UFMA - Universidade Federal do Maranhão
São Luís, 26 de maio de 2024
Dupla: Maxwell Pires Silva e Grace Kelly Barros Araujo dos Santos

Estrutura de Dados II

Relatório - Trabalho de Implementação II

Introdução

O presente relatório visa demonstrar a lógica e os códigos utilizados para resolver as questões propostas no trabalho de implementação de Índices Remissivos, utilizando as estruturas de dados Tabela Hash, Árvore AVL, Árvore Rubro-Negra e Árvore B, bem como explicar como os resultados foram alcançados com os algoritmos desenvolvidos. Na Main está presente um menu de escolha para a rápida análise de cada questão individualmente.

Objetivos

Implementar o índice remissivo e operações básicas das estruturas apresentadas em sala de aula, aplicando restrições instruídas no enunciado das questões do trabalho.

Problema 1

O problema trata sobre realizar a leitura de um arquivo e armazenar no índice remissivo palavras com mais de X caracteres e as linhas nas quais elas ocorrem, além das operações de remoção e busca de palavras e impressão do índice.

- A primeira estrutura implementada foi a Tabela Hash. Ela contém flags para controle de ocupação, dobra de tamanho após atingir uma certa porcentagem de espaços ocupados e permite a escolha do tratamento de colisões.
  A flag de controle de ocupação é definida na classe Entry, cujo atributos são os campos a serem preenchidos dentro da tabela.

```
package Hash;

public class Entry<K, V> {  7 usages
    K key;  7 usages
    V value;  5 usages
    boolean deleted; //controla ocupação de espaço  8 usages

    public Entry(K key, V value) {  1 usage
        this.key = key;
        this.value = value;
        this.deleted = false;
    }
}
```

O método resize é implementado na classe HashTable. Ele é responsável por dobrar o tamanho da tabela: criar outra tabela e inserir as entradas da antiga na tabela mais recente. Cada entrada armazenada representa um par chave-valor.

```java
//dobra o tamanho da tabela
private void resize() {  1 usage
    Entry<K, V>[] oldTable = table;
    table = new Entry[oldTable.length * 2];
    size = 0;

    for (Entry<K, V> entry : oldTable) {
        if (entry != null && !entry.deleted) {
            put(entry.key, entry.value);
        }
    }
}
```

Os tipos de sondagens foram definidos em uma classe do tipo *enum*. É possível realizar a escolha de qual estratégia será utilizada na classe Main, onde há um menu para teste desta e das demais questões.

```java
package Hash;

public enum ProbingType {  8 usages
    LINEAR,  2 usages
    QUADRATIC  1 usage
}
```

```java
case 1:
    //escolhendo sondagem
    System.out.println("Digite 1 para escolher Tentativa Linear ou digite 2 para escolher Tentativa Quadrática: ");
    int probing = 0;
    boolean next = false;
    while(!next) {

        try {
            probing = scanner.nextInt();
            if (probing == 1) {
                probingType = ProbingType.LINEAR;
                next = true;

            } else if (probing == 2) {
                probingType = ProbingType.QUADRATIC;
                next = true;

            } else {
                System.out.println("Entrada Inválida");
            }
            hashtable = new HashTable<>(probingType);
        } catch (Exception e) {
            System.out.println("Entrada Inválida");
            scanner.nextLine();
        }

    }
```

Os seguintes passos do menu são responsáveis pela escolha do valor de X e do tipo de sondagem. Após isso, é realizada a leitura e armazenamento das palavras e chaves na tabela.

```
--------  ÍNDICE REMISSIVO --------
--------        MENU       --------
Problema 1:
[1] - Utilizar Tabela Hash
[2] - Utilizar Árvore AVL
[3] - Utilizar Árvore Rubro-Negra
[4] - Utilizar Árvore B
[5] - Sair

Selecione uma opção: 1
Digite 1 para escolher Tentativa Linear ou digite 2 para escolher Tentativa Quadrática:
1
Insira um valor inteiro X: o índice armazenará palavras maiores que X:
4

-----  TABELA HASH -----

[1] - Buscar palavra:
[2] - Remover palavra:
[3] - Imprimir Índice:
[4] - Voltar.
```

Após a inserção dos elementos na tabela, é possível escolher entre as operações disponíveis: é possível buscar ou remover uma palavra específica e imprimir o índice. Na busca, além da chave e das linhas relacionadas à chave, também é retornado os passos necessários para encontrar a chave. Na impressão do índice, são exibidas as palavras armazenadas e as linhas de suas ocorrências.

```
-----  TABELA HASH -----

[1] - Buscar palavra:
[2] - Remover palavra:
[3] - Imprimir Índice:
[4] - Voltar.
3
Palavra: acaba
  Linha: 4
Palavra: academia
  Linha: 10
Palavra: aconteceu
  Linha: 5
Palavra: acreditar
  Linha: 5
```

```
-----  TABELA HASH -----

[1] - Buscar palavra:
[2] - Remover palavra:
[3] - Imprimir Índice:
[4] - Voltar.
1
Digite a palavra que deseja buscar:
jackson
Número de passos necessários em busca da chave 'jackson': 1
Palavra 'jackson' encontrada nas seguintes linhas:
Linha: 9
```

- A segunda estrutura implementada foi a Árvore AVL. Ela permite a escolha do limite do fator de balanceamento, dando mais flexibilidade além da altura entre -2 e 2 permitida tradicionalmente nesse tipo de árvore.

```java
package TreeAVL;


public class AVLTree<T extends Comparable<T>> {  4 usages

    private AVLNode<T> root;  6 usages
    private int max_height_difference; //diferença de altura entre as subarvores  9 usages


    public AVLTree(int max_height_difference) {  1 usage
        this.max_height_difference = max_height_difference;
    }
```

No menu, é possível definir o limite de altura entre as subárvores e também o valor de X. Logo após é realizada a inserção das palavras e suas linhas de ocorrência no índice.

```
--------  ÍNDICE REMISSIVO --------
--------         MENU       --------
Problema 1:
[1] - Utilizar Tabela Hash
[2] - Utilizar Árvore AVL
[3] - Utilizar Árvore Rubro-Negra
[4] - Utilizar Árvore B
[5] - Sair

Selecione uma opção: 2
Insira um valor inteiro X para definir o limite do fator de balanceamento:
1
Insira um valor inteiro X: o índice armazenará palavras maiores que X:
10
```

Seguindo esse passo, é possível acessar um menu específico para as operações do índice remissivo implementado na árvore AVL. Há as opções de buscar e remover uma palavra (chave) e imprimir o índice.

```
-----  ÁRVORE AVL -----

[1] - Buscar palavra:
[2] - Remover palavra:
[3] - Imprimir Índice:
[4] - Voltar.
3
Palavra: imediatamente (Linhas: [6, 12])
Palavra: emocionante (Linhas: [6])
Palavra: problematicas (Linhas: [10])
Palavra: problematica (Linhas: [11])
Palavra: rafjeujfejsdi (Linhas: [12])
```

```
-----  ÁRVORE AVL -----

[1] - Buscar palavra:
[2] - Remover palavra:
[3] - Imprimir Índice:
[4] - Voltar.
1
Digite a palavra que deseja buscar:
imediatamente
Palavra 'imediatamente' encontrada nas seguintes linhas:
Linha: [6, 12]
Número de passos: 1
```

Rubro Negra

Agora, na Arvore Rubro-negra, foram usadas 3 classes, Arvore_Rubro, Node_Rubro e Rubro_Reader. Esta última é semelhante ao Reader da AVL, onde fazemos a leitura do arquivo e já montamos a árvore ao mesmo tempo que encontramos as palavras no texto.
Nos outros dois arquivos foram implementadas as funções de inserção, remoção e busca, com a diferença de que o Nó da Arvore passou a ter um ArrayList para armazenar as linhas em que uma determinada chave (Palavra) ocorre. Além disso, a função de inserção foi alterada para quando tentar inserir uma palavra que já existe na Arvore, apenas adicionar uma nova linha no Arraylist daquela palavra.

```
public void insert(T palavra, int index) {
    Node_Rubro<T> newNode = new Node_Rubro<>(palavra, index);
    Node_Rubro<T> y = nullN;
    Node_Rubro<T> x = root;

    newNode.left = nullN;
    newNode.right = nullN;

    boolean existe_igual = false;
    while (x != nullN && existe_igual == false) {
        y = x;
        if (newNode.data.compareTo(x.data) == 0) {
            x.addindex(index);
            existe_igual = true;
        } else {
            if (newNode.data.compareTo(x.data) < 0) {
                x = x.left;
            } else {
                x = x.right;
            }
        }
    }

    if (existe_igual == false){
        newNode.parent = y;
        if (y == nullN) {
            root = newNode;
            root.parent = nullN;
        } else if (newNode.data.compareTo(y.data) < 0) {
            y.left = newNode;
        } else {
            y.right = newNode;
        }
        newNode.cor = RED;
        fixInsert(newNode);
    }
}
```

Temos um exemplo de indice remissivo onde x = 10 no arquivo que consta na pasta do trabalho

```
Árvore Rubro após ler do Arquivo:
imediatamente: 6,12 E: emocionante D: problematicas
emocionante: 6 E: null D: null
problematicas: 10 E: problematica D: rafjeujfejsdi
problematica: 11 E: null D: null
rafjeujfejsdi: 12 E: null D: null

Árvore Rubro após inserir a palavra pneu:
imediatamente: 6,12 E: emocionante D: problematicas
emocionante: 6 E: null D: null
problematicas: 10 E: problematica D: rafjeujfejsdi
problematica: 11 E: pneu D: null
pneu: 10 E: null D: null
rafjeujfejsdi: 12 E: null D: null

Árvore Rubro após remover a palavra pneu:
imediatamente: 6,12 E: emocionante D: problematicas
emocionante: 6 E: null D: null
problematicas: 10 E: problematica D: rafjeujfejsdi
problematica: 11 E: null D: null
rafjeujfejsdi: 12 E: null D: null

Árvore Rubro após remover a palavra problematicas:
imediatamente: 6,12 E: emocionante D: rafjeujfejsdi
emocionante: 6 E: null D: null
rafjeujfejsdi: 12 E: problematica D: null
problematica: 11 E: null D: null
```

Explicando cada parte:
Primeiro, no print, temos a cor do Nó, com sua chave e em quais linhas essas chave ocorre. Além disso, ao lado tempo o filho esquerdo representado por E: “FilhoEsquerdo” e o filho direito representado por D: “FilhoDireito”
Após ler o arquivo, temos 5 palavras. Em que os Nós vermelhos são problematica e rafjeujfejsdi.
Depois inserimos uma palavra nova, para tentarmos a inserção. Inserimos “pneu”. Como o “Tio” de onde pneu vai ser inserido é vermelho, o pai, tio e avô de pneu são recoloridos.
Depois removemos o próprio pneu, que não gera grandes diferenças, pois é a remoção de um nó folha.
Depois removemos a palavra “problemáticas”, que deve ser substituída por um nó preto, então adentramos nas condições quando o nó que substitui o nó que foi removido é preto. Nesse caso, seu filho à direita deve tomar seu lugar, pq usamos o sucessor na nossa implementação, verifica-se então o irmão de rafjeujfejsdi, que é um nó preto e seus filhos também são preto, logo deve-se mudar a cor do irmão e subir o ponteiro para atualizar o pai e continuar a verificação. Isso ocorre e é finalizada a inserção.

Problema 2
Árvore B
No Problema 2 é necessário a implementação de uma estrutura de dados de uma Árvore B, com suas funções de inserção, busca e remoção. Além disso, deveria ser implementado uma espécie de salvamento da árvore em um arquivo, para que dessa forma, a árvore pudesse ser recuperada e seus dados das chaves, que correspondem a um id de usuário de um banco de dados, não fosse perdido.

Esta implementação foi realizada com duas classes, a classe Arvore_B e a classe Node_B. No Node B foram implementadas as funções únicas para um nó, inserção, remoção e busca. Para na Arvore_B apenas manipularmos o uso dessas funções para toda a árvore. Ainda na Arvore_B temos as funções para salvar o nosso Objeto Árvore em um arquivo, e para carregar o Arquivo. Ao salvar, sobrescrevemos o conteúdo que tinha ali. E a cada inserção ou remoção, sempre salvamos o arquivo.

```
Árvore B carregada do arquivo:
Level 0 [10]
Level 1 [5, 6, 7]
Level 1 [12, 13, 20, 30, 45]

---- MENU: ARVORE B ----
[1] Inserir Nova Chave
[2] Remover Chave
[3] Mostrar Estrutura da Arvore
[4] Sair
```

Já temos uma árvore pré carregada do código. E o Menu. Podemos inserir, por exemplo, uma nova chave que force o split no segundo nó de nível 1. Vamos inserir o 70 e imprimir a arvore novamente:

```
Árvore B carregada do arquivo:
Level 0 [10, 20]
Level 1 [5, 6, 7]
Level 1 [12, 13]
Level 1 [30, 45, 70]

---- MENU: ARVORE B ----
[1] Inserir Nova Chave
[2] Remover Chave
[3] Mostrar Estrutura da Arvore
[4] Sair
```

Observemos que a Inserção gera o Split gerando um novo Nó com 40, 45, 70 e levando o 20 ao Nível 0 da Arvore.

Em seguida vamo remover o 12:

```
Árvore B carregada do arquivo:
Level 0 [7, 20]
Level 1 [5, 6]
Level 1 [10, 13]
Level 1 [30, 45, 70]

---- MENU: ARVORE B ----
[1] Inserir Nova Chave
[2] Remover Chave
[3] Mostrar Estrutura da Arvore
[4] Sair
```

Veja que ela fez a rotação correta com o irmão da esquerda do nó de onde estava o 12.
Em seguida, vamos remover o 5:

```
Árvore B carregada do arquivo:
Level 0 [20]
Level 1 [6, 7, 10, 13]
Level 1 [30, 45, 70]

---- MENU: ARVORE B ----
[1] Inserir Nova Chave
[2] Remover Chave
[3] Mostrar Estrutura da Arvore
[4] Sair
```

Por fim, ele não pode pedir emprestado ao irmão esquerdo, pois não tem, nem ao irmão direito, pois ele já tem o número mínimo de chaves. Então realiza-se o Merge.

Ainda temos a função de busca, mas ela já é utilizada implicitamente quando vamos remover, então não foi necessária demonstração.

Conclusão

Dessa forma, podemos lidar com a implementação das estruturas de Hash, Árvore B, Árvore AVL e Árvore Rubro Negra. O presente trabalho ainda destaca a importância da utilização de arquivos para salvar nossas estruturas de dados e carregá-las sem a perda de dados, além da leitura direta de arquivos .txt e busca de palavras para construir o índice remissivo.
Além disso, a utilização do Generics permite que as chaves utilizadas nas nossas estruturas possam ser de vários tipos, permitindo que elas funcionem em diversas situações.