1. (a) Recall the HEAP operation, \textbf{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.

(b) The algorithm \textbf{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 \textbf{PUSHDOWN}.)

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

(d) What is the time complexity of \textbf{MAKEHEAP}?
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.

\[ 10, 21, 34, 23, 37, 36 \]

(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.)

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

(c) Consider sorting \( n = 4 \) elements by any comparison-based algorithm. Derive the minimum number of comparisons needed in the worst-case.
5. We want an efficient algorithm to sort $n$ records $G[0..n-1]$, where each record $G[i]$ contains a key $G[i].KEY$ 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 $SORT(G, n)$. Give a very brief explanation of the algorithm first. Also, provide comments along with your code.

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

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

$\begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
\end{array}$