**Homework 02 -- Questions 1 - 5**

1. What is binary tree and binary search tree? Discuss their similarities and differences.

A binary tree is a tree that satisfies the following two properties: Each internal node of the tree (nodes with at least one child) has at most two children, and the children of a node are an ordered pair, where one is the left child and the other is the right child (in a proper binary tree, each internal node has exactly two children). A binary search tree has these properties as well, but it has an additional property. In a binary search tree, the value stored at any node is always greater than the value of its left child and is always less than the value of its right child. Thus, it is possible for the left child of a node containing 8 to contain 9 in a binary tree, but this is not possible in a binary search tree.

2. How can we traverse the nodes of a tree? In which scenario we should use which traversal mechanism?

We can traverse the nodes of a tree in three different ways: preorder, postorder, and inorder traversal. In preorder traversal, each node of the tree is visited and then its children are visited by recursively calling the preorder traversal for each child of the parent node. This method of traversal is useful for printing a structured document, where each piece of information should be printed within its respective category or subcategory.

In postorder traversal, first each child of a node is visited by recursively calling the postorder traversal method, and then the parent node is visited. In essence, descendant nodes are always visited before their parents, and the root of the tree will always be the last node visited. This method is useful for any case where we want to retrieve the external items before the internal items, since it always visits the left subtree of a node, then the right subtree of the node, and finally the node itself. Computing space used by files in a directory, deleting a tree, or getting the postfix expression of an arithmetic tree are all cases where we would use postorder traversal.

In inorder traversal, each node is visited after its left subtree and before its right subtree. That is, if a left subtree exists, the inorder method is recursively called on the left child, then the node itself is visited, and finally, if a right subtree exists, the inorder method is recursively called on the right child. Because inorder traversal relies on defined left and right subtrees, it is only useable for binary trees. Inorder traversal is most useful for binary search trees, as it retrieves all of the nodes in increasing order.

3. What are the advantages of binary search over a linear search?

A binary search’s advantage over a linear search comes from its runtime advantage. In the worst case scenario, a linear search has 0(n) runtime complexity, whereas a binary search has 0(log(n)) runtime complexity. This is because a binary search eliminates half of the searchable elements for every operation. The search method visits the middle element and compares it to the target. If it is greater than the target, the right half of the remaining elements is removed, and vice versa if it is less than the target. This makes the binary search much faster than the linear search, although it only works on sorted data.

4. What is a priority-queue? Discuss some use cases of priority-queue.

A priority-queue is list which stores a collection of entries. An entry is defined as a pair of elements where the first element is a key and the second element is a value. Each entry’s level of priority in the priority-queue is determined by its key, and thus the keys must be comparable objects (the item with the “lowest” key is considered highest priority). Depending on the implementation of the priority-queue, entries can either be sorted when they are added to the collection so that the first value in the collection always has the lowest key (insertion sort), or the entry with the lowest key can be found when an item needs to be removed, in which case the list remains unsorted. Alternatively, the priority-queue can be implemented using a heap, in which case the lowest key is stored at the root of the heap.

Priority-queues are useful for any situation in which the elements stored in a list need to be removed in a particular order based on the precedence of each element. An auction, for example, needs to keep track of each bidder while ensuring that the top bidder has priority for receiving the item. Another example where a priority-queue might be useful is a guest list where there are VIP slots. The priority-queue would ensure that VIPs have precedence over non-VIPs.

5. What is a heap? What is the advantage of the heap over a stack? What is the time complexity to get the minimum item from min-heap?

A heap is a binary tree with nodes storing keys (comparable values) such that every internal node that is not the root always has a key that is greater in value than that of its parent (meaning the child node is lower priority). It is also a complete binary tree, meaning that at the depth (h – 1) = i in the tree (the depth just before the furthest depth) there are 2i nodes, and of the nodes at depth i, if any are external, they are to the right of the internal nodes. Essentially, this means that a heap is filled out row by row from left to right when key are inserted, and the last node is the rightmost node at the height of the tree (maximum depth).

The heap has several advantages over the stack. The stack, being linear, does not benefit from the hierarchical structure of a tree like the heap does. Thus a stack cannot achieve 0(log(n)) runtime complexity for certain operations involved in sorting and moving values. The heap is also a more flexible data structure than the stack, as stack memory needs to be stored in a continuous block of memory space, while the nodes of a heap can be stored anywhere in memory. This means that while a stack can run into a stack overflow error, a heap’s only constraint is the amount of memory in the computer itself.

The removeMin() method for a heap runs in 0(log(n)) time complexity. The first step of this method is to swap the positions of the root node with the last node in the tree. Since the heap is implemented using an array, this is an 0(1) operation. Then, the method removes the last node from the heap, which is also an 0(1) operation. Finally, the heap must restore its order using the downheap support method, which continually swaps nodes downward in the tree as long as the key of the current node is greater than the key of its child. This final operation has a runtime of 0(log(n)) because the height of the heap is equal to log(n) (n is the number of nodes). Thus, the entire method has 0(log(n)) runtime complexity.

**Programming Problems (6 – 10)**

Question 6:

Discussion:

Verification:

Question 7:

Discussion:

Verification:

Question 8:

Discussion:

Verification:

Question 9:

Discussion:

Verification:

Question 10:

Discussion:

Verification: