EC504 Homework 3

Problem 1

Determine whether the following statements are true or false, and briefly explain why:

1A

If doubling the size ($N \rightarrow 2N$) causes the execute time $T(N)$ of an algorithm to increase by a factor of 4, then $T(N) \in O(4N)$.

False. Assume that $T(N) \in O(4N)$. Then by definition:

$$\exists c,\ N_0\ s.t.\ T(N) \leq 4cN\ \forall N \geq N_0$$

Assume we operate in the asymptotic region, that is $N \geq N_0$; then $T(N) \leq 4cN$. If we double the input size from $N \rightarrow 2N$, then per the asymptotic bound:

$$T(2N) \leq 4c(2N) \Longrightarrow T(2N) \leq 8cN$$

But we also assumed that doubling the input size from $N \rightarrow 2N$ quadruples the run time:

$$T(2N) = 4T(N) \Longrightarrow T(2N) \leq 16cN$$

Thus, by doubling the input size from $N \rightarrow 2N$, it is possible to violate the $O(4N)$ asymptotic bound; therefore, by contradiction, $T(N) \notin O(4N)$.

1B

The height of a binary tree is the maximum number of edges from the root to any leaf path. The maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$.

True. Let $N(h) = 2^{h+1} - 1$ be the maximum number of nodes found in a binary tree of height h. Use proof by induction to prove the claim.

Base case: $h = 0$. If a tree has height $h = 0$, then it consists just of the root node. Evaluating the above equation:

$$N(0) = 2^1 - 1 = 2 - 1 \Longrightarrow N(0) = 1$$

The base case holds, so we can proceed with the induction step.

Induction step: Assume the claim holds for h, namely the maximum number of nodes in a binary tree of height h is $N(h) = 2^{h+1} - 1$. Show that if the claim holds for h, then it must hold for $h + 1$.

Assume we start with a full binary tree of height h. The h^{TH} level of a binary tree can hold a maximum of 2^h nodes; to build a full binary tree of height $h + 1$, all we have to do is add two child nodes to each leaf node in the existing height h binary tree. Since there are 2^h leaf nodes in a height h binary tree, we need to add $2^h \times 2 = 2^{h+1}$ nodes to build a binary tree of height $h+1$ with the maximum number of nodes. By the induction hypothesis, there are $N(h) = 2^{h+1} - 1$ total nodes in the height h binary tree. Thus, in the height $h+1$ binary tree, the maximum number of nodes is given by:

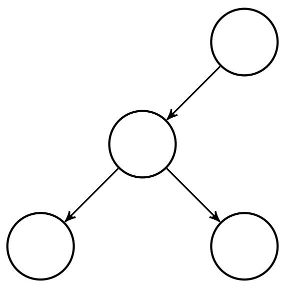
$$N(h+1) = N(h) + 2^{h+1} = \left(2^{h+1} - 1\right) + 2^{h+1} = 2 \times 2^{h+1} - 1 = 2^{h+2} - 1$$

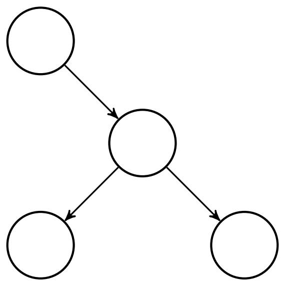
Thus, $N(h+1) = 2^{h+2} - 1$, and the induction step holds. Thus, we have proven by induction that the maximum number of nodes in a binary tree of height h is $N(h) = 2^{h+1} - 1$.

1C

In a binary search tree with no repeated keys, deleting the node with key x, followed by deleting the node with key y, will result in the same search tree as deleting the node with key y, then deleting the node with key x.

False. We discussed this problem in class today - Dr. Brower mentioned there is a counter-example that looked like one of the following:





I haven't been able to quite work out the counterexample.

1D

Inserting numbers 1 ... n into a binary min-heap in that order will take $O(n)$ time.

True. Inserting *any* sequence of n distinct integers into a binary min-heap is accomplished in $O(n)$ time; we use the array as is, and then heapify the tree bottom-to-top.

1E

The second smallest element in a binary min-heap with all elements with distinct values will always be a child of the root.

True. Let x_{min} be the smallest element in the binary min-heap (i.e., the root) and let x^* be the second smallest element in the binary min-heap. Thus $x_{min} < x^* < x$ for all other elements x in the heap. Assume x^* is not already a child of the root node. The binary min-heap requires that parent nodes are smaller in value than their child nodes: $x_p < x_{c_1}$, $x_p < x_{c_2}$. If x^* is not already a child of the root node, it must be somewhere else in the rest of the tree: assume x^* is a child node to node $x_{p'}$. By construction, $x_{p'} > x^*$; thus the min-heap property is violated. To restore the requirement, x^* must "bubble up" the tree, eventually settling as a child of the root node: $x_{min} < x^*$. Thus, the second smallest element in a binary min-heap with distinct values will always be a child of the root node.

Problem 2

This exercise is to learn binary search tree operations:

2A

Draw the sequence of binary search trees which results from inserting the following values in left-to-right order, assuming no balancing:

15, 10, 31, 25, 34, 56, 78, 12, 14, 13

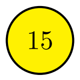


Figure 1: Insert 15 as Root

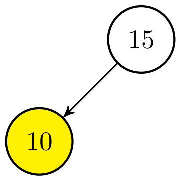


Figure 2: Inserting 10: 10 < 15, Insert Left.

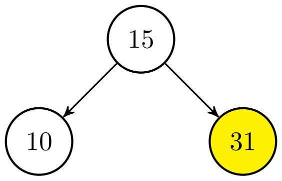


Figure 3: Inserting 31: 31 > 15, Insert Right.

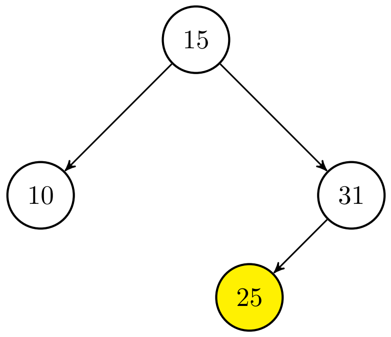


Figure 4: Inserting 25: 25 > 15, Move right. 25 < 31, Insert left.

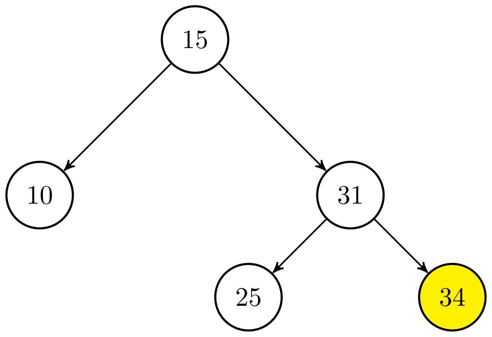


Figure 5: Inserting 34: 34 > 15, Move right. 34 > 31, Insert right.

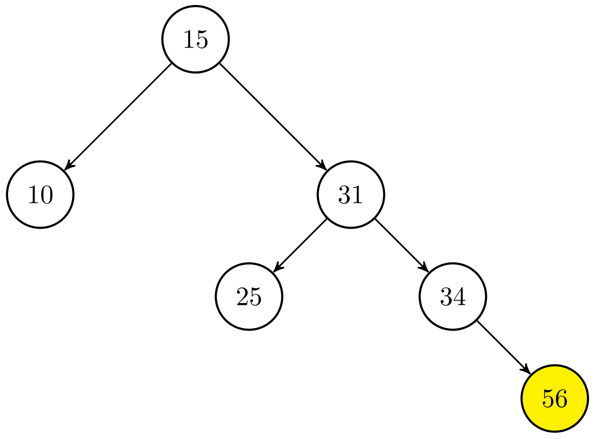


Figure 6: Inserting 56: 56 > 15, Move right. 56 > 31, Move right. 56 > 34, Insert right.

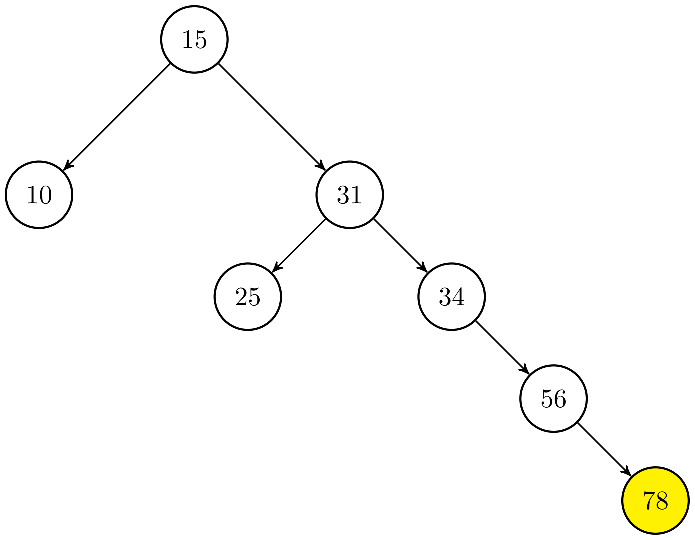


Figure 7: Inserting 78: 78 > 15, Move right. 78 > 31, Move right. 78 > 34, Move right. 78 > 56, Insert right.

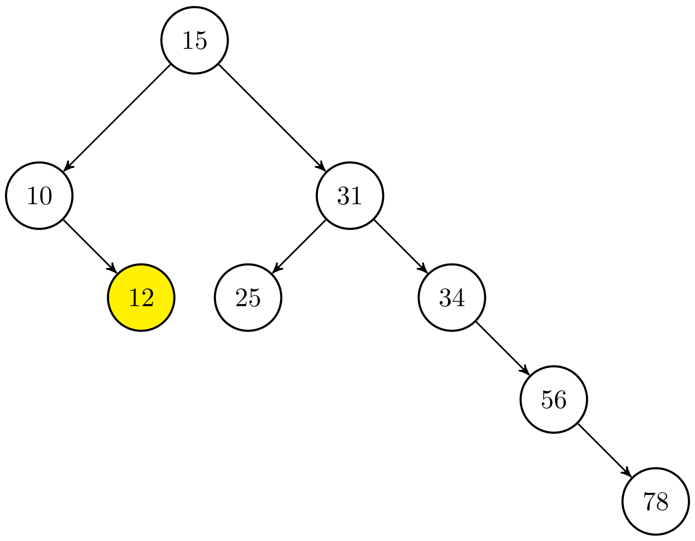


Figure 8: Inserting 12: 12 < 15, Move left. 12 > 10, Insert right.

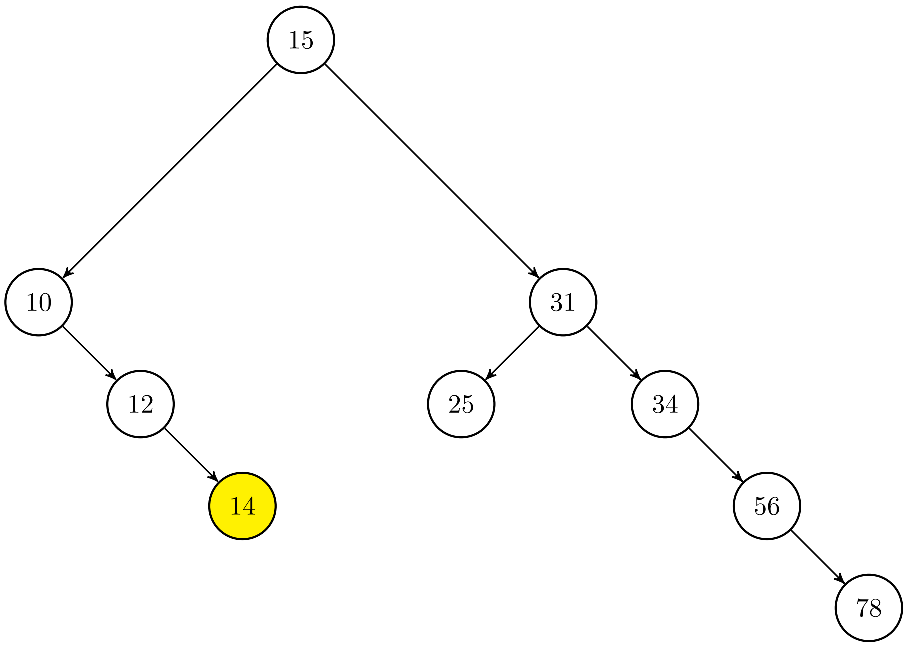


Figure 9: Inserting 14: 14 < 15, Move left. 14 > 10, Move right. 14 > 12, Insert right.

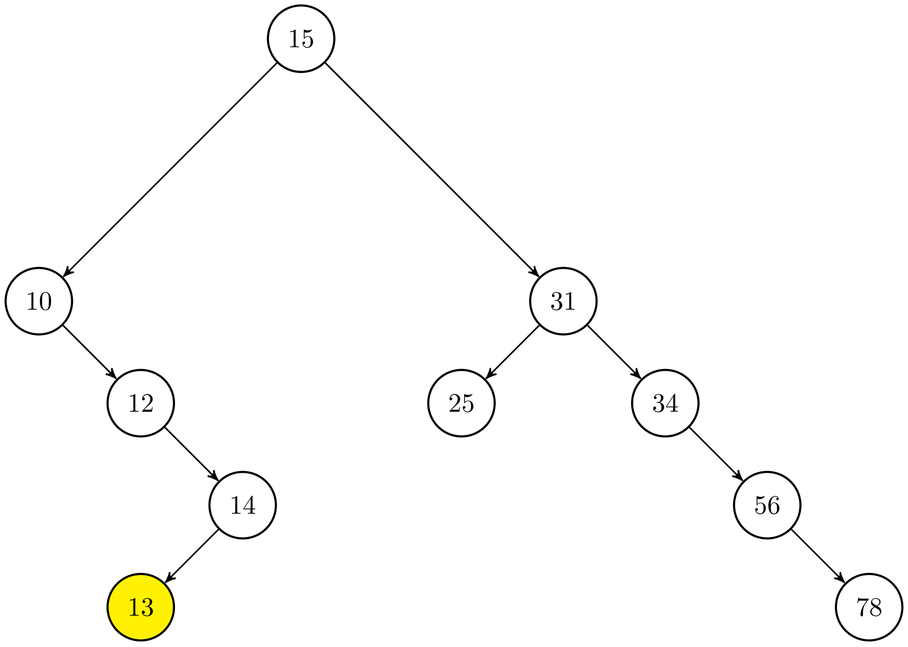


Figure 10: Inserting 13: 13 < 15, Move left. 13 > 10, Move right. 13 > 12, Move right. 13 < 14, Insert left.

2B

Starting from the tree at the end of the previous part, draw the sequence that results from deleting the following nodes in left-to-right order:

15, 31, 12, 14

We start with the complete tree:

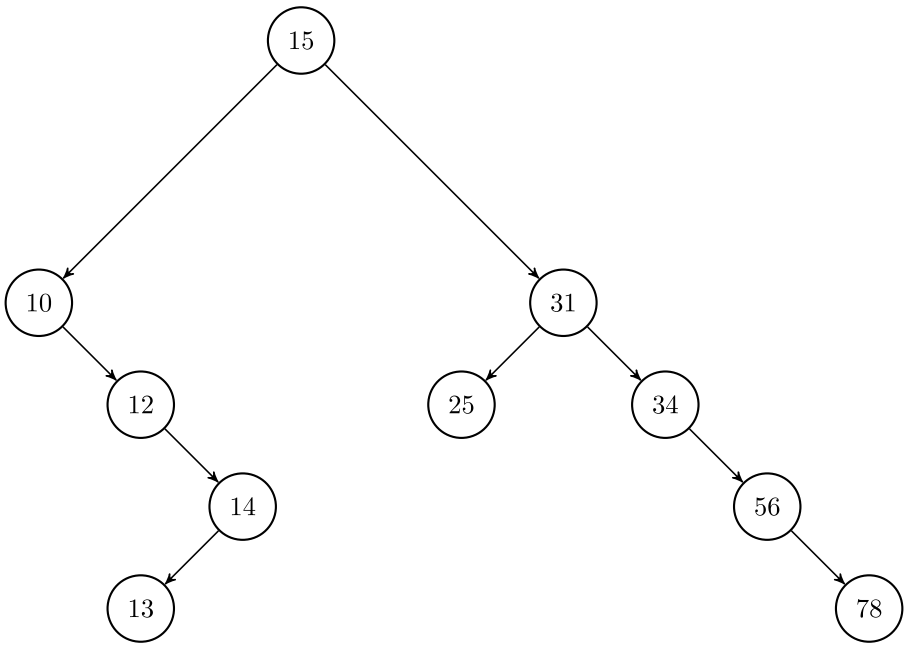


Figure 11: Full Tree

To delete 15, we note that it has two child nodes, and thus we must use the rotation scheme. Let $z = 15$, $q = NULL$, $\ell = 10$, $r = 31$, $y = 25$, and $x = NULL$. The rotation scheme proceeds as follows:

1. Move the sub-tree rooted at x to y.
2. Set y to be the parent of r.
3. Set q to be the parent of y. (Here, $q = NULL$ because we are deleting the root node - we must replace it with another node to maintain the tree.)
4. Set y to be the parent of ℓ.
5. Clear the pointers associated with node z - it is now deleted from the BST.

Following this procedure, we arrive at the tree:

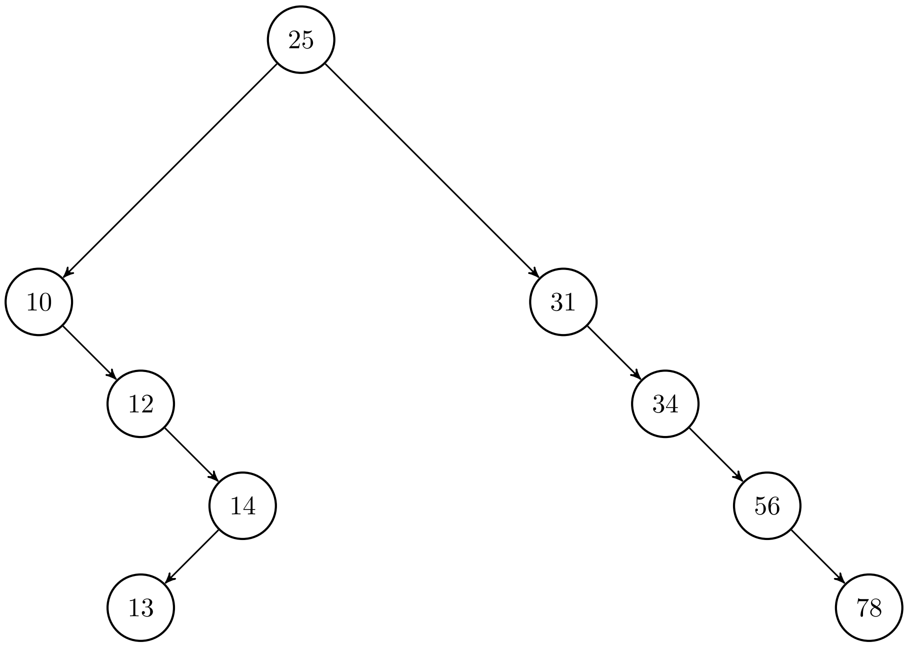


Figure 12: Delete 15: Set 25 (the successor to 15) to be the parent of the left and right subtrees rooted at 10 and 31 respectively.

To delete 31, we note that it now has a single child node; thus we just need to adjust the pointers to move the remaining nodes to their correct locations in the BST:

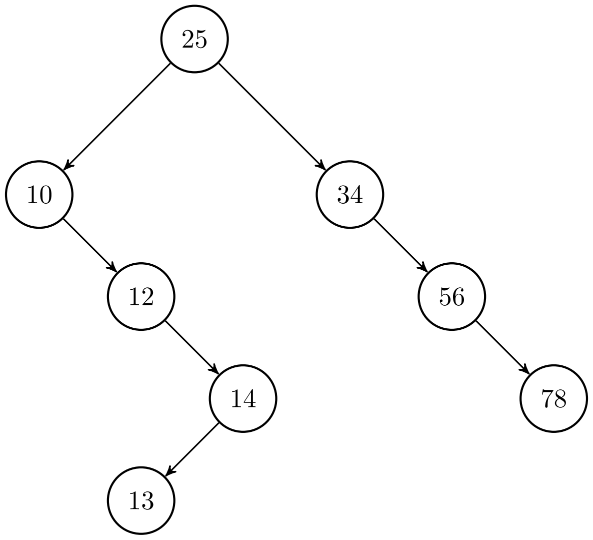


Figure 13: Delete 31: Since there is only one child node, we just need to adjust pointers to move the subtree rooted at 34 up a level to maintain the BST property.

To delete 12, we note that it has a single child node; thus we just need to repeat the same process as above: adjust the pointers to move the remaining nodes to their correct locations in the BST:

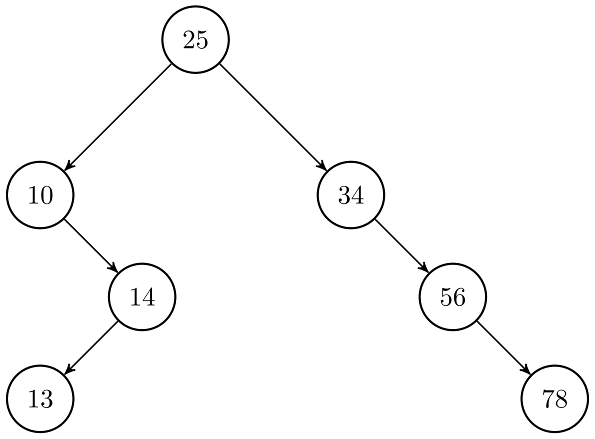


Figure 14: Delete 12: Since there is only one child node, we just need to adjust pointers to move the subtree rooted at 14 up a level to maintain the BST property.

To delete 14, we note that it has a single child node; thus we just need to repeat the same process as above: adjust the pointers to move the remaining nodes to their correct locations in the BST:

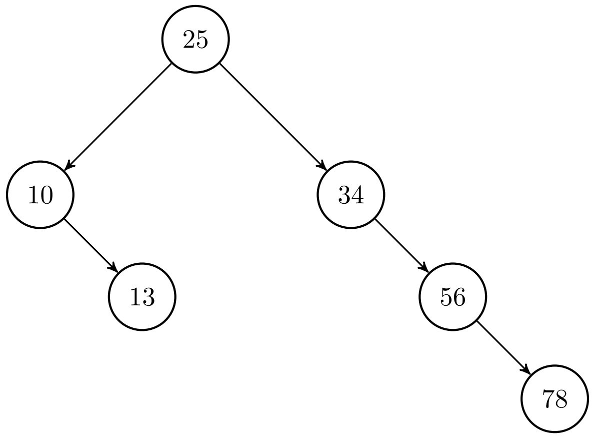


Figure 15: Delete 14: Since there is only one child node, we just need to adjust pointers to move the subtree rooted at 13 up a level to maintain the BST property.

2C

After deleting them, draw the sequence of reinserting left-to-right in reverse order:

14, 12, 31, 15

into the tree, and comment on the result.

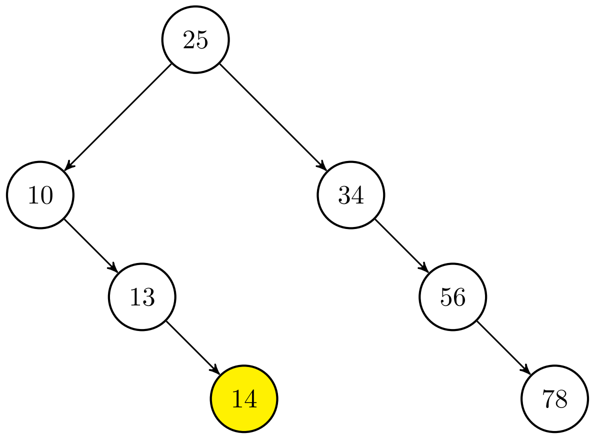


Figure 16: Inserting 14: 14 < 25, Move left. 14 > 10, Move right. 14 > 13, Insert right.

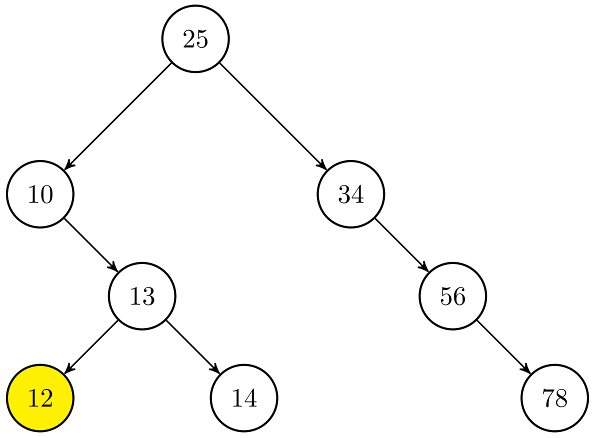


Figure 17: Inserting 12: 12 < 25, Move left. 12 > 10, Move right. 12 < 13, Insert left.

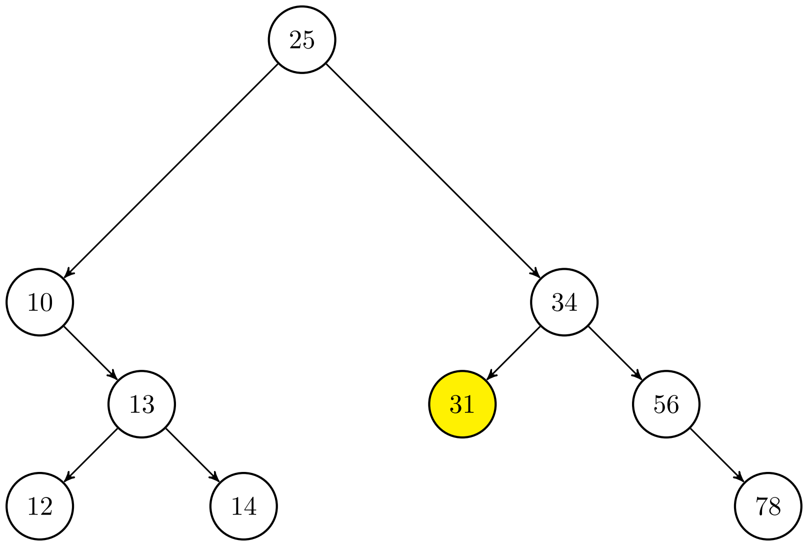


Figure 18: Inserting 31: 31 > 25, Move right. 31 < 24, Insert left.

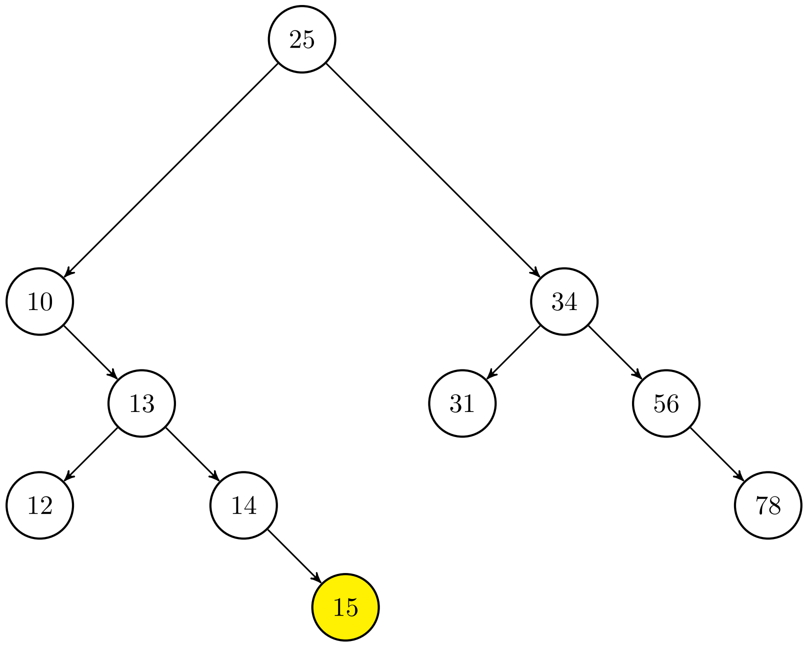


Figure 19: Inserting 15: 15 < 25, Move left. 15 > 10, Move right. 15 > 13, Move right. 15 > 14, Insert right.

The tree is clearly different than the original starting tree at the end of problem 2A. In general, it does not appear that deleting a set of nodes and inserting them back in reverse order recovers the original tree.

Problem 3

Problem 3A: Exercises 6.1-3, 6.1-4, 6.1-6, 6.3-3

Exercise 6.1-3

Show that in any sub-tree of a max-heap, the root of the sub-tree contains the largest value occurring anywhere in that sub-tree.

Let $T = \{a[i] : i \in [n]\}$ be an n-element max-heap with nodes $i = 1,\ 2,\ \dots\ n$. Let $P(i)$ be the parent to node i. A max-heap satisfies the following properties:

- The root node $a[1]$ contains the largest element in the heap. The root node has no parent node: $P(i) = NULL$.
- Consider non-root node $i,\ i \neq 1$. The **max-heap property** is described as follows: $a[P(i)] \geq a[i]$. The max-heap property holds for all non-root nodes.

Since T is a valid max-heap, the max-heap property will hold for all non-root nodes $i = 2,\ 3,\ \dots\ n$; namely, $a[P(i)] \geq a[i]\ \forall i \geq 2$. Let S_1 be set of all paths we could traverse starting at the root of the sub-tree T and stopping at a leaf node. Since the max-heap property guarantees that $a[P(i)] \geq a[i]$, if we traverse along nodes in path $s = \big(a[1]\ s_2\ s_3\ \dots s_\ell\big)$ of length ℓ, we construct the partial ordering:

$$a[1] \geq s_2 \geq s_3 \geq \dots \geq s_\ell$$

Thus, pick any node i^* as the root of subtree T_{i^*}. Let S_{i^*} be the set of all paths we could traverse starting at node i^* and stopping at a leaf node. Since we are not changing the number of nodes in tree T, every path $s' \in S_{i^*}$ must be a sub-path of some path $s \in S_1$. Since path $s \in S_1$ corresponds to the partial ordering:

$$a[1] \geq s_2 \geq s_3 \geq \dots \geq s_\ell$$

And path $s' = \big(a[i^*]\ s'_2\ s'_3\ \dots s_{\ell'}\big) \in S_{i^*}$ of length ℓ' is a subset of such a path, s' must also correspond to a partial ordering:

$$a[i^*] \geq s'_2 \geq s'_3 \geq \dots \geq s'_{\ell'}$$

Thus, the max-heap property is preserved even if we look at a sub-tree $T_{i^*} \subseteq T$; thus, the root i^* of any subtree T_{i^*} must contain the largest value in that subtree.

Exercise 6.1-4

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

The smallest element in a max-heap with all distinct elements must reside at a leaf node. Let x_{min} be the minimum element in the max-heap; by definition, $x_{min} < x_i$ for all elements x_i in the max-heap. Recall that max-heap property - for node all nodes $i \geq 2$ (i.e., all non-root nodes), $A[P(i)] > A[i]$; in other words, the parent every non-root node of i is larger than node i itself. Assume that x_{min} was located at non-leaf node j: $A[j] = x_{min}$. Let node k contain element $x_k : A[k] = x_k$. Let node j be the parent to node k. Since $P(k) = j$, For the max-heap property to hold, we require that $A[P(k)] > A[k] \iff A[j] > A[k]$. But we have assumed that x_{min} was located at node j, while some other distinct element $x_k > x_{min}$ was located at node k. Thus, the max-heap property is not satisfied, since $x_{min} \not> x_k$ by definition. Thus, x_{min} **cannot** be found on a non-leaf node in a max-heap; it must be found at a leaf node where it has no child nodes.

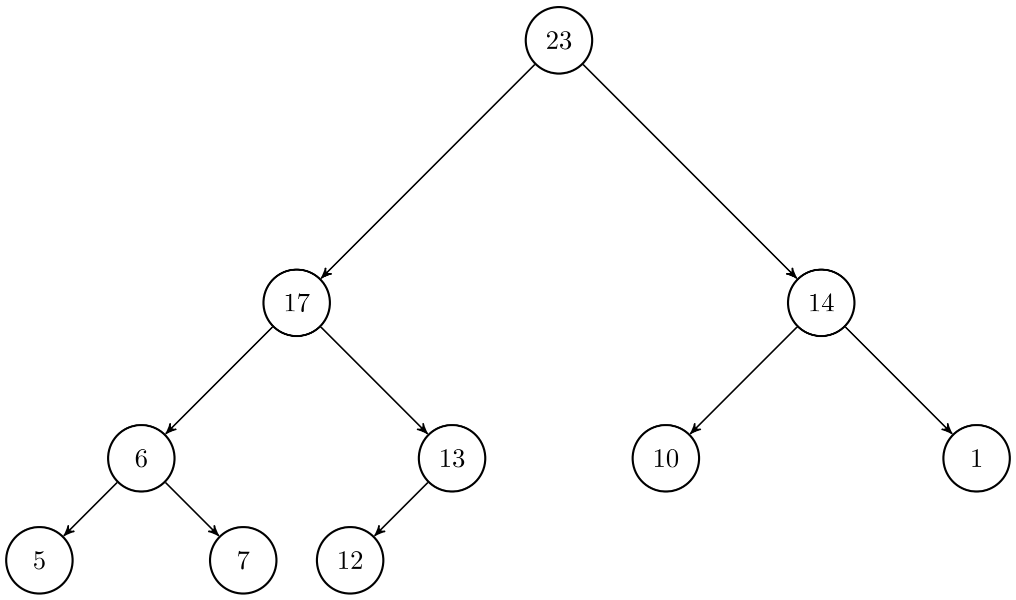
Exercise 6.1-6

Is the array with values:

$$23,\ 17,\ 14,\ 6,\ 13,\ 10,\ 1,\ 5,\ 7,\ 12$$

a max heap?

It is much easier to analyze the structure graphically:



Recall the max-heap property: for non-root node i with parent node $P(i)$, we require $a[P(i)] \geq a[i]$. We see that node 7 has parent 6; thus the max-heap property is violated. Thus this structure is **NOT** a max-heap.

Exercise 6.3-3

Show that there are at most:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

nodes of height h in any n-element heap.

First, assume that $n = 2^m - 1$ $m \in \mathbb{N}$, $m \geq 1$. If we construct a binary heap with these n elements, the resulting tree is a perfect binary tree. In a perfect binary tree with $2^m - 1$ elements, the number of nodes of height h is easy to calculate, since the sum of the height and depth of nodes in the tree are constant: $h + d = m - 1$. At depth d, there are $N(d) = 2^d$ nodes; using the height-depth invariant, the number of nodes of height h in a perfect binary tree with $2^m - 1$ elements is given as:

$$N_{2^m-1}(h) = 2^d = 2^{m-h-1} = \frac{2^m}{2^{h+1}} = \left\lceil \frac{2^m - 1}{2^{h+1}} \right\rceil$$

$$\therefore N_{2^m-1}(h) = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Now, let's assume that $n = 2^x < 2^m - 1,\ m \in \mathbb{N},\ m \geq 1$, such that $\lceil x \rceil = m$; that is, 2^m is the first power of 2 greater than n. Since we are constructing a heap, we completely fill a level with elements before inserting elements in the next level; thus, since $n < 2^m - 1$, there will be several missing nodes in the last level of the perfect binary tree, as illustrated in the example below:

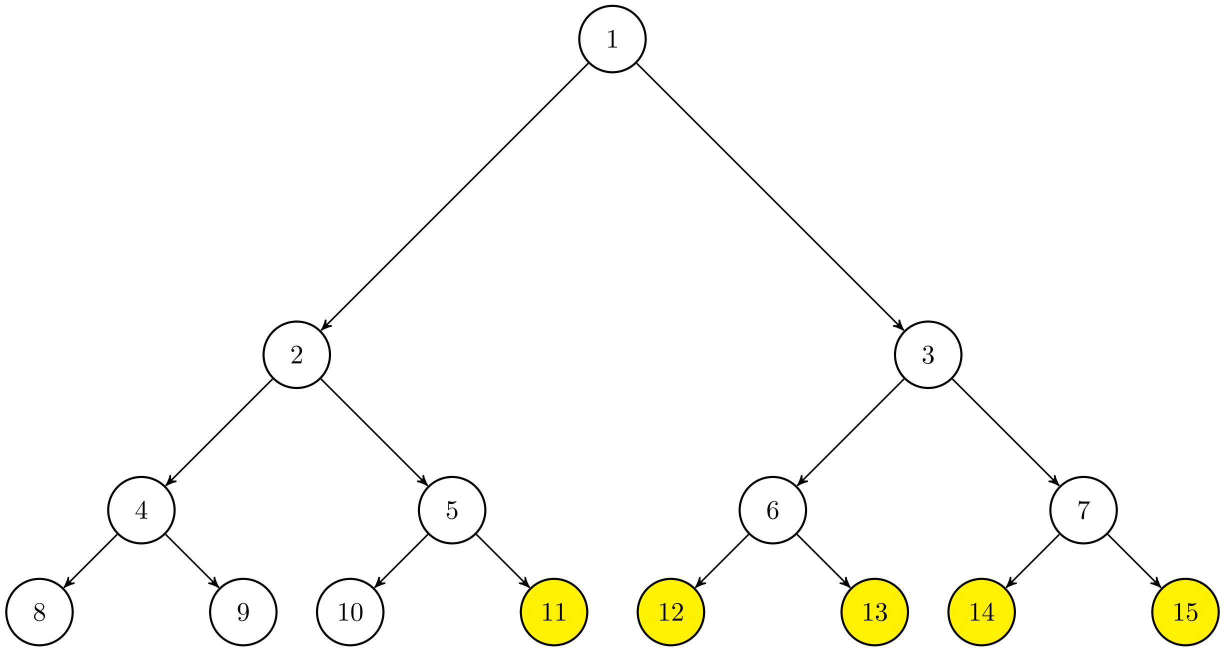


Figure 20: Binary heap with $n = 10$ elements (Nodes 1 - 10) compared to perfect binary heap with $n = 2^4 - 1 = 15$ elements (Nodes 1 - 15). Nodes 11 - 15 are found in the perfect binary tree, but not in the heap with 10 elements.

To recover the n-element heap from the $2^m - 1$ element perfect tree, we just need to delete extra leaf nodes in the last row of the perfect tree. Let z_0 be the number of deleted nodes of height 0. By definition, the height h of a node is the longest path to a leaf node. If we delete z_0 leaf nodes, each of height 0, then $\lfloor z_0/2 \rfloor$ parent nodes, previously of height 1, become new leaf nodes of height 0. Thus, the number of nodes of height $h = 0$ in the n-element heap is given as:

$$N_n(0) = N_{2^m-1}(0) - z_0 + \left\lfloor \frac{z_0}{2} \right\rfloor$$

$$\therefore N_n(0) \leq N_{2^m-1}(0)$$

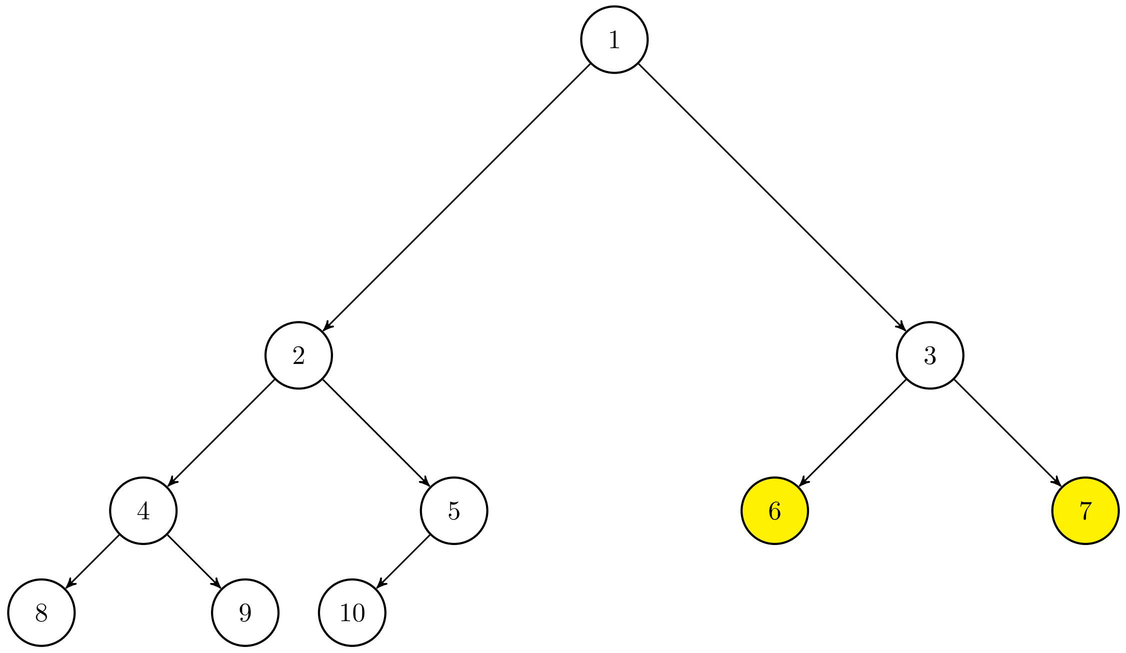


Figure 21: Binary heap with $n = 10$ elements (Nodes 1 - 10) after deleting the extra nodes. Nodes 6 and 7, which were previously parent nodes of height $h = 1$, are now leaf nodes of height $h = 0$.

But this change propagates up the tree! In the perfect tree, there were originally $N_{2^m-1}(1)$ nodes of height 1. Let z_1 be the number of nodes, originally of height 1, but are now height 0. After updating the height of z_1 nodes, $\lfloor z_1/2 \rfloor$ parent nodes, originally of height 2, are now one level closer to the leaves, and thus become nodes of height 1. Thus, the number of nodes of height $h = 1$ in the n-element heap is given as:

$$N_n(1) = N_{2^m-1}(1) - z_1 + \left\lfloor \frac{z_1}{2} \right\rfloor$$

$$\therefore N_n(1) \leq N_{2^m-1}(1)$$

Without loss of generality, we can repeat this argument up the entire tree. Since we have already computed $N_{2^m-1}(h)$:

$$N_{2^m-1}(h) = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

And we can upper bound $N_n(h)$ with $N_{2^m-1}(h)$ for each height h, we have that:

$$N_n(h) \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil \quad \square$$

Problem 3B: Exercises 12.3-2, 12.3-3, 12.3-4, B.5-4

Exercise 12.3-2

Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of of nodes examined when the value was first inserted into the tree.

Let N_x be the number of nodes examined when inserting node x into the binary search tree. Let S_x be the set of nodes visited when inserting node x into the binary search tree. Every time a node is visited, add it to S_x. To insert node x into the binary search tree, we start at the root node. Append the root node R to S_x, and perform the comparison to determine which side of the subtree to place x. Move to the next node accordingly; say we move to node X_2. Then, we append X_2 to the node list S_x, and perform the comparison to determine which side of the subtree rooted at node X_2 to place x. This process until we finally place x - we will have examined nodes $S_x = \{R,\ X_2,\ X_3\ \dots X_m\}$, where $|S_x| = N_x$, to determine the proper location of x in the binary search tree. Some time later, we wish to search for node x in the binary search tree. Let S'_x be the set of nodes visited when searching for x in the binary search tree. We start the search at the root node. Append the root R to S'_x, and perform the comparison to determine which side of the subtree x is located. Move to the next node accordingly; we will move to node X_2, the **same** node we moved to when looking for the location to place x! We perform the comparison again, moving to node X_3. At node X_3, we append it to S'_X and continue. We will traverse the same nodes: $R,\ X_2,\ X_3,\ \dots X_m$. At X_m, we perform the comparison, which moves us to node X^*, which contains x. Once at X^*, we must perform the last comparison to confirm we have found x; thus, we must evaluate $N'_x = |S'_x| = |S_x \cup X^*| = N_x + 1$. Thus, when searching for x, we must examine $N_x + 1$ nodes.

Exercise 12.3-3

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

Let $P(n)$ be the time required to print out the contents of any n-element binary search tree. By using the recursive definition of the in-order-print function, we have that $P(n) = \Theta(n)$ for any n-element binary search tree. The most significant portion of the proposed sorting algorithm is the time required to build the binary search tree by inserting the n elements into the tree one-by-one. In the many tree-based structures we have studied so far, we have seen that insertion operations require time that scales with the height of the tree. In the best-case insertion, the n elements that are inserted into the binary search tree form a perfect binary tree with height $\Theta(\lg n)$. Inserting a new element into an existing tree of height $\Theta(\lg n)$ requires traversal down a path of length $\Theta(\lg n)$. Thus, building a binary search tree by inserting n elements one-by-one takes $I_B(n) = \Theta(n \lg n)$ time in the best-case insertion. In the worse-case insertion, then n elements that are inserted into the binary search tree form a linked list with height $\Theta(n)$ - this occurs if the n elements are already sorted. Inserting a new element into an existing linked list of height $\Theta(n)$ requires traversal down a path of length $\Theta(n)$. Thus, building a binary search tree by inserting n elements one-by-one takes $I_W(n) = \Theta(n^2)$ time in the worst-case insertion. Thus, the total time for the sorting algorithm is:

$$T(n) = \begin{cases} I_B(n) + P(n) = \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n), & Best \\ I_W(n) + P(n) = \Theta(n^2) + \Theta(n) = \Theta(n^2), & Worst \end{cases}$$

Exercise 12.3-4

Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y then x? Argue why it is or give a counterexample.

See problem 1C, this is the same question.

Exercise B.5-4

Use induction to show that a nonempty binary tree with n nodes has height at least $\lfloor \lg n \rfloor$.

Base case: Let $n = 1$. Since the binary tree consists of just 1 node, it must have height 0; thus $H(1) = 0$. Using the equation $H(n) \geq \lfloor \lg n \rfloor$:

$$H(1) \geq \lfloor \lg 1 \rfloor \implies H(1) \geq 0$$

Thus, the base case holds and we can proceed with the rest of the induction.
Induction step: Assume $H(n) \geq \lfloor \lg n \rfloor$; show that if this holds, then $H(n+1) \geq \lfloor \lg n + 1 \rfloor$.

Recall the definition of the height H of a binary tree: the height H is the length of the **longest** path from the root node to a leaf node. Thus, there are two extreme ways of organizing a binary tree with n nodes - a linked list of height $n-1$, and a complete binary tree of height $\approx \lg n$ (We will be more precise with the scaling later). Thus, we can bound the height $H(n)$ of any tree with n nodes as:

$$\lg n \lessapprox H(n) \leq n - 1$$

Let $2^{m-1} \leq n \leq 2^m - 2$. In other words, $n = 2^x,\ x \in [m-1,\ m)$ By the induction hypothesis:

$$H(n) \geq \lfloor \lg n \rfloor = \lfloor \lg 2^x \rfloor = \lfloor x \rfloor = m - 1$$

Thus, $H(n) \geq m - 1$. To get $n+1$ elements in the tree, we add one more node. Assume we can place it such that $H(n+1) = H(n)$. Thus is possible since we cannot form a perfect tree with $n \in [2^{m-1},\ 2^m - 2]$ elements; thus, we can always form a tree where adding an additional node does not change the height of the tree. Thus, $H(n+1) \geq m - 1$. Since $n = 2^x$, we can write $n + 1 = 2^y$ for some $y > x,\ y \in [2^{m-1},\ 2^m - 2]$. Thus:

$$\lfloor \lg n + 1 \rfloor = \lfloor \lg 2^y \rfloor = \lfloor y \rfloor = m - 1$$

.

Since $H(n+1) \geq m - 1$, we can write:

$$H(n+1) \geq m - 1 = \lfloor \lg n + 1 \rfloor \implies H(n+1) \geq \lfloor \lg n + 1 \rfloor$$

Now assume $n = 2^m - 1$; in other words, we can form a perfect tree with n elements; the minimum height of an n element tree is the same: $H(n) \geq m - 1$. Now, when we add one more element, the height of the perfect binary tree must increase; thus $H(n+1) \geq H(n) + 1 = (m-1) + 1 \implies H(n+1) geqm$. But:

$$n + 1 = (2^m - 1) + 1 = 2^m \implies \lfloor \lg n + 1 \rfloor = \lfloor \lg 2^m \rfloor = m$$

Therefore, $H(n+1) \geq \lfloor \lg n + 1 \rfloor$. Since $H(n+1) \leq \lfloor \lg n + 1 \rfloor$ in both scenarios, the induction step holds. Thus, by induction, we have shown that the minimum height of a binary tree with n nodes is $H(n) \geq \lfloor \lg n \rfloor$.

Problem 4

Determine whether the following statements are true or false, and briefly explain why:

4A

A heapsort algorithm for a given list first forms a max-heap with the elements in that list, then extracts the elements of the heap one by one from the top. This algorithm will sort a list of n elements in time of $O(\log n)$.

False. Heapsort has a run-time of $O(n \log n)$ - each element must roughly travel a path of length $O(\log n)$ for the max heap to be maintained after each iteration. Since there are n elements to sort, heapsort takes $O(n \log n)$ time to sort n elements.

4B

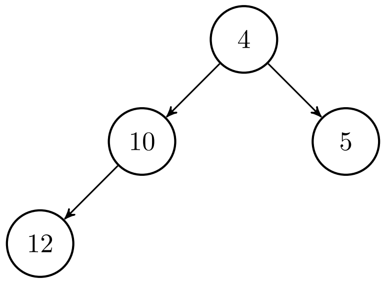
A connected, undirected, acyclic graph with N nodes and $N-1$ edges is a tree.

True. A tree is defined as a connected, undirected, acylic graph; if there are N nodes in the tree and the entire structure is connected, then there must be $N-1$ edges.

4C

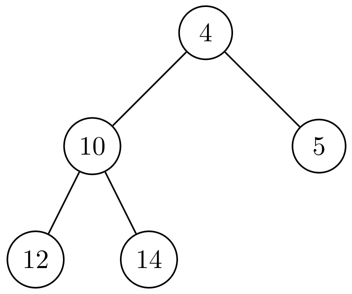
A binary heap of n elements is a full binary tree for all possible values of n.

False. A full binary tree is one where each node has either 0 or 2 child nodes. A binary heap with n elements, while always complete (meaning it is filled consecutively), does not have to be full; it is possible for a node to have just one child node, as shown in the counterexample below:



4D

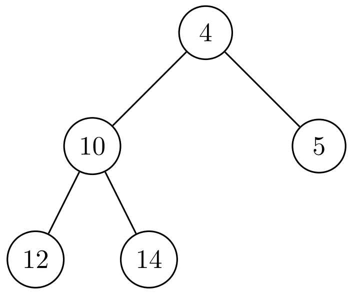
The following tree is a valid min-heap:



True. A min-heap satisfies the min-heap property: $A[P(i)] \leq A[i]\ \forall i \geq 2$. In other words, the parent of non-root node i must be smaller than node i. Each node in the above tree satisfies the min-heap property, so the above tree is a valid min-heap.

4E

The following tree can appear in a binomial min-heap:



False. A binomial heap order k contains binary trees that have depths following the binomial coefficients found in Pascal's triangle; thus, the depths should take the form $1,\ 2,\ 1$. However, in the given heap, we see the depths take the form $1,\ 2,\ 2$, which are not the binomial coefficients. Thus, this tree cannot appear in a binomail min-heap.

4F

Given an array of positive integers $a[i]$, maximizing $\sum i \cdot a[i]$ over permutations of the array requires sorting $a[i]$ in assigning order.

True. Sorting is just a special case of the optimization problem:

$$\max \sum_i i \cdot a[i]$$

Thus, if we want to find the above maximum, all we really need to do is just sort the array - we will find the maximum sum, no special optimization techniques required.

Problem 5

Given the following list of elements:

35, 22, 11, 98, 55, 66, 77, 80, 85, 90, 95

5A

Draw the sequence of binary min-heaps which results from inserting the following values in the order in which they appear into an empty heap.

The min-heap satisfies the **min-heap property**: $A[P(i)] \leq i \; \forall i \geq 2$. To build the min-heap, we first start by throwing every element into the binary heap, ignoring the min-heap property:

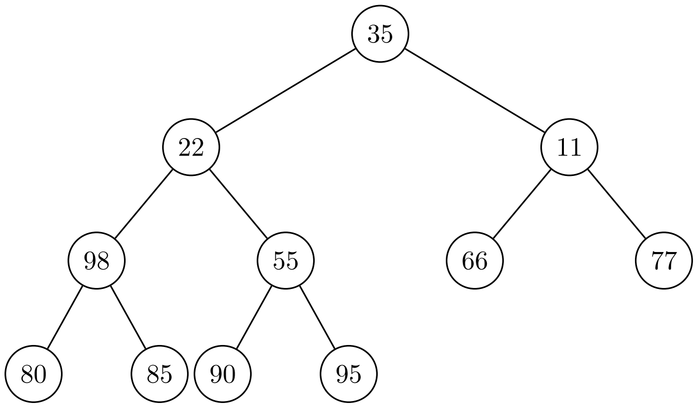


Figure 22: Starting min-heap

Then we run the heapify algorithm, "bubbling" low-value nodes to the top of the heap by swapping node i with its parent node $P(i)$ if $A[P(i)] > A[i]$. The incremental binary heaps are shown in the figures below:

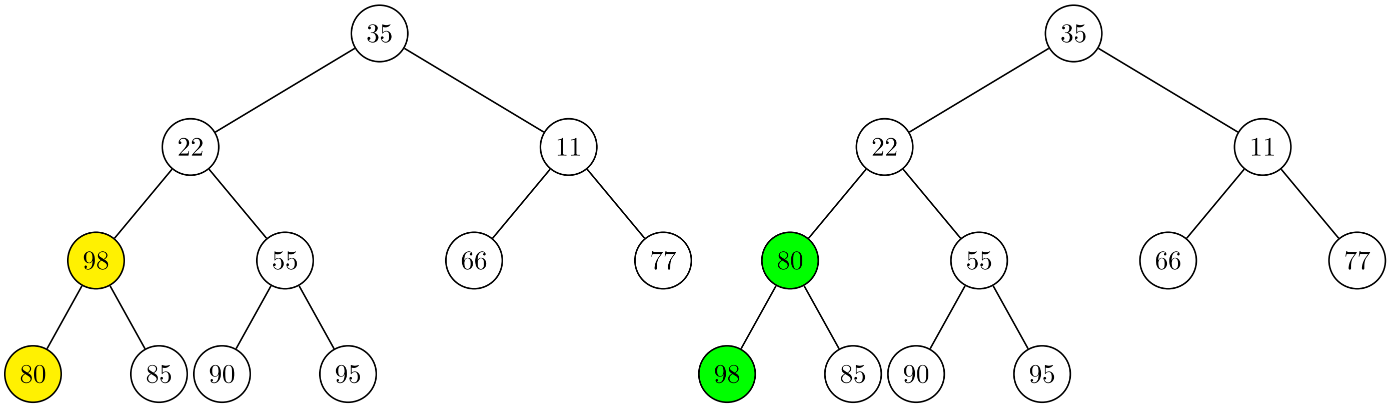


Figure 23: Node 77 - no child nodes, pass. Node 66 - no child nodes, pass. Node 55: 55 < 90 and 55 < 95, min-heap property holds, no swaps necessary. Node 98: 98 > 80 and 98 > 85, swap node 98 with node 80, the smaller of the two child nodes.

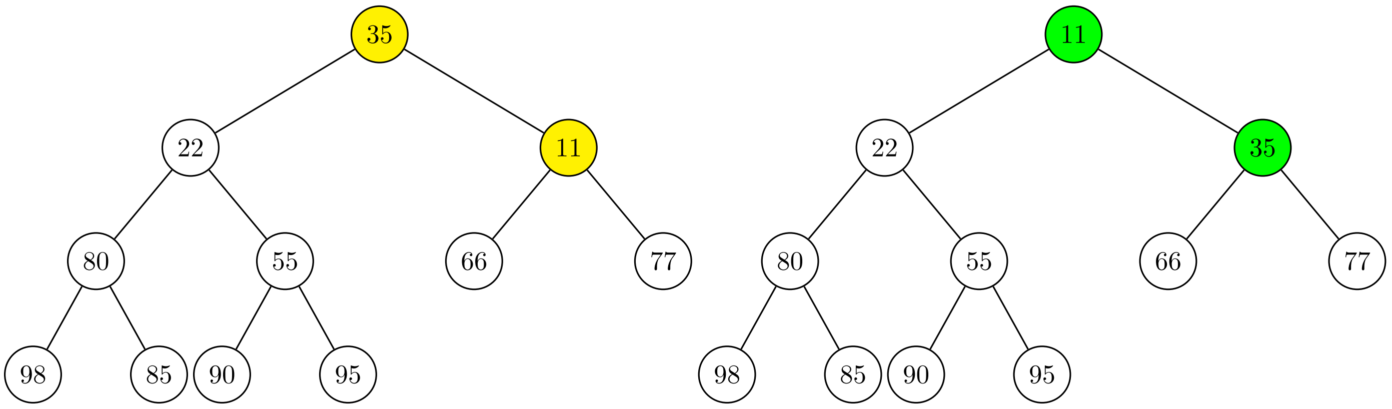


Figure 24: Node 11: 11 < 66 and 11 < 77, min-heap property holds, no swaps necessary. Node 22: 22 < 80 and 20 < 55, min-heap property holds, no swaps necessary. Node 35: 35 > 22 and 35 > 11, swap node 35 with node 11, the smaller of the two child nodes.

The final min heap is shown below:

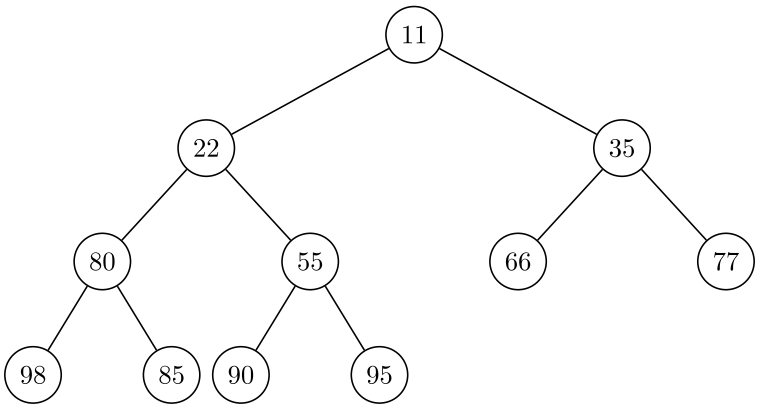


Figure 25: Final min-heap - the min-heap property holds for all nodes.

5B

Compare this answer with inserting them into the BST tree.

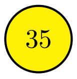


Figure 26: Insert 35 as Root

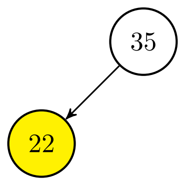


Figure 27: Inserting 22: 22 < 35, Insert left.

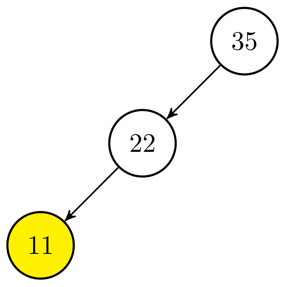


Figure 28: Inserting 11: 11 < 35, Move left. 11 < 22, Insert left.

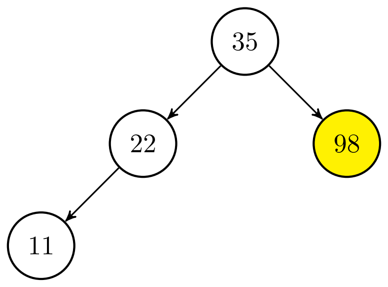


Figure 29: Inserting 98: 98 > 35, Insert right.

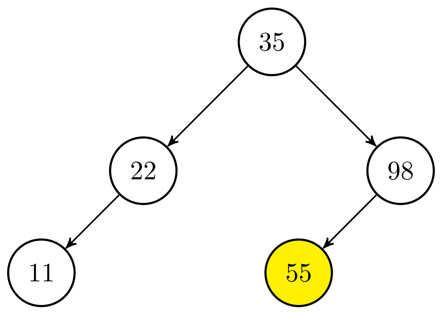


Figure 30: Inserting 55: 55 > 35, Move right. 55 < 98, Insert left.

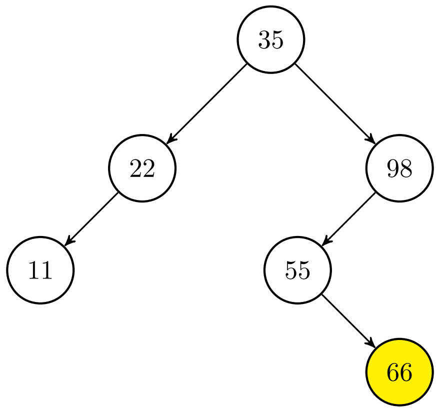


Figure 31: Inserting 66: 66 > 35, Move right. 66 < 98, Move left. 66 > 55, Insert right.

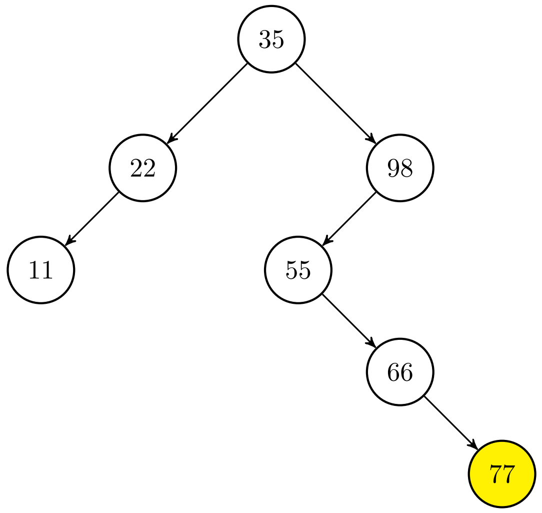


Figure 32: Inserting 77: 77 > 35, Move right. 77 < 98, Move left. 77 > 55, Move right. 77 > 66, Insert right.

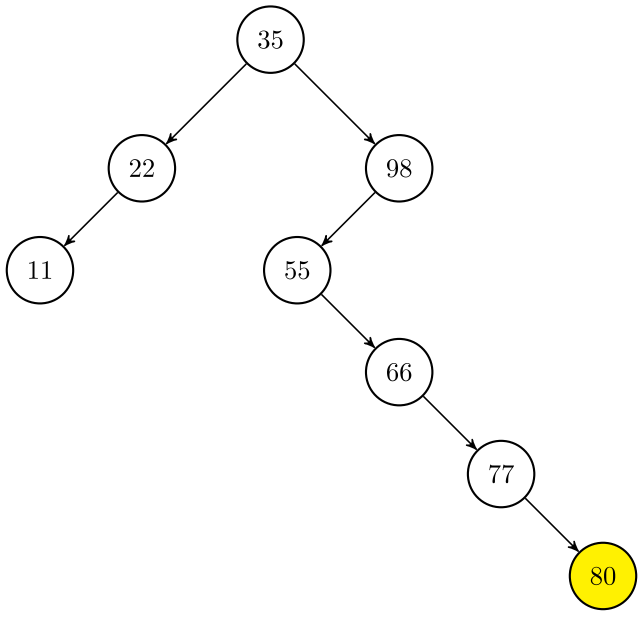


Figure 33: Inserting 80: 80 > 35, Move right. 80 < 98, Move left. 80 > 55, Move right. 80 > 66, Move right. 80 > 77, Insert right.

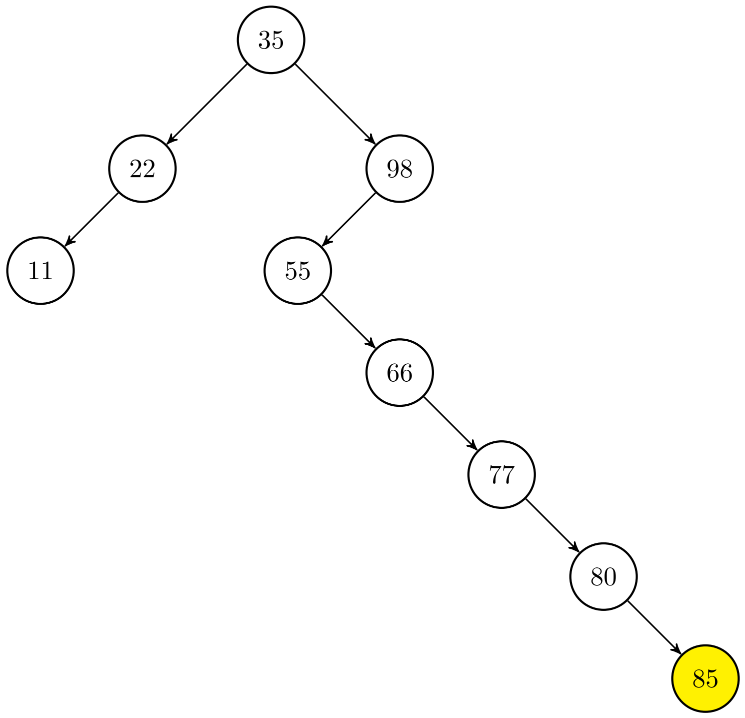


Figure 34: Inserting 85: 85 > 35, Move right. 85 < 98, Move left. 85 > 55, Move right. 85 > 66, Move right. 85 > 77, Move right. 85 > 80, Insert right.

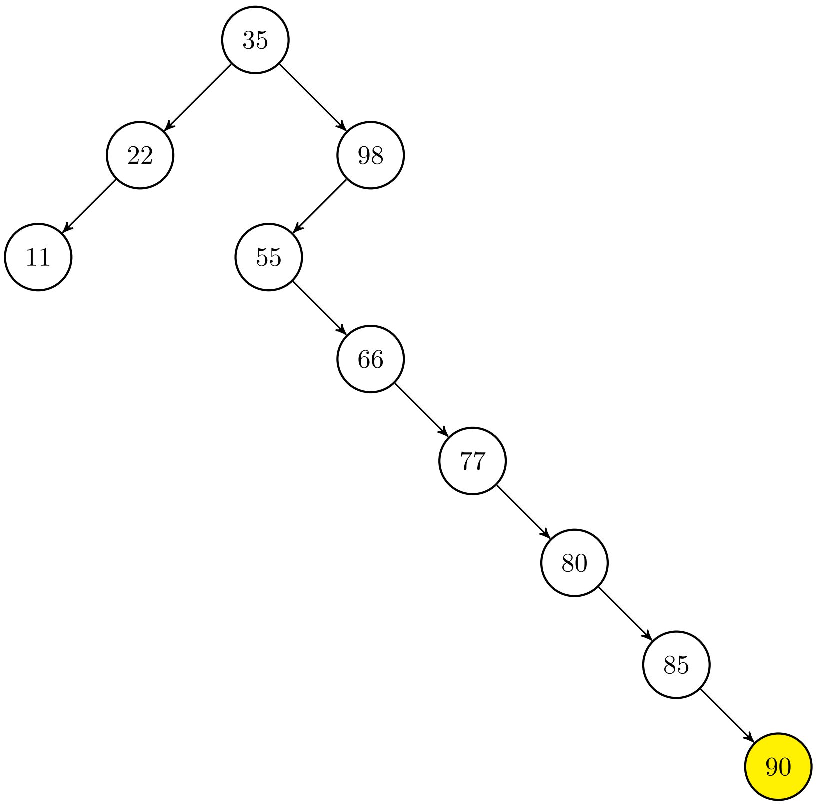


Figure 35: Inserting 90: 90 > 35, Move right. 90 < 98, Move left. 90 > 55, Move right. 90 > 66, Move right. 90 > 77, Move right. 90 > 80, Move right. 90 > 85, Insert right.

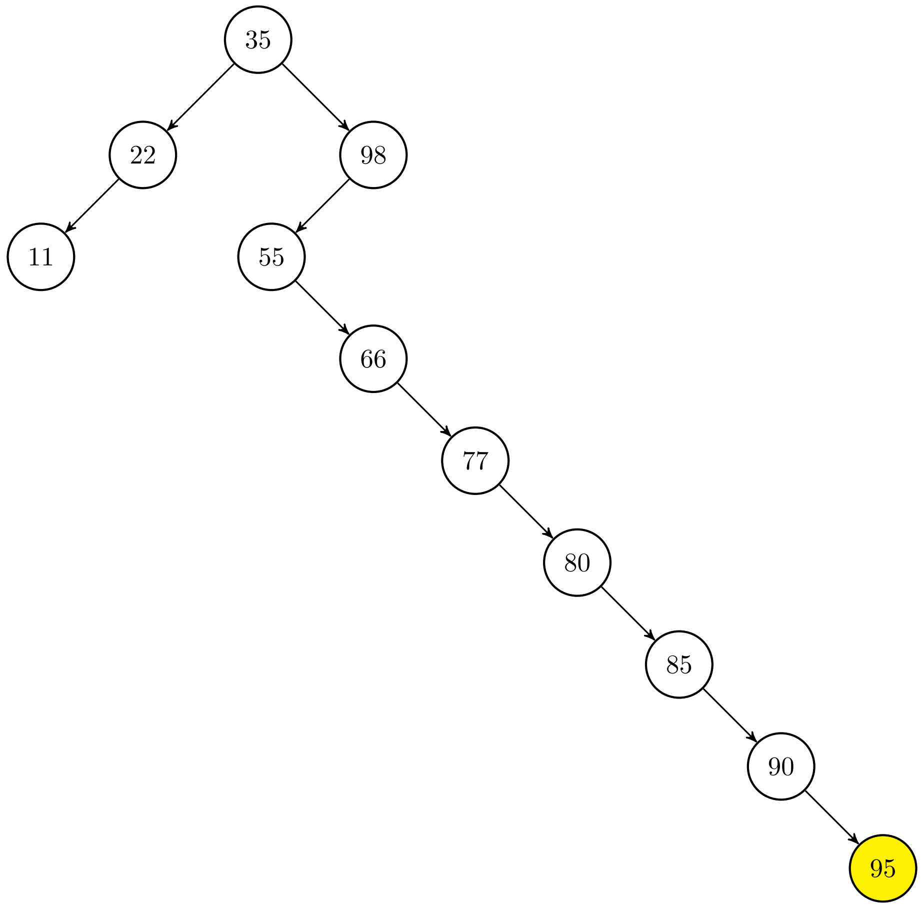


Figure 36: Inserting 95: 95 > 35, Move right. 95 < 98, Move left. 95 > 55, Move right. 95 > 66, Move right. 95 > 77, Move right. 95 > 80, Move right. 95 > 85, Move right. 95 > 90, Insert right.

A binary search tree is a relatively poor data structure for this particular set of data given that most of the elements already appear in sorted order, resulting in a highly unbalanced tree.

5C

Compare this answer with inserting them into the AVL tree.

We can improve the balance in the structure by using the AVL tree which imposes the following property on each node in the tree. Let $T_L(i)$ and $T_R(i)$ be the left subtree rooted at node i and the right subtree rooted at node i respectively. Let $H_L(i)$ and $H_R(i)$ be the height of left subtree $T_L(i)$ and right subtree $T_R(i)$ respectively. The AVL property requires:

$$|H_L(i) - H_R(i)| \leq 1 \ \forall i$$

Maintaining this property requires careful balancing via *single rotations* and *double rotations*, as illustrated by the four cases below:

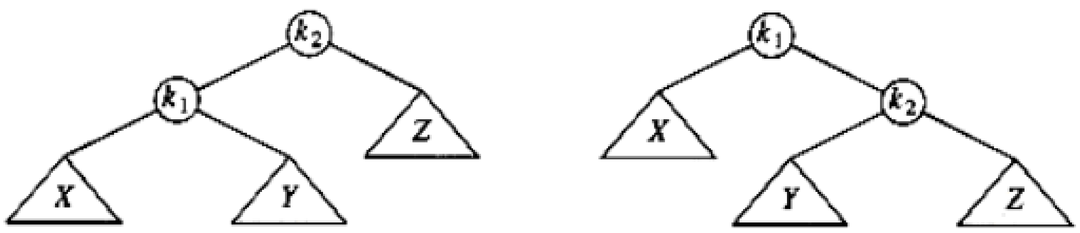


Figure 4.31 Single rotation

Figure 37: AVL Single Rotation

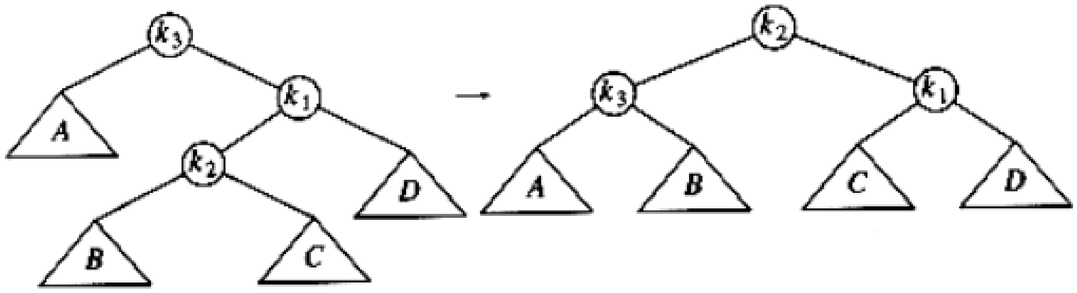


Figure 4.33 (Right-left) double rotation

Figure 38: AVL RL Double Rotation

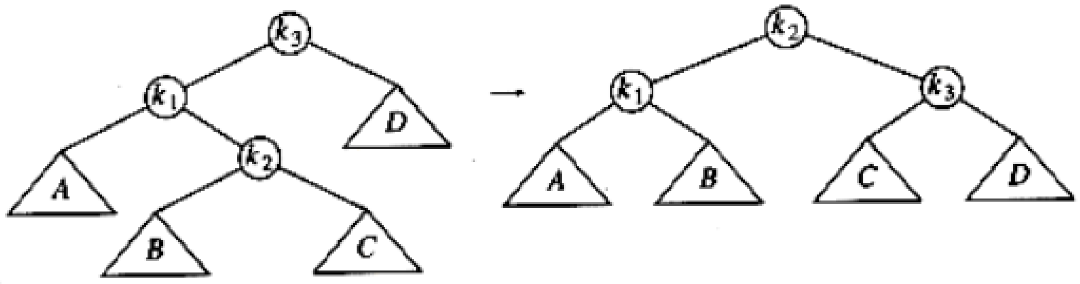


Figure 4.34 (Left-right) double rotation

Figure 39: AVL LR Double Rotation

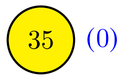


Figure 40: Insert 35 as Root. There are no child nodes, so the AVL property is maintained.

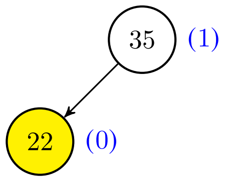


Figure 41: Insert 22: 22 < 35, Insert left. The AVL property is maintained for each node in the tree, so no need to rebalance.

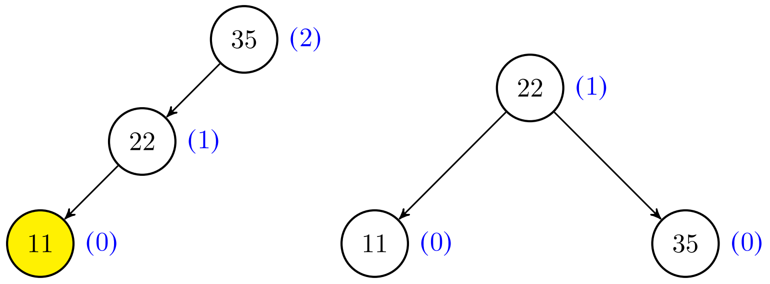


Figure 42: Insert 11: 11 < 35, Move left. 11 < 22, Insert left. The AVL property is violated for node 35; this format fits the case of a single rotation with $k_1 = 22$, $k_2 = 35$, $X = 11$, $Y = NULL$, and $Z = NULL$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.

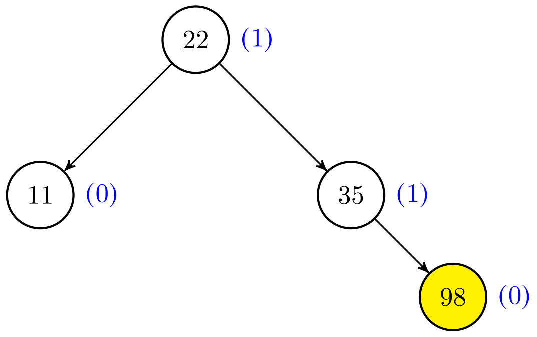


Figure 43: Insert 98: 98 > 22, Move right. 98 > 35, Insert right. The AVL property is maintained for each node in the tree, so no need to rebalance.

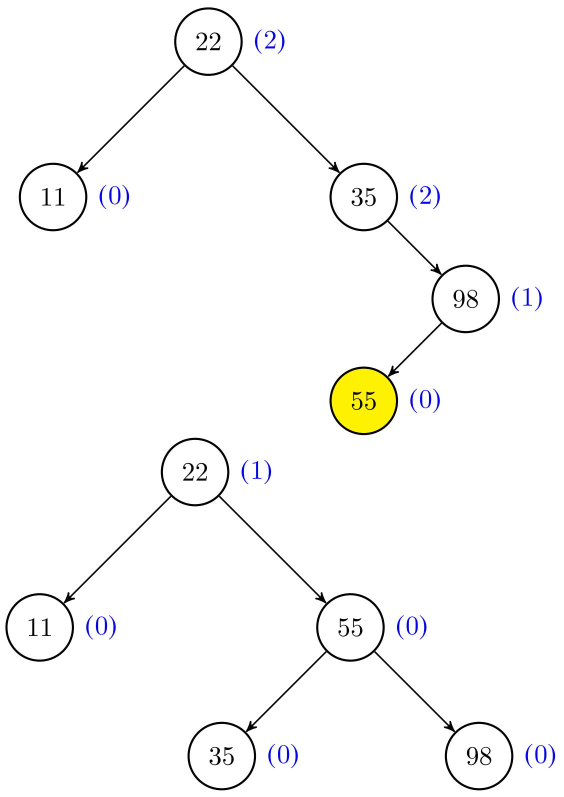
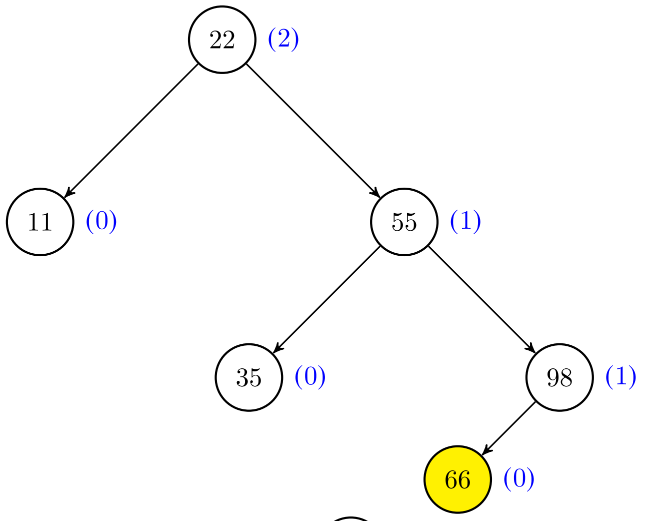


Figure 44: Insert 55: 55 > 22, Move right. 55 > 35, Move right. 55 < 98, Insert left. The AVL property is violated for node 35; this format fits the case of a double rotation with $k_1 = 98$, $k_2 = 55$, $k_3 = 35$, $A = NULL$, $B = NULL$, $C = NULL$, and $D = NULL$. After performing the double rotation, we obtain the tree on the right, which maintains the AVL property.



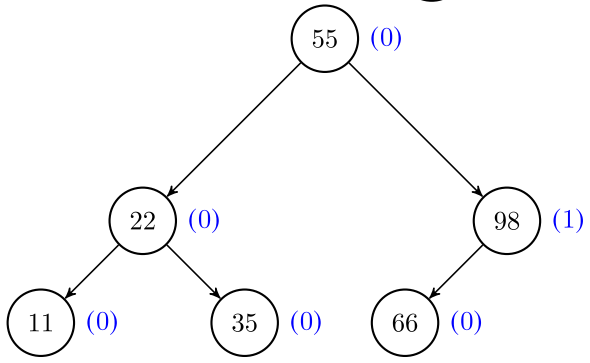
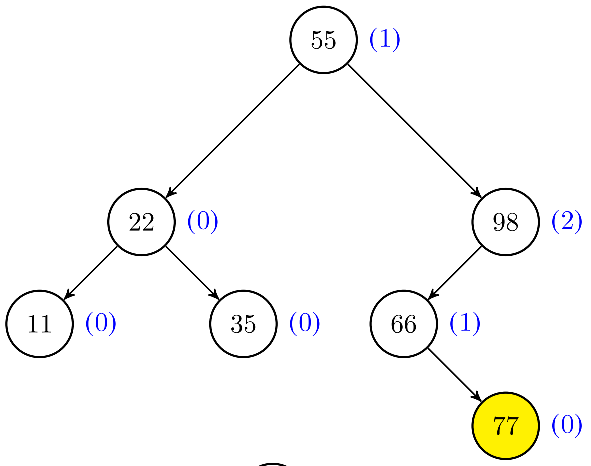


Figure 45: Insert 66: 66 > 22, Move right. 66 > 55, Move right. 66 < 98, Insert left. The AVL property is violated for node 22; this format fits the case of a single rotation with $k_1 = 22$, $k_2 = 55$, $X = 11$, $Y = 35$, and $Z = 98$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.



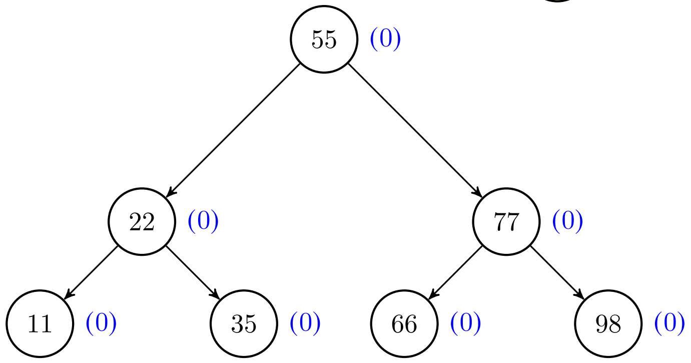


Figure 46: Insert 77: 77 > 66, Move right. 77 < 98, Move left. 77 > 66, Insert right. The AVL property is violated for node 98; this format fits the case of a double rotation with $k_1 = 66$, $k_2 = 77$, $k_3 = 98$, $A = NULL$, $B = NULL$, $C = NULL$ and $D = NULL$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.

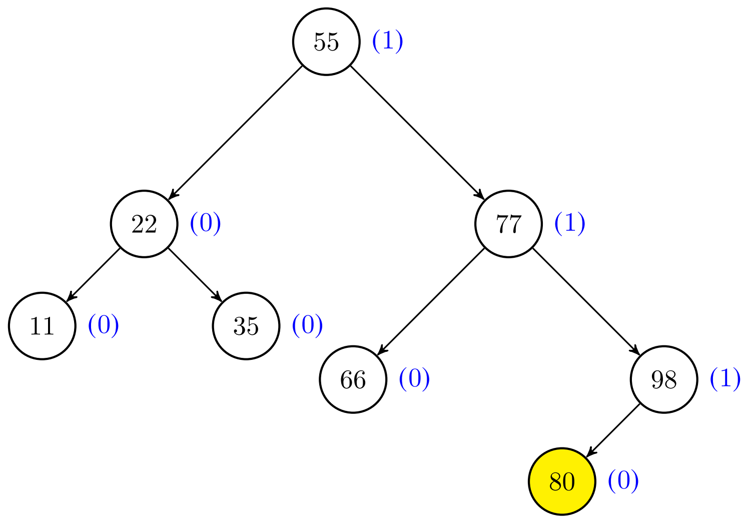


Figure 47: Insert 80: 80 > 55, Move right. 80 > 77, Move right. 80 < 98, Insert left. The AVL property is maintained for each node in the tree, so no need to rebalance.

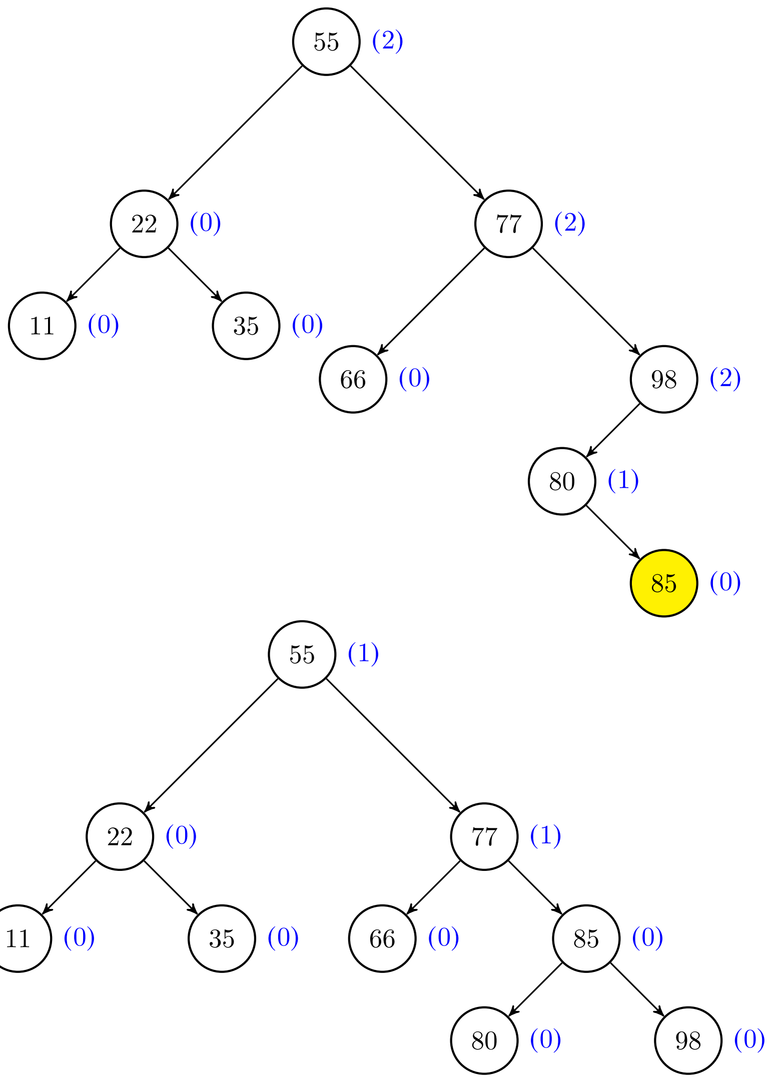


Figure 48: Insert 85: 85 > 55, Move right. 85 > 77, Move right. 85 < 98, Move left. 85 > 80, Insert right. The AVL property is violated for node 98; this format fits the case of a double rotation with $k_1 = 80$, $k_2 = 85$, $k_3 = 98$, $A = NULL$, $B = NULL$, $C = NULL$ and $D = NULL$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.

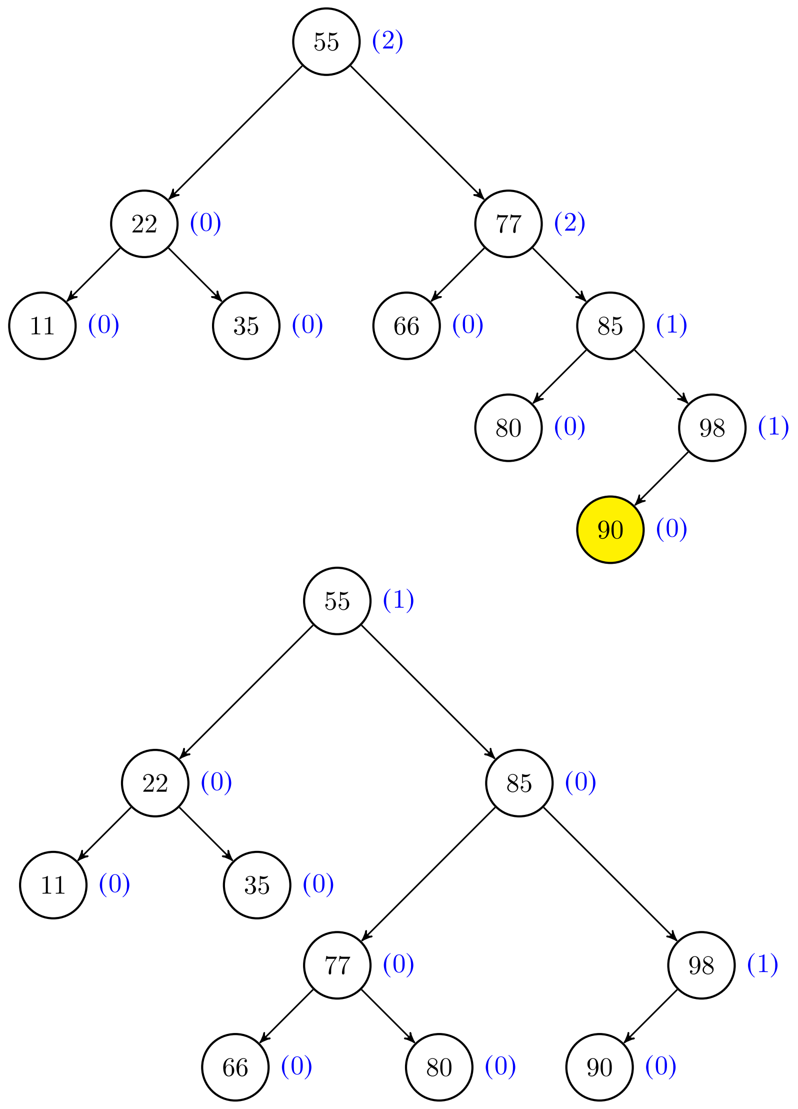
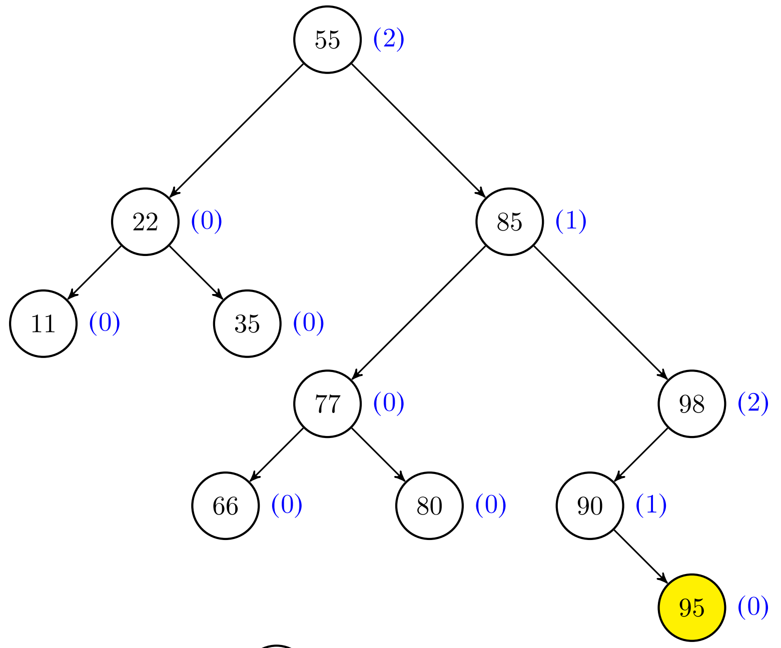


Figure 49: Insert 90: 90 > 55, Move right. 90 > 77, Move right. 90 > 85, Move right. 90 < 98, Insert left. The AVL property is violated for node 77; this format fits the case of a single rotation with $k_1 = 77$, $k_2 = 85$, $X = 66$, $Y = 80$, and $Z = 98$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.



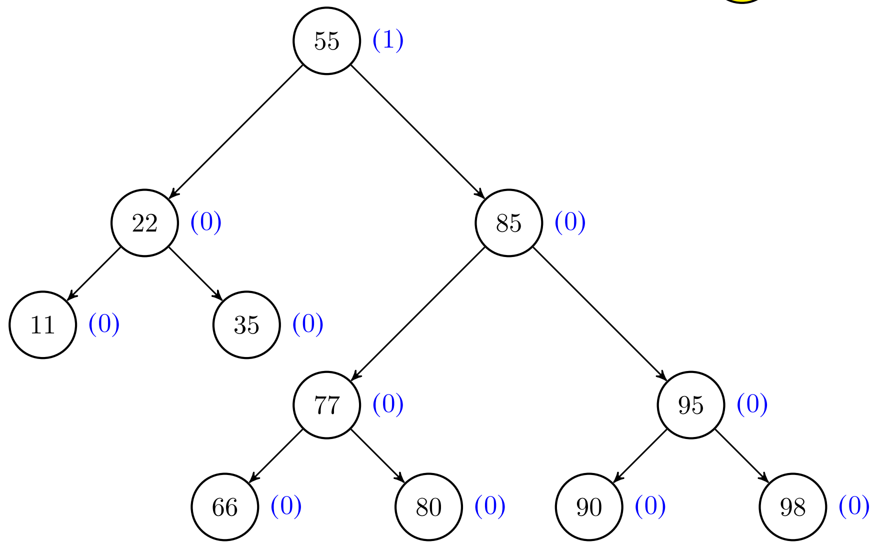


Figure 50: Insert 95: 95 > 55, Move right. 95 > 85, Move right. 95 < 98, Move left. 95 > 90, Insert right. The AVL property is violated for node 98; this format fits the case of a double rotation with $k_1 = 90$, $k_2 = 95$, $k_3 = 98$, $A = NULL$, $B = NULL$, $C = NULL$ and $D = NULL$. After performing the double rotation, we obtain the tree on the right, which maintains the AVL property.

As shown above, the AVL tree is signficantly more balanced than the binary search tree!

Problem 6

You are interested in compression. Given a file with characters, you want to find the binary code which satisfies the prefix property (no conflicts) and which minimizes the number of bits required. As an example, consider an alphabet with 8 symbols, with relative weights (frequency) of appearance in an average text file given below:

alphabet	A	L	G	O	R	I	T	H
weights	68	20	5	30	18	15	19	12

6A

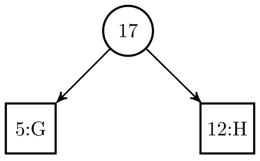
Determine the Huffman code by constructing a tree with **minimum external path length**:

$$\sum_{i=1}^{n} w_i d_i$$

(Arrange tree with smaller weights to the left)

At each round, select the two smallest weights w_1, w_2 , $w_1 \leq w_2$, build tree with parent $w_{12} = w_1 + w_2$, left child w_1, and right child w_2. Put symbol w_{12} back into the weight table for the next iteration.

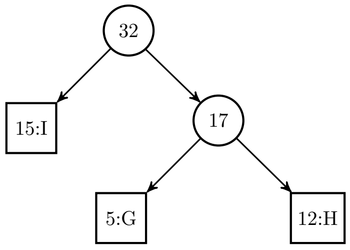
alphabet	A	L	G	O	R	I	T	H
weights	68	20	5	30	18	15	19	12



alphabet	A	L	O	R	I	T	GH
weights	68	20	30	18	15	19	17

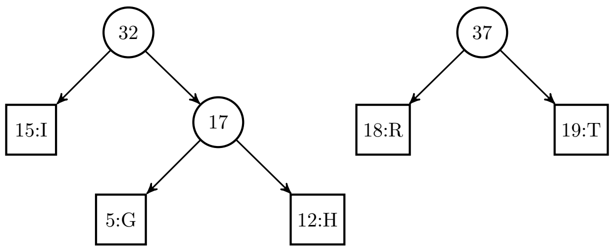
---

alphabet	A	L	O	R	I	T	GH
weights	68	20	30	18	15	19	17



alphabet	A	L	O	R	T	IGH
weights	68	20	30	18	19	32

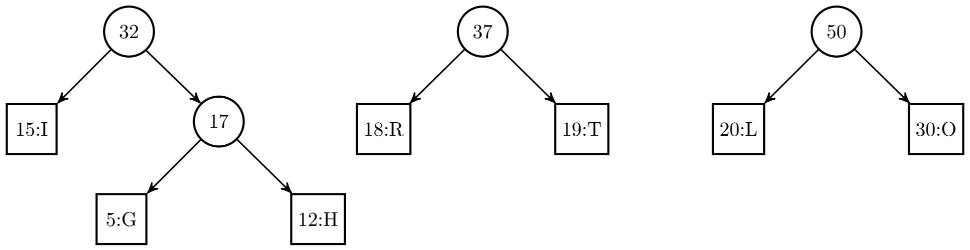
alphabet	A	L	O	R	T	IGH
weights	68	20	30	18	19	32



alphabet	A	L	O	RT	IGH
weights	68	20	30	37	32

---

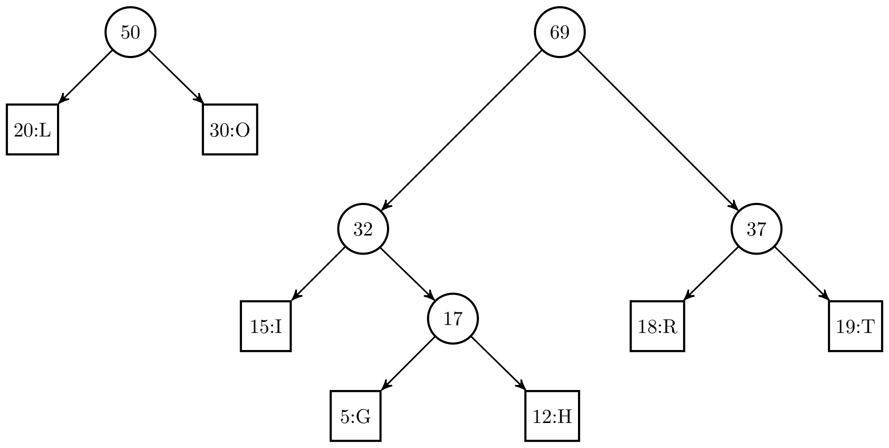
alphabet	A	L	O	RT	IGH
weights	68	20	30	37	32



alphabet	A	LO	RT	IGH
weights	68	50	37	32

---

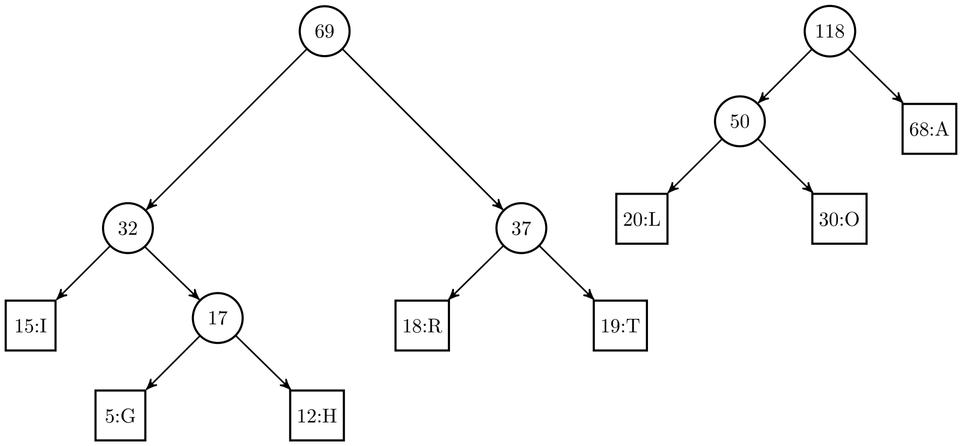
alphabet	A	LO	RT	IGH
weights	68	50	37	32



alphabet	A	LO	RTIGH
weights	68	50	69

---

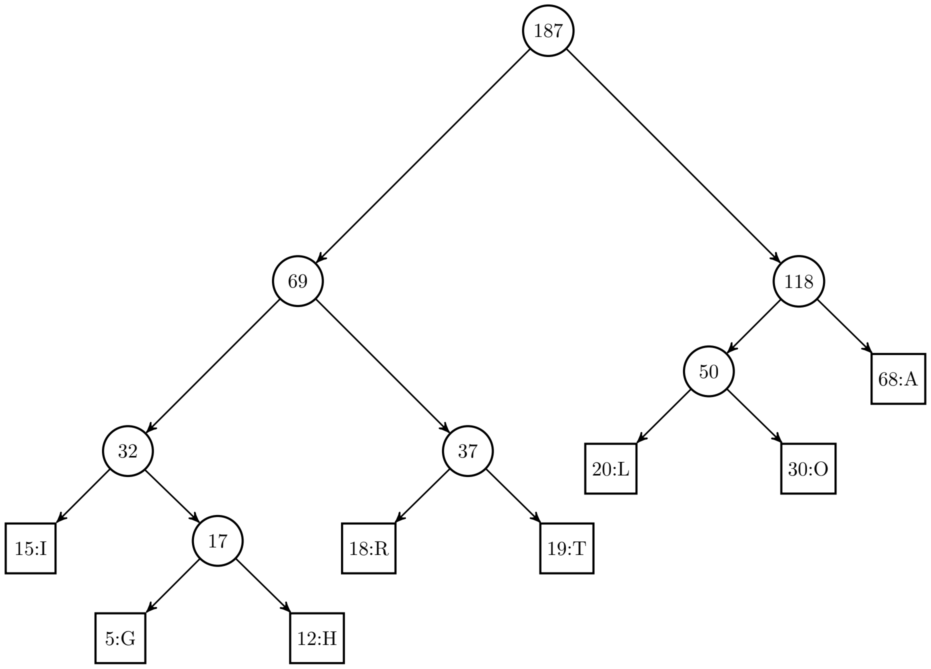
alphabet	A	LO	RTIGH
weights	68	50	69



alphabet	ALO	RTIGH
weights	118	69

---

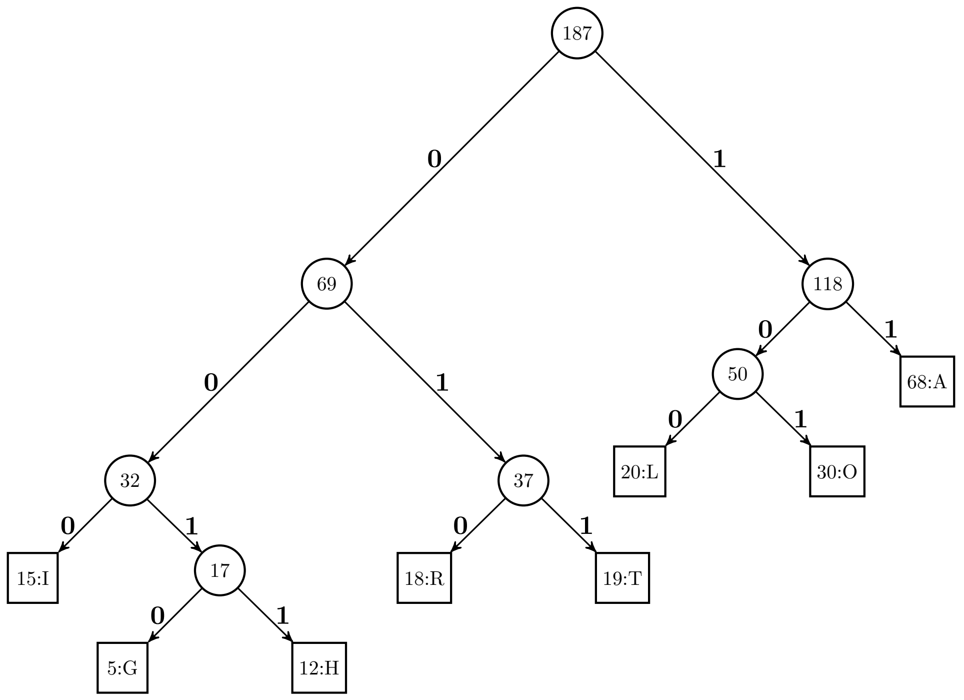
alphabet	ALO	RTIGH
weights	118	69



6B

Identify the code for each letter and list the number of bits for each letter and compute the average number of bits per symbol in this code. Is it less than 3?

With the Huffman tree completed in 6A, we can add bit assignments:



The corresponding Huffman Code is shown in the table below:

Symbol	Weight	Code	Number of Bits N_i	Probability $p_i = w_i/W$
A	68	11	2	68/187
L	20	100	3	20/187
G	5	0010	4	5/187
O	30	101	3	30/187
R	18	010	3	18/187
I	15	000	3	15/187
T	19	011	3	19/187
H	12	0011	4	12/187

To compute the average bits per symbol, we use:

$$ABPS = \sum_{i=1}^{|C|} N_i p_i$$

where $|C|$ is the size of the source alphabet, N_i is the number of bits encoding symbol i from the alphabet, and p_i is the probability of symbol i appearing. Thus:

$$ABPS = \sum_{i=1}^{|C|} N_i p_i = \frac{1}{W} \sum_{i=1}^{8} N_i w_i = \frac{1}{187} \left[2(68) + 3(20) + 4(5) + 3(30) + 3(18) + 3(15) + 3(19) + 4(12)\right]$$

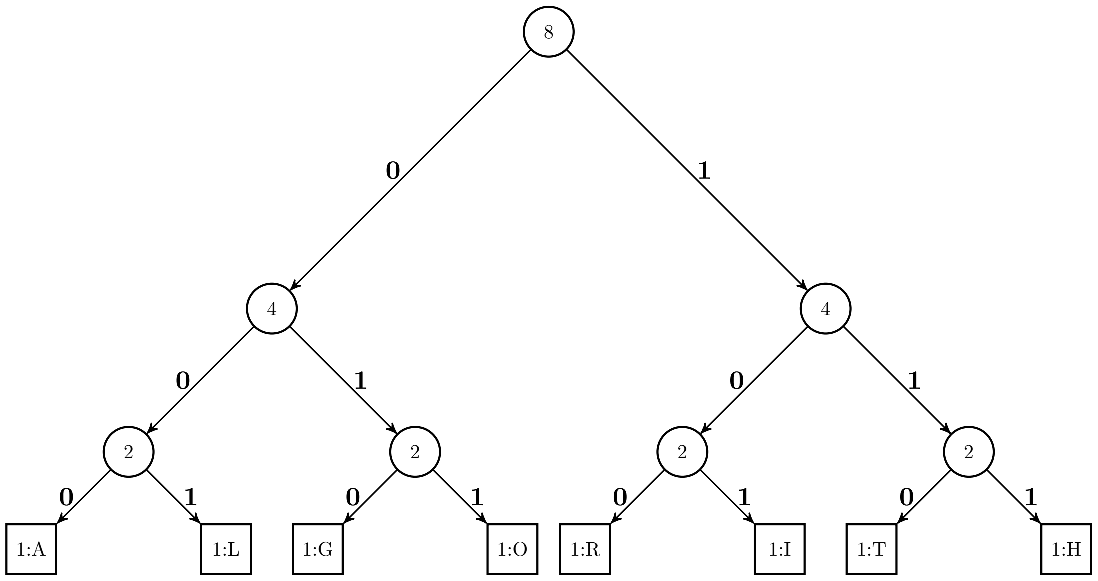
$$\therefore ABPS = 2.727$$

As expected, the average number of bits required for representing each symbol in the alphabet is less than 3, at $ABPS \approx 2.727$. The compression ratio, while optimal for this alphabet, is not particularly low given that many of the symbols occur with almost uniform frequency, thus limiting the possible space savings we could achieve using compression.

6C

Give an example of weights for these 8 symbols that would saturate 3 bits per letter. What would the Huffman tree look like? Is a Huffman code tree always a full tree?

If every symbol had uniform weights ($w_i = 1$) for all symbols in the alphabet, then the average bits per symbol would saturate at 3 bits per symbol. This is equivalent to a fixed-length encoding scheme (i.e. ASCII), which uses the same number of bits to represent each symbol, regardless of frequency. The Huffman tree is shown below:



A Huffman code will always produce a full tree; a full binary tree is a tree where every node has either 0 or 2 child nodes. At each round, there are three possible actions to take:

1. w_1 and w_2 are single elements, not trees. We join them together to form a subtree with parent node w_{12}, left child node w_1, and right child node w_2. The subtree is full, since the new node w_{12} has exactly 2 child nodes and w_1 and w_2 are both leaf nodes with 0 child nodes.

2. w_1 is a single element, but w_2 is a sub-tree. We join them to form a new subtree with parent node w_{12}, left child node w_1, and right child node w_2. Node w_{12} has exactly 2 child nodes - w_1 and w_2. Node w_1 is a leaf node, so it has 0 child nodes. w_2 is the root of a sub-tree that could only have been created by following Case 1 at some point in the algorithm; thus, the sub-tree rooted at w_2 must be a full tree. Therefore, the overall sub-tree must be a full tree.

3. w_1 and w_2 are both subtrees. We join them together to form a subtree with parent node w_{12}, left child nodes w_1, and right child node w_2. Node w_{12} has exactly 2 child nodes - w_1 and w_2, both of which are roots of subtrees. However, both w_1 and w_2 are roots of subtrees that could only have been created

by following Case 1 or Case 2 at some point in the algorithm; thus both sub-trees rooted at w_1 and w_2 respectively must be full trees. Therefore, the overall sub-tree must be a full tree.

Therefore, a Huffman code will always produce a full binary tree.

Problem 7

Given the following list of $N = 10$ elements:

28 14 7 4 6 30 36 33 10 40

7A

Insert them sequentially into a BST (Binary Search Tree). Compute the total height $T_H(N)$ and the total depth $T_D(N)$, where H is the height of the root. (Note the height of a node is the longest path length to a leaf whereas its depth is its unique distance from the root).

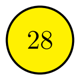


Figure 51: Insert 28 as Root

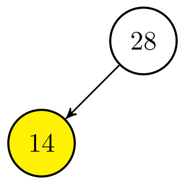


Figure 52: Insert 14: $14 < 28$, Insert left.

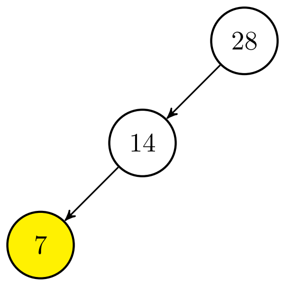


Figure 53: Insert 7: $7 < 28$, Move left. $7 < 14$, Insert left.

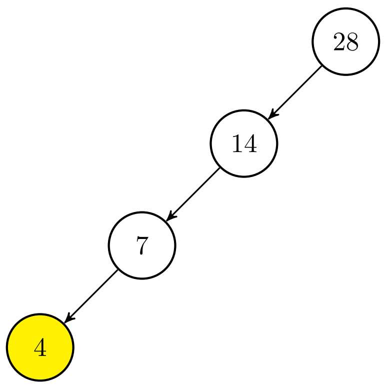


Figure 54: Insert 4: $4 < 28$, Move left. $4 < 14$, Move left. $4 < 7$, Insert left.

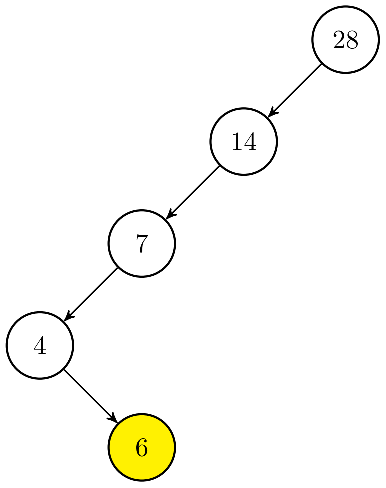


Figure 55: Insert 6: $6 < 28$, Move left. $6 < 14$, Move left. $6 < 7$, Move left. $6 > 4$, Insert right.

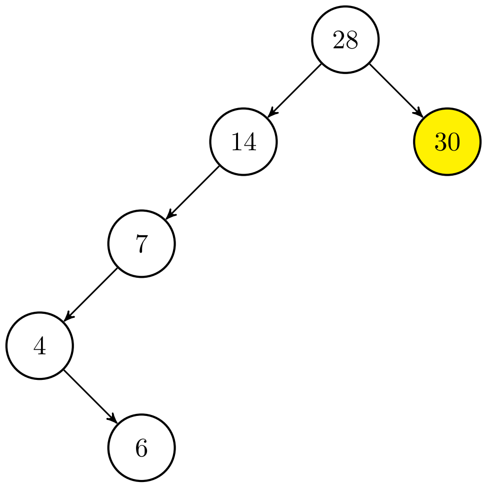


Figure 56: Insert 30: 30 > 28, Insert right.

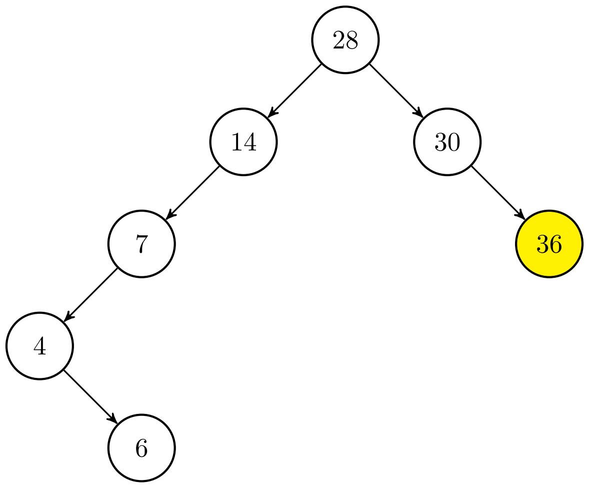


Figure 57: Insert 36: 36 > 28, Move right. 36 > 30, Insert right.

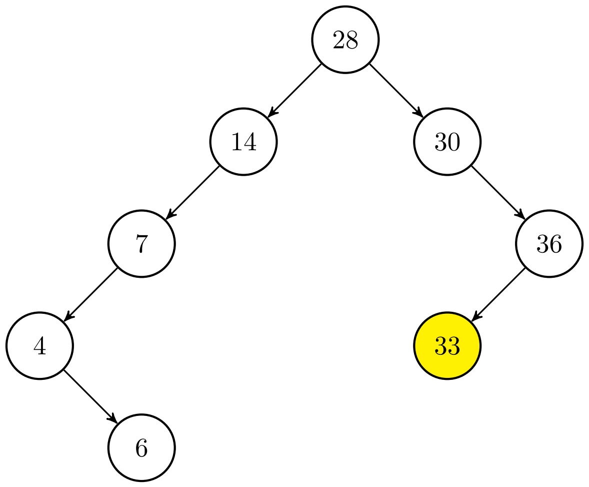


Figure 58: Insert 33: 33 > 28, Move right. 33 > 30, Move right. 33 < 36, Insert left.

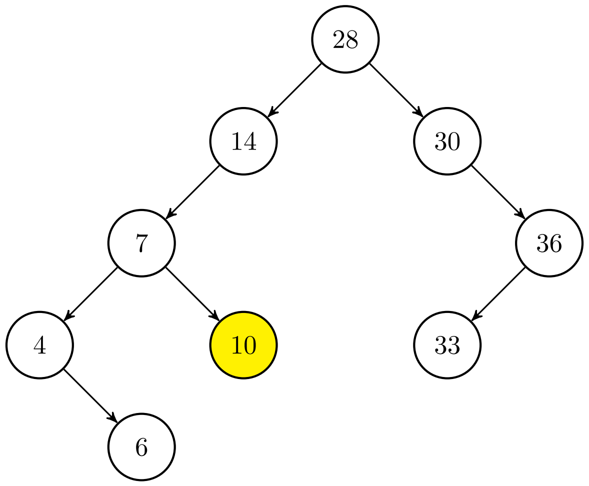


Figure 59: Insert 10: 10 < 28, Move left. 10 < 14, Move left. 10 > 7, Insert right.

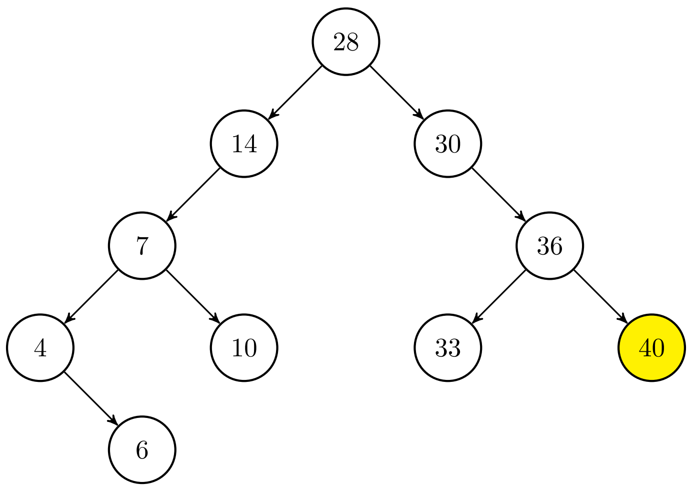


Figure 60: Insert 40: 40 > 28, Move right. 40 > 30, Move right. 40 > 36, Insert right.

7B

Find H, $T_H(N)$, $T_D(N)$ and check the sum rule:

$$T_H(N) + T_D(N) \leq HN$$

The following table summarizes the height and depth information of each node using the definitions described in 7A:

Node	h	d
28	4	0
14	3	1
30	2	1
7	2	2
36	1	2
4	1	3
10	0	3
33	0	3
40	0	3
6	0	4

By definition, H is the height of the root node - looking at the table, we see that $H = 4$. There are $N = 10$ nodes in the tree. We can compute $T_H(N)$ and $T_D(N)$ as follows:

$$T_H(N) = \sum_{i=1}^{N} h_i, \qquad T_D(N) = \sum_{i=1}^{N} d_i$$

where h_i and d_i is the height of node i and depth of node i respectively.
Thus, computing the total height and total depth, we find: $T_H(N) = 13$, $T_D(N) = 22$
We can check the sum rule:

$$T_H(N) + T_D(N) \leq HN \iff 13 + 22 \leq 40 \implies 35 \leq 40 \checkmark$$

7C

Insert them sequentially into an empty AVL tree, restoring the AVL property after each insertion. Show the AVL tree which results after each insertion and name the type of rotation (RR or LL zig-zig or RL or LR zig-zag).

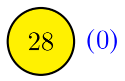


Figure 61: Insert 28 as Root. There are no child nodes, so the AVL property is maintained.

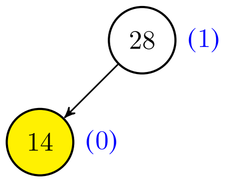


Figure 62: Insert 14: 14 < 28, Insert left. The AVL property is maintained for each node in the tree, so no need to rebalance.

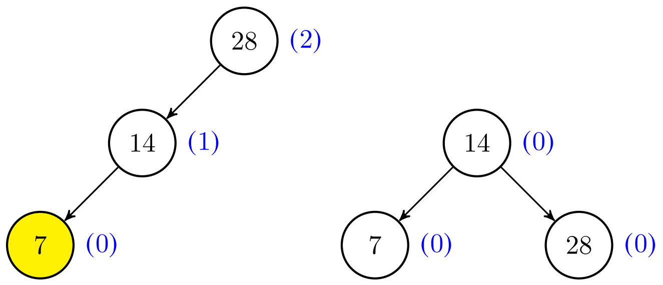


Figure 63: Insert 7: 7 < 28, Move left. 7 < 14, Insert left. The AVL property is violated for node 28; this format fits the case of a left-left single rotation with $k_1 = 14$, $k_2 = 28$, $X = 7$, $Y = NULL$, and $Z = NULL$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.

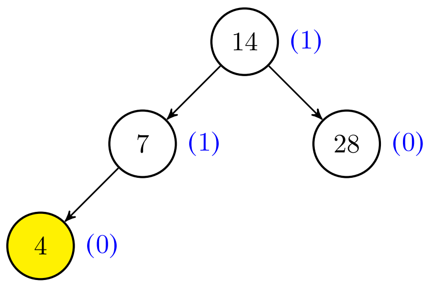


Figure 64: Insert 4: 4 < 14, Move left. 4 < 7, Insert left. The AVL property is maintained for each node in the tree, so no need to rebalance.

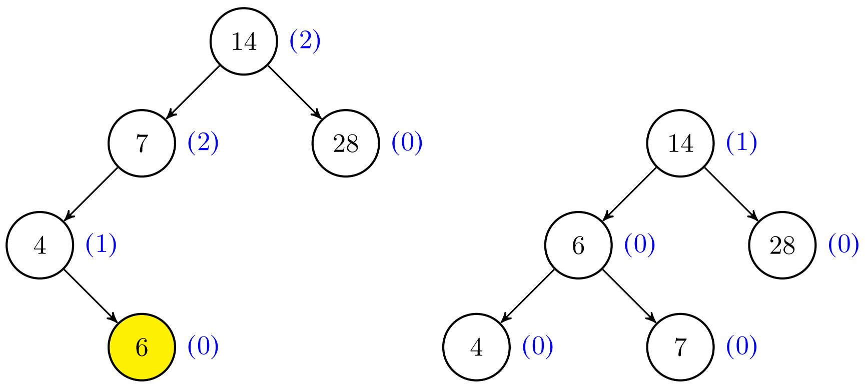


Figure 65: Insert 6: 6 < 14, Move left. 6 < 7, Move left. 6 > 4, Insert right. he AVL property is violated for node 7; this format fits the case of a left-right double rotation with $k_1 = 4$, $k_2 = 6$, $k_3 = 7$, $A = NULL$, $B = NULL$, $C = NULL$, and $D = NULL$. After performing the double rotation, we obtain the tree on the right, which maintains the AVL property.

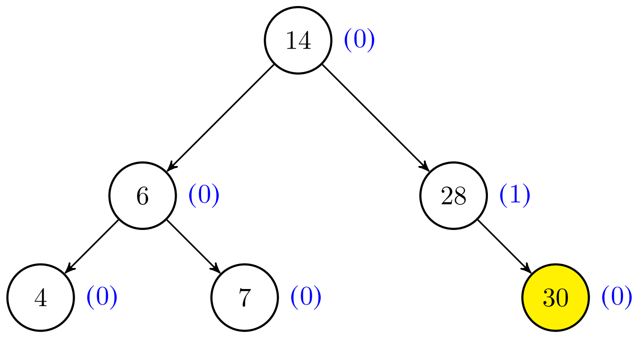
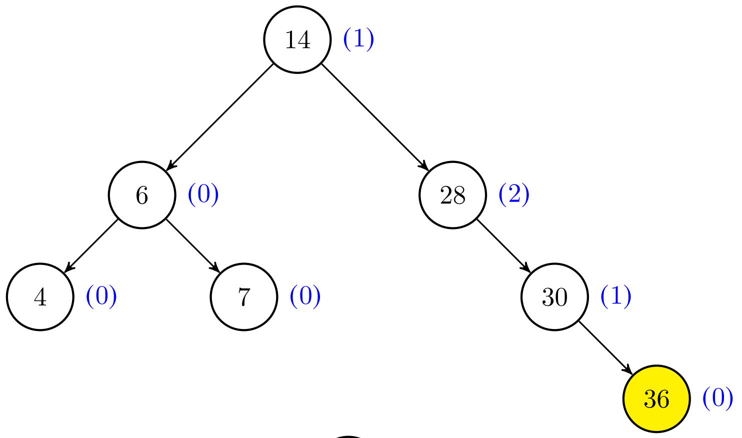


Figure 66: Insert 30: 30 > 14, Move right. 30 > 28, Insert right. The AVL property is maintained for each node in the tree, so no need to rebalance.



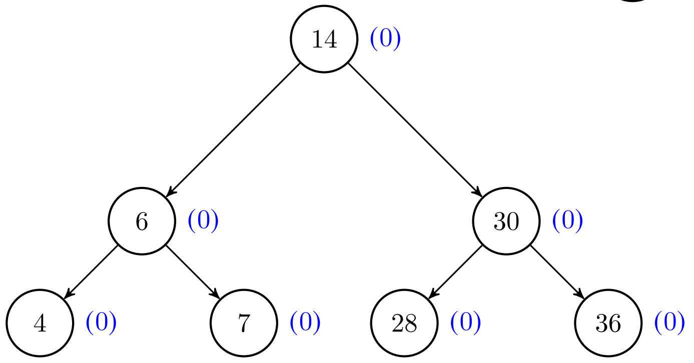


Figure 67: Insert 30: 30 > 14, Move right. 30 > 28, Insert right. The AVL property is violated for node 28; this format fits the case of a right-right single rotation with $k_1 = 28$, $k_2 = 30$, $X = NULL$, $Y = NULL$, and $Z = 36$. After performing the single rotation, we obtain the tree on the right, which maintains the AVL property.

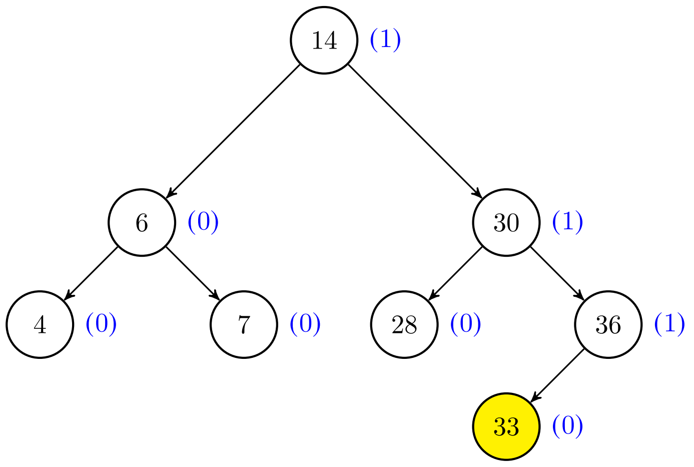


Figure 68: Insert 33. 33 > 14, Move right. 33 > 30, Move right. 33 < 36, Insert left. The AVL property is maintained for each node in the tree, so no need to rebalance.

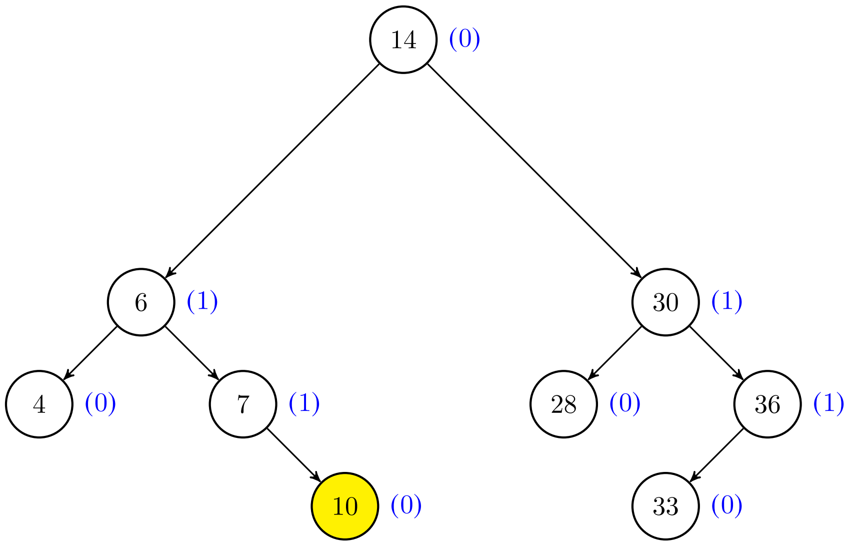


Figure 69: Insert 10. 10 < 14, Move left. 10 > 6, Move right. 10 > 7, Insert right. The AVL property is maintained for each node in the tree, so no need to rebalance.

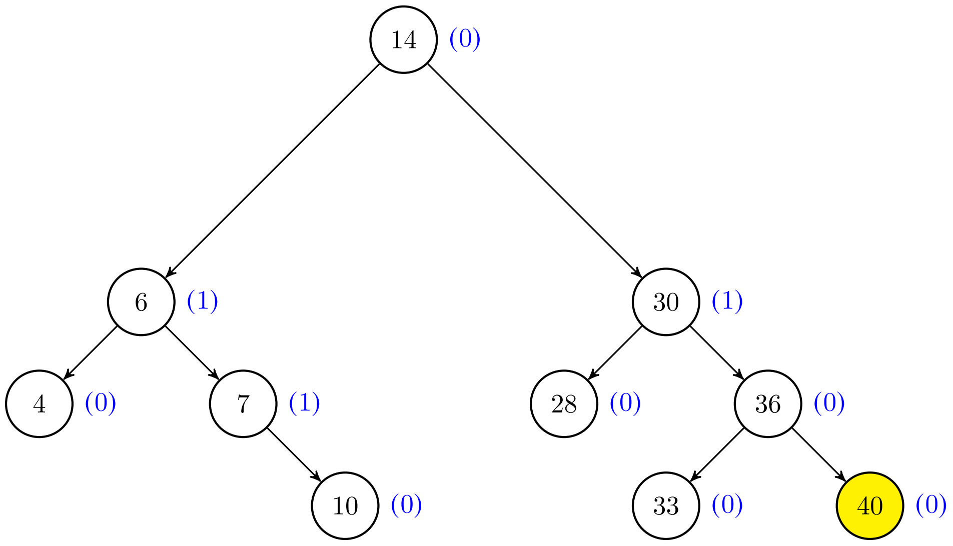


Figure 70: Insert 40. 40 > 14, Move right. 40 > 30, Move right. 40 > 36, Insert right. The AVL property is maintained for each node in the tree, so no need to rebalance.

7D

Has the final AVL tree decreased the total height $T_H(N)$ and the total depth $T_D(N)$? What are the new values? What is the new value of $T_H(N) + T_D(N)$?

The following table summarizes the height and depth information of each node using the definitions described in 7A:

Node	h	d
14	3	0
6	2	1
30	2	1
4	0	2
7	1	2
28	0	2
36	1	2
10	0	3
33	0	3
40	0	3

Using the $T_H(N)$ and $T_D(N)$ equations above, we compute the new total height and new total depth as: $T_H(N) = 9$ $T_D(N) = 19$. Using the AVL tree, we managed to decrease both the total height and total depth; the sum is given as $T_H(N) + T_D(N) = 27$.