Heaps

xo Reading: CLR §7

Heap Contract

A heap is a mutable priority queue data structure supporting the following operations:

**EmptyHeap** (*Returns an empty heap.*)

**BuildHeap** (*A*)
Constructs and returns a heap containing the *n* elements of array *A* in *O*(*) time.

**Heap-Insert** (*H*, *key*)
Modifies *H* by inserting *key* into an *n*-element heap *H* in *O*(*lg*(*n*)) time. (Really want to insert value with key *key*, but this simplifies description of algorithm.)

**Heap-Extract-Max** (*H*)
Deletes from *H* and returns the largest key of *n*-element heap *H* in *O*(*lg*(*n*)) time

Heap Sort

Given the above heap operations, it’s easy to construct a guaranteed *O*(*n*(*lg*(*n*)) ) sorting algorithm:

**HeapSort** (*A*)

\[
\begin{align*}
H & \leftarrow \text{BuildHeap}(A) \\
\text{for } & i \leftarrow \text{length}[A] \text{ downto } 1 \text{ do} \\
A[i] & \leftarrow \text{Heap-Extract-Max}(H)
\end{align*}
\]

We will see below that the heap used by **HeapSort** can be stored within the argument array *A*, so that **HeapSort** can be an in-place sorting algorithm.
Definitions

The **binary address** of a node in a binary tree specifies the order in which it would be visited in a breadth first traversal. For example:

Operations on binary addresses:

- Left(address) = 2 * address
- Right(address) = (2 * address) + 1
- Parent(address) = address div 2

An \( n \)-element binary tree is **complete** if the set of binary addresses of its nodes is \{1, 2, \ldots, n\}. For example:

An \( n \)-element binary tree is **full** if it a complete tree of height \( h \) with \( 2^h - 1 \) nodes.

A **heap** is a complete binary tree satisfying the heap condition:

At every node in a heap, the node value is greater than or equal to all the values in its subtrees.

A heap of with **heap_size** elements can be represented as an array segment \( A[1..\text{heap_size}] \).
Insertion and Extraction

HeapInsert(A, key)
heap_size[A] ← heap_size[A] + 1
A[heap_size[A]] ← key
Bubble-Up(A, heap_size[A])

Bubble-Up(A, address)
while address > 1 and lt(A[Parent(address)], A[address]) do
    swap(A, address, Parent(address))
▷ Can get by with fewer assignments; See CLR
    address ← Parent(address)

Analysis:

Heap-Extract-Max(A)
if heap_size[A] < 1 then
    error "heap underflow"
    max ← A[1]
heap_size[A] ← heap_size[A] - 1
BubbleDown(A, 1)
return max

Bubble-Down(A, address)
▷ This function is called Heapify in CLR
if Left(address) ≤ heap_size[A]
    and lt(A[address], A[Left(address)]) then
    largest ← Left(address)
else
    largest ← address
if Right(address) ≤ heap_size[A]
    and lt(A[largest], A[Right(address)]) then
    largest ← Right(address)
if largest ≠ address then
    swap(A, address, largest)
    Bubble-Down(A, largest)

Analysis:
Constructing a Heap

Naive version of Build-Heap:

\[
\text{Build-Heap}(A)
\]
\[
\text{for } i \leftarrow 1 \text{ to } \text{length}[A] \text{ do}
\]
\[
\text{Uses array slots for heap storage!}
\]
\[
\text{Heap-Insert}(A, A[i])
\]

Analysis:

Clever version of Build-Heap:

\[
\text{Build-Heap}(A)
\]
\[
\text{heap\_size}[A] \leftarrow \text{length}[A]
\]
\[
\text{for } i \leftarrow (\text{length}[A] \text{ div } 2) \text{ downto } 1 \text{ do}
\]
\[
\text{Bubble-Down}(A, i)
\]

Analysis:

Note that never more than \((n/2)\) nodes of height \(h\) in a tree with \(n\) elements.