2
Estruturas Hierárquicas

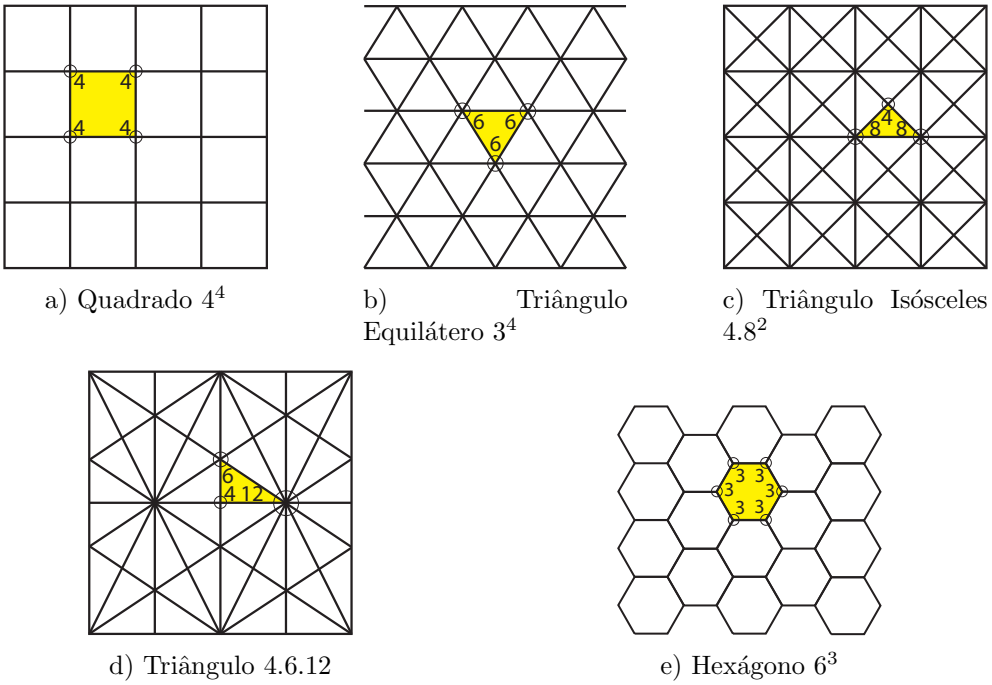
Para localizar um objeto de uma maneira eficiente, é preciso *decompor* o espaço em pequenas partes, utilizando uma de suas *representações*, *indexar* cada parte de uma única forma e *armazená-la* numa estrutura apropriada. Por exemplo, o sistema de endereços dos correios decompõe uma cidade em blocos, representados pelos bairros, indexados pelo CEP de forma única, sendo essas informações armazenadas no endereço de cada carta.

Este capítulo descreve algumas estruturas de decomposição, representação, indexação e de armazenamento espacial. Seu principal objetivo é dar uma visão geral simplificada das diferentes estruturas espaciais existentes. Para uma descrição mais detalhada, consultar os livros clássicos de Samet (17, 18) sobre estruturas de dados e suas aplicações.

2.1
Decomposição

Existem infinitas maneiras de decompor um espaço. Em particular, uma maneira simples de decompor o plano é escolher uma pequena peça, e a partir dela tentar preencher todo o plano, encaixando e colando infinitas peças que possuem a mesma forma que a peça original, sem sobrepor e sem deixar buracos. Esse tipo de decomposição é chamado de *ladrilhamento*, e a pequena peça inicialmente escolhida para decompor o plano é chamada de *ladrilho*. Como os ladrilhos mais comuns são os que possuem formas poligonais, considera-se sempre que um ladrilho é um polígono. O *grau* do vértice de um ladrilho é o número de arestas que se conectam a ele dentro do ladrilhamento. Os ladrilhos são descritos pelo uso de uma notação, baseada no grau de cada vértice, da seguinte forma: sejam $g_0, g_1, \ldots, g_n$ os n graus na ordem crescente dos n vértices de um ladrilho; então sua notação é definida como sendo $[g_0.g_1.\ \ldots\ .g_n]$. Se para algum $i \in \{0, \ldots, n\}$, tem-se $g_i = g_{i+1} = \cdots = g_{i+m}$; então sua notação também pode ser escrita como sendo $[g_0.g_1.\ \ldots\ .g_i^m.g_{i+m+1}.\ \ldots\ .g_n]$. A Figura 2.1, por exemplo, ilustra alguns ladrilhamentos, em particular o ladrilhamento da Figura 2.1(c), que tem a forma de um triângulo isósceles, onde o primeiro vértice tem grau 4, enquanto

os outros dois vértices têm grau oito cada um; então nesse caso sua notação é dada por $[4.8^2]$.



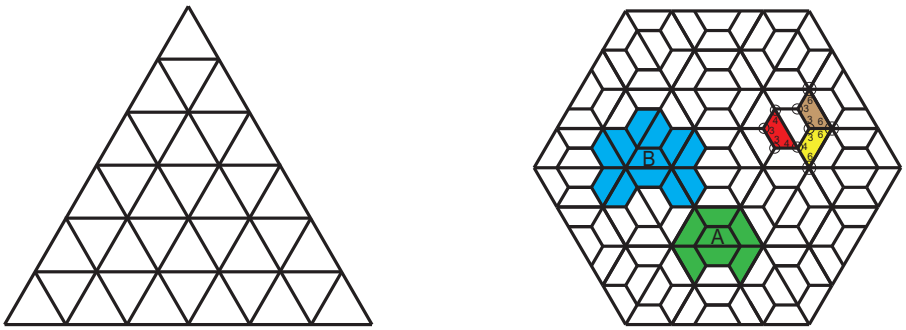
a) Quadrado 4^4 b) Triângulo Equilátero 3^4 c) Triângulo Isósceles 4.8^2

d) Triângulo 4.6.12 e) Hexágono 6^3

Figura 2.1: Exemplos de ladrilhamento do plano.

Um ladrilhamento em que o polígono formado da união de um ladrilho com os ladrilhos com os quais ele compartilha uma aresta ou um vértice, é o mesmo polígono para qualquer ladrilho. Nesse caso esse ladrilhamento é dito ser *isoedral*. Por exemplo, considere os dois ladrilhamentos da Figura 2.2. Um consiste de triângulos equiláteros (Figura 2.2(a)) e é descrito por $[6^3]$. O outro consiste de trapézios (Figura 2.2(b)) e é descrito por $[3^2.4^2]$, por $[3^2.6^2]$, ou ainda por $[3.4.6^2]$. É fácil verificar que os triângulos eqüiláteros são isoedrais, pois, para qualquer ladrilho escolhido, o polígono formado pela união desse ladrilho com os ladrilhos com os quais ele compartilha um vértice ou uma aresta, sempre é um hexágono. Isso já não acontece com os trapézios, como pode ser visto pelos ladrilhos A e B.

Um ladrilhamento é dito *regular* se o seu ladrilho é um polígono regular. O conjunto de ladrilhos formado por uma seqüência de ladrilhos de mesma forma, contidos inteiramente uns nos outros, é chamado de *ladrilho hierárquico*, cuja forma não precisa ser necessariamente igual a do ladrilho inicial. O *nível* de um ladrilho hierárquico é a quantidade de ladrilhos existente na seqüência. Um ladrilho hierárquico de nível $k > 0$ é dito ser *similar*, se ele possui a mesma forma de seu ladrilho hierárquico de nível 0. Para que um ladrilho hierárquico possua um nível infinito, é necessário que ele seja similar. Um *ladrilhamento hierárquico* é um ladrilhamento formado de ladrilhos hierárquicos. Um *ladrilha-*



a) Isoedral b) Não isoedral

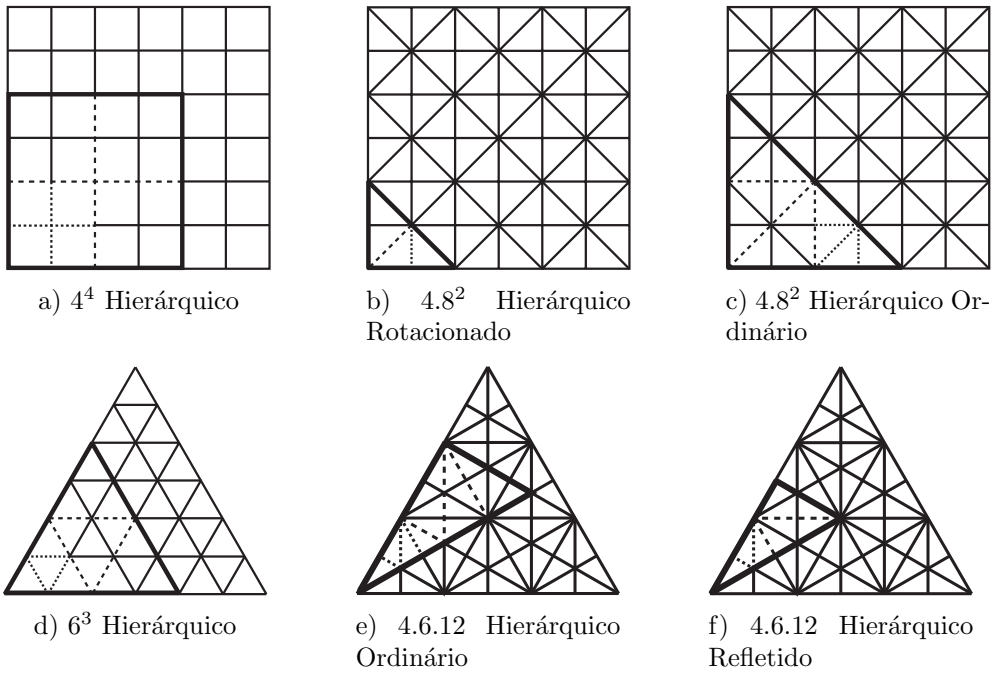
Figura 2.2: Ladrilhamento isoedral e não-isoedral.

mento hierárquico similar é um ladrilhamento hierárquico formado de ladrilhos hierárquicos similares.

O critério mais importante a se discutir é a distinção entre hierarquias *limitadas* e *ilimitadas* de ladrilhamento. Um ladrilhamento hierárquico é dito *ilimitado* se ele contém um ladrilho hierárquico de nível infinito. Um ladrilhamento hierárquico é *limitado* se ele não é similar. O ladrilhamento hexagonal $[3^6]$ da Figura 2.1(e) é limitado, pois todos os seus ladrilhos hierárquicos possuem nível 0. Bell *et al.* (3), afirmam que apenas quatro ladrilhamentos são ilimitados, os quais estão ilustrados na Figura 2.3. Deles $[4^4]$, consistindo do ladrilho quadrado, e o $[6^3]$, consistindo de ladrilho triângulo equilátero, são os bem conhecidos ladrilhamentos regulares (2). Seus ladrilhos hierárquicos são a Figura 2.3(a) e a Figura 2.3(d), respectivamente. Os ladrilhamentos $[4^4]$ e $[6^3]$ podem gerar um número infinito de ladrilhos hierárquicos diferentes, onde cada ladrilho hierárquico de nível n consiste de n^2 ladrilhos.

Os demais ladrilhamentos triangulares não-regulares e ilimitados $[4.8^2]$, correspondentes à Figura 2.1(c), e o [4.6.12], correspondentes à Figura 2.1(d), são junções dos centróides dos ladrilhos de $[4^4]$ e $[6^3]$, respectivamente, de ambos os seus vértices e pontos médios de suas arestas. O ladrilhamento $[4.8^2]$ possui um ladrilhamento ilimitado hierárquico ordinário, que corresponde à Figura 2.3(c), e um ladrilhamento ilimitado hierárquico rotacionado, que corresponde à Figura 2.3(b), exigindo uma rotação de 135° graus entre níveis. Já o ladrilhamento [4.6.12] tem um ladrilhamento ilimitado hierárquico ordinário, que corresponde à Figura 2.3(e), e um ladrilhamento ilimitado hierárquico refletido, que corresponde à Figura 2.3(f), que exige uma reflexão do ladrilho básico entre níveis.

No caso dos tipos de hierarquias $[4.8^2]$ e [4.6.12], o ladrilhamento não é similar sem uma rotação ou uma reflexão, quando a hierarquia não é ordinária. Isso pode ser visto observando-se o uso de pontos na Figura 2.3 para delimitar o ladrilho no seu ladrilho hierárquico. Do mesmo modo, linhas tracejadas são



a) 4^4 Hierárquico

b) 4.8^2 Hierárquico Rotacionado

c) 4.8^2 Hierárquico Ordinário

d) 6^3 Hierárquico

e) 4.6.12 Hierárquico Ordinário

f) 4.6.12 Hierárquico Refletido

Figura 2.3: Ladrilhamentos ilimitados, os pontos são para delimitar o ladrilho no seu ladrilho hierárquico.

usadas para delimitar os componentes dos ladrilhos de segundo nível. Para as hierarquias ordinárias de $[4.8^2]$ e de [4.6.12], cada ladrilho hierárquico de nível n consiste de n^2 ladrilhos. Para a hierarquia reflexiva de [4.6.12], cada ladrilho hierárquico de nível n consiste de $3n^2$ ladrilhos, enquanto para a hierarquia rotacionada de $[4.8^2]$, cada ladrilho hierárquico de nível n consiste de $2n^2$ ladrilhos.

Ladrilhos são ditos *adjacentes*, quando compartilham ao menos um vértice ou uma aresta. Dois ladrilhos são ditos *vizinhos* se eles são adjacentes. Um ladrilhamento é *uniformemente adjacente* se as distâncias entre o centróide de um ladrilho e os centróides de todos os seus vizinhos são iguais. O *número de adjacência* de um ladrilhamento é o número de distâncias diferentes de centróides internos entre qualquer ladrilho e seus vizinhos. No caso de $[4^4]$, existem apenas duas distâncias adjacentes; por sua vez $[6^3]$ tem três distâncias adjacentes. Um ladrilhamento é dito ter *orientação uniforme*, se os ladrilhos podem ser transformados uns nos outros por uma translação do plano, que não envolve rotação e nem reflexão. O ladrilhamento $[4^4]$ possui orientação uniforme; já o ladrilhamento $[6^3]$, não.

Para representar dados no plano Euclidiano, os ladrilhos ilimitados são preferenciais. Para uma decomposição única do plano, os ladrilhamentos $[4.8^2]$ e [4.6.12] não são aplicados. Comparando os ladrilhamentos quadrado $[4^4]$ e triangular $[6^3]$, vemos que eles se diferem em termos de *adjacência* e *orientação*. Então, o ladrilhamento $[4^4]$ é mais adequado que o ladrilhamento $[6^3]$, quando

a orientação uniforme e a distancia de adjacência mínima são propriedades consideradas importantes para um ladrilhamento. Além disso, o ladrilhamento $[4^4]$, devido à sua geometria simples, é mais usado na prática.

2.2 Representação

Métodos de decomposição geram diferentes maneiras de se representar um espaço. Normalmente a representação é escolhida para uma tarefa específica, e a escolha é altamente influenciada pelo tipo de operação a ser feita no dado. Para representar eficientemente dados multidimensionais, é preciso agregar dados próximos e similares. Uma maneira eficiente é efetuar *subdivisões* no espaço separando-o em regiões de forma hierárquica ou não. O balanço entre a eficiência do agrupamento de uma subdivisão *versus* quantidade de informação a ser armazenada é discutido nesta seção.

2.2.1 Representação Não-hierárquica

A maneira mais simples de se representar um dado multidimensional é por um *reticulado regular*, que corresponde à decomposição do espaço obtida a partir do ladrilhamento $[4^4]$. Um reticulado regular subdivide igualmente o espaço em subespaços que possuem a mesma forma e o mesmo tamanho. Essa representação é um extremo da relação subdivisão eficiente *versus* informação armazenada, pois não sendo adaptativa, não garante uma boa subdivisão, mas, por outro lado, requer quase nenhuma informação além do dado.

2.2.2 Representação Hierárquica

Uma *subdivisão hierárquica* do espaço é uma seqüência de regiões $(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_m)$ tal que a seqüência é encaixante: $\mathcal{R}_i \supset \mathcal{R}_{i+1}\ \forall i \in 0, \ldots, m-1$. Isto significa que toda região $\mathcal{R}_{i+1}$ da seqüência está contida na região antecessora $\mathcal{R}_i\ \forall i = 0, \ldots, m-1$. Por exemplo, um ladrilho hierárquico (seção 2.1) é um tipo de subdivisão hierárquica. A Figura 2.4 é um exemplo de uma subdivisão hierárquica composta de uma seqüência de cinco regiões quadradas.

Cada região da seqüência é chamada de *nó*. A região zero da seqüência, que é a região original antes de ser subdividida, é chamada de *nó raiz*, ou simplesmente *raiz*. A última região da seqüência, aquela que está contida em todas as outras, mas que não contém nenhuma outra, é chamada de *nó folha*, ou simplesmente *folha*.

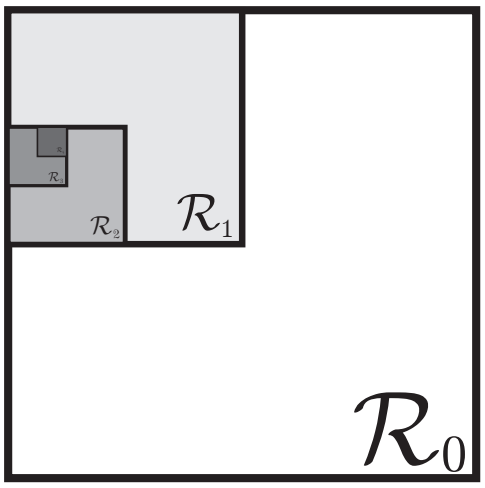


Figura 2.4: Uma subdivisão hierárquica composta de uma seqüência de cinco regiões quadradas.

O nó que corresponde à posição *zero* da seqüência representa o *nível zero* da subdivisão hierárquica, o nó que corresponde à posição *um* da seqüência representa o *nível um* da subdivisão hierárquica, o nó que corresponde à posição *dois* da seqüência representa o *nível dois* da subdivisão hierárquica, e assim por diante. O *nível máximo*, ou apenas *nível* de uma subdivisão hierárquica, é a quantidade de nós da seqüência.

Existem relações que dependem de uma referência: seja S_i com $i \in \{0, \ldots, m\}$ algum nó da seqüência, então o nó S_j com $j \in \{0, \ldots, i-1\}$ pertence à *ascendência* do nó S_i. Por outro lado, o nó S_k com $k \in \{i+1, \ldots, m\}$ pertence à *descendência* do nó S_i. O nó S_{i-1} é chamado de *nó pai*, ou simplesmente *pai*, em relação ao nó S_i $\forall i = 1, \ldots, m$. Já o nó S_{i+1} é chamado de *nó filho*, ou simplesmente *filho*, em relação ao nó S_i $\forall i = 0, \ldots, m-1$.

Uma *representação hierárquica* é uma representação constituída de subdivisões hierárquicas. O *nível* de uma representação hierárquica é o maior dos níveis máximos de suas subdivisões hierárquicas. Um ladrilhamento hierárquico (seção 2.1), por exemplo, é um tipo de representação hierárquica.

2.2.3 Particionamento binário do espaço

Particionamento binário do espaço ou *bsp* é uma representação hierárquica do espaço d-dimensional, em que as suas subdivisões hierárquicas são constituídas de seqüências de subconjuntos convexos subdivididos através de hiperplanos. Existem muitas maneiras de escolher os hiperplanos de subdivisão, por isso, essa representação hierárquica necessita de muita informação armazenada, pois cada nó deve conter informações do seu hiperplano de sub-

divisão, que nesse caso possui posição e direção arbitrárias. A representação *bsp* é o outro extremo da relação subdivisão eficiente *versus* informação armazenada, pois subdivide o espaço da maneira mais adaptável possível, mas exige um grande armazenamento de informação.

Para uma nuvem de pontos, uma maneira eficiente de se construir uma *bsp*, para agrupar pontos similares, encontra-se em (5): em cada passo da subdivisão o hiperplano de subdivisão possui a posição que corresponde à mediana dos pontos pertencentes à região a ser subdividida, e a direção do hiperplano é a direção da primeira componente principal dos mesmos pontos. Essa escolha faz com que a *bsp* tenha aproximadamente o mesmo número de pontos por nó folha.

2.2.4 KD-*Tree*

A representação hierárquica *kd-tree* é um caso particular da *bsp*, onde são usados apenas hiperplanos de subdivisão que são perpendiculares aos eixos coordenados, com o objetivo de diminuir a quantidade de informação armazenada. Existem muitas maneiras de escolher os hiperplanos de subdivisão, mas necessariamente, devem ser perpendiculares aos eixos, exigindo menos informação a ser armazenada, pois cada nó deve conter informações do seu hiperplano de subdivisão, que nesse caso possui uma posição arbitrária, porém, sua direção é conhecida, devendo ser perpendicular a um dos eixos coordenados. A *kd-tree* é também uma representação eficiente, um pouco menos que a *bsp*, pois limita a maneira de subdividir o espaço com o objetivo de reduzir o armazenamento de informação. É um bom equilíbrio da relação subdivisão eficiente *versus* informação armazenada.

Para uma nuvem de pontos, a maneira clássica de se construir uma *kd-tree* é: em cada passo da subdivisão, a posição do hiperplano de subdivisão é a mediana dos pontos pertencentes à região a ser subdividida, e as escolhas dos eixos coordenados para alinhar o hiperplano devem formar um ciclo ordenado. Essa escolha faz com que a *kd-tree* tenha aproximadamente o mesmo número de pontos por nó folha. A escolha do ponto mediano é necessária apenas para o balanceamento, e não uma exigência da *kd-tree*.

2.2.5 $2^{\mathrm{d}}-$*Tree*

A 2^{d}–*tree* é uma representação hierárquica do espaço d-dimensional, em que as suas subdivisões hierárquicas são constituídas de seqüências de subespaços $(\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m)$ tais que todo subespaço $\mathcal{Q}_{i+1}$ da seqüência tem

seu volume menor numa razão de 2^d do que o volume do seu subespaço antecessor $\mathcal{Q}_i$:

$$V(\mathcal{Q}_{i+1}) = \frac{V(\mathcal{Q}_i)}{2^d} \quad \forall i = 0, \ldots, m-1.$$

As posições dos subespaços são fixas e independem do espaço, exigindo pouca informação a ser armazenada.

2.3 Buscas

As buscas têm o objetivo de acessar os dados similares agregados pelas representações hierárquicas. Existem vários tipos de busca para as estruturas hierárquicas. Na sua maioria, são procedimentos que procuram informações de proximidade, retornando o endereço de armazenamento daquele pedido. Em alguns casos, o procedimento procura vários endereços de armazenamento que estão próximos daquele dado de entrada.

A forma clássica de efetuar esses procedimentos de busca numa estrutura hierárquica se baseia, na sua maioria, em percorrer a hierarquia, começando pela raiz, e perguntando a cada nó se tal procedimento de busca procede na sua descendência. Para o caso positivo, é perguntado para os nós filhos daquele nó, se tal procedimento de busca procede na sua descendência, e assim por diante, até encontrar um nó folha que o procedimento de busca seja positivo. O nó folha é o fim da hierarquia, e onde geralmente estão as informações buscadas.

2.4 Indexação

A indexação de uma representação é uma enumeração de dados, associando os similares a um mesmo índice. Um fator importante na escolha da representação de dados multidimensionais está no fato de que a estrutura resultante minimiza o número de índices que não correspondem a nenhum dado. Isto tem um impacto crítico no desempenho de operações como armazenamento, atualização e buscas. Ter uma estrutura, porém, que satisfaça essas exigências nem sempre é possível. Como as técnicas para dados *uni*dimensionais são mais conhecidas, dados multidimensionais devem ser primeiramente mapeados em dados *uni*dimensionais, e então representá-los usando uma estrutura adequada. Como num espaço unidimensional existe uma *ordem* natural dos elementos, deve-se definir uma *ordem* coerente no espaço multidimensional, para que o mapeamento esteja bem definido preservando localização espacial. A primeira subseção descreve alguns tipos mais conhecidos de ordenações do plano.

No caso de uma representação hierárquica, em que as subdivisões hierárquicas geram regiões (nós) similares, é possível definir um único índice

para cada nó, chamado de *código local*, que o identifica dentro da estrutura hierárquica. Assim é possível buscar um nó apenas pelo seu código local e não pela hierarquia, fornecendo acesso direto ao nó. De uma maneira informal, permite-se chamar um nó diretamente, sem ter que referenciar a sua família. A segunda subseção descreve alguns tipos mais conhecidos de códigos locais para uma *quadtree* ($2^2 - tree$).

2.4.1 Tipos de Ordenação

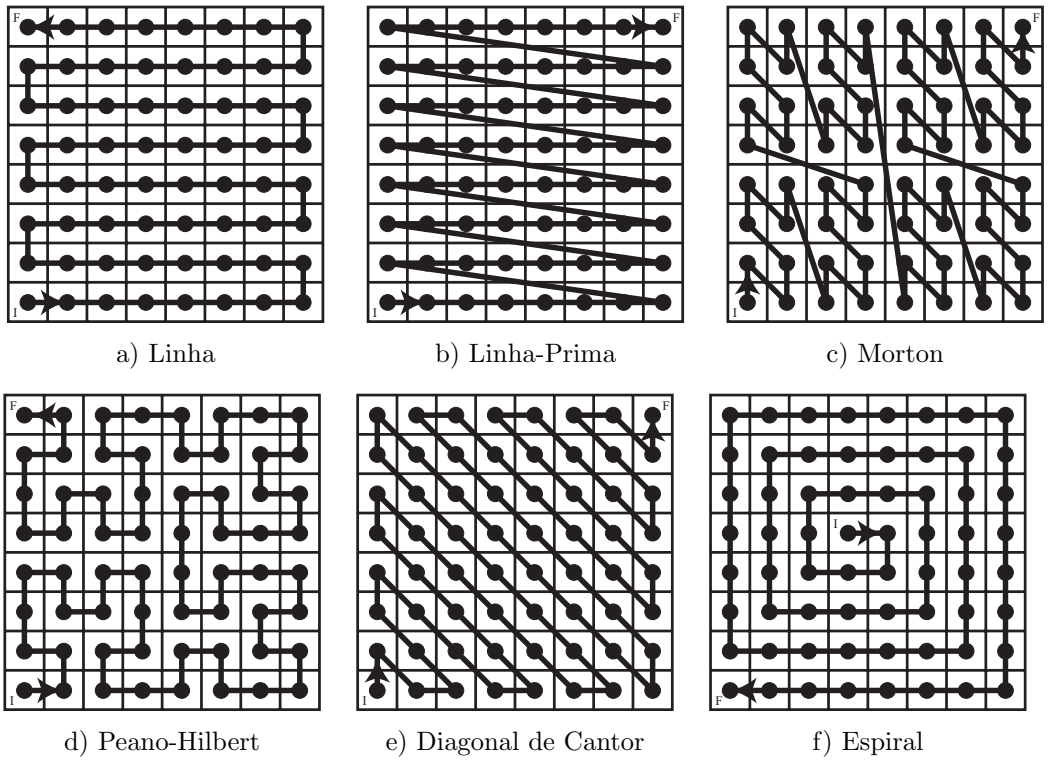


Figura 2.5: Exemplos de ordenação.

O primeiro propósito dos métodos de ordenação é o de aperfeiçoar o armazenamento, e processar seqüências de dados multidimensionais, mapeando-as em uma única dimensão. Goodchild e Grandfield (8) discutem vários métodos de ordenações espaciais, sendo que alguns estão ilustrados na Figura 2.5 e nas tabelas abaixo. Cada um tem características diferentes. A ordem *linha* (Figura 2.5(a)), também conhecida como *raster-scan*, e a ordem *linha-prima* (Figura 2.5(b)) são similares, bem como a ordem *Morton* (Figura 2.5(c), (28)) e a ordem *Peano-Hilbert* (Figura 2.5(d), (30, 10)). A primeira diferença entre as ordens linha e linha-prima, e entre as ordens Morton e Peano-Hilbert, é que as primeiras têm a propriedade de que todo elemento é vizinho do elemento anterior na seqüência, tornando assim o grau de localidade ligeiramente superior. Ambas as ordens Morton e Peano-Hilbert esgotam hierarquicamente

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Tabela 2.1: Posições de exemplo

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Tabela 2.2: Ordenação Linha

15	14	13	12
8	9	10	11
7	6	5	4
0	1	2	3

Tabela 2.3: Ordenação Linha Prima

5	7	13	15
4	6	12	14
1	3	9	11
0	2	8	10

Tabela 2.4: Ordenação Morton

5	6	9	10
4	7	8	11
3	2	13	12
0	1	14	15

Tabela 2.5: Ordenação Peano-Hilbert

9	10	14	15
3	8	11	13
2	4	7	12
0	1	5	6

Tabela 2.6: Ordenação Diagonal de Cantor

6	7	8	9
5	0	1	10
4	3	2	11
15	14	13	12

Tabela 2.7: Ordenação Espiral

um sub-quadrante antes de passar para o próximo. A ordem Peano-Hilbert possui propriedade de percorrer o espaço continuamente. A ordem Morton é simétrica, ao contrário da ordem Peano-Hilbert. A geração do índice da posição na seqüência, de um elemento do espaço, não é tão fácil para a ordem Peano-Hilbert como é para as demais ordens.

As outras ordens da Figura 2.5 são as ordens *diagonal de Cantor* (Figura 2.5(e)) e a *espiral* (Figura 2.5(f)). A ordem diagonal de Cantor percorre a imagem, saindo de um ponto da origem, visitando os elementos numa ordem similar à ordem linha-prima, com a diferença de que os elementos são visitados na ordem em que aumentam as suas distâncias $l1$ (para $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ temos $d_m = |x_1 - x_2| + |y_1 - y_2|$). Então, esta ordem é boa para ordenar um espaço que é ilimitado em duas direções a partir do primeiro quadrante. Por sua vez, a ordem espiral é atraente quando o espaço é ilimitado em quatro direções saindo da origem.

As ordens mais interessantes para estruturas de dados hierárquicas são as ordens Morton e Peano-Hilbert, por possuírem a propriedade de exaurirem o quadrante de forma hierárquica, podendo também ser usadas para ordenar um

espaço que teria sido agregado em quadrados. Das duas ordens, a mais usada é a ordem Morton, por causa de sua simplicidade no processo de conversão entre o índice e a posição espacial do elemento multidimensional.

2.4.2 Código Local

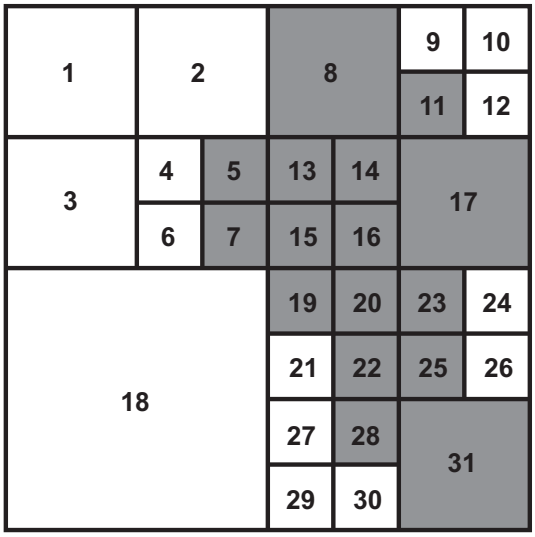


Figura 2.6: Exemplo de uma *quadtree*

Um *código local* é uma seqüência de dígitos ou símbolos que forma um índice de um nó numa representação hierárquica. Cada símbolo representa uma orientação da posição do nó, ao longo do seu caminho até a raiz. Em alguns casos, o nível do nó, ou é gravado no código local e calculado a partir dele, ou armazenado separadamente, podendo ter um número de dígitos fixos ou variados. Na Figura 2.6 está um exemplo de uma *quadtree* (2^2-*tree*), com índices numerados de acordo com o percorrimento em profundidade da árvore. Serão dados abaixo três exemplos de códigos locais numa *quadtree*.

Código Local Simples Considere uma *quadtree* de nível n, tendo sua raiz denotada por R e P um nó de nível m com $m \leq n$. Seja a seqüência de nós $[P_0, \ldots, P_m]$ o caminho de nós numa *quadtree*, que vai da raiz R até o nó P com $P_0 = R$, $P_m = P$ e $P_i = PAI(P_{i+1})$, para $i = 0, \ldots, m-1$, onde a função PAI retorna o nó pai do nó de entrada da função. Considere o caminho de orientações dos filhos numa *quadtree* que vai da raiz R até o nó P, sendo a seqüência de códigos de orientação $< C_0, \ldots C_{m-1} >$ com $C_i = FILHO(P_i)$, para $i = 0, \ldots, m-1$, onde a função $FILHO$ retorna 0, 1, 2 ou 3, que são as orientações $NO(0)$, $NE(1)$, $SO(2)$ e $SE(3)$ de P_i com relação a P_{i-1}. O *código local simples* do nó P pertencente à *quadtree* de nível

n é dado por:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 1, \ldots, m \\ C_P^s = \sum_{i=0}^{m-1} 4^{m-i-1} C_i \end{cases}$$

O nó 13 da Figura 2.6, possui a seqüência de códigos de orientação dada por $< NE, SO, NO >$, ou seja $< 1, 2, 0 >$, e então o seu código local simples é igual a $C_{13}^s = 4^2 \cdot 1 + 4^1 \cdot 2 + 4^0 \cdot 0 = 24$. A *quadtree* da Figura 2.7 é a *quadtree* da Figura 2.6 indexada por códigos locais simples. Essa definição de código local simples exige que o nível do nó seja armazenado separadamente, pois os nós 3 e 18 da Figura 2.6, possuem o mesmo código local simples igual a 2, logo sem os seus níveis não seria possível diferenciá-los.

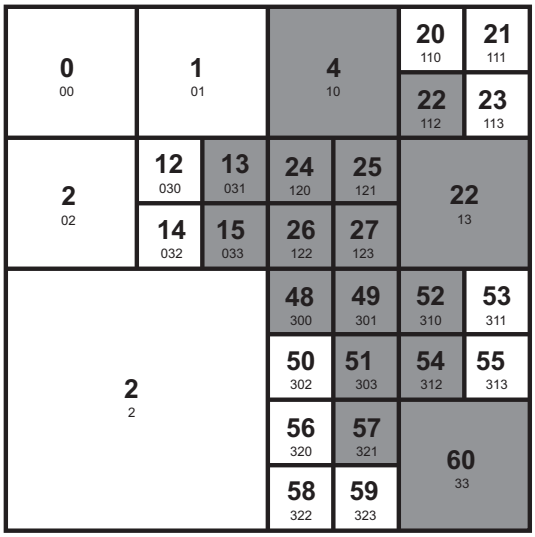


Figura 2.7: *Quadtree* codificada com o código local simples

Código Local de Tamanho Fixo Para uma formulação de código local de tamanho fixo, isto é, com quantidade de códigos de orientação sendo a mesma para qualquer nó, é preciso definir um novo código que represente uma ausência de orientação, sendo diferente das orientações usadas para a orientação dos filhos na *quadtree*. Para um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$), seja a seguinte formulação de código local:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 0, \ldots, m-1 \\ C_i = 4 & \forall i = m, \ldots, n \\ C_P^{fl} = \sum_{i=0}^{n-1} 5^{n-i-1} C_i \end{cases}$$

Essa formulação de código local é chamada de *código local de tamanho fixo* ou *código local FL* (do inglês FL = *Fixed Length*) e foi definida por

Gargantini em (15). O nó 13 da Figura 2.6 possui a seqüência de orientações dada por $< 1, 2, 0 >_{13}$, então seu código local FL é igual a $C_{13}^{fl} = 5^2 \cdot 1 + 5^1 \cdot 2 + 5^0 \cdot 0 = 35$. A *quadtree* da Figura 2.8 é a *quadtree* da Figura 2.6 indexada por códigos locais FL.

Nessa formulação, os códigos locais sempre possuem tamanhos fixos iguais a n; o número 4 representa a ausência de códigos de orientação para nós folhas que possuem níveis menores que o nível n da *quadtree*. Observe que não é preciso armazenar o nível do nó; ele pode ser calculado contando o número de divisões sucessivas desse código local por 5, cujos restos das divisões são os códigos de orientação decodificados, antes de encontrar um resto igual a 4. Quando são armazenados apenas os nós folhas, a *quadtree* é chamada de *quadtree linear FL*, podendo ainda reconstruir explicitamente a *quadtree* original a partir dos códigos locais FL (15).

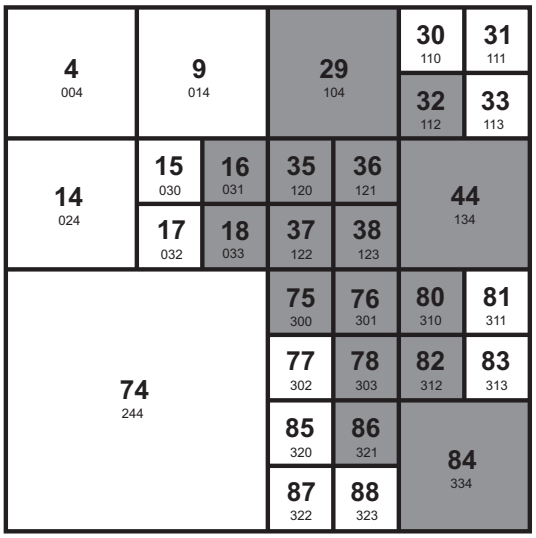


Figura 2.8: *quadtree* codificada com o código local FL

Código Local de Tamanho Variado Para a operação de decodificação do código local, são feitas divisões sucessivas, e os restos das divisões são os códigos de orientação decodificados. É importante que esse processo seja rápido e simples de se calcular. A decodificação do código local deve ser de tal modo que os dígitos sejam obtidos na ordem em que se percorre a *quadtree*, começando da raiz até as folhas. Infelizmente, da maneira que foram definidos os códigos anteriores, obtém-se os dígitos na ordem inversa. Uma codificação que satisfaz essa exigência de decodificação pode ser obtida alterando a forma do cálculo do código local de um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$),

para:

$$\begin{cases} C_i = FILHO(P_i) + 1 \quad \forall i = 0, \ldots, m-1 \\ C_P^{vl} = \sum_{i=0}^{m-1} 5^i C_i \end{cases}$$

Essa formulação de código local é chamada de *código local de tamanho variado* ou *código local VL* (do inglês VL = *Variable Length*), e foi definida por Gargantini em (7). O nó 13 da Figura 2.6, possui a seqüência de orientações dada por $< 2, 3, 1 >$, então seu código local VL é igual a $C_{13}^{vl} = 5^0 \cdot 2 + 5^1 \cdot 3 + 5^2 \cdot 1 = 42$. A *quadtree* da Figura 2.9 é a *quadtree* da Figura 2.6 indexada por códigos locais VL. Uma representação de nós de uma *quadtree* usando qualquer tipo de código local tem o potencial de economizar espaço, quando comparado com a representação explícita de uma *quadtree*; mais espaço ainda pode ser economizado se armazenarmos apenas os códigos locais dos nós folhas. A lista de códigos locais VL dos nós folhas de uma *quadtree* é chamada de *quadtree linear VL*, podendo ainda reconstruir explicitamente a *quadtree* original, a partir dos códigos locais VL (7).

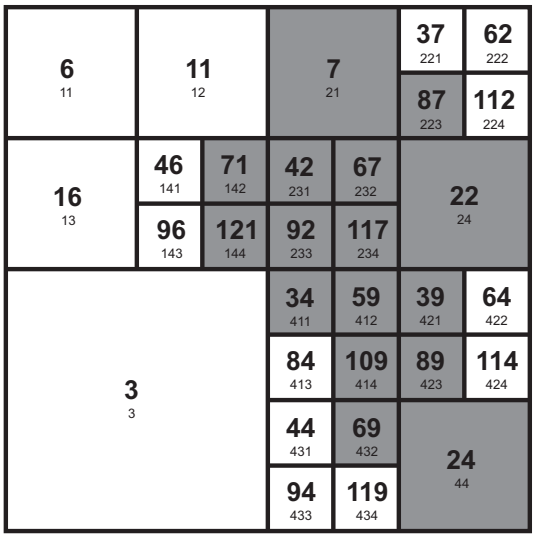


Figura 2.9: *Quadtree* codificada com o código local VL

Tamanho Fixo *versus* **Tamanho Variado** O código local FL tem várias desvantagens quando comparado com o código local VL. Uma delas é que os códigos locais são grandes, pois todos possuem sempre n dígitos, ao contrário dos códigos locais VL, cujos números de dígitos são variáveis. Outra desvantagem é que refinando a *quadtree* em mais um nível, é preciso recodificar os nós existentes multiplicando todos os códigos por 5; já os códigos locais VL não se alteram com o refinamento da *quadtree*. E também, quando é preciso obter o caminho da raiz da árvore até um nó específico, a decodificação do

código local VL já está na ordem certa de percorrimento, enquanto que a decodificação do código local FL está na ordem inversa.

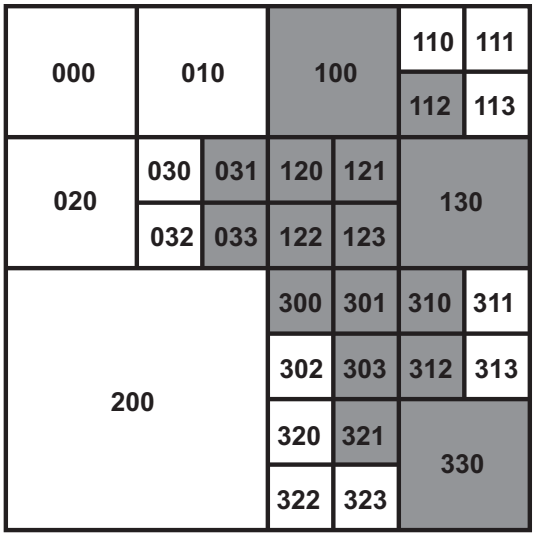


Figura 2.10: *Quadtree* codificada com o código local FD

Código Local de Profundidade Fixa Para evitar o uso de operações de módulo e divisões inteiras no processo de decodificação, Gargantini em (14) introduziu o chamado *código local de profundidade fixa* ou *código local FD* (do inglês FD = *Fixed Depth*), também conhecido como *quadnode*. Ele consiste da seqüência de códigos das orientações NO, NE, SO e SE representados pelos números 0, 1, 2 e 3, respectivamente, similar aos códigos locais simples, VL e FL, mas sem a codificação. Para uma *quadtree* de nível n, o código local FD de um nó de nível m ($m < n$) possui n dígitos, cujos $n - m$ últimos são preenchidos por 0. Sua formulação para um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$), é a seguinte:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 0, \ldots, m-1 \\ C_i = 0 & \forall i = m, \ldots, n \\ C_P^{fd} = \sum_{i=0}^{n-1} 10^{n-i-1} C_i \end{cases}$$

O nó 13 da Figura 2.6 possui a seqüência de orientações dada por $< 1, 2, 0 >$; então seu código local FD é igual a $C_{13}^{fd} = 120$. A *quadtree* da Figura 2.10 é a *quadtree* da Figura 2.6 indexada por códigos locais FD. As operações de decodificação são evitadas, pois o código já está na base 10, e os códigos orientação são os dígitos do código local FD. Sua única desvantagem é que o nível não é conhecido, tendo que ser armazenado separadamente. Por

exemplo, para um nó com código local FD igual a 0, não se sabe o seu nível; o mesmo acontece para o nó 1 da Figura 2.6.

2.5 Armazenamento

Esta seção descreve alguns dos tipos mais conhecidos de armazenamento de estruturas hierárquicas. O armazenamento na forma de árvore possui um acesso em média logarítmico, mas possui uma sobrecarga associada à sua hierarquia. Já num armazenamento numa *hash table*, o acesso varia entre linear e constante.

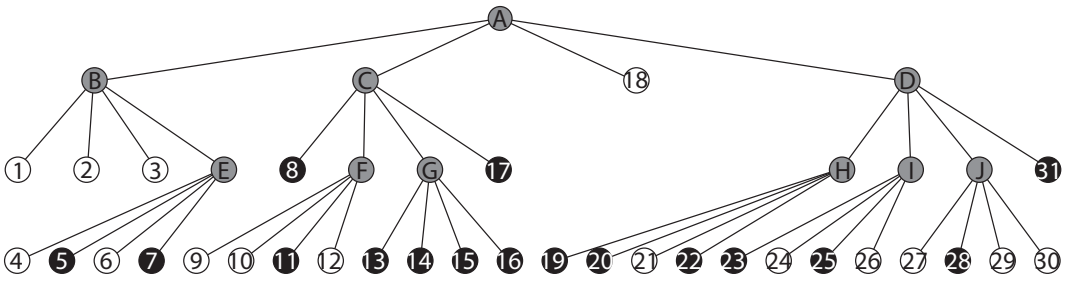
2.5.1 Vetor

É maneira mais simples e conhecida de se armazenar dados. A maioria das linguagens possui um formato nativo de armazenamento em vetor.

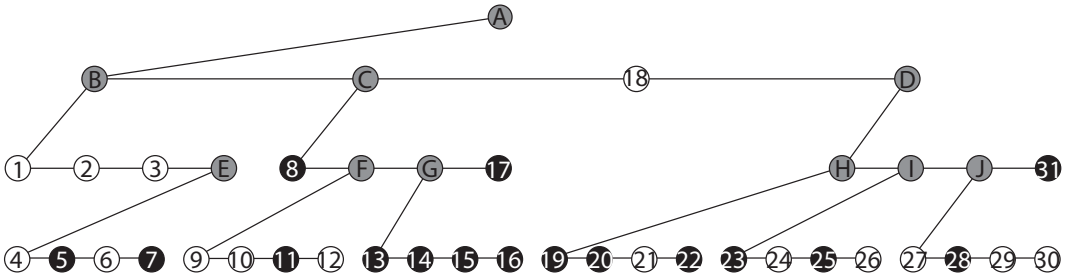
2.5.2 Árvore

Uma *árvore* é uma estrutura de armazenamento hierárquica que imita a estrutura de árvore da natureza, com o conjunto de raiz e nós conectados, mas com a diferença de possuírem sua raiz no topo e os ramos para baixo. O vocabulário é similar ao das representações hierárquicas. O *nó raiz*, ou apenas *raiz*, é o nó no topo da árvore. O *nó pai*, ou apenas *pai*, de um nó qualquer, é aquele que está diretamente conectado acima dele na hierarquia. *Nó filho*, ou apenas *filho*, de um nó qualquer, está diretamente conectado abaixo dele na hierarquia. Nós com o mesmo nó pai são chamados de *nós irmãos*, ou apenas *irmãos*. Os nós chamados de *nós folhas*, ou apenas *folhas*, são aqueles que não possuem nós filhos. Note que cada nó da árvore pode ser considerado um nó raiz para a sub-árvore que contém todos os nós que estão conectados abaixo dele na hierarquia. Um *ramo* é uma seqüência de nós tal que o primeiro é o pai do segundo, o segundo é o pai do terceiro, e assim por diante. O *tamanho de um ramo* é o número de nós da seqüência. A *profundidade* ou *nível* de uma árvore é o tamanho do maior ramo que ela possui. As operações de busca e armazenamento numa árvore possuem tempos de acesso com uma complexidade média que varia entre linear no pior caso, e logarítmica no melhor caso, dependendo do número de filhos.

A implementação de uma árvore pode ser feita de várias formas. Na maneira tradicional, cada nó pai possui um ponteiro para cada filho, tendo opcionalmente um ponteiro para o pai. Chamaremos esse tipo de *implementação*



a) Implementação clássica



b) Implementação filho-irmão

Figura 2.11: Armazenamento da *quadtree* da Figura 2.6 na forma de árvore com diferentes implementações.

clássica. Outra forma ocorre quando cada nó pai possui um ponteiro para uma lista encadeada de filhos, tendo opcionalmente um ponteiro para o pai, que chamaremos de *implementação filho-irmão*. Na Figura 2.11 encontra-se exemplos de um armazenamento da *quadtree* da Figura 2.6 na forma de árvore com diferentes implementações. A Figura 2.11(a) é a implementação clássica, e a Figura 2.11(b) é a implementação filho-irmão.

Árvore Binária A *árvore binária* é uma árvore que possui no máximo dois filhos por nó. Assim cada nó pode ser identificado como sendo o filho esquerdo ou o filho direito. A *árvore binária de busca* é uma árvore binária que possui uma ordem pré-determinada no dado armazenado em cada nó. Como, por exemplo, o filho esquerdo armazena um valor sempre menor do que o armazenado no filho direito do mesmo pai. As operações de busca e armazenamento numa árvore binária possuem tempos de acesso com uma complexidade média que varia entre linear igual a $O(n)$ no pior caso, e logarítmica igual a $O(\log_2(n))$ no melhor caso, onde n é o nível da árvore.

Árvore AVL A árvore *AVL* é uma árvore binária auto-balanceada, no sentido de que o nível de duas sub-árvores de qualquer nó se diferem de no máximo um. O fator de balanço de um nó é o nível da sua sub-árvore da direita, menos o nível da sua sub-árvore da esquerda. Um nó com fator de balanço 1, 0, ou -1 é considerado balanceado. Um nó com qualquer outro fator de balanço

é considerado desbalanceado, exigindo assim uma modificação na árvore. O fator de balanço de cada nó é armazenado no próprio nó, ou calculado a partir dos níveis das suas sub-árvores. A árvore AVL tem esse nome devido aos seus dois inventores: Adelson-Velsky & Landis (1). As operações de busca e armazenamento numa árvore AVL possuem tempos de acesso com uma complexidade logarítmica igual a $\Theta(\log_2(n))$, onde n é o nível da árvore.

2.5.3 *Hash Table*

A *hash table* é uma estrutura de dados similar a uma agenda de telefone, podendo rapidamente determinar se um item pertence ou não a um subconjunto, olhando a primeira letra do nome. Sua principal idéia é a de diminuir as buscas necessárias durante o acesso ou armazenamento dos dados, pela divisão do conjunto de itens em subconjuntos menores, de tal forma que cada busca pertença apenas a um subconjunto. Um subconjunto é determinado mapeando todos os itens num conjunto de inteiros por uma função, armazenando os itens nas posições definidas por esses inteiros em um vetor.

Mais especificamente, seja $\mathcal{U}$ um conjunto de n itens a ser armazenados em um vetor $T[0, 1, \ldots, m-1]$ de tamanho m. A *função de hash* é a função

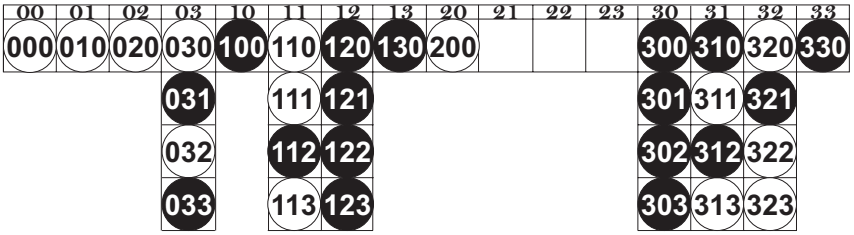
$$\begin{array}{rccc} h: & \mathcal{U} & \rightarrow & \{0, 1, \ldots, m-1\} \\ & x & \mapsto & h(x) \end{array}$$

que mapeia cada item $x \in \mathcal{U}$ num recipiente $T[h(x)]$ de posição $h(x)$ do vetor $T[0, 1, \ldots, m-1]$. Esse vetor de recipientes é chamado de *hash table*.

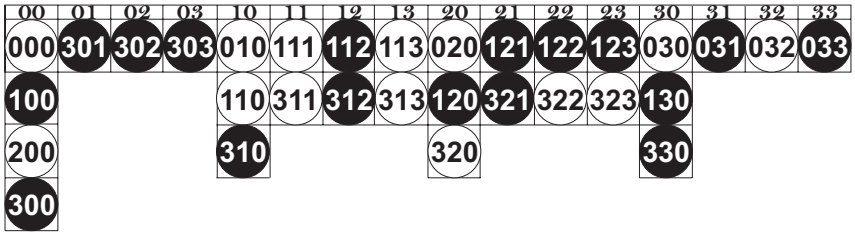
Se $n = m$, então a função de *hash* pode ser definida como $h(x_i) = i$. Tal função de *hash bijetora*, que mapeia todos os itens na *hash table*, preenchendo todos os recipientes, sem criar colisões, é chamada de *função de hash perfeita*. Tais funções só podem ser geradas para um conteúdo estático, limitando as suas aplicações. Para os outros casos, existirão itens x e $y \in \mathcal{U}$ tais que $h(x) = h(y)$. Quando isso acontece, é dito que x e y *colidem*. Os métodos mais comuns para resolver as *colisões* são o *método do encadeamento* e o *método do endereçamento aberto*. Uma discussão detalhada da diferença dos dois métodos pode ser encontrada em (27).

Método do Encadeamento Uma *hash table* onde cada recipiente $T[h(x_i)]$ não possui apenas um item, mas uma lista encadeada de itens $(h(x_i^1), \ldots, h(x_i^j))$ que possuem a mesma função de *hash* $h(x_i^1) = \cdots = h(x_i^j)$, é chamada de *hash table encadeada*. Para verificar se um item x está na *hash table*, é preciso varrer toda a lista encadeada $T[h(x_i)]$. Então o tempo total de

remoção e inserção de um item x numa *hash table* encadeada é de $O(l(x))$, onde $l(x)$ é o tamanho da lista encadeada $T[h(x)]$.



a) A função de *hash* é igual aos dois *primeiros* dígitos do código local FD



b) A função de *hash* é igual aos dois *últimos* dígitos do código local FD

Figura 2.12: Armazenamento da *quadtree* da Figura 2.6 na forma de *hash table* com diferentes funções de *hash*.

A Figura 2.12 representa o armazenamento apenas dos nós folhas da *quadtree* da Figura 2.6 na forma de uma *hash table* encadeada com funções de *hash* diferentes, onde os itens são os nós da *quadtree* com os seus respectivos códigos locais FD sem os níveis. A função de *hash* da Figura 2.12(a) é igual aos dois primeiros dígitos do código local FD de cada nó. Já a função de *hash* da Figura 2.12(b) é igual aos dois últimos dígitos do código local FD de cada nó. A existência de recipientes vazios e várias listas encadeadas de colisões contendo quatro entradas na *hash table* da Figura 2.12(a) mostra um exemplo de uma função de *hash* que não é eficiente, ao contrário do que acontece na *hash table* da Figura 2.12(b).

No pior caso, todos os n itens seriam registrados num mesmo inteiro, e por conseqüência disto seriam armazenados num mesmo recipiente. Existiria então apenas uma longa lista encadeada de n entradas, logo o tempo de acesso teria uma complexidade linear igual a $O(n)$. Para calcular o tempo esperado de uma busca sem sucesso de um item x não pertencente ao conjunto $\mathcal{U}$ de n itens, que está armazenado em uma *hash table* encadeada $T[0, 1, \ldots, m-1]$ de tamanho m, supõe-se usualmente que a função de *hash* é completamente aleatória, isto é, $\forall x$ e $y \in \mathcal{U}$, tais que $x \neq y$, a probabilidade de serem mapeados num mesmo valor $h(x) = h(y)$ é igual a $1/m$: $x \neq y \Rightarrow P[h(x) = h(y)] = 1/m$. Se o item buscado x não pertence ao conjunto $\mathcal{U}$, o valor esperado de $l(x)$ pode

ser calculado definindo a variável indicadora

$$C_{x,y} = \begin{cases} 0 & \text{se } h(x) \neq h(y) \\ 1 & \text{se } h(x) = h(y) \end{cases} .$$

E então temos que

$$x \neq y \Rightarrow P[h(x) = h(y)] = E[C_{x,y}] = \frac{1}{m}.$$

Como o tamanho da lista encadeada $T[h(x)]$ é igual à quantidade de itens que colidem com x, logo

$$l(x) = \sum_{y \in T} C_{x,y}.$$

Portanto, o valor esperado de $l(x)$ é

$$E[l(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}.$$

Essa razão é chamada de *fator de balanço* de uma *hash table*, e como ela independe do item escolhido, é definida como sendo

$$\alpha = \frac{n}{m}.$$

A análise feita implica que o tempo médio esperado para uma busca sem sucesso numa *hash table* encadeada é $\Theta(\alpha)$. Então, enquanto a quantidade n de itens difere do tamanho m da *hash table* de uma constante, o tempo de busca é constante. Uma busca com sucesso possui um tempo esperado menor do que uma busca sem sucesso, logo o tempo esperado para uma busca com sucesso numa *hash table* encadeada é constante. O mesmo acontece para a inserção e para a remoção.

Listas encadeadas não são as únicas estruturas de armazenamento que podem ser usadas para um armazenamento de uma *hash table* encadeada, qualquer estrutura de armazenamento pode ser usada para armazenar as listas encadeadas. Por exemplo, se é possível definir uma ordem no conjunto $\mathcal{U}$, cada lista encadeada pode ser armazenada numa árvore binária de busca. Isso reduziria o tempo médio de busca no pior caso para $\Theta(\log_2 l(x))$, e sobre a suposição de uma função de *hash* aleatória, o tempo de busca esperado seria de $\Theta(\log_2 \alpha)$.

Outra possibilidade é armazenar as listas encadeadas em outras *hash tables*. Para ser eficiente, deve-se garantir que o fator de balanço para as *hash tables* secundárias é sempre uma constante menor do que 1, fazendo assim com

que o fator de balanço da *hash table* principal seja também constante.

Método do Endereçamento Aberto Outro método capaz de resolver as colisões é chamado de *endereçamento aberto*. Ao invés de depender de estruturas de armazenamento secundárias, este método resolve as colisões investigando por outro lugar na *hash table*. Considere uma seqüência de funções de *hash* $\{h_0, h_1, \ldots, h_{m-1}\}$, tais que para qualquer item $x \in \mathcal{U}$ a *seqüência investigadora* $\{h_0(x), h_1(x), \ldots, h_{m-1}(x)\}$ é uma permutação de $\{0, 1, \ldots, m-1\}$. Isto significa que diferentes funções de *hash* da seqüência mapeiam x em diferentes posições na *hash table*. Note que sempre se deve ter $n \geq m$ e então o fator de balanço é sempre menor do que um: $\alpha \leq 1$. Ao inserir um item x, se a primeira posição $h_0(x)$ estiver ocupada na *hash table*, outra posição $h_1(x)$ é sugerida, e assim por diante, até encontrar uma posição $h_i(x)$ vazia, no caso em que a *hash table* não estiver completamente cheia.

A busca de um item x numa *hash table* com endereçamento aberto é feita da seguinte forma: começando com a primeira função de *hash* h_i $(i = 0)$ da seqüência $\{h_0, h_1, \ldots, h_{m-1}\}$, se o item x estiver na posição $h_i(x)$ da *hash table*, $x = T[h_i(x)]$, o item x foi encontrado e a busca acaba. Caso contrário, se essa posição estiver vazia, $T[h_i(x)] = \varnothing$, é porque o item x não está na *hash table*. E, por fim, se a posição $h_i(x)$ não está vazia, $T[h_i(x)] \neq \varnothing$, é feito o mesmo processo com a segunda função de *hash* da seqüência, e assim por diante. Se todas as funções de *hash* foram testadas e não se tem um veredito, é porque o item x não está na *hash table*, e a mesma está completamente cheia.

A inserção de um item x numa *hash table* com endereçamento aberto se comporta da mesma forma, mas com uma única diferença: quando a posição $h_i(x)$ estiver vazia, $T[h_i(x)] = \varnothing$, o item x é inserido nessa posição, $T[h_i(x)] = x$. E, ao final, quando todas as funções de *hash* forem testadas e o item x não foi inserido, conclui-se que a *hash table* está completamente cheia.

A remoção de um item numa *hash table* com endereçamento aberto é um pouco mais complicada. Não se pode simplesmente remover um item da *hash table*, pois é preciso saber que o recipiente foi preenchido em algum momento para garantir a sincronia, ao inserir ou buscar itens das funções de *hash* da seqüência investigadora. Portanto, quando um item é removido de um recipiente, deve ser marcado como sendo um recipiente *desperdiçado*. É claro que ao inserir e remover muitos itens, a *hash table* com endereçamento aberto ficaria com muitos recipientes vazios, mas que foram anteriormente marcados como desperdiçados, impedindo uma nova inserção. Por isso, para garantir um bom desempenho, é preciso respeitar duas regras: primeira, quando o número de itens armazenados numa *hash table* de tamanho m exceder $\frac{m}{4}$,

deve-se construir uma nova *hash table* com o tamanho dobrado; segunda, se o número de recipientes marcados ultrapassar $\frac{m}{2}$, deve-se construir uma nova *hash table* que utilize recipientes marcados.

Para calcular o tempo esperado de uma busca sem sucesso de um item x não pertencente ao conjunto $\mathcal{U}$ de n itens, que está armazenado em uma *hash table* com endereçamento aberto $T[0, 1, \ldots, m-1]$ de tamanho m, é preciso observar que o valor inicial $h_0(x)$ é igual a algum inteiro do conjunto $\{0, 1, \ldots, m-1\}$ pela definição da seqüência investigadora. Usando o modelo anterior, isto implica que a probabilidade do recipiente $T[h_0(x)]$ estar ocupado é igual a $\frac{n}{m}$. Se a primeira investigação for ignorada, a seqüência investigadora resultante $\{h_1(x), \ldots, h_{m-1}(x)\}$ é igual a alguma permutação do conjunto menor $\{0, 1, \ldots, m-1\} \setminus \{h_0(x)\}$ (sem o elemento $h_0(x)$). Isto implica que se o recipiente $T[h_0(x)]$ estiver ocupado, a busca segue recursivamente. Para usar essa recursão, estará suposto que o recipiente $T[h_0(x)]$ nunca será investigado. Então, a recorrência do número esperado de investigações, em função de m e n, implica:

$$E[T(m,n)] = 1 + \frac{n}{m} E[T(m-1, n-1)].$$

O caso trivial é $T(m, 0) = 1$. Se a *hash table* estiver vazia, a primeira investigação sempre retorna um recipiente vazio. Será provado por indução que $E[T(m,n)] \leq \frac{m}{(m-n)}$:

$$\begin{aligned} E[T(m,n)] &= 1 + \tfrac{n}{m} E[T(m-1, n-1)] \\ &\leq 1 + \tfrac{n}{m} \cdot \tfrac{m-1}{m-n} && \text{[hipótese de indução]} \\ &\leq 1 + \tfrac{n}{m} \cdot \tfrac{m}{m-n} && [m-1 < m] \\ &= \tfrac{m}{m-n} \, . \end{aligned}$$

Reescrevendo em termos do fator de balanço $\alpha = \frac{n}{m}$, tem-se que $E[T(m,n)] \leq \frac{1}{(1-\alpha)}$. Então, o tempo médio esperado de uma busca sem sucesso numa *hash table* com endereçamento aberto é constante, pois $\alpha \leq 1$, a não ser que a *hash table* esteja quase toda completa. O mesmo acontece para uma busca com sucesso e para uma inserção.

Na prática, a seqüência de funções de *hash* que investigam a existência de um item x na *hash table* é criada de acordo com uma das seguintes heurísticas:

Investigação Linear: Dada uma função de *hash* $h(x)$, define-se h_i como sendo $h_i(x) = (h(x) + i) \mod m$. Uma seqüência simples, mas as colisões tendem a agrupar os itens deixando muitos buracos na função de *hash*. Para garantir que a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ seja uma permutação do conjunto $\{0, 1, \ldots, m-1\}$, é preciso ter m relativamente primo com i.

Investigação Quadrática: Dada uma função de *hash* $h(x)$, define-se h_i como sendo $h_i(x) = (h(x)+i^2) \mod m$. Infelizmente, para alguns valores de m a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ não é uma permutação do conjunto $\{0, 1, \ldots, m-1\}$. Isto pode ser evitado tomando m como sendo um número primo. Embora a investigação quadrática não sofra dos mesmos problemas que a investigação linear, ela possui o seguinte problema de agrupamento: se dois itens possuem o mesmo valor inicial, as respectivas seqüências investigadoras serão as mesmas.

Hashing **Duplo:** Dadas duas funções de *hash* $h(x)$ e $h'(x)$, define-se h_i como sendo $h_i(x) = (h(x) + i \cdot h'(x)) \mod m$. Para garantir que a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ seja uma permutação do conjunto $\{0, 1, \ldots, m-1\}$, é necessário que os valores da função de *hash* $h'(x)$ e o tamanho da função de *hash* m sejam primos entre si, bastando simplesmente tomar m como sendo um primo. O duplo *hashing* evita os problemas de agrupamento que as investigações lineares e quadráticas possuem.

Enfim, o método do endereçamento aberto é mais apropriado quando se tem uma pequena quantidade de itens para armazenar, pois, por não possuir um armazenamento externo, o número de elementos deve ser menor ou igual ao tamanho da *hash table*. E, na prática, para uma *hash table* com poucos recipientes vazios, as seqüências de funções de *hash* têm mais dificuldade de armazenar um item, podendo até não conseguir preencher toda a *hash table*.

Nas análises a seguir, será considerado que a complexidade média do acesso de um item numa *hash table* é de ordem constante.