Tree Algorithms 10

The special properties of trees allow us to create algorithms that are specialized for trees and work more efficiently than general graph algorithms. This chapter presents a selection of such algorithms.

Section 10.1 introduces basic concepts and algorithms related to trees. A central problem is finding the diameter of a tree, i.e., the maximum distance between two nodes. We will learn two linear time algorithms for solving the problem.

Section 10.2 focuses on processing queries on trees. We will learn to use a tree traversal array to process various queries related to subtrees and paths. After this, we will discuss methods for determining lowest common ancestors, and an offline algorithm which is based on merging data structures.

Section 10.3 presents two advanced tree processing techniques: centroid decomposition and heavy-light decomposition.

10.1 Basic Techniques

A *tree* is a connected acyclic graph that consists of n nodes and $n - 1$ edges. Removing any edge from a tree divides it into two components, and adding any edge creates a cycle. There is always a unique path between any two nodes of a tree. The *leaves* of a tree are the nodes with only one neighbor.

As an example, consider the tree in Fig. 10.1. This tree consists of 8 nodes and 7 edges, and its leaves are nodes 3, 5, 7, and 8.

In a *rooted* tree, one of the nodes is appointed the *root* of the tree, and all other nodes are placed underneath the root. The lower neighbors of a node are called its *children*, and the upper neighbor of a node is called its *parent*. Each node has exactly one parent, except for the root that does not have a parent. The structure of a rooted

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_10

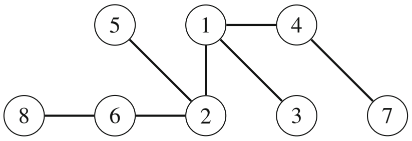


Fig. 10.1 A tree that consists of 8 nodes and 7 edges

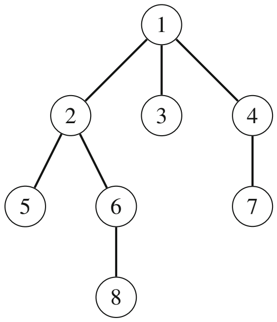


Fig. 10.2 A rooted tree where node 1 is the root node

tree is recursive: each node of the tree acts as the root of a *subtree* that contains the node itself and all nodes that are in the subtrees of its children.

For example, Fig. 10.2 shows a rooted tree where node 1 is the root of the tree. The children of node 2 are nodes 5 and 6, and the parent of node 2 is node 1. The subtree of node 2 consists of nodes 2, 5, 6, and 8.

10.1.1 Tree Traversal

General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree, and it is not possible to reach a node from more than one direction.

A typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
void dfs(int s, int e) {
    // process node s
    for (auto u : adj[s]) {
        if (u != e) dfs(u, s);
    }
}
```

The function is given two parameters: the current node *s* and the previous node *e*. The purpose of the parameter *e* is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x:

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic Programming Dynamic programming can be used to calculate some information during a tree traversal. For example, the following code calculates for each node s a value count[s]: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children, so we can calculate the number of nodes recursively as follows:

```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

Binary Tree Traversals In a binary tree, each node has a left and right subtree (which may be empty), and there are three popular tree traversal orderings:

- *pre-order*: first process the root node, then traverse the left subtree, then traverse the right subtree
- *in-order*: first traverse the left subtree, then process the root node, then traverse the right subtree
- *post-order*: first traverse the left subtree, then traverse the right subtree, then process the root node

For example, in Fig. 10.3, the pre-order is [1, 2, 4, 5, 6, 3, 7], the in-order is [4, 2, 6, 5, 1, 3, 7], and the post-order is [4, 6, 5, 2, 7, 3, 1].

If we know the pre-order and in-order of a tree, we can reconstruct its exact structure. For example, the only possible tree with pre-order [1, 2, 4, 5, 6, 3, 7] and

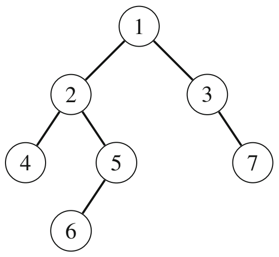


Fig. 10.3 A binary tree

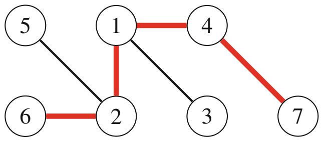


Fig. 10.4 A tree whose diameter is 4

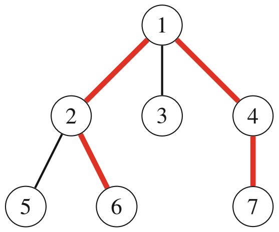


Fig. 10.5 Node 1 is the highest point on the diameter path

in-order [4, 2, 6, 5, 1, 3, 7] is shown in Fig. 10.3. The post-order and in-order also uniquely determine the structure of a tree. However, if we only know the pre-order and post-order, there may be more than one tree that matches the orderings.

10.1.2 Calculating Diameters

The *diameter* of a tree is the maximum length of a path between two nodes. For example, Fig. 10.4 shows a tree whose diameter is 4 that corresponds to a path of length 4 between nodes 6 and 7. Note that the tree also has another path of length 4 between nodes 5 and 7.

Next we will discuss two $O(n)$ time algorithms for calculating the diameter of a tree. The first algorithm is based on dynamic programming, and the second algorithm uses depth-first searches.

First Algorithm A general way to approach tree problems is to first root the tree arbitrarily and then solve the problem separately for each subtree. Our first algorithm for calculating diameters is based on this idea.

An important observation is that every path in a rooted tree has a *highest point*: the highest node that belongs to the path. Thus, we can calculate for each node x the length of the longest path whose highest point is x. One of those paths corresponds to the diameter of the tree. For example, in Fig. 10.5, node 1 is the highest point on the path that corresponds to the diameter.

We calculate for each node x two values:

- `toLeaf`(x): the maximum length of a path from x to any leaf
- `maxLength`(x): the maximum length of a path whose highest point is x

For example, in Fig. 10.5, `toLeaf`$(1) = 2$, because there is a path $1 \to 2 \to 6$, and `maxLength`$(1) = 4$, because there is a path $6 \to 2 \to 1 \to 4 \to 7$. In this case, `maxLength`$(1)$ equals the diameter.

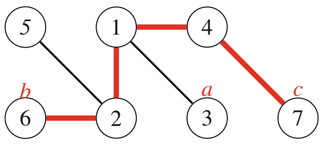


Fig. 10.6 Nodes a, b, and c when calculating the diameter

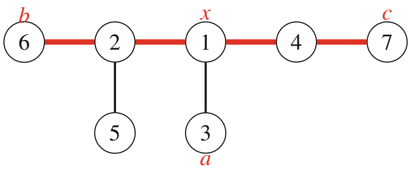


Fig. 10.7 Why does the algorithm work?

Dynamic programming can be used to calculate the above values for all nodes in $O(n)$ time. First, to calculate `toLeaf`(x), we go through the children of x, choose a child c with the maximum `toLeaf`(c), and add one to this value. Then, to calculate `maxLength`(x), we choose two distinct children a and b such that the sum `toLeaf`(a) + `toLeaf`(b) is maximum and add two to this sum. (The cases where x has less than two children are easy special cases.)

Second Algorithm Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

For example, Fig. 10.6 shows a possible way to select nodes a, b, and c when calculating the diameter for our example tree.

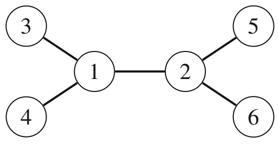
This is an elegant method, but why does it work? It helps to draw the tree so that the path that corresponds to the diameter is horizontal and all other nodes hang from it (Fig. 10.7). Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b, node c, or some other node that is at least as far from node x. Thus, this node is always a valid choice for an endpoint of a path that corresponds to the diameter.

10.1.3 All Longest Paths

Our next problem is to calculate for every tree node x a value `maxLength`(x): the maximum length of a path that begins at node x. For example, Fig. 10.8 shows a tree and its `maxLength` values. This can be seen as a generalization of the tree diameter problem, because the largest of those lengths equals the diameter of the tree. Also, this problem can be solved in $O(n)$ time.

Once again, a good starting point is to root the tree arbitrarily. The first part of the problem is to calculate for every node x the maximum length of a path that goes *downwards* through a child of x. For example, the longest path from node 1 goes

Fig. 10.8 Calculating maximum path lengths



$\texttt{maxLength}(1) = 2$
$\texttt{maxLength}(2) = 2$
$\texttt{maxLength}(3) = 3$
$\texttt{maxLength}(4) = 3$
$\texttt{maxLength}(5) = 3$
$\texttt{maxLength}(6) = 3$

Fig. 10.9 The longest path that starts at node 1

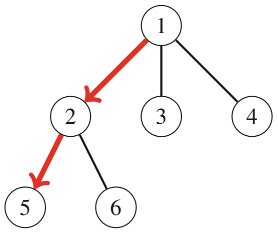


Fig. 10.10 The longest path from node 3 goes through its parent

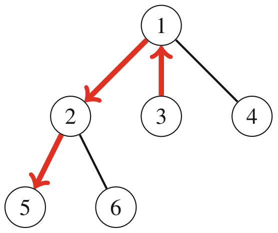
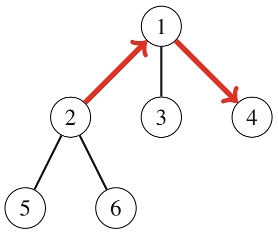


Fig. 10.11 In this case, the second longest path from the parent should be chosen



through its child 2 (Fig. 10.9). This part is easy to solve in $O(n)$ time, because we can use dynamic programming as we have done previously.

Then, the second part of the problem is to calculate for every node x the maximum length of a path *upwards* through its parent p. For example, the longest path from node 3 goes through its parent 1 (Fig. 10.10). At first glance, it seems that we should first move to p and then choose the longest path (upwards or downwards) from p. However, this *does not* always work, because such a path may go through x (Fig. 10.11). Still, we can solve the second part in $O(n)$ time by storing the maximum lengths of *two* paths for each node x:

- $\texttt{maxLength}_1(x)$: the maximum length of a path from x to a leaf
- $\texttt{maxLength}_2(x)$ the maximum length of a path from x to a leaf, in another direction than the first path

For example, in Fig. 10.11, $\texttt{maxLength}_1(1) = 2$ using the path $1 \to 2 \to 5$, and $\texttt{maxLength}_2(1) = 1$ using the path $1 \to 3$.

Finally, to determine the maximum-length path from node x upwards through its parent p, we consider two cases: if the path that corresponds to $\texttt{maxLength}_1(p)$ goes through x, the maximum length is $\texttt{maxLength}_2(p) + 1$ and otherwise the maximum length is $\texttt{maxLength}_1(p) + 1$.

10.2 Tree Queries

In this section we focus on processing *queries* on rooted trees. Such queries are typically related to subtrees and paths of the tree, and they can be processed in constant or logarithmic time.

10.2.1 Finding Ancestors

The kth *ancestor* of a node x in a rooted tree is the node that we will reach if we move k levels up from x. Let $\texttt{ancestor}(x, k)$ denote the kth ancestor of a node x (or 0 if there is no such an ancestor). For example, in Fig. 10.12, $\texttt{ancestor}(2, 1) = 1$ and $\texttt{ancestor}(8, 2) = 4$.

An easy way to calculate any value of $\texttt{ancestor}(x, k)$ is to perform a sequence of k moves in the tree. However, the time complexity of this method is $O(k)$, which may be slow, because a tree of n nodes may have a path of n nodes.

Fortunately, we can efficiently calculate any value of $\texttt{ancestor}(x, k)$ in $O(\log k)$ time after preprocessing. As in Sect. 7.5.1, the idea is to first precalculate all values of $\texttt{ancestor}(x, k)$ where k is a power of two. For example, the values for the tree in Fig. 10.12 are as follows:

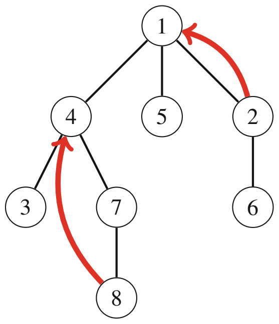


Fig. 10.12 Finding ancestors of nodes

x	1	2	3	4	5	6	7	8
`ancestor`$(x, 1)$	0	1	4	1	1	2	4	7
`ancestor`$(x, 2)$	0	0	1	0	0	1	1	4
`ancestor`$(x, 4)$	0	0	0	0	0	0	0	0
...								

Since we know that a node always has less than n ancestors, it suffices to calculate $O(\log n)$ values for each node and the preprocessing takes $O(n \log n)$ time. After this, any value of `ancestor`(x, k) can be calculated in $O(\log k)$ time by representing k as a sum where each term is a power of two.

10.2.2 Subtrees and Paths

A *tree traversal array* contains the nodes of a rooted tree in the order in which a depth-first search from the root node visits them. For example, Fig. 10.13 shows a tree and the corresponding tree traversal array.

An important property of tree traversal arrays is that each subtree of a tree corresponds to a subarray in the tree traversal array such that the first element of the subarray is the root node. For example, Fig. 10.14 shows the subarray that corresponds to the subtree of node 4.

Subtree Queries Suppose that each node in the tree is assigned a value and our task is to process two types of queries: updating the value of a node and calculating the sum of values in the subtree of a node. To solve the problem, we construct a tree traversal array that contains three values for each node: the identifier of the node, the size of the subtree, and the value of the node. For example, Fig. 10.15 shows a tree and the corresponding array.

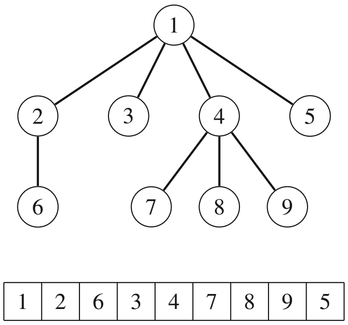


Fig. 10.13 A tree and its tree traversal array

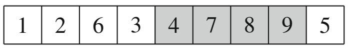
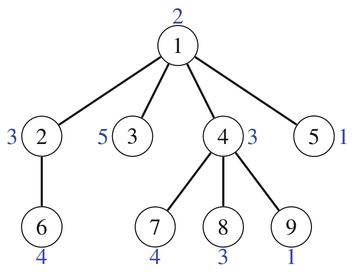


Fig. 10.14 The subtree of node 4 in the tree traversal array



node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

Fig. 10.15 A tree traversal array for calculating subtree sums

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

Fig. 10.16 Calculating the sum of values in the subtree of node 4

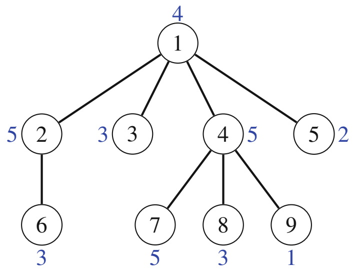
Using this array, we can calculate the sum of values in any subtree by first determining the size of the subtree and then summing up the values of the corresponding nodes. For example, Fig. 10.16 shows the values that we access when calculating the sum of values in the subtree of node 4. The last row of the array tells us that the sum of values is $3 + 4 + 3 + 1 = 11$.

To answer queries efficiently, it suffices to store the last row of the array in a binary indexed or segment tree. After this, we can both update a value and calculate the sum of values in $O(\log n)$ time.

Path Queries Using a tree traversal array, we can also efficiently calculate sums of values on paths from the root node to any node of the tree. As an example, consider a problem where our task is to process two types of queries: updating the value of a node and calculating the sum of values on a path from the root to a node.

To solve the problem, we construct a tree traversal array that contains for each node its identifier, the size of its subtree, and the sum of values on a path from the root to the node (Fig. 10.17). When the value of a node increases by x, the sums of all nodes in its subtree increase by x. For example, Fig. 10.18 shows the array after increasing the value of node 4 by 1.

To support both the operations, we need to be able to increase all values in a range and retrieve a single value. This can be done in $O(\log n)$ time using a binary indexed or segment tree and a difference array (see Sect. 9.2.3).



node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	9	14	12	10	6

Fig. 10.17 A tree traversal array for calculating path sums

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	10	15	13	11	6

Fig. 10.18 Increasing the value of node 4 by 1

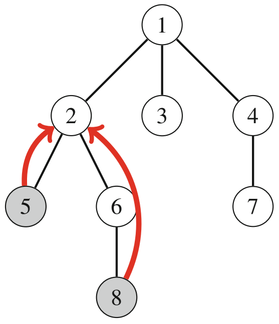


Fig. 10.19 The lowest common ancestor of nodes 5 and 8 is node 2

10.2.3 Lowest Common Ancestors

The *lowest common ancestor* of two nodes of a rooted tree is the lowest node whose subtree contains both the nodes. For example, in Fig. 10.19 the lowest common ancestor of nodes 5 and 8 is node 2.

A typical problem is to efficiently process queries that require us to find the lowest common ancestor of two nodes. Next we will discuss two efficient techniques for processing such queries.

First Method Since we can efficiently find the kth ancestor of any node in the tree, we can use this fact to divide the problem into two parts. We use two pointers that initially point to the two nodes whose lowest common ancestor we should find.

First, we make sure that the pointers point to nodes at the same level in the tree. If this is not the case initially, we move one of the pointers upwards. After this, we

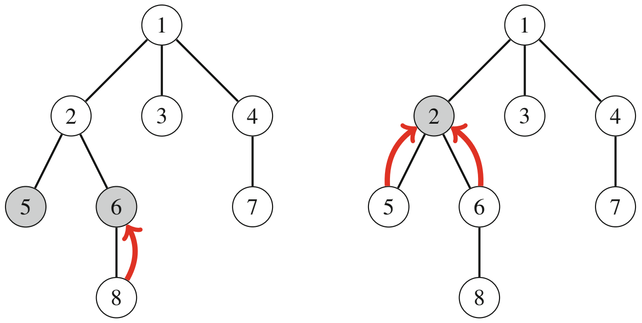


Fig. 10.20 Two steps to find the lowest common ancestor of nodes 5 and 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Fig. 10.21 An extended tree traversal array for processing lowest common ancestor queries

determine the minimum number of steps needed to move both pointers upwards so that they will point to the same node. The node to which the pointers point after this is the lowest common ancestor. Since both parts of the algorithm can be performed in $O(\log n)$ time using precomputed information, we can find the lowest common ancestor of any two nodes in $O(\log n)$ time.

Figure 10.20 shows how we can find the lowest common ancestor of nodes 5 and 8 in our example scenario. First, we move the second pointer one level up so that it points to node 6 which is at the same level with node 5. Then, we move both pointers one step upwards to node 2, which is the lowest common ancestor.

Second Method Another way to solve the problem, proposed by Bender and Farach-Colton [3], is based on an extended tree traversal array, sometimes called an *Euler tour tree*. To construct the array, we go through the tree nodes using depth-first search and add each node to the array *always* when the depth-first search walks through the node (not only at the first visit). Hence, a node that has k children appears $k + 1$ times in the array, and there are a total of $2n - 1$ nodes in the array. We store two values in the array: the identifier of the node and the depth of the node in the tree. Figure 10.21 shows the resulting array in our example scenario.

Now we can find the lowest common ancestor of nodes a and b by finding the node with the *minimum* depth between nodes a and b in the array. For example, Fig. 10.22 shows how to find the lowest common ancestor of nodes 5 and 8. The minimum-depth node between them is node 2 whose depth is 2, so the lowest common ancestor of nodes 5 and 8 is node 2.

Note that since a node may appear several times in the array, there may be multiple ways to choose the positions of nodes a and b. However, any choice correctly determines the lowest common ancestor of the nodes.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1
				↑											

Fig. 10.22 Finding the lowest common ancestor of nodes 5 and 8

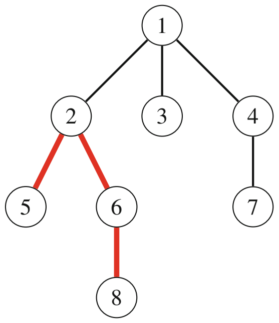


Fig. 10.23 Calculating the distance between nodes 5 and 8

Using this technique, to find the lowest common ancestor of two nodes, it suffices to process a range minimum query. A usual way is to use a segment tree to process such queries in $O(\log n)$ time. However, since the array is static, we can also process queries in $O(1)$ time after an $O(n \log n)$ time preprocessing.

Calculating Distances Finally, consider the problem of processing queries where we need to calculate the distance between nodes a and b (i.e., the length of the path between a and b). It turns out that this problem reduces to finding the lowest common ancestor of the nodes. First, we root the tree arbitrarily. After this, the distance of nodes a and b can be calculated using the formula

$$\texttt{depth}(a) + \texttt{depth}(b) - 2 \cdot \texttt{depth}(c),$$

where c is the lowest common ancestor of a and b.

For example, to calculate the distance between nodes 5 and 8 in Fig. 10.23, we first determine that the lowest common ancestor of the nodes is node 2. Then, since the depths of the nodes are $\texttt{depth}(5) = 3$, $\texttt{depth}(8) = 4$, and $\texttt{depth}(2) = 2$, we conclude that the distance between nodes 5 and 8 is $3 + 4 - 2 \cdot 2 = 3$.

10.2.4 Merging Data Structures

So far, we have discussed *online* algorithms for tree queries. Those algorithms are able to process queries one after another in such a way that each query is answered before receiving the next query. However, in many problems, the online property is not necessary, and we may use *offline* algorithms to solve them. Such algorithms

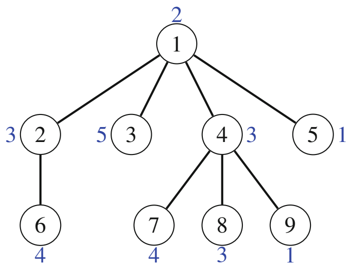


Fig. 10.24 The subtree of node 4 contains two nodes whose value is 3

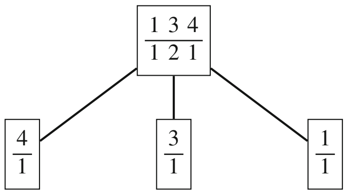


Fig. 10.25 Processing queries using map structures

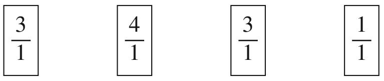


Fig. 10.26 Merging map structures at a node

are given a complete set of queries which can be answered in any order. Offline algorithms are often easier to design than online algorithms.

One method to construct an offline algorithm is to perform a depth-first tree traversal and maintain data structures in nodes. At each node s, we create a data structure $\mathtt{d}[s]$ that is based on the data structures of the children of s. Then, using this data structure, all queries related to s are processed.

As an example, consider the following problem: We are given a rooted tree where each node has some value. Our task is to process queries that ask to calculate the number of nodes with value x in the subtree of node s. For example, in Fig. 10.24, the subtree of node 4 contains two nodes whose value is 3.

In this problem, we can use map structures to answer the queries. For example, Fig. 10.25 shows the maps for node 4 and its children. If we create such a data structure for each node, we can easily process all given queries, because we can handle all queries related to a node immediately after creating its data structure.

However, it would be too slow to create all data structures from scratch. Instead, at each node s, we create an initial data structure $\mathtt{d}[s]$ that only contains the value of s. After this, we go through the children of s and *merge* $\mathtt{d}[s]$ and all data structures $\mathtt{d}[u]$ where u is a child of s. For example, in the above tree, the map for node 4 is created by merging the maps in Fig. 10.26. Here the first map is the initial data structure for node 4, and the other three maps correspond to nodes 7, 8, and 9.

The merging at node s can be done as follows: We go through the children of s and at each child u merge $\mathtt{d}[s]$ and $\mathtt{d}[u]$. We always copy the contents from $\mathtt{d}[u]$ to $\mathtt{d}[s]$. However, before this, we *swap* the contents of $\mathtt{d}[s]$ and $\mathtt{d}[u]$ if $\mathtt{d}[s]$ is smaller than d[u]. By doing this, each value is copied only $O(\log n)$ times during the tree traversal, which ensures that the algorithm is efficient.

To swap the contents of two data structures a and b efficiently, we can just use the following code:

```
swap(a,b);
```

It is guaranteed that the above code works in constant time when a and b are C++ standard library data structures.

10.3 Advanced Techniques

In this section, we discuss two advanced tree processing techniques. Centroid decomposition divides a tree into smaller subtrees and processes them recursively. Heavy-light decomposition represents a tree as a set of special paths, which allows us to efficiently process path queries.

10.3.1 Centroid Decomposition

A *centroid* of a tree of n nodes is a node whose removal divides the tree into subtrees each of which contains at most $\lfloor n/2 \rfloor$ nodes. Every tree has a centroid, and it can be found by rooting the tree arbitrarily and always moving to the subtree that has the maximum number of nodes, until the current node is a centroid.

In the *centroid decomposition* technique, we first locate a centroid of the tree and process all paths that go through the centroid. After this, we remove the centroid from the tree and process the remaining subtrees recursively. Since removing the centroid always creates subtrees whose size is at most half of the size of the original tree, the time complexity of such an algorithm is $O(n \log n)$, provided that we can process each subtree in linear time.

For example, Fig. 10.27 shows the first step of a centroid decomposition algorithm. In this tree, node 5 is the only centroid, so we first process all paths that go through

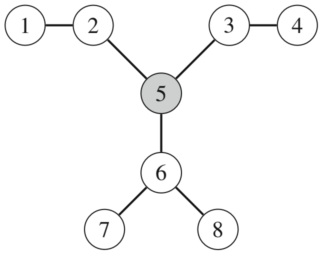


Fig. 10.27 Centroid decomposition

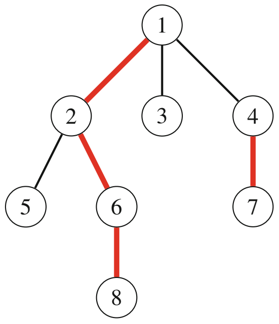


Fig. 10.28 Heavy-light decomposition

node 5. After this, node 5 is removed from the tree, and we process the three subtrees $\{1, 2\}$, $\{3, 4\}$, and $\{6, 7, 8\}$ recursively.

Using centroid decomposition, we can, for example, efficiently calculate the number of paths of length x in a tree. When processing a tree, we first find a centroid and calculate the number of paths that go through it, which can be done in linear time. After this, we remove the centroid and recursively process the smaller trees. The resulting algorithm works in $O(n \log n)$ time.

10.3.2 Heavy-Light Decomposition

Heavy-light decomposition[1] divides the nodes of a tree into a set of paths that are called *heavy* paths. The heavy paths are created so that a path between any two tree nodes can be represented as $O(\log n)$ subpaths of heavy paths. Using the technique, we can manipulate nodes on paths between tree nodes almost like elements in an array, with only an additional $O(\log n)$ factor.

To construct the heavy paths, we first root the tree arbitrarily. Then, we start the first heavy path at the root of the tree and always move to a node that has a maximum-size subtree. After this, we recursively process the remaining subtrees. For example, in Fig. 10.28, there are four heavy paths: 1–2–6–8, 3, 4–7, and 5 (note that two of the paths only have one node).

Now, consider any path between two nodes in the tree. Since we always chose the maximum-size subtree when creating heavy paths, this guarantees that we can divide the path into $O(\log n)$ subpaths so that each of them is a subpath of a single heavy path. For example, in Fig. 10.28, the path between nodes 7 and 8 can be divided into two heavy subpaths: first 7–4, then 1–2–6–8.

The benefit of heavy-light decomposition is that each heavy path can be treated like an array of nodes. For example, we can assign a segment tree for each heavy path and support sophisticated path queries, such as calculating the minimum node value in a path or increasing the value of every node in a path. Such queries can be

[1] Sleator and Tarjan [29] introduced the idea in the context of their link/cut tree data structure.

processed in $O(\log^2 n)$ time,[2] because each path consists of $O(\log n)$ heavy paths and each heavy path can be processed in $O(\log n)$ time.

While many problems can be solved using heavy-light decomposition, it is good to keep in mind that there is often another solution that is easier to implement. In particular, the techniques presented in Sect. 10.2.2 can often be used instead of heavy-light decomposition.

[2]The notation $\log^k n$ corresponds to $(\log n)^k$.