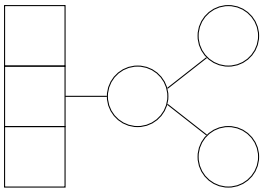
Chapter 18 Practice Exercises

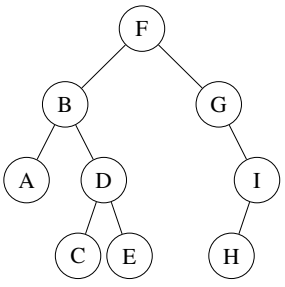
Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. What is the worst-case time complexity of searching for an element in an hash table containing n elements, where separate chaining is used, and elements are chained together using an AVL tree? An example of this is depicted in the following figure:



   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

2. Consider the following tree:



   Which of the following nodes is **NOT** an internal node?
   A) Node B
   B) Node D
   C) Node F
   D) Node I
   E) All of the above are internal nodes

3. What is the worst-case auxiliary space complexity of implementing a binary tree using an underlying array of size n?
   A) $\Theta(\log(n))$
   B) $\Theta(n)$
   C) $\Theta(n\log(n))$
   D) $\Theta(n^2)$
   E) $\Theta(2^n)$

4. Consider two binary trees, one implemented using an array-based approach and one implemented using a pointer-based approach. Which of the following correctly depicts the worst-case time complexities of inserting an arbitrary value into each binary tree?
   A) Array-based: $\Theta(1)$ Pointer-based: $\Theta(1)$
   B) Array-based: $\Theta(1)$ Pointer-based: $\Theta(n)$
   C) Array-based: $\Theta(\log(n))$ Pointer-based: $\Theta(\log(n))$
   D) Array-based: $\Theta(\log(n))$ Pointer-based: $\Theta(n)$
   E) Array-based: $\Theta(n)$ Pointer-based: $\Theta(n)$

5. If you used an array to represent a binary tree, where the root is at index 1 and the left and right children of node n are $2n$ and $2n+1$ respectively, which of the following indices would be **filled** in a rightward-facing stick (e.g., a tree where there are no left children)?
   A) Index 2
   B) Index 17
   C) Index 31
   D) Index 49
   E) Index 64

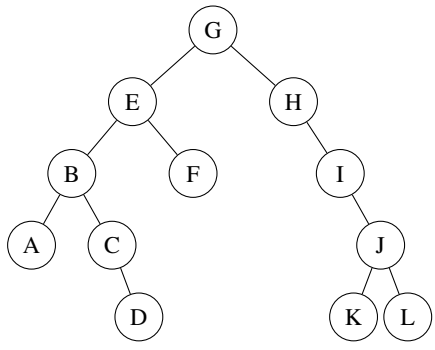
6. Suppose that you are using an array to represent a tree that can have three children: a left child, a middle child, and a right child. The root node has index 1, its children have indices 2, 3, 4, and so on (e.g., node 2's children would have indices 5, 6, and 7). Which of the following indices would be **filled** in a leftward-facing stick (e.g., every node with children can only have a left child)?
   A) Index 13
   B) Index 26
   C) Index 41
   D) Index 81
   E) Index 82

7. Suppose that you are using an array to represent a tree that can have three children: a left child, a middle child, and a right child. The root node has index 1. You know that all indices of your array are empty with the exception of indices that follow the form 3^n, where n is an integer (e.g., 1, 3, 9, 27, 81, etc.). What can you infer about this tree?
   A) This tree is a stick, and every node only has a left child
   B) This tree is a stick, and every node only has a middle child
   C) This tree is a stick, and every node only has a right child
   D) The best-case time complexity of searching this tree for an element is $\Theta(\log(n))$
   E) The worst-case time complexity of searching for an element in this tree is $\Theta(\log(n))$

8. Suppose that you want to insert 10 distinct elements into a binary search tree. How many worst-case trees are possible for these 10 elements? Note: the following are the first ten powers of two (from 2^1 to 2^{10}), in case you find them useful: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.
   A) 2
   B) 511
   C) 512
   D) 1023
   E) 1024

9. For which of the following element insertion orders would the resulting binary search tree exhibit a worst-case $\Theta(n)$ complexity when searching for a value?
   A) 51, 13, 38, 41, 49, 46, 47, 22, 53, 8
   B) 53, 8, 51, 13, 22, 49, 47, 38, 41, 46
   C) 51, 53, 8, 46, 49, 38, 41, 13, 47, 22
   D) 8, 13, 46, 41, 47, 53, 51, 38, 22, 49
   E) None of the above

For questions 10-13, consider the tree below:



10. Node ___ is the inorder predecessor of node G, and node ___ is the inorder successor of node G.
    A) A, L
    B) D, K
    C) E, H
    D) F, H
    E) F, K

11. Which of the following statements about the tree is **FALSE**?
    A) Node B is a sibling of node I
    B) Node E is an ancestor of node C
    C) Node K is a descendant of node H
    D) Node L is a child of node J
    E) None of the above

12. Which of the following statements about the tree is **TRUE**?
    A) The height of node B is one greater than the height of node A
    B) The height of node F is one greater than the height of node C
    C) The height of node F is equal to the height of node B
    D) The height of node I is equal to the height of node B
    E) More than one of the above

13. Which of the following is the correct postorder traversal of this tree?
    A) G, E, B, A, C, D, F, H, I, J, K, L
    B) D, A, C, B, F, E, K, L, J, I, H, G
    C) A, D, C, B, F, E, G, K, L, J, I, H
    D) A, D, C, B, F, E, K, L, J, I, H, G
    E) None of the above

14. Which of the following statements is **TRUE**?
    - **A)** The inorder successor of the root node can have two children
    - **B)** The array-based binary tree implementation is most efficient for trees where most nodes only have at most one child
    - **C)** The worst-case time complexity of inserting an element into a pointer-based binary tree is better than the worst-case time complexity of inserting an element into an array-based one
    - **D)** For a pointer-based implementation of a binary tree with n nodes, the best-case auxiliary space complexity is equal to the worst-case auxiliary space complexity
    - **E)** More than one of the above

15. A binary tree is defined to be *proper* if every node either has zero or two children. A binary tree is defined to be *complete* if every depth of the tree has the maximum number of nodes possible (with the potential exception of the lowest depth, which must be filled from left to right). Suppose you insert the following ten elements into a binary search tree, in the following order:

    16, 44, 37, 51, 41, 30, 66, 69, 64, 13

    Which of the following statements is **TRUE**?
    - **A)** The resulting tree is proper and complete
    - **B)** The resulting tree is proper but not complete
    - **C)** The resulting tree is complete but not proper
    - **D)** The resulting tree is neither proper nor complete
    - **E)** Not enough information is given to answer the question

16. Suppose that you know that a tree has fewer than five nodes. Which of the following **CANNOT** be true?
    - **A)** The tree is proper and complete
    - **B)** The tree is proper but not complete
    - **C)** The tree is complete but not proper
    - **D)** The tree is neither proper nor complete
    - **E)** None of the above (i.e., all of the above could potentially be true)

17. You are given the following function that takes in the root of a binary tree:

```
1  void foo(Node* root) {
2    if (!root) {
3      return;
4    } // if
5    visit(root->value);
6    foo(root->left);
7    foo(root->right);
8  } // foo()
```

    What does this function do?
    - **A)** It conducts a preorder traversal
    - **B)** It conducts a postorder traversal
    - **C)** It conducts an inorder traversal
    - **D)** It conducts a level order traversal
    - **E)** None of the above

18. You are given the following function that takes in the root of a binary tree:

```
1  void bar(Node* root) {
2    if (!root) {
3      return;
4    } // if
5    bar(root->left);
6    visit(root->value);
7    bar(root->right);
8  } // bar()
```

    What does this function do?
    - **A)** It conducts a preorder traversal
    - **B)** It conducts a postorder traversal
    - **C)** It conducts an inorder traversal
    - **D)** It conducts a level order traversal
    - **E)** None of the above

19. You are given the following function that takes in the root of a binary tree:

```
1  int32_t baz(Node* root) {
2    if (!root) {
3      return 0;
4    } // if
5    return 1 + std::max(baz(root->left), baz(root->right));
6  } // baz()
```

What does this function do?

A) It returns the number of internal nodes in the tree
B) It returns the number of leaf nodes in the tree
C) It returns the number of leaf nodes in the tree, plus one for the root
D) It returns the height of the tree
E) None of the above

20. You are given the following function that takes in the root of a binary tree:

```
1  int32_t qux(Node* root) {
2    if (!root) {
3      return 0;
4    } // if
5    if (!root->left && !root->right) {
6      return 0;
7    } // if
8    return 1 + qux(root->left) + qux(root->right);
9  } // qux()
```

What does this function do?

A) It returns the number of internal nodes in the tree
B) It returns the number of leaf nodes in the tree
C) It returns the number of leaf nodes in the tree, plus one for the root
D) It returns the height of the tree
E) None of the above

21. Which of the following statements is/are **TRUE**?

I. Only knowing the preorder and postorder traversals of a binary tree is not enough to uniquely identify that binary tree.
II. Given a binary search tree of integers, the first number in its postorder traversal must be the smallest element in the tree.
III. In a binary search tree, the inorder predecessor of the root node cannot have a right child.

A) I only
B) I and II only
C) I and III only
D) II and III only
E) I, II, and III

22. Suppose you are given the following preorder and inorder traversals of a binary tree:

Preorder: 12, 6, 9, 18, 15, 13, 11
Inorder: 6, 18, 9, 12, 13, 15, 11

What is the postorder traversal of this tree?

A) 9, 18, 6, 11, 13, 15, 12
B) 9, 18, 6, 13, 11, 15, 12
C) 18, 9, 6, 12, 13, 11, 15
D) 18, 9, 6, 13, 11, 15, 12
E) None of the above

23. Suppose you are given the following preorder and inorder traversals of a binary tree:

Preorder: 14, 6, 23, 18, 37, 41, 2, 29
Inorder: 23, 18, 6, 37, 14, 2, 29, 41

What is the postorder traversal of this tree?

A) 18, 23, 6, 37, 2, 29, 41, 14
B) 18, 23, 37, 6, 29, 2, 41, 14
C) 23, 18, 37, 6, 2, 29, 41, 14
D) 37, 18, 23, 6, 29, 2, 41, 14
E) None of the above

24. Suppose you are given the following postorder and inorder traversals of a binary tree:

    Postorder: 14, 22, 43, 18, 23, 35, 12, 27, 16
    Inorder: 22, 14, 23, 43, 18, 16, 12, 35, 27

    What is the preorder traversal of this tree?
    A) 14, 43, 22, 18, 23, 16, 27, 35, 12
    B) 14, 43, 22, 18, 35, 23, 12, 27, 16
    C) 16, 23, 22, 14, 18, 43, 27, 12, 35
    D) 16, 23, 22, 18, 14, 43, 27, 12, 35
    E) None of the above

25. Suppose you are given the following postorder and inorder traversals of a binary tree:

    Postorder: 32, 5, 2, 7, 11, 19, 28, 20
    Inorder: 5, 32, 20, 28, 11, 7, 2, 19

    What is the preorder traversal of this tree?
    A) 20, 5, 32, 28, 11, 19, 7, 2
    B) 20, 5, 32, 28, 19, 11, 7, 2
    C) 20, 28, 11, 7, 2, 19, 5, 32
    D) 20, 32, 5, 2, 7, 11, 19, 28
    E) None of the above

26. For a certain binary tree, its inorder traversal is the exact reverse of its postorder traversal. What can you infer about this binary tree?
    A) No node in the tree has a left child (i.e., a rightward-facing stick)
    B) No node in the tree has a right child (i.e., a leftward-facing stick)
    C) Its preorder traversal is the exact reverse of its inorder traversal
    D) Its preorder traversal is the same as its postorder traversal
    E) More than one of the above

27. Which of the following data structures is most suitable for conducting a level order traversal of a tree?
    A) Hash table
    B) Array
    C) Linked list
    D) Stack
    E) Queue

28. Suppose you are given the following preorder and inorder traversals of a binary tree:

    Preorder: 15, 24, 10, 33, 9, 16, 21
    Inorder: 10, 24, 15, 33, 16, 9, 21

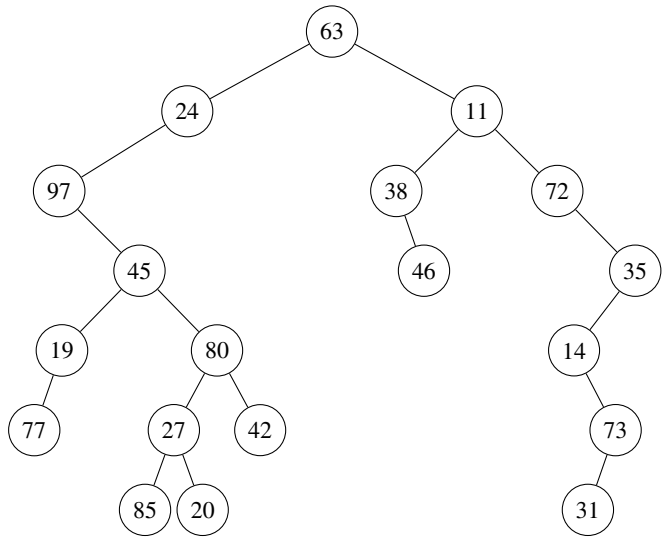
    What is the level order traversal of this tree?
    A) 10, 24, 15, 33, 9, 16, 2
    B) 10, 24, 16, 2, 9, 33, 15
    C) 15, 24, 10, 16, 33, 9, 2
    D) 15, 24, 33, 10, 9, 16, 2
    E) None of the above

29. Suppose you are given the following postorder and inorder traversals of a binary tree:

    Postorder: 62, 47, 11, 25, 42, 34, 50
    Inorder: 50, 11, 62, 47, 34, 25, 42

    What is the level order traversal of this tree?
    A) 11, 25, 34, 42, 47, 50, 62
    B) 50, 11, 62, 34, 47, 42, 25
    C) 50, 34, 11, 42, 47, 25, 62
    D) 50, 34, 11, 47, 62, 42, 25
    E) None of the above

For questions 30-35, consider the tree below:



30. In this tree, the inorder predecessor of the root node has a value of a, and the inorder successor of the root has a value of b. What is the value of $a + b$?
    A) 35
    B) 62
    C) 70
    D) 80
    E) None of the above

31. In this tree, the first element of a preorder traversal has a value of c, and the last element of a postorder traversal has a value of d. What is the value of $c + d$?
    A) 94
    B) 108
    C) 126
    D) 140
    E) None of the above

32. What is the value of the first element in an inorder traversal of this tree?
    A) 63
    B) 77
    C) 85
    D) 97
    E) None of the above

33. What is the value of the third to last element in an inorder traversal of this tree (i.e., [… , __, **X**, __, __])?
    A) 14
    B) 31
    C) 35
    D) 73
    E) None of the above

34. The root node of 63 is the _____ value in an inorder traversal of this tree.
    A) 1st
    B) 10th
    C) 11th
    D) 19th (last)
    E) None of the above

35. In which of the following traversals does the value of 24 (left child of root) occur at the latest position?
    A) Preorder
    B) Postorder
    C) Inorder
    D) Level order
    E) More than one of the above (i.e., there is a tie for the latest occurrence of 24)

36. Consider a tree that satisfies the following conditions:
    1. The tree is a binary search tree of integers.
    2. The number of elements in the root node's left and right subtrees are the same.
    3. There are no duplicate values in the tree.
    4. The first element of an inorder traversal is 1.
    5. The last element of an inorder traversal is 13.
    6. The last element of a postorder traversal is 5.

    What is the largest possible integer you can attain by summing up all the values in a tree that satisfies the above constraints?
    A) 61
    B) 70
    C) 82
    D) 91
    E) None of the above

37. In a binary tree with 12 nodes, what is the smallest possible number of leaf nodes that can exist?
    A) 1
    B) 2
    C) 3
    D) 6
    E) None of the above

38. In a *complete* binary tree with 281 nodes, how many of the nodes are leaves?
    A) 35
    B) 70
    C) 140
    D) 141
    E) None of the above

39. Suppose you want to insert the value 19 into a binary search tree. Which of the following is an *invalid* sequence of nodes that could be visited during the insertion?
    A) 23, 14, 16, 18, 21, 20
    B) 15, 16, 20, 17, 21, 18
    C) 33, 30, 27, 24, 21, 18
    D) 1, 4, 36, 5, 34, 17, 18
    E) None of the above

40. Given a binary search tree of size n, what is the worst-case time complexity of finding the inorder successor of the root node, if you use the most efficient algorithm?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

41. Which of the following correctly summarizes the worst-case time complexities of searching for a value in a binary tree versus a binary search tree, each of size n?
    A) Binary tree: $\Theta(\log(n))$ Binary search tree: $\Theta(\log(n))$
    B) Binary tree: $\Theta(\log(n))$ Binary search tree: $\Theta(n)$
    C) Binary tree: $\Theta(n)$ Binary search tree: $\Theta(\log(n))$
    D) Binary tree: $\Theta(n)$ Binary search tree: $\Theta(n)$
    E) None of the above

42. Given a tree with n nodes, where each node can have at most k children, what is the worst-case auxiliary space that could be required to implement this tree using an array-based approach?
    A) $\Theta(n)$
    B) $\Theta(nk)$
    C) $\Theta(k\log(n))$
    D) $\Theta(n^k)$
    E) $\Theta(k^n)$

43. Given a binary search tree with n nodes and height h, what is the worst-case time complexity of searching for a value, in terms of n and h?
    A) $\Theta(\log(h))$
    B) $\Theta(\log(n))$
    C) $\Theta(n\log(h))$
    D) $\Theta(h)$
    E) None of the above

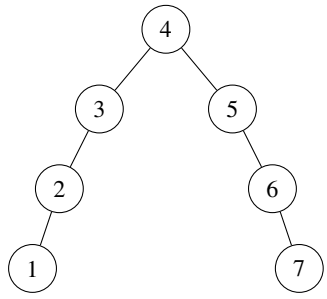
44. In a complete binary tree of size n, the maximum number of nodes in the tree that could have exactly one child is _____ if n is odd, and _____ if n is even.
    A) 0, 0
    B) 0, 1
    C) 1, 0
    D) $\lfloor n/2 \rfloor$, $n/2$
    E) $\lceil n/2 \rceil$, $n/2$

45. You are given a *binary search tree* with the following preorder traversal:

    Preorder: 5, 3, 1, 2, 4, 7, 6

    What is the postorder traversal of this tree?
    A) 2, 1, 4, 3, 6, 7, 5
    B) 2, 1, 4, 3, 7, 6, 5
    C) 1, 2, 4, 3, 6, 7, 5
    D) 1, 2, 4, 3, 7, 6, 5
    E) Not enough information is provided to answer this question

46. True or false? The following tree is balanced.



    A) True
    B) False

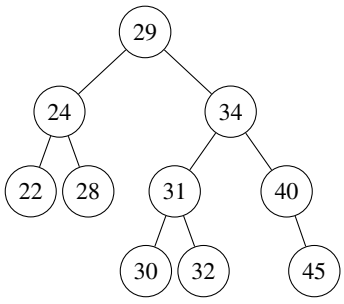
47. Which of the following statements about AVL trees is **TRUE**?
    A) To balance any tree with a node that has a negative balance factor, one should conduct a single rightward rotation about this node
    B) To balance any tree with a node that has a negative balance factor, one should conduct a single leftward rotation about this node
    C) A tree with more elements on its right side than its left side will have a positive balance factor
    D) Both A and B
    E) None of the above

48. You are given an array of unsorted elements, and you want to sort these elements and print them out in sorted order. What is the worst-case time complexity of doing this if you push all the elements into an AVL tree and then perform an inorder traversal?
    A) $\Theta(\log(n))$
    B) $\Theta(n)$
    C) $\Theta(n\log(n))$
    D) $\Theta(n^2)$
    E) $\Theta(n^2\log(n))$

49. What is the worst-case time complexity of searching for an element in an AVL tree with $n^4 3^n$ elements, in terms of n?
    A) $\Theta(\log(n))$
    B) $\Theta(n)$
    C) $\Theta(n\log(n))$
    D) $\Theta(n^4)$
    E) $\Theta(3^n)$

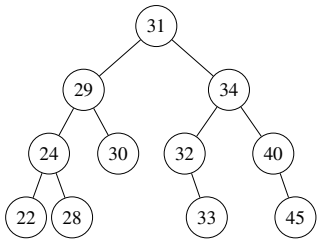
50. For which of the following operations is a triple rotation possible?
    A) Inserting an element into an AVL tree containing 31 nodes
    B) Inserting an element into an AVL tree containing 32 nodes
    C) Inserting an element into an AVL tree containing 33 nodes
    D) Both A and B
    E) None of the above

51. Consider the following AVL tree:

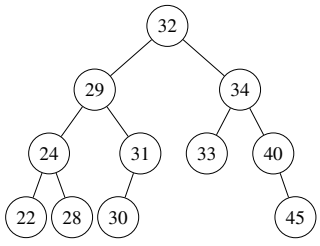


What does the AVL tree look like after 33 is inserted and all rotations are completed?

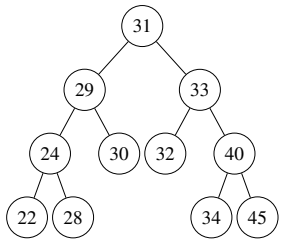
A)



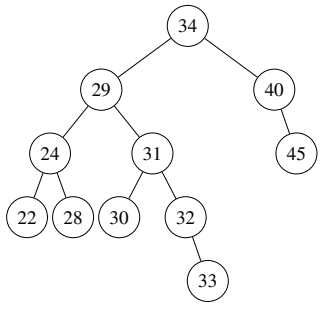
B)



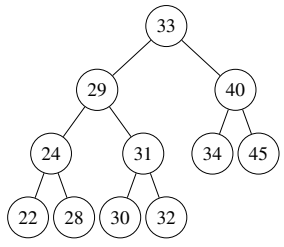
C)



D)



E)



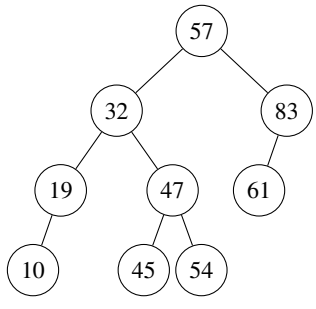
52. Insert the following elements into an empty AVL tree, rebalancing when necessary.

23, 26, 24, 25, 29, 11, 13, 9, 8, 16, 17

After all the elements are inserted, what is the level-order traversal of this tree?

A) 23, 11, 24, 8, 16, 26, 9, 13, 17, 25, 29
B) 23, 11, 25, 8, 16, 24, 29, 9, 13, 17, 26
C) 24, 13, 26, 9, 17, 25, 29, 8, 11, 16, 23
D) 24, 13, 26, 17, 9, 25, 29, 16, 23, 8, 11
E) None of the above

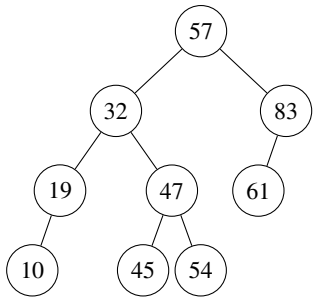
53. Consider the following AVL tree:



Suppose you deleted 57 from the AVL tree above and rebalanced the tree by replacing the root with the **inorder successor**. What is the *level-order* traversal of the resulting tree?

A) 32, 19, 61, 10, 47, 83, 45, 54
B) 47, 32, 61, 19, 45, 54, 83, 10
C) 54, 32, 83, 19, 47, 61, 10, 45
D) 61, 32, 83, 19, 47, 10, 45, 54
E) None of the above

54. Consider the following AVL tree:



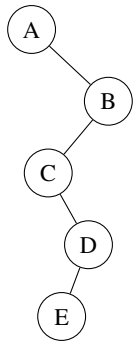
Suppose you deleted 57 from the AVL tree above and rebalanced the tree by replacing the root with the **inorder predecessor**. What is the *postorder* traversal of the resulting tree?

A) 10, 19, 32, 45, 47, 54, 61, 83
B) 10, 19, 45, 47, 32, 61, 83, 54
C) 10, 45, 19, 47, 61, 32, 83, 54
D) 45, 54, 10, 47, 83, 19, 61, 32
E) None of the above

55. Suppose you have a tree with a height of 5, where the leaf nodes have height 1. Let a represent the maximum number of nodes in a balanced AVL tree of height 5, and let b represent the minimum number of nodes in a balanced AVL tree of height 5. What is the value of $a - b$?

A) 12
B) 19
C) 20
D) 31
E) 43

For questions 56-57, consider the following stick tree:



56. You are told to balance this stick. To do this, start from the bottom node and move upwards toward the root, calculate the balance factor, and rotate whenever necessary. After the stick is balanced, which node becomes the root node?

A) A
B) B
C) C
D) D
E) E

57. After the stick is balanced, what is the balance factor of the root node?

A) -2
B) -1
C) 0
D) 1
E) 2

58. You are told that a certain AVL tree has the following postorder traversal:

8, 11, 21, 14, 30, 28, 35, 37, 34, 26

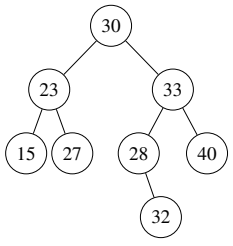
Suppose you add the values of 9 and 29 to this AVL tree, in this order, and balance the tree accordingly. What is the *preorder* traversal of the resulting tree?

A) 26, 14, 8, 9, 11, 21, 34, 28, 30, 29, 37, 35
B) 26, 14, 9, 8, 11, 21, 34, 29, 28, 30, 37, 35
C) 26, 14, 34, 8, 21, 28, 37, 11, 30, 35, 9, 29
D) 26, 14, 34, 9, 21, 29, 34, 8, 11, 28, 30, 35
E) Not enough information is provided to answer this question

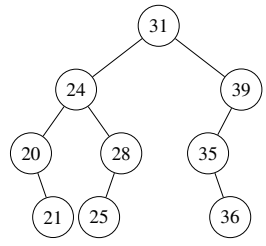
59. Given an empty AVL tree, what is the minimum number of rotations required to insert n values into the AVL tree, provided that you know what these n values are beforehand and can choose the insertion order?
    A) 0
    B) 1
    C) $\log(n)$
    D) n
    E) $n\log(n)$

60. Which of the following is a valid AVL tree?

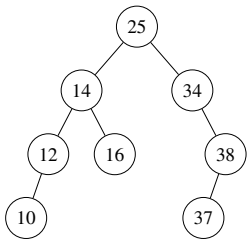
    A)

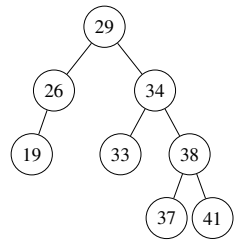
    C)

    B)

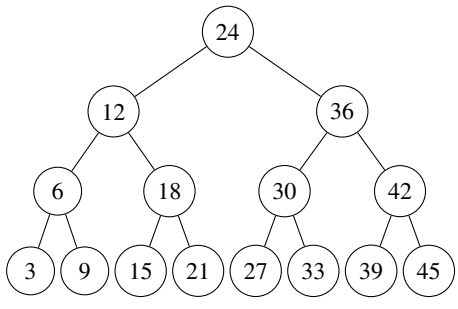
    

    D)

    E) More than one of the above

61. You are given the following AVL tree. How many more insertions (at a minimum) will you need if you want the root to have a height of 6, assuming that rebalancing is completed after each insertion? The height of a tree consisting of a single node is 1 (so the root of the tree currently has a height of 4).

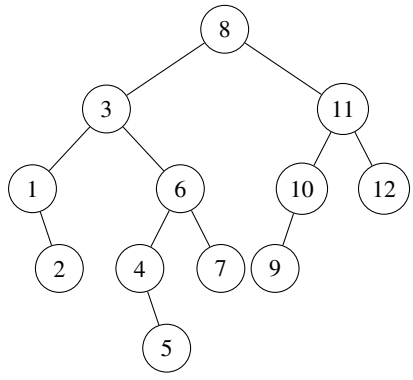
    A) 5
    B) 6
    C) 7
    D) 8
    E) 9

62. How many rotations are needed to perform the following operations on an initially empty AVL tree? Note that a double rotation operation counts as two rotations. For this problem, deletions replace the node to be removed with the inorder successor.

    Insert 17, Insert 11, Insert 15, Insert 14, Insert 12, Insert 16, Insert 13, Delete 15

    A) 3
    B) 4
    C) 5
    D) 6
    E) 7

63. You are given the following AVL tree. What is the maximum number of rotations that can be induced from a single deletion from this tree? For this problem, deletions replace the node to be removed with the inorder successor. A double rotation operation counts as two rotations.



A) 1
B) 2
C) 3
D) 4
E) 5

64. What is the smallest possible size for an AVL tree where the deletion of a single node could trigger seven rotations? A double rotation operation counts as two rotations.
A) 33
B) 88
C) 128
D) 129
E) It is impossible for a single deletion to ever invoke seven rotations

65. What is the maximum number of comparisons that could be required when inserting an element into an AVL tree of size 33? Only consider comparisons with other values in the tree to determine the position of insertion (i.e., before any rebalancing takes place).
A) 4
B) 5
C) 6
D) 7
E) 8

66. You want to design an AVL tree that supports duplicate elements and satisfies the following binary search tree invariants at all times:
- All values to the left of a given node are strictly smaller
- All values to the right of a given node are greater than or equal

Which of the following implementation strategies can guarantee that these invariants will always hold true for all possible inputs?
I. Upon insertion into the AVL tree, always insert equal values as a new node into the left subtree.
II. Upon insertion into the AVL tree, always insert equal values as a new node into the right subtree.
III. Store an extra counter with each element that represents the number of times it appears in the AVL tree, and have duplicates increment that counter upon insertion.

A) I only
B) II only
C) III only
D) I and III only
E) II and III only

67. Which of the following could be the postorder traversal of a balanced binary search tree?
I. 11, 12, 14, 13, 17, 16, 15
II. 11, 13, 12, 16, 15, 17, 14
III. 12, 11, 14, 16, 17, 15, 13

A) I only
B) II only
C) III only
D) I and III only
E) I, II, and III

68. Which of the following data structures is most commonly used to implement ordered lookup containers in the STL, such as the `std::map<>` and `std::set<>`?
    - **A)** Linked list
    - **B)** Hash table
    - **C)** Binary heap
    - **D)** Self-balancing binary search tree
    - **E)** More than one of the above

69. Suppose you are given an object of custom type `T`, where `T` supports `operator<` and `operator==` but does not implement a hash function. Assume no hash function is written for `T`, which of the following statements is true?
    - **A)** `T` can be used as the key type of an `std::map<>`, but not an `std::unordered_map<>`
    - **B)** `T` can be used as the key type of an `std::unordered_map<>`, but not an `std::map<>`
    - **C)** `T` can be used as the key type for both `std::unordered_map<>` and `std::map<>`
    - **D)** `T` can be used as the key type for neither `std::unordered_map<>` nor `std::map<>`
    - **E)** None of the above

70. Suppose you wanted to iterate over a `std::map<>` and print all keys in a that are strictly less than or equal to some value `K`. To accomplish this, you should use _____ to find the point at which to start the traversal and _____ to find the point at which to end the traversal.
    - **A)** `std::map::begin()` `std::map::upper_bound(K)`
    - **B)** `std::map::upper_bound(K)` `std::map::end()`
    - **C)** `std::map::find(K)` `std::map::begin()`
    - **D)** `std::map::begin()` `std::map::lower_bound(K)`
    - **E)** `std::map::begin()` `std::map::end()`

71. Suppose you wanted to iterate over a `std::map<>` and print all keys in a that are strictly greater than some value `K`. To accomplish this, you should use _____ to find the point at which to start the traversal and _____ to find the point at which to end the traversal.
    - **A)** `std::map::begin()` `std::map::lower_bound(K)`
    - **B)** `std::map::upper_bound(K)` `std::map::end()`
    - **C)** `std::map::find(K)` `std::map::end()`
    - **D)** `std::map::begin()` `std::map::lower_bound(K)`
    - **E)** `std::map::begin()` `std::map::end()`

72. You are given a `std::map<>` of values. The following statements are true about this `std::map<>`:
    - The types of the key and value in this map are both `int32_t`.
    - The key-value pair `{15, 16}` exists in this map.
    - Calling `.upper_bound(15)` on this map returns an iterator that points to the key-value pair `{17, 18}`.

    Which of the following key-value pairs cannot possibly be in this `std::map<>`?
    - **A)** `{13, 14}`
    - **B)** `{14, 15}`
    - **C)** `{16, 15}`
    - **D)** `{18, 17}`
    - **E)** More than one of the above

73. The Project 4 autograder just got released! On the first day, eight students submitted. The scores that each of these students received on their first submission are shown in the table below.

Student ID	522	883	768	417	130	614	349	265
Score	34.9	56.7	21.4	75.4	61.1	35.7	23.1	45.7

    Suppose these students are inserted into a `std::map<int32_t, double>` named `P4Scores`, where *Student ID* represents the key and *Score* represents the value. If `it = P4Scores.end()`, what is the value of `(--it)->first`?
    - **A)** 130
    - **B)** 265
    - **C)** 417
    - **D)** 768
    - **E)** 883

74. Suppose you wanted to print out the student ID with the highest score, 417. Which of the following would successfully accomplish this?
    - **A)** Inserting student 417 **first** into a `std::unordered_map<double, int32_t>` named `P4Scores` with *Score* as the key and *Student ID* as the value, setting an iterator `it` to `P4Scores.begin()`, and printing `it->second`
    - **B)** Inserting student 417 **last** into a `std::unordered_map<double, int32_t>` named `P4Scores` with *Score* as the key and *Student ID* as the value, setting an iterator `it` to `P4Scores.end()`, and printing `(--it)->second`
    - **C)** Inserting student 417 **last** into a `std::map<double, int32_t>` named `P4Scores` with *Score* as the key and *Student ID* as the value, setting an iterator `it` to `P4Scores.begin()`, and printing `it->second`
    - **D)** Inserting student 417 **first** into a `std::map<double, int32_t>` named `P4Scores` with *Score* as the key and *Student ID* as the value, setting an iterator `it` to `P4Scores.end()`, and printing `(--it)->second`
    - **E)** Inserting student 417 **last** into a `std::map<double, int32_t>` named `P4Scores` with *Score* as the key and *Student ID* as the value, setting an iterator `it` to `P4Scores.end()`, and printing `it->second`

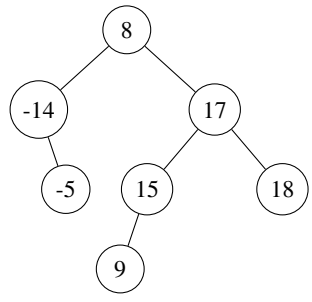
75. Which of the following correctly summarizes the average-case time complexities of inserting n elements into a `std::unordered_map<>` versus a `std::map<>`?
    - **A)** `std::unordered_map<>`: $\Theta(n)$ `std::map<>`: $\Theta(n)$
    - **B)** `std::unordered_map<>`: $\Theta(n)$ `std::map<>`: $\Theta(n\log(n))$
    - **C)** `std::unordered_map<>`: $\Theta(n^2)$ `std::map<>`: $\Theta(n)$
    - **D)** `std::unordered_map<>`: $\Theta(n^2)$ `std::map<>`: $\Theta(n\log(n))$
    - **E)** None of the above

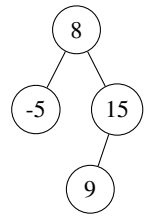
76. You are given a list of distinct integers that are not guaranteed to be in any order. If you want to print these integers out in order, which of the following implementations could potentially give you the worst asymptotic runtime for completing this task?
    - **A)** Inserting each integer into a minimum priority queue one-by-one, and then popping out and printing each value
    - **B)** Inserting each integer into a vector one-by-one and calling `std::sort()`, and then iterating from `.begin()` to `.end()`
    - **C)** Inserting each integer into a binary search tree one-by-one, and then performing an inorder traversal
    - **D)** Inserting each integer into a `std::set<>` one-by-one, and then iterating from `.begin()` to `.end()`
    - **E)** All of the methods above share the same worst-case time complexity

77. You are given a binary search tree, as well as two integers, `min_val` and `max_val`. Implement a function that removes all nodes in the BST whose values are less than `min_val` or greater than `max_val` while maintaining the binary search tree property. You may assume that `min_val` $\leq$ `max_val`. Each node of the tree is defined as follows:

```
1  struct Node {
2    int32_t val;
3    Node* left;
4    Node* right;
5    Node(int32_t val_in) : val{val_in}, left{nullptr}, right{nullptr} {}
6  };
```

Example: Given the tree on the left and the values `min_val` $= -12$ and `max_val` $= 15$, your function should trim the tree so that all elements are in the range $[-12, 15]$ (as shown by the tree on the right):





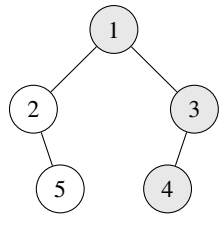

```
Node* trim_bst(Node* root, int32_t min_val, int32_t max_val);
```

Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space, where n is the number of nodes in the tree.

78. Given a binary tree, return the values of the rightmost nodes at each level, ordered from top to bottom. Each node is defined as follows:

```
1  struct Node {
2    int32_t val;
3    Node* left;
4    Node* right;
5    Node(int32_t val_in) : val{val_in}, left{nullptr}, right{nullptr} {}
6  };
```

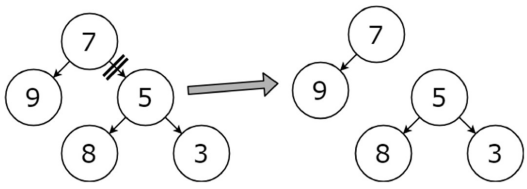
Example: Given the following tree, you would return `[1, 3, 4]`.




```
std::vector<int32_t> get_rightmost_values(Node* root);
```

Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space, where n is the number of nodes in the tree.

79. You are given a binary tree with n nodes. Write a function that checks if it is possible to partition the tree into two subtrees that have equal sums after removing exactly *one* edge of the original tree. For example, the following tree would return **`true`**, as removing the edge marked with a double slash would split the tree into two smaller subtrees that both have a sum of 16:



Each node of the tree shares the same structure as a node in the previous two problems (with `val`, `*left`, and `*right` member variables).

```
bool equal_sum_subtree(Node* root);
```

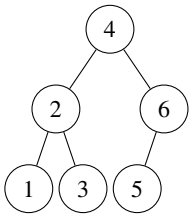
Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space. *Hint: trees are recursive structures, so think recursion! It may be helpful to write a helper function that can find the sum of a subtree when given its root.*

80. You are given the root of a *complete* binary search tree of integers. Implement a function that converts this binary search tree into a binary min-heap. Each node of the tree shares the same structure as a node in the previous problems (with `val`, `*left`, and `*right` members).

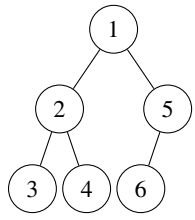
```
Node* convert_bst_to_min_heap(Node* bst_root);
```

Recall that a min-heap is a complete tree (all levels are filled except possibly the last, which must be filled from left to right) such that the value of each node is less than or equal to the values of its children. For example, given the complete binary search tree on the left, one solution would be to convert it into the binary min-heap on the right:

Binary Search Tree



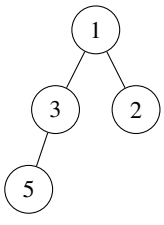
Min-Heap



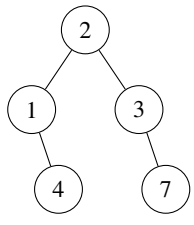
Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space, where n is the number of nodes in the tree.

81. You are given two binary trees. Implement a function that merges these two trees together by summing up the nodes that overlap. Return the root node of the combined tree. For example, given the left two trees are input, you would return the tree on the right.

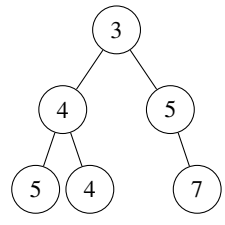
Input Tree 1



Input Tree 2



Output Tree




```
Node* merge_trees(Node* t1, Node* t2);
```

Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space, where n is the number of nodes in the tree.

82. You are given a sorted array. Implement a function that builds a **balanced** binary search tree from the contents of this array and returns the root node of the newly built tree.

```
Node* array_to_bst(const std::vector<int32_t>& nums);
```

Your solution should run in $O(n)$ time and use $O(n)$ auxiliary space, where n is the size of the input array.

83. Implement the following `StockPriceTracker` class, which can be used to return the price of a known stock at any point in time. This class supports two operations that you will need to implement:
    - `void update(const std::string& symbol, double price, int32_t timestamp);`
      - This indicates that the price of the stock with symbol `symbol` changed to a value of `price` at timestamp `timestamp`.
    - `double get_price(const std::string& symbol, int32_t timestamp);`
      - This returns the price of the stock with symbol `symbol` at timestamp `timestamp`. You may assume that the `symbol` exists and has been updated before at a timestamp less than or equal to `timestamp`.

    Here are some example operations that demonstrate this behavior:

```
StockPriceTracker spt;
spt.update("NVDA", 739.00, 1);
spt.update("MSFT", 409.49, 3);
spt.update("TSLA", 188.71, 4);
spt.update("NVDA", 739.07, 3);
spt.update("TSLA", 188.65, 6);

// The following prints out the prices of the three stocks at timestamp 5
std::cout << spt.get_price("NVDA", 5) << std::endl; // prints 739.07
std::cout << spt.get_price("MSFT", 5) << std::endl; // prints 409.49
std::cout << spt.get_price("TSLA", 5) << std::endl; // prints 188.71
```

    An outline of this class is provided below:

```
class StockPriceTracker {
private:
  // TODO: Add any data structures here!
public:
  void update(const std::string& symbol, double price, int32_t timestamp) {
    // TODO: Implement this
  } // update()

  double get_price(const std::string& symbol, int32_t timestamp) {
    // TODO: Implement this
  } // get_price()
};
```

Chapter 18 Exercise Solutions

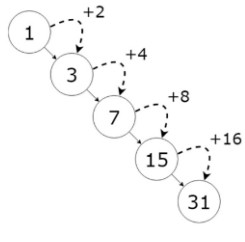
1. **The correct answer is (B).** The worst-case time complexity of finding the index an element should be at is $\Theta(1)$ for separate chaining. After finding this index, the worst-case time complexity of finding an element in the attached AVL tree is $\Theta(\log(n))$. This is the dominant complexity, so the time complexity of the overall procedure is also $\Theta(\log(n))$.

2. **The correct answer is (E).** An internal node has children; all four nodes have children, so they are all internal nodes.

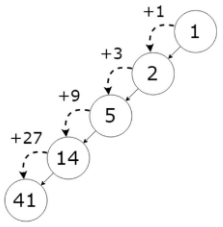
3. **The correct answer is (E).** In the case of a rightward-facing stick tree, the last element would be found at index $2^n - 1$ for an array-based binary tree. Thus, the worst-case auxiliary space complexity would be $\Theta(2^n)$.

4. **The correct answer is (E).** The worst-case of insert is $\Theta(n)$ for both cases. For an array-based tree, we could have to traverse the entire array before we find an open position to insert the new element (and if the array was completely full, we would also need to reallocate). The same applies for a pointer based tree; regardless of what algorithm you use to insert a new node into the tree, you could end up visiting $\Theta(n)$ nodes if you get unlucky and never find an open spot.

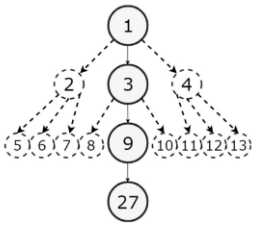
5. **The correct answer is (C).** The right child of the root is located at index 3. Since each additional row of the binary tree adds 2^n nodes, we would keep on adding powers of two until we get an answer choice, which happens to be 31. In addition, the rightmost node of any row has an index of the form $2^n - 1$, which can also be used to get 31.



6. **The correct answer is (C).** The left child of the root is located at index 2. Since each additional row of this tree has 3^n nodes (as the tree can have three children), we can continue adding 3^n until we get an answer of 41. We start with +1 instead of +3 in this case because we are dealing with the left child.



7. **The correct answer is (B).** This tree only has middle children, as shown below:



8. **The correct answer is (C).** Given any set of elements, you can either add the smallest or the largest element to build a worst-case tree. For the first 9 elements, you can select either the smallest or largest value to add, which is 2 choices for 9 decisions. Once you reach the final value, you only have one remaining element to insert. Thus, the number of ways you can build a worst-case tree using 10 elements is $2\times2\times2\times2\times2\times2\times2\times2\times2\times1 = 2^9 = 512$.

9. **The correct answer is (B).** Every value added to the tree in this sequence is either the smallest or largest of the elements that are remaining:
   - 53 is the largest element in the set of elements 53, 8, 51, 13, 22, 49, 47, 38, 41, 46
   - 8 is the smallest element in the set of elements 8, 51, 13, 22, 49, 47, 38, 41, 46
   - 51 is the largest element in the set of elements 51, 13, 22, 49, 47, 38, 41, 46
   - 13 is the smallest element in the set 13, 22, 49, 47, 38, 41, 46
   - 22 is the smallest element in the set 22, 49, 47, 38, 41, 46
   - 49 is the largest element in the set 49, 47, 38, 41, 46
   - 47 is the largest element in the set 47, 38, 41, 46
   - 38 is the smallest element in the set 38, 41, 46
   - 41 is the smallest element in the set 41, 46

   If you draw this tree out, you will see that it is a stick, which would result in worst-case search for 46 (or any number bigger than 41 and less than 47).

10. **The correct answer is (D).** To identify the inorder predecessor, we go left once, and then as far right as possible (i.e., keep on going right until you reach a node with no right child): this gives us node F. To identify the inorder successor, we go right once and then as far left as possible: this gives us node H.

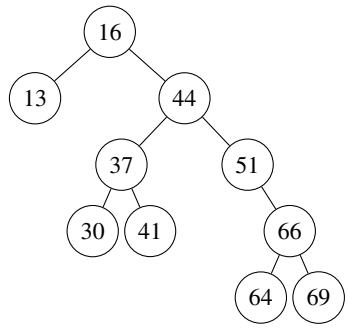
11. **The correct answer is (A).** Nodes B and I do not share the same parent, so node B is not a sibling of node I.

12. **The correct answer is (D).** If we consider leaf nodes as nodes with a height of 1, choice (A) is false because node F has height 1 (left) and node C has height 2 (since D is its child). Choice (B) is false because node B has height 3 (counting from the farthest leaf that is a child of B, B is the 3rd node from the bottom) and node A has height 1 (leaf). Choice (C) is false because node F has height 1 and node B has height 3. Choice (D) is true because nodes B and I both have height 3 (counting from the farthest leaf that is a child of I, I is the 3rd node from the bottom).

13. **The correct answer is (D).** For a postorder traversal, we print out the left child, then the right child, then the parent. Here, we go all the way to the very left and print out node A (as it has no children). Then, we go up a level to B. We check if node B has a right child: it does, so we visit it, node C. We then check if node C has a left child (it doesn't) or a right child (it does, so we head down to D). D is a leaf, so we print it out and move back to C, print out C, and then move back to B. Now that node B is all covered, we move up a level to node E and check if it has a right child. It does, so we visit, node F. Node F is a leaf, so we print it out before moving back to E. We've covered all of E's children, so then we print out E. Since the left side of the tree is done, we complete the same process for the right side (check of left children, then check for right children, then visit the parent). This produces the sequence in choice (D).

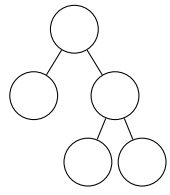
14. **The correct answer is (D).** Choice (A) is false because the inorder successor cannot have two children; otherwise, it wouldn't be the inorder successor since there is a left child. Choice (B) is false because using an array for a sparse tree is very wasteful, as many of the indices are left empty. Choice (C) is false since the worst-case time complexity of inserting an element is $\Theta(n)$ for both array-based and pointer-based binary trees. Choice (D) is true, however, since the best-case and worst-case auxiliary space complexities for a pointer-based binary tree are both $\Theta(n)$, as it only needs as much space as the number of elements there are (no need for wasted space, as with the array-based implementation).

15. **The correct answer is (D).** The following tree is created:



This tree is not complete since 44 has children but 13 (which is to the left of 44 at the same depth) does not. This tree is not proper since 51 only has a right child.

16. **The correct answer is (B).** This is the smallest tree you can build that is proper but not complete:



17. **The correct answer is (A).** A preorder traversal visits the parent, then recursively the left child, then recursively the right child, which matches this code.

18. **The correct answer is (C).** An inorder traversal visits the left child, then the parent, then the right child. This matches the pattern for an inorder traversal.

19. **The correct answer is (D).** This matches the equation for the height of a tree, which is 1 + the maximum height of the left and right child.

20. **The correct answer is (A).** This returns the number of internal nodes in the tree, since it is counting the number of nodes where the left and right child are not nullptr.

21. **The correct answer is (C).** Statement I is true because a preorder and postorder alone cannot differentiate between a single left child or right child of a node (see example 18.5.3 for an explanation why). Statement II is false if the smallest element in the tree has a right child, since the postorder traversal would visit that right child before its parent. Statement III is true because the root's inorder predecessor by definition cannot have a right child, since the right child would then the largest value smaller than the root.

22. **The correct answer is (D).** When given a preorder and inorder traversal, the first thing we do is to identify the root node of this tree. This is 12, as 12 is the first element of the preorder traversal. Now that we know this, let's split the inorder traversal into the left and right subtrees:

$$\underline{6, 18, 9}, \mathbf{12}, \underline{13, 15, 11}$$

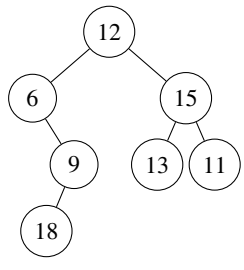
We look back at the preorder traversal to determine what the roots of the left and right subtrees are. The root of the left subtree must be 6, as it appears before 18 and 9. Similarly, the root of the right subtree must be 15, as it appears before 13 and 11:

$$\mathbf{12,} \boxed{\mathbf{6,} 9, 18,} \boxed{\mathbf{15,} 13, 11}$$

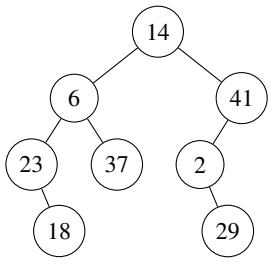
Now let's go back to the inorder and split the left and right subtree into even smaller subtrees:

$$6, \underline{18, 9}, \mathbf{12,} \underline{13}, 15, \underline{11}$$

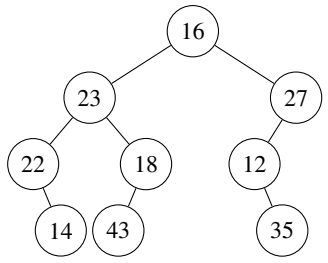
From this, we can tell that 18 and 9 form the right subtree of 6, and that 13 and 11 are the left and right children of 15. Returning to the preorder, we can conclude that 9 is the child of 6 (since 9 comes before 18). Then, going back to the inorder, we can conclude that 18 is the left child of 9 (since 18 precedes 9). The final tree is as follows:



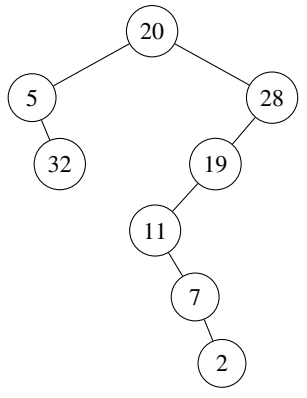
23. **The correct answer is (B).** Using the same process as the previous problem, we would get the following tree:



24. **The correct answer is (C).** The process for identifying this tree is identical to the process described in the previous question. However, since we have a postorder rather than a preorder traversal, we have to consider the last element in the postorder as the root of its respective subtree (rather than the first element as with the preorder). The final tree is as follows:



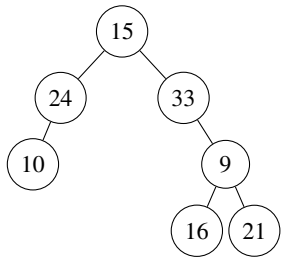
25. **The correct answer is (B).** Using the same process as the previous problem, we would get the following tree:



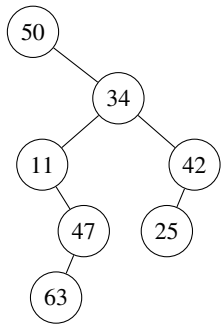
26. **The correct answer is (A).** In a postorder traversal, the last element is the root. Thus, if the inorder is the exact reverse of the postorder, the root node must be the first element in order, or the leftmost node. We can continue this process for the subtrees that remain: the right child of the root node must be the left node of its subtree, and its right child must be the leftmost node of its own subtree, etc. This means that each node can only have a right child, else the last element of the postorder would not be the first of the inorder. Choice (C) is false because the preorder would be the same as its inorder, and choice (D) is false because the preorder would be the reverse of the postorder.

27. **The correct answer is (E).** Queues work well for level order traversals, since elements that are discovered first are also explored first, allowing the search to travel downwards level by level since elements at one level must all be considered before moving to the next. As you will see in the next chapter, this is known as a breadth-first search.

28. **The correct answer is (D).** This is the tree that matches the given preorder and inorder traversals:



29. **The correct answer is (C).** This is the tree that matches the given preorder and inorder traversals:



30. **The correct answer is (B).** To find the inorder predecessor, we go left once and then as far right as possible. Since the left child of the root, 24, does not have any right children, it itself is the inorder predecessor. To find the inorder successor, we go right once and then as far left as possible. In this case, we end up at 38 (heading down the leftmost children of 11 until an element is found without a left child). The sum of 24 and 38 is 62.

31. **The correct answer is (C).** The first element of a preorder traversal and the last element of a postorder traversal are both the root, which has a value of 63. Thus, the value of $c + d$ is $63 + 63 = 126$.

32. **The correct answer is (D).** To determine the first element in an inorder traversal, we go left as far as possible until we reach a node with no left child. In this case, this is node 97.

33. **The correct answer is (B).** The third to last element in an inorder traversal is 31. The only two elements in a later position are 73 and 35, which are to the right of 31.

34. **The correct answer is (C).** There are ten nodes in the left subtree of 63. Therefore, 63 must be the 11th node in an inorder traversal.

35. **The correct answer is (E).** There is a tie between the inorder and postorder traversal, where 24 is in the 10th position. 24 is 2nd in the preorder and level order traversals.

36. **The correct answer is (A).** The last element of a postorder traversal is the root of the tree, so 5 must be the root. The first element of an inorder traversal is the smallest element in the tree, which must be 1. As a result, if you want to maximum the sum of the tree's values, there can only be 4 values in the left subtree of 5 to ensure no duplicates: 1, 2, 3, and 4. The last element of an inorder traversal is the largest value in the tree, and since the number of elements in the left and right subtrees are equal, the values in the right subtree must be 10, 11, 12, and 13 to satisfies all the conditions. The total sum of this tree is therefore $1 + 2 + 3 + 4 + 5 + 10 + 11 + 12 + 13 = 61$.

37. **The correct answer is (A).** A stick tree will always have only one leaf node, so one is the answer for a binary tree of any size.

38. **The correct answer is (D).** In a complete tree with n nodes, $\lceil n/2 \rceil$ are leaf nodes.

39. **The correct answer is (B).** When searching in a binary search tree, queries for values less than the target value of 19 should be an increasing subsequence that gets closer to 19, while queries for values larger than 19 should be a decreasing subsequence that gets closer to 19. In this case, the subsequence 20, 21 in choice (B) cannot be possible, since there would be no way we would encounter 21 if we already encountered 20 earlier (since our search path would look in the left subtree of 20, in which 21 cannot exist).

40. **The correct answer is (C).** To identify the inorder successor, we traversal one node to the right, and then go as far left as possible from this node's left child. As a result, it is possible for a tree to be configured in a way such that finding the inorder successor could require you to visit every node, resulting in a time complexity of $\Theta(n)$.

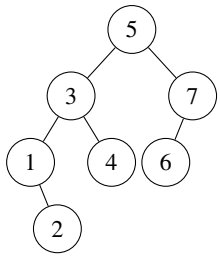
41. **The correct answer is (D).** Since the binary search tree is not guaranteed to be balanced, there is no invariant that is preventing it from becoming a stick like tree. Thus, the worst-case time complexity of search could potentially end up as $\Theta(n)$ for both binary trees and binary search trees.

42. **The correct answer is (E).** In general, the space complexity of a tree with at most k children is $\Theta(k^n)$ (see the end of 18.2.1 for an explanation).

43. **The correct answer is (D).** When searching in a binary search tree, every comparison brings you down a level of the tree (i.e., you can either visit the left child or the right child). Therefore, the number of comparisons you need is limited by the height of the tree, or h, resulting in a worst-case time complexity of $\Theta(h)$.

44. **The correct answer is (B).** Since complete binary trees must be filled from left to right, if we number each node in a tree in level order (where root is one), then an even numbered node must be a left child, and an odd numbered node must be a right child. Because of this, if there are an even number of nodes, only one node can have one child (the parent of the final element in a level order traversal), and if there are an odd number of nodes, no nodes can have only one child (due to the invariant of adding children from left to right on any given level).

45. **The correct answer is (A).** Although it may initially seem that this problem is not solvable, you are actually given the inorder traversal as well due to the fact that the tree is a binary search tree. In a BST, the inorder traversal is simply the elements in ascending order: 1, 2, 3, 4, 5, 6, 7. Using this information, we can construct the following tree that matches these traversals:



46. **The correct answer is (B).** Node 3 has a balance factor of +2, and node 5 has a balance factor of -2. Thus, the tree is not balanced.

47. **The correct answer is (E).** None of the options are true. Choice (A) and (B) are false because it is not enough to conduct a single rotation to balance a tree with a negative balance factor. Depending on the size of the negative balance factor and the structure of the tree, multiple rotations may be needed. Option (C) is false because a tree with more elements on its right side will have a negative balance factor.

48. **The correct answer is (C).** The time complexity of inserting n elements into an AVL tree with an insert complexity of $\Theta(\log(n))$ is $\Theta(n\log(n))$. The time complexity of printing these elements out is $\Theta(n)$. The insertion process dominates, so the overall time complexity is $\Theta(n\log(n))$.

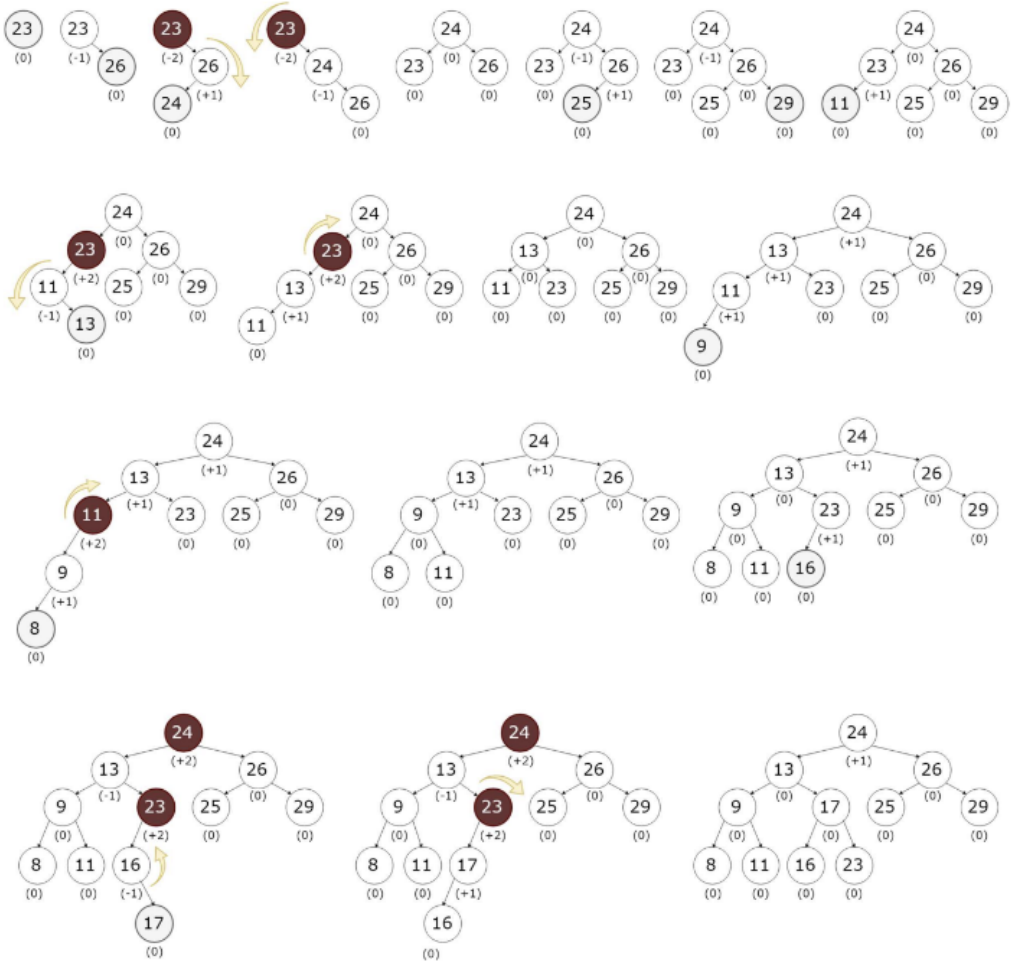
49. **The correct answer is (B).** The worst-case time complexity of searching for an element in an AVL tree with n elements is $\Theta(\log(n))$. Since there are $n^4 3^n$ elements in the tree, the worst-case time complexity is:

$$\Theta(\log(n^4 3^n)) = \Theta(\log(n^4) + \log(3^n)) = \Theta(4\log(n) + n\log(3)) = \Theta(\log(n) + n) = \Theta(n)$$

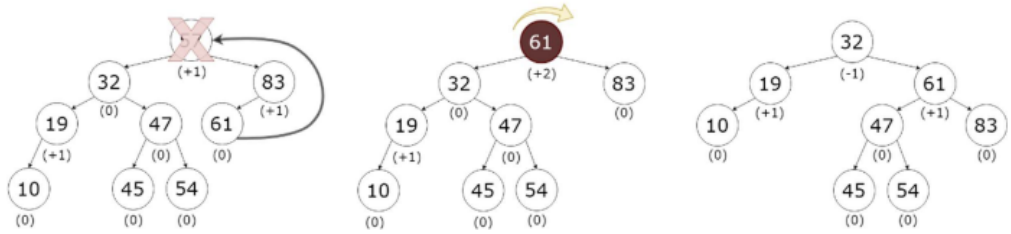
50. **The correct answer is (E).** Insertions will always require either 1 or 2 rotations (after the single or double rotation, the rest of the tree is guaranteed to be balanced). Deletions, on the other hand, may require up to $\Theta(\log(n))$ rotations.

51. **The correct answer is (A).** After inserting 33 as the right child of 32, there is a zigzag imbalance between 29 (-2) and 34 (+1). To fix this, a right rotation is performed on 34 (34 becomes the right child of 31, and 32 becomes the left child of 34), and then a left rotation is performed on 29 (29 becomes 31's left child, and 30 becomes 29's right child). This gives us the tree in option (A).

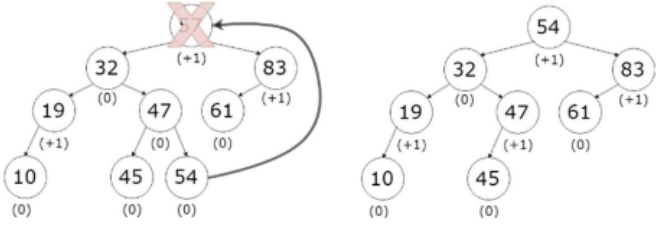
52. **The correct answer is (C).** The process is shown below:



53. **The correct answer is (A).** The process is shown below:



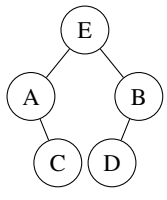
54. **The correct answer is (D).** The process is shown below:



55. **The correct answer is (B).** The maximum number of nodes an AVL tree of height H can have is $2^h - 1$. This is the case where all h levels of the tree are entirely filled, if leaf nodes have height 1. Here, since we have an AVL tree of height 5, the maximum number of nodes this tree can have is $2^5 - 1$, or 31. The minimum number of nodes an AVL tree can have is $1 + N(h-1) + N(h-2)$, where $N(h)$ is the minimum number of nodes an AVL tree of height h can have. Why is this the case? A tree with height h must have at least one child with height $h-1$, so we'll give the other child a height of $h-2$ to minimize the number of nodes we have. Thus, $1 + N(h-1) + N(h-2)$ is the root node + minimum number of nodes in bigger child + minimum number of nodes in smaller child. We know that $N(0)$ is 0 (since empty trees have zero nodes) and $N(1)$ is 1 (since this is a root-only tree, which only has one node). We can solve for $N(5)$ recursively knowing this information:
    - $N(0) = 0$
    - $N(1) = 1$
    - $N(2) = 1 + N(1) + N(0) = 1 + 1 + 0 = 2$
    - $N(3) = 1 + N(2) + N(1) = 1 + 2 + 1 = 4$
    - $N(4) = 1 + N(3) + N(2) = 1 + 4 + 2 = 7$
    - $N(5) = 1 + N(4) + N(3) = 1 + 7 + 4 = 12$

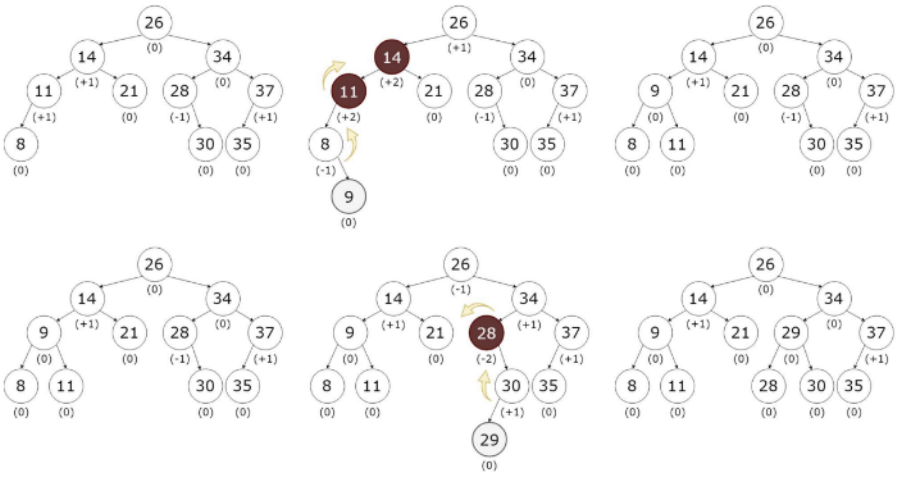
    Since $a = 31$ and $b = 12$, the value of $a - b = 31 - 12 = 19$.

56. **The correct answer is (E).** Starting from the bottom and moving up, the first imbalance that requires a rotation is the -2 balance factor on C with +1 balance factor on D. We rotate right on D and then rotate left on C. After these rotations, E becomes B's left child, and C and D become E's left and right children, respectively. The next imbalance occurs with node B, which has a +2 balance factor. Its child, E, has a balance factor of 0, so we can just perform a right rotation on B. This moves B to the right child of E, D to the left child of B, and E to the right child of A. The last imbalance happens at node A with a balance factor of -2. After rotating left on A, we get the following tree:



57. **The correct answer is (C).** The balance factor of the root node is 0, as shown above (heights of left and right subtrees are the same).

58. **The correct answer is (C).** While you are only given the postorder traversal, the inorder traversal can also be determined from the information given. Since this is an AVL tree, the inorder is simply the elements in sorted order: 8, 11, 14, 21, 26, 28, 30, 34, 35, 37 (recall that AVL is just a balanced binary search tree). Adding 9 and 29 produces the following:



59. **The correct answer is (A).** Since you know the values you want to insert, you can avoid any rotations at all by recursively inserting the median values into the tree. This ensures that you will never insert a value that could potentially cause an imbalance.

60. **The correct answer is (D).** Tree (A) is not a valid AVL tree because 28 is in the right subtree of 30. Tree (B) is not a valid AVL tree because the balance factor of 34 is -2. Tree (C) is not a valid AVL tree because the balance factor of 39 is +2. Only tree (D) is a valid AVL tree with no balance factor whose absolute value is greater than 1.

61. **The correct answer is (B).** This can be best solved by trying out some values, but a total of six insertions are needed at a minimum. One such insertion order that works is 2, 4, 10, 23, 46, and 1. The idea is that you will have to insert one node to be the one at a depth of 6, and one additional node to balance each of the five levels above it.

62. **The correct answer is (D).** Six rotations are needed: twice after inserting 15 (RL11, RR17), twice while inserting 12 (RR14, RL11), and twice after deleting 15 (RL12, RR16).

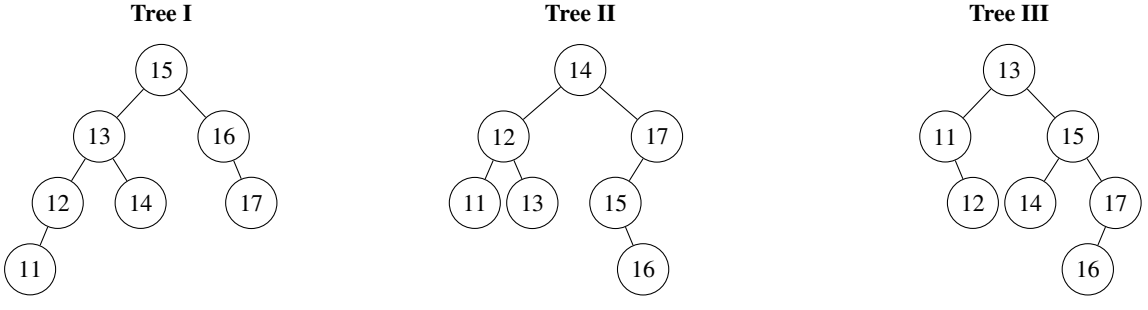
63. **The correct answer is (C).** A maximum of 3 rotations can be induced after deleting 12: a right rotation on 11, a left rotation on 3, and a right rotation on 8.

64. **The correct answer is (B).** See the remark in section 18.7.4, which goes over the formula for calculating the maximum rotations possible from a single deletion.

65. **The correct answer is (D).** See the remark in section 18.7.4. The possible smallest tree of height 7 has a size of 33, so you could have to potentially complete 7 values before you can complete an insertion in this tree.

66. **The correct answer is (C).** I and II are actually not guaranteed to meet the invariants — even if an equal element is always inserted on one side of the tree, a future rotation could potentially bring it into the other side of the tree. Thus, only choice III can be successfully used to satisfy the given invariants (since we do not have to worry about equal elements ending up on the wrong side).

67. **The correct answer is (D).** Only I and III are balanced binary search trees (note that the inorder is just the values in ascending order). The three trees are shown below:



68. **The correct answer is (D).** Ordered lookup containers in the STL require $\Theta(\log(n))$ time lookup, which can only be supported with a self-balancing binary search tree of the provided choices. This can be done with an AVL tree, or any other self-balancing tree that supports this logarithmic complexity (e.g., red-black tree).

69. **The correct answer is (A).** Maps are ordered using `operator<` and `operator==`, so the custom object `T` can be inserted into a `std::map<>` without any issue. However, hash tables require a key to be hashed before insertion, so the absence of a hash function for `T` means that it cannot be used as the key of an `std::unordered_map<>`.

70. **The correct answer is (A).** The first value that is greater than `K` is referenced by `map::upper_bound()`, so you should iterate from `map::begin()` to `map::upper_bound(K)` to print all keys that are less than or equal to `K`.

71. **The correct answer is (B).** Similar to the previous problem, the first value that is greater than `K` is referenced by `map::upper_bound()`, so you should iterate from `map::upper_bound(K)` to `map::end()` to print all keys that are greater than `K`.

72. **The correct answer is (C).** Since calling `.upper_bound(15)` returns an element with a key of 17, that means that 17 is the first key that is strictly greater than 15. As a result, it would be impossible for an element with a key of 16 to exist in the map.

73. **The correct answer is (E).** The iterator `--it` points to the value before the end iterator, or the largest key in the map. Since `first` represents the key, the value of `(--it)->first` is 883.

74. **The correct answer is (D).** To be able to easily access the highest score value, we would want to store the score as the key of an ordered map, which removes options (A) and (B). The value of the largest score would then be accessible as the iterator directly before the end iterator, or option (D).

75. **The correct answer is (B).** The average-case time complexity of inserting an element into an `std::unordered_map<>` is $\Theta(1)$, and the average-case time complexity of inserting an element into a `std::map<>` is $\Theta(\log(n))$. Thus, the overall time complexity of inserting n elements are $\Theta(n)$ for an `std::unordered_map<>`, and $\Theta(n\log(n))$ for a `std::map<>`.

76. **The correct answer is (C).** All of the options are $\Theta(n\log(n))$ in the worst-case, with the exception of the binary search tree. Note that a binary search tree can only guarantee $\Theta(n\log(n))$ if it is balanced, but that is not the case here. As a result, you could end up with an overall time complexity of $\Theta(n^2)$ if the tree is oriented as a stick like tree.

77. One possible solution is as follows. Here, we have to make sure that a node's left and right children are fixed before we remove the node. This is best done by traversing the tree in a postorder fashion. If a node is smaller than the minimum allowable value, we remove the node and set its right child as the new root. On the other hand, if a node is larger than the maximum allowable value, we remove the node and set its left child as the new root.

```
Node* trim_bst(Node* root, int32_t min_val, int32_t max_val) {
  if (!root) {
    return root;
  } // if
  root->left = trim_bst(root->left, min_val, max_val);
  root->right = trim_bst(root->right, min_val, max_val);
  if (root->val < min_val) {
    Node* right_child = root->right;
    delete root;
    return right_child;
  } // if
  if (root->val > max_val) {
    Node* left_child = root->left;
    delete root;
    return left_child;
  } // if
  return root;
} // trim_bst()
```

78. The solution for this problem can be achieved by going down the tree level by level and returning the nodes on each level that is farthest to the right. One method is to use a queue and iterate over the levels of the tree, similar to performing a level order traversal. Once the end of a level is reached, the remaining node is pushed into the result. An implementation of this solution is shown below:

```
std::vector<int32_t> get_rightmost_values(Node* root) {
  if (!root) {
    return {};
  } // if
  std::queue<Node*> bfs;
  std::vector<int32_t> result;
  bfs.push(root);
  while (!bfs.empty()) {
    size_t level_size = bfs.size();
    Node* current = nullptr;
    for (size_t i = 0; i < level_size; ++i) {
      current = bfs.front();
      bfs.pop();
      if (current->left) {
        bfs.push(current->left);
      } // if
      if (current->right) {
        bfs.push(current->right);
      } // if
    } // for i
    result.push_back(current->val);
  } // while
  return result;
} // get_rightmost_values()
```

This problem can be solved recursively as well by making a recursive call into the right child of each level. An implementation of this is shown below: note that an additional call to the left child afterward is needed in the case that the right subtree does not cover the full depth of the entire tree.

```
std::vector<int32_t> get_rightmost_values(Node* root) {
  std::vector<int32_t> rhs_view;
  traverse(root, rhs_view, 0);
  return rhs_view;
} // get_rightmost_values()

void traverse(Node* root, std::vector<int32_t>& rhs_view, int32_t level) {
  if (!root) {
    return;
  } // if
  if (rhs_view.size() == level) {
    rhs_view.push_back(root->val);
  } // if
  traverse(root->right, rhs_view, level + 1);
  traverse(root->left, rhs_view, level + 1);
} // traverse()
```

79. One possible solution is as follows. After removing an edge from a parent to a child, the subtree with the child as the root must be half the sum of the original tree. We can store the sums of every subtree using recursion and a helper function that calculates the sum of each subtree. After doing so, we can check if half the sum was recorded; if it is, we return true.

```
bool equal_sum_subtree(Node* root) {
  std::vector<int32_t> subtree_sums;
  int32_t total_sum = sum(root, subtree_sums);
  // remove the sum of the original tree since we can only split from non-root nodes
  subtree_sums.pop_back();
  if (total_sum % 2) {
    return false;
  } // if
  for (int32_t i : subtree_sums) {
    if (i == total_sum / 2) {
      return true;
    } // if
  } // for i
  return false;
} // equal_sum_subtree()

int32_t sum(Node* root, std::vector<int32_t>& subtree_sums) {
  if (!root) {
    return 0;
  } // if
  int32_t left_sum = sum(root->left, visited);
  int32_t right_sum = sum(root->right, visited);
  int32_t subtree_sum = left_sum + right_sum + root->val;
  subtree_sums.push_back(subtree_sum);
  return subtree_sum;
} // sum()
```

80. Since the given binary search tree is already complete, we can reuse it to build our min-heap. To do so, we can store the inorder traversal of the binary search tree in separate array. Then, we can perform a preorder traversal of the binary search tree and copy over the nodes from the array to the nodes of the tree one by one, from left to right. This works because the elements in the array must be sorted from smallest to largest, and a preorder traversal ensures that parent values are always filled before their children, maintaining the heap invariant. One implementation of this solution is shown below:

```
Node* convert_bst_to_min_heap(Node* bst_root) {
  std::vector<int32_t> vec;
  size_t idx = 0;
  inorder(bst_root, vec);
  write_min_heap(bst_root, vec, &idx);
  return bst_root;
} // convert_bst_to_min_heap()

void inorder(Node* root, std::vector<int32_t>& vec) {
  if (!root) {
    return;
  } // if
  inorder(root->left, vec);
  vec.push_back(root->val);
  inorder(root->right, vec);
} // inorder
```

```
18  void write_min_heap(Node* root, std::vector<int32_t>& vec, size_t* idx) {
19    if (!root) {
20      return;
21    } // if
22    root->val = vec[*idx++];
23    write_min_heap(root->left, vec, idx);
24    write_min_heap(root->right, vec, idx);
25  } // write_min_heap()
```

81. The simplest solution is to use recursion to add the values of nodes together. This is shown below:

```
1   Node* merge_trees(Node* t1, Node* t2) {
2     // If either node is missing, return the value of the other node
3     if (!t1) {
4       return t2;
5     } // if
6     if (!t2) {
7       return t1;
8     } // if
9     t1->val += t2->val;
10    t1->left = merge_trees(t1->left, t2->left);
11    t1->right = merge_trees(t1->right, t2->right);
12    return t1;
13  } // merge_trees()
```

An alternative solution would be to use two stacks to keep track of the nodes, popping and adding values along the way. We start off by pushing the root nodes into the stack. Then, while the stack is not empty, we remove a pair of nodes and add the values to the first tree. If a node does not exist for the first tree, we simply append the node for the second tree (as the absence of a node is equivalent to a value of 0).

```
1   Node* merge_trees(Node* t1, Node* t2) {
2     // If either node is missing, return the value of the other node
3     if (!t1) {
4       return t2;
5     } // if
6     if (!t2) {
7       return t1;
8     } // if
9     std::stack<Node*> s1, s2;
10    s1.push(t1);
11    s2.push(t2);
12    while (!s1.empty()) {
13      Node* curr1 = s1.top();
14      Node* curr2 = s2.top();
15      s1.pop();
16      s2.pop();
17      curr1->val += curr2->val;
18      if (curr1->left && curr2->left) {
19        s1.push(curr1->left);
20        s2.push(curr2->left);
21      } // if
22      else if (!curr1->left) {
23        curr1->left = curr2->left;
24      } // else if
25      if (curr1->right && curr2->right) {
26        s1.push(curr1->right);
27        s2.push(curr2->right);
28      } // if
29      else if (!curr1->right) {
30        curr1->right = curr2->right;
31      } // else if
32    } // while
33  } // merge_trees()
```

82. A realization to be made here is that, since the array is sorted, the element in the middle must be the root of the balanced tree. Furthermore, we can conclude that the middle element of the elements to the left of the root and the middle element of the elements to the right of the root are the left and right children of the root node, respectively. This allows us to use recursion as an elegant solution to this problem, as shown:

```
Node* array_to_bst(const std::vector<int32_t>& nums) {
  return array_to_bst_helper(nums, 0, nums.size());
} // array_to_bst()

Node* array_to_bst_helper(const std::vector<int32_t>& nums, size_t start_idx, size_t end_idx) {
  if (start_idx > end_idx) {
    return nullptr;
  } // if
  size_t middle_idx = (start_idx + end_idx) / 2;
  Node* root = new Node(arr[middle_idx]);
  root->left = array_to_bst_helper(nums, start_idx, middle_idx - 1);
  root->right = array_to_bst_helper(nums, middle_idx + 1, end_idx);
  return root;
} // array_to_bst_helper()
```

83. To solve this problem, we will need to take advantage of two data structures: one to map each stock symbol to price/timestamp information, and another to map timestamp to price. We do not need to know the order of our stock symbols, so an `std::unordered_map<>` is sufficient to store this information. However, our `get_price()` function will need to know about the order of timestamps, so we would need to use an ordered container, such as a `std::map<>`. As a result, one potential solution is to store our data in the form of an unordered map that maps each stock symbol to an ordered map from timestamp to price. Updating would therefore a price to this map, while getting the price would find the largest update timestamp that precedes the given timestamp to query. An implementation of this is shown below:

```
class StockPriceTracker {
private:
  std::unordered_map<std::string, std::map<int32_t, double>> stock_price_map;
public:
  void update(const std::string& symbol, double price, int32_t timestamp) {
    stock_price_map[symbol][timestamp] = price;
  } // update()

  double get_price(const std::string& symbol, int32_t timestamp) {
    // based on assumption that symbol exists and a previous timestamp has been updated before
    auto& timestamp_map = stock_price_map[symbol];
    auto first_larger = timestamp_map.upper_bound(timestamp);
    return (--first_larger)->second;
  } // get_price()
};
```