UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CÂMPUS DE SÃO JOSÉ DO RIO PRETO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANDRÉ FURLAN
HIAGO MATHEUS BRAJATO

ÁRVORES ESPECÍFICAS: TRIE E PATRÍCIA

Propósitos e Propriedades
Análises e Complexidades

São José do Rio Preto – São Paulo
2019

RESUMO

Árvores podem ser definidas como estruturas de dados que caracterizam uma relação hierárquica entre os dados que as compõem. Esse tipo de estrutura é bastante útil para solucionar problemas que vão desde estruturas de armazenamento em banco de dados até representações de espaços de busca. As árvores descritas neste texto têm uma particularidade própria: Contrariamente de outras árvores, estas podem ser usadas sem maiores adaptações para armazenar tanto dados numéricos quanto alfanuméricos. São elas: As Tries e Patricia trees.

Palavras-chave: Árvore; Trie; Patricia; Alfanuméricos.

ABSTRACT

Trees can be defined as data structures that characterize a hierarchical relationship between the data that compose them. This type of structure is very useful for solving problems ranging from database storage structures to representations of search spaces. The trees described in this text have a peculiarity of their own: Unlike other trees, these trees can be used without further adaptations to store both numeric and alphanumeric data. They are: The Trie and Patricia trees.

Keywords: Tree; Trie; Patricia; Alphanumeric.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 ÁRVORES TRIE

Uma Árvore Trie (termo definido em 1960 pelo professor Edward Fredkin a partir da abstração da palavra “retrieval”) pode ser descrita como uma estrutura de dados utilizada para armazenar de forma hierárquica cadeias de caracteres e suportar uma rápida procura de registros, sendo sua principal aplicação a procura por padrões e prefixos.

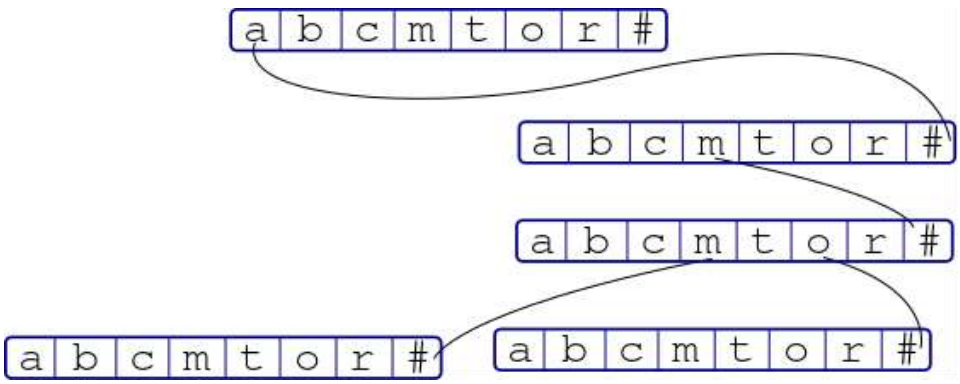
O grande diferencial das Árvores Trie em relação as, por exemplo, árvores de busca binárias (ABB), é que o valor a ser armazenado (daqui em diante chamaremos esse valor de chave) não precisa ser totalmente guardado em alguma estrutura interna da árvore (nó), pois a própria estrutura da árvore Trie consegue representar senão toda, boa parte da chave.

A principal ideia por trás do uso de uma Trie, e, que a difere da ABB, reside no fato de que na Trie as chaves são tratadas como elementos divisíveis (chaves compostas), ao contrário do que acontece em uma ABB, na qual cada chave é tratada como um elemento indivisível. Em uma árvore Trie, cada nó possui como valor implícito um caractere de um alfabeto pré-definido, dessa forma, ao realizar a pesquisa de uma chave na estrutura, o caractere inicial da chave é comparado com o nó inicial, e, se o caractere for encontrado, os seguintes serão comparados com o restante dos nós da subárvore até que o valor da chave completa seja encontrado ou não dentro da estrutura. O caminho desde a raiz até um nó não-folha ou não-terminal, pode ser denominado prefixo de chave. Dessa forma, conclui-se que uma Trie pode ser utilizada para um tipo restrito de procura de padrões, denominado “procura de palavra”.

O esquema de armazenamento das Árvores Trie se dá pela criação de vetores organizados de forma hierárquica, como é de costume entre estruturas do tipo árvore. O primeiro nível da trie é composto tão e somente por um único vetor, no qual é colocado o primeiro caractere da chave que se pretende armazenar, caso a chave tenha mais de um caractere então será criado dinamicamente um outro vetor em um nível abaixo do primeiro, e, assim por diante até que todos os caracteres da chave sejam mapeados.

A Figura 1 abaixo ilustra como é a estrutura de uma árvore trie:

Figura 1 – Estrutura de uma Árvore Trie



Fonte: Drozdek Adam, Data Structures and algorithms in C++)

Tendo como base a Figura 1 acima, podemos analisar a quantidade de espaço necessária para uma Trie como o número de *links* requeridos. Dessa forma temos que a quantidade de links em uma Trie corresponde à aproximadamente N * R * w, onde:

- N: corresponde ao número de chaves presentes na Árvore;
- R: indica que cada chave presente na estrutura possui um nó associado à um valor, sendo que este nó também contém links para outros nós;
- w: tamanho médio da chave (número de caracteres).

Dentro da estrutura de uma Trie há características interessantes que merecem certa atenção: Uma delas reside no fato de os caracteres estarem implícitos dentro de cada nó, sendo definidos na verdade pelo índice do ponteiro de um nó (link), além disso, faz-se importante expor que essa estrutura trabalha, normalmente com a forma de implementação denominada **R-WAY**, a qual apresenta a seguinte estrutura de nós: Cada nó possui um vetor de ponteiros (cada ponteiro é um caractere do alfabeto de tamanho **R**, dessa forma temos vários caminhos a seguir a partir do caractere indicado pelo ponteiro: **WAY**) e um valor (indica se o nó é o caractere final de uma chave ou não). A Figura 2 exibe a implementação de um nó em uma Trie.

Figura 2 - Implementação de um Nó em Trie

```
public class TrieNode {
    private TrieNode[] arrSubTries;
    private boolean isLeaf = false;
}
```

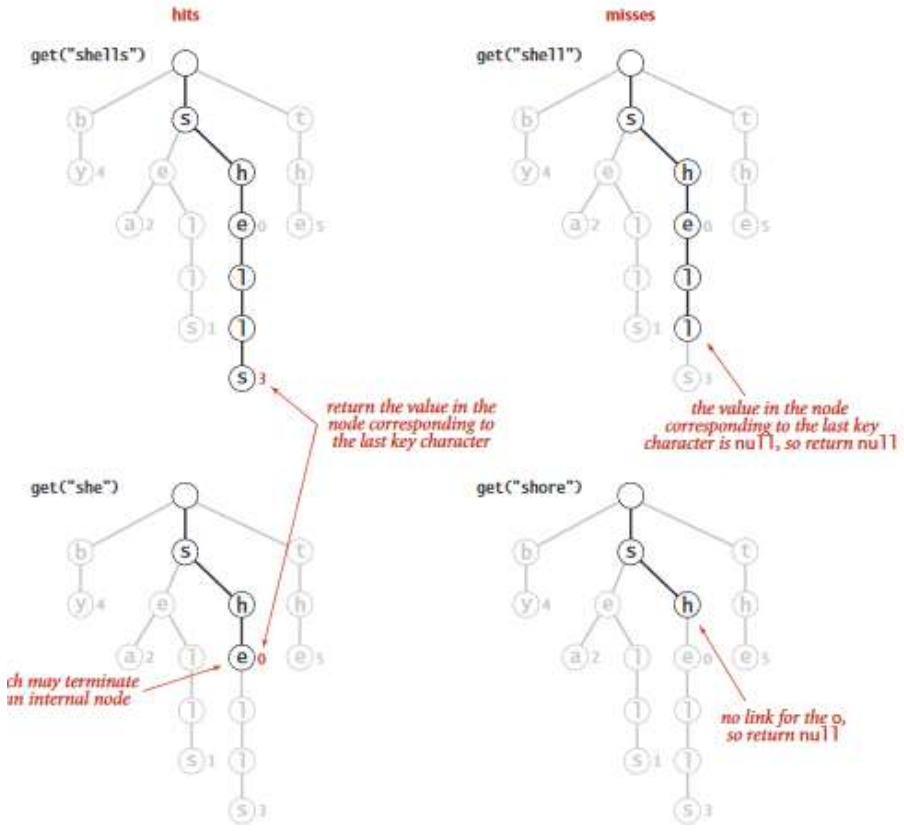
Fonte: Elaborado pelo autor

1.1 PESQUISA EM TRIES

O processo de busca de uma chave dentro de uma Trie é relativamente simples. O procedimento começa pelo nó raiz: O primeiro caractere da chave é localizado de acordo com seu valor, após isso, seguimos o link correspondente ao segundo caractere, e, assim por diante até que o último caractere da chave seja encontrado.

Na Figura 3 pode ser visto o processo de busca a partir de uma abstração da estrutura. Nos dois processos de busca à esquerda (“shells” e “she”) as chaves são encontradas e retornadas como resultado (hits), enquanto nas pesquisas à direita (“shell” e “shore”) as chaves não são encontradas e valores nulos são retornados (misses).

Figura 3 – Abstração do processo de pesquisa em uma Trie



Fonte: R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

É importante notar que, além dos links, o valor armazenado dentro dos nós desempenha um grande papel no comportamento de busca de chaves em uma Trie, impactando diretamente na decisão de pesquisa. Tal característica pode ser notada a

partir da Figura 3 acima nas pesquisas das chaves “shell” e “she”. No processo de busca de “shell”, o nó com o último caractere “l” possui valor nulo, indicando que o mesmo não é o fim da cadeia de caracteres de uma chave e, logo, “shell” não é uma palavra presente na estrutura até o momento. Já no procedimento de pesquisa da chave “she” o nó com o último caractere “e” possui um valor corresponde não-nulo, indicando que o mesmo é o último caractere de uma palavra presente no alfabeto, desta forma a chave “she” é retornada como resultado da pesquisa. Em algumas implementações também podem haver “flags” indicando se aquele nó é ou não o fim da cadeia (folha), a partir disso um valor nulo é retornado como resultado da pesquisa caso o nó não seja folha (qualquer nó interno pode ser marcado como folha na implementação de Tries).

A corretude do método de Pesquisa se faz relativamente simples. Caso a chave pesquisada pertença à estrutura a mesma será retornada, caso contrário um link nulo será o resultado, logo, a corretude pode ser demonstrada a partir da Indução nas subárvores.

1.2 INSERÇÃO EM TRIES

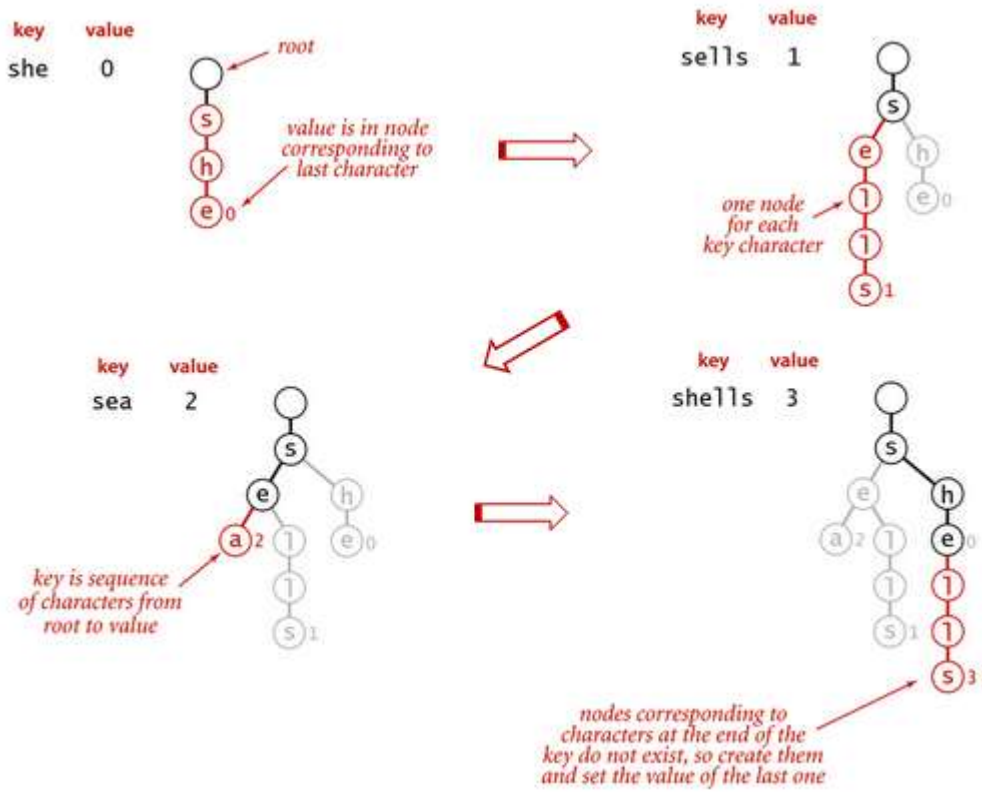
O processo de inserção de novas chaves em uma Árvore Trie é muito semelhante ao de Árvores de Busca Binária. É possível notar que em ambas ocorre um processo de pesquisa antes da inserção de um novo elemento, sendo que, no caso das Tries, utilizamos os caracteres como guia até que o último caractere da chave ou um link nulo seja encontrado. A partir do processo de pesquisa, duas possibilidades de inserção de chave são analisadas:

- A chave a ser inserida possui todos seus caracteres encontrados em um caminho da estrutura, dessa forma não se faz necessário inserir novos caracteres, e, o que é realizado é a atribuição de um valor não nulo ao nó que contém o último caractere da chave, ou a marcação daquele nó como uma folha dependendo da implementação.
- A chave a ser inserida não possui todos seus caracteres encontrados em um caminho, ou seja, um link nulo (ou a indicação de nó folha) é encontrado antes do último caractere da chave. Nessa situação é necessário que novos caracteres sejam inseridos na estrutura e, que ao último caractere da chave

que solicita a inserção seja atribuído um valor não-nulo ou seja marcado como folha, indicando que esse caractere é o último de uma palavra presente na estrutura.

Nas Figura 4 abaixo, pode ser visualizada a abstração do processo de inserção de quatro chaves dentro da Árvore Trie.

Figura 4 – Abstração da Inserção de uma nova chave em uma Trie



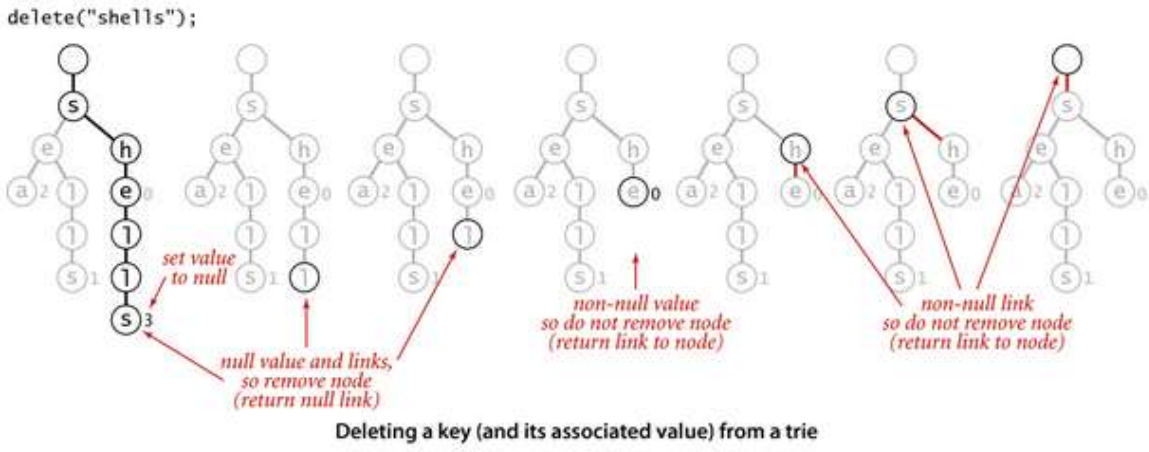
Fonte: R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

A corretude do método de Inserção, assim como na função anteriormente abordada, se faz trivial. Caso a String a ser inserida já esteja presente na estrutura, a prova da corretude já se encontra realizada, caso contrário, novos nós serão criados e o nó do último caractere será retornado, provando que a nova chave foi inserida na Trie (tal comportamento pode ser visualizado a partir dos códigos que se encontram no repositório citado no Apêndice A).

1.3 EXCLUSÃO EM TRIES

A exclusão de uma chave em uma Árvore Trie requer como primeiro passo uma pesquisa comum para a descoberta do nó que corresponde ao último caractere da chave. A partir disso, o nó descoberto tem seu valor trocado para null, e, então, é feita uma sub-pesquisa para verificar se os nós anteriores a este possuem valor nulo (o nó é removido) ou não (o nó é preservado). Na Figura 5 pode ser observada a abstração de um processo de remoção de chave em uma Trie.

Figura 5 – Abstração da exclusão de uma chave na Árvore Trie



Fonte: R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

A corretude do método de Exclusão é singelamente exemplificada a partir da abstração acima apresentada. Caso a String que requer exclusão não seja uma chave na Trie, não é necessária a remoção de nós (null é retornado); se a chave foi encontrada, o valor do nó do último caractere é alterado para null, e, se possível, o nó e seus antecessores são removidos caso seus respectivos valores sejam nulos (o último nó removido é retornado).

1.4 ANÁLISES E COMPLEXIDADES DAS OPERAÇÕES SOBRE TRIES

A análise da complexidade de tempo (T(n)) dos métodos de pesquisa, inserção e exclusão se fazem relativamente simples e análogos, sendo que suas respectivas

complexidades serão tratadas tomando como base o número de acessos à estrutura.[1] As análises abaixo realizadas podem ser melhor exemplificadas a partir da visualização dos métodos responsáveis (ver Apêndice A).

Pesquisa: o método “get()” realiza chamadas recursivas até a chave completa o um nó nulo seja retornado, dessa forma temos que T(n):

- Pior Caso: **O(n)**: “n” corresponde ao tamanho da chave (quantidade de caracteres que a chave possui. Exemplo: “car” possui tamanho 3);
- Caso Médio: **O(n):** a String pesquisada possui alguns caracteres dentro da Árvore, no entanto a chave completa não está presente. Como não podemos afirmar qual a quantidade de caracteres presentes ou não, temos que este caso pode ser representado por uma Progressão Aritmética (P.A.), resultando em ((n+1)/2) pela soma de seus termos;
- Melhor Caso: **O(1)**: ocorre quando o primeiro caractere da String não é encontrado na estrutura.

Inserção: o método “add()” realiza chamadas recursivas até que o nó do último caractere seja inserido e retornado (o método de pesquisa não é contabilizado):

- Pior Caso: **O(n)**: chave a ser inserida não possui nenhum prefixo presente na Árvore, logo todos seus caracteres necessitam ser inseridos na estrutura. A complexidade equivale ao tamanho da chave;
- Caso Médio: **O(n):** a chave a ser inserida possui prefixo com certo número de caracteres dentro da estrutura, no entanto o restante da chave necessita ser inserido. Assim como no método de pesquisa temos que ((n+1)/2);
- Melhor Caso: **O(1)**: a chave a ser inserida já se encontra presente na Árvore. Dessa forma é necessário apenas que o nó do último caractere seja marcado como folha (ou que um valor não nulo seja atribuído).

Exclusão: o método “delete()” realiza chamadas recursivas até que o último nó removido seja retornado (o método de pesquisa não é contabilizado).

- Pior Caso: **O(n)**: a chave a ser excluída terá todos seus caracteres removidos, pois todos nós do caminho percorrido possuem valores nulos (não são folhas).

[1] As complexidades serão analisadas e demonstradas a partir da Notação O (Big Oh!).

Dessa forma necessitamos acessar os nós correspondentes a cada caractere (n corresponde ao tamanho da chave);

- Caso Médio: **O(n):** a chave a ser excluída possui ao menos um nó interno com valor não nulo (existem nós folhas no caminho), assim apenas os nós posteriores ao último nó folha são removidos. Dessa forma temos que a probabilidade da quantidade de caracteres que devem ser removidos é a mesma para qualquer nó do caminho, resultando em uma P.A. equivalente a ((n+1)/2);
- Melhor Caso: **O(0)**: a chave a ser excluída não está presente na estrutura, não é necessário que nenhum nó seja removido.

2 ÁRVORES PATRICIA

Patricia é um acrônimo para Practical Algorithm To Retrieve Information Coded In Alphanumeric (definida por Donald Morrison em 1968), sendo a mesma definida como uma Trie Compactada Binária na qual os nós de ramificação (branch nodes) e de elemento (element nodes) foram fundidos em um único nó.[2]

Em uma Trie, para representar cada nó é necessário que seja criada uma lista de ponteiros do tamanho do alfabeto atual, o que pode resultar em um grande desperdício de memória, a partir disto, a Patricia realiza a otimização da complexidade de espaço eliminando espaços extras não utilizados.

Em termos de organização dos dados, uma árvore Patrícia se assemelha muito com uma árvore binária, visto que ambas admitem apenas dois filhos por nó, no entanto uma diferença considerável deve ser notada: Nas Árvores Patrícia não se compara chaves inteiras, mas sim apenas um bit de sua representação binária (as chaves devem ser previamente convertidas para um número binário). Tal comparação é realizada a partir de um campo denominado “bit diferenciador” ou “bit de comparação”, o qual é responsável por armazenar a posição do bit em questão (a posição é armazenada como um número inteiro).

[2] Devido a limitação de espaço as implementações devem ser consultadas no repositório de código apresentado no Apêndice A.

2.1 CRIAÇÃO DO “BIT DIFERENCIADOR” OU “BIT DE COMPARAÇÃO”

Para se determinar qual bit deve ser verificado é necessário converter para binário o valor do nó pai e o valor do nó que se pretende inserir, é importante que essas duas cadeias binárias tenham o mesmo tamanho, caso tenham tamanhos diferentes deve-se completar com zeros à esquerda. Após isso é feita uma comparação bit a bit das cadeias até que se encontre a primeira posição em que os valores se diferenciem, ou seja, quando o bit de uma for 1 e o bit de outra for 0, à essa posição dá-se o nome de “**bit diferenciador**” ou “**bit de comparação**”

Uma característica importante da árvore Patrícia é que as posições dos bits diferenciadores dos níveis mais profundos da árvore são necessariamente maiores do que as posições dos bits diferenciadores dos nós mais próximos à raiz. Tomando isso como base, será possível determinar quando um nó é folha ou não, pois diferentemente da árvore binária convencional, as folhas da Patricia não apontam para nós nulos.

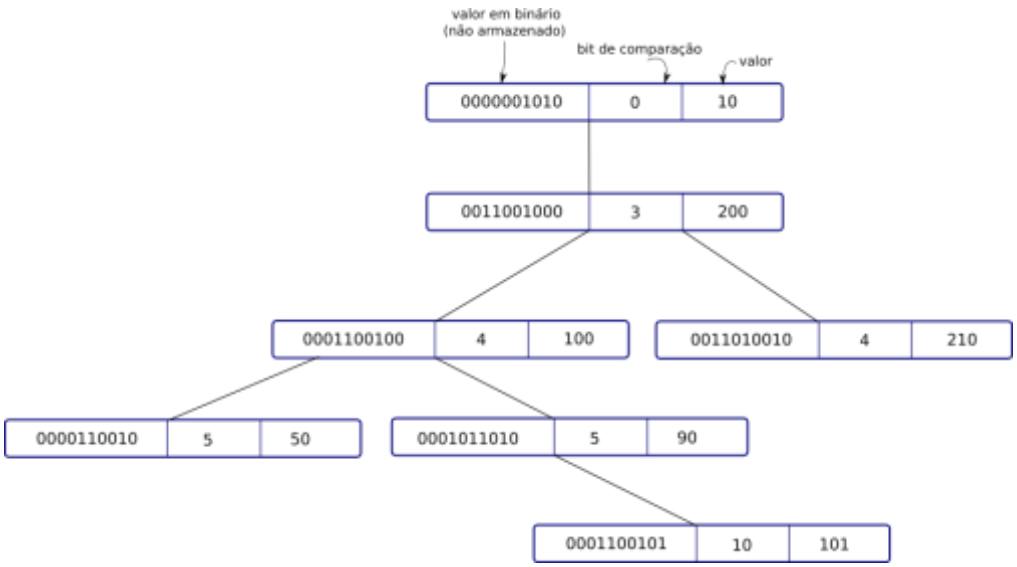
2.2 FORMATO DOS DADOS

As Árvores Patricia podem ser estruturadas com dois possíveis formatos de valores de chaves dependendo da implementação abordada: Formato binário ou formato original. O armazenamento na forma original facilita a leitura e comparação posteriores, enquanto o armazenamento no formato binário aumenta a velocidade da árvore, visto que não é preciso converter o valor de cada nó para binário a cada nova comparação. Como o próprio nome da árvore permite concluir, uma árvore Patrícia pode armazenar qualquer informação que possa ser codificada em caracteres ou números.

Ao contrário da árvore binária comum, o nó raíz da árvore Patrícia contém apenas um filho, sendo que esse nó raiz possui o valor do bit de comparação igual a zero, visto que não é possível determinar uma diferença com o nó anterior pois o mesmo não existe.

A Figura 6 exemplifica a estrutura de uma árvore Patrícia numérica:

Figura 6 – Árvore Patricia Numérica



Fonte: Elaborado pelo autor

Uma das grandes vantagens da árvore Patrícia reside no fato de que sua complexidade de espaço se faz mais otimizada que às Tries, sendo que a quantidade de nós presentes na estrutura é exatamente igual a quantidade de chaves armazenadas na mesma, ou seja, para N chaves armazenadas ocupa-se N nós. A estrutura abordada suporta, assim como as Tries, todas as operações básicas.

A Figura 7 demonstra como o nó de uma Patricia é implementado, é interessante notar que sua estrutura de dados se assemelha muito ao formato do nó de uma Árvore Binária.

Figura 7 – Nó da Árvore Patricia

```
class PatriciaNode {
    int bitNumber;
    int data;
    PatriciaNode leftChild, rightChild;
}
```

Fonte: Elaborado pelo autor

Na estrutura acima podemos visualizar uma variável chamada **bitNumber**, a qual é responsável por armazenar o bit de comparação, enquanto a variável denominada **data** cuida do armazenamento do valor em si (neste caso **data** é do tipo inteiro, no entanto qualquer tipo poderia ser atribuído). Além disso dois ponteiros **leftChild** e **rightChild** são representados, sendo os mesmos responsáveis por

apontar respectivamente para o filho da esquerda e da direita.

2.3 PESQUISA EM PATRICIAS

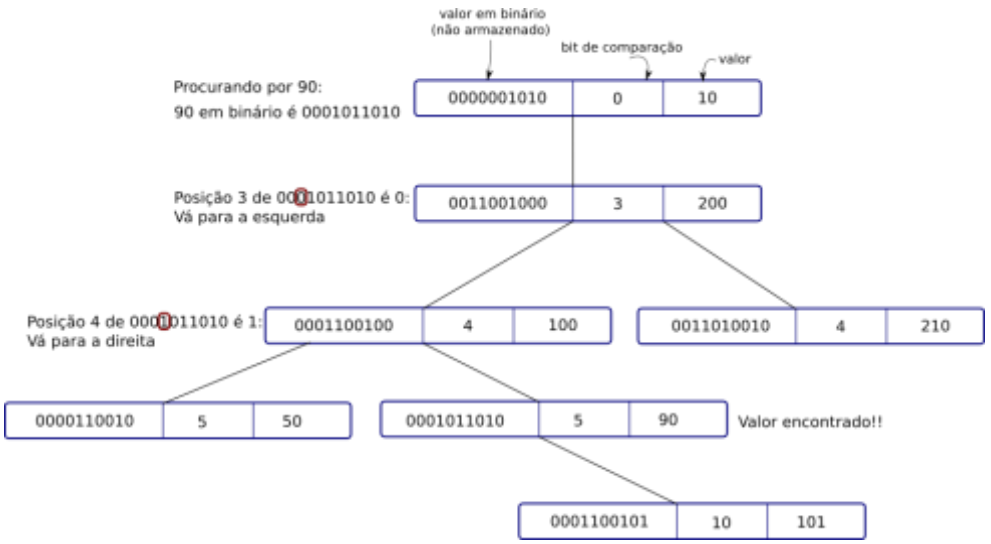
A pesquisa dentro das Árvores Patricia se faz, novamente, análoga à estrutura de uma árvore binária, sendo que a grande diferença se dá pela forma com que os valores das chaves são comparados. Na estrutura abordada os bits de suas respectivas representações binárias são comparados e não os valores de cada nó.

É importante ressaltar que nesta árvore os nós folhas apontam para si mesmos e/ou para seu respectivos avós ou pais, criando assim uma estrutura circular. Sabendo disso, a pesquisa na árvore deve continuar até que sua base seja alcançada, ou seja, quando a posição do bit de comparação do próximo nó é menor do que o bit de comparação do nó anterior.

Enquanto a condição acima for falsa é necessário decidir quais ramos e sub ramos da árvore devem ser explorados, e, é neste momento que os bits de comparação mostram sua importância: O valor que está sendo pesquisado deve ser convertido para sua respectiva cadeia binária e, em cada nó, um desses bits deve ser comparado de acordo com a posição indicada pelo bit de comparação do nó atual. Caso a posição indicada pelo bit de comparação seja zero, deve-se pegar o ramo da esquerda, do contrário, deve-se pegar o ramo da direita até que finalmente o valor seja encontrado. Nesta abordagem, caso o valor procurado não exista dentro da árvore, uma folha da mesma será retornada, o que tem grande utilidade na inserção de um novo elemento.

Na Figura 8 pode ser visualizado como é feita pesquisa, a partir da comparação pelos bits diferenciadores.

Figura 8 – Pesquisa em uma Árvore Patricia



Fonte: Elaborado pelo autor

2.4 INSERÇÃO EM PATRICIAS

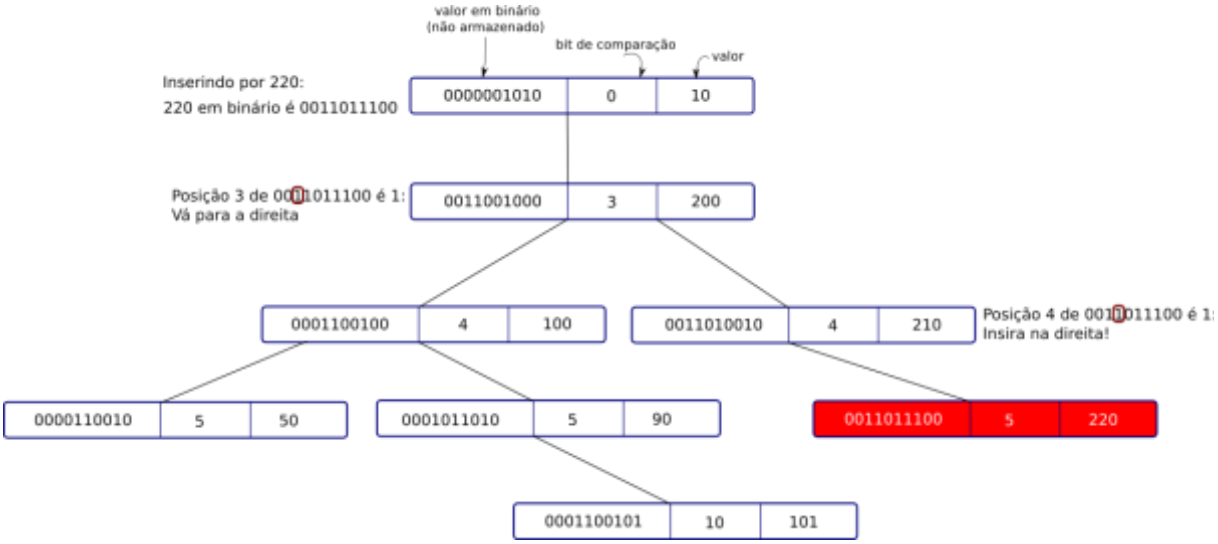
A inserção de uma chave dentro de uma árvore Patrícia requer primeiramente que seja verificado o primeiro nó, caso o mesmo seja nulo, um novo nó, cujo **bitNumber** = 0 e **data** = valor fornecido, é criado. Em seguida o valor que se quer adicionar é pesquisado dentro da árvore, caso esse valor já exista, a inserção é cancelada, do contrário, o algoritmo continua sua execução.

Como visto anteriormente, o algoritmo de pesquisa na árvore Patrícia sempre retorna um nó, o qual corresponde a uma das folhas da árvore, e que será o pai do novo nó a ser inserido. O bit de comparação do novo nó é definido usando as instruções anteriormente fornecidas, feito isso, na cadeia binária do valor a ser inserido (valor previamente convertido), verifica-se a posição definida pelo bit de comparação, caso o bit tenha valor zero armazena-se o novo nó à esquerda, caso contrário à direita.

É importante ressaltar que neste tipo de árvore um nó nunca aponta para uma posição nula, quando uma folha é inserida a mesma passa a apontar para si mesma e/ou para seu respectivo nó avó/pai.

A Figura 9 abaixo demonstra o processo de inserção em uma Patricia.

Figura 9 – Inserção em uma Árvore Patricia



Fonte: Elaborado pelo autor

2.5 EXCLUSÃO EM PATRICIAS

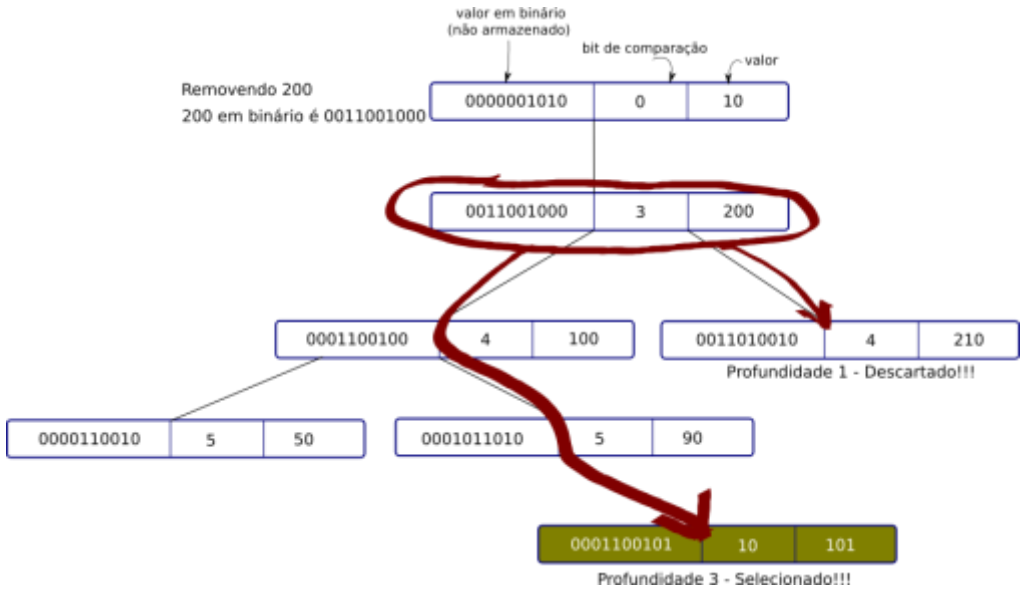
De todas as operações vistas até o momento, esta se faz a mais complexa. A remoção de uma folha é trivial: basta determinar quais são os nós que apontam para essa folha usando uma variação do algoritmo de pesquisa e fazer com que esses nós apontem para si mesmos ou para seus respectivos avós para, em seguida, liberar a memória ocupada pelo nó que se deseja remover.

No caso da remoção de um nó que não seja uma folha, o algoritmo aumenta significativamente em termos de complexidade. Neste caso é necessário definir qual é o antecessor do nó que se deseja remover, além disso é preciso também determinar quais são os respectivos filhos deste nó. Feito isso deve-se determinar qual será o nó que substituirá o atual (se houver), para isso, usaremos o mesmo algoritmo que é usado para tal procedimento em uma árvore binária: A partir do nó que se deseja remover, deve-se selecionar filho a esquerda e adicionar 1 a um contador, em seguida, deve-se percorrer todos os nós da direita até que se chegue a um nó folha, não esquecendo que a cada nó visitado o contador deve ser novamente incrementado em 1. Também a partir do nó que se deseja remover, se deve selecionar o filho a direita e adicionar 1 a um outro contador, em seguida há que se percorrer todos os nós da esquerda até que se chegue a uma folha, não esquecendo que a cada nó visitado o contador deve ser aumentado em 1.

A esses contadores daremos o nome de profundidade, caso a primeira profundidade seja maior, o nó que substituirá o nó excluído será o nó resultante da primeira verificação, caso contrário, será o nó da segunda verificação. Feito isso é necessário a modificação dos valores do bit de comparação para que o mesmo corresponda a sua respectiva camada.

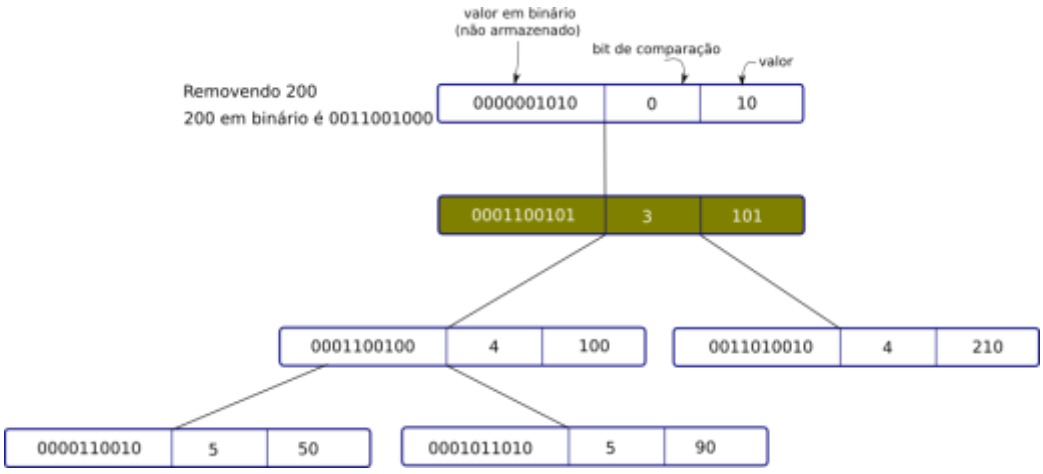
As Figuras 10 e 11 abaixo demonstram a abstração do processo de exclusão em uma Patricia.

Figura 10 – Localizando o nó substituído em uma Árvore Patricia



Fonte: Elaborado pelo autor

Figura 11 – Substituindo o nó a ser excluído



Fonte: Elaborado pelo autor

2.6 ANÁLISES E COMPLEXIDADES DAS OPERAÇÕES SOBRE PATRICIAS

Visto que a organização dos dados numa árvore Patricia é quase idêntica às árvores binárias, sua complexidade de tempo (T(n)) para os métodos de pesquisa, inserção e exclusão se faz extremamente análoga às ABB's, a partir do princípio de que sua estrutura não esteja degenerada, ou seja, equivalha à uma AVL:

- Pior Caso: **O(log n)**: Onde "n" corresponde a quantidade de elementos na estrutura;
- Caso Médio: **O(log n)**: Onde "n" é quantidade de elementos na estrutura;
- Melhor Caso: **O(1)**: Ocorre quando o primeiro nó é o nó que se deseja encontrar / excluir.

REFERÊNCIAS

Árvores Trie e Patricia. Disponível em <https://profschreiner.files.wordpress.com/2015/01/arvoredigital.pdf>. Acesso em 10 de Abril de 2019.

Busca Digital (Trie e Árvore Patrícia). Disponível em: <http://www.ufjf.br/jairo_souza/files/2009/12/6-Strings-Pesquisa-Digital.pdf>. Acesso em 28 de Abril de 2019.

Compressed tries (Patricia tries). Disponivel em:<http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie02.html>. Acesso em 5 de maio de 2019.

Compressing Radix Trees Without (Too Many) Tears. Disponível em: <https://medium.com/basecs/compressing-radix-trees-without-too-many-tears-a2e658adb9a0>. Acesso em 13 de maio de 2019.

Da Silva, Osmar .Q., **Estrutura de Dados e Algoritmos Usando C - Fundamentos e Aplicações**. Rio de Janeiro, CIÊNCIA MODERNA, 2007.

Dinesh P. Mehta, Sartaj Sahni. **Handbook of data structures and applications**: CHAPMAN & HALL/CRC, 2005.

Drozdek Adam, **Data Structures and algorithms in C++,** Boston, Cengage Learning, 2005.

Fundamental Data Structures. Disponível em: <http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf>. Acesso em 29 abril 2019.

Goodrich Michael and Tamassia Roberto, Trad. Copstein Bernardo, Pompermeier B. Leandro. **Estrutura de dados e algoritmos em Java**: Porto Alegre, BOOKMAN, 2007.

Java Algorithms and Clients. Disponível em: <https://algs4.cs.princeton.edu/code/>. Acesso em 13 de maio de 2019.

llvm-project - Revision 360608. Disponível em: <https://llvm.org/svn/llvm-project/test-suite/trunk/MultiSource/Benchmarks/MiBench/network-patricia/>. Acesso em 1 maio 2019.

Pesquisa digital. Disponível em <https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsearch.pdf>. Acesso em 11 de abril de 2019.

Radix Tree Reference. Disponível em: <http://www.voidcn.com/article/p-zrnpwmrn-tn.html>. Acesso em 12 maio 2019.

Sedgewick Robert, **Algorithms in C Third Edition**, Boston, Princeton University, 1946.

Sedgewick Robert, Wayne Kevin, **Algorithms fourth edition**: Boston, Princeton University, 2011.

Tenenbaum Aaron , Langsam Yedidyah, Augenstein Moshe J., Trad. Souza Tereza C. P. **Estruturas de Dados Usando C**. São Paulo, MAKRON Books, 1995.

Tries (árvores digitais). Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html>. Acesso em 29 abril 2019

Tries. Disponível em:<https://algs4.cs.princeton.edu/lectures/52Tries.pdf>. Acesso em 10 de maio de 2019.

Ziviani Nivio, **Projetos de Algoritmos Com Implementações em Pascal e C 4 ed.** São Paulo, Thomson Pioneira, 2007.

What is the difference between radix trees and Patricia tries?. Disponível em:< https://cs.stackexchange.com/questions/63048/what-is-the-difference-between-radix-trees-and-patricia-tries>.Acesso em 3 de maio de 2019.

APÊNDICE A – Códigos e Licenças

Para acessar os códigos fontes com as implementações correspondentes acesse:

https://github.com/ensismoebius/algoritmos

Os códigos fontes estão sob a licença **GPL versão 3**.