1. (a) Recall the HEAP operation, **PUSHDOWN** (*dtype* A[], *int* r, *int* n), where A[1..n] is the array containing a heap of n elements and r is the index of the root node to be fixed. Initially, the left and right subtrees of r already satisfy the heap property, and at the end the entire tree (or subtree) rooted at r will satisfy the heap property.

Write a recursive version of this function.

```c
void PUSHDOWN (dtype A[], int r, int n) {
    int smallerchild;
    if (2*r > n) return; // return if r is a LEAF node
    if (2*r == n or A[2*r] <= A[2*r+1]) smallerchild = 2*r;
    else smallerchild = 2*r + 1;
    if (A[r] > A[smallerchild]) {
        SWAP(A[r],A[smallerchild]);
        PUSHDOWN(A, smallerchild, n);
    }
}
```

(b) The algorithm **MAKEHEAP** starts with an array of n arbitrary values and establishes a HEAP. Write a recursive version of this algorithm. (This algorithm makes call(s) to **PUSHDOWN**.)

```c
void MAKEHEAP (dtype A[], int r, int n) {
    if (2*r > n) return; // return if r is a LEAF node or beyond
    MAKEHEAP(A,2*r,n); //Make heap of the left subtree
    MAKEHEAP(A,2*r+1,n); //Make heap of the right subtree
    PUSHDOWN(A,r,n);
}
```

Initial call is made with r = 1.

(c) Let \( f(n) \) be the worst-case number of key comparisons for **MAKEHEAP**. Write a recurrence for \( f(n) \). (To simplify the analysis, assume the heap is a FULL binary tree.)

**Solution:** First, let us analyze the worst-case number of key-comparisons in **PUSHDOWN(A, 1, n)**. For each level of the heap, two comparisons are made. Thus, the total is \( 2 \lceil \log n \rceil \).

\[
   f(n) = \begin{cases} 
   2f\left(\frac{n-1}{2}\right) + 2\lceil \log n \rceil, & n > 1 \\
   0, & n = 1 
   \end{cases}
\]

The recurrence may be slightly simpler if it is expressed as inequality:

\[
   f(n) \leq \begin{cases} 
   2f\left(\frac{n}{2}\right) + 2\log n, & n > 1 \\
   0, & n = 1 
   \end{cases}
\]

The recurrence may also be written in terms of height \( h \) instead of \( n \).

(d) What is the time complexity of **MAKEHEAP**?

**Solution:** It is easy to show (by induction) that the solution of the above recurrence is: \( f(n) \leq An + B \log n + C \). Therefore, the time complexity of **MAKEHEAP** is \( O(n) \).
2. (a) Insert the following sequence of elements into an AVL tree, starting with an empty tree. Show the tree after each balancing operation.

10, 20, 15, 25, 30, 16, 18, 19

(b) Show the final AVL tree below. Then, delete element 30 in the AVL tree.
3. Consider a hash table with integer keys and the hash function \( h(x) = x \mod 13 \). Assume linear open-addressing. Assume every table entry has only two possible markings: (1) Free, and (2) Occupied.

(a) Insert the following sequence of elements, starting with an empty table.

10, 21, 34, 23, 37, 36

Insert: 10, 21, 34, 23, 37, 36
\( h(x) = 10, 8, 8, 10, 11, 10 \)

(b) Show the final table below (after the above insertions). Then delete element 21. Show the sequence of move operations to fill the hole.
4. Consider the comparison tree used to prove the lower bound for sorting.

(a) Clearly state what was proved. (State only the theorem, not the proof.)

**Theorem:** Any comparison-based sorting algorithm (for arbitrary values) must in the worst-case make at least \((\log n!)\) comparisons, which is \(\Omega(n \log n)\).

(b) Prove that \(\log n! = \Omega(n \log n)\).

\[
\log n! = \sum_{i=1}^{n} \log i \\
\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\
\geq \sum_{i=\lceil n/2 \rceil}^{n} \log \lceil n/2 \rceil \\
\geq \lceil n/2 \rceil \cdot \log \lceil n/2 \rceil \\
\geq (n/2) \cdot \log(n/2) \\
\geq (n/2) \cdot \log n - n/2 \\
= \Omega(n \log n).
\]

(c) Consider sorting \(n = 4\) elements by any comparison-based algorithm. Derive the minimum number of comparisons needed in the worst-case.

**Solution:** From the comparison tree, we know the minimum number of comparisons in the worst-case is

\[
\lceil \log n! \rceil = \lceil \log 4! \rceil = \lceil \log 24 \rceil = 5.
\]

\(\square\)
5. We want an efficient algorithm to sort \( n \) records \( G[0..n-1] \), where each record \( G[i] \) contains a key \( G[i].K \) with possible values of either 0 or 1. The sorting must be done IN-PLACE, using only \( O(1) \) amount of additional space.

(a) Write a procedure (the code) for \( \text{SORT}(G, n) \). Give a very brief explanation of the algorithm first. Also, provide comments along with your code.

```
This sorting is very similar to the PARTITION algorithm in quicksort.

```

```c
void SORT (dtype G[ ], int n) {
    int i, j;
    i = 0;
    j = n - 1;
    while (i<j) {
        while (i<j and G[i].K == 0) i=i+1; //left-to-right scan
        while (i<j and G[i].K == 1) j=j-1; //right-to-left scan
        if (i<j) {
            SWAP (G[i],G[j]);
            i=i+1; j=j-1
        }
    }
}
```

(b) What is the time complexity of this algorithm?

**Solution:** The time complexity is \( \Theta(n) \).

(c) Illustrate how the sorting is done on the following array of key values. Show the result after each data movement.

```
0 1 0 1 0 0 1 0 0 0 1 1 0 1 1
```

![Sorting Diagram]