"Um cínico certa vez disse: afortunado aquele que não espera nada, pois nunca se desapontará" (Benjamin Graham).

Árvores B

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

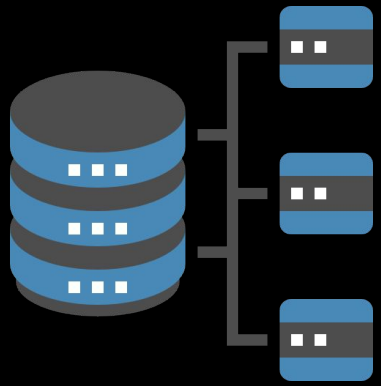
Problema

Muitas vezes a árvore pode não caber na memória principal.

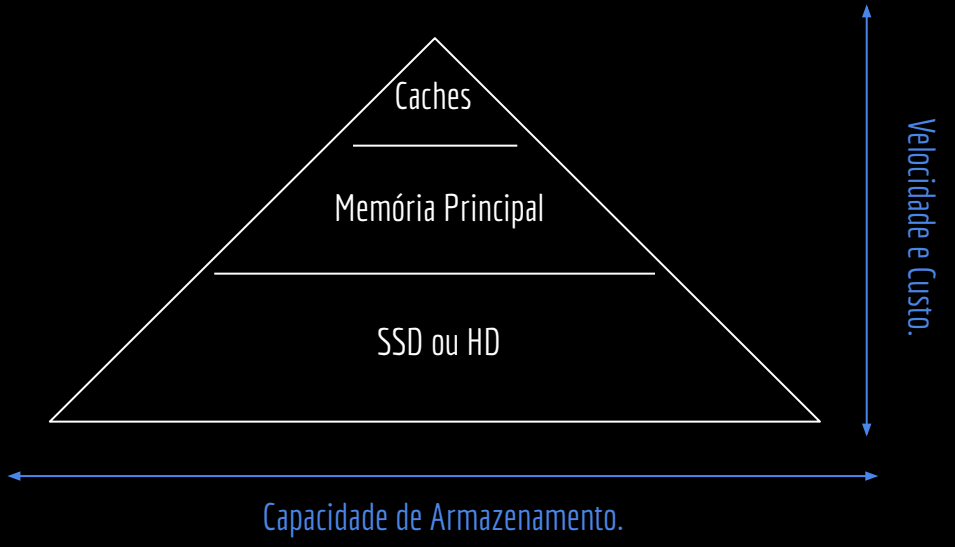
Necessário acessar a memória secundária.

HD, SSD, ...

Problema recorrente em sistemas de banco de dados, por exemplo.



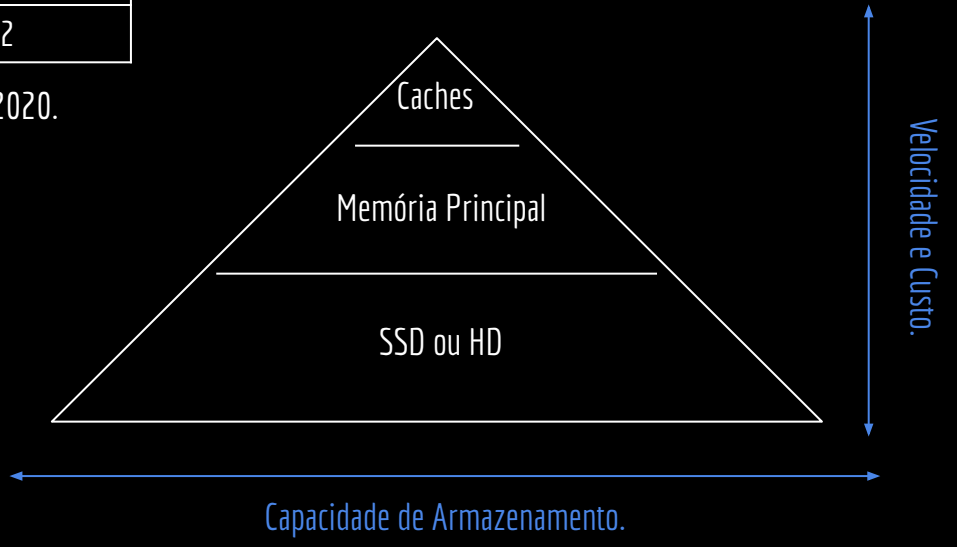
Um pouco de arquitetura...



Um pouco de arquitetura...

Tecnologia	Tempo de acesso típico	Dólares por GiB em 2020
SRAM	0.5 – 2.5 ns	U$500 – U$1000
DRAM	50 – 70 ns	U$3 – U$6
Flash	5.000 – 50.000 ns	U$0,03 – U$0,12
Disco Magnético	5.000.000 – 20.000.000 ns	U$0,01 – U$0,02

Patterson, Hennessy. Computer Organization and Design RISC-V Edition. 2020.



Blocos

Diferente da memória principal, que comumente acessa os dados a byte, Discos Rígidos e SSDs enviam os dados em blocos.

- Mitigar atrasos.

Um bloco tem tamanho típico entre 512 e 4096 bytes.

Dica:

- Para verificar os sistemas de arquivos, use `df -h`
- Para verificar o tamanho do bloco no em um sistema, use sudo `blockdev --getbsz /dev/Nome`

Árvores B

Árvores B (B-Trees) são similares a árvores Red-Black.

Mas são melhores em minimizar o número de acessos ao armazenamento secundário.

Um nodo de uma árvore B pode ter múltiplos filhos.

- De alguns a milhares;
- Graus altos;
- Depende das propriedades do armazenamento secundário.

A altura de qualquer nodo em uma Árvore B é $O(\log_x n)$.

- Como o grau dos nodos é maior, a base do logarítmo (x) pode ser maior que 2.

Árvores B

Árvores B copiam blocos da memória secundária para a principal conforme necessário.

Alterações na árvore são escritas na memória secundária.

Agora o tamanho da memória principal agora não é um fator limitante.

Árvores B

Em uma Árvore B, o nodo é tipicamente projetado para ter o mesmo tamanho do bloco.

```
struct nodo{
      //...
};
```

Árvores B

Uma árvore B é uma árvore com raiz, e com as seguintes propriedades:

1. Todo nodo possui:
    a. O número *n* de chaves armazenadas;
    b. As chaves, armazenadas de forma monotonicamente crescente (não decrescente);
    c. Um booleano informando se o nodo é uma folha.

Árvores B

Uma árvore B é uma árvore com raiz, e com as seguintes propriedades:

1. Todo nodo possui:
    a. O número *n* de chaves armazenadas;
    b. As chaves, armazenadas de forma monotonicamente crescente (não decrescente);
    c. Um booleano informando se o nodo é uma folha.
2. Todo nodo interno contém $n+1$ ponteiros para seus filhos $c_1, c_2, ..., c_{n+1}$.
    a. As folhas não possuem esses ponteiros.

Árvores B

Uma árvore B é uma árvore com raiz, e com as seguintes propriedades:

1. Todo nodo possui:
    a. O número *n* de chaves armazenadas;
    b. As chaves, armazenadas de forma monotonicamente crescente (não decrescente);
    c. Um booleano informando se o nodo é uma folha.
2. Todo nodo interno contém $n+1$ ponteiros para seus filhos $c_1, c_2, ..., c_{n+1}$.
    a. As folhas não possuem esses ponteiros.
3. Considere k_i uma chave qualquer armazenada na subárvore apontada por c_i em um nodo x. Então:
    a. $k_1 \leq x.chave_1 \leq k_2 \leq x.chave_2 \leq ... \; k_n \leq x.chave_n \leq k_{n+1}$

Árvores B

Uma árvore B é uma árvore com raiz, e com as seguintes propriedades:

4. Todas as folhas estão no mesmo nível.
    a. Que é igual a altura *h* da árvore.

Árvores B

Uma árvore B é uma árvore com raiz, e com as seguintes propriedades:

4. Todas as folhas estão no mesmo nível.
    a. Que é igual a altura *h* da árvore.
5. Todo nodo possui um limite inferior e superior para o número de chaves. O limite inferior é definido como $t \geq 2$ (grau mínimo).
    a. Todo nodo, exceto a raiz, precisa de pelo menos $t - 1$ chaves.
    b. Todo nodo interno, exceto a raiz, precisa de pelo menos t filhos.
    c. Toda árvore não vazia precisa de uma raiz com pelo menos uma chave.
    d. Todo nodo pode possuir no máximo $2t-1$ chaves.
        i. Logo, um nodo interno pode ter no máximo 2t filhos.
        ii. Um nodo é dito **cheio** se contém exatamente $2t-1$ chaves.

Formalismo

Primeiro, vamos provar que $\sum_{k=0}^{n-1} ar^k = \sum_{k=1}^{n} ar^{k-1} = a\frac{r^n - 1}{r - 1}, \forall r \neq 1$

Prova baseada em en.wikipedia.org/wiki/Geometric_series#Sum

Formalismo

$$s_n = \sum_{k=1}^{n} ar^{k-1}$$

$$s_n = ar^0 + ar^1 + \cdots + ar^{n-1}$$

Formalismo

$$s_n = \sum_{k=1}^{n} ar^{k-1}$$

$$s_n = ar^0 + ar^1 + \cdots + ar^{n-1}$$

$$rs_n = ar^1 + ar^2 + \cdots + ar^n$$

Formalismo

$$s_n = \sum_{k=1}^{n} ar^{k-1}$$

$$s_n = ar^0 + ar^1 + \cdots + ar^{n-1}$$

$$rs_n = ar^1 + ar^2 + \cdots + ar^n$$

$$s_n - rs_n = ar^0 - ar^n$$

Formalismo

$$s_n = \sum_{k=1}^{n} ar^{k-1}$$

$$s_n = ar^0 + ar^1 + \cdots + ar^{n-1}$$

$$rs_n = ar^1 + ar^2 + \cdots + ar^n$$

$$s_n - rs_n = ar^0 - ar^n$$

$$s_n(1 - r) = a(1 - r^n)$$

Formalismo

$$s_n = \sum_{k=1}^{n} ar^{k-1}$$

$$s_n = ar^0 + ar^1 + \cdots + ar^{n-1}$$

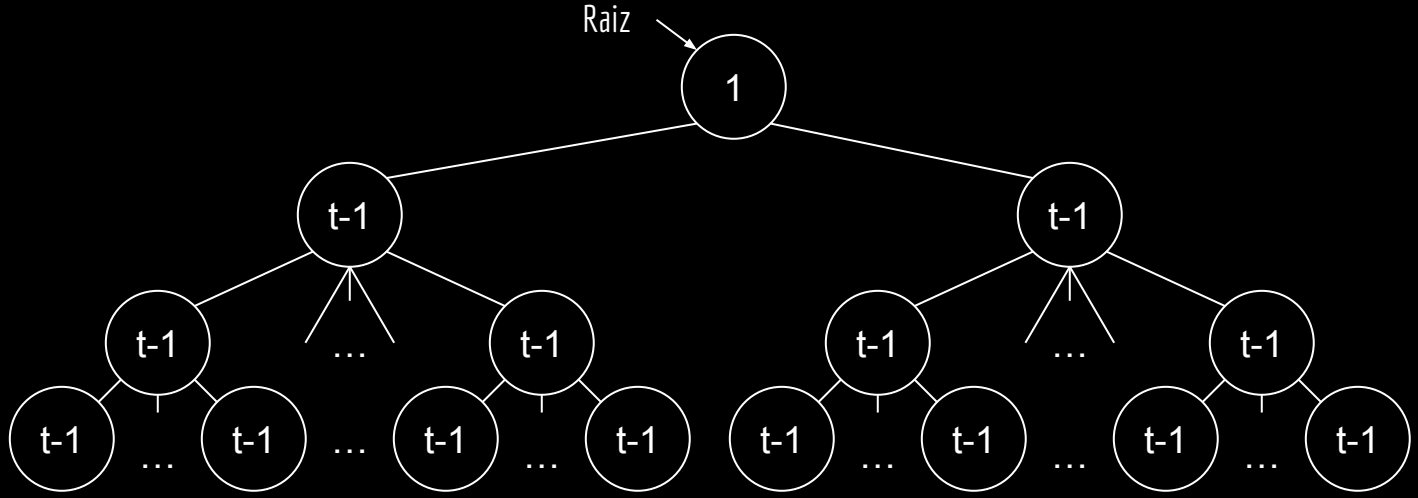
$$rs_n = ar^1 + ar^2 + \cdots + ar^n$$

$$s_n - rs_n = ar^0 - ar^n$$

$$s_n(1-r) = a(1-r^n)$$

$$s_n = a\frac{(1-r^n)}{(1-r)} = a\frac{(r^n-1)}{(r-1)}, \forall r \neq 1$$

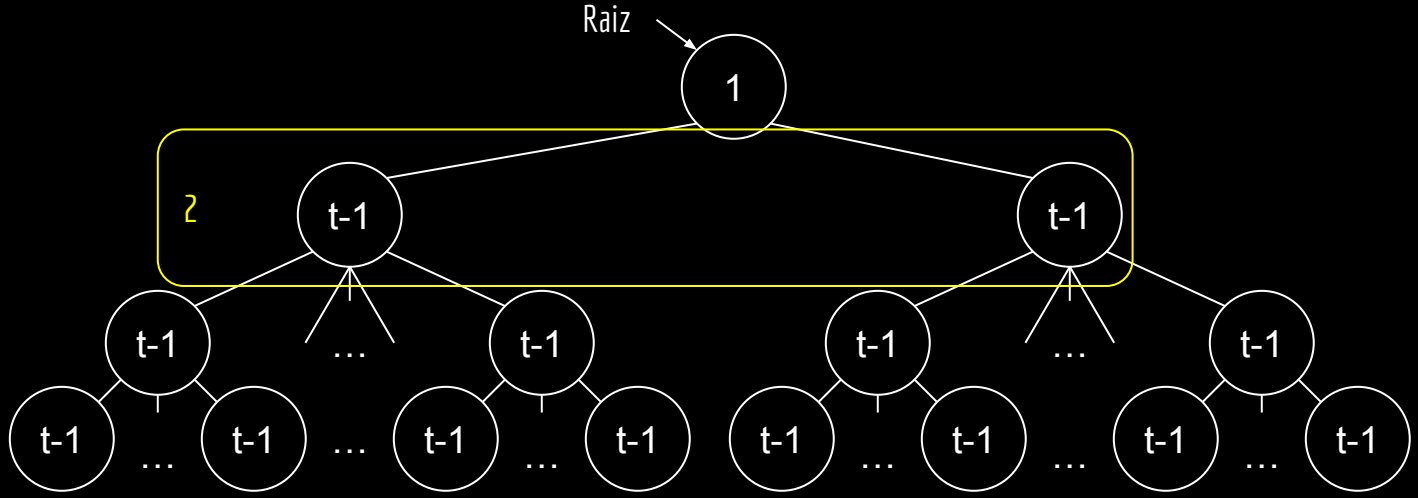
Formalismo



Altura *h* de uma Árvore B não vazia, com t ≥ 2:

Pela definição, a árvore possui uma raiz com pelo menos uma chave. Todos os demais nodos possuem pelo menos *t -1* chaves.

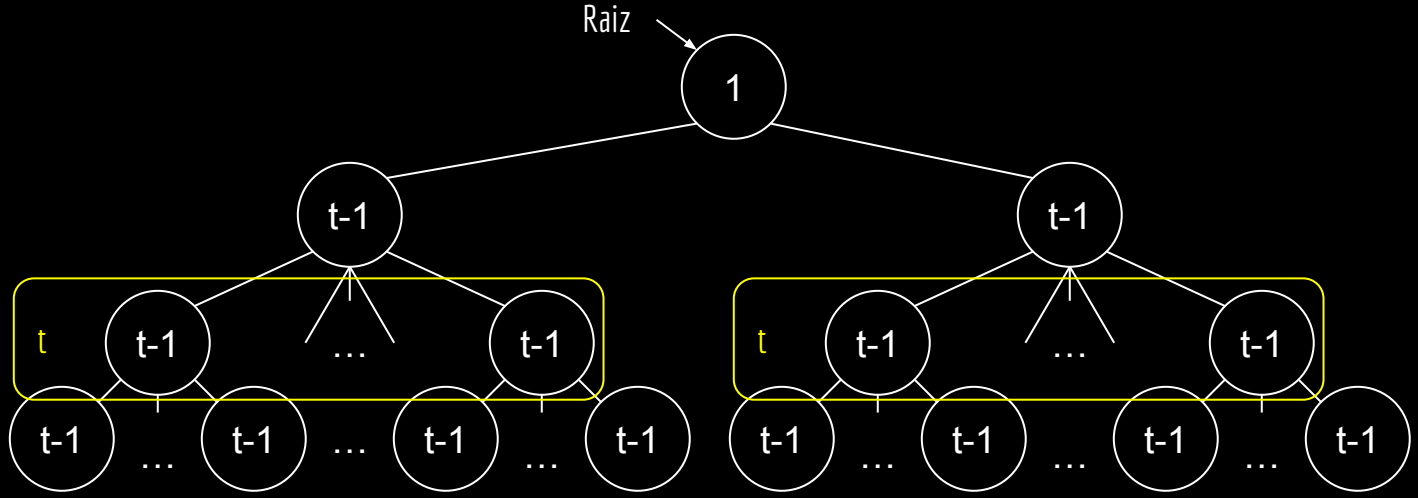
Formalismo



Altura *h* de uma Árvore B não vazia, com t ≥ 2:

Pela definição, a árvore possui uma raiz com pelo menos uma chave. Todos os demais nodos possuem pelo menos *t -1* chaves. No nível 1, temos pelo menos 2 nodos.

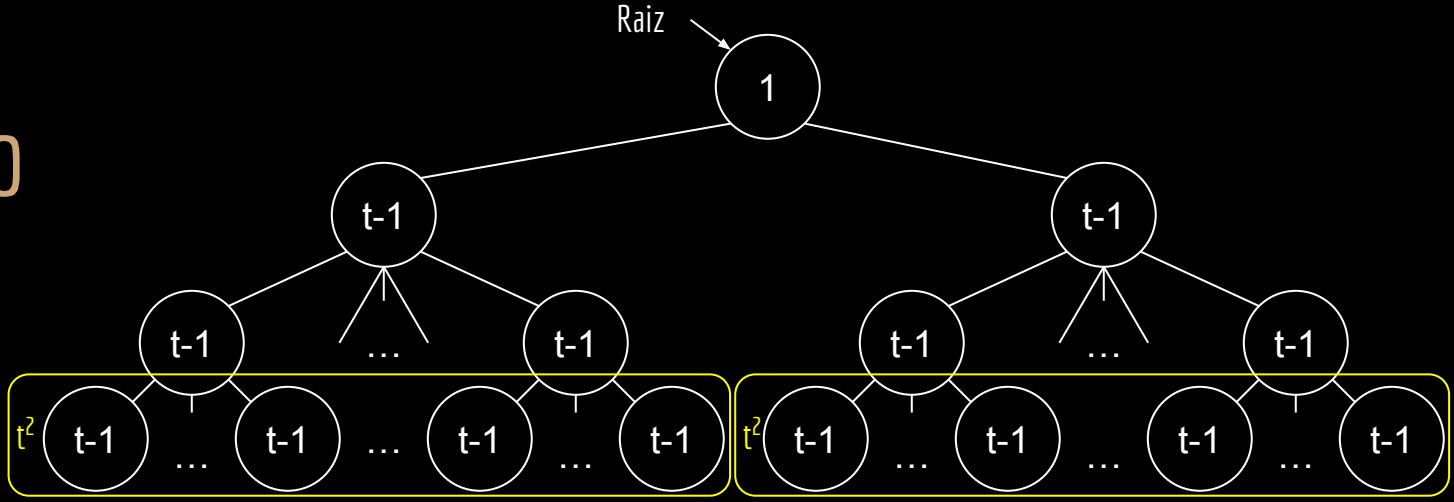
Formalismo



Altura *h* de uma Árvore B não vazia, com t ≥ 2:

Pela definição, a árvore possui uma raiz com pelo menos uma chave. Todos os demais nodos possuem pelo menos *t -1* chaves. No nível 1, temos pelo menos 2 nodos. No nível 2 temos pelo menos *2t* nodos.

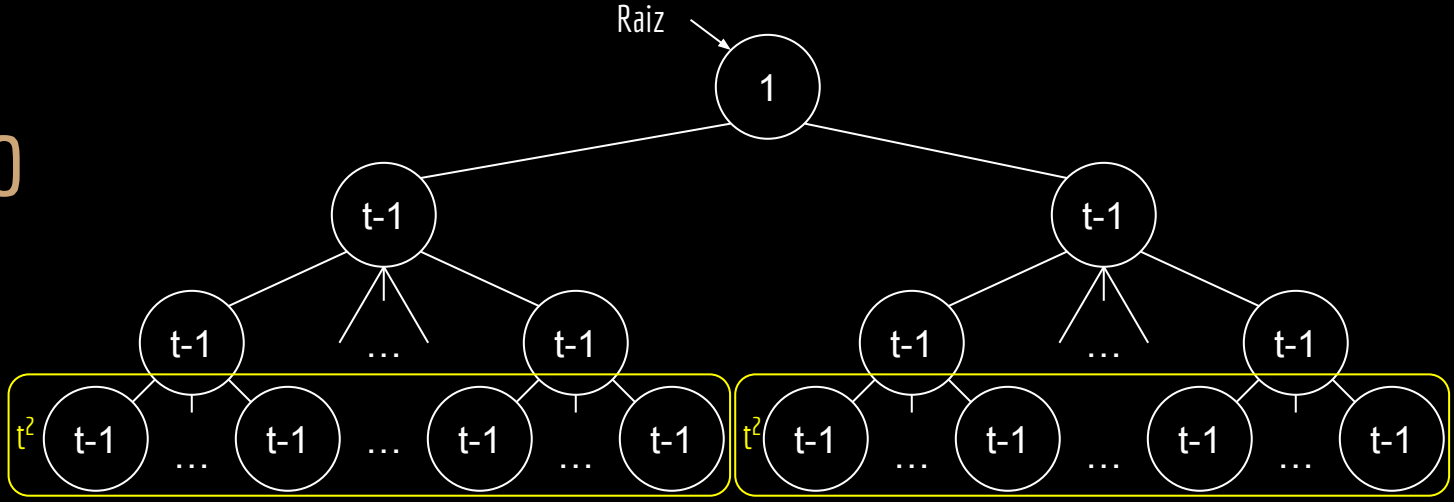
Formalismo



Altura *h* de uma Árvore B não vazia, com t ≥ 2:

Pela definição, a árvore possui uma raiz com pelo menos uma chave. Todos os demais nodos possuem pelo menos *t -1* chaves. No nível 1, temos pelo menos 2 nodos. No nível 2 temos pelo menos *2t* nodos. No nível 3 temos pelo menos $2t^2$ nodos.

Formalismo

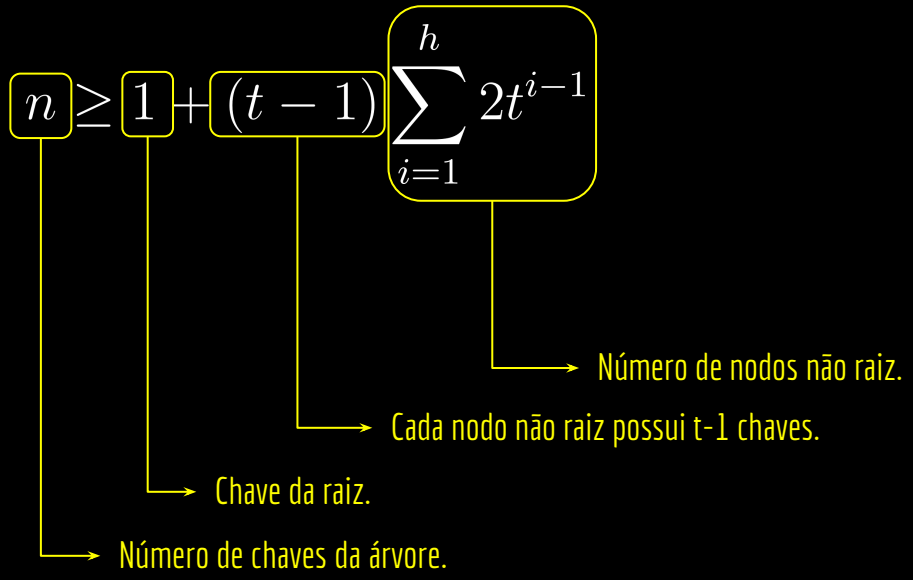


Altura *h* de uma Árvore B não vazia, com t ≥ 2:

Pela definição, a árvore possui uma raiz com pelo menos uma chave. Todos os demais nodos possuem pelo menos *t -1* chaves. No nível 1, temos pelo menos 2 nodos. No nível 2 temos pelo menos *2t* nodos. No nível 3 temos pelo menos $2t^2$ nodos. No nível h temos pelo menos $2t^{h-1}$ nodos.

Formalismo

Temos então que:



Formalismo

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

$$n \geq 1 + (t-1) 2 \frac{t^h - 1}{t - 1}$$

Formalismo

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

$$n \geq 1 + (t-1)2\frac{t^h - 1}{t-1}$$

$$n \geq 2t^h - 1$$

Formalismo

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

$$n \geq 1 + (t-1) 2 \frac{t^h - 1}{t-1}$$

$$n \geq 2t^h - 1$$

$$h \leq \log_t \frac{n+1}{2}$$

Formalismo

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

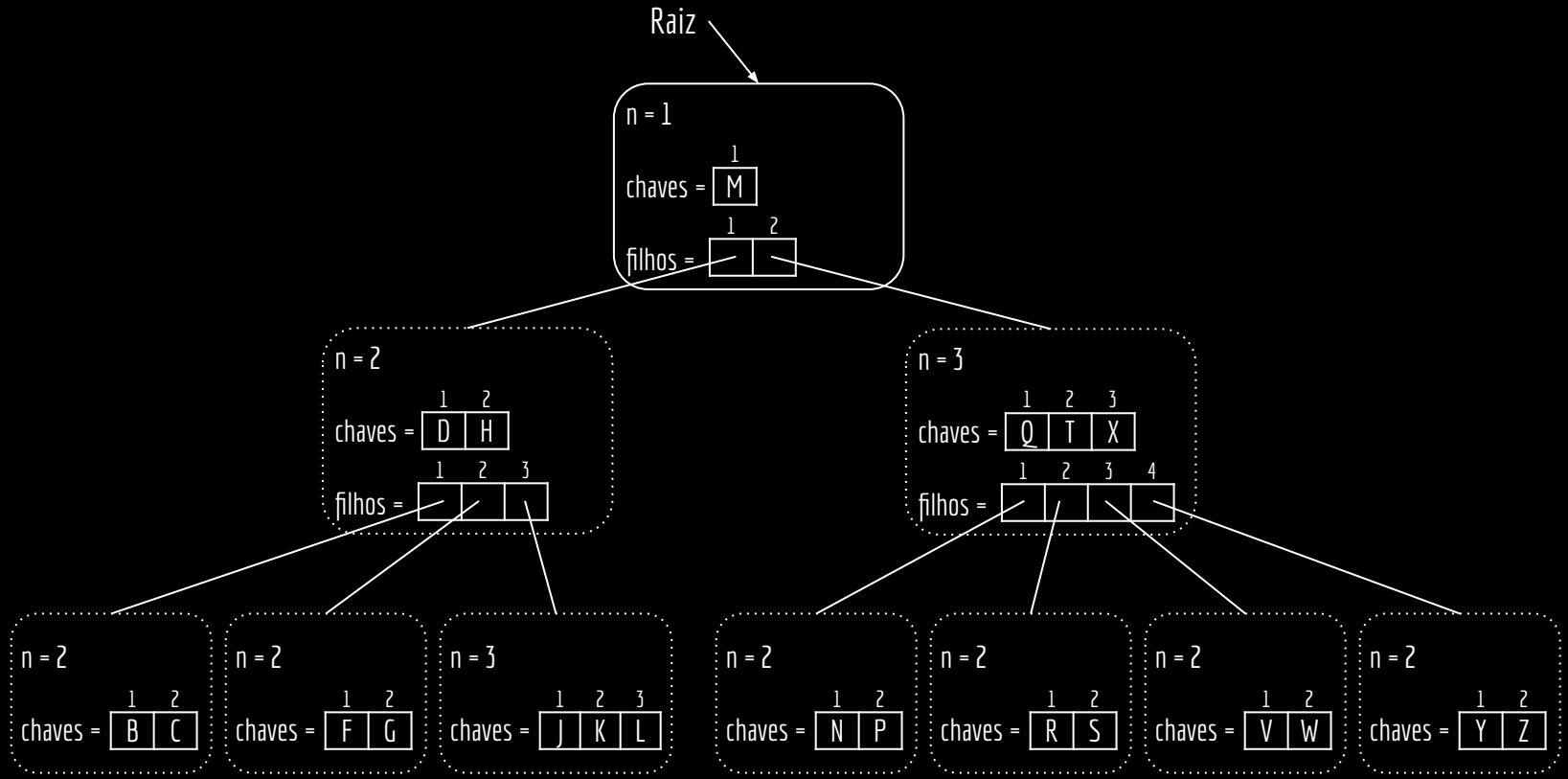
$$n \geq 1 + (t-1) 2 \frac{t^h - 1}{t-1}$$

$$n \geq 2t^h - 1$$

$$h \leq \log_t \frac{n+1}{2}$$

Como as operações dependem da algura h da árvore, as operações têm um custo O(h) = O($\log_t$ n).

Exemplo de Árvore B



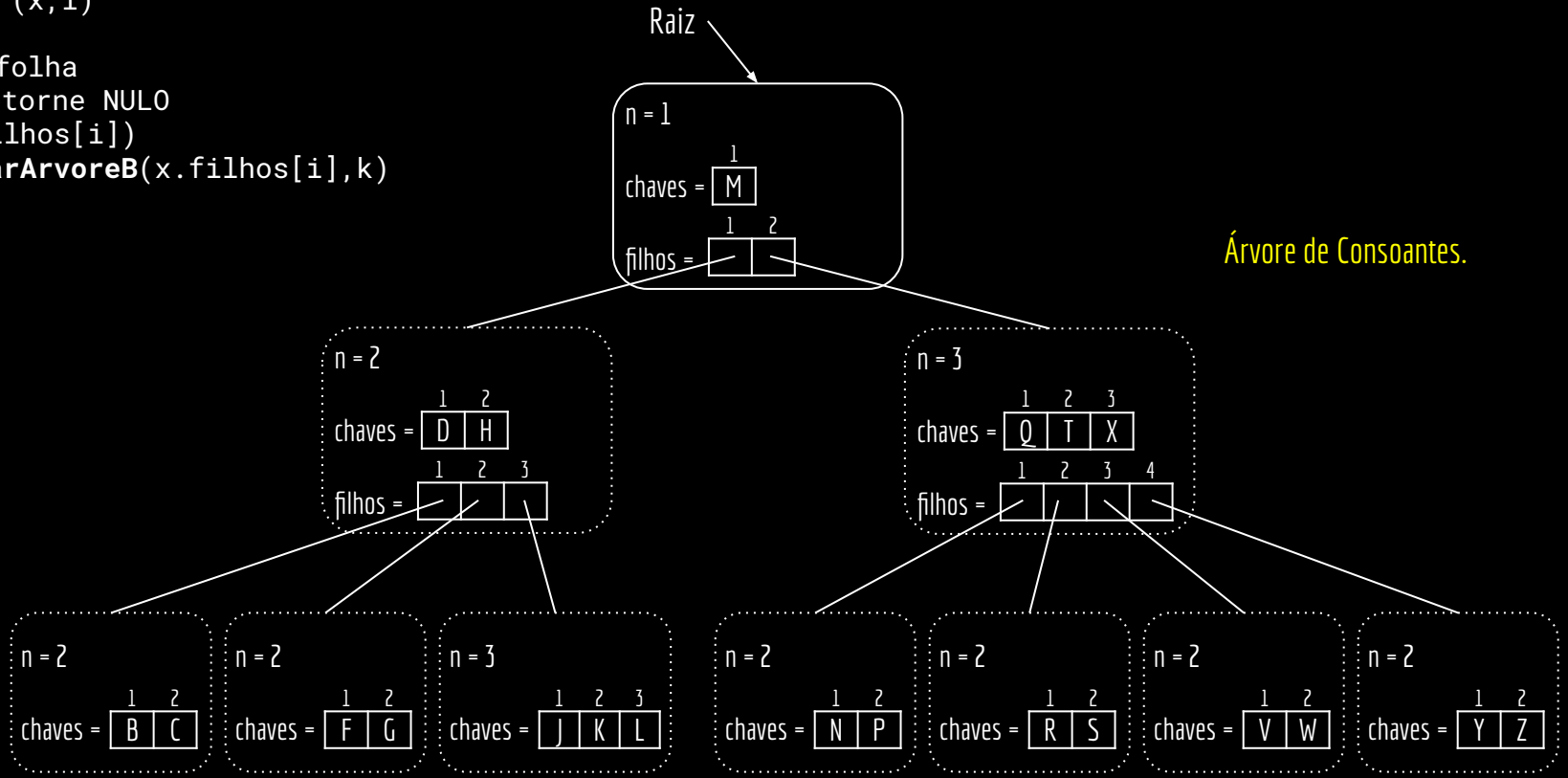
Busca

```
função buscarArvoreB(x,k)
entrada: nodo x a partir de onde começar a busca, e a chave k a ser buscada
saída:o nodo y que possui a chave, e o índice i da chave no nodo, ou NULO para
chave não encontrada.

i = 1
enquanto i < x.n e k > x.chaves[i]
     i = i+1
se i < x.n e k == x.chaves[i]
     retorne (x,i)
senão
     se x é folha
          retorne NULO
carregar(x.c[i])//carregar próximo nodo do armazenamento secundário
retorne buscarArvoreB(x.c[i],k)
```

```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
     i = i+1
se i ≤ x.n e k == x.chave[i]
     retorne (x,i)
senão
     se x é folha
          retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

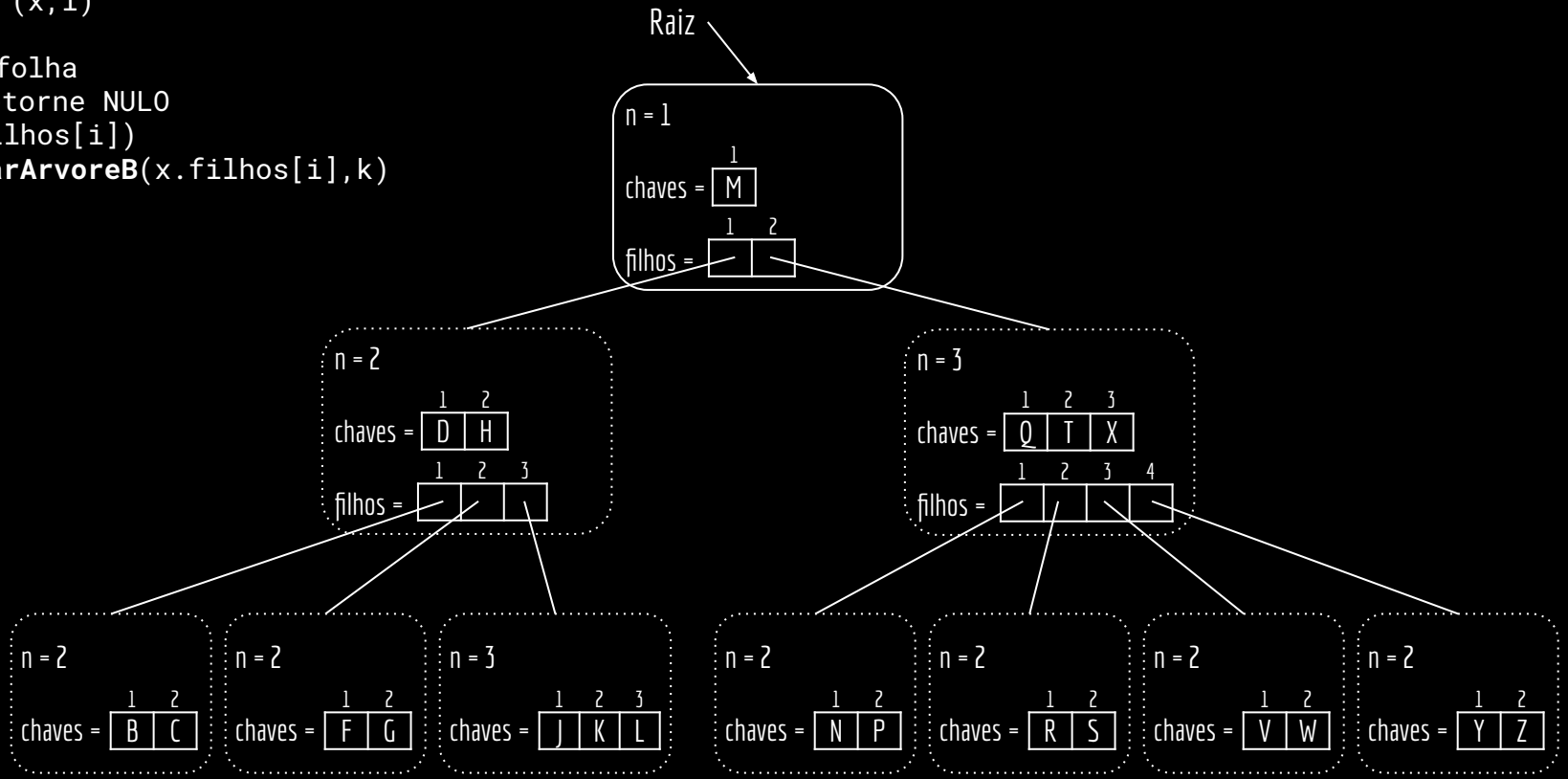
buscarArvoreB(raiz,R).



Árvore de Consoantes.

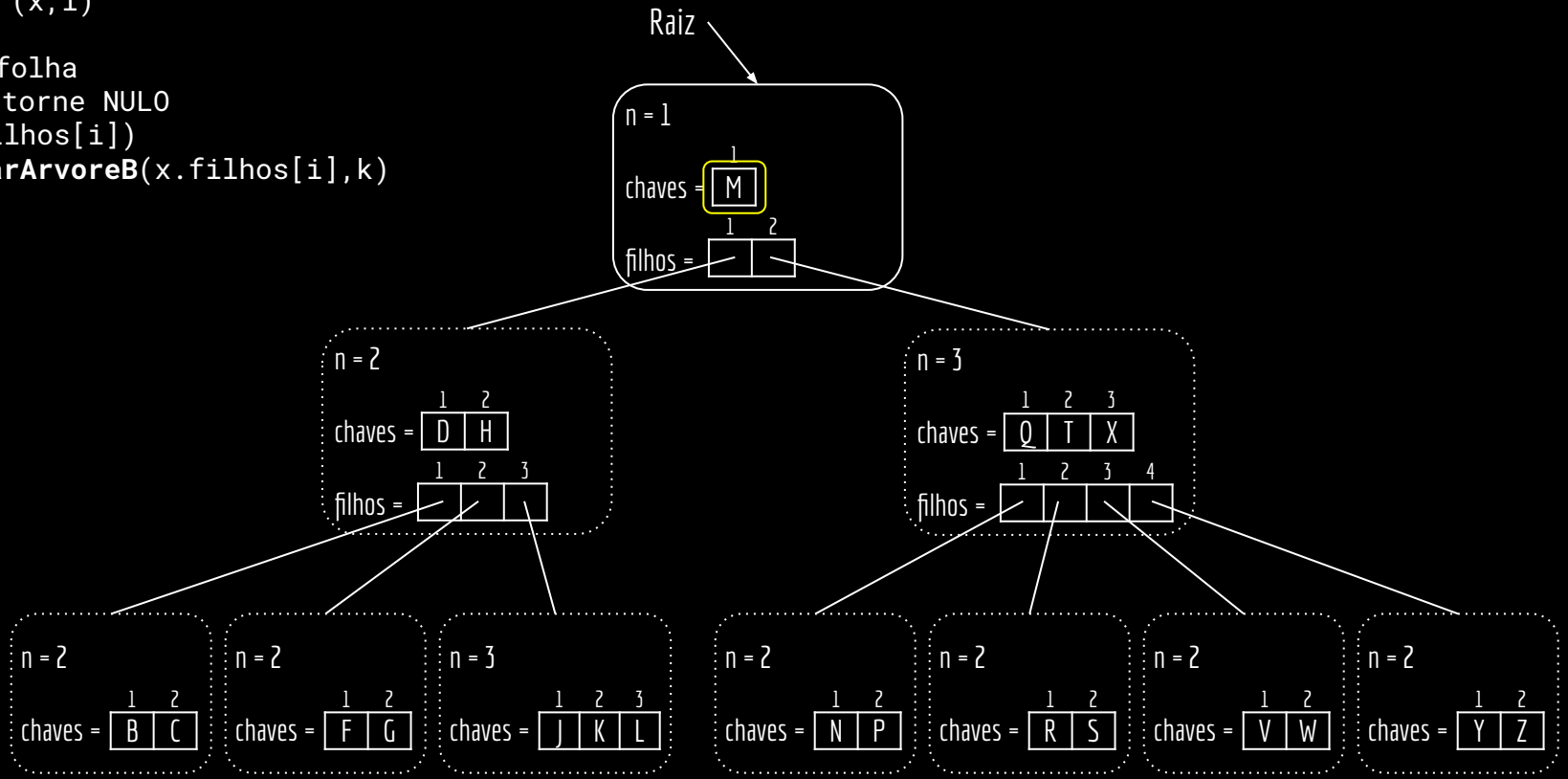
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
```



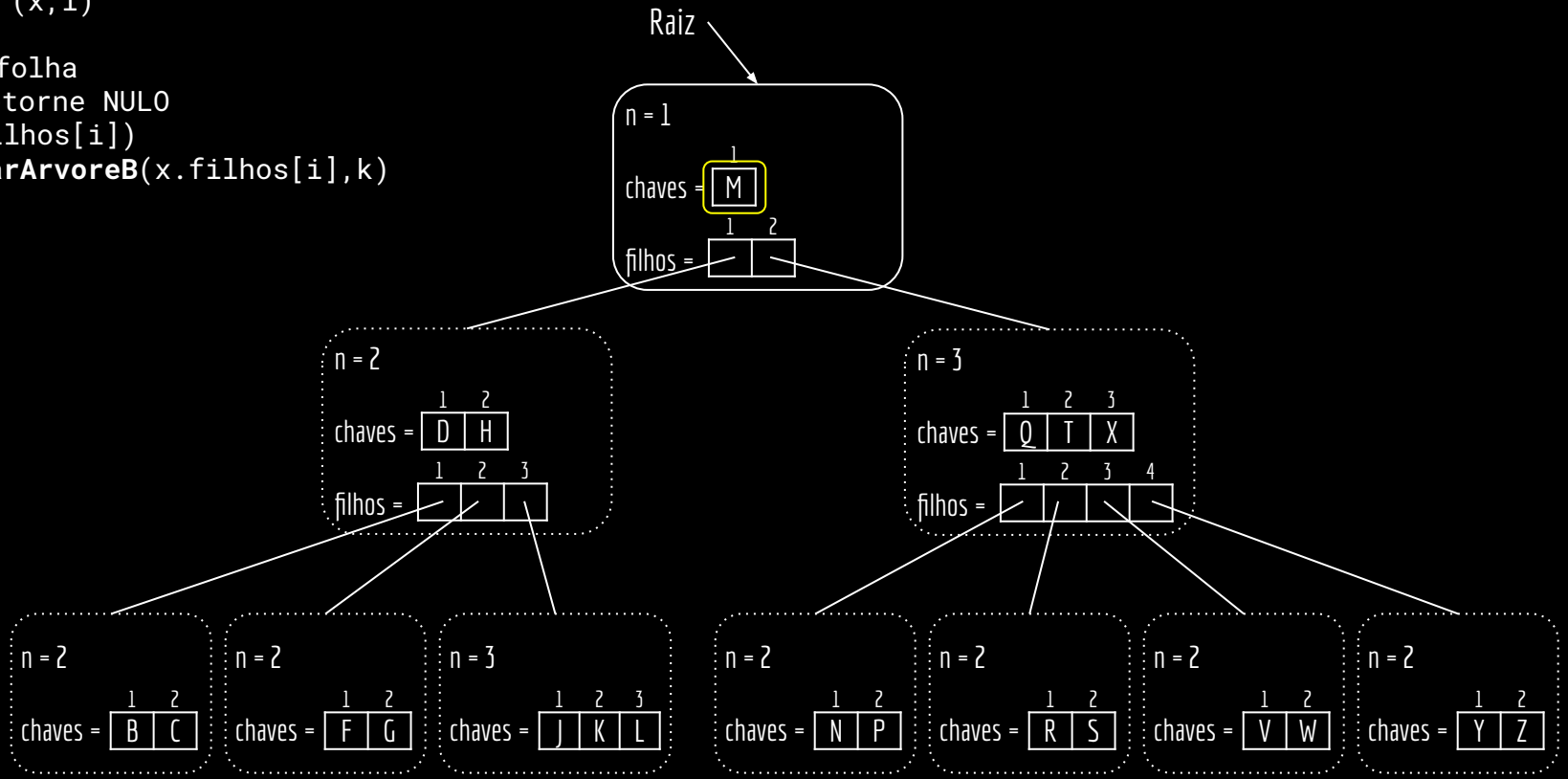
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
```



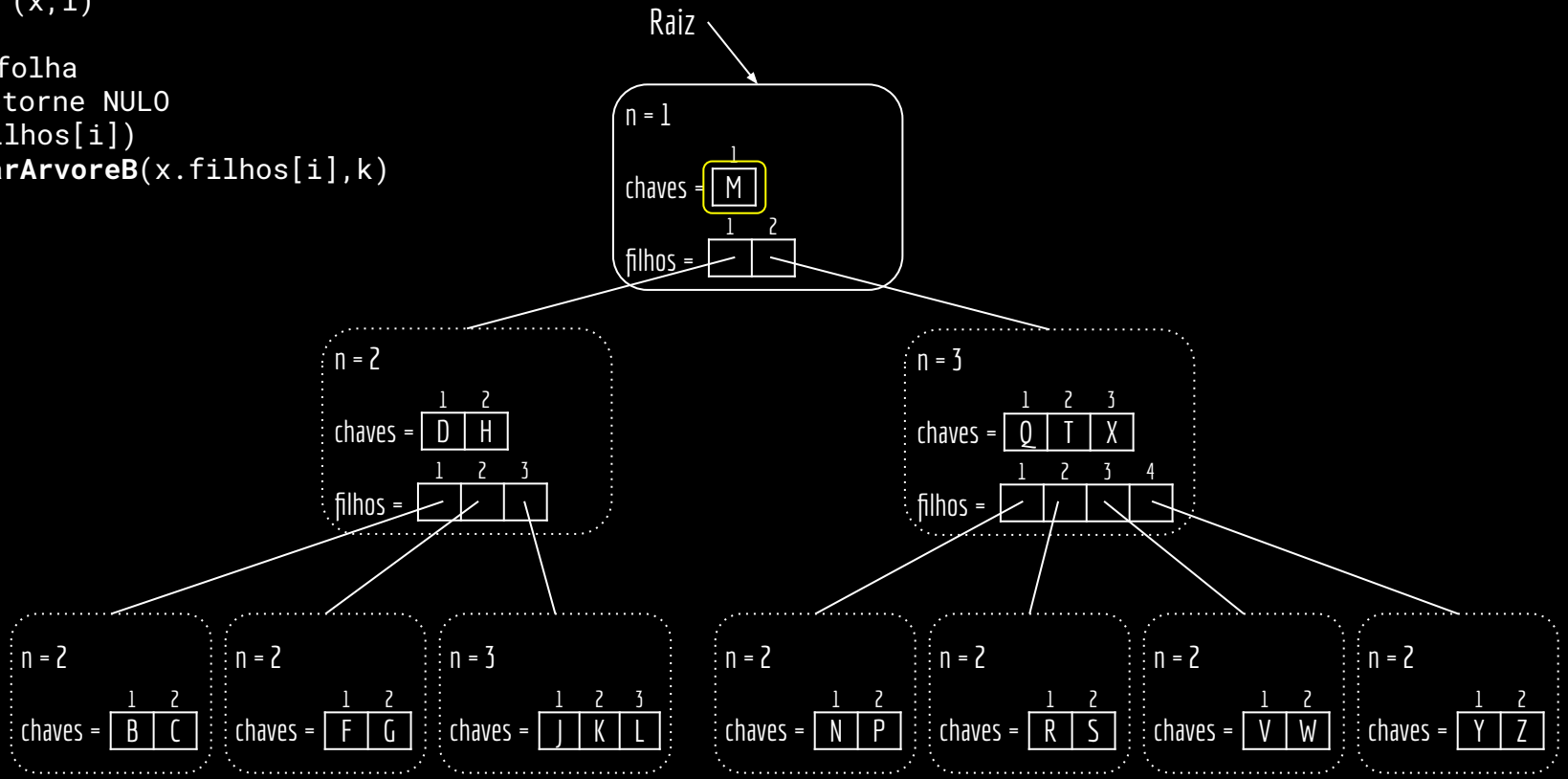
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```



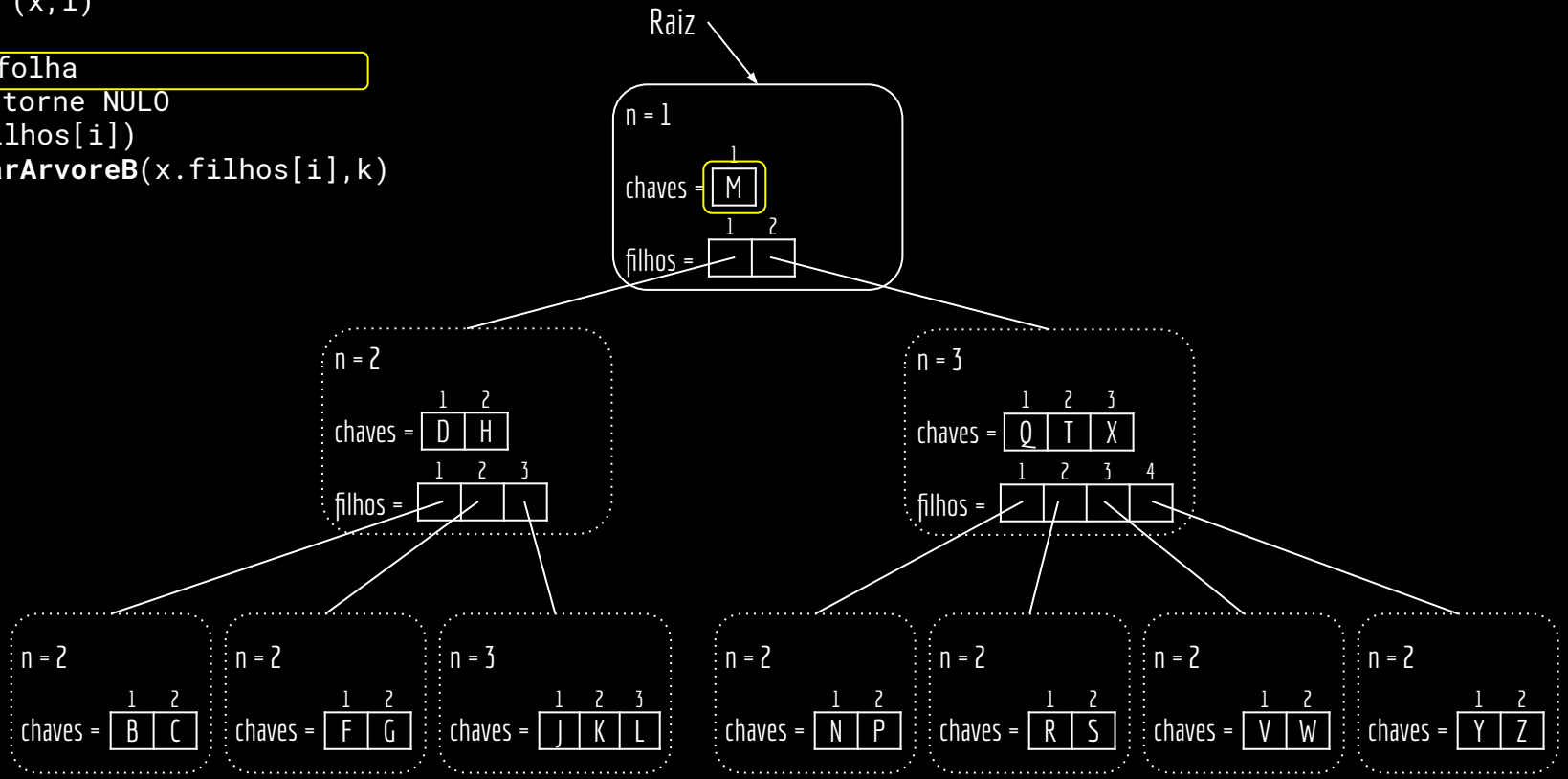
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```



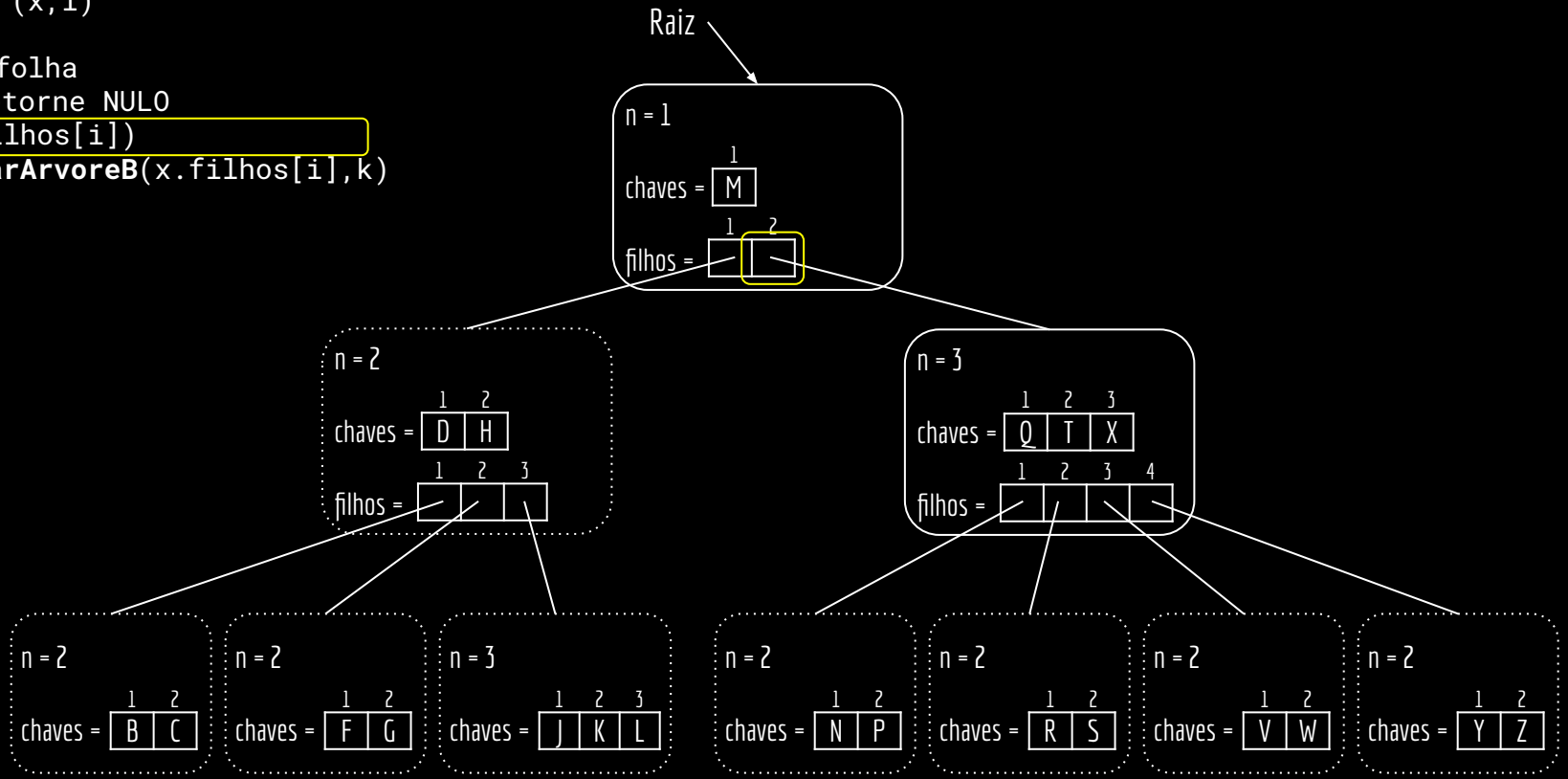
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```



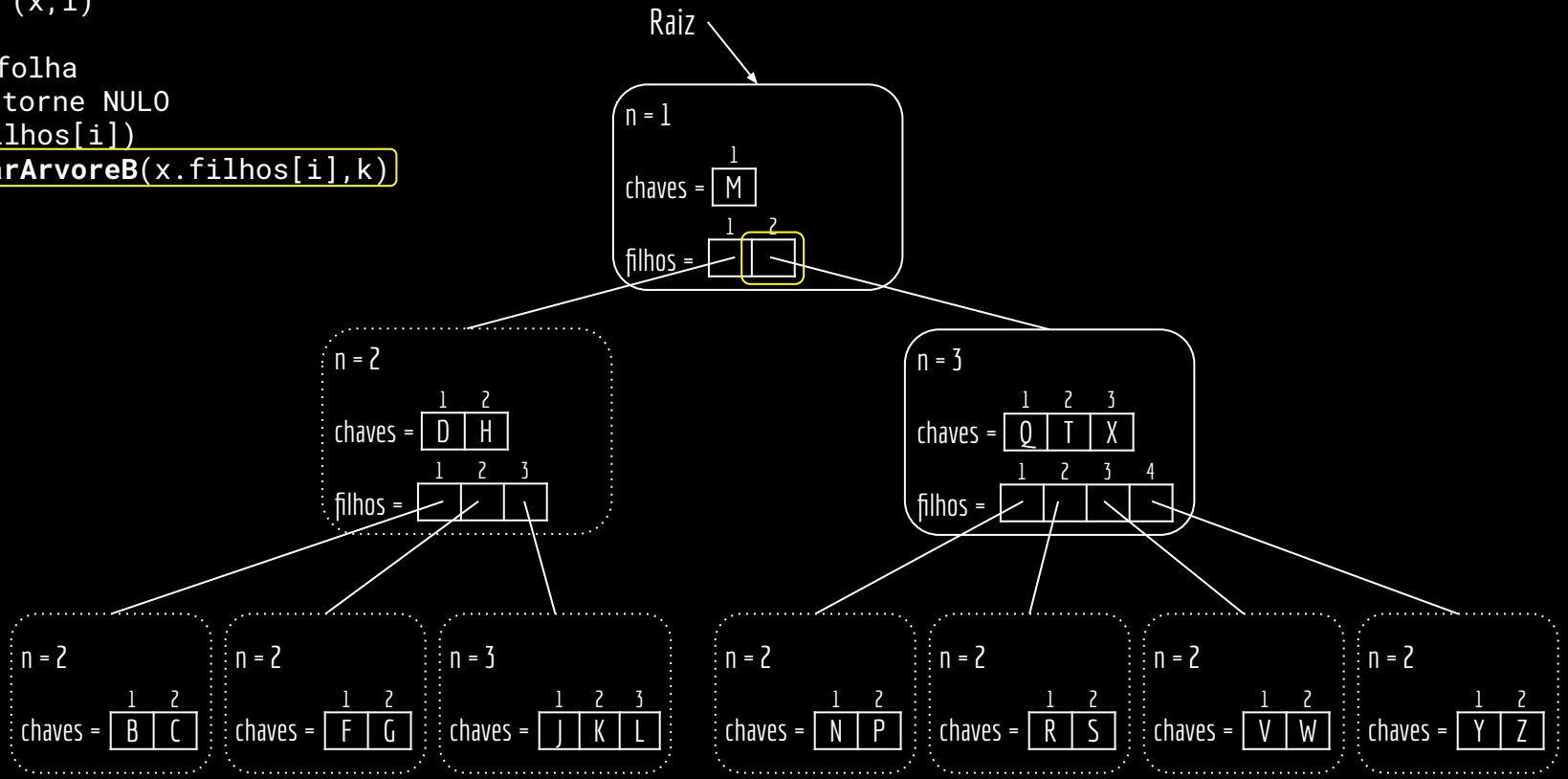
```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

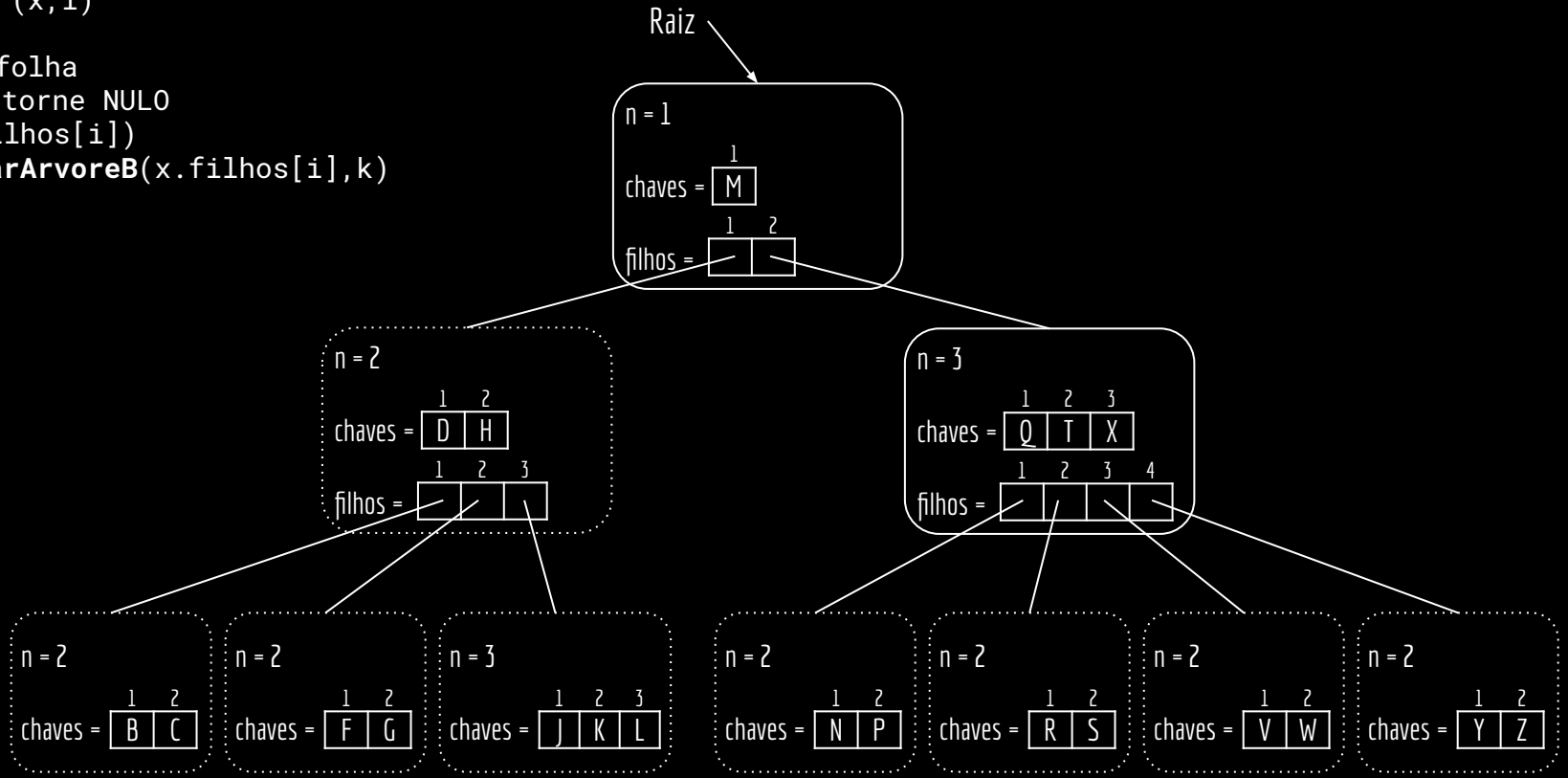
```
buscarArvoreB(raiz,R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

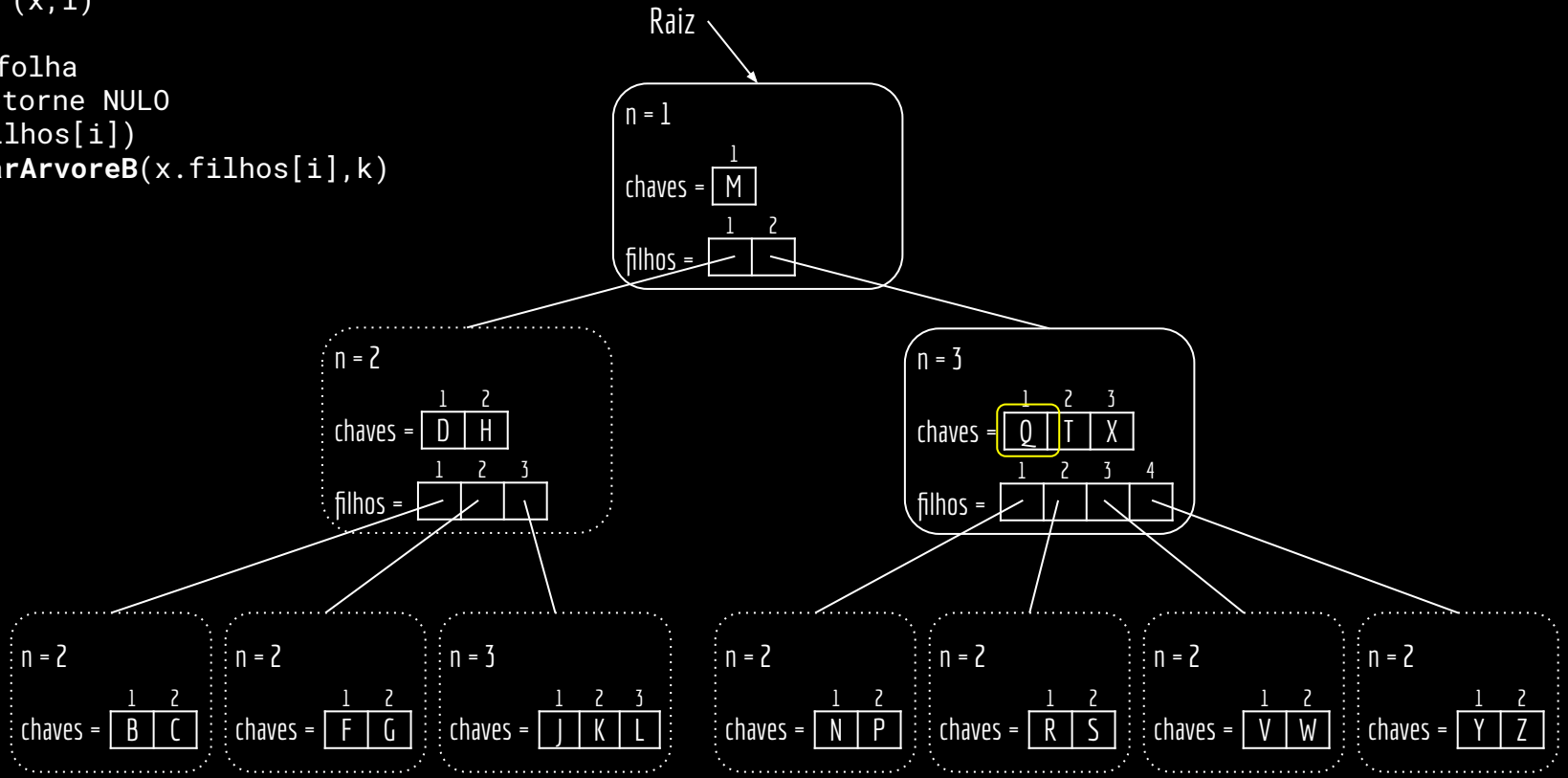
```
buscarArvoreB(x.filhos[2],R)
i
1
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

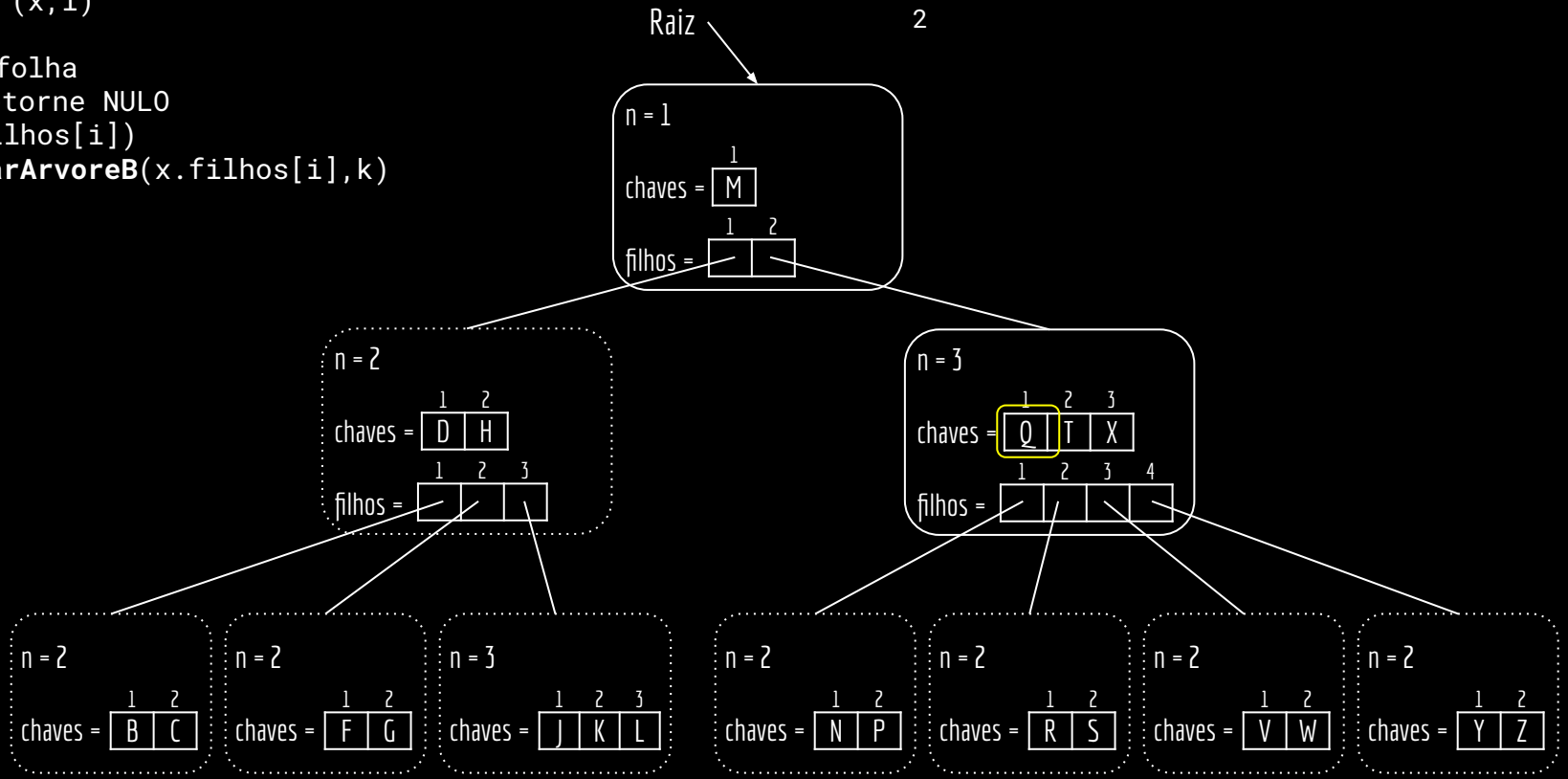
```
buscarArvoreB(x.filhos[2],R)
i
1
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

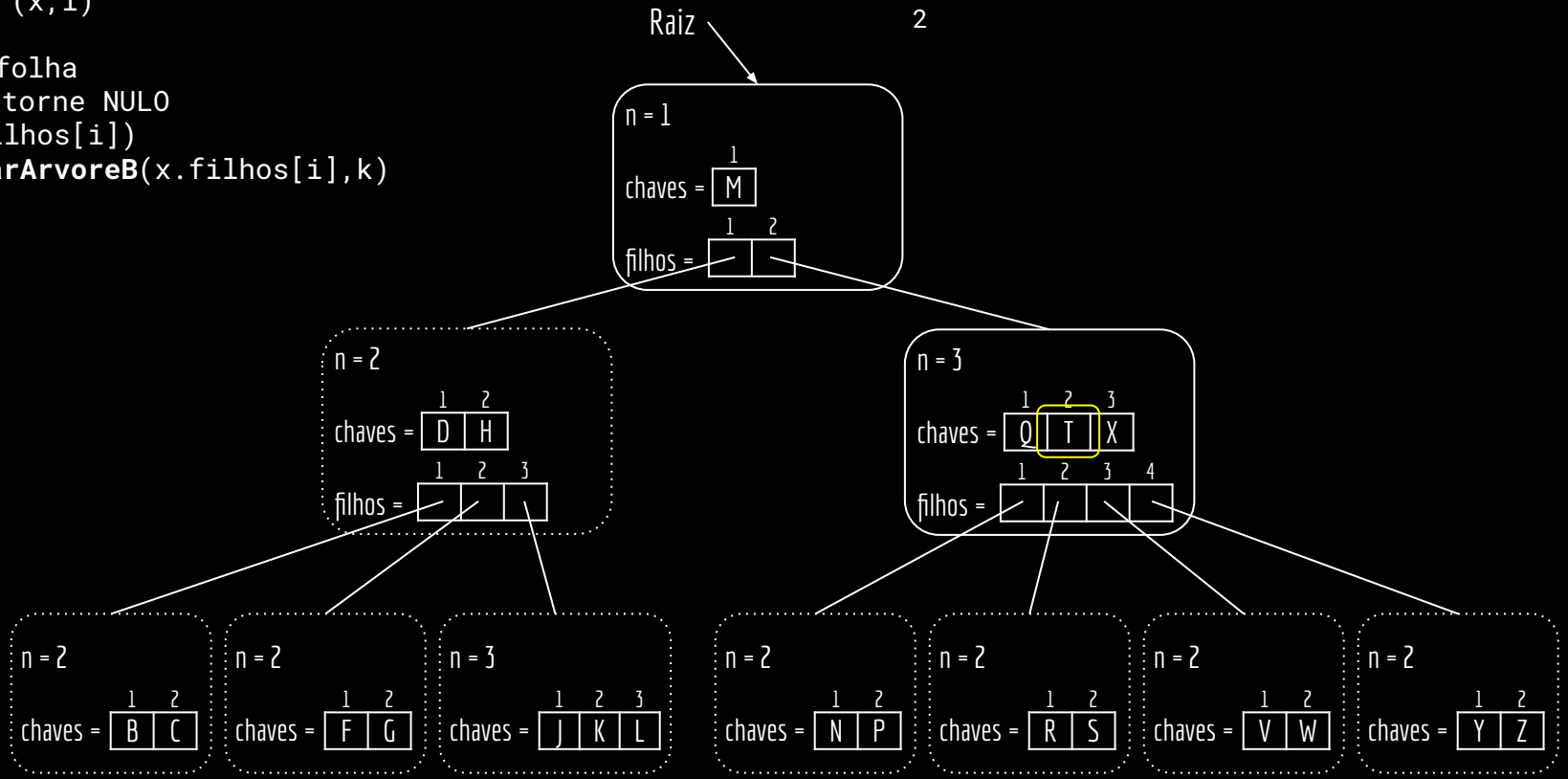
```
buscarArvoreB(x.filhos[2],R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

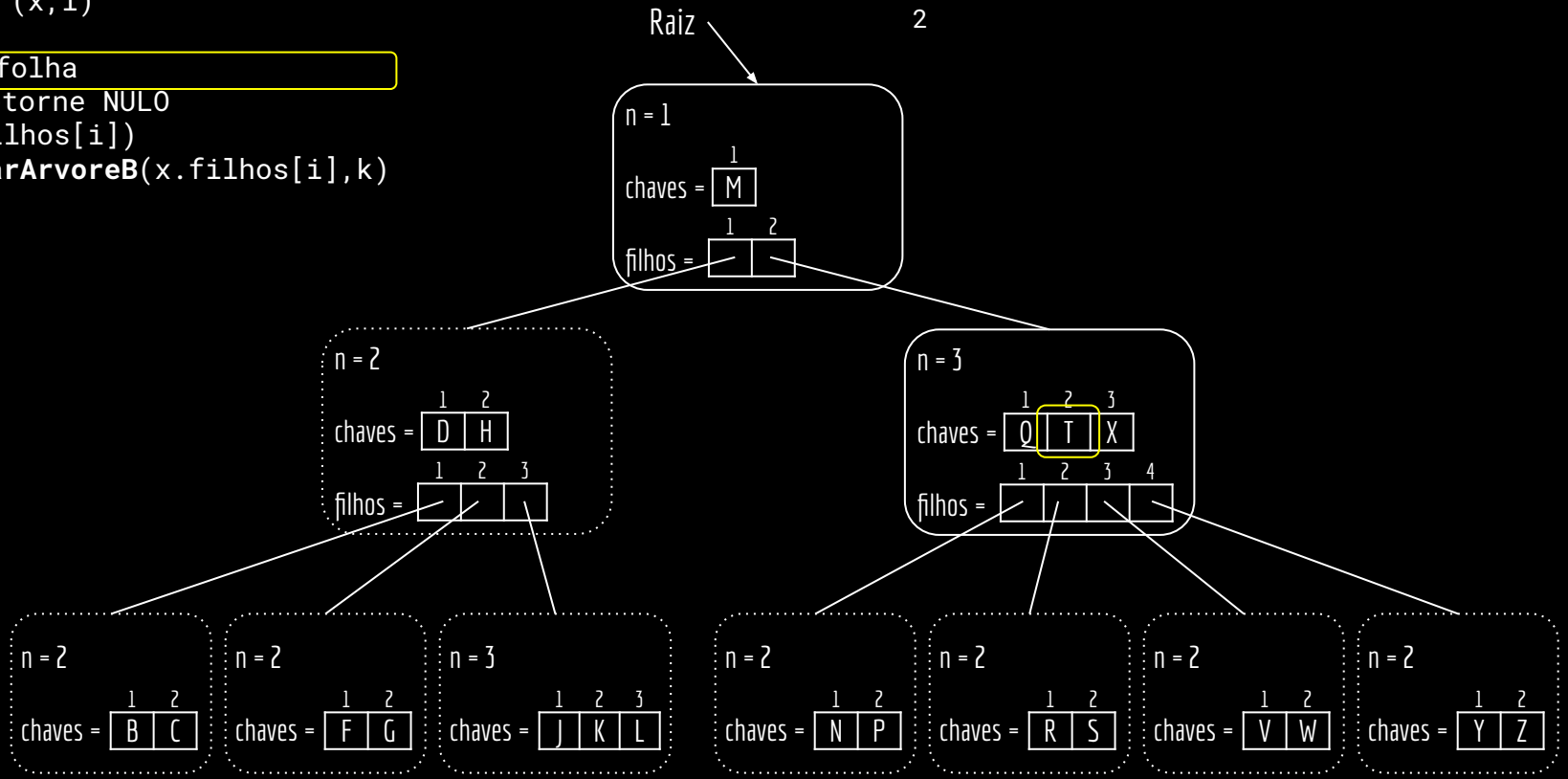
```
buscarArvoreB(x.filhos[2],R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

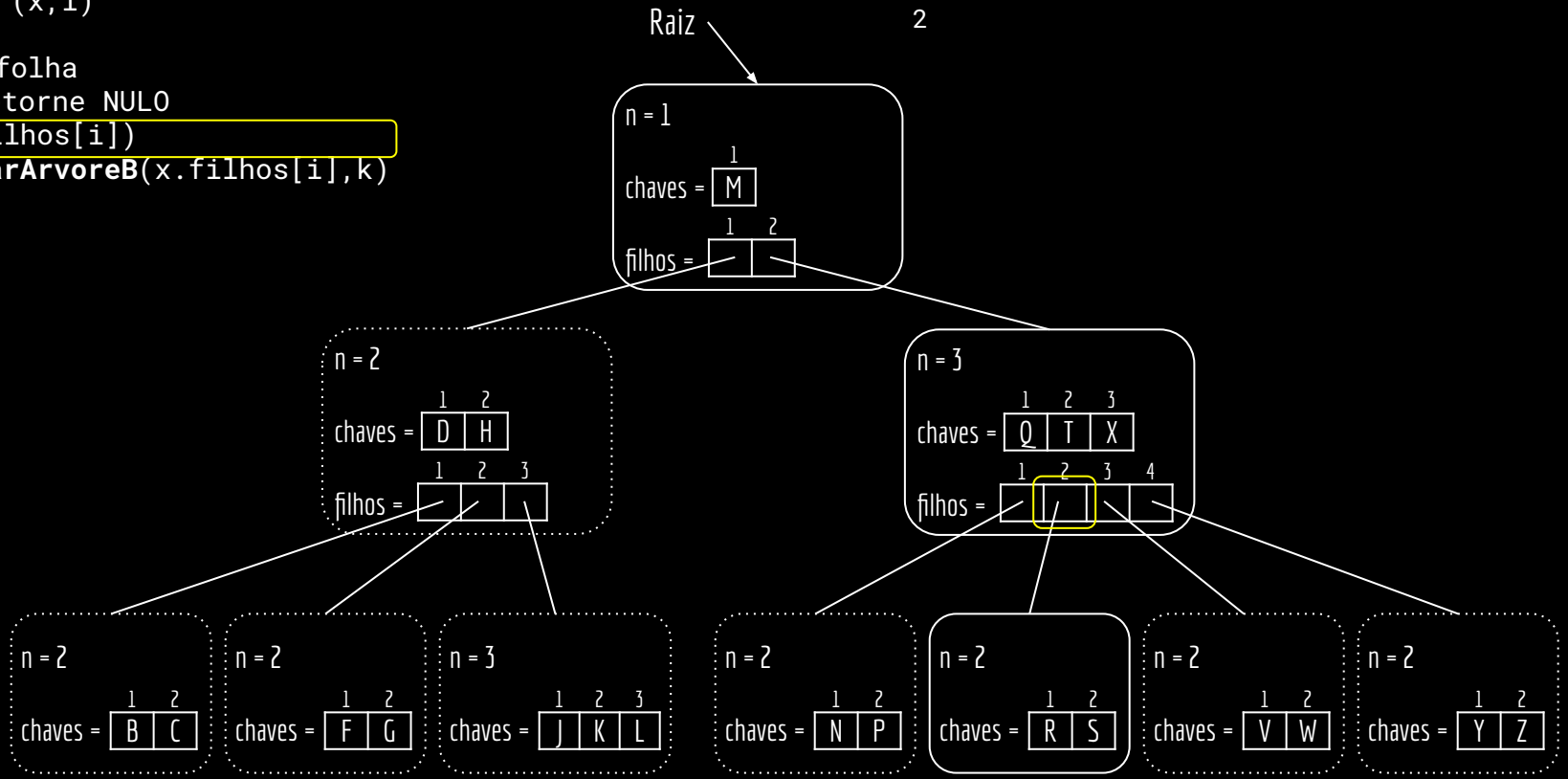
```
buscarArvoreB(x.filhos[2],R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

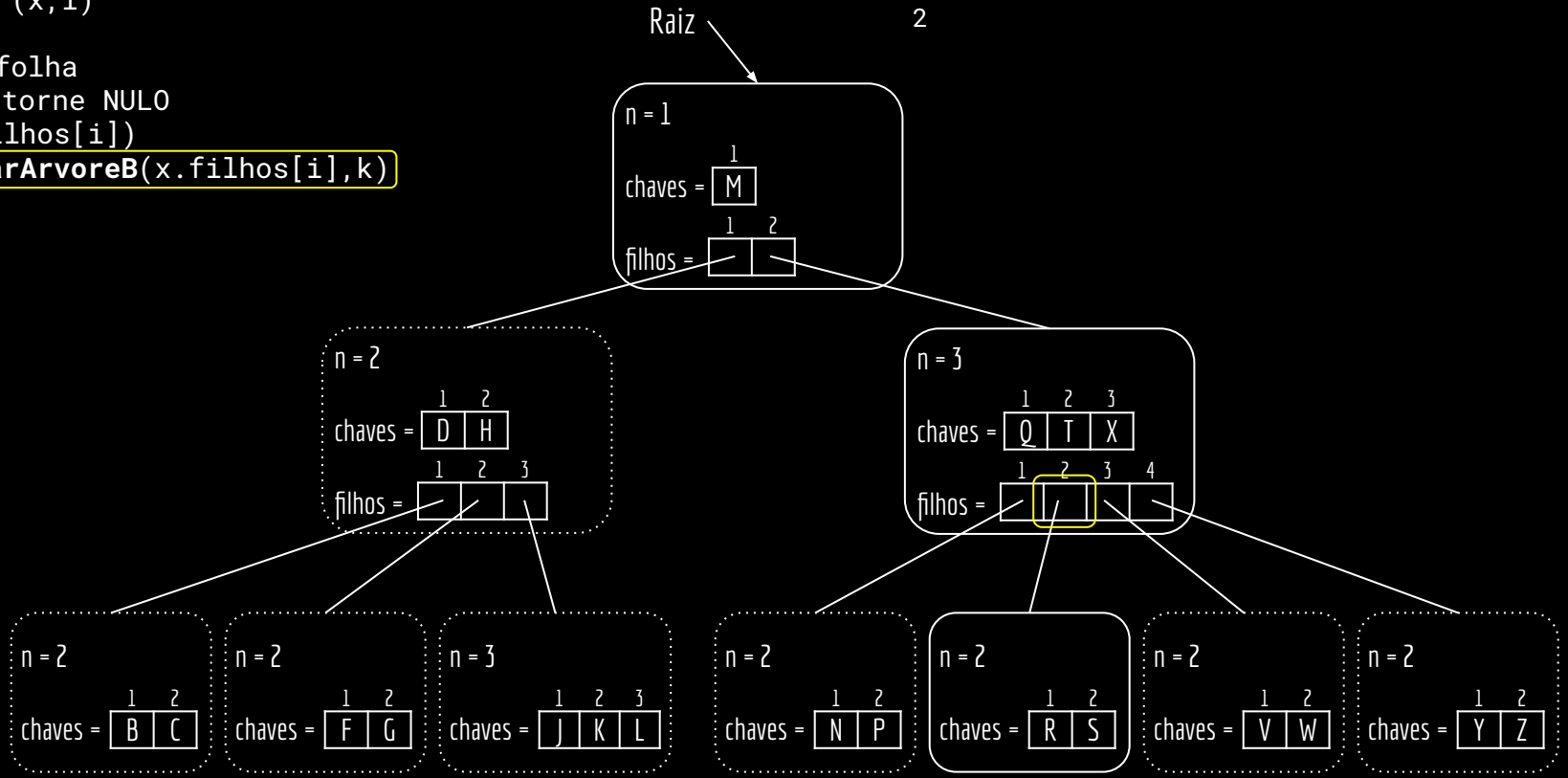
```
buscarArvoreB(x.filhos[2],R)
i
1
2
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
2
```

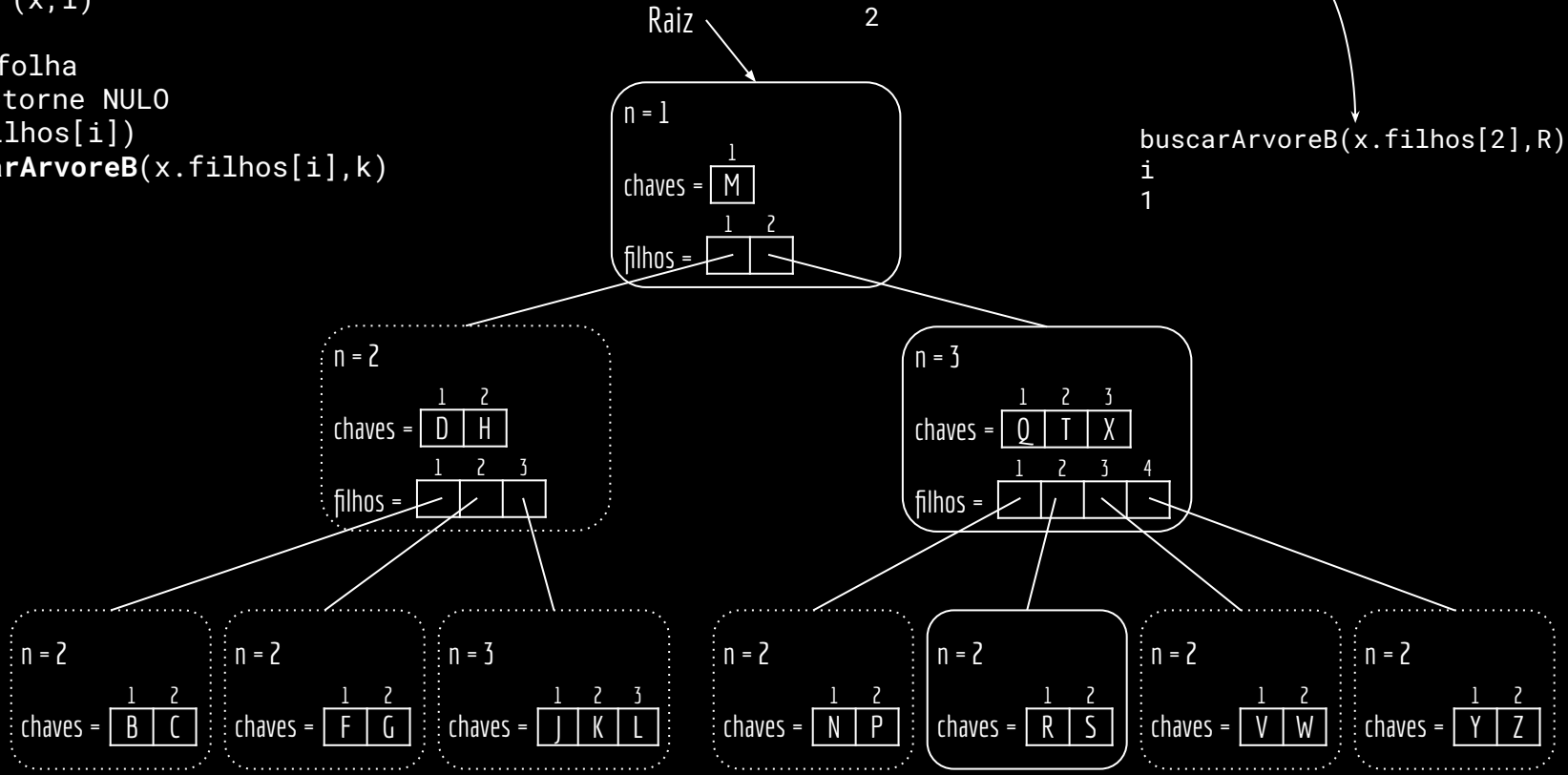


```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
```

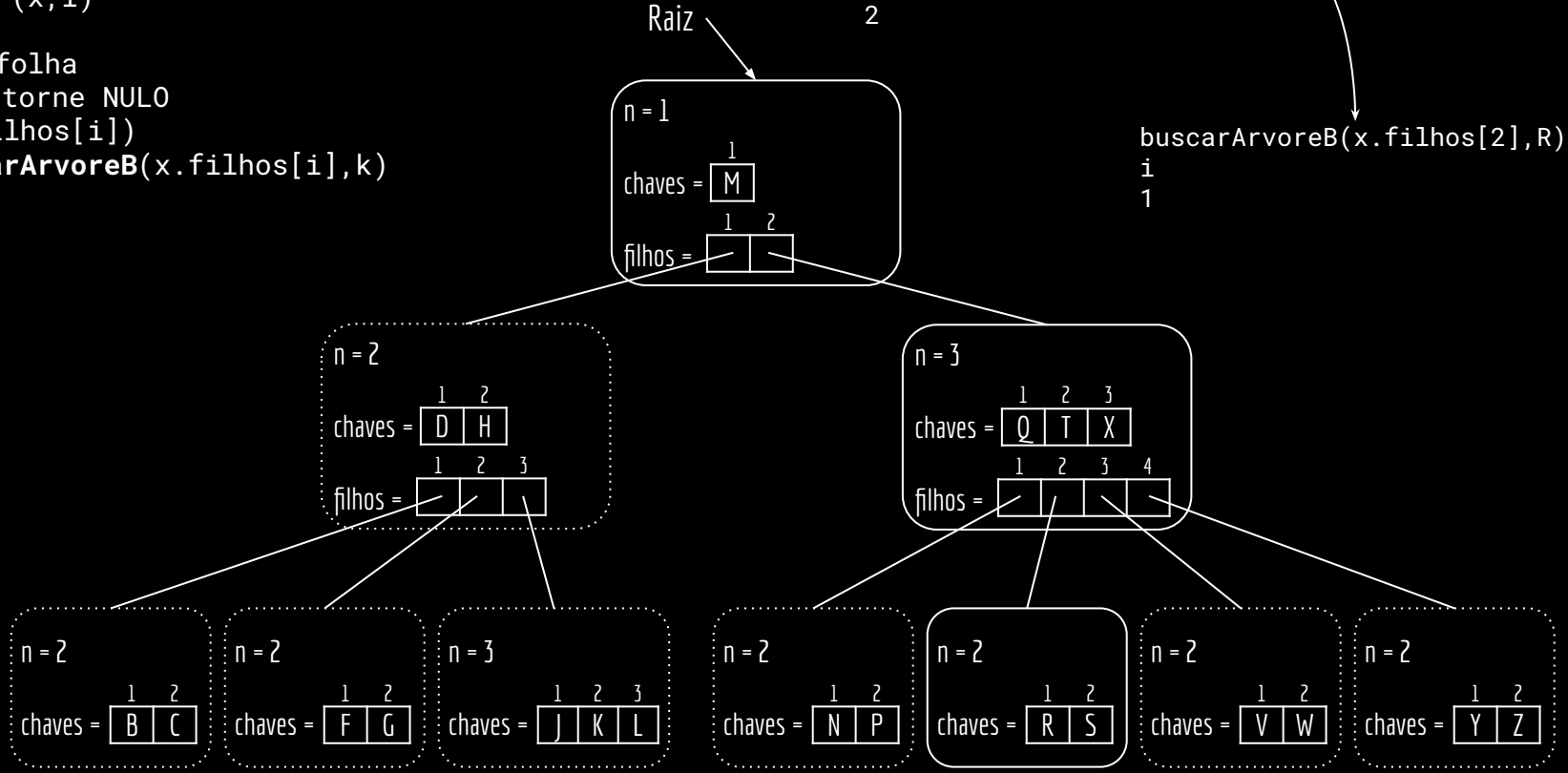


```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
```

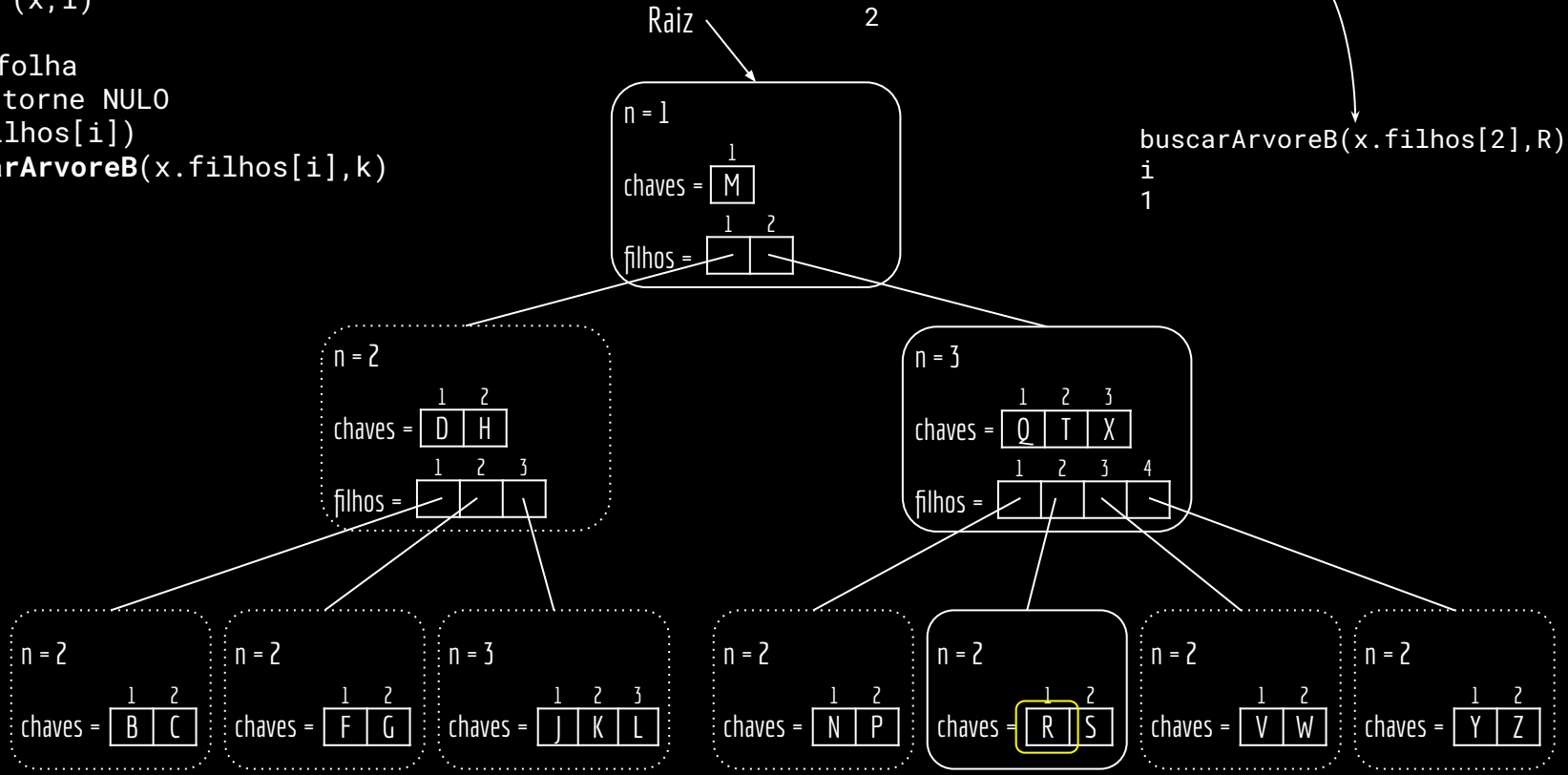


```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
2
```

```
buscarArvoreB(x.filhos[2],R)
i
1
```



```
função buscarArvoreB(x,k)
i = 1
enquanto i ≤ x.n e k > x.chave[i]
      i = i+1
se i ≤ x.n e k == x.chave[i]
      retorne (x,i)
senão
      se x é folha
            retorne NULO
carregar(x.filhos[i])
retorne buscarArvoreB(x.filhos[i],k)
```

```
buscarArvoreB(raiz,R)
i
1
2
```

(x,1)

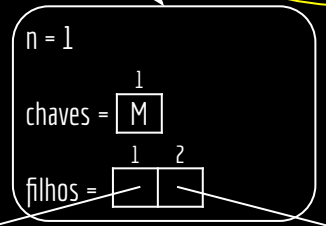
(x,1)

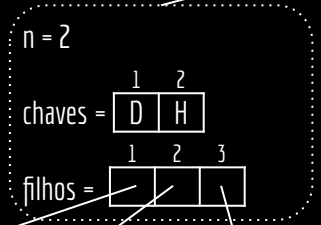
```
buscarArvoreB(x.filhos[2],R)
i
1
2
```

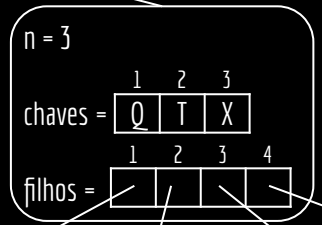
(x,1)

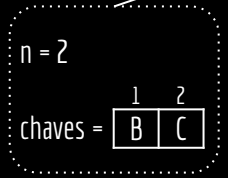
```
buscarArvoreB(x.filhos[2],R)
i
1
```

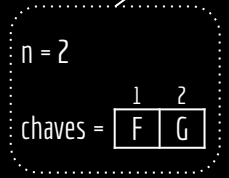
Raiz

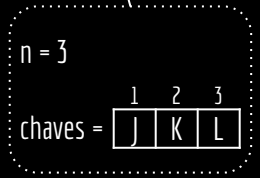


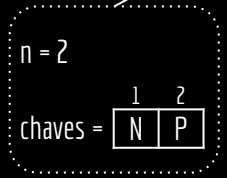


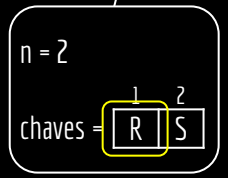


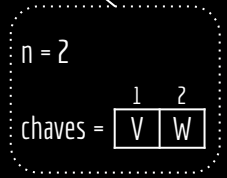


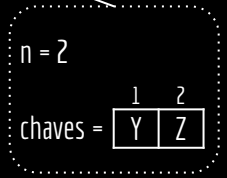












Custo

```
função buscarArvoreB(x,k)
entrada: nodo x a partir de onde começar a busca, e a chave k a ser buscada
saída:o nodo y que possui a chave, e o índice i da chave no nodo, ou NULO para
chave não encontrada.

i = 1
enquanto i ≤ x.n e k > x.chaves[i]
     i = i+1
se i ≤ x.n e k == x.chaves[i]
     retorne (x,i)
senão
     se x é folha
          retorne NULO
carregar(x.c[i])//carregar próximo nodo do armazenamento secundário
retorne buscarArvoreB(x.c[i],k)
```

Custo

```
função buscarArvoreB(x,k)
entrada: nodo x a partir de onde começar a busca, e a chave k a ser buscada
saída:o nodo y que possui a chave, e o índice i da chave no nodo, ou NULO para
chave não encontrada.

i = 1
enquanto i ≤ x.n e k > x.chaves[i]
     i = i+1
se i ≤ x.n e k == x.chaves[i]
     retorne (x,i)
senão
     se x é folha
          retorne NULO
carregar(x.c[i])//carregar próximo nodo do armazenamento secundário
retorne buscarArvoreB(x.c[i],k)
```

Busca sequencial. O algoritmo custa $O(t.log_t\ n)$ por conta disso. Podemos ter um custo $O(log_t\ n)$ ao trocar por uma busca binária. Fica como exercício.

Struct nodo

Como pode ser uma struct nodo típica?

Struct nodo

Como pode ser uma struct nodo típica?

```c
#include <stdbool.h>
#include <stddef.h>

#define GRAU_MINIMO 2  // grau mínimo t dos nodos

struct nodo {
   int n;  // número de chaves atualmente armazenadas
   int chaves[2 * GRAU_MINIMO - 1];
   struct nodo *filhos[2 * GRAU_MINIMO];
   bool ehFolha;  // true se é folha
};
```

Struct nodo

Como pode ser uma struct nodo típica?

```c
#include <stdbool.h>
#include <stddef.h>

#define GRAU_MINIMO 2  // grau mínimo t dos nodos

struct nodo {
   int n;  // número de chaves atualmente armazenadas
   int chaves[2 * GRAU_MINIMO - 1];
   struct nodo *filhos[2 * GRAU_MINIMO];
   bool ehFolha;  // true se é folha
};
```

Para possibilitar a carga dos dados do armazenamento secundário, precisaríamos ainda de algo para indicar em que bloco do armazenamento cada nodo filho se encontra.

Alocando uma árvore vazia

Para criar uma árvore vazia, utilize:

```
função criarArvoreB(T)

x = alocarNodo()
x.ehFolha = verdadeiro
x.n = 0
armazenar(x)
T.raiz = x
```

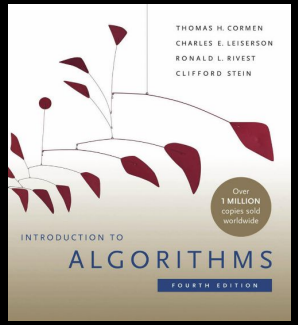
Armazena o nodo no armazenamento secundário.

Exercícios

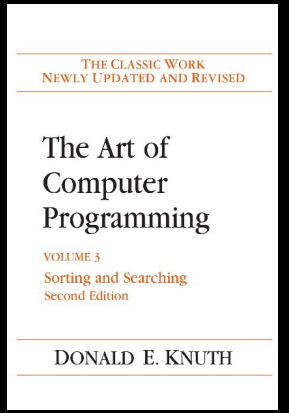
1. Modifique o algoritmo de busca para que seja utilizada uma busca binária.
2. Discorra, mesmo que informalmente, que quando usada a busca binária do item 1, buscar uma chave vai sempre custar $O(log_t n)$, independentemente do tamanho de t escolhido (Cormen et. al. 2022).
3. Implemente os algoritmos discutidos em C.
    a. Considere que tudo está na memória principal para não precisar criar arquivos no disco.
    b. Se você realmente quer uma aventura, implemente considerando que nodos podem estar no disco.
        i. Você vai precisar adicionar mais informações na struct nodo.

Referências

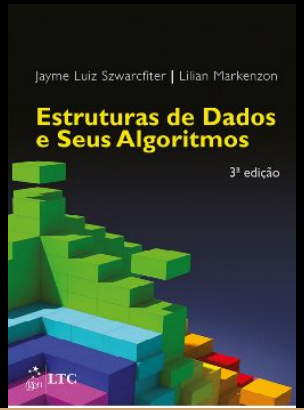
T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e Prática. 4a ed. 2022.



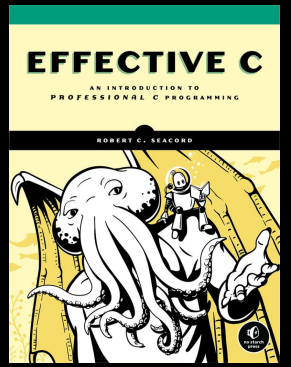
Knuth, D. The Art of Computer Programming: Volume 3: Sorting and Searching. 1998.



Szwarcfiter, Markenzon, L. Estruturas de dados e seus algoritmos. 2010.



Seacord, R. C. Effective C: An ntroduction to Professional C Programming. 2020.



Licença