Árvore Binária de Pesquisa

O que é?

Estrutura de dados na qual o custo de operações de inserção, remoção e substituição possuem a mesma eficiência.

A estrutura

- Nó raiz: primeiro elemento da árvore, que está localizado no nível 0. Cada nó pode ter 0, 1 ou 2 filhos.
- Nó interno: todo os nós que possuem filho(s).
- Folha: nós sem filhos.
- Arcos: arestas (ponteiros) que conectam os vértices (nós).

A organização

- Todos os elementos à esquerda da raiz são menores que ela.
- Todos os elementos à direita da raiz são maiores que ela.
- Não possui elementos repetidos.

Altura

A altura de uma árvore se dá pela distância do nó raiz ao elemento pertencente ao nível mais baixo da árvore. A altura de uma árvore que possui apenas um elemento (nó raiz) é 0.

INSERÇÃO EM ÁRVORE BINÁRIA

Funcionamento básico:

- Passo 1: se a raiz estiver vazia, inserir o elemento nela.

- Passo 2: caso o passo 1 seja falso, verificar se o elemento é menor do que a raiz. Se for, chamar o método recursivamente, para a inserção da subárvore.
- Passo 3: se o passo 1 e 2 forem falsos e o elemento for maior que a raiz, chamar recursivamente o método para a inserção da subárvore da direita.
- Passo 4: caso tudo seja falso, se o elemento for igual ao da raiz, não inseri-lo na árvore.

PESQUISA EM ÁRVORE BINÁRIA

- Método boolean.
- Passo 1: se a raiz estiver vazia, retornar um número negativo.
- Passo 2: Se !(Passo1) && o elemento procurado for igual ao da raiz, retornar uma pesquisa positiva.
- Passo 3: os passos anteriores forem falsos e o elemento de pesquisa for menor que o da raiz, chamar o método de pesquisa para a subárvore esquerda.
- Passo 4: se todos os outros passos forem falsos, o elemento procurada é maior que a raiz. Logo deve-se chamar o método de pesquisa para a subárvore da direita.

Análise de complexidade da pesquisa

- Melhor caso: $\theta(1)$.
- Pior caso: $\theta(n)$.
- Caso médio: $\theta(lg(n))$

Passo a passo

```
class No
{
	public int elemento;
	public No esq;
	public No dir;
	public No(int elemento)
	{
		this(elemento, null, null);
	 }
	public No(int elemento, No esq, No dir)
	{
		this.elemento = elemento;
		this.esq = esq;
		this.dir = dir;
	 }
 }


public class ÁrvoreBinária
{
	private No raiz;
	public ArvoreBinaria()    // inicializa nossa árvore
	{                                 // com a raiz apontado
		raiz = null;              // para null
	 }
	public void inserir(int x) throws Exception { }
	public boolean pesquisar(int x) { }
	public void remover(int x) throws Exception { }
	public void mostrarCentral() { }
	public void mostrarPre() { }
	public void mostrarPos() { }
}
```

OBS: o nó raiz precisa ser privado pois se ele for modificado, perdemos nossa estrutura.
Curiosidade: o endereço de uma árvore é o endereço da sua raiz. é comum falar “considere a árvore r” em vez de “considere a árvore cuja raiz tem endereço r”. Uma boa maneira de resolver isso é comar o nó raiz de “árvore”.

Condição necessária para a árvore estar vazia: raiz apontando para null.

Método de inserção

```
public void inserir(int x) throws Exception
{
        raiz = inserir(x, raiz);
}
```

Esse método acima é público, será chamado quando quiserem inserir um elemento na árvore. Como nosso nó raiz é privado, quem chama o método não precisa saber que ele existe e por isso, o parâmetro é apenas um inteiro.
Esse método irá chamar um outro inserir, recursivo, que recebe um inteiro, um ponteiro e retorna outro ponteiro do tipo Nó.
Eles podem ter o mesmo nome pois possuem parâmetros diferentes.

```
private No inserir(int x, No i) throws Exception
{
        if (i == null)   // se não tiver elemento inserido, ja insere o elemento
        {
                i = new No(x);
        } else if (x < i.elemento)
        {
                i.esq = inserir(x, i.esq);
        } else if (x > i.elemento)
        {
                i.dir = inserir(x, i.dir);
        } else {                // nesse caso, o elemento seria igual a um ja existente
                throw new Exception("Erro!");
        }
        return i;
}
```

Notas:

O Nó i aponta para o mesmo lugar que a raiz, possuem o mesmo valor mas lembrar que se alterarmos o i, não estamos alterando a raiz.
Quando retornamos o i, a raiz recebe esse valor. Porque? O i é uma variável local do nosso método de inserção, que se inicia com o mesmo valor da raiz mas depois muda, apontando para o elemento inserido. Quando a raiz recebe i, ocorre a ligação do novo elemento com a árvore.
Para inserir qualquer elemento, o primeiro valor mandado para i será o da raiz, mas a cada chamada recursiva o valor é de uma "nova raiz", a raiz de uma subárvore. Vale lembrar que a raiz da árvore não é modificada em momento nenhum.

Método de pesquisa

Serão dois métodos de pesquisa, um público e um privado, seguindo a mesma lógica dos métodos de inserção.

```
public boolean pesquisar (int x)
{
	return pesquisar(x, raiz);
}

private boolean pesquisar (int x, No i)
{
	boolean resp;
	if( i == null)
	{
		resp = false;
	}
```

```
        else if (x == i.elemento)
        {
                resp = true;
        }else if (x < i.elemento)
        {
                resp = pesquisar(x, i.esq)
        }else
        {
                resp = pesquisar(x, i.dir);
        }
        return resp;
}
```

Notas:
Esse método não modifica nossa árvore nem precisa de tratamento de exceções.
Condição de parada: i.elemento ser igual a null ou ser o elemento procurado.

Métodos de remoção

Durante uma remoção, podemos nos deparar com 3 cenários: remover uma folha, um nó com um filho ou um nó com dois filhos.
No momento de remover o elemento, o nó i vai estar apontando para ele, certo? Agora é o momento de fazer os testes para verificar se esse nó tem filhos e se sim, quantos.
Verificamos se um dos ponteiros (esq, ou dir) é null, se for, é retornada o valor do ponteiro que não foi analisado. Como assim? Se i.esq for null, ja retornamos o

valor de i.dir (não precisamos verificar seu conteúdo). Lembrando que essa chamada é recursiva, esse valor retorna para a última chamada que foi a do ponteiro do seu pai. Assim o pai vai receber o ponteiro que seu filho tinha para o único filho ou para null, caso fosse uma folha.

Se o nó tiver dois filhos, temos duas opções: substituir o valor que vai ser removido, pelo maior valor da subárvore esquerda (valor mais à direita possível dessa subárvore) ou substituir o valor que vai ser removido, pelo menor valor possível da subárvore à direita (valor mais à esquerda possível dessa subárvore). Essa escolha deve ser tomada no início do código, uma vez decidida, todos os elementos serão removidos com o mesmo parâmetro. Vou exemplificar substituindo o nó removido pelo maior valor da subárvore esquerda.

Agora fazemos o que? Chamamos um novo método (anterior, que vai retornar um Nó). Esse método recebe dois ponteiros como argumentos. O primeiro, nó i, é o endereço do elemento que vai ser removido. O segundo, nó j, é o elemento do ponteiro a esquerda do seu anterior. Esse ponteiro vai caminhando para a direita até que o j.dir seja null. O elemento de i vai receber o elemento de j (removemos i) e j recebe j.esq (temos certeza que não tem mais nada a direita, não tem diferença se tiver algo a esquerda). J é retornado para quem o chamou, linkando ele na árvore e assim adiante até acabar os empilhamentos da nossa chamada recursiva.

A exceção do código é “i” ser nulo, logo, não ter o elemento para remover.

```
public void remover (int x) throw new Exception
{
```

```
        raiz = remover(x,raix);
}

private No remover(int x, No i) throws Exception
{
        if (i==null)
        {
                throw new Exception (“Erro!”);

        }else if (x < i.elemento)
        {
                i.esq = remover(x, i.esq);

        }else if (x > i.elemento)
        {
                i.dir = remover(x, i.dir);

        }else if (i.dir == null)
        {
                i = i.esq;

        }else if (i.esq == null)
        {
                i = i.dir;
        }else
        {
                i.esq = anterior(i, i.esq);
        }
        return i;
}
private No anterior (No i, No j)
{
        if (j.dir != null) j.dir = anterior(i, j.dir);
```

```
        else
        {
                i.elemento = j.elemento; j = j.esq;
        }
        return j;
}
```

Obs: a raiz sempre recebe o retorno pq como é recursivo, o último elemento retornado sempre vai ser o endereço da raiz e assim a árvore já fica modificada, caso o processo que ela sofreu fosse modificador.

Dúvida: se eu quiser ter esse elemento removido, como faço? Tem que mandar um array ou uma lista como argumento para a função de remover?

Método mostrar

Temos três métodos diferentes de mostrar, cada um com uma ordem diferente.

- Mostrar central: mostra os elementos em ordem crescente.

```
public void mostrarCentral()
{
        mostrarCentral (raiz);
}
```

```
private void mostrarCentral(No i)
{
	if (i != null)
	{
		mostrarCentral (i.esq);
		Sop(i.elemento + " ");
		mostrarCentral(i.dir);
	}
}
```

Mostrar pós ordem: printa primeiro o conteúdo de todas as subárvore e, depois, da raiz.

Dúvidas:

```
204    /*
205     * Esse método vai somar todos os elementos de
uma árvore
206     */
207    public int somar()
208    {
209       return somar(raiz, 0);
210    }
211
212    private int somar(No i, int somar)
213    {
214       if (i != null)
215       {
216          MyIO.println ("elemento: " + i.elemento);
217          somar += i.elemento;
218          somar(i.esq, somar);
219          somar(i.dir, somar);
220       }
221
222       return somar;
223
224    }
```

tive que criar uma variável global