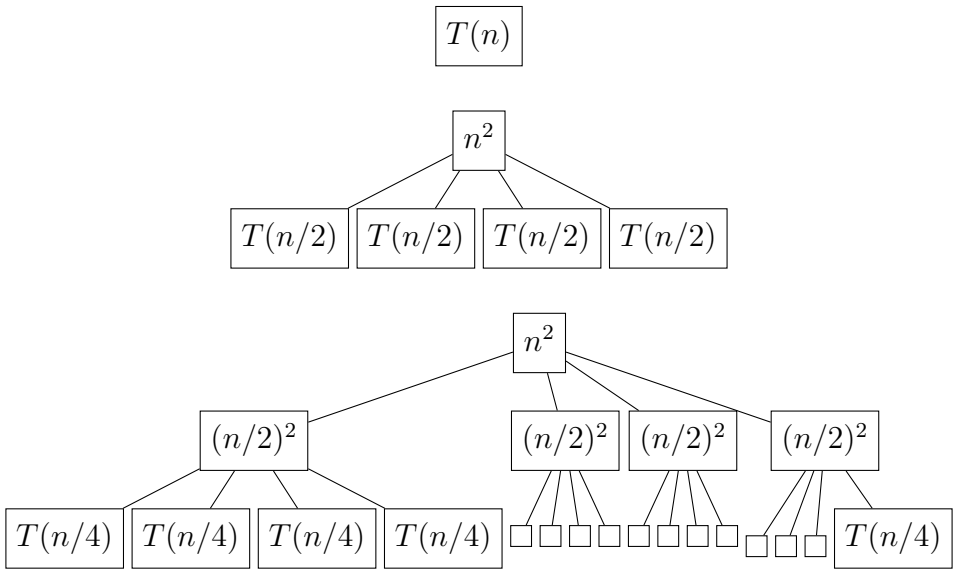
1 Problem 1 (15 points).

Use the recursion tree to solve the recurrence: $T(n) = 4T(n/2) + n^2$. You need to show the changes of the recursion tree step by step for at least three steps as what we did in the lecture slides (three pictures), and then show how you get the result by observing the recursion tree



1. Sum of elements on row 1: $n^2 = 4^0(n/2^0)^2$
2. Sum of elements on row 2: $4^1(n/2^1)^2 = 4n^2/4 = n^2$
3. Sum of elements on row 3: $4^2(n/2^2)^2 = 16n^2/16 = n^2$
4. Sum of elements on row k, where each element is size 1:
   $4^k(n/2^k)^2 = 4^k n^2/2^{2k} = 4^k n^2/(2^2)^k = 4^k n^2/4^k = n^2$
5. Total sum of all rows in the tree: kn^2
6. $T(n) = \Theta(kn^2) = \Theta(n^2)$, because k is a constant.

2 Problem 2 (15 points).

Use the inductive proof technique to show that $T(n) = 5T(n/4) + n^2 = O(n^2)$. That is, you need to find two positive constant $n0$ and c, such that when $n \geq n_0$, $T(n) \leq cn^2$.

Claim: If $T(n) = 5T(n/4) + n^2$, then $T(n) \leq cn^2$ for some constant c

Proof. Choose $c = 2$

1. Let $T(1) = 1$ (constant)
2. Base case $(n = n_0 = 4) \rightarrow T(4) = 5T(4/4) + 4^2 = 5(1) + 16 = 21 \leq 2(4^2) = 2(16) = 32$
3. Assume $T(n) \leq cn^2$ is true for all $n = 4, \ldots k$ for some $k \geq n_0$
4. Induction: When $n = k + 1$

$$\begin{aligned} T(k+1) &= 5T(\frac{k+1}{4}) + (k+1)^2 \\ &\leq c(\frac{k+1}{4})^2 + (k+1)^2 && \text{From assumption} \\ &\leq c\frac{(k+1)^2}{16} + (k+1)^2 \\ &\leq (k+1)^2(\frac{c}{16} + 1) \\ &\leq c(k+1)^2 \rightarrow cn^2 && \text{Ignore constant values in time complexity} \end{aligned}$$

5. This shows the claim is true when $n = k + 1$

□

3 Problem 3 (15 points; 3 points each).

Solve the following recurrences using the Master Theorem.

3.a $T(n) = 25T(n/5) + n^2 + \log n$

1. $a = 25, b = 5$
2. $n^{\log_b a} = n^{\log_5 25} = n^2$
3. $f(n) = n^2 + \log n = \Theta(n^2) = \Theta(n^{\log_b a})$
4. Case 2 is satisfied: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^2 \log n)$

3.b $T(n) = 25T(n/5) + 2n^3 + n\log n$

1. $a = 25, b = 25$
2. $n^{\log_b a} = n^{\log_5 25} = n^2$
3. $f(n) = 2n^3 + n\log n = \Omega(n^{\log_5 25+\epsilon})$ for any positive $\epsilon \leq 1$. (1)
4. $af(n/b) = 25(2(\frac{n}{5})^3 + \frac{n}{5}\log\frac{n}{5}) = 25(2n^3/125 + \frac{1}{5}n\log n/5) = 1/5(2n^3 + n\log n/5) \leq cf(n) = c(2n^3 + n\log n)$ for any positive constant $c > 1/5$ (2)
5. Case 3 is satisfied: $T(n) = \Theta(f(n)) = \Theta(2n^3 + n\log n) = \Theta(n^3)$

3.c $T(n) = 25T(n/5) + 3n^4 - 3n^2$

1. $a = 25, b = 5$
2. $n^{\log_b a} = n^{\log_5 25} = n^2$
3. $f(n) = 3n^4 - 3n^2 = \Omega(n^{\log_5 25+\epsilon})$ for any positive $\epsilon \leq 2$. (1)
4. $af(n/b) = 25(3(n/5)^4 - 3(n/5)^2) = 25(3n^4/625 - 3n^2/25) = 1/25(3n^4 - 3n^2) \leq cf(n) = c(3n^4 - 3n^2)$ for any positive constant $c > 1/25$ (2)
5. Case 3 is satisfied: $T(n) = \Theta(f(n)) = \Theta(3n^4 - 3n^2) = \Theta(n^4)$

3.d $T(n) = 125T(n/5) + 4n^2 + 5n\log n$

1. $a = 125, b = 5$
2. $n^{\log_b a} = n^{\log_5 125} = n^3$
3. $f(n) = 4n^2 + 5n\log n = O(n^{\log_5 125-\epsilon})$ for any positive $\epsilon \leq 1$.
4. Case 1 is satisfied: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

3.e $T(n) = 125T(n/5) + 5n^3 + 2n^2$

1. $a = 125, b = 5$
2. $n^{\log_b a} = n^{\log_5 125} = n^3$
3. $f(n) = 5n^3 + 2n^2 = \Theta(n^3) = \Theta(n^{\log_b a})$
4. Case 2 is satisfied: $T(n) = \Theta(n^{\log_b a}\log n) = \Theta(n^3\log n)$

4 Problem 4 (35 points).

Suppose you are given a BST by having the access to its root node. Every node of the BST has a left child link and right child link, but does NOT have a parent link. All the keys in the BST are distinct. Dr. Trouble wants you to convert this BST into a double linked list in a time-efficient manner, so that the nodes in the doubled linked list are in ascending order. Dr. Trouble also wants you to do it in a space-efficient manner. In particular, he does NOT allow you to allocate new space for the linked list, instead you are asked to recycle/reuse the existing tree nodes. In other words, you have to re-link all the tree nodes using the existing left/right child pointers to build up the double linked list.

Present your key idea and the algorithm's structure. You may not have to write every detail of the code in a real-world programming language, but must be precise and clear about the main algorithmic idea, and reason why your algorithm works. Give the pseudocode of your algorithm and its time complexity of your algorithm using the big-oh notation and make the bound as tight as possible.

READ NEXT PAGE

4.a Algorithm Idea

The goal to solve the problem is taking advantage of the way binary search trees work to recursively find the smallest element in each section of the tree. This information can be used to determine where the next link in the doubly linked list needs to go.

Two static *null* pointers, prev and cur, will be initialized before the recursive function starts. These pointers will aid in keeping track of which node of the tree needs to be linked to the next smallest element in ascending order.

The initial call of the recursive function is given the root of the tree, that is, the top-most element of the tree. After checking if it is empty, the function starts recursively calling itself using the current root's left child. This effectively splits the tree and the recursion is now operating off the smaller tree made up of the left side going down. Initially, this will keep being called recursively until reaching the bottom, finding the smallest element in the whole tree.

Here's where the pointers come in. the prev pointer gets set to cur, which is initially null, and cur gets set to the current root being dealt with. In this case, that root is the smallest element. the right element of prev is set to cur and cur's left link is set to prev *only* if prev is not null, meaning there is an element smaller than cur. Since prev is null when it's a the smallest element, this is never called meaning the left link of the smallest element remains null.

At this point, the function now starts recursively calling itself using the right side instead. This has the same effect of splitting the tree to a smaller section, but this time uses the right link instead of the left. (This follows divide and conquer since for every root element, there is a separate recursive call handling both the left and right sides of its child links) Since each time the recursion steps to the right, it follows the same trend of continuously calling the left side, it will find the next smallest element, and the locations of cur and prev will allow it to connect the previous "largest" element with the current smallest element of the right side.

This recursion will continue until reaching the far right side of the original root, in which case the null check will simply return and the entire function will end.

(note: A "smallest" variable can be used to store the smallest element once when prev is null for the first time, and can be used to replace where "root" is initially pointing, but is not included in the pseudocode as it is not directly relevant to the main funcionality of the algorithm)

4.b Pseudocode

```
prev = null, cur = null
    traverse(root)
      if(root is null) return // base case
      traverse(root.left) // recursive call to the left
      prev = cur
      cur = root // iterates between tree members
      if prev is not null
        prev.right = cur
        cur.left = prev // links the two together both ways
      traverse(root.right) // recursive call to the right
      return root
```

4.c Time complexity

The algorithm has a similar look to merge sort. Looking at the average case, the left and right side initially will have n/2 elements to go through, and the recursive calls get called twice per element. Additionally, this process will happen for every element in the tree, meaning the total time cost is $T(n) = 2T(n/2) + \Theta(n)$

The Master Theorem shows $n^{\log_b a} = n\log_2 2 = n$, and $f(n) = n$, so case 2 of the Master Theorem is satisfied, which makes the time complexity in big-O notation $O(n) = n \log n$.

4.d Code snippets

5 Problem 5 (20 points).

Conduct an independent study/research on the comparison between the hash tables and binary search trees in different scenarios. Do your research using whatever resource and discuss with your peers on this topic. Present whatever you find in your own language, and provide the source (url, book/article's title/author/year) of the ideas/opinions that you presented if they are not your owns

5.a Similarities

Hash tables and binary search trees have several similarities, in spite of operating very differently. For one, they both use a key system for making sure each element in the table/tree is unique. They both are used to store information, and they can also both be used for various data analysis functions such as search, insert, delete, etc.

5.b Differences

However, these two data structures have many more differences than similarities. This is because their intended functionalities are very different from each other.

The biggest difference is how the data is stored. in a hash map, a structure is created that simply stores keys and data. There's no sorting involved, no change in structure, just a table that can store values and keys. Compare this to binary search trees, which are obviously shaped like a tree. This tree structure is used to give the structure sorting functionality, where smaller values get sent to the left side and larger values to the right side. On top of sorting, this allows binary search trees to be iterated in a much more meaningful way. While hash tables don't really need to be iterated due to its capability of just having lookup functionality, BSTs can be iterated to find, store, and delete data in an algorithmic way, using the aforementioned tree structure.

Another major difference, which is contributed to by the differing structures, is how long various tasks take. While BSTs have time complexity that is based on the input size of how many values are stored, good hash tables are capable of having a constant time complexity no matter what the input size is. This makes hash tables generally much faster than BSTs, which is especially helpful if the more limited functionality of them is sufficient for what is needed.

A similar difference is the memory costs of the two. The major downside of hash tables, other than limited functionality, is that hash tables have to be a fixed size when created. This means that if a hash table is created larger than the number of elements needed to be stored to prevent collisions, the extra unused space ends up being wasted memory. Contrast this with BSTs, which are able to expand their size *as needed*. This is because each node is created with an empty pointer to the left and right, and these are simply reassigned to the new nodes that are added once the suitable location is determined by the sorting algorithm. So, while hash tables are typically faster, BSTs win in the memory usage.

Source: `https://www.baeldung.com/cs/hash-table-vs-balanced-binary-tree`