Problem 1.

a) In each iteration, `Smallest` recursively calls itself with the first half of the array it was passed and the second half of the array it was passed. As such, the recurrence relation is given by

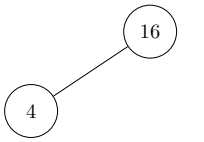
$$T(n) = 2T(n/2) + c$$

where c is some constant. We can now solve this using the master theorem. We have $a = 2$, $b = 2$, and $f(n) = c$. We have that $\log_b(a) = \log_2(2) = 1$, so $n^{\log_b(a)} = n$. We see that $f(n) \in O(n^{1-\epsilon})$ for $\epsilon = 0.5$ (although any $0 < \epsilon < 1$ would work), so case 1 applies. We therefore conclude $T(n) \in \Theta(n)$.

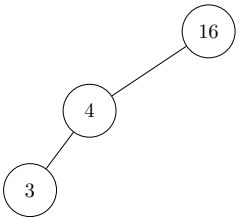
b) The result of each operation is shown below.

1. `Insert(16)`

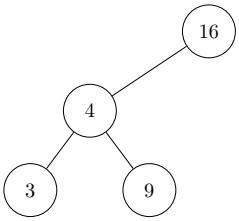
2. `Insert(4)`



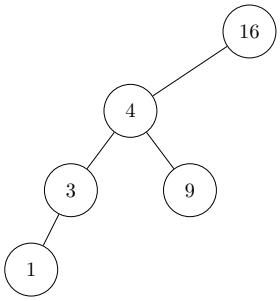
3. `Insert(3)`



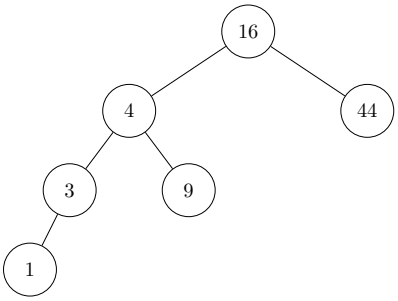
4. `Insert(9)`



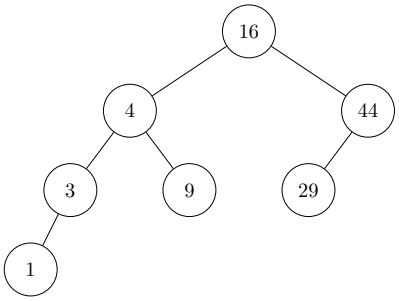
5. `Insert(1)`



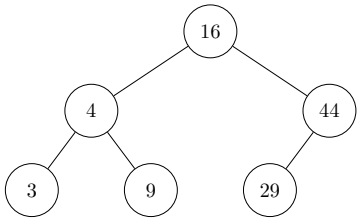
6. `Insert(44)`



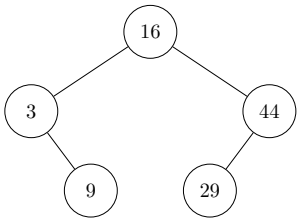
7. `Insert(29)` 🚑



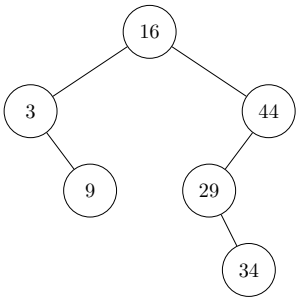
8. `Delete(1)` 🚒



9. `Delete(4)` 🚐

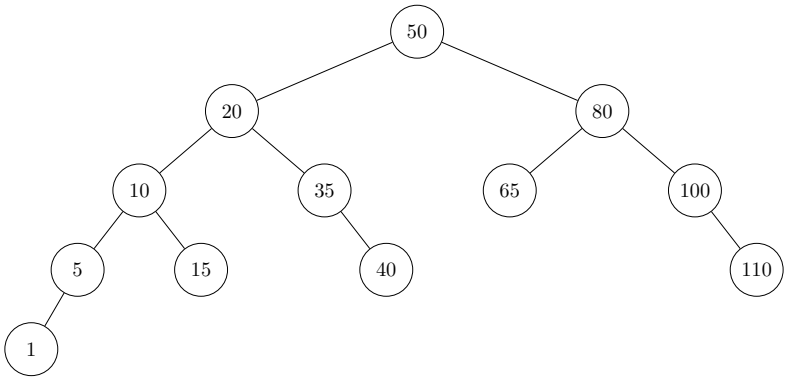


10. `Insert(34)` 🚚

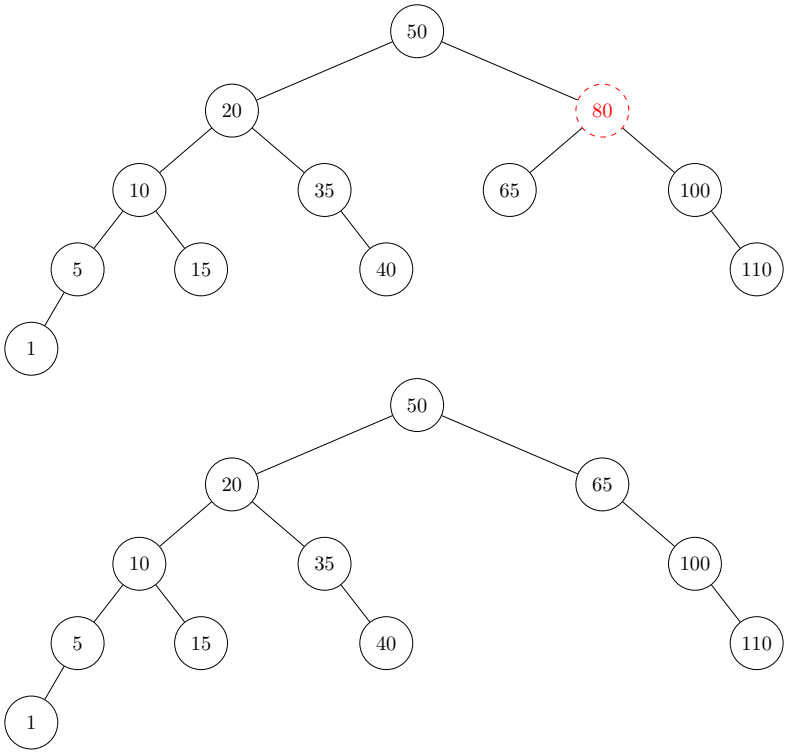


c) The result of each operation is shown below.

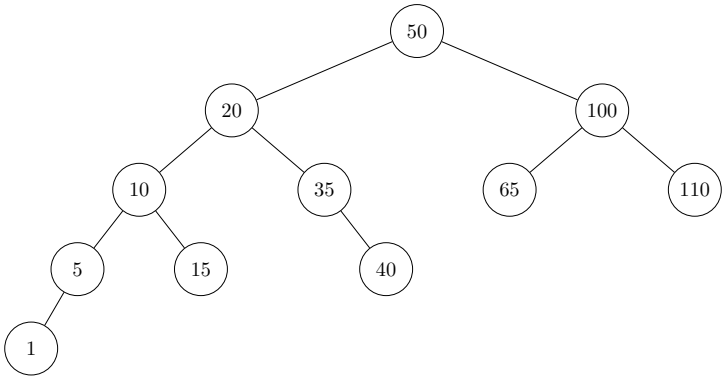
1. Original ✨AVL✨ tree with height 4.



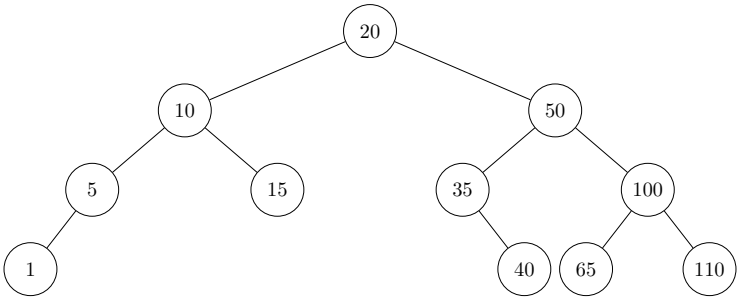
2. `Delete(80)`



3. Single rotate left



4. Single rotate right



d) The state of the hash table after each operation is given below.

1. `Insert(18)` $\rightarrow h(18) = 0$

0	18
1	
2	
3	
4	
5	
6	
7	
8	

2. `Insert(22)` $\rightarrow h(22) = 4$

0	18
1	
2	
3	
4	22
5	
6	
7	
8	

3. `Insert(32)` $\rightarrow h(32) = 5$

0	18
1	
2	
3	
4	22
5	32
6	
7	
8	

4. `Insert(8)` $\rightarrow h(8) = 8$

0	18
1	
2	
3	
4	22
5	32
6	
7	
8	8

5. `Insert(6)` $\rightarrow h(6) = 6$

0	18
1	
2	
3	
4	22
5	32
6	6
7	
8	8

6. `Insert(10)` $\rightarrow h(10) = 1$

0	18
1	10
2	
3	
4	22
5	32
6	6
7	
8	8

7. `Insert(20)` $\rightarrow h(20) = 2$

0	18
1	10
2	20
3	
4	22
5	32
6	6
7	
8	8

8. `Insert(16)` $\rightarrow h(16) = 7$

0	18
1	10
2	20
3	
4	22
5	32
6	6
7	16
8	8

9. `Insert(2)` $\rightarrow h(2) = 2$

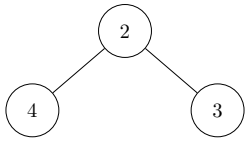
0	18
1	10
2	$20 \leftrightarrow 2$
3	
4	22
5	32
6	6
7	16
8	8

10. `Insert(29)` $\rightarrow h(29) = 2$

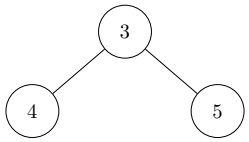
0	18
1	10
2	$20 \leftrightarrow 2 \leftrightarrow 29$
3	
4	22
5	32
6	6
7	16
8	8

e) Description of algorithm: let A be the final array that will contain all n elements. We first construct a min-heap from the smallest element of each of the k sorted lists. We then extract the first element from the min-heap and add it to A. After this, we add the next smallest element from the list from which the extracted element came from. We repeat this process until all elements have been added to A.

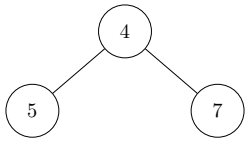
We now demonstrate this algorithm with an example. Let $L_1 = [2, 5]$, $L_2 = [4, 8]$, and $L_3 = [3, 7]$ be our sorted lists. We first construct a min-heap with the first elements of each of the sorted lists (2, 4, 3) as shown below.



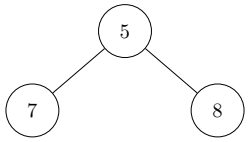
We extract 2 from the min-heap and place it into A. At this point, $A = [2]$. Since this element came from L_1, we place 5 into the min-heap now.



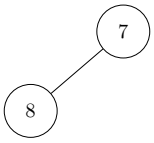
We extract 3 from the min-heap and place it into A. At this point, $A = [2, 3]$. Since this element came from L_3, we place 7 into the min-heap now.



We extract 4 from the min-heap and place it into A. At this point, $A = [2, 3, 4]$. Since this element came from L_2, we place 8 into the min-heap now.



We extract 5 from the min-heap and place it into A. At this point, $A = [2, 3, 4, 5]$. This element came from L_1, but we have gone through all elements of L_1. As such, we do not insert anything into the min-heap.

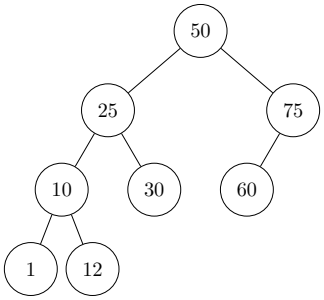


We extract 7 from the min-heap and place it into A. At this point, $A = [2, 3, 4, 5, 7]$. This element came from L_3, but we have gone through all elements of L_3. As such, we do not insert anything into the min-heap.

We finally extract 8 from the min-heap and place it into A. We then have $A = [2, 3, 4, 5, 7, 8]$.

f)

- Suppose $|R_v| \geq |L_v|$. If we assume $|L_v| = k$, then $|R_v| = 2k$ in the worst case. Since the entire tree contains n nodes, we can say $k + 2k + 1 = n$. This means that $k = (n-1)/3$ in the worst case and therefore $|R_v| = 2(n-1)/3$. Since only the root needs to be nearly balanced, we can achieve the maximum tree height through a linear distribution of nodes on the right side (basically a linked list after the root). As such, the maximum number of levels is $2(n-1)/3+1$ making the maximum height $2(n-1)/3$. We conclude that for a tree with 7 nodes, the maximum height is $2(7-1)/3 = 4$.
- As given in problem 7 of problem set #3, solving the recurrence yields $H_{\max}(n) = \log_{3/2}((n+2)/3) + 1$. For $n = 8$, $H_{\max}(8) \approx 3.97$. Below is an example where we attempt to place as many nodes as possible in the left direction.



As we can see, there is no way to move a node such that the height of the tree is greater than 3 while ensuring the tree is nearly balanced. It appears that using the solution to

the equation provides an upper bound to the maximum height if we take the floor of the result (i.e. $\lfloor 3.97 \rfloor = 3$).

- As shown in the previous part, the height of a nearly balanced tree is in $O(\log(n))$. Also, when searching for an element, we know the worst case is when we need to search all the way from the root to the lowest node. Since the height is in $O(\log(n))$, we conclude that the running time for searching for an element is in $O(\log(n))$.

g) We have 10 bottles of wine 🍷. From problem 9 of problem set #3, we claim we only need $k = \lfloor \log(n) \rfloor + 1 = 4$ testers 😋. Consider the binary representation of 10, 1010_2, which has $k = 4$ digits. The poisonous bottle 😭 has an unknown index X where $0001_2 \leq X \leq 1010_2$. Tester i tests all bottles of wine whose index (in binary) is 1 at position i. If tester i dies after a month 💀, it means that the index of the poisonous bottle in binary has a 1 at position i. Otherwise, if the tester survives 😎, it means that the index of the poisonous bottle in binary has a 0 at position i. This allows us to determine exactly which bottle of wine is poisoned.

For example, let's say the fifth bottle (0101_2) is poisoned. Tester 1 tests bottles 1 (0001_2), 3 (0011_2), 5 (0101_2), 7 (0111_2), and 9 (1001_2). Since tester 1 dies, we now know the poisoned bottle must contain a 1 at binary position 1. We repeat similar logic for testers 2, 3, and 4 to determine if the poisoned bottle must contain a 1 at binary position 2, 3, and 4 respectively. Eventually, this allows us to conclude the index of the poisoned bottle of wine is $0101_2 = 5$.

Problem 2.

a) For a 3-ary heap implicitly represented by array A, given index i we have

- $A[\lfloor \frac{i-2}{3} + 1 \rfloor]$ is the parent of $A[i]$,
- $A[3i - 1]$ is the left child of $A[i]$,
- $A[3i]$ is the middle child of $A[i]$,
- $A[3i + 1]$ is the right child of $A[i]$.

b) We define the height h as the number of edges on the longest root-to-leaf path. To calculate the height of a 3-ary heap with n, elements, we note that the number of nodes at some level i is 3^i. Let's say that our 3-ary heap contains filled levels from level 0 up to and including level k. Then the number of nodes up to and including level k is

$$\sum_{i=0}^{k} 3^i = \frac{3^{k+1} - 1}{3 - 1} = \frac{3^{k+1} - 1}{2}.$$

We now note that the last node, node n, either resides in the last filled level (level k) or the level after the last filled level (level $k + 1$). Therefore we have the inequality

$$\frac{3^{k+1} - 1}{2} \leq n \leq \frac{3^{k+2} - 1}{2}.$$

Multiplying by 2, adding 1, applying a logarithm with base 3, and subtracting 1 from all expressions yields

$$k \leq \log_3(2n + 1) - 1 \leq k + 1.$$

Notice that if the last node, node n, is the last node in level k, then the last node is k edges away from the root node. Otherwise, the last node is in level $k + 1$ and $\log_3(2n + 1) - 1$ will not be an integer value. As a result, we conclude that applying the ceiling operator to $\log_3(2n + 1) - 1$ will yield the right level. We conclude the height the height of a 3-ary heap of n elements is given by

$$h = \lceil \log_3(2n + 1) - 1 \rceil.$$

c) The code for `MaxHeapify` for a 3-ary max-heap is provided below, written in Lua. Note that Lua arrays, unlike arrays in most programming languages, begin indexing at 1.

```
1  function GetLeftChild(i)
2      return 3 * i - 1
3  end
4
```

```
function GetMiddleChild(i)
    return 3 * i
end

function GetRightChild(i)
    return 3 * i + 1
end

function GetHeapSize(A)
    return #A
end

function MaxHeapify(A, i)
    LeftChild = GetLeftChild(i)
    MiddleChild = GetMiddleChild(i)
    RightChild = GetRightChild(i)
    HeapSize = GetHeapSize(A)
    Largest = i

    if LeftChild <= HeapSize and A[LeftChild] > A[Largest] then
        Largest = LeftChild
    end

    if MiddleChild <= HeapSize and A[MiddleChild] > A[Largest] then
        Largest = MiddleChild
    end

    if RightChild <= HeapSize and A[RightChild] > A[Largest] then
        Largest = RightChild
    end

    if not (Largest == i) then
        A[i], A[Largest] = A[Largest], A[i]
        MaxHeapify(A, Largest)
    end
end
```

We can see that the number of times `MaxHeapify` calls itself is bounded by the height of the 3-ary heap. From the previous part, we know the height is given by $h = \lceil \log_3(2n+1) - 1 \rceil$. We note that the dominant term in the height is $\log_3(2n+1)$ which is trivially in $O(\log_3(n))$.

We also know

$$\log_3(n) = \frac{\log(n)}{\log(3)} \in O(\log(n)).$$

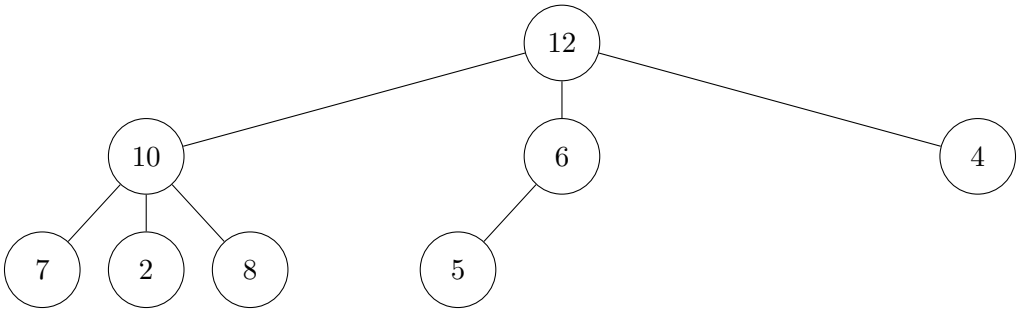
As such, we can conclude that the running time of `MaxHeapify` is in $O(\log(n))$.

d) The code for `BuildMaxHeap` for a 3-ary max-heap is provided below, written in Lua. Note that line 2 is essentially declaring a loop that starts its counter at $\lfloor \text{HeapSize}(A)/2 \rfloor$ and decrements down to 1.

```
1 function BuildMaxHeap(A)
2     for i = math.floor(GetHeapSize(A) / 2), 1, -1 do
3         MaxHeapify(A, i)
4     end
5 end
```

We can see that `BuildMaxHeap` makes about $n/2$ calls to `MaxHeapify`. As a result, we can conclude that the running time of `BuildMaxHeap` is in $O(n \log(n))$.

e) After invoking `BuildMaxHeap`, we have $A = [12, 10, 6, 4, 7, 2, 8, 5]$. This indeed satisfies the conditions of a 3-ary heap as shown below.



Problem 3.

The algorithm `FindInRange` is given below, written in Lua. Note that Lua arrays begin indexing at 1. `FindInRange` is initially called with $i = 1$ (i.e. the root node).

```
function GetLeftChild(i)
    return 2 * i
end

function GetRightChild(i)
    return 2 * i + 1
end

function GetHeapSize(A)
    return #A
end

function FindInRange(A, i, x, y)
    HeapSize = GetHeapSize(A)

    if A[i] < x then
        return
    end

    if A[i] <= y then
        print(A[i])
    end

    local LeftChild = GetLeftChild(i)
    local RightChild = GetRightChild(i)

    if LeftChild <= HeapSize then
        FindInRange(A, LeftChild, x, y)
    end

    if RightChild <= HeapSize then
        FindInRange(A, RightChild, x, y)
    end
end
```

In the worst case, all elements satisfy $x \leq z \leq y$. In this case, `FindInRange` touches each element exactly once and hence the worst case running time is in $O(n)$. We conclude $O(n) \in O(k)$ since $n = k$ in the worst case.

The algorithm is more efficient than a linear search (unless all elements satisfy $x \leq z \leq y$ in which case the efficiency is equal). Notice that if the root of any given subtree is less than x, that means all children of that subtree will be less than x. This means all the elements in that subtree won't satisfy $x \leq z \leq y$. This knowledge gives us the ability to not search the subtree if its root is less than x. This means that in an average case, our algorithm doesn't have to iterate recursively through all elements in the tree making it more efficient than a linear search which must iterate through all elements of A no matter what.

Problem 4.

- The worst case occurs when, for each recursive call, the partitions produced during partitioning are of size $n-1$. When this happens, each recursive call processes a partition that is only one smaller than the previous recursive call. This means we require $n-1$ recursive calls until we reach a partition of size 1. Since each recursive call i needs to do $n-i$ comparisons against the jars and $n-i-1$ comparisons against the lids, the running time in the worst case is in $O(n^2)$.

  More formally, the number of key comparisons required by the algorithm is given by the recurrence

  $$T(n) = \begin{cases} 0, & n \leq 1, \\ T(n_1) + T(n-1-n_1) + (2n-1), & n \geq 2, \end{cases}$$

  where n_1 is the pivot chosen. Note that $0 \leq n_1 \leq n-1$. The recurrence is similar to quicksort with the difference that each recursive call needs to do $2n-1$ comparisons (we compare a lid against n jars and a jar against $n-1$ lids, so $n+n-1 = 2n-1$).

  The worst case occurs when $n_1 = n-1$. Then we have

  $$T(n) = \begin{cases} 0, & n \leq 1, \\ T(n-1) + (2n-1), & n \geq 2. \end{cases}$$

  Expanding out the recurrence, we have

  $$\begin{aligned} T(n) &= T(n-1) + (2n-1) \\ &= T(n-2) + (2n-3) + (2n-1) \\ &= T(n-3) + (2n-5) + (2n-3) + (2n-1) \\ &= \dots \\ &= T(1) + 3 + 5 + \dots + (2n-5) + (2n-3) + (2n-1) \\ &= 3 + 5 + \dots + (2n-5) + (2n-3) + (2n-1). \end{aligned}$$

  This is essentially all odd numbers up to $2n-1$ except for 1. So we have

  $$\begin{aligned} T(n) &= \sum_{i=1}^{n} (2i-1) - 1 \\ &= n^2 - 1. \end{aligned}$$

  So, $T(n) \in \Theta(n^2)$ in the worst case.

- We have $n = 7$. We assume our algorithm solves all subproblems correctly (i.e. our algorithm holds for any array with size < 7).

We first take the last lid l_4 as the pivot. We then compare this lid against all n jars. We are able to match l_4 with j_4. Furthermore, all jars smaller than l_4 are placed to the left of j_4 (let us denote this set by J_s) and all jars larger than l_4 are placed to the right of j_4 (let us denote this set by J_l).

We then use jar j_4 to compare against all lids (except l_4). This allows us to partition all lids smaller than j_4 to the left of l_4 (let us denote this set by L_s) and all lids larger than j_4 to the right of l_4 (let us denote this set by L_l).

We now apply our algorithm to each new jar and lid pair, i.e., we use our algorithm on J_s and L_s, then on J_l and L_l. Each of these sets contain $n = 3$ elements, so based on our assumption, our algorithm will match all jars to their corresponding lids.

- As described in the previous part, the worst case running time is in $O(n^2)$. This scenario is guaranteed to happen if the lids are in sorted order from smallest to largest: for each recursive call, the lid selected is the largest in the partition, so all other lids are placed in partition L_s.

  The best case occurs when, for each recursive call, the partitions produced during partitioning are approximately of size $n/2$. When this happens, each recursive call produces a partition that is about half the size of the previous recursive call. This means we require approximately $\log(n)$ recursive calls until we reach a partition of size 1. Since each recursive call i needs to do $n - i$ comparisons, the running time in the best case is in $O(n \log(n))$.