CSCI 470: Solving Recurrence Relations (continued), Heaps

Vijay Chaudhary

September 06, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. A quick update
2. Solving Recurrence
3. Recursion Tree
4. Master method to solve recurrence relations
5. Heaps

A quick update

Course Material on GitHub

- Course materials have been posted here: https://github.com/vijayko/csci-470 (a temporary arrangement)
- Homework 01 will be posted today here.
- If you are new to the class, please fill out this form: https://tinyurl.com/csci-470-form
- No quiz today. :(

Solving Recurrence

Making a good guess

Unfortunately, there is no general way to make a good guess. However, we can use recursion trees to make a good guess. If a recurrence is similar to one you have seen before, that can be a really good guess.

Possible pitfalls

Sometimes, following a right guess may not lead to the expected implication. For instance, in case of $T(n) = 2T(\lfloor n/2 \rfloor) + n$, if we guess that $T(n) \leq cn$, we will end up with:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq 2 * c * (n/2) + n \\ &= cn + n \\ &= (c+1)n \\ &= O(n), \Leftarrow \textbf{wrong!!} \end{aligned}$$

which **does not imply**, $T(n) \leq cn$, because $cn + n > cn$ for any choice of c.

In this case, we will have to go with a stronger inductive hypothesis, such as $T(n) \leq cn - d$ where $d \geq 0$.

Change of variable

Sometimes changing variable can be helpful. Here is an example for that.

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n.$$

Let's rename $m = \lg n$. That gives us $n = 2^m$. (This is just "inverting" lg to the right side). Now we re-write the previous equation as:

$$T(2^m) = 2T(\lfloor 2^{m/2} \rfloor) + m$$

Change of variable continued

To make it even simpler, we can rename $S(m) = T(2^m)$, which yields:

$$S(m) = 2S(m/2) + m$$

which looks much similar to $T(n) = 2T(n/2) + n$, which was $T(n) = O(n \lg n)$.

Therefore, $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Recursion Tree

Solving recurrence using recursion tree

$$T(n) = 2T(n/2) + cn$$

Solving recurrence using recursion tree

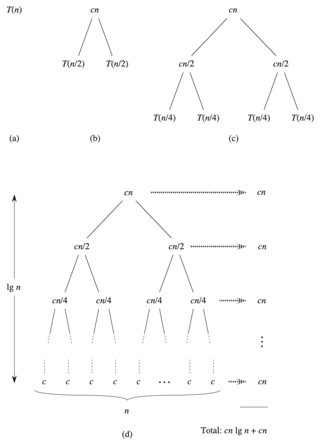

Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Master method to solve recurrence relations

Master Method

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

,

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Master Method: Example 1

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Example: $T(n) = 9T(n/3) + n$.

We have $a = 9$, $b = 3$, $f(n) = n$. Now, we calculate $n^{\log_3 9} = n^{\log_3 3^2} = n^2 = \Theta(n^2)$. Now, if we can show $f(n) = O(n^{\log_3 9 - \epsilon})$ for $\epsilon > 0$, then $T(n) = \Theta(n^{\log_3 9})$. For $\epsilon = 1$, we can see that $n^{\log_3 9 - 1} = n = f(n)$, and $f(n) = n = O(n^{\log_3 9 - 1})$, therefore, by case 1, $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$.

Master Method: Example 2

Case 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

Example: $T(n) = T(2n/3) + 1$.

Here, $a = 1$, $b = 3/2$, and $f(n) = 1$. $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = \Theta(1)$. And we have $f(n) = 1$, therefore we should go with case 2. By case 2, the $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_{3/2} 1} \lg n) = \Theta(\lg n)$.

Heaps

Heaps

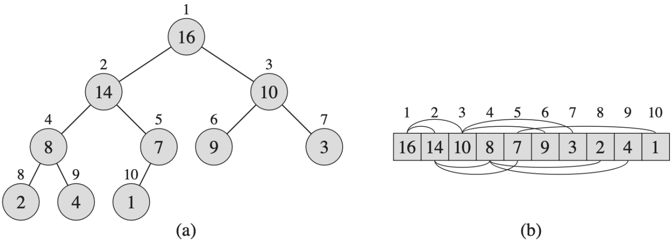

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

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?

1. What are the minimum and maximum numbers of elements in a heap of height h?
2. Show that an n-element heap has height $\lfloor \lg n \rfloor$.

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

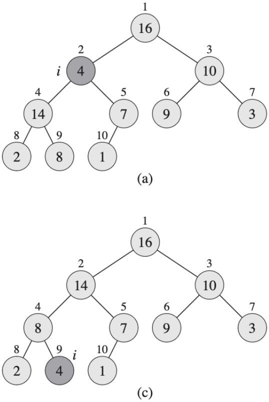

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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)

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 $i + 1, i + 2, ..., n$ is the root of a max-heap.

Referencing previous figure

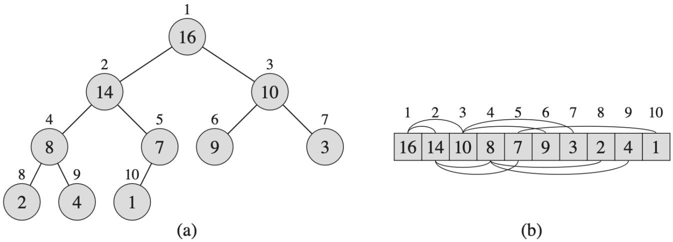

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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 cntd

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 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 cntd

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)

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