Teoria das Árvores

Árvores Binárias

- **Definição:**

Árvores binárias de pesquisa que são projetadas para um acesso rápido à informação. Idealmente a árvore deve ser razoavelmente equilibrada e a sua altura será dada (no caso de estar completa) por h=log2 (n+1) O tempo de pesquisa tende a O(log2N) Porém, com sucessivas inserções de dados principalmente ordenados, ela pode se degenerar para O(n).

- **Árvores Completas:**

São aquelas que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas.

Inserção de elementos em uma árvore binária de busca

• O primeiro elemento inserido assumirá o papel de raiz da árvore;
• Todo novo elemento entrará na árvore como uma folha;
• Se o elemento for menor ou igual à raiz será inserido no ramo da esquerda. Caso contrário, no ramo da direita (para árvores decrescentes inverte-se a regra).

Remoção de elementos em uma árvore binária de busca

Considerando que podemos remover qualquer elemento de uma árvore, podem ocorrer as seguintes situações:

1. O Elemento a ser removido é um nó folha (sem filhos à esquerda e à direita);
2. O Elemento a ser removido possui apenas um filho (à direita ou à esquerda);
3. O Elemento a ser removido possui dois filhos.

- ## Travessia da árvore:

- **pré-ordem** (os filhos de um nó são processados após o nó)

    - Cima para baixo
- **pós-ordem** (os filhos são processados antes do nó)
    - Baixo para cima
- **em-ordem** , em que se processa o filho à esquerda, o nó, e finalmente o filho à direita.

- Métodos de Balanceamento

- Há duas categorias: dinâmico e global (ou estático).
- O rebalanceamento dinâmico mantém a árvore balanceada toda vez que um nó é inserido ou removido. AVL é o melhor exemplo
- O global permite a árvore crescer sem limites e somente faz o balanceamento quando tal necessidade é acionada, externamente.

Existe alguma razão para evitarmos algoritmos de busca em árvore recursivos (memória, complexidade, etc)? Justifique.

Sim, pois quando essa busca acontece no pior dos casos, ou seja, a árvore é degenerada, a busca irá percorrer todos os nós de um lado, apenas para depois desempilhar os valores na pilha de recursão e retornar nulo quando chegar na raiz, caso não encontre o valor. Isso faz com que a complexidade do algoritmo aumente, sendo ele O(logn), consequentemente aumentando seu tempo de execução.

E ainda existem outras formas de busca, como a busca iterativa, que também possui loops mas que não utilizam recursão, e sim repetição, resultando numa complexidade e tempo de execução menores.

AVL

Características:

- É uma árvore altamente balanceada, isto é, nas inserções e exclusões, procura-se executar uma rotina de balanceamento tal que as alturas das sub-árvores esquerda e sub-árvores direita tenham alturas bem próximas
- Idealmente a árvore deve ser razoavelmente equilibrada e a sua altura será dada (no caso de estar completa) por - h=log2(n+1)
- A árvore AVL tem **complexidade O(log n)** para todas operações e ocupa espaço n, onde n é o número de nós da árvore.

Complexidade da árvore AVL em notação O

	Média	Pior caso
Espaço	$O(n)$	$O(n)$
Busca	$O(\log n)$	$O(\log n)$
Inserção	$O(\log n)$	$O(\log n)$
Deleção	$O(\log n)$	$O(\log n)$

- **Definição:** Uma árvore AVL é uma árvore na qual as alturas das subárvores esquerda e direita de cada nó diferem no máximo por uma unidade.

- ## Como ocorre a Busca

A busca é a mesma utilizada em árvore binária de busca.
A busca pela chave de valor K inicia sempre pelo nó raiz da árvore. Seja pt_u um ponteiro para o nó u sendo verificado. Caso o pt_u seja nulo então a busca não foi bem sucedida (K não está na árvore ou árvore vazia). Verificar se a chave K igual pt_u->chave (valor chave armazenado no nó u), então a busca foi bem sucedida. Caso contrário, se K < pt_u->chave então a busca segue pela subárvore esquerda; caso contrário, a busca segue pela subárvore direita.

Exemplo de algoritmo de busca em Java.

```java
// O método de procura numa AVL é semelhante ao busca binária de uma árvore binária de busca comum.
public BSTNode<T> search(T element) {
    return search(element, this.root);
}

// Método auxiliar à recursão.
private BSTNode<T> search(T element, BSTNode<T> node) {
    if (element == null || node.isEmpty()) {
        return new BSTNode<T>();
    }

    if (node.isEmpty() || node.getData().equals(element)) {
        return node;
    } else if (node.getData().compareTo(element) > 0) {
        return search(element, node.getLeft());
    } else {
        return search(element, node.getRight());
    }
}
```

Inserção

Para inserir um novo nó de valor K em uma árvore AVL é necessária uma busca por K nesta mesma árvore. Após a busca, o local correto para a inserção do nó K será em uma subárvore vazia de uma folha da árvore. Depois de inserido o nó, a altura do nó pai e de todos os nós acima deve ser atualizada. Em seguida o algoritmo de rotação simples ou dupla deve ser acionado para o primeiro nó pai desregulado.

Os parâmetros p e mudou_h são passados por referência. O ponteiro p aponta para o nó atual. O parâmetro mudou_h é do tipo lógico e informa ao chamador se a subárvore apontada por p mudou sua **altura.**

Como identificar mudança de altura?

Considerar que o nó p é raiz da subárvore T_p e houve inserção em uma de suas subárvores.

Caso a subárvore T_p tenha mudado de altura, decrementar fb (inserção na subárvore esquerda) ou incrementar fb (inserção na subárvore direita).

Caso 1: Ao inserir um nó folha, a subárvore T_p passa de altura 0 para altura 1, então T_p mudou de altura.

Caso 2: fb=0 antes da inserção foi alterado para 1 ou -1, então a subárvore T_p mudou de altura.

Caso 3: fb=1 ou -1 antes da inserção, passou a ter valor 0, então a subárvore T_p não mudou de altura.

Caso 4: O fb passou a ter valor -2 ou 2 após a inserção, então há necessidade de aplicação de alguma operação de rotação. Após a rotação, a subárvore T_p terá a mesma altura anterior à inserção.

Exemplo de algoritmo de inserção em Java

```
/* Por definição, a árvore AVL é uma árvore binária de busca (BST).
*  Por este motivo utiliza-se aqui a mesma definição (classe) de Nós que uma BST simples.
*/
public void insert(T element) {
   insertAux(element);
   BSTNode<T> node = search(element); // Pode-se utilizar o mesmo search exemplificado acima.
   rebalanceUp(node);
}
```

```
private void insertAux(T element) {
    if (element == null) return;
    insert(element, this.root);
}

private void insert(T element, BSTNode<T> node) {
    if (node.isEmpty()) {
        node.setData(element);
        node.setLeft(new BSTNode<T>());
        node.setRight(new BSTNode<T>());
        node.getLeft().setParent(node);
        node.getRight().setParent(node);
    } else {
        if (node.getData().compareTo(element) < 0) {
            insert(element, node.getRight());
        } else if (node.getData().compareTo(element) > 0) {
            insert(element, node.getLeft());
        }
    }
}
```

Algoritmos de complemento à inserção e/ou algoritmos para identificar desbalanceamento em Java

```
protected void rebalanceUp(BSTNode<T> node) {
    if (node == null || node.isEmpty()) return;
    rebalance(node);
    if (node.getParent() != null) {
        rebalanceUp(node.getParent());
    }
}

protected int calculateBalance(BSTNode<T> node) {
    if (node == null || node.isEmpty()) return 0;
    return height(node.getRight()) - height(node.getLeft());
}

protected void rebalance(BSTNode<T> node) {
    int balanceOfNode = calculateBalance(node);

    if (balanceOfNode < -1) {
        if (calculateBalance(node.getLeft()) > 0) {
            leftRotation(node.getLeft());
        }
        rightRotation(node);
    } else if (balanceOfNode > 1) {
```

```
        if (calculateBalance(node.getRight()) < 0) {
            rightRotation(node.getRight());
        }
        leftRotation(node);
    }
}
```

Rotação para Direita e para Esquerda em Java

```
protected void leftRotation(BSTNode<T> no) {
        BTNode<T> noDireito = no.getRight();

        no.setRight(noDireito.getLeft());

        noDireito.getLeft().setParent(no);
        noDireito.setLeft(no);
        noDireito.setParent(no.getParent());
        no.setParent(noDireito);

        if (no != this.getRoot()) {
                if (noDireito.getParent().getLeft() == no) {
                        noDireito.getParent().setLeft(noDireito);
                } else {
                        noDireito.getParent().setRight(noDireito);
                }
        } else {
                        this.root = (BSTNode<T>) noDireito;
        }
}

protected void rightRotation(BSTNode<T> no) {
        BTNode<T> noEsquerdo = no.getLeft();

        no.setLeft(noEsquerdo.getRight());

        noEsquerdo.getRight().setParent(no);
        noEsquerdo.setRight(no);
        noEsquerdo.setParent(no.getParent());
        no.setParent(noEsquerdo);

        if (no != this.getRoot()) {
                if (noEsquerdo.getParent().getLeft() == no) {
                        noEsquerdo.getParent().setLeft(noEsquerdo);
                } else {
                        noEsquerdo.getParent().setRight(noEsquerdo);
                }
        } else {
                this.root = (BSTNode<T>) noEsquerdo;
```

```
        }
    }
```

Remoção

O primeiro passo para remover uma chave K consiste em realizar uma busca binária a partir do nó raiz. Caso a busca encerre em uma subárvore vazia, então a chave não está na árvore e a remoção não pode ser realizada. Caso a busca encerre em um nó u o nó que contenha a chave então a remoção poderá ser realizada da seguinte forma:

Caso 1: O nó u é uma folha da árvore, apenas exclui-lo.

Caso 2: O nó u tem apenas uma subárvore, necessariamente composta de um nó folha, basta apontar o nó pai de u para a única subárvore e excluir o nó u.

Caso 3: O nó u tem duas subárvores: localizar o nó v predecessor ou sucessor de K, que sempre será um nó folha ou possuirá apenas uma subárvore; copiar a chave de v para o nó u; excluir o nó v a partir da respectiva subárvore de u.

O último passo consiste em verificar a desregulagem de todos nós a partir do pai do nó excluído até o nó raiz da árvore. Aplicar rotação simples ou dupla em cada nó desregulado.

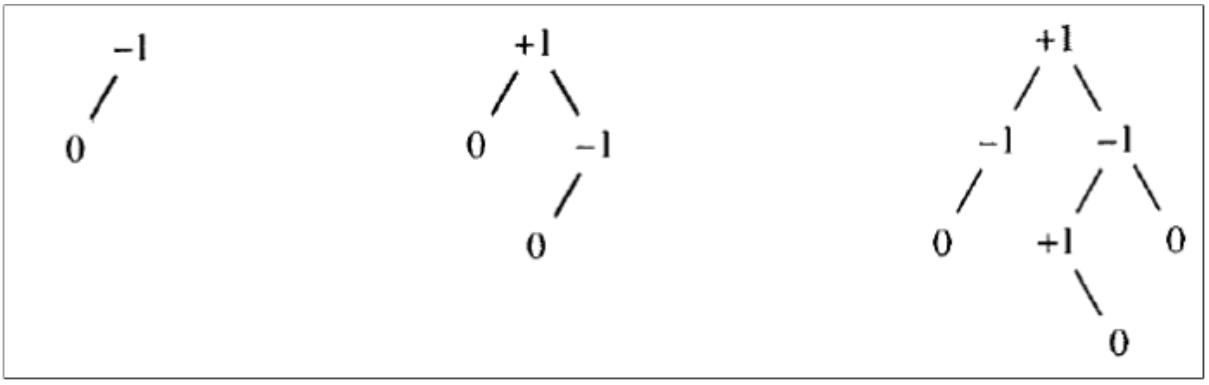
Exemplo de algoritmo de remoção em Java

```
    public void remover(int valor) {
            removerAVL(this.raiz, valor);
        }
```

```
	private void removerAVL(No atual, int valor) {
		if (atual != null) {

			if (atual.getChave() > valor) {
				removerAVL(atual.getEsquerda(), valor);

			} else if (atual.getChave() < valor) {
				removerAVL(atual.getDireita(), valor);

			} else if (atual.getChave() == valor) {
				removerNoEncontrado(atual);
			}
		}
	}

	private void removerNoEncontrado(No noARemover) {
		No no;

		if (noARemover.getEsquerda() == null || noARemover.getDireita() ==
null) {

			if (noARemover.getPai() == null) {
				this.raiz = null;
				noARemover = null;
				return;
			}
			no = noARemover;

		} else {
			no = sucessor(noARemover);
			noARemover.setChave(no.getChave());
		}

		No no2;
		if (no.getEsquerda() != null) {
			no2 = no.getEsquerda();
		} else {
			no2 = no.getDireita();
		}

		if (no2 != null) {
			no2.setPai(no.getPai());
		}

		if (no.getPai() == null) {
			this.raiz = no2;
		} else {
			if (no == no.getPai().getEsquerda()) {
				no.getPai().setEsquerda(no2);
```

```
                } else {
                        no.getPai().setDireita(no2);
                }
                verificarBalanceamento(no.getPai());
        }
        no = null;

}
```

- **Balanceamento: hd – he ϵ {0, 1, -1}**



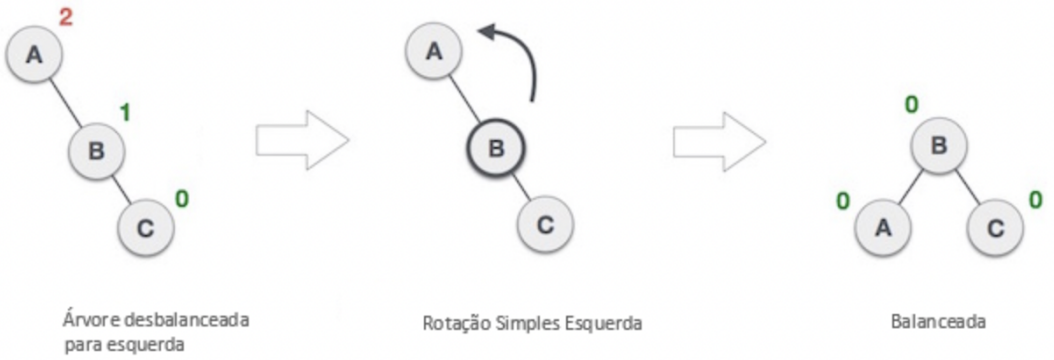
Se o **fator de balanceamento** de qualquer nó ficar menor do que -1 ou maior do que 1 então a árvore tem que ser balanceada.

Rotações em AVL

Nas árvores AVL, após cada operação, como inserção e exclusão, o fator de balanceamento de cada nó precisa ser verificado. Se cada nó satisfizer a condição do fator de balanceamento, a operação pode ser concluída. Caso contrário, a árvore precisa ser rebalanceada utilizando as operações de rotação. Existem quatro rotações e elas são classificadas em dois tipos.

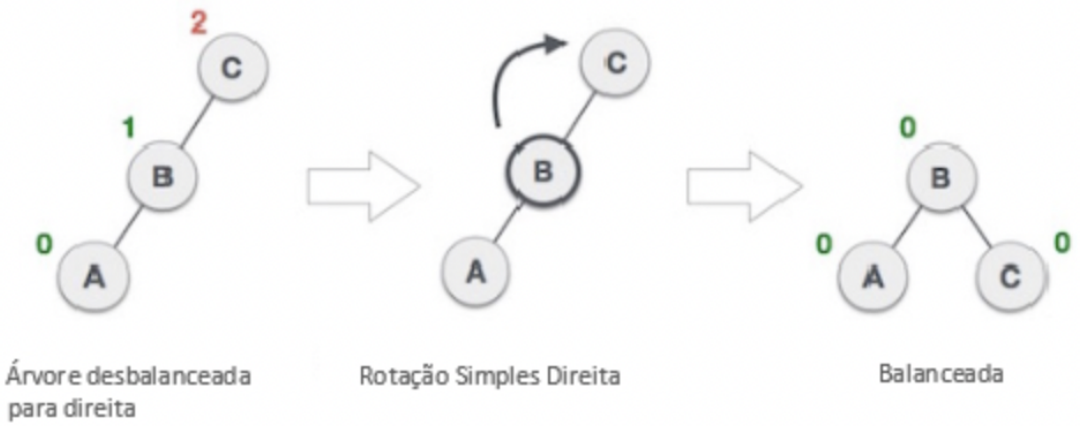
Rotação simples à esquerda (rotação SE - RR)

Na rotação simples à esquerda, cada nó se move uma posição para a direita da posição atual.



Rotação simples à direita (rotação SD - LL)

Na rotação simples à direita, cada nó se move uma posição para a direita da posição atual.



Rotação dupla à direita (rotação DD)

As rotações duplas à direita são uma combinação de uma única rotação para a esquerda seguida de uma rotação para a direita.

Primeiro, cada nó se move uma posição para a esquerda. Depois, se move uma posição para a direita da posição atual.

Rotação dupla à esquerda (rotação DE)

As rotações duplas à esquerda são uma combinação de uma única rotação para a direita seguida de uma rotação para a esquerda. Primeiro, cada nó se move uma posição para a direita. Depois, se move uma posição para a esquerda da posição atual.

OBS: As operações de rotação possuem tempo constante, já que a única coisa que acontece nessas ações é a mudança de alguns ponteiros. Assim como as rotações, recuperar o fator de balanceamento e atualizar a altura também possuem tempo constante. Dessa forma, a complexidade de inserção em uma AVL continua sendo O(h), em que h é a altura da árvore, tal qual em uma árvore binária de busca. Mas, como a AVL é balanceada, a altura da árvore é O(Log n), sendo também a

complexidade de inserção de nó nesse tipo de árvore.

Árvore B

Definição:

- Árvore B é uma estrutura que, ao invés de armazenar chaves em nós individuais, utiliza um bloco de chaves, chamado página, para armazenar vários valores. Cada nó de uma árvore B é uma página.
- Muito utilizada para armazenamento e recuperação de dados. Ao contrário da AB, pode ter mais do que dois filhos

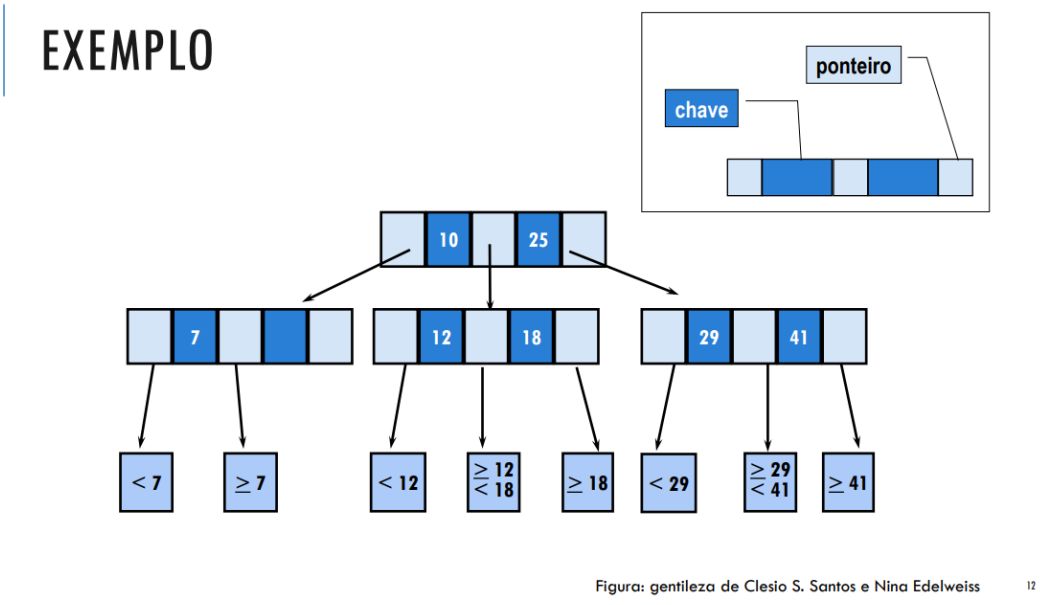
Características:

Árvores B possuem ponteiros que apontam para os múltiplos caminhos e possui auto-balanceamento. Ela costuma não ter uma altura tão grande comparada à outras árvores, pelo fato de armazenar blocos de chaves (as páginas).

OBS: existem dois conceitos para determinação do número de ordem de uma árvore, O conceito de Rudolf Bayer e Edward McCreight, 1972, criadores da árvore B, constata que a ordem de uma árvore é a metade da capacidade de chaves que sua página pode armazenar. Dessa forma, sendo uma árvore com ordem **3**, suas páginas podem armazenar entre **3** e **6** chaves.
Já o conceito de Donald Knuth, 1978, constata que a ordem uma árvore B é a quantidade máxima de filhos que uma página da árvore pode ter, e que cada página contém, sendo **d** a ordem dessa árvore, entre (2d - 1 / 3) e d - 1 chaves. Então, sendo uma árvore com ordem **5**, suas páginas podem armazenar entre **2** e **4** chaves, e pode ter no máximo **5** filhos. Esta variação de Knuth é chamada de Árvore B*.

Cada árvore B possui uma ordem, que determina as características dela. Uma Árvore B de ordem ***d*** deve ter as seguintes propriedades:

- a raiz da árvore OU é uma folha, ou seja, é o único nó da árvore, OU possui no mínimo 2 filhos;
- cada nó interno possui no mínimo d + 1 filhos;
- cada nó possui no máximo 2d + 1 filhos;
- as chaves das páginas devem ser ordenadas do menor para o maior;
- cada página possui entre d e 2d chaves, exceto o nó raiz, que possui entre 1 e 2d chaves;
- a quantidade de ponteiros de uma página é a quantidade de chaves + 1.
- folhas estão sempre no mesmo nível;



- Podem ter n chaves por nó (Filhos) → Ordem m
- Arruma a estrutura em forma serial na memória secundária
    - Usada para implementar BD
- Complexidade no pior caso :
- Os valores dos nós ficam junto com as chaves

- As buscas são feitas carregando apenas algumas partes por vez, para poupar memória
- São perfeitamente balanceadas -> Sempre que add um valor, reorganiza aquele nó com base na ordem

A altura mínima de uma árvore B com *n* número de nós e *m* número máximo de filhos que um nó não-raiz pode ter é:

$$h_{min} = \lceil \log_m(n+1) \rceil - 1$$

A altura máxima de uma árvore B com *n* número de nós e *t* número mínimo de filhos que um nó pode ter é:

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor \text{ and } t = \lceil \frac{m}{2} \rceil$$

Usos e Vantagens

Em algumas aplicações, a quantidade de nós é grande demais para serem armazenadas somente em memória, então é necessário o uso de memória secundária. Isso causa um grande gasto de tempo para acesso a um só nó de dados. Então é usada a Árvore B, que tem mais de uma chave por nó. Ela é usada para busca eficiente para dados armazenados em memória secundária (disco rígido).

Árvores B são usadas em:

- sistemas de arquivos do Windows, Mac e Linux
- bancos de dados como ORACLE, Db2, SQL, PostgreSQL, INGRES
- servidores também utilizam a abordagem de árvore B
- utilizado em sistemas CAD (computer-aided design)

Vantagens

- Altura menor comparada a outras árvores;
- Ideais para uso como índice de arquivo em disco;
- Como é uma árvore baixa, são necessários poucos acessos em disco até chegar ao ponteiro para o bloco que contém o registro necessário;
- indexação multinível;

Desvantagens

- São baseadas em estruturas de dados baseados em disco, e tem alto uso dele;
- Um pouco lenta comparada com outras árvores, não é a melhor escolha em todos os casos;

Complexidade

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)

💡 Quanto mais chaves em um nó, menos níveis a árvore tem e mais otimizada a busca é

💡 Os galhos que os nós podem ter, tendo por ex 3 ou 4 em um nó só ligado a valores diferentes se entende por:

- Dois galhos que saem de um valor x : Um vai para nós com valores menores que x e outro entre x e y
- Um galho que sai de y (que está ao lado de x) : Pode ir para um valor maior que y

Ou seja, os vários links que saem de cada valor estão diretamente relacionados a outros valores menores ou maiores que eles

Busca

- Carrega o primeiro nó e verifica se o valor está antes ou depois do nó (direita ou esquerda) com base no valor se é maior ou menor, e vai descendo pelos nós da árvore
- Para cada nó não folha visitado:
    - Se o nó tem a chave, retorna a chave
    - Se não, desce para o filho apropriado, baseado no valor da chave se é menor ou maior que os das chaves do nó atual
- Se chegar num nó folha e não for encontrado na posição que deveria estar, retorna NULL.

Algoritmo de Busca em Java

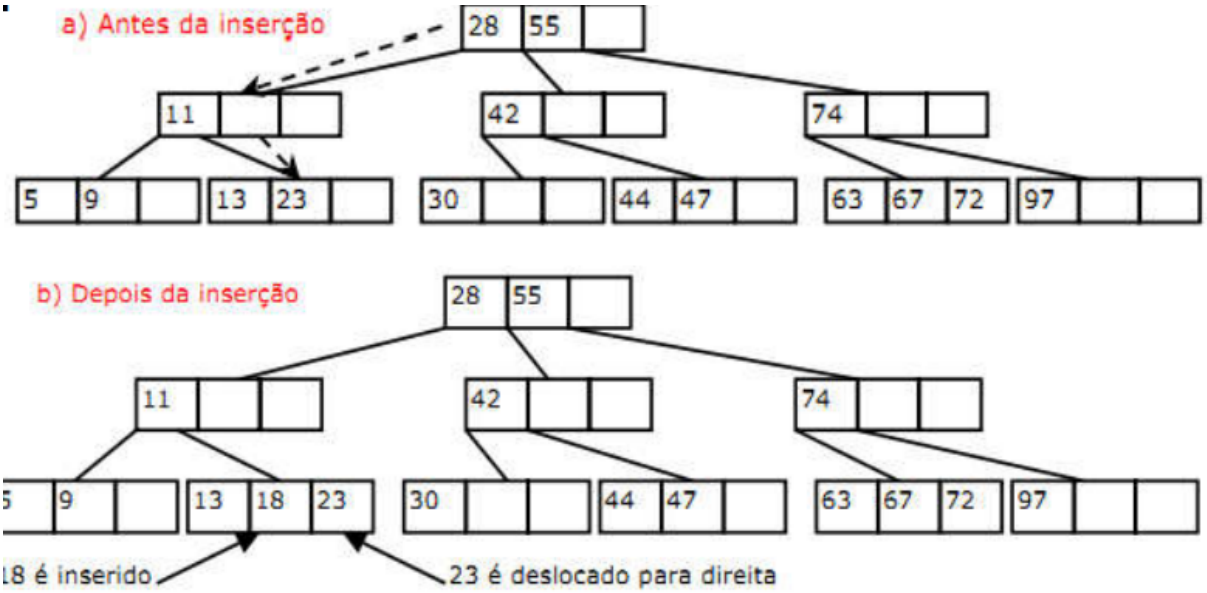
```
public BNodePosition<T> search(T element) {
        return searchAux(root, element);}
private BNodePosition<T> searchAux(BNode<T> node, T element) {
        int i = 0;
        BNodePosition<T> nodePosition = new BNodePosition<T>();
        while (i <= node.elements.size() && element.compareTo(node.elements.get(i)) >
0) {
                i++;}
        if (i <= node.elements.size() && element.equals(node.elements.get(i))) {
                nodePosition.position = i;
                nodePosition.node = node;
                return nodePosition;}
        if (node.isLeaf()) {
                return new BNodePosition<T>();}
        return searchAux(node.children.get(i), element);
}
```

Inserção

- Vai ordenando os valores conforme for inserindo, ou seja, mudando de posição caso entre um valor menor ou valor que o que estava. De forma que a árvore fique balanceada e que a página fique ordenada da menor para a maior chave.

Como é feita a inserção:

- Executar algoritmo de busca, que identifica a posição em que a chave deverá ser inserida. Se a inserção for válida, insere a chave no nó adequado.



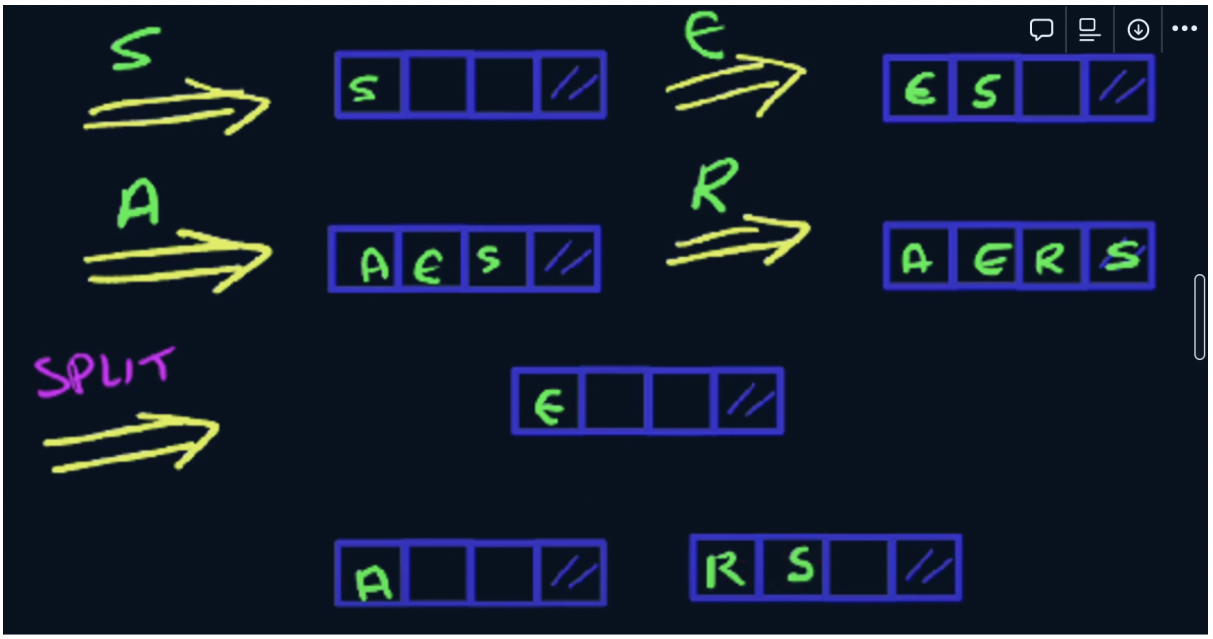
Algoritmo de Inserção em Java

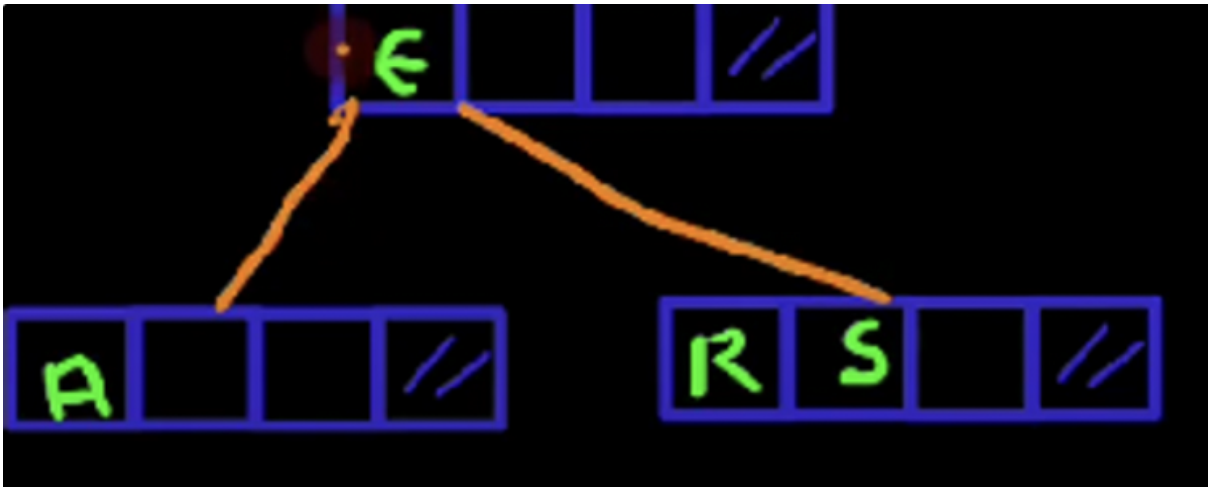
```java
//Metodo de Inserção na ArvoreB
//parametros: k - chave a ser inserida
public void insere(int k) {
    //Verifica se a chave a ser inserida existe
    if (BuscaChave(raiz, k) == null) { //só insere se não houver, para evitar duplicação de chaves
        //verifica se a chave está vazia
        if (raiz.getN() == 0) {
            raiz.getChave().set(0, k);//seta a chave na primeira posição da raiz
            raiz.setN(raiz.getN() + 1);
        } else { //caso nao estaja vazia
            No r = raiz;
            //verifica se a raiz está cheia
            if (r.getN() == ordem - 1) {//há necessidade de dividir a raiz
                No s = new No(ordem);
                raiz = s;
                s.setFolha(false);
                s.setN(0);
                s.getFilho().set(0, r);
                divideNo(s, 0, r);//divide nó
                insereNoNaoCheio(s, k);//depois de dividir a raiz começa inserindo apartir da raiz

            } else {//caso contrario começa inserindo apartir da raiz
                insereNoNaoCheio(r, k);
            }
        }
        nElementos++;//incrementa o numero de elemantos na arvore
    }
}
```

```java
//Método para inserir uma chave em um nó não cheio
//Paâmetros: x - nó a ser inserido, k - chave a ser inserida no nó x
public void insereNoNaoCheio(No x, int k) {
    int i = x.getN() - 1;
    //verifica se x é um nó folha
    if (x.isFolha()) {
        //adquire a posição correta para ser inserido a chave
        while (i >= 0 && k < x.getChave().get(i)) {
            x.getChave().set(i + 1, x.getChave().get(i));
            i--;
        }
        i++;
        x.getChave().set(i, k);//insere a chave na posição i
        x.setN(x.getN() + 1);

    } else {//caso x não for folha
        //adquire a posição correta para ser inserido a chave
        while ((i >= 0 && k < x.getChave().get(i))) {
            i--;
        }
        i++;
        //se o filho i de x estiver cheio, divide o mesmo
        if ((x.getFilho().get(i)).getN() == ordem - 1) {
            divideNo(x, i, x.getFilho().get(i));
            if (k > x.getChave().get(i)) {
                i++;
            }
        }
        //insere a chave no filho i de x
        insereNoNaoCheio(x.getFilho().get(i), k);
    }

}
```

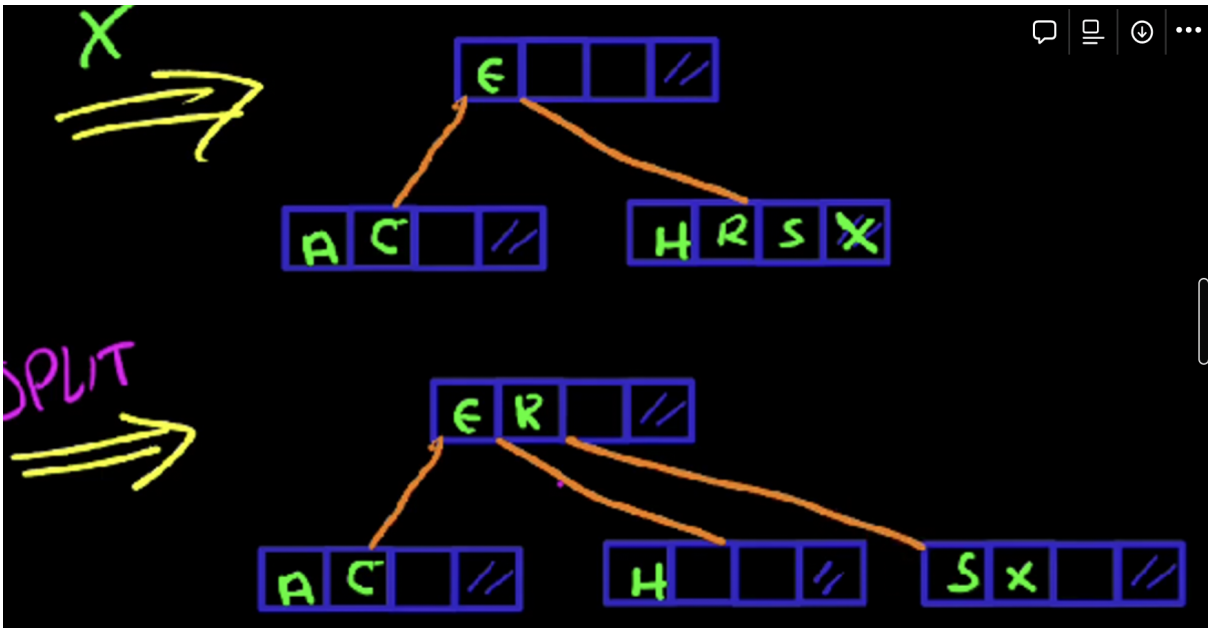
Em caso de página cheia:

- Quando o último campo é ocupado, faz o **split**
    - Divide o nó na metade e pega o último elemento da primeira metade. Sobe ele para um nó acima, criando uma nova raiz
    - Os elementos que sobraram são redivididos em novos nós abaixo do que subiu
    - Ex 1: O **E** subiu e os outros elementos viraram outros nós

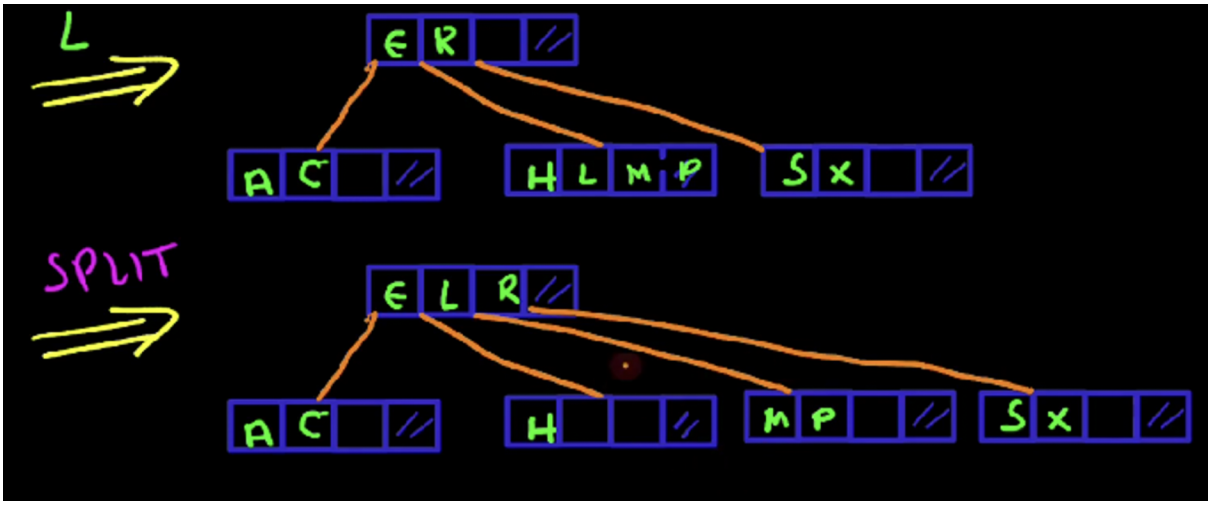

S
S
E
E S
A
A E S
R
A E R S
SPLIT
E
A
R S


E
A
R S

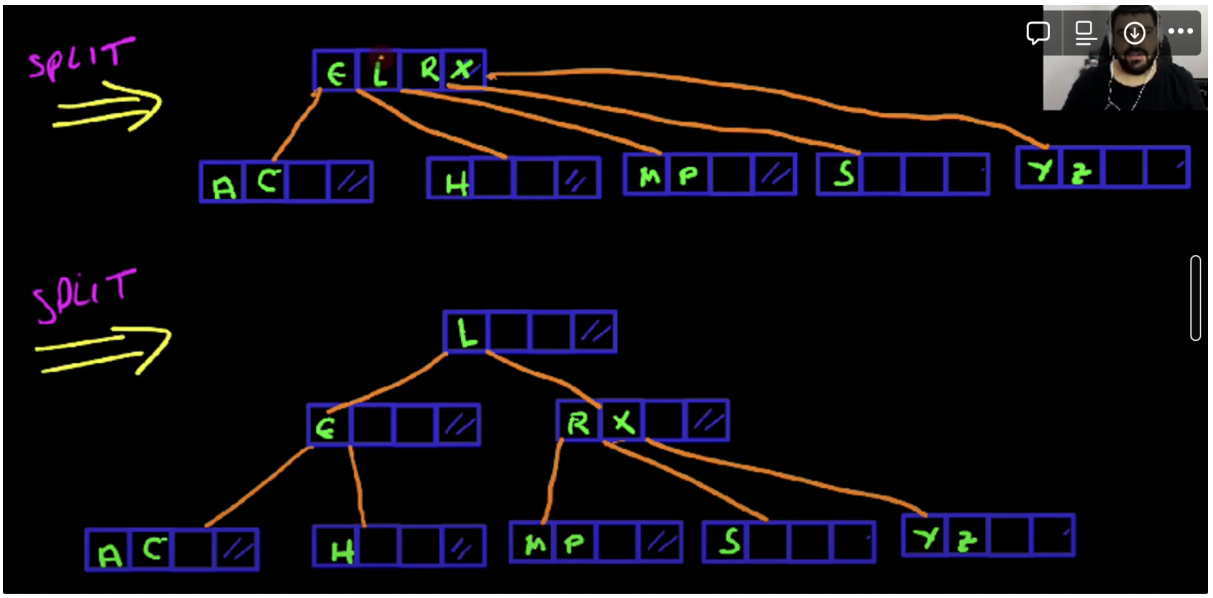
- Ex 2 :



- Ex 3 :



- Quando o Split tem que dividir o nó raiz, criando outro nivel → Mesma coisa dos outros



Algoritmo de split

```java
//Método de divisão de nó
//Parâmetros: x - nó Pai, y - nó Filho e i - índice i que indica que y é o i-ésimo filho de x.
public void divideNo(No x, int i, No y) {
    int t = (int) Math.floor((ordem - 1) / 2);
    //cria nó z
    No z = new No(ordem);
    z.setFolha(y.isFolha());
    z.setN(t);

    //passa as t ultimas chaves de y para z
    for (int j = 0; j < t; j++) {
        if ((ordem - 1) % 2 == 0) {
            z.getChave().set(j, y.getChave().get(j + t));
        } else {
            z.getChave().set(j, y.getChave().get(j + t + 1));
        }
        y.setN(y.getN() - 1);
    }

    //se y nao for folha, pasa os t+1 últimos flhos de y para z
    if (!y.isFolha()) {
        for (int j = 0; j < t + 1; j++) {
            if ((ordem - 1) % 2 == 0) {
                z.getFilho().set(j, y.getFilho().get(j + t));
            } else {
                z.getFilho().set(j, y.getFilho().get(j + t + 1));
            }

        }
    }

    y.setN(t);//seta a nova quantidade de chaves de y

    //descola os filhos de x uma posição para a direita
    for (int j = x.getN(); j > i; j--) {
        x.getFilho().set(j + 1, x.getFilho().get(j));
    }

    x.getFilho().set(i + 1, z);//seta z como filho de x na posição i+1

    //desloca as chaves de x uma posição para a direita, para podermos subir uma chave de y
    for (int j = x.getN(); j > i; j--) {
        x.getChave().set(j, x.getChave().get(j - 1));
    }

    //"sobe" uma chave de y para z
    if ((ordem - 1) % 2 == 0) {
        x.getChave().set(i, y.getChave().get(t - 1));
        y.setN(y.getN() - 1);

    } else {
        x.getChave().set(i, y.getChave().get(t));
    }

    //incrementa o numero de chaves de x
    x.setN(x.getN() + 1);

}
```

Remoção

Duas situações possíveis:

- Chave X está em um nó folha. Nesse caso, simplesmente retira-se a chave
- Se não estiver em um nó folha, pegar a chave imediatamente maior e substituir por ela

No caso de uma remoção fazer com que a quantidade de chaves fique menor do que o mínimo possível, existem duas soluções possíveis:

Concatenação: duas páginas podem ser unidas, concatenadas, se elas forem irmãs adjacentes e se juntas possuírem menos de 2d chaves (número máximo de chaves).
(Páginas são irmãs adjacentes se têm o mesmo pai e são apontadas por ponteiros adjacentes desse pai)

Passo a passo:

- Seja página X pai das páginas Y e Z, que são irmãs adjacentes
- Z teve uma chave removida e que ficou com menos de d chaves
- Transferir chaves de Z para Y
- Transferir a chave de X que separava os ponteiros de Y e Z para Y
- Eliminar página Z e ponteiro

Redistribuição: ocorre quando a soma de chaves de páginas irmãs adjacentes é maior que 2d.

Passo a passo:

- Seja página X pai das páginas Y e Z, que são irmãs adjacentes
- Colocar em Y d chaves
- Colocar em X a chave d + 1
- Colocar em Z as chaves restantes

Se ambas concatenação e redistribuição forem possíveis (se possuírem 2 nós adjacentes, cada um levando a uma solução diferente), optar pela redistribuição, que é menos custosa, não se propaga e evita que o nó fique cheio, deixando espaço para futuras inserções

Algoritmo de Deleção

```java
//Método de Remoção de uma determinada chave da arvoreB
public void Remove(int k) {
    //verifica se a chave a ser removida existe
    if (BuscaChave(this.raiz, k) != null) {
        //N é o nó onde se encontra k
        No N = BuscaChave(this.raiz, k);
        int i = 1;

        //adquire a posição correta da chave em N
        while (N.getChave().get(i - 1) < k) {
            i++;
        }

        //se N for folha, remove ela e deve se balancear N
        if (N.isFolha()) {
            for (int j = i + 1; j <= N.getN(); j++) {
                N.getChave().set(j - 2, N.getChave().get(j - 1));//desloca chaves quando tem mais de uma
            }
            N.setN(N.getN() - 1);
            if (N != this.raiz) {
                Balanceia_Folha(N);//Balanceia N
            }
        } else {//caso contrário(N nao é folha), substitui a chave por seu antecessor e balanceia a folha onde se encontrava o ancecessor
            No S = Antecessor(this.raiz, k);//S eh onde se encontra o antecessor de k
            int y = S.getChave().get(S.getN() - 1);//y é o antecessor de k
            S.setN(S.getN() - 1);
            N.getChave().set(i - 1, y);//substitui a chave por y
            Balanceia_Folha(S);//balanceia S
        }
        nElementos--;//decrementa o numero de elementos na arvoreB
    }
}
```

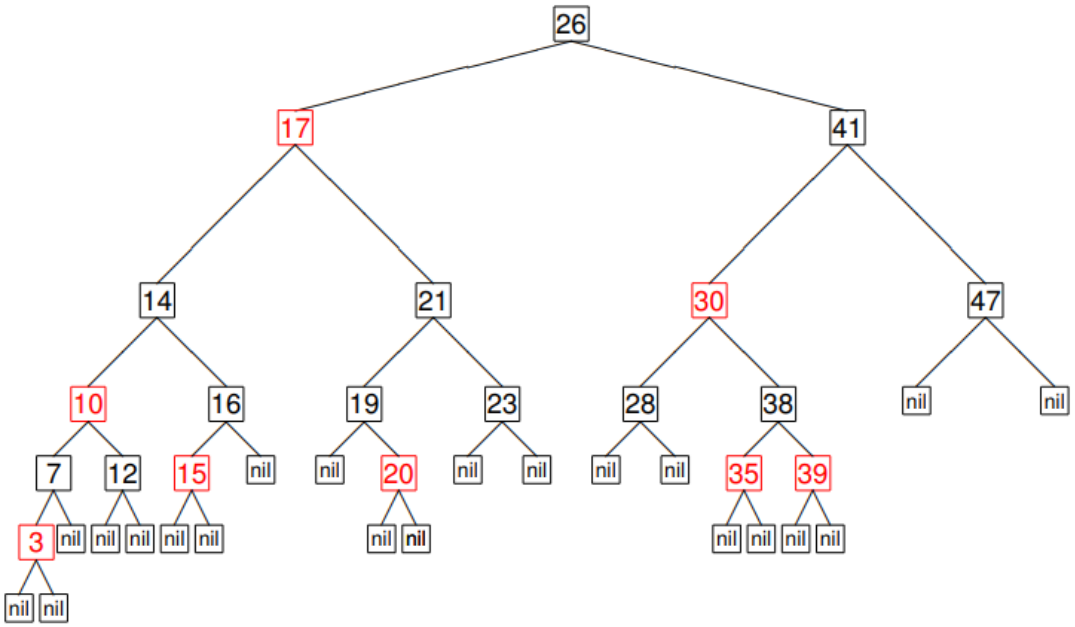
Árvore Rubro-Negra

Definição:

Uma árvore rubro-negra é uma árvore binária de busca auto balanceada, projetada para busca de dados na memória principal (RAM), em que cada nó possui os atributos abaixo:

- cor (1 bit): pode ser vermelho ou preto.
- key (e.g. inteiro): indica o valor de uma chave.
- left, right: ponteiros que apontam para a subárvore esquerda e direita, resp.
- pai: ponteiro que aponta para o nó pai. O campo pai do nó raiz aponta para nil.

Exemplo:



Uma árvore rubro-negra é uma árvore binária de busca, com algumas propriedades adicionais. Quando um nó não possui um filho (esquerdo ou direito), então vamos supor que ao invés de apontar para nil, ele

aponta para um nó fictício, que será uma folha da árvore. Assim, todos os nós internos contêm chaves e todas as folhas são nós fictícios.
As propriedades da árvore rubro-negra são:

1. Todo nó da árvore ou é vermelho ou é preto.
2. A raiz é **sempre** preta;
3. Toda folha (nil - leaf) é preta;
4. Se um nó é vermelho, então ambos os filhos são pretos.
5. Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos;
6. É uma BST auto balanceada.

Características:

A altura h de uma árvore rubro-negra de n chaves ou nós internos é no máximo 2 log(n + 1).

Complexidade:

Complexities in big O notation		
	Space complexity	
Space	$O(n)$	
	Time complexity	
Function	Amortized	Worst Case
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(1)$[2]	$O(\log n)$[1]
Delete	$O(1)$[2]	$O(\log n)$[1]

Diferenças entre RB e AVL

- AVL requer mais rotações para ser balanceada;
- Rubro negra: máximo de duas rotações para cada balanceamento;
- A busca é mais rápida em uma AVL, já que é estritamente balanceada;
- Inserção e deleção são mais rápidas na rubro negra, já que requer menos rotações.

Busca

1. Começando pelo nó raiz da árvore, verificar se o valor do nó a ser buscado é maior ou menor do que o nó atual
2. Se for menor, o nó atual passa a ser o filho esquerdo do nó antigo (nesse caso, o raiz)
3. Se for maior, o nó atual passa a ser o filho direito do nó antigo
4. Realizar esse procedimento até encontrar (ou não) o nó desejado
5. Retornar o valor booleano da busca

Código:

```
 private boolean searchNode(RedBlackNode node, int value) {
        boolean check = false;
        while ((node != nullNode) && check != true)
        {
            int nodeValue = node.element;
            if (value < nodeValue)
                node = node.leftChild;
            else if (value > nodeValue)
                node = node.rightChild;
            else
            {
                check = true;
                break;
            }
            check = searchNode(node, value);
        }
        return check;
    }
```

Inserção

1. Seja x e y nós raiz e folha da árvore, respectivamente
2. Checar se o nó raiz está vazio ou não. Se sim, o nó inserido será adicionado como nó raiz de cor preta
3. Se não, comparar o nó raiz com o novo nó. Se o novo nó for maior que a raiz, percorrer pela subárvore direita. Se não, pela esquerda
4. Repetir passo 3 até chegar na folha
5. Fazer o pai do nó raiz também ser pai do novo nó
6. Se o valor do nó folha for menor que o do novo nó, novo nó será filho esquerdo. Se for maior, filho direito
7. Fazer filhos do novo nó como sendo nulos
8. Novo nó será vermelho
9. Restaurar as propriedades da árvore performando rotações ou mudando as cores dos nós.

Algoritmo para manutenção das propriedades da árvore

1. Performar os passos até que o pai p do nó inserido seja vermelho
2. Se p for filho esquerdo do nó avô do nó inserido
   2.1. Caso 1
   - Quando a cor do filho direito do nó avô do nó inserido for vermelho, transforme a cor de ambos os filhos do avô para preto e faça avô ficar vermelho
   - Designar o nó avô para o nó inserido

   2.2. Caso 2
   - Se não, se o nó inserido for o filho direito do pai, designar p para o nó inserido e realizar rotação esquerda

   2.3. Caso 3
   - Transformar pai para preto e avô para vermelho e performar rotação direita para o nó inserido
3. Se p não for filho esquerdo do nó avô
   3.1. Se o filho esquerdo do avô for vermelho, transformar ambos os filhos do avô em preto e avô em vermelho

3.2. Designar avô para o nó inserido
3.3. Se não, se o nó inserido for filho esquerdo do pai, designar nó pai para o filho e performar rotação direita
3.4. Transformar nó pai para preto e avô para vermelho
3.5. Realizar rotação esquerda para nó avô
4. Fazer nó raiz preto

Remoção

Algoritmo para remoção de nó

1. Realizar a deleção padrão de árvore binária de busca. Fazendo isso, sempre é deletado um nó que ou é folha ou só tem um filho (se for interno, copia-se o sucessor e recursivamente chama a remoção para o sucessor, sendo o sucessor sempre folha ou nó com um filho). Então só é necessário cuidar de casos em que o nó é folha ou só tem um filho. Seja v o nó a ser deletado e u o filho que o substitui (u será nulo quando v for folha, e nulo é sempre preto)
2. Caso simples - Se ou u ou v são vermelho, deleta-se v e marca-se u como preto
3. Se ambos u e v forem preto

   3.1. Colorir u como double black. Agora deve-se converter double black em apenas black

   3.2. Fazer os seguintes passos enquanto u for double black e não for raiz. Seja s o irmão de u

   3.2.1 Se s for preto e um de seus filhos for vermelho, realizar rotação. Seja r o filho vermelho de s. Existem quatro possíveis alternativas dependendo das posições

   a) Caso esquerda esquerda - s é filho esquerdo de seu pai e r é filho direito de s ou ambos os filhos de s são vermelhos.

Aqui, o pai p substitui o nó deletado, nó irmão s se torna novo pai e o filho esquerdo r substitui s

b) Caso esquerda direita - s é filho esquerdo de seu pai e r é filho direito de s. Nessa situação, filho r substitui s e s se torna filho esquerdo de r. Após isso, r se torna pai de p e s.

c) Caso direita direita - s é filho direito de seu pai e r é filho direito de s ou ambos os filhos de s são vermelhos. Nesse caso, o pai p substitui o nó deletado, nó irmão s se torna novo pai e o filho direito r substitui s

d) Caso direita esquerda - s é filho direito de seu pai e r é filho esquerdo de s. Nesse caso, filho r substitui s e s se torna filho direito de r. Após isso, r se torna pai de p e s.

3.2.2 Se s for preto e ambos seus filhos são preto, recolorir e recorrer ao pai se ele for preto

3.2.3 Se s for vermelho, performar rotação para levantar o antigo irmão, recolorir ele e o pai. O novo irmão é sempre preto. Existem duas alternativas para esse caso

a) Caso esquerda - s é filho esquerdo do pai. Rotacionar para direita o pai p

b) Caso direita - s é filho direito do pai. Rotacionar para esquerda o pai p

3.3. Se u for raiz, colorí-lo para apenas black e retornar (altura de pretos da árvore reduz em 1).

Questões de Árvores:

1) Árvores B são frequentemente utilizadas para indexação de bancos de dados. Nesse contexto, analise as afirmativas a seguir sobre esse tipo de estrutura de dados.

I. São balanceadas.
II. Os nós podem ter mais de dois filhos.
III. A altura da árvore é O(lg(N)).
Está correto o que se afirma em

A) I, apenas.

B) I e II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta Correta: E

2) Uma Árvore-B de ordem m é uma árvore m-direcional tal que todas as folhas estão no mesmo nível. 2. Uma Árvore-B de ordem m é uma árvore m-direcional tal que todos os nós internos, com exceção da raiz, estão restritos a terem no máximo 2 filhos não vazios. 3. Uma Árvore-B de ordem m é uma árvore m-direcional tal que a raiz deve ter pelo menos m filhos não vazios.

Assinale a alternativa que indica todas as afirmativas **corretas**.

Alternativas

A É correta apenas a afirmativa 1.

B É correta apenas a afirmativa 2.

C São corretas apenas as afirmativas 1 e 2.

D São corretas apenas as afirmativas 1 e 3.

E São corretas apenas as afirmativas 2 e 3.

Resposta correta: B

3) Quanto aos conceitos de árvore binária, assinale a alternativa correta.
Alternativas

A Operações que utilizam recursão não podem ser realizadas sobre árvores binárias.

B A árvore pode ser vazia, isto é, não ter nenhum elemento.

C Uma árvore estritamente binária com *n* folhas tem *2n2* - 1 nós.

D A altura de um nó é o comprimento do menor caminho do nó até o seu primeiro descendente.

Resposta Correta: B

4) **[POSCOMP 2016 – FUNDATEC]** A operação de destruição de uma árvore requer um tipo de percurso em que a liberação de um nó é realizada apenas após todos os seus descendentes terem sido também liberados. Segundo essa descrição, a operação de destruição de uma árvore deve ser implementada utilizando o percurso

- A) em ordem.
- B) pré-ordem.
- C) central.
- D) simétrico.
- E) pós-ordem.

Resposta:

Considerando que a raiz das árvores contém o ponteiro para os dois nós associados (sub-árvore da esquerda e sub-árvore da direita) então é necessário visitar todas subárvores e liberando os nós folha e posteriormente os nós raiz. Ou seja, esse percurso deve deixar a raiz por ultimo, caracterizando o percurso pós-ordem

Resposta correta (E)

5) **[POSCOMP 2016 – FUNDATEC]** Uma árvore balanceada T que armazena n chaves é uma árvore binária de pesquisa na qual

- A) A diferença entre as alturas de suas subárvores permanece constante em todo o caso, após inserções ou remoções de chaves.
- B) As operações de inserção e remoção de chaves em nodos internos v de T seguem um padrão linear de tempo de execução.
- C) A propriedade da altura/balanceamento é determinada pela extensão do caminho mais curto entre um nodo interno v até o nodo raiz de T.
- D) A variação da altura dos nodos filhos de cada nodo interno v de T é de, no máximo, uma unidade.
- E) o tempo de execução para todas as operações fundamentais sobre cada nodo interno v de T se mantém constante.

Resposta:

Existem diversos tipos de árvores balanceadas (técnicas para manter o balanceamento). Uma árvore é balanceada se a diferença de altura entre duas sub-árvores difere em apenas 1 unidade.

5) **[POSCOMP 2010 – COPS – UEL]** Em uma Árvore B de ordem, temos que:

(i) cada nó contém no mínimo m registros (em+1 descendentes) e no máximo 2m registros (e 2m+1 descendentes), exceto o nó raiz que pode conter entre 1 e 2m registros;

(ii) todas os nós folha aparecem no mesmo nível.

Sobre Árvores B, é correto afirmar:

- a) O particionamento de nós em uma Árvore B ocorre quando um registro precisa ser inserido em um nó com 2m registros.
- b) O particionamento de nós em uma Árvore B ocorre quando um registro precisa ser inserido em um nó com menos de 2m registros.
- c) O particionamento de nós em uma Árvore B ocorre quando a chave do registro a ser inserido contém um valor(conteúdo) intermediário entre os valores das chaves dos registros contidos no mesmo nó.
- d) O particionamento de nós ocorre quando é necessário diminuir a altura da árvore.
- e) Em uma Árvore B, aumenta em um nível sua altura, toda vez que ocorre o particionamento de um nó.

Resposta:

Resposta correta (A)

6) **[POSCOMP 2009]** Quais das seguintes propriedades não se aplicam a árvores rubro-negras?

- A) Todo nó é vermelho ou preto.
- B) Todo nó folha é preto.
- C) Se um nó é preto, ambos seus filhos são vermelhos.
- D) Se um nó é vermelho, ambos seus filhos são negros.
- E) Todos os caminhos simples entre um nó e suas folhas descendentes contêm omesmo número de nós pretos.

Resposta:

Resposta correta (E)

7) Em uma árvore binária de busca balanceada do tipo AVL, as alturas das duas sub-árvores de um nó qualquer diferem em no máximo 1. A construção de uma árvore desse tipo, inicialmente vazia, por meio da inserção sucessiva de nós, utiliza uma certa operação para manter o balanceamento desejado quando necessário. Essa operação é
A) empilhamento.

B) desempilhamento.

C) concatenação.

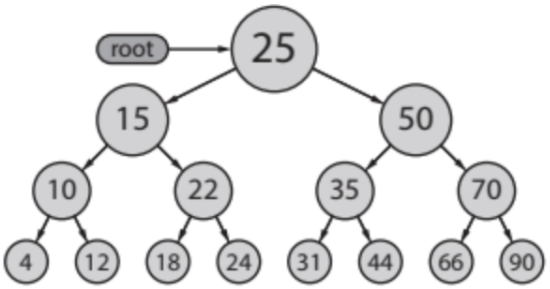
D) rotação.

E) poda.

Resposta:

Resposta correta (D)

Observe a figura abaixo que apresenta uma árvore.



Assinale a alternativa que apresenta sequência decorrente do percurso pré-ordem (*pre-order*) dessa árvore binária.

A) 90,70,66,50,35,44,31,25,24,22,1815,12,4,10

B) 4,10,12,15,18,22,24,25,31,35,44,50,66,70,90

C) 4,12,10,18,22,24,15,31,44,35,66,90,70,50,25

D) 4,12,18,24,31,44,66,90,10,22,35,70,15,50,25

E) 25,15,10,4,12,22,18,24,50,35,31,44,70,66,90

Responder

Parabéns! Você acertou!