

UNIVERSIDADE DE SÃO PAULO  
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**Estruturas de dados retroativas**  
*Um estudo sobre Union-Find e ...*

Felipe Castro de Noronha

MONOGRAFIA FINAL  
MAC 499 — TRABALHO DE  
FORMATURA SUPERVISIONADO

Supervisora: Prof<sup>a</sup>. Dr<sup>a</sup>. Cristina Gomes Fernandes

São Paulo  
2022

*O conteúdo deste trabalho é publicado sob a licença CC BY 4.0  
(Creative Commons Attribution 4.0 International License)*

*Dedico este trabalho a meus pais e todos aqueles que me ajudaram durante esta caminhada.*



[illegible]



## Resumo

Felipe Castro de Noronha. **Estruturas de dados retroativas: Um estudo sobre Union-Find e ...**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

[illegible]

**Palavras-chave:** Palavra-chave1. Palavra-chave2. Palavra-chave3.





# Abstract

Felipe Castro de Noronha. **Retroactive data structures: A study about Union-Find** *and*. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

[illegible]

**Keywords:** Keyword1. Keyword2. Keyword3.



# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Retroatividade Parcial . . . . .	1
1.2	Retroatividade Total . . . . .	1
<b>2</b>	<b>Link-Cut Trees</b>	<b>3</b>
2.1	Ideia . . . . .	3
2.2	Definições . . . . .	3
2.3	Splay Trees . . . . .	4
2.3.1	Splay . . . . .	5
2.3.2	Rotate . . . . .	6
2.3.3	Métodos auxiliares . . . . .	6
2.4	Operações . . . . .	6
2.4.1	Access . . . . .	6
	<b>Referências</b>	<b>7</b>



# Capítulo 1

## Introdução

Estruturas de dados retroativas bla bla bla

### 1.1 Retroatividade Parcial

### 1.2 Retroatividade Total



## Capítulo 2

# Link-Cut Trees

Neste capítulo, apresentaremos a estrutura de dados Link-Cut Tree, introduzida por **SLEATOR e TARJAN (1981)**. Esta árvore serve como base para as estruturas retroativas apresentadas nos próximos capítulos.

### 2.1 Ideia

A Link-Cut Tree é uma estrutura de dados que nos permite manter uma floresta, onde os nós de cada árvore possuem um número arbitrário de filhos. Igualmente importante, essa estrutura nos fornece o seguinte conjunto de operações:

- *make\_root(u)*: enraíza no vértice  $u$  a árvore que o contém.
- *link(u, v, w)*: dado que os vértices  $u$  e  $v$  estão em árvores separadas, transforma  $v$  em raiz de sua árvore e o liga como filho de  $u$ , colocando peso  $w$  na nova aresta criada.
- *cut(u, v)*: retira da árvore a aresta com pontas em  $u$  e  $v$ , efetivamente separando estes vértices e criando duas novas árvores.

Além disso, a Link-Cut Tree possui a capacidade de realizar operações agregadas nos vértices, isto é, consultas acerca de propriedades de uma sub-árvore ou de um caminho entre dois vértices. Em particular, estamos interessados na rotina *maximum\_edge(u, v)*, que nos informa o peso máximo de uma aresta no caminho entre os vértices  $u$  e  $v$ .

Todas essas operações consomem tempo  $O(\log n)$  amortizado, onde  $n$  é o número de vértices na floresta.

### 2.2 Definições

Primeiramente, precisamos fazer algumas definições acerca da estrutura que vamos estudar.

Chamamos de árvores representadas as árvores genéricas que nossa estrutura sintetiza.

Para a representação que a Link-Cut Tree utiliza internamente dividimos uma árvore representada em caminhos vértice-disjunto, os caminhos preferidos. Por conveniência, definimos o início de um caminho preferido como o vértice mais profundo contido nele.

Se uma aresta faz parte de um caminho preferido, chamamos ela de aresta preferida. Ademais, colocamos a propriedade de que um vértice pode ter no máximo uma aresta preferida com a outra ponta em algum de seus filhos. Caso tal aresta exista, ela liga um vértice á seu filho preferido.

Finalmente, representamos cada caminho preferido com uma árvore auxiliar, no caso, uma Splay Tree (seu funcionamento é explicado na próxima seção). Para isso, cada vértice é armazenado na árvore auxiliar utilizando sua profundidade como chave de ordenação. Ademais, cada árvore auxiliar possui um ponteiro para o caminho preferido imediatamente acima de seu fim, exceto no caminho preferido que contem a raiz da árvore representada.

**TODO:** colocar imagem de uma arvore representada e sua arvore auxiliar.

## 2.3 Splay Trees

No artigo original, os autores utilizam uma árvore binaria enviesada como estrutura para as árvores auxiliares. Porém, 4 anos depois, [SLEATOR e TARJAN \(1985\)](#) apresentaram a Splay Tree, que possibilita realizarmos as operações necessárias para a manipulação dos caminhos preferidos em tempo  $O(\log n)$  amortizado, com uma implementação muito mais limpa do que a da versão original.

A Splay Tree é uma árvore binaria de busca auto-ajustável, capaz de realizar as operações de inserção, deleção e busca. Em particular, para seu uso como arvore auxiliar, estamos interessados na sua operação *splay*, que traz um nó para a raiz da árvore através de suscetivas rotações. Mas antes de nos aprofundarmos neste método, vamos examinar como os caminhos preferidos são representados aqui.

Primeiramente, em nosso uso, a ordenação dos nós na Splay Tree é dada pela profundidade destes na Link-Cut Tree. Note que, não guardamos explicitamente esses valores, ao invés disso utilizamos a ideia de chave implícita, isto é, só nos preocupamos em manter a ordem relativa dos nós após as operações de separação e união das árvores. A contrapartida deste método é perda da capacidade de realizarmos buscas por chave na Splay Tree, porem isto não nos representa um problema.

Ademais, para podermos lidar com os pesos nas arestas da Link-Cut Tree, fazemos com que cada aresta da árvore representada vire um nó na árvore auxiliar. Isso nos permite calcular o peso máximo de uma aresta em um caminho preferido, dado que podemos facilmente obter o peso máximo de um vértice em uma Splay Tree.

**TODO:** colocar imagem de um preferred path e sua respectiva splay tree.

Além disso, como usamos a profundidade dos nós na árvore representada como chave para a árvore auxiliar, temos que todo os nós na sub-arvore esquerda da raiz de uma Splay Tree tem uma profundidade menor que a raiz, enquanto os nós a direita tem uma profundidade maior. Contudo, ao realizamos uma operação *make\_root(u)* fazemos com



que todos os nós que estavam acima de  $u$  na árvore representada se tornem parte de sua sub-árvore. Para isso, incluímos na Splay Tree um mecanismo para inverter a ordem de todos os filhos de uma árvore auxiliar, efetivamente invertendo a orientação de um caminho preferido.

**TODO:** colocar imagem de uma Splay antes e depois da inversão, assim como sua árvore representada.

Com isso, os nós da árvore auxiliar tem os seguintes campos:

- *parent*: apontador para o pai na Splay Tree. Caso o nó em particular seja a raiz da árvore auxiliar, este campo armazena um ponteiro para o vértice que está logo acima do fim deste caminho preferido na árvore representada.
- *left\_child* e *right\_child*: apontadores para os filhos de um nó na Splay Tree.
- *value*: utilizado para guardar o peso de uma aresta da árvore representada transformado em vértice na árvore auxiliar.
- *is\_reversed*: valor booleano para sinalizar se uma sub-árvore deve ter sua ordem invertida ou não, isto é, se todas as posições de filhos esquerdos e direitos devem ser invertidos nessa sub-árvore.
- *max\_subtree\_value*: guarda o valor máximo armazenado na sub-árvore do respectivo nó.

### 2.3.1 Splay

Com a estrutura apresentada, podemos partir para a explicação de sua principal operação, a *splay*. Em poucas palavras, este método é responsável por receber um nó e fazer com que ele vire a raiz da Splay Tree, através de diversas rotações. Ademais, as operações de *splay* contribuem para diminuir a profundidade da árvore, melhorando o seu consumo de tempo.

**TODO:** Colocar figura de uma Splay antes e depois do Splay em uma folha

De modo a facilitarmos nossa explicação, chamamos *parent* o pai de um nó  $u$  e de *grandparent* o pai de *parent*. Primeiramente, recebemos um nó  $u$  da Splay Tree, e enquanto este nó não é raiz de nossa árvore, fazemos a seguinte rotina:

- Checo se *parent* é a raiz da árvore, caso positivo, vou para o último item.
- Caso contrário, iremos propagar o valor booleano *is\_reversed* de *grandparent* e em seguida o de *parent*, fazendo as devidas reversões caso necessárias. Isso nos fornece a invariante de que iremos fazer a comparação a seguir entre os filhos corretos.
- Em seguida, checamos se *grandparent*, *parent* e  $u$  estão em uma orientação de *zig-zig*, *zag-zag* ou *zig-zag*, como exemplificadas na figura abaixo. Dependendo da orientação, fazemos uma rotação em  $u$  ou em *parent*, sempre com a ideia de diminuirmos em 1 a profundidade de  $u$ .
- Por último, fazemos uma rotação em  $u$ , o que o coloca na posição que inicialmente estava o nó *grandparent*.

**TODO:** Colocar figura mostrando configurações de zig-zig, zag-zag e zig-zags.

Ao sair da função *splay*, o nó *u* estará na raiz de sua árvore auxiliar. Além disso, seu valor booleano *is\_reversed* estará nulo, pois as reversões já terão sido propagadas aos seus filhos, e seu *max\_subtree\_value* vai estar atualizado, contendo o maior valor presente na Splay Tree.

**TODO:** Colocar código ou pseudocódigo da função *splay*?

### 2.3.2 Rotate

Agora, vamos olhar a função responsável por realizar as rotações. Basicamente ela pode ser fatorada em 4 partes:

- Primeiramente propagamos as reversões de *grandparent*, *parent* e *u*, garantido que estaremos acessando e manipulando os filhos corretos destes respectivos nós.
- Em seguida, caso o *parent* não seja a raiz da Splay Tree, trocamos ele de lugar com *u*, efetivamente colocando *u* como algum dos filhos de *grandparent*.
- Agora, basta colocarmos *parent* como algum dos filhos *u*, espelhando a orientação inicial em que *u* estava como filho de *parent*.
- Por último, recalculamos os valores máximos nas sub-árvores de *parent* e de *u*.

### 2.3.3 Métodos auxiliares

Para finalizar, nossa Splay Tree possui 2 métodos auxiliares, o *join\_right\_subtree* e o *split\_left\_subtree*.

O primeiro vai ser utilizado para a implementação da função *access()* na Link-Cut Tree. Essa rotina nos permite ligarmos uma nova sub-árvore ao filho direito do nó atual. Já o segundo será usado na função *cut()*, ele efetivamente separa a sub-árvore esquerda de um dado nó, criando 2 novas Splay Trees no processo.

Com isso, temos todas as ferramentas necessárias para manipularmos a Splay Tree em seu uso como árvore auxiliar.

## 2.4 Operações

### 2.4.1 Access

## Referências

- [SLEATOR e TARJAN 1981] Daniel D. SLEATOR e Robert Endre TARJAN. “A data structure for dynamic trees”. Em: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pgs. 114–122. ISBN: 9781450373920. DOI: [10.1145/800076.802464](https://doi.org/10.1145/800076.802464). URL: <https://doi.org/10.1145/800076.802464> (citado na pg. 3).
- [SLEATOR e TARJAN 1985] Daniel D. SLEATOR e Robert Endre TARJAN. “Self-adjusting binary search trees”. Em: *J. ACM* 32.3 (jul. de 1985), pgs. 652–686. ISSN: 0004-5411. DOI: [10.1145/3828.3835](https://doi.org/10.1145/3828.3835). URL: <https://doi.org/10.1145/3828.3835> (citado na pg. 4).