csce750 — Analysis of Algorithms
Fall 2019 — Lecture Notes: Binary heaps and Heapsort

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Heap definition

CLRS 6

A **max-priority queue** is a data structure that supports these operations:

- $\textsc{Insert}(H, x)$ — insert element x into the queue
- $\textsc{FindMax}(H)$ — return the largest element in the queue
- $\textsc{DeleteMax}(H)$ — remove the largest element from the queue

We will use a data structure called a **binary max-heap** to implement these.

Everything we say about max-priority queues and max-heaps can be inverted to get min-priority queues and min-heaps.

You have likely seen heaps before. We're covering them here for a few reasons:

- *They're a good example of how careful analysis can lead to better results than naive analysis.*
- *We'll use priority queues in other algorithms later.*
- *We'll also study an alternative implementation of the priority queue idea called a* Fibonacci heap*, and it will be useful to compare its performance to this standard 'binary heap.'*

2 Heap conditions

A heap physically stored as an array (starting at index 1), but we think of it as an essentially complete binary tree, stored top-to-bottom and left-to-right.

- $\text{parent}(i) = \lfloor i/2 \rfloor$.
- $\text{left}(i) = 2i$.
- $\text{right}(i) = 2i + 1$.

Rule: For every $i > 1$, a max-heap has $A[i] < A[\text{parent}(i)]$.

3 (Partially) Building a heap

Given an array A of length n and an index i, assume that the subtrees rooted at $\text{left}(i)$ and $\text{right}(i)$ are max-heaps, and turn the tree rooted at i into a max-heap:

```
MAXHEAPIFY(A, n, i)
  l = left(i)
  r = right(i)
  z = i
  if l ≤ n and A[z] ≤ A[l] then
    z = l
  end if
  if r ≤ n and A[z] ≤ A[r] then
    z = r
  end if
  if z ≠ i then
    swap A[i] with A[z]
    MAXHEAPIFY(A, n, z)
  end if
```

(Idea: Let $A[i]$ 'sink' as far as it needs to.)

4 MaxHeapify analysis

Let h denote the height of the tree rooted at i.

The time for MAXHEAPIFY at i is $\Theta(h)$.

5 Building a heap

We can iterate this process to turn an unordered array into a heap.

```
BUILDMAXHEAP(A, n)
  for i = ⌊n/2⌋, ..., 1 do
    MAXHEAPIFY(A, n, i)
  end for
```

Comments:

- The leaves (from $\lceil n/2 \rceil$ to n) are trivially heaps already. No need to MAXHEAPIFY them.
- Invariant: At the start of iteration i, each node $i+1, i+2, \ldots, n$ is the root of a heap.

6 BuildMaxHeap analysis: Trivial bound

$$T(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} O(\lg n) = O(n \lg n)$$

7 BuildMaxHeap analysis: A better bound

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \lg n \rfloor} \left(\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \right) \\ &\le cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &= cn \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h \\ &= cn \frac{1/2}{(1-(1/2))^2} = 2cn = O(n) \end{aligned}$$

(See CLRS Eq A.8.)

The expression $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ tells us the number of nodes in the tree that are roots of subtrees with height h. For example, for $h = 0$ there are $\left\lceil \frac{n}{2} \right\rceil$ leaves. The $O(h)$ is from our analysis of MAXHEAPIFY.

8 HeapSort

```
HEAPSORT(A, n)
  BUILDMAXHEAP(A, n)
  for i = n, ..., 2 do
    swap A[1] and A[i]
    MAXHEAPIFY(A, i − 1, 1)
  end for
```

9 Priority queue operations

```
INSERT(H, x)
  n = n + 1
  H[n] = x
  i = n
  while i > 1 and A[parent(i)] < A[i] do
    swap A[i] and A[parent(i)]
    i = parent(i)
  end while
```

```
FINDMAX(H)
  return H[1]
```

```
DELETEMAX(H)
  H[1] = H[n]
  n = n − 1
  MAXHEAPIFY(H, n, 1)
```