

# Data Structures – Heaps & Huffman (Lecture Notes)

Authors: Refat Othman and Diaeddin Rimawi

## Lecture 1 — Heaps and Heap Sort

### 1. What is a Heap (Priority Queue)?

A **heap** is a tree-based data structure that supports efficient retrieval of the **highest priority** element. We usually implement it as a **complete binary tree** stored in an array. A **priority queue** is an abstract data type that exposes operations like **insert** and **extract-min** or **extract-max**, and a heap is a standard way to implement it.

We define two core properties for a **binary heap**:

1. **Shape property**: the tree is complete, we fill each level from left to right without gaps.
2. **Heap property**: for a **min-heap**, every node is less than or equal to its children. For a **max-heap**, every node is greater than or equal to its children.

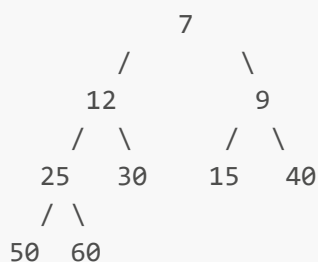
**Array representation** for a node at index  $i$  (0-based):

- left child at  $2i + 1$ , right child at  $2i + 2$ , parent at  $(i - 1) // 2$ .

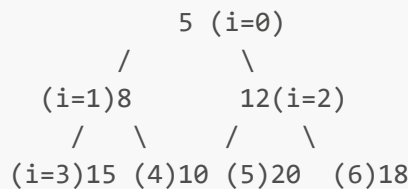
*Example (array  $\rightarrow$  tree, 0-based):* Array  $A = [7, 12, 9, 25, 30, 15, 40, 50, 60]$

- $i=0$  (7)  $\rightarrow$  children at 1,2  $\rightarrow$  12,9
- $i=1$  (12)  $\rightarrow$  children at 3,4  $\rightarrow$  25,30
- $i=2$  (9)  $\rightarrow$  children at 5,6  $\rightarrow$  15,40
- $i=3$  (25)  $\rightarrow$  child at 7  $\rightarrow$  50
- $i=4$  (30)  $\rightarrow$  child at 8  $\rightarrow$  60
- Parent examples: parent of index 5 (15) is  $(5-1)//2 = 2$  (value 9); parent of index 4 (30) is  $(4-1)//2 = 1$  (value 12).

ASCII visualization:



**Example:** Given the array  $[5, 8, 12, 15, 10, 20, 18]$ , which represents a heap:



Index mapping:

- Index 0 → left=1 (8), right=2 (12)
- Index 1 → left=3 (15), right=4 (10)
- Index 2 → left=5 (20), right=6 (18) This mapping works because the heap is a complete binary tree stored in an array.

## 2. Types of Heaps

We commonly use:

- **Binary heap**: min-heap or max-heap, complete binary tree, array-backed.
- **d-ary heap**: each node has **d** children (useful to reduce height when **decrease-key** is frequent).
- **Binomial heap**: a forest of binomial trees, supports fast **merge** (union).
- **Fibonacci heap**: supports very fast **amortized decrease-key** and **merge**, used in theoretical improvements of algorithms (e.g., Dijkstra). Implementation is more complex.
- **Pairing heap** and **Leftist heap**: pointer-based heaps that support efficient meld operations.

For this course, we implement and use the **binary heap**.

## 3. Heapify: definition and strategies

**Heapify** transforms a partially ordered array or subtree to satisfy the heap property.

- **Heapify-down** (sift-down): fix a violation at a node by swapping it with its best child and recursing downward. We use it after we remove the root or during bottom-up heap construction.
- **Heapify-up** (sift-up): fix a violation by swapping a node with its parent while the heap property is violated. We use it after insertion at the end.

## 4. Core operations

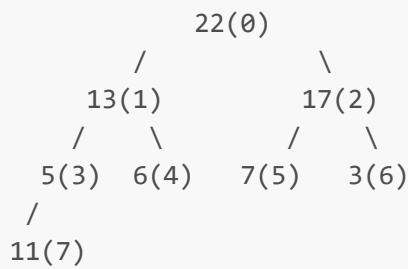
Below, each core operation is shown **step-by-step** with a small **min-heap** example and ASCII visualization. We use **0-based** indexing.

### 4.1 Build-heap (bottom-up **heapify-down**)

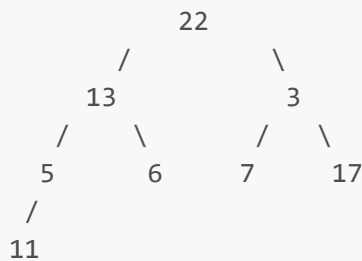
**Input array** (unsorted): **A** = [22, 13, 17, 11, 6, 7, 3, 5] (n = 8)

- Last internal index =  $\lfloor (n-2)/2 \rfloor = 3$ . We call **heapify-down(i)** for **i** = 3, 2, 1, 0.

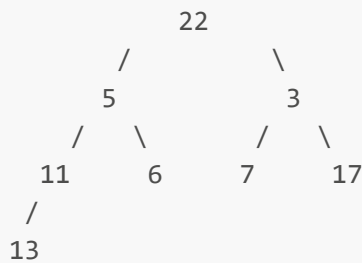
**i** = 3 (value 11), children at 7 and 8 → values 5 and — → swap with 5. **A** = [22, 13, 17, 5, 6, 7, 3, 11]



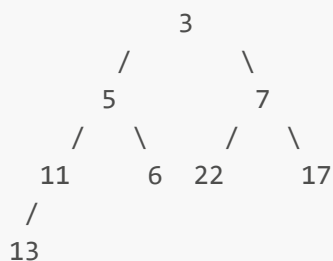
$i = 2$  (value 17), children at 5 and 6  $\rightarrow$  7 and 3  $\rightarrow$  swap with 3, then stop.  $A = [22, 13, 3, 5, 6, 7, 17, 11]$



$i = 1$  (value 13), children at 3 and 4  $\rightarrow$  5 and 6  $\rightarrow$  swap with 5.  $A = [22, 5, 3, 13, 6, 7, 17, 11] \rightarrow$  at index 3, children 7 and 8  $\rightarrow$  11 and  $- \rightarrow 11 < 13$ , swap.  $A = [22, 5, 3, 11, 6, 7, 17, 13]$



$i = 0$  (value 22), children at 1 and 2  $\rightarrow$  5 and 3  $\rightarrow$  swap with 3  $\rightarrow$  at index 2, children 5 and 6  $\rightarrow$  7 and 17  $\rightarrow$  swap with 7  $\rightarrow$  stop.  $A = [3, 5, 7, 11, 6, 22, 17, 13]$

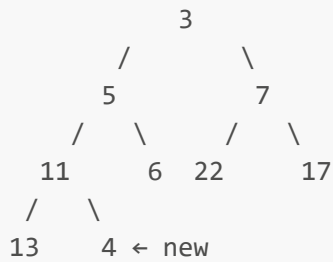


**Result:** a valid **min-heap**. **Cost:**  $O(n)$ .

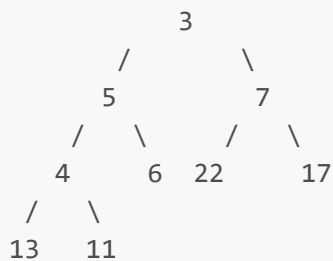
#### 4.2 Insert (push) with **heapify-up**

Start with heap  $A = [3, 5, 7, 11, 6, 22, 17, 13]$ . Insert 4.

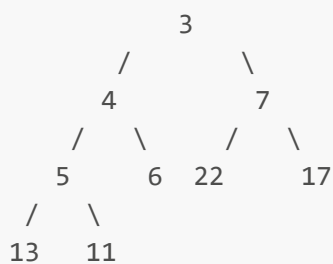
1. Append at end:  $A = [3, 5, 7, 11, 6, 22, 17, 13, 4]$



2. **heapify-up** from index 8: parent  $(8-1)//2 = 3$  value 11. Swap 4  $\leftrightarrow$  11.  $A = [3, 5, 7, 4, 6, 22, 17, 13, 11]$



3. New index 3, parent  $(3-1)//2 = 1$  value 5.  $4 < 5$ , swap.  $A = [3, 4, 7, 5, 6, 22, 17, 13, 11]$

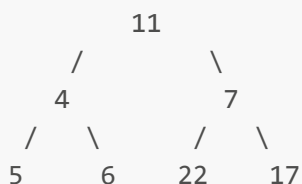


4. New index 1, parent 0 value 3.  $4 \geq 3$ , stop. **Cost:**  $O(\log n)$ .

### 4.3 Delete-min (pop root) with **heapify-down**

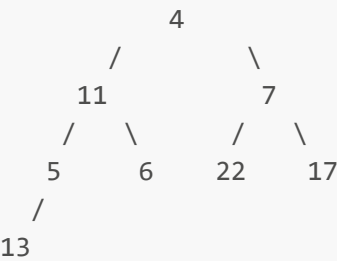
Start with heap  $A = [3, 4, 7, 5, 6, 22, 17, 13, 11]$ .

1. Swap root with last and remove last:  $A = [11, 4, 7, 5, 6, 22, 17, 13]$  (popped value was 3).

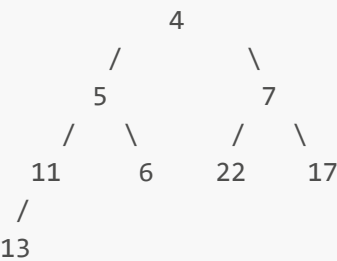


/  
13

2. **heapify-down** from index 0: compare 11 with children 4 and 7 → smallest 4 at 1 → swap.  $A = [4, 11, 7, 5, 6, 22, 17, 13]$



3. At index 1: children 5 and 6 → smallest 5 at 3 → swap.  $A = [4, 5, 7, 11, 6, 22, 17, 13]$



4. At index 3: child 13 only →  $11 \leq 13$ , stop. **Cost:**  $O(\log n)$ .

4.4 Peek

Return  $A[0]$  without modification. **Cost:**  $O(1)$ .

5. Time complexity

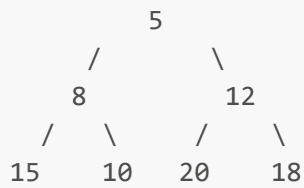
Operation	Binary Heap Complexity
build-heap (bottom-up)	$O(n)$
insert	$O(\log n)$
peek	$O(1)$
delete-min / delete-max	$O(\log n)$
heapify-up / heapify-down	$O(\log n)$

6. Visual examples

We visualize heaps as trees and arrays. We use a **min-heap** in the examples.

### 6.1 Example heap (three full levels)

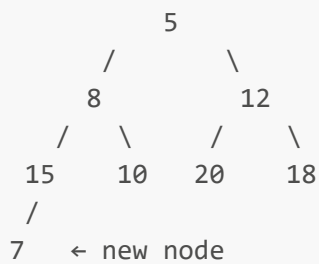
Array: [5, 8, 12, 15, 10, 20, 18]



Indices and children (0-based):  $0 \rightarrow (1,2)$ ,  $1 \rightarrow (3,4)$ ,  $2 \rightarrow (5,6)$ .

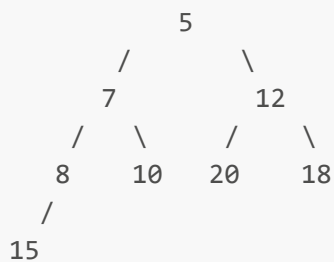
### 6.2 Insert 7 (heapify-up)

Start array: [5, 8, 12, 15, 10, 20, 18] Append 7  $\rightarrow$  [5, 8, 12, 15, 10, 20, 18, 7]



$7 < 15$  swap  $\rightarrow$  [5, 8, 12, 7, 10, 20, 18, 15]  $7 < 8$  swap  $\rightarrow$  [5, 7, 12, 8, 10, 20, 18, 15]  $7 > 5$  stop.

Final heap:

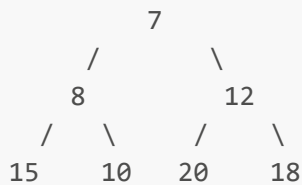


### 6.3 Delete-min (pop root)

Start array: [5, 7, 12, 8, 10, 20, 18, 15] Swap root with last, pop last: [15, 7, 12, 8, 10, 20, 18]

Heapify-down from index 0: 15 compares children 7 and 12, smallest is 7, swap  $\rightarrow$  [7, 15, 12, 8, 10, 20, 18] At index 1, children 8 and 10, smallest is 8, swap  $\rightarrow$  [7, 8, 12, 15, 10, 20, 18] At index 3, leaf, stop.

Final heap:



## 6.4 Build-heap from array

Input: [15, 10, 7, 18, 20, 12, 8] We call **heapify-down** from last internal index  $i = \lfloor (n-2)/2 \rfloor = 2$  down to 0:

- $i=2$ , node 7, children 12, 8, already fine.
- $i=1$ , node 10, children 18, 20, already fine.
- $i=0$ , node 15, children 10, 7, swap with 7  $\rightarrow$  [7, 10, 15, 18, 20, 12, 8], then **heapify-down** at index 2 with children 12, 8, swap with 8  $\rightarrow$  [7, 10, 8, 18, 20, 12, 15].

Result is a min-heap with three levels.

## 7. Full Python implementation (binary min-heap, simplified list-based version)

```

class MinHeap:
    def __init__(self, initial_data=None):
        self.data = []
        if initial_data:
            self.data = list(initial_data)
            self.build_heap()

    def left_child(self, index):
        return 2 * index + 1

    def right_child(self, index):
        return 2 * index + 2

    def parent(self, index):
        return (index - 1) // 2

    def heapify_down(self, index):
        size = len(self.data)
        while True:
            left = self.left_child(index)
            right = self.right_child(index)
            smallest = index

            if left < size and self.data[left] < self.data[smallest]:
                smallest = left
            if right < size and self.data[right] < self.data[smallest]:
                smallest = right
            if smallest == index:
                break

```

```

        self.data[index], self.data[smallest] = self.data[smallest],
self.data[index]
        index = smallest

    def heapify_up(self, index):
        while index > 0:
            parent_index = self.parent(index)
            if self.data[index] < self.data[parent_index]:
                self.data[index], self.data[parent_index] =
self.data[parent_index], self.data[index]
                index = parent_index
            else:
                break

    def build_heap(self):
        for i in range((len(self.data) - 2) // 2, -1, -1):
            self.heapify_down(i)

    def push(self, value):
        self.data.append(value)
        self.heapify_up(len(self.data) - 1)

    def peek(self):
        if not self.data:
            raise IndexError("peek from empty heap")
        return self.data[0]

    def pop(self):
        if not self.data:
            raise IndexError("pop from empty heap")
        self.data[0], self.data[-1] = self.data[-1], self.data[0]
        min_value = self.data.pop()
        if self.data:
            self.heapify_down(0)
        return min_value

    def __len__(self):
        return len(self.data)

    def __repr__(self):
        return f"MinHeap({self.data})"

```

## 8. Selection sort, then heap sort

### 8.1 Selection sort

**Idea:** repeatedly select the minimum from the unsorted suffix and swap it into position  $i$ .

**Process** for array  $A$  of length  $n$ :

1. for  $i$  from  $0$  to  $n-2$ 
  - 1.1 find index  $m$  of minimum in  $A[i..n-1]$
  - 1.2 swap  $A[i]$  and  $A[m]$



**Implementation (Python):**

```
def selection_sort(a):
    n = len(a)
    for i in range(n-1):
        m = i
        for j in range(i+1, n):
            if a[j] < a[m]:
                m = j
        a[i], a[m] = a[m], a[i]
    return a
```

**Complexity:** comparisons  $\approx n(n-1)/2$ , time  $O(n^2)$

**8.2 Heap sort**

**Idea:** build a heap once, then extract the minimum repeatedly. We improve the selection of the minimum from  $O(n)$  per step to  $O(\log n)$  per step.

**Process:**

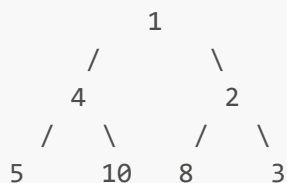
1. Build min-heap in  $O(n)$  from the array.
2. Repeatedly **pop** the root and append to output, each pop  $O(\log n)$ . Total  $O(n \log n)$  time,  $O(1)$  extra space if we do it in-place with a max-heap.

**Heap sort using MinHeap (with min-heap visualization):**

- Build a **min-heap** in the array (to extract minimums first).
- For **end** from  $n-1$  down to **1**: swap  $a[0]$  and  $a[\text{end}]$ , reduce heap size, **heapify-down**(0).
- Result becomes ascending order.

**ASCII visualization** for heap sort (min-heap, descending) on  $[4, 10, 3, 5, 1, 8, 2]$ :

Initial array  
 $[4, 10, 3, 5, 1, 8, 2]$   
 Build min-heap



Extract min → collect ascending:  $[1, 2, 3, 4, 5, 8, 10]$

**Implementation using the MinHeap class above (order with min-heap):**

```
def heapsort_inplace(array):
    # Create a MinHeap from the given unsorted array
```

```

heap = MinHeap(array)
sorted_list = []
# Extract all elements in ascending order
while len(heap) > 0:
    sorted_list.append(heap.pop())
return sorted_list

```

**Complexity:** build  $O(n)$ , then  $n-1$  heapify steps  $O(\log n)$  each, total  $O(n \log n)$ .

## Lecture 2 — Huffman Coding with Heaps

### 1. What is Huffman coding and why we use it

**Huffman coding** is a greedy compression algorithm that assigns **variable-length** prefix-free binary codes to characters based on their **frequencies**. More frequent characters get **shorter codes**, which reduces the total number of bits. We use it in compressors and file formats.

#### Key properties:

- Code is **prefix-free** (no code is a prefix of another), which enables unique decoding.
- The algorithm builds an **optimal** prefix code for a given set of symbol frequencies.

### 2. Main idea

1. Count character frequencies.
2. Put each character as a leaf node with its frequency into a **min-heap**.
3. While the heap has more than one node, we repeatedly extract the two smallest nodes, merge them into a new internal node whose frequency is the sum, and push it back.
4. The final node is the root of the Huffman tree.
5. Assign 0 to left edges, 1 to right edges, read codes by traversing from root to leaves.

### 3. Pseudocode

```

Huffman(F):
    heap ← empty min-heap
    for each (char c, freq f) in F:
        push(heap, Node(c, f))
    while size(heap) > 1:
        x ← pop(heap) // min frequency
        y ← pop(heap)
        z ← Node(char=None, freq=x.f + y.f, left=x, right=y)
        push(heap, z)
    root ← pop(heap)
    return BuildCodes(root)

BuildCodes(root):
    codes = {}
    def dfs(node, path):
        if node.char is not None:

```

```

        codes[node.char] = path
        return
    dfs(node.left, path + "0")
    dfs(node.right, path + "1")
dfs(root, "")
return codes

```

**Complexity:** building uses a heap with at most  $\sigma$  symbols, we perform  $\sigma - 1$  merges. Each heap operation costs  $O(\log \sigma)$ , total  $O(\sigma \log \sigma)$ . Encoding a text of length  $n$  then costs  $O(n)$  once we have the code table.

---

#### 4. Worked example with visualization: "abracadabra" (step-by-step)

We encode the text:

```
abracadabra
```

Length: 11 characters.

##### 4.1 Frequency table (why)

We first count how often each symbol appears. Huffman prioritizes low-frequency symbols with longer codes and high-frequency symbols with shorter codes. For **abracadabra**:

```

'a': 5
'b': 2
'r': 2
'c': 1
'd': 1

```

Rationale: assigning shorter codes to frequent **a** (5) will minimize the total bit-length.

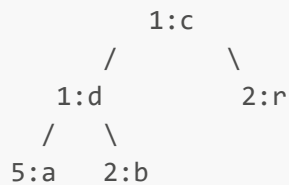
##### 4.2 Min-heap initialization (one insertion at a time)

We build a **min-heap** keyed by frequency. For equal frequencies we tie-break lexicographically to make the example deterministic.

Start with empty heap `[]` (array, level-order):

1. insert **(a,5)** → `[a:5]`
2. insert **(b,2)** → `[b:2, a:5]`
3. insert **(r,2)** → `[b:2, a:5, r:2]`
4. insert **(c,1)** → `[c:1, b:2, r:2, a:5]`
5. insert **(d,1)** → `[c:1, d:1, r:2, a:5, b:2]`

**ASCII min-heap after all inserts:**



(Shown as **freq:char**; the root is the smallest frequency.)

### 4.3 Iterative merges (why and how)

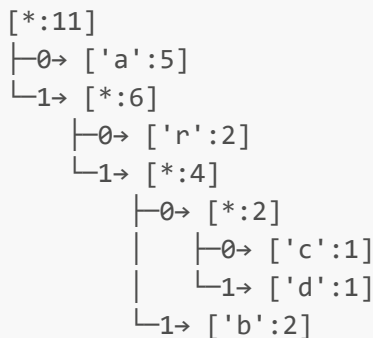
At each step we **pop the two smallest** items, **merge** them into a new internal node whose frequency is their sum, then **push** it back. This greedy rule ensures optimal total cost.

*Notation:* leaves shown as **char:freq**; internal nodes as **•:freq**.

- **Step 1:** Pop **c:1** and **d:1** → push **•:2**. Heap becomes [**•:2**, **b:2**, **r:2**, **a:5**].
- **Step 2:** Pop **•:2** and **b:2** (tie-break) → push **•:4**. Heap becomes [**r:2**, **•:4**, **a:5**].
- **Step 3:** Pop **r:2** and **•:4** → push **•:6**. Heap becomes [**a:5**, **•:6**].
- **Step 4:** Pop **a:5** and **•:6** → push **•:11** (the root). Heap becomes [**•:11**].

### 4.4 Final Huffman tree (then derive codes)

We place the **smaller** child on the **left** to keep things deterministic. Left edge is **0**, right edge is **1**.



#### Codes (path from root):

- **a: 0** (depth 1)
- **r: 10** (depth 2)
- **b: 111** (depth 3)
- **c: 1100** (depth 4)
- **d: 1101** (depth 4)

Why these lengths? The greedy merges push rare symbols (**c,d**) deeper, giving them longer codes, while frequent **a** sits closest to the root.

### 4.5 Encode the text (cost analysis)

Replace each character with its code and concatenate.

- Original size (ASCII):  $11 \times 8 = 88$  bits.
- Compressed size (Huffman):  $5 \times \text{len}('a') + 2 \times \text{len}('b') + 2 \times \text{len}('r') + 1 \times \text{len}('c') + 1 \times \text{len}('d') = 5 \times 1 + 2 \times 3 + 2 \times 2 + 1 \times 4 + 1 \times 4 = 5 + 6 + 4 + 4 + 4 = 23$  bits.

**Compression ratio**  $\approx 23 / 88 \approx 0.261$  (about **74% smaller**).

#### 4.6 Decode back (why it's unambiguous)

We read the bitstream left-to-right from the root. Because the code is **prefix-free**, no codeword is a prefix of another, so as soon as we reach a leaf we emit its character and reset to the root. This guarantees exact reconstruction of **abracadabra**.

## Exercises

Exercise 1 — Kth smallest element in an unsorted array

**Problem:** Given an array **A** and an integer **k** (1-indexed), find the kth smallest element. **Approaches:**

- **Min-heap:** build min-heap **O(n)**, pop **k-1** times, answer at next pop **O(k log n)**.

---

**Assignment for students:** Using what you learned in Lecture 1, implement a MaxHeap class from scratch (supporting insert, pop, heapify-up, heapify-down) and demonstrate it by inserting a set of integers, then repeatedly popping to display the elements in descending order.