6.2-3 *(p.156)* *What is the effect of calling* MAX-HEAPIFY(A, i) *when the element* $A[i]$ *is larger than its children?*

There is no effect. All three **if** conditions fail, *largest* is set to i, and the process terminates wihtout having changed anything in the heap. ■

6.2-4 *(p.156)* *What is the effect of calling* MAX-HEAPIFY(A, i) *for* $i > A.heap\text{-}size/2$*?*

If $i > A.heap\text{-}size/2$, then node i has no children (and it is either at the second lowest or lowest level of the binary tree). Moreover, LEFT(i) and RIGHT(i) are larger than $A.heap\text{-}size$, meaning that lines 3 and 6 (the first two **if** conditions) of the algorithm will return errors, because the array index will be out of range. ■

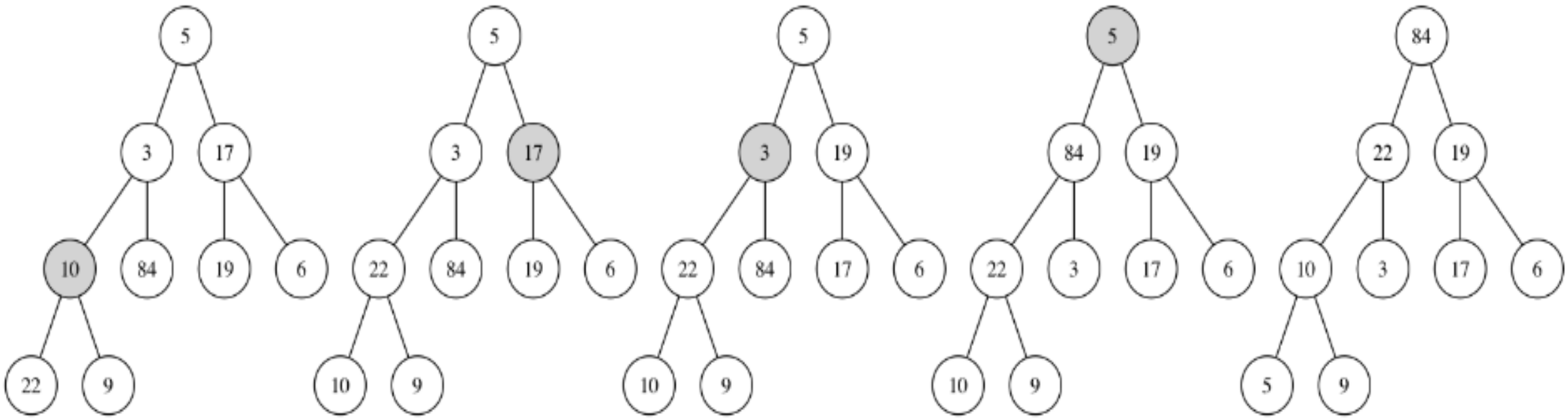
6.4-3 *(p.160)* *What is the running time of* HEAPSORT *on an array* A *of length* n *that is already sorted in increasing order? What about decreasing order?*

If A is sorted in increasing order, BUILD-MAX-HEAP will attain the maximum running time of $\Theta(n)$, since it tries to order the array in a decreasing order. The $n-1$ calls MAX-HEAPIFY$(A, 1)$ will take at most $O(\log_2(n))$ time (there are no particular time saves from max-heapifying an ordered array), hence the running time of HEAPSORT will be $O(n \log_2(n))$.

If A is sorted in decreasing order, MAX-HEAPIFY(A, i) has running time $O(1)$ for any i (since it never calls itself recursively, as $largest = i$ for all i. However, this makes no difference in the running time of BUILD-MAX-HEAP, as we still get $\Theta(n)$ running time due to the $\lfloor n/2 \rfloor$ calls to MAX-HEAPIFY. Here as well the $n-1$ calls MAX-HEAPIFY$(A, 1)$ will take at most $O(\log_2(n))$ time (completely reversing the order of an array certainly does not save time). Hence the running time of HEAPSORT will be $O(n \log_2(n))$. ■

> Using figure 6.3 as a model, illustrate the operation of `BUILD-MAX-HEAP` on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

Oh boy.



Exercise 6.3.2

> Why do we want the loop index i in line 2 of `BUILD-MAX-HEAP` to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

Otherwise we won't be allowed to call `MAX-HEAPIFY`, since it will fail the condition of having the subtrees be max-heaps. That is, if we start with 1, there is no guarantee that $A[2]$ and $A[3]$ are roots of max-heaps.

Exercise 6.3.3

> Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

First, let's observe that the number of leaves in a heap is $\lceil n/2 \rceil$ (exercise 6.1-7). Let's prove it by induction on h.

Base: $h = 0$. The number of leaves is $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil$.

Step: Let's assume it holds for nodes of height $h-1$. Let's take a tree and remove all it's leaves. We get a new tree with $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ elements. Note that the nodes with height h in the old tree have height $h-1$ in the new one.

We will calculate the number of such nodes in the new tree. By the inductive assumption we have that T, the number of nodes with height $h-1$ in the new tree, is:

$$T = \left\lceil \lfloor n/2 \rfloor / 2^{h-1+1} \right\rceil < \left\lceil (n/2)/2^h \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

As mentioned, this is also the number of nodes with height h in the old tree.

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

Upper bound

$$T(n) \le c_1(n-1)^2 + c_2 n$$
$$= c_1(n^2 - 2n + 1) + c_2 n$$
$$= c_1 n^2 - 2c_1 n + c_1 + c_2 n$$

In order to let $T(n) \le c_1 n^2$, we could set $-2c_1 n + c_1 + c_2 n \le 0$

$$-2c_1 n + c_1 + c_2 n \le 0$$
$$c_1(1 - 2n) \le -c_2 n$$
$$c_1(2n - 1) \ge c_2 n$$
$$c_1 \ge \frac{c_2 n}{2n-1}$$

Because the largest value of $\frac{n}{2n-1}$ for $n \ge 1$ is 1, we can choose $c_1 = c_2$

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

The procedure to prove lower bound is similar with upper bound.

Therefore, $T(n) = \theta(n^2)$

Q2: Exercise 7.2-2 (10 points)

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

- If all elements are the same, the quick sort partition return index $q = r$. This means, the problem with size n is reduced to one subproblem with size $n-1$. So the recurrence is $T(n) = T(n-1) + n$. By iteration method, $T(n) = \Theta(n^2)$.

Q3: Exercise 7.2-3 (10 points)

Show that the running time of `QuickSort` is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

- In each partition, the pivot element is always the smallest element. Therefore, each partition will produce two subarrays. The first subarray contains only one element (the smallerest element) and this element is in its correct position. The second subarray contains the remaining elements.
  In this case, the running time recurrence will be $T(n) = T(n-1) + n$. Thus, $T(n) = \Theta(n^2)$.

(CLRS 22.2-3) Analyse BFS running time if the graph is represented by an adjacency-matrix.

Solution: If the input graph for BFS is represented by an adjacency-matrix A and the BFS algorithm is modified to handle this form of input, the running time will be the size of A, which is $\Theta(V^2)$. This is because we have to modify BFS to look at every entry in A in the `for` loop of the algorithm, which may or may not be an edge.

22.3-3

Show the parenthesis structure of the depth-first search of Figure 22.4.

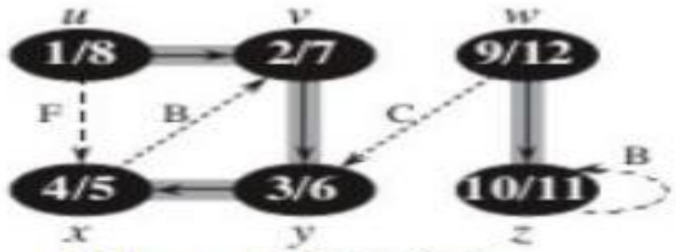
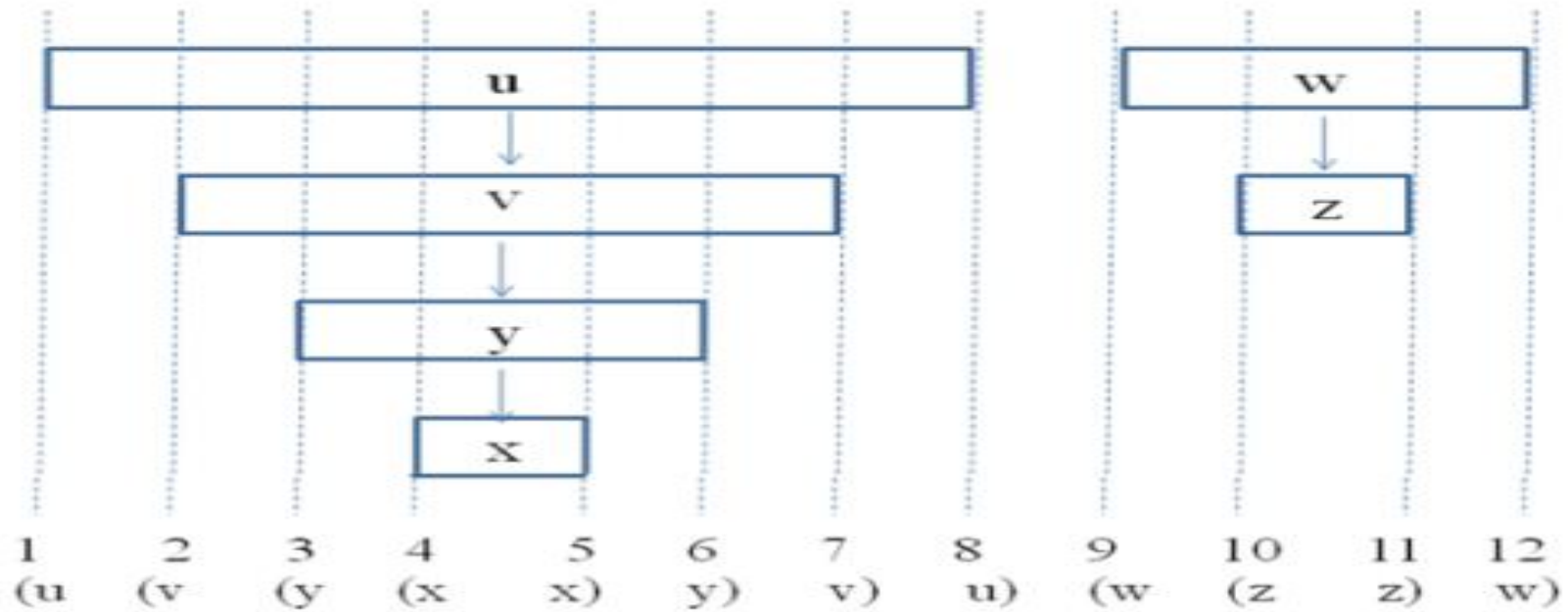


Figure 22.4 (p)

Solution:



Answer : (u (v (y (x x) y) v) u) (w (z z) w)

Exercises 24.1-4

Modify the Bellman-Ford algorithm so that it sets d[v] to -∞ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

Answer

```
Bellman-Ford-New(G, w, s)
        Initialize-Single-Source(G, s)
        for i <- 1 to |V[G]| - 1
                do for each edge (u,v) ∈ E[G]
                        do Relax[u, v, w]
        for each edge (u, v) ∈ E[G]
                do if d[v] > d[u] + w(u, v)
                        d[v] = -∞
        for each v such that d[v] = -∞
                do Follow-And-Mark-Pred(v)

Follow-And-Mark-Pred(v)
        if π[v] != nil and d[π[v]] != -∞
                do d[π[v]] = -∞
                Follow-And-Mark-Pred(π[v])
        else
                return
```

24.3-2

> Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Consider any graph with a negative cycle. $\textbf{RELAX}$ is called a finite number of times but the distance to any vertex on the cycle is $-\infty$, so Dijkstra's algorithm cannot possibly be correct here. The proof of theorem 24.6 doesn't go through because we can no longer guarantee that

$$\delta(s, y) \leq \delta(s, u).$$