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

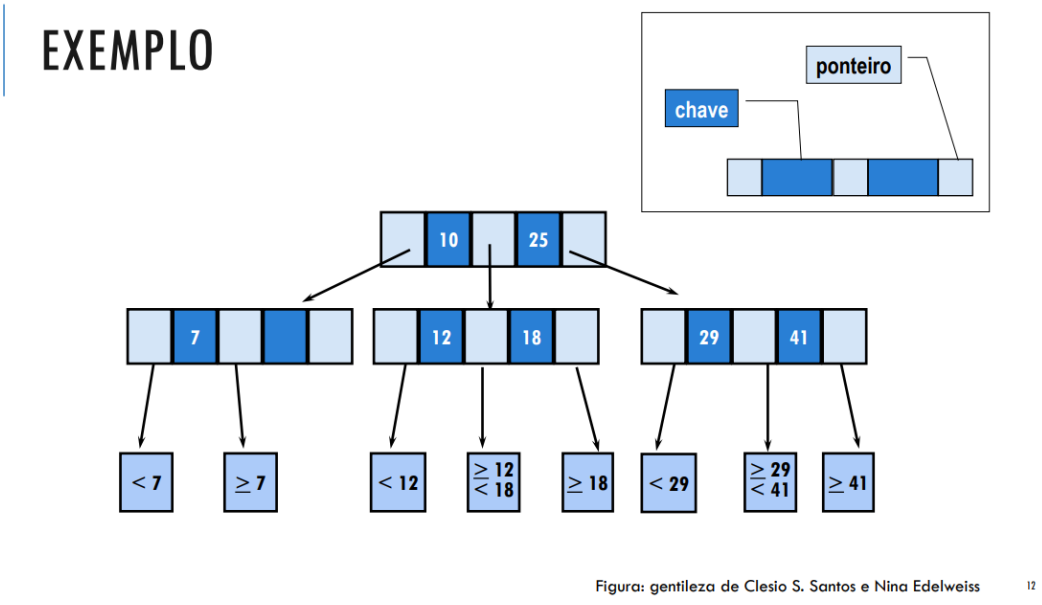
Características:

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

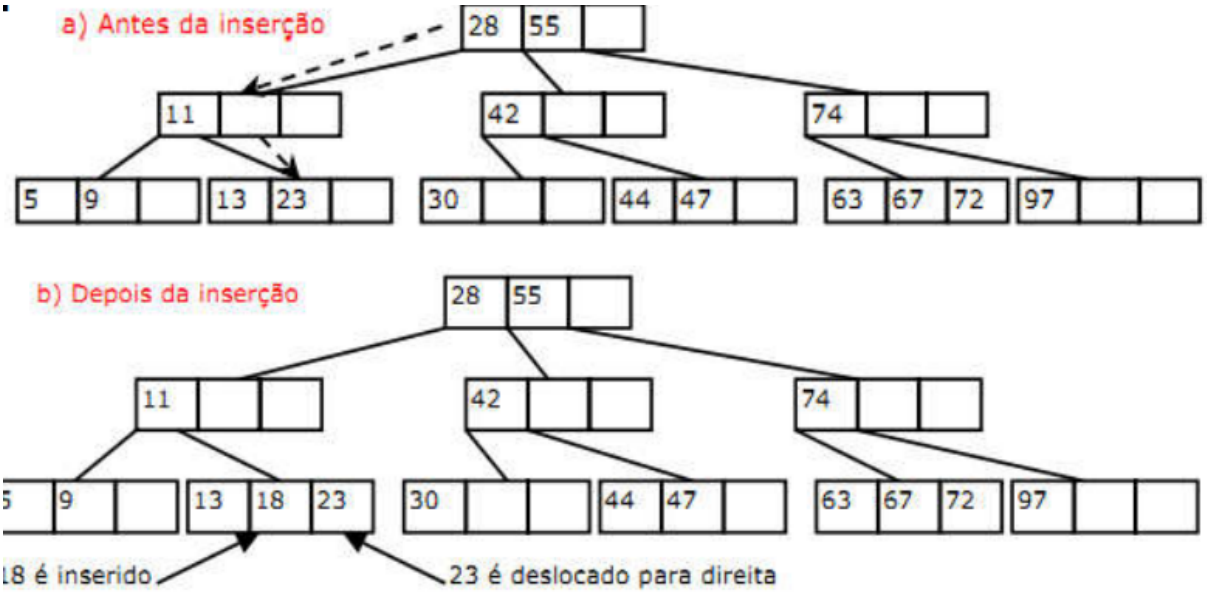
```
public BNodePosition<T> search(T element) {
        return searchAux(root, element);}
private BNodePosition<T> searchAux(BNode<T> node, T element) {
        int i = 0;
        BNodePosition<T> nodePosition = new BNodePosition<T>();
        while (i <= node.elements.size() && element.compareTo(node.elements.get(i))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



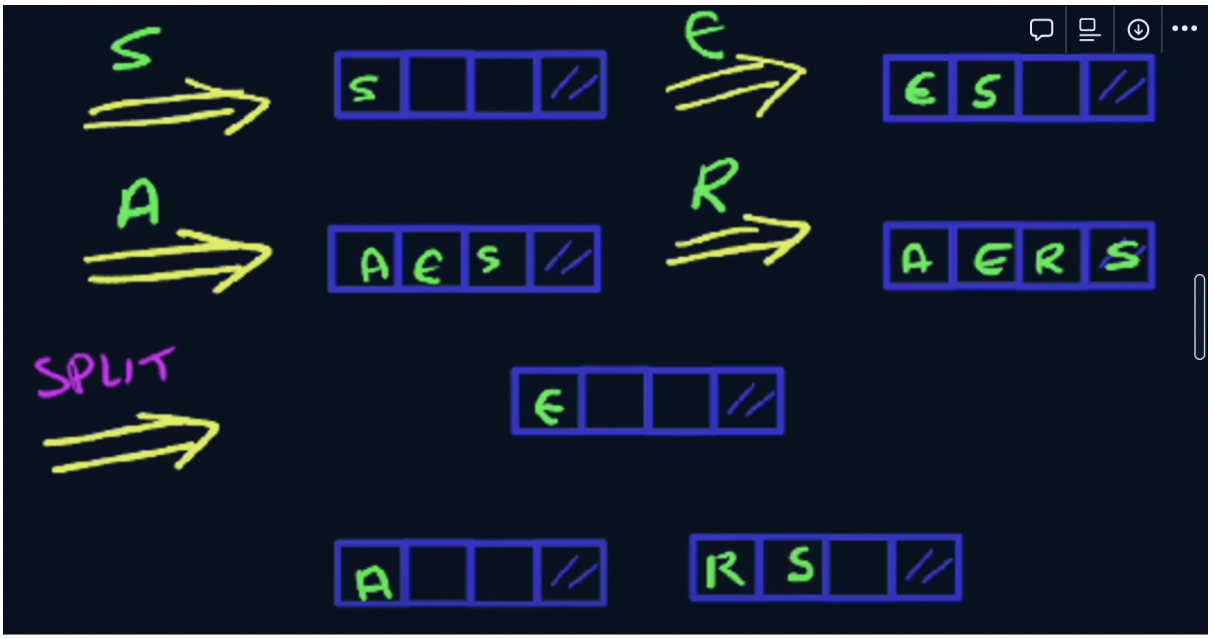
Algoritmo de Inserção em Java

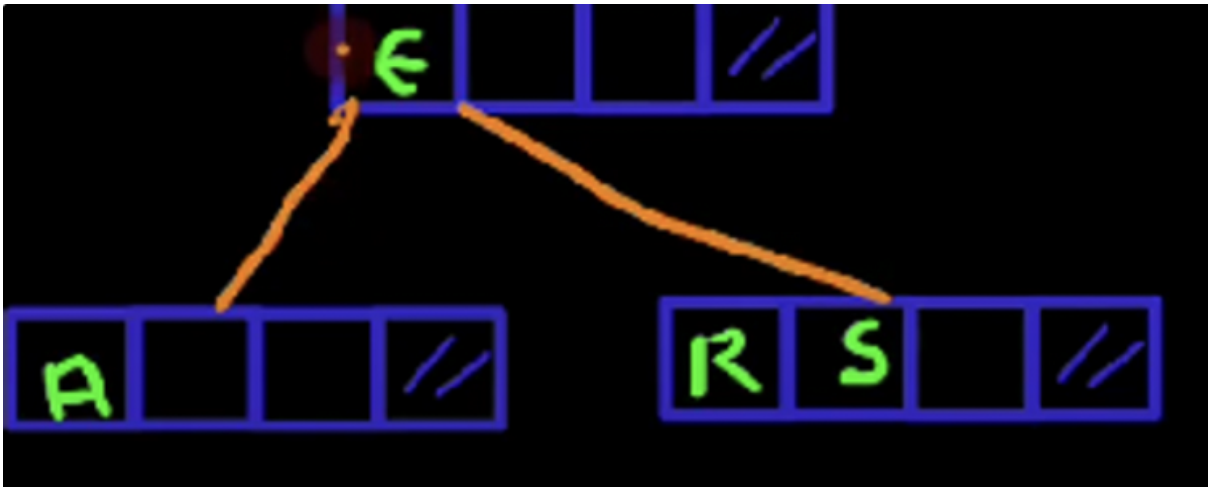
```java
//Metodo de Inserção na ArvoreB
//parametros: k - chave a ser inserida
public void insere(int k) {
    //Verifica se a chave a ser inserida existe
    if (BuscaChave(raiz, k) == null) { //só insere se não houver, para evitar duplicação de chaves
        //verifica se a chave está vazia
        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);
    }

}
```

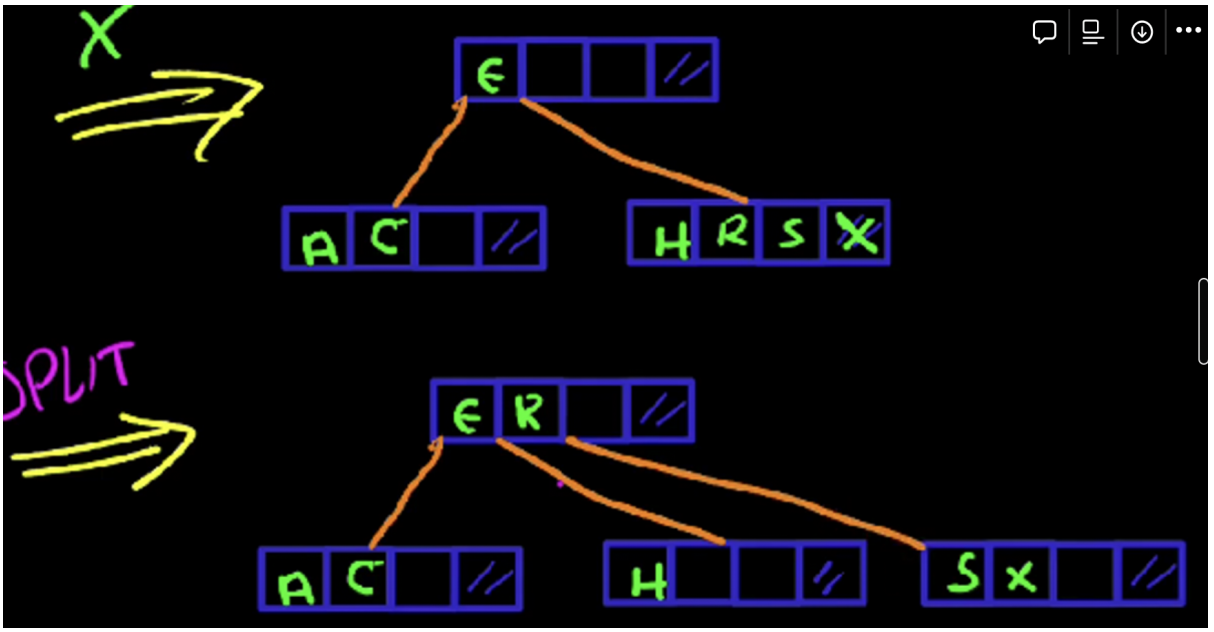
Em caso de página cheia:

- 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

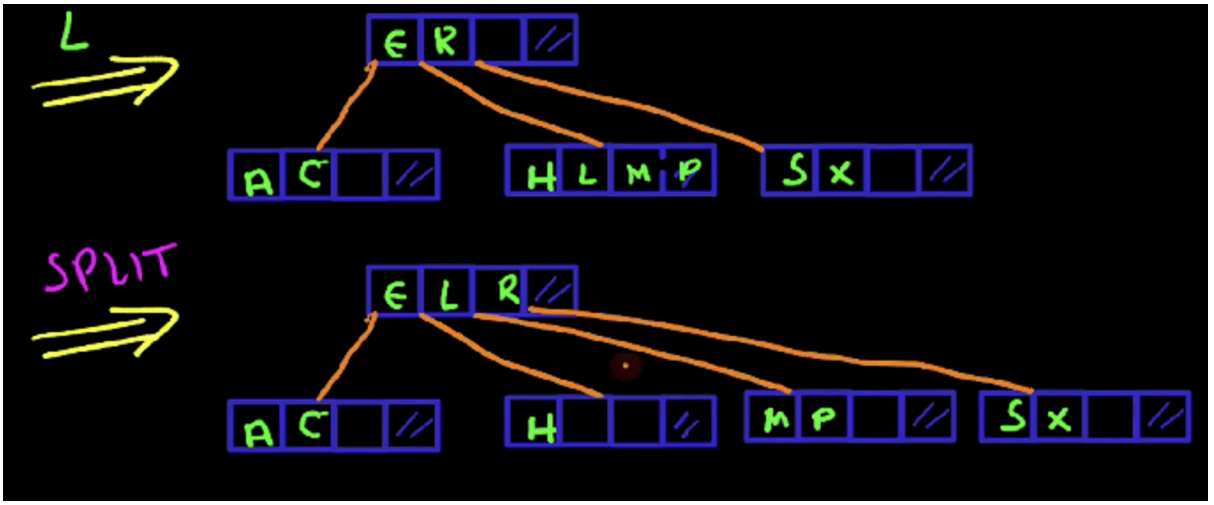

S
S
E
E S
A
A E S
R
A E R S
SPLIT
E
A
R S


E
A
R S

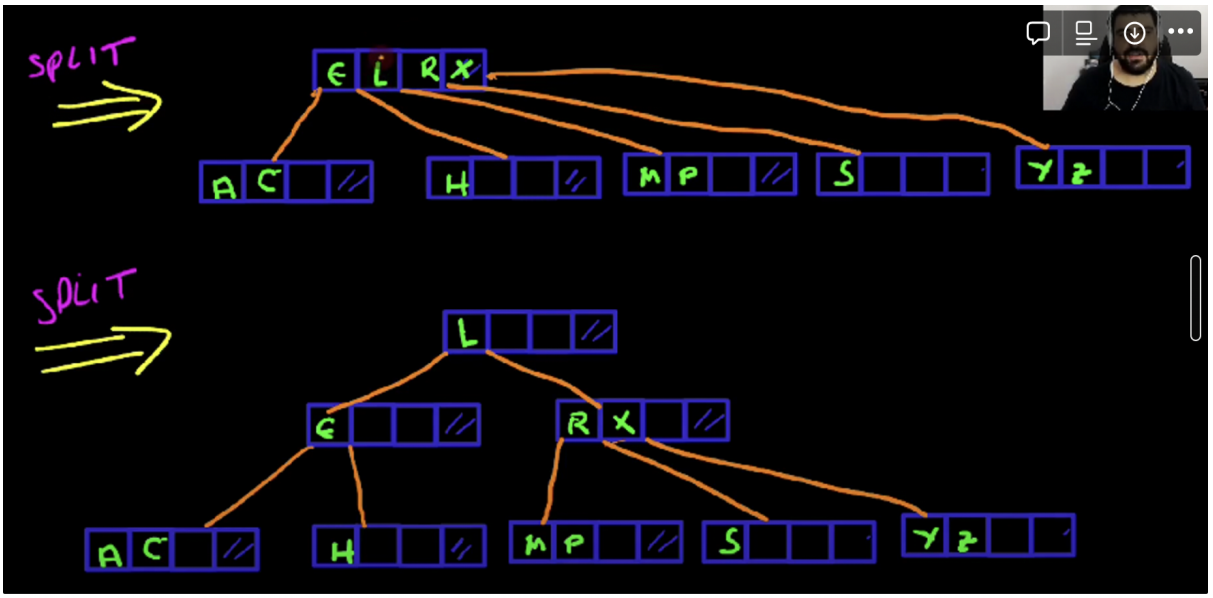
- Ex 2 :



- 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
    }
}
```