1. Describe a modification to the binary search-tree data structure that would support two index-based operations for a sorted map in $O(h)$ time, where $h$ is the height of the tree.

   (a) atIndex($i$): Return the position $p$ of the entry at index $i$ of a sorted map.
   (b) indexOf($p$): Return the index $i$ of the entry at position $p$ of a sorted map.

**Solution.**

The key idea is to store with a node the number of internal nodes in its left subtree, accessed through a position function $num\text{Left}()$. This can clearly be updated in $O(h)$ time upon insert and removals. As these algorithms are internal to the sorted map, let the binary tree be named $T$, i.e., we have access to the Binary Tree ADT functionality through $T$.

**Algorithms.**

The algorithm for atIndex($i$) is shown in Algorithm 1. Essentially, we walk down the tree. If our current index is greater than our search index, we move left. If our current index is less than our target, we move right and modify the search index. We repeat until we find our target index.

```
Algorithm 1 atIndex($i$)
Input: Index $i$
Output: Position of the entry at $i$ of the sorted map
1: Position $p$ ← $T$.root()
2: while $i \neq p$.numLeft() do
3:   if $i < p$.numLeft() then
4:     $p$ ← $T$.left($p$)
5:   else { $i > p$.numLeft() }
6:     $p$ ← $T$.right($p$)
7:     $i$ ← $i - 1 - p$.numLeft()
8: return $p$
```

The algorithm for indexOf($p$) is shown in Algorithm 2. Essentially, we walk up the tree until we reach the root. At each step, if we are a right child of our parent node then we adjust our current index. Otherwise, we do nothing. The function isRightChild determines if the second parameter is a child node of the first parameter. This is a constant time operation.
Algorithm 2 `indexOf(p)`

**Input:** Position `p`

**Output:** Index of the entry at position `p` of the sorted map

1. Index `i ← p.numLeft()`
2. while `p ≠ ∅` do
3.   Position `par ← T.parent(p)`
4.   if `T.isRightChild(par, p)` then
5.     `i ← i + 1 + par.numLeft()`
6.   `p ← par`
7. return `i`

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in $O(h)$ time.

*Proof.* The algorithm performs a walk down a single path of the tree. At each node there is a constant number of operations of integer manipulation and ADT access (children information) which all operate in $O(1)$ time. Since the path length is bound by the height of the tree this algorithm runs in $O(h)$ time.

**Theorem 2.** Algorithm 2 runs in $O(h)$ time.

*Proof.* The algorithm performs a walk up a single path of the tree. At each node there is a constant number of operations of integer manipulation and ADT access (parent information) which all operate in $O(1)$ time. Since the path length is bound by the height of the tree this algorithm runs in $O(h)$ time.

**Memory Complexity.**

**Theorem 3.** Algorithm 1 uses $O(1)$ additional memory.

*Proof.* A constant number of temporary variables are used to iterate through the tree. Thus, there is $O(1)$ additional memory is required.

**Theorem 4.** Algorithm 2 uses $O(1)$ additional memory.

*Proof.* A constant number of temporary variables are used to iterate through the tree. Thus, there is $O(1)$ additional memory is required.
2. Write and analyze an algorithm to find and construct a list of all elements within a range of keys \((k_1, k_2)\) for a binary tree \(T\). The algorithm must run in \(O(s + h)\) time, where \(s\) is the number of elements in the range and \(h\) is the height of the tree.

**Algorithm.**

The algorithm is described in Algorithm 3. Essentially, begin by searching for the smallest node greater than or equal to \(k_1\), this is called \(\text{ceilingNode}\). Then, from this node, walk through the tree in an inorder fashion constructing the list of keys between \(k_1\) and \(k_2\). Return the list of values.

**Algorithm 3 Elements in Range**

| Input: | Keys \(k_1\) and \(k_2\) |
| Output: | List of elements whose keys are in the range provided |

1. List \(l \leftarrow \emptyset\)  
2. Node \(n \leftarrow \text{ceilingNode}(k_1)\)  
3. while \(n.\text{getKey}() \leq k_2\) do  
4. \(l.\text{add}(n.\text{getValue}())\)  
5. \(n \leftarrow \text{inOrderNext}(n)\)  
6. return \(l\)

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in \(O(s + h)\) time, where \(s\) is the number of elements in the range \([k_1, k_2]\) and \(h\) is the height of the tree.

**Proof.** The search for the ceiling node takes \(O(h)\) time with a standard tree search algorithm. Then we do an inorder walk through \(s\) tree nodes. The function \(\text{inOrderNext}\) will run in amortized constant time over the \(s\) nodes. Thus, in total the algorithm takes \(O(s + h)\) time. \(\square\)

**Memory Complexity.**

**Theorem 6.** Algorithm 3 uses \(O(s)\) additional memory.

**Proof.** The ceiling search function and inorder next functions can both be implemented iteratively and thus take \(O(1)\) extra memory. Constructing a list of \(s\) items is the only extra memory. Thus, the algorithm uses \(O(s)\) extra memory. \(\square\)
3. **Bonus.** Show that any $n$-node binary tree can be converted to any other $n$-node binary tree using $O(n)$ rotations. Provide a statement and proof by construction.

**Solution.**

**Theorem 7.** *Any $n$-node binary tree can be converted to any other $n$-node binary tree using $O(n)$ rotation.*

*Proof.* We break this into two parts. First, we need to show that any binary tree can be converted to an intermediate structure. If this is possible, then we know, that we can convert between any two trees, which we show in the second part.

We can convert any $n$-node binary tree into an $n$-node left-linear tree. Start at the left-most node. While the node has a right-child, rotate it, otherwise go to the nodes parent. Repeat until you finish rotating the new root. By rotating, we mean that convert a node that has a left and right child into the structure of the left being a child of the node and the node being a child of the right (a line). This will work on any structure of a binary tree.

Second, take two $n$-node binary trees, $A$ and $B$. Convert them both to $n$-node left-linear trees and record the rotations that occurred. Then to the $n$-node left-linear trees apply the reverse of the recorded rotations. This will convert the structure of one onto the other. 

\qed