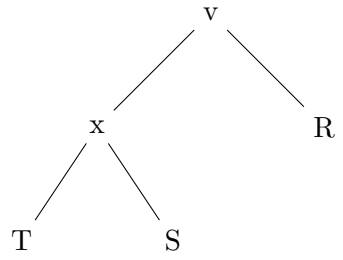
Question 1

We are proving that a single right rotation restores balance after inserting a node into the left subtree of v, causing $weight(v.left) > weight(v.right) \times 4$.

Original Tree Structure and Justifications

Before insertion, the tree structure is as follows:



- v is the root node.
- x is the left child of v.
- R is the right subtree of v.
- T and S are the left and right subtrees of x, respectively.

Weights are defined as:

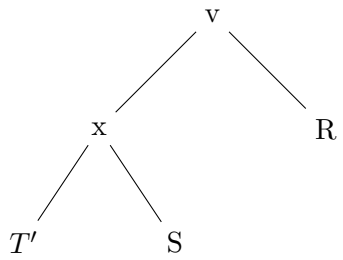
$$weight(n) = size(n) + 1$$
$$weight(v.left) = weight(x)$$
$$weight(v.right) = weight(R)$$

Assumption for Single Right Rotation

A node is inserted into T, creating a new subtree T', which causes $weight(v.left) > weight(v.right) \times 4$. Additionally, we assume $weight(x.right) < weight(x.left) \times \frac{5}{3}$. These conditions imply that a single right rotation is necessary and sufficient to balance the tree.

Post Insertion

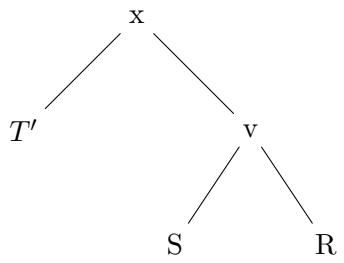
After the insertion, the tree structure becomes:



$$weight(v.left) = weight(x)$$
$$weight(x.left) = weight(T')$$
$$weight(x.right) = weight(S)$$

Right Rotation

Performing a single right rotation results in the following tree structure:



- x becomes the new root.
- T' remains the left subtree of x.
- v becomes the right child of x.
- S becomes the left subtree of v.
- R remains the right subtree of v.

Verifying Balance After Rotation

After the rotation, the weights are:

$$weight(x.left) = weight(T')$$
$$weight(x.right) = weight(v)$$
$$weight(v.left) = weight(S)$$
$$weight(v.right) = weight(R)$$

The new weights can be expressed as:

$$weight(x) = weight(T') + weight(v) + 1$$
$$weight(v) = weight(S) + weight(R) + 1$$

To confirm that the tree is balanced, we need to ensure that the balancing equations hold:

$$\frac{1}{4} \leq \frac{weight(x.left)}{weight(x.right)} \leq 4$$
$$\frac{1}{4} \leq \frac{weight(T')}{weight(v)} \leq 4$$

Given:

$$weight(v.left) > weight(v.right) \times 4$$
$$weight(x.right) < weight(x.left) \times \frac{5}{3}$$

These conditions ensure that the tree remains balanced after the rotation.

Conclusion

By verifying the weights and conditions before and after the rotation, we conclude that a single right rotation successfully rebalances the tree when $weight(v.left) > weight(v.right) \times 4$ after inserting a node into the left subtree. This completes the proof for the given case.

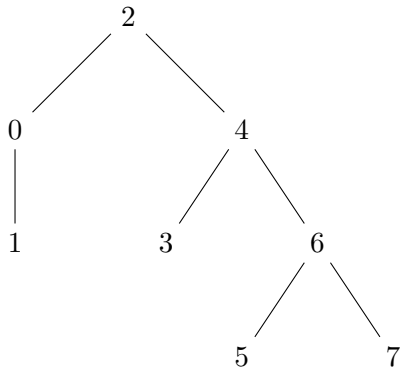
Question 2

Using the AVL union algorithm:

```
if T_1 == nil:
    return T_2
if T_2 == nil:
    return T_1

k = T_2.key
(L, R) = split(T_1, k)
L' = union(L, T_2.left)
R' = union(R, T_2.right)
return join(L', k, R')
```

Trees T_1 and T_2



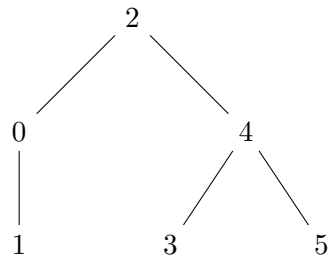
6

Steps of the Algorithm

Initial Check: Neither T_1 nor T_2 is nil.

Extract Key and Split:

- $k = T_2.key = 6$
- Split T_1 at $k = 6$:
  - Left subtree (L) of T_1 with keys less than 6:

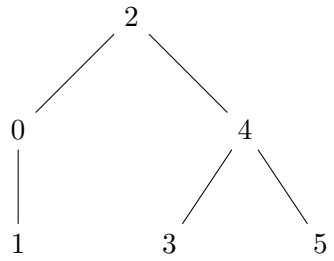


– Right subtree (R) of T_1 with keys greater than 6:

7

Recursive Union Calls:

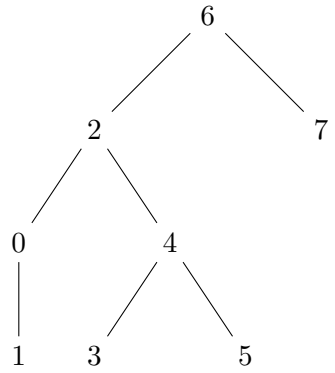
- $L' = \text{union}(L, \text{nil}) = L$:



- $R' = \text{union}(R, \text{nil}) = R$:

7

Join: Join L', $k = 6$, and R':



Conclusion

The union of T_1 and T_2 results in a new AVL tree with the root 6, not 2. Thus, $\text{union}(T_1, T_2) \neq T_1$, as the resulting tree is structurally different from T_1. This process maintains AVL properties, ensuring the tree is balanced.

Question 5

Adjacency List for the Left Tree

To produce the left tree using BFS starting at node 1:

$$
\begin{aligned}
1 &: [2, 3, 4] \\
2 &: [3, 5] \\
3 &: [5, 6] \\
4 &: [3, 6] \\
5 &: [7, 8] \\
6 &: [8, 9] \\
7 &: [] \\
8 &: [] \\
9 &: []
\end{aligned}
$$

Adjacency List for the Right Tree

To produce the right tree using BFS starting at node 1:

$$
\begin{aligned}
1 &: [4, 3, 2] \\
2 &: [3, 5] \\
3 &: [5, 6] \\
4 &: [6, 3] \\
5 &: [7, 8] \\
6 &: [8, 9] \\
7 &: [] \\
8 &: [] \\
9 &: []
\end{aligned}
$$

Explanation for Invalid BFS

This BFS is invalid as node 8 must be a child of node 5 given an adjacency list where node 5 is a child of node 2, node 6 is a child of node 3, and node 4 has no children.

Consider how the first three levels can only be a result of an adjaceny list where $1 : [2, 3, 4]$. This results in the following order of traversal at each level: $[5, 6]$ then $[7, 8, 9]$. The implication is that $[7, 8]$ must be traversed as children of 5.

Question 6

1. Maximum Number of Edges in an Undirected Graph

Statement: The maximum number of edges in an undirected graph G (no self-edges) with n vertices is $\frac{n(n-1)}{2}$.

Proof:

- Each pair of vertices can be connected by exactly one edge.
- The total number of ways to choose 2 vertices from n vertices is given by:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

- Thus, the maximum number of edges in an undirected graph with n vertices is $\frac{n(n-1)}{2}$.

2. No Cycle in an Undirected Graph with n Vertices and $n-1$ Edges

Statement: For every undirected graph G (no self-edges) with n vertices and $n-1$ edges, G contains no cycle.

Proof by Contradiction:

- Assume G contains a cycle.
- A graph with n vertices and $n-1$ edges must be a tree if connected.
- Trees are acyclic. Adding an edge to a tree introduces a cycle.
- Given G has $n-1$ edges, it must be a tree and thus acyclic.
- Therefore, G cannot contain a cycle.

Question 4

Worst Case for Both Functions

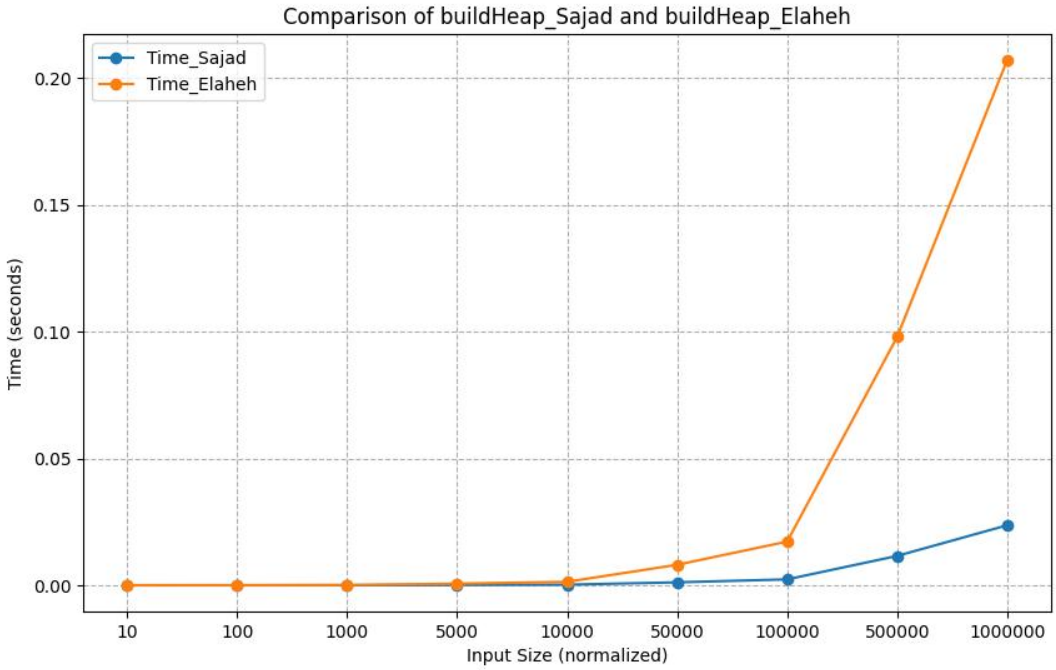
We can choose a reverse sorted array to be the worst-case input for both `buildHeap_Sajad` and `buildHeap_Elaheh`. This ensures the maximum number of comparisons and swaps needed to maintain the heap property.

Testing Logic

First, we generate input arrays of specified sizes in descending order to use as worst-case inputs for the `buildHeap` functions. Next, we run each version of `buildHeap` (`buildHeap_Sajad` and `buildHeap_Elaheh`) separately for the generated input array. We execute each function approximately 100 times to get accurate measurements of the running time. Finally, we measure the time taken for each `buildHeap` function to complete using the input arrays, utilizing timing functions to measure the running time in C.

Results

We perform the tests for at least 10 different input sizes, ranging between 10 and 1,000,000. We plot the running time on the y-axis against the input size on the x-axis for both versions of `buildHeap`. Here are the results:



Analysis

From the observations, `buildHeap_Sajad` consistently outperforms `buildHeap_Elaheh` across all input sizes. The time complexity of `buildHeap_Sajad` appears to be closer to $O(n)$, while the time complexity of `buildHeap_Elaheh` is closer to $O(n \log n)$.

The theoretical complexity of `buildHeap_Sajad` involves constructing the heap by first inserting all elements into the heap array and then applying the `heapify` function from the middle of the array to the root. The `heapify` operation ensures that the subtree rooted at a given node is a min-heap. In the worst case, `heapify` needs to be called on all nodes from the middle of the array down to the root, and each `heapify` call takes logarithmic time in the height of the heap, resulting in a linear total time complexity of $O(n)$.

On the other hand, `buildHeap_Elaheh` constructs the heap by inserting each element individually into the heap. Each insertion operation takes $O(\log n)$ time because the heap needs to maintain its properties after each insertion. Therefore, inserting n elements individually results in a total time complexity of $O(n \log n)$.

Proof of Theoretical Complexity

Theoretical Complexity of `buildHeap_Sajad`

The `buildHeap_Sajad` function first inserts all elements into the heap array and then applies the `heapify` function from the middle of the array to the root. We need to prove that this results in an overall time complexity of $O(n)$.

1. **Heapify Operation**:

- `heapify` is called on each node from $\lfloor n/2 \rfloor$ down to the root.
- The time to `heapify` a node is proportional to the height of the node in the heap. For a complete binary tree, the height of the heap is $\log n$.

2. **Summation of Work Done**:

- Nodes at height h have $\lceil n/2^{h+1} \rceil$ nodes.
- The work done for nodes at height h is $\lceil n/2^{h+1} \rceil \cdot O(h)$.
- Summing over all heights, from 0 to $\log n - 1$:

$$\sum_{h=0}^{\log n - 1} \left(\frac{n}{2^{h+1}} \cdot O(h) \right)$$

3. **Simplifying the Summation**:

$$O\left(\sum_{h=0}^{\log n - 1} \frac{nh}{2^{h+1}} \right)$$

- This series converges to a constant value because the denominator grows exponentially while the numerator grows linearly.
- Therefore, the overall time complexity is $O(n)$.

Theoretical Complexity of `buildHeap_Elaheh`

The `buildHeap_Elaheh` function constructs the heap by inserting each element individually into the heap. We need to prove that this results in a total time complexity of $O(n \log n)$.

1. **Insertion Operation**:

- Each insertion operation takes $O(\log n)$ time because the heap needs to maintain its properties after each insertion.

2. **Total Work Done**:

- Inserting n elements, where each insertion takes $O(\log n)$ time:

$$\sum_{i=1}^{n} O(\log n) = n \cdot O(\log n) = O(n \log n)$$

3. **Conclusion**:

- Therefore, the total time complexity of `buildHeap_Elaheh` is $O(n \log n)$.

Conclusion

From both the experimental data and theoretical analysis, we can conclude that `buildHeap_Sajad` has better performance and lower time complexity compared to `buildHeap_Elaheh`. The experimental results show that `buildHeap_Sajad` is faster, and the theoretical analysis confirms that it has a linear time complexity $O(n)$ as opposed to the $O(n \log n)$ time complexity of `buildHeap_Elaheh`.