

Árvores B

Projeto de arquivos
Prof. Allan Rodrigo Leite

Árvores B

- Estrutura de árvore autobalanceada para dados classificados
 - Acesso sequencial
 - Acesso aleatório (pesquisa)
- Possui complexidade de tempo logarítmico
 - Acesso aleatório: $O(\log n)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
- Idealizada por Rudolf Bayer e Edward McCreight (1971)
 - Muito utilizada atualmente em banco de dados e sistemas de arquivos

Relação entre árvores binárias e árvores B

- Árvore binária
 - Recomendado para **classificação interna**
 - Trabalha com dados em memória principal
 - Estrutura de árvore autobalanceada para dados classificados
 - Possui complexidade de tempo logarítmico
- Árvore B
 - Recomendado para **classificação externa**
 - Trabalha com dados em memória secundária
 - Objetiva reduzir o acesso aos dispositivos de memória secundária
 - É uma generalização de árvores binárias
 - Porém, os nós podem ter mais de dois filhos

Conceitos da árvores B

- Nó da árvore
 - Armazena um conjunto não vazio de chaves
 - As chaves são armazenadas em ordem crescente
 - Os nós também são chamados de páginas
- Páginas
 - Refere-se à organização de dados em blocos
 - Visa reduzir o acesso de E/S a partir técnica de paginação de dados
 - Exemplos
 - Paginação de memória principal
 - Sistema de arquivos
 - Gerenciadores de bancos de dados

Conceitos da árvores B

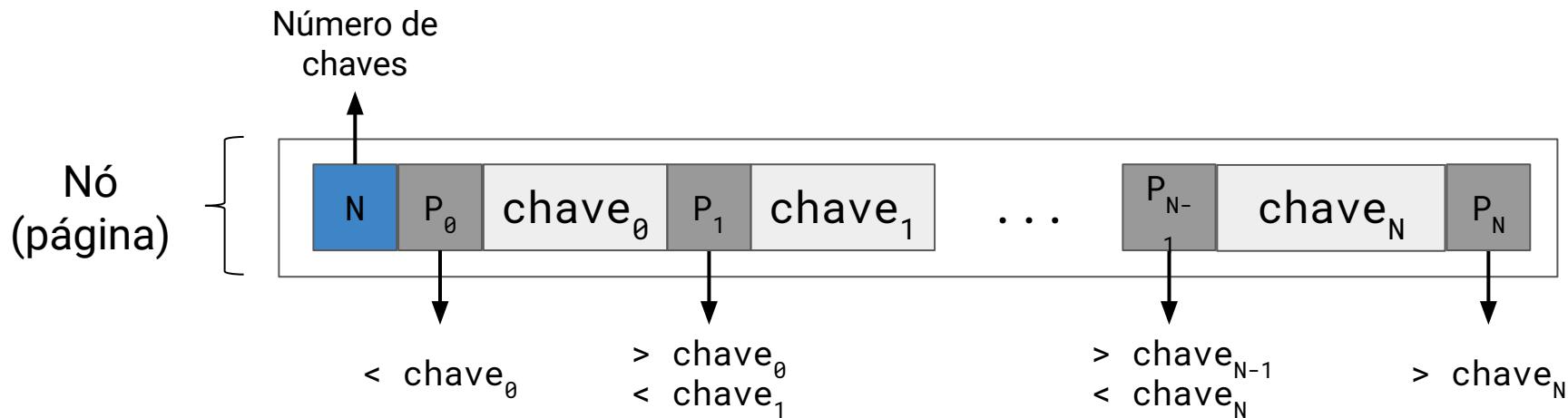
- Quantidade de chaves em um nó
 - Todo nó possui uma quantidade mínima e máxima de chaves
 - Exceto o nó raiz que não possui quantidade mínima
 - Para cada chave há um ponteiro a esquerda e direita para os nós filhos
 - A quantidade de filhos de um nó será o número de chaves mais um
- Ordem da árvore (Bayer e McCreight, 1972)
 - O número mínimo de chaves representa ordem da árvore
 - Todo nó (exceto a raiz) deve ter pelo menos 50% de ocupação
 - Nós intermediários e nós folhas devem seguir esta taxa de ocupação
 - Exemplo de árvore com ordem $M = 2$
 - Mínimo: 2 chaves
 - Máximo: 4 chaves

Conceitos da árvores B

- Propriedades da árvore
 - Todos os nós folhas estão no mesmo nível
 - A árvore cresce a partir da raiz
 - Uma árvore binária típica cresce a partir das folhas
 - As chaves de um nó devem estar ordenados de forma crescente
- Operações em uma árvore B
 - Adicionar uma chave
 - Remover uma chave
 - Localizar uma chave
 - Percorrer a árvore

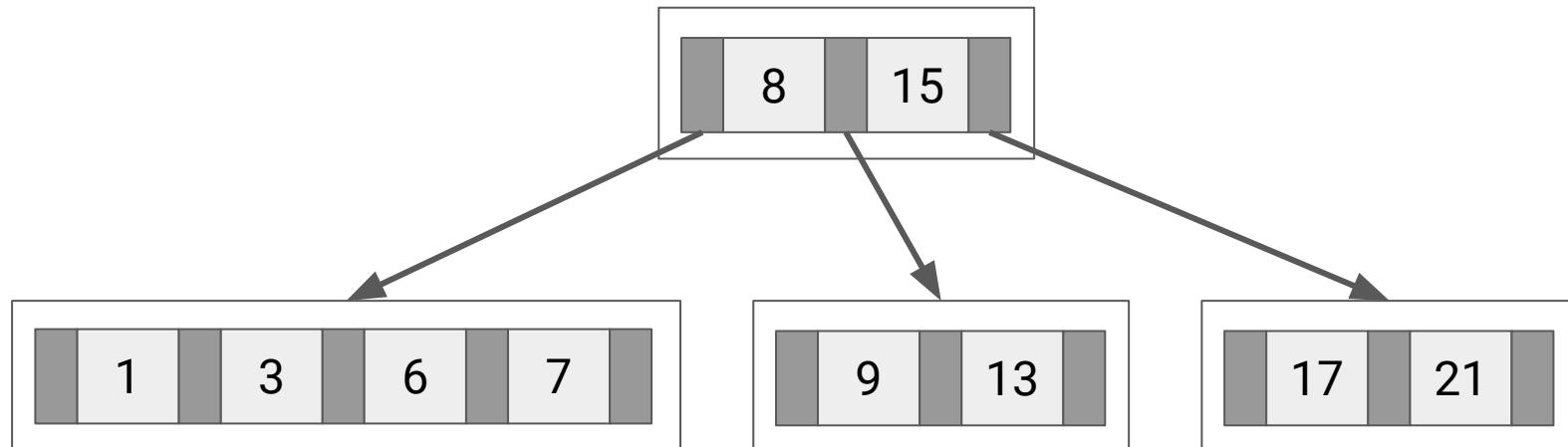
Estrutura da árvores B

- Estrutura do nó
 - Conjunto não vazio de chaves
 - Cada chave é precedida por um ponteiro para um nó filho
 - As chaves de registros são usualmente códigos numéricos



Estrutura da árvores B

- Estrutura da árvore
 - Nó raiz é a exceção para quantidade mínima de chaves
 - Os filhos de um nó é igual ao número de chaves mais um
 - Todos os nós folhas estão no mesmo nível da árvore



Estrutura da árvores B

- Estrutura da árvore

```
typedef struct no {
    int total;
    int* chaves;
    struct no** filhos;
    struct no* pai;
} No;
```

```
typedef struct arvoreB {
    No* raiz;
    int ordem;
} ArvoreB;
```

Operações em árvores B

- Criar uma árvore B com uma ordem predefinida

```
ArvoreB* criaArvore(int ordem) {
    ArvoreB* a = malloc(sizeof(ArvoreB));
    a->ordem = ordem;
    a->raiz = criaNo(a);

    return a;
}
```

Operações em árvores B

- Criar uma árvore B com uma ordem predefinida (cont.)

```
No* criaNo(ArvoreB* arvore) {
    int max = arvore->ordem * 2;
    No* no = malloc(sizeof(No));

    no->pai = NULL;

    no->chaves = malloc(sizeof(int) * (max + 1));
    no->filhos = malloc(sizeof(No) * (max + 2));
    no->total = 0;

    for (int i = 0; i < max + 2; i++)
        no->filhos[i] = NULL;

    return no;
}
```

Operações em árvores B

- Percorrer uma árvore a partir de um nó

```
void percorreArvore(No* no, void (visita)(int chave)) {
    if (no != NULL) {
        for (int i = 0; i < no->total; i++){
            percorreArvore(no->filhos[i], visita);

            visita(no->chaves[i]);
        }

        percorreArvore(no->filhos[no->total], visita);
    }
}
```

Operações em árvores B

- Localizar uma chave em uma árvore

```
int localizaChave(ArvoreB* arvore, int chave) {
    No *no = arvore->raiz;

    while (no != NULL) {
        int i = pesquisaBinaria(no, chave);

        if (i < no->total && no->chaves[i] == chave) {
            return 1; //encontrou
        } else {
            no = no->filhos[i];
        }
    }

    return 0; //não encontrou
}
```

Operações em árvores B

- Localizar uma chave em uma árvore (cont.)

```
int pesquisaBinaria(No* no, int chave) {
    int inicio = 0, fim = no->total - 1, meio;
    while (inicio <= fim) {
        meio = (inicio + fim) / 2;
        if (no->chaves[meio] == chave) {
            return meio; //encontrou
        } else if (no->chaves[meio] > chave) {
            fim = meio - 1;
        } else {
            inicio = meio + 1;
        }
    }
    return inicio; //não encontrou
}
```

Operações em árvores B

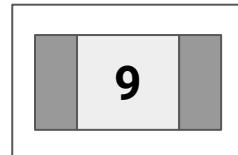
- Adicionar novas chaves
 - Inicia localizando o nó que deverá ser adicionada a chave
 - Use uma busca tradicional, porém, retorna o nó ao invés da chave
 - Insere a chave no nó localizado anteriormente
 - Deve prever os limites de chaves em um nó
 - Transbordo ou *overflow*: número de chaves excede a capacidade
 - Se ocorrer transbordo, deve ser realizado uma divisão do nó
 - Divisão ou *split*: separa as chaves contidas em um nó em dois
 - Cada nó (atual e o novo) possuirão 50% de ocupação
 - Após a divisão, a chave central da divisão deve ser promovida
 - Esta adição da chave promovida deve ser recursiva
 - Caso a divisão ocorrer na raiz, será adicionado um novo nível na árvore

Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, 11, 7, 8, 5, 1

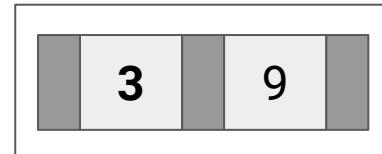
Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: **9**, 3, 11, 7, 8, 5, 1



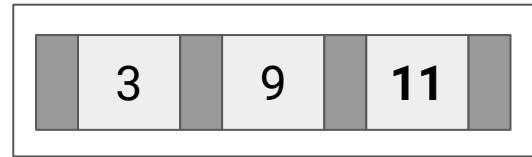
Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, **3**, 11, 7, 8, 5, 1



Estrutura da árvores B

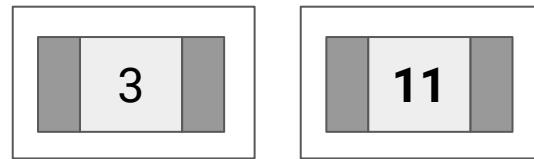
- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, **11**, 7, 8, 5, 1



Overflow

Estrutura da árvores B

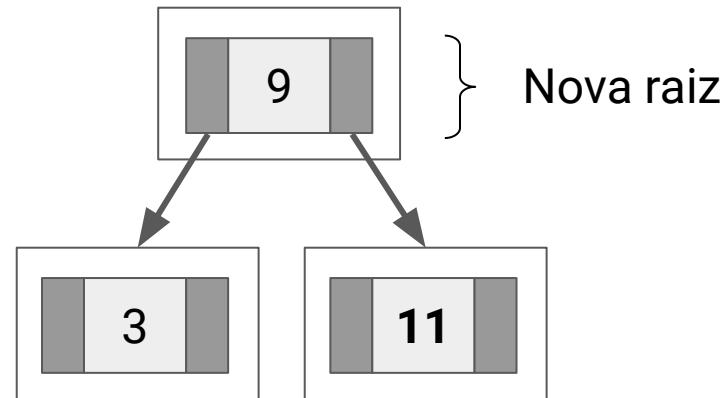
- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, **11**, 7, 8, 5, 1



Split

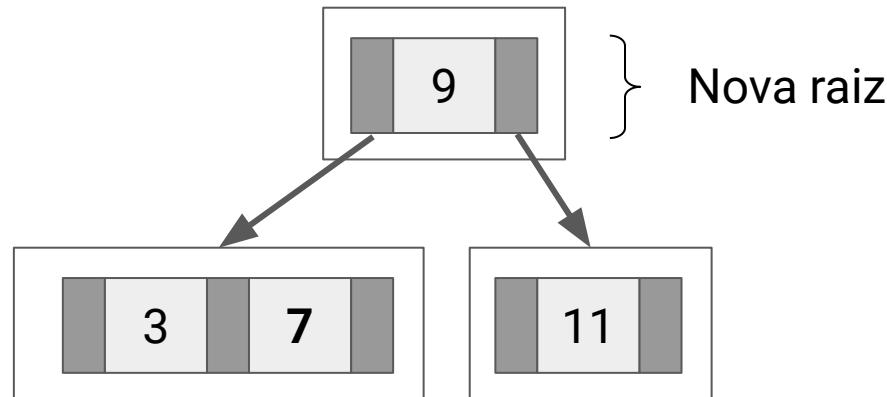
Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, **11**, 7, 8, 5, 1



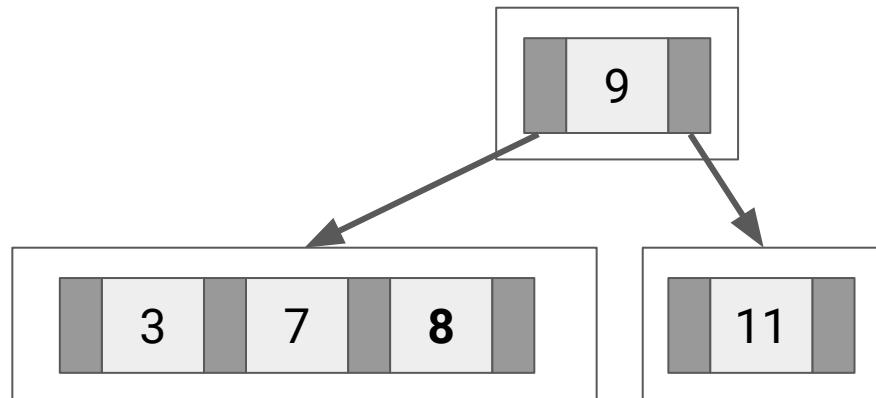
Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, 11, 7, 8, 5, 1



Estrutura da árvores B

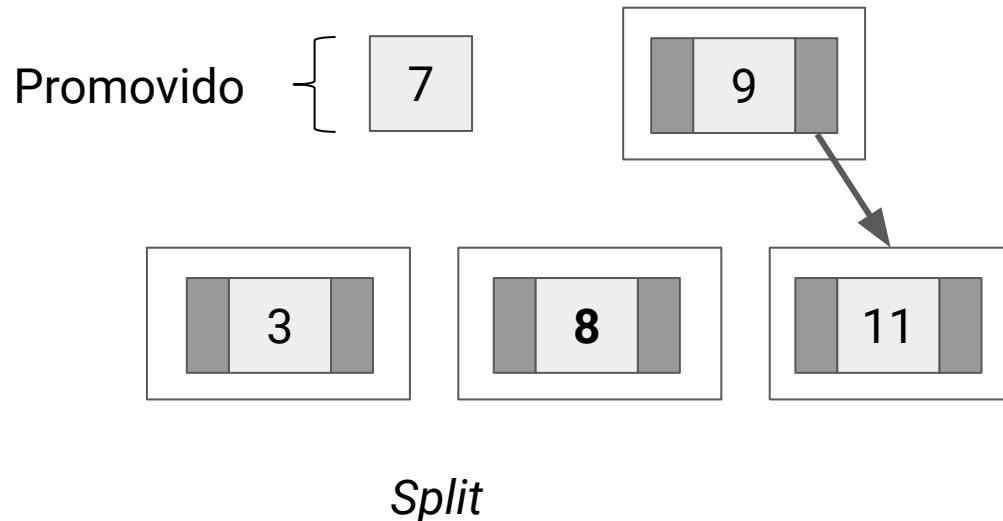
- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, 11, 7, **8**, 5, 1



Overflow

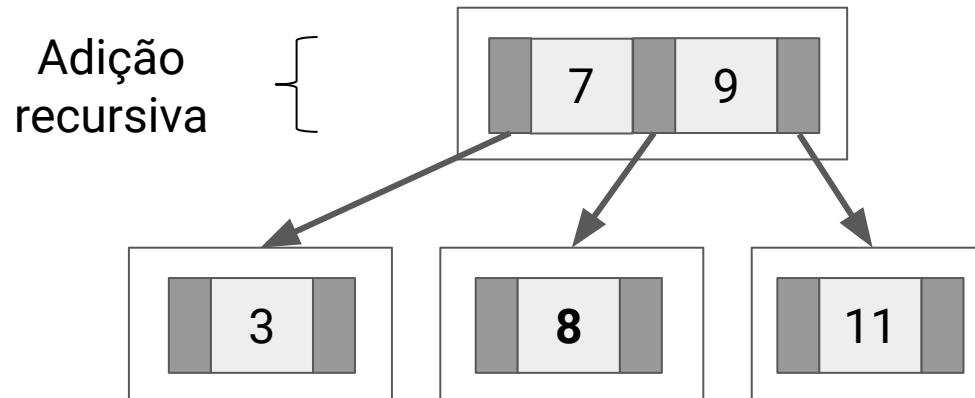
Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, 11, 7, **8**, 5, 1



Estrutura da árvores B

- Exemplo de adição de chaves em uma arvore de ordem $M = 1$
 - Chaves: 9, 3, 11, 7, **8**, 5, 1



Operações em árvores B

- Localizar um nó a partir de uma chave

```
No* localizaNo(ArvoreB* arvore, int chave) {  
    No *no = arvore->raiz;  
  
    while (no != NULL) {  
        int i = pesquisaBinaria(no, chave);  
  
        if (no->filhos[i] == NULL)  
            return no; //encontrou nó  
        else  
            no = no->filhos[i];  
    }  
  
    return NULL; //não encontrou nenhum nó  
}
```

Operações em árvores B

- Adicionar chave em um nó

```
void adicionaChaveNo(No* no, No* novo, int chave) {  
    int i = pesquisaBinaria(no, chave);  
  
    for (int j = no->total - 1; j >= i; j--) {  
        no->chaves[j + 1] = no->chaves[j];  
        no->filhos[j + 2] = no->filhos[j + 1];  
    }  
  
    no->chaves[i] = chave;  
    no->filhos[i + 1] = novo;  
  
    no->total++;  
}
```

Operações em árvores B

- Verificar se houve transbordo (*overflow*) em um nó

```
int transbordo(ArvoreB *arvore, No *no) {
    return no->total > arvore->ordem * 2;
}
```

Operações em árvores B

- Dividir chaves (*split*) de um nó em um novo nó

```
No* divideNo(ArvoreB* arvore, No* no) {
    int meio = no->total / 2;
    No* novo = criaNo(arvore);
    novo->pai = no->pai;

    for (int i = meio + 1; i < no->total; i++) {
        novo->filhos[novo->total] = no->filhos[i];
        novo->chaves[novo->total] = no->chaves[i];
        if (novo->filhos[novo->total] != NULL) novo->filhos[novo->total]->pai = novo;

        novo->total++;
    }

    novo->filhos[novo->total] = no->filhos[no->total];
    if (novo->filhos[novo->total] != NULL) novo->filhos[novo->total]->pai = novo;
    no->total = meio;
    return novo;
}
```

Operações em árvores B

- Adicionar nova chave na árvore

```
void adicionaChave(ArvoreB* arvore, int chave) {  
    No* no = localizaNo(arvore, chave);  
  
    adicionaChaveRecursivo(arvore, no, NULL, chave);  
}
```

Operações em árvores B

- Adicionar nova chave na árvore (cont.)

```
void adicionaChaveRecursivo(ArvoreB* arvore, No* no, No* novo, int chave) {  
    adicionaChaveNo(no, novo, chave);  
    if (transbordo(arvore, no)) {  
        int promovido = no->chaves[arvore->ordem];  
        No* novo = divideNo(arvore, no);  
  
        if (no->pai == NULL) {  
            No* pai = criaNo(arvore);  
            pai->filhos[0] = no;  
            adicionaChaveNo(pai, novo, promovido);  
  
            no->pai = pai;  
            novo->pai = pai;  
            arvore->raiz = pai;  
        } else  
            adicionaChaveRecursivo(arvore, no->pai, novo, promovido);  
    }  
}
```

Árvores B

Projeto de arquivos
Prof. Allan Rodrigo Leite