CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

João Pedro Pereira de Assis Castro Santos
Juliana Silva Cruz Sartori

REMOÇÃO DA SBB, BUSCA EM TRIE E PATRICIA

Timóteo

2022

1 Introdução

A árvore SBB já foi apresentada no documento anterior, por conseguinte, neste relatório irá ser destrinchado com relação às árvores Tie e Patrícia. Contudo, de forma mais geral a remoção na SBB, quando um nó folha é referenciado por um apontador vertical, ele é retirado da subárvore à esquerda ou a direita, tornando-a menor na altura após a retirada, isso acontece quando o nó a ser retirado possui dois descendentes, e assim sucessivamente.

Em primeira análise, com relação a árvore Trie, é uma estrutura de dados, que armazena array associativos (cadeia de caracteres), esta tem que ser ordenada. Foi definida em 1960 por Edward Fredkin, seu nome originou do Retrieval(Relacionado à Recuperação de Informações). Além disso, elas são ótimas para suportar tarefas de pesquisas em textos de grandes dimensões, assim como fazer uma pesquisa em um dicionário. Para elaborar essa arvore, é necessário que cada nó contendo informações sobre um ou mais símbolos do alfabeto, assim, uma dada sequência de arestas pode formar qualquer palavra (chave) possível com base nesse alfabeto; não existe limite para o tamanho de uma sequência (e portanto para o tamanho de uma chave); as sequências têm comprimento variável. Destarte, como aplicações pode-se ter buscas, ou seja, localizar um dado que corresponde a chave informada, conter inserções, ou seja, fazer a introdução de uma informação, avançando os caracteres já existentes na árvores ou não, caso esta seja inicialmente nula e remoções, quando se faz uma busca e este item existe na árvore, logo é removido. Porém, ela possui uma desvantagem é a formação de caminhos de uma só direção para chaves com um grande número de bits em comum.

Entretanto, o algoritmo para construção da árvore Patrícia é baseado no método de pesquisa digital, mas sem apresentar o inconveniente citado para o caso das Tries, sendo uma representação compacta de uma trie onde os nós que teriam apenas um filho são agrupados nos seus antecessores. A árvore PATRICIA foi apresentada pela primeira vez em 1968 por Donald R. Morrison no artigo Practical Algorithm To Retrieve Information Codec In Alphanumeric, publicado no Journal of ACM (sigla para Association for Computing Machinery). A motivação inicial de Morrison foi otimizar a busca de arquivos em bibliotecas.

Em um segundo momento, é de sábia compreensão que esta estrutura de dados, também realiza operações como busca, inserção e remoção de dados. A busca é similar a da Trie, com a diferença de que ao chegar em um nó, é comparado apenas um caractere, contra a comparação de substrings inteiras que acontece na Trie. Contudo, com relação a inserção, também é similar a pesquisar por essa string até o ponto onde a busca é encerrada, pois a string não é encontrada na árvore. Se a busca é encerrada em uma aresta, um novo nó é criado nessa aresta. Esse nó armazena a posição do caractere que distingue a chave destino daquela aresta e a chave que se deseja inserir, e tem como filhos o nó que estava na extremidade seguinte da aresta e um novo nó com a parte restante da nova chave. Se a busca for encerrada em um nó, então um nó filho é criado e o restante da nova chave é usado como rótulo para a resta entre os dois. Já a remoção, é o oposto da inserção, primeiro faz a busca, e se este existir na árvore é removido .

2 Desenvolvimento

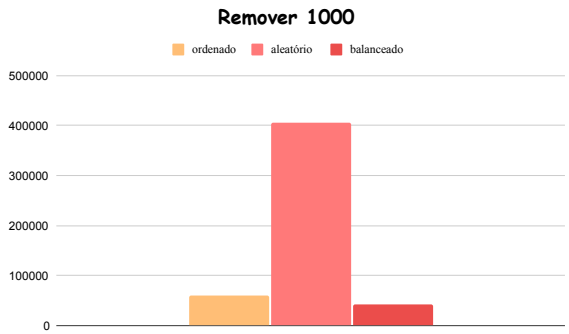
No experimento feito para este relatório, foi realizado a análise da remoção na árvore SBB e a análise de busca nas árvores Patricia e Trie, feitas com os algoritmos abaixo:

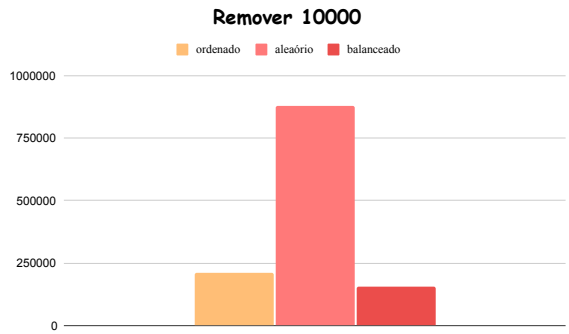
Código da remoção da árvore SBB:

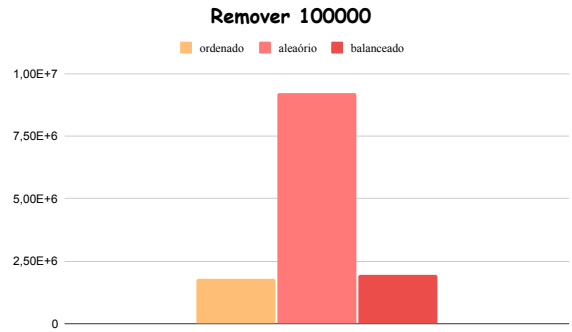
```
public boolean delete(int data) {
      boolean testdelete = false;
   return deleteNodeHelper(this.root, data);
  }
  private boolean deleteNodeHelper(Node node, int key) {
    Node z = TNULL;
    Node x, y;
    while (node != TNULL) {
      if (node.data == key) {
        z = node;
      }

      if (node.data <= key) {
        node = node.right;
      } else {
        node = node.left;
      }
    }

    if (z == TNULL) {
      return false;
    }

    y = z;
    int yOriginalColor = y.color;
    if (z.left == TNULL) {
      x = z.right;
      rbTransplant(z, z.right);
      testdelete = true;
    } else if (z.right == TNULL) {
      x = z.left;
      rbTransplant(z, z.left);
      testdelete = true;
    } else {
      y = minimum(z.right);
      yOriginalColor = y.color;
      x = y.right;
      if (y.parent == z) {
        x.parent = y;
        testdelete=true;
```

```
      } else {
        rbTransplant(y, y.right);
        testdelete=true;
        y.right = z.right;
        y.right.parent = y;
      }

      rbTransplant(z, y);
      y.left = z.left;
      y.left.parent = y;
      y.color = z.color;
    }
    if (yOriginalColor == 0) {
      fixDelete(x);
    }
    return testdelete;
}
```

Gráficos dos tempos de remoção na árvore SBB em nanossegundos:







Código da árvore Trie:

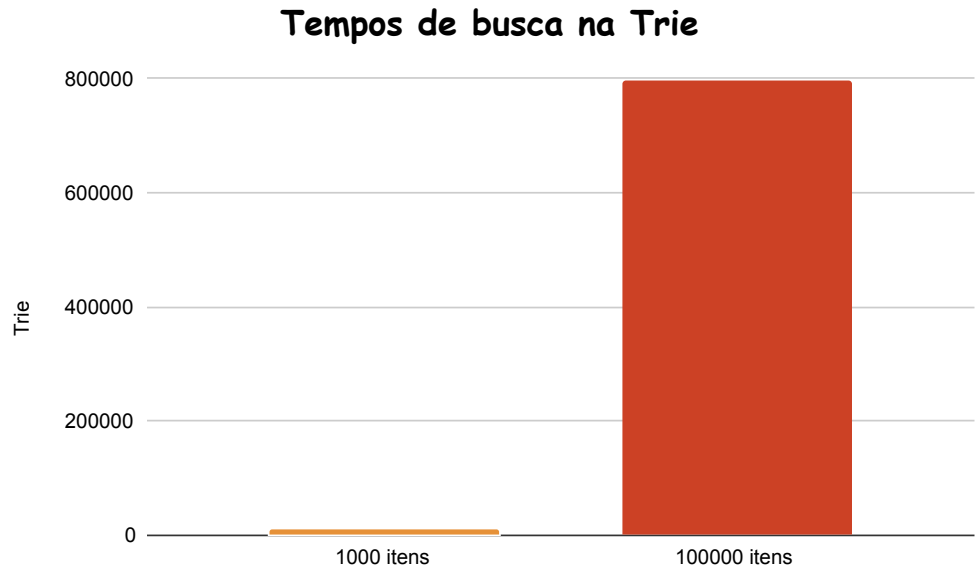
```
public class TrieTree {
    public static class Noh{

        //a  posio  do vetor simboliza o caractere
        public boolean finishWord;                                  //Indica se o
            caminho formou uma word
        public Noh vetNOfilhos[];                                   //Possiveis
```

```
        proximos caracteres

    //Construtor padro
    public Noh(){
        finishWord = false;
        vetNOfilhos = new Noh[26];                          //26 qtd de
            letras do alfabeto

        for(int i = 0; i < 26; i++){                        //Definindo
            um valor (Setando) para null
            vetNOfilhos[i] = null;
        }
    }
}

private Noh raiz;                                           //Noh raiz da
    arvore Trie
public int qtdNos;
public String vet[];

//construtor padrao
public TrieTree(int qtdNos){
    raiz = new Noh();
    this.qtdNos = qtdNos;
    vet = new String[(int)(qtdNos * 0.01)];
}

// Busca uma palavra na Trie
public boolean search( String word ){

    word = word.toLowerCase();                              //mudando a
        palavra para letras minusculas

    Noh noAux = this.raiz;

    for(int j = 0; j < word.length(); j++){

        if( noAux.vetNOfilhos[ word.charAt(j) - 'a' ] == null ){    //Nao existe
            noh correspondente
            return false;
        }
        noAux = noAux.vetNOfilhos[word.charAt(j) - 'a' ];
    }

    if( ( noAux != null ) && ( noAux.finishWord == true ) ){
        return true;                                        //foi
            encontrado o noh
    }
```

```
        return false;                                                    //senao
            encontarr eh falso
    }

    //Insere uma palavra na Trie
    public boolean insert( String word){

        word = word.toLowerCase();

        if( search(word) == true ) {                                      //Caso em
            que a palavra ja foi adicionada
            return false;
        }

        Noh noAux = this.raiz;

        for( int i = 0; i < word.length(); i++){

            if( noAux.vetNOfilhos[ word.charAt(i) - 'a'] == null ){
                noAux.vetNOfilhos[ word.charAt(i) - 'a'] = new Noh();     //add se o
                    noh nao existir
            }

            noAux = noAux.vetNOfilhos[ word.charAt(i) - 'a'];
        }

        noAux.finishWord = true;

        return true;
    }

    // Remover uma palavra na Trie
    public boolean delete( String word ){

        word = word.toLowerCase();

        if( search(word) == false ){                                    //A palavra
            nao esta na Trie
            return false;
        }

        Noh noAux = this.raiz;
        Noh noAuxTerminal = noAux;
        int indiceDepoisDoTerminal = word.charAt(0) - 'a';

        for( int j = 0; j < word.length(); j++){

            if( noAux.finishWord == true ){
```

```
                noAuxTerminal = noAux;
                indiceDepoisDoTerminal = word.charAt(j) - 'a';
            }

            noAux = noAux.vetNOfilhos[ word.charAt(j) - 'a' ];
        }

        noAuxTerminal.vetNOfilhos[indiceDepoisDoTerminal] = null;

        return true;
    }
}
```

Gráficos dos tempos de busca na árvore Trie em nanossegundos:



Código da árvore Patrícia:

```
public class PaTrieciaTree {
 private static abstract class PatNode {
    }

    private static class PatNodeInt extends PatNode {

        int index;
        PatNode left, right;
    }

    private static class PatEndNode extends PatNode {

        int item;
    }
```

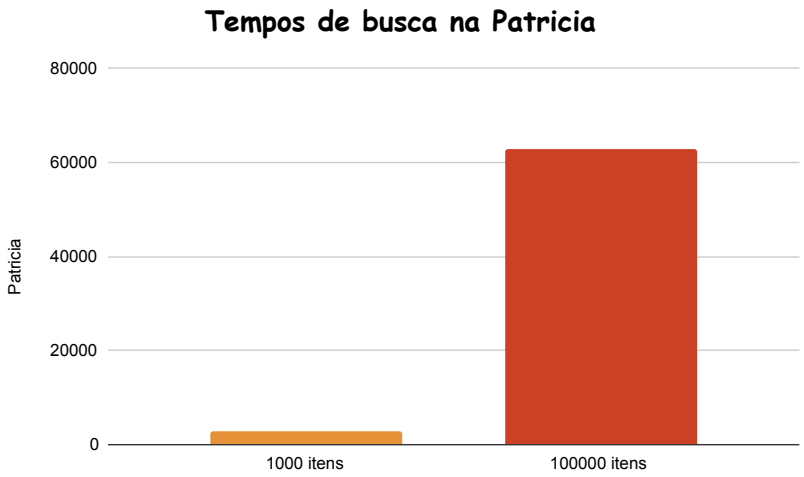
```
private PatNode root;
private int nbitsitem;
public int nodeAmount;

public void PatriciaTree(int nbitsitem, int nodeAmount) {
    this.root = null;
    this.nbitsitem = nbitsitem;
    this.nodeAmount = nodeAmount;

}

private int bit(int i, int k) {
    if (i == 0) {
        return 0;
    }
    int c = (int) k;
    for (int j = 1; j <= this.nbitsitem - i; j++) {
        c = c / 2;
    }
    return c % 2;
}

private boolean isEndNode(PatNode p) {
    Class class1 = p.getClass();
    return class1.getName().equals(PatEndNode.class.getName());
}

private PatNode createIntNode(int i, PatNode left, PatNode right) {
    PatNodeInt p = new PatNodeInt();
    p.index = i;
    p.left = left;
    p.right = right;
    return p;
}

private PatNode createEndNode(int k) {
    PatEndNode p = new PatEndNode();
    p.item = k;
    return p;
}

private boolean search(int k, PatNode t) {
    if (this.isEndNode(t)) {
        PatEndNode aux = (PatEndNode) t;
        if (aux.item == k) {
            return true;
        } else {
```

```
            return false;
        }
    } else {
        PatNodeInt aux = (PatNodeInt) t;
        if (this.bit(aux.index, k) == 0) {
            return search(k, aux.left);
        } else {
            return search(k, aux.right);
        }
    }
}

private PatNode insertBetween(int k, PatNode t, int i) {
    PatNodeInt aux = null;
    if (!this.isEndNode(t)) {
        aux = (PatNodeInt) t;
    }
    if (this.isEndNode(t) || (i < aux.index)) { // @{\it Cria um novo n\'o
        externo}@
        PatNode p = this.createEndNode(k);
        if (this.bit(i, k) == 1) {
            return this.createIntNode(i, t, p);
        } else {
            return this.createIntNode(i, p, t);
        }
    } else {
        if (this.bit(aux.index, k) == 1) {
            aux.right = this.insertBetween(k, aux.right, i);
        } else {
            aux.left = this.insertBetween(k, aux.left, i);
        }
        return aux;
    }
}
public void insert(int insertItem) {

    this.root = this.insert(insertItem, this.root);
}
private PatNode insert(int insertItem, PatNode currentNode) {
    if (currentNode == null) {
        return this.createEndNode(insertItem);
    } else {
        PatNode p = currentNode;
        while (!this.isEndNode(p)) {
            PatNodeInt aux = (PatNodeInt) p;
            if (this.bit(aux.index, insertItem) == 1) {
                p = aux.right;
            } else {
```

```
                p = aux.left;
            }
        }
        PatEndNode aux = (PatEndNode) p;
        int i = 1; // @{\it acha o primeiro bit diferente}@
        while ((i <= this.nbitsitem)
                && (this.bit(i, insertItem) == this.bit(i, aux.item))) {
            i++;
        }
        if (i > this.nbitsitem) {
            return currentNode;
        } else {
            return this.insertBetween(insertItem, currentNode, i);
        }
    }
}

private void central(PatNode parentNode, PatNode next, String msg) {
    if (next != null) {
        if (!this.isEndNode(next)) {
            PatNodeInt aux = (PatNodeInt) next;
            central(next, aux.left, "left");
            if (parentNode != null) {
                System.out.println("N pai: " + ((PatNodeInt) parentNode).index + "
                    " + msg + " Int: " + aux.index);
            } else {
                System.out.println("N pai: " + parentNode + " " + msg + " Int: " +
                    aux.index);
            }
            central(next, aux.right, "right");
        } else {
            PatEndNode aux = (PatEndNode) next;
            if (parentNode != null) {
                System.out.println("N pai: " + ((PatNodeInt) parentNode).index + "
                    " + msg + " Ext: " + aux.item);
            } else {
                System.out.println("N pai: " + parentNode + " " + msg + " Ext: " +
                    aux.item);
            }
        }
    }
}

public void print() {
    this.central(null, this.root, "root");
}

public boolean search(int searchItem) {
```

```
        return search(searchItem, this.root);
    }
}
```

Gráficos dos tempos de busca na árvore Patricia em nanossegundos:



Com base na análise dos gráficos de tempo gerados pela execução dos algoritmos com código de contagem de tempo, é possível observar comportamentos de cada estrutura de árvore.

3 Conclusão

Contudo, pode-se concluir que a análise gráfica indica que a árvore trie possui uma busca mais rápida se comparado a árvore binária e ela também requer menos espaço quando contém um grande número de cadeias curtas, porque as chaves não são armazenadas de forma explícita e os nós das chaves iniciais comuns são compartilhados, sendo muito útil para localizar uma palavra em um dicionário. Entretanto, a árvore Patricia possui caminhos mais curtos, contendo apenas referências das casas onde as palavras se diferenciam, sendo de grande ajuda para a organização. Comparando a árvore Trie com a Patrícia, a árvore trie tem desenvolvimento mais simples, mas tem o problema de quando existirem muitas palavras com várias casas diferentes, ter vários ramos de diferenciação, ficando quase igual em termos de tamanho à árvore trie. Em reação a remoção de SBB, o pior tempo esperado era o demonstrado por gráfico, ou seja, o tempo aleatório, pois no caso ordenado, não é formada uma lista encadeada igual nas outras árvores, já que a SBB se balanceia na inserção, ela forma uma árvore balanceada, assim como na inserção balanceada, o mesmo vale para o caso aleatório, assim sendo, a estrutura da busca para remoção é mais caótica, pois não se sabe qual número será selecionado por PRNG para ser removido, podendo demorar vários segundos para encontrar algum valor que existe na árvore, mas o que mais retarda esse processo, é a inserção completamente aleatória, resultando em operações demoradas para o rebalanceamento em cada inserção sucessiva, podendo fazer um valor máximo de rotações por um valor "ruim"(um valor muito distante de sua posição, que deve mover vários termos para se posicionar) ser o próximo inserido, sendo então o pior caso.

Referências