CSCI 470: Heaps (continued), Partition Procedure

Vijay Chaudhary

September 11, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. A quick update
2. Heaps
3. Heapsort
4. Partition

A quick update

Course Material on GitHub

- Corrected a couple of mistakes from lecture notes 03: https://github.com/vijayko/csci-470 (a temporary arrangement)
- Please let me know if there are any errors in the lecture notes to be corrected.
- You can always clone the repo with all updated/corrected notes.
- Homework 01 has been posted and it's due on **Mon Sep 18**.
- Exam 1 will be pushed at least by one lecture session.
- If you are new to the class, please fill out this form: https://tinyurl.com/csci-470-form

Heaps

Heaps

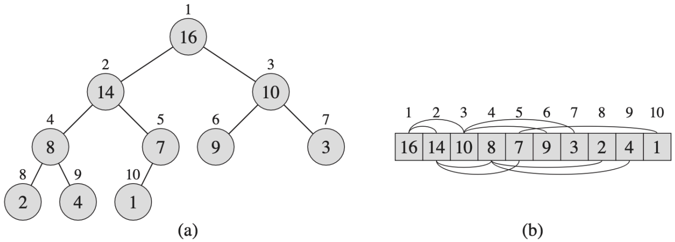

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Implementation of the heap

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Can we verify these using Figure 6.1?

Definition of height in a binary heap

- We define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root.
- What's going to be the height of node 4 in Figure 6.1?

Maintaining heap property

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heapsize and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heapsize and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

Assumptions while calling Max-Heapify

When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus, violating the max-heap property.

Max-Heapify: Illustration

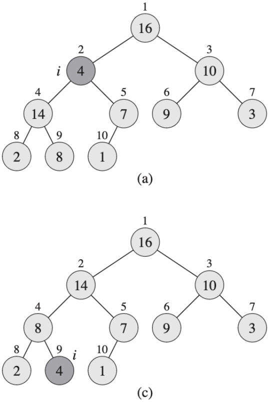

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Classwork

Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., n.

Building a heap

BUILD-MAX-HEAP(A)

$A.heapsize = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

Illustration

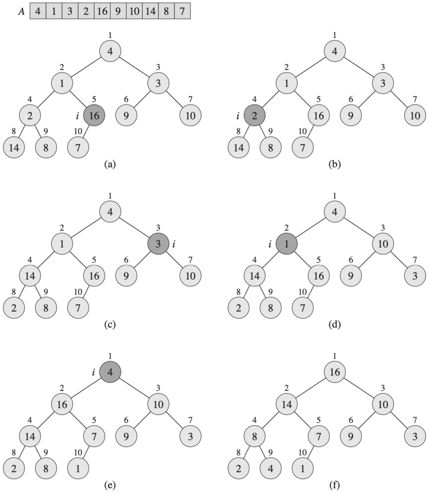

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Correctness of building a heap

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.

Referencing previous figure

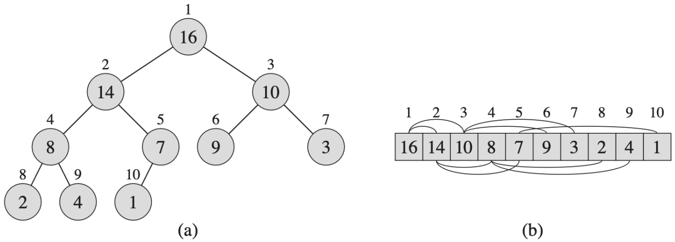

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Correctness

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., n is a leaf and is thus the root of a trivial max-heap.

Correctness continued

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the $\text{Max-Heapify}(A, i)$ to make node i a max-heap root. Moreover, the Max-Heapify call preserves the property that nodes $i + 1$, $i + 2$, ..., n are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Correctness continued

Termination: At termination $i = 0$. By the loop invariant, each node 1, 2, ..., n is the root of a max-heap. In particular, node 1 is.

Runtime

Simpler upper bound: We can compute a simpler upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. Although this is a correct upper bound, it's not a tight bound for this problem.

Runtime: A tight upper bound

- An n-element heap has a height of $\lfloor \lg n \rfloor$.
- An n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes for any height h. (Exercise 6.3-3 from CLRS)

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

Please refer Lecture Note 5 & 6 for this derivation.

Heapsort

Heapsort

HEAPSORT(A)

BUILD-MAX-HEAP(A)
for $i = A.length$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heapsize = A.heapsize - 1$
    MAX-HEAPIFY($A, 1$)

Runtime

Here line 1 takes $O(n)$, the loop runs n times out of which the body of the for loop runs $n - 1$. $\textsc{Max-Heapify}(A, 1)$ in line 5 takes $O(\lg n)$, because the height of the heap is $O(\lg n)$. Putting it together, the runtime of heapsort is $O(n \lg n)$.

Heapsort: Illustration

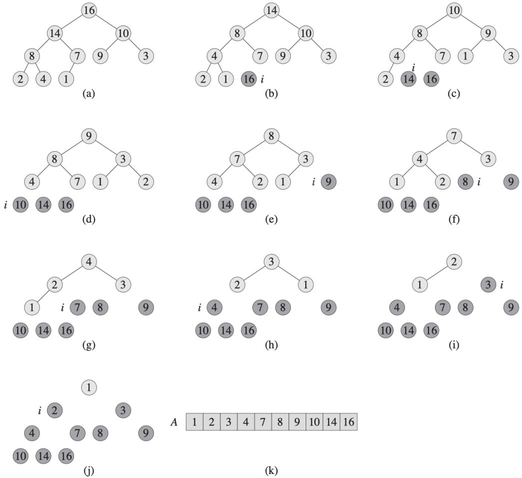

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Partition

Quick-Sort

Before we discuss quick sort, we will go over PARTITION procedure, used in this algorithm.

Partition Procedure

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Partition Illustration