Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Beatriz Figueiredo Marouelli

Um estudo sobre estruturas de dados retroativas

São Paulo
Dezembro de 2019

Um estudo sobre estruturas de dados retroativas

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisora: Cristina G. Fernandes

São Paulo
Dezembro de 2019

Resumo

O paradigma da retroatividade introduz uma maneira de controlar o histórico de operações realizadas sobre determinada estrutura de dados, modificando sua implementação tradicional para incluir a associação de cada operação a um instante de tempo, possibilitando a inclusão ou exclusão de operações no passado. O primeiro estudo sobre esse paradigma foi produzido por Demaine, Iacono e Langerman, que elaboraram um artigo apresentando a definição formal de retroatividade, uma extensa demonstração sobre a não existência de um método genérico para converter qualquer estrutura numa versão retroativa, e provaram que as seguintes estruturas possuem versão retroativa eficiente: fila, fila dupla, union-find e fila de prioridades. Este trabalho pretende produzir um texto introdutório para apresentar esse tipo de estrutura e as técnicas envolvidas.

Abstract

The retroactivity paradigm introduces a way to control the history of operations performed on a given data structure, modifying its traditional implementation to include the association of each operation with a given time, enabling the inclusion or exclusion of operations in the past. The first study of this paradigm was produced by Demaine, Iacono, and Langerman. They present the formal definition of retroactivity, an extensive demonstration of the non-existence of a generic method for converting any structure to a retroactive version, and prove that the following structures have an efficient retroactive version: queue, deque, union-find and priority queue. This work intends to produce an introductory text to present this type of structure and the techniques involved.

Sumário

1 Introdução **1**

2 Preliminares **3**

2.1 A lista de atualizações 3

2.1.1 Retroatividade parcial 3

2.1.2 Retroatividade total 4

2.2 Premissas das implementações 4

2.3 Árvore AVL 4

2.4 Versão tradicional das estruturas 6

2.4.1 Fila 6

2.4.2 Pilha 6

2.4.3 Fila de prioridade 7

3 Fila Parcialmente Retroativa **9**

3.1 Interface e comentários da implementação 9

3.2 Estrutura interna: lista duplamente ligada com árvore auxiliar 10

3.3 Implementação da fila parcialmente retroativa 13

3.4 Exemplo de execução 14

3.5 Análise do consumo de tempo das operações 15

4 Fila Totalmente Retroativa **17**

4.1 Exemplo abstrato 17

4.2 Interface e comentários de implementação 18

4.3 Estrutura interna: árvore de operações 18

4.4 Implementação da fila totalmente retroativa 20

4.5 Exemplo de execução 22

4.6 Análise do consumo de tempo das operações 23

5 Pilha Totalmente Retroativa **25**

5.1 Apresentação 25

5.2 Estrutura interna: árvore de operações 26

5.3 Implementação da pilha totalmente retroativa 28

5.4 Exemplo de execução 30
5.5 Análise de consumo de tempo 31

6 Fila de Prioridade Parcialmente Retroativa 33
6.1 Definições 33
6.2 Interface e comentários de implementação 34
6.3 Estruturas internas: árvore de operações e árvore de valores presentes 35
6.4 Implementação da fila de prioridade parcialmente retroativa 36
6.5 Exemplo de execução 38
6.6 Análise de consumo de tempo das operações 40

7 Possíveis Aplicações 41

8 Conclusões 43

Referências Bibliográficas 45

Capítulo 1

Introdução

"Uma pessoa sai de casa às 06:30 da manhã e deixa seu quarto bagunçado. Ela volta à noite, e sente que devia ter arrumado o quarto antes, para lhe poupar o trabalho agora. Imagine se estivesse em seu poder inserir uma ação *arruma-quarto* às 06:15 daquela manhã..."

A situação descrita é completamente fictícia, e infelizmente ainda não é possível alterar o passado humano. No entanto, computacionalmente falando, parece razoável existir um mecanismo capaz de provocar mudanças de estado alterando o histórico de ações.

Uma estrutura de dados determina como os valores de sua coleção se relacionam, e define um conjunto de operações que podem ser aplicadas nesses dados.

A proposta da estrutura de dados retroativa é ser esse mecanismo que controla as operações antecedentes. Ela é classificada como **temporal**, isto é, suas operações estão intimamente ligadas a uma linha de tempo, e ela permite que seu passado seja modificado, e em alguns casos, consultado.

A principal fonte de informações deste estudo foi o artigo *Retroactive data structures*, escrito por Demaine *et al.*(4). Nele seus autores demonstram que não há uma maneira generalizável de obter a versão retroativa de uma estrutura de dados. Logo, cada versão deve ser pensada individualmente e usar estratégias personalizadas.

Além desse artigo foram encontradas outras referências ao assunto de retroatividade. No campo de geometria computacional, Dickerson, Eppstein e Goodrich (5) apresentam a clonagem de diagramas de Voronoi através de estruturas retroativas, e Goodrich e Simons (6) descrevem uma estrutura retroativa para resolver os problemas *range search* e *nearest neighbor*.

Num artigo complementar àquele que introduz a retroatividade, Demaine *et al.* (3) implementam uma fila de prioridades totalmente retroativa. Decorrente desse complemento, Sunita e Garg (7) mostram como dinamizar o algoritmo de Dijkstra usando uma fila de prioridades totalmente retroativa.

Este texto se propõe a esclarecer as estratégias, que foram descritas no artigo de referência principal, usadas para implementar a versão retroativa de algumas estruturas básicas.

O capítulo 2 pode ser dividido em duas partes: definições básicas sobre retroatividade

e descrição da versão tradicional das estruturas. Essa segunda parte pode ser consultada à medida que for necessário.

Os capítulos seguintes descrevem as ideias por trás das estruturas retroativas estudadas: a fila, a pilha e a fila de prioridade. Por fim, o capítulo 6 exemplifica alguns contextos nos quais essas estruturas podem ser úteis.

O código fonte das estruturas desenvolvidas está disponibilizado no GitHub através do repositório biamarou/MAC0499.

Capítulo 2

Preliminares

2.1 A lista de atualizações

Um problema estrutural pode ser facilmente rearranjado sob a perspectiva retroativa. Em geral, uma estrutura de dados recebe operações de modificação (*update*) e operações de consulta (*query*), que são executadas obedecendo uma certa ordem.

Assumindo que apenas uma operação é feita a cada instante de tempo, tem-se a lista $U = [u(t_1), \ldots, u(t_m)]$ de *updates* da estrutura, onde $u(t_i)$ é uma operação de modificação efetuada no instante t_i e $t_1 < t_2 < \cdots < t_m$.

Normalmente, numa estrutura de dados tradicional, as operações são executadas na ordem em que são dadas. Isso equivale a dizer, utilizando a notação apresentada, que a lista U sofre apenas inserções de operações de modificação da estrutura no extremo final e as consultas são sempre realizadas sobre a estrutura resultante de todas as modificações efetuadas até o momento presente.

Nesta nova perspectiva da retroatividade, as operações estão indexadas por um instante de tempo, e isso permite que a lista U possa sofrer inserções ou remoções ao longo de toda sua extensão, assim como as consultas passam a poder se referir a qualquer momento de existência da estrutura. Especificamente:

- *Add(t,u)*: insere na lista U a operação de modificação u no instante t;
- *Remove(t)*: remove da lista U a operação de modificação do instante t;
- *Query(t,q)*: executa a consulta q no estado da estrutura no instante t.

2.1.1 Retroatividade parcial

Uma estrutura é dita *parcialmente retroativa* se ela permite que na lista U sejam inseridas ou removidas operações de modificação em qualquer instante de tempo, ou seja, possivelmente no passado, e não apenas inserções no presente como nas estruturas usuais. As operações de consulta continuam sendo realizadas apenas no presente.

2.1.2 Retroatividade total

Na definição anterior pode-se observar que, como a própria nomenclatura insinua, o controle do histórico é parcial. As operações permitem alterar o passado da estrutura, mas não observá-lo diretamente. Quando uma estrutura é totalmente retroativa, além de conseguir remover ou inserir operações de modificação no passado, também é possível fazer consultas sobre o estado da estrutura em qualquer instante de tempo, não apenas no presente.

2.2 Premissas das implementações

Para implementar estruturas cujas operações estão ligadas a um instante de tempo, foi necessário partir de algumas pressuposições:

- As operações estão associadas a instantes de tempo diferentes;
- O tempo é discreto, ou seja, cada instante é representado por um número inteiro não negativo;
- A consistência na ordem das operações não é verificada; isso é responsabilidade do usuário. Por exemplo, numa pilha espera-se que exista pelo menos uma operação *empilha* efetuada antes de cada operação *desempilha*.

2.3 Árvore AVL

Nos capítulos seguintes são propriamente apresentadas as implementações de algumas estruturas retroativas. Por trás delas estão estruturas de dados já conhecidas, ainda que com algumas mudanças.

A estrutura tradicional utilizada como principal ponto de partida foi a *AVL Tree*, nomeada assim pelas iniciais de seus criadores Adel'son-Vel'skii e Landis (1), por isso será feita uma pequena revisão do seu funcionamento.

Essa estrutura de dados é uma árvore binária de busca balanceada, que possui a seguinte propriedade: as alturas das subárvores filhas de qualquer nó diferem por no máximo um. Se em algum momento essa diferença for maior do que um, é necessário fazer um rebalanceamento através de rotações para restaurar essa propriedade. Originalmente ela apresenta as operações de inserção, remoção e busca, e para essa versão implementada também foi inclusa a operação de ranqueamento.

O ranqueamento não faz parte da configuração original dessa árvore, mas ele se mostrou necessário para implementar as estruturas retroativas, como será mostrado adiante. Ele determina que uma chave ocupa posição k se existem outras $k - 1$ chaves de menor valor na estrutura, onde $k \geqslant 1$. O ranqueamento pode ser utilizado de duas maneiras: obtendo a posição de um dado elemento ou obtendo qual elemento ocupa uma dada posição.

As operações de busca, inserção e remoção custam $O(\lg(n))$ no pior caso, sendo n o número de nós na árvore no momento anterior à operação.

Operação	Pior Caso
Inserção	$O(\lg(n))$
Remoção	$O(\lg(n))$
Busca	$O(\lg(n))$
Ranque	$O(\lg(n))$

Tabela 2.1: *Consumo de tempo das operações na AVL, sendo n o número de nós.*

2.4 Versão tradicional das estruturas

Nos capítulos seguintes são apresentadas as versões retroativas da fila, pilha e fila de prioridade. Para manter um comparativo de desempenho, segue a versão básica dessas estruturas, descritas por Cormen *et al.* (2).

2.4.1 Fila

- Enqueue(value): recebe um elemento e o insere no final da fila.
- Dequeue(): remove e devolve o elemento que está no começo da fila.
- First(): devolve o elemento que está no começo da fila.
- Last(): devolve o elemento que está no final da fila.

Operação	Consumo de tempo
Enqueue	O(1)
Dequeue	O(1)
First	O(1)
Last	O(1)

Tabela 2.2: *Consumo de tempo das operações na fila tradicional.*

2.4.2 Pilha

- Push(value): recebe um elemento e o insere no topo da pilha.
- Pop(): remove e devolve o elemento que está no topo da pilha.
- Top(): devolve o elemento que está no topo da pilha.

Operação	Consumo de tempo
Push	O(1)
Pop	O(1)
Top	O(1)

Tabela 2.3: *Consumo de tempo das operações na pilha tradicional.*

2.4.3 Fila de prioridade

- INSERT(value): recebe um elemento e o insere na fila.
- DELETE MIN(): remove e devolve o elemento cuja chave é mínima.
- MINIMUM(): devolve o elemento cuja chave é mínima.

Operação	Consumo de tempo
INSERT	$O(\lg(n))$
DELETE MIN	$O(\lg(n))$
MINIMUM	$O(1)$

Tabela 2.4: *Consumo de tempo das operações na fila de prioridade tradicional, baseada num binary heap.*

Capítulo 3

Fila Parcialmente Retroativa

A primeira estrutura abordada é a fila, tanto em sua versão parcial como em sua versão totalmente retroativa. As operações de uma fila tradicional são *enqueue, dequeue, first* e *last*, sendo as duas primeiras de modificação e as duas últimas de consulta.

Como apresentado no capítulo anterior, as operações de modificação precisam ser associadas ao instante de tempo em que ocorreram. Conforme será mostrado mais adiante, não necessariamente todas as operações de modificação são armazenadas, apenas aquelas que são indispensáveis para responder às consultas a que cada tipo de retroatividade se propõe.

3.1 Interface e comentários da implementação

A estrutura foi implementada como uma classe que suporta os seguintes métodos:

- Init_Partial_Queue(): instancia uma fila parcialmente retroativa vazia.
- Add_Enqueue(time, value): insere a operação *enqueue(value)* na lista de operações no instante *time*.
- Add_Dequeue(time): insere a operação *dequeue()* na lista de operações no instante *time*.
- Remove_Enqueue(time): remove a operação *enqueue* que ocorreu no instante *time* da lista de operações.
- Remove_Enqueue(time): remove a operação *dequeue* que ocorreu no instante *time* da lista de operações.
- Query_First(): devolve o primeiro elemento da fila no presente.
- Query_Last(): devolve o último elemento da fila no presente.

A versão da fila parcialmente retroativa apresentada por Demaine *et al.*(4) é implementada através de duas estruturas de dados interligadas: uma lista duplamente ligada auxiliada

por uma árvore balanceada, detalhadas na próxima seção. A fila também possui o ponteiro *front*, que aponta para a célula da lista que contém seu primeiro elemento. Como o último elemento da fila coincide com o que está contido na última célula da lista, ele é acessado através do ponteiro da própria lista ligada.

Quando uma consulta é realizada, o valor retornado é o valor da célula apontada pelo ponteiro *front*, se a consulta é QUERY_FIRST(), ou da célula apontada pelo ponteiro *last* da lista ligada, se a consulta é QUERY_LAST().

3.2 Estrutura interna: lista duplamente ligada com árvore auxiliar

A fila parcial usa uma lista ligada para manter as operações ordenadas pelos seus instantes de tempo. As células dessa lista possuem os campos *time*, *value*, *past* e *next*, que guardam, respectivamente, o instante de tempo da operação, o valor associado a operação, o apontador para a célula anterior e o apontador para a célula seguinte.

Como a inserção e a remoção das células precisa conservar de modo eficiente a ordenação do tempo, a lista faz uso de uma estrutura auxiliar: uma árvore balanceada, como a apresentada na seção 2.3, cuja chave é o tempo e o valor é uma referência à célula da lista correspondente àquele tempo. Vale ressaltar que o tempo de chaveamento está contido na célula da lista, acessível pela referência. Essa dinâmica entre as duas estruturas é ilustrada pela Figura 3.1.

- INIT_LINKED_LIST(): inicializa uma lista vazia, que possui como atributos três apontadores: o *head*, que aponta para a primeira célula da lista, o *last*, que aponta para a última e a *tree*, que é a árvore de células da lista, inicialmente vazia.
- INSERT(time, value): insere uma nova célula com o par (*time*, *value*) na lista na posição apropriada.
- DELETE(time): remove a célula da lista que possui instante *time*.
- FORWARD(pointer): recebe um apontador *pointer* para uma célula da lista e o desloca para a célula seguinte da lista.
- BACKWARD(pointer): recebe um apontador *pointer* para uma célula da lista e o desloca para a célula anterior da lista.

Código 1 Lista Ligada: inicialização

```
1: function INIT_LINKED_LIST()
2:     tree ← INIT_AVL_TREE()
3:     head ← nil
4:     last ← nil
```

O uso da árvore auxiliar é detalhado nos códigos de inserção (Código 2) e remoção (Código 3) apresentados a seguir. Na inserção utiliza-se a rotina SEARCH(*time*), que devolve a célula cuja referência esteja no nó pai da nova célula a ser inserida. Isso porque é garantido que elas serão vizinhas na lista, por propriedade da árvore binária de busca.

Utiliza-se também o método INSERT(cell) da árvore, que insere um novo nó cujo valor é a referência para a célula *cell* passada como argumento.

Código 2 Lista Ligada: inserção

```
 1: function INSERT(time, value)
 2:     tree_parent ← tree.SEARCH(time)
 3:     if tree_parent = nil then                          ▷ list is empty
 4:         new_cell ← INIT_CELL(time, value, nil, nil)
 5:         head ← new_cell
 6:         last ← head
 7:     else if time < tree_parent.time then
 8:         new_cell ← INIT_CELL(time, value, tree_parent.past, tree_parent)
 9:         if tree_parent.past then
10:             tree_parent.past.next ← new_cell
11:             tree_parent.past ← new_cell
12:         if tree_parent = head then
13:             head ← new_cell
14:     else
15:         new_cell ← INIT_CELL(time, value, tree_parent, tree_parent.next)
16:         if tree_parent.next then
17:             tree_parent.next.past ← new_cell
18:             tree_parent.next ← new_cell
19:         if tree_parent = last then
20:             last ← new_cell
21:     tree.INSERT(new_cell)
```

Na remoção, o método DELETE(time) da árvore remove o nó cujo instante é *time* e devolve a célula referenciada por ele.

Código 3 Lista Ligada: remoção

```
1:  function DELETE(time)
2:      cell ← tree.DELETE(time)
3:      if cell = head then
4:          head ← cell.next
5:      else
6:          cell.past.next ← cell.next
7:      if cell = last then
8:          last ← cell.past
9:      else
10:         cell.next.past ← cell.past
```

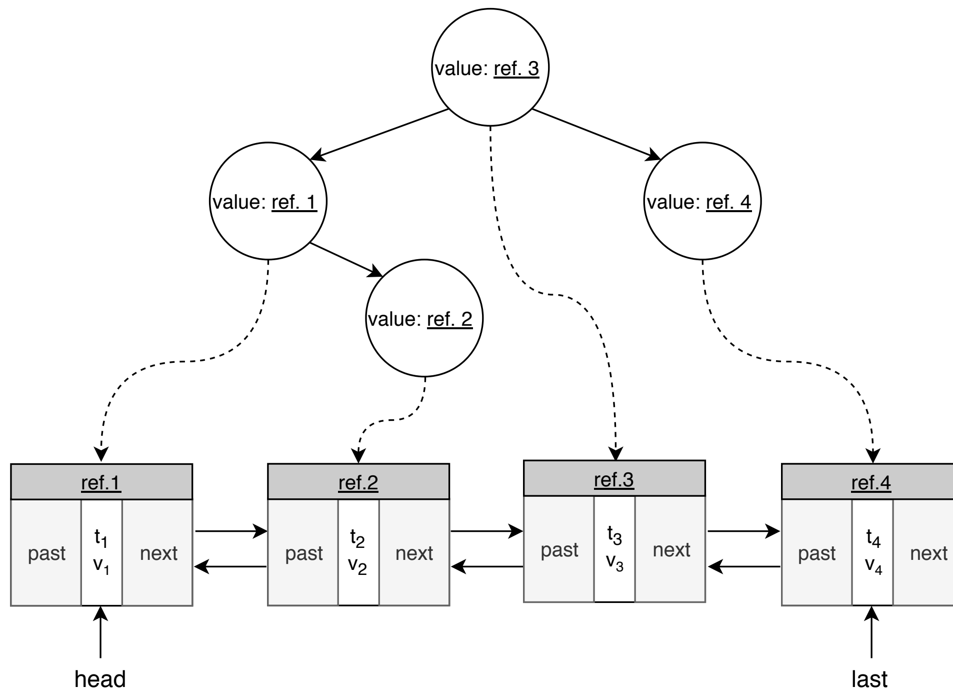

Figura 3.1: *Exemplificação de como a lista ligada se relaciona com a árvore de busca binária.*

3.3 Implementação da fila parcialmente retroativa

Código 4 Classe PartialQueue

```
1: function Init_Partial_Queue()
2:     enqueue_list ← Init_Linked_List()
3:     front ← nil
4: function Add_Enqueue(time, value)
5:     enqueue_list.Insert(time, value)
6:     if front = nil then
7:         front ← enqueue_list.head
8:     else if time < front.time then
9:         enqueue_list.Backward(front)
10: function Add_Dequeue(time)
11:     enqueue_list.Backward(front)
12: function Remove_Enqueue(time)
13:     if time ⩽ front.time then
14:         enqueue_list.Forward(front)
15:     enqueue_list.Delete(time)
16: function Remove_Dequeue(time)
17:     enqueue_list.Forward(front)
18: function Query_First()
19:     return front.value
20: function Query_Last()
21:     return enqueue_list.last.value
```

No caso da fila parcialmente retroativa, a operação *enqueue* é a única que será armazenada na lista. Como as consultas são feitas sempre no presente, o controle de quem está no começo ou fim da fila é feito apenas pelos apontadores. Quando um *dequeue* é inserido ou removido, a única alteração que a fila pode sofrer, a depender do instante da operação, é o deslocamento de seu apontador *front*.

Dessa maneira, apenas quando um método envolve a operação *enqueue* é que as estruturas da lista ligada, e da árvore associada a ela, são alteradas, sendo inseridos ou removidos os nós e as células correspondentes. Já a atualização dos ponteiros da lista ocorre tanto nas modificações ocasionadas por *enqueue* como por *dequeue*.

Quando um *enqueue* é inserido antes da célula apontada pelo *front*, este é movido para seu antecessor. Quando um *dequeue* é inserido, independentemente do instante, o ponteiro *front* é movido para seu sucessor.

O tratamento de uma remoção também é dividido em dois casos. No caso da remoção de um *enqueue*, a célula da lista e o nó correspondente da árvore são removidos, e na atualização

dos ponteiros, se o instante removido é menor ou igual ao do *front*, então este é movido para seu sucessor. Já no caso da remoção de um *dequeue*, ele é movido para seu antecessor.

3.4 Exemplo de execução

Para ilustrar como a lista ligada se comporta quando operações são executadas na fila parcial (Tabela 3.1), segue uma simples sequência de operações:

ADD_ENQUEUE(5, M)	time: 5, value: M
ADD_ENQUEUE(18, F)	time: 18, value: F
ADD_ENQUEUE(2, D)	time: 2, value: D
ADD_ENQUEUE(11, K)	time: 11, value: K
ADD_DEQUEUE(3)	time: 3, value: _

Tabela 3.1: *Sequência de operações da fila parcialmente retroativa.*

O diagrama a seguir (Figura 3.2) ilustra os estados da lista ligada após a execução das operações apresentadas na Tabela 3.1, com exceção das duas primeiras, que já são contabilizadas no início da figura. A célula destacada em cada versão da lista é aquela que está no começo da fila, apontada pelo ponteiro *front*, e nas laterais estão indicadas as operações que são adicionadas ou removidas.

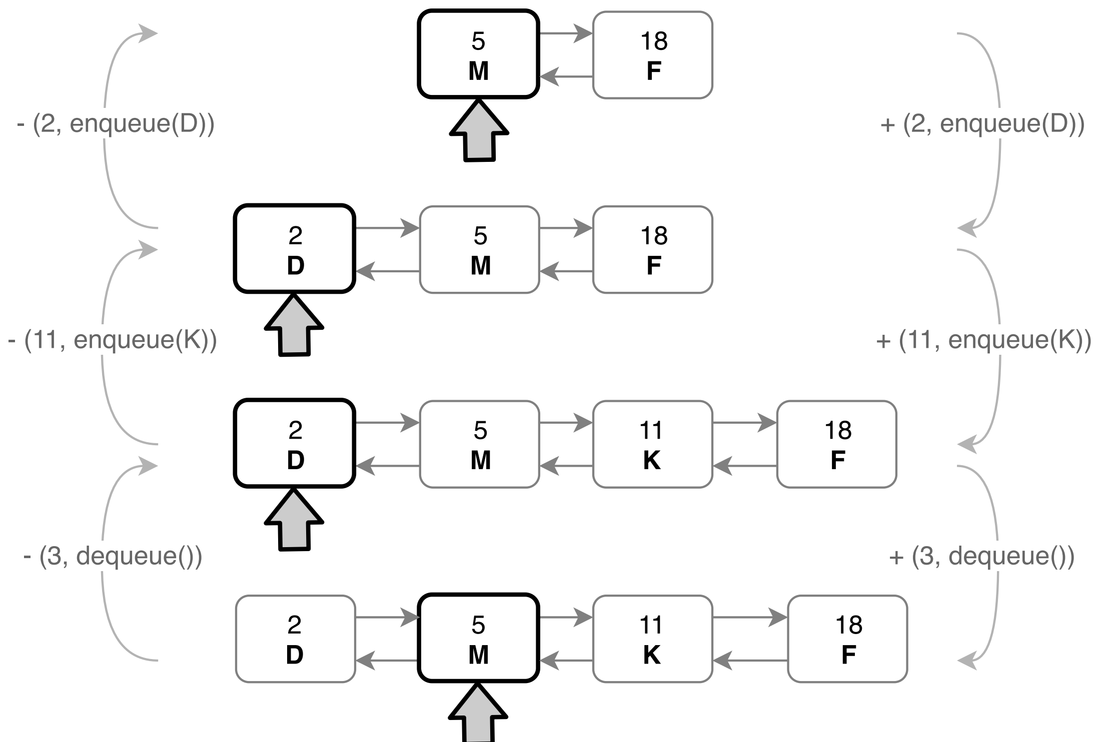

Figura 3.2: *Estado da fila parcial depois das últimas três operações da sequência 3.1*

Seguindo a mesma lógica do diagrama anterior (Figura 3.2), a Figura 3.3 mostra mais detalhadamente como ambas as estruturas da fila se modificam. Em cada região da figura tem-se o estado da árvore e da lista conforme elas recebem as operações da Tabela 3.1.

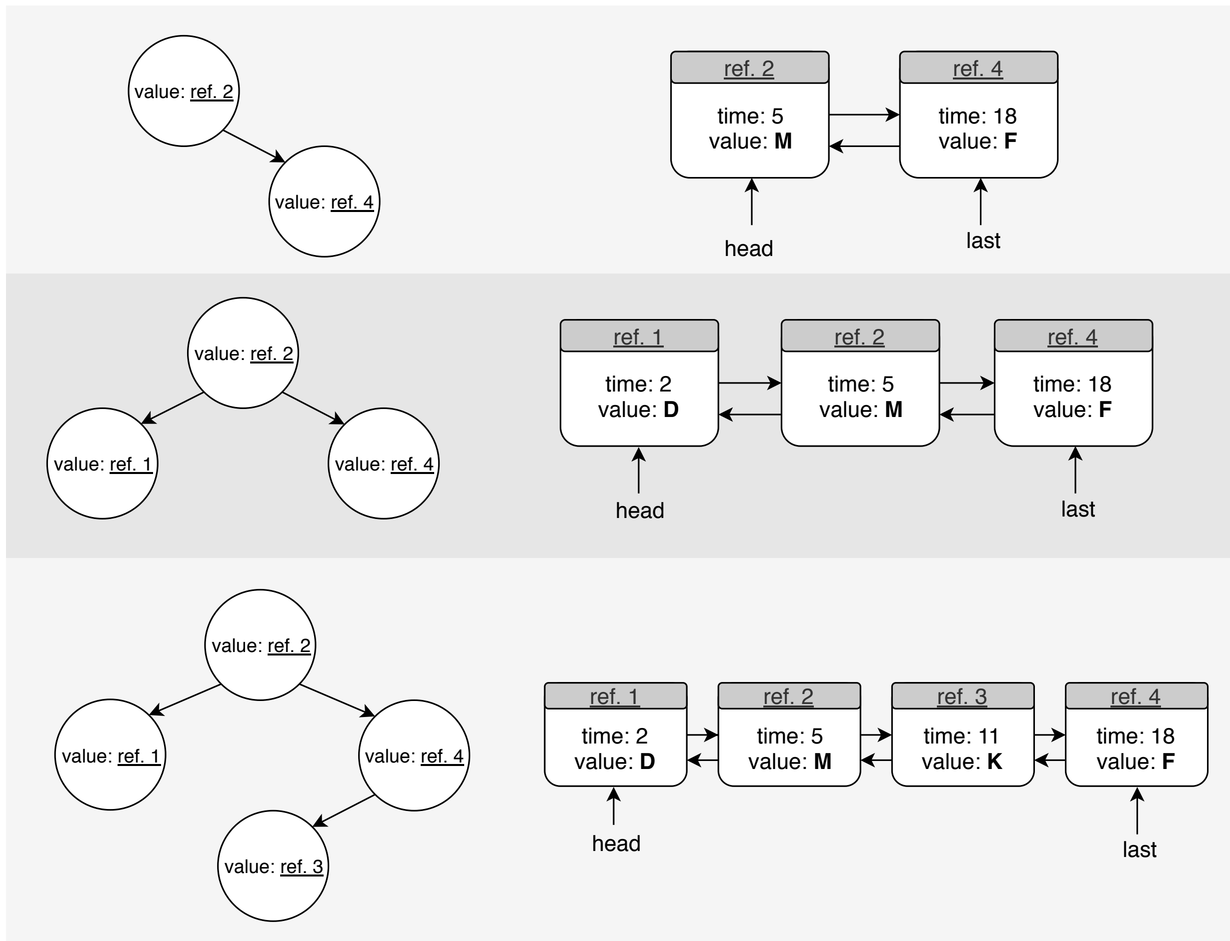

Figura 3.3: *Estado das estruturas que compõem a fila parcial depois das últimas três operações da sequência 3.1. Interessante notar que a cabeça da lista nem sempre coincide com o primeiro da fila.*

3.5 Análise do consumo de tempo das operações

A inserção ou remoção de um *enqueue* está atrelada a uma busca na árvore, logo essas operações gastam tempo proporcional a $\lg(n)$, sendo n o número de nós. A inserção ou remoção de um *dequeue* consiste em mover o ponteiro *front* para sua célula imediatamente anterior ou posterior, gastando tempo constante. Por fim, cada consulta é um simples acesso ao conteúdo de uma célula através de ponteiros, também gastando tempo constante. Segue um resumo do consumo de tempo das operações (Tabela 3.2).

Operação	Pior Caso
ADD_ENQUEUE	O(lg(n))
ADD_DEQUEUE	O(1)
REMOVE_ENQUEUE	O(lg(n))
REMOVE_DEQUEUE	O(1)
QUERY_FIRST	O(1)
QUERY_LAST	O(1)

Tabela 3.2: *Consumo de tempo das operações na fila parcial, onde n é o número de nós da árvore.*

Capítulo 4

Fila Totalmente Retroativa

Na fila totalmente retroativa, além de inserir ou remover as operações *enqueue* e *dequeue* em qualquer instante de tempo, é necessário responder às consultas *first* e *last* em qualquer instante de tempo. Na versão parcial estas estavam restritas ao tempo presente.

Para tornar isso possível, nessa estrutura será necessário calcular rapidamente quantas operações *dequeue* ocorreram até um dado instante. Para tanto, serão mantidas as operações *dequeue* ordenadas pelo instante em que ocorreram.

Isso será fundamental para saber quem é o primeiro elemento da fila em qualquer momento, pois num dado instante t, sendo d_t o número de *dequeues* que ocorreram até esse instante, o primeiro elemento da fila será aquele associado ao (d_t+1)-ésimo *enqueue* ocorrido, isto é, o elemento correspondente ao *enqueue* ocorrido logo após os d_t primeiros *enqueues*, cujos elementos foram removidos pelos *dequeues*.

4.1 Exemplo abstrato

Para mostrar a dinâmica dessa estrutura, seguem alguns exemplos. As operações *enqueue()* e *dequeue()* são representadas por $\mathbf{E}()$ e $\mathbf{D}()$, ao longo de uma linha do tempo, e os elementos inseridos são letras passadas como argumento. Ao lado, o estado da fila é representado em cada instante de tempo, onde o começo da fila fica no extremo esquerdo. Embaixo estão duas consultas *first()* em instantes diferentes, representada por $\mathbf{F}()$. É fácil notar como a mudança nas operações afeta o resultado das consultas.

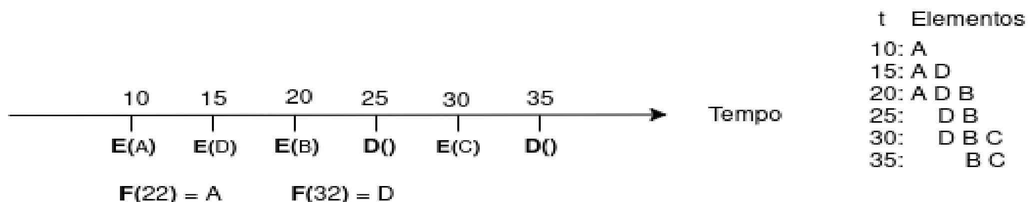

Figura 4.1: *Linha do tempo de operações numa fila totalmente retroativa.*

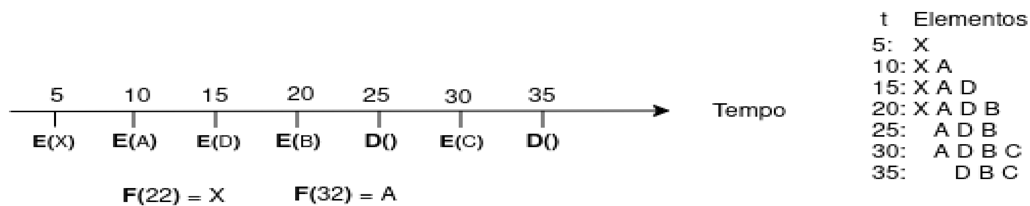

Figura 4.2: *Linha do tempo acrescida da operação enqueue no instante 5.*

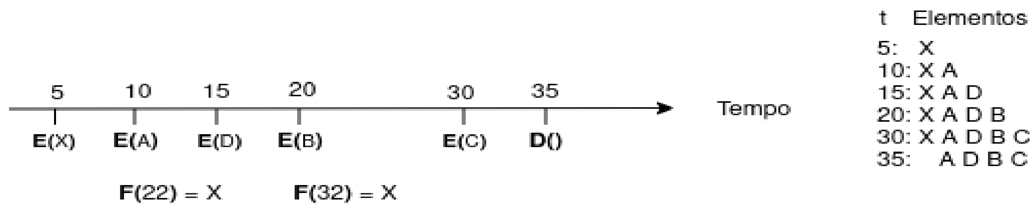

Figura 4.3: *Linha do tempo decrescida da operação dequeue no instante 25.*

4.2 Interface e comentários de implementação

Para a implementação desta estrutura foi construída uma classe similar à apresentada para a fila parcial. As operações de modificação continuam as mesmas, mas as operações de consulta agora podem ser feitas em qualquer instante de tempo:

- Query_Kth(time, k): devolve o elemento que ocupa a posição k no instante *time*.
- Query_First(time): devolve o elemento que está no começo da fila no instante *time*.

4.3 Estrutura interna: árvore de operações

A árvore utilizada nessa implementação possui dois tipos de nó: o nó interno e a folha. A informação principal ficará armazenada nas folhas, enquanto que os nós internos servirão para auxiliar as buscas. Ambos possuem o atributo *is_leaf*, que assume valor verdadeiro para uma folha e falso para um nó interno.

Os nós internos carregam como sua chave o *min_right_time*, que é o menor instante armazenado numa folha de sua subárvore direita. Além disso, os nós carregam também o atributo *leaves*, que é o número de folhas da árvore enraizada naquele nó, e os ponteiros *left* e *right*, que apontam para as subárvores esquerda e direita respectivamente.

Como dito anteriormente, a folha ficará encarregada de armazenar propriamente a informação. Os seus atributos são *time*, que guarda o instante em que a operação ocorreu e *value*, que guarda o valor associado à operação.

- Init_Avl_Tree(): inicializa uma árvore vazia. Ela possui como atributo apenas um ponteiro *root* para sua raiz, que no início é nulo.

- INSERT(time, value): cria e insere uma nova folha com os parâmetros *time* e *value* recebidos.
- DELETE(time): busca e remove a folha cuja a chave é o instante *time*.
- COUNT(time): devolve o número de folhas com chave$\leqslant$ *time*.
- KTH(time, k): devolve o valor da k-ésima folha da árvore, se ela está associada a um instante $\leqslant$ *time*. Caso contrário devolve nulo.

Dentre essas rotinas, as duas últimas são as que merecem ser mostradas (Código 5 e Código 6) em maiores detalhes, por serem menos convencionais. As demais são rotinas tradicionais da árvore binária de busca.

Código 5 Árvore: número de folhas com chave $\leqslant$ time

```
1: function COUNT(time)
2:     if root ≠ nil then
3:         return _COUNT(root, time, 0)
4:     return 0
5: function _COUNT(node, time, counter)
6:     if node.is_leaf then
7:         if node.time ⩽ time then
8:             counter ← counter + 1
9:         return counter
10:    else if time < node.min_right_time then
11:        return _COUNT(node.left, time, counter)
12:    else:
13:        if node.left.is_leaf then
14:            counter ← counter + 1
15:        else
16:            counter ← counter + node.left.leaves
17:        return _COUNT(node.right, time, counter)
```

Código 6 Árvore: k-ésima folha até o instante time

```
1: function Kth(time, k)
2:     node ← _Kth(root, time, k)
3:     if node.time > time then
4:         return nil
5:     return node.value
6: function _Kth(node, k)
7:     if node.is_leaf then
8:         return node
9:     else if node.left.is_leaf then
10:        if k = 1 then
11:            return _Kth(node.left, k)
12:        return _Kth(node.right, k - 1)
13:    else
14:        if k > node.left.leaves then
15:            return _Kth(node.right, k - node.left.leaves)
16:        return _Kth(node.left, k)
```

4.4 Implementação da fila totalmente retroativa

No lugar da lista duplamente ligada usada na versão parcial, a fila totalmente retroativa faz uso de duas árvores de busca binária: uma para armazenar suas operações *enqueue* e outra para as operações *dequeue*. Quando uma operação é inserida ou removida, ela é simplesmente inserida ou removida de sua árvore correspondente.

Para responder às consultas Query_First(time) e Query_Kth(time, k) são contados quantos *dequeues* ocorreram até o instante *time* (d_{time}). O elemento que ocupa a posição $d_{time} + 1$ na árvore de *enqueues* é retornado como o primeiro elemento no instante *time*, e o que ocupa a posição $d_{time} + k$ é devolvido como o k-ésimo elemento no instante *time*.

Código 7 Classe TotalQueue

```
1: function Init_Total_Queue()
2:     enqueue_tree ← Init_Avl_Tree()
3:     dequeue_tree ← Init_Avl_Tree()
4: function Add_Enqueue(time, value)
5:     enqueue_tree.Insert(time, value)
6: function Add_Dequeue(time)
7:     dequeue_tree.Insert(time, nil)
8: function Remove_Enqueue(time)
9:     enqueue_tree.Delete(time)
10: function Remove_Dequeue(time)
11:     dequeue_tree.Delete(time)
12: function Query_Kth(time, k)
13:     d ← dequeue_tree.count(time)
14:     return enqueue_tree.Kth(time, d + k)
15: function Query_First(time)
16:     return Query_Kth(time, 1)
```

4.5 Exemplo de execução

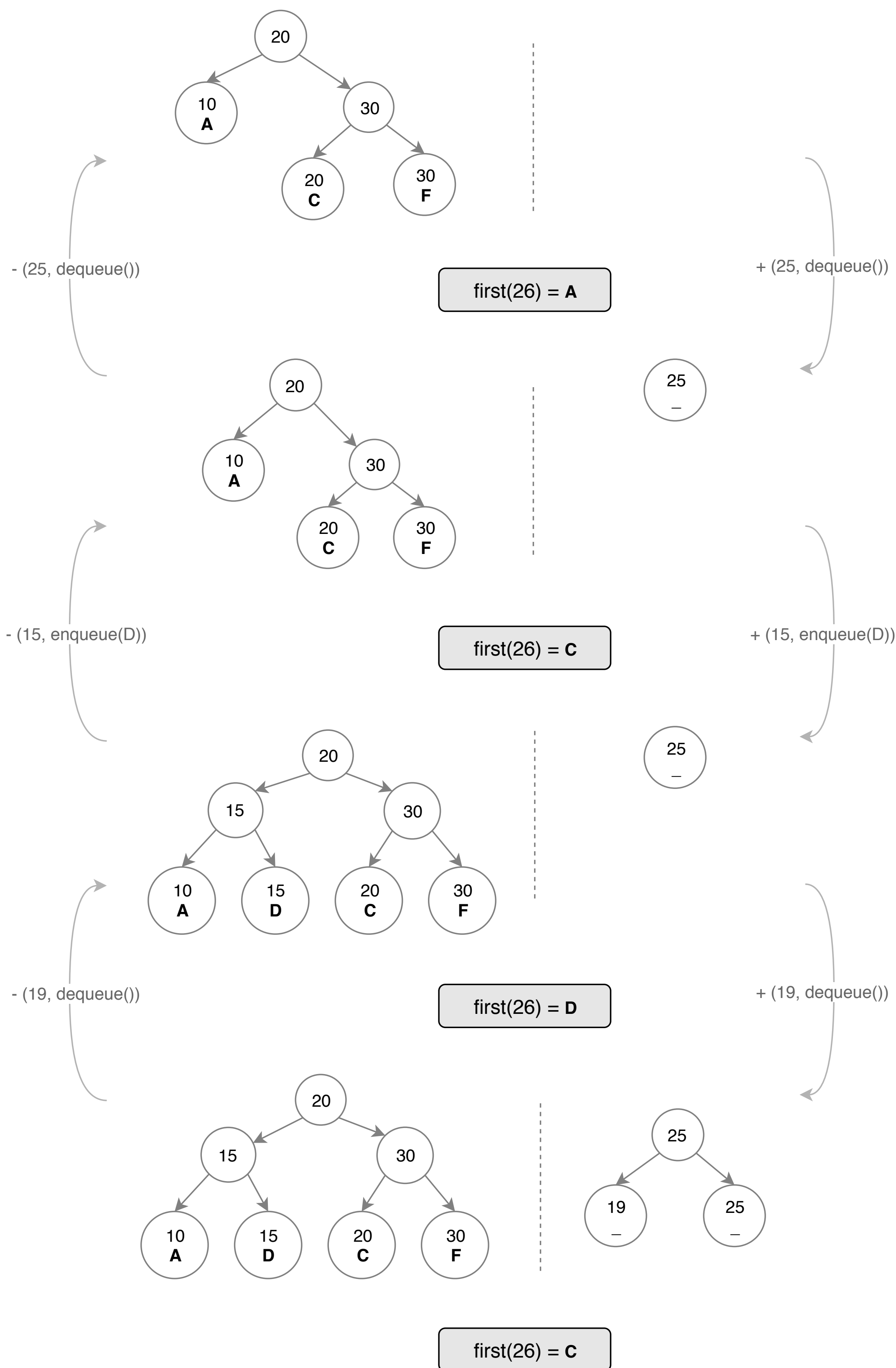

Figura 4.4: *Diagrama das árvores de operações recebendo inserções, remoções e consultas. A árvore da esquerda trata os enqueue(), e a da direita dos dequeue()*

O diagrama anterior (Figura 4.4) ilustra os estados da árvore de *enqueues* (à esquerda) e da árvore de *dequeues* (à direita) após a execução das operações apresentadas na Tabela 4.1, sendo as duas primeiras operações já contabilizadas no início da figura. Destacada embaixo de cada estado há uma consulta *first*, que para efeitos de comparação é realizada sempre no mesmo instante. Nas laterais da figura estão indicadas as operações que são adicionadas ou removidas.

ADD_ENQUEUE(10, A)	time: 10, value: A
ADD_ENQUEUE(20, C)	time: 20, value: C
ADD_ENQUEUE(30, F)	time: 30, value: F
ADD_DEQUEUE(25)	time: 25, value: _
ADD_ENQUEUE(15, D)	time: 15, value: D
ADD_DEQUEUE(19)	time: 19, value: _

Tabela 4.1: *Sequência de operações da fila totalmente retroativa, sendo as últimas três simuladas no diagrama 4.4.*

4.6 Análise do consumo de tempo das operações

Operação	Pior Caso
ADD_ENQUEUE	O(lg(n))
ADD_DEQUEUE	O(lg(n))
REMOVE_ENQUEUE	O(lg(n))
REMOVE_DEQUEUE	O(lg(n))
QUERY_FIRST	O(lg(n))
QUERY_KTH	O(lg(n))

Tabela 4.2: *Consumo de tempo das operações na fila total, onde n é o número de operações executadas.*

Capítulo 5

Pilha Totalmente Retroativa

A terceira estrutura apresentada será a pilha, apenas em sua versão totalmente retroativa. Não há na literatura uma descrição dessa estrutura; a apresentação aqui exibida é original. A inspiração para ela surgiu da maneira como a fila de prioridade parcialmente retroativa foi construída. A fila de prioridade será deixada por último por ser mais complexa, e a pilha será um bom aquecimento.

As operações de uma pilha tradicional são *push*, *pop* e *top*, as duas primeiras causam modificações na estrutura e a última realiza uma consulta sobre o estado da estrutura. Uma outra operação de consulta, não tão tradicional, é a *kth*, que devolve o valor na k-ésima posição da pilha, a partir do topo.

5.1 Apresentação

- Init_Total_Stack(): inicializa uma pilha vazia.
- Add_Push(time, value): insere a operação *push(value)* no instante *time*.
- Add_Pop(time): insere a operação *pop()* no instante *time*.
- Remove_Push(time): remove a operação *push* que ocorreu no instante *time*.
- Remove_Pop(time): remove a operação *pop* que ocorreu no instante *time*.
- Query_Top(time): devolve o elemento do topo da pilha no instante *time*.
- Query_Kth(time, k): devolve o elemento que ocupa a k-ésima posição no instante *time*.

Diferentemente da abordagem da fila totalmente retroativa, as operações *push* e *pop* não são mantidas em estruturas diferentes. Dessa vez, a estratégia para manter a eficiência envolve guardar os dois tipos de operação numa mesma árvore de busca binária, e atribuir pesos a eles. A operação *push* receberá peso $+1$ e a operação *pop* receberá peso -1.

5.2 Estrutura interna: árvore de operações

A árvore utilizada nessa implementação possui nós internos e folhas, nos quais ambos os tipos de nó possuem o atributo booleano *is_leaf* para diferenciá-los, e o nó interno continua usando como chave o *min_right_time*, que representa o menor instante de uma folha da subárvore direita.

No entanto, a pilha segue uma estratégia diferente da fila, e essa estratégia demanda alguns atributos novos nos nós da árvore. Agora as folhas têm pesos associados a elas, que podem ser positivos ou negativos.

Ao olhar para as folhas em sequência, aquelas de peso positivo são anuladas pelas de peso negativo que se seguirem. Dessa maneira, torna-se necessário contabilizar a soma dos pesos das folhas e o saldo de folhas não anuladas em cada subárvore.

Para tanto, os dois tipos de nós serão acrescidos dos atributos *weight* e *leftover*. Quando se tratar de um nó interno, esses dois atributos representarão a soma de todos os pesos nas folhas de sua subárvore, e o saldo de folhas positivas correspondente. Já quando se tratar de uma folha, esses atributos corresponderão apenas à operação armazenada ali.

- INIT_AVL_TREE(): inicializa uma árvore vazia que possui um ponteiro *root* para sua raiz, que no início é nulo.

- INSERT(time, value, weight): cria e insere uma nova folha com os parâmetros *time*, *value* e *weight* recebidos.

- DELETE(time, weight): remove a folha que possui instante *time* e peso *weight*.

- KTH(time, k): devolve o *value* da *k*-ésima folha que possui peso positivo, e não foi anulada, até o instante *time*.

Dessa sequência de rotinas, essa nova versão da função KTH (Código 8) merece maior atenção. Nela, busca-se na árvore a folha de instante *time*. Ela é retornada caso seja positiva e corresponda à posição buscada. Caso contrário, passa a ser necessário ir para folhas anteriores. Neste caso, se a folha for negativa, precisa ser levado em conta esse peso negativo, para que ele seja anulado.

Código 8 Árvore: operação positiva não anulada que ocupa uma dada posição

```
1: function _Kth(node, time, k)
2:     if node.is_leaf then
3:         if node.weight = -1 or k ≠ 1 then
4:             return [nil, node.weight]
5:         else
6:             return [node.value]
7:     else if time < node.min_right_time then
8:         return _Kth(node.left, time, k)
9:     else
10:        kth_right ← _Kth(node.right, time, k)
11:        if kth_right[0] = nil then
12:            k ← k - kth_right[1]
13:            if node.left.leftover ⩾ k then
14:                return [_Get_Value(node.left, k)]
15:            else
16:                return [nil, kth_right[1] + node.left.weight]
17:        else
18:            return kth_right
19: function _Get_Value(node, k)
20:     if node.is_leaf then
21:         return node.value
22:     else if node.right.leftover ⩾ k then
23:         return _Get_Value(node.right, k)
24:     else
25:         return _Get_Value(node.left, k - node.right.weight)
```

5.3 Implementação da pilha totalmente retroativa

Código 9 Classe TotalStack

```
1: function INIT_TOTAL_STACK()
2:     update_tree ← INIT_AVL_TREE()
3: function ADD_PUSH(time, value)
4:     update_tree.INSERT(time, value, 1)
5: function ADD_POP(time)
6:     update_tree.INSERT(time, value, -1)
7: function REMOVE_PUSH(time)
8:     update_tree.DELETE(time, 1)
9: function REMOVE_POP(time)
10:    update_tree.DELETE(time, -1)
11: function QUERY_TOP(time)
12:    update_tree.KTH(time, 1)
13: function QUERY_KTH(time, k)
14:    update_tree.KTH(time, k)
```

Num determinado instante t, para encontrar quem ocupa o topo da pilha, busca-se na árvore o primeiro sufixo positivo e não nulo que termina em t. Em outras palavras, o elemento que está sendo buscado é aquele associado ao primeiro *push* que, retrocedendo nas folhas da árvore de operações a partir do instante t, torna em valor $+1$ a soma dos pesos dessas operações.

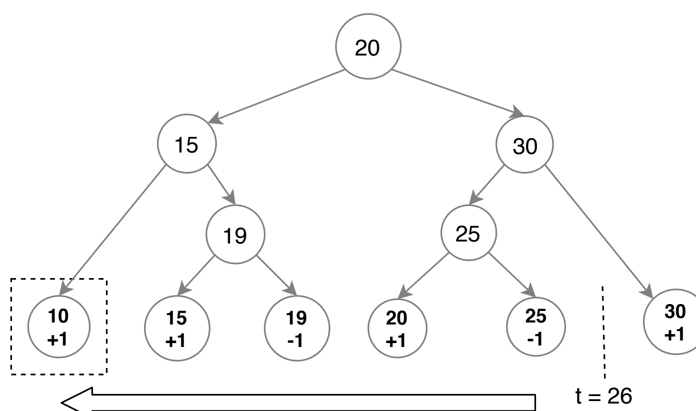

Figura 5.1: *Abstração da busca do topo da pilha no instante* 26*. A última operação encontrada antes desse instante é um pop(), então o sufixo vale* -1*. Para encontrar o topo, o sufixo de folhas até o instante* 26 *deve valer* $+1$*, como passa a valer no instante* 10*.*

No caso da busca pelo k-ésimo elemento da pilha num instante t, o procedimento é

similar. No entanto, o elemento buscado é aquele associado ao *push* que torna o valor do sufixo das operações até t em $+k$, ao invés de $+1$ como na busca pelo topo.

5.4 Exemplo de execução

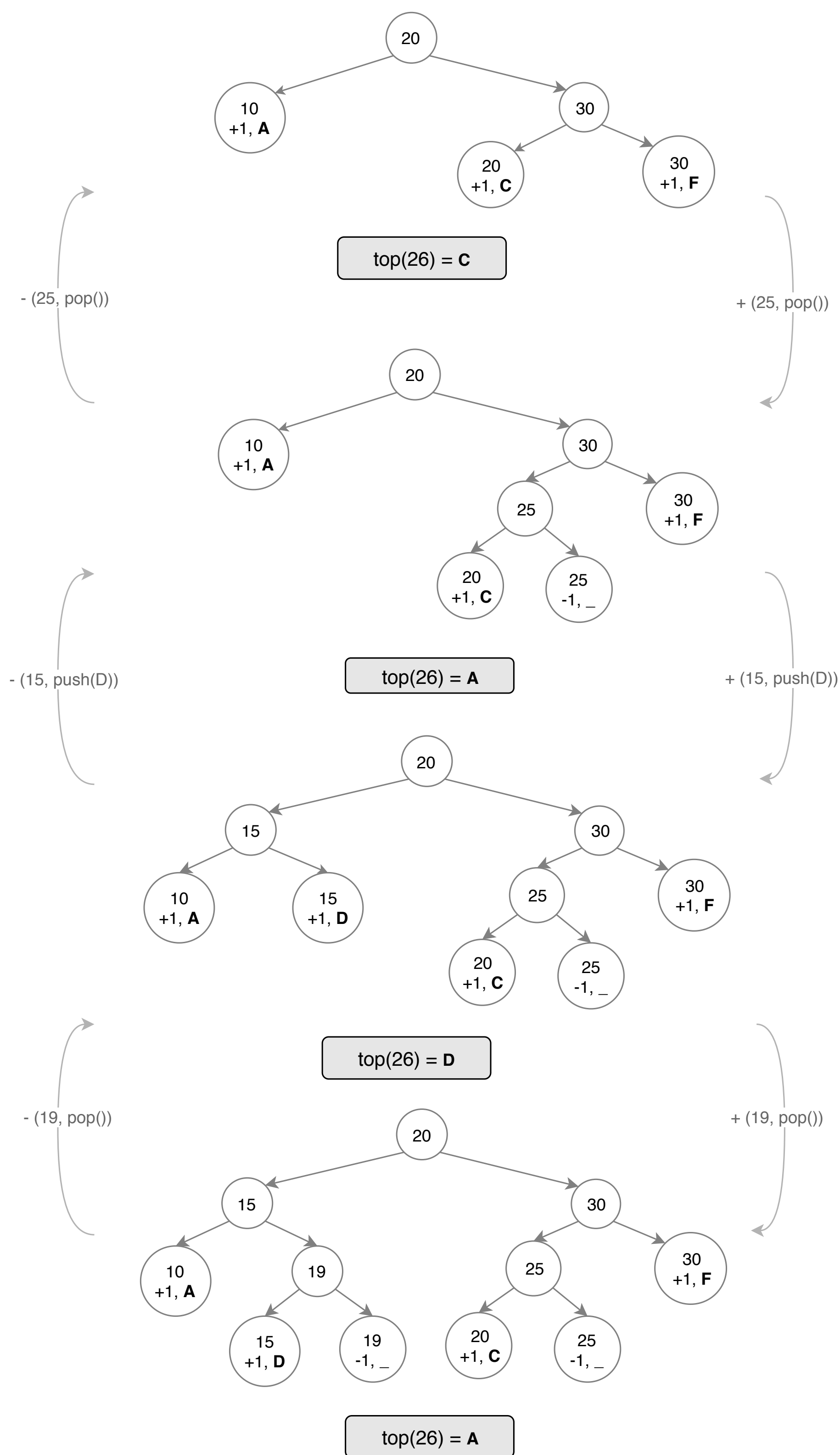

Figura 5.2: *Diagrama da árvore de operações recebendo inserções, remoções e consultas.*

O diagrama anterior (Figura 5.2) ilustra os estados da árvore de *pushes* e *pops* após a execução das operações apresentadas na Tabela 5.1, sendo as duas primeiras operações já contabilizadas no início da figura. Destacada embaixo de cada estado há uma consulta *top*, que para efeitos de comparação é realizada sempre no mesmo instante. Nas laterais da figura estão indicadas as operações que são adicionadas ou removidas.

ADD_PUSH(10, A)	time: 10, value: A
ADD_PUSH(20, C)	time: 20, value: C
ADD_PUSH(30, F)	time: 30, value: F
ADD_POP(25)	time: 25, value: _
ADD_PUSH(15, D)	time: 15, value: D
ADD_POP(19)	time: 19, value: _

Tabela 5.1: *Sequência de operações da pilha totalmente retroativa, sendo as últimas três simuladas no diagrama 5.2.*

5.5 Análise de consumo de tempo

Operação	Pior Caso
ADD_PUSH	O(lg(n))
ADD_POP	O(lg(n))
REMOVE_PUSH	O(lg(n))
REMOVE_POP	O(lg(n))
QUERY_TOP	O(lg(n))
QUERY_KTH	O(lg(n))

Tabela 5.2: *Consumo de tempo das operações na pilha total, onde n é o número de operações executadas.*

Capítulo 6

Fila de Prioridade Parcialmente Retroativa

Neste capítulo é apresentada a estrutura mais sofisticada dentre as apresentadas até agora, e isso se deve por sua extrema sensibilidade à qualquer modificação. A simples inserção de uma operação pode desencadear várias alterações. Para que esse efeito cascata seja tratado eficientemente, é necessário representá-lo de maneira sucinta, a fim de evitar-se custos extras de manutenção da estrutura. Será apresentada apenas a versão parcial onde, lembrando pela última vez, dá-se suporte para consultas apenas no presente e alterações em qualquer instante.

Numa fila de prioridade tradicional, as operações suportadas são: *insert(key)*, que insere a chave *key* na fila; *delete()*, que remove o elemento de maior prioridade; *get()*, que devolve o elemento de maior prioridade. A prioridade pode ser definida seguindo qualquer critério, sendo os mais comuns os de máximo e de mínimo. A fila apresentada segue prioridade mínima, isto é, os valores mais baixos têm maior prioridade.

6.1 Definições

Ao longo do capítulo será necessário utilizar uma nomenclatura própria para referenciar alguns conceitos e resultados. Para uma chave k_i, sejam respectivamente t_i e d_i seus instantes de inserção e remoção, onde $i = 1, 2, \ldots$. Sejam Q_t e Q_{now} os conjuntos de chaves contidas na fila no instante t e no presente. Considere também que $I_{\geqslant t} = \{k_i \mid t_i \geqslant t\}$ é o conjunto de chaves inseridas após o instante t e que $D_{\geqslant t} = \{k_i \mid d_i \geqslant t\}$ é o conjunto de chaves removidas após o instante t. Por último, o termo **ponte** se refere a qualquer instante t em que $Q_t \subseteq Q_{now}$.

Para ilustrar melhor cada termo, segue um gráfico (Figura 6.1) representando uma sequência de operações numa fila de prioridade inicialmente vazia. Neste gráfico, o eixo das abscissas representa o tempo, e o eixo das ordenadas representa o valor das chaves. Cada segmento horizontal representa um item dentro da fila, onde seu extremo esquerdo

(t_i, k_i) representa o instante de inserção e seu extremo direito (d_i, k_i) representa o instante de remoção.

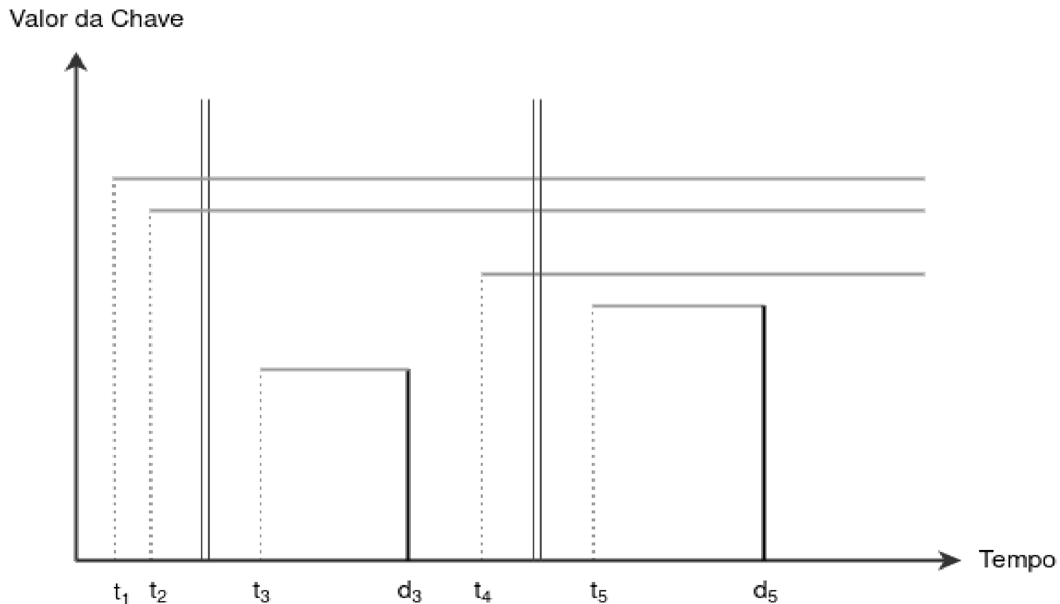

Figura 6.1: *Gráfico com uma sequência de operações. As linhas tracejadas são instantes de inserção, as linhas verticais em negrito são instantes de remoção de mínimo e as linhas horizontais são o período de existência da chave na fila. As linhas verticais duplas representam pontes.*

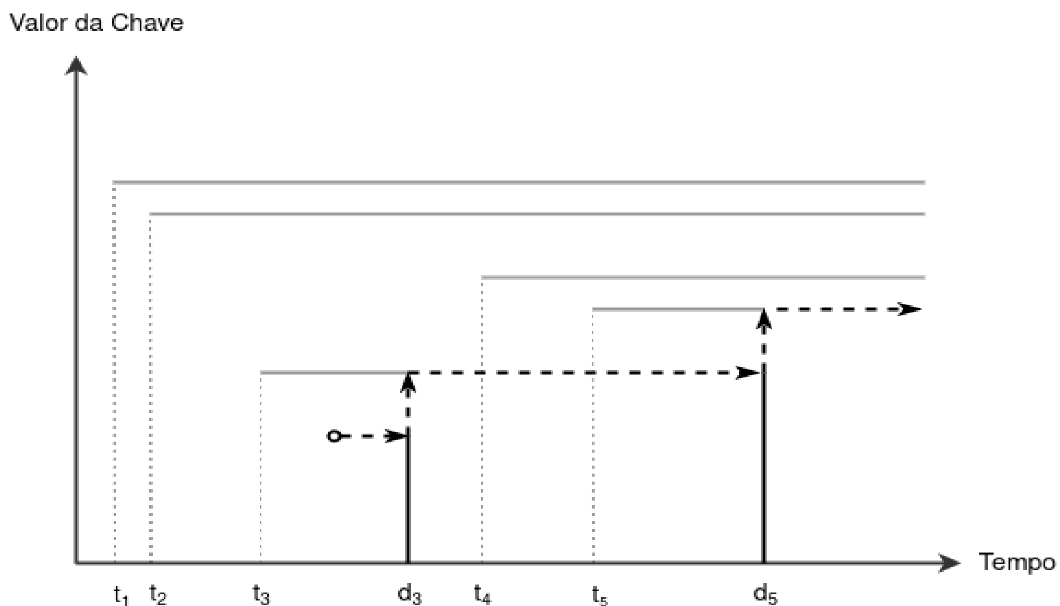

Figura 6.2: *Sequência de operações 6.1 com acréscimo de uma operação enqueue. O efeito da inclusão dessa operação está representado pelas linhas tracejadas em negrito.*

6.2 Interface e comentários de implementação

Novamente seguindo a versão da estrutura apresentada por Demaine *et al.* (4), essa fila de prioridade utiliza duas árvores de busca. A primeira, chamada *q_now*, representa o conjunto

de elementos que estão na fila no instante presente, e a segunda, chamada *updates*, guarda todas as operações inseridas.

As operações suportadas por ela são as seguintes:

- INIT_PRIORITY_QUEUE(): inicializa uma fila de prioridade parcial vazia.
- ADD_INSERT(time, value): adiciona a operação *insert(value)* no instante *time*.
- ADD_DELETE_MIN(time): insere a operação *delete_min()* no instante *time*.
- REMOVE_INSERT(time): remove a operação *insert()* do instante *time*.
- REMOVE_DELETE_MIN(time): remove a operação *delete_min()* do instante *time*.
- QUERY_MIN(): devolve a chave mínima do momento presente.

A tática de associar pesos às operações, usada pela pilha totalmente retroativa, foi inspirada por esta estrutura. Nela as operações também recebem pesos: as operações *insert()* cujos elementos estão disponíveis no presente, recebem peso 0, e aquelas cujos elementos já não estão mais disponíveis no presente, recebem peso 1; a operação *delete_min()* recebe peso -1.

6.3 Estruturas internas: árvore de operações e árvore de valores presentes

A versão abordada da fila de prioridades será a versão parcial, isto é, as consultas são suportadas apenas no momento presente. Isso traz a necessidade de separar os elementos que estão acessíveis no presente dos que não estão.

A fim de cumprir esse papel, além de uma árvore para guardar as operações, a fila de prioridades faz uso de uma segunda árvore binária de busca, com os elementos presentes no tempo corrente na fila de prioridade, usando o valor desses elementos como chave.

A árvore de operações é semelhante a da pilha: possui dois tipos de nó, o nó interno e a folha, e os atributos são os mesmos, com exceção de dois novos atributos que serão exclusividade da fila de prioridades.

Esses dois novos atributos são o *max_out* e *min_in*, que fazem parte da composição do nó interno. O *max_out* armazena o elemento de maior valor, numa folha da subárvore do nó, que não está presente na fila de prioridade no instante corrente. Já o *min_in* armazena o elemento de menor valor, numa folha da subárvore do nó, que está na fila de prioridade no tempo presente.

Apesar de ainda não estar muito nítida a razão desses atributos serem necessários, na seção de implementação fica claro que eles são fundamentais para garantir a eficiência nas operações.

- INIT_AVL_TREE(): inicializa uma árvore vazia que possui um ponteiro *root* para sua raiz, que no início é nulo.
- INSERT(time, value, weight): cria e insere uma nova folha com os parâmetros *time*, *value* e *weight* recebidos.
- DELETE(time, weight): remove a folha que possui instante *time* e peso *weight*.

6.4 Implementação da fila de prioridade parcialmente retroativa

Código 10 Classe PriorityQueue - Init/Query

```
1: function INIT_PRIORITY_QUEUE()
2:     q_now ← INIT_AVL_TREE()
3:     updates_tree ← INIT_AVL_TREE()
4: function QUERY_MIN()
5:     return q_now.MIN()
```

A implementação das operações da fila de prioridade parcial foi baseada em alguns resultados demonstrados por Demaine *el al.* (4). A seguir, esses resultados são enunciados, seguidos pelos trechos de código inspirados por eles.

Lemma 6.4.1. *Após uma operação Insert(t, insert(k)), o elemento a ser inserido em Q_{now} é*

$$max(k, \max_{k' \in D_{\geqslant t}} k').$$

Corollary 6.4.1.1. *Após uma operação Delete(t), onde a operação no instante t é delete-min, o elemento a ser inserido em Q_{now} é*

$$\max_{k' \in D_{\geqslant t}} k'.$$

Lemma 6.4.2. *Seja t' a última ponte antes de t. Então*

$$\max_{k' \in D_{\geqslant t}} k' = \max_{k' \in I_{\geqslant t'} - Q_{now}} k'.$$

Lemma 6.4.3. *Após uma operação Insert(t, delete-min()), o elemento a ser removido de Q_{now} é*

$$\min_{k \in Q_{t'}} k,$$

onde t' é a primeira ponte depois do instante t.

Corollary 6.4.3.1. *Após uma operação Delete(t), onde a operação no instante t é insert(k), o elemento a ser removido de Q_{now} é k, se $k \in Q_{now}$; senão o elemento removido é*

$$\min_{k \in Q_{t'}} k,$$

onde t' é a primeira ponte depois do instante t.

Código 11 Classe PriorityQueue - Add

```
1: function Add_Insert(time, value)
2:     t_bridge ← updates.Last_Bridge_Before(time)
3:     max_bridge ← updates.Max_After_Bridge(t_bridge)
4:     if (value > max_bridge.value) then
5:         updates.Insert(time, value, 0)
6:         value_Qnow ← value
7:     else
8:         updates.Set_Weight_Zero(max_bridge.time)
9:         updates.Insert(time, value, 1)
10:        value_Qnow ← max_bridge.value
11:    q_now.Insert(value_Qnow)
12: function Add_Delete_Min(time)
13:    t_bridge ← updates.First_Bridge_After(time)
14:    min_bridge ← updates.Min_Before_Bridge(t_bridge)
15:    updates.Set_Weight_One(min_bridge.time)
16:    updates.Insert(time, value, -1)
17:    q_now.Delete(min_bridge.value)
```

Código 12 Classe PriorityQueue - Remove

```
1:  function REMOVE_INSERT(time)
2:      del_node ← updates.SEARCH_NODE(time)
3:      if del_node.weight = 0 then
4:          value_Qnow ← del_node.value
5:      else if del_node.weight = 1 then
6:          time_bridge ← updates.FIRST_BRIDGE_AFTER(time)
7:          min_bridge ← updates.MIN_BEFORE_BRIDGE(time_bridge)
8:          updates.SET_WEIGHT_ONE(min_bridge.time)
9:          value_Qnow ← min_bridge.value
10:     updates.DELETE(time, del_node.weight)
11:     q_now.DELETE(value_Qnow)
12: function REMOVE_DELETE_MIN(time)
13:     time_bridge ← updates.LAST_BRIDGE_BEFORE(time)
14:     max_bridge ← updates.MAX_AFTER_BRIDGE(time_bridge)
15:     updates.SET_WEIGHT_ZERO(max_bridge.time)
16:     updates.DELETE(time, del_node.weight)
17:     q_now.INSERT(max_bridge.value)
```

6.5 Exemplo de execução

ADD_INSERT(10, A)	time: 10, value: A
ADD_INSERT(20, C)	time: 20, value: C
ADD_INSERT(30, F)	time: 30, value: F
ADD_DELETE_MIN(25)	time: 25, value: _
ADD_INSERT(15, D)	time: 15, value: D
ADD_DELETE_MIN(19)	time: 19, value: _

Tabela 6.1: *Sequência de operações da fila parcial de prioridade, sendo as últimas três simuladas no diagrama 6.3.*

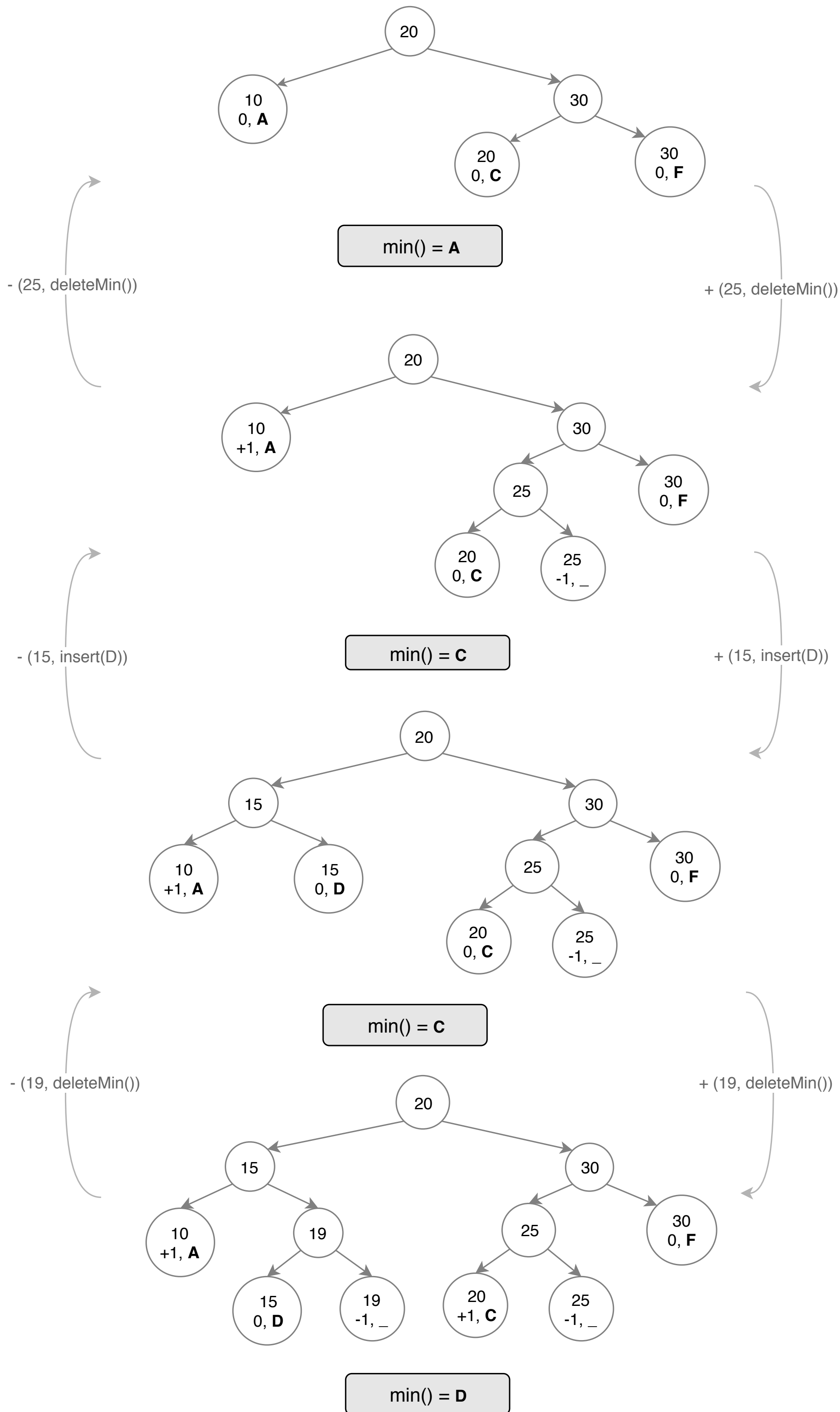

Figura 6.3: *Diagrama da árvore de operações recebendo inserções, remoções e consultas.*

6.6 Análise de consumo de tempo das operações

Operação	Pior Caso
ADD_INSERT	O(lg(n))
ADD_DELETE_MIN	O(lg(n))
REMOVE_INSERT	O(lg(n))
REMOVE_DELETE_MIN	O(lg(n))
QUERY_MIN	O(lg(n))

Tabela 6.2: *Consumo de tempo das operações na fila parcial de prioridade, onde n é o número de operações executadas.*

Capítulo 7

Possíveis Aplicações

No escopo deste trabalho não foi possível usar as estruturas numa aplicação maior, apesar delas estarem implementadas e poderem ser testadas. Contudo esta seção, também inspirada pelas sugestões apresentadas por Demaine *et al.*(4), mostra alguns contextos passíveis de utilização.

Em sistemas é bastante comum se deparar com situações em que seria interessante possuir controle do histórico de operações. A retroatividade oferece isso e um passo a mais, ela também propaga as alterações do histórico por toda linha do tempo. Em outras palavras, quando um determinado dado é alterado, a partir daquele momento todos os pontos em que aquele dado foi usado são tratados.

Uma abordagem retroativa poderia ser útil nos seguintes casos:

- **Falhas de segurança:** modificações no sistema podem ser causadas por um usuário sem autorização. Elas devem ser corrigidas, assim como as operações subsequentes a essas alterações.

- **Fontes contaminadas:** um aparelho coletando informações do ambiente pode entrar em mal-funcionamento, e os dados erroneamente coletados durante um período de tempo precisam ser removidos.

- **Recuperação de dados:** após recuperar dados de operações que não foram processadas, retroativamente executá-las no momento correto.

- **Protocolos *online*:** por problemas de conexão, requisições podem acabar sendo enviadas tardiamente ou na ordem errada; elas poderiam ser marcadas com um *timestamp* e executadas retroativamente na hora que foram enviadas, mantendo a ordem de envio.

- **Dinamização:** alguns algoritmos podem ser dinamizados, como é o caso do *Dijkstra* (7). Uma estrutura retroativa é usada para inserir ou remover arcos e o algoritmo consegue atualizar os caminhos eficientemente.

Capítulo 8

Conclusões

Esse trabalho se propunha a introduzir de modo acessível o paradigma de estruturas retroativas, apresentar a implementação das estruturas estudadas e utilizá-las numa aplicação maior.

Ao longo da dissertação foram mostradas detalhadamente as estruturas: fila parcial e totalmente retroativa, pilha totalmente retroativa e fila de prioridade parcialmente retroativa. Em especial, a pilha retroativa não foi encontrada na literatura como as demais, sendo original a implementação apresentada.

Em cada uma das estruturas são usadas estratégias diferentes de implementação, que começam simples e vão se tornando mais elaboradas, mostrando perspectivas não usuais sobre estruturas de dados convencionais. Elas trazem um olhar refrescante sobre estruturas básicas. Dentre elas, a árvore binária de busca merece o grande destaque, pois ela que possibilita a implementação eficiente de todas as estruturas retroativas apresentadas.

Embora o escopo de tempo não tenha permitido aplicar essas estruturas retroativas para resolver um problema mais específico, fica claro que elas são cabíveis de serem usadas em diversos contextos, como segurança, coleta de dados, protocolos, que usufruiriam da flexibilidade de manipular o histórico de operações.

Referências Bibliográficas

[1] **Adel'son-Vel'skii e Landis(1962)** G. M. Adel'son-Vel'skii e E. M. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146. Citado na pág. 4

[2] **Cormen** *et al.***(2009)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to algorithms*. terceira edição. Citado na pág. 6

[3] **Demaine** *et al.***(2015)** E. D. Demaine, T. Kaler, Q. Liu, A. Sidford e A. Yedidia. Polylogarithmic fully retroactive priority queues via hierarchical checkpointing. *Lecture Notes in Computer Science*, 9214. Citado na pág. 1

[4] **Demaine** *et al.***(2007)** Erik D. Demaine, John Iacono e Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(13). Citado na pág. 1, 9, 34, 36, 41

[5] **Dickerson** *et al.***(2010)** M. T. Dickerson, D. Eppstein e M. T. Goodrich. Cloning voronoi diagrams via retroactive data structures. *de Berg M., Meyer U. (eds) Algorithms – ESA 2010*, Vol. 6346. Citado na pág. 1

[6] **Goodrich e Simons(2011)** M. T. Goodrich e J. A. Simons. Fully retroactive approximate range and nearest neighbor searching. *ISAAC 2011. Lecture Notes in Computer Science, vol 7074. Springer, Berlin, Heidelberg*. Citado na pág. 1

[7] **Sunita e Garg(2018)** Sunita e D. Garg. Dynamizing dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. *Journal of King Saud University – Computer and Information Sciences*. Citado na pág. 1, 41