

### ASSIGNMENT 3

Submitted by : Rajwinder kaur

Student number : 216907602

Username: rkaur3

**Question 1 :** *Describe a method for performing a card shuffle of a list of  $2n$  elements, by converting it into two lists. A card shuffle is a permutation where a list  $L$  is cut into two lists,  $L1$  and  $L2$ , where  $L1$  is the first half of  $L$  and  $L2$  is the second half of  $L$ , and then these two lists are merged into one by taking the first element in  $L1$ , then the first element in  $L2$ , followed by the second element in  $L1$ , the second element in  $L2$ , and so on*

**Solution:**

Step 1 : divide the list of length  $2n$  to two equal parts i.e  $L1$  and  $L2$

**Code : for splitting into two lists.**

```
public static Node[] FrontBackSplit(Node L) {
    Node frontL1, backL2;

    // if length is less than 2, handle separately
    if (L == null || L.next == null) {
        frontL1 = L;
        backL2 = null;
        return new Node[] { frontL1, backL2 };
    }
    Node slow = L;
    Node fast = L.next;
    // Fast moves two nodes, and 'slow' moves by one node
    while (fast != null) {
        fast = fast.next;
        if (fast != null) {
            slow = slow.next;
            fast = fast.next;
        }
    }
    // 'slow' is before the midpoint in the list, so split it in two
    // at that point.
    frontL1 = L; // it point to the beginning of the original 2n list
    backL2 = slow.next; // pointer at the element next to mid point. as such in case the length is odd
    slow.next = null; // extra element goes o the first list.
    return new Node[] { frontL1, backL2 };
}
```

**Complexity:**

Its time complexity is  $O(n)$

#### Algorithm FrontBackSplit(Node L)

Dnodes  $L1, L2$

//BASE CASE: if only one element in linked list.

Then  $L1 = L; L2 = \text{NULL}$

Return new node[] { frontL1, BackL2};

Else

/\* create two nodes fast and slow  
everytime small moves one step fast  
moves 2 steps. \*/

**while** (fast != null) {

fast  $\leftarrow$  fast.next //moves by two steps

if (fast != null)

slow  $\leftarrow$  slow.next //moves by one step

fast  $\leftarrow$  fast.next //moves by two steps

}

//once fast is NULL then the node slow is  
the mid point. So split there.

FrontL1 = L;

BackL2= slow.next;

Slow.next= null;

// we have two nodes split in two halves

### ASSIGNMENT 3

Submitted by : Rajwinder kaur

Student number : 216907602

Username: rkaur3

```
public void Merge(Node a, Node b) {  
  
    Node temp = a; // it will be needed to get the head of the new list  
    while (a != null && b != null) {  
  
        Node a1 = a.next;  
        Node b1 = b.next;  
  
        a.next = b;  
        b.next = a1;  
  
        a = a1;  
        b = b1;  
    }  
  
    System.out.println("\n final Mergred List");  
    display(temp);  
  
    System.out.println("\nRemaining Second List");  
    display(b); // b will be pointing to the ahead of the remaining list  
}  
  
public void display(Node head) {  
  
    Node currNode = head;  
    while (currNode != null) {  
  
        System.out.print("->" + currNode.data);  
        currNode = currNode.next;  
    }  
}
```

#### Algorithm steps for merging :

- Let Node A and Node B are the starting of two linked list i.e head of list L1 and L2
- Take Node temp = A ( the head of L1 ).
- let Node A1= A.next // L1's next element  
Node B1 = B.next; // L2's next element
- Make A.next points to the B.
- Make B.next = A1. (at this point B is inserted between A and A1).
- Do the above two steps till one of the list points to null
- Print the list using temp node.

**Question 2 :** The balance factor of an internal position  $p$  of a proper binary tree is the difference between the heights of the right and left subtrees of  $p$ . Show how to specialize the Euler tour traversal of Section 8.4.6 to print the balance factors of all the internal nodes of a proper binary tree.

**Solution:** Euler tour traversal is flexible and preorder, In order and postorder are the special cases of the euler tour traversal. As per question euler tour algorithm as mentioned in 8.4.6 is modified to print the balance factor of each node.

If post order traversal is the best approach for printing balancing factor for every single node in the tree. So in post order traversal it will calculate the height of every single node from the bottom and then grow upwards. So its the best way that could be used in the least time.

*Algorithm PostOrder (v)*

*For each child w of v*

*Calculate max height of the left child of w*

*Calculate max height of right child of w*

*Calculate balancing factor = max height on the left side – max height on the right side.*

*System.out.println ( “ balancing factor of node “ + w + “ is “ + balancing factor);*

*Post order(w);*

The modified eulerTour algorithm for printing balance factor for each node from section 8.4.6 is :

### ASSIGNMENT 3

Submitted by : Rajwinder kaur

Student number : 216907602

Username: rkaur3

```
Algorithm eulerTour( $T, p$ ):  
perform the “pre visit” action for position  $p$   
  
if  $p$  has a left child  $lc$  then  
    balanceFactor( $lc$ );  
eulerTour( $T, lc$ ) { recursively tour the left subtree of  $p$  }  
  
perform the “in visit” action for position  $p$   
    balanceFactor( $p$ );  
  
if  $p$  has a right child  $rc$  then  
    balanceFactor( $rc$ );  
eulerTour( $T, rc$ ) { recursively tour the right subtree of  $p$  }  
perform the “post visit” action for position  $p$   
  
END
```

```
Algorithm void balanceFactor(TreeNode start) {  
    Bf  $\leftarrow$  height(start.left)-height(start.right)  
    Print Bf  
End Algorithm  
  
Algorithm height(TreeNode start) {  
    if(start == null) return 0;  
    return 1 + Math.max(height(start.left), height(start.right))  
    end algorithm
```

### ASSIGNMENT 3

Submitted by : Rajwinder kaur

Student number : 216907602

Username: rkaur3

**Complexity analysis:** For visiting the left, right, child node time complexity is  $O(1)$ .

Depth function is  $O(dp+1)$  , maximum it could be  $O(n)$ .

So out overall time complexity for this is  $O(n)$

space complexity is  $O(n)$

The depth method at position  $p$  runs in  $O(dp+1)$  time where  $dp$  is its depth

**Code for program that displays the balancing node for every single node of a tree :**

**CODE DISPLAYED ON NEXT PAGE**

```

public class BinarySearchTree {
    class TreeNode {
        public int key;
        public TreeNode p;
        public TreeNode left;
        public TreeNode right;

        public TreeNode() {
            p = left = right = null;
        }

        public TreeNode(int k) {
            key = k;
            p = left = right = null;
        }
    }

    public TreeNode root;

    public BinarySearchTree() {
        root = null;
    }

    public void insert(int k) {
        TreeNode newNode = new TreeNode(k);

        if(root == null) {
            root = newNode;
        } else {
            TreeNode parent = root;
            while(true) {
                if(parent.key == k) {
                    return;
                }
                if(parent.key > k) {
                    if(parent.left == null) {
                        parent.left = newNode;
                        newNode.p = parent;
                        return;
                    } else {
                        parent = parent.left;
                    }
                } else {
                    if(parent.right == null) {
                        parent.right = newNode;
                        newNode.p = parent;
                        return;
                    } else {
                        parent = parent.right;
                    }
                }
            }
        }
    }
}

```

### ASSIGNMENT 3

Submitted by : Rajwinder kaur

Student number : 216907602

Username: rkaur3

```
public void printReport() {
    System.out.printf("%-5s%-8s%-5s\n", "Key", "Height", "Balance Factor");
    printReport(root, 1);
}

private void printReport(TreeNode start, int height) {
    if(start == null) {
        return;
    }
    System.out.printf("%-5d%-8d%-5d\n", start.key, height, balanceFactor(start));
    printReport(start.left, height + 1);
    printReport(start.right, height + 1);
}

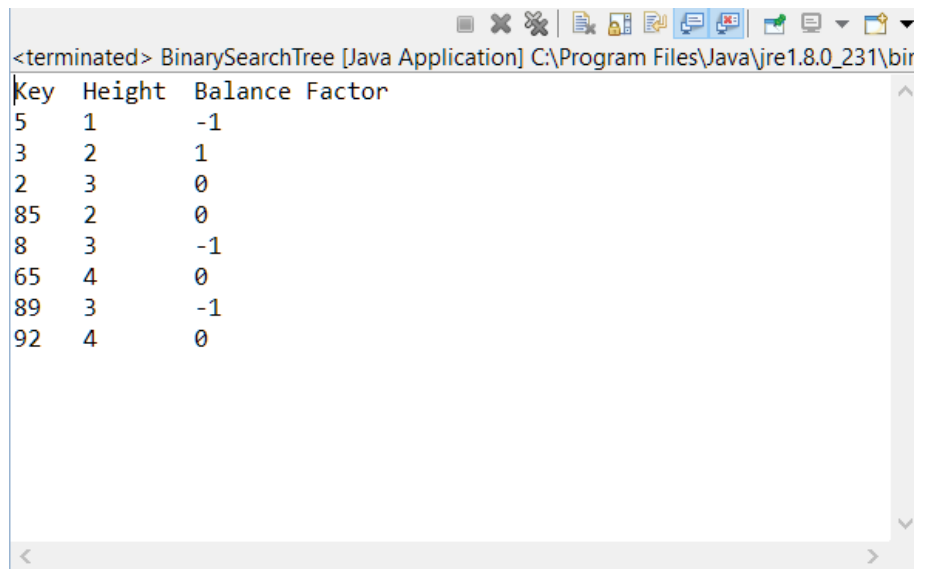
private int depth(TreeNode start) {
    if(start == null) {
        return 0;
    }
    return 1 + Math.max(depth(start.left), depth(start.right));
}

private int balanceFactor(TreeNode start) {
    return depth(start.left) - depth(start.right);
}

public static void main(String[] args) {
    int[] array = { 5, 85, 89, 3, 2, 8, 65, 92 };
    BinarySearchTree bst = new BinarySearchTree();
    for (int i = 0; i < array.length; i++)
        bst.insert(array[i]);

    bst.printReport();
}
```

#### OUTPUT OF CODE:



Key	Height	Balance Factor
5	1	-1
3	2	1
2	3	0
85	2	0
8	3	-1
65	4	0
89	3	-1
92	4	0

### QUESTION 3 :

Let  $S$  be a set of  $n$  points in the plane with distinct integer  $x$ - and  $y$ -coordinates. Let  $T$  be a complete binary tree storing the points from  $S$  at its external nodes, such that the points are ordered left to right by increasing  $x$ -coordinates. For each node  $v$  in  $T$ , let  $S(v)$  denote the subset of  $S$  consisting of points stored in the subtree rooted at  $v$ . For the root  $r$  of  $T$ , define  $top(r)$  to be the point in  $S = S(r)$  with maximal  $y$ -coordinate. For every other node  $v$ , define  $top(v)$  to be the point in  $S$  with highest  $y$ -coordinate in  $S(v)$  that is not also the highest  $y$ -coordinate in  $S(u)$ , where  $u$  is the parent of  $v$  in  $T$  (if such a point exists). Such labeling turns  $T$  into a **priority search tree**. Describe a linear-time algorithm for turning  $T$  into a priority search tree. Implement this approach.

#### Solution.

Space is  $O(n)$

According to question  $S(v)$  where  $v$  is a node with the maximum  $y$  value in the set  $S$  such that in subset of tree with root  $S(v)$  root node will have the highest  $y$  value. Such a priority tree can be made by combining the Binary tree and priority queue ADT.

Anything that is below node  $v$  (have highest  $y$  coordinate) have low value of key. here key determines the priority level. So the values on the left side of the tree will have lower  $x$  values and values on the right side of the tree have higher  $x$  values.

#### Creating the priority search tree:

For a given set of points  $S$ , we create a priority search tree as follows:

1. If  $S$  is empty, we return a NULL pointer and do not continue.
2. The point  $P$  in  $S$  with the the greatest  $Y$ -coordinate becomes the root  $R$ .
3. If  $P$  is empty,  $R$  is also a leaf; we return  $R$  and stop.
4. Let  $X(P)$  be a value such that half of points in set  $S - P$  have  $X$ -coordinate lower than  $X(P)$ , and half have higher.
5. Recursively create a priority search tree on the lower half of  $S - P$ , let its root be the left child of  $R$ .
6. Recursively create a priority search tree on the upper half of  $S - P$ , let its root be the right child of  $R$ .



```

Algorithm PSTCreate(Node point p)
Leftdata[ ], rightdata[];
If(node==null ) return null;
If(p !=null)
For(every point p)
    If(p.key< node_key)
        Leftdata.append(p)
    Else
        rightdata.append(p);

Left_subtree = PSTCreate(leftdata)
Right_subtree= PSTCreate(rightdata)
Return node // Node containing the node_key, node_point and the left and right subtrees
}
    
```

### Searching a key inside priority search tree :

1. If the tree is NULL, we return without finding any points.
2. Let R be the root of the tree, x be its X-coordinate, y be its Y-coordinate, and  $X(R)$  be the value of the axis separating the X-ranges of R's child subtrees.
3. Compare y to  $y'$ . If  $y < y'$ , we return without finding any points (all other nodes in the tree will have an even smaller Y-coordinate).
4. If  $x' \leq x \leq x''$ , report the root point.
5. If  $x' < X(R)$ , the X-range of the left subtree must overlap with the X-range of the query. Recursively search the left subtree of R.
6. If  $X(R) < x''$ , the X-range of the right subtree must overlap with the X-range of the query. Recursively search the right subtree of R.

```

Algorithm point_search_tree (tree, min_key, max_key, max_priority)

    root = get_root_node(tree) // root is the root of the tree i.e. v in subtree s(v) in with root v
result = [ ] // empty array
if get_child_count(root) > 0 {
    if get_point_priority(root) > max_priority
        return null // Nothing interesting will exist in this branch. Return

    if min_key <= get_point_key(root) <= max_key
        result.append(get_point(node))

    if min_key < get_node_key(root)
        result.append(search_tree(root.left_sub_tree, min_key, max_key, max_priority))

    if get_node_key(root) < max_key
        result.append(search_tree(root.right_sub_tree, min_key, max_key, max_priority))

    return result

else { // This is a leaf node
    if get_point_priority(root) < max_priority and min_key <= get_point_key(root) <= max_key // is leaf point of
interest?
        result.append(get_point(node))

```

## Complexity analysis:

priority search tree on n points takes up space  $O(n)$ , as each point represents one node . The height of the tree is  $O(\log n)$ , since the nodes are partitioned in half at each level. It remains to show that for a query with an answer of size k time will be  $O(k + \log n)$ .