Congratulations! You passed!

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

Go to next item

1. Assume you are storing a complete binary tree as a contiguous list of keys in an array such that the root's key is stored at location 1 of the array, the keys from all of the nodes at the next level of the tree are stored in left-to-right order in subsequent locations in the array, then similarly for all of the nodes of each subsequent level. **1 / 1 point**

At what array location would the key stored in the node that is the left child of the right child of the root?

6

Correct
Correct. The root is at position 1. Its right child is at 2*(1) + 1 = 3 and position 3's left child is at 2*(3) = 6.

2. When using an array to store a complete tree, why is the root node stored at index 1 instead of at the front of the array at index 0? **1 / 1 point**

- We use index zero as a guard to prevent overstepping the root when propagating up the tree from its leaf nodes, which would cause a memory access fault.
- This makes the math for finding children and parents simpler to compute and to explain.
- We avoid using index 0 to avoid confusion with the value of 0 (**nullptr**) that we normally store in the child pointer of a node to indicate that child does not exist.
- Array index 0 is used to store the number of nodes in the complete tree stored in the array.

Correct
Yep. It is worth wasting one memory location to make programming and documentation simpler and less bug-prone. In particular, this lets us find the parent of a given node simply by using integer division by 2.

3. When is a binary tree a min-heap? **1 / 1 point**

- When every node's value is less than its parent's value.
- When every node's value is greater than its parent's value.
- When the leaf nodes represent the smallest values in the tree, and every leaf node is smaller than the root.
- When every node's value lies between the maximum value of its left child's subtree and the minimum value of its right child's subtree.

Correct
A non-empty binary tree is a min-heap if the root is less than either or both of its children (if any), and the subtrees of its children are min-heaps. This is equivalent to the definition from the video lesson and equivalent to the answer.

4. How should one insert a new value into a heap to most efficiently maintain a balanced tree? **1 / 1 point**

- Maintain the heap as a complete tree and insert a new value at the one new node position that keeps the tree as a complete tree. Then continually exchange the new value with the value of its parent until the new value is in node where it is greater than the value of its parent.

- Maintain the heap as an AVL tree. Walk down the tree from the root exploring the left children first, then the right children, until a node is found that is greater than the new value. Insert a new node with the new value at that position and make the previous node the left child of that new node. Then call the appropriate rotation routine to rebalance the tree if the height balance factor magnitude of the new node or its parent reaches two.
- Maintain the heap as an array. Walk down the tree from the root at position 1 in the array, exploring the left children first, then the right children, until a node position is found whose value is greater than the new value. Copy the value at that position and all subsequent positions in the array to one greater position in the array, and store the new value at that position.
- Maintain the heap as a balanced binary search tree. Walk down the tree from the root exploring the left children first, then the right children, until a node is found that is greater than the new value. Insert a new node with the new value at that position and make the previous node the left child of that new node. Rebalance the tree if the height balance factor magnitude of the new node or its parent exceeds one.

Correct
This is the best way to maintain a balanced heap, but requires back propagation to ensure the min-heap properly remains valid.

5. The removeMin operation removes the root of a min-heap tree. Which of the following implements removeMin efficiently while maintaining a balanced min-heap tree. **1 / 1 point**

- Increment the address used to indicate the base location of the array storing the complete binary tree.
- Delete the root and if the root has two children, then merge its left subtree with its right subtree by inserting each right subtree node value into the left subtree. Then delete the right subtree.
- Replace the root value with the value of the last leaf (rightmost node at the bottom level) of a complete binary tree, and delete the last leaf. Then repeatedly exchange this last-leaf value with the smaller of the values of its node's children until this last-leaf value is smaller than the values of its node's children, if any.
- Set the root value to +infinity. If the left child is smaller than the right child, perform a Right-Rotation, otherwise perform a Left-Rotation. Repeat this process at the new infinity-node location until the infinity node is a leaf, then remove and delete it.

Correct
This last-leaf node corresponds to the end of the array so that there are never any missing values in the middle of the array.

6. How many nodes of a complete binary tree are leaf nodes? **1 / 1 point**

- About half.
- Unknown. Could be one. Could be all.
- About the square root.
- About a fourth.

Correct
Let n be the number of nodes in the complete binary tree. If the complete binary tree is also a perfect tree, then all of the leaf nodes are at the bottom level, and there are exactly n/2 + 1/2 of them. (n is odd for a perfect binary tree.)

Now consider deleting these nodes one at a time in an order that keeps the tree complete. The first node you delete is the right child of a parent so it decreases n by one and decreases the number of leaf cells by one. The second node you delete will be that parent's left child, turning the parent into a new leaf node, so decreasing n by one but not decreasing the number of leaf nodes. As you continue removing leaf nodes from right to left on that bottom level, you are reducing n by one but the number of leaf nodes by an average of 1/2 (by one for every right child and by zero for every left child).

Hence about half of a perfect binary tree's nodes are leaf nodes, and this continues as you delete any or all of the nodes in the bottom level in a right-to-left order that keeps the binary tree complete.

7. Recall that the heapifyDown procedure takes a node index whose children (if any) are heaps, but the value of the node might not satisfy the heap property compared to its children's values. This procedure then swaps the node's value with the smallest child value larger than it (if any), and then calls itself on that smallest child node it just swapped values with to further propagate that value down the heap until it finds a valid location for it.

```
1   template <class T>
2   void Heap<T>::_heapifyDown(int index) {
3       if (!_isLeaf(index)) {
4           T minChildIndex = _minChild(index);
5           if (item_[index] > item_[minChildIndex] ) {
6               std::swap( imem_[index], item_[minChildI]);
7               _heapifyDown(minChildIndex);
8           }
9       }
10  }
```

When you call heapifyDown on a given node, what is the maximum number of times heapifyDown is called (including that first call) to find a valid location for the initial value of that node?

- ○ heapifyDown is only called once since its children are already heaps.
- ○ The maximum number of times heapifyDown is called is the number of nodes in its subtree.
- ◉ The maximum number of times heapifyDown is called is one plus the height of the node.
- ○ The maximum number of times heapifyDown is called is the number of non-leaf nodes in its subtree.

✓ **Correct**
heapifyDown is recursive, but it is only called on one of its children, so it walks down only one chain of descendants, not all of its descendants.

8. What is the run-time algorithmic complexity of calling heapifyDown on every non-leaf node in a complete tree of n nodes? **1 / 1 point**

- ◉ O(n)
- ○ O(n lg n)
- ○ O(1)
- ○ O(n^2)

✓ **Correct**
The run-time of calling heapifyDown on a node is proportional to the height of the node. About half of the nodes are leaf nodes, about a quarter have height 1, about an eighth have height 2, about a sixteenth have height 3, and so on. This summation of heights converges to n, the number of nodes in the tree. Hence running heapifyDown on every non-leaf node has a run-time complexity of O(n).

9. Which of the following is the fastest way to build a heap of n items? **1 / 1 point**

- ◉ Create a complete tree of the items in any order, then call heapifyDown on every non-leaf node from the bottom of the tree up to the root.
- ○ Create a complete tree of the items in any order, then call heapifyUp on every node in the tree from the root down to the leaf nodes.
- ○ Create a heap with a single node holding the first item. Then insert the remaining n-1 items into the heap.
- ○ Create a complete tree of the items in any order. If this is not a heap, then swap the items between two randomly chosen nodes and check again. Repeat the random swapping until you get a heap.

✓ **Correct**
As we showed before, this runs in O(n) time.

10. Which of the following is NOT a step of the heap sort algorithm? **1 / 1 point**

- (●) Insert the next item into the current heap.
- ( ) Load the data in any order into a complete tree.
- ( ) Remove the root node.
- ( ) Run heapifyDown on every non-leaf node.

Correct
The heap sort algorithm, as presented in the video lessons, does not rely on the insertion of any item into an existing heap.