



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Chapter 5:

Heaps, priority queues, and Heapsort

Keep the maximum accessible without repeated full scans

Finding the maximum of an unsorted array can require inspecting all n records. Repeating that scan after each removal leads to quadratic work.

Give the array useful structure

A max-heap keeps a maximum at a known position. After removal, repair follows one short path. A min-heap provides the symmetric service for a minimum.

We will introduce tree vocabulary, encode a nearly complete tree in an array, add heap order, and study repair and construction before priority queues and Heapsort.

The tree view explains the operations; the array view explains their implementation.

A binary heap combines shape, order, and array storage

Shape A nearly complete binary tree.

Order A local comparison between every parent and its children.

Storage One compact array, with no explicit child pointers.

Heap at a glance

In a max-heap, each parent is at least as large as its children, so a maximum is at the root. A min-heap reverses the comparisons and keeps a minimum at the root.

Here **heap** names a data structure. It is distinct from the memory region used for dynamic allocation or garbage collection.

We first define the shape and its representation, then add the ordering rule.

A binary tree distinguishes left and right children

Definition (Binary tree)

A *binary tree* is a rooted tree in which every node has at most two children, distinguished as its *left child* and *right child*.

- The **root** is the node with no parent.
- A **leaf** has no children; all other nodes are **internal nodes**.
- The **subtree** rooted at a node contains that node and all its descendants.

An arbitrary binary tree can have missing children in many places. Direct array layouts may then contain unused gaps.

Depth counts from the root; height counts toward a leaf

Depth of a node

The number of edges from the root to that node. The root has depth zero.

Height of a node

The number of edges on a longest downward path from that node to a leaf. Every leaf has height zero.

The height of the tree is the height of its root.

Depth and height measure different paths. Height will bound how far a displaced record can move downward.

A nearly complete shape leaves no gaps

An arbitrary binary tree may have missing children in many places, leaving gaps in a direct array layout.

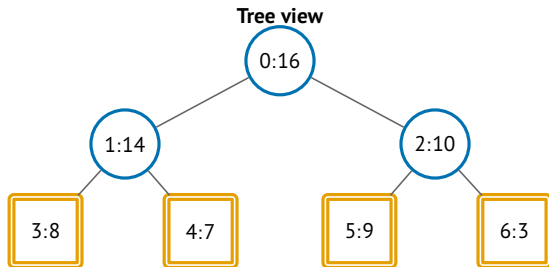
Definition (Nearly complete binary tree)

A binary tree is *nearly complete* if every level except possibly the last is full and the nodes of the last level occupy the leftmost available positions.

Read each level from left to right, starting at the root. Only the right end of the last level can be missing, so the array needs no gaps.

This shape makes explicit child links unnecessary.

Read the same seven records level by level



Each label $i : v$ gives an array index i and the value $v = A[i]$. The tree and array are two views of the same records, not two copies.

Lines show parent-child relationships determined by indices. They are not explicit pointers stored by the implementation.

Keep array indices separate from stored values

	Array view						
index i	0	1	2	3	4	5	6
value $A[i]$	16	14	10	8	7	9	3

The levels contain [16], then [14, 10], then [8, 7, 9, 3]. Appending them gives the displayed array.

The top row gives addresses; the bottom row gives stored values.

In the tree, a label $i : v$ means that index i stores value $v = A[i]$. For example, $2 : 10$ means $A[2] = 10$.

The index is part of the representation, not part of the key.

Compute relatives from zero-based indices

For a node at array position i ,

$$\text{left}(i) = 2i + 1, \quad \text{right}(i) = 2i + 2,$$

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \quad (i > 0).$$

- Position 0 is the root and has no parent.
- A child exists only if its computed index is less than `heap_size`.
- Logical heap size and physical array length are different quantities.

Locate the relatives of 2 : 10

Its children are at $2(2) + 1 = 5$ and $2(2) + 2 = 6$, storing 9 and 3. Its parent is at $\lfloor (2-1)/2 \rfloor = 0$, storing 16.

The active heap can shrink inside a fixed array

`length(A)` The number of physical array positions.

`heap_size` The number of positions that currently belong to the heap.

Initially these may be equal. In Heapsort, only the logical heap size decreases:

$$\underbrace{A[0 \dots \text{heap_size} - 1]}_{\text{active heap}} \quad \underbrace{A[\text{heap_size} \dots \text{length}(A) - 1]}_{\text{sorted suffix}}.$$

Released positions stay in the same array. They contain final records and no longer participate in heap repair.

Nearly complete trees have logarithmic height

Height counts edges on a longest downward path; a leaf has height zero.

Lemma (Height of a binary heap)

A nearly complete tree containing $n \geq 1$ nodes has height $\lfloor \log_2 n \rfloor$.

All levels above the last are full. For height h ,

$$2^h \leq n \leq 2^{h+1} - 1 \implies h = \lfloor \log_2 n \rfloor.$$

An operation following one root-to-leaf or leaf-to-root path therefore visits only $O(\log n)$ levels.

Heap order compares each child with its parent

Definition (Max-heap and min-heap)

A *max-heap* is a nearly complete binary tree, stored in the level-order array described above, in which every non-root position i satisfies

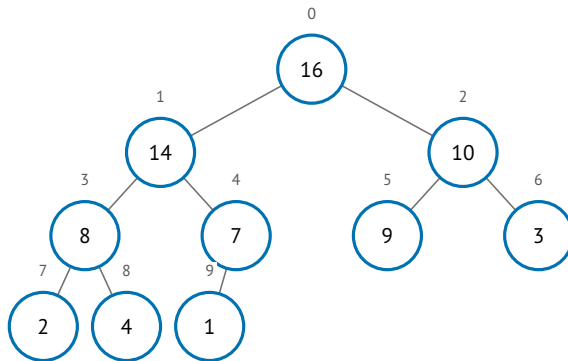
$$A[\text{parent}(i)] \geq A[i].$$

A *min-heap* has the same shape and reverses the inequality.

Here stored values are compared. For general records, compare their keys, as in the pseudocode.

Following parents can only increase or preserve the value, so the root contains a maximum. Siblings and different subtrees need not be ordered.

Check heap order along every parent-child edge



zero-based index above each node; stored value inside

$$A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$$

The parent 14 dominates 8 and 7; the parent 10 dominates 9 and 3. Every other edge satisfies the same rule. The root stores the maximum, 16.

A heap exposes an extreme key without sorting everything

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

is a max-heap, but it is not sorted.

Two independent requirements

Shape gives compact array storage and logarithmic height.

Parent-child order exposes the maximum or minimum. Neither condition implies the other.

The values 7 and 9 lie in different subtrees and need not occur in numerical order. The array is a level-order traversal, not a sorted sequence.

A heap is not a binary search tree: arbitrary-key search takes $\Theta(n)$ time in the worst case.

Move a small parent down through the larger child

Precondition for downward repair

The left and right child subtrees of i are already max-heaps. The only possible violation is at their common parent.

1. Compare the parent with its existing children.
2. If a child is larger, exchange the parent with the larger child.
3. Continue only in the subtree that received the displaced record.

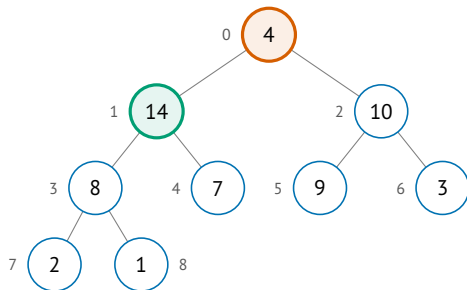
Choose the larger child so that the new parent dominates both children. The untouched subtree remains a heap. If the parent is already largest, stop.

Downward repair follows one recursive branch

```
Max-heapify(A, i, heap_size)
  left <- 2*i+1
  right <- 2*i+2
  largest <- i
  if left < heap_size and key(A[left]) > key(A[largest])
    largest <- left
  if right < heap_size and key(A[right]) > key(A[largest])
    largest <- right
  if largest != i
    exchange A[i] and A[largest]
    Max-heapify(A, largest, heap_size)
```

Evaluate each and from left to right: check the index before accessing the array.

At the root, choose the larger child



Both child subtrees are already max-heaps. Only the root value 4 violates heap order.

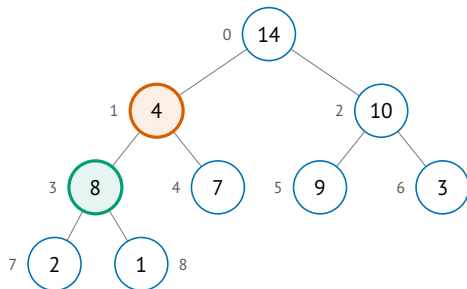
Compare its children, 14 and 10. Choose 14 at position 1, then exchange positions 0 and 1.

The new root dominates both children. Only the subtree receiving 4 can still contain a violation.

4 displaced; choose child 14
index outside; value inside

$$A^{(0)} = [4, 14, 10, 8, 7, 9, 3, 2, 1].$$

Continue only where the displaced value moved



4 displaced; choose child 8
index outside; value inside

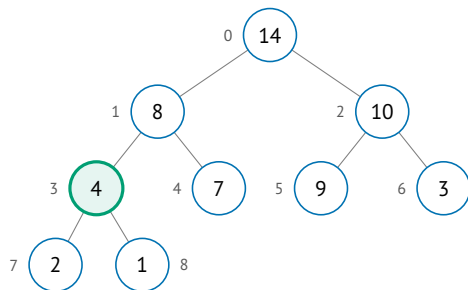
$$A^{(1)} = [14, 4, 10, 8, 7, 9, 3, 2, 1].$$

After exchanging positions 0 and 1, the root is correct. The displaced 4 is now at position 1.

Its children are 8 and 7. Choose 8 at position 3, then exchange positions 1 and 3.

The other subtree is untouched. Any remaining violation follows 4 to position 3.

Stop when the displaced value dominates its children



4 is in its final repair position
index outside; value inside

$$A^{(2)} = [14, 8, 10, 4, 7, 9, 3, 2, 1].$$

After exchanging positions 1 and 3, the displaced 4 reaches position 3.

Its children are 2 and 1. Since $4 \geq 2$ and $4 \geq 1$, repair stops.

The whole array is now a max-heap. Repair followed only the path $0 \rightarrow 1 \rightarrow 3$.

Repair cost is bounded by one path

Let n be the number of occupied records. Count key comparisons, assignments, and array accesses as constant-time operations.

Each exchange moves the displaced record down one level. Root repair takes $O(\log n)$ time.

Implementation and auxiliary space

The displayed recursive procedure may use $O(\log n)$ stack space. An iterative implementation follows the same path with $\Theta(1)$ auxiliary space.

Heapify-up repairs a key that has become larger than its parent: exchange with the parent and repeat upward.

Upward repair also takes $O(\log n)$ time.

Build child heaps before repairing their parent

One root repair cannot build a heap from an arbitrary array: the two child subtrees may violate its precondition.

Start where heap order is already valid

For an array of length n , positions $\lfloor n/2 \rfloor, \dots, n-1$ are leaves. Every leaf is already a heap.

Process internal nodes from $\lfloor n/2 \rfloor - 1$ down to 0.

Children have larger indices, so both child subtrees are heaps when their parent is processed.

Bottom-up construction finishes at the root

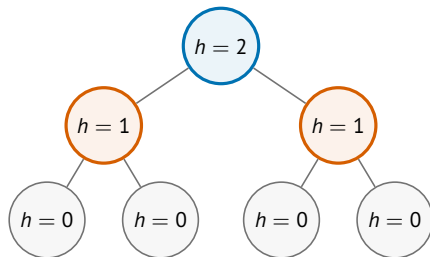
```
Build-max-heap(A)
  heap_size <- length(A)
  for i <- floor(length(A)/2)-1 downto 0
    Max-heapify(A, i, heap_size)
  return heap_size
```

Each call combines two child heaps with their parent into a larger heap.

For $n \leq 1$, the loop is empty; otherwise the last call makes the entire array a heap.

Most nodes can move down only a short distance

Possible repair distance



Each label gives the node height h , the maximum number of downward steps available to a repair. Among these seven nodes, four are leaves, two have height one, and only the root has height two.

Count nodes at each possible repair height

Height	Nodes	Maximum steps	Upper bound
0	4	0	$4 \cdot 0 = 0$
1	2	1	$2 \cdot 1 = 2$
2	1	2	$1 \cdot 2 = 2$

$$4 \cdot 0 + 2 \cdot 1 + 1 \cdot 2 = 4 \text{ possible downward steps.}$$

This is an upper-bound accounting by height, not an exact comparison count for one arrangement of keys.

Constant checks still take time, even when no downward step occurs.

Sum short repairs instead of charging every node full height

The product $n \cdot O(\log n)$ is valid but loose. At most $\lceil n/2^{h+1} \rceil$ nodes have height h .

A repair at height h costs $O(h + 1)$, including its constant checks. Let $H = \lfloor \log_2 n \rfloor$.

$$\begin{aligned} T_{\text{build}}(n) &\leq c \sum_{h=0}^H \left\lceil \frac{n}{2^{h+1}} \right\rceil (h + 1) \\ &\leq 2cn \sum_{h=0}^{\infty} \frac{h + 1}{2^{h+1}} = O(n). \end{aligned}$$

For $h \leq H$, $n/2^{h+1} \geq 1/2$, so $\lceil x \rceil \leq 2x$ justifies removing the ceiling.

The loop makes $\lfloor n/2 \rfloor$ calls with constant overhead each, giving $\Omega(n)$. Thus construction takes $\Theta(n)$ time.

A priority queue retrieves by urgency rather than arrival

Definition (Priority queue)

A max-priority queue supports

Maximum, Extract-max, Insert, Increase-key.

A min-priority queue provides the symmetric minimum-oriented operations.

- Maximum reads the root of a nonempty max-heap in $\Theta(1)$ time.
- Extract-max removes that record and restores a valid heap.

Examples include the next event in a simulation and the next tentative vertex in a graph algorithm. The abstract operations specify the task; a binary heap is one representation.

Replace the root, shorten the heap, and repair

```
Heap-extract-max(A, heap_size)
  if heap_size = 0
    error Empty-heap
  maximum <- A[0]
  A[0] <- A[heap_size-1]
  heap_size <- heap_size-1
  if heap_size > 0
    Max-heapify(A, 0, heap_size)
  return (maximum, heap_size)
```

The physical array stays allocated. With one occupied record, extraction leaves an empty heap and skips repair.

Insertions and key increases use upward repair

Insert

Place the new record in the next free leaf position, extend the logical heap, and repair upward.

Increase-key

At a known heap position, increase the key and exchange with its parent while the parent is smaller.

Each operation takes $O(\log n)$ time for the heap work.

Insertion assumes available capacity; increasing a key assumes that its heap position is known.

Choose a representation for the operation mix

Worst-case costs on n stored records:

Operation	Unsorted array	Sorted array	Binary heap
Find maximum	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Extract maximum	$\Theta(n)$	$\Theta(1)$	$O(\log n)$
Insert	$\Theta(1)$	$\Theta(n)$	$O(\log n)$
Search arbitrary key	$\Theta(n)$	$O(\log n)$	$\Theta(n)$

Assume a free array slot for insertion, no resizing, and the maximum at the occupied end of the sorted array.

A heap supports repeated extreme-key access well; it is not a general ordered-search structure.

Account for capacity and record locations

Available space is an assumption

Constant-time insertion into an unsorted array assumes a free slot. Resizing a full dynamic array needs amortized analysis, developed in Module B.

An identifier is not an array index

Applications that update priorities by record identifier often keep a separate map from identifiers to heap positions. Every heap exchange must update that map.

Include the cost of finding the position when assessing a complete priority update.

Grow a sorted suffix outside the active heap

Build a max-heap, then repeatedly move its root maximum to the last active position.

1. Exchange the root with the last record of the active heap.
2. Reduce `heap_size` by one: that record is now final.
3. Repair the new root to restore heap order in the active prefix.

One array, two regions

$$\underbrace{A[0 \dots \text{heap_size} - 1]}_{\text{active max-heap}} \quad \underbrace{A[\text{heap_size} \dots n - 1]}_{\text{sorted final suffix}}.$$

The physical length n does not change.

Heapsort repeatedly places the active maximum

```
Heapsort(A)
  heap_size <- Build-max-heap(A)
  for end <- length(A)-1 downto 1
    exchange A[0] and A[end]
    heap_size <- heap_size-1
    Max-heapify(A, 0, heap_size)
```

The active heap and the final suffix share the same array.

Empty and one-record inputs execute no sorting-loop iterations.

Each iteration fixes one more largest record

At the start of each iteration

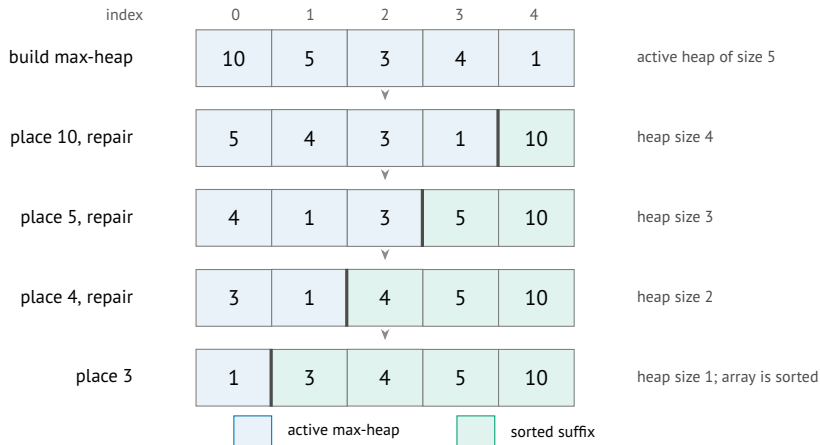
The prefix $A[0 \dots \text{heap_size} - 1]$ is a max-heap. The suffix contains the largest records in final nondecreasing order.

The root is the largest record not yet placed. Swapping it with the last heap position extends the final suffix by one.

Root repair restores the heap property without touching that suffix.

When one active record remains, the whole array is sorted.

The heap shrinks while the sorted suffix grows



After construction, each step exchanges the root, reduces the heap size, and repairs the root. The physical length stays five.

Read the running time from the code

Let $n = \text{length}(A)$. Count comparisons, exchanges and array accesses as constant-time operations.

```

Heapsort(A)
  heap_size <- Build-max-heap(A)           # Theta(n), once
  for end <- length(A)-1 downto 1         # n-1 iterations
    exchange A[0] and A[end]              # Theta(1)
    heap_size <- heap_size-1              # Theta(1)
    Max-heapify(A, 0, heap_size)          # O(log i)

```

Here i is the active heap size *after* the decrement: $i = \text{heap_size} = \text{end}$.

The loop control costs $\Theta(n)$ in total. The repair cost is *per iteration*, not the cost of the whole loop.

Upper bound

After each extraction, the active heap has size $i = n - 1, n - 2, \dots, 1$. For $i \geq 2$, root repair costs at most $O(\log i)$.

Construction, loop control, exchanges and the final size-one repair together contribute $O(n)$.

$$\begin{aligned}
 T(A) &= O(n) + \sum_{i=2}^{n-1} O(\log_2 i) \\
 &= O(n) + O\left(\sum_{i=1}^{n-1} \log_2 i\right) \\
 &= O(n) + O((n-1) \log_2 n) = O(n \log n).
 \end{aligned}$$

Conclusion

Heapsort takes $O(n \log n)$ time on every input. This calculation gives an upper bound; a matching worst-case lower bound needs a separate argument.

A lower bound on the sum is not a lower bound on the code

The numerical sum $\sum_{i=1}^{n-1} \log_2 i$ is $\Theta(n \log n)$. Its largest half already gives

$$\begin{aligned} \sum_{i=1}^{n-1} \log_2 i &\geq \sum_{i=\lceil n/2 \rceil}^{n-1} \log_2 i \\ &\geq \lfloor n/2 \rfloor \log_2(n/2) = \Omega(n \log n). \end{aligned}$$

The upper bound follows from $\sum_{i=1}^{n-1} \log_2 i \leq (n-1) \log_2 n$.

Keep the logical direction

We bounded each repair *above* by a logarithm. A repair may stop early, so this numerical lower bound does not show that the algorithm performs that much work.

Lower bound

Consider n *distinct* keys. Their relative input order has $n!$ possibilities.

- A deterministic comparison-sorting algorithm must distinguish those orders.
- Each comparison has two outcomes, so its executions form a binary decision tree.
- The tree needs at least $n!$ reachable leaves; a tree of height h has at most 2^h leaves.

$$2^h \geq n! \quad \implies \quad h \geq \log_2(n!).$$

The height is the *longest* execution: some input needs at least this many comparisons.

Heapsort is a comparison sort, so this lower bound applies to its worst case. Chapter 6 develops the decision-tree model in more detail.

Worst-case complexity

Using just the largest half of the factors in $n!$, for $n \geq 2$,

$$\begin{aligned}\log_2(n!) &= \sum_{i=1}^n \log_2 i \\ &\geq \sum_{i=\lceil n/2 \rceil}^n \log_2 i \geq \frac{n}{2} \log_2(n/2) = \Omega(n \log n).\end{aligned}$$

Each comparison takes constant time in our cost model. Therefore

$$T_{\text{worst}}(n) = \Omega(n \log n), \quad T_{\text{worst}}(n) = O(n \log n).$$

$$T_{\text{worst}}(n) = \Theta(n \log n).$$

This is a worst-case statement. It does not say that every repair descends to a leaf, or that every input takes $\Theta(n \log n)$ time.

Equal keys show why the best case needs qualification

Suppose all input keys are equal.

- Neither strict $>$ test selects a child during repair.
- Each root repair stops after constant work.
- Construction and the $n - 1$ sorting iterations together take $\Theta(n)$ time.

For the displayed implementation

The best case over inputs allowing repeated keys is $\Theta(n)$. An unqualified best-case $\Theta(n \log n)$ claim is false.

An average-case claim also needs an input distribution and a key-domain assumption.

Space and stability depend on the implementation

Auxiliary space

Heapsort reuses the input array. With iterative repair it needs $\Theta(1)$ auxiliary space. The displayed recursive repair may instead use $O(\log n)$ stack space.

Stability

Heapsort is not stable: a long-distance exchange can reverse the relative order of equal-key records.

Distinguish the fixed input array from extra storage and from the recursion stack.

The three methods offer different guarantees

Merge sort Uses an auxiliary buffer, is stable, and guarantees $\Theta(n \log n)$ time.

Heapsort Reuses the input array, has worst-case $\Theta(n \log n)$ time, and is not stable.

Quicksort Partitions in place and often has excellent locality; its ordinary deterministic version has a quadratic worst case.

A heap is a reusable representation. Its partial order supports sorting and priority-driven algorithms.

Exercises on heap shape, order, and repair

Keep logical heap size separate from physical array length.

Exercise 1

For a ten-element zero-based heap, write the parent and child indices of every position. Identify the leaves and the height of every internal node. Write each node as $i : A[i]$ to keep indices and values distinct.

Exercise 2

Draw both the array and tree views of $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. List every parent-child comparison implied by the max-heap property and give two pairs whose relative order is unspecified.

Exercise 3

Trace Max-heapify on $[4, 14, 10, 8, 7, 9, 3, 2, 1]$. After every exchange, state which subtree can still contain a violation.

Exercises on bottom-up construction

Exercise 4

Build a max-heap from $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$ using the bottom-up algorithm. Show the array after every nontrivial repair.

Exercise 5

Explain why multiplying n calls by $O(\log n)$ gives a correct but non-tight bound for `Build-max-heap`. Group the nodes by height and derive the linear bound.

Exercise 8

Implement iterative downward repair and measure the number of levels traversed during bottom-up construction on random and decreasing arrays. Compare the measurements with the distribution-of-heights analysis.

Exercises on sorting and priority-queue workloads

Exercise 6

Trace Heapsort on $[5, 3, 17, 10, 84, 19, 6, 22, 9]$. At each iteration mark the active heap and the sorted suffix.

Exercise 7

Compare an unsorted array, a sorted array, and a heap for a workload containing i insertions, e maximum extractions, and q arbitrary searches. Describe two different workloads that lead to different preferred representations.

Reading pointers

- CLRS, *Introduction to Algorithms*, 3rd edition, Chapter 6.
- Kleinberg and Tardos, Section 2.5 (priority queues).
- Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*, heap and heapsort sections in Chapter 3.

Thank you!



UNIVERSITY
OF TRIESTE