Abstract

A comprehensive presentation is made of data structures used to represent spatial data. Spatial data consists of points, lines, rectangles, regions, surfaces, and volumes. Such data is used in applications in computer graphics, computer vision, data base management systems, computer-aided design, robotics, geographic information systems, image processing, computational geometry, pattern recognition, and other areas. The focus is on hierarchical data structures such as quadtrees, octrees, and other related hierarchical representations. These representations are based on the principle of recursive decomposition. A number of operations drawn from applications in which such data structures find use are examined in greater detail.
CHAPTER 3

ALTERNATIVE QUADTREE REPRESENTATIONS

3.1.

3.1.1.

3.1.2. CONSTRUCTING AN FD LOCATIONAL CODE (insert on p. 41 at the end of Sec. 2.1.1)

There are many ways of constructing the FD locational code. At present, from the standpoint of computational complexity, all the methods are asymptoti-
cally the same. In other words, given a $2^n \times 2^n$ image, they require $O(n)$ time. Nevertheless, some practical improvements are possible by use of table lookup techniques and shift operations. In this section, we will only discuss the quadtree. However, the extension to octrees and higher dimensions is straightforward (see Exercises 2.1.2.5 and 2.1.2.6).

The simplest way to construct the locational code is to do it directly from the quadtree. This is similar to the technique described in procedure CODE1. At times, it is preferable to construct the code directly from the $x$ and $y$ coordinate values of the upper leftmost pixel in each block (assuming that the origin is in the upper left corner of the image). This technique is known as bit interleaving. In this case, there are several ways to proceed. The simplest is to apply a variant of procedure CODE1 to the binary representations of the $x$ and $y$ coordinate values in an alternating manner.

A somewhat more straightforward approach is to convert the $x$ and $y$ coordinate values individually to their interleaved coordinates under the assumption that the other value is zero, and then apply a logical or (i.e., lor) operation to the results. An even simpler approach is to temporarily ignore the distinction between the $x$ and $y$ coordinates and interleave each value with zero (i.e., insert a 0 to the left of each bit position). For example, assuming $n=3$, 6 becomes 010100. The operation of inserting the intervening leading zeros is termed dilation, and the resulting number is called a dilated integer by Schrack [Schr92], while the opposite process of converting a dilated integer to its conventional representation is termed contraction [Schr92]. Next, assuming that the $y$ coordinate is the most significant, shift the dilated integer corresponding to the $y$ coordinate value by one bit to the left, and then perform a logical or (i.e., lor) operation on it and the dilated integer corresponding to the $x$ coordinate value.

As an example of this techniques, suppose that $n=6$ and we want to compute the FD locational code of the $2 \times 2$ block whose upper leftmost pixel is at (14,26). In this case, the binary representation of the pixel’s coordinate values is
Applying dilation to 14 yields 000001010100 while applying dilation to 26 yields 000101000100. Now, shifting the dilated integer corresponding to the y coordinate value by one to the left (assuming that the incoming bit is a 0) and applying a logical or (i.e., lor) operation with the value of the x coordinate yields 001011011100 which is 732 in base 10.

The drawback of the above technique is that the process of forming the dilated integer also takes $O(n)$ time. Nevertheless, this process can be speeded up by using table lookup methods. For a $2^n \times 2^n$ image, we can obtain the FD locational code by use of a two-dimensional table which contains one entry for each possible pair of $x$ and $y$ coordinate values. Such a table requires $2^n \times 2^n$ entries which is clearly prohibitive. However, we can reduce the table size significantly by performing the lookup operation more than once [Shaf90d]. In particular, for a $2^n \times 2^n$ image, we can decompose the group of bits comprising each coordinate value into $n/m$ subgroups of $m$ bits, and use a table with $2^n \times 2^m$ entries. These subgroups of $m$ bits are successively looked up in the table, and the results are concatenated to form the desired locational code. This process also makes use of logical or (i.e., lor), logical and (i.e., land), and shift (i.e., lsh) operations. The result is that the decoding now takes time proportional to $n/m$ with $2^m$ space.

The space requirements can be reduced further by applying table lookup to the subgroups of $x$ and $y$ coordinate values separately, and then applying a logical or (i.e., lor) operation prior to reapplying the process to the remaining subgroups. In such a case, we would need two tables of $2^m$ entries apiece. The space requirements can be reduced even further to just one table of size $2^n$ by looking up both subgroups of $x$ and $y$ coordinate values in the same table (i.e., obtaining their dilated integer representations), shifting the coordinate that is deemed the most significant by one bit to the left, and then applying a logical or (i.e., lor) operation prior to reapplying the process to the remaining subgroups.

For example, if $n=12$, then using subgroups of 6 bits means that we only require a table of $2^6=64$ entries to obtain the FD locational code with just two iterations. Using subgroups of 4 bits requires a table of just $2^4=16$ entries, and we obtain the FD locational code with three iterations.

An implementation of the above encoding process is given below by procedure ENCODE. It makes use of table DIL, illustrated in Table D.1 for $m=4$, to yield the dilated integer representation of the groups of $x$ and $y$ coordinate values.

```plaintext
fd_locationalcode procedure ENCODE(N,M,X,Y); /* Given a $2^n \times 2^n$ image, apply bit interleaving in groups of M bits to compute the FD locational code of the block whose upper leftmost pixel is at location (X,Y) (assuming that the origin is in the upper leftmost corner of the image). DIL is a table indicating a mapping from integers to their dilated integer values for numbers ranging from 0 to $2^M -1$. The y coordinate is more significant than the x coordinate. Assume that N is divisible by M. */ begin value integer X,Y,N,M; global integer array DIL[0:2^M -1]; ...
```

<table>
<thead>
<tr>
<th>Table D.1. DIL[I].</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>DIL[I] 0 1 4 5 16 17 20 21 64 65 68 69 80 81 84 85</td>
</tr>
</tbody>
</table>

3-2
integer I, MASK;
fd locationalcode P;
MASK ← 2↑M − 1;
P ← 0;
for I ← 0 step 1 until (N/M)−1 do
begin
P ← P lor ((DIL[X land MASK] lor (DIL[Y land MASK] lsh 1)) lsh 2*M+1);
X ← X lsh −M; /* next subgroup of M bits of x */
Y ← Y lsh −M; /* next subgroup of M bits of y */
end;
return(P);
end;

Decoding an FD locational code to obtain the x and y coordinate values of
the upper leftmost pixel in the relevant block is the reverse of the encoding
process. Once again, for a 2^n×2^n image, this will require \(O(n)\) time. The straightforward
approach is to process the locational code two bits at a time, thereby obtaining
the values of the x and y coordinates one bit at a time. This process can be
speeded up by applying table lookup methods in an analogous manner to that used
in conjunction with the encoding method. The difference is that the tables are
larger. In particular, if we want to use the same number of groups as we did in the
encoding process, then each subgroup will have to be twice as large as the one
used in the encoding process (see Exercise 2.1.2.1).

The speeded-up decoding process requires two tables - one for each of the
coordinate values. Let \(m\) denote the number of bits of the FD locational code
that we are decoding at one time. Assuming that the y coordinate is the most
significant, we can reduce the number of tables to one by shifting the FD locational
code by one bit to the right before the table lookup operation for the y coordinate.
The size of the table is \(2^{m−1}\) since we are only interested in \(m−1\) of the bits in
either case (i.e., for the x and y coordinate).

The size of the table can be reduced further by using the dilated integer
representation of the x and y coordinate values in the table lookup operation. The
dilated integer is calculated by applying a logical and (i.e., land) operation to the
FD locational code with a mask containing a 0 and 1 in alternating positions - i.e.,
for \(n=6\), the mask is 010101010101 which is equal to \((4^n−1)/3\) (see Exercise
2.1.2.2). When we are decoding \(m\) bits at a time, then the mask is only \(m\) bits
long. The mask is also the largest possible dilated integer - i.e., \((2^m−1)/3\). Thus
the table need only contain \((2^m−1)/3+1\) entries instead of \(2^{m−1}\). Of course, \(m\)
should be a multiple of the dimension of the image (i.e., 2 in this case).

For example, if \(n=12\), then using two iterations (i.e., \(m=12\)) requires a
table of \((2^{12}−1)/3+1=1366\) entries. On the other hand, using four iterations (i.e.,
\(m=6\)) requires a table of \((2^{6}−1)/3+1=22\) entries which is more reasonable. Using
six iterations (i.e., \(m=4\)) requires a table of \((2^{4}−1)/3+1=6\), entries which is not
really worth the added expense of separate table lookup and shifting operations.

An implementation of the above decoding process is given below by procedure
DECODE. It makes use of table CON, illustrated in Table D.2 for \(m=6\), to
yield the integer representation of the dilated integers corresponding to the groups
of interleaved x and y coordinate values. Entries with a ‘‘?’’ are undefined as
they do not correspond to valid dilated integers.
procedure DECODE(P,N,M,X,Y);
/* Given a $2^N \times 2^N$ image, decode the FD locational code $P$ in groups of $M$ bits to compute $(X,Y)$ which is the location of the upper leftmost pixel (assuming that the origin is in the upper leftmost corner of the image) in the block corresponding to $P$. CON is a table indicating a mapping from dilated integers to the actual integer values of numbers ranging from 0 to $2^M - 1$. The $y$ coordinate is more significant than the $x$ coordinate. Assume that $2^N$ is divisible by $M$, and that $M$ is divisible by 2. */
begin
value fd locational code $P$;
value integer $N,M$;
reference integer $X,Y$;
global integer array CON[0:(2$^M$ − 1)/3];
integer I,MASK;
MASK ← (2$^M$ − 1)/3; /* Calculate mask to extract dilated integer */
X ← 0;
Y ← 0;
for I ← 0 step 1 until (2$^N$/M) − 1 do
begin
X ← X lor ((CON[P land MASK]) lsh M/2);
Y ← Y lor ((CON[(P lsh −1) land MASK]) lsh M/2);
P ← P lsh −M; /* next subgroup of $M$ bits of $P$ */
end;
end;xercises

3.1: Suppose that an FD locational code is being decoded by use of table lookup techniques with subgroups of bits. Show that if the decoding process uses the same number of subgroups as the encoding process, then the number of bits in each subgroup is twice the number of bits in the encoding process.

3.2: Given a $2^n \times 2^n$ image, show that the maximum dilated integer value of a pixel coordinate is $(4^n − 1)/3$. 

3-4
3.3: Modify procedure \textsc{encode} to deal with the case that \( M \) is not a divisor of \( N \).

3.4: Modify procedure \textsc{decode} to deal with the case that \( M \) is not a divisor of \( 2^N \).

3.5: Generalize procedure \textsc{encode} to deal images of arbitrary dimension (say \( D \)).

3.6: Generalize procedure \textsc{decode} to deal images of arbitrary dimension (say \( D \)).

3.7: [Donald E. Knuth] How would you add two dilated integers, say \( a \) and \( b \), in two-dimensions.
CHAPTER 4

NEIGHBOR FINDING TECHNIQUES

4.1.

4.2.

4.3. NEIGHBOR FINDING IN POINTER-LESS REPRESENTATIONS

In Chapter 3 we presented a number of pointer-less quadtree (octree) representations. Although their use may lead to significant savings in space, they vary in the ease with which they can be manipulated. In particular, as we discuss below, performing neighbor finding requires quite a bit of work with some of them, although it is possible. Recall that pointer-less quadtree (octree) representations can be grouped into two categories. The first treats the image as a collection of leaf nodes while the second represents the image in the form of a traversal of the nodes of its tree.

In this section, we concentrate on the locational code because it is the most efficient. We give algorithms for finding neighbors in all directions for the general case of a \( d \)-dimensional image and for a quadtree that are represented by the FD locational code. We also show how to do neighbor finding when only the FD locational codes of the black nodes are stored. Next, we briefly discuss the use of FL and VL linear quadtrees. We conclude with an equally brief mention of the use of DF-expressions. Recall that the latter is a traversal of the nodes of the tree. The sections dealing with FL and VL linear quadtrees and DF-expressions are very specialized. They are included primarily for completeness and can be skipped.

4.3.1. FD LINEAR QUADTREE

When an image is represented as a collection of the leaf nodes comprising it, there are a number of methods of representing the individual leaf nodes. In the FD linear quadtree (octree), each leaf node (black and white) is represented by its FD locational code. Assuming an image of side length \( 2^n \), there are two equivalent ways to define the FD locational code of each leaf node of side length \( 2^k \). The first technique treats the FD locational code as being \( n \) digits long, where the leading \( n-k \) digits contain the directional codes that locate the leaf along a path from the root of the tree. The \( k \) trailing digits contain the directional codes (usually 0) that locate the pixel in the block’s NW (LDB) corner. The second technique treats the FD locational code as having \( 2n \) (3\( n \)) bits for a quadtree (octree) and is obtained by interleaving the bits that comprise the values of the \( x \) and \( y \) (and \( z \)) coordinates of the pixel (voxel) in the block’s NW (LDB) corner. In the rest of this section, we assume the bit interleaving definition.
The directional codes are numeric equivalents of the different quadrants (i.e., 0, 1, 2, 3 for NW, NE, SW, SE, respectively) and octants (i.e., 0, 1, 2, 3, 4, 5, 6, 7 for LDB, LDF, LUB, LUF, RDF, RDB, RUF, RUF, respectively). We assume that the origin is in the NW (LDB) corner of the block. Therefore, for a block of side length $2^k$, the directional codes stored in the $k$ trailing digits are 0. When the FD locational code is viewed as the result of bit interleaving, a block of side length $2^k$ is represented by its pixel (voxel) for which bit interleaving yields the minimum value in the block.

A node’s FD locational code is implemented by a record of type `fd_locationalcode` with 3 fields `CODE`, `LEV`, and `COL` corresponding to the minimum bit-interleaved value in the node’s block, the level of the node, and the color of the node, respectively. For an image of maximum side length $2^k$, the code, say $P$, is implemented as an integer.

Given a node $A$ with FD locational code $P$, finding its neighbor in direction $t$ of size greater than or equal to $A$ is conceptually simple. During this process, we first compute $T$, the code component of the FD locational code of the equal-sized desired neighbor, say $Q$. The result is analogous to calculating the address of the pixel (voxel) in the desired neighboring block with the minimum bit-interleaved value, except that it is more complex since we are not dealing with the individual values of the $x$, $y$ (and $z$) coordinates of the address. Instead, we are dealing with the interleaved binary representations of the individual coordinate values.

The address of a neighbor of equal size can be calculated by adding or subtracting the size of the block to the individual coordinate values, depending on the direction of the desired neighbor. Since the block is represented by its FD locational code, this process is not so easy. In particular, it would seem that we need to unpack the individual coordinate values, increment or decrement them as necessary, and then repack them to form the new interleaved value. This process makes use of procedures `DECODE` and `ENCODE` described in Section 2.1.1.

Tocher [Toch54] describes a method of performing the addition and subtraction that does not require unpacking the individual coordinate values of the FD locational code. This method is applied to neighbor finding by Schrack [Schr92]. It takes advantage of the observation that if we only want to increment the value of the $x$ coordinate component of the FD locational code, then we can use normal addition by preceding it with a logical or (i.e., lor) operation that loads 1s in every bit position that does not correspond to the $x$ coordinate. This will propagate the carry bit between the bit positions that correspond to the $x$ coordinate, and will skip over the bit positions corresponding to the remaining coordinates. Of course, we must also save the values of the bit positions that do not correspond to the $x$ coordinate prior to the addition, and restore them after the addition.

The exact sequence of operations is as follows. For the moment, let us assume that we are only incrementing the value of the $x$ coordinate of the code component, say $C$, of FD locational code $P$, and that our increment has the value 1. This means that we are only considering neighbors in the E direction in the case of a quadtree and in the R direction in the case of an octree. Let $M_x$ be a mask that has a 1 in every bit position that correspond to the $x$ coordinate. Similarly, let $M_y$ be a mask that has a 1 in every bit position that does not correspond to the $x$ coordinate (for an alternative, see Exercise 3.4.1.3).
Save in $S$ the values of the bit positions that do not correspond to the $x$ coordinate by applying a logical and (i.e., land) operation to $C$ and $M_x$.

Load a 1 in every bit position of $C$ that does not correspond to the $x$ coordinate by applying a logical or (i.e., lor) operation to $C$ and $M_x$.

Add 1 to $C$.

Load a 0 in every bit position of $C$ that does not correspond to the $x$ coordinate by applying a logical and (i.e., land) operation to $C$ and $M_x$.

Reset the values of the bit positions of $C$ that do not correspond to the $x$ coordinate to their values at the start of step (1) by applying a logical or (i.e., lor) operation to $C$ and $S$.

$C$ will now reflect the $x$ coordinate value of the neighbor in the E direction in the case of a quadtree and in the R direction in the case of an octree. In order to increment the other coordinate values (still assuming that the value of the increment is 1), we simply reapply steps (1)-(5) to each coordinate value with the appropriate masks. Note that the increment in step (3) differs depending on the coordinate. In particular, the increment is really $2^i$ for the coordinate that corresponds to the $i^{th}$ least significant bit ($i=0$ for the least significant coordinate which is $x$ in our example).

Now, if we want to find neighbors in more than one direction (i.e., the value of several coordinates will change), then we simply apply steps (1)-(5) as many times as we have directional changes. Moreover, at each successive application of steps (1)-(5), we use the value of $C$ from the previous application. Thus at the end of this process, $C$ has the desired value. Note that this process is equivalent to incrementing the individual coordinates separately and then applying a logical or (i.e., lor) operation to all of them.

So far, we have only shown how to perform addition on the FD locational code. Moreover, the value of our increment was restricted to 1. For a quadtree, assuming that our origin is in the upper leftmost corner of each block, this means that we can compute pixel-size neighbors of pixels in the E, S, and SE directions. Similarly, for an octree, assuming that the origin is in the LDB corner, we can compute voxel-size neighbors of voxels in the R, U, F, RU, UF, RF, and RUF directions. However, we need to be able to compute neighbors of greater than or equal size for nodes of all sizes (not just pixels and voxels), and in all directions. We will now show how to remove these restrictions in a step-by-step manner.

Computing a neighbor in an arbitrary direction requires that we be able to perform subtraction. Subtraction is quite straightforward once we realize that the addition process described in steps (1)-(5) will work provided that negative numbers are represented using twos complement notation. Of course, when using subtraction, the increment in step (3) is negated. However, when combining several directions (e.g., NE) we must be careful to negate only the bit positions corresponding to the negative direction (but see Exercise 3.4.1.9).

For example, suppose that we wish to compute the NE neighbor of a pixel in a 16×16 image at position (11,10) with FD locational code 11001110 (assuming that $x$ is the least significant coordinate). The E neighbor is represented by an increment of 1, while the N neighbor is represented by an increment of -2. The resulting combined increment has a value of $(-1,1)$ which has a binary representation of 10101011. It is not the same as combining increments of 1 and -2 in step (3) to yield -1. Applying step (1)-(5) as many times as we have directional changes (i.e., twice) yields 11010010 which is position (12,9) as expected.
In order to facilitate the computation of the increment, we make use of the function $INC$ to contain the values of the increments for all possible directions. Table D.3 contains the definition of $INC$ for a $2^d \times 2^d$ two-dimensional image. In general, for a $d$-dimensional image, $INC$ has $3^d - 1$ different values (see Exercise 3.4.1.12).

<table>
<thead>
<tr>
<th>Table D.3. $INC(I)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
</tr>
<tr>
<td>INC(I)</td>
</tr>
</tbody>
</table>

Now, let us turn our attention to computing equal-size neighbors for nodes of arbitrary size (i.e., side length $2^j$ where $j > 0$). This is quite easy and we once again use steps (1)-(5). The only difference is that, given an image of dimension $d$ (e.g., $d = 2$ for a quadtree and $d = 3$ for an octree), the increment in step (3) is now $2^d j^+1$ or $-2^d j^+1$ for the coordinate that corresponds to the $i^{th}$ least significant bit (recall that in our example the $x$ coordinate corresponds to the least significant bit - i.e., $i = 0$). The negative increment is used when the direction is negative (i.e., N and W in the case of a quadtree, and L, D, and B in the case of an octree). The increment is obtained by applying the function $INC$ and then multiplying the result by $2^d j$ or simply by shifting the result to the left by $d \cdot j$ bits.

Once the code component, say $C$, of the FD locational code of the neighbor of equal size (of the node with FD locational code $P$) has been computed, we must determine if it actually exists, as well as its color. Assume, without loss of generality, that the collection of FD locational codes, say $L$, is implemented as a list of records of type $fd\_locational\_code$. Search $L$ for the FD locational code with a code that has the maximum value that is still less than or equal to that of $C$, say $R$ - i.e., $\text{CODE}(R) \leq C$. If the level of $R$ is greater than or equal to that associated with $C$ (i.e., $\text{LEV}(P)$), then $R$'s node contains the node represented by $C$, and $R$ is returned as the FD locational code of the neighbor. For example, in Figure the path to the northern neighbor of node 7 (with FD locational code 210) is 032. However, in the list of FD locational codes for this image, the nearest FD locational code is 000, which corresponds to node 1 which is larger than node 7.

Otherwise, there is no leaf node in the tree that corresponds to the desired neighbor, and the neighbor is gray (see Exercise 3.4.1.1). In this case, we return an FD locational code for a gray node at level $\text{LEV}(P)$ with $C$ as the path from the root to it. For example, in Figure the code for the equal-size eastern neighbor of node 13 (with FD locational code 300) is 210. 210 is present in the list of FD locational codes for this image and it corresponds to node 7. However, node 7 is smaller than 13 (i.e., its level is smaller) and thus we return its locational code but mark it as gray and set the LEV field to the same value as that of 13.

When the node whose neighbor is sought is along the border of the space represented by the image, then, depending on the direction, the neighbor may possibly not exist (e.g., the E neighbor of node 5 in Figure 3.1). In such a case, our procedure will perform a wraparound in the sense that it will return 1 as the

\[1\text{Recall that we use the convention that for a } 2^n \times 2^n \text{ image the root is at level } n \text{ while a pixel is at level } 0.\]
neighbor. This is because we are performing arithmetic and we have relied on
overflow being ignored. This is pretty standard especially when using a twos com-
plement representation for negative numbers. If this is not desired, then we must
check for overflow after each iteration of step (3) (see Exercise 3.4.1.10). In par-
cular, if the direction is positive, then overflow implies wraparound while if the
direction is negative, then the absence of an overflow implies wraparound.

Given a \( 2^n \times 2^n \) image whose quadtree contains \( m \) leaf nodes that are kept
in a list sorted by their FD locational codes, the neighbor computation process
described above has a worst-case execution time of \( O(\log_2 m + n) \). This assumes
that the addition of two \( n \) bit numbers takes \( O(n) \) time. However, from a practi-
cal standpoint we can say that it is constant since the word size is fixed and we
assume that addition is a unit operation. Under such an assumption, the worst-case
execution time is \( O(\log_2 m) \).

Now let us look at the general case. Procedure \textsc{gen fd neighbor}, given
below, is used to calculate the FD locational code of the neighbor of a node, say \( A \),
with FD locational code \( p \), in all directions in a \( d \)-dimensional FD linear quadtree.
It implements steps (1)-(5) of the algorithm described above. The masks needed
for steps (2) and (4) are represented by the mask array \textsc{mask2} and the variable
\textsc{mask4}, respectively. They do not need to be recomputed each time procedure
\textsc{gen fd neighbor} is invoked. Instead, global values can be used.

\textsc{mask4} corresponds to \( M_x \) - i.e., it has a 1 in every bit position that
corresponds to the \( x \) coordinate. It is computed by observing that it is the sum of
the powers of \( 2^d \) - i.e., \( 2^d \) where \( i \) ranges from 0 to \( n-1 \). This sum has the value
\( (2^{dn}-1)/(2^d-1) \). The remaining masks (i.e., \( M_y, M_z, \cdots \)) are obtained by shifting
\( M_x \) to the left by a number equal to the coordinates bit position relative to that of
the \( x \) coordinate. Mask \textsc{mask2}[i] corresponds to the values of \( M_{x_i} \) for coordinate
\( i \). It is computed by complementing (i.e., \text{not} which is a logical not operation) the
value of \textsc{mask4} and then applying a left circular shift of \( i \) bits.

The increment in step (3) is computed by shifting the result of the applica-
tion of function \textsc{inc} to the left by a number of bits equal to the size of the node
whose neighbor is being sought, i.e., \( A \). Function \textsc{inc} was given in Table D.3 for
a \( 2^2 \times 2^2 \) two-dimensional image. In general, for a \( d \)-dimensional image, \textsc{inc} has
\( 3^d-1 \) different values. \textsc{inc} can be precomputed for a \( d \)-dimensional image by
computing its values for the \( 2^d \) different \( (d-1) \)-dimensional adjacencies compris-
ing it. For example, in two (three) dimensions these are \( N, S, E, \) and \( W \) (\( L, R, D, U, \)
\( B \), and \( F \)). These adjacencies can be considered as pairs of principal directions
where one corresponds to a unit step in the positive direction while the other
corresponds to a unit step in the negative direction.

Of course, these directions are really the same as the coordinates. For
example, assuming that the direction of the \( x \) coordinate is the least significant, we
have that a unit step in the positive \( x \) direction has an \textsc{inc} value of 1 while a step
in the negative \( x \) direction has an \textsc{inc} value of \( \Sigma_{i=0}^{n-1} 2^i \), which is the same as
\textsc{mask4}. Assigning an order for these directions means that for an image of side
length \( 2^n \), positive direction \( i \) (\( 0 \leq i < d \)) has an \text{inc} entry of \( 2^i \) while negative direc-
tion \( i \) has an \text{inc} entry of \( 2^i \cdot \text{mask4} \). The \text{inc} values of combinations of the principal
directions (e.g., NW, LDB, etc.) are obtained by applying logical or (i.e., \text{lor}) or
addition operations to the \text{inc} values of the component principal directions.

4-5
fd_locationalcode procedure GEN_FD_NEIGHBOR(D,N,L,I,P);
/* Given a D-dimensional quadtree of a side of width 2^N
   represented by a linear quadtree in the form of a list L, of the FD
   locational codes of its nodes, return the FD locational code of the
   neighbor in direction I of a node with FD locational code P. If no
   neighbor exists, then NIL is returned. Assume N>0. */
begin
value integer D,N;
value pointer fd_list L;
value direction I;
value pointer fd_locationalcode P;
pointer fd_locationalcode B,Q;
integer C,J,MASK4;
integer array MASK2[0:D-1];
if LEV(P) ≥ N then return(NIL); /* No neighbor exists in direction I */
MASK4 ← (2↑(D*N) -1)/(2↑D-1); /* Calculate mask for step (4) */
MASK2[0] ← lnot(MASK4); /* Take 1's complement */
for J←1 step 1 until D-1 do /* Calculate masks for step (2) */
    MASK2[J] ← MASK2[J-1] lsh 1; /* Left circular shift by one bit */
C ← 0;
for J←0 step 1 until D-1 do
    C ← C lor (((CODE(P) lor MASK2[J]) + ((INC(I) lsh LEV(P)*D) land (MASK4 lsh J)))
    !land (MASK4 lsh J));
B ← MAXLEQ(C,L); /* Find maximum FD locational code in L that is ≤C */
if LEV(B)≥LEV(P) then return(B)
else /* The neighbor is smaller; create a new gray node */
    begin
        Q ← create(fd_locationalcode);
        CODE(Q) ← C;
        LEV(Q) ← LEV(P);
        COL(Q) ← ‘GRAY’;
        return(Q);
    end;
end;

Procedure GEN_FD_NEIGHBOR can be streamlined for the case when d=2.
In particular, there is no need for the array MASK2 since in the case of the x
coordinate, M_x is twice the value of MASK4 and in the case of the y coordinate, M_y
is MASK4. There is also no need for a loop running through the two dimensions
(indexed by J in GEN_FD_NEIGHBOR). The result is given below by procedure QT_.
FD_NEIGHBOR.

fd_locationalcode procedure QT_FD_NEIGHBOR(N,L,I,P);
/* Given a 2^N×2^N image represented by a linear
   quadtree in the form of a list L, of the FD locational codes of its
   nodes, return the FD locational code of the neighbor in direction I of a
   node with FD locational code P. If no neighbor exists, then NIL is
   returned. Assume N>0. */
begin
value integer N;
value pointer fd_list L;
value direction I;
value pointer fd_locationalcode P;
pointer fd_locationalcode B,Q;
integer C,MASK4;
if LEV(P) ≥ N then return(NIL); /* No neighbor exists in direction I */
MASK4 ← (4↑N -1)/3; /* Calculate mask for step (4) */
C ← (((CODE(P) lor (MASK4 lsh 1)) + ((INC(I) lsh LEV(P)*2) land MASK4)) land MASK4) lor
    !((CODE(P) lor (MASK4 +2) land MASK4+2));
B ← MAXLEQ(C,L); /* Find maximum FD locational code in L that is ≤C */
if LEV(B)≥LEV(P) then return(B)

4-6
else /* The neighbor is smaller; create a new gray node */
begin
Q ← create(fd_locationalcode);
CODE(Q) ← C;
LEV(Q) ← LEV(P);
COL(Q) ← ‘GRAY’;
return(Q);
end;
end

Suppose that we only store the FD locational codes of the black nodes (as proposed by Gargantini [Garg82a]). In such a case, determining the neighbor's color and size is more complex. Assume an image of side length $2^n$. Moreover, for the purpose of the following explanation, let us treat the value of the CODE field, say $F$, as if it is an array of type $fd_{-path}$ of $n$ directional codes stored in the order $F[n-1]F[n-2]...F[1]F[0]$. To find the neighbor of node $A$, with FD locational code $P$, of size greater than or equal to $A$, we first construct the code component of the equal-sized neighbor, say $Q$, with level $k = LEV(P)$.

Assume that an equal-sized neighbor exists (i.e., we are not at the border of the image), and let $C$ denote its code component. Now, search the list of FD locational codes $L$, using binary search, for $B$ and $H$, the FD locational codes whose code components have the maximum value that is still less than or equal to $C$ and the minimum value that is still greater than $C$, respectively - that is, $CODE(B) \leq C$ and $C < CODE(H)$.

If $CODE(B) = C$ and $LEV(B) = k$, then $B$ is the FD locational code of the neighbor that is black and of equal size. If $LEV(B) > k$ and $C[j] = CODE(B)[j]$ for $LEV(B) \leq j < n$, then $C$ is a descendant of $CODE(B)$ and hence $B$ is the FD locational code of the neighbor that is black and of side length $2^{LEV(B)}$. If $C[j] = CODE(H)[j]$ for $k \leq j < n$, then $CODE(H)$ is a descendant of $C$ and hence $C$ is the code component of the neighbor that is gray and of equal size.

Otherwise, the neighbor is white and we must determine its FD locational code and its size. To do this, we must examine both $CODE(B)$ and $CODE(H)$ to see which one has the largest number of consecutive leading digits that match those of $C$ (see Exercise 3.4.1.15). Without loss of generality, suppose that it is $B$ and that the first digit that mismatches is at position $j$, i.e., $CODE(B)[j]$. Although not immediately obvious, this means that the neighbor is white and its side length is $2^j$ (see Exercise 3.4.1.16). The CODE field of its FD locational code is $C[i]$ for the digits in position $i$ ($j \leq i < n$), and the remaining digits contain a 0 directional code. For example, in Figure the western neighbor of node 4, with FD locational code 120, is white node 1, with FD locational code 000.

Procedure GEN_BLACK_FD_NEIGHBOR, given below, is used to calculate the FD locational code of the neighbor of a node, say $A$ with FD locational code $P$, in an FD linear quadtree where only the black nodes are represented. It uses procedures MAXLEQ and MINGT (not given here) to search the list of FD locational codes that comprises the linear quadtree to find the FD locational codes of the nearest nodes to $A$’s equal-sized neighbor.

fd_locationalcode procedure GEN_BLACK_FD_NEIGHBOR(D,N,L,I,P);
/* Given a D-dimensional quadtree of a side of width $2^N$
represented by a linear quadtree in the form of a list L, of the FD
locational codes of its BLACK nodes, return the FD locational code of
the neighbor in direction I of a node with FD locational code P. If no
neighbor exists, then NIL is returned. Assume $N > 0$. */

begin
value integer D,N;
value pointer fd_list L;
value direction I;
value pointer fd_locationalcode P;
pointer fd_locationalcode B,H,Q;
integer C,J,K,M,MASK4;
integer array MASK2[0:D−1];
if $\text{LEV}(P) \geq N$ then return(NIL); /* No neighbor exists in direction $I$ */
$\text{MASK4} \leftarrow (2^{(D+N)} - 1)/(2^{D − 1});$ /* Calculate mask for step (4) */
$\text{MASK2}[0] \leftarrow \lnot(\text{MASK4});$ /* Take $I$’s complement */
for $J=1$ step 1 until $D−1$ do /* Calculate masks for step (2) */
$\text{MASK2}[J] \leftarrow (\text{MASK2}[J−1] \landsh 1; /* Left circular shift by one bit */
$C \leftarrow 0;
for J=0$ step 1 until $D−1$ do
$C \leftarrow C$ lor $((\text{CODE}(P)$ lor $\text{MASK2}[J]) + ((\text{INC}(I)$ lsh $\text{LEV}(P)$ lor $D)$ land $(\text{MASK4}$ lsh $J)))$
$\text{lop}(\text{MASK4}$ lsh $J));$
$\text{B} \leftarrow \text{MAXLEQ}($C,L); /* Find maximum FD locational code in $L$ that is $\leq C$ */
$\text{H} \leftarrow \text{MINGT}($C,L); /* Find minimum FD locational code in $L$ that is $> C$ */
$M \leftarrow N−1;$
if not(null($B$)) then
begin /* The neighbor is not before the first FD locational code in $L$ */
while $M \geq \text{LEV}($B$)$ and $((\text{CODE}($B$)$ xor $C)$ lsh $−D∗M)=0$ do $M \leftarrow M−1;$
/* The xor combined with the right shift is the same as $\text{CODE}($B$)[M]=\text{C}[M]$ */
if $M<\text{LEV}($B$)$ then
return($B$); /* Neighbor is black and of greater than or equal size */
end;
$Q \leftarrow \text{create(fd_locationalcode)};
J \leftarrow N=1;
if not(null($H$)) then
begin /* The neighbor is not after the last FD locational code in $L$ */
while $((\text{CODE}($H$)$ xor $C)$ lsh $−D∗J)=0$ do $J \leftarrow J−1;$
/* The xor combined with the right shift is the same as $\text{CODE}($H$)[J]=\text{C}[J]$ */
if $J<\text{LEV}($P$)$ then
begin /* $H$ is a descendant of $C$ */
$\text{CODE}($Q$) \leftarrow C;
\text{LEV}($Q$) \leftarrow \text{LEV}($P$);
\text{COL}($Q$) \leftarrow ‘GRAY’;
return($Q$);
end;
end;
$C \leftarrow C$ land $(\lnot(2^{\min(M,J)∗D−1}))$;
/* Equivalent to zeroing the $\min(M,J)$−1 least significant directional digits */
$\text{CODE}($Q$) \leftarrow C;
\text{LEV}($Q$) \leftarrow \min(M,J);
\text{COL}($Q$) \leftarrow ‘WHITE’;
return($Q$);
end;
end;

Exercises

4.1: After having calculated the FD locational code of an equal-sized neighbor, say $Q$ with code $C$, procedures GEN_FD_NEIGHBOR and its quadtree counterpart determine $Q$’s real size by searching the list of FD locational codes for the FD locational code whose code component has the maximum value still less than or equal to that of $C$, say $R$ - i.e., $\text{CODE}($R$)\leq C$. The levels of the code components are compared to determine the size of the neighbor. Prove that this technique works correctly.

4.2: Suppose that the FD locational code is implemented in such a way that the trailing $k$ digits of the code component of a node at level $k$ are 3 instead of 0.
How does this affect the algorithm for computing the neighbors of greater than or equal size?

4.3: [Donald E. Knuth] The following is an alternative to steps (1)-(5) which serve as the basis of procedure GEN.FD.NEIGHBOR.

1. Save $c$ in $s$.
2. Load a 1 in every bit position of $c$ that does not correspond to the $x$ coordinate by applying a logical or (i.e., lor) operation to $c$ and $M_o$.
3. Add 1 to $c$.
4. Apply an exclusive or (i.e., xor) operation to $c$ with $s$.
5. Load a 0 in every bit position of $c$ that does not correspond to the $x$ coordinate by applying a logical and (i.e., land) operation to $c$ and $M_x$.
6. Apply an exclusive or (i.e., xor) operation to $c$ with $s$.

Show that the results are equivalent.

4.4: Is the method of Exercise 4.3 better or worse than our original method?

4.5: The procedures described in this section made use of parameter transmittal by ‘value.’ Can you make them more efficient by transmitting parameters by ‘reference’?

4.6: What features must a programming language provide so that the algorithms discussed in this section can be implemented efficiently? Compare an implementation in ADA, C, PASCAL, and MODULA 2. Which one yields the most efficient encoding?

4.7: Rewrite procedure GEN.FD.NEIGHBOR so that it runs as fast as possible.

4.8: Adapt procedure GEN.FD.NEIGHBOR to deal with octrees.

4.9: Suppose that you are only allowed transitions in one of the directions (i.e., only in one of $x$, $y$, or $z$). Show how to avoid step (2) for a negative transition in the direction of coordinate $i$ from a block with code $c$ of side length $2^k$.

4.10: In the case of wraparound, our proposed solution checks for the presence or absence of overflow at the conclusion of step (3). Explain why this is correct even though it would seem that for a $d$-dimensional image, we must examine the carry bit from the $d$ most significant bit position of the appropriate coordinate.

4.11: Consider a $d$-dimensional image of side width $2^n$. Suppose that you store every CODE field of an FD locational code in a word with a maximum of $c$ ($c>d.n$) bits. Prove that procedure GEN.FD.NEIGHBOR handles addition overflow correctly.

4.12: Prove that for a $d$-dimensional image, function INC has $3^d-1$ possible values.

4.13: Write a procedure to implement function INC for a $d$-dimensional image of side length $2^n$. The idea is to devise a table so that the values can be obtained by table lookup. To do this, you will need to devise a way to specify the various combinations of directions so that the appropriate table entry can be accessed.

4.14: Procedure GEN.BLACK.FD.NEIGHBOR contains a loop of the form ‘while $M2\leq\text{LEV}(B)$ and (((CODE(B) xor C) Isth–D*M)=0) do $M \leftarrow M–1;’ and another loop of the form ‘while (((CODE(H) xor C) Isth–D*I)=0 do $I \leftarrow I–1;’. Why is there no need to check for ‘$2\leq\text{LEV}(P)$’ in the second loop?

4.15: When the neighboring node is white, why does procedure GEN.BLACK.FD.-
examine both CODE(B) and CODE(H) to see which one has the largest number of consecutive leading digits that match those of C?

4.16: Continuing Exercise 4.15, suppose that the first mismatching directional digit is in position j. Why are all remaining digits after the mismatch set to 0? In other words, how do we know that the neighbor’s side length is 2^j?

4.17: The FD locational code of a neighbor of equal size can also be obtained by a process that is analogous to that used for finding neighbors in a pointer-based representation [Same89a]. Given a node A with FD locational code P, finding its neighbor in direction t of size greater than or equal to A is done as follows. During this process, we first construct C, the code component of the FD locational code of the equal-sized desired neighbor, say Q. Starting with the digit position in the code corresponding to the link from A to its father (i.e., CODE(P)[LEV(P)]), reflect each directional code in the designated direction (and assign it to the corresponding entry in C) until encountering the nearest common ancestor of A and Q. The nearest common ancestor is detected by using the function ADJ given previously in Table Note that unlike the algorithm for computing neighbors in a pointer-based quadtree (octree), there is no need to descend the tree since reflection occurs while ascending the tree. Once the FD locational code of the equal-sized neighbor has been computed, the rest of the process is identical to that used in GEN.FD.NEIGHBOR. Write a procedure FD.QT.GTEQ.NEIGHBOR to implement this technique. To facilitate the expression of the algorithm, implement the CODE field of the fd_locationalcode record as an array of type fd_path of n directional codes stored in the order P[n-1]P[n-2]...P[1]P[0].

4.18: Rewrite procedure FD.QT.GTEQ.NEIGHBOR in Exercise 4.17 so that the field CODE is accessed via bit manipulation instructions instead of being treated as an array.

4.19: In Section 2.1.4 we discussed the two-dimensional run encoding (2DRE) [Lauz84]. It first constructs a linear quadtree in which each of the black and white leaf nodes is encoded with the Morton Matrix number of the pixel that occupies its lower right corner - i.e., the pixel having the maximum Morton Matrix number in the leaf node. It assumes that the origin is at the upper left corner of the universe. This is a variant of the FD locational code, which only uses the path from the root to the node and does not require recording the level of the node. The list of leaf nodes is sorted by their corresponding Morton Matrix numbers. Show how to compute edge and vertex-neighbors of greater than or equal size using such a representation. The result should also indicate the color of the neighbor.

4.20: Suppose that you have a list of the Morton Matrix numbers of a quadtree as described in Exercise 4.19. The 2DRE is formed by viewing this list as a set of subsequences of Morton Matrix numbers corresponding to nodes of the same color and discarding all but the last element of each subsequence of blocks of the same color. Show how to compute edge and vertex-neighbors of greater than or equal size when a quadtree is represented by the 2DRE method. The result should also indicate the color of the neighbor.

4.21: Show how to compute face, edge, and vertex-neighbors of greater than or equal size of a face, an edge, and a vertex when an octree is represented by a list of the Morton Matrix numbers of its leaf nodes. The result should also indicate the color of the neighbor.

4.22: Consider the adaptation of 2DRE discussed in Exercise 4.20 scheme to an octree (the result is a three-dimensional run encoding (3DRE)). Show how to compute face, edge, and vertex-neighbors of greater than or equal size when an octree is represented by the 3DRE method. The result should also indicate the color of the neighbor.
4.3.2. FL LINEAR QUADTREE

The FL locational code is similar to the FD locational code. It is also a fixed length code. The difference is that the FD locational code requires an additional field to record the level of a node, while the FL locational code has an additional possible value for the directional code, which is a don’t care. Assuming an image of side length $2^n$, the FL locational code of each leaf node of side length $2^k$ is $n$ digits long where the $k$ trailing digits contain a don’t care directional code, and the leading $n-k$ digits contain the directional codes that locate the leaf along a path from the root of the tree.

The FL locational code of a quadtree (octree) node is a base 5 (9) number. However, all arithmetic operations on the FL locational code are performed using base 4 (8). The directional codes are numeric equivalents of the different quadrants and octants (i.e., 0, 1, 2, and 3 for NW, NE, SW, and SE, respectively, for a quadtree) and 4 denotes a don’t care. The FL locational code can also be defined by representing the don’t care directional code by 0 and the remaining directional codes by 1-4 for a quadtree and 1-8 for an octree (see Exercise 3.4.2.3).

Each leaf node in the FL linear quadtree (octree) is represented by its FL locational code. In the version that we describe in this section, we only store the FL locational codes of the black nodes, as proposed by Gargantini [Garg82a]. The FL locational codes of the remaining white (as well as gray) nodes can be inferred from the locational codes of the black leaf nodes.

Finding the neighbor of a node with locational code of size greater than or equal to $A$ is achieved in a manner similar to that used for an FD locational code (recall procedure GEN.BLACK.FD.NEIGHBOR in Section 4.3.1). Once again we first find a neighbor of equal size. However, we cannot use addition or subtraction as in the case of the FD locational code due to the presence of the don’t care digits. Thus we must use an analog of the procedure used for a pointer-based tree representation (see Exercise 4.17).

The key difference is that there is never a need to traverse links since the only operation is one of bit manipulation. This difference is of no consequence when the tree is represented in internal memory. However, when storage requirements are such that the tree is represented in external memory (e.g., using a B-tree as in [Same84d, Same87a, Shaf89a]), then this difference is very important. It means that there is no need to traverse links between nodes on different pages, a situation that could cause a page fault. In such a situation, bit manipulation will, in most cases, be considerably faster than link traversal.

Once the FL locational code of the neighbor of equal size has been found, search the list of FL locational codes $L$, using binary search, for $B$ and $H$, the FL locational codes whose code components have the maximum value that is still less than $C$ and the minimum value that is still greater than or equal to $C$, respectively - that is, $\text{CODE}(B) < C$ and $C \leq \text{CODE}(H)$. This procedure is analogous to that used in GEN.BLACK.FD.NEIGHBOR for FD locational codes (see Exercise 3.4.2.2). The only slight difference is that there $\text{CODE}(B) \leq C$ and $C < \text{CODE}(H)$ (see Exercise 3.4.2.3).
Decoding the individual directional codes of a particular FL locational code is a complicated process, which requires use of division operations. Unfortunately, since the directional codes are represented by base 5 (9) numbers, we cannot replace division operations by shifts unless we are willing to waste much of one bit for each digit by using 3 (4) bits to represent each of the directional codes. For this reason, the FD locational code is more popular. Also, most implementations retain the locational codes of all of the leaf nodes (not just the black ones).

Exercises

4.23: When looking for the neighbor of equal size you need to know the size of the current node. How would you do this?

4.24: Explain the details of how to find the neighbor of greater than or equal size in an FL linear quadtree once the FL locational code, say C, of the equal-sized neighbor at level k has been found. Assume that B and H are the FL locational codes whose codes have the maximum value that is still less than C and the minimum value that is still greater than or equal to C, respectively.

4.25: Once the FL locational code of an equal-sized neighbor has been found, say C, the list of FL locational codes is searched for the FL locational codes B and H such that CODE(B)<C and CODE(H)>C. On the other hand, in the case of an FD locational code, the search is for the FD locational codes B and H such that CODE(B)>C and CODE(H)<C. Explain the reason for the difference.

4.26: Write a procedure FL_ALL_QT_GTEQ_NEIGHBOR analogous to FD_QT_GTEQ_NEIGHBOR in Exercise 4.17 that computes a neighbor of greater than or equal size in an FL linear quadtree that contains the FL locational codes of all the leaf nodes (i.e., both black and white).

4.27: Write a procedure FL_QT_GTEQ_NEIGHBOR analogous to FD_QT_GTEQ_NEIGHBOR in Exercise 4.17 that computes a neighbor of greater than or equal size in an FL linear quadtree that only contains the FL locational codes of the black nodes.

4.3.3. VL LINEAR QUADTREE

The VL locational code is similar to the FL locational code with the principal difference being that it is based on using a variable-length locational code. It is obtained by reversing the FL locational code and letting 0 represent the don’t care directional code, while the values 1, 2, 3, and 4 represent NW, NE, SW, and SE, respectively. Thus, the leading digits correspond to the don’t care directional codes. Since the leading digits are 0, there is no reason to explicitly represent them in the VL locational code. This is why we have a variable-length code. In fact, this is its advantage over the FL locational code - i.e., the code components are shorter. Assuming an image of side length 2^n, the VL locational code of each leaf node of side length 2^k is just n–k digits long. For example, the VL locational code of node 4 in Figure is the base 5 number 23.

Each leaf node in the VL linear quadtree (octree) is represented by its VL locational code. In the version that we describe in this section, we only store the VL locational codes of the black nodes. The VL locational codes of the remaining white (as well as gray) nodes can be inferred from the locational codes of the black leaf nodes.
Finding the neighbor of a node \( A \) with VL locational code \( P \) of size greater than or equal to \( A \) is achieved in a manner similar to that used for an FD locational code (recall procedure \texttt{GEN.BLACK.FD.NEIGHBOR} in Section 4.3.1). Once again we first find a neighbor of equal size. However, we cannot use addition or subtraction as in the case of the FD locational code, in part, because of the presence of the \textit{don't care} digits, and their order. Thus we must use an analog of the procedure used for a pointer-based tree representation (see Exercise 4.17). Once again, there is never a need to traverse links since the only operation is one of bit manipulation.

Once we have calculated the VL locational code of the neighbor of equal size, the remaining process is more complex than that used with the FD locational code since when the list of VL locational codes is kept in a sorted order, the sequence corresponds to a form of a breadth-first traversal of the tree. This means that VL locational codes do not satisfy the relation that the descendants of a given FD locational code, \( Q \), immediately precede it in a sorted list. This was useful because we can determine the identity of the desired FD locational code, say \( D \), by examining the maximum FD locational code that is less than or equal to \( D \) and the minimum FD locational code that is greater than \( D \).

The VL locational code of a neighbor that is greater than or equal in size is obtained as follows. Calculate the VL locational code of the neighbor, say \( D \). If \( D \) does not correspond to a black leaf node, then we may need to search for a containing node in up to \( n \) lists - one for each level. This is in contrast to procedure \texttt{GEN.BLACK.FD.NEIGHBOR} that only requires a search of the list of FD locational codes of the entire image.

\textit{Exercises}

\textbf{4.28}: Suppose that the VL linear quadtree and octree contain the VL locational codes of all the leaf nodes (i.e., both black and white). Rewrite procedures \texttt{VL.QT.GTEQ.NEIGHBOR} and \texttt{VL.OT.GTEQ.NEIGHBOR}, and the procedures that they invoke, to cope with this definition.

\textbf{4.29}: Write a pair of procedures \texttt{VL.QT.GTEQ.NEIGHBOR} and \texttt{VL.OT.GTEQ.NEIGHBOR} to compute a neighbor of greater than or equal size in a VL linear quadtree and octree that only contain the VL locational codes of the black nodes.

\textbf{4.3.4. DF-EXPRESSIONS}

The second pointer-less representation that we discussed is one that represents the image in the form of a traversal of the nodes of its tree. This is commonly termed a DF-expression [Kawa80a] (see Section 2.2). Neighbor finding is not very convenient using such a representation since there is no notion of random access. For example, a node is only identified by its position in the traversal of the tree and is represented by a code corresponding to its type (i.e., black, white, or gray). We have no explicit information on the path from the root of the tree to it. Thus, to perform neighbor finding, it is necessary to start at the root of the tree and locate the node whose neighbor we are seeking, remember the path to it, compute the path to its neighbor, and then sequentially search the list for the neighbor starting at the root of the tree.

This is a cumbersome process and what usually happens is that algorithms that require neighbor finding are recoded in such a way that neighbor finding is
avoided. For example, Samet and Tamminen [Same85f, Same84e] present a method for computing geometric properties of images represented by pointer-less quadtree representations that does not require neighbor finding operations (see Section 5.1.3). Instead, they use a data structure termed an *active border*.

*Exercises*

**4.30:** Show how to compute face, edge, and vertex-neighbors of greater than or equal size in an image that is represented as a DF-expression. The input is an index into the list representing the DF-expression and the output should be a pointer to the element in the list corresponding to the neighbor.
SOLUTIONS TO CHAPTER 4

4.1: The comparison between the values of the code components is feasible because the trailing \( k \) digits of the code components of a node at level \( k \) are 0. Recall that the FD locational code corresponds to the coordinates of a specific pixel (or voxel) in the block (e.g., for an octree, it is the voxel in the LDB corner of the block when the origin is in the LDB corner of the image).

4.2: In procedures \textsc{fnoneighbor} and \textsc{qt widening} we would have to find the minimum FD locational code in \( L \) that is \( \geq P \) instead of the maximum FD locational code in \( L \) that is \( \leq P \). Thus, we need to replace ‘\( \max \leq (P,L) \)’ by ‘\( \min \geq (P,L) \)’.

4.3: The key is the fact that the exclusive or operation (i.e., \( \oplus \)) is commutative and associative, and that \( AA = 0 \) and \( A0 = A \). The first three steps are equivalent to the first three steps of the previous approach. Letting \( cI \) denote the result of each step in the new method, we find that after step (5) we have \( c5 \leftarrow (c3S) \land M_4 \). This means that all bit positions corresponding to the \( x \) coordinate are set to \( c3 \) while all those corresponding to the remaining coordinates are set to 0. Applying step (6) means that all bit positions corresponding to the \( x \) coordinate are set to \( (c3S) \) which is the same as \( c3 \) since \( sS = 0 \) and \( cS = c3 \). Similarly, all bit positions corresponding to the remaining coordinates are set to \( 0S \) which is \( S \).

4.4: The answer depends on the instruction set of the computer [Hen90]. There are two factors. The first is the number of operands for each instruction. Assume that each instruction has two source operands and a destination. When the three operands are distinct, we have a three-operand instruction format, while when one of the operands is both a source and a destination, we have a two-operand instruction format. The second is how many of the operands may be memory addresses. When one of the operands (usually the destination) in the two-operand format is in a general-purpose register, we have a register-memory computer [Hen90]. Each of steps (2)-(5) of the original method and steps (1)-(6) of the method of Exercise 4.3 can be executed by one instruction on a register-memory computer. Step (1) of the original method requires us to save the result in another address or register than the one operated on by steps (2)-(5). This will require two instructions on a register-memory computer, while it only needs one instruction on a computer with a three-operand instruction format. Thus the methods are equivalent on a register-memory computer, and the original method is superior on a three-operand instruction format.

4.5: You must be careful not to overwrite the locational code of the node whose neighbor is being sought.

4.6: It should have bit manipulation.

4.7: Replace the shifts of \textsc{mask4} by table lookups and precompute the value of ‘\( \text{INC}(I) \) Ish \( \text{LEV}(P) \cdot D \)’.

4.9: Assuming a \( d \)-dimensional image, such a transition can be represented by an increment value of \( -2^{d+i} \). The \( k \cdot d+1 \) least significant bits of this value are 0 and the remaining bits are 1. Treat the increment as a simple negative number and do not zero the bit positions corresponding to the other coordinates. This means that there is no need for step (2) which loads a 1 in the bit positions of \( c \) that do not correspond to coordinate \( i \).

4.10: Recall that step (2) loaded a 1 in every bit position that did not correspond to
the direction being incremented during the particular iteration of steps (1)-(5).

4.11: This is guaranteed by the fact that a logical and (i.e., land) operation is performed after incrementing each coordinate value.

4.12: In each of the $d$ directions we have three choices - a positive motion, a negative motion, and no motion. The case of no motion in all $d$ directions is impossible, hence the $3^{d-1}$ values. See also Exercise 3.3.1.2.

4.14: Because $\text{CODE(H)} > C$ and thus equality for all directional digit positions is impossible.

4.15: It means that their nearest common ancestor is closer.

4.16: Since we chose one of $B$ and $H$, say $B$, this means that $C$ must correspond to the largest possible white node consistent with the absence of any other black nodes between it and the code component of $H$. If any of these digits would be nonzero, this would contradict either the premise that $B$ and $H$ are the FD locational codes with the maximum code component value that is still less than or equal to $C$, and the minimum code component value that is still greater than $C$, respectively; or the premise that the code component of $B$ has the largest number of consecutive leading digits that match those of $C$.

4.17d locationalcode procedure FD_QT.GTEQ.NEIGHBOR(N,L,I,P);
/* Given a $2^{m} \times 2^{n}$ image represented by a linear
quadtree in the form of a list $L$, of the FD locational codes of its nodes,
return the FD locational code of the neighbor in direction $I$ of a node with
FD locational code $P$. If no neighbor exists, then NIL is returned with no
wraparound. Assume $N \geq 0$. */
begin
value integer N;
value pointer fd_list L;
value direction I;
value pointer fd_locationalcode P;
f_path array $T[0:N-1]$;
pointer fd_locationalcode B,Q;
if LEV(P)N then return(NIL); /* No neighbor exists in direction $I$ */
$T \leftarrow$ if TYPE(I)="EDGE" then

!FDL_QT_EQ.EDGE.NEIGHBOR(N,I,CODE(P),LEV(P))
|else FDL_QT_EQ.VERTEX.NEIGHBOR(N,I,CODE(P),LEV(P));
if null(T) then return(NIL) /* No neighbor exists in direction $I$ */
else B $\leftarrow$ MAXLEQ(T,L); /* Find maximum FD locational code in $L$ that is $\leq T$ */
if LEV(B)$\geq$LEV(P) then return(B)
else /* The neighbor is smaller; create a new gray node */
begin
$Q \leftarrow$ create(fd_locationalcode);
CODE(Q) $\leftarrow$ $T$;
LEV(Q) $\leftarrow$ LEV(P);
COL(Q) $\leftarrow$ ‘GRAY’;
return(Q);
end;
end;
begin
    P[LEVEL] ← REFLECT(I,P[LEVEL]);
    LEVEL ← LEVEL+1;
end
until LEVEL=N or not(ADJ(I,P[LEVEL-1]));
return(if not(ADJ(I,P[LEVEL-1])) then NIL
        else P);
end;

path array procedure FDL(QT_EQ_VERTEX_NEIGHBOR(N,I,P,LEVEL);
/* Given a 2^N image represented by an FD linear quadtree, return the path from the root to the vertex-neighbor in direction I of a node at level LEVEL whose path from the root is P. */
begin
    value integer N;
    value vertex I;
    value path array P[0:N-1];
    value integer LEVEL;
    value direction code PREVIOUS;
    do
        begin
            PREVIOUS ← P[LEVEL];
            P[LEVEL] ← REFLECT(I,PREVIOUS);
            LEVEL ← LEVEL+1;
        end
    until LEVEL=N or not(ADJ(I,PREVIOUS));
    return(if ADJ(I,PREVIOUS) then NIL
            else if COMMON_EDGE(I,PREVIOUS)≠Ω then
                FDL(QT_EQ_EDGE_NEIGHBOR(N,COMMON_EDGE(I,PREVIOUS),P,LEVEL))
            else P);
end;

4.19: You must first determine the size (i.e., level) of the node, which requires looking up the next element in the sequence.

4.23: Starting at the position of the least significant directional digit in the FL locational code of the current node, find the position of the first directional digit that does not contain a don’t care directional code, say at digit position k.

4.24: If C=CODE(H), then H is the FL locational code of the neighbor that is black and of equal size. If there exists m such that c[j]=CODE(H)[j] for k<m≤j<n, and CODE(H)[j]=DON'T CARE for 0≤j<n, then c is a descendant of CODE(H) and H is the FL locational code of the neighbor that is black and of side length 2^m. If c[j]=CODE(B)[j] for k≤j<n, then CODE(B) is a descendant of C and C is the value of the CODE field of the FL locational code of the neighbor that is gray and of equal size.

Otherwise, the neighbor is white and we must determine its FL locational code and its size. To do this, we must examine both CODE(B) and CODE(H) to see which one has the largest number of consecutive leading digits that match those of C. Without loss of generality, suppose that it is B and that the first digit that mismatches is at position j - i.e., CODE(B)[j]. This means that the neighbor is white and its side length is 2^j. The value of the CODE field of its FL locational code is C[i] for the digits in position i (j≤i<n), and the remaining digits contain a don’t care directional code.

4.25: The reason is that if a node A is contained in another node B, then the FD locational code of B is less than or equal to that of A, whereas the situation was reversed when using the FL locational code since in a quadtree (octree) a don’t care directional code is denoted by 4 (8) rather than 0 as is effectively the case for an FD locational code.