Congratulations! You passed!

Grade received 100% **Latest Submission Grade** 100% **To pass** 80% or higher

Go to next item

1. For a linked structure of edges and nodes to be a tree, which of the following is not required to be true? **1 / 1 point**

- (●) Every node has zero, one or two children.
- ( ) Every node has a parent except for one single root node.
- ( ) Every node is connected to every other node by some path of edges.
- ( ) If any two nodes are connected, they are connected by only one path of unique nodes and edges.

Correct
This is true of a **binary** tree, but not of a tree in general.

2. Which data structure below supports the fastest run time for finding an item in a sorted list of items? **1 / 1 point**

- (●) Array
- ( ) Linked List
- ( ) Binary Search Tree
- ( ) All of these data structures have the same run time complexity for finding an item in a sorted list of items.

Correct
A sorted array has a worst-case run time of O(lg n) which is faster than the other options.

3. What is the height of the binary search tree created by inserting the following values one at a time in the following order of insertion: **1 2 3 4 5 6 7** ? **1 / 1 point**

- ( ) 2
- ( ) 3
- (●) 6
- ( ) 7

Correct
The height of a tree is the number of edges from the root to its fathest descendant. Since these values are inserted one at a time and are already in sorted order, the binary search tree ends up resembling a linked list from the root following the right child link through the list, with six edges.

4. Which of the following is NOT true of a perfect binary search tree of a list of n ordered items? **1 / 1 point**

- (●) The worst-case run time to find an item is O(n).
- ( ) Every non-leaf node has two children.
- ( ) If the height of the tree is h, then n = 2^(h+1) - 1.
- ( ) All of the leaf nodes are at the same level.

Correct
The worst-case run time to find an item in a perfect binary search tree is O(lg n), because its height h = lg(n+1) - 1.

5. Which of the following is NOT a full binary tree? **1 / 1 point**

- (●) The binary tree consisting of the subtree of ancestors of any node in any perfect binary tree.
- ( ) A single node.
- ( ) The binary search tree created by inserting the following values one at a time: **4 2 3 5 1**.
- ( ) A perfect binary tree.

Correct
Every node in a full binary tree has zero or two children, whereas the non-leaf nodes of the subtree of ancestors would consist of nodes each having only a single child.

6. Which of the following is not a true statement about a complete binary tree? **1 / 1 point**

- ( ) The height of a complete binary tree of n nodes is floor(lg n).
- (●) Any tree that contains a node with a single child is not a complete binary tree.
- ( ) The worst-case run time for finding an object in a complete binary search tree of an ordered list of n items is O(lg n).
- ( ) No node in a complete binary tree has only a right child.

Correct
A complete binary tree can have one node with a single child.

7. Which one of the following functions outputs the keys of a binary search tree in item order when the root node is passed to it as its parameter. **1 / 1 point**

( )
```
1  void print(TreeNode *node){
2      if (!node) return;
3      std::cout << node->key << " ";
4      print(node->left);
5      print(node->right);
6  }
```

(●)
```
1  void print(TreeNode *node){
2      if (!node) return;
3      print(node->left);
4      std::cout << node->key << " ";
5      print(node->right);
6  }
```

( )
```
1  void print(TreeNode *node){
2      if (!node) return;
3      print(node->left);
4      print(node->right);
5      std::cout << node->key << " ";
6  }
```

- ○ None of these outputs all of the keys of the binary search tree in item order.

✓ **Correct**
This is an in-order traversal that prints the values of the descendants of the node's left child, then the node's value, then the values of the descendants of its right child.

8. Consider the binary search tree built by inserting the following sequence of integers, one at a time: **5, 4, 7, 9, 8, 6, 2, 3, 1** **1 / 1 point**

Which method below will properly remove node **4** from the binary search tree?

- ○ Find the in order predecessor (IOP) of node **4**, which is node **3**. Remove node **3** from the tree by setting the right pointer of its parent (node **2**) to **nullptr**. Then copy the key and any data from node **3** to node **4**, turning node **4** into a new node **3**, and delete the old node **3**.
- ◉ Set the *left* pointer of node **5** to point to the node pointed to by the *left* pointer of node **4**, and then delete node **4**.
- ○ Find the in order predecessor (IOP) of node **4**, which is node **3**. Remove node **3** from the tree by setting the right pointer of its parent (node **2**) to point to the node pointed to by the left pointer of node **3**. Then copy the key and any data from node **3** to node **4**, turning node **4** into a new node **3**, and delete the old node **3**.
- ○ Set the *left* pointer of node **5** to **nullptr**, and then delete node **4**.

✓ **Correct**
This is the correct way to remove node **4** from the binary search tree because node 4 has only one child, its left child.

9. Suppose that we have numbers between 1 and 1000 in a binary search tree and we want to search for the number **363**. Which of the following sequences can **not** be the sequence of nodes visited in the search? **1 / 1 point**

- ◉ **925, 202, 911, 240, 912, 245, 363**
- ○ **2, 399, 387, 219, 266, 382, 381, 278, 363**
- ○ **924, 220, 911, 244, 898, 258, 362, 363**
- ○ **2, 252, 401, 398, 330, 344, 397, 363**

✓ **Correct**
- 925 is the root.
- 202 is 925's left child.
- 911 is 202's right child.
- 240 is 911's left child.
- 912 cannot be any child of 240 because that would place it in the left subtree of 911, but 912 is not less than 911.

10. Given any binary tree with 128 nodes where each node has a *left* pointer and a *right* pointer, how many of these pointers are set to **nullptr**? **1 / 1 point**

129

✓ **Correct**
The number of node pointers to **nullptr** is equal to the number of nodes plus one. For a tree with just one node (the root), the number of pointers to **nullptr** is two. Adding a child node to any tree replaces one pointer to **nullptr** with a pointer to the new node, which then has two pointers to **nullptr**. Hence, for every new node, there is a net increase of one pointer to **nullptr**.