Heapsort

Binary Heaps
Partition
Worst-Case, Best-Case, Average-Case
Randomized Quicksort
Formal Analysis
Heap Property

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node $i$ other than the root,

$$A[\text{PARENT}(i)] \geq A[i].$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A min-heap is organized in the opposite way; the min-heap property is that for every node $i$ other than the root,

$$A[\text{PARENT}(i)] \leq A[i].$$

The smallest element in a min-heap is at the root.

Max-Heapify

MAX-HEAPIFY($A, i$)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
   4. largest = $l$
5. else largest = $i$
6. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
   7. largest = $r$
8. if largest $\neq i$
   9. exchange $A[i]$ with $A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)

Build-Max-Heap

BUILD-MAX-HEAP($A$)

1. $A.\text{heap-size} = A.length$
2. for $i = \lfloor A.length/2 \rfloor$ down to 1
3. MAX-HEAPIFY($A, i$)
Build-Max-Heap Illustration, Part 1

Heapsort Algorithm

\textsc{Heapsort}(A)

1. \textsc{Build-Max-Heap}(A)
2. for \( i = A.length \) downto 2
4. \( A.heap-size = A.heap-size - 1 \)
5. \textsc{Max-Heapify}(A, 1)

Build-Max-Heap Illustration, Part 2

Heapsort Illustration, Part 1
Heapsort Illustration, Part 2

Heap-Extract-Max

**HEAP-EXTRACT-MAX**($A$)
1. if $A$.heap-size < 1
2. error “heap underflow”
3. $\text{max} = A[1]$
5. $A$.heap-size = $A$.heap-size − 1
6. MAX-HEAPIFY($A$, 1)
7. return $\text{max}$

Heap-Increase-Key

**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 $A[\text{PARENT}(i)] < A[i]$
5. exchange $A[i]$ with $A[\text{PARENT}(i)]$
6. $i = \text{PARENT}(i)$

Priority Queue Operations

A **priority queue** is a data 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 $S = S \cup \{x\}$.
- **MAXIMUM**($S$) returns the element of $S$ with the largest key.
- **EXTRACT-MAX**($S$) removes and returns the element of $S$ with the largest key.
- **INCREASE-KEY**($S$, $x$, $k$) increases the value of element $x$’s key to the new value $k$, which is assumed to be at least as large as $x$’s current key value.
Max-Heap-Insert

**Max-Heap-Insert** \((A, key)\)

1. \(A.heap-size = A.heap-size + 1\)
2. \(A[A.heap-size] = -\infty\)
3. **Heap-Increase-Key** \((A, A.heap-size, key)\)

**Heap-Increase-Key Illustration**

(a) 
(b) 
(c) 
(d)