1. What is the time complexity of searching for a specific element in a balanced Binary Search Tree (BST)?
   1. O(1)
   2. **O(log n)**
   3. O(n)
   4. O(n log n)
2. Which of the following operations can be efficiently performed on a Binary Search Tree?
   1. Sorting elements
   2. Inserting elements at the end
   3. **Finding the median element**
   4. Merging two trees
3. Which type of binary tree ensures that the height difference between the left and right subtrees of any node is at most 1?
   1. **AVL Tree**
   2. Red-Black Tree
   3. Splay Tree
   4. B-Tree
4. In a self-balancing Binary Search Tree, what is the maximum height for a tree with n nodes to guarantee O(log n) search complexity?
   1. n/2
   2. 2 \* log(n)
   3. **log(n)**
   4. n
5. Which rebalancing operation is performed when a new node is inserted into an AVL Tree?
   1. **Rotation**
   2. Height reduction
   3. Transplant
   4. Deletion
6. The inorder traversal of a Binary Search Tree produces elements in:
   1. **Ascending order**
   2. Descending order
   3. Random order
   4. Reverse order
7. Which of the following is NOT a property of a Red-Black Tree?
   1. Every node is either red or black
   2. The root is always black
   3. No two red nodes can be adjacent
   4. **The longest path from the root to a leaf is twice the shortest path**
8. In a Binary Search Tree, which traversal technique is used to visit nodes in ascending order?
   1. **Inorder**
   2. Preorder
   3. Postorder
   4. Level-order
9. The B-Tree is primarily used for:
   1. Maintaining a sorted list
   2. **Searching in external storage**
   3. Implementing priority queues
   4. Representing binary trees
10. Which data structure is used to implement a dynamic set of elements so that insertion, deletion, and search operations can be performed efficiently?
    1. Stack
    2. Queue
    3. Hash Table
    4. **Binary Search Tree**
11. Which of the following is NOT a property of a Splay Tree?
    1. Self-adjusting data structure
    2. Accessed nodes are moved to the root
    3. **Every node has a height factor**
    4. Operations on recently accessed elements are faster
12. Which balancing technique is used in the B+ Tree to maintain the tree structure?
    1. Rotation
    2. **Splitting**
    3. Merging
    4. Deletion
13. What is the time complexity for finding the minimum element in a Binary Search Tree?
    1. O(1)
    2. **O(log n)**
    3. O(n)
    4. O(n log n)
14. A Binary Search Tree is converted into a Doubly Linked List using which traversal?
    1. **Inorder**
    2. Preorder
    3. Postorder
    4. Level-order
15. Which type of Binary Search Tree allows duplicate elements?
    1. AVL Tree
    2. Red-Black Tree
    3. Splay Tree
    4. **Multiset**
16. In a Binary Search Tree, the predecessor of a node is:
    1. The node with the smallest key greater than the given node
    2. **The node with the largest key smaller than the given node**
    3. The left child of the given node
    4. The right child of the given node
17. What is the primary advantage of using a Red-Black Tree over an AVL Tree?
    1. **Faster insertion and deletion operations**
    2. Smaller memory footprint
    3. Better support for multithreading
    4. Simpler implementation
18. Which of the following operations does NOT require rotation in an AVL Tree?
    1. Left-Left Imbalance
    2. **Left-Right Imbalance**
    3. Right-Left Imbalance
    4. Right-Right Imbalance
19. A self-balancing Binary Search Tree can be used to implement which data structure with efficient time complexity?
    1. Stack
    2. Queue
    3. **Priority Queue**
    4. Hash Table
20. Which of the following statements is true regarding AVL trees?
    1. AVL trees are unbalanced binary search trees.
    2. **AVL trees are a type of self-balancing binary search trees.**
    3. AVL trees are best suited for small datasets.
    4. AVL trees have O(n) worst-case complexity for insertion and deletion.
21. What will be the output of the following C++ program?

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

Node\* insert(Node\* root, int key) {

// AVL tree insertion code

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

std::cout << root->key;

return 0;

}

* 1. 10
  2. 20
  3. **30**
  4. 40

1. What is the height difference allowed between the left and right subtrees of any node in an AVL tree?
   1. 0
   2. **1**
   3. 2
   4. 3
2. What will be the output of the following C++ program?

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

Node\* insert(Node\* root, int key) {

// AVL tree insertion code

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 15);

root = insert(root, 5);

std::cout << root->left->key;

return 0;

}

* 1. 5
  2. 10
  3. **15**
  4. 20

1. An AVL tree with n nodes has a height of \_\_\_\_\_\_ at most.
2. n - 1
3. n
4. log(n)
5. **2 \* log(n)**
6. What will be the output of the following C++ program?

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

Node\* insert(Node\* root, int key) {

// AVL tree insertion code

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 12);

std::cout << root->right->left->key;

return 0;

}

* 1. 5
  2. 10
  3. **12**
  4. 15

1. In an AVL tree, what is the maximum number of rotations required during an insertion to maintain the balance factor?
2. 1
3. **2**
4. 3
5. 4
6. What will be the output of the following C++ program?

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

Node\* insert(Node\* root, int key) {

// AVL tree insertion code

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 15);

root = insert(root, 30);

root = insert(root, 25);

std::cout << root->right->right->key;

return 0;

}

* 1. 15
  2. 20
  3. **25**
  4. 30

1. AVL tree is an example of a \_\_\_\_\_.
2. Search tree
3. Traversal tree
4. Priority tree
5. **Balanced tree**
6. What will be the output of the following C++ program?

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

Node\* insert(Node\* root, int key) {

// AVL tree insertion code

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 5);

root = insert(root, 15);

std::cout << root->left->right->key;

return 0;

}

* 1. 5
  2. 10
  3. **15**
  4. 20

1. What is the time complexity of searching an element in an AVL tree with n nodes?
   1. O(1)
   2. **O(log n)**
   3. O(n)
   4. O(n log n)
2. What will be the output of the following C++ program?

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

};

int main() {

Node\* root = nullptr;

root = new Node{10, nullptr, nullptr};

root->left = new Node{5, nullptr, nullptr};

root->right = new Node{15, nullptr, nullptr};

cout << "Root key: " << root->key << endl;

cout << "Left child key: " << root->left->key << endl;

cout << "Right child key: " << root->right->key << endl;

return 0;

}

* 1. **Root key: 10, Left child key: 5, Right child key: 15**
  2. Root key: 5, Left child key: 10, Right child key: 15
  3. Root key: 15, Left child key: 10, Right child key: 5
  4. Root key: 10, Left child key: 15, Right child key: 5

1. For a binary search tree to be an AVL tree, the balance factor of each node must be in the range of \_\_\_\_\_\_.
   1. **-1 to 1**
   2. -2 to 2
   3. 0 to 1
   4. -1 to 0
2. What will be the output of the following C++ program?

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

};

int main() {

Node\* root = nullptr;

root = new Node{10, nullptr, nullptr};

root->left = new Node{5, nullptr, nullptr};

root->right = new Node{15, nullptr, nullptr};

root->left->left = new Node{3, nullptr, nullptr};

root->left->right = new Node{8, nullptr, nullptr};

cout << "Root key: " << root->key << endl;

cout << "Left child key: " << root->left->key << endl;

cout << "Right child key: " << root->right->key << endl;

cout << "Left grandchild key: " << root->left->left->key << endl;

cout << "Right grandchild key: " << root->left->right->key << endl;

return 0;

}

* 1. **Root key: 10, Left child key: 5, Right child key: 15, Left grandchild key: 3, Right grandchild key: 8**
  2. Root key: 5, Left child key: 10, Right child key: 15, Left grandchild key: 3, Right grandchild key: 8
  3. Root key: 15, Left child key: 10, Right child key: 5, Left grandchild key: 8, Right grandchild key: 3
  4. Root key: 10, Left child key: 15, Right child key: 5, Left grandchild key: 3, Right grandchild key: 8

1. Which of the following operations can unbalance an AVL tree?
2. Insertion
3. Deletion
4. **Both insertion and deletion**
5. Searching
6. What will be the output of the following C++ program?

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

};

void insert(Node\*& root, int key) {

if (root == nullptr) {

root = new Node{key, nullptr, nullptr};

return;

}

if (key < root->key)

insert(root->left, key);

else

insert(root->right, key);

}

int main() {

Node\* root = nullptr;

insert(root, 12);

insert(root, 7);

insert(root, 17);

cout << "Root key: " << root->key << endl;

cout << "Left child key: " << root->left->key << endl;

cout << "Right child key: " << root->right->key << endl;

return 0;

}

* 1. **Root key: 12, Left child key: 7, Right child key: 17**
  2. Root key: 7, Left child key: 17, Right child key: 12
  3. Root key: 12, Left child key: 17, Right child key: 7
  4. Root key: 17, Left child key: 12, Right child key: 7

1. What is the worst-case time complexity for inserting a node into an AVL tree with n nodes?
2. **O(log n)**
3. O(n)
4. O(n log n)
5. O(1)
6. What will be the output of the following C++ program?

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

};

void insert(Node\*& root, int key) {

if (root == nullptr) {

root = new Node{key, nullptr, nullptr};

return;

}

if (key < root->key)

insert(root->left, key);

else

insert(root->right, key);

}

int getHeight(Node\* root) {

if (root == nullptr)

return -1;

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

return max(leftHeight, rightHeight) + 1;

}

int main() {

Node\* root = nullptr;

insert(root, 10);

insert(root, 5);

insert(root, 15);

insert(root, 3);

insert(root, 8);

cout << "Height of the AVL tree: " << getHeight(root) << endl;

return 0;

}

* 1. Height of the AVL tree: 2
  2. **Height of the AVL tree: 3**
  3. Height of the AVL tree: 4
  4. Height of the AVL tree: 5

1. An AVL tree becomes a degenerate tree if \_\_\_\_\_\_.
2. All nodes have the same value
3. **The height difference between left and right subtrees is greater than 1**
4. It has only one node
5. It has no nodes
6. What will be the output of the following C++ code snippet?

bool isAVL(Node\* root) {

if (root == nullptr)

return true;

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

int balanceFactor = abs(leftHeight - rightHeight);

return (balanceFactor <= 1) && isAVL(root->left) && isAVL(root->right);

}

int main() {

Node\* root = nullptr;

insert(root, 10);

insert(root, 5);

insert(root, 15);

insert(root, 3);

insert(root, 8);

if (isAVL(root))

cout << "The tree is an AVL tree." << endl;

else

cout << "The tree is not an AVL tree." << endl;

return 0;

}

* 1. **The tree is an AVL tree.**
  2. The tree is not an AVL tree.
  3. The program will not compile due to an error.
  4. The program will run into an infinite loop.

40) bool isBST(Node\* root, int minValue, int maxValue) {

if (root == NULL)

return true;

if (root->data <= minValue || root->data >= maxValue)

return false;

return isBST(root->left, minValue, root->data) &&

isBST(root->right, root->data, maxValue);

}

int main() {

Node\* root = newNode(5);

root->left = newNode(3);

root->right = newNode(7);

root->left->left = newNode(2);

root->left->right = newNode(6);

cout << (isBST(root, INT\_MIN, INT\_MAX) ? "Yes" : "No") << endl;

return 0;

}

a) Yes

**b) No**

c) Compilation error

d) Runtime error

41) What is the role of give snippet?

Node\* fun(Node\* root, int key) {

if (root == NULL || root->data == key)

return root;

if (key < root->data)

return fun(root->left, key);

return fun(root->right, key);

}

a) Print inorder

b) print postorder

**c) search key**

d) fin maximum key