Selected Homework Solutions – Unit 2

CMPSC 465

Exercise 6.1-1

Problem: What are the minimum and maximum numbers of elements in a heap of height h?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $1+2+2^2+2^3+\ldots+2^h=2^{h+1}-1$ elements (if it is complete) and at least $2^h-1+1=2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Exercise 6.1-3

Problem: Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

To prove:

For any subtree rooted at node k of a max-heap $A[1, 2, \ldots, n]$, the property $P(k)$:

The node k of the subtree rooted at k contains the largest value occurring anywhere in that subtree.

Proof:

Base Case:

When $k \in \left[\lfloor n/2 \rfloor + 1, n\right]$, k is a leaf node of a max-heap since $\lfloor n/2 \rfloor$ is the index of the last parent, and the subtree rooted at k just contains one node. Thus, node k contains the largest value in that subtree.

Inductive Step:

Let $k \in \left[1, \lfloor n/2 \rfloor\right]$ s.t. k is an internal node of a max-heap, and assume $\forall i \in Z$ s.t. $k < i \leq n$, $P(i)$ is true. i.e.

The node i of the subtree rooted at i contains the largest value occurring anywhere in that subtree.

[inductive hypothesis]

Now let us consider node k:

1. k's left child $2k$ and right child $2k + 1$ contain the largest value of k's left and right subtree, respectively. (by the inductive hypothesis that for $k < i \leq n$, $P(i)$ is true)
2. k's value is larger than its left child $2k$ and right child $2k + 1$. (by the max-heap property)

So, we can conclude that node k contains the largest value in the subtree rooted at k.

Thus, by the principle of strong mathematical induction, $P(k)$ is true for all nodes in a max-heap.

Exercise 6.1-4

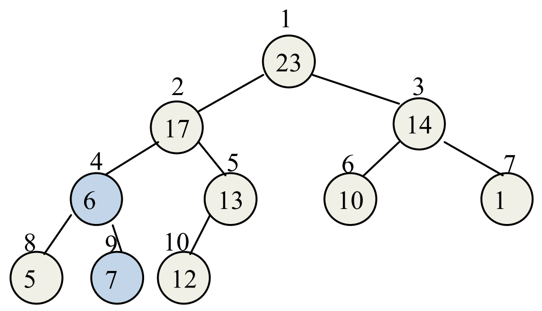
Problem: Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

The smallest element can only be one of leaf nodes. If not, it will have its own subtree and is larger than any element on that subtree, which contradicts the fact that it is the smallest element.

Exercise 6.1-6

Problem: Is the array with values (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) a max-heap?

Consider this illustration of the heap:



So, this array isn't a max-heap. As shown in the figure above, the value of node 9 is greater than that of its parent node 4.

Exercise 6.4-2

We argue the correctness of HEAPSORT using the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ lagest elements of $A[1..n]$, sorted.

Proof:
We need to show that this invariant is true prior to the first step, that each iteration of the loop maintains the invariant. It provides the property to show correctness of HEAPSORT.

Initialization:
Prior to the first iteration of the loop, $i = n$. The subarray $A[1..i]$ is a max-heap due to BUILD-MAX-HEAP. The subarray $A[i + 1..n]$ is empty, hence the claim about it being sorted is vacuously true.

Maintenance:
For each iteration, By the loop invariant, $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, so the $A[1]$ contains the $(n-i+1)$st largest element. The execution of line 3 exchanges $A[1]$ with $A[i]$, so it makes the subarray $A[i..n]$ contain the $(n-i+1)$ largest elements of $A[1..n]$, sorted. The heap size of A is decreased in to heap-size$(A) = i - 1$. Now the subarray $A[1..i-1]$ is not a max-heap, since the root node violates the map-heap property. But the children of the root maintain the max-heap property, so we can restore the max-heap property by calling MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1..i-1]$. Consequently, the loop invariant is reestablished for the next iteration.

Termination:
At termination, $i = 1$. By the loop invariant, the subarray $A[2.. n]$ contains the $n - 1$ largest elements of $A[1..n]$, sorted, and thus, $A[1]$ contains the smallest element of $A[1..n]$. $A[1..n]$ is a sorted array.

Exercise 6.4-3

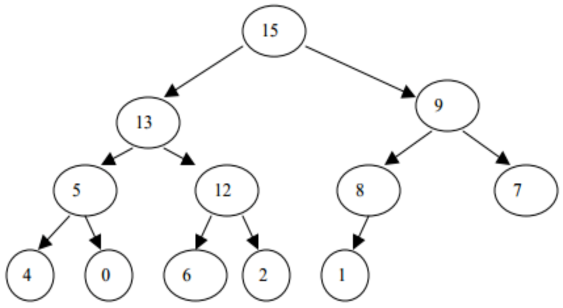
Problem: What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

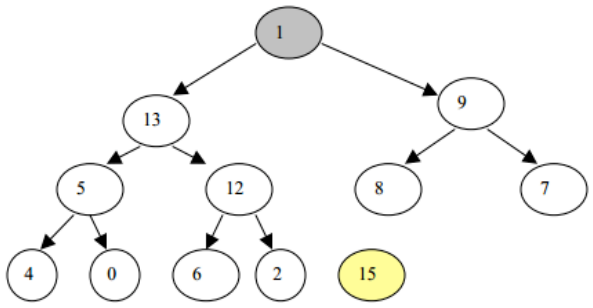
The running time of HEAPSORT on an array of length that is already sorted in increasing order is $\Theta(n \lg n)$, because even though it is already sorted, it will be transformed back into a heap and sorted.

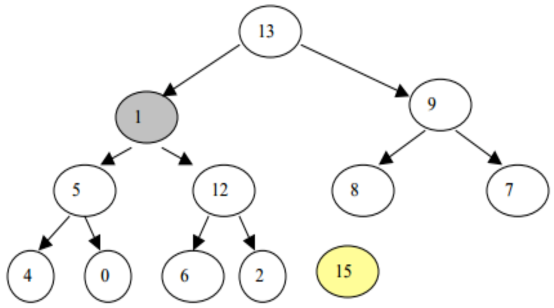
The running time of HEAPSORT on an array of length that is sorted in decreasing order will be $\Theta(n \lg n)$. This occurs because even though the heap will be built in linear time, every time the element is removed and HEAPIFY is called, it could cover the full height of the tree.

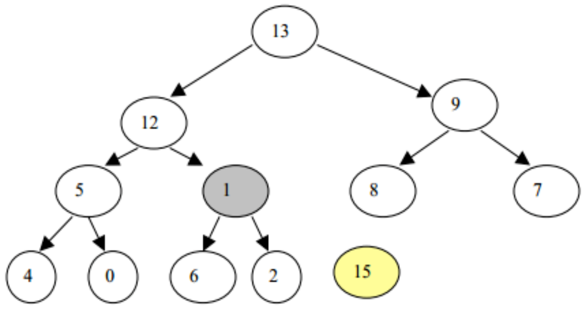
Exercise 6.5-1

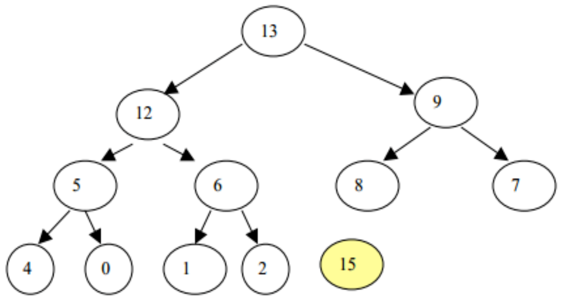
Problem: Illustrate the operation of HEAP-EXTRACT-MAX on the heap A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.





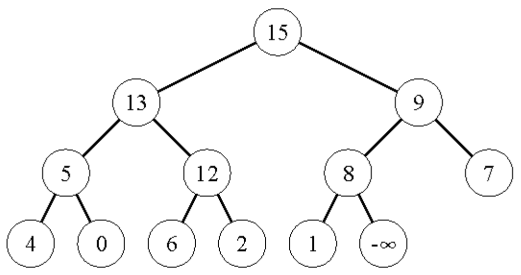




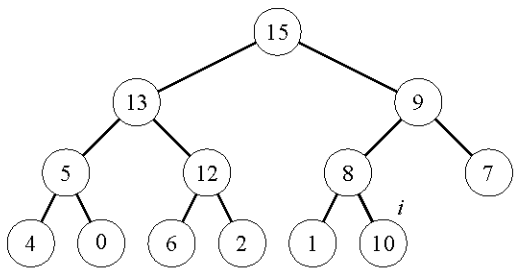


Exercise 6.5-2

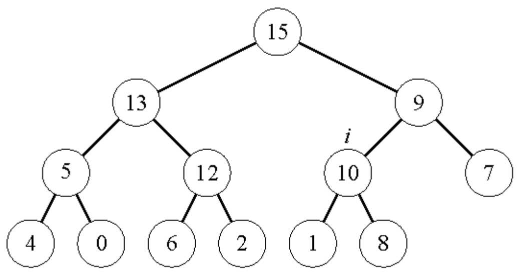
Problem: Illustrate the operation of MAX-HEAP-INSERT (A, 10) on the heap A = (15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1).



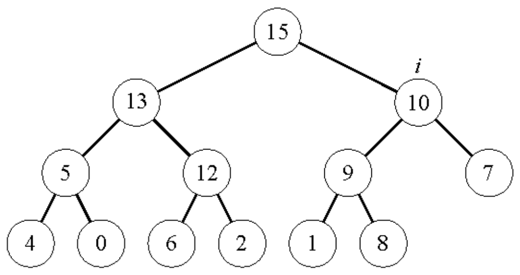
(a)



(b)



(c)



(d)

Exercise 7.1-3

Problem: Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Since each iteration of the for loop involves a constant number of operations and there are n iterations total, the running time is $\Theta(n)$.

Exercise 7.2-1

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

To prove:

For all integers n s.t. $n \geq 1$, the property $P(n)$:

The closed form $T(n) \leq cn^2$ and $T(n) \geq dn^2$, for some constants c and d, s.t. $c > 0$ and $d > 0$, matches the recurrence $T(n) = T(n–1) + \Theta(n)$.

Proof:

We will reason with strong induction.

Base case:

Let $n = 1$. $T(1) = T(0) + \Theta(1) = \Theta(1) \leq c1^2$, if c is large enough. $T(1) = T(0) + \Theta(1) = \Theta(1) \geq d1^2$, if d is small enough. So $P(1)$ holds.

Inductive Step:

Let $k \in Z$ s.t. $k \geq 1$ and assume $\forall i \in Z$ s.t. $1 \leq i \leq k–1$, $P(i)$ is true. i.e.

$T(i) \leq ci^2$ and $T(i) \geq di^2$ matches the recurrence. [inductive hypothesis]

Consider $T(k)$ to find an upper bound:

$T(k\ ) = T(k–1) + \Theta(k)$.	by using the recurrence definition
$\leq c(k–1)^2 + c_1k$	by subs. from inductive hypothesis, where c_1 is some constant
$= c(k^2 – 2k + 1) + c_1k$	since $(k–1)^2 = k^2 – 2k + 1$
$= ck^2 – (2c–c_1)\mathrm{k} + c$	by combining like terms
$\leq ck^2$	as long as $c > c_1$

Consider $T(k)$ to find a lower bound:

$T(k\ ) = T(k–1) + \Theta(k)$.	by using the recurrence definition
$\geq d(k–1)^2 + c_1k$	by subs. from inductive hypothesis, where c_1 is some constant
$= d(k^2 – 2k + 1) + c_1k$	since $(k–1)^2 = k^2 – 2k + 1$
$= dk^2 + (c_1–2d)\mathrm{k} + d$	by combining like terms
$\geq dk^2$	as long as $c_1–2d \geq 0$ or $0 < d \leq c_1/2$

So $P(k)$ is true.

So, by the principle of strong mathematical induction, $P(n)$ is true for all integers n s.t. $n \geq 1$, and constants $c > c_1$, $0 < d \leq c_1/2$.

Since $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$, we obtain that $T(n) = \Theta(n^2)$.

Exercise 7.2-4

Problem: Banks often record transactions on an account in order of the times of the transaction, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

INSERTION-SORT's running time on perfectly-sorted input runs in $\Theta(n)$ time. So, it takes almost $\Theta(n)$ running time to sort an almost-sorted input with INSERTION-SORT. However, QUICKSORT requires almost $\Theta(n^2)$ running time, recalling that it takes $\Theta(n^2)$ time to sort perfectly-sorted input. This is because when we pick the last element as the pivot, it is usually the biggest one, and it will produce one subproblem with close to $n - 1$ elements and one with 0 elements. Since the cost of PARTITION procedure of QUICKSORT is $\Theta(n)$, the recurrence running time of QUICKSORT is $T(n) = T(n - 1) + \Theta(n)$. In another problem, we, use the substitution method to prove that the recurrence $T(n) = T(n - 1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. So we use INSERTION-SORT rather than QUICKSORT in this situation when the input is almost sorted.

Exercise 8.4-2

Problem: Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserve its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

The worst case for bucket sort occurs when the all inputs fall into a single bucket, for example. Since we use INSERTION-SORT for sorting buckets and INSERTION-SORT has a worst case of $\Theta(n^2)$, the worst case run time for bucket sort is $\Theta(n^2)$.

By using an algorithm like MERGE-SORT with worst case run time time of $O(n \lg n)$ instead of INSERTION-SORT for sorting buckets, we can ensure that the worst case of bucket sort is $O(n \lg n)$ without affecting the average case running time.

Exercise 8.4-3

Problem: Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

$$\begin{aligned} E[X^2] &= 1^2 * P(\text{head in one flip}) + 0^2 * P(\text{tail in one flip}) \\ &= 1 * 1/2 + 0 * 1/2 \\ &= 1/2 \end{aligned}$$

$$\begin{aligned} E^2[X] &= E[X] * E[X] && \text{as the two flips are independent} \\ &= 1/2 * 1/2 && \text{as } E[X] = 1/2 \\ &= 1/4 \end{aligned}$$