4041 Homework 2

Fletcher Gornick

September 27, 2021

6.5

6.5-8

The operation `HEAP-DELETE(A,i)` deletes the item in node i from heap A. Give an implementation of `HEAP-DELETE` that runs in $O(\lg n)$ time for an n-element max-heap.

```
HEAP-DELETE(A,i)
  if (i < 1 or i > A.heap-size)
    return error "not a valid node"

  else
    swap(A[i], A[A.heap-size])
    popVal = A[A.heap-size]
    A.heap-size -= 1

    HEAP-INCREASE-KEY(A,i,A[i])
    MAX-HEAPIFY(A,i)
    return popVal
```

6.5-9

Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* a min-heap for k-way merging.)

The most efficient way to sort pre-sorted lists into one large list is to use a priority queue, or more specifically, a min-heap. First thing to do is put the first elements of each list into a binary heap structure. Once this is done, we can pop the root node, and replace it with it's corresponding next element, then the priority queue will heapify everything for us in $\lg k$ time. We keep appending the root nodes of the min-heap onto our new sorted list until all the lists in the heap have been emptied. When an array runs out of elements in our min-heap, we can just call `HEAP-DELETE` to completely remove the node for us, and finally continue until every element has been removed from the heap.

It takes $O(k)$ time to build the heap out of the lists. Each value is extracted from the heap in constant time $O(1)$, but we extract n times, so we can treat it as $O(n)$. And for re-organizing the heap, it takes $O(\lg k)$ time, which is called n times (for each time an element is removed) making it $O(n \lg k)$. Finally there are the times where we need to delete an element from the heap when a list runs out of elements. This takes $O(\lg k)$ time, and is called k times, making it $O(k \lg k)$. Therefore our whole algorithm runs in $O(k) + O(n) + O(n \lg k) + O(k \lg k) = O(n \lg k)$ time.

Technically the above time complexities weren't 100% accurate because every time `HEAP-DELETE` is called, there's one less element on the heap, making it's run time a bit more efficient than $k \lg k$. This also makes the `HEAPIFY` call a bit more efficient too, because instead of $n \lg k$, it's more like $n_1 \lg k + n_2 \lg(k-1) + \cdots + n_k \lg 1$, where $n_1, \ldots, n_k$ represent the number of elements in each input list. But this is essentially equal to $n \lg k$, and big-O notation is pretty lenient.

2.3

2.3-2

Rewrite the `MERGE` procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

```
MERGE(A,p,q,r)
  n1 = q - p + 1
  n2 = r - q

  new arr L[1 ... n1+1]
  for (int i = 1; i <= n1; i++)
    L[i] = A[p + i - 1]

  new arr R[1 ... n2+1]
  for (int j = 1; j <= n1; j++)
    L[i] = A[q + j]

  int i,j = 1
  int k = p;
  while (i <= n1 and j <= n2)
    if (L[i] < R[j])
      A[k++] = L[i++]          // assignment followed by increment
    else
      A[k++] = R[j++]

  if (i <= n1)
    for (; i <= n1; i++)       // initial condition already stated => not necessary
      A[k++] = L[i]            // k incremented after assignment, and i incremented
  if (j <= n2)                 // through the for loop
    for (; j <= n2; j++)
      A[k++] = R[j]
```

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

BASE CASE: $n = 2$. $2 \lg 2 = 2$, so the base case holds.

INDUCTIVE STEP: Assume $T(a) = a \lg a$ is true if $a = 2^b$ for some $b \in \mathbb{Z}, b > 1$. We can use this assumption to show that $T(c) = c \lg c$, where $c = 2^{b+1}$.

$$\begin{aligned} T(c) &= T(2^{b+1}) \\ &= 2T(2^b) + 2^{b+1} \\ &= 2 \cdot 2^b \lg(2^b) + 2^{b+1} \\ &= 2^{b+1}(\lg(2^b) + 1) \\ &= 2^{b+1}(\lg(2^b) + \lg(2)) \\ &= 2^{b+1} \lg(2 \cdot 2^b) \\ &= 2^{b+1} \lg(2^{b+1}) \\ &= c \lg c \end{aligned}$$

Since we know that $T(2) = 2 \lg 2$ (base case), and that $T(2^k) = 2^k \lg(2^k) \Rightarrow T(2^{k+1}) = 2^{k+1} \lg(2^{k+1})$ for $k > 1$ (inductive step), we know that the solution to the above recurrence must be $T(n) = n \lg n$.

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \ldots n]$ we recursively sort $A[1 \ldots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \ldots n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

We can express this recurrence as $T(n)$. First, note that every time we insert an element, it takes $O(n)$ time worst case. It also takes $T(n-1)$ time to sort the list before inserting the nth element. Finally, if the list only has one element, then the list can be sorted in constant time. So we can write the recurrence like this...

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n-1) + O(n) & \text{if } n > 1 \end{cases}$$

2.3-5

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

```
int BINARY_SEARCH(A, val) {
  return BINARY_SEARCH_HELPER(A, val, 0, A.length)
}


int BINARY_SEARCH_HELPER(A, val, p, r) {
  if (p > r) return -1
  q = floor((p+r) / 2)
  if (A[q] == val) return q
  if (A[q] > val)
    return BINARY_SEARCH_HELPER(A, val, p, q-1)
  if (A[q] < val)
    return BINARY_SEARCH_HELPER(A, val, q+1, r)
}
```

This algorithm has a worst case running time of $\Theta(\lg n)$ because every recursive call of $T(n)$ has run time of $T(n/2) + \Theta(1)$, so if we say n is a power of 2, and set it equal to 2^k for some k, we can solve this relation using induction. First, for the base case $n = 1$, $T(1) = 0$ because we don't need to do a compare. Next, if we assume $T(2^k) = \Theta(\lg 2^k) = \Theta(k)$ for some k, we can show $T(2^{k+1}) = \Theta(k + 1)$...

$$T(2^{k+1}) = T(2^k) + \Theta(1) = \Theta(k) + \Theta(1) = \Theta(k + 1)$$

Therefore the complexity of binary search with an array size of 2^k is $\Theta(k)$. Now if we say $n = 2^k$, we get that the run time complexity of binary search for an array of size n is $\Theta(\lg n)$.

7.2

7.2-1

Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

By representing $\Theta(n)$ as $c_1 n$ we can substitue Θ and T with constants c_1 and c_2.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &\le c_2(n-1)^2 + c_1 n \\ &= c_2 n^2 - 2c_2 n + c_2 + c_1 n \end{aligned}$$

If we assume $-2c_2 n + c_2 + c_1 n \le 0$ for some large n, then we get $T(n) \le c_2 n^2$. We can find the specific values of c_2 where this is the case.

$$\begin{aligned} -2c_2 n + c_2 + c_1 n &\le 0 \\ \Rightarrow c_2(1-2n) &\le -c_1 n \\ \Rightarrow c_2(2n-1) &\ge c_1 n \\ \Rightarrow c_2 &\ge \frac{c_1 n}{2n-1} \end{aligned}$$

and this fractions largest value is when $n = 1$, so $c_2 \ge c_1$, so finally we get

$$T(n) \le c_2 n^2 - 2c_2 n + c_2 + c_1 n \le c_2 n^2 - 2c_2 n + c_2 + c_2 n \le c_2 n^2$$

And since $T(n) \le c_2 n^2$ for some c_2, it must be the case that $T(n) = \Theta(n^2)$.

7.3

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

The chances of a randomized algorithm running with it's worst-case complexity is so small that it doesn't really make sence to represent it that way. A function like randomized quick sort is much asymptotically faster than it's $O(n^2)$ worst-case run time suggests, and it's because of this fact, that it makes much more sense to look at the algorithm's average run time instead.