Estruturas de Dados e Algoritmos II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2019/2020

Pseudo-código

Exemplo

PESQUISA-LINEAR(V, k)

```
n <- |V|                              // inicialização
i <- 1

while i <= n and V[i] != k do        // pesquisa
    i <- i + 1

if i <= n then                        // resultado:
    return i                          //  - sucesso

return -1                             //  - insucesso
```

`|V|`	nº de elementos de um vector — $O(1)$
`V[1..|V|]`	elementos do vector
`and` e `or`	só é avaliado o segundo operando se necessário
`variável.campo`	acesso a um campo de um “objecto”

Análise da complexidade (1)

Exemplo

Análise da complexidade temporal, no pior caso, da função PESQUISA-LINEAR, por linha de código

1. Obtenção da dimensão de um vector, afectação: operações com complexidade (temporal) constante

$$O(1) + O(1) = O(1)$$

2. Afectação: $O(1)$
3. Acessos a `i`, `n`, `V[i]` e `k`, comparações e saltos condicionais com complexidade constante

$$4\,O(1) + 2\,O(1) + 2\,O(1) = O(1)$$

Executada, no pior caso, $|\mathtt{V}|+1$ vezes

$$(|\mathtt{V}| + 1) \times O(1) = O(|\mathtt{V}|)$$

Análise da complexidade (2)

Exemplo

4. Acesso a `i`, soma e afectação: $O(1) + O(1) + O(1) = O(1)$
   Executada, no pior caso, `|V|` vezes

$$\texttt{|V|} \times O(1) = O(\texttt{|V|})$$

5. Acesso a `i` e `n`, comparação e salto condicional com complexidade constante

$$2\,O(1) + O(1) + O(1) = O(1)$$

6. Saída de função com complexidade constante: $O(1)$
7. Saída de função com complexidade constante: $O(1)$

Análise da complexidade (3)

Exemplo

Juntando tudo

$$
\begin{aligned}
O(1) + O(1) + O(|\mathtt{V}|) + O(|\mathtt{V}|) + O(1) + \max\{O(1), O(1)\} &= \\
= 4\, O(1) + 2\, O(|\mathtt{V}|) &= \\
= O(|\mathtt{V}|)
\end{aligned}
$$

No pior caso, a função PESQUISA-LINEAR tem complexidade temporal linear na dimensão do vector `V`

Se n representar a dimensão do vector `V`, o tempo $T(n)$ que a função demora a executar tem complexidade linear em n

$$T(n) = O(n)$$

Isto significa que o tempo que a função demora a executar varia linearmente com a dimensão do *input*

A notação O (1)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0}\ 0 \leq f(n) \leq c\, g(n)\}$$

- $O(n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0}\ 0 \leq f(n) \leq c\, n\}$

  $n = O(n) \qquad 2n + 5 = O(n) \qquad \log n = O(n) \qquad n^2 \neq O(n)$

- $O(n^2) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0}\ 0 \leq f(n) \leq c\, n^2\}$

  $n^2 = O(n^2) \qquad 4n^2 + n = O(n^2) \qquad n = O(n^2) \qquad n^3 \neq O(n^2)$

- $O(\log n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0}\ 0 \leq f(n) \leq c \log n\}$

  $1 + \log n = O(\log n) \qquad \log n^2 = O(\log n) \qquad n \neq O(\log n)$

Escreve-se $f(n) = O(g(n))$ em vez de $f(n) \in O(g(n))$

Lê-se $f(n)$ é O de $g(n)$

A notação O (2)

$$\Omega(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq c\, g(n) \leq f(n)\}$$

$$n = \Omega(n) \qquad n^2 = \Omega(n) \qquad \log n \neq \Omega(n^2)$$

$$\Theta(g(n)) = \{f(n) : \exists_{c_1,c_2,n_0>0} \text{ t.q. } \forall_{n\geq n_0}\ 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\}$$

$$3n^2 + n = \Theta(n^2) \qquad n \neq \Theta(n^2) \qquad n^2 \neq \Theta(n)$$

$$o(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n\geq n_0}\ 0 \leq f(n) < c\, g(n)\}$$

$$n = o(n^2) \qquad n^2 \neq o(n^2)$$

$$\omega(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n\geq n_0}\ 0 \leq c\, g(n) < f(n)\}$$

$$n = \omega(\log n) \qquad n^2 = \omega(\log n) \qquad \log n \neq \omega(\log n)$$

A notação O (3)

Traduzindo...

$f(n) = O(g(n))$ — $f(n)$ não cresce mais depressa que $g(n)$

$f(n) = o(g(n))$ — $f(n)$ cresce mais devagar que $g(n)$

$f(n) = \Omega(g(n))$ — $f(n)$ não cresce mais devagar que $g(n)$

$f(n) = \omega(g(n))$ — $f(n)$ cresce mais depressa que $g(n)$

$f(n) = \Theta(g(n))$ — $f(n)$ e $g(n)$ crescem com o mesmo ritmo

Informação persistente (1)

Enquadramento

- Estruturas de dados em memória central desaparecem quando programa termina
- Volume dos dados pode não permitir
  - o armazenamento em memória central
  - o seu processamento sempre que é necessário aceder-lhes
- Dados persistentes, em memória secundária, requerem estruturas de dados persistentes

Condicionantes

- Acessos a memória secundária (10^{-3} s) muito mais caros que acessos à memória central (10^{-9} s)
- Transferências entre a memória central e a memória secundária processadas por páginas (4096 *bytes* é uma dimensão comum)

Informação persistente (2)

Dados em memória secundária

Estratégia

Minimizar o número de acessos a memória secundária

- Adaptando as estruturas de dados
- Usando estruturas de dados especialmente concebidas

  Em ambos os casos, procura-se tirar o maior partido possível do conteúdo das páginas acedidas

- Fazendo *cacheing* da informação

Cuidados

Garantir que a informação em memória secundária se mantém actualizada

- Operações só ficam completas quando as alterações são escritas na memória secundária

B-Trees

B-Trees

Objectivos

Grandes quantidades de informação

Armazenamento em memória secundária

Indexação eficiente

Minimização de acessos a memória secundária

B-Trees

Características (1)

São árvores

Princípios semelhantes aos das árvores binárias de pesquisa

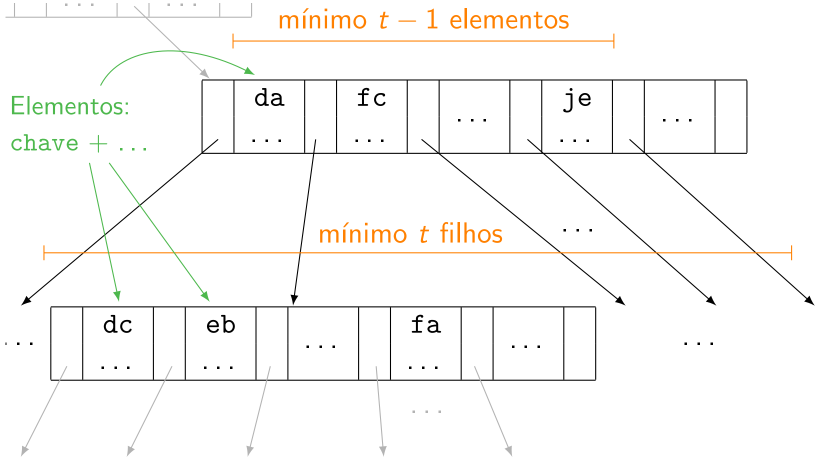
Perfeitamente equilibradas

Nós com número variável de filhos (pelo menos 2)

Nós com número variável de elementos

B-Trees

Estrutura dos nós internos (exceptuando a raiz)



B-Trees

Características (2)

Os nós internos das *B-trees* (excepto a raiz) têm, pelo menos, $t \geq 2$ filhos

t é o grau (de ramificação) mínimo de uma *B-tree*

A ordem de uma *B-tree* é $m = 2t$

Cada nó tem capacidade para $2t - 1$ elementos

Ocupação de um nó (excepto a raiz)

- entre $t - 1$ e $2t - 1$ elementos
- entre t e $2t$ filhos (excepto as folhas)

Ocupação da raiz (de uma *B-tree* não vazia)

- entre 1 e $2t - 1$ elementos
- entre 2 e $2t$ filhos (excepto se for folha)

B-Trees

Características (3)

Um nó interno com e elementos tem $e+1$ filhos

Em todos os nós, verifica-se:

$$\mathit{chave}(\text{elemento}_1) \leq \mathit{chave}(\text{elemento}_2) \leq \ldots \leq \mathit{chave}(\text{elemento}_e)$$

Em todos os nós internos, verifica-se:

$$\mathit{chaves}(\text{filho}_1) \leq \mathit{chave}(\text{elemento}_1) \leq \mathit{chaves}(\text{filho}_2) \leq$$
$$\leq \mathit{chave}(\text{elemento}_2) \leq \ldots \leq \mathit{chave}(\text{elemento}_e) \leq \mathit{chaves}(\text{filho}_{e+1})$$

Todas as folhas estão à mesma profundidade

B-Trees

Implementação

Conteúdo de um nó	(campo)
► ocupação	`n`
► elementos ($2t - 1$)	`key[1 .. 2t-1]`
► filhos ($2t$)	`c[1 .. 2t]`
► é-folha?	`leaf`

Um nó ocupa uma, duas páginas (do disco, do sistema de ficheiros, ...)

O valor de t depende do espaço ocupado pelos elementos e da dimensão pretendida para um nó

A raiz é mantida sempre em memória

B-TREE-CREATE(T)

```
x <- ALLOCATE-NODE()          // cria um novo nó
x.leaf <- TRUE                //   sem filhos
x.n <- 0                      //   nem elementos
DISK-WRITE(x)                 // e guarda-o em disco
T.root <- x                   // este nó é a raiz da
                              // nova B-tree
```

(*Introduction to Algorithms*, Cormen et al.)

B-TREE-SEARCH(x, k)

```
i <- 1
while i <= x.n and k > x.key[i] do
    i <- i + 1
if i <= x.n and k = x.key[i] then
    return (x, i)
if x.leaf then
    return NIL
DISK-READ(x.c[i])
return B-TREE-SEARCH(x.c[i], k)
```

Pesquisa (recursiva) do elemento com chave k na subárvore cuja raiz é o nó x

Assume que x já está em memória quando a função é chamada

Altura máxima de uma *B-tree*

Nível	Número mínimo de nós	Número mínimo de elementos
0	1	1
1	2	$2\,(t-1)$
2	$2t$	$2t\,(t-1)$
3	$2t^2$	$2t^2(t-1)$
4	$2t^3$	$2t^3\,(t-1)$
	$\vdots$	
h	$2t^{h-1}$	$2t^{h-1}\,(t-1)$

Número de elementos de uma árvore com altura h

$$n \geq 1+\sum_{i=0}^{h-1} 2t^i(t-1) = 1+2(t-1)\frac{1-t^h}{1-t} = 1-2(1-t^h) = 2t^h-1$$

Altura de uma árvore com n elementos

$$n \geq 2t^h - 1 \quad \equiv \quad t^h \leq \frac{n+1}{2} \quad \equiv \quad h \leq \log_t \frac{n+1}{2}$$

B-Trees

Comportamento da pesquisa

Altura de uma árvore com n elementos

$$h \leq \log_t \frac{n+1}{2} = O(\log_t n)$$

Número de nós acedidos no pior caso

$$O(h) = O(\log_t n)$$

Complexidade temporal da pesquisa no pior caso

$$O(t \log_t n)$$

Alturas de árvores

Elementos	abp	*B-tree*			
		$t = 32$		$t = 64$	
	mínima	mínima	máxima	mínima	máxima
10^6	19	3	3	2	3
10^9	29	4	5	4	4
10^{12}	39	6	7	5	6

B-TREE-INSERT(T, k)

```
r <- T.root
if r.n = 2t - 1 then              // se a raiz está cheia...
    s <- ALLOCATE-NODE()
    T.root <- s                   // cria uma nova raiz...
    s.leaf <- FALSE
    s.n <- 0
    s.c[1] <- r                   // de que a antiga é filho
    B-TREE-SPLIT-CHILD(s, 1)  // e explode a antiga raiz
    B-TREE-INSERT-NONFULL(s, k)
else
    B-TREE-INSERT-NONFULL(r, k)
```

A inserção é efectuada numa única passagem pela árvore

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

```
y <- x.c[i]                          // nó a explodir (filho i)
z <- ALLOCATE-NODE()                 // novo filho i+1
z.leaf <- y.leaf
z.n <- t - 1
for j <- 1 to t - 1 do               // transfere metade dos
    z.key[j] <- y.key[j + t]         // elementos para o novo nó
if not y.leaf then
    for j <- 1 to t do               // e metade dos filhos
         z.c[j] <- y.c[j + t]
y.n <- t - 1
for j <- x.n + 1 downto i + 1 do     // abre espaço em x para o
    x.c[j + 1] <- x.c[j]             // novo filho
x.c[i + 1] <- z
for j <- x.n downto i do             // abre espaço para o
    x.key[j + 1] <- x.key[j]         // elemento a promover
x.key[i] <- y.key[t]
x.n <- x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

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

```
i <- x.n
if x.leaf then  // se está numa folha, insere o elemento
    while i >= 1 and k < x.key[i] do
        x.key[i + 1] <- x.key[i]
        i <- i - 1
    x.key[i + 1] <- k
    x.n <- x.n + 1
    DISK-WRITE(x)
else             // senão, desce para o filho apropriado
    while i >= 1 and k < x.key[i] do
        i <- i - 1
    i <- i + 1
    DISK-READ(x.c[i])
    if x.c[i].n = 2t - 1 then    // o filho está cheio?
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key[i] then
            i <- i + 1
    B-TREE-INSERT-NONFULL(x.c[i], k)
```

B-Trees — Remoção do elemento com chave k (1)

Remoção do elemento efectuada numa única passagem pela árvore

Se o nó corrente contém o elemento na posição i ...

1. ... e é uma folha
   Remove o elemento
2. ... e é um nó interno
   a. se o filho i tem mais do que $t - 1$ elementos
      - substitui o elemento a remover pelo seu predecessor, que é removido da subárvore com raiz c_i

   b. senão, se o filho $i + 1$ tem mais do que $t - 1$ elementos
      - substitui o elemento a remover pelo seu sucessor, que é removido da subárvore com raiz c_{i+1}

   c. senão
      - funde os filhos i e $i + 1$
      - continua a partir do novo filho i (onde agora está o elemento a remover)

B-Trees — Remoção do elemento com chave k (2)

Se o nó corrente não contém o elemento

3. Se o nó corrente não é folha, seja i o índice do filho que é raiz da subárvore onde o elemento poderá estar

   Se o filho i tem mais do que $t - 1$ elementos

   - continua a partir do filho i

   Se o filho i tem $t - 1$ elementos

   a. se algum dos irmãos esquerdo ou direito de i tem mais do que $t - 1$ elementos
      - transfere um elemento para o filho i, por empréstimo de um irmão nessas condições
      - continua a partir do filho i

   b. senão
      - funde o filho i com o irmão esquerdo ou direito
      - continua a partir do nó que resultou da fusão

Se, terminada a remoção, a raiz fica vazia e não é folha

- o seu (único) filho passa a ser a nova raiz e a altura diminui

B-TREE-DELETE(T, k)

```
r <- T.root
B-TREE-DELETE-SAFE(r, k)        // remove o elemento
if r.n = 0 and not r.leaf then  // raiz vazia com filho?
    r <- r.c[1]
    FREE-NODE(T.root)
    T.root <- r                 // o filho é a nova raiz
```

B-TREE-DELETE-SAFE(x, k)

```
i <- 1
while i <= x.n and k > x.key[i] do
    i <- i + 1
if i <= x.n and k = x.key[i] then
    if x.leaf then
        B-TREE-DELETE-FROM-LEAF(x, i)                   // Caso 1
    else
        B-TREE-DELETE-FROM-INTERNAL-NODE(x, i)          // Caso 2
else if not x.leaf then
    B-TREE-DELETE-FROM-SUBTREE(x, i)                    // Caso 3
```

B-TREE-DELETE-FROM-LEAF(x, i)

```
for j <- i to x.n - 1 do                                // Caso 1
    x.key[j] <- x.key[j + 1]                            // Caso 1
x.n <- x.n - 1                                          // Caso 1
DISK-WRITE(x)                                           // Caso 1
```

B-TREE-DELETE-FROM-INTERNAL-NODE(x, i)

```
y <- x.c[i]
DISK-READ(y)
if y.n > t - 1 then                              // Caso 2a
    x.key[i] <- B-TREE-DELETE-MAX(y)             // Caso 2a
    DISK-WRITE(x)                                // Caso 2a
else
    z <- x.c[i + 1]
    DISK-READ(z)
    if z.n > t - 1 then                          // Caso 2b
        x.key[i] <- B-TREE-DELETE-MIN(z)         // Caso 2b
        DISK-WRITE(x)                            // Caso 2b
    else
        B-TREE-MERGE-CHILDREN(x, i)              // Caso 2c
        B-TREE-DELETE-SAFE(x.c[i], k)            // Caso 2c
```

B-TREE-DELETE-FROM-SUBTREE(x, i)

```
y <- x.c[i]
DISK-READ(y)
if y.n = t - 1 then
    borrowed <- FALSE
    if i > 1 then
        z <- x.c[i - 1]
        DISK-READ(z)
        if z.n > t - 1 then                            // Caso 3a
            B-TREE-BORROW-FROM-LEFT-SIBLING(x, i)      // Caso 3a
            borrowed <- TRUE                           // Caso 3a
        else
            m <- i - 1
    if not borrowed and i <= x.n then
        z <- x.c[i + 1]
        DISK-READ(z)
        if z.n > t - 1 then                            // Caso 3a
            B-TREE-BORROW-FROM-RIGHT-SIBLING(x, i)     // Caso 3a
            borrowed <- TRUE                           // Caso 3a
        else
            m <- i
    if not borrowed then                               // Caso 3b
        B-TREE-MERGE-CHILDREN(x, m)                    // Caso 3b
        y <- x.c[m]                                    // Caso 3b
B-TREE-DELETE-SAFE(y, k)
```

B-TREE-MERGE-CHILDREN(x, i)

```
y <- x.c[i]                      // fusão do filho i
z <- x.c[i + 1]                  // com o i+1
y.key[t] <- x.key[i]
for j <- 1 to t - 1 do           // muda conteúdo de
    y.key[t + j] <- z.key[j]     // c[i+1] para c[i]
if not y.leaf then
    for j <- 1 to t do           // incluindo filhos
        y.c[t + j] <- z.c[j]
y.n <- 2t - 1                    // c[i] fica cheio
for j <- i + 1 to x.n do
  x.key[j - 1] <- x.key[j]
for j <- i + 2 to x.n + 1 do
  x.c[j - 1] <- x.c[j]
x.n <- x.n - 1
FREE-NODE(z)                  // apaga c[i+1] antigo
DISK-WRITE(y)
DISK-WRITE(x)
```

(NOTA: Os nós `x`, `x.c[i]` e `x.c[i+1]` já foram lidos para memória)

B-TREE-BORROW-FROM-LEFT-SIBLING(x, i)

```
y <- x.c[i]                          // irmão esquerdo
z <- x.c[i - 1]                      // do nó i é o i-1
for j <- t - 1 downto 1 do           // abre espaço para
    y.key[j + 1] <- y.key[j]         // a nova 1ª chave
y.key[1] <- x.key[i - 1]
x.key[i - 1] <- z.key[z.n]
if not y.leaf then
    for j <- t downto 1 do           // abre espaço para
        y.c[j + 1] <- y.c[j]         // o novo 1º filho
    y.c[1] <- z.c[z.n + 1]
y.n <- t
z.n <- z.n - 1
DISK-WRITE(z)
DISK-WRITE(y)
DISK-WRITE(x)
```

(NOTA: Os nós `x`, `x.c[i - 1]` e `x.c[i]` já foram lidos para memória)

B-TREE-BORROW-FROM-RIGHT-SIBLING(x, i)

Exercício

B-TREE-DELETE-MAX(x)

Exercício

(o nó x tem mais do que $t - 1$ elementos;
a função devolve o elemento removido)

B-TREE-DELETE-MIN(x)

Exercício

(o nó x tem mais do que $t - 1$ elementos;
a função devolve o elemento removido)

Árvore com grau de ramificação mínimo t e com n elementos

Altura $\quad h = O(\log_t n)$

Complexidade temporal das operações

pesquisa, inserção, remoção

$$O(th) = O(t \log_t n)$$

Número de nós acedidos (nas operações acima)

$$O(h) = O(\log_t n)$$

Tries

A estrutura de dados *trie*

Uma *trie* é uma árvore cujos nós têm filhos que correspondem a símbolos do alfabeto das chaves

Uma chave está contida numa *trie* se o percurso que ela induz na *trie*, a partir da sua raiz, termina num nó que marca o fim de uma chave

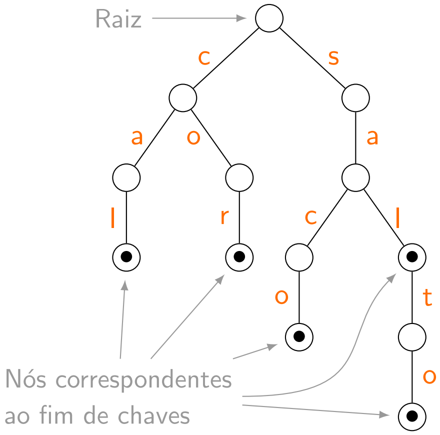
As *tries* apresentam algumas características que as distinguem de outras estruturas de dados

1. A complexidade das operações não depende do número de elementos que ela contém
2. As chaves não têm de estar explicitamente contidas na *trie*
3. As operações não se baseiam em comparações entre chaves

Uma *trie*

Exemplo

Representação de uma *trie* com as chaves (palavras) cal, cor, saco, sal e salto



Tries

d – dimensão do alfabeto (nº de símbolos distintos)

Chaves

k[1..m] – chave

$|k| = m$

Conteúdo dos nós (implementação com vector de filhos)

c[1..d] – filhos
p – pai (opcional)

word – TRUE sse a chave que termina no nó está na *trie*
ou
element – elemento associado à chave que termina(ria) no nó

TRIE-SEARCH(T, k)

```
x <- T.root
i <- 1
while x != NIL and i <= |k| do
    x <- x.c[k[i]]
    i <- i + 1
return x != NIL and x.word
```

Argumentos

T – *trie*

k – chave (palavra)

TRIE-SEARCH(T, k) — Complexidade

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 return x != NIL and x.word
```

Análise da complexidade para o pior caso

- Linhas 1, 2, 4, 5 e 6, e testes da linha 3: custo constante

$$O(1) + O(1) + (m+1)O(1) + m\,O(1) + m\,O(1) + O(1) \;=\; 4\,O(1) + 3m\,O(1) \;=\; 3\,O(m) \;=\; O(m)$$

TRIE-INSERT(T, k)

```
if T.root = NIL then
    T.root <- ALLOCATE-NODE()
    T.root.p <- NIL
x <- T.root
i <- 1
while i <= |k| and x.c[k[i]] != NIL do
    x <- x.c[k[i]]
    i <- i + 1
TRIE-INSERT-REMAINING(x, k, i)
```

ALLOCATE-NODE() devolve um novo nó da *trie* com

c[1..d] = NIL
p = NIL
word = FALSE

TRIE-INSERT-REMAINING(x, k, i)

```
y <- x
for j <- i to |k| do
    y.c[k[j]] <- ALLOCATE-NODE()
    y.c[k[j]].p <- y
    y <- y.c[k[j]]
y.word <- TRUE
```

Função que acrescenta, a partir do nó x, os nós necessários para incluir na *trie* o sufixo da chave k que ainda não está na *trie* e que começa no i-ésimo símbolo da chave

Se $i > |k|$, só afecta a marca de fim de palavra no nó x

TRIE-DELETE(T, k) (1)

```
x <- T.root
i <- 1
while x != NIL and i <= |k| do
    x <- x.c[k[i]]
    i <- i + 1
if x != NIL and x.word then
    x.word <- FALSE          // k deixa de estar na trie
...
```

Falta remover os nós da *trie* que deixam de ter um papel útil, por não corresponderem ao fim de uma palavra nem terem filhos

TRIE-DELETE(T, k) (2)

```
...
if x != NIL and x.word then
    x.word <- FALSE         // k deixa de estar na trie
    repeat
        i <- i - 1
        childless <- TRUE              // x tem filhos?
        j <- 1
        while j <= d and childless do
            if x.c[j] != NIL then
                childless <- FALSE
            j <- j + 1
        if childless then      // se não tem, é apagado
            y <- x.p
            if y = NIL then
                T.root <- NIL     // a trie ficou vazia
            else
                y.c[k[i]] <- NIL
            FREE-NODE(x)
          x <- y
    until x = NIL or not childless or x.word
```

Complexidade temporal das operações sobre uma *trie*

Implementação com vector de filhos — Resumo

Pesquisa da palavra k

$$O(m)$$

Inserção da palavra k

$$O(m)$$

Remoção da palavra k

$$O(m\,d)$$

Complexidade espacial

$$O(n\,w\,d)$$

Onde

$m = |\mathsf{k}|$

d é o número de símbolos do alfabeto

n é o número de palavras na *trie*

w é comprimento médio das palavras na *trie*

Programação dinâmica

Programação dinâmica

Método usado na construção de soluções iterativas para problemas cuja solução recursiva tem uma complexidade elevada (exponencial, em geral)

Aplica-se, normalmente, a problemas de optimização

- Um problema de optimização é um problema em que se procura minimizar ou maximizar algum valor associado às suas soluções

Corte de varas

Uma empresa compra varas de aço, corta-as e vende-as aos pedaços

O preço de venda de cada pedaço depende do seu comprimento

Problema

Como cortar uma vara de comprimento n de forma a maximizar o valor de venda?

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	7	11	11	17	20	20	24	27

Corte de varas

Caracterização de uma solução óptima (1)

Soluções possíveis, para uma vara de comprimento 10

- Um corte de comprimento 1, mais as soluções para uma vara de comprimento 9
- Um corte de comprimento 2, mais as soluções para uma vara de comprimento 8
- Um corte de comprimento 3, mais as soluções para uma vara de comprimento 7

  ...

- Um corte de comprimento 9, mais as soluções para uma vara de comprimento 1
- Um corte de comprimento 10, mais as soluções para uma uma vara de comprimento 0

Qual a melhor?

Corte de varas

Caracterização de uma solução óptima (2)

Sejam os tamanhos dos cortes possíveis

$$1, 2, \ldots, n$$

com preços

$$p_1, p_2, \ldots, p_n$$

O maior valor de venda de uma vara de comprimento n é o máximo que se obtém

- fazendo um corte inicial de comprimento $1 \leq i \leq n$, de valor p_i, somado com
- o maior valor de venda de uma vara de comprimento $n - i$

Corte de varas

Função recursiva

Corte de uma vara de comprimento n

Tamanho dos cortes: $i = 1, \ldots, n$
Preços: $P = (p_1\ p_2\ \ldots\ p_n)$

$r_P[0..n]$: função t.q. $r_P[l]$ é o maior preço que se pode obter para uma vara de comprimento l, dados os preços P

$$r_P[l] = \begin{cases} 0 & \text{se} \quad l = 0 \\ \max\limits_{1 \leq i \leq l} \{p_i + r_P[l-i]\} & \text{se} \quad l > 0 \end{cases}$$

Preço máximo (chamada inicial da função): $r_P[n]$

Corte de varas

Implementação recursiva

CUT-ROD(p, l)

```
if l = 0 then
  return 0
q <- −∞
for i <- 1 to l do
  q <- max(q, p[i] + CUT-ROD(p, l - i))
return q
```

Argumentos

- p Preços das varas de comprimentos $\{1, 2, \ldots, n\}$
- l Comprimento da vara a cortar

Chamada inicial da função: CUT-ROD(p, n)

Corte de varas

Alguns números

Número de cortes possíveis

$$2^{n-1}$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 3+1$

$1+1+2 \quad 1+2+1 \quad 2+1+1 \quad 1+1+1+1$

Número de cortes distintos possíveis

$$O\left(\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}\right)$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 1+1+2 \quad 1+1+1+1$

Corte de varas

Implementação recursiva com *memoização*

MEMOIZED-CUT-ROD(p, n)

```
let r[0..n] be a new array
for l <- 0 to n do
     r[l] <- −∞
return MEMOIZED-CUT-ROD-2(p, n, r)
```

MEMOIZED-CUT-ROD-2(p, l, r)

```
if r[l] = −∞ then
  if l = 0 then
    q <- 0
  else
    q <- −∞
    for i <- 1 to l do
      q <- max(q, p[i] + MEMOIZED-CUT-ROD-2(p, l - i, r))
  r[l] <- q
return r[l]
```

NB: isto não é programação dinâmica

Corte de varas

Cálculo iterativo de $r[n]$ (1)

p_i	1	5	7	11	11	17	20	20	24	27

Preenchimento do vector r

	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	7	11	12	17	20	22	25	28

1. Caso base: $r[0] \leftarrow 0$
2. $r[1] \leftarrow \max\{\, p_1 + r[0] \,\} = \max\{\, 1 + 0 \,\}$
3. $r[2] \leftarrow \max\{\, p_1 + r[1], p_2 + r[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$
4. $r[3] \leftarrow \max\{\, p_1 + r[2], p_2 + r[1], p_3 + r[0] \,\} =$ $= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$

   ...

11. $r[10] \leftarrow \max\{\, p_1 + r[9], p_2 + r[8], \ldots, p_4 + r[6], \ldots, p_{10} + r[0] \,\}$

Corte de varas

Cálculo iterativo de $r[n]$ (2)

BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] be a new array
r[0] <- 0
for l <- 1 to n do
  q <- −∞
  for i <- 1 to l do
    q <- max(q, p[i] + r[l - i])
  r[l] <- q
return r[n]
```

Corte de varas
Complexidade

Complexidade de BOTTOM-UP-CUT-ROD($p_1\ p_2\ \ldots\ p_n$)

Ciclo 3–7 é executado n vezes

Ciclo 5–6 é executado l vezes, $l = 1, \ldots, n$

$$1 + 2 + \ldots + n = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} = \Theta(n^2)$$

Todas as operações têm custo constante

Complexidade temporal $\Theta(n^2)$

Complexidade espacial $\Theta(n)$

Corte de varas

Construção da solução (1)

O valor máximo por que é possível vender uma vara é calculado pela função BOTTOM-UP-CUT-ROD

Quais os cortes a fazer para obter esse valor?

Para o preenchimento da posição `l` do vector `r[]`, é escolhido o valor máximo de `p[i] + r[l - i]`

O facto de o valor máximo incluir a parcela `p[i]` significa a inclusão de um pedaço de vara de comprimento `i`

Logo, o valor máximo por que é possível vender uma vara de comprimento `l` (vector `s[]`) será obtido

- com um pedaço de comprimento `i` e
- o valor máximo por que é possível vender uma vara de comprimento `l - i`

Corte de varas

Construção da solução (2)

p_i		1	5	7	11	11	17	20	20	24	27
	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	7	11	12	17	20	22	25	28
$s[i]$		1	2	3	4	1	6	7	2	2	4

1. Caso base: $r[0] \leftarrow 0$
2. $r[1] \leftarrow \max\{\, p_1 + r[0] \,\} = \max\{\, 1 + 0 \,\}$, $s[1] \leftarrow 1$
3. $r[2] \leftarrow \max\{\, p_1 + r[1], p_2 + r[0] \,\} = \max\{\, 1+1, 5+0 \,\}$, $s[2] \leftarrow 2$
4. $r[3] \leftarrow \max\{\, p_1 + r[2], p_2 + r[1], p_3 + r[0] \,\} =$
   $= \max\{\, 1+5, 5+1, 7+0 \,\}$, $s[3] \leftarrow 3$

   ...

11. $r[10] \leftarrow \max\{\, p_1 + r[9], p_2 + r[8], \ldots, p_4 + r[6], \ldots, p_{10} + r[0] \,\}$,
    $s[10] \leftarrow 4$

Corte de varas

Construção da solução (3)

$s[1..n]$: $s[l]$ é o primeiro corte a fazer numa vara de comprimento l

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[1..n] be new arrays
r[0] <- 0
for l <- 1 to n do
  q <- −∞
  for i <- 1 to l do
    if q < p[i] + r[l - i] then
      q <- p[i] + r[l - i]
      s[l] <- i          // corte feito na posição i
  r[l] <- q
return r and s
```

Corte de varas

Resolução completa

PRINT-CUT-ROD-SOLUTION(p, n)

```
(r, s) <- EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
print "The best price is ", r[n]
while n > 0 do
  print s[n]
  n <- n - s[n]
```

Programação dinâmica

Condições de aplicabilidade

A programação dinâmica aplica-se a problemas que apresentam as características seguintes:

Subestrutura óptima (*Optimal substructure*)

- Um problema tem subestrutura óptima se uma sua solução óptima é construída com recurso a soluções óptimas de subproblemas

Subproblemas repetidos (*Overlapping subproblems*)

- Existem subproblemas repetidos quando os subproblemas de um problema têm subproblemas em comum

Programação dinâmica

Aplicação

1. Caracterização de uma solução óptima
2. Formulação recursiva do cálculo do valor de uma solução óptima
3. Cálculo iterativo do valor de uma solução óptima, por tabelamento
4. Construção de uma solução óptima