15.3-6 Imagine que você queira fazer o câmbio de uma moeda por outra e percebe que, em vez de trocar diretamente uma moeda por outra, seria melhor efetuar uma série de trocas intermediárias por outras moedas, por fim tendo em mãos a moeda que queria. Suponha que você possa trocar n moedas diferentes, numeradas de 1,2,...,n, que começará com a moeda 1 e quer terminar com a moeda n. Você tem, para cada par de moedas i e j, uma taxa de câmbio r_{ij}, o que significa que, se você começar com d unidades da moeda i, poderá trocá-las por dr_{ij} unidades da moeda j. Uma sequência de trocas pode acarretar uma comissão, que depende do número de trocas que você faz. Seja c_k a comissão cobrada quando você faz k trocas. Mostre que, se $c_k = 0$ para todo $k = 1,2,...,n$, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n exibe subestrutura ótima. Então, mostre que, se as comissões c_k são valores arbitrários, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n não exibe necessariamente subestrutura ótima.

15.4 Subsequência comum mais longa

Em aplicações biológicas, muitas vezes, é preciso comparar o DNA de dois (ou mais) organismos diferentes. Um filamento de DNA consiste em uma cadeia de moléculas denominadas ***bases***, na qual as bases possíveis são adenina, guanina, citosina e timina. Representando cada uma dessas bases por sua letra inicial, podemos expressar um filamento de DNA como uma cadeia no conjunto finito {A,C,G,T}. (O Apêndice C dá a definição de uma cadeia.) Por exemplo, o DNA de um organismo pode ser S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, e o DNA de outro organismo pode ser S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Uma razão para a comparação de dois filamentos de DNA é determinar o grau de "semelhança" entre eles, que serve como alguma medida da magnitude da relação entre os dois organismos. Podemos definir (e definimos) a semelhança de muitas maneiras diferentes. Por exemplo, podemos dizer que dois filamentos de DNA são semelhantes se um deles for uma subcadeia do outro. (O Capítulo 32 explora algoritmos para resolver esse problema.) Em nosso exemplo, nem S_1 nem S_2 é uma subcadeia do outro. Alternativamente, poderíamos dizer que dois filamentos são semelhantes se o número de mudanças necessárias para transformar um no outro for pequeno. (O Problema 15-3 explora essa noção.) Ainda uma outra maneira de medir a semelhança entre filamentos S_1 e S_2 é encontrar um terceiro filamento S_3 no qual as bases em S_3 aparecem em cada um dos filamentos S_1 e S_2; essas bases devem aparecer na mesma ordem, mas não precisam ser necessariamente consecutivas. Quanto mais longo o filamento S_3 que pudermos encontrar, maior será a semelhança entre S_1 e S_2 . Em nosso exemplo, o filamento S_3 mais longo é GTCGTCGGAAGCCGGCCGAA.

Formalizamos essa última noção de semelhança como o problema da subsequência comum mais longa. Uma subsequência de uma determinada sequência é apenas a sequência dada na qual foram omitidos zero ou mais elementos. Em termos formais, dada uma sequência $X = \langle x_1, x_2, ..., x_m \rangle$, uma outra sequência $Z = \langle z_1, z_2, ..., z_k \rangle$ é uma ***subsequência*** de X se existir uma sequência estritamente crescente $\langle i_1, i_2, ..., i_k \rangle$ de índices de X tais que, para todo $j = 1, 2, ..., k$, temos $x_i = z_j$. Por exemplo, $Z = \langle B, C, D, B \rangle$ é uma subsequência de $X = \langle A, B, C, B, D, A, B \rangle$ com sequência de índices correspondente $\langle 2, 3, 5, 7 \rangle$.

Dadas duas sequências X e Y, dizemos que uma sequência Z é uma ***subsequência comum*** de X e Y se Z é uma subsequência de X e Y. Por exemplo, se $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$, a sequência $\langle B, C, A \rangle$ é uma subsequência comum das sequências X e Y. Porém, a sequência $\langle B, C, A \rangle$ não é uma subsequência comum *mais longa* (LCS — longest common subsequence) de X e Y, já que tem comprimento 3, e a sequência $\langle B, C, B, A \rangle$, que também é comum a X e Y, tem comprimento 4. A sequência $\langle B, C, B, A \rangle$ é uma LCS de X e Y, assim como a sequência $\langle B, D, A, B \rangle$, visto que não existe nenhuma subsequência comum de comprimento 5 ou maior.

No ***problema da subsequência comum mais longa***, temos duas sequências $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$, e desejamos encontrar uma subsequência comum de comprimento máximo de X e Y. Esta seção mostra como resolver o problema da LCS eficientemente, usando programação dinâmica.

Etapa 1: Caracterização de uma subsequência comum mais longa

Uma abordagem de força bruta para resolver o problema da LCS seria enumerar todas as subsequências de X e conferir cada subsequência para ver se ela também é uma subsequência de Y, sem perder de vista a subsequência mais longa encontrada. Cada subsequência de X corresponde a um subconjunto dos índices $\{1, 2, ..., m\}$ de X. Como X tem 2^m subsequências, essa abordagem requer tempo exponencial, o que a torna impraticável para sequências longas.

Porém, o problema da LCS tem uma propriedade de subestrutura ótima, como mostra o teorema a seguir. Como veremos, as classes naturais de subproblemas correspondem a pares de "prefixos" das duas sequências de entrada. Mais precisamente, dada uma sequência $X = \langle x_1, x_2, .., x_m \rangle$, definimos o i-ésimo ***prefixo*** de X, para $i = 0, 1, ..., m$, como $X_i = \langle x_1, x_2, ..., x_i \rangle$. Por exemplo, se $X = \langle A, B, C, B, D, A, B \rangle$, então $X_4 = \langle A, B, C, B \rangle$ e X_0 é a sequência vazia.

Teorema 15.1 (Subestrutura ótima de uma LCS)

Sejam $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ as sequências, e seja $Z = \langle z_1, z_2, ..., z_k \rangle$ qualquer LCS de X e Y.

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma LCS de X e Y_{n-1}.

Prova (1) Se $z_k \neq x_m$, então podemos anexar $x_m = y_n$ a Z para obter uma subsequência comum de X e Y de comprimento $k + 1$, contradizendo a suposição de que Z é uma subsequência comum *mais longa* de X e Y. Assim, devemos ter $z_k = x_m = y_n$. Agora, o prefixo Z_{k-1} é uma subsequência comum de comprimento $(k - 1)$ de X_{m-1} e Y_{n-1}. Desejamos mostrar que ela é uma LCS. Suponha, por contradição, que exista uma subsequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que $k - 1$. Então, anexar $x_m = y_n$ a W produz uma subsequência comum de X e Y cujo comprimento é maior que k, o que é uma contradição.

(2) Se $z_k \neq x_m$, então Z é uma subsequência comum de X_{m-1} e Y. Se existisse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W seria também uma subsequência comum de X_m e Y, contradizendo a suposição de que Z é uma LCS de X e Y.

(3) A prova é simétrica a (2).

O modo como o Teorema 15.1 caracteriza subsequências comuns mais longas nos diz que uma LCS de duas sequências contém uma LCS de prefixos das duas sequências. Assim, o problema de LCS tem uma propriedade de subestrutura ótima. Uma solução recursiva também tem a propriedade de subproblemas sobrepostos, como veremos em breve.

Etapa 2: Uma solução recursiva

O Teorema 15.1 subentende que devemos examinar um ou dois subproblemas quando queremos encontrar uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Se $x_m = y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}. Anexar $x_m = y_n$ a essa LCS produz uma LCS de X e Y. Se $x_m \neq y_n$, então devemos resolver dois subproblemas: encontrar uma LCS de X_{m-1} e Y e encontrar uma LCS de X e Y_{n-1}. A mais longa de qualquer dessas duas LCS é uma LCS de X e Y. Como esses casos esgotam todas as possibilidades, sabemos que uma das soluções ótimas de subproblemas certamente aparecerá dentro de uma LCS de X e Y.

É fácil ver a propriedade de subproblemas sobrepostos no problema da LCS. Para encontrar uma LCS de X e Y, pode ser necessário encontrar as LCS de X e Y_{n-1} e de X_{m-1} e Y. Porém, cada um desses subproblemas tem o subsubproblema de encontrar uma LCS de X_{m-1} e Y_{n-1}. Muitos outros subproblemas compartilham subsubproblemas.

Como ocorreu no problema de multiplicação de cadeias de matrizes, nossa solução recursiva para o problema da LCS envolve estabelecer uma recorrência para o valor de uma solução ótima. Vamos definir $c[i, j]$ como o

comprimento de uma LCS das sequências X_i e Y_j. Se $i = 0$ ou $j = 0$, uma das sequências tem comprimento 0 e, portanto, a LCS tem comprimento 0. A subestrutura ótima do problema da LCS dá a fórmula recursiva

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1,j-1]+1 & \text{se } i,j > 0 \text{ e } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j. \end{cases} \tag{15.9}$$

Observe que, nessa formulação recursiva, uma condição no problema restringe os subproblemas que podemos considerar. Quando $x_i = y_j$, podemos e devemos considerar o subproblema de encontrar a LCS de X_{i-1} e Y_{j-1}. Caso contrário, consideramos os dois subproblemas de encontrar uma LCS de X_i e Y_{j-1} e de X_{i-1} e Y_j. Nos algoritmos de programação dinâmica que já examinamos — para corte de hastes e para multiplicação de cadeias de matrizes —, não descartamos nenhum subproblema por causa de condições no problema. O algoritmo para encontrar uma LCS não é o único algoritmo de programação dinâmica que descarta subproblemas com base em condições no problema. Por exemplo, o problema da distância de edição (ver o Problema 15-3) tem essa característica.

Etapa 3: Cálculo do comprimento de uma LCS

Tendo como base a equação (15.9), seria fácil escrever um algoritmo recursivo de tempo exponencial para calcular o comprimento de uma LCS de duas sequências. Contudo, visto que o problema da LCS tem somente $Q(mn)$ subproblemas distintos, podemos usar programação dinâmica para calcular as soluções de baixo para cima.

O procedimento LCS-LENGTH toma duas sequências $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ como entradas. Armazena os valores $c[i, j]$ em uma tabela $c[0 .. m, 0 .. n]$ e calcula as entradas em ***ordem orientada por linha*** (isto é, preenche a primeira linha de c da esquerda para a direita, depois a segunda linha, e assim por diante). O procedimento também mantém a tabela $b[1 .. m, 1 .. n]$ para ajudar a construir uma solução ótima. Intuitivamente, $b[i, j]$ aponta para a entrada da tabela correspondente à solução ótima de subproblema escolhida ao se calcular $c[i, j]$. O procedimento retorna as tabelas b e c; $c[m, n]$ contém o comprimento de uma LCS de X e Y.

```
LCS-LENGTH (X, Y)
1   m = X.comprimento
2   n = Y.comprimento
3   sejam b[1..m, 1..n] e c[0..m, 0..n] tabelas novas
4   for i = 1 to m
5       c[i,0] = 0
6   for j = 0 to n
7       c[0, j] = 0
8   for i = 1 to m
9       for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j- 1]
17              b[i, j] = "←"
18  return c, b
```

A Figura 15.8 mostra as tabelas produzidas por LCS-LENGTH nas sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O tempo de execução do procedimento é Q(mn), já que cada entrada de tabela demora o tempo Q(1) para ser calculada.

Etapa 4: Construção de uma LCS

A tabela b retornada por LCS-LENGTH nos habilita a construir rapidamente uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Simplesmente começamos em $b[m, n]$ e percorremos a tabela seguindo as setas. Sempre que encontramos uma " " na entrada $b[i, j]$, ela implica que $x_i = y_j$ é um elemento da LCS que LCS-LENGTH encontrou. Com esse método, encontramos os elementos da LCS em ordem inversa. O procedimento recursivo a seguir imprime uma LCS de X e Y na ordem direta adequada. A invocação inicial é PRINT-LCS(b, X, *X.comprimento*, *Y.comprimento*).

```
PRINT-LCS(b, X, i, j)
1   if i == 0 ou j == 0
2      return
3   if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6   elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8   else PRINT-LCS(b, X, i, j − 1)
```

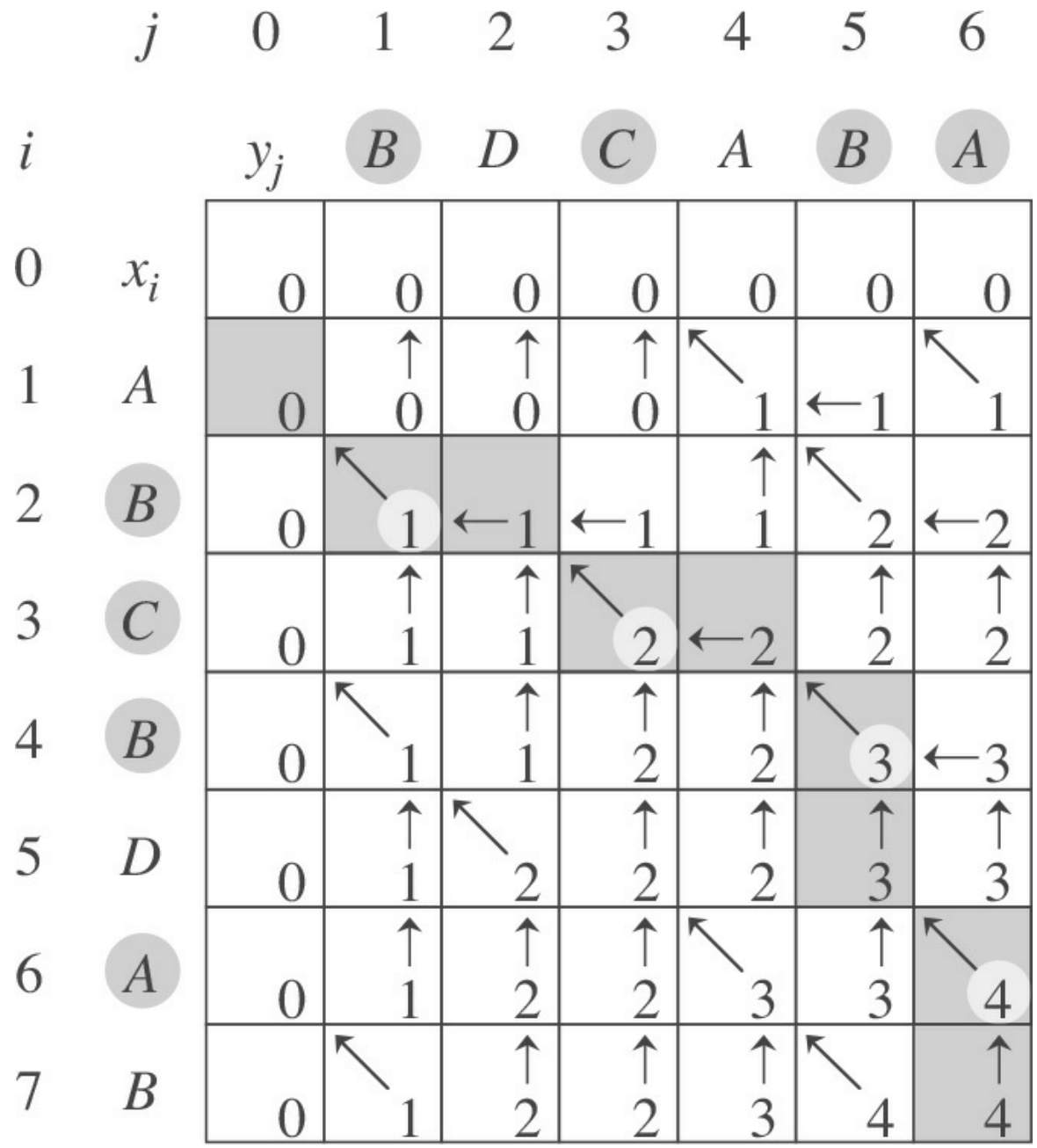

Figura 15.8 As tabelas c e b calculadas por LCS-LENGTH para as sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O quadrado na linha i e coluna j contém o valor de $c[i, j]$ e a seta adequada para o valor de $b[i, j]$. A entrada 4 em $c[7, 6]$ — o canto inferior direito da tabela — é o comprimento de uma LCS $\langle B, C, B, A \rangle$ de X e Y. Para $i, j > 0$, a entrada $c[i, j]$ depende apenas de $x_i = y_j$ e dos valores nas entradas $c[i - 1, j]$, $c[i, j - 1]$ e $c[i - 1, j - 1]$, que são calculados antes de $c[i, j]$. Para reconstruir os elementos de uma LCS, siga as setas $b[i, j]$ desde o canto inferior direito; a sequência está sombreada. Cada "\" na sequência sombreada corresponde a uma entrada (destacada) para a qual $x_i = y_j$ é membro de uma LCS.

Para a tabela b na Figura 15.8, esse procedimento imprime $BCBA$. O procedimento demora o tempo $O(m + n)$, já que decrementa no mínimo um de i e j em cada fase da recursão.

Melhorando o código

Depois de ter desenvolvido um algoritmo, você, muitas vezes, constatará que é possível melhorar o tempo ou o espaço que ele utiliza. Algumas mudanças podem simplificar o código e melhorar fatores constantes, porém, quanto ao

mais, não produzem nenhuma melhora assintótica no desempenho. Outras podem resultar em economias assintóticas significativas de tempo e de espaço.

Por exemplo, no algoritmo LCS podemos eliminar totalmente a tabela b. Cada entrada $c[i, j]$ depende apenas de três outras entradas na tabela $c[i - 1, j - 1]$, $c[i - 1, j]$ e $c[i, j - 1]$. Dado o valor de $c[i, j]$, podemos determinar em tempo $O(1)$ de qual desses três valores foi usado para calcular $c[i, j]$, sem inspecionar a tabela b. Assim, podemos reconstruir uma LCS em tempo $O(m + n)$ usando um procedimento semelhante a PRINT-LCS. (O Exercício 15.4-2 pede que você dê o pseudocódigo.) Embora economizemos espaço $Q(mn)$ por esse método, o requisito de espaço auxiliar para calcular uma LCS não diminui assintoticamente, já que, de qualquer modo, precisamos do espaço $Q(mn)$ para a tabela c.

Entretanto, podemos reduzir os requisitos de espaço assintótico para LCS-LENGTH, já que esse procedimento só precisa de duas linhas da tabela c por vez: a linha que está sendo calculada e a linha anterior. (De fato, como o Exercício 15.4-4 pede que você mostre, podemos usar apenas um pouquinho mais que o espaço para uma linha de c para calcular o comprimento de uma LCS.) Esse aperfeiçoamento funciona se necessitamos apenas do comprimento de uma LCS; se >precisarmos reconstruir os elementos de uma LCS, a tabela menor não guardará informações suficientes para reconstituir nossas etapas no tempo $O(m + n)$.

Exercícios

15.4-1 Determine uma LCS de $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ e $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$.

15.4-2 Dê o pseudocódigo para reconstruir uma LCS partindo da tabela c concluída e das sequências originais $X = \langle x_1, x_2, ..., x_m\rangle$ e $Y = \langle y_1, y_2, ..., y_n\rangle$ em tempo $O(m + n)$, sem usar a tabela b.

15.4-3 Dê uma versão memoizada de LCS-LENGTH que seja executada no tempo $O(mn)$.

15.4-4 Mostre como calcular o comprimento de uma LCS usando apenas $2 \cdot \min(m, n)$ entradas na tabela c mais o espaço adicional $O(1)$. Em seguida, mostre como fazer a mesma coisa usando $\min(m, n)$ entradas mais o espaço adicional $O(1)$.

15.4-5 Dê um algoritmo de tempo $O(n_2)$ para encontrar a subsequência monotonicamente crescente mais longa de uma sequência de n números.

15.4-6 ★ Dê um algoritmo de tempo $O(n \lg n)$ para encontrar a subsequência mais longa monotonicamente crescente de uma sequência de n números. (*Sugestão*: Observe que o último elemento de uma subsequência candidata de comprimento i é no mínimo tão grande quanto o último elemento de uma subsequência candidata de comprimento $i - 1$. Mantenha as subsequências candidatas ligando-as por meio da sequência de entrada.)

15.5 ÁRVORES DE BUSCA BINÁRIA ÓTIMAS

Suponha que estejamos projetando um programa para traduzir texto do inglês para o francês. Para cada ocorrência de cada palavra inglesa no texto, precisamos procurar sua equivalente em francês. Um modo de executar essas operações de busca é construir uma árvore de busca binária com n palavras inglesas como chaves e suas equivalentes francesas como dados satélites. Como pesquisaremos a árvore para cada palavra individual no texto, queremos que o tempo total gasto na busca seja tão baixo quanto possível. Poderíamos assegurar um tempo de busca $O(\lg n)$ por ocorrência usando uma árvore vermelho-preto ou qualquer outra árvore de busca binária balanceada. Porém, as palavras aparecem com frequências diferentes, e uma palavra usada frequentemente como *the* pode aparecer longe da raiz, enquanto uma palavra raramente usada como *machicolation* apareceria perto da raiz. Tal organização reduziria a velocidade da tradução, já que o número de nós visitados durante a busca de uma chave em uma árvore de