Set 3 Homework, Analysis of Algorithms

Jay R Bolton

May 10, 2012

- p 166: 6.5-6
- p 167: 6-1,6-2
- p 178: 7.2-1, 7.2-5
- p 180: 7.3-1
- p 284: 7.4-2
- p 185: 7-2, 7-4

Chapter 6

6.5-6 Do 'exchange' in 'Heap-Increase-Key' with one assignment.

The original:

$$
\begin{array}{l}
HeapIncreaseKey(A, i, key): \\
\quad if\ key < A[i] \\
\quad\quad error\ \text{“new key is smaller than current key”} \\
\quad A[i] = key \\
\quad while\ i > 1\ and\ A[Parent(i)] < A[i] \\
\quad\quad exchange\ A[i]\ with\ A[Parent(i)] \\
\quad I = Parent(i)
\end{array}
$$

With three assignments:

$$
\begin{array}{l}
HeapIncreaseKey(A, i, key): \\
\quad if\ key < A[i] \\
\quad\quad error\ \text{“new key is smaller than current key”} \\
\quad A[i] = key \\
\quad while\ i > 1\ and\ A[Parent(i)] < A[i] \\
\quad\quad tmp = A[i] \\
\quad\quad A[i] = A[Parent(i)] \\
\quad\quad A[Parent(i)] = tmp \\
\quad\quad i = Parent(i)
\end{array}
$$

With one assignment:

$HeapIncreaseKey(A, i, key)$:
  $if\ key < A[i]$
    $error$ "new key is smaller than current key"
  $while\ i > 1\ and\ A[Parent(i)] < key$
    $A[i] = A[Parent(i)]$
    $i = Parent(i)$
  $A[i] = key$

That was a real fun little puzzle.

6-1 **(a)** No. The counterexample is $[N, 1, 2, 3]$. BMH produces $[N, 3, 2, 1]$ while BMH' produces $[N, 3, 1, 2]$. Both are heaps.

(b) Max-Heap-Insert requires $\Theta(lg\ n)$ time. In Build-Max-Heap', we are looping that function $n - 1$ times. Everything else is constant, so our bound is $\Theta(n\ lg\ n)$.

6-2 **(a)** Same way, but you'd have to store or pass d and the children would be calculated with $di + 1$ through $di + d$ where 'i' is the current index.

(b) The height would be $log_d(n)$.

(c)

ExtractMax(A)
  if A.heapsize < 1
    error "$heap\ underflow$"
  $max = A[1]$
  $A[1] = A[A.\text{heapsize}]$
  MaxHeapify($A, 1$)
  return max

MaxHeapify(A, i)
  $largest = i$
  $for\ c = di + 1\ upto\ di + d$
    $if\ c \leq A.\text{heapsize}\ and\ A[c] > A[largest]$
      $largest = c$
  $if\ largest \neq i$
    $exchange\ A[i]\ with\ A[largest]$
    MaxHeapify($A, largest$)

ExtractMax remains unchanged, but MaxHeapify must now loop d times through all subtrees. Its complexity will be $\mathcal{O}(log_b n)$

(d, e) Both Insert and IncreaseKey can be implemented the same since neither depend on the selection of children.

6-3 **(a)**

2	3	4	5
8	9	12	14
16	∞	∞	∞

(b) Y[1,1] will be the least element in the matrix (least of the least of the columns and least of the least of the rows). If Y[1,1] is infinity/null, then there is no least element.

If Y[1,1] contains a non-null element then that means we have a least element. We have at least one element in that case, where m and n are 1.

Chapter 7

7.2-1 Prove: $T(n) = T(n-1) + \Theta(1)$ has complexity $\Theta(n^2)$.

$$
\begin{aligned}
\text{Inductive Hypothesis: } & T(n) \leq c \cdot n^2 \\
\text{Also: } & T(n) \geq c \cdot n^2 \\
& \text{Induction:} \\
T(n) &\leq b(n-1)^2 + n^2 \text{ for some constant b} \\
&\leq bn^2 + n^2 \\
&= (b+1)n^2 \\
&\leq c \cdot n^2 \\
T(n) &\geq b(n-1)^2 + n^2 \text{ for some constant b} \\
&\geq n^2 \\
&= 1 \cdot n^2 \\
&= c \cdot n^2
\end{aligned}
$$

7.2-5 For the minimum depth, our recurrence is $T(n) = 2T(n/2) + n$. The proportion will be one-half to one-half, which is our best case.

For the maximum depth, our recurrence is $T(n) = 2T(n-1) + n$. The proportion in this case is $\frac{n-1}{n}$ to $\frac{1}{n}$. This is the worst case.

The height of the minimum depth is $lg_2 n$ and the height of the maximum depth is n.

7.3-1 Because the worst case may be asymptomatic and the randomized version closer to average.

7.4-2 By induction with hypothesis: $T(n) \geq c \cdot n \ lg \ n$

The best-case recurrence is $T(n) = 2T(n/2) + n$, where each subproblem is partitioned evenly in two.

$$
\begin{aligned}
T(n) &\geq 2(n/2)lg(n/2) + dn \\
&= n \ lg(n/2) + dn \\
&= n \ lg \ n - n \ lg \ 2 + dn \\
&= n \ lg \ n - n + dn \\
&\geq c \cdot n \ lg \ n
\end{aligned}
$$

7-2 **(a)** With all elements equal, the running time would be worst case at $\mathcal{O}(n^2)$

(b) Rewrite partition so that all elements equal to the pivot are in the middle.

```
partition(a,p,r):
  q,t = p
  for j = p to r - 1:
    if a[j] < a[r]:
      exchange a[j] and a[t]
      exchange a[t] and a[q]
```

```
    q++, t++
  if a[j] == a[r]:
    exchange a[j] with a[t]
    t++
exchange a[r] and a[t]
return (q,t)
```

Wow that was really tricky. Took me forever to figure out.

(c) Not much to this one. Just make the recursive quicksort call on (A,p,q-1) and (A,t+1,r)

(d) We would take out the special condition that element values be distinct; now it is $\mathcal{O}(n\ lg\ n)$ even with repeated elements.

7-4 Woah neat! We partition the list as we normally do, and then repeatedly quicksort the first partition, working our way forward every partition, until we reach the end of the list.

The n-sized stack depth is achieved on the worst case, such as the sorted list, we will have to pass n, n-1, n-2, etc until it is finished.

To optimize the stack depth to $\Theta(lg\ n)$, we can take the smaller of the two partitions, sort it first, then sort the second partition using tail recursion.