Group 44:
Ricardo Rei nº 78047
Miguel Carvalho nº 78052

TÉCNICO LISBOA

MSc in Computer Science and Engineering

Advanced Algorithms 2016-2017

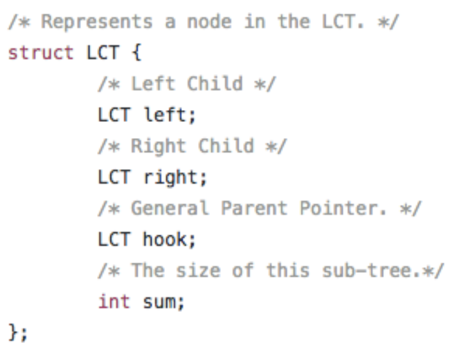
Connectivity in Forests:

In this project students had to implement an API to maintain the connectivity information of an underlying forest. Maintaining the connectivity of information is a well known problem for computing nearest common ancestor, solving various network flow problems including finding maximum flows, blocking flows, and acyclic flows, computing certain kinds of constrained minimum spanning trees and implementing the network simplex algorithm for minimum-cost flows.

1. INTRODUCTION:

The goal of this project is to maintain the connectivity information of an underlying forest. To store this information, we are going to use the Link/Cut tree data structure (with some minor changes that we will explain in section 2 and 3). The motivation behind this is that if we represent the forest as a regular collection of parent pointer trees, it might take us a long time to find the root of a given node, however if we represent each tree in the forest as a Link/Cut tree we can do all operations in *O(log(n))* amortized time, where *n* is the number of nodes in the structure.
The forest in this project consists of a set of vertexes linked by edges, such that no set of edges forms a cycle.
The API developed to solve this project consists in 3 commands: L, C and Q which we will explain better in sections 3.

2. DATA STRUCTURE:



```
/* Represents a node in the LCT. */
struct LCT {
        /* Left Child */
        LCT left;
        /* Right Child */
        LCT right;
        /* General Parent Pointer. */
        LCT hook;
        /* The size of this sub-tree.*/
        int sum;
};
```

Figure 1 - LCT node structure in C.

The Link/Cut tree's underlying structure is shown in figure 1.
The left and right pointers are used to represent the pointers in the auxiliary splay tree. The hook contains the parent of that node. If the parent points back to the node either by a right pointer or a left pointer than it represents a parent otherwise the hook is representing a path pointer and the node pointed by the hook is a path parent. The sum field should represent the depth of the node but in the context of this project it is only used as a bit. If the sum field is -1 than the tree is conceptually reversed, meaning that the order of the nodes in the sub-tree is from right to left instead of left to right.

Group 44:
Ricardo Rei nº 78047
Miguel Carvalho nº 78052

3. AUXILIARY FUNCTION

In order to better understand the project API, it's required to understand 3 functions: *splay*, *access* and *reRoot*.
The *access* and *splay* functions are responsible for the logarithmic time bound on the desired commands and together with *reRoot* keep our data structure updated.

(1) splay (u):

This function is similar to a move-to-root where it does rotations bottom-up along the accessed path and moves the node u until it is the root of the tree. These rotations are done in pairs, according to the structure of the accessed path, with the auxiliary function called *splayingStep*.
The *splayingStep* auxiliary function receives a node and evaluates 3 main cases:

Case 1 (zig):
if the parent of u is the tree root, rotates the edge joining u with his parent (terminal case).

Case 2 (zig-zig):
if the parent of u is not the root and u and his parent are both left or both right children, rotate the edge joining the parent of u with its grandparent, and then rotate the edge joining u with his parent.

Case 3 (zig-zig):
if the parent of u is not the root and u is a left child and his parent is a right child, or vice-versa, rotate the edge joining u with his parent and then rotate the edge joining u with his new parent.

(2) access (array, u):

The *access* function receives an array with all the nodes and an Int *u* that represents the position of the node in that array that we want to access. This function changes the represented tree in order to make the path from the root to node *u* the preferred path. To do this we start by splaying *u*, which brings *u* to the root of the auxiliary tree. We then disconnect the right sub-tree. The root of the disconnected tree will have a hook (of type path-parent pointer) to *u*.
After that we walk up in the represented tree changing the preferred paths. To do this we follow the hook (representing a path pointer, since u was splayed), while that hook is not NULL we splay the node pointed by the hook and then change the preferred child of that node to *u*, finally we splay *u* again.
Although we only present the underlying data-structure of the nodes in the next section the access code is already shown in this section for logical reasons.

```
void access(LCT t, int v)
{
        LCT node = &t[v];
        LCT w = NULL;
        splay(node);
        node->right = NULL;

        while (node->hook !=  NULL)
        {
                w = node->hook;
                splay(w);
                w->right = node;
                splay(node);
        }
}
```

Figure 2 - C implementation of access operation. Receives an array of nodes, and the position of the node we want to access.

(3) reRoot (array, u):

The *reRoot* function receives an array with all the nodes and an Int *u* that represents the position of the node in that array that we want to make the root of the represented tree. In order to make *u* the root of the represented tree we first need to access *u*. After accessing *u*, *u* will end up in the top of his auxiliary tree with no right children. We now need to flip *u* (meaning that the order of the nodes in the sub-tree is from right to left instead of left to right) and perform another access to *u* in order to unflip it.
After using the reRoot function *u* is the left most node of the represented tree.

```
void reRoot(LCT t, int v)
{
        access(t, v);
        t[v].sum *= -1; /* invert node */
        access(t, v);
}
```

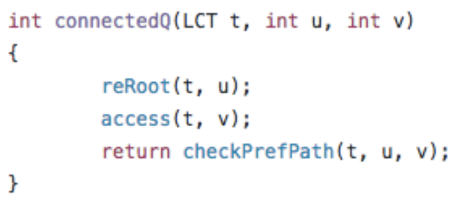
Figure 3 - reRoot operation implemented in C.

4. PROJECT API:

4.1 Command Q:

Command Q is followed by a single space, followed by vertex number u, followed by single space, followed by vertex number v and enter. This command will call the *connectedQ* function and return T if u and v are connected and F otherwise.

The *connectedQ* function receives the array containing all the nodes and two Ints *u* and *v* that correspond to the index of the nodes *u* and *v*, which are the nodes we want to verify whether or not they are connected. The *connectedQ* function asserts True or False if the two nodes *u* and *v* are connected through some path or not, respectively. In order to do this and since in the context of the project there is no fixed root we first define *u* as the root of the represented tree with the *reRoot* function. Then we *access v* which brings *v* to the top of the auxiliary tree. At last, since *u* is the root of the represented tree, *u* is the left most node, and *v* is on the top of his auxiliary tree. If we start walking up in the auxiliary tree of *u* and we end up in *v*, then the two nodes are connected.



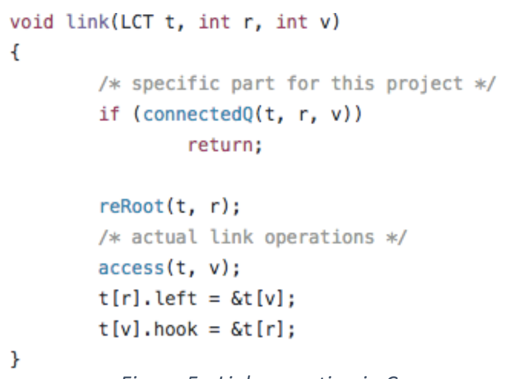
```
int connectedQ(LCT t, int u, int v)
{
        reRoot(t, u);
        access(t, v);
        return checkPrefPath(t, u, v);
}
```

Figure 4 - connectedQ function.

Note: The checkPrefPath is an auxiliary function that walks up the path of u until it finds v.

4.2 Command L:

Command L is followed by a single space, followed by vertex number u, followed by single space, followed by vertex number v and enter. This command will call the *link* function linking u and v, if possible.
The *link* function receives the array containing all the nodes and two Ints *r* and *v* that correspond to the index of the nodes *r* and *v* we want to link. The link function combines 2 trees into 1 by adding an edge between nodes *r* and *v*. For this operation to be performed we have to assure that there was no edge already linking *v* and *w*, and also that they are not connected through some other path, because this would create a cycle. In order to verify these conditions, the *link* function starts by calling the *connectedQ* function and if *connectedQ* returns true, *link* does nothing. Otherwise we perform a *reRoot* to *r* and then *access v*. These two operations will guarantee that *r* is the left most node of his tree and *v* is on the top of his tree, and we can simply *link* them by making *v* the left child of *r* and making the hook of *v* point to *r*.



```
void link(LCT t, int r, int v)
{
        /* specific part for this project */
        if (connectedQ(t, r, v))
                return;

        reRoot(t, r);
        /* actual link operations */
        access(t, v);
        t[r].left = &t[v];
        t[v].hook = &t[r];
}
```

Figure 5 - Link operation in C.

4.3 Command C:

Command C is followed by a single space, followed by vertex number *u*, followed by single space, followed by vertex number *v* and enter. This command will call the *cut* function between *u* and *v*.
The cut function receives the array containing all the nodes and two Ints *r* and *v* that correspond to the index of the nodes *r* and *v* we want to cut. The cut function removes the edge between 2 nodes and as a result the node *r* becomes the root of the new sub-tree.
In order to perform this operation, we start by making *r* the root of his represented tree and performing an access to *v*. If the left child of *v* ends up being *r* and *r* has no right child, then we can cut them by making the hook of *r* point to NULL and removing the left child of *v*.

```c
void cut(LCT t, int r, int v)
{
        /* specific part for this project */
        reRoot(t, r);
        /* actual link operations */
        access(t, v);
        if (t[v].left == &t[r] && t[r].right == NULL)
        {
                t[v].left->hook = NULL;
                t[v].left = NULL;
        }
}
```

Figure 6 - cut function implemented in C.

5. EXPERIMENTAL ANALYSIS:

In our tests we wanted to show that our API has a *O(log(n))* amortized complexity, where *n* is the number of nodes in the structure. The tests we present next were run in a MacBook Air, Processor 1,4 GHz Intel Core i5 with a memory of 4GB 1600 MHz DDR3. This laptop runs a OS X EL Capitan Version 0.11.6.
The tests consisted in a set of inputs with varying number of nodes and 1MB operations where 75% of them are links and 25% are cuts. We choose this ratio of link and cuts because we wanted a high percentage of nodes linked together. The Q operation is never called since the L operation uses Q and Q is being implicitly called by L.
For each input we analyzed the time of each operation and we made a sum of all those times. The initial idea was to divide the sum by the number of operations but since the number of operations is constant and is super high, if we divide the sum by the number of the operations we end up with really small numbers. So we decide to only use the sum of all the operations.

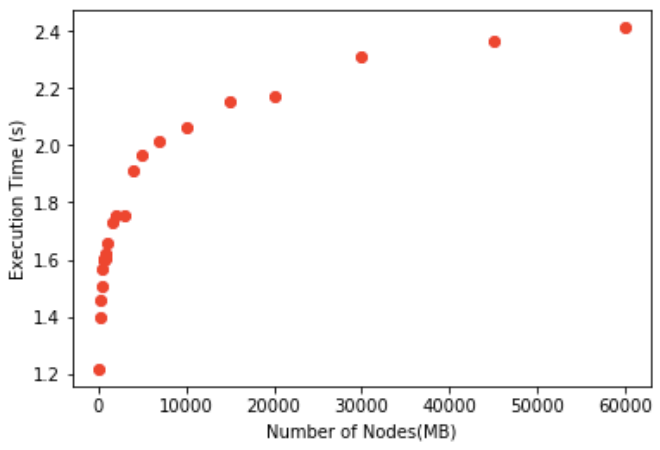


Figure 7 - Execution time vs number of nodes in the structure.

As we can see by the results showed in figure 7 the execution time of the sum of the times of 1MB operations demonstrates a logarithmic behavior with the variation of the number of nodes.

6. DISCUSSION:

For this project the goal was to develop an API for maintaining the connectivity of information using the Link Cut Tree structure that allows the *link*, *cut* and *connectedQ* operation to be executed in *O(log(n))* where n is the number of vertexes. After implementing all the necessary operations, we demonstrate with an experimental analysis that, in fact, this API allows these operations to be executed on the requested *O(log(n)).*

7. REFERENCES:

- Self-Adjusting Binary Search Trees, Daniel Dominic Sleator and Robert Endre Tarjan 1985

- A Data Structure for Dynamic Trees, Daniel D.Sleator and Robert Endre Tarjan 1985