CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Samuel Oliveira Ferraz Porto

ESTRUTURAS DE DADOS: ÁRVORE

Timóteo

2023

Samuel Oliveira Ferraz Porto

ESTRUTURAS DE DADOS: ÁRVORE

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Leonardo Lacerda Alves
Coorientador: Beltrano de Silva e Tal

Timóteo

2023

.

.

.

Sumário

1 O que é árvore?

Uma árvore é uma estrutura de dados amplamente utilizada na ciência da computação e na programação para representar hierarquias e relações entre elementos. Ela é composta por nós (ou vértices) interconectados por arestas. Cada nó possui um valor (ou chave) associado e pode ter zero ou mais nós filhos, dependendo da organização hierárquica.(LADEIRA, 2021)

1.1 Implementação

```java
public void inserir(No node, int valor) {
  //verifica se a árvore já foi criada
   if (node != null) {
    //Verifica se o valor a ser inserido é menor que o nodo corrente da árvore, se sim vai para subár
    if (valor < node.valor) {
        //Se tiver elemento no nodo esquerdo continua a busca
        if (node.esquerda != null) {
            inserir(node.esquerda, valor);
        } else {
            //Se nodo esquerdo vazio insere o novo nodo aqui
            System.out.println("  Inserindo " + valor + " a esquerda de " + node.valor);
            node.esquerda = new No(valor);
        }
    //Verifica se o valor a ser inserido é maior que o nodo corrente da árvore, se sim vai para subár
    } else if (valor > node.valor) {
        //Se tiver elemento no nodo direito continua a busca
        if (node.direita != null) {
            inserir(node.direita, valor);
        } else {
            //Se nodo direito vazio insere o novo nodo aqui
            System.out.println("  Inserindo " + valor + " a direita de " + node.valor);
            node.direita = new No(valor);
        }
    }
  }
}
```

Fonte: (PREISS, 2000)

1.2 Exemplos

1.2.1 Método de inserção fora de ordem

```java
public class Arvore {
    NoArvore raiz;
    public Arvore() {
        raiz = null;
    }
    public void inserir(int valor) {
        raiz = inserirRec(raiz, valor);
    }
    private NoArvore inserirRec(NoArvore raiz, int valor) {
        if (raiz == null) {
            raiz = new NoArvore(valor);
            return raiz;
        }
        if (valor < raiz.valor) {
            raiz.esquerda = inserirRec( raiz:raiz.esquerda, valor);
        } else if (valor > raiz.valor) {
            raiz.direita = inserirRec( raiz:raiz.direita, valor);
        }
        return raiz;
    }
```

Baseado nos conceitos de (MAIDA, 2015)

1.2.2 Método de exclusão no meio

```
public void remover(int valor) {
    raiz = removerRec(raiz, valor);
}
private NoArvore removerRec(NoArvore raiz, int valor) {
    if (raiz == null) {
        return raiz;
    }
    if (valor < raiz.valor) {
        raiz.esquerda = removerRec( raiz:raiz.esquerda, valor);
    } else if (valor > raiz.valor) {
        raiz.direita = removerRec( raiz:raiz.direita, valor);
    } else {
        // Nó com um ou nenhum filho
        if (raiz.esquerda == null) {
            return raiz.direita;
        } else if (raiz.direita == null) {
            return raiz.esquerda;
        }
        raiz.valor = encontrarMenorValor( raiz:raiz.direita);
        raiz.direita = removerRec( raiz:raiz.direita,  valor:raiz.valor);
    }
    return raiz;
```

```
    private int encontrarMenorValor(NoArvore raiz) {
        int menorValor = raiz.valor;
        while (raiz.esquerda != null) {
            menorValor = raiz.esquerda.valor;
            raiz = raiz.esquerda;
        }
        return menorValor;
    }
}
```

Baseado nos conceitos de (MAIDA, 2015)

1.2.3 Método de impressão por iteração

```
public void percorrerPorIteracao() {
    if (raiz == null) {
        return;
    }

    Stack<NoArvore> pilha = new Stack<>();
    pilha.push( item:raiz);

    while (!pilha.isEmpty()) {
        NoArvore no = pilha.pop();
        System.out.print(no.valor + " ");

        if (no.direita != null) {
            pilha.push( item:no.direita);
        }

        if (no.esquerda != null) {
            pilha.push( item:no.esquerda);
        }
    }
```

Baseado nos conceitos de (MAIDA, 2015)

1.3 Explicação

O código implementa uma estrutura de dados de árvore de busca em Java. Há funções para inserção, remoção e percurso da árvore.
A classe NoArvore representa um nó da árvore de busca. Possui três atributos: valor (que simboliza o valor numérico do nó), esquerda (que simboliza o nó filho esquerdo) e direita (nó filho direito). O construtor NoArvore cria um novo nó com o valor especificado e inicializa os filhos como null.
A classe Arvore representa a árvore de busca. Possui um atributo chamado raiz que aponta para a raiz da árvore. O construtor Arvore inicializa a raiz como null.
O método inserirRec realiza a inserção de um novo valor na árvore de forma recursiva. Se a raiz passada for null, cria um novo nó com o valor fornecido e o retorna. Se o valor inserido é menor que o valor da raiz, a inserção é feita na subárvore esquerda. Caso contrário, é feita na subárvore direita.
Método removerRec realiza a remoção de um nó com um valor específico da árvore de forma recursiva. Se a raiz passada for null, retorna null. Se o valor a ser removido for menor que o valor da raiz, a remoção é feita na subárvore esquerda. Se for maior, a remoção é feita na subárvore direita. Se o valor a ser removido for igual ao valor da raiz, diferentes casos são tratados: se o nó não tiver filhos ou tiver apenas um filho, ele é removido; se tiver dois filhos, seu valor é substituído pelo menor valor da subárvore direita e o processo de remoção é aplicado a esse valor.
O método encontrarMenorValor, como o proprio nome diz, encontra e retorna o menor valor na subárvore especificada.
O método percorrerPorIteracao realiza um percurso em profundidade da árvore usando iteração com uma pilha. Começa empilhando o nó raiz e, em seguida, itera até que a pilha esteja vazia, desempilhando nós, imprimindo seus valores e empilhando seus filhos direito e esquerdo.

1.4 Pós e Contras

Prós:

O algoritmo é eficiente para busca: Uma árvore de busca é otimizada para operações de busca, inserção e remoção, tendo um tempo médio de busca de O(log n) quando está balanceada. Isso faz com que seja uma escolha eficiente para armazenar e recuperar dados ordenados.
É uma estrutura ordenada: A árvore de busca mantém os elementos em ordem, o que pode ser útil para operações que envolvem ordenação ou busca por intervalos.
Tem flexibilidade: A implementação da árvore de busca é flexível e permite a realização de várias operações, como inserção, remoção e travessia, podendo ser recursiva ou iterativa como é no exemplo.
Espaço eficiente: Comparada com outras estruturas de dados, como listas encadeadas, a árvore binária de busca pode ser mais eficiente em termos de espaço, especialmente quando bem balanceada.

Contras:

Se a árvore não for balanceada adequadamente, ela pode degenerar em uma lista encadeada, o que aumentaria o tempo de busca para O(n), onde n é o número de elementos na árvore.
Enquanto as operações de busca, inserção e remoção são eficientes em média, pode haver casos em que uma operação pode ser ineficiente, especialmente se a árvore estiver desbalanceada.
A implementação correta e eficiente de uma árvore de busca pode ser complexa, especialmente quando se trata de manter a árvore balanceada.
A árvore binária de busca pode consumir mais memória do que algumas outras estruturas de dados, especialmente se não for balanceada.

2 Árvore Rubro-Negra

Uma árvore rubro-negra é uma estrutura de dados que mantém o equilíbrio em uma árvore binária através de um conjunto de regras sobre cores de nós e a ordem das inserções e remoções. Isso resulta em um desempenho previsível e eficiente para várias operações de árvore binária.

São as seguintes regras:

[1] Cada nó é considerado vermelho ou preto.

[2] A raiz da árvore é sempre um nó preto.

[3] Qualquer nó nulo é preto.

[4] Se um nó for vermelho, seus filhos serão sempre pretos. Ou seja, um nó vermelho não pode ter qualquer filho vermelho.

[5] Qualquer caminho de um nó para um nó nulo deve conter o mesmo número de nós pretos

(OLIVEIRA, 2015)

2.1 Prós e Contras

Prós:

As árvores rubro-negras garantem que a altura da árvore seja proporcional ao logaritmo do número de nós, sendo positivo para operações de busca, inserção e remoção. Isso resulta em um desempenho médio de operações mais eficiente. Pois, ao realizar inserções ou remoções, as árvores rubro-negras realizam reorganizações que preservam as propriedades da árvore, mantendo-a balanceada.

Outra benefício é que se as operações de inserção e remoção são frequentes ou dinâmicas, as árvores rubro-negras podem ser mais eficazes do que árvores binárias de busca não balanceadas.

Contras:

A lógica para manter as propriedades de árvore rubro-negra durante as operações pode ser complexa, o que torna a implementação mais desafiadora em comparação com árvores binárias simples.

As árvores rubro-negras requerem um bit extra de armazenamento por nó para representar a cor do nó, o que pode aumentar o overhead de memória em comparação com estruturas de dados mais simples.

Embora as árvores rubro-negras tenham bom desempenho em média, em casos ex-

tremos específicos, podem ser menos eficientes do que outras estruturas balanceadas, como árvores AVL.

2.2 Situações mundo real

As árvores rubro-negras são benéficas em situações em que as operações de busca, inserção e remoção precisam ser eficientes e previsíveis. Como, por exemplo, em sistemas de gerenciamento de memória, em que as árvores rubro-negras podem ser usadas para rastrear blocos livres e alocados, permitindo um gerenciamento eficiente de alocação e liberação de memória.

2.3 Algoritmos semelhantes

Alguns algoritmos semelhantes são: Árvore AVL, Árvores Splay, Tabelas Hash e Skip Lists. (SONG, 2008)

2.4 Exemplos

2.4.1 Método de inserção

```
public int insereRN(Conteudo conteudo) {
    NoRN p;
    NoRN raiz = null;
    p = insereRNRec(raiz, conteudo);
    if (p != null) {
        raiz = p;
        raiz.cor = NoRN.PRETO;
        int nNos = 0;
        nNos++;
    }
    return p != null ? 1 : 0;
}
```

(OLIVEIRA, 2015)

2.4.2 Método de exclusão

```
void exclusaoComRotacaoDupla(No x) {
    No y = x.esquerda;
    No z = y.direita;

    rotacaoEsquerda( x: y);
    rotacaoDireita( y: x);

    y.cor = x.cor;
    x.cor = 1;
    z.cor = 0;
}
```

(OLIVEIRA, 2015)

2.4.3 Método de impressão por iteração

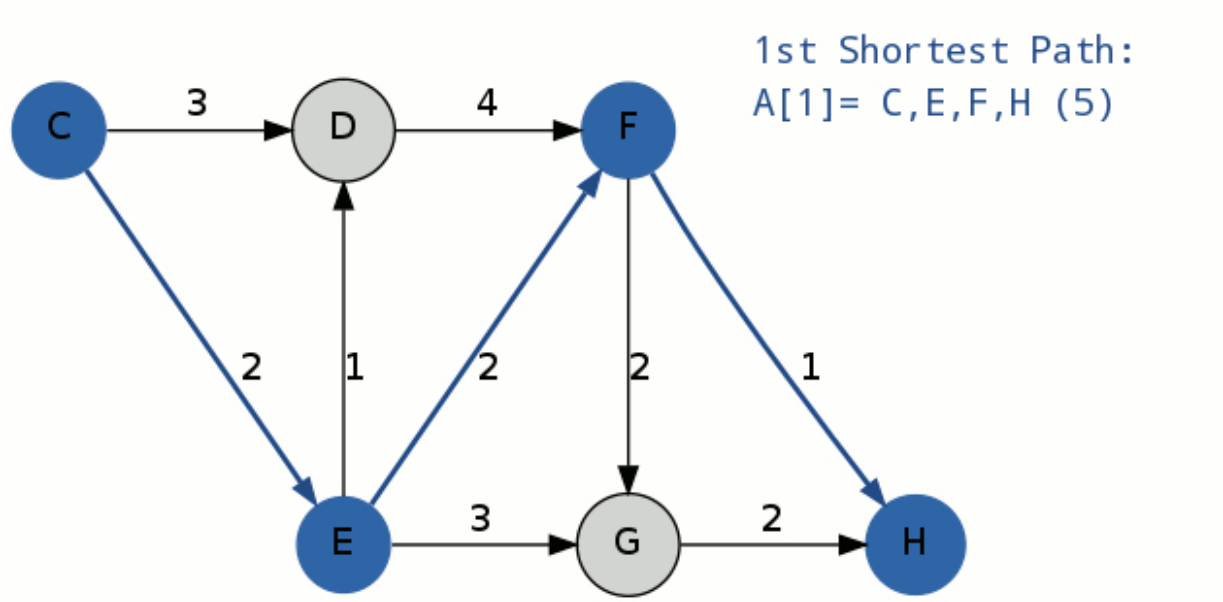
```
public void percorrerPorIteracao() {
    if (raiz == null) {
        return;
    }

    Stack<NoArvore> pilha = new Stack<>();
    pilha.push( item:raiz);

    while (!pilha.isEmpty()) {
        NoArvore no = pilha.pop();
        System.out.print(no.valor + " ");

        if (no.direita != null) {
            pilha.push( item:no.direita);
        }

        if (no.esquerda != null) {
            pilha.push( item:no.esquerda);
        }
    }
```

(OLIVEIRA, 2015)

3 Algoritmo Yen de 1971

3.1 Introdução

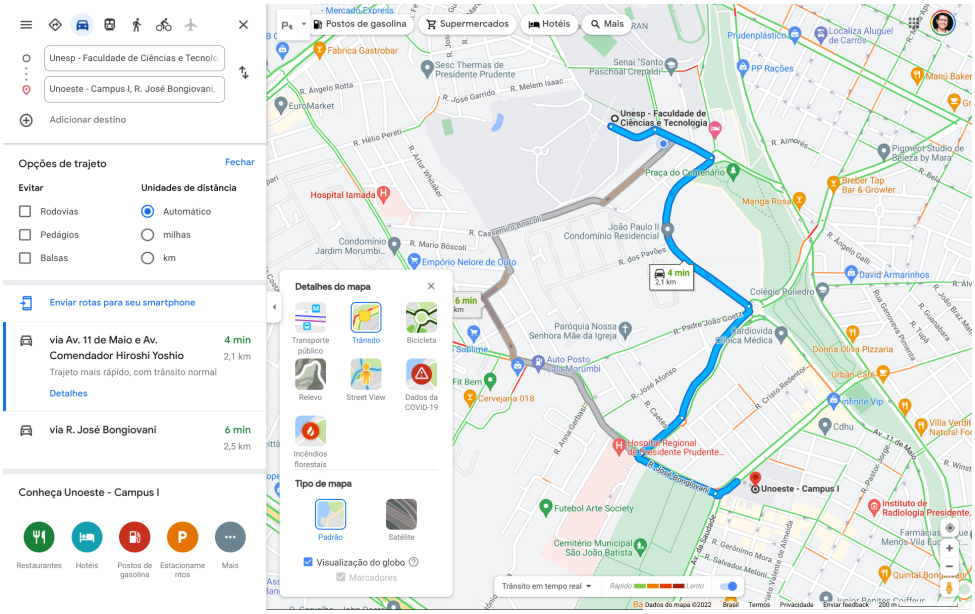
O algoritmo foi desenvolvido por Jin Y. Yen em 1971. Trata-se de um algoritmo para encontrar o caminho mais curto entre dois vértices em um grafo ponderado. É uma variação do algoritmo de Dijkstra e é usado para calcular o caminho mais curto de um vértice de origem para um vértice de destino em um grafo ponderado. (BENATTI, 1993)



Exemplo de grafo

3.2 Algoritmo de Yen no cotidiano

Esse algoritmo é utilizado em muitas situações do cotidiano, como nos sistemas de roteamento de trânsito para encontrar várias rotas alternativas entre dois pontos em uma rede de estradas. (BOAVENTURA, 2003)



Sistema de roteamento

3.3 Aplicação

```
public List<List<Aresta>> menorCaminho(No origem, No destino, int k) {
    PriorityQueue<List<Aresta>> filaCaminho = new PriorityQueue<>((c1, c2) -> {
        int peso1 = calcularPesoCaminho(c1);
        int peso2 = calcularPesoCaminho( c1: c2);
        return Integer.compare( x: peso1,  y: peso2);
    });
    while (true) {
        List<Aresta> caminhoMaisCurto = encontrarCaminhoMaisCurto(origem, destino);
        if (caminhoMaisCurto == null) {
            break;
        }

        filaCaminho.add( e: caminhoMaisCurto);
        removeAresta( caminho: caminhoMaisCurto);

        if (filaCaminho.size() >= k) {
            break;
        }
    }

    return new ArrayList<>( c: filaCaminho);
}
```

```
private List<Aresta> encontrarCaminhoMaisCurto(No origem, No destino) {
    Map<No, Aresta> anterior = new HashMap<>();
    Map<No, Integer> distancia = new HashMap<>();
    Set<No> visitados = new HashSet<>();
    PriorityQueue<No> fila = new PriorityQueue<>((n1, n2) -> Integer.compare( x: distancia.get( key: n1),
            y: distancia.get( key: n2)));
    for (No no : nos) {
        distancia.put( key: no,  value: Integer.MAX_VALUE);
    }
    distancia.put( key: origem,  value: 0);
    fila.add( e: origem);
    while (!fila.isEmpty()) {
        No noAtual = fila.poll();
        if (noAtual.equals( obj: destino)) {
            break;
        }
        if (!visitados.contains( o: noAtual)) {
            visitados.add( e: noAtual);
            for (Aresta aresta : getArestasSaindoDe(noAtual)) {
                No vizinho = aresta.getDestino();
                int pesoAresta = aresta.getPeso();
                int distanciaTentativa = distancia.get( key: noAtual) + pesoAresta;
                if (distanciaTentativa < distancia.get( key: vizinho)) {
                    distancia.put( key: vizinho,  value: distanciaTentativa);
                    anterior.put( key: vizinho,  value: aresta);
```

```
                    fila.add( e: vizinho);
                }
            }
        }
    }
    List<Aresta> caminhoMaisCurto = new ArrayList<>();
    No noAtual = destino;
    while (anterior.containsKey( key: noAtual)) {
        Aresta aresta = anterior.get( key: noAtual);
        caminhoMaisCurto.add( e: aresta);
        noAtual = aresta.getOrigem();
    }
    Collections.reverse( list: caminhoMaisCurto);
    return caminhoMaisCurto;
}

private void removeAresta(List<Aresta> caminho) {
    for (Aresta aresta : caminho) {
        arestas.remove( o: aresta); // Remove temporariamente a aresta do grafo
    }
}
```

Referências

BENATTI, H. G. Homeomorfismo em grafos: algoritmos e complexidade computacional. 1993. Citado na página 13.

BOAVENTURA, P. O. Grafos: teoria, modelos, algoritmos. 2003. Citado na página 13.

LADEIRA, L. Estruturas de dados, tipos de busca, árvores e tipos de árvores. 2021. Citado na página 3.

MAIDA, J. P. Árvore binária - implementação em java. Rio de Janeiro, Brasil, 2015. Citado nas páginas 5, 6 e 7.

OLIVEIRA, U. Árvores rubro negras. João Pessoa, Paraíba, 2015. Citado nas páginas 10, 11 e 12.

PREISS, B. Estruturas de dados e algoritmos: Padrões de projetos orientados a objeto com java. 2000. Citado na página 4.

SONG, S. W. Árvore rubro-negra - - estruturas de dados. Universidade de São Paulo, São Paulo, 2008. Citado na página 11.