Heaps

Adnan Aziz

CLRS Chapter 6
Important—these heaps have nothing to do with the heap memory used for dynamically
allocated objects!
Recall from Appendix B.6 that a *binary tree* is defined on a finite set of nodes, and either
- contains no nodes, or
- has a designated root node, with the remaining nodes partitioned into a left subtree,
  and a right subtree.

In a *full binary tree*—each node is either leaf, or of degree 2. In a *complete binary tree*—all
leaves have same depth, all internal nodes have degree 2.

**Lemma 1** A complete binary tree of height $h$ has $2^{h+1} - 1$ nodes, of which $2^h - 1$ are internal.
Also any nonempty binary tree with $n$ nodes has height at least $\lceil \lg n \rceil$.

In CLRS—view heap as being a complete binary tree. At each node, store a “value”. Tree
is “completely filled”, except perhaps for lowest level where values are inserted from left up
to some node.
Another way of looking at heaps—a natural way to number nodes in a complete binary tree
on height $h$ is to assign number $1, 2, \ldots, 2^{h+1} - 1$ left-to-right, top-to-bottom. A heap is a
complete binary tree minus nodes in the range $2^h + k$ to $2^{h+1} - 1$ for some $k$ such that
$0 \leq k \leq 2^h - 1$.
A heap is an ideal data structure when you maintaining a set of ordered values, and are
doing *insert*, *delete*, and *extract-max* operations.
- However, it’s not much good for testing membership, or anything else for that matter
Very neat way of implementing a heap: use an array $A$, along with two attributes

1. $\text{length}[A]$—the size of the array

2. $\text{heap} - \text{size}[A]$—the number of elements of $A$ that are part of the heap

Root of the heap is given by $A[1]$. 
Crucial observation—do not need to explicitly have parent, left and right child fields

- given index for an element $i$, parent, leftchild and rightchild of $i$ are at index $\lfloor i/2 \rfloor$, $2 \cdot i$ and $2 \cdot i + 1$.

Note that these operations can be computed very efficiently by left/right shifts.
Heaps are required to satisfy the heap property— for every node, other than root, we have $A[\text{parent}(i)] \geq A[i]$.

- this implies that the largest element stored in a heap is stored at the root, also

- value at any node $x$ is greater than or equal to the value stored at any node in the subtrees rooted in $x$

Several heaps on $\{1, 2, 3, 4, 7, 8, 9, 10, 14, 16\}$.
Let’s first see intuitively how to insert, extract-max:

- example illustrating insert: add at end, “float up” the inserted value

- example illustrating extract-max: swap end value with root, reduce count, “float down” the root

In the extract-max example, we implicitly were doing the following:

- start with an array $A$, and an index $i$; assume the subtrees rooted at left($i$) and right($i$) are already heaps, but $A[i]$ may be less than its children

How can we efficiently convert the tree rooted at $i$ into a heap?
Formally, performed by the heapify routine—“float down” the value $A[i]$
Conceptually:

- determine the largest of the three elements $A[i], A[\text{left}(i)], A[\text{right}(i)],$

- then if $A[i]$ is largest, we’re done,
• otherwise, swap element at \( i \) with larger of the two children
  – this may break heap property at child \( \Rightarrow \) need to recur

In pseudo-code:

```plaintext
// Assumption: Binary tree rooted at left[i] and right[i] are
// heaps, but A[i] may be smaller than children
heapify(A,i)
    l <- left(i)
    r <- right(i)
        then largest <- l
    else largest <- i
        then largest <- r
    if ( largest != i )
        then exchange A[i] <-> A[largest]
        heapify(A,largest)
```

Observe—run time is some constant times the height of the tree, i.e., \( O(\log n) \).

How can we use `heapify(A,i)` to build a heap?

```plaintext
build-heap(A)
    heap-size(A) <- length[A]
    for i <- floor( length[A]/2) downto 1
        do heapify(A,i)
```

Try on the following heap \( A = \{ 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \} \).

Easy bound on runtime—\( O(n \cdot \log n) \).
• More sophisticated analysis—Θ(n), since many calls are made to very short trees
  – specifically, there are at most \( \lceil n/2^{h+1} \rceil \) notes of height \( h \)

Can use this to sort—first build heap, then move the top element to the end, reduce heap size by 1, heapify

**Priority queues**

In practice, heap-sort is not competitive with quicksort. However, heaps can be used for more than sorting. A priority queue is a data structure for maintaining a set \( S \) of elements each with an associated key.

• Supports the operations \texttt{insert}, \texttt{heap-max}, and \texttt{extract-max}.

Implementations:

• \texttt{heap-max}: trivial

• \texttt{heap-extract-max}: swap max with last element, reduce size by 1, heapify the top element

\[
\text{heap-extract-max}(A) \\
\text{if heap-size}(A) < 1 \\
\quad \text{then error "heap underflow!"} \\
\text{max} \leftarrow A[1] \\
\text{heapsize}[A] \leftarrow \text{heapsize}[A] - 1 \\
\text{heapify}(A,1) \\
\text{return max}
\]

• \texttt{heap-insert}: conceptually: insert at the end, then move up appropriately
heap-insert(A, key)

heapsize[A] <- heapsize[A] + 1

i <- heapsize[A]

// it’s not important that A[heapsize] be initialized

while (i > 1 && A[parent(i)] < key)

do A[i] <- A[parent(i)]

i <- parent(i)

A[i] <- key

Lower bounds on sorting

Any sorting procedure which uses only comparison must have time complexity $\Omega(n \lg n)$.

- The idea is that there are $n!$ different arrays on \(\{1, 2, \ldots, n\}\) of length $n$, and each compare splits the set of possible arrays into two subsets, one of which is at least half as big as the original set. Thus some array must require at least $\lg n!$ compares; by Stirling’s approximation, this is $\Omega(n \lg n)$.

You should read about techniques that can sort in $\Theta(n)$ time—“radix sort” in CLRS 8.3; the catch is that they only work when the values of the array entries lie in a small range. If the values are arbitrary but there’s only a small number of possible values, then too you can sort more efficiently than $\Omega(n \lg n)$. 