CSE 5311 Homework 2 Solution

Problem 6.2-6

Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

Answer

If you put a value at the root that is less than every value in the left and right subtrees, then MAX-HEAPIFY will be called recursively until a leaf is reached. To make the recursive calls traverse the longest path to a leaf, choose values that make MAX-HEAPIFY always recurse on the left child. It follows the left branch when the left child is greater than or equal to the right child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, MAX-HEAPIFY will be called h times (where h is the heap height, which is the number of edges in the longest path from the root to a leaf), so its running time will be ,$\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which MAX-HEAPIFY's running time is $\Theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

Problem 6-2 Analysis of d-ary heaps

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY(A, i, k), which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

Answer

a.

We can represent a d-ary heap in a 1-dimensional array as follows. The root resides in $A[1]$, its d children reside in order in $A[2]$ through $A[d+1$, their children reside in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The following two procedures map a node with index i to its parent and to its j-th child (for $1 \leq j \leq d$), respectively.

```
D–ARY–PARENT( i )
        return  $\lfloor (i-2)/d+1 \rfloor$

D–ARY–CHILD( i ,  j )
        return  $d(i+1)+j+1$
```

To convince yourself that these procedures really work, verify that

```
D–ARY–PARENET(D–ARY–CHILD( i ,  j ) )  =  i ,
```

for any $1 \leq j \leq d$. Notice taht the binary heap procedures are a special case of the above procedures when $d = 2$.

b.

Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n) = \Theta(\lg d/\lg n)$.

c.

The procedure HEAP-EXTRACT-MAX given in the text for binary heaps works fine for d-ary heaps too. The change needed to support d-ary heaps is in MAX-HEAPIFY, which must compare the argument node to all d children instead of just 2 children. The running time of HEAP-EXTRACT-MAX is still the running time for MAX-HEAPIFY, but that now takes worst-case time proportional to the product of the height of the heap by the number of children examined at each node (at most d), namely $\Theta(d \log_d n) = \Theta(d \lg d/\lg n)$.

d.

The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for d-ary heaps too, assuming that HEAP-INCREASE-KEY works for d-ary heaps. The worst-case running time is still $\Theta(h)$, where h is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a d-ary heap, this is $\Theta(\log_d n) = \Theta(\lg d/\lg n)$.

e.

The HEAP-INCREASE-KEY procedure with two small changes works for d-ary heaps. First, because the problem specifies that the new key is given by the parameter k, change instances of the variable key to k. Second, change calls of PARENT to calls of D-ARY-PARENT from part (a).

In the worst case, the entire height of the tree must be traversed, so the worstcase running time is $\Theta(h) = \Theta(\log_d n) = \Theta(\lg d/\lg n)$.

Problem 7.2-2

What is the running time of QUICKSORT when all elements of the array A have the same value?

Answer

It is $\Theta(n^2)$, since one of the partitions is always empty (see exercise 7.1.2).

Problem 7-1 Hoare partition correctness

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C.A.R. Hoare:

```
HOARE-PARTITION(A, p, r)
  x = A[p]
  i = p - 1
  j = r + 1
  while TRUE
      repeat
          j = j - 1
      until A[j] <= x
      repeat
          i = i + 1
      until A[i] >= x
      if i < j
          exchange A[i] with A[j]
      else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$, showing the values of the array and auxiliary values after each iteration of the while loop in lines 4-13.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p..r]$ contains at least two elements, prove the following:

b. The indices i and j are such that we never access an element of A outside the subarray $A[p..r]$.

c. When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

d. Every element of $A[p..j]$ is less than or equal to every element of $A[j+1..r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two paritItions $A[p..j]$ and $A[j+1..r]$. Since $p \leq j < r$, this split is always nontrivial.

Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

Answer

Demonstration

At the end of the loop, the variables have the following values:

```
x = 13
j = 9
i = 10
```

Correctness

The indices will not walk of the array. At the first check $i < j$, $i = p$ and $j \geq p$ (because $A[p] = x$). If $i = j$, the algorithm will terminate without accessing "invalid" elements. If $i < j$, the next loop will also have indices i and j within the array, (because $i \leq r$ and $j \geq p$). Note that if one of the indices gets to the end of the array, then i won't be less or equal to j any more.

As for the return value, it will be at least one less than j. At the first iteration, either (1) $A[p]$ is the maximum element and then $i = p$ and $j = p < r$ or (2) it is not and $A[p]$ gets swapped with $A[j]$ where $j \leq r$. The loop will not terminate and on the next iteration, j gets decremented (before eventually getting returned). Combining those two cases we get $p \leq j < r$.

Finally, it's easy to observe the following invariant:

Before the condition comparing i to j, all elements $A[p..i-1] \leq x$ and all elements $A[j+1..r] \geq x$.

Initialization. The two repeat blocks establish just this condition.

Maintenance. By exchanging $A[i]$ and $A[j]$ we make the $A[p..i] \leq x$ and $A[j..r] \geq x$. Incrementing i and decrementing j maintain this invariant.

Termination. The loop terminates when $i \geq j$. The invariant still holds at termination.

The third bit follows directly from this invariant.

Implementation

There's some C code below.

```
#include <stdbool.h>

int hoare_partition(int A[], int p, int r) {
    int x = A[p],
        i = p - 1,
        j = r,
        tmp;

    while(true) {
        do { j--; } while (!(A[j] <= x));
        do { i++; } while (!(A[i] >= x));

        if (i < j) {
            tmp = A[i]; A[i] = A[j]; A[j] = tmp;
        } else {
            return j;
        }
    }
}

void quicksort(int A[], int p, int r) {
    if (p < r - 1) {
        int q = hoare_partition(A, p, r);
        quicksort(A, p, q + 1);
        quicksort(A, q + 1, r);
    }
}
```

Problem 8.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Answer

If it is linear for at least half of the inputs, then using the same reasoning as in the text, it must hold that:

$$\frac{n!}{2} \leq 2^n$$

This holds only for small values for n. Same goes for the other:

$$\frac{n!}{n} \leq 2^n$$

And:

$$\frac{n!}{2^n} \leq 2^n \Leftrightarrow n! \leq 4^n$$

All those have solutions for small $n < n_0$, but don't hold for larger values.

In contrast, insertion sort gets its work done in $\Theta(n)$ time in the best case. But this is a $1/n!$ fraction of the inputs, which is smaller than $1/2^n$.

Problem 8-3 Sorting variable-length items

1. You are given an array of integers, where different integers may have different number of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

2. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time.
   (Note that the desired order here is the standard alphabetical order; for example; a < ab < b.)

Answer

The numbers

For the numbers, we can do this:

1. Group the numbers by number of digits and order the groups Radix sort each group
2. Let's analyze the number of steps. Let G_i be the group of numbers with i digits and $c_i = |G_i|$. Thus:

$$T(n) = \sum_{i=1} nc_i \cdot i = n$$

The strings

For the strings, we can do this:

1. Groups the strings by length and order the groups
2. Starting i on the maximum length and going down to 1, perform counting sort on the ith character. Make sure to include only groups that have an ith character.

If the groups are subsequent subarrays in the original array, we're performing counting sort on a subarray ending on the last index of the original array.

Problem 9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

Answer

a. Sort the numbers using merge sort or heapsort, which take $\Theta(n \lg n)$ worst-case time. (Don't use quicksort or insertion sort, which can take $\Theta(n^2)$ time.) Put the i largest elements (directly accessible in the sorted array) into the output array, taking $\Theta(i)$ time.
Total worst-case running time: $\Theta(n \lg n + i) = \Theta(n \lg n)$ (because $i \leq n$).

b. Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX i times to get the i largest elements, in $\Theta(i \lg n)$ worst-case time, and store them in reverse order of extraction in the output array. The worst-case extraction time is $\Theta(i \lg n)$ because

   - i extractions from a heap with $O(n)$ elements takes $i \cdot O(\lg n) = O(i \lg n)$ time, and
   - half of the i extractions are from a heap with $\geq n/2$ elements, so those $i/2$ extractions take $(i/2)\Omega(\lg(n/2))\Omega(\lg n)$ time in the worst case.

   Total worst-case running time: $\Theta(n + i \lg n)$.

c. Use the SELECT algorithm of Section 9.3 to find the ith largest number in $\Theta(n)$ time. Partition around that number in $\Theta(n)$ time. Sort the i largest numbers in $\Theta(i \lg i)$ worst-case time (with merge sort or heapsort). Total worst-case running time: $\Theta(n + i \lg i)$.

Note that method (c) is always asymptotically at least as good as the other two methods, and that method (b) is asymptotically at least as good as (a). (Comparing (c) to (b) is easy, but it is less obvious how to compare (c) and (b) to (a). (c) and (b) are asymptotically at least as good as (a) because n, $i \lg i$, and $i \lg n$ are all $O(n \lg n)$. The sum of two things that are $O(n \lg n)$ is also $O(n \lg n)$.)

Problem 12.3-3

We can sort a given set of n numbers by first building a binary search tree contain- ing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst- case and best-case running times for this sorting algorithm?

Answer

Here's the algorithm:

```
TREE-SORT(A)
let T be an empty binary search tree
for i = 1 to n
        TREE-INSERT(T, A[i]);
INORDER-TREE-WALK(T.root)
```

Worst case: $\Theta(n^2)$—occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.

Best case: $\Theta(n \lg n)$—occurs when a binary tree of height $\Theta(\lg n)$ results from the repeated TREE-INSERT operations.

Problem 13.1-4

Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

Answer

After absorbing each red node into its black parent, the degree of each node black node is

- 2, if both children were already black,
- 3, if one child was black and one was red, or
- 4, if both children were red.

All leaves of the resulting tree have the same depth.

Problem 13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Answer

In the longest path, at least every other node is black. In the shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the length of the shortest path.

We can say this more precisely, as follows:

Since every path contains $\text{bh}(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $\text{bh}(x)$. By definition, the longest path from x to a descendant leaf has length $\text{height}(x)$. Since the longest path has $\text{bh}(x)$ black nodes and at least half the nodes on the longest path are black (by property 4), $\text{bh}(x) \leq \text{height}(x)/2$, so

$$\text{length of longest path} = \text{height}(x) \leq 2 \cdot \text{bh}(x) \leq \text{twice length of shortest path.}$$