Homework 4 - Written assignment

Pratyay Dutta (Codeforces ID : pdutt005)

May 25, 2024

Contents

1 Weight balanced trees

1.1 Prove weight relation

We know that the weight of each node (u) :

$$w(u) = 1 + u' \tag{1}$$

(where u' is the size of the subtree u). According to definition, the size of a subtree is the maximum distance from the root to the leaves of that subtree i.e the size of a leaf node is 0. From the given equation, the weight of a leaf node is 1.
Following from (1) :

$$w(lc(u)) = 1 + lc(u)' \tag{2}$$

(lc(u)' is the size of the subtree of the left child of u)

$$w(rc(u)) = 1 + rc(u)' \tag{3}$$

(rc(u)' is the size of the subtree of the right child of u)
Now, we know that the size of subtree :

$$u' = lc(u)' + rc(u)' + 1 \tag{4}$$

(The size of a tree is the sum of sizes of its subtrees along with itself)
From (4) :

$$u' + 1 = lc(u)' + 1 + rc(u)' + 1 \tag{5}$$

Substituting $u' + 1$ with $w(u)$ from (1) in LHS and using (2) and (3) in the RHS:
w(u) = w(lc(u))+w(rc(u))
Hence Proved

1.2 Prove height bound

In a weight balanced tree, we can show that the balance condition effectively limits the growth of subtree sizes. The subtree has to have a weight which is a fraction of the weight of the parent node and hence this ensures that with incoming insertions and deletions, a particular subtree does not grow in a skewed way and hence prevents the case where there is a a single(or very few) nodes in a level. A subtree contains a significant part of the parent. Let us consider the case where a corresponding subtree has the minimum possible number of nodes (thereby having the maximum possible height). We have been told that the value of α can be considered a constant throughout our weight balanced tree. We have given :

$$\alpha < \frac{w(lc(u))}{w(u)} < 1 - \alpha \tag{6}$$

Therefore for minimum value of w(lc(u)) and w(rc(u)):

$$min(w(lc(u)) = \alpha w(u) \tag{7}$$
$$min(w(rc(u)) = \alpha w(u) \tag{8}$$

Now, we have the weight relation:

$$w(u) = w(lc(u)) + w(rc(u)) \tag{9}$$

Therefore, from (7) and (8) and (9):

$$w(u) = 2\alpha w(u) \tag{10}$$
$$\alpha = 0.5 \tag{11}$$
$$\tag{12}$$

To satisfy balance condition (assuming $/alpha$ is constant for all nodes in the tree) : w(lc(u)) and w(rc(u)) has to be at least 0.5w(u). This means that as we traverse down each level the number of node at least doubles.
If its a leaf node, height = 0 and number of nodes = 1.
When h=1, n is at least 2
When h=2, n is at least 4.
Therefore, we can find that at height h = H, n is at least 2^H(through induction). Therefore:

$$n \geq 2^H \tag{13}$$
$$log(n) \geq Hlog2 \tag{14}$$
$$H \leq log(n) \tag{15}$$

This means that the height of a tree with n nodes is at max log(n). Therefore, **a weight balanced tree having n nodes has height O(log(n).**

1.3 Rebuilding tree instead of rotating

1.3.1 Total cost to rebuild almost balanced subtree

To flatten a tree, we do tree traversals and include each node in an array. To make this process simpler we will employ the following strategy:

- Flatten an imbalanced BST into a sorted array.
- Reconstruct an almost balanced BST from a sorted array.

For flattening :

- Initialize an empty array to store the sorted elements.
- Perform an inorder traversal of the BST.
- At each node:
  - Recursively traverse the left subtree.
  - Append the value of the current node to the array.
  - Recursively traverse the right subtree.
- Build an almost balanced BST from a sorted array.

In this algo, we look at each node exactly once and append it to the sorted array. So for a subtree having n nodes, the time complexity to flatten it is O(n).
For reconstructing a BST from sorted array:

- Initialize an empty BST.
- Find the middle element of the sorted array and make it the root of the BST.
- Recursively:
  - Construct the left subtree using the elements to the left of the middle element in the sorted array.
  - Construct the right subtree using the elements to the right of the middle element in the sorted array.
- Return the root of the BST.

For each element in the array, a new tree node is created and returned. Since there are n elements, and each element requires a constant amount of work to create a node and assign its left and right children (from the recursive calls), this contributes linearly to the overall time complexity. Thus, the overall process involves visiting each element once and performing a constant amount of work for each, leading to a linear time complexity of O(n) for building an almost balanced BST from a sorted array.
Total Time complexity : O(n) + O(n) = **O(n)**

1.3.2 We only need to rebuild once

If we have a case such that inserting a number of elements skews a subtree, then we have to rebuild and balance the subtree.
The process of rebalancing involves identifying the lowest (in terms of tree depth) unbalanced subtree that violates the weight-balance condition due to the operation. Let's denote this subtree as T_u rooted at node u.
After any insertion or deletion, the imbalance is localized along the path from the changed node to the root(imbalance propagates upwards from imbalanced subtree to the root of the tree). The subtree T_u is the first subtree encountered (from the changed node upwards) that is unbalanced.
We can imagine that the imbalance by a certain subtree is the size mismatch with either its related child or its parent. So if we balance that subtree with respect to the constraints, then its balanced with respect to the parent as well as the other same level children.
By rebuilding T_u into a perfectly balanced subtree T'_u, we ensure that T'_usatisfies the weight-balance condition for its size.We can do this by following the algorithm described in the last question(by flattening into a sorted array and then making the most balanced BST possible from it).
Now once the subtree is balanced into T'_u:

- All subtrees below the given subtree has to be balanced because we are rebuilding it as the most balanced BST from the flattened elements.
- Balanced subtree T'_u has the minimum height possible for its size and follows the weight balanced tree criteria.
- All subtrees above u were already balanced before the insertion/deletion. Replacing T_u with T'_u does not change the weight of T_u significantly (it remains within the acceptable range of total subtree size), so the parent of u and all ancestors up to the root still satisfy the weight-balance condition after the rebuild.
- This localized rebuilding ensures that the entire tree becomes balanced because:
  - The rebuilt subtree T'_u is balanced.
  - The balance condition for the parent of u and all ancestors was not violated by the rebuilding, as the weight criteria are maintained.
  - Since the imbalance was localized to the path affected by the insertion/deletion, correcting the imbalance at T_u corrects the imbalance for the entire tree.

1.3.3 Amortized cost of rebuilding

After each insertion:

- Sometimes we rebuild.
- Sometimes,we do not.

Cost to rebuild = O(n)
Therefore, for the insertion and rebuilding excercise, we sometimes pay O(1)(no rebuilding) and sometimes, when we rebuild, we pay O(n).
The logic is that we keep only inserting, till we hit an imbalance and then we balance the tree by rebuilding. We can also make sense of the fact that as the size of tree increases (as n increases), the number of insertions before we reach an imbalance, increases. Keeping in mind the weight balance criterion, the imbalance occurs if we insert m items where $m \geq (1-\alpha)w(u)$.
Since :

$$\alpha < \frac{w(lc(u))}{w(u)} < 1 - \alpha \tag{16}$$

Since $1-\alpha$ is a constant, therefore we can consider $1-\alpha$ as a constant k.
Therefore, we see that the minimum number of insertions before we rebuild is kn.
This means, for **m insertions**:

- If $m \leq kn$: Cost of rebuilding = O(1), (no rebuilding)
- $(m \geq kn)$: Cost of rebuilding = O(m).

Therefore, to find the amortized cost, we find the cost for each insertion:

- If $m \leq kn$: Cost of rebuilding = O(1/m) = O(1)
- $(m \geq kn)$: Cost of rebuilding = O(m/m) = O(1)

Therefore, amortized cost of rebuilding is **O(1)**.

1.3.4 Amortized cost for each insertion

While inserting an element, in worst case, we have to traverse through every node from root to the leaf. We have to traverse O(log(n)) times to find a point of insertion since the height of the tree is O(log(n)). For inserting an element, there can be two cases:

- Insertion without rebuild : Building upon the intuition from the previous question, after inserting an element, if the balance criteria is met, then we do not rebuild. Therefore, total cost = cost(insertion).
- Insertion with rebuild : If after insertion, it violates the balance criterion, we rebuild. Therefore, total cost = cost(insertion)+cost(rebuild).

Deriving from the logic talked about in the previous question, the number of insertions we need to hit imbalance is directly proportional to the number of nodes in the tree.
This means, for **m insertions**:

- If $m \leq kn$: Total cost = cost(insertion) = O(mlog(n))
- $(m \geq kn)$: Total cost = cost(insertion) + cost(rebuilding) = O(mlog(n)) + O(m)

Therefore, to find the amortized cost, we find the cost for each insertion:

- If $m \leq kn$: Total cost = O(mlog(n)/m) = O(log(n)).
- $(m \geq kn)$: Total cost = O(mlog(n)/m) + O(m/m) = O(log(n)) + O(1) = O(log(n)).

Therefore, amortized cost of insertion is **O(log(n))**.

2 Union Find

According to Prof.Yan Gu's instructions, this question has been shifted to HW5.

3 Spanning trees

3.1 Spanning tree definition

A spanning tree of a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is a subgraph $T = (V_T, E_T)$ of G that satisfies the following conditions:

1. $V_T = V$, meaning T includes all the vertices of G.
2. T is a tree, which implies:
   - T is connected: There is a path between every pair of vertices in T.
   - T contains no cycles: There is exactly one path between every pair of vertices in T.
3. $|E_T| = |V| - 1$, which means the number of edges in T is one less than the number of vertices in G.

Thus, a spanning tree T of G is a minimally connected subgraph that includes all of G's vertices without forming any cycles.

3.2 How many edges in spanning tree

Given a graph G = (E,V) with positive edge weights, the number of edges in its spanning tree = $|V| - 1$.
Reason :

- Adding any edge to a spanning tree will create a cycle, violating the tree's acyclic property. This is because, in a tree, there is exactly one unique path between any two vertices. Introducing an additional edge between any two vertices would offer an alternative path, creating a loop or cycle.
- The property of having $n-1$ edges for n vertices ensures that the spanning tree is the most efficient way to connect all vertices in the graph without redundancy. This efficiency is crucial for applications such as network design, where minimizing the number of connections can reduce costs while maintaining full network connectivity.

Therefore, the spanning tree o a graph always contains $|V| - 1$ edges.

3.3 Maximum Spanning tree

1. **Algorithm** To find the Maximum Spanning Tree of a graph, we use Prim's algorithm. Instead of finding the lightest edge at each node, we find the heaviest edge. The algorithm works as follows:

   - **Initialization**
     - Choose a starting vertex v_0 arbitrarily.
     - Initialize an empty set T to store the edges of the maximum spanning tree.
     - Initialize a priority queue Q with the vertices of the graph.
     - Set the priority of each vertex in Q to $-\infty$ except for v_0, which is set to 0.
   - **Main Loop**
     - While Q is not empty:
       * Extract the vertex v with the maximum priority from Q. Priority here, means the edge weights. So the vertex with maximum priority means the vertex with the highest edge weight connected to the node.
       * Add the edge (v, u) to T, where u is the parent of v in the maximum spanning tree.
       * Update the priorities of all vertices adjacent to v in Q by taking the maximum of their current priority and the weight of the edge (v, u).
   - **Output**
     - Output the set T containing the edges of the maximum spanning tree.

2. **Time Complexity** : V is the vertices of the graph. Since our algorithms works on the vertices, we can consider V to be n. Also, it is known that in a complete graph, $E = O(V^2)$ (If all vertices connected to all other vertices, $E = V(V-1)/2 \approx n^2$. We will compute the time complexity in terms of n.

   - In the main loop :
     - The main loop runs V times, where V is the number of vertices in the graph.
       * Extracting the vertex with maximum priority from Q: $O(\log V)$ (getting max element from a max binary heap and then restructuring the heap takes O(logV) time).
       * Updating priorities of adjacent vertices: $O(E)$, where E is the number of edges in the graph.
     - Total time complexity of Step 2: $O(V \cdot \log V + E) \approx O(V \log V + E)$

   Overall Time Complexity: $O(V \log V + E)$. Here V=n and $E \approx n^2$. Therefore, time complexity can be expressed as $O(n \log n + n^2) = O(n^2)$

3. **Correctness of the algorithm** :

   - At the start, we choose an arbitrary vertex and make it the initial tree. This is trivially a maximum spanning tree of the graph consisting of a single vertex, hence the loop invariant holds.
   - At each step, assuming we have a maximum spanning tree T for a subset V' of vertices, we select the heaviest edge e that connects a vertex in V' with a vertex outside V'. By adding this edge and the new vertex to T, we maintain the invariant that T is a maximum spanning tree for its vertices because the selection of the heaviest edge ensures the maximum property is preserved.
   - The algorithm terminates when $V' = V$, meaning T includes all vertices. At this point, T is a maximum spanning tree because throughout the algorithm, we ensured that the added edges are the heaviest possible that do not form a cycle, guaranteeing that T has the maximum total weight among all spanning trees.

4 Basic Programming Problems

Submitted last week.

5 Bonus Problems

5.1 Melody : Submission ID - 247728932

Introduction: Our algorithm combines hash mapping and binary search techniques to efficiently solve the problem.

Problem Description: Given a melody represented as a sequence of integer notes, the goal is to compute the length of the longest theme. A theme is defined as a subsequence that:

- Appears again elsewhere in the melody, possibly transposed.
- Is disjoint from at least one of its occurrences.

Input Processing:

1. Input Collection: The algorithm starts by collecting an integer n representing the number of notes, followed by the notes themselves, concatenated into a single list called `song`.
2. Difference Calculation: It calculates the difference between each pair of consecutive notes, adjusted by adding 100 to ensure positivity and facilitate handling of transpositions.

Core Algorithm The algorithm employs binary search and hashing to find the longest repeating theme as follows:

- Binary Search: A binary search is conducted on the length of the theme, with the search range from 1 to half the total song length, as a theme cannot exceed half the song's length.
- Hashing for Subsequence Identification : For each candidate length k, the algorithm uses a rolling hash to efficiently compare subsequences of notes. We implement our hashing technique in the following way:
  - A rolling hash is computed for subsequences of length k, using a base of 26 and modulo $2^{63} - 1$ to minimize collisions and manage large numbers.
  - The hash map `seen` records the last index where each hash value occurred. A repeat of the same hash value at a distance greater than k indicates a repeating, transposable theme.
- Result Determination : The algorithm adjusts the binary search range based on the presence or absence of repeating themes and ultimately prints the length of the longest theme found, accounting for zero-based indexing.

Conclusion: This algorithm efficiently identifies the longest repeating theme in a musical melody by leveraging binary search to minimize the search space and rolling hashes for quick subsequence comparison.

Time Complexity : Input Processing:

- Reading and storing the input song requires $O(n)$ time, where n is the number of notes.
- Calculating the differences between consecutive notes also takes $O(n)$ time.

Binary Search on Theme Length:

- The algorithm uses binary search to find the length of the longest theme, with a time complexity of $O(\log n)$.

Hashing for Subsequence Identification:

- For each candidate length k, the algorithm computes a rolling hash for each subsequence, which takes $O(n+k)$ time per binary search iteration. However, considering the worst case where k is proportional to n, this step is $O(n)$.

Overall Time Complexity:

- Combining the binary search and hashing steps, the overall time complexity of the algorithm is $O(\log n \cdot n)$.

Assumptions:

- The analysis assumes hash map operations such as insertion and lookup are $O(1)$ on average.

Conclusion: The time complexity analysis reveals that the given code operates efficiently for sequences of moderate length, with an overall time complexity of $O(\log n \cdot n)$. This efficiency is achieved through the use of binary search and rolling hash techniques.

5.2 Another Dance Show : Submission ID - 248082215

I did a brute force solution with which i achieved 0.5 points. The following is the explanation of my approach. The algorithm comprises several key steps:

1. We sort the list of heights.
2. Iteratively selecting trios to minimize the height difference between the two shorter dancers.
3. Keeping track of selected dancers to avoid reselection.
4. Calculating the total disharmonious index for the formed trios.

We loop over all possible differences from 0,1,2 till we reach the number of groups or till we exhaust our list of dancers. We first select all duos with difference 0 and then with difference 1 and so on.

Time Complexity Analysis The time complexity of the algorithm is analyzed as follows:

- The main part of the algorithm involves nested loops, leading to a time complexity of $O(n^3)$ for trio formation.
- The final calculation of the disharmonious index is $O(m)$.

Overall Time Complexity: The dominant factor in the algorithm's time complexity is $O(n^3)$, attributed to the nested loops for trio formation.