

Heaps

CLRS Part II, Chapter 6

A binary tree is defined on a finite set of nodes, and either

- contains no nodes, or
- is a root, a left subtree, and a right subtree.

Full binary tree—each node is either leaf, or of degree 2. Complete binary tree—all leaves have same depth, all internal nodes have degree 2.

Fact: A complete binary tree of height h has $2^{h+1} - 1$ nodes, of which $2^h - 1$ are internal. Nonempty binary tree with n nodes has height at least $\lceil \lg n \rceil$.

Heap—ideal data structure when you are doing inserts, delete, and extract-max. (But bear in mind, it's not much good for lookups, or anything else for that matter.)

In CLR—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 a point.

Another (in my opinion more intuitive way) of looking at heaps—a natural way to number nodes in a complete binary tree on height h is to assign number $1, 2, \dots, 2^{h+1} - 1$ left-to-right, top-to-bottom. Then 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 $0 \leq k \leq 2^h - 1$.

Very neat way of implementing a heap: use an array A , along with two attributes

1. $length[A]$ —the size of the array
2. $heap - size[A]$ —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 represent pointers to left, right child

- 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

Let's first see intuitively how to insert, extract-max:

- example illustrating insert
- example illustrating extract-max

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 $\text{left}(i)$ and $\text{right}(i)$ are already heaps, but $A[i]$ may be less than its children

How to efficiently convert tree rooted at i into a heap?

Formally, performed by the *heapify* routine — “float” the value $A[i]$ down the heap

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 pseudocode:

```

heapify(A, i)
  l ← left(i)
  r ← right(i)
  if ( l ≤ heap-size[A] and A[l] > A[i] )

```

```

    then largest <- l
    else largest <- i
  if ( r <= heap-size[A] and A[r] > A[largest]
    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 to use `heapify(A,i)` to build a heap?

```

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— $\Theta(n)$ (since many calls are made to very short trees)

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

Priority queues

Data structure for maintaining a set S of elements each with an associated key.

- Supports the operations `insert`, `heap-max`, and `extract-max`.

Implementations:

- `heap-max`: trivial

- heap-extract-max: swap max with last elt, reduce size by 1, heapify the top elt

```
heap-extract-max(A)
  if heap-size(A) < 1
    then error "heap underflow!"
  max <- A[1]
  heapsize[A] <- heapsize[A] - 1
  heapify(A,1)
  return max
```

- 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
```

Can prove that any sorting procedure which uses only comparison must have time complexity $\Omega(n \lg n)$. You should read about techniques that can sort in $\Theta(n)$ time—“radix sort” in CLRS 8.3.