Lecture: Analysis of Algorithms (CS583 - 004)

Amarda Shehu

Spring 2019

1. Outline of Today's Class
    - Sorting in O(n lg n) Time: Heapsort

Heapsort

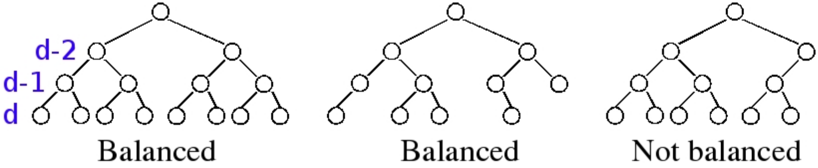
- Desired property of Mergesort: Time complexity of $O(nlgn)$
- Desired property of Insertion sort: Sorting in place
- Heapsort combines these two properties
- Heapsort illustrates a powerful algorithm design technique: a data structure organizes information during execution

Heap Data Structure

- Array object regarded as nearly complete binary tree
- Attributes:
  - $\mathrm{length(A)}$: # elements in A
  - $\mathrm{heap\_size(A)}$: # elements in heap
- Heap property: value of parent $\geq$ values of children
- Filled from root to leaves, left to right
  - Root of tree is stored in $A[1]$
  - Given index i of a node:
    - parent: $\mathrm{PARENT(i)} \leftarrow \lfloor \mathrm{i}/2 \rfloor$
    - left child: $\mathrm{LEFT(i)} \leftarrow 2\mathrm{i}$
    - right child: $\mathrm{RIGHT(i)} \leftarrow 2\mathrm{i} + 1$

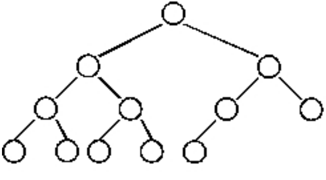
Heaps as a Balanced Binary Tree

- In a tree:
  - Depth of a node is distance of node from root
  - Depth of a tree is depth of deepest node
  - Height is the opposite of depth
  - Height and depth are often confused
- A binary tree of depth d is balanced if all nodes at depths 0 through $d - 2$ have two children
- Illustration of balanced and unbalanced binary trees:

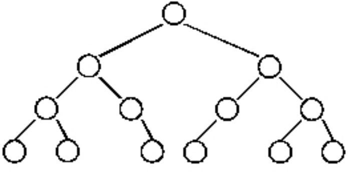


Heap as a Left-justified Balanced Binary Tree

- A balanced binary tree of depth d is left-justified if:
  1. It has 2^k nodes at depth k $\forall k < d$
  2. All leaves at depth d are as far left as possible



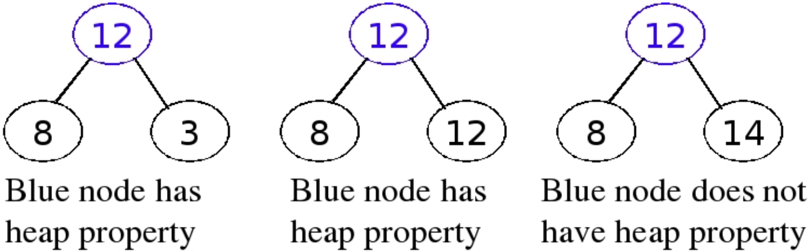
Left-justified



Not left-justified

Back to the Heap Property

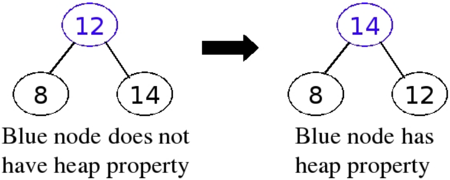
- In a max-heap: $A[\mathrm{PARENT(i)}] \geq \mathrm{A[i]}, \forall i \geq 1$
- In a min-heap: $A[\mathrm{PARENT(i)}] \leq \mathrm{A[i]}, \forall i \geq 1$
- We will focus on the heap property in max-heaps for sorting:



- Leaf nodes automatically have the heap property. Why?
- A binary tree is a heap if all nodes in it have the heap property
- What can you say about the root in a max/min-heap?

Maintaining the Heap Property in a Max-heap

- Given a node that does not have the heap property, one can give it the heap property by exchanging its value with that of the larger child:



- Note: Upon the exchange, the heap property may be violated in the subtree rooted at the child
- The $\mathrm{MAX-HEAPIFY}$ subroutine restores the heap property on the subtree rooted at index i
- How? The value at $A[i]$ is floated down in the max-heap

MAX-HEAPIFY: Pseudocode and Time Complexity

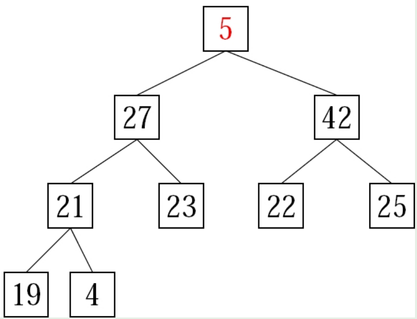
MAX-HEAPIFY(array A, index i)

1: **if** i is not a leaf and $A[\mathrm{LEFT(i)}]$ or $A[\mathrm{RIGHT(i)}] > A[i]$ **then**
2: let k denote larger child
3: swap$(A[i], A[k])$
4: $\mathrm{MAX-HEAPIFY(A, k)}$

Time Complexity: MAX-HEAPIFY

- Let $H(i)$ denote running time
- Show $H(i) \in O(lgn)$
- Hint: Down the tree we go

Trace MAX-HEAPIFY(A,1)



BUILD-MAX-HEAP: Pseudocode and Time Complexity

BUILD-MAX-HEAP(A, size n)

for $i \leftarrow \lfloor \frac{n}{2} \rfloor$ to 1 **do**
    MAX-HEAPIFY(A, i)

Time: BUILD-MAX-HEAP

- Why is $\lfloor \frac{n}{2} \rfloor$ bound sufficient?
- Hint: # internal nodes in heap?
- Let $B(n)$ be running time
- Show $B(n) \in O(nlgn)$

Trace BUILD-MAX-HEAP(A,10)

Array

9	4	6	10	2	11	16	1	15	8
1	2	3	4	5					10

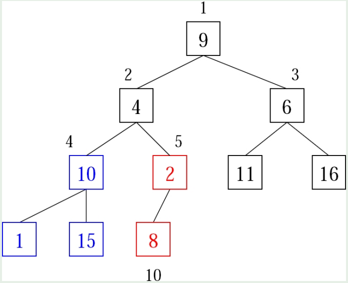


Figure: Trace on above array.

Tighter Asymptotic Bound on BUILD-MAX-HEAP

1. Show that an n-element heap has depth (and height) $\lfloor lgn \rfloor$.
2. Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

$$\begin{aligned} B(n) &= \sum_{i=1}^{n/2} H(i) = \sum_{h=0}^{\lfloor lgn \rfloor} \{\lceil \frac{n}{2^{h+1}} \rceil O(h)\} \\ &\in O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}) \end{aligned}$$

Note: $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$ (A.8)

Hence:

$$\begin{aligned} O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}) &= O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\ &= O(n) \end{aligned}$$

- Conclusion: A heap can be built in $O(n)$ time.

HEAPSORT: Pseudocode and Time Complexity

HEAPSORT(A)

```
1: BUILD-MAX-HEAP(A)
2: for i ← A.length to 2 do
3:     swap([A[1], A[i]])
4:     A.heap-size ← A.heap-size - 1
5:     MAX-HEAPIFY(A, 1)
```

Basic Idea Behind HEAPSORT

- What property holds for root after BUILD-MAX-HEAP?
- Why can we put it at index i?
- Why do we need to run MAX-HEAPIFY after swap?

Time:HEAPSORT

- HEAPSORT takes $O(nlgn)$.
- BUILD-MAX-HEAP runs in linear time - $O(n)$
- There are $n-1$ calls to MAX-HEAPIFY
- Each call takes $O(lgn)$ time
- So: T(HEAPSORT(A, n))
  $\in O(n+(n-1)lgn)$
  $\in O(nlgn)$

An Important Application of Heaps

- A priority queue maintains a set S of elements, each one associated with a *key*
- Max- or min-priority queues help to rank jobs for scheduling
- Operations for a max-priority queue:
  1. INSERT(S, x) inserts element x in the set S
  2. MAXIMUM(S) returns the element of S with the largest key
  3. EXTRACT-MAX(S) removes from S and returns the element with the largest key
  4. INSERT-KEY(S, x, k) increases the value of the key of x to the new value k, which is assumed to be at least as large as the current value of the key of x