Algorithms and Data Structures 2

8 – HEAPSORT

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Recap**
  - MERGE-SORT
  - QUICKSORT
- **HEAPSORT**
- **Lower bounds for comparison sorts**
  - Decision tree model

Recap

- **MERGE-SORT and QUICKSORT are two efficient divide-and-conquer sorting algorithm**

	MERGE-SORT	QUICKSORT
Best case running time	O(n log n)	O(n log n)
Average case running time	O(n log n)	O(n log n)
Worst case running time	O(n log n)	$O(n^2)$
Space complexity	O(n)	O(log n)
Stable	Yes	No

HEAPSORT

- **Efficient O(n log n) sorting algorithm**
  - Originally invented by Williams in 1964
- **Inspired by SELECTION-SORT (see Lab sheet 1)**
  - Divide the input array into a sorted and an unsorted region
  - Iteratively extract the maximum from the unsorted region and move it to the sorted region
  - We use a heap data structure rather than a linear-time search to find the maximum

- **The heap data structure is useful for HEAPSORT, but it also makes an efficient priority queue**

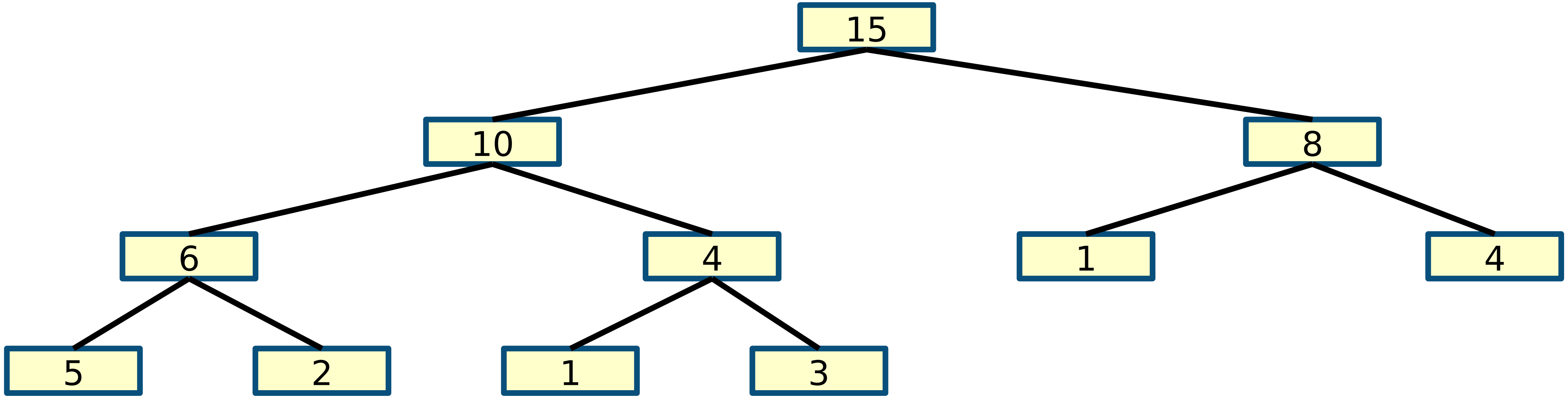
The heap data structure

- **A heap is a nearly complete binary tree that satisfies the heap property**

> **if p is a parent node of c, then the the value of p is either greater than or equal to the value of c**

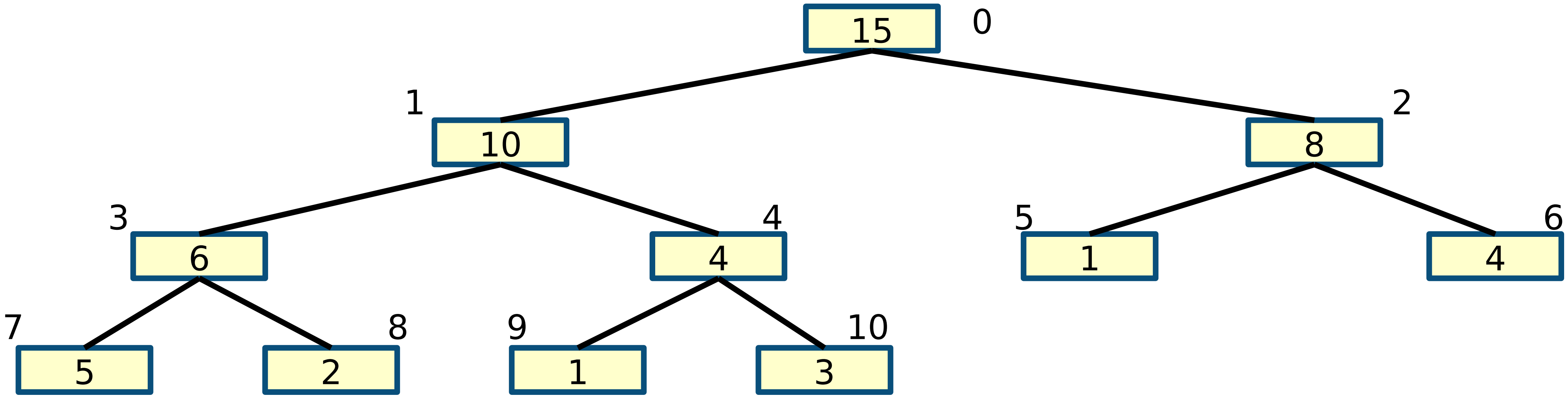
- **This is also called max-heap because the the maximum element is stored at the root of the heap**
  - A min-heap stores the minimum element at the root and has dual heap property
- **Heaps can be implemented as arrays**

Max-heap example



- The max-heap property holds for each subtree
- Nearly complete: all levels are complete but the last one

Max-heap example



– A max-heap is represented as an array by assigning index 0 starting from the root and then increasing the index while going downwards from left to right on each tree level

15	10	8	6	4	1	4	5	2	1	3

HEAPSORT

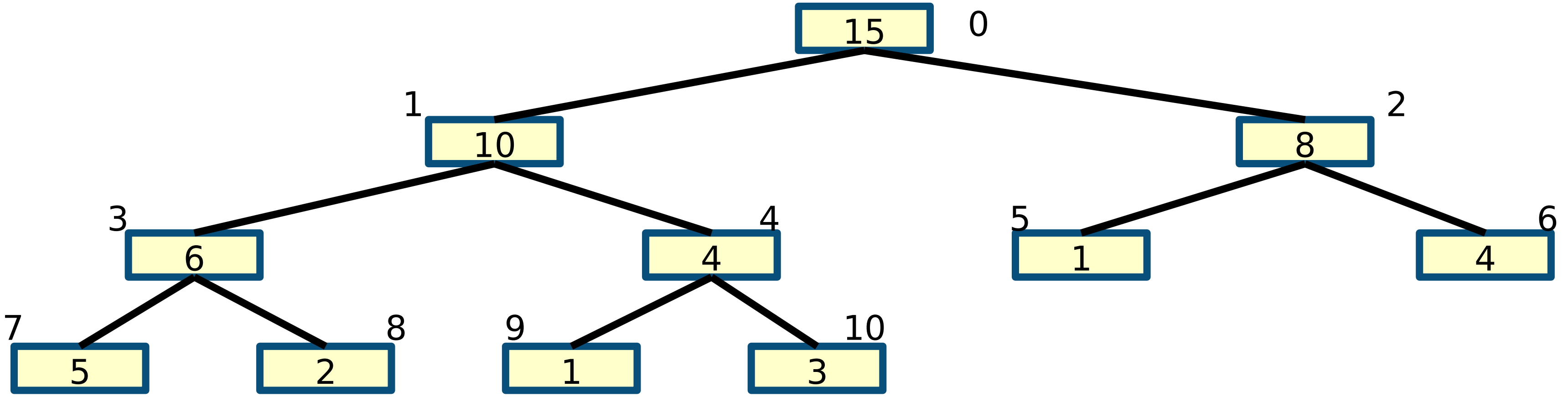
- **Input: Array A of size n**
- **Output: sorted array A**

- **BUILD-MAX-HEAP(A)**
  - build a max-heap from an unordered input array in linear time
- **MAX-HEAPIFY(A,0,s)**
  - Maintain the max-heap property on A[0..s-1]
  - Assume A[0..s-1] is a max-heap but root A[0] might be smaller than its children, thus violating the max-heap property
  - O(log n)

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n   // the size of the heap
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

Example execution of HEAPSORT(A)

- **Assume BUILD-MAX-HEAP(A) produces the heap below**




```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

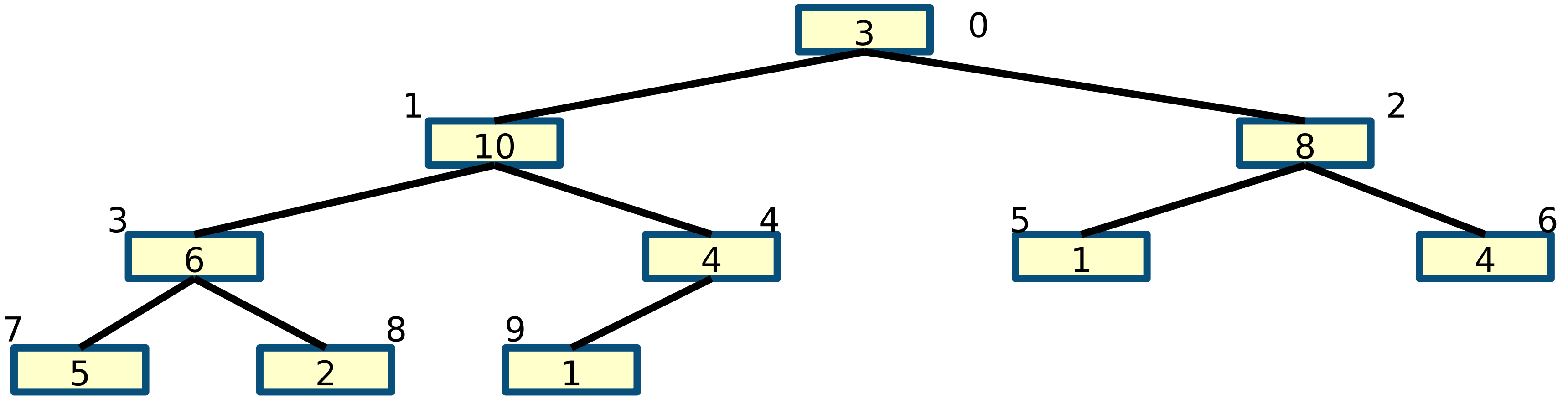
15	10	8	6	4	1	4	5	2	1	3

– Elements in the heap x

– Sorted elements x

s = n = 11

Example execution of HEAPSORT(A)

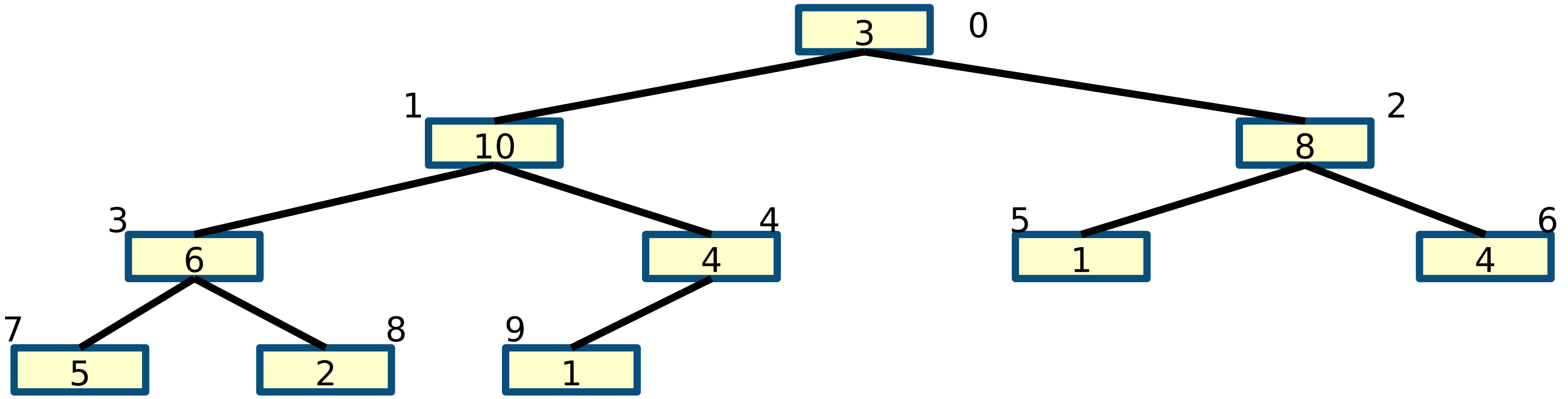



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

3	10	8	6	4	1	4	5	2	1	15
										i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

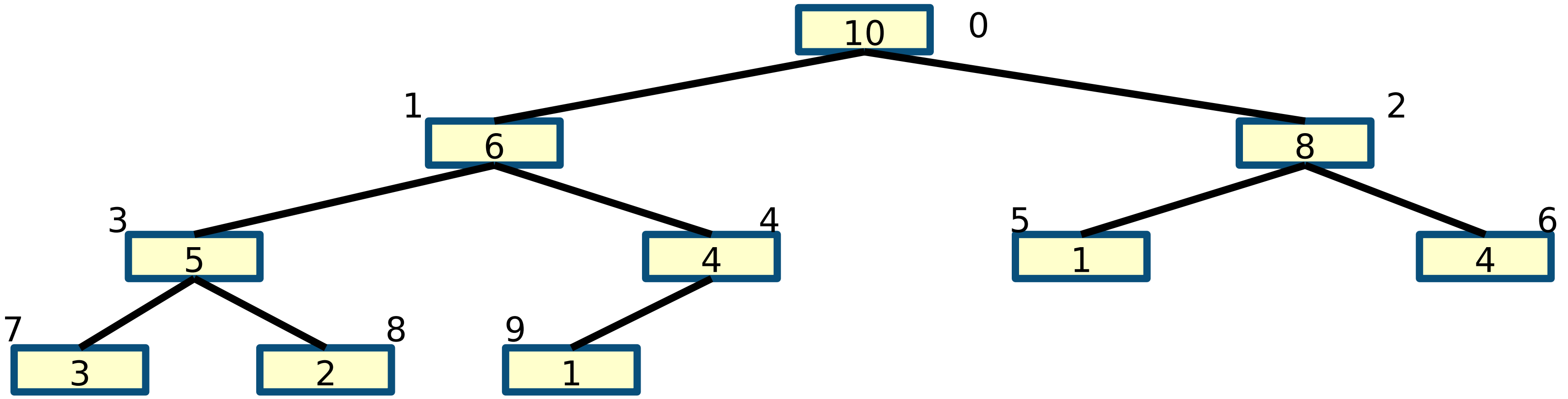



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

3	10	8	6	4	1	4	5	2	1	15
										i

- Heap has now s=10 elements
- The heap property is not satisfied: invoke MAX-HEAPIFY(A,0,10)

Example execution of HEAPSORT(A)

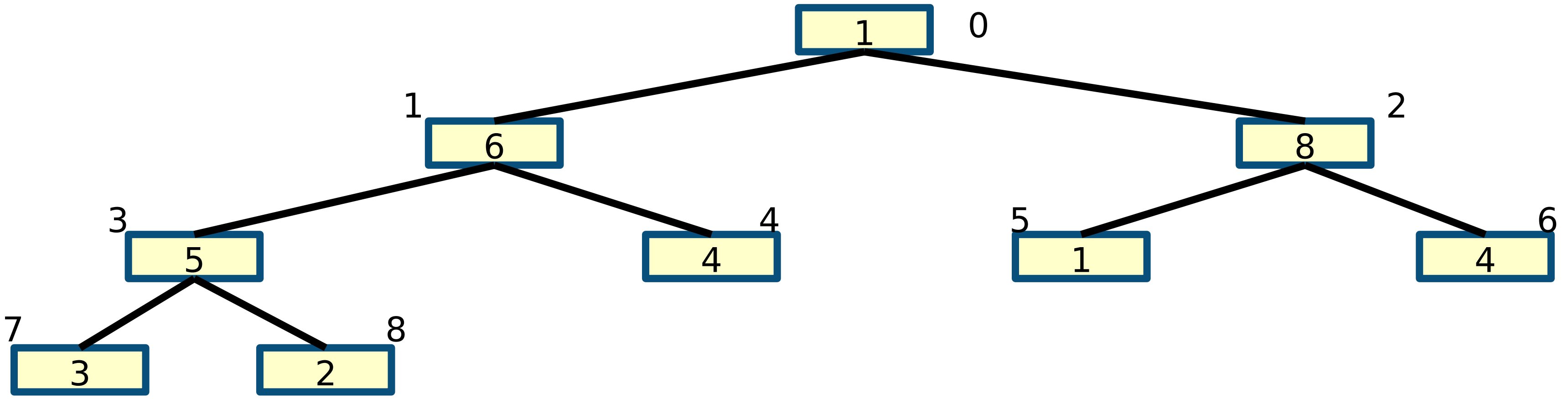



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

10	6	8	5	4	1	4	3	2	1	15
										i

– MAX-HEAPIFY recursively swaps the root with its largest child stopping when the heap property is satisfied

Example execution of HEAPSORT(A)



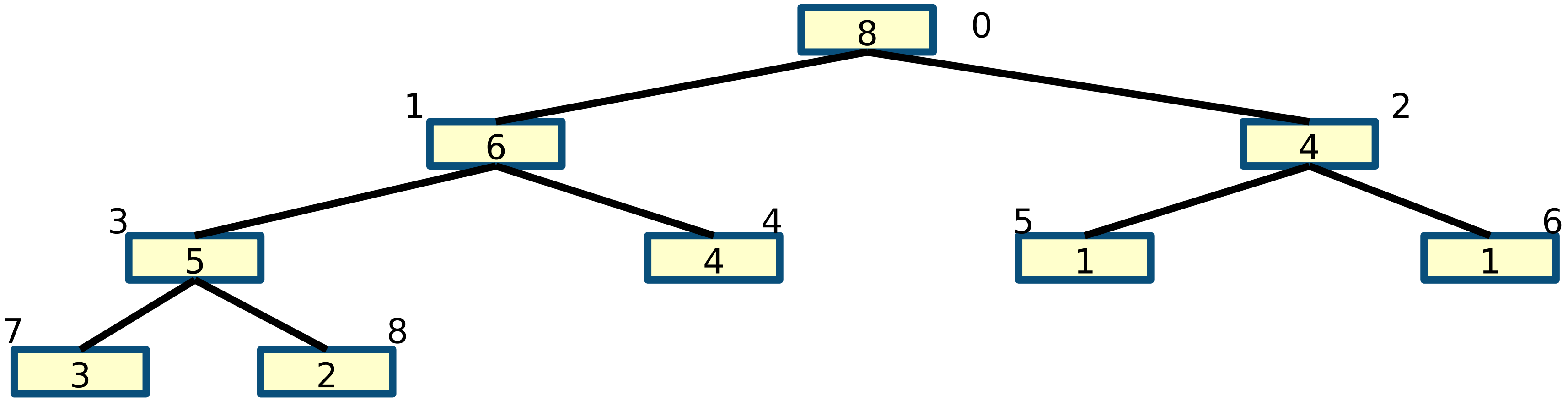

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

1	6	8	5	4	1	4	3	2	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

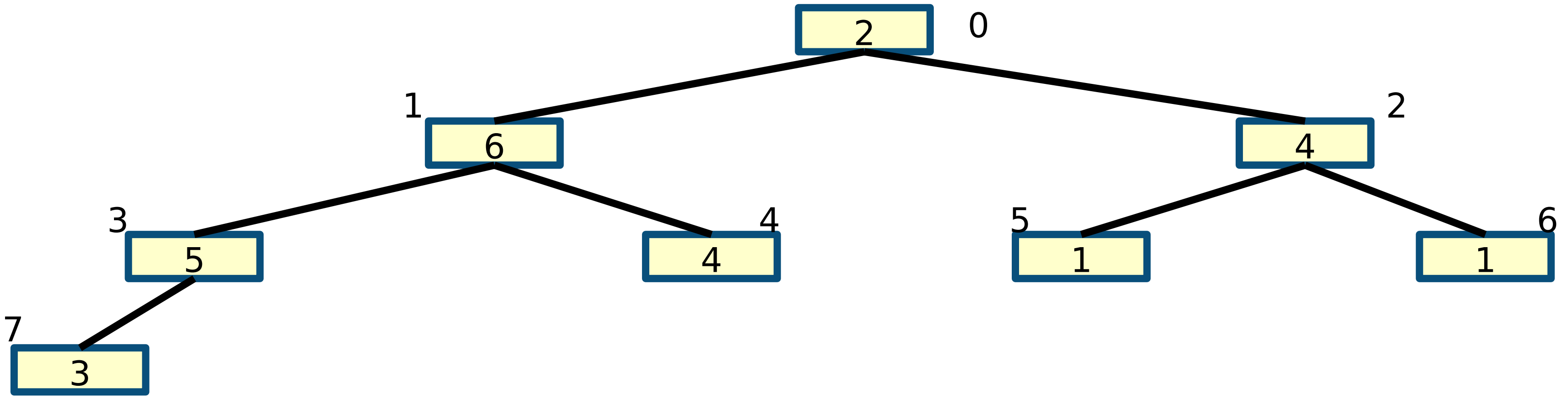



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

8	6	4	5	4	1	1	3	2	10	15
									i	

– Heap restored

Example execution of HEAPSORT(A)



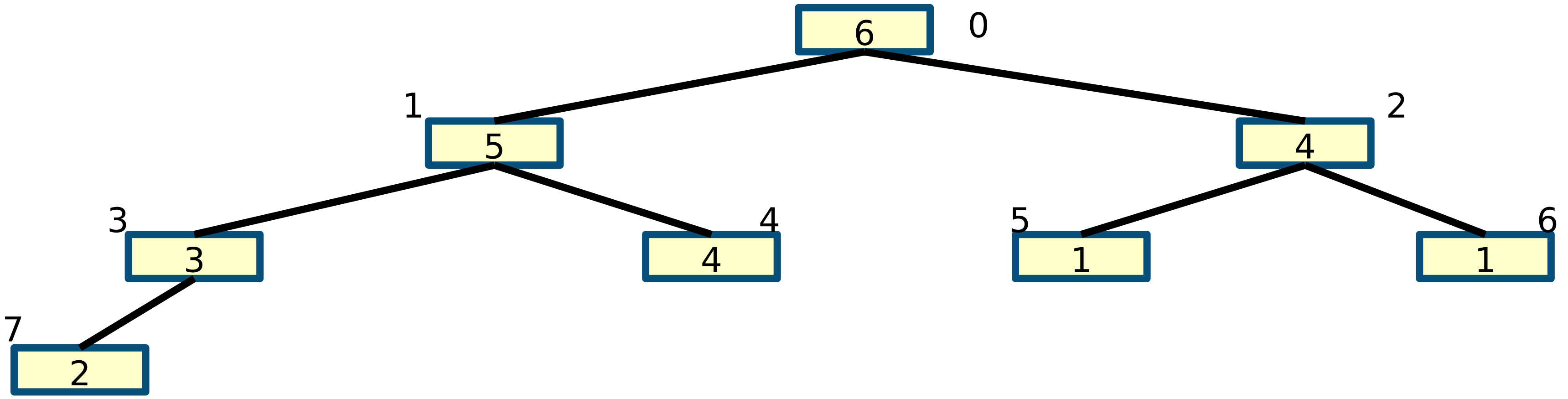

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

2	6	4	5	4	1	1	3	8	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

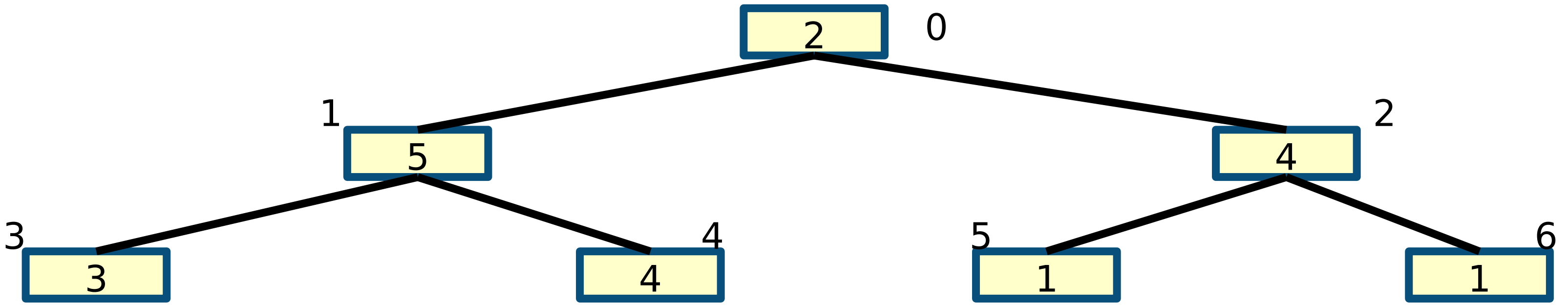



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

6	5	4	3	4	1	1	2	8	10	15
								i		

– Heap restored

Example execution of HEAPSORT(A)

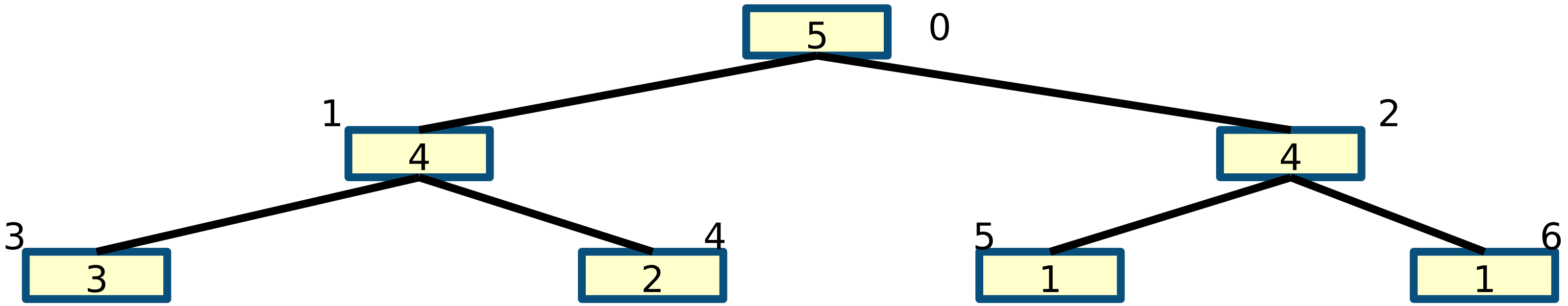



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

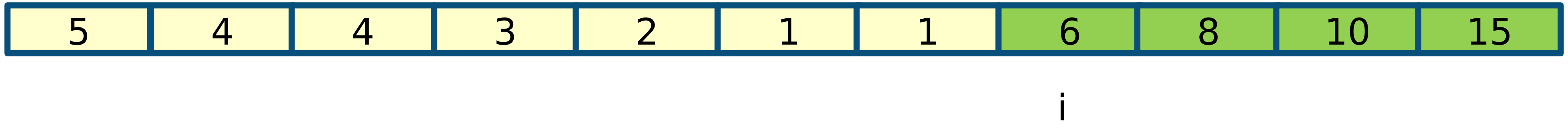
2	5	4	3	4	1	1	6	8	10	15
							i			

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

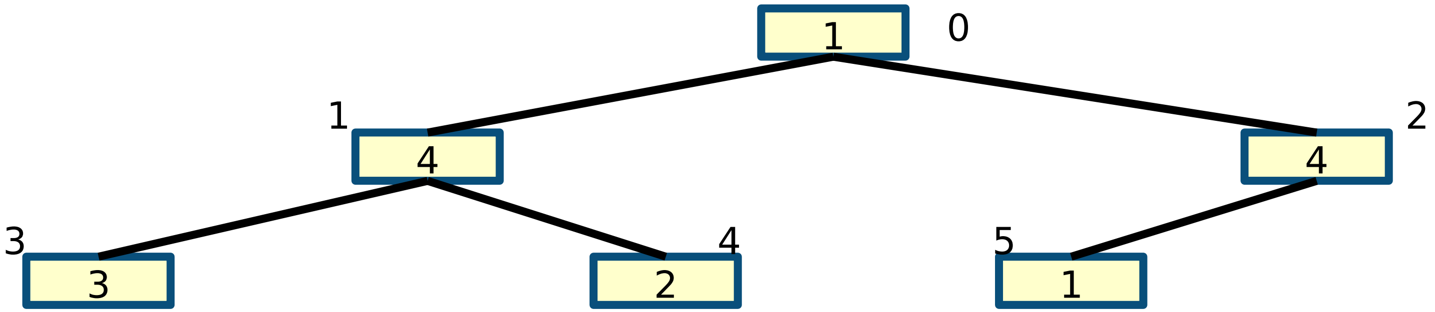



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```



- Heap restored

Example execution of HEAPSORT(A)



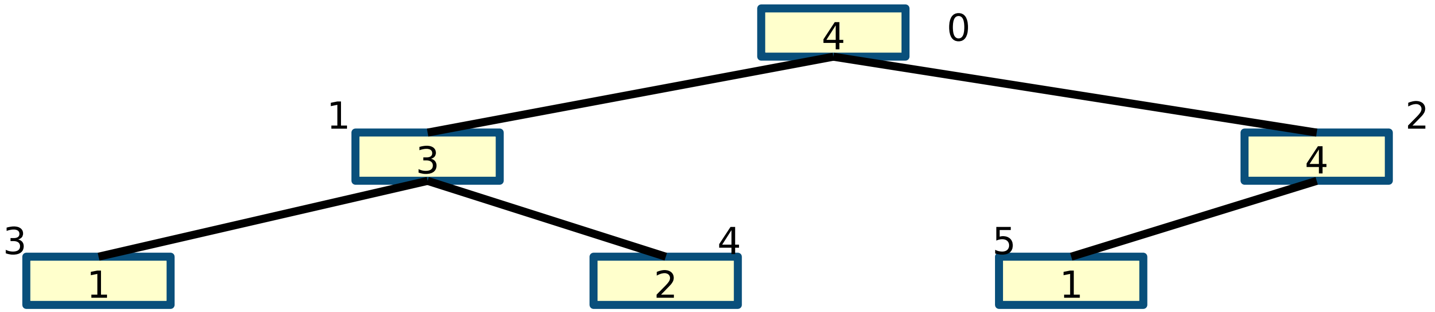

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

1	4	4	3	2	1	5	6	8	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

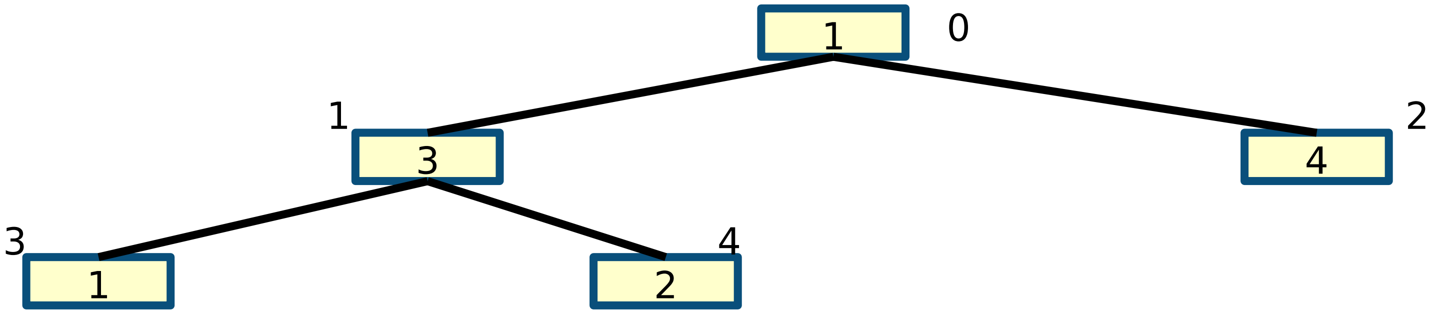



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

4	3	4	1	2	1	5	6	8	10	15
						i				

– Heap restored

Example execution of HEAPSORT(A)



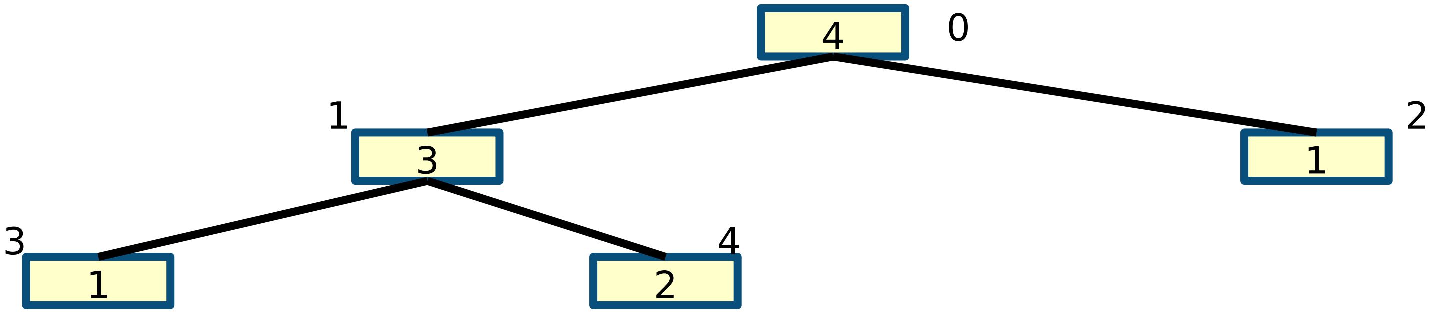

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

1	3	4	1	2	4	5	6	8	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)



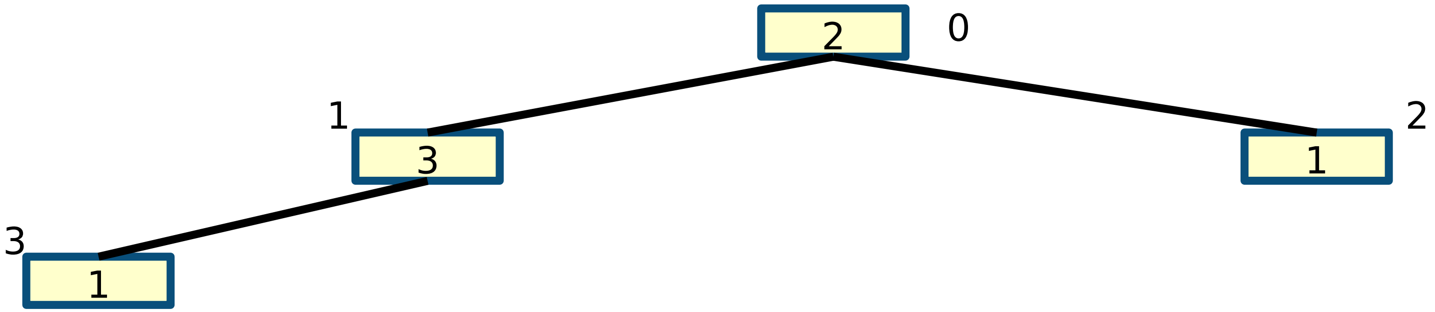

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

4	3	1	1	2	4	5	6	8	10	15

i

– Heap restored

Example execution of HEAPSORT(A)



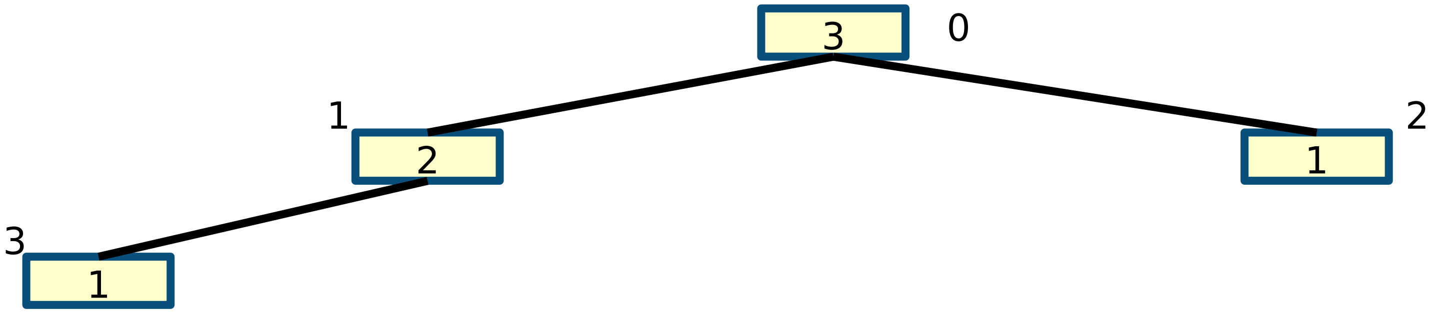

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

2	3	1	1	4	4	5	6	8	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

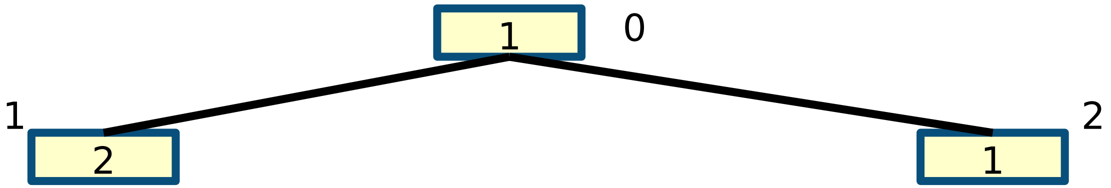



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

3	2	1	1	4	4	5	6	8	10	15
				i						

– Heap restored

Example execution of HEAPSORT(A)



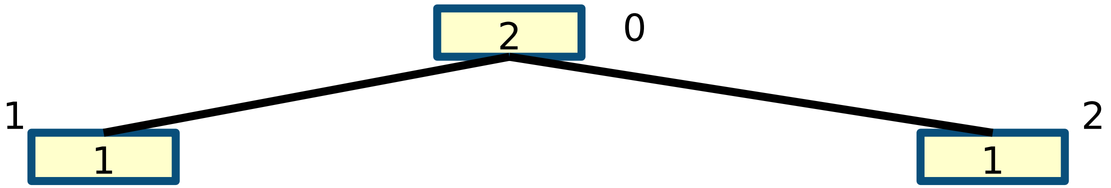

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

1	2	1	3	4	4	5	6	8	10	15

i

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)

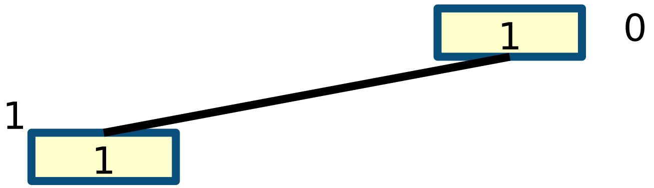



```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

2	1	1	3	4	4	5	6	8	10	15
			i							

– Heap restored

Example execution of HEAPSORT(A)




```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

1	1	2	3	4	4	5	6	8	10	15
		i								

- Swap A[0] with A[i]
- A[0] is the maximum so it is in its final position after the swap

Example execution of HEAPSORT(A)


```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

– Termination

MAX-HEAPIFY and BUILD-MAX-HEAP

```
LEFT(i)
  return (2 * i) + 1
```

```
RIGHT(i)
  return (2 * i) + 2
```

```
MAX-HEAPIFY(A,i,n)
  l := LEFT(i)
  r := RIGHT(i)
  if l < n and A[l] > A[i]
    largest := l
  else largest := i
  if r < n and A[r] > A[largest]
    largest := r
  if largest != i
    SWAP(A[i],A[largest])
    MAX-HEAPIFY(A,largest,n)
```

```
BUILD-MAX-HEAP(A)
  for i = (n / 2) - 1 downto 0
    MAX-HEAPIFY(A,i,n)
```

Running time

- **BUILD-MAX-HEAP is O(n)**
- **In HEAPSORT extracting the maximum takes O(1) (pick element A[0]) but we have to restore the heap at each step with MAX-HEAPIFY which takes O(log n)**
  - In SELECTION-SORT extracting the maximum element takes O(n)
- **There are n - 1 iterations of for loop**
- **T(n) = O(n) + O(n log n) = O(n log n)**
- **This applies both to the best and worst case**

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  s := n
  for i = n-1 downto 1
    SWAP(A[0],A[i])
    s := s - 1
    MAX-HEAPIFY(A,0,s)
```

Properties

- **In place: O(1) space complexity**
- **Not stable**

Comparison sorts

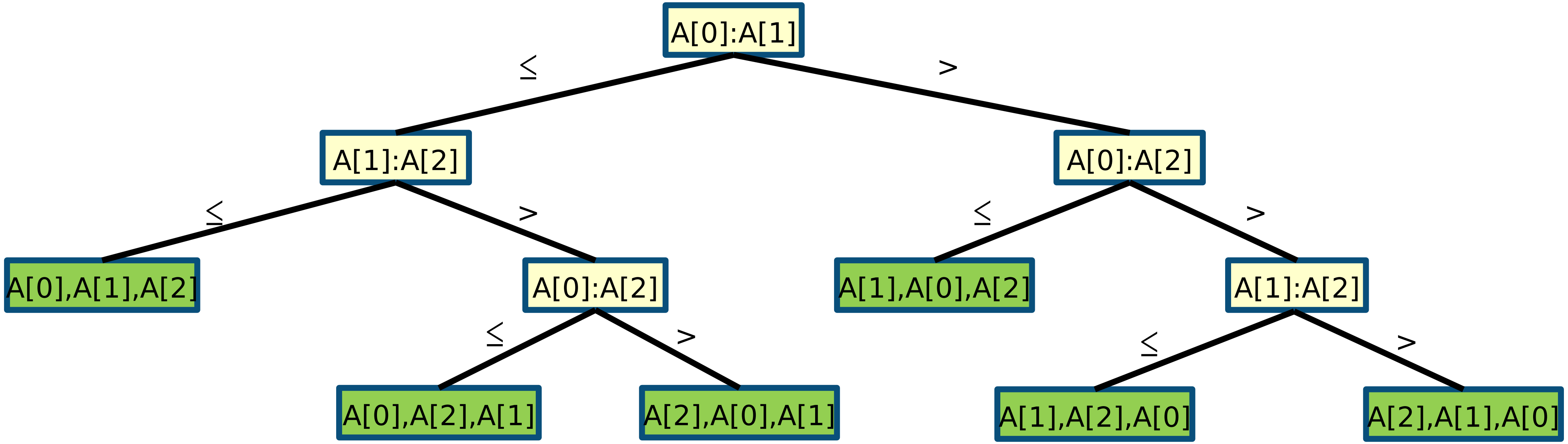
- **All the sorting algorithms we have studied so far are comparison sorts**
  - The sorted order is determined only by comparing the input elements

- **They can be studied abstractly using the decision tree model**
  - Consider only comparisons and ignore all other aspects of the algorithm
  - Introduced by Ford and Johnson in “A tournament problem”. The American Mathematical Monthly, 66(5):387–389, 1959.

- **A decision tree is a full binary tree that represents the comparisons between elements that are performed by a sorting algorithm operating on an input of a given size**
  - Assume all the input elements are distinct
  - Assume all comparisons have the form A[i] A[j]

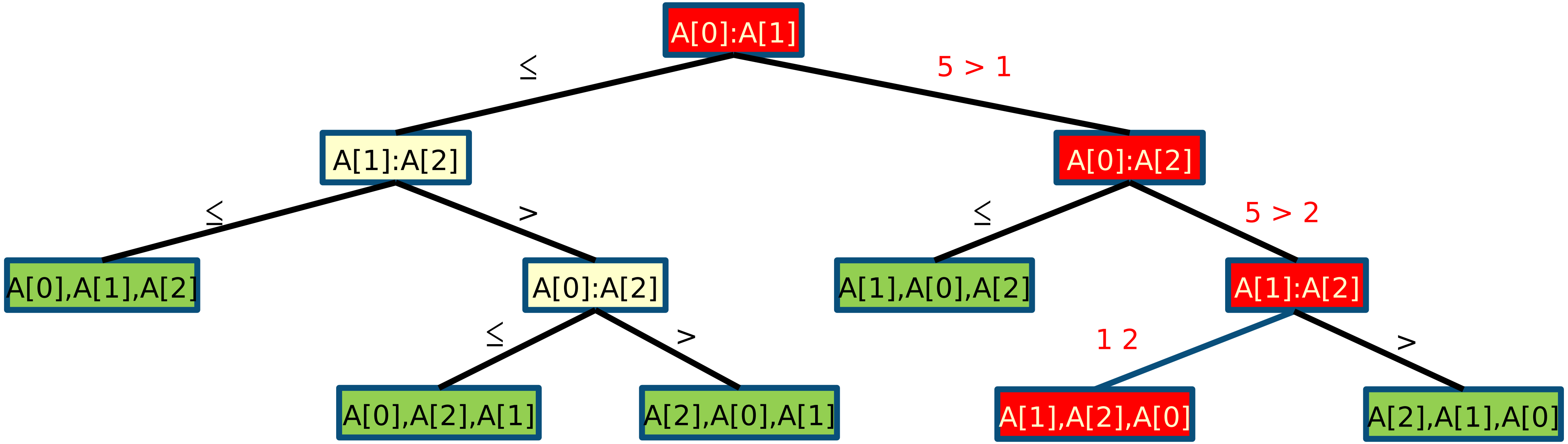
Decision tree example

- **Sort three elements A[0..2]**
  - Node A[i]:A[j] indicates a comparison between A[i] and A[j]
  - Green leaves are all the sorted permutations of the input



Decision tree example

- **The execution of an algorithm corresponds to tracing a path from the root to a leaf**
  - Example: the sorted permutation for A = [5,1,2] is (A[1],A[2],A[0])= [1,2,5]



Decision tree

- **Any correct sorting algorithm must be able to produce each permutation of its input**
  - $n!$ permutations on n elements
  - Therefore, there are $n!$ leaves in the decision tree

- **Each of leaf must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort**

- **The length of the longest path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the sorting algorithm performs**
  - This is also the height of the decision tree

A lower bound for comparison sort

- **We need to compute a lower bound on the height of the decision tree**
- **Recall the following facts**
  - A binary tree of height h has at most 2^h leaves
  - $n^{n/2}$ $n!$ n^n
- **Then we have**
  - $n!$ 2^h
  - $\log 2^h = h$ $\log n!$
  - $h \log n^{n/2} = n/2 \log n = \Omega(n \log n)$
- **Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case**
  - MERGE-SORT and HEAPSORT are asymptotically optimal
  - No comparison sort exists that is faster by more than a constant factor

Summary

- **HEAPSORT**
  - Heap data structure
  - Running time
  - Properties
- **Lower bounds for comparison sorts**
  - Decision tree model