

# Chapter 6: Heapsort

## Introduction

- Heapsort is an  $\Theta(n \lg n)$  worst-case sorting algorithm like merge sort. Like insertion sort, it runs **in-place**, which means that it requires only  $\Theta(1)$  additional memory to run. Thus it combines the best features of insertion sort and merge sort.
- **Algorithm design technique: using a data structure to manage information.** Heapsort works by using a **heap** data structure. This data structure is also useful for managing priority queues as well for many other algorithms.

## Heaps

- Definition: The **(binary) heap** data structure is an array object with the following two properties:
  - The heap can be viewed as a **nearly complete binary tree**, that is a binary tree where all levels, except the last level, must be full and all nodes in the last level need to be as far left as possible.
  - The values in the heap satisfy a **heap property**. For a **max-heap**, the value at a node is  $\geq$  the value at its child node(s). For a **min-heap**, the value at a node is  $\leq$  the value(s) at the child node(s).
- **We can map a heap easily to the indices of an array.** An array  $A[1 : n]$  that represents a heap is an object with attribute  $A.heap-size$ , which represents how many elements in the heap are stored in the array. That is, only the elements in  $A[1 : A.heap-size]$ , where  $0 \leq A.heap-size \leq n$ , are valid elements of the heap. Then, the root of the heap is at  $A[1]$ .
- For a given node at index  $i$ , we can find its parent, left and right child with the following one-line procedures.

PARENT( $i$ )

1. return  $i/2$  // integer division

LEFT( $i$ )

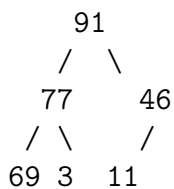
1. return  $2i$

RIGHT( $i$ )

1. return  $2i + 1$

*Advanced note* In most languages,  $PARENT(i) = i/2$  can be written as  $i >> 1$  (shift right by 1 bit). Similarly,  $LEFT(i) = 2i$  can be written as  $i << 1$  (shift left by 1 bit) and  $RIGHT(i) = 2i + 1$  can be written as  $i << 1 | 1$  (shift left by followed by bitwise or-ing 1 to add 1). This provides efficient implementations of these basic operations.

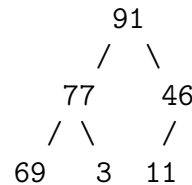
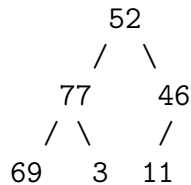
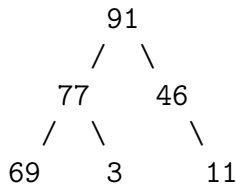
- Now we can restate the max-heap property as: For every node  $i$  other than the root,  $A[PARENT(i)] \geq A[i]$ . The largest element in a max-heap is at the root.
- Similarly, we can restate the min-heap property as: For every node  $i$  other than the root,  $A[PARENT(i)] \leq A[i]$ . The smallest element in a min-heap is at the root.
- Here is an example of a max-heap.



- The heap above represented as an array (root is  $A[1]$ )

91	77	46	69	3	11
----	----	----	----	---	----

- **In-class Exercise.** Which of the following are max-heaps and which are not?



- The height  $h$  of a heap with  $n$  nodes:  $h = \Theta(\lg n)$ .
- **Ex. 6.1-1** A heap with height  $h$  will have the minimum and maximum number of nodes  $n$  as follows.

$$\text{Minimum } n = 1 + 2 + 2^2 + \dots + 2^{h-1} + 1 = 2^h$$

$$\text{Maximum } n = 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$$

From the above two equations, we can derive  $h = \Theta(\lg n)$ .

- **Recommended Exercises:** Ex. 6.1-3, 6.1-4, 6.1-5, 6.1-6.

## Maintaining the Heap Property

MAX-HEAPIFY(A, i) // heapification downward

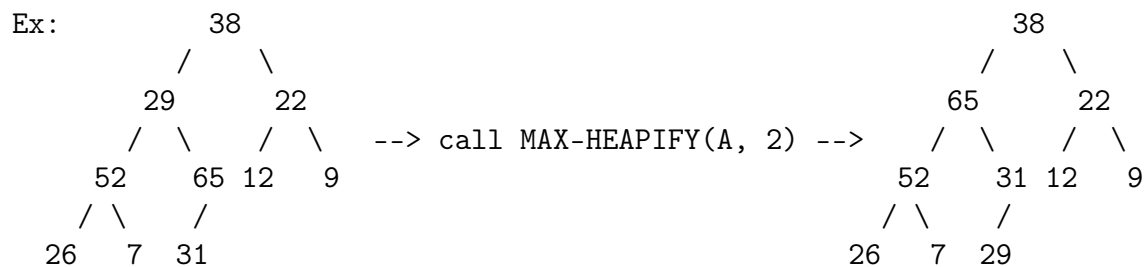
Pre-condition: Both the left and right subtrees of node i are max-heaps  
and i is less than or equal to heap-size[A]

Post-condition: The subtree rooted at node i is a max-heap

```

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)

```



- Running time analysis for MAX-HEAPIFY(A, i):

The element  $A[i]$  will be swapped down along a tree path. Thus, the running time for the procedure is  $O(h)$ , where  $h = \Theta(\lg n)$  is the tree height

- Recommended Exercises:** Ex. 6.2-1, 6.2-3, 6.2-4, 6.2-5.

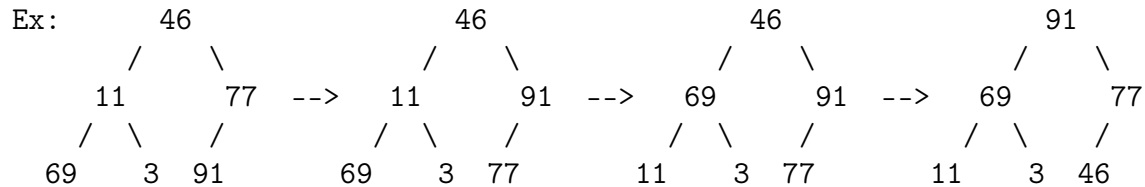
## Building a Heap

Using bottom-up approach to convert an array  $A[1\dots n]$  to a max-heap by calling a sequence of MAX-HEAPIFY procedures, starting at the last non-leaf node and ending at the root (works backwards on the array).

BUILD-MAX-HEAP(A)

//A[1:n] is an unsorted array

1. A.heap-size = n
2. for i = n/2 downto 1 //skip the leaves
3.     do MAX-HEAPIFY(A, i)



- Running time analysis for Build-Max-Heap (A):

Call MAX-HEAPIFY about  $n/2$  times  $\implies$  Build-Max-Heap (A) takes  $O(n \lg n)$ .

Actually,  $O(n \lg n)$  is an asymptotic upper bound but it is not tight. The tight upper bound is  $O(n)$ .

- **Recommended Exercise:** Ex. 6.3-1, 6.3-3.

## The Heapsort Algorithm

1. Make the input array A to a max-heap by calling Build-Max-Heap(A) procedure. We know A[1] has the largest element.
2. Exchange  $A[1] \leftrightarrow A[\text{heap-size}[A]]$  and then decrement heap-size[A] by one.
3. Call MAX-HEAPIFY(A, 1) to re-heapify A[1..heap-size[A]].
4. Repeatedly perform Step 2 and Step 3 until heap-size[A] = 1.

HEAPSORT(A)

// array A[1:n] is unsorted

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

- **Run-time analysis of HEAPSORT:** HEAPSORT calls BUILD-MAX-HEAP(A) once and calls MAX-HEAPIFY(A)  $n - 1$  times. Thus, the running time is  $O(n \lg n)$ .
- HEAPSORT uses a special data structure to solve a problem.

- HEAPSORT is very similar to selection sort - in each iteration, pick the largest element in the remaining set of elements and put it to the correction position (the “last” position). The difference is that they use different ways to pick the largest element.
- **Loop invariant:** At the start of each iteration of the **for** loop of lines 2–5, the subarray  $A[1 : i]$  is a max-heap containing the  $i$  smallest elements of  $A[1 : n]$ , and the subarray  $A[i + 1 : n]$  contains the  $n - i$  largest elements of  $A[1 : n]$ , sorted.

**Recommended Exercises:** 6.4-1, 6.4-2.

## Priority Queues

- A **priority queue** is a data structure for maintaining a set  $S$  of elements, each with an associated value called a **key**. It allows us to access elements in order of the priority represented by the key value.
- A **max-priority queue** supports the insert, maximum, extract-max and increase-key operations efficiently.
- A **min-priority queue** supports the insert, minimum, extract-min and decrease-key operations efficiently.
- The root of a max-heap contains the largest value. If we build a priority queue using a max-heap based on the priority values of elements, then the next element to be extracted from the queue is always located at the root. Similarly we can use a min-heap to build a min-priority queue.
- *Applications:* job scheduler, discrete event simulation.

HEAP-MAXIMUM(A)

//O(1) time

1. return  $A[1]$

HEAP-EXTRACT-MAX(A)

//O(lg  $n$ ) time

1. if  $A.\text{heap-size} < 1$

2.     then error -- heap underflow

3.  $\text{max} = A[1]$

4.  $A[1] = A[A.\text{heap-size}]$

5.  $A.\text{heap-size}--$

6. MAX-HEAPIFY(A, 1)

7. return  $\text{max}$

```

HEAP-INCREASE-KEY(A, i, key)
//O(lg n) time
1. if key < A[i]
2.   then error -- new key must be larger than current key
3. A[i] = key
4. while i > 1 and A[PARENT(i)] < A[i]
5.   exchange A[i] and A[PARENT(i)]
6.   i = PARENT(i)

```

```

MAX-HEAP-INSERT(A, key)
//O(log n) time
1. heap-size[A]++
2. A[A.heap-size] = negative-infinity
3. HEAP-INCREASE-KEY(A, A.heap-size, key)

```

- **Recommended Exercises:** Ex 6.5-1, 6.5-2, 6.5-3, 6.5-4, 6.5-7, 6.5-8.
- **Think-pair-share:** Ex 6.5-7: Show how to implement a FIFO queue with a priority queue. Show how to implement a stack using a priority queue.
- **Implementation issues**
  - A priority queue can be implemented by extending a heap implementation based on an array or based on a binary tree structure using pointers.
  - In an actual application, we will have objects with key values and satellite data. That requires some careful refactoring of the pseudo-code above.
  - Review **PriorityQueue** class from Java docs.