# CS-600-A Homework 2

Zijing Huang 10414952

**C-2.8 Describe the structure and pseudocode for an array-based implementation of an index-based list that achieves O(1) time for insertions and removals at index 0, as well as insertions and removals at the end of the list. Your implementation should also provide for a constant-time get method.**

Suppose the array has enough place and it never overflow. I will set up two pointers. One point to the first place of array called fp, the other point to the last place called lp.

When the array is empty, they are both in the 0.

When we do an index 0 insertion, if fp is a positive we insert the element to array[fp]. If fp is a non-zero negative, we insert the element to array[array.length + fp]. And then fp = fp - 1 and size = size + 1. If we remove from index 0, if fp is a positive, we simplely delete the element at fp. If fp is negative, we delete the element at array.length + fp. And then fp ++ and size = size – 1.

The same idea, if we want to insert an element to the end of the list, array[++lp] = element and size = size - 1. If we want to remove the element to the end of the list, lp = lp – 1, delete the element at lp and size = size – 1.

Get method (get(i)) can be written as following: if i + fp is a negative, then return array[array.length + i + fp - 1], else return array[i + fp].

**C-2.20 Let T be a binary tree with n nodes. Give a linear-time method that uses the methods of the BinaryTree interface to traverse the nodes of T by increasing values of the level numbering function p given in Exercise R-2.8. This traversal is known as the *level order traversal*.**

**Algorithm**: LevelOrderTraversal(BinaryTree T):

Input: A Binary Tree with a level numbering function

Output: a queue of nodes with their level numbers

q = new Queue()

q = enqueuer(T.root)

While q is not a null do

v <- q.dequeue()

//do something with v.value

q.enqueue(v.leftnode)

q.enqueue(v.rightnode)

Because we access each node the one time, so it is in a linear time.

**A-2.2 Suppose you work for a company, iPuritan.com, that has strict rules for when two employees, x and y, may date one another, requiring approval from their lowest- level common supervisor. The employees at iPuritan.com are organized in a tree, T, such that each node in T corresponds to an employee and each employee, z, is considered a supervisor for all of the employees in the subtree of T rooted at z (including z itself). The lowest-level common supervisor for x and y is the employee lowest in the organizational chart, T, that is a supervisor for both x and y. Thus, to find a lowest-level common supervisor for the two employees, x and y, you need to find the *lowest common ancestor* (LCA) between the two nodes for x and y, which is the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself). Given the nodes corresponding to the two employees x and y, describe an efficient algorithm for finding the supervisor who may approve whether x and y may date each other, that is, the LCA of x and y in T. What is the running time of your method?**

**Algorithm**: LowestCommonAncestor(BinaryTree x, BinaryTree y, BinaryTree root):

Input: A BinaryTree node root T, a node x and a node y

Output: The lowest common ancestor of x and y

If root is null or x is root or y is root:

Return root

BinaryTree leftNode <- LowestCommonAncestor(x, y, root.left)

BinaryTree rightNode <- LowestCommonAncestor(x, y, root.right)

If leftNode = null:

Return rightNode

Else if rightNode = null:

Return leftNode

Return root

The running time of the algorithm is O(h), h is the height of the tree.

**R-3.6 Give a pseudocode description of an algorithm to find the element with smallest key in a binary search tree. What is the running time of your method?**

The idea is about finding the left most node of the tree. If we the current node has a left child, then we give the child’s address to its parent. Until the current node has no left child, this current node has the smallest key in the binary search tree.

**Algorithem**: FindSmallestKey(BineryTree root):

Input: The root of the binary search tree

Output: The minimum key

Min <- root

while min.left ≠ null

min <- min.left

return min

The running time of the algorithm is O(h), h is the height of the tree.

**C-3.3 Describe how to perform the operation findAllElements(k), which returns every element with a key equal to k (allowing for duplicates) in an ordered set of n key- value pairs stored in an ordered array, and show that it runs in time O(log n + s), where s is the number of elements returned.**

Since it is ordered, we can use binary search for this question.

1. Suppose we call the key with highest value Hi, the key with lowest value Lo, the value in the middle position of sets M.
2. If the value at M > k, then Lo = M; redo 1.
3. If the value at M < k, then Hi = M; redo 1.
4. if the value at M = k, then the next s value would be the answer we find.

Because binary search cost O(logn) time, n is the length of array. And s is the number of duplicates that we found, so the time complexity would be O(logn + s).

**C-3.6 Describe how to perform an operation removeAllElements(k), which removes all key-value pairs in a binary search tree T that have a key equal to k, and show that this method runs in time O(h + s), where h is the height of T and s is the number of items returned.**

The same idea as the previous question

**Algorithm**: FindAllElements(target, root):

Input: Search key target, and the root of a binary search tree T

Output: The position of the value that equals to k.

if root = null then

return null

if root.key= target then

return root

else if root.key < target then

return FindAllElements(target, root.right)

else if target < root.key then

return FindAllElements(target, root.left)

**Algorithm**: removeAllElements(target, root):

Input: Search key target, and the root of a binary search tree T

Output: The updated bineryTree T.

cur <- FindAllElements(target, root)

par <- cur.parent

while cur.child = cur

remove(cur)

remove(cur)

We spend O(h) time to find the node that equals to the target. And then we spend s to remove all the duplicate. So the total time complexity is O(h + s)

**A-3.2 Imagine that you work for a database company, which has a popular system for maintaining sorted sets. After a negative review in an influential technology web- site, the company has decided it needs to convert all of its indexing software from using sorted arrays to an indexing strategy based on using binary search trees, so as to be able to support insertions and deletions more efficiently. Your job is to write a program that can take a sorted array, A, of n elements, and construct a binary search tree, T , storing these same elements, so that doing a binary search for any element in T will run in O(log n) time. Describe an O(n)-time algorithm for doing this conversion.**

1. To solve this problem, I would like to take the element at half of A as the root of the tree.
2. And then we split this array into 3 parts: [0, half the array -1], root, [half the array +1, the end of the array].
3. We recursively put the first part and the third part into 1 and connect the parent and children nodes we got.

Since we access all single element in the array the one time, so the total running time complexity is O(n).