Design and Implementation of a link/cut tree in C++

Bachelor Thesis

submitted: May 2021

by: Felix Krömer
mail@felixkroemer.com
born March 29th 1999
in Stadthagen

Student ID Number: 1634604

University of Mannheim
Chair of Practical Computer Science III
D – 68159 Mannheim
Phone: +49 621 181–2587, Fax +49 621 181–2588
Internet: `https://www.wim.uni-mannheim.de/moerkotte`

Contents

1 Introduction

Algorithms that are used to solve problems in computer science often require the use of special data structures that allow them to run efficiently. Dinic's algorithm, for example, is concerned with solving the maximum flow problem, which is finding the maximum possible flow rate in a flow network. A flow network is a directed graph where each edge has a capacity and there exist two special nodes called a source, which only has outgoing edges, and a sink, which only has incoming edges. The goal is now to maximize the flow from the source to the sink under the constraints that the capacity of each edge is not exceeded and that the sum of flows entering a node is equal to the sum of flows leaving it. The source and sink are exempt from the last rule. [3]

On its own the algorithm has a running time of $O(V^2 * E)$ where V is the number of nodes in the graph and E is the number of edges. However, it can be improved to run in $O(V * E * log(V))$ if it is used in conjunction with a data structure that allows for the efficient management of a set of trees, also called a forest. The trees in the forest need to be able to merge and split by adding or removing edges between them. One data structure that is able to efficiently perform these types of operations is the link/cut tree. It maintains a set of rooted trees in which two trees can be linked to combine them into one or cut which achieves the opposite effect. Besides these, many more operations are possible. For example, the root of a tree can be found, the lowest common ancestor of two nodes can be determined, or it can be checked if one node is the descendant of another node. While its most significant application is solving numerous network flow problems, such as the maximum flow problem, the link/cut tree also has applications in solving the dynamic connectivity problem. [8, 5]

The goal of this paper is to design and implement a link/cut tree in C++. The implementation will not only comprise the basic functionality but also more advanced features such as the ability to maintain subtree information. The first chapter will explain the structure and operations of the link/cut tree. The second will deal with the details of its implementation and the third and final chapter will evaluate the implementation.

2 Link/Cut tree

2.1 Structure and Intention

The Link/Cut tree is a data structure that was invented by Sleator and Tarjan in 1982 [8] and can be used for the efficient management of a set of rooted trees, which is also referred to as a forest. The set of trees can be modified using the following operations, which will be explained in detail in later sections. This first section will focus on the structure of the trees in the forest but not on their operations.

- **createTree()** - Add a new tree to the forest that consists of a single node.
- **link(p, q)** - If p is the root of a tree in the forest, connect the subtree rooted at p to q by adding an edge. This assumes that p and q do not belong to the same tree.
- **cut(p)** - Disconnect the subtree rooted at p and add the resulting tree to the set of trees. If p is the root of a tree, the cut operation has no effect.

The following operations can be used to analyze the trees in the forest. While they do no change the structure of a tree as it presents itself to the user, they can have an impact on the inner representation of the tree.

- **findRoot(p)** - Return the root of the tree that p belongs to.
- **lowestCommonAncestor(p, q)** - If p and q are part of the same tree, return their lowest common ancestor.
- **isDescendant(p, q)** - If p and q are part of the same tree, test if p is a descendant of q.

All operations above run in *O(log(n))* amortized time. This means that in a worst case series of operations the average cost of a single operation is in *O(log(n))*. Some operations may have a higher cost, but they will usually lead to a lower cost for subsequent operations.

The findRoot operation is especially important as it can be used to determine whether two nodes are part of the same tree by comparing the output of *findRoot* on both nodes. It also helps to show why a trivial implementation of trees using parent pointers is usually not sufficient in many applications. If such an implementation was used, finding the root would take time equivalent to the length of the path from a node p to its root, which would lead to a worst case running time of *O(n)* where n is the number of nodes in the tree. While similarly expensive operations can occur in the link/cut tree, they usually reduce the cost of the following operations. The same does not apply to the trivial implementation, where the cost of *findRoot* is always proportional to the depth of a node.[8, 9]

2.1.1 Represented tree

To solve the aforementioned problem and reduce the cost of finding the root of a tree that a node belongs to, the tree is partitioned into vertex-disjoint paths and each path is represented by an auxiliary data structure. The splay tree is the most commonly used data structure for this purpose. For now, while only the structure and not the operations of the link/cut tree are examined, it is sufficient to regard it as a regular binary search tree, where each tree node stores a reference to its left child, right child, and parent. What differentiates it from a generic binary search tree will be explained in Section 2.2.

When decomposing a tree into paths, the child of a node that is on the same path as its parent is referred to as the *preferred child* and it is connected to its parent via a *preferred edge*. Accordingly, the paths that the tree is decomposed into are called *preferred paths*. All siblings of a preferred child are connected to their parent via normal, *non-preferred edges*. Because these siblings are the topmost nodes in their respective preferred paths, their parent is called the *path-parent*. Thus, every topmost node in a preferred path, besides the root, is a non-preferred child and it is connected to the path-parent via a non-preferred edge. If a child node is connected to its parent via a non-preferred edge, it is not on the same path and therefore not stored in the same auxiliary data structure. Preferred paths can have a length of one if a node is connected to its parent via a non-preferred edge and it has no preferred child. This means that a node can have children, but none of whom have to be a preferred child. Figure 2.1 shows one such decomposition of a tree into preferred paths.

To the user, the distinction between preferred and non-preferred edges is irrelevant. He is only presented with an ordinary tree structure that supports the operations listed above. It does not have to be balanced and any node can have an arbitrary number of children. The tree described above, which the user directly interacts with, is referred to as the *represented tree.* It is also sometimes called the *abstract* or *real* tree. Its name indicates that it is represented by a separate data structure, which is the *virtual tree.* It is called virtual because it resembles how the tree is stored in memory and its operations are therefore very similar to the implementation of the link/cut tree in code. While the represented and the virtual tree share the same nodes and carry the same information, they are linked up differently. If two nodes are connected in the represented tree, either via a preferred or a non-preferred edge, it does not mean that these nodes share an edge in the virtual tree or that a pointer between such nodes exists in memory. Hence, a link/cut tree can take two forms, represented and virtual. Both describe the same link/cut tree but utilize different concepts and terminology. [1]

The modifier *real* will be used to denote relations between nodes on the represented tree. Thus, a *real parent* is the parent of a node in the represented tree. The same concept applies to terms such as children, ancestors, descendants, subtrees and other terms that are commonly used in connection with tree data structures. It may be omitted if the intent is clear and the possibility of confusion with the virtual tree is limited. Also, the terms preferred, non-preferred, or path-parent only apply to the represented tree. Only the represented tree is decomposed into preferred paths and only nodes within the represented tree are connected using preferred and non-preferred edges. The following section will explain the virtual tree and its relation to the represented tree in more detail. The terms that are introduced there do not apply to the represented tree but only to the virtual tree.

2.1.2 Virtual tree

The virtual tree is a tree of splay trees, in which each splay tree resembles one preferred path in the represented tree. How a splay tree can be used to describe a path will be explained at the end of this section. Nodes in the virtual tree are connected via *dashed and solid edges*, where solid edges connect nodes within splay trees, while dashed edges connect nodes across splay trees. Because every

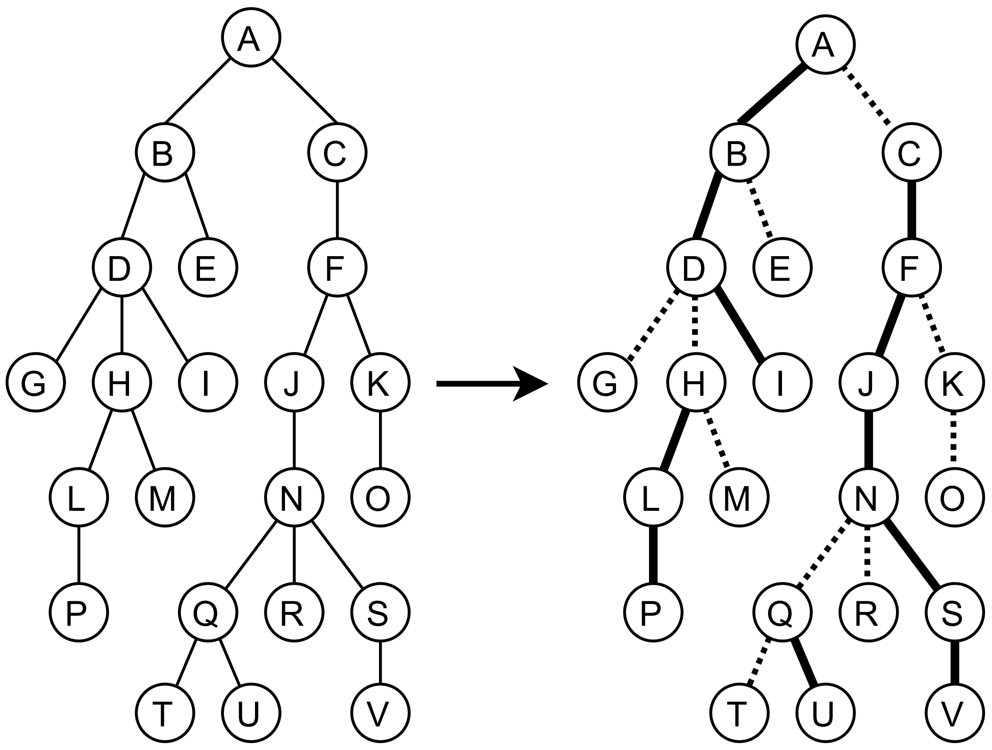

Figure 2.1: A represented tree on the left is partitioned into preferred paths on the right. Preferred edges are represented by solid edges and non-preferred edges are represented by dashed edges. The root node A is on the same preferred path as nodes B, D and I. The nodes C, F, J, N, S and V are on a separate preferred path. Node C is connected to the path-parent via a non-preferred edge and it is a non-preferred child. Node G is on a preferred path that only includes itself.

node in the virtual tree belongs to a splay tree and splay trees are binary trees, every node in the virtual tree can have a left and a right child. If a node in the virtual tree has a left or a right child, they are stored in the same splay tree and are connected to their parent via a solid edge. Because children in a splay tree have a pointer to their parent, solid edges can be considered bidirectional. Therefore, a solid edge between two nodes indicates that both nodes point at each other. [6]

On top of the left and right children, a node in the virtual tree can have an arbitrary number of children that it has no reference to. This is the case if the parent of a node belongs to a separate splay tree. The parent has no way of knowing about this additional child because it only has pointers to children that are on the same splay tree. The only node whose parent can be on another splay tree is the root and the edge that connects a root in a splay tree to its parent is a dashed edge. It can be regarded as directed and will be depicted as such because only one node points to the other and the other node does not know it is being pointed at. Although the virtual tree represents how the tree is stored in memory, there is no notion of preferred or non-preferred edges in code, but only pointers. Preferred and non-preferred edges only reflect if both nodes point at each other or only one node points at the other. In the code, every node stores its key as well as pointers to its left child, right child, and parent.

Similar to the term *real*, the term *solid* will be used to denote relations between nodes that are in the same splay tree. Thus, a node's solid children may include its left child, right child, or both. If a child is called the left or right child, then the term *solid* will be omitted because it is apparent that a child on the same splay tree is implied. The concept of left and right children does not apply to nodes on the represented tree. Also, children of a node in the virtual tree that are not solid children will be referred to as *dashed children.* A node in the virtual tree can therefore have at most two solid children, but an arbitrary number of dashed children, which connect to it via dashed edges.

Dashed edges, in the virtual tree, can be likened to non-preferred edges, found in the represented tree, in that they connect two splay trees, which correspond to two preferred paths. The difference is that the non-preferred edge and the dashed edge connect different kinds of nodes. In the represented tree, a non-preferred edge connects the topmost node in a preferred path to the path-parent, while in the virtual tree a dashed edge connects the root of a splay tree to the path-parent. The root of the splay tree is usually not the topmost node in the path that it resembles. This can be shown by explaining how a splay tree, and binary search trees in general, can be used to resemble sequences, or in the context of the link/cut tree, paths in a tree.

In binary search trees every node stores a key that has a greater value than any key in its left subtree and that has a smaller value than any key in its right subtree. In the context of the link/cut tree, the value of the key of a node corresponds to its height in the preferred path. Therefore, every node that is located in the left subtree of an element in the splay tree can be regarded as an ancestor of said element, while an element in the right subtree can be seen as its descendant. The sequence that the tree resembles can thus be obtained by recursively determining the ordered sequence of the left subtree, appending the element and then appending the ordered sequence of the right subtree. Of course it is equally viable to regard nodes in the left subtree of a node as being descendants and the nodes in the right subtree as ancestors, but in the following sections only the earlier notion will be applied. To convert the virtual tree into the represented tree, every splay tree can be converted to a path. The represented tree can then be assembled by connecting the topmost node in each path to its path-parent, which is the parent of the root in the corresponding splay tree.

Figure 2.3 shows how a represented tree can be portrayed as a tree of splay trees. Every preferred path is represented by a splay tree where the order of the splay tree corresponds to the sequence of nodes in the preferred path. Splay trees with a different structure but the same nodes can yield the same preferred path. For example, there are a total of five different splay trees that convert to the sequence of the preferred path with the nodes H,L,P, if they are traversed inorder using the method described above. They are depicted in Figure 2.2. The splay tree with the number 5 is the one that can be found in Figure 2.3. The number of splay trees with n nodes that can convert to the same sequence is the Catalan number of $n + 1$.

Almost all operations that are supported by the link/cut tree make use of a secondary operation called *expose*. While it does not change the structure of the represented tree, it determines the partitioning into preferred paths and therefore has a significant impact on the structure of the virtual tree. Its functionality

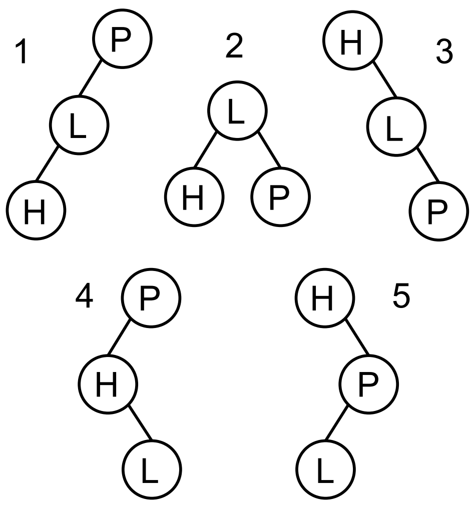

Figure 2.2: The five different types of splay trees that all resemble the sequence H, L, P

largely depends on the type of auxiliary data structure that is used and its explanation, along with explanations of the operations that depend on it, will be postponed until after the splay tree has been introduced.

2.2 Splay trees

The splay tree is a data structure that was proposed by Sleator and Tarjan in 1985. [7] It is a type of binary search tree that ensures that recently and frequently accessed elements are located close to the root, while elements that are rarely accessed tend to have have a higher depth. Because frequently accessed elements are kept close to the root, they can be easily accessed again in future operations. This property is achieved by splaying an element, which is moving it to the root of its tree whenever it is accessed. This operation may have a negative impact on the balance of the tree. Unlike balanced trees, such as Red-Black or AVL trees, the splay tree does not guarantee that the height difference between its nodes always stays within a certain limit. After a splay has been performed, the tree may be more unbalanced than it was before the operation. Through a series of splay operations the tree may even become degenerate, in which case the cost of a single splay operation would be equivalent to the number of nodes in the tree. Balanced trees perform restructuring operations whenever necessary to make sure that the restrictions that they put on the structure of the tree always

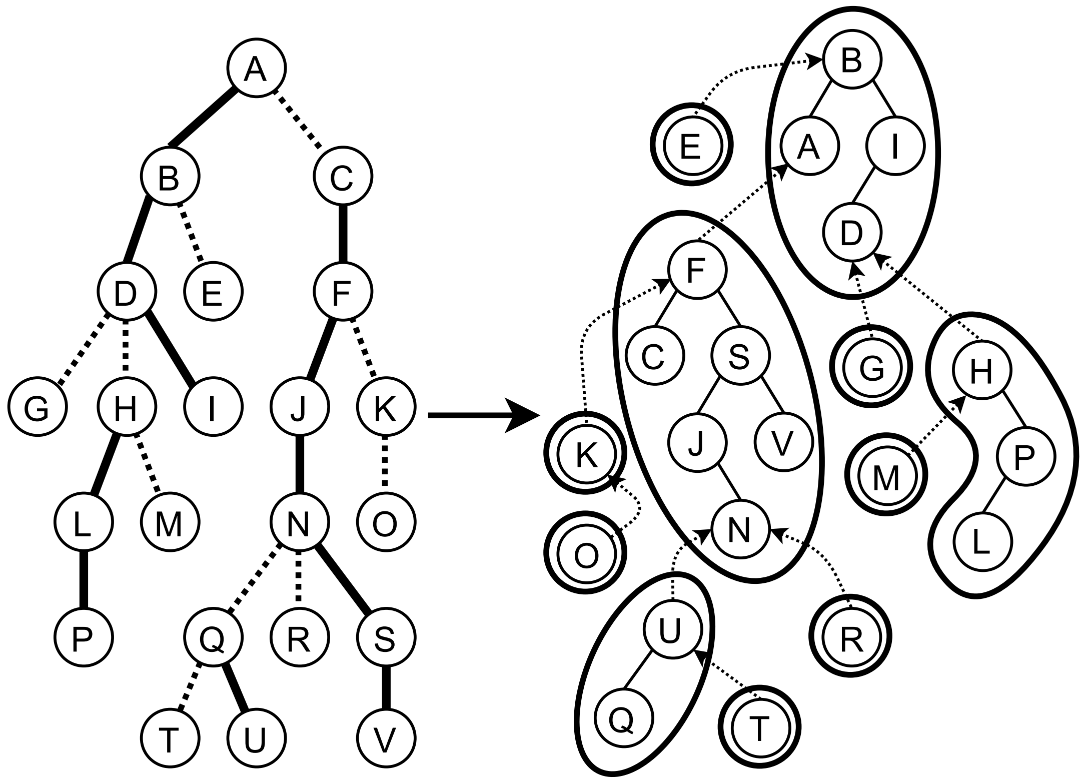

Figure 2.3: A represented tree, decomposed into preferred paths, (on the left) is depicted as a virtual tree (on the right). The nodes within splay trees are connected using undirected, solid edges. The root of each splay tree points to the path-parent using a directed, dashed edge. The nodes of each splay tree are encircled. Nodes that are on a preferred path of their own are on a splay tree that also only contains themselves. These include nodes E, K, O, R, G, M and T.

hold, disregarding the order of operations. They thereby optimize for the worst case runtime of a single operation. However, the per operation runtime is often not the relevant factor in applications where a sequence of operations must be performed. In these cases only the runtime of the entire sequence is important and single operations can have a considerably worse runtime than the average if they lower the cost of subsequent operations. On this basis, the splay tree does not guarantee a worst-case per operation runtime of *O(log(n))* that many balanced data structures can provide, but it can guarantee an amortized runtime of *O(log(n))* over a worst-case series of operations. Additionally, the nodes in the splay tree do not have to store additional information that may be required to keep the tree balanced. Each node only has to store references to its left child, right child, and parent, besides its key.

2.2.1 Tree rotations

The splay operation makes an element the root of its splay tree by repeatedly performing left and right tree rotations, which change the structure of a tree while preserving its order. A right rotation on, around, or rooted at node v is depicted in Figure 2.4. v is labeled the *root* and it is the parent of node w, which is referred to as the *pivot*. Before the operation, w is the left child of v, α and β are the left and right subtrees of w, γ is the right subtree of v, and v has a parent node p. Note that all subtrees and the parent node are optional and that the operation can be performed if they are empty or do not exist. The only requirement is the existence of a left child for the splayed node v. After the operation, which can be likened to a clockwise rotation around the root, v is now the right child of w and has increased its depth by one. w, which is now the parent of v, has decreased its depth by one and v's former parent, if v had a parent, is now w's parent. α is still the left subtree of w, although the depth of every node that is contains has also decreased. Vice versa, γ is still the right subtree of v and its nodes have increased their depth. The nodes contained in β retained their depth but the subtree now has v as its parent. A left rotation is the exact inverse operation. It is also depicted in Figure 2.4, below the right rotation.

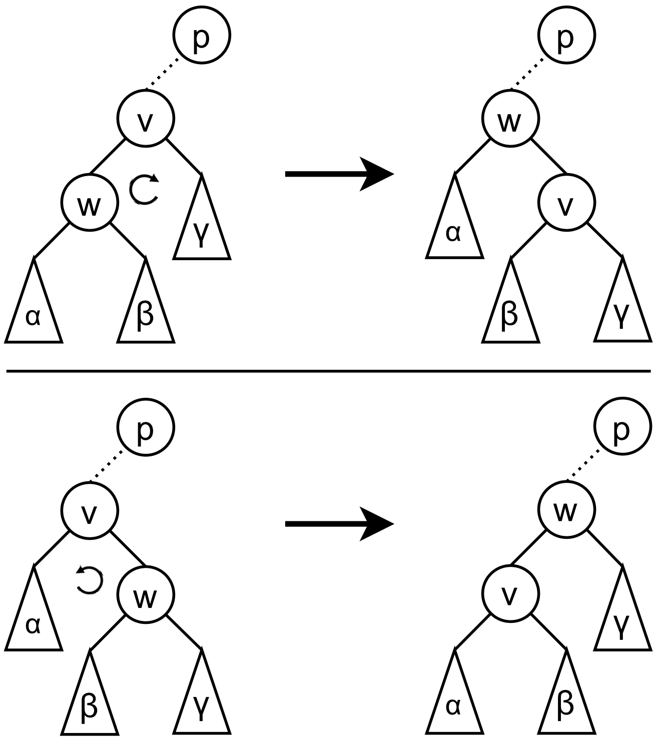

Figure 2.4: Right and left tree rotations

```
RotateRight($v$)
1  parent(left($v$)) = parent($v$)
2  if parent($v$) $\neq$ NIL
3      if left(parent($v$)) == $v$
4          left(parent($v$)) = left($v$)
5      if right(parent($v$)) == $v$
6          right(parent($v$)) = left($v$)
7  parent($v$) = left($v$)
8  if right(left($v$)) $\neq$ NIL
9      parent(right(left($v$))) = $v$
10     left($v$) = right(left($v$))
11 else left($v$) = NIL
12 right(parent($v$)) = $v$
```

In the code, *parent(v)* returns a reference to the parent of a node v or NIL if v does not have a parent. *left(v)* and *right(v)* return the left and right children if they exist. Right and left rotations can be performed by changing a constant number of pointers. They therefore run in *O(1)* and are used as a basis in the analysis of the splay operation.

2.2.2 Splay operation

The two types of tree rotations are then aggregated into three steps, each containing one to two tree rotations, which are referred to as *zig*, *zig-zig*, and *zig-zag*. Each of these steps has an inverse variant which results in a total of six suboperations. The overall splay operation then consists of a series of these three or rather six steps. Which of these steps is performed depends on the relative positioning between the splayed node, its parent and its grandparent or the existence thereof.

A zig step may be performed at the end of a splay operation, when the parent of the splayed node is the root of the tree and the splayed node has no grandparent. In this case a single rotation around the parent of the splayed node is sufficient to make it the root of the tree. If the splayed node is the left child of its parent, a right rotation is performed, otherwise a left rotation is performed. Depictions of the two variants are included in Figure 2.4. v is the root and w is the splayed node. What must be noted is that in a normal rotation v could have a parent. This is not possible during a *zig* step because v must already be the root of the splay tree.

The zig-zig step is performed if both the splayed node and its parent are left children or if both are right children of their respective parents. In the former case two right rotations, in the latter two left rotations are performed. In both cases the first rotation is performed on the grandparent of the splayed node and the second on its parent. They are depicted in Figure 2.5.

The zig-zag step is either carried out if the splayed node is the left child of its parent and its parent is the right child of its grandparent or if it is the right child of its parent and its parent is the left child of its grandparent. In the former case, first the tree rooted at the parent is rotated right after which the tree rooted at the grandparent is rotated left. In the latter a left rotation around the parent and then a right rotation around the grandparent is performed. Both variants are depicted in Figure 2.6.

To move the splayed node to the root, it would be sufficient to exclusively perform rotations on the splayed node's parent, which is equal to the *zig* step. The benefit

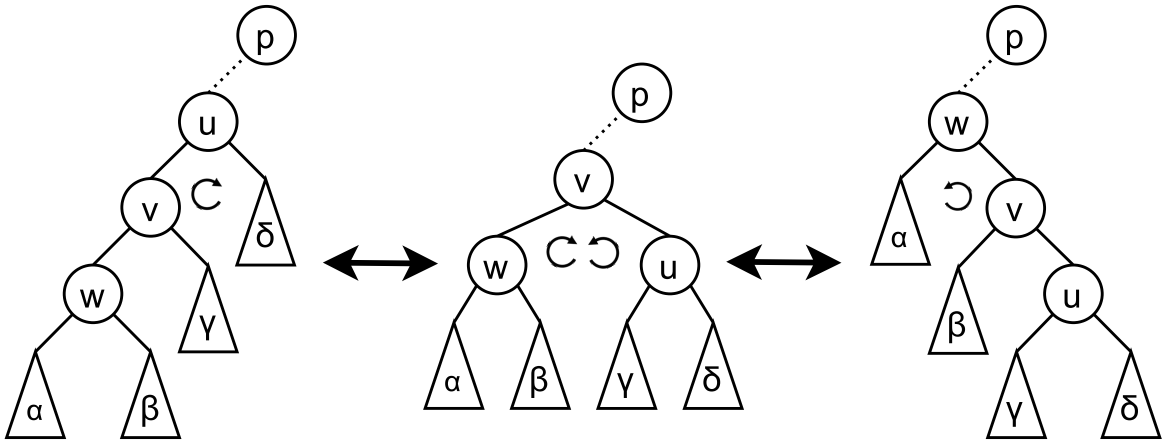

Figure 2.5: zigzig step

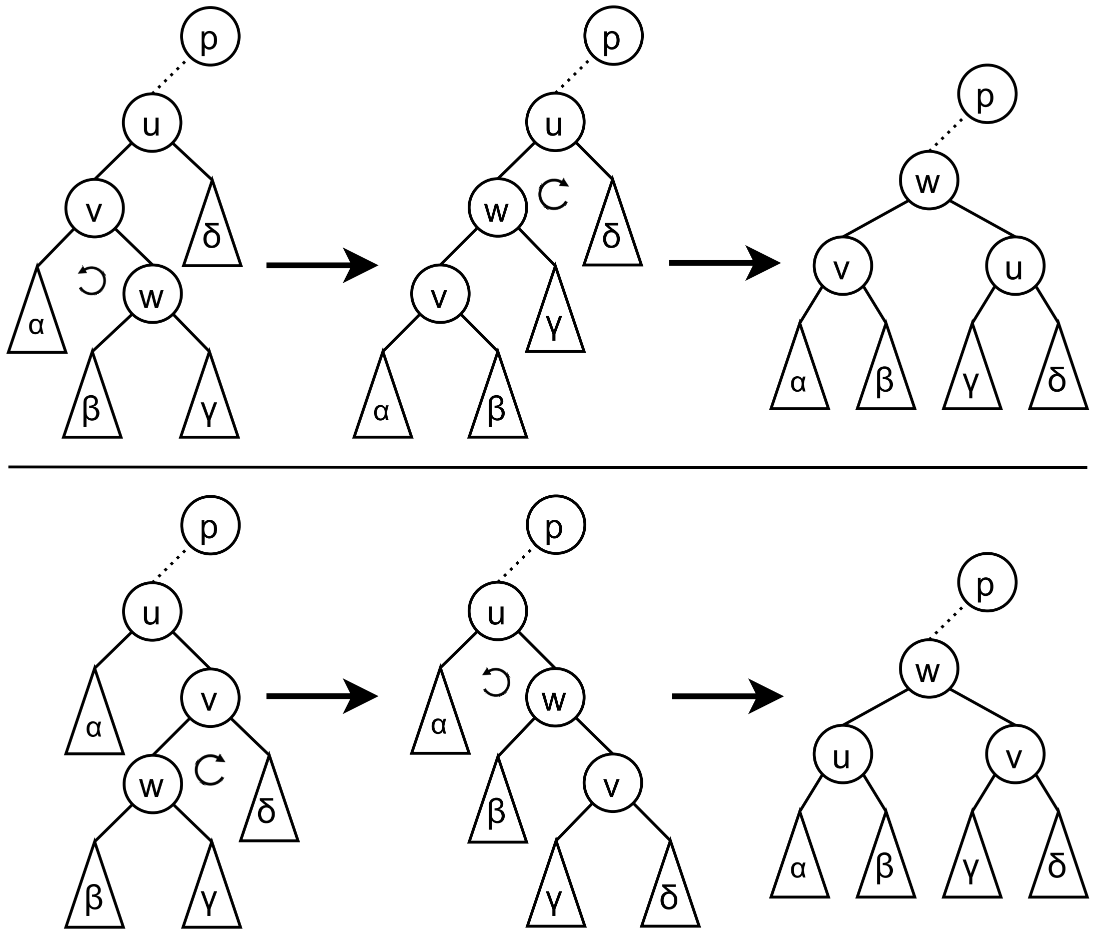

Figure 2.6: zigzag step

of performing rotations in pairs and sometimes rotating around the splayed node's grandparent is that it reduces the depth of every node on the path from the splayed node to the root by about half which automatically makes the tree more balanced and helps to improve the running time of a splay operation.

2.3 Operations

The following section will explain the expose operation, which is central to all other operations supported by the link/cut tree. It will make heavy use of the splay operation, which was described in the previous section.

2.3.1 Expose

The effect of the expose operation on the virtual tree can be compared to the splay operation in splay trees. By exposing a node, it is moved to the root of the virtual tree as opposed to the root of the splay tree. The difference is that the underlying tree data structure is not a splay tree, which is conceptually just a binary search tree, but a tree of splay trees. If a splayed node v was located in the same splay tree as the current root of the virtual tree, a simple splay would suffice to move it to the root. If it is not, then it must first be moved into the same splay tree as the root before it is made the root of the virtual tree. The steps that are applied to achieve this condition can be described differently based on whether they are viewed from the perspective of the represented or the virtual tree. First, the effects on the represented tree will be explained. Afterwards, the applied steps will be translated into what they mean for the virtual tree. [8, 4]

From the perspective of the represented tree, exposing a node v puts it on the same preferred path as the root of the represented tree. To achieve this, the path from v to the root is followed. If on this path a node u, which may be v itself, is encountered, that is connected to its parent via a non-preferred edge, this node is converted to a preferred child, making the edge to its parent a preferred edge. If the parent of u already had a preferred child, the edge connecting it to its parent must be converted to a non-preferred edge, making v's sibling a regular child. The operation of creating a new preferred edge and possibly breaking an already existing one is called a splice. This step is repeated until n is the root of

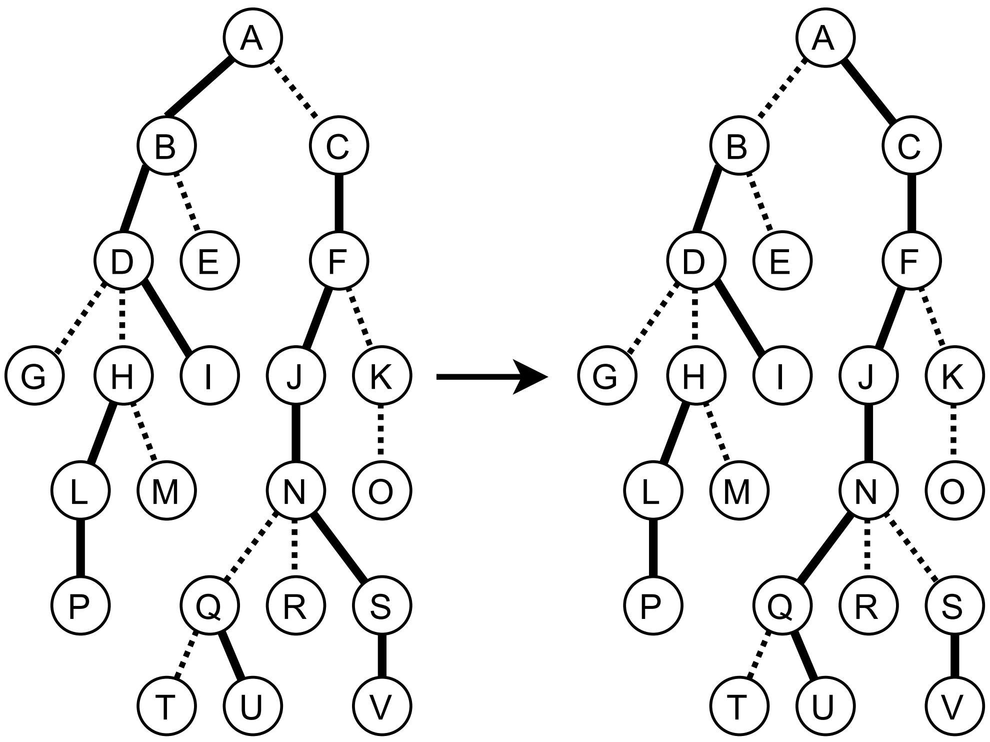

Figure 2.7: Effect of an expose operation on node Q. The initial represented tree is shown on the left, the represented tree after the operation is shown on the right. On the path from Q to the root there are two non-preferred edges, so two splices are necessary. The first converts the non-preferred edge connecting Q to N to a preferred edge and converts the preferred edge from S to N to a normal, non-preferred edge. The second converts the non-preferred edge from C to A to a preferred edge and the preferred edge from B to A to a non-preferred edge. Q is now on the same preferred path as the root and has kept its preferred child U.

the represented tree after which *v* will be on the same preferred path as the root. During these steps, the structure of the represented tree remains unaffected. The only thing that changes is the partitioning of the represented tree into preferred paths, which the user is oblivious to. An expose operation from the perspective of the represented tree is depicted in Figure 2.7

```
EXPOSE($v$)
1  SPLAY($v$)
2  while parent($v$)
3        SPLAY(parent($v$))
4        right(parent($v$)) = $v$
5        SPLAY($v$)
```

The following paragraph describes the expose operation from the perspective of the virtual tree. The steps that are taken are practically equivalent to the implementation in code. This is because the virtual tree closely resembles how the link/cut tree is stored in memory. There is a difference, however. In the code there is no notion of solid or dashed edges, but only pointers. In Section 2.1.2 it was shown that a solid edge between two nodes indicates that both nodes point at each other, whereas a dashed edge indicates that only one node points at the other and the other does not know it is pointed at. So when a dashed edge is converted to a solid edge, this indicates that the node with the incoming dashed edge now also points at the node with the outgoing dashed edge. Likewise, the conversion of a solid to a dashed edge implies that one node lost its pointer to the other, while the other kept theirs.

In line 1 v is splayed, making it the root of its splay tree. Splaying a node has no effect on the represented tree as dashed edges remain unaffected. All splaying does is reorganize the nodes within a splay tree without affecting their order and hence without affecting the corresponding preferred path. Because v is now the root of its splay tree, it can have an outgoing dashed edge to a node on a different splay tree, which it received when it was rotated into the position of the root. If no such edge exists, then v's parent pointer is *NIL* and v is already the root of the virtual tree, so no further actions are required. If v is not the root, then its path-parent is splayed as well, making it the root of its splay tree. This, again, has no effect on the represented tree.

Now the right subtree of the parent, which contains all nodes on its preferred path that are its real descendants, must be swapped with v. This is the equivalent to the splice operation described above. By making v the right child of its parent, a solid edge between v and the parent is formed. This implies that v and the parent are now on the same preferred path, which in turn indicates that the

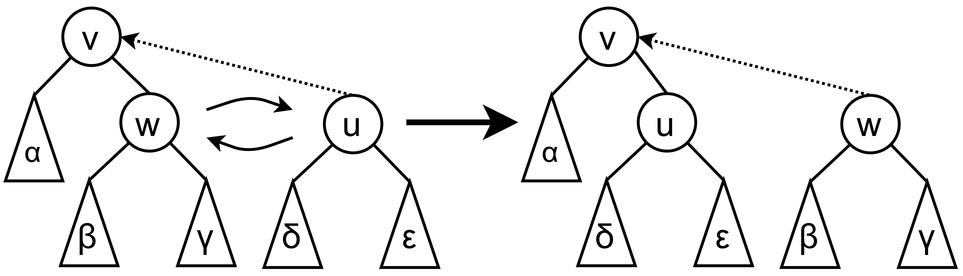

Figure 2.8: Effect of a splice operation on the virtual tree. The subtree rooted at w is replaced with the subtree rooted at u. The edge connecting u and v becomes solid, whereas the edge between w and v becomes dashed. This has the effect that node v loses its reference to its child w, but w still points at v.

edge connecting the topmost node in *v*'s preferred path to the path-parent was converted from a non-preferred edge to a preferred edge. At the same time, the solid edge from the former right child of the parent to the parent is transformed into a dashed edge, which also implies the conversion of a preferred edge to a non-preferred one in the represented tree. The splice operation, viewed from the perspective of the virtual tree, is depicted in Figure 2.8.

After *v* has been made the right child of its parent, a single rotation on the root is performed. This makes the former parent the left child of *v* and also gives *v* the next path-parent pointer to a node on a different splay tree. The previously described operation is then repeated until *v* does not have an outgoing dashed edge anymore. This is the case if *v* is made the root of the virtual tree which might already be the case after the first splay operation and before any splices. *v* being the root of the virtual tree means that all of the nodes in its left subtree are its real ancestors. This subtree must therefore also include the root of the represented tree, which explains why exposing a node puts it on the same preferred path as the root. If no left subtree exists, then *v* itself must be the root. Figure 2.9 shows the effect of exposing the node Q on the virtual tree. It uses the same example that was used in Section 2.2.

What must be noted, is that the right subtree of *v* never changes after the first splay operation has been performed. This means that if *v* has real descendants that are on the same preferred path as *v* before the expose operation, they will still be on the same preferred path afterwards. It is also possible to disconnect *v*'s right subtree after the first splay, as it would be during a splice. This has no

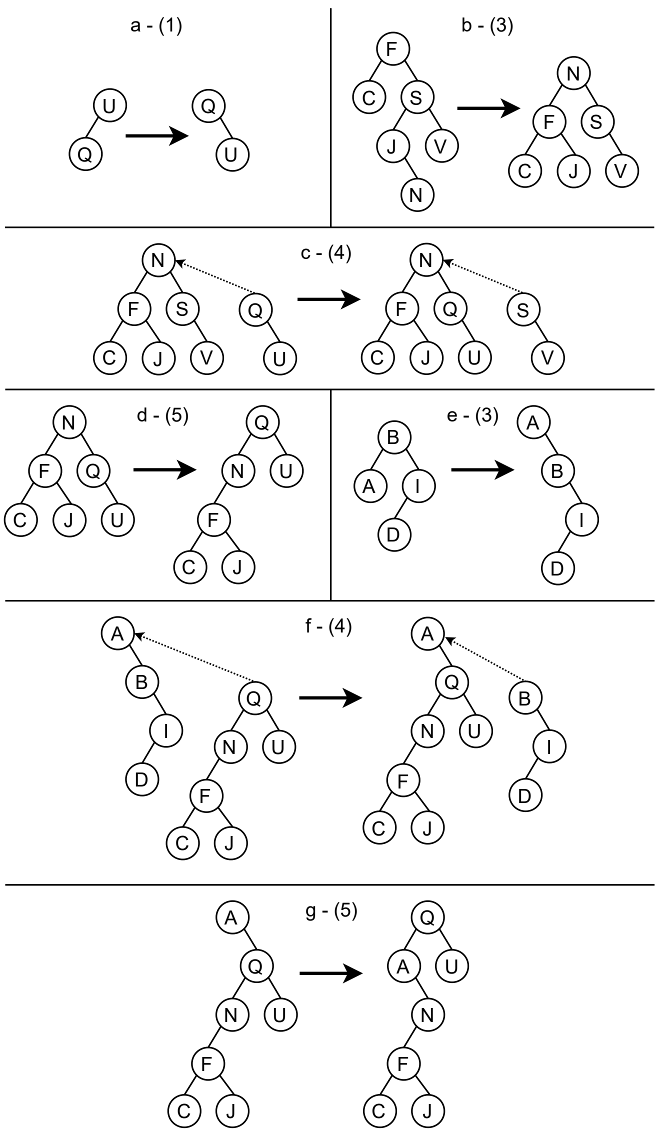

Figure 2.9: The effect of exposing node Q. The letter in each box specifies the step in the expose operation. The number in brackets references the corresponding line in the code. All dashed edges that are not relevant to the operation are not displayed. The first step is to splay node Q. Because the parent of Q, U, is already the root of the splay tree, a single right rotation on U is performed. This gives Q the reference to the path-parent, which is N. In a second step N is splayed. Then the right subtree of N, which is the subtree rooted at S, is replaced with the subtree rooted at Q. In the virtual tree, a dashed edge is converted to a solid edge and a solid edge to a dashed edge. S still keeps its reference to its parent, but its parent loses its reference to S. This is why the edge between S and N becomes dashed and therefore directed. After this, Q is splayed again, which makes it the root of its splay tree and gives it access to the next path-parent. Then the same three steps are repeated until Q is the root of the virtual tree. In step 5 the new path-parent of Q, which is A, is splayed. The splay tree rooted at Q is made its right child (6) and Q is splayed again (7). This completes the expose operation because Q does not have access to an additional path-parent. In the code this means that Q's parent pointer is *NIL*, so the test in line 2 fails and the routine terminates.

impact on the running time of the expose operation, but does have an impact on the implementation of some of the other methods supported by the link/cut tree. Some become easier to implement, while others can become more complex. This will be shown when the other operations are explained.

The expose operation runs in *O(log(n))* amortized time. This means that in a worst case series of operations the average cost of a single *expose* is in *O(log(n))*. Some operations may have a higher cost, but they will usually lead to a lower cost of the other operations. [8]

2.3.2 Link

Two trees in the forest can be linked by creating a preferred or non-preferred edge from the root of one of the trees to an arbitrary node in the other tree. For this operation, two nodes, v and u, are required. v is the root of one of the represented trees. u is any node in a separate tree.

```
LINK($v, u$)
1  EXPOSE($v$)
2  EXPOSE($u$)
3  parent($v$) = $u$
```

The first step in linking the trees of v and u is to expose both v and u. Now there are two variants of making v a real child of u. The first is to make u the parent of v in the virtual tree without making v the left or right child of u. This creates a dashed edge from v to u, so they will still be on separate splay trees. v becomes a real child of u because v has no left subtree in the virtual tree after being exposed. If it had a left subtree then the nodes contained in it would be higher in the corresponding preferred path. If this were the case, not v but a node in said subtree would be the new real child of u. However, if v is the root of its represented tree, then it can not have a left subtree. If it had a left subtree then this subtree would contain the root. A link operation using this variant is displayed in Figure 2.10.

```
LINK($v, u$)
1  EXPOSE($v$)
2  EXPOSE($u$)
3  left($v$) = $u$
4  parent($u$) = $v$
5  right($u$) = NIL
```

The second variant combines the splay trees of v and u by making u the left child of v. v will have no left subtree after being exposed as was described above, so no subtree must be replaced. Making u the left child has the effect that every node in the solid subtree rooted at u will be located higher than v in its preferred path. This, however, does not automatically make v the real child of u. If u has a right subtree, the nodes contained in it would be located lower than u but higher than v in the preferred path. Hence, the parent of v in the preferred path would not be u but a node in said subtree. This is why the right subtree of u must be disconnected, meaning that the parent, u, loses its reference to its right child, while the child, which is the root of the removed subtree, keeps the reference to its parent. This converts a solid edge to a dashed edge in the virtual tree and a preferred edge to non-preferred edge in the represented tree.

As described in Section 2.3.1, depending on the implementation of the expose operation, the right subtree of u may already be disconnected when u is exposed. If it is not, then it must be removed manually during the link operation. Because u is now a solid child of v and v is on the same splay tree, v must also be made the parent of u in order to restore the property that children in splay trees have references to their parents. A depiction of this variant is also included in Figure 2.10.

The running time of both variants is equivalent to the running time of *expose* because the expose operation is performed a constant number of times and only a constant number of pointers is changed.

2.3.3 Cut

Cutting a node v causes the real subtree rooted at v to be removed from its parent in the represented tree. This creates an additional represented tree which is added to the forest.

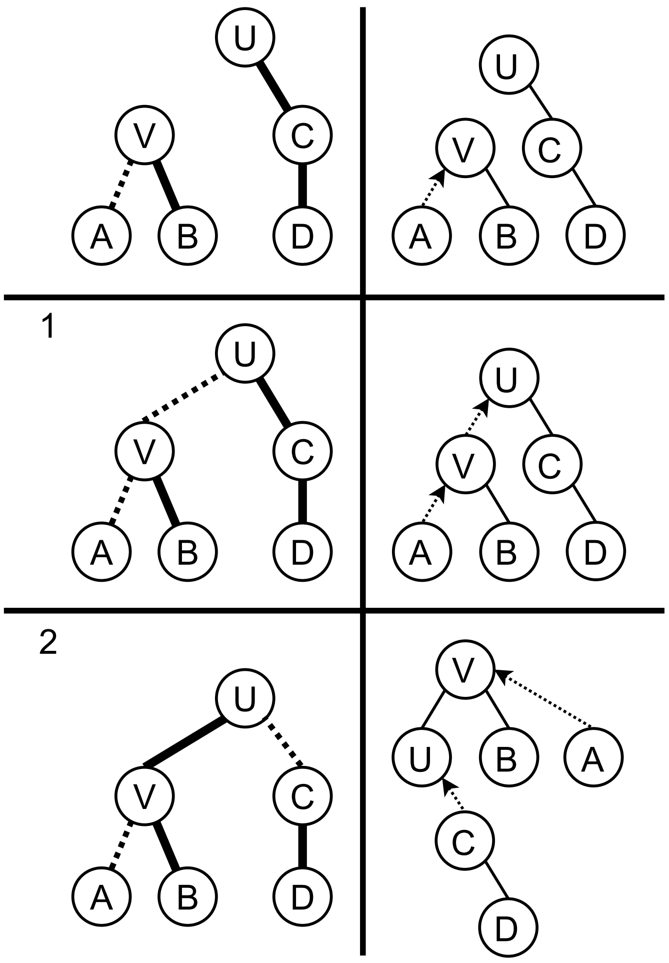

Figure 2.10: Effect of the link operation. The left upper box shows the initial represented trees, the right upper box shows the corresponding initial virtual trees. Because v and u are already the roots of their virtual trees, the expose operations are redundant. The effect of linking v to u using the first link variant is depicted in the two center boxes. In the represented tree, v becomes a non-preferred child of u. In the combined virtual tree (center right), v is attached to u with a dashed edge. In the second variant, which is depicted in the two bottom boxes, u is made the left child of v and the edge connecting u to its right child is converted to a dashed edge, which makes v a real child of u and puts it on the same preferred path.

```
Cut(v)
1 expose(v)
2 if left(v) ≠ nil
3       parent(left(v)) = nil
4       left(v) = nil
```

First, v is exposed. Now all nodes that are higher than v in its preferred path are located in its left subtree. By removing the solid edge connecting v to its left child, v becomes the root of a new represented tree. The difference between removing a solid edge and disconnecting a subtree, as it is performed during *expose*, is that the root of the removed subtree loses the reference to its parent. Thus, after the cut has been performed, no node on either of the two trees still has a reference to any node on the other tree. If v does not have a left child after being exposed, then it is the root of its represented tree and cut has no effect. A cut operation is depicted in Figure 2.11. The running time of the cut operation is equivalent to the running time of *expose* because *expose* is performed once and only a constant number of pointers are changed.

2.3.4 findRoot

The findRoot operation returns the root of the represented tree that a node belongs to. It can be used to infer whether two nodes are part of the same tree by using *findRoot* on both of them and comparing the result.

```
findRoot(v)
1 expose(v)
2 while left(v) ≠ nil
3       v = left(v)
4 splay(v)
5 return v
```

As with most other operations supported by the link/cut tree, finding the root of the represented tree that a node v belongs to first requires exposing v. After the expose operation, v will be on the same preferred path as the root of the represented tree. Because the left subtree of the root in a virtual tree contains

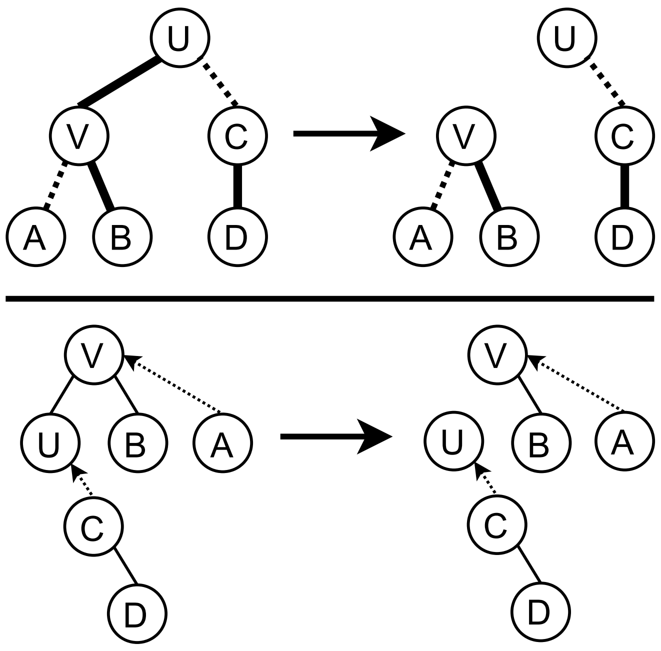

Figure 2.11: Effect of cutting node V on the represented tree (top) and the virtual tree (bottom). In the represented tree the edge connecting V to its parent U is removed. In the virtual tree, V has already been exposed. The solid edge connecting V to its left child U is removed.

all its real ancestors, it must also include the real root. In fact, it must be the leftmost node in v's splay tree. If it was part of the right subtree of another node in the splay tree, then this node would be higher than the real root in the represented tree. Therefore, the leftmost node in the splay tree must be found. The code shows how this can be performed. Finally, after the root has been found, it must be splayed or accessed. Both has the same effect because the real root is already on the same splay tree as the root of the virtual tree, which is v. Figure 2.12 shows how the root of the represented tree can be found.

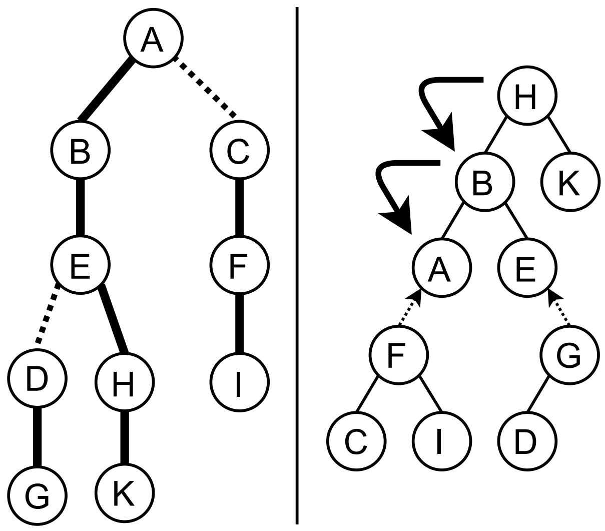

Figure 2.12: Finding the root of node H. H is already the root of the virtual tree, so exposing H is redundant. The leftmost node in H's splay tree is determined, which must be the root of the represented tree. Afterwards the real root is splayed, which is not shown in the figure.

2.3.5 LowestCommonAncestor

The code for finding the lowest common ancestor of two nodes v and u assumes that the two nodes are not equal and that v and u are part of the same tree. *LowestCommonAncestor* will make use of a subroutine called *isRoot*, which determines if a node is the root of a splay tree.

```
IsRoot($v$)
1  if PARENT($v$) == NIL
2      return TRUE
3  else
4      if left(parent($v$)) $\neq v$ and right(parent($v$)) $\neq v$
5          return TRUE
6      else return FALSE
```

The first line checks if v is the root of the virtual tree by testing if the parent pointer is *NIL*. If it is, then *true* is returned. Afterwards, it checks if v is either the left or the right child of its parent. If it is neither the left or right child then it must be the root of a splay tree and *true* is returned. Otherwise false is returned.

```
LowestCommonAncestor($v, u$)
 1  expose($v$)
 2  expose($u$)
 3  if isRoot($v$)
 4      return parent($v$)
 5  elseif isRoot(parent($v$)) and parent(parent($v$)) $\neq$ NIL
 6      return parent(parent($v$))
 7  else
 8      if left(parent($v$)) == $v$
 9          return $v$
10      else return $u$
```

To find the lowest common ancestor of v and u, first v is exposed, which makes it the root of the virtual tree. After v has been exposed, but before u has been exposed, there are four possible scenarios. In the first two scenarios u is on a different splay tree than v. In the latter two it is on the same. For cases 1 and 2 the node that is the last path-parent on the path from u to the root of the virtual tree is of special importance. It will be referred to as w. Now u is exposed aswell. Because w was a path-parent on the path from u to the real root, it was splayed in order to replace its right subtree with the splay tree rooted at u. It was either in the left or in the right subtree of v before exposing u.

If it was in the left subtree, then v was rotated into the right subtree of the new root w when w was splayed. Hence, after w was splayed, v was either the right child of w, if a *zig* or *zig-zag* step was performed last, or the right child of the right child, if a *zig-zig* step was performed last. In any case, the subtree rooted at v (*zig*, *zig-zag*) or the parent of v (*zig-zig*) must have been replaced with the subtree rooted at u. This means that either v or the parent of v became the root of a splay tree but not the root of the virtual tree. In both cases w must be the lowest common ancestor of v and u. This can be explained by looking at the represented tree instead of the virtual tree. w is the first node that is both on the path from v to the root as well as on the path from u to the root, which makes in the lowest common ancestor. If v was made the right child of w, after the splay of w, then it becomes the root of a splay tree. This is tested in line 3. If it is the root of a splay tree, then its parent, w, is returned. If a *zig-zig* was performed, v is now the right child of a node that is the root of a splay tree but not the root of the virtual tree. In line 5 this is tested by first performing *isRoot* on v's parent and then checking if the parent of v's parent is *NIL*. If it was *NIL*, then v would still be on the same splay tree as the root of the virtual tree and not on a separate one. If line 5 evaluates to *true*, the parent of v's parent is returned, which is also w.

If w is in the right subtree, then v was rotated into the left subtree of the new root w, after w was splayed. In this case, it is not replaced with the splay tree rooted at u, but remains the left child of either w or the left child of w. It will still be a left child even after w's right subtree has been replaced with the subtree rooted at u and u was finally splayed. It will also still be on the same splay tree as the new root u. Lines 3 and 5 will fail because they test if v is on a splay tree other than the one containing the root. Then line 8 evaluates to *true* and v is returned. v is the lowest common ancestor because it is in the left subtree of u after u has been exposed. It is therefore on the path from u to the real root of the tree and must be the lowest common ancestor.

In the third case, u is in the left subtree of v after v has been exposed but before u has been exposed. Therefore, no splices are performed and exposing u shifts v into the right subtree of the new root of the virtual tree, u. If a zig or a zig-zag step are performed last, then v is the right child of the root, if a zig-zig step is performed last, then v is the right child of the right child of the root. In any case v will not be the root of a splay tree, which is tested in line 3, and line 5 will also not evaluate to *true* because either u is the parent and the root of the virtual tree (*zig*, *zig-zag*) or the parent of v is not the root of a splay tree (*zig-zig*). Line 8 also fails, because v was rotated into the right subtree and not the left and can therefore only be a right child, so u is returned. u must be the lowest common ancestor because it is on the same preferred path as v and it must also be located higher than v in said preferred path because it was in v's left subtree before being exposed. Case 4 is the exact opposite. u is in the right subtree of v before being exposed. After it is exposed, v is either the left child of u or the left child of the left child of u. Again, the tests in lines 3 and 5 fail, but line 8 succeeds, so v is returned. This is correct because u and v are on the same splay tree and therefore on the same preferred path. Furthermore, u is in the right subtree of v before being exposed and therefore lower in the preferred path.

The running time is, again, identical to that of *expose* because *expose* is performed twice and the remaining parts of the code can be performed in constant time.

2.3.6 isDescendant

```
isDescendant($v, u$)
1  expose($v$)
2  expose($u$)
3  if isRoot($v$) or isRoot(parent($v$)) and parent(parent($v$)) $\neq$ NIL
4          return FALSE
5  else
6          if left(parent($v$)) == $v$
7                  return FALSE
8          else return TRUE
```

The code for checking if a node v is a descendant of a node u follows the same principles as *lowestCommonAncestor*. Again, v and u are exposed. If v ends up on a splay tree that does not contain u after u has been exposed, then there exists a lowest common ancestor that is neither v nor u. Why this lowest common ancestor must exist is described in the prior section. This means that v can not be a descendant of node u. If v is on the same splay tree as u after u has been exposed, then either v is a descendant of u or u is a descendant of v. If v has been moved into the left subtree of u, then it is a real ancestor of u because it is higher in the preferred path. This is tested in line 6. If it is on the same splay tree but has been rotated into the right subtree of u, then this can only mean that u is on the path from v to the root of the represented tree. u was therefore already in the left subtree of v before it was exposed. Hence, v is a descendant of u. This, again, follows the same ideas presented in Section 2.3.5.

2.3.7 findParent

findParent returns a pointer to the real parent of a node or *NIL* if it does not exist. As opposed to the other operations presented so far, *findParent* does not make use of the expose subroutine and is not special to the splay or link/cut tree. With minor adaptations this method can be used to find the parent in any binary search tree.

```
findParent($v$)
 1  if left($v$)
 2      $v$ = left($v$)
 3      while right($v$) $\neq$ NIL
 4          $v$ = right($v$)
 5      return $v$
 6  else
 7      if parent($v$)
 8          while TRUE
 9              if isRoot($v$)
10                  return parent($v$)
11              else
12                  if right(parent($v$)) == $v$
13                      return parent($v$)
14                  else $v$ = parent($v$)
15      else return NIL
```

If a node v has a left child, then this child contains the nodes that are higher in the preferred path. It therefore must also contain the parent. If the left subtree exists, the rightmost node in that subtree must be the parent of v. This follows the same idea as *findRoot* introduced in Section 2.3.4. If the left subtree does not exists, then the parent must be a node along the path from v to the root of the virtual tree. If no parent exists, then v is the root of the virtual tree and *NIL* is returned. Otherwise the path to the root is followed until a node w is encountered that is the right child of its parent. The parent of w must be v's real parent because v is the leftmost node in the parent's right subtree, which is equal to its real child. If the root of the splay tree is encountered before a node w is found, then v is the topmost node in its preferred path and the path-parent must be the parent of v.

Finding the preferred child of a node v is very similar to finding its parent. If a right subtree exists, then the real child is the leftmost node in that subtree. Otherwise it is the parent of the first node on the path from v to the root of v's splay tree that is the left child of its parent. If no such node exists along this path, then there is no preferred child and v is the lowest node in its preferred path.

2.4 Augmentations

The nodes in the virtual tree can be augmented to maintain aggregate information about their subtrees. This information might include the minimal or maximal value of a node in the subtree, the number of nodes in the subtree, the number of dashed children or other metrics. [2] This requires every node in the virtual tree to store one or multiple additional values. Here each node will store three extra values. The first is the number of nodes in the subtree rooted at a node v, which will be referred to as the *subtree size.* This includes v itself. The second is the number of nodes in the subtrees rooted at the dashed children of v and the third is the number of nodes in the subtrees rooted at dashed children of v's solid descendants. The former will be called the *size of dashed children* and the latter the *size of dashed descendants.* The size of dashed children is included in the size of dashed descendants, which is itself included in the subtree size of a node. While these values by themselves seem arbitrary and irrelevant, they will later be used to deduce useful information about a node in the represented tree. Figure 2.13 shows an example of a virtual tree whose nodes maintain this information.

Of course this information must be updated whenever the structure of the virtual tree changes. Operations that may affect on the structure of the virtual tree include tree rotations, *link*, *cut*, and *expose.* It will now be shown how the invalidations caused by these operations can be fixed.

There will be three types of operations to repair the invalidations caused by alterations of the virtual tree. The first will be used to update subtree information which depends on a nodes solid children. If a nodes subtree information of this kind becomes invalid due to a change in the virtual tree, but the subtree information of its solid children remains valid, then the information of the children can be used to update the information of the parent. In this example, the values that this applies to are the subtree size and the size of dashed descendants. It does not apply to the size of dashed children because this value only depends on the dashed children of a node, but is not associated with the solid children. The operation that fixes this type of information will be referred to as *update_aggregate.*

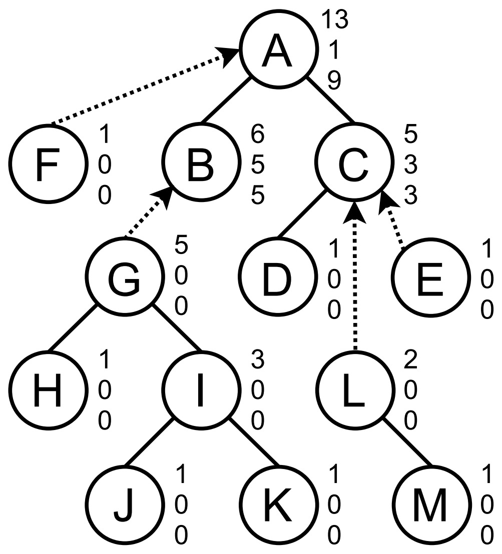

Figure 2.13: Subtree information maintained by the nodes. The first of the tree numbers besides each node is the size of the subtree that each node is rooted at. The middle number is the size of all dashed children that a node has and the bottom number is the number of nodes that are in subtrees rooted at the dashed children of a nodes solid descendants. Node M has no descendants, so the subtree size is 1 and the other two values are 0. Node B has one dashed child, G, which has a subtree size of 5 and no solid children, so the size of dashed children and dashed descendants is 5. B's own subtree size is 6 because a value of 1 needs to be added for B itself. Node A has one dashed child with a subtree size of 1 and its solid descendants have three dashed children(G, L, E) with a total size of 9 nodes. Also it has 3 solid descendants which leads to a subtree size of 13.

```
UPDATE_AGGREGATE($v$)
1 sizeSubtree($v$) = sizeDashedChildren($v$) + 1
2 sizeDashedDescendants($v$) = sizeDashedChildren($v$)
3 if left($v$)
4       sizeSubtree($v$) += sizeSubtree(left($v$))
5       sizeDashedDescendants($v$) += sizeDashedDescendants(left($v$))
6 if right($v$)
7       sizeSubtree($v$) += sizeSubtree(right($v$))
8       sizeDashedDescendants($v$) += sizeDashedDescendants(right($v$))
```

It is assumed that the subtree information in the left and right child of a node that *update_aggregate* is called on is correct and that the size of dashed children for every node in the tree is valid at the point when it is called. If this is true, then the size of the subtree can be updated by simply adding the size of the left subtree, the size of the right subtree, the size of dashed children and a constant value one for the node itself. The size of dashed descendants can be calculated similarly by adding the size of dashed descendants of the right and left child and then adding the size of dashed children. Afterwards the subtree information of a node will be correct again. *update_aggregate* will now be used to fix the subtree information after a tree rotation.

2.4.1 Rotations

If a tree rotation occurs, then the subtree size and the size of dashed descendants of the nodes involved in the rotation may become invalid. The size of dashed children remains unaffected because each node keeps its dashed children. After the tree rotation has been performed, only the information in the root and the pivot will be invalid. The two children of the pivot and the other child of the root remain unaffected. This was demonstrated in Figure 2.4. Therefore, *update_aggregate* needs to be called on the root and the pivot, which was rotated into the position of the root. However, the order of these two calls is not arbitrary. Because the root is the child of the pivot after the rotation, the pivot's subtree information can not be fixed until the subtree information of the root has been fixed. *update_aggregate* will need to be called first on the root and only then on the pivot.

2.4.2 Expose

Next, the invalidations caused by *expose* will be repaired. During a splice the path-parent's size of dashed children and size of dashed descendants will be affected because it loses a solid child and gains a dashed child. This was shown in Figure 2.9. A call to *update_aggregate_expose* on the path-parent after each splice will fix both values. The subtree size remains unaffected.

```
UPDATE_AGGREGATE_EXPOSE($v, w, u$)
1  sizeDashedChildren($v$) $-=$ sizeSubtree($w$)
2  sizeDashedDescendants($v$) $-=$ sizeSubtree($w$)
3  if $u \neq$ NIL
4      sizeDashedChildren($v$) $+=$ sizeSubtree($u$)
5      sizeDashedDescendants($v$) $+=$ sizeSubtree($u$)
```

v is the path-parent of the splice operation. w is the now solid and former dashed child of v and u is the now dashed and former solid child of v. u can be *NIL* because it is possible that the path-parent did not have a solid child before the splice. w's subtree size must be subtracted from v's size of dashed children because it is no longer a dashed child. The same applies to v's size of dashed descendants. Likewise, u's subtree size must be added to v's size of dashed children and size of dashed descendants because it is no longer a solid child.

2.4.3 Link

When fixing the invalidations caused by *link*, only the first of the two link variants introduced in Section 2.3.2 will be looked at. In this variant, a node v, which is the root of its represented tree, is made the dashed child of a node u on a separate tree. This has an impact on all three metrics of u.

```
UPDATE_AGGREGATE_LINK($v, u$)
1  sizeDashedChildren($u$) $+=$ sizeSubtree($v$)
2  UPDATE_AGGREGATE($u$)
```

u's size of dashed children increases by the subtree size of v. If u has solid children, their subtree information remains unchanged. Calling *update_aggregate* therefore corrects the subtree size and the size of dashed descendants of u.

2.4.4 Cut

Cutting a node v has no impact on its dashed children because they will still point at v. Because the node that is cut loses a solid child, as was explained in Section 2.3.3, the subtree size and the size of dashed descendants will have to be adjusted. This can be accomplished with a single call to *update_aggregate* after v was cut. The information in the solid child that was not removed is still correct and the other child is now *NIL*, so the preconditions for *update_aggregate* are met.

2.4.5 Aggregate methods

Information on a node in the virtual tree is usually not as important as collecting information about the same node in the represented tree. The information about a nodes subtree in the virtual tree can, however, be used to make statements about the node in the represented tree. For example, if a node v is the root of the virtual tree, then the number of real descendants of this node is equivalent to the number of nodes in its right subtree plus the number of nodes in all subtrees that are rooted at dashed children of it. Both of these components are being maintained as described above. The number of nodes in the real subtree rooted at v can therefore be calculated as follows.

```
getRealSize($v$)
1  expose($v$)
2  return sizeSubtree(right($v$)) + sizeDashedChildren($v$) + 1
```

v must be exposed because otherwise there might be nodes that are not in the right subtree of v, but which are real descendants nonetheless. The constant value one is added for node v itself.

It is also possible to determine the depth of a node. If a node v is the root of the virtual tree, the number of nodes on the path from v to the real root is the subtree size of the left child minus the size of dashed descendants of the left child. This number is always greater or equal to zero because the size of dashed descendants is included in the subtree size. The resulting figure is the number of solid descendants in v's left subtree, which are all the nodes on the path from v to the real root.

```
getDepth($v$)
1  expose($v$)
2  return sizeSubtree(left($v$)) − sizeDashedDescendants(left($v$))
```

To compute the depth of a node it would also be possible to directly maintain information about the number of a node's solid descendants instead of the size of dashed descendants. However, it is not possible to compute the depth with information on the subtree size and the size of dashed children alone as it would not be possible to discern between nodes that are real descendants and nodes that are not. Both of these operation's cost is equivalent to that of *expose*.

3 Implementation

The following chapter will describe the implementation of the link/cut tree in C++. The library does not have any third party dependencies and heavily makes use of templates, which is why it is header-only. This simplifies the build process at the cost of longer compilation time. This drawback is, however, kept to a minimum because the combined library is only a few hundred lines in length.

CMake was chosen as the build tool as it allows for an easy cross-platform integration of the library. The build process using CMake involves two steps. First standard build scripts for different build systems can be generated based on configuration which is read from CMakeLists.txt files. It supports the creation of build scripts for build tools such as make, ninja, or the visual studio build tool. The build scripts are then used to compile the project for any specific platform. CMake is therefore not a regular build system but rather a platform-independent generator for build systems.

The library consists of three classes. The first class, LctNode, resembles a node in the virtual tree. Its structure and operations are very similar to the code examples which were already presented in Section 2.1.2. The next class, OpTreeNode, inherits from LctNode and puts additional constraints on the structure of the tree, in that any node can have at most a left and a right child or an only child. The final class, LinkCutTree, resembles a container for the two types of nodes. These three types will now be presented in detail.

3.1 LctNode

The LctNode class defines the following members.

```
template<typename T> class LctNode {
public:
    LctNode(const T& aContent, int aID = idCounter++);

    const T& getContent() const;
    T& getContent();
    int getID();

    virtual bool link(LctNode* aOther);
    virtual void cut();

    virtual LctNode* findRoot();
    virtual LctNode* lowestCommonAncestor(LctNode* aOther);
    bool isDescendant(LctNode* aOther);

    int getVirtualSize();
    int getRealSize();
    int getDepth();

    virtual LctNode* findChild();
    virtual LctNode* findParent();
    template<typename F> void path(F aFunction);
    template<typename F> LctNode* find_if(F aFunction);

    friend class LctUtils;
    static int idCounter;

protected:
    LctNode* _left, * _right, * _parent;
    T _content;
    int _id;

    void rotR();
    void rotL();
    void splay();
    void expose();
    bool isRoot();
    int _sizeDashedChildren;
    int _sizeSubtree;
    int _sizeDashedDescendants;

    virtual void update_aggregate();
    virtual void update_aggregate_expose(LctNode* aNewChild, LctNode
        * aFormerChild);
    virtual void update_aggregate_link(LctNode* aNewChild);
};
```

The instance variables *_left*, *_right*, and *_parent* are pointers to other nodes. The value that each node holds is stored in an instance variable called *_content*, which has the type of the template parameter T. The class can therefore be templated to hold any data type. It can be any class, struct, or union and there are no requirements that the type must meet. Additionally, every class has an int *_id* which helps to identify nodes.

The class defines a constructor that takes a reference to type T and and optional int. It initializes all pointers with the value *nullptr*. The reference to type T is used to initialize the member *_content*. If an int is passed, it is assigned to the member *_id*. Otherwise it is initialized with the value of the static field *idCounter* which is then incremented. Each instance also contains three integers which are used to maintain subtree information. The idea behind these variables was introduced in section 2.4. *_sizeDashedChildren* and *_sizeDashedDescendants* are assigned 0 because a node does not have children when initialized. Likewise, *_sizeSubtree* is initialized with a value of 1. All members are protected and only *_content* and *_id* can be retrieved via getters.

getContent can be used to retrieve the content of a node. There are two versions of it. One can be called from a constant context and returns a constant reference, while the other is not constant and returns a regular reference. *getID* returns the ID that was assigned to the node on initialization. There is no option to change the ID of a node after it has been created. Also there are no checks that two nodes do not share the same id. The assignment of ids to nodes is left to the user.

The methods *rotL* and *rotR* perform left and right tree rotations. All methods related to the splay tree, which are *rotL*, *rotR* and *splay*, as well as *expose*, *isRoot* and the methods related to augmentation are protected members so only the

LctNode class itself and any inheriting class will have access to them. These methods are not part of the public interface that the library presents to the user, but are only used internally. *expose* and *splay* could be made public members because they can not be used to change the structure of the represented tree and also do not reveal any internal information. However, they remain protected because there is no real reason for the user to expose or splay a node.

The three aggregate methods are defined as virtual and can be overridden by an inheriting class if other aggregate information needed to be maintained. The overriding methods should call the base methods or the subtree information in the base class will not be maintained. *link* and *cut* are also marked as virtual so their behavior can be adjusted in subclasses. The benefit of this will become clear in the next section.

Most of the methods of the LctNode class were already introduced when the operations of the link/cut tree were presented. However, the code examples used in said section did not include calls to the aggregate functions. To highlight the differences between the real implementations and the code examples, the methods that call aggregate functions will be included here aswell. They are *link*, *cut*, *expose* and the tree rotations. Also, the methods *path* and *findIf* will be explained. The implementations of the other methods are practically identical to the code examples which were already presented.

3.1.1 Link

```
template<typename T> bool LctNode<T>::link(LctNode<T>* aOther) {
    expose();
    if (_left) {
        return false;
    }
    aOther->expose();
    _parent = aOther;
    aOther->update_aggregate_link(this);
    return true;
}
```

As opposed to the code example given in Section 2.3.2, the implementation of *link* returns a bool to indicate if a link operation was successful. It fails if the node object that it is called on is not the root of a represented tree. This is tested in line 3. In the code example the assumption was made that this would not be the case. *update_aggregate* is called after the link has been completed to restore the aggregate information of the parent, which receives a new virtual child. The effects of this was explained in Section 2.4.

3.1.2 Cut

```
template<typename T> void LctNode<T>::cut() {
    expose();
    if (_left) {
        _left->_parent = nullptr;
        _left = nullptr;
        update_aggregate();
    }
}
```

The only addition to the method *cut* is a call to *update_aggregate* on the node that *cut* was performed on. This is necessary because the root of the virtual tree loses a child which affects the *_sizeSubtree* and *_sizeDashedDescendants* members. Other than that it is equivalent to the code in section 2.3.3.

3.1.3 Expose

```
template<typename T> void LctNode<T>::expose() {
    splay();
    while (_parent) {
        _parent->splay();
        _parent->update_aggregate_expose(this, _parent->_right);
        _parent->_right = this;
        splay();
    }
}
```

In the implementations of *link* and *cut*, the aggregate functions were called after the operations had been performed. In *expose*, *update_aggregate_expose* is called during a splice. This is because it requires a pointer to the former right child of the path-parent as a parameter, which would no longer be accessible after the splice. Instead of creating a new variable to cache this pointer, *update_aggregate_expose* is called before a splice is completed.

3.1.4 Rotations

```
template<typename T> void LctNode<T>::rotR() {
    _left->_parent = _parent;
    if (_parent) {
        if (_parent->_left == this) {
            _parent->_left = _left;
        }
        if (_parent->_right == this) {
            _parent->_right = _left;
        }
    }
    _parent = _left;
    if (_left->_right) {
        _left->_right->_parent = this;
        _left = _left->_right;
    }
    else {
        _left = nullptr;
    }
    _parent->_right = this;
    update_aggregate();
    _parent->update_aggregate();
}
```

Only the right rotation is displayed because the left rotation is the exact inverse operation. After the rotation has been performed, two calls to *update_aggregate* are necessary. The first restores the properties of node that *rotR* is called on. It is now the right child of its former left child. The reason why *update_aggregate* is called first on this node and not its parent, is that the information of the parent depends on the correctness of its children's information. Therefore, the subtree information of this node needs to be fixed before the information of its parent can be fixed.

3.1.5 path and findIf

path and *findIf* use *findRoot* to move along the path from a node v to the root of v's represented tree and perform an action on every node on that path. The action that they perform is passed to them via a functor. The difference between the two is that *path* performs the action on every node on the path, whereas *findIf* returns the first node for which the functor evaluates to true. *path* and *findIf* are implemented as follows.

```
template<typename T> template<typename F>
void LctNode<T>::path(F aFunction)
{
    LctNode* lTemp = this;
    do {
        aFunction(lTemp);
    } while ((lTemp = lTemp->findParent()) != nullptr);
}
```

```
template<typename T> template<typename F>
LctNode<T>* LctNode<T>::find_if(F aFunction)
{
    LctNode* lTemp = this;
    do {
        if (aFunction(lTemp)) {
            return lTemp;
        }
    } while ((lTemp = lTemp->findParent()) != nullptr);
    return nullptr;
}
```

The function object passed to *path* must implement a method void aFunction(LctNode* aNode), which is called on every node on the path to the root. The functor passed to *findif* must implement a method bool aFunction(Node* aNode), which is called on the nodes on the path to the root until it evaluates to true. The corresponding node is then returned.

3.2 OpTreeNode

```
template<typename T> class OpTreeNode : public LctNode<T> {
public:
    OpTreeNode(const T& aContent, int aID = idCounter++);

    OpTreeNode* findRoot();
    OpTreeNode* findParent();
    OpTreeNode* findChild();
    OpTreeNode* lowestCommonAncestor(LctNode<T>* aOther);

    enum flagType {
        IS_LEFT_CHILD,
        IS_RIGHT_CHILD,
        IS_ONLY_CHILD,
        HAS_LEFT_CHILD,
        HAS_RIGHT_CHILD,
        HAS_ONLY_CHILD
    };

    void cut();
    bool link(LctNode<T>* aOther);

    bool isLeftDescendant(OpTreeNode<T>* aOther);
    bool isRightDescendant(OpTreeNode<T>* aOther);

    bool linkLeft(OpTreeNode* aOther);
    bool linkRight(OpTreeNode* aOther);
    bool linkOnly(OpTreeNode* aOther);

    friend class LctUtils;
private:
    std::vector<bool> _flags;
};
```

The class OpTreeNode extends the LctNode class and puts restrictions on the structure of the tree. Every instance of the OpTreeNode class can have either a left and/or a right or an only child. The additional information that is required to enable this behavior is stored in a bool vector called *_flags*. Every node maintains information on whether it is and on whether is has a left/right/only child. This information is then queried and updated during links and cuts.

The constructor defined by the OpTreeNode class is equivalent to that of LctNode. It takes a reference to the template parameter T and an optional int and forwards both to the constructor of the base class. All methods in the base class that return a pointer to the base class are overridden to return pointers to the derived class instead.

The *link* method defined in LctNode is overridden to always return false. In its place, three new methods are defined. *linkLeft* checks if the node passed as a parameter already has a left child or an only child. If it does, then *false* is returned. Otherwise the inherited link method is called. If it succeeds, the parent's *HAS_LEFT_CHILD* and the child's *IS_LEFT_CHILD* flags are set and *true* is returned. If *link* fails because the node that *linkLeft* is called on is not the root of its represented tree, then *false* is returned and the flags remain unaffected. The *linkRight* and *linkOnly* methods work accordingly.

The method *cut* is also overridden and adjusts the flags of the node that *cut* is called on and its parent in the represented tree. If the node that was cut was the right child of its parent, the *IS_RIGHT_CHILD* flag of the child and *HAS_RIGHT_CHILD* flag of the parent are set to false. Similar adjustments are made if the node is the left or the only child of its parent. The parent is found by using the *findParent* method defined in the base class.

Two methods, *isLeftDescendant* and *isRightDescendant* can be used to determine if a node is in the left or in the right subtree of another node. They both take a pointer to an OpTreeNode as a parameter and forward it to *isDescendant* in the base class. If it returns true, they determine the preferred child of the node that *isDescendant* is called on using *findChild*, which is also defined in the base class. This child must exist because the node that was passed as the parameter would not be a descendant otherwise. They then check if the child's *IS_LEFT_CHILD* or *IS_RIGHT_CHILD* flag is set and return true or false accordingly.

3.3 LinkCutTree

```
template<typename V, template<typename> class T = LctNode>
class LinkCutTree {
public:
    LinkCutTree<V, T>();
    T<V>* createTree(const V& aContent, int id = T<V>::idCounter++);
    T<V>* operator[](int aID);

private:
    std::map<int, T<V>> _nodes;
};
```

The LinkCutTree class is a container for the two types of nodes. It can either store LctNode type nodes or OpTreeNode type nodes. The class takes two template parameters. The first parameter, V, determines the type of objects that the nodes should hold. As with the other two classes, there are no requirements that this type must meet. The second parameter, T, is a template template parameter that determines the type of node that should be used. Per default it is set to LctNode. Generally this type must posses a constructor that takes a reference to an object of type V and an int.

Internally, the nodes are stored in a map with the key type int and the mapped type T<V>. A new node can be inserted by calling the createTree method with a reference to an object of type V and an optional int as the ID for the new node. If no int is passed then the value of the *idCounter* of T<V> is passed and then incremented.

If a node with the given ID is already present in the map, then no new value is inserted and a pointer to the already existing node is returned. Otherwise a new node is created by forwarding the parameters to the constructor of T<V> and mapping it to its ID. A pointer to this newly created node is then returned. Because nodes are mapped by their ID, multiple nodes with the same content can be inserted. Only the ID has to be unique.

Nodes can be retrieved by passing an int as the ID to the subscript operator. If a node with the given ID exists, a pointer to that node is returned. Otherwise *nullptr* is returned.

4 Evaluation

4.1 Setup

This section will compare the implementation of the link/cut tree to a trivial implementation of a rooted tree, which will be referred to as a *trivial tree*. Each node in the trivial tree stores a parent pointer and an associated value. Finding the root of a node can therefore be achieved by following parent pointers until a node is reached whose parent pointer is *NIL*. This node must then be the root. The equivalent of the *link* operation is simply setting the parent pointer of the root of a trivial tree to be a node on a separate trivial tree. To cut a node its parent pointer must be set to *NIL*.

The link and cut operations of the trivial tree run in constant time. It would not make sense to compare them to the link and cut operations of the link/cut tree, which run in *O(log(n))* amortized time. This section will therefore limit itself to a comparison of the findRoot operation, which is also the more relevant operation in many applications. The cost of finding the root of a trivial tree that a node v belongs to is equivalent to the depth of v. In the worst case, this is equal to the number of nodes in the tree. The worst case cost for a single operation, however, is often not relevant because in most applications a series of findRoot operations needs to be performed and only the cost of this entire sequence of operations needs to be considered. In the trivial tree the running time of a series of findRoot operations on randomly selected nodes is equivalent to the average depth of the nodes in that tree. If the number of findRoot operations in a sequence is high enough, higher cost and lower cost operations will even out and the total cost of said sequence will be proportional to the average depth.

The implementations of the link/cut and the trivial tree will therefore be compared on different tree types by first generating trees of different sizes, performing sequences of findRoot operations on the trees and then comparing the time that it took for either of the two implementations to perform the sequences of operations. If a sufficiently high number of findRoot operations is performed per tree size, the ratio between the cumulative execution time required by the link/cut and the trivial tree will approach a constant limit. The absolute execution times that will be presented in this section are highly dependent on the system in use, whereas the ratio between the execution times should be approximately equal in any environment.

The following chapter will compare both implementations based on random non-binary trees, full ternary trees, full binary trees, unbalanced binary trees and degenerate binary trees. The trivial implementation is expected to perform comparatively better if the height difference between the leafs of the tree is low. The link/cut tree will perform better the more unbalanced the tree is because the average cost of *findRoot* on a randomly selected node will be less than the average cost of finding a root in the trivial tree.

The implementation of the link/cut tree does, however, have a considerably higher overhead compared to the trivial implementation. Each call to findRoot in a link/cut tree can and in most cases does change the structure of the virtual tree, which results in a lot of additional operations that the trivial implementation does not have to carry out. While this lowers the cost for subsequent operations, it can have a negative impact on the performance if the number of nodes in a tree is low. Only when the number of nodes in a tree reaches a certain limit and the tree is sufficiently unbalanced, are the benefits of the link/cut tree expected to offset its overhead.

In this chapter the term *performance* will refer to the ratio between the average execution time of the trivial implementation and the average execution time of the link/cut tree. An increase in performance will therefore mean that the ratio increased and that it took the link/cut tree comparatively less time to perform a sequence of operations compared to a different result. It does not imply that the absolute average execution time of the link/cut tree improved, but only that it improved in comparison to that of the trivial implementation. In most tests that will be presented in the next sections the absolute execution times for both the link/cut tree and the trivial implementation will rise with an increasing tree size. This does not automatically imply that the performance increased or decreased.

The tests were carried out on a machine with an Intel Core i5-7200u CPU running Windows 10 Enterprise LTSC 1809 build 17763.1935. The Microsoft Visual C++

(MSVC) compiler version 19.28.29337 was used to compile the code.

4.2 Tests

4.2.1 Random non-binary trees

Non-binary trees of different sizes will be created from randomly generated Prüfer sequences. The height difference between the leaf nodes of the trees can differ based on the randomly selected sequences. Therefore, multiple trees of each size will be generated and a random sequence of findRoot operations will be performed on each of them. Because the trees that are generated in this test are random, the method that is used is different from the other tests. Here one sequence of operations is performed for each uniquely generated tree but multiple trees for each size are tested. In the other tests there is one deterministic tree per size and the different sequences of operations will be performed on the same tree. The number of unique trees and sequences per size that will be generated and tested will depend on a constant time limit but is at least 50 in order to minimize the effect of statistical outliers. The time that it takes for each of the implementations to process the sequences of operations is summed and compared for each tree size. The resulting ratio will approach a limit if a sufficient number of sequences is performed. Of course this limit can only be approximated and the test will only show the general development in performance. Trees of ten different sizes ranging from 50 to 25600 nodes will be compared. The size of each tree will be double the size of the respective smaller tree in order to test the performance on a logarithmic scale. This will help to compare the performance for both smaller and larger sized trees.

The number of queries per tree is set to twice its size. Therefore, the expected number of times that each node will be queried is 2. Some nodes will be queried more than twice while others will not be queried at all. This mimics the behavior which can be found in many practical applications. It is uncommon that each node will be queried the exact same number of times. Usually a small number of nodes is queried disproportionately often, while others are rarely queried if at all. This distribution benefits the link/tree because nodes that are queried are moved to the root of the virtual tree. If a node is queried multiple times in close

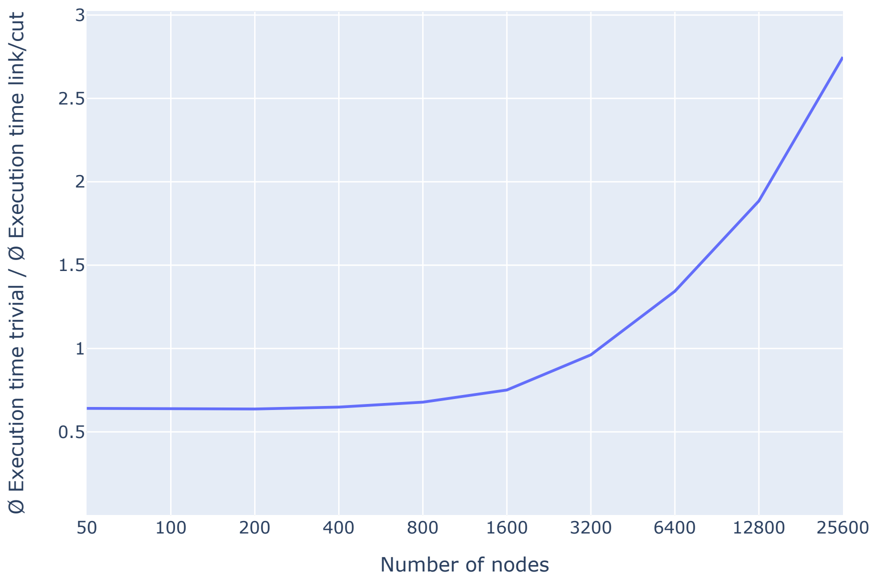

Figure 4.1: Performance for random non-binary trees for tree sizes ranging from 50 nodes to 25600 nodes

succession, it is likely that it will still be located close to the root. Subsequent findRoot operations that are performed on these nodes will therefore be less costly and improve the performance of the link/cut tree.

Figure 4.1 shows that the performance of the link/cut tree improves with the number of nodes in the tree. At up to 800 nodes it is about 32 percent slower than the trivial implementation. At 3200 nodes the ratio is approximately 1 and at 12800 it is already about 90 percent faster than the trivial implementation. Next, the same test will be carried out but on a linear scale. The smallest tree size will be 2000 nodes and it will increase by 2000 each step up to a size of 32000. This test will help to evaluate more closely how the ratio develops at larger tree sizes.

Figure 4.2 shows that that the improvement in performance is nearly linear. At 28000 nodes the link/cut tree is about three times as fast as the trivial implementation. However, at these sizes the data becomes less reliable because less trees could be tested for a given size within a feasible time frame.

Figure 4.3 shows the absolute average execution times per tree size. For the

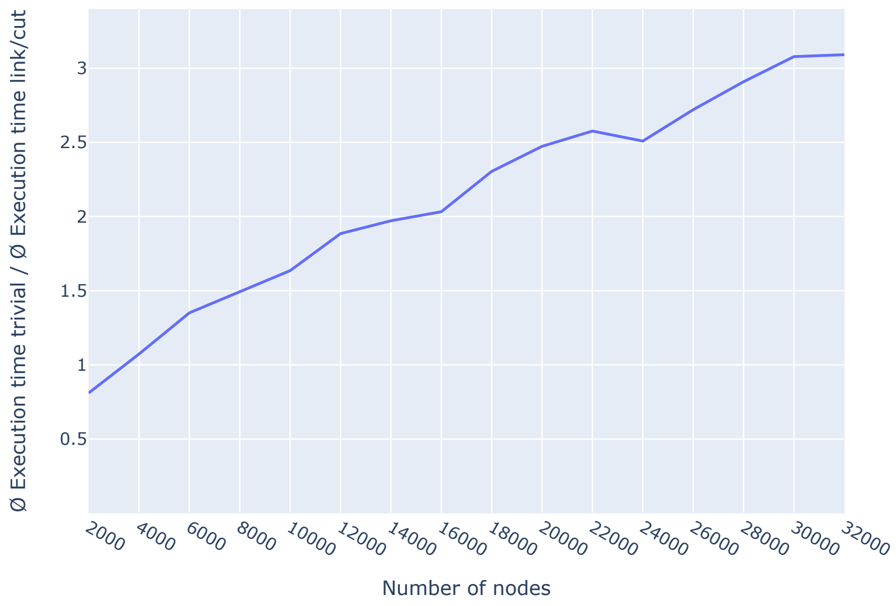

Figure 4.2: Performance for random non-binary trees for tree sizes ranging from 2000 nodes to 32000 nodes

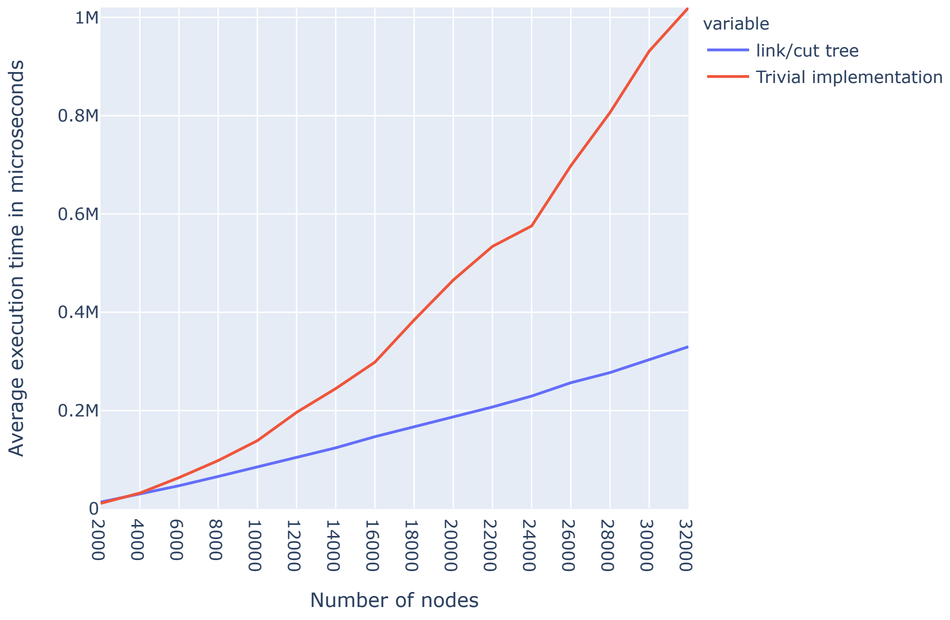

Figure 4.3: Average execution time for random non-binary trees for tree sizes ranging from 2000 nodes to 32000 nodes

link/cut tree, the time required to process one sequence of operations is nearly proportional to the number of nodes in the tree, whereas the trivial implementation shows an exponential increase.

4.2.2 Unbalanced binary trees

Each node in the unbalanced binary tree but the leafs will have a left and a right child and every left child will be a leaf. The tree is therefore heavily unbalanced, but not as unbalanced as degenerate trees, which will be tested in the next section. The link/cut tree is expected to perform worse on these types of trees than on degenerate trees but still better than the trivial implementation, which is expected to perform worse even at lower tree sizes because the average depth of nodes is high. The findRoot operation of the link/cut tree is therefore expected to be on average less costly than finding the root of a randomly selected node in the trivial tree.

Just as in Section 4.2.1, trees of 10 different sizes will be created and compared on a logarithmic scale in order to evaluate the performance both for smaller and larger sized trees. The smallest will have a height of 25 which corresponds to a size of 51 nodes and the largest will have a height of 12800 (25601 nodes). Afterwards, the same test will be performed on a linear scale where the smallest sized tree has 2001 (height 1000) nodes and the largest tree has 32001 (height 16000) nodes with increments of 2000. The sizes are chosen so that the test results here will be comparable to the results of the test on random non-binary trees, which will on average be more balanced than the unbalanced binary trees that are generated using the method described above.

Figure 4.4 includes both the test results from this section as well as those from Section 4.2.1. The plot shows that the rate at which the performance increases is much higher compared to that of random non-binary trees. At 1601 nodes the link/cut tree is already about 30 percent faster than the trivial implementation. In the random non-binary tree of the same size it is still 25 percent slower. The link/cut tree begins to outperform the trivial implementation at between a height of 400 to 800 nodes which corresponds to a size of 801 and 1601 nodes. In the random non-binary tree of the same size, this happens only at sizes higher than 4000 nodes.

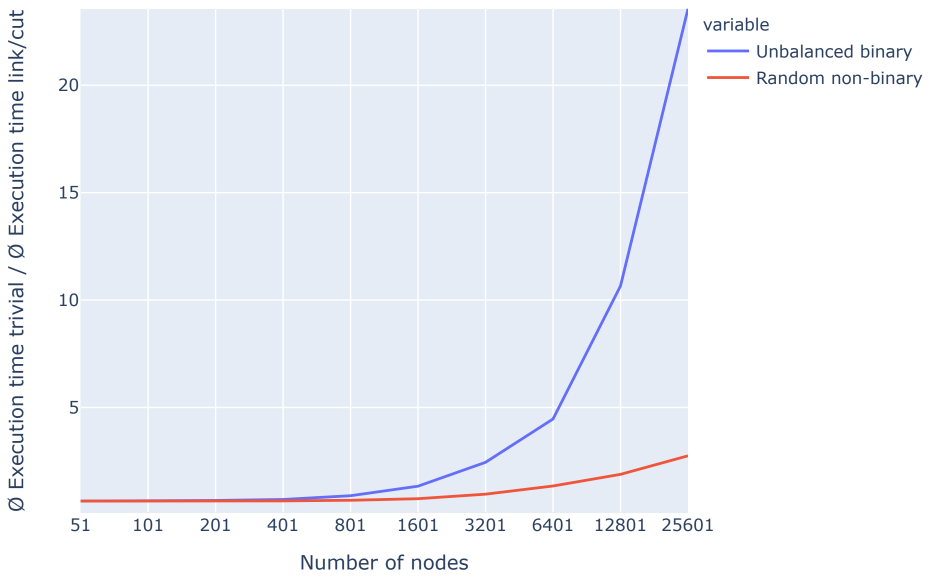

Figure 4.4: Performance for unbalanced and random non-binary trees for tree sizes ranging from 51 nodes to 25601 nodes

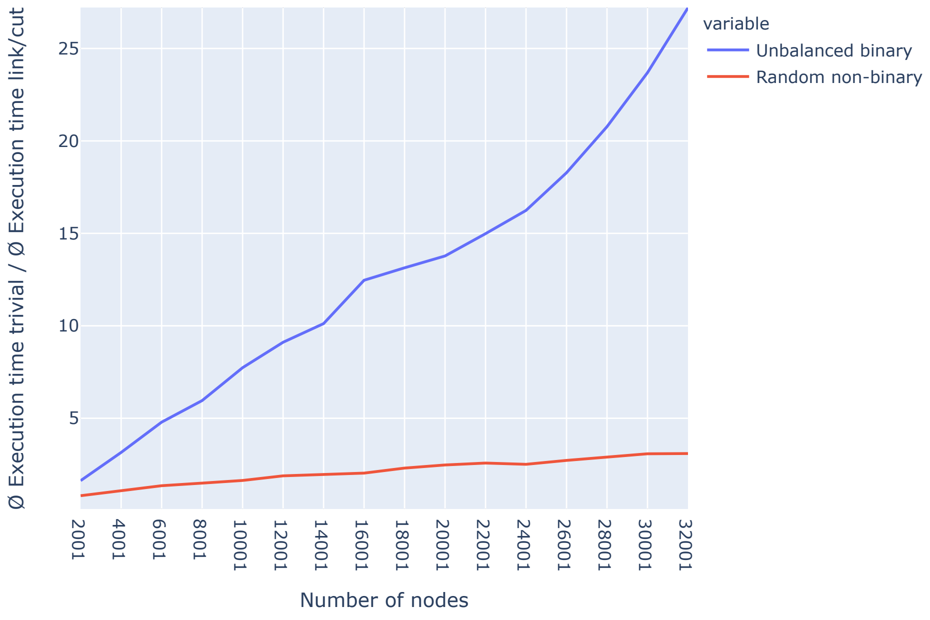

Figure 4.5: Performance for unbalanced and random non-binary trees for tree sizes ranging from 2001 nodes to 32001 nodes

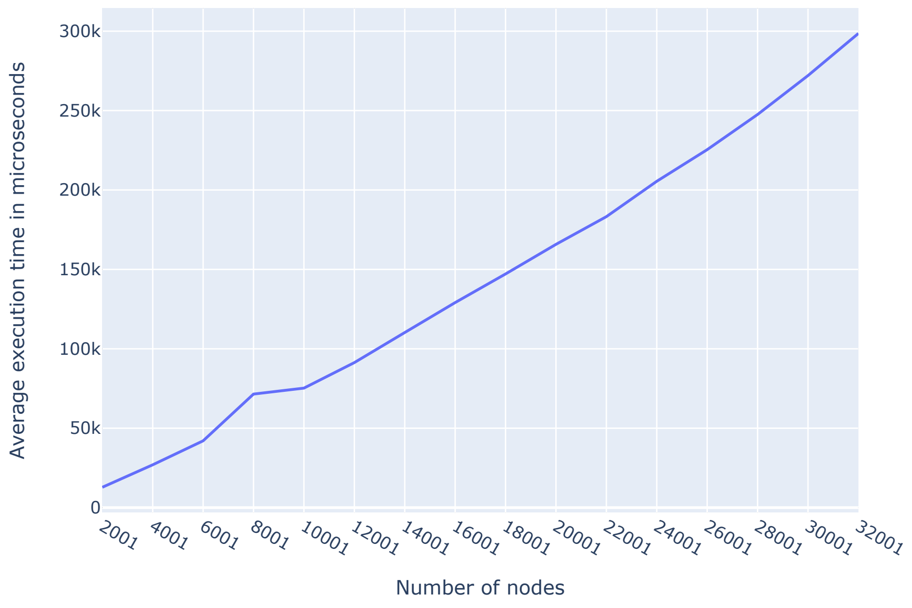

Figure 4.6: Average execution time of the link/cut tree for unbalanced trees for tree sizes ranging from 2001 nodes to 32001 nodes

The comparison on a linear scale, which is depicted in Figure 4.5, shows that the rate at which performance increases is slightly exponential. When random non-binary trees were tested, the rate at which the performance increases was approximately constant for all tree sizes.

Figure 4.6 shows that the absolute average execution time of the link/cut tree is nearly proportional to the number of nodes in the tree even if the tree is unbalanced. The average execution time of the trivial implementation, which is shown in Figure 4.7, grows exponentially with the size of the tree.

4.2.3 Degenerate binary trees

In a degenerate tree every parent node has exactly one child. Therefore, the tree resembles a list. The trivial implementation is expected to perform worst on these types of trees because they are maximally unbalanced and the average depth of nodes is highest compared to any other type of tree given a certain number of nodes. The performance of the link/cut tree on the other hand should not be

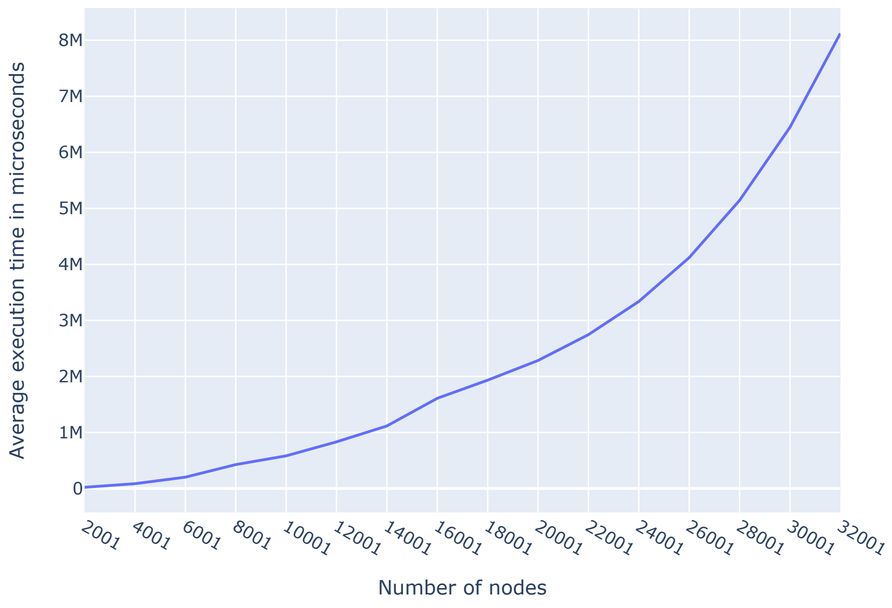

Figure 4.7: Average execution time of the trivial implementation for unbalanced trees for tree sizes ranging from 2001 nodes to 32001 nodes

impacted by the fact that the tree is unbalanced because the cost of its findRoot operation is not correlated to the depth of a node in the tree.

The tests that will be carried out on this tree type will be equivalent to those performed on random non-binary and unbalanced binary trees. First, the performance will be tested for tree sizes ranging from 50 to 25600 nodes. The results are depicted in Figure 4.8. This plot and all the following contain the results for the equivalent tests on random non-binary and unbalanced trees. Figure 4.9 shows the same results, but is capped at a size of 1600 in order to more clearly show how the performance develops at smaller tree sizes.

The link/cut tree starts to outperform the trivial implementation already at a tree size of around 400. For the unbalanced tree the same applies only after around 900 nodes. Figure 4.10 shows that the performance increases linearly up to a size of 16000 but then starts to accelerate. The same behavior could be observed on unbalanced binary trees. At 16000 nodes the link/cut tree is about 21 times faster for degenerate trees, 12 times faster for unbalanced trees, and only two times faster for random non-binary trees.

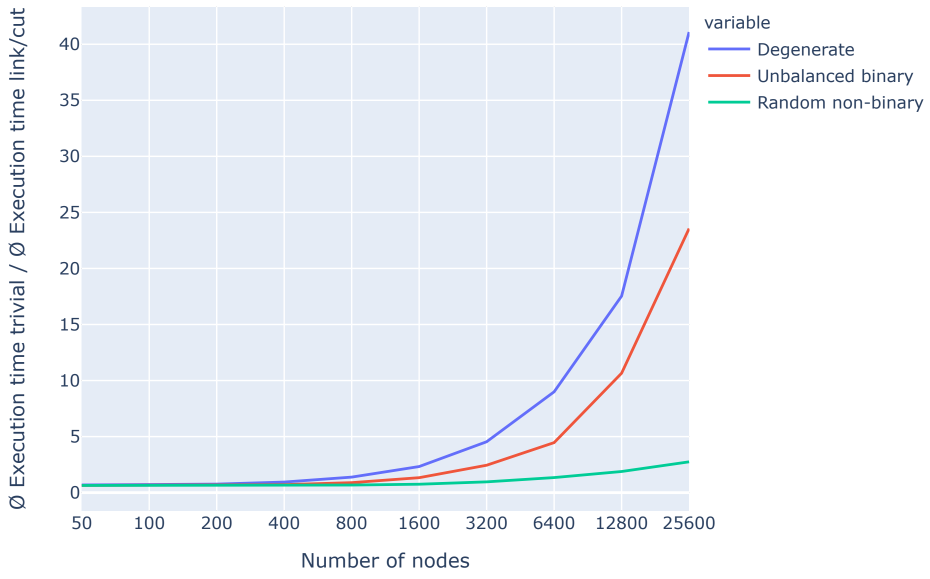

Figure 4.8: Performance for degenerate, unbalanced and random non-binary trees for tree sizes ranging from 50 nodes to 25600 nodes

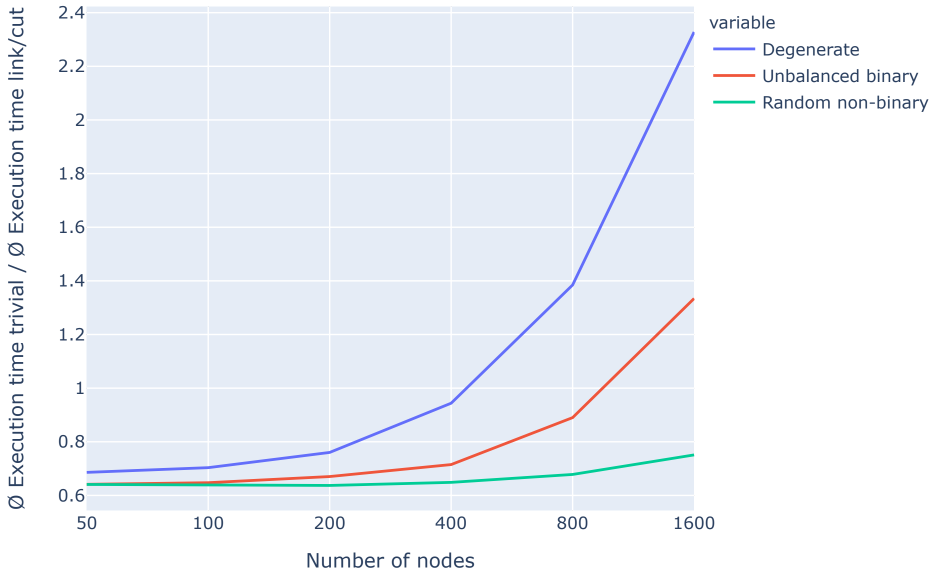

Figure 4.9: Performance for degenerate, unbalanced and random non-binary trees for tree sizes ranging from 50 nodes to 1600 nodes

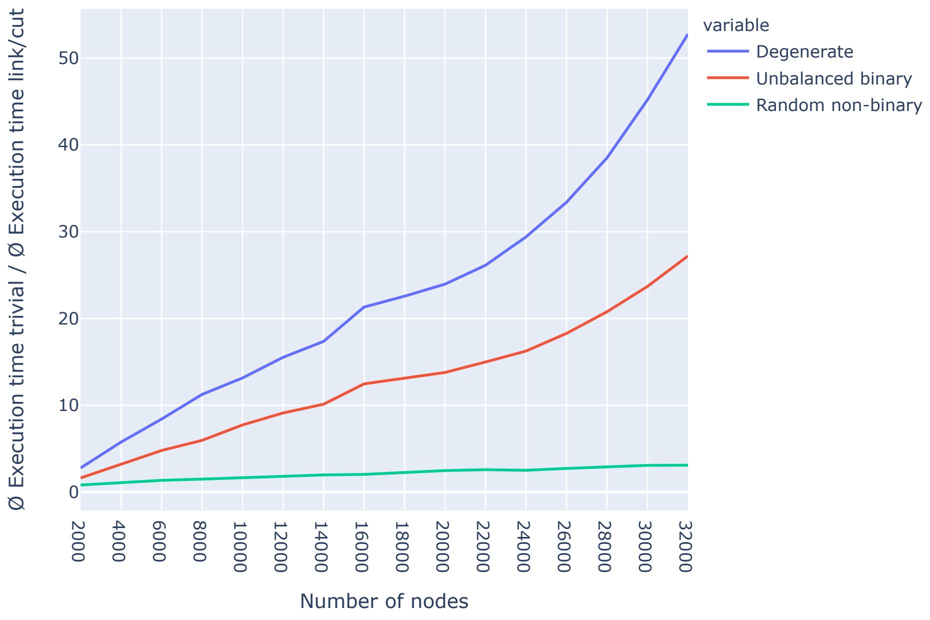

Figure 4.10: Performance for degenerate, unbalanced and random non-binary trees for tree sizes ranging from 2000 nodes to 32000 nodes

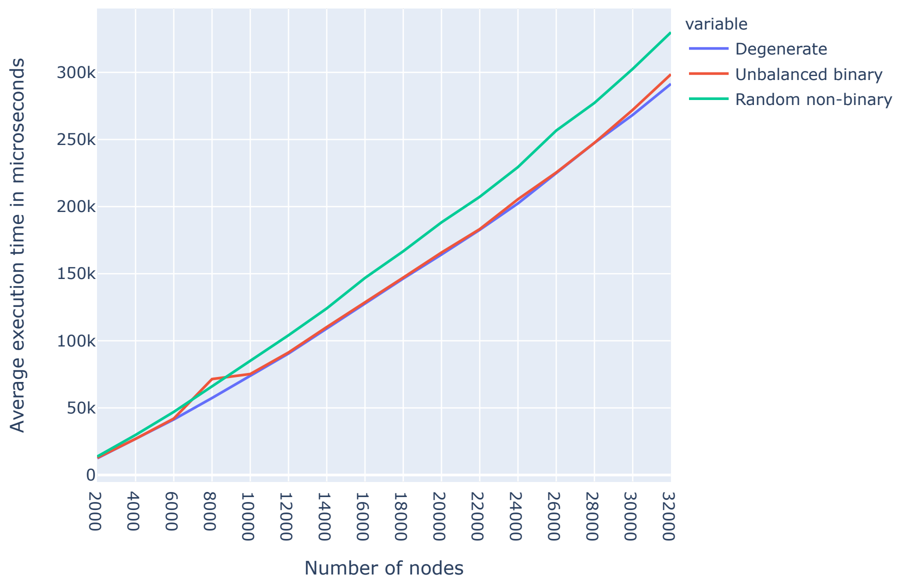

Figure 4.11: Average execution time of the link/cut tree for degenerate, unbalanced and random non-binary trees for tree sizes ranging from 2000 nodes to 32000 nodes

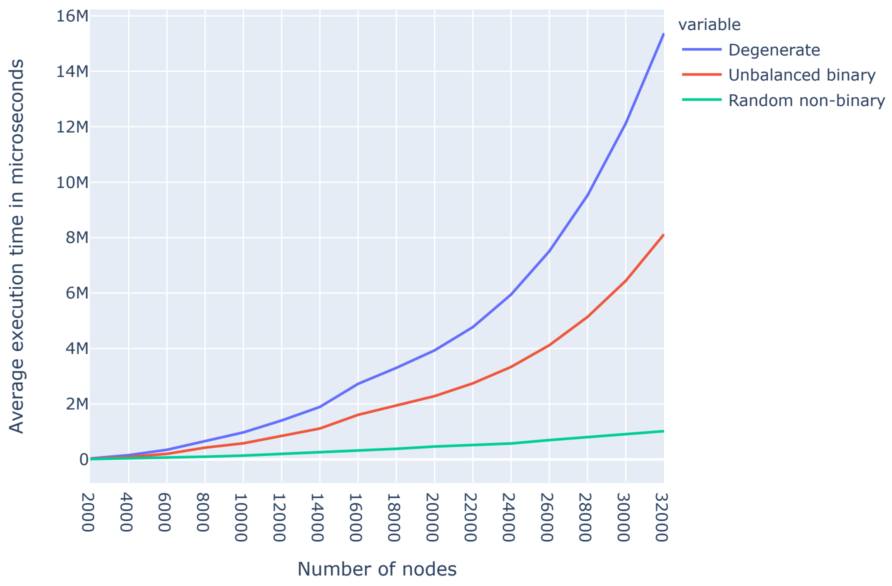

Figure 4.12: Average execution time of the trivial implementation for degenerate, unbalanced and random non-binary trees for tree sizes ranging from 2000 nodes to 3200 nodes

Figure 4.11 shows that the average execution time increases linearly with the tree size and is similar for all three tree types, although it performs a bit slower on random non-binary trees. The average execution time of the trivial implementation, which is shown in figure 4.12, increases exponentially with the tree size for all three types. It increases fastest for degenerate trees, followed by unbalanced trees and random non-binary trees.

4.2.4 Full ternary trees

8 different sized full ternary trees will be compared ranging from a height of 3 to a height of 10. A height of 3 indicates a size of 40 nodes and a tree of height 10 has 88573 nodes. The trivial implementation is expected to outperform the link/cut tree at any size because the average depth of the nodes is low and the tree is perfectly balanced. Every sequence of findRoot operations will be twice as long as the size of the tree and the nodes in the sequence will be chosen randomly for the same reasons discussed in Section 4.2.1. The number of findRoot sequences that is performed for each tree size will again depend on a constant time limit,

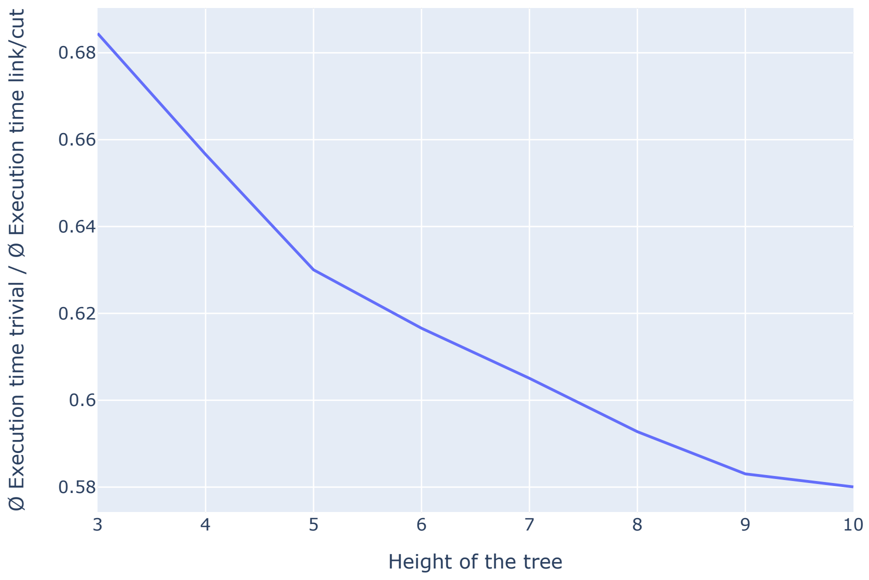

Figure 4.13: Performance for ternary trees for tree heights from 3 to 10

but is at least 50 for each size. The execution times of the sequences of operations that are performed on the different trees will be summed for both implementations and the two sums will be divided to calculate the performance.

Figure 4.13 shows that the performance of the link/cut tree decreases with the height of the ternary tree. At height 3 (40 nodes) it is about 32 percent slower and at height 10 42 percent slower than the trivial implementation. However, the rate at which the performance decreases is not linear but is slowing down with an increasing height. The reasons for this decrease are most likely the same as those explained in the Section 4.1. The impact of the overhead diminishes while nodes that are queried multiple times tend to stay near the root where subsequent findRoot operations on them are less costly.

The absolute average execution time of both implementations rises exponentially, which is depicted in Figure 4.14.

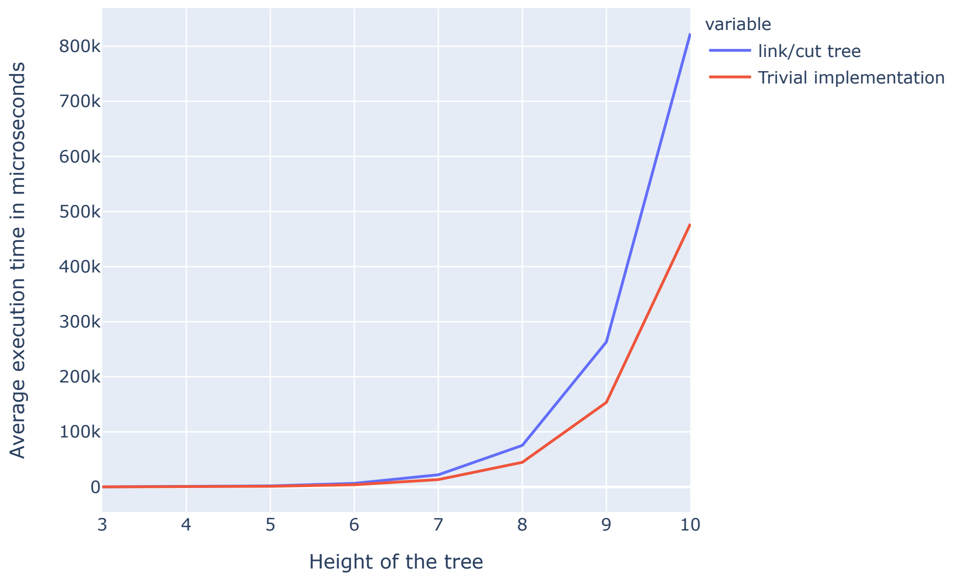

Figure 4.14: Average execution time for ternary trees for tree heights from 3 to 10

4.2.5 Full binary trees

Full binary trees with heights ranging from 5 to 15 will be tested. A binary tree of height 5 has 63 nodes and a binary tree of height 15 has 65535 nodes. As with full ternary trees, the trivial implementation is, again, expected to perform better than the link/cut tree because the tree is balanced. However, it is not as clear if the performance will increase or decrease compared to ternary trees. The average execution time of the trivial implementation is expected to increase because the average depth of nodes in full binary trees is higher than in full ternary trees. The effect on the link/cut tree is not as apparent. Therefore, the ratio could improve or worsen depending on if the execution time of the link/cut tree increases or decreases relatively more than that of the trivial implementation. The test will be carried out using the same method that was used when the implementations were tested on ternary trees. The resulting graph is depicted in Figure 4.15.

Figure 4.16 shows a comparison between the test on binary and the test on ternary trees. It only includes heights that were tested for both tree types. At a height of 5 the performance is about equal. In both cases the link/cut tree is

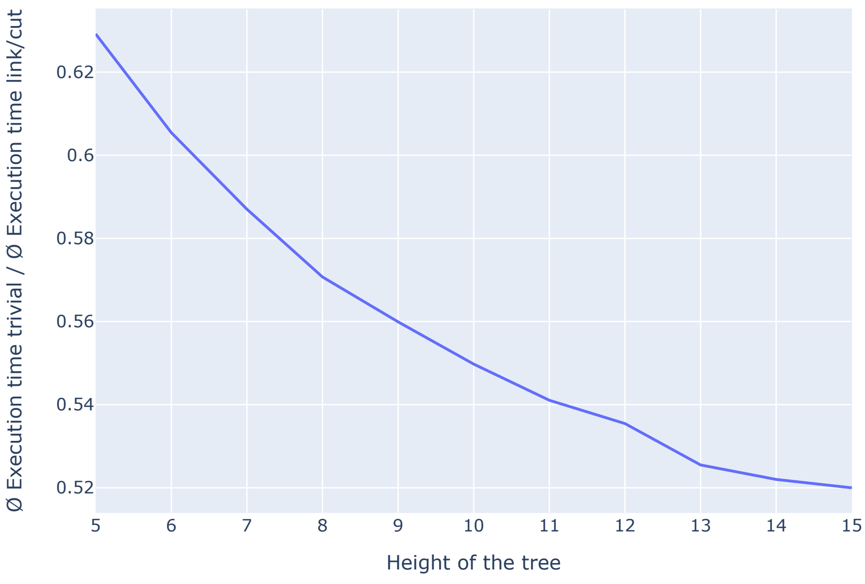

Figure 4.15: Performance for binary trees for tree heights from 5 to 15

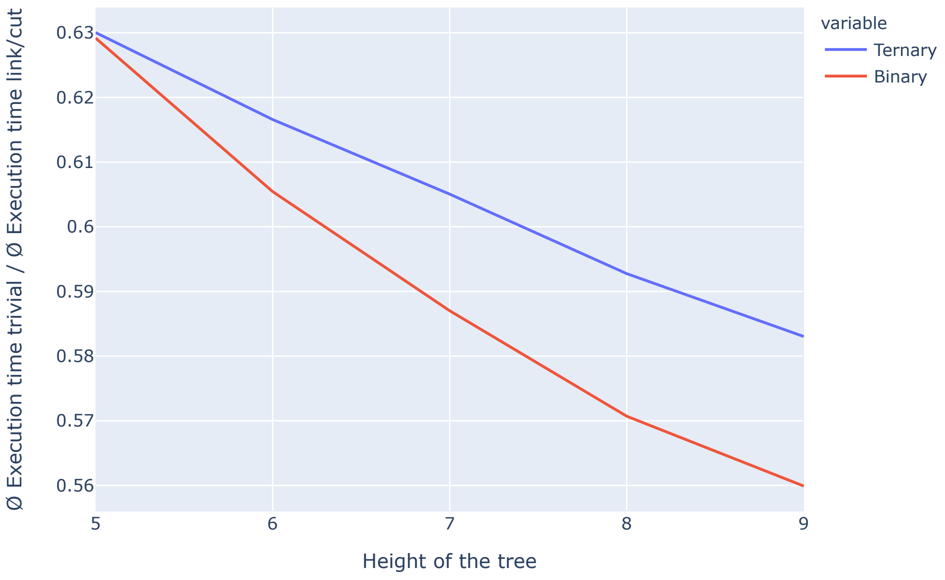

Figure 4.16: Performance for binary trees and ternary trees for tree heights from 5 to 9

about 37 percent slower compared to the trivial implementation. Although the performance decreases for both types of trees at heights larger than 5, it decreases less for ternary trees. At height 8 the link/cut tree is about 43 percent slower when tested on binary trees, but only 41 percent slower when tested on ternary trees. If the number of nodes of a ternary and a binary tree is approximately equal, as is the case for a binary tree of size 6 (127 nodes) and a ternary tree of size 4 (121 nodes), then the link/cut tree performs better on full ternary trees than on full binary trees. It is about 35 percent slower for ternary trees but 40 percent slower for binary trees.

Bibliography

[1] Eric Demaine. 6.851: Advanced Data Structures, Lecture 19. 2012. https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf.

[2] Eric Demaine. 6.851: Advanced Data Structures, Recording of Lecture 19. 2013. https://www.youtube.com/watch?v=XZLN6NxEQWo.

[3] Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 01 1970.

[4] Jeff Erickson. CS 573: Topics in Analysis of Algorithms, Lecture 7. 2006. https://jeffe.cs.illinois.edu/teaching/datastructures/2006/notes/07-linkcut.pdf.

[5] Andrew Goldberg and Robert Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. volume 36, pages 388–397, 01 1988.

[6] Phillip Klein and Shay Moses. *Optimization Algorithms for Planar Graphs*, chapter Splay trees and link-cut trees. Unpublished Book Draft, pages 1–28.

[7] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 26(3):114–122, 1981.

[8] Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[9] Robert Endre Tarjan and Renato Fonseca Werneck. Dynamic Trees in Practice. *ACM J. Exp. Algorithmics*, 14, January 2010.

Declaration of Authorship / Eidesstattliche Erklärung

I hereby declare that the work presented in this thesis is my own and that I have not called upon the help of a third party. In addition, I affirm that neither I nor anybody else has previously submitted this work or parts of it to obtain credits elsewhere. I have clearly marked and acknowledged all quotations and references that have been taken from the works of others. All secondary literature and other sources are marked and listed in the bibliography. The same applies to all charts, diagrams and illustrations as well as to all Internet resources. Moreover, I consent to my paper being electronically stored and sent anonymously in order to be checked for plagiarism. I know that if this declaration is not made, the paper may not be graded.

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

F. Krömer

Mannheim, May 30, 2021 Signature

Assignment of Usage Rights / Abtretungserklärung

I hereby grant the University of Mannheim, Chair of Practical Computer Science III, Prof. Dr. Guido Moerkotte, extensive, exclusive, unlimited and unrestricted usage rights to the work described in this thesis. This includes the right to use the results in research and teaching. In particular, this includes the right to reproduce, distribute, translate, change and transfer the results to third parties, as well as any further results derived from them. Whenever the results of my work are used in their original form or in a revised version, I will be mentioned by name as a co-author, in compliance with the copyright rules. This assignment does not include any commercial use.

Hinsichtlich meiner Masterarbeit räume ich der Universität Mannheim/Lehrstuhl für Praktische Informatik III, Prof. Dr. Guido Moerkotte, umfassende, ausschließliche unbefristete und unbeschränkte Nutzungsrechte an den entstandenen Arbeitsergebnissen ein.

Die Abtretung umfasst das Recht auf Nutzung der Arbeitsergebnisse in Forschung und Lehre, das Recht der Vervielfältigung, Verbreitung und Übersetzung sowie das Recht zur Bearbeitung und Änderung inklusive Nutzung der dabei entstehenden Ergebnisse, sowie das Recht zur Weiterübertragung auf Dritte.

Solange von mir erstellte Ergebnisse in der ursprünglichen oder in überarbeiteter Form verwendet werden, werde ich nach Maßgabe des Urheberrechts als Co-Autor namentlich genannt. Eine gewerbliche Nutzung ist von dieser Abtretung nicht mit umfasst.

F. Krömer

Mannheim, May 30, 2021 Signature