(b) Consider the recurrence

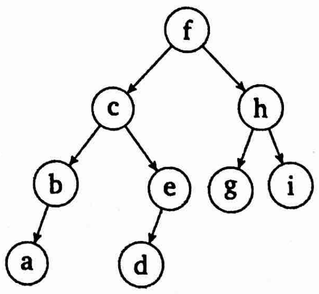
$$T(n) = \begin{cases} 1 & \text{if } n \le 2 \\ 2 \cdot T(\sqrt{n}) + \log n & \text{otherwise} \end{cases}$$

Solve the above recurrence using the tree method. First unroll two levels of the tree and draw this unrolling as a tree. Finish your analysis by completing the missing entries of this table.

# of nodes at level i	2^i
input size at level i	$n^{\frac{1}{2^i}}$
work per node at level i	$\log n^{\frac{1}{2^i}} = \boxed{\frac{1}{2^i} \log n}$
total work at level i	$\cancel{2^i} \cdot \frac{1}{\cancel{2^i}} \log n = \boxed{\log n}$
level of base case	$n^{\frac{1}{2^i}} = 2 \Rightarrow \boxed{i = \log(\log n)}$
number of nodes in the base case level	$2^{\log(\log n)}$
expression for recursive work	$\sum_{i=0}^{\log(\log n)-1} \log n = \boxed{\log n * \log(\log n)}$
expression for non-recursive work	$1 * \log n = \boxed{\log n}$
closed form for total work	$\log n + \log n * \log(\log n)$
simplest Big-Θ for the total work	$\because \log n < \log n \cdot \log(\log n)$ $\therefore \Theta(\log n \cdot \log(\log n))$

3. Binary Search Trees

(a) Consider the following Binary Search tree:



Write down the

(i) in-order traversal a b c d e f g h i

(ii) preorder traversal f c b a e d h g i

(iii) postorder traversal a b d e c g i h f

Do you notice an interesting property of the in-order traversal? What is it?

(b) Let a binary search tree be defined by the following class:

(Next Page !)

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

        // constructors omitted for clarity
    }
}
```

In class, we saw how to search for an element in a binary search tree. This question will demonstrate that binary search trees are more powerful. Describe an algorithm that calculates the k'th smallest element in the tree. (k, the input, is a number in $\{1, 2, \cdots, n\}$, where n is the number of nodes in the tree). You may find it helpful to modify the definition of IntTreeNode in order to accomplish this. Your algorithm should run in $O(h)$ time, where h is the height of the tree. An $O(n)$ solution will be given partial credit.

4. AVL Tree Implementation

(Next page !)

Write pseudocode for the AVL tree methods Balance, RotateLeft, and RotateRight. Assume that the rest of the data structure is implemented as in the Java code here `https://courses.cs.washington.edu/courses/cse373/18su/files/homework/AVLTree.java`.

You may use the skeleton of Balance from that code as a guide. Note you are not required to write your solution in Java, pseudocode is sufficient.

5. Hashing

Let the capacity of the hash table be 10 and the hash function be $h(x) = x$. Insert elements

$$42, 102, 12, 33, 25, 14, 62$$

to a hash table

(Next Page !)

(a) that uses linear probing

(b) that uses quadratic probing

Write down the total number of collisions and the hash table after all insertions in both cases. Why is the secondary clustering in quadratic probing less problematic than the primary clustering in linear probing (i.e. why are there fewer collisions)?

3 (b) Sine the kth smallest element in the tree, there're two sides we set two number of Node, one is on left, one is on right. So first set these in constructors.

```
private class IntTreeNode {
    public int data;
    public IntTreeNode left;
    public IntTreeNode Right;
    public int left_Size ;   // number of Node on left.
    public int RightSize;   // number of Node on Right.
```

Then we set a method call "search" that to find the k.

```
public Search (int k, IntTreeNode node ; int base);
    if k is equal to the base + node.left size
            return the node
       // if base is 0, let assume i is the position, all work from 1
   to i-1 will be at left side (since they are less). then it
 k > k-1   we only focus on the i and i+1 node.
} else if (k <= left size), k will be on left side.
        return Search (k, node.left. base)
} else   //  k > left size + 1  (the base root)
        return   Search (k, noderight. base+ left size+1).
```