ADA.
Heap.

Angel Napa

January 1, 2026

1 Heaps

A *(binary) heap* data structure is an array (indexed from 1) that can be seen as a nearly complete binary tree. As shown in the following image.

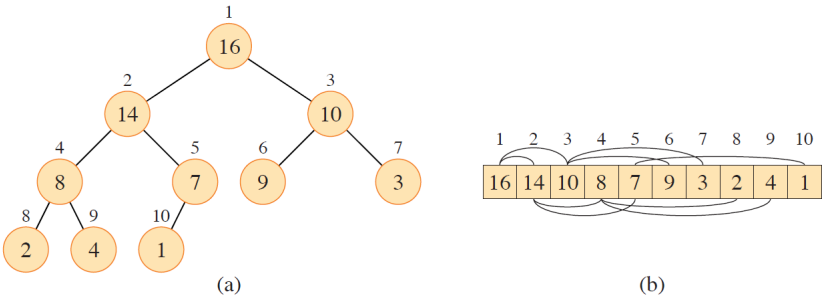

Figure 1: Cormen, Introduction to Algorithms

Each node of the tree correspond to an element of the array, and each level is full, except possibly the last one, which is filled from the left up to a point.

An array A that represents a heap has two attributes: $A.length$, $A.heap-size$, where $heap\text{-}size \leq length$ (we will go into detail later on)

The root of the tree is the element $A[1]$.

Given an index i of a node, note that

- The father of node i has index $\lfloor i/2 \rfloor$.
- Its left child has index $2i$.
- Its right child has index $2i + 1$.

Proof. We will prove, by induction on i, that the left and right children of index i are $2i$ and $2i + 1$ respectively.

If $i = 1$ then the left and right children are $2 = 2i$ and $3 = 2i+1$ respectively.

Now suppose that $i > 1$. By the induction hypothesis, the left and right children of $i-1$ are $2(i-1) = 2i-2$ and $2(i-1)+1 = 2i-1$.

Since the children of i are the nodes that immediately follow the children of $i-1$, we have that the left and right children of i are $2i$ and $2i+1$.

Now we will show that the parent of i is $\lfloor i/2 \rfloor$.

If i is even then $i = 2k$ for some integer k and thus i is the left child of $k = i/2 = \lfloor i/2 \rfloor$. If i is odd then $i = 2k+1$ for some integer k and thus i is the right child of $k = (i-1)/2 = \lfloor i/2 \rfloor$. □

Therefore, we can access the parent and children of an index i in constant time:

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Figure 2: Taken from the book Cormen, Introduction to Algorithms

The *height of a node* in a heap is the number of edges on the maximum path from that node to a leaf (this path only uses descendants). The *height of a heap* is the height of its root.

Exercise 1.1. *What is the minimum and maximum number of elements in a heap with height h?*

Exercise 1.2. *Prove that a heap with n nodes has height $\lfloor \lg n \rfloor$*

Property 1.1. *A heap with n nodes has height $\Theta(\lg n)$*

Proof. Directly from Exercise 1.2. □

2 Heap Property

There are two types of heaps: Max-heaps and Min-heaps. Depending on the type, the corresponding property must be satisfied.

- In a *Max-heap*, for each node i, $A[\text{PARENT}(i)] \geq A[i]$
- In a *Min-heap*, for each node i, $A[\text{PARENT}(i)] \leq A[i]$

For this chapter, we will mainly use Max-heaps.

Many times our heap is not satisfying the max-heap property. The following algorithm is responsible for modifying the heap so that it does.

The algorithm takes as input a heap A and an index i such that the heaps with roots LEFT(i) and RIGHT(i) **are max-heaps** (already satisfy the property). Upon completion of the algorithm, the heap with root i will be a Max-heap.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

Figure 3: Taken from the book Cormen, Introduction to Algorithms

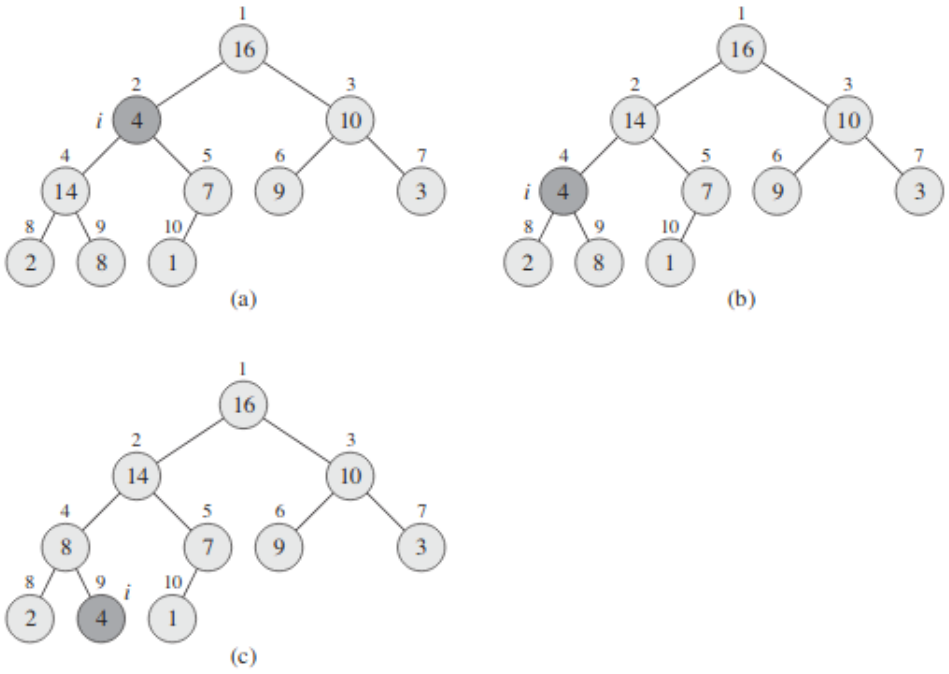

Figure 4: Taken from the book Cormen, Introduction to Algorithms

Execution time of MAX-HEAPIFY. Note that lines 1 to 9 take constant time. Then, in the worst case, we have that

$$T(n) \leq T(2n/3) + k.$$

How do we obtain the $2n/3$? Let n_i and n_d be the number of nodes in each left and right subtree respectively. It is clear that $n_i + n_d = n - 1$. Also observe that $n_i \leq 2n_d + 1$ (exercise), which implies that $n_d \geq \frac{n_i - 1}{2}$. Then $n - 1 = n_i + n_d \geq n_i + \frac{n_i - 1}{2} = \frac{3n_i - 1}{2}$. This implies that $n_i \leq \frac{2n-1}{3}$. Therefore, in the worst case, the left subtree will have a size of $\frac{2n-1}{3} \leq \frac{2n}{3}$.

By solving the recurrence using the master theorem, we get $T(n) = \Theta(\lg n)$ in the worst case (thus the algorithm is $O(\lg n)$).

Exercise 2.1. *Is an array sorted in increasing order a Min-heap? Is it a Max-heap?*

Exercise 2.2. *Consider the following array:* $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$. *Is it a Min-heap? Is it a Max-heap?*

Exercise 2.3. *Run the routine* MAX-HEAPIFY(A,3) *on the array* $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$

3 Building a Max-Heap

In this section, we will show how to build a Max-Heap from any array $A[1..n]$.

To do this, we will use MAX-HEAPIFY. Consider the following algorithm, called BUILD-MAX-HEAP. The algorithm receives an array $A[1..n]$ and rearranges its elements so that the resulting array is a Max-Heap.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Figure 5: Taken from the book Cormen, Introduction to Algorithms

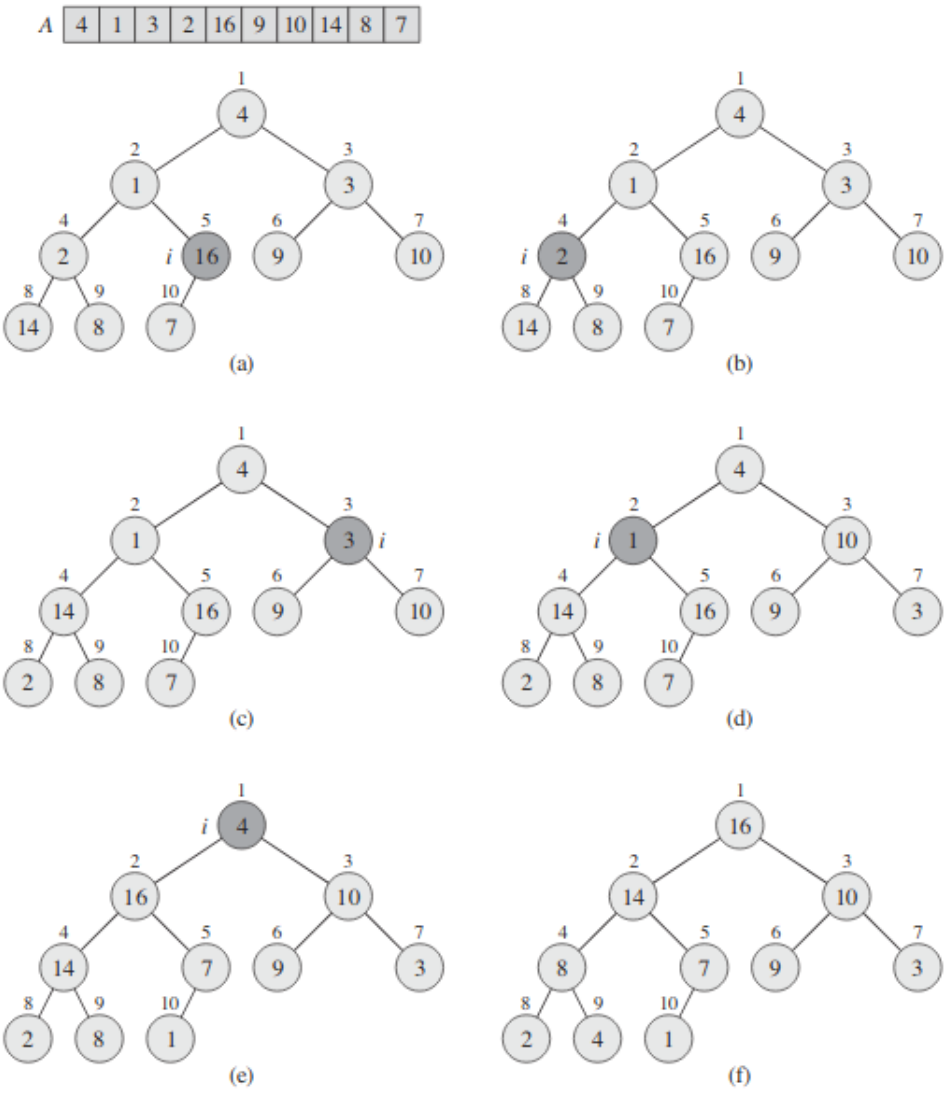

Figure 6: Taken from the book Cormen, Introduction to Algorithms

Consider the invariant: "At the beginning of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a Max-Heap".

- Initialization: At the beginning of the first iteration, we have $i = \lfloor n/2 \rfloor$. Since each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf (exercise), this leaf is a trivial Max-Heap and the property holds.
- Maintenance: By the invariant, given a node i, its children $2i$ and $2i+1$ are

Max-Heaps. Then, by using the subroutine MAX-HEAPIFY, the subtree with root i will also be a Max-Heap.

- Termination: Upon termination, we have $i = 0$ and therefore each node $1, 2, \ldots, n$ is the root of a Max-Heap. Thus A is already a Max-Heap.

We will analyze the time complexity of BUILD-MAX-HEAP.

A first analysis indicates that we make approximately $n/2$ calls to the subroutine MAX-HEAPIFY, which consumes time $O(\lg n)$. Therefore, we have a time complexity of $O(n \lg n)$.

However, the n of each recursive call is always less than the original n. It is convenient to express the execution time of each call in terms of the height of the node in question. Since, if the height of i is h, a call to MAX-HEAPIFY(A, i) will consume time $R(h) = O(h)$. Suppose this time is less than or equal to hk.

We have the following property.

Property 3.1. *In a heap with n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.*

Then, since we know that the height of a heap with n nodes is $\lfloor \lg n \rfloor$ (see a previous exercise), we obtain that the execution time $T(n)$ of BUILD-MAX-HEAP is given by:

$$\begin{aligned}
T(n) &= \sum_{h=1}^{\lfloor \lg n \rfloor} |\{i : \text{the height of } i \text{ is } h\}| \cdot R(h) \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil kh \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} (\frac{n}{2^{h+1}} + 1)kh \\
&= \frac{kn}{2} \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \\
&\leq \frac{kn}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \\
&= kn + k \sum_{h=0}^{\lfloor \lg n \rfloor} h \qquad (1) \\
&= O(n)
\end{aligned}$$

Where equation (1) holds because $\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$ (see introductory exercises).

Exercise 3.1. *Illustrate the operation* BUILD-MAX-HEAP *on the array* $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$.

Exercise 3.2. *What happens if instead of decrementing i, we use a for loop incrementing i from 1 to $\lfloor A.length/2 \rfloor$?*

Exercise 3.3. *Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes with height h in a heap with n nodes.*

4 The Heapsort Algorithm

We want to sort an array in *ascending* order.

The idea of the algorithm is to take advantage of the fact that in a Max-Heap, the element with the greatest value is located at the root, i.e., the element $A[1]$. Therefore, we can swap this element with the element at position $A[n]$ and reorganize the heap $A[1..n-1]$ to turn it into a Max-Heap. After this, the second smallest element will be at position $A[1]$.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 7: Taken from the book Cormen, Introduction to Algorithms

Since BUILD-MAX-HEAP takes $O(n)$ time and there are $n-1$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time, the execution time of the HEAPSORT algorithm is $O(n \lg n)$.

Exercise 4.1. *Illustrate the operation of* HEAPSORT *on the array* $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.

Exercise 4.2. *What is the execution time of* HEAPSORT *when the array is already sorted?*

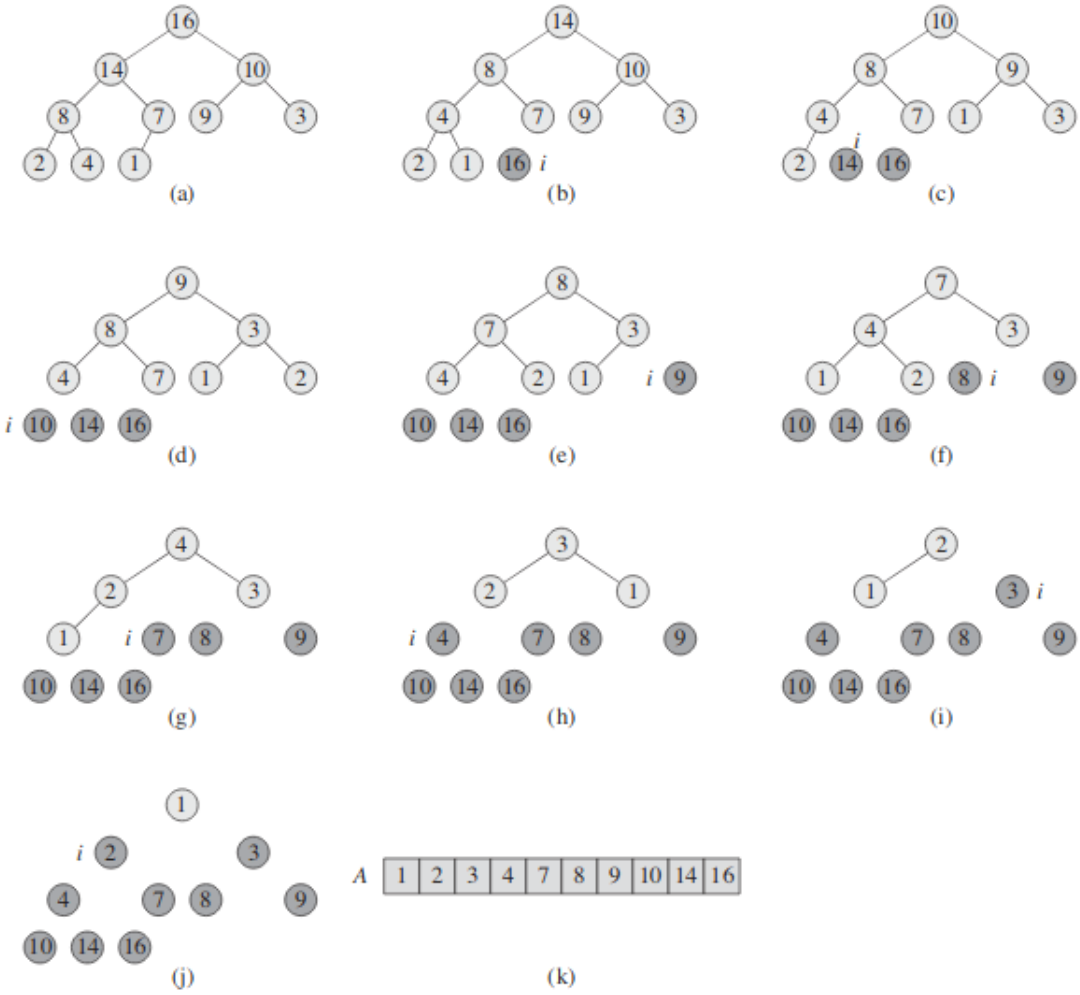

Figure 8: Taken from the book Cormen, Introduction to Algorithms

5 Priority Queue

The Heap data structure has more uses besides Heapsort; in this section, we will present one of the most popular ones: priority queues. We will focus on Max-Priority queues.

A *priority queue* is an array $A[1..n]$, whose values are called *keys*. We have the following operations:

- HEAP-MAXIMUM(A): returns the element with the highest value in A. We will do this in $\Theta(1)$.

- HEAP-EXTRACT-MAX(A): removes and returns the element in A with the highest value. We will do this in $O(\lg n)$.
- HEAP-INCREASE-KEY(A, i, key): increments the value of the key of element $A[i]$ to the new value key. We will do this in $O(\lg n)$.
- MAX-HEAP-INSERT(A, key): inserts the element with value key into A. We will do this in $O(\lg n)$.

Input: a max-heap A

Output: the maximum element in A

```
HEAP-MAXIMUM(A)
1  return A[1]
```

Figure 9: Taken from the book Cormen, Introduction to Algorithms

Input: a max-heap A

Output: the maximum element in A. Also removes the element from the heap.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Figure 10: Taken from the book Cormen, Introduction to Algorithms

Input: a max-heap A, an index i, and a value key

Increments the value of $A[i]$ to key, and modifies A so that it remains a max-heap.

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

Figure 11: Taken from the book Cormen, Introduction to Algorithms

Input: a max-heap A and a value key

Inserts the value key into A.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Figure 12: Taken from the book Cormen, Introduction to Algorithms

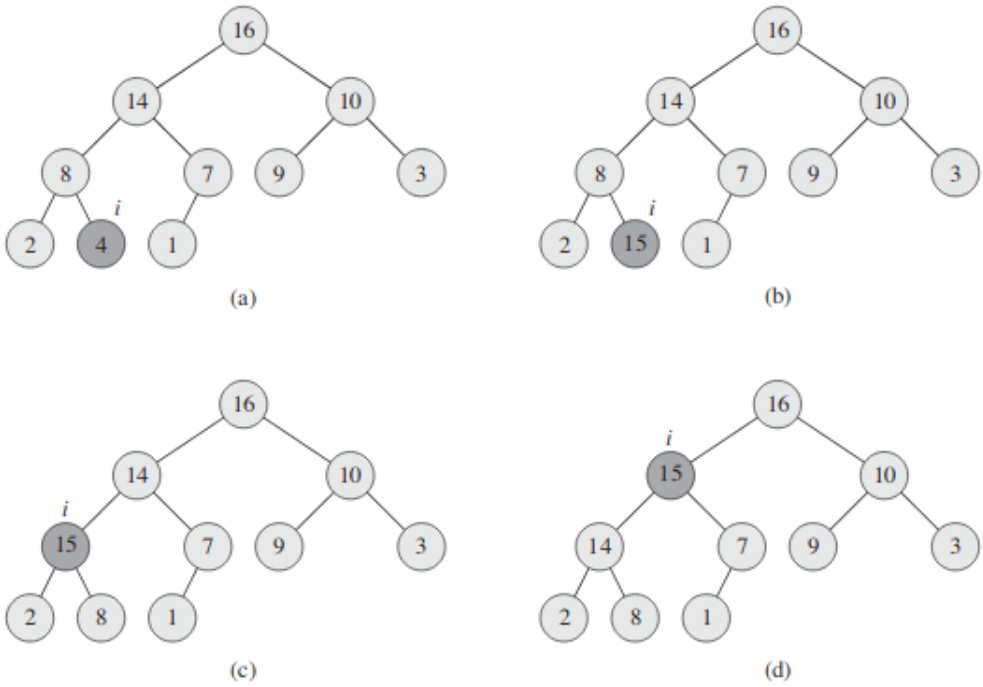

Figure 13: Simulation of HEAP-INCREASE-KEY. Taken from the book Cormen, Introduction to Algorithms

Exercise 5.1. *Simulate the operation* HEAP-EXTRACT-MAX *on* $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$.

Exercise 5.2. *Simulate the operation* MAX-HEAP-INSERT$(A, 10)$ *on* $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$.

Exercise 5.3. *Design an operation* HEAP-DELETE(A, i).