ECE36800 Data structures

Heapsort and priority queues

Chapter 9 (pp. 361-403)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Simple selection sort
- Tournament sort
- Heap as a priority queue
- Heapsort

- Reference pages pp. 361-403

Selection sort in general

- Sort by selecting keys (and the associated record) and placing them in their proper (sorted) positions
- On each pass through an unsorted collection, pick the smallest (largest) element in this collection and move it to its proper position

Simple selection sort

Sort n integers r[0] to r[n-1] in ascending order

```
for i ← n-1 downto 1
  max_index ← 0
  for j ← 1 to i
    if r[j] ≥ r[max_index]
      max_index ← j
  r[i] ↔ r[max_index]
```

Selection

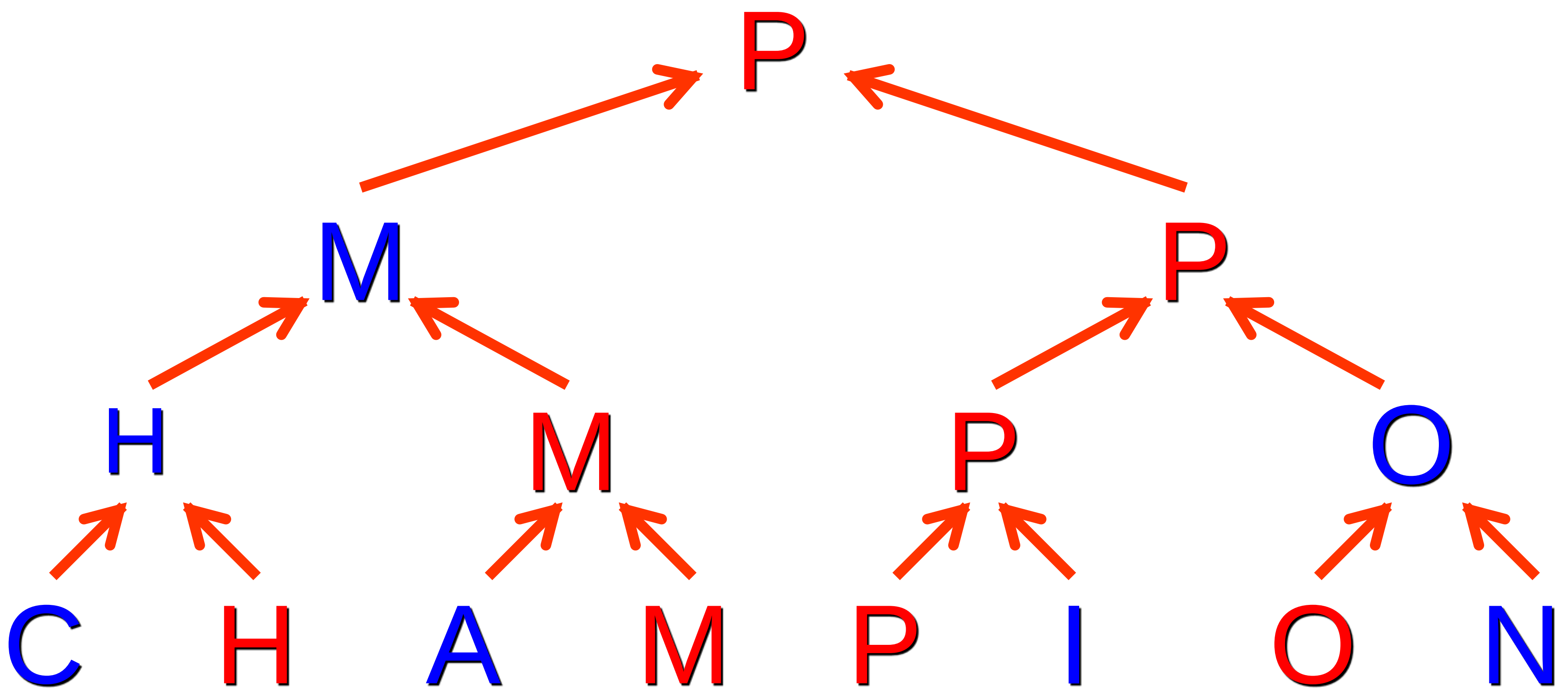
Simple selection sort

- Space complexity: $O(1)$, in-place
- Time complexity:
  - Number of swaps: $n – 1 = O(n)$
  - Number of comparisons: $(n – 1) + (n – 2) + … + 1 = n(n-1)/2 = O(n^2)$
- Not faster even for sorted input
- Simple insertion sort is preferred

Tournament sort: Playoff

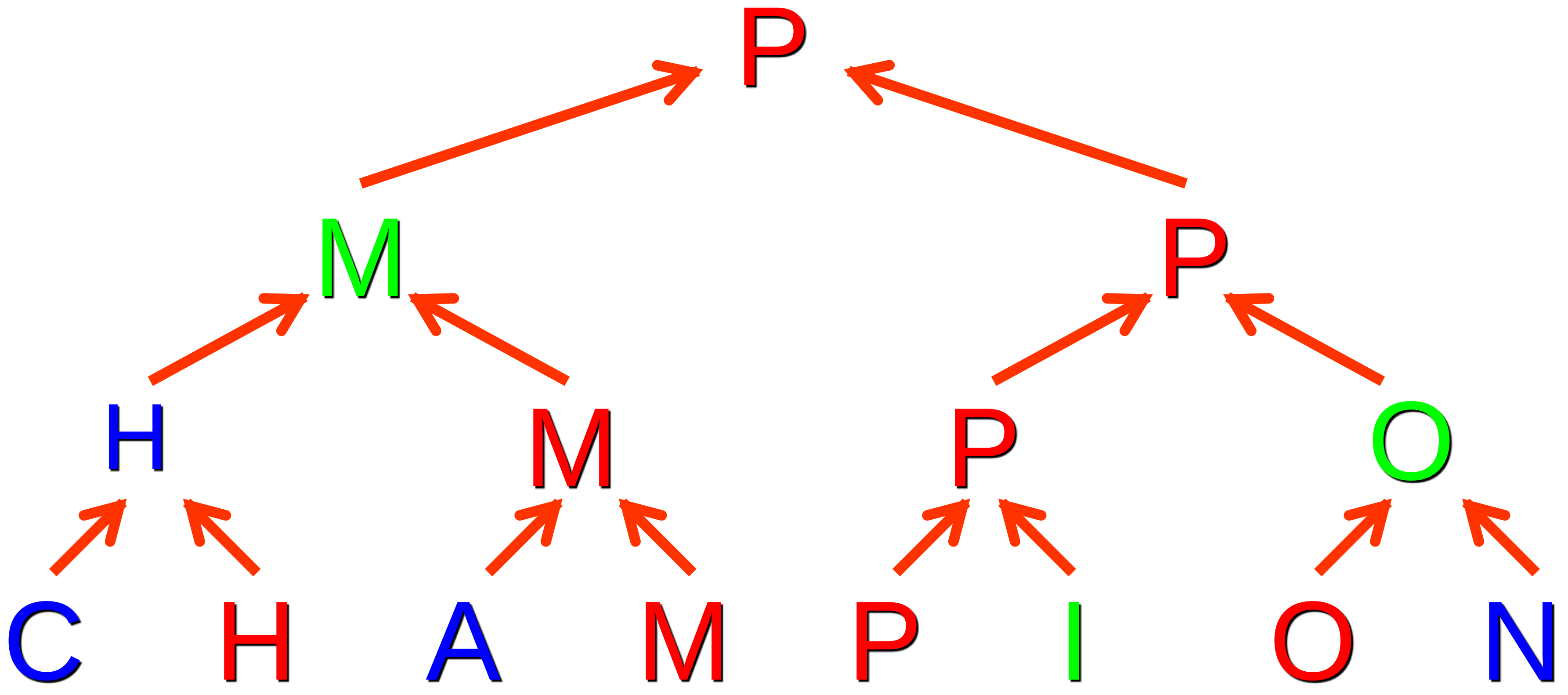
- Largest item has to go through O(n) comparisons in simple selection sort
- Playoff
  - Winners proceed to next round of tournament
  - The champion goes through O(log n) comparisons
  - Total number of comparisons: O(n)
- Everybody remembers the champion
  - Who is second, third, or … last?
  - How many comparisons to find out

Tournament sort: Champion



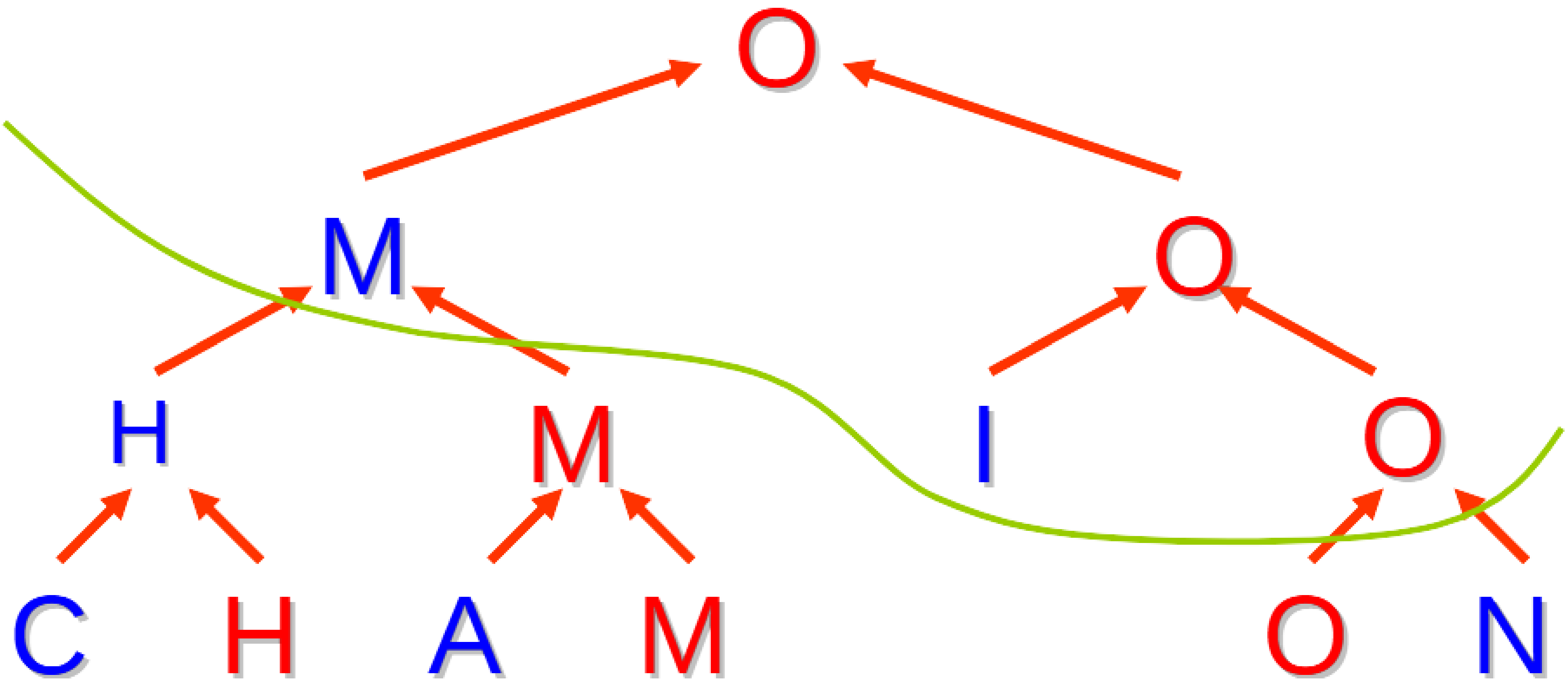
Tournament sort: Second

- Only those who lost to the champion can be second
- O(log n) lost to the champion



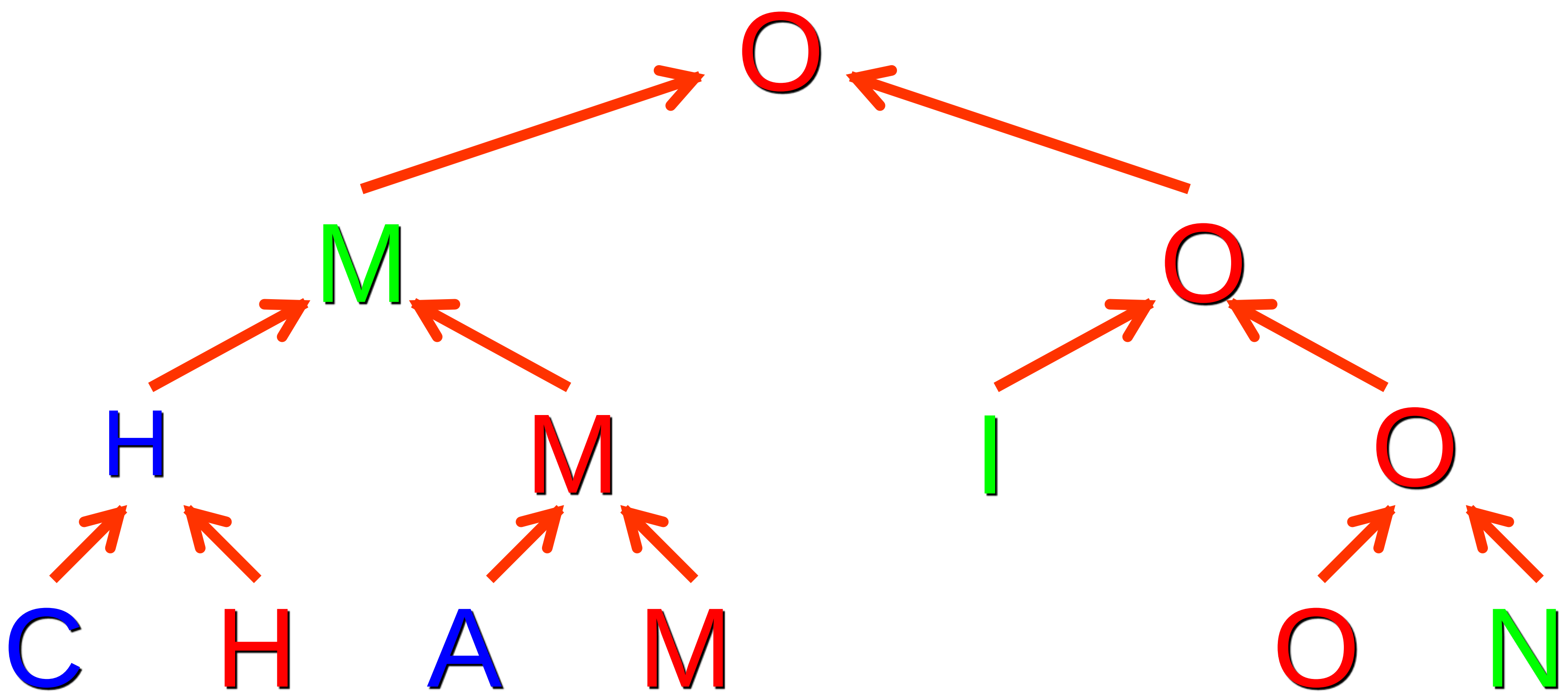
Tournament sort: Second

- Run a mini-tournament among them

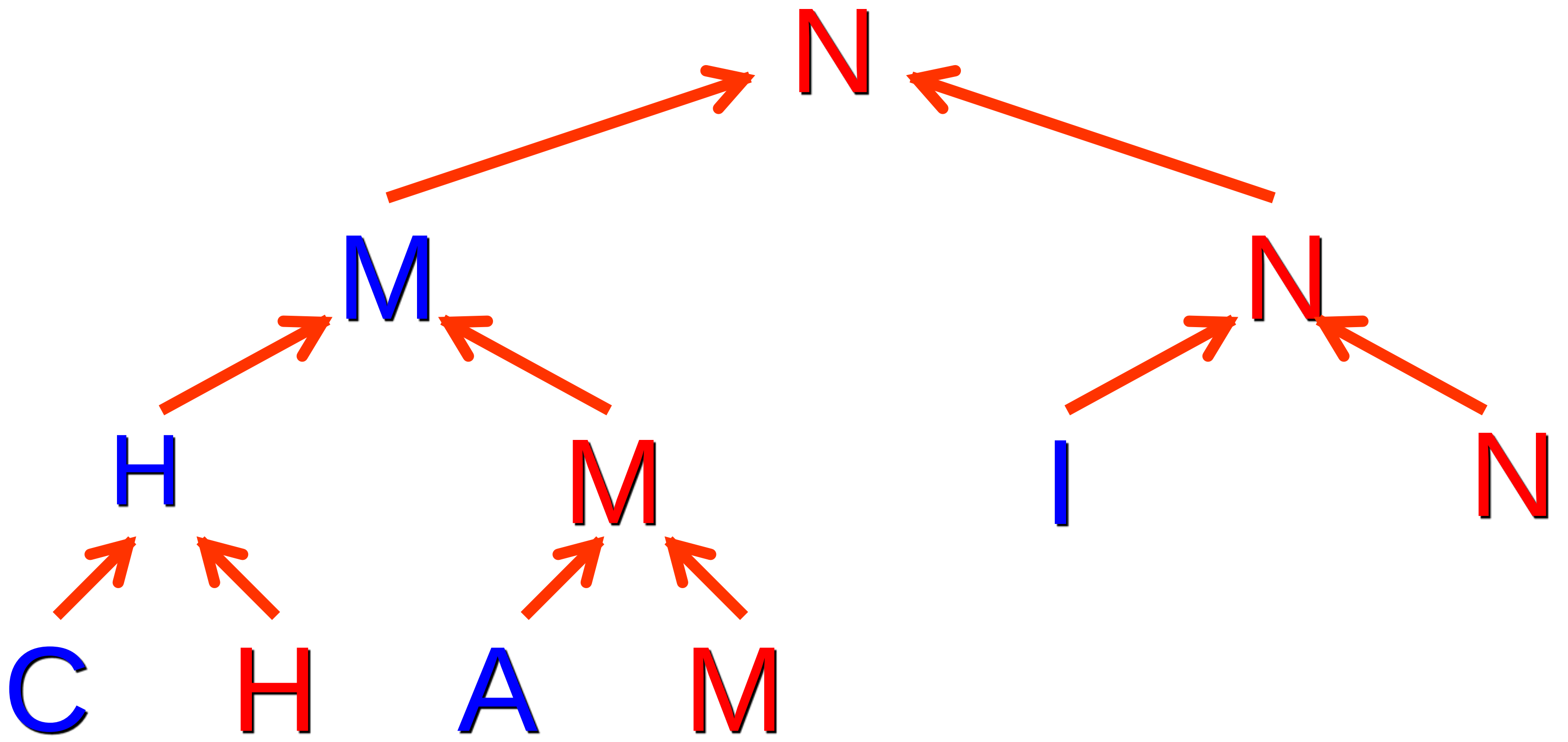


Tournament sort: Third

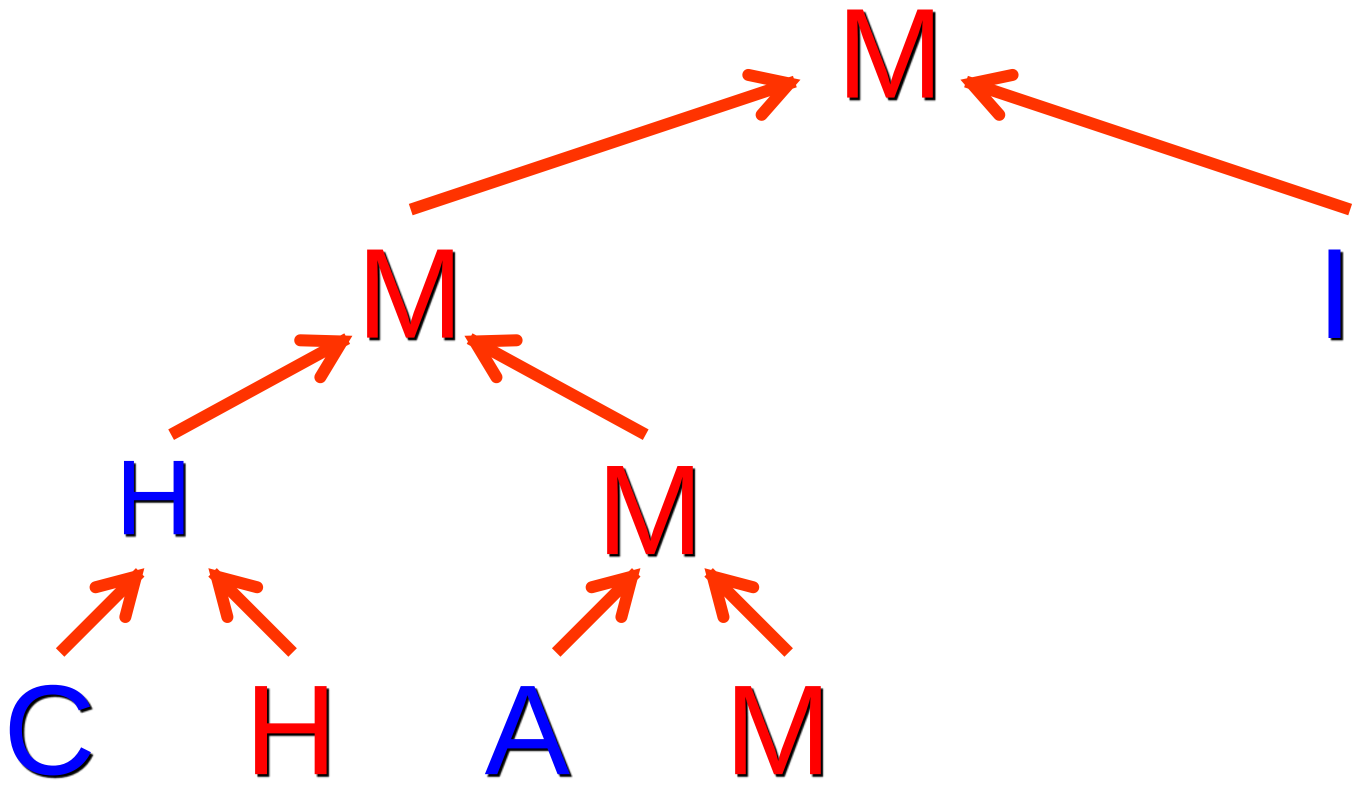
- Only those who lost to the second can be third



Tournament sort: Third



Tournament sort: Fourth



Tournament sort

- Finding the champion takes O(n) comparisons
- Finding the order of the rest takes O(n log n) comparisons
- Overall time complexity: O(n log n)
- O(n) space complexity because of the tournament tree

Heapsort

- Use a descending heap, a tree implemented using array
- Also called max-heap
- A max-heap is an almost complete binary tree (and sometimes a complete binary tree) where the value of each child is equal to or less than the value of its parent
- Ordering in a heap is similar in spirit to the ordering in a tournament tree
- Heap offers O(log n) insert and remove operations
  - It is a priority queue with O(log n) enqueue and dequeue operations
  - Array implementation of heap allows in-place sort

Heapsort

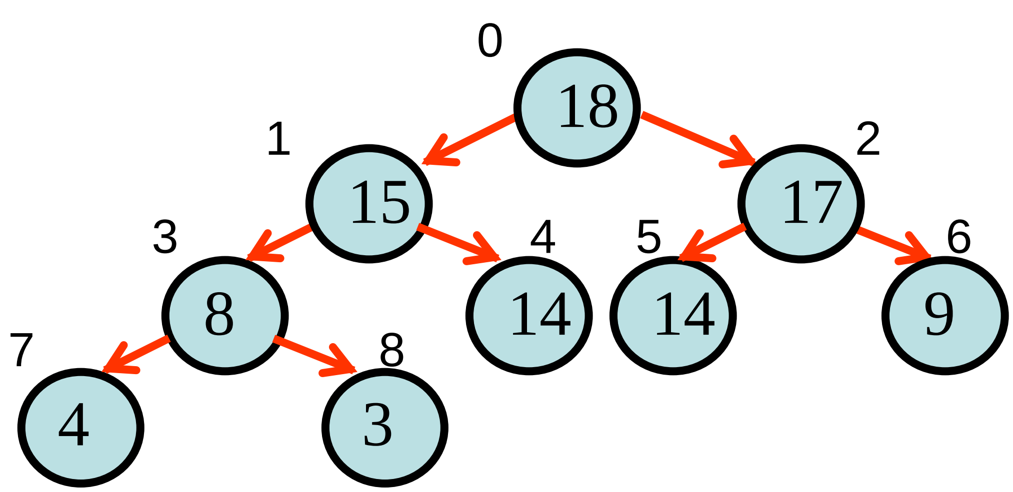
- Selection sort that uses a max-heap as a priority queue for sorting

```
Initialize(PQ) // max-heap as a PQ
for i ← 0 to n-1
  // insert r[i] with priority k[i]
  Enqueue(PQ, r[i], k[i])
for i ← n-1 downto 0
  r[i] ← Dequeue(PQ)
```

Max-heap: Tree as an array

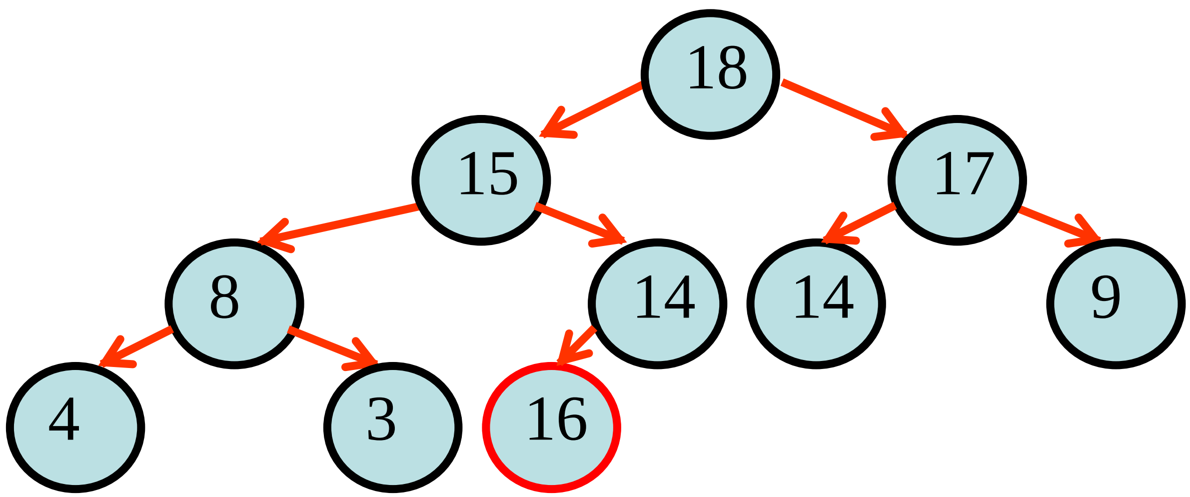
- Array is { 18, 15, 17, 8, 14, 14, 9, 4, 3 }
- Root node has index 0, and we increment the index from top to bottom, left to right
- For a tree with n nodes, node of index i, $0 \leq i < n$,
  - Parent of i > 0 has index (i – 1)/2; otherwise, no parent
  - Left of i has index (2*i + 1) if (2*i + 1) < n; otherwise, no left
  - Right of i has index (2*i + 2) if (2*i + 2) < n; otherwise, no right

- The value of each child is equal to or less than the value of its parent
- Assume a variable to store heap size or size of priority queue



Max-heap: Enqueue element

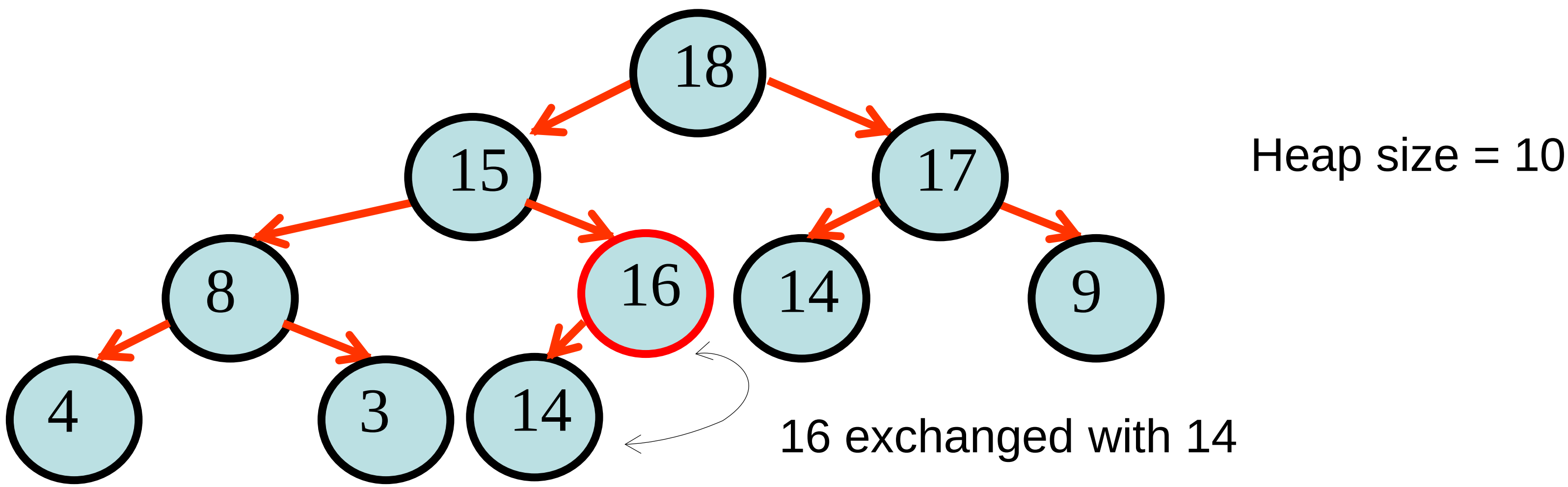
- Enqueue new element: 16
  - Assume array is large enough
  - Increment heap size
  - Put the new element at the end of the array (the next leaf node):



Heap size = 10

Max-heap: Upward heapify or sifting up

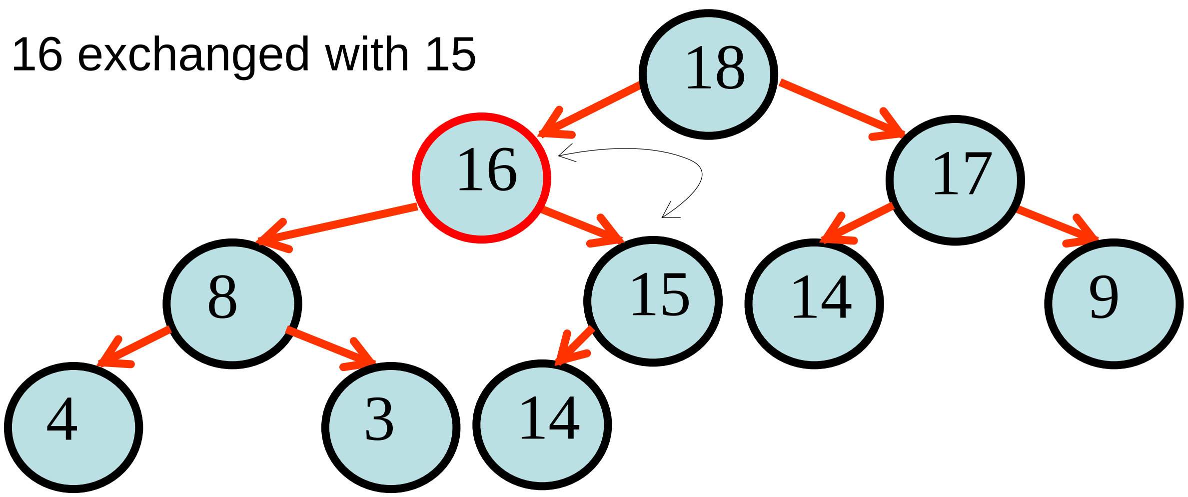
- Enqueue new element: 16
  - As long as it is larger than its parent, if a parent exists, exchange positions with the parent
  - Similar to one pass of insertion sort, where we insert the new element into the correct position along the path from the leaf node to the root



Max-heap: Upward heapify or sifting up

- New array = { 18, 16, 17, 8, 15, 14, 9, 4, 3, 14 }

16 exchanged with 15



Heap size = 10

Heap as a priority queue

- Max-heap of heap size n currently stored in positions array[0, …, n-1] (the size of actual array is larger than n)
- To enqueue, increment heap size, store the new item at location n
- Perform Upward_heapify(array[], n) to restore max-heap property
  - Note the similarity between Upward_heapify and one pass of insertion sort

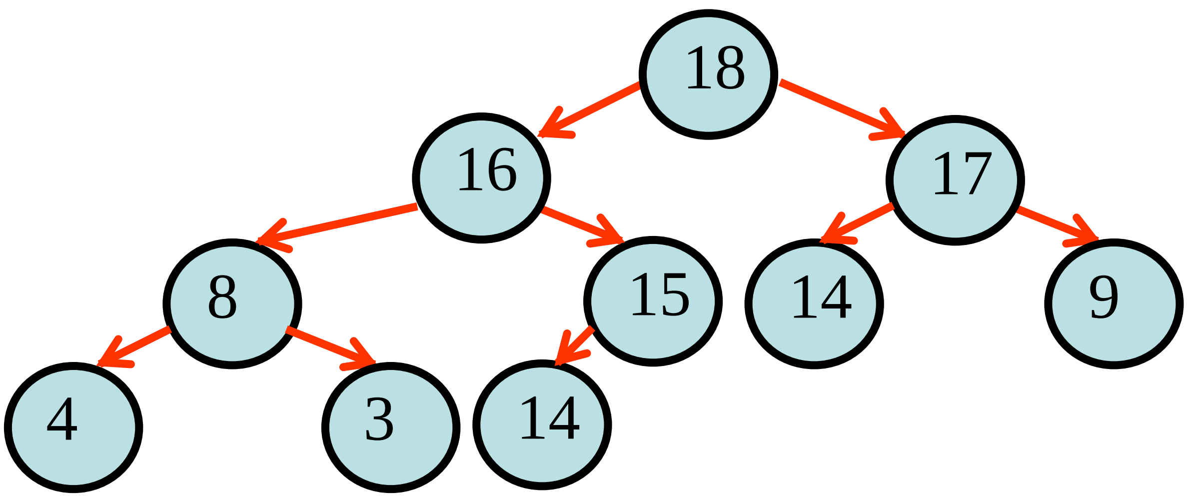
```
Upward_heapify(array[], n):
  new ← array[n]
  child ← n
  parent ← (child-1)/2
  while child > 0 and array[parent] < new
    array[child] ← array[parent]
    child ← parent
    parent ← (child-1)/2
  array[child] ← new
```

Enqueue time complexity

- Traverse all the way to the root (new element is the new maximum)
- Overall time complexity of enqueue operation is O(log n)
- Overall space complexity of enqueue operation is O(1)

Max-heap: Dequeue operation

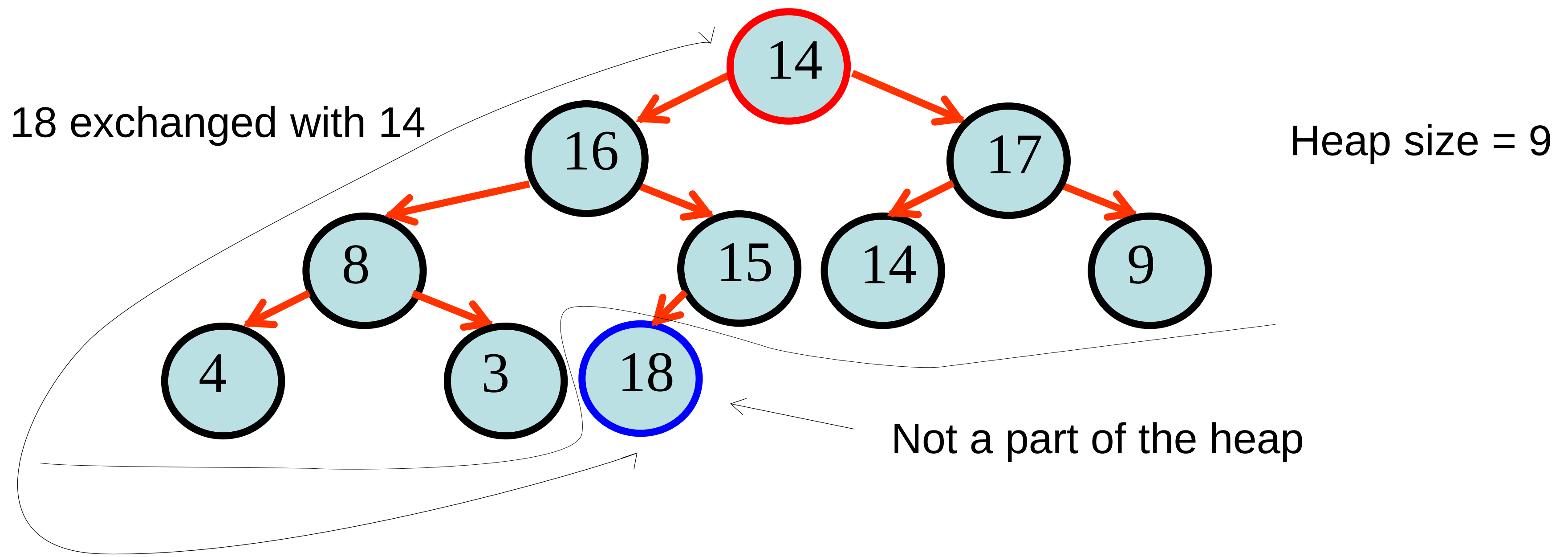
- Array = { 18, 16, 17, 8, 15, 14, 9, 4, 3, 14 }
- Dequeue operation has to remove 18, the element stored at the root node



Heap size = 10

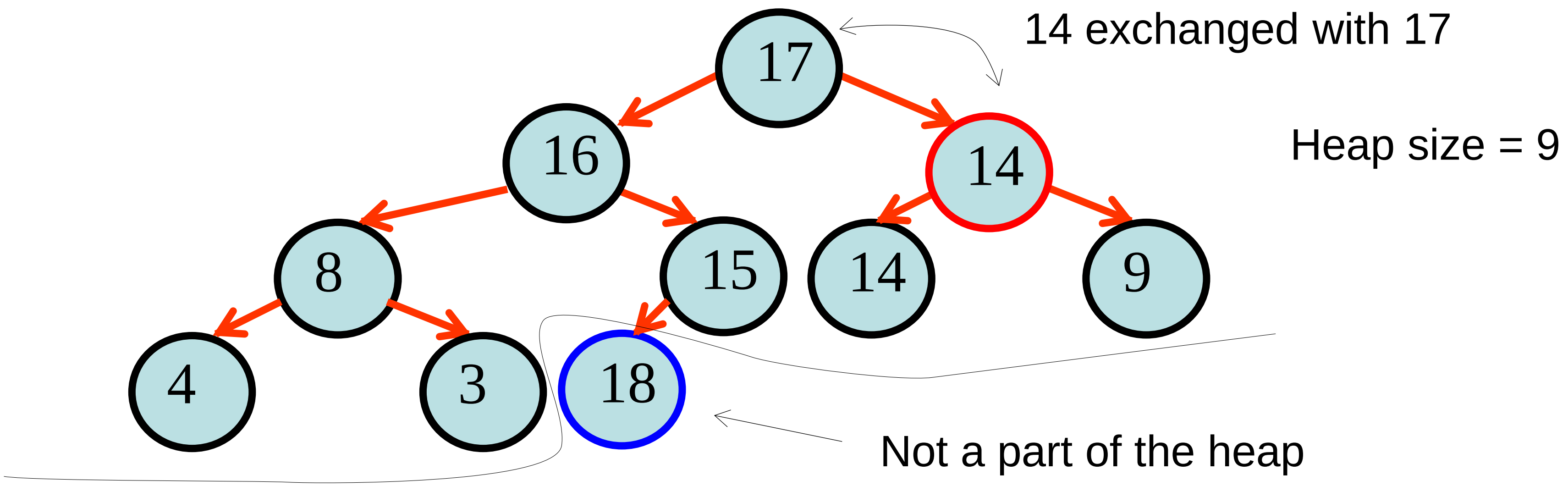
Max-heap: Dequeue operation

- Exchange contents of root node and the last leaf node
- Decrement the heap
- Valid part of the array to store the heap contents becomes { 14, 16, 17, 8, 15, 14, 9, 4, 3 }
  - array[Heap size] serves as a temporary variable to store the max element



Max-heap: Downward heapify or sifting down

- If the new value at the root node is less than its larger child, exchange with the larger child
- Keep "sifting down" the new value by exchanging with its larger child as long as the larger child is greater
  - Stop when there is no larger child
- New array: { 17, 16, 14, 8, 15, 14, 9, 4, 3 }
- Return the element at array[Heap size] as the dequeued element



Downward_heapify

- Assume that the left tree and right tree of the entry at position i in array[0, …, n] are max-heaps
  - n+1 valid entries in array
  - It has a flavor of insertion sort, inserting the entry currently at i into an appropriate position along a path to a leaf node
  - It also has a hint of binary search to choose the path of larger children

```
Downward_heapify(array[], i, n):
  temp ← array[i];
  while ((j = 2*i+1) ≤ n)
    if (j < n and array[j] < array[j+1])
      j ← j+1
    if (temp ≥ array[j])
      break
    else
      array[i] ← array[j]
      i ← j
  array[i] ← temp
```

Binary search

Insertion sort

Dequeue time complexity

- Traverse all the way to the leaf
- Overall time complexity of dequeue operation is O(log n)
- Overall space complexity of dequeue operation is O(1)

Heapsort: Sorting using a max-heap

- With a max-heap as descending priority queue, the algorithm is just the general selection sort
  - There is no need to separately maintain the heap size variable because the variable i serves a double duty

```
for i ← 1 to n-1
  Upward_heapify(r[], i)
for i ← n-1 downto 1
  r[i] ↔ r[0]
  Downward_heapify(r[], 0, i-1)
```

Enqueue n-1 times to build max-heap

Dequeue n-1 times

- Heapsort uses O(1) memory, therefore in-place

Heapsort time complexity

- Worst-case time complexity
  - Enqueueing one element when there are i elements in heap: O(log i) = O(log n)
  - Dequeueing when there are i elements in heap: O(log i) = O(log n)
  - Enqueueing n – 1 times and dequeueing n – 1 times, heapsort has complexity O(n log n)
- Best-case time complexity when all elements are the same
  - Enqueue takes O(1) and dequeue takes O(1)
  - Overall complexity is O(n)

Building max-heap more efficiently

- Use Downward_heapify to build max-heap
  - The leaf nodes are all max-heaps
  - Start from the bottom-most, right-most node with a child (or children), apply Downward_heapify from bottom to top, right to left
  - Takes O(n) to build the max-heap
  - Overall complexity of heapsort is still O(n log n)

```
for i ← n/2 - 1 downto 0
  Downward_heapify(r[], i, n-1)
for i ← n-1 downto 1
  r[i] ↔ r[0]
  Downward_heapify(r[], 0, i-1)
```

Build max-heap

Dequeue n-1 times