Heapsort

Manuel Serna-Aguilera

Heapsort is a neat sorting algorithm, its running time is $\Theta(n \cdot \log_2(n))$, like merge sort, but it also sorts in-place. It also uses a data structure called a heap, but heaps are also good for making efficient priority queues.

Introduction to Heaps

We can use an array to describe a (binary) heap, which is simply a nearly complete binary tree as seen in figure 1. By nearly complete it means the lowest level of the tree may or may not be completely full. The tree representation is just so we can better visualize the procedures we will perform on the array (a more accurate low-level view).

An array A represents a heap and has two attributes:

- $A.\mathit{length}$: the length/size of the array.
- $A.\mathit{heap\text{-}size}$: number of valid elements in A such that $0 \leq A.\mathit{heap\text{-}size} \leq A.\mathit{length}$.

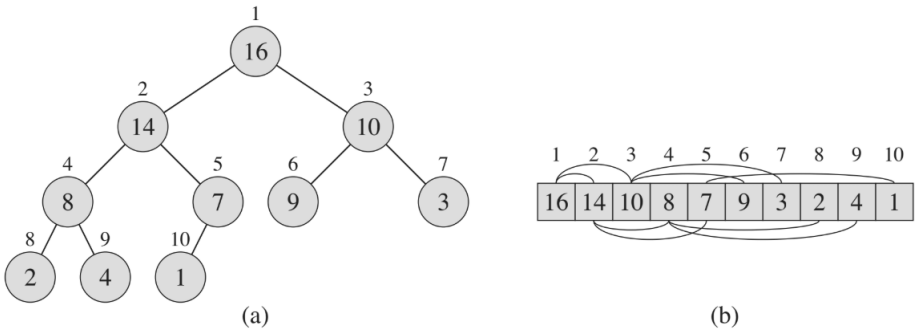

Figure 1: A max heap viewed as a **(a)** tree and **(b)** an array. Notice how labels and values in each node in the tree correspond to an index in the array. The height of the tree is $\Theta(\log_2(n))$.

The root of the tree is $A[1]$, and given an index i of a node, we can compute:

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

There are two kinds of binary heaps:

- **Max heaps**: Must satisfy the **max heap** property, where the parent node at index i is greater than or equal to its children.
  Formally, $A[\text{PARENT}(i) \geq A[i]]$. With the largest node being the root. Heapsort utilizes max heaps.

- **Min heaps**: The opposite of the max heap property; the **min heap** property is satisfied when the parent node at index i is less than or equal to its children. Formally, $A[\text{PARENT}(i) \leq A[i]]$.

Max Heapify

Max heapify is a procedure that maintains the max heap property. It assumes that the binary trees rooted at the left and right are max heaps, but the current node i might be smaller than its children, thus potentially violating the max heap property.

If the max heap property is violated, then we must look at the left and right children of node i, and swap i with whichever child has the bigger value. If the parent is already the largest, then the subtree rooted at i is already a max heap, and the procedure terminates. Moving down a branch of the binary tree takes $O(\log_2(n))$, or $O(h)$, where the height of the tree is $h = \log_2(n)$. Figure 2 illustrates an example of using the procedure on a single node, soon we will use it as a subroutine to build 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 // index of largest element is left child
 5  else
 6        largest = i // out of parent and left child, parent is larger
 7  // compare largest out of parent and left with the right (if it's in the valid heap)
 8  if r ≤ A.heap-size and A[r] > A[largest]
 9        largest = r // right child has the largest value
10  if largest ≠ i
11        // largest node was a child, keep going down to ensure max-heap property is not violated
12        exchange A[i] with A[largest]
13        MAX-HEAPIFY(A, largest)
```

Since we go down a single branch in each call to MAX-HEAPIFY, it costs $O(\log_2(n))$ time.

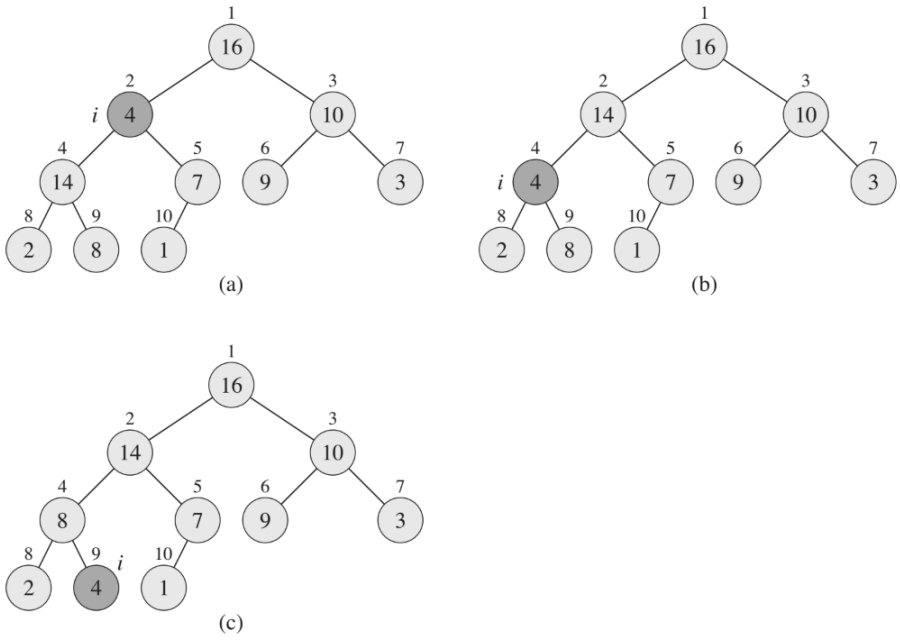

Figure 2: The process of calling MAX-HEAPIFY$(A, 2)$. **(a)** Looking at $i = 2$, which has a value of 4, is clearly smaller relative to both of its children. We will swap $i = 2$'s 4 with its left child's ($i = 4$) value of 14. **(b)** Now $i = 4$ may violate the max heap property, and indeed it does, so we perform the swapping action again; swap the 4 with the 8 (the value of the right child). **(c)** Hooray! The procedure finishes as there are no children to compare with (in this case).

Building a heap

The MAX-HEAPIFY procedure can be used in a bottom-up manner to convert an array $A[1\,..\,n]$, where $n = A.\mathit{length}$, into a max heap. The BUILD-MAX-HEAP procedure starts at the last parent, and works its way up to the root, making every subtree 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)
```

Each call to MAX-HEAPIFY costs $O(\log_2(n))$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \cdot \log_2(n)$, but it can be bounded to be $O(n)$.

Here is an example of the procedure BUILD-MAX-HEAP(A) being called.

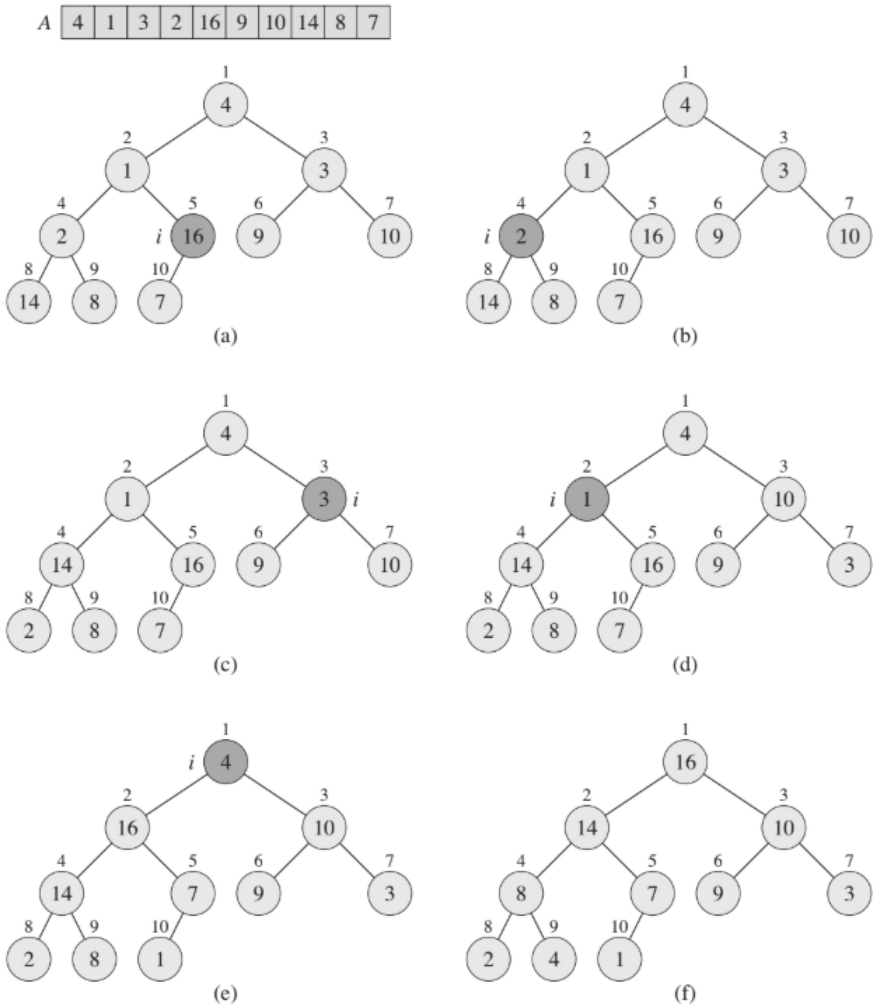

Figure 3: **(a)** Start at $i = 5$, it is larger than its left child, so no call to MAX-HEAPIFY, move up to next parent. **(b)** At $i = 4$, swap the values 2 and 14 to conform to the max-heap property, move up to the next parent. **(c)** At $i = 3$, swap 3 and 10, move up to the next parent. **(d)** At $i = 2$, swap 1 and 16. Now the value 1 is at $i = 5$, but its left child at $i = 10$ has a bigger value than it, so swap again. Now the subtree rooted at $i = 2$ is a valid max heap. **(b)** At the root, we swap, 4 with 16, then 4 with 14, and then 4 with 8. **(f)** The final max heap.

Heapsort

The heapsort algorithm starts by calling BUILD-MAX-HEAP to build a max heap on the input array. We remove the value from the root by swapping it with the last value in the valid max heap, and place it at the end of the array. We then maintain the max heap and repeat the process for the max heap of size $n - 1$ down to a heap of size 2. We then end up with an array with sorted values.

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
5      MEAX-HEAPIFY(A, 1)
```

Heapsort takes $O(n \cdot \log_2(n))$ time, since

- BUILD-MAX-HEAP takes $O(n)$ time.
- Each of the $n - 1$ MAX-HEAPIFY calls takes time $O(\log_2(n))$.

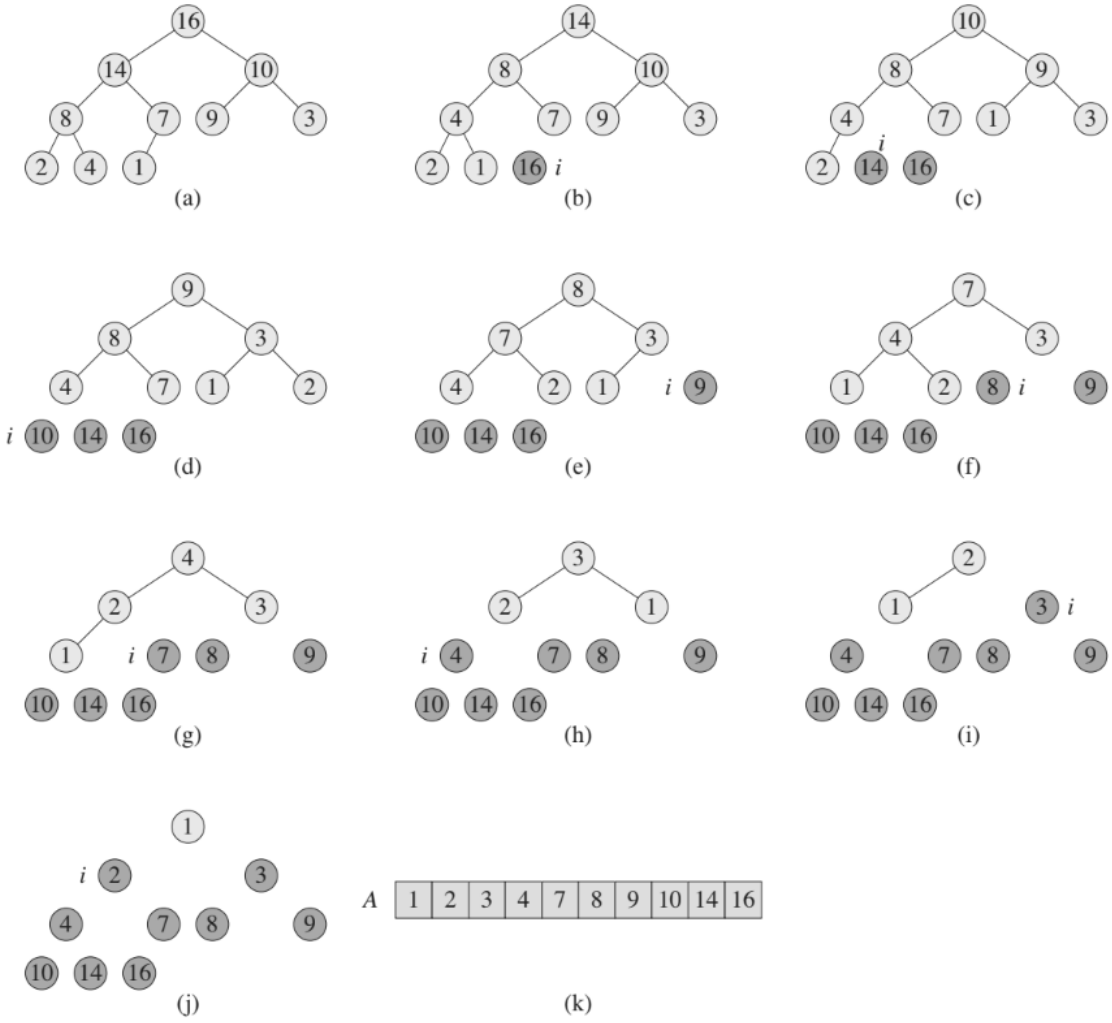

Figure 4: The operation of HEAPSORT. **(a)** The max heap resulting after executing line 1. **(b) - (j)** The max heap after iterating through each node, notice how, visually, the larger values are swapped to be towards the bottom of the tree. **(k)** The array representation of the finished result (basically equivalent to (j)).

Priority Queues

Even though heapsort is beat by quicksort pretty much every time, it is still very useful for making other data structures, like priority queues.

A **priority queue** is a dat structure for maintaining a set S of elements, each with an associated value called a **key**. A **max-priority queue** supports the following operations

> INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$. Has running time $O(\log_2(n))$.
>
> MAXIMUM(S) returns the element of S with the largest key. Has running time $\Theta(1)$ since the root is the biggest value.
>
> EXTRACT-MAX(S) **removes** and returns the element of S with the largest key. Has running time of $O(\log_2(n))$ since it performs only a constant amount of work on top of the $O(\log_2(n))$ time for calling MAX-HEAPIFY.
>
> INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k such that *old key* $\leq$ *new key*. Has running time $O(\log_2(n))$. Refer to figure 5 at the end of the document to see the process visualized.

Do note that MAX-HEAPIFY and HEAP-INCREASE-KEY can be thought of as maintaining the max heap property by going down and up a branch, respectively. This will be useful when writing the delete procedure.

Max-priority queues can be used for scheduling purposes. A **min-priority queue** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator, where events are simulated in order of their time of occurrence (time acts as the key).

Here is the pseudocode provided by the book.

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

HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  // Now adjust max-heap by propagating largest value to A[1]
5  A[1] = A[A.heap-size]
6  A.heap-size = A.heap-size − 1
7  MAX-HEAPIFY(A, 1)
8  return max

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  // Compare parent and current values, go up to satisfy max-heap property
5  while i > 1 and A[PARENT(i)] < A[i]
6       exchange A[i] with A[PARENT(i)]
7       i = PARENT(i)

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)

HEAP-DELETE(A, i)
1  swap A[i] and A[A.heap-size]
2  A.heap-size = A.heap-size − 1
3  if A[i] > A[PARENT(i)]
4       HEAP-INCREASE-KEY(A, i, A[i])
5  else
6       MAX-HEAPIFY(A, i) // check if we need to stop or keep going down
```

For the HEAP-DELETE procedure, since we are going up or down a single branch of the binary tree, the running time is $O(\log_2(n))$. Note that in the call to MAX-HEAPIFY, we check for which child branch to go down, and remember it also checks when to stop, so that is all we need to write.

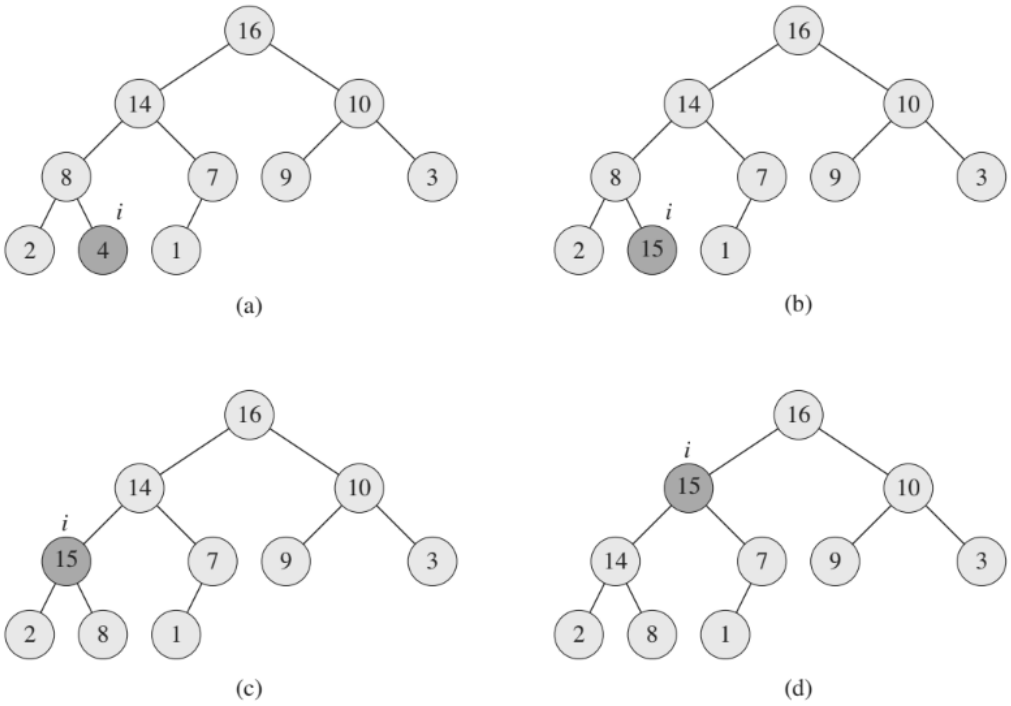

Figure 5: The process of calling HEAP-INCREASE-KEY. **(a)** The procedure is called on i whose key right now is 4. **(b)** The node's value is then increased to 15, the max heap property is clearly violated, we must now go up the binary tree to fix this. **(c)** Going up one level, 15 is not less than 14, the max heap property is still violated, thus we must keep going. **(d)** Final adjusted max heap, 15 is greater than each of its childrens' keys but less than its parent's key.