**Chapter-2**

***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.

**Answer:** If there is an array with appropriate length which does not overflow. Suppose there are two pointers where the first position is fp and last is lp. The situation where the array has not values the index value of first and last place would be 0. If the first position of the array is positive we insert the element to array[fp]. If fp is non-zero negative, then the element is added to array [array.length + fp] | fp = fp -1 and size = size +1. If index 0 is removed the first positive position then first position is simply deleted. If first position is negative then element at array/length +fp is deleted and then fp ++ and size = size -1. The similar idea is implemented when element is added to the end of the list, lp =lp -1, the last position is deleted and size =size -1. The get method (get(i)) can be written as: i +fp is a negative, then return array [array.length + i + fp - 1], else return array [i+fp].

**List.java**

public class List {

int array [ ];

int front, size;

public List (int size) {

array = new int[size];

front = 0;

this.size = 0;

}

public void insertAtFront(int item) {

if (size == array.length) {

System.*out*.println("Overflow!!");

return;

}

front = (front - 1 + array.length) % array.length;

array[front] = item;

size = size + 1;

}

public int deleteAtFront() {

if (size == 0) {

System.*out*.println("Underflow");

return Integer.*MAX\_VALUE*;

}

int deleted = array[front];

front = (front + 1 + array.length) % array.length;

size = size - 1;

return deleted;

}

public void insertAtEnd(int item) {

if (size == array.length) {

System.*out*.println("Overflow!!");

return;

}

array [(front + size) % array.length] = item;

size = size + 1;

}

public int deleteAtEnd() {

if (size == 0) {

System.*out*.println("Underflow");

return Integer.*MAX\_VALUE*;

}

int deleted = array [(front + size) % array.length];

size = size - 1;

return deleted;

}

public int get (int index) {

if (index < 0 || index >= size)

throw new ArrayIndexOutOfBoundsException("Index lies between 0 to " + (size - 1));

return array [(front + index) % array.length];

}

@Override

public String toString() {

StringBuilder builder = new StringBuilder();

builder.append("[");

for (int i = 0; i < size - 1; ++i) {

builder.append(get(i)).append(",");

}

builder.append(get(size - 1)).append("]");

return builder.toString();

}

}

**Test.java**

public class Test {

public static void main (String [] args) {

List myList = new List (5);

myList.insertAtFront(1);

System.*out*.println(myList);

myList.insertAtEnd(2);

System.*out*.println(myList);

myList.insertAtFront(3);

System.*out*.println(myList);

System.*out*.println(myList.get(2));

myList.deleteAtFront();

System.*out*.println(myList);

myList.deleteAtEnd();

System.*out*.println(myList);

}

}

***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*.

**Answer:** 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 lowestlevel

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?

**Answer:** 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.

**Chapter – 3**

***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?

**Answer:** Left most node of the tree is to be found. So, the current node has a left child then child’s address is provided to its parent. Till the current node has null child on left, the current node has smallest key in the binary search tree.

Algorithm: 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* keyvalue

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.

**Answer:** This array is ordered so binary search can be used.

Suppose the highest value in the key is H, the key with the lowest value is L, the value in the middle position of sets M.

If value at M > K, then L =M; repeat step 1.

If value at M < k, then H =M; repeat step 1.

If the value at M =k, then the next s value would be the answer.

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

**Answer:** The concept is same as C 3.3

* 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,

return null

if root.key= target,

return root

else if root.key < target,

return FindAllElements(target, root.right)

else if target < root.key,

return FindAllElements(target, root.left)

* 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 website,

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.

**Answer:**

1. To solve, element at half of A as the root of the tree is selected.

2. The array is divided into 3 parts: [0, half the array -1], root, [half the array +1, the end of the array].

3. The first part and the third part into 1 and connect the parent and children nodes available.

All elements of the array are accessed at least one so the total running time complexity is O (n).