Estruturas de dados retroativas:
Aplicações na dinamização de algoritmos

José Wagner de Andrade Júnior

Universidade Federal de Itajubá

Mestrado em Ciência e Tecnologia da Computação

UNIFEI Junho 2020

Universidade Federal de Itajubá

José Wagner de Andrade Júnior

Estruturas de dados retroativas: Aplicações na dinamização de algoritmos

Dissertação apresentada ao Programa de Mestrado em Ciência e Tecnologia da Computação da Universidade Federal de Itajubá, como parte dos requisitos para a obtenção do título de Mestre em Ciência e Tecnologia da Computação.

Área de concentração: Ciência de Computação

Orientador: Prof. Dr. Rodrigo Duarte Seabra

Itajubá - MG
Junho / 2020

Agradecimentos

Aos meus pais e irmãos, que me incentivaram durante toda a minha jornada acadêmica.

Ao professor Rodrigo, por ter sido meu orientador e ter desempenhado tal função com dedicação e amizade, e a todos os professores que fizeram parte da minha formação.

A Universidade Federal de Itajubá, essencial no meu processo de formação profissional, pela dedicação, e por tudo o que aprendi ao longo dos anos dos cursos, tanto de graduação quanto de pós-graduação.

Aos meus amigos, por compartilharem comigo tantos momentos de descobertas e aprendizado e por todo o companheirismo ao longo deste percurso. Agradeço principalmente os meus amigos da maratona de programação, que foram fundamentais para meu desenvolvimento como programador e pessoa.

A todos aqueles que contribuíram, de alguma forma, para a realização deste trabalho.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior pela concessão da bolsa de mestrado e pelo apoio financeiro para a realização desta pesquisa.

Resumo

A retroatividade em programação é um conceito que pode ser definido como o estudo da modificação da linha temporal em uma estrutura de dados, bem como a análise dos efeitos dessa modificação através de toda a sua existência. Em geral, essa análise e implementação tendem a serem mais custosas do ponto de vista computacional, observando-se que uma modificação no passado pode gerar um efeito cascata por toda a existência dessa estrutura. O conceito de retroatividade gera ferramentas e estruturas que otimizam as soluções para a natureza desses problemas temporais. Esse tipo de estrutura pode ser utilizada nas aplicações das mais diversas naturezas, desde em algoritmos de caminho mínimo, aplicações em segurança e até em aplicações geométricas. Nessa dissertação, tem-se os subsídios teóricos sobre essas estruturas, um material detalhado sobre a implementação das estruturas mais comuns utilizando o paradigma da retroatividade, e a implementação de alguns problemas que podem ser resolvidos utilizando técnicas de retroatividade, como, por exemplo, o algoritmo de árvore geradora mínima totalmente dinâmica. Para cada estrutura, foram executados testes práticos sobre as estruturas retroativas e seu desempenho foi comparado às outras implementações dessas mesmas estruturas. Os testes mostraram que as implementações retroativas propostas por Demaine *et. al* (2007) obtiveram os melhores resultados do ponto de vista temporal. Além disso, foram propostos dois algoritmos que utilizam os conceitos de retroatividade para sua construção: o algoritmo para o problema da árvore geradora mínima totalmente retroativa e o algoritmo do caminho mínimo a partir de um vértice inicial fixo em grafos dinâmicos. Seja m o tamanho da linha temporal em que a estrutura está implementada, $V(G)$ e $A(G)$ o conjunto de vértices e arestas de um grafo G respectivamente. Foi alcançada a complexidade de tempo amortizada de $O(\sqrt{m} \cdot \log |V(G)|)$ por operação de atualização ou consulta, para o problema da árvore geradora mínima totalmente retroativa. Para o algoritmo do caminho mínimo, a partir de um vértice inicial fixo em grafos dinâmicos, por meio do algoritmo proposto por Sunita *et. al* [52], foi obtida a complexidade temporal de $O(|A(G)| \cdot \lg |V(G)|)$ por modificação, utilizando filas de prioridade com retroatividade não-consistente.

Palavras-chave: Retroatividade. Estrutura de dados. Geometria Computacional. Grafos.

Abstract

The retroactivity in programming is the study of a modification in a timeline for a data structure and the effects that this modification exerts throughout its existence. In general, the analysis and implementation tend to be more costly computationally, because a modification on these data structure in the past can generate a cascade effect through all the data structure timeline. The concept of retroactivity generates tools and structures that optimize the solutions facing these temporal problems. This type of data structure can be used in, for example, shortest path algorithms, security applications, and geometric problems. In this thesis, we have the theoretical subsidies about these data structures, a detailed material about the implementation of this structures, using retroactivity, and the implementation of some problems that retroactivity can be used, for example, the fully dynamic minimum spanning tree problem. For each data structure, we executed practical tests about this data retroactive data structures and a comparison between these solutions and other approaches. The tests showed that the retroactive implementations proposed by Demaine *et. al* (2007) [13] obtained the best results from a temporal point of view. It was proposed two algorithms which used the retroactivity concepts inside its development: the fully retroactive minimum spanning tree and the single source dynamic shortest path problem in dynamic graphs. Let m be data structure's timeline, $V(G)$ and $A(G)$ the sets of vertices and edges from graph G. We reached an amortized time complexity $O(\sqrt{m} \cdot \lg |V(G)|)$ per query/update operation in the fully retroactive minimum spanning tree algorithm. The algorithm to solve the single source dynamic shortest path problem in dynamic graphs proposed by Sunita *et. al* [52] obtained a time complexity $O(|A(G)| \cdot \lg |V(G)|)$ per modification using a non-oblivious retroactive priority queue.

Keywords: Retroactivity. Data stuctures. Geometry. Graph.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de símbolos

$\overline{G}$	Grafo com as arestas invertidas
$A(G)$	O conjunto das arestas de um grafo G
ACM	Árvore de caminhos mínimos *(Shortest path tree)*
BFS	Busca em largura *(Breadth first search)*
$C(e)$	Custo da aresta e
$e.u$	Extremo inicial da aresta e
$e.v$	Extremo final da aresta e
FPR	Fila de prioridade retroativa
m	Tamanho da linha temporal de uma estrutura de dados
MST	Árvore geradora mínima *(Minimum spanning tree)*
n	Número de elementos de uma estrutura de dados
$T(m)$	Complexidade para consultas em uma estrutura com linha temporal de tamanho m
$U(m)$	Complexidade temporal de atualização em uma estrutura com linha temporal de tamanho m
$V(G)$	O conjunto dos vértices de um grafo G

Capítulo 1

Introdução

Estruturas de dados (ED's) são objetos computacionais que permitem aos programadores desenvolverem algoritmos eficientes do ponto de vista de espaço de armazenamento e velocidade de execução. Essas estruturas são utilizadas na maioria dos aplicativos desenvolvidos atualmente, de modo implícito, na forma de bibliotecas inerentes à linguagem de programação, ou explicitamente por códigos desenvolvidos pelo programador especificamente para a aplicação.

Em geral, as ED's permitem dois tipos de operações: acesso e modificação dos dados armazenados em sua estrutura [12]. As operações de acesso consultam algum estado da estrutura, enquanto as operações de modificação alteram o estado dessas estruturas.

A alteração das ED's só permite modificações e acessos em sua configuração atual, ou seja, estados anteriores dessas estruturas são perdidos sempre que modificações são realizadas. Para solucionar essa defasagem de informação sobre estruturas no passado, criou-se o conceito de estruturas de dados ditas temporais, nas quais, de alguma forma, se torna possível acessar instâncias anteriores dessas estruturas [13].

Na literatura, existem dois tipos de estruturas de dados temporais: *persistentes* [18, 35, 47] e *retroativas* [13]. Tanto em estruturas *persistentes* quanto nas *retroativas* é possível realizar atualizações e consultas no passado, sendo que a diferença entre elas ocorre após uma atualização ser realizada na estrutura. Em estruturas persistentes, a cada atualização, uma nova versão da estrutura é criada, derivada das partes que foram modificadas na atualização. Como forma de ilustrar uma possível situação, pode-se considerar a criação de um novo ramo em um arquivo de controle de versões, em que partes do passado nesse novo ramo são alteradas, porém, sem afetar o arquivo original. Nas estruturas de dados retroativas, essa divisão com a nova versão não acontece, sendo que o foco de estudo consiste em como essa alteração afeta a estrutura em algum ponto ao longo do tempo das atualizações.

1.1 Justificativa

Com a evolução computacional, bem como face à miniaturização dos componentes de *hardware*, os sistemas atuais devem ser capazes de suportar um grande volume de dados, e, consequentemente, uma quantidade expressiva de operações. Esses dados são recebidos e as operações executadas em máquinas que nem sempre possuem requisitos adequados em termos de capacidade de processamento e memória. Em algumas dessas aplicações, é necessário que se mantenha um histórico de operações utilizadas, bem como a alteração de algumas dessas informações, o que pode gerar uma inconsistência nos dados posteriormente inseridos ou modificados. Por exemplo, um sistema de controle de temperatura que contenha vários sensores espalhados em determinada região, e que, ocasionalmente, enviam informações para o sistema. Se um dos sensores apresentar algum problema, pode ser necessária a remoção dos dados desse sensor no sistema, o que pode ocasionar uma alteração nas temperaturas máxima e mínima locais, bem como a média destas temperaturas, entre outras operações interessantes ao sistema. Caso haja muitos dados, o custo de realizar cálculos é elevado e, nesse caso, as operações podem ser suportadas por alguma estrutura retroativa.

Outra aplicação interessante da retroatividade é na área de segurança. Considere que foi descoberto em um sistema um conjunto de operações maliciosas realizadas por um usuário não autorizado. Então, é necessária a remoção das ações desse usuário, porém, sem a alteração de operações realizadas pelos usuários autorizados. Nesse caso, como somente o estado atual do sistema importa, uma solução parcialmente retroativa seria suficiente para a natureza desse problema.

A partir dessas considerações, o conceito de retroatividade gera ferramentas para que seja possível a solução desses problemas de maneira eficiente do ponto de vista computacional. Todavia, em linhas gerais, a retroatividade não é uma técnica que pode ser aplicada diretamente a todas as estruturas de dados [13].

A menção de estruturas retroativas no trabalho de Demaine *et al.* [13] abriu possibilidades para a dinamização de algoritmos, antes considerados estáticos e difíceis de serem adaptados. Com isso, partes da instância do problema puderam ser modificadas após a execução completa do algoritmo. Em seu trabalho, Demaine *et al.* [13] propõem as ED's, focando exclusivamente nos aspectos teóricos dessas estruturas. Sua proposição com relação a essas ED's permite a dinamização de alguns algoritmos como o algoritmo de Dijkstra e o algoritmo *union-find* para a união de conjuntos. Existem outros algoritmos passíveis de serem dinamizados, área na qual esta pesquisa se propõe, além da implementação dessas ED's.

O estudo e implementação das ED's e soluções relacionadas à retroatividade têm grande interesse de pesquisadores da área, e abre caminho para a solução eficiente de problemas que são enfrentados pelos usuários, otimizando o uso de processamento e memória por parte de

seus dispositivos. Isso torna essa pesquisa totalmente viável e de grande valia para futuros estudos na área, podendo atingir uma quantidade expressiva de usuários dessas ED's ou de problemas que as utilizem.

1.2 Objetivos

O objetivo geral desta dissertação consiste em estudar estruturas de dados retroativas no que tange à dinamização de algoritmos, abrangendo suas implementações, as análises de alguns problemas e aplicações, bem como a proposição de possíveis soluções nas quais essas estruturas possam ser utilizadas.

Estruturas de dados avançadas relacionadas a retroatividade não são abordadas comumente em cursos de graduação e pós-graduação no Brasil. Somente uma aula de um curso de estruturas de dados avançadas do *Massachusetts Institute of Technology* (MIT 6.851) foi encontrada, aula essa em que essas ED's foram apresentadas. Espera-se que este trabalho possa ser utilizado por docentes em cursos de estruturas de dados avançadas, bem como represente a criação de um material, em português, relacionado ao tema em questão.

A partir do objetivo geral, os objetivos específicos da pesquisa são:

- Compilar uma lista dos algoritmos de retroatividade nas estruturas de dados mais clássicas;
- Propor alguns problemas e soluções relacionadas a cada situação;
- Comparar implementações das estruturas de dados retroativas com relação a alguns critérios como tempo de execução, consumo de memória, entre outros;
- Criar um repositório aberto à comunidade científica com as implementações desses algoritmos na linguagem C++.

1.3 Organização da dissertação

O texto está dividido em cinco capítulos com base na seguinte estrutura:

- Capítulo 1 - contém a introdução, a justificativa e os objetivos da pesquisa;
- Capítulo 2 - realiza uma breve contextualização sobre estruturas de dados temporais e a apresentação de alguns trabalhos relacionados a retroatividade;
- Capítulo 3 - contém a aplicação da retroatividade em algumas estruturas de dados;

- Capítulo 4 - apresenta alguns problemas nos quais a retroatividade pode ser aplicada;
- Capítulo 5 - contém testes práticos realizados as estruturas de dados retroativas propostas;
- Capítulo 6 - engloba as conclusões desta pesquisa e as propostas para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta subsídios teóricos sobre temas relacionados a estruturas de dados temporais. A teoria sobre estruturas de dados persistentes e retroativas tem suma importância na elaboração de algoritmos complexos relacionados à dinamização de problemas estáticos, como o problema de caminhos mínimos em grafos, árvore geradora mínima, união de conjuntos, entre outros. A primeira seção consiste em uma breve introdução sobre estruturas de dados. A segunda seção explica os tipos de estruturas de dados temporais existentes na literatura. Na terceira seção, tem-se a definição de estruturas de dados persistentes, seguida da exposição da base teórica relacionada a estruturas de dados retroativas, apresentada na quarta seção. Finalmente, na quinta seção, é apresentado um conjunto de trabalhos selecionados referentes a estruturas de dados retroativas.

2.1 Estruturas de dados

Uma estrutura de dados é um conjunto de valores e seus relacionamentos, além das operações sobre esses valores e relacionamentos [12, 40]. As ED's são escolhidas de acordo com a necessidade do problema, sendo que algumas priorizam a natureza das operações exercidas sobre os valores e relacionamentos, porém consumindo mais espaço em memória ou custo computacional de processamento. Com as ED's, também é possível utilizar os conceitos de programação orientada a objetos, como a abstração e o encapsulamento.

Dentre esses conjuntos de relacionamentos, há dois tipos principais de estruturas. Estruturas lineares são aquelas que mantém seus itens de forma independente de seus conteúdos, ou seja, os valores dos conteúdos armazenados não interferem no funcionamento da estrutura. Exemplos dessas estruturas são filas, pilhas e listas encadeadas. Já as estruturas associativas são aquelas que levam em consideração os atributos dos valores armazenados, e esses valores,

por vezes, podem afetar o estado final da estrutura. Como exemplo desse tipo de estrutura, tem-se filas de prioridade e árvores binárias.

Inicialmente, estruturas de dados são objetos que perdem informação sobre seu passado à medida que são aplicadas operações sobre os dados nelas contidos. Por exemplo, ao remover um objeto qualquer do topo de uma pilha, perde-se informação sobre versões anteriores da estrutura. Em outras palavras, só é possível consultar o elemento do topo da pilha correspondente ao tempo atual. Contudo, em alguns problemas, é necessário que se tenha acesso às versões anteriores da estrutura, e, em alguns casos, observar a modificação dessa estrutura ao realizar uma operação no passado.

2.2 Estruturas de dados temporais

Existem aplicações nas quais seria interessante obter algumas informações sobre como a estrutura se comportaria ao se realizar uma modificação em seu passado. Uma das estratégias para a solução desse tipo de problema consiste em fazer o *rollback*[1] de todas as operações até o tempo da alteração e depois refazê-las na complexidade correspondente à estrutura de dados não retroativa. Todavia, em algumas aplicações, realizar o *rollback* das operações pode ser muito custoso, e nem sempre é uma solução eficiente.

Na literatura, existem dois tipos de estruturas de dados temporais: *persistentes* [18, 35, 47] e *retroativas* [13]. Tanto em estruturas *persistentes* quanto em estruturas *retroativas* é possível realizar atualizações e consultas no passado. A diferença entre elas ocorre após uma atualização ser realizada na estrutura.

Em estruturas persistentes, a cada atualização, uma nova versão da estrutura é criada, derivada das partes da estrutura que foram modificadas nessa atualização. Para fins de exemplificação, seria como a criação de um novo ramo em um arquivo de controle de versão, em que partes do passado nesse novo ramo são alteradas, porém sem afetar o arquivo original.

Em estruturas de dados retroativas, essa divisão com a nova versão não acontece, e o objeto de estudo consiste em como essa alteração afeta a estrutura em algum ponto ao longo do tempo das atualizações.

2.3 Persistência

Estruturas de dados persistentes foram introduzidas por Driscoll, Sleator e Tarjan [18, 35, 47] e permitem realizar operações em versões já existentes da estrutura. Na literatura existem

[1] O termo *rollback* é utilizado para realizar a recuperação de informação de alguma estrutura de dados reprocessando informações sobre ela

dois tipos de ED's persistentes. Uma ED é *parcialmente* persistente quando é possível realizar operações de acesso em versões antigas da estrutura e promover modificações somente na versão mais nova. Já uma ED é *totalmente* persistente se permite operações de acesso e modificação em todas as versões temporais da estrutura.

Pode-se considerar as versões de uma estrutura de dados persistente como vértices em um dígrafo G com $|V(G)|$ vértices, onde $v \in V$ é uma versão da estrutura e $|A(G)|$ arestas, sendo que uma aresta em $A(G)$ conecta um par de vértices (a,b) se a versão b da estrutura foi criada a partir da versão a da estrutura. Se a ED for *parcialmente* persistente, o dígrafo consiste em um caminho simples, enquanto se a ED for *totalmente* persistente, o dígrafo é representado por uma árvore enraizada na primeira versão da estrutura, que corresponde ao conjunto de dados vazio.

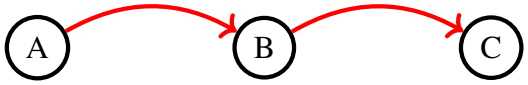

Figura 2.1 Persistência Parcial. Fonte: O autor.

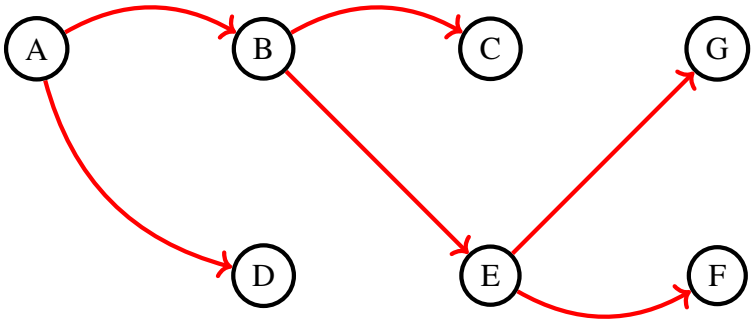

Figura 2.2 Persistência Total. Fonte: O autor.

Na Figura 2.1 tem-se um exemplo de como seria o dígrafo para os estados em uma estrutura parcialmente persistente. Nesse caso, como uma nova estrutura sempre é criada a partir de uma versão mais recente, as novas versões vem do último nó adicionado, constituindo um caminho. No exemplo ilustrado na Figura 2.2, verifica-se a possibilidade de partir de uma versão antiga da estrutura, gerando, assim, uma árvore enraizada na versão inicial da estrutura.

É possível tornar uma estrutura parcialmente persistente aumentando os nós de modo a armazenar todas as alterações realizadas [18]. Usando essa técnica, um nó contém informação sobre todas as operações realizadas naquela versão, ou seja, "aumenta" o nó em questão, o que dá-se o nome de *fat node*.

Outra técnica para modificação de uma estrutura para sua versão persistente é a técnica chamada *node copying*. Essa versão foi apresentada por Driscoll, Sarnak *et al.* [18]. A técnica consiste em definir um número máximo de modificações por nó, fazendo sua cópia quando o número de operações sobre esse nó exceder o valor máximo definido.

Em estruturas de dados mais complexas, como árvores de segmento, a modificação de uma dada versão consiste na modificação de vários nós dessa árvore. Nesse caso, existe também a noção de *path copying*, em que todos os nós alterados são copiados como um efeito cascata pelas versões da estrutura. Nesse caso, a complexidade da persistência nessa estrutura depende do número de nós alterados na operação.

2.4 Retroatividade

A literatura referente a retroatividade considera algumas versões de estruturas de dados com essa característica. A *retroatividade parcial* permite ao usuário saber somente como alterações que ele realiza no passado influenciam na estrutura no tempo atual. Já estruturas *totalmente retroativas* permitem ao usuário tanto a atualização quanto a consulta no passado. Também existe o conceito de *retroatividade não-consistente*, introduzido por Kanat Tangwongsan [53], que são estruturas que permitem ao usuário saber, após uma alteração no passado, qual o primeiro instante após a atualização em que a nova estrutura se tornará inconsistente, e, dessa forma, propagar a atualização em cada um dos elementos inconsistentes. Estruturas com retroatividade parcial são mais eficientes e menos complexas que estruturas com retroatividade completa.

Existem duas linhas de pesquisas na área de estruturas retroativas. A primeira diz respeito à generalização da aplicação da retroatividade nas estruturas em geral, no sentido de transformar toda estrutura em sua versão retroativa utilizando essas técnicas [13]. Já a outra linha diz respeito a uma vertente mais aplicável da retroatividade, e, por isso, é estudada caso a caso de acordo com a estrutura.

É importante notar que assim como apontado por Demaine *et. al* [13], o tempo não é uma dimensão ordinária. Em outras palavras, uma abordagem a ser pensada na implementação de estruturas retroativas diz respeito à adição de mais uma dimensão, que representa o objeto "tempo" na estrutura. Por exemplo, suponha uma fila de prioridade que, à primeira vista, seria uma estrutura na qual a adição de mais uma dimensão resolveria o problema da retroatividade nessa estrutura. Assim, nessa nova estrutura, existiriam duas dimensões, uma representando o tempo de execução das operações, e outra representando os valores adicionados na estrutura.

Na Figura 2.3, tem-se a representação visual da adição de uma dimensão temporal em uma estrutura de fila de prioridade. As linhas horizontais correspondem às linhas de

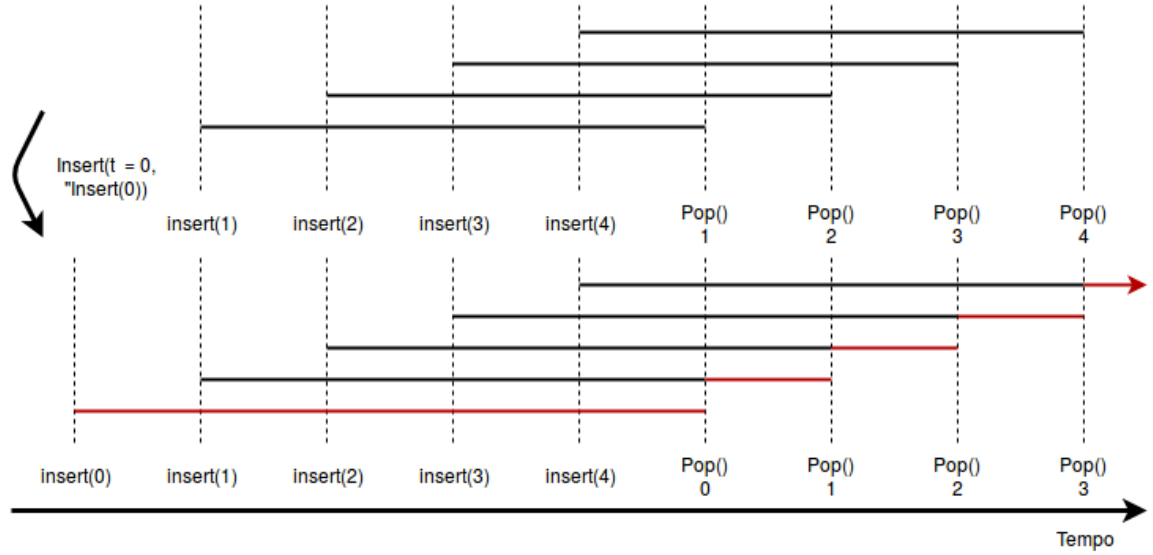

Figura 2.3 Exemplo da tentativa de utilização de duas dimensões na representação de uma fila de prioridade. Fonte: O autor.

existência de cada um dos elementos através do tempo. A posição no eixo y corresponde ao valor do elemento na estrutura. Após a realização da operação *Insert(0)* na estrutura, todos os valores removidos, utilizando-se a operação *Pop()*, são alterados. Ainda nessas figuras, as linhas de existência de cada elemento aumentam (linhas em vermelho), gerando um tipo de efeito cascata por todos os elementos. Ou seja, o efeito cascata criado pela inserção da operação não pode ser tratado simplesmente adicionando-se uma outra dimensão no problema, necessitando, portanto, de novas técnicas para tratar esse tipo de problema.

2.4.1 Retroatividade parcial

Em geral, uma ED envolve uma sequência de operações de atualização e consulta durante o tempo. Uma ED é dita *parcialmente retroativa* se, além de permitir atualizações e consultas no *tempo atual*, permite alterações no seu passado [13]. Seja $U = \{u_{t_1}, u_{t_2}, \ldots, u_{t_n}\}$ um conjunto de atualizações em que u_{t_i} é a i-ésima operação de atualização realizada no tempo t_i, tal que $t_1 < t_2 < \cdots < t_n$. Suponha, sem perda de generalização, que não existam duas atualizações ao mesmo tempo na estrutura (em algumas estruturas, a execução de duas operações concorrentemente pode gerar inconsistências).

Uma versão parcialmente retroativa de uma ED deve implementar as seguintes operações:

- *Insert(t, u):* insere no conjunto U uma operação de atualização u no tempo t, tal que $t \cap U = \emptyset$;

- *Delete(t):* remove a operação de atualização u_t do conjunto U de operações;
- *Query():* realiza uma operação de consulta na estrutura no tempo atual.

O principal desafio dessas funções consiste no efeito cascata que pode ser provocado em uma ED ao realizá-las. Uma alteração realizada no tempo t_i potencialmente pode afetar as alterações realizadas em todo t_j, tal que $t_j > t_i$. Porém, o fato de que cada consulta será realizada apenas no estado atual da estrutura permite implementações implícitas dessas mudanças cascata, e, em muitos casos, possibilita uma diminuição na complexidade temporal e espacial dessas ED's.

2.4.2 Retroatividade total

No conceito apresentado anteriormente captura-se somente uma noção parcial de retroatividade, pois, apesar de serem permitidas alterações nos estados anteriores da estrutura, não se permite a observação do passado em si. Uma estrutura é dita *totalmente retroativa* se, além de permitir alterações no passado, também permite consultas ao seu passado.

Lema 1. *Qualquer estrutura parcialmente retroativa que consome tempo $U(m)$ para atualizações e tempo $T(m)$ para consultas no presente pode ser transformada em uma versão totalmente retroativa com custo amortizado $O(U(m)\sqrt{m})$, por atualização, e tempo $O(T(m)\sqrt{m})$ para consultas, utilizando $O(U(m)\sqrt{m})$ de espaço [13].*

Demonstração. É possível definir $\sqrt{m}$ pontos em que tem-se a estrutura parcialmente retroativa correspondente às operações realizadas antes do tempo t. Escolhendo os pontos em que se tem a estrutura parcialmente retroativa otimamente, cada atualização será realizada em, no máximo, $\sqrt{m}$ pontos (os pontos definidos pela estrutura). Já as consultas no tempo t podem ser realizadas encontrando a última estrutura parcialmente retroativa no tempo $t' \leq t$, tal que t' seja um dos pontos definidos anteriormente, e, a partir dessa estrutura, realizar as operações no intervalo $[t'+1, t]$ na estrutura parcialmente retroativa, obtendo a versão da estrutura no tempo t. □

O lema proposto por Demaine *et. al* [13] define um limite superior no qual toda estrutura parcialmente retroativa pode ser transformada em sua versão totalmente retroativa, obtendo uma relação entre esses tipos de estrutura. Nesse contexto, qualquer otimização em uma versão parcialmente retroativa da estrutura pode ser transferida para a sua versão totalmente retroativa, utilizando a técnica anteriormente apresentada.

Todavia, é importante observar que isso não define o limite de diferença espacial e temporal entre esses dois tipos de estrutura. Por exemplo, em estruturas como a fila e a pilha,

é possível obter a estrutura totalmente retroativa com complexidade melhor do que utilizando a transformação genérica.

2.4.3 Retroatividade não-consistente

O conceito de retroatividade não-consistente foi apresentado por Acar *et al.* [53] e consiste em descobrir após uma certa inserção ou remoção de operação no passado, qual será a próxima operação na linha temporal da estrutura que se tornará inconsistente. Apesar da diferença de abordagem dos modelos, o método *rollback* descrito por Demaine *et. al* [13] também é aplicável na transformação de uma estrutura em sua versão *não-consistente*.

Lema 2. *Dada uma estrutura na qual cada operação consome tempo $T(n)$ no pior caso, o método* rollback *permite uma estrutura ter seu estado retroativo não-consistente consumindo tempo $O(rT(n))$, onde r é o número de operações realizadas após a modificação [13].*

O método de *rollback* claramente não é o mais eficiente em casos gerais para a modificação de uma estrutura para sua versão persistente. Entretanto, existem ED's nas quais o método de *rollback* é o mais eficiente possível. Por exemplo, suponha uma ED com um contador X e suporta a operação $incrementa()$, que adiciona X com um certo valor e retorna seu valor logo após a adição. Inicialmente, realizam-se m operações do tipo $incrementa()$. Após a realização dessas operações, volta-se no tempo e realiza-se uma operação no início de todas as outras. Ou seja, todas as operações $incrementa()$ posteriores à última realizada terão que ser refeitas e reimpressas. Por esse motivo, não existe para essa estrutura um método mais eficiente nesse caso.

Acar *et. al* [53] apresentaram as seguintes complexidades para estruturas com retroatividade não consistente (Tabela 2.1):

Estrutura de dados	Tempo	Espaço
Dicionário	$O(\lg(\lg(m)))$	$O(n)$
Fila	$O(\lg(\lg(m)))$	$O(n)$
Pilha	$O(\lg(n)\lg(\lg(n)))$	$O(n)$
Fila de Prioridade	$O(\lg(n))$	$O(n)$

Tabela 2.1 Complexidade alcançada por Acar *et al.* [53] para implementação da retroatividade não-consistente. Fonte: Acar *et al.* [53]

Na Tabela 2.1, a variável n consiste no número de operações realizadas até o momento e m é o número de diferentes pontos temporais existentes. As complexidades propostas por Acar *et al.* [53] só podem ser obtidas pois, nesse tipo de estrutura, somente é necessário reportar a primeira operação inconsistente ao usuário, não sendo necessária a posterior correção da estrutura até o tempo atual.

Tipo	Ação	Modificação	Consulta
Retroatividade Parcial	Corrige toda a estrutura após uma modificação	Em qualquer versão da estrutura	Somente na versão mais recente da estrutura
Retroatividade Total	Corrige toda a estrutura após uma modificação	Em qualquer versão da estrutura	Em qualquer versão da estrutura
Retroatividade Não-consistente	Retorna a próxima operação inconsistente sem correção da estrutura	Em qualquer versão da estrutura	Em qualquer versão da estrutura

Tabela 2.2 Síntese dos tipos de estruturas de dados retroativas. Fonte: O Autor.

A Tabela 2.2 contém a síntese dos conceitos apresentados sobre os tipos existentes de retroatividade. Duas dessas versões alteram a estrutura atual após a modificação, enquanto o tipo não-consistente de retroatividade somente retorna a próxima operação não consistente após uma modificação.

2.5 Trabalhos relacionados

Demaine *et al.* [13] e Acar *et al.* [53] foram os primeiros a definirem as ideias sobre retroatividade e as possíveis maneiras de transformar uma estrutura comum em sua versão retroativa. Demaine *et al.* [13] propuseram transformar qualquer estrutura parcialmente retroativa em sua versão totalmente retroativa com um aumento multiplicativo de $\sqrt{m}$ na complexidade e na memória, em que m consiste no tamanho da linha temporal que a estrutura está contida. Essa pesquisa também mostra como obter versões retroativas de filas, pilhas e filas de prioridade *(min-heap)*.

Demaine *et al.* [13] também explicam como gerar uma fila de prioridade totalmente retroativa com um acréscimo multiplicativo na complexidade temporal e espacial de $\sqrt{m}$ por meio de várias filas de prioridade parcialmente retroativas. Acar *et al.* [53] propuseram

uma outra classe de problemas relacionados à retroatividade, que é a noção de retroatividade não-consistente. Nessa classe de problemas, o que se deseja é, a partir da sequência de operações realizadas durante toda a estrutura, descobrir, após uma modificação, o primeiro instante de tempo após a modificação em que a estrutura se tornou inconsistente. Essa noção é importante para resolver problemas em que uma modificação no passado gera algum tipo de efeito cascata na estrutura, e as mudanças geradas por esse efeito afetam outras estruturas na resolução do problema em que a estrutura está sendo aplicada.

Dickerson *et al.* [16] utilizaram estruturas de dados retroativas para uma aplicação geométrica. Nesse estudo, Dickerson *et al.* propuseram um algoritmo em que é possível clonar Diagramas de Voronoi utilizando estruturas de dados retroativas. Esse algoritmo é uma ε-aproximação do clone de um diagrama de Voronoi, permitindo a clonagem desse diagrama em tempo $O(n\lg(\frac{1}{\varepsilon}))$.

Demaine *et al.* [14] propuseram um decréscimo na complexidade temporal e espacial para a versão totalmente retroativa de filas de prioridade. Foi provado que é possível gerar uma fila de prioridade totalmente retroativa em tempo polilogarítmico $O(\lg^2(n))$ por atualização e $O(\lg^2(m)\lg(\lg(n)))$ por consulta, tendo um acréscimo espacial de $O(m\,lg(m))$.

Garg *et al.* [25] e Sunita *et al.* [52] tratam de uma aplicação da retroatividade não-consistente para a dinamização de grafos em problemas de obtenção de caminho mínimo. Pode-se observar que, nesse caso, a alteração de uma aresta no grafo gera um efeito cascata na árvore de caminhos gerada pelo algoritmo de Dijkstra, afetando o vetor de distâncias mínimas a partir de um ponto. Portanto, é necessário a utilização da versão não-retroativa da estrutura, pois, dessa forma, é possível atualizar o vetor de distâncias de uma maneira mais otimizada que simplesmente percorrer todo o grafo novamente. Porém, no pior caso, o algoritmo mais otimizado ainda percorrerá todos os vértices desse grafo (no caso de uma aresta afetar todos os caminhos mínimos de todos os vértices).

Chen *et al.* [9] propõem uma otimização na complexidade temporal com relação à transformação de estruturas parcialmente retroativas em suas versões totalmente retroativas. No artigo de Demaine *et al.* [13], foi provado que, se é gasto complexidade temporal de $T(n,m)$ em uma estrutura parcialmente retroativa, é possível transformá-la em sua versão totalmente retroativa com complexidade temporal de $\mathrm{O}(T(n,m)\sqrt{m})$, em que n é o tamanho da estrutura e m é o número de operações em sua linha temporal. Já no estudo apresentado por Chen *et al.* [9], foi provado que é possível reduzir a diferença de complexidade entre as retroatividades parcial e total em $O(\min\{\sqrt{m}, n\lg m^{1-o(1)}\})$.

Henziger e Wu [32] apresentaram limites inferiores e superiores para versões totalmente retroativas de algoritmos em grafos, como conectividade e manutenção do vértice de maior grau em um grafo dinâmico. Eles também demonstraram que é possível definir, através da

conjectura OMv (proposta por Henzinger *et al.* [31]), que não existe estrutura totalmente retroativa que mantenha a conectividade de grafos dinâmicos, ou uma estrutura de dados retroativa incremental que mantenha o grau máximo de um vértice de um grafo que consuma menos que $O(n^{1-\varepsilon})$ para qualquer $\varepsilon > 1$. No artigo proposto por Henziger e Wu também estende-se a noção de retroatividade para problemas em grafos. Os problemas apresentados e as soluções anteriormente propostas para grafos dinâmicos são considerados versões *parcialmente* retroativas de suas versões em grafos estáticos, uma vez que modificações podem ser realizadas em versões temporais anteriores do grafo. No entanto, somente o estado mais recente da estrutura em questão pode ser acessada.

Nessa dissertação, é apresentada uma solução para o problema da árvore geradora mínima incremental *totalmente* retroativa, em que todas as versões temporais de um grafo estão disponíveis para a consulta, além de permitir adições de arestas em qualquer uma dessas versões.

Capítulo 3

Implementação retroativa das estruturas

Neste capítulo implementa-se a retroatividade em suas formas não-consistente, parcial e total para as estruturas fila, pilha, *union-find* e fila de prioridade.

3.1 Fila

Uma fila é um tipo abstrato de dados que implementa a política FIFO *(first in, first out)*, na qual o primeiro elemento a ser inserido na estrutura é o primeiro elemento a ser removido [61, 12]. Filas geralmente são utilizadas para modelagem de problemas reais, bem como na implementação de algoritmos em que os objetos do problema precisam esperar, e a ordem de chegada desses elementos é o que importa para seu processamento. Por exemplo, o algoritmo de busca em largura (BFS) utiliza uma fila para o processamento dos vértices [12]. É possível, por exemplo, utilizar uma fila retroativa para promover a dinamização desse algoritmo.

As operações implementadas em uma fila são:

- *Enqueue(x):* insere na estrutura o elemento x;
- *Dequeue():* remove o elemento mais antigo da estrutura;
- *Front():* consulta o próximo elemento a ser removido;
- *Back():* consulta o último elemento inserido.

As operações de *Enqueue(x)* e *Dequeue()* são operações de atualização, enquanto as operações *Front()* e *Back()* são operações de consulta.

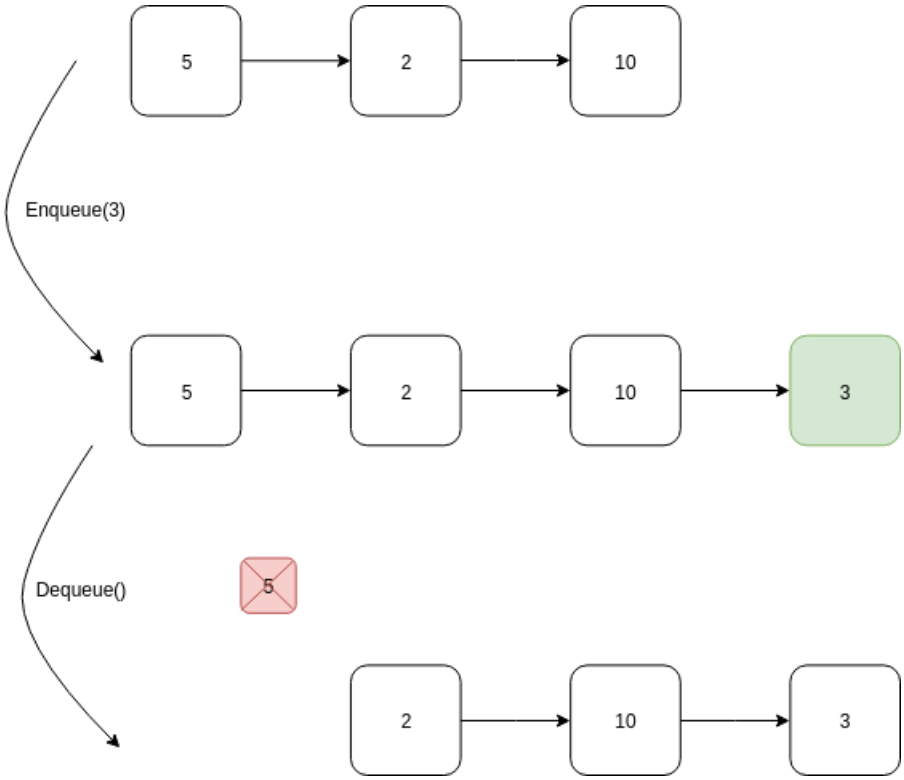

Figura 3.1 Exemplo de uma fila. Fonte: O autor.

Na Figura 3.1 tem-se a visualização da estrutura fila. Os elementos estão organizados de maneira que o elemento adicionado mais recentemente está mais à direita na imagem. É possível observar a mudança na fila após uma operação de $Enqueue(3)$, em que o elemento é colocado no fim da fila, e também a execução de uma operação $Dequeue()$, que no estado atual da estrutura remove o elemento com o valor 5.

As operações retroativas de uma fila podem ser escritas da seguinte forma:

- *Insert(t, Enqueue(x)):* insere na estrutura o elemento x no tempo t;
- *Insert(t, Dequeue()):* remove o elemento mais antigo que esteja na estrutura inserido até o tempo t;
- *Delete(t, Enqueue(x)):* remove a operação *Enqueue* realizada na estrutura no tempo t. Para a execução desta operação, a estrutura deve conter uma operação de inserção da função no tempo correspondente;
- *Delete(t, Dequeue()):* remove a operação *Dequeue* realizada na estrutura no tempo t. Para a execução desta operação, a estrutura deve conter uma operação de *Dequeue* no tempo correspondente;

- *Front(t):* obtém o próximo elemento a ser removido na fila no tempo t;
- *Back(t):* obtém o último elemento inserido na fila no tempo t;

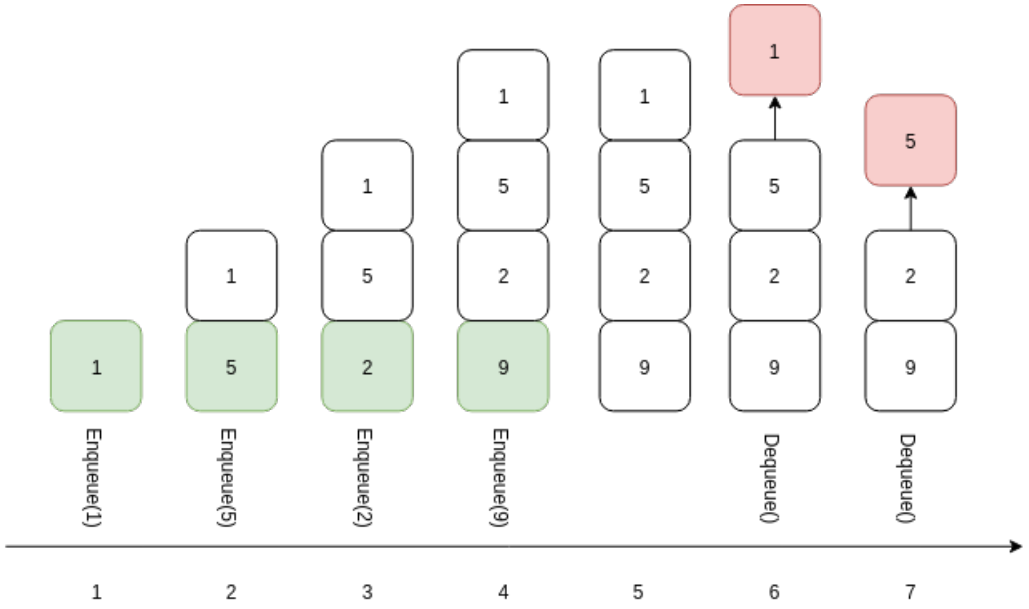

Figura 3.2 Exemplo de fila retroativa. Fonte: O autor

Na Figura 3.2, tem-se o exemplo das operações em uma fila retroativa sobre uma linha temporal de tamanho $m = 7$. Os elementos mais recentes são inseridos no início da estrutura, enquanto os itens removidos são tirados do fim da estrutura.

Já na Figura 3.3, é possível observar o efeito da adição de uma operação na linha temporal da estrutura fila. A partir da modificação realizada, tem-se uma mudança da estrutura com relação aos próximos elementos a serem removidos. Na Figura 3.2, as operações *Front* realizadas em todos os momentos da estrutura retornariam os elementos $\{1,1,1,1,1,5,2\}$, enquanto na Figura 3.3 a estrutura retornaria os itens $\{1,1,1,1,5,2,9\}$.

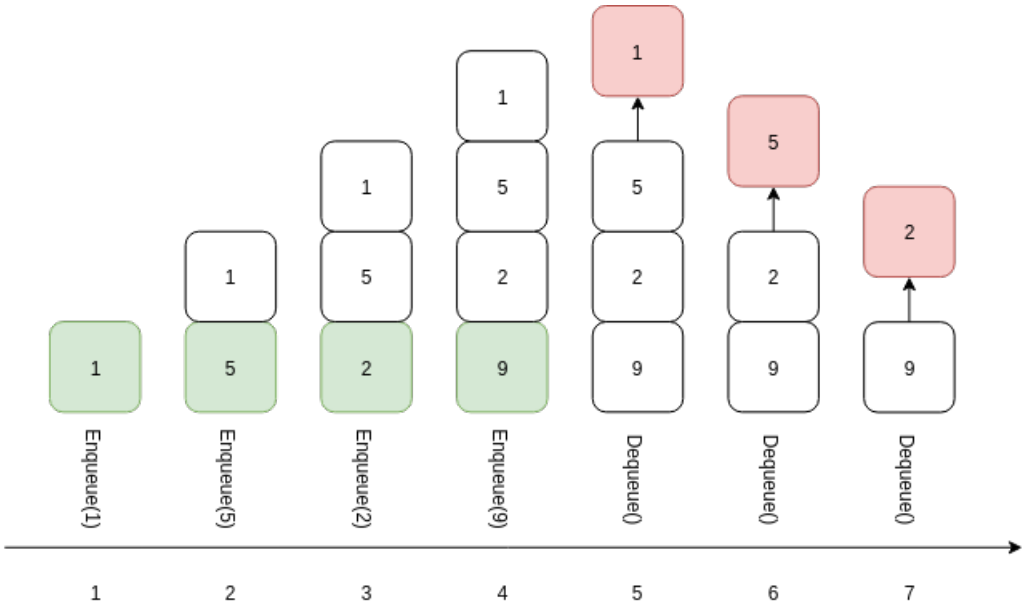

Figura 3.3 Exemplo de fila retroativa após a inserção de uma operação de uma operação *Dequeue* no tempo 5. Fonte: O autor

As implementações da estrutura retroativa fila, na linguagem C++, como as descritas a seguir, podem ser encontradas em https://github.com/juniorandrade1/Master/tree/master/src/Queue.

3.1.1 Retroatividade parcial da fila

Na retroatividade parcial aplicada a uma fila, as operações de consulta *Front* e *Back* são sempre realizadas no estado mais recente da estrutura. Portanto, para a fila parcialmente retroativa, o argumento temporal da função será omitido nessas operações. Demaine *et al.* [13] propuseram um algoritmo para a implementação de filas parcialmente retroativas.

Lema 3. *É possível obter uma fila parcialmente retroativa que consome tempo* $O(1)$ *nas consultas e* $O(\lg n)$ *nas atualizações [13].*

Demonstração. Seja E uma estrutura de dados que mantém as operações *Insert(t, Enqueue(x))* ordenadas por tempo e dois ponteiros: B o ponteiro para o último elemento inserido na sequência e F o ponteiro para o próximo elemento a ser removido. Quando uma operação *Insert(t, Enqueue(x))* é retroativamente inserida, ela é adicionada em E. Se ela ocorre em um tempo posterior à operação apontada por F, então basta mover F para seu predecessor. Quando uma operação *Delete(t, Enqueue(x))* é realizada, esta é removida de E. Além disso, se essa operação ocorrer em um tempo anterior a operação apontada por F, é

necessário mover esse ponteiro para seu sucessor. Nesse caso, quando uma operação *Insert(t, Dequeue())* é realizada, basta avançar F para seu sucessor, enquanto a operação *Delete(t, Dequeue())* retrocede F para seu predecessor.

Já o ponteiro B somente é atualizado quando ocorre uma operação no fim da estrutura. Ou seja, B é movido para seu sucessor ao se realizar a operação *Insert(t, Enqueue(x))*, sendo que t é o tempo mais recente na estrutura. Além disso, B é movido para seu predecessor ao se realizar a operação *Delete(t, Enqueue(x))* no maior tempo na estrutura. □

A estrutura que é capaz de implementar inserções e remoções em tempo logarítmico no número elementos é a árvore binária balanceada de busca (*BBST*), como, por exemplo, *Red-Black Trees* [28], *Splay-Trees* [50] ou *Treaps* [3]. As operações de consulta podem ser realizadas por meio da consulta dos valores nos ponteiros B e F nessa árvore. A atualização dos ponteiros B e F através da inserção e remoção de operações tem custo temporal amortizado $O(1)$, enquanto as inserções consomem, no pior caso, $O(\lg n)$.

No Algoritmo 1, tem-se a implementação das funções parcialmente retroativas de uma fila. As variáveis têm os mesmos nomes utilizados na definição das operações do Lema 3. Ou seja, a variável E é árvore binária de busca balanceada (*Treap*), enquanto B e F são dois ponteiros que apontam os nós da árvore que contém os elementos inicial e final da fila, respectivamente.

Inicialmente, a estrutura está vazia. Após a inserção do primeiro elemento na fila, é necessária a atualização dos ponteiros B e F, operação realizada com base na condição da linha 9. Caso contrário, o ponteiro referente ao início da fila é atualizado sempre que uma operação é realizada anteriormente ao nó apontado por F. Em caso afirmativo, move-se F para seu predecessor (linha 11). A variável B sempre será o maior elemento de E, e portanto, B pode ser atualizado somente atribuindo a B o ponteiro relativo à última posição da árvore (linha 13).

Na deleção de uma operação de $Enqueue(x)$, é necessária a atualização de F se essa operação está sendo realizada em um tempo anterior ao do nó apontado por F. Então, move-se o ponteiro F para seu sucessor (linha 23). A seguir, se o elemento apontado por F é o elemento a ser removido, é necessária a atualização de F, caso contrário, após a remoção da operação $Enqueue(x)$ realizada no tempo t, perderia-se o nó apontado por F, o que tornaria a estrutura inconsistente. Portanto, atualiza-se F para seu sucessor (linha 26).

Algoritmo 1 Operações em uma fila parcialmente retroativa

```
 1: BST::Treap E
 2: BST::Treap::Iterator B, F
 3:
 4: função INSERTENQUEUE(t, x)
 5:     E.insert(t, x)
 6:     se E.size() == 1 então
 7:         B = E.begin()
 8:         F = E.begin()
 9:     senão
10:         se F.first > t então
11:             F = F.prev()
12:         fim se
13:         B = E.end()
14:     fim se
15: fim função
16:
17: função INSERTDEQUEUE(t)
18:     F = F.next()
19: fim função
20:
21: função DELETEENQUEUE(t)
22:     se F.first >= t então
23:         F = F.next()
24:     fim se
25:     E.erase(t)
26:     F = F.next()
27: fim função
28:
29: função DELETEDEQUEUE(t)
30:     F = F.prev()
31: fim função
32:
```

3.1.2 Retroatividade total da fila

Demaine *et al.* [13] também propuseram uma solução com complexidade logarítmica por operação para a implementação de uma fila totalmente retroativa.

Lema 4. *Existe uma estrutura totalmente retroativa para a implementação de uma fila que consome tempo* $O(\lg n)$.

Demonstração. É possível manter duas árvores binárias de busca, armazenando os tempos de inserção e remoção dos elementos em uma fila. Seja T_e a árvore que suporta a inserção dos elementos dessa fila ordenados por tempo de inserção, e T_d uma árvore contendo os tempos de remoção dessa fila. As operações de adição de uma operação de inserção ou remoção são realizadas simplesmente inserindo os elementos nas respectivas árvores. Suponha, sem perda de generalidade, que as operações sempre serão consistentes, ou seja, para que uma operação de $Dequeue(t)$ seja realizada, sempre haverá pelo menos um elemento na fila correspondente no tempo t. Para obter-se o próximo elemento a ser removido da fila no tempo t, basta consultar quantos elementos foram removidos até o tempo t através de T_d. Seja k o número de elementos removidos da fila até o tempo t, então o elemento que será removido da fila na próxima iteração é o k-ésimo elemento da árvore T_e. □

Como pode ser visto no Algoritmo 2, as operações para a implementação de uma fila completamente retroativa se baseiam em uma árvore binária de busca balanceada aleatoriamente. Essa estrutura deve ser capaz de suportar, além das operações triviais como inserção e deleção de nós, funções dos seguintes tipos:

- Dada uma chave t, encontrar o número de elementos com chaves menores ou iguais a t;
- Dado um inteiro k, encontrar qual o k-ésimo elemento do encaminhamento em ordem dessa árvore.

Essas informações podem ser armazenadas em uma árvore binária mantendo-se, para cada nó da árvore, um inteiro que representa o tamanho da sua sub-árvore. A primeira operação pode ser feita percorrendo a árvore e analisando o valor da chave de um nó com relação ao valor procurado. Se o valor do nó atual for menor ou igual a t, basta percorrer recursivamente através do filho direito desse nó na árvore, somando o tamanho da sub-árvore esquerda.

Todo elemento de uma sub-árvore esquerda de um nó é menor que t, e, portanto, deve ser contado na resposta. Se t for maior que o valor do nó, basta fazer o encaminhamento para a

esquerda desse nó. De maneira similar, a segunda operação pode ser implementada em uma árvore binária de busca. Com isso, tem-se a implementação de uma fila totalmente retroativa em tempo logarítmico no número de elementos da árvore por operação.

Algoritmo 2 Operações em uma fila totalmente retroativa

```
1:  BST::Treap T_e, T_d
2:
3:  função INSERTENQUEUE(t, x)
4:      T_e.insert(t, x)
5:  fim função
6:
7:  função INSERTDEQUEUE(t)
8:      T_d.insert(t, 1)
9:  fim função
10:
11: função DELETEENQUEUE(t)
12:     T_e.erase(t)
13: fim função
14:
15: função DELETEDEQUEUE(t)
16:     T_d.erase(t)
17: fim função
18:
19: função GETKTH(t, k)
20:     f ← T_d.orderOfKey(t) + T_d.find(t)
21:     retorna T_e.findByOrder(k + f)
22: fim função
23:
24: função FRONT(t)
25:     retorna getKth(t, 1)
26: fim função
```

3.1.3 Retroatividade não-consistente da fila

Na implementação da fila retroativa não-consistente, mantém-se novamente dois conjuntos ordenados, representando as operações de inserção (T_e) e de remoção (T_d) na estrutura. Porém, é necessário manter mais informações em cada elemento desses conjuntos. No

conjunto T_e são mantidos o tempo de inserção, o valor do elemento e o tempo de remoção correspondente a esse valor inserido (sendo nulo quando não removido). Já em T_d, mantém-se as remoções ordenadas por tempo, além do tempo do elemento removido pela inserção dessa operação. É considerado aqui que a estrutura sempre será consistente, de modo que toda operação *Dequeue* realizada será executada em um estado não vazio da fila.

Assim, para uma operação $Insert(t, Enqueue(x))$, basta inserir $(t, x, null)$ em T_e e define-se a próxima operação inconsistente como a próxima operação de *Dequeue* após o tempo t. Já para a operação $Insert(t, Dequeue())$, é inserido $(t, Front(t))$ em T_d e a operação inconsistente a ser reportada é a primeira operação de *Dequeue* após t.

Realizar uma operação de remoção corresponde a deletar um elemento de T_e ou T_d. Quando essa atualização acontece, é necessário atualizar o par correspondente. Realizar uma operação $Delete(t, Enqueue(x))$ remove esse elemento de T_e, além de retornar como inconsistente a operação no conjunto T_d que corresponde à deleção realizada nesse elemento. De maneira análoga, a operação $Delete(t, Dequeue())$ remove de T_d o elemento correspondente a essa operação, atualiza o elemento do conjunto T_e que corresponde à deleção realizada por essa operação, e retorna como inconsistente a próxima operação do conjunto T_d após o tempo t.

Esses retornos de operações inconsistentes são realizados eficientemente, uma vez que os elementos dos conjuntos T_e e T_d estão conectados entre si. Esses conjuntos podem ser implementados por meio de árvores binárias de busca.

Algoritmo 3 Operação de inserção da operação $Enqueue(x)$ no tempo t em uma fila com retroatividade não-consistente

```
1: função INSERTENQUEUE(t, x)
2:     T_e.insert(t, {x, NULL})
3:     Treap :: iterator nextGEQThanT ← T_d.lowerBound(t)
4:     se nextGEQThanT == NULL então
5:         retorna NULL
6:     senão
7:         retorna (*nextGEQThanT).second
8:     fim se
9: fim função
```

No Algoritmo 3 tem-se a implementação da função de inserção da operação $Enqueue(x)$ no tempo t. Essa função retorna o tempo da próxima operação *Dequeue* inconsistente. Inicialmente, a função insere o elemento x no conjunto T_e no tempo t. Após a inserção, é realizada uma consulta no conjunto T_d para a obtenção da primeira operação *Dequeue*

realizada após o tempo t (linha 3). A função *lower_bound(t)* na estrutura de dados *Treap* retorna um ponteiro para o primeiro elemento dessa estrutura que contém uma chave maior ou igual ao valor t, retornando NULL caso não existam elementos com chaves maiores ou iguais a t.

No Algoritmo 4 tem-se a implementação da inserção da operação $Dequeue()$ no tempo t em uma fila com retroatividade não-consistente. Inicialmente, encontra-se o objeto correspondente ao próximo elemento a ser removido no tempo t, utilizando a mesma abordagem apresentada para a obtenção desse elemento na fila totalmente retroativa. A função $Front$ retorna o nó do conjunto T_e, que representa o próximo elemento a ser removido. Essa função retorna um ponteiro (*iterator*), que, ao ser instanciado, retorna um par correspondente à chave e o valor do nó retornado pela função. Esse elemento pode estar ou não relacionado a uma operação de *Dequeue*. Caso esse elemento esteja relacionado, é necessário remover a relação entre essas operações. Portanto, é obtida a operação *Dequeue* relacionada ao elemento $front$, e esse relacionamento é removido. A seguir, é atualizado o elemento a ser retirado da fila pela operação $Dequeue(t)$ e inserido no conjunto T_d a operação realizada. Finalmente, é retornada a próxima operação *Dequeue* após o tempo t como inconsistente.

Algoritmo 4 Função para inserção da operação $Dequeue()$ no tempo t em uma fila com retroatividade não-consistente

1: **função** INSERTDEQUEUE(t)
2: $Treap :: iterator\ frontIt \leftarrow Front(t)$
3: $pair < int, pair < T, int >> front \leftarrow *frontIt$;
4: **se** front.second.second != NULLVALUE **então**
5: $BST :: Treap :: iterator\ elementPointedByTIt \leftarrow T_d.lowerBound(front.second.second)$
6: $pair < int, int > elementPointedByT \leftarrow *elementPointedByTIt$
7: $elementPointedByT.second \leftarrow NULLVALUE$
8: $elementPointedByTIt.modify(elementPointedByT.first, elementPointedByT.second)$
9: **fim se**
10: $front.second.second \leftarrow t$
11: $frontIt.insert(t, front.first)$
12: $BST :: Treap :: iterator\ nextInconsistenceIt \leftarrow ++td.lowerBound(t)$
13: **se** $nextInconsistenceIt == NULL$ **então**
14: **Retorna** $NULLVALUE$;
15: **senão**
16: **Retorna** $(*nextInconsistenceIt).first$;
17: **fim se**
18: **fim função**

Algoritmo 5 Operação de deleção da operação $Enqueue(x)$ no tempo t em uma fila com retroatividade não-consistente

1: **função** DELETEENQUEUE(t)
2: $BST :: Treap :: iterator\ enqueueTIt \leftarrow T_e.lowerBound(t)$
3: $pair < int, pair < T, int >> enqueue \leftarrow *enqueueTIt$;
4: $T_e.erase(enqueue.first)$
5: **Retorna** $enqueue.second.second$
6: **fim função**

No Algoritmo 5 tem-se a implementação da operação de deleção da operação $Enqueue(x)$ no tempo t. Nessa operação, remove-se o elemento do conjunto T_e e retorna-se a operação relacionada a esse elemento como a primeira operação *Dequeue* inconsistente.

Algoritmo 6 Operação de deleção da operação $Dequeue()$ no tempo t em uma fila com retroatividade não-consistente

```
1:  função DELETEDEQUEUE(t)
2:                                          ▷ Encontra e deleta a operacao no tempo t
3:      BST :: Treap :: iterator deleteTIt ← T_d.lowerBound(t)
4:      pair < int, int > deleteT ← *deleteTIt
5:      T_d.erase(deleteT.first)
6:
7:                  ▷ Encontra e modifica a operacao Enqueue apontada pela Dequeue(t)
8:      BST :: Treap :: iterator enqueueTIt ← T_e.lowerBound(deleteT.second)
9:      pair < int, pair < T, int >> enqueueT ← *enqueueTIt
10:     enqueueT.second.second ← NULLVALUE
11:     enqueueTIt.modify(enqueueT.first, enqueueT.second)
12:
13:                                  ▷ Encontra a proxima operacao inconsistente
14:     BST :: Treap :: iterator nextInconsistenceIt ← T_d.lowerBound(t)
15:     se nextInconsistenceIt == NULL então
16:         Retorna NULLVALUE
17:     senão
18:         Retorna (*nextInconsistenceIt).first
19:     fim se
20: fim função
```

No Algoritmo 6 é realizada a deleção de uma operação $Dequeue()$ no tempo t em uma fila com retroatividade não-consistente. Remove-se a operação *Dequeue* no tempo t do conjunto de operações T_d. A seguir, é necessária a atualização do elemento do conjunto T_e removido da fila pela operação *Dequeue* que está sendo deletada. Então, é retornada como inconsistente a próxima operação *Dequeue* no conjunto T_d.

É importante reiterar que todas as execuções das operações anteriormente definidas somente retornam a próxima operação *Dequeue* que se torna inconsistente após uma modificação. Como os conjuntos T_e e T_d são implementados sobre árvores binárias de busca auto-balanceáveis, as operações são realizadas em tempo logarítmico no número de elementos desses conjuntos. Para obter a estrutura consistente, deve-se realizar, recursivamente, as atualizações por toda a estrutura, que, no pior caso, acessa todos os elementos dos dois conjuntos.

3.2 Pilha

Uma pilha é um tipo abstrato de dados, introduzida em 1946 por Alan M. Turing, que implementa a política chamada LIFO *(last in, first out)*, em que o último elemento a ser inserido na estrutura é o primeiro elemento a sair dela [61, 12]. Pilhas possuem aplicações em áreas intrínsecas à computação, como no tratamento do fluxo de execução de chamadas recursivas, e na aplicação de algoritmos como a busca em profundidade *(dfs)* em um grafo [12].

Comumente, as operações que esta estrutura de dados suporta são:

- *Push(x):* insere na estrutura o elemento x;
- *Pop():* remove o elemento mais recentemente adicionado na pilha;
- *Peak():* obtém o elemento mais recentemente adicionado na pilha.

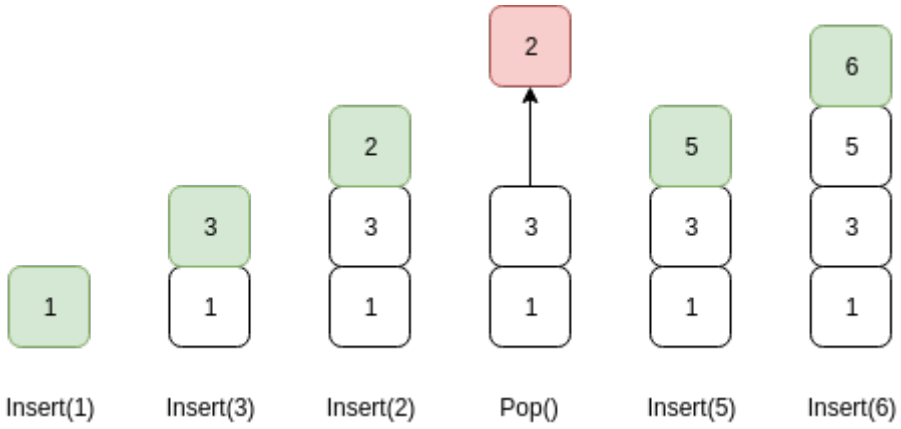

Figura 3.4 Exemplo de uma pilha. Fonte: O Autor.

Na Figura 3.4, tem-se a a representação de uma pilha comum. Pode-se observar que, nessa estrutura, os elementos são inseridos e removidos somente a partir de um de seus extremos.

Pela natureza das operações, em sua versão não retroativa, estas podem ser realizadas em tempo constante para cada inserção/remoção. Na versão retroativa da estrutura, pode-se definir as seguintes operações:

- *Insert(t, Push(x)):* insere na estrutura o elemento x no tempo t;
- *Insert(t, Pop()):* insere na estrutura a operação de remoção do elemento mais recentemente adicionado na pilha no tempo t;

- *Remove(t, Push(x)):* remove da estrutura o evento *Push* realizado no tempo t. Esse evento precisa estar presente na estrutura para que a operação seja realizada;
- *Remove(t, Pop()):* remove da estrutura o evento *Pop* realizado na estrutura no tempo t. Esse evento precisa estar presente na estrutura para que a operação seja realizada;
- *Peak(t):* obtém o elemento mais recentemente adicionado na pilha no tempo t.

Observe que para a retroatividade parcial, a variável t na operação de consulta *Peak(t)* sempre será o valor de tempo que representa o estado atual da estrutura.

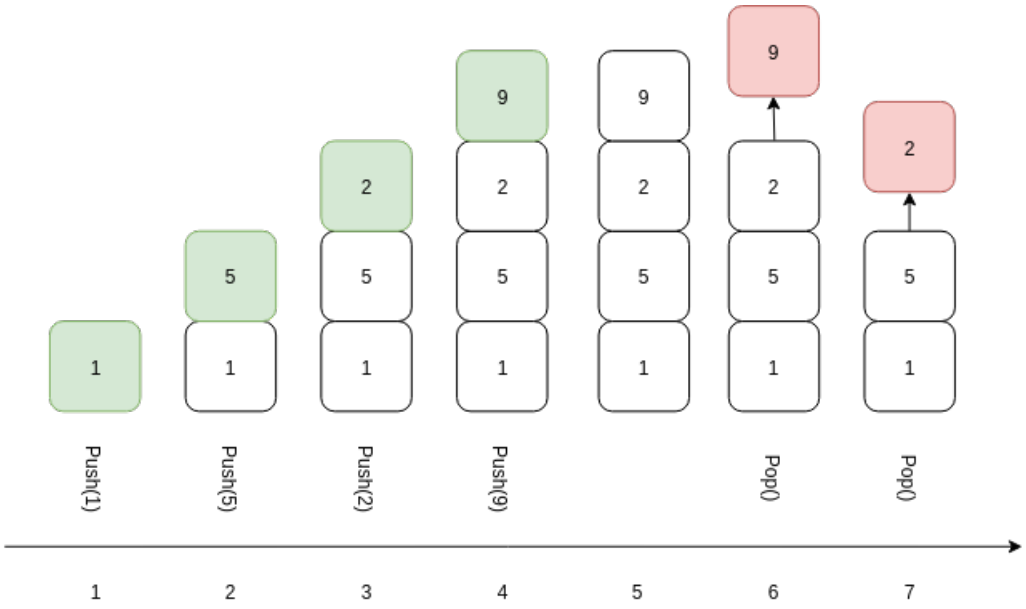

Figura 3.5 Exemplo de pilha retroativa. Fonte: O autor

Na Figura 3.5 tem-se uma pilha retroativa implementada sobre uma linha temporal de tamanho $m = 7$. Os elementos em verde representam itens adicionados, enquanto os itens em vermelho correspondem aos itens removidos. Como em uma pilha, em uma consulta, é possível obter somente um extremo da estrutura. Neste caso, a execução das operações *Peak* em todos os possíveis tempos resultariam no retorno dos elementos $\{1,5,2,9,9,2,5\}$.

Na Figura 3.6, é possível ver o efeito da adição da operação *Pop* no tempo 5 da linha temporal da estrutura. Os eventos anteriores à adição da operação não são afetados nesse caso. Porém, todas as operações posteriores à adição da operação afetam elementos diferentes. No exemplo, tanto a operação realizada no tempo 6 quanto a operação realizada no tempo 7 realizam a retirada de elementos distintos com relação à sua execução na Figura 3.5. Além disso, a execução da operação *Peak* em todos os possíveis tempos da linha do tempo retornaria os elementos $\{1,5,2,9,2,5,1\}$.

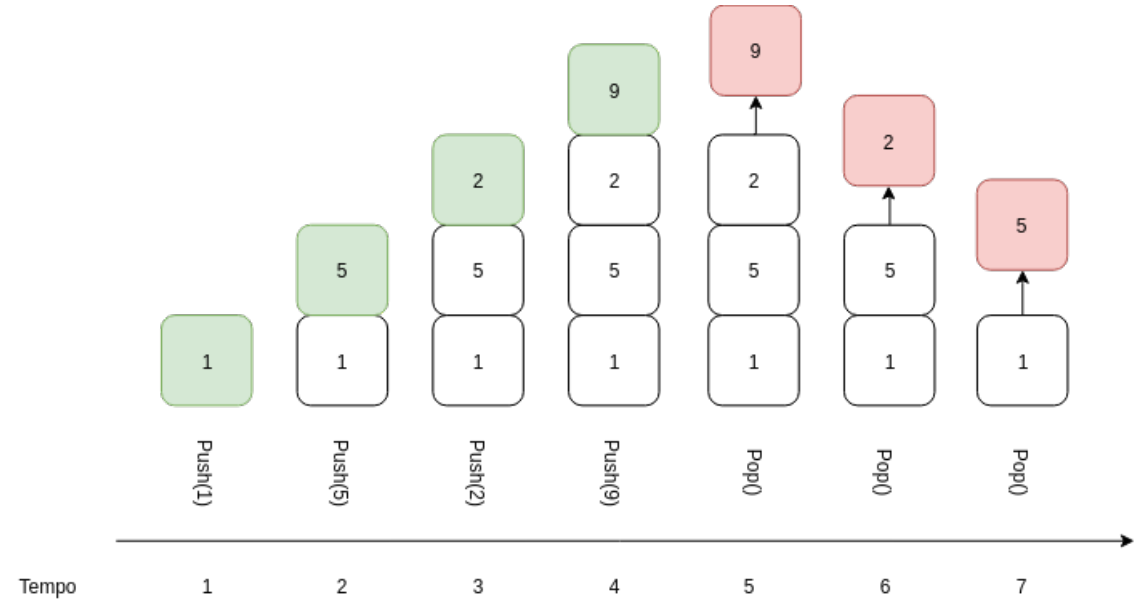

Figura 3.6 Exemplo de pilha retroativa após a adição de uma operação *Push* no tempo 5. Fonte: O autor

Foram realizadas implementações em C++ da estrutura retroativa pilha como as descritas a seguir, que podem ser encontradas em https://github.com/juniorandrade1/Master/tree/master/src/Stack.

3.2.1 Retroatividade parcial da pilha

A retroatividade parcial de uma pilha se torna um pouco mais complexa que a retroatividade parcial em uma fila, uma vez que as remoções e consultas são realizadas no mesmo extremo da estrutura. Entretanto, é possível obter uma estrutura que realize as operações retroativas em tempo logarítmico, por operação, no número de elementos da pilha.

Lema 5. *Existe uma solução para a implementação da retroatividade parcial de uma pilha que consome tempo* $O(\lg n)$ *por operação [13].*

Demonstração. Uma possível implementação para uma pilha em um vetor V consiste em utilizar uma variável P para representar o topo da pilha. Ou seja, inicialmente, $P = 0$ e a cada operação $Push(x)$, atualiza-se o valor de $V[P] = x$ e incrementa-se P em uma unidade. Já para as operações $Pop()$, basta decrementar P.

Na implementação retroativa dessa estrutura, pode-se manter as operações $Push(x)$ e $Pop()$ em uma lista ordenada pelo tempo da operação, associando um peso +1 para as operações do tipo $Push(x)$ e -1 para operações do tipo $Pop()$. Assim, o valor de P no tempo t

consiste na soma do prefixo dos pesos até o tempo t. Essa soma de prefixo pode ser mantida em uma árvore binária em que os elementos dessa lista são as folhas dessa estrutura.

Para encontrar o valor de $V[P]$ no tempo atual, basta encontrar o último tempo $t' \leq t$, tal que $V[P]$ tenha sido atualizado. □

A árvore utilizada para a implementação da estrutura que mantém a soma de prefixos em tempo logarítmico por operação é chamada árvore de segmentos. Nessa estrutura, os elementos de uma lista ligada são associados aos nós folha dessa árvore, e cada nó contém informações sobre um segmento dessa lista.

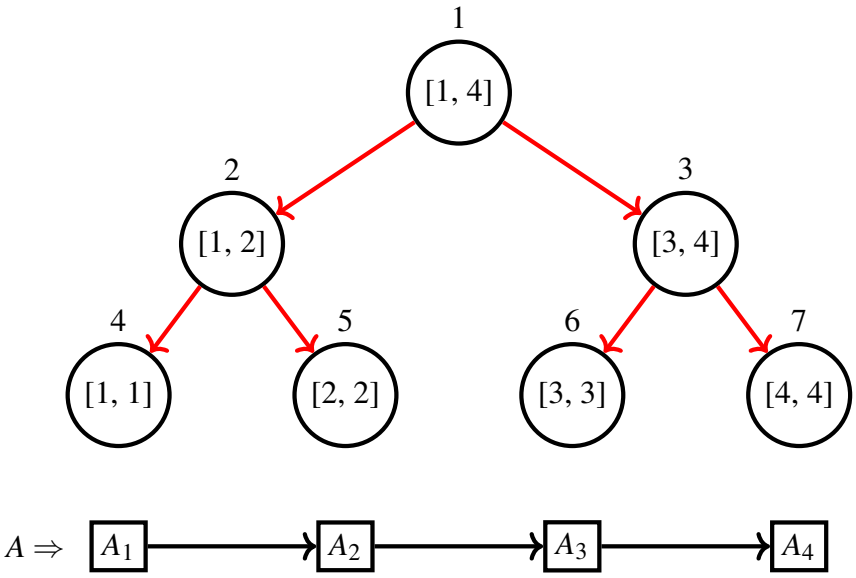

Figura 3.7 Exemplo de árvore de segmentos. Fonte: O autor.

Essa árvore torna possível a obtenção de informações relevantes com relação à lista ligada em operações mais simples, como a soma dos valores de um subintervalo dessa lista ligada, bem como operações mais complexas, como operações de máximo e de mínimo de um subintervalo.

Na Figura 3.7, tem-se um exemplo de uma possível árvore de segmentos implementada sobre uma lista ligada. Na imagem, existe uma lista ligada com quatro elementos, em que cada nó folha da árvore corresponde a um elemento da lista. Já os outros nós correspondem à união da informação sobre os seus nós filhos. Para a implementação desse problema, utilizou-se uma árvore de segmentos na qual cada nó que não seja folha, correspondente ao intervalo [l, r] na lista ligada, contém exatamente dois filhos, correspondendo aos intervalos [l, $\lfloor \frac{l+r}{2} \rfloor$] e [$\lfloor \frac{l+r}{2} \rfloor + 1$, r], respectivamente.

Para o problema da pilha, manter a soma de um prefixo nessa árvore consiste em, para cada nó dessa árvore, manter a soma dos pesos do intervalo que o nó corresponde. Portanto, obter

a soma de prefixo de uma posição i da lista ligada consiste em procurar o nó correspondente a essa posição. As pesquisas são realizadas sempre partindo da raiz. Para encontrar essa posição dentro da árvore, basta recursivamente percorrer a árvore encaminhando para os nós onde i está contido no intervalo. Seja R o nó raiz da subárvore atual na recursão, definido pelo intervalo $[l, r]$. Então, tem-se duas situações:

- Se $i \leq \lfloor \frac{l+r}{2} \rfloor$, então i está contido na sub-árvore esquerda de R, bastando seguir para o nó da esquerda;
- Se $i > \lfloor \frac{l+r}{2} \rfloor$, então i está na subárvore direita de R, ou seja, toda a soma da sub-árvore da esquerda está contida na soma de prefixo de i, então, além de seguir para o nó da direita de R, deve-se somar os valores contidos na sub-árvore esquerda de R.

A complexidade temporal para encontrar a soma de prefixo dinâmica é equivalente à altura da árvore de segmentos correspondente à lista ligada. Como a cada nível da árvore o número de nós cai pela metade, a altura dessa árvore é $\lg n$. Para realizar essa soma de prefixo e não percorrer todos os nós dessa árvore, utiliza-se a técnica de *lazy propagation*. Nessa técnica, sabendo-se que o intervalo de um nó está totalmente contido no intervalo de uma atualização, insere-se uma *flag*[2] para uma posterior atualização daquele nó, e interrompe-se a propagação. Assim, ao existir alguma outra operação que acesse esse nó, a operação o atualizará, e passará essa *flag* para os seus nós filhos.

Além disso, para a criação da versão retroativa da pilha, precisa-se manter a menor e a maior soma de prefixo. Assim, seja P o valor da soma de prefixo da lista ligada, e, consequentemente, a posição na lista ligada do topo da pilha. Uma operação *Peak* pode ser realizada descendo na árvore de segmentos, procurando pelo nó mais à direita em que P está entre o valor mínimo e máximo das somas de prefixo contidas naquele nó.

Algoritmo 7 Estrutura do nó da árvore

```
1: função STRUCT NODE
2:     Inteiro ps, maxIns, minIns
3:     TAD data
4: fim função
```

No Algoritmo 7, tem-se a implementação da estrutura interna de um nó na árvore de intervalos. As variáveis *ps*, *maxIns* e *minIns* representam a soma das somas de prefixo de um dado nó, o máximo e mínimo das somas de prefixo do intervalo, respectivamente. Além

[2] *flag* consiste em uma marcação em um nó para posterior atualização

disso, tem-se a variável *data*, que armazena o valor do objeto que se deseja armazenar na estrutura, sendo T um tipo abstrato de dados.

Algoritmo 8 Implementação da função de propagação

```
1:  função PROPAGATE(no, l, r)
2:      se lz[no] não contém atualizações então
3:          fim função
4:      fim se
5:      tr[no].ps+ = (r − l + 1) ∗ lz[no]
6:      tr[no].minIns+ = lz[no]
7:      tr[no].maxIns+ = lz[no]
8:      se l ≠ r então
9:          nxt ← (no ∗ 2)
10:         mid ← (l + r)/2;
11:         lz[nxt]+ = lz[no]
12:         lz[nxt + 1]+ = lz[no]
13:     fim se
14:     lz[no] = 0
15: fim função
```

No Algoritmo 8, tem-se a implementação da função de propagação em uma árvore de segmentos. Nessa função, são passados como argumentos o nó que está sendo acessado no momento e seu intervalo de abrangência $[l, r]$. Nela, utiliza-se o vetor auxiliar lz como o valor de propagação atrasada nos nós. A variável $lz[no]$ retém o valor que precisa ser adicionado a cada posição no intervalo contido pela nó atual. Então, o valor da variável que corresponde à soma dos prefixos do intervalo aumenta proporcionalmente ao número dos nós folha que essa subárvore contém, que é exatamente o tamanho do intervalo (linha 3). Como cada um dos valores das posições contidas no intervalo aumenta em $lz[no]$, o valor do máximo e mínimo aumentará (ou diminuirá) exatamente em $lz[no]$ (linhas 6 e 7). Posteriormente, é realizada a atualização dos filhos do nó atual, que recebem os valores propagados de seu pai (linhas 9 a 12).

Já no Algoritmo 9, tem-se a implementação de uma das funções que utilizarão a função de propagação. Observe que sempre antes de acessar um nó, seja ele a raiz da subárvore observada ou um dos seus nós filhos, ele deve passar pela função de propagação para não existir o risco de obter informações erradas por falta de propagação. Nessa função, pode-se notar como o encaminhamento é realizado na implementação. A variável *no* corresponde a

um intervalo $[l,r]$ no vetor, e deseja-se atualizar a operação no tempo t para uma inserção com valor $data$.

As inserções das operações $Pop()$ são realizadas similarmente à inserção da operação *Push* com relação à sua implementação na árvore de segmentos. A única diferença diz respeito à atualização da variável lz, que nesse caso é decrementada em uma unidade.

Algoritmo 9 Implementação da inserção da operação *Push* na árvore de segmentos

```
1:  função UPDATE(no, l, r, i, j)
2:      propagate(no, l, r)
3:      se l == r i.e. nó atual é folha então
4:          lz[no] + +
5:          propagate(no, l, r)
6:          tr[no].data ← data
7:          tr[no].minIns ← tr[no].ps
8:          tr[no].maxIns ← tr[no].ps
9:          fim função
10:     fim se
11:     nxt ← (no ∗ 2)
12:     mid ← (l + r)/2
13:     se t ≤ mid então
14:         updateInsert(nxt, l, mid, t, data)
15:         lz[nxt + 1] + +
16:     senão
17:         updateInsert(nxt + 1, mid + 1, r, t, data)
18:     fim se
19:     propagate(nxt, l, mid)
20:     propagate(nxt + 1, mid + 1, r)
21:     tr[no] = tr[nxt] + tr[nxt + 1]
22: fim função
```

Na atualização da árvore de segmentos na operação $Insert(t, Push(data))$, o primeiro estado a ser verificado é se o nó atual é uma folha. Em caso afirmativo, simplesmente atualiza-se a *flag* de propagação e finaliza-se a recursão (linhas 3-9). Caso contrário, ainda é necessária a atualização dos nós filhos do nó atual. Como a atualização do nó está sendo realizada na posição t, é necessário encontrar o nó folha correspondente a essa posição. Logo, se t está contido no intervalo da subárvore esquerda, é necessária a atualização da subárvore esquerda desse nó, e, além disso, se faz inevitável a atualização da subárvore direita desse nó,

pois como se trata de uma soma de prefixo, todos as posições maiores que a posição do nó atual serão afetadas (linhas 13-16).

Com isso, é possível implementar as operações de uma fila parcialmente retroativa em tempo logarítmico no número de operações.

3.2.2 Retroatividade total da pilha

Para a retroatividade total dessa estrutura, também existe uma solução em tempo logarítmico no número de operações.

Lema 6. *Existe uma estrutura totalmente retroativa para a implementação de uma pilha que consome tempo* $O(\lg n)$ *[13].*

Demonstração. A solução para obter-se uma pilha totalmente retroativa em tempo logarítmico no número de operações é similar à implementação descrita para a sua versão parcialmente retroativa. A única diferença diz respeito à obtenção do topo da pilha para um tempo arbitrário t. Para isso, suponha que no tempo t a soma de prefixo seja a variável R. Assim, se A é o vetor que representa a pilha no tempo t, tem-se que o topo da pilha está em $A[R]$. Isso pode ser feito encontrando em A a última operação antes do tempo t, subindo na árvore, e depois descendo de modo a encontrar a subárvore mais à direita, de modo que R esteja entre os valores mínimo e máximo dessa subárvore. □

Em outras palavras, em um primeiro momento, é efetuada a descida a partir da raiz da árvore até o nó folha com tempo t. A partir desse nó, sobe-se na árvore até encontrar um nó no qual o valor do mínimo e máximo da subárvore representada pelo nó seja R e que o último nó visitado seja um filho da direita do nó atual. Essa última verificação é necessária para que se mantenha a propriedade de encontrar um índice menor que t. A seguir, move-se para o filho da esquerda desse nó (garantindo a última propriedade) e, posteriormente, procurando a subárvore mais à direita, que obedece a propriedade do mínimo e máximo conterem R.

Ao subir na árvore a partir do nó folha da posição R, o ponteiro encontra-se em nós que, com certeza, contém R em seu intervalo representado. Ao andar um passo para a esquerda antes de começar a procurar a subárvore mais à direita, garante-se que sempre estará em um nó em que t não está contido, e t é maior que a última posição contida pelo nó.

Na Figura 3.8 tem-se a visualização das informações que serão mantidas em uma árvore de segmentos. Essa árvore representa a adição da operação *Push* nos tempos 1 e 2 (observe que a soma de prefixo aumenta em uma unidade nesses nós). Os valores internos ao nó representam o intervalo que cada nó abrange, enquanto os valores acima de cada nó representam os valores de mínimo e máximo das operações de inserção de todas as subárvores que aquele nó

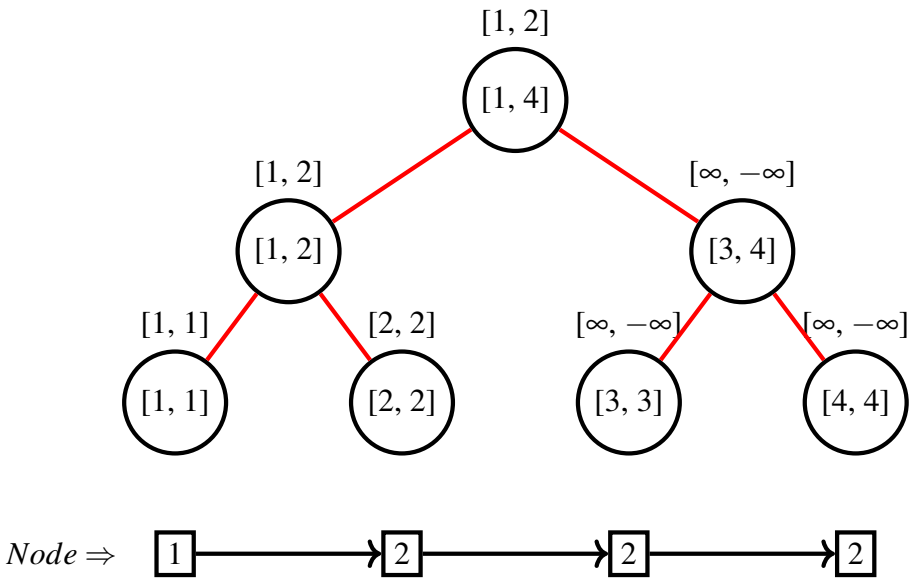

Figura 3.8 Manutenção das informações em cada nó. Fonte: O autor.

contém. Por esse motivo, todos os nós que representam os intervalos entre [3, 4] estão com valores padrão.

Assim, suponha que será realizada a consulta $Peak(3)$. Primeiramente, sabe-se que pelo valor da soma de prefixo, o valor de R nesse caso será 2. Ou seja, precisa-se procurar na árvore a última vez que 2 apareceu através de um incremento no vetor de soma de prefixos, e que seja antes do tempo $t = 3$. O primeiro passo do algoritmo consiste em descer através da raiz até o nó com tempo $t = 3$. A seguir, sobe-se a partir do nó [3, 3] em direção a raiz, observando se, em algum momento, o valor de mínimo e máximo do nó contém R. Isso só ocorre nesse caso no nó raiz da árvore. A partir daí atualiza-se a posição de busca para o nó filho à esquerda do nó atual, no caso, o vértice [1, 2]. Com isso, garante-se que o fim do intervalo do nó atual será menor que o tempo atual. Assim, não se fazem necessárias mais verificações com relação ao t atual, pois qualquer nó filho desse vértice está atrás do tempo t, e possui o mesmo subproblema resolvido na versão parcial da estrutura.

No Algoritmo 10 tem-se a implementação da consulta do topo em uma pilha totalmente retroativa. Na linha 3, procura-se o nó folha correspondente ao tempo t. A seguir, entre as linhas 4 e 12, procura-se o primeiro nó pai do nó folha encontrado em que o valor procurado i está entre o mínimo e o máximo do nó, e que o último nó visitado seja um nó direita nessa árvore. Depois disso, na linha 14, anda-se um passo para a esquerda na árvore, garantindo que a subárvore atual está em posições menores que t. Após isso, entre as linhas 20 a 29, procura-se a subárvore mais à direita que mantém as restrições anteriormente descritas.

Algoritmo 10 Implementação da função de consulta em uma pilha totalmente retroativa

```
1: função PEAK(t)
2:     t++
3:     no ← getNode(1,1,n,t)
4:     enquanto no ≠ 1 faça
5:         d ← (no&1)
6:         no/ = 2
7:         nxt ← (no∗2)
8:         propagate(nxt,L[nxt],R[nxt])
9:         propagate(nxt+1,L[nxt+1],R[nxt+1])
10:        se i ≥ tr[nxt].minIns e i ≤ tr[nxt].maxIns e d > 0 então
11:            fim enquanto;
12:        fim se
13:    fim enquanto
14:    no∗ = 2
15:    propagate(no,L[no],R[no])
16:    se i < tr[no].minIns ou i > tr[no].maxIns então
17:        no++
18:    fim se
19:    propagate(no,L[no],R[no])
20:    enquanto L[no] ≠ R[no] faça
21:        nxt ← (no∗2)
22:        propagate(nxt,L[nxt],R[nxt])
23:        propagate(nxt+1,L[nxt+1],R[nxt+1])
24:        se i ≥ tr[nxt+1].minIns e i ≤ tr[nxt+1].maxIns e L[nxt+1] < t então
25:            no ← nxt+1
26:        senão
27:            no ← nxt
28:        fim se
29:    fim enquanto
30:    Retorna tr[no].data
31: fim função
```

Com isso, no pior caso, o algoritmo de consulta descerá uma vez, subirá uma vez, e descerá novamente na árvore inteira. Como a árvore contém altura $\lg n$, a complexidade final dessa função é $3 * \lg n = O(\lg n)$.

3.2.3 Retroatividade não-consistente da pilha

A retroatividade não-consistente consiste em reportar a primeira operação de retirada da pilha que se tornou inconsistente após a adição ou remoção das operações inerentes à estrutura.

Para implementação de uma pilha não-consistente, pode-se utilizar a mesma lógica relacionada à codificação dessas estruturas em retroatividade parcial e total. É possível manter a pilha em um vetor, como o utilizado nas outras formas de retroatividade, e, assim, o problema se resume a trabalhar com um vetor de somas acumuladas. Além disso, é possível manter dois conjuntos T_{push} e T_{pop} com as operações *Push* e *Pop* ordenadas pelo seu tempo de realização. Ambos os conjuntos contém como chave os tempos de inserção das operações. T_{push} tem como valor de uma chave um par contendo o valor x inserido e o tempo da operação *Pop* relacionada a essa inserção (sendo nulo caso não exista operação relacionada). Já T_{pop} contém como valor o tempo da operação *Push* que a operação *Pop* está relacionada.

Seja V o vetor em que a pilha está implementada, e p o índice em que o elemento x foi adicionado no vetor V após a execução da operação $Insert(t, Push(x))$. Então, a próxima operação inconsistente após essa inserção é a operação *Pop* que retira da pilha o elemento da posição p.

Algoritmo 11 Função que insere a operação $Push(x)$ no tempo t em uma pilha com retroatividade não-consistente

```
1:  função INSERTPUSH(t, x)
2:      prefixTree.updateInsert(t,x)
3:      T_push.insert(t, std :: makePair(x, NULLVALUE))
4:      BST :: Treap :: iteratornextInconsistenceIt ← T_pop.lowerBound(t)
5:      se nextInconsistenceIt == NULL então
6:          Retorna NULLVALUE
7:      senão
8:          Retorna (*nextInconsistenceIt).first
9:      fim se
10: fim função
```

No Algoritmo 11 tem-se a função que implementa a inserção de uma operação $Push(x)$ no tempo t em uma pilha com retroatividade não-consistente. Inicialmente, é atualizada a árvore com a soma de prefixos referente às operações realizadas na pilha. A seguir, é inserida no conjunto T_{push}, a operação conforme explicado anteriormente. A priori, essa operação *Push* não contém uma operação *Pop* relacionada. Caso exista uma operação *Pop* que afeta a inserção atual, essa ação será retornada como inconsistente, e, posteriormente será corrigida.

A inserção da operação *Pop* no tempo t deixa inconsistente a próxima operação *Pop* após o tempo t.

Algoritmo 12 Operação de inserção da operação *Pop* no tempo t em uma pilha com retroatividade não-consistente

```
1:  função INSERTPOP(t)
2:      pair < int, T > peak ← prefixTree.getPeakPair(prefixTree.getPrefixSum(t), t)
3:      BST :: Treap :: iterator peakIt ← T_push.lowerBound(peak.first)
4:      pair < int, std :: pair < T, int >> peakPair ← ∗peakIt
5:      peakPair.second.second ← t
6:      peakIt.modify(peakPair.first, peakPair.second)
7:
8:      T_pop.insert(t, peak.first)
9:      prefixTree.updateDelete(t)
10:
11:     BST :: Treap :: iterator nextInconsistenceIt ← ++tpop.lowerBound(t)
12:     se nextInconsistenceIt == NULL então
13:         retorna NULLVALUE
14:     senão
15:         retorna (∗nextInconsistenceIt).first
16:     fim se
17: fim função
```

No Algoritmo 12, tem-se a implementação da função para a inserção de uma operação *Pop* no tempo t em uma pilha com retroatividade não-consistente. Uma operação *Pop* no tempo t retira da estrutura o elemento que está no topo da pilha em t. Portanto, essa operação *Pop* está diretamente relacionada ao elemento do topo da pilha nesse tempo t. Assim, inicialmente, é obtido o elemento do topo da pilha utilizando a estrutura que mantém a soma de prefixos (linha 2). Esse elemento do topo da pilha no tempo t deve ser atualizado no conjunto T_{push}, e, portanto, ele é procurado na estrutura utilizando-se a função *lower_bound*, e posteriormente, atualizado (linhas 3 a 6). A seguir, é inserida no conjunto T_{pop}, a operação *Pop* no tempo t, e essa operação é relacionada ao elemento que foi retirado da fila por essa ação. A seguir, é encontrada a operação *Pop* realizada na estrutura após o tempo t (linhas 11 a 16).

Algoritmo 13 Operação de deleção da operação $Push(x)$ no tempo t em uma pilha com retroatividade não-consistente

```
1: função DELETEPUSH(t)
2:     BST :: Treap :: iterator f ← T_push.lowerBound(t)
3:     nextInconsistenceTime ← (*f).second.second
4:     T_push.erase(t)
5:     prefixTree.erasePush(t)
6:     retorna nextInconsistenceTime
7: fim função
```

No Algoritmo 13, é apresentada a implementação da função para remoção da operação $Push(x)$ no tempo t. A remoção dessa operação deixa inconsistente a operação Pop que está relacionada a essa inserção. Então, consulta-se em T_{push} qual o tempo da operação Pop relacionada à operação $Push(x)$ que está sendo removida. Após a obtenção da próxima inconsistência, retira-se a operação $Push(x)$ tanto do conjunto T_{push} quanto da estrutura que armazena a soma dos prefixos. Finalmente, retorna-se o tempo da próxima operação Pop inconsistente, caso exista.

No Algoritmo 14, tem-se a implementação para a função de deleção da operação Pop em uma pilha com retroatividade não-consistente. Inicialmente, procura-se no conjunto T_{pop} o elemento que foi removido no tempo t. Essa operação está relacionada a uma operação *Push*, e a informação de qual é a operação *Push* relacionada à operação Pop que está sendo removida está armazenada no conjunto T_{pop}. Esse *Push* é atualizado, pois a operação Pop que o afetava está sendo removida (linhas 5 a 8). Novamente, é removida, do conjunto T_{pop}, a operação realizada no tempo t, e retornada como inconsistente a primeira operação Pop realizada após o tempo t.

Algoritmo 14 Operação de deleção da operação *Pop* no tempo t em uma pilha com retroatividade não-consistente

```
1:  função DELETEPOP(t)
2:      BST :: Treap :: iterator popIt = T_pop.lowerBound(t)
3:      pair<int,int> popPair ← *popIt
4:
5:      BST :: Treap :: iterator pushIt ← T_push.lowerBound(popPair.first)
6:      pair<int,std :: pair<T,int>> pushPair ← *pushIt
7:      pushPair.second.second ← NULLVALUE
8:      pushIt.modify(pushPair.first, pushPair.second)
9:
10:     T_pop.erase(t)
11:     prefixTree.erasePop(t)
12:
13:     BST :: Treap :: iterator nextInconsistenceIt ← T_pop.lowerBound(t)
14:     se nextInconsistenceIt == NULL então
15:         retorna NULLVALUE
16:     senão
17:         retorna (*nextInconsistenceIt).first
18:     fim se
19: fim função
```

3.3 Fila de prioridade

Fila de prioridade é um tipo abstrato de dados que mantém uma coleção de elementos, cada um com uma prioridade associada, o que define sua posição na fila [12].

Essa estrutura é comumente implementada sobre um *Fibonacci Heap*, que é um tipo de estrutura que permite operações de inserção, remoção e extração do mínimo em tempo logarítmico no número de elementos da estrutura. Porém, não será utilizada essa abordagem para a implementação da fila de prioridade retroativa.

As funções que essa estrutura implementa em sua versão original são:

- *Push(x):* insere na estrutura o elemento x;
- *Pop():* remove o elemento de maior valor adicionado na estrutura;
- *GetPeak():* obtém o elemento de menor valor da estrutura no tempo atual.

Já em sua versão retroativa, definimos:

- *Insert(t, Push(x)):* insere na estrutura o elemento x no tempo t;
- *Insert(t, Pop()):* remove o elemento de maior valor adicionado na estrutura no tempo t;
- *Delete(t, Push(x)):* remove a operação *Push* realizada na estrutura no tempo t. A estrutura deve conter uma operação *Push* correspondente;
- *Delete(t, Pop()):* remove a operação *Pop* realizada na estrutura no tempo t. A estrutura deve conter uma operação *Pop* correspondente;
- *GetPeak(t):* obtém o menor valor no tempo t da estrutura.

Essa estrutura de dados pode ser aplicada em problemas como o *vertical ray shooting* [13], em que existem alguns segmentos de reta horizontais no espaço $\mathbb{R}^2$, e algumas consultas que correspondem a remover o menor elemento do conjunto contido em uma coordenada x.

Foram realizadas implementações em C++ da estrutura retroativa fila de prioridade como as descritas a seguir, que podem ser encontradas em https://github.com/juniorandrade1/Master/tree/master/src/Priority_Queue.

3.3.1 Retroatividade parcial da fila de prioridade

A retroatividade parcial de uma fila de prioridade consiste em realizar atualizações no passado e consultas no estado atual da estrutura. Porém, mesmo com a restrição de que as consultas sejam realizadas no estado atual da estrutura, a sua implementação não é trivial, uma vez que uma modificação no passado pode gerar um efeito cascata.

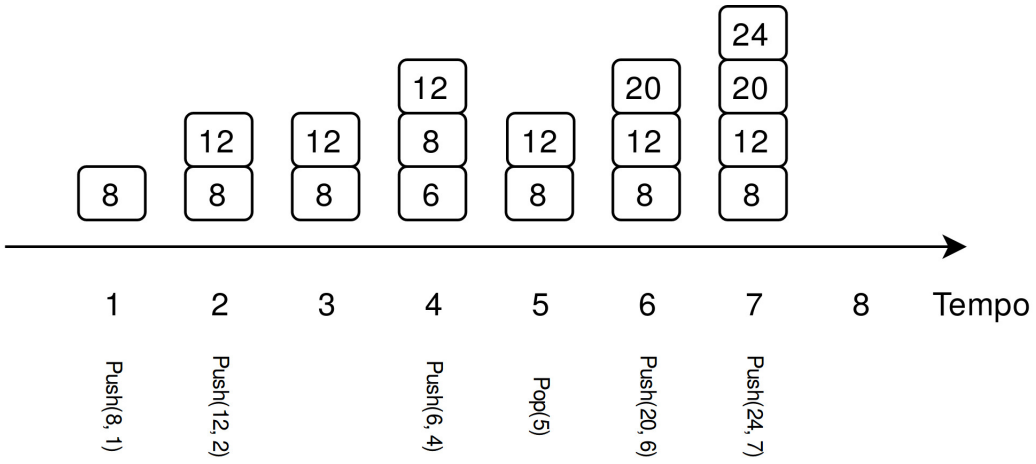

Figura 3.9 Fila de prioridade retroativa. Fonte: O autor

Por exemplo, na Figura 3.9, tem-se a representação da linha temporal de uma fila de prioridade, em que os menores elementos são removidos e consultados pelas funções *Pop* e *GetPeak*. Até o momento, apenas uma operação *Pop* foi realizada no tempo 5. Ou seja, supondo que fosse realizada uma operação *GetPeak* em cada tempo entre 1 e 7, as soluções seriam $\{8,8,8,6,8,8,8\}$

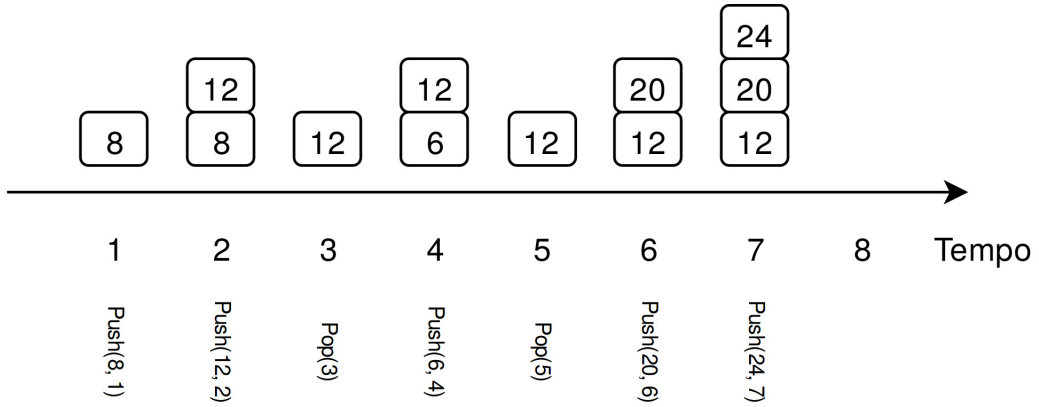

Figura 3.10 Fila de prioridade retroativa após a adição de uma operação $Pop(3)$. Fonte: O autor.

Na Figura 3.10, tem-se a adição de uma operação $Pop()$ no tempo 3. Imediatamente, os valores das consultas $GetPeak()$ mudam para $\{8,8,12,6,12,12,12\}$. Como se trata da versão parcialmente retroativa de uma fila de prioridade, o que importa para o algoritmo é que os elementos que estão na fila no tempo atual são $\{12,20,24\}$ ao invés dos elementos $\{8,12,20,24\}$, que corresponde ao estado da fila anterior à inserção da operação *Pop* no tempo 3.

Seja Q_t a fila de prioridade com as operações realizadas até o tempo t e Q_{now} a fila de prioridade no tempo atual.

Lema 7. Insert(t, Push(x)) *insere em* Q_{now} $\max\{x, x' | x' \text{ foi deletado após o tempo } t\}$ *[13]*

Demonstração. Uma inserção qualquer na fila de prioridade insere um novo elemento na fila no tempo atual. Esse elemento pode ser o elemento x adicionado no tempo t ou algum outro elemento que estava na fila após o tempo t que foi deletado. O elemento que é mantido até o tempo presente é o máximo entre o item adicionado e algum outro elemento deletado após o tempo t. □

Entretanto, a deleção, em linhas gerais, tende a ser custosa do ponto de vista computacional e ocasiona o problema do efeito cascata na estrutura. Para evitar esse efeito, é

necessário analisar o problema por meio de outra perspectiva, sendo possível reescrever o lema 7. Define-se na fila de prioridade uma ponte no tempo t se $Q_t \subseteq Q_{now}$. Ou seja, são tempos em que tem-se versões "estáveis" da fila do tempo atual.

Lema 8. *Seja t' a primeira ponte antes do tempo t, então* $\max\{x' | x' \text{ foi deletado após o tempo } t\} = \max\{x' \notin Q_{now} | x' \text{ foi inserido após o tempo } t'\}$ *[13]*

Para a implementação de uma fila de prioridade parcialmente retroativa, é necessário que se mantenha três árvores binárias balanceadas distintas. A primeira árvore binária representa exatamente a fila atual Q_{now}. Já a segunda árvore de busca representa as operações de inserção dos elementos que não estão em Q_{now} no tempo atual. Dentro de cada nó dessas árvores, é necessário manter algumas informações importantes, como o mínimo, o máximo e o tamanho da sub-árvore enraizada no nó em questão.

Algoritmo 15 Exemplo da estrutura do nó utilizado na árvore binária balanceada na implementação do problema (*Treap*)

```
1: template <typename K, typename V>
2: função TREAPNODE
3:     TADKey key
4:     TADData data
5:     Inteiro prior, sz
6:     Node *l, *r
7:     pair < TADData, TADKey > mn, mx
8: fim função
```

No Algoritmo 15 é apresentada a estrutura, para cada nó, utilizada pelas árvore Q_{now} e pela segunda árvore de busca com os elementos que não estão contidos na fila de prioridade no tempo atual. Cada nó contém uma chave *(key)* e um valor *(data)* em sua estrutura. As variáveis *sz*, *mn* e *mx* representam, respectivamente, o tamanho da sub-árvore e os valores mínimo e máximo na subárvore enraizada desse nó. As outras variáveis correspondem aos valores relacionados com a criação da estrutura. As variáveis *l* e *r* representam as sub-árvores esquerda e direita do nó, enquanto *prior* define a prioridade de cada nó. Isso é necessário uma vez que a estrutura se baseia em uma árvore binária de busca auto-balanceável.

A terceira árvore contém todas as atualizações realizadas na fila, tanto as de inserção, quanto as de retirada do valor mínimo. Nessa árvore, as chaves são os tempos de inserção, enquanto os valores são definidos da seguinte forma:

- 0 para *Insert(x)* no tempo t com $x \in Q_{now}$;

- +1 para *Insert(x)* no tempo t com $x \notin Q_{now}$;
- -1 para *Pop()* no tempo t.

Todo nó x na árvore contém a soma de todos os valores que estão contidos em sua sub-árvore. Com essa terceira árvore, é possível descobrir os tempos t que são pontes. Uma ponte é definida por todas as somas de prefixo na terceira árvore que contém soma nula. Observe que essa árvore nunca contém somas de prefixo negativas, pois isso implicaria em operações de remoção em filas vazias.

Existem algumas maneiras para, dado um tempo t, encontrar a primeira ponte anterior e posterior a t. A ideia mais direta consiste em iterar, a partir do tempo t, procurando linearmente pelas pontes a partir de t. No pior caso, essa solução tem complexidade linear no número de operações realizadas. Outra abordagem para o mesmo problema consiste em efetuar uma busca binária a partir do tempo t na árvore contendo as somas de prefixo. Assim, é possível encontrar as pontes em complexidade $O(\lg^2 n)$.

Algoritmo 16 Implementação da função para encontrar a próxima ponte após o tempo t.

```
1:  função GETNEXTBRIDGE(t)
2:      lo ← t
3:      hi ← n
4:      enquanto lo < hi faça
5:          md ← (lo+hi)/2
6:          se prefixTree.query(t, md).min então
7:              lo ← md + 1
8:          senão
9:              hi ← md
10:         fim se
11:     fim enquanto
12:     retorna lo
13: fim função
```

No Algoritmo 16, tem-se a implementação da função que permite encontrar a primeira ponte após o tempo t. Essa função é executada em tempo $O(\lg^2 n)$, pois cada consulta na árvore de prefixos é executada em tempo $O(\lg n)$ assim como o laço da linha 4 até a linha 10. Nesse laço, é realizada uma busca binária procurando pelo primeiro instante de tempo t' em que o valor mínimo da consulta do intervalo $[t, md]$ é igual a 0. Se isso for verdadeiro, então existe pelo menos uma ponte no intervalo $[t, md]$. Como a função de pesquisa é

monótona, a utilização da busca binária é válida. A implementação da função para encontrar a primeira ponte antes do tempo t segue exatamente o mesmo padrão da função apresentada no Algoritmo 16.

Para otimizar as consultas das pontes, pode-se utilizar uma ideia similar à apresentada na estrutura retroativa pilha, armazenando, na terceira árvore citada o máximo e o mínimo das somas de prefixo em cada sub-árvore, obtendo a complexidade temporal de $O(\lg n)$ na consulta das pontes anterior e posterior a t.

Por exemplo, a obtenção da primeira ponte após o tempo t consiste em encontrar o menor índice $i \geq t$, tal que a soma de prefixo tenha valor igual a zero. Pela garantia da consistência das operações na fila de prioridade, não existirá soma de prefixo negativa nessa estrutura. Sendo assim, o mínimo de uma soma de prefixo nunca será menor que zero. Portanto, se no intervalo $[t, \infty]$ o mínimo da soma de prefixo for igual a zero, existe um conjunto P de índices (não necessariamente apenas um) com a soma de prefixo igual a zero. No conjunto P, a primeira ponte após o tempo t será igual ao valor mínimo mantido em P. Dentro de um nó da árvore de segmentos, pode-se armazenar, além dos valores mínimo e máximo dentro do conjunto, os índices de menor e maior valor de ocorrência da soma de prefixo mínima. Desta maneira, é possível obter a primeira ponte anteriormente e posteriormente ao tempo t utilizando a própria consulta na árvore de segmentos, retirando a busca binária realizada no Algoritmo 16. Essa abordagem consome tempo logarítmico no tamanho da linha temporal em que a árvore de segmentos está implementada.

As outras operações inerentes a uma fila de prioridade parcialmente retroativa podem ser implementadas tendo como base as ideias previamente discutidas.

Lema 9. *Após uma operação $Insert(t, Pop())$, o elemento a ser removido de Q_{now} é:*

$$\min_{x \in Q_{t'}} x$$

onde t' é a primeira ponte após o tempo t.

O problema principal da formulação do lema de inserção do $Pop(t)$ é que não se tem explicitamente Q_t para todo tempo t, caso contrário, resolveria-se o problema da fila de prioridade totalmente retroativa. Porém, isso é um pouco diferente se forem analisados somente os tempos considerados pontes na estrutura. Seja $I_{>t}$ todos os elementos inseridos após o tempo t na fila de prioridade (independentemente se foram removidos posteriormente e não estão mais em Q_{now}). Então, tem-se que a seguinte propriedade é verdadeira:

$$Q_{t'} = Q_{now} \cap I_{>t'}$$

Assim, para encontrar o valor mínimo em Q_t, basta encontrar em Q_{now}, dentre todos os elementos com inserção após o tempo t', o menor desses elementos. Como Q_{now} está armazenado em uma árvore binária balanceada, tem-se que é possível obter o mínimo desses elementos mantendo na árvore binária Q_{now} os valores de mínimo de uma subárvore em cada nó. O mesmo pode ser com operações como máximo e soma, caso necessário.

Algoritmo 17 Inserção da operação *Push(t, data)*

```
1:  função INSERTPUSH(t, data)
2:      tl ← bridges.getLastBridge(t)
3:      Q_del.insert(t, data)
4:      type[t] ← 2
5:      bridges.update(t, 1)
6:      add ← Q_del.getMaximumValueAfter(tl)
7:      Q_del.erase(add.second)
8:      Q_now.insert(add.second, add.first)
9:      type[add.second] ← 1
10:     bridges.update(add.second, −1)
11: fim função
```

No Algoritmo 17, tem-se a implementação da função de inserção da operação *Push* na estrutura retroativa no tempo t e com os valores dos dados $data$. A variável $data$ tem tipo T, pois equivale a um tipo qualquer de dados que tem uma ordenação lógica. Essa estrutura utiliza as três árvores citadas anteriormente, Q_{del}, Q_{now} e $bridges$, que representam os elementos que não estão na fila atual, os elementos que estão na fila atual e a árvore representando a linha temporal das operações na estrutura armazenadas na forma de somas de prefixo por meio dos valores atribuídos pelas regras descritas. Além disso, é definido, para cada tempo t através da variável $type$, o tipo de operação que está sendo realizada no tempo t. A variável $type[t]$ é definida da seguinte forma:

- $type[t] = 1$: existe uma operação de inserção no tempo t e esse valor está contido na árvore Q_{now};
- $type[t] = 2$: existe uma operação de inserção no tempo t e esse valor está contido na árvore Q_{del};
- $type[t] = 3$: existe uma operação de remoção do elemento mínimo no tempo t.

Na linha 2, tem-se a procura pela primeira ponte antes do tempo t, seguida da inserção do elemento atual na árvore dos elementos não adicionados a Q_{now}, uma vez que não se sabe

se o elemento inserido atualmente é o elemento a ser inserido em Q_{now}. Com a inserção, é necessária a atualização do tipo de operação realizada na variável $type$, que é atualizada na linha 4. Após essa operação, atualiza-se na linha 5 a variável $bridge$, mantendo-se a linha temporal das operações atualizada. Na linha 6, procura-se pelo nó que contém o maior valor possível dentre os inseridos que não estão na fila atualmente e que foram inseridos após a ponte situada no tempo t'. Esse elemento é o objeto que deve ser inserido em Q_{now}. As próximas três linhas contém a inserção na fila atual, bem como a deleção do elemento do conjunto dos elementos que foram removidos da fila e a atualização da linha temporal total da estrutura em $bridges$.

Algoritmo 18 Inserção da operação de remoção Pop(t)

```
1:  função INSERTPOP(t)
2:      type[t] ← 3
3:      tl ← bridges.getNextBridge(t)
4:      bridges.update(t, −1)
5:      pair < T, int > rem ← Q_now.getMinimumValueBefore(tl)
6:      Q_now.erase(rem.second)
7:      Q_del.insert(rem.second, rem.first)
8:      bridges.update(rem.second, 1)
9:      type[rem.second] ← 2
10: fim função
```

De maneira similar, tem-se o Algoritmo 18, que realiza a inserção da operação Pop na estrutura retroativa no tempo t. Na linha 3, procura-se a primeira ponte posterior ao tempo t, e, em seguida, é realizada a atualização da estrutura de pontes, uma vez que uma operação Pop foi realizada no tempo t. Como definido anteriormente, a fila de prioridade no tempo t' equivale à intersecção dos elementos atualmente na fila com os elementos inseridos após o tempo t'. Essa operação é realizada na linha 5. Novamente, as últimas três linhas são equivalentes à remoção do elemento da fila atual, à inserção dele na árvore dos elementos que não estão na fila atualmente, e à atualização da linha temporal da estrutura.

No Algoritmo 19, é necessário tratar dois tipos de casos. No primeiro caso, o elemento que foi inserido no tempo t será removido e ele está contido na fila de prioridade correspondente ao tempo atual na estrutura. Portanto, a sua simples remoção da árvore Q_{now} já é suficiente para a atualização da estrutura.

Caso contrário, o elemento inserido na fila não está contido na fila de prioridade atual. Isso significa que foi executada uma operação Pop em um elemento que não estará mais presente na linha do tempo da estrutura. Portanto, é necessário encontrar em Q_{now} o elemento que

não estará mais presente na fila, o que é realizado por meio da ideia de pontes utilizadas nas funções anteriores nas linhas 7 à 9. Então, esse elemento encontrado será removido de Q_{now}, pois, em algum momento, ele será retirado da fila por uma função *Pop* que anteriormente retirava o elemento que foi inserido no tempo t. Das linhas 12 à 15, retira-se o elemento que estava na linha do tempo da estrutura, mas não estava na fila de prioridade presente. As outras linhas seguem a mesma lógica das outras funções, em que são atualizadas as árvores cumulativas relacionadas às pontes, bem como atualizações dos tipos de operações realizadas no tempo t.

Algoritmo 19 Função *RemovePush(t)* em uma fila de prioridade parcialmente retroativa

```
1:  função REMOVEPUSH(t)
2:      se type[t] == 1 então
3:          Q_now.erase(t)
4:          type.erase(t)
5:      senão
6:          tl ← bridges.getNextBridge(t)
7:          pair<T, int> add ← Q_now.getMinimumValueBefore(tl)
8:          Q_now.erase(add.second)
9:          Q_del.insert(add.second, add.first)
10:         type[add.second] ← 2
11:         bridges.update(add.second, 1)
12:         Q_del.erase(t)
13:         bridges.update(t, −1)
14:         type.erase(t)
15:     fim se
16: fim função
```

Já no Algoritmo 20, tem-se a implementação da função de remoção de uma operação do tipo *Pop* na estrutura parcialmente retroativa. A remoção de uma operação *Pop* não remove nenhum elemento da estrutura como um todo, mas modifica as árvores Q_{del} e Q_{now}. Essa remoção insere um novo elemento na fila de prioridade atual, elemento proveniente da árvore dos elementos que foram removidos da fila atual. Então, a princípio, encontra-se em Q_{del} esse elemento a ser re-inserido na estrutura atual, e, posteriormente, inserido na fila Q_{now}, como implementado entre as linhas 4 e 6. Novamente, atualiza-se a árvore relacionada à soma de prefixo, e o tipo da operação no tempo t (que, no caso, deixa de existir).

Algoritmo 20 Função *RemovePop(t)* em uma fila de prioridade parcialmente retroativa

```
1: função REMOVEPOP(t)
2:     bridges.update(t, 1)
3:     tl ← bridges.getLastBridge(t)
4:     pair<T, int> add = Q_del.getMaximumValueAfter(tl)
5:     Q_del.erase(add.second)
6:     Q_now.insert(add.second, add.first)
7:     type[add.second] ← 1
8:     bridges.update(add.second, −1)
9:     type.erase(t)
10: fim função
```

Pode-se observar que as funções são, de certa forma, inversas entre si. Na função de inserção da operação *Push*, procura-se o elemento de maior valor posterior à última ponte. Já na função de remoção da operação de inserção, tem-se o inverso. Ou seja, procura-se o elemento de menor valor que ocorre anteriormente à próxima ponte.

Todas as funções apresentadas utilizam apenas árvores binárias balanceadas em suas operações, e, portanto, a complexidade final de uma fila de prioridade parcialmente retroativa é logarítmica no número de operações realizadas.

3.3.2 Retroatividade total em $O(\sqrt{m}\lg n)$ da fila de prioridade

A diferença entre uma fila de prioridade parcialmente retroativa e totalmente retroativa está nas consultas, que agora podem ser realizadas em qualquer tempo da estrutura. Nesse caso, o efeito cascata da estrutura se torna complicado de tratar diretamente. Entretanto, é possível obter a estrutura totalmente retroativa com um custo multiplicativo extra $O(\sqrt{m})$, em que m é o número de operações realizadas.

A solução para a fila de prioridade totalmente retroativa em tempo $O(\sqrt{m}\lg n)$ por operação é alcançada da seguinte forma. Definem-se alguns pontos na linha temporal que existirão filas de prioridade parcialmente retroativas. Para cada atualização no tempo t, percorre-se cada fila de prioridade parcialmente retroativa nos pontos definidos t', tal que $t' > t$. Esse conjunto de operações será executado no pior caso em tempo $O(p \cdot \lg n)$, em que p é o número de pontos definidos. Ou seja, caso existam muitos pontos interessantes, a complexidade total do algoritmo crescerá uma vez que cada atualização deve percorrer todos esses pontos.

Já para as funções de consulta no tempo t, basta encontrar o primeiro ponto $t' < t$, e a partir dessa fila de prioridade parcialmente retroativa $Q_{t'}$, executar todas as operações

realizadas entre os tempos t' e t em $O(\lg n)$ por operação. Ou seja, caso os pontos sejam escolhidos arbitrariamente, podem existir muitas operações entre as estruturas parcialmente retroativas adjacentes, gerando uma execução da operação de consulta de maneira mais lenta.

Então, necessita-se definir um valor de tamanho do intervalo em que os pontos interessantes não existam em grande número, pois as operações de atualização ficarão lentas, e ao mesmo tempo o número de operações entre dois adjacentes quaisquer também seja pequeno. Essas grandezas são inversamente proporcionais, uma vez que ao aumentar-se o número de pontos interessantes, diminui-se o número de operações entre dois pontos de interesse, e vice-versa.

Suponha que as operações de atualização e consulta na estrutura sejam equiprováveis. Então, tem-se:

Lema 10. *O ponto ótimo de escolha para m operações realizadas de consulta e de atualização distribuídas equiprovavelmente pela linha do tempo da estrutura são intervalos de tamanho* $\sqrt{m}$.

Demonstração. Pode-se definir a função de complexidade da estrutura da seguinte forma: como consultas e atualizações são equiprováveis, se os pontos de interesse forem escolhidos otimamente, eles serão distribuídos uniformemente pela linha temporal. Ou seja, se a linha temporal tem m momentos, e deseja-se deixar b atualizações entre cada ponto de interesse, tem-se que esses pontos estarão nas posições $\{0, b, 2 \cdot b, 3 \cdot b, \cdots, \lceil \frac{m}{b} \rceil\}$. Assim, aumentar b consiste em diminuir o número de pontos de interesse, porém, aumentando o número de operações entre pontos interessantes.

Ou seja, deseja-se encontrar um ponto de equilíbrio em que tanto operações de atualização quanto de consulta são executadas de modo que o tempo de execução da maior delas seja o mínimo possível. Então, seja U o tempo de atualização e C o tempo de consultas, é necessário encontrar b tal que:

$$U \cdot \frac{m}{x} = C \cdot x$$

Como o tempo de atualização e consulta são os mesmos, tem-se:

$$\frac{m}{b} = b \Rightarrow m = b^2 \Rightarrow b = \sqrt{m}$$

Portanto, o tamanho do intervalo ótimo é igual ao número de pontos interessantes, pois tanto consulta como atualização na estrutura de fila de prioridade parcialmente retroativa consomem tempo $\lg(n)$, o que completa a prova.

□

Algoritmo 21 Variáveis utilizadas em uma fila de prioridade totalmente retroativa

```
1: Inteiro m
2: Inteiro b
3: vector< PartialPriorityQueue< T > > p;
4: multiset< Operation > all;
```

No Algoritmo 21, tem-se o exemplo das variáveis utilizadas em uma fila de prioridade totalmente retroativa. A variável *m* consiste no tamanho do conjunto de operações realizadas, como descrito anteriormente; *b* representa o tamanho ótimo do intervalo escolhido para a criação dos pontos de interesse. Em *p*, é armazenado o conjunto de filas de prioridade parcialmente retroativas, em que o *i*-ésimo elemento representa o ponto de interesse $i \cdot b$. A variável *all* representa todas as operações realizadas, ordenadas por tempo.

Algoritmo 22 Função *InsertPush(t, data)* em uma fila de prioridade totalmente retroativa

```
1: função INSERTPUSH(t, data)
2:     st ← t/b
3:     para i == st + 1 até i < p.size() faça
4:         p[i].insertPush(t, data)
5:         i++
6:     fim para
7:     all.insert(Operation(t, 0, data))
8: fim função
```

No Algoritmo 22, tem-se a implementação da função de inserção de um *Push* na estrutura. Em *st*, mantêm-se o índice do primeiro ponto de interesse antes do tempo t. A seguir, insere-se nos pontos de interesses após o primeiro ponto de interesse antes do tempo t o elemento atual no tempo t. Depois, insere-se a operação realizada na lista de operações. Nesse caso, o segundo parâmetro corresponde à operação realizada. As operações de inserção da operação *Pop*, bem como as remoções das operações de atualização podem ser realizadas da mesma maneira que as apresentadas anteriormente.

Algoritmo 23 Função *getPeak(t)* em uma fila de prioridade totalmente retroativa

```
1:  função GETPEAK(t)
2:      st ← t/b
3:      vector < Operation > op
4:      multiset :: iterator it ← all.lowerBound(Operation(st ∗ b, −1, −1))
5:      enquanto it ≠ all.end() e it− > t ≤ t faça
6:          op.emplaceBack(∗it)
7:          it++
8:      fim enquanto
9:
10:     para i ← 0 e i < (int)op.size() faça
11:         se op[i].op == push então
12:             p[st].insertPush(op[i].t, op[i].data)
13:         senão
14:             p[st].insertPop(op[i].t)
15:         fim se
16:         i++
17:     fim para
18:     peak ← p[st].getPeak()
19:
20:     reverse(op.begin(), op.end())
21:     para i ← 0 e i < (int)op.size() faça
22:         se op[i].op == push então
23:             p[st].removePush(op[i].t)
24:         senão
25:             p[st].removePop(op[i].t)
26:         fim se
27:         i++
28:     fim para
29:     retorna peak
30: fim função
```

A implementação da função de consulta em uma fila de prioridade totalmente retroativa pode ser vista no Algoritmo 23. A função se divide em três partes principais. A primeira delas consiste em descobrir quais operações foram realizadas entre o último ponto de interesse e o ponto t. Na função, st corresponde ao índice do último ponto de interesse encontrado.

Então, o tempo em que o último ponto de interesse se encontra é $st \cdot b$. Então, adiciona-se em op as operações realizadas no intervalo $[st \cdot b, t]$. Sabendo-se que o índice do último ponto de interesse é st, então basta adicionar em $p[st]$ todas as operações realizadas entre $st \cdot b$ e t. Assim, ao final das inserções das operações desse intervalo, tem-se a fila de prioridade retroativa correspondente a Q_t, e, então, basta retirar o elemento que está no topo da fila.

Entre as linhas 20 a 28, são realizadas as operações inversas de modo a reverter as inserções das operações realizadas no ponto de interesse, mantendo a invariante do problema para consultas futuras. Com isso, é possível implementar uma fila de prioridade totalmente retroativa com complexidade $O(\sqrt{m} \lg n)$.

3.3.3 Retroatividade total em tempo poli-logarítmico da fila de prioridade

Demaine *et al.* [14] propuseram uma fila de prioridade em tempo $O(\lg^2 n)$ por atualização. É possível transformar uma estrutura de dados *time-fusible* com um custo multiplicativo logarítmico extra por meio de uma técnica chamada *hierarchical checkpointing*. Duas estruturas E_1 e E_2 são ditas *time-fusibles* se elas representam a mesma estrutura em tempos consecutivos disjuntos.

Em outras palavras, seja $I_{E_1} = [l_1, r_1]$ o intervalo temporal correspondente à estrutura E_1 e $I_{E_2} = [l_2, r_2]$ o intervalo temporal relativo à estrutura E_2. Logo, essas estruturas são *time-fusible* se $r_1 < l_2$ e $r_1 + 1 = l_2$. Ou seja, sua união gera uma estrutura $E_f = E_1 \cup E_2$ que abrange o intervalo $I_{E_f} = [l_1, r_2]$.

Com essa definição, é possível gerar uma árvore binária sobre a linha do tempo da estrutura, de modo que cada nó represente um segmento contínuo dessa linha temporal. Essa transformação é denotada por Demaine *et al.* como *hierarchical checkpointing* [14].

O primeiro passo para a construção de uma estrutura utilizando a técnica consiste na criação da *checkpoint tree* - uma árvore binária de busca balanceada em que cada nó contém uma estrutura parcialmente retroativa que contenha todos as atualizações realizadas nas subárvores daquele nó. Essa árvore é similar à árvore de segmentos apresentada anteriormente, com a diferença que cada nó contém uma fila de prioridade parcialmente retroativa correspondente ao intervalo que o nó abrange.

Como a fila de prioridade parcialmente retroativa já foi explicada anteriormente, ela será utilizada como uma ferramenta para encontrar, após cada atualização, quais elementos devem ser inseridos ou removidos do estado atual da estrutura. Para cada nó, serão mantidos dois conjuntos auxiliares: Q_{now}, contendo o conjunto de todos os objetos contidos no tempo

$t = \infty$ e Q_{del}, contendo o conjunto de todos os objetos removidos da fila em algum ponto do passado.

Se a fila de prioridade estiver vazia no tempo t e uma operação de deleção de um elemento for executada, então, essa operação irá inserir uma chave com valor infinito em Q_{del} (equivalente a uma inserção de um valor ∞ no tempo t e imediatamente a sua remoção da fila no tempo t).

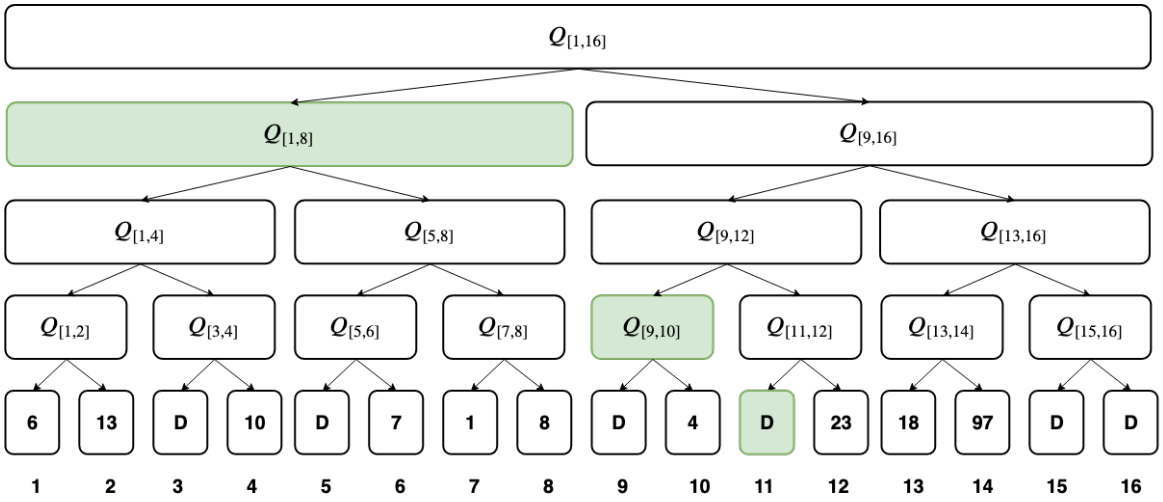

Figura 3.11 Exemplo de *checkpoint tree* criada a partir de uma fila de prioridade. Fonte: O Autor.

Na Figura 3.11, tem-se a representação de uma *checkpoint tree* implementada sobre as filas de prioridade parcialmente retroativas. $Q_{[l,r]}$ representa que a fila de prioridade abrange as alterações realizadas no intervalo $[l,r]$ na linha temporal da estrutura. Nesse caso, tem-se uma linha temporal de tamanho 16, e está sendo realizada uma consulta na estrutura no tempo 11. Os elementos em verde correspondem aos intervalos de interesse na consulta desse intervalo, enquanto os nós folha correspondem às operações realizadas. Operações de inserção correspondem aos valores inseridos, enquanto a letra D corresponde às operações de remoção do elemento mínimo no tempo t da estrutura.

A Figura 3.12 mostra como cada um desses nós da árvore mantém as informações de operações realizadas em cada segmento. Cada nó contém dois conjuntos, Q_{now} e Q_{del}, correspondente aos elementos presentes na fila de prioridade e aos elementos que foram removidos da fila de prioridade naquele intervalo. Portanto, para obter o intervalo que abrange a consulta no tempo $t = 11$ em uma fila de prioridade totalmente retroativa, basta encontrar $Q_{[1,8]} \cup Q_{[9,10]} \cup Q_{[11,11]}$.

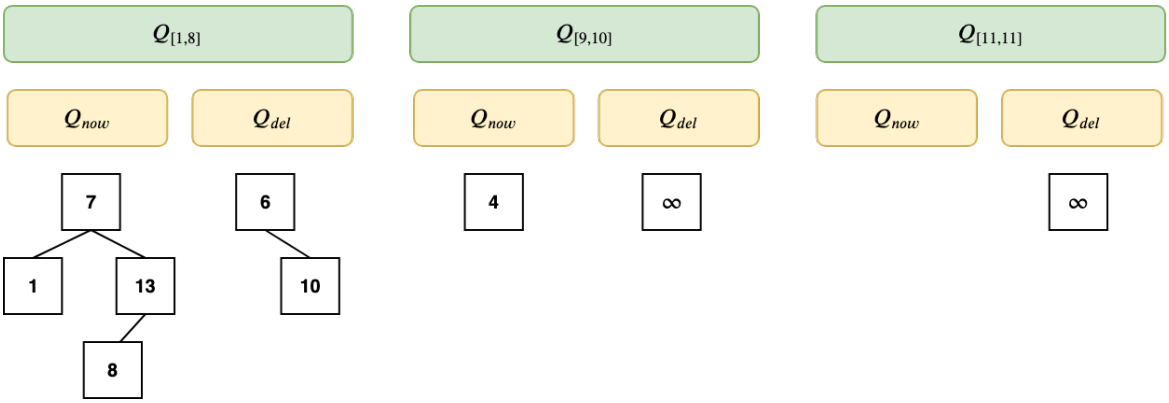

Figura 3.12 Exemplo de consulta em uma *checkpoint tree*. Fonte: O Autor.

Algoritmo para unir duas filas de prioridade parcialmente retroativas

Considere duas filas de prioridade Q_1 e Q_2 que são *time-fusibles* entre si (ou seja, com intervalos $[l_1, r_1]$ e $[l_2, r_2]$ tal que $r_1 + 1 = l_2$). Então, a estrutura Q_3 é gerada a partir da união das estruturas Q_1 e Q_2, abrangendo o intervalo $[l_1, r_2]$, pode ser descrita por:

$$Q_{3,now} = Q_{2,now} \cup \max -A\{Q_{1,now} \cup Q_{2,del}\}$$

$$Q_{3,del} = Q_{1,del} \cup \min -D\{Q_{1,now} \cup Q_{2,del}\}$$

onde $A = |Q_{1,now} \cup Q_{2,del}| - |Q_{2,del}|$, $D = |Q_{2,del}|$ e $\max -C\{S\}$ denota os C maiores elementos em S, assim como $\min -C\{S\}$ representa os C menores elementos no conjunto S.

Observe que esses conjuntos só serão unidos nas consultas, uma vez que só se quer restaurar um intervalo da fila de prioridade correspondente ao período de tempo $[-\infty, t]$ nesse tipo de operação. Definidos os conjuntos, pode-se escrever o Algoritmo 24 para a representação da união dessas filas de prioridade parcialmente retroativas. Nesse Algoritmo, $getSplitKey(D, T)$ retorna um valor x em que todos os sub-conjuntos de T devem ser divididos de modo que a quantidade de valores menores ou iguais a x seja igual a D. Essa função pode ser implementada em tempo logarítmico no tamanho do intervalo dos elementos resultantes da união de $Q_{1,now}$ e $Q_{2,del}$ por meio de uma busca binária.

Algoritmo 24 Algoritmo para a união de duas filas de prioridade parcialmente retroativas com intervalos contíguos

1: **função** MERGE(Q_1, Q_2)
2: $D \leftarrow |Q_{2,del}|$
3: $T \leftarrow \{Q_{1,now} \cup Q_{2,del}\}$
4: $x \leftarrow getSplitKey(D, T)$
5: $Q_{3,now} \leftarrow Q_{2,now} \cup T_{>x}$
6: $Q_{3,del} \leftarrow Q_{1,del} \cup T_{\leq x}$
7: retorna Q_3
8: **fim função**

É necessária uma estrutura de dados que seja capaz de armazenar esses conjuntos, além de dividir um dado conjunto T em dois sub-conjuntos T_1 e T_2, em que $T_1 = T_{\leq x}$ e $T_2 = T_{>x}$ para um dado valor x qualquer. Pode-se utilizar uma árvore binária balanceada para manter os conjuntos Q_{now} e Q_{del}, e assim, realizar a divisão descrita na altura da árvore.

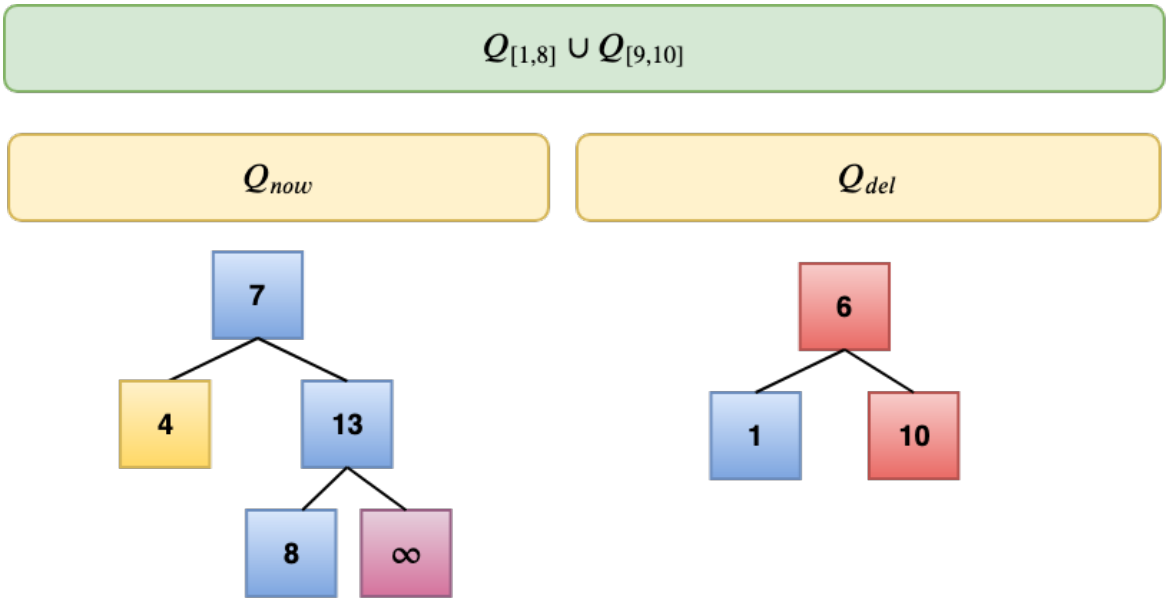

Figura 3.13 Exemplo de união de dois nós em uma *checkpoint tree*. Fonte: O Autor.

Na Figura 3.13, tem-se a representação visual da união dos dois primeiros intervalos da consulta no tempo 11 relacionada à fila de prioridade apresentada na Figura 3.11. Os elementos em azul e vermelho correspondem respectivamente, às árvores Q_{now} e Q_{del} do segmento $Q_{[1,8]}$, enquanto os elementos coloridos em amarelo e roxo correspondem a essas estruturas relacionadas a $Q_{[9,10]}$. Os elementos em azul e roxo são unidos pela operação

$\{Q_{1,now} \cup Q_{2,del}\}$, sendo que os menores $D = |Q_{2,del}|$ deles são colocados no novo conjunto Q_{del}, enquanto o restante desses elementos são inseridos em Q_{now}.

Os elementos em $Q_{1,del}$ e $Q_{2,now}$ (em amarelo e vermelho, respectivamente) não são afetados por essa operação de união, uma vez que todos os elementos do conjunto $Q_{1,del}$ não serão recuperados na união desses segmentos. De maneira similar, todo elemento que está na fila de prioridade em $Q_{2,now}$ não poderá ser removido por uma operação de remoção ocorrida em um tempo anterior à sua inserção.

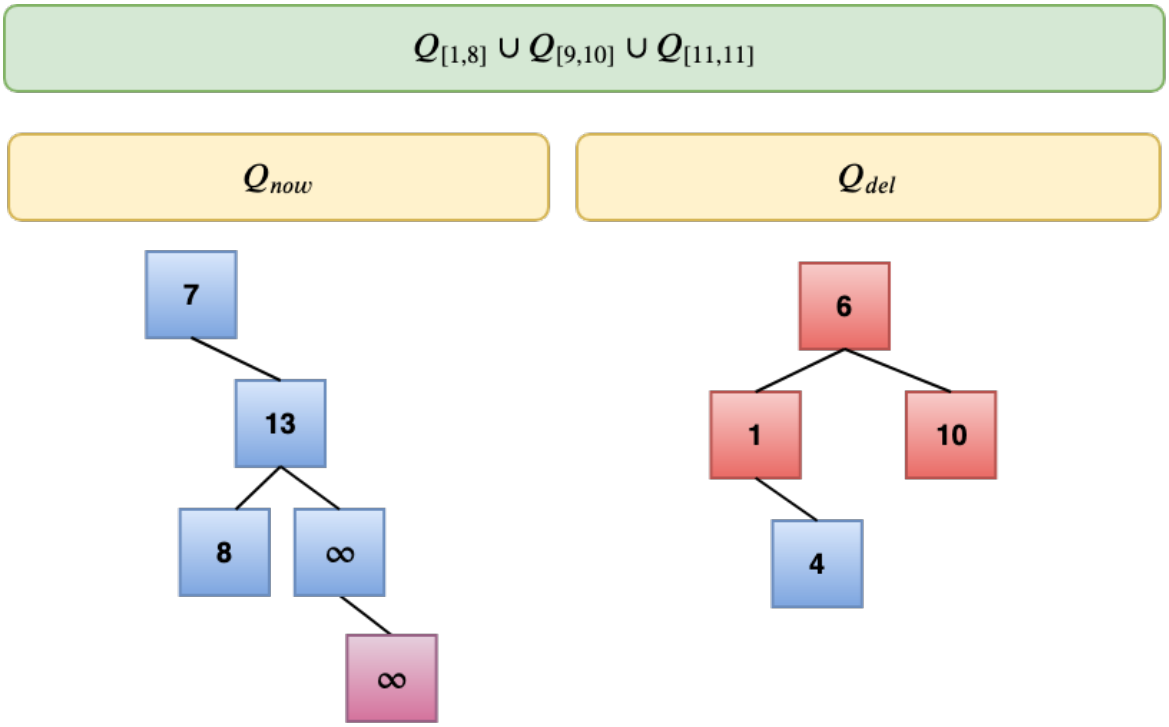

Figura 3.14 Exemplo de união de dois nós em uma *checkpoint tree* (continuação). Fonte: O Autor.

Na Figura 3.14, é realizada a mesma operação, porém, levando em consideração que $Q_{[1,8]}$ e $Q_{[9,10]}$ já foram unidas pela operação anterior. Agora, os blocos em azul e vermelho correspondem aos elementos de Q_{now} e Q_{del} correspondentes ao intervalo $\{Q_{1,now} \cup Q_{2,del}\}$, enquanto os elementos em amarelo e roxo correspondem ao segmento $Q_{[11,11]}$. Assim, obter o elemento mínimo do intervalo $Q_{1,11}$ é equivalente à obter o menor valor de Q_{now} que, após essas uniões, pode ser obtido em tempo logarítmico no número de elementos do conjunto.

Utilizando a implementação de uma árvore binária de busca sem nenhuma modificação para cada conjunto da estrutura, perde-se informação sobre a árvore anterior após a realização de uma divisão por um valor x. Uma possível solução seria a cópia da árvore binária inteira, e, somente após essa operação, realizar a divisão. Porém, essa solução consome tempo linear

no número de elementos da árvore. Para solucionar esse problema, pode-se utilizar a versão totalmente persistente de uma árvore binária balanceada, em que uma modificação em uma árvore cria não uma cópia, mas uma nova versão da árvore, deixando de copiar vértices não modificados por essa divisão.

3.3.4 *Cartesian Tree* totalmente persistente

Para as consultas na implementação da fila de prioridade totalmente retroativa em tempo polilogarítmico, é necessária a construção de uma estrutura que permita a manutenção de várias versões de uma árvore binária balanceada. Existem alguns métodos para a transformação de uma estrutura para a sua versão persistente.

Assim como para a implementação das estruturas anteriores, foi empregada a árvore binária de busca *cartesian tree* [44]. Porém, nessa versão da fila de prioridade totalmente retroativa, será utilizada à versão persistente dessa estrutura. Nessa implementação, utilizou-se a técnica de *path-copying* [18] para a obtenção da versão persistente. Ou seja, para cada nó da raiz até um nó que contenha uma modificação da estrutura, cria-se um novo nó copiando-o e fazendo as modificações nesse novo nó criado.

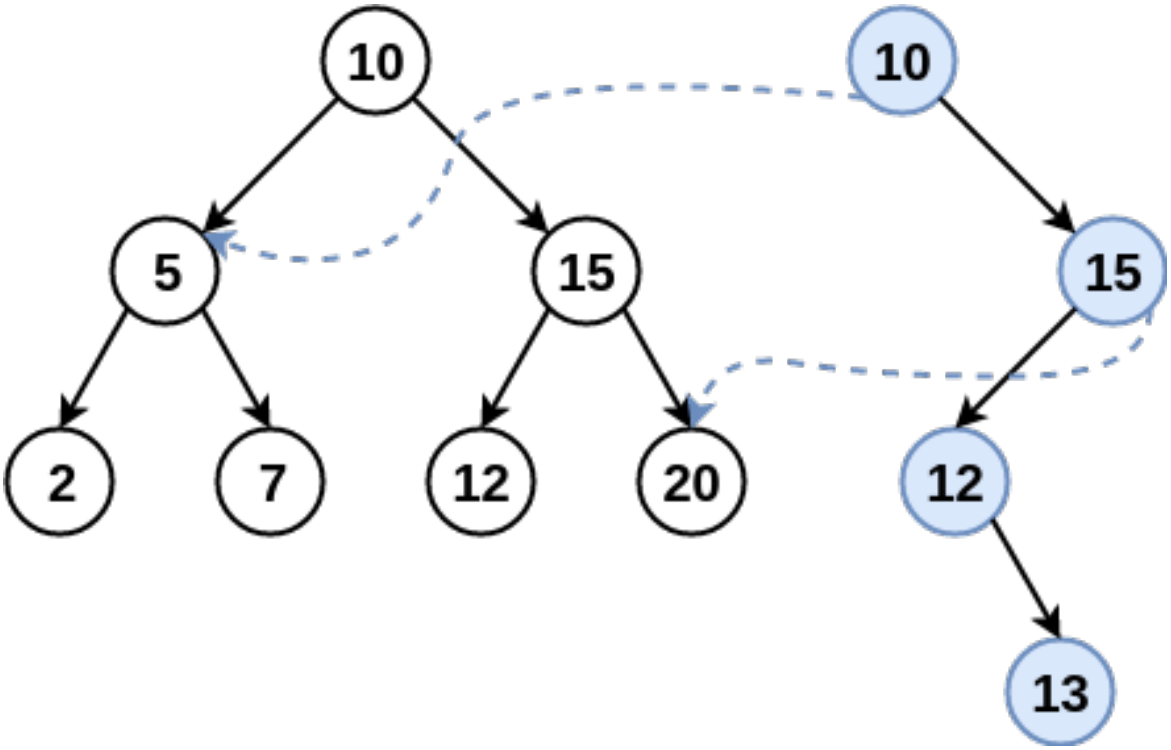

Figura 3.15 Exemplo de persistência por *path-copying* em uma ABB. Fonte: O Autor.

Na Figura 3.15, tem-se a representação visual de uma operação de inserção de uma árvore binária. No exemplo, é inserido o elemento 13 na árvore. Inicia-se a operação como uma inserção comum em uma árvore binária. Ou seja, se o valor x inserido for menor que o valor do nó atual, caminha-se na árvore para a esquerda, e, em caso contrário, para a direita, repetindo a operação enquanto o nó atual não for nulo.

No exemplo, uma inserção em uma árvore binária comum percorreria o caminho $\{10, 15, 12\}$ para a inserção do elemento 13. Na versão persistente, a inserção percorre o mesmo caminho, porém, esses nós são copiados para a criação da nova versão. Ao copiar-se um nó, além do valor, copia-se também os ponteiros para os filhos da esquerda e da direita desse nó. Por exemplo, na Figura 3.15, inicialmente o nó 10 é copiado, e os ponteiros da esquerda e da direita desse novo nó apontam para $\{5, 15\}$, respectivamente (aresta em azul tracejada). Após a análise do valor do nó, o algoritmo chama recursivamente a função de inserção para direita, criando o nó 15 em azul (aresta em preto). Observe que uma das arestas tracejadas em azul criadas pela execução da operação de cópia foi atualizada para uma de cor preta pela recursividade da operação de inserção. Em outras palavras, todas as arestas em azul pontilhadas correspondem às arestas entre versões distintas da estrutura, enquanto as arestas em preto correspondem às ligações entre os nós criados na mesma versão.

Utilizando essa técnica, é possível gerar uma nova versão da estrutura percorrendo somente a altura da árvore binária.

Algoritmo 25 Função para cópia de um nó

```
1: função COPY(p)
2:     pTreapNode cpy ← new TreapNode(p_key, p_data)
3:     cpy_l = p_l
4:     cpy_r = p_r
5:     retorna update(cpy)
6: fim função
```

No Algoritmo 25, tem-se a implementação da função de cópia de um nó p. Em uma cópia, todos os atributos desse nó são copiados, inclusive os ponteiros para os filhos da esquerda e da direita desse nó. A função $update(p)$ atualiza e retorna informações sobre o máximo, o mínimo, e o tamanho da subárvore enraizada em p.

Algoritmo 26 Operação de divisão de uma árvore binária auto-balanceada persistente

```
1:  função SPLIT(t, key, a, b)
2:      se t == NULL então
3:          a ← b ← NULL
4:          retorna;
5:      fim se
6:      pTreapNode aux;
7:      t ← copy(t)
8:      se key < t_key então
9:          split(t_l, key, a, aux)
10:         t_l ← aux
11:         b ← update(t)
12:     senão
13:         split(t_r, key, aux, b)
14:         t_r ← aux
15:         a ← update(t)
16:     fim se
17:     retorna
18: fim função
```

No Algoritmo 26, tem-se a implementação da função de divisão da árvore binária t por um valor key, gerando as árvores $a_{\leq key}$ e $b_{>key}$. Portanto, se a árvore atual for nula, implica que a função de divisão chegou ao final, e que as duas árvores geradas da divisão da árvore vazia também são vazias. Após essa verificação, o nó atual com certeza será modificado pela divisão, e, portanto, na linha 7, o nó em questão é copiado.

Na Figura 3.16, tem-se a visualização da execução de uma operação de divisão de uma árvore enraizada em t por um valor x. No caso exemplificado, o valor x é maior que o valor da raiz da árvore, e, por esse motivo, essa imagem se refere às operações realizadas entre as linhas 13 a 16 do Algoritmo 26. Inicialmente, a função de divisão é chamada recursivamente para o filho da direita da árvore, retornando dois ponteiros aux e b. Os valores que estão em aux são menores que o valor x da divisão da árvore, porém, maior que a raiz da árvore atual, e, portanto, aux se torna o filho direita da árvore. A subárvore b já está corretamente dividida pela recursividade do algoritmo. Finalmente, atribui-se a com a subárvore t após o corte, e assim, a contém todos os valores menores ou iguais a x, enquanto b contém o restante dos valores. De maneira simétrica, tem-se o algoritmo quando o valor x é menor que o valor do nó raiz da subárvore atual.

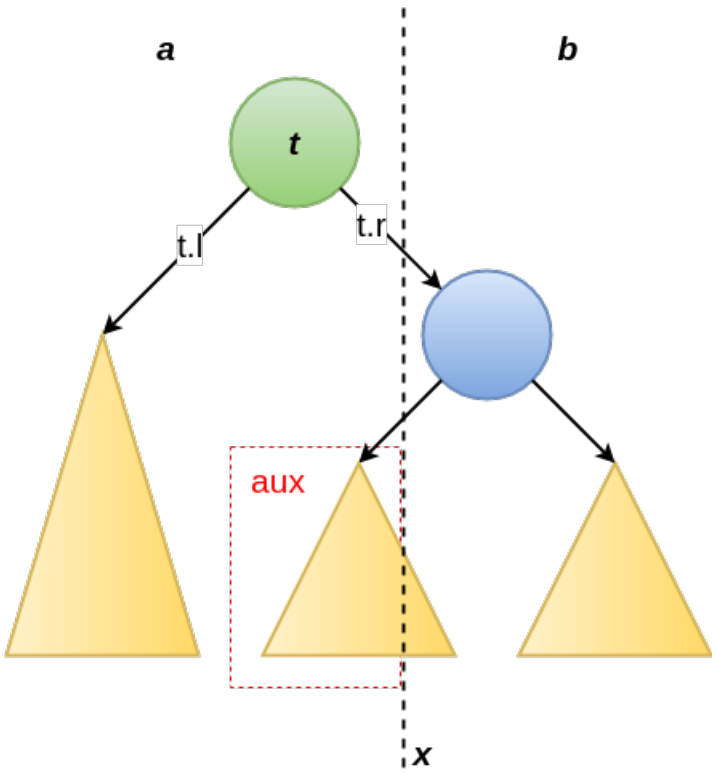

Figura 3.16 Exemplo de uma divisão por um valor x em uma árvore binária. Fonte: O Autor.

Algoritmo 27 Operação de inserção em uma *cartesian-tree* persistente

```
1:  função INSERT(t, it)
2:      t ← copy(t)
3:      se t == NULL então
4:          t ← it
5:      senão se it_y < t_y então
6:          split(t, it_key, it_l, it_r)
7:          t ← it
8:      senão
9:          se it_key < t_key então
10:             t_l ← insert(t_l, it)
11:         senão
12:             t_r = insert(t_r, it)
13:         fim se
14:     fim se
15:     retorna update(t)
16: fim função
```

De posse da operação de divisão de uma árvore binária por um valor x, é possível implementar a função de inserção, como mostrado no Algoritmo 27. Em uma *cartesian-tree*, cada elemento possui, além de seu par chave/valor, uma variável que representará seu balanceamento. Ela é também comumente chamada *Treap* $(tree + heap \rightarrow Treap)$, pois combina as propriedades de busca de uma árvore binária com uma *heap* binária. Em uma *heap* binária, todo elemento somente contém elementos que sejam menores que ele em sua subárvore. Essa condição deve ser mantida para que a propriedade de *heap* se satisfaça.

No Algoritmo 27 essa condição é mantida pela variável auxiliar y, que mantém o balanceamento da árvore através dessa propriedade. Em outras palavras, seja (x, y) um par relacionado a um nó correspondente ao valor x na árvore binária, em que deseja-se manter a *heap* formada pelos valores relacionados a y.

Como se trata de uma *cartesian-tree* persistente, o primeiro passo consiste em copiar os nós do caminho entre a raiz e o elemento inserido. Na linha 3, tem-se o caso base em que a raiz está nula, e, portanto, a raiz da árvore atual é justamente o elemento atual. Após essa verificação, trata-se a propriedade de *heap* da árvore. Ou seja, se a prioridade do elemento inserido atualmente for menor que a prioridade da raiz atual da subárvore, isso significa que o elemento inserido atualmente não pode descer mais na árvore, pois isso não obedeceria as propriedades de uma *heap*. Portanto, divide-se a subárvore pela chave do nó inserido e define-se o nó inserido atual como raiz das duas árvores geradas pela divisão. Caso contrário, a inserção do novo elemento segue as operações normais de uma árvore binária comum.

Algoritmo 28 Operação de união de duas árvores binárias auto-balanceadas persistentes

```
1:  função MERGE(l, r)
2:      se l ou r == NULL então
3:          retorna l?l : r
4:      fim se
5:      m ← getSize(l)
6:      n ← getSize(r)
7:      se rand(m+n) < m então
8:          l ← copy(l)
9:          l_r ← merge(l_r, r)
10:         retorna update(l)
11:     senão
12:         r ← copy(r)
13:         r_l ← merge(l, r_l)
14:         retorna update(r)
15:     fim se
16: fim função
```

No Algoritmo 28, tem-se a implementação da função referente à união de duas árvores binárias auto-balanceáveis l e r, em que o maior valor em l é menor que o menor valor em r. Na união de duas dessas árvores, existem duas possibilidades:

- Utilizar o nó atual da árvore l como pai do nó atual da árvore r;
- Utilizar o nó atual da árvore r como pai do nó atual da árvore l.

Após a definição da política de união, chama-se recursivamente o algoritmo de união, enquanto l e r não são nulos. Porém, a definição de uma política estática para a união dos nós pode gerar uma árvore totalmente desbalanceada.

Em uma *cartesian tree* totalmente persistente, além da variável y utilizada para a manutenção da propriedade de *heap* da árvore, utiliza-se a probabilidade proporcional ao tamanho da árvore para a definição da política adotada na iteração. Portanto, duas execuções diferentes da união das mesmas árvores podem gerar resultados diferentes, contudo, mantendo uma altura logarítmica amortizada em ambos os casos. Na linha 7 do Algoritmo 28 tem-se a escolha probabilística da política, como apresentado.

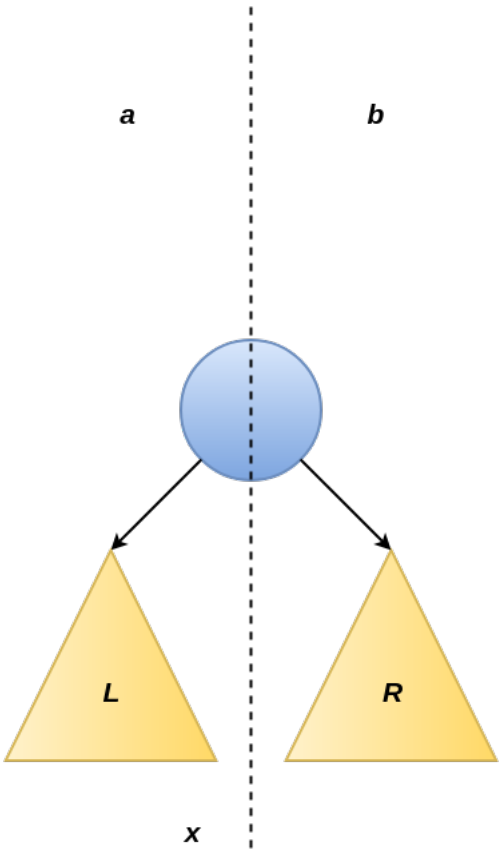

Figura 3.17 Exemplo de união de duas árvores binárias. Fonte: O Autor.

Na Figura 3.17, tem-se a visualização da união de duas árvores L e R, em que o maior dos valores em L é menor ou igual a x, enquanto o menor dos valores em R é maior que x. O nó em azul representa a raiz resultado da operação de união, que poderá ser um nó de L ou um nó de R, dependendo da distribuição probabilística proporcional ao tamanho das subárvores enraizadas em L e R, respectivamente. Quanto maior o número de nós de uma árvore, maior é a esperança de sua altura. Por essa razão o algoritmo tem mais chances de escolher como raiz a árvore com o maior número de nós.

Algoritmo 29 Operação de remoção em uma *cartesian-tree* persistente

```
função ERASE(t, key)
    t ← copy(t)
    se t_key == key então
        t ← merge(t_l, t_r)
    senão
        se key < t_key então
            t_l ← erase(t_l, key)
        senão
            t_r = erase(t_r, key)
        fim se
    fim se
    retorna update(t)
fim função
```

Por fim, a função de união de duas árvores permite a criação da função de remoção de um elemento em uma *cartesian-tree*. Novamente, pela natureza persistente da estrutura, é necessário que se copie todo o caminho até a remoção do nó, que é realizada pela verificação da linha 3 no Algoritmo 29.

Lema 11. *É possível gerar uma árvore binária totalmente persistente com complexidade temporal* $O(\lg n)$ *e com custo adicional em espaço de* $O(\lg n)$ *por operação.*

Demonstração. Como a implementação somente acessa os elementos do caminho entre a raiz e uma modificação, a complexidade temporal de uma operação nessa árvore é proporcional à sua altura.

Em uma *cartesian-tree*, a altura da árvore é equivalente a um algoritmo guloso para encontrar a maior subsequência crescente em um vetor de tamanho n com elementos distintos de 1 a n aleatoriamente dispostos. Um algoritmo guloso para encontrar essa subsequência consiste em pegar o próximo elemento que é maior que o atual. Em uma permutação aleatória, o tamanho esperado dessa subsequência, utilizando o algoritmo descrito, é $O(\lg n)$. □

3.3.5 Retroatividade não-consistente da fila de prioridade

Assim como em outras estruturas, existe a versão de uma fila de prioridade que retorna a primeira operação que se torna inconsistente após uma modificação em uma versão do passado da estrutura. É possível implementar essa versão em tempo logarítmico no número de operações por meio da utilização de árvores binárias balanceadas.

Para a implementação dessa estrutura, é viável a utilização da representação das operações em um plano cartesiano bidimensional. No modelo proposto por Acar *et. al* [53], cada elemento k na estrutura é representado por um segmento horizontal $([t_1,t_2],k)$, representando o espaço temporal em que esse elemento pertencerá a estrutura. Um segmento vertical $(t,[k_1,k_2])$ representa que uma consulta foi realizada no tempo t, partindo da posição k_1, e retornando a chave k_2 inserida na estrutura. Na estrutura apresentada, as consultas sempre serão realizadas com $k_1 = -\infty$. Pode-se assumir, sem perda de generalidade, que todas as operações e valores de elementos são distintos.

Então, serão mantidos dois conjuntos de segmentos: um conjunto H dos segmentos horizontais (representando a linha do tempo dos elementos inseridos); e um conjunto V dos segmentos verticais representando as consultas realizadas. O conjunto das retas horizontais será mantido em uma árvore de segmentos. Nessa estrutura, é possível consultar, dado um valor t, quais segmentos na estrutura que passam por t. Além disso, com essa estrutura, é possível obter o segmento de menor altura que passa pelo instante t, e, se essa estrutura for implementada sobre a linha do tempo gerada por uma fila de prioridade, o resultado da operação *GetPeak*.

Já o conjunto das retas verticais será mantido em uma *priority search tree (PST)* [45, 56]. Na estrutura proposta por McCreight, cada nó da árvore contém um ponto (x,y) que pode ser representado no espaço R^2. Para todo nó t dessa árvore, três condições devem ser respeitadas:

- $y(t) \geq y(esquerda(t))$: filho da esquerda de t tem o valor y menor ou igual ao valor de seu pai;
- $y(t) \geq y(direita(t))$: filho da direita de t tem o valor y menor ou igual ao valor de seu pai;
- Todo nó t contém um valor $X(t)$ em que todos os pontos com coordenada x menores ou iguais a $X(t)$ estão contidas no filho esquerdo de t, enquanto todos os pontos com coordenada x maiores que $X(t)$ estão contidos no filho direito de t.

Com essa árvore, é possível responder consultas como:

1. Encontrar, dado x_0, x_1 e y_1, um ponto (x,y) tal que $x_0 \leq x \leq x_1$ e $y \leq y_1$, de modo que x seja mínimo (ou máximo);
2. Encontrar, dado x_0, x_1, um ponto (x,y) tal que $x_0 \leq x \leq x_1$ e $y \leq y_1$, de modo que x seja mínimo;
3. Mostrar, dado x_0, x_1 e y_1, todos os pontos (x,y) tal que $x_0 \leq x \leq x_1$ e $y \leq y_1$.

McCreight [45] demonstrou que (1) e (2) são respondidas em tempo $O(\lg n)$, enquanto (3) é respondida utilizando $O(r+\lg n)$ de tempo, em que r é o número de pontos reportados pela execução da operação (3). Além disso, essa estrutura ocupa espaço $O(n)$.

Esses tipos de consulta são comumente chamadas de operações no espaço $R^{1.5}$, uma vez que é possível observá-las como uma consulta "retangular" em que um dos lados desse retângulo não existe.

Como o conjunto V será mantido em uma *PST*, deve-se definir o que será considerado para a geração de um ponto $p = (x,y)$. Nesse conjunto, será mantido, para cada operação de retirada realizada em uma fila de prioridade no tempo t, o valor v de um ponto $p = (t,v)$.

Teorema 1. *Após a realização da operação $Insert(t, Push(x))$, a próxima inconsistência será uma operação do conjunto V realizada no tempo t' tal que t' seja mínimo, $t' \geq t$ e que o elemento retornado por essa operação, digamos k, seja maior ou igual a x.*

Demonstração. Inicialmente, é inserido, no conjunto H, o elemento x no tempo t. Esse elemento pode ou não ser removido no futuro na linha temporal da estrutura. O objeto inserido será acessado pela primeira vez por uma operação realizada após o tempo t. Algumas das operações de remoção realizadas posteriormente ao tempo t podem remover elementos menores que x, e elas não serão inconsistentes pela execução da operação (vide Figura 3.18). Assim, somente deve-se considerar elementos removidos maiores ou iguais a x. A operação mostrada pelo teorema pode ser executada realizando a consulta (2) na árvore em tempo $O(\lg n)$. □

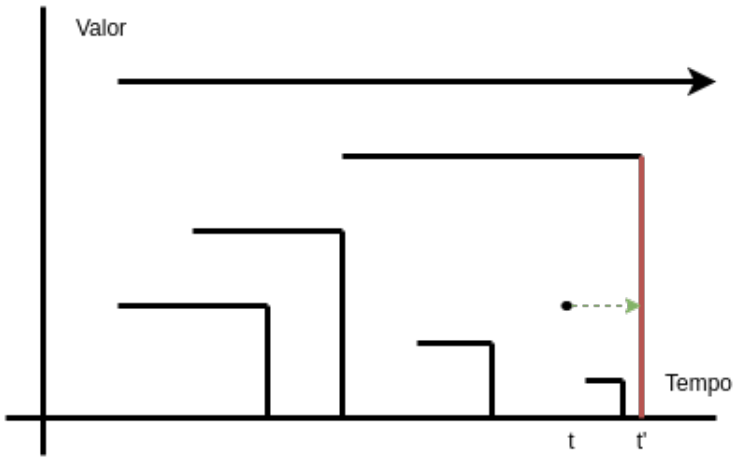

Figura 3.18 Exemplo de inconsistência causada pela operação $Insert(t, Push(x))$. Fonte: O Autor.

Na Figura 3.18 é mostrada a representação no plano cartesiano de uma fila de prioridade. Segmentos horizontais representam a inserção/remoção dos elementos nessa estrutura, en-

quanto segmentos verticais representam operações de retirada do menor elemento da fila. Em um segmento horizontal, o tempo de inserção de um valor é representado por seu início, enquanto a sua remoção é representada por seu fim. Na imagem, é inserido um ponto no tempo t que existirá na estrutura até o tempo t', sendo essa a primeira operação inconsistente.

Teorema 2. *Após a realização da operação $Insert(t, Pop())$, o valor retirado da fila por essa operação será o menor valor x tal que $t_{ins} \leq t$ e $t \leq t_{del}$. Além disso, a próxima inconsistência será o elemento do conjunto V que contenha o menor t' tal que $t' \geq t$, e que o elemento retornado por essa operação, digamos k, seja maior ou igual a x.*

Demonstração. O valor da chave que será removido por essa operação pode ser obtido através de uma consulta na árvore de segmentos em que o conjunto H está implementado. Dentre todos os segmentos que contém o tempo t, o segmento que será afetado pela operação será o de menor altura. Seja s o segmento $s = ([t_{ins}, t_{del}], x)$, representando que o elemento x existiu na estrutura pelo intervalo temporal $[t_{ins}, t_{del}]$. Logo, pela definição da consulta, o segmento s deverá ser cortado, pois a operação Pop no tempo t remove o elemento contido em s. Com isso, é possível definir qual elemento foi retirado por essa operação e reajustar o segmento que representa a existência desse elemento. Finalmente, é necessário que se reporte qual a próxima operação inconsistente na estrutura, que consiste no primeiro tempo t' que exista uma operação no conjunto V em que o elemento retornado seja maior ou igual ao elemento retirado pela operação $Insert(t, Pop())$. □

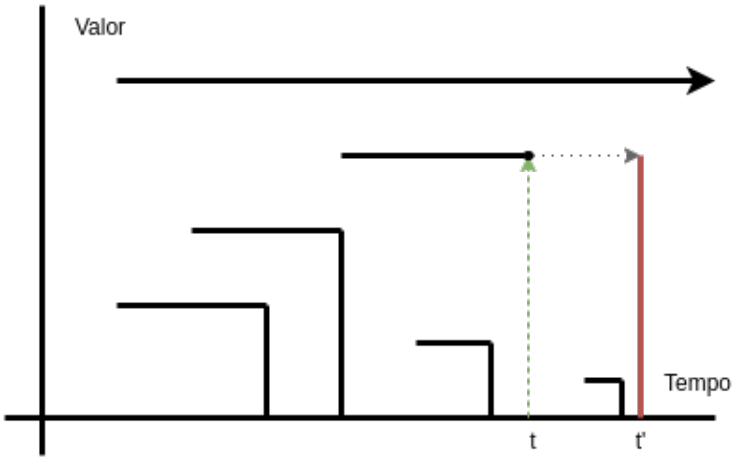

Figura 3.19 Exemplo de inconsistência causada pela operação $Insert(t, Pop())$. Fonte: O Autor.

Na Figura 3.19, é possível visualizar o efeito causado pela inserção de uma operação Pop no tempo t. Essa operação gera um segmento que se choca com o elemento de menor valor

que existe na estrutura no tempo t. A linha temporal desse elemento, que era removido pela operação *Pop* no tempo t', é cortada e passa a ter seu fim no tempo t, deixando a operação realizada em t' inconsistente.

Teorema 3. *Após a realização da operação $Delete(t, Push(x))$, a próxima inconsistência será a primeira operação do conjunto V no tempo t' tal que $t' \geq t$ e que o elemento retornado por essa operação, digamos k, seja igual a x.*

Demonstração. Após a remoção da operação realizada no tempo t, todas as operações de remoção/consulta que antes tocavam no segmento que representava a linha temporal do elemento x se tornarão inconsistentes. Caso existam várias dessas operações, somente a primeira delas será reportada. Se não existirem operações realizadas nesse segmento, não haverá operação inconsistente na estrutura. □

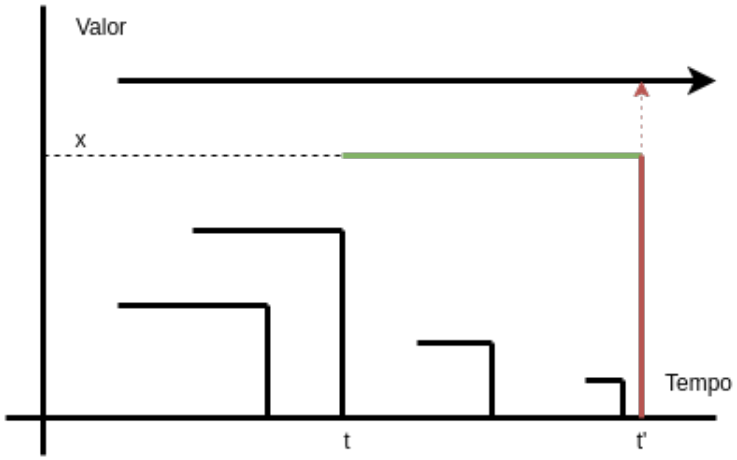

Figura 3.20 Exemplo de inconsistência causada pela operação $Delete(t, Push(x))$. Fonte: O Autor.

Na Figura 3.20, tem-se a ilustração que representa o efeito causado na representação geométrica pela deleção de uma operação *Push* no tempo t. A operação que antes removia o elemento, x no tempo t', agora remove o menor elemento em t' após a deleção da inserção de x.

Teorema 4. *Após a realização da operação $Delete(t, Pop())$, a próxima inconsistência será a primeira operação do conjunto V no tempo t' tal que $t' \geq t$, e que o elemento retornado por essa operação, digamos k, seja maior ou igual a x após a remoção da operação realizada no tempo t do conjunto V.*

Demonstração. Como V está implementado sobre uma *PST*, é possível consultar o primeiro tempo t' que contenha uma operação inconsistente após a remoção da operação Pop. Além disso, é necessário atualizar o tamanho do segmento horizontal que corresponde ao elemento removido por essa operação, uma vez que esse elemento existirá até a próxima operação inconsistente. □

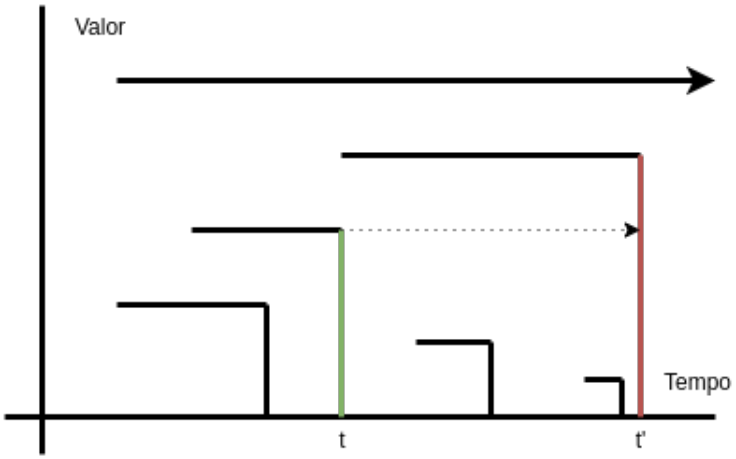

Figura 3.21 Exemplo de inconsistência causada pela operação $Delete(t, Pop())$. Fonte: O Autor.

A Figura 3.21 mostra o efeito da execução de uma operação $Delete(t, Pop())$ na fila de prioridade não consistente. O segmento em verde corresponde à operação Pop retirada da estrutura, enquanto o segmento em vermelho consiste na próxima operação inconsistente.

Com os conjuntos V e H, é possível obter a fila de prioridade não-consistente, mantendo as representações da linha do tempo dos elementos na estrutura, de modo que seja possível a atualização dessas representações após uma modificação no passado.

O Algoritmo 30 apresenta a implementação em C++ na função para inserção da operação $Push(x)$ no tempo t na estrutura. Na função, primeiramente, é inserido o segmento $([t, \infty], x)$ no conjunto H, que representa a linha temporal do elemento x na estrutura. Esse elemento inicialmente tem a linha temporal infinita, uma vez que ainda está inconsistente na estrutura. Então, é retornada a primeira operação do conjunto V inconsistente, o elemento que contém a menor coordenada no eixo das abscissas em V que seja maior que t e com $y \geq x$.

Algoritmo 30 Função para a inserção da operação $Insert(t, Push(x))$ em uma fila de prioridade retroativa não-consistente

```
1: função INSERTPUSH(t, x)
2:     H.insertSegment(t, ∞, x)
3:     retorna V.getMinXGreaterThanOrEqualY(t, ∞, x)
4: fim função
```

Já no Algoritmo 31, tem-se a implementação que retorna a primeira operação inconsistente após a realização da inserção de uma operação de retirada na fila no tempo t. Inicialmente, é encontrado o elemento que será retirado pela inserção da operação *Pop* no tempo t. Esse elemento é o menor dentre todos os elementos que existem no tempo t, ou seja, o segmento em H que contém a menor altura. Após a obtenção desse elemento, atualiza-se o segmento desse objeto a fim de atualizar o seu término, pois esse elemento será agora removido pela operação *Pop* inserida. Logo, a próxima inconsistência ocorrerá na operação em V que removia o elemento x, que consiste no instante de tempo t_{del} no algoritmo.

Algoritmo 31 Função para a inserção da operação $Insert(t, Pop())$ em uma fila de prioridade retroativa não-consistente

```
1: função INSERTPOP(t)
2:     x ← H.getMinY(t)
3:     t_ins ← H.getStartTimeFromKey(x)
4:     t_del ← H.getFinishTimeFromKey(x)
5:     H.eraseSegment(t_ins, t_del, x)
6:     new_tdel ← t
7:     H.insertSegment(tins, new_tdel, x)
8:     V.insertPoint(t, x)
9:     retorna t_del
10: fim função
```

Algoritmo 32 Função para a deleção da operação $Push(x)$ realizada no tempo t em uma fila de prioridade retroativa não-consistente

1: **função** DELETEPUSH(t)
2: $t_{ins} \leftarrow t$
3: $t_{del} \leftarrow H.getFinishTimeFromStartTime(t_{ins})$
4: $x \leftarrow H.getKeyFromStartTime(t_{ins})$
5: $H.eraseSegment(t_{ins}, t_{del}, x)$
6: **retorna** t_{del}
7: **fim função**

No Algoritmo 32 tem-se a implementação da função que retorna a primeira inconsistência após a realização da operação $Delete(t, Push(x))$. Primeiramente, são obtidas as informações sobre o segmento que contém o início no tempo t, bem como qual elemento foi inserido no tempo t. O elemento x existe na linha do tempo da estrutura no intervalo $[t_{ins}, t_{del}]$, e não existirá mais. Portanto, remove-se do conjunto H esse segmento. Logo, a operação que removia x se torna inconsistente, e, por isso, t_{del} é retornado como inconsistente.

Finalmente, o Algoritmo 33 mostra como realizar a deleção da operação Pop, efetuada no tempo t. Inicialmente, corrige-se o segmento horizontal afetado pela operação Pop. O elemento passa a existir pela linha do tempo inteira, uma vez que antes da correção das operações de remoção não existe uma operação Pop que afete esse elemento. Portanto, x deixa de existir somente em $[t_{ins}, t_{del}]$ e passa a existir no intervalo $[t_{ins}, \infty]$. Em seguida, é removido o segmento vertical que representa a operação no conjunto V, e, posteriormente, retornada como inconsistente a próxima operação do conjunto V que tenha afetado um elemento maior ou igual a x.

Algoritmo 33 Função para a deleção da operação $Pop()$ realizada no tempo t em uma fila de prioridade retroativa não-consistente

1: **função** DELETEPOP(t)
2: $x \leftarrow V.getYFromX(t)$
3: $t_{ins} \leftarrow H.getStartTimeFromKey(x)$
4: $t_{del} \leftarrow H.getFinishTimeFromStartTime(t_{ins})$
5: $H.eraseSegment(t_{ins}, t_{del}, x)$
6: $H.insertSegment(t, \infty, x)$
7: $V.erasePoint(t, x)$
8: **retorna** $V.getMinXGreaterThanOrEqualY(t, \infty, x)$
9: **fim função**

3.4 Union-find

Union-find é uma estrutura que permite ao programador ter controle sobre alguns conjuntos disjuntos, e que traz funções que proporcionam a união desses conjuntos em tempo sub-logarítmico por operação no número de conjuntos [12].

As operações principais dessa estrutura de dados são:

- *MakeSet(n)*: cria uma família de n conjuntos independentes entre si;
- *UnionSet(u, v)*: une dois conjuntos que contém os elementos u e v;
- *SameSet(u, v)*: retorna se dois elementos u e v estão contidos no mesmo conjunto.

Essas funções podem ser usadas em uma série de problemas, como a solução proposta por Kruskal em 1957 [41] para o problema da árvore geradora mínima.

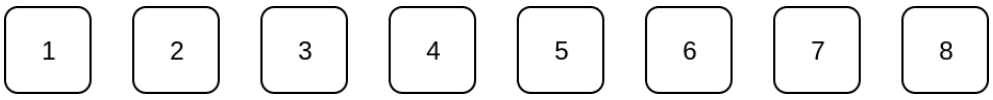

Figura 3.22 Resultado dos conjuntos após execução de *MakeSet(8)*. Fonte: O Autor.

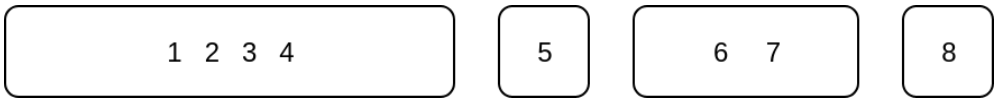

Figura 3.23 Resultado dos conjuntos após execução de algumas chamadas da função *UnionSet*. Fonte: O Autor.

Na Figura 3.22, tem-se a representação da chamada da função *MakeSet(8)*, que cria oito conjuntos distintos. Na Figura 3.23, tem-se a execução de algumas chamadas da função *UnionSet(x, y)*, que são, respectivamente, *UnionSet(1, 2)*, *UnionSet(1, 3)*, *UnionSet(1, 4)*, *UnionSet(6, 7)*.

Para a versão retroativa para o *union-find*, serão utilizadas as seguintes modificações das funções para a estrutura estática, definidas a seguir:

- $Insert(t, UnionSet(u,v))$: une dois conjuntos que contém os elementos u e v no tempo t;
- $SameSet(u,v,t)$: retorna se dois conjuntos u e v estão conectados no tempo t.

Lema 12. *Existe uma versão totalmente retroativa para a implementação de uma estrutura para união de conjuntos (union-find) que consome tempo* $O(\lg n)$.

Demonstração. Suponha, sem perda de generalidade, que dois conjuntos u e v nunca serão unidos mais de uma vez entre si. Pode-se modelar o problema como um grafo no qual cada vértice é um conjunto unitário, e cada aresta entre dois desses vértices corresponde a uma união entre dois conjuntos distintos. Seja G o grafo gerado pela execução de uma sequência de operações da estrutura. Com isso, a cada união desses conjuntos de vértices no tempo t, será adicionada uma nova aresta $e = \{u, v\}$ em G com custo $C(e)$. Para obter $SameSet(u, v, t)$, basta observar o valor das arestas no caminho entre u e v. Se houver um caminho entre u e v em que o custo de nenhuma aresta exceda t, u e v estão contidos no mesmo conjunto. Esse caminho que minimiza a aresta com maior custo no caminho entre todos os pares de vértices sempre estará contido em uma das árvores geradoras mínimas de G.

Utilizando essa fato, é possível utilizar a estrutura *Link-cut tree* [51], estrutura na qual é possível manter a árvore geradora mínima de G, de maneira dinâmica, bem como consultar o mínimo de um caminho entre u e v em G consumindo tempo $O(\lg n)$ por operação. □

No Algoritmo 34, tem-se a implementação de um *union-find* totalmente retroativo. Ambas as funções utilizam a estrutura auxiliar *Link-cut tree* para verificação tanto da conectividade entre dois vértices quanto para a obtenção do maior elemento no caminho entre dois nós distintos.

Algoritmo 34 Operações em um *union-find* totalmente retroativo

```
1:  LinkCutTree lct;
2:
3:  função UNIONSET(u, v, t)
4:      lct.atualizaVertice(k, t)
5:      se não lct.conectado(u, v) então
6:          lct.link(u, k)
7:          lct.link(v, k)
8:          k++
9:      senão
10:         id ← lct.maiorVertice(u, v)
11:         c ← lct.tempoVertice(id)
12:         se c > t então
13:             lct.cut(u, id)
14:             lct.cut(v, id)
15:             lct.link(u, k)
16:             lct.link(v, k)
17:             k++
18:         fim se
19:     fim se
20: fim função
21:
22: função SAMESET(u, v, t)
23:     se nao lct.conectado(u, v) então
24:         Retorna falso;
25:     senão
26:         Retorna lct.maiorVertice(u, v) ≤ t
27:     fim se
28: fim função
```

No Algoritmo 34, considera-se uma conversão das arestas em vértices, retirando a necessidade de modificação da estrutura de dados para a obtenção da aresta de maior tempo em G. Ou seja, toda aresta e entre dois vértices u e v com custo $C(e) = t$ é transformada em um vértice k com custo associado $C(k) = t$, e com arestas $\{v,k\}$ e $\{u,k\}$. Na Figura 4.4 é apresentada a forma na qual o grafo original é convertido em um grafo para ser trabalhado em uma *Link-cut tree*.

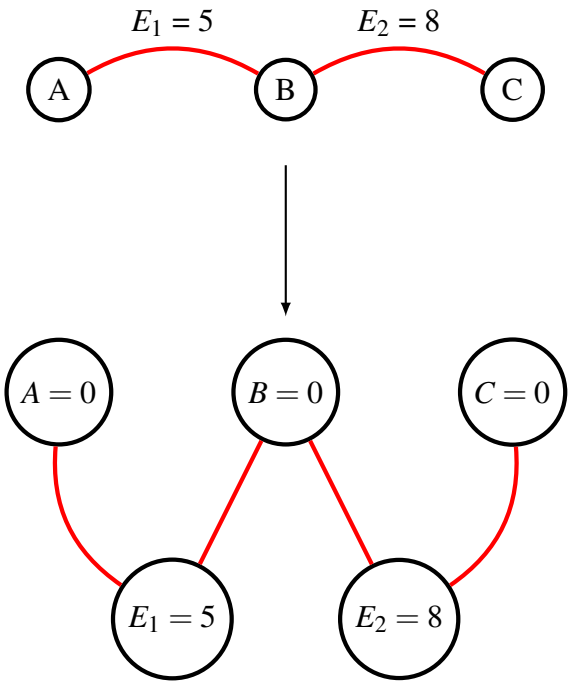

Figura 3.24 Exemplo de conversão do grafo. Fonte: O autor

Inicialmente, tem-se o grafo original com três vértices e duas arestas ligando os respectivos vértices. No exemplo, as arestas foram nomeadas E_1 e E_2 e adicionadas nos tempos cinco e oito, respectivamente. Na estrutura de dados *Link-cut tree* considera-se cada vértice com um peso agregado e cria-se um vértice para cada aresta, em que cada novo vértice representa uma aresta e com um custo agregado $C(e)$. Com isso, encontrar a aresta de custo máximo entre dois vértices consiste em encontrar o vértice de custo máximo entre eles.

Sem perda de generalidade, suponha que todas as arestas são adicionadas em tempos distintos não nulos. No Algoritmo 34, a função $UnionSet(u,v,t))$ é dividida em dois casos:

- Se os vértices u e v não estão na mesma componente conexa, simplesmente cria-se o vértice k com valor t, que representará a aresta entre u e v, e adicionam-se conexões dos vértices u e v com o vértice k;

- Senão, procura-se pelo vértice de maior tempo entre u e v (função *maiorVertice(u, v)*). Como todos os nós que representam vértices reais do grafo têm valor zero, essa função sempre retornará um vértice que representa uma aresta. Seja x o vértice encontrado. Então, se esse vértice conter um valor maior que t, deve-se remover x do grafo e usar a aresta atual para manter a conexidade entre u e v, otimizando o uso dos tempos de modo a manter uma árvore na qual o valor da maior aresta seja o menor possível.

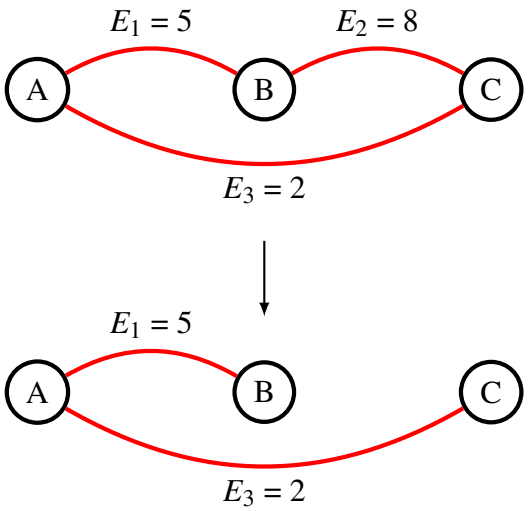

Figura 3.25 Exemplo da otimização do grafo final. Fonte: O autor

Na Figura 3.25, tem-se o exemplo de uma otimização para um grafo. Após a inserção de uma aresta que cria um ciclo no grafo que representa os conjuntos, pode-se remover a aresta de maior tempo contida nesse ciclo. Isso é sempre possível, pois os caminhos em que o maior tempo é o menor possível sempre estão contidos na árvore geradora mínima do grafo. No Algoritmo 34, a implementação da identificação da aresta de maior custo do ciclo e a sua possível remoção pode ser encontrada entre as linhas 12 a 18. Portanto, utilizando a *Link-cut tree* para armazenar a árvore geradora mínima do grafo de maneira dinâmica, é possível obter a estrutura de união de conjuntos totalmente retroativa em tempo logarítmico no tamanho da estrutura por operação de atualização ou consulta.

Capítulo 4

Dinamização de algoritmos

Este capítulo apresenta uma breve introdução sobre alguns problemas recorrentes em computação. Posteriormente, apresenta-se uma explicação detalhada sobre a dinamização desses algoritmos, através de técnicas utilizadas em estruturas de dados retroativas, fornecendo embasamento teórico de como otimizá-los.

4.1 Árvore geradora mínima

Seja G um grafo valorado e não direcionado com pesos nas arestas. Uma árvore geradora mínima (*MST*, do inglês *minimum spanning tree*) é um dos possíveis subgrafos acíclicos G' de G cuja a soma das arestas de G' é a menor possível [12]. Como G' é um grafo conexo acíclico com n vértices, o número de arestas em G' é $n-1$, e, portanto, G' é uma árvore, o que dá nome ao problema.

Na Figura 4.1, tem-se o exemplo de uma das possíveis árvores geradoras mínimas do grafo apresentado. Os números acima de cara aresta representam o custo de utilizá-la, e as arestas em vermelho representam as arestas de uma das possíveis soluções ótimas. É possível gerar outra árvore geradora mínima desse grafo removendo a aresta (a,b) ou (b,e) e adicionando a aresta (e,f).

Existem aplicações que usam a árvore geradora mínima como parte do pré-processamento, como, por exemplo, em algoritmos relacionados ao problema do caixeiro viajante [4, 10], registro e segmentação de imagem [26], regionalização de áreas sócio geográficas (agrupando-as como áreas homogêneas, regiões contínuas) [58, 2], comparação de dados ecotoxicológicos [5], *design* de circuitos [11], redes de sensores *wireless* [55, 37, 33] e em teoria dos jogos [27].

Christofides [10] propôs um algoritmo de aproximação para o problema do caixeiro viajante métrico por meio de um preprocessamento relacionado à árvore geradora mínima. O

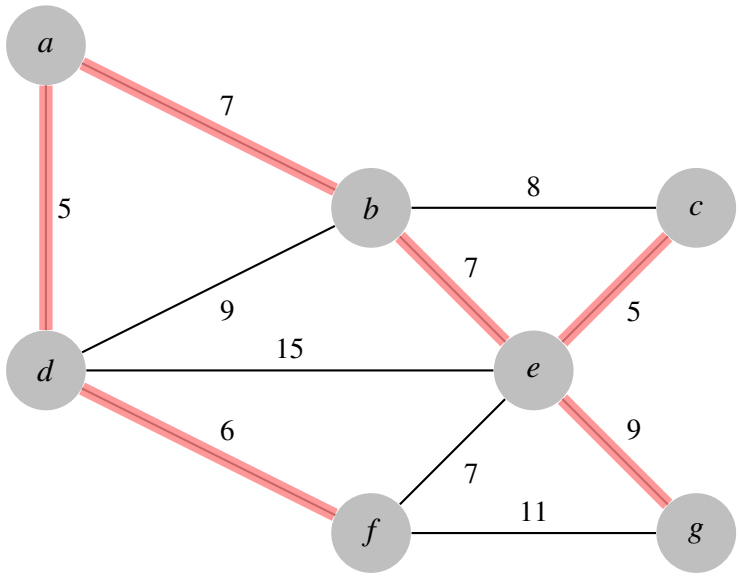

Figura 4.1 Exemplo de árvore geradora mínima. Fonte: O autor.

problema do caixeiro viajante consiste em encontrar, para um dado conjunto de pontos, o menor caminho que passe por todos os pontos do conjunto [42]. No problema do caixeiro viajante métrico, a distância entre os pontos deve obedecer a desigualdade triangular, ou seja, para todo conjunto de três pontos a, b e c, $distancia(a,b) \leq distancia(a,c) + distancia(c,b)$ [36]. Christofides mostrou que a solução apresentada por ele, no pior caso, geraria uma resposta que se diferenciaria da solução ótima para o caixeiro viajante em, no máximo, 50%.

Allison *et al.* [2] propuseram uma aplicação de árvore geradora mínima para observar segregação de massas em estrelas. Eles aplicaram o algoritmo de árvore geradora mínima ao *Orion Nebula Cluster (ONC)* e mostraram que o método proposto é capaz de detectar a segregação de massa com uma boa razão de aproveitamento.

Ma *et al.* [43] utilizaram o algoritmo de árvore geradora mínima em registro de imagens com a entropia de Rényi, como métrica de dissimilaridade entre as imagens. O algoritmo proposto foi testado em aplicações de ressonância magnética e registro de imagens sobre mapas de altura de terreno. Tal abordagem se mostrou acurada e robustoa

Novos algoritmos de mapeamento genético são necessários de modo a optimizar processos com grandes conjuntos de dados e mapas genéticos de alta densidade. Wu *et al.* [57] apresentaram um novo algoritmo para ordenar marcadores em um mapa genético. Com este algoritmo, foi provado que é possível determinar, eficientemente, a ordem correta dos marcadores, computando a árvore geradora mínima do grafo associado ao problema.

Existem três algoritmos mais conhecidos para a obtenção de uma árvore geradora mínima de um grafo: o algoritmo de Kruskal, o algoritmo de Prim, e o algoritmo de Boruvka

[41, 8, 12]. Os três algoritmos são ditos "gulosos", no sentido de que fazem suas escolhas visando a melhor opção atual na seleção das arestas da árvore geradora mínima, e não revertem a escolha no decorrer das iterações do método.

4.1.1 Algoritmo de Kruskal

O algoritmo de Kruskal é executado a partir da ordenação das arestas por seu custo agregado. Inicialmente, tem-se um grafo G sem arestas. A cada passo do algoritmo, adiciona-se a aresta de menor peso que conecta duas florestas distintas. A utilização de uma aresta conectando a mesma componente geraria um ciclo, em que uma das arestas desse poderiam ser removidas de modo a diminuir o custo, mantendo a conectividade, e, por isso, não são usadas.

A complexidade do algoritmo depende da implementação utilizada na verificação da conectividade dos conjuntos de vértices. Pode-se utilizar a implementação do *Union-find* apresentada no Capítulo 2, o que permite atingir a complexidade de tempo da ordem $O(|A(G)| \cdot \lg |V(G)|)$ [12]. No Algoritmo 35, tem-se o pseudo-código do algoritmo de Kruskal, como anteriormente apresentado.

Algoritmo 35 Algoritmo de Kruskal

```
1:  função MST-KRUSKAL(G)
2:      T ← ∅
3:      custo ← 0
4:      para cada aresta e ∈ A(G), em ordem crescente faça
5:          se e.u e e.v não estão no mesmo conjunto então
6:              T ← T ∪ e
7:              custo ← custo + C(e)
8:          fim se
9:      fim para
10:     Retorna custo
11: fim função
```

4.1.2 Algoritmo de Prim

O algoritmo de Prim consiste em aumentar a árvore geradora do grafo de maneira gulosa, a partir de um vértice inicial de escolha. A cada iteração, o algoritmo escolhe a aresta de custo mínimo dentre todas as possíveis opções de arestas, em que uma das extremidades está na árvore geradora que está sendo criada e a outra incide sobre um vértice não visitado. Tal estratégia, assim como no algoritmo de *Kruskal*, previne a criação de ciclos. Como o algoritmo cria a árvore geradora a partir de um vértice, a detecção da criação de um ciclo

se torna mais fácil, uma vez que são adicionados novos vértices à árvore somente se eles não foram anteriormente visitados, visto que não são criadas árvores disjuntas em passos intermediários do algoritmo.

De modo a implementar o algoritmo de Prim eficientemente, precisa-se de uma estrutura que seja capaz de selecionar, dentre todas as arestas possíveis, aquela que tem o menor peso. Para isso, utiliza-se uma fila de prioridade, que permite a implementação desse algoritmo em tempo $O(|A(G)| \cdot \lg |V(G)|)$.

Algoritmo 36 Algoritmo de Prim

```
1:  função MST-PRIM(G, s)
2:      para todo v ∈ V(G) faça
3:          bst_v ← ∞
4:          vis_v ← false
5:      fim para
6:      vis_s ← true
7:      bst_s ← 0
8:      para todo e ∈ A(G) que contenha s faça
9:          se bst_{e.v} > C(e) então
10:             bst_{e.v} = C(e)
11:             Q.push(e.v)
12:         fim se
13:     fim para
14:     custo ← 0
15:     enquanto Q nao vazia faça
16:         u ← Q.extractMinVertex()
17:         se u nao visitado então
18:             vis_u ← true
19:             custo ← custo + bst_u
20:             para todo e ∈ A(G) que contenha u faça
21:                 se bst_{e.v} > C(e) então
22:                     bst_{e.v} = C(e)
23:                     Q.push(e.v)
24:                 fim se
25:             fim para
26:         fim se
27:     fim enquanto
28:     Retorna custo
29: fim função
```

No Algoritmo 36, tem-se o pseudo-código referente ao algoritmo de Prim. Nessa função, a MST é gerada a partir do vértice s. A posição u do vetor *bst* representa a aresta de menor custo que conecta u à MST que está sendo gerada. Já um índice u no vetor *vis* representa se

o vértice já foi anteriormente processado pelo algoritmo. Inicialmente, insere-se em uma fila de prioridade os vértices adjacentes a s, ordenados pelo custo das arestas que ligam esses vértices a s. A seguir, é extraído da fila de prioridade o vértice com o menor custo, digamos u, que representa o vértice que será adicionado à MST atual se ele não foi visitado anteriormente. Após a adição de u, são atualizados os vértices candidatos a serem adicionados na MST na próxima iteração, adicionando as arestas que estão conectadas a u.

4.1.3 Algoritmo de Boruvka

O algoritmo de Boruvka [8] também encontra a árvore geradora mínima de um grafo com a restrição de que todas as arestas tenham pesos distintos. O algoritmo é, de certa forma, uma variação das ideias usadas por Prim e Kruskal nos seus algoritmos. Seja G o grafo que deseja-se obter a árvore geradora mínima, com vértices $V(G)$ e arestas $A(G)$. O algoritmo cria um grafo T, com os vértices de G, inicialmente sem arestas. A cada passo do algoritmo, para cada componente conexa de T, é procurada em $A(G)$ uma aresta e de menor custo e que tenha um de seus extremos em outra componente conexa. Então, e é adicionado ao conjunto de arestas de T. Esse passo é realizado até o momento em que exista somente uma componente conexa em T, e, como o algoritmo sempre conecta duas componentes conexas distintas de T, esse grafo contém $|V(G)|-1$ arestas. Portanto, T é uma árvore.

Algoritmo 37 Algoritmo de Boruvka

```
1:  função BORUVKA(G)
2:      T ← MakeSet(|V(G)|)
3:      custo ← 0
4:      enquanto |T| > 1 faça
5:          para cada componente conexa p ∈ T faça
6:              e ← {min(e)|e.u ∈ p ∧ e.v ∉ c}
7:              se DifferentSet(e.u, e.v) então
8:                  UnionSet(e.u, e.v)
9:                  custo ← custo + C(e)
10:             fim se
11:         fim para
12:     fim enquanto
13:     Retorna custo
14: fim função
```

No Algoritmo 37, tem-se a implementação do pseudo-código da solução proposta por Boruvka [8]. Inicialmente, criam-se os conjuntos de vértices separados, e, enquanto esse conjunto tiver mais do que uma componente, significa que existem ao menos dois sub-grafos desconexos no grafo. Então, procura-se, para cada componente p, a aresta de menor custo que tem uma das suas extremidades em p e a sua outra extremidade fora de p. Se essa aresta ainda não tiver sido inserida na árvore, adiciona-se essa aresta à árvore geradora mínima. Essa verificação é necessária, pois a utilização de arestas repetidas é sub-ótimo para a geração da árvore geradora de menor custo.

4.1.4 Técnicas de dinamização de árvore geradora mínima

Para a obtenção da árvore geradora mínima em um grafo estático, existem algoritmos eficientes e que são executados em tempo $O(|A(G)| \cdot \lg |V(G)|)$, como os apresentados anteriormente. Porém, para uma aplicação real, em alguns casos, os problemas nos quais esses algoritmos são aplicados estão em constante mudança e, por isso, as soluções de Prim e Kruskal não podem ser usadas diretamente.

Para grafos dinâmicos, pode-se considerar dois tipos de algoritmos: *online*, que são algoritmos que respondem as inserções imediatamente, isto é, retornam o valor da MST do novo grafo G imediatamente após a adição de uma aresta, ou *offline*, em que as inserções são todas lidas da entrada, processadas, e, posteriormente, respondidas.

O algoritmo *offline* proposto por Eppstein [20] para obter o MST em um grafo após algumas modificações funciona com base em duas operações: redução e contração. A fase de redução consiste em, dado um conjunto de operações em um bloco de modificações, encontrar um conjunto de arestas do grafo que não serão alteradas pela adição das arestas feitas nessas operações. Quando encontradas, essas arestas podem ser removidas do grafo sem alterar os resultados nesse bloco de operações. Além disso, o grafo resultante conterá um número menor de arestas.

Teorema 5. *Seja G um grafo não direcionado com peso nas arestas, e S um subconjunto de suas arestas. Seja T a MST de $G - S$. Então, não importa como os pesos das arestas em S sejam modificados, a MST de G conterá as arestas em $T \cup S$ [20].*

Teorema 6. *Assuma que está sendo realizada uma redução em um bloco de b atualizações. Então pode-se reduzir o número de arestas em um grafo de $|A(G)|$ para, no máximo, $|V(G)| + b - 1$ em tempo $|E(G)|$ [20].*

Estes dois teoremas são importantes, pois definem um limite superior sobre o número de arestas no grafo após as etapas de redução. A contração, similarmente, consiste em encontrar um conjunto de arestas que devem ser utilizadas em todas as MST's no bloco de operações. É possível contrair todas essas arestas e unir todos os vértices conectados por essas arestas. Com isso, é possível obter um grafo com um número menor de vértices.

Teorema 7. *Seja G um grafo não direcionado com pesos nas arestas, e S um conjunto de suas arestas. Seja T a MST de G, onde as arestas de S possuem pesos atribuídos que são menores que quaisquer outras arestas no grafo. Então, não importa como as arestas em S são modificadas, a MST de G sempre conterá as arestas de $T - S$ [20].*

Teorema 8. *Assumindo que é realizada a contração em um bloco com b atualizações. Então, é possível reduzir o número de vértices no grafo de $|V(G)|$ para, no máximo, $b + 1$ em tempo $O(|E(G)|)$ [20].*

Os teoremas 3 e 4 definem um limite superior para o número de vértices do grafo após a realização da contração do grafo com relação a um bloco de operações. Os teoremas apresentados são importantes, pois permitem a execução de algoritmos menos eficientes em instâncias menores do grafo com, no máximo, $b+1$ arestas e b vértices quando reduzidos e contraídos por um conjunto de operações de tamanho b.

Contudo, b deve ser escolhido cuidadosamente, pois se b for grande, existirão muitas operações de atualização por bloco, e o grafo contraído conterá muitas arestas e vértices. Por outro lado, se b for pequeno, será necessário realizar as contrações e reduções muitas vezes, o que aumentaria a complexidade do algoritmo.

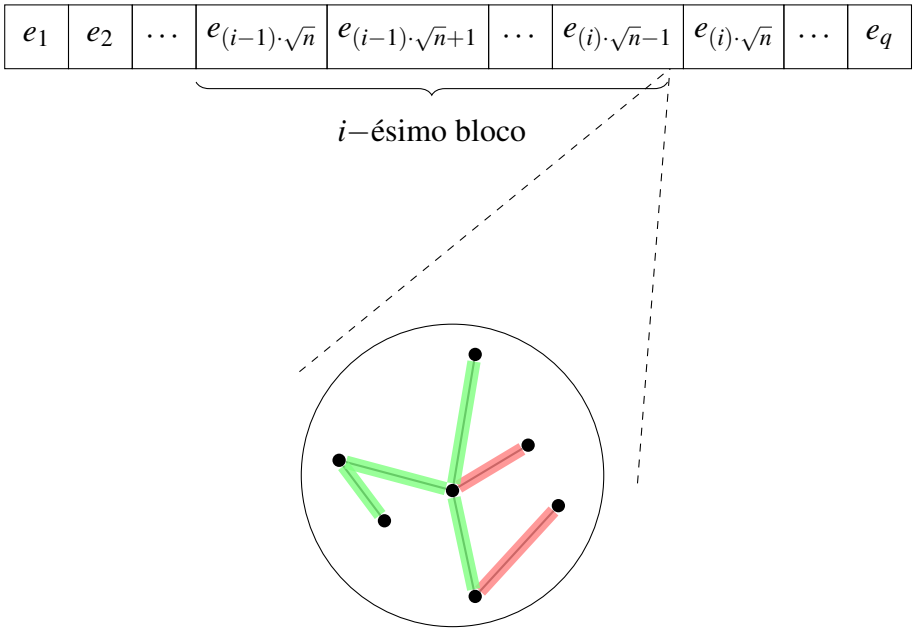

Figura 4.2 Exemplo de adição das arestas em T para gerar a MST desejada. As arestas verdes representam aquelas que estão antes do início do i-ésimo bloco, e, as vermelhas, representam as arestas que estão dentro do i-ésimo bloco

Com essas operações, é possível definir o seguinte algoritmo: divida as modificações em blocos de tamanho b, e, para processar as alterações realizadas nesse bloco, insira as arestas atualizadas nesse bloco em uma MST T. Após isso, obtêm-se todas as arestas antes desse bloco em ordem não-decrescente e adiciona-se à T. Como resultado do Teorema 8, o número de vértices no grafo após a remoção das arestas do bloco será no máximo $b+1$. Então, para cada atualização nesse bloco, é possível executar o algoritmo de Kruskal ou Prim nesse novo grafo contraído em tempo $O(b \cdot \lg b)$ por consulta. Novamente, se b é pequeno, será necessário inserir as arestas do bloco e comprimir o grafo muitas vezes. O número de vezes que a operação de reconstrução será executada é equivalente ao número de blocos

existentes. O número de blocos existentes considerando um conjunto de q operações em blocos de tamanho b é $\lceil \frac{q}{b} \rceil$.

A Figura 4.2 mostra uma execução de um algoritmo *offline*, onde foi escolhido o bloco de tamanho $b = \sqrt{|V(G)|}$. É apresentado o exemplo de um grafo com as arestas adicionadas antes do bloco atual. Existem apenas b arestas nesse bloco, e, então, somente b componentes serão criadas quando essas arestas são removidas do grafo. Na Figura 4.2, e_i representa a sequência de operações ordenadas por tempo.

No entanto, Eppstain [21] propôs um algoritmo por divisão e conquista nessa sequência de operações. A ideia principal é iniciar o algoritmo com blocos de tamanho b, reduzir, contrair e reduzir o grafo novamente. Depois dessas operações, o grafo contém, no máximo, $b+1$ vértices e b arestas. Então, o bloco é dividido na metade, e recursivamente é realizada essa sequência de operações enquanto existir mais de uma operação de modificação. Quando houver apenas uma modificação, no grafo existirá, no máximo, dois vértices com uma aresta. Após $\lg q$ iterações, o algoritmo estará nesse estado e, portanto, o algoritmo é executado em tempo $O(q \lg q)$.

Já Sleator e Tarjan [51] propuseram um algoritmo *online* proveniente de uma estrutura criada para a manutenção de árvores dinâmicas em que cada operação realizada tem custo amortizado $O(\lg(n))$ por operação. Essa estrutura, chamada *Link-cut tree*, se aproveita do funcionamento de uma *Splay-tree* (árvore binária auto-balanceável) para a obtenção dos custos logarítmicos por operação. Seja T uma árvore geradora mínima de G. Após a inserção de uma nova aresta, um ciclo será criado. Ou seja, qualquer aresta desse ciclo, se removida, pode gerar uma nova árvore geradora (não necessariamente mínima). Para a geração de uma árvore geradora que seja mínima, o algoritmo propõe a remoção da maior aresta desse ciclo, de modo a manter a propriedade de árvore desse grafo, minimizando o custo da árvore final.

A Figura 4.3 apresenta a execução da adição de uma aresta em uma MST previamente gerada através do algoritmo *online* proposto por Sleator e Tarjan. A aresta em amarelo foi adicionada, que gerou um ciclo e quebrou a propriedade de árvore. Dessa forma, o algoritmo encontra, dentro do ciclo gerado, a aresta de maior custo e a remove do grafo (aresta em azul). Essa operação é realizada com a estrutura citada anteriormente, e, por isso, pode ser executada de maneira eficiente.

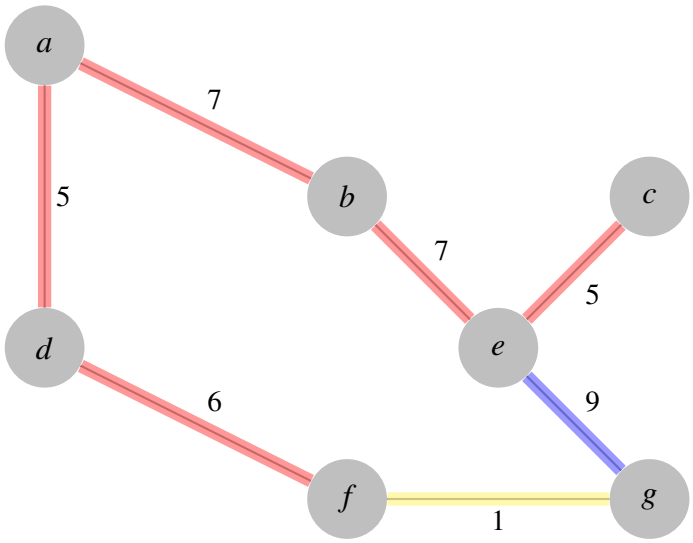

Figura 4.3 Após a adição da aresta $E = \{f, g\}$ (em amarelo), a solução ótima consiste em remover a aresta de maior valor do ciclo gerado por E, que é a aresta $\{e, g\}$ (em azul). A MST desse novo grafo é 31. Fonte: O autor.

4.2 MST totalmente retroativa

4.2.1 Notação Básica

Seja G um grafo não-direcionado com $|V(G)|$ vértices e $|E(G)|$ arestas, onde $V(G)$ é o conjunto de vértices de G e $A(G)$ o conjunto de arestas de G. Seja T^* a MST de G. Como se trata da versão totalmente retroativa do problema, G_i e T_i^* serão o grafo no tempo i e uma MST de G no tempo i, respectivamente. G_0 representa o grafo no instante inicial, enquanto todas as alterações serão realizadas em um espaço temporal iniciando no índice 1.

O problema a ser resolvido consiste em um conjunto de q operações sobre uma linha do tempo de tamanho m, onde dois tipos de operações são permitidas:

- Tipo 1: Adicionar uma aresta $e = \{u, v\}$, com custo $C(e)$, no tempo t em G;
- Tipo 2: Obter a MST de G_i.

A ideia principal do algoritmo proposto consiste em unir duas técnicas propostas por Demaine *et al.* [13] e Sleator *et al.* [51] para resolver o problema apresentado.

4.2.2 Compressão da linha temporal em blocos de $\sqrt{m}$

Suponha, sem perda de generalidade, que todas as inserções foram feitas em tempos distintos. Uma possível solução para resolver o problema apresentado consiste em manter T_i^* para

todos os possíveis tempos entre 0 e m, e, a cada adição de aresta, digamos no tempo p, re-executar o algoritmo de Kruskal/Prim para todas as árvores T_j^*, tal que $j \geq p$. Isso é computacionalmente custoso, pois é necessária a atualização de todas as árvores armazenadas após o tempo p, levando a uma complexidade temporal de $O(|E(G)| \cdot \lg |V(G)|)$ em cada uma dessas árvores, para cada adição de aresta, além de um grande consumo de memória (armazenar um grafo de tamanho $|V(G)|$ para cada possível tempo no intervalo [0, m]).

A técnica aplicada nesse algoritmo consiste em manter as árvores armazenadas em memória a cada intervalo de tempo b. Isto é, armazenam-se somente as árvores $\{T_0^*, T_b^*, T_{2b}^*, \ldots, T_{\lfloor m/b \rfloor}^*\}$. Então, as operações do tipo 1 podem ser executadas em tempo $O(\lfloor m/b \rfloor \cdot |E(G)| \cdot \lg |V(G)|)$ (pior caso adiciona uma aresta em cada uma das árvores armazenadas). As arestas serão mantidas em um vetor *edge* de modo que $edge_i$ armazena a aresta adicionada no tempo i (se existir). Já as consultas do tipo 2 podem ser executadas da seguinte forma:

- Se $i \cong 0 \pmod{b}$, então, a árvore está armazenada em memória, e, portanto, é suficiente obter o custo da MST armazenada em T_i^*;
- Senão, i está entre dois múltiplos de b. É suficiente obter a última árvore pré-calculada ($T_{\lfloor i/b \rfloor \cdot b}^*$), adicionar as arestas não adicionadas entre a última árvore e o tempo de consulta atual $(edge_{\lfloor i/b \rfloor \cdot b+1}, edge_{\lfloor i/b \rfloor \cdot b+2}, \ldots, edge_i)$.

Com essa técnica, é preciso escolher um valor ótimo para a variável b. Se b é muito pequeno, o número de árvores armazenadas em memória é grande, e isso não é bom para o desempenho tanto do ponto de vista espacial quanto do ponto de vista temporal. Se b é muito grande, a performance temporal do algoritmo nas operações do tipo 2 ficará comprometida. Seja m o tamanho da linha do tempo do problema e R a complexidade temporal de alguma estrutura de dados na qual as MST's ficarão armazenadas. A complexidade do algoritmo é dada por:

$$O(R \cdot \max(b, \frac{m}{b}))$$

Então, é necessário encontrar b que minimize o máximo entre b e $\frac{m}{b}$. Estes valores são inversamente proporcionais, e existe um ponto de equilíbrio entre eles quando:

$$b = \frac{m}{b} \Rightarrow b^2 = m \Rightarrow b = \sqrt{m}$$

Por esse motivo, será utilizado $b = \sqrt{m}$. Observe que esse valor está considerando que os tipos de operação serão realizados de maneira equiprovável.

4.2.3 Recomposição de MST T_i^* utilizando *Link-cut tree*

A maneira encontrada para otimizar o algoritmo proposto é manter, para cada bloco de $\sqrt{m}$, uma estrutura de dados que permita a adição de uma aresta em um grafo, a remoção de uma aresta em um grafo e obter a MST desse grafo de maneira a não re-executar os algoritmos clássicos para a obtenção de árvores geradoras mínimas. Sleator *et al.* [51] propuseram uma estrutura chamada *Link-cut tree*, que permite a manutenção de árvores dinâmicas (em que arestas são adicionadas ou removidas do grafo) e realizar todas as operações anteriormente descritas em tempo $O(\lg|V(G)|)$ amortizado [50].

Com isso, é possível utilizar essa árvore para gerar T_i^* entre as consultas, adicionando as arestas, como explicado, e fazer um *rollback* nas operações feitas em T_i^*, utilizando a estrutura para restaurar o estado inicial de T_i^* para as próximas consultas. Como em cada bloco existem, no máximo, $\sqrt{m}$ arestas, a consulta da MST de G_i pode ser feita com complexidade temporal amortizada de $O(\sqrt{m} \cdot \lg|V(G)|)$.

Link-cut tree

A *Link-cut tree* (LCT) é uma estrutura de dados baseada em um tipo de árvore binária balanceada chamada *Splay tree*. A ideia principal dessa estrutura de dados consiste em manter o caminho entre o nó mais recentemente usado e a raiz da árvore em uma árvore binária auto-balanceável. Com essa técnica, é possível obter uma complexidade amortizada de $O(\lg|V(G)|)$ por operação [51].

Como se trata de uma árvore binária de busca implícita, manter a operação de um caminho consiste em manter essa informação nos nós dessa árvore binária de busca. Portanto, se é necessário manter o nó com mais custo entre o caminho entre um nó v e a raiz r da árvore, basta manter o nó de custo máximo na árvore binária de busca gerada pelo acesso do nó v. Essa estrutura também permite operações de re-enraizamento da árvore em tempo logarítmico no número de nós da árvore.

Algoritmo 38 Maior vértice no caminho entre u e v

```
1: função GETMAXNODEONPATH(u, v)
2:     LCT.reroot(u)
3:     LCT.splay(v)
4:     LCT.getMaximumCostNode(root)
5: fim função
```

No algoritmo 38, tem-se o pseudocódigo para a obtenção do vértice com maior custo entre u e v na estrutura. A operação $reroot(u)$ modifica a raiz da árvore na *Link-cut tree (LCT)* para o vértice u; $splay(v)$ é a operação que mantém o caminho entre v e a raiz em uma

única árvore binária de busca. Como u é a raiz da árvore, e todos os vértices do caminho entre v e u estão nessa única árvore binária de busca, é possível obter o vértice de maior valor no caminho entre u e v em tempo $O(\lg|V(G)|)$ amortizado. Observe que a operação $reroot(u)$ modifica a raiz da árvore armazenada na estrutura e **não** tem relação com o re-enraizamento realizado pela splay-tree que mantém os caminhos dessa árvore.

Como no problema apresentado, o custo incide nas arestas e não nos vértices, ainda é necessário modificar a modelagem para utilizar a *Link-cut tree* de maneira direta. Seja $e = \{u, v\}$ com custo $C(e)$. É possível ver e como um vértice de custo $C(e)$, de tal modo que esse novo vértice fictício conecta u e v. Com essa abordagem é possível obter a maior aresta no caminho entre u e v sem grandes modificações na estrutura de dados, simplesmente obtendo o valor de um nó.

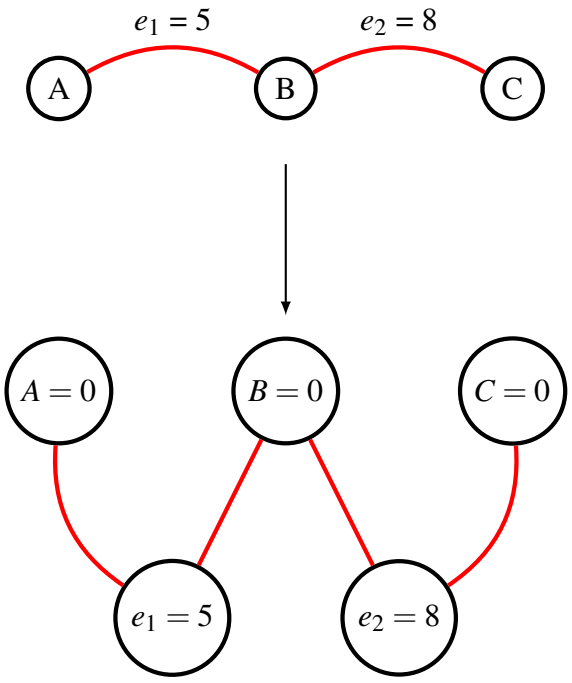

Figura 4.4 Exemplo de conversão do grafo

A Figura 4.4 mostra como realizar a conversão de um grafo para permitir a sua utilização em uma *Link-cut tree*. Na Figura 4.4 existe um grafo com três vértices (A, B, C) e duas arestas os conectando (arestas $e_1 = \{A, B\}$ com custo 5 e $e_2 = \{B, C\}$ com custo 8). A conversão cria um novo grafo com cinco vértices, em que cada um deles contém custos agregados. Os vértices que representam as arestas têm o custo que essas arestas possuíam, enquanto os vértices que representam os nós do grafo original têm custo zero.

Implementação

A implementação consiste em processar dois tipos de operações. Seja b o tamanho do bloco escolhido e t o tempo que a aresta e é adicionada. A atualização consiste em duas sub-operações:

- Adicionar a aresta e no tempo t;
- Adicionar a aresta e em todos os blocos após o tempo de inserção da aresta. Em outras palavras, adicionar e para todas as árvores T_j^*, tal que $j > \lceil \frac{t}{b} \rceil$.

O primeiro item é sobre atualizar as árvores nas MST's no início dos blocos. O segundo item tem relação com a recomposição da árvore que vai ser realizada no segundo tipo de operação. Aqui, a i-ésima MST contém todas as arestas no intervalo entre $[0, i \cdot b)$.

Algoritmo 39 Pseudocódigo para as operações do tipo 1 (inserção de arestas)

```
1: função ADDEDGE(e, t)
2:     edge_t ← e                          ▷ Adiciona a aresta e no tempo t
3:     i ← ⌈t/b⌉                           ▷ Encontra o indice do bloco que t pertence
4:     enquanto i ≤ ⌊m/b⌋ faça             ▷ Atualiza todos os blocos seguintes
5:         T_i*.link(e)                    ▷ Atualize o i-ésimo bloco
6:         i ← i + 1
7:     fim enquanto
8: fim função
```

Note que, no Algoritmo 39, as linhas 2 e 5 consistem exatamente na execução dos items previamente mencionados. Na linha 3, são mantidas as arestas no tempo em que foram adicionadas. Considerando que o algoritmo contém uma *Link-cut tree* mantendo a MST para cada bloco inicial, o pseudocódigo é executado em tempo $O(\sqrt{m} \cdot \lg |V(G)|)$.

Para a recomposição da i-ésima árvore geradora mínima, inicialmente é necessário encontrar a qual bloco t pertence. Seja i o bloco em que a consulta é realizada. Se $t \cong 0 \pmod{b}$, basta obter a resposta em tempo $O(1)$ na árvore pré-calculada T_t^*. Senão, é necessária a adição de algumas arestas para a obtenção da árvore geradora mínima desejada.

No Algoritmo 40, tem-se o pseudocódigo com a implementação da função que responde as consultas do tipo 2. Esse procedimento pode ser dividido em três partes principais: verificação de quais arestas serão adicionadas após a última MST armazenada em memória; adição dessas arestas a essa última MST armazenada e, finalmente, a realização do *rollback* de todas as operações feitas entre essa atualização, a fim de se recuperar a MST inicial do inicio do bloco de operações.

Algoritmo 40 Pseudocódigo para as operações do tipo 2 (consultas)

```
1: função GETMST(t)
2:     needAdd ← ∅
3:     i ← ⌊t/b⌋
4:     j ← i·b+1
5:     enquanto j ≤ t faça          ▷ Obtendo as arestas adicionadas após a última MST armazenada
6:         se edge_t ≠ ∅ então
7:             needAdd.insert(edge_t)
8:         fim se
9:         j ← j+1
10:    fim enquanto
11:
12:    operations ← ∅               ▷ Atualizando a última MST armazenada
13:    para e ∈ needAdd faça
14:        se e.u e e.v nao conectada então
15:            T_i^*.link(e)
16:            operations.insert("link", e)
17:        senão
18:            maxEdge ← T_i^*.getMaxEdgePath(e.u, e.v)
19:            se C(maxEdge) > C(e) então
20:                T_i^*.cut(maxEdge)
21:                T_i^*.link(e)
22:                operations.insert("cut", maxEdge)
23:                operations.insert("link", e)
24:            fim se
25:        fim se
26:    fim para                     ▷ Obtendo a resposta
27:    ans ← T_i^*.MST()
                                    ▷ Recuperando a última árvore armazenada
28:    reverse(operations)
29:    para op ∈ operations faça
30:        se op.operation = "link" então
31:            T_i^*.cut(op.e)
32:        senão
33:            T_i^*.link(op.e)
34:        fim se
35:    fim para
36:    Retorna ans
37: fim função
```

A primeira parte é resolvida entre as linhas 2 a 10. Primeiramente, é encontrado o bloco i que t está contido. A seguir, é adicionado as arestas entre $i \cdot b + 1$ (início do bloco que contém

t) e t para o conjunto *needAdd*. Estas arestas são aquelas que não estão incluídas na última árvore armazenada.

A segunda parte consiste em, para cada aresta no conjunto *needAdd*, digamos $e \in$ *needAdd*, verificar se os extremos de e estão conectados. Em caso negativo, a MST atual certamente conterá essa aresta. Se os extremos de e estão na mesma componente conexa, a adição dessa aresta nesse grafo gerará um ciclo. Então, e fará parte da resposta se e somente se houver uma aresta f tal que f está contida no caminho entre $e.u$ e $e.v$ e $C(f) > C(e)$, onde C é a função de custo agregado em uma aresta. Isto é verdade, uma vez que é possível trocar as arestas e e f, mantendo a conectividade, e não violando as propriedades de uma árvore geradora de G, removendo o ciclo criado pela adição de e.

Após estas operações, é possível obter a MST no tempo t realizando $\sqrt{m} \cdot \lg |V(G)|$ operações, porque entre $i \cdot b + 1$ e t existirão, no máximo, $\sqrt{m}$ novas arestas (pela escolha de b), e toda operação *link/cut* é realizada em tempo $O(\lg |V(G)|)$ amortizado. Após estas inserções, o estado inicial da estrutura é perdido. Então, todas as operações realizadas na MST devem ser desfeitas. Como as operações são inversamente relacionadas entre si, a recomposição da árvore pode ser realizada sem grandes dificuldades.

4.3 Caminho de custo mínimo

O problema do caminho mais curto entre dois vértices de um grafo foi sugerido e resolvido por Edsger W. Dijkstra em 1959 [17]. Ele é amplamente pesquisado pela comunidade científica, tendo em vista suas diversas aplicações em problemas desde os mais clássicos, como planejamento de rotas para carros [19, 23], robôs [46], planejamento de roteamento [7, 34, 38], problemas de otimização modeláveis a grafos [48] e em outras áreas, como na biologia [59].

Com o passar do tempo, algumas variantes do problema de caminho mais curto foram propostas. Uma dessas variantes é descrita da seguinte forma: suponha que após a execução do algoritmo de Dijkstra, alguma aresta tenha seu valor alterado. Claramente após sua execução, se torna difícil a reutilização do resultado, pois a modificação nessa aresta pode alterar totalmente a árvore de caminhos mínimos. Uma alternativa seria reexecutar o algoritmo sempre que uma aresta fosse alterada no grafo. Contudo, essa solução nem sempre é viável. Em certos casos, é possível reutilizar algumas informações da árvore gerada anteriormente para solucionar o problema. Esse problema é chamado problema do caminho mínimo em grafos dinâmicos *(dynamic shortest path problem)*.

Os problemas de caminho mínimo em grafos dinâmicos têm sido classificados na literatura de duas maneiras: semi-dinâmico *(semi-dynamic)* e totalmente dinâmico *(fully dynamic)*.

Algoritmos de caminho mínimo semi-dinâmico podem ser de dois tipos: incrementais, que permitem apenas inserções de arestas/incremento do custo das arestas já existentes, e decrementais, que permitem apenas remoções de arestas/decremento do custo das arestas já existentes.

Como o algoritmo proposto por Dijkstra utiliza uma fila de prioridade para construir essa solução, a utilização de uma fila de prioridade retroativa pode ser usada para a construção de um algoritmo em que é possível alterar o valor das arestas. Mesmo com muitas alterações em seus valores, é possível obter um algoritmo com complexidade menor do que reexecutar o algoritmo original a cada modificação do grafo. O algoritmo proposto por Sunita e Deepak [52] utiliza a fila de prioridade retroativa não-consistente para alcançar uma complexidade $O(n * log(m))$ por atualização, para a solução da versão totalmente dinâmica do caminho mínimo. A noção de estruturas retroativas não-consistentes permite que seja possível descobrir onde será a próxima posição que a fila de prioridade iria falhar caso uma aresta fosse inserida ou alterada.

4.3.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra consiste em gerar o menor caminho em um grafo sem ciclos negativos, a partir de um vértice s, para todos os outros vértices do grafo, gerando uma árvore de caminhos mínimos.

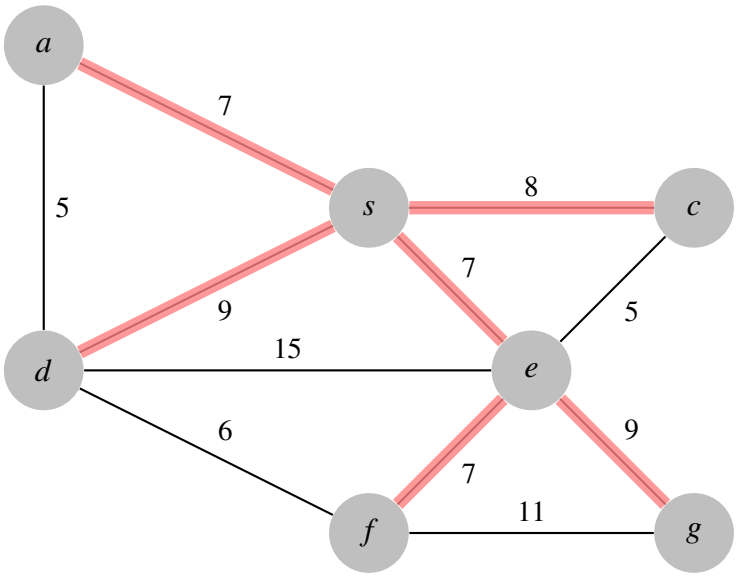

Figura 4.5 Exemplo de árvore de caminhos mínimos tendo como raiz o vértice s. Os números acima das arestas representam o custo de se utilizar a aresta. As arestas em vermelho representam as arestas de uma das possíveis soluções ótimas. Fonte: O Autor

Como pode-se observar na Figura 4.5, tem-se o exemplo de uma árvore de caminhos mínimos. Uma árvore de caminhos mínimos é definida por um subgrafo de G, digamos T, onde T é um grafo sem ciclos, enraizado em um vértice s, em que o caminho mínimo de s para qualquer outro vértice em $v \in V(T)$ é o mesmo, tanto em G quanto em T.

Seja d_u a menor distância possível entre um vértice $u \in V(G)$ e s (o vértice inicial). Inicialmente, $d_u = \infty \forall u \in G$ e $d_s = 0$. A cada iteração, o algoritmo de Dijkstra encontra o vértice u com a menor distância com relação a s já processado. Então, o algoritmo procura pelos vértices adjacentes a u, digamos v, conectados por uma aresta $e = \{u, v\}$, tal que $d_u + C(e) < d_v$, em que $C(e)$ é o custo da aresta e. Ao encontrar vértices com essa característica, o algoritmo realiza a atualização da distância mínima de v com relação a s, pois existe um caminho partindo de s, passando por u, e terminando em v, com custo menor a outro caminho que anteriormente chegava a v. Esse processo é chamado *relaxamento das arestas*. A cada passo do algoritmo, um dos $|V(G)|$ vértices do grafo é processado. Para encontrar o elemento de menor distância com relação a s, o algoritmo original de Dijkstra percorria toda a lista de vértices, o que gerava uma complexidade de $O(|V(G)|^2)$. Após alguns anos, uma variante que utilizava uma fila de prioridade para a descoberta do elemento de menor distância de s foi proposta por Fredman e Tarjan [54], o que otimizava o algoritmo original, alcançando uma complexidade de $O(|V(G)| + |A(G)| \cdot \lg(|V(G)|))$. O pseudocódigo referente à solução pode ser visto no Algoritmo 41.

Algoritmo 41 Algoritmo de Dijkstra

```
1:  função DIJKSTRA(G, s)
2:      d_i ← ∞ ∀ i ∈ G
3:      p_i ← NULL ∀ i ∈ G
4:      d_s ← 0
5:      q.push({d_s, s})          ▷ q é uma fila de prioridade ordenada pelo primeiro atributo
6:      enquanto q não vazia faça
7:          {w, u} ← q.extractMin()
8:          se d_u == w então
9:              para toda aresta e ∈ A(G) adjacente a u faça
10:                 v ← e.v
11:                 se d_u + C(e) < d_v então
12:                     d_u ← d_v + C(e)
13:                     p_u ← v
14:                     q.push({d_v, v})
15:                 fim se
16:             fim para
17:         fim se
18:     fim enquanto
19: fim função
```

No Algoritmo 41, inicialmente, tem-se a inicialização do vetor d, como explicado anteriormente. Em seu estado inicial, nenhum vértice foi processado, exceto o vértice s. Os vértices são mantidos em ordem crescente por sua distância com relação a s na estrutura que representa uma fila de prioridade. Nessa estrutura, além da distância do vértice com relação a s, é mantido o índice do vértice.

O algoritmo de Dijkstra nem sempre é utilizado em sua forma padrão nas aplicações. Em certas situações, são realizadas algumas otimizações nesses algoritmos. O algoritmo que utiliza o algoritmo proposto por Dijkstra como base para a geração de um método mais eficiente, é chamado algoritmo de busca A* [29, 60]. Esse algoritmo foi desenvolvido para um robô autônomo que pudesse planejar suas próprias movimentações, sendo comumente utilizado em várias aplicações com essa natureza.

4.3.2 Técnicas de dinamização dos algoritmos de caminho mínimo

Algumas soluções foram apresentadas para versões específicas desse problema. Gallo [24] apresentaram uma solução para o problema incremental semi-dinâmico do caminho mínimo

em 1981. No mesmo ano, Even e Shiloach [22] propuseram uma solução para o caso decremental que era executada em tempo $O(|V(G)| \cdot |A(G)|)$.

Ramanligam e Reps [49] propuseram um algoritmo que se mostrou o mais eficiente na prática, apesar de sua alta complexidade no pior caso. A ideia do algoritmo consiste em identificar o conjunto de vértices afetados pela alteração do peso de uma aresta. Esse conjunto é dividido em dois subconjuntos de vértices. Um deles consiste nos vértices que foram atualizados e não podem ser recalculados utilizando a atual topologia da árvore de caminhos mínimos. O outro subconjunto consiste nos vértices que podem ser atualizados utilizando essa topologia de árvore. Em outras palavras, seja S o conjunto dos vértices que tiveram sua distância mínima alterada pela mudança da aresta, A o conjunto dos vértices que não podem ser recalculados utilizando a topologia atual da árvore, e B o conjunto dos vértices que podem ser recalculados utilizando a topologia atual, tem-se que $S = A \cup B$. O conjunto de vértices A é inserido em uma fila, enquanto o conjunto B é inserido em uma *heap*. Assim, é possível atualizar os caminhos mínimos dos vértices na fila por meio dos vértices armazenados na *heap*.

Existem algumas especializações e variações desse algoritmo, como o algoritmo proposto por King e Thorup [39], em que para cada vértice da árvore de caminhos mínimos se tem um *link* para o próximo potencial caminho mínimo a partir desse vértice. Demterscu *et al.* [15] também propuseram algumas variações envolvendo um mecanismo mais simples para descobrir o conjunto de vértices alterados que estão na fila proposta pelo algoritmo de Ramanligam e Reps.

Dinamização utilizando retroatividade não-consistente

Com as ferramentas apresentadas na seção 3.3, é possível dinamizar o problema de caminho mínimo em grafos dinâmicos. A união das técnicas para a dinamização do algoritmo de *Dijkstra* [17] foi proposto por Sunita *et. al* [52]. Nessa abordagem, é realizada a substituição da *min-heap* utilizada no algoritmo de *Dijkstra* para a versão retroativa não-consistente da fila de prioridade sendo possível propagar as modificações realizadas na fila de prioridade para acomodar as mudanças dinâmicas no grafo.

Para a dinamização do algoritmo, será utilizado como base o Algoritmo 41. Será considerado que a execução do algoritmo já foi realizada, e o vetor de distâncias d já foi completamente preenchido, gerando a árvore de caminhos mínimos de G. Então, o custo de uma aresta $C(e)$ será modificada. Existem dois casos a serem considerados:

- O custo da aresta e diminui: nesse caso, pode existir um novo caminho partindo do vértice inicial s para os outros vértices de G, passando pela aresta e;

- O custo da aresta e aumenta: caso essa aresta faça parte do único caminho mínimo, o incremento do custo de e pode acarretar a modificação do caminho mínimo na ACM, uma vez que se e estava na árvore de caminhos mínimos de G, após seu incremento, a aresta e poderá ser removida da ACM.

Portanto, para cada caso, será realizada a execução de um algoritmo diferente, ambos usando a fila de prioridade retroativa. O primeiro passo, em ambos os algoritmos, consiste em atualizar o grafo com o novo incremento/decremento do peso da aresta e. Além disso, é utilizada uma *heap* H para manter as inconsistências que aparecerem durante uma atualização, de modo que essas inconsistências fiquem ordenadas por tempo. Em caso de mesmo tempo para duas inconsistências, é utilizada a distância do vértice com inconsistência no tempo t. Essa *heap* mantém em sua estrutura os elementos na forma $(t_{ins}, d_v, v, t_{del})$, onde t_{ins} é o tempo de inserção de um vértice, d_v é a distância desse vértice, v corresponde ao vértice atual e t_{del} corresponde ao tempo de deleção do vértice. Finalmente, são corrigidas as inconsistências geradas pela modificação da aresta na FPR através da *heap* H.

Algoritmo 42 Algoritmo de Dijkstra Dinâmico - Caso incremental

função DYNAMICDIJKSTRAINC(G, e, δ)
▷ **Passo 1** - Atualizar Aresta
$C(e) \leftarrow C(e) + \delta$
▷ **Passo 2** - Atualizar vetor de distâncias da ACM
$distanciaV \leftarrow d_{e.v}$
para toda aresta $e' \in A(\overline{G})$ adjacente a $e.v$ **faça**
se $d_{e'.v} + C(e') < d_{e'.u}$ **então**
$d_{e'.u} \leftarrow d_{e'.v} + C(e')$
$p_{e'.u} \leftarrow e'.v$
fim se
fim para
se $d_{e.v} == distanciaV$ **então**
fim função ▷ Não houve modificação na ACM
fim se
▷ **Passo 3** - Propagar modificação
$t_{ins} \leftarrow tempoInsercao(e.v)$
$t_{del} \leftarrow tempoDelecao(e.v)$
$Delete(t_{ins})$ ▷ Deleta a operação Push realizada no tempo t
$Insert(t_{ins}, Push(\{d_{e.v}, e.v\}))$

▷ Cada operação executada na fila de prioridade não consistente adiciona na *heap* H os tempos inconsistentes para propagação das atualizações
enquanto H não vazia **faça**
$x \leftarrow H.getMin()$ ▷ Extrai da FPR o vértice a ser atualizado com o menor distância de s
se x é sucessor de $e.v$ na ACM **então**
Vai para o **Passo 2**
senão
Incrementa o tempo de deleção do vértice x em 1 na FPR
fim se
fim enquanto

fim função

No algoritmo 42, tem-se o pseudocódigo da solução para o problema incremental do caminho mínimo. Na função apresentada, a aresta e que pertence ao grafo G tem seu custo incrementado em δ. O incremento da aresta e pode ou não acarretar uma modificação na ACM. Se a aresta e não fazia parte da ACM, o seu incremento não afetará a solução final. Todavia, se a aresta fazia parte da ACM, o seu incremento modificará a ACM se houver um caminho até $e.v$ que não passa por $e.u$, e que seja menor que o caminho mínimo a partir de s, utilizando a aresta e. Nesse caso, o vetor de distâncias é atualizado e repassado para a FPR, que retornará as operações inconsistentes com a realização da modificação. Essas operações inconsistentes serão extraídas da *heap* H e corrigidas. Correções também podem gerar inconsistências, e todas elas são inseridas em H, até que a FPR esteja consistente novamente.

Já no Algoritmo 43, tem-se o pseudocódigo para o caso decremental. Na função, a aresta e do grafo G tem seu custo decrementado de δ. Inicialmente, atualiza-se o custo da aresta e. O decréscimo do custo da aresta pode gerar uma alteração por toda a árvore de caminhos mínimos caso esta aresta não faça parte da ACM inicialmente. Se essa aresta faz parte da ACM antes da modificação, a diminuição do custo afetará todos os sucessores de $e.v$ na ACM. Isso também acontece no caso em que a diminuição do custo de e gera a sua adição na ACM, substituindo outra aresta já existente. Em todos esses casos, é necessária a realização do processo de relaxamento das arestas provenientes de nós afetados pela modificação.

Segundo Sunita *et. al* [52], o algoritmo descrito tem complexidade $O(|E(G)| \cdot \lg(|V(G)|))$ no pior caso por modificação. Porém, a execução de testes realizada por Sunita *et. al* sugere que o algoritmo proposto tem um melhor desempenho que os algoritmos propostos por Ramanligam e Reps [49], e que a reexecução do algoritmo para grafos estáticos.

Algoritmo 43 Algoritmo de Dijkstra Dinâmico - Caso decremental

função DYNAMICDIJKSTRADEC(G, e, δ)
    ▷ **Passo 1** - Atualizar Aresta
    $C(e) \leftarrow C(e) - \delta$
    ▷ **Passo 2** - Atualizar vetor de distâncias da ACM
    se $d_{e.v} > d_{e.u} + C(e)$ **então**
        $d_{e.v} \leftarrow d_{e.u} + C(e)$
        $p_{e.v} \leftarrow e.u$
    senão
        fim função
    fim se
    ▷ **Passo 3** - Propagar modificação
    $t_{ins} \leftarrow tempoInsercao(e.v)$
    $t_{del} \leftarrow tempoDelecao(e.v)$
    $Delete(t_{ins})$ ▷ Deleta a operação Push realizada no tempo t
    $Insert(t_{ins}, Push(\{d_{e.v}, e.v\}))$

    ▷ Cada operação executada na fila de prioridade não consistente adiciona na *heap* H os tempos inconsistentes para propagação das atualizações
    enquanto H não vazia **faça**
        $x \leftarrow H.getMin()$ ▷ Extrai da FPR o vértice a ser atualizado com o menor distância de s
        se $e' \in E(G) | e'.u == x$ e $d_{e'.v} > d_{e'.u} + C(e')$ **então**
            $d_{e'.v} \leftarrow d_{e'.u} + C(e')$
            $p_{e'.v} = e'.u$
            Passo 3
        senão
            Incrementa o tempo de deleção do vértice x em 1 na FPR
        fim se
    fim enquanto

fim função

Capítulo 5

Testes Empíricos

Este capítulo apresenta os testes empíricos realizados sobre as versões retroativas das estruturas de dados apresentadas nessa dissertação e foram executados em um computador com processador Intel Core i5-4200U CPU @ 1.60GHz x 4 com 8 gigabytes de memória. Os testes foram gerados de maneira totalmente aleatória, sempre mantendo a consistência (ou seja, as operações realizadas sempre serão válidas para todos os tempos da estrutura), e de modo que todas as operações foram realizadas em tempos distintos. Além disso, fixou-se que todas as versões da estrutura estão contidas no intervalo entre 1 e 10^5, uma vez que em certos algoritmos testados, o tamanho da linha do tempo influencia no consumo de memória e na complexidade temporal dessas estruturas.

Para aferição de desempenho, foram utilizadas duas ferramentas: *gtest*, uma ferramenta do *Google* para automação de testes e avaliação de desempenho temporal de um código, e *Valgrind*, para o cálculo do consumo de memória dos algoritmos.

5.1 Execução dos testes em uma fila retroativa

Foram gerados testes aleatórios para avaliação das versões retroativas da fila. Em cada conjunto de testes foram gerados cinco tipos de operação:

- *Insert(t, Enqueue(x))*: insere o elemento x na fila no tempo t;
- *Insert(t, Dequeue())*: remove o elemento mais antigo na estrutura que foi inserito até o tempo t;
- *Delete(t, Enqueue(x))*: remove a operação de inserção do elemento x no tempo t;
- *Delete(t, Dequeue())*: remove a operação *Dequeue* realizada na estrutura no tempo t;

- *Front(t)*: retorna o próximo elemento a ser removido da fila no tempo t. Na fila parcialmente retroativa, t é sempre o tempo presente.

As operações foram geradas mantendo-se a consistência da estrutura por meio de sua linha temporal do ponto de vista retroativo. Por exemplo, uma operação do tipo *Delete(t, Dequeue())* não será gerada caso não exista outra operação do tipo *Insert(t, Dequeue())* previamente executada na estrutura. Além disso, não existirão operações que sejam inconsistentes com relação à estrutura em si, como, por exemplo, a execução de uma operação retirada de um elemento da fila em um tempo em que ela está vazia.

Os arquivos gerados para teste da estrutura podem ser encontrados no repositório referente à dissertação no seguinte link https://github.com/juniorandrade1/Master/tree/master/tests/Datasets/Queue. Cada arquivo contém inicialmente um inteiro n correspondente ao número de operações realizadas na estrutura, seguido de n linhas contendo as operações referentes à fila retroativa no formato anteriormente descrito.

5.1.1 Testes em uma fila parcialmente retroativa

Na Figura 5.1 é apresentado um teste comparativo entre duas implementações distintas da fila de prioridade parcialmente retroativa. A primeira implementação *(Demaine (2007))* corresponde à solução proposta por Demaine *et al.* (2007) [13] para a manutenção de uma fila de prioridade parcialmente retroativa. A segunda solução *(Força Bruta)* trata de um algoritmo que armazena todas as atualizações realizadas na estrutura em uma árvore binária de busca, ordenadas por tempo, e, após a execução de uma operação de consulta, efetua todas as modificações armazenadas até o tempo mais recente.

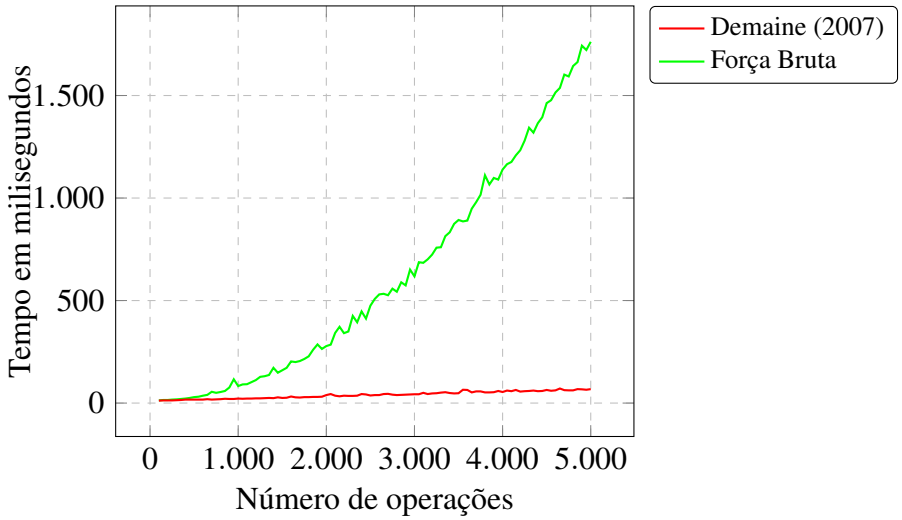

Figura 5.1 Execução da fila parcialmente retroativa comparada à solução padrão. Fonte: O autor.

Pode-se observar que, para instâncias menores, o algoritmo força-bruta é competitivo pela constante alta do algoritmo *Demaine (2007)*. Porém, em entradas aleatórias com mais de mil operações, o algoritmo *Demaine (2007)* já é executado aproximadamente cinco vezes mais rápido que o algoritmo padrão, e essa diferença de tempo de processamento entre os dois algoritmos cresce rapidamente à medida que o número de operações aumenta.

Isso é explicado pela complexidade temporal teórica dos algoritmos com relação às operações de consulta. O algoritmo *Demaine (2007)* realiza uma operação de consulta com complexidade temporal $O(\lg n)$, enquanto o algoritmo *força bruta* necessita percorrer todas as operações realizadas, consumindo tempo $O(n)$. Nas duas soluções testadas, tanto no algoritmo retroativo quanto no algoritmo *força bruta*, as operações de modificação como *Enqueue* e *Dequeue* são executadas com uma complexidade temporal de $O(\lg n)$.

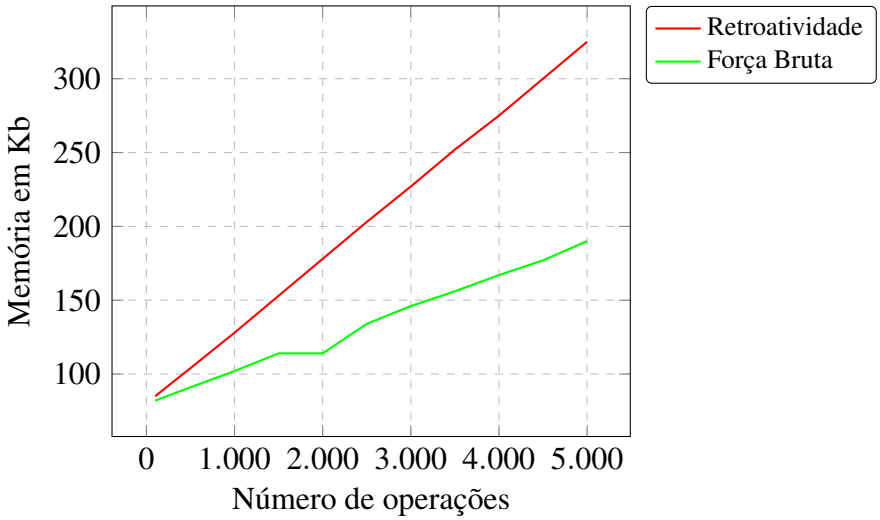

Figura 5.2 Consumo de memória de uma fila parcialmente retroativa. Fonte: O autor.

Contudo, o consumo de memória do algoritmo retroativo é maior que o da solução *força bruta* para o problema, como pode ser visto na Figura 5.2. A imagem confirma a tendência do algoritmo proposto por Demaine *et al.* (2007) [13] para a fila parcialmente retroativa de ser mais custoso em termos de memória, uma vez que esse algoritmo utiliza duas árvores binárias balanceadas, enquanto no algoritmo *força bruta* é necessário somente a manutenção de uma árvore binária balanceada.

5.1.2 Testes em uma fila totalmente retroativa

Na Figura 5.3 tem-se o gráfico comparativo com a velocidade de execução de duas abordagens para a implementação da fila totalmente retroativa. Nesse caso, a execução do algoritmo força-bruta é ligeiramente mais veloz que a sua versão parcialmente retroativa. Isso é explicado pelo fato de que na fila parcialmente retroativa, é necessário executar todas as operações realizadas na estrutura, enquanto na fila totalmente retroativa essa reconstrução depende do valor t correspondente à versão da estrutura consultada. Novamente, pode-se observar que, nos casos maiores, a vantagem da utilização da estrutura apresentada é expressiva.

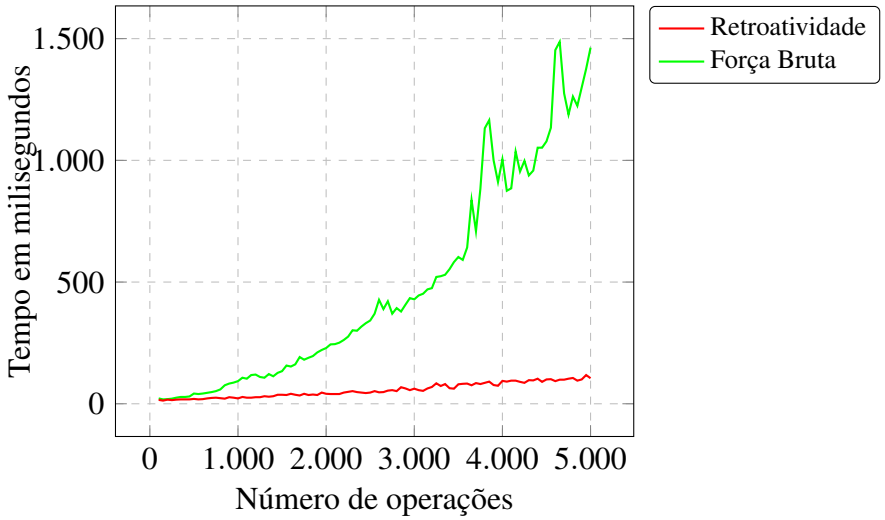

Figura 5.3 Desempenho da fila totalmente retroativa. Fonte: O autor.

Na Figura 5.4 são apresentados o consumo de memória pelos algoritmos propostos. Novamente, o algoritmo retroativo proposto por Demaine *et al.* (2007) [13] consome mais memória que o algoritmo *força bruta*, pois armazena uma árvore binária balanceada a mais que o algoritmo *força bruta*. Contudo, é importante observar que em ambos os algoritmos para a fila retroativa, tanto em sua versão parcialmente retroativa quanto em sua versão totalmente retroativa, a complexidade de memória é de $O(n)$.

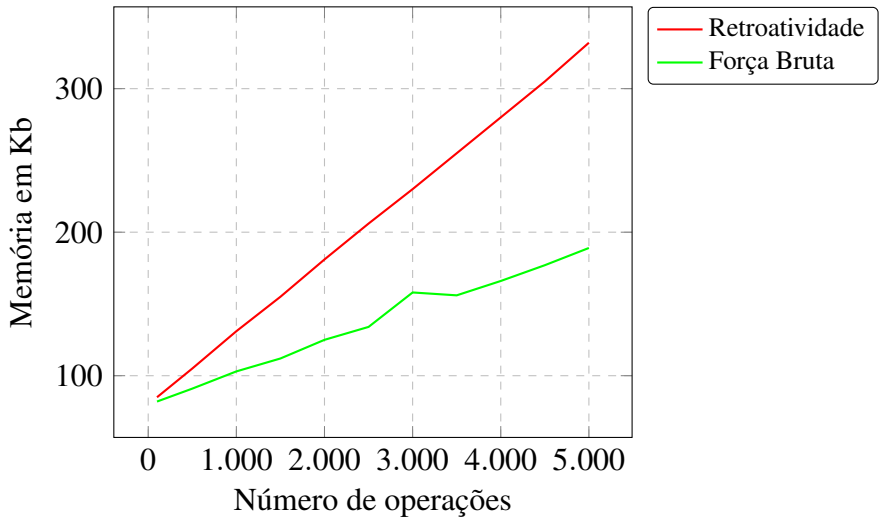

Figura 5.4 Consumo de memória de uma fila totalmente retroativa. Fonte: O autor.

5.2 Execução dos testes em uma pilha

Foram gerados testes aleatórios para avaliação das versões retroativas da pilha. Em cada conjunto de testes foram gerados cinco tipos de operações distintas:

- *Insert(t, Push(x))*: insere o elemento x no topo da pilha no tempo t;
- *Insert(t, Pop())*: remove o elemento no topo da pilha no tempo t;
- *Delete(t, Push(x))*: remove a operação de inserção do elemento x no tempo t;
- *Delete(t, Pop())*: remove a operação *Dequeue* realizada na estrutura no tempo t;
- *Peek(t)*: retorna o elemento do topo da pilha no tempo t. Na fila parcialmente retroativa, t é sempre o tempo presente.

As operações foram geradas mantendo-se a consistência da estrutura através de sua linha temporal do ponto de vista retroativo. Por exemplo, uma operação do tipo *Delete(t, Push(x))* não será gerada caso não exista outra operação do tipo *Insert(t, Push(x))* previamente executada na estrutura. Além disso, não existirão operações que sejam inconsistentes com relação à estrutura em si, como, por exemplo, a execução de uma operação de consulta *Peek(t)* em uma pilha vazia.

Os arquivos gerados para teste da estrutura podem ser encontrados no repositório referente à dissertação no seguinte link https://github.com/juniorandrade1/Master/tree/master/tests/Datasets/Stack. Cada arquivo contém inicialmente um inteiro n correspondente ao número de operações realizadas na estrutura, seguido de n linhas contendo as operações referentes a pilha retroativa no formato anteriormente descrito. Mais detalhes sobre o formato das operações nos arquivos de teste podem ser encontrados no link anteriormente citado.

5.2.1 Testes em uma pilha parcialmente retroativa

Na Figura 5.5, tem-se a execução de testes aleatórios em uma pilha parcialmente retroativa. Foram testadas duas abordagens referentes à implementação da versão da pilha parcialmente retroativa. A primeira abordagem, chamada de *força bruta*, consiste em uma solução menos sofisticada para o problema, armazenando todas as operações de modificação realizadas na estrutura ordenadas por tempo, e, a cada operação de consulta, executando e simulando todas as operações realizadas até o instante de tempo atual. Já a abordagem *Demaine (2007)* consiste na implementação do algoritmo proposto por Demaine *et al.* (2007) [13] para a manutenção de uma pilha retroativa.

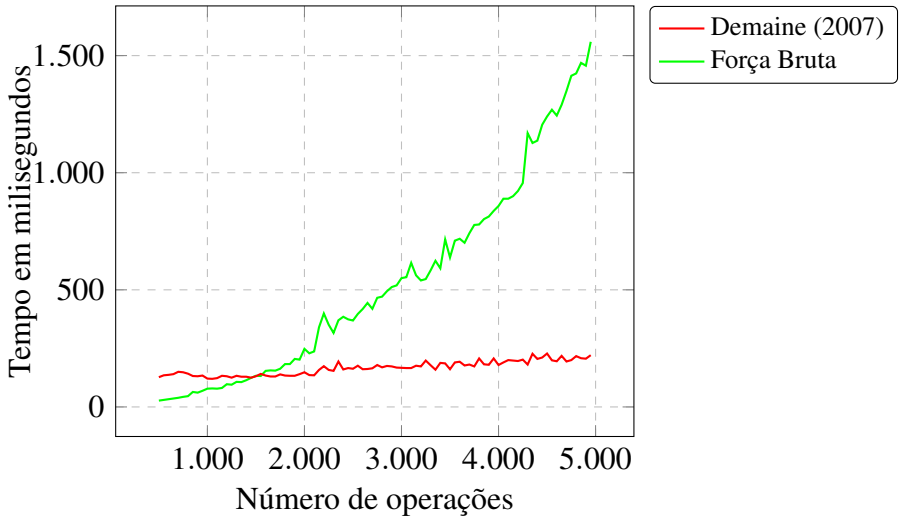

Figura 5.5 Teste de velocidade de execução de uma pilha parcialmente retroativa. Fonte: O autor

Pode-se observar que pela natureza da estrutura em que a pilha está implementada no algoritmo *Demaine (2007)*, a constante na complexidade temporal desse algoritmo é mais alta. Portanto, para testes extremamente pequenos, o algoritmo *força bruta* é mais rápido que o algoritmo *Demaine (2007)*. Porém, à medida que o número de operações cresce, o algoritmo que implementa a persistência parcial proposta por Demaine *et al.* (2007) abre larga vantagem com relação ao tempo de execução, chegando a ser trinta vezes mais rápido em casos com cinco mil operações.

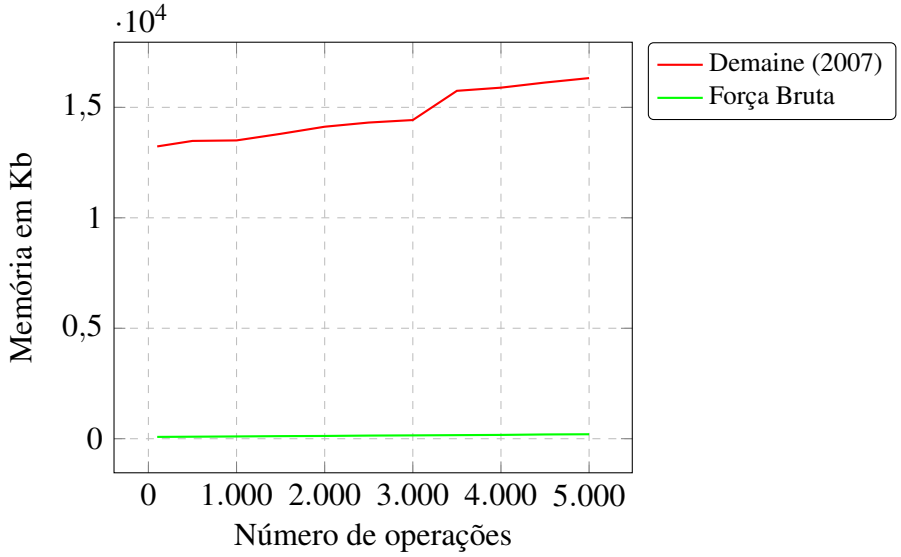

Figura 5.6 Consumo de memória de uma pilha parcialmente retroativa. Fonte: O autor

Na Figura 5.6 é apresentado o consumo de memória da execução dos mesmos testes em uma pilha parcialmente retroativa. É notável a diferença de consumo em termos de memória da implementação baseada no algoritmo de Demaine *et al.* (2007) com relação ao algoritmo *força bruta*. O algoritmo *Demaine (2007)* consome mais memória, pois pela utilização de uma árvore de segmentos, o número de operações realizadas só afeta o tempo de execução desse algoritmo. O consumo de memória do algoritmo *Demaine (2007)* é limitado pelo tamanho da linha temporal em que a estrutura está implementada. Já o algoritmo *força bruta* tem seu consumo de memória proporcional ao número de operações, independentemente do tamanho da linha temporal, o que confere a esse algoritmo no aspecto espacial.

5.2.2 Testes em uma pilha totalmente retroativa

Para a pilha totalmente retroativa, foram implementados dois algoritmos similares aos utilizados para teste de uma pilha parcialmente retroativa. O algoritmo *Demaine (2007)* implementa a solução proposta por Demaine *et al.* (2007) [13] para a manutenção de uma pilha totalmente retroativa. O algoritmo *força bruta* se trata de uma modificação da implementação utilizada para a pilha parcialmente retroativa, com a diferença que, assim que uma operação de consulta é realizada no tempo t, somente as operações anteriores ao tempo t são consideradas para a obtenção da resposta.

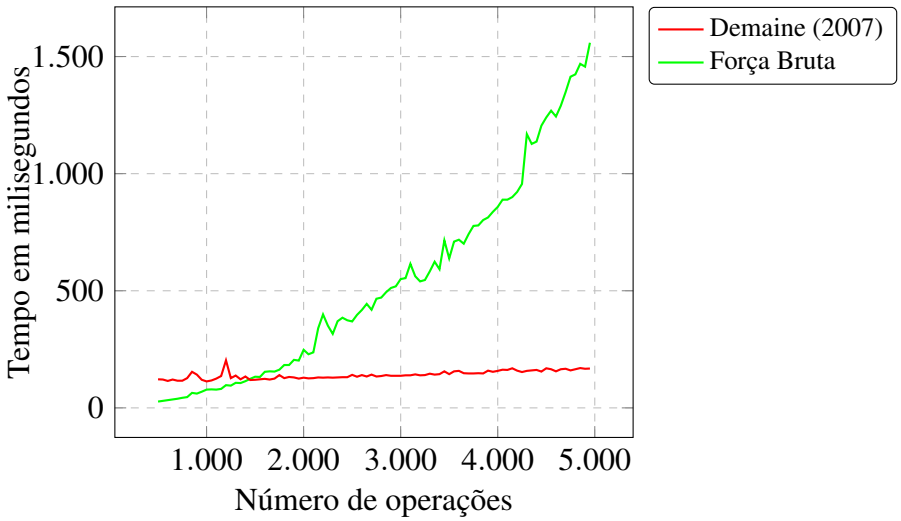

Figura 5.7 Teste de velocidade de execução de uma pilha totalmente retroativa.

Na Figura 5.7, mais uma vez, pode-se observar a vantagem na utilização da estrutura totalmente retroativa com relação à sua versão ingênua. Para casos com poucas operações, o algoritmo trivial é mais veloz que a sua versão retroativa, pois a constante implícita na

complexidade do algoritmo retroativo é alta. Porém, à medida que o número de operações cresce, a pilha totalmente retroativa é mais rápida que a sua implementação trivial.

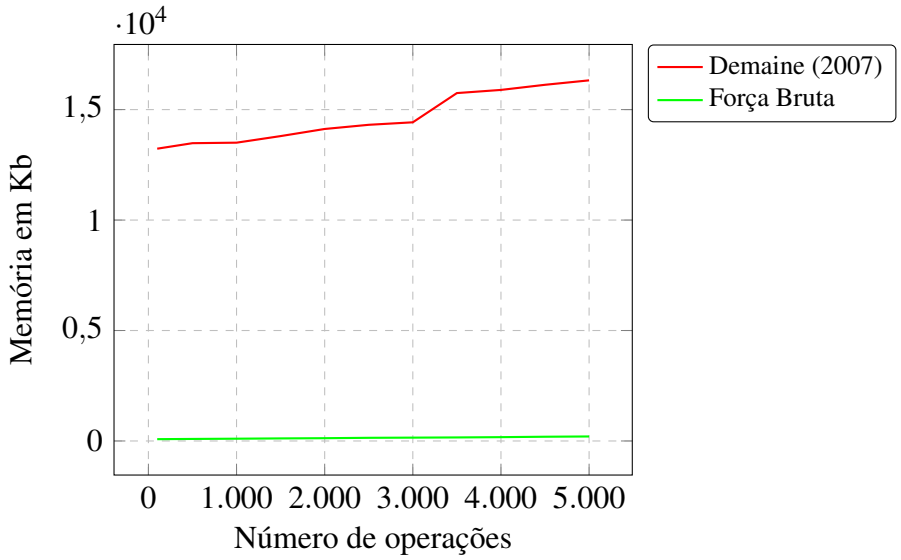

Figura 5.8 Consumo de memória de uma pilha totalmente retroativa. Fonte: O autor

Na Figura 5.8 são apresentados os resultados relacionados ao consumo de memória na execução dos mesmos casos de teste utilizados na Figura 5.7. Novamente, existe uma diferença considerável entre o consumo de memória dos dois algoritmos. O algoritmo *força bruta* consome memória proporcional ao número de operações realizadas, enquanto o algoritmo implementado, baseado na solução proposta por Demaine *et al.* (2007) para a manutenção de uma pilha totalmente retroativa de maneira otimizada, consome memória proporcional ao tamanho da linha do tempo em que a estrutura está inserida.

5.3 Execução dos testes em uma fila de prioridade

Foram gerados testes aleatórios para avaliação das versões retroativas da fila de prioridade. Em cada conjunto de testes foram gerados cinco tipos de operações distintas:

- *Insert(t, Push(x))*: insere o elemento x na fila de prioridade no tempo t;
- *Insert(t, Pop())*: remove o menor elemento da fila de prioridade no tempo t;
- *Delete(t, Push(x))*: remove a operação de inserção do elemento x no tempo t;
- *Delete(t, Pop())*: remove a operação *Pop* realizada na estrutura no tempo t;

- *GetPeak(t)*: retorna o menor elemento na estrutura no tempo t. Na fila de prioridade parcialmente retroativa, t é sempre o tempo presente.

As operações foram geradas mantendo-se a consistência da estrutura através de sua linha temporal do ponto de vista retroativo. Por exemplo, uma operação do tipo *Delete(t, Push(x))* não será gerada caso não exista outra operação do tipo *Insert(t, Push(x))* previamente executada na estrutura. Além disso, não existirão operações que sejam inconsistentes com relação à estrutura em si, como, por exemplo, a execução de uma operação de consulta *GetPeak(t)* em uma fila de prioridade vazia.

Os arquivos gerados para teste da estrutura podem ser encontrados no repositório referente à dissertação no seguinte link https://github.com/juniorandrade1/Master/tree/master/tests/Datasets/Priority_Queue. Cada arquivo contém, inicialmente, um inteiro n correspondente ao número de operações realizadas na estrutura, seguido de n linhas contendo as operações referentes à fila de prioridade retroativa no formato anteriormente descrito. Mais detalhes sobre o formato das operações nos arquivos de teste podem ser encontrados no link anteriormente citado.

5.3.1 Testes em uma fila de prioridade parcialmente retroativa

A seguir, são apresentados os testes para a fila de prioridade parcialmente retroativa. Foram testados dois tipos de algoritmo: *força bruta*, em que todas as operações são armazenadas ordenadas por tempo em uma árvore binária de busca balanceada, e, a cada consulta, todas as operações são executadas de maneira ordenada até o tempo presente, e o algoritmo *Demaine (2007)*, que consiste na implementação da retroatividade parcial proposta por Demaine *et al.* (2007) [13].

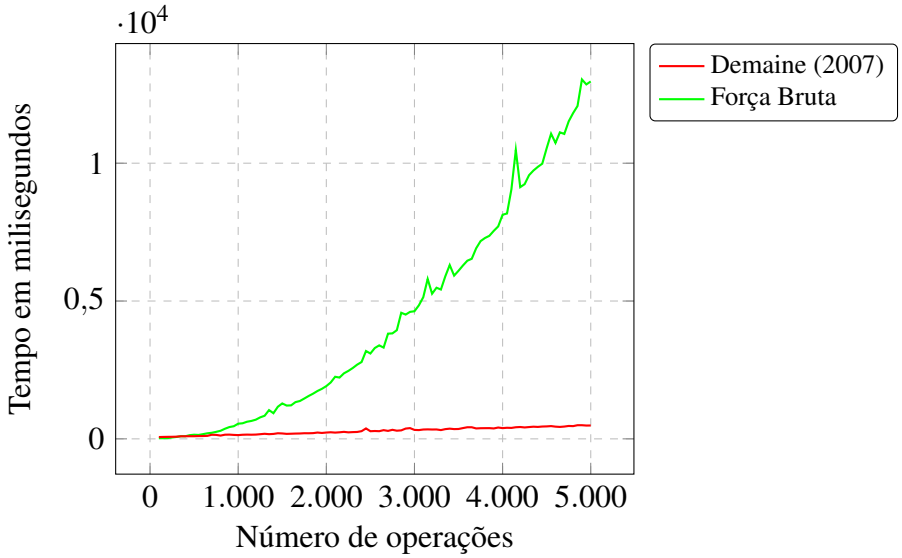

Figura 5.9 Teste de velocidade de execução de uma fila de prioridade parcialmente retroativa. Fonte: O autor.

Na Figura 5.9, é possível observar o gráfico relacionado aos testes em uma fila de prioridade parcialmente retroativa. Os testes de velocidade mostraram que, no caso geral, a fila de prioridade parcialmente retroativa apresentada por Demaine *et al.* (2007) é consideravelmente mais rápida que sua implementação *força bruta*. A partir de mil operações realizadas já é possível que o algoritmo *Demaine (2007)* seja, em média, cinco vezes mais rápido que o algoritmo força bruta. Para casos próximos a 5000 operações, o desempenho da abordagem de *Demaine (2007)* chega a ser 25 vezes mais rápido.

Na Figura 5.10 é apresentado o resultado para o teste de memória das implementações da fila de prioridade parcialmente retroativa. O algoritmo *Demaine (2007)* tem uma curva de crescimento um pouco mais inclinada que a implementação *força bruta* da estrutura. Isso acontece porque na implementação proposta por Demaine *et al.* (2007), é necessário o armazenamento de duas árvores binárias balanceadas q_{del} e q_{now}, ao passo que o algoritmo *força bruta* necessita apenas do armazenamento de um conjunto desse tipo. Essa diferença no consumo de memória também é agravada, uma vez que, no geral, cada nó de uma árvore binária contém uma série de outros atributos, como seus ponteiros para os seus descendentes, além de armazenar o próprio objeto inserido.

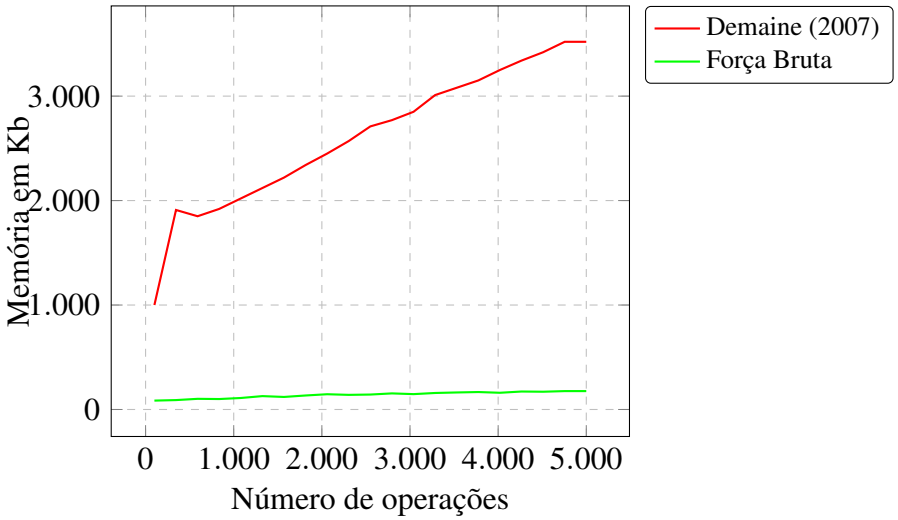

Figura 5.10 Consumo de memória de uma fila de prioridade parcialmente retroativa. Fonte: O autor.

5.3.2 Testes em uma fila de prioridade totalmente retroativa

A seguir são apresentados os testes de três algoritmos distintos para a implementação da fila de prioridade totalmente retroativa. O algoritmo *força bruta* corresponde à mesma solução apresentada nos testes da versão parcialmente retroativa da estrutura, diferenciando-se apenas no momento da operação *GetPeak(t)*, em que somente as modificações realizadas até o tempo t são consideradas. A segunda implementação proposta (*Demaine (2007)*) consiste na solução apresentada por Demaine *et al.* (2007) [13], que possibilita a transformação de uma estrutura parcialmente retroativa para uma totalmente retroativa consumindo um fator multiplicativo extra, tanto de espaço quanto de memória, de $O(\sqrt{m})$. O terceiro algoritmo (*Demaine (2015)*) [14] corresponde à implementação da fila de prioridade totalmente retroativa por meio da utilização da *checkpoint tree*.

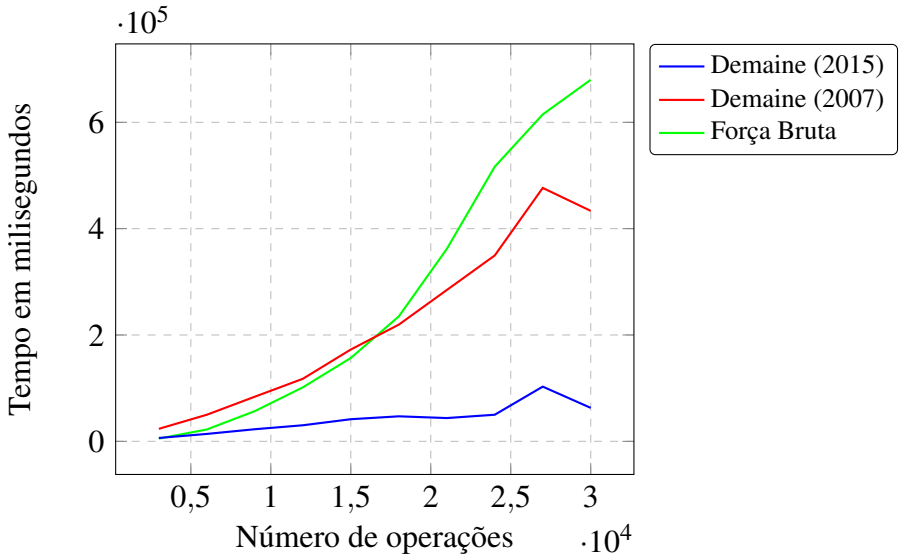

Figura 5.11 Teste de velocidade de execução de uma fila de prioridade totalmente retroativa. Fonte: O autor.

Como analisado anteriormente, o algoritmo *Demaine (2007)* tem complexidade $O(\sqrt{n}\lg n)$ para as atualizações, e $O(\sqrt{m}\lg n)$ para as consultas. Portanto, para as operações de inserção, o algoritmo proposto tende a ser mais lento, porém, para as operações de consulta, o algoritmo retroativo é mais rápido para casos maiores. O algoritmo *Demaine (2007)* também contém uma constante alta na complexidade, por isso, em casos com poucas operações/elementos, o algoritmo força bruta é mais rápido que a solução proposta.

O algoritmo *Demaine (2007)* é mais "estável" do ponto de vista temporal, uma vez que a curva de crescimento do algoritmo *força bruta* é mais inclinada que a solução retroativa. Já o algoritmo *Demaine (2015)* apresentou um desempenho consideravelmente melhor que as implementações anteriores, sendo mais veloz desde as execuções dos algoritmos com o número de elementos menor. Isso tem relação com a velocidade da execução das operações de atualização na estrutura, que é realizada em $O(\lg^2 m)$ por operação, em vista da complexidade $O(\sqrt{m}\cdot\lg m)$ no caso do algoritmo genérico para transformação de uma versão parcialmente retroativa para sua versão totalmente retroativa com $\sqrt{m}$ *checkpoints*.

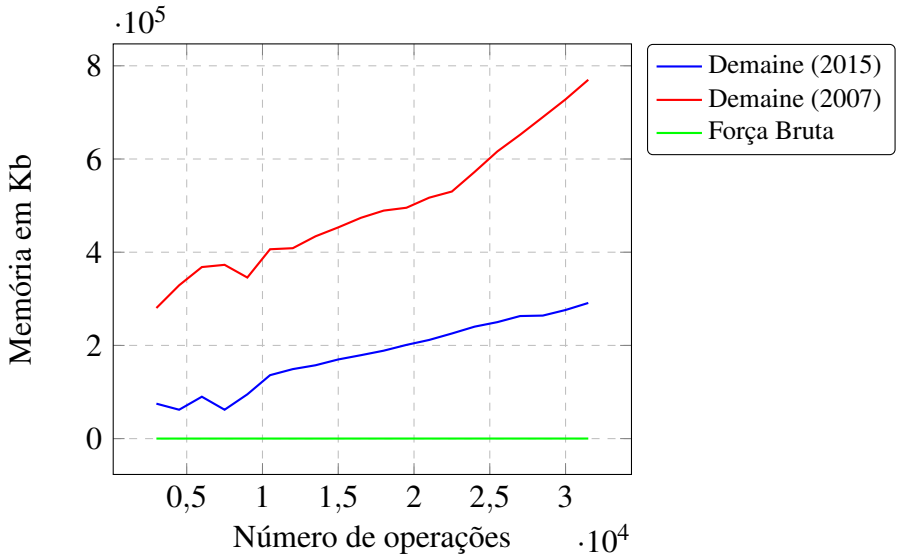

Figura 5.12 Consumo de memória de uma fila de prioridade totalmente retroativa. Fonte: O autor

Na Figura 5.12, são apresentados os testes referentes ao consumo de memória nas implementações da fila de prioridade totalmente retroativa. O algoritmo *força bruta* tem um consumo de memória ínfimo comparado ao consumo de memória dos algoritmos mais elaborados. O algoritmo *força bruta* somente mantém as operações ordenadas, consumindo muito tempo de processamento nas consultas, mas com um baixo consumo de memória. O algoritmo *Demaine (2007)* consome muita memória, pois além de armazenar $\sqrt{m}$ filas de prioridade parcialmente retroativas, ao realizar uma operação de atualização, no pior caso, o objeto atualizado será inserido em $\sqrt{m}$ filas de prioridade parcialmente retroativas. O algoritmo *Demaine (2015)* mantém uma *checkpoint-tree* que, no pior caso de uma atualização, modifica $\log m$ filas de prioridade parcialmente retroativas, o que explica a diferença de consumo de memória nesses dois algoritmos.

5.4 Testes em uma árvore geradora mínima totalmente retroativa

Além do algoritmo proposto, foram implementadas outras duas soluções para o problema da árvore geradora mínima totalmente retroativa. As soluções escolhidas consistem em modificações do algoritmo de Prim e Kruskal. Os seguintes algoritmos foram testados:

1. Algoritmo LCT + $\sqrt{m}$ *totalmente retroativo*: blocos de $\sqrt{m}$ e a *link-cut tree* no início de cada bloco para reconstrução da resposta nas consultas;

2. Algoritmo de Kruskal *totalmente retroativo*: re-execução do algoritmo de Kruskal em cada operação de consulta;

3. Algoritmo de Prim *totalmente retroativo*: re-execução do algoritmo de Prim para calcular as operações de consulta.

O primeiro algoritmo consiste na solução proposta por essa dissertação. Seja a linha do tempo de tamanho m, armazenar $b = \lfloor \sqrt{m} \rfloor$ árvores geradoras mínimas representando $B = \{T_0^*, T_b^*, T_{2b}^*, \ldots, T_{\lfloor m/b \rfloor}^*\}$. Cada MST é armazenada em uma *link-cut-tree* e permite a adição de arestas em tempo amortizado de $O(\lg |V(G)|)$.

O algoritmo *totalmente retroativo* baseado na solução proposta por Kruskal para a árvore geradora mínima consiste em armazenar todas as arestas adicionadas em G ordenadas por tempo em memória. Em cada atualização, é inserida a aresta adicionada a esse conjunto. Este conjunto pode ser mantido por uma árvore binária de busca balanceada em tempo logarítmico no número de elementos. A operação de consulta $GetMST(t)$ é realizada obtendo todas as arestas adicionadas antes do tempo t e executar o algoritmo de Kruskal considerando somente essas arestas. No pior caso, todas as operações de consulta serão realizadas no tempo mais recente, gerando a execução do algoritmo de Kruskal no conjunto completo das arestas adicionadas, o que leva o algoritmo a uma complexidade temporal de $O(|E(G)| \cdot \lg |V(G)|)$ em cada operação de consulta.

No algoritmo *totalmente retroativo* baseado na solução proposta por Prim para o problema da MST, novamente são mantidas todas as arestas adicionadas na MST ordenadas por tempo. A diferença entre esse algoritmo e a solução previamente apresentada ocorre quando uma consulta é realizada. Nesse caso não se utiliza o algoritmo de Kruskal, mas sim o algoritmo de Prim. Utilizando essa abordagem, cada consulta é executada em tempo $O(|E(G)| \cdot \lg |V(G)|)$ no pior caso.

Nos algoritmos baseados nas soluções de Prim e Kruskal, não são armazenados os grafos intermediários em memória, como ocorre no algoritmo $LCT + \sqrt{m}$. As arestas são armazenadas em memória ordenadas por tempo de inserção e o grafo só é gerado quando uma operação de consulta é realizada.

Teoricamente falando, os algoritmos 2 e 3 são mais rápidos com relação à operação de atualização, uma vez que o primeiro algoritmo precisa atualizar todos os grafos armazenados após o tempo da adição da aresta. Contudo, nas consultas em grafos com muitas arestas, o primeiro algoritmo tende a ser mais veloz que os outros, pois não necessita processar todas as arestas sempre que uma operação desse tipo é efetuada.

Os testes mostraram um desempenho similar em grafos com um baixo número de vértices. No entanto, assim que o número de vértices aumentou, a performance do algoritmo proposto foi consideravelmente melhor do ponto de vista temporal.

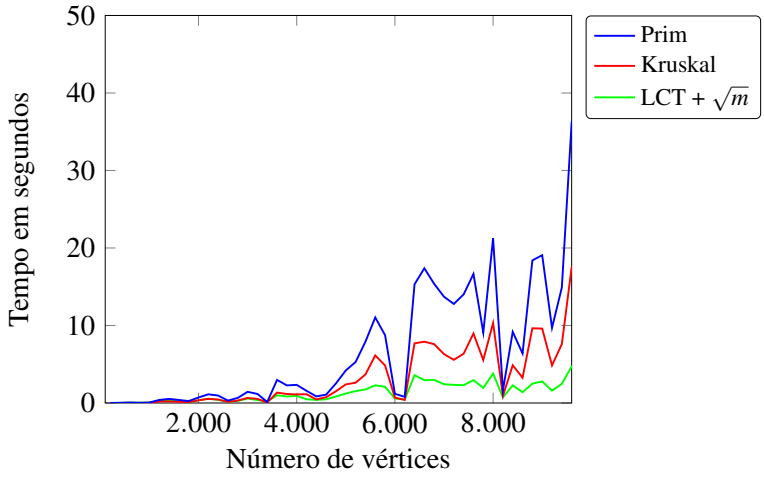

Figura 5.13 Grafos aleatórios com um grande número de operações (consulta e atualização) e com o tamanho da linha do tempo fixos.

Como pode ser visto na Figura 5.13, em casos pequenos, os algoritmos de Kruskal e Prim apresentaram uma performance similar à performance do algoritmo proposto; contudo, em casos maiores, a vantagem do algoritmo $LCT + \sqrt{m}$ é alta. Em alguns casos, a vantagem entre o desempenho do algoritmo proposto e os algoritmos de Prim e Kruskal *totalmente retroativos* é próxima. Isso é explicado pela aleatoriedade no número de operações de modificação. Nos casos em que o número de consultas é pequeno, a média de consumo temporal é baixa em todos os algoritmos. Sendo assim, foram re-executados testes fixando um número maior de operações realizadas.

A Figura 5.14 mostra que, apesar do alto número de operações retroativas realizadas, o algoritmo proposto mantém seu bom desempenho temporal. Quanto maior o número de vértices e de operações realizadas, maior a diferença do desempenho dos algoritmos testados. Contudo, existem algumas grandes variações de desempenho nos algoritmos de Kruskal e Prim, que podem ser explicadas pelo modo com que os algoritmos foram desenvolvidos. Nos algoritmos de Kruskal e Prim, assim que uma árvore geradora mínima é construída, os algoritmos finalizam a sua execução retornando o resultado. Por exemplo, no algoritmo de Kruskal, isso acontece quando o algoritmo realiza a união de $|V(G)| - 1$ vértices. Quando isso ocorre, o grafo é uma árvore, e, pela maneira que essa árvore é construída no algoritmo de Prim, esta árvore é uma MST.

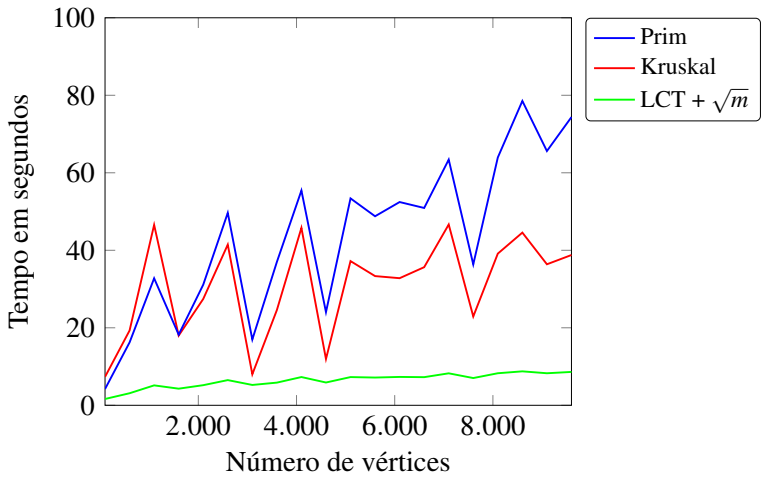

Figura 5.14 Grafos aleatórios com um grande número de operações (consulta e atualização) e tamanho da linha do tempo fixa.

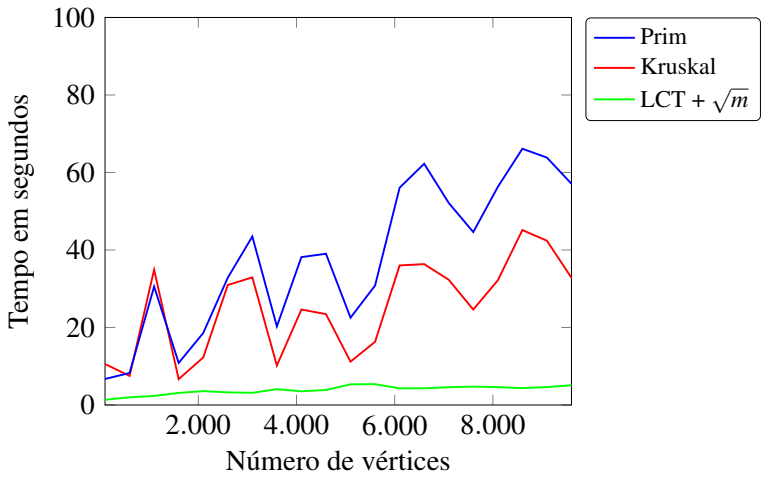

Figura 5.15 Operações de consulta no tempo mais recente em grafos totalmente aleatórios com um número fixo de operações (consulta e atualização)

A Figura 5.15 mostra que, com as consultas todas sendo realizadas no fim da linha temporal (tempo presente), apesar da variação no algoritmo de Kruskal e Prim, a diferença de performance entre o algoritmo proposto contra a modificação dos algoritmos de Prim e Kruskal aumentou. Isto pode ser explicado pelo fato de que no fim da linha temporal, o segundo e o terceiro algoritmos precisam processar um número muito maior de arestas que a solução que utiliza a *link-cut tree* e a redução por $\sqrt{m}$.

Capítulo 6

Conclusão

Estruturas de dados retroativas são estruturas que permitem a consulta e a modificação das suas varias versões através do tempo. Essas estruturas podem ser utilizadas tanto na dinamização de algoritmos como, por exemplo, o algoritmo de Dijkstra [52], quanto em problemas geométricos, como na clonagem de diagramas de Voronoi [16] e a consulta de predecessores em segmentos de reta no plano bidimensional.

Neste trabalho apresentou-se os subsídios teóricos relacionados a retroatividade e algumas aplicações nos quais as estruturas de dados retroativas podem ser utilizadas.

6.1 Contribuições do trabalho

No Capítulo 2, apresentou-se uma técnica geral para a transformação de uma estrutura parcialmente retroativa para uma estrutura totalmente retroativa, permitindo a transformação genérica de uma estrutura de dados para sua versão totalmente retroativa com um fator multiplicativo $\sqrt{m}$.

No Capítulo 3, foram realizadas implementações das estruturas fila, pilha, fila de prioridade e união de conjuntos *(union-find)* em suas formas retroativas na linguagem *C++*. Também apresentou-se a teoria sobre todas as estruturas retroativas, assim como de outras estruturas auxiliares necessárias para a sua implementação. Além disso, para cada uma dessas estruturas, foram realizados testes de desempenho, comparando-as com implementações menos eficientes.

Na estrutura fila, foram aplicadas árvores binárias de busca balanceadas para a implementação de todas as suas versões retroativas, incluindo a versão não-consistente da estrutura. Para a implementação das suas versões retroativas foram armazenadas duas árvores, uma contendo as operações de *Enqueue* enquanto a outra mantinha as operações de *Dequeue*.

Assim, obteve-se uma estrutura que suporta as operações, em um escopo retroativo, em tempo logarítmico no número de operações realizadas.

Para a implementação das versões parcialmente e totalmente retroativa da pilha, utilizou-se a estrutura de dados árvore de segmentos, em que cada vértice dessa estrutura contém informações sobre um intervalo contínuo de tempo. Neste caso, cada operação *Push* realizada no tempo t incrementava a posição t em uma unidade, enquanto as operações Pop realizadas no tempo t decrementavam t em uma unidade. Uma árvore de segmentos foi utilizada para a implementação dessas operações de maneira otimizada, obtendo, assim, uma complexidade temporal de $O(\lg m)$ por operação de modificação ou consulta.

Implementou-se a fila de prioridade parcialmente retroativa através de duas árvores binárias de busca. Elas mantém os conjuntos Q_{now}, que consiste nos elementos presentes na estrutura no tempo mais recente, e Q_{del}, que contém os elementos que foram removidos por uma operação Pop nessa estrutura. Com esses dois conjuntos foi possível obter a estrutura fila de prioridade parcialmente retroativa em complexidade temporal de $O(\lg n)$ por operação.

A versão totalmente retroativa da fila de prioridade foi implementada de duas maneiras, ambas usando as técnicas propostas por Demaine *et al.* [13, 14]. A primeira técnica utilizada foi a transformação da versão parcialmente retroativa da estrutura para sua versão totalmente retroativa, utilizando a manutenção de cópias da estrutura parcialmente retroativa em pontos estratégicos da linha temporal. Essas cópias são espalhadas pela linha temporal da estrutura, em intervalos de $\sqrt{m}$, e permitem a consulta e atualização da estrutura em tempo $\sqrt{m} \cdot \lg n$. Já a segunda solução implementada para a estrutura consiste na utilização da propriedade de que duas filas parcialmente retroativas são *time-fusible*, e, por isso, é possível combiná-las, gerando um novo objeto que compreende a união dos intervalos temporais que essas estruturas representam. Assim, é possível gerar uma árvore de segmentos especial *(checkpoint-tree)* e com isso obter uma solução com complexidade poli-logarítmica.

Os testes empíricos realizados nessas estruturas constataram um desempenho temporal mais otimizado quando comparado a implementações diretas, em que as operações são refeitas até o tempo desejado a cada consulta.

Ao final, no Capítulo 4, foram apresentadas duas aplicações das técnicas para geração das estruturas retroativas: o problema do caminho mínimo em grafos dinâmicos, em que foi utilizada uma fila de prioridade com retroatividade não-consistente para a solução; e o problema da árvore geradora retroativa, em que utilizou-se uma técnica similar à generalização para obter acesso às *MST*s existentes na linha temporal de um grafo dinâmico.

As principais contribuições do trabalho podem ser resumidas em:

- Compilação da lista dos algoritmos de retroatividade nas estruturas de dados mais clássicas;

- Implementação das estruturas de dados retroativas na linguagem C++;
- Criação de um repositório aberto à comunidade com essas implementações;
- Proposição de solução para o problema da árvore geradora mínima dinâmica;
- Comparação de implementações diferentes das estruturas de dados retroativas, com relação aos desempenhos espacial e temporal;
- Geração de um material relacionado a estruturas de dados retroativas em português, tendo em vista a escassez de material sobre esse assunto nesse idioma.

6.2 Sugestões para trabalhos futuros

Neste trabalho, foram escolhidas árvores binárias balanceadas aleatoriamente (*cartesian tree*) para a manutenção de conjuntos de dados. Elas foram escolhidas pela facilidade de sua manutenção através da execução de operações, como unir duas árvores binárias distintas, ou dividir uma árvore binária, dado um valor. Essa árvore possui uma complexidade temporal amortizada de $O(\lg n)$, porém, segundo Heger [30], existem árvores que performam de maneira mais otimizada que a *cartesian tree* em um contexto geral, como, por exemplo, *AVL's* [1] e árvores rubro-negras [6]. Portanto, seria pertinente a implementação dessas estruturas utilizando alguma dessas árvores citadas ao invés da utilização da *cartesian tree*.

Além disso, seria relevante o aprofundamento do estudo dessas estruturas do ponto de vista geométrico, uma vez que o fator "tempo" em estruturas retroativas pode ser visto, dependendo da situação, como um eixo adicional em um sistema de coordenadas cartesiano.

Referências Bibliográficas

[1] Adel'son-Vel'skii, G. M. and Landis, E. M. (1962). An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences.

[2] Allison, R. J., Goodwin, S. P., Parker, R. J., Portegies Zwart, S. F., De Grijs, R., and Kouwenhoven, M. (2009). Using the minimum spanning tree to trace mass segregation. *Monthly Notices of the Royal Astronomical Society*, 395(3):1449–1454.

[3] Aragon, C. R. and Seidel, R. G. (1989). Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science*, pages 540–545. IEEE.

[4] Arora, S. (1998). Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems. *J. ACM*, 45(5):753–782.

[5] Assunção, R. M., Neves, M. C., Câmara, G., and da Costa Freitas, C. (2006). Efficient regionalization techniques for socio-economic geographical units using minimum spanning trees. *International Journal of Geographical Information Science*, 20(7):797–811.

[6] Bayer, R. (1972). Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306.

[7] Bertocchi, F., Bergamo, P., Mazzini, G., and Zorzi, M. (2003). Performance comparison of routing protocols for ad hoc networks. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 2, pages 1033–1037. IEEE.

[8] Boruvka, O. (1926). O jistém problému minimálním.

[9] Chen, L., Demaine, E. D., Gu, Y., Williams, V. V., Xu, Y., and Yu, Y. (2018). Nearly optimal separation between partially and fully retroactive data structures. *arXiv preprint arXiv:1804.06932*.

[10] Christofides, N. (1976). Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

[11] Chu, C. and Wong, Y.-C. (2005). Fast and accurate rectilinear steiner minimal tree algorithm for VLSI design. In *Proceedings of the 2005 international symposium on Physical design*, pages 28–35. ACM.

[12] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

[13] Demaine, E., Iacono, J., and Langerman, S. (2007). Retroactive Data Structures. *ACM Transactions on Algorithms*, 3(2). 20 pp.

[14] Demaine, E., Kaler, T., Liu, Q., Sidford, A., and Yedidia, A. (2015). Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing. In *Proc. of the 14th Workshop on Algorithms and Data Structures (WADS)*, volume 9214 of *Lecture Notes in Computer Science*, pages 263–275.

[15] Demetrescu, C. and Italiano, G. F. (2004). A new approach to dynamic all pairs shortest paths. *Journal of the ACM (JACM)*, 51(6):968–992.

[16] Dickerson, M. T., Eppstein, D., and Goodrich, M. T. (2010). Cloning voronoi diagrams via retroactive data structures. In *European Symposium on Algorithms*, pages 362–373. Springer.

[17] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1:269–271.

[18] Driscoll, J., Sarnak, N., Sleator, D., and Tarjan, R. (1989). Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124.

[19] Eklund, P. W., Kirkby, S., and Pollitt, S. (1996). A dynamic multi-source dijkstra's algorithm for vehicle routing. In *1996 Australian New Zealand Conference on Intelligent Information Systems. Proceedings. ANZIIS 96*, pages 329–333. IEEE.

[20] Eppstein, D. (1994). Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17(2):237–250.

[21] Eppstein, D., Italiano, G. F., Tamassia, R., Tarjan, R. E., Westbrook, J., and Yung, M. (1992). Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 13(1):33–54.

[22] Even, S. and Shiloach, Y. (1981). An On-Line Edge-Deletion Problem. *Journal of the ACM*, 28(1):1–4.

[23] Fan, D. and Shi, P. (2010). Improvement of dijkstra's algorithm and its application in route planning. In *2010 seventh international conference on fuzzy systems and knowledge discovery*, volume 4, pages 1901–1904. IEEE.

[24] Gallo, G. (1980). Reoptimization procedures in shortest path problem. *Rivista di matematica per le scienze economiche e sociali*, 3(1):3–13.

[25] Garg, D. et al. (2019). A retroactive approach for dynamic shortest path problem. *National Academy science letters*, 42(1):25–32.

[26] Gower, J. C. and Ross, G. J. (1969). Minimum spanning trees and single linkage cluster analysis. *Applied statistics*, pages 54–64.

[27] Granot, D. and Huberman, G. (1981). Minimum cost spanning tree games. *Mathematical programming*, 21(1):1–18.

[28] Guibas, L. J. and Sedgewick, R. (1978). A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE.

[29] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

[30] Heger, D. A. (2004). A disquisition on the performance behavior of binary search tree data structures. *European Journal for the Informatics Professional*, 5(5):67–75.

[31] Henzinger, M., Krinninger, S., Nanongkai, D., and Saranurak, T. (2015). Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 21–30, New York, NY, USA. Association for Computing Machinery.

[32] Henzinger, M. and Wu, X. (2019). Upper and lower bounds for fully retroactive graph problems.

[33] Huang, G., Li, X., and He, J. (2006). Dynamic minimal spanning tree routing protocol for large wireless sensor networks. In *Industrial Electronics and Applications, 2006 1ST IEEE Conference on*, pages 1–5. IEEE.

[34] Kang, H. I., Lee, B., and Kim, K. (2008). Path planning algorithm using the particle swarm optimization and the improved Dijkstra algorithm. In *Computational Intelligence and Industrial Application, 2008. PACIIA'08. Pacific-Asia Workshop on*.

[35] Kaplan, H. (2004). *Handbook on Data Structures and Applications*, chapter 31, Persistent Data Structures. CRC Press. 27 pp.

[36] Khamsi, M. A. and Kirk, W. A. (2011). *An introduction to metric spaces and fixed point theory*, volume 53. John Wiley & Sons.

[37] Khan, M., Pandurangan, G., and Kumar, V. A. (2009). Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 20(1):124–139.

[38] Kim, H.-S., Lee, J.-H., and Jeong, Y.-S. (2003). Method for finding shortest path to destination in traffic network using Dijkstra algorithm or Floyd-warshall algorithm.

[39] King, V. and Thorup, M. (2001). A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *International Computing and Combinatorics Conference*, pages 268–277. Springer.

[40] Kleinberg, J. and Tardos, E. (2006). *Algorithm design*. Pearson Education India.

[41] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

[42] Lawler, E. L., Lenstra, J. K., Kan, A. R., Shmoys, D. B., et al. (1985). *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley New York.

[43] Ma, B., Hero III, A. O., Gorman, J. D., and Michel, O. J. (2000). Image Registration with Minimum Spanning Tree Algorithm. In *ICIP*, pages 481–484.

[44] Martínez, C. and Roura, S. (1998). Randomized binary search trees. *Journal of the ACM (JACM)*, 45(2):288–323.

[45] McCreight, E. M. (1985). Priority search trees. *SIAM Journal on Computing*, 14(2):257–276.

[46] Noto, M. and Sato, H. (2000). A method for the shortest path search by extended Dijkstra algorithm. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*.

[47] Okasaki, C. (1999). *Purely Functional Data Structures*. Cambridge University Press.

[48] Peyer, S., Rautenbach, D., and Vygen, J. (2009). A generalization of dijkstra's shortest path algorithm with applications to vlsi routing. *Journal of Discrete Algorithms*, 7(4):377–390.

[49] Ramalingam, G. and Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305.

[50] Sleator, D. and Tarjan, R. (1985). Self-adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686.

[51] Sleator, D. D. and Tarjan, R. E. (1983). A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391.

[52] Sunita and Garg, D. (2018). Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. *Journal of King Saud*.

[53] Tangwongsan, G. E. and Blelloch, U. A. (2007). Non-oblivious retroactive data structures. Technical report.

[54] Tarjan, R. and Fredman, M. L. (1984). Fibonacci heaps and their uses in improved network optimization algorithms. In *25th Annual Symposium on Foundations of Computer Science*, pages 338–346.

[55] Upadhyayula, S. and Gupta, S. K. (2007). Spanning tree based algorithms for low latency and energy efficient data aggregation enhanced convergecast (dac) in wireless sensor networks. *Ad Hoc Networks*, 5(5):626–648.

[56] Van Kreveld, M., Schwarzkopf, O., de Berg, M., and Overmars, M. (2000). *Computational geometry algorithms and applications*. Springer.

[57] Wu, Y., Bhat, P. R., Close, T. J., and Lonardi, S. (2008). Efficient and accurate construction of genetic linkage maps from the minimum spanning tree of a graph. *PLoS genetics*, 4(10):e1000212.

[58] Wu, Z. and Leahy, R. (1993). An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 15(11):1101–1113.

[59] Yoon, J., Blumer, A., and Lee, K. (2006). An algorithm for modularity analysis of directed and weighted biological networks based on edge-betweenness centrality. *Bioinformatics*, 22(24):3106–3108.

[60] Zeng, W. and Church, R. L. (2009). Finding shortest paths on real road networks: the case for A. *International journal of geographical information science*, 23(4):531–543.

[61] Ziviani, N. et al. (2004). *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton.