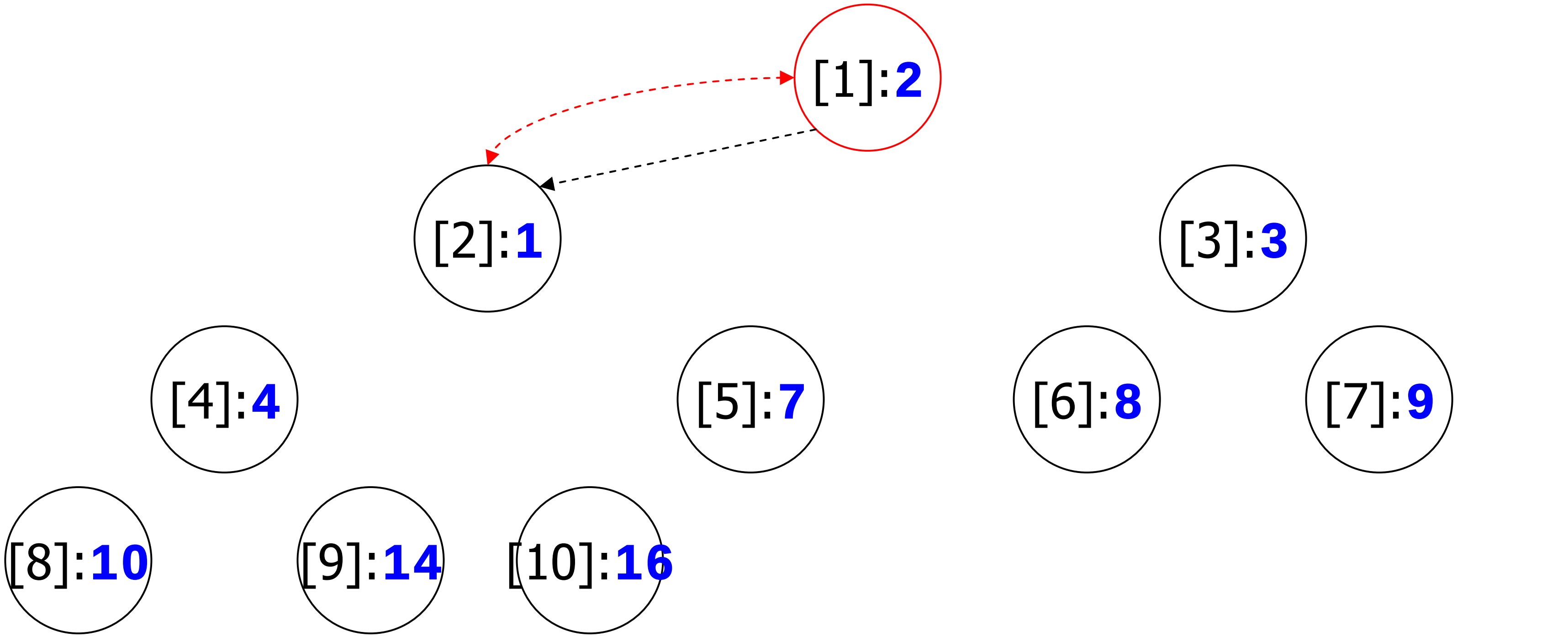
Heap Sort



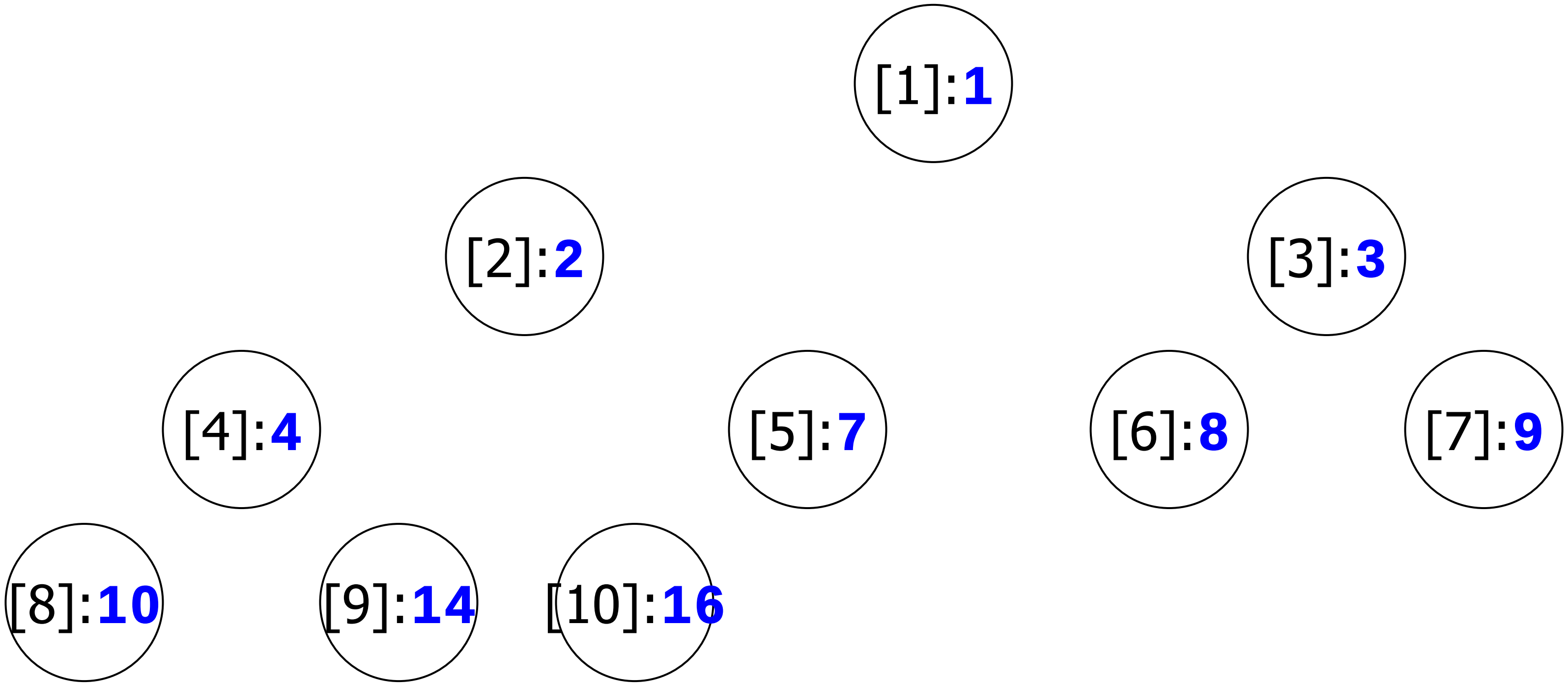
Heap Sort



Heap Sort



Heap Sort



Priority Queues

- Another use of the heap data structure.
- Exploit the max-heap property (or min-heap).
- Operations:
  - insert an element, $O(\lg n)$
  - get the max (or the min) element, $O(1)$
  - remove the max (or the min) element, $O(\lg n)$
  - increase the key of an element. $O(\lg n)$
  - Used in the STL (C++).

Important Procedure: Increase Key

- Propagate the key upwards until the max-heap property is restored.

```
heap-increase-key(A,i,key):
if key < A[i] then error
A[i] = key
while i > 1 and A[parent(i)] < A[i] do
    swap(A[i], A[parent(i)])
    i = parent(i)
done
```

Increasing a Key