INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Lucas França de Oliveira

ESTUDO SOBRE ESTRUTURAS DE DADOS PERSISTENTES

Trabalho de Graduação
2018

Curso de Engenharia da Computação

CDU 621.38

Lucas França de Oliveira

ESTUDO SOBRE ESTRUTURAS DE DADOS PERSISTENTES

Orientador

Prof. Dr. Carlos Alberto Alonso Sanches (ITA)

ENGENHARIA DA COMPUTAÇÃO

São José dos Campos
Instituto Tecnológico de Aeronáutica

2018

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

França de Oliveira, Lucas
Estudo sobre Estruturas de Dados Persistentes / Lucas França de Oliveira.
São José dos Campos, 2018.
41f.

Trabalho de Graduação – Curso de Engenharia da Computação– Instituto Tecnológico de Aeronáutica, 2018. Orientador: Prof. Dr. Carlos Alberto Alonso Sanches.

1. Estrutura de Dados. 2. Persistência. 3. Implementação. I. Instituto Tecnológico de Aeronáutica. II. Título.

REFERÊNCIA BIBLIOGRÁFICA

FRANÇA DE OLIVEIRA, Lucas. **Estudo sobre Estruturas de Dados Persistentes**. 2018. 41f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Lucas França de Oliveira
TITULO DO TRABALHO: Estudo sobre Estruturas de Dados Persistentes.
TIPO DO TRABALHO/ANO: Trabalho de Conclusão de Curso (Graduação) / 2018

Lucas França de Oliveira
Rua H8A, 118
12.228-460 – São José dos Campos–SP

ESTUDO SOBRE ESTRUTURAS DE DADOS PERSISTENTES

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação

---

Lucas França de Oliveira

Autor

Carlos Alberto Alonso Sanches (ITA)

Orientador

Prof. Dr. Cecília de Azevedo Castro César
Coordenadora do Curso de Engenharia da Computação

São José dos Campos, 10 de novembro de 2018.

Aos amigos da Graduação pelo apoio durante os cinco anos de curso. Aos competidores da Maratona de Programação pela inspiração pelo tema.

Agradecimentos

À Deus, ao nosso Senhor Jesus Cristo e ao Espírito Santo pela paixão e talento com programação competitiva e com o estudo de algoritmos e estruturas de dados.

À minha amada noiva, Marina Barros González Cordeiro, pelo precioso tempo compartilhado com a elaboração deste projeto.

Aos meus pais, Paulo Sérgio Nogueira de Oliveira e Maria das Neves Paiva França de Oliveira, e também aos meus irmãos e resto da minha família, pelo apoio moral, afetivo e financeiro no ingresso ao ITA e nos cinco anos de graduação.

À Divisão de Ciências Fundamentais, à Divisão de Engenharia Eletrônica e à Divisão de Ciência da Computação, e a todo o seu corpo docente, pelo curso de engenharia ministrado nos últimos cinco anos e por todos os conhecimentos nele adquirido.

Ao Professor Carlos Alberto Alonso Sanches, por aceitar ser orientador deste Trabalho de Graduação, por suas orientações nos estudos, codificações e testes das estruturas neste projeto apresentadas e pela revisão deste relatório.

Ao professor Armando, pelo apoio moral, financeiro e de estudo nas edições da Maratona de Programação 2014, 2015, 2016, 2017 e 2018 e pela sua participação como coach.

Aos meus companheiros de equipe da maratona Rodrigo Ferreira Roim (2014), Felipe Vincent Yannik Romero Pereira (2014), Lucas Soares Ferreira (2015, 2016 e 2017), Felipe Viana Sousa (2015 e 2016) e Matheus de Oliveira Leão (2017).

Aos membros da iniciativa ITAbits pelo apoio durante treinamentos, competições, divulgação.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

Resumo

Este trabalho tem por objetivo a apresentação do tema de estruturas de dados persistentes. Para isso, apresentar-se-ão os conceitos respectivos: uma estrutura de dados, em um contexto de operações de atualização e de consulta é dita efêmera se ela guarda apenas sua última versão no tempo. É dita parcialmente persistente se permite consultas sobre qualquer versão anterior. É dita totalmente persistente se permite consultas e atualizações sobre qualquer versão, caso no qual a linha do tempo se ramifica e se torna uma árvore. É dita confluentemente persistente se é totalmente persistente e permite a junção de duas ramificações da linha do tempo métodos genéricos de transformação de estruturas efêmeras em persistentes. Apresentar-se-ão, também, implementações e testes das estruturas comuns apresentadas na disciplina de CES-11 em suas formas totalmente persistentes: pilha $O(1)$, fila $O(logn)$, deque $O(logn)$, fila de prioridades $O(logn)$, árvore de segmentos $O(logn)$ e árvore de busca binária $O(logn)$. Incluir-se-á, também, uma estrutura extra: array confluentemente persistente com consultas do tipo RSQ e RMQ, inserção e junção em $O(logn)$. As implementações serão feitas em C++ e visam a aplicação fácil em competições de programação.

Abstract

Vou escrever isso quando o resumo for aprovado

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ED	Estrutura de Dados
STL	Standard Library

Sumário

1 Introdução

1.1 Objetivo

O objetivo deste trabalho é o estudo das chamadas estruturas de dados persistentes e de métodos genéricos de conversão de estruturas tradicionais em formas persistentes, assim como a implementação na linguagem C++ e teste das principais estruturas de dados vistas nas disciplinas de CES-11, CES-23 e CT-234. Existe pouca referência no acervo bibliográfico do ITA a respeito deste tema, dando a este trabalho o objetivo de incrementar dita referência e complementá-la com os ditos métodos e implementações.

Tem-se, também, o objetivo de fornecer aos participantes da Maratona de Programação, vertente brasileira da competição Association of Computer Machinery - International Collegiate Programming Contest (ACM-ICPC), uma referência para o fácil aprendizado do tema. Este projeto vem anexado a implementações na linguagem C++ de programação, a mais usada na competição, que podem ser rapidamente aplicadas no contexto de programação competitiva.

As técnicas de validação são baseadas na execução de testes de controle e de geração de testes aleatórios de larga escala, usados para aferição do tempo de execução e estimação da complexidade assintótica.

1.2 Motivação

As estruturas de dados persistentes também são conhecidas como estruturas imutáveis ou funcionais, e são de fundamental importância para o funcionamento de linguagens funcionais. Essas linguagens possuem atualmente um espaço considerável na indústria, apresentando-se como uma alternativa para a Programação Orientada a Objetos. Tomaremos o exemplo da linguagem Elixir de programação, derivada da Erlang.

Nesta linguagem, ao se instanciar um objeto, seja em uma declaração de variável ou como parâmetro de chamada de função, atribui-se imediatamente a ele o valor e impossibita-se, no escopo da função, a alteração de seu conteúdo. Para que se possa

FIGURA 1.1 – As linguagens de programação Elixir e Erlang.

atualizar o objeto, é necessário uma nova chamada, no qual durante a criação do escopo é gerada uma nova versão imutável do objeto. Aparentemente, isso torna a linguagem ineficiente, pois toda operação requer a recriação de estrutura inteira. No entanto, isso não ocorre. Como estudaremos mais adiante, internamente a linguagem faz uma alteração conservando a versão anterior, e cada escopo de função tem um ponto de acesso distinto à estrutura. As estruturas de coleção da linguagem Elixir são totalmente persistentes.

As estruturas de dados persistentes também podem ser usadas na otimização de algoritmos em geral, sendo elas uma poderosa ferramenta de resolução de problemas, seja na área de programação competitiva, a qual inspirou este trabalho, seja na área acadêmica ou na indústria. Alguns exemplos de problemas serão abordados nos capítulos seguintes.

2 Persistência

2.1 O conceito de persistência

Uma estrutura de dados é uma forma de organizar dados em programas de computador. Ou seja, guardam algum tipo de informação e cumprem o papel de atualizar ou consultar esta informação de uma forma que seja útil e eficiente para o software que a usa. Neste trabalho, abordaremos as estruturas de dados que suportam dois tipos de operação: de consulta, que extraem alguma informação dela sem alterá-la; e de atualização, que modificam os dados nela contidos.

O conceito de persistência está relacionado à manutenção de algo que, normalmente, seria descartado. Esta propriedade persiste no tempo. No âmbito das estruturas de dados, quando se fala que ela persiste no tempo, esta propriedade é o estado geral da estrutura. Uma operação de atualização, a priori, destrói o estado antigo e gera um novo com alguma modificação. Uma estrutura persistente não descarta o estado antigo, mas em vez disso mantém na memória todas as suas versões representadas de alguma maneira.

Estruturas de dados ordinárias são efêmeras no sentido que, ao se realizar uma operação de atualização, a versão antiga é destruída e apenas a nova versão é mantida na memória. Aqui faremos uma representação em dígrafo das versões do tempo. Cada vértice do grafo representa uma versão da estrutura. Inicialmente o grafo possui apenas um nó - o estado inicial. Este é o nó 0. As operações, seja de consulta ou de atualização, são realizadas em cima de uma versão representada por um nó i. A segunda cria uma nova versão - um novo vértice $j = 1+$ o número do nó mais alto já existente - e o liga à versão antiga no qual a operação inicialmente foi feita. A aresta tem direção da versão antiga para a mais nova. Estruturas efêmeras possuem uma topologia de lista ligada, pois todas as operações só podem ser realizadas em cima do de índice mais alto. O dígrafo de versões é chamado *lista de versões*.

Uma estrutura de dados é dita parcialmente persistente se ela permite realizar consultas em qualquer versão do tempo. Ou seja, o dígrafo continua com a topologia de lista, pois atualizações são permitidas apenas no último nó. Consultas, no entanto, podem ser feitas em cima de qualquer nó i.

Uma estrutura de dados é dita totalmente persistente se ela permite realizar consultas e atualizações em qualquer versão do tempo. Desta forma, pode ocorrer a ramificação da linha do tempo. O dígrafo agora possui uma topologia de árvore e é chamado *árvore de versões*. A raíz dessa árvore é a versão inicial e o histórico de uma versão é o caminho da raíz até o seu nó.

Uma estrutura de dados é dita confluentemente persistente se ela é totalmente persistente e admite uma operação extra de atualização que é aplicada sobre dois ou mais nós do grafo para gerar um ou mais vértices novos. É uma operação de junção ou de atualização que requer informações de várias versões no tempo. O grafo agora possui a topologia de um grafo direcionado acíclico.

Persistência no âmbito de estruturas de dados foi formalmente introduzida por Driscoll, Sarnak, Sleator e Tarjan (DRISCO NEIL SARNAK; TARJAN, 1986), porém já era estudada anteriormente na implementação de linguagens funcionais.

2.2 Métodos genéricos de tornar estruturas de dados parcialmente persistentes

O estudo de estruturas persistentes segue dois caminhos: técnicas gerais para tornar qualquer estrutura de dados parcialmente persistente; ou técnicas para tornar alguma estrutura de dados específica persistente, que serão exemplificadas em capítulos posteriores.

Primeiramente, restringiremos nosso estudo às chamadas *Estruturas de dados linkadas*. São todas as estruturas formadas por um conjunto finito de nós. Cada nó possui um número fixo de campos que podem ser classificados em dois tipos: campo de informação, que detém uma parte de informação de um determinado tipo; campo de ponteiro, que detém um ponteiro para outro nó ou o ponteiro especial NULL. O acesso a uma estrutura de dados linkada é feita por um conjunto finito de pontos de entrada: ponteiros para um ou mais nós. Podemos pensar em uma estrutura de dados linkada como um dígrafo em que os nós possuem grau de saída fixo. São exemplos: pilha implementada com lista ligada, fila implementada com lista ligada, árvore de busca binária, árvore de segmentos.

As operações nesse tipo de estrutura são realizadas normalmente com algoritmos de grafo. São compostas de passos aplicados a algum nó. Um passo inicial é executado em um ou mais nós apontados pelos pontos de entrada. Cada nó processado pode iniciar um novo passo em algum nó por ele apontado. O conjunto de todos os nós processados por uma operação é denominado conjunto dos nós acessados. Se a tarefa for de consulta, a resposta da consulta depende apenas das informações nos nós acessados. Se for de atualização, apenas os nós acessados podem sofrer modificações em seus campos. Este

segundo tipo de operação também pode criar novos vértices, que também fazem parte do conjunto acessado, ou deletar vértices. Definimos o tempo de uma operação como o tamanho do conjunto de acesso, e usaremos esta métrica para o cálculo de complexidade de agora em diante.

O problema abordado nesta seção é desenvolver um método genérico de tornar estruturas tipificadas acima de uma forma efêmera para uma forma persistente. Fazer isso implica em criar uma nova estrutura que possui internamente a representação de todas as versões no tempo simultaneamente. Essa nova estrutura também será linkada ou com alguma extensão, ou seja, usaremos uma estrutura efêmera para simular uma persistente em um tempo de execução próximo da estrutura original. As duas formas mais conhecidas de fazê-lo, além da forma óbvia de fazer a cópia total, são o método do *Nó gordo* e o de *Cópia de nós*.

2.2.1 Método da cópia total para persistência total

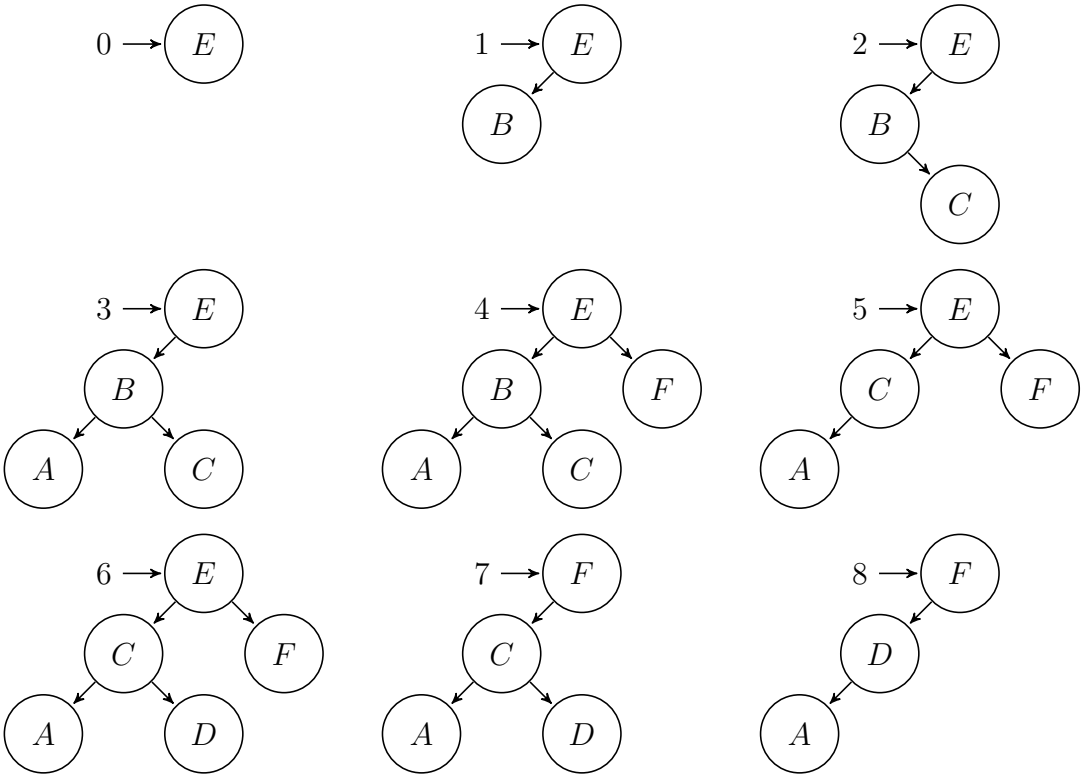

FIGURA 2.1 – Árvore de busca binária totalmente persistente com o método de cópia total. Ponteiros NULL omitidos. Resultado das operações: +E, +B, +C, +A, +F, -B, +D, -E, -C, onde + é inserção e -, remoção.

Este método consiste na cópia profunda completa da estrutura a cada operação de atualização. Mantém-se um conjunto de pontos de entrada pra cada versão. Quando se executa uma atualização sobre uma versão i, copia-se totalmente e profundamente - todos

os campos de ponteiro apontam para uma cópia do nó apontado - a estrutura alcançável a partir da posição i da array de pontos de entrada. Os pontos de entrada da cópia são colocadas ao final da array de pontos de entrada. A partir dela, a operação de atualização é feita normalmente sobre a cópia. Uma vez encerrada a atualização, aquela versão se torna imutável e, portanto, é preservada na história. Para se fazer uma cópia, é necessário apenas a versão a qual se está modificando, o que implica que podemos gerar uma nova versão a partir de qualqeur anterior, garantindo persistência total.

Devido à cópia, cada operação de consulta se torna $O(n + k)$ no tempo, em que k é o número de acessos a nós na operação sobre a versão efêmera, e $O(n + f)$ na memória, em que f é a quantidade de nós criados durante a atualização efêmera. O tempo e espaço de consulta é o mesmo.

2.2.2 Método do Nó gordo para persistência parcial

Este método consiste em gravar, em cada nó, todas as informações de todas as versões de cada campo. Desta forma, toda vez que é feita uma modificação em um nó, não se sobrescreve a informação antiga com uma nova. Em vez disso, Adiciona-se um novo campo ao vértice com a nova informação. Para isso, é necessária a capacidade de tornar um nó arbitrariamente "gordo" - ele deve poder aumentar a quantidade de campos de acordo com a quantidade de atualizações feitas sobre ele. Além disso, cada campo deve ser identificado com um *timestamp*, a versão no tempo à qual aquele campo faz jus. Isso se aplica tanto nos campos de informação quanto de ponteiro.

Assim, a simulação da estrutura efêmera se dá da seguinte maneira: quando um passo acessa um nó já existente para atualizar de uma versão i para uma versão $i + 1$, usam-se os campos com o maior *timestamp* menor que ou igual a i para colher a informação. Em seguida, realiza-se o passo no vértice e, ao aplicar as modificações, gravam-se com um novo *timestamp* $i + 1$ sem remover ou modificar nenhum campo existente de *timestamp* menor que ou igual a i. É garantido que $i+1$ é maior que qualquer campo existente antes da operação pois estamos no caso de persistência parcial. Caso já haja algum campo com o *timestamp* $i + 1$, sobrescreve-se ele. Isso pode acontecer com algoritmos que processam um nó mais de uma vez em uma única operação, os quais não são restringidos para as estruturas de dados linkadas. Na criação de um vértice, ele é criado normalmente, sendo adicionado a cada um de seus campos o *timestamp* da versão nova. No caso de deleção, não se destrói nenhum vértice. As referências a ele na nova versão são setadas como NULL.

Outra modificação necessária é a utilização de uma array de conjuntos de pontos de entrada, em que a posição i guarda o conjunto de pontos de entrada da estrutura para a versão i. Ao se fazer uma operação sobre uma versão i, inicia-se o algoritmo pelos pontos

de entrada da estrutura na posição equivalente da array. Adicionar uma posição nova nessa array será $O(1)$ se o tamanho do conjunto de pontos de entrada da estrutura em qualquer verão for constante, o que é o caso para a maioria das estruturas aqui estudadas. Desta forma, o acesso à estrutura persistente será $O(1)$, assim como na forma efêmera.

Assim, a nova estrutura de dados linkada mantém, internamente, todas as versões no tempo da estrutura de dados linkada original.

A forma parcialmente persistente requer $O(1)$ espaço por nó acessado por operação de atualização. Se, em uma operação, o número de nós acessados na estrutura efêmera for k, na nova também o será, o que garante a mesma complexidade em número de nós acessados. Caso o recolhimento e escrita de campos com um determinado *timestamp* também sejam $O(1)$, o tempo de execução por nó também será de mesma complexidade que na estrutura efêmera.

A forma mais simples de se implementar isso é trocando todos os campos de um nó por uma estrutura de dados do tipo *dicionário* ou *mapa* - uma estrutura que tem um conjunto de chaves e a cada chave tem associado um valor. No nosso caso, a chave é o *timestamp* da versão e o valor associado é o valor daquele campo para aquela versão. Pode-se usar uma *hash table* para isso, que garantirá tempo de acesso aos campos em $O(1)$ amortizado ou um *tree map*, que adicionará um fator $O(\log t)$ em todas as operações, em que t é o número de versões diferentes daquele nó.

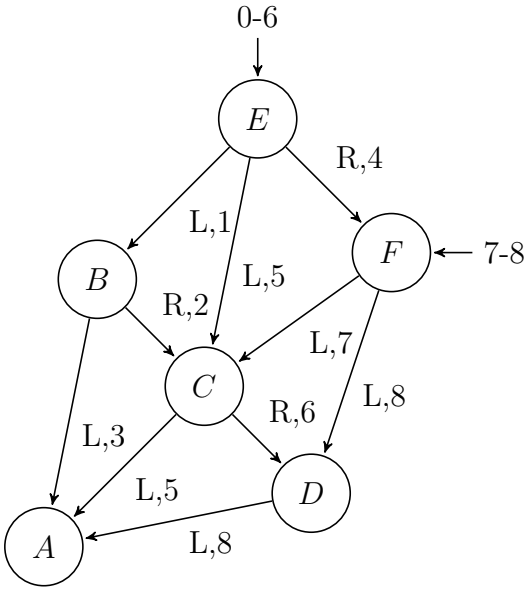

FIGURA 2.2 – Árvore de busca binária parcialmente persistente com o método do nó gordo. Ponteiros NULL omitidos. As arestas possuem a marca L para filho esquerdo, R para direito e a *timestamp*. Resultado das operações: +E, +B, +C, +A, +F, -B, +D, -E, -C, onde + é inserção e -, remoção.

O motivo deste método funcionar para a persistência parcial consiste no fato da linha do tempo de um nó - da mesma forma que a linha do tempo da estrutura inteira - ser

uma lista. Se houve uma modificação em um tempo i e a seguinte em $j > i$, todos os valores durante as versões k, $i < k < j$, pertencem à mesma linha do tempo. Como j foi a primeira atualização depois de i, o valor para todo k, $i < k < j$, é o de i. Por isso, ao se procurar um valor em um nó para a versão i, procura-se o que tem maior *timestamp* menor que ou igual a i.

Para a persistência total, por outro lado, a linha do tempo se ramifica e forma uma árvore, o que faz com que o método de busca de *timestamp* citado se torne incorreto. Considere o seguinte exemplo: um nó x é criado na em uma versão 1 com valor A. Na atualização 2 a partir de 1, é colocado um valor B nele. Em uma atualização 3 a partir de 1 também, o nó não é acessado. Ao se fazer uma consulta ao nó x versão 3, ele deveria retornar A, pois a versçao 3 é derivada da versão 1. No entanto, o valor encontrado é B, pois possui o maior *timestamp* (2) menor que ou igual a 3. A modificação necessária para se corrigir isso seria manter uma estrutura de árvore T em que cada nó é uma versão e aponta para sua versão de origem. Para saber que *timestamp* usar ao se consultar um nó x procurando pela versão i, procura-se em T o maior *timestamp* que está presente no nó x no caminho do vértice i até a raíz. Entretanto, fazer essa busca causaria um aumento maior que $O(1)$ ou $O(\log t)$ no tempo de acesso de um nó.

2.2.3 Método de cópia de nós para persistência parcial

Para resolver o problema do acesso a um campo de *timestamp* i causado pela adição do *dicionário*, introduzimos o método de cópia de nós. Neste método, não usaremos dicionário e os nós não ficarão gordos: eles terão apenas um número constante de campos semelhantes aos da estrutura efêmera. Eles terão também uma *timestamp* de qual versão eles fazem parte, um ponteiro *leader* que aponta para o líder da família, como será definido abaixo.

Se um nó x existe na estrutura efêmera, existirá uma família de nós na versão persistente em que cada nó dela representa uma versão no tempo de x. Essa família representa a lista de versões daquele nó de acordo com modificações. A adição de elementos é feita de forma crescente em *timestamp*. Ela também possui um líder, um nó especial que possui um ponteiro para o nó da família com o maior *timestamp*. Todos os membros de uma família possuem um ponteiro para um mesmo líder. Este serve para que possamos, a partir de qualquer membro de uma família, acessar o membro mais recente em $O(1)$. Uma forma alternativa seria cada membro apontar para o mais recente de menor *timestamp*, montando uma lista ligada em ordem crescente de *timestamp*, mas o acesso ao elemento mais recente seria $O(t)$, caso implementado da forma mais simples. Técnicas mais avançadas podem amortizar para $O(1)$, mas nesse caso usaremos um nó líder.

Assim como no método do nó gordo, temos uma array de conjunto de pontos de entrada

que funcionará exatamente da mesma maneira: quando uma nova versão i for criada, uma nova posição i será adicionada com os pontos de entrada da estrutura equivalentes a essa nova versão.

No primeiro método, as operações de consulta funcionam da mesma maneira que na estrutura efêmera. A única diferença é que, ao se acessar um campo em um nó por uma versão na versão i, procura-se a versão com maior *timestamp* que não seja maior que i. Neste método, é ainda mais fácil: cada nó possui apenas uma versão de cada campo, então a simulação de operações de consulta é exatamente a mesma. Basta fazer a entrada pela posição certa na array de conjunto de pontos de entrada.

As operações de atualização da versão i para $i+1$, por sua vez, são feitas da seguinte maneira: Inicializa-se um conjunto S de nós criados. Ao se criar um novo vértice, coloca-se sua *timestamp* como $i+1$ e seus campos são preenchidos com os valores padrão, sendo NULL para todos os ponteiros exceto *leader*. Coloca-se esse novo vértice em S. Quando um novo nó é criado na estrutura efêmera, inicia-se uma nova família na persistente. Para isso, cria-se um novo líder nessa família e o primeiro nó com as informações iniciais, um apontando para o outro. Informações específicas daquele passo são feitas mediante uma modificação. Ao se deletar um vértice na estrutura efêmera, todas as referências a qualquer membro da família dele na versão $i+1$ são setadas como NULL. Ao se acessar (não necessariamente modificar), durante um passo de um algoritmo, um nó x de versão $j \leq i$, verifica-se por meio de *leader* se já foi criado um de versão $i+1$. Caso positivo, aplicam-se as modificações nesse nó. Caso contrário, copia-se esse nó, colocando a *timestamp* $i+1$ nele e fazendo as modificações no novo vértice. Faz-se o líder da família apontar para esse novo nó, e adiciona-o a S. Todos os campos de ponteiros da cópia devem apontar para a versão mais recente de seus apontados, ou seja, se temos um nó recém copiado x e um campo de ponteiro p, faz se $x->p = x->p->leader->recent$. Caso haja acesso a um de *timestamp* $i+1$, modifica-se direto nele normalmente. No final de uma operação de atualização, processam-se todos os nós em S fazendo com que qualquer campo de ponteiro aponte para a versão mais recente da família do nó apontado.

Este método funciona de forma semelhante ao método do nó gordo no sentido que é mantida, para cada vértice, uma lista de todos os valores no tempo daquele nó. Cada elemento da lista é identificado com um *timestamp*. No primeiro método, essa lista é implementada dentro do nó por meio de um *dicionário*. Neste, ela é implícita: o vínculo de duas versões acontece por meio da cópia. A otimização está no fato de não ser necessária a busca pelo maior *timestamp* menor que ou igual a i em cada nó: o nó precedente, que possui a referência para o nó atual, já possui a referência para a versão certa. Todas as operações adicionadas em cima das tradicionais possuem complexidade $O(1)$ em tempo e em número de nós acessados. Ou seja, se uma operação de atualização acessar k nós, serão acessados e criados $O(k)$ novos nós. O espaço, para uma operação $O(1)$, torna-se

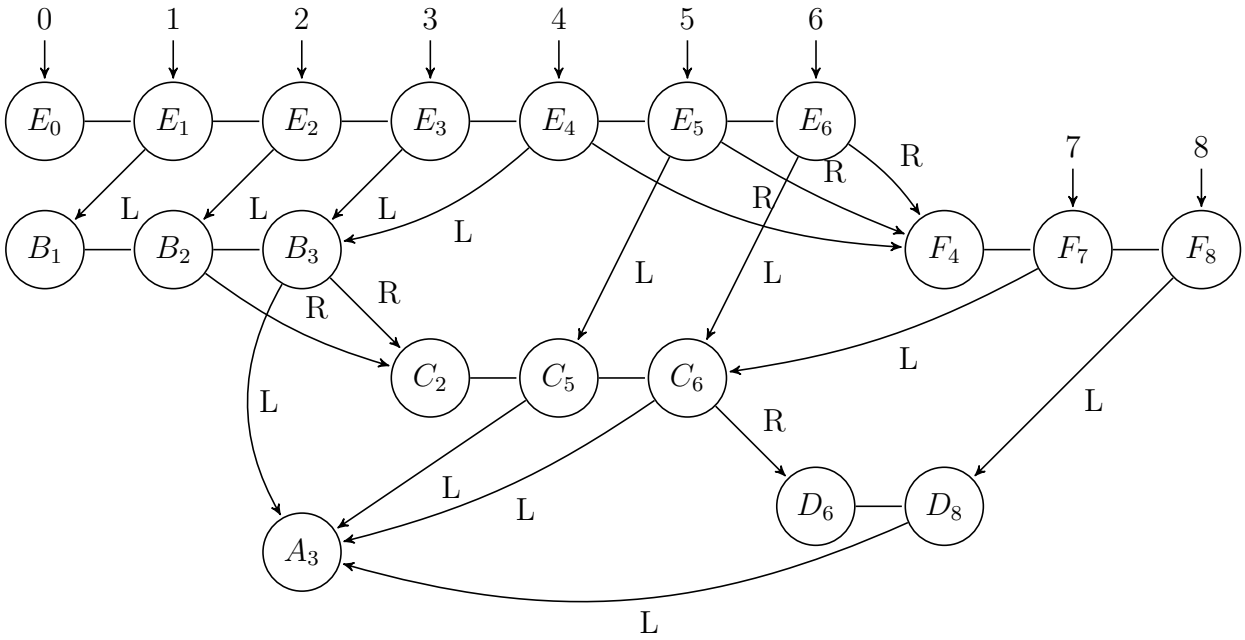

FIGURA 2.3 – Árvore de busca binária parcialmente persistente com o método de cópia de nós. Ponteiros NULL omitidos. As arestas possuem a marca L para filho esquerdo e R para direito. Nós de mesma família estão ligados por traços. Líderes de família omitidos. Nós possuem o valor e a *timestamp*. Resultado das operações: +E, +B, +C, +A, +F, -B, +D, -E, -C, onde + é inserção e -, remoção.

$O(k)$. Consultas são exatamente iguais.

Na persistência total, a família de um nó pode adquirir a topologia de uma árvore. Uma ramificação acarretaria na existência de duas versões "mais recentes" de uma família, o que acarretaria o nó de líder se tornar gordo, aniquilando a vantagem deste método em relação ao do nó gordo. Ademais, como as atualizações podem ser feitas sobre qualquer versão do tempo, o acesso a uma versão mais recente no tempo não seria suficiente. O líder deveria manter um histórico completo - a árvore de versões T completa - para poder orientar as atualizações a fazer as modificações corretas. Recaímos aqui no mesmo problema do método do nó gordo.

2.2.4 Método de cópia de nós para persistência total em caso de árvore

Acima vimos as vantagens e desvantagens de usar os dois métodos para a implementação de persistência parcial. No geral, vimos que o método do nó gordo consegue implementá-la com um fator extra máximo de $O(\log t)$ para cada operação, enquanto o de cópia de nós possui complexidade $O(1)$ extra para cada acesso a um nó durante uma operação. Ambos possuem $O(k)$ extra em memória para operações que acessam k nós.

No entanto, nenhum dos métodos vistos acima suporta a persistência total, que é o

principal alvo das aplicações das estruturas de dados persistentes apresentadas na introdução deste trabalho. Porém, existe um caso particular de estruturas de dados linkadas em que o método de cópia de nós permite a implementação da persistência total. Este é o caso em que cada nó possui no máximo um ponteiro apontando para ele na versão efêmera, incluindo os ponteiros de entrada. A estrutura efêmera possui, portanto, topologia de árvore, na qual a raíz é apontada por um ponto de entrada da estrutura.

Modifica-se o método da seguinte maneira: não é mais necessária a presença do líder, pois a única referência a um nó y deve ser aquela que fez a cópia de y, portanto não é necessário verificar se já existe uma versão mais recente. Se existe, o nó que está acessando terá referência a ela e não fará uma nova cópia. Isso permite também eliminar a *timestamp*. O vértice persistente finalmente se assemelha totalmente à sua versão efêmera. A deleção e inserção de um nó funcionam da mesma maneira. O acesso é sempre feito sobre uma cópia $\overline{x}$ de um nó x durante um passo de uma atualização. Antes do primeiro passo, faz-se uma cópia dos nós apontados pelos pontos de entrada. Caso um passo se propague para um nó y, gera-se uma cópia $\overline{y}$ e atualizam-se todas as referências a y em $\overline{x}$ para $\overline{y}$. Caso contrário, mantém-se a referência ao y original.

Perceba que isso faz com que os nós se tornem imutáveis: nenhuma operação, seja de consulta ou de atualização, modificará um campo em um nó já criado. Se forem observados apenas os nós alcançáveis a partir do ponto de entrada da versão i, a estrutura será perfeitamente igual à sua versão efêmera para aquela versão. Portanto, pode-se acessar qualquer versão no tempo da estrutura, garantindo a persistência parcial. Além disso, percebe-se que a única coisa necessária para gerar uma nova versão da estrutura é a estrutura antiga em si, a qual temos acesso. Assim, obtém-se persistência total.

A topologia de nossa estrutura persistente é de um grafo direcionado acíclico, sendo cada apontada por um ponteiro de entrada da estrutura. Uma operação que acessa um nó x necessariamente acessa todos os nós descendentes de x em um caminho único até o ponto de entrada da respectiva versão, o que implica que, se existe uma cópia $\overline{x}$ de x, ela é apontada por uma lista de cópias do caminho de x até a raíz. Desta forma, se uma operação na forma efêmera acessa k nós, ela irá criar k novos nós na forma persistente, o que aumenta a complexidade de memória de $O(1)$ para $O(k)$, mas mantém a complexidade de tempo e de nós acessados em $O(k)$, os mesmos resultados do método de cópia de nós tradicional.

Essa abordagem nos permitirá fazer a implementação de todas as estruturas de dados propostas como objetivo deste trabalho, pois são, em sua maioria, estruturas em árvore. As estruturas de pilha, fila e deque serão implementadas com métodos ad hoc - métodos específicos para fazer aquela estrutura funcionar de forma persistente.

3 Pilha persistente

3.1 Versão efêmera

A pilha é uma estrutura de dados que mantém uma lista de elementos. Um lado dessa lista é definido como a base e o outro como o topo. Visualmente, desenha-se a lista na vertical, com a base embaixo. É uma estrutura muito importante, tendo aplicações em diversos algoritmos nas mais diversas áreas da indústria. Alguns exemplos: implementação da pilha em uma linguagem de programação que admite recursão, solução de diversos problemas matemáticos, entre outros. Ela é lecionada na disciplina de CES-11.

As operações suportadas são:

- $push(x)$: insere um elemento x no topo da pilha.
- $top()$: retorna o elemento no topo da pilha.
- $pop()$: remove o elemento no topo da pilha.
- $size()$: retorna o tamanho da pilha.
- $clear()$: limpa a pilha (esvazia a lista).

A biblioteca STL da linguagem C++ possui uma implementação da pilha que suporta as 4 primeiras operações. Aqui, apresentaremos uma implementação nossa com lista ligada, de forma a que a pilha se configure como uma estrutura de dados linkada.

Primeiramente, definimos nosso nó:

```
struct Node {
   int data;
   Node *nxt;
   Node(Node *_nxt, int _data) :
      nxt(_nxt), data(_data) { }
};
```

Ele possui um número inteiro *data* que representa as informações no nó. Esse tipo pode facilmente ser estendido para outros. Adiciona-se, também, um construtor para facilitar a alocação de nós.

A pilha em si possui um ponteiro para o topo e um contador para o tamanho. A lista ligada está no sentido do topo para a base, o que facilita o acesso ao novo topo durante a remoção. Em sua inicialização, o primeiro começa como NULL e o segundo, zero.

```
class Stack {
private:
    Node* back;
    int size;
public:
    Stack() : back(NULL), size(0) { }
```

A operação clear deve deletar todos os nós e reinicializar a pilha com os valores iniciais. Ela deve também ser chamada pelo destrutor para garantir que não haja vazamento de memória. Neste exemplo, a pilha faz todo o gerenciamento de memória internamente.

```
    ~Stack() { Clear(); }
    void Clear() {
        Node *aux;
        while(back != NULL) {
            aux = back;
            back = back->nxt;
            delete aux;
        }
        size = 0;
    }
```

O ponteiro *back* aponta para o topo da pilha, portanto a operação *top* deve retornar seu conteúdo. O nó no topo aponta para o próximo, portanto a operação *pop* deve deletar o nó *back* antigo e deve fazer este ponteiro avançar para o próximo, além de decrementar *size*. A inserção de um novo nó valor deve incrementar o tamanho *size*, criar um novo nó, fazê-lo apontar para o topo antigo e fazer *back* apontar para ele. Assim, as operações *top*, *pop* e *push* são implementadas da seguinte maneira:

```
    int Top() { return back->data; }
    void Pop() {
        Node *prv = back;
        back = back->nxt;
        delete prv;
        size--;
    }
    void Push(int data) {
        back = new Node(back, data);
        size++;
```

```
    }
```

Encerrando a classe, adicionamos as funções *size* e *empty*, que são verificações triviais sobre a variável *size*.

```
    int Size() { return size; }
    bool Empty() { return size == 0; }
};
```

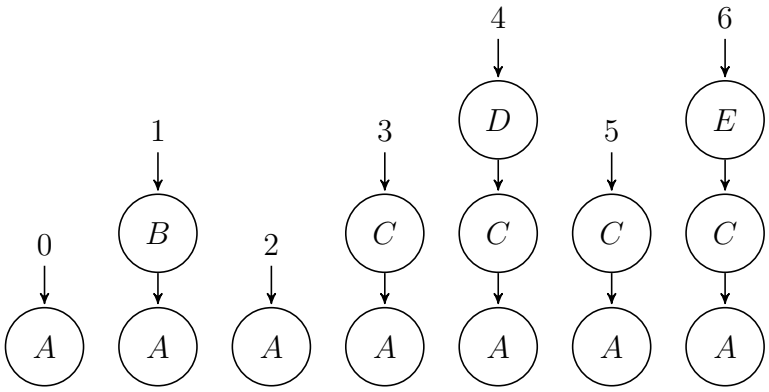

FIGURA 3.1 – As versões no tempo de uma pilha, os pontos de entrada representando as versões no tempo de *back*. Resultado das operações: +A, +B, -, +C, +D, -, +E, onde + é inserção (*push*) e -, remoção (*pop*). Ponteiros NULL omitidos.

3.2 Versão totalmente persistente

Entre as estruturas de dados propostas na introdução, essa é uma das que apresenta o requisito para aplicação da persistência total mostrado na última seção do capítulo anterior. Na versão efêmera, cada nó da pilha possui uma única referência: um nó anterior ou o ponteiro *back*, caso ele seja o topo. Aplicaremos então o método de cópia de nós simplificado para alcançar persistência total. Como os nós na versão efêmera já são naturalmente imutáveis, o método do nó gordo acaba por ser o mesmo que o de cópia simplificada. Com efeito, neste último, não é feita a cópia de vértice algum.

Com efeito, esta é uma estrutura em que não é necessário efetuar a cópia de nós. Na versão efêmera, não se faz modificação alguma em um nó depois que ele é criado. Os nós efêmeros já são imutáveis. O único campo que muda é o ponteiro *back*. Portanto, modificamos o ponteiro *Node *back* para uma uma lista de ponteiros *vector<Node*> back* usando a classe *vector* da C++ STL. Ela implementa uma array que cresce de tamanho ao adicionar elementos no final. Todas as suas operações são amortizadas em $O(1)$. $back[i]$ representa o ponteiro para o topo da pilha na versão i. Faremos, também, uma modificação no nó: o tamanho da pilha é armazenado no nó, em vez da classe principal.

Primeiramente, definimos nosso nó persistente:

```
struct PNode {
    int data, size;
    PNode *nxt;
    PNode(PNode *_nxt, int _data, int _size) :
        nxt(_nxt), data(_data), size(_size) { }
};
```

data e *nxt* funcionam da mesma maneira que na versão efêmera *size* representa o tamanho da pilha a partir daquele nó. Adiciona-se, também, um construtor para facilitar a alocação de nós.

Temos agora nossa array *back* e uma array com todos os nós para gerenciamento de memória "garbage collection". Em sua inicialização, *back*[0] se inicia com o ponteiro NULL.

```
class PersistentStack {
private:
    std::vector<PNode*> back;
    std::vector<PNode*> nodes;
public:
    PersistentStack() { back.push_back(NULL); }
```

A operação clear deve deletar todos os nós, limpar e reinicializar *back* com os valores iniciais. Ela deve também ser chamada pelo destrutor para garantir que não haja vazamento de memória. Neste exemplo, a pilha faz todo o gerenciamento de memória internamente. Desta vez é necessário usar o método de "garbage collection" para se efetuar a deleção dos nós corretamente.

```
    ~PersistentStack() { Clear(); }
    void Clear() {
        for(int i = 0; i < (int)nodes.size(); i++) {
            delete nodes[i];
        }
        back.clear(); back.push_back(NULL);
        nodes.clear();
    }
```

O ponteiro *back*[*i*] aponta para o topo da pilha na versão i, portanto a operação *top* deve retornar seu conteúdo na respectiva versão. O nó no topo aponta para o próximo, portanto a operação *pop* deve colocar no final de *back* um ponteiro para o nó seguinte a *back*[*i*]. A inserção de um novo nó valor deve criar um novo nó, fazê-lo apontar para o topo *back*[*i*] e colocá-lo no final de *back*. Todas as operações de atualização retornam o índice da nova versão em vez de serem *void*. Assim, as operações *top*, *pop* e *push* são implementadas da seguinte maneira:

```
    int Top(int ver) { return back[ver]->data; }
    int Pop(int ver) {
        back.push_back(back[ver]->nxt);
        return int(back.size())-1;
    }
    int Push(int ver, int data) {
        back.push_back(new PNode(back[ver], data, Size(ver) + 1));
        nodes.push_back(back.back());
        return int(back.size())-1;
    }
```

Encerrando a classe, adicionamos as funções *size*, *empty* e *lastestVersion*, que são verificações triviais sobre os nós apontados por *back* em suas respectivas versões ou sobre a array em si.

```
    int Size(int ver) {
        if (back[ver] == NULL) return 0;
        return back[ver]->size;
    }
    int Empty(int ver) { return Size(ver) == 0; }
    int LatestVersion() { return int(back.size())-1; }
};
```

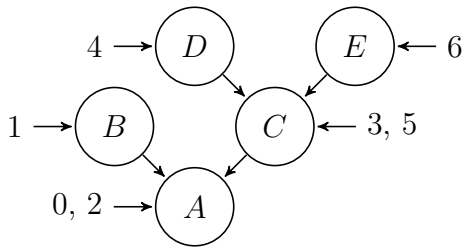

FIGURA 3.2 – Pilha totalmente persistente pelo método de cópia de nós simplificado, o ponteiro de entrada i representando $back[i]$. Resultado das operações: +A, +B, -, +C, +D, -, +E, onde + é inserção (*push*) e -, remoção (*pop*). Ponteiros NULL omitidos.

3.3 Testes

Os testes desta estrutura de dados foram separados em testes de resposta correta e testes de tempo. Para ambos, foram executados 1000 casos de testes. Cada caso de teste compreende em uma sequência de 10000 operações de atualização, seguidos da verificação sobre todas as versões criadas em relação às operações de consulta *size*, *top* e *empty*.

Foi utilizada, para controle, a pilha da C++ STL. Ela foi usada para implementar a persistência total pelo método de cópia total, em que a estrutura é copiada inteiramente a

cada modificação. Nos testes de resposta correta, a bateria foi executada com as mesmas entradas para nossa estrutura e para a de controle, tendo suas saídas comparadas nos testes de consulta. Eles apresentaram resposta correta em todos os casos. Nos testes de tempo, a bateria de testes foi realizada separadamente entre a versão de controle e a nossa, tendo seus tempos medidos. Todos os testes foram feitos tanto com persistência parcial quanto total. Os resultados, média de cada um dos 1000 casos de teste, de tempo estão na tabela a seguir:

TABELA 3.1 – Comparação de tempos de execução da pilha persistente

	Persistência Parcial (μs)	Persistência Total (μs)
Método de cópia total	7598	7062
Método de cópia de nós	713	1024

Observa-se uma considerável redução no tepmo de execução entre ambos os métodos. Deve-se levar em consideraçaoq ue o primeiro é implementado completamente por meio de cópias de classes *stack* da C++ STL e, portanto, possui uma constante alta de operação, mas que se torna insignificante se comparada à cópia da estrutura. O segundo faz uso de alocação dinâmica de momória e, também, da classe *vector* da C++ STL, o que trás uma constante de complexidade muito mais alta. Apesar disso, nossa estrutura se mostrou cerca de sete vezes mais rápida que $default$ do C++ copiada.

4 Fila persistente

4.1 Versão efêmera

A fila é uma estrutura de dados que mantém uma lista de elementos. Um lado dessa lista é definido como a frente e o outro como a trás. Visualmente, desenha-se a lista na horizontal, não importando o lado de frente e de trás. É uma estrutura muito importante, tendo aplicações em diversos algoritmos nas mais diversas áreas da indústria. Alguns exemplos: implementação da busca em largura sobre grafos, implementação de sistemas de fila de espera, entre outros. Ela é lecionada na disciplina de CES-11.

As operações suportadas são:

- $push(x)$: insere um elemento x no final da fila.
- $front()$: retorna o elemento na frente da fila.
- $pop()$: remove o elemento na frente da fila.
- $size()$: retorna o tamanho da fila.
- $clear()$: limpa a fila (esvazia a lista).

A biblioteca STL da linguagem C++ possui uma implementação da fila que suporta as 4 primeiras operações. Aqui apresentaremos uma implementação nossa com lista ligada, de forma a que a fila se configure como uma estrutura de dados linkada.

Primeiramente, definimos o nosso nó, muito semelhante ao da pilha:

```
struct Node {
   const int data;
   Node *nxt;
   Node(Node *_nxt, int _data) :
      nxt(_nxt), data(_data) { }
};
```

Ele possui um número inteiro *data* que representa as informações do nó. Esse tipo pode facilmente ser estendido para outros. Adiciona-se, também, um construtor para facilitar a alocação de nós.

A fila em si possui um ponteiro para frente e um para a trás, além de um contador para o tamanho. A lista ligada funciona no sentido da frente para trás, o que facilita o acesso à próxima frente durante a remoção. Em sua inicialização, os ponteiros se iniciam como NULL e o contador, como zero.

```
class Queue {
private:
    Node *front, *back;
    int size;
public:
    Queue() : front(NULL), back(NULL), size(0) {}
```

A operação *clear* deve deletar todos os nós e reinicializar a fila com os valores iniciais. Ela deve também ser chamada pelo destrutor para garantir que não haja vazamento de memória. Neste exemplo, a fila faz todo o gerenciamenteo de memória internamente.

```
    ~Queue() { Clear(); }
    void Clear() {
        while (front != NULL) {
            Node *aux = front;
            front = front->nxt;
            delete aux;
        }
        back = NULL;
        size = 0;
    }
```

O ponteiro $front$ aponta para a frente a fila, portanto a operação $front$ deve retornar seu conteúdo. O nó na frente aponta para o próximo, portanto a operação *pop* deve deletar o nó $front$ antigo e deve fazer esse ponteiro avançar para o próximo, além de decrementar *size*. Caso esse próximo seja NULL, deve-se fazer *back* apontar para NULL também. A inserção de um elemento deve incrementar *size* e deve alocar um novo nó na frente de *back* e deve-se fazer este ponteiro avançar. Caso o *back* antigo seja NULL, deve-se fazer $front$ apontar para o novo vértice. Este inicialmente aponta para NULL. Assim, as operações $front$, *pop* e *push* são implementadas da seguinte maneira:

```
    int Front() { return front->data; }
    void Pop() {
        Node *prv = front;
        front = front->nxt;
        delete prv;
        size--;
```

```
        if (front == NULL) back = NULL;
    }
    void Push(int data) {
        if (front != NULL) back = back->nxt = new Node(NULL, data);
        else front = back = new Node(NULL, data);
        size++;
    }
```

Encerrando a classe, adicionamos as funções *size* e *empty*, que são verificações triviais sobre a variável *size*.

```
    int Size() { return size; }
    bool Empty() { return size == 0; }
};
```

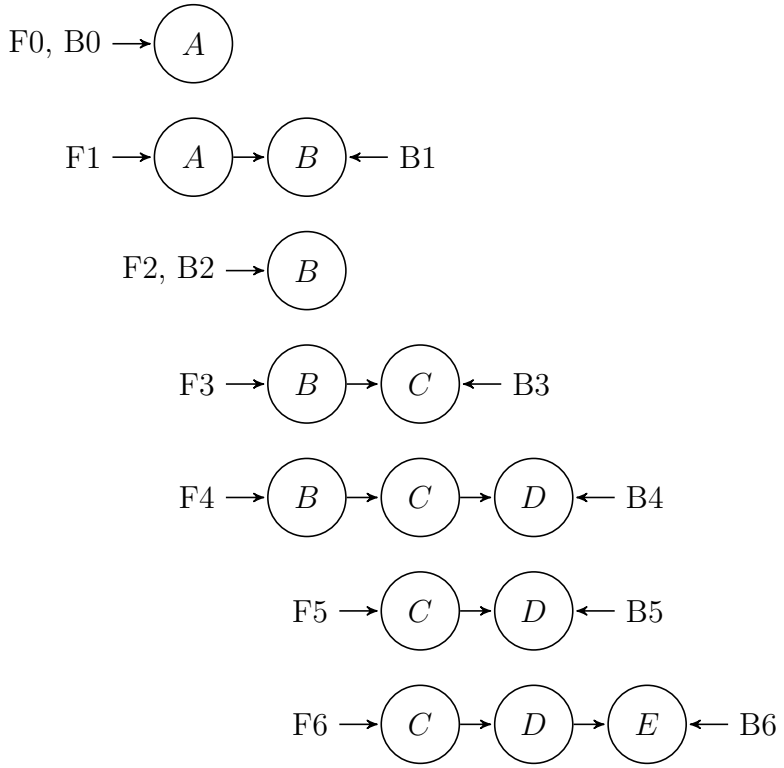

FIGURA 4.1 – As versões no tempo de uma fila, os pontos de entrada representando as versões no tempo de *front* e *back*. Resultado das operações: +A, +B, -, +C, +D, -, +E, onde + é inserção (*push*) e -, remoção (*pop*). Ponteiros NULL omitidos.

4.2 Versão parcialmente persistente

Sendo uma estrutura com nós não imutáveis - a inserção muda o campo *nxt* do nó apontado por *back* -, os métodos do nó gordo e o de cópia de nós se tornam métodos diferentes. Sendo também uma estrutura em que um nó pode ser apontado por dois

ponteiros - *nxt* de outro vértice e *back* -, não se pode aplicar a versão simplificada do de cópia de nós para persistência total. Esta estrutura se torna aplicável para os métodos apresentados no caítulo 2. Entretanto, caso aplicássemos o nó gordo, cada *dicionário* teria no máximo duas entradas e, para o de cópia de nós, cada um seria copiado uma única vez. Existe, em vez disso, um método *adhoc* que, internamente, implementa a mesma coisa que ambos os métodos citados, mas não possui as altas contantes de acesso a um *dicionário* ou de copiar um nó - alocação dinâmica de memória.

Este método é dito *adhoc* pois advém de uma propriedade específica da fila e, portanto, pode somente ser aplicado a ela. Trata-se fo fato das operaçções de atualização somente moverem pontos de entrada ou alterarem uma única vez um ponteiro de um campo de NULL para um novo nó. A descrição de como usar isso de forma a implementar a persistência parcial pe dada a seguir:

Primeiramente, definimos a trás da fila não como o nó que aponta para NULL, mas o que é apontado pelo ponteiro *back* relativo à respectiva versão da fila. Observe que, em uma remoção, o ponteiro $front$ avança e a estrutura perde totalmente acesso à frente antiga. A nova fila é um subconjunto da antiga e, portanto, podemos representá-la utilizando os nós da antiga. A nova versão não interferirá com a antiga pois não modifica nenhum nó. Na adição, modifica-se *nxt* do elemento apontado por *back*. No entanto, como o fim da fila da versão antiga não muda, esse campo será inútil na representação da versão antiga. Assim, podemos representar a fila como uma única lista ligada.

O nó usado é o mesmo da versão efêmera:

```
struct Node {
   const int data;
   Node *nxt;
   Node(Node *_nxt, int _data) :
      nxt(_nxt), data(_data) { }
};
```

Precisamos agora de uma array de ponteiros *back* e $front$ em que a i-ésima posição representa os pontos de entrada da versão i. Estes inicialmente são NULL. Teremos também uma array de tamanhos *size* em que a i-ésima posição representa o tamanho na versão i. A posição inicial é zero. Adicionaremos também uma array que contém ponteiros para todos os nós. Assim como na pilha totalmente persistente, usaremos essa array para fazer o controle de memória no estilo "garbage collection", ou seja, a deleção ocorrerá exclusivamente no método *clear*.

```
class AQueue {
    std::vector<Node*> front, back, nodes;
    std::vector<int> size;
public:
    AQueue() {
```

```
        front.push_back(NULL);
        back.push_back(NULL);
        size.push_back(0);
    }
```

A função *clear* deve deletar todos os nós já alocados, que estão armazenados em *nodes*. Ela deve também limpar e reinicializar $front$, *back* e *size*. O destrutor deve chamá-la.

```
    ~AQueue() { Clear(); }
    void Clear() {
        for(int i = 0; i < (int)nodes.size(); i++) {
            delete nodes[i];
        }
        nodes.clear();
        back.clear(); back.push_back(NULL);
        front.clear(); front.push_back(NULL);
        size.clear(); size.push_back(0);
    }
```

A operação $front$ deve apenas fazer uma acesso à frente da versão desejada. A operação *pop* insere no final de $front$ o nó seguinte na lista à frente da últiam versão. Caso este seja nulo, insere-se no final de *back* NULL, caso contrário insere-se o vértice apontado pelo final de *back*. A operação *push* cria um novo nó com o novo valor. Caso o final de *back* aponte para NULL, adiona-se o novo nó no final tanto de *back* quanto de $front$. Caso contrário, faz-se o campo *nxt* do *back* antigo apontar para ele e adicoina-se o novo nó ao final de *back*. No final de $front$ adiona-se a frente antiga. *size* é atualizado de acordo.

```
    int Front(int ver) { return front[ver]->data; }
    void Pop() {
        front.push_back(front.back()->nxt);
        if (front.back() == NULL) back.push_back(NULL);
        else back.push_back(back.back());
        size.push_back(size.back()-1);
    }
    void Push(int data) {
        Node *newNode = new Node(NULL, data);
        nodes.push_back(newNode);
        if (front.back() != NULL) {
            back.back()->nxt = newNode;
            back.push_back(newNode);
            front.push_back(front.back());
        }
        else {
            back.push_back(newNode);
            front.push_back(newNode);
```

```
        }
        size.push_back(size.back()+1);
    }
```

Encerrando a classe, temos as funções *size*, *empty* e $lastestVersion$, verificações triviais sobre a array *size*.

```
    int Size(int ver) { return size[ver]; }
    bool Empty(int ver) { return size[ver] == 0; }
    int LatestVersion() { return int(back.size())-1; }
};
```

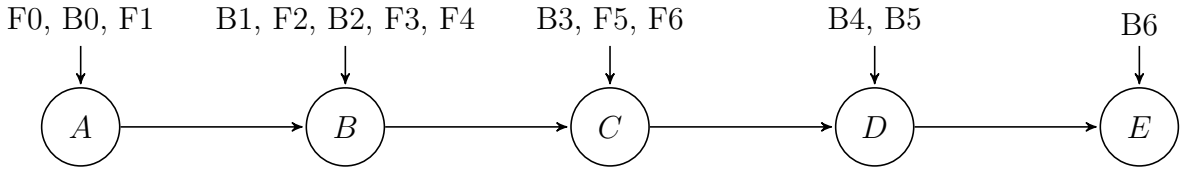

FIGURA 4.2 – Fila parcialmente persistente, os pontos de entrada representando as versões no tempo de $front$ e *back*. Resultado das operações: +A, +B, -, +C, +D, -, +E, onde + é inserção (*push*) e -, remoção (*pop*). Ponteiros NULL omitidos.

4.3 Versão totalmente Pprsistente

Referências

DRISCO NEIL SARNAK, D. D. S. J. R.; TARJAN, R. E. Making data structures persistent. **Concurrency in Practice and Experience**, v. 38, n. 1, p. 86–124, fev. 1986. Disponível em: <https://www.cs.cmu.edu/~sleator/papers/making-data-structures-persistent.pdf>. Acesso em: 20 apr. 2018.

Apêndice A - Topicos de Dilema Linear

A.1 Ancestral de Nível

Suponha que temos uma árvore T de n nós e raíz 1. Queremos representá-la de uma forma que nos permita responder á seguinte pergunta de forma eficiente: dado um vértice u, qual o seu i-ésimo ancestral? Ou seja, se cada nós tiver um ponteiro pro seu pai e andarmos por esse ponteiro i vezes a partir de u, em qual nó pararemos?

A forma mais simples é representando a árvore por nós com o ponteiro para o pai. Cada nó guardará $O(1)$ de memória, tornando a estrutura inteira $O(n)$ de memória. Desta forma, a consulta pode ser respondida por meio de uma iteração de i elementos por esse ponteiro a partir de u. No pior caso, o tempo fica $O(n)$, caso em que a árvore é uma lista.

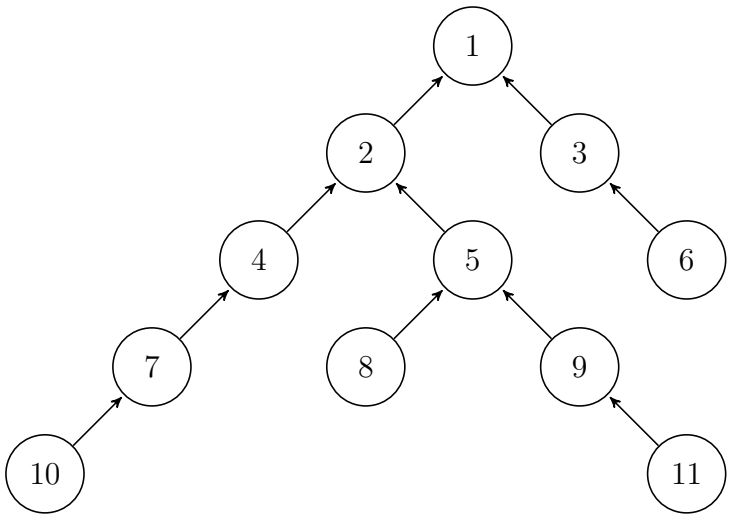

FIGURA A.1 – Representação de uma árvore por nós e ponteiro para o pai. A raíz aponta para NULL e tem seu pai omitido. Árvore $O(n)$ memória e $O(n)$ por consulta.

Uma outra forma de se representar é cada nó possuindo uma array com todo o caminho da raíz até ele. Isso permitirá fazer a consulta em tempo $O(1)$, mas a construção de cada nó requerirá $O(n)$ de tempo: Constrói-se a partir da raíz, cada nó puxa a lista do pai e

adiciona ele próprio. Cada nó requerirá, também, $O(n)$ campos de memória.

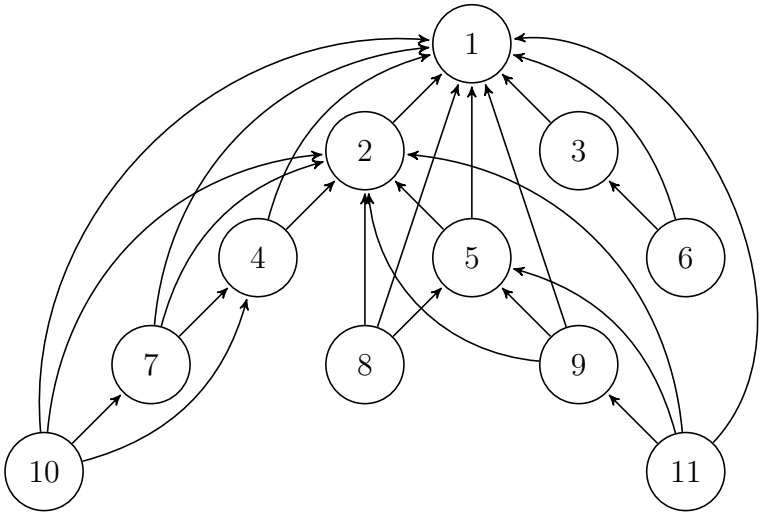

FIGURA A.2 – Representação de uma árvore por nós e ponteiro para todos os nós ancestrais. Árvore $O(n^2)$ memória e $O(1)$ por consulta.

Essa forma ainda não é interessante devido ao alto custo em memória. Usando o paradigma da divisão e conquista, usaremos a seguinte representação, que é um meio termo das duas apresentadas acima: Cada nó guardará um ponteiro para um ancestral se e somente se a distância destes for uma potência de dois. Ou seja, cada nó terá uma array *par* cuja posição i representa o 2^i-ésimo ancestral mais próximo do vértice. Desta forma, se um nó possui no máximo $O(n)$ ancestrais, sua array *par* terá tamanho $O(\log n)$. Uma consulta por um ancestral de nível k pode ser decomposta em no máximo $\lceil \log_2 k \rceil$ consultas cujos saltos são potências de dois. Logo, uma consulta pode ser feita em $O(\log n)$.

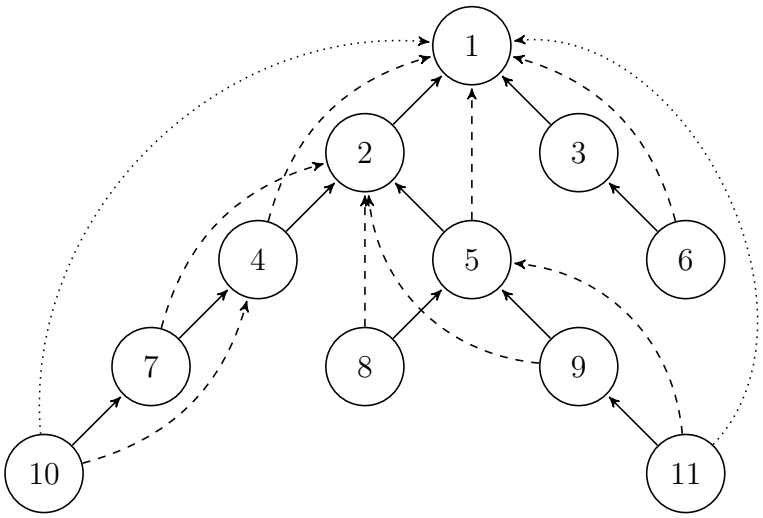

FIGURA A.3 – Representação de uma árvore por nós e ponteiro para todos os nós ancestrais binários. Árvore $O(n \log n)$ memória e $O(\log n)$ por consulta. Linahs grossas são ponteiros de salto 1, traços 2 e pontos 4.

Anexo A - Exemplo de um Primeiro Anexo

A.1 Uma Seção do Primeiro Anexo

Algum texto na primeira seção do primeiro anexo.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N	4. N DE PÁGINAS
TC	25 de março de 2015	DCTA/ITA/DM-018/2015	41

5. TÍTULO E SUBTÍTULO:

Estudo sobre Estruturas de Dados Persistentes

6. AUTOR(ES):

Lucas França de Oliveira

7. INSTITUIÇÃO(ÕES)/ÓRGAO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Cupim; Cimento; Estruturas

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Cupim; Dilema; Construção

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Mestrado. Programa de Pós-Graduação em Engenharia Aeronáutica e Mecânica. Área de Sistemas Aeroespaciais e Mecatrônica. Orientador: Prof. Dr. Adalberto Santos Dupont. Coorientadora: Profª. Drª. Doralice Serra. Defesa em 05/03/2015. Publicada em 25/03/2015.

11. RESUMO:

Este trabalho tem por objetivo a apresentação do tema de estruturas de dados persistentes. Para isso, apresentar-se-ão os conceitos respectivos: uma estrutura de dados, em um contexto de operações de atualização e de consulta é dita efêmera se ela guarda apenas sua última versão no tempo. É dita parcialmente persistente se permite consultas sobre qualquer versão anterior. É dita totalmente persistente se permite consultas e atualizações sobre qualquer versão, caso no qual a linha do tempo se ramifica e se torna uma árvore. É dita confluentemente persistente se é totalmente persistente e permite a junção de duas ramificações da linha do tempo métodos genéricos de transformação de estruturas efêmeras em persistentes. Apresentar-se-ão, também, implementações e testes das estruturas comuns apresentadas na disciplina de CES-11 em suas formas totalmente persistentes: pilha $O(1)$, fila $O(logn)$, deque $O(logn)$, fila de prioridades $O(logn)$, árvore de segmentos $O(logn)$ e árvore de busca binária $O(logn)$. Incluir-se-á, também, uma estrutura extra: array confluentemente persistente com consultas do tipo RSQ e RMQ, inserção e junção em $O(logn)$. As implementações serão feitas em C++ e visam a aplicação fácil em competições de programação.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**