Tutorial 6

Heap data structure and HeapSort

18YZALG – Basics of Algorithmization, Summer Semester 2026

Today

- Heap as a **data structure**: what it stores and what it makes cheap.
- Heap operations: **sift up** / **sift down**.
- HeapSort: sort in-place with a **worst-case** guarantee.
- One practical bonus: **Top-K** with a heap.

Today

- Heap as a **data structure**: what it stores and what it makes cheap.
- Heap operations: **sift up** / **sift down**.
- HeapSort: sort in-place with a **worst-case** guarantee.
- One practical bonus: **Top-K** with a heap.

Today

- Heap as a **data structure**: what it stores and what it makes cheap.
- Heap operations: **sift up** / **sift down**.
- HeapSort: sort in-place with a **worst-case** guarantee.
- One practical bonus: **Top-K** with a heap.

Today

- Heap as a **data structure**: what it stores and what it makes cheap.
- Heap operations: **sift up** / **sift down**.
- HeapSort: sort in-place with a **worst-case** guarantee.
- One practical bonus: **Top-K** with a heap.

Heap: working definition

Min-heap (priority queue)

A **complete binary tree** where every node is $\leq$ its children.

So the minimum element is always at the root.

Fast operations

- `peek-min`: $\mathcal{O}(1)$
- `push`: $\mathcal{O}(\log n)$
- `pop-min`: $\mathcal{O}(\log n)$

What it is *not*

- Not a fully sorted array.
- Only the **minimum** is guaranteed to be easy.

Heap: working definition

Min-heap (priority queue)

A **complete binary tree** where every node is $\leq$ its children.

So the minimum element is always at the root.

Fast operations

- `peek-min`: $\mathcal{O}(1)$
- `push`: $\mathcal{O}(\log n)$
- `pop-min`: $\mathcal{O}(\log n)$

What it is *not*

- Not a fully sorted array.
- Only the **minimum** is guaranteed to be easy.

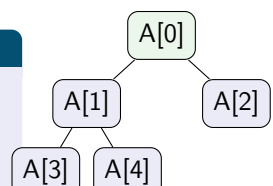
Heap in an array (index formulas)

Array representation

Store the complete binary tree level-by-level in an array A.

For node at index i

- parent: $\lfloor (i-1)/2 \rfloor$
- left child: $2i+1$
- right child: $2i+2$



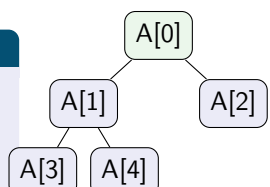
Heap in an array (index formulas)

Array representation

Store the complete binary tree level-by-level in an array A.

For node at index i

- parent: $\lfloor (i-1)/2 \rfloor$
- left child: $2i+1$
- right child: $2i+2$



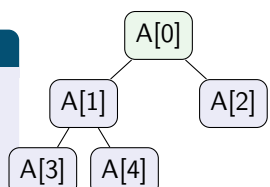
Heap in an array (index formulas)

Array representation

Store the complete binary tree level-by-level in an array A.

For node at index i

- parent: $\lfloor (i-1)/2 \rfloor$
- left child: $2i+1$
- right child: $2i+2$



Core operation: sift down (min-heap)

Idea

If a node is too large, swap it with the smaller child until the heap property is restored.

Pseudocode

```
SIFT-DOWN(A, i, n):          # heap is A[0:n)
  while True:
    l = 2*i + 1
    r = 2*i + 2
    if l >= n: return
    m = l
    if r < n and A[r] < A[l]:
      m = r
    if A[i] <= A[m]: return
    swap(A[i], A[m])
    i = m
```

Core operation: sift down (min-heap)

Idea

If a node is too large, swap it with the smaller child until the heap property is restored.

Pseudocode

```
SIFT-DOWN(A, i, n):          # heap is A[0:n)
  while True:
    l = 2*i + 1
    r = 2*i + 2
    if l >= n: return
    m = l
    if r < n and A[r] < A[l]:
      m = r
    if A[i] <= A[m]: return
    swap(A[i], A[m])
    i = m
```

Building a heap is $\mathcal{O}(n)$ (surprise!)

- **Heapify**: run `sift-down` from the last parent down to the root.
- Deeper nodes are cheap to fix (they move only a little).
- Total work sums to $\mathcal{O}(n)$, not $n \log n$.

Practical meaning

If you already have all data, building the heap is fast — then `pop-min` gives a sorted stream.

Building a heap is $\mathcal{O}(n)$ (surprise!)

- **Heapify**: run `sift-down` from the last parent down to the root.
- Deeper nodes are cheap to fix (they move only a little).
- Total work sums to $\mathcal{O}(n)$, not $n \log n$.

Practical meaning

If you already have all data, building the heap is fast — then `pop-min` gives a sorted stream.

Building a heap is $\mathcal{O}(n)$ (surprise!)

- **Heapify**: run `sift-down` from the last parent down to the root.
- Deeper nodes are cheap to fix (they move only a little).
- Total work sums to $\mathcal{O}(n)$, not $n \log n$.

Practical meaning

If you already have all data, building the heap is fast — then `pop-min` gives a sorted stream.

Building a heap is $\mathcal{O}(n)$ (surprise!)

- **Heapify**: run `sift-down` from the last parent down to the root.
- Deeper nodes are cheap to fix (they move only a little).
- Total work sums to $\mathcal{O}(n)$, not $n \log n$.

Practical meaning

If you already have all data, building the heap is fast — then `pop-min` gives a sorted stream.

Building a heap is $\mathcal{O}(n)$ (surprise!)

- **Heapify**: run `sift-down` from the last parent down to the root.
- Deeper nodes are cheap to fix (they move only a little).
- Total work sums to $\mathcal{O}(n)$, not $n \log n$.

Practical meaning

If you already have all data, building the heap is fast — then `pop-min` gives a sorted stream.

HeapSort (max-heap version)

Algorithm sketch

```
HEAPSORT(A):
  BUILD-MAX-HEAP(A)              # O(n)
  for end in n-1 .. 1:
    swap(A[0], A[end])           # max goes to final position
    SIFT-DOWN-MAX(A, 0, end)     # restore heap on A[0:end)
```

- Time: $\mathcal{O}(n \log n)$ in best/avg/worst.
- Extra memory: $\mathcal{O}(1)$ (in-place).
- Not stable (swaps jump far).

HeapSort (max-heap version)

Algorithm sketch

```
HEAPSORT(A):
  BUILD-MAX-HEAP(A)              # O(n)
  for end in n-1 .. 1:
    swap(A[0], A[end])           # max goes to final position
    SIFT-DOWN-MAX(A, 0, end)     # restore heap on A[0:end)
```

- Time: $\mathcal{O}(n \log n)$ in best/avg/worst.
- Extra memory: $\mathcal{O}(1)$ (in-place).
- Not stable (swaps jump far).

HeapSort (max-heap version)

Algorithm sketch

```
HEAPSORT(A):
  BUILD-MAX-HEAP(A)              # O(n)
  for end in n-1 .. 1:
    swap(A[0], A[end])           # max goes to final position
    SIFT-DOWN-MAX(A, 0, end)     # restore heap on A[0:end)
```

- Time: $\mathcal{O}(n \log n)$ in best/avg/worst.
- Extra memory: $\mathcal{O}(1)$ (in-place).
- Not stable (swaps jump far).

HeapSort (max-heap version)

Algorithm sketch

```
HEAPSORT(A):
  BUILD-MAX-HEAP(A)              # O(n)
  for end in n-1 .. 1:
    swap(A[0], A[end])           # max goes to final position
    SIFT-DOWN-MAX(A, 0, end)     # restore heap on A[0:end)
```

- Time: $\mathcal{O}(n \log n)$ in best/avg/worst.
- Extra memory: $\mathcal{O}(1)$ (in-place).
- Not stable (swaps jump far).

Measured examples (two stories)

Sorting random integers (ms)

n	Quick	Merge	HeapSort
2000	1.75	1.99	2.40
4000	4.50	4.33	5.50
8000	8.26	9.32	11.7
16000	17.1	19.9	25.4

Top-K (K=50) is where heaps shine

n	Sort all	Heap Top-K
20000	2.31	0.271
50000	6.57	0.582
100000	14.1	1.11

Heap Top-K is $\mathcal{O}(n \log k)$ vs full sort $\mathcal{O}(n \log n)$.

Wrap-up

- Heap = **priority queue**: fast access to current min/max.
- HeapSort = in-place $\mathcal{O}(n \log n)$ with **worst-case guarantee**.
- If you don't need full order, ask for **Top-K** instead of sorting everything.

Wrap-up

- Heap = **priority queue**: fast access to current min/max.
- HeapSort = in-place $\mathcal{O}(n \log n)$ with **worst-case guarantee**.
- If you don't need full order, ask for **Top-K** instead of sorting everything.

Wrap-up

- Heap = **priority queue**: fast access to current min/max.
- HeapSort = in-place $\mathcal{O}(n \log n)$ with **worst-case guarantee**.
- If you don't need full order, ask for **Top-K** instead of sorting everything.