Árvore B

Estrutura de Dados Avançada — QXD0015

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

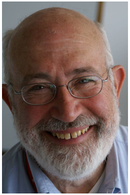
Universidade Federal do Ceará

1º semestre/2025

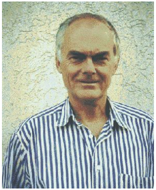
Árvore B

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Generalização das árvores de busca balanceadas
  - Otimizada para acesso a grandes volumes de dados.
- Projetada para acessar dados em memória secundária.
  - Discos magnéticos: HD
- Muito utilizadas em SGBDs, relacionais ou não.
- Inventada por Rudolf Bayer e Edward Meyers McCreight em 1971 enquanto trabalhavam no Boeing Scientific Research Labs.



Edward Meyers



Rudolf Bayer

Hierarquia de Memória

A memória do computador é dividida em uma hierarquia:

- SSD (Solid-State Drive) ou HDD (Hard Disk Drive)
    - Memória permanente, onde gravamos arquivos
    - Chamada de memória secundária

Hierarquia de Memória

A memória do computador é dividida em uma hierarquia:

- SSD (Solid-State Drive) ou HDD (Hard Disk Drive)
  - Memória permanente, onde gravamos arquivos
  - Chamada de memória secundária
- RAM (Random-Access Memory)
  - Onde são armazenados os programas em execução
    - e a memória alocada pelos mesmos
  - Memória volátil, é apagada se o computador é desligado

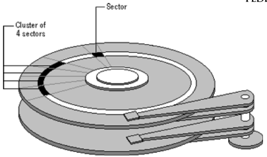
Hierarquia de Memória

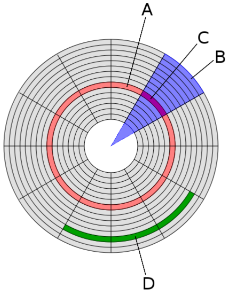
A memória do computador é dividida em uma hierarquia:

- SSD (Solid-State Drive) ou HDD (Hard Disk Drive)
  - Memória permanente, onde gravamos arquivos
  - Chamada de memória secundária
- RAM (Random-Access Memory)
  - Onde são armazenados os programas em execução
    - e a memória alocada pelos mesmos
  - Memória volátil, é apagada se o computador é desligado
- Memória Cache
  - Muito próxima do processador para ter acesso rápido
  - A informação é copiada da RAM para a Cache

Discos (HDD)

- Tecnologia barata
- Alta capacidade de armazenamento
- Informações armazenadas em trilhas
- Trilhas são divididas em setores
- Aplicações sempre acessam o disco em unidades de blocos (páginas)
  - Exemplo: 512 bytes a 4096 bytes
  - Acesso a disco é muito custoso
- Se a página está na memória, podemos acessá-la. Se não está, precisamos lê-la na memória secundária.





(A) Trilha
(B) Setor geométrico
(C) Setor de trilha
(D) Bloco

Problemas

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Acesso a dados em disco é lento (ordens de magnitude mais devagar que na memória principal — milissegundos versus nanosegundos).
  - Acesso custoso.
- Quantidade de dados manipulados não cabe na memória principal
- **Ideia:** Manter certas páginas na memória secundária e trazer apenas um número constante delas para a memória principal.
  - O acesso a dados no HD é otimizado para trazer múltiplas páginas por vez.
  - É preciso uma estrutura de dados que armazene múltiplos dados de uma única vez e seja eficiente.

Árvore B

Árvore D-ária de Busca

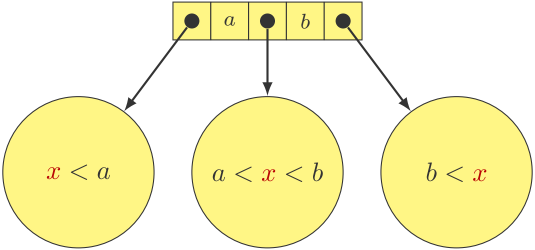
Podemos generalizar árvores binárias de busca

- Exemplo: árvores ternárias de busca
  - Nó pode ter 0, 1, 2 ou 3 filhos

Árvore D-ária de Busca

Podemos generalizar árvores binárias de busca

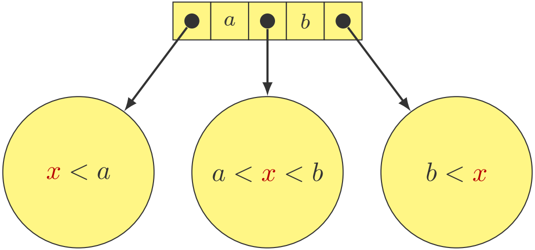
- Exemplo: árvores ternárias de busca
  - Nó pode ter 0, 1, 2 ou 3 filhos



Árvore D-ária de Busca

Podemos generalizar árvores binárias de busca

- Exemplo: árvores ternárias de busca
  - Nó pode ter 0, 1, 2 ou 3 filhos



Como fazer busca nesse tipo de árvore?

Árvore B — Definição

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

Árvore B — Definição

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

(1) A raiz $T.root$ é uma folha ou tem no mínimo dois filhos.

Árvore B — Definição

Universidade Federal do Ceará
Campus Quixadá

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

(1) A raiz $T.root$ é uma folha ou tem no mínimo dois filhos.

(2) Cada nó diferente da raiz possui no mínimo $D - 1$ chaves e, portanto, pelo menos D filhos.

Árvore B — Definição

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

(1) A raiz $T.root$ é uma folha ou tem no mínimo dois filhos.

(2) Cada nó diferente da raiz possui no mínimo $D-1$ chaves e, portanto, pelo menos D filhos.

(3) Cada nó tem no máximo $2D-1$ chaves e, portanto, no máximo $2D$ filhos. Dizemos que um nó está cheio se ele tem exatamente $2D-1$ chaves.

Árvore B — Definição

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

(1) A raiz $T.root$ é uma folha ou tem no mínimo dois filhos.

(2) Cada nó diferente da raiz possui no mínimo $D - 1$ chaves e, portanto, pelo menos D filhos.

(3) Cada nó tem no máximo $2D - 1$ chaves e, portanto, no máximo $2D$ filhos. Dizemos que um nó está cheio se ele tem exatamente $2D - 1$ chaves.

(4) Todas as folhas estão no mesmo nível.

Árvore B — Definição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Seja $D \geq 2$ um número natural.

Árvores B são árvores D-árias de busca com as 4 propriedades a seguir:

(1) A raiz $T.root$ é uma folha ou tem no mínimo dois filhos.

(2) Cada nó diferente da raiz possui no mínimo $D - 1$ chaves e, portanto, pelo menos D filhos.

(3) Cada nó tem no máximo $2D - 1$ chaves e, portanto, no máximo $2D$ filhos. Dizemos que um nó está cheio se ele tem exatamente $2D - 1$ chaves.

(4) Todas as folhas estão no mesmo nível.

- O número D é chamado de grau mínimo da árvore B.
- A estrutura apresentada satisfaz ainda as 3 propriedades a seguir:

Árvore B – Definição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x

Árvore B – Definição

1. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$

Árvore B – Definição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$
   - $x.leaf$ é um booleano que indica se x é uma folha ou não.

Árvore B – Definição

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$
   - $x.leaf$ é um booleano que indica se x é uma folha ou não.

II. Cada nó interno x contém $x.n + 1$ ponteiros $x.c[1] \ldots x.c[x.n + 1]$

Árvore B – Definição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$
   - $x.leaf$ é um booleano que indica se x é uma folha ou não.

II. Cada nó interno x contém $x.n + 1$ ponteiros $x.c[1] \ldots x.c[x.n + 1]$
   - $x.c[i]$ é o ponteiro para o i-ésimo filho

Árvore B – Definição

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$
   - $x.leaf$ é um booleano que indica se x é uma folha ou não.

II. Cada nó interno x contém $x.n + 1$ ponteiros $x.c[1] \ldots x.c[x.n + 1]$
   - $x.c[i]$ é o ponteiro para o i-ésimo filho
   - Nós folhas não têm filhos. Logo, os ponteiros $x.c[i]$ das folhas são nulos ou indefinidos.

Árvore B – Definição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

I. Cada nó x tem os seguintes campos:
   - $x.n$ é o número de chaves armazenadas em x
   - $x.key[i]$ é i-ésima chave armazenada
     - $x.key[1] \leq x.key[2] \leq \cdots \leq x.key[x.n]$
   - $x.leaf$ é um booleano que indica se x é uma folha ou não.

II. Cada nó interno x contém $x.n+1$ ponteiros $x.c[1] \ldots x.c[x.n+1]$
   - $x.c[i]$ é o ponteiro para o i-ésimo filho
   - Nós folhas não têm filhos. Logo, os ponteiros $x.c[i]$ das folhas são nulos ou indefinidos.

III. se a chave k_i está na subárvore $x.c[i]$, então

$$k_1 \leq x.key[1] \leq k_2 \leq x.key[2] \leq \cdots \leq x.key[x.n] \leq k_{x.n+1}$$

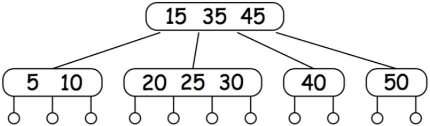
Exemplo de árvore B

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A árvore B mais simples ocorre quando $D = 2$. Todo nó interno nesta árvore tem 2,3 ou 4 filhos.

Exemplo de árvore B

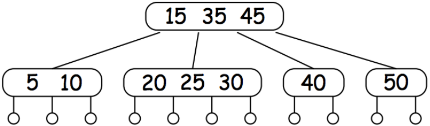
- A árvore B mais simples ocorre quando $D = 2$. Todo nó interno nesta árvore tem 2,3 ou 4 filhos.
- Esse tipo de árvore é chamada de árvore 2-3-4.



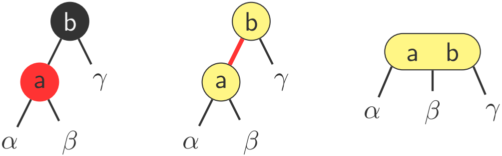
Exemplo de árvore B

- A árvore B mais simples ocorre quando $D = 2$. Todo nó interno nesta árvore tem 2,3 ou 4 filhos.

- Esse tipo de árvore é chamada de árvore 2-3-4.



- São "equivalentes" às árvores rubro-negras



Escolhendo o grau mínimo D

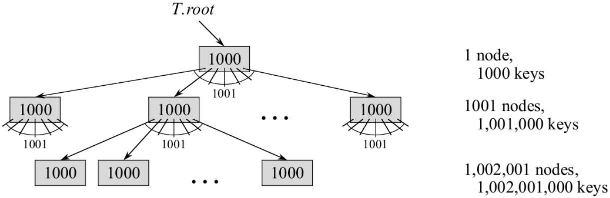
Queremos que um nó caiba em uma página do disco, mas não queremos utilizar mal a página do disco.

Escolha D máximo tal que um nó com $2D$ filhos caiba na página.

Escolhendo o grau mínimo D

Queremos que um nó caiba em uma página do disco, mas não queremos utilizar mal a página do disco.
Escolha D máximo tal que um nó com $2D$ filhos caiba na página.

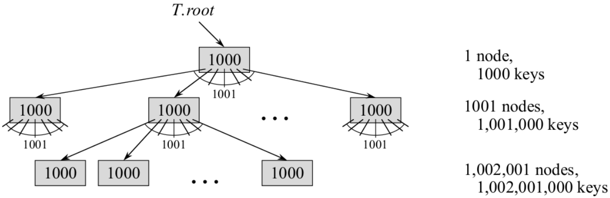
- **Exemplo:** numa árvore com 1000 chaves em cada nó e com altura $h = 2$, armazenamos até 10^9 chaves (1 bilhão).



Escolhendo o grau mínimo D

Queremos que um nó caiba em uma página do disco, mas não queremos utilizar mal a página do disco.
Escolha D máximo tal que um nó com $2D$ filhos caiba na página.

- **Exemplo:** numa árvore com 1000 chaves em cada nó e com altura $h = 2$, armazenamos até 10^9 chaves (1 bilhão).



Consideramos que o registro está junto com a chave. Ou então temos um ponteiro para o registro

Balanceamento

Altura de uma árvore B

Teorema: Seja $n \geq 1$ o número total de chaves de uma árvore B com altura h e grau mínimo $D \geq 2$. Então,

$$h \leq \log_D \left(\frac{n+1}{2} \right) + 1.$$

Altura de uma árvore B

Teorema: Seja $n \geq 1$ o número total de chaves de uma árvore B com altura h e grau mínimo $D \geq 2$. Então,

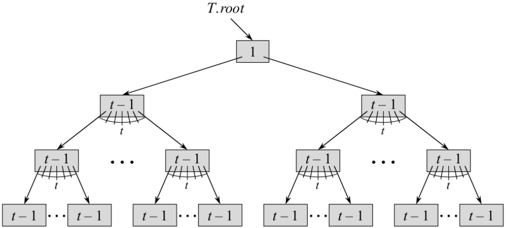
$$h \leq \log_D \left(\frac{n+1}{2} \right) + 1.$$

Demonstração:

- Seja T uma árvore B. Então, sua raiz contém pelo menos uma chave e todos os outros nós contém pelo menos $D - 1$ chaves.

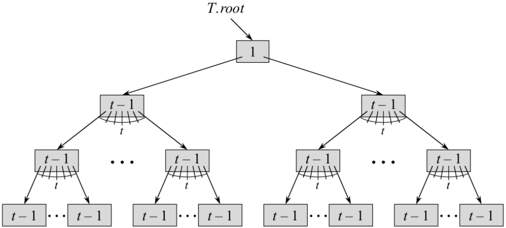
Continuação da demonstração

- Assim, T, cuja altura é h, contém 1 nó no nível 1, pelo menos 2 nós no nível 2, pelo menos $2D$ nós no nível 3, pelo menos $2D^2$ nós no nível 4, e assim por diante, até que no nível h ela tem pelo menos $2D^{h-2}$ nós.



Continuação da demonstração

- Assim, T, cuja altura é h, contém 1 nó no nível 1, pelo menos 2 nós no nível 2, pelo menos $2D$ nós no nível 3, pelo menos $2D^2$ nós no nível 4, e assim por diante, até que no nível h ela tem pelo menos $2D^{h-2}$ nós.



- Assim, o número de chaves n de T satisfaz a desigualdade:

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} \tag{1}$$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$1 + 2(D-1)\sum_{i=2}^{h} D^{i-2} = 1 + 2(D-1)\left(\frac{D^{h-1}-1}{D-1}\right) = 1 + 2D^{h-1} - 2.$$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$1 + 2(D-1)\sum_{i=2}^{h} D^{i-2} = 1 + 2(D-1)\left(\frac{D^{h-1}-1}{D-1}\right) = 1 + 2D^{h-1} - 2.$$

Logo, $n \geq 2D^{h-1} - 1$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$1 + 2(D-1)\sum_{i=2}^{h} D^{i-2} = 1 + 2(D-1)\left(\frac{D^{h-1}-1}{D-1}\right) = 1 + 2D^{h-1} - 2.$$

Logo, $n \geq 2D^{h-1} - 1 \implies D^{h-1} \leq \frac{n+1}{2}$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$1 + 2(D-1)\sum_{i=2}^{h} D^{i-2} = 1 + 2(D-1)\left(\frac{D^{h-1}-1}{D-1}\right) = 1 + 2D^{h-1} - 2.$$

Logo, $n \geq 2D^{h-1} - 1 \implies D^{h-1} \leq \frac{n+1}{2} \implies h - 1 \leq \log_D \frac{n+1}{2}$

Continuação da demonstração

$$n \geq 1 + (D-1)\sum_{i=2}^{h} 2D^{i-2} = 1 + 2(D-1)\sum_{i=2}^{h} D^{i-2}$$

Vamos usar a fórmula: $$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$1 + 2(D-1)\sum_{i=2}^{h} D^{i-2} = 1 + 2(D-1)\left(\frac{D^{h-1}-1}{D-1}\right) = 1 + 2D^{h-1} - 2.$$

Logo, $n \geq 2D^{h-1} - 1 \implies D^{h-1} \leq \frac{n+1}{2} \implies h - 1 \leq \log_D \frac{n+1}{2} \implies h \leq \log_D \frac{n+1}{2} + 1.$ ■

Busca na árvore B

Leitura e escrita de páginas

Se x é ponteiro para um objeto na memória secundária

Leitura e escrita de páginas

Se x é ponteiro para um objeto na memória secundária

- $\text{DISK-READ}(x)$: lê x da memória secundária

Leitura e escrita de páginas

Se x é ponteiro para um objeto na memória secundária

- $\text{DISK-READ}(x)$: lê x da memória secundária
- $\text{DISK-WRITE}(x)$: grava x na memória secundária

Leitura e escrita de páginas

Se x é ponteiro para um objeto na memória secundária

- $\text{DISK-READ}(x)$: lê x da memória secundária
- $\text{DISK-WRITE}(x)$: grava x na memória secundária

Adotamos duas convenções nos pseudocódigos das funções que veremos:

Leitura e escrita de páginas

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Se x é ponteiro para um objeto na memória secundária

- $\text{DISK-READ}(x)$: lê x da memória secundária
- $\text{DISK-WRITE}(x)$: grava x na memória secundária

Adotamos duas convenções nos pseudocódigos das funções que veremos:

- A raiz da árvore B está sempre na memória principal, de modo que não precisamos realizar um $\text{DISK-READ}(x)$ na raiz. No entanto, devemos realizar um $\text{DISK-WRITE}(x)$ da raiz sempre que a raiz for modificada.

Leitura e escrita de páginas

Se x é ponteiro para um objeto na memória secundária

- $\text{DISK-READ}(x)$: lê x da memória secundária
- $\text{DISK-WRITE}(x)$: grava x na memória secundária

Adotamos duas convenções nos pseudocódigos das funções que veremos:

- A raiz da árvore B está sempre na memória principal, de modo que não precisamos realizar um $\text{DISK-READ}(x)$ na raiz. No entanto, devemos realizar um $\text{DISK-WRITE}(x)$ da raiz sempre que a raiz for modificada.

- Quaisquer nós que forem passados como parâmetro devem ter sido obtidos anteriormente por uma chamada a $\text{DISK-READ}(x)$.

Busca na Árvore B

Para procurar a chave k em um nó x:

Busca na Árvore B

Para procurar a chave k em um nó x:

- Basta verificar se a chave está em x. Se estiver, retorna o par (x, i), contendo o nó x e o índice i tal que $x.key[i] = k$.

Busca na Árvore B

Para procurar a chave k em um nó x:

- Basta verificar se a chave está em x. Se estiver, retorna o par (x, i), contendo o nó x e o índice i tal que $x.key[i] = k$.
- Se não estiver em x e o nó x não for folha, basta buscar no filho correto. Se o nó x for folha, então retorna `NIL`.

Busca na Árvore B

Universidade Federal do Ceará
Campus Quixadá

Para procurar a chave k em um nó x:

- Basta verificar se a chave está em x. Se estiver, retorna o par (x, i), contendo o nó x e o índice i tal que $x.key[i] = k$.
- Se não estiver em x e o nó x não for folha, basta buscar no filho correto. Se o nó x for folha, então retorna `NIL`.

Busca na Árvore B

Para procurar a chave k em um nó x:

- Basta verificar se a chave está em x. Se estiver, retorna o par (x, i), contendo o nó x e o índice i tal que $x.key[i] = k$.
- Se não estiver em x e o nó x não for folha, basta buscar no filho correto. Se o nó x for folha, então retorna `NIL`.

B-TREE-SEARCH(x, k)

$i = 1$
while $i \leq x.n$ **and** $k > x.key[i]$
    $i = i + 1$
if $i \leq x.n$ **and** $k == x.key[i]$
    return (x, i)
elseif $x.leaf$
    return NIL
else
    DISK-READ$(x.c[i])$
    return B-TREE-SEARCH$(x.c[i], k)$

Busca na Árvore B – Complexidade

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

B-TREE-SEARCH(x, k)

while $i \leq x.n$ **and** $k > x.key[i]$
    $i = i + 1$
if $i \leq x.n$ **and** $k == x.key[i]$
    return (x, i)
elseif $x.leaf$
    return NIL
else
    DISK-READ$(x.c[i])$
    return B-TREE-SEARCH$(x.c[i], k)$

Busca na Árvore B – Complexidade

B-Tree-Search(x, k)

```
i = 1
while i ≤ x.n and k > x.key[i]
    i = i + 1
if i ≤ x.n and k == x.key[i]
    return (x, i)
elseif x.leaf
    return NIL
else
    DISK-READ(x.c[i])
    return B-Tree-Search(x.c[i], k)
```

- A busca em uma árvore B de altura h, acessa $O(h) = O(log_D n)$ páginas de disco, onde n é o número de nós da árvore B.

- Como $x.n < 2D$, o loop **while** itera $O(D)$ vezes dentro de cada nó. Logo o tempo total de CPU é $O(Dh) = O(Dlog_D n)$.

Criando uma Árvore B vazia

Criando uma Árvore B vazia

$\textsc{B-Tree-Create}(T)$

$x = \textsc{Allocate-Node}()$
$x.\mathit{leaf} = \text{TRUE}$
$x.n = 0$
$\text{DISK-WRITE}(x)$
$T.\mathit{root} = x$

- A função $\textsc{B-Tree-Create}(T)$ usa um procedimento auxiliar $\textsc{Allocate-Node}()$, que aloca uma página de disco para ser usada como um novo nó.

Criando uma Árvore B vazia

B-TREE-CREATE(T)

```
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

- A função B-TREE-CREATE(T) usa um procedimento auxiliar ALLOCATE-NODE(), que aloca uma página de disco para ser usada como um novo nó.
- Assumimos que o nó criado por esta função não necessita de DISK-READ(x) pois não há nenhum dado a ser lido do disco quando o nó vazio é alocado.

Criando uma Árvore B vazia

B-TREE-CREATE(T)

$x = \text{ALLOCATE-NODE}()$
$x.\mathit{leaf} = \text{TRUE}$
$x.n = 0$
$\text{DISK-WRITE}(x)$
$T.\mathit{root} = x$

- A função B-TREE-CREATE(T) usa um procedimento auxiliar ALLOCATE-NODE$()$, que aloca uma página de disco para ser usada como um novo nó.
- Assumimos que o nó criado por esta função não necessita de DISK-READ(x) pois não há nenhum dado a ser lido do disco quando o nó vazio é alocado.
- Requer $O(1)$ operações de disco e tem complexidade $O(1)$.

Inserção

Inserção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A inserção acontece sempre em um nó folha.

Inserção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A inserção acontece sempre em um nó folha.
- Contudo, na árvore B não podemos simplesmente criar uma nova folha e inserir a nova chave nela. (Porquê?)

Inserção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A inserção acontece sempre em um nó folha.
- Contudo, na árvore B não podemos simplesmente criar uma nova folha e inserir a nova chave nela. (Porquê?)
- Ao invés disso, inserimos a nova chave em uma folha já existente.
  - Problema: a folha pode já estar cheia (tem $2D - 1$ chaves).

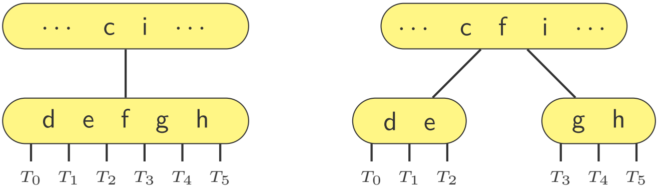
Inserção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A inserção acontece sempre em um nó folha.
- Contudo, na árvore B não podemos simplesmente criar uma nova folha e inserir a nova chave nela. (Porquê?)
- Ao invés disso, inserimos a nova chave em uma folha já existente.
    - Problema: a folha pode já estar cheia (tem $2D - 1$ chaves).
- Introduzimos uma operação que **divide** um nó cheio y na **chave mediana** ($y.chave[D]$) em dois nós com $D - 1$ chaves cada.

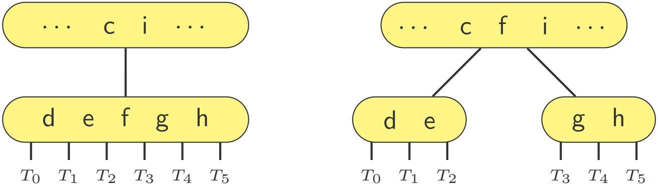
Inserção — Exemplo de divisão de nó

- Exemplo: Árvore B com grau mínimo $D = 3$.



Inserção — Exemplo de divisão de nó

- Exemplo: Árvore B com grau mínimo $D = 3$.



- Inserimos $y.chave[D]$ no nó pai para representar a quebra.
  - Problema: mas o pai poderia estar cheio... O que fazer?

Inserção

- Podemos programar para inserir uma chave em uma árvore B numa única descida da raiz até uma folha.
  - Solução: À medida que vamos descendo, dividimos todo nó cheio que encontrarmos pelo caminho. Assim, o pai de um nó nunca estará cheio.

Inserção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Podemos programar para inserir uma chave em uma árvore B numa única descida da raiz até uma folha.
  - Solução: À medida que vamos descendo, dividimos todo nó cheio que encontrarmos pelo caminho. Assim, o pai de um nó nunca estará cheio.
- Para isso, não esperamos descobrir se vamos precisar dividir um nó cheio durante a inserção.
- Ao invés disso, à medida que formos descendo na árvore procurando a posição correta da nova chave, nós dividimos todo nó cheio que encontrarmos pelo caminho.

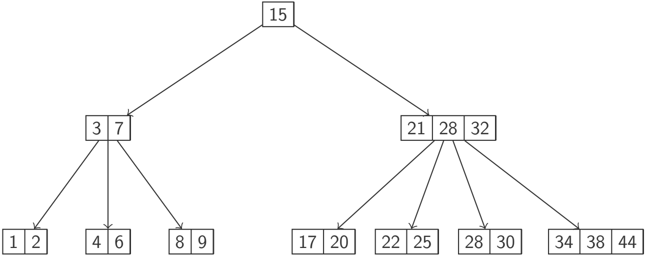
Inserção

- Podemos programar para inserir uma chave em uma árvore B numa única descida da raiz até uma folha.
  - Solução: À medida que vamos descendo, dividimos todo nó cheio que encontrarmos pelo caminho. Assim, o pai de um nó nunca estará cheio.
- Para isso, não esperamos descobrir se vamos precisar dividir um nó cheio durante a inserção.
- Ao invés disso, à medida que formos descendo na árvore procurando a posição correta da nova chave, nós dividimos todo nó cheio que encontrarmos pelo caminho.
- Assim, onde quer que precisarmos dividir um nó cheio, teremos a certeza de que o seu pai não está cheio.

Exemplo: Árvore B com grau mínimo $D = 2$

Inserir chave 39

Atenção para o tamanho máximo de um nó



Algoritmo: Dividindo um nó cheio

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Vamos criar um procedimento `B-Tree-Split-Child(x,i)`
- Este procedimento toma como entrada um nó x não cheio e um índice i tal que $x.c[i]$ é um filho cheio de x.
- Supõe que ambos os nós x e $x.c[i]$ estão na memória principal.
- O procedimento então divide este filho em dois e ajusta x de modo que ele tenha um filho adicional.

Algoritmo: Dividindo um nó cheio

- Vamos criar um procedimento `B-Tree-Split-Child(x,i)`
- Este procedimento toma como entrada um nó x não cheio e um índice i tal que $x.c[i]$ é um filho cheio de x.
- Supõe que ambos os nós x e $x.c[i]$ estão na memória principal.
- O procedimento então divide este filho em dois e ajusta x de modo que ele tenha um filho adicional.
- A fim de dividir uma raiz cheia, depois criaremos um outro procedimento que faz a raiz ser filha de uma nova raiz vazia, de modo que possamos usar o procedimento $\text{B-TREE-SPLIT-CHILD}(x, i)$. Fazendo isso, a árvore aumenta sua altura em 1.
- **Divisão** é o único modo pelo qual uma árvore B aumenta de tamanho.

Algoritmo: Dividindo um nó cheio

Algoritmo: Dividindo um nó cheio

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

B-TREE-SPLIT-CHILD(x, i)

```
y = x.c[i]
z = ALLOCATE-NODE()
z.leaf = y.leaf
z.n = D − 1
for j = 1 to D − 1
    z.key[j] = y.key[j + D]
if not y.leaf
    for j = 1 to D
        z.c[j] = y.c[j + D]
y.n = D − 1
for j = x.n + 1 downto i + 1
    x.c[j + 1] = x.c[j]
x.c[i + 1] = z
for j = x.n downto i
    x.key[j + 1] = x.key[j]
x.key[i] = y.key[D]
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

- Este procedimento toma como entrada um nó x não cheio e um índice i tal que $x.c[i]$ é um filho cheio de x.
- Supõe que ambos os nós x e $x.c[i]$ estão na memória principal.
- O procedimento então divide este filho em dois e ajusta x de modo que ele tenha um filho adicional.
- A fim de dividir uma raiz cheia, primeiro fazemos a raiz filha de uma nova raiz vazia, de modo que possamos usar o procedimento B-TREE-SPLIT-CHILD(x, i). Fazendo isso, a árvore aumenta sua altura em 1.
- **Divisão** é o único modo pelo qual uma árvore B aumenta de tamanho.

Algoritmo: Dividindo um nó cheio

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

B-TREE-SPLIT-CHILD(x, i)

```
1   y = x.c[i]
2   z = ALLOCATE-NODE()
3   z.leaf = y.leaf
4   z.n = D − 1
5   for j = 1 to D − 1
6         z.key[j] = y.key[j + D]
7   if not y.leaf
8         for j = 1 to D
9               z.c[j] = y.c[j + D]
10  y.n = D − 1
11  for j = x.n + 1 downto i + 1
12        x.c[j + 1] = x.c[j]
13  x.c[i + 1] = z
14  for j = x.n downto i
15        x.key[j + 1] = x.key[j]
16  x.key[i] = y.key[D]
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

Complexidade

- Esta função executa em tempo $\Theta(D)$ devido aos laços das linhas $5 - 6$ e $8 - 9$.
- Os outros laços rodam em tempo $O(D)$.
- São executadas $O(1)$ operações em disco.

Algoritmo: Inserindo uma chave

Vamos inserir a chave k na árvore T

- Verificamos se não é necessário dividir a raiz.

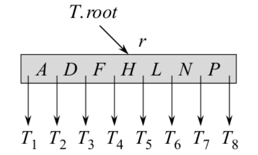
Algoritmo: Inserindo uma chave

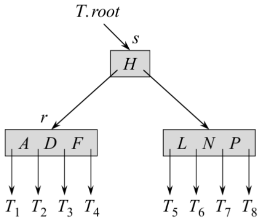
Vamos inserir a chave k na árvore T

- Verificamos se não é necessário dividir a raiz.

B-TREE-INSERT(T, k)

```
r = T.root
if r.n == 2D − 1
    s = ALLOCATE-NODE()
    s.leaf = FALSE
    s.n = 0
    s.c[1] = r
    T.root = s
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else
    B-TREE-INSERT-NONFULL(r, k)
```





Inserindo chave k em um nó não-cheio x

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

B-TREE-INSERT-NONFULL(x, k)

- Se x for uma folha, percorremo-lo de trás para frente, movendo as chaves para trás uma posição, até encontrar a posição correta da chave k. Inserimos a chave k na posição correta, ajustamos o tamanho de x e escrevemos x no disco.
- Se x não for folha, encontramos a subárvore $x.c[i]$ onde k deve ser inserido. Lemos o nó $x.c[i]$ do disco. Se $x.c[i]$ estiver cheio, dividimos esse nó em dois. Por fim, invocamos a função de inserção em nó não cheio.

Inserindo chave k em um nó não-cheio x

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

B-TREE-INSERT-NONFULL(x, k)

- Se x for uma folha, percorremo-lo de trás para frente, movendo as chaves para trás uma posição, até encontrar a posição correta da chave k. Inserimos a chave k na posição correta, ajustamos o tamanho de x e escrevemos x no disco.
- Se x não for folha, encontramos a subárvore $x.c[i]$ onde k deve ser inserido. Lemos o nó $x.c[i]$ do disco. Se $x.c[i]$ estiver cheio, dividimos esse nó em dois. Por fim, invocamos a função de inserção em nó não cheio.

B-TREE-INSERT-NONFULL(x, k)

```
i = x.n
if x.leaf
    while i ≥ 1 e k < x.key[i]
        x.key[i + 1] = x.key[i]
        i = i − 1
    x.key[i + 1] = k
    x.n = x.n + 1
    DISK-WRITE(x)
else
    while i ≥ 1 and k < x.key[i]
        i = i − 1
    i = i + 1
    DISK-READ(x.c[i])
    if x.c[i].n == 2D − 1
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key[i]
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c[i], k)
```

Qual a complexidade da inserção?

- Para uma árvore B de altura h, B-TREE-INSERT realiza $O(h)$ acessos a disco, já que $O(1)$ operações de leitura/escrita são realizadas em cada nível da árvore.
- Além disso, o tempo de CPU é $O(D)$ em cada nível da árvore. Logo, ao todo, o tempo de CPU é $O(Dh) = O(D \log_D n)$.

Exercícios

Exercício

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Mostre o resultado de inserir as chaves 6, 19, 17, 11, 3, 12, 8, 20, 22, 23, 13, 18, 14, 16, 1, 2, 24, 25, 4, 26, 5 em uma árvore B vazia com grau mínimo $D = 2$.

Remoção

Remoção

- A remoção é mais complicada que a inserção.
    - A chave a ser removida pode ocorrer em qualquer nó da árvore.
    - Quando removemos uma chave de um nó interno, teremos que reorganizar os filhos do nó.

Remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A remoção é mais complicada que a inserção.
  - A chave a ser removida pode ocorrer em qualquer nó da árvore.
  - Quando removemos uma chave de um nó interno, teremos que reorganizar os filhos do nó.
- Como na inserção, devemos ter certeza de que a remoção não viola as propriedades da árvore B.
  - Cada nó deve continuar com pelo menos $D - 1$ chaves. Exceto a raiz que tem que ter pelo menos uma chave.

Remoção

- Na inserção, a chave é sempre inserida em uma folha. Na remoção, assim como fizemos nas árvores AVL e rubro-Negra, também vamos sempre remover uma chave k de uma folha. Há dois modos de fazer isso:

(1) Se a chave k se encontra em uma folha: a chave é simplesmente removida.

Remoção

- Na inserção, a chave é sempre inserida em uma folha. Na remoção, assim como fizemos nas árvores AVL e rubro-Negra, também vamos sempre remover uma chave k de uma folha. Há dois modos de fazer isso:

(1) Se a chave k se encontra em uma folha: a chave é simplesmente removida.

(2) Se a chave k não se encontra em uma folha: k é substituída pelo seu sucessor. Como uma árvore B é uma árvore de busca, temos a garantia de que o sucessor de k está em uma folha.

Remoção

- Na inserção, a chave é sempre inserida em uma folha. Na remoção, assim como fizemos nas árvores AVL e rubro-Negra, também vamos sempre remover uma chave k de uma folha. Há dois modos de fazer isso:

(1) Se a chave k se encontra em uma folha: a chave é simplesmente removida.

(2) Se a chave k não se encontra em uma folha: k é substituída pelo seu sucessor. Como uma árvore B é uma árvore de busca, temos a garantia de que o sucessor de k está em uma folha.

   - A retirada e substituição das chaves pressupõem que elas sejam acompanhadas de sua informação.

Remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Na inserção, a chave é sempre inserida em uma folha. Na remoção, assim como fizemos nas árvores AVL e rubro-Negra, também vamos sempre remover uma chave k de uma folha. Há dois modos de fazer isso:

(1) Se a chave k se encontra em uma folha: a chave é simplesmente removida.

(2) Se a chave k não se encontra em uma folha: k é substituída pelo seu sucessor. Como uma árvore B é uma árvore de busca, temos a garantia de que o sucessor de k está em uma folha.

  - A retirada e substituição das chaves pressupõem que elas sejam acompanhadas de sua informação.

- Quando a chave é retirada, o número de chaves do nó pode resultar menor que $D - 1$, o que contraria a propriedade da árvore B.

Remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Na inserção, a chave é sempre inserida em uma folha. Na remoção, assim como fizemos nas árvores AVL e rubro-Negra, também vamos sempre remover uma chave k de uma folha. Há dois modos de fazer isso:

(1) Se a chave k se encontra em uma folha: a chave é simplesmente removida.

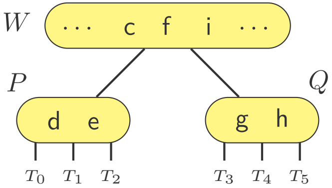
(2) Se a chave k não se encontra em uma folha: k é substituída pelo seu sucessor. Como uma árvore B é uma árvore de busca, temos a garantia de que o sucessor de k está em uma folha.

  - A retirada e substituição das chaves pressupõem que elas sejam acompanhadas de sua informação.

- Quando a chave é retirada, o número de chaves do nó pode resultar menor que $D-1$, o que contraria a propriedade da árvore B.
  - Existem dois tratamentos para esse problema, denominados concatenação e redistribuição.

Definição: irmãos adjacentes

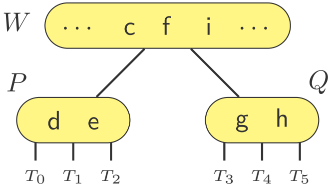
- Dois nós P e Q são chamados irmãos adjacentes se têm o mesmo pai W e são apontados por dois ponteiros adjacentes $W.c[i]$ e $W.c[i+1]$ em W.



Definição: irmãos adjacentes

Universidade Federal do Ceará
Campus Quixadá

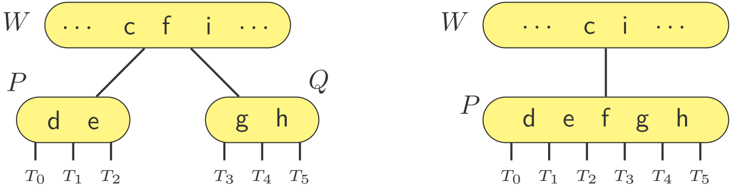
- Dois nós P e Q são chamados irmãos adjacentes se têm o mesmo pai W e são apontados por dois ponteiros adjacentes $W.c[i]$ e $W.c[i+1]$ em W.



- P e Q podem ser concatenados se são irmãos adjacentes e juntos possuem menos do que $2D-1$ chaves. A concatenação agrupa as entradas dos dois nós P e Q em um nó só.

Concatenação

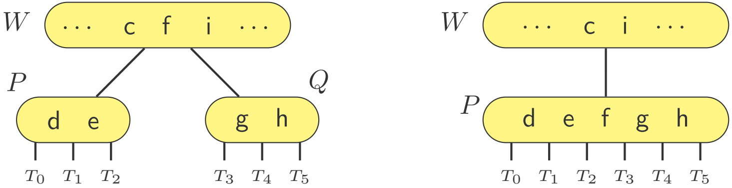
- Exemplo: Árvore B com grau mínimo $D = 3$.



Concatenação

Universidade Federal do Ceará
Campus Quixadá

- Exemplo: Árvore B com grau mínimo $D = 3$.



- No nó pai W, a chave que separa os ponteiros para P e Q deixa de existir. Esta chave passa a fazer parte do novo nó concatenado e o ponteiro para o nó Q desaparece, uma vez que o nó Q é liberado.

- Como a soma do número de chaves de P e Q era menor que $2D - 1$, o novo nó tem, no máximo, $2D - 1$ chaves.

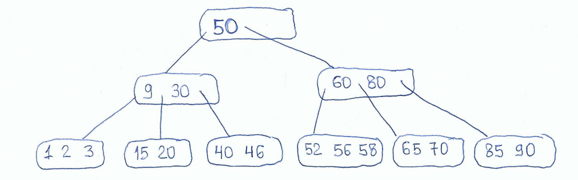
Concatenação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Observação:** Como foi retirada uma chave do nó pai W, observa-se que a concatenação é propagável.
- Ou seja, na nova situação, caso W contenha menos que $D - 1$ chaves e cada um de seus dois irmãos adjacentes tenham no máximo $D - 1$ chaves, o processo se repete.
    - Eventualmente, a propagação pode atingir a raiz, o que acarreta a diminuição da altura da árvore.

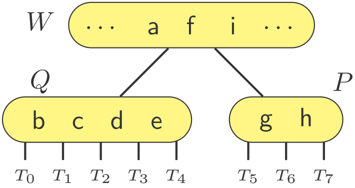
Exemplo de Concatenação

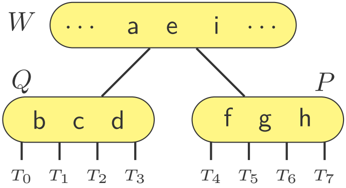
- Retirar a chave 40. Grau mínimo $D = 3$



Redistribuição

- A redistribuição acontece quando um nó P tem menos que $D - 1$ chaves e um irmão adjacente seu Q possui mais do que $D - 1$ chaves.
  - Neste caso, as chaves de Q podem ser redistribuídas para P.
  - Basta tomar uma chave e trazê-la para o nó P via pai W.





Redistribuição

- **Atenção:** A redistribuição também pode acontecer da direita para a esquerda.

Redistribuição

- **Atenção:** A redistribuição também pode acontecer da direita para a esquerda.
- Se necessário, para manter um maior equilíbrio entre as páginas da árvore B, pode ser tomado emprestado um número maior de chaves de forma que P e Q fiquem com aproximadamente o mesmo número de chaves.

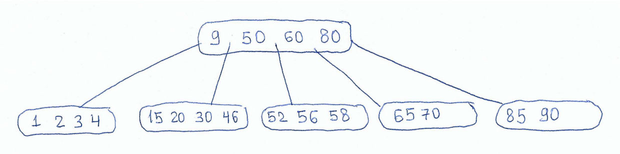
Redistribuição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Atenção:** A redistribuição também pode acontecer da direita para a esquerda.
- Se necessário, para manter um maior equilíbrio entre as páginas da árvore B, pode ser tomado emprestado um número maior de chaves de forma que P e Q fiquem com aproximadamente o mesmo número de chaves.
- A redistribuição não é propagável. A página W, pai de P e Q, é modificada, mas seu número de chaves permanece o mesmo.

Exemplo de Redistribuição

- Retirar a chave 65. Grau mínimo $D = 3$.



Algoritmo de remoção

O algoritmo de remoção de uma chave k de uma árvore B aqui apresentado é inteiramente recursivo e segue os seguintes passos:

Algoritmo de remoção

O algoritmo de remoção de uma chave k de uma árvore B aqui apresentado é inteiramente recursivo e segue os seguintes passos:

(1) Inicialmente, deve-se buscar a chave k recursivamente, semelhante ao que foi feito no algoritmo `B-Tree-Search`.

Algoritmo de remoção

O algoritmo de remoção de uma chave k de uma árvore B aqui apresentado é inteiramente recursivo e segue os seguintes passos:

(1) Inicialmente, deve-se buscar a chave k recursivamente, semelhante ao que foi feito no algoritmo `B-Tree-Search`.

(2) **Caso Base 1**: Chegou-se a um nó folha P e k não foi encontrada nele. Neste caso, nada se faz, pois k não está na árvore.

Algoritmo de remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

O algoritmo de remoção de uma chave k de uma árvore B aqui apresentado é inteiramente recursivo e segue os seguintes passos:

(1) Inicialmente, deve-se buscar a chave k recursivamente, semelhante ao que foi feito no algoritmo `B-Tree-Search`.

(2) **Caso Base 1**: Chegou-se a um nó folha P e k não foi encontrada nele. Neste caso, nada se faz, pois k não está na árvore.

(3) **Caso Base 2**: Chegou-se a um nó folha P e k foi encontrada nele. Neste caso, remove-se a chave k de P e ajusta-se a posição das chaves remanescentes.

Algoritmo de remoção

O algoritmo de remoção de uma chave k de uma árvore B aqui apresentado é inteiramente recursivo e segue os seguintes passos:

(1) Inicialmente, deve-se buscar a chave k recursivamente, semelhante ao que foi feito no algoritmo `B-Tree-Search`.

(2) **Caso Base 1**: Chegou-se a um nó folha P e k não foi encontrada nele. Neste caso, nada se faz, pois k não está na árvore.

(3) **Caso Base 2**: Chegou-se a um nó folha P e k foi encontrada nele. Neste caso, remove-se a chave k de P e ajusta-se a posição das chaves remanescentes.

(4) **Caso Geral 1**: Chegou se a um nó W não-folha e percebeu-se que k não está nele, mas sim em um filho $W.child[i]$. Descer recursivamente até este filho.

Algoritmo de remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(5) **Caso geral 2**: Chegou se a um nó W não-folha e percebeu-se que k encontra-se na posição $W.key[i]$ dele. Com certeza, W tem um filho $W.child[i+1]$. Buscar, recursivamente, a chave sucessora de k na subárvore $W.child[i+1]$. Seja z essa chave, e P o nó onde z se encontra. Substitua a chave k por z e remova z de P (recaímos no Caso Base 2).

Algoritmo de remoção

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(5) **Caso geral 2**: Chegou se a um nó W não-folha e percebeu-se que k encontra-se na posição $W.key[i]$ dele. Com certeza, W tem um filho $W.child[i+1]$. Buscar, recursivamente, a chave sucessora de k na subárvore $W.child[i+1]$. Seja z essa chave, e P o nó onde z se encontra. Substitua a chave k por z e remova z de P (recaímos no Caso Base 2).

(6) **Regulagem dos nós**: Na volta de cada chamada recursiva, o nó filho P que foi invocado naquela chamada deve ser checado para ver se contém pelo menos $D-1$ chaves. Em caso negativo, uma das operações de regulagem deve ser feita. Devemos olhar os irmãos adjacentes de P:

- **(6.1)** Se P tiver um irmão adjacente Q com mais do que $D-1$ chaves, execute a operação de redistribuição em P e Q.

- **(6.2)** Caso contrário, ambos os irmãos adjacentes de P possuem no máximo $D-1$ chaves. Então, execute a operação de concatenação em P e um de seus irmãos Q.

Exercício

Escreva o pseudocódigo para a remoção em árvores B.

Você pode considerar as funções a seguir:

- `void concatenate(BNode *x, int i)`

  Função que concatena o filho $x.c[i]$ e o filho $x.c[i+1]$. A soma do número de chaves de $x.c[i]$ e $x.c[i+1]$ é menor que $2d-1$ quando esta função é chamada. Após essa operacão ser realizada, o nó $x.c[i]$ fica como no máximo $2d-1$ chaves e o nó $x.c[i+1]$ é liberado (deleted). Após essa operação, a chave $x.key[i]$ que separava $x.c[i]$ e $x.c[i+1]$ no nó x, deixa de existir em x e passa a fazer parte do filho $x.c[i]$. Com isso, o pai x deve ter seus atributos também atualizados.

Exercício (cont.)

- `void borrowFromLeft(Bnode *x, int i)`

  Função que redistribui as chaves do filho x.c[i-1] para o filho x.c[i] do nó x. Esses dois filhos são irmãos adjacentes. O filho x.c[i] tem menos do que d-1 chaves e o filho x.c[i-1] tem mais do que d-1 chaves. Após esta operação, x.c[i] passa a ter exatamente uma chave a mais e o filho x.c[i-1] passa a ter uma chave a menos. Além disso, a chave que separa estes dois irmãos adjacentes tambem é modificada, porém, o pai fica com a mesma quantidade de chaves.

Exercício (cont.)

- `void borrowFromRight(Bnode *x, int i)`

  Função que redistribui as chaves do filho x.c[i+1] para o filho x.c[i] do node x. Esses dois filhos são irmãos adjacentes. O filho x.c[i] tem menos do que d-1 chaves e o filho x.c[i+1] tem mais do que d-1 chaves. Após esta operação, x.c[i] passa a ter exatamente uma chave a mais e o filho x.c[i+1] passa a ter uma chave a menos. Além disso, a chave que separa estes dois irmãos adjacentes também é modificada, porém, o pai fica com a mesma quantidade de chaves.

Exercício (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- `void remove_from_leaf(Bnode *x, int i)`

  Esta função recebe o nó folha x e um indice i e remove a chave no índice i

Exercício (cont.)

- `void remove_key(Bnode *x, int k)`

  Recebe como argumento um ponteiro x para a raiz de uma subárvore B e um valor de chave k e remove esta chave da árvore enraizada neste nó se e somente se a chave estiver na árvore; caso contrario, não faz nada e a árvore permanece como estava.

  Após o retorno de cada chamada recursiva, esta função deve ajustar o nó que foi visitado se for necessário, pois a árvore deve permanecer uma árvore B após a remoção. Ou seja, essa função deve fazer o rebalanceamento do nó visitado.

  No caso em que a chave k encontra-se em um nó não-folha, esta função invoca uma outra função recursiva para continuar a remoção, chamada `removeSuccessor`, que é explicada no próximo slide.

Exercício (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

A tarefa de remoção do nó pode ser divida em duas funções recursivas, declaradas a seguir.

- `void removeSuccessor(Bnode *x, int i, Bnode *y)`

  Esta função recursiva recebe como entrada um nó x tal que $x.key[i]$ é a chave que queremos remover. Porém, quando esta função é chamada, sabemos que x não é um nó folha. Logo, a chave $x.key[i]$ possui um sucessor, que está na subárvore com raiz y. Portanto, esta função encontrará a chave Z sucessora da chave $x.key[i]$, copiará a chave Z em $x.key[i]$ e, então, removerá a chave Z do nó original ao qual ela pertencia. Esta função deve regular os nós que foram visitados toda vez que for necessário.

  A primeira chamada para esta função é algo assim:
  `removeSuccessor(x, i, x.c[i+1]);`

FIM